UNIVERSITAT JAUME I

Grado en Ingeniería Informática

Trabajo Final de Grado

Algoritmos de aprendizaje automatizado para el meta compresor de datos Blosc

Autor:
Alberto Sabater Morales

Supervisor:
Francesc Alted Abad
Tutores académicos:
Amelia Simó Vidal
Enrique Quintana Ortí

Fecha de lectura: 26 de junio de 2017
Curso académico 2016/2017

Resumen

Este documento recopila los detalles y procedimientos más importantes del Proyecto de Final de Grado realizado a partir de la Estancia en Prácticas en la empresa de Francesc Alted. El principal objetivo fue el desarrollo de algoritmos de aprendizaje automatizado para el compresor de datos *Blosc*.

Actualmente, el compresor *Blosc* dispone de tantas opciones a configurar para el usuario que resulta difícil conseguir la óptima para cada caso de uso y además requiere tener conocimientos sobre el mismo para utilizarlo de forma eficaz. Con el objetivo de reducir la complejidad de uso, durante la estancia en prácticas se desarrollaron algoritmos de clasificación supervisada para que escogieran las opciones óptimas por el usuario, según el resultado que éste quisiera obtener.

Además de los procedimientos seguidos en el desarrollo de estos algoritmos, también se detallan los fundamentos teóricos en los que se basan y se realiza una comparativa entre los mismos en cuanto a precisión y rendimiento. Las técnicas de clasificación supervisada que se han utilizado son: análisis discriminante, regresión multinomial, máquinas de vector soporte, vecinos más próximos y bosques aleatorios.

Palabras clave

Compresión, Aprendizaje automático, Clasificación supervisada

Keywords

Compression, Machine learning, Supervised classification.

Índice general

Capítulo 1

Introducción

1.1. Contexto y motivación del proyecto

Este Proyecto de Final de Grado, junto a la Estancia en Prácticas, forma parte del plan de formación establecido por el Grado en Ingeniería Informática. El objetivo consistió en profundizar en las técnicas de aprendizaje automatizado o clasificación supervisada y su aplicación a la mejora de un compresor de datos denominado *Blosc*.

A lo largo de los años se ha empleado la compresión para superar limitaciones físicas de la informática. Desde conseguir transmitir información más rápido cuando las conexiones a Internet no alcanzaban ni para descargar las noticias de la localidad, hasta acelerar el proceso de entrenamiento de un algoritmo de aprendizaje automático [11]. Para casos como éstos resulta fundamental el uso de librerías de compresión ultra rápidas como *Blosc*, que permiten acelerar procesos de entrada/salida que a día de hoy soportan enormes cantidades de tráfico, debido a la enorme cantidad de datos que se generan, tanto por redes sociales, como por dispositivos inteligentes conectados a Internet.

Este proyecto nace motivado por la necesidad de simplificar el uso del meta compresor de datos *Blosc*, una librería desarrollada por Francesc Alted que permite compresión y descompresión muy rápida de datos sin pérdida de información. A la hora de utilizar *Blosc* el usuario debe estar familiarizado con el mismo y las opciones que ofrece. Con el objetivo de superar esta barrera, se aplicarán diversas técnicas de aprendizaje automatizado para desarrollar un algoritmo que decida por el usuario las opciones de *Blosc* según los resultados que quiere obtener. Estos algoritmos básicamente consisten en técnicas de clasificación supervisada donde, a partir de unas variables descriptivas o características $x_i = (x_{i1}, \ldots, x_{ip})$, se infiere la variable respuesta o valor objetivo y_i. En este proyecto las características son los resultados de la compresión que

desea el usuario y las características de los datos; y las variables objetivo son las opciones de *Blosc*.

En el Capítulo 2, se detallan los procedimientos seguidos durante la estancia en prácticas, desde la generación de los datos de pruebas de compresión, hasta los análisis y comparativas de algoritmos de aprendizaje automatizado. En el Capítulo 3, se presentan los fundamentos teóricos de las técnicas utilizadas. El análisis de las técnicas y los resultados se detallan en el Capítulo 6 y por último en el Capítulo 6 se presentan las conclusiones.

Capítulo 2

Estancia en prácticas

2.1. Introducción

Francesc Alted es un consultor y desarrollador de software independiente con más de 15 años de experiencia en los lenguajes de programación *C* y *Python*, y casi 25 años de especialización en el uso de técnicas de compresión para la aceleración de los procesos de comunicación entre sistemas informáticos de entrada/salida (I/O).

Durante los últimos años ha desarrollado *Blosc*, un meta-compresor multihilo de altas prestaciones optimizado para datos binarios. *Blosc* permite comprimir conjuntos de datos con distintos compresores (códecs), filtros de precompresión y niveles de compresión, lo cual permite elegir las condiciones más óptimas ante cualquier escenario. A parte de *Blosc*, también es el autor principal de otras dos librerías, *PyTables* y *bcolz*, que complementan a *Blosc* ayudando a gestionar y estructurar grandes conjuntos de datos.

Durante la estancia en prácticas se propuso elaborar un estudio del meta-compresor de datos *Blosc* y mejorarlo por medio de sofisticadas técnicas de aprendizaje automatizado (*Machine Learning*).

Este proyecto consistió en desarrollar un algoritmo basado en técnicas de aprendizaje automatizado que permite utilizar *Blosc* de forma eficiente y sencilla. El algoritmo escoge todas las opciones disponibles al utilizar *Blosc* según los requisitos que pide el usuario. De tal forma, el usuario sólo tiene que elegir entre unas pocas opciones, según los resultados que quiere obtener, evitando tener que probar todas las opciones de *Blosc* hasta encontrar las que se adecuan a su caso.

2.2. Software utilizado

Para desarrollar el proyecto se ha utilizado el lenguaje de programación *Python*, concretamente la distribución proporcionada por Anaconda, una plataforma libre que incluye los paquetes más populares de *Python* para la ciencia de datos.

2.2.1. Blosc

Como ya se ha descrito en la introducción, *Blosc* es un meta-compresor de datos de altas prestaciones sin pérdida de información. Los algoritmos de compresión sin pérdida se caracterizan por buscar redundancias en los datos para poderlos comprimir. Esto implica que datos muy parecidos entre sí se pueden comprimir fácilmente mientras que, para datos muy diferentes, el problema es más complejo. Actualmente, para utilizar *Blosc* de forma eficiente hace falta dedicarle tiempo, tanto al aprendizaje de su funcionamiento, como para configurar las distintas opciones que ofrece. Para comprimir, *Blosc* recibe un vector de datos que pueden ser de cualquier tipo, ya que él los tratará a nivel de bytes, y comprimirá según las opciones proporcionadas. Para configurar *Blosc* hay que determinar las siguientes opciones:

- **Número de hilos**: al soportar compresión multihilo, hay que decidir cuántos hilos utilizará en la compresión. La opción por defecto es utilizar el máximo proporcionado por el procesador, pero en los experimentos realizados se ha limitado a 4 ya que es un número bastante habitual en el mercado actual.
- **Tamaño de tipo**: éste es el tamaño en bytes del tipo de datos a comprimir. Aunque se puede especificar de manera totalmente independiente, lo normal es establecer el tamaño en bytes del tipo de datos, por ejemplo: 4 bytes para números de 32 bits, 8 bytes para números de 64 bits, 24 bytes para cadenas de texto *ASCII* de 24 caracteres, etc.
- **Códec**: este es el algoritmo de compresión/descompresión a utilizar, probablemente la opción más importante a la hora de utilizar *Blosc*. Los algoritmos disponibles son: *BloscLZ*, *LZ4*, *LZ4HC*, *Snappy*, *Zstd* y *Zlib*. Cada algoritmo suele ser óptimo en distintos casos de uso por lo que conviene estudiarlos y probarlos para poder decidir.
- **Filtro**: éstos son filtros realizados sobre los datos antes de realizar la compresión. Básicamente reorganizan los bytes o bits del bloque de datos a comprimir de manera que a la izquierda se acumulan los bytes o bits más significativos de cada dato, siendo que un dato es un bloque de bytes del tamaño de tipo, y a la derecha los bytes o bits menos significativos. En la Fig. 2.1 se puede observar el esquema de esta reorganización. Este filtro es útil puesto que, por ejemplo, con enteros de 32 bits con un valor menor de 256, se acumularían 7 bytes por entero a la izquierda del bloque con todo ceros lo que incrementa

la redundancia y consecuentemente la compresión. *Blosc* proporciona tres opciones para el filtro: no utilizarlo (*noshuffle*), filtrar a nivel de byte (*shuffle*) o filtrar a nivel de bit (*bitshuffle*).

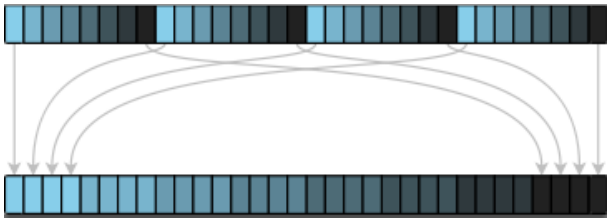


Figura 2.1: Esquema del filtro. Arriba 4 datos de 8 bytes de tamaño, abajo el bloque de datos remezclado

- **Nivel de compresión**: esta opción regula la cantidad de compresión a utilizar por el algoritmo seleccionado. Cada algoritmo ofrece distintas escalas para seleccionar la cantidad de compresión a utilizar. *Blosc* normaliza esas escalas entre 1 y 9.
- **Tamaño de bloque**: es el tamaño del bloque de datos que utiliza *Blosc* a nivel interno para realizar la compresión. Generalmente un tamaño menor implica que comprimirá más rápido puesto que el bloque a comprimir cabrá dentro de la caché del procesador. Sin embargo, el ratio de compresión se reduce porque busca redundancias en un conjunto menor de datos.

2.2.2. Git

Es el software de control de versiones utilizado para gestionar el código desarrollado. Junto al repositorio en línea `www.bitbucket.org` nos ha permitido realizar el seguimiento del proyecto. Por medio de *Git* se realizaban actualizaciones del código del programa (*commits*) que eran almacenadas en *bitbucket*. Además permite comentar el código en la web, mostrando las diferencias entre versiones de forma que establece un buen canal de comunicación sobre el código.

2.2.3. Python

Es un lenguaje de programación interpretado cuya filosofía enfatiza la legibilidad del código. Se ha escogido este lenguaje por su popularidad en el sector de la ciencia de los datos y porque la librería para *Python* ofrecida por *Blosc* es la más accesible. Este éxito se debe en parte a iniciativas como *PyData*, una comunidad de desarrolladores y usuarios de herramientas de

software libre centradas en la ciencia de los datos para *Python*, y *NumFOCUS*, una organización sin ánimo de lucro que da apoyo y promueve proyectos científicos de software libre.

A continuación se describen las librerías más importantes empleadas:

- **python-blosc**: permite utilizar el meta compresor *Blosc* desde *Python*. Se ha utilizado para realizar las pruebas de compresión con las distintas opciones disponibles en Blosc.
- **PyTables**: es una librería para la gestión de grandes cantidades de datos de forma eficiente en forma de bases de datos jerárquicas. Ha permitido el acceso a los datos utilizados para realizar las pruebas de compresión.
- **NumPy**: es la base para la computación científica en *Python*. Proporciona una potente estructura de datos de tipo matricial y capacidades del álgebra lineal, transformaciones de Fourier y números aleatorios.
- **pandas**: es la librería fundamental para el análisis de datos en *Python*. Proporciona la estructura de datos **DataFrame** que permite tratar los datos de forma eficaz con facilidad. Ha permitido almacenar, analizar y tratar los datos de las pruebas de compresión.
- **matplotlib**: es la librería por excelencia para la visualización de los datos. Permite realizar gráficas con todo lujo de detalles al estilo de *MATLAB*. Se ha empleado para visualizar los datos durante el análisis.
- **scikit-learn**: es una librería orientada al aprendizaje automatizado. Se ha empleado durante el análisis y desarrollo de los algoritmos de aprendizaje automatizado.

2.2.4. HDF5

Es un formato de ficheros diseñado para organizar y guardar grandes conjuntos de datos. Las siglas HDF significan Formato de Datos Jerárquico (*Hierarchical Data Format*). Originalmente desarrollado por el National Center of Supercomputing Applications, ahora lo mantiene The HDF Group, una organización sin ánimo de lucro con el objetivo de asegurar un desarrollo continuo en las tecnologías de *HDF5*.

Para extraer datos de las pruebas de compresión se han utilizado ficheros en este formato por lo que resulta interesante describir la estructura de un fichero *HDF5*, la cual está formada mayoritariamente por dos tipos de objetos:

- **Conjuntos de datos**: éstos son matrices de datos.
- **Grupos**: son estructuras contenedores que pueden guardar conjuntos de datos y otros grupos en su interior.

2.2.5. Jupyter Notebook

Es una aplicación web libre que permite crear y compartir documentos que contienen bloques de código con sus correspondientes salidas por pantalla y bloques de texto enriquecido con lenguaje *Markdown*, soportando *LaTex* y *HTML*. Resulta de especial utilidad a la hora de desarrollar el análisis de los datos llevando un registro de las técnicas empleadas y además permitiendo la generación de informes tanto en *HTML* como en *pdf*.

Dado que el formato de los cuadernos solo se puede visualizar mientras estás ejecutando la aplicación de *jupyter*, se decidió guardarlos también en formato *HTML* y *Python*, utilizando el código proporcionado por Jonathan Whitmore [14].

Para facilitar el desarrollo se creó una carpeta llamada *notebooks* donde guardar todo lo relacionado con los cuadernos *jupyter* y basándose en las recomendaciones de Jonathan Whitmore [15] se estableció la siguiente estructura:

- **data**: almacena todos los ficheros de datos, generalmente ficheros *csv*.
- **deliver**: contiene los cuadernos *jupyter* importantes, es decir, que están listos para entregar o presentar como resultados del análisis.
- **develop**: guarda el resto de cuadernos a modo de registro diario de los experimentos. El objetivo no es tener cuadernos limpios sino establecer un registro de la pruebas realizadas de forma que se puedan recopilar después en un cuaderno limpio.
- **src**: en caso de que el código de algún cuaderno sea muy extenso, conviene recopilarlo aquí en forma de paquetes y así mejorar la legibilidad del análisis.
- **figures**: recopila las figuras y gráficas relevantes para así poder consultar un histórico de los cambios en las imágenes.

2.3. Objetivos

El principal objetivo de este proyecto es desarrollar un algoritmo que, según las necesidades del usuario final, decida automáticamente el conjunto óptimo entre las opciones del metacompresor *Blosc*. También se pretende:

- Estudiar el comportamiento de las distintas opciones de *Blosc* sobre distintos conjuntos de datos.

- Extraer características que relacionen los datos con los resultados de la compresión.
- Automatizar el análisis de los datos para adaptar los datos de entrenamiento del algoritmo bajo demanda.
- Realizar un estudio de las distintas técnicas de aprendizaje automatizado.

2.4. Metodología y tareas realizadas

Para el desarrollo del proyecto se optó por una metodología de desarrollo de software ágil: se dedicaban unos minutos a diario para comentar el estado del proyecto y además una reunión a la semana para la refactorización del código y la planificación de la próxima semana. Dado que el cliente final es el propio Francesc, tenía una realimentación continua de calidad sobre el código.

Para la planificación del proyecto se ha optado por dividirlo en cuatro partes: la generación de datos de pruebas de compresión con Blosc, el análisis de lo datos recolectados, la generación de los datos de entrenamiento y, finalmente, la aplicación y análisis de las distintas técnicas de aprendizaje automatizado. Estas partes se comentan en detalle a continuación.

2.4.1. Generación de datos de pruebas de compresión con Blosc

El objetivo de esta tarea era el de generar datos con ficheros de datos reales para después tratarlos y prepararlos para las técnicas de aprendizaje automatizado. Para ello había que familiarizarse con *Blosc* y preparar un programa en *Python* que extrajera datos de ficheros en formato *hdf5*, después les aplicase las distintas pruebas de compresión y descompresión, y finalmente guardara los datos de las pruebas en un fichero *csv*.

Para realizar las pruebas se partió originalmente del programa *compress_ptr.py* incluido en *python-blosc* [2] y se modificó hasta desarrollar el programa final que se encuentra en el Anexo A. A continuación se detalla el flujo del programa.

En primer lugar el programa comprueba si ya existe el fichero *csv* final. De no ser así lo crea vacío con los nombres de las distintas columnas. A continuación recorre la lista de ficheros en formato *hdf5* especificada; para cada fichero recorre los distintos conjuntos de datos (que en la práctica son vectores o tablas de datos de un tipo determinado) y los divide en fragmentos de datos (en adelante, *chunks*) de 16 megabytes, de los cuales se extraerán características globales. La elección de tamaño de *chunk* de 16 megabytes se debe a que, como muestra Alistair Miles

en su estudio sobre *Blosc* [12], a partir de dicho tamaño apenas hay mejora en las velocidades de compresión y descompresión.

Una vez se van generando los *chunks*, se extrae de cada uno las características estudiadas que quedan recogidas en la Tabla 2.1. Se han seleccionado estas características puesto que sirven para determinar la dispersión, varianza o rango de los datos, propiedades que están relacionadas con la redundancia, la cual a su vez es la base de la compresión sin pérdida de información.

Característica	**Descripción**
Filename	Nombre del fichero del que proviene.
DataSet	Nombre del conjunto de datos dentro del fichero del que proviene.
Chunk_Number	Número de chunk dentro del conjunto de datos.
Chunk_Size	Tamaño del chunk.
Table	Variable categórica que indica el tipo de estructura de la que provienen los datos: - 0: vector - 1: tabla - 2: tabla por columnas
DType	Especifica el tipo de los datos en formato *numpy.dtype*.
Mean	Media de los datos.
Median	Mediana de los datos.
Sd	Desviación típica de los datos.
Skew	Coeficiente de asimetría de los datos.
Kurt	Coeficiente de apuntamiento de los datos.
Min	Mínimo absoluto de los datos.
Max	Máximo absoluto de los datos.
Q1	Primer cuartil de los datos.
Q3	Tercer cuartil de los datos.
N_Streaks	Número de rachas consecutivas por encima o por debajo de la mediana.

Tabla 2.1: Características de *chunk* extraídas

Seguidamente para cada *chunk* se procede a realizar las pruebas de compresión y descompresión con cada combinación de opciones posibles en *Blosc*, que son las descritas en el apartado 2.2.1. Conforme se van realizando las pruebas, se extraen las características descritas en la Tabla 2.2, y se van añadiendo filas a una estructura *DataFrame* auxiliar del paquete *pandas*. Finalmente al terminar las pruebas se escriben todos los datos del *chunk* al fichero *csv*.

Característica	**Descripción**
Codec	Nombre del algoritmo compresor.
Filter	Nombre del filtro.
CL	Nivel de compresión.
Block_Size	Tamaño de bloque.
CRate	Ratio de compresión obtenido.
CSpeed	Velocidad de compresiónen GB/s.
DSpeed	Velocidad de descompresión en GB/s.

Tabla 2.2: Características extraídas en las pruebas de compresión

2.4.2. Análisis de los datos recogidos

Esta tarea consiste en analizar los datos recogidos en la anterior. Por tanto el objetivo es obtener un análisis descriptivo de los datos, de forma que posteriormente se pueda generar un análisis bajo demanda para otros ficheros de datos. Además el objetivo es también familiarizarse con los paquetes de *pandas* y *matplotlib*, así como realizar algunas comprobaciones y observaciones sobre los datos.

Con respecto al análisis, éste se estructuró de la siguiente manera:

- **Descripción general**: describe la muestra y sus variables. Principalmente se describen los conjuntos de datos extraídos y se observan métricas de las pruebas de compresión generales para un códec, filtro, nivel de compresión y tamaño de bloque en concreto.
- **Correlaciones del tamaño de bloque**: se muestran gráficos de líneas relacionando el tamaño de bloque con los ratios y velocidades de compresión. Además también se comparan estas curvas según el nivel de compresión.
- **Comportamiento del nivel de compresión**: muy parecido al anterior pero para el nivel de compresión.

- **Comparativa de tablas**: se comparan las pruebas de compresión sobre los datos en forma de tabla según si son orientadas por fila o por columna.

- **Correlación entre códecs rápidos y el resto**: se analiza si es buena la correlación entre los algoritmos de compresión más rápidos como: *BloscLZ* y *LZ4* con nivel de compresión 1 y el resto de algoritmos.

- **Correlación entre características de chunk y de pruebas de compresión**: finalmente se realiza un gráfico personalizado enfrentando características de chunk con los ratios y velocidades de compresión. El objetivo es detectar de manera aproximada si se observa alguna correlación fuerte.

Para observar con más detalle el análisis realizado, en el Anexo B se encuentra el informe final obtenido.

2.4.3. Generación de datos de entrenamiento

Una vez ya familiarizados con los datos de las pruebas, hay que extraer de ellos unos datos que nos sirvan para entrenar posteriormente el algoritmo de aprendizaje. Cabe destacar que en este caso se busca aprendizaje supervisado. Por tanto, a las características extraídas anteriormente, hay que añadir las opciones que tendría el usuario final para así poder extraer entre todas las pruebas aquellas que resultan óptimas para dichas opciones. Además, considerando la velocidad de extracción y la correlación de las medidas de compresión de *BloscLZ* y *LZ4* con nivel de compresión uno y el resto de medidas de compresión, se decide añadir éstas como características.

Tras estudiar el caso se plantearon las opciones y codificaciones de la Tabla 2.3, donde **IN_CR** hace referencia al ratio de compresión, **IN_CS** a la velocidad de compresión e **IN_DS** a la velocidad de descompresión

La codificación especifica si el usuario le da importancia o no a la medida en cuestión. Para cada opción se extrae una combinación de valores objetivo: **Codec**, **Filter**, **CL**, **Block_Size** siguiendo el procedimiento detallado en el Capítulo 6. En la Fig. 2.2 se aprecia el esquema de clasificación descrito.

Tras extraer los valores objetivo se comprobó la cantidad de clases distintas resultantes y se observó que el compresor *Zlib* nunca era elegido y *Snappy* solo aparecía en dos ocasiones. Por tanto se descartaron y el número de opciones de Códec pasó de seis a cuatro. A continuación se comprobó la eficiencia del tamaño de bloque automático y las opciones seleccionadas con bloque automático y se sustituyeron por su valor real para simplificar la cantidad de tamaños de bloque resultantes.

Codificación			Descripción
IN_CR	**IN_CS**	**IN_DS**	
1	0	0	Máximo ratio de compresión.
0	1	0	Máxima velocidad de compresión.
0	0	1	Máxima velocidad de descompresión.
0	1	1	Equilibrio entre velocidad de compresión y descompresión.
1	1	0	Equilibrio entre ratio y velocidad de compresión.
1	0	1	Equilibrio entre ratio de compresión y velocidad de descompresión.
1	1	1	Equilibrio entre las tres medidas.

Tabla 2.3: Opciones de usuario y su codificación.

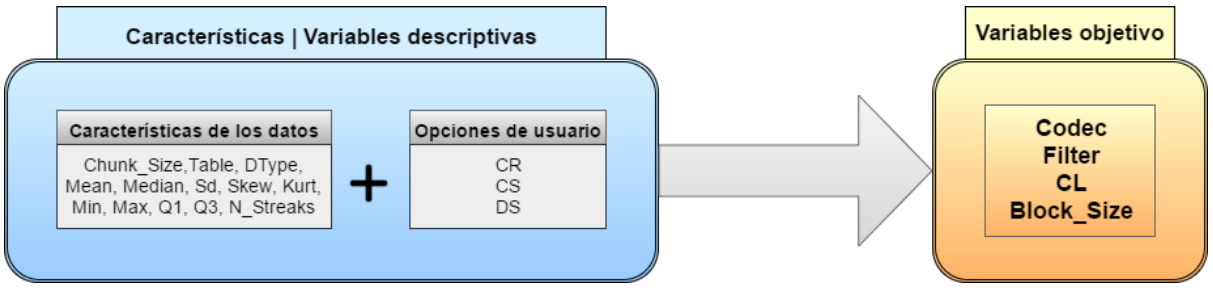


Figura 2.2: Esquema de clasificación

Seguidamente se tratan las características categóricas **Table** y **DType**, transformando la primera en dos variables binarias: **is_Table** e **is_Columnar**; y la segunda en tres variables binarias: **is_Int**, **is_Float** e **is_String**, y una continua **Type_Size** que indica el tamaño en bytes del tipo.

Finalmente se transforman las variables objetivo categóricas **Códec**, **Filter**, **CL** y **Block_Size** de forma que se convierten todas en variables binarias de las cuales, **cinco** son para **Códec**, **tres** para **Filter**, **nueve** para **CL** y otras **nueve** para **Block_Size**.

2.4.4. Análisis de las técnicas de aprendizaje automatizado

Finalmente con los datos de entrenamiento se ha procedido al estudio de los algoritmos de aprendizaje automatizado. El objetivo de este análisis es obtener el algoritmo final que se integrará en la librería de *Blosc* y que permitirá utilizarla eligiendo tres simples parámetros **IN_CR**, **IN_CS** e **IN_DS**.

Para este análisis se han comparado los siguientes tipos de algoritmos de clasificación supervisada ofrecidos por la librería de *scikit-learn*: **análisis discriminante**, **regresión multinomial**, **bosques aleatorios**, **vecinos más próximos** y **máquinas de vector soporte**. Además cada uno de estos clasificadores aceptan distintos parámetros que se detallan a lo largo del Capítulo 3.

Dado que la clasificación a realizar es de cuatro variables (**Codec**, **Filter**, **CL** y **Block_Size**), y cada variable representa una elección entre distintas opciones o **clases**, se trata de un problema de clasificación de múltiples clases y salidas (conocido como **multioutput-multiclass** dentro de *scikit-learn*). A pesar de que varios métodos de los escogidos no soportan de forma nativa **multioutput-multiclass**, *scikit-learn* proporciona métodos para adaptarlos, realizando las múltiples clasificaciones de forma secuencial y reutilizando los resultados de cada clasificación para la siguiente.

Scikit-learn proporciona varias **métricas de puntuación** para evaluar la precisión de las clasificaciones, pero ninguna de ellas funciona para **multioutput-multiclass**, excepto la predeterminada, que para este problema, no se adapta del todo. La métrica de precisión por defecto considera únicamente los aciertos totales, es decir cuándo acierta todas las variables: **Codec**, **Filter**, **CL** y **Block_Size**. Esto es demasiado restrictivo ya que acertar el nivel de compresión y tamaño de bloque con total precisión resulta imposible pues la diferencia entre niveles y tamaños cercanos es poco significativa. Por tanto, para puntuar las clasificaciones y analizar los distintos algoritmos, se utilizaron las siguientes métricas: una métrica **personalizada**, la puntuación de **Brier** y la **predeterminada** de *scikit-learn*. En la sección 3.2 se puede encontrar toda la información detallada de las distintas métricas.

Una vez decididas las puntuaciones a emplear para evaluar los algoritmos se realiza un estudio de cada uno siguiendo los siguientes pasos:

1. **Curvas de aprendizaje**: primero se dibujan las curvas de aprendizaje de cada clasificador para así observar si se beneficia o no de añadir más datos de entrenamiento y decidir una cantidad de datos a utilizar posteriormente.
2. **Curvas de validación**: mediante el uso de curvas de validación sobre los parámetros del clasificador que no dependen de otros se escogen los más adecuados evitando el sobreajuste (overfitting) y el sesgo (underfitting).

3. **Validación cruzada**: cuando múltiples parámetros del clasificador influyen entre ellos en la clasificación se realiza una validación cruzada anidada basada en el ejemplo de *scikit-learn* [8]. Al realizar la validación se exploran los distintos parámetros del clasificador posibles y se busca la combinación de todos ellos que optimiza tanto la métrica personalizada como la puntuación de Brier.

4. **Interpretación de resultados**: tras obtener los datos de la validación cruzada se comparan y se escogen los mejores parámetros del clasificador para extraer las puntuaciones finales.

5. **Reducción del número de características**: ya con el clasificador decidido se estudia la reducción del número de características. Se determina la importancia de las mismas basándose en la velocidad de extracción y en la importancia para el clasificador (en caso de que la proporcione).

Para finalizar se comparan los distintos algoritmos clasificadores teniendo en cuenta tanto la velocidad de predicción como la precisión del mismo. Todos estos análisis se pueden encontrar en el Anexo D.

2.5. Planificación temporal de las tareas

En cuanto al coste temporal del proyecto se estableció una primera estimación de las tareas a realizar durante cada quincena. Aunque surgieron contratiempos, debido a la falta de experiencia en el ámbito de la minería de datos y *Python*, se solventaron en la medida de lo posible eliminando tareas como la integración del algoritmo final en *Blosc* y reduciendo la cantidad de algoritmos de clasificación a analizar.

En principio se pretendía repartir las tareas en seis quincenas con 25 horas de trabajo semanal, de la siguiente manera:

- **Primera quincena**: adaptación a *Python* y aprendizaje de los paquetes a emplear como *numpy*, *pandas* y *scikit-learn*. Además aprender los detalles de funcionamiento de *Blosc* y su paquete para *Python*.

- **Segunda quincena**: desarrollo del programa generador de datos de las pruebas de compresión con Blosc sobre distintos ficheros.

- **Tercera quincena**: análisis descriptivo de los datos de compresión obtenidos para automatizar la generación de informes para ficheros específicos bajo demanda y familiarizarse con las librerías de *pandas* y *matplotlib*.

- **Cuarta quincena**: generación de los datos de entrenamiento finales a utilizar por los distintos algoritmos de aprendizaje automatizado.
- **Quinta y sexta**: análisis de los distintos algoritmos de aprendizaje automatizado y comparación entre ellos teniendo en cuenta la efectividad de los mismos y la velocidad de predicción.

Durante el desarrollo se mantuvo la planificación originalmente estimada, pero se realizaron pequeñas modificaciones de manera constante sobre tareas que ya se habían realizado. Esto se debió a que, en muchas ocasiones, al hacer las tareas de análisis de los datos generados, se observaban detalles que no se habían considerado anteriormente, por lo que se refinaban y repetían las generaciones de datos de manera constante cada semana. Por ejemplo, al realizar el análisis descriptivo se plantearon nuevas características a extraer en las pruebas de compresión, como el número de rachas y el tipo de los datos, por lo que se tuvieron que generar los datos de nuevo.

Cabe destacar que la generación de datos de las pruebas de compresión era un proceso costoso, por lo que para evitar retrasar el proyecto se seguía avanzando con los datos anteriores y durante el fin de semana se lanzaba el programa generador de datos en una máquina remota, para a principios de semana disponer de los datos actualizados.

2.6. Grado de consecución de los objetivos propuestos

Con respecto a los objetivos propuestos se han cumplido casi en su totalidad, a excepción de la implementación final del algoritmo dentro de la librería *Blosc* debido a la falta de tiempo. Por una parte se ha desarrollado un proceso de análisis del funcionamiento de *Blosc* consiguiendo determinar características estadísticas influyentes en el proceso de compresión de datos.

Por otro lado se ha comparado el comportamiento de diferentes tipos de algoritmos clasificadores de aprendizaje automatizado, evaluándolos según los aspectos más relevantes, que en este caso son la precisión y la velocidad de predicción. A partir de aquí solo queda seleccionar el algoritmo que más se adecue y extraer su fórmula de predicción para su posterior implementación en *Blosc*.

Cabe destacar que se ha aprendido a utilizar diferentes técnicas de minería de datos por medio del lenguaje de programación *Python* y además se ha automatizado todo este proceso de análisis permitiendo que se pueda repetir para otros datos. Finalmente destacar que, como el tipo de clasificación es bastante complejo, sería recomendable repetir todo el proceso con más datos, para evitar el sobreajuste a los datos de entrenamiento utilizados.

2.7. Conclusiones

Como conclusiones de mi estancia en prácticas con el consultor Francesc Alted, he tenido una primera experiencia laboral bastante satisfactoria. He aprendido nuevas técnicas y metodologías de la minería de datos y además, ha sido utilizando herramientas de desarrollo de software de amplia aceptación en la ciencia de datos. Destacar que en todo momento me he sentido apoyado por mi supervisor y tutora, el primero ayudándome en todos los aspectos relacionados con la programación en *Python* y la segunda orientándome en las metodologías propias de la minería de datos.

En definitiva ha sido una excelente estancia en prácticas donde he desarrollado mis habilidades de programación y de análisis de datos, lo cual se adecua perfectamente al doble grado de Ingeniería Informática y Matemática Computacional.

Capítulo 3

Algoritmos clasificadores de aprendizaje automatizado

3.1. Motivación y objetivos

Durante los últimos años la minería de datos se ha convertido en uno de los campos más demandados dentro de las ciencias aplicadas a los computadores. Esto se debe a la aparición de nuevos dispositivos y conceptos como el Internet de las cosas (*Internet of Things*, abreviado IoT), que han aumentado la generación de datos de forma abrumadora, alcanzando cifras de 72 exabytes de datos generados al mes en internet durante 2015 [6].

Estos datos se han estructurado desde los comienzos de la era digital en dispositivos de almacenamiento de muchas maneras: bases de datos, hojas de cálculo, ficheros de registro, páginas web, etc. Gracias a esto posteriormente se pueden realizar consultas sobre los datos como, por ejemplo, "¿Cuántos clientes son mayores de 30 años?", algo que sin duda alguna resulta útil. Sin embargo, los datos contienen mucha más información que no se encuentra a simple vista y es aquí donde la minería de datos entra en juego.

El principal objetivo de la minería de datos es extraer información de un conjunto de datos y transformarlo en una estructura interpretable para uso posterior. Esto se consigue mediante la aplicación de técnicas de inteligencia artificial y estadística.

Aunque la minería de datos abarca muchas otras utilidades, este trabajo se centra en la clasificación supervisada o técnicas de aprendizaje automatizado, a través de las cuales se pretende que el ordenador aprenda a reconocer patrones en los datos sin ser explícitamente programado para ello. En los problemas de clasificación supervisada, a partir de las variables descriptivas o

características observadas en los objetos de la muestra, se desea inferir la variable respuesta u objetivo. La muestra queda estructurada de la siguiente manera:

$$X = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1p} \\ x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{np} \end{pmatrix} \qquad Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

siendo n el tamaño de la muestra, p el número de características, x_{ij} el valor observado de la característica j-ésima en el objeto i-ésimo, $y_i = (y_{i1}, \ldots, y_{iR})$ la clase del objeto i-ésimo (multivariante) y R es el número de variables que componen y. A partir de aquí la muestra se divide en dos partes: una de validación de tamaño N y otra de entrenamiento de tamaño $n - N$. Las técnicas de aprendizaje automatizado son entrenadas con la muestra de entrenamiento y posteriormente aplican sus reglas para transformar las variables x_{i1}, x_{i2}, $\ldots$ x_{ip} en un valor objetivo y_i que se comprueba con la muestra de validación.

A continuación se explican los métodos de clasificación supervisada o de aprendizaje automatizado que se han utilizado a lo largo del proyecto, a saber: análisis discriminante, regresión multinomial, bosques aleatorios, vecinos más próximos y máquinas de vector soporte. Hay que comentar que de ellos ya se conocía el análisis disriminante estudiado en la asignatura *MT1033 - Fundamentos Estadísticos de la Minería de Datos* y, en cuanto al resto, se ha realizado una búsqueda bibliográfica para profundizar en ellos.

Los contenidos teóricos relacionados con los métodos de clasificación que se describen a continuación han sido extraídos mayoritariamente del libro de Daniel Peña, *Análisis de Datos Multivariantes* [13] y del libro *The Elements of Statistical Learning* de Trevor Hastie [9].

3.2. Métricas de puntuación

Antes de comenzar a hablar de los métodos de clasificación vamos a comentar un ingrediente importante en la aplicación práctica de estos métodos. Se trata de la forma de evaluar la calidad de las predicciones. Aunque en muchos casos puede resultar sencillo (por ejemplo, en una clasificación binaria se puede contar el porcentaje de aciertos y es fácil de interpretar), en el caso estudiado no es fácil. Esto se debe a que existen cuatro variables distintas, cada una de clasificación múltiple (multiclase). Por tanto, si se miden los aciertos totales (sobre las 4 variables), el porcentaje de aciertos es muy bajo y no aporta información relevante. Además en el caso de las variables del nivel de compresión y el tamaño de bloque se desea ser más permisivo. En particular, errar por una diferencia de 2 en el nivel de compresión no es extremadamente malo. Una **métrica de puntuación** es una función que mide la precisión o el error de un

método de clasificación sobre una muestra de validación. Para mantener el criterio de que una puntuación mayor es mejor, en el caso de medir el error se utilizará el valor en negativo. La siguiente ecuación representa una **métrica de puntuación** genérica:

$$f = \frac{1}{N} \sum_{t=1}^{N} \sum_{j=1}^{R} f_j(\hat{y}_{tj}, y_{tj}),$$

donde N es el tamaño de la muestra de validación, R el número de variables objetivo, f_j es la función de puntuación de la variable j-ésima, $\hat{y}_{tj}$ es el valor predicho de la variable j-ésima en la observación t e y_{tj} el valor real. Para el estudio realizado se han empleado las siguientes **métricas de puntuación**:

- **Predeterminada**: aunque la **puntuación predeterminada** sea muy restrictiva igualmente se considera a la hora de evaluar los algoritmos de clasificación. Además, también se tiene en cuenta por separado para cada variable objetivo, para ayudar a interpretar los resultados finales. Esta métrica es un valor entre 0 y 1, resultado de contar el número de aciertos y dividirlos por el tamaño de la muestra de validación, y obedece la siguiente fórmula:

$$f = \frac{1}{N} \sum_{t=1}^{N} \delta_t \qquad \delta_t = \begin{cases} 1 & si \;\; \hat{y}_t = y_t \\ 0 & resto \end{cases}$$

- **Métrica personalizada**: se definió una función para puntuar los aciertos de forma que tuviese en cuenta niveles de compresión y tamaños de bloque cercanos casi como aciertos. Básicamente la puntuación es un valor entre 0 y 1 que se calcula de la siguiente forma: si se acierta el códec y el filtro se suma 0,5, y después tanto en el nivel de compresión como en el tamaño de bloque se calcula un valor entre 0 y 0,25 en función de la diferencia entre la predicción y el dato real. La fórmula que define esta métrica es la siguiente:

$$f = \frac{1}{N} \sum_{t=1}^{N} 0,5 \; \delta_{ts_1} + f_s(\hat{y}_{ts_2}, y_{ts_2}) + f_s(\hat{y}_{ts_3}, y_{ts_3}),$$

  donde s_1 representa las clases del códec y filtro, s_2 las clases de los niveles de compresión, s_3 las clases del tamaño de bloque y

$$f_s(\hat{y}_{ts}, y_{ts}) = 0,25 \; \frac{(8 - |\hat{y}_{ts} - y_{ts}|)^2}{8^2}$$

  siendo 8 la máxima distancia posible entre dos niveles de compresión o tamaños de bloque.

- **Puntuación de Brier**: en este caso se adaptó la función de puntuación propuesta por Glenn W. Brier en 1950 [5] al problema de variable objetivo multidimensional. Esta puntuación mide el error de las probabilidades de clasificación y sólo se puede aplicar si el

método de clasificación lo permite. Esta adaptación resulta en:

$$f = -\frac{1}{N}\sum_{t=1}^{N}\sum_{k=1}^{R}\sum_{j=1}^{r_k}(\hat{p}_{tj} - y_{tj})^2,$$

donde r_k es el número de clases de la variable k-ésima, $\hat{p}_{tj}$ es la probabilidad predicha de que pertenezca a la clase número j e y_{tj} es el valor real de la variable objetivo, es decir, un 1 o 0 según si pertenece o no a la clase j. Se utiliza el valor en negativo dado que mide el error y se quiere mantener el criterio de que cuanto mayor sea el valor en la puntuación mejor es el resultado.

3.3. Análisis discriminante lineal

El método de clasificación conocido como análisis discriminante lineal puede plantearse desde dos puntos de vista: uno estadístico y otro geométrico. El planteamiento estadístico del problema consiste en considerar una muestra donde cada observación puede venir de dos o más poblaciones distintas. Cada observación se corresponde a una variable aleatoria x de dimensión p, cuya distribución se conoce en los distintos grupos considerados. El análisis discriminante se fundamenta en el supuesto de que las variables de las observaciones siguen el modelo normal multivariante y es óptimo en caso de cumplirse. A continuación se describe el método de clasificación para el caso más sencillo entre dos poblaciones que posteriormente se generalizará para múltiples grupos.

3.3.1. Clasificación entre dos poblaciones

Planteamiento

Sean P_1 y P_2 dos poblaciones distintas de una variable aleatoria vectorial x, p-variante, donde x es continua y se conocen las funciones de densidad de las poblaciones f_1, f_2. Ahora, dado un nuevo elemento x_0 con valores conocidos de sus p variables, se pretende clasificar en una de las dos poblaciones. Si se conocen las probabilidades a priori π_1, π_2, con $\pi_1 + \pi_2 = 1$, de que el elemento pertenezca a una de las poblaciones, su distribución de probabilidad es

$$f(x) = \pi_1 f_1(x) + \pi_2 f_2(x)$$

y conociendo x_0 se pueden calcular las probabilidades de que haya sido generado por cada una de las poblaciones por medio del teorema de Bayes. Estas son:

$$P(1|x_0) = \frac{f_1(x_0)\pi_1}{\pi_1 f_1(x_0) + \pi_2 f_2(x_0)} \qquad \text{y} \qquad P(2|x_0) = \frac{f_2(x_0)\pi_2}{\pi_1 f_1(x_0) + \pi_2 f_2(x_0)}.$$

Se clasificará x_0 en la población más probable; por tanto se clasificará en P_2 si:

$$\pi_2 f_2(x_0) > \pi_1 f_1(x_0)$$

Además se puede cuantificar el coste de errar al clasificar en la población contraria, nombrando este coste como $c(2|1)$ y $c(1|2)$, donde $c(i|j)$ es el coste de clasificación en P_i de una unidad que pertenece a P_j. Para minimizar el error se cambia la regla de clasificación anterior de la siguiente manera. Se clasificará en P_2 si:

$$\frac{\pi_2 f_2(x_0)}{c(2|1)} > \frac{\pi_1 f_1(x_0)}{c(1|2)} \tag{3.1}$$

Ahora la desigualdad expresa que, a igualdad del resto de términos, se clasificará en la población P_2 si

- su probabilidad inicial π_2 es más alta,
- la verosimilitud de que x_0 venga de P_2 es más alta,
- el coste de error al clasificar en P_2 es menor.

Obtención de la función discriminante lineal

Aquí se aplica el planteamiento anterior al caso en que f_1 y f_2 siguen distribuciones normales con distintos vectores de medias pero misma matriz de covarianzas. Por tanto las funciones de densidad para un elemento cualquiera x son:

$$f_i(x) = \frac{e^{\left[-\frac{1}{2}(x-\mu_i)'V^{-1}(x-\mu_i)\right]}}{(2\pi)^{p/2}|V|^{1/2}}.$$

Sustituyendo en la Ec. (3.1) se obtiene:

$$\frac{e^{\left[-\frac{1}{2}(x-\mu_2)'V^{-1}(x-\mu_2)\right]}\pi_2}{\cancel{(2\pi)^{p/2}|V|^{1/2}}c(2|1)} > \frac{e^{\left[-\frac{1}{2}(x-\mu_1)'V^{-1}(x-\mu_1)\right]}\pi_1}{\cancel{(2\pi)^{p/2}|V|^{1/2}}c(1|2)} \implies$$

$$\log \frac{e^{\left[-\frac{1}{2}(x-\mu_2)'V^{-1}(x-\mu_2)\right]}\pi_2}{c(2|1)} > \log \frac{e^{\left[-\frac{1}{2}(x-\mu_1)'V^{-1}(x-\mu_1)\right]}\pi_1}{c(1|2)}.$$

Por tanto, clasificamos x en P_2 si

$$-\frac{1}{2}(x-\mu_2)'V^{-1}(x-\mu_2) + \log\frac{\pi_2}{c(2|1)} > -\frac{1}{2}(x-\mu_1)'V^{-1}(x-\mu_1) + \log\frac{\pi_1}{c(1|2)}. \tag{3.2}$$

Nombrando D_i^2 a la distancia de Mahalanobis entre x y la media de la población i, μ_i:

$$D_i^2 = (x-\mu_i)'V^{-1}(x-\mu_i),$$

se reescribe la regla de clasificación como: clasificar x en P_2 si

$$D_1^2 - \log\frac{\pi_1}{c(1|2)} \;>\; D_2^2 - \log\frac{\pi_2}{c(2|1)}. \tag{3.3}$$

Destacar que, en el caso en que costes de error y probabilidades a priori son $c(1|2) = c(2|1)$; $\pi_1 = \pi_2$, la regla se reduce a: **clasificar en $\boldsymbol{P_2}$ si $\boldsymbol{D_1^2 > D_2^2}$**. Es decir, clasifica en la población más cercana midiendo con la distancia de Mahalanobis. Además si $\boldsymbol{V = I\sigma^2}$, la regla equivale a utilizar la **distancia euclídea**.

Interpretación geométrica

Para poder interpretar geométricamente la regla de clasificación anterior, se reescribe la Ec. (3.3) eliminando el término en común $x'V^{-1}x$, que no depende de la población, de las distancias de Mahalanobis:

$$\cancel{\frac{1}{2}x'V^{-1}x} - \mu_i'V^{-1}x + \frac{1}{2}\mu_i'V^{-1}\mu_i - \log\frac{\pi_i}{c(i|j)}.$$

Por tanto la frontera de separación en la clasificación es:

$$-\mu_1'V^{-1}x + \frac{1}{2}\mu_1'V^{-1}\mu_1 = -\mu_2'V^{-1}x + \frac{1}{2}\mu_2'V^{-1}\mu_2 - \log\frac{c(1|2)\pi_2}{c(2|1)\pi_1}$$

que, reescrita como función de x se convierte en:

$$(\mu_2-\mu_1)'V^{-1}x = (\mu_2-\mu_1)'V^{-1}\left(\frac{\mu_2+\mu_1}{2}\right) - \log\frac{c(1|2)\pi_2}{c(2|1)\pi_1}.$$

Para simplificar definimos

$$\boldsymbol{w = V^{-1}(\mu_2 - \mu_1)}, \tag{3.4}$$

de forma que la frontera queda como

$$w'x = w'\left(\frac{\mu_2+\mu_1}{2}\right) - \log\frac{c(1|2)\pi_2}{c(2|1)\pi_1},$$

que es la **ecuación del hiperplano** que tiene como vector normal a w.

En el caso de que $c(1|2)\pi_2 = c(2|1)\pi_1$ x se clasificará en P_2 si

$$w'x > w'\left(\frac{\mu_2+\mu_1}{2}\right) \quad\Longleftrightarrow\quad w'x - w'\mu_1 > w'\mu_2 - w'x.$$

Dividiendo a ambos lados por la norma de w, obtenemos $u = w/\|w\|$, el vector unitario, de forma que $u'x$ es la proyección de x en la recta de dirección u y $u'\mu_i$ las proyecciones de las medias. Como se observa en la Fig. 3.1, el hiperplano divide las poblaciones de forma que las proyecciones en la recta están lo más separadas posibles.

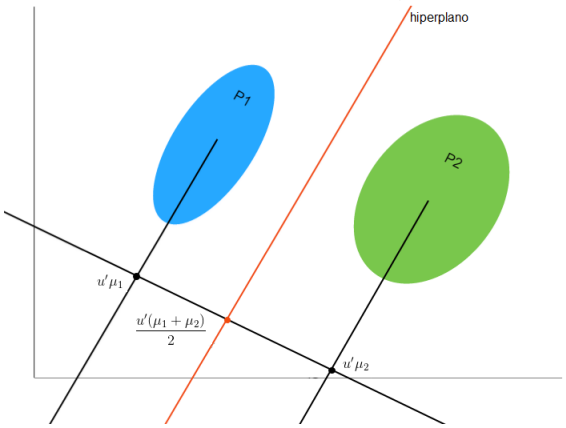


Figura 3.1: Representación del hiperplano y proyecciones que dividen dos poblaciones

3.3.2. Generalización para varias poblaciones

El objetivo ahora es dividir el espacio E_x en G regiones $A_1, \ldots, A_g, \ldots, A_G$ tales que si x pertenece a A_i el punto se clasifica en la población P_i. Entonces, A_g queda definida por los puntos con máxima probabilidad de ser generados por P_g:

$$A_g = \{x \in E_x | \pi_g f_g(x) > \pi_i f_i(x) \quad \forall i \neq g\}.$$

Esto, en el caso de que las probabilidades π_i sean iguales, y las distribuciones $f_i(x)$ sean normales con misma matriz de varianzas, equivale a calcular la distancia de Mahalanobis del punto a clasificar al centro de cada población y asignarle la población más cercana. Para minimizar las distancia de Mahalanobis $\boldsymbol{(x - \mu_g)' V^{-1} (x - \mu_g)}$, se elimina el termino $\boldsymbol{x' V^{-1} x}$ que aparece en todas las ecuaciones y se minimiza el indicador lineal

$$L_g(x) = -2\mu_g' V^{-1} x + \mu_g' V^{-1} \mu_g.$$

Definiendo $\boldsymbol{w_g = V^{-1} \mu_g}$, la regla se queda en

$$\min_g (w_g' \mu_g - 2 w_g' x).$$

Para interpretarla se observa la frontera entre dos poblaciones (i, j)

$$A_{ij}(x) = L_i(x) - L_j(x) = 0 \qquad \Longrightarrow$$
$$A_{ij}(x) = 2(\mu_i - \mu_j)'V^{-1}x + (\mu_i - \mu_j)'V^{-1}(\mu_i + \mu_j) = 0,$$

llamando $\boldsymbol{w_{ij} = V^{-1}(\mu_i - \mu_j) = w_i - w_j}$, se puede escribir la frontera que separa las poblaciones P_i, P_j como:

$$w'_{ij}x = w'_{ij}\left(\frac{\mu_i + \mu_j}{2}\right).$$

La interpretación es la misma que con dos poblaciones.

Además para G poblaciones basta con encontrar

$$r = \text{mín}(G - 1, p)$$

direcciones de proyección. Aunque se pueden construir $G(G-1)/2$ vectores w_{ij} a partir de las G medias con $G - 1$ vectores. Los demás son linealmente dependientes de éstos. Sean los $G - 1$ vectores $w_{i,i+1}$, para $i = 1, \ldots, G - 1$ a partir de éstos se puede definir cualquier $w_{i,j}$. Por ejemplo

$$w_{i,i+2} = V^{-1}(\mu_i - \mu_{i+2}) = V^{-1}(\mu_i - \mu_{i+1}) - V^{-1}(\mu_{i+1} - \mu_{i+2}) = w_{i,i+1} - w_{i+1,i+2}.$$

Por tanto, si $p > G - 1$ el número máximo de vectores w es $G - 1$ ya que el resto se deducen de éstos y, cuando $p \leq G - 1$, como son vectores de $\mathbb{R}^p$, el máximo número de vectores linealmente independientes es p.

En el libro de *Peña* [13] podemos ver que esto es equivalente a utilizar los vectores propios $u_k;\ k = 1, \ldots, r$ de la matriz $W^{-1}B$, donde

$$W = \frac{\sum_{g=1}^{G}\sum_{i=1}^{n_g}(x_i^{(g)} - \mu_g)(x_i^{(g)} - \mu_g)'}{n - G}, \qquad B = \frac{\sum_{g=1}^{G} n_g(\mu_g - \mu)(\mu_g - \mu)'}{G - 1}$$

n_g es el tamaño del grupo g-ésimo y $x_i^{(g)}$ es la observación i-ésima del grupo g-ésimo. Con los vectores propios se construye la matriz U que contiene en columnas los vectores propios de $W^{-1}B$ y se calculan las proyecciones de cada $\mu_g;\ g = 1, \ldots, G$ como $z_{\mu_g} = U'\mu_g$. Finalmente para clasificar una observación x_0, se calcula $z_0 = U'x_0$ y se clasifica en el grupo g cuya distancia entre z_0 y z_{μ_g} sea mínima.

3.4. Regresión multinomial

En la sección anterior se ha mostrado como el análisis discriminante permite clasificar adecuadamente datos cuando la distribución conjunta de las observaciones es normal multivariante.

Sin embargo, en muchos casos los datos no son normales, por lo que no hay garantías de que el análisis discriminante funcione correctamente. En este apartado introducimos un método alternativo que considera la variable objetivo y, como variable respuesta binaria, y aplica un modelo de regresión sobre la misma.

3.4.1. Regresión logística

Como en la sección anterior abordaremos el problema de clasificación entre dos poblaciones y posteriormente lo extenderemos para múltiples. para este modelo definimos pues la variable de clasificación y, que será 0 cuando pertenece a la primera población P_1, y 1 cuando pertenece a la segunda P_2. Por tanto, la muestra estará formada por n elementos del tipo (x_i, y_i), siendo x_i el vector de características e y_i la variable binaria objetivo. Como primera idea se podría formular un modelo de regresión lineal:

$$y = \beta_0 + \beta_1' x + \varepsilon, \qquad \varepsilon \ N(0,1)$$

que como se puede observar en el Apéndice 13.2 del libro de Daniel Peña [13], esto equivaldría a la función lineal discriminante con $c(1|2) = c(2|1)$; $\pi_1 = \pi_2$ Ec. 3.3.

Tomando esperanzas para $x = x_i$:

$$E[y|x_i] = \beta_0 + \beta_1' x_i \tag{3.5}$$

si p_i es la probabilidad de que y tome el valor 1 cuando $x = x_i$:

$$p_i = P(y = 1|x_i)$$

como la variable y es de Bernoulli, con probabilidades p_i y $1 - p_i$ para los valores uno y cero, respectivamente, su esperanza será:

$$E[y|x_i] = p_i \cdot 1 + (1 - p_i) \cdot 0 = p_i, \tag{3.6}$$

e igualando (3.5) con (3.6) se obtiene:

$$p_i = \beta_0 + \beta_1' x_i \tag{3.7}$$

Sin embargo, al estimar el modelo lineal (3.5), las predicciones de $\hat{y}_i$ (es decir, $\hat{p}_i$ por Ec. (3.7)) no se garantiza que estén entre cero y uno, lo cual supone un problema para interpretar la regla de clasificación. Para que el modelo proporcione directamente la probabilidad de pertenecer a cada población se debe transformar la variable de respuesta para limitarla entre cero y uno. Escribiendo:

$$p_i = F(\beta_0 + \beta_1' x_i),$$

como se busca p_i entre cero y uno, se busca F que sea no decreciente y acotada en este intervalo. Esto se corresponde con las funciones de distribución. Aunque se puede escoger entre varias funciones de distribución viables se detallará únicamente el modelo utilizando la función de distribución logística F dada por:

$$p_i = \frac{1}{1 + e^{-\beta_0 - \beta_1' x_i}}.$$

Esta función es muy ventajosa por la continuidad y por la siguiente propiedad:

$$1 - p_i = \frac{e^{-\beta_0 - \beta_1' x_i}}{1 + e^{-\beta_0 - \beta_1' x_i}} = \frac{1}{1 + e^{\beta_0 + \beta_1' x_i}} \qquad \Longrightarrow$$

$$g_i = \log \frac{p_i}{1 - p_i} = \beta_0 + \beta_1' x_i, \tag{3.8}$$

donde la variable *Logit* g, representa en escala logarítmica la diferencia entre las probabilidades de pertenecer a ambas poblaciones. Además al ser una función lineal de las variables explicativas facilita la estimación e interpretación del modelo.

Estimación del modelo por MV

Como la variable y es de tipo Bernoulli, la función de probabilidad para y_i cualquiera es:

$$P(y_i) = p_i^{y_i}(1 - p_i)^{1 - y_i},$$

y para la muestra:

$$P(y_1, \ldots, y_n) = f(y_1) \cdot f(y_2) \cdot \ldots \cdot f(y_n) = \prod_{i=1}^{n} p_i^{y_i}(1 - p_i)^{1 - y_i}.$$

A continuación se toman logaritmos y se simplifica:

$$\log P(y_1, \ldots, y_n) = \log \left(\prod_{i=1}^{n} p_i^{y_i}(1 - p_i)^{1 - y_i} \right) = \sum_{i=1}^{n} \log(p_i^{y_i}(1 - p_i)^{1 - y_i}) =$$

$$\sum_{i=1}^{n} (y_i \log p_i + (1 - y_i) \log(1 - p_i)) = \sum_{i=1}^{n} (y_i \log p_i + \log(1 - p_i) - y_i \log(1 - p_i)) =$$

$$\sum_{i=1}^{n} \left[y_i \log \left(\frac{p_i}{1 - p_i} \right) + \log(1 - p_i) \right].$$

Teniendo en cuenta (3.8) y la siguiente igualdad

$$1 - p_i = \frac{1}{1 + e^{\beta_0 + \beta_1' x_i}},$$

se deja la expresión en función de los parámetros de interés $\boldsymbol{\beta}$:

$$L(\boldsymbol{\beta}) = \sum_{i=1}^{n} y_i \boldsymbol{x_i'\beta} - \log(1-p_i)^{-1} = \sum_{i=1}^{n} y_i \boldsymbol{x_i'\beta} - \sum_{i=1}^{n} \log\left(1 + e^{\boldsymbol{x_i'\beta}}\right).$$

Ahora con la función soporte $L(\boldsymbol{\beta})$, puesto que la función de verosimilitud es siempre cóncava, para obtener los estimadores de máxima verosimilitud se deriva e iguala a cero para buscar el máximo:

$$\frac{\partial L(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = \sum_{i=1}^{n} y_i \boldsymbol{x_i} - \sum_{i=1}^{n} \boldsymbol{x_i}\left(\frac{e^{\boldsymbol{x_i'\beta}}}{1 + e^{\boldsymbol{x_i'\beta}}}\right) = \sum_{i=1}^{n} \boldsymbol{x_i}\left(y_i - \frac{1}{1 + e^{-\boldsymbol{x_i'\beta}}}\right) = 0 \implies$$

$$\sum_{i=1}^{n} y_i \boldsymbol{x_i} = \sum_{i=1}^{n} \boldsymbol{x_i}\hat{p}_i \implies \sum_{i=1}^{n} \boldsymbol{x_i}(y_i - \hat{p}_i) = 0. \tag{3.9}$$

Como no se puede obtener una solución analítica para el valor de $\hat{\boldsymbol{\beta}}_{MV}$, se recurre a un algoritmo tipo *Newton-Raphson*. Desarrollando el vector $(\partial L(\boldsymbol{\beta})/\partial \boldsymbol{\beta}))$ alrededor de un punto $\boldsymbol{\beta_a}$, se tiene

$$\frac{\partial L(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = \frac{\partial L(\boldsymbol{\beta_a})}{\partial \boldsymbol{\beta}} + \frac{\partial^2 L(\boldsymbol{\beta_a})}{\partial \boldsymbol{\beta}\partial \boldsymbol{\beta'}}(\boldsymbol{\beta} - \boldsymbol{\beta_a})$$

donde, para que el punto $\boldsymbol{\beta_a}$, corresponda al máximo de verosimilitud su primera derivada debe anularse. Con la condición $\partial L(\boldsymbol{\beta_a})/\partial \boldsymbol{\beta}) = 0$, se tiene que

$$\boldsymbol{\beta_a} = \boldsymbol{\beta} + \left(\frac{\partial^2 L(\boldsymbol{\beta_a})}{\partial \boldsymbol{\beta}\partial \boldsymbol{\beta'}}\right)^{-1}\left(\frac{\partial L(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}}\right). \tag{3.10}$$

Esta expresión muestra cómo obtener el punto máximo $\boldsymbol{\beta_a}$, a partir de un punto próximo cualquiera $\boldsymbol{\beta}$. La expresión depende de la matriz de segundas derivadas que, en el óptimo, es la inversa de la matriz de covarianzas asintótica de los estimadores MV. Su expresión se obtiene derivando la expresión (3.9):

$$\hat{\boldsymbol{M}}^{-1} = \left(\frac{\partial^2 L(\boldsymbol{\beta_a})}{\partial \boldsymbol{\beta}\partial \boldsymbol{\beta'}}\right) = \sum_{i=1}^{n} \boldsymbol{x_i x_i'} w_i \tag{3.11}$$

donde

$$w_i = \frac{e^{\boldsymbol{x_i'\beta}}}{\left(1 + e^{\boldsymbol{x_i'\beta}}\right)^2} = p_i(1-p_i).$$

Reemplazando en (3.10) las expresiones (3.11) y (3.9) se obtiene el siguiente método para obtener un nuevo estimador $\hat{\boldsymbol{\beta}}_{\boldsymbol{k+1}}$ a partir de un estimador inicial $\hat{\boldsymbol{\beta}}_{\boldsymbol{k}}$

$$\hat{\boldsymbol{\beta}}_{\boldsymbol{k+1}} = \hat{\boldsymbol{\beta}}_{\boldsymbol{k}} + \left(\sum_{i=1}^{n} \boldsymbol{x_i x_i'} \hat{w}_i \right)^{-1} \left(\sum_{i=1}^{n} \boldsymbol{x_i} (y_i - \hat{p}_i) \right)$$

donde $\hat{p}_i$ y $\hat{w}_i$ se calculan con el estimador inicial $\hat{\boldsymbol{\beta}}_{\boldsymbol{k}}$. El método puede reescribirse como:

$$\hat{\boldsymbol{\beta}}_{\boldsymbol{k+1}} = \hat{\boldsymbol{\beta}}_{\boldsymbol{k}} + \left(X' \hat{W} X \right)^{-1} X' \left(Y - \hat{Y} \right), \tag{3.12}$$

donde $\hat{W}$ es una matriz diagonal con términos $\hat{p}_i(1 - \hat{p}_i)$. A partir de aquí se escoge un valor inicial $\hat{\boldsymbol{\beta}}_{\mathbf{0}}$ y se itera hasta que el método converge, aunque esta condición no está garantizada.

3.4.2. Generalización para G poblaciones: regresión multinomial

Para este caso se plantea el modelo *logit* para G poblaciones. Aquí p_{ig} es la probabilidad de que la observación i pertenezca a la clase g, se puede escribir:

$$p_{ig} = \frac{e^{\boldsymbol{\beta_{0g}} + \boldsymbol{\beta'_{1g}} \boldsymbol{x_i}}}{1 + \sum_{j=1}^{G-1} e^{\boldsymbol{\beta_{0j}} + \boldsymbol{\beta'_{1j}} \boldsymbol{x_i}}}, \quad j = 1, \ldots, G-1,$$

y

$$p_{iG} = \frac{1}{1 + \sum_{j=1}^{G-1} e^{\boldsymbol{\beta_{0j}} + \boldsymbol{\beta'_{1j}} \boldsymbol{x_i}}}, \quad j = 1, \ldots, G-1.$$

Con esto se garantiza que $\sum_{g=1}^{G} p_{ig} = 1$. La comparación entre dos categorías se realiza de la manera esperada

$$\frac{p_{ig}}{p_{ij}} = \frac{e^{\boldsymbol{\beta_{0g}} + \boldsymbol{\beta'_{1g}} \boldsymbol{x_i}}}{e^{\boldsymbol{\beta_{0j}} + \boldsymbol{\beta'_{1j}} \boldsymbol{x_i}}} = e^{(\boldsymbol{\beta_{0g}} - \boldsymbol{\beta_{0j}})} e^{(\boldsymbol{\beta'_{1g}} - \boldsymbol{\beta'_{1j}}) \boldsymbol{x_i}}.$$

Por lo que respecta a la estimación de estos modelos, éstos son extensiones del desarrollo del anterior apartado y no se entrará en detalles.

3.5. Árboles de decisión

Los árboles de decisión son un modelo predictivo en forma de grafo tipo árbol donde se transforman las variables observadas de un objeto $x_i = (x_{i1}, \ldots, x_{ip})$, representadas por las ramas del árbol, a conclusiones sobre el valor de la variable objetivo y_i a predecir del objeto (representado en las hojas del árbol). Cuando la variable objetivo puede tomar un número finito de valores se les llama árboles de clasificación, las ramas representan decisiones sobre las características y las hojas son clases. Por el contrario cuando la variable objetivo puede tomar valores continuos (números reales) se les llama árboles de regresión. En la Fig. 3.2 se muestra un ejemplo de árbol de decisión.

Basándose en árboles de decisión, también se han desarrollado otras técnicas conocidas como métodos de conjuntos (*ensemble methods*) que utilizan internamente varios árboles de decisión. Algunas de estas técnicas son: árboles mejorados (*Boosted Trees*), empaquetado (*Bagging*) y bosques aleatorios (*Random Forest*). En este trabajo se ha experimentado únicamente con los bosques aleatorios.

Además existen varios algoritmos para la construcción de árboles de decisión, siendo los más notables: ID3, C4.5 y CART (*Classification And Regression Tree*). Dado que los árboles de decisión implementados en *scikit-learn* están basados en CART, a continuación se describe el funcionamiento de dicho algoritmo.

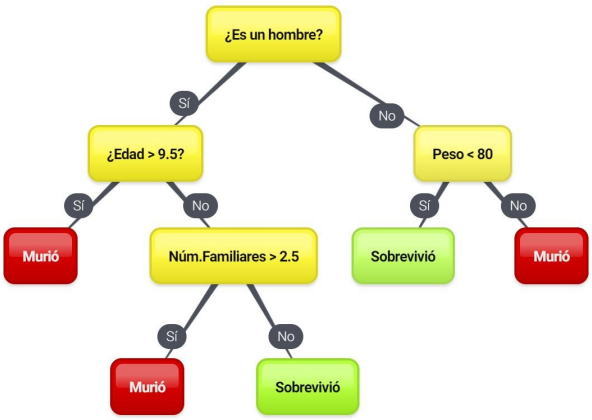


Figura 3.2: Ejemplo de árbol de decisión

3.5.1. Algoritmo CART

El algoritmo CART (*Classification And Regression Tree*) de construcción de árboles de decisión fue introducido por primera vez en 1984 de la mano de Breiman [4]. El procedimiento no se basa en ningún modelo estadístico formal, sino que realiza particiones binarias por medio de los valores de las variables, y por tanto se asemeja más al comportamiento de toma de decisiones de un humano.

A partir de los datos de entrenamiento el algoritmo comienza seleccionando una variable x_i para la cual obtiene un punto de corte c, de tal modo que separa los datos según si $x_i \leq c$ o $x_i > c$. De este nudo inicial saldrán dos ramas y para cada una se repetirá el proceso de selección de variables y punto de corte, volviendo a dividir los datos en dos en cada rama. El procedimiento se repite hasta que se hayan clasificado todas las observaciones en su grupo o cuando se cumplan las condiciones de corte. Para construir el árbol hay que tener en cuenta las siguientes decisiones.

1. Selección de variables y puntos de corte

Para seleccionar la variable que se utilizará para hacer la partición en el nudo actual se calcula primero la proporción de observaciones que pasan por el nudo para cada grupo.

Sea G el número de grupos y $p(g|t)$ las proporciones de las observaciones que llegan al nudo t pertenecientes a cada una de las clases. Se define la **impureza** o **entropía** del nudo t como:

$$I(t) = -\sum_{g=1}^{G} p(g|t) \log p(g|t). \tag{3.13}$$

Esta medida es no negativa y mide la diversidad. Por ejemplo, para $G = 2$ sería:

$$I(t) = -p \log p - (1-p) \log(1-p).$$

El valor que maximizaría la función sería $p = 0{,}5$ y cuando p se acerca a 0 o a 1, la función se aproxima a 0. Para el caso de G grupos la función se acercará a 0 cuando para algún g, $p(g|t) \approx 1$; por otro lado la función alcanzará su máximo cuando $p(g|t) = \frac{1}{G} \quad \forall g \in G$.

A continuación se considera el conjunto formado por todas las preguntas q posibles del tipo: ¿Es $x_i < a$?, $\forall i = 1, \ldots, p$ y $a \in (-\infty, \infty)$. Es decir, el conjunto de todas las posibles divisiones para todas las variables. Para cada pregunta q, en el nodo t, llamamos t_S y t_N a los nudos que surgen de t como consecuencia de responder a la pregunta q con un *"Sí"* o con un *"No"*; respectivamente se consideran como p_S y p_N las proporciones de observaciones del nudo

t resultantes. Llamando $I(t_N)$ e $I(t_S)$ a las correspondientes impurezas, se define el **cambio de entropía** producido tras el nudo t por la pregunta q como:

$$\Delta I(t, q) = I(t) - p_S I(t_S) - p_N I(t_N).$$

Se escoge para cada nodo la pregunta que maximiza el cambio de entropía, es decir, aquella pregunta que maximiza la homgeneidad y por tanto minimiza la entropía en los nudos resultantes (t_S y t_N) tras la partición.

Otra posibilidad, menos utilizada en la práctica, pero que lleva implementada *scikit-learn* es utilizar como medida de impureza Ec. (3.13) el indice de Gini.

$$I_{gini}(t) = \sum_{g=1}^{G} p(g|t)(1 - p(g|t))$$

2. Determinación de nudos terminales

El proceso anterior debería continuar hasta que se hayan clasificado todas las observaciones correctamente en su grupo. Sin embargo, esto puede generar muchos nudos cuando el número de variables es grande y se pueden plantear métodos para simplificar o *"podar"* el árbol resultante. Un método podría ser establecer un número mínimo de observaciones en el nudo para realizar la división o establecer que pare al alcanzar cierto grado de homogeneidad en el grupo (sin que todas las observaciones sean del mismo grupo). Otra posibilidad es simplemente parar por **máxima profundidad** del árbol, que consiste simplemente en *"podar"* el árbol a una altura determinada o más bien dejar de entrenarlo más allá de esa altura y establecer la clase de esos últimos grupos por mayoría. Este último método ha sido el utilizado en nuestros análisis.

3. Asignación de grupos a los nudos terminales

Una vez alcanzado el nudo terminal, para determinar su valor, se escoge aquel del que hayan más observaciones que pasen por ese nudo, es decir, $\max p(g|t)$.

3.5.2. Bosques aleatorios

Uno de los principales problemas de los árboles de decisión es el sobreajuste a los datos de entrenamiento. En particular, si no se establecen criterios de corte, el árbol siempre se entrena de forma que se adecua perfectamente a los datos de entrenamiento, lo que implica que las predicciones sean sensibles a datos distintos de la muestra de entrenamiento. Por esto surgen

los bosques aleatorios que corrigen este tipo de errores y mejoran significativamente la precisión de los árboles.

Los bosques aleatorios son una combinación de árboles de decisión de forma que cada árbol es generado de forma aleatoria conforme al siguiente procedimiento. Cada árbol del bosque formado por B árboles se construye de la siguiente manera:

1. Para cada nudo t del árbol:
   a) Seleccionar m variables al azar de las p variables.
   b) Escoger el mejor punto de corte entre las m variables.
   c) Dividir el nudo en dos.
2. El árbol resultante construido T_b se añade al conjunto de árboles $\{T_b\}_1^B$.

Sea $\hat{C}_b(x)$ la predicción del b-ésimo árbol del bosque aleatorio para la variable objetivo y. Se define la predicción del bosque aleatorio como

$$\hat{C}^B(x) = \textit{voto por mayoría}\{\hat{C}_b(x)\}_1^B;$$

de este mismo modo la probabilidad predicha de que x pertenezca a la clase r se define como

$$\hat{P}_r^B(x) = \frac{\textit{card}\{T_b \ \textit{t.q.}\ \hat{C}_b(x) = r\}_1^B}{B}.$$

Otra posibilidad que lleva implementada *scikit-learn* es utilizar una muestra **bootstrap** para construir cada árbol. Una muestra **bootstrap** se obtiene realizando muestreo aleatorio con reemplazamiento de toda la muestra hasta obtener el tamaño de la muestra original.

En el artículo de Leo Breiman, *Random Forests* [3], se demuestra que el error de generalización converge hasta cierto límite al aumentar el número de árboles que forman el bosque, lo cual, explica porque no se sobreajustan a los datos. Finalmente cabe destacar que con este método se consiguen resultados comparables con los métodos de mejora adaptativa (*AdaBoost*), puesto que:

1. Tienen una precisión muy similar o incluso mejor.
2. Es robusto contra el ruido y datos atípicos.
3. Es más rápido que la mejora adaptativa y el empaquetado (*Bagging*).
4. Proporciona estimaciones con probabilidades, útiles para calcular errores, correlaciones e importancia de las características.
5. Los cálculos son simples y fáciles de paralelizar.

3.6. Vecinos más próximos

El método de clasificación de los vecinos más próximos destaca por su simplicidad y buenos resultados en poblaciones no normales. El método consiste en:

1. Establecer una medida de distancia entre puntos de la muestra.
2. Calcular las distancias entre el punto que se pretende clasificar, x_0 y el resto de puntos de la muestra.
3. Seleccionar los k puntos (**vecinos**) más próximos a x_0 de la muestra.
4. Calcular la proporción de los k puntos que pertenece a cada una de las clases.
5. Clasificar el punto x_0 en la clase que tenga mayor proporción de vecinos.

Este método se conoce como k-vecinos más próximos; en el caso de $k = 1$ simplemente se le asigna a cada elemento la clase del vecino más próximo, lo que implica sobreajuste en la clasificación. Por el contrario, si se elige un valor de k excesivamente grande, la clasificación estará sobregeneralizada y no acertará en sus predicciones. Por ello el problema clave de este método es encontrar un valor adecuado de k. Para ello se suele utilizar $k = \sqrt{n_g}$ donde n_g es el tamaño de grupo medio o también se pueden probar distintos valores de k dentro de un rango y elegir el que obtenga menor error de clasificación. En la Fig. 3.3 se puede observar la diferencia en las fronteras de los grupos al clasificar con $k = 1$ o $k = 15$.

3.7. Máquinas de vector soporte

Las máquinas de vector soporte, en adelante SVM (Support Vector Machines), es un método que, al igual que el análisis discriminante, busca fronteras lineales para separar los datos pero en lugar de buscar una reducción del espacio de los datos para resolver el problema, busca un espacio de dimensión mayor donde los puntos se puedan separar linealmente.

Sea una muestra de n elementos descritos por $x_i \in \mathbb{R}^p$ para las características de la observación i e y_i la variable binaria de clasificación con posibles valores -1 y +1 en vez de los habituales cero y uno. El conjunto de n datos será separable de forma lineal si se puede encontrar un vector $w \in \mathbb{R}^p$ que genere un hiperplano que separe perfectamente las observaciones. Por ejemplo, las observaciones de la clase $y_i = -1$ verificaran que $w'x_i + b < -1$, para un determinado escalar b, mientras que las de clase $y_i = 1$ obedecen a $w'x_i + b > 1$. Las dos igualdades presentadas pueden escribirse conjuntamente como:

$$y_i(w'x_i + b) \geq 1 \quad \text{para } i = 1, \ldots, n. \tag{3.14}$$

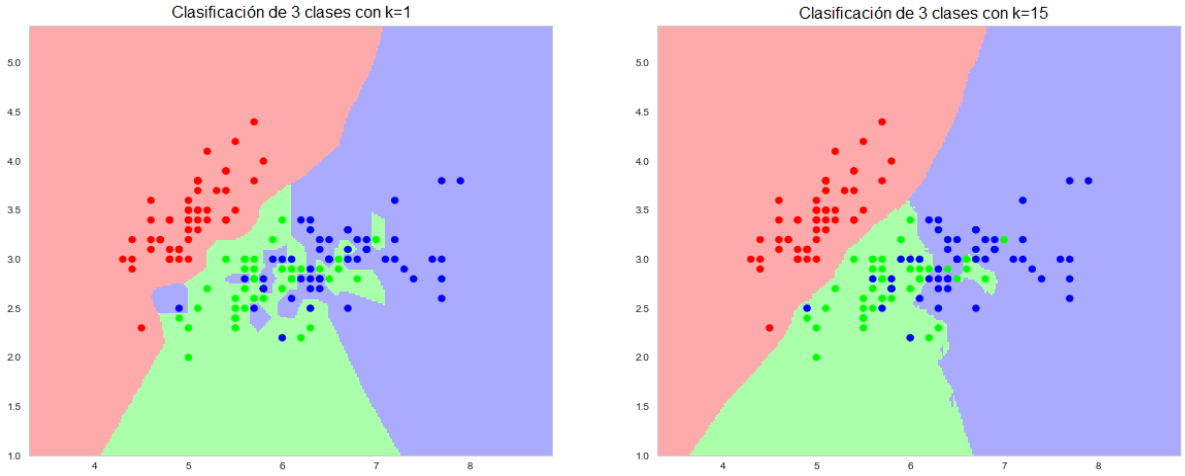


Figura 3.3: Ejemplo de vecinos más próximos con distintos valores de k

Sea

$$f(x_i) = w'x_i + b,$$

el valor del hiperplano de separación óptima entre los dos grupos para un punto x_i. La distancia entre un punto x_i y el hiperplano se obtiene mediante la proyección del punto en la dirección w y se calcula mediante $w'x_i/\|w\|$. Como los puntos cumplen $y_i(w'x_i + b) \geq 1$ faltaría maximizar las distancias de los puntos al plano por lo que se busca el máximo de

$$\frac{y_i(w'x_i + b)}{\|w\|}, \quad \text{para } i = 1, \ldots, n.$$

Para que se cumpla esto el numerador tendrá que ser positivo y el denominador mínimo, lo que conduce al siguiente problema de optimización:

$$\begin{aligned} &\textbf{minimizar } \|w\|^2 \\ &\textbf{sujeto a } y_i(w'x_i + b) \geq 1, \quad i = 1, \ldots, n. \end{aligned}$$

En este caso particular el resultado de esta maximización es el mismo vector w de la Ec. (3.4).

Para extender esta idea al caso en que los datos no son separables linealmente (ver Fig. 3.4), se introduce una serie de variables ξ_i en las restricciones que se convierten en:

$$\begin{aligned} w'x_i + b &\geq 1 - \xi_i, \quad para \quad y_i = 1; \\ w'x_i + b &\geq -1 + \xi_i, \quad para \quad y_i = -1; \\ \xi_i &\geq 0 \ \forall i. \end{aligned}$$

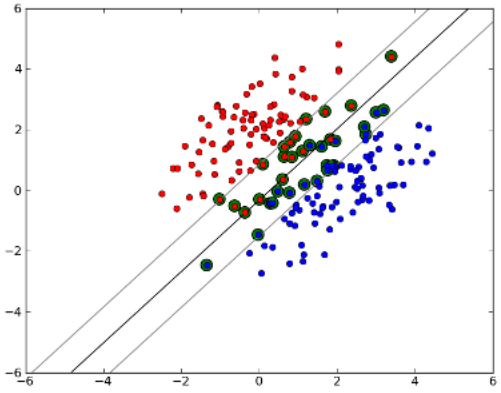


Figura 3.4: Ejemplo clasificador SVM con datos no separables linealmente

Entonces, para que se produzca un error, el correspondiente valor ξ_i debe exceder la unidad y por tanto $\sum_{i=1}^{n} \xi_i$ es un límite superior del número total de errores de entrenamiento. De esta forma se modifica la función objetivo a minimizar sumando $C(\sum_{i=1}^{n} \xi_i)^k$, donde C es un parámetro escogido por el usuario, que se corresponde con la penalización de los errores. Entonces el problema de optimización se transforma en:

$$\textbf{minimizar } \|w\|^2 + C\left(\sum_{i=1}^{n} \xi_i\right)^k$$
$$\textbf{sujeto a } y_i(w'x_i + b) \geq \quad 1 - \xi_i \text{ y } \xi_i \geq \quad 0 \ \forall i,$$

donde el parámetro C determina el equilibrio entre el tamaño del margen y asegura que x_i caiga en el lado correcto del margen. Para valores de C lo suficientemente pequeños se comportará igual que en el caso de ser linealmente separables, pero en caso de no poder separarse de forma lineal igualmente aprende una regla de clasificación viable.

Con este planteamiento, se trata de un problema de programación convexa para cualquier entero positivo k; para $k = 2$ y $k = 1$ es además un problema de programación cuadrática con la particularidad de que con $k = 1$ resolviendo por la dualidad de Lagrange se obtiene el problema simplificado

$$\textbf{maximizar } f(\alpha_1 \ldots \alpha_n) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y_i \alpha_i (x_i x_j) y_j \alpha_j \tag{3.15}$$

$$\textbf{sujeto a } \sum_{i=1}^{n} \alpha_i y_i = 0, \text{ y } 0 \leq \alpha_i \leq C \ \forall i$$

como el problema es una función cuadrática de α_i sujeta a restricciones lineales se puede resolver por medio de algoritmos de programación cuadráticos. Las variables α_i se definen de forma que

$$w = \sum_{i=1}^{n} \alpha_i y_i x_i$$

además, $\alpha_i = 0$ cuando x_i cae en el lado correcto del margen y $0 \leq \alpha_i \leq C$ cuando x_i cae en el margen. El valor de b se puede recuperar encontrando un x_i en el margen y resolviendo

$$y_i(w'x_i + b) = 1$$

Finalmente destacar que, aunque se trata de un clasificador de tipo lineal, se pueden aplicar trucos *kernel* (originalmente propuestos por Aizerman [1]) para realizar clasificaciones no lineales. Para ello se sustituye la función lineal que define la frontera por una función en un espacio de Hilbert con núcleo reproductor $f(x) = \sum_{i=1}^{n} \alpha_i K(x, x_i)$, usando por ejemplo la función de núcleo gaussiana $K(x_i, x_j) = e^{-\gamma \|x_i - x_j\|^2}$ para $\gamma > 0$. La complejidad matemática de este procedimiento excede de los objetivos de aprendizaje de este Trabajo de Final de Grado y por ello nos limitamos a comentar que la Ec. 3.15 se convierte en:

$$\textbf{maximizar } f(\alpha_1 \ldots \alpha_n) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y_i \alpha_i K(x_i, x_j) y_j \alpha_j$$

Capítulo 4

Desarrollo del software

En este capítulo se detallan los procedimientos y tecnologías utilizadas en el desarrollo del proyecto. Primero de todo, comentar que dada la naturaleza del proyecto, no se han podido aplicar técnicas de programación orientada a objetos. Consecuentemente, no se han realizado diagramas de clases ni de casos de uso. Al tratarse de un proyecto de minería de datos, éste está compuesto por distintos procesos en los que se extraen y/o transforman los datos de entrada hasta que finalmente se extraen conocimientos interpretables y conclusiones sobre los mismos. En la Fig. 4.1 se muestran los procesos implicados en un proyecto de minería de datos.

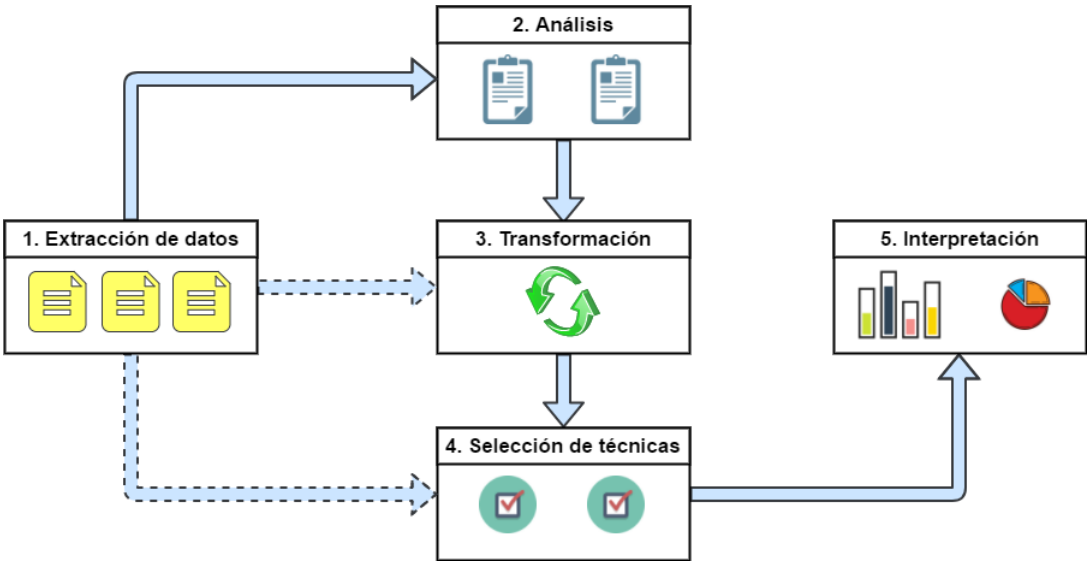


Figura 4.1: Diagrama de procesos.

En un proyecto de minería de datos se parte de un conjunto de datos, el cual se ha seleccionado, o en nuestro caso se extrae. A partir de estos datos, si están "limpios", se pueden seleccionar y aplicar las técnicas de minería de datos sobre ellos. Sin embargo, el caso más habitual es que no estén "limpios"por lo que hay que transformarlos para que se puedan aplicar las técnicas de minería de datos. Adicionalmente se puede realizar un análisis de los propios datos que permita extraer conocimiento de los mismos y que ayude en la selección de técnicas. Finalmente se interpretan y evalúan las técnicas de minería de datos empleadas.

A continuación se describe el desarrollo de software durante los procesos involucrados a lo largo del proyecto que han sido: **extracción de los datos**, **análisis de los datos extraídos**, **transformación de los datos de entrada**, **selección y aplicación de las técnicas de clasificación** e **interpretación, y evaluación de las técnicas empleadas**.

4.1. Extracción de los datos

Para comenzar un proyecto de minería de datos es necesario disponer de un conjunto de datos sobre el que trabajar. Como no se disponía de datos de pruebas de compresión con todas las opciones de *Blosc* se tuvo que generar el conjunto de datos, realizando pruebas de compresión sobre distintos ficheros en formato *HDF5*. Para ello, se realizó un programa en *Python* (ver Anexo A) para la realización de las pruebas de compresión con *Blosc* y la generación de los datos en formato *csv* con la información extraída. Para la realización del programa se emplearon las siguientes librerías:

- **PyTables**, para explorar y extraer los datos contenidos en ficheros *HDF5*.
- **python-blosc**, el *wrapper* de *Blosc* para *Python*, para realizar las pruebas de compresión en los datos extraídos.
- **numpy** y **scipy**, para la extracción de propiedades estadísticas de los datos.
- **pandas**, para la construcción del fichero *csv* con la información extraída.

A continuación se presenta el código principal del programa de generación de datos:

```
1 FILENAMES = ('HiSPARC.h5',)
2 PATH = '/home/francesc/datasets/tests/'
3 BLOCK_SIZES = (0, MINIMUM_SIZE, KB16, KB32, KB64, KB128, KB256, KB512, MB, MB2)
4 C_LEVELS = range(1, 10)
5 COLS = ['Filename', 'DataSet', 'Table', 'DType', 'Chunk_Number', 'Chunk_Size',
6     'Mean', 'Median', 'Sd', 'Skew', 'Kurt', 'Min', 'Max', 'Q1', 'Q3',
7     'N_Streaks', 'Block_Size', 'Codec', 'Filter', 'CL', 'CRate', 'CSpeed',
8     'DSpeed']
```

```
9  blosc.set_nthreads(4)
10
11 if not os.path.isfile('blosc_test_data.csv'):
12   pd.DataFrame(columns=COLS).to_csv(
13     'blosc_test_data.csv', sep='\t', index=False)
14
15 for filename in FILENAMES:
16   for path, d_type, table, buffer in file_reader(PATH + filename):
17     n_chunks = calculate_nchunks(buffer.dtype.itemsize, buffer.size)
18     print("Starting tests with %-s %-s t%-s" %(filename, path, table))
19     if buffer.dtype.kind in ('S', 'U'):
20       is_string = True
21       filters = (blosc.NOSHUFFLE,)
22     else:
23       is_string = False
24       filters = (blosc.NOSHUFFLE, blosc.SHUFFLE, blosc.BITSHUFFLE)
25     for i, chunk in enumerate(chunk_generator(buffer)):
26       chunk_id = (filename, path, table, d_type, i + 1,
27       chunk.size * chunk.dtype.itemsize / MB)
28       if is_string:
29         chunk_features = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
30       else:
31         chunk_features = extract_chunk_features(chunk)
32         chunk_features += (calculate_streaks(chunk,
33                            chunk_features[1]),)
34         df = pd.DataFrame()
35       for block_size in BLOCK_SIZES:
36         blosc.set_blocksize(block_size)
37         for codec in blosc.compressor_list():
38           for filt in filters:
39             for clevel in C_LEVELS:
40               row_data = chunk_id + chunk_features\
41                 + (block_size / 2**10, codec,
42                 blosc.filters[filt], clevel)\
43                 + test_codec(chunk, codec, filt, clevel)
44               df = df.append(
45                 dict(zip(COLS, row_data)), ignore_index=True)
46       print("%5.2f%%%-s %-s t%-s chunk %d completed" %
47           ((i + 1) / n_chunks * 100, filename, path, table, (i + 1)))
48       with open('blosc_test_data.csv', 'a') as f:
49         df = df[COLS]
50         df.to_csv(f, sep='\t', index=False, header=False)
51       print('CHUNK WRITED')
```

Como se observa en el programa, en las primeras líneas (1-9), se declara una tupla con los ficheros *HDF5* sobre los que realizar las pruebas de compresión, los tamaños de bloque a utilizar en las pruebas, el rango de niveles de compresión a emplear, la cantidad de hilos que utilizará *Blosc* y los nombres de las columnas del fichero *csv*. Después se comprueba si ya existe el fichero *csv*, y de no ser así, se crea con los nombres de las columnas (líneas 11-13).

A continuación, para cada fichero, se extraen los distintos **chunks** con la función generadora **file_reader('filename')** y, si los datos son cadenas de texto, solo incluye en la lista de filtros a utilizar por *Blosc* el **noshuffle** (líneas 15-25). Para cadenas de texto, la utilización de filtros de precompresión no es muy efectiva, por lo que se decidió descartar la utilización de filtros para ese caso. La función **file_reader('filename')** recorre los **DataSets** del fichero *HDF5* y si son mayores de 8 kilobytes los extrae. Se decidió ignorar tamaños inferiores debido a que en esos caso la compresión no es muy efectiva, la opción del tamaño de bloque pierde sentido y utilizar varios hilos tampoco resulta efectivo.

Para cada **chunk** se extraen sus propiedades estadísticas (líneas 25-33), en caso de ser datos de tipo cadena de texto se sustituyen por ceros y, posteriormente, se realizan las pruebas de compresión con las distintas opciones (líneas 34-45), registrando en un **DataFrame** de *pandas* la información extraída. Finalmente, se escriben los datos del **DataFrame** en fichero y se muestra información de progreso con respecto a la cantidad de **chunks** del **DataSet** (líneas 46-51).

Para la ejecución de este programa se utilizó una máquina remota, que permitió ejecutar todas las pruebas de compresión durante los fines de semana, evitando pausar el desarrollo del proyecto. Debido a que este programa fue modificado en varias ocasiones, mayoritariamente para añadir alguna propiedad estadística al *csv* y corregir fallos menores de formato del *csv*, la ejecución del mismo se lanzaba los viernes para así, el lunes siguiente, disponer de los datos actualizados.

4.2. Análisis de los datos

Una vez seleccionados los datos se decidió realizar un análisis de los mismos. Aunque es un proceso opcional, permite familiarizarse con ellos y, en nuestro caso, automatiza la generación de análisis bajo demanda para ficheros de datos concretos. A partir de aquí, se adoptó una metodología de desarrollo basada en **Jupyter Notebook**, una aplicación web que ha sido detallada en la sección 2.2.5. Con esta metodología se comenzó a trabajar con **notebooks**, los cuales son cuadernos *Jupyter* compuestos por bloques de texto y bloques de código con su correspondiente salida.

Con los datos del fichero *csv* comentado en la anterior sección, se comenzó a realizar la exploración y análisis de los mismos. Para ello se emplearon las siguientes librerías: **pandas**, que permitió trabajar con los datos del fichero *csv* cómodamente, gracias a la estructura **DataFrame** y, **matplotlib**, que facilitó la realización de gráficas sobre los datos. Antes de realizar el **notebook** final con los resultados del análisis se hicieron varios **notebooks** para familiarizarse con las librerías e ir definiendo los objetivos del análisis. Estos cuadernos se recopilaron para crear el **notebook** final que se encuentra en el Anexo B.2. Para evitar la sobrecarga de código dentro del cuaderno se recopiló todo el código, en forma de librería *Python*, en **custom_plots.py**

(ver Anexo B.1). Esta librería contiene funciones auxiliares para la extracción y visualización de los datos por medio de gráficas de **matplotlib** personalizadas. Los contenidos y resultados del análisis se detallan en la sección 5.1.

4.3. Transformación de los datos de entrada

Para que los datos extraídos pudiesen ser utilizados por las técnicas de clasificación supervisada de *scikit-learn* era necesario transformarlos. En primer lugar, al ser una clasificación de tipo supervisada, había que definir las opciones óptimas de compresión para cada **chunk** y para cada caso de uso a la hora de comprimir. Además, las variables categóricas de los datos se tienen que transformar en una variable binaria por categoría para que éstas puedan ser interpretadas por los algoritmos de clasificación. Se optó por utilizar un cuaderno *jupyter* para la transformación porque se querían realizar algunas comprobaciones sobre los datos resultantes. El cuaderno se puede encontrar en el Anexo C y los detalles de su contenido se encuentran en la sección 5.2.

4.4. Selección y aplicación de las técnicas de clasificación

Una vez los datos están preparados para aplicar las técnicas de clasificación supervisada, hay que decidir cuáles utilizar y con qué parámetros. Para escoger las técnicas más apropiadas al caso se utilizó el diagrama de la Fig. 4.2, el cual se extrajo de la documentación oficial de *scikit-learn* [7]. Siguiendo las recomendaciones del diagrama se escogieron los siguientes métodos: máquinas de vector soporte (**SVC**), vecinos más próximos (**KNeighborsClassifier**) y bosques aleatorios (un tipo de **EnsembleClassifier**). Adicionalmente, se utilizaron el análisis discriminante lineal y la regresión multinomial.

En cada algoritmo de clasificación había que decidir los parámetros a utilizar. Con dicho fin, se estudiaron los algoritmos, tanto a nivel teórico, como de la documentación ofrecida por *scikit-learn*. Además, para evaluar la calidad de los algoritmos de clasificación había que decidir qué métricas de puntuación utilizar.

Al tratarse de un problema de clasificación de múltiples variables (**multioutput-mutliclass**), las métricas de puntuación ofrecidas por *scikit-learn* no estaban adaptadas. Por ello se definieron nuestras propias métricas de puntuación en **scoring_functions.py** (ver Anexo D.2). Para especificar una métrica de puntuación, se define una función que recibe un algoritmo de clasificación entrenado, la muestra con las características X y las variables objetivo y; y devuelve la puntuación asociada.

Por otro lado, partiendo de la clase **MultiOutputClassifier** de *scikit-learn*, que permite

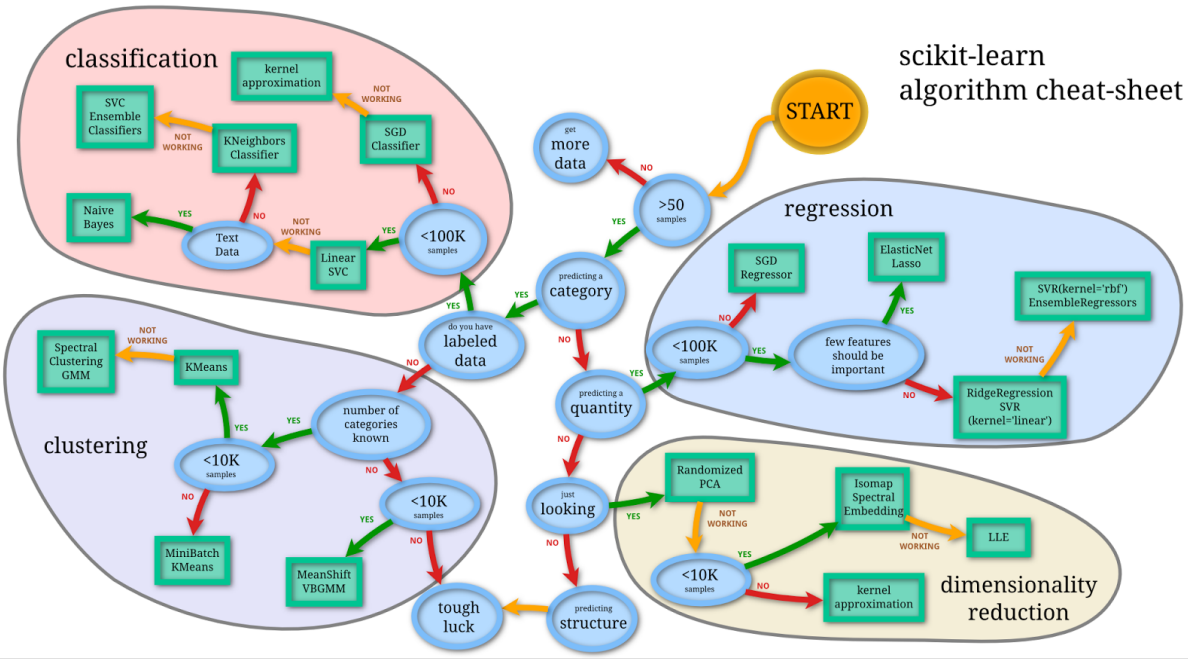


Figura 4.2: Diagrama para escoger algoritmos de *scikir-learn*

que algoritmos que no están pensados para la clasificación de tipo **multioutput-multiclass** puedan ser utilizados, se definió la clase **ChainedMultiOutputClassifier** (ver Anexo D.3). Esta clase, en lugar de predecir las distintas variables objetivo que componen y independientemente, las estima secuencialmente utilizando las variables objetivo ya estimadas para la siguiente predicción.

La validación de los parámetros se hizo utilizando la herramienta **GridSearch** de *scikit-learn*, que permite probar las distintas combinaciones de parámetros y escoge la mejor en base a una métrica de puntuación. La validación por **GridSearch** se realizó en el programa **nested_hypertuner.py** (ver Anexo D.4) y, además, se realizaron curvas de validación (ver sección 5.3.1) para validar los parámetros que no afectaban al resto.

4.5. Interpretación y evaluación de las técnicas de clasificación

Finalmente, con las técnicas de clasificación supervisada ya seleccionadas, se procede con la interpretación y evaluación de las mismas. Con este fin se realizaron varios **notebooks**, cuatro

para evaluar de forma general cada técnica y uno con la comparativa entre todos los algoritmos de clasificación. Para estos **notebooks**, se crearon funciones auxiliares en **ml_plots.py** (ver Anexo D.1), que servían para mostrar gráficas y tablas de puntuación sin cargar los **notebooks** de código. Los contenidos de estos cuadernos *jupyter* se detallan en la sección 5.3 y se pueden encontrar en el Anexo D.

Capítulo 5

Análisis de los Resultados

En este capítulo se presentan y discuten los métodos utilizados para el análisis de los resultados de los diferentes procedimientos. Los informes realizados a destacar son: el análisis de los datos de pruebas de compresión, la generación de los datos de entrenamiento, el análisis de cada algoritmo clasificador de aprendizaje automatizado, y finalmente la comparativa de algoritmos clasificadores.

5.1. Análisis de los datos de pruebas de compresión

El objetivo de este análisis es describir en términos generales el comportamiento de las pruebas de compresión a grandes rasgos. Aunque el informe presentado se ha realizado para todo el conjunto de ficheros, este análisis se puede filtrar por fichero y conjuntos de datos específicos del mismo. Esto puede resultar útil para el estudio en detalle de un tipo de fichero en particular. Este análisis se encuentra en el Anexo B.

Los objetivos de este análisis son:

- Relacionar el tamaño de bloque con las medidas de compresión y decompresión.
- Comprobar el comportamiento de los niveles de compresión sobre las pruebas.
- Comparar los datos de compresión de tablas por filas y por columnas.
- Determinar si existe correlación entre los algoritmos compresores *BloscLZ* o *LZ4* con nivel de compresión 1 y el resto.

- Observar si existe correlación directa entre las características extraídas de cada conjunto de datos y las medidas de compresión y descompresión.

Tamaño de bloque

El tamaño de bloque es una de las opciones de *Blosc* que determina el tamaño del bloque de datos que utilizará a nivel interno. A grandes rasgos un tamaño inferior implica que los datos caben en la caché del procesador y por tanto *Blosc* trabaje sobre ellos más rápido; por el contrario cuando el tamaño aumenta los datos no caben en caché y se ralentiza el proceso. Además, cuánto más grande es el tamaño de bloque significa también que los algoritmos compresores buscan redundancias en espacios mayores por lo que consiguen mayor compresión.

Para observar el comportamiento del tamaño de bloque se hacen las gráficas de las medias de las medidas de compresión: ratio de compresión, velocidad de compresión y velocidad de descompresión; para cada tamaño de bloque, para códec y filtro con nivel de compresión cinco (Ver pág. 90-96 Anexo B).

A partir de estas gráficas se comprueba que: el nivel de compresión es proporcional al tamaño de bloque, la velocidad de compresión se beneficia más de un tamaño de bloque pequeño o intermedio, dependiendo del algoritmo compresor, y la velocidad de descompresión es inversamente proporcional al tamaño de bloque. Además parece que los algoritmos compresores *Snappy* y *Zlib* tienen peores resultados que el resto.

Finalmente se comparan estos gráficos en 4 niveles distintos de compresión, se observa que el comportamiento es el esperado y que el tamaño de bloque automático aumenta con el nivel de compresión, que es lo esperado.

Nivel de compresión

El nivel de compresión de *Blosc* permite regular la compresión del algoritmo compresor. Aunque no todos los códecs tienen las mismas escalas, en *Blosc* se han adaptado todas en un rango del 1 al 9.

Como en el anterior caso se realizan gráficas de las medias de las medidas de compresión para cada nivel de compresión y para cada codec y filtro con tamaño de bloque fijo a 256 KB (Ver pág. 96-100 Anexo B).

El comportamiento observado en las gráficas es el esperado salvo para *Zstd* que, en los niveles de compresión más altos, pierde ratio de compresión. Además, vuelve a destacar el rendimiento

de *Snappy* y *Zlib*, que se sitúa por debajo del de sus competidores, *LZ4* y *Zstd*, respectivamente.

Tablas por columnas

A la hora de trabajar con tablas o vectores multidimensionales los datos se han estructurado de dos maneras: por filas o por columnas. Sea n el número de datos y k el número de columnas o dimensiones de los datos, los datos almacenados en HDF5 normalmente tienen la forma (n, k); sin embargo al almacenarlos por columnas (k, n) se aumenta la redundancia puesto que los datos de una misma columna se parecen más entre sí. Esto se tuvo en cuenta a la hora de generar los datos de las pruebas de compresión y, en el caso que los datos, eran tablas se hicieron las pruebas para su forma original y para la forma traspuesta.

Por tanto se realizó una comparativa de estas tablas (Ver pág. 100-101 Anexo B), en ella se observa que, aunque las velocidades de compresión y descompresión no varían, para el ratio de compresión sí que hay diferencias significativas. El ratio de compresión se llega a doblar en algunos casos para las tablas normales y aunque en algunos casos no hay mejora, nunca empeora.

Correlaciones entre BloscLZ, LZ4 y el resto

Considerando características de los datos, se consideró utilizar los códecs de *Blosc* más rápidos, *BloscLZ* y *LZ4* para calcular su ratio y velocidad de compresión y utilizar éstos como caracterización de la complejidad de compresión de los datos. Aunque parece obvio que debe de haber correlación con el resto de algoritmos compresores, tampoco es trivial, pues cada algoritmo tiene diferencias significativas entre ellos. Para confirmar la correlación se extrajeron los datos de cada códec y filtro con tamaño de bloque fijo y con niveles de compresión uno y nueve; posteriormente se calcularon los coeficientes de Pearson entre los códecs más rápidos y el resto. El coeficiente de Pearson entre dos muestras se define como:

$$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}},$$

donde X e Y serían las muestras, x_i es una observación de la muestra X y $\bar{x}$ es la media de X. Este coeficiente expresa la correlación lineal entre las muestras, en escala entre -1 y 1 interpretándose respectivamente como correlación lineal negativa y positiva. Cuánto más cercano a cero significa que no hay correlación.

En las gráficas realizadas (Ver pág. 102-104 Anexo B) se observa que las correlaciones en el ratio de compresión son muy buenas, por encima de 0,7 en datos numéricos y de 0,5 en cadenas

de texto para el códec *BloscLZ*. Por otro lado en las velocidades de compresión la correlación no es tan fuerte, por encima de 0,6 en datos numéricos y por encima de 0,3 en texto. Aún así se considera buen resultado dado que indica una ligera correlación y serán los algoritmos clasificadores los que determinen finalmente si esta medida caracteriza bien o no los datos. Finalmente destacar que con *LZ4* los datos son muy parecidos e incluso un poco mejores.

Correlaciones entre características de los datos y pruebas de compresión

Teniendo en cuenta el funcionamiento general de los algoritmos compresores, básicamente, se buscan datos parecidos y repetidos para comprimir esa información. Se escogieron características estadísticas de los datos que informasen sobre la cantidad de repeticiones y variabilidad; éstas fueron: la media, la desviación típica, el coeficiente de asimetría, el coeficiente de apuntamiento, el mínimo, el máximo, los tres cuartiles y el número de rachas (se define una racha como valores consecutivos que se encuentran por encima o por debajo de la mediana).

Con el objetivo de poder observar correlaciones entre estas características y las medidas de compresión se realizaron gráficas de pares enfrentándolas. En el caso de mínimos y cuartiles se utilizó el rango total y el rango intercuartílico. Para estas gráficas se filtró con el códec *LZ4*, filtro *shuffle*, nivel de compresión 5 y tamaño de bloque de 256 KB (Ver pág. 104-105 Anexo B); de no ser así la variabilidad arruinaría los gráficos.

Aunque en algunas características como los coeficientes de apuntamiento y asimetría, parecen estar bastante correlacionados con el ratio de compresión, del resto de propiedades no se pueden extraer conclusiones claras. Por esto se dejará que sean los propios algoritmos de aprendizaje automatizado los que informen de la influencia e importancia de estas características.

5.2. Generación de los datos de entrenamiento

En esta sección se detalla el proceso seguido para la generación de los datos de entrenamiento. El objetivo del aprendizaje automatizado es que, a partir de las características de los datos y las opciones que decida el usuario, el algoritmo escoja la opción más adecuada de códec, filtro, nivel de compresión y tamaño de bloque. Por tanto para cada fragmento de datos (*chunk*), que queda identificado por: el nombre de fichero, el conjunto de datos dentro del fichero y el número de *chunk* dentro del conjunto; se tiene que escoger de entre todas las pruebas realizadas sobre él, las más adecuadas a las opciones de usuario.

Opciones de usuario

Cada usuario tiene unas necesidades especificas a la hora de utilizar el compresor *Blosc*. Por ejemplo, las necesidades más comúnes serían obtener la máxima compresión posible, la máxima velocidad de compresión o la máxima velocidad de descompresión. Sin embargo, el usuario puede necesitar opciones más complejas, como por ejemplo máxima velocidad con un mínimo de compresión o buscar máxima compresión y velocidad de descompresión. Además para asegurar que las opciones son viables, se eliminaron todos los datos en los que el ratio de compresión era inferior a 1,1, es decir, se espera comprimir al menos un 10 % y, si no es posible, lo más conveniente será no comprimir.

Para escoger las opciones que no son simplemente el máximo de un valor se plantea la siguiente función de distancia para cada prueba de compresión realizada :

$$f(x, y, z) = C_1(x - x_{max})^2 + C_2(y - y_{max})^2 + C_3(z - z_{max})^2 \tag{5.1}$$

donde x, y, z son respectivamente los valores del ratio de compresión, de la velocidad de compresión y de la velocidad de descompresión, mientras que los máximos son los mayores valores de cada métrica dentro del *chunk*. Mientras tanto los coeficientes C_1, C_2, C_3 toman los valores 0 o 1 según si se le da o no importancia a la métrica que le corresponde. Como cada métrica sigue unas escalas, para evitar que las de mayor escala influyan más en el valor de la función se normalizan todas las métricas aplicando:

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}}.$$

De esta manera quedan todas dentro del rango $[0, 1]$ y la fórmula de distancia (5.1) se convierte en:

$$f(x', y', z') = C_1(x' - 1)^2 + C_2(y' - 1)^2 + C_3(z' - 1)^2 \tag{5.2}$$

donde x', y', z' son los valores de las métricas normalizados.

Ahora solo queda extraer el mínimo de dicha función dentro de todas las pruebas realizadas en el *chunk* para las 7 combinaciones posibles de los valores de C_1, C_2 y C_3 (serían 8 combinaciones, pero la opción (0, 0, 0) no tiene sentido). Por ejemplo: para los coeficientes (0, 1, 0) que significan que al usuario solo le interesa la velocidad de compresión, la función es $f(y') = (y' - 1)^2$ y buscar el mínimo es equivalente a escoger el máximo. Para los coeficientes (1, 1, 1), que significan que al usuario le importan todas las métricas por igual, la función es $f(x', y', z') = (x' - 1)^2 + (y' - 1)^2 + (z' - 1)^2$ y minimizarla implica escoger la opción con las métricas más equilibradas.

Una vez extraído el mínimo de la función solo hay que extraer el códec, filtro, nivel de compresión y tamaño de bloque que serán las variables objetivo para ese *chunk* con las opciones

de usuario correspondientes que son los coeficientes C_1, C_2 y C_3. Estos coeficientes se añadirán como las opciones de usuario tras renombrarlos como **IN_CR**, **IN_CS** y **IN_DS**. En la Tabla 5.1 se aprecian la codificación de las distintas opciones y su descripción.

Codificación			
IN_CR	**IN_CS**	**IN_DS**	**Descripción**
1	0	0	Máximo ratio de compresión.
0	1	0	Máxima velocidad de compresión.
0	0	1	Máxima velocidad de descompresión.
0	1	1	Equilibrio entre velocidad de compresión y descompresión.
1	1	0	Equilibrio entre ratio y velocidad de compresión.
1	0	1	Equilibrio entre ratio de compresión y velocidad de descompresión.
1	1	1	Equilibrio entre las tres medidas.

Tabla 5.1: Opciones de usuario y su codificación.

Por tanto cada fila de los datos de entrenamiento estará formada por: las variables identificativas de *chunk*, las opciones de usuario, las características de *chunk* y las variables objetivo. Además se añaden también como características las métricas de compresión obtenidas con *BloscLZ* y *LZ*4 con nivel de compresión uno. En la Fig. 5.1 se recopila la información guardada en cada fila de los datos de entrenamiento.

Figura 5.1: Esquema de las filas de los datos de entrenamiento

Eliminación de *Zlib* y *Snappy*

Tras haber extraído las opciones óptimas para cada opción se realizaron comprobaciones sobre las variables objetivo seleccionadas. Rápidamente se detectó que el códec *Zlib* nunca era seleccionado, pues como se había detectado con anterioridad, *Zstd* le ganaba en todas las ocasiones. Por otro lado *Snappy* solo era seleccionado 2 veces y se le consideró como un dato atípico. Por tanto, después de observar que la opción justo por detrás de *Snappy* era muy parecida,

se eliminó. Finalmente considerando 4 códecs se observó que aparecían seleccionadas 488 combinaciones posibles de un total de 1080 (combinaciones de códec, filtro, nivel de compresión y tamaño de bloque) que, sin contar el tamaño de bloque (puesto que tiene correlación con el nivel de compresión) son 97 combinaciones de 108, por lo que los datos están bastante completos y equilibrados.

Tamaño de bloque automático

A continuación se comprobó cuántas veces se correspondía el tamaño de bloque automático con el de la mejor opción, lo cual se producía en un 27 % de las ocasiones, nada mal, teniendo en cuenta que es un algoritmo simple que se basa apenas en el nivel de compresión seleccionado, y cuyo rango de tamaños es más limitado que el considerado en esta ocasión. Tras esta comprobación se cambian los valores del tamaño de bloque automático por el tamaño de bloque real utilizado, para ello basta buscar el tamaño de bloque con el mismo ratio de compresión.

Procesamiento de las variables categóricas

Los algoritmos de clasificación supervisada de *scikit-learn* sólo admiten variables binarias. Por ello, hay que procesar las variables categóricas para que puedan ser utilizadas. Con respecto a las características de *chunk* se transforman las variables:

- **Table**: que era una categórica de 3 valores, 0 para datos en forma de vector, 1 para tablas y 2 para tablas por columnas. Esta variable se transforma en dos binarias: **is_Table** indicando si es una tabla o no, e **is_Columnar** indicando si está orientada por columnas o no.
- **DType**: que indica el tipo de los datos y su tamaño, se divide en 4 variables. Tres variables binarias **is_Float**, **is_Int** e **is_String**, indicando respectivamente si son datos de coma flotante, enteros o cadenas de texto. La última variable es numérica **Type_Size** y se corresponde con el tamaño del tipo en bytes.

Finalmente las variables objetivo se transforman en variables binarias de la siguiente manera:

- **Codec**: se transforma en 4 variables binarias una para cada códec: **BloscLZ**, **LZ4**, **LZ4HC** y **Zstd**.
- **Filter**: una variable para cada filtro: **Noshuffle**, **Shuffle** y **Bitshuffle**.

- **CL**: para cada nivel se hace una variable binaria: **CL1**, **CL2**, **CL3**, **CL4**, **CL5**, **CL6**, **CL7**, **CL8** y **CL9**.
- **Block_Size**: al igual que en la anterior se transforma en 9 variables binarias: **Block_8**, **Block_16**, **Block_32**, **Block_64**, **Block_128**, **Block_256**, **Block_512**, **Block_1024** y **Block_2048**.

5.3. Análisis de los algoritmos de clasificación

Para realizar el análisis final de los algoritmos de clasificación se efectuaron dos tipos de análisis. El primer tipo de análisis consiste en comprobar el funcionamiento del algoritmo de clasificación y ajustar los parámetros que ofrece *scikit-learn*. Se realizaron cuatro informes de este tipo: uno para análisis discriminante y regresión multinomial, otro para máquinas de vector soporte, un tercero para vecinos más próximos y finalmente otro para bosques aleatorios. Dado que son muy parecidos entre sí, solo se detallará el procedimiento seguido con los bosques aleatorios, ya que es el informe más completo; el resto se pueden encontrar en el Anexo D.

El segundo tipo de análisis se corresponde con la comparativa final del rendimiento de los algoritmos. En él se compara tanto las puntuaciones obtenidas como las velocidades de predicción.

5.3.1. Análisis individual: Bosques aleatorios

En este análisis se exploran las opciones que ofrece el paquete *scikit-learn* para el algoritmo de aprendizaje automático de bosques aleatorios, que dentro del paquete se llama *RandomForestClassifier*, en adelante **RFC**. El informe se compone de cuatro partes que se detallarán en las próximas secciones y son: las curvas de aprendizaje, las curvas de validación, la validación cruzada de parámetros y los resultados obtenidos.

Curvas de aprendizaje

La curva de aprendizaje muestra la puntuación obtenida por la muestra de entrenamiento y por la de validación en función del tamaño de muestra de entrenamiento escogido. Esta herramienta permite saber si el algoritmo se beneficiará de aumentar el tamaño de la muestra de entrenamiento y si sufrirá problemas de **sobreajuste** o **sesgo**.

El **sobreajuste** (**variance** en inglés) es el error introducido en el algoritmo debido a fluctuaciones en la muestra de entrenamiento (por ejemplo: datos atípicos o muestras homogéneas) y provoca que el algoritmo se ajuste a esas fluctuaciones y falle al predecir otros datos. Cuando se produce **sobreajuste**, la puntuación con la muestra de entrenamiento es alta y con la muestra de validación es baja.

El **sesgo** (**bias** en inglés) es el error introducido por suposiciones del algoritmo de aprendizaje (por ejemplo: uso de distancias en datos no normalizados o suponer que los datos siguen cierta distribución cuando no es así) y provoca que no se ajuste para nada a los datos y falle. Esto

se detecta cuando las puntuaciones con la muestra de entrenamiento y con la de validación son bajas.

En la Fig. 5.2, podemos observar las curvas de entrenamiento para RFC. Simplemente destaca que en este caso el algoritmo se beneficia al aumentar el tamaño de la muestra. Sin embargo, se observa que hay cierto sobreajuste ya que la puntuación obtenida con la muestra de entrenamiento es casi perfecta y, sin embargo, la obtenida con la muestra de validación no consigue pasar de $0,8$.

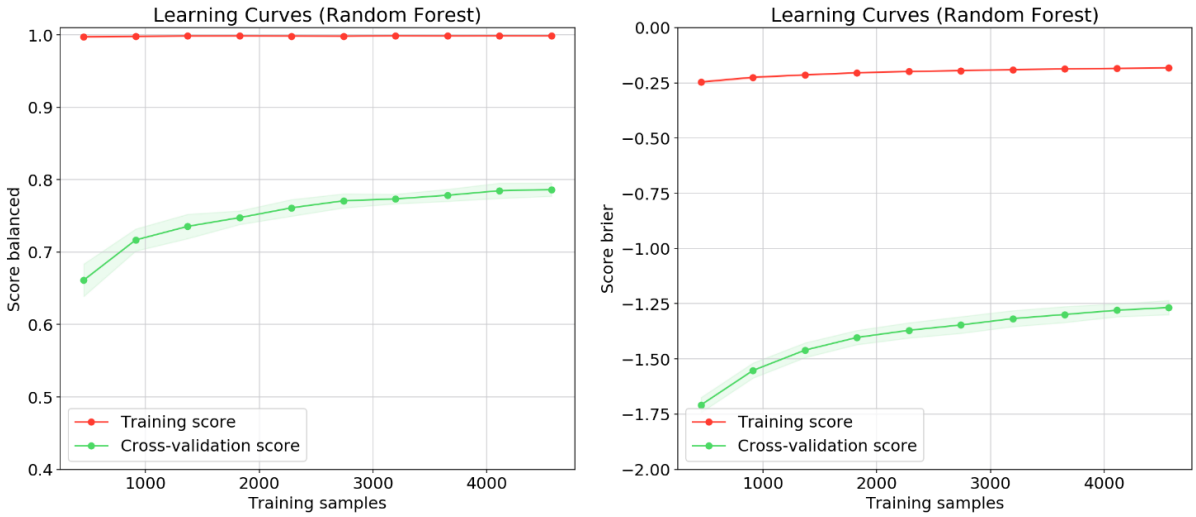


Figura 5.2: Curvas de entrenamiento de RFC con la puntuación personalizada a la izquierda y la de Brier a la derecha. En rojo la puntuación con la muestra de entrenamiento y en verde la puntuación con la muestra de validación.

Curvas de validación

La curva de validación representa las puntuaciones de las muestras de entrenamiento y validación en función de un parámetro del algoritmo de aprendizaje. Aunque para validar los parámetros del algoritmo se utilice la validación cruzada con *GridSearch*, una herramienta de *scikit-learn* que se detalla en la siguiente sección, en ocasiones resulta interesante utilizar curvas de validación en parámetros continuos para estimar el sobreajuste y el sesgo que producen sobre el algoritmo.

En la Fig. 5.3 se observan las curvas de validación para el parámetro **max_depth** de RFC.

Este parámetro, como se comentó en la Sección 3.5.2, sirve para cortar, o más bien limitar el tamaño de los arboles de decisión dentro del bosque pues, al alcanzar el valor de **max_depth** los árboles no se siguen ramificando. En la gráfica se observa que, para valores pequeños de **max_depth**, se produce sesgo y conforme aumenta el valor se produce sobreajuste. Para evitarlo se escogerá un valor intermedio como 10 o 12.

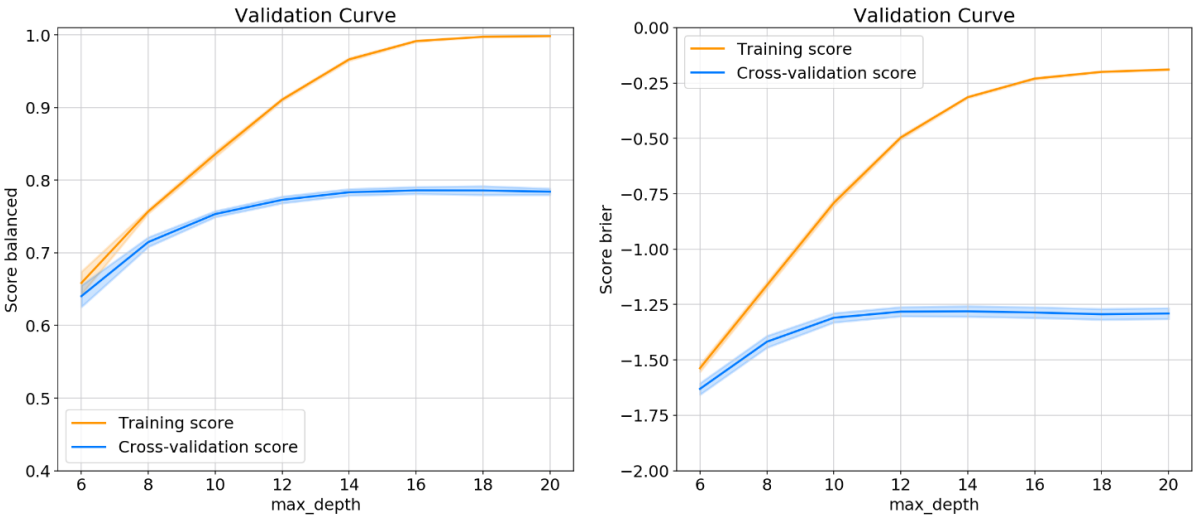


Figura 5.3: Curvas de validación de RFC para el parámetro *max_depth* con la puntuación personalizada a la izquierda y la de Brier a la derecha. En amarillo la puntuación para la muestra de entrenamiento y en azul la puntuación para la muestra de validación.

Además del parámetro **max_depth**, también se han realizado curvas de validación para **n_estimators** y **max_features**, que son respectivamente el número de árboles dentro del bosque y el número de características a considerar como mínimo para ramificar. Las curvas de estos parámetros son crecientes pero no mejoran apenas por lo que se escogieron valores pequeños para evitar un aumento en el tiempo de predicción y entrenamiento del algoritmo. Para **n_estimators** se escogió 20 y para **max_features** 10.

Validación cruzada de parámetros

La mejor manera de validar los parámetros de un algoritmo en *scikit-learn* es mediante la herramienta **GridSearch**. Para utilizarla hay que proporcionarle el nombre de los parámetros y el rango de valores que se quieren probar en cada uno. De esta manera **GridSearch** prueba todas las combinaciones posibles hasta dar con la óptima. Para encontrar esta métrica óptima

hay que especificarle la métrica de puntuación a utilizar y las divisiones de las muestra de entrenamiento y validación a emplear, para las cuales se utilizó **ShuffleSplit**, otra herramienta de *scikit-learn* que realiza las divisiones de forma aleatoria. Ya que la validación de todas las combinaciones de parámetros puede resultar costosa, se realizó en un programa *Python* aparte (ver Anexo D), guardando los resultados para ser mostrados posteriormente en el informe.

En el informe (ver Anexo D.8) se muestran los resultados de la validación cruzada que en el caso de RFC se utilizó sobre tres parámetros: **criterion** (valores *gini* o *entropy*), **bootstrap** (valores *True* o *False*) y **class_weight** (valores *None* o *Balanced*). El **criterion** es la regla de ramificación a seguir por los árboles de decisión; **bootstrap** indica si utiliza o no la técnica de muestreo para entrenar los árboles, y **class_weight** permite asignar pesos a las variables objetivo según la cantidad de veces que aparecen en la muestra. Se realizó la validación cruzada para las dos métricas propuestas (Brier y la personalizada), resultando que cada una decidía un **criterion** y **bootstrap** distinto. Por tanto, para desempatar se utilizó la métrica por defecto de *scikit-learn* que resultó ser mayor para los parámetros escogidos por la métrica personalizada.

En la Fig. 5.4 se muestran las puntuaciones de la validación cruzada, tanto por la métrica personalizada como por la de Brier. Finalmente se escogieron los parámetros de la personalizada: **criterion** *entropy*, **bootstrap** *False* y **class_weight** *None*.

Validación con métrica personalizada
Report

Name	Score
balanced	0.8196 +/-(0.0072)
brier	-1.2470 +/-(0.0435)
normal	0.2987 +/-(0.0140)
codec	0.9508 +/-(0.0041)
filter_	0.9503 +/-(0.0064)
codec_filter	0.9131 +/-(0.0075)
c_level	0.4917 +/-(0.0176) ~ 0.7143 +/-(0.0151)
block	0.4611 +/-(0.0155) ~ 0.7395 +/-(0.0054)
cl_block	0.3029 +/-(0.0135) ~ 0.7277 +/-(0.0096)

Validación con la métrica de Brier
Report

Name	Score
balanced	0.8062 +/-(0.0067)
brier	-1.2402 +/-(0.0332)
normal	0.2775 +/-(0.0136)
codec	0.9490 +/-(0.0056)
filter_	0.9482 +/-(0.0054)
codec_filter	0.9104 +/-(0.0077)
c_level	0.4638 +/-(0.0182) ~ 0.6922 +/-(0.0131)
block	0.4350 +/-(0.0142) ~ 0.7132 +/-(0.0096)
cl_block	0.2795 +/-(0.0158) ~ 0.7017 +/-(0.0096)

Figura 5.4: Comparativa de puntuaciones para las dos validaciones de parámetros. Balanced es la puntuación personalizada, normal la predeterminada de *scikit-learn*. El resto son puntuaciones separadas por variable objetivo.

Selección de características

En la Fig. 5.4 se encuentran las puntuaciones obtenidas con el algoritmo RFC utilizando todas las características. En esta sección se utiliza RFC para la selección de características, puesto que para el algoritmo final interesa extraer únicamente las necesarias para no ralentizar el proceso de predicción. En la Fig. 5.5 se pueden observar las características ordenadas por su influencia en el algoritmo RFC. De entre estas se escogieron las que tenían puntuación mayor a $0,02$ y se eliminaron **BLZ_DSpeed**, **LZ4_CRate**, **LZ4_CSpeed** y **LZ4_DSpeed**. Las características de velocidad de descompresión se eliminaron para no cargar la extracción del ratio y la velocidad de compresión con otro espacio de memoria para la descompresión, y las de *LZ4* porque es más fácil adaptar el código de *BloscLZ*. Hay que destacar que, aunque las características de *LZ4* y *BloscLZ* tenga mucha importancia en el algoritmo, al estar muy correlacionadas, cuando se elimina una, la otra *"hereda"* el peso de la eliminada.

Ranking de características

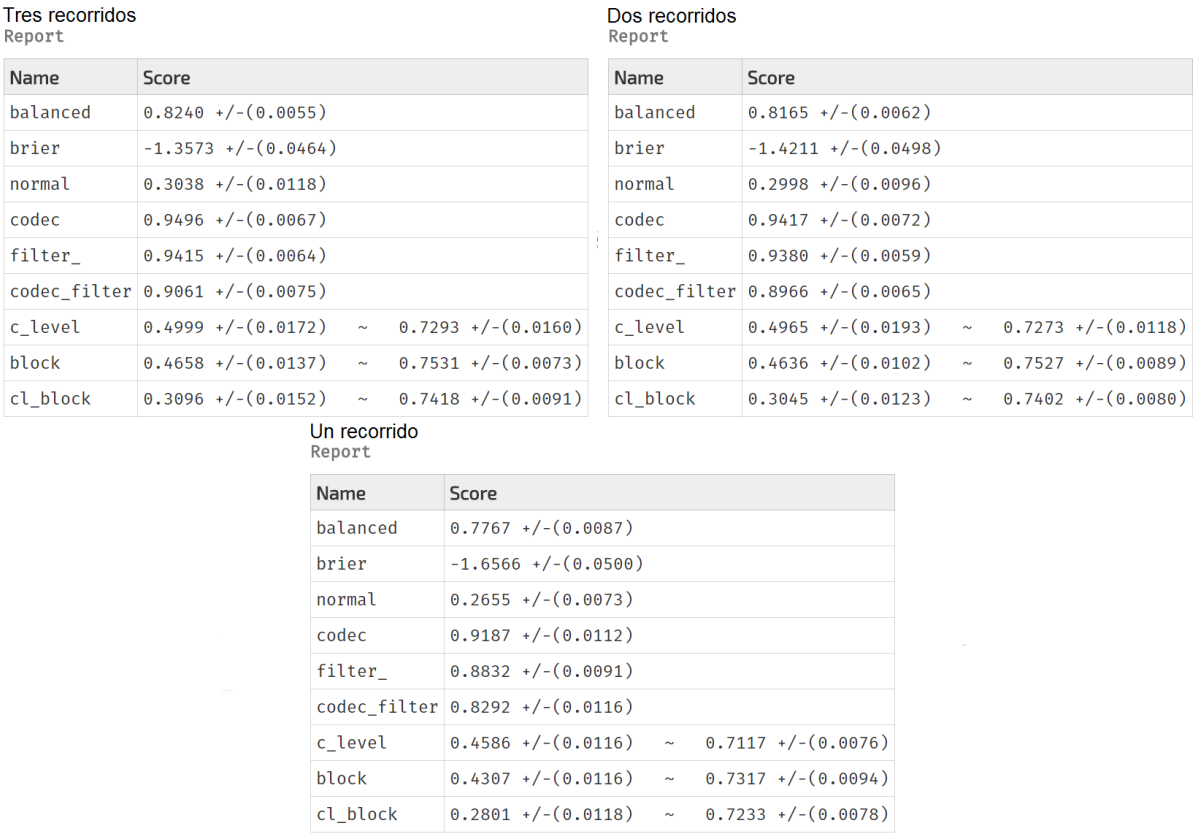
```
IN_CS        --> 0.137211464106
IN_CR        --> 0.131111625479
IN_DS        --> 0.0974040195938
Sd           --> 0.0725884746762
BLZ_CRate    --> 0.0650706591705
LZ4_CRate    --> 0.0577350273716
LZ4_CSpeed   --> 0.0478638587605
N_Streaks    --> 0.0472820616428
BLZ_CSpeed   --> 0.0406333720582
LZ4_DSpeed   --> 0.0396800663834
Max          --> 0.0344295790066
Mean         --> 0.0342350747248
Kurt         --> 0.0313547351925
BLZ_DSpeed   --> 0.0285255993956
Skew         --> 0.027954199065
Min          --> 0.0210788529098
Q3           --> 0.0190268628242
is_Float     --> 0.0155081537992
is_Int       --> 0.0133159625712
Q1           --> 0.0100059230634
Median       --> 0.00909992360608
Chunk_Size   --> 0.0089115679826
Type_Size    --> 0.00504329884318
is_String    --> 0.00199125161123
is_Table     --> 0.00159127362086
is_Columnar  --> 0.00134711254228
```

Figura 5.5: Ranking de características según el algoritmo RFC.

Debido a que una de las principales características de *Blosc* es la velocidad, el tiempo de extracción de las características no puede ser excesivamente alto. Por ello se consideraron tres conjuntos de características según las veces que era necesario recorrer los datos para extraerlas. En la Tabla 5.2 se observan las características añadidas en cada recorrido.

Primer recorrido					
IN_CR	IN_CS	IN_DS	BLZ_CRate	BLZ_CSpeed	
Segundo recorrido					
Sd	Mean	Kurt	Skew	Max	Min
Tercer recorrido					
N_Streaks					

Tabla 5.2: Características por recorrido.



Tres recorridos
Report

Name	Score
balanced	0.8240 +/-(0.0055)
brier	-1.3573 +/-(0.0464)
normal	0.3038 +/-(0.0118)
codec	0.9496 +/-(0.0067)
filter_	0.9415 +/-(0.0064)
codec_filter	0.9061 +/-(0.0075)
c_level	0.4999 +/-(0.0172) ~ 0.7293 +/-(0.0160)
block	0.4658 +/-(0.0137) ~ 0.7531 +/-(0.0073)
cl_block	0.3096 +/-(0.0152) ~ 0.7418 +/-(0.0091)

Dos recorridos
Report

Name	Score
balanced	0.8165 +/-(0.0062)
brier	-1.4211 +/-(0.0498)
normal	0.2998 +/-(0.0096)
codec	0.9417 +/-(0.0072)
filter_	0.9380 +/-(0.0059)
codec_filter	0.8966 +/-(0.0065)
c_level	0.4965 +/-(0.0193) ~ 0.7273 +/-(0.0118)
block	0.4636 +/-(0.0102) ~ 0.7527 +/-(0.0089)
cl_block	0.3045 +/-(0.0123) ~ 0.7402 +/-(0.0080)

Un recorrido
Report

Name	Score
balanced	0.7767 +/-(0.0087)
brier	-1.6566 +/-(0.0500)
normal	0.2655 +/-(0.0073)
codec	0.9187 +/-(0.0112)
filter_	0.8832 +/-(0.0091)
codec_filter	0.8292 +/-(0.0116)
c_level	0.4586 +/-(0.0116) ~ 0.7117 +/-(0.0076)
block	0.4307 +/-(0.0116) ~ 0.7317 +/-(0.0094)
cl_block	0.2801 +/-(0.0118) ~ 0.7233 +/-(0.0078)

Figura 5.6: Puntuaciones por cantidad de recorridos.

En la Fig. 5.6 se muestran las puntuaciones obtenidas con cada uno de los 3 conjuntos de características. Para el caso de RFC la diferencia entre dos o tres recorridos no es muy significativa. Sin embargo, con un único recorrido ya se pierde un $0,03$ en la puntuación personalizada y la por defecto. Destacar que también se probó como conjunto de primer recorrido el formado por **IN_CR**, **IN_CS**, **IN_DS** y las características del segundo recorrido; pero las puntuaciones eran mucho peores, algo esperable teniendo en cuenta que las del primer recorrido tienen una posición más alta en el ranking de características.

Finalmente se utilizó una variante de la herramienta **MultiOutputClassifier** proporcionada por *scikit-learn*. Esta variante se hizo a partir del programa original de *scikit-learn*, *multioutput.py* [10]. La variante **multioutput_chained.py** se encuentra en el Anexo D y permite utilizar el objeto **ChainedMultiOutputClassifier**, que al pasarle un clasificador como RFC, se encarga de que las predicciones de cada variable objetivo se utilicen para predecir la siguiente, de forma que, si éstas tienen correlación, los resultados mejoran. En la Fig. 5.7 se observan los resultados para el conjunto de características de tres recorridos, aunque los resultados mejoran significativamente hay que tener en cuenta que las predicciones son más lentas.

ChainedMultiOutput

Report

Name	Score
balanced	0.8518 +/-(0.0047)
brier	0.0000 +/-(0.0000)
normal	0.3385 +/-(0.0091)
codec	0.9532 +/-(0.0047)
filter_	0.9504 +/-(0.0060)
codec_filter	0.9123 +/-(0.0076)
c_level	0.5563 +/-(0.0147) ~ 0.7867 +/-(0.0095)
block	0.5107 +/-(0.0059) ~ 0.8002 +/-(0.0072)
cl_block	0.3432 +/-(0.0111) ~ 0.7920 +/-(0.0053)

Figura 5.7: Puntuaciones con **ChainedMultiOutputClassifier**, la puntuación de Brier es cero porque no se implementaron las predicciones probabilísticas

5.3.2. Comparativa de algoritmos

Tras haber estudiado individualmente cada algoritmo se realizó un informe donde se compara el rendimiento de los distintos algoritmos empleados. Para ello se realizan comparativas de las puntuaciones en distintos casos de uso y las velocidades de predicción de los distintos algoritmos.

Comparativa por fichero

En esta primera comparación se escogen todos los datos menos los de un fichero como muestra de entrenamiento y el resto, los de un sólo fichero, como muestra de validación. En la Fig. 5.8 se muestran las puntuaciones de cada algoritmo para cada fichero. Como se puede observar no son nada esperanzadoras, y es que cada fichero es un mundo aparte.

Cada fichero tiene un número diferente de conjuntos de datos (**DataSets**), que a su vez se dividen en fragmentos (**chunks**) de 16 MB, o menos si son más pequeños, sobre los que se realizan las pruebas. Como comprobaremos en las siguientes secciones, la clave está en los **DataSets**, pues son los que cargan con la mayor variabilidad entre las clasificaciones. Por ejemplo un **DataSet** puede contener datos de la temperatura en una ciudad, que son números pequeños y fáciles de comprimir, mientras que otro puede contener información de rayos cósmicos, que son números muy grandes y difíciles de comprimir. Sin embargo, en un fichero pueden haber 50 **DataSets** o tan sólo uno como el caso del fichero *GSSTF_NCEP*, el cuál consigue las mejores puntuaciones.

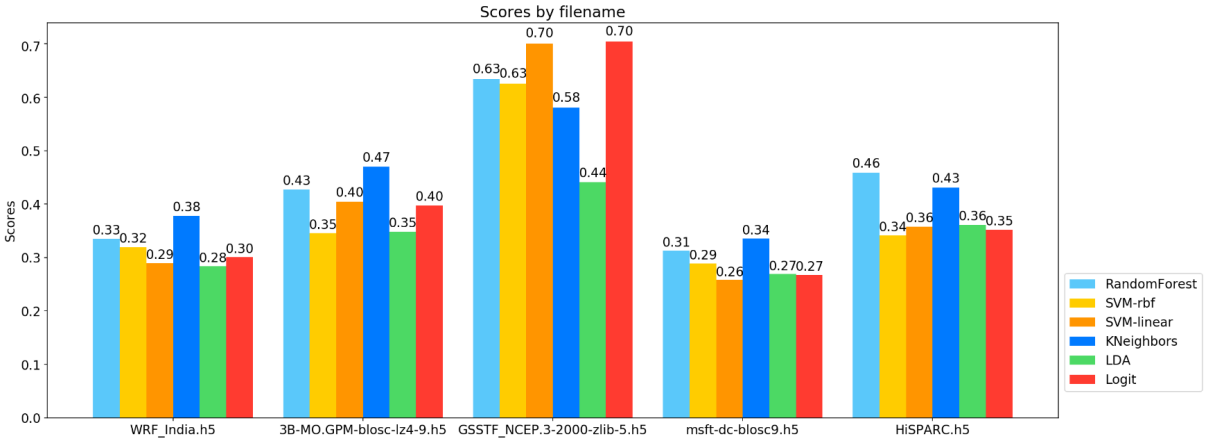


Figura 5.8: Puntuaciones de la comparativa por fichero. *RandomForest* son los bosques aleatorios, *SVM* máquinas de vector soporte lineales o con la función de base radial gaussiana, *KNeighbors* vecinos más próximos, *LDA* análisis discriminante y *Logit* regresión multinomial.

Comparativa por grupos de DataSets

Para esta comparativa se mezclan los **DataSets** por igual y se dividen en 5 grupos; después se coge un grupo como muestra de validación y el resto como muestra de entrenamiento. Esta comparativa pretende ilustrar el comportamiento que tendría el algoritmo en casos genéricos

fuera de la muestra de entrenamiento, si se realizase un estudio para escoger cuidadosamente los **DataSets** de entrenamiento. En la Fig. 5.9 se muestran las puntuaciones obtenidas, donde se aprecia que los algoritmos más precisos son bosques aleatorios y vecinos más próximos. Sin embargo, conviene tener en cuenta la anterior sección, donde aunque solo había un único caso, los ganadores eran máquinas de vector soporte lineales y regresión logística, por lo que en caso de ampliar el estudio se deberían seguir considerando.

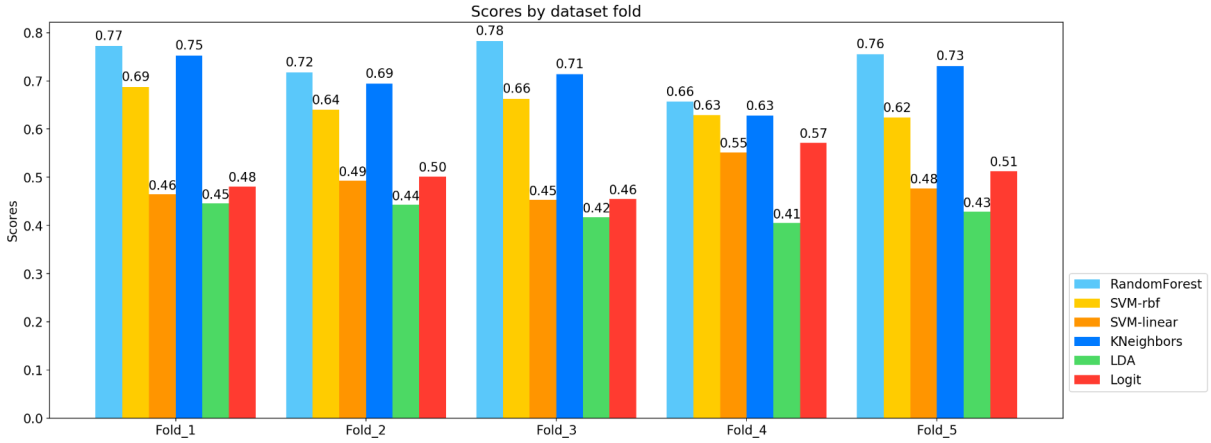


Figura 5.9: Puntuaciones de la comparativa por grupos equilibrados de **DataSets**.

Para finalizar en la Fig. 5.10 se muestra en detalle un desglose de las puntuaciones por variables objetivo: **Codec**, **Filter**, **CL** y **Block_Size** para el tercer grupo; además de la puntuación al utilizar el **ChainedMultiOutputClassifier**. Conviene tener en cuenta la enorme mejora que supone aplicar las predicciones de forma encadenada en el análisis discriminante, la regresión logística y las máquinas de vector soporte, donde se aprecian mejoras en las puntuaciones de $0,1$ e incluso $0,2$ para análisis discriminante.

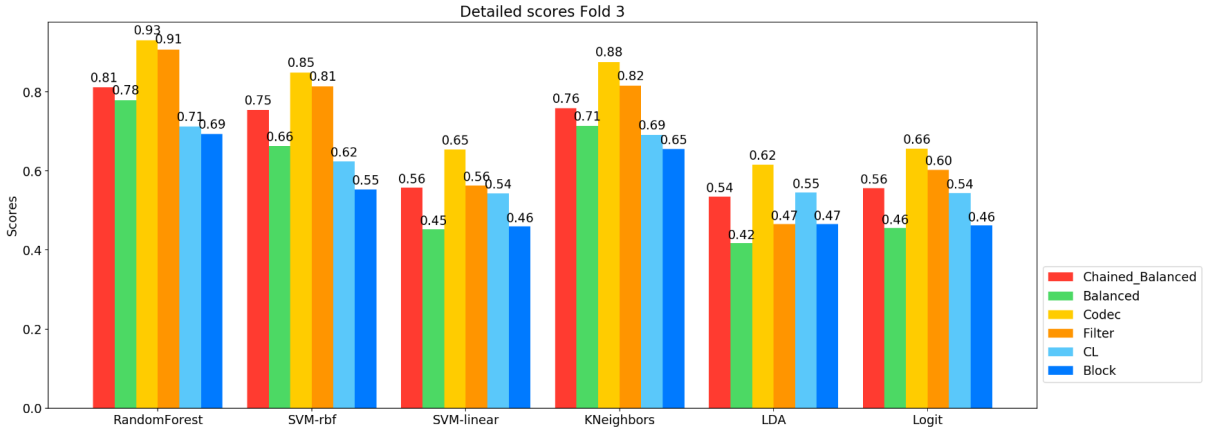


Figura 5.10: Puntuaciones detalladas para el tercer grupo.

Comparativa por *chunks*

Después de evaluar la precisión de los algoritmos sobre datos muy distintos de los de entrenamiento, se plantea estudiar la precisión cuando se sabe que los datos a predecir son parecidos a los de entrenamiento. En concreto interesa saber si, a partir de un primer *chunk*, se pueden realizar buenas predicciones del resto de *chunks* de un **DataSet**. De esta manera se puede estimar lo bien que se comportaría el algoritmo al aplicarlo en flujos de datos. En la Fig. 5.11 se observan los resultados de entrenar los algoritmos con el primero del **DataSet** y validarlo con el resto de *chunks*. El claro ganador para este caso son los bosques aleatorios. Aun así también destacan las puntuaciones de las máquinas de vector soporte y vecinos más próximos.

Para la última comparativa de precisión se realizó una división equilibrada de *chunks*. Ésta es muy parecida a la realizada con los grupos de **DataSets** pero con la particularidad de que se escogen muy pocos datos de entrenamiento. Para ello de todos los **DataSets** se escogen únicamente los formados por un único *chunk* y éstos se dividen por la mitad para entrenar y validar. En la Fig. 5.12 se muestran los resultados, donde los claros ganadores vuelven a ser bosques aleatorios, vecinos más próximos y máquinas de vector soporte.

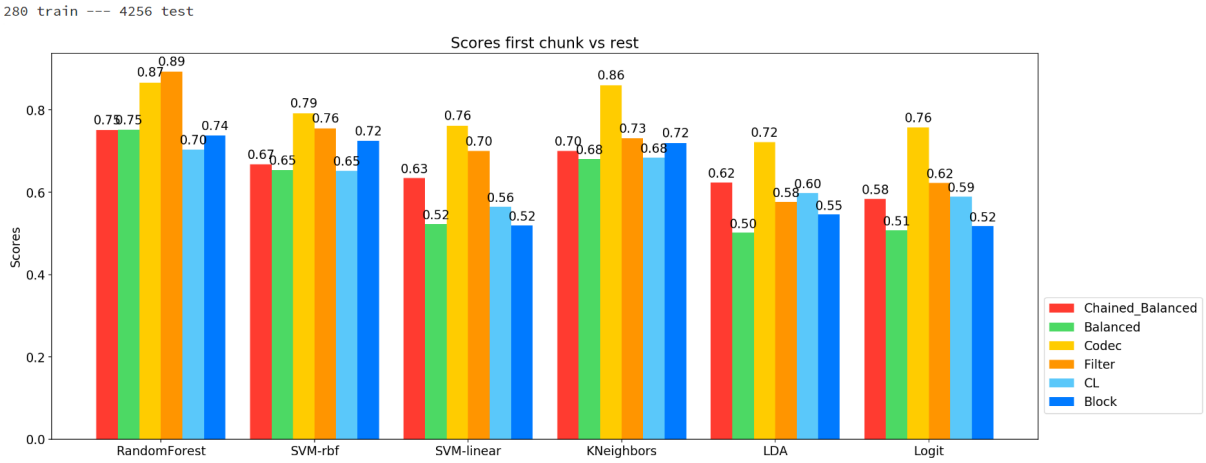


Figura 5.11: Puntuaciones detalladas para el primer *chunk* contra el resto.

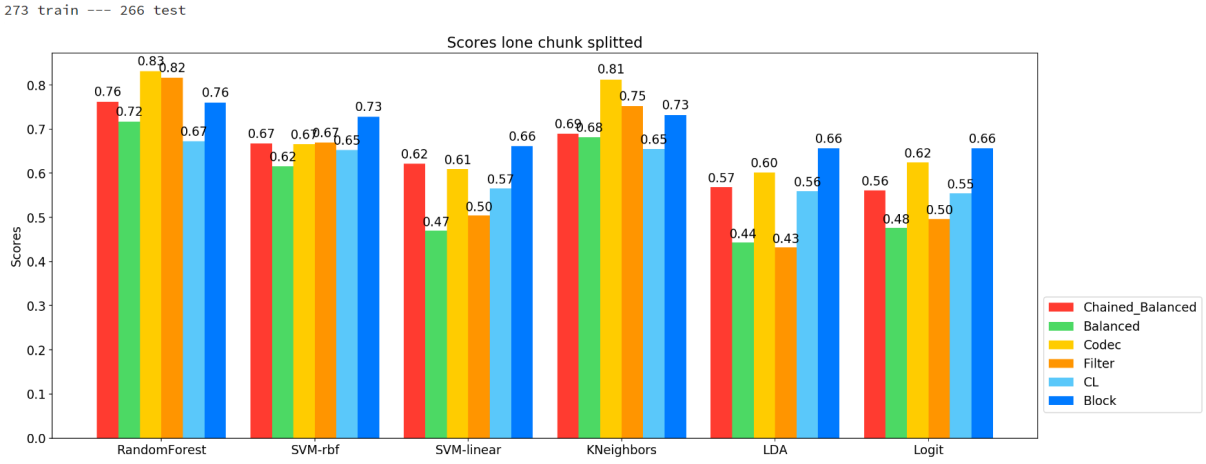


Figura 5.12: Puntuaciones detalladas para la mitad de *chunks*.

Comparativa de velocidad

Finalmente se realiza una prueba de velocidad de predicción, donde simplemente se mide el tiempo medio de una predicción para cada algoritmo. Como se observa en la Fig. 5.13, el más lento con diferencia son los bosques aleatorios, seguido de máquinas de vector soporte con función de base radial Gaussiana. Por otro lado máquinas de vector soporte lineales, análisis discriminante y regresión multinomial están igualados; y el ganador es vecinos más próximos.

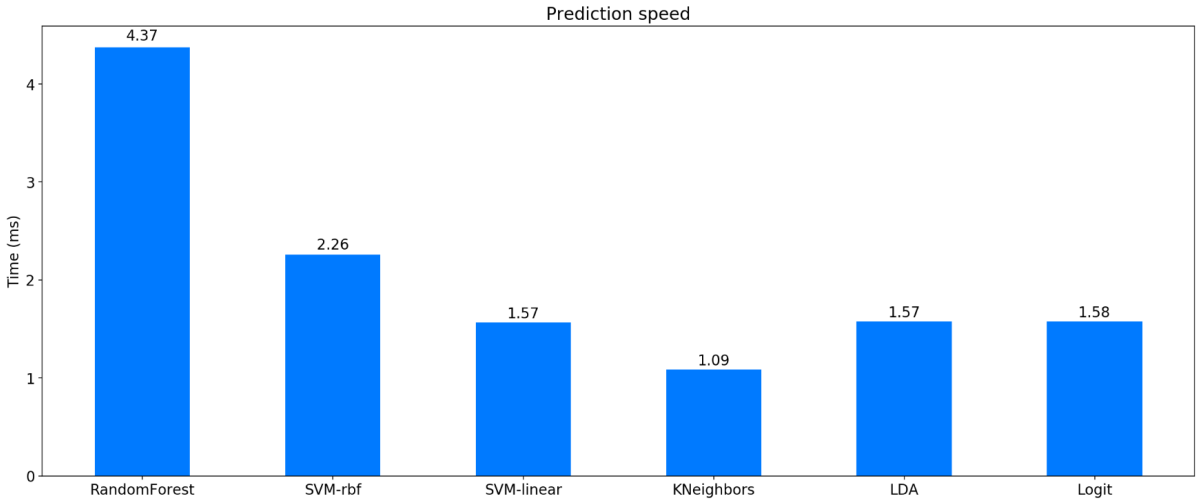


Figura 5.13: Puntuaciones detalladas para el tercer grupo.

Capítulo 6

Conclusiones

El objetivo principal del proyecto era desarrollar un algoritmo de aprendizaje automatizado que fuese capaz de determinar las opciones (**códec**, **filtro**, **nivel de compresión** y **tamaño de bloque**) del meta compresor de datos *Blosc*, según las características de los datos a comprimir y el objetivo del usuario.

Para ello se ha desarrollado un proyecto de minería de datos que ha implicado desde la generación de los datos hasta la construcción de algoritmos de clasificación supervisada y su comparación de rendimiento en distintos casos de uso. Para poder llevar a cabo estas tareas ha sido necesario un estudio teórico previo de las técnicas de clasificación supervisada, así como la práctica de las mismas, para poder escoger de entre todas ellas las más adecuadas al caso. Los algoritmos de clasificación utilizados han sido análisis discriminante, regresión multinomial, máquinas de vector soporte, vecinos más próximos y bosques aleatorios, que se han detallado en el Capítulo 3.

Con respecto a los casos de uso a considerar en el desarrollo del algoritmo clasificador se han considerado dos: el primero un algoritmo genérico para cualquier tipo de datos y el segundo un algoritmo ajustado a un tipo de datos en concreto. Para ambos casos habría que continuar con el estudio iniciado en este proyecto para terminar de refinar y producir el algoritmo final a incluir dentro de *Blosc*.

Con respecto al primer caso de uso, los ficheros que se disponen actualmente para generar datos de pruebas de compresión son insuficientes. Cuando lo que se pretende caracterizar es cualquier tipo de datos en sí, la variabilidad es casi infinita. En este estudio la mayoría de los datos eran numéricos pero también pueden aparecer datos en forma de texto por ejemplo. Para proseguir dentro de este caso de uso, habría que decidir el conjunto de características a extraer de los datos e implementar dentro de *Blosc* la extracción para poder aplicarla de forma

efectiva sobre cualquier tipo de datos a nivel de byte. Por el momento se han considerado tres conjuntos de características los cuales se ha demostrado que funcionan relativamente bien en el Capítulo 6. Una vez implementado en *Blosc* la extracción de características, habría que ampliar la cantidad de ficheros de datos y realizar un estudio en profundidad de los tipos de conjuntos de datos, probablemente con técnicas de clasificación no supervisada. Finalmente habría que utilizar los resultados de este estudio para construir una muestra de datos de entrenamiento lo más equilibrada posible y comparar el rendimiento de los algoritmos de clasificación ya estudiados.

Por otro lado en el caso de ajustar el algoritmo a un tipo en concreto de datos, ya se ha demostrado en el Capítulo 6 que la precisión es bastante buena, entorno al 75 % con la puntuación personalizada y acertando el **códec** y **filtro** en aproximadamente un 90 % de los casos. Por tanto aquí se vislumbran dos posibilidades: producir algoritmos adaptados bajo demanda de clientes a un tipo en concreto de datos; o producir un sistema de compresión orientado a flujos de datos. El primer caso sería muy sencillo de realizar y se ha demostrado la eficiencia en este proyecto. En el segundo caso habría que continuar la investigación, tanto en la optimización del tiempo de predicción como en la elaboración de un sistema de entrenamiento continuo para el algoritmo clasificador.

Bibliografía

[1] M. A. Aizerman, E. A. Braverman y L. Rozonoer. «Theoretical foundations of the potential function method in pattern recognition learning.» En: *Automation and Remote Control,* Automation and Remote Control, 25. 1964, págs. 821-837.

[2] Francesc Alted. *Blosc benchmark example*. 2014. URL: https://github.com/Blosc/python-blosc/blob/master/bench/compress_ptr.py.

[3] L. Breiman. «Random Forests». En: *Machine Learning* 45 (octubre de 2001), págs. 5-32. DOI: 10.1023/A:1010933404324.

[4] L. Breiman y col. *Classification and Regression Trees*. The Wadsworth and Brooks-Cole statistics-probability series. Taylor & Francis, 1984. ISBN: 9780412048418. URL: https://books.google.es/books?id=JwQx-WOmSyQC.

[5] Glenn W. Brier. *Verification of forecasts expressed in terms of probability*. Enero de 1950. URL: https://docs.lib.noaa.gov/rescue/mwr/078/mwr-078-01-0001.pdf.

[6] Cisco. *The Zettabyte Era — Trends and Analysis*. Junio de 2016. URL: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html.

[7] scikit-learn developers. *Choosing the right estimator*. 2016. URL: http://scikit-learn.org/stable/tutorial/machine_learning_map/.

[8] scikit-learn developers. «Nested versus non-nested cross-validation». En: *scikit-learn user guide*. Diciembre de 2016, págs. 1010-1013. URL: http://scikit-learn.org/stable/_downloads/scikit-learn-docs.pdf.

[9] T. Hastie, R. Tibshirani y J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition*. Springer Series in Statistics. Springer New York, 2009. ISBN: 9780387848587. URL: https://books.google.es/books?id=tVIjmNS3Ob8C.

[10] Tim Head. *multioutput.py*. 2016. URL: https://github.com/scikit-learn/scikit-learn/blob/14031f6/sklearn/multioutput.py.

[11] Fengan Li y col. «When Lempel-Ziv-Welch Meets Machine Learning: A Case Study of Accelerating Machine Learning using Coding». En: *CoRR* abs/1702.06943 (2017). URL: http://arxiv.org/abs/1702.06943.

[12] Alistair Miles. *Genotype compressor benhcmark*. Septiembre de 2016. URL: http://alimanfoo.github.io/2016/09/21/genotype-compression-benchmark.html.

[13] D. Pena. *Analisis de datos multivariantes*. S.A. MCGRAW-HILL, 2002. ISBN: 9788448136109. URL: https://books.google.es/books?id=TrVlAAAACAAJ.

[14] Jonathan Whitmore. *Jupyter Notebook post save hook for .py and .html files*. 2015. URL: https://gist.github.com/jbwhit/881bdeeaae3e4128947c/.

[15] Jonathan Whitmore. *Jupyter Notebook Tips and Tricks*. 2015. URL: https://github.com/jbwhit/jupyter-tips-and-tricks.

Anexo A

Generador de datos de las pruebas de compresión

En este anexo se presenta el código realizado en *Python* que realiza las pruebas de compresión sobre los datos dentro de un fichero *hdf5* y escribe los resultados en un fichero *csv*. El fichero del código se llama **test_data_generator.py** que incluye el programa principal y las funciones auxiliares que utiliza.

A.1. Código test_data_generator.py

```
1  from __future__ import print_function
2  import functools
3  import os.path
4  from queue import Queue
5  from sys import platform
6  import blosc
7  import tables
8  import numpy as np
9  import scipy.stats as stats
10 import pandas as pd
11 if platform == 'win32':
12     from time import clock as time
13 else:
14     from time import time as time
15
16 SPEED_UNIT = 2**30
17 CHUNK_SIZE = 2**24
18 MINIMUM_SIZE = 2**13
19 KB32, KB128, MB = 2**15, 2**17, 2**20
```

```
KB64, KB256, MB2 = 2**16, 2**18, 2**21
KB16, KB512 = 2**14, 2**19


def test_codec(chunk, codec, filter_name, clevel):
    """
    Compress the chunk and return tested data.

    Parameters
    ----------
    chunk: bytes-like object (supporting the buffer interface)
        The data to be compressed.
    codec : string
        The name of the compressor used internally in Blosc. It can be
        any of the supported by Blosc ('blosclz', 'lz4', 'lz4hc',
        'snappy', 'zlib', 'zstd' and maybe others too).
    filter_name : int
        The shuffle filter to be activated.  Allowed values are
        blosc.NOSHUFFLE, blosc.SHUFFLE and blosc.BITSHUFFLE.
    clevel : int
        The compression level from 0 (no compression) to 9
        (maximum compression).
    Returns
    -------
    out: tuple
        The associated compression rate, compression speed and
        decompression speed (in GB/s).
    Raises
    ------
    TypeError
        If bytesobj doesn't support the buffer interface.
    ValueError
        If bytesobj is too long.
        If typesize is not within the allowed range.
        If clevel is not within the allowed range.
        If cname is not a valid codec.
    """
    t0 = time()
    c = blosc.compress_ptr(chunk.__array_interface__['data'][0],
                           chunk.size, chunk.dtype.itemsize,
                           clevel=clevel, shuffle=filter_name, cname=codec)
    tc = time() - t0
    out = np.empty(chunk.size, dtype=chunk.dtype)
    times = []
    for i in range(3):
        t0 = time()
        blosc.decompress_ptr(c, out.__array_interface__['data'][0])
        times.append(time() - t0)
    chunk_byte_size = chunk.size * chunk.dtype.itemsize
    rate = chunk_byte_size / len(c)
    c_speed = chunk_byte_size / tc / SPEED_UNIT
    d_speed = chunk_byte_size / min(times) / SPEED_UNIT
```

```
    # print("  *** %-8s, %-10s, CL%d *** %6.4f s / %5.4f s " %
    #       ( codec, blosc.filters[filter], clevel, tc, td), end='')
    # print("\tCompr. ratio: %5.1fx" %rate)
    return rate, c_speed, d_speed


def chunk_generator(buffer):
    """
    Generate data chunks of 16 MB in `buffer`.

    Parameters
    ----------
    buffer : numpy.array
        Buffer array of data

    Returns
    -------
    out : numpy.array
        A chunk of 16 MB extracted from the original buffer.
    """
    mega = int(CHUNK_SIZE / buffer.dtype.itemsize)
    max, r = divmod(buffer.size, mega)
    for i in range(max):
        yield buffer[i * mega: (i + 1) * mega]
    if r != 0:
        yield buffer[max * mega: buffer.size]


def dataset_extractor(dataset):
    """
    Extracts the data inside hdf5 `dataset` node.

    Parameters
    ----------
    dataset : dataset node
        An hdf5 dataset node

    Returns
    -------
    out : tuple
        A tuple with the path to the data, the dtype, a number 0 (not a table),
         1 (table) or 2 (column wise table) and the numpy array with the data.
    """
    if hasattr(dataset, 'colnames'):
        for col_name in dataset.colnames:
            col = dataset.col(col_name)
            if col.size * col.dtype.itemsize > MINIMUM_SIZE:
                yield dataset._v_pathname + '.' + col_name, col.dtype, 1, \
                    col[:].reshape(functools.reduce(
                        lambda x, y: x * y, col.shape))
                col_shape = dataset.description.__getattribute__(
                    col_name).shape
```

```
                if (len(col_shape) > 1 or
                        (len(col_shape) > 0 and col_shape[0] > 1)):
                    yield dataset._v_pathname + '.' + col_name, col.dtype, 2,
                    np.moveaxis(col, 0, -1)[:].reshape(functools.reduce(
                        lambda x, y: x * y, col.shape))
    else:
        yield dataset._v_pathname, dataset.dtype, 0,
        dataset[:].reshape(functools.reduce(lambda x, y: x * y, dataset.shape))


def file_reader(filename):
    """
    Generate the buffers of data in `filename`.

    Parameters
    ----------
    filename : str
        The name of the HDF5 file.

    Returns
    -------
    out : tuple
        A tuple with the path to the data, the dtype, a number 0 (not a table),
         1 (table) or 2 (column wise table) and the numpy array with the data.
    """
    with tables.open_file(filename) as f:
        group_queue = Queue()
        group_queue.put(f.root)
        while not group_queue.empty():
            try:
                for child in group_queue.get():
                    if hasattr(child, 'dtype'):
                        if child.size_in_memory > MINIMUM_SIZE:
                            yield from dataset_extractor(child)
                    elif hasattr(child, '_v_children'):
                        group_queue.put(child)
            except TypeError:
                pass
            except tables.HDF5ExtError:
                pass


def calculate_nchunks(type_size, buffer_size):
    """
    Calculate the number of chunks.

    Parameters
    ----------
    type_size : int
        The type size in bytes.
    buffer_size : int
        The buffer size in number of elements.
```

```
176
177     Returns
178     -------
179     out : int
180         The number of chunks associated with the buffer and chunk size.
181     """
182     chunks_aux = int(CHUNK_SIZE / type_size)
183     q, r = divmod(buffer_size, chunks_aux)
184     n_chunks = q
185     if r != 0:
186         n_chunks += 1
187     return n_chunks
188
189
190 def extract_chunk_features(chunk):
191     """
192     Extract the statistics features in `chunk`.
193
194     Parameters
195     ----------
196     chunk : numpy.array
197         An array of numbers.
198
199     Returns
200     -------
201     out : tuple
202         A tuple containing the mean, median, standard deviation, skewness,
203         kurtosis, minimum, maximum and quartiles.
204     """
205     if np.isnan(chunk).any():
206         return np.nanmean(chunk), np.nanmedian(chunk), np.nanstd(chunk),
207         stats.skew(chunk, nan_policy='omit'),
208         stats.kurtosis(chunk, nan_policy='omit'), np.nanmin(chunk),
209         np.nanmax(chunk),  np.nanpercentile(chunk, 25),
210         np.nanpercentile(chunk, 75)
211     else:
212         return np.mean(chunk), np.median(chunk), np.std(chunk),
213         stats.skew(chunk), stats.kurtosis(chunk), np.min(chunk), np.max(chunk),
214         np.percentile(chunk, 25), np.percentile(chunk, 75)
215
216
217 def calculate_streaks(chunk, median):
218     """
219     Calculate number of streaks.
220
221     Parameters
222     ----------
223     chunk : numpy.array
224         An array of numbers.
225     median : number
226         The median of the chunk.
227
```

```
    Returns
    -------
    out : int
        Number of streaks above/below median of the chunk.
    """
    streaks = 1
    above = chunk[0] > median
    for number in chunk[1:]:
        if above != (number > median):
            streaks += 1
            above = not above
    return streaks


FILENAMES = ('HiSPARC.h5',)
PATH = '/home/francesc/datasets/tests/'
BLOCK_SIZES = (0, MINIMUM_SIZE, KB16, KB32, KB64, KB128, KB256, KB512, MB, MB2)
C_LEVELS = range(1, 10)
COLS = ['Filename', 'DataSet', 'Table', 'DType', 'Chunk_Number', 'Chunk_Size',
        'Mean', 'Median', 'Sd', 'Skew', 'Kurt', 'Min', 'Max', 'Q1', 'Q3',
        'N_Streaks', 'Block_Size', 'Codec', 'Filter', 'CL', 'CRate', 'CSpeed',
        'DSpeed']
blosc.set_nthreads(4)

if not os.path.isfile('blosc_test_data.csv'):
    pd.DataFrame(columns=COLS).to_csv(
        'blosc_test_data.csv', sep='\t', index=False)

for filename in FILENAMES:
    for path, d_type, table, buffer in file_reader(PATH + filename):
        n_chunks = calculate_nchunks(buffer.dtype.itemsize, buffer.size)
        print("Starting tests with %s %s t%s" % (filename, path, table))
        if buffer.dtype.kind in ('S', 'U'):
            is_string = True
            filters = (blosc.NOSHUFFLE,)
        else:
            is_string = False
            filters = (blosc.NOSHUFFLE, blosc.SHUFFLE, blosc.BITSHUFFLE)
        for i, chunk in enumerate(chunk_generator(buffer)):
            chunk_id = (filename, path, table, d_type, i + 1,
                        chunk.size * chunk.dtype.itemsize / MB)
            if is_string:
                chunk_features = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
            else:
                chunk_features = extract_chunk_features(chunk)
                chunk_features += (calculate_streaks(chunk,
                                                     chunk_features[1]),)
            df = pd.DataFrame()
            for block_size in BLOCK_SIZES:
                blosc.set_blocksize(block_size)
                for codec in blosc.compressor_list():
                    for filt in filters:
```

```
                        for clevel in C_LEVELS:
                            row_data = chunk_id + chunk_features\
                                + (block_size / 2**10, codec,
                                    blosc.filters[filt], clevel)\
                                + test_codec(chunk, codec, filt, clevel)
                            df = df.append(
                                dict(zip(COLS, row_data)), ignore_index=True)
            print("%5.2f%%%s %s t%s chunk %d completed" %
                  ((i + 1) / n_chunks * 100, filename, path, table, (i + 1)))
            with open('blosc_test_data.csv', 'a') as f:
                df = df[COLS]
                df.to_csv(f, sep='\t', index=False, header=False)
            print('CHUNK WRITED')
```

Anexo B

Análisis descriptivo de las pruebas de compresión

El informe realizado de análisis descriptivo de los datos extraídos en las pruebas de compresión se hizo en *jupyter notebook*. Además para que el informe fuese más legible se extrajo la mayor parte del código a un modulo *Python* que contiene todas las funciones auxiliares utilizadas para realizar gráficas personalizadas. A continuación se presenta el modulo *Python* **custom_plots.py** con las funciones auxiliares mencionadas y el informe con el análisis descriptivo **blosc_test_analysis.ipynb**.

B.1. Código custom_plots.py

```
1
2  import itertools
3  import numpy as np
4  import matplotlib.pyplot as plt
5  import matplotlib.lines as mlines
6  from scipy.stats.stats import pearsonr
7
8  # DICTIONARIES FOR BUILDING PERSONALIZED GRAPHS
9  COLOR_PALETTE = {'blosclz': '#5AC8FA', 'lz4': '#4CD964', 'lz4hc': '#FF3B30',
10                  'snappy': '#FFCC00', 'zlib': '#FF9500', 'zstd': '#5856D6'}
11 MARKER_DICT = {'noshuffle': 'o', 'shuffle': 'v', 'bitshuffle': 's'}
12
13 # DIFFERENT COLUMN LISTS FROM THE DATAFRAME FOR SELECTING SPECIFIC INFO
14 COLS = ['Filename', 'DataSet', 'Table', 'DType', 'Chunk_Number', 'Chunk_Size',
15         'Mean', 'Median', 'Sd', 'Skew', 'Kurt', 'Min', 'Max', 'Q1', 'Q3',
16         'N_Streaks', 'Block_Size', 'Codec', 'Filter', 'CL', 'CRate', 'CSpeed',
17         'DSpeed']
```

```
DESC_SET = ['DataSet', 'DType', 'Table', 'Chunk_Size']
CHUNK_FEATURES = ['Chunk_Size', 'Mean', 'Median', 'Sd',
                  'Skew', 'Kurt', 'Min', 'Max', 'Q1', 'Q3', 'N_Streaks']
TEST_FEATURES = ['CRate', 'CSpeed', 'DSpeed']
ALL_FEATURES = CHUNK_FEATURES + TEST_FEATURES

# AUX LIST FOR SELECTING DATAFRAMES
TYPES = ['float', 'int', 'str']
BLOCK_SIZES = [8, 16, 32, 64, 128, 256, 512, 1024, 2048]

# PATH TO FIGURES
FIG_PATH = '../figures/Blosc-test-analysis-'


# AUX FUNCTIONS
def outlier_lim(data):
    """Return the outliers limits."""

    q1 = np.percentile(data, 25)
    q3 = np.percentile(data, 75)
    dif = q3 - q1

    return q1 - 1.5 * dif, q3 + 1.5 * dif, dif


# PLOTTING FUNCTIONS
def custom_boxplot(ax, y, title='', ylabel=''):
    """Customized boxplot style."""

    meanpointprops = dict(marker='D', markeredgecolor='black',
                          markerfacecolor='#FF3B30', markersize=8)
    medianprops = {'color': '#4CD964', 'linewidth': 3}
    boxprops = {'edgecolor': 'black', 'linestyle': '-', 'facecolor': '#EFEFF4'}
    whiskerprops = {'color': 'black', 'linestyle': '-'}
    capprops = {'color': 'black', 'linestyle': '-'}
    flierprops = {'color': 'black', 'marker': 'o'}

    ax.boxplot(y,
               medianprops=medianprops,
               boxprops=boxprops,
               whiskerprops=whiskerprops,
               capprops=capprops,
               flierprops=flierprops,
               meanprops=meanpointprops,
               meanline=False,
               showmeans=True,
               widths=0.5,
               patch_artist=True)

    ax.set_title(title)
    ax.set_ylabel(ylabel)

```

```
    return ax


def custom_centered_scatter(ax, x, y):
    """Customized scatter plot ignoring outliers."""

    x_lim = outlier_lim(x)
    y_lim = outlier_lim(y)
    x_, y_ = [], []
    for i in range(len(x)):
        if (x_lim[0] <= x.iloc[i] <= x_lim[1] and
                y_lim[0] <= y.iloc[i] <= y_lim[1]):
            x_.append(x.iloc[i])
            y_.append(y.iloc[i])
    ax.scatter(x_, y_, color='#007AFF', marker='.', linewidth=3)
    if x_lim[2] > 0 and y_lim[2] > 0:
        ax.set_xlim(x_lim[0:2])
        ax.set_ylim(y_lim[0:2])
        fit = np.polyfit(x_, y_, deg=1)
        x_ = np.sort(x_)
        ax.plot(x_, fit[0] * x_ + fit[1], color='#FF3B30', linewidth=3)
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)

    return ax


def custom_pearson_scatter(ax, x_, y_, c_, m_, title, dtype):
    """Customized scatter plot for correlations."""

    for x, y, c, m, size in zip(x_, y_, c_, m_, np.asarray(
            list((itertools.repeat([10, 16], len(c_))))).flatten()):
        ax.plot(x, y, alpha=0.8, c=c, marker=m, linewidth=3, markersize=size)
    ax.set_axisbelow(True)
    ax.yaxis.grid(color='#CECED2')
    ax.xaxis.grid(color='#CECED2')
    ax.set_xlabel('Pearson C.Rates')
    ax.set_ylabel('Pearson C.Speeds')
    ax.set_title(title + ' - PEARSON - ' + dtype.upper())
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)

    return ax


def custom_sc_legend(fig):
    """Customized legend for the correlation plots."""

    handles = [mlines.Line2D([], [], color=color, marker='o', linestyle=' ',
                             markersize=10, label=label)
               for label, color in COLOR_PALETTE.items()]
    handles += [mlines.Line2D([], [], color='k', marker='o', linestyle=' ',
```

```
                                markersize=10, label='NOSHUFFLE'),
                   mlines.Line2D([], [], color='k', marker='v', linestyle=' ',
                                markersize=10, label='SHUFFLE'),
                   mlines.Line2D([], [], color='k', marker='s', linestyle=' ',
                                markersize=10, label='BITSHUFFLE'),
                   mlines.Line2D([], [], color='k', marker='o', linestyle=' ',
                                markersize=10, label='Compression Level 1'),
                   mlines.Line2D([], [], color='k', marker='o', linestyle=' ',
                                markersize=16, label='Compression Level 9')]
    labels = [label for label in COLOR_PALETTE.keys()]
    labels += ['NOSHUFFLE', 'SHUFFLE', 'BITSHUFFLE',
               'Compression Level 1', 'Compression Level 9']
    fig.legend(handles=handles, labels=labels, loc='lower left',
               ncol=2, bbox_to_anchor=(1, 0.05))
    fig.tight_layout()

    return fig


def custom_lineplot_tests(ax, x, rates, c_speeds, d_speeds, title='',
                          cl_mode=False):
    """Customized line plot for blosc test data."""

    ax.plot(x, rates, color='#007AFF', marker='o', markersize=8, linewidth=3)
    ax.set_axisbelow(True)
    ax.yaxis.grid(color='#CECED2')
    ax.xaxis.grid(color='#CECED2')
    ax.set_ylabel('Compression Rate', color='#007AFF')
    ax.tick_params('y', colors='#007AFF')
    blue_line = mlines.Line2D([], [], color='#007AFF', marker='o',
                              markersize=8, label='Compression ratio')
    plt.legend(handles=[blue_line], loc=2, bbox_to_anchor=(0., 1.01, 0., .102))
    ax2 = ax.twinx()
    ax2.plot(x, c_speeds, color='#FF3B30',
             marker='o', markersize=8, linewidth=3)
    ax2.plot(x, d_speeds, color='#4CD964',
             marker='o', markersize=8, linewidth=3)
    ax2.set_ylabel('Speed (GB/s)', color='k')
    ax.set_yticks(np.linspace(ax.get_ybound()[0], ax.get_ybound()[1], 6))
    ax2.set_yticks(np.linspace(ax2.get_ybound()[0], ax2.get_ybound()[1], 6))
    red_line = mlines.Line2D([], [], color='#FF3B30', marker='o',
                             markersize=8, label='Compression')
    green_line = mlines.Line2D([], [], color='#4CD964', marker='o',
                               markersize=8, label='Decompression')
    plt.legend(handles=[red_line, green_line], loc=1,
               bbox_to_anchor=(0., 1.01, 1., .102))
    if not cl_mode:
        ax.set_xscale('log', basex=2)
        ax.set_xticks(x)
        ax.set_xticklabels(['Auto', '8K', '16K', '32K', '64K',
                            '128K', '256K', '512K', '1MB', '2MB'])
        ax.set_xlabel('Block Size')
```

```
174     else:
175         ax.set_xticks(x)
176         ax.set_xlabel('Compression level')
177     ax.set_title(title)
178     ax.spines['top'].set_visible(False)
179     ax2.spines['top'].set_visible(False)
180 
181     return ax
182 
183 
184 def boxplot_data_builder(df):
185     """Build data for the boxplots."""
186 
187     rates = []
188     c_speeds = []
189     d_speeds = []
190     indices = []
191     for i in range(3):
192         if i == 2:
193             dfaux = df[df.DType.str.contains('U') | df.DType.str.contains('S')]
194         else:
195             dfaux = df[df.DType.str.contains(TYPES[i])]
196         if dfaux.size > 0:
197             rates.append([dfaux['CRate']])
198             c_speeds.append([dfaux['CSpeed']])
199             d_speeds.append([dfaux['DSpeed']])
200             indices.append(i)
201     return rates, c_speeds, d_speeds, indices
202 
203 
204 def paint_dtype_boxplots(df):
205     """Paint boxplots structured by dtype."""
206 
207     rates, c_speeds, d_speeds, indices = boxplot_data_builder(df)
208     n = len(rates)
209     if n > 1:
210         fig = plt.figure(figsize=(20, 24))
211         for i in range(n * 3):
212             aux = 300 + n * 10 + i + 1
213             pos = i %n
214             ax = fig.add_subplot(aux)
215             if i < n:
216                 custom_boxplot(ax, rates[pos],
217                                'C.Rates-' + TYPES[indices[pos]],
218                                'Compression Rates')
219             elif i < n * 2:
220                 custom_boxplot(ax, c_speeds[pos],
221                                'C.Speeds-' + TYPES[indices[pos]],
222                                'Compression Speeds (GB/s)')
223             else:
224                 custom_boxplot(ax, d_speeds[pos],
225                                'D.Speeds-' + TYPES[indices[pos]],
```

```
                                    'Decompression Speeds (GB/s)')
    else:
        fig = plt.figure(figsize=(20, 8))
        custom_boxplot(fig.add_subplot(131), rates[0],
                       'C.Rates-' + TYPES[indices[0]], 'Compression Rates')
        custom_boxplot(fig.add_subplot(132), c_speeds[0],
                       'C.Speeds-' + TYPES[indices[0]],
                       'Compression Speeds (GB/s)')
        custom_boxplot(fig.add_subplot(133), d_speeds[0],
                       'D.Speeds-' + TYPES[indices[0]],
                       'Decompression Speeds (GB/s)')
    fig.suptitle('Test features boxplots')
    plt.savefig(FIG_PATH + 'Test features boxplots' +
                '.png', bbox_inches='tight')


def block_cor_data_builder(df, onlystr, cl_mode):
    """Build data for the block correlation graphs"""

    rates = []
    c_speeds = []
    d_speeds = []
    indices = []
    block_values = [0] + BLOCK_SIZES
    if onlystr:
        options = [2]
    else:
        options = range(3)
    for i in options:
        if i == 2:
            dfaux = df[df.DType.str.contains('U') | df.DType.str.contains('S')]
        else:
            dfaux = df[df.DType.str.contains(TYPES[i])]
        if dfaux.size > 0:
            if not cl_mode:
                rates.append([dfaux[dfaux.Block_Size == size]
                              ['CRate'].mean() for size in block_values])
                c_speeds.append([dfaux[dfaux.Block_Size == size]
                                 ['CSpeed'].mean() for size in block_values])
                d_speeds.append([dfaux[dfaux.Block_Size == size]
                                 ['DSpeed'].mean() for size in block_values])
                indices.append(i)
            else:
                rates.append([dfaux[dfaux.CL == cl]['CRate'].mean()
                              for cl in list(range(10))[1:]])
                c_speeds.append([dfaux[dfaux.CL == cl]['CSpeed'].mean()
                                 for cl in list(range(10))[1:]])
                d_speeds.append([dfaux[dfaux.CL == cl]['DSpeed'].mean()
                                 for cl in list(range(10))[1:]])
                indices.append(i)

    return rates, c_speeds, d_speeds, indices
```

```
278
279
280 def paint_block_cor(df, title='', onlystr=False, cl_mode=False):
281     """Paint custom lineplots structured by dtype."""
282
283     rates, c_speeds, d_speeds, indices = block_cor_data_builder(
284         df, onlystr, cl_mode)
285     if not cl_mode:
286         x = [1] + BLOCK_SIZES
287     else:
288         x = list(range(10))[1:]
289     n = len(rates)
290     fig = plt.figure(figsize=(20, 8))
291     sup_title = 'Block Size'
292     if cl_mode:
293         sup_title = 'Compression Level'
294     fig.suptitle(sup_title + ' comparison with ' + title, fontsize=16)
295     for i in range(n):
296         pos = 100 + n * 10 + i + 1
297         ax = fig.add_subplot(pos)
298         custom_lineplot_tests(ax, x, rates=rates[i], c_speeds=c_speeds[i],
299                               d_speeds=d_speeds[i],
300                               title='dtype - ' + TYPES[indices[i]],
301                               cl_mode=cl_mode)
302     if n > 1:
303         fig.tight_layout()
304     plt.subplots_adjust(top=0.85)
305     plt.savefig(FIG_PATH + sup_title + ' comparison with ' +
306                 title + '.png', bbox_inches='tight')
307
308
309 def paint_all_block_cor(df, filter_name, c_level=5, cl_mode=False,
310                         block_size=0):
311     """Paint all custom lineplots for filter."""
312
313     if filter_name == 'noshuffle':
314         onlystr = True
315     else:
316         onlystr = False
317     for codec in df.drop_duplicates(subset=['Codec'])['Codec']:
318         if codec == 'blosclz' and filter_name == 'shuffle':
319             if not cl_mode:
320                 paint_block_cor(df[(df.CL == c_level) & (df.Codec == codec) &
321                                    (df.Filter == 'bitshuffle')],
322                                 codec.upper() + '-BITSHUFFLE-CL' +
323                                 str(c_level), onlystr, cl_mode)
324             else:
325                 paint_block_cor(
326                     df[(df.Block_Size == block_size) & (
327                         df.Codec == codec) & (df.Filter == 'bitshuffle')],
328                     codec.upper() + '-BITSHUFFLE-BLOCK' + str(block_size),
329                     onlystr, cl_mode)
```

```
330         if not cl_mode:
331             paint_block_cor(df[(df.CL == c_level) & (df.Codec == codec) &
332                                (df.Filter == filter_name)],
333                             codec.upper() + '-' + filter_name.upper() + '-CL' +
334                             str(c_level), onlystr, cl_mode)
335         else:
336             paint_block_cor(
337                 df[(df.Block_Size == block_size) & (
338                     df.Codec == codec) & (df.Filter == filter_name)],
339                 codec.upper() + '-' + filter_name.upper() + '-BLOCK' +
340                 str(block_size), onlystr, cl_mode)
341
342
343 def paint_cl_comparison(df, filter_name, codec):
344     """Paint custom plots comparing compression levels and block sizes."""
345
346     data = []
347     c_levels = [1, 3, 6, 9]
348     for c_level in c_levels:
349         data.append(block_cor_data_builder(df[(df.CL == c_level) &
350                                               (df.Codec == codec) &
351                                               (df.Filter == filter_name)],
352                                            False, False))
353     block_sizes = [1] + BLOCK_SIZES
354     n = len(data[0][0])
355     for i in range(n):
356         fig = plt.figure(figsize=(20, 16))
357         fig.suptitle('Compression Level and block size comparison ' +
358                      codec.upper() + '-' +
359                      TYPES[data[0][3][i]].upper(), fontsize=16)
360         for j in range(4):
361             pos = 200 + 20 + j + 1
362             ax = fig.add_subplot(pos)
363             custom_lineplot_tests(ax, block_sizes, data[j][0][i],
364                                   data[j][1][i], data[j][2][i],
365                                   title='C-Level ' + str(c_levels[j]))
366         if n > 1:
367             fig.tight_layout()
368         plt.subplots_adjust(top=0.9, hspace=0.2)
369         plt.savefig(FIG_PATH + 'Compression Level and block size comparison ' +
370                     codec.upper() + '-' +
371                     TYPES[data[0][3][i]].upper() + '.png', bbox_inches='tight')
372
373
374 def pearson_cor_data_builder(df, cname, clevel):
375     """Build data for codec correlation graphs."""
376
377     pearson_rates = [[], [], []]
378     pearson_c_speeds = [[], [], []]
379     codecs_cl = [[], [], []]
380     colors = [[], [], []]
381     markers = [[], [], []]
```

```
    for codec in df.drop_duplicates(subset=['Codec'])['Codec']:
        for filt in ['noshuffle', 'shuffle', 'bitshuffle']:
            df_blz1 = df[(df.Codec == cname) & (df.CL == clevel)
                         & (df.Filter == 'noshuffle')]
            for c_level in [1, 9]:
                df_codec = df[(df.Codec == codec) & (
                    df.CL == c_level) & (df.Filter == filt)]
                for i in range(3):
                    if i == 2:
                        dfaux = df_codec[df_codec.DType.str.contains(
                            'U') | df_codec.DType.str.contains('S')]
                        df_blz_aux = df_blz1[df_blz1.DType.str.contains(
                            'U') | df_blz1.DType.str.contains('S')]
                    else:
                        dfaux = df_codec[df_codec.DType.str.contains(TYPES[i])]
                        df_blz_aux = df_blz1[df_blz1.DType.str.contains(
                            TYPES[i])]
                    if dfaux.size > 0:
                        pearson_rates[i].append(
                            pearsonr(df_blz_aux['CRate'], dfaux['CRate']))
                        pearson_c_speeds[i].append(
                            pearsonr(df_blz_aux['CSpeed'], dfaux['CSpeed']))
                        codecs_cl[i].append(
                            codec + '-' + filt + '-' + str(c_level) + '-' +
                            TYPES[i])
                        colors[i].append(COLOR_PALETTE[codec])
                        markers[i].append(MARKER_DICT[filt])

    return pearson_rates, pearson_c_speeds, codecs_cl, colors, markers


def paint_codec_pearson_corr(df, cname, clevel):
    """Paint custom graphs for codec correlation."""

    pearson_rates, pearson_c_speeds, codecs_cl, colors, markers = \
        pearson_cor_data_builder(df, cname, clevel)

    for i in range(3):
        if len(pearson_rates[i]) > 0:
            fig = plt.figure(figsize=(10, 9))
            ax = fig.add_subplot(111)
            custom_pearson_scatter(ax, [x[0] for x in pearson_rates[i]],
                                   [x[0] for x in pearson_c_speeds[i]],
                                   colors[i], markers[i], cname.upper() +
                                   '-CL' + str(clevel), TYPES[i])
            custom_sc_legend(fig)
            plt.savefig(FIG_PATH + ax.get_title() +
                        '.png', bbox_inches='tight')


def custom_pairs(df, col_names):
    """Paint scatter matrix plot."""
```

```
434
435     print('%d points' % df.shape[0])
436     fig, axs = plt.subplots(3, len(col_names), sharex='col', sharey='row')
437     fig.set_size_inches(20, 12)
438     for j, y in enumerate(['CRate', 'CSpeed', 'DSpeed']):
439         for i, x in enumerate(col_names):
440             custom_centered_scatter(axs[j, i], df[x], df[y])
441             if j == 2:
442                 axs[j, i].set_xlabel(x)
443             if i == 0:
444                 axs[j, i].set_ylabel(y)
445     fig.tight_layout()
446     fig.suptitle(str(col_names) + ' VS Test Features', fontsize=16)
447     plt.subplots_adjust(top=0.95, hspace=0.01, wspace=0)
448     plt.savefig(FIG_PATH + str(col_names) +
449                 'VS Test Features' + '.png', bbox_inches='tight')
```

B.2. Cuaderno blosc_test_analysis.ipynb

Análisis de las pruebas realizadas con Blosc

Objetivos del análisis

- Relacionar el tamaño de bloque con las medidas de compresión y decompresión.
- Comprobar el comportamiento de los niveles de compresión sobre las pruebas.
- Comparar los datos de compresión de tablas normales y columnares.
- ¿Existe correlación entre blosclz o lz4 con nivel de compresión 1 y el resto de codecs?
- **[Punto muerto]** ¿Existe correlación entre las características del chunk y las medidas de compresión y decompresión?

Descripción de la muestra

El DataFrame en cuestión está formado por las características extraídas de un array de datos al comprimirlo y descomprimirlo mediante blosc. En cada fichero aparecen distintos conjuntos de datos los cuáles dividimos en fragmentos de 16 MegaBytes y sobre los cuales realizamos las pruebas de compresión y decompresión.
Cada fila se corresponde con los datos de realizar los test de compresión sobre un fragmento (*chunk*) de datos específico con un tamaño de bloque, codec, filtro y nivel de compresión determinados.

Variable	Descripción
Filename	nombre del fichero del que proviene.
DataSet	dentro del fichero el conjunto de datos del que proviene.
Table	0 si los datos vienen de un array, 1 si vienen de tablas y 2 para tablas columnares.
DType	indica el tipo de los datos.
Chunk_Number	número de fragmento dentro del conjunto de datos.
Chunk_Size	tamaño del fragmento.
Mean	la media.
Median	la mediana.
Sd	la desviación típica.
Skew	el coeficiente de asimetría.
Kurt	el coeficiente de apuntamiento.
Min	el mínimo absoluto.
Max	el máximo absoluto.
Q1	el primer cuartil.
Q3	el tercer cuartil.
N_Streaks	número de rachas seguidas por encima o debajo de la mediana.
Block_Size	el tamaño de bloque que utilizará Blosc para comprimir.
Codec	el codec de blosc utilizado.
Filter	el filtro de blosc utilizado.
CL	el nivel de compresión utilizado.
CRate	el ratio de compresión obtenido.
CSpeed	la velocidad de compresión obtenida en GB/s.
DSpeed	la velocidad de decompresión obtenida en GB/s.

In [1]:
```
%matplotlib inline
%config InlineBackend.figure_format='retina'

%load_ext autoreload
%autoreload 2

%load_ext version_information
%version_information numpy, scipy, matplotlib, pandas
```

Software	Version
Python	3.5.3 64bit [GCC 4.4.7 20120313 (Red Hat 4.4.7-1)]
IPython	5.1.0
OS	Linux 4.9.16 gentoo x86_64 with debian stretch sid
numpy	1.12.1
scipy	0.19.0
matplotlib	2.0.0
pandas	0.19.2
Sat Apr 29 10:46:42 2017 UTC	

In [2]:
```
import os
import sys
sys.path.append("../src/")

from IPython.display import display
import matplotlib
from matplotlib import pyplot as plt
import pandas as pd

import custom_plots as cst

pd.options.display.float_format = '{:,.3f}'.format
matplotlib.rcParams.update({'font.size': 12})
```

Descripción general

Cargamos el csv entero, comprobamos que no faltan campos y mostramos un breve resumen.

In [3]:
```
# LOAD WHOLE CSV
my_df = pd.read_csv('../data/blosc_test_data_final.csv.gz', sep='\t')
# SORT COLUMNS
my_df = my_df[cst.COLS]
# CHECK MISSING DATA
if not my_df.isnull().any().any():
    print('No missing data')
else:
    print("Missing data")
```

```
No missing data
```

In [4]:
```
# SUMMARY OF THE DATAFRAME
display(my_df[cst.COLS[5:]].describe())
```

	Chunk_Size	Mean	Median	Sd	Skew	Kurt	Min
count	1,127,520.000	1,127,520.000	1,127,520.000	1,127,520.000	1,127,520.000	1,127,520.000	1,127,520.000
mean	14.384	5,855,203,352,860,118.000	5,855,510,553,455,461.000	3,338,034,656,223.127	12.276	2,960.293	5,849,131,049,179
std	3.965	88,991,425,790,313,680.000	88,996,095,055,340,192.000	50,744,320,308,850.523	40.009	20,641.213	88,899,134,721,54
min	0.015	-996.946	-999.000	0.000	-134.250	-3.000	-999.000
25%	16.000	0.000	0.000	0.183	0.062	-0.888	-14.210
50%	16.000	0.101	0.000	2.679	3.052	12.431	0.000
75%	16.000	3.121	0.001	59.326	9.994	184.219	0.000
max	16.000	1,358,459,542,578,043,904.000	1,358,622,091,149,467,648.000	795,493,396,001,273.125	497.825	316,831.759	1,356,998,404,761

Como se puede observar hay mucha variabilidad en nuestros datos, lo cual es bueno.

Veamos cuantos conjuntos de datos tiene el fichero.

In [5]:
```
sets = my_df.drop_duplicates(subset=['DataSet', 'Table'])[cst.DESC_SET]
print('First ten datasets')
display(sets.head(n=10))
print('There are %d datasets' % (sets.shape[0]))
```

```
First ten datasets
```

	DataSet	DType	Table	Chunk_Size
0	/U	float32	0.000	16.000
85860	/V	float32	0.000	16.000
150660	/Grids/G1/precipAllObs	int32	0.000	0.738
152280	/Grids/G1/surfPrecipLiqRateProb	float32	0.000	0.015
153900	/Grids/G1/surfPrecipLiqRateUn	float32	0.000	0.015
155520	/Grids/G1/surfPrecipTotRateDiurnalAllObs	int32	0.000	1.107
157140	/Grids/G1/surfPrecipTotRateProb	float32	0.000	0.015
158760	/Grids/G1/surfPrecipTotRateUn	float32	0.000	0.015
160380	/Grids/G2/precipAllObs	int32	0.000	16.000
170100	/Grids/G2/surfPrecipLiqRateProb	float32	0.000	5.889

```
There are 120 datasets
```

Tablas de referencia de los conjuntos de datos

Procedemos a mostrar un resumen de las características extraídas de cada conjunto de datos.

In [6]:
```
for dataset in sets.drop_duplicates(subset=['DataSet'])['DataSet']:
        set_info = sets[sets.DataSet == dataset]
        print('SUMMARY')
        print(set_info)
        aux_set = my_df[my_df.DataSet == dataset].drop_duplicates(subset=['Chunk_Number'])
        if aux_set.shape[0] > 1:
            display(aux_set.describe()[cst.CHUNK_FEATURES])
        else:
            display(aux_set[cst.CHUNK_FEATURES])
```

```
SUMMARY
  DataSet    DType  Table  Chunk_Size
0      /U  float32  0.000      16.000
```

	Chunk_Size	Mean	Median	Sd	Skew	Kurt	Min	Max	Q1	Q3	N_Streaks
count	53.000	53.000	53.000	53.000	53.000	53.000	53.000	53.000	53.000	53.000	53.000
mean	15.726	14.314	9.838	12.731	0.624	-0.720	-15.663	48.531	4.747	24.449	94,648.358
std	1.995	4.350	3.881	2.527	0.153	0.319	4.249	9.016	3.546	6.457	19,730.974
min	1.475	5.762	2.096	6.404	0.228	-1.164	-28.273	27.146	-1.446	11.317	13,756.000
25%	16.000	10.708	7.198	11.469	0.532	-0.921	-17.664	43.313	1.943	18.781	82,271.000
50%	16.000	15.430	9.564	13.268	0.649	-0.793	-14.891	50.507	4.873	25.905	95,832.000
75%	16.000	17.302	13.019	14.752	0.738	-0.624	-12.637	53.064	7.273	27.890	109,934.000
max	16.000	22.909	16.944	17.010	1.072	0.549	-9.488	62.922	10.356	38.366	124,896.000

```
SUMMARY
      DataSet    DType  Table  Chunk_Size
85860      /V  float32  0.000      16.000
```

	Chunk_Size	Mean	Median	Sd	Skew	Kurt	Min	Max	Q1	Q3	N_Streaks
count	40.000	40.000	40.000	40.000	40.000	40.000	40.000	40.000	40.000	40.000	40.000
mean	15.900	2.073	1.526	4.976	0.311	0.301	-16.644	23.037	-1.197	4.881	194,369.825
std	0.631	2.517	1.696	2.391	0.510	0.742	4.047	9.672	1.472	3.369	36,781.843
min	12.009	-1.626	-1.256	2.474	-0.600	-0.370	-29.640	13.443	-4.887	0.552	141,024.000
25%	16.000	-0.240	-0.088	3.414	-0.031	-0.162	-19.447	16.708	-2.397	2.357	162,933.000
50%	16.000	1.886	1.721	4.303	0.253	0.240	-16.237	20.683	-0.907	4.500	186,578.000
75%	16.000	3.743	2.851	5.930	0.644	0.484	-13.344	25.571	-0.204	6.422	219,685.000

max	16.000	9.091	4.968	14.419	1.748	4.040	-11.483	56.161	1.253	15.373	293,423.000

SUMMARY

	DataSet	DType	Table	Chunk_Size
150660	/Grids/G1/precipAllObs	int32	0.000	0.738

	Chunk_Size	Mean	Median	Sd	Skew	Kurt	Min	Max	Q1	Q3	N_Streaks
150660	0.738	46,750.635	42,412.000	42,964.463	1.123	2.123	0.000	211,383.000	121.000	79,434.750	27,744.000

SUMMARY

	DataSet	DType	Table	Chunk_Size
152280	/Grids/G1/surfPrecipLiqRateProb	float32	0.000	0.015

	Chunk_Size	Mean	Median	Sd	Skew	Kurt	Min	Max	Q1	Q3	N_Streaks
152280	0.015	0.044	0.037	0.040	1.346	3.059	0.000	0.352	0.011	0.066	1,032.000

SUMMARY

	DataSet	DType	Table	Chunk_Size
153900	/Grids/G1/surfPrecipLiqRateUn	float32	0.000	0.015

	Chunk_Size	Mean	Median	Sd	Skew	Kurt	Min	Max	Q1	Q3	N_Streaks
153900	0.015	0.092	0.048	0.123	2.762	12.094	0.000	1.414	0.011	0.124	992.000

SUMMARY

	DataSet	DType	Table	Chunk_Size
155520	/Grids/G1/surfPrecipTotRateDiurnalAllObs	int32	0.000	1.107

	Chunk_Size	Mean	Median	Sd	Skew	Kurt	Min	Max	Q1	Q3	N_Streaks
155520	1.107	1,947.943	272.000	2,888.095	2.804	13.277	0.000	24,063.000	0.000	3,094.000	31,604.000

SUMMARY

	DataSet	DType	Table	Chunk_Size
157140	/Grids/G1/surfPrecipTotRateProb	float32	0.000	0.015

	Chunk_Size	Mean	Median	Sd	Skew	Kurt	Min	Max	Q1	Q3	N_Streaks
157140	0.015	0.050	0.043	0.040	1.218	2.721	0.000	0.352	0.018	0.072	1,137.000

SUMMARY

	DataSet	DType	Table	Chunk_Size
158760	/Grids/G1/surfPrecipTotRateUn	float32	0.000	0.015

	Chunk_Size	Mean	Median	Sd	Skew	Kurt	Min	Max	Q1	Q3	N_Streaks
158760	0.015	0.101	0.064	0.121	2.739	12.272	0.000	1.414	0.022	0.133	1,099.000

SUMMARY

	DataSet	DType	Table	Chunk_Size
160380	/Grids/G2/precipAllObs	int32	0.000	16.000

	Chunk_Size	Mean	Median	Sd	Skew	Kurt	Min	Max	Q1	Q3	N_Streaks
count	6.000	6.000	6.000	6.000	6.000	6.000	6.000	6.000	6.000	6.000	6.000
mean	15.703	183.354	173.500	107.464	2.317	9.421	0.000	910.000	116.167	222.333	198,604.333
std	0.727	6.775	10.710	1.604	0.023	0.173	0.000	0.000	5.742	4.502	14,750.195
min	14.219	176.954	163.000	105.856	2.292	9.218	0.000	910.000	111.000	218.000	179,642.000
25%	16.000	177.709	164.250	106.084	2.296	9.273	0.000	910.000	111.500	218.500	188,721.000
50%	16.000	181.741	171.500	107.211	2.316	9.426	0.000	910.000	114.500	221.500	197,790.500
75%	16.000	187.916	181.750	108.644	2.338	9.577	0.000	910.000	119.750	225.250	210,105.250
max	16.000	193.347	188.000	109.676	2.342	9.605	0.000	910.000	125.000	229.000	216,495.000

SUMMARY

	DataSet	DType	Table	Chunk_Size
170100	/Grids/G2/surfPrecipLiqRateProb	float32	0.000	5.889

	Chunk_Size	Mean	Median	Sd	Skew	Kurt	Min	Max	Q1	Q3	N_Streaks
170100	5.889	0.045	0.009	0.074	2.804	12.031	0.000	1.000	0.000	0.063	291,171.000

SUMMARY

	DataSet	DType	Table	Chunk_Size
171720	/Grids/G2/surfPrecipLiqRateUn	float32	0.000	5.889

	Chunk_Size	Mean	Median	Sd	Skew	Kurt	Min	Max	Q1	Q3	N_Streaks
171720	5.889	0.094	0.004	0.337	12.404	321.944	0.000	26.186	0.000	0.051	288,953.000

SUMMARY

	DataSet	DType	Table	Chunk_Size
173340	/Grids/G2/surfPrecipTotRateDiurnalAllObs	int32	0.000	16.000

	Chunk_Size	Mean	Median	Sd	Skew	Kurt	Min	Max	Q1	Q3	N_Streaks
count	9.000	9.000	9.000	9.000	9.000	9.000	9.000	9.000	9.000	9.000	9.000
mean	15.703	7.629	0.000	12.705	1.798	4.317	0.000	102.222	0.000	13.778	154,746.778
std	0.891	1.237	0.000	0.843	0.792	4.134	0.000	17.683	0.000	8.059	56,353.353
min	13.328	5.467	0.000	11.541	0.728	-1.091	0.000	65.000	0.000	0.000	80,589.000
25%	16.000	7.239	0.000	11.730	1.005	-0.012	0.000	93.000	0.000	14.000	109,781.000
50%	16.000	8.051	0.000	12.935	1.725	4.011	0.000	113.000	0.000	18.000	161,634.000
75%	16.000	8.505	0.000	13.343	2.259	6.763	0.000	113.000	0.000	19.000	207,840.000
max	16.000	9.073	0.000	13.710	2.875	9.907	0.000	114.000	0.000	20.000	223,100.000

SUMMARY

	DataSet	DType	Table	Chunk_Size
187920	/Grids/G2/surfPrecipTotRateProb	float32	0.000	5.889

	Chunk_Size	Mean	Median	Sd	Skew	Kurt	Min	Max	Q1	Q3	N_Streaks
187920	5.889	0.050	0.017	0.075	2.606	10.682	0.000	1.000	0.000	0.074	305,495.000

SUMMARY

```
                DataSet    DType  Table  Chunk_Size
1116180  /s506/events.n3  float32  1.000       7.224
```

	Chunk_Size	Mean	Median	Sd	Skew	Kurt	Min	Max	Q1	Q3	N_Streaks
1116180	7.224	0.860	0.553	1.942	-134.250	74,975.796	-999.000	310.790	0.000	1.207	946,423.000

SUMMARY

```
                DataSet    DType  Table  Chunk_Size
1117800  /s506/events.n4  float32  1.000       7.224
```

	Chunk_Size	Mean	Median	Sd	Skew	Kurt	Min	Max	Q1	Q3	N_Streaks
1117800	7.224	0.890	0.603	1.544	13.214	1,255.583	-1.000	321.740	0.000	1.259	947,489.000

SUMMARY

```
                DataSet    DType  Table  Chunk_Size
1119420  /s506/events.t1  float32  1.000       7.224
```

	Chunk_Size	Mean	Median	Sd	Skew	Kurt	Min	Max	Q1	Q3	N_Streaks
1119420	7.224	-370.239	12.500	611.587	0.584	0.980	-999.000	5,022.500	-999.000	15.000	845,656.000

SUMMARY

```
                DataSet    DType  Table  Chunk_Size
1121040  /s506/events.t2  float32  1.000       7.224
```

	Chunk_Size	Mean	Median	Sd	Skew	Kurt	Min	Max	Q1	Q3	N_Streaks
1121040	7.224	-307.761	12.500	598.034	0.377	0.818	-999.000	5,017.500	-999.000	20.000	938,072.000

SUMMARY

```
                DataSet    DType  Table  Chunk_Size
1122660  /s506/events.t3  float32  1.000       7.224
```

	Chunk_Size	Mean	Median	Sd	Skew	Kurt	Min	Max	Q1	Q3	N_Streaks
1122660	7.224	-386.432	12.500	616.676	0.677	1.328	-999.000	5,385.000	-999.000	17.500	911,415.000

SUMMARY

```
                DataSet    DType  Table  Chunk_Size
1124280  /s506/events.t4  float32  1.000       7.224
```

	Chunk_Size	Mean	Median	Sd	Skew	Kurt	Min	Max	Q1	Q3	N_Streaks
1124280	7.224	-346.851	12.500	621.689	0.554	0.866	-999.000	5,122.500	-999.000	22.500	934,425.000

SUMMARY

```
                    DataSet    DType  Table  Chunk_Size
1125900  /s506/events.t_trigger  float32  1.000       7.224
```

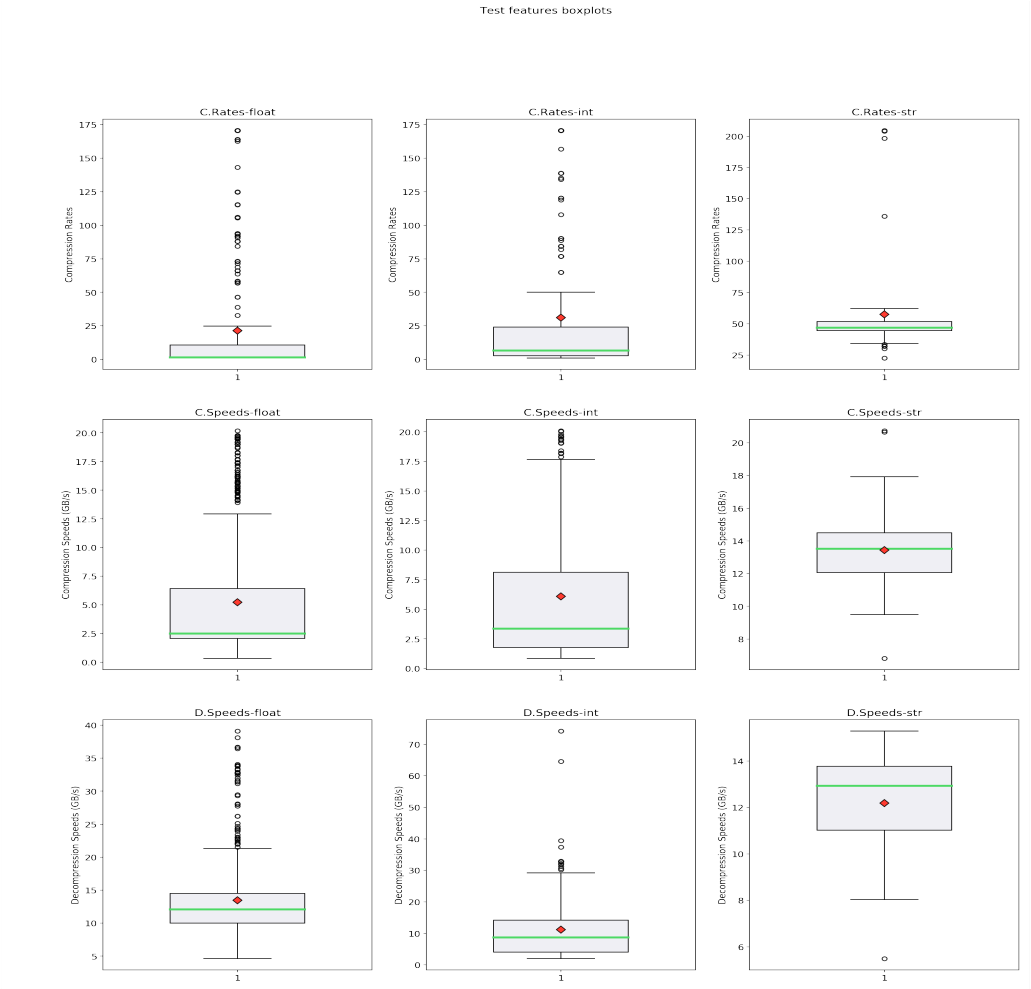
	Chunk_Size	Mean	Median	Sd	Skew	Kurt	Min	Max	Q1	Q3	N_Streaks
1125900	7.224	250.095	32.500	414.191	1.686	1.643	-999.000	4,997.500	22.500	267.500	940,519.000

No entraremos en detalles sobre cada conjunto de datos, simplemente nos conviene tener estas tablas como referencia rápida en caso de detectar anomalías en algún conjunto en concreto.

Detección de datos atípicos

Para evitar que los diagramas de caja esten plagados de datos atípicos, procedemos a filtrar con el codec blosclz, filtro shuffle, nivel de compresión 5 y tamaño de bloque automático para buscar con detenimiento datos atípicos.

In [7]:
```
df_outliers = my_df[(my_df.Block_Size == 0) & (my_df.CL == 5) &
                    (my_df.Codec == 'blosclz') &
                    (my_df.Filter == 'noshuffle')]
cst.paint_dtype_boxplots(df_outliers)
```



Mostramos a continuación los datos atípicos

In [20]:
```
for i in range(2):
    dfaux = df_outliers[df_outliers.DType.str.contains(cst.TYPES[i])]
    if dfaux.size > 0:
        cr_lim = cst.outlier_lim(dfaux['CRate'])
        cs_lim = cst.outlier_lim(dfaux['CSpeed'])
        ds_lim = cst.outlier_lim(dfaux['DSpeed'])
        result = dfaux[(dfaux.CRate < cr_lim[0]) |
                       (dfaux.CRate > cr_lim[1]) |
                       (dfaux.CSpeed < cs_lim[0]) |
                       (dfaux.CSpeed > cs_lim[1]) |
                       (dfaux.DSpeed < ds_lim[0]) |
                       (dfaux.DSpeed > ds_lim[1])][cst.ALL_FEATURES]
        if result.size > 0:
            print('%d %s OUTLIERS' % (result.shape[0],
                                      cst.TYPES[i].upper()))
            display(result.head())
```

```
81 FLOAT OUTLIERS
```

	Chunk_Size	Mean	Median	Sd	Skew	Kurt	Min	Max	Q1	Q3	N_Streaks	CRate	CSpeed	DSpeed
84244	1.475	7.835	6.866	6.404	0.414	0.209	-12.035	27.183	4.119	11.317	13,756.000	1.000	1.739	29.472
200884	2.215	0.456	0.000	1.532	8.146	210.120	0.000	122.311	0.000	0.000	28,860.000	2.935	3.793	38.118

202504	2.215	0.650	0.000	2.098	4.997	32.385	0.000	43.932	0.000	0.000	29,002.000	3.036	4.552	33.353
213844	2.215	0.036	0.000	0.104	5.367	54.662	0.000	4.711	0.000	0.000	28,858.000	2.935	4.428	39.103
215464	2.215	0.044	0.000	0.127	4.180	23.454	0.000	3.249	0.000	0.000	29,004.000	3.036	4.566	36.434

38 INT OUTLIERS

	Chunk_Size	Mean	Median	Sd	Skew	Kurt	Min	Max	Q1	Q3	N_Streaks	CRate	CSpeed	DSpeed
199264	2.445	0.000	0.000	0.027	99.485	18,684.148	0.000	8.000	0.000	0.000	505.000	156.816	20.032	74.193
212224	2.445	0.002	0.000	0.055	50.616	3,993.813	0.000	8.000	0.000	0.000	1,523.000	134.316	19.042	22.407
231664	16.000	0.000	0.000	0.000	0.000	-3.000	0.000	0.000	0.000	0.000	1.000	170.639	19.035	29.024
251104	2.445	0.001	0.000	0.045	47.237	3,683.632	0.000	7.000	0.000	0.000	1,533.000	135.187	19.078	64.619
278644	16.000	0.000	0.000	0.000	0.000	-3.000	0.000	0.000	0.000	0.000	1.000	170.639	19.401	30.187

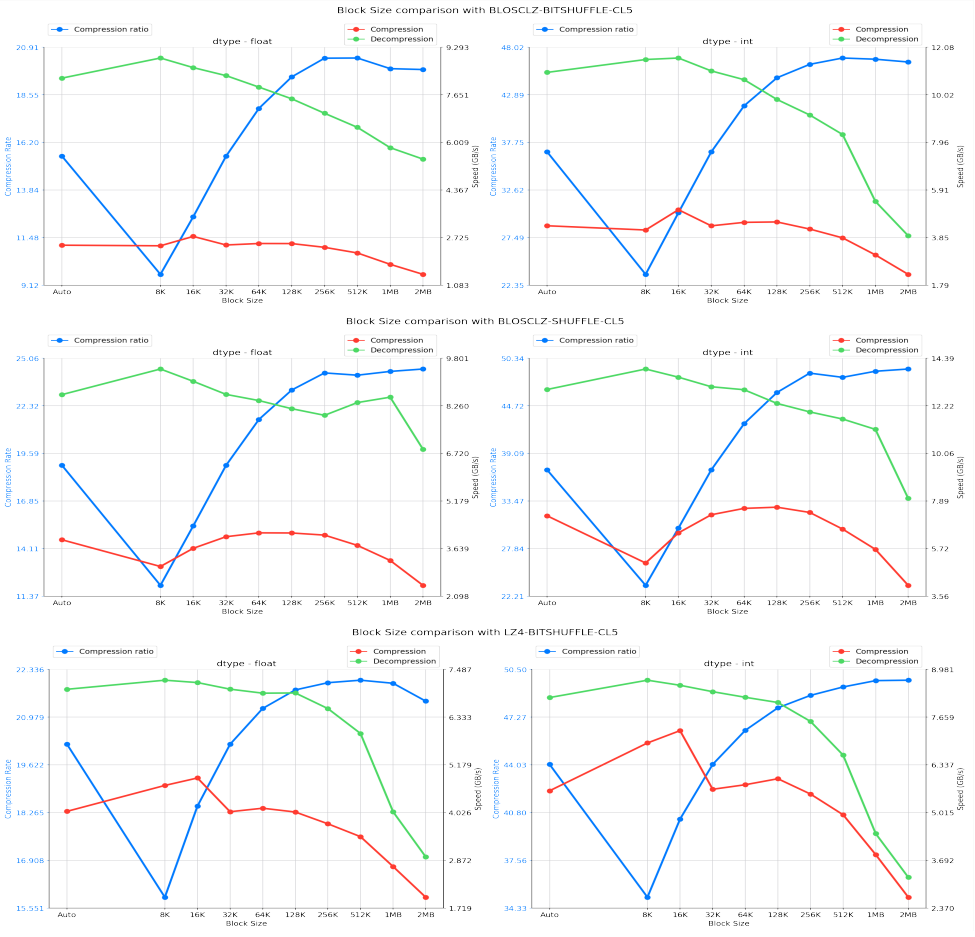
No mostramos los datos atípicos de tipo string dado que no extraemos ninguna característica de chunk que podamos comentar.
En cuanto a los datos atípicos observamos que la mayoría son series de números idénticos o muy parecidos, siempre con un rango intercuartílico de 0.

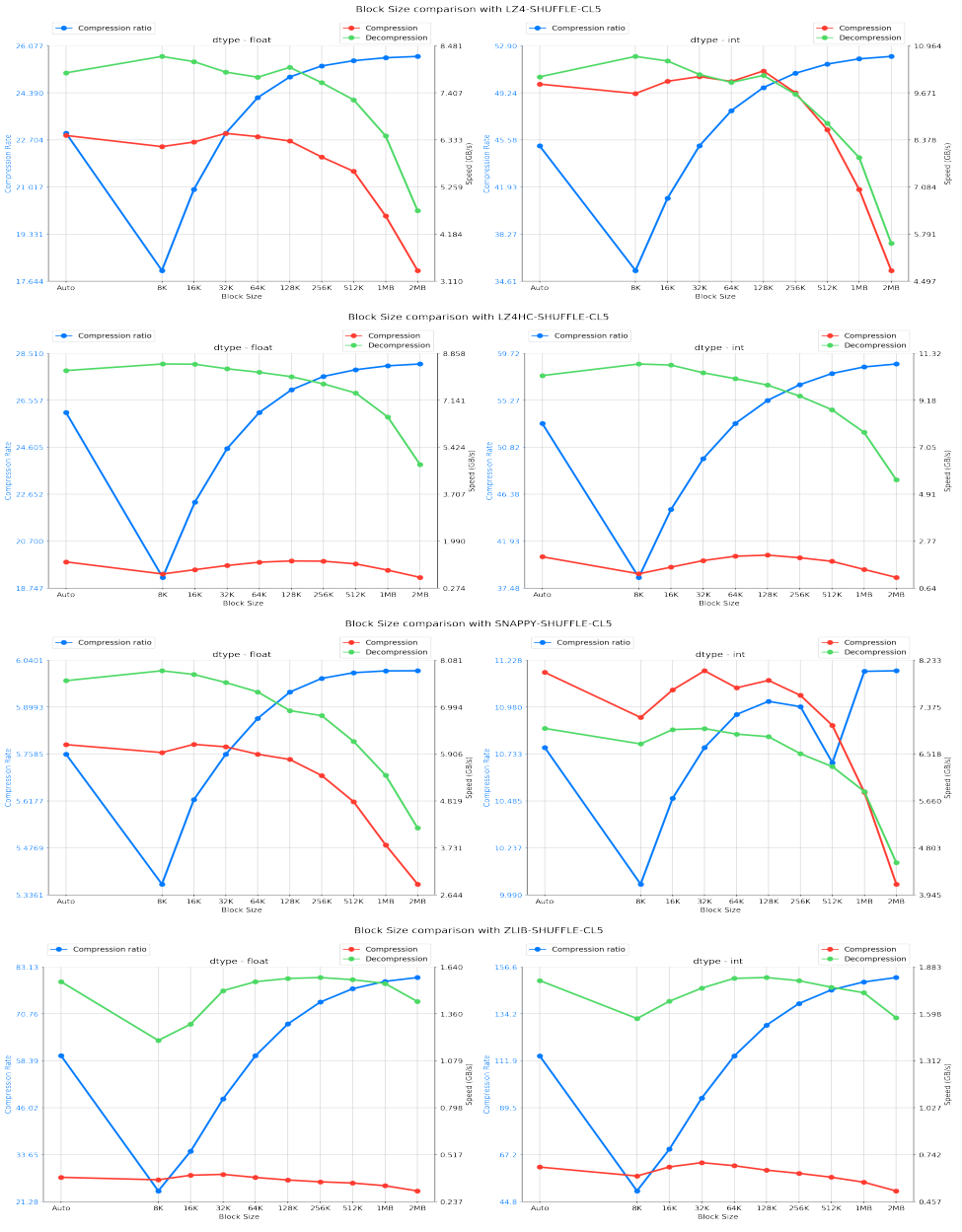
Correlaciones Block Size

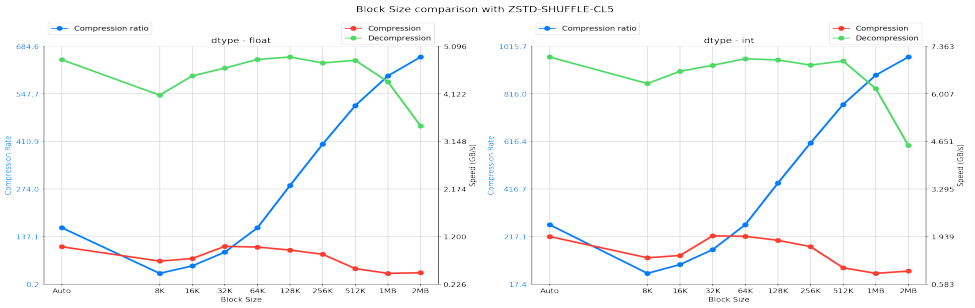
Aquí pretendemos observar la correlación entre el tamaño de bloque y las medidas de compresión, para ello filtramos los datos por tipo, codec, filtro, nivel de compresión y tamaño de bloque; y calculamos la media de su ratio de compresión y velocidades de compresión/decompresión.

Las gráficas presentan los ratios de compresión (en azul) y las velocidades de compresión y de descompresión (en rojo y verde) medios para cada tamaño de bloque. Primero mostramos estos datos para los datos de tipo float y de tipo int.

```
In [9]: cst.paint_all_block_cor(my_df, 'shuffle', c_level=5)
```

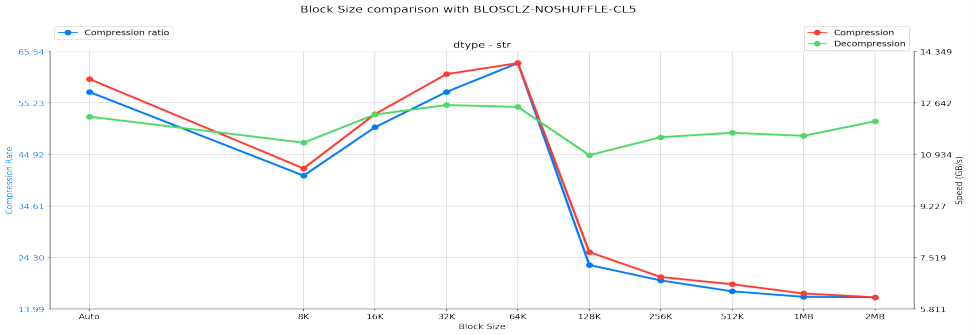


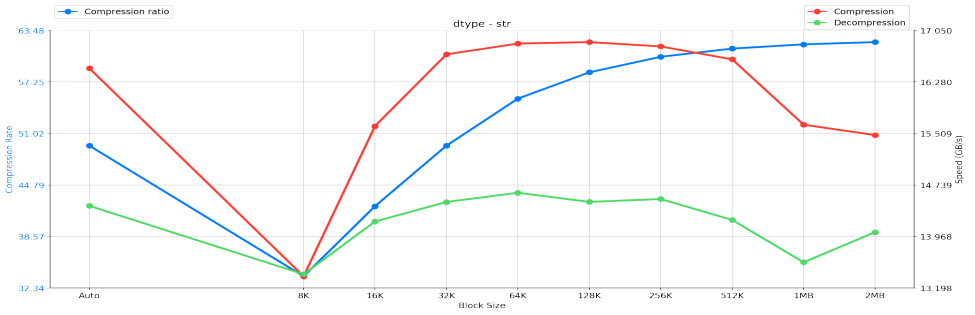

Block Size comparison with LZ4-SHUFFLE-CL5
dtype - float
dtype - int
Block Size comparison with LZ4HC-SHUFFLE-CL5
dtype - float
dtype - int
Block Size comparison with SNAPPY-SHUFFLE-CL5
dtype - float
dtype - int
Block Size comparison with ZLIB-SHUFFLE-CL5
dtype - float
dtype - int
Compression ratio
Compression
Decompression
Compression Rate
Speed (GB/s)
Block Size
Auto
8K
16K
32K
64K
128K
256K
512K
1MB
2MB

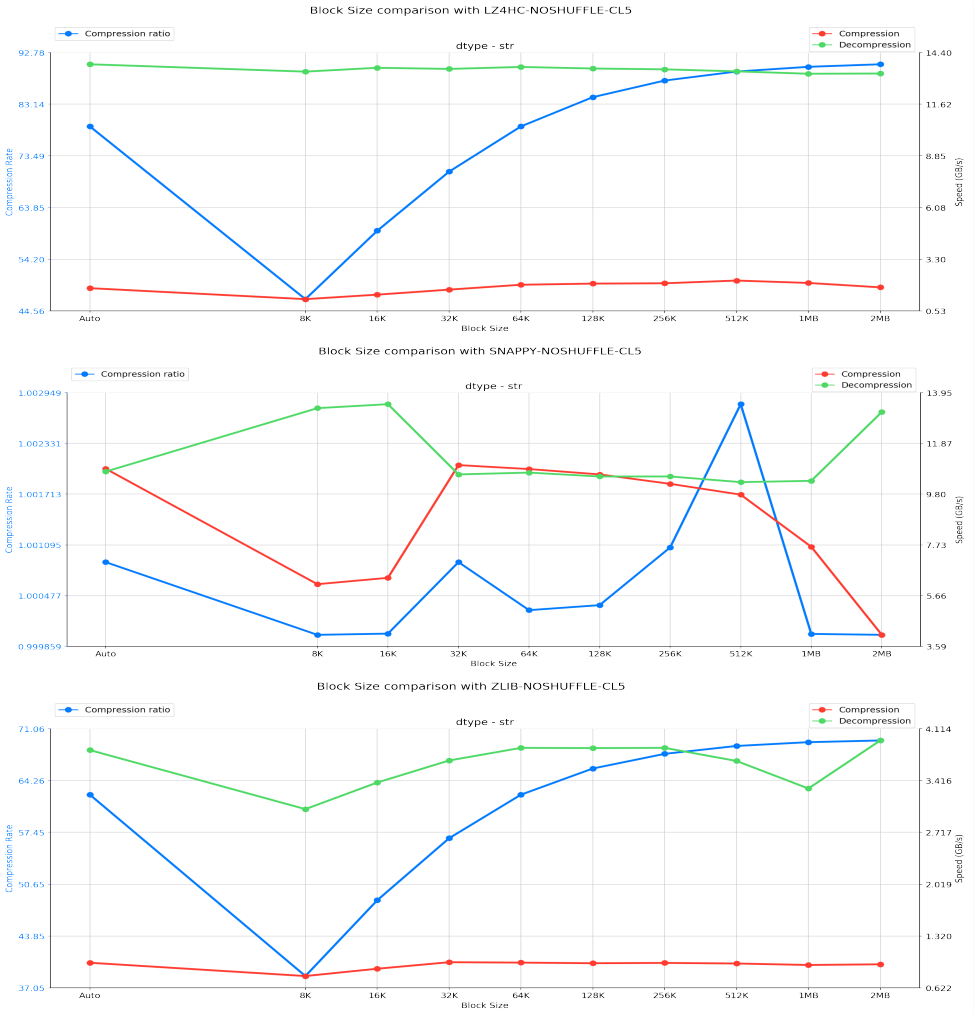


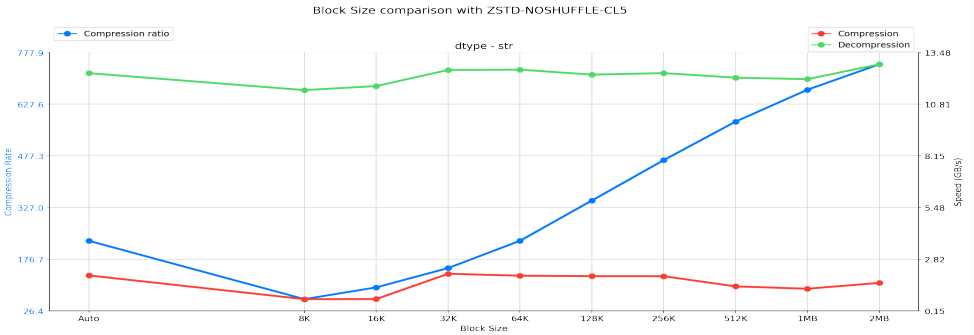
Aquí se muestran los mismos gráficos pero para los datos del tipo cadenas de texto

```
In [10]: cst.paint_all_block_cor(my_df, 'noshuffle')
```






Block Size comparison with LZ4HC-NOSHUFFLE-CL5
dtype - str
Compression ratio
Compression
Decompression
Compression Rate
Speed (GB/s)
92.78
83.14
73.49
63.85
54.20
44.56
14.40
11.62
8.85
6.08
3.30
0.53
Auto
8K
16K
32K
64K
128K
256K
512K
1MB
2MB
Block Size
Block Size comparison with SNAPPY-NOSHUFFLE-CL5
dtype - str
Compression ratio
Compression
Decompression
Compression Rate
Speed (GB/s)
1.002949
1.002331
1.001713
1.001095
1.000477
0.999859
13.95
11.87
9.80
7.73
5.66
3.59
Auto
8K
16K
32K
64K
128K
256K
512K
1MB
2MB
Block Size
Block Size comparison with ZLIB-NOSHUFFLE-CL5
dtype - str
Compression ratio
Compression
Decompression
Compression Rate
Speed (GB/s)
71.06
64.26
57.45
50.65
43.85
37.05
4.114
3.416
2.717
2.019
1.320
0.622
Auto
8K
16K
32K
64K
128K
256K
512K
1MB
2MB
Block Size

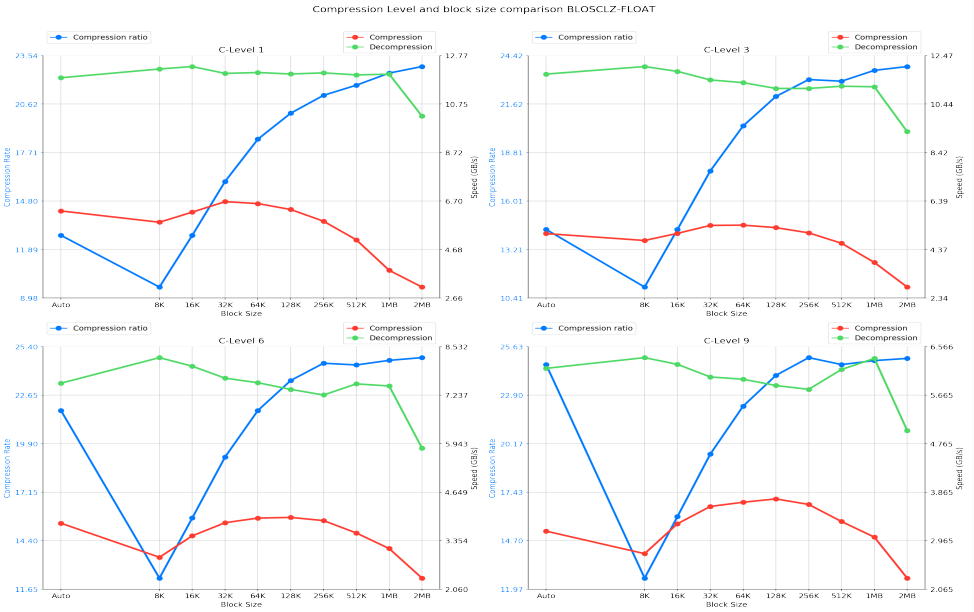


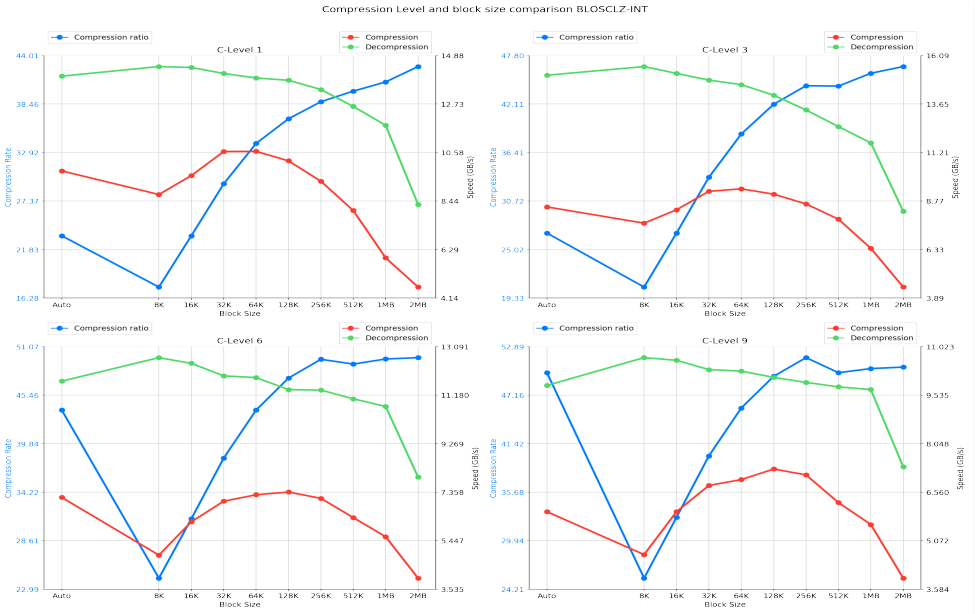
Como podemos observar, al aumentar el tamaño de bloque suele aumentar el ratio de compresión pero parece converger hasta un límite entre los tamaños de 512 KB y 2 MB. Además cuando el tamaño de bloque es menor en general las velocidades son más rápidas.

Por otro lado destaca el comportamiento de Snappy pues no parece comprimir muy bien con respecto al resto. Por otro lado Zlib parece ser inferior en todo a Zstd.

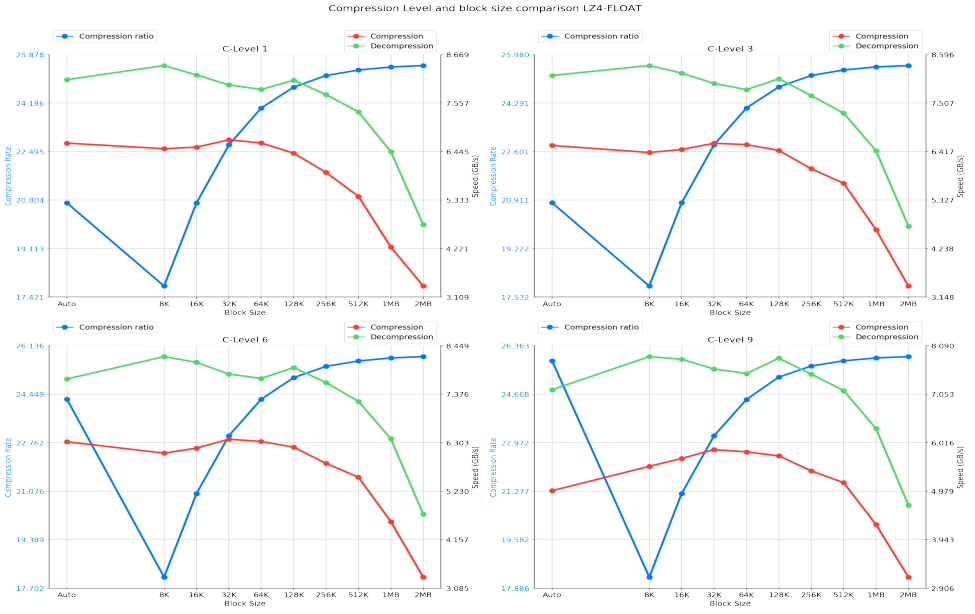
Aquí se presentan las mismas gráficas pero alterando el nivel de compresión para ver como afecta al tamaño de bloque.

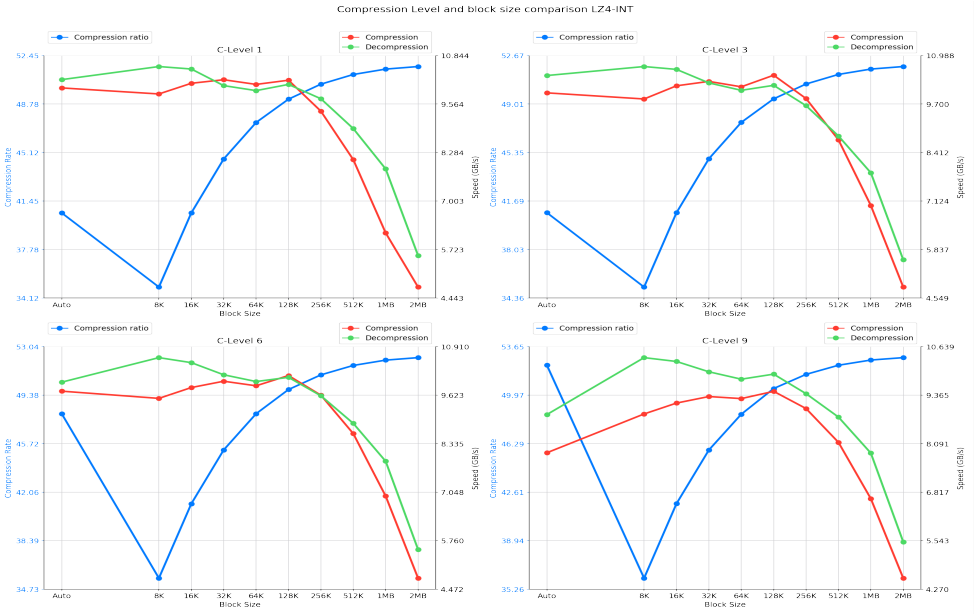
```
In [11]: cst.paint_cl_comparison(my_df, 'shuffle', 'blosclz')
```





In [12]:
```
cst.paint_cl_comparison(my_df, 'shuffle', 'lz4')
```



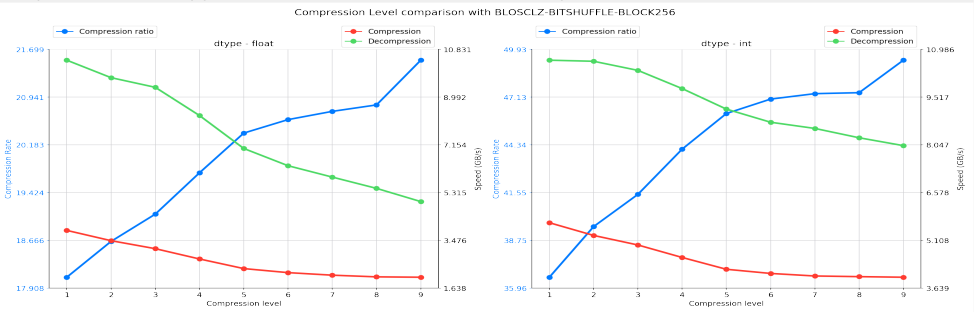


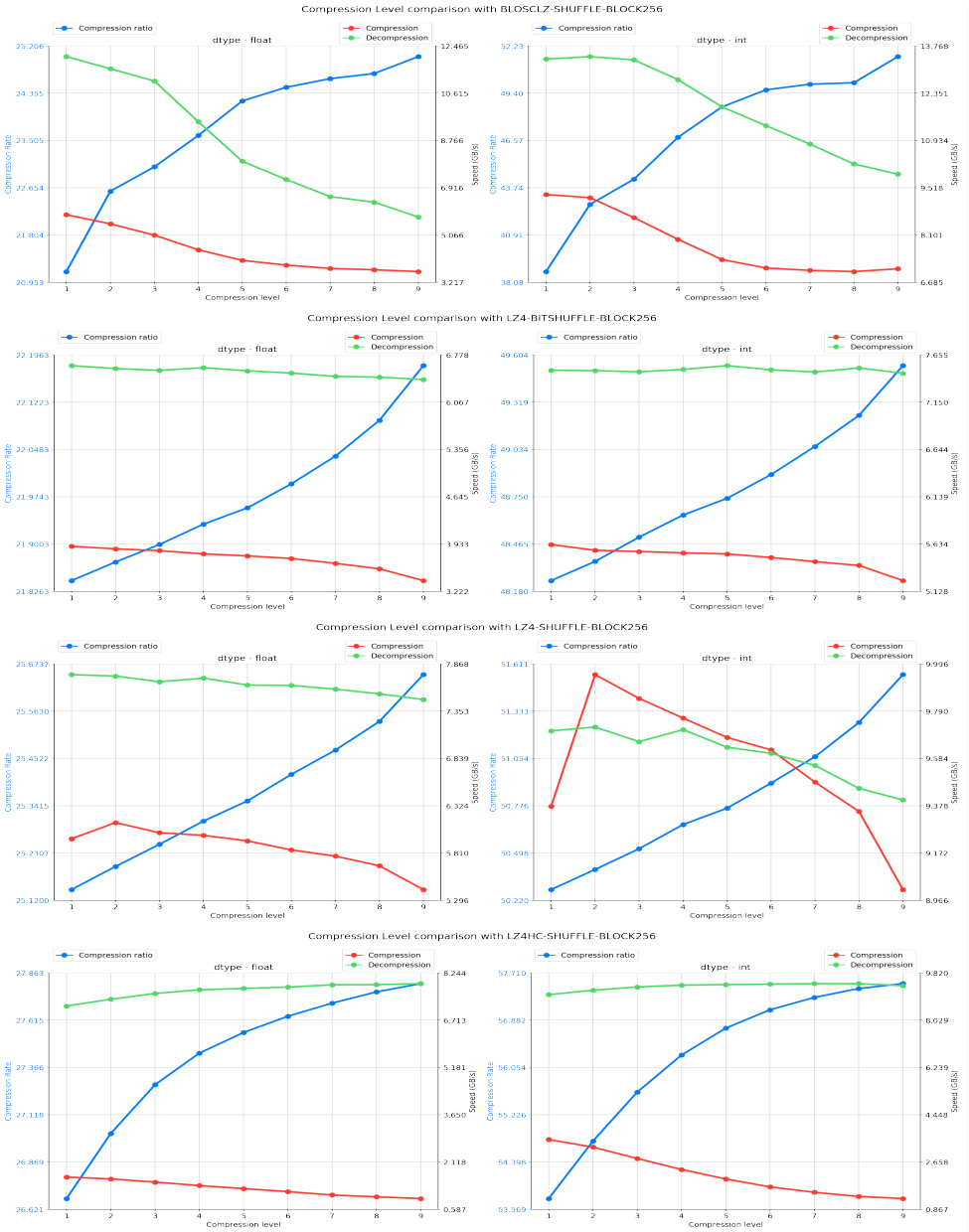
Los resultados son los esperados el comportamiento es en general el mismo, simplemente suben los ratio de compresión y bajan las velocidades a medida que aumenta el nivel de compresión. Por otra parte destaca el comportamiento del tamaño de bloque automático observamos que está programado para que aumente conjuntamente con el nivel de compresión.

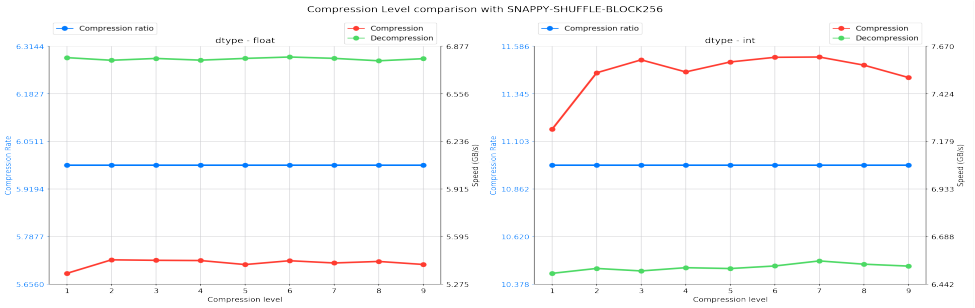
Comparación de niveles de compresión

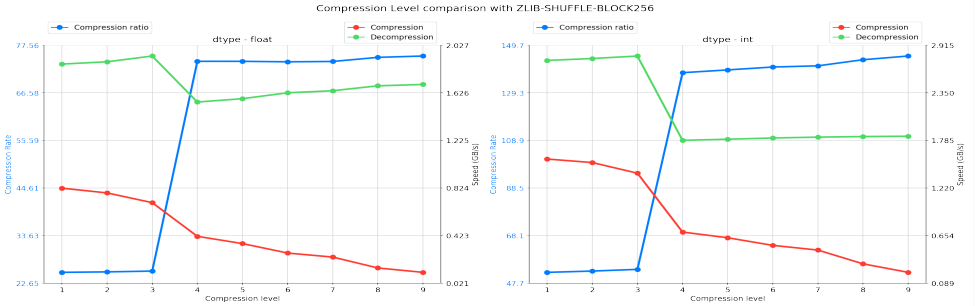
Al igual que en el anterior caso hacemos los mismos gráficos pero observando el nivel de compresión.

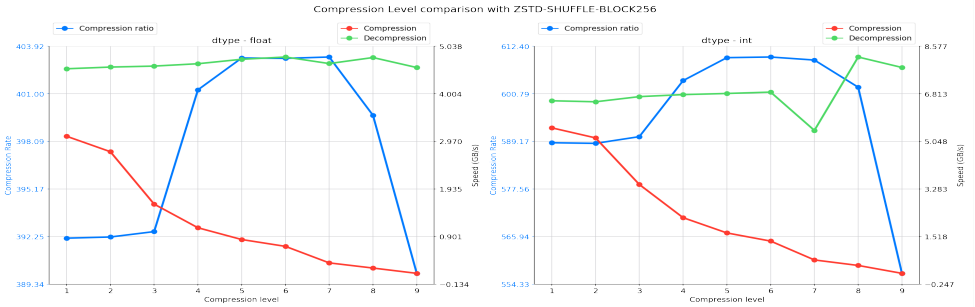
In [13]:
```
# BLOCK SIZE --> CL
cst.paint_all_block_cor(my_df, 'shuffle', block_size=256, cl_mode=True)
```



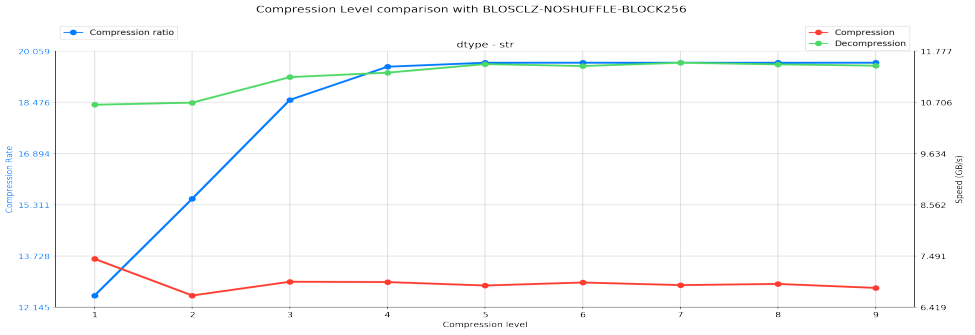

Compression Level comparison with BLOSCLZ-SHUFFLE-BLOCK256
dtype - float
dtype - int
Compression ratio
Compression
Decompression
Compression Rate
Speed (GB/s)
Compression level
Compression Level comparison with LZ4-BITSHUFFLE-BLOCK256
Compression Level comparison with LZ4-SHUFFLE-BLOCK256
Compression Level comparison with LZ4HC-SHUFFLE-BLOCK256

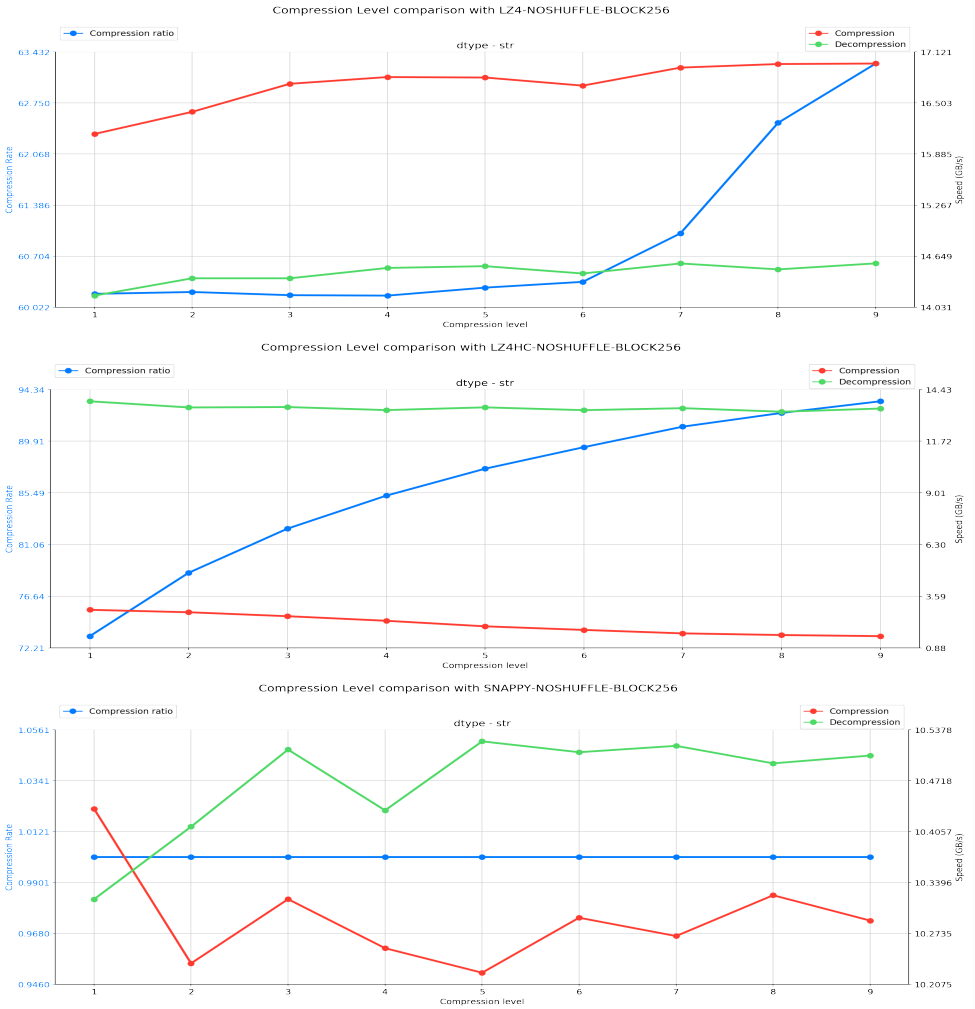


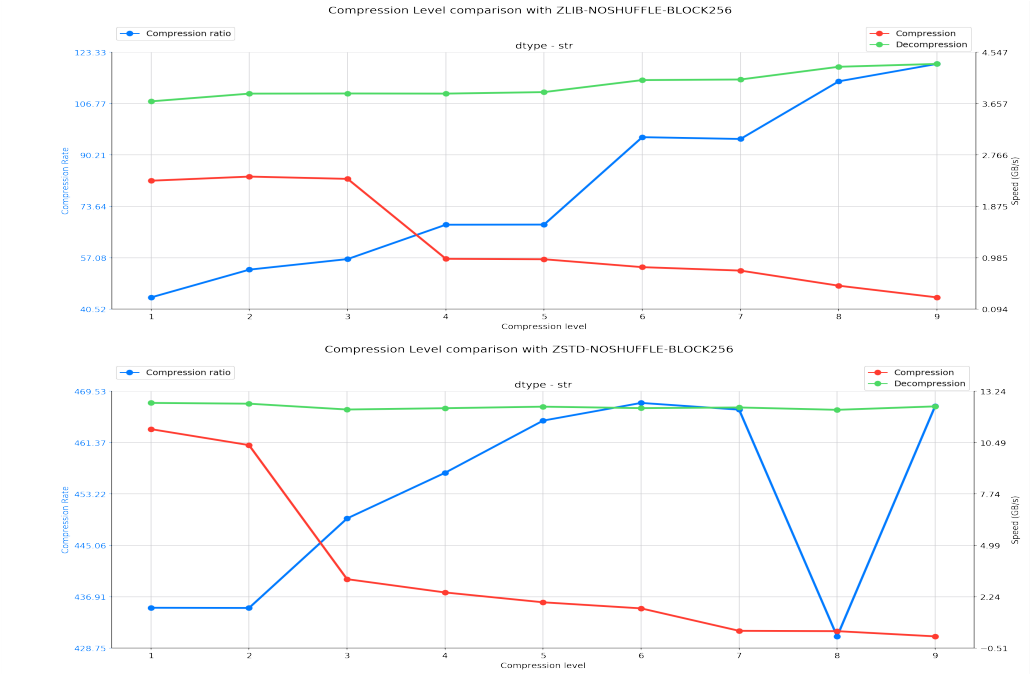





```
In [14]: cst.paint_all_block_cor(my_df, 'noshuffle', block_size=256, cl_mode=True)
```




Compression Level comparison with LZ4-NOSHUFFLE-BLOCK256
Compression ratio
Compression
Decompression
dtype - str
63.432
62.750
62.068
61.386
60.704
60.022
Compression Rate
17.121
16.503
15.885
15.267
14.649
14.031
Speed (GB/s)
1
2
3
4
5
6
7
8
9
Compression level
Compression Level comparison with LZ4HC-NOSHUFFLE-BLOCK256
Compression ratio
Compression
Decompression
dtype - str
94.34
89.91
85.49
81.06
76.64
72.21
Compression Rate
14.43
11.72
9.01
6.30
3.59
0.88
Speed (GB/s)
1
2
3
4
5
6
7
8
9
Compression level
Compression Level comparison with SNAPPY-NOSHUFFLE-BLOCK256
Compression ratio
Compression
Decompression
dtype - str
1.0561
1.0341
1.0121
0.9901
0.9680
0.9460
Compression Rate
10.5378
10.4718
10.4057
10.3396
10.2735
10.2075
Speed (GB/s)
1
2
3
4
5
6
7
8
9
Compression level



Destaca el comportamiento de Snappy de nuevo vuelve a ser el más raro de todos, el nivel de compresión no cambia nada. Por otro lado Zlib tiene un cambio brusco a partir del nivel de compresión 3, esto se debe a que a partir de ese nivel activa métodos más potentes a la hora de comprimir. Finalmente Zstd parece hacer lo mismo que Zlib, pero parece que en los últimos niveles de compresión no funciona bien, pues pierde ratio de compresión.

Tablas columnares VS Tablas normales

En el caso de que los datos esten en forma de tabla, si la tabla contiene más de una columna se realizan dos pruebas de compresión, una guardando los datos como tabla normal, fila por fila y otra guardándolos columnarmente.

```
In [15]: df_col = my_df[my_df.Table == 2]
         if df_col.size > 0:
             sets = df_col.drop_duplicates(subset=['DataSet'])
             for dataset in sets['DataSet']:
                 dfaux = my_df[my_df.DataSet == dataset]
                 normal_table = dfaux[dfaux.Table == 1][cst.TEST_FEATURES]
                 normal_table.columns = ['N_CRate', 'N_CSpeed', 'N_DSpeed']
                 col_table = dfaux[dfaux.Table == 2][cst.TEST_FEATURES]
                 col_table.columns = ['COL_CRate', 'COL_CSpeed', 'COL_DSpeed']
                 result = pd.concat([normal_table, col_table])
                 result = result[['N_CRate', 'COL_CRate', 'N_CSpeed',
                                  'COL_CSpeed','N_DSpeed', 'COL_DSpeed']]
                 print(sets[sets.DataSet == dataset][cst.DESC_SET])
                 display(result.describe())
```

```
                          DataSet    DType  Table  Chunk_Size
1001160  /msft/table.values_block_0  float64  2.000      16.000
```

	N_CRate	COL_CRate	N_CSpeed	COL_CSpeed	N_DSpeed	COL_DSpeed
count	9,720.000	9,720.000	9,720.000	9,720.000	9,720.000	9,720.000
mean	10.462	17.942	2.063	2.281	5.760	4.986
std	7.310	33.079	2.237	2.462	4.037	3.204
min	1.000	1.000	0.002	0.002	0.361	0.345
25%	4.342	5.133	0.413	0.401	2.599	2.595
50%	8.844	8.050	1.249	1.557	4.425	4.276
75%	14.532	15.924	3.038	3.397	8.164	6.744
max	39.004	297.005	10.584	15.156	28.628	28.773

```
                          DataSet DType  Table  Chunk_Size
1025460  /msft/table.values_block_2  |S49  2.000      16.000
```

	N_CRate	COL_CRate	N_CSpeed	COL_CSpeed	N_DSpeed	COL_DSpeed
count	12,960.000	12,960.000	12,960.000	12,960.000	12,960.000	12,960.000
mean	52.373	147.258	6.509	6.845	11.466	10.729

std	45.062	737.652	5.962	6.167	3.926	3.685
min	1.000	1.000	0.005	0.007	1.862	1.881
25%	7.624	21.154	1.176	1.012	10.437	9.548
50%	46.981	47.146	4.621	5.745	12.911	11.920
75%	72.699	86.410	11.163	11.211	13.708	13.126
max	234.129	10,131.164	20.215	22.567	34.169	30.888

```
                                DataSet  DType  Table  Chunk_Size
1046520  /s501/events.pulseheights  int16  2.000      12.999
```

	N_CRate	COL_CRate	N_CSpeed	COL_CSpeed	N_DSpeed	COL_DSpeed
count	1,620.000	1,620.000	1,620.000	1,620.000	1,620.000	1,620.000
mean	1.412	1.408	1.788	1.545	5.182	4.917
std	0.314	0.307	2.634	2.243	4.310	4.134
min	1.000	1.000	0.002	0.002	0.398	0.366
25%	1.070	1.079	0.118	0.114	1.533	1.571
50%	1.479	1.473	0.650	0.624	4.019	3.790
75%	1.711	1.702	2.320	2.132	6.754	6.321
max	2.041	2.034	10.864	10.954	13.834	14.090

```
                               DataSet  DType  Table  Chunk_Size
1051380  /s501/events.integrals  int32  2.000      16.000
```

	N_CRate	COL_CRate	N_CSpeed	COL_CSpeed	N_DSpeed	COL_DSpeed
count	3,240.000	3,240.000	3,240.000	3,240.000	3,240.000	3,240.000
mean	2.199	2.196	1.574	1.473	4.854	4.707
std	0.478	0.465	1.848	1.672	3.330	3.265
min	1.000	1.000	0.003	0.003	0.503	0.363
25%	1.979	1.983	0.194	0.184	2.044	1.979
50%	2.197	2.184	0.935	0.899	4.277	4.094
75%	2.555	2.538	2.265	2.120	6.912	6.646
max	3.337	3.353	10.366	7.975	14.669	14.785

```
                                DataSet  DType  Table  Chunk_Size
1077300  /s503/events.pulseheights  int16  2.000       7.706
```

	N_CRate	COL_CRate	N_CSpeed	COL_CSpeed	N_DSpeed	COL_DSpeed
count	1,620.000	1,620.000	1,620.000	1,620.000	1,620.000	1,620.000
mean	1.476	1.609	1.685	1.849	5.506	5.006
std	0.364	0.407	2.447	2.514	4.916	4.439
min	1.000	1.000	0.003	0.003	0.463	0.445
25%	1.085	1.216	0.117	0.131	1.623	1.594
50%	1.517	1.733	0.697	0.763	3.803	3.878
75%	1.811	1.954	2.334	2.602	6.985	6.048
max	2.157	2.304	10.535	10.696	15.687	15.782

```
                               DataSet  DType  Table  Chunk_Size
1080540  /s503/events.integrals  int32  2.000      15.412
```

	N_CRate	COL_CRate	N_CSpeed	COL_CSpeed	N_DSpeed	COL_DSpeed
count	1,620.000	1,620.000	1,620.000	1,620.000	1,620.000	1,620.000
mean	2.322	2.681	1.632	2.165	4.971	5.347
std	0.570	0.561	1.851	2.453	3.185	3.381
min	1.000	1.309	0.004	0.006	0.644	0.652
25%	1.984	2.540	0.204	0.266	2.411	2.482
50%	2.245	2.748	1.010	1.200	4.540	5.224
75%	2.684	3.026	2.361	3.383	7.110	7.711
max	3.470	3.780	9.585	12.086	13.480	12.727

```
                                DataSet  DType  Table  Chunk_Size
1104840  /s506/events.pulseheights  int16  2.000      14.449
```

	N_CRate	COL_CRate	N_CSpeed	COL_CSpeed	N_DSpeed	COL_DSpeed
count	1,620.000	1,620.000	1,620.000	1,620.000	1,620.000	1,620.000
mean	1.414	1.410	1.946	1.703	5.586	5.287
std	0.315	0.308	2.804	2.389	4.372	4.332
min	1.000	1.000	0.002	0.003	0.481	0.409
25%	1.069	1.083	0.133	0.126	1.838	1.702
50%	1.487	1.479	0.722	0.683	4.954	4.501
75%	1.702	1.696	2.572	2.316	7.118	6.470
max	2.040	2.034	11.275	11.068	13.450	13.589

```
                               DataSet  DType  Table  Chunk_Size
1109700  /s506/events.integrals  int32  2.000      16.000
```

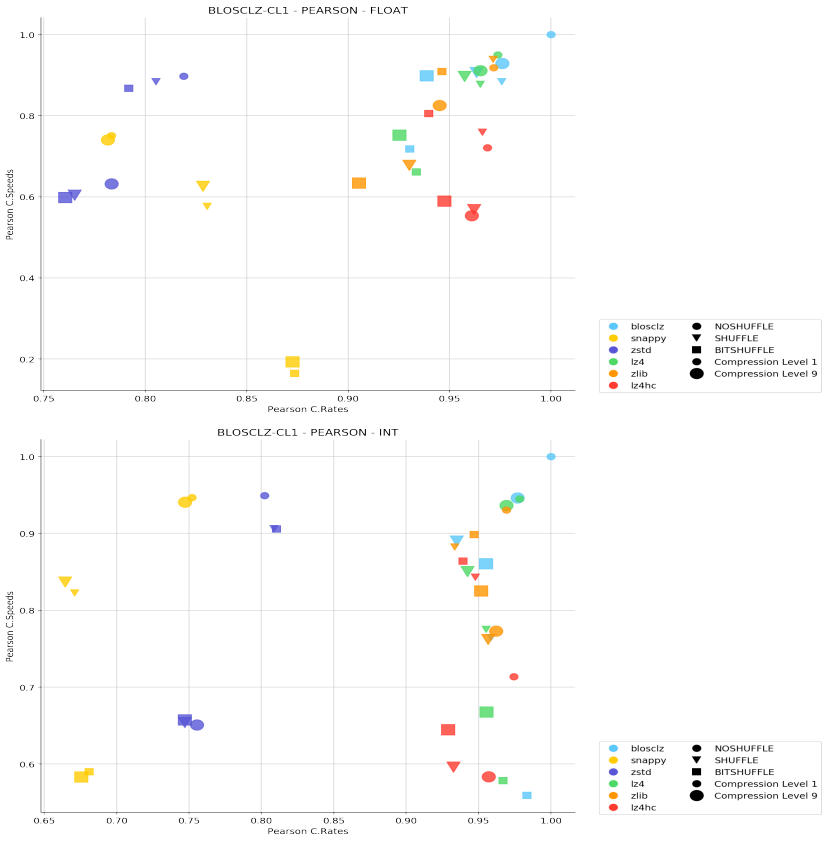
	N_CRate	COL_CRate	N_CSpeed	COL_CSpeed	N_DSpeed	COL_DSpeed
count	3,240.000	3,240.000	3,240.000	3,240.000	3,240.000	3,240.000
mean	2.204	2.203	1.673	1.551	5.080	4.912
std	0.473	0.462	1.915	1.721	3.341	3.268
min	1.000	1.000	0.004	0.005	0.592	0.558
25%	1.985	1.987	0.213	0.205	2.255	2.286
50%	2.213	2.202	0.969	0.936	4.682	4.568
75%	2.545	2.544	2.388	2.253	7.217	6.879
max	3.330	3.390	10.567	8.706	13.782	13.920

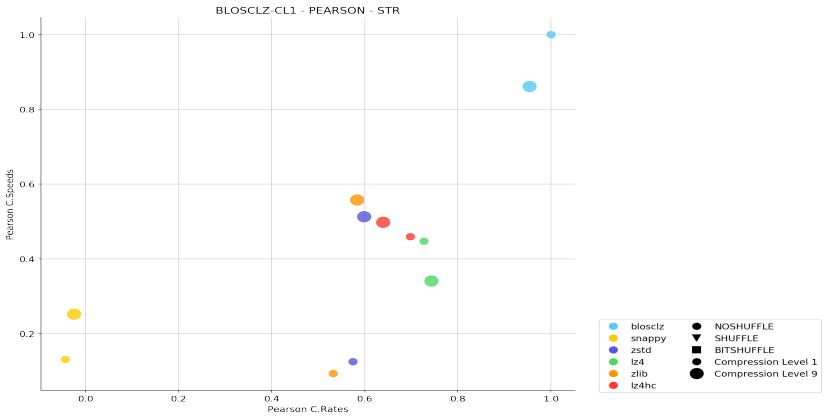
Como era de esperar, parece que las tablas columnares son más comprimibles. Aunque hay casos en los que se comprimen igual, nunca se comprimen menos.

Correlaciones Blosclz-CL1 VS Otros

Para poder visualizar todas estas correlaciones calculamos directamente el coeficiente de pearson asociado entre los datos de blosclz con nivel de compresión 1 y el resto.

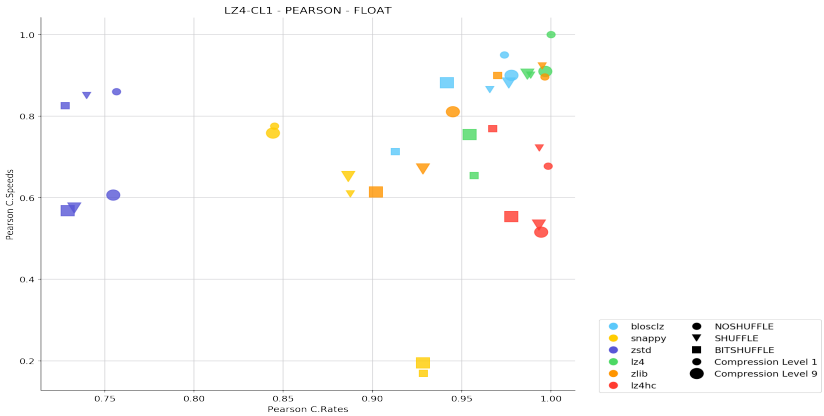
```
In [16]: cst.paint_codec_pearson_corr(my_df, 'blosclz', 1)
```

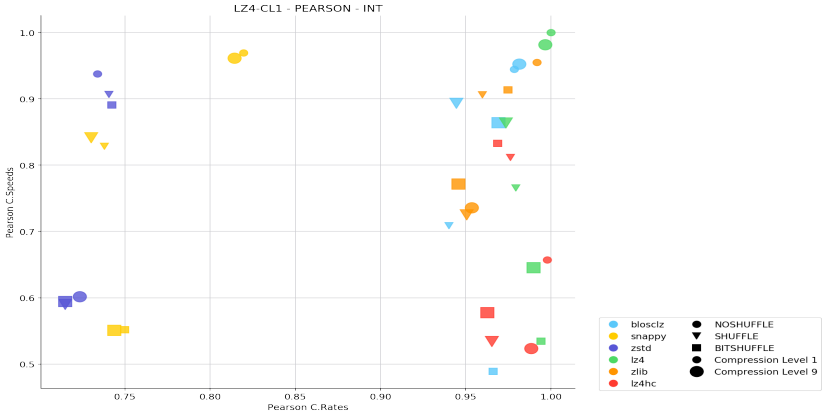


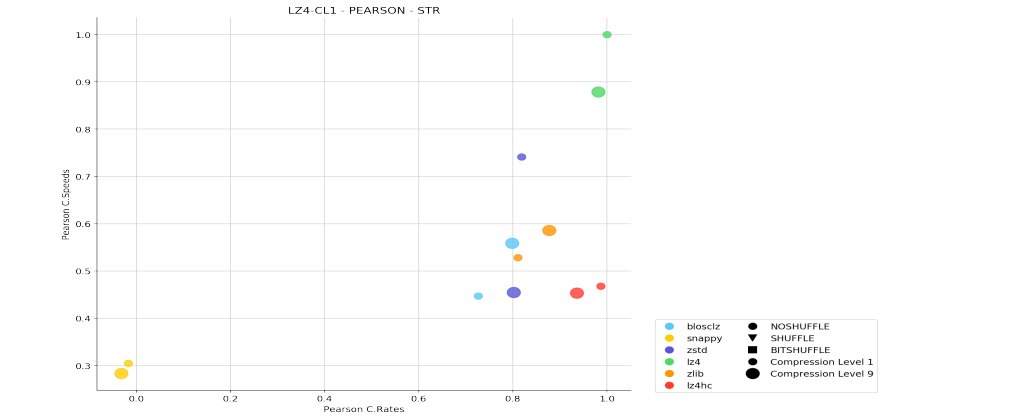


Aquí hacemos los mismo para LZ4

```
In [17]: cst.paint_codec_pearson_corr(my_df, 'lz4', 1)
```







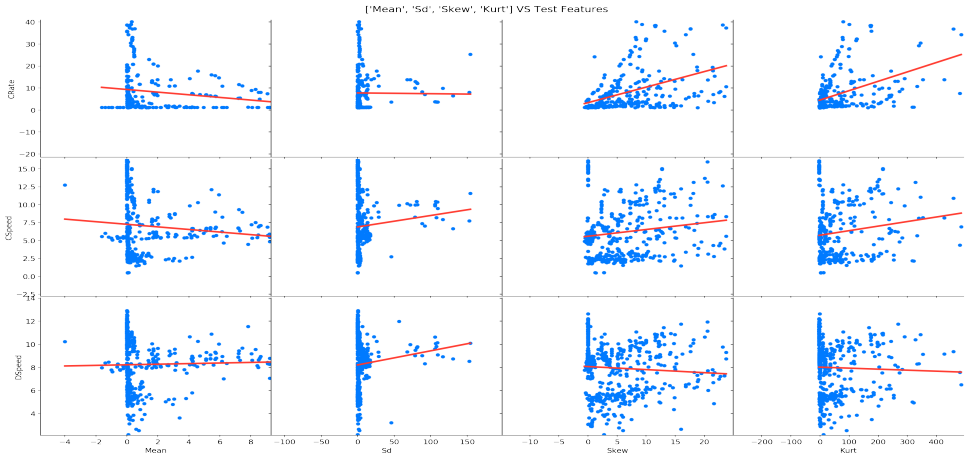
Los resultados son bastante buenos, además era de esperar. Aunque LZ4 tiene mejores resultados ambas opciones parecen lo suficientemente buenas.

Correlaciónes entre características de chunk y pruebas de compresión

Aquí se trata de observar las correlaciones entre características de chunk seleccionadas y las pruebas de compresiones. Para ello se utiliza un gráfico de pares personalizado. Además los datos se filtran por codec, filtro, nivel de compresión y tamaño de bloque, sino no tendría sentido los gráficos debido a la enorme variabilidad que habría.

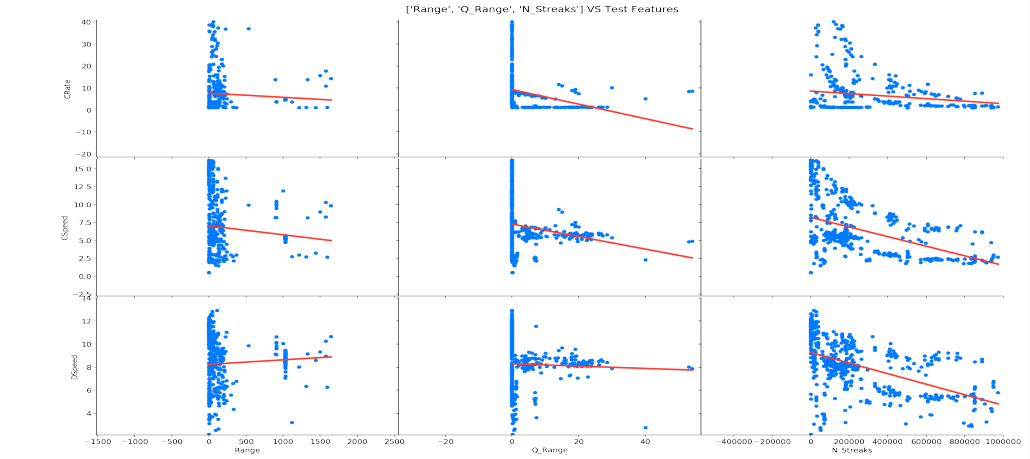
In [18]:
```
dfaux = my_df[(my_df.Codec == 'lz4') & (my_df.Block_Size == 256) &
              (my_df.Filter == 'shuffle') & (my_df.CL == 5) &
              (my_df.DType.str.contains('float') |
               my_df.DType.str.contains('int'))]
cols = ['Mean', 'Sd', 'Skew', 'Kurt']
cst.custom_pairs(dfaux, cols)
```

```
680 points
```



In [19]:
```
cols = ['Range', 'Q_Range', 'N_Streaks']
dfaux = dfaux.assign(Range=dfaux['Max'] - dfaux['Min'])
dfaux = dfaux.assign(Q_Range=dfaux['Q3'] - dfaux['Q1'])
cst.custom_pairs(dfaux, cols)
```

```
680 points
```



Aunque se podría plantear decir que a mayor rango y número de rachas disminuye el ratio de compresión, no sería muy adecuado sacar conclusiones de estos gráficos. Hay demasiada variabilidad en los datos en sí como para extraer conclusiones de un simple gráfico, será mejor que estas correlaciones las busquen los algoritmos de clasificación en sí.

Anexo C

Generador de los datos de entrenamiento

Para la generación de los datos de entrenamiento se utilizó de nuevo un cuaderno *jupyter*, pues aunque se trata de un programa de mero procesamiento de datos resultaba interesante destacar algunas observaciones. Por tanto, en el cuaderno **training_data_generator.ipynb** mostrado a continuación se encuentra el código de generación de los datos de entrenamiento.

C.1. Cuaderno training_data_generator.ipynb

Generador de los datos de entrenamiento

Objetivos del análisis

- Extraer el data frame final con los datos preparados para entrenar algoritmos machine learning.

In [1]:
```
%load_ext autoreload
%autoreload 2

%matplotlib inline

%load_ext version_information
%version_information numpy, pandas
```

Software	Version
Python	3.5.3 64bit [GCC 4.4.7 20120313 (Red Hat 4.4.7-1)]
IPython	5.1.0
OS	Linux 4.9.16 gentoo x86_64 with debian stretch sid
numpy	1.12.1
pandas	0.19.2
Sat Apr 29 13:34:01 2017 UTC	

In [2]:
```
import numpy as np
import pandas as pd
from IPython.display import display
import matplotlib.pyplot as plt

pd.options.display.float_format = '{:,.3f}'.format
```

In [3]:
```
CHUNK_ID = ["Filename", "DataSet", "Table", "Chunk_Number"]
CHUNK_FEATURES = ["Table", "DType", "Chunk_Size", "Mean", "Median",
                  "Sd", "Skew", "Kurt", "Min", "Max", "Q1", "Q3",
                  "N_Streaks"]
OUT_OPTIONS = ["Block_Size", "Codec", "Filter", "CL"]
TEST_FEATURES = ["CRate", "CSpeed", "DSpeed"]
COLS = ["Filename" , "DataSet", "Chunk_Number"] + CHUNK_FEATURES
       + OUT_OPTIONS + TEST_FEATURES
IN_TESTS = ['BLZ_CRate', 'BLZ_CSpeed', 'BLZ_DSpeed', 'LZ4_CRate',
            'LZ4_CSpeed', 'LZ4_DSpeed']
IN_USER = ['IN_CR', 'IN_CS', 'IN_DS']
```

In [4]:
```
df = pd.read_csv('../data/blosc_test_data_final.csv.gz', sep='\t')
my_df = df[(df.CL != 0) & (df.CRate > 1.1)]
```

Extracción de los datos por chunk

In [5]:
```
%%time
# DATAFRAME WITH DISTINCT CHUNKS
chunks_df = my_df.drop_duplicates(subset=CHUNK_ID)
print("%d rows" % chunks_df.shape[0])
chunk_tests_list = []
# FOR EACH CHUNK
for index, row in chunks_df.iterrows():
    # DATAFRAME WITH CHUNK TESTS
    chunk_tests_list.append(my_df[(my_df.Filename == row["Filename"]) &
                                  (my_df.DataSet == row["DataSet"]) &
                                  (my_df.Table == row["Table"]) &
                                  (my_df.Chunk_Number == row["Chunk_Number"])])
```

```
725 rows
CPU times: user 1min 36s, sys: 76 ms, total: 1min 36s
Wall time: 1min 36s
```

Selección de opciones para cada chunk

In [6]:
```
%%time
training_df = pd.DataFrame()
for chunk_test in chunk_tests_list:
    # EXTRACT MAX MIN AND SOME AUX MAX INDICES
    i_max_crate, i_max_c_speed, i_max_d_speed = \
        chunk_test['CRate'].idxmax(), chunk_test['CSpeed'].idxmax(),\
        chunk_test['DSpeed'].idxmax()
    max_crate, max_c_speed, max_d_speed = \
        (chunk_test.ix[i_max_crate]['CRate'],
         chunk_test.ix[i_max_c_speed]['CSpeed'],
         chunk_test.ix[i_max_d_speed]['DSpeed'])

    min_crate, min_c_speed, min_d_speed = (chunk_test['CRate'].min(),
                                           chunk_test['CSpeed'].min(),
                                           chunk_test['DSpeed'].min())
    # NORMALIZED COLUMNS
    chunk_test = chunk_test.assign(N_CRate=(chunk_test['CRate']
                                            - min_crate)
                                           / (max_crate - min_crate),
                                   N_CSpeed=(chunk_test['CSpeed']
                                             - min_c_speed)
                                            / (max_c_speed - min_c_speed),
                                   N_DSpeed=(chunk_test['DSpeed']
                                             - min_d_speed)
                                            / (max_d_speed - min_d_speed))
    # DISTANCE FUNC COLUMNS
```

```
    chunk_test = chunk_test.assign(Distance_1=(chunk_test['N_CRate'] - 1)**2
                                   + (chunk_test['N_CSpeed'] - 1)**2,
                                   Distance_2=(chunk_test['N_CRate'] - 1)**2
                                   + (chunk_test['N_DSpeed'] - 1) ** 2,
                                   Distance_3=(chunk_test['N_CRate'] - 1)**2
                                   + (chunk_test['N_DSpeed'] - 1)**2
                                   + (chunk_test['N_CSpeed'] - 1)**2,
                                   Distance_4=(chunk_test['N_CSpeed'] - 1)**2
                                   + (chunk_test['N_DSpeed'] - 1) ** 2)
    # BALANCED INDICES
    i_balanced_c_speed, i_balanced_d_speed,
    i_balanced, i_balanced_speeds = (chunk_test['Distance_1'].idxmin(),
                                     chunk_test['Distance_2'].idxmin(),
                                     chunk_test['Distance_3'].idxmin(),
                                     chunk_test['Distance_4'].idxmin())
    indices = [i_max_d_speed, i_max_c_speed, i_balanced_speeds,
               i_max_crate, i_balanced_d_speed, i_balanced_c_speed,
               i_balanced]
    # TYPE FILTER FOR LZ_DATA
    d_type = chunk_test.iloc[0]['DType']
    filter_name = 'noshuffle'
    if 'float' in d_type or 'int' in d_type:
        filter_name = 'shuffle'
    aux = df[(df.CL == 1) & (df.Block_Size == 0) &
             (df.Filter == filter_name) &
             (df.Filename == chunk_test.iloc[0]['Filename']) &
             (df.DataSet == chunk_test.iloc[0]['DataSet']) &
             (df.Table == chunk_test.iloc[0]['Table']) &
             (df.Chunk_Number == chunk_test.iloc[0]['Chunk_Number'])]
    lz_data = np.append(aux[aux.Codec == 'blosclz'][TEST_FEATURES].values[0],
                        aux[aux.Codec == 'lz4'][TEST_FEATURES].values[0])
    # APPEND ROWS TO TRAINING DATA FRAME
    for i in range(len(indices)):
        in_1, r = divmod((i+1), 4)
        in_2, in_3 = divmod(r, 2)
        training_df = training_df.append(
                        dict(zip(COLS + IN_TESTS + IN_USER,
                        np.append(np.append(chunk_test.ix[indices[i]][COLS].values,
                        lz_data),
                        [in_1, in_2, in_3]))),
                        ignore_index=True)
```

```
CPU times: user 3min 6s, sys: 44 ms, total: 3min 6s
Wall time: 3min 6s
```

Algunas comprobaciones

In [7]:

```
print('DISTINCT MAX RATE')
distinct_max_rate = training_df[(training_df.IN_CR == 1) &
                                (training_df.IN_CS == 0) &
                                (training_df.IN_DS == 0)]\
                        .drop_duplicates(subset=OUT_OPTIONS)
                        [OUT_OPTIONS + TEST_FEATURES]
print('%s rows' % distinct_max_rate.shape[0])
display(distinct_max_rate.head())
print('DISTINCT MAX C.SPEED')
distinct_max_c_speed = training_df[(training_df.IN_CR == 0) &
                                   (training_df.IN_CS == 1) &
                                   (training_df.IN_DS == 0)]\
                         .drop_duplicates(subset=OUT_OPTIONS)
                         [OUT_OPTIONS + TEST_FEATURES]
print('%s rows' % distinct_max_c_speed.shape[0])
display(distinct_max_c_speed.head())
print('DISTINCT MAX D.SPEED')
distinct_max_d_speed = training_df[(training_df.IN_CR == 0) &
                                   (training_df.IN_CS == 0) &
                                   (training_df.IN_DS == 1)]\
                        .drop_duplicates(subset=OUT_OPTIONS)
                        [OUT_OPTIONS + TEST_FEATURES]
print('%s rows' % distinct_max_d_speed.shape[0])
display(distinct_max_d_speed.head())
print('DISTINCT BALANCED CSPEED')
distinct_balanced_c_speed = training_df[(training_df.IN_CR == 1) &
                                        (training_df.IN_CS == 1) &
                                        (training_df.IN_DS == 0)]\
                            .drop_duplicates(subset=OUT_OPTIONS)
                            [OUT_OPTIONS + TEST_FEATURES]
print('%s rows' % distinct_balanced_c_speed.shape[0])
display(distinct_balanced_c_speed.head())
print('DISTINCT BALANCED DSPEED')
distinct_balanced_d_speed = training_df[(training_df.IN_CR == 1) &
                                        (training_df.IN_CS == 0) &
                                        (training_df.IN_DS == 1)]\
                            .drop_duplicates(subset=OUT_OPTIONS)
                            [OUT_OPTIONS + TEST_FEATURES]
print('%s rows' % distinct_balanced_d_speed.shape[0])
display(distinct_balanced_d_speed.head())
print('DISTINCT BALANCED SPEED')
distinct_balanced_speed = training_df[(training_df.IN_CR == 0) &
                                      (training_df.IN_CS == 1) &
                                      (training_df.IN_DS == 1)]\
                          .drop_duplicates(subset=OUT_OPTIONS)
                          [OUT_OPTIONS + TEST_FEATURES]
```

```
print('%s rows' % distinct_balanced_speed.shape[0])
display(distinct_balanced_speed.head())
print('DISTINCT BALANCED')
distinct_balanced = training_df[(training_df.IN_CR == 1) &
                                (training_df.IN_CS == 1) &
                                (training_df.IN_DS == 1)]\
                    .drop_duplicates(subset=OUT_OPTIONS)
                    [OUT_OPTIONS + TEST_FEATURES]
print('%s rows' % distinct_balanced.shape[0])
display(distinct_balanced.head())
```

```
DISTINCT MAX RATE
27 rows
```

	Block_Size	Codec	Filter	CL	CRate	CSpeed	DSpeed
3	2,048.000	zstd	shuffle	9.000	1.366	0.014	3.320
654	0.000	zstd	shuffle	9.000	49.115	0.066	4.596
661	0.000	zstd	shuffle	8.000	1.212	0.011	1.145
668	0.000	zstd	shuffle	7.000	1.179	0.013	1.050
696	2,048.000	zstd	bitshuffle	9.000	4.588	0.057	2.761

```
DISTINCT MAX C.SPEED
131 rows
```

	Block_Size	Codec	Filter	CL	CRate	CSpeed	DSpeed
1	64.000	lz4	shuffle	3.000	1.244	6.440	8.259
8	16.000	lz4	shuffle	1.000	1.233	6.758	8.311
22	0.000	lz4	shuffle	1.000	1.217	6.533	8.251
29	64.000	lz4	shuffle	1.000	1.225	6.336	8.421
50	64.000	lz4	shuffle	4.000	1.226	6.166	8.820

```
DISTINCT MAX D.SPEED
189 rows
```

	Block_Size	Codec	Filter	CL	CRate	CSpeed	DSpeed
0	32.000	blosclz	shuffle	1.000	1.108	4.402	9.849
7	32.000	blosclz	bitshuffle	2.000	1.110	3.669	9.728
14	16.000	blosclz	shuffle	4.000	1.170	2.678	9.861
21	8.000	blosclz	shuffle	2.000	1.119	4.288	9.845
28	64.000	blosclz	shuffle	3.000	1.129	3.476	9.781

```
DISTINCT BALANCED CSPEED
98 rows
```

	Block_Size	Codec	Filter	CL	CRate	CSpeed	DSpeed
5	256.000	lz4	shuffle	4.000	1.252	5.825	8.554
12	256.000	lz4	shuffle	8.000	1.255	6.033	8.148
19	128.000	lz4	shuffle	5.000	1.248	6.201	9.488
26	256.000	lz4	shuffle	6.000	1.242	5.558	8.574
33	0.000	lz4	shuffle	8.000	1.241	5.440	8.557

```
DISTINCT BALANCED DSPEED
167 rows
```

	Block_Size	Codec	Filter	CL	CRate	CSpeed	DSpeed
4	256.000	lz4hc	shuffle	8.000	1.310	0.134	8.219
11	128.000	lz4hc	shuffle	8.000	1.300	0.153	8.513
25	256.000	lz4hc	shuffle	5.000	1.292	0.190	8.783
32	128.000	lz4hc	shuffle	9.000	1.286	0.117	8.351
39	256.000	lz4hc	shuffle	7.000	1.290	0.147	8.150

```
DISTINCT BALANCED SPEED
157 rows
```

	Block_Size	Codec	Filter	CL	CRate	CSpeed	DSpeed
2	32.000	lz4	shuffle	3.000	1.239	6.157	8.908
9	16.000	lz4	shuffle	5.000	1.238	6.366	9.139
16	0.000	lz4	shuffle	4.000	1.239	6.501	9.179
23	128.000	lz4	shuffle	2.000	1.233	6.000	9.044
30	64.000	lz4	shuffle	2.000	1.226	6.196	8.831

```
DISTINCT BALANCED
148 rows
```

	Block_Size	Codec	Filter	CL	CRate	CSpeed	DSpeed
6	256.000	lz4	shuffle	5.000	1.252	5.722	8.805
13	128.000	lz4	shuffle	6.000	1.250	6.042	9.115
20	128.000	lz4	shuffle	5.000	1.248	6.201	9.488
27	256.000	lz4	shuffle	6.000	1.242	5.558	8.574
34	0.000	lz4	shuffle	8.000	1.241	5.440	8.557

In [8]:
```
distinct_total = training_df.drop_duplicates(subset=OUT_OPTIONS)
                            [OUT_OPTIONS + TEST_FEATURES]
print('%d distinct options from a total of %d' %
      (distinct_total.shape[0], 1620))
distinct_total_noblock = distinct_total.drop_duplicates(subset=OUT_OPTIONS[1:4])
print('%d distinct options from a total of %d' %
      (distinct_total_noblock.shape[0], 162))
print('Distinct codecs %d' %
      distinct_total.drop_duplicates(subset=['Codec']).shape[0])
print('Distinct filters %d' %
      distinct_total.drop_duplicates(subset=['Filter']).shape[0])
```

```
print('Distinct CL %d' %
      distinct_total.drop_duplicates(subset=['CL']).shape[0])
print('Distinct block sizes %d' %
      distinct_total.drop_duplicates(subset=['Block_Size']).shape[0])
display(distinct_total.describe())
```

```
487 distinct options from a total of 1620
96 distinct options from a total of 162
Distinct codecs 5
Distinct filters 3
Distinct CL 9
Distinct block sizes 10
```

	Block_Size	CL	CRate	CSpeed	DSpeed
count	487.000	487.000	487.000	487.000	487.000
mean	311.737	5.335	327.070	6.583	13.948
std	553.841	2.583	1,585.596	6.718	11.312
min	0.000	1.000	1.101	0.004	0.474
25%	16.000	3.000	1.268	0.763	8.743
50%	64.000	6.000	4.112	4.860	10.421
75%	256.000	8.000	49.268	9.370	14.523
max	2,048.000	9.000	10,645.442	23.848	86.345

Zlib queda descartado dado que nunca es seleccionado como óptimo.

In [9]:
```
display(training_df[training_df.Codec == 'snappy']
        [IN_USER + TEST_FEATURES + OUT_OPTIONS])
```

	IN_CR	IN_CS	IN_DS	CRate	CSpeed	DSpeed	Block_Size	Codec	Filter	CL
1681	0.000	1.000	0.000	21.195	20.349	9.824	0.000	snappy	noshuffle	7.000
4291	0.000	0.000	1.000	11.049	7.158	13.803	128.000	snappy	noshuffle	3.000

Snappy ha sido seleccionado en dos ocasiones. Por tanto podríamos considerar que tenemos 488/1080 opciones totales y sin contar el tamaño de bloque 97/108.

In [10]:
```
print('%d blosclz classes from 270' %
      distinct_total[distinct_total.Codec == 'blosclz'].shape[0])
print('%d lz4 classes from 270' %
      distinct_total[distinct_total.Codec == 'lz4'].shape[0])
print('%d lz4hc classes from 270' %
      distinct_total[distinct_total.Codec == 'lz4hc'].shape[0])
print('%d zstd classes from 270' %
      distinct_total[distinct_total.Codec == 'zstd'].shape[0])
```

```
144 blosclz classes from 270
168 lz4 classes from 270
102 lz4hc classes from 270
71 zstd classes from 270
```

Debido a que Snappy solo es seleccionado en dos ocasiones lo consideraremos como datos atípicos y por tanto los sustituimos por la segunda mejor opción.

In [11]:
```
# ELIMINAMOS SNAPPY
for i, row in (training_df[training_df.Codec == 'snappy']).iterrows():
    aux = df[(df.Filename == row['Filename']) &
             (df.DataSet == row['DataSet']) &
             (df.Table == row['Table']) &
             (df.Chunk_Number == row['Chunk_Number']) &
             (df.Codec != 'snappy')]
    i_max_crate, i_max_c_speed, i_max_d_speed = aux['CRate'].idxmax(),
                                                aux['CSpeed'].idxmax(),\
                                                aux['DSpeed'].idxmax()
    max_crate, max_c_speed, max_d_speed = (aux.ix[i_max_crate]['CRate'],
                                           aux.ix[i_max_c_speed]['CSpeed'],
                                           aux.ix[i_max_d_speed]['DSpeed'])

    min_crate, min_c_speed, min_d_speed = (aux['CRate'].min(),
                                           aux['CSpeed'].min(),
                                           aux['DSpeed'].min())
    # NORMALIZED COLUMNS
    aux = aux.assign(N_CRate=(aux['CRate'] - min_crate)
                     / (max_crate - min_crate),
                     N_CSpeed=(aux['CSpeed'] - min_c_speed)
                     / (max_c_speed - min_c_speed),
                     N_DSpeed=(aux['DSpeed'] - min_d_speed)
                     / (max_d_speed - min_d_speed))
    aux['Distance'] = row['IN_CR']*(aux['N_CRate'] - 1)**2
                      + row['IN_DS']*(aux['N_DSpeed'] - 1)**2
                      + row['IN_CS']*(aux['N_CSpeed'] - 1)**2
    index = aux['Distance'].idxmin()
    training_df.loc[i, TEST_FEATURES + OUT_OPTIONS] = \
        aux.ix[index][TEST_FEATURES + OUT_OPTIONS]
```

Tamaño de bloque automático

In [12]:
```
%%time
count = training_df[training_df.Block_Size == 0].shape[0]
for i, row in training_df.iterrows():
    block = row['Block_Size']
    aux = df[(df.Filename == row['Filename']) &
             (df.DataSet == row['DataSet']) &
             (df.Table == row['Table']) &
             (df.Chunk_Number == row['Chunk_Number']) &
             (df.Codec == row['Codec']) &
             (df.Filter == row['Filter']) &
             (df.CL == row["CL"])]
```

```
    crate = aux[aux.Block_Size == 0]['CRate'].values[0]
    auto_block = aux[(aux.CRate == crate) &
                     (aux.Block_Size != 0)]['Block_Size'].values[0]
    if block != 0:
        if auto_block == block:
            count += 1
    else:
        training_df.loc[i, 'Block_Size'] = auto_block
```

```
CPU times: user 24min 29s, sys: 284 ms, total: 24min 29s
Wall time: 24min 29s
```

In [13]:
```
print("%d from %d --> %d %%" %
      (count, training_df.shape[0], count / training_df.shape[0] * 100))
```

```
1372 from 5075 --> 27 %
```

In [14]:
```
training_df.drop_duplicates(subset=['Block_Size'])
```

	BLZ_CRate	BLZ_CSpeed	BLZ_DSpeed	Block_Size	CL	CRate	CSpeed	Chunk_Number	Chunk_Size	Codec	...	Max	Mean	Median
0	1.110	3.959	7.963	32.000	1.000	1.108	4.402	1.000	16.000	blosclz	...	27.146	5.762	3.093
1	1.110	3.959	7.963	64.000	3.000	1.244	6.440	1.000	16.000	lz4	...	27.146	5.762	3.093
3	1.110	3.959	7.963	2,048.000	9.000	1.366	0.014	1.000	16.000	zstd	...	27.146	5.762	3.093
4	1.110	3.959	7.963	256.000	8.000	1.310	0.134	1.000	16.000	lz4hc	...	27.146	5.762	3.093
8	1.110	5.626	9.007	16.000	1.000	1.233	6.758	2.000	16.000	lz4	...	27.775	6.145	4.291
11	1.110	5.626	9.007	128.000	8.000	1.300	0.153	2.000	16.000	lz4hc	...	27.775	6.145	4.291
21	1.100	5.482	9.048	8.000	2.000	1.119	4.288	4.000	16.000	blosclz	...	33.807	6.450	2.096
82	1.072	5.368	9.172	512.000	8.000	1.243	5.329	12.000	16.000	lz4	...	49.183	10.937	7.797
654	1.326	3.743	15.588	1,024.000	9.000	49.115	0.066	1.000	0.738	zstd	...	211,383.000	46,750.635	42,412.06

```
9 rows × 32 columns
```

Preparación de inputs para scikit-learn

In [15]:
```
from sklearn.preprocessing import binarize
from sklearn.preprocessing import OneHotEncoder
training_df = training_df.assign(
                is_Table=binarize(training_df['Table'].values.reshape(-1,1), 0),
                is_Columnar=binarize(training_df['Table'].values.reshape(-1,1), 1),
                is_Int=training_df['DType'].str.contains('int').astype(int),
                is_Float=training_df['DType'].str.contains('float').astype(int),
                is_String=(training_df['DType'].str.contains('S') |
                           training_df['DType'].str.contains('U')).astype(int))
import re
def aux_func(s):
    n = int(re.findall('\d+', s)[0])
    isNum = re.findall('int|float', s)
    if len(isNum) > 0:
        return n // 8
    else:
        return n
training_df['Type_Size'] = [aux_func(s) for s in training_df['DType']]
```

Preparación de outputs para scikit-learn

In [16]:
```
training_df = training_df.assign(
                 Blosclz=(training_df['Codec'] == 'blosclz').astype(int),
                 Lz4=(training_df['Codec'] == 'lz4').astype(int),
                 Lz4hc=(training_df['Codec'] == 'lz4hc').astype(int),
                 Snappy=(training_df['Codec'] == 'snappy').astype(int),
                 Zstd=(training_df['Codec'] == 'zstd').astype(int),
                 Noshuffle=(training_df['Filter'] == 'noshuffle').astype(int),
                 Shuffle=(training_df['Filter'] == 'shuffle').astype(int),
                 Bitshuffle=(training_df['Filter'] == 'bitshuffle').astype(int))
enc_cl = OneHotEncoder()
enc_cl.fit(training_df['CL'].values.reshape(-1, 1))
new_cls = enc_cl.transform(training_df['CL'].values.reshape(-1, 1)).toarray()
enc_block = OneHotEncoder()
enc_block.fit(training_df['Block_Size'].values.reshape(-1, 1))
new_blocks = enc_block.transform(training_df['Block_Size']
                                 .values.reshape(-1, 1)).toarray()
for i in range(9):
    cl_label = 'CL' + str(i+1)
    block_label = 'Block_' + str(2**(i+3))
    training_df[cl_label] = new_cls[:, i]
    training_df[block_label] = new_blocks[:, i]
```

In [17]:
```
training_df.to_csv('../data/training_data.csv', sep='\t', index=False)
```

Anexo D

Análisis de los algoritmos de clasificación

En este anexo se presentan los informes realizados para cada algoritmo de clasificación estudiado. Además también se incluyen los programas auxiliares que utilizan. Los programas son: **ml_plots.py** que incluye funciones auxiliares para las gráficas, **scoring_functions.py** donde se definen las métricas de puntuación personalizadas, **multioutput_chained.py** donde se define la clase **ChainedMultiOutputClassifier** y **nested_hypertuner.py** que realiza las validaciones cruzadas de los parámetros de cada algoritmo clasificador.

Los informes realizados con *jupyter notebook* son: **lda_logit_analysis.ipynb**, **SVM_analysis.ipynb**, **KNeig_analysis.ipynb**, **RFC_analysis.ipynb** y **classifiers_comparison.ipynb** este último es la comparativa entre los distintos algoritmos, mientras que el resto son los análisis individuales de cada uno.

D.1. Código ml_plots.py

```
1
2  from collections import Counter
3  from matplotlib import pyplot as plt
4  import numpy as np
5  from sklearn.model_selection import learning_curve
6  from sklearn.model_selection import validation_curve
7  from sklearn.model_selection import cross_val_score
8  import scoring_functions as sf
9
10
```

```
def plot_learning_curve(ax, estimator, title, X, y, scoring, ylim=None, cv=None,
                        n_jobs=1, train_sizes=np.linspace(.1, 1.0, 5)):
    """
    Generate a simple plot of the test and training learning curve.

    Parameters
    ----------
    estimator : object type that implements the "fit" and "predict" methods
        An object of that type which is cloned for each validation.

    title : string
        Title for the chart.

    X : array-like, shape (n_samples, n_features)
        Training vector, where n_samples is the number of samples and
        n_features is the number of features.

    y : array-like, shape (n_samples) or (n_samples, n_features), optional
        Target relative to X for classification or regression;
        None for unsupervised learning.

    ylim : tuple, shape (ymin, ymax), optional
        Defines minimum and maximum yvalues plotted.

    cv : int, cross-validation generator or an iterable, optional
        Determines the cross-validation splitting strategy.
        Possible inputs for cv are:
          - None, to use the default 3-fold cross-validation,
          - integer, to specify the number of folds.
          - An object to be used as a cross-validation generator.
          - An iterable yielding train/test splits.

        For integer/None inputs, if ``y`` is binary or multiclass,
        :class:`StratifiedKFold` used. If the estimator is not a classifier
        or if ``y`` is neither binary nor multiclass, :class:`KFold` is used.

        Refer :ref:`User Guide <cross_validation>` for the various
        cross-validators that can be used here.

    n_jobs : integer, optional
        Number of jobs to run in parallel (default 1).
    """
    ax.set_title(title)
    if ylim is not None:
        ax.set_ylim(*ylim)
    ax.set_xlabel("Training examples")
    ax.set_ylabel("Score %s" %scoring.__name__)
    train_sizes, train_scores, test_scores = learning_curve(
        estimator, X, y, cv=cv, n_jobs=n_jobs, train_sizes=train_sizes,
        scoring=scoring)
    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
```

```
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)
    ax.yaxis.grid(color='#CECED2')
    ax.xaxis.grid(color='#CECED2')

    ax.fill_between(train_sizes, train_scores_mean - train_scores_std,
                    train_scores_mean + train_scores_std, alpha=0.1,
                    color="#FF3B30")
    ax.fill_between(train_sizes, test_scores_mean - test_scores_std,
                    test_scores_mean + test_scores_std, alpha=0.1,
                    color="#4CD964")
    ax.plot(train_sizes, train_scores_mean, 'o-', color="#FF3B30",
            label="Training score")
    ax.plot(train_sizes, test_scores_mean, 'o-', color="#4CD964",
            label="Cross-validation score")

    plt.legend(loc="best")

    return ax


def plot_validation_curve(ax, estimator, X, Y, param_name, param_range,
                          scoring, ylim,  cv):
    train_scores, test_scores = validation_curve(estimator, X, Y,
                                                 param_name=param_name,
                                                 param_range=param_range,
                                                 cv=cv, scoring=scoring,
                                                 n_jobs=-1)

    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)

    ax.set_title("Validation Curve")
    ax.set_xlabel(param_name)
    ax.set_ylabel("Score %s" %scoring.__name__)
    ax.set_ylim(*ylim)
    lw = 2
    ax.yaxis.grid(color='#CECED2')
    ax.xaxis.grid(color='#CECED2')

    ax.plot(param_range, train_scores_mean, label="Training score",
            color="#FF9500", lw=lw)
    ax.fill_between(param_range, train_scores_mean - train_scores_std,
                    train_scores_mean + train_scores_std, alpha=0.2,
                    color="#FF9500", lw=lw)
    ax.plot(param_range, test_scores_mean, label="Cross-validation score",
            color="#007AFF", lw=lw)
    ax.fill_between(param_range, test_scores_mean - test_scores_std,
                    test_scores_mean + test_scores_std, alpha=0.2,
                    color="#007AFF", lw=lw)
```

```
    plt.legend(loc="best")

    return ax


def plot_nested_cv(non_nested_score, nested_score):
    score_difference = non_nested_score - nested_score

    print('Average difference of %.6f with std. dev. of %.6f.'
          %(score_difference.mean(), score_difference.std()))

    # Plot scores on each trial for nested and non-nested CV
    plt.figure(figsize=(20, 8))
    plt.subplot(121)
    non_nested_scores_line, = plt.plot(non_nested_score, color='#FF3B30')
    nested_line, = plt.plot(nested_score, color='#007AFF')
    plt.ylabel("score", fontsize="14")
    plt.legend([non_nested_scores_line, nested_line],
               ["Non-Nested CV", "Nested CV"],
               bbox_to_anchor=(0, 1, 0, 0))

    # Plot bar chart of the difference.
    plt.subplot(122)
    difference_plot = plt.bar(range(20), score_difference, color='#007AFF')
    plt.xlabel("Individual Trial #")
    plt.legend([difference_plot],
               ["Non-Nested CV - Nested CV Score"],
               bbox_to_anchor=(0, 1, .8, 0))
    plt.ylabel("score difference", fontsize="14")
    plt.tight_layout()
    plt.suptitle("Non-Nested and Nested Cross Validation",
                 x=.5, y=1.1, fontsize="15")

    return plt


def print_nested_winners(nested_clf, non_nested_clf):
    clf_name = non_nested_clf[0].__class__.__name__
    if (clf_name == 'RandomForestClassifier' or clf_name == 'ExtraTreesClassifier
    '):
        params = ('criterion', 'bootstrap', 'class_weight')
    elif (clf_name == "MultiOutputClassifier"):
        params = ('estimator__C', 'estimator__gamma')
    else:
        params = ('n_neighbors', 'weights')
    values = [[] for i in range(len(params))]
    for est in non_nested_clf:
        aux_dict = est.get_params()
        for i in range(len(params)):
            values[i].append(aux_dict.get(params[i]))
    print('Non Nested Winners')
    for i in range(len(params)):
```

```
        print('%s --> %s' %(params[i], Counter(values[i])))
    values = [[] for i in range(len(params))]
    for estimators in nested_clf:
        for est in estimators:
            aux_dict = est.get_params()
            for i in range(len(params)):
                values[i].append(aux_dict.get(params[i]))
    print('Nested Winners')
    for i in range(len(params)):
        print('%s --> %s' %(params[i], Counter(values[i])))


class ReportList(list):
    def _repr_html_(self):
        html = ["<table><caption><b>Report</b></caption><thead><tr><th>Name</th>\
        <th>Score</th></tr>"]
        for i, row in enumerate(self):
            html.append("<tr>")
            html.append("<td>%s</td>" %(row[0]))
            if i < 6:
                html.append(
                    "<td>%.4f +/-(%.4f)</td>" %(row[1][0],
                                                row[1][1]))
            else:
                html.append(
                    "<td>%.4f +/-(%.4f)   ~   \
                    %.4f +/-(%.4f)</td>" %(row[1][0],
                                           row[1][1],
                                           row[1][2],
                                           row[1][3]))
            html.append("</tr>")
        html.append("</table>")
        return ''.join(html)


SCORES = [sf.balanced, sf.brier, None, sf.codec, sf.filter_,
          sf.codec_filter, sf.c_level, sf.block, sf.cl_block]
NICE_SCORES = [sf.c_level_nice, sf.block_nice, sf.cl_block_nice]


def cross_val_report(estimator, cv, X, y, no_brier=False):
    report = ReportList()
    for i, scoring in enumerate(SCORES):
        if scoring is None:
            name = 'normal'
            scores = cross_val_score(estimator, cv=cv, X=X, y=y,
                                     scoring=scoring)
            values = [scores.mean(), scores.std()]
        elif no_brier and scoring.__name__ == 'brier':
            values = [0, 0]
            name = scoring.__name__
        else:
```

```
            scores = cross_val_score(estimator, cv=cv, X=X, y=y,
                                     scoring=scoring)
            values = [scores.mean(), scores.std()]
            name = scoring.__name__
        if (i > 5):
            scores = cross_val_score(estimator, cv=cv, X=X, y=y,
                                     scoring=NICE_SCORES[i - 6])
            values.extend([scores.mean(), scores.std()])
        report.append([name, values])
    return report
```

D.2. Código scoring_functions.py

```

import numpy as np
import random
import pandas as pd


def brier(predictor, X, y):
    probs = predictor.predict_proba(X)
    sorted_probs = []
    score = 0
    for i in range(len(probs[0])):
        list_aux = []
        for j in range(25):
            if probs[j][i].shape[0] > 1:
                list_aux.append(probs[j][i][1])
            else:
                list_aux.append(0)
        sorted_probs.append(list_aux)
    for i in range(y.shape[0]):
        aux = np.square(sorted_probs[i] - y[i])
        score += np.sum(aux[0:4]) + np.sum(aux[4:7]) + \
            np.sum(aux[7:16]) + np.sum(aux[16:25])
    return -score / y.shape[0]


def balanced(predictor, X, y):
    ypred = predictor.predict(X)
    score = 0
    for i in range(y.shape[0]):
        if (y[i, 0:7] == ypred[i, 0:7]).all():
            score += 0.5
        score += (8 - abs(np.argmax(y[i, 7:16] == 1) -
                          np.argmax(ypred[i, 7:16] == 1)))**2 / 64 * 0.25
        score += (8 - abs(np.argmax(y[i, 16:25] == 1) -
                          np.argmax(ypred[i, 16:25] == 1)))**2 / 64 * 0.25
    return score / y.shape[0]

```

```
38
39 def codec_filter(predictor, X, y):
40     ypred = predictor.predict(X)
41     score = 0
42     for i in range(y.shape[0]):
43         if (y[i, 0:7] == ypred[i, 0:7]).all():
44             score += 1
45     return score / y.shape[0]
46
47
48 def codec(predictor, X, y):
49     ypred = predictor.predict(X)
50     score = 0
51     for i in range(y.shape[0]):
52         if (y[i, 0:4] == ypred[i, 0:4]).all():
53             score += 1
54     return score / y.shape[0]
55
56
57 def filter_(predictor, X, y):
58     ypred = predictor.predict(X)
59     score = 0
60     for i in range(y.shape[0]):
61         if (y[i, 4:7] == ypred[i, 4:7]).all():
62             score += 1
63     return score / y.shape[0]
64
65
66 def c_level(predictor, X, y):
67     ypred = predictor.predict(X)
68     score = 0
69     for i in range(y.shape[0]):
70         if (y[i, 7:16] == ypred[i, 7:16]).all():
71             score += 1
72     return score / y.shape[0]
73
74
75 def c_level_nice(predictor, X, y):
76     ypred = predictor.predict(X)
77     score = 0
78     for i in range(y.shape[0]):
79         score += (8 - abs(np.argmax(y[i, 7:16] == 1) -
80                           np.argmax(ypred[i, 7:16] == 1)))**2 / 64
81     return score / y.shape[0]
82
83
84 def block(predictor, X, y):
85     ypred = predictor.predict(X)
86     score = 0
87     for i in range(y.shape[0]):
88         if (y[i, 16:25] == ypred[i, 16:25]).all():
89             score += 1
```

```
    return score / y.shape[0]


def block_nice(predictor, X, y):
    ypred = predictor.predict(X)
    score = 0
    for i in range(y.shape[0]):
        score += (8 - abs(np.argmax(y[i, 16:25] == 1) -
                          np.argmax(ypred[i, 16:25] == 1)))**2 / 64
    return score / y.shape[0]


def cl_block(predictor, X, y):
    ypred = predictor.predict(X)
    score = 0
    for i in range(y.shape[0]):
        if (y[i, 7:25] == ypred[i, 7:25]).all():
            score += 1
    return score / y.shape[0]


def cl_block_nice(predictor, X, y):
    ypred = predictor.predict(X)
    score = 0
    for i in range(y.shape[0]):
        score += (8 - abs(np.argmax(y[i, 7:16] == 1) -
                          np.argmax(ypred[i, 7:16] == 1)))**2 / 64 * 0.5
        score += (8 - abs(np.argmax(y[i, 16:25] == 1) -
                          np.argmax(ypred[i, 16:25] == 1)))**2 / 64 * 0.5
    return score / y.shape[0]
```

D.3. Código multioutput_chained.py

```

import numpy as np

from abc import ABCMeta
from sklearn.base import BaseEstimator, clone
from sklearn.base import RegressorMixin, ClassifierMixin
from sklearn.utils.fixes import parallel_helper
from sklearn.utils import check_array, check_X_y
from sklearn.utils.validation import check_is_fitted, has_fit_parameter
from sklearn.externals.joblib import Parallel, delayed
from sklearn.externals import six

__all__ = ["ChainedMultiOutputRegressor", "ChainedMultiOutputClassifier"]


def _fit_estimator(estimator, X, y, sample_weight=None):
    estimator = clone(estimator)
```

```
    if sample_weight is not None:
        estimator.fit(X, y, sample_weight=sample_weight)
    else:
        estimator.fit(X, y)
    return estimator


class MultiOutputEstimator(six.with_metaclass(ABCMeta, BaseEstimator)):

    def __init__(self, estimator, n_jobs=1):
        self.estimator = estimator
        self.n_jobs = n_jobs

    def fit(self, X, y, sample_weight=None):
        """ Fit the model to data.
        Fit a separate model for each output variable.
        Parameters
        ----------
        X : (sparse) array-like, shape (n_samples, n_features)
            Data.
        y : (sparse) array-like, shape (n_samples, n_outputs)
            Multi-output targets. An indicator matrix turns on multilabel
            estimation.
        sample_weight : array-like, shape = (n_samples) or None
            Sample weights. If None, then samples are equally weighted.
            Only supported if the underlying regressor supports sample
            weights.
        Returns
        -------
        self : object
            Returns self.
        """

        if not hasattr(self.estimator, "fit"):
            raise ValueError(
                "The base estimator should implement a fit method")

        X, y = check_X_y(X, y,
                         multi_output=True,
                         accept_sparse=True)

        if y.ndim == 1:
            raise ValueError("y must have at least two dimensions for "
                             "multi target regression but has only one.")

        if (sample_weight is not None and
                not has_fit_parameter(self.estimator, 'sample_weight')):
            raise ValueError("Underlying regressor does not support"
                             " sample weights.")

        self.estimators_ = Parallel(n_jobs=self.n_jobs)(delayed(_fit_estimator)(
            self.estimator, np.hstack([X, y[:, :i]]), y[:, i], sample_weight) for
```

```
        i in range(y.shape[1]))
70          return self
71
72      def predict(self, X):
73          """Predict multi-output variable using a model
74           trained for each target variable.
75          Parameters
76          ----------
77          X : (sparse) array-like, shape (n_samples, n_features)
78              Data.
79          Returns
80          -------
81          y : (sparse) array-like, shape (n_samples, n_outputs)
82              Multi-output targets predicted across multiple predictors.
83              Note: Separate models are generated for each predictor.
84          """
85          check_is_fitted(self, 'estimators_')
86          if not hasattr(self.estimator, "predict"):
87              raise ValueError(
88                  "The base estimator should implement a predict method")
89
90          X = check_array(X, accept_sparse=True)
91          y = np.empty(shape=(X.shape[0], len(self.estimators_)))
92          for i, e in enumerate(self.estimators_):
93              pre = np.asarray([e.predict(np.hstack([X, y[:, :i]]))]).T
94              y[:, i] = pre[:, 0]
95          return y
96
97
98  class ChainedMultiOutputRegressor(MultiOutputEstimator, RegressorMixin):
99      """Multi target regression
100     This strategy consists of fitting one regressor per target. This is a
101     simple strategy for extending regressors that do not natively support
102     multi-target regression.
103     Parameters
104     ----------
105     estimator : estimator object
106         An estimator object implementing `fit` and `predict`.
107     n_jobs : int, optional, default=1
108         The number of jobs to run in parallel for `fit`. If -1,
109         then the number of jobs is set to the number of cores.
110         When individual estimators are fast to train or predict
111         using `n_jobs>1` can result in slower performance due
112         to the overhead of spawning processes.
113     """
114
115     def __init__(self, estimator, n_jobs=1):
116         super(ChainedMultiOutputRegressor, self).__init__(estimator, n_jobs)
117
118     def score(self, X, y, sample_weight=None):
119         """Returns the coefficient of determination R^2 of the prediction.
120         The coefficient R^2 is defined as (1 - u/v), where u is the regression
```

```
        sum of squares ((y_true - y_pred) ** 2).sum() and v is the residual
        sum of squares ((y_true - y_true.mean()) ** 2).sum().
        Best possible score is 1.0 and it can be negative (because the
        model can be arbitrarily worse). A constant model that always
        predicts the expected value of y, disregarding the input features,
        would get a R^2 score of 0.0.
        Notes
        -----
        R^2 is calculated by weighting all the targets equally using
        `multioutput='uniform_average'`.
        Parameters
        ----------
        X : array-like, shape (n_samples, n_features)
            Test samples.
        y : array-like, shape (n_samples) or (n_samples, n_outputs)
            True values for X.
        sample_weight : array-like, shape [n_samples], optional
            Sample weights.
        Returns
        -------
        score : float
            R^2 of self.predict(X) wrt. y.
        """
        # XXX remove in 0.19 when r2_score default for multioutput changes
        from sklearn.metrics import r2_score
        return r2_score(y, self.predict(X), sample_weight=sample_weight,
                        multioutput='uniform_average')


class ChainedMultiOutputClassifier(MultiOutputEstimator, ClassifierMixin):
    """Multi target classification
    This strategy consists of fitting one classifier per target. This is a
    simple strategy for extending classifiers that do not natively support
    multi-target classification
    Parameters
    ----------
    estimator : estimator object
        An estimator object implementing `fit`, `score` and `predict_proba`.
    n_jobs : int, optional, default=1
        The number of jobs to use for the computation. If -1 all CPUs are used.
        If 1 is given, no parallel computing code is used at all, which is
        useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are
        used. Thus for n_jobs = -2, all CPUs but one are used.
        The number of jobs to use for the computation.
        It does each target variable in y in parallel.
    Attributes
    ----------
    estimators_ : list of `n_output` estimators
        Estimators used for predictions.
    """

    def __init__(self, estimator, n_jobs=1):
```

```
173         super(ChainedMultiOutputClassifier, self).__init__(estimator, n_jobs)
174
175     def predict_proba(self, X):
176         """Probability estimates.
177         Returns prediction probabilites for each class of each output.
178         Parameters
179         ----------
180         X : array-like, shape (n_samples, n_features)
181             Data
182         Returns
183         -------
184         T : (sparse) array-like, shape = (n_samples, n_classes, n_outputs)
185             The class probabilities of the samples for each of the outputs
186         """
187         check_is_fitted(self, 'estimators_')
188         if not hasattr(self.estimator, "predict_proba"):
189             raise ValueError("The base estimator should implement"
190                              "predict_proba method")
191         y = []
192         for e in self.estimators_:
193             y.append(np.asarray(e.predict(np.hstack([X, y]))).T)
194         results = np.dstack([estimator.predict_proba(np.hstack([X, y[:, :i]]))
195                              for i, estimator in enumerate(self.estimators_)])
196         return results
197
198     def score(self, X, y):
199         """Returns the mean accuracy on the given test data and labels.
200         Parameters
201         ----------
202         X : array-like, shape [n_samples, n_features]
203             Test samples
204         y : array-like, shape [n_samples, n_outputs]
205             True values for X
206         Returns
207         -------
208         scores : float
209             accuracy_score of self.predict(X) versus y
210         """
211         check_is_fitted(self, 'estimators_')
212         n_outputs_ = len(self.estimators_)
213         if y.ndim == 1:
214             raise ValueError("y must have at least two dimensions for "
215                              "multi target classification but has only one")
216         if y.shape[1] != n_outputs_:
217             raise ValueError("The number of outputs of Y for fit {0} and"
218                              " score {1} should be same".
219                              format(n_outputs_, y.shape[1]))
220         y_pred = self.predict(X)
221         return np.mean(np.all(y == y_pred, axis=1))
```

D.4. Código nested_hypertuner.py

```
1
2  import sys
3  import time
4  import pandas as pd
5  import numpy as np
6  from sklearn.ensemble import RandomForestClassifier
7  from sklearn.ensemble import ExtraTreesClassifier
8  from sklearn.preprocessing import scale
9  from sklearn.multioutput import MultiOutputClassifier
10 from sklearn.neighbors import KNeighborsClassifier
11 from sklearn.svm import SVC
12 from sklearn.model_selection import GridSearchCV
13 from sklearn.model_selection import ShuffleSplit
14 from sklearn.externals import joblib
15 from scoring_functions import balanced
16 from scoring_functions import brier
17
18
19 NUM_TRIALS = 20
20 SCORES = [balanced, brier]
21 DF = pd.read_csv('../data/training_data.csv', sep='\t')
22 IN_OPTIONS = ['IN_CR', 'IN_CS', 'IN_DS', 'is_Table', 'is_Columnar', 'is_Int',
23               'is_Float', 'is_String', 'Type_Size', 'Chunk_Size', 'Mean',
24               'Median', 'Sd', 'Skew', 'Kurt', 'Min', 'Max', 'Q1', 'Q3',
25               'N_Streaks', 'BLZ_CRate', 'BLZ_CSpeed', 'BLZ_DSpeed',
26               'LZ4_CRate', 'LZ4_CSpeed', 'LZ4_DSpeed']
27 OUT_CODEC = ['Blosclz', 'Lz4', 'Lz4hc', 'Zstd']
28 OUT_FILTER = ['Noshuffle', 'Shuffle', 'Bitshuffle']
29 OUT_LEVELS = ['CL1', 'CL2', 'CL3', 'CL4', 'CL5', 'CL6', 'CL7', 'CL8', 'CL9']
30 OUT_BLOCKS = ['Block_8', 'Block_16', 'Block_32', 'Block_64',
31               'Block_128', 'Block_256', 'Block_512', 'Block_1024',
32               'Block_2048']
33 OUT_OPTIONS = OUT_CODEC + OUT_FILTER + OUT_LEVELS + OUT_BLOCKS
34 X, Y = scale(DF[IN_OPTIONS].values), DF[OUT_OPTIONS].values
35 ESTIMATORS = []
36 ESTIMATORS.append(('KNei', KNeighborsClassifier(n_jobs=-1),
37                    {'n_neighbors': [5, 10, 15, 30, 50],
38                     'weights': ['uniform', 'distance'],
39                     }, 2))
40 ESTIMATORS.append(('SVC', MultiOutputClassifier(SVC(
41     decision_function_shape='ovr')),
42     {'estimator__C': [1, 10, 100, 1000],
43      'estimator__gamma': [0.1, 0.01, 0.001]}, 1))
44 ESTIMATORS.append(('RFC', RandomForestClassifier(
45     n_estimators=40, max_depth=14, n_jobs=-1),
46     {'criterion': ['gini', 'entropy'],
47      'bootstrap': [True, False],
48      'class_weight': [None, 'balanced']}, 2))
49 ESTIMATORS.append(('ETC', ExtraTreesClassifier(
50     n_estimators=40, max_depth=14, n_jobs=-1),
```

```
    {'criterion': ['gini', 'entropy'],
     'bootstrap': [True, False],
     'class_weight': [None, 'balanced']}, 2))
TOTAL = len(ESTIMATORS) * len(SCORES) * NUM_TRIALS - NUM_TRIALS


def main():
    count = 0
    for estimator_opt, estimator, p_grid, idx in ESTIMATORS:
        if estimator_opt == 'SVC':
            splits = 4
        else:
            splits = 10
        for score in SCORES[:idx]:
            non_nested_scores = np.zeros(NUM_TRIALS)
            non_nested_estimators = []
            nested_scores = np.zeros(NUM_TRIALS)
            nested_estimators = []
            for i in range(NUM_TRIALS):
                start_time = time.time()
                print('Estimator %s, Scorer: %s iteration %d ---- %.2f %% %
                      (estimator_opt, score.__name__, i, count / TOTAL * 100))
                inner_cv = ShuffleSplit(n_splits=splits, test_size=0.25)
                outer_cv = ShuffleSplit(n_splits=splits, test_size=0.25)
                clf = GridSearchCV(estimator=estimator, param_grid=p_grid,
                                   cv=inner_cv, n_jobs=-1, scoring=score)
                clf.fit(X, Y)
                non_nested_scores[i] = clf.best_score_
                non_nested_estimators.append(clf.best_estimator_)

                winner_estimators = []
                outer_scores = []
                for train_index, test_index in outer_cv.split(X, Y):
                    clf = GridSearchCV(estimator=estimator, param_grid=p_grid,
                                       cv=inner_cv, n_jobs=-1, scoring=score)
                    clf.fit(X[train_index], Y[train_index])
                    winner_estimators.append(clf.best_estimator_)
                    outer_scores.append(
                        score(clf, X[test_index], Y[test_index]))
                nested_scores[i] = np.mean(outer_scores)
                nested_estimators.append(winner_estimators)
                count += 1
                print('Time: %f minutes' %((time.time() - start_time) / 60))
            joblib.dump(non_nested_scores, estimator_opt +
                        'non_nested_scores_' +
                        score.__name__ + '.pkl', compress=('zlib', 3))
            joblib.dump(nested_scores, estimator_opt + 'nested_scores_' +
                        score.__name__ + '.pkl', compress=('zlib', 3))
            joblib.dump(non_nested_estimators, estimator_opt +
                        'non_nested_estimators_' + score.__name__ + '.pkl',
                        compress=('zlib', 3))
            joblib.dump(nested_estimators, estimator_opt +
```

```
103                             'nested_estimators_' +
104                             score.__name__ + '.pkl', compress=('zlib', 3))
105
106
107 if __name__ == "__main__":
108     main()
```

D.5. Cuaderno lda_logit_analysis.ipynb

LDA and Logit Analysis

In [1]:
```
%matplotlib inline
%config InlineBackend.figure_format='retina'

%load_ext autoreload
%autoreload 2

%load_ext version_information
%version_information numpy, scipy, matplotlib, pandas, scikit-learn
```

Software	Version
Python	3.5.3 64bit [GCC 4.4.7 20120313 (Red Hat 4.4.7-1)]
IPython	5.1.0
OS	Linux 4.9.16 gentoo x86_64 with debian stretch sid
numpy	1.12.1
scipy	0.19.0
matplotlib	2.0.0
pandas	0.19.2
scikit-learn	0.18.1
Fri May 05 16:05:39 2017 UTC	

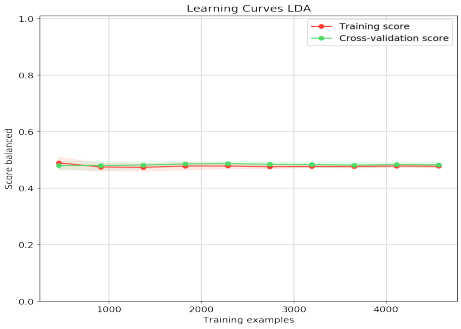
In [2]:
```
import os
import sys
sys.path.append("../src/")

from IPython.display import display
import pandas as pd
import matplotlib
import numpy as np
from matplotlib import pyplot as plt
from sklearn.externals import joblib
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import cross_val_score
from sklearn.multioutput import MultiOutputClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import SGDClassifier
from multioutput_chained import ChainedMultiOutputClassifier
from sklearn.preprocessing import scale
import ml_plots as mp
import scoring_functions as sf
import warnings
warnings.filterwarnings('ignore')

pd.options.display.float_format = '{:,.3f}'.format
matplotlib.rcParams.update({'font.size': 12})
```

In [3]:
```
SCORES = [(sf.balanced, 0, 1.01)]
IN_OPTIONS = ['IN_CR', 'IN_CS', 'IN_DS', 'is_Table', 'is_Columnar',
              'is_Int', 'is_Float', 'is_String', 'Type_Size', 'Chunk_Size',
              'Mean', 'Median', 'Sd', 'Skew', 'Kurt', 'Min', 'Max', 'Q1',
              'Q3', 'N_Streaks', 'BLZ_CRate', 'BLZ_CSpeed',
              'BLZ_DSpeed', 'LZ4_CRate', 'LZ4_CSpeed', 'LZ4_DSpeed']
OUT_CODEC = ['Blosclz', 'Lz4', 'Lz4hc', 'Zstd']
OUT_FILTER = ['Noshuffle', 'Shuffle', 'Bitshuffle']
OUT_LEVELS = ['CL1', 'CL2', 'CL3', 'CL4', 'CL5', 'CL6', 'CL7', 'CL8', 'CL9']
OUT_BLOCKS = ['Block_8', 'Block_16', 'Block_32', 'Block_64', 'Block_128',
              'Block_256', 'Block_512', 'Block_1024', 'Block_2048']
OUT_OPTIONS = OUT_CODEC + OUT_FILTER + OUT_LEVELS + OUT_BLOCKS
CUSTOM3_IN = ['IN_CS', 'IN_DS', 'IN_CR', 'BLZ_CRate', 'Sd', 'BLZ_CSpeed',
              'Mean', 'Kurt', 'Skew', 'N_Streaks', 'Max', 'Min']
CUSTOM2_IN = ['IN_CS', 'IN_DS', 'IN_CR', 'BLZ_CRate', 'Sd', 'BLZ_CSpeed',
              'Mean', 'Kurt', 'Skew', 'Max', 'Min']
CUSTOM_IN = ['IN_CS', 'IN_DS', 'IN_CR', 'BLZ_CRate', 'BLZ_CSpeed']
```

In [4]:
```
df = pd.read_csv('../data/training_data.csv', sep='\t')
X, Y = scale(df[CUSTOM3_IN].values), df[OUT_OPTIONS].values
```

LDA - Curva de aprendizaje

In [5]:
```
title = "Learning Curves LDA"
lda = MultiOutputClassifier(LinearDiscriminantAnalysis())
cv = ShuffleSplit(n_splits=10, test_size=0.1)
fig = plt.figure(figsize=(20,8))
n = 121
for score in SCORES:
    mp.plot_learning_curve(
        fig.add_subplot(n), lda, title, X, Y, scoring=score[0],
        ylim=(score[1], score[2]), cv=cv, n_jobs=-1,
        train_sizes=np.linspace(.1, 1.0, 10))
    n += 1
```



LDA - Resultados

In [6]:
```
print('Normal')
cv = ShuffleSplit(n_splits=10, test_size=0.5, random_state=1)
lda = MultiOutputClassifier(LinearDiscriminantAnalysis())
display(mp.cross_val_report(lda, cv, X, Y, True))
print('Chained')
lda = ChainedMultiOutputClassifier(LinearDiscriminantAnalysis())
display(mp.cross_val_report(lda, cv, X, Y, True))
```

Normal

Report

Name	Score
balanced	0.4796 +/-(0.0082)
brier	0.0000 +/-(0.0000)
normal	0.0610 +/-(0.0059)
codec	0.7032 +/-(0.0172)
filter_	0.5821 +/-(0.0085)
codec_filter	0.4138 +/-(0.0142)
c_level	0.1937 +/-(0.0105) ~ 0.5667 +/-(0.0051)
block	0.1455 +/-(0.0044) ~ 0.5241 +/-(0.0045)
cl_block	0.1029 +/-(0.0054) ~ 0.5454 +/-(0.0041)

Chained

Report

Name	Score
balanced	0.6150 +/-(0.0110)
brier	0.0000 +/-(0.0000)
normal	0.1247 +/-(0.0089)
codec	0.7908 +/-(0.0064)
filter_	0.6708 +/-(0.0083)
codec_filter	0.5589 +/-(0.0130)
c_level	0.3127 +/-(0.0136) ~ 0.6129 +/-(0.0188)
block	0.3558 +/-(0.0116) ~ 0.7293 +/-(0.0105)
cl_block	0.1807 +/-(0.0105) ~ 0.6711 +/-(0.0139)

In [7]:
```
CUSTOM2_IN = ['IN_CS', 'IN_DS', 'IN_CR', 'BLZ_CRate', 'Sd', 'BLZ_CSpeed',
              'Mean', 'Kurt', 'Skew', 'Max', 'Min']
X = scale(df[CUSTOM2_IN].values)
print('Normal')
lda = MultiOutputClassifier(LinearDiscriminantAnalysis())
display(mp.cross_val_report(lda, cv, X, Y, True))
print('Chained')
lda = ChainedMultiOutputClassifier(LinearDiscriminantAnalysis())
display(mp.cross_val_report(lda, cv, X, Y, True))
```

Normal

Report

Name	Score
balanced	0.4778 +/-(0.0075)
brier	0.0000 +/-(0.0000)
normal	0.0597 +/-(0.0055)
codec	0.7048 +/-(0.0173)
filter_	0.5707 +/-(0.0083)
codec_filter	0.4106 +/-(0.0125)
c_level	0.1951 +/-(0.0116) ~ 0.5667 +/-(0.0050)
block	0.1376 +/-(0.0054) ~ 0.5231 +/-(0.0046)
cl_block	0.1029 +/-(0.0053) ~ 0.5449 +/-(0.0041)

Chained

Report

Name	Score
balanced	0.6144 +/-(0.0079)
brier	0.0000 +/-(0.0000)
normal	0.1314 +/-(0.0087)
codec	0.7928 +/-(0.0069)
filter_	0.6611 +/-(0.0052)
codec_filter	0.5554 +/-(0.0083)
c_level	0.3186 +/-(0.0122) ~ 0.6156 +/-(0.0179)
block	0.3553 +/-(0.0137) ~ 0.7309 +/-(0.0101)
cl_block	0.1863 +/-(0.0090) ~ 0.6733 +/-(0.0130)

In [8]:
```
CUSTOM_IN = ['IN_CS', 'IN_DS', 'IN_CR', 'BLZ_CRate', 'BLZ_CSpeed']
X = scale(df[CUSTOM_IN].values)
print('Normal')
lda = MultiOutputClassifier(LinearDiscriminantAnalysis())
display(mp.cross_val_report(lda, cv, X, Y, True))
print('Chained')
lda = ChainedMultiOutputClassifier(LinearDiscriminantAnalysis())
display(mp.cross_val_report(lda, cv, X, Y, True))
```

Normal

Report

Name	Score
balanced	0.4549 +/-(0.0092)
brier	0.0000 +/-(0.0000)
normal	0.0500 +/-(0.0059)
codec	0.7039 +/-(0.0150)
filter_	0.5250 +/-(0.0105)
codec_filter	0.3653 +/-(0.0156)
c_level	0.1825 +/-(0.0094) ~ 0.5669 +/-(0.0052)
block	0.1361 +/-(0.0054) ~ 0.5222 +/-(0.0049)
cl_block	0.1022 +/-(0.0038) ~ 0.5446 +/-(0.0043)

Chained

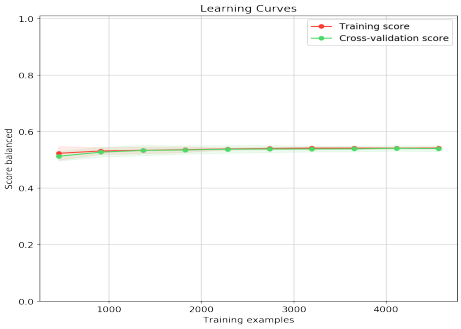
Report

Name	Score
balanced	0.6016 +/-(0.0120)
brier	0.0000 +/-(0.0000)
normal	0.1209 +/-(0.0068)
codec	0.7943 +/-(0.0063)
filter_	0.6451 +/-(0.0092)
codec_filter	0.5316 +/-(0.0141)
c_level	0.3156 +/-(0.0139) ~ 0.6120 +/-(0.0249)
block	0.3574 +/-(0.0140) ~ 0.7311 +/-(0.0112)
cl_block	0.1867 +/-(0.0096) ~ 0.6715 +/-(0.0163)

Logit - Curva de aprendizaje

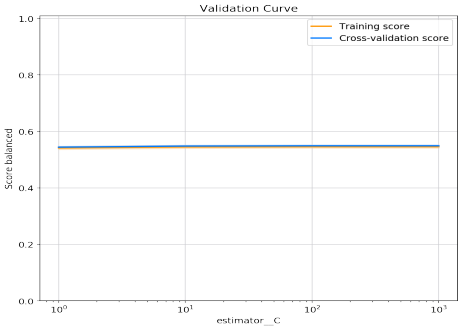
In [9]:
```
df = pd.read_csv('../data/training_data.csv', sep='\t')
X, Y = scale(df[CUSTOM3_IN].values), df[OUT_OPTIONS].values
```

In [10]:
```
title = "Learning Curves"
lda = MultiOutputClassifier(LogisticRegression())
cv = ShuffleSplit(n_splits=10, test_size=0.1)
fig = plt.figure(figsize=(20,8))
n = 121
for score in SCORES:
    mp.plot_learning_curve(
        fig.add_subplot(n), lda, title, X, Y, scoring=score[0],
        ylim=(score[1], score[2]), cv=cv, n_jobs=-1,
        train_sizes=np.linspace(.1, 1.0, 10))
    n += 1
```



Logit - Curva de validación

In [11]:
```
PARAM_NAMES = ['estimator__C']
PARAM_RANGES = [[1, 10, 100, 1000]]
cv = ShuffleSplit(n_splits=10, test_size=0.5)
clf = MultiOutputClassifier(LogisticRegression(solver='newton-cg'))
fig = plt.figure(figsize=(20, 8))
n = 121
for score in SCORES:
    ax = mp.plot_validation_curve(
        fig.add_subplot(n), clf, X, Y, param_name=PARAM_NAMES[0],
        param_range=PARAM_RANGES[0], cv=cv, scoring=score[0],
        ylim=(score[1], score[2]))
    ax.semilogx()
    n += 1
```



Logit - Resultados

In [12]:
```
print('Normal')
clf = MultiOutputClassifier(LogisticRegression(solver='newton-cg'))
display(mp.cross_val_report(clf, cv, X, Y, True))
print('Chained')
clf = ChainedMultiOutputClassifier(LogisticRegression(solver='newton-cg'))
display(mp.cross_val_report(clf, cv, X, Y, True))
```

```
Normal
```

Report

Name	Score
balanced	0.5397 +/-(0.0024)
brier	0.0000 +/-(0.0000)
normal	0.0810 +/-(0.0053)
codec	0.7857 +/-(0.0072)
filter_	0.6887 +/-(0.0069)
codec_filter	0.5394 +/-(0.0085)
c_level	0.1878 +/-(0.0150) ~ 0.5624 +/-(0.0042)
block	0.1438 +/-(0.0059) ~ 0.5229 +/-(0.0051)
cl_block	0.1015 +/-(0.0037) ~ 0.5416 +/-(0.0042)

```
Chained
```

Report

Name	Score
balanced	0.6204 +/-(0.0079)

brier	0.0000 +/-(0.0000)
normal	0.1454 +/-(0.0076)
codec	0.8460 +/-(0.0085)
filter_	0.7440 +/-(0.0075)
codec_filter	0.6396 +/-(0.0115)
c_level	0.3435 +/-(0.0105) ~ 0.5519 +/-(0.0281)
block	0.3004 +/-(0.0113) ~ 0.6409 +/-(0.0148)
cl_block	0.1717 +/-(0.0078) ~ 0.5971 +/-(0.0126)

In [13]:
```
CUSTOM2_IN = ['IN_CS', 'IN_DS', 'IN_CR', 'BLZ_CRate', 'Sd', 'BLZ_CSpeed',
              'Mean', 'Kurt', 'Skew', 'Max', 'Min']
X = scale(df[CUSTOM2_IN].values)
print('Normal')
clf = MultiOutputClassifier(LogisticRegression(solver='newton-cg'))
display(mp.cross_val_report(clf, cv, X, Y, True))
print('Chained')
clf = ChainedMultiOutputClassifier(LogisticRegression(solver='newton-cg'))
display(mp.cross_val_report(clf, cv, X, Y, True))
```

Normal

Report

Name	Score
balanced	0.5388 +/-(0.0053)
brier	0.0000 +/-(0.0000)
normal	0.0816 +/-(0.0036)
codec	0.7863 +/-(0.0068)
filter_	0.6798 +/-(0.0054)
codec_filter	0.5343 +/-(0.0054)
c_level	0.1970 +/-(0.0090) ~ 0.5634 +/-(0.0055)
block	0.1433 +/-(0.0034) ~ 0.5204 +/-(0.0043)
cl_block	0.1034 +/-(0.0041) ~ 0.5417 +/-(0.0029)

Chained

Report

Name	Score
balanced	0.6147 +/-(0.0073)
brier	0.0000 +/-(0.0000)
normal	0.1416 +/-(0.0052)
codec	0.8478 +/-(0.0041)
filter_	0.7437 +/-(0.0143)
codec_filter	0.6449 +/-(0.0140)
c_level	0.3489 +/-(0.0191) ~ 0.5747 +/-(0.0178)
block	0.2876 +/-(0.0137) ~ 0.6193 +/-(0.0149)
cl_block	0.1783 +/-(0.0088) ~ 0.6061 +/-(0.0163)

In [14]:
```
CUSTOM_IN = ['IN_CS', 'IN_DS', 'IN_CR', 'BLZ_CRate', 'BLZ_CSpeed']
X = scale(df[CUSTOM_IN].values)
print('Normal')
clf = MultiOutputClassifier(LogisticRegression(solver='newton-cg'))
display(mp.cross_val_report(clf, cv, X, Y, True))
print('Chained')
clf = ChainedMultiOutputClassifier(LogisticRegression(solver='newton-cg'))
display(mp.cross_val_report(clf, cv, X, Y, True))
```

Normal

Report

Name	Score
balanced	0.5287 +/-(0.0028)
brier	0.0000 +/-(0.0000)
normal	0.0723 +/-(0.0051)
codec	0.7839 +/-(0.0068)
filter_	0.6468 +/-(0.0075)
codec_filter	0.5114 +/-(0.0071)
c_level	0.1752 +/-(0.0073) ~ 0.5651 +/-(0.0032)
block	0.1420 +/-(0.0050) ~ 0.5238 +/-(0.0038)
cl_block	0.1009 +/-(0.0033) ~ 0.5437 +/-(0.0038)

Chained

Report

Name	Score
balanced	0.6066 +/-(0.0086)
brier	0.0000 +/-(0.0000)
normal	0.1443 +/-(0.0092)
codec	0.8486 +/-(0.0072)
filter_	0.7129 +/-(0.0134)
codec_filter	0.6217 +/-(0.0069)
c_level	0.3508 +/-(0.0140) ~ 0.5666 +/-(0.0145)
block	0.2701 +/-(0.0089) ~ 0.6315 +/-(0.0202)
cl_block	0.1783 +/-(0.0065) ~ 0.5895 +/-(0.0179)

D.6. Cuaderno SVM_analysis.ipynb

Algoritmos SVM

In [1]:
```
%matplotlib inline
%config InlineBackend.figure_format='retina'

%load_ext autoreload
%autoreload 2

%load_ext version_information
%version_information numpy, scipy, matplotlib, pandas, scikit-learn
```

Software	Version
Python	3.5.3 64bit [GCC 4.4.7 20120313 (Red Hat 4.4.7-1)]
IPython	5.1.0
OS	Linux 4.9.16 gentoo x86_64 with debian stretch sid
numpy	1.12.1
scipy	0.19.0
matplotlib	2.0.0
pandas	0.19.2
scikit-learn	0.18.1
Sat May 06 19:12:34 2017 UTC	

In [2]:
```
import os
import sys
sys.path.append("../src/")

from IPython.display import display
import pandas as pd
import matplotlib
import numpy as np
from matplotlib import pyplot as plt
from sklearn.externals import joblib
from sklearn.svm import SVC, LinearSVC
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import cross_val_score
from sklearn.multioutput import MultiOutputClassifier
from multioutput_chained import ChainedMultiOutputClassifier
from sklearn.preprocessing import scale
import ml_plots as mp
import scoring_functions as sf

pd.options.display.float_format = '{:,.3f}'.format
matplotlib.rcParams.update({'font.size': 12})
```

In [3]:
```
SCORES = [(sf.balanced, 0.2, 1.01)]
IN_OPTIONS = ['IN_CR', 'IN_CS', 'IN_DS', 'is_Table', 'is_Columnar',
              'is_Int', 'is_Float', 'is_String', 'Type_Size', 'Chunk_Size',
              'Mean', 'Median', 'Sd', 'Skew', 'Kurt', 'Min', 'Max', 'Q1',
              'Q3', 'BLZ_CRate', 'BLZ_CSpeed', 'BLZ_DSpeed', 'LZ4_CRate',
              'LZ4_CSpeed', 'LZ4_DSpeed']
OUT_CODEC = ['Blosclz', 'Lz4', 'Lz4hc', 'Zstd']
OUT_FILTER = ['Noshuffle', 'Shuffle', 'Bitshuffle']
OUT_LEVELS = ['CL1', 'CL2', 'CL3', 'CL4', 'CL5', 'CL6', 'CL7', 'CL8','CL9']
OUT_BLOCKS = ['Block_8', 'Block_16', 'Block_32', 'Block_64', 'Block_128',
              'Block_256', 'Block_512', 'Block_1024', 'Block_2048']
OUT_OPTIONS = OUT_CODEC + OUT_FILTER + OUT_LEVELS + OUT_BLOCKS
```

In [4]:
```
df = pd.read_csv('../data/training_data.csv', sep='\t')
X, Y = scale(df[IN_OPTIONS].values), df[OUT_OPTIONS].values
```

```
/home/shurberto/miniconda3/lib/python3.5/site-packages/sklearn/preprocessing/data.py:160: UserWarning: Numerical issues were encountered when centering the data and might not be solved. Dataset may contain too large values. You may need to prescale your features.
  warnings.warn("Numerical issues were encountered "
```

In [5]:
```
clf = LinearSVC()
clf = MultiOutputClassifier(clf)
```

SVM rbf - Curva de aprendizaje

svc = SVC(kernel='rbf')

clf = MultiOutputClassifier(svc) title = "Learning Curves (rbf SVC)" cv = ShuffleSplit(n_splits=10, test_size=0.1) fig = plt.figure(figsize=(20,8)) n = 121 for score in SCORES: mp.plot_learning_curve(fig.add_subplot(n), clf, title, X, Y, scoring=score[0], ylim=(score[1], score[2]), cv=cv, n_jobs=-1, train_sizes=np.linspace(.1, 1.0, 10)) n += 1

SVM rbf - Validación cruzada de hiperparámetros

In [6]:
```
nested_clf = joblib.load(
    '../src/SVCnested_estimators_my_accuracy_scorer.pkl')
non_nested_clf = joblib.load(
    '../src/SVCnon_nested_estimators_my_accuracy_scorer.pkl')
mp.print_nested_winners(nested_clf, non_nested_clf)
```

```
Non Nested Winners
estimator__C --> Counter({1000: 20})
estimator__gamma --> Counter({0.1: 20})
Nested Winners
estimator__C --> Counter({1000: 61, 100: 19})
estimator__gamma --> Counter({0.1: 80})
```

In [7]:
```
del nested_clf
del non_nested_clf
```

SVM rbf - Resultados

In [8]:
```
clf1 = MultiOutputClassifier(
    SVC(C=1000, gamma=0.1, decision_function_shape='ovr'))
cv = ShuffleSplit(n_splits=5, test_size=0.25, random_state=1)
mp.cross_val_report(clf1, cv, X, Y, True)
```

Report

Name	Score
balanced	0.7861 +/-(0.0067)
brier	0.0000 +/-(0.0000)
normal	0.2380 +/-(0.0119)
codec	0.9344 +/-(0.0054)
filter_	0.9087 +/-(0.0040)
codec_filter	0.8564 +/-(0.0092)
c_level	0.4317 +/-(0.0185) ~ 0.7000 +/-(0.0112)
block	0.4017 +/-(0.0090) ~ 0.7313 +/-(0.0050)
cl_block	0.2492 +/-(0.0127) ~ 0.7157 +/-(0.0076)

In [9]:
```
CUSTOM3_IN = ['IN_CS', 'IN_DS', 'IN_CR', 'BLZ_CRate', 'Sd', 'BLZ_CSpeed',
              'Mean', 'Kurt', 'Skew', 'N_Streaks', 'Max', 'Min']
X = scale(df[CUSTOM3_IN].values)
clf1 = MultiOutputClassifier(
    SVC(C=1000, gamma=0.1, decision_function_shape='ovr'))
mp.cross_val_report(clf1, cv, X, Y, True)
```

```
/home/shurberto/miniconda3/lib/python3.5/site-packages/sklearn/preprocessing/data.py:160: UserWarning: Numerical issues were encountered when centering the data and might not be solved. Dataset may contain too large values. You may need to prescale your features.
  warnings.warn("Numerical issues were encountered "
```

Report

Name	Score
balanced	0.7061 +/-(0.0125)
brier	0.0000 +/-(0.0000)
normal	0.1831 +/-(0.0096)
codec	0.8928 +/-(0.0066)
filter_	0.8591 +/-(0.0108)
codec_filter	0.7850 +/-(0.0119)
c_level	0.3411 +/-(0.0188) ~ 0.6297 +/-(0.0113)
block	0.2994 +/-(0.0187) ~ 0.6245 +/-(0.0210)
cl_block	0.1970 +/-(0.0091) ~ 0.6271 +/-(0.0132)

In [10]:
```
CUSTOM2_IN = ['IN_CS', 'IN_DS', 'IN_CR', 'BLZ_CRate', 'Sd', 'BLZ_CSpeed',
              'Mean', 'Kurt', 'Skew', 'Max', 'Min']
X = scale(df[CUSTOM2_IN].values)
clf1 = MultiOutputClassifier(
    SVC(C=1000, gamma=0.1, decision_function_shape='ovr'))
mp.cross_val_report(clf1, cv, X, Y, True)
```

```
/home/shurberto/miniconda3/lib/python3.5/site-packages/sklearn/preprocessing/data.py:160: UserWarning: Numerical issues were encountered when centering the data and might not be solved. Dataset may contain too large values. You may need to prescale your features.
  warnings.warn("Numerical issues were encountered "
```

Report

Name	Score
balanced	0.6272 +/-(0.0084)
brier	0.0000 +/-(0.0000)
normal	0.1359 +/-(0.0113)
codec	0.8870 +/-(0.0039)
filter_	0.7325 +/-(0.0088)
codec_filter	0.6602 +/-(0.0114)
c_level	0.3024 +/-(0.0198) ~ 0.6121 +/-(0.0125)
block	0.2293 +/-(0.0065) ~ 0.5762 +/-(0.0088)
cl_block	0.1658 +/-(0.0124) ~ 0.5941 +/-(0.0088)

In [11]:
```
CUSTOM_IN = ['IN_CS', 'IN_DS', 'IN_CR', 'BLZ_CRate', 'BLZ_CSpeed']
X = scale(df[CUSTOM_IN].values)
clf1 = MultiOutputClassifier(
    SVC(C=1000, gamma=0.1, decision_function_shape='ovr'))
mp.cross_val_report(clf1, cv, X, Y, True)
```

Report

Name	Score
balanced	0.5954 +/-(0.0060)
brier	0.0000 +/-(0.0000)
normal	0.1196 +/-(0.0058)
codec	0.8821 +/-(0.0048)
filter_	0.6826 +/-(0.0075)
codec_filter	0.6156 +/-(0.0074)
c_level	0.2772 +/-(0.0133) ~ 0.5927 +/-(0.0174)
block	0.1931 +/-(0.0077) ~ 0.5577 +/-(0.0093)
cl_block	0.1417 +/-(0.0103) ~ 0.5752 +/-(0.0101)

In [12]:
```
CUSTOM_IN = ['IN_CS', 'IN_DS', 'IN_CR']
X = scale(df[CUSTOM_IN].values)
clf1 = MultiOutputClassifier(
```

```
    SVC(C=1000, gamma=0.1, decision_function_shape='ovr'))
mp.cross_val_report(clf1, cv, X, Y, True)
```

Report

Name	Score
balanced	0.3990 +/-(0.0279)
brier	0.0000 +/-(0.0000)
normal	0.0402 +/-(0.0205)
codec	0.7554 +/-(0.0048)
filter_	0.3545 +/-(0.0560)
codec_filter	0.2657 +/-(0.0493)
c_level	0.1064 +/-(0.0310) ~ 0.5645 +/-(0.0156)
block	0.1176 +/-(0.0093) ~ 0.5002 +/-(0.0115)
cl_block	0.0924 +/-(0.0076) ~ 0.5323 +/-(0.0120)

In [13]:
```
CUSTOM3_IN = ['IN_CS', 'IN_DS', 'IN_CR', 'BLZ_CRate', 'Sd', 'BLZ_CSpeed',
              'Mean', 'Kurt', 'Skew', 'N_Streaks', 'Max', 'Min']
X = scale(df[CUSTOM3_IN].values)
clf1 = ChainedMultiOutputClassifier(
    SVC(C=1000, gamma=0.1, decision_function_shape='ovr'))
mp.cross_val_report(clf1, cv, X, Y, True)
```

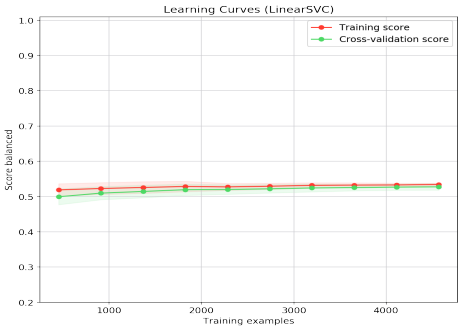
```
/home/shurberto/miniconda3/lib/python3.5/site-packages/sklearn/preprocessing/data.py:160: UserWarning: Numerical issues were encountered when centering the data and might not be solved. Dataset may contain too large values. You may need to prescale your features.
  warnings.warn("Numerical issues were encountered "
```

Report

Name	Score
balanced	0.8011 +/-(0.0037)
brier	0.0000 +/-(0.0000)
normal	0.3045 +/-(0.0119)
codec	0.9054 +/-(0.0058)
filter_	0.8801 +/-(0.0059)
codec_filter	0.8206 +/-(0.0070)
c_level	0.4994 +/-(0.0122) ~ 0.7628 +/-(0.0061)
block	0.4739 +/-(0.0149) ~ 0.8005 +/-(0.0060)
cl_block	0.3165 +/-(0.0152) ~ 0.7816 +/-(0.0052)

LinearSVC - Curva de aprendizaje

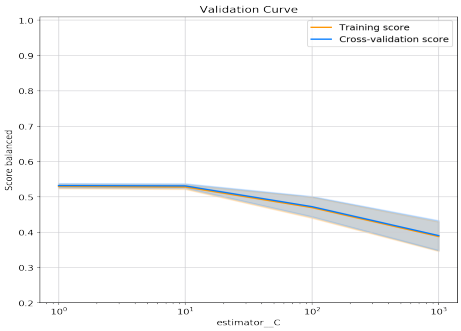
In [14]:
```
title = "Learning Curves (LinearSVC)"
cv = ShuffleSplit(n_splits=10, test_size=0.1)
fig = plt.figure(figsize=(20,8))
n = 121
for score in SCORES:
    mp.plot_learning_curve(
        fig.add_subplot(n), clf, title, X, Y, scoring=score[0],
        ylim=(score[1], score[2]), cv=cv, n_jobs=-1,
        train_sizes=np.linspace(.1, 1.0, 10))
    n += 1
```



LinearSVC - Curva de validación

In [15]:
```
PARAM_NAMES = ['estimator__C']
PARAM_RANGES = [[1, 10, 100, 1000]]
cv = ShuffleSplit(n_splits=10, test_size=0.5)
clf = MultiOutputClassifier(LinearSVC())
fig = plt.figure(figsize=(20, 8))
n = 121
for score in SCORES:
    ax = mp.plot_validation_curve(
        fig.add_subplot(n), clf, X, Y, param_name=PARAM_NAMES[0],
        param_range=PARAM_RANGES[0], cv=cv, scoring=score[0],
        ylim=(score[1], score[2]))
    ax.semilogx()
```

```
    n += 1
```



LinearSVC - Resultados

In [16]:
```
print('Normal')
clf = MultiOutputClassifier(LinearSVC())
display(mp.cross_val_report(clf, cv, X, Y, True))
print('Chained')
clf = ChainedMultiOutputClassifier(LinearSVC())
display(mp.cross_val_report(clf, cv, X, Y, True))
```

```
Normal
```

Report

Name	Score
balanced	0.5294 +/-(0.0061)
brier	0.0000 +/-(0.0000)
normal	0.0768 +/-(0.0043)
codec	0.7748 +/-(0.0121)
filter_	0.6686 +/-(0.0139)
codec_filter	0.5160 +/-(0.0155)
c_level	0.1733 +/-(0.0084) ~ 0.5621 +/-(0.0045)
block	0.1390 +/-(0.0051) ~ 0.5212 +/-(0.0035)
cl_block	0.1000 +/-(0.0051) ~ 0.5405 +/-(0.0038)

```
Chained
```

Report

Name	Score
balanced	0.6194 +/-(0.0067)
brier	0.0000 +/-(0.0000)
normal	0.1429 +/-(0.0077)
codec	0.8407 +/-(0.0130)
filter_	0.7246 +/-(0.0135)
codec_filter	0.6183 +/-(0.0136)
c_level	0.3244 +/-(0.0159) ~ 0.5429 +/-(0.0221)
block	0.3168 +/-(0.0090) ~ 0.6557 +/-(0.0162)
cl_block	0.1678 +/-(0.0059) ~ 0.6108 +/-(0.0111)

In [17]:
```
CUSTOM2_IN = ['IN_CS', 'IN_DS', 'IN_CR', 'BLZ_CRate', 'Sd', 'BLZ_CSpeed',
              'Mean', 'Kurt', 'Skew', 'Max', 'Min']
X = scale(df[CUSTOM2_IN].values)
print('Normal')
clf = MultiOutputClassifier(LinearSVC())
display(mp.cross_val_report(clf, cv, X, Y, True))
print('Chained')
clf = ChainedMultiOutputClassifier(LinearSVC())
display(mp.cross_val_report(clf, cv, X, Y, True))
```

```
/home/shurberto/miniconda3/lib/python3.5/site-packages/sklearn/preprocessing/data.py:160: UserWarning: Numerical issues were encountered when centering the data and might not be solved. Dataset may contain too large values. You may need to prescale your features.
  warnings.warn("Numerical issues were encountered "
Normal
```

Report

Name	Score
balanced	0.5305 +/-(0.0070)
brier	0.0000 +/-(0.0000)
normal	0.0710 +/-(0.0070)
codec	0.7675 +/-(0.0076)
filter_	0.6630 +/-(0.0161)
codec_filter	0.5220 +/-(0.0124)
c_level	0.1724 +/-(0.0049) ~ 0.5647 +/-(0.0062)
block	0.1384 +/-(0.0046) ~ 0.5223 +/-(0.0043)

cl_block	0.1036 +/-(0.0035)	~	0.5368 +/-(0.0034)

Chained

Report

Name	Score
balanced	0.6208 +/-(0.0106)
brier	0.0000 +/-(0.0000)
normal	0.1435 +/-(0.0076)
codec	0.8437 +/-(0.0088)
filter_	0.7249 +/-(0.0169)
codec_filter	0.6210 +/-(0.0135)
c_level	0.3272 +/-(0.0101) ~ 0.5428 +/-(0.0246)
block	0.3063 +/-(0.0088) ~ 0.6535 +/-(0.0206)
cl_block	0.1712 +/-(0.0090) ~ 0.6138 +/-(0.0077)

In [18]:
```
CUSTOM_IN = ['IN_CS', 'IN_DS', 'IN_CR', 'BLZ_CRate', 'BLZ_CSpeed']
X = scale(df[CUSTOM_IN].values)
print('Normal')
clf = MultiOutputClassifier(LinearSVC())
display(mp.cross_val_report(clf, cv, X, Y, True))
print('Chained')
clf = ChainedMultiOutputClassifier(LinearSVC())
display(mp.cross_val_report(clf, cv, X, Y, True))
```

Normal

Report

Name	Score
balanced	0.5197 +/-(0.0046)
brier	0.0000 +/-(0.0000)
normal	0.0688 +/-(0.0052)
codec	0.7777 +/-(0.0063)
filter_	0.6350 +/-(0.0094)
codec_filter	0.4986 +/-(0.0121)
c_level	0.1657 +/-(0.0045) ~ 0.5626 +/-(0.0066)
block	0.1363 +/-(0.0042) ~ 0.5204 +/-(0.0041)
cl_block	0.1002 +/-(0.0034) ~ 0.5416 +/-(0.0043)

Chained

Report

Name	Score
balanced	0.6050 +/-(0.0074)
brier	0.0000 +/-(0.0000)
normal	0.1520 +/-(0.0117)
codec	0.8480 +/-(0.0091)
filter_	0.6983 +/-(0.0136)
codec_filter	0.6115 +/-(0.0093)
c_level	0.3255 +/-(0.0123) ~ 0.5523 +/-(0.0307)
block	0.3167 +/-(0.0151) ~ 0.6641 +/-(0.0235)
cl_block	0.1704 +/-(0.0077) ~ 0.6031 +/-(0.0189)

D.7. Cuaderno KNeig_analysis.ipynb

Vecinos más próximos

In [1]:
```
%matplotlib inline
%config InlineBackend.figure_format='retina'

%load_ext autoreload
%autoreload 2

%load_ext version_information
%version_information numpy, scipy, matplotlib, pandas, scikit-learn
```

Software	Version
Python	3.5.3 64bit [GCC 4.4.7 20120313 (Red Hat 4.4.7-1)]
IPython	5.1.0
OS	Linux 4.9.16 gentoo x86_64 with debian stretch sid
numpy	1.12.1
scipy	0.19.0
matplotlib	2.0.0
pandas	0.19.2
scikit-learn	0.18.1
Sat May 06 10:20:07 2017 UTC	

In [2]:
```
import os
import sys
sys.path.append("../src/")

from IPython.display import display
import pandas as pd
import matplotlib
import numpy as np
from matplotlib import pyplot as plt
from sklearn.externals import joblib
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import cross_val_score
from sklearn.multioutput import MultiOutputClassifier
from multioutput_chained import ChainedMultiOutputClassifier
from sklearn.preprocessing import scale
import ml_plots as mp
import scoring_functions as sf
import warnings
warnings.filterwarnings('ignore')

pd.options.display.float_format = '{:,.3f}'.format
matplotlib.rcParams.update({'font.size': 12})
```

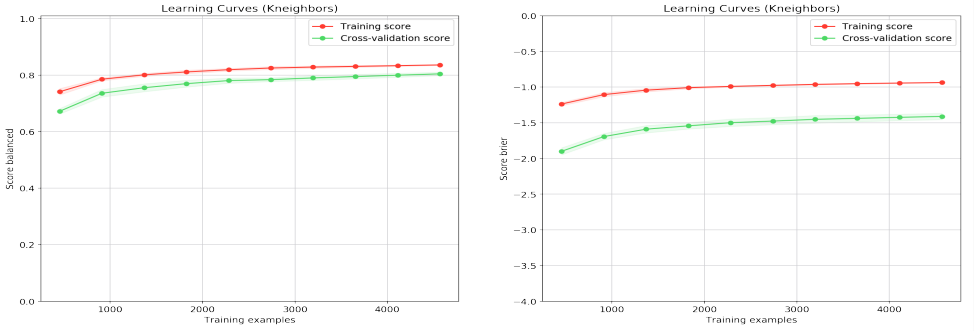
In [3]:
```
SCORES = [(sf.balanced, 0, 1.01), (sf.brier, -4, 0)]
IN_OPTIONS = ['IN_CR', 'IN_CS', 'IN_DS', 'is_Table', 'is_Columnar',
              'is_Int', 'is_Float', 'is_String', 'Type_Size', 'Chunk_Size',
              'Mean', 'Median', 'Sd', 'Skew', 'Kurt', 'Min', 'Max', 'Q1',
              'Q3', 'N_Streaks', 'BLZ_CRate', 'BLZ_CSpeed',
              'BLZ_DSpeed', 'LZ4_CRate', 'LZ4_CSpeed', 'LZ4_DSpeed']
OUT_CODEC = ['Blosclz', 'Lz4', 'Lz4hc', 'Zstd']
OUT_FILTER = ['Noshuffle', 'Shuffle', 'Bitshuffle']
OUT_LEVELS = ['CL1', 'CL2', 'CL3', 'CL4', 'CL5', 'CL6', 'CL7', 'CL8', 'CL9']
OUT_BLOCKS = ['Block_8', 'Block_16', 'Block_32', 'Block_64', 'Block_128',
              'Block_256', 'Block_512', 'Block_1024', 'Block_2048']
OUT_OPTIONS = OUT_CODEC + OUT_FILTER + OUT_LEVELS + OUT_BLOCKS
```

In [4]:
```
df = pd.read_csv('../data/training_data.csv', sep='\t')
X, Y = scale(df[IN_OPTIONS].values), df[OUT_OPTIONS].values
```

In [5]:
```
clf = KNeighborsClassifier(weights='uniform')
```

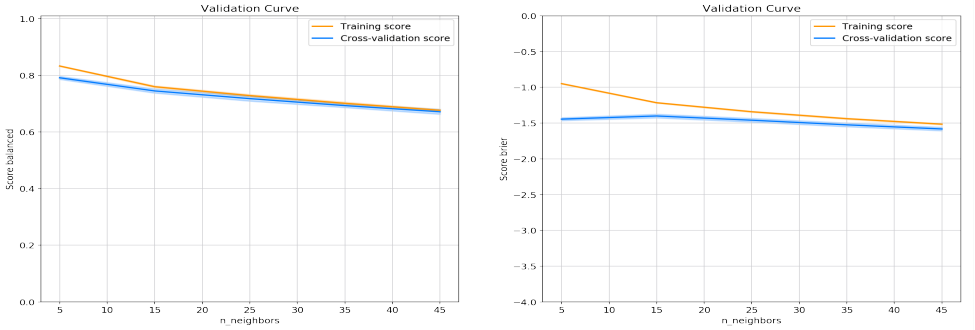
KNeigh - Curvas de aprendizaje

In [6]:
```
title = "Learning Curves (Kneighbors)"
cv = ShuffleSplit(n_splits=10, test_size=0.1)
fig = plt.figure(figsize=(20,8))
n = 121
for score in SCORES:
    mp.plot_learning_curve(
        fig.add_subplot(n), clf, title, X, Y, scoring=score[0],
        ylim=(score[1], score[2]), cv=cv, n_jobs=-1,
        train_sizes=np.linspace(.1, 1.0, 10))
    n += 1
```



KNeigh - Curvas de validación

In [7]:
```
PARAM_NAMES = ['n_neighbors']
PARAM_RANGES = [[5, 15, 25, 35, 45]]
cv = ShuffleSplit(n_splits=10, test_size=0.25)
fig = plt.figure(figsize=(20, 8))
n = 121
for score in SCORES:
    mp.plot_validation_curve(
        fig.add_subplot(n), clf, X, Y, param_name=PARAM_NAMES[0],
        param_range=PARAM_RANGES[0], cv=cv, scoring=score[0],
        ylim=(score[1], score[2]))
    n += 1
```



KNeigh - Validación cruzada de hiperparámetros

In [8]:
```
nested_clf = joblib.load(
    '../src/KNeinested_estimators_my_accuracy_scorer.pkl')
non_nested_clf = joblib.load(
    '../src/KNeinon_nested_estimators_my_accuracy_scorer.pkl')
mp.print_nested_winners(nested_clf, non_nested_clf)
```

```
Non Nested Winners
n_neighbors --> Counter({5: 20})
weights --> Counter({'distance': 20})
Nested Winners
n_neighbors --> Counter({5: 200})
weights --> Counter({'distance': 200})
```

In [9]:
```
nested_clf = joblib.load(
    '../src/KNeinested_estimators_my_brier_scorer.pkl')
non_nested_clf = joblib.load(
    '../src/KNeinon_nested_estimators_my_brier_scorer.pkl')
mp.print_nested_winners(nested_clf, non_nested_clf)
```

```
Non Nested Winners
n_neighbors --> Counter({15: 20})
weights --> Counter({'distance': 20})
Nested Winners
n_neighbors --> Counter({15: 200})
weights --> Counter({'distance': 200})
```

In [10]:
```
del nested_clf
del non_nested_clf
```

KNeigh - Resultados

In [11]:
```
clf1 = KNeighborsClassifier(n_neighbors=5, weights='uniform')
mp.cross_val_report(clf1, cv, X, Y)
```

Report

Name	Score
balanced	0.7930 +/-(0.0053)
brier	-1.4488 +/-(0.0268)
normal	0.2631 +/-(0.0123)
codec	0.9278 +/-(0.0072)
filter_	0.9264 +/-(0.0063)
codec_filter	0.8754 +/-(0.0081)
c_level	0.4482 +/-(0.0112) ~ 0.6937 +/-(0.0065)
block	0.4264 +/-(0.0109) ~ 0.7160 +/-(0.0044)
cl_block	0.2621 +/-(0.0074) ~ 0.7072 +/-(0.0047)

In [12]:
```
clf1 = KNeighborsClassifier(n_neighbors=5, weights='distance')
mp.cross_val_report(clf1, cv, X, Y)
```

Report

Name	Score
balanced	0.8051 +/-(0.0047)
brier	-1.4469 +/-(0.0220)
normal	0.2766 +/-(0.0091)
codec	0.9361 +/-(0.0067)
filter_	0.9318 +/-(0.0048)
codec_filter	0.8843 +/-(0.0087)
c_level	0.4658 +/-(0.0104) ~ 0.7157 +/-(0.0087)
block	0.4355 +/-(0.0076) ~ 0.7286 +/-(0.0047)
cl_block	0.2813 +/-(0.0130) ~ 0.7221 +/-(0.0070)

In [13]:
```
CUSTOM3_IN = ['IN_CS', 'IN_DS', 'IN_CR', 'BLZ_CRate', 'Sd', 'BLZ_CSpeed',
              'Mean', 'Kurt', 'Skew', 'N_Streaks', 'Max', 'Min']
X = scale(df[CUSTOM3_IN].values)
clf1 = KNeighborsClassifier(n_neighbors=5, weights='distance')
mp.cross_val_report(clf1, cv, X, Y, True)
```

Report

Name	Score
balanced	0.7883 +/-(0.0062)
brier	0.0000 +/-(0.0000)
normal	0.2746 +/-(0.0072)
codec	0.9299 +/-(0.0061)
filter_	0.9016 +/-(0.0083)
codec_filter	0.8486 +/-(0.0099)
c_level	0.4650 +/-(0.0070) ~ 0.7147 +/-(0.0079)
block	0.4355 +/-(0.0088) ~ 0.7366 +/-(0.0069)
cl_block	0.2944 +/-(0.0082) ~ 0.7265 +/-(0.0058)

In [14]:
```
CUSTOM2_IN = ['IN_CS', 'IN_DS', 'IN_CR', 'BLZ_CRate', 'Sd', 'BLZ_CSpeed',
              'Mean', 'Kurt', 'Skew', 'Max', 'Min']
X = scale(df[CUSTOM2_IN].values)
clf1 = KNeighborsClassifier(n_neighbors=5, weights='distance')
mp.cross_val_report(clf1, cv, X, Y, True)
```

Report

Name	Score
balanced	0.7559 +/-(0.0095)
brier	0.0000 +/-(0.0000)
normal	0.2416 +/-(0.0057)
codec	0.9124 +/-(0.0071)
filter_	0.8548 +/-(0.0113)
codec_filter	0.7964 +/-(0.0083)
c_level	0.4396 +/-(0.0084) ~ 0.6984 +/-(0.0099)
block	0.4021 +/-(0.0115) ~ 0.7172 +/-(0.0061)
cl_block	0.2504 +/-(0.0120) ~ 0.7114 +/-(0.0057)

In [15]:
```
CUSTOM_IN = ['IN_CS', 'IN_DS', 'IN_CR', 'BLZ_CRate', 'BLZ_CSpeed']
X = scale(df[CUSTOM_IN].values)
clf1 = KNeighborsClassifier(n_neighbors=5, weights='distance')
mp.cross_val_report(clf1, cv, X, Y, True)
```

Report

Name	Score
balanced	0.7130 +/-(0.0084)
brier	0.0000 +/-(0.0000)
normal	0.2284 +/-(0.0105)
codec	0.9041 +/-(0.0095)
filter_	0.7874 +/-(0.0139)
codec_filter	0.7338 +/-(0.0105)
c_level	0.4225 +/-(0.0102) ~ 0.6900 +/-(0.0064)
block	0.3741 +/-(0.0073) ~ 0.7093 +/-(0.0074)
cl_block	0.2491 +/-(0.0111) ~ 0.6961 +/-(0.0077)

In [16]:
```
CUSTOM_IN = ['IN_CS', 'IN_DS', 'IN_CR', 'Mean', 'Max', 'Min']
```

```
X = scale(df[CUSTOM_IN].values)
clf1 = KNeighborsClassifier(n_neighbors=5, weights='distance')
mp.cross_val_report(clf1, cv, X, Y, True)
```

Report

Name	Score
balanced	0.6961 +/-(0.0080)
brier	0.0000 +/-(0.0000)
normal	0.1877 +/-(0.0120)
codec	0.8461 +/-(0.0089)
filter_	0.8556 +/-(0.0090)
codec_filter	0.7460 +/-(0.0077)
c_level	0.3667 +/-(0.0151) ~ 0.6509 +/-(0.0097)
block	0.3392 +/-(0.0097) ~ 0.6644 +/-(0.0164)
cl_block	0.1909 +/-(0.0129) ~ 0.6573 +/-(0.0134)

In [17]:
```
CUSTOM_IN = ['IN_CS', 'IN_DS', 'IN_CR']
X = scale(df[CUSTOM_IN].values)
clf1 = KNeighborsClassifier(n_neighbors=5, weights='distance')
mp.cross_val_report(clf1, cv, X, Y, True)
```

Report

Name	Score
balanced	0.4339 +/-(0.0245)
brier	0.0000 +/-(0.0000)
normal	0.0434 +/-(0.0207)
codec	0.6855 +/-(0.0699)
filter_	0.4243 +/-(0.0474)
codec_filter	0.3110 +/-(0.0430)
c_level	0.1856 +/-(0.0513) ~ 0.5962 +/-(0.0413)
block	0.1601 +/-(0.0277) ~ 0.5657 +/-(0.0439)
cl_block	0.0789 +/-(0.0426) ~ 0.5704 +/-(0.0375)

In [18]:
```
CUSTOM3_IN = ['IN_CS', 'IN_DS', 'IN_CR', 'BLZ_CRate', 'Sd', 'BLZ_CSpeed',
              'Mean', 'Kurt', 'Skew', 'N_Streaks', 'Max', 'Min']
X = scale(df[CUSTOM3_IN].values)
clf1 = ChainedMultiOutputClassifier(
    KNeighborsClassifier(n_neighbors=5, weights='distance'))
mp.cross_val_report(clf1, cv, X, Y, True)
```

Report

Name	Score
balanced	0.8205 +/-(0.0058)
brier	0.0000 +/-(0.0000)
normal	0.3146 +/-(0.0088)
codec	0.9298 +/-(0.0082)
filter_	0.9036 +/-(0.0095)
codec_filter	0.8549 +/-(0.0117)
c_level	0.5189 +/-(0.0079) ~ 0.7723 +/-(0.0103)
block	0.4898 +/-(0.0136) ~ 0.8039 +/-(0.0050)
cl_block	0.3318 +/-(0.0121) ~ 0.7887 +/-(0.0070)

D.8. Cuaderno RFC_analysis.ipynb

Algoritmos Random Forest

In [1]:
```
%matplotlib inline
%config InlineBackend.figure_format='retina'

%load_ext autoreload
%autoreload 2

%load_ext version_information
%version_information numpy, scipy, matplotlib, pandas, scikit-learn
```

Software	Version
Python	3.5.3 64bit [GCC 4.4.7 20120313 (Red Hat 4.4.7-1)]
IPython	5.1.0
OS	Linux 4.9.16 gentoo x86_64 with debian stretch sid
numpy	1.12.1
scipy	0.19.0
matplotlib	2.0.0
pandas	0.19.2
scikit-learn	0.18.1
Sat May 06 16:57:06 2017 UTC	

In [2]:
```
import os
import sys
sys.path.append("../src/")

from IPython.display import display
import pandas as pd
import matplotlib
import numpy as np
from matplotlib import pyplot as plt
from sklearn.externals import joblib
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import cross_val_score
from sklearn.multioutput import MultiOutputClassifier
from multioutput_chained import ChainedMultiOutputClassifier
import ml_plots as mp
import scoring_functions as sf
import warnings
warnings.filterwarnings('ignore')

pd.options.display.float_format = '{:,.3f}'.format
matplotlib.rcParams.update({'font.size': 12})
```

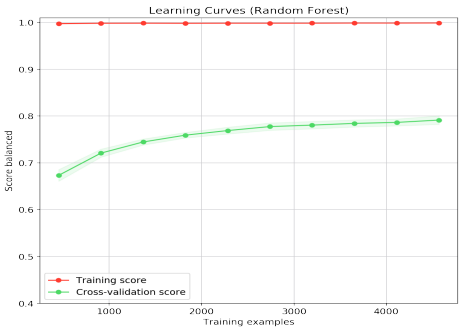
In [3]:
```
SCORES = [(sf.balanced, 0.4, 1.01), (sf.brier, -2, 0)]
IN_OPTIONS = ['IN_CR', 'IN_CS', 'IN_DS', 'is_Table', 'is_Columnar',
              'is_Int', 'is_Float', 'is_String', 'Type_Size', 'Chunk_Size',
              'Mean', 'Median', 'Sd', 'Skew', 'Kurt', 'Min', 'Max', 'Q1',
              'Q3', 'N_Streaks', 'BLZ_CRate', 'BLZ_CSpeed',
              'BLZ_DSpeed', 'LZ4_CRate', 'LZ4_CSpeed', 'LZ4_DSpeed']
OUT_CODEC = ['Blosclz', 'Lz4', 'Lz4hc', 'Zstd']
OUT_FILTER = ['Noshuffle', 'Shuffle', 'Bitshuffle']
OUT_LEVELS = ['CL1', 'CL2', 'CL3', 'CL4', 'CL5', 'CL6', 'CL7', 'CL8', 'CL9']
OUT_BLOCKS = ['Block_8', 'Block_16', 'Block_32', 'Block_64', 'Block_128',
              'Block_256', 'Block_512', 'Block_1024', 'Block_2048']
OUT_OPTIONS = OUT_CODEC + OUT_FILTER + OUT_LEVELS + OUT_BLOCKS
```

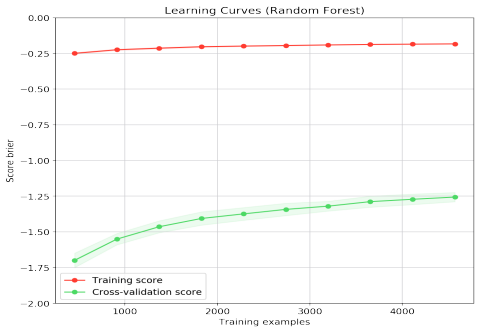
In [4]:
```
df = pd.read_csv('../data/training_data.csv', sep='\t')
X, Y = df[IN_OPTIONS].values, df[OUT_OPTIONS].values
```

In [5]:
```
rfc = RandomForestClassifier(n_estimators=50, n_jobs=-1)
```

RFC - Curvas de aprendizaje

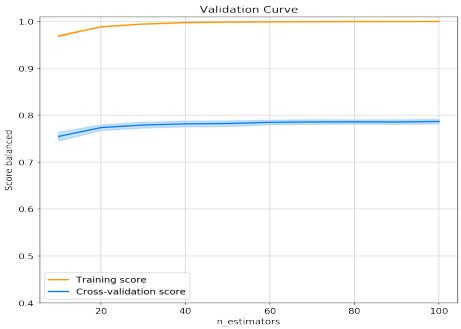
In [6]:
```
title = "Learning Curves (Random Forest)"
cv = ShuffleSplit(n_splits=10, test_size=0.1)
fig = plt.figure(figsize=(20,8))
n = 121
for score in SCORES:
    mp.plot_learning_curve(
        fig.add_subplot(n), rfc, title, X, Y, scoring=score[0],
        ylim=(score[1], score[2]), cv=cv, n_jobs=-1,
        train_sizes=np.linspace(.1, 1.0, 10))
    n += 1
```

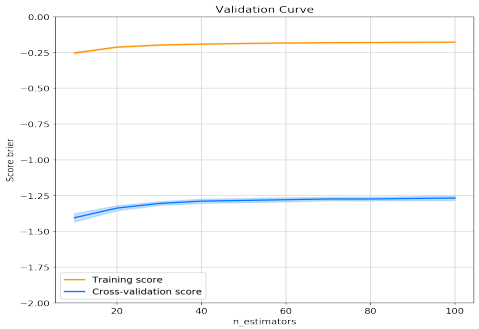


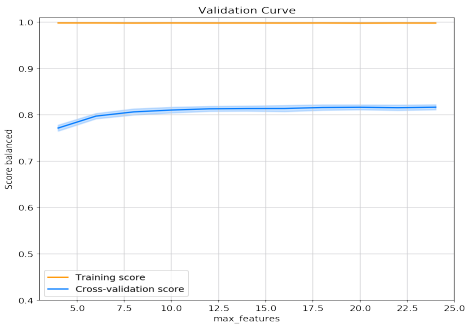


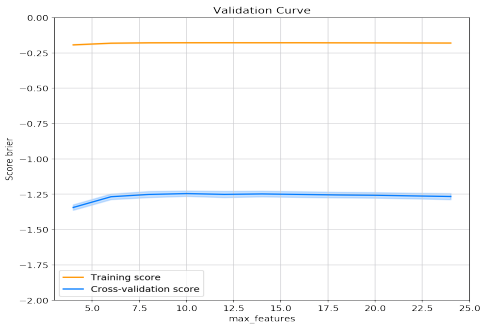
RFC - Curvas de validación

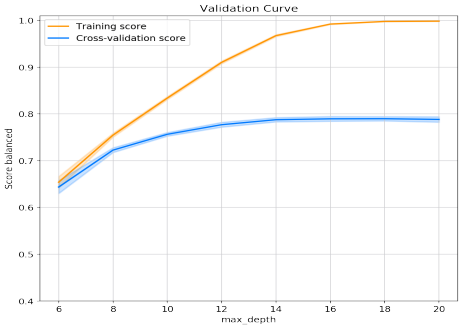
In [7]:
```
PARAM_NAMES = ['n_estimators', 'max_features', 'max_depth']
PARAM_RANGES = [[10, 20, 30, 40, 50, 60, 70, 80, 90, 100],
                [4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24],
                [6, 8, 10, 12, 14, 16, 18, 20]]
cv = ShuffleSplit(n_splits=10, test_size=0.25)
for i in range(3):
    fig = plt.figure(figsize=(20, 8))
    n = 121
    for score in SCORES:
        mp.plot_validation_curve(
            fig.add_subplot(n), rfc, X, Y, param_name=PARAM_NAMES[i],
            param_range=PARAM_RANGES[i], cv=cv, scoring=score[0],
            ylim=(score[1], score[2]))
        n += 1
```

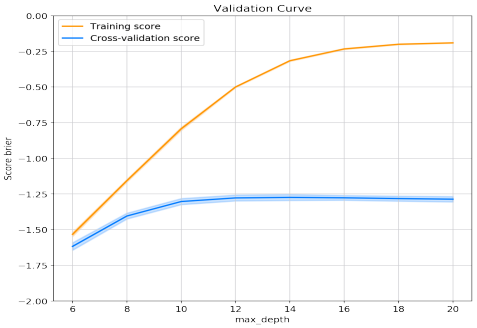












RFC - Validación cruzada de hiperparámetros

In [8]:
```
nested_clf = joblib.load(
    '../src/RFCnested_estimators_my_accuracy_scorer.pkl')
non_nested_clf = joblib.load(
    '../src/RFCnon_nested_estimators_my_accuracy_scorer.pkl')
```

In [9]:
```
mp.print_nested_winners(nested_clf, non_nested_clf)
```

```
Non Nested Winners
criterion --> Counter({'entropy': 20})
bootstrap --> Counter({False: 20})
class_weight --> Counter({None: 20})
Nested Winners
criterion --> Counter({'entropy': 191, 'gini': 9})
bootstrap --> Counter({False: 200})
class_weight --> Counter({None: 200})
```

In [10]:
```
nested_clf = joblib.load(
    '../src/RFCnested_estimators_my_brier_scorer.pkl')
non_nested_clf = joblib.load(
    '../src/RFCnon_nested_estimators_my_brier_scorer.pkl')
mp.print_nested_winners(nested_clf, non_nested_clf)
```

```
Non Nested Winners
criterion --> Counter({'gini': 16, 'entropy': 4})
bootstrap --> Counter({True: 20})
class_weight --> Counter({None: 20})
Nested Winners
criterion --> Counter({'gini': 157, 'entropy': 43})
bootstrap --> Counter({True: 200})
class_weight --> Counter({None: 200})
```

In [11]:
```
del nested_clf
del non_nested_clf
```

RFC - Resultados

In [12]:
```
clf1 = RandomForestClassifier(
    n_estimators=20, max_features=10, max_depth=12, bootstrap=False,
    criterion='entropy')
clf2 = RandomForestClassifier(
    n_estimators=20, max_features=10, max_depth=12, bootstrap=True,
    criterion='gini')
clf3 = RandomForestClassifier(
    n_estimators=20, max_features=10, max_depth=12, bootstrap=True,
    criterion='entropy')
cv = ShuffleSplit(n_splits=10, test_size=0.25, random_state=1)
```

In [13]:
```
print('Entropy')
mp.cross_val_report(clf1, cv, X, Y)
```

```
Entropy
```

```
Report
```

Name	Score
balanced	0.8196 +/-(0.0072)
brier	-1.2470 +/-(0.0435)
normal	0.2987 +/-(0.0140)
codec	0.9508 +/-(0.0041)
filter_	0.9503 +/-(0.0064)
codec_filter	0.9131 +/-(0.0075)
c_level	0.4917 +/-(0.0176) ~ 0.7143 +/-(0.0151)
block	0.4611 +/-(0.0155) ~ 0.7395 +/-(0.0054)
cl_block	0.3029 +/-(0.0135) ~ 0.7277 +/-(0.0096)

In [14]:
```
print('Entropy bootstrap')
mp.cross_val_report(clf2, cv, X, Y)
```

```
Entropy bootstrap
```

```
Report
```

Name	Score
balanced	0.7997 +/-(0.0081)

brier	-1.2344 +/-(0.0333)
normal	0.2673 +/-(0.0108)
codec	0.9485 +/-(0.0041)
filter_	0.9458 +/-(0.0066)
codec_filter	0.9076 +/-(0.0077)
c_level	0.4593 +/-(0.0211) ~ 0.6858 +/-(0.0160)
block	0.4251 +/-(0.0100) ~ 0.7022 +/-(0.0085)
cl_block	0.2734 +/-(0.0154) ~ 0.6956 +/-(0.0105)

In [15]:
```
print('Gini bootstrap')
mp.cross_val_report(clf3, cv, X, Y)
```

```
Gini bootstrap
```

Report

Name	Score
balanced	0.8062 +/-(0.0067)
brier	-1.2402 +/-(0.0332)
normal	0.2775 +/-(0.0136)
codec	0.9490 +/-(0.0056)
filter_	0.9482 +/-(0.0054)
codec_filter	0.9104 +/-(0.0077)
c_level	0.4638 +/-(0.0182) ~ 0.6922 +/-(0.0131)
block	0.4350 +/-(0.0142) ~ 0.7132 +/-(0.0096)
cl_block	0.2795 +/-(0.0158) ~ 0.7017 +/-(0.0096)

In [16]:
```
clf1.fit(X,Y)
```

```
RandomForestClassifier(bootstrap=False, class_weight=None,
            criterion='entropy', max_depth=12, max_features=10,
            max_leaf_nodes=None, min_impurity_split=1e-07,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=20, n_jobs=1,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
```

In [17]:
```
my_dict = dict(zip(clf1.feature_importances_, IN_OPTIONS))
NEW_IN = []
for elem in np.sort(clf1.feature_importances_)[::-1]:
    print('%-12s --> %s' % (my_dict.get(elem), elem))
    if elem > 0.01:
        NEW_IN.append(my_dict.get(elem))
```

```
IN_CR        --> 0.148466477344
IN_CS        --> 0.130313557787
IN_DS        --> 0.10534829654
Sd           --> 0.067778712108
BLZ_CRate    --> 0.0674005819952
LZ4_CRate    --> 0.0660674689345
LZ4_CSpeed   --> 0.0400586743433
N_Streaks    --> 0.0396061864108
LZ4_DSpeed   --> 0.0373611980624
BLZ_CSpeed   --> 0.0368829565262
Mean         --> 0.0361343372503
Max          --> 0.0354673229451
Q3           --> 0.0345403385022
Kurt         --> 0.0315574961573
BLZ_DSpeed   --> 0.0278866373071
Skew         --> 0.0234427007331
is_Int       --> 0.0135220832663
Median       --> 0.0125091547104
Chunk_Size   --> 0.0106062301475
is_Float     --> 0.00897978537354
Q1           --> 0.00788054330216
Min          --> 0.00732846416934
is_String    --> 0.00450878531519
is_Table     --> 0.00285664487791
Type_Size    --> 0.00227379804071
is_Columnar  --> 0.00122150874843
```

In [18]:
```
X = df[NEW_IN].values
```

In [19]:
```
clf1 = RandomForestClassifier(
    n_estimators=20, max_features=10, max_depth=12, bootstrap=False,
    criterion='entropy')
mp.cross_val_report(clf1, cv, X, Y)
```

Report

Name	Score
balanced	0.8237 +/-(0.0080)
brier	-1.2717 +/-(0.0413)
normal	0.3052 +/-(0.0136)
codec	0.9528 +/-(0.0053)
filter_	0.9491 +/-(0.0067)
codec_filter	0.9153 +/-(0.0079)
c_level	0.4964 +/-(0.0174) ~ 0.7207 +/-(0.0147)
block	0.4631 +/-(0.0161) ~ 0.7454 +/-(0.0083)
cl_block	0.3075 +/-(0.0160) ~ 0.7327 +/-(0.0112)

In [20]:
```
clf1.fit(X, Y)
```

```
RandomForestClassifier(bootstrap=False, class_weight=None,
            criterion='entropy', max_depth=12, max_features=10,
            max_leaf_nodes=None, min_impurity_split=1e-07,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=20, n_jobs=1,
```

```
                oob_score=False, random_state=None, verbose=0,
                warm_start=False)
```

In [21]:
```
my_dict = dict(zip(clf1.feature_importances_, NEW_IN))
NEW_IN2 = []
for elem in np.sort(clf1.feature_importances_)[::-1]:
    print('%-12s --> %s' % (my_dict.get(elem), elem))
    if elem > 0.025:
        NEW_IN2.append(my_dict.get(elem))
```

```
IN_CR        --> 0.130644891879
IN_CS        --> 0.121037174859
IN_DS        --> 0.095625911333
Sd           --> 0.0875813899201
BLZ_CRate    --> 0.0834730929038
LZ4_CRate    --> 0.0819683164911
N_Streaks    --> 0.0520481216005
LZ4_CSpeed   --> 0.0494029069734
LZ4_DSpeed   --> 0.0394371475015
BLZ_CSpeed   --> 0.0373661339009
Max          --> 0.0359438571296
Mean         --> 0.0351539163797
Kurt         --> 0.0336371519158
BLZ_DSpeed   --> 0.0301338373435
Q3           --> 0.0260208413754
Skew         --> 0.0241941140644
Median       --> 0.013533189131
is_Int       --> 0.0133624624045
Chunk_Size   --> 0.00943554289396
```

In [22]:
```
X = df[NEW_IN2].values
clf1 = RandomForestClassifier(
    n_estimators=20, max_features=10, max_depth=12, bootstrap=False,
    criterion='entropy')
mp.cross_val_report(clf1, cv, X, Y)
```

Report

Name	Score
balanced	0.8237 +/-(0.0060)
brier	-1.3075 +/-(0.0511)
normal	0.3004 +/-(0.0164)
codec	0.9526 +/-(0.0059)
filter_	0.9440 +/-(0.0058)
codec_filter	0.9091 +/-(0.0103)
c_level	0.5009 +/-(0.0188) ~ 0.7256 +/-(0.0115)
block	0.4634 +/-(0.0139) ~ 0.7494 +/-(0.0077)
cl_block	0.3083 +/-(0.0152) ~ 0.7361 +/-(0.0077)

In [23]:
```
clf1.fit(X, Y)
my_dict = dict(zip(clf1.feature_importances_, NEW_IN2))
for elem in np.sort(clf1.feature_importances_)[::-1]:
    print('%-12s --> %s' % (my_dict.get(elem), elem))
```

```
IN_CR        --> 0.121871073429
IN_CS        --> 0.112380120629
Sd           --> 0.105631104076
LZ4_CRate    --> 0.0907992047207
IN_DS        --> 0.0870232074235
BLZ_CRate    --> 0.0849226989933
N_Streaks    --> 0.0597568902948
LZ4_CSpeed   --> 0.0545806485144
Kurt         --> 0.0516180420481
LZ4_DSpeed   --> 0.0491060988069
BLZ_CSpeed   --> 0.0438321764081
Mean         --> 0.0369895465782
BLZ_DSpeed   --> 0.0351976830763
Max          --> 0.034441598845
Q3           --> 0.0318499061565
```

In [24]:
```
CUSTOM3_IN = ['IN_CS', 'IN_DS', 'IN_CR', 'BLZ_CRate', 'Sd', 'BLZ_CSpeed',
              'Mean', 'Kurt', 'Skew', 'N_Streaks', 'Max', 'Min']
X = df[CUSTOM3_IN].values
clf1 = RandomForestClassifier(
    n_estimators=20, max_features=8, max_depth=12, bootstrap=False,
    criterion='entropy')
display(mp.cross_val_report(clf1, cv, X, Y))
clf1.fit(X, Y)
my_dict = dict(zip(clf1.feature_importances_, CUSTOM3_IN))
for elem in np.sort(clf1.feature_importances_)[::-1]:
    print('%-12s --> %s' % (my_dict.get(elem), elem))
```

Report

Name	Score
balanced	0.8240 +/-(0.0055)
brier	-1.3573 +/-(0.0464)
normal	0.3038 +/-(0.0118)
codec	0.9496 +/-(0.0067)
filter_	0.9415 +/-(0.0064)
codec_filter	0.9061 +/-(0.0075)
c_level	0.4999 +/-(0.0172) ~ 0.7293 +/-(0.0160)
block	0.4658 +/-(0.0137) ~ 0.7531 +/-(0.0073)
cl_block	0.3096 +/-(0.0152) ~ 0.7418 +/-(0.0091)

```
BLZ_CRate    --> 0.149275347548
IN_CR        --> 0.137693395289
Sd           --> 0.112265139683
IN_CS        --> 0.112113780803
```

```
IN_DS         --> 0.0964968848491
N_Streaks     --> 0.0891541754574
BLZ_CSpeed    --> 0.0863057496674
Kurt          --> 0.0550968792025
Mean          --> 0.0542762784394
Max           --> 0.0494618342218
Skew          --> 0.0395579950027
Min           --> 0.0183025398374
```

In [25]:
```
CUSTOM2_IN = ['IN_CS', 'IN_DS', 'IN_CR', 'BLZ_CRate', 'Sd', 'BLZ_CSpeed',
              'Mean', 'Kurt', 'Skew', 'Max', 'Min']
X = df[CUSTOM2_IN].values
clf1 = RandomForestClassifier(
    n_estimators=20, max_features=8, max_depth=12, bootstrap=False,
    criterion='entropy')
display(mp.cross_val_report(clf1, cv, X, Y))
clf1.fit(X, Y)
my_dict = dict(zip(clf1.feature_importances_, CUSTOM2_IN))
for elem in np.sort(clf1.feature_importances_)[::-1]:
    print('%-12s --> %s' % (my_dict.get(elem), elem))
```

Report

Name	Score
balanced	0.8165 +/-(0.0062)
brier	-1.4211 +/-(0.0498)
normal	0.2998 +/-(0.0096)
codec	0.9417 +/-(0.0072)
filter_	0.9380 +/-(0.0059)
codec_filter	0.8966 +/-(0.0065)
c_level	0.4965 +/-(0.0193) ~ 0.7273 +/-(0.0118)
block	0.4636 +/-(0.0102) ~ 0.7527 +/-(0.0089)
cl_block	0.3045 +/-(0.0123) ~ 0.7402 +/-(0.0080)

```
BLZ_CRate     --> 0.181184730064
Sd            --> 0.135795449367
IN_CR         --> 0.127324647265
IN_CS         --> 0.108883495718
BLZ_CSpeed    --> 0.102740443093
IN_DS         --> 0.0853819248846
Kurt          --> 0.0659618187906
Mean          --> 0.0650178728188
Max           --> 0.0585048023312
Skew          --> 0.047419608212
Min           --> 0.021785207456
```

In [26]:
```
CUSTOM_IN = ['IN_CS', 'IN_DS', 'IN_CR', 'BLZ_CRate', 'BLZ_CSpeed']
X = df[CUSTOM_IN].values
clf1 = RandomForestClassifier(
    n_estimators=20, max_features=4, max_depth=12, bootstrap=False,
    criterion='entropy')
display(mp.cross_val_report(clf1, cv, X, Y))
clf1.fit(X, Y)
my_dict = dict(zip(clf1.feature_importances_, CUSTOM_IN))
for elem in np.sort(clf1.feature_importances_)[::-1]:
    print('%-12s --> %s' % (my_dict.get(elem), elem))
```

Report

Name	Score
balanced	0.7767 +/-(0.0087)
brier	-1.6566 +/-(0.0500)
normal	0.2655 +/-(0.0073)
codec	0.9187 +/-(0.0112)
filter_	0.8832 +/-(0.0091)
codec_filter	0.8292 +/-(0.0116)
c_level	0.4586 +/-(0.0116) ~ 0.7117 +/-(0.0076)
block	0.4307 +/-(0.0116) ~ 0.7317 +/-(0.0094)
cl_block	0.2801 +/-(0.0118) ~ 0.7233 +/-(0.0078)

```
BLZ_CRate     --> 0.414547820254
BLZ_CSpeed    --> 0.279735766264
IN_CR         --> 0.112420496888
IN_CS         --> 0.111697815272
IN_DS         --> 0.0815981013218
```

In [27]:
```
CUSTOM_IN = ['IN_CS', 'IN_DS', 'IN_CR', 'LZ4_CRate', 'LZ4_CSpeed']
X = df[CUSTOM_IN].values
clf1 = RandomForestClassifier(
    n_estimators=20, max_features=4, max_depth=12, bootstrap=False,
    criterion='entropy')
display(mp.cross_val_report(clf1, cv, X, Y))
clf1.fit(X, Y)
my_dict = dict(zip(clf1.feature_importances_, CUSTOM_IN))
for elem in np.sort(clf1.feature_importances_)[::-1]:
    print('%-12s --> %s' % (my_dict.get(elem), elem))
```

Report

Name	Score
balanced	0.7802 +/-(0.0074)
brier	-1.6447 +/-(0.0473)
normal	0.2677 +/-(0.0114)
codec	0.9304 +/-(0.0068)
filter_	0.8758 +/-(0.0056)

codec_filter	0.8312 +/-(0.0094)
c_level	0.4711 +/-(0.0117) ~ 0.7221 +/-(0.0096)
block	0.4326 +/-(0.0130) ~ 0.7396 +/-(0.0079)
cl_block	0.2828 +/-(0.0144) ~ 0.7322 +/-(0.0061)

```
LZ4_CRate     --> 0.395924348978
LZ4_CSpeed    --> 0.289277542278
IN_CR         --> 0.120163085278
IN_CS         --> 0.107634450453
IN_DS         --> 0.087000573013
```

In [28]:
```
CUSTOM_IN = ['IN_CS', 'IN_DS', 'IN_CR', 'Mean', 'Max', 'Min']
X = df[CUSTOM_IN].values
clf1 = RandomForestClassifier(
    n_estimators=20, max_features=4, max_depth=12, bootstrap=False,
    criterion='entropy')
display(mp.cross_val_report(clf1, cv, X, Y))
clf1.fit(X, Y)
my_dict = dict(zip(clf1.feature_importances_, CUSTOM_IN))
for elem in np.sort(clf1.feature_importances_)[::-1]:
    print('%-12s --> %s' % (my_dict.get(elem), elem))
```

Report

Name	Score
balanced	0.7501 +/-(0.0061)
brier	-1.6290 +/-(0.0315)
normal	0.2177 +/-(0.0086)
codec	0.8852 +/-(0.0087)
filter_	0.9084 +/-(0.0080)
codec_filter	0.8149 +/-(0.0077)
c_level	0.4113 +/-(0.0144) ~ 0.6787 +/-(0.0137)
block	0.3848 +/-(0.0124) ~ 0.6891 +/-(0.0079)
cl_block	0.2257 +/-(0.0105) ~ 0.6836 +/-(0.0060)

```
Mean          --> 0.311441870777
Max           --> 0.270196884785
IN_CS         --> 0.12458594747
IN_CR         --> 0.12231805242
IN_DS         --> 0.0888430619235
Min           --> 0.0826144353476
```

In [29]:
```
CUSTOM_IN = ['IN_CS', 'IN_DS', 'IN_CR']
X = df[CUSTOM_IN].values
clf1 = RandomForestClassifier(
    n_estimators=20, max_features=3, max_depth=12, bootstrap=False,
    criterion='entropy')
display(mp.cross_val_report(clf1, cv, X, Y))
clf1.fit(X, Y)
my_dict = dict(zip(clf1.feature_importances_, CUSTOM_IN))
for elem in np.sort(clf1.feature_importances_)[::-1]:
    print('%-12s --> %s' % (my_dict.get(elem), elem))
```

Report

Name	Score
balanced	0.3846 +/-(0.0298)
brier	-2.3943 +/-(0.0201)
normal	0.0248 +/-(0.0250)
codec	0.7570 +/-(0.0097)
filter_	0.3281 +/-(0.0638)
codec_filter	0.2377 +/-(0.0570)
c_level	0.1178 +/-(0.0307) ~ 0.5633 +/-(0.0153)
block	0.1215 +/-(0.0100) ~ 0.4994 +/-(0.0098)
cl_block	0.0968 +/-(0.0103) ~ 0.5314 +/-(0.0108)

```
IN_CS         --> 0.409653469971
IN_CR         --> 0.398350906174
IN_DS         --> 0.191995623855
```

MultiOutputs

In [30]:
```
CUSTOM3_IN = ['IN_CS', 'IN_DS', 'IN_CR', 'BLZ_CRate', 'Sd', 'BLZ_CSpeed',
              'Mean', 'Kurt', 'Skew', 'N_Streaks', 'Max', 'Min']
X = df[CUSTOM3_IN].values
clf1 = ChainedMultiOutputClassifier(
    RandomForestClassifier(
        n_estimators=20, max_features=8, max_depth=12, bootstrap=False,
        criterion='entropy'))
display(mp.cross_val_report(clf1, cv, X, Y, True))
```

Report

Name	Score
balanced	0.8518 +/-(0.0047)
brier	0.0000 +/-(0.0000)
normal	0.3385 +/-(0.0091)
codec	0.9532 +/-(0.0047)
filter_	0.9504 +/-(0.0060)
codec_filter	0.9123 +/-(0.0076)
c_level	0.5563 +/-(0.0147) ~ 0.7867 +/-(0.0095)
block	0.5107 +/-(0.0059) ~ 0.8002 +/-(0.0072)
cl_block	0.3432 +/-(0.0111) ~ 0.7920 +/-(0.0053)

D.9. Cuaderno classifiers_comparison.ipynb

Comparativa de algoritmos clasificadores

In [1]:
```
%matplotlib inline
%config InlineBackend.figure_format='retina'

%load_ext autoreload
%autoreload 2

%load_ext version_information
%version_information numpy, scipy, matplotlib, pandas, scikit-learn
```

Software	Version
Python	3.5.3 64bit [GCC 4.4.7 20120313 (Red Hat 4.4.7-1)]
IPython	5.1.0
OS	Linux 4.9.16 gentoo x86_64 with debian stretch sid
numpy	1.12.1
scipy	0.19.0
matplotlib	2.0.0
pandas	0.19.2
scikit-learn	0.18.1
Fri May 05 17:37:38 2017 UTC	

In [2]:
```
import os
import sys
sys.path.append("../src/")

import random
from IPython.display import display
import pandas as pd
import matplotlib
import numpy as np
from matplotlib import pyplot as plt
from sklearn.externals import joblib
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.svm import LinearSVC
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import cross_val_score
from sklearn.multioutput import MultiOutputClassifier
from multioutput_chained import ChainedMultiOutputClassifier
from sklearn.preprocessing import StandardScaler
import ml_plots as mp
import scoring_functions as sf
import warnings
import time
warnings.filterwarnings('ignore')

pd.options.display.float_format = '{:,.3f}'.format
matplotlib.rcParams.update({'font.size': 12})
```

In [3]:
```
IN_OPTIONS = ['IN_CR', 'IN_CS', 'IN_DS', 'is_Table', 'is_Columnar',
              'is_Int', 'is_Float', 'is_String', 'Type_Size', 'Chunk_Size',
              'Mean', 'Median', 'Sd', 'Skew', 'Kurt', 'Min', 'Max', 'Q1',
              'Q3', 'N_Streaks', 'BLZ_CRate', 'BLZ_CSpeed',
              'BLZ_DSpeed', 'LZ4_CRate', 'LZ4_CSpeed', 'LZ4_DSpeed']
OUT_CODEC = ['Blosclz', 'Lz4', 'Lz4hc', 'Zstd']
OUT_FILTER = ['Noshuffle', 'Shuffle', 'Bitshuffle']
OUT_LEVELS = ['CL1', 'CL2', 'CL3', 'CL4', 'CL5', 'CL6', 'CL7', 'CL8', 'CL9']
OUT_BLOCKS = ['Block_8', 'Block_16', 'Block_32', 'Block_64', 'Block_128',
              'Block_256', 'Block_512', 'Block_1024', 'Block_2048']
OUT_OPTIONS = OUT_CODEC + OUT_FILTER + OUT_LEVELS + OUT_BLOCKS
CUSTOM_IN = ['IN_CS', 'IN_DS', 'IN_CR', 'BLZ_CRate', 'Sd', 'BLZ_CSpeed',
             'Mean', 'Kurt', 'Skew', 'Max', 'Min', 'N_Streaks']
CHUNK_ID = ['Filename', 'DataSet', 'Table', 'Chunk_Number']
```

In [4]:
```
df = pd.read_csv('../data/training_data.csv', sep='\t')
rfc = RandomForestClassifier(
    n_estimators=20, max_features=8, max_depth=12, bootstrap=False,
    criterion='entropy')
svc = MultiOutputClassifier(
    SVC(C=1000, gamma=0.1, decision_function_shape='ovr'))
lsvc = MultiOutputClassifier(LinearSVC())
knc = KNeighborsClassifier(n_neighbors=5, weights='distance')
lda = MultiOutputClassifier(LinearDiscriminantAnalysis())
lrc = MultiOutputClassifier(LogisticRegression(solver='newton-cg'))
names = ['RandomForest', 'SVM-rbf', 'SVM-linear', 'KNeighbors', 'LDA',
         'Logit']
classifiers = [rfc, svc, lsvc, knc, lda, lrc]
```

In [5]:
```
rfc_ch = ChainedMultiOutputClassifier(
    RandomForestClassifier(
        n_estimators=20, max_features=8, max_depth=12, bootstrap=False,
        criterion='entropy'))
svc_ch = ChainedMultiOutputClassifier(
    SVC(C=1000, gamma=0.1, decision_function_shape='ovr'))
lsvc_ch = ChainedMultiOutputClassifier(LinearSVC())
```

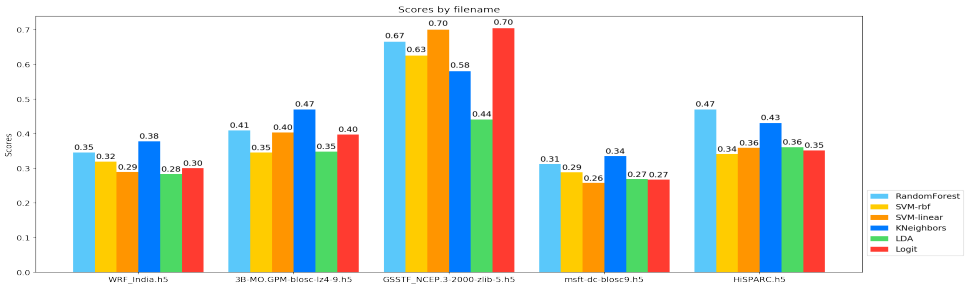
```
knc_ch = ChainedMultiOutputClassifier(
    KNeighborsClassifier(n_neighbors=5, weights='distance'))
lda_ch = ChainedMultiOutputClassifier(LinearDiscriminantAnalysis())
lrc_ch = ChainedMultiOutputClassifier(
    LogisticRegression(solver='newton-cg'))
classifiers_ch = [rfc_ch, svc_ch, lsvc_ch, knc_ch, lda_ch, lrc_ch]
```

In [6]:
```
def train_and_test(train_df, test_df, detailed=False):
    scaler = StandardScaler().fit(train_df[CUSTOM_IN].values)
    X_train, Y_train = (scaler.transform(train_df[CUSTOM_IN].values),
                        train_df[OUT_OPTIONS].values)
    X_test, Y_test = (scaler.transform(test_df[CUSTOM_IN].values),
                      test_df[OUT_OPTIONS].values)
    scores = []
    for i, clf in enumerate(classifiers):
        clf.fit(X_train, Y_train)
        if detailed:
            classifiers_ch[i].fit(X_train, Y_train)
            scores.append(sf.balanced(classifiers_ch[i], X_test, Y_test))
            scores.append(sf.balanced(clf, X_test, Y_test))
            scores.append(sf.codec(clf, X_test, Y_test))
            scores.append(sf.filter_(clf, X_test, Y_test))
            scores.append(sf.c_level_nice(clf, X_test, Y_test))
            scores.append(sf.block_nice(clf, X_test, Y_test))
        else:
            scores.append(sf.balanced(clf, X_test, Y_test))
    return scores
def autolabel(ax, rects):
    """
    Attach a text label above each bar displaying its height
    """
    for rect in rects:
        height = rect.get_height()
        ax.text(rect.get_x() + rect.get_width()/2., 1.01*height,
                '%.2f' % (height),
                ha='center', va='bottom')
def plot_clf_scores(title, x_labels, detailed=False):
    fig = plt.figure(figsize=(20, 8))
    width = 0.14
    if detailed:
        ind = np.arange(6)
    else:
        ind = np.arange(len(filenames))
    ax = fig.add_subplot(111)
    colors = ['#5AC8FA', '#FFCC00', '#FF9500', '#007AFF', '#4CD964',
              '#FF3B30']
    if detailed:
        colors = ['#FF3B30', '#4CD964', '#FFCC00', '#FF9500', '#5AC8FA',
                  '#007AFF']
    rects = []
    if detailed:
        for i in range(6):
            rects.append(
                ax.bar(ind + width * i,
                       [scores[i + j * 6] for j in range(len(names))],
                       width, color=colors[i]))
    else:
        for i, name in enumerate(names):
            rects.append(
                ax.bar(ind + width * i,
                       [e[i] for e in all_scores],
                       width, color=colors[i]))
    if detailed:
        ax.legend([rec[0] for rec in rects],
                  ['Chained_Balanced', 'Balanced', 'Codec', 'Filter', 'CL',
                   'Block'],  loc='lower left', bbox_to_anchor=(1, 0.05))
    else:
        ax.legend([rec[0] for rec in rects], names,  loc='lower left',
                  bbox_to_anchor=(1, 0.05))
    ax.set_ylabel('Scores')
    ax.set_title(title)
    ax.set_xticks(ind + width * 2.5)
    ax.set_xticklabels(x_labels)
    for rec in rects:
        autolabel(ax, rec)
```

Puntuaciones CV-Fichero

In [7]:
```
all_scores = []
filenames = []
for filename in df.drop_duplicates(subset=['Filename'])['Filename']:
    filenames.append(filename)
    test_df = df[df.Filename == filename]
    train_df = df[df.Filename != filename]
    all_scores.append(train_and_test(train_df, test_df))
```

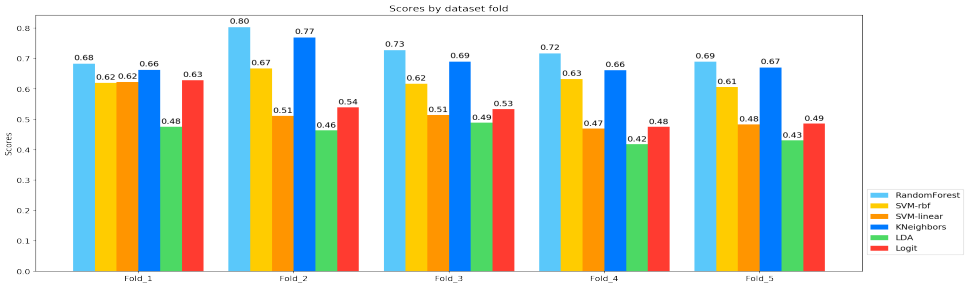
In [8]:
```
plot_clf_scores('Scores by filename', filenames)
```



Puntuaciones CV-DataSets

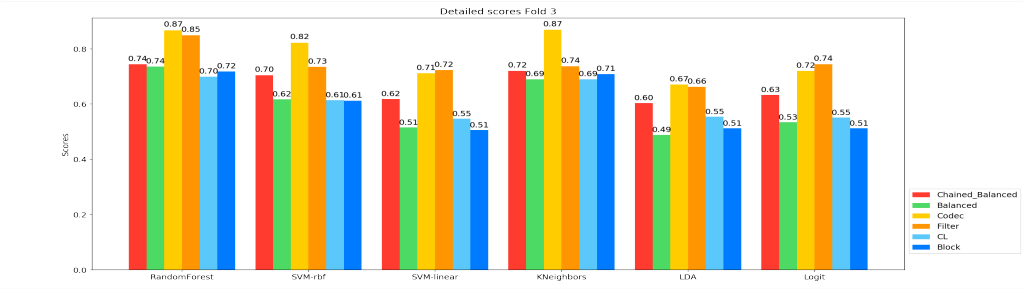
In [9]:
```
dataset_rows = []
set_id = ['Filename', 'DataSet', 'Table']
for index, row in df.drop_duplicates(
    subset=['Filename', 'DataSet', 'Table'])[set_id].iterrows():
    dataset_rows.append(row)
N = len(dataset_rows)
split_size = N // 5
total = list(range(N))
current = total
test_indices = []
for i in range(5):
    test = random.sample(current, N // 5)
    test_indices.append(test)
    current = [x for x in current if x not in test]

all_scores = []
for i in range(5):
    train_i = [x for x in total if x not in test_indices[i]]
    train_df = pd.DataFrame()
    test_df = pd.DataFrame()
    for j in train_i:
        train_df = train_df.append(
            df[(df.Filename == dataset_rows[j]['Filename']) &
               (df.DataSet == dataset_rows[j]['DataSet']) &
               (df.Table == dataset_rows[j]['Table'])])
    for k in test_indices[i]:
        test_df = test_df.append(
            df[(df.Filename == dataset_rows[k]['Filename']) &
               (df.DataSet == dataset_rows[k]['DataSet']) &
               (df.Table == dataset_rows[k]['Table'])])
    all_scores.append(train_and_test(train_df, test_df))
```

In [10]:
```
plot_clf_scores('Scores by dataset fold', ('Fold_1', 'Fold_2', 'Fold_3',
                                           'Fold_4', 'Fold_5'))
```



In [11]:
```
train_i = [x for x in total if x not in test_indices[2]]
train_df = pd.DataFrame()
test_df = pd.DataFrame()
for j in train_i:
    train_df = train_df.append(
        df[(df.Filename == dataset_rows[j]['Filename']) &
           (df.DataSet == dataset_rows[j]['DataSet']) &
           (df.Table == dataset_rows[j]['Table'])])
for k in test_indices[2]:
    test_df = test_df.append(
        df[(df.Filename == dataset_rows[k]['Filename']) &
           (df.DataSet == dataset_rows[k]['DataSet']) &
           (df.Table == dataset_rows[k]['Table'])])
scores = train_and_test(train_df, test_df, True)
```

In [12]:
```
plot_clf_scores('Detailed scores Fold 3', names, detailed=True)
```



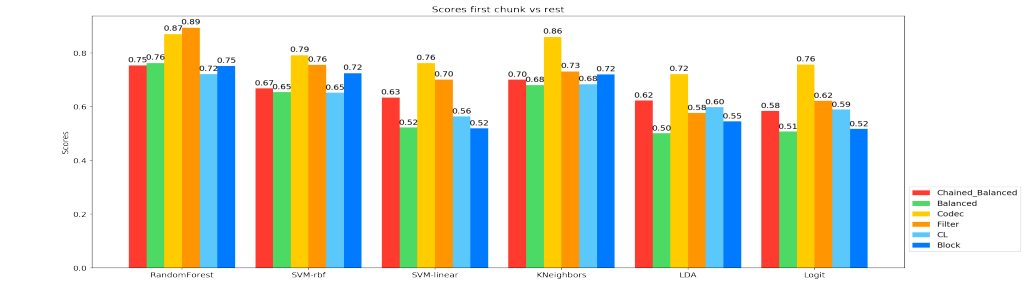
Puntuaciones CV-Chunk

Primer chunk vs el resto

In [13]:
```
dataset_rows = []
for index, row in df[df.Chunk_Number == 2].drop_duplicates(
    subset=['Filename', 'DataSet', 'Table'])[set_id].iterrows():
    dataset_rows.append(row)

all_scores = []
train_df = pd.DataFrame()
test_df = pd.DataFrame()
for j in range(len(dataset_rows)):
    train_df = train_df.append(
        df[(df.Filename == dataset_rows[j]['Filename']) &
           (df.DataSet == dataset_rows[j]['DataSet']) &
           (df.Table == dataset_rows[j]['Table']) &
           (df.Chunk_Number == 1)])
    test_df = test_df.append(
        df[(df.Filename == dataset_rows[j]['Filename']) &
           (df.DataSet == dataset_rows[j]['DataSet']) &
           (df.Table == dataset_rows[j]['Table']) &
           (df.Chunk_Number != 1)])
scores = train_and_test(train_df, test_df, True)
```

In [14]:
```
plot_clf_scores('Scores first chunk vs rest', names, detailed=True)
print('%s train --- %s test' % (train_df.shape[0], test_df.shape[0]))
```

```
280 train --- 4256 test
```



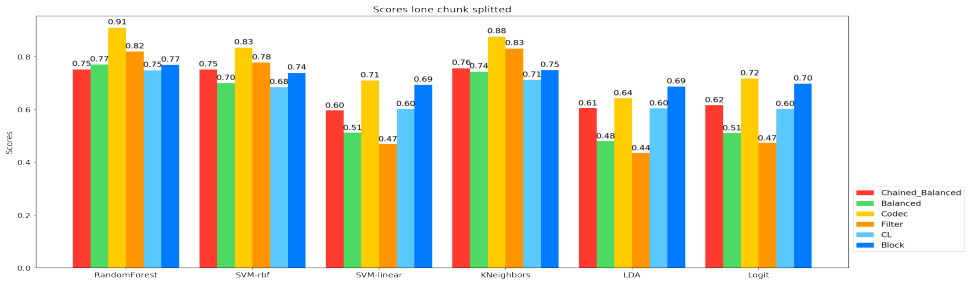
Chunks únicos mitad vs mitad

In [15]:
```
sets = []
for index, row in df[df.Chunk_Number == 2].drop_duplicates(
    subset=['Filename', 'DataSet', 'Table']).iterrows():
    sets.append(row['DataSet'])

dataset_rows = df.drop_duplicates(
    subset=['Filename', 'DataSet', 'Table'])[~df.DataSet.isin(sets)]
dataset_rows = dataset_rows.reset_index(drop=True)

N = dataset_rows.shape[0]
total = list(range(N))
test_i = random.sample(total, N // 2)
train_i = [x for x in total if x not in test_i]

train_df = pd.DataFrame()
test_df = pd.DataFrame()
for j in train_i:
    train_df = train_df.append(
        df[(df.Filename == dataset_rows.ix[j]['Filename']) &
           (df.DataSet == dataset_rows.ix[j]['DataSet']) &
           (df.Table == dataset_rows.ix[j]['Table'])] )
```

```
for k in test_i:
    test_df = test_df.append(
        df[(df.Filename == dataset_rows.ix[k]['Filename']) &
           (df.DataSet == dataset_rows.ix[k]['DataSet']) &
           (df.Table == dataset_rows.ix[k]['Table'])])
scores = train_and_test(train_df, test_df, True)
```

In [16]:
```
plot_clf_scores('Scores lone chunk splitted', names, detailed=True)
print('%s train --- %s test' % (train_df.shape[0], test_df.shape[0]))
```

```
273 train --- 266 test
```

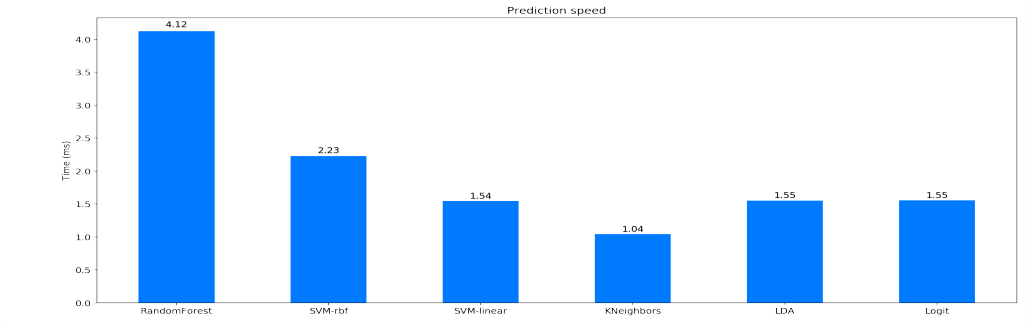


Velocidades

In [17]:
```
def atomic_benchmark_estimator(estimator, X_test, verbose=False):
    """Measure runtime prediction of each instance."""
    n_instances = X_test.shape[0]
    runtimes = np.zeros(n_instances, dtype=np.float)
    for i in range(n_instances):
        instance = X_test[[i], :]
        start = time.time()
        estimator.predict(instance)
        runtimes[i] = time.time() - start
    if verbose:
        print("atomic_benchmark runtimes:", min(runtimes), scoreatpercentile(
            runtimes, 50), max(runtimes))
    return runtimes
```

In [18]:
```
X, Y = df[CUSTOM_IN].values, df[OUT_OPTIONS].values
runtimes_mean = []
for clf in classifiers:
    shuffle = ShuffleSplit(n_splits=5, test_size=0.5, random_state=5)
    runtimes = []
    for train_index, test_index in shuffle.split(X, Y):
        scaler = StandardScaler().fit(X[train_index])
        X_train, Y_train = scaler.transform(X[train_index]), Y[train_index]
        X_test = scaler.transform(X[test_index])
        clf.fit(X_train, Y_train)
        runtimes.append(atomic_benchmark_estimator(clf, X_test).mean())
    runtimes_mean.append(np.mean(runtimes))
```

In [20]:
```
fig = plt.figure(figsize=(20, 8))
width = 0.50
ind = np.arange(len(names))
ax = fig.add_subplot(111)
colors = ['#4CD964', '#FFCC00', '#FF9500', '#5AC8FA', '#007AFF', '#FF3B30']
rects = []
rects.append(
    ax.bar(ind + width * i,
           [e*10**3 for e in runtimes_mean],
           width, color=colors[4]))
ax.set_ylabel('Time (ms)')
ax.set_title('Prediction speed')
ax.set_xticks(ind + width * i)
ax.set_xticklabels(names)

for rec in rects:
    autolabel(ax, rec)
```


Prediction speed
Time (ms)
4.12
2.23
1.54
1.04
1.55
1.55
RandomForest
SVM-rbf
SVM-linear
KNeighbors
LDA
Logit