Clasificación

Prefacio

Antes de comenzar con la sección propiamente dicha del tema **clasificación**, cabe aclarar los siguientes conceptos, que a los fines prácticos, espero guíen a una mejor implementación e interpretación de los modelos a través del entendimiento de los datos.

Tipos de variables

Independientemente si hacemos un análisis descriptivo, predictivo o prescriptivo, siempre trabajamos con *variables*.

Una *variable* es una **magnitud** que puede tomar cualquier valor de los comprendidos en un conjunto. Puede adoptar forma numérica o categórica.

- **Variable cuantitativa**: aquella que se expresa de forma numérica. Para las que tiene sentido realizar operaciones aritméticas.
    - Continua: toma infinitos valores dentro de un intervalo (ej: edades, temperaturas, precios, humedad, etc).
    - Discreta: toma finitos valores de un intervalo (ej: cantidad de habitaciones, hijos, etc).
- **Variable cualitativa**: aquella que se expresa en palabras. Sirven para **categorizar** elementos.
    - Nominal: expresa un nombre (ej: paises, animales, colores, etc).
    - Ordinal: expresa diferentes niveles y orden (ej: ranking de satisfacción, posición en competencias, etc).

Parámetros

Es cualquier característica que puede ayudar a definir o clasificar un sistema en particular (es decir, un evento, proyecto, objeto, situación, etc.). Es decir, un parámetro es un elemento de un sistema que es útil, o crítico, al identificar el sistema, o al evaluar su desempeño, estado, condición, etc. (definición extraída de link).

Son las *variables* (del propio modelo matemático, no de los datos) que se estiman durante el proceso de entrenamiento. Por lo que sus valores no los indica manualmente el científico de datos, sino que son obtenidos. Recordemos los parámetros que se estiman en una regresión lineal simple (b0 y b1).

Los parámetros son fundamentales en los modelos de aprendizaje automático. Ya que son aquella parte de los modelos que se formaliza mediante el aprendizaje de los datos y son necesarios para realizar las predicciones. Por lo que estimarlos correctamente es una tarea clave.

También se dividen los modelos en aquellos que tienen un número fijo de parámetros, los cuales se denominan **paramétricos** y que no varían independientemente de si varia el tamaño del set de datos, y por el otro lado, están los modelos cuyo valor de parámetros es variable, por lo que se consideran **no paramétricos.**

La forma más habitual de estimar los parámetros de los modelos es mediante algoritmos de optimización, como, por ejemplo, el descenso del gradiente. Algunos ejemplos de parámetros de modelos de aprendizaje automático son:

- Los coeficientes en una regresión lineal o logística.
- Los pesos en una red neuronal artificial.
- Los vectores de soporte en una máquina de vectores de soporte.

Hiperparámetros

En el aprendizaje automático , un **hiperparámetro** es un parámetro cuyo valor se utiliza para controlar el proceso de aprendizaje, que difieren de los valores de los parámetros que se obtienen a través del entrenamiento.

Los diferentes algoritmos de entrenamiento de modelos requieren diferentes hiperparámetros, algunos algoritmos simples (como la regresión de OLS) no requieren ninguno. Dados estos hiperparámetros, el algoritmo de entrenamiento aprende los parámetros de los datos. Por ejemplo, la regresión LASSO es un algoritmo que agrega un hiperparámetro de regularización a la regresión de mínimos cuadrados ordinarios (penalización L1), que debe configurarse antes de estimar los parámetros a través del algoritmo de entrenamiento.

En las redes neuronales por ejemplo, existe un hiperparametro que controlan el ajuste de los parámetros que se llama *learning rate* (α) o tasa de aprendizaje. Otro puede ser la regularización aplicada para controlar el descenso de gradiente que sea más suave y más rápido: *Momentum, RMSprop, Adam, etc.* Además de la cantidad de neuronas por capa, cantidad de capas ocultas, etc. son otros hiperparametros dentro de las redes neuronales que podemos ajustar.

En el caso de **random forest** podemos ajustar la cantidad de estimadores, es decir la cantidad de árboles de decisión que construyen el algoritmo (*n_estimadores*). Una gran cantidad de estimadores mejoran la predicción pero lo hace más lento debido al requerimiento computacional. Otro es la cantidad máxima de caracteristicas que se considera para dividir un nodo (*max_features*). Hay otros como *n_jobs* que controlan la cantidad de procesadores a utilizar por el algoritmo, entre otros.

Introducción

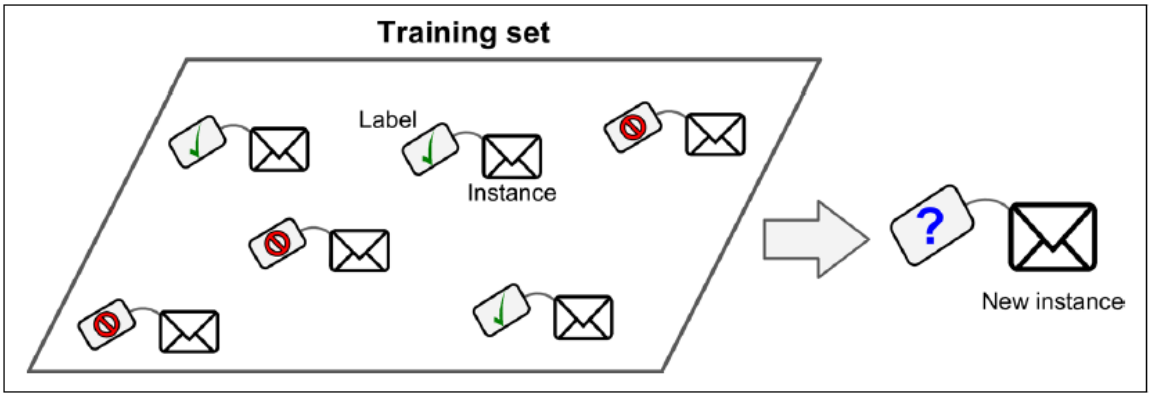
En un problema de *clasificación* se busca asignar automáticamente una etiqueta a un ejemplo sin etiquetar. En machine learning, el problema de clasificación se resuelve mediante un algoritmo de aprendizaje de clasificación que toma una colección de ejemplos etiquetados como entradas y produce un modelo que puede tomar un ejemplo sin etiquetar como entrada y generar directamente una etiqueta o generar un número que puede ser utilizado por el analista de datos para deducir la etiqueta fácilmente.

En un problema de clasificación, una etiqueta es miembro de un conjunto **finito** de clases.

Si el tamaño del conjunto de clases es **dos** (“enfermo”/“saludable”, “spam”/“no_spam”), hablamos de clasificación **binaria** (también llamada binomial en algunos libros).

La clasificación **multiclase** (también llamada multinomial) es un problema de clasificación con tres o más clases.

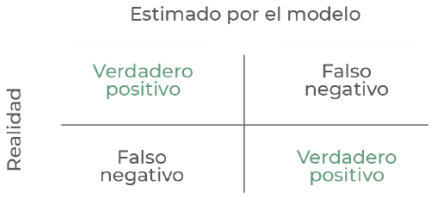
La detección de spam es un famoso ejemplo de clasificación.

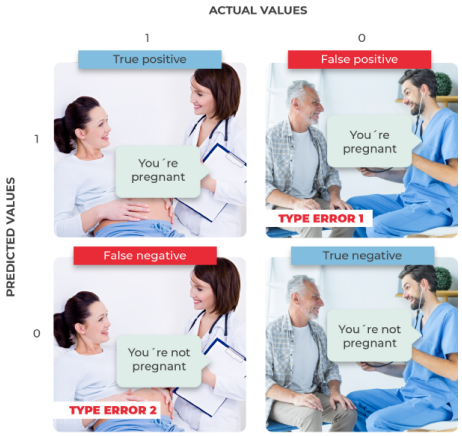


Otro ejemplos son:

- Clasificación entre diabetes vs no diabetes
- Clasificación Fraude vs no fraude
- Clasificación tipo de flor/plantas entre una variedad finita discreta.
- Clasificación de tipo de animales entre una variedad finita discreta.
- Clasificación si el día va estar “lluvioso”, “nublado” o “soleado”.
- Clasificación de mensajes positivos, racistas, negativos, etc.

Métricas de clasificación



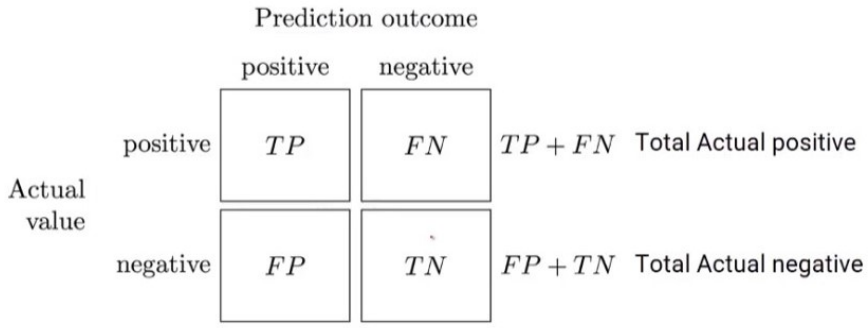


En un problema de clasificación binario (por ejemplo: diagnosticar si una persona está enferma o no) podemos caer en dos tipos de errores.

1. Error tipo 1: es la probabilidad **α** (nivel de significancia), de estimar que la persona no esta enferma, cuando si lo estaba.
2. Error tipo 2: es la probabilidad **β**, de decirle a la persona que está enferma cuando en realidad no lo está.

Teniendo en cuenta esos dos errores que se pueden cometer cuando hacemos predicciones de variables categóricas (**Si/No)** en situaciones que no se puede permitir errores, es que existen las métricas de error, para observar la performance del modelo y ajustarlo de ser necesario.

- **Confusion Matrix**: Esta es una métrica para problemas de clasificación. Es una matriz en la que nxn viene dado por el número de clases. Por ejemplo, en clasificación binaria tendremos matriz de confusión 2x2. Donde las filas representan los valores reales y las columnas representan el resultado de la predicción.

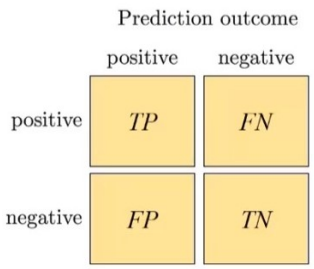


- **Accuracy**: Esta es una métrica **popular** para la tarea de clasificación. La definición formal dice: *es la proporción de valores de predicción correctos sobre el valor total de predicción*.

Accuracy: It is the ratio of correct predicted values over the total predicted values.

$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{Total Predictions}}$$

$$\text{Accuracy} = \frac{TP + TN}{TP + FN + FP + TN}$$

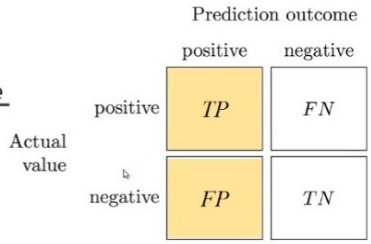


- Alternativas al accuracy (en problemas con dataset desbalanceado):
  - True Positive Rate: TPR = TP/(Tp + Fn) este valor alto es deseable.
  - False Negative Rate: FNR = FN/(Tp + Fn) este valor bajo representa un buen modelo.
  - True Negative Rate: TNR = TN/(Fp + Tn) este valor alto es deseable.
  - False Positive Rate: FPR = FP/(Fp + Tn) este valor bajo es deseable.
- **Precision**: Se define como la proporción de los valores predichos como positivos que en realidad fueron positivos (palabra clave: Predicción).

Out of all the positive predictions, how many are actually positive.

$$\textit{precision} = \frac{\textit{Predictions Actually Positive}}{\textit{Total Predicted positive}}$$

$$\text{precision} = \frac{TP}{TP + FP}$$

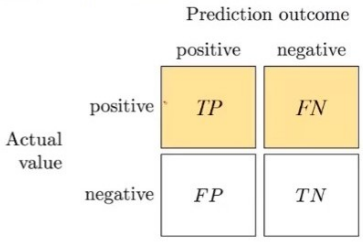


- **Recall**: De todos los positivos reales, ¿cuántos se han pronosticado como positivos?.

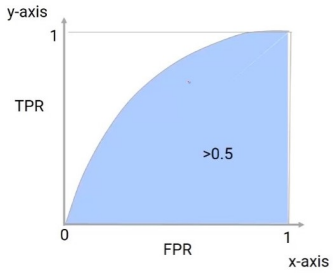
Out of all actual positive, how many are predicted positive.

$$\textit{recall} = \frac{\textit{Predictions Actually Positive}}{\textit{Total Actual Positive}}$$

$$\text{recall} = \frac{TP}{TP + FN}$$



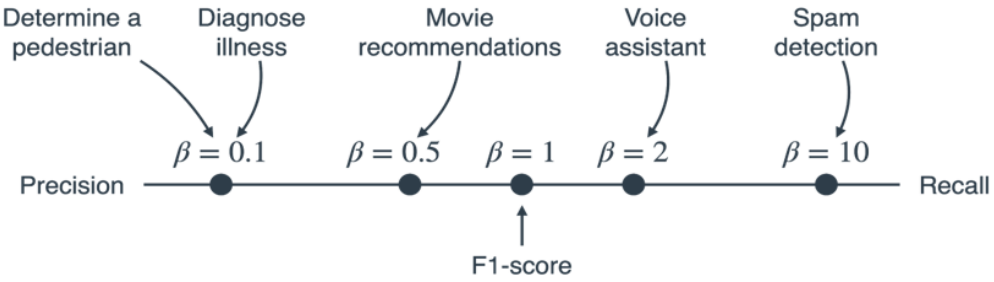
- **AUC-ROC**: respectivamente significan ***a**rea **u**nder **c**urve* y ***r**eceiver **o**perating **c**haracteristic.* Esta métrica es útil para evaluar clasificación **Binaria**. Nos muestra el equilibrio entre los verdaderos positivos y los falsos positivos.



- **F1-score:** es otra medida de performance del modelo entrenado. Tiene en cuenta tanto la *sensibilidad* como la *precisión* . Se puede interpretar como una media armónica de precisión y recuperación, donde una puntuación F1 alcanza su mejor valor en 1 y su peor puntuación en 0. La fórmula para la puntuación F1 es:

$$F_\beta = \frac{1+\beta^2}{\frac{\beta^2}{Recall} + \frac{1}{Precision}}$$

$$F1 = 2 * (precision * recall)/(precision + recall)$$



Pipeline

Una secuencia de componentes de procesamiento de datos se denomina canalización de datos. Las canalizaciones son muy comunes en los sistemas de aprendizaje automático, ya que hay una gran cantidad de datos para manipular y muchas transformaciones de datos para aplicar. Los componentes normalmente se ejecutan de forma asíncrona. Cada componente extrae una gran cantidad de datos, los procesa y escupe el resultado en otro almacén de datos, y luego, algún tiempo después, el siguiente componente en la canalización extrae estos datos y escupe su propia salida, y así sucesivamente. Cada componente es bastante independiente: la interfaz entre los componentes es simplemente el almacén de datos. Esto hace que el sistema sea bastante fácil de comprender (con la ayuda de un gráfico de flujo de datos), y diferentes equipos pueden enfocarse en diferentes componentes. Además, si un componente se descompone, los componentes posteriores a menudo pueden continuar funcionando normalmente (al menos por un tiempo) simplemente usando la última salida del componente averiado. Esto hace que la arquitectura sea bastante robusta. Por otro lado, un componente averiado puede pasar desapercibido durante un tiempo si no se implementa un seguimiento adecuado. Los datos se vuelven obsoletos y el rendimiento general del sistema cae.

Regresión logística

En una regresión lineal, la variable respuesta ***Y*** es del tipo *cuantitativa*. Pero a veces, necesitamos que sea *cualitativa*. Es decir, conocer una respuesta del tipo **categórica**.

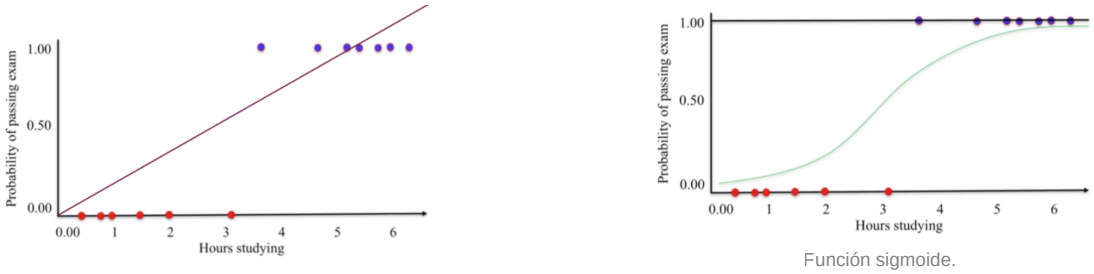
Existen varios enfoques para predecir variables categóricas en el proceso de *clasificación.* Uno de estos enfoques es la regresión logística, que recibe su nombre ya que la respuesta toma dos valores posibles [0,1]

La regresión logística predice una **probabilidad** de pertenecer a una **etiqueta** determinada.

En **regresión lineal** la función de **pérdida** (loss function) viene dada por la suma cuadrática de los residuos (1/n)∑(f(xi)-Yi)^2. Todos los algoritmos basados en modelos (también pueden ser basados en instancias) tienen una función de pérdida, y lo que hacemos en cada modelo es tratar de minimizar la función objetivo que se denomina **función de costo**. Ejemplo en regresión lineal sería el promedio de la pérdida del error cuadrático.

> 💡 En regresión lineal se usa el error cuadrático, por que si se usara el valor absoluto, no se puede calcular la segunda derivada, lo cual hace que la función no sea suave, lo cual resulta inconveniente a la hora de usar algebra lineal para tratar de optimizar el modelo o alcanzar un mínimo de la función de costo

La **regresión logística** como dijimos es un algoritmo de clasificación. El nombre viene de la formulación matemática que se expresa mediante una función no lineal cuyos parámetros son una función lineal.
Se explica aplicada al caso de *clasificación binaria*, pero puede extenderse a una *clasificación multiclase*. En una regresión logística binaria buscamos dos posibles resultados [0,1] que van a ser las categorías. Si usáramos regresión lineal esta puede tomar valores desde -∞ a +∞, lo cual no nos sirve. Una función que tiene las propiedades de tener como límites mínimos y máximos ente [0,1] es la función **sigmoide**.



Función sigmoide.

La función para calcular la probabilidad de que los datos de entrada pertenezcan a una clase u otra se da por medio del calculo de los parámetros de una regresión lineal, a la cual simplemente se le aplica una función **no lineal** como lo es la función *Sigmoide.* De esta manera evitamos que la f(X), si fuese lineal, pudiese tomar valores f(X)>1 o f(X)<0. De esta forma podemos tener valores ajustados entre [0,1] que son las salidas buscadas para **clasificar**.

$$f_{\mathbf{w},b}(\mathbf{x}) \stackrel{\text{def}}{=} \frac{1}{1+e^{-(\mathbf{w}\mathbf{x}+b)}}$$

Para entrenar el modelo se utiliza un criterio de optimización que se denominada **máxima probabilidad**, y acá en vez de disminuir el error como en regresión lineal, ahora maximizamos de la posibilidad de los datos en entrenamiento, acorde al modelo.

$$L_{\mathbf{w},b} \stackrel{\text{def}}{=} \prod_{i=1\ldots N} f_{\mathbf{w},b}(\mathbf{x}_i)^{y_i}\left(1-f_{\mathbf{w},b}(\mathbf{x}_i)\right)^{(1-y_i)}$$

En esta ecuación f(Xi) representa la probabilidad de tomar un valor entre [0,1] ya que buscamos aproximarnos mediante el cálculo de sus parámetros.

La expresión dentro del operador *Pi mayúscula* puede resultar intimidante pero es solo la forma matemática de decir: cuando yi = 1 la probabilidad estará dada por el primer factor, y cuando yi=0 la probabilidad estará dada por el segundo factor.
Se usa el operador Pi en vez de Sigma (como en regresión), ya que la posibilidad de N observaciones de etiquetas para N entradas, es el producto de la posibilidad de cada observación al asumir que son independientes.

De esta forma podemos calcular la probabilidad, de que dados los datos de entrada, pertenezca a una etiqueta u otra.

Y finalmente para corregir el error tenemos en cuenta la probabilidad de ambas categorías dentro de la función de costo:

$$LogL_{\mathbf{w},b} \overset{\text{def}}{=} \ln(L(_{\mathbf{w},b}(\mathbf{x})) = \sum_{i=1}^{N} y_i \ln f_{\mathbf{w},b}(\mathbf{x}) + (1 - y_i) \ln (1 - f_{\mathbf{w},b}(\mathbf{x}))$$

Función de costo de para una clasificación binaria, usualmente se denota con la letra J.

El logaritmo aumenta monótonamente, por lo que maximizar la probabilidad es equivalente a maximizar la probabilidad del logaritmo. Además, se puede hacer uso de la propiedad de logaritmo log(wb)=log(w)+log(b). Muchas ecuaciones se simplifican significativamente porque uno obtiene sumas donde antes tenía productos y así se puede maximizar simplemente tomando derivadas para ver hacia donde crece y decrece una función.

Descenso de gradiente

El método por el cuál se ajustan los parámetros que minimizan el **costo** y **maximizan** la probabilidad de una clase en particular, se denomina **Descenso de Gradiente**. Este método lo veremos con mayor profundidad en redes neuronales.

Este método parte de la premisa de que para una iteración dada, se toma un dato X, se calcula su probabilidad y en base a esto el costo. El gradiente nos permite conocer, para ese valor dado de costo hacia donde crece la función.

Esto se calcula con el vector *nabla* ∇J que equivale a la derivada parcial de J con respecto a la variable que deseemos buscar. En el caso de la regresión lineal o logística serán con respecto a w y b, dado una sola variable de entrada.

Se considera el - ∇J para regresión, ya que necesitamos apuntar hacia donde **decrece** el valor de la función costo dado un parámetro, este lugar se llama **mínimo global** en regresión lineal simple y puede haber **mínimos locales** para otros modelos (redes neuronales).

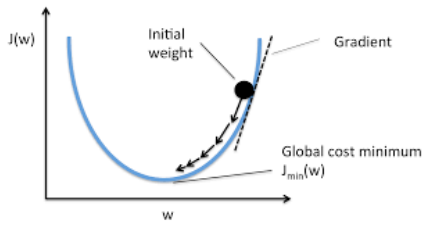


Figura que muestra como el descenso de gradiente ayuda a lograr un mínimo en la función costo ajustando el valor de w.

Entonces, dado el vector gradiente descente de la función costo ∇J podemos calcular las derivadas parciales respecto de w y b, para así poder actualizar dichos parámetros de la siguiente forma:

$$w = w0 - \alpha \delta J / \delta w \qquad b = b0 - \alpha \delta J / \delta b$$

Y mediante el ajuste de los parámetros de la regresión lineal, que es el argumento de la función sigmoide, podemos disminuir el coste (dado una etiqueta en particular) donde la probabilidad sea máxima, que coincide con el mínimo global.

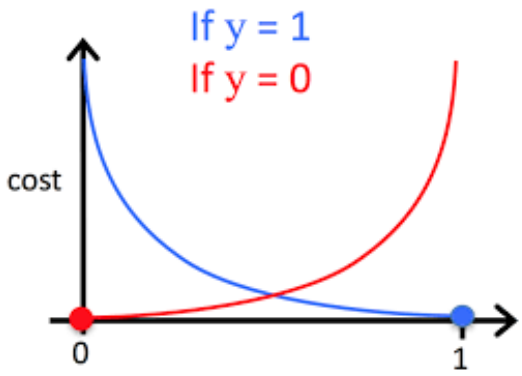


Figura que muestra como dada las etiquetas el descenso del costo se hace por medio del -log de cada función para maximizar la probabilidad de la categoría 0 u 1.

Ventajas

- Fácil de implementar, interpretar y muy eficiente.
- No solo proporciona una medida de cuán apropiado es un predictor (tamaño del coeficiente), sino también su dirección de asociación (positiva o negativa).
- Buena precisión para muchos conjuntos de datos simples y linealmente separables.

Desventajas

- Tiende al overfitting.
- Es difícil obtener relaciones complejas mediante regresión logística.

Naive Bayes

Este modelo esta basado en el **teorema de bayes** y en **probabilidad condicional**.

Primeramente vamos aclarar algunos conceptos:

- Eventos independientes: la probabilidad de que ocurran dos eventos *P(A⋂B)* si ambos son independientes entre sí, entonces P(A⋂B) = P(A) * P(B). Un ejemplo sería tirar una moneda dos veces, donde cada evento es independiente entre si.
- Eventos dependientes: la probabilidad de que ocurran dos eventos *P(A⋂B)* si ambos son dependientes entre sí, entonces P(A⋂B) = P(A/B)*P(B) = P(B/A)*P(A)
- Probabilidad condicional: es la probabilidad de que ocurra un evento, habiendo ocurrido otro, los cuales son dependientes entre si. Entonces el calculo de desprende de la ecuación anterior: P(A/B) = P(A⋂B)/P(B).
- Teorema de Bayes: se deriva de la anterior ecuación para conocer la probabilidad condicionada entre eventos independientes P(A/B) = P(B/A)*P(A)/P(B).

Teniendo en cuenta los conceptos y la ecuación formalizada de **Bayes**, podemos pensar que si queremos aplicar este teorema en clasificación sería de la siguiente forma:

Dada una característica X la probabilidad de que pertenezca a una etiqueta L se puede obtener mediante la siguiente expresión:

$$P(L/X) = P(X/L) * P(L)/P(X)$$

Y acá es donde entra **Naive** que significa *ingenuo*, y esto es porque va a **suponer** que los eventos son independientes entre si (para muchas variables) entonces nos queda de la siguiente forma:

$$P(L/X) = [P(X1/L) * P(X2/L) * P(L)]/[P(X1) * P(X2)]$$

Ejemplo: tenemos los datos de las condiciones climáticas, con los cuales se decidió si un partido de fútbol se jugó y no se jugó.

Las variables del tipo son [*Cielo, Temperatura, Humedad, Viento*] y son del tipo categoricas. Las etiquetas o clases son dos [*No, Si*], por lo cual estamos ante un problema de clasificación binaria.

Cielo	Temperatura	Humedad	Viento	Se jugó
Lluvia	Calor	Alta	No	**No**
Lluvia	Calor	Alta	Sí	**No**
Nublado	Calor	Alta	No	**Sí**
Soleado	Templado	Alta	No	**Sí**
Soleado	Frío	Normal	No	**Sí**
Soleado	Frío	Normal	Sí	**No**
Nublado	Frío	Normal	Sí	**Sí**
Lluvia	Templado	Alta	No	**No**
Lluvia	Frío	Normal	No	**Sí**
Soleado	Templado	Normal	No	**Sí**
Lluvia	Templado	Normal	Sí	**Sí**
Nublado	Templado	Alta	Sí	**Sí**
Nublado	Calor	Normal	No	**Sí**
Soleado	Templado	Alta	Sí	**No**

Para implementar ***Naive Bayes*** primero debemos hacer unos cálculos:

P(Si) = 9/14

P(No) = 5/14

P(Soleado/Si) = 3/9, P(Soleado/No) = 2/5

P(Nublado/Si) = 4/9 , P(Nublado/No) = 0/5

P(Lluvioso/Si) = 2/9, P(Lluvioso/No) = 3/5

P(Calor/Si) = 2/9, P(Calor/No) = 2/5

P(Templado/Si) = 4/9, P(Templado/No) = 2/5

P(Frio/Si) = 3/9, P(Frio/No) = 1/5

P(Alta/Si) = 3/9, P(Alta/No) = 4/5

P(Si/Si) = 6/9, P(Si/No) = 2/5

P(No/Si) = 3/9, P(No/No) = 3/5

Entonces, ahora digamos que queremos saber cual es la probabilidad de **SI** jugar un partido, dado que está lloviendo, hace calor, mucha humedad y no hay viento:

$$P(SI/X) = (2/9 * 2/9 * 3/9 * 3/9 * 9/14)/[(2/9 * 2/9 * 3/9 * 3/9 * 9/14) + (3/5 * 2/5 * 4/5 * 3/5 * 5/14)]$$

$$P(SI/X) = 0.0035273/[0.0035273 + 0.04114]$$

$$P(SI/X) = 0,076$$

Ahora para las mismas características calculamos la probabilidad de que **NO** se jugará:

$$P(NO/X) = (3/5 * 2/5 * 4/5 * 3/5 * 5/14)/[(2/9 * 2/9 * 3/9 * 3/9 * 9/14) + (3/5 * 2/5 * 4/5 * 3/5 * 5/14)]$$

$$P(NO/X) = 0.04114/[0.0035273 + 0.04114]$$

$$P(NO/X) = 0.92$$

Entonces, podemos concluir que para las *condiciones climáticas* propuestas hay un 92% de probabilidad de que no se juega el partido de fútbol.

Ventajas

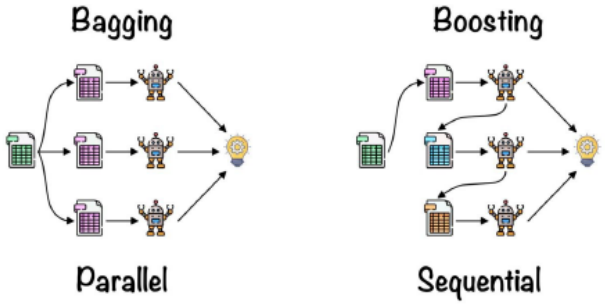
- Este mismo cálculo realiza **Naive Bayes** cuando se implementa en Python con Scikit-Learn. Al ser tan sencillo de calcular resulta en un algoritmo extremadamente rápido de entrenar y de utilizar en para predecir.
- No posee hiperparámetros para ajustar lo cual también le confiere cierta ventaja.
- Con pocos datos de entrenamiento el modelo ajusta bastante bien.
- Así como se mostro en el ejemplo para clasificación binaria también sirve para clasificación multiclase.

Desventajas

- Aunque son clasificadores bastantes buenos, los algoritmos Naive Bayes son conocidos por ser pobres estimadores. Por lo cual hay que tener cuidado a la hora de elegirlos para hacer inferencias.
- La presunción Naive de independencia de los datos muy probablemente no refleja como son los datos realmente.
- Si se introduce una nueva variable que nunca se observó durante el entrenamiento puede ocurrir que cause un mal cálculo ya que no posee frecuencia (zero frecuency).

Métodos de ensamble

Una forma de ajustar un sistema *inteligente* es intentar combinar los modelos que mejor funcionen. El conjunto de modelos a menudo funcionará mejor que el mejor modelo individual. Así es el caso los bosques aleatorios (random forest) funcionan mejor que los árboles de decisión individuales en los que se basan), especialmente si los modelos individuales cometen errores muy diferentes.



Supongamos que hacemos una pregunta compleja a miles de personas aleatorias y luego recopilamos sus respuestas. Seguramente, encontraremos que en promedio las respuestas resultarán mejor que la respuesta de un solo experto. **Esto se llama la sabiduría de la multitud**.

De manera similar, si agregamos las predicciones de un grupo de predictores (como clasificadores o regresores), a menudo obtendremos mejores predicciones que con el mejor predictor individual.

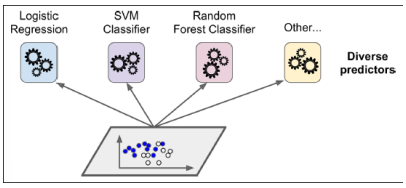
La técnica que agrupa varios predictores se denomina **ensamble** (*Ensemble*) y se entrena mediante el método de *Ensemble Learning*. Por ejemplo, se puede entrenar un grupo de clasificadores de árboles de decisión, cada uno en un subconjunto aleatorio diferente del conjunto de entrenamiento. Para hacer predicciones, simplemente obtenemos las predicciones de todos los árboles individuales, luego se considera la clase que obtuvo la mayor cantidad de votos.

El ensamble de árboles de decisión se denomina **bosque aleatorio** (*random forest*) y, a pesar de su simplicidad, es uno de los algoritmos de aprendizaje automático más potentes disponibles en la actualidad.

Analizaremos los métodos de ensamble más populares, incluidos bagging, boosting, y stacking.

Votación de clasificadores

Supongamos que hemos entrenado algunos clasificadores (regresión logística, svm clasificador, k-vecinos cercanos, bosque aletorio) todos con aproximadamente 80% de accuracy.



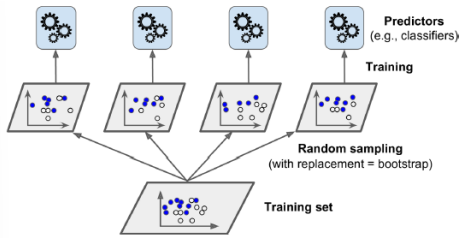
Una forma de lograr un clasificador con un exactitud (*accuracy*) mayor es agrupar las predicciones de cada clasificador y predecir en base a la clase que tenga más votos. Este método se conoce como *hard votting* clasificador. Este método funciona muy bien, aunque los clasificadores que componen el ensamble apenas pasen el 51% de accuracy, al hacer un *ensemble learning* podemos obtener un accuracy de hasta el 75%.

Esto si los clasificadores son independientes, es decir que sean distintos (svm, random forest,etc) o que se no entrenen con el mismo conjunto de datos, ya que si no, son propensos de repetir los mismos errores, lo cual no ayuda a mejorar el accuracy, por ejemplo. Lo ideal es que sean independientes los clasificadores que conformen el ensamble, cada uno con su porción de datos individuales.

Bagging

Entonces, como dijimos un método de lograr que sean independientes es con diversos algoritmos de base o bien, usando el mismo algoritmo para cada predictor, pero entrenar cada uno con sub-porciones del conjunto de entrenamiento de forma aleatoria.

El sub muestreo del dataset para cada predictor, se hace con remplazo, es decir que en cada iteración va a cambiar el subset que se le asigna. De ahí viene el nombre **Bagging** por su nombre en ingles: bootstrap aggregating, agregado de arranque.



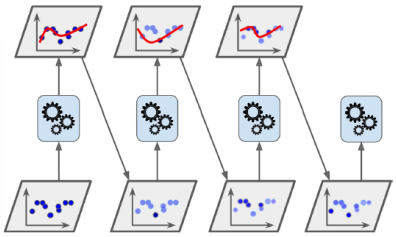
Una vez entrenado, la prediccion de cada predictor se vota, como el método de *hard votting*, y se elije la clase con más votos (la clase con mayor frecuencia). En el caso de regresión numérica se toma el valor promedio de los valores predichos por cada predictor.

Boosting

La idea del boosting, es similar a las anteriores, combinar diferentes algoritmos que performan bajo para mejorar sus predicciones.

Boosting entrena predictores de forma *secuencial*, donde cada uno intenta corregir a su predesor.

Existen mucho métodos de boosting pero los mas populares son **Adaboost** (adaptative boosting) y **Gradient Boosting**.

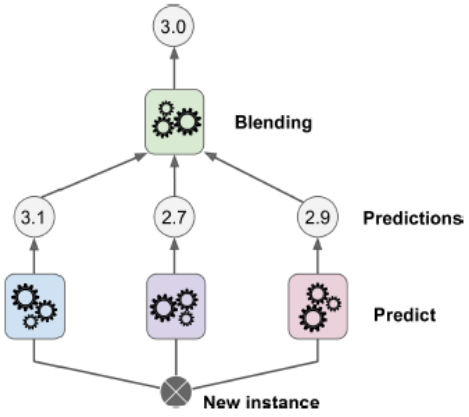


Una forma de que un predictor corrija a su predecesor es prestar un poco más de atención a los datos de entrenamiento que el predecesor no ajustó bien. Esto da como resultado nuevos predictores que se centran cada vez más y más en los casos difíciles. Esta es la técnica utilizada por AdaBoost.
Por ejemplo, para construir un clasificador AdaBoost, se entrena y usa un clasificador de base (árbol de decisión) para hacer predicciones en el conjunto de entrenamiento. Luego se incrementa el peso relativo de los datos de entrenamiento mal clasificados. Un segundo clasificador se entrena usando los pesos actualizados y nuevamente hace predicciones, se actualizan los pesos nuevamente y así sucesivamente. Finalmente para hacer una predicción AdaBoost calcula la predicción de todos los predictores y los pondera usando los pesos actualizados en cada instancia, para así elegir la predicción que tiene más votos.

Stacking

Este método de ensamble va más allá que una simple votación de las predicción de cada predictor. Usa estas predicciones como entrada (input) para un último modelo que devolverá una salida.

Pero para eso debe estar entrenado, para la cual se lo entrena con el método clásico *hold-out* de separar el dataset en entrenamiento, testeo y validación.



Por ejemplo: se divide el dataset en *entrenamiento* y *testeo.* Con los datos de *entrenamiento* entrenamos los predictores base. Una vez entrenados, hacemos predicciones usando los datos de *testeo*. Estas predicciones las usamos como datos de entrada para entrenar el último predictor final *Blending.*

Random Forest

Es un método versátil de aprendizaje supervisado, capaz de realizar operaciones de regresión y de clasificación. Lleva a cabo también:

- Reducción de dimensiones.
- Trata los valores perdidos o faltantes.
- Trata valores atípicos (outliers).
- Realiza otros pasos esenciales de exploración de datos.

Es un tipo de método de aprendizaje por ensamble, donde un conjunto de modelos débiles se combinan para formar un modelo más poderoso.

> La única característica de diferencia a **Random Forest** del método **Bagging**, es que random forest selecciona de forma aleatoria en cada división un subconjunto de variables.

En Random Forest se ejecutan muchos algoritmos de árbol de decisiones en lugar de uno solo, donde, para clasificar un nuevo objeto basado en sus atributos (X), cada árbol de decisión da una clasificación y finalmente la decisión con mayor "votos" es la predicción del algoritmo final.

Este modelo crea múltiples árboles de decisión y los fusiona para obtener una predicción más precisa y estable.

- Tiene casi los mismos hiperparámetros que un árbol de decisión.
- Agrega aleatoriedad adicional al modelo, mientras crecen los árboles, buscando la mejor característica de un "subconjunto" aleatorio de características.

Una gran cualidad del algoritmo *bosque aleatorio* es que resulta fácil medir la importancia relativa de cada característica en la predicción. Scikit-Learn mide la importancia de una característica al observar *cuanto* los nodos del árbol que usan una característica reducen la impureza en todos los árboles del bosque. La impureza en clasificación se da a través de *Gini impurity* que es un valor entre [0;0.5] que se le da a cada característica para saber de forma relativa que tan importante es esa característica para continuar abriendo nodos, con respecto a otras (menor valor de Gini menor impuridad de la información mejor característica a considerar); en regresión esta impuridad se maneja mediante la varianza.

$$GiniIndex = 1 - \sum_j p_j^2$$

Otro criterio en clasificación para evaluar los nodos de decisión que maximicen la ganancia de información, es la *entropy*, cuya matemática es similar a *Gini,* es decir el función de probabilidad de la característica que se esté calculando pero también incluye el logaritmo lo cual le da un rango de valores de [0;1]. Resulta más sensible para muchas características, pero al incluir el logaritmo hace que sea más lento de calculo, por lo que *Gini* resulta más eficiente en el sentido de velocidad de cálculo.

$$Entropy = -\sum_j p_j \cdot log_2 \cdot p_j$$

Calcula esta puntuación automáticamente para cada característica después del entrenamiento y escala los resultados para que la suma de todas las importancias den uno. De esta forma, se pueden destacar características menos importantes, ya que demasiadas características nos puede llevar al overfitting (aunque muy pocas nos llevarían a un underfitting).

Hiperparámetros importantes

Los hiperparámetros en *bosque aleatorio* se usan tanto para aumentar el poder predictivo del modelo, así como, para que sea rápido y eficiente.

1. Hiperparámetros que aumentar el poder predictivo:
    a. n_estimators: cantidad de árboles de decisión que componen el bosque.

b. max_features: número de características que el bosque considera para dividir un nodo.

c. min_sample_leaf: la cantidad mínima de hojas requeridas para dividir un nodo interno.

2. Hiperparámetros que aumentan la velocidad del modelo:

a. n_jobs: se especifica cuantos procesadores se pueden utilizar por el modelo.

b. random_state: toma valores enteros positivos, permite que el modelo sea replicable.

c. oob_score: utiliza el método de reservar datos *out of bag,* fuera de la bolsa de entrenamiento, para luego validar con estos que nunca observó.

Ajuste de hiperparámetros

Una vez que tenemos un modelo de clasificación entrenado para una tarea de clases binarias o multi-clase, lo que hacemos es evaluar su performance.

Para esto usamos las métricas que nos muestran cuan bueno es el poder de generalización del modelo, que tan preciso o sensible es, o bien, cuales variables eran más significativas para obtener el mejor modelo con la cantidad necesaria de variables.

Si las métricas usadas para evaluarlo, nos muestran valores alejados de los que deseamos obtener, existe una técnica que es ajustar las condiciones de entrenamiento para que se alcance un modelo óptimo.

Las condiciones de entrenamiento están dadas por:

- La cantidad de datos que se posee.
- Procesamiento adecuada a los datos que garanticen un adecuado ajuste de parámetros.
- Seleccionar subconjuntos de entrenamiento, validación y testeo que sean representativos de la distribución de clases. Para esto se recomienda usar *K-Folds.*
- Ajustar hiperparámetros propios del modelo elegido.

Este ultimo punto suele ser el más sencillo de encarar, ya que obtener más datos es una tarea a veces casi imposible, el procesamiento seguramente se realizó a conciencia adecuadamente, así como el armado de subconjuntos de entrenamiento, validación y testeo. Por lo cual resulta más accesible y necesario ajustar los hiperparámetros que controlan el proceso de entrenamiento.

Esta tarea se puede acceder de forma manual, es decir, elegir un puñado de hiperparámetros conocidos y variar los valores que pueden tomar, a medida que se realizan métricas para evaluar la performance del modelo con cada combinación de valores de hiperparámetros.

Pero esta tarea puede ser extenuante, por lo que existe una mejor alternativa: **Grid Search**. Este método, como su nombre dice, es una búsqueda grillada de hiperparámetros seleccionados con sus valores. Y lo que hace el modelo es entrenar iterativamente realizando todas las combinaciones posibles de hiperparámetros hasta dar con la combinación que mejores resultados obtenga.

El enfoque de *Grid Search* resulta óptimo cuando se está explorando pocas combinaciones de valores de hiperparámetros, pero cuando el espacio de búsqueda de hiperparámetros es grande, a menudo es preferible utilizar **Randomized Search** en su lugar. Esta búsqueda lo que hace es en lugar de probar todas las combinaciones posibles, evalúa un número determinado de combinaciones aleatorias seleccionando un valor aleatorio para cada hiperparámetro en cada iteración.

Referencias

1. James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An introduction to statistical learning* (Vol. 112, p. 18). New York: springer.
2. VanderPlas, J. (2016). *Python data science handbook: Essential tools for working with data*. " O'Reilly Media, Inc."
3. Géron, A. (2019). *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media.
4. Certificación Universitaria en Data Science. Mundos E - Universidad Nacional de Cordoba.
5. Certificación en Machine Learning - Machine Learning for Beginners. Analytics Vidhya.

Made with ❤️ Ignacio Bosch