Uso de algoritmos de aprendizaje automático aplicados a bases de datos genéticos.

Pedro Morell Miranda
Máster en Estadística y Bioinformática
Programación para la Bioinformática

Pau Andrio Balado

20/05/2017

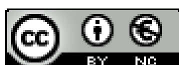
Esta obra está sujeta a una licencia de Reconocimiento-NoComercial-SinObraDerivada 3.0 España de Creative Commons

Licencias alternativas (elegir alguna de las siguientes y sustituir la de la página anterior)

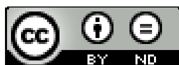
A) Creative Commons:

Esta obra está sujeta a una licencia de Reconocimiento-NoComercial-SinObraDerivada 3.0 España de Creative Commons

Esta obra está sujeta a una licencia de Reconocimiento-NoComercial-CompartirIgual 3.0 España de Creative Commons



Esta obra está sujeta a una licencia de Reconocimiento-NoComercial 3.0 España de Creative Commons



Esta obra está sujeta a una licencia de Reconocimiento-SinObraDerivada 3.0 España de Creative Commons

Esta obra está sujeta a una licencia de Reconocimiento-CompartirIgual 3.0 España de Creative Commons

Esta obra está sujeta a una licencia de Reconocimiento 3.0 España de Creative Commons

B) GNU Free Documentation License (GNU FDL)

Copyright © AÑO TU-NOMBRE.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free So-

ftware Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled "GNU Free Documentation License".

C) Copyright

© (el autor/a)
Reservados todos los derechos. Está prohibido la reproducción total o parcial de esta obra por cualquier medio o procedimiento, comprendidos la impresión, la reprografía, el microfilme, el tratamiento informático o cualquier otro sistema, así como la distribución de ejemplares mediante alquiler y préstamo, sin la autorización escrita del autor o de los límites que autorice la Ley de Propiedad Intelectual.

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Uso de algoritmos de aprendizaje automático aplicados a bases de datos genéticos.*
Nombre del autor:	*Pedro Morell Miranda*
Nombre del consultor/a:	
Nombre del PRA:	*Pau Andrio Balado*
Fecha de entrega (mm/aaaa):	05/2017
Titulación::	*Máster en Bioestadística y Bioinformática*
Área del Trabajo Final:	*Programación para la Bioinformática*
Idioma del trabajo:	***Castellano***
Palabras clave	*Genómica, Machine Learning, Python*

Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados y conclusiones del trabajo.*

El campo del Aprendizaje Automático, o Machine Learning, se ha desarrollado de una manera increíble en las últimas décadas. De forma paralela, los métodos de ultrasecuenciación han hecho que podamos procesar genomas enteros y producir una cantidad enorme de información genómica de forma relativamente rápida y barata.

Dado que los algoritmos de Machine Learning nos permiten, entre otras cosas, deducir patrones de grandes cantidades de datos, el objetivo de este análisis es el de estudiar la aplicación de diversos algoritmos de Machine Learning a datos genómicos.

Además, usamos datos de HapMap y 1000 Genomes para analizar cómo influye la metodología en los resultados obtenidos. El criterio de clasificación es la población de origen.

Los resultados muestran un relativo buen rendimiento para ambas bases de datos. Sin embargo, vemos que los datos de 1000 Genomes resultan más difíciles de clasificar que los de HapMap, en especial si intentamos clasificarlos a nivel de población, pese a tener sets de entrenamiento mucho más grandes.

Los datos Hapmap funcionan especialmente bien con todos los algoritmos utilizados tanto para aprendizaje no supervisado como supervisado (con la excepción de con Redes Neurales). Esto puede deberse a que los datos proceden de tres poblaciones muy distintas. Sin embargo, 1000 Genomes funciona mejor con Redes Neurales o Support Vector Machine que con el resto de algoritmos supervisados, mientras

que el clustering se ha mostrado especialmente complicado.

Estos resultados muestran que uno de los factores claves es la relación entre las muestras y lo diferentes que sean los grupos.

Abstract (in English, 250 words or less):

The field of Machine Learning have seen an unprecedented development in the last decades. On a similar way, thanks to Next Generation Sequencing, we can now process whole genomes and produce huge amounts of genomic data with relative speed and low costs.

Due to Machine Learning ability to extract complex patterns from big amounts of data, the goal of this analysis is to study the application of several popular Machine Learning algorithms to genomic data.

Also, we used data from HapMap and 1000 Genomes in order to analyse how this algorithms are influenced by different methodologies of collecting and processing the samples. The classification criteria is the sample's origin population, and with 1000 Genomes was also included a supergroup analysis with meta-populations.

Results show a relatively good performance with both databases, being the samples from 1000 Genomes harder to classify (only the meta-population classification had acceptable performance rates). This contradicts one of our initial hipothesis, since 1000 Genomes have more samples for training.

For Hapmap every algorithm for non-supervised and supervised learning performed well (except for Artificial Neural Networks). We suppose that this has something to do with the fact that the samples come from very different populations. On the other side, 1000 Genomes data works better with Artificial Neural Networks and Support Vector Machines than with the other supervised algorithms, and clustering have been specially complicated to achieve.

This results highlight that one of the chief factors is the relationship between samples and the genetic difference between classes.

Índice

1. Introducción....1
1.1 Contexto y justificación del Trabajo....1
1.2 Objetivos del Trabajo....2
1.3 Enfoque y método seguido....2
1.4 Planificación del Trabajo....3
1.5 Breve sumario de productos obtenidos....4
1.6 Breve descripción de los otros capítulos de la memoria....5
2. Los datos:....6
3. Hipótesis:....7
4. Pre-procesado y transformación de los datos:....8
5. Algoritmos utilizados:....9
5.1 Clustering:....9
5.1.1 K-Means (Slonim et al., 2013, Hartigan, 1975):....9
5.1.2 DBSCAN (Ester et al., 1996):....13
5.1.3 HDBSCAN (Campello et al.,2013) :....15
5.2: Clasificación:....20
5.2.1 Naives Bayes (Domingos & Pazzani, 1997):....20
5.2.2 Random Forest (Breiman, 2001):....21
5.2.3 Support Vector Machine (SVM) (Suykens & Vandewalle, 1999, Furey et al. 2000):....23
5.2.4 Redes Neurales Artificiales (Hinton, 1985, Glorot & Bengio, 2010, Kingma et al., 2014):....26
6. Entrenando los algoritmos:....29
7. Evaluando el Rendimiento de los Modelos:....30
8. Resultados:....33
9. Comparativa del rendimiento de los algoritmos:....50
10. Conclusiones....55
11. Glosario....57
12. Bibliografía....58
13. Anexo:....61
13.1 Código con que se pre-procesaron los datos:....61
13.1.1: Hapmap para cada archivo....61
13.1.2: Función para juntar los archivos....64
13.1.3: Pre-procesado de los datos de 1000 Genomes....65

Lista de figuras

Imagen1 1
Imagen2 2
Imagen3 3
Imagen4 3
Imagen5 3
Imagen6 3
Imagen7 3
Imagen8 3
Imagen9 4
Imagen45 4
2Imagen10 6
2Imagen11 6
Imagen98 10
Imagen38 10
Imagen39 11
Imagen40 11
Imagen41 12
Imagen12 12
Imagen42 12
Imagen43 14
Imagen13 14
Imagen44 15
Imagen97 16
Imagen46 16
Imagen47 17
Imagen48 18
Imagen49 18
Imagen50 19
Imagen14 19
Imagen15 20
Imagen51 21
Imagen52 22
Imagen53 22
Imagen16 23
Imagen54 24
Imagen55 25
Imagen17 25
Imagen56 26
Imagen57 27
Imagen18 28
Imagen19 30
Imagen20 31
Imagen22 31
Imagen21 32
Imagen23 33
Imagen24 33

Imagen25 33
Imagen26 34
Imagen27 34
Imagen28 34
Imagen31 35
Imagen32 35
Imagen58 35
Imagen29 36
Imagen30 36
Imagen69 37
Imagen70 37
Imagen71 38
Imagen72 38
Imagen73 39
Imagen61 39
Imagen62 40
Imagen63 40
Imagen64 41
Imagen65 41
Imagen66 42
Imagen67 42
Imagen68 43
Imagen74 43
Imagen75 44
Imagen76 44
Imagen77 45
Imagen78 45
Imagen79 46
Imagen80 46
Imagen81 47
Imagen82 47
Imagen33 49
Imagen34 49
Imagen35 49
Imagen36 50
Imagen59 50
Imagen60 51
Imagen83 52
Imagen84 52
Imagen85 52
Imagen86 52
Imagen87 52
Imagen88 52
Imagen89 53
Imagen90 53
Imagen91 53
Imagen92 53
Imagen93 53
Imagen94 54
Imagen95 54

Imagen96..54

1. Introducción

1.1 Contexto y justificación del Trabajo

Los algoritmos de aprendizaje automático o, en su forma anglosajona, de Machine Learning, son un campo de la Ciencia de Datos que ha avanzado mucho en la última década. Esto, acompañado por el éxito de varios de estos en varias famosas competiciones de programación ha provocado una ola de expectación en torno a su aplicación en actividades del día a día, como el detector de spam de nuestro correo electrónico o nuevos algoritmos de procesado de imagen y vídeo como el que usa la popular aplicación Prisma (Gatys et al., 2015) .

Productos como los coches autónomos o el texto predictivo también se basan en este tipo de tecnología, y su aplicación está en auge, gracias al incremento en la capacidad de computación de ordenadores y móviles modernos. Debido al incremento de la investigación en Bioinformática, y las expectativas del sector médico y científico en este campo, se están investigando varias aplicaciones de esta tecnología.

Actualmente, para relacionar o clasificar individuos existen dos aproximaciones: necesitas conocer qué características los hacen similares (presentan este genotipo en los genes X,Y,Z...) o una aproximación de fuerza bruta (procesado de secuencias enteras base a base comparando las diferentes muestras). El primer caso es muy costoso, pues requiere conocer experimentalmente el efecto de cada gen con el que trabajamos, mientras que el segundo es muy costoso computacionalmente, pues las secuencias, sobre todo gracias a las nuevas técnicas de ultrasecuenciación (Next Generation Sequencing, o NGS), son enormes. Si tenemos que comparar a una secuencia modelo cada vez, esta forma de clasificación se muestra muy ineficiente.

Por el contrario, el Machine Learning nos ofrece una alternativa que, pese a ser también muy cara computacionalmente, nos permite comparar una gran cantidad de individuos a la vez. Esto se debe a que creamos un modelo estadístico que determine a qué clase es más probable que pertenezca cada nuevo individuo.

El paso de crear el modelo a partir de unos datos convenientemente pre-procesados se le conoce como entrenamiento, y el resultado es un modelo capaz, con un cierto margen de error, de clasificar nuevos casos a partir de información extraída de los datos con que se le ha entrenado.

Esta aproximación es muy potente, pues no es necesario conocer de antemano qué patrones o diferencias presenta cada grupo. Este hecho, además, nos permite considerar una gran cantidad de variables, tanto numéricas como categóricas.

1.2 Objetivos del Trabajo

- Analizar cómo se comportan una serie de algoritmos muy populares dentro del Machine Learning cuando son entrenados con datos de bases de datos genómicos.
- Aplicar estos algoritmos a dos bases de datos diferentes, y analizar las posibles diferencias entre los resultados de uno y otro y las causas de estas, de haberlas.
- Aprovechando que se realizan los análisis, intentar extraer alguna conclusión de los datos con los que se ha trabajado.

1.3 Enfoque y método seguido

Para analizar el comportamiento de los algoritmos de Machine Learning el enfoque es muy directo. La única opción es entrenar diferentes algoritmos y posteriormente comparar la precisión con que responde cada uno. Para ello, seleccionamos algunos de los algoritmos más populares dentro del campo del Machine Learning.

Como los datos genómicos se caracterizan por ser muy grandes, y los algoritmos de Machine Learning suelen tardar bastante en ejecutarse, decidimos reducir la cantidad de SNPs con la que trabajar. Por un lado, podríamos haber usado una Selección de Características (Feature Selection en inglés), pero nos decantamos por buscar un motivo biológico para seleccionar estas variantes para poder disponer de un mejor contexto a la hora de interpretar los resultados y que no se tratase solo de las variantes más discriminantes.

Por tanto, decidimos usar todas las mutaciones de una sola base (Single Nucleotide Polymorfism, o SNP) de los genes relacionados con el Complejo Mayor de Histocompatibilidad presentes en el cromosoma 6. Son 99 genes, y nos deja con una lista de 42472 SNPs asociados a estos. Este es un set de datos de un tamaño considerable sin llegar a ser computacionalmente prohibitivo y, al estar todos situados en el mismo cromosoma, facilita el trabajo de procesado. A parte de ofrecer un set de datos apropiado para el análisis, estos genes han demostrado estar relacionados con la sensibilidad diferencial de diferentes poblaciones a varias enfermedades, como el VIH (Kaslow et al., 1996), tanto en humanos como en otras especies (Paterson et al, 1997).

Para evaluar cómo de bien funciona un algoritmo hay varias métricas posibles (Accuracy, f1, precisión, AUC, etc.). Para nuestras medidas hemos elegido principalmente la Accuracy, ya que el f1 y AUC, las otras más usadas, están pensadas para clasificadores binarios y requieren de adaptación para aplicarlos a nuestros datos.

Por otro lado, se han empleado varios métodos basados en Cross Validation (Kohavi, 1995). Esta técnica se basa en dividir nuestros datos de forma aleatoria en un número de “paquetes” que determinamos nosotros. Posteriormente ejecutamos diversas permutaciones de

nuestros datos, de forma que cada “paquete” pase una vez por el puesto de testeo. De esta forma podemos ejecutar diversas veces el entrenamiento y evaluar como se comporta nuestro algoritmo ante datos nuevos sin necesidad de conseguir más datos.

1.4 Planificación del Trabajo

Los datos necesarios para realizar nuestros análisis se encuentran publicados y disponibles mediante el servidor FTP del NCBI y del Sanger Institute. Todos los algoritmos que vamos a emplear son, a su vez, open source, y se pueden obtener gratuitamente de diferentes formas.

Para facilitar la gestión del trabajo y las versiones empleadas todo el proceso (descarga de librerías necesarias, control de versiones, descarga de programas necesarios, etc.) se realizaron mediante la plataforma Anaconda del grupo Continuum. Esta ofrece una suite con diversos programas para el análisis y la representación de datos, basados en el lenguaje de programación Python, muy utilizado por la comunidad científica debido a que tiene una gran cantidad de librerías y herramientas científicas y para el análisis de datos (Oliphant, 2007;Millman & Alvazis, 2011). En concreto nos interesan dos programas dentro de esta distribución: Jupyter Notebooks y Spyder.

Jupyter Notebook es la nueva versión de Ipython Notebooks (Perez &Granger, 2007), una herramienta web que nos permite crear documentos con texto y código en Python o R, y mostrar los resultados tal cual se van ejecutando. Esto la hace una herramienta imprescindible para muchos, ya que facilita la presentación de análisis al poder exportase a html o a pdf fácilmente.

Por otro lado, Spyder es un editor IDE que cuenta con varias características útiles para el procesado de datos, y que editores enfocados en programación no tienen (visualizador de variables, historial de comandos, etc.). Es similar (y hasta cierto punto está inspirado en) RStudio, herramienta súper-popular entre los usuarios de R, el otro lenguaje de programación de referencia entre científicos.

Durante todo este proceso se usarán las librerías Numpy (van der Walt et al., 2011) y Pandas (McKinney, 2010). Numpy es una librería matemática muy popular en Python, y la mayor parte de las librerías científicas están basadas o son compatibles con ella. Pandas es una librería específica para procesado de datos y está basada en Numpy. Además de estas dos librerías, básicas para el manejo y procesado de los datos, usaremos matplotlib (Hunter, 2007), una librería para la representación de datos, y scikit-learn (Pedregosa et al., 2011), librería especializada en algoritmos y funciones auxiliares para Machine Learning.

En cuanto a las tareas a realizar, estas se dividen en estas fases:

- Documentación y diseño.
- Transformación de los datos
- Implementación de los diferentes Algoritmos
- Comparación del rendimiento y creación de una web para presentar los resultados
- Realización de la memoria y defensa del trabajo.

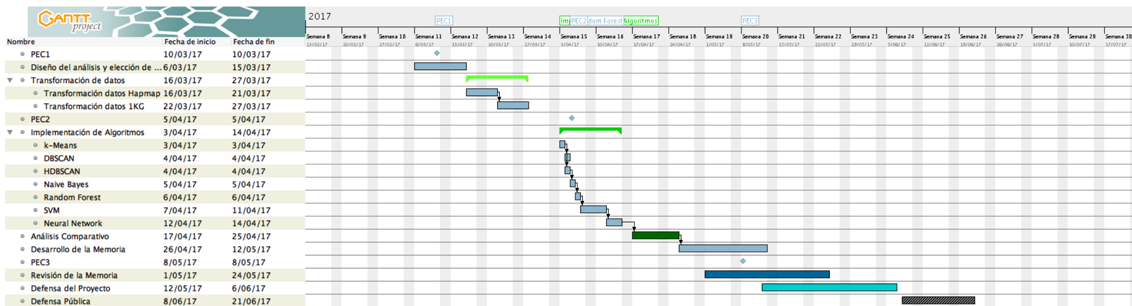
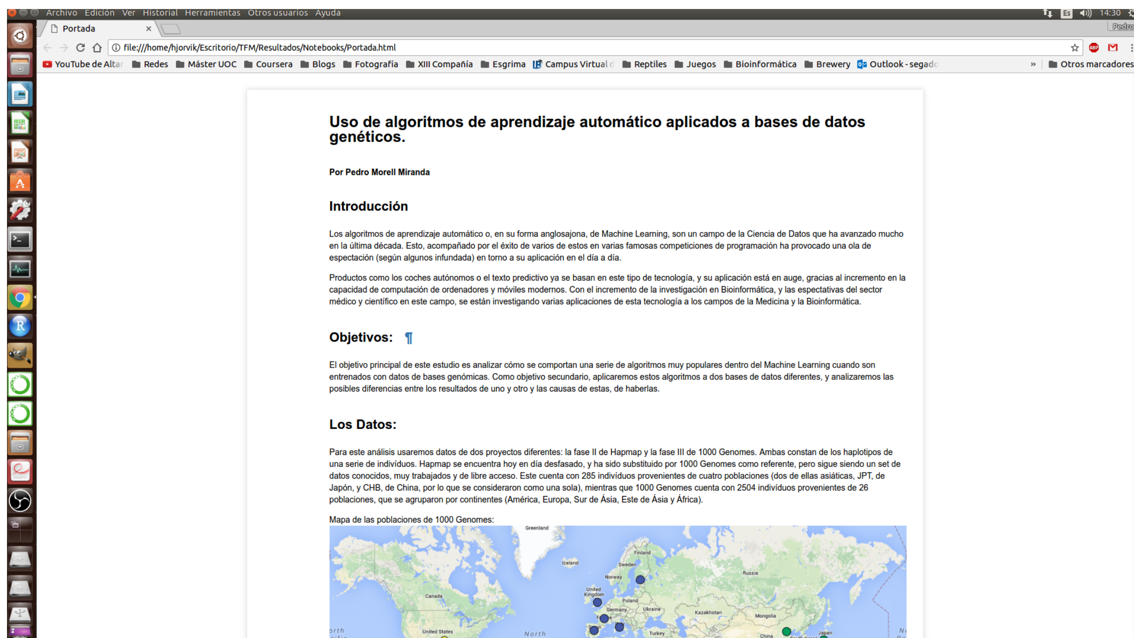


Diagrama de Gantt con la planificación del trabajo

1.5 Breve sumario de productos obtenidos

El producto final es una página web que resume el contenido de los análisis y enlaza a diferentes páginas para cada algoritmo y a una página final en la cual se compara el rendimiento de todos ellos.



Captura de pantalla de la página principal de la web.

1.6 Breve descripción de los otros capítulos de la memoria

- Los datos: Descripción de los proyectos de donde obtenemos los datos y descripción de los mismos.
- Hipótesis: Descripción de las hipótesis previas de lo que esperamos con este análisis.
- Pre-procesado y transformación de los datos: Capítulo donde se describe el proceso llevado a cabo para adaptar los datos a los algoritmos de Machine Learning.
- Algoritmos usados: Descripción del funcionamiento de los diferentes algoritmos empleados.
- Entrenando los Algoritmos: descripción de los métodos empleados para seleccionar el mejor clasificador en cada caso.
- Evaluando el Rendimiento de los Modelos: descripción de los métodos y herramientas seleccionados para evaluar el rendimiento de los diferentes clasificadores usados

2. Los datos:

Los datos que usaremos para los análisis son los de los proyectos [HapMap](#) (fase II) y [1000 Genomes](#) (fase III).

HapMap es un proyecto, iniciado en 2002, para crear un mapa del haplotipo humano, y su última fase se lanzó en 2010. Cuenta con 285 individuos de 3 poblaciones. A partir de 2016, este proyecto se retiró de la web del NCBI en beneficio de 1000 Genomes, pero sigue estando disponible para descargar en el servidor FTP.

Los datos están agrupados en un archivo por cada una de estas poblaciones y cada cromosoma, y en formato tabular, con las variantes como filas. En las columnas tenemos las 10 primeras con información como el cromosoma, la posición de la variante, genotipo salvaje/original, institución que hay procesado el genoma, etc., mientras que el resto de columnas corresponden a los diferentes individuos.

1000 Genomes es un proyecto iniciado entre 2008 y 2015, cuyo objetivo era crear el catálogo de variación genotípica humana más grande hecho público hasta la fecha. Al ser posterior a Hapmap, este proyecto se benefició de la rápida evolución en tecnologías de Next Generation Sequencing (NGS), por lo que presenta una mayor cantidad de individuos y mayor número de variantes detectadas.

Mapa 1000 Genomes

Los datos de este proyecto están agrupados por cromosoma en formato vcf, un formato tabular mucho más opaco que el de Hapmap. Sin embargo, este formato, característico del análisis de datos provenientes de NGS cuenta con varias herramientas que facilitan su procesado.

Por último, la lista de SNPs que usaremos la obtenemos mediante una búsqueda en Gene, la base de datos de genes del NCBI.

3. Hipótesis:

Este análisis contará con varias fases:

- La primera, donde se explorarán los datos, comprobará si las variantes seleccionadas son características de las poblaciones de donde proviene cada muestra (si los individuos muestran agregación entre ellos).
- En la segunda parte, se analizará el rendimiento de cada base de dato con los diferentes algoritmos de clasificación seleccionados. En este caso esperamos que 1000 Genomes muestre un mejor rendimiento que Hapmap, ya que presenta muchos más ejemplares para entrenar.
- Sin embargo, aprovechando que tenemos dos tipos de etiquetas (grupo y supergrupo poblacional), evaluaremos también el rendimiento de ambos, donde esperamos que los supergrupos sean más fáciles de clasificar correctamente.

4. Pre-procesado y transformación de los datos:

Para que los datos provenientes de estas bases de datos puedas ser usados para entrenar algoritmos de Machine Learning es necesario llevar a cabo varias modificaciones, dependiendo del formato del archivo.

- Archivos HapMap:

    - Importar los tres archivos de HapMap y la lista con los SNPs. Para esto usamos Pandas, ya que crea una matriz con nombres (denominada Data Frame) y nos ofrece varias funciones asociadas que utilizaremos a lo largo del proceso.
    - Una vez importamos, seleccionamos solo los SNPs presentes en la lista. Hacer esto al principio reduce el tamaño de los archivos y por tanto el tiempo de computación.
    - Transponemos los Data Frames de las poblaciones. Esto significa que ahora los SNPs son las columnas mientras que los individuos y las 10 columnas descriptivas son las filas. Este paso es importante pues de serie, los algoritmos de Machine Learning interpretarán que las filas son las entradas y las columnas las características.
    - Como los valores faltantes están codificados en un formato que Pandas no reconoce, los cambiamos al formato de Numpy NaN.
    - Codificamos cada entrada en los Data Frames de forma que, si muestra un wild type sea 0, si es heterocigoto sea 1, y si es mutante homocigoto sea 2. Los valores NaN se substituyen por la moda del SNP para su población.
    - El último paso es unir los tres archivos.

Index	rs#	SNPalleles	chrom	pos	strand	genome_build	center	protLSID	assayLSID	panelLSID	QC_code	NA06985	NA06991	NA06993	NA06993.dup	NA06994	NA07000	
0	rs7754266	A/G	chr6	94609	+	ncbi_b36	illumina	urn:LSID:il…	urn:LSID:il…	urn:lsid:dc…	QC+	AG	AG	AA	NN	AA	AA	AA
1	rs9393087	C/T	chr6	94901	+	ncbi_b36	sanger	urn:lsid:il…	urn:lsid:sa…	urn:lsid:dc…	QC+	TT	TT	TT	TT	TT	TT	TT
2	rs12192290	A/T	chr6	95272	+	ncbi_b36	sanger	urn:lsid:il…	urn:lsid:sa…	urn:lsid:dc…	QC+	AA	AA	AA	AA	AA	AT	AA
3	rs11962658	A/C	chr6	96774	+	ncbi_b36	sanger	urn:lsid:il…	urn:lsid:sa…	urn:lsid:dc…	QC+	AA	AA	AA	AA	AA	AA	AA
4	rs7742004	C/G	chr6	97749	+	ncbi_b36	perlegen	urn:lsid:pe…	urn:lsid:pe…	urn:lsid:dc…	QC+	GG	GG	GG	NN	GG	GG	GG
5	rs2107722	A/C	chr6	98500	+	ncbi_b36	perlegen	urn:lsid:pe…	urn:lsid:pe…	urn:lsid:dc…	QC+	CC	CC	CC	NN	CC	CC	CC
6	rs1929630	A/C	chr6	99536	+	ncbi_b36	affymetrix	urn:LSID:af…	urn:LSID:af…	urn:lsid:dc…	QC+	AC	AC	CC	NN	CC	CC	CC

Datos Hapmap sin procesar.

Index	rs1000117	rs1000399	rs1000492	rs1000976	rs1001210	rs1001211	rs1001289	rs1001290	rs1001383	rs1001384	rs1001509
NA18500	2	2	2	0	2	0	0	0	0	2	0
NA18501	2	2	2	0	2	0	0	0	0	1	0
NA18502	2	1	2	0	2	0	0	1	0	2	0
NA18503	2	1	2	0	2	0	0	1	2	2	0
NA18504	2	1	2	0	2	0	0	1	1	2	0
NA18505	2	0	2	0	2	0	0	1	2	2	0
NA18506	2	2	2	0	2	0	0	2	0	1	0

Datos Hapmap una vez pre-procesados y listos para utilizar.

- Archivos 1000 Genomes: En este caso solo tenemos un archivo, perteneciente al cromosoma en cuestión. Para el pre-procesado de este usaremos, además de Numpy y Pandas, PyVCF, una librería específica para trabajar con archivos en formato VCF, y la herramienta del terminal GATKtools (McKenna et al.,2010), desarrollada por el Broad Institute para trabajar con archivos provenientes de NGS.

  - El primer paso es usar la herramienta Variant Selector de GATKtools en el terminal, seleccionando el genoma de referencia usado por 1000 Genomes, el archivo y la lista de genes. Esto nos deja con un archivo VCF solo con las variantes del listado.
  - Para importar el archivo VCF usamos PyVCF. Este es un *lazy parser.* Esto quiere decir que no cargará en memoria todo el archivo, sino solo aquellas entradas que requiramos. Esto es importante para trabajar con archivos VCF, pues fácilmente pueden llegar a varios gigas (52 Gb pesa el archivo original del cromosoma 6). En caso de intentar acceder al archivo entero para procesarlo, el ordenador se quedaría sin memoria disponible, y el sistema se caería.
  - Iterando sobre cada SNP presente en el archivo, creamos una serie con todos los individuos y su genotipo ya codificado (PyVCF tiene una herramienta para esto), y luego la almacenamos en un Data Frame. Este es un proceso bastante lento, pero nos permite construir en un solo paso el Data Frame con el formato en que queremos los datos.
  - Al terminar de iterar, tenemos una matriz con genotipos codificados, individuos en las filas y SNPs en las columnas.

```
NA21099 NA21100 NA21101 NA21102 NA21103 NA21104 NA21105 NA21106 NA21107 NA21108 NA21109 NA21110 NA21111 NA21112 NA21113 NA21114 NA21115 NA21116 NA21117 NA21118 NA21119 NA21120 NA21122 NA21123 NA21124
NA21125 NA21126 NA21127 NA21128 NA21129 NA21130 NA21133 NA21135 NA21137 NA21141 NA21142 NA21143 NA21144
6	12005957	rs146869298	G	T	100	PASS	AA=G|||;AC=19;AF=0.00379393;AFR_AF=0.0113;AMR_AF=0;AN=5008;DP=20409;EAS_AF=0;EUR_AF=0.004;NS=2504;SAS_AF=0;VT=SNP	GT	0|
0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|
0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|
0	0|0	0|0	0|1	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|
0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|
0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|
0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	1|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|
0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|
0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|
0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|
0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|
0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|
0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|
0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|
0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|0	0|
```

Datos de 1000 Genomes en formato VCF.

Index	rs146869298	rs376111606	rs114425758	rs530077249	rs180829290	rs10947708	rs539138946	rs552361360	rs566287732
HG00096	0	0	0	0	0	0	0	0	0
HG00097	0	0	0	0	0	0	0	0	0
HG00099	0	0	0	0	0	0	0	0	0
HG00100	0	0	0	0	0	0	0	0	0
HG00101	0	0	0	0	0	1	0	0	0

Datos de 1000 Genomes una vez procesados.

5. Algoritmos utilizados:

La mayoría de los algoritmos utilizados, así como otras funciones auxiliares (métricas, técnicas de Cross-Validation, etc.) pertenecen a la librería de aprendizaje automático scikit-earn. Esta librería, basada también en Numpy, es la más completa no solo en algoritmos, sino en funciones adicionales para facilitar el diagnostico y la selección de parámetros. El único algoritmo que no proviene de scikit-learn es HDBSCAN, que actualmente se encuentra en la versión en desarrollo de la librería y aun no ha sido incluido.

Para representar los diferentes resultados gráficamente usamos matplotlib, una librería basada en las opciones de representación gráfica de Matlab, completamente compatible con Numpy y sus derivados.

5.1 Clustering:

Los algoritmos de clustering o aglomeración forman parte de lo que se conoce como Machine Learning no supervisado (ya que no conocemos a priori qué clases tienen los datos), y se suele usar para explorar la distribución de los datos.

5.1.1 K-Means (Slonim et al., 2013, Hartigan, 1975):

Este es, probablemente, el método de clustering más común. Fue propuesto por Hugo Steinhaus en 1957, y poco después Stuart Lloyd desarrolló lo que se conoce como el algoritmo estándar, aunque este no salió de los laboratorios Bell hasta 1983. En 1975 Hartigan y en 1979 Wong propusieron una versión más eficiente del mismo, pero, pese a ser esta última la más usada actualmente, este sigue conociéndose como el algoritmo de Lloyd.

Este es un algoritmo que usa una técnica de refinamiento iterativo. Esto significa que el algoritmo realizará una serie de pasos para obtener un mejor resultado, hasta que la mejora se detiene, y llega a la conclusión de que esa es la mejor versión.

K-Means se llama así debido a que k, el número de clusters, deber ser definido. Una vez definido, el algoritmo asigna cada uno de los n ejemplos a uno de los k clusters, con el objetivo de minimizar las diferencias entre miembros del mismo cluster y aumentar las diferencias entre los clusters.

A menos que k y n sean muy pequeñas, es imposible computar los clusters óptimos de entre todas las posibles combinaciones. Por tanto, el algoritmos utiliza una aproximación heurística para encontrar las soluciones óptimas locales. Esto significa que empieza con una estimación inicial, y en cada iteración la va modificando ligeramente para

comprobar si los cambios mejoran la homogeneidad dentro de cada cluster.

La forma de comparar la homogeneidad es mediante un cálculo de distancias muy similar al de otros algoritmos, como k-Nearest Neighbours (kNN). Estos algoritmos usan los valores de las features como coordenadas en un hiperplano, y luego calculan la distancia euclidea entre los puntos.

El primer paso al ejecutar el algoritmo es seleccionar k puntos al azar que servirán como centro de los clusters. Este tipo de asignación hace que el resultado final dependa, en cierta medida, de qué centroides son elegidos en un principio, y hay varias técnicas para evitarlo (por ejemplo, asignar de forma aleatoria cada punto a un cluster, pasando directamente al siguiente paso), pero todos ellos añaden nuevos sesgos que debemos tener en cuenta.

Una vez asignados los centros iniciales, cada punto es asignado al cluster de cuyo centro esté más cerca. Esto se realiza mediante un diagrama de Voronoi generado a partir de los centros:

$$S_i^{(t)} = \left\{ x_p : ||x_p - m_i^{(t)}|| \leq ||x_p - m_j^{(t)}|| \forall 1 \leq j \leq k \right\}$$

Fórmula del Diagrama de Voronoi.

Como ya hemos dicho, el algoritmo clásico utiliza la distancia euclidea, aunque otras fórmulas para calcular la distancia entre los puntos son también aplicables, como la de Manhattan o la de Minkowski.

$$d(x,y) = \sqrt{\sum_i^n (x_i - y_i)^2} = \sqrt{\sum_i x_i^2 + \sum_i y_i^2 - 2\sum_i x_i y_i}$$

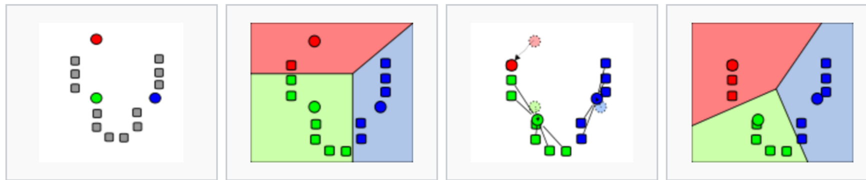
Fórmula de la Distancia Euclídea

Una vez que la fase de asignación (de cada punto a su cluster más cercano) se ha completado, el algoritmo pasa a la fase de actualización. El primer paso para esta fase es cambiar la localización de los centros iniciales. Esta nueva localización es conocida como centroide, y es calculada como la posición media de los puntos asignados actualmente a un cluster.

$$m_i^{(t+1)} = 1/|S_i^{(t)} \sum_{x_j \epsilon S_i^{(t)}} x_j$$

Fórmula para calcular el Centroide.

Demostración del algoritmos estándar



1) *k* centroides iniciales (en este caso *k*=3) son generados aleatoriamente dentro de un conjunto de datos (mostrados en color).

2) *k* grupos son generados asociándole el punto con la media más cercana. La partición aquí representa el diagrama de Voronoi generado por los centroides.

3) EL centroide de cada uno de los *k* grupos se recalcula.

4) Pasos 2 y 3 se repiten hasta que se logre la convergencia.

Ejemplo simplificado de las fases de Asignación y Actualización

Como ya hemos explicado, todos estos pasos dependen de un número de clusters que el usuario impone a los datos. En caso de que el número de clusters no sea adecuado para los datos, el resultado no tendrá sentido. Si fijamos un número demasiado bajo de clusters, habrá falta de homogeneidad dentro de los clusters, pero si elegimos un valor demasiado grande, nos arriesgamos al overfitting. Por tanto, es vital saber encontrar cuál es el número de clusters óptimo para nuestros datos.

Lo ideal es tener un conocimiento *a priori* de la naturaleza de nuestros datos que nos pueda servir para elegir la cantidad de clusters. Sin embargo, habrá veces en que no conozcamos *a priori* la cantidad de clusters. Para estos casos, existen varias técnicas, aunque la más utilizada es el “método del codo”. Este se basa en ejecutar el algoritmo con diferentes valores de k y representar gráficamente la variación de homogeneidad o heterogeneidad dentro de los clusters. Como ya hemos dicho anteriormente, el máximo está en un cluster para cada individuo, por lo que el objetivo no es maximizar o minimizar estas medidas, sino buscar la k a partir de la cual la variación de homogeneidad (o heterogeneidad) desciende.

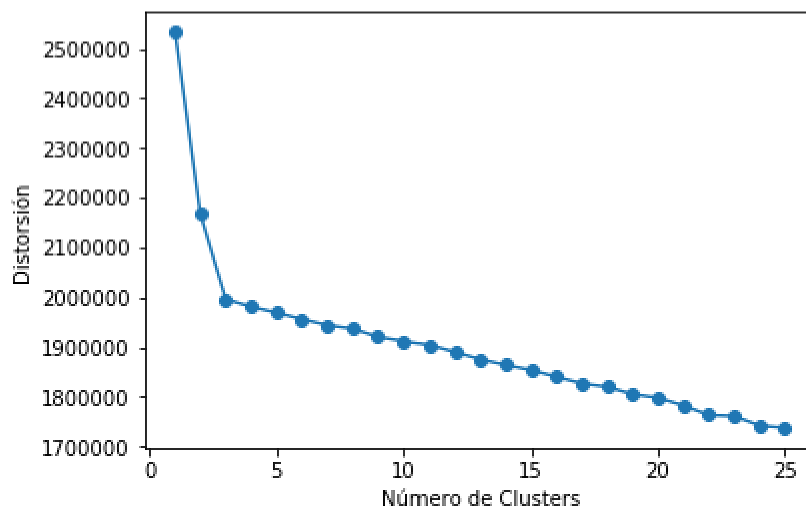
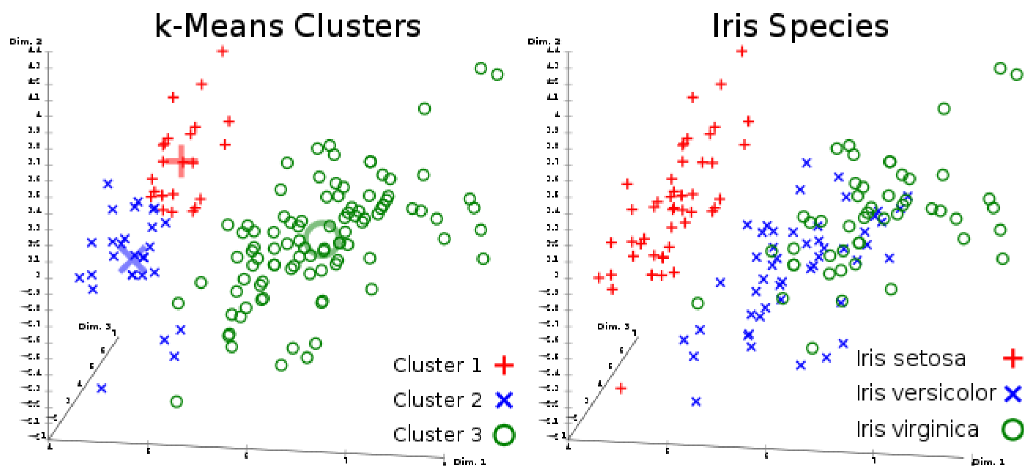


Gráfico de la Heterogeneidad (Distorsión) para cada k.

Fortalezas	Debilidades
Usa principios simples que se pueden explicar en términos no estadíscicos	No tan sofisticado como otros algoritmos de clustering más modernos
Muy flexible, y puede ser adaptado con ajustes sencillos	Como usa un elemento de aleatoriedad, no asegura encontrar el set óptimo de clusters
Se comporta de forma bastante competente en la mayoría de problemas del "mundo real"	Requiere estimar el número de clusters
	No es bueno para clusters no esféricos o con una densidad variable

La mayor parte de las ventajas y desventajas de esta lista ya las hemos repasado, pero K-Means tiene una desventaja importante, derivada de su formulación, que no hemos tratado: no es capaz de tratar con clusters no esféricos. Esto se debe a cómo se asigna la pertenencia a un cluster para cada punto (se le asigna al cluster cuyo centroide esté más cerca). Esto significa que clusters continuos, pero lineales, o esféricos pero que se solapen, serán erroneamente asignados.



Ejemplo de clasificación erronea de datos lineales o solapados con K-Means

Además de esto, tanto este como el resto de algoritmos de clustering que usaremos utilizan distancias euclideas, y las medidas no están estandarizadas. Esto significa que padecerán lo que se conoce como la «maldición de la dimension» o el efecto Hughes. Esto significa que, a más dimensiones tengan nuestros datos, más problemas tendrá el algoritmo para calcular las distancias. Para reducir este efecto, podemos aplicar una PCA.

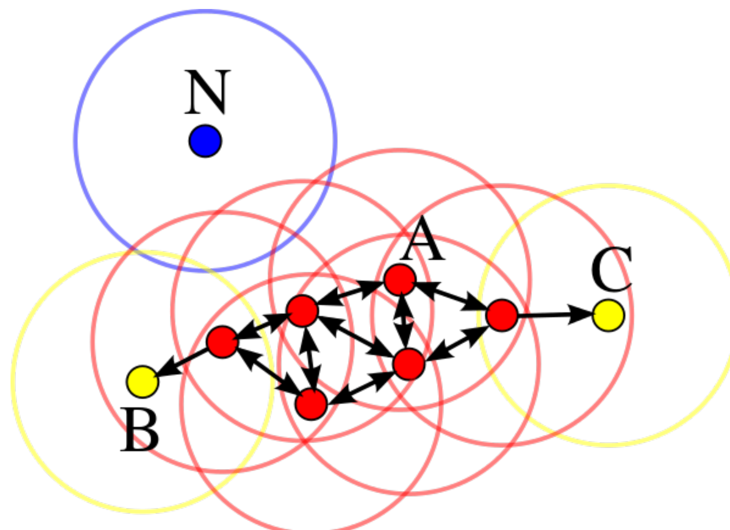
Como ya hemos mencionado, este es uno de los algoritmos de clustering más populares, por lo que era imprescindible aplicarlo a nuestros datos y comprobar cómo se comportaba.

5.1.2 DBSCAN (Ester et al., 1996):

El algoritmo DBSCAN (Density-based Spatial Clustering of Applications with Noise, o Clustering espacial de aplicaciones con ruido basado en densidad) fue propuesto por Ester et al. en 1996, y es un algoritmo de clustering que se basa en identificar las regiones con n o más puntos a una distancia determinada (epsilon), siendo n y epsilon valores seleccionados por el usuario.

Si consideramos un conjunto de puntos para clusterizar, DBSCAN los clasificará en tres tipos de puntos:

- Núcleo: un punto p es un punto núcleo si al menos n puntos estás a una distancia epsilon de él y estos puntos son directamente alcanzables desde p (que no tiene otro punto en medio).
- Borde o Frontera: un punto q es un punto Borde si es alcanzable desde p, es decir, si está a una distancia epsilon o menos de al menos un punto núcleo, pero de menos de n.
- Ruido: Un punto que no cumple ninguna de las condiciones anteriores será considerado como ruido.



Clusterización con DBSCAN: En rojo, los núcelos, en amarillo los Bordes, y en azul el Ruido.

Cada cluster debe tener, como mínimo, un núcleo, y una serie de puntos alcanzables desde este. Los puntos Borde forman parte del cluster, pero actúan como frontera, aislando los núcleos de otros posibles puntos alcanzables.

Un concepto importante es que la relación de ser alcanzable no es simétrica. Ningún punto puede ser alcanzable desde un punto que no sea un núcleo, sin importar la distancia a la que esté. Esto significa que un punto que no sea núcleo puede ser alcanzado, pero nada puede ser alcanzado desde este.

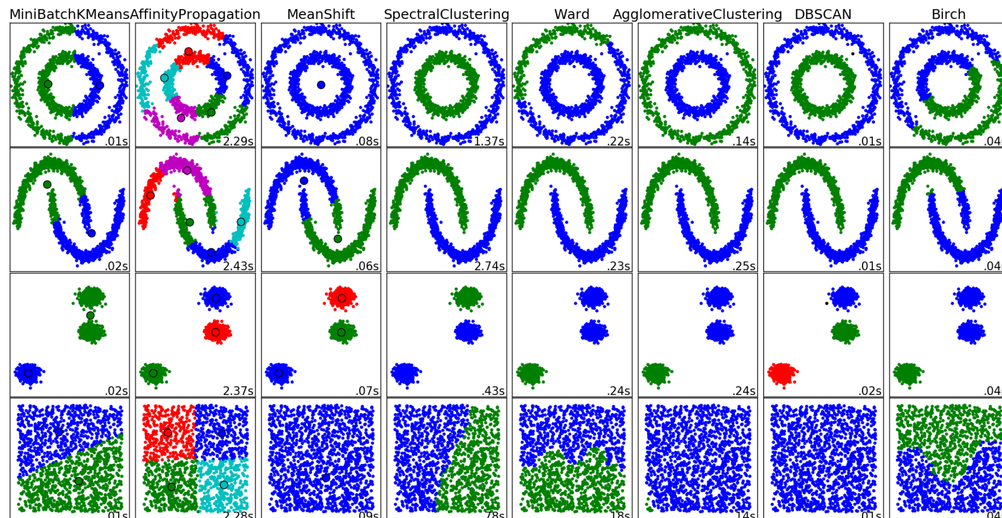
Existen algunas variantes, como DBSCAN* (Campello et al, 2013) que no hacen esta distinción entre Núcleos y Bordes, dando como resultado

un clustering menos excluyente. Como desventaja, este algoritmo funciona mucho peor gestionando el ruido.

Fortalezas	**Debilidades**
No es necesario preasignar el número de clusters	Depende de la distancia entre puntos (Maldicion de dimensionalidad)
Funciona bien con clusters no esféricos	Tiene problemas con densidades variables
Mayormente insensible al orden de los datos	A veces presenta problemas separando clusters muy cercanos

En comparación con K-Means, este algoritmo tiene la clara ventaja de que la elección del número de clusters no depende del usuario, que la distribución de estos viene dada por la distribución de los puntos y no seleccionada aleatoriamente y que, al depender de la distancia entre un punto y otro, este algoritmo no es sensible a la forma del cluster ni al orden de los datos.

Como desventajas, es extremadamente sensible a la Maldición de dimensionalidad, además de ser un algoritmo relativamente conservador cuando se aplica a clusters con densidades variables, ya que tiende a definir como bordes puntos que podrían alcanzar otros puntos del cluster, que serán clasificados como ruido. También tenemos el caso contrario: clusters muy cercanos pueden ser clasificados como uno mismo si alguno de sus puntos Borde son considerados como Núcleo.



Clusterización de diferentes tipos de datos y como se ajustan diferentes algoritmos a estos.

Como podemos ver en la figura anterior, DBSCAN se comporta bien con los clusters con formas no circulares y con clusters a diferentes distancias, pero falla en diferenciar los clusters cuando los puntos presentan una densidad homogénea.

Aplicamos este algoritmo para compararlo con los resultados de K-Means, y para ver si las limitaciones de aquel hacían que el clustering fuese muy diferente.

5.1.3 HDBSCAN (Campello et al.,2013) :

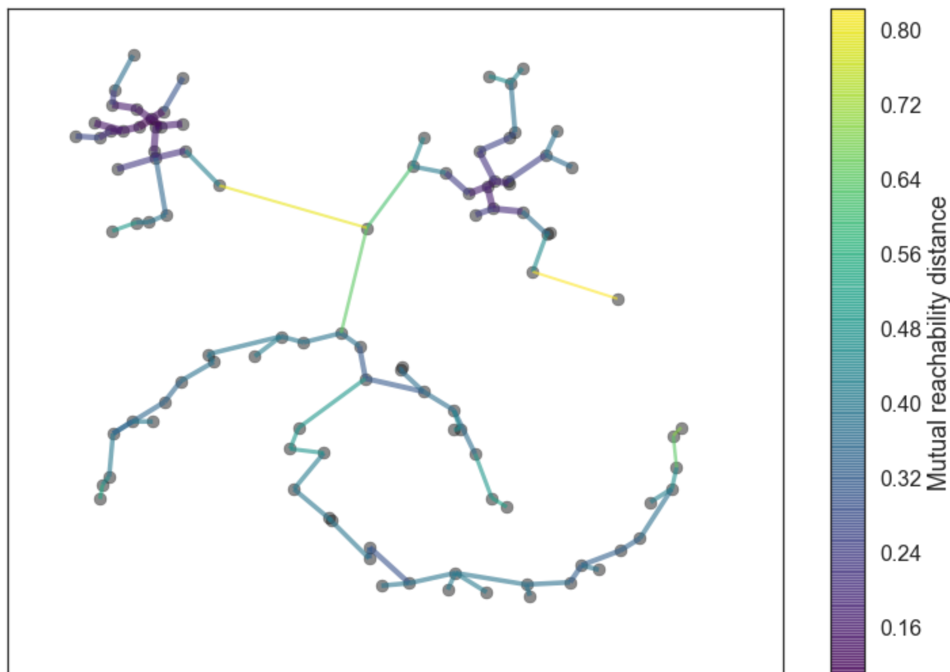
Este algoritmo, desarrollado por Campello et al. (los mismos autores de DBSCAN*), extiende DBSCAN convirtiéndolo en un algoritmo de clusterización jerárquica, utilizando entonces una técnica para extraer una “clusterización plana” (Campello et al. 2013) basada en la estabilidad de los diferentes grupos.

El primer paso para la ejecución de HDBSCAN es la creación de una matriz de distancias a la que aplicar un algoritmo sencillo y rápido, como k-Nearest Neighbours, que identifique las regiones con mayor densidad. A partir de esta, se calcula la Distancia de Alcance Mutuo, que mantiene igual la distancia entre puntos en zonas de alta densidad, pero incrementa la de los puntos poco agregados.

$$d_{mreach-k}(a, b) = max[core_k(a), core_k(b), d(a, b)]$$

Fórmula de la Distancia de Alcance Mutuo, donde d(a,b) es la métrica original de distancia entre a y b.

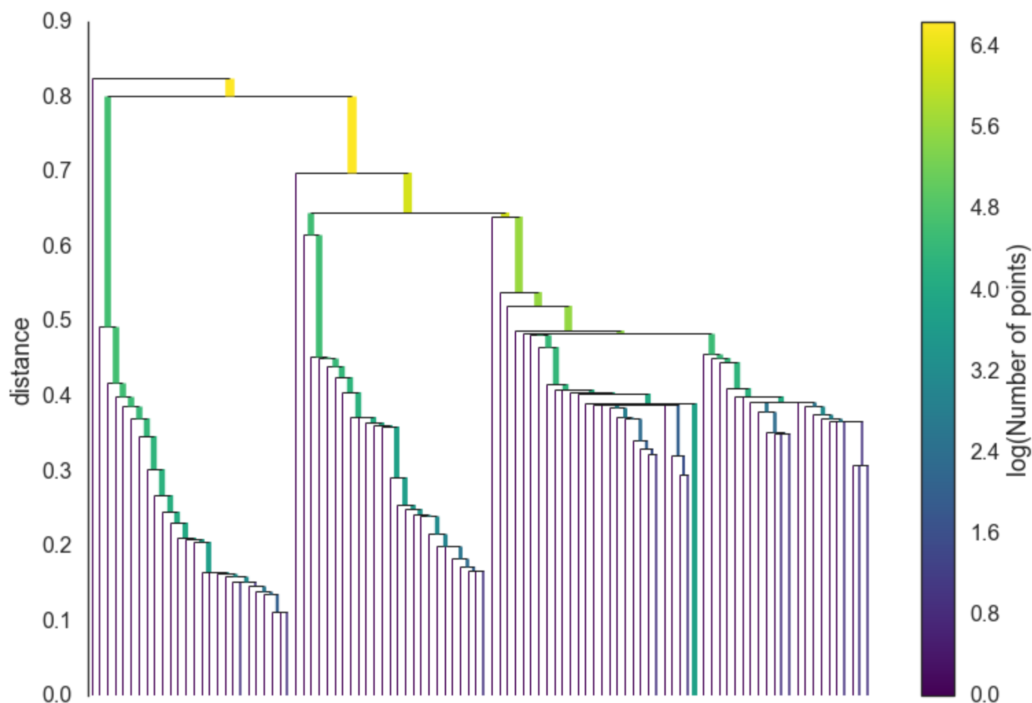
A continuación el algoritmo usa el algoritmo de Pilgrim (Jarnik, 1930), un “Greedy algorithm” que encuentra el árbol con menos ramificaciones (Least Spanning Tree) entre los diferentes puntos.



Ejemplo de Árbol con menos ramificaciones

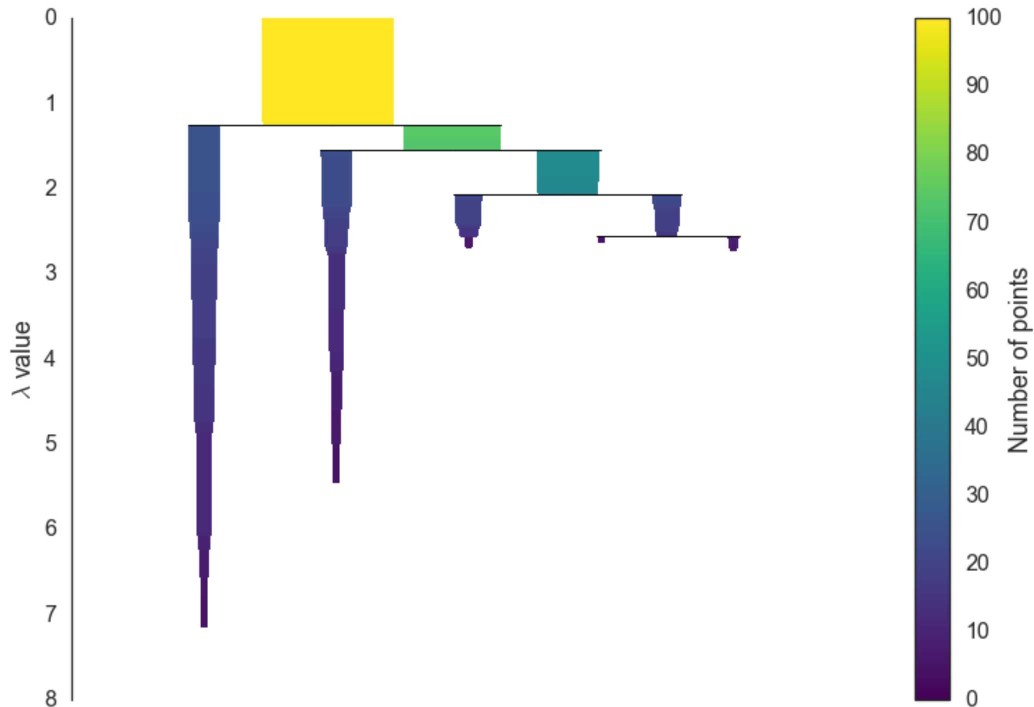
Una vez tenemos el Árbol con menos ramificaciones, el algoritmo sigue el proceso contrario para crear una jerarquía de componentes conectados: en lugar de buscar un centro e ir desplazándose hacia los

bordes, ordenamos los bordes del árbol por distancia e iteramos, justando los bordes cercanos. El resultado es similar al siguiente dendrograma:



Dendrograma con la clusterización de los bordes.

Hasta ahora, los que hemos realizado es un Robust Single Linkage (Sibson, 1973). Para conseguir un "cluster plano", necesitamos condensar el árbol de clusters, y para ello usamos DBSCAN. El primer paso es reducir este árbol mediante el único hiperparámetro que necesita HDBSCAN: el tamaño mínimo del cluster. Con este valor, podemos recorrer la jerarquía, preguntando en cada división si una de las dos ramas tiene menos puntos que el tamaño mínimo. Si los tiene, consideramos que estos puntos están "fuera del cluster", y si no, estos heredarán la clasificación. Tras recorrer todo el arbol, el dendrograma anterior termina de esta forma:



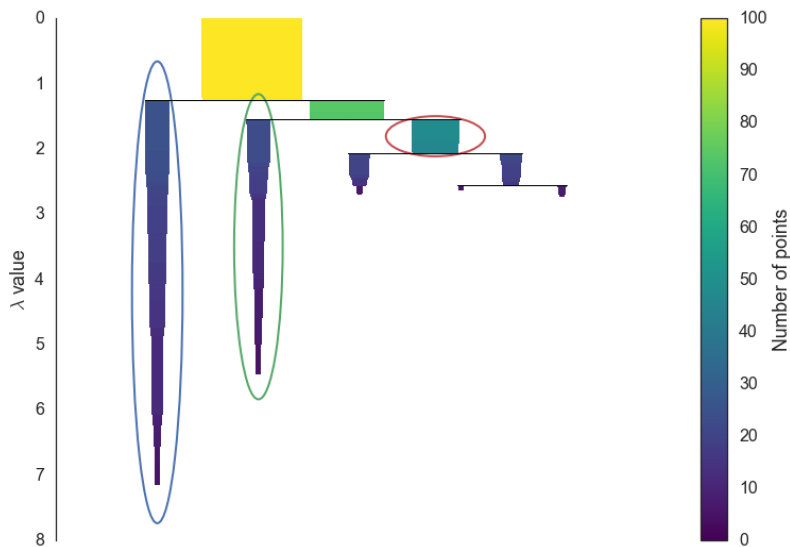
Dendrograma del árbol jerárquico reducido

Una vez aquí, extraemos los clusters. Para ello, en lugar de usar la distancia, usamos su inversa (λ). Definimos la estabilidad de un cluster como la suma de la diferencia entre la λ de cada punto y la del inicio del cluster de los puntos que pertenecen a este:

$$\sum_{p \epsilon cluster} (\lambda_p - \lambda_{birth})$$

Fórmula de la Estabilidad del Cluster

Si la suma de la estabilidad de los clusters "hijos" (más bajos en la jerarquía) es mayor que la del "padre", esta pasa a ser considerada. Si la del cluster "padre" es mayor, este se selecciona como cluster seleccionado, y sus descendientes como agregaciones dentro de este.



Dendrograma con los clusters seleccionados.

Fortalezas	Debilidades
Basado en DBSCAN	Lento de procesar
No es necesario asignar la épsilon	Puede no funcionar bien con datos muy grandes
Más conservador que DBSCAN	
Funciona mejor con clusters cercanos	

Este algoritmo tiene como ventaja que calcula de forma automática la distancia óptima, por lo que se ajusta mucho mejor a los datos que DBSCAN, funcionando mejor con clusters cercanos, mientras mantiene muchas de las ventajas de este.

Por el contrario, la principal desventaja que tiene es que es muy lento de entrenar, pues *de facto* está procesando varios algoritmo de clustering.

Utilizamos este algoritmo para compararlo con DBSCAN, pues es menos sensible al efecto Hughes. Además, es un método de clustering que últimamente está ganando bastante popularidad.

5.2: Clasificación:

Los algoritmos de clasificación pertenecen al grupo del Machine Learning supervisado, ya que necesitan de un set de entrenamiento en el que las categorías sean conocidas. Estos se basan en ajustar diferentes modelos matemáticos a los datos, para poder clasificar nuevas entradas (con categoría desconocida) a una de las categorías para las que ha sido entrenado el modelo.

5.2.1 Naives Bayes (Domingos & Pazzani, 1997):

Los conceptos estadísticos en que se basa este tipo de algoritmos para clasificar han existido durante siglos, teniendo sus bases en el propio Teorema de Bayes y los conocidos como Métodos Bayesianos.

Los clasificadores basados en métodos Bayesianos usan los datos de entrenamiento para calcular una probabilidad observada para cada clase basada en la evidencia aportada por las features. Cuando el clasificador es aplicado a datos sin etiquetar, este utiliza esas probabilidades para la clase más probable de acuerdo a los valores de las features.

En general, los clasificadores Bayesianos se aplican típicamente a problemas donde la información de numerosos atributos debe ser considerada simultáneamente para poder estimar la probabilidad de un evento.

Aunque Naive Bayes no es el único algoritmo de Machine Learning que usa métodos Bayesianos de probabilidad para clasificar, sí es el más común de ellos, sobretodo en clasificación de texto, donde se ha convertido en el algoritmo estándar.

Fortalezas	**Debilidades**
Simple, rápido y muy efectivo	Presupone igualdad de importancia e independencia de las features
Puede manejar datos ruidosos o desaparecidos	No es muy bueno para datasets con gran cantidad de features numéricas
Necesita relativamente pocos ejemplos para entrenar, pero también se comporta bien con sets grandes	Las probabilidades estimadas son menos fiables que las clases predichas
Fácil obtener las probabilidades estimadas para una predicción	

Naive Bayes es llamado así (ingenuo) debido a que asume que existe independencia entre las features, lo que raramente ocurre. En nuestro caso, es de suponer que las variaciones cercanas en el cromosoma se hereden juntas, por ejemplo. Sin embargo, la facilidad para entrenarlo y el buen resultado de este algoritmo en una gran variedad de aplicaciones hacen que esta sea una asunción aceptable en muchos casos.

Además de asumir la independencia entre features, las implementaciones de Naive Bayes de scikit-learn dependen de la distribución de los datos. Siendo estos una gran cantidad de variantes

diferentes, presuponemos normalidad, eligiendo el algoritmo GaussianNB (Zhang, 2004).

$$P(x_i|y) = 1/\sqrt{2\pi\sigma_y^2} exp(-(x_i - y_y)^2/2\sigma_{)}^2$$

Fórmula que usa Gaussian NB para calcular la similaridad entre features.

Elegimos este algoritmo porque es un buen clasificador, muy fácil de entrenar y que suele funcionar bien con datos muy diversos.

5.2.2 Random Forest (Breiman, 2001):

Este algoritmo, desarrollado por Breiman y Cutler, se basa en utilizar un método de meta-aprendizaje, utilizando varios clasificadores (en este caso Árboles de decisión) que, combinados, terminan dando un modelo de conjunto (o como se conoce en inglés, un *ensemble mode*l).

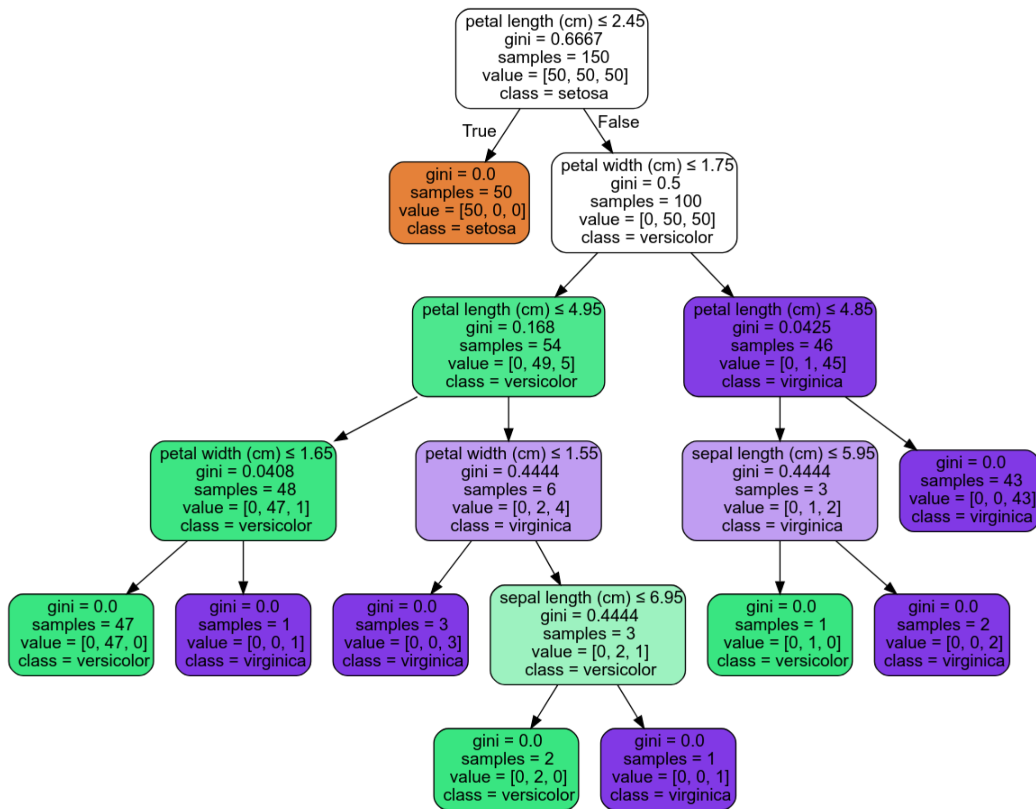
El método que usa Random Forest para generar diferentes modelos a partir de los datos se conoce como *boostrap aggregatting*, o *bagging (Breiman, 1996)*. Este método genera una serie de datasets de entrenamiento a partir de los datos originales mediante *boostrap sampling*, para luego ser usados para entrenar un solo modelo con cada uno de los datos de entrenamiento. Posteriormente, las predicciones de estos modelos son combinadas mediante votos (para clasificadores) o medias (para regresores).

En el caso de Random Forest, como ya hemos dicho antes, el algoritmo usado para el bagging es un Árbol de Decisión (Breiman et al., 1984). Este algoritmo utiliza una serie de reglas sencillas (basadas en estrategías Divide y Vencerás) para clasificar a los diferentes individuos.

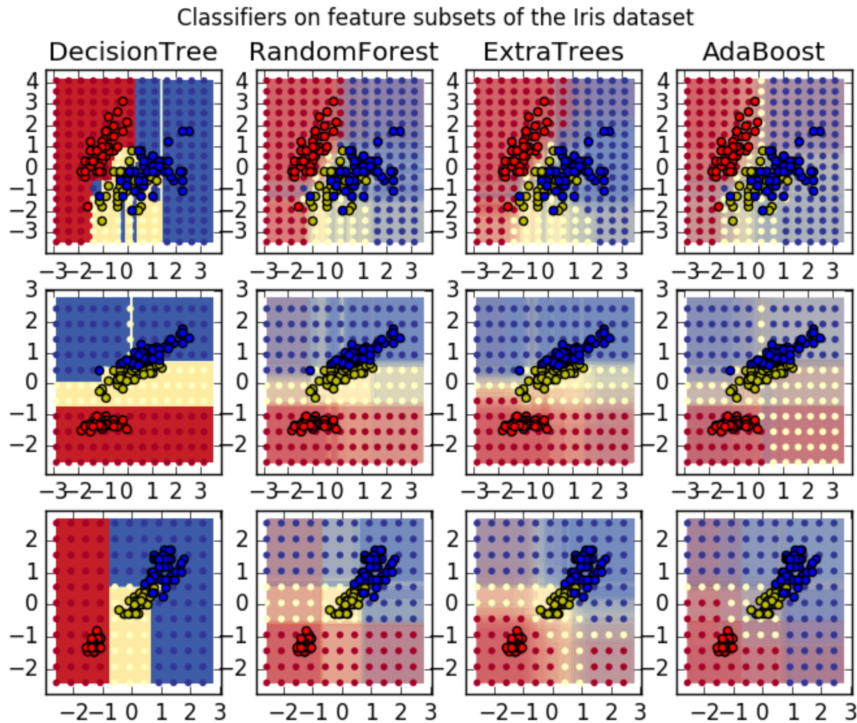
Además del bagging, este algoritmo aplica una Feature Seleccion, para añadir diversidad adicional a los arboles de decisión.

Este algoritmo combina versatilidad con potencia en una sola aproximación al Machine Learning. Como el ensemble solo usa un pequeño subconjunto aleatorio de los datos, Random Forest puede trabajar con datasets increíblemente grandes, donde la “maldición de la dimensionalidad” haría fracasar a otros algoritmos.

Si los comparamos con otros métodos de ensemble, Random Forest es más fácil de entrenar y tiene menos tendencia al overfitting.



Ejemplo de Árbol de Decisión



Comparativa de diferentes algoritmos de ensemble.

El algoritmo incluido en scikit-learn, en lugar de usar el método empleado por Breitman & Cutlet para predecir, basado en votaciones, utiliza las probabilidades de cada modelo, en una aproximación Bayesiana.

Fortalezas	**Debilidades**
Modelo "todo-terreno"	Al contrario que un Árbol de Decisión, no es facilmente interpretable
Puede manejar datos ruidosos o desaparecidos al igual que features contínuas o categóricas	Difícil ajustar a los datos
Selecciona solo las features importantes	
Puede ser usado con datos con muchas features	

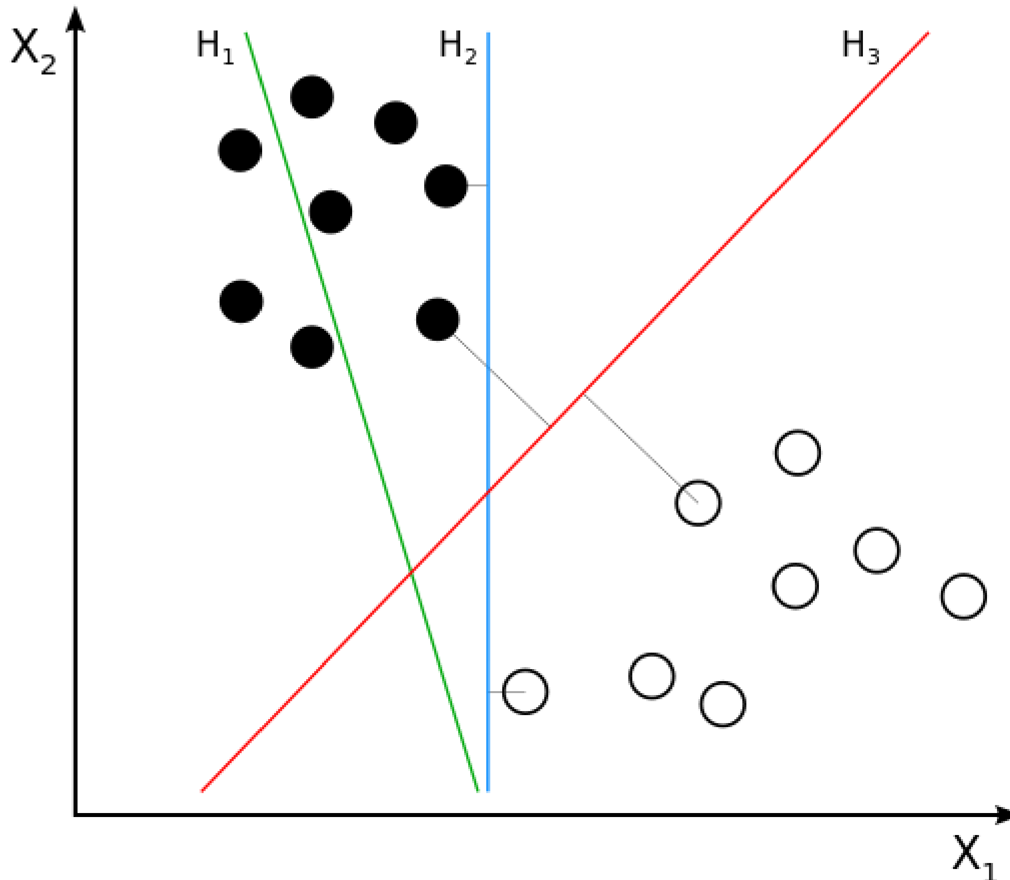
Este es un algoritmo de amplia aplicación, pues es capaz de procesar una gran cantidad de datos en relativamente poco tiempo (aunque esto depende del número de árboles que queramos entrenar) y se comporta bien con grandes cantidades de datos.

Elegimos este algoritmo debido a que este funciona bien, en teoría, con datos con muchas features.

5.2.3 Support Vector Machine (SVM) (Suykens & Vandewalle, 1999, Furey et al. 2000):

Una SVM puede ser imaginada como una superficie que crea la unión entre puntos de los datos representados de forma multidimensional, siendo cada punto un ejemplo y sus features. El objetivo de una SVM es crear un plano de unión, llamado hiperplano, que divida el espacio de forma que cree particiones homogéneas para cada lado. De esta forma, las SVM utilizan conceptos de Nearest Neighbours instanciados y de modelos de regresión lineal para crear una poderosa herramienta de clasificación, permitiendole a los SVM crear relaciones muy complejas.

El objetivo de este algoritmo es el de encontrar un hiperplano óptimo que separe los puntos de una clase de los de otra. Para eso, utiliza la separación óptima: el Maximum Margin Hyperplane (Hiperplano de máximo margen, MMH). El MMH es el hiperplano que separa ambas clases de forma que el margen entre este y los puntos más cercanos de cada clase sean lo mayor posible.



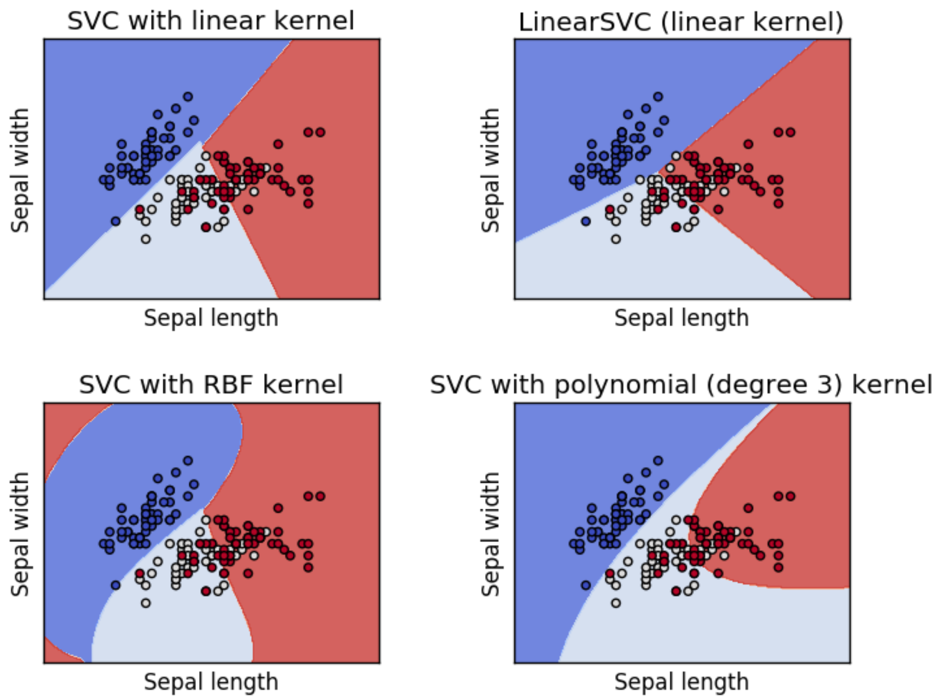
Ejemplo de SVM. H1 no separa las clases, H2 las separa pero está muy cerca de los puntos, mientras que H3 se corresponde con la definición de MMH.

Los vectores de suporte, o Support Vectors que le dan nombre al algoritmo, son los puntos más cercanos al hiperplano. Cada clase debe tener, por lo menos, uno, aunque pueden ser varios. Usando solo los vectores de soporte es posible definir el MMH, característica clave de los SVM, ya que provee de una forma muy compacta de almacenar el modelo de clasificación.

Como no siempre es posible ajustar un hiperplano de forma que todos los puntos terminen en el lado correspondiente, el algoritmo presenta un parámetro C que controla la compensación de estos errores de entrenamiento y los márgenes rígidos, creando un márgen blando que permita algunos errores de clasificación a la vez que los penaliza.

La forma más sencilla de establecer un hiperplano es siempre la línea recta. Sin embargo, no siempre nos van a permitir nuestros datos hacer esto. Debido a las limitaciones computacionales de las máquinas de aprendizaje lineal, estas no suelen poder ser usadas en aplicaciones del mundo real. Sin embargo, las representaciones por medio de funciones Kernel (Hoffman et al., 2008) ofrece una solución a este problema,

proyectando la información a un espacio con mayor dimensionalidad, el cual aumenta la capacidad computacional de la máquina de aprendizaje lineal. Dependiendo del tipo de Kernel que usemos, obtendremos diferentes hiperplanos:



Ejemplo de aplicación de SVM con diferentes Kernels.

Fortalezas	Debilidades
Puede ser usado para clasificación o problemas numéricos	Encontrar el mejor modelo requiere probar diferentes kernels e hiperparámetros
No muy influenciable por datos ruidosos y sin mucha tendencia al overfitting	Lento de entrenar, sobre todo con datos con muchas features
Popular debido a su alta Accuracy y a recientes éxitos en competiciones de Data Mining	El resultado es un complejo modelo "caja negra", prácticamente imposible de interpretar
Más fáciles de usar que las Redes Neurales	

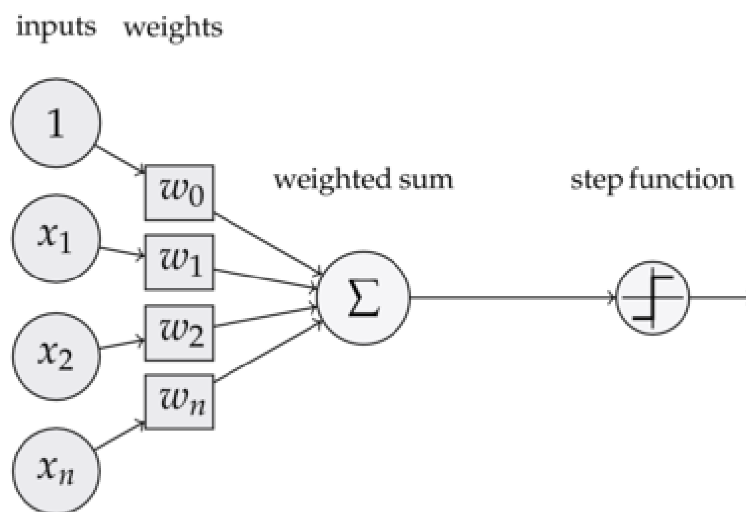
Además de las ventajas listadas anteriormente, hay que destacar que este tipo de algoritmos suele ser comparado (y usado en los mismo campos) que las Redes Neurales más sencillas, como el Multi-Layered Perceptron (MLP). Ambos tipos de algoritmos están considerados de “caja negra”, pues generan modelos tan complejos que es imposible analizar cómo se toman las decisiones. Sin embargo, el hecho de que los SVM sean más fáciles de entrenar, tengan menos tendencia al overfitting y los modelos entrenados sean más robustos y generalizables hacen que este tipo de algoritmos sean muy populares en comparación.

Este algoritmo lo elegimos porque se suelen ajustarse bien a modelos relativamente complejos y no tiene tendencia al sobreajuste (overfitting). Además de eso, este tipo de algoritmos son muy populares.

5.2.4 Redes Neurales Artificiales (Hinton, 1985, Glorot & Bengio, 2010, Kingma et al., 2014):

Este tipo de algoritmo imita la forma en que las neuronas se comportan para crear una red de nodos. El primer modelo artificial de neurona es el conocido como la Neurona McCullock-Pitts, y fue publicado por estos dos matemáticos en 1943. Este modelo describe la neurona como una puerta con un output binario. La información entra por un lado (dendritas), y si la señal supera un cierto nivel, una señal de salida es pasada por el otro lado (axón).

Unos años después se presentaba el primer concepto de Perceptrón (Rosenblatt, 1958). Este algoritmo calculaba automáticamente el peso óptimo de los coeficientes que eran entonces multiplicados a las features para hacer una decisión (si la neurona se activa o no). En un contexto de aprendizaje supervisado, esto significaba que este algoritmo nos permitía predecir si un individuo pertenecía a una clase u otra.



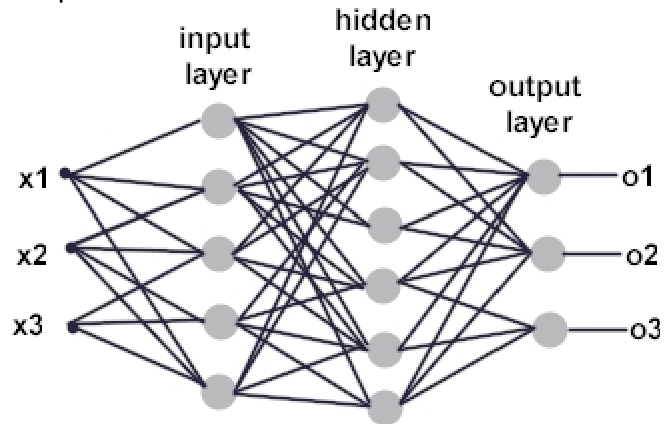
Esquema de un Perceptrón

Que la neurona se active o no depende de dos funciones matemáticas: la función de transferencia , que modula la señal de entrada de acuerdo a unos pesos que son ajustados durante el entrenamiento, y la función de activación, que es la que determina si la señal que le llega tras ser multiplicada por la función de transferencia es suficientemente grande como para activarse o no.

Hay una gran cantidad de funciones de activación, dependiendo del tipo de Red Neural que queramos construir, pero la más común es la función sigmoide.

Pese a que el funcionamiento del Perceptrón fue descrito hace casi 60 años, la verdadera potencialidad de este tipo de algoritmo es que no tiene por qué reducirse a una sola neurona, sino que se pueden crear diseños realmente complejos.

Actualmente el diseño más sencillo de Red Neural que se usa es el Multiple-layered Perceptron (Perceptrón de múltiples capas, o MLP, ya mencionado antes). Este diseño es un Perceptrón con lo que se conoce como una “capa oculta”.



La cantidad de neuronas presentes en cada capa es completamente configurable, y depende totalmente del tipo de datos que tengamos y de lo fácil que le resulte a la Red Neural ajustarse a este. Sin embargo, el funcionamiento de cada neurona es el mismo que el del Perceptrón.

Además de definir la estructura de la Red, definiendo el número de neuronas, también podemos definir si las redes estarán o no completamente conectadas, la dirección en que fluirá la información o si alguna de las capas retendrá parte de la información para usarla más tarde, entre otras cosa. Esto hace que los diseños de las Redes Neurales Artificiales sean a la vez muy potentes y muy difíciles de entrenar.

Últimamente se han hecho muy populares las redes neurales conocidas como Deep Neural Network, es decir, Redes con más de una capa de oculta. Estas Redes presentan un gran potencial para procesar datos complejos, como imágenes, vídeo, o texto, y ciertos laboratorios, como los de IBM o Google han hecho grandes avances en este campo.
Sin embargo, todas las Redes Neurales son muy susceptibles al overfitting debido al gran número de iteraciones que realizan los datos, por lo que hay que utilizar diferentes estrategias para evitar esto (olvido selectivo, backpropagation, etc.).

Fortalezas	Debilidades
Puede ser usado para clasificación o problemas numéricos	Extremamente intenso computacionalmente y lento de entrenar
Capaz de modelar patrones muy complejos	Tendencia al overfittin
Presupone muy pocas cosas sobre los datos	El resultado es un complejo modelo "caja negra", prácticamente imposible de interpretar

En general, las Redes Neurales no suelen ser algoritmo que se apliquen sin un motivo, pues son mucho más costosas computacionalmente que la mayoría de algoritmos. Sin embargo, han demostrado reiteradamente que, para ciertas aplicaciones, merece la pena el esfuerzo que lleva dar con el modelo adecuado para nuestros datos. Desde predicción de textos hasta coches autónomos o manipulación de imagen, las Redes Neurales artificiales presentan gran cantidad de aplicaciones hoy en día.

Elegimos este algoritmo por su gran capacidad para ajustarse a modelos complejos.

6. Entrenando los algoritmos:

En este apartado hablaremos de las estrategias empleadas para entrenar los algoritmos de clasificación. El primer paso para entrenar un algoritmo de Machine Learning es separar los datos en sets de entrenamiento y testeo. Esta separación debe realizarse de forma aleatoria, y el objetivo es poder comprobar cómo se comporta nuestro modelo tras el entrenamiento.

Como Naives Bayes no requiere ningún hiperparámetro (o valores necesarios para que el algoritmo se ejecute), el siguiente paso no se llevó a cabo con durante su entrenamiento. Para el resto, fue necesario comprobar qué combinación de hiperparámetros da los mejores resultados para nuestros datos.

Para encontrar la mejor combinación hay varias aproximaciones. La primera, es a mano, basándonos en nuestro conocimiento sobre los datos y repitiendo el entrenamiento con diferentes valores. Esto es lento, y no nos asegura que vayamos a encontrar la mejor opción. Sin embargo, scikit-learn ofrece un par de funciones que nos permiten encontrar la versión que nos da los mejores resultados:

- Randomized Search CV: Esta función usa un diccionario con una lista de valores para cada hiperparámetro que queramos probar, y , utilizando Cross Validation, analiza n combinaciones aleatorias y se queda con la mejor de entre ellas. La ventaja de este método es que es relativamente rápido, pero como desventaja tiene que puede que no de con la combinación óptima al hacer combinaciones aleatorias.
- Grid Search CV: De forma parecida a Randomized Search, Grid Search necesita un diccionario de Python donde se especifiquen los valores a probar para cada hiperparámetro. Sin embargo, esta función usa una aproximación de “fuerza bruta”. Esto significa que entrena modelos con Cross Validation para cada combinación de valores. Obviamente, este método es lento, sobretodo con algoritmos como SVM o Redes Neurales.

Pese al aumento considerable en el tiempo de entrenamiento, decidimos usar Grid Search CV ya que nos ofrece la seguridad de que estamos usando la mejor combinación.

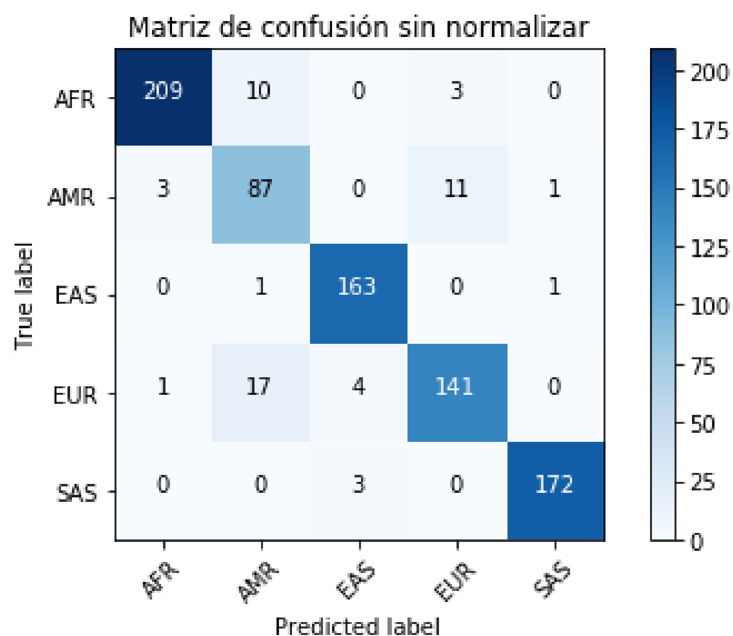
7. Evaluando el Rendimiento de los Modelos:

Para evaluar el rendimiento del modelo, utilizamos tres estrategias diferentes. Además de estas, existen otras métricas y formas de representarlos, pero la mayoría las descartamos porque están pensadas para categorías binarias, y pese a que se puedan adaptar a múltiples categorías, no suelen ser buenas para comparar (como por ejemplo las curvas ROC o la f1).

Como medida estándar para medir el rendimiento usaremos la Accuracy, o el porcentaje de predicciones correctas (Kohavi, 1995).

Además, empleamos tres técnicas para evaluar el rendimiento:

- Matriz de confusión: Esta es una matriz en que se evalúa la concordancia de las predicciones de nuestro clasificador entrenado con respecto a las categorías reales. Esto crea una diagonal de aciertos, y todos los valores que no estén en la diagonal serán fallos. Es una forma muy sencilla de evaluar visualmente el resultado de la predicción de nuestros datos de testeo.

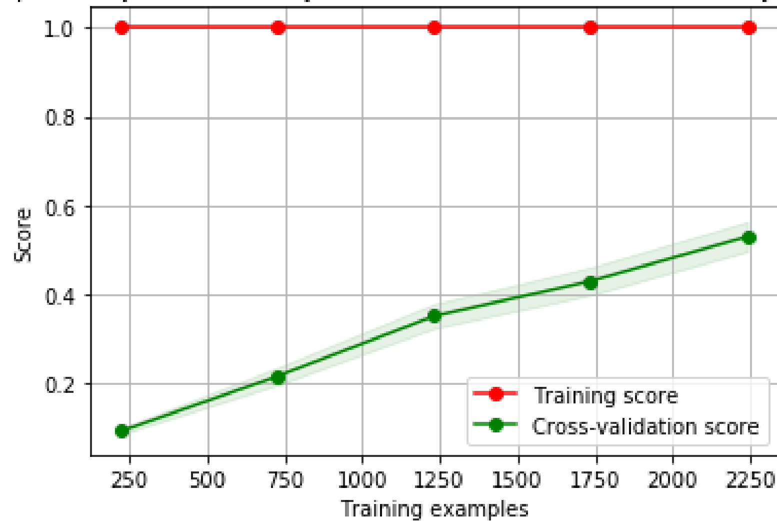


Ejemplo de Matriz de Confusión

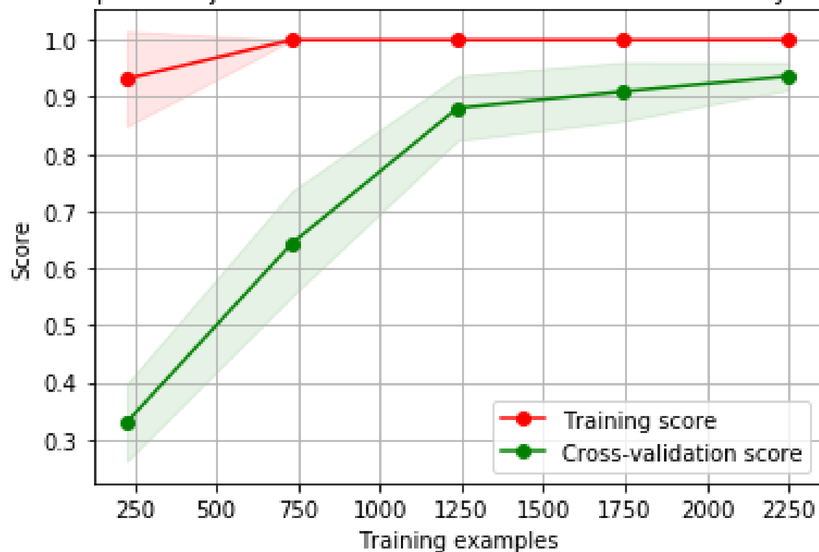
- Cross Validation Score: este método realiza una k-fold Cross Validation (siendo k el número de sub-sets de datos que crea, y la cantidad de modelos diferentes que se crearán con estos) y devuelve la Accuracy de cada intento. Para resumir los resultados, calculamos la media de estos y la varianza. Este

método es más robusto que el simple entrenamiento y testeo realizado con la matriz de confusión ya que estamos probando con k sets de datos diferentes seleccionados aleatoriamente.

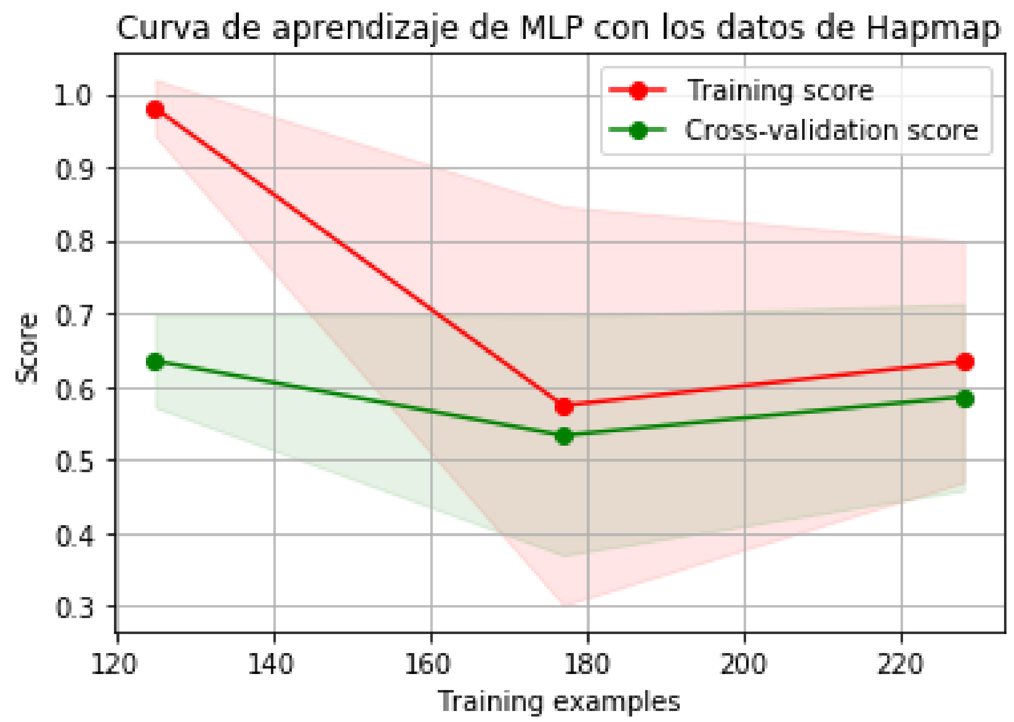
- Curva de Aprendizaje: El gráfico de la Curva de Aprendizaje evalúa cómo se comporta el algoritmo dependiendo de la cantidad de muestras que tenga durante el entrenamiento. Esto es especialmente útil para comprobar si existe overfitting o underfitting.



Curva de aprendizaje con Overfitting



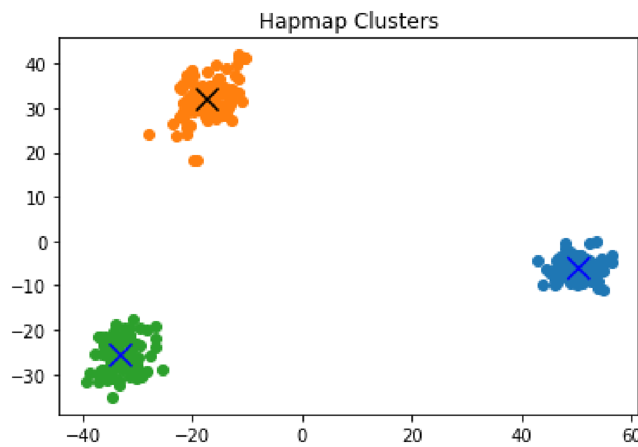
Curva de aprendizaje normal



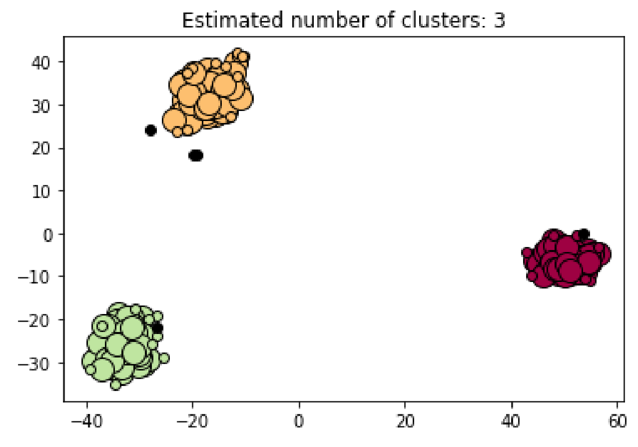
Curva de Aprendizaje con Underfitting

8. Resultados:

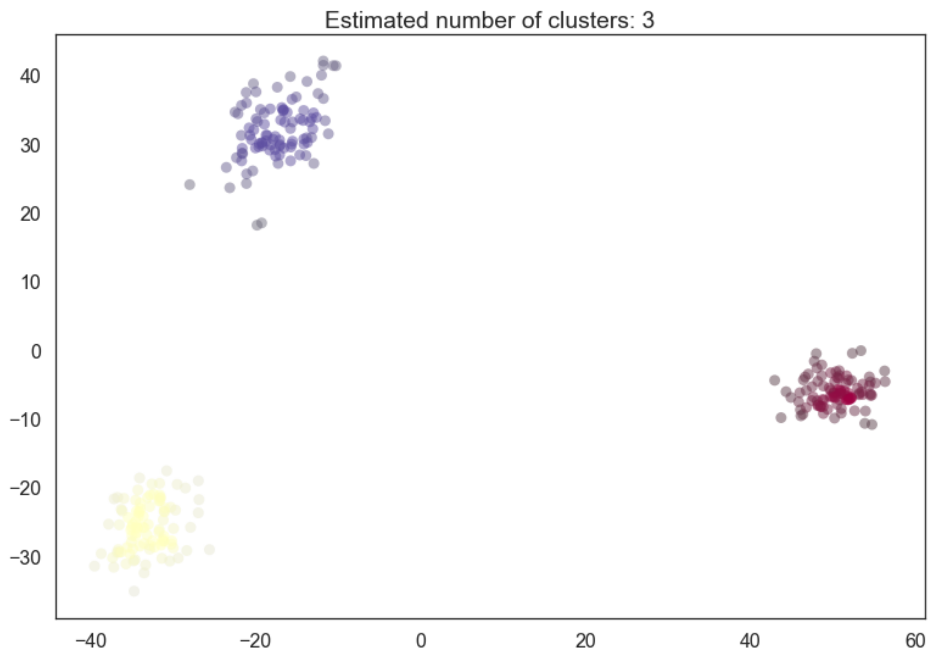
- Clustering:
  Los resultados del clustering son muy dispares entre ambas bases de datos. Por un lado, Hapmap ofrece con los tres algoritmos usados unos resultados bastante buenos, con tres clusters muy discretos:



K-Means de HapMap

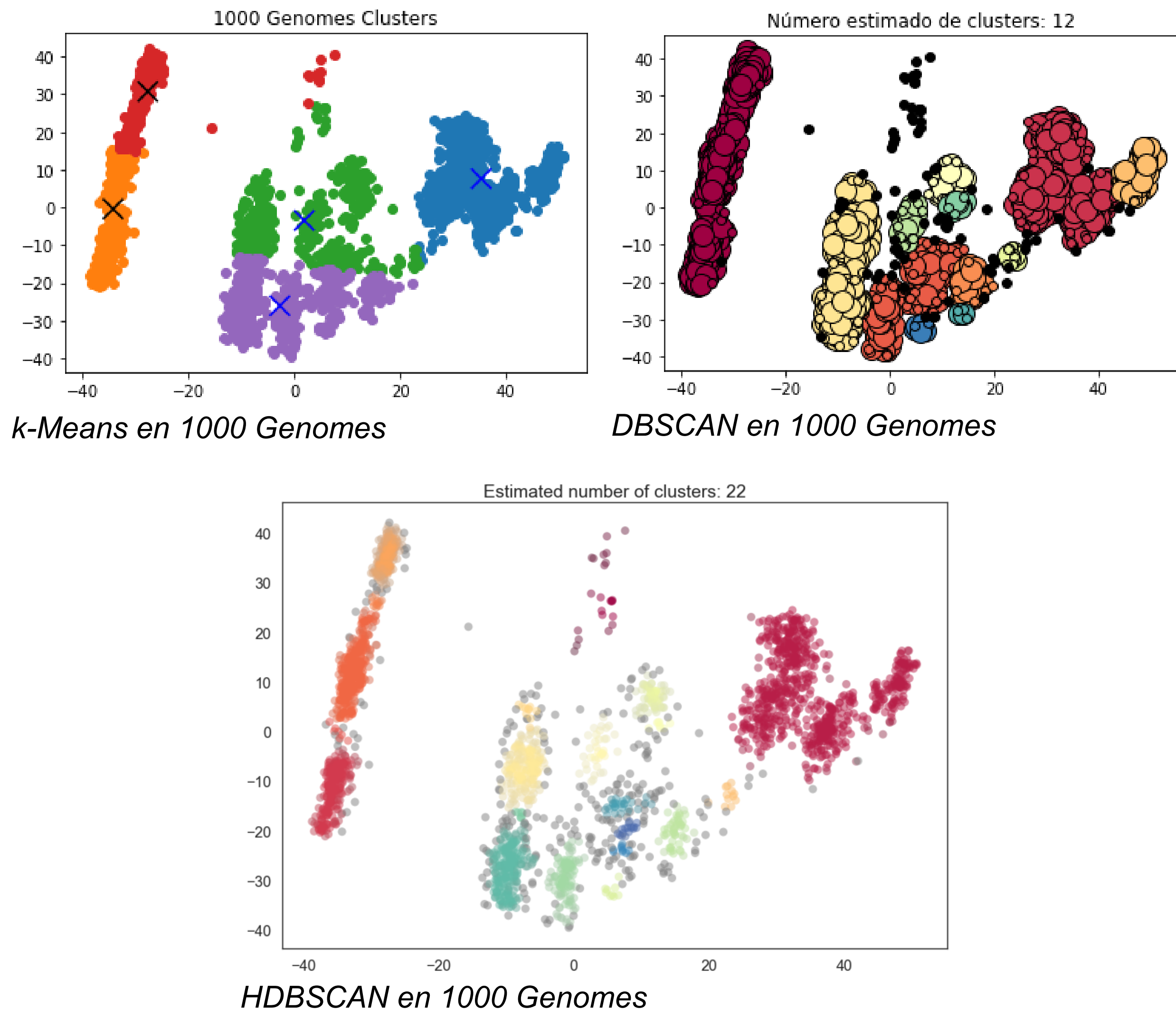


DBSCAN de HapMap



HDBSCAN de HapMap

Los tres métodos funcionan de forma muy dispar, por el contrario, con los datos de 1000 Genomes:

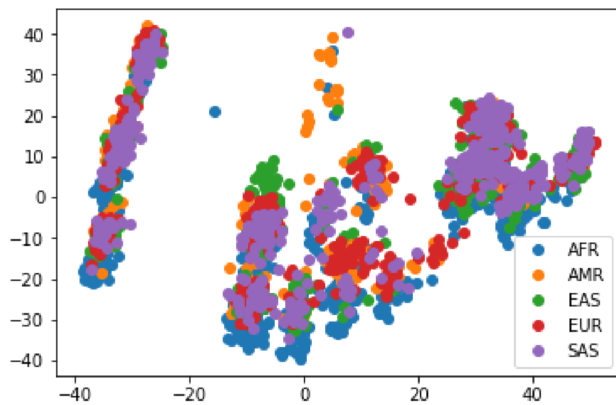


k-Means en 1000 Genomes

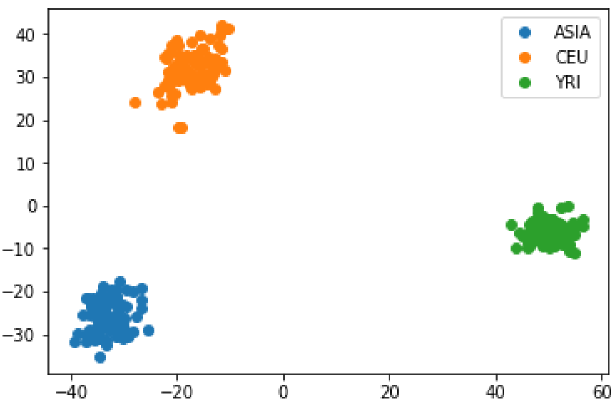
DBSCAN en 1000 Genomes

HDBSCAN en 1000 Genomes

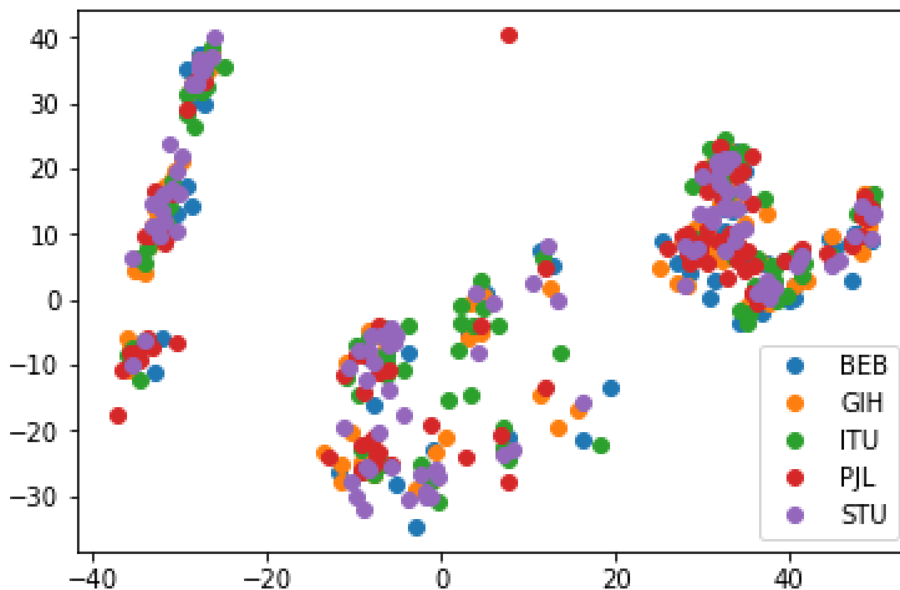
Al ver que ninguno de los algoritmos coincide, ni de lejos, podemos plantearnos diferentes hipótesis. Una, es que las limitaciones de cada algoritmo hagan que ninguno se ajuste a los datos. Este puede ser el caso de k-Means, que es incapaz de lidiar con clusters lineales como los de la mitad izquierda. Sin embargo, ver la diferencia de clusters entre DBSCAN y HDBSCAN, y que esos números no coincidan con las categorías con que trabajaremos más adelante, no deja de llamar la atención. Para ver cómo se distribuyen los datos con respecto a su población de origen, representamos las dos primeras componentes principales (como en los gráficos de arriba) esta vez coloreando por grupo continental:



1000 Genomes por supergrupos



HapMap por localidades



Representación de los diferentes grupos dentro de SAS

Como podemos ver, mientras que HapMap está aglomerado por localidades, 1000 Genomes, pese a tener aglomeraciones visibles en el gráfico, presenta individuos de todos los supergrupos y grupos en cada uno de ellos. Esto, añadido a la linealidad de los clusters, y la variabilidad en la densidad, hace que los resultados de la clusterización sean poco fiables.

Estos resultados confirman nuestra primera hipótesis: los datos muestran una agregación, clara en el caso de HapMap, y ligada claramente a la población de origen, y más difusa y de características desconocidas en el caso de 1000 Genomes.

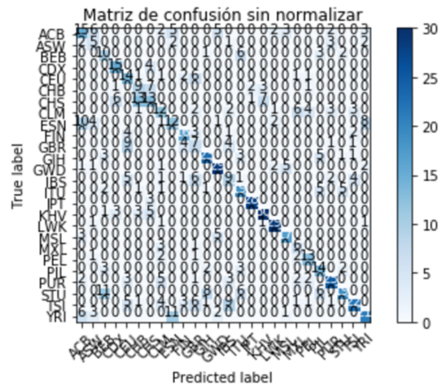
- Clasificación:

A continuación, detallamos los resultados de todos los algoritmos:

- Matrices de confusión:
  - Naive Bayes:

```
In [9]: #Matriz de confusión para el clasificador de 1000 Genomes con grupos
        confmat = confusion_matrix(pred_kg,kg_lab_test)
        plot_confusion_matrix(confmat, classes = np.unique(kg_labels), title='
        plt.show()
        print('Accuracy score:',accuracy_score(kg_lab_test,pred_kg))
```

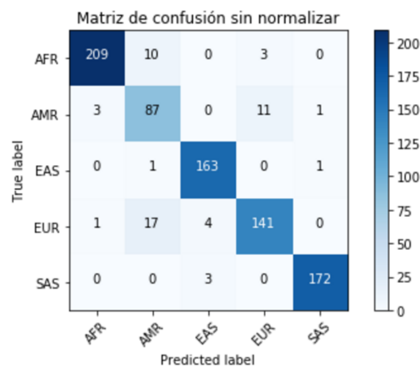
```
Confusion matrix, without normalization
```




```
Accuracy score: 0.522370012092
```

```
In [10]: #Matriz de confusión para el clasificador de 1000 Genomes con supergrupos
         confmat = confusion_matrix(pred_sup,kg_sup_test)
         plot_confusion_matrix(confmat, classes = np.unique(kg_sup_labels), title='
         plt.show()
         print('Accuracy score:',accuracy_score(kg_sup_test,pred_sup))
```

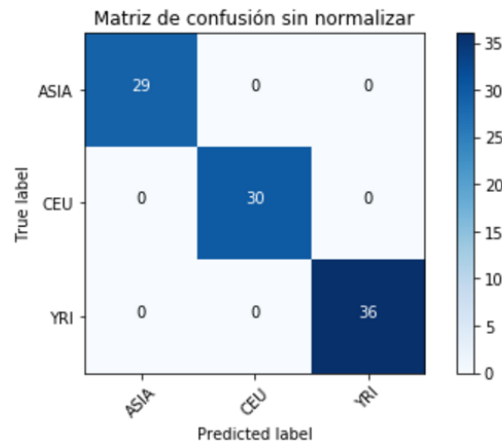
```
Confusion matrix, without normalization
```




```
Accuracy score: 0.933494558646
```

```
In [11]: #Matriz de confusión para el clasificador de Hapmap
confmat = confusion_matrix(pred_hap,hap_lab_test)
plot_confusion_matrix(confmat, classes = np.unique(hap_
plt.show()
print('Accuracy score:',accuracy_score(hap_lab_test,pre
```

```
Confusion matrix, without normalization
```

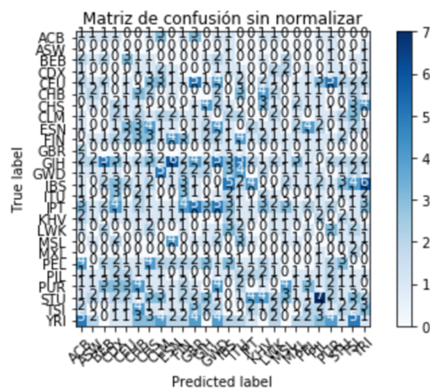



```
Accuracy score: 1.0
```

- Random Forest:

```
In [15]: #Matriz de confusión para el clasificador de 1000 Genomes con grupos
confmat = confusion_matrix(pred_kg,kg_lab_test)
plot_confusion_matrix(confmat, classes = np.unique(kg_labels), title=
plt.show()
print('Accuracy score:',accuracy_score(kg_lab_test,pred_kg))
```

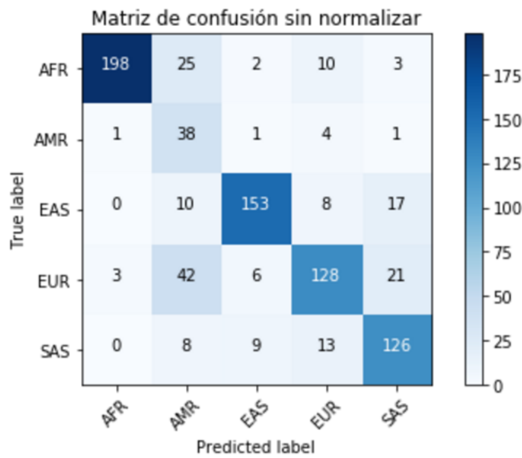
```
Confusion matrix, without normalization
```




```
Accuracy score: 0.037484885127
```

In [16]:
```
#Matriz de confusión para el clasificador de 1000 Genomes con supergrupos
confmat = confusion_matrix(pred_sup,kg_sup_test)
plot_confusion_matrix(confmat, classes = np.unique(kg_sup_labels), title='I
plt.show()
print('Accuracy score:',accuracy_score(kg_sup_test,pred_sup))
```

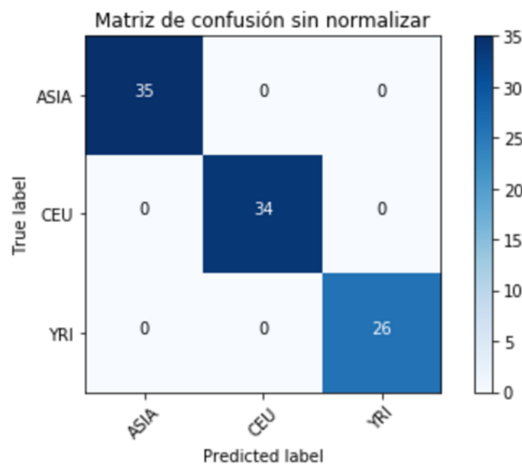
```
Confusion matrix, without normalization
```




```
Accuracy score: 0.777509068924
```

In [17]:
```
#Matriz de confusión para el clasificador de Hapmap
confmat = confusion_matrix(pred_hap,hap_lab_test)
plot_confusion_matrix(confmat, classes = np.unique(hap_
plt.show()
print('Accuracy score:',accuracy_score(hap_lab_test,pre
```

```
Confusion matrix, without normalization
```

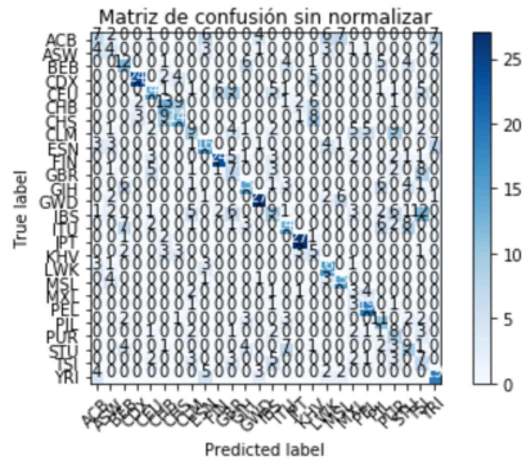



```
Accuracy score: 1.0
```

- Support Vector Machine:

In [28]:
```
#Matriz de confusión para el clasificador de 1000 Genomes con grupos
confmat = confusion_matrix(pred_kg,kg_lab_test)
plot_confusion_matrix(confmat, classes = np.unique(kg_labels), title='M
plt.show()
print('Accuracy score:',accuracy_score(kg_lab_test,pred_kg))
```

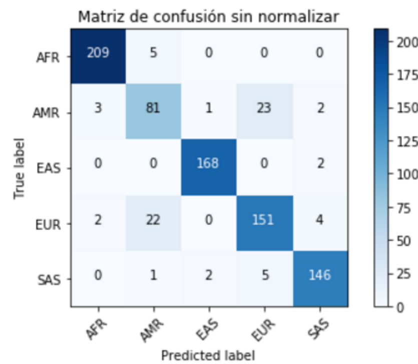
```
Confusion matrix, without normalization
```




```
Accuracy score: 0.420798065296
```

In [34]:
```
#Matriz de confusión para el clasificador de 1000 Genomes con supergrupos
confmat = confusion_matrix(pred_sup,sup_lab_test)
plot_confusion_matrix(confmat, classes = np.unique(kg_sup_labels), title='Ma
plt.show()
print('Accuracy score:',accuracy_score(sup_lab_test,pred_sup))
```

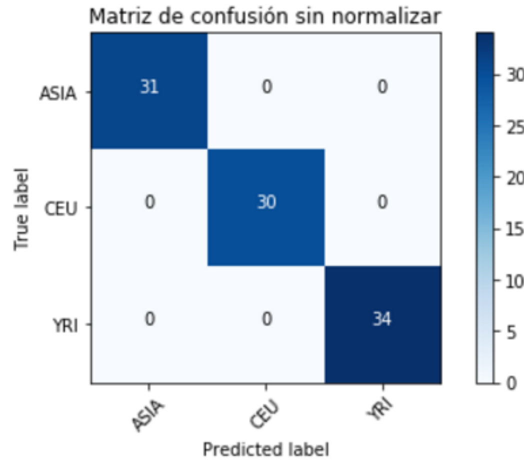
```
Confusion matrix, without normalization
```




```
Accuracy score: 0.912938331318
```

```
In [45]: #Matriz de confusión para el clasificador de Hapmap
confmat = confusion_matrix(pred_hap,hap_lab_test)
plot_confusion_matrix(confmat, classes = np.unique(hap_
plt.show()
print('Accuracy score:',accuracy_score(hap_lab_test,pre
```

```
Confusion matrix, without normalization
```

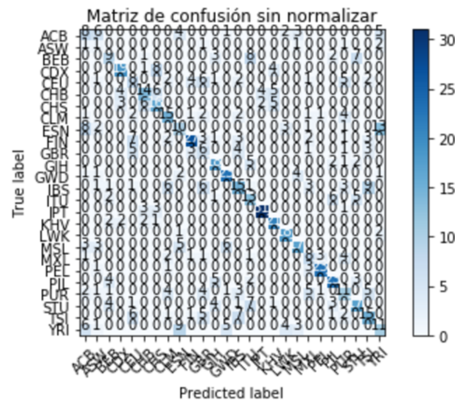



```
Accuracy score: 1.0
```

- Artificial Neural Network:

```
In [96]: #Matriz de confusión para el clasificador de 1000 Genomes con grupos
confmat = confusion_matrix(pred_kg,kg_lab_test)
plot_confusion_matrix(confmat, classes = np.unique(kg_labels), title='
plt.show()
print('Accuracy score:',accuracy_score(kg_lab_test,pred_kg))
```

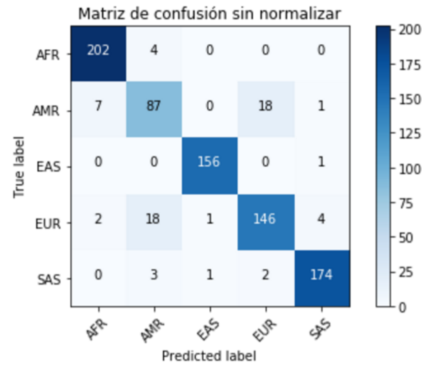
```
Confusion matrix, without normalization
```




```
Accuracy score: 0.480048367594
```

In [108]:
```
#Matriz de confusión para el clasificador de 1000 Genomes con supergrupos
confmat = confusion_matrix(pred_sup,sup_lab_test)
plot_confusion_matrix(confmat, classes = np.unique(kg_sup_labels), title='Ma
plt.show()
print('Accuracy score:',accuracy_score(sup_lab_test,pred_sup))
```

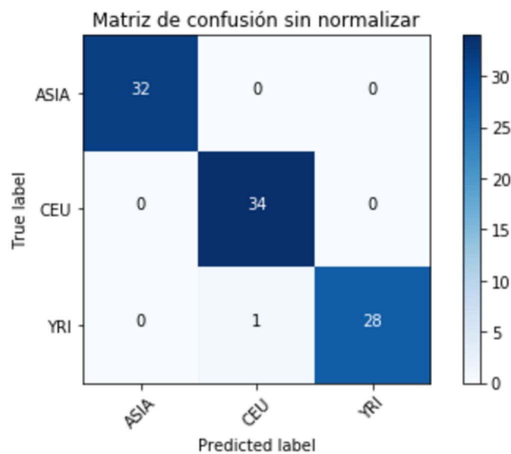
```
Confusion matrix, without normalization
```




```
Accuracy score: 0.925030229746
```

In [109]:
```
#Matriz de confusión para el clasificador de Hapmap
confmat = confusion_matrix(pred_hap,hap_lab_test)
plot_confusion_matrix(confmat, classes = np.unique(hap_label
plt.show()
print('Accuracy score:',accuracy_score(hap_lab_test,pred_hap
```

```
Confusion matrix, without normalization
```



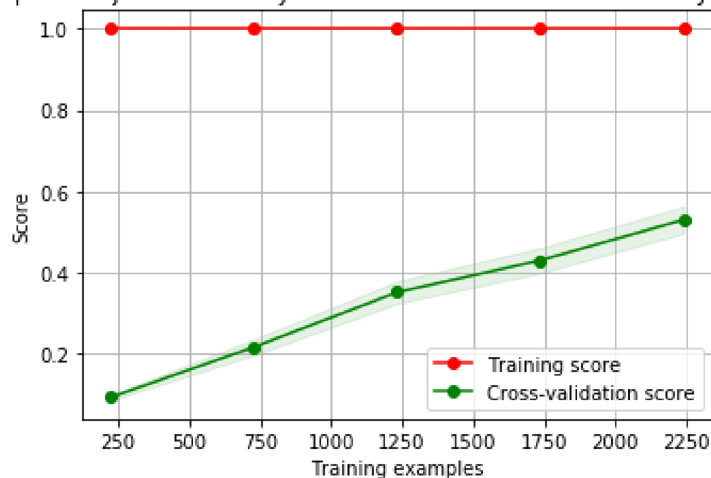

```
Accuracy score: 0.989473684211
```

- Cross validation Score:

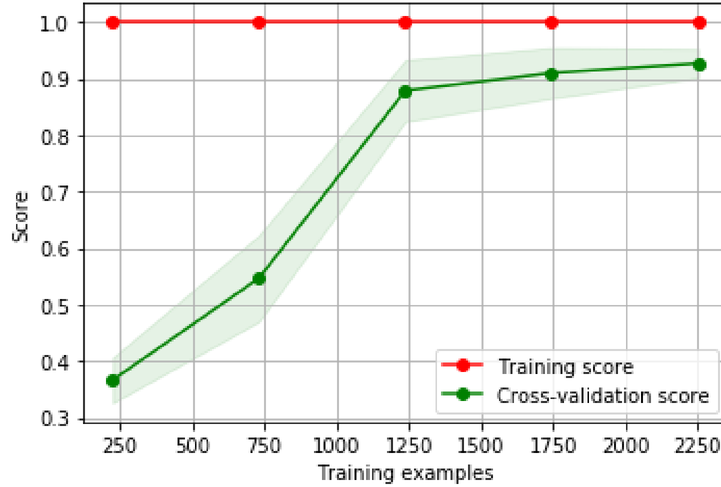
  La siguiente tabla representa los valores de la Accuracy media de los diferentes algoritmos para cada una de las poblaciones tras haber realizado una 10-fold Cross Validation (10 divisiones de datos y 10 iteraciones del algoritmo).

	Naive Bayes	**Random Forest**	**SVM**	**MLP**
1000 Genomes con grupos	0,53 +/- 0,07	0,3 +/- 0,04	0,45 +/- 0,06	0,47 +/- 0,09
1000 Genomes con supergrupos	0,93 +/- 0,05	0,78 +/- 0,08	0,91 +/- 0,05	0,93 +/- 0,06
Hapmap	1	1	1	0,77 +/- 0,42

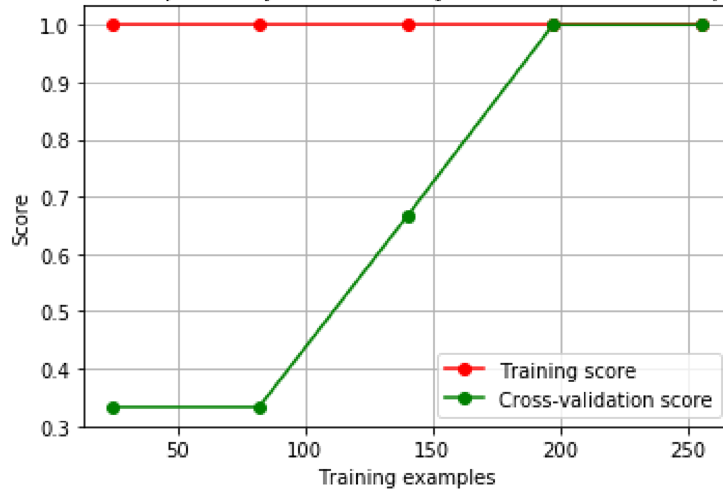
- Curvas de aprendizaje:
  - Naive Bayes:



Learning Curve de Naive Bayes para los datos de 1000 Genomes y Grupos. Se aprecia un fuerte Overfitting.

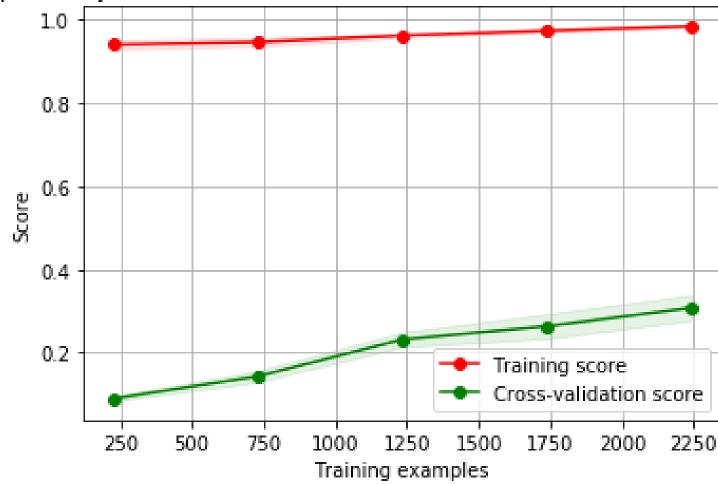


Learning Curve de Naive Bayes para los datos de 1000 Genomes y Supergrupos. El overfitting es despreciable, con una score que supera el 0.9.

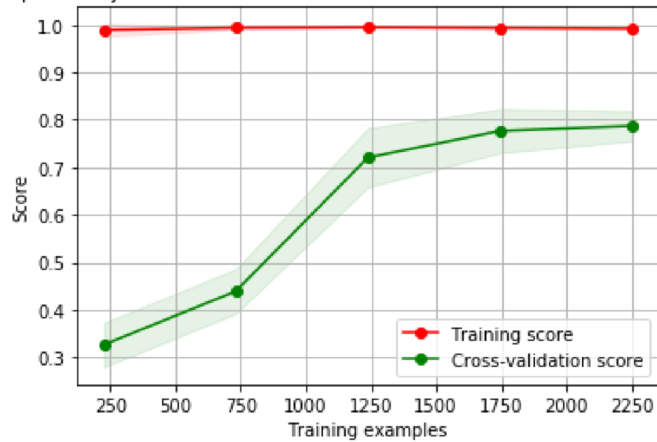


Learning Curve de Naive Bayes para Hapmap. La clasificación es perfecta a partir de 200 ejemplos de entrenamiento.

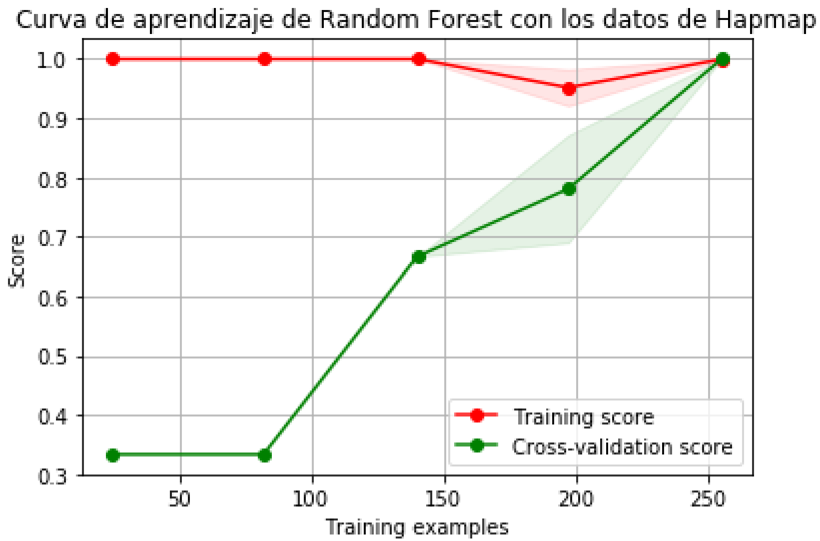
- Random Forest:



Learning Curve de Random Forest para 1000 Genomes y Grupos. Vemos un fuerte overfitting.

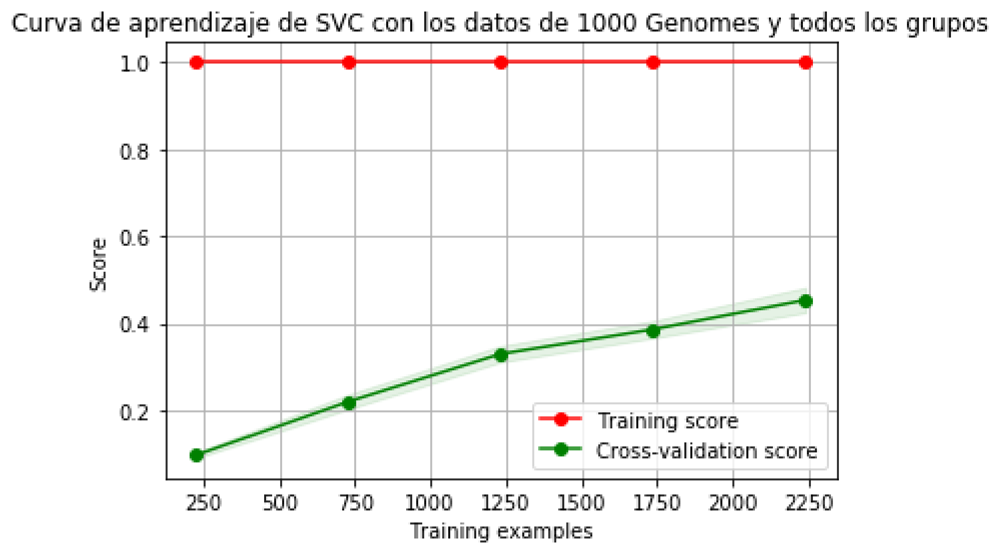


Learning Curve de Random Forest para 1000 Genomes y Supergrupos. El overfitting es el mayor de entre todos los algoritmo para estos datos, rozando apenas lo aceptable.

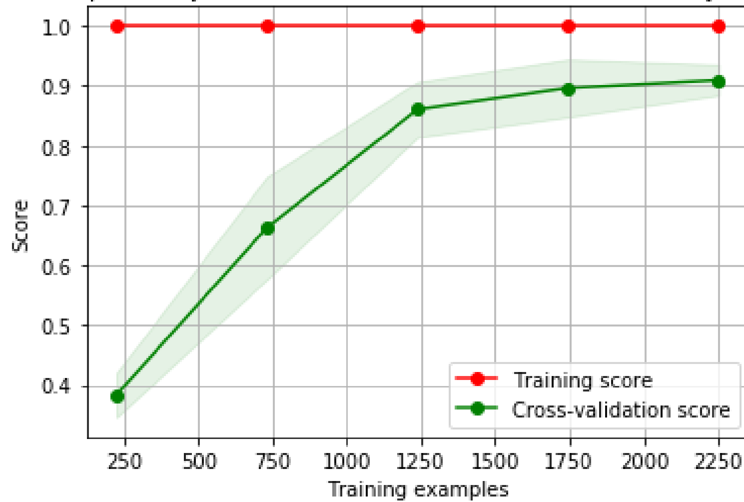


Learning Curve para Random Forest para Hapmap. En este caso los resultados son óptimos, pero solo si tenemos todas las muestras.

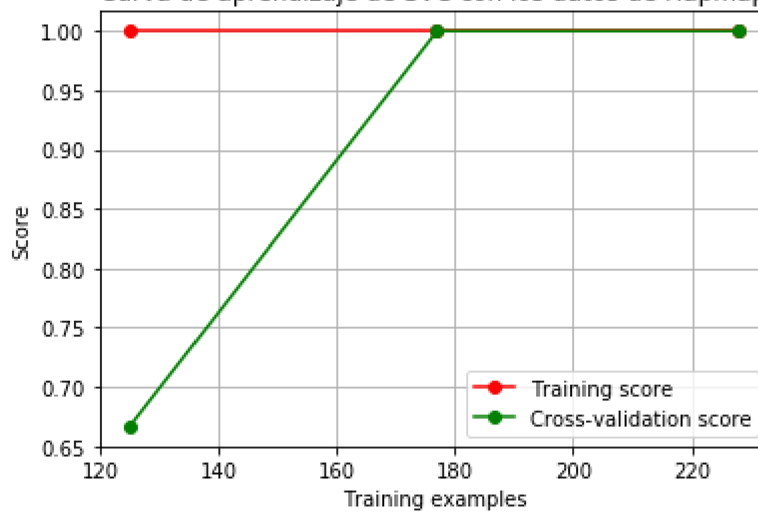
- Support Vector Machine:



Learning Curve de SVC para 1000 Genomes y Grupos. Como en el resto de algoritmos, se aprecia un fuerte overfitting.

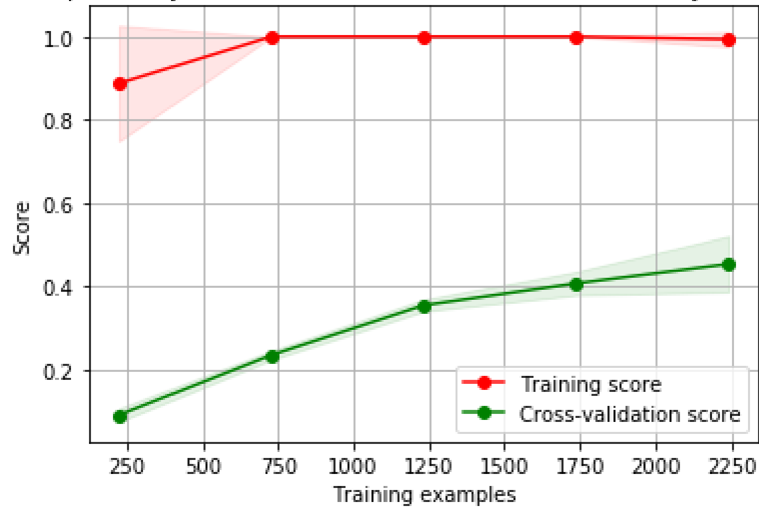


Learning Curve de SVC para 1000 Genomes y Supergrupos. El overfitting vuelve a ser aceptable, superando el 0.9 con muchas muestras.

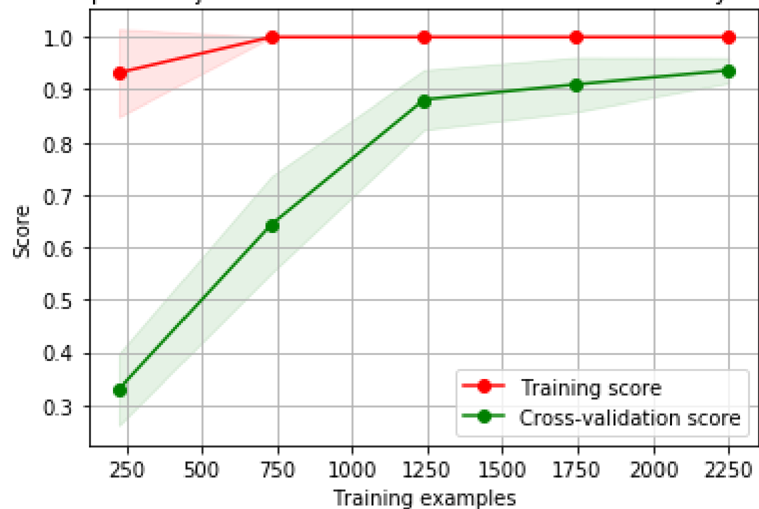


Learning Curve de SVC para Hapmap. Este es el algoritmo que llega a una Accuracy del 100% con menos muestras.

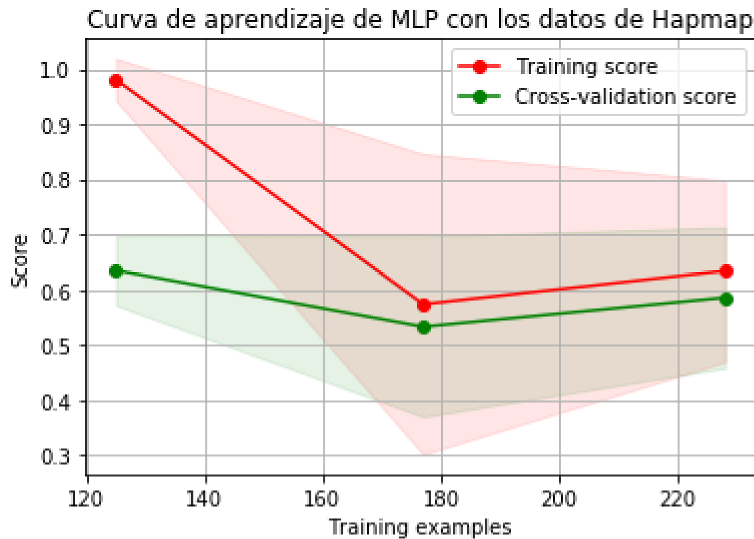
- Artificial Neural Network:



Learning Curve de MLP para 1000 Genomes y Grupos. Seguimos teniendo un fuerte overfitting que explica la gran cantidad de errores en la matriz de confusión.



Learning Curve de MLP para 1000 Genomes y Supergrupos. Este algoritmo muestra la mejor curva para estos datos.



Learning Curve de MLP para Hapmap. Este es el único caso en que apreciamos underfitting. Esto puede deberse a una mala configuración de la Red Neural o al menor número de sujetos con los que entrenar. Este underfitting explica la alta variabilidad de los resultados de este clasificador.

- Interpretación:

  Los resultados de la clasificación vuelven a dar resultados muy diferentes entre ambas bases de datos. Por un lado, salvo la Red Neural, todos los algoritmo presentan una muy buena capacidad de predicción con los datos Hapmap. Por el otro, cuando entrenamos los modelos con los datos de 1000 Genomes con todos los grupos, la capacidad de predicción está muy por debajo de un mínimo aceptable (en torno al 40~50%). Sin embargo, estos mismos datos clasificados con los supergrupos muestra una buena capacidad de predicción en prácticamente todos los algoritmos.

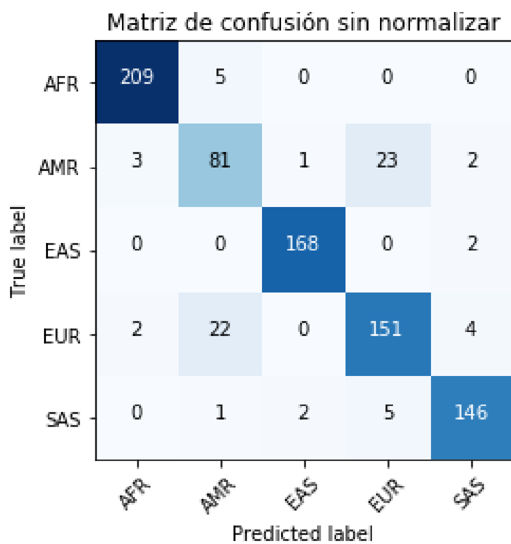
  Estos resultados son curiosos, pues por un lado tenemos unos datos con una Accuracy del 100% (Hapmap), mientras que los otros dan lugar a un clasificador muy malo, con un fuerte overfitting, y otro clasificador sin apenas overfitting y una capacidad de predicción que ronda el 90%.

  Estos resultados ya contradicen una de nuestras hipótesis: los datos de 1000 Genomes, pese a tener más muestras, presentan en todos los casos un rendimiento más bajo que HapMap. ¿A qué puede deberse esto? Aquí es donde se vuelve interesante haber realizado el clustering. En este, hemos comprobado que, mientras que los datos de HapMap son muy discretos, y por lo tanto fáciles de clasificar, los de 1000 Genomes se superponen entre sí.

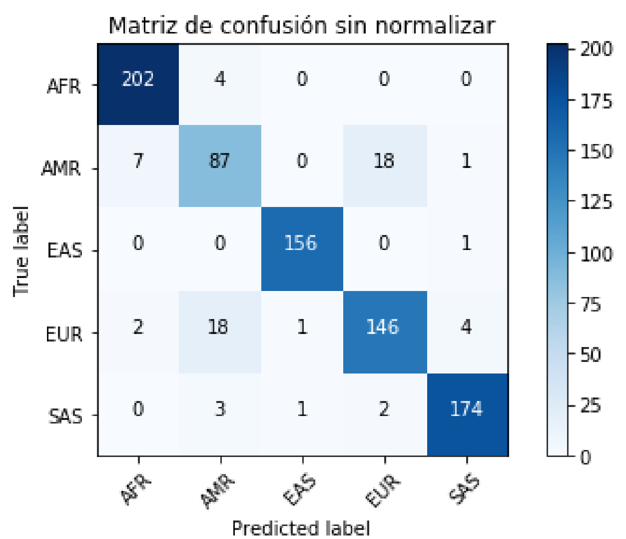
  Esto no significa forzosamente que no podamos entrenar un modelo que se ajuste a nuestros datos, pero sí explicaría por qué el set de datos con menos individuos presenta los mejores resultados.

Pero entonces, ¿a qué se debe la diferencia entre los resultados de 1000 Genomes usando grupos y supergrupos? La explicación más sencilla está en la selección de las muestras. Los grupos dentro de cada supergrupo se seleccionaron por tener relación genética entre ellos (por eso vemos grupos como el de Utah, o CEU, clasificado como europeo o los grupos de africanos de USA o Barbados). Es bastante comprensible que los clasificadores tengan problemas para diferenciar entre tantas clases tan estrechamente relacionadas entre sí. Esto explica que la mayoría de algoritmos no sean capaces de superar el 50% de Accuracy con los grupos, pero ronden el 90% con supergrupos, ya que los supergrupos están mucho más separados genéticamente entre ellos que los grupos que los forman.

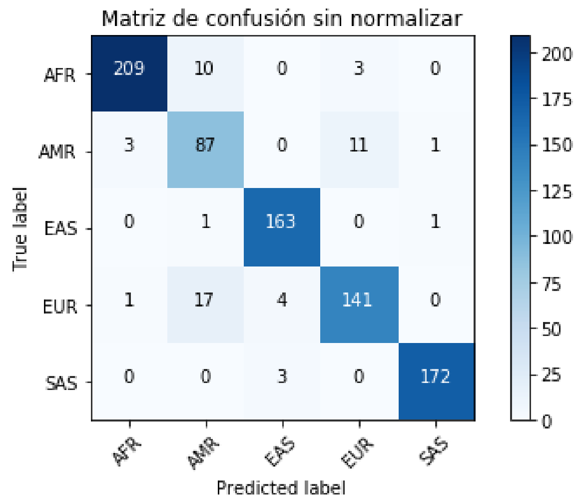
También es interesante el hecho de que en todas las matrices de confusión de 1000 Genomes con supergrupos se aprecia una cantidad sorprendentemente alta de errores relacionados con el supergrupo americano. Este se compone de muestras de diferentes localidades centro y sud-americanas. Recalco esto porque nos encontramos con una gran cantidad de errores de clasificación relacionados con europeos (americanos clasificados como europeos o europeos clasificados como americanos) y, en menor medida, con africanos, lo que podría deberse a cierto porcentaje de "hibridación" entre estas poblaciones.



Matriz de confusión de SVM



Matriz de confusión de Red Neural

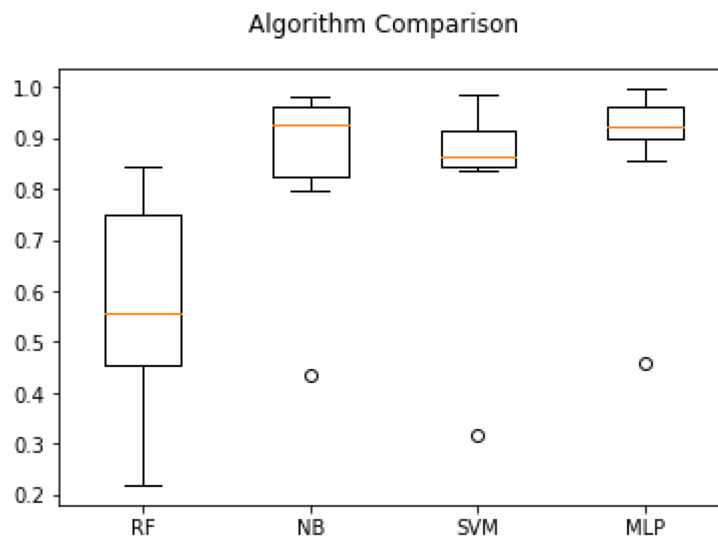


Matriz de confusión de Naive Bayes

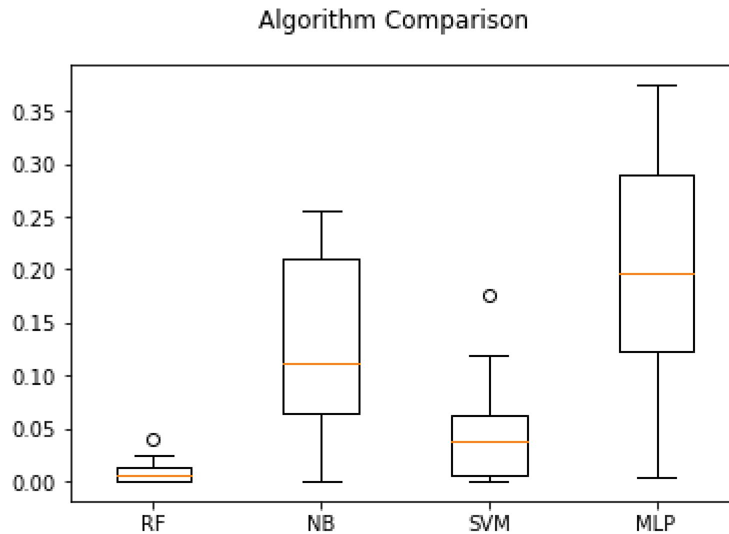
9. Comparativa del rendimiento de los algoritmos:

Para comparar el rendimiento de los diferentes algoritmos es necesario que se cumplan una serie de requisitos, o la comparativa no será válida. Para realizar esta comparativa utilizamos el método de la 10-fold Cross Validation (una Cross Validation en que realizamos 10 particiones aleatorias). Para que estas particiones sean siempre las mismas, las creamos nosotros fijando la 'semilla' del generador aleatorio, de forma que este siempre será igual.

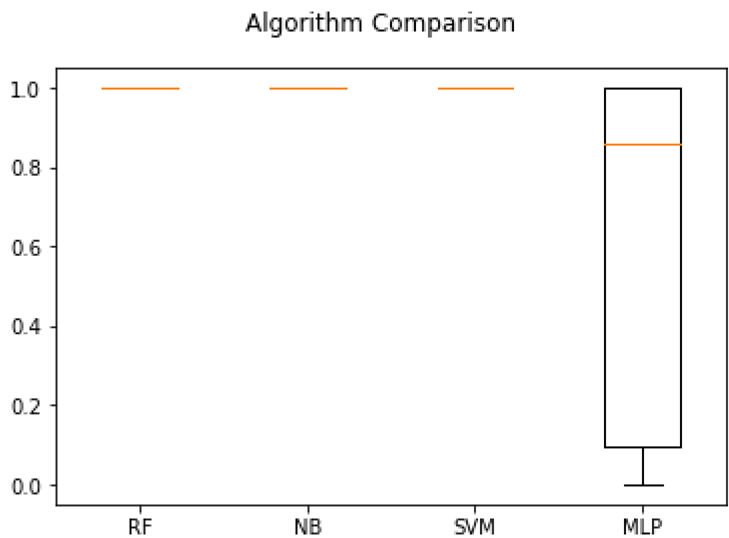
Una vez realizada la Cross Validation, obtenemos estos gráficos de cajas:



Accuracy de los diferentes algoritmos entrenado con 1000 Genomes y Supergrupos



Accuracy de los diferentes algoritmos entrenados con 1000 Genomes y Grupos



Accuracy de los algoritmos entrenados con Hapmap.

Como podemos ver, en el mejor de los casos el rendimiento de 1000 Genomes con grupos locales es del 35~40%, lo que descarta a todos los clasificadores. Sin embargo, tanto MLP (Multi-Layered Perceptron, la Red Neural simple de Scikit Learn) como NB (Naive Bayes) muestran un muy buen rendimiento en el caso de los supergrupos, seguidos por SVM. Random Forest parece tener problemas también en clasificar también estos datos.

Por otro lado, vemos un diagrama de cajas muy extraño en HapMap. En este vemos como todos los algoritmos, salvo MLP, presentan un consenso de clasificación con un 100% de efectividad. Sin embargo,

MLP, dependiendo del set de datos, presenta una variabilidad enorme (prácticamente de 0 a 100%). Esto se deba seguramente a la menor cantidad de muestras, pues un problema recurrente al entrenar MLP fue que tras las 200 iteraciones que hace el algoritmo, no encontraba un modelo óptimo.

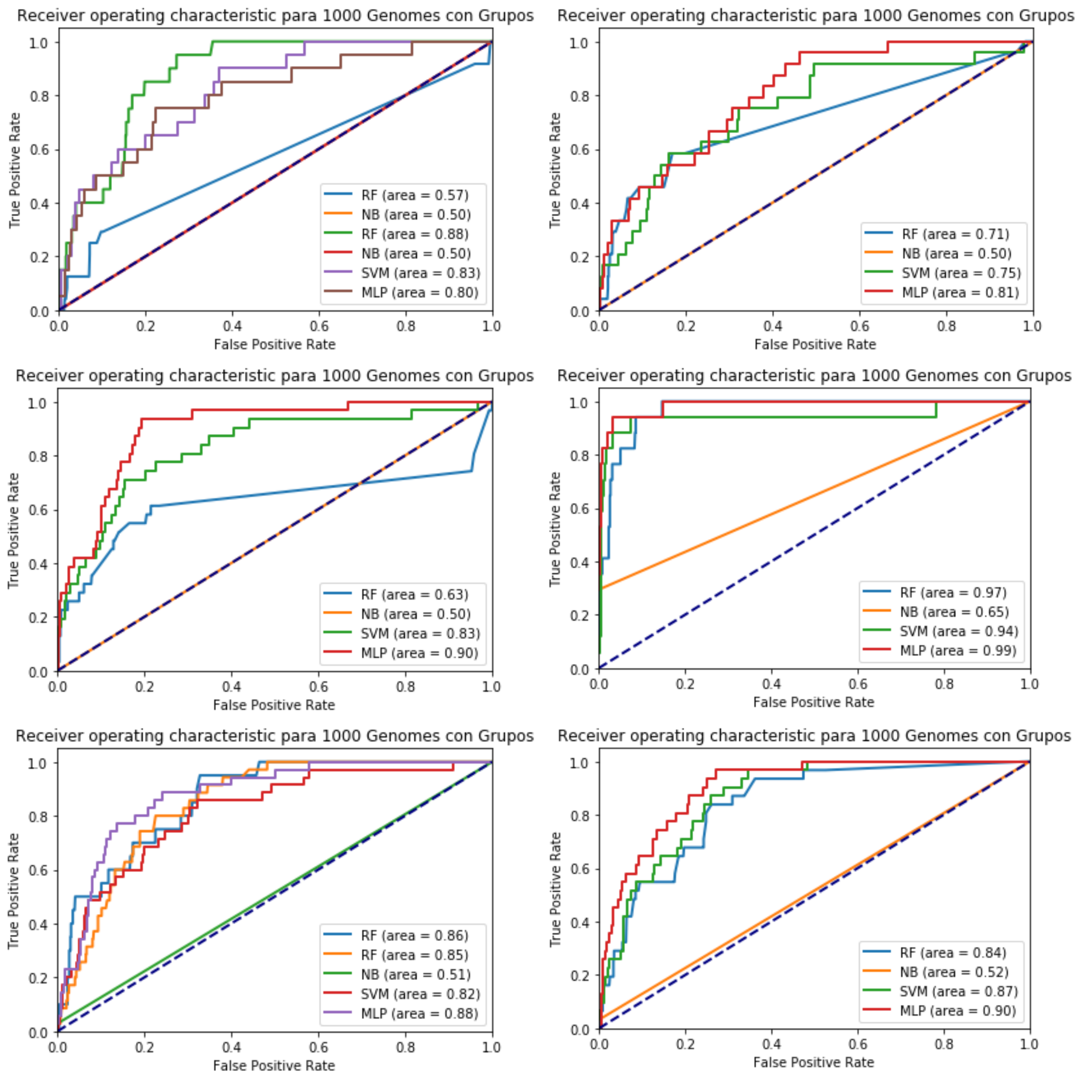
El hecho de que el algoritmo que mejor funciona para uno no lo haga para el otro grupo pone en evidencia la importancia una vez más del diseño del experimento y los criterios a la hora de recoger las muestras.

Para profundizar más en cómo se comportan los algoritmos, utilizamos las Reciever Operating Characteristic Curve (ROC Curve). Este tipo de gráfico, que en sus orígenes se usaba en radiocomunicación, representa la sensibilidad frente a la especificidad en un sistema binario conforme varía la discriminación.

Esto se traduce en que representamos gráficamente la relación entre el Ratio de Verdaderos Positivos y el Ratio de Falsos Positivos para una clase. Para cuantificar esta relación usamos el Área Bajo la Curva (Area Under the Curve, o AUC).

Este tipo de gráfico se suele usar para analizar cómo se comporta un algoritmo para cada una de sus clases o para cada iteración de una Cross Validation. Para comparar diferentes algoritmos, es necesario comparar solamente una de las clases para cada algoritmo. Si un algoritmo presenta una consistencia entre clases, podemos considerar que este es mejor clasificador que otro, pese a que muestre resultados similares en general.

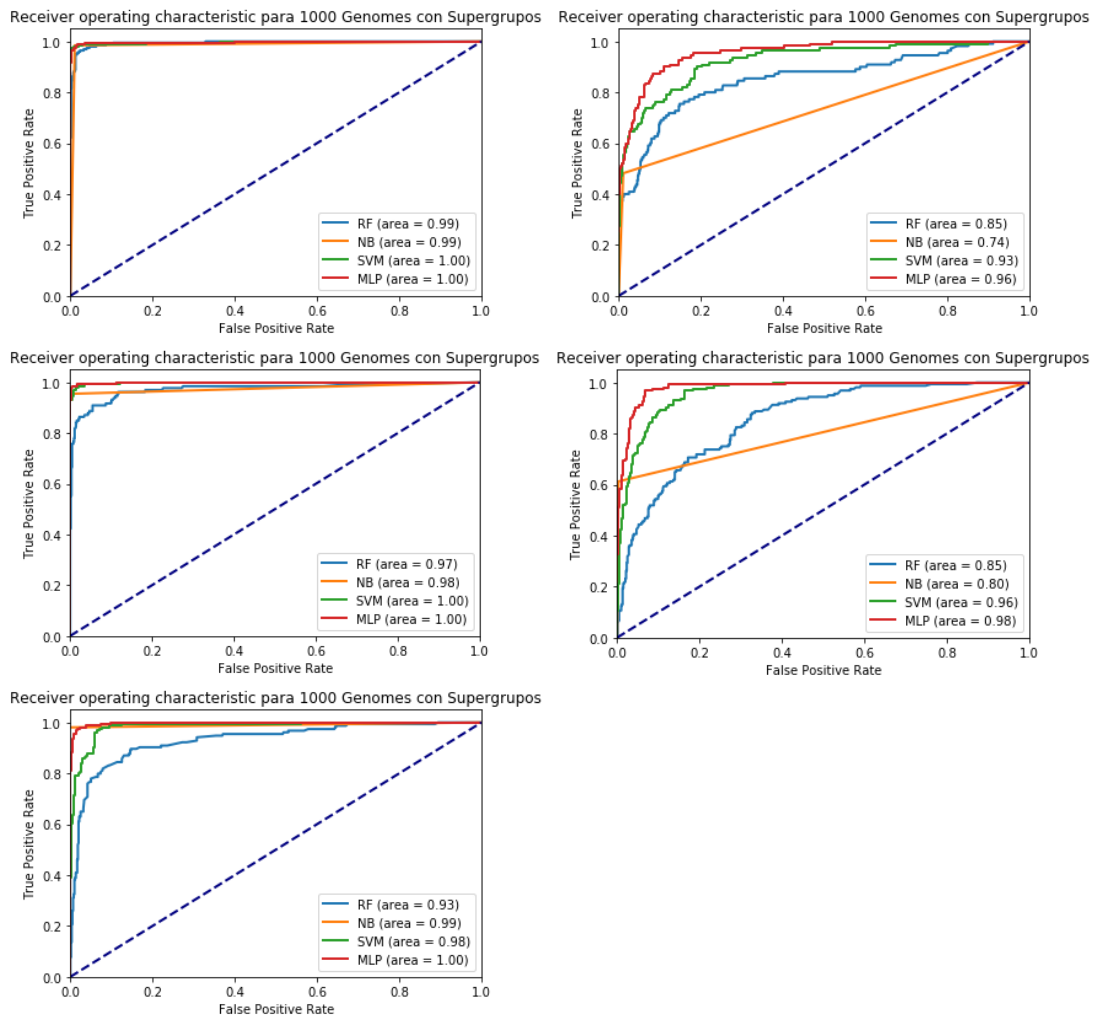
- ROC Curves para 1000 Genome con grupos:



Como podemos ver en los diferentes gráficos, la diferencia en el pomportamiento de los algoritmos para cada clase es muy diferente, siendo Naive Bayes y Random Forest especialmente inconstantes. Vemos, sobretodo en Naive Bayes, varias clases en que la clasificación es totalmente aleatoria, mientras que Random Forest en algunas clases se comporta mejor que el resto, y en otras da resultados peores que los aleatorios, haciendo que el resultado final sea sub-óptimo.

Sin embargo, tanto SVC como MLP muestran unos resultados bastante buenos y regulares en todas las clases.

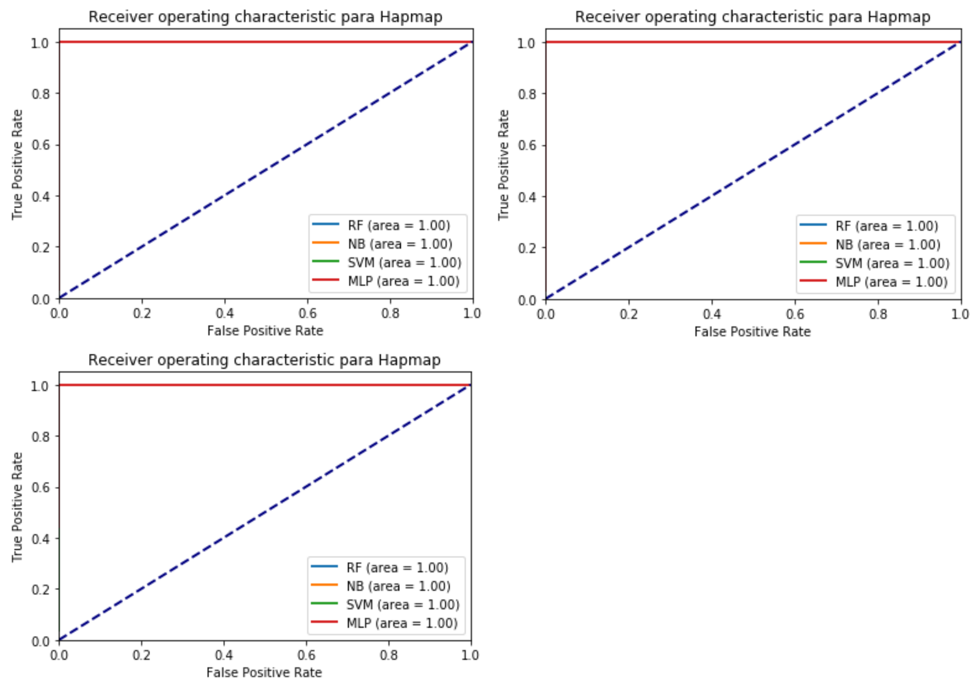
- ROC Curves para 1000 Genomes y Supergrupos:



Como podemos ver en las gráficas de la derecha, con supergrupos los clasificadores, en general, se comportan bastante bien. Sin embargo, Random Forest siempre presenta una AUC ligeramente menor que el resto de algoritmos. En las dos clases de la izquierda, por el contrario, Naive Bayes se comporta de una forma un tanto extraña, lo que puede estar relacionado con la naturaleza «ingenua» del algoritmo. En estas dos clases todos los algoritmos se comportaron peor, lo que refuerza la idea de que ambas están relacionadas.

Como en el caso de 1000 Genomes con Grupos, Multi-Layered Perceptron ha mostrado los mejores resultados en todas las clases, seguido de cerca en AUC y consistencia de Support Vector Classifier.

- ROC Curves para Hapmap:



En este caso el análisis es poco informativo, pues todos los clasificadores muestran unos valores de 1.

10. Conclusiones

- Conclusiones del análisis de los datos:
  - Los datos genómicos, pre-procesados adecuadamente, son perfectamente aptos para el aprendizaje automático.
  - La calidad de los resultados y la fiabilidad de las predicciones dependerá del diseño del proyecto y de la recolección de los datos, por lo que es importante tenerlo en cuenta a la hora de seleccionar qué bases de datos vamos a usar.
  - Al trabajar con genomas, es necesario tener en cuenta la demografía y la historia de las poblaciones de donde provienen nuestros datos para contar con posible mestizaje que dificulte el funcionamiento de los clasificadores.
  - De los tres algoritmos de clustering, es complicado seleccionar uno, pues depende de la distribución y naturaleza de los datos. Lo que más influye es la relación entre las muestras.
  - De los algoritmos de clasificación, la Red Neural (Multi-Layered Perceptron) ha sido la que mejores resultados ha dado para los datos de 1000 Genomes, tanto con Grupos como con Supergrupos. Sin embargo, este algoritmo no muestra tan buenos resultados con Hapmap, donde el resto de clasificadores sí funcionaron perfectamente. Un posible paso futuro sería comprobar si otras configuraciones no tenidas en cuenta para este análisis son capaces de ajustarse a estos datos (MLP con backpropagation, añadir una segunda Hidden Layer, incrementar el ratio de aprendizaje, cambiar la función de activación, etc.). Al ser estos algoritmos tan complejos, muchas veces este tipo de underfitting se debe justamente a un mal diseño. La otra opción, en caso de necesitar un algoritmo genérico que se pueda aplicar a ambas sin preocupars, es usar Support Vector Classifier. La versión de clasificación de Support Vector Machine en scikit-learn ha demostrado se tan constante como MLP, perdiendo solo un poco de su capacidad predictiva con datos complejos, pero manteniendo una performance perfecta en Hapmap. Preferimos este a Naive Bayes ya que es más consistente a la hora de clasificar las distintas clases, pese a que Naive Bayes muestra una Accuracy máxima mejor.
- Reflexión sobre el logro de los objetivos:
  - Los objetivos generales del trabajo se han realizado con éxito, aunque debido a los problemas expuestos anteriormente sobre la variación del rendimiento dependiendo de los datos, el objetivo principal, que era analizar como se comportan los algoritmos de *Machine Learning* con datos genómicos pasa a ser menos extrapolable a otros experimentos, pues cada base de datos requerirá un análisis previo para comprobar qué clasificadores funcionan mejor.

- Análisis del seguimiento de la planificación y la metodología:
    - Con respecto a la planificación, una serie de graves errores de *hardware* y *sofware* llevaron a la pérdida total de la información a mitad del proyecto, por lo que toda la planificación tuvo que ser alterada. En un principio se iba a procesar uno de los cromosomas enteros, extraer las variantes más relevantes mediante *feature selection* y posteriormente realizar los análisis. Por desgracia eso no fue posible por el retraso acumulado. Para compensar este hecho, en la siguiente versión del proyecto, además de realizar una copia de back-up en un disco duro externo, se seleccionaron una serie de variantes relacionadas con una característica biológica que nos ofreciera una cantidad 'comedida' de variantes, pero a la vez suficientes como para poder clasificar.
    - En relación al diseño de un servidor web para mostrar los resultados, debido a la falta de tiempo se recurrió a ejecutar el código mediante las Jupyter Notebooks, una herramienta que nos permite combinar código ejecutable y texto. Es ligeramente más lenta que trabajar en un terminal o un procesador de texto, pero gracias a la facilidad para exportar a html, pudimos obtener unos resultados presentables en formato web sin necesidad de programarlo.
    - En cuanto a la metodología, no hubo cambios importantes entre la primera versión y la segunda, aunque sí cambiaron las herramientas que se emplearon, sobre todo en la sección de pre-procesado. Como en un principio se iba a procesar un cromosoma entero en Python, utilizamos Dask, una herramienta basada en Pandas pero que, de forma similar a PyVCF, no carga todos los datos en la memoria. Esta librería está pensada para trabajar con grandes set de datos y tiene formas de alimentar directamente algoritmos de Machine Learning, por lo que era una buena opción. Al cambiar la estrategia para ahorrar tiempo (el pre-procesado original de 1000 Genomes tardó 9 días de computación ininterrumpida), Dask se volvió innecesario.
- Líneas de trabajo futuro:
    - Debido a los problemas mencionados arriba, que ocasionaron una inesperada falta de tiempo, no pudimos implementar más algoritmos que los más populares, y que fueron los que pusimos en un principio en el planning. Sin embargo, siempre se consideró la opción de aplicar otros, como Stochastic Gradient Descent, o explorar alguna implementación de Deep Learning y otras redes neurales más complejas usando Tensorflow, lo que hubiese permitido utilizar el potencial del GPU, mucho más rápido para procesar este tipo de cálculos.

11. Glosario

- Machine Learning: tecnología informática que, basándose en modelos estadísticos, infiere información de un conjunto de datos.
- Genómica: disciplina de la Biología que se dedica a estudiar cómo se comporta el genoma.
- SNP: Single Nucleotide Polymorphism, es una mutación (o polimorfismo) de un solo nucleótido.
- VCF: Variant Calling Format, formato tabular usado en Bioinformática para almacenar los resultados de un Variant Calling, o proceso de identificación de variantes con respecto a un genoma de referencia.
- CPU: Core Processor Unit, es el procesador central de un ordenador.
- GPU: Graffic Processor Unit, es el procesador de la targeta gráfica.
- CUDA: tecnología que permite transferir ciertos cálculos de la CPU a la GPU. Esto es interesante para le procesamiento de imagen, audio y vídeo, o para trabajar con redes neurales y deep learning, ya que la GPU es más rápida que la CPU realizando este tipo de cálculos.
- Deep Learning: Red Neural de más de una capa oculta.

12. Bibliografía

1. Campello, R. J. G. B.; Moulavi, D.; Zimek, A.; Sander, J. (4 de abril de 2013). «A framework for semi-supervised and unsupervised optimal extraction of clusters from hierarchies». *Data Mining and Knowledge Discovery* (en inglés) 27 (3): 344-371. doi:10.1007/s10618-013-0311-4. ISSN 1384-5810
2. Hartigan, J. A.; Wong, M. A. (1979). «Algorithm AS 136: A K-Means Clustering Algorithm». *Journal of the Royal Statistical Society, Series C (Applied Statistics)* 28 (1): 100–108. JSTOR 2346830.
3. Hofmann, T., Schölkopf, B., & Smola, A. J. (2008). Kernel methods in machine learning. *The annals of statistics*, 1171-1220.
4. Jarník, V. (1930), "O jistém problému minimálním" [About a certain minimal problem], *Práce Moravské Přírodovědecké Společnosti* (in Czech), 6: 57–63.
5. Lloyd, S. P. (1957). «Least square quantization in PCM». *Bell Telephone Laboratories Paper*. Publicado mucho más tarde en la revista: Lloyd., S. P. (1982). «Least squares quantization in PCM». *IEEE Transactions on Information Theory* 28 (2): 129–137. doi:10.1109/TIT.1982.1056489. Consultado el 15 de abril de 2009.
6. Steinhaus, H. (1957). «Sur la division des corps matériels en parties». *Bull. Acad. Polon. Sci.* (en francés) 4 (12): 801–804. MR 0090073. Zbl 0079.16403.
7. Breiman, L. (1996). Bagging predictors. *Machine learning*, *24*(2), 123-140.
8. Breiman, L. Random Forests, Machine Learning, 45(1), 5-32, 2001. (publisher link)
9. Breiman,L., Friedman, J., Olshen, R. and Stone, C. "Classification and Regression Trees", Wadsworth, Belmont, CA, 1984.
10. Campello, R., Moulavi, D., Sander,J. (14 de abril de 2013). Pei, J., Tseng, V. S., Cao, L., Motoda, H., Xu, G., eds. *Density-Based Clustering Based on Hierarchical Density Estimates*. Lecture Notes in Computer Science (en inglés). Springer Berlin Heidelberg. pp. 160-172. doi:10.1007/978-3-642-37456-2_14. ISBN 9783642374555.
11. Campello,R., Moulavi, D. & Sander,J. *Density-Based Clustering Based on Hierarchical Density Estimates* Advances in Knowledge Discovery and Data Mining, Springer, pp 160-172. 2013 DOI: 10.1007/978-3-642-37456-2_14 (publisher link)
12. Domingos, P. & Pazzani, M. (1997). On the optimality of the simple Bayesian classifier under zero-one loss. *Machine Learning*. 29: 103–137. (*publisher link*)
13. Ester, M., Kriegel, H. P., Sander,J. & Xu,X. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining, Portland, OR, AAAI Press, pp. 226-231. 1996 (publisher link)

14. Furey, T. S., Cristianini, N., Duffy, N., Bednarski, D. W., Schummer, M., & Haussler, D. (2000). Support vector machine classification and validation of cancer tissue samples using microarray expression data. *Bioinformatics*, *16*(10), 906-914. (publisher link)
15. Gatys, L., Ecker, A. & Bethge, M. A Neural Algorithm of Artistic Style, 2015. Preprint: *https://arxiv.org/abs/1508.06576v2*
16. Glorot, X. & Bengio, Y. "Understanding the difficulty of training deep feedforward neural networks. International Conference on Artificial Intelligence and Statistics. 2010. (publisher link)
17. H. Zhang (2004). The optimality of Naive Bayes. Proc. FLAIRS.
18. Hartigan, J. Clustering Algorithms, de John Wiley & sons, New York, Sidney, Toronto. 1975. (publisher link)
19. Hinton, G. Connectionist learning procedures. Artificial intelligence 40.1 (1989): 185-234. (publisher link)
20. How HDBSCAN Works: http://nbviewer.jupyter.org/github/lmcinnes/hdbscan/blob/master/notebooks/How%20HDBSCAN%20Works.ipynb fecha: 15/05/2017
21. Hunter, J. Matplotlib: A 2D Graphics Environment, Computing in Science & Engineering, 9, 90-95 (2007), DOI:10.1109/MCSE.2007.55 (publisher link)
22. Kaslow, R. A., Carrington, M., Apple, R., Park, L., Munoz, A., Saah, A. J., ... & Detels, R. (1996). Influence of combinations of human major histocompatibility complex genes on the course of HIV–1 infection. *Nature medicine*, *2*(4), 405-411.
23. Kingma, D. & Ba, J. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014). (publisher link)
24. Kohavi, R, A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection, International Joint Conference on Artificial Intelligence (LICAI), 1995 (publisher link)
25. McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, *5*(4), 115-133.
26. McKenna, A., Hanna, M., Banks, E., Sivachenko, A., Cibulskis, K., Kernytsky, A., Garimella, K., Altshuler, D., Gabriel, S., Daly, M & DePristo, MA. The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data, (2010) *GENOME RESEARCH 20:1297-303* (publisher link)
27. McKinney, E. Data Structures for Statistical Computing in Python, Proceedings of the 9th Python in Science Conference, 51-56 (2010) (publisher link)
28. Millman, J. and Aivazis, M. Python for Scientists and Engineers, Computing in Science & Engineering, 13, 9-12 (2011), DOI:10.1109/MCSE.2011.36 (publisher link)

29. Oliphant, T. Python for Scientific Computing, Computing in Science & Engineering, 9, 10-20 (2007), DOI:10.1109/MCSE.2007.58 (publisher link)

30. Paterson, S., Wilson, K., & Pemberton, J. M. (1998). Major histocompatibility complex variation associated with juvenile survival and parasite resistance in a large unmanaged ungulate population (Ovis aries L.). *Proceedings of the National Academy of Sciences*, *95*(7), 3714-3719.

31. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M. & Duchesnay, E. Scikit-learn: Machine Learning in Python, Journal of Machine Learning Research, 12, 2825-2830 (2011) (publisher link)

32. Pérez, F. & Granger, B. IPython: A System for Interactive Scientific Computing, Computing in Science & Engineering, 9, 21-29 (2007), DOI:10.1109/MCSE.2007.53 (publisher link)

33. PyVCF Documentation: http://pyvcf.readthedocs.io/en/latest/FILTERS.html#module-vcf.utils , fecha: 10/04/2017.

34. Raschka, S. Python Machine Learning, PACKT Publishing, Birminham, 2015.

35. Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, *65*(6), 386.

36. Sibson, R.,Comput J (1973) 16 (1): 30- 34. DOI:https://doi.org/10.1093/comjnl/16.1.30

37. Slonim, N, Aharoni, E & Crammer, K. Hartigan's K-Means Versus Lloyd's K-Means – Is it Time for a Change? Twenty-Third International Joint Conference on Artificial Intelligence, 2013 (publisher link)

38. Suykens, J. A., & Vandewalle, J. (1999). Least squares support vector machine classifiers. *Neural processing letters*, *9*(3), 293-300. (publisher link)

39. van der Walt, S., Colbert, C. and Varoquaux, G. The NumPy Array: A Structure for Efficient Numerical Computation, Computing in Science & Engineering, 13, 22-30 (2011), DOI:10.1109/MCSE.2011.37 (publisher link)

13. Anexo:

Junto a este documentos se adjuntan varios archivos HTML. El principal es Portada.html, y enlaza con el resto. Es importante que todos los archivos estén en el mismo directorio para que los enlaces funcionen.

13.1 Código con que se pre-procesaron los datos:

13.1.1: Hapmap para cada archivo

```
# -*- coding: utf-8 -*-
"""
Spyder Editor

This is a temporary script file.
"""

import pandas as pd
import numpy as np

#Leemos los archivos
ceu = pd.read_csv('/home/hjorvik/Escritorio/TFM/Datos/Hapmap/genotypes_chr6_CEU_r24_QC+.b36_fwd.txt', sep = ' ')
asia = pd.read_csv('/home/hjorvik/Escritorio/TFM/Datos/Hapmap/genotypes_chr6_JPT+CHB_r24_QC+.b36_fwd.txt', sep = ' ')
yri = pd.read_csv('/home/hjorvik/Escritorio/TFM/Datos/Hapmap/genotypes_chr6_YRI_r24_QC+.b36_fwd.txt', sep = ' ')
HCL = pd.read_csv('/home/hjorvik/Escritorio/TFM/Datos/HCL_trim.txt', sep = '\t')

HCL = HCL.iloc[:,0]
print(len(HCL))


ceu_t = ceu[ceu.index.isin(HCL.index)]
asia_t = asia[asia.index.isin(HCL.index)]
yri_t = yri[yri.index.isin(HCL.index)]
```

```
#Transponemos, asignamos las columnas por genes y eliminamos los datos que no interesan
ceu_t = ceu_t.transpose()
asia_t = asia_t.transpose()
yri_t = yri_t.transpose()
print("Transpose done")

ceu_t.columns = ceu_t.loc['rs#',:]
asia_t.columns = asia_t.loc['rs#',:]
yri_t.columns = yri_t.loc['rs#',:]

ceu_t.drop(['rs#','chrom','strand','genome_build','center','protLSID','panelLSID','assayLSID','QC_code'],axis = 0, inplace = True)
asia_t.drop(['rs#','chrom','strand','genome_build','center','protLSID','panelLSID','assayLSID','QC_code'],axis = 0, inplace = True)
yri_t.drop(['rs#','chrom','strand','genome_build','center','protLSID','panelLSID','assayLSID','QC_code'],axis = 0, inplace = True)
print("Transformaciones hechas")

#Buscamos los valores vacíos y los sustituimos por la moda de su gen
ceu_t.replace('NN', np.NaN, inplace=True)
asia_t.replace('NN', np.NaN, inplace=True)
yri_t.replace('NN', np.NaN, inplace=True)



#ceu_m = ceu_t.fillna(ceu.mode())
#asia_m = asia.fillna(asia.mode().iloc[0])
#yri_m = yri.fillna(yri.mode().iloc[0])

#Codificamos el genotipo, guardamos y reseteamos los Dataframes para no saturar la memoria

def cod(df):
    count = 0
    df_c = pd.DataFrame()
    for i in range(len(df.columns)):
        genotype = df.iloc[0, i]
```

```
        wild = genotype[0]
        mutant = genotype[2]
        temp = pd.Series(name = df.columns[i])
        for j in range(len(df.index[2:])):
            if df.iloc[j+2,i] == wild+wild:
                temp.loc[df.index[j+2]]=0
            elif df.iloc[j+2,i] == mutant+mutant:
                temp.loc[df.index[j+2]] = 2
            elif df.iloc[j+2,i] == mutant+wild or df.iloc[j+2,i] == wild+mutant:
                temp.loc[df.index[j+2]]=1
            else:
                m = df.iloc[:,i].mode().iloc[0]
                if m == wild+wild:
                    temp.loc[df.index[j+2]] = 0
                elif m == mutant+mutant:
                    temp.loc[df.index[j+2]] = 2
                elif m == mutant+wild or m == wild+mutant:
                    temp.loc[df.index[j+2]]=1
        count += 1
        print(count,' of ', len(df.columns))
        df_c[df.columns[i]]=temp
    return df_c

ceu_c = cod(ceu_t)
ceu_c.to_csv('/home/hjorvik/Escritorio/TFM/Resultados/CEU_c.txt')



asia_c = cod(asia_t)
asia_c.to_csv('/home/hjorvik/Escritorio/TFM/Resultados/JPT+CHB_c.txt')

yri_c = cod(yri_t)
yri_c.to_csv('/home/hjorvik/Escritorio/TFM/Resultados/YRI_c.txt')
```

13.1.2: Función para juntar los archivos

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Apr 20 14:37:06 2017

@author: pedro
"""
import pandas as pd



yri = pd.read_csv('/home/hjorvik/Escritorio/TFM/Resultados/YRI_c.txt', sep = ',',index_col=0)
ceu = pd.read_csv('/home/hjorvik/Escritorio/TFM/Resultados/CEU_c.txt', sep = ',',index_col=0)
asia = pd.read_csv('/home/hjorvik/Escritorio/TFM/Resultados/JPT+CHB_c.txt', sep = ',',index_col=0)

#hap = pd.merge(yri, ceu, how = 'outer')
#hapmap = pd.merge(hap, asia, how = 'inner')
hapmap = pd.concat([yri,ceu,asia])
#print(hapmap.columns,len(hapmap.index))
#print(hap.iloc[0:3,0:3])
#print(ceu.iloc[0:3,0:3])
hapmap.to_csv('/home/hjorvik/Escritorio/TFM/Resultados/chrm6.txt')
```

13.1.3: Pre-procesado de los datos de 1000 Genomes

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue May  2 07:15:02 2017

@author: hjorvik
"""
import pandas as pd
import vcf
```

#Importamos ambos archivos:

```
kg = vcf.Reader(open('/home/hjorvik/Escritorio/TFM/Datos/1KG/chrm6_trimed.vcf','r'))
```

#Seleccionamos solamente los nombres de dbSNP

```
count = 0
kg_c = pd.DataFrame()
for record in kg:
    temp = pd.Series()
    for sample in record.samples:
        temp[sample.sample] = sample.gt_type

    kg_c[record.ID] = temp
    count +=1
    print(count)

          kg_c.to_csv('/home/hjorvik/Escritorio/TFM/Resultados/1kg.txt')
```