**Evaluación de Modelos en Problemas de Clasificación**

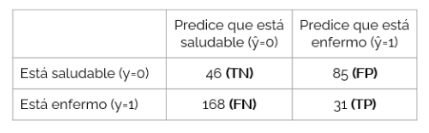
En **Problemas de Clasificación**, el **objetivo** es **predecir** la **pertenencia o no pertenencia** de un caso **a una clase**.

Outcomes: Declaraciones específicas y mensurables que permiten saber si se alcanzó o no los objetivos.

Los Outcomes en una clasificación y en función de la *Tasa de Acierto* pueden dividirse en cuatro clases:

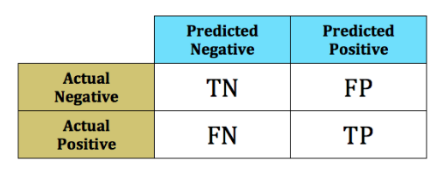
1. **Verdaderos Positivos (TP):** Clase positiva clasificada correctamente.
2. **Verdaderos Negativos (TN):** Clase negativa clasificada correctamente.
3. **Falsos Positivos (FP):** Clase Negativa clasificada como positiva.
4. **Falsos Negativos (FN):** ClasePositiva clasificada como negativa.

Ejemplo conceptual: Tenemos un modelo que determina si un individuo pertenece o no a la clase “Enfermo = 1”:



En este caso, se prefiere evitar los Falsos Negativos (está enfermo pero predecimos que está saludable).

A esta matriz se la llama **Matriz de Confusión¸**y es una tabla de doble entrada donde se describen los resultados reales vs los resultados predichos:

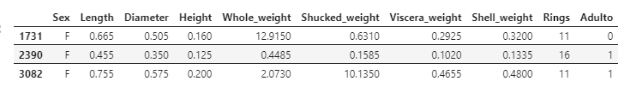
****

Ejemplo: Vamos a trabajar con un Dataset de abulones (especie de moluscos), vamos a buscar predecir si se trata o no de un molusco adulto a partir de sus features (0 = no es adulto; 1 = es adulto).

**En Python:**

df = pd.read\_csv(‘../Data/abalone.csv’, sep = ‘;’)

df.sample(3)

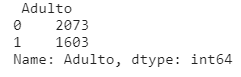


# Armamos la matriz de features y el target:

X = df.drop([‘Adulto’, ‘Sex’], axis = 1)

y = df[‘Adulto’]

y.groupby(y).count()



# Creamos el modelo KNN con 9 vecinos.

from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n\_neighbors = 9)

# Generamos los Datasets de train y test:

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.30, random\_state = 56)

# Ajustamos a los datos de entrenamiento

knn.fit(X\_train, y\_train)

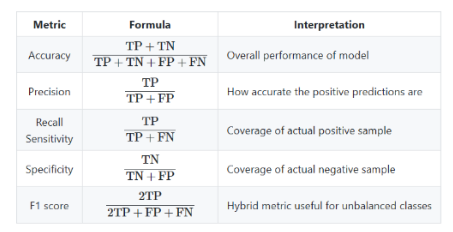
# Predecimos etiquetas para los datos de test:

y\_pred = knn.predict(X\_test)

# Tenemos el modelo listo. Ahora tenemos que usar métricas para evaluar si está performando de acuerdo a lo que necesitamos o no.

**Métricas:**

A partir de la Matriz de Confusión, podemos derivar distintas métricas para evaluar los modelos de clasificación:



**En Python:**

# Con el dataset de testeo, vamos a usar y\_test para compararlo con y\_pred (valores reales vs valores predichos).

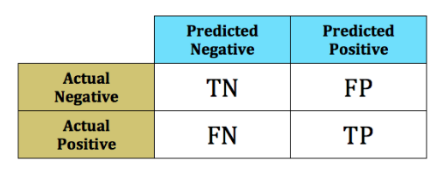
from sklearn.metrics import confusión\_matrix

confusion = confusion\_matrix(y\_test, y\_pred)

print(confusion)



Recordando la matríz de confusión, podemos calcular:

****

TP: 415 (adultos predichos correctamente).

TN: 573 (no adultos predichos correctamente).

FP: no adultos predichos adultos (72).

FN: adultos predichos no adultos (43).

**En Python:**

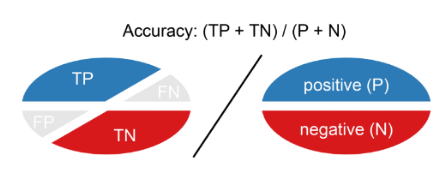
TP = confusión [1,1]

TN = confusión [0,0]

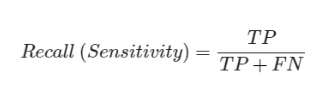
FP= confusión [0,1]

FN= confusión [1,0]

**Accuracy:** Hace foco en la detección de TP y TN. Esto puede ser una desventaja si solo necesitamos enfocarnos exclusivamente en la detección solo de los TP o solo de los TN. Esta métrica no detecta si no tengo casos positivos no predichos. Si tenemos **problemas de clases desbalanceadas,** entonces hay que evitar que los modelos predigan todas las instancias como pertenecientes a la clase mayoritaria. Ejemplo práctico: Default financiero (pocas personas con créditos terminan no pagando sus cuotas)

****

**Recall (Sensitivity/True Positive Rate):** Proporción de positivos correctamente predichos:



Mide la capacidad del modelo de detectar los verdaderos positivos (TP) sobre todos los casos positivos (TP + FN)

**En Python:**

from sklearn.metrics import recall\_score

print(recall\_score(y\_test, y\_pred))

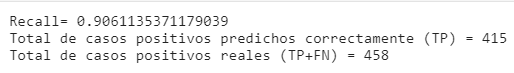


# Otra opción es usar la fórmula:

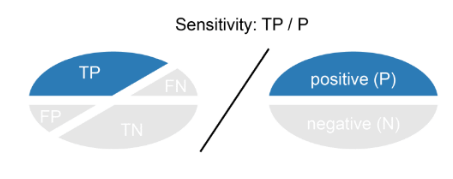
print(‘Recall=’, () TP)/(TP + TN)

print(‘Total de casos positivos predichos correctamente: ‘,(TP))

print(‘Total de casos positivos reales () TP + FN) =’,(TP+TN)

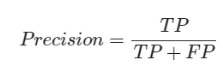


En este caso, el indicador nos dice que del total de casos positivos que hubo, detectamos correctamente el 90%:



Es una medida sensible a Falsos Negativos: En la medida en la que haya más falsos negativos, el recall nos va a ir dando más bajo. A diferencia del accuracy score, se enfoca sólo en los casos positivos. Entonces muestra cómo funciona nuestro modelo en relación al foco de interés de nuestro negocio. Es particularmente útil en aquellos casos en los que la ocurrencia de falsos negativos es inaceptable, como el hecho de que un adulto sea clasificado como un no adulto.

**Precisión:** También llamada **Positive Predictive Value**, es la **proporción de positivos correctamente predichos sobre el total de predicciones positivas**:



Lo que se mide en este caso es qué tan preciso es el clasificador para predecir instancias positivas. Son los Verdaderos Positivos sobre la totalidad de casos positivos predichos (Verdaderos Positivos + Falsos Positivos).

**En Python:**

from sklearn.metrics import precisión\_score

print(precision\_score(y\_test, y\_pred))



# Otro Approach:

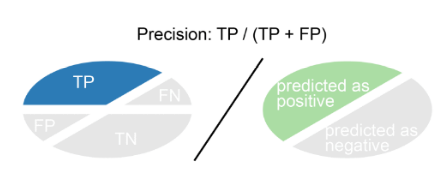
print(‘Recall: ‘,(TP)/(TP + FP))

print(‘Total de casos positivos predichos correctamente (TP) =’,(TP))

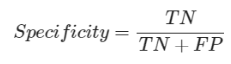
print(‘Total de casos positivos predichos (TP + FP) =’,(TP + FP))



Esto nos está diciendo que de todos los casos clasificados como positivos, el 85% fue bien clasificado. Es sensible a los falsos positivos y a medida que tengamos más falsos positivos, este indicador irá disminuyendo. Nos sirve en aquellos casos en los que queramos estar seguros de nuestros verdaderos positivos.



**Specificity:** También llamado **True Negative Rate** (**TNR**). Proporción de negativos correctamente predichos sobre el total de casos negativos:



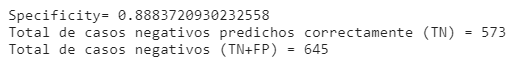
Mide qué tan específico es el clasificador al predecir instancias negativas. Se calcula como los verdaderos Negativos sobre los casos negativos totales (Verdaderos Negativos + Falsos Positivos).

**En Python:**

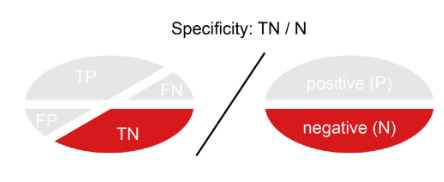
print(‘Specificity =’, (TN)/(TN + FP))

print(‘Total de casos negativos predichos correctamente (TN) =’,(TN))

print(‘Total de casos negativos (TN + FP) =’,(TN + FP))



El 88% de los casos clasificados como negativos fueron clasificados correctamente.



Este indicador es sensible a los falsos positivos. En la medida en la que haya falsos positivos, el indicador irá bajando. Un ejemplo practico en el que es importante es en medicina: Indicar una enfermedad que implica tratamiento con drogas potentes cuando dicha persona no está enferma (falso positivo).

Entonces:

**Recall:**  No podemos aceptar Falsos Negativos



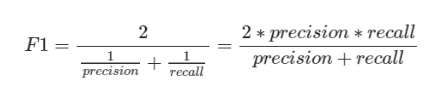
**Specificity:** No podemos aceptar falsos positivos



**Precisión:** Queremos estar seguros de los verdaderos positivos:



**F1-Score:** Media armónica entre Recall y Precisión



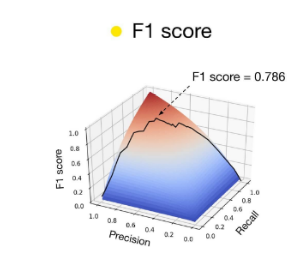
Cuanto mayor es esta métrica, mejor es el modelo. Para que esto ocurra, es necesario que tanto precision como recall sean altos, ya que si alguno es bajo, el resultado de F1 será bajo.

**En Python:**

from sklearn.metrics import f1\_score

print(f1\_score(y\_test, Y\_pred))





Una ventaja que tiene usar la media armónica es que el resultado no es sensible a valores altos de una de las dos variables (precisión o recall). Pero sí en cambio es sensible a los valores extremos bajos, que tienen mucho peso en el resultado final.

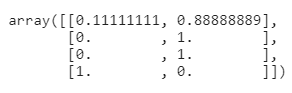
**Umbrales (Thresholds)**: Al realizar las predicciones, se asigna un valor de clase a cada observación. Dichas asignaciones se guardan en y\_pred, siendo 0 en caso de no adulto y 1 en caso de adulto. También es posible predecir **valores de probabilidad de pertenencia a una clase**. En estos casos en vez de 0 o 1, tendremos valores continuos que pueden oscilar entre 0 y 1. Cuanto más cercano a 0 sea dicho valor, más probable será que pertenezca a la clase negativa; cuanto más cercano a 1 sea dicho valor, más probable será que pertenezca a la clase positiva. Con el método **predict\_proba()** es posible acceder a estos valores de probabilidad del modelo.

**En Python:**

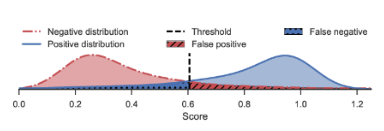
y\_pred\_proba = knn.predict\_proba(X\_test)

El resultado de este método será un array con dos probabilidades para cada instancia del set: p(y=0) y p(y=1); en ese orden. La primer columna es la probabilidad de pertenecer a la clase 0 (negativa); mientras que la segunda columna es la probabilidad de pertenecer a la clase 1 (positiva).

y\_pred\_proba[5:9]

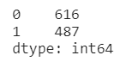


El método **predict** asume que si p(y=1)>0.5 entonces la predicción del modelo será y = 1. El **umbral/threshold** que está usando este método es 0,5.



**En Python:**

pd.Series(y\_pred\_proba[:,1]>0.5).astype(int).value\_counts()



# Es igual a lo que había predicho el método predict.

Pd.Series(y\_pred).value\_counts()



El posible **modificar el umbral** asignando un valor menor o mayor al default (0,5); mientras esté entre 0 y 1 (porque se trata de una probabilidad). Si definimos por ejemplo, el umbral en 0.3, vamos a tener una definición menos rigurosa de adulto. En este caso lo que estamos haciendo es correr el umbral hacia la izquierda. Hay dos maneras de hacer esto en Python: usar la función **binarize** del módulo **preprocessing** de sklearn; o bien generar un booleano y pasarlo a int.

**En Python:**

from sklearn.preprocessing import binarize

y\_pred\_03 = (y\_pred\_proba, threshold = 0.3)[:,1]

y\_pred\_03 = (y\_pred\_proba[:,1]>0.3).astype(int)

#Ahora vamos a comparar las métricas entre los valores de umbrales 0.5 y la nueva recién definida 0.3:

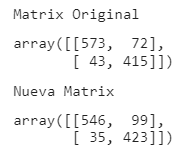
print(‘Matrix original’)

display(confusión\_matrix(y\_test, y\_pred))

print(‘’)

print(‘Nueva Matrix’)

display(confusión\_matrix(y\_test, y\_pred\_03))



# Accuracy:

print(‘Acc umbral 0.5 =’, accuracy\_score(y\_test, y\_pred))

print(‘Acc umbral 0.3 =’, accuracy\_score(y\_test, y\_pred\_03))



# Recall:

print(‘Recall umbral 0.5=’, recall\_score(y\_test, y\_pred))

print(‘Recall umbral 0.3=’, recall\_score(y\_test, y\_pred\_03))



# Precision:

print(‘Precision umbral 0.5=’, precision\_score(y\_test, y\_pred))

print(‘Precision umbral 0.3=’, precision\_score(y\_test, y\_pred\_03))



# Specificity:

def specificity(y\_true, y\_pred):

tn, fp, fn, tp = confusion\_matrix(y\_true, y\_pred).ravel()

sp = tn / (tn + fp)

return(sp)

print(‘Spec umbral 0.5 =’, specificity(y\_test, y\_pred))

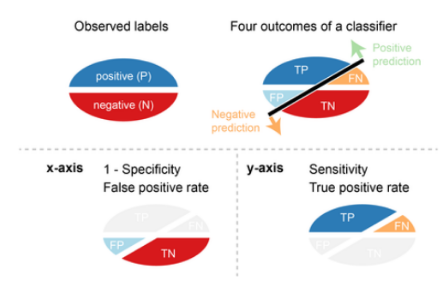
print(‘Spec umbral 0.3 =’, specificity(y\_test, y\_pred\_03))



Modificar un umbral nos sirve para buscar un valor tal que maximice la cantidad de verdaderos positivos y verdaderos negativos, y minimice la cantidad de falsos positivos y falsos negativos. En función del problema que estemos analizando, vamos a modificar el umbral de forma tal que nos ayude a detectar positivos o negativos, según lo que necesitemos y lo que más nos convenga.

**Curva ROC y AUC (Área bajo la Curva):**

La **Curva ROC** evalúa la relación entre **specificity** (o su inversa, 1 – FPR, tasa de falsos negativos) y **sensitivity o recall** (también llamada **TPR, Tasa de Verdaderos Positivos**).



Se estiman specicity y sensitivity de un modelo para distintos umbrales de decisión. Por ejemplo, consideramos 4 matrices de confusión, resultantes de elegir 4 umbrales de decisión (de más a menos exigentes: umbral 1 = 1; umbral 2 = 0.75, umbral 3 = 0.5, umbral 4 = 0).

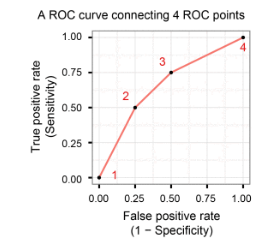
Con umbral 1, un caso se va a considerar positivo sólo cuando su valor de probabilidad sea mayor que 1; entonces todos los casos se van a clasificar como negativos, maximizando specificity en 1.

Con umbral 4, un caso se va a considerar positivo cuando su valor de probabilidad sea mayor o igual que 0; entonces todos los casos se van a clasificar como positivos, maximizando recall o sensitivity en 1.

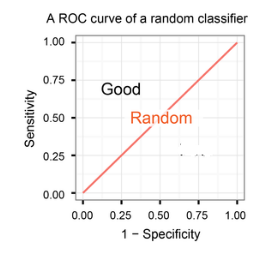


El aumento en el umbral nos vuelve más flexible, ganando sensitivity a costa de perder specificity.

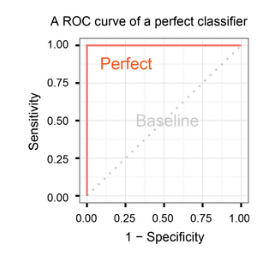
La curva ROC grafica sensitivity en el eje y, y FPR (Fale Positive Rate, que es 1 – Specificity) en el eje X. El motivo por el cual se pone FPR en vez de specificity en X es para que los dos ejes del gráfico aumenten en forma positiva.



En un **modelo random**, los Verdaderos Positivos serían iguales a los falsos positivos para todos los umbrales:

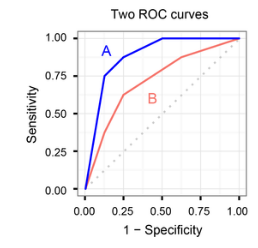


En el **modelo perfecto**, se **maximizaría tanto** la **sensitivity** como la **specificity:**

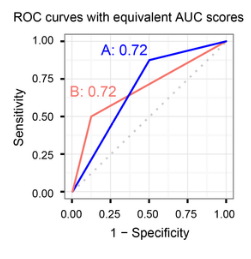


En general, los buenos modelos van a estar en algún lugar entre random y el modelo perfecto.

La **gran ventaja** de la Curva ROC es que **permite comparar visualmente la performance de dos modelos** a partir de su sensitivity y su specificity. En este gráfico de ejemplo, el modelo A es mejor que el modelo B:



Dos modelos pueden tener un mismo AUC (área bajo la curva), pero ser distintos al visualizar su curva ROC:



**En Python:**

from sklearn.metrics import roc\_curve

fpr\_log, tpr\_log, thr\_log = roc\_curve(y\_test, y\_pred\_proba[:,1])

df = pd.DataFrame(dict(fpr = fpr\_log, tpr=tpr\_log , thr = thr\_log))

plt.axis([0, 1.01, 0, 1.01])

plt.xlabel(‘1-Specificity’)

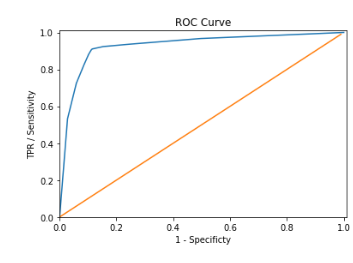
plt.ylabel(‘TPR / Sensitivity’)

plt.title(‘ROC Curve’)

plt.plot(df[‘fpr’], df[]’tpr’])

plt.plot(np.arange(0.1, step = 0.01), np.arange(0.1, step = 0.01))

plt.show()



Con la función **auc()** del **módulo metrics** de **sklearn**, es posible calcular la AUC. Para esto va a usar los valores de TPR y FPR generados al usar la función ROC\_Curve. El primer argumento corresponde a los valores de FPR y el segundo a los valores de TPR. El output será la AUC.

**En Python:**

from sklearn.metrics import auc

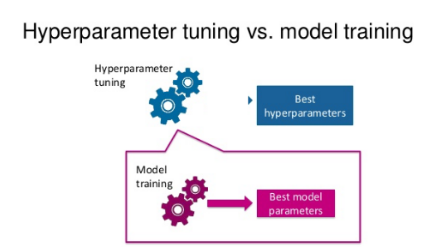
print(‘AUC=’, auc(fpr\_log, tpr\_log))



**Grid Search y Random Search:**

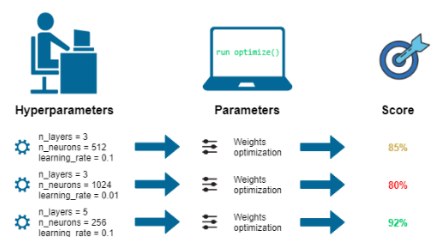
**Hiperparámetros:** **Características externas** de un **modelo**. Deben **definirse** al implementar el modelo, **antes de entrenarlo**. IE: el valor *K* del algoritmo KNeighborsClassifier.

**Parámetros**: **Características internas** de un **modelo**. Se estiman a partir del entrenamiento con los datos. IE: Coeficientes de una regresión lineal.



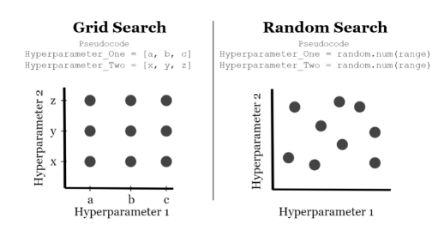
No podemos saber de antemano cuáles son los mejores valores de hiperparámetros para resolver un problema determinado. Puede que una configuración determinada de hiperparámetros funcione bien en un conjunto de datos y horrible en otro similar. La llamada **hyperparameter tuning** (optimización o ajuste de hiperparámetros) es una **parte esencial del Machine Learning**.

Los modelos pueden tener muchos hiperparámetros. **Encontrar la mejor combinación** nos lleva a un **problema de búsqueda de información**.



Para hacer esto, contamos con **dos métodos de** **Hyperparameter Tuning:**

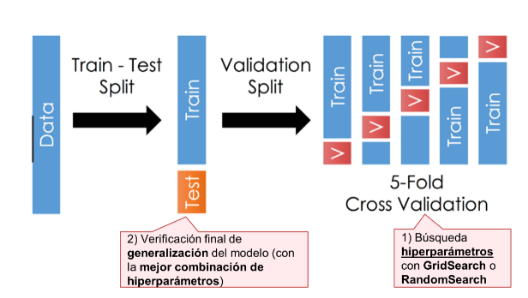
1. **Grid Search:** **Busca la mejor combinación** de Hiperparámetros **dentro de una** **Grilla** (grid) **especificada previamente**. Hace una búsqueda exhaustiva para cada valor de la Grilla.
2. **Random Search**: **Selecciona** en forma **aleatoria** un **subset de los hiperparámetros.**



Evaluar todas las combinaciones de valores de los hiperparámetros tiene un **costo de cómputo**. Esto nos llevará en algunos casos a trabajar con una **Grilla Reducida**, o usar **Random Search** que selecciona un **subset de combinaciones**.

Por cada combinación de valores de hiperparámetros, ambos métodos hacen lo siguiente:

* Los aplican sobre el dataset de train.
* Los evalúa con *Cross Validation.*
* Registra el Score.
* Una vez hecho esto para todas las búsquedas, selecciona la combinación con más alto score.
* La aplica sobre Train.
* Predice sobre Test.



Cuestiones a tener en cuenta a la hora de implementar sklearn sobre hiperparámetros:

* Un **estimador** (modelo sobre el cual trabajaremos). IE: Modelo KNN.
* Un **espacio de parámetros** donde hacer la búsqueda. IE: n\_neighbors con rango de valores entre 1 y 30.
* Un **método de búsqueda** sobre los modelos candidatos (GridSearch o RandomSearch).
* Un **esquema de validación cruzada**, seleccionando la **cantidad de folds**. IE: Cross Validation con 10 folds.
* La **métrica de evaluación** para elegir el mejor modelo. IE: Maximizar accuracy.

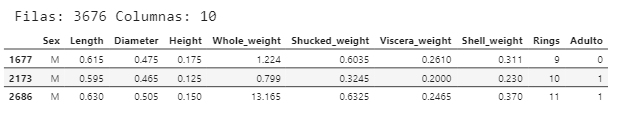
**En Python:**

# Vamos a aplicar GridSearch y RandomSearch sobre el dataset de abalones:

df = pd.read\_csv(‘../Data/abalone.csv’, sep = ‘;’)

print(‘Filas: ’, df.shape[0], ‘Columnas ’, df.shape[1])

df.sample[3]



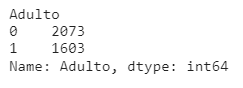
# Armamos la matriz de features (X) y el target (y):

X = df.drop([‘Adulto’, ‘Sex’], axis = 1)

y = df[‘Adulto’]

# Vemos la distribución de los labels:

print(y.groupby(y).count())



# Hacemos el split entre train y test:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 30, random\_state = 56)

print(X\_train.shape, X\_test.shape, y\_train.shape, y\_test.shape)



# Ahora vamos a estandarizar las features, ya que las escalas entre algunas de ellas son distintas:

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X\_train = pd.DataFrame(scaler.fit\_transform(X\_train))

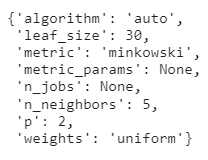
X\_test = pd.DataFrame(scaler.fit\_transform(X\_test))

# Vamos a usar el método **get\_params** para conocer todos los hiperparámetros de los modelos. Vamos a ver por ejemplo, los hiperparámetros del modelo KNN:

from sklearn.neighbors import KNeighborsClassifier

model = KNeighborsClassifier()

model.get\_params()



# Antes de usar RandomSearch o GridSearch, conviene realizar una prueba manual con distintos valores para los hiperparámetros a evaluar. Por ejemplo, vamos a calcular el valor de accuracy para distintos valores K del hiperparámetro n\_neighbors, para de esta forma conocer su valor óptimo. Para la evaluación usaremos el método de Cross Validation. Usando el método **cross\_val\_score** podemos evaluar los resultados de una métrica en un esquema de validación cruzada. Con el parámetro **scoring=’accuracy’** podemos indicar la métrica. Usamos el parámetro **cv=10** para definir **10 folds**. Podemos reemplazar este parámetro usando previamente el método **StratifiedKFold**, que asegura que los 10 folds cuenten con casos seleccionados en forma aleatoria manteniendo la distribución original de las etiquetas.

#n\_splits = Permite determinar la cantidad de folds

# random\_state = Permite definir una semilla que asegura reproductividad.

# shuffle = asegura que las particiones se harán en forma aleatoria.

from sklearn.model\_selection import StratifiedKFold

folds = StratifiedKFold(n\_field = 10, random\_state = 19, shuffle = True)

# Con un for podemos recorrer distintos valores de K, evaluar los scores de la validación cruzada de los datos de train y guardar el promedio de los scores:

from sklearn.model\_selection import cross\_val\_score

k\_range = list(range(1, 31))

k\_scores = []

for k in k\_range

knn = KNeighborsClassifier(n\_neighbors = k)

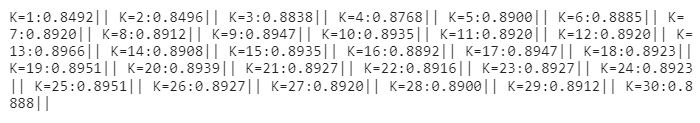
scores = cross\_val\_score(knn, X\_train, y\_train, cv = folds, scoring = ‘accuracy’)

k\_scores.append(scores.mean())

# Entonces en la lista k\_scores encontramos los valores promedio de accuracy para los distintos valores de k.

for i, value in enumerate(k\_scores):

print(‘K=’, i+1, ‘:’,’%.4f’%value,’||’end=’ ‘,sep=’’)



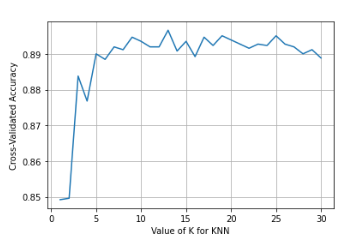
# Vamos a graficar los resultados:

plt.plot(k\_range, k\_scores)

plt.xlabel(‘Value of K for KNN’)

plt.ylabel(‘Cross-Validated Accuracy’)

plt.grid()



El k óptimo es 13, con un score de validación de 0.8966.

Con el método **GridSearchCV**, podemos implementar en forma automática la búsqueda y análisis de los hiperparámetros que realizamos en forma manual:

* **Estimador:** Modelo instanciado.
* **param\_grid** es la grilla de parámetros para recorrer. Se trata de un diccionario con claves (nombre hiperparámetro, lista de valores).
* **cv** = cantidad de folds para la validación cruzada.
* **Scoring** = métrica a evaluar.
* **n\_jobs** = número de procesadores que corren en paralelo. Este es opcional, pero es importante porque las búsquedas exhaustivas pueden consumir mucha CPU.

**En Python:**

from sklearn.model\_selection import GridsearchCV

# Estimador: El modelo instanciado.

knn =KNeighborsClassifier()

# Espacio de parámetros: Grilla de parámetros que queremos testear. Diccionario con las claves nombre = rango de valores.

k\_range =list(range(1,31))

param\_grid = dict(n\_neighbors=k\_range)

print(param\_grid)



# Esquema de Validación cruzada. Definimos la cantidad de folds con el método StratifiedKFold

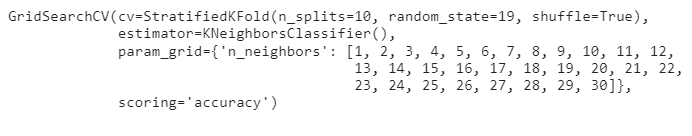
folds = StratifiedKFold(n\_splits = 10, random\_state = 19, shuffle = True)

# El método de búsqueda es GridSearchCV. La métrica de evaluación es Accuracy.

grid = GridSearchCV(knn, param\_grid, cv=folds, scoring=’accuracy’)

# Entrenamos el modelo usando el método .fit de grid. Puede que tarde mucho porque hace una búsqueda exhaustiva.

grid.fit(X\_train, y\_train)



# El método **best\_estimator\_** de grid selecciona el mejor modelo. Informa el número óptimo de vecinos; es similar a lo observado en el proceso manual:

grid.best\_estimator\_



# El método **best\_score\_** de grid nos indica cuál fue la performance promedio del score de validación del grid search:

grid.best\_score\_



El método **best\_params\_** de grid nos muestra los valores seleccionados de los hiperparámetros:

grid.best\_params\_



# Ahora vamos a ver cóm funciona el modelo en los datos de testeo. Con el método **predict** de **GridSearchCV** aplicado al objeto grid que usa el modelo entrenado con los datos de train y la mejor combinación de hiperparámetros, podemos usar el modelo para que haga predicciones sobre los datos de testeo:

y\_pred\_grid = grid.predict(X\_test)

# Vamos a ver la Matriz de Confusión:

confusion = confusion\_matrix(y\_test, y\_pred\_grid)

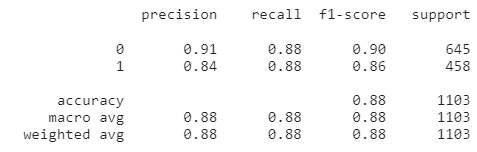
print(confusion)



# Podemos conseguir un resumen de las principales métricas de evaluación usando **classification\_report**, que podemos importar de sklearn.

from sklearn.metrics import classification\_report, confusión matrix

print(classification\_report(y\_test, y\_pred\_grid))



# Podemos ver las métricas por label de la clase (0 o 1). Vemos que performa mejor para la clase negativa que para la positiva.

**accuracy** es 0.88.

**support** indica el total de casos por etiqueta (sería un equivalente a un value\_counts)

**macro avg:** Media de cada métrica de evaluación para el total de casos.

**weighted avg:** Idem macro avg, pero calculando la **media ponderada** por support (teniendo en cuenta la cantidad de casos de cada clase). Toma en cuenta el potencial **desbalanceo de clases**. En el caso de este ejemplo, dan similares porque no tenemos clases desbalanceadas.

# Ahora vamos a usar el otro método: RandomizedSearchCV. La diferencia contra GridSearch es que en lugar de explorar todo el espacio de hiperparámetros, lo que hace es explorar **combinaciones al azar**. Puede tener sus ventajas al trabajar con datasets muy pesados, o espacios de hiperparámetros muy grandes.

from sklearn.model\_selection import RandomizedSearchCV

# Los parámetros de RandomizedSearchCV son los mismos que los de GridSearchCV, más el parámetro **n\_iter** que indica el número de combinaciones de hiperparámetros que queremos seleccionar al azar de la grilla que definamos. Por default, el valor de este parámetro es 10.

# Volvemos a armar la grilla de hiperparámetros.

k\_range = list(range(1,31))

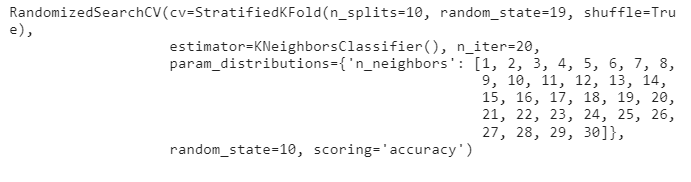
param\_grid = dict(n\_neighbors = k\_range)

print(param\_grid)



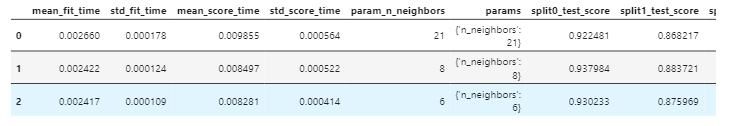
# Ahora realizamos el tuneo con combinaciones aleatorias de k:

random = RandomizedSearchCV(knn, para\_grid, n\_iter = 20, cv = folds, scoring = ‘accuracy’, shuffle = True)



# Ahora analizamos los resultados, usando los mismos métodos que habíamos usado con GridSearch:

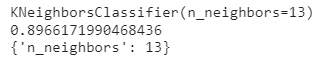
pd.DataFrame(random.cv\_results\_).head(3)



print(random.best\_estimator\_)

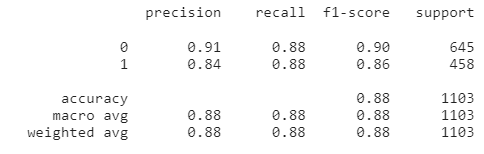
print(random.best\_score\_)

print(random.best\_params\_)



# RandomizedSearchCV obtuvo el mismo resultado que GridSearchCV, sin probar tantas combinaciones.

print(classification\_report(y\_test, random.predict(X\_test)))



confusion = confusion\_matrix(y\_test, random.predict(X\_test))

print(confusion)



**Resumen:**

**Métricas:**

El objetivo en problemas de clasificación es predecir la pertenencia o la probabilidad de pertenencia de un caso a una clase.

Las métricas de evaluación de desempeño de estos modelos son:

* Matriz de Confusión
* Accuracy
* Sensitivity o recall
* Specificity
* Precision
* F1-Score
* Curva ROC
* AUC (Area Under Curve)

Estas métricas ayudan a entender qué tan bien nuestro modelo está identificando la pertenencia de un caso a una clase. Cada una de las distintas herramientas o métodos nos permite hacer foco en distintos aspectos de esta performance y evaluar cuáles son los tipos de errores que más se está cometiendo.

Estas métricas también nos sirven para comparar distintos modelos que entrenemos, y decidir cuál es el óptimo para usar en función de nuestro problema.

**Grid Search:**

**Hiperparámetros:** Características externas a un modelo que no se aprenden de forma directa a partir del entrenamiento con datos. Deben definirse previamente a dicho entrenamiento. Son muy importantes a la hora de entrenar un modelo, porque impactan en su desempeño. No se puede saber de antemano cuales son los valores más convenientes.

**Parámetros:** Características o propiedades internas de un modelo. Sus valores se estiman a partir del entrenamiento con datos.

Tenemos dos grandes métodos o procesos para buscar los mejores hiperparámetros (hyper-parameter tuning), entre otros.

* **GridSearch**: Hace una búsqueda exhaustiva para cada valor de una grilla de hiperparámetros y elige la combinación entre los mismos que maximice una métrica determinada.
* **RandomizedSearch**: Hace la mejor búsqueda de hiperparámetros a partir de seleccionar en forma aleatoria un subset de hiperparámetros. Esto achica el espacio de búsqueda y reduce el tiempo de cómputo.

**Pasos a seguir para buscar hiperparámetros:**

1. Elegir un **estimador.**
2. Elegir un **espacio de Hiperparámetros**.
3. Elegir un **método de búsqueda (RandomSearch / GridSearch)**
4. Definir un esquema de **validación cruzada**.
5. Definir una **métrica de evaluación**.