# COMPARATIVA Y ANÁLISIS DE ALGORITMOS DE APRENDIZAJE AUTOMÁTICO PARA LA PREDICCIÓN DEL TIPO PREDOMINANTE DE CUBIERTA ARBÓREA

JUAN ZAMORANO RUIZ

MÁSTER EN INGENIERÍA EN INFORMÁTICA,

FACULTAD DE INFORMÁTICA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

UNIVERSIDAD COMPLUTENSE DE MADRID


Trabajo Fin Máster en Ingeniería de Computadores

Madrid, 5 de Julio 2018

Director : Enrique Martín Martín

Convocatoria de junio
Calificación : 9,5

# Autorización de Difusión

JUAN ZAMORANO RUIZ

Junio 2018

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: “Comparativa y análisis de algoritmos de aprendizaje automático para la predicción del tipo predominante de cubierta arbórea”, realizado durante el curso académico 2017-2018 bajo la dirección de Enrique Martín Martín en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

# Resumen

Podríamos decir que vivimos en el día de hoy rodeados de la generación de gran cantidad de datos. Los humanos no somos capaces de procesar por nosotros mismos muchos de ellos y es por ello que nos vemos en la necesidad de buscar herramientas que nos permitan almacenar, procesar y extraer la información que, tratada y evaluada según nuestras necesidades, nos de la capacidad de obtener el beneficio necesario para un objetivo concreto.

Para el tratamiento de esta información podemos enfocarnos en la rama de la Inteligencia Artificial denominada "*Machine Learning*" o aprendizaje automático que permite a las máquinas aprender y tomar decisiones con futuros datos proporcionados de forma automática.

Podríamos decir que el aprendizaje se realiza gracias a la detección de patrones dentro de un conjunto de datos de forma que el mismo algoritmo es capaz de predecir qué tipo de situaciones podrían darse. Lo que pretendemos es dar uso al histórico de datos para que, organizados adecuadamente y tratados en bloque, genere una base de datos que puede ser usada para predecir futuros comportamientos, o, por ejemplo, podamos clasificar un nuevo conjunto de datos no observados previamente. Un ejemplo de este tipo de problema podemos encontrarlo en la técnica de reconocimiento facial.

Para este proyecto se hace uso de la biblioteca Scikit-learn para machine learning en el lenguaje de programación Python. Ésta posee gran cantidad de algoritmos para clasificación y que se comportan de distinta forma dependiendo de la cantidad y distribución de los datos proporcionados. De estos algoritmos se pueden extraer métricas y uso de recursos de la máquina, que pueden ser usadas para poder llevar a cabo una comparación y análisis.

Se realizará una comparación y análisis del comportamiento de cada algoritmo en la predicción de 7 tipos de cubiertas forestales con el uso de variables cartográficas a través de valores tomados en 4 áreas distintas del parque nacional de Roosevelt en el norte del estado de Colorado.

# Palabras clave

Scikit-learn, Aprendizaje automático, Big Data, Python, Inteligencia Artificial, Comparativa, Análisis.

# Abstract

We could say that we live today surrounded by the generation of large amount of data. The majority of that data is too much huge. We need to give to humans the proper tools where to store, process and extract the information that, treated and made available to us could give the right information where we can extract a benefit for a specific goal.

For the treatment of this information we can focus on the branch of Artificial Intelligence called machine learning which allows machines to learn and make decisions automatically with future data provided.

We could say that learning is done thanks to the detection of patterns within a set of data where the algorithm is able to predict what kind of situations could occur. What we intend to do is to use the historical data which, properly sorted and treated in blocks, it generates a database that can be used to predict future behaviors, or, for example, we can classify a new data set not previously observed. An example of this could be the facial recognition technique.

For this project, we will use the library for Python programming language scikit-learn, which is used for machine learning. It has a large number of algorithms for classification where we will find different behaviour depending on the quantity and distribution of the data provided. From these algorithms we can extract some metrics, such as the use of resources from the machine, which can be used to carry out a comparison and analysis.

A comparison and analysis of the behavior of each algorithm will be made in the prediction of 7 forest cover types with the use of cartographic variables through values taken in 4 different areas of the Roosevelt National Park in the north of the state of Colorado.

# Keywords

Scikit-learn, Machine learning, Big Data, Python, Artificial intelligence, Comparative, Review.

# Índice

# Agradecimientos

“El mejor profeta del futuro es el pasado”
Lord Byron

Gracias a todos aquellos que confiaron en mi desde que comencé este Máster de Ingeniería Informática. A pesar de los obstáculos que uno pueda encontrarse en el camino siempre es adecuado pensar que todo esfuerzo tiene su recompensa. Esa recompensa la veo hoy plasmada en este trabajo que refleja los años de trabajo y dedicación para completarlo.

A mi tutor, Enrique Martín Martín, por despertar en mí la curiosidad por los datos en su asignatura y por apoyarme en todo momento para conseguir llevar a cabo este trabajo.

# Introducción

Como idea principal, podemos decir que el aprendizaje automático (*Machine Learning*) es la ciencia de los algoritmos que se encarga de darle sentido a los datos. Este concepto es adecuado indicarlo pues cada vez estamos rodeados de más información de la que creemos, algo que si sabemos procesarlo y hacemos un adecuado uso de los algoritmos de auto-aprendizaje nos puede ayudar a identificar patrones y con ello clasificar o predecir partiendo de nuevos conjuntos de datos que no han sido tratados previamente.

Esta técnica se basa en el uso datos estadísticos, probabilísticos y de técnicas de optimización para dar la capacidad de aprender a las máquinas por sí solas. Existen gran cantidad de algoritmos de aprendizaje automático que son usados para llevar a cabo tareas de clasificación o de diagnóstico como puede ser la predicción del cáncer de próstata o de mama [13], o por ejemplo el uso de una de estas técnicas para la clasificación del nivel de cáncer de piel de un paciente [16] .

Para poder llevar a cabo lo anteriormente mencionado es necesario: proveer los datos, almacenarlos y, finalmente, procesarlos para obtener los resultados que buscamos.

Para proveer los datos es necesario encontrar una fuente fiable que proporcione una cantidad y calidad de información suficiente como para nutrir nuestro sistema de los datos que le permita llevar a cabo el procesamiento adecuado y así extraer información fiable.

Una fuente interesante, y que hemos contemplado como la más adecuada, para encontrar conjuntos de datos que son de libre uso y a los cuales se puede aplicar aprendizaje automático en diversos campos es en la página web www.kaggle.com [1]. La página es una plataforma considerada como el “hogar” de la ciencia de los datos y del aprendizaje automático, en la que podemos extraer grandes colecciones de datos públicas y de alta calidad. Además es un lugar donde los “científicos de datos” pueden compartir información y los distintos algoritmos que han desarrollado para ponerlo en conocimiento de todo el que sea miembro de la plataforma y pueda colaborar para mejorar. En ella se plantean numerosos problemas a los que se le puede aplicar la ciencia del aprendizaje automático ya sea para clasificación o predicción de futuros datos, dependiendo de la naturaleza del problema. Concretamente el conjunto de datos que vamos a tratar está orientado a la clasificación de la cubierta arbórea predominante en 4 áreas del bosque Roosevelt al norte del estado de Colorado. Se trata de cerca de 600.000 instancias y que puede ser descargado desde https://archive.ics.uci.edu/ml/datasets/covertype. [68]

Para el almacenamiento y procesamiento de los datos usaremos la biblioteca pandas [71], orientada a Python, que permite trabajar de forma flexible con estructuras de datos relacionales y etiquetadas. Para el procesamiento de estos datos vamos desarrollar un programa en Python y que está disponible para su descarga y uso en el siguiente repositorio

público: https://bitbucket.org/juzaru18/trees/. En él haremos uso de diferentes bibliotecas para el procesamiento de los datos y la extracción de métricas y así llevar a cabo las comparativas y análisis. Una de las bibliotecas que utilizaremos es la de libre uso scikit-learn orientada al lenguaje de programación Python y que a partir de ahora nombraremos como sklearn. En ella disponemos de numerosos algoritmos para clasificación que han sido implementados previamente y que nos permiten alimentar y extraer la información que necesitemos con el uso correcto del flujo de datos, es decir, saber qué información debemos proveer y cual podemos extraer tras el procesamiento.

Si bien es bueno indicar que cada algoritmo escogido mostrará un comportamiento diferente, ya sea por la técnica utilizada para procesar y clasificar, como por la distribución y naturaleza de los datos suministrados. Así de cada algoritmo podremos extraer valores como la precisión que ha tenido con un conjunto de datos de entrenamiento o el consumo de recursos de la máquina en la que se ejecute.

Lo que buscamos así para la comparativa a llevar a cabo en este trabajo es, por un lado entender de forma más precisa cómo funciona cada clasificador y, por otra parte, definiremos unos criterios que nos ayudarán a tomar la decisión de cuál es el mejor para el tratamiento del problema planteado.

Los criterios que podemos tomar para nuestro análisis y comparativa pueden ser:

- Recursos del sistema
    - Memoria usada para ajuste o entrenamiento
    - Tiempo necesario para el ajuste
    - Tiempo necesario para la predicción de las etiquetas.
- Valores del propio algoritmo
    - Exactitud, sensibilidad, precision entre otros para clasificar un conjunto de test de datos sin etiqueta y otros valores que se pueden medir y que explicaremos posteriormente.

De los datos previos podemos generar tablas donde almacenaremos los valores que cada clasificador ha generado para así dibujar gráficas comparativas e indicar cuál sería la mejor opción a elegir para tratar el problema en el que estamos implicados.

Podemos definir que queremos obtener una exactitud cercana al 100% además de un error de predicción bajo, valores que nos indicarán una confianza mayor en dicho clasificador para aquellos datos suministrados que no hayan sido observados previamente. Estas métricas las explicaremos en los siguientes capítulos para su mejor comprensión.

# Plan de trabajo

El trabajo a realizar en este estudio se va a dividir en varias fases, las cuales detallaremos incluyenda fecha con una visualización mejor de los tiempos empleados:

1. Septiembre 2017: Búsqueda de un conjunto de datos con un tamaño y características adecuadas para poder se considerada como viable a tratar en una máquina de uso cotidiano, como puede ser un portátil, y nos permita ejecutar los algoritmos para extraer la información de los mismos y su posterior comparativa.
2. Octubre 2017 - Enero 2018: Desarrollo de un programa para la recolección, procesamiento, almacenamiento y visualización de los datos procesados. Todo esto incluye dos fases, las cuales son:
    a. Octubre - Noviembre 2017: Estudio del uso de bibliotecas para desarrollar lo anteriormente mencionado.
    b. Diciembre 2017- Enero 2018: Desarrollo de programa y pruebas de recolección de datos. Definición de métodos y estructuras de datos para el programa. De esta forma conseguiremos una comprensión y utilización sencilla del mismo programa.
3. Febrero - Marzo 2018 : Análisis de los datos extraídos en el programa para llevar a cabo una comparativa entre los distintos clasificadores disponibles y cómo ha sido su comportamiento a la hora de tratar los datos proporcionados. Algunos de estos datos como el uso de memoria no se podrán extraer internamente de las librerías disponibles, por lo que se hará uso de ejecuciones o bibliotecas externas, que se explicarán en posteriores capítulos. La medición y extracción de datos relevantes se realiza en varias fases:
    a. Primera quincena de Febrero 2018: Análisis y comparación distintas métricas para la obtención de datos relevantes para la comparación de algoritmos, ejecución del programa para las distintas técnicas estudiadas y anotación de datos a mano como el uso de memoria.
    b. Segunda quincena Febrero 2018: Uso de bibliotecas en el lenguaje de programación usado para la visualización adecuada de las métricas obtenidas.
4. Marzo- Mayo 2018: Redacción de memoria y extracción de conclusiones indicando, a través de los valores obtenidos y con el uso de gráficas y tablas, cúal es el mejor clasificador a usar para el problema que estamos tratando.

# Introduction

As a main idea, we can say that machine learning is the science of algorithms that is responsible for giving meaning to the data. This concept is appropriate due to this topic is closer than we think. It´s something that if we use in the right way of self-learning algorithms, we can help identify patterns. With that information, we can classify or predict starting from new datasets that have not been previously trained.

This technique is based on the use of statistical, probabilistic and optimization techniques that allows machines ability to learn. There are many automatic learning algorithms that are used to carry out classification or diagnostic tasks such as the prediction of prostate or breast cancer [13], or for example the use of one of these techniques for classification of a patient's skin cancer level [16].

In order to carry out the mentioned above, we need to: provide the data, store it and, finally, process it to obtain the results we are looking for. To provide the data it is necessary to find a reliable source that provides a sufficient amount and quality of information to feed our data system that allows it to carry out the appropriate processing and thus extract reliable information.

An interesting source, and which we have considered as the most appropriate, to find data sets that are free to use and to which automatic learning can be applied in various fields is on the website https://www.kaggle.com [1]. This page is a platform considered as the "home" of data science and machine learning, where we can extract large collections of public and high-quality data. It is also a place where the "data scientists" can share information and the different algorithms they have developed to make it known to everyone who is a member of the platform and can collaborate to improve it. There we can find a variety of challenges where we can apply the science of machine learning either for classification or prediction of future data depending on the nature of the problem. Specifically, the data set that we are going to discuss is oriented towards the classification of the predominant tree cover in 4 areas of the Roosevelt forest in the north of the state of Colorado. It is about 600,000 instances and it can be downloaded from [65]

For data processing we will develop a script in Python which it is available for download and use in the following public repository: https://bitbucket.org/juzaru18/trees/. In it we will use different libraries for the processing of data and the extraction of metrics to carry out the comparison and analysis. One of the libraries that we will use is the open source scikit-learn oriented to the developing language Python and which from now on we will name it as sklearn. In it we have numerous classification algorithms that have been previously implemented and

that allow us to feed and extract the information we need with the correct use of the data flow, that is, to know what information we must provide and what we can extract after processing.

It is good to know that each algorithm chosen will show a different behavior, either by the technique used to process and classify as per the distribution and nature of the data provided. Thus, we can extract values from each algorithm such as the precision it has had with a set of training data or the consumption of resources of the machine in which it is executed.

What we are looking for in the comparison to be carried out in this work is, on the one hand, to understand more precisely how each classifier works and, on the other hand, we will define some criteria that will help us make the decision of which is the best for the treatment of the problem suggested.

The criteria that we can take for our analysis and comparison will be:

- System resources
    - Memory used.
    - Time to process the data
- Values extracted from classifier itself:
    - Accuracy, recall score, precision to classify a set of tests and other values that can be measured and that we will explain later.

From the previous data we can generate tables where we will store the values that each classifier has generated. Then we can draw comparative graphs and indicate which would be the best option to choose to deal with the problem in which we are involved.

We can define that we want to obtain for example a low memory usage, an accuracy close to 100% and an error deviation close to 0 when it predicts the probabilities for each class on new unseen dataset, values that will indicate a greater confidence on that classifier.

## Project schedule

The work to be done in this study will be divided into several phases, which we will detail including dates with a better visualization of the times employed:

1. September 2017: Research for a dataset with a size and appropriate features to be considered viable to be treated in a laptop machine and allow us to fit the algorithms in order to extract information from them and their subsequent comparison.
2. October 2017- January 2018: Development of a script for the collection, processing, storage and visualization of data stored. All this includes several phases, which are:
    a. October 2017 - November 2017: Study for the libraries to be used to develop the script mentioned above.
    b. December 2018 - January 2018: script development and data collection tests. Definition of methods and data structures for the script.

3. February - March 2018: Analysis of the data extracted in the script to carry out a comparison between the different available classifiers and how their behavior has been by the time we will deal with data provided. Some of these data, such as memory usage, can´t be extracted internally from the available libraries, so external executions will be used, which will be explained in later chapters. The measurement and extraction of relevant data is done in several phases
    a. First half of February 2018: Analysis and comparison of different metrics to obtain relevant data for the comparison of algorithms, data such as the use of memory.
    b. Second half of February 2018: Use of libraries for Python development language used for the proper visualization of the obtained metrics.
4. March 2018 - May 2018: Memory redaction and extraction of conclusions indicating, through the values obtained and with the use of graphs and tables, which is the best classifier to use for the problem we are dealing with.

# Capítulo 1. Preliminares

## 1.1. Aprendizaje automático en la ciencia de los datos

Hoy en día podemos indicar que tenemos una gran cantidad de datos, tanto estructurados como sin estructurar. Así el aprendizaje automático ha ido evolucionando en los últimos años en la rama que conocemos como inteligencia artificial de forma que podemos tener máquinas capaces de predecir gran cantidad de información basándose en la que ya tiene almacenada y procesada previamente. Así tenemos muchas aplicaciones de las que podemos sacar partido del aprendizaje automático, entre ellas está la de clasificar, como puede ser por ejemplo detectar si un correo es considerado como spam o no. En 10 años el aprendizaje automático ha conquistado la industria del día a día ayudando a los humanos a aprender de los mismos, como por ejemplo posicionamiento de una web en un motor de búsqueda, reconocimiento de voz, reconocimiento de imágenes, o algo tan sonado hoy en día como son los coches autónomos.

### 1.1.1. Tipos de aprendizaje automático

Dependiendo del tipo de aprendizaje podemos dividirlo en 3 tipos:

- Aprendizaje con clase o supervisado
- Aprendizaje sin clase o no supervisado
- Aprendizaje reforzado

El aprendizaje con clase [26,33] también conocido como aprendizaje supervisado, es aquel donde el objetivo es aprender un patrón a partir de un conjunto de datos que utilizaremos de entrenamiento, y que nos permitirá realizar predicciones de conjuntos de datos no observados previamente. De esta forma, podemos indicar que el concepto de "supervisado" proviene de que a partir de datos usados de ejemplo, que usaremos como entrada para entrenar nuestro modelo, sabemos cuál es la salida deseada.

Para el aprendizaje sin clase [14,24] también encontrado o conocido como aprendizaje no supervisado, tenemos que manejar datos sin etiquetar o datos sin estructura de los cuales no tenemos información previa que nos ayude a identificar patrones o saber a qué clase pertenecen los datos. Así este tipo de aprendizaje trata de crear distintos subgrupos o subconjuntos a partir de una exploración del conjunto de datos indicado, basándose por ejemplo en la similitud de las características.

Un ejemplo de aplicaciones usando este tipo de aprendizaje tenemos la agrupación de genes y proteínas con similar funcionalidad o la agrupación de documentos para poder explorarlos de una forma más rápida, como por ejemplo Google Images.

Por último tenemos el aprendizaje reforzado [27,65]. El objetivo de este tipo de aprendizaje es conseguir un sistema que mejore su aprendizaje a medida que el entorno que lo rodea va cambiando o, dicho de otra forma, es extraer qué acciones deben ser elegidas en los diferentes estados para maximizar la recompensa. Así si la clasificación se realizó correctamente entonces ganará confianza dicha clasificación a través de una función de recompensa. De esta forma se pueden usar técnicas de exploración de un conjunto de estados y a través de prueba-error producir la salida adecuada habiendo obtenido la mejor recompensa posible. Como ejemplo en este tipo de aprendizaje tenemos el AlphaGo [60], desarrollado por Google DeepMind, cuando en 2015 fue la primera máquina en ganar en el juego de mesa Go a un jugador profesional.

### 1.1.2. La clasificación en el aprendizaje automático

Como se ha comentado previamente, el aprendizaje automático puede ser usado en varios campos. Uno de ellos muy usado es el de la clasificación, donde podemos usar diferentes técnicas o algoritmos que nos ayudarán a extraer información relevante de la que podemos dar uso dependiendo del objetivo buscado.

Un ejemplo comúnmente conocido es el de la clasificación binaria para la detección de correo spam. Queremos detectar y clasificar un correo entrante para saber si lo consideramos como spam o no. Deberíamos tener en cuenta que este tipo de información es más sofisticada yd humanamente no seríamos capaces de identificar si es o no malintencionado. Así, gracias al aprendizaje automático es posible procesar las distintas características que conforman un correo de este tipo y por lo tanto usarlo como filtro para que sea adecuadamente clasificado antes que llegue a nuestro buzón. El uso del aprendizaje automático nos ayduaría a una clasificación más acertada de la que seríamos capaz de realizar por nosotros mismos.

Dentro de las numerosas técnicas que el aprendizaje automático nos ofrece, en este proyecto nos vamos a centrar en la clasificación. Sabemos que vamos a partir de un conjunto de datos previamente etiquetado y con valores discretos. Con el uso de algoritmos de aprendizaje automático para clasificación estos serán capaces de identificar patrones y “aprender”. Partiremos, inicialmente, de un conjunto de entrenamiento con un número de características y con el conocimiento de su etiqueta o salida. Cuando proveamos dicho algoritmo con un nuevo conjunto de datos desconocido, será capaz de indicar con cierta exactitud qué etiqueta debe tener cada uno de los elementos del nuevo conjunto.

Existen distintos algoritmos de aprendizaje automático para clasificación, dependiendo de la técnica usada por cada algoritmo, la precisión de acierto con cada dato será mejor o peor ya que la distribución de los datos y la naturaleza de los mismos podrá afectar de una manera u otra los cálculos internos que él mismo realice.

Como conclusión podemos decir que la clasificación es una técnica que, bien empleada, puede servir de ayuda a la hora de tomar decisiones y que, gracias a la ayuda de las máquinas, nos puede servir para llevar a cabo tareas que de forma humana serían imposibles de realizar. Así, cuando queremos llevar a cabo la clasificación de un conjunto de objetos que poseen gran cantidad de variables, el aprendizaje automático podrá ser usado para facilitar dicha tarea y conseguir una mejor precisión.

### 1.1.3. Aprendizaje automático en Python: el uso de la herramienta Scikit-learn

Podemos indicar que existen varios lenguajes de programación que son útiles para su uso en el aprendizaje automático, entre ellos tenemos Python, R o Java. Todo ellos ofrecen un abanico numeroso de posibilidades y de bibliotecas para la aplicación de métodos o algoritmos en inteligencia artificial.

De entre todos ellos el más usado y conocido en la comunidad de las ciencias de los datos es Python, el cual tiene además un ligero enfoque con la programación orientada a objetos. Podemos indicar que la computación de datos a bajo nivel en otro tipo de lenguajes es mejor y se puede obtener un mejor rendimiento que el uso de Python, pero sin embargo en él podemos encontrar una serie de librerías fáciles e intuitivas que ayudan en el manejo, procesado y visualización del conjunto de datos objetivo a ser analizado. Ejemplos como las librerías NumPy y SciPy han sido desarrolladas sobre otras capas como Fortran o C para operaciones vectorizadas de gran rendimiento en arrays multidimensionales.

Python es fácil de usar, siendo un lenguaje real de programación, el cual ofrece mejor estructura y soporte para programas extensos que scripts desarrollados en shell o batch. Permite modularizar nuestro programa y por lo tanto el código puede ser reusado en futuros desarrollos.

Existen gran cantidad de bibliotecas disponibles para usar en Python. En el trabajo que nos concierne nos vamos a basar en aquellas que sirvan de uso para aprendizaje automático. De entre ellas nos vamos a referir a la siguiente: **Scikit-learn** [45], una de las bibliotecas de libre uso de aprendizaje automático más utilizadas y populares en el día de hoy.

Podemos referirnos a scikit-learn como una caja de herramientas para su uso en Python pensada para ser utilizada en las ciencias del aprendizaje automático y la minería de datos. Existen otro tipo de librerías scikit disponibles que pueden ser usadas para distintos motivos,

como por ejemplo scikit-cuda para dar uso de librerías de GPU o scikit-image donde existe una colección importante de algoritmos para el procesado de imágenes.

El paquete Scikit-learn es de simple descarga e instalación, la cual, atendiendo a tener instaladas ciertas librerías dependientes de la misma, como son Numpy y Scipy, es compatible con distintos sistemas operativos y de fácil uso.

La misma posee una API de fácil uso y permite usar varios módulos que se pueden importar en cualquier proyecto de Python para darles uso según el objetivo que necesitemos. En nuestro caso usaremos diferentes algoritmos de aprendizaje automático para clasificación, y así llevar a cabo un estudio del comportamiento de los mismos en el conjunto de datos indicado previamente.

Algunas de las principales características de la API de scikit-learn pueden ser [11]:

- **Consistencia:** Los objetos que tenemos disponibles comparten una interfaz consistente y simple donde podemos encontrar unos métodos iguales para todos.
    - Clasificadores: Se encarga de definir mecanismos de instanciación de objetos y donde podemos aplicar el método `fit()` para que aprenda un modelo de un conjunto de datos que le pasamos como argumento.
        - Predición: ésta interfaz da la capacidad a un clasificador de añadir el método `predict()` que se encarga de coger un vector de instancias de testeo para predecir la salida de cada una basándose en el modelo que ha aprendido en la fase de aprendizaje (`fit()`). Esto se hace en aprendizaje supervisado para hacer una estimación del comportamiento del clasificador para nuevas instancias no vistas previamente. Algunos de los clasificadores pueden implementar también el método `predict_proba()` para tener una idea del comportamiento del mismo, donde obtendremos una matriz de probabilidades de la que cada instancia pertenece a una clase u otra. Por último, con estos métodos podemos saber en aprendizaje supervisado la precisión que ha tenido, ya que al saber las etiquetas que corresponden, podemos estimar cuántas instancias ha predecido correctamente y cuántas no.
    - Transformadores: En alguna ocasión necesitamos que nuestros datos sean pre-procesados o filtrados para poder pasarlos al clasificador siguiendo una estandarización previa. Debido a esto tenemos la opción de usar el método `transform()` que, una vez hecho el ajuste previamente con el método `fit()`, se encarga de asociar cada dato original del conjunto en uno nuevo dependiendo del que queramos aplicar. Como ejemplo a esto tenemos el

método de preprocesado `StandardScaler()` para poder normalizar los vectores de atributos en bruto a una representación que es más adecuada para los clasificadores, donde por ejemplo necesitamos que los datos tenga una distribución en una misma escala. Scikit-learn permite hacer ambas llamadas (`fit()` y `transform()`) en un solo método que podemos llamar `fit_transform()`. Técnicas para la reducción de la dimensión de los datos, que estudiaremos en capítulos posteriores, hacen uso de este tipo de escalado.

- **Representación de los datos**: En scikit-learn se representan los datos como un array multi-dimensional haciendo uso del paquete NumPi o el uso de *dataframe* que nos permite el paquete pandas disponible para Python.
- **No hay cambios de clases:** Los conjuntos de datos son representados como arrays de la clase Numpy o matrices de la librería SciPy por lo que las estructuras de datos no son inventadas ni deben ser analizadas para que Scikit-learn pueda procesarlas.
- **Flujo de datos**: Se pueden reusar las estructuras de datos pasando como argumento a un clasificador el mismo conjunto de datos transformado varias veces.
- **Instanciación por defecto:** Todos los algoritmos disponibles se pueden usar sin tener que configurar los parámetros a la hora de usarlos, los mismos vienen configurados con unos valores por defecto. Esto nos puede ayudar si queremos realizar una comparación o análisis de valores por defecto de forma rápida. Por otro lado tenemos la opción de modificar esos valores por defecto, que son los hiper-parámetros, y que a través de distintas técnicas podemos saber cuáles son los valores con los que obtendremos mejores resultados, algo que estudiaremos en capítulos posteriores.

Esta librería por ejemplo tiene disponible un paquete para el pre-procesado de los datos (`sklearn.preprocessing`), donde tenemos disponibles diferentes funciones útiles para filtrar los datos disponibles previo al entrenamiento de los distintos algoritmos. En esta librería existen otros paquetes útiles que nos ayudarán a llevar a cabo el estudio que nos concierne, como por ejemplo el optimizado de los hiper-parámetros que nos permite encontrar en una búsqueda exhaustiva cuáles son los mejores parámetros que se le pueden proveer a cada algoritmo para encontrar la mejor precisión, parámetros que no se aprenden en el entrenamiento del mismo. Otro paquete importante que será utilizado en capítulos posteriores es el de reducción de atributos de los vectores de datos. Con el uso del algoritmo conocido como PCA (*Principal Component Analysis*) [25] conseguiremos reducir el número de atributos que no son relevantes del conjunto de datos.

### 1.1.4. Elección de los parámetros de un algoritmo

Cómo se ha indicado previamente, scikit-learn da la posibilidad de realizar la llamada a cualquier algoritmo de clasificación utilizando unos atributos por defecto y el conjunto de entrenamiento que estemos tratando a analizar.

Sin embargo, y dado que esta librería es muy flexible y ajustable, es posible y recomendable utilizar para cada algoritmo un conjunto de datos para su entrenamiento y sus atributos que mejor se ajusten a los datos a tratar para obtener el mejor rendimiento en la clasificación.

Para este objetivo, sklearn ofrece la posibilidad de utilizar herramientas que automáticamente nos den estos dos parámetros, éstas son cross validation y grid search.

Ambas herramientas vamos a proceder a explicarlas a continuación:

#### Cross validation

Uno de los pasos importantes a tener en cuenta cuando queremos entrenar un algoritmo de aprendizaje automático supervisado es estimar cuál es su comportamiento, es decir, con qué precisión clasificará el modelo creado con un algoritmo elegido aquellos datos nuevos que no han sido vistos previamente.

Un conocido error metodológico para estos casos es el de sobreentrenar el algoritmo con unos datos para los que ya conocemos el resultado deseado, esto es comúnmente conocido en inglés como *overfitting* o en castellano “sobreajuste”, porque nos estamos ajustando demasiado a ese conjunto de datos específico, pero, ¿qué ocurre si los nuevos datos no se parecen tanto al conjunto de entrenamiento?. Por otro lado es posible que nuestro modelo sea demasiado simple y no obtengamos valores adecuados en su entrenamiento. Estaremos perdiendo la tendencia de los datos, en este caso estaríamos sufriendo “underfitting” o también “subajuste”, esto indica que no generaliza bien y puede ocurrir por ejemplo cuando intentamos ajustar un algoritmo lineal para un conjunto de datos que no sigue una distribución lineal. En la siguiente imagen mostramos la diferencia entre estos ajustes:

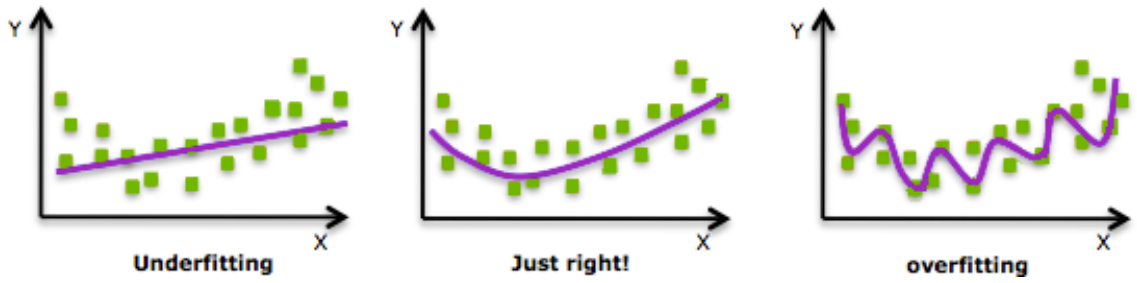



*Ilustración 1. Comparación de ajustes*
*https://towardsdatascience.com/train-test-split-and-cross-validation-in-python-80b61beca4b6*

Es necesario que busquemos la mejor forma de evaluar nuestro modelo cuidadosamente y, de esta forma, asegurarnos que vamos a obtener el mejor rendimiento del mismo. Para ello existen técnicas de validación cruzada, comúnmente conocidas como *cross-validation* [4,32], que nos indicarán cómo de preciso se comportará el algoritmo para datos nuevos no observados.

### ¿En qué consiste el método de *cross-validation*?

Es una técnica que se usa para poder evaluar los resultados de un análisis para que así se pueda garantizar que son independientes de la partición entre los datos que usamos del "conjunto de entrenamiento" y los datos del "conjunto de prueba".

Esta técnica consiste en dividir el conjunto de datos que disponemos para entrenar en dos partes, una que llamaremos "conjunto de entrenamiento" y otro "conjunto de prueba". Podemos decir que el primero es usado para, como su nombre indica, entrenar nuestro algoritmo, y el segundo es usado para probar y testear el rendimiento del mismo, ya que sabemos la salida que debe proporcionar, podemos preguntarle a nuestro algoritmo entrenado qué salidas piensa que deben tener los datos del conjunto de prueba y así saber si tiene una idea acertada de lo que debe predecir.

Lo normal es que se divida el conjunto de datos en un 70% para entrenamiento y un 30% para su validación o también un 80/20 , aunque este ratio debe ser considerado dependiendo del tamaño de nuestro conjunto de datos. Por ejemplo, si nuestro conjunto de datos no tiene suficientes entradas, el 30% puede no ser suficiente información para las distintas clases a clasificar para formar el conjunto de prueba.

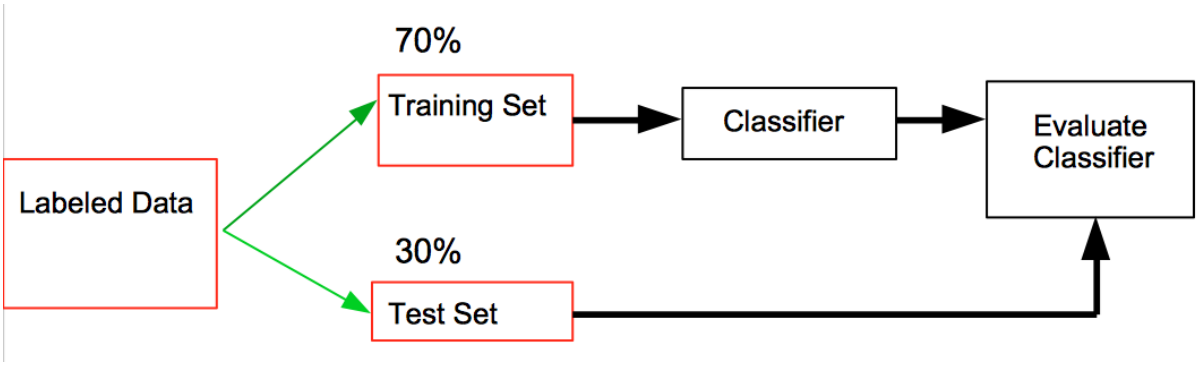



*Ilustración 2. Cross-validation*

Otro punto que debemos tener en cuenta es saber cómo están distribuidos las clases en ambos conjuntos, el de entrenamiento y el de prueba. Así es adecuado que exista una distribución equitativa de todas las clases que tenemos para que estén distribuidas en ambos conjuntos. Imaginemos que los datos están ordenados y que el 70% que hemos entrenado no tiene todas las clases, no estaríamos entrenando todas las posibilidades. Por lo tanto, el mejor camino para solventar este problema es el de dividir el conjunto de entrenamiento de

una forma aleatoria, es aquí donde el método de *cross_validation* entra en juego, esto siempre y cuando podamos encontrar los datos balanceados. En caso que no sea así la técnica de validación cruzada no servirá.

El módulo *model_selection* del paquete scikit-learn provee de distintos métodos que nos permitirán llevar a cabo esta división del conjunto para proceder a su entrenamiento y posterior evaluación.

El método de validación cruzada es muy similar a la anterior mencionada de dividir el conjunto en 70/30 pero aplicada a más subconjuntos. Lo que hacemos con estos métodos es dividir el conjunto en k subconjuntos y entrenar k-1 de esos subconjuntos para comprobar luego con el último conjunto que no se ha entrenado. En la siguiente imagen se observa como funciona este método:

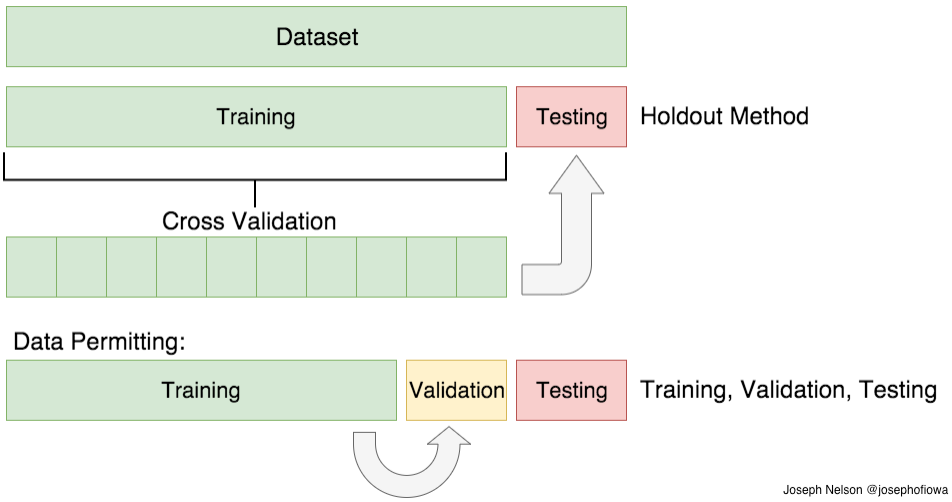



*Ilustración 3. Ejemplo de cross-validation*
*https://towardsdatascience.com/train-test-split-and-cross-validation-in-python-80b61beca4b6*

Existen muchas técnicas para validación cruzada, algunas de ellas que tendremos en cuenta para nuestro trabajo son:

## K-fold cross validation

En esta técnica lo que se hace es dividir el conjunto de entrenamiento en K subconjuntos, uno de los subconjuntos se usa como prueba y los otros K-1 como conjuntos de entrenamiento. Esto se realiza  K veces y se extrae la media aritmética de cada una de las divisiones realizadas para saber la precisión final de nuestro modelo entrenando. En la siguiente imagen podemos ver mejor cómo se comporta esta técnica:

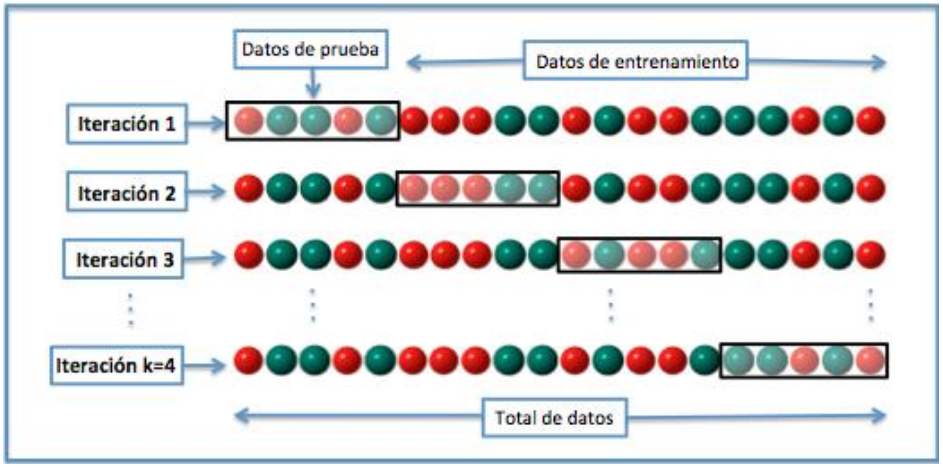



*Ilustración 4. k-fold cross-validation*
*https://es.wikipedia.org/wiki/Validaci%C3%B3n_cruzada#/media/File:K-fold_cross_validation.jpg*

Es normal encontrar el uso de esta técnica con 10 divisiones llamada *10-fold cross validation* pero tiene el problema del coste de computación para cada iteración si tenemos un conjunto de datos demasiado grande.

## Stratified KFold

Esta técnica de estratificación es una variación de KFold y se encarga de asegurar que cada clase está representada por igual en cada división que se hace. Como muchos algoritmos clasifican atendiendo a una ponderación de las clases que han entrenado, lo que se intenta con esta estrategia es que esa ponderación sea lo más justa posible, para que todas las clases sean representadas.

## Shuffle Split

Es un permutador aleatorio de valores en el conjunto de datos y proporciona índices para dividir datos en conjuntos de entrenamiento y prueba. Se encarga primero de barajar los datos y luego proporciona un conjunto de entrenamiento y otro para comprobación o testeo.

## Stratified Shuffle Split

Esta técnica es una mezcla de *Stratified Kfold* + *Shuffle Split* para que podamos conseguir una frecuencia lo más proporcional posible de todas las clases que tenemos tanto para el conjunto de entrenamiento como el de prueba. Así nos aseguramos que nuestros algoritmos se van a ajustar a un conjunto de entrenamiento donde nos aseguramos encontrar todos los tipos de clases. Esta será la técnica usada para tratar nuestro problema.

### La búsqueda exahustiva de hiper-parámetros : Grid Search

Podemos decir que el aprendizaje automático de un algoritmo tiene dos tipos de parámetros: uno de ellos es el conjunto de de parámetros de modelo, datos que le pasamos para su entrenamiento y sobre los que el algoritmo se ajusta. El otro parámetro son los valores que podemos ajustar al mismo algoritmo para realizar los cálculos necesarios en su aprendizaje, estos son conocidos como hiper-parámetros, aquellos que no se aprenden dentro del mismo algoritmo en su entrenamiento.

Imaginemos que ya tenemos seleccionados los algoritmos que queremos entrenar, pues bien, todos o la mayoría de ellos dan la posibilidad de ajustar cada uno de sus hiper-parámetros. Los hiper-parámetros puede aceptar valores en distintos rangos y dependiendo de cada uno obtendremos un rendimiento mejor o peor. Como es evidente, queremos sacar la mejor rendimiento de cada algoritmo, pero hacer una evaluación del comportamiento de cada uno con cada uno de los hiper-parámetros puede ser tedioso y muy costoso en tiempo si lo hacemos manualmente.

La librería scikit-learn posee una técnica que ayuda en la búsqueda de una combinación óptima en los valores de los hiper-parámetros, esta técnica se llama *grid search*.

Es posible y recomendable que se realice la búsqueda de estos parámetros para obtener el mejor rendimiento al usar la técnica *cross-validation*.

Todo parámetro que le pasemos al algoritmo puede ser optimizado. Como cada uno puede tener distintos nombres de parámetros, es posible averiguarlo con la siguiente llamada: `estimator.get_params()`, donde *estimator* puede ser cualquiera de los algoritmos, tanto de clasificación o regresión, que podemos usar dentro de la biblioteca de scikit-learn.

La búsqueda de los parámetros va a consistir en lo siguiente:

- La elección del algoritmo al que le queremos buscar los parámetros, de los muchos que hay disponibles y que se desarrollarán más adelante los seleccionados.
- Un espacio de parámetros.
- Un método para la búsqueda de los parámetros.
- Un esquema de *cross-validation*.
- Una función de puntuación.

Uno de los métodos para la búsqueda de los parámetros es el que hemos comentado previamente, *grid search*, y que scikit-learn provee a través del método `GridSearchCV`, el cual se va a encargar de comprobar exhaustivamente todas las combinaciones en el espacio de parámetros y sus valores posibles seleccionado por nosotros.

Es adecuado indicar que la computación para este tipo de método crece exponencialmente a medida que tenemos mayor número de candidatos posibles. Así es

necesario tener en cuenta el tiempo necesario para la búsqueda del mejor candidato y que dependerá del espacio a probar y del conjunto de datos que estemos tratando.

Esta técnica se encarga de la búsqueda exhaustiva del mejor candidato a través de una malla de valores posibles especificados en su parámetro `param_grid`, que es un diccionario o lista de diccionarios con los distintos parámetros posibles y los valores que queremos que testee.

Por otra parte el tipo de *cross-validation* que usa esta búsqueda por defecto es la de 3-fold cross validation que es igual que la técnica k-fold explicada previamente pero en esta ocasión solo usa 3 pliegues o divisiones. Este valor se pueden modificar y debido al procesamiento limitado de nuestra máquina, para nuestro trabajo vamos a optar por realizar 2 pliegues estableciendo el siguiente parámetro `cv=2`, debido al coste de computación y así obtener los datos en un tiempo razonable.

Por último, para evaluar el comportamiento del algoritmo en el conjunto de posibles parámetros elegidos, *GridSearch* usa por puntuación la misma que el algoritmo que se le pasa en el parámetro `estimator`, y que por defecto es *Accuracy Score,* métrica que indica el porcentaje de ejemplos del conjunto de entrenamiento que el algoritmo ha clasificado correctamente. Esta forma de puntuación se puede cambiar para la función `GridSearchCV` a través de su parámetro scoring, sin embargo la vamos a dejar por defecto.

Todos esto es importante tenerlo en cuenta para entender cómo funciona y qué información podemos extraer del método `GridSearchCV`. Esto es debido a que usa la misma API que nos encontramos para cada algoritmo, es decir, usando el método fit en el conjunto de datos, se evaluan todas las combinaciones posibles de parámetros para el algoritmo. Al finalizar todo el espacio de posibles valores para los distintos parámetros podremos extraer información importante como por ejemplo con el método `best_params_`, que devuelve un diccionario con los parámetros con los que hemos conseguido la mejor puntuación, indicada previamente como *Accuracy Score*.

### 1.1.5. Clasificación y evaluación: Métricas de un algoritmo

Previamente hemos indicado el uso de la precisión de un algoritmo como medida de rendimiento del mismo. Sin embargo existen otras métricas que pueden ser consideradas para evaluar el rendimiento de un clasificador, métricas que nos servirán para nuestro trabajo en la medida de poder evaluar y comparar distintos algoritmos de clasificación Estas métricas nos ayudarán a elegir cuál es el mejor de todos los algoritmos evaluando los distintos valores obtenidos de las métricas.

Para entender las distintas métricas que vamos a plantear a continuación es necesario entender previamente de donde se extraen las mismas. Para ello vamos a proceder a explicar

lo que es la matriz de confusión [55], una matriz que nos ayuda a saber cuál ha sido el rendimiento de un algoritmo. Esta matriz simplemente nos muestra en un cuadro los siguientes valores, una vez realizado el entrenamiento y posterior validación con aquellos datos que tenemos para testear usando la validación cruzada: falsos positivos, falsos negativos, verdaderos positivos y verdaderos negativos. Para entender mejor este concepto procedamos a mostrar cómo sería esta matriz para una clasificación binaria:

| | | **Predicción** | |
|---|---|---|---|
| | | **Positivos** | **Negativos** |
| **Observación** | **Positivos** | Verdaderos Positivos (VP) | Falsos Negativos (FN) |
| | **Negativos** | Falsos Positivos(FP) | Verdaderos Negativos(VN) |

*Ilustración 5. Matriz de confusión para clasificación binaria*

Como podemos observar, las etiquetas conocidas corresponden a las filas, y las que el algoritmo predice a las columnas.Para definir correctamente estos valores podemos indicar que:

**Verdaderos Positivos (VP)**: tenemos que es la cantidad de aquellas observaciones que fueron clasificados como pertenecientes a una clase y acertó.

**Falsos Positivos (FP)**: Esas observaciones que no pertenecían a una clase pero se llegaron a considerar perteneciente a ellas, son considerados positivos pero como sabemos la salida también sabemos que se ha equivocado.

**Falsos Negativos (FN)**: Aquellas observaciones que pertenecían a una clase pero no se consideraron en ella, considerados como negativos.

**Verdaderos Negativos (VN)**: es la cantidad de aquellas observaciones que no pertenecían a una clase y las clasificó correctamente.

De todo algoritmo podemos sacar esta matriz de confusión, así tenemos el método confusion_matrix del paquete `sklearn.metrics` [74] de donde podemos extraer los 4 valores indicados previamente.

A partir de estos 4 valores previos podemos sacar nuevas métricas de rendimiento que nos ayudarán a extraer más información.

Estas métricas sirven para medir clasificaciones binarias. Como nuestro problema está enfocado a clasificación multiclase es necesario sacar el valor promedio para cada una, así para cada métrica tenemos $VP_i$ que son los Verdaderos Positivos para la clase i, $VN_i$ = Verdaderos Negativos para la clase i, $FP_i$= Falsos Positivos para la clase i y por último $FN_i$ = Falsos Negativos para la clase i. Las métricas que vamos a tener en cuenta en este trabajo son las siguientes:

**Exactitud**

Nos sirve para sacar el porcentaje de observaciones que ha clasificado correctamente, cuanto más cercano sea su valor a 1 mejor será.

Su fórmula es:

$$\text{Exactitud} = \frac{VP \ + \ VN}{VP + VN + FP + FN}$$

$$\text{Exactitud promedio} = \frac{1}{C}\sum_{i=1}^{C}\frac{VP_i + VN_i}{VP_i + FP_i + VN_i + FN_i}$$

Donde C es el número de clases cuando realizamos clasificación multiclase.

**Sensibilidad**

Mide la habilidad del clasificador de encontrar todas las observaciones positivas, buscamos que este valor sea lo mayor en una escala del 0 al 100. Su fórmula es como sigue:

$$\text{Sensibilidad} = \frac{VP}{(VP \ + \ FN)}$$

$$\text{Sensibilidad promedio} = \frac{1}{C}\sum_{i=1}^{C}\frac{VP_i}{VP_i + FN_i}$$

Donde C es el número de clases, que en nuestro caso serán 7

**Precisión**

La precisión mide la habilidad de un clasificador de no etiquetar como positivo una observación que se debe considerar como negativa. Cuanto mayor sea este valor mejor será. Su fórmula es :

$$Precision \ = \frac{VP}{VP + \ FP}$$

$$\text{Precision promedio} = \frac{1}{C}\sum_{i=1}^{C}\frac{VP_i}{VP_i + FP_i}$$

Donde C es el número de clases, que en nuestro caso serán 7

**Log Loss**

Esta métrica [18] mide la incertidumbre que ha tenido nuestro clasificador con respecto a la etiqueta real que correspondía, es decir, si la clasificación ha variado o desviado mucho o poco con respecto a lo que debía ser. Lo que se pretende es que este valor sea lo más cercano a 0. Lo que hace es cuantificar la precisión del clasificador penalizando las clasificaciones falsas. Minimizando este valor quiere decir que maximizamos la precisión del clasificador.

Para calcular este valor, el clasificador asigna una probabilidad de pertenecer a cada clase para cada ejemplo. Matemáticamente la fórmula para calcular este valor es:

$$\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log p_{ij}$$

donde N es el número de instancias en nuestro conjunto de datos, M es el número de etiquetas posibles, $y_{ij}$ es un valor binario que nos dice si para la instancia i su etiqueta es j, y $p_{ij}$ es la probabilidad con el que el clasificador le ha asignado la etiqueta j a la instancia i.

**Matriz de confusión**

Para el problema que estamos tratando hablamos de una clasificación de multi-clase donde tenemos 7 tipos de cubierta arbórea. Así tenemos la siguiente matriz de confusión que extraemos del método `confusion_matrix` de la clase `sklearn.metrics` que nos da uno de los algoritmos que vamos a estudiar:

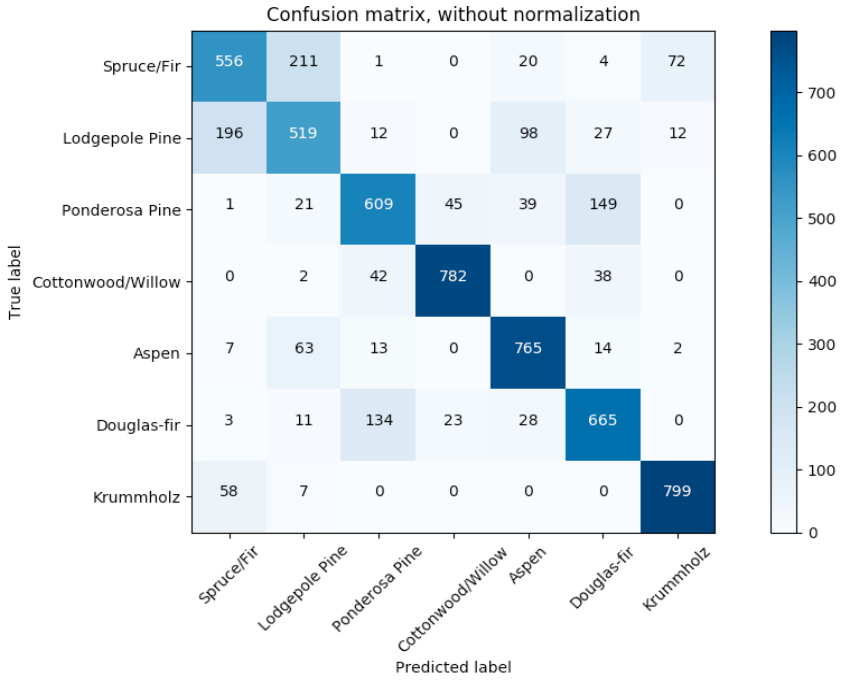



*Ilustración 6. Matriz de confusión para DecisionTrees sin normalizar*

Esta visualización de matriz de confusión sirve para evaluar la calidad de la salida de cada algoritmo. Aquellos valores de la diagonal son los que representan la cantidad de valores de cada clase en los cuales se ha predecido una etiqueta y corresponde con el mismo que debería indicar, cuánto menor número existan en las casillas fuera de la diagonal indicará que mejor ha clasificado las instancias.

También tenemos la opción de normalizar los datos ya que puede ser interesante en caso de que los datos están desbalanceados y nos permite saber qué clase está siendo mal clasificada. Así para la misma matriz de confusión anterior tenemos:

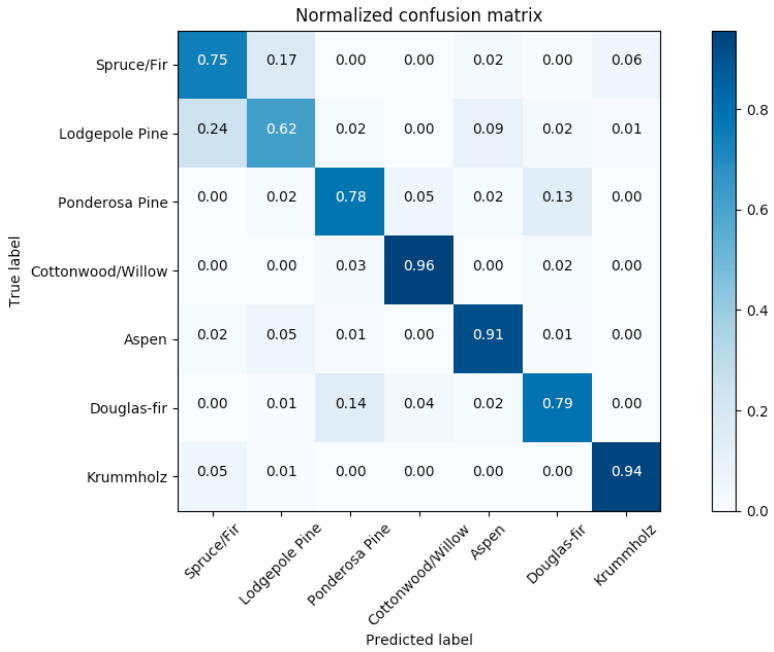



*Ilustración 7. Matriz de confusión para DecisionTrees normalizado*

En esta ocasión podemos ver que clasifica muy bien para las clases *Cottonwood/Willow* o *Krummholz* pero tiene problemas para la clase *Lodgepole Pine* porque habrá algunas características que no le ayudan a distinguir entre las clases.

## 1.2. Algoritmos para clasificación

El paquete Scikit-learn posee gran variedad de algoritmos que pueden ser usados para el tratamiento y procesado del conjunto de datos que le pasemos como argumento. Muchos de ellos tienen la posibilidad de predecir valores continuos futuros en el tiempo o de clasificar información, ya sea a partir de datos observados previamente o no.

Vamos a centrarnos en el campo de la clasificación basada en conjuntos de datos previamente observados, aprendizaje supervisado, identificando a qué categoría pertenece un objeto nuevo que no hemos observado previamente.

Existen distintas categorías para estos algoritmos atendiendo al tipo de clasificación que realicen. Vamos a destacar los más usados e importantes que podemos encontrar para resolver numerosos problemas de clasificación en los que basaremos nuestro proyecto y los que analizaremos. Así podemos encontrar los siguientes dentro de estas categorías según su naturaleza:

- Linear Models:
    - Logistic Regression
    - Perceptron
- Neural Network
    - MultiLayerPerceptron

- Discriminant Analysis:
    - Linear Discriminant Analysis
    - Quadratic Discriminant Analysis
- Support Vector Machines:
    - Linear Support Vector Machines
    - Polynomial Support Vector Machines
- Neighbors:
    - Kneighbors
- NaiveBayes:
    - BernouilliNB
    - GaussianNB
    - MultinomialNB
- Trees:
    - DecisionTree
- Ensemble:
    - AdaBoost
    - RandomForest

A continuación vamos a proceder a explicar el funcionamiento de cada uno. Como comentamos en la sección anterior, la librería scikit-learn nos permite hacer la llamada a cada algoritmo con los respectivos argumentos, realizando las operaciones y cálculos necesarios para identificación de patrones y clasificar nuevos datos suministrados.

El análisis del comportamiento a través de métricas de cada algoritmo nos ayudarán a entender mejor cómo funciona cada uno internamente y así extraer conclusiones de los valores para llevar a cabo la comparativa.

Además de los algoritmos indicados previamente existen otros que también pueden ser usados para clasificación pero, por no ser tan relevantes o tan usados en otros sistemas, los citaremos e incluiremos sus datos aunque no analizemos en profundidad su comportamiento.

Muchos de ellos son capaces de realizar clasificación multiclase pero hay otros que por su naturaleza son binarios, distinción entre dos clases. Para solventar el problema donde tenemos que clasificar tres o más clases, se usan distintas estrategias para así convertirlos en clasificadores multiclase. Estas técnicas son conocidas como *one-vs-one* y *one-vs-all* o también conocida como *one-vs-rest* (OvR). Cada una de ellas se comporta de la siguiente forma:

- *one-vs-one*: Se encarga de dividir el problema de N clases en N(N-1)/2 subproblemas binarios. Se realizan clasificaciones entre dos clases {Ci, Cj} donde obtendremos un grado de confianza en favor de Ci en el rango de [0,1]. Cada subproblema binario se

almacena en una matriz de votos. Para extraer cual es la clase a la que pertenece se usa la estrategia del voto ponderado  la que alcanza la mayor confianza total es la que se predice. [21]

- *one-vs-all*: Esta estrategia se basa en realizar una clasificación por cada clase que tengamos. Si tenemos 3 clases a distinguir entre nuestro conjunto de datos entonces realizaremos 3 clasificaciones binarias. Así por ejemplo, para la clasificación de la primera clase, vamos a considerar positivos todos aquellos que cumplan la función que le pasemos, y negativos todos los demás que no la cumplan, aquí estaríamos haciendo una clasificación binaria, y así haremos para todas las demás clases. Supongamos que $f_i(x)$ es la clasificación i-ésima para el ejemplo x. La clasificación final para f(x) = arg $\max_i$ $f_i(x)$, es decir, el máximo valor de todas las clasificaciones binarias realizadas. [54, 49]

### 1.2.1. Linear Models

#### Perceptron

El perceptrón es un modelo de neurona simple. En 1958 el psicólogo Frank Rosenblat [52] desarrolló este modelo basado en el de McCulloch y Pitts [36] y en una regla de aprendizaje basada en la corrección del error, a este modelo le llamó Perceptrón. Lo que más llamó la atención en este modelo es su capacidad para aprender a reconocer patrones. El perceptrón se basa en una serie de sensores o entradas desde donde recibe los datos a clasificar o reconocer y donde tenemos además una neurona de salida para indicarnos si pertenece a una clase u otra, activándose o no dependiendo si la salida es 1 o 0.

Para explicar en qué consiste el algoritmo del perceptrón simple vamos a explicar en qué funciones se basa para llevar a cabo una clasificación de datos, basada en el concepto de Perceptrón simple.

Vamos a comenzar suponiendo que tenemos la función *f* de $R^n$ en {-1, 1}, a la que le podemos aplicar un patrón de entrada $x = (x_1, x_2, ..., x_n)^T \in R^n$ y donde tendremos una salida deseada $z \in \{-1, 1\}$, o lo que es lo mismo, $f(x) = z$.

Ese patrón de entrada lo vamos a considerar como cada característica que tiene nuestro conjunto de datos y al que le vamos a pasar dicha función. Como tenemos un número de patrones de entrada para llevar a cabo la clasificación vamos a tener la siguiente relación: $\{x^1, z^1\}$, $\{x^2, z^2\}$....$\{x^p, z^p\}$, donde $x^i$ es el patrón de entrada $i \in R^n$ y $z = f(x^i)$.

La función lo que realiza es una partición del conjunto de entradas en dos espacios, por una parte tendríamos los patrones de entrada cuya salida es +1 y por otro los patrones cuya salida es -1, así podemos decir que esta función es capaz de distinguir entre dos clases.

Debido a que nuestro problema trata la clasificación de más de dos clases, el perceptrón simple admite la técnica de clasificación *one-vs-all* debido a que las salidas que obtenemos son continuas y numéricas, técnica explicada previamente en este capítulo.

¿Y cómo podemos construir un modelo que cumpla con esa función? Partiremos para ello de una unidad de proceso bipolar que cumple la siguiente función:

$$\left\{ \begin{matrix} 1\ si\ w_1x_1 + w_2x_2 + \cdots + w_nx_n \geq \theta \\ -1\ si\ w_1x_1 + w_2x_2 + \cdots + w_nx_n \leq \theta \end{matrix} \right\}$$

donde tenemos que los parámetros $w_i$ son los llamados pesos sinápticos. Estos pesos supondrán la importancia o la ponderación que le damos a cada valor de entrada. Por otro lado tenemos la suma ponderada que llamamos potencial sináptico y por último el umbral que es $\theta$. Cuando la salida de dicha función es 1 se dice que está activa y en caso contrario su salida será -1 o que está inactiva.

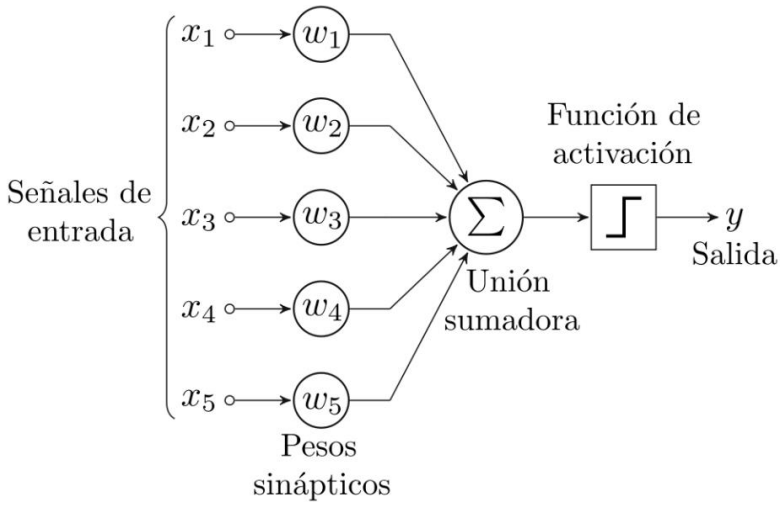



*Ilustración 8. Ejemplo de Perceptrón 5 unidades*
*https://es.wikipedia.org/wiki/Perceptr%C3%B3n#/media/File:Perceptr%C3%B3n_5_unidades.svg*

El valor inicial de los pesos sinápticos y el umbral se realiza a través de un proceso adaptativo tomando unos valores iniciales aleatorios que se irán modificando según se acerque o no a la salida deseada en el entrenamiento. La regla para la modificación de los pesos sinápticos es la conocida Regla de aprendizaje del Perceptrón Simple . Esta regla consiste en lo siguiente: $w_j(k+1) = w_j(k) + \Delta w_j(k)$, para k = 1, 2, … y donde tenemos que $\Delta w_j(k) = \eta(k)[z(k)-y(k)]x_j(k)$, explicándolo podemos decir que, la variación del peso para $w_j$ es proporcional al producto del error (z(k)-y(k)) por la componente j-ésima del patrón de entrada que hemos introducido en la iteración k, es decir, $x_j(k)$. La constante ɳ es lo que llamamos tasa de aprendizaje e indica el valor por el que cambiaremos cada peso sináptico.

Así esta regla es de detección y corrección del error ya que aprende, o modifica los pesos, cuando existe una equivocación.

El algoritmo del perceptrón funciona de la siguiente forma:

1. Iniciamos los pesos sinápticos con valores aleatorios en el intervalo [-1,1]. Ir al paso 2 con k=1
2. Iteración k-ésima:
    a. Calcular la salida y(k) según la función comentada previamente
3. Corregir pesos sinápticos:
    a. Si la salida de y(k) ≠ z(k) (la salida no es la deseada), entonces modificamos pesos aplicando la regla de aprendizaje del perceptrón simple comentada previamente.
4. Fin
    a. Si los pesos sinápticos no se han modificados en las últimas *p* iteraciones entonces la red se ha estabilizado
    b. En otro caso ir al paso 2 con k = k+1

El problema de este tipo de algoritmo es que solo es capaz de clasificar todos aquellos patrones que son linealmente separables. Dos conjuntos de patrones son linealmente separables si un número de ellos al aplicar la función son $>= \theta$ y otro número de ellos $< \theta$.

Así encontraremos que el algoritmo del perceptrón no convergerá fácilmente si el conjunto de datos que estemos tratando no es separable linealmente y por lo tanto tiene sus limitaciones a pesar de su facilidad de implementación.

### Logistic Regression

A pesar de su nombre, Regresión Logística es un modelo para clasificación, no para regresión. Este algoritmo es muy simple de implementar pero sin embargo es difícil que converja si los datos que estemos tratando no son separables linealmente. Es un clasificador lineal binario pero que usa la técnica OvR (One versus Rest) que vimos en la página 34. [21]

En estadística, Regresión Logística es un modelo de regresión donde la variable dependiente es categórica, aquella que puede coger valores fijos o un número de valores posibles. Estas variables dependientes son aquellas clases objetivos que queremos predecir mientras que las variables independientes son las distintas características que componen nuestro conjunto de datos y que vamos a usar para entrenar nuestro modelo.

Para comprender cómo funciona este método es necesario también entender el ratio de probabilidad, que es la probabilidad con la que ocurra un evento. Este ratio lo podemos escribir de la siguiente forma: $\frac{p}{1-p}$ donde p es la probabilidad para que ocurra un evento

positivo, donde indicamos con qué certeza se va a producir un evento, por ejemplo si una persona comprará una casa. En una clasificación multiclase tendremos la probabilidad de que un dato pertenezca a una de las distintas posibles clases disponibles. Podemos ir más allá definiendo lo que es la función logit que es el logaritmo de las probabilidades: logit(p) = $log\frac{p}{(1-p)} = log(p) - log(1-p)$. Esta función toma valores 0 y 1 para una clasificación binaria y los transforma en valores reales, los cuales se pueden usar para expresar una relación lineal entre las características del conjunto de datos y las probabilidades logísticas[48]:

$$logit(p(y=1|x)) = w0x0 + w1x1 + \cdots + wmxm = \sum_{i=0}^{m} wixi = \mathrm{w^T x}$$

Como estamos interesados en la probabilidad de si un ejemplo pertenece o no a una clase, vamos a usar la inversa de la función logit y que podemos llamar como la función logística y en muchas ocasiones también nombrada como la función sigmoidea por una representación en forma de S:

$$\phi(z) = \frac{1}{1+e^{-z}}$$

Aquí podemos indicar que *z* es la entrada a la red y resulta de la combinación lineal de los pesos y las respectivas características de cada ejemplo, es decir, $z=w^Tx = w_0 + w_1x_1+...+w_mx_m$. Cada característica tendrá un peso debido a que cada una de ellas influirá más o menos para la decisión final de a qué clase pertenece. Por ejemplo si queremos realizar una clasificación para determinar si un individuo es feliz o está triste, podemos asignarle pesos positivos a características que influyan en su estado de ánimo para ser feliz, o podemos asignar pesos negativos a aquellas características que influyan negativamente.

Por lo tanto para predecir si un ejemplo pertenece a una clase u otra solo tenemos que multiplicar cada característica con su correspondiente peso. La puntuación obtenida es conocida como *logit score*. Este valor lo pasaremos por la función softmax para extraer la probabilidad final. La función softmax es famosa por ser usada para calcular las probabilidades y además nos va a dar una salida en el rango (0,1). La suma de todas la salidas nos daría 1. Esta función lo que hace es coger cada *logit score*, previamente calculado, y encuentra la probabilidad. Su fórmula es como sigue:

$$\sigma: \mathbb{R}^K \rightarrow [0,1]^K$$

$$\sigma(z)_j = \frac{e^{zj}}{\sum_{k=1}^{K} e^{zk}} \; for\; j = 1, \ldots, K.$$

Esta función es usada para representar una distribución categórica, una distribución de probabilidad sobre K diferentes posibles salidas.

Una representación de cómo sería el flujo de este tipo de algoritmo podríamos contemplarlo de la siguiente forma:

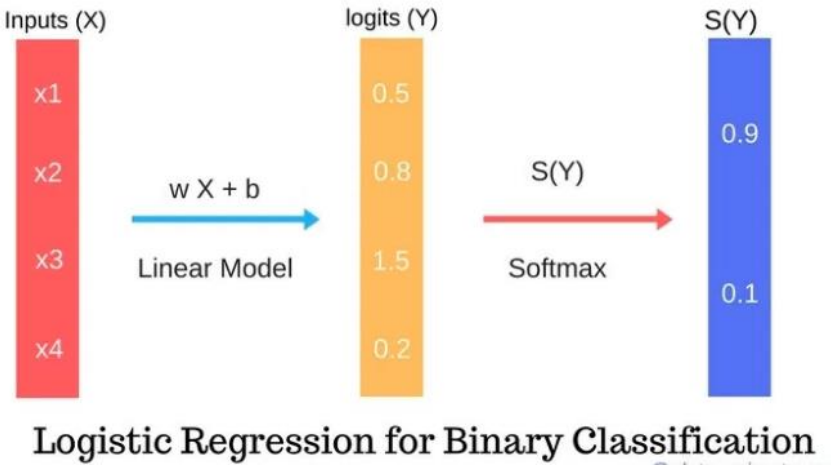



*Ilustración 9. Clasificación binaria usando Logistic Regression*
*dataaspirant.com/2017/03/02/how-logistic-regression-model-works/*

Este algoritmo es muy usado en muchos campos, incluyendo entre ellos el aprendizaje automático, la mayoría de ellos usados a nivel médico, como por ejemplo para averiguar la gravedad de un paciente se ha llegado a usar este tipo de algoritmo [62] o por ejemplo para predicción de que se produzca una tormenta geomagnética usando modelos de este tipo [46]. Su uso también se puede ver reflejado en el campo de la ingeniería, especialmente para saber la probabilidad de fallo de un proceso, un producto o un sistema [44] .

### 1.2.3. Neural Network

#### MultiLayerPerceptron

El Perceptrón Multicapa es una generalización del Perceptrón Simple y surgió como consecuencia de las limitaciones que tenía a la hora de clasificar conjuntos de datos que no eran linealmente separables. Minsky y Papert [39] pudieron mostrar en 1969 que combinando el uso de varios Perceptrones simples, vamos a considerarlo como capas ocultas, solucionaría el problema de clasificación para problemas que no son lineales. Sin embargo para esta solución no existía una definición de cómo poder adaptar los pesos sinápticos para cada perceptrón en la capa oculta, ya que la regla del Perceptrón simple no puede aplicarse en este problema. A pesar de esto, la idea de combinación de varios Perceptrones simples sirvió de ayuda para los estudios realizados por Rummelhart, Hinton y Williams cuando en 1986 [53] presentaron una forma de retropropagación del error cometido y cómo poder adaptar los pesos sinápticos a través de una regla, conocida como regla delta generalizada o retropropagación.

En las redes neuronales, el Perceptrón multicapa es una de las arquitecturas más usadas para la resolución de problemas, debido a su capacidad como aproximador fundamental y a su facilidad de uso y aplicabilidad. Esto no quiere decir que sea una implementación perfecta ya que también posee distintos problemas y limitaciones como por ejemplo el proceso de aprendizaje para problemas complejos con gran cantidad de variables.

La arquitectura de un Perceptrón multicapa [39], como su nombre indica, se basa en una estructuración de sus neuronas en varios niveles o capas. Esta arquitectura es una red de alimentación hacia adelante (*feedforward*) y donde podemos encontrar una capa de entrada, que podemos llamar sensores o el conjunto de características de nuestro conjunto de datos, otra capa de salida, que serán los distintos tipos posibles a clasificar, y un número determinado de capas intermedias de unidades de proceso, que podemos llamar también como ocultas ya que no tienen conexión con el exterior. El papel que desempeña la capa oculta o intermedia es la de una proyección de los patrones de entrada en un cubo cuya dimensión viene dada por el número de unidades de la capa oculta. Con esto lo que se pretende es realizar una proyección en la que resulten separables linealmente los patrones de entrada de forma que el valor que consigamos en la capa de salida sea lo más correcta posible.

Como ejemplo a esto tenemos el problema de clasificar la función binaria XOR. Si representamos dicha función en el espacio y realizamos la proyección de los puntos tal como se ha explicado previamente podemos comprobar que conseguimos que dichos datos de entrada sean separables linealmente, y por lo tanto con ello una clasificación adecuada de sus patrones.

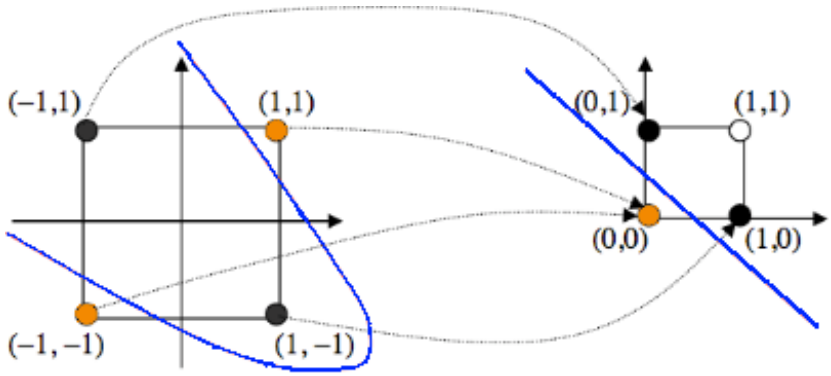



*Ilustración 10. Proyección de los patrones de entrada*
*http://www.lcc.uma.es/~munozp/documentos/modelos_computacionales/temas/Tema5MC-05.pdf*

Las unidades de salidas están conectadas sólo con la última capa oculta. En este tipo de red lo que se pretende es establecer una relación entre un conjunto de entrada y otro de salida, así tenemos la siguiente relación: $(x_1,x_2,x_3...x_n) \in R^n \dashrightarrow (y_1,y_2,y_3....y_m) \in R^m$. De esta forma partimos de un conjunto *p* de patrones de entrenamiento donde sabemos que para el patrón

de entrada ($x^k_1$,$x^k_2$,...$x^k_n$) le corresponde la salida ($y^k_1$, $y^k_2$,..$y^k_m$) con k =1,2,..p. En la siguiente figura mostramos cómo sería una representación de este tipo de red neuronal:

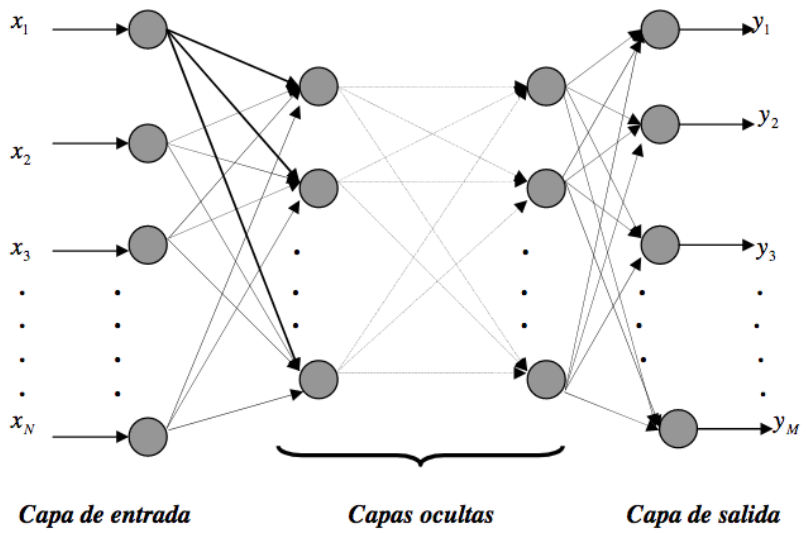



*Ilustración 11.Topología de un Perceptrón Multicapa*
*http://www.lcc.uma.es/~munozp/documentos/modelos_computacionales/temas/Tema5MC-05.pdf*

Para entender esta relación indicamos que la capa de entrada tiene tantas neuronas como variables o características tiene nuestro conjunto de datos. Por otra parte en la capa de salida tendremos tantas unidades de proceso como salidas deseadas existentes en nuestro sistema.

La determinación del número de capas ocultas es algo que no está establecido de forma concreta. El uso de un mayor número de capas ocultas no asegura que obtengamos una mejor precisión a la hora de encontrar la salida deseada pues los métodos empleados para entrenar este tipo de red puede resultar más costoso con un mayor número de capas y puede llevar a un tiempo demasiado largo para encontrar los mejores valores de su entrenamiento.

## Función de red neuronal multicapa

La computación realizada en este tipo de red neuronal para extraer la salida $y_i$ suponiendo una red con una sola capa oculta sería de la siguiente forma:

$$y_i = g_1\left(\sum_{j=1}^{L} w_{ij} s_j\right) = g_1\left(\sum_{j=1}^{L} w_{ij}\left(g_2\left(\sum_{r=1}^{N} t_{jr} x_r\right)\right)\right)$$

Aquí tenemos que $w_{ij}$ es el peso sináptico de la conexión entre la unidad de salida *i* y la unidad de proceso oculta *j*. *L* sería el número de unidades de proceso en la capa oculta; $g_1$ sería la función de transferencia para las unidades de proceso de la capa de salida, las cuales pueden ser la función identidad, la tangente hiperbólica o una función logística; $t_{jr}$ es el peso sináptico que conecta la unidad de proceso *j* de la capa oculta con la entrada *r.* Por último tenemos la función $g_2$ que es la función de transferencia de las unidades de proceso de la

capa oculta, las cuales también pueden ser del tipo mencionado previamente para las unidades de proceso de la capa de salida.

Para la finalización del diseño de este tipo de red neuronal tenemos que identificar cuales son los pesos sinápticos que se deben asociar para cada entrada de las unidades de proceso. Así se realiza un proceso de entrenamiento a partir del conjunto de patrones de entrenamiento del que partimos y se realizará una evaluación del error cometido para ir adaptando la ponderación de cada peso sináptico. Vamos a proceder a la explicación de este algoritmo de adaptación de los pesos sinápticos conocido como regla delta o algoritmo de retropropagación.

## Regla Delta o algoritmo de retropropagación

Este algoritmo lo que pretende es conseguir el menor error cometido para cada salida deseada, de esta forma queremos que las salidas previstas sean lo más parecidas a la salida deseada. Lo que pretende es la determinación de los pesos sinápticos de forma que el error total cometido sea el mínimo:

$$E = \frac{1}{2}\sum_{k=1}^{p}\sum_{i=1}^{M}(z_i(k) - y_i(k))^2$$

El algoritmo de retropropagación utiliza el método del *gradiente descendiente* y que hace uso del vector gradiente, siendo uno de los más usados para este tipo de algoritmo.

Este algoritmo se puede explicar en los siguientes pasos [39]:

1. Iniciar con unos pesos sinápticos cualesquiera, elegidos al azar.
2. Elegir el conjunto de datos para la capa de entrada elegidos al azar que servirán para su entrenamiento.
3. Esperar que la red genere un vector de datos de salida a través del a propagación hacia adelante.
4. Comparación de la salida obtenida con la deseada
5. La diferencia obtenida entre la salida deseada y la obtenida se usará para volver a ajustar los pesos sinápticos de las neuronas de la capa de salida.
6. Ese error cometido es propagado hacia atrás a través de cada unidad de procesamiento de cada capa oculta hasta llegar al ajuste de los pesos de las unidades de la capa de entrada.

7. Este proceso se irá repitiendo hasta que se alcance unos valores de error mínimo al que queremos llegar o se alcance un número de repeticiones máximo a llevar a cabo en dicho algoritmo.

Este tipo de algoritmo tiene el inconveniente de tener que realizar muchas iteraciones para funciones que en su representación visual contienen estructuras de valle grandes y estrechas. De hecho, el gradiente de descenso es la dirección en la cual la función de pérdida decrece más rápidamente pero esto no quiere decir que converga de una forma más rápida. Un ejemplo que muestra este tipo de problema lo encontramos en la siguiente representación:

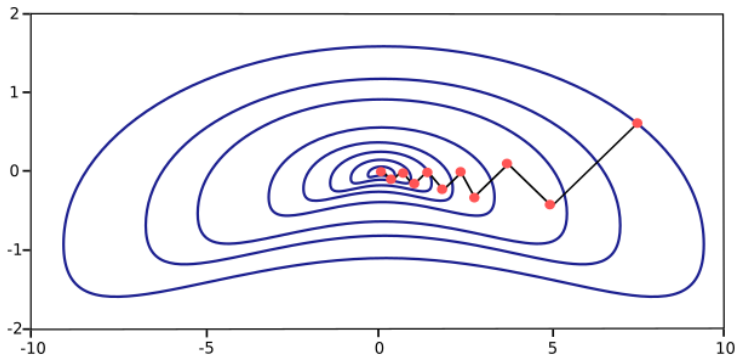



*Ilustración 12. Iteraciones del gradiente descendiente*
*https://www.neuraldesigner.com/blog/5_algorithms_to_train_a_neural_network*

Existen otros algoritmos para entrenar redes neuronales donde hacen usos de distintos métodos para llevar a cabo de una forma diferente la convergencia de la red y conseguir con ellos mejores resultados u obtenerlos de forma más eficiente, sin embargo hacen un mayor uso de recursos al tener que almacenar más estructuras de datos como puede ser la matriz Hessiana. Estos distintos algoritmos son Método de Newton, Gradiente conjugado, Método Cuasi-Newton, Algoritmo de Levenberg-Marquardt [15,22,40]

En la siguiente gráfica se muestra una comparativa de uso de memoria y velocidad de convergencia de cada uno:

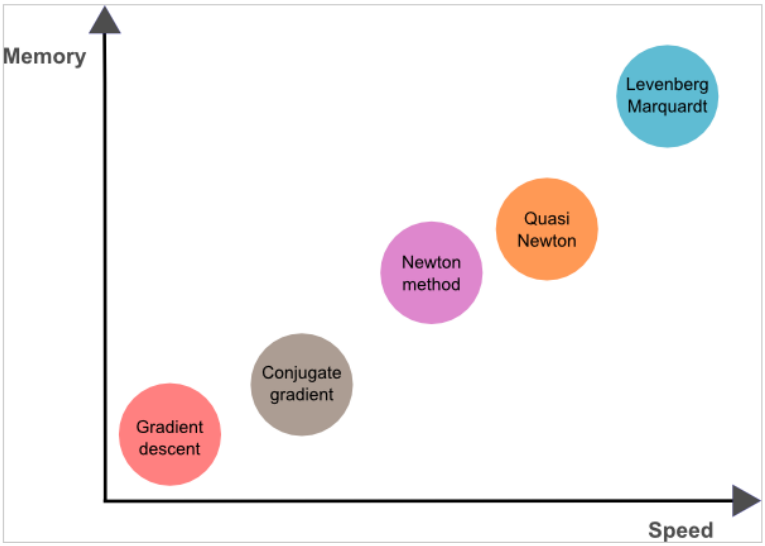



*Ilustración 13. Comparación memoria y velocidad*
*http://www.cs.us.es/~fsancho/?e=165*

Las redes neuronales multicapa adquirieron gran importancia en los años 80 como una solución de aprendizaje automático al encontrarse aplicaciones interesantes como el reconocimiento de voz o reconocimiento de imágenes [12] , pero se encontró con una competencia de una implementación más simple como son las máquinas de vectores de soporte (*support vector machines*) que estudiaremos a continuación.

El interés por el algoritmo de retropropagación volvió a recoger interés cuando aumentaron los trabajos relacionados con *deep learning* y que se basan en estas redes neuronales.

### 1.2.4. Support Vector Machines

Este algoritmo, también llamado abreviadamente SVM, puede ser considerado como una ampliación del Perceptrón. Es importante saber de qué forma este algoritmo obtiene el hiperplano más óptimo para una clasificación binaria, un hiperplano que se encargará de separar aquellos ejemplos que pertenecen a una u otra clase. Para entender cómo funciona es mejor mostrarlo con un ejemplo. Supongamos que tenemos un conjunto de datos con 2 características, por su facilidad de visionado, donde podemos distinguir la existencia de 2 clases. Vamos a buscar una línea que nos permita separar los dos conjuntos de datos:

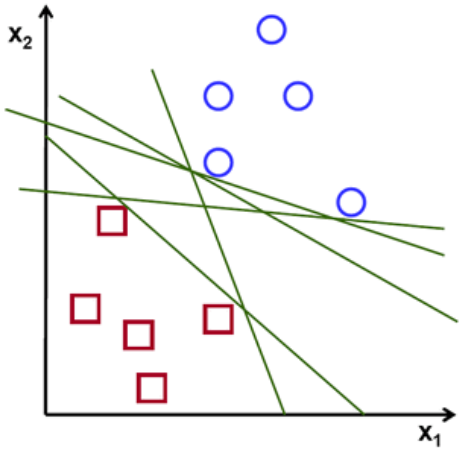



*Ilustración 14. Búsqueda de línea de separación.*
*https://docs.opencv.org/2.4/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html*

En la anterior imagen, ¿Cuál podríamos considerar que es la que mejor nos distingue las dos clases? Podríamos decir que una línea no nos interesa si pasa muy cerca de uno de los puntos ya que va a ser más sensible y no va a generalizar correctamente. Lo que queremos conseguir es una línea lo más lejos posible de todos los puntos. La misión del algoritmo SVM se basa en encontrar el hiperplano que de la distancia mínima más grande a los puntos del conjunto de entrenamiento. Esa distancia es conocida como margen y la misión del hiperplano encontrado es de maximizarlo.

Para el ejemplo anterior, la representación del hiperplano encontrado en el algoritmo SVM quedaría de la siguiente forma:

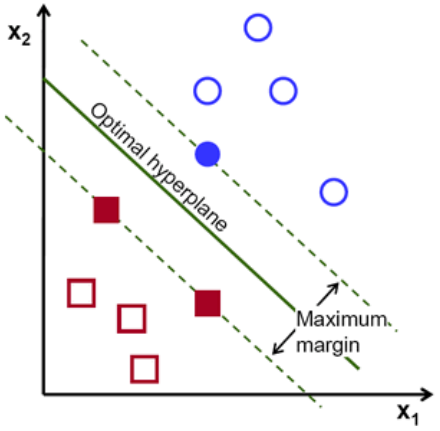



*Ilustración 15. Hiperplano que separa los dos conjuntos.*
*https://docs.opencv.org/2.4/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html*

Para encontrar este hiperplano óptimo es necesario llevar a cabo una explicación de distintos conceptos y cómo el algoritmo llega a encontrarlo.

La fórmula que define un hiperplano es la siguiente: $f(x) = \beta_0 + \beta^T x$ donde $\beta$ es el vector de pesos y $\beta_0$ son las *bias*. El hiperplano puede ser representado de varias formas con valores distintos en $\beta$ y en $\beta_0$. Para una mejor comprensión vamos a optar por usar $|\beta_0 + \beta^T x|=1$ donde

x es el vector de puntos del conjunto de entrenamiento más cercanos al hiperplano, estos son los llamados vectores de soporte o *support vector*. Para saber la distancia existente entre el hiperplano (β, $\beta_0$) y esos puntos (x) usaremos la geometría, donde tenemos que: dist = $\frac{|\beta 0 + \beta T x|}{||\beta||}$. Como la ecuación del hiperplano es igual a 1, definida previamente, la ecuación de distancia nos quedaría así: dist = $\frac{1}{|\beta||}$. Como indicamos previamente el margen que buscamos, denotado como M, es el doble de la distancia al ejemplo más cercano, ya que queremos que esté lo suficientemente separado de estos vectores de soporte, así tenemos : M=$\frac{2}{||\beta||}$. Finalmente el problema de maximizar M es equivalente al problema de minimizar L(β) sujeto a varias restricciones. Estas restricciones son las que van a modelar el requisito para que el hiperplano clasifique correctamente los ejemplos del conjunto de entrenamiento $x_i$ . Formalmente podemos decir:

$$\min_{\beta 1 \beta 0} L(\beta) = \frac{1}{2}||\beta||^2 \; subject \; to \; y_i(\beta^T x_i + \beta 0) \geq 1 \; \forall i$$

donde en $y_i$ tenemos las etiquetas del conjunto de entrenamiento.

Este tipo de problema es conocido como optimización lagrangiana y que puede ser resuelto usando los multiplicadores de Lagrange para obtener el vector peso.

Las restricciones modelan el requisito para que el hiperplano clasifique correctamente todos los ejemplos de entrenamiento β y las bias $\beta_0$ del hiperplano óptimo [42].

Pero, ¿qué ocurre si los datos con los que tratamos no son linealmente separables?. Supongamos que tenemos el siguiente conjunto de datos donde queremos extraer el hiperplano óptimo:

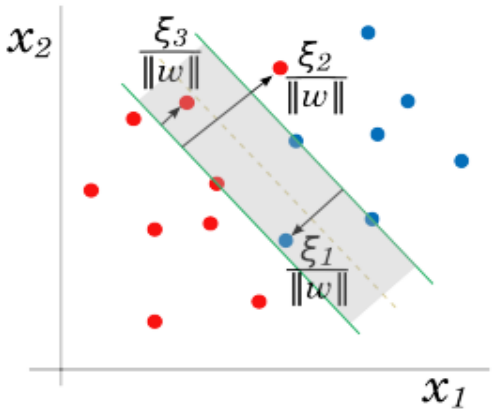



*Ilustración 16. Tolerancia a fallos en el hiperplano.*
*http://efavdb.com/svm-classification/#mjx-eqn-problem*

Para los datos que no son linealmente separables lo que hacemos es rebajar las restricciones penalizando en este caso las instancias que son clasificadas erróneamente con

el parámetro de costo C y las variables de holgura ξi que son las que definen la cantidad de puntos que están en el lado equivocado del margen.

#### Linear Support Vector Machine

Para encontrar el hiperplano que separa los dos conjuntos de datos se utiliza álgebra lineal. Una forma de ver este tipo de algoritmo es utilizar el producto interno de dos observaciones concretas, el producto escalar de dos vectores, en lugar de las mismas observaciones. Esto quiere decir que el producto escalar de dos vectores es la suma de la multiplicación de cada par de valores de entrada de cada vector. El núcleo del algoritmo de *Linear SVM* se encarga de definir la distancia entre los vectores y el nuevo dato. Cuanto más complejo sea el núcleo (como *Polynomial Kernel* o *Radial Kernel*) mejor será la clasificación ya que permiten líneas curvas que mejor pueden encajar con la distribución de los datos.

Para este tipo de algoritmo el *kernel* o núcleo es lo que llamamos producto escalar y que representamos de la siguiente forma:

$K(x,x_i) = sum(x*x_i)$, siendo x el vector del dato que estamos observando y $x_i$ el vector de soporte i.

#### Polynomial Support Vector Machine

Al igual que *Linear* pero en esta ocasión se puede usar como núcleo un polinomio, en este caso tendremos $K(x,x_i) = 1+ sum(x*x_i)^d$ donde d indica el grado del polinomio. Si d=1 entonces el núcleo sería el mismo que del tipo *Linear*. Como comentamos previamente a mayor grado permitirá el uso de un ajuste de vectores mas curvos pero esto también implica un mayor coste computacional.

### 1.2.5. Discriminant Analysis

#### Linear Discriminant Analysis

Este tipo de algoritmo se maneja fácil en aquellos casos donde las frecuencias existentes dentro de cada clase son desiguales y el rendimiento se ha evaluado atendiendo a unos datos generados aleatoriamente. Este método lo que intenta es maximizar el ratio de varianza que existe tanto para los datos que hay dentro de cada clase como la varianza que existe entre las distintas clases en cualquier conjunto de datos en particular. Con esto se asegura que se consiga la máxima separación entre clases dibujando una región de separación entre las clases dadas [8].

Las fórmulas para realizar estos cálculos derivan de modelos probabilísticos básicos encargados de sacar un modelo de la clase probable condicionada a través del dato P(X|y=k)

para cada clase k. Las predicciones se pueden obtener a través de la regla de Bayes y seleccionamos la clase k que maximize esta probabilidad condicionada. Si queremos usar este modelo como un clasificador es necesario estimar a partir del conjunto de entrenamiento cuál es la clase a priori a la que pertenece y las matrices de covarianzas.

La clasificación que se realiza en este tipo de clasificador es la creación de un modelo de probabilidad de cada clase siguiendo una distribución gausiana. En este caso tenemos que las distribuciones gaussianas de cada clase son las mismas y por lo tanto comparten mismo tipo de matriz de covarianza. De esta forma lo que conseguiremos son unas superficies de decisión lineales entre las clases. [75]

Como ejemplo de uso de este tipo de algoritmo lo podemos encontrar para el reconocimiento automático de rostros [17] o para la clasificación de documentos [66]. Linear Discriminant Analysis se encarga de extraer un conjunto de características que nos da la información más relevante para llevar a cabo la clasificación. Esto se realiza a través de un análisis de los vectores propios de matrices de dispersión con el objetivo de maximizar las variaciones entre clases y minimizar las variaciones dentro de la misma clase. De esta forma se podrán discriminar las instancias para saber si pertenecen a una clase u otra [35].

#### Quadratic Discriminant Analysis

La clasificación usando este método es muy parecida a la versión anterior *Linear* con la diferencia de que en esta ocasión no se asume cómo están las distribuciones gaussianas de las clases, y por lo tanto usa las distribuciones posteriores para estimar la clase dado un ejemplo de testeo, dando lugar a superficies de decisión de forma cuadrática. Estos parámetros gaussianos para cada clase se pueden obtener desde el conjunto de entrenamiento con una estimación de probabilidad máxima. [61]

### 1.2.6. Neighbors

#### K-Neighbors

El algoritmo *K-Nearest Neighbors*, también llamado KNN, es uno de los más populares para el reconocimiento de patrones y correlaciones. Su uso lo podemos encontrar por ejemplo en modelado financiero para prever cómo estarán los mercados [28], para detección de spam [5], o también en el campo de la medicina para saber si un paciente que ha ingresado por un infarto al corazón es probable que sufra otro atendiendo a su historial médico [30].

Este algoritmo es muy sencillo de implementar para clasificación. No asume cómo están distribuidos los datos, algo que en el mundo real así ocurre ya que la mayoría de los datos que se observan no obedecen a una distribución teórica como por ejemplo se hace para el algoritmo de regresión logística (*Logistic Regression*). Por lo tanto elegir este algoritmo puede

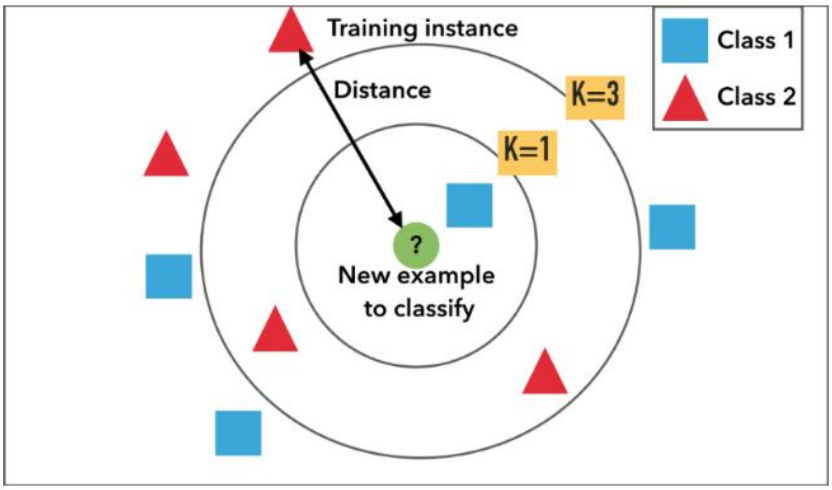
ser una buena primera idea si no tenemos conocimiento de cómo están distribuidos los datos que queremos clasificar.

Además podemos indicar que este método se considera “perezoso” en el sentido que apenas usa los datos de entrenamiento para construir un modelo generalizado y usarlo para clasificar, sino que esos datos son más bien usados en la fase de testeo. Esto se debe a que la clasificación se basa en la búsqueda de características similares con cada uno de los datos que disponemos en el conjunto de entrenamiento, y por ello necesita también mayor almacenamiento en memoria al tener que usar este conjunto en su totalidad para comparar uno por uno.

También no es paramétrico [2], es decir, que no hace suposiciones de cómo están distribuidos los datos, supone que el mejor modelo de los datos son los mismos datos.

La clasificación para cada nuevo objeto observado se basa en la asignación a la clase que obtenga mayor votación de sus *k* vecinos más cercanos. El valor *k* será el número de vecinos más cercanos con características similares para realizar la comparación y llevar a cabo la elección.

Supongamos el siguiente ejemplo para clasificar:




Example of k-NN classification. The test sample (green circle) should be classified either to the first class of blue squares or to the second class of red triangles. If k = 3 (solid line circle) it is assigned to the second class because there are 2 triangles and only 1 square inside the inner circle. If k = 5 (dashed line circle) it is assigned to the first class (3 squares vs. 2 triangles inside the outer circle).

*Ilustración 17. Ejemplo de clasificación KNN.*
*https://medium.com/@adi.bronshtein/a-quick-introduction-to-k-nearest-neighbors-algorithm-62214cea29c7*

El proceso para clasificar el nuevo ejemplo seguiría el algoritmo de k-vecinos como indica en la referencia [63].

Como ventajas podemos destacar que no necesita una adaptación para clasificación de más de dos clases y que es sencillo de implementar, por otro lado, como inconvenientes encontramos que es muy sensible a datos irrelevantes y la dimensionalidad de los mismos,

es sensible al ruido y lento en el caso que tengamos muchos datos de entrenamiento. Es adecuado indicar también la importancia de la elección de la función distancia para la elección de los vecinos al ejemplo en observación, en la que normalmente se utiliza Euclídea. Aun así existen otros métodos mejorados para la elección del vecino en este algoritmo usando la indexación del vector distancia [73]

### 1.2.7. Naive Bayes

El clasificador Naive Bayes es muy utilizado y da grandes resultados cuando se usa para clasificación y análisis de texto. [59]

Para entender cómo funciona este clasificador es necesario entender el teorema de Bayes del que hace uso. Este teorema [71,50] funciona utilizando la probabilidad condicionada o la probabilidad de que algo pase teniendo conocimiento de lo que ha ocurrido previamente. Así podremos calcular la probabilidad con la que va a ocurrir algo basándonos en la probabilidad condicional. La fórmula para calcular esta probabilidad sería:

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

Donde tenemos que [38]:

- P(*h*) es la probabilidad a priori de la hipótesis *h*
- P(D) es la probabilidad al observar el conjunto de entrenamiento D
- P(D|*h*) es la probabilidad de observar el conjunto de entrenamiento D en un universo donde se verifica la hipótesis *h*
- P(*h*|D) es la probabilidad a posteriori de h cuando se ha observado el conjunto de entrenamiento D

Para entender este teorema supongamos el siguiente ejemplo extraído de la siguiente referencia [76]:

*"Supongamos que un ingeniero está buscando agua en un terreno. A priori, se sabe que la probabilidad de que haya agua en dicha finca es del 60%. No obstante, el ingeniero quiere asegurarse mejor y decide realizar una prueba que permite detectar la presencia o no de agua. Dicha prueba tiene una fiabilidad del 90%, es decir, habiendo agua, la detecta en el 90% de los casos. También, cuando realmente no hay agua, la prueba predice que no hay agua en el 90% de los casos.*

*Por tanto, pudiendo hacer uso de dicha prueba ¿qué es más probable, que haya agua o que no?"*

Aquí lo que podemos observar es que tenemos una probabilidad de que pueda haber agua del 60% y que está condicionada por la probabilidad del 90% de que acierte la prueba. Así Thomas Bayes nos dió la solución a este problema con su teorema y la fórmula

mencionada previamente. Para la selección de la hipótesis que utiliza la técnica del Máximo a Posteriori, también conocido como MAP y que es como sigue:

$h_{MAP}$= argmax$_{h¢H}$P(*h*|*D*) = argmax$_{h¢H}\frac{P(D|h)P(h)}{P(D)}$= argmax$_{h¢H}$P(*D*|*h*)*P*(*h*) y como muchas veces las hipótesis son equiparables no es necesario multiplicar por P(*h*) y por lo tanto nos queda $h_{ML}$= argmax$_{h¢H}$P(*D*|*h*) y a este resultado es lo que se llama máxima verosimilitud (*maximum likelihood*) [38] .

Atendiendo al problema anterior de la búsqueda de agua, aquella con la probabilidad más alta será la solución al problema mencionado, es decir, la de saber si encontraremos agua en dicho terreno.

El clasificador bayesiano se va a encargar de extraer la clasificación más probable para un nuevo ejemplo dado un conjunto de entrenamiento y de donde extraerá las distintas hipótesis para pertenecer a una clase u otra, es decir, realiza el cálculo de $h_{MAP}$(x) siendo x nuestro dato a clasificar con un número de atributos o características.

Para la descripción de este algoritmo partimos con los siguientes datos: cada instancia *x* tiene un número *n* de características o atributos que podemos describir como: <$a_1$,$a_2$,$a_3$,...$a_n$>; la función objetivo f(x) puede tomar cualquier valor dentro del conjunto finito V. La clasificación para la instancia x nos la da el valor de la máxima probabilidad a posteriori : $V_{MAP}$ = argmax$_{vj¢V}$P($a_1$,$a_2$,$a_3$,...$a_n$ |$v_j$)P($v_j$).

El clasificador bayesiano supone que los atributos son independientes con respecto al objetivo que estamos buscando, o la clase a la que queremos asociar el nuevo ejemplo, por lo tanto tenemos que : P($a_1$,$a_2$,$a_3$,...$a_n$|$v_j$) = $\Pi_i$P($a_i$|$v_j$). Por último la aproximación que dará este clasificador será de la siguiente forma: $V_{naivebayes}$= argmax$_{vj¢V}$P($v_j$)$\Pi_i$P($a_i$|$v_j$).

Dependiendo del tipo de problema y datos que estemos tratando y cómo estén distribuidos los mismos existen distintos tipos de clasificación bayesiana. Así tenemos:

### BernouilliNB

Este tipo es usado para datos que que están distribuidos acorde a distribución variada de Bernouilli, es decir, pueden existir múltiples atributos en nuestros datos pero se asume que cada uno tiene una variable binaria, verdadero o falso, 1 o 0. Por lo tanto este algoritmo funciona muy bien para cuando los atributos tienen valores binarios.

### GaussianNB

Si los valores de los atributos son continuos entonces se asume que los valores de las distintas clases se distribuyen acorde a una distribución Gausiana, es decir, una Distribución Normal. Por ello el uso de este tipo sería el más adecuado en caso que tengamos esta distribución de los datos.

#### MultinomialNB

Si los valores con los que contamos están distribuidos atendiendo a distintos valores entonces es preferible usar Naive Bayes Multinomial, siendo uno de los algoritmos de clasificación estándar usado por ejemplo para clasificación o categorización de texto. Por ejemplo cada evento en la clasificación de texto representa la ocurrencia de una palabra en un documento.

### 1.2.8. Trees

#### DecisionTree

El uso de árboles de decisión lo podemos encontrar en varios campos, tales como extracción y clasificación de texto, a nivel médico para detección de cáncer, problemas del corazón o detección de diferentes enfermedades como el parkinson [41] o si un paciente debe ser hospitalizado al tener el dengue [29], o también lo podemos encontrar en los mercados de valores para saber como se va a comportar la bolsa [64].

Si hay algo que bien podemos destacar de este tipo de algoritmos y la visualización de su efectividad es por sus reglas de clasificación fácilmente comprensibles para los humanos. La idea viene a través de la conocida estructura de los árboles donde podemos encontrar una raíz y unos nodos, donde veremos sus respectivas ramas y las hojas. Un árbol de decisión empieza por el nodo raíz y se despliega hacia abajo en dos o más ramas de izquierda a derecha. El nodo donde termina la cadena es el que llamaremos nodo hoja.

La interpretación de un árbol de decisión sería de la siguiente forma: cada nodo intermedio del árbol lo vamos a considerar como un atributo o característica de los datos a tratar, siendo el nodo raíz aquel atributo más relevante, y cada hoja corresponderá con una clase o etiqueta.

En los árboles de decisión se realiza un agrupamiento de los datos basándose en los valores de los atributos de los datos que tenemos. Se realiza una división de clases atendiendo a aquel atributo que mejor distinga entre unos otros y se aplica recursivamente este proceso hasta que todos los datos de un subconjunto que se esté tratando pertenezca a la misma clase [3, 20].

El pseudocódigo de un árbol de decisión sería de la siguiente forma:

1. Situamos en el nodo raíz aquel atributo que sea más relevante para decisión.
2. Dividir el conjunto de entrenamiento en subconjuntos donde tengamos datos que contengan un mismo valor para el atributo elegido en el paso 1.
3. Repetir el paso 1 y 2 hasta que hemos encontrado todas las hojas en cada rama del árbol.

Al principio todo el conjunto de entrenamiento es considerado en la raíz y para este tipo de algoritmos es preferible que los valores de los datos sean categóricos. Para decidir qué atributo forma parte del nodo se realiza a través de un proceso de aproximación estadística. Para ello se usan diferentes técnicas para identificar cual es el atributo que debemos considerar como nodo y tomar la decisión, estos son:

- **Ganancia de información**. En esta técnica utiliza la información contenida en cada atributo [7]
- **Índice Gini**. Es una métrica para averiguar cuántas veces un elemento seleccionado aleatoriamente se identifica incorrectamente, aquel que tenga el índice más bajo será el elegido.[58]

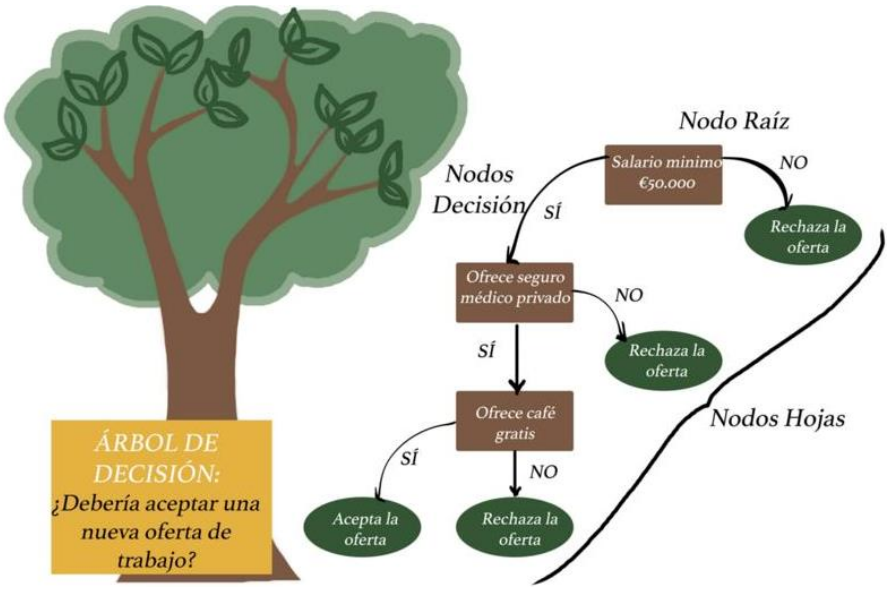



*Ilustración 18. Ejemplo de Decision Tree.*
*https://towardsdatascience.com/decision-tree-hugging-b8851f853486*

## Implementaciones de algoritmos de árboles de decisión¿Qué algoritmo usa scikit-learn?

Una vez explicada la base de cómo clasifica un algoritmo de árbol de decisión es necesario indicar que existen distintas implementaciones del mismo partiendo de la misma base. Así podemos encontrar las siguientes:

- **ID3 (*Iterative Dichotomiser 3*)**: Fué desarrollado por Ross Quinlan [47] . Se encarga de crear un árbol con muchos caminos y encuentra para cada nodo la característica categórica que obtiene mejor ganancia de todas. Estos árboles se desarrollan hasta su máximo tamaño y luego se realizan técnicas de poda para mejorar y generalizar para datos que no han sido vistos previamente, aquellos datos nuevos que queremos clasificar.

- **C4.5**: Es una evolución de ID3 quitando la restricción de que los atributos deben ser categóricos usando el concepto de información de la entropía [23]. Hace una partición de los valores continuos en distintos conjuntos de valores discretos.
- **C5.0**: Es la última versión del algoritmo de Quinlan y tiene licencia de propietario. Usa menos memoria y construye árboles más precisos que C4.5. [77]
- **CART** (*Classification and Regression Trees*): Muy similar a C4.5 pero sirve para usar con variables numéricas como objetivo por lo tanto se puede usar para regresión. Este algoritmo construye árboles binarios dejando en el nodo para decisión el atributo que tenga la mayor ganancia de información, técnica que vimos previamente.

Es necesario indicar que Scikit-learn utiliza el algoritmo CART optimizado [78]. Como ventajas de este tipo de algoritmos podemos ver que son muy fáciles de interpretar debido a la lógica que utiliza a la hora de tomar las decisiones, las posibilidades de sus hiper-parámetros son muy pocas, donde la interpretación y visualización son muy fáciles, usando la biblioteca `graphviz` [79] permite visualizar el árbol de decisión, algo que mostraremos en capítulos posteriores en el momento de tratar el conjunto de datos seleccionado. Como inconvenientes en este tipo de algoritmos podemos indicar que podremos encontrar bajo porcentaje de precisión si lo comparamos con otros métodos de aprendizaje automático en caso que los datos tengan una distribución uniforme sin grandes diferencias, es posible que exista sobreajuste cuando no se realiza poda de las ramas en las que debe parar cuando se dan condiciones que ya se han contemplado en otras ramas y por lo tanto es posible que de un mayor error.

### 1.2.8. Ensemble

El objetivo de este tipo de métodos es el de combinar las predicciones de varios clasificadores para así construir uno nuevo que sea más robusto y preciso.

Podemos encontrar dos tipos dependiendo de la técnica empleada para encontrar un nuevo clasificador y mejorar la precisión [80]:

- Métodos de refuerzo: Se van creando clasificadores de forma secuencial intentando corregir la dirección del anterior atendiendo al error cometido en sus reglas. La idea es construir un clasificador más preciso partiendo de varios con menor precisión o ajuste. De este tipo vamos a estudiar el método AdaBoost (*Adaptative Boosting*)
- Métodos de promedio: Se crean diferentes clasificadores independientes y luego se realiza un promedio de sus predicciones. Usando el promedio es mejor que si usamos la predicción de uno solo ya que la varianza se reduce. En este tipo de método vamos a estudiar RandomForest.

### AdaBoost

Este algoritmo creado por Freund y Schapire es una de las mejores opciones para

[20] árboles de decisión binarios de clasificación y es usado en numerosos campos como puede ser para la detección de intrusiones en la red [27], para la detección de objetos [70] o para clasificación de texto [56] y también para la detección de distintas enfermedades del corazón [37] . Éste método se encarga de analizar un número de algoritmos débiles, que serán árboles de decisión pequeños de profundidad 1, que tienen poca precisión a través del conjunto de entrenamiento modificado en cada iteración. Secuencialmente formará uno nuevo más preciso con la combinación de todos ellos asignando un peso en la votación final para la elección de las mejores reglas de cada uno.

La modificación de los datos en cada iteración de refuerzo consiste en aplicar pesos a cada uno de los ejemplos del conjunto de entrenamiento. Al principio estos pesos son todos iguales y en la primera iteración se realiza una clasificación de los datos tal y como están. En cada iteración los pesos de cada dato son modificados y se vuelve a realizar una clasificación con esos nuevos pesos. A los datos que no fueron modificados adecuadamente se les incrementará el peso y se realizará de manera contraria para aquellos que no lo fueron, así aquellos datos que son más difíciles de clasificar se les da más importancia para que el siguiente clasificador sencillo ponga más interés en aquellos que tienen más peso.

Para saber cómo funciona mejor este algoritmo podemos ampliar esta información en la referencia. [57]

### RandomForest

En este tipo de algoritmos de conjunto se realiza una búsqueda de un clasificador más preciso con el uso de árboles de decisión. Su uso lo podemos encontrar para clasificación del tipo de cobertura de tierra [34] . Cada árbol de decisión que se crea se hace a partir de una secuencia aleatoria de un subconjunto de datos del conjunto de entrenamiento. La diferencia con un árbol de decisión es que en el nodo donde tenemos que hacer la división de los datos no se toma aquel atributo con mayor ganancia o índice gini, sino que se coge la mejor división entre un subconjunto aleatorio de todo el conjunto de características. Esto provocará una desviación mayor del árbol pero al usar la media de varios árboles la varianza decrementa [81].

La forma en que clasifica este algoritmo es de la siguiente forma: dado un ejemplo con sus respectivas características se sitúa en cada uno de los árboles de decisión que hay en el bosque y que el algoritmo ha creado. Cada árbol dará una clasificación y ese árbol le da una votación a esa clase que ha predicho. El bosque elige la clasificación a través de aquella clase que haya sido más votada [9, 10].

La implementación que scikit-learn hace de esto es la combinación de varios clasificadores basándose en su predicción probabilística, en lugar de usar la votación que cada clasificador le dé a la clase del ejemplo clasificado.

## 1.3. La dimensión de los datos y la importancia de su reducción

La dimensión de los datos es un factor importante a tener en cuenta en el día de hoy para el almacenamiento y el procesamiento de los mismos, ya que no solo hace el entrenamiento lento sino que muchos atributos pueden desviar al algoritmo para encontrar la mejor solución. Existen distintas técnicas que nos permiten la extracción y proyección hacia un nuevo conjunto de datos con menor número de atributos y conseguir con ello métricas muy parecidas al conjunto original pero con un coste menor a nivel de computación y almacenamiento. Para utilizar estas técnicas es importante tener en cuenta que para algunos casos perderemos calidad de los datos y, aunque en algunos casos el entrenamiento sea más rápido es posible que no obtengamos la misma precisión. Es por ello necesario estudiar cómo afectan estas técnicas a los distintos algoritmos que vamos a estudiar y ver si merece la pena aplicar o no reducción en el número de atributos. Además estas técnicas no solo permiten aumentar velocidad y reducir espacio de almacenamiento, sino que además permite una mejor visualización de los datos. Por ejemplo si reducimos a 2 o 3 características podríamos visualizarlos en una gráfica y nos ayudaría a entender mejor cómo están distribuidos.

Una de estas técnicas es la extracción de características, que busca la reducción de estos datos pero con el objetivo de mantener aquellos atributos que contienen la información más relevante. Dentro de estas técnicas vamos a estudiar una muy conocida, usada y también implementada en el paquete scikit-learn: PCA (Principal Component Analysis) [82], una técnica de transformación de los datos de forma lineal que ayuda a identificar patrones basado en la correlación que existe entre atributos. PCA [25] se encarga de encontrar el hiperplano que se encuentra más cercano a los datos, y los proyecta en un nuevo espacio más pequeño que el original. Los ejes ortogonales, que son los componentes principales donde se proyectan estos nuevos datos, del nuevo sub-espacio los podemos interpretar como las direcciones de la varianza máxima sabiendo que el nuevo conjunto de características son ortogonales entre sí. Así lo podemos observar en la siguiente imagen:

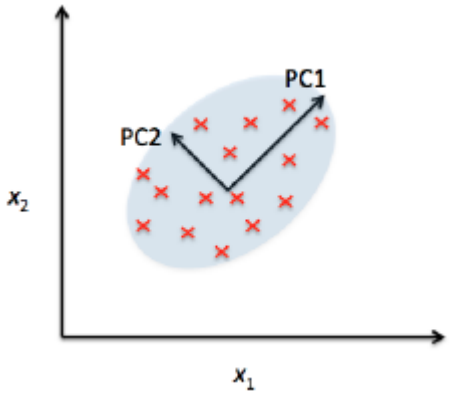



*Ilustración 19. Componentes principales diferentes*
*https://www.safaribooksonline.com/library/view/python-machine-learning/9781783555130/*

Lo que PCA quiere conseguir es proyectar un vector con una dimensión en otro con una dimensión menor. Así, en el nuevo vector del subespacio buscado, la primera componente será aquel atributo que tenga la mayor varianza posible y así sucesivamente, teniendo en cuenta que no existe correlación entre ellas, es decir, son ortogonales entre sí aquellos componentes que quedan en el nuevo vector.

Es adecuado indicar que PCA es sensible a la distribución de los datos y a la escala en la que se encuentren. Es por ello que primero hay que tener todos los datos en la misma escala para dar a todas las características la misma importancia, y luego aplicar esta técnica de reducción. Esto es algo que tendremos en cuenta cuando analizemos como es posible la reducción de la dimensión de nuestro conjunto de datos que estamos tratando.

En capítulos posteriores estudiaremos cómo afecta esta técnica a nuestro conjunto de datos y analizaremos si afecta o no a las distintas métricas que podemos obtener de los distintos algoritmos.

# Capítulo 2. El conjunto de datos a tratar

En el momento de utilizar aprendizaje automático para clasificación siempre nos vamos a encontrar con dos retos que debemos afrontar: la elección del algoritmo adecuado por un lado, y por el otro que los datos con los que estamos tratando no tengan la calidad suficiente para tratarlos, esto puede ser por ejemplo datos perdidos, poca cantidad de datos para entrenamiento o datos irrelevantes que no den información.

En este capítulo vamos a estudiar cómo debería ser la elección de los datos a entrenar y, posteriormente, cuál debe ser el algoritmo adecuado para los datos escogidos en este trabajo, aquel que mejor se adecua a su distribución atendiendo a distintas métricas que podemos extraer en el análisis que vamos a efectuar de los mismos en los próximos capítulos.

## 2.1. Conceptos a tener en cuenta

Desde pequeños hemos aprendido a clasificar e identificar distintos objetos atendiendo a sus características, formas, colores, etc. Además hemos aprendido, algo inherente para el ser humano, como por ejemplo sabemos identificar lo que es un plátano, o quizás como al nacer reconocemos la voz o el olor de la madre. Con esto queremos decir que, con una cantidad pequeña de características somos capaces de clasificar e identificar.

En el aprendizaje automático la clasificación necesita gran cantidad de datos para poder entrenar muchos de los algoritmos disponibles para poder llegar a realizar una clasificación. Muchos problemas que puedan parecer simples necesitan miles de ejemplos de entrenamiento y otros más complejos, como el reconocimiento de imágenes o de voz donde necesitamos millones de ejemplos para poder tener una clasificación de confianza.

Es importante también que los datos que tengamos en el conjunto de entrenamiento sean lo suficientemente representativos para el conjunto de datos que queremos testear, es decir, que el modelo obtenido a través del conjunto de entrenamiento nos sirva para poder clasificar adecuadamente nuevos datos que no hayan sido vistos previamente. Si el modelo que hemos entrenado sigue una estructura lineal pero el nuevo dato a clasificar no tiene ninguna relación con los datos de entrenamiento, entonces no conseguiremos una precisión adecuada para identificar o clasificar ese dato.

También es necesario tener en cuenta que los datos que tenemos para entrenar tengan una calidad aceptable, si esto no es así entonces el entrenamiento no será adecuado y el algoritmo no llegará a encontrar un modelo adecuado que nos sirva para clasificar como debe ser. Imaginemos que queremos obtener información de una encuesta online a través de un portal web, si los usuarios por ejemplo no rellenan la casilla de edad o aficiones estaríamos perdiendo una información relevante y tendríamos que decidir si ese atributo lo consideramos

o no parte del conjunto de entrenamiento. Por ello es importante que estos datos sean fiables y hayan sido cotejados o comprobados previamente antes de pasarlos a nuestro algoritmo.

Por último es importante saber que los datos con los que vamos a entrenar nuestro algoritmo tengan al mayor número de atributos que sean relevantes para extraer un modelo de ellos. Es posible que los datos tengan gran cantidad de atributos o características, pero también es cierto que algunas son más relevantes o importantes que otras, y por ello nos dan más información a la hora de crear un modelo, algunas de ellas pueden introducir ruido o datos que no interesan para obtenerlo, provocando además un mayor coste de cómputo al tener que realizar mayor cantidad de cálculos. Para esto existen técnicas como la extracción de atributos o la selección de ellos que nos permiten obtener rendimientos similares y, como consecuencia, menor coste de computación como el uso de memoria que es el más afectado cuando es necesario tratar con cantidades muy grandes de datos, por ejemplo el reconocimiento de imágenes. Una técnica para esto es, como hemos explicado en el capítulo anterior, PCA (*Principal Component Analysis*) [82] que estudiaremos en capítulos posteriores y que el paquete scikit-learn ofrece optimizado para usar a través de su API . Usando esta técnica podremos ver si afecta o no a las métricas que extraemos de los algoritmos y podremos ver si todos los atributos son relevantes a la hora de realizar el entrenamiento.

## 2.2. Elección del conjunto de datos

Existen gran cantidad de sitios donde podemos obtener conjuntos de datos de uso libre y que pueden servirnos para realizar el estudio que queramos realizar o, en concreto, el análisis que queremos llevar a cabo atendiendo al comportamiento que cada algoritmo tiene con un conjunto de datos seleccionado. Hay páginas donde se pueden encontrar repositorios de conjuntos de datos disponibles para su descarga y uso por el público en general ayudando a la búsqueda, personalización y hospedaje de los mismos que pueden requerir un desembolso económico que la mayoría no posee. De entre los repositorios para descargar conjuntos de datos fiables tenemos los siguientes ejemplos:

- Kaggle: [1] Plataforma denominada como el hogar de la ciencia de los datos donde la comunidad comparte sus trabajos de aprendizaje automático en distintos lenguajes y donde se puede encontrar gran cantidad de conjuntos de datos disponibles para su descarga y abiertos al público que quiera hacer uso de ellos.
- Conjuntos de datos de Amazon AWS: [6] A través de sus plataformas Amazon EC2, Amazon Athena, AWS Lambda y Amazon EMR tenemos disponibles datos almacenados para realizar estudios y su uso para compartir con la comunidad.
- OpenDataSoft: [43] Plataforma donde ciudades y organizaciones “inteligentes” comparten información abierta al público para que la comunidad contribuya con el uso

de la misma para afrontar los distintos retos que los datos de hoy en día nos presentan para un futuro mejor.

- UC Machine Learning Repository: [69]Centro para el aprendizaje automático y sistemas inteligentes donde se pueden encontrar conjuntos de datos disponibles para la comunidad científica donde llevar a cabo distintos estudios.

Para la elección de los datos es adecuado tener en consideración cuál será el objetivo del trabajo y qué elementos queremos tener en cuenta para el análisis a llevar a cabo. Para poder realizar este análisis hay que contar con que tenemos que considerar un conjunto de datos con un número de ejemplos significativo que nos sirva para modelar nuestros algoritmos y que nos permita realizar las operaciones pertinentes sin que conlleve un coste de computación demasiado elevado.

Debido a que la compilación y procesado de los datos descargados se realizarán en un PC con especificaciones de nivel medio vamos a intentar considerar aquellos que cumplan con las siguientes características:

- **Uso de almacenamiento moderado**. No podemos considerar aquellos que ocupen del orden de Terabytes.
- **El poder procesar los datos sin uso de GPU**. Este tipo de problemas están más enfocados para procesamiento de imágenes y que suponen un coste e infraestructura significativos para su implementación.
- **Tiempo de cómputo moderado**. Esto es interesante para la extracción de modelos a través del uso de los distintos algoritmos previamente estudiados y de donde podremos realizar el estudio y análisis de los mismos atendiendo a su comportamiento con las métricas seleccionadas.
- **Tipo de clasificación**. Podemos tener un tipo de clasificación binaria , elección entre cierto o falso, o multiclase donde tenemos 3 o más clases a clasificar. En esta ocasión nos gustaría que estuviera enfocada a multiclase para observar cómo los algoritmos utilizan la técnica *One-vs-Rest* [49] ya que muchos están enfocados a clasificación binaria y usan esta técnica para clasificaciones de más de dos clases, además su complejidad es interesante al no tener que clasificar solo entre cierto y falso.

Tras un análisis de distintos conjuntos de datos, su naturaleza, su distribución y atendiendo a que sea un problema tipo clasificación, hemos decidido finalmente la elección de un problema que se adecua y que resulta interesante para nuestro estudio. Éste es la competición para el problema de la predicción del tipo de cubierta arbórea forestal extraído de la plataforma online Kaggle [31] que sirve para clasificar categorías de bosques usando variables cartográficas.

## 2.3. ¿De donde procede?

Este conjunto de datos está almacenado en la plataforma online UCI Machine Learning Repository [69].

La idea de este tipo de competición se basa en la predicción del tipo de cobertura forestal, es decir, cual es el tipo de árbol más predominante en un área, teniendo como información variables estrictamente cartográficas. Este tipo de clasificación de cobertura forestal la determinó el área del Servicio Forestal de Estados Unidos (USFS) para cuadrículas de 30 x 30 metros. Estos datos fueron guardados tal y como se tomaron, por lo que no han sido transformados en una escala en el que todos formen parte de un mismo rango, además contienen atributos en forma binaria para variables cualitativas independientes como áreas silvestres o tipo de suelo, variables que derivaron a partir de los datos obtenidos del *US Geological Survey* y USFS.

Los datos provienen de 4 áreas silvestres en el Bosque Nacional de Roosevelt en el norte del estado de Colorado. Estas áreas representan bosques con un mínimo de cambios provocados por el hombre por lo que los tipos de estas cubiertas forestales son representativos de procesos ecológicos naturales en lugar de repoblaciones o cambios realizados por el hombre.

## 2.4. ¿Que características tiene?

Según podemos encontrar en la referencia donde está almacenado este conjunto de datos [68] podemos decir que:

- Hablamos de un conjunto de datos variado con distintos valores para cada atributo que pueden ser cuantitativos y cualitativos.
- Tenemos un total de 581.012 instancias.
- En las características de los atributos podremos encontrarlos de tipo categóricos y numéricos.
- Tenemos un total de 54 atributos o características de las cuales estudiaremos más adelante si todas ellas son relevantes.
- No existen valores perdidos y es algo que podremos comprobar en el próximo capítulo.

Todos estos datos nos servirán para llevar a cabo los ajustes de los distintos algoritmos que forman parte de nuestro estudio, de esta forma poder analizar los distintos valores que podemos extraer de todos ellos y realizar un análisis de ellos.

## 2.5. ¿Cuales son sus atributos?

Los distintos atributos de los que se compone cada ejemplo del conjunto que estamos tratando son:

- Altitud : valor de tipo cuantitativo, expresado en metros representando la elevación en metros de cada ejemplo tomado.
- Aspecto: valor de tipo cuantitativo y que expresa el aspecto en grados acimut que ofrece el ejemplo, que es un dato para cartografiar.
- Sombra : valor de tipo cuantitativo expresando los grados de pendiente que tiene el ejemplo donde se ha tomado.
- Distancia horizontal hasta hidrología: cuantitativo que expresa en metros la distancia horizontal hasta el recurso hidrológico más cercano.
- Distancia vertical hasta hidrología: cuantitativo al igual que el anterior pero en esta ocasión expresa en metros la distancia vertical.
- Distancia horizontal hasta carreteras: atributo de tipo cuantitativo que refleja la distancia horizontal en metros hasta la carretera más próxima.
- Sombra a las 9 de la mañana: de tipo cuantitativo y que indica el índice de sombra en el solsticio de verano a las 9 de la mañana, un índice que va en el rango de 0 hasta 255.
- Sombra al mediodía: al igual que el anterior pero al mediodía.
- Sombra a las 3 de la tarde: similar que el anterior pero en esta ocasión valor tomado a las 3 de la tarde.
- Distancia horizontal al punto de fuego más cercano: de tipo cuantitativo, muestra la distancia horizontal en metros a un posible punto de fuego.
- Área silvestre: 4 columnas binarias de tipo cualitativo para cada una de las áreas silvestres que se tratan en este problema. 1 indica presencia y 0 ausencia.
- Tipo de suelo: 40 columnas binarias de tipo cualitativo para indicar el tipo de suelo. 1 indica presencia y 0 ausencia dentro de 40 tipos de suelos posibles.
- Tipo de cobertura : 7 tipos de cobertura arbórea posibles para este estudio. Es de tipo entero donde podemos tener valores del 1 al 7. Estos son los distintos valores que queremos predecir y en los que se centra nuestro trabajo para la extracción de métricas. Estos 7 valores serían los distintos tipos de cobertura arbórea y son:

    1. Spruce/Fir
    2. Lodgepole Pine
    3. Ponderosa Pine
    4. Cottonwood/Willow
    5. Aspen
    6. Douglas-fir
    7. Krummholz

# Capítulo 3. Procesamiento y extracción del conjunto de datos

En este capítulo vamos a ver los distintos pasos tomados para poder llevar a cabo el análisis de los algoritmos disponibles. Además veremos cómo tratamos y procesamos los datos de los que partimos para conseguir el objetivo deseado, algo que hemos programado en lenguaje Python y que puede ser descargado en el repositorio público siguiente para su uso y testeo : https://bitbucket.org/juzaru18/trees/. De esta forma, se busca no solo comprender el funcionamiento de cada algoritmo sino poder llevar a cabo un análisis y comparación de los mismos atendiendo a distintos valores recogidos.

También mostraremos las bibliotecas para el manejo de estos datos, técnicas usadas para un ajuste más preciso de los algoritmos usados o el uso de bibliotecas para la reducción de dimensiones para así intentar conseguir un menor coste de computación.

## 3.1. Lenguaje y bibliotecas utilizadas

Como comentamos en capítulos previos, scikit-learn es una bibilioteca diseñada para su uso en el lenguaje de programación Python, en este caso se ha utilizado la versión 3.6.3. A su vez se han utilizado otros paquetes para el procesado de los datos, su visualización y tratamiento para operaciones numéricas que el mismo paquete scikit-learn necesita para su uso. Así tenemos los siguientes:

- Graphviz 0.8.1: Software de visualización de grafos de código abierto. Paquete usado para la visualización de los árboles generados por los algoritmos de árboles de decisión. [79]
- Matplotlib 2.1.1: Paquete para visualización de gráficas en Python. [83]
- Memory-profiler 0.50.0: Paquete para estimar el uso de memoria en un código de Python. [84]
- Numpy 1.13.3: Paquete para computación científica en Python. [85]
- Pandas 0.21.1: Paquete para el manejo de estructuras de datos en Python. [71]
- Scikit-learn 0.19.1: Paquete usado para aprendizaje automático en Python. [45]
- Scipy 1.0.0 : Basado en Python y usado para operaciones matemáticas, científicas y de ingeniería. [86]
- Seaborn 0.8.1: Paquete basado en matplotlib que sirve para visualización estadística de datos en Python. [87]

## 3.2. Preprocesado del conjunto de datos

A la hora de llevar a cabo un problema de clasificación es adecuado y conveniente saber con qué conjunto de datos estamos tratando, saber qué valores tiene, la distribución de estos datos, si existe correlación entre ellos, si se pueden eliminar algunos, si existen datos perdidos en el conjunto o si todos los datos son numéricos o no.

Para ello el paquete Pandas da la posibilidad de usar métodos que nos permiten visualizar el estado del conjunto de datos.

Primero vamos a mostrar de qué tipo son todos los atributos de los datos de partida, ya que debemos tener en cuenta que todos los algoritmos funcionan con datos numéricos. En caso de que fueran categóricos entonces tendríamos que codificarlos para que todos fueran numéricos.

Lo primero que hacemos es cargar el conjunto de datos desde el fichero CSV descargado desde el repositorio [68] y que hemos preparado para llevar a cabo este trabajo, para la generación del *dataframe* del paquete Pandas. Para ello ejecutamos el siguiente comando en Python: `data = pd.read_csv('trees.csv')`, siendo data la variable donde vamos a generar el *dataframe* de los datos, pd es la biblioteca pandas importada como pd, el método read_csv que permite la lectura de un archivo en formato csv, y por último el archivo donde tenemos las 581000 instancias, que en este caso es `trees.csv`.

Para comprobar de qué tipo es cada atributo que hemos cargado pandas ofrece esta posibilidad ejecutando el método dtypes asociado al conjunto que hemos cargado.

Ejecutando por lo tanto data.dtypes obtenemos la siguiente salida:

```
Elevation                           int64
Aspect                              int64
Slope                               int64
Horizontal_Distance_To_Hydrology    int64
Vertical_Distance_To_Hydrology      int64
Horizontal_Distance_To_Roadways     int64
Hillshade_9am                       int64
Hillshade_Noon                      int64
Hillshade_3pm                       int64
Horizontal_Distance_To_Fire_Points  int64
Wilderness_Area1                    int64
Wilderness_Area2                    int64
Wilderness_Area3                    int64
Wilderness_Area4                    int64
Soil_Type1                          int64
```

| | |
|---|---|
| **Soil_Type2** | **int64** |
| **Soil_Type3** | **int64** |
| **Soil_Type4** | **int64** |
| **Soil_Type5** | **int64** |
| **Soil_Type6** | **int64** |
| **Soil_Type7** | **int64** |
| **Soil_Type8** | **int64** |
| **Soil_Type9** | **int64** |
| **Soil_Type10** | **int64** |
| **Soil_Type11** | **int64** |
| **Soil_Type12** | **int64** |
| **Soil_Type13** | **int64** |
| **Soil_Type14** | **int64** |
| **Soil_Type15** | **int64** |
| **Soil_Type16** | **int64** |
| **Soil_Type17** | **int64** |
| **Soil_Type18** | **int64** |
| **Soil_Type19** | **int64** |
| **Soil_Type20** | **int64** |
| **Soil_Type21** | **int64** |
| **Soil_Type22** | **int64** |
| **Soil_Type23** | **int64** |
| **Soil_Type24** | **int64** |
| **Soil_Type25** | **int64** |
| **Soil_Type26** | **int64** |
| **Soil_Type27** | **int64** |
| **Soil_Type28** | **int64** |
| **Soil_Type29** | **int64** |
| **Soil_Type30** | **int64** |
| **Soil_Type31** | **int64** |
| **Soil_Type32** | **int64** |
| **Soil_Type33** | **int64** |
| **Soil_Type34** | **int64** |
| **Soil_Type35** | **int64** |
| **Soil_Type36** | **int64** |
| **Soil_Type37** | **int64** |

| | |
|---|---|
| **Soil_Type38** | **int64** |
| **Soil_Type39** | **int64** |
| **Soil_Type40** | **int64** |
| **Cover_Type** | **int64** |

Podemos observar que todos los datos introducidos en los atributos son numéricos y por lo tanto no es necesario hacer una codificación de los mismos para que funcionen con los posibles algoritmos de clasificación.

Por otro lado queremos saber la distribución de los datos y si contamos con datos perdidos o no suministrados. El paquete Pandas permite establecer la opción de ver una descripción de todas las columnas, donde podemos ver la cuenta total de cada una, la desviación estándar, los valores máximos y mínimos o la media de todos los valores para cada columna entre ellos. La biblioteca pandas permite establecer la opción `data.set_option('display.max_columns', None)` y ejecutando el comando `data.describe()` obtenemos dichos valores. Algunos ejemplos serían:

| | Elevation | Aspect | Slope |
|---|---|---|---|
| **count** | **581012.00000** | **581012.00000** | **581012.00000** |
| **mean** | **2959.365301** | **155.656807** | **14.103704** |
| **std** | **279.984734** | **111.913721** | **7.488242** |
| **min** | **1859.000000** | **0.000** | **0.000** |
| **25%** | **2809.000000** | **58** | **9** |
| **50%** | **2996** | **127** | **13** |
| **75%** | **3858** | **360** | **66** |
| **max** | **3858** | **360** | **66** |

La visualización de todos ellos nos demuestra que no hay datos perdidos ya que la cuenta es la misma.

Por último podemos ver que muchos de los datos no están estandarizados o no tienen la misma escala. Como describimos previamente fueron introducidos tal y como se tomaron, en bruto, y por lo tanto es adecuado que se lleve a cabo una estandarización de los mismos previo al entrenamiento de algunos algoritmos o para la el uso de técnicas de reducción de dimensión que hemos visto en capítulos anteriores. Esto es algo que veremos más adelante.

## 3.3. Estandarización y escalado de los datos

Como hemos comentado previamente, el conjunto de datos seleccionado se provee con una serie de atributos o características numéricas y que están disponibles tal y como se tomaron. Debido a esto intuimos que no existe un estándar de distribución de los mismos y por lo tanto no se ajustan a una escala de iguales valores. Así lo podemos ver en el apartado anterior tomando como ejemplo 3 de los atributos disponibles donde vemos que la elevación, aspecto o sombra toman valores en distintas escalas.

Muchos de los algoritmos de aprendizaje automático usados en el paquete scikit-learn y técnicas de reducción de la dimensión como PCA, necesitan o requieren de una normalización del conjunto de datos.

Así, por ejemplo hay muchos elementos que se usan para la función objetivo de algunos algoritmos de aprendizaje, como puede ser el núcleo de tipo gaussiano RBF para los algoritmos de las máquinas de vectores de soporte, donde asumen que todos los atributos que le pasamos al algoritmo están centrados en 0 y tienen una varianza de tipo unitaria. Esto es debido a que si tenemos un atributo con una varianza mayor que las otras entonces el algoritmo pondrá más atención a dicha característica y por lo tanto la función objetivo se centrará en buscar la más adecuada para dicho atributo. Por ello es adecuado que para este tipo de algoritmos a todos los atributos les demos la misma importancia y así poder tener más opciones que puedan dar un resultado mejor.

El paquete scikit-learn a través de su método `StandardScaler` que forma parte del módulo `sklearn.preprocessing` nos permite llevar a cabo esta técnica. Así con éste método podremos transformar el conjunto de entrenamiento para computar la media y la desviación estándar y después aplicar lo mismo al conjunto de testeo para poderlo usar en algoritmos que trabajan mejor con este tipo de distribución. Además, la técnica de reducción de dimensión PCA (Principal Component Analysis) realiza ya ésta estandarización de los datos ya que se encarga de darle a todos los atributos la misma importancia y luego seleccionar por orden cuales son más o menos relevantes. Aunque el método de scikit-learn `PCA` [82] del módulo `decomposition` ya hace esto internamente, estudiaremos en su momento como sería para nuestro conjunto de datos y así entender mejor en qué consiste.

Esto es algo que no siempre es necesario hacer, sino que, dependiendo de la naturaleza de algoritmo que estemos usando, tenemos la opción de realizar este normalizado en los datos donde podremos obtener mejores resultados.

Como ejemplo de cómo quedarían los datos tras esta estandarización tendríamos la siguiente información tras ejecutar en Python los siguientes comandos:

1. Primero realizamos una visualización de la información almacenada en nuestro conjunto de datos de entrenamiento:

```
print(X_train)
[[2946   81 7 ...,0     0     0]
 [3247  356   15 ...,   0     0     0]
 [2744  282   29 ...,   0     0     0]
 ...,
 [2587   48   15 ...,   0     0     0]
 [2666   13   14 ...,   0     0     0]
 [3259  175   14 ...,   0     0     0]]
```

Como podemos ver están en distintas escalas.

2. Importamos el módulo preprocessing donde tenemos disponible la clase StandardScaler:

   ```
   from sklearn import preprocessing
   ```

3. Aplicamos escalado estándar y amoldamos a todo el conjunto de entrenamiento :

   ```
   X_scaled=preprocessing.StandardScaler().fit(X_train)
   ```

4. Extraemos la media de cada atributo:

   ```
   X_scaled.mean_
  array([  2.95925617e+03,   1.55499892e+02,   1.41010479e+01,
         2.69516262e+02,   4.63828791e+01,   2.35051907e+03,
         2.12180960e+02,   2.23336435e+02,   1.42502169e+02,
         1.98040295e+03,   4.48995348e-01,   5.12554462e-02,
         4.36052401e-01,   6.36968046e-02,   5.14865702e-03,
         1.29011478e-02,   8.32538332e-03,   2.13396343e-02,
         2.75627723e-03,   1.14848983e-02,   1.81948720e-04,
         3.04887044e-04,   1.93996676e-03,   5.61188863e-02,
         2.14207736e-02,   5.17152355e-02,   2.99182706e-02,
         1.03759946e-03,   7.37629946e-06,   4.94457940e-03,
         5.88628697e-03,   3.26278313e-03,   6.88454616e-03,
         1.56475899e-02,   1.47771866e-03,   5.74417026e-02,
         9.97619914e-02,   3.66970898e-02,   8.58109504e-04,
         4.46511994e-03,   1.84653363e-03,   1.66458491e-03,
         1.98102816e-01,   5.21578135e-02,   4.39873324e-02,
         9.02785291e-02,   7.74978609e-02,   2.84233406e-03,
         3.29228832e-03,   1.91783786e-04,   4.99129597e-04,
         2.68890703e-02,   2.37639781e-02,   1.50574860e-02])
   ```

5. Y podemos ver la escala en la que ha quedado nuestro conjunto de datos de entrenamiento:

```
X_scaled.scale_
array([  2.79999971e+02,   1.11837491e+02,   7.49067371e+00,
         2.12568892e+02,   5.82841709e+01,   1.55848675e+03,
         2.67525547e+01,   1.97656366e+01,   3.82887517e+01,
         1.32479016e+03,   4.97391722e-01,   2.20518311e-01,
         4.95893844e-01,   2.44212042e-01,   7.15691858e-02,
         1.12848164e-01,   9.08629260e-02,   1.44513855e-01,
         5.24278568e-02,   1.06550436e-01,   1.34876097e-02,
         1.74583530e-02,   4.40023100e-02,   2.30151161e-01,
         1.44782333e-01,   2.21451507e-01,   1.70361873e-01,
         3.21950748e-02,   2.71592435e-03,   7.01436422e-02,
         7.64960038e-02,   5.70275142e-02,   8.26870557e-02,
         1.24107787e-01,   3.84126933e-02,   2.32684665e-01,
         2.99682393e-01,   1.88017056e-01,   2.92809350e-02,
         6.66722029e-02,   4.29316194e-02,   4.07653538e-02,
         3.98570057e-01,   2.22345173e-01,   2.05066933e-01,
         2.86580384e-01,   2.67379772e-01,   5.32377234e-02,
         5.72839346e-02,   1.38472743e-02,   2.23356322e-02,
         1.61759229e-01,   1.52313005e-01,   1.21781600e-01])
```

6. Y por último quedaría la transformación de todo el conjunto a dicha escala, al igual que haríamos para el conjunto de testeo:

```
X_scaled.transform(X_train)
```

Todo esto lo que hace es calcular primero la media de todos los datos y la desviación típica, seguidamente se encarga de transformar o escalar cada instancia y sus variables al rango [-1,1]. Es importante saber cómo funciona esto ya que puede llevarnos a una mala aplicación, distorsionando demasiado los datos, y por tanto repercutir en nuestro análisis.

## 3.4. Selección del modelo

### 3.4.1. Validación cruzada

Previamente vimos que para hacer un entrenamiento y validación del comportamiento de un modelo es necesario hacer una división del conjunto de datos que tenemos. Para ello vimos que el uso de la técnica de *cross validation* (validación cruzada) nos permitíria hacer este análisis.

De las distintas clases disponibles hemos usado para nuestro proyecto la de StratifiedShuffleSplit explicada en el capítulo 1. Con esta técnica podremos hacer un entrenamiento de los algoritmos donde encontremos un modelo general a la hora de clasificar un nuevo dato no visto previamente.

Para poder realizar esta validación cruzada necesitamos dividir el conjunto de datos inicial en dos: uno con los datos y sus atributos y otro donde tendremos las correspondientes clases que le corresponde a cada instancia. Para ello damos uso de la siguiente función:

```
def preprocess(train):
    labels = train.Cover_Type.values
    train = train.drop(['Cover_Type'], axis=1)
    return train, labels
```

La llamada a esta función la haríamos en nuestro código de la siguiente forma:

```
X, y = preprocess(train)
```

En la variable “X” tendríamos cada instancia con sus atributos sin el último campo (correspondiente a la clase de cobertura) y en la variable “y” tendríamos las clases correspondientes a cada instancia. Así tenemos:

```
print(y)
[5 5 2 ..., 3 3 3]
```

Una vez llevado a cabo este paso procedemos a dividir el conjunto de datos en entrenamiento y testeo usando la técnica de validación cruzada mencionada previamente y así procedamos a evaluar cómo se comportan nuestros algoritmos. Para llevar a cabo esto ejecutamos el siguiente código:

1. Definimos los valores para esta técnica de cross-validation, usando un 70% de los datos para entrenamiento y un 30% para testear cómo se comportaría para nuevos datos. Además indicamos que realice 10 divisiones. Así lo definimos con la siguiente línea en Python:

   ```
   sss = StratifiedShuffleSplit(10, test_size=0.3, random_state=0)
   ```

2. Aplicamos cross-validation al conjunto de datos. Con esta técnica vamos a obtener en la variable X_train el conjunto de instancias de entrenamiento, en X_test el conjunto que usaremos para testear, y_train las etiquetas correspondientes al conjunto de entrenamiento, y por último en la variable y_test las etiquetas correspondientes al conjunto de testeo. Todos estos valores serán necesarios para entrenar, validar y extraer métricas de rendimiento de los disintos algoritmos usados en este trabajo, métricas que veremos a continuación cómo podemos medirlas y cuantificarlas. Así tendríamos los siguientes comandos ejecutados en Python:

   ```
   for train_index, test_index in sss.split(X, y):
   ```

```
X_train,X_test=X.values[train_index],X.values[test_index]
y_train, y_test = y[train_index], y[test_index]
```

Podemos saber la dimensión de cada variable. Ejecutando el siguiente comando obtenemos la siguiente salida para la variable **X_train** por ejemplo:

```
X_train.shape
(406708, 54)
```

Podemos ver que tenemos 406708 instancias de entrenamiento que suponen el 70% del total (581012), y el 30% restante se usarán como instancias para testear el comportamiento del algoritmo que estemos evaluando. Con ese conjunto de validación lo que haremos es preguntarle a los algoritmos qué etiquetas corresponden a cada ejemplo según los atributos que tiene, y con el entrenamiento que ha tenido nos dirá la etiqueta que debería corresponder. Esa etiqueta ya la sabemos, pero se la hemos ocultado al algoritmo para de esa forma ver si clasifica bien o no.

### 3.4.2. Optimización de los hiper-parámetros de los algoritmos

Como vimos en capítulos anteriores, explicamos cómo llevaremos a cabo el ajuste de los parámetros que podemos proveer a cada uno de los algoritmos para conseguir el mejor rendimiento del mismo, como por ejemplo la precisión para clasificar un conjunto de datos. Así por ejemplo para el algoritmo de las máquinas de vectores de soporte podemos estudiar si conseguimos una mejor precisión para un kernel lineal o de tipo polinomial, o por ejemplo para el algoritmo de árboles de decisión, cuál es el método que mejor ajusta la elección del atributo que formará parte de cada nodo del árbol.

Para este problema hemos optado por usar la técnica *GridSearch,* encargada de realizar una búsqueda exhaustiva en todo el espacio de parámetros posibles elegidos por nosotros para cada algoritmo. La puntuación que se tiene en cuenta para este método por defecto es la precisión, métrica explicada en el capítulo 1, y, una vez finalizada la búsqueda de los hiper-parámetros, podremos saber los que producen mejor precisión para cada algoritmo.

Existe la posibilidad de pasarle varios parámetros al método GridSearchCV, algunos son opcionales pero en nuestro problema le pasaremos dos de ellos: el algoritmo que queremos evaluar y un diccionario de los distintos parámetros que le podemos pasar a dicho algoritmo con los valores posibles para cada parámetro. Así tenemos el siguiente ejemplo para el algoritmo de árboles de decisión:

```
DecisionTreeGrid = dict(max_depth=[2, 4, 6, None],
                        max_features=[10, 20, 30, 40, 50, None],
                        min_samples_split=[2, 5, 7, 10],
                        min_samples_leaf=[1, 3, 6, 8, 10],
                        min_weight_fraction_leaf=[0.0, 0.1, 0.5],
                        splitter=["best", "random"],
```

```
                    criterion=["gini", "entropy"],
                    random_state=[1, 5, 10]),
```

Aquí podemos observar en la búsqueda de parámetros donde por ejemplo comprobamos la estrategia para elegir la división de cada nodo, clave splitter, o las dos técnicas para identificar cual es el atributo que debemos considerar como nodo y tomar la decisión, clave criterion, que pueden ser el índice gini o la máxima entropía.

Así, usando el siguiente método, podremos saber cuales son los mejores parámetros para cada uno. Para ésta búsqueda usamos validación cruzada estratificada dividiendo en 2 partes (cv=2), como explicamos en el capítulo 1, restricciones de computación:

```
def grid_search(classifier, parameters):
    clf = classifier
    print("=" * 100)
    print("GridSearch with classifier %s with parameters %s" %
    (clf.__class__.__name__, parameters))
    grid = GridSearchCV(clf, param_grid=parameters, scoring='accuracy',
    cv=2, n_jobs=1)
    start = time()
    grid.fit(X_train, y_train)

    print("GridSearchCV took %.2f seconds for %d candidate parameter
    settings." % (time() - start, len(grid.cv_results_['params'])))
    print("The best parameters for classifier %s  is %s with a score of
    %0.2f" % (clf.__class__.__name__, grid.best_params_,
    grid.best_score_))
    print("=" * 100)
```

Realizando la llamada a esta función para el ejemplo de árboles de decisión obtenemos la siguiente respuesta:

**[Parallel(n_jobs=1)]: Done 34560 out of 34560 | elapsed: 468.3min finished**

**GridSearchCV took 28107.46 seconds for 17280 candidate parameter settings.**

**The best parameters for classifier DecisionTreeClassifier  is {'criterion': 'entropy', 'max_depth': None, 'max_features': 50, 'min_samples_leaf': 1, 'min_samples_split': 2, 'min_weight_fraction_leaf': 0.0, 'random_state': 5, 'splitter': 'best'} with a score of 0.91**

**====================================================================**

**script duration: 28116.762016296387 seconds**

Como se puede observar la técnica de *GridSearch* disponible en el paquete de scikit-learn nos permite acceder a los parámetros que nos dan la mejor precisión a través de su atributo `best_params`, y de donde podemos saber la precisión usando dichos parámetros con el atributo `best_score`. Al ser un método de tipo malla, a mayor número de parámetros mayor cantidad de posibilidades y por lo tanto mayor necesidad de cómputo. En los parámetros para llamar a la función `GridSearchCV` tenemos la opción de indicar el número de ejecuciones paralelas para hacer la búsqueda, el parámetro `n_jobs`, y por lo tanto obtener la respuesta en menor tiempo, pero es un parámetro que depende también del número de cores disponibles en nuestro sistema, por defecto este parámetros es 1 pero para la búsqueda en nuestro problema lo pondremos a -1 usando con ello todo los cores de nuestro equipo. Este parámetro es atractivo de considerar para cuando sea necesario en aquellos conjuntos de datos suficientemente grandes y que requieran de un coste de cómputo excesivo para un equipo de uso doméstico.

### 3.4.3. Métricas para analizar y comparar los algoritmos

De alguna forma necesitamos cuantificar el comportamiento que los algoritmos de aprendizaje automático tienen con el conjunto de datos que vamos a tratar.

El módulo metrics del paquete scikit-learn implementa distintas utilidades para medir cómo se ha comportado el algoritmo a la hora de clasificar el conjunto de datos de validación. Muchas de las métricas están pensadas para tareas de clasificación binaria, sin embargo para clasificaciones de tipo multi-clase los datos son tratados como una colección de varios problemas binarios, uno por cada clase que queremos clasificar, donde la mayoría usa la técnica de OneVsRest .

Algunas de las métricas que vamos a tratar en este trabajo y que vimos en el capítulo 1 son [74]:

- `accuracy_score`: se encarga de computar la exactitud, ya sea la fracción o el número de ejemplos que se han clasificado correctamente. Si el número de etiquetas conocidas se corresponden con el mismo número de etiquetas predecidas entonces tendremos una precisión 1.0. En esta métrica, mejor será nuestro clasificador cuanto más cerca esté del valor 1.0.
- `precission_score`: este valor marca la habilidad que ha tenido el clasificador de no marcar un ejemplo perteneciente a una clase cuando en realidad pertenece a otra. Cuanto más cercano sea este valor a 1.0 mejor será

- `recall_score`: valor que marca la habilidad que ha tenido el clasificador de evaluar todos los ejemplos en su correspondiente clase. Cuanto más cerca esté este valor de 1.0 mejor se habrá comportado.
- `confusion_matrix`: este método nos proporciona una matriz de confusión y así poder evaluar como se ha comportado nuestro clasificador. Podremos con este método mostrar de forma visual cómo se ha comportado en una tabla bidimensional donde las filas corresponden a las etiquetas conocidas y las columnas a las que el clasificador ha predecido.
- `log_loss`: este valor mide el ruido que el clasificador que estemos evaluando introduce a la hora de predecir unos valores a los que no conoce su etiqueta, sabiendo cual es dicha etiqueta. Se encarga de medir cual es la desviación que tendría nuestro clasificador para nuevos valores proporcionados que no hayan sido vistos. Cuanto más cercano sea este valor a 0 menor desviación habrá tenido y mejor se habrá comportado.

  La mayoría de los algoritmos, tras un entrenamiento previo, tienen la posibilidad de hacer una llamada al método `predict_proba` que devuelve una estimación de probabilidad para cada clase en el conjunto de testeo que obtenemos de la validación cruzada. Con ese método tendremos una matriz de probabilidades para cada clase, habiendo aplicado una clasificación *One vs Rest,* y `log_loss` indicará cuál ha sido la desviación sabiendo de antemano cuales era las verdaderas etiquetas que correspondían.

  En caso que el algoritmo no tenga el método `predict_proba` entonces se podrá usar el método `decision_matrix` para el conjunto de testeo donde da una puntuación para cada clase, usada luego por el método `log_loss`.

Estas métricas las provee el propio paquete scikit-learn pero también queremos comparar otras métricas que nos den una información del comportamiento de cada uno de los algoritmos evaluados. Entre ellas queremos saber, por ejemplo, el tiempo que necesita cada algoritmo para entrenar el conjunto de entrenamiento. Para esto hacemos uso del paquete time disponible para Python y donde calcularemos la diferencia del tiempo desde que se llamó al entrenamiento hasta que terminó. Obtendremos un valor en segundos que nos servirá para saber cuánto ha tardado para realizar su ajuste.

Por otra parte, otro valor a tener en cuenta que nos gustaría evaluar es el uso de memoria que cada algoritmo hace del sistema. Debido a que scikit-learn no provee ningún método para calcular esto nos tenemos que basar en el uso de paquetes externos que nos permitan cuantificar estos valores. Así tenemos el paquete `memory_profiler`[84] que está disponible para su descarga y uso. Nos permite saber cuánto consume cada línea de nuestro código y

de esta forma extraer dicha información. Así como ejemplo tenemos el siguiente código en python que nos permite saber cuanta memoria es consumida en el entrenamiento del algoritmo de árboles de decisión:

1. Importamos el paquete memory_profiler: `from memory_profiler import profile`
2. Indicamos la función o trozo de código del que queremos extraer el uso de memoria utilizado, la variable precision indica con cuantos decimales de precisión queremos que nos muestre el valor. Definimos la función `fitting` que llamaremos con el clasificador que queremos entrenar y el conjunto X_train y y_train con los que vamos a entrenar :

```
@profile(precision=2)
def fitting(classifier, x, y):
      classifier.fit(x, y)
```

3. Llamamos al método para que nos muestre en consola los valores. La llamada debemos hacerla de la siguiente forma: `mprof run trees.py` y obtendremos la siguiente salida:

```
Line #      Mem usage    Increment    Line Contents
================================================
335  1142.14 MiB    1142.14 MiB   @profile(precision=2)
336                               def fitting(classifier,x, y):
337  1232.20 MiB      90.06 MiB       classifier.fit(x, y)
```

Podemos observar en la llamada al entrenamiento (`fit`) se produce un incremento en el uso de memoria de 90,06 Megabytes.

Y así podremos hacer para cada algoritmo que contemplamos en nuestro trabajo.

## 3.5. Reducción de la dimensión de los datos: PCA

Como comentamos en capítulos anteriores, la dimensión de los datos es algo importante a tener en cuenta, debido a que una mayor dimensión corresponde con un almacenamiento y computación mayor, y por lo tanto es adecuado tener en consideración la posibilidad de reducir el conjunto de datos que estemos tratando. Existen técnicas que nos permiten aplicar una reducción de la dimensión consiguiendo resultados muy parecidos con las ventajas que ello supone: menor necesidad de almacenamiento y menor coste de computación consiguiendo resultados en un menor tiempo.

Ya vimos que scikit-learn a través de su módulo `decomposition` da la posibilidad de aplicar varias técnicas para reducción de la dimensión. Una de las más conocidas y aplicadas en el mundo de las ciencias es la de Principal Component Analysis, o también conocido como PCA. Esta será la técnica que apliquemos a nuestro conjunto de datos y observaremos cómo

afecta a nuestros algoritmos donde veremos si obtienen iguales o mejores resultados atendiendo a precisión, tiempo de cómputo, desviación de la clasificación, etc. Como ya indicamos previamente este método se basa en extraer a través de un conjunto de datos con una dimensión o número de atributos, un nuevo conjunto con menor número de componentes o atributos cuyos valores siguen direcciones opuestas, es decir, no existe una similitud entre ellas y por lo tanto marcan diferencias a la hora de clasificar. Esta técnica se basa en el uso de matrices de convergencia y otras optimizaciones que ya están desarrolladas a la hora de llamar al método `PCA` del módulo `decomposition` de scikit-learn. Sin embargo, para poder entender cómo funciona éste método y aplicarlo sin saber a ciencia cierta qué hace, vamos a explicar los pasos que realiza el mismo para nuestro conjunto de datos.

Para entender este algoritmo es necesario indicar los pasos que se llevan a cabo para extraer el nuevo vector con menor número de componentes [48].

1. Debido a que esta técnica es sensible a la distribución que tiene el conjunto de datos, para que todas las características tengan la misma importancia estandarizamos primero el conjunto a una escala. El método `StandardScaler` del módulo `preprocessing` nos permite ajustar esos datos a la misma escala. Así lo usamos en Python como sigue para nuestro conjunto de datos, donde tenemos en la variable X el conjunto usado:

   ```
   sc = StandardScaler()
   X_train_std = sc.fit_transform(X)
   ```

2. Construimos seguidamente la matriz de covarianza, donde tendremos una matriz de dimensiones *dxd* donde d es el número de características de nuestro conjunto de datos. Tendremos por lo tanto una matriz de dimensión 54x54. Un valor positivo de covarianza nos indica que ambos atributos incrementan o decrementan juntos, mientras que un valor negativo indican que lo hacen en direcciones opuestas : `cov_mat = np.cov(X_train_std.T)`

3. Descomponemos dicha matriz de covarianzas en dos vectores: el de vectores propios y el de valores propios. Los vectores propios representan la dirección de la máxima varianza de los componentes principales, mientras que los valores propios indican su magnitud. Esto es una transformación lineal del espacio y, el valor propio es el producto escalar de un vector propio por el factor de escala. El paquete NumPy permite la extracción de esto a través de su método `linal.eig`: `eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)`

Como queremos crear un nuevo conjunto de datos con un número menor de características vamos a seleccionar los k vectores propios con mayor varianza, sus valores

propios. Como los valores propios indican la varianza de cada vector propio, podemos ordenarlos y así extraer los k atributos con mayor varianza.

Tenemos la opción de sacar en una gráfica el ratio explicado de las varianzas de los valores propios. Ese ratio o porcentaje de dichos valores propios es simplemente el cociente entre la ese valor propio y la suma de todos los valores propios. En Python tenemos la opción de usar el método `cumsum` del módulo `NumPy`. Seguidamente podemos poner en una gráfica con matplotlib y su método step para mostrar ordenadamente cada varianza y cuánto aporta al total. Así tenemos:

```
tot = sum(eigen_vals)
var_exp = [(i / tot) for i in sorted(eigen_vals, reverse=True)]
cum_var_exp = np.cumsum(var_exp)
plt.step(range(1, 54), cum_var_exp, where='mid', label='cumulative
explained variance')
plt.bar(range(1, 54), var_exp, alpha=0.5, align='center',
label='individual explained variance')
plt.ylabel('Explained variance ratio')
plt.legend(loc='best')
plt.show()
```

Obtenemos la siguiente gráfica:

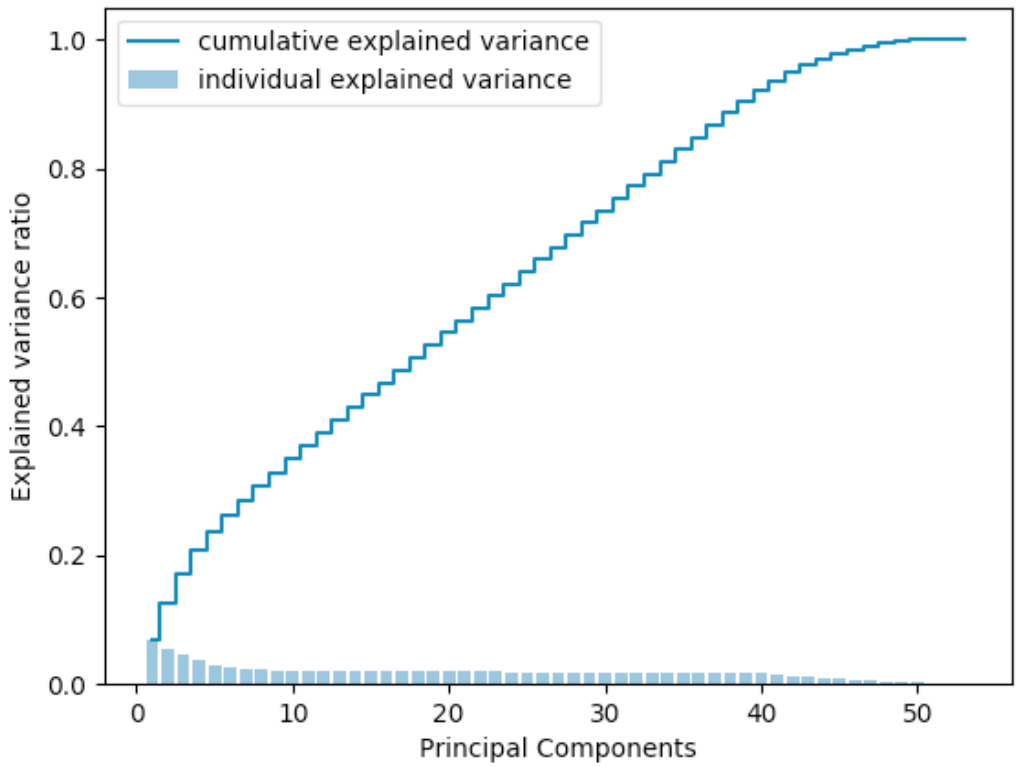



*Ilustración 20. Varianza individual y acumulada*

Aquí podemos observar el ratio de varianza explicada de los distintos atributos disponibles y cómo cada uno aporta al total de la varianza. Así, podemos ver que de 54 características 30 de ellas aportan casi el 80% de la varianza total, es decir que esas 30 son relevantes para una clasificación. Además observamos que aproximadamente 14 de ellas apenas aportan información que ayude para llevar a cabo dicha clasificación, pues serán valores poco relevantes.

Es adecuado indicar que esta técnica de Principal Component Analysis ignora las etiquetas de los atributos ya que la varianza mide la dispersión de valores a lo largo del eje de atributos.

4. El siguiente paso que tenemos que dar entonces para sacar aquellos k atributos con mayor varianza es ordenar los vectores propios atendiendo a sus valores propios:

```
eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:, i]) for i in range(len(eigen_vals))]
eigen_pairs.sort()
```

5. Seleccionamos aquellos vectores propios que tengan las k mayores varianzas. Así por ejemplo si seleccionamos 3 tendríamos en el vector la representación de cerca del 20% de la varianza total. En np tenemos la importación de paquete NumPy de Python. Tendríamos el siguiente código, donde guardamos en la variable `X_train_pca` el nuevo vector de datos con las 3 componentes principales:

```
w = np.hstack((eigen_pairs[0][1][:, np.newaxis],
                   eigen_pairs[1][1][:, np.newaxis],
                   eigen_pairs[2][1][:, np.newaxis],))
X_train_std[0].dot(w)
X_train_pca = X_train_std.dot(w)
```

Ésto sería aproximadamente el proceso que sigue el algoritmo de PCA (Principal Component Analysis) usado en scikit-learn para llevar a cabo la reducción de la dimensión del conjunto de datos. En la llamada a la constructora `PCA` indicamos en uno de sus atributos el número de componentes hasta el que queremos reducir, la variable n_components. Seguidamente se utilizará el método `fit()` y seguidamente transform() para así ajustar el conjunto de datos al nuevo con el número de componentes indicado. Así tendríamos:

```
pca = PCA(n_components=n)
pca.fit(x_train)
X_train_pca = pca.transform(x_train)
```

```
X_test_pca = pca.transform(x_test)
```

donde n es el número de componentes principales que queremos mantener en el nuevo conjunto de datos.

## 3.6. Clasificación y técnica usada para su análisis

Para el análisis y evaluación de los distintos algoritmos que se han comentado previamente hemos llevado a cabo el desarrollo de un programa en Python que nos ayude al procesamiento y ejecución de todo el conjunto de ellos, programa disponible para su descarga y uso en el siguiente repositorio público https://bitbucket.org/juzaru18/trees/.

En dicho script nos ayudamos de distintos paquetes disponibles para su uso en Python para, tanto el manejo de estructuras de datos con el paquete pandas [71], como el procesamiento de los mismos para aplicar aprendizaje automático dando uso al paquete scikit-learn, o también la visualización de datos estadísticos como puede ser el paquete Seaborn [87] o matplotlib [83] , que nos ayudará a visualizar de forma intuitiva los resultados obtenidos de los distintos algoritmos y poder compararlos.

Para tener un orden y mejor comprensión del uso del script hemos definido varias funciones que pueden ser llamadas desde la función principal del programa según los resultados que queramos obtener para nuestro conjunto de datos y algoritmos disponibles. Así se han definido funciones para la siguientes tareas adecuadas para esta tarea de análisis:

- Preprocesado de la información : Los datos son importados en formato texto desde su origen, luego son transformados a formato CSV, y finalmente con el paquete pandas los importamos en una estructura de datos, con el método `pd.read_csv`. Ya que nos vamos a basar en aprendizaje supervisado, en dicha estructura de datos nos vamos a encontrar para cada instancia tanto sus atributos como la etiqueta que le corresponde. Es por ello que dividimos el conjunto en dos estructuras de datos, uno con los atributos y otro con sus etiquetas correspondientes. Así es como hacemos en el siguiente método:

  ```
  def preprocess(train):
        labels = train.Cover_Type.values
        train = train.drop(['Cover_Type'], axis=1)
        return train, labels
  ```

  y que aplicando todo este proceso en el programa principal sería:

  ```
  train = pd.read_csv('trees.csv')
  X, y = preprocess(train)
  ```

Donde X e y son las variables donde guardaremos el conjunto con los atributos y sus etiquetas respectivamente. `trees.csv` es el archivo donde tenemos los datos almacenados en formato CSV.

- Búsqueda exhaustiva de los hiper-parámetros: para ello se hace uso de la función `grid_searchCV` que hemos indicado en el capítulo de preliminares en la elección de los parámetros de un algoritmo. Así para cada algoritmo se ha intentado buscar en un número de posibilidades acorde a lo que estamos buscando y donde hemos tenido distinta cantidad de posibilidades con diferentes tiempos de computación para ésta búsqueda exhaustiva. Por ejemplo, para la búsqueda de los hiper-parámetros del clasificador *MultiLayerPerceptron* (o Perceptrón multicapa) hemos obtenido los siguientes resultados:

  **GridSearch with classifier MLPClassifier with parameters {'hidden_layer_sizes': (4, 5, 7), 'activation': ['identity', 'logistic', 'tanh', 'relu'], 'solver': ['lbfgs', 'sgd', 'adam'], 'alpha': [0.001, 0.01, 0.1], 'batch_size': [100, 200], 'learning_rate': ['constant', 'invscaling', 'adaptive']}**

  **Fitting 2 folds for each of 648 candidates, totalling 1296 fits**

  **[Parallel(n_jobs=2)]: Done 37 tasks | elapsed: 22.7min**

  **[Parallel(n_jobs=2)]: Done 158 tasks | elapsed: 76.6min**

  **[Parallel(n_jobs=2)]: Done 361 tasks | elapsed: 150.4min**

  **[Parallel(n_jobs=2)]: Done 644 tasks | elapsed: 203.0min**

  **[Parallel(n_jobs=2)]: Done 1009 tasks | elapsed: 265.9min**

  **[Parallel(n_jobs=2)]: Done 1296 out of 1296 | elapsed: 330.3min finished**

  **GridSearchCV took 19898.19 seconds for 648 candidate parameter settings.**

  **The best parameters for classifier MLPClassifier is {'activation': 'relu', 'alpha': 0.01, 'batch_size': 200, 'hidden_layer_sizes': 7, 'learning_rate': 'constant', 'solver': 'adam'} with a score of 0.71**

  Donde vemos que ha tardado 330 minutos para 648 candidatos posibles y teniendo que realizar 1296 entrenamientos, uno por cada iteración pues *gridsearch* usa 2-fold *cross validation*, y usando 2 cores de nuestro procesador para un procesamiento paralelo.

- Aplicación de *cross validation* para el conjunto de datos: Para aplicar *cross validation* a nuestro conjunto de datos y entrenar nuestros algoritmos con esta distribución de los datos realizamos los siguientes pasos:

- Definimos el tipo de cross validation que queremos hacer donde definimos el número de pliegues (10) y cuanta cantidad de datos vamos a usar para el testeo, en este caso un 30% (test_size=0.3), el 70% se usará para entrenamiento:

  ```
  sss = StratifiedShuffleSplit(10, test_size=0.3,random_state=0)
  ```

- Dividimos el conjunto de datos aplicando el siguiente código:

  ```
  for train_index, test_index in sss.split(X, y):
        X_train,
        X_test=X.values[train_index], X.values[test_index]
        y_train, y_test = y[train_index], y[test_index]
  ```

  Donde tenemos en `X_train` el conjunto de entrenamiento con los atributos, `X_test` el conjunto de testeo con sus atributos, `y_train` las etiquetas correspondientes al conjunto de entrenamiento y por último en la variable `y_test` tendremos las etiquetas correspondientes al conjunto de testeo. Todas estas variables y sus datos se utilizarán a posteriori para entrenar y evaluar los algoritmos seleccionados para clasificar este conjunto de datos.

- Entrenamiento de los algoritmos: para llevar a cabo el entrenamiento y extracción de los datos relevantes para el análisis se ha definido una función para realizar todo este proceso y facilitar los datos. Así tenemos:

  ```
  def compare_classifiers(classifiers, X_train, y_train, X_test, y_test, conf_matrix=False):
  ```

  Donde le pasamos en el argumento classifiers una lista con los clasificadores que queremos evaluar y en los que iremos iterando para ir entrenando. También le pasamos las variables `X_train`, `y_train`, `X_test` y `y_test` que extraímos previamente para *cross validation*. Por último, con el argumento `conf_matrix` le indicamos si queremos que muestre la matriz de confusión de cada algoritmo para mostrarnos visualmente cómo ha realizado la clasificación cada algoritmo. Es un valor booleano que por defecto está a False por lo que si no lo indicamos explícitamente no mostrará dicha matriz de confusión.

  ```
  log_cols = ["Classifier", "Accuracy", "Score", "Log Loss", "Precision Score", "Recall Score", "Time", “Time to predict”]
  log = pd.DataFrame(columns=log_cols)
  ```

  Creamos una estructura de datos con varias columnas donde vamos a guardar los distintos datos que queremos analizar.

```
for clf in classifiers:
```

Iteramos para cada clasificador que tenemos en la lista pasada en el argumento y realizamos las siguientes operaciones:

```
print("Fit for classifier --> ", clf.__class__.__name__)
start = time()
clf.fit(X_train, y_train)
time_spent = (time() - start)
```

Mostramos por consola el clasificador que estamos tratando, guardamos la hora en que se ha iniciado el entrenamiento, entrenamos el clasificador, y guardamos el tiempo invertido en entrenar porque también lo queremos medir.

```
score = clf.score(X_test, y_test)
y_pred = clf.predict(X_test)
acc = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
print("Accuracy of the classifier: {:.3%}".format(acc))
print("Precision score of the classifier: {:.3%}".format(precision))
print("Score of the classifier: {:.3%}.".format(score))
class_report = classification_report(y_test, y_pred)
print ("Classification report: ")
print(class_report)
if hasattr(clf, "predict_proba"):
            train_predictions = clf.predict_proba(X_test)
else:
            conf_mat = confusion_matrix(y_true=y_test,
            y_pred=y_pred)
            train_predictions = clf.decision_function(X_test)
ll = log_loss(y_test, train_predictions)
```

En el código previo calculamos una serie de datos que explicamos en el capítulo de preliminares donde hablamos de las métricas que podemos obtener de un algoritmo, como puede ser la exactitud (*accuracy*) o *log_loss* entre otros, datos que mostramos por consola como podemos ver a continuación.

```
entry = pd.DataFrame([[clf.__class__.__name__, acc * 100, score *
100, ll, precision * 100, recall*100, time_spent,time_spent*2.3]],
columns=log_cols)
log = log.append(entry)
```

Almacenamos en la variable log los resultados que hemos obtenido de cada clasificador.

Y por último devolvemos esta tabla para posteriormente ser procesada y mostrada en gráficas que indicaremos a posteriori:

```
return log
```

Como ejemplo de salida para el algoritmo *DecisionTreeClassifier* y con los hiper-parámetro que nos ha dado la búsqueda exhaustiva obtenemos lo siguiente:

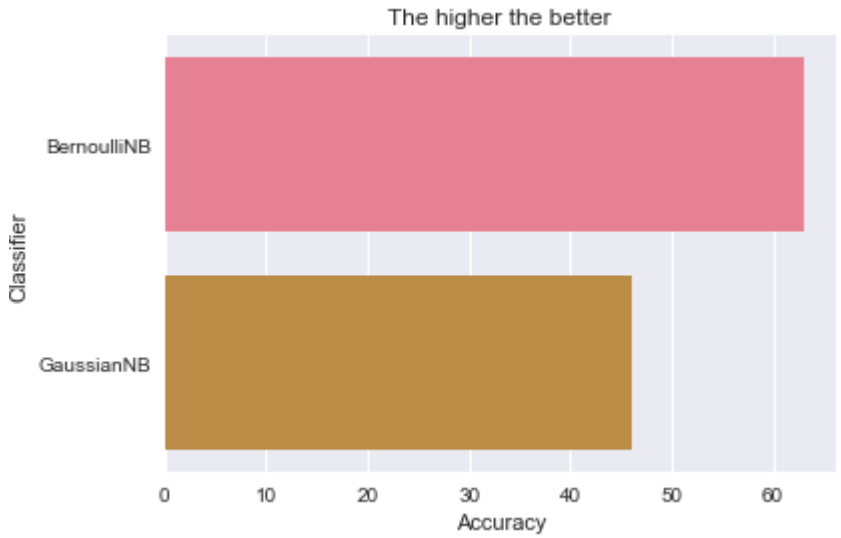
```
Fit for classifier -->  DecisionTreeClassifier
 took 6.97 seconds
==============================
DecisionTreeClassifier
***** Results *****
took 0.08 seconds to predict on test dataset
Accuracy of the classifier: 93.374%
Precision score of the classifier: 93.376%
Score of the classifier: 93.374%.
Classification report:
                precision    recall      f1-score   support

      1          0.93         0.93        0.93        63552
      2          0.94         0.94        0.94        84991
      3          0.93         0.92        0.92        10726
      4          0.85         0.82        0.83        824
      5          0.82         0.82        0.82        2848
      6          0.86         0.87        0.87        5210
      7          0.95         0.94        0.94        6153

avg / total       0.93         0.93        0.93      174304

Compute recall of the classifier: 93.374%
Log Loss: 2.2884634235865193
==============================
```

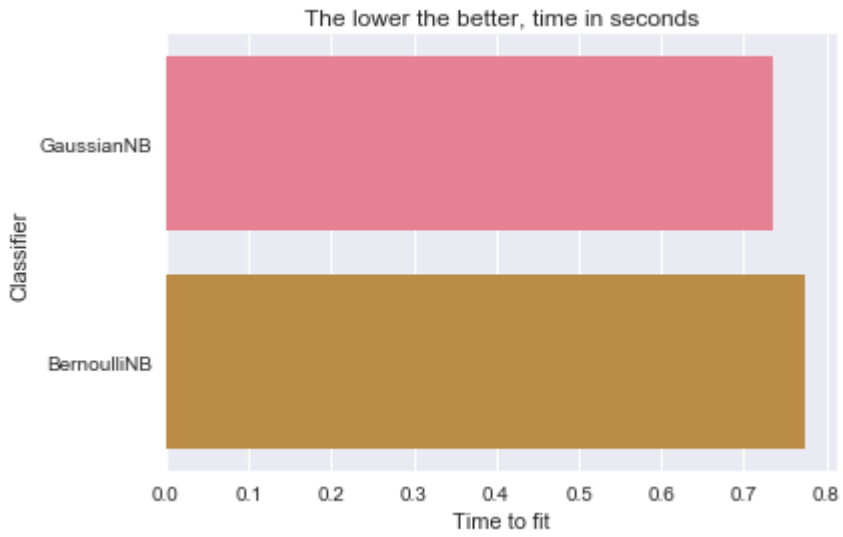
- Visualización gráfica de resultados: para mostrar de forma más visual e intuitiva los resultados que hemos obtenido de la lista de clasificadores y comparar cuál puede ser mejor damos uso del paquete Seaborn [87] que se basa en matplotlib para proveer una interfaz a alto nivel para dibujar gráficos estadísticos.

Para ello hacemos uso de la tabla que nos da el anterior método para entrenar los clasificadores. De esa forma llamamos a otro método para mostrar gráficamente la comparación de cada valor. Así tenemos:

```
def show_graphs(table):
            values_sorted = table.sort_values(["Accuracy", "Recall”],
            ascending=False)
            time_sorted = table.sort_values(["Time"],
            ascending=False)
            log_sorted = table.sort_values(["Log Loss"],
            ascending=False)
            precision_sorted = table.sort_values(["Precision Score"],
            ascending=False)
            time_predict_sorted = table.sort_values(["Time Predict"],
            ascending=False)
```

Donde generamos distintas tablas para poder ordenar los valores según criterios y así visualizarlos de una forma más intuitiva. Así por ejemplo ordenamos según *Accuracy*, tiempo empleado en el entrenamiento, *Log loss* o el tiempo que tarda en predecir los valores de testeo entre algunos. Dependiendo del tipo de variable será mejor cuanto mayor o al revés, así por ejemplo *log loss* es mejor cuanto más cerca de 0, mientras que *Accuracy* es mejor cuanto más cerca de 100.
Así tenemos el siguiente ejemplo para NaiveBayes usando kernel Bernouilli o Gaussiano:






- Reducción de dimensión de los datos: para llevar a cabo la reducción de la dimensión de los datos a través de PCA (Principal Component Analysis) y comprobar cómo afecta a nuestros algoritmos lo realizamos a través de un método. Así definimos la función siguiente:

  ```
  def pca(classifiers, x_train, y_train, x_test, y_test, n)
  ```

  En ella le pasamos como parámetros: la lista de clasificadores, las 4 estructuras de datos que hemos obtenido de *cross validation* para llevar a cabo el entrenamiento y en esta ocasión el parámetro n donde le indicamos con cuántas componentes principales queremos que se quede nuestro conjunto de datos. Procederemos después a entrenarlos y extraer de nuevo los distintos valores que hicimos previamente como *log loss*, tiempo de entrenamiento, tiempo de predicción, etc. En esta ocasión, antes de realizar el entrenamiento, deberemos ajustar tanto el conjunto de entrenamiento como el de testeo al número de componentes principales al que queremos reducir. Esto lo conseguiremos con la siguientes líneas:

  ```
  pca = PCA(n_components=n)
  pca.fit(x_train)
  X_train_pca = pca.transform(x_train)
  X_test_pca = pca.transform(x_test)
  ```

Todas estas funciones se utilizarán para llevar a cabo la extracción de los valores que queremos comparar de cada algoritmo de clasificación y así realizar un análisis de cuáles son mejores. Habiendo estudiado previamente cómo es el comportamiento de cada algoritmo nos ayudará a saber el porqué de dichos valores.

Obtendremos los datos atendiendo a las distintas clases de algoritmos que tenemos para ver que diferencia existen entre clasificadores del mismo tipo, para luego proceder a la comparación de los mejores de cada clase.

# Capítulo 4. Evaluación de los algoritmos

En esta sección mostraremos los valores obtenidos de cada algoritmo de los seleccionados y donde llevaremos a cabo una evaluación y comparación de los mismos. Debido a que para la extracción del consumo de memoria requiere una ejecución externa y anotación a mano, guardaremos los datos en CSV para posteriormente importarlos y mostrarlos con nuestro paquete de visualización Seaborn.

Mostraremos una gráfica de cada métrica que queremos comparar (*accuracy,* r*ecall score*, *log loss*, time, time to predict y *memoria*) para cada tipo de clasificador sin aplicar reducción de la dimensión de los datos y posteriormente realizaremos una comparación de los mismos usando PCA, viendo cómo afecta cuando reducimos a 20, 30 o 40 atributos. De esta forma comprobaremos si dicha reducción afecta al rendimiento de los algoritmos y el porqué de dicha variación. Los valores mostrados serán ordenados atendiendo a lo que estamos buscando (por ejemplo, para *accuracy,* mejor cuanto mayor; o *log loss,* mejor cuanto menor), para un mejor entendimiento, mostrando en el primer puesto el que da mejor resultado, dependiendo el valor que estemos comparando, así nos ayudará a identificar cual es el mejor.

Por otra parte vamos a tener también en cuenta la estandarización de los 10 primeros atributos para que estén en el rango [-1,1]. Como ya indicamos en su momento, los valores de las instancias y sus distintos atributos fueron guardados y se proporcionan tal y como se tomaron. Esto quiere decir que no todos se encuentran en la misma escala y como hemos estudiado, para muchos algoritmos es adecuado normalizarlos para que todos los atributos tengan la misma importancia a cada característica, evitando con ello que el algoritmo se desvíe en la generación del modelo con aquellas características que tienen unos valores más dispares.

Usaremos la clase `StandardScaler` del módulo de `sklearn.preprocessing` donde se realiza un centrado y escalado de cada atributo independientemente con la computación de datos estadísticos relevantes de las instancias en el conjunto que estamos tratando. `StandardScaler` se encarga de almacenar la media y la desviación estándar para posteriormente transformar los datos a esos rangos. Comprobaremos como esta estandarización afecta, para bien o para mal, a los algoritmos que estamos estudiando, información que también mostraremos en las siguientes gráficas.

La información siguiente de las tablas y gráficas nos indican aquellas métricas extraídas de la clasificación realizada por cada uno de los algoritmos usados para este trabajo. Así por ejemplo encontramos en las tablas las siguientes columnas: *Clasificador*, Ex (Exactitud), LL (Log Loss), Sens (Sensibilidad), Ta (Tiempo de ajuste), Tp (Tiempo de predicción), Mem

(Memoria usada en el ajuste en Megabytes). En aquellas casillas donde encontremos para la memoria un "-" indica que el valor obtenido para el uso de memoria es negativo y por lo tanto no lo consideramos para su comparación o no consideramos que sea un valor de confianza a tener en cuenta.

El hecho de que tengamos una exactitud mayor no indica que el algoritmo sea mejor ya que es solo útil cuando tenemos el mismo número de observaciones para cada clase y cuando las predicciones que hacemos sobre las clases tienen igual importancia. Este dato de exactitud no es una métrica adecuada cuando los datos están desbalanceados y es por ello adecuado fijarnos en otras métricas como puede ser *Log Loss* donde lo que se busca no es solo optimizar el número de aciertos sino también la seguridad con que el modelo que estamos tratando acierta y donde penalizamos los fallos cometidos, cuanto más cercano sea el valor a 0 mejor habrá sido la clasificación puesto que habrá cometido menores errores.

Otra métrica que nos ayuda a saber cómo el modelo ha clasificado es la matriz de confusión que nos da una información relevante con respecto a saber en qué clases se equivoca más el modelo o qué clase tiene más predicción. Estas matrices de confusión las mostraremos para una última comparación donde extraemos los 10 mejores modelos atendiendo a las métricas de Exactitud y Log Loss independientemente y donde veremos cuáles de ellos nos dan mejores métricas.

## 4.1. Linear Models

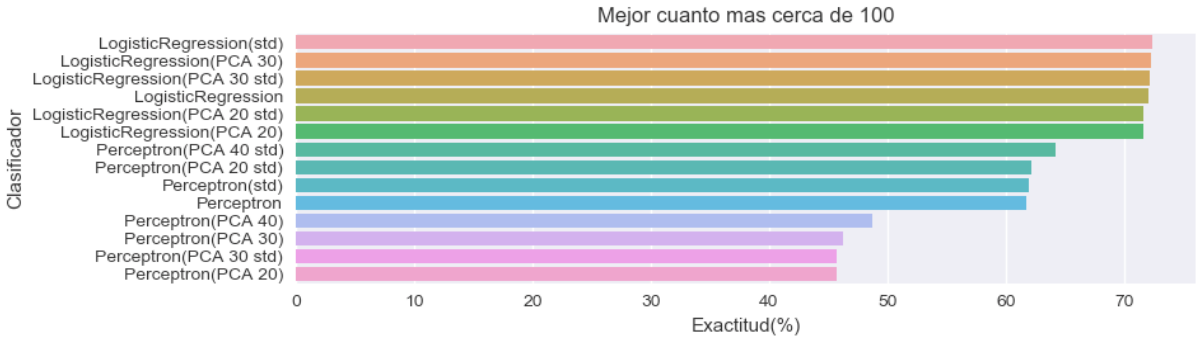
**Tabla de valores obtenidos**

| Clasificador | Ex | LL | Sens | Ta | Tp | Mem(MB) |
|---|---|---|---|---|---|---|
| **Logistic Regression** | 72,09 | 0,64 | 0,72 | 18.251,40 | 0,05 | 24,48 |
| **Perceptron** | 61,67 | 6,47 | 0,62 | 4,69 | 0,06 | 9,31 |
| **Logistic Regression (std)** | 72,39 | 0,63 | 0,72 | 869,41 | 0,03 | - |
| **Perceptron(std)** | 61,97 | 6,52 | 0,62 | 4,40 | 0,04 | 2,90 |
| **Logistic Regression (PCA 20)** | 71,56 | 0,65 | 71,56 | 7.857,61 | 0,03 | - |
| **Perceptron (PCA 20)** | 45,68 | 8,89 | 45,68 | 3,05 | 0,03 | 4,62 |
| **Logistic Regression (PCA 30)** | 72,21 | 0,64 | 72,21 | 11.935,54 | 0,04 | - |
| **Perceptron (PCA 30)** | 46,23 | 12,55 | 46,23 | 4,31 | 0,05 | 4,32 |
| **Perceptron (PCA 40)** | 48,70 | 13,46 | 48,70 | 3,82 | 0,04 | 9,62 |
| **Logistic Regression (PCA 20 std)** | 71,62 | 0,65 | 71,62 | 354,01 | 0,03 | - |

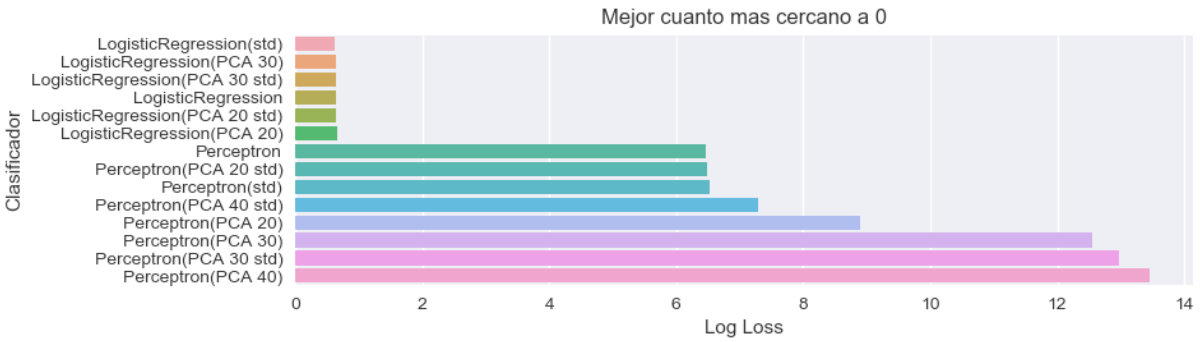
| Clasificador | Ex | LL | Sens | Ta | Tp | Mem(MB) |
|---|---|---|---|---|---|---|
| **Perceptron (PCA 20 std)** | 62,15 | 6,48 | 62,15 | 2,66 | 0,03 | 3,41 |
| **LogisticRegression (PCA 30 std)** | 72,17 | 0,64 | 72,17 | 430,00 | 0,03 | - |
| **Perceptron (PCA 30 std)** | 45,68 | 12,97 | 45,68 | 3,51 | 0,04 | 9,35 |
| **Perceptron (PCA 40 std)** | 64,18 | 7,30 | 64,18 | 3,89 | 0,03 | - |

## Gráficas

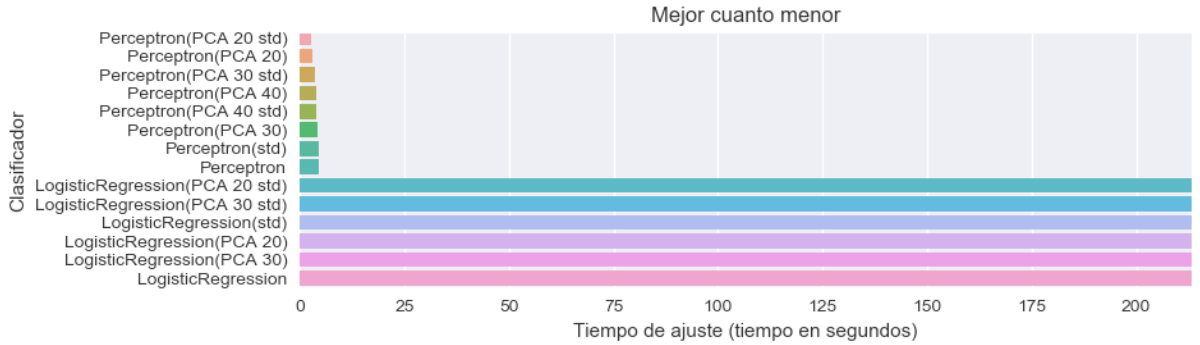
Exactitud




Log Loss




Tiempo de ajuste



## Análisis

Los hiper-parámetros con los que hemos obtenido estos valores son:

LogisticRegression

- *warm_start=True*: reusar la solución de la llamada anterior del ajuste para la siguiente inicialización.
- *fit_intercept=True*: especificamos que queremos usar una constante para ser añadida a la función de decisión.
- *solver='newton-cg'*: método a usar en regresión logística
- *multi_class='multinomial'*: la pérdida minimizada es la pérdida multinomial en toda la distribución de probabilidad.
- *max_iter=500*: número máximo de iteraciones que queremos realice el método para llegar a un ajuste. En caso que no converja se detendrá en dicho número.
- *n_jobs=2*: cantidad de cores a usar para llevar a cabo iteraciones paralelas. Para todos los algoritmos que permitían indicar este hiper-parámetro se ha indicado.

Perceptron

- *alpha=0.001*: Constante que multiplica el término de regularización si es usado.
- *fit_intercept= True*: Si la intercepción debe ser estimada o no. Si es falso, se supone que los datos ya están centrados.
- *penalty='l1'*: la penalización o término de regularización a ser usado
- *shuffle= True* : barajamos los datos en cada iteración
- *warm_start = True* : al igual que para *LogisticRegression*
- *n_jobs= 2*: usamos 2 cores del procesador para llevar a cabo iteraciones paralelas.

Para las técnicas lineales podemos observar que obtenemos una exactitud cercana al 73% para el caso de *LogisticRegression* y cercana al 65% para el Perceptrón en el mejor de los casos. Sí que es cierto que el mejor resultado lo conseguimos al estandarizar los datos pero también vemos que no se aleja demasiado si usamos PCA 40 sin necesidad de estandarizarlos.

Aplicar la técnica de reducción de atributos para el Perceptrón es algo que le afecta negativamente y que podemos ver en las últimas posiciones si atendemos a la columna de exactitud o Log Loss donde apreciamos valores cercanos a 13.

Es adecuado indicar el tiempo empleado necesario para el ajuste del algoritmo *LogisticRegression* y si lo comparamos con el Perceptrón podemos ver que necesitaremos mayor cómputo para este conjunto de datos. En la gráfica de tiempo hemos tenido que realizar una ampliación de los datos más cercanos al Perceptrón para tener una idea más cercana.

Para el ajuste de *LogisticRegression* se han necesitado más de 18.000 segundos que son cerca de 5 horas y donde conseguimos como máximo un 72% de exactitud. Sin embargo, al estandarizar dichos datos observamos como ese tiempo de ajuste cae drásticamente y donde podemos observar que reducimos el tiempo de ajuste en 17.400 segundos consiguiendo además una mejor exactitud a la hora de clasificar. El mejor tiempo lo conseguimos con un estandarizado de los datos, donde aplicando PCA 20 necesitamos 354 segundos para realizar el ajuste, penalizando sin embargo en 10 puntos menos la clasificación.

El tiempo necesario para el ajuste del Perceptrón se puede considerar casi ínfimo en comparación con LogisticRegression sabiendo que solo tiene una neurona para hacer el ajuste y por lo tanto el tiempo de computación para esto es casi inapreciable. Sin embargo observamos porcentajes de clasificación bajos y que no nos dan la confianza suficiente.

Si atendemos a los valores obtenidos en la métrica Log Loss podemos ver que conseguimos los mejores resultados aplicando LogisticRegression con valores cercanos a 0,60 en comparación con el Perceptrón que tenemos valores casi de 13. Además si estandarizamos los datos conseguimos la menor penalización y por lo tanto un modelo a tener en cuenta debido a su adecuada clasificación.

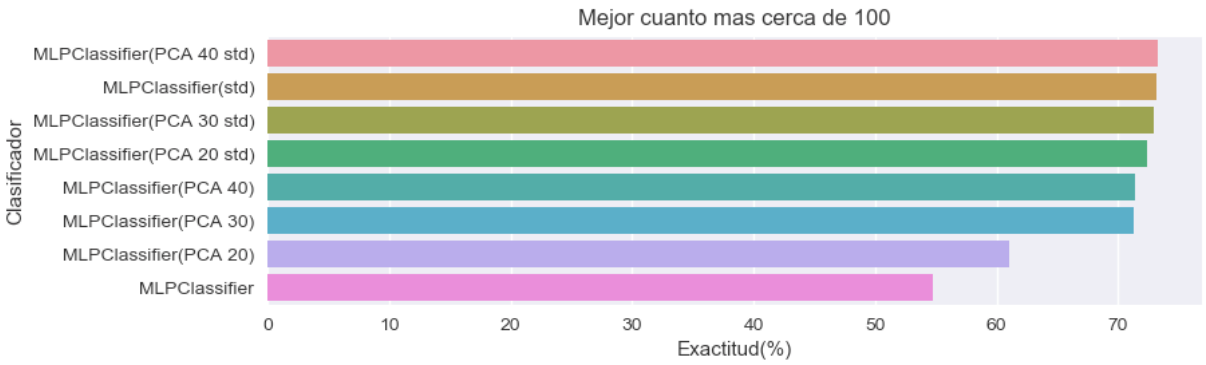
## 4.2. Neural Network

**Tabla de valores obtenidos**

| Clasificador | Ex | LL | Sens | Ta | Tp | Mem(MB) |
|---|---|---|---|---|---|---|
| **MLPClassifier** | 54,72 | 1,11 | 54,72 | 10,08 | 0,10 | 31,47 |
| **MLPClassifier (std)** | 73,12 | 0,62 | 73,12 | 8,04 | 0,08 | 14,55 |
| **MLPClassifier (PCA 20)** | 61,01 | 0,87 | 61,01 | 8,09 | 0,08 | 65,34 |
| **MLPClassifier (PCA 30)** | 71,24 | 0,70 | 71,24 | 6,74 | 0,09 | 58,21 |
| **MLPClassifier (PCA 40)** | 71,41 | 0,68 | 71,41 | 8,20 | 0,09 | 302,39 |
| **MLPClassifier (PCA 20 std)** | 72,36 | 0,64 | 72,36 | 6,73 | 0,08 | 56,09 |
| **MLPClassifier (PCA 30 std)** | 72,88 | 0,62 | 72,88 | 6,93 | 0,08 | 87,00 |
| **MLPClassifier (PCA 40 std)** | 73,23 | 0,62 | 73,23 | 7,11 | 0,09 | 308,99 |

## Gráficas

Exactitud




Log Loss


Tiempo de ajuste


## Análisis

Los hiper-parámetros con los que hemos obtenido estos valores son:

- *shuffle=True*: barajar las instancias para cada iteración.
- *hidden_layer_sizes=7*: cantidad de capas ocultas
- *validation_fraction=0.333:*La proporción de datos de entrenamiento para dejar de lado como conjunto de validación para la detención temprana.
- *solver = 'adam'*: se refiere a un optimizador estocástico basado en gradiente propuesto por Kingma, Diederik y Jimmy Ba.
- *max_iter=162*: maximo numero de iteraciones. El método escogido iterará hasta que converja pero es posible que se encuentren distintos mínimos locales y no termine.

- *batch_size=200*: tamaño de mini-lotes para optimizadores estocásticos.
- *tol=0.081977*: tolerancia para la optimización
- *activation='relu':* función de activación para las neuronas en las capas ocultas. Esta función es la función unitaria lineal rectificada que devuelve f(x) = max(0, x)
- *early_stopping=False*: Si usamos parada temprana para terminar el entrenamiento cuando la validación no está mejorando.

Para esta técnica podemos observar que conseguimos mejor exactitud cuando estandarizamos los datos y sobretodo extraemos la mejor al aplicar PCA 40 dejando solo 40 atributos para llevar a cabo la clasificación.

El tiempo de ajuste lo podemos considerar adecuado, no llegando a ser superior de los 10 segundos cuando tenemos todas las características. El mejor resultado lo encontramos cuando aplicamos PCA 20 y los datos estandarizados con cerca de 7 segundos para su ajuste consiguiendo de exactitud cerca de un 1% por debajo del mejor.

Si atendemos a los datos observados en la gráfica de Log Loss podemos observar que se reduce a más de la mitad cuando estandarizamos los mismos y que obtenemos un valor de 0,62 con respecto al 1,11 cuando estaban sin estandarizar, también se ve reflejado en la exactitud que aumenta considerablemente.

## 4.3. Discriminant Analysis

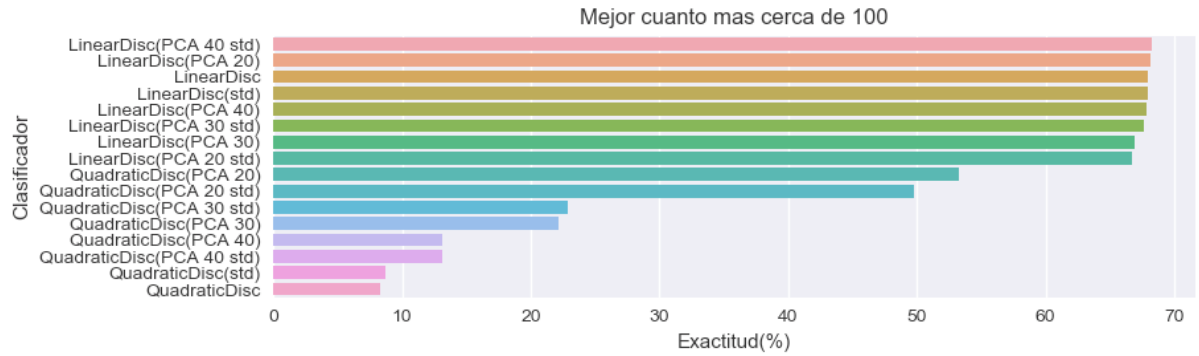
**Tabla de valores obtenidos**

| Clasificador | Ex | LL | Sens | Ta | Tp | Mem(MB) |
|---|---|---|---|---|---|---|
| **LinearDiscriminant** | 67,97 | 0,83 | 0,68 | 3,33 | 0,06 | 33,58 |
| **QuadraticDiscriminant** | 8,29 | 31,52 | 0,08 | 1,87 | 0,98 | - |
| **LinearDiscriminant (std)** | 67,97 | 0,83 | 0,68 | 3,38 | 0,04 | 187,88 |
| **QuadraticDiscriminant (std)** | 8,71 | 31,46 | 0,09 | 2,28 | 0,96 | 79,04 |
| **LinearDiscriminant (PCA 20)** | 68,13 | 0,78 | 68,13 | 0,93 | 0,03 | 29,41 |
| **QuadraticDiscriminant (PCA 20)** | 53,23 | 3,75 | 53,23 | 0,71 | 0,37 | 6,5 |
| **LinearDiscrimiant (PCA 20 std)** | 66,7 | 0,79 | 66,7 | 1,05 | 0,03 | 47,07 |
| **QuadraticDiscriminant (PCA 20 std)** | 49,8 | 4,27 | 49,8 | 0,84 | 0,41 | 6,5 |
| **LinearDiscriminant (PCA 30)** | 66,9 | 0,84 | 66,9 | 1,57 | 0,03 | 46,89 |

| Clasificador | Ex | LL | Sens | Ta | Tp | Mem(MB) |
|---|---|---|---|---|---|---|
| **QuadraticDiscriminant(PCA 30)** | 22,2 | 16,99 | 22,2 | 1,3 | 0,53 | 0,08 |
| **LinearDiscriminant (PCA 30 std)** | 67,66 | 0,82 | 67,66 | 1,8 | 0,04 | 154,94 |
| **QuadraticDiscriminant (PCA 30 std)** | 22,89 | 17,02 | 22,89 | 1,44 | 0,59 | - |
| **LinearDiscriminant (PCA 40)** | 67,81 | 0,83 | 67,81 | 2,94 | 0,06 | 472 |
| **QuadraticDiscriminant (PCA 40)** | 13,13 | 29,81 | 13,13 | 2,08 | 0,79 | 64,68 |
| **LinearDiscriminant (PCA 40 std)** | 68,23 | 0,81 | 68,23 | 2,14 | 0,04 | 30,1 |
| **QuadraticDiscriminant (PCA 40 std)** | 13,1 | 29,86 | 13,1 | 1,57 | 0,78 | 37,53 |

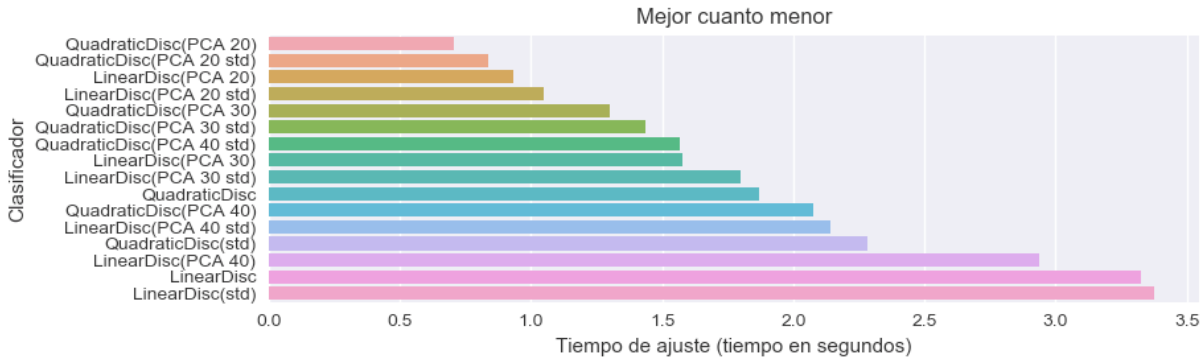
## Gráficas

Exactitud




Log Loss

Tiempo de ajuste




## Análisis

Para esta técnica podemos observar que conseguimos una exactitud en la clasificación cercana al 70% usando para el tipo LinearDiscriminantAnalysis con 40 componentes principales y los datos estandarizados. Sin embargo los peores valores los obtenemos cuando usamos QuadraticDiscriminantAnalysis. A pesar de ser un algoritmo que usa superficies de decisión de tipo cuadrático y por lo tanto más flexibles, no conseguimos unos valores muy fiables que llegan como máximo al 53% (con PCA 20) de ejemplos de validación correctamente clasificados. También es importante indicar que obtenemos una advertencia a la hora de ajustar con este algoritmo donde nos indica que las variables son colineales (**warnings.warn("Variables are collinear")**) y es por ello que conseguimos unos valores bajos. Existen estudios que ayudan a entender este problema [67] entre las variables y que posiblemente, con la eliminación de variables que den lugar a esta advertencia conseguiremos mejores resultados. Así lo podemos ver si comparamos Quadratic sin PCA y con ella. Observamos que la exactitud aumenta del 8% a cerca del 53% y además no obtenemos la advertencia de que las variables sean colineales entre ellas cuando se realiza el ajuste.

Siguiendo con el análisis de esta técnica vemos que además los valores de Log Loss no son muy altos para las técnicas LinearDiscriminantAnalysis donde vemos que no alcanzan la unidad. Si lo comparamos con los obtenidos para QuadraticDiscriminantAnalysis observamos mucha penalización alcanzando valores cercanos a 31.

Por último, observando el tiempo de ajuste de cada uno vemos que a menor número de componentes, menor tiempo empleado, algo que resulta lógico porque tiene menos parámetros que comparar y ajustar. Así vemos que el que menos tarda en ajustar es Quadratic con PCA 20 y el que más Linear con los datos estandarizados. Sin embargo estamos hablando de tiempos que no superan los 3 segundos para su entrenamiento, teniendo en cuenta que son más de 400.000 instancias.

Atendiendo a los valores obtenidos en la columna de Log Loss podemos ver que tenemos en los primeros puestos la estrategia de *LinearDiscriminantAnalysis* obteniendo el valor más

bajo con PCA20 de 0,78, si lo comparamos con los valores obtenidos atendiendo a QuadraticDiscriminantAnalysis de 31,52 podemos intuir que no es adecuado usar la estrategia de Quadratic.

Una vez analizado estos valores obtenidos, podríamos decir que la opción que mejor rendimiento tiene atendiendo a exactitud y Log Loss es LinearDiscriminantAnalysis con técnica de PCA 40.

## 4.4. Support Vector Machines

Para esta técnica no se han podido extraer datos para llevar a cabo las métricas debido a que no hemos conseguido que el ajuste termine en un tiempo considerable. Tras dejar el algoritmo realizando el ajuste del mismo (método fit()) durante 72 horas aproximadamente tuvimos que abortar la ejecución del mismo, al abortar el ajuste no es posible realizar la clasificación y por lo tanto extraer los valores que estamos tratando en el trabajo. Esto nos indica que para los datos que estamos tratando no es posible el uso de esta técnica con la capacidad de cómputo con la que contamos.

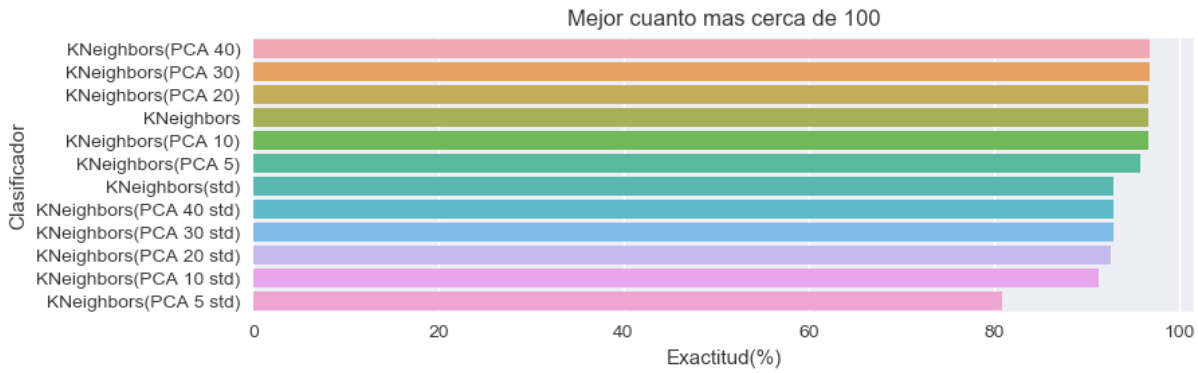
## 4.5. Neighbors

**Tabla de valores obtenidos**

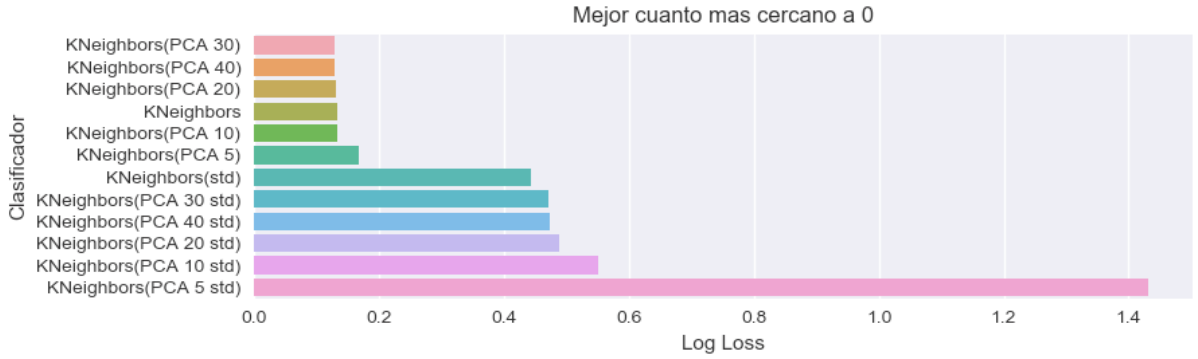
| Clasificador | Ex | LL | Sens | Ta | Tp | Mem(MB) |
|---|---|---|---|---|---|---|
| **KNeighbors** | 96,70 | 0,13 | 97 | 5,38 | 36,59 | 191,03 |
| **KNeighborsClassifier(PCA 5)** | 95,83 | 0,17 | 95,83 | 1,07 | 7,67 | 6,74 |
| **KNeighbors(PCA 10)** | 96,68 | 0,13 | 96,68 | 1,51 | 13,38 | 1,77 |
| **KNeighbors(PCA 20)** | 96,75 | 0,13 | 96,75 | 2,42 | 22,19 | 21,35 |
| **KNeighbors(PCA 30)** | 96,78 | 0,13 | 96,78 | 3,31 | 25,79 | 17,77 |
| **KNeighbors(PCA 40)** | 96,78 | 0,13 | 96,78 | 4,35 | 29,26 | 23,57 |
| **KNeighbors(std)** | 92,99 | 0,44 | 93 | 5,87 | 237,52 | 23,22 |
| **KNeighbors(PCA 5 std)** | 80,95 | 1,43 | 80,95 | 1,05 | 19,15 | 6,74 |
| **KNeighbors(PCA 10 std)** | 91,38 | 0,55 | 91,38 | 1,45 | 82,89 | 12,79 |
| **KNeighbors(PCA 20 std)** | 92,60 | 0,49 | 92,60 | 2,57 | 163,95 | 21,81 |
| **KNeighbors(PCA 30 std)** | 92,89 | 0,47 | 92,89 | 3,64 | 204,74 | 2,66 |
| **KNeighbors(PCA 40 std)** | 92,90 | 0,47 | 92,90 | 4,04 | 253,20 | 20,61 |

## Gráficas

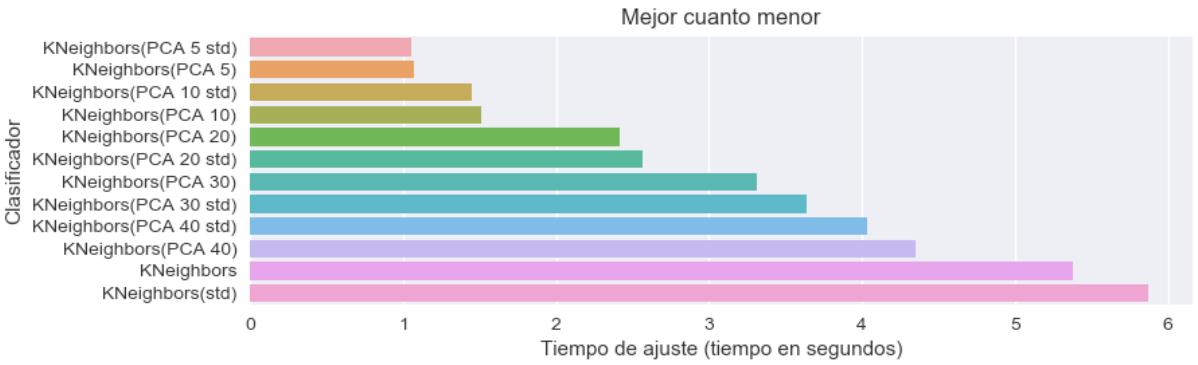
Exactitud




Log Loss




Tiempo de ajuste




## Análisis

Los hiper-parámetros con los que hemos obtenido estos valores son:

- *n_neighbors=5:* cantidad de vecinos cercanos con los que comparamos para identificar a qué clase pertenece.
- *weights='uniform'*: el peso que se le asocia a cada atributo será repartido de forma uniforme.

- *algorithm='auto'*: el algoritmo usado para el ajuste será el más apropiado basado en los datos pasados en el fit()
- *p=1*: usa la métrica de distancia Euclídea para la comparación con los vecinos más cercanos

Con los datos obtenidos para esta técnica podemos ver que es muy fiable y que además implica poco costo de computación en su uso por el poco tiempo necesario para su ajuste. Analizando en profundidad los distintos valores que tenemos podemos ver que conseguimos una exactitud cercana al 97% con PCA 40, lo que nos quiere decir que para este algoritmo no es necesario contar con todos los atributos de los que partimos si queremos obtener el valor más alto, comparado con el uso de todos los atributos existe una diferencia en exactitud de unas centésimas, pero con un tiempo de ajuste menor.

Curioso tener en cuenta cómo el mismo método usando solo 5 características principales consigue métricas bastante fiables con una exactitud del 95,83% y con un tiempo de ajuste 5 veces menor que usando todas las características, cerca de 1 segundo frente a 5 respectivamente. Esto además implica un uso menor de memoria con 6.74 Mb frente a los 191 Mb usados con todos los atributos disponibles.

Atendiendo a los valores de exactitud y Log Loss conjuntamente podemos ver que si estandarizamos el conjunto de datos obtenemos valores inferiores que si los dejamos tal y como han sido obtenidos los mismos. Esto es debido a que los valores que toman los distintos atributos le sirven al algoritmo para poder identificar con el vecino si son de la misma clase. Al estandarizar los valores estamos dando la misma importancia a todos los atributos y esto puede llevar a no identificar adecuadamente las clases. A pesar de ello conseguimos exactitudes cercanas al 90%.

Si nos centramos en la métrica de Log Loss podemos ver que en todos encontramos valores muy bajos cercanos a 0,13 para la mayoría, y vemos que son más altas cuando estandarizamos los datos.

Sí que es adecuado e interesante observar el tiempo de predicción que hemos obtenido para esta técnica. Los tiempos de ajuste soy muy cortos porque, tal y como explicamos en el capítulo 1 cuando estudiamos en profundidad cómo funcionan estos algoritmos, para Kneighbors apenas necesita entrenamiento para ajustarse, sino que una vez que tiene todos los datos en memoria, entonces realiza la predicción de cada uno de los ejemplos de testeo con respecto a sus vecinos. Es por esto que los tiempos de predicción son más altos que usando otras técnicas y además encontramos que este tiempo es mucho mayor cuando estandarizamos los datos debido, en parte, a que existen menos diferencias entre ellos y necesita más tiempo de procesamiento para clasificar cada instancia nueva.

Según los valores obtenidos, analizados y centrándonos en exactitud y Log Loss podemos decir que las mejores métricas las encontramos para el uso de esta técnica con PCA 40.

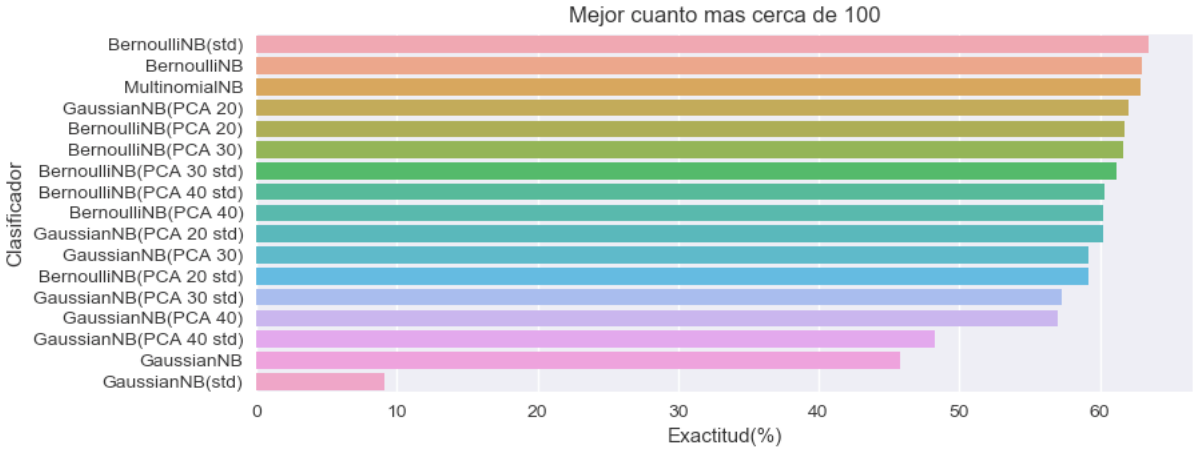
## 4.6. Naive Bayes

**Tabla de valores obtenidos**

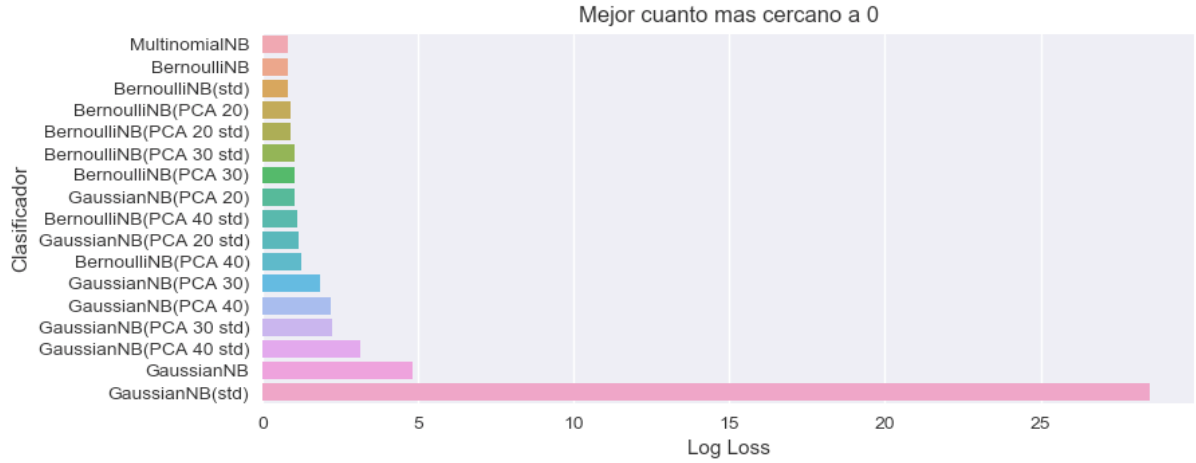
| Clasificador | Ex | LL | Sens | Ta | Tp | Mem(MB) |
|---|---|---|---|---|---|---|
| **BernoulliNB(PCA 20)** | 61,79 | 0,90 | 61,79 | 0,37 | 0,11 | - |
| **GaussianNB(PCA 20)** | 62,07 | 1,05 | 62,07 | 0,36 | 0,39 | 4,41 |
| **BernoulliNB(PCA 30)** | 61,66 | 1,02 | 61,66 | 0,47 | 0,16 | 10,29 |
| **GaussianNB(PCA 30)** | 59,21 | 1,85 | 59,21 | 0,44 | 0,60 | 92,85 |
| **BernoulliNB(PCA 40)** | 60,27 | 1,25 | 60,27 | 0,65 | 0,22 | 306,48 |
| **GaussianNB(PCA 40)** | 56,98 | 2,21 | 56,98 | 0,57 | 0,73 | 104,04 |
| **BernoulliNB(PCA 20 std)** | 59,14 | 0,92 | 59,14 | 0,34 | 0,11 | - |
| **GaussianNB(PCA 20 std)** | 60,26 | 1,15 | 60,26 | 0,34 | 0,40 | 6,44 |
| **BernoulliNB(PCA 30 std)** | 61,16 | 1,02 | 61,16 | 0,48 | 0,16 | 46,78 |
| **GaussianNB(PCA 30 std)** | 57,28 | 2,22 | 57,28 | 0,49 | 0,59 | 91,76 |
| **BernoulliNB(PCA 40 std)** | 60,32 | 1,12 | 60,32 | 0,65 | 0,19 | 306,48 |
| **GaussianNB(PCA 40 std)** | 48,24 | 3,14 | 48,24 | 0,44 | 0,73 | 132,20 |
| **BernoulliNB** | 62,96 | 0,81 | 62,96 | 0,87 | 0,16 | - |
| **GaussianNB** | 45,82 | 4,84 | 45,82 | 0,65 | 0,85 | 124,96 |
| **BernoulliNB(std)** | 63,43 | 0,83 | 63,43 | 0,69 | 0,16 | - |
| **GaussianNB(std)** | 9,12 | 28,50 | 9,12 | 0,68 | 0,81 | 122,50 |
| **MultinomialNB** | 62,90 | 0,80 | 62,90 | 0,22 | 0,04 | 48,98 |

## Gráficas

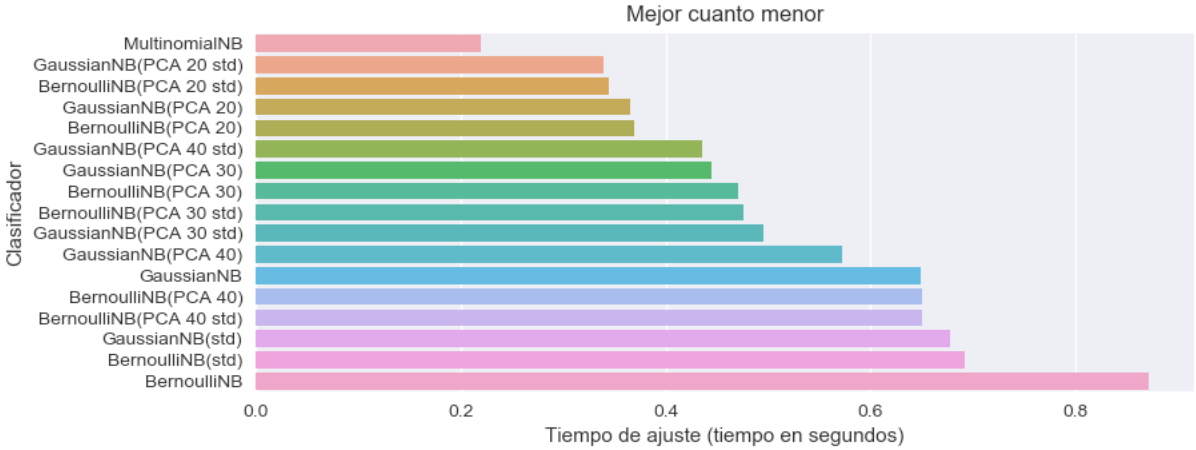
Exactitud




Log Loss




Tiempo de ajuste



### Análisis

Para estos algoritmos no se ha realizado un ajuste de sus hiper-parámetros debido a que el número de los mismos es muy bajo y el cambio de los mismos no supone una diferencia de métricas que indiquen merezca la pena contemplarlos.

Según las métricas obtenidas podemos observar que el uso de esta técnica para clasificar nuestro conjunto de datos no produce resultados que nos permitan confiar en ellos. La mejor exactitud la obtenemos aplicando el algoritmo de BernouilliNB con los datos estandarizados, pero alcanzando un valor cercano al 63%. Este algoritmo está diseñado para conjuntos de datos que tienen sus características de forma binaria, así como sabemos que tenemos 44 características de tipo binario, estas características serán relevantes para este tipo de Naive Bayes y, además, con los datos estandarizados quiere decir que los acotamos en una escala de [-1,1] siendo parecido a una distribución binaria. Así podemos ver que ocupa la mayoría de las primeras posiciones si hablamos de exactitud. Usando el tipo MultinomialNB también vemos que obtenemos resultados cercanos a BernouilliNB, aunque es más conveniente usarlo para aquellos conjuntos de datos con datos discretos de atributos, sobre todo se usa para clasificación de textos donde podemos tener un número discreto de palabras y normalmente la distribución multinomial requiere recuento de características en formato entero.

Al aplicar PCA vemos que no afecta demasiado a la clasificación que obtenemos sin aplicarla. Sin embargo, el tiempo de ajuste sí que lo vemos reducido aunque apenas es apreciable teniendo en cuenta que contamos con valores que van de 0.22 segundos para MultinomialNB hasta los 0.87 segundos con BernouilliNB.

Atendiendo a los datos de *Log Loss* podemos ver que la menor penalización la obtenemos usando el modelo obtenido para MultinomialNB donde tenemos un valor de 0.80 que nos da una confianza adecuada para indicar que puede ser un buen clasificador, a pesar que hemos obtenido una exactitud del 63%. Estos datos pueden deberse, como hemos visto en el apartado de Análisis discriminante, por la co-linealidad de los atributos.

## 4.7. Trees

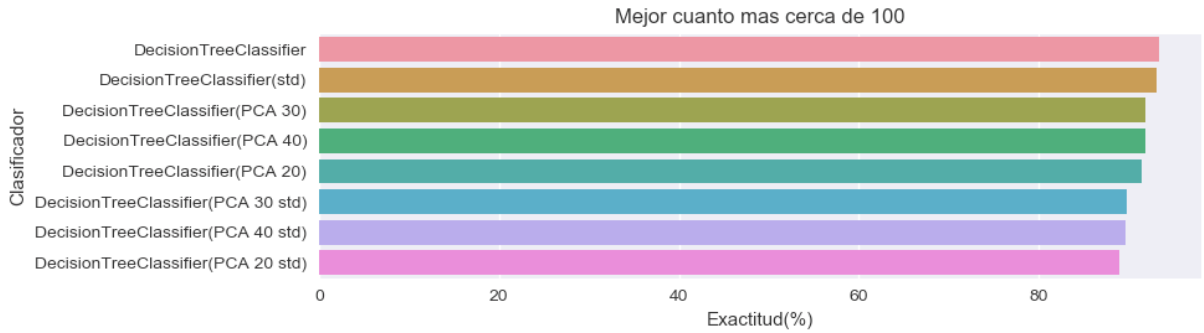
**Tabla de valores obtenidos**

| Clasificador | Ex | LL | Sens | Ta | Tp | Memoria(MB) |
|---|---|---|---|---|---|---|
| **DecisionTree** | 93,37 | 2,29 | 93,37 | 6,37 | 0,09 | 94,44 |
| **DecisionTree (std)** | 93,05 | 2,40 | 93,05 | 4,82 | 0,09 | 93,03 |
| **DecisionTree (PCA 20)** | 91,48 | 2,94 | 91,48 | 3,35 | 0,08 | 47,75 |

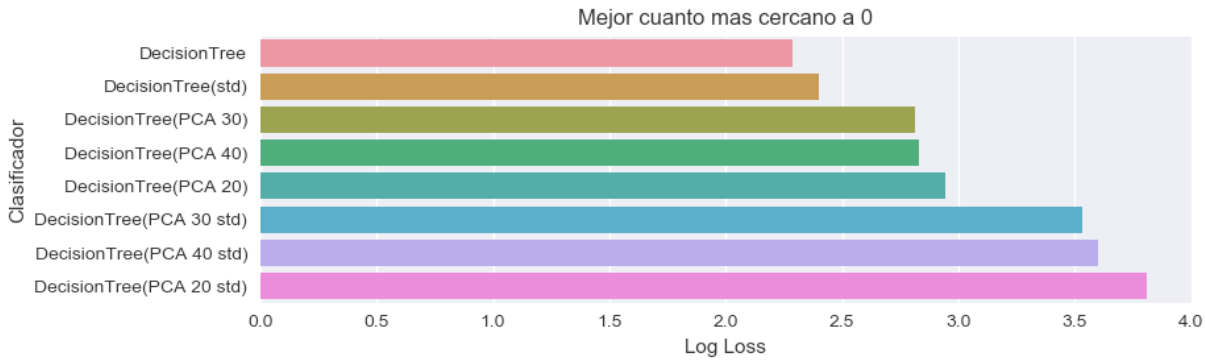
| Clasificador | Ex | LL | Sens | Ta | Tp | Memoria(MB) |
|---|---|---|---|---|---|---|
| **DecisionTree (PCA 30)** | 91,85 | 2,82 | 91,85 | 5,54 | 0,10 | 11,73 |
| **DecisionTree (PCA 40)** | 91,80 | 2,83 | 91,80 | 7,37 | 0,12 | 11,66 |
| **DecisionTree (PCA 20 std)** | 88,97 | 3,81 | 88,97 | 3,60 | 0,10 | 34,70 |
| **DecisionTree (PCA 30 std)** | 89,78 | 3,53 | 89,78 | 5,94 | 0,10 | 8,19 |
| **DecisionTree (PCA 40 std)** | 89,58 | 3,60 | 89,58 | 8,49 | 0,11 | 12,43 |

## Gráficas

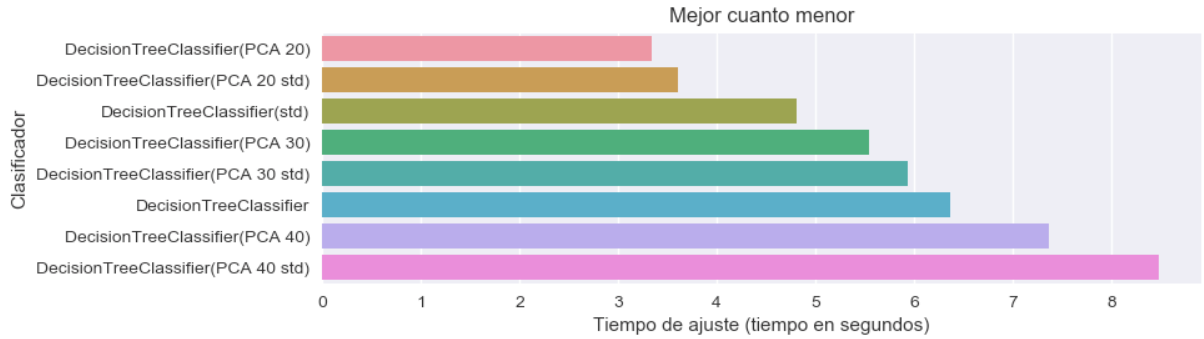
Exactitud




Log Loss




Tiempo de ajuste



### Análisis

Los hiper-parámetros con los que hemos obtenido estos valores son:

- *splitter="random"*: estrategia para dividir el árbol de forma aleatoria a la hora de elegir el atributo que irá en el nodo.
- *max_leaf_nodes*=None: número ilimitado de nodos hoja.
- *min_samples_leaf=1*: mínimo número de ejemplos requeridos para estar en el nodo hoja, en esta ocasión es el usado por defecto, 1.
- *min_samples_split=2*: número mínimo requerido de ejemplos para llevar a cabo la división de los nodos. En esta ocasión se usa el valor por defecto, 2.
- *min_weight_fraction_leaf=0.0*:
- *criterion='entropy'*: criterio usado para llevar a cabo la elección del atributo que estará en el nodo. En este caso se selecciona criterio de la máxima entropía.
- *random_state=5*: la semilla usada para la generación del número aleatorio usado para el splitter “random”.
- *max_features=None*: el máximo número de características a usar para llevar a cabo la clasificación. En este caso el valor None indica que usaremos todas
- *max_depth=None*: no se contempla una profundidad máxima para los árboles por ello se alcanzará la que requiera.

Con los datos que hemos obtenido para esta técnica podemos ver que obtenemos la mejor exactitud cuándo usamos los datos tal y como nos los proporcionan y además no se realiza un estandarizado de los mismos. Esto es debido al uso de los algoritmos para decidir qué característica forma parte del nodo, si optamos por tener los datos estandarizados la búsqueda de máxima entropía no será tan influyente y por lo tanto puede llevar a una elección equivocada en un nodo del árbol. Por otra parte observamos también que obtenemos mejores métricas usando el total de atributos con un coste de computación mayor, aunque superado cuando usamos PCA 40 o PCA 40 con los datos estandarizados.

Si atendemos a la columna de *Log Loss* podemos ver que obtenemos valores muy altos por encima de 2 en todos ellos y por lo tanto indicando una penalización alta cuando lo que buscamos es un acercamiento al valor 0. Aquí tenemos otro ejemplo de que, a pesar de haber obtenido exactitudes la mayoría por encima del 90%, si los datos están desbalanceados puede provocar que aquellos que tienen más ejemplos le den más peso al valor de la exactitud.

Es posible visualizar gráficamente cómo se desarrolla el árbol de decisión para nuestro conjunto de datos y con los hiper-parámetros elegidos. Así con el uso de la biblioteca graphviz [79] que nos permite generar un archivo jerárquico .dot después de haber realizado el ajuste con las instancias de entrenamiento. Seguidamente generamos un archivo PDF que podemos

visualizar y analizar qué valores se han tomado como atributos para el nodo y las reglas usadas.

Debido a que el árbol es muy grande y no es posible mostrarlo entero, procederemos a mostrar una parte de él para indicar cómo sería el aspecto de este archivo:

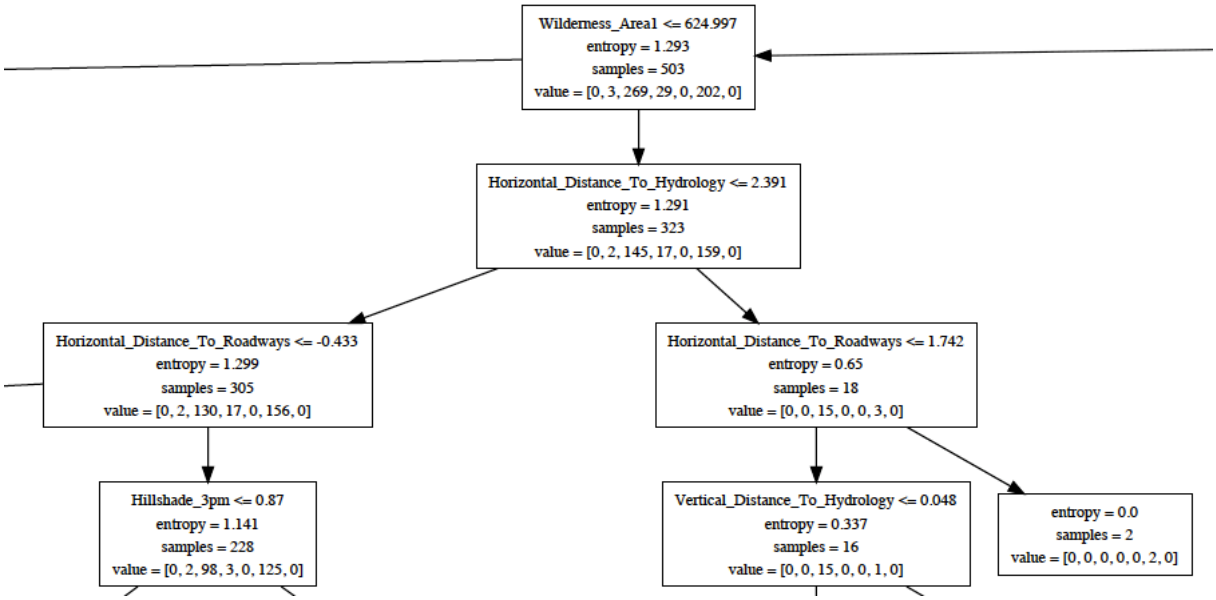
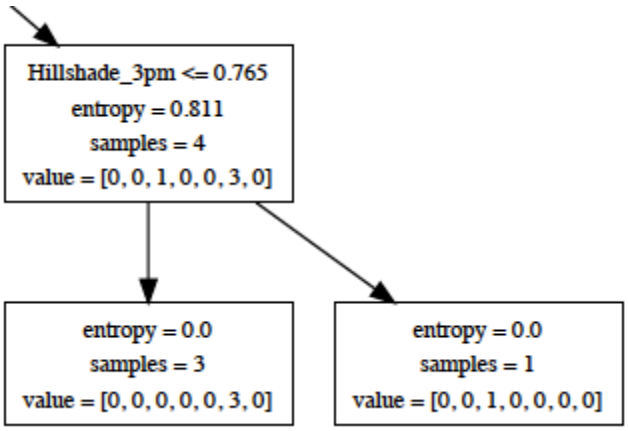



*Ilustración 21. Muestra de clasificación Decision Trees usando librería graphviz*

En cada nodo podemos ver el atributo que se ha seleccionado para decidir si se toma un camino u otro, también podemos ver la entropía que existe en cada nodo, la cantidad de instancias que cumplirían la restricción del nodo y por último un listado de cuantas instancias de cada clase tendríamos en dicho nodo.

Un ejemplo de nodo hoja sería el siguiente:



## 4.8. Ensemble

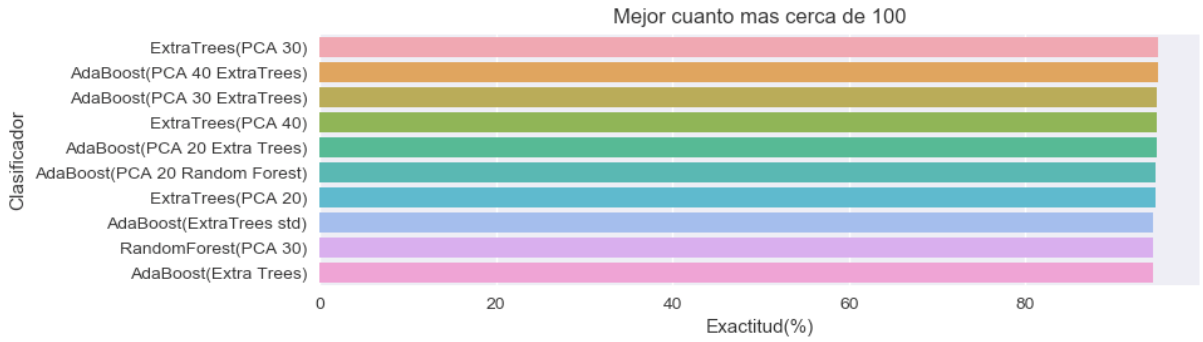
**Tabla de valores obtenidos**

| Clasificador | Ex | LL | Sens | Ta | Tp | Memoria (MB) |
|---|---|---|---|---|---|---|
| **AdaBoost (DecisionTrees)** | 61,05 | 1,93 | 61,05 | 85.35 | 1,27 | 11,51 |
| **AdaBoost (RandomForests)** | 94,42 | 1,84 | 0,94 | 1.209,38 | 67,03 | - |
| **AdaBoost (Extra Trees)** | 94,42 | 1,84 | 94,42 | 1.100,42 | 63,43 | 167,89 |
| **RandomForest** | 94,05 | 0,26 | 94,05 | 22,42 | 1,08 | 147,80 |
| **ExtraTrees** | 93,62 | 0,28 | 0,94 | 16,11 | 1,32 | 219,88 |
| **AdaBoost (DecisionTrees std)** | 61,05 | 1,93 | 61,05 | 61,49 | 1,29 | 118,32 |
| **AdaBoost** | 93,60 | 1,83 | 93,60 | 22,74 | 1,19 | - |
| **AdaBoost(ExtraTrees std)** | 94,46 | 1,84 | 94,46 | 1.092,09 | 58,98 | 319,45 |
| **RandomForest (std)** | 93,88 | 0,27 | 93,88 | 22,93 | 0,87 | - |
| **AdaBoost (PCA 20 Decision)** | 61,92 | 1,93 | 61,92 | 116,17 | 1,07 | - |
| **AdaBoost** | 94,78 | 1,83 | 94,78 | 2.585,63 | 45,36 | - |
| **AdaBoost** | 94,81 | 1,83 | 94,81 | 558,45 | 62,83 | - |
| **RandomForest (PCA 20)** | 94,23 | 0,26 | 94,23 | 43,15 | 0,76 | - |
| **ExtraTrees (PCA 20)** | 94,76 | 0,24 | 94,76 | 10,94 | 1,03 | - |
| **AdaBoost** | 55,90 | 1,94 | 55,90 | 119,60 | 1,12 | - |
| **AdaBoost** | 93,94 | 1,83 | 93,94 | 572,49 | 66,66 | - |
| **RandomForest** | 92,66 | 0,34 | 92,66 | 44,44 | 0,96 | - |
| **ExtraTrees (PCA 20 std)** | 93,68 | 0,30 | 93,68 | 11,39 | 1,08 | - |
| **AdaBoost** | 94,93 | 1,83 | 94,93 | 663,04 | 63,01 | - |
| **RandomForest (PCA 30)** | 94,45 | 0,25 | 94,45 | 57,29 | 0,83 | - |
| **ExtraTrees (PCA 30)** | 94,95 | 0,24 | 94,95 | 14,74 | 1,07 | - |
| **ExtraTrees (PCA 30 std)** | 93,82 | 0,31 | 93,82 | 14,64 | 1,06 | - |
| **AdaBoost** | 94,11 | 1,83 | 94,11 | 655,18 | 62,62 | - |
| **AdaBoost (PCA 40 Decision)** | 58,77 | 1,94 | 58,77 | 231,63 | 1,28 | - |

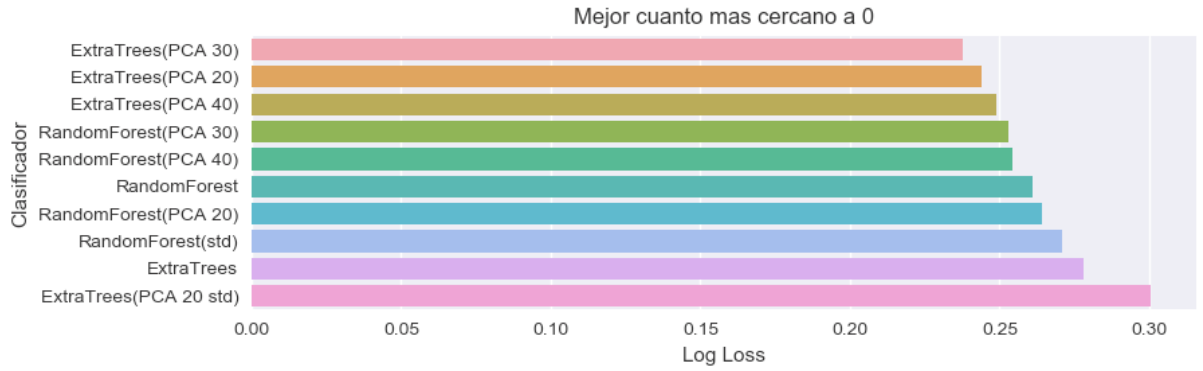
| Clasificador | Ex | LL | Sens | Ta | Tp | Memoria (MB) |
|---|---|---|---|---|---|---|
| **AdaBoost (PCA 40 ExtraTrees)** | 94,95 | 1,83 | 94,95 | 749,47 | 61,25 | - |
| **RandomForest (PCA 40)** | 94,39 | 0,25 | 94,39 | 73,55 | 0,84 | - |
| **ExtraTrees (PCA 40)** | 94,90 | 0,25 | 94,90 | 16,89 | 1,07 | - |
| **AdaBoost (PCA 40 std)** | 93,95 | 1,83 | 93,95 | 732,63 | 62,45 | - |
| **ExtraTrees (PCA 40 std)** | 93,71 | 0,31 | 93,71 | 17,32 | 1,07 | - |

## Gráficas

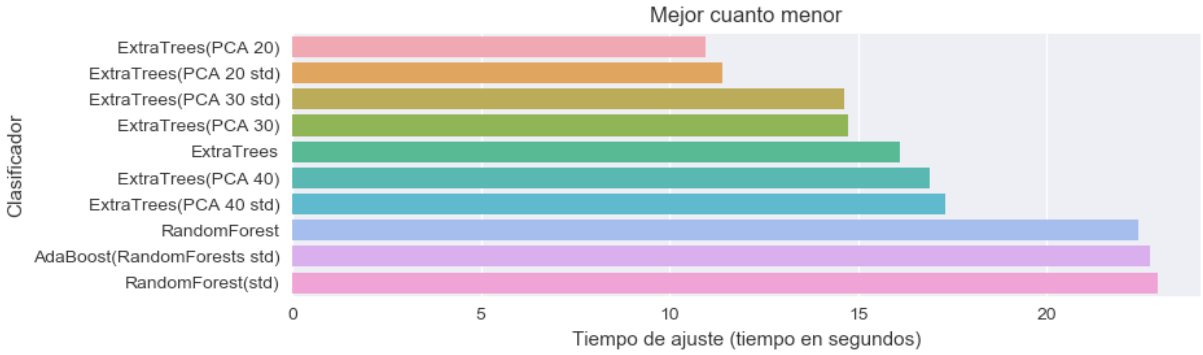
Exactitud




Log Loss



Tiempo de ajuste




## Análisis

Los hiper-parámetros con los que hemos obtenido estos valores son:

- *n_estimators = 50*: cantidad de árboles de profundidad 1 que usaremos para luego llevar a cabo la clasificación
- *learning_rate=1:* La tasa de aprendizaje reduce la contribución de cada clasificador por este rango.
- *algorithm='SAMME':* Algoritmo usado para calcular las probabilidades.

Debido a la cantidad de distintos modelos que se han comparado en esta técnica se han seleccionado para las gráficas los 10 mejores para una mejor comprensión y lectura.

En esta técnica se han incluido otros tipos de algoritmos de tipo conjuntos para así incluirlos en la comparativa. Así por ejemplo AdaBoost usa *DecisionTrees* de profundidad 1 por defecto, pero tenemos la opción de cambiar dichos árboles por otro tipo como *ExtraTrees* o *RandomForests*.

Podemos observar en los datos recogidos que obtenemos exactitudes similares usando técnicas de conjuntos donde usamos árboles distintos de los de decisión que son los que usa AdaBoost por defecto. Así por ejemplo vemos que la mayoría de AdaBoost por defecto solo llega casi al 92% de exactitud aunque juega a su favor el tiempo de ajuste de 65 segundos comparado con los más de 2.500 para AdaBoost usando *RandomForests*.

Destacamos la exactitud obtenida con el algoritmo *ExtraTrees* donde conseguimos valores cercanos al 95% y un Log Loss de 0,24 y además son los más rápidos con tiempos de ajustes que no llegan a los 18 segundos. Este tipo de algoritmo cuyo nombre proviene de ***Ext****remely* ***Ra****ndomized* ***Trees*** se encarga de aleatorizar aún más la construcción de los árboles donde vemos que la elección del atributo que se encargará de dividir los conjuntos desde el nodo será el que marque la varianza del árbol. Este tipo de solución funciona muy bien para aquellos conjuntos de datos que tienen muchas características numéricas que varían continuamente, así podemos observar según los resultados obtenidos.

Otro punto interesante a contemplar en estos métodos es la estandarización de los datos. La aplicación de esta técnica a los datos no ayuda a conseguir una mejor exactitud y es por ello que, a pesar que exista una covarianza mayor entre atributos, sí que permite que al haber mayores diferencias consiga llevar a cabo una mejor clasificación.

Aplicar PCA para estos algoritmos influye positivamente para la mayoría de ellos donde podemos observar en las primeras posiciones que tienen aplicada la reducción de atributos. Así vemos que para *ExtraTrees* con PCA 30 conseguimos la mejor exactitud muy parecido también con *AdaBoost* usando *ExtraTrees* con PCA 40, lo que nos dice que no todos los atributos son influyentes para llevar a cabo la clasificación en ésta técnica.

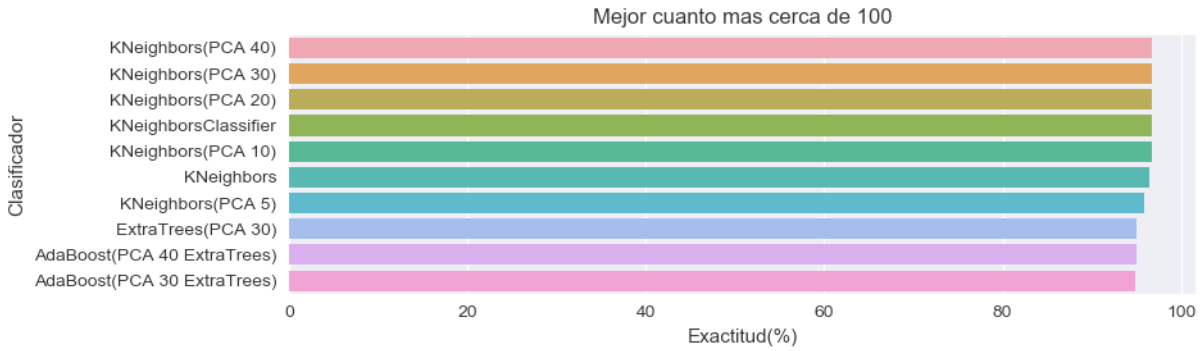
Observando los datos de *Log Loss* podemos ver valores muy cercanos a 0 obteniendo para los mejores (ExtraTrees PCA 20, 30 y 40) por debajo de 0,25. Esto nos quiere decir que el modelo obtenido para ExtraTrees es el que mejor se ajusta para nuestro conjunto de datos que estamos tratando.

## 4.9. Los 10 mejores

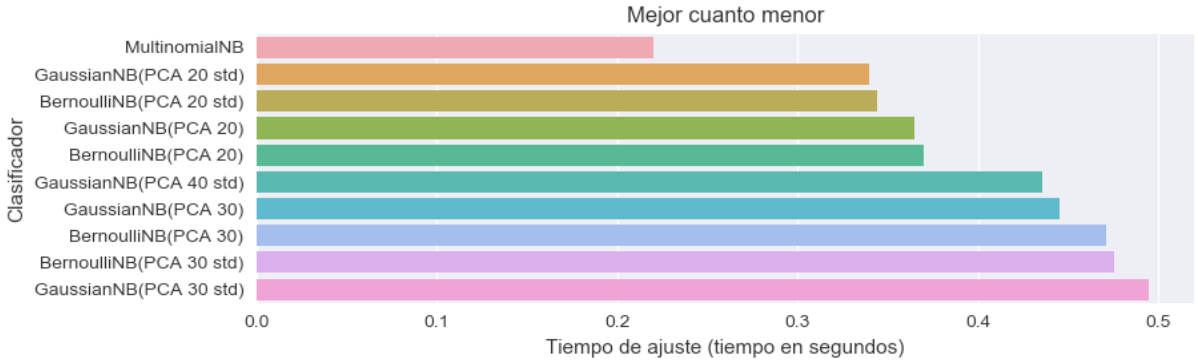
En esta parte vamos a seleccionar aquellos algoritmos con los 10 mejores resultados atendiendo a las gráficas que hemos comentado previamente, exactitud y tiempo, y además incluyendo la de *log loss* que es la métrica que miden en las competiciones de Kaggle, plataforma de la que hemos extraído la información para este trabajo.

Así tenemos los siguientes datos:

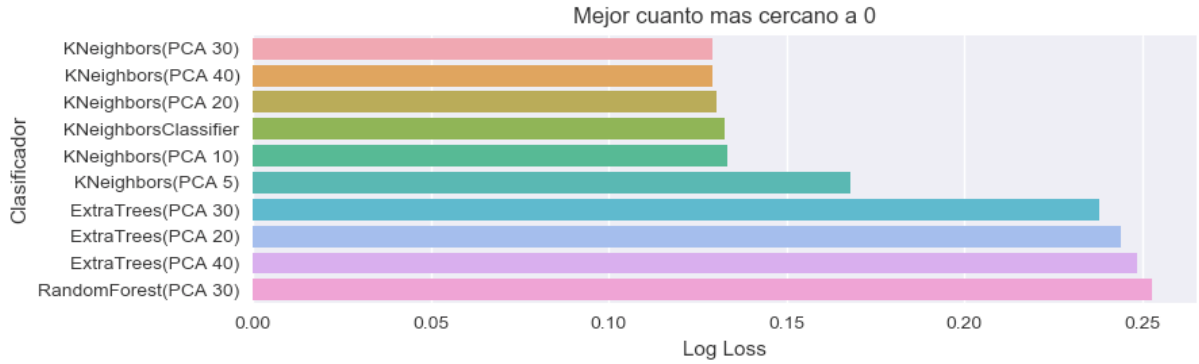
Exactitud



Tiempo de ajuste




Log Loss




Según los datos que hemos obtenido para cada una de nuestras técnicas y los distintos algoritmos que forman parte de ellas hemos extraído aquellas que se pueden considerar como las 10 mejores, pero, ¿cuál de las métricas debemos considerar para decir que es la mejor a tener en cuenta para una clasificación? Muchas de ellas son relevantes y nos dan información de cómo se ha realizado la misma, pero no quiere decir que sean las que se deben tener en cuenta. Esto es debido, como hemos comentado previamente, a que si los datos están desbalanceados porque tenemos más ejemplos de una clase que de otra, quiere decir que van a tomar más peso a la hora de extraer exactitud, sensibilidad o precisión, pero sin embargo es posible que para otras clases donde tenemos menos ejemplos se haya confundido más y por lo tanto no nos sirve para clasificar esa clase. Sí es cierto que tener una mayor exactitud generalmente va a generar un menor log loss pero es necesario también tener en cuenta estos valores a la hora de decidir cuál es mejor y cuál peor.

En la gráfica de exactitud podemos observar una gran presencia para el modelo obtenido para KNeighbors ya sea aplicando PCA o sin ella, y también podemos ver su presencia en las primeras posiciones de Log Loss donde tenemos valores por debajo de 0,15. Para ExtraTrees vemos algo parecido: usando el modelo con PCA 30 obtenemos una exactitud de casi el 95% y un Log Loss de 0,24. Esto es debido a que Log Loss se basa en los cálculo de

probabilidades para cada una de las clases y de esta forma cuanto más cercano al 0 sea este valor quiere decir que las probabilidades han sido más certeras.

Si tenemos en cuenta el tiempo empleado para llevar a cabo el ajuste podemos observar que los mejores valores los obtienen las estrategias de Naive Bayes pero esto no nos dice que vaya a ser mejor clasificador, como así hemos visto en los datos obtenidos en las tablas. Si tenemos una capacidad de computación adecuada podremos seleccionar el mejor algoritmo que se ajuste a nuestro conjunto de datos a clasificar consiguiendo unos mejores resultados, que es de lo que se pretende en este tipo de problemas.

Observando los datos anteriores podemos decir que el mejor clasificador para nuestro conjunto de datos es usando la técnica de KNeighbors usando 30 o 40 componentes principales puesto que los valores obtenidos son muy parecidos. La diferencia la tendremos en el momento del tiempo de predicción puesto que con 30 componentes tarda 4 segundos menos en obtener los resultados.

# Capítulo 5. Conclusiones

El objetivo de este trabajo era el de realizar una evaluación de distintas técnicas y estrategias de clasificación usando la biblioteca de aprendizaje automático scikit-learn para el lenguaje de programación Python. Esta evaluación consistió en realizar la clasificación para saber qué cubierta arbórea es la predominante para distintas áreas del bosque de Roosevelt en el norte del estado de Colorado atendiendo a un conjunto de datos de cerca de 600.000 instancias obtenido de UCI Machine Learning Repository [68] y de donde podríamos extraer distintas métricas que nos ayudan a evaluar y analizar cada una de las técnicas que disponemos para realizar dicha clasificación.

De cada técnica y para cada algoritmo que hemos contemplado se ha realizado una búsqueda exhaustiva en un espacio de posibilidades acotado para encontrar los hiper-parámetros que permitan obtener los mejores resultados sobre nuestro conjunto de datos y, posteriormente, se ha realizado un entrenamiento de cada uno de ellos con dichos hiper-parámetros para así poder realizar una clasificación. A la hora de realizar dicho proceso se ha usado una técnica de validación cruzada donde obtenemos un número de ejemplos para poder entrenar el algoritmo y otra cantidad de ejemplos, que llamamos de validación, para poner a prueba cada algoritmo entrenado y de ahí extraer sus métricas.

Para el procesado de los datos, entrenamiento de los algoritmos y extracción de las métricas para realizar el análisis, hemos llevado a cabo el desarrollo de un programa en el lenguaje Python que nos permia llevar a cabo de forma automática y sencilla este trabajo. El programa Python junto con las métricas extraídas en formato CSV está disponible para su descarga y prueba en el repositorio público siguiente: https://bitbucket.org/juzaru18/trees/.

Como podemos ver, todo aprendizaje supervisado requiere de varias etapas si queremos conseguir que la clasificación sea lo más fiable posible. Así hemos realizado los siguientes pasos para llevar a cabo este trabajo:

1. Preprocesado de los datos: antes de iniciar el entrenamiento de cualquier algoritmo que seleccionemos es adecuado realizar una visualización previa de la naturaleza de nuestros datos. Así podemos saber qué distribución tienen, de qué tipo son, si existen datos perdidos o qué atributo corresponde a la etiqueta de clasificación. Ya que tratamos de aprendizaje supervisado, contamos con un conjunto de datos que nos servirá para entrenar nuestros algoritmos y así, posteriormente, realizar la clasificación para aquellos nuevos no visualizados previamente. También muchos algoritmos requieren estandarizar o normalizar los datos en la escala [0,1] o centrarlos en el rango [-1,1] debido a que son más sensibles a las covarianzas entre los atributos o la distribución de los datos. Esto es debido a la naturaleza del algoritmo que estamos

tratando y que puede llevar a resultados diferentes si no se lleva a cabo dicho pre-procesado de los datos, previo a su entrenamiento.

2. Entrenamiento del algoritmo: esto implica:
    i. **Búsqueda de la estrategia a usar para realizar la validación cruzada**. Como hemos visto en el capítulo 1, en la elección de los parámetros del algoritmo, es adecuado usar una técnica de validación cruzada a la hora de entrenar nuestro algoritmo. Así conseguiremos extraer de dicho algoritmo un modelo que se ajuste de una forma generalizada a la naturaleza del conjunto de datos a tratar.
    ii. **Optimización de los hiper-parámetros para el algoritmo**. La gran mayoría de las distintas técnicas (*Linear, Trees, Ensemble*, etc) tienen la posibilidad de indicar expresamente unos parámetros, que no son aprendidos cuando los entrenamos, como por ejemplo el parámetro *learning-rate* para el Perceptrón, o qué técnica se usa para decidir qué atributo dividirá los datos en el árbol a partir de cada nodo. Como hemos visto, la biblioteca scikit-learn da la posibilidad de una búsqueda exhaustiva en un espacio finito creado por nosotros para obtener estos hiper-parámetros.
    iii. **Cuantificación de la calidad de las predicciones realizadas por el algoritmo**. A la hora de evaluar cada algoritmo, tras el entrenamiento de cada uno de ellos y con el uso de validación cruzada, podemos extraer distintas métricas que nos ayudan a saber cómo se ha comportado el algoritmo seleccionado tras su entrenamiento. Así como comentamos en el capítulo 1, sección “métricas de un algoritmo”, podemos cuantificar la calidad del algoritmo atendiendo a distintas métricas que indican como ha llevado la clasificación para el conjunto de validación. De esta forma podemos establecer criterios que nos ayuden a llevar a cabo un análisis y comparación entre las distintas técnicas y algoritmos que podemos usar de la biblioteca scikit-learn.
3. Uso del modelo obtenido: una vez realizado el entrenamiento y evaluados según las métricas seleccionadas, el usuario podrá elegir aquel que mejor se ajuste a su criterio y podrá realizar la clasificación para nuevos datos que no han sido vistos previamente, datos que no tienen etiqueta y que queremos clasificarlos usando el algoritmo seleccionado habiendo entrenado previamente.

En este trabajo nos hemos centrado en las dos primeras partes. La primera de ellas que es el preprocesado del conjunto de datos para así saber qué distribución tenían, si algún dato

estaba perdido e incluso para la división de los mismos en atributos y sus respectivas etiquetas. Y por otro, y el más importante en el que centramos este trabajo es el entrenamiento de los algoritmos seleccionados de cada técnica disponibles en la biblioteca de scikit-learn. Con el entrenamiento podemos extraer las distintas métricas de cada algoritmo para hacer la comparación y análisis.

Tras los datos extraídos en el entrenamiento de cada técnica podemos extraer las siguientes conclusiones concluir:

- **La búsqueda de los hiper-parámetros para cada uno de los algoritmos es influyente**. Para ello debemos tener en cuenta la capacidad de cómputo que poseemos para realizar dicha búsqueda. A mayor capacidad de cómputo, podremos evaluar un rango de posibilidades mayor para cada algoritmo. Debemos tener en cuenta la elección de la métrica que se utilizará para encontrar los mejores parámetros en dicha búsqueda exhaustiva, parámetro scoring del método GridSearchCV, donde podemos usar todos aquellos que están dentro de las métricas de clasificación, como pueden *accuracy_score* para usar la exactitud o quizás *neg_log_loss* para usar log loss.
- **La cantidad de instancias en el conjunto es algo que también influye**. Así hemos podido observar cuando hemos intentado realizar el ajuste para la técnica de *Support Vector Machines*. La búsqueda de los hiperplanos que permitan la separación adecuada para llevar a cabo a la clasificación puede llevar mucho tiempo de ajuste, siendo su complejidad cuadrática cuando el número de ejemplos es muy grande y por lo tanto se complica cuando éste número supera los 10000 ejemplos [88]
- **El tipo de datos con el que estamos tratando influye.** Así lo hemos podido comprobar para algunos ejemplos de algoritmos donde vemos que al estandarizar los datos se consiguen mejores resultados. La estandarización del conjunto de datos es algo que para muchos algoritmos de aprendizaje automático influye, es posible que se comporten diferente si cada característica o atributo no tienen una distribución normal. Así vemos por ejemplo para los algoritmos de la técnica *Linear* (los regularizadores L1 o L2) o Support Vector Machines (el kernel RBF) donde las funciones objetivo asumen que todos los atributos están centrados al 0 y tienen una varianza unitaria. Esto lo hemos podido comprobar en los modelos lineales tanto para LogisticRegression donde obtenemos mejores valores cuando los datos están estandarizados, además de un tiempo de ajuste drásticamente menor, como para incluso el Perceptrón donde estandarizando los datos y con PCA 40 consigue aumentar su ratio de acierto (exactitud).

  Este ejemplo lo podemos ver reflejado en las técnicas de NaiveBayes donde aplicando estandarización conseguimos la mejor exactitud para BernouilliNB, tal y como vimos

en el Capítulo 1, puesto que funciona muy bien para aquellos valores binarios almacenados en los atributos en el rango [0,1]

- **El uso de la técnica de reducción de la dimensión de los datos a través de PCA (Principal Component Analysis) influye en los algoritmos**. En algunos casos observamos que mejoran ciertos valores, como puede ser la exactitud o el tiempo de ajuste, sin embargo, para otros algoritmos no observamos estas mejoras. Así por ejemplo vemos que en la mayoría de algoritmos el uso de esta técnica mejora la exactitud, lo que quiere decir que existen atributos que desvían su atención en el ajuste porque puede ocurrir que dos de ellos tengan valores muy parecidos o en rangos iguales y por lo tanto eliminar uno de ellos ayudará a mejorar el entrenamiento. Mejorar el entrenamiento no solo quiere decir que consigamos mejores métricas de exactitud, log loss o sensibilidad, sino también hemos podido observar que el tiempo de ajuste se reduce y en algunos de ellos drásticamente como podemos ver para LogisticRegression con PCA 30 y los datos estandarizados, el tiempo de ajuste se reduce 42 veces con respecto al mismo método sin aplicar PCA ni datos estandarizados.
- **Cada algoritmo tiene un uso de memoria distinto en su ajuste o entrenamiento**. Como hemos visto cada algoritmo tiene un uso diferenciado de la memoria disponible del sistema cuando está realizando su entrenamiento con el conjunto de datos. Si bien podemos intuir que aplicando PCA este uso debe disminuir, no es un valor que hayamos podido apreciar adecuadamente con la biblioteca usada para este trabajo *memory profiler* [84].

Por último, atendiendo a las métricas obtenidas podemos concluir que el mejor algoritmo a usar para el conjunto de datos que hemos tratado es el de KNeighborsClassifier usando PCA con 40 componentes principales. Llegamos a esta conclusión debido a su alto ratio de acierto (96,78% de exactitud), su bajo valor de error en las probabilidades (0,13 de log loss) y su bajo tiempo de ajuste (5,35 segundos).

# Trabajo futuro

Las posibilidades del aprendizaje automático hoy en día soy muy variadas. Es por ello que este trabajo ofrece la oportunidad de ampliar conocimientos e investigación en la misma línea que se ha ido siguiendo para el mismo.

Un ejemplo de ello podría ser el uso de una capacidad de cómputo mayor, tanto en la parte de búsqueda exhaustiva de los hiper-parámetros de cada algoritmo como en la parte de ajuste de los algoritmos donde obtendremos valores en menor tiempo y podemos obtener métricas para la técnica de *Support Vector Machines* y así compararla. Ejemplos para esto

podría ser la solución de utilizar estructuras de datos de pandas distribuidas en una granja de servidores usando Dask [51] para así realizar un procesado de los datos con mayor velocidad y asegurándonos que el conjunto de datos puede ser almacenado si es demasiado grande. Otro ejemplo puede ser el uso de Apache Spark [89] para así distribuir los datos en una granja de servidores y conseguir un rendimiento mayor para su procesado.

Otro ejemplo de trabajo futuro sería el desarrollo de una biblioteca más fiable para la determinación del uso de memoria que se realiza en el tiempo de ajuste por cada algoritmo que queremos tratar. Usando la biblioteca de *memory profiler* nos hemos encontrado que para muchos casos nos mostraba valores de uso de memoria negativos y es algo que no nos da mucha confianza teniendo en cuenta los cálculos que deben hacer. En dicha biblioteca sería también interesante añadir el cálculo de uso de CPU que se realiza en el ajuste para cada algoritmo y donde así podremos saber cuál de ellos requiere de más uso.

Un trabajo interesante que se puede llevar a cabo partiendo de este mismo es la comparación de la librería de scikit-learn con otras que también se usan para clasificación y que son interesantes estudiar. Así tenemos el caso de TensorFlow [90] que es una biblioteca de código abierto para aprendizaje automático y que puede usarse tanto para computación en CPU como en GPU por lo que podemos ganar en computación y comparar sus resultados con respecto a los obtenidos con scikit-learn.

La mayoría de los algoritmos estudiados también tienen la posibilidad de usarse para regresión, es decir, predecir los valores futuros continuos basándose en los datos pasados. Sería interesante desarrollar una línea de investigación que extraiga las métricas de los algoritmos previamente mencionados para ver cómo se comportan para un conjunto de datos que esté pensado para ese tipo de problema.

Por último, el uso de la técnica para llevar a cabo la clasificación y extracción de las métricas requiere de unos conocimientos y un previo estudio del funcionamiento del programa desarrollado. Podría resultar interesante el desarrollo de una interfaz gráfica que ayude a seleccionar las distintas opciones que se han utilizado en el mismo, por ejemplo si queremos usar GridSearch, mostrar las gráficas, entrenar los algoritmos (todos o solo para una técnica), etc, y de esta forma facilitar el análisis.

# Chapter 5. Conclusions

The goal of this project was to perform an analysis and evaluation of different classification techniques and strategies using the scikit-learn machine learning library for the Python programming language. This review consisted in studying the behavior of the different techniques when they´re making a classification, which consisted in knowing which tree cover is predominant in different areas of the Roosevelt forest in the north of the state of Colorado based. It was based on a dataset close to 600,000 Instances obtained from the UCI Machine Learning Repository [66] and from where we could extract different metrics that help us evaluate and analyze each of the techniques we have to perform this classification.

From the observed behavior we could extract different metrics that helped us to evaluate and analyze each of the techniques that scikit-learn has to perform a classification.

For each technique and for each algorithm that we have observed, an exhaustive search has been carried out in a space of limited possibilities to find the best hyper-parameters that, in our dataset, obtained better results and, subsequently, a training of each of them with those hyper-parameters in order to make a classification. When carrying out this process, a cross-validation technique has been used where we obtain a number of examples to train the algorithm and another number of examples, which we call validation dataset, to test each trained algorithm and extract its metrics from there.

For the processing of the data, training of the algorithms and subsequent extraction of the metrics to perform the analysis, we have carried out the development of a program in the Python language that allows us to carry out this work automatically and easily. The program in Python together with the metrics extracted in CSV format are available for download and testing in the public repository: https://bitbucket.org/juzaru18/trees/.

As we can see all supervised learning requires several stages if we want to make our classification as reliable as possible. So, we have been taking this next sections to carry out this project:

1. Preprocessing the data: before starting the training of any algorithm that we select, it is appropriate to preview the nature of our data. In that way we can know what distribution they have, what type they are, if there is lost data or what attribute corresponds to the classification label, if we got them in raw format. Since we are dealing with supervised learning, we have a set of data that will help us to train our algorithms and, subsequently, perform the classification for those new data not previously visualized. Many algorithms also require standardizing the data in the scale [0,1] or centering them in the range [-1,1] because they are more sensitive to the covariance between the attributes or the distribution of the data,

for example Gaussian. This is due to the nature of the algorithm that we are dealing with and that can lead to different results if the preprocessing of the data is not carried out prior to its training

2. Training of the algorithm: this implies:
    i. **Search for the strategy to be used to perform cross-validation.** As we have seen in chapter 1, in the choice of algorithm parameters, it is appropriate to use a cross-validation technique when training our algorithm. This way we will be able to extract from this algorithm a model that fits in a generalized way to the nature of the data set to be treated
    ii. **Optimization of algorithm hyper-parameters**. The vast majority of the different techniques (Linear, Trees, Ensemble, etc.) have the possibility to expressly indicate some parameters, which are not learned when we train them, such as the learning-rate parameter for the Perceptron, or which technique is used to decide which attribute will divide the data in the tree from each node. As we have seen scikit-learn gives the possibility of an exhaustive search in a finite space created by us to obtain these hyper-parameters optimization.
    iii. **Quantification of the quality of the predictions made by the algorithm.** When evaluating algorithms, after the training of each one and using cross validation, we can extract different metrics that help us to know how the selected algorithm has behaved after the training. Thus, as we discussed in chapter 1, section "metrics of an algorithm", we can quantify the quality of the algorithm according to different metrics that indicate how it has carried out the classification for the validation set. In this way we can establish criteria that help us carry out an analysis and comparison between the different techniques and algorithms that we can use from the scikit-learn library.
3. Use of the obtained model: once the training is done and evaluated according to the selected metrics, the user can choose the one that best fits their criteria and can perform the classification for new data that has not been previously seen, data that is not labeled and where we want to make a classification using the algorithm selected having previously trained.

In this work we have focused on the first two parts. The first one is the preprocessing of the data set in order to know what distribution they had, if any data was lost or even to divide them into attributes and their respective labels. And on the other, and the most important one in which we focus this work is the training of the algorithms selected for each technique available

in the scikit-learn library. With the training we can extract the different metrics of each algorithm to make the comparison and analysis

After the data extracted in the training of each technique we can give the following conclusions:

- **The hyper-parameters search for each of the algorithms is influential**. For this we must take into account the computing capacity we have to perform this exhaustive search. The higher the computation capacity, the better we can evaluate a range of possibilities for each algorithm. We must take into account the choice of metric that will be used to find the best parameters in this search, where we can use all those that are within the classification metrics, such as accuracy_score to use accuracy or perhaps neg_log_loss to find the best log loss
- **The number of instances in the dataset is something that also influences**. This is what we observed when we tried to make the training for the Support Vector Machines technique. The search for hyperplanes that allow adequate separation to carry out the classification can take a long time to adjust, being its complexity quadratic when the number of examples is very large and therefore it becomes impractical when this number exceeds more than 10,000 examples.
- **The type of data which we are dealing influences**. In this way we have been able to verify it for some examples of algorithms where we see that when standardizing the data we obtained better results. The standardization of the dataset is something that for many machine learning algorithms influences, it is possible that they behave differently if each characteristic or attribute does not have a normal distribution, as for example happens with the Gaussian type distribution. This is the case, for example, for the algorithms of the Linear technique (the L1 or L2 regulators) or Support Vector Machines (the RBF kernel) where the objective functions assume that all the attributes are centered at 0 and have a unit variance. We have seen this in the linear models for both LogisticRegression where we obtain better values when the data is standardized, in addition to a drastically shorter adjustment time, as for even the Perceptron where standardizing the data and with PCA 40 manages to increase its accuracy ratio (accuracy).

  This example can be seen in the NaiveBayes techniques where applying standardization we get the best accuracy for BernouilliNB, as we saw in Chapter 1, since it works very well for those binary values stored in the attributes [0,1].
- Each algorithm has a different memory usage in its adjustment or training

  While we can intuit that applying PCA this use should decrease, it is not a value that we have been able to appreciate adequately with the library used for this work memory profiler [84]

Finally, based on the obtained metrics, we can conclude that the best algorithm to use for the data set we have tried for this work is that of KNeighborsClassifier using PCA with 40 main components. We reached this conclusion due to its high accuracy ratio (96.78% accuracy), its low error value in the probabilities (0.13 of loss) and its low adjustment time (4.35 seconds).

## Future work

The possibilities of machine learning today are very varied. That is why this work offers the opportunity to expand knowledge and research in the same line that has been followed for it.

An example of this could be the use of a greater computing capacity both in the exhaustive search part of the hyper-parameters of each algorithm and in the adjustment part of the algorithms. There we will obtain values in less time and we can obtain metrics for the Support Vector Machines technique and so compare it.

Examples for this could be the solution of using distributed panda data structures in a server farm using Dask [49] o perform a processing of the data with greater speed and making sure that the dataset can be stored if it is too large. Another example can be the use of Apache Spark [89] to distribute the data in a server farm and achieve greater performance for processing.

Another example of future work would be the development of a more reliable library for the determination of the memory use that is made in the time of adjustment for each algorithm that we want to deal with. Using the memory profiler library we have found that for many cases it showed us negative memory usage values and it is something that does not give us much confidence considering the calculations that should be done. In this library it would also be interesting to add the calculation of CPU usage that is made in the adjustment for each algorithm and where we can know which of them requires more CPU use.

An interesting work that can be carried out starting from this is the comparison of the library of scikit-learn with others that are also used for classification and that are interesting to study. So we have the case of TensorFlow [87] which is an open source library for machine learning and can be used for both CPU and GPU computing so we can gain in computing and compare their results with those obtained with scikit-learn.

Most of the algorithms studied also have the possibility of being used for regression, that is, predicting the continuous future values based on the past data. It would be interesting to develop a research line that extracts the metrics of the aforementioned algorithms to see how they behave for a data set that is designed for that type of problem.

Finally, the use of the technique to carry out the classification and extraction of the metrics requires some knowledge and a previous study of the operation of the developed program. It

could be interesting to develop a graphical interface that helps to select the different options that have been used in it, for example if we want to use GridSearch, display the graphs, train the algorithms (all or only for one technique), etc.

# Bibliografía


[1] A. Goldbloom. 2017. Your Home for Data Science. Retrieved from https://www.kaggle.com/

[2] Ricardo Aler Mur. Clasificadores Knn-I. Retrieved from http://ocw.uc3m.es/ingenieria-informatica/analisis-de-datos/transparencias/KNNyPrototipos.pdf

[3] J Ali, R Khan, N Ahmad, and I Maqsood. 2012. Random forests and decision trees. *IJCSI Int. J. Comput. Sci. Issues* 9, 5 (2012), 272–278.

[4] Sylvain Arlot and Alain Celisse. 2009. A survey of cross-validation procedures for model selection. 4, (2009), 40–79. DOI:https://doi.org/10.1214/09-SS054

[5] W.A. Awad and S.M. ELseuofi. 2011. Machine Learning Methods for Spam E-Mail. *Int. J. Comput. Sci. Inf. Technol.* 3, 1 (2011), 173–184.

[6] AWS. Open Data on AWS. Retrieved April 20, 2018 from https://aws.amazon.com/es/government-education/open-data/

[7] B Azhagusundari and Antony Selvadoss Thanamani. 2013. Feature Selection based on Information Gain. *Int. J. Innov. Technol. Explor. Eng.* 2, 2 (2013), 18–21. DOI:https://doi.org/2278-3075

[8] S Balakrishnama and a Ganapathiraju. 1998. Linear Discriminant Analysis - a Brief Tutorial. *Compute* 11, (1998), 1–9. DOI:https://doi.org/http://www.isip.piconepress.com/publications/reports/1998/isip/lda/

[9] G Biau. 2012. Analysis of a random forests model. *J. Mach. Learn. Res.* 13, (2012), 1063–1095.

[10] Leo Breiman. 2001. Full-Text. (2001), 1–33. DOI:https://doi.org/10.1017/CBO9781107415324.004

[11] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake Vanderplas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. 2013. API design for machine learning software: experiences from the scikit-learn project. (2013), 1–15. Retrieved from http://arxiv.org/abs/1309.0238

[12] Wen Yuan Chen, Sin Horng Chen, and Cheng Jung Lin. 1996. A speech recognition method based on the sequential multi-layer perceptrons. *Neural Networks* 9, 4 (June 1996), 655–669. DOI:https://doi.org/10.1016/0893-6080(95)00140-9

[13] Joseph A. Cruz and David S. Wishart. 2006. Applications of machine learning in cancer prediction and prognosis. *Cancer Inform.* 2, (2006), 59–77. DOI:https://doi.org/10.1177/117693510600200030

[14] Peter Dayan. 2009. Unsupervised learning. *MIT Encycl. Cogn. Sci.* (2009), 1–7. DOI:https://doi.org/10.1007/BF00993379

[15] Georgia Tech Ece. Quasi-Newton Methods. 1, 1–7.

[16] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. 2017. Dermatologist-level classification of skin cancer with deep neural networks. *Nature* 542, (January 2017), 115. Retrieved from http://dx.doi.org/10.1038/nature21056

[17] Kamran Etemad and Rama Chellappa. 1997. Discriminant analysis for recognition of human face images. *J. Opt. Soc. Am. A* 14, 8 (1997), 1724–1733. DOI:https://doi.org/10.1364/JOSAA.14.001724

[18] Silvia Figini, Università Pavia, Via San Felice, I- Pavia, and Mario Maggi. 2014. Performance of credit risk prediction models via proper loss functions. 64, January (2014).

[19] Yoav Freund and Llew Mason. 1999. The alternating decision tree learning algorithm. *Proceeding Sixt. Int. Conf. Mach. Learn.* (1999), 10 str. DOI:https://doi.org/10.1007/s13398-014-0173-7.2

[20] Yoav Freund and Robert E. Schapire. 1995. A desicion-theoretic generalization of on-line learning and an application to boosting. 139, (1995), 23–37. DOI:https://doi.org/10.1007/3-540-59119-2_166

[21] Mikel Galar, Alberto Fern, Edurne Barrenechea, and Humberto Bustince. Selecci ´ on din ´ amica de clasificadores para la estrategia Uno-contra-Uno : Evitando los clasificadores no competentes.

[22] Henri P Gavin. 2017. The Levenberg-Marquardt method for nonlinear least squares curve-fitting problems. (2017), 1–19. DOI:https://doi.org/10.1080/10426914.2014.941480

[23] Z. Ghahramani. 2000. Information Theory. *Encycl. Cogn. Sci.* 1991 (2000), 1–7. DOI:https://doi.org/10.1002/0470018860.s00643

[24] Zoubin Ghahramani. 2004. Unsupervised Learning BT - Advanced Lectures on Machine Learning. *Adv. Lect. Mach. Learn.* 3176, Chapter 5 (2004), 72–112. DOI:https://doi.org/10.1007/978-3-540-28650-9_5

[25] Abdi Hervé and Williams Lynne J. 2010. Principal component analysis. *Wiley Interdiscip. Rev. Comput. Stat.* 2, 4 (June 2010), 433–459. DOI:https://doi.org/10.1002/wics.101

[26] Hadi Hormozi, Elham Hormozi, and Hamed Rahimi Nohooji. 2012. The Classification of the Applicable Machine Learning Methods in Robot Manipulators. *Int. J. Mach. Learn. Comput.* 2, 5 (2012), 560–563. DOI:https://doi.org/10.7763/IJMLC.2012.V2.189

[27] W Hu, W Hu, and S Maybank. 2008. AdaBoost-Based Algorithm for Network Intrusion Detection. *IEEE Trans. Syst. Man, Cybern. Part B* 38, 2 (2008), 577–583. DOI:https://doi.org/10.1109/TSMCB.2007.914695

[28] Sadegh Bafandeh Imandoust and Mohammad Bolandraftar. 2013. Application of K-Nearest Neighbor ( KNN ) Approach for Predicting Economic Events : Theoretical Background. *Int. J. Eng. Res. Appl.* 3, 5 (2013), 605–610.

[29] Lee V J., Lye D C., Sun Y., and Leo Y S. 2009. Decision tree algorithm in deciding hospitalization for adult patients with dengue haemorrhagic fever in Singapore. *Trop. Med. Int. Heal.* 14, 9 (August 2009), 1154–1159. DOI:https://doi.org/10.1111/j.1365-3156.2009.02337.x

[30] M. Akhil jabbar, B.L. Deekshatulu, and Priti Chandra. 2013. Classification of Heart Disease Using K- Nearest Neighbor and Genetic Algorithm. *Procedia Technol.* 10, (2013), 85–94. DOI:https://doi.org/10.1016/j.protcy.2013.12.340

[31] Kaggle. Kaggle Forest Cover Type Prediction. Retrieved from https://www.kaggle.com/c/forest-cover-type-prediction/kernels

[32] Ron Kohavi. 1995. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. *Appear. Int. Jt. Conf. Articial Intell.* 5, (1995), 1–7. DOI:https://doi.org/10.1067/mod.2000.109031

[33] Sotiris B. Kotsiantis. 2007. Supervised Machine Learning: A Review of Classification Techniques. *Informatica* 31, (2007), 249–268. DOI:https://doi.org/10.1115/1.1559160

[34] Arun D. Kulkarni and Barrett Lowe. 2016. Random Forest for Land Cover Classification. *Int. J. Recent Innov. Trends Comput. Commun.* 4, 3 (2016), 58–63.

[35] T Li, S H Zhu, and M Ogihara. 2006. Using discriminant analysis for multi-class classification: an experimental investigation. *Knowl. Inf. Syst.* 10, 4 (2006), 453–472. DOI:https://doi.org/10.1007/s10115-006-0013-y

[36] W S McCulloch and W Pitts. 1943. A Logical Calculus of the Idea Immanent in Nervous Activity. *Bull. Math. Biophys.* 5, (1943), 115–133. DOI:https://doi.org/10.1007/BF02478259

[37] Kathleen H Miao, Julia H Miao, and George J Miao. 2016. Diagnosing Coronary Heart Disease Using Ensemble Machine Learning. *Int. J. Adv. Comput. Sci. Appl.* 7, 10 (2016), 30–39.

[38] Eduardo Morales. 2012. Aprendizaje Bayesiano Contenido. 2012, (2012), 1–86.

[39] Perceptron Multicapa. 1969. Perceptron Multicapa. (1969), 1–49. Retrieved from http://bibing.us.es/proyectos/abreproy/12166/fichero/Volumen+1+-+Memoria+descriptiva+del+proyecto%252F3+-+Perceptron+multicapa.pdf

[40] Pure Newton. 2009. Newton ' s Method. 1, (2009), 83–95.

[41] C Warren Olanow, Ray L Watts, and William C Koller. 2001. An algorithm (decision tree) for the management of Parkinson's disease (2001): *Neurology* 56, suppl 5 (June 2001), S1 LP-S88. Retrieved from http://n.neurology.org/content/56/suppl_5/S1.abstract

[42] OpenCV. Introduction to Support Vector Machines.

[43] OpenDataSoft. Next Generation Data Sharing.

[44] Sanjay Kumar Palei and Samir Kumar Das. 2009. Logistic regression model for prediction of roof fall risks in bord and pillar workings in coal mines: An approach. *Saf. Sci.* 47, 1 (January 2009), 88–96. DOI:https://doi.org/10.1016/J.SSCI.2008.01.002

[45] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion,

Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2012. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12, (2012), 2825–2830. DOI:https://doi.org/10.1007/s13398-014-0173-7.2

[46] K. Prasad, S. K. Dash, and U. C. Mohanty. 2010. A logistic regression approach for monthly rainfall forecasts in meteorological subdivisions of India based on DEMETER retrospective forecasts. *Int. J. Climatol.* 30, 10 (2010), 1577–1588. DOI:https://doi.org/10.1002/joc.2019

[47] J. R. Quinlan. 1986. Induction of Decision Trees. *Mach. Learn.* 1, 1 (1986), 81–106. DOI:https://doi.org/10.1023/A:1022643204877

[48] Sebastian Raschka. 2015. *Python Machine Learning*. DOI:https://doi.org/10.1007/s13398-014-0173-7.2

[49] Ryan Rifkin, Aldebaro Klautau, and Klautau@ieee Org. 2004. In Defense of One-Vs-All Classification. *J. Mach. Learn. Res.* 5, (2004), 101–141. DOI:https://doi.org/10.1007/BF00718004

[50] Irina Rish. 2001. An empirical study of the naive Bayes classifier. *Empir. methods Artif. Intell. Work. IJCAI* 22230, JANUARY 2001 (2001), 41–46. DOI:https://doi.org/10.1039/b104835j

[51] Matthew Rocklin. 2015. Dask : Parallel Computation with Blocked algorithms and Task Scheduling. *Proc. 14th Python Sci. Conf.* Scipy (2015), 130–136. Retrieved from https://spark.apache.org/

[52] F Rosenblatt. 1958. The perceptron: A probabilistic model for information storage and organization in …. *Psychol. Rev.* 65, 6 (1958), 386–408. DOI:https://doi.org/10.1037/h0042519

[53] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *Nature* 323, (October 1986), 533. Retrieved from http://dx.doi.org/10.1038/323533a0

[54] Ryan Rifkin. 2008. Multiclass Classification. *9.520 Cl. 06* (2008), 59.

[55] a. K. Santra and C. Josephine Christy. 2012. Genetic Algorithm and Confusion Matrix for Document Clustering. *Int. J. Comput. Sci.* 9, 1 (2012), 322–328. Retrieved from http://ijcsi.org/papers/IJCSI-9-1-2-322-328.pdf

[56] M Sarnovsky and M Vronc. 2014. Distributed boosting algorithm for classification of text documents. In *2014 IEEE 12th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*, 217–220. DOI:https://doi.org/10.1109/SAMI.2014.6822410

[57] Robert E. Schapire. 2013. Explaining adaboost. *Empir. Inference Festschrift Honor Vladimir N. Vapnik* (2013), 37–52. DOI:https://doi.org/10.1007/978-3-642-41136-6_5

[58] Wenqian Shang, Houkuan Huang, Haibin Zhu, Yongmin Lin, Youli Qu, and Zhihai Wang. 2007. A novel feature selection algorithm for text categorization. *Expert Syst. Appl.* 33, 1 (July 2007), 1–5. DOI:https://doi.org/10.1016/J.ESWA.2006.04.001

[59] Hiroshi Shimodaira. 2015. Text Classification using Naive Bayes. 4 (2015).

[60] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Fan Hui, Laurent Sifre, George Van Den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. DOI:https://doi.org/doi:10.1038/nature24270

[61] Santosh Srivastava, Maya Gupta, and Béla Frigyik. 2007. Bayesian quadratic discriminant analysis. *J. Mach. Learn. Res.* 8, (2007), 1277–1305.

[62] M Strano and B M Colosimo. 2006. Logistic regression analysis for experimental determination of forming limit diagrams. *Int. J. Mach. Tools Manuf.* 46, 6 (2006), 673–682. DOI:https://doi.org/https://doi.org/10.1016/j.ijmachtools.2005.07.005

[63] N. Suguna and K. Thanushkodi. 2010. An Improved k-Nearest Neighbor Classification Using Genetic Algorithm. *Int. J. Comput. Sci. Issues* 7, 4 (2010), 18–21.

[64] Axel Sundén. 2010. Trading based on classification and regression trees. *Math.Kth.Se* (2010). Retrieved from http://www.math.kth.se/matstat/seminarier/reports/M-exjobb10/100308b.pdf

[65] Richard S Sutton and Andrew G Barto. 2015. Reinforcement Learning : An Introduction. (2015).

[66] Kari Torkkola. 2001. Linear discriminant analysis in document classification. *ICDM Work. Text MiningICDM Work. Text Min.* (2001). Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.16.4044&rep=rep1&type=pdf

[67] Næs Tormod and Mevik Bjørn-Helge. 2001. Understanding the collinearity problem in regression and discriminant analysis. *J. Chemom.* 15, 4 (April 2001), 413–426. DOI:https://doi.org/10.1002/cem.676

[68] UCI. Cover Type repository. Retrieved October 20, 2017 from https://archive.ics.uci.edu/ml/datasets/covertype

[69] UCI. UCI Machine Learning Repository. Retrieved from http://archive.ics.uci.edu/ml/index.php

[70] P Viola and M Jones. 2001. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, I-511-I-518 vol.1. DOI:https://doi.org/10.1109/CVPR.2001.990517

[71] W. McKinney. pandas: a python data analysis library. Retrieved October 20, 2017 from https://pandas.pydata.org/pandas-docs/stable/index.html

[72] Daniel Wolpert and Zoubin Ghahramani. 2005. Bayes rule in perception, action and cognition. *Science (80-. ).* (2005), 1–4. Retrieved from http://eprints.pascal-network.org/archive/00001354/

[73] Cui Yu, Beng Chin Ooi, Kian-Lee Tan, and H. V. Jagadish. 2001. Indexing the Distance: An Efficient Method to KNN Processing. *Int. Conf. Very Large Databases* (2001), 421–430.

[74] Model evaluation: quantifying the quality of predictions. Retrieved from http://scikit-learn.org/stable/modules/model_evaluation.html

[75] Linear and Quadratic Discriminant Analysis. Retrieved from http://scikit-learn.org/stable/modules/lda_qda.html

[76] Clasificador Naïve Bayes. Retrieved January 8, 2018 from http://naivebayes.blogspot.com.es/

[77] Is See5/C5.0 Better Than C4.5? Retrieved February 10, 2018 from http://rulequest.com/see5-comparison.html

[78] Tree algorithms: ID3, C4.5, C5.0 and CART. Retrieved from http://scikit-learn.org/stable/modules/tree.html#tree-algorithms-id3-c4-5-c5-0-and-cart

[79] What is Graphviz. Retrieved December 10, 2017 from http://www.graphviz.org/

[80] Ensemble methods. Retrieved January 15, 2018 from http://scikit-learn.org/stable/modules/ensemble.html#ensemble-methods

[81] Random Forests. Retrieved February 21, 2018 from http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn-ensemble-randomforestclassifier

[82] scikit-learn PCA. Retrieved from http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

[83] Matplotlib. Retrieved November 20, 2017 from https://matplotlib.org/

[84] Memory Profiler. Retrieved March 2, 2018 from https://pypi.org/project/memory_profiler/

[85] NumPy. Retrieved October 3, 2017 from http://www.numpy.org/

[86] Scipy. Retrieved October 4, 2017 from https://www.scipy.org/

[87] seaborn: statistical data visualization. Retrieved February 10, 2018 from https://seaborn.pydata.org/

[88] SVC sklearn. Retrieved January 25, 2018 from http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

[89] Apache Spark. Retrieved May 20, 2018 from https://spark.apache.org/

[90] TensorFlow. Retrieved May 10, 2018 from https://www.tensorflow.org/