

Guia de ejercicios

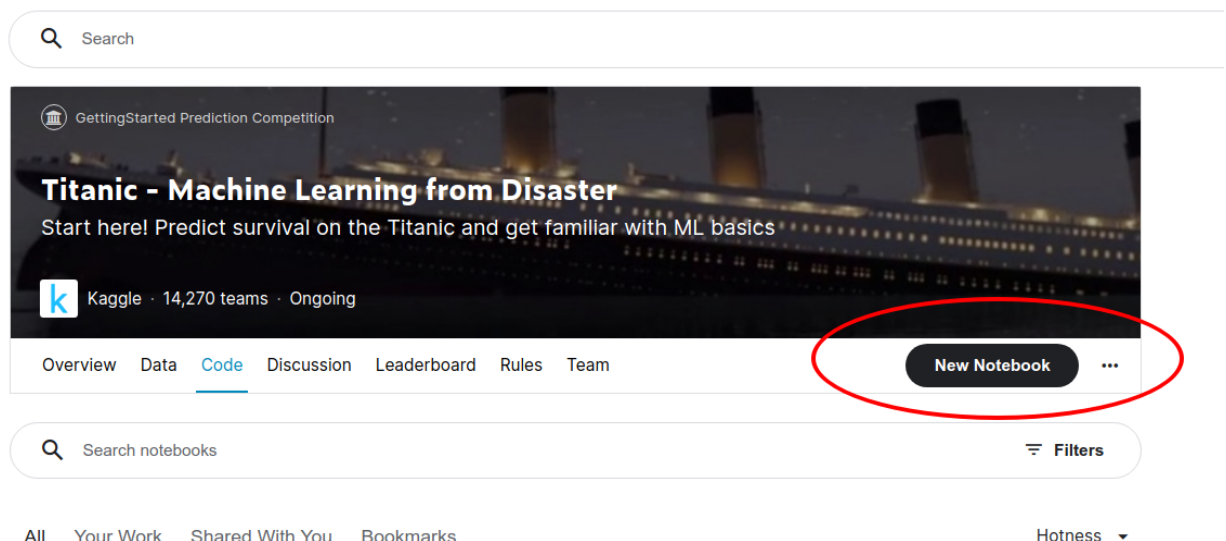
Visualización de datos con Pandas

Sobrevivientes del Titanic

La idea es trabajar con la visualización de datos, utilizando este conjunto de datos provisto por Kaggle. Se trata de la lista de pasajeros reales del Titanic, indicando: sexo, edad, costo del pasaje, número y clase de cabina, etc. e indicando si sobrevivió o no.

Pueden encontrar aquí una descripción de los datos: <https://www.kaggle.com/c/titanic/data>

1. Ingresar a Kaggle, y en la sección “Code” para este conjunto de datos crear una nueva “Notebook” jupyter de trabajo: <https://www.kaggle.com/c/titanic/code>



2. Utilizando Pandas obtener los siguientes valores y visualizaciones:
 - a. Mostrar las primeras 5 filas incluyendo la columna:

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

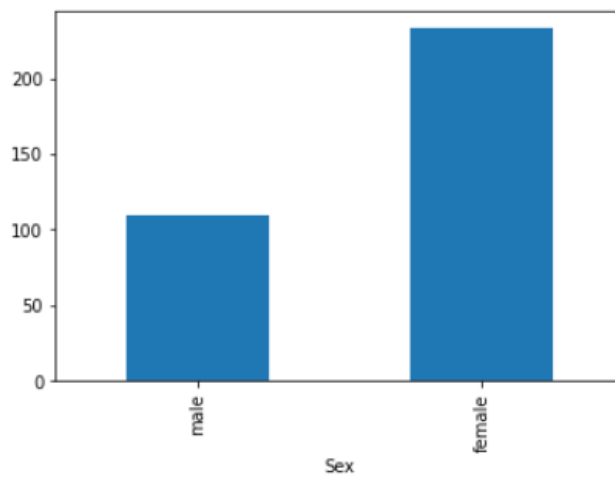
b. Cantidad de pasajeros por género:

```
Sex
female    314
male      577
dtype: int64
```

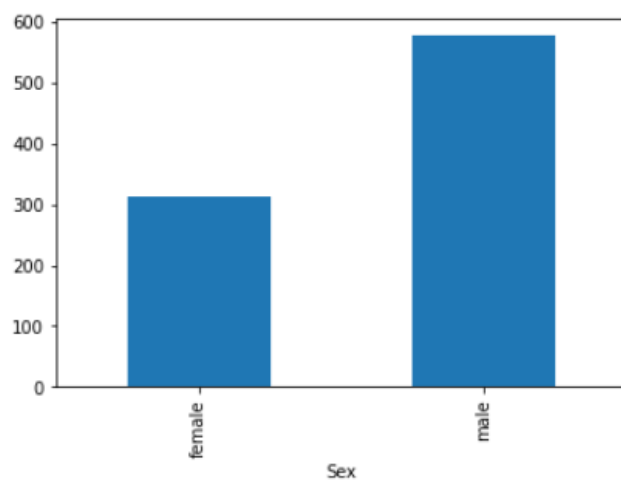
c. Cantidad de pasajeros que sobrevivieron por género:

```
Sex
male      109
female    233
Name: Survived, dtype: int64
```

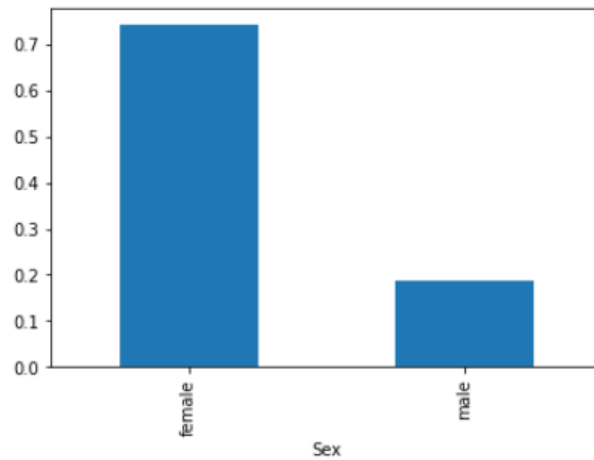
d. Graficar pasajeros totales por género:



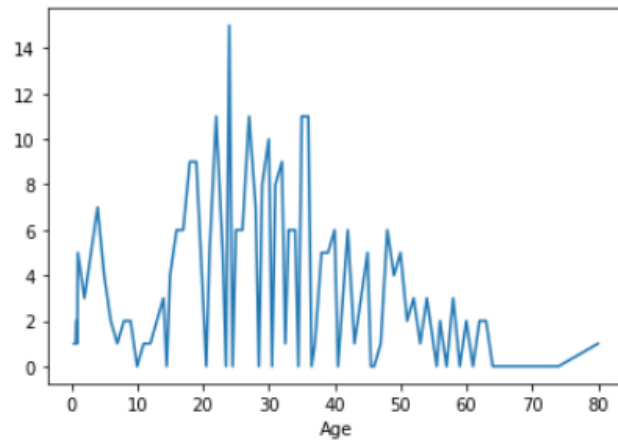
e. Graficar pasajeros sobrevivientes por género:



f. Graficar % de pasajeros sobrevivientes por género:



g. Sobrevivientes por edad

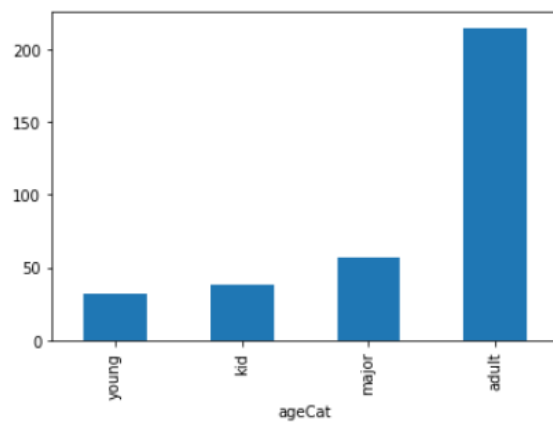


h. Crear una nueva columna que se una categoría, utilizando la edad y estos criterios:

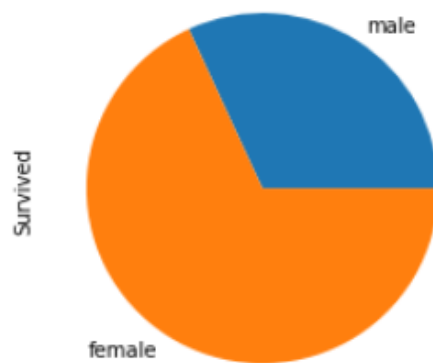
```
Age > 0 and 'Age <= 10:      return 'kid'
Age > 10 and Age <= 18:      return 'young'
Age > 18 and Age <= 60:      return 'adult'
else return 'senior'
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	ageCat
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S	adult
1	2	1	1	Cummings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C	adult
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S	adult
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S	adult
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S	adult

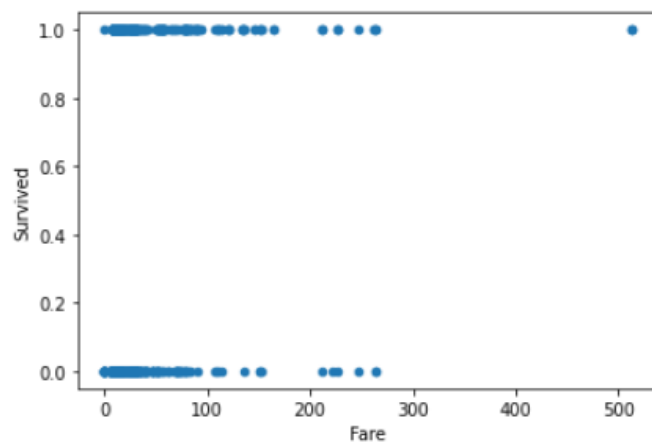
- i. Graficar la nueva columna, ordenada por barras según los sobrevivientes



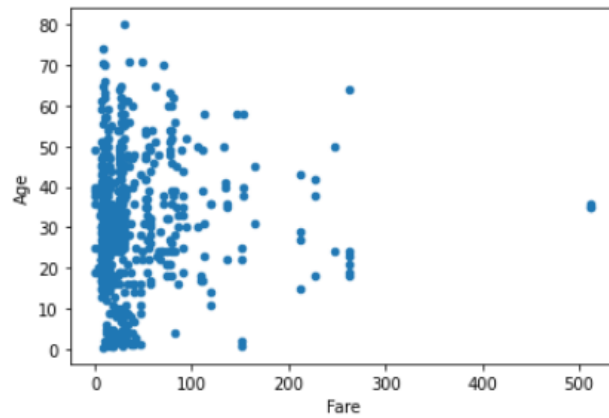
- j. Graficar los sobrevivientes por género en un gráfico de torta:



- k. Graficar correlación entre las variables "Survived" y "Fare"



I. Graficar correlación entre las variables “Age” y “Fare”



3. Ejercicios adicionales:

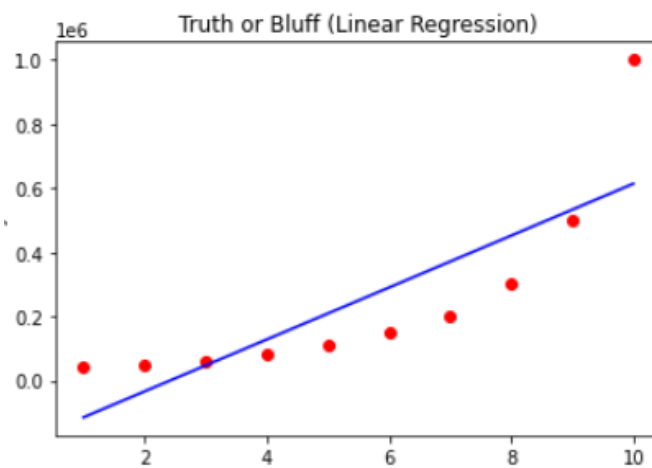
- a. Graficar cantidad de sobrevivientes por categoría (**Pclass**)
- b. Graficar % sobrevivientes por categoría (**Pclass**)
- c. Mostrar un gráfico de torta el % de sobrevivientes por categoría
- d. Verificar si hay correlación entre la edad y la categoría

Linear regression y Polynomial regression

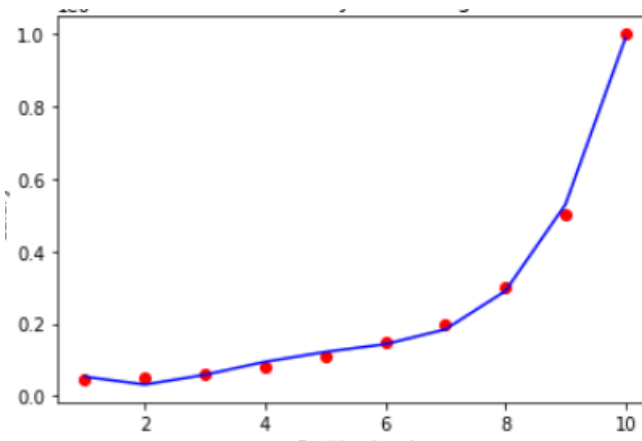
1. Copiar y pegar los siguientes valores para X e Y respectivamente:

```
x = np.array([[ 1],[ 2],[ 3],[ 4],[ 5],[ 6],[ 7],[ 8],[ 9],[10]])
y = np.array([ 45000, 50000, 60000, 80000, 110000, 150000,
200000,300000, 500000, 1000000])
```

2. Graficar los puntos.
3. Entrenar un modelo de regresión lineal para encontrar la recta que mejor ajuste y graficar



4. Entrenar un modelo de regresión polinomial y graficar. Utilizar:
`from sklearn.preprocessing import PolynomialFeatures`
Usar grado 4 y la función:
`fit_transform`



Regresión Logística

Sobrevivientes del Titanic

1. Utilizando nuevamente el conjunto de datos de sobrevivientes del titanic, calcular la regresión logística entre las variables "Age" y "Survived".
 - a. Usar la biblioteca:

```
from sklearn.linear_model import LogisticRegression
```
 - b. Para poder utilizar la función, Fit hay que verificar dos cosas:
 - i. Que no haya datos NaN
 - ii. Que cada elemento en la fila X, sea un vector.
2. Graficar la curva de regresión para el valor 1 de "Survived"
 - a. Utilizar
 - i. `import matplotlib.pyplot as plt`
3. Graficar con otro color la curva de regresión para el valor 0 de "Survived"
 - a. Podría quedar algo así:
4. ¿Qué tipo de relación hay?
5. Agregar una nueva variable: "Fare" al entrenamiento del modelo
6. Volver a graficar, pero esta vez tomamos un corte del modelo 3D: solo la edad 20 años. De forma que en el eje X queden los valores posibles del "Fare", de 0 al máximo Fare existente. En el eje Y la probabilidad de sobrevivir o morir que es calculada por el modelo regresión logística, pero solo para la edad de 20, esta variable queda fija.
7. ¿Cómo queda ahora la curva?

K-Means



Olivetti Faces

1. Cargar el conjunto de datos que ya viene con la biblioteca de sklearn:

```
sklearn.datasets.fetch_olivetti_faces()
```

https://scikit-learn.org/stable/modules/generated/sklearn.datasets.fetch_olivetti_faces.html

(Cada imagen se aplanar en un vector de 4096 valores de entre 0 y 1)

2. Dividirlo en 3 conjuntos:
 - a. Validacion
 - b. entrenamiento
 - c. Prueba
3. Entrenar K-Means. Son 40 personas diferentes
4. Graficar las caras por grupo, según lo haya agrupado K-Means
5. ¿Hay caras similares?

Clasificación de textos y análisis de sentimientos

IMDB - WEKA

Descargar el siguiente conjuntos de datos:

- Críticas etiquetadas de IMDB: <https://goo.gl/dwCgoe>

Se trata de un conjunto de críticas cinematográficas extraídas de IMDB. Son 2000 archivos de texto previamente etiquetados como positivos y negativos. Hay 1000 archivos (es decir críticas) positivas y 1000 críticas negativas.

De ese conjunto el 66% se utilizará para entrenar un clasificador y el 33% restante se utilizará para validar su precisión y exhaustividad.

Se deberán entrenar los siguientes métodos de clasificación:

- Bayes Naïves.
 - Bayes Naïves Multinomial.
1. Indicar la precisión y exhaustividad de cada método de clasificación.
 2. Indicar la medida F1 de cada método de clasificación.
 3. Indicar la matriz de confusión.
 4. Buscar en la documentación de WEKA la diferencia entre ambos métodos.

Ver en el anexo de la guía de ejercicios como utilizar WEKA

IMDB - SentiWordNet

Utilizando el conjunto de datos anterior, clasificar las críticas en Positivas y Negativas utilizando un diccionario (lexicón) de sentimientos. En particular [SentiWordNet](#).

Utilizar la biblioteca de Python **nltk** para ello. Dejamos abajo algunas ayudas para su uso en una máquina júpiter.

1. Instalar el paquete si no existe:

```
!pip install nltk #! Permite ejecutar comandos del sistema operativo
```

2. Importar la librería y descargar el diccionario

```
import nltk
nltk.download('sentiwordnet')
nltk.download('wordnet')
from nltk.corpus import sentiwordnet as swn
```

3. Ejemplo, para ver las posibilidades de la palabra “good” (bueno en inglés)

```
good = swn.senti_synsets('good', 'n')
```

```
list(good)
```

Lo anterior va a mostrar una lista grande de **SentiSynset**, algo así como sinónimos. Las letras indican de qué forma es usada la palabra:

N: Sustantivo (noun)

A: Adjetivo

R: Adverbio

V: Verbo

O mostrará una S para indicar un sinónimo. El número indica la acepción, ya que una misma palabra aún utilizada en la misma categoría gramatical (sustantivo, por ejemplo) podría tener diversas acepciones.

4. Si quisiéramos solo ver “good” como adjetivo. Usamos:

```
good = swin.senti_synsets('good', 'a')
```

5. Y para calcular el score de dicha palabra utilizamos un código como el siguiente:

```
good = swin.senti_synsets('good', 'a')
```

6. `posscore=0`

```
negscore=0
```

```
for synst in good:
```

```
    posscore=posscore+synst.pos_score()
```

```
    negscore=negscore+synst.neg_score()
```

```
print(posscore)
```

```
print(negscore)
```

7. Teniendo en cuenta lo anterior clasificar las críticas usando el Lexicón SentiWordNet.

Cómo a priori no sabemos si una palabra es de tipo Verbo, Adjetivo, Sustantivo o

Adverbio, utilizar estas cuatro categorías para cada palabra. Verificar:

a. Indicar la precisión y exhaustividad

b. Indicar la medida F1

c. Comparar con los resultados obtenidos en el punto anterior.

8. Utilizar un parser superficial, para detectar la categoría gramatical de cada palabra. Y mejorar el código anterior usando solo el puntaje de la palabra para la categoría gramatical detectada. (Es decir solo A, R, N o V según se haya detectado).

Código de ayuda:

a. `nltk.download('maxent_ne_chunker')`

```
nltk.download('words')
```

```
nltk.download('punkt')
```

```
nltk.download('averaged_perceptron_tagger')
```

b. `sentence = "We saw the yellow dog"`

```
tokens = nltk.word_tokenize(sentence)
```

```
print(tokens)
```

```
tagged = nltk.pos_tag(tokens)
```

```
entities = nltk.chunk.ne_chunk(tagged)
```

```
print(entities)
```

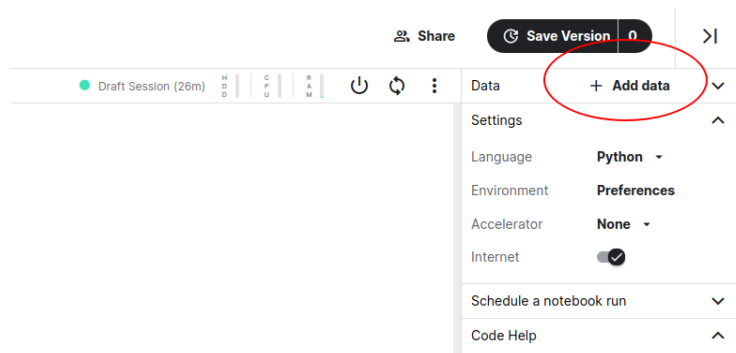
9. ¿Cuanto mejoraron los estimadores respecto al punto anterior?

Emociones - Bayes Naïve in Python

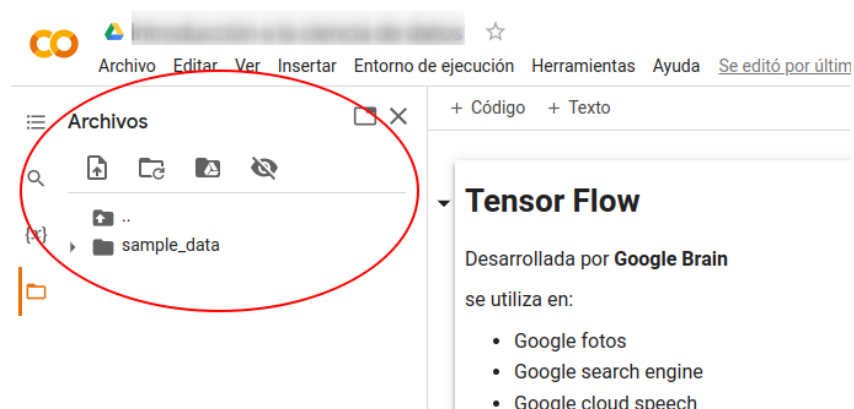
Cargar el siguiente conjuntos de datos en una máquina Jupyter:

https://drive.google.com/file/d/1TMaC1GIK7Cv2aOyIJS_vofXpNtcLQm0U/view

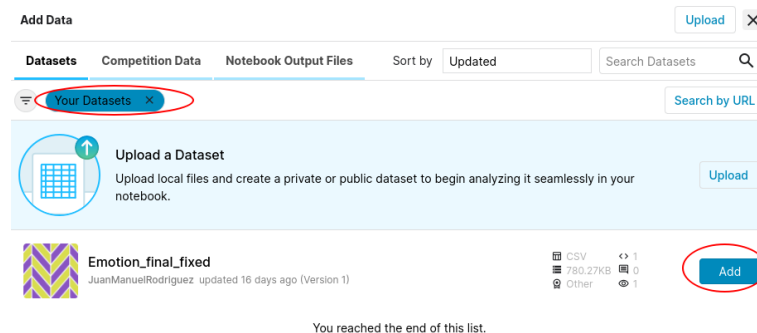
Descompriman el ZIP en su máquina local y luego lo suben a una vm Jupyter de Kaggle:



O de Google Colab:



Si ya lo tenían cargado en Kaggle pueden reutilizarlo, luego de hacer click en “Add Data”:



Y sino siempre pueden ejecutar una máquina Jupyter de forma local.

1. Cargamos los datos en Pandas y exploramos

```
import pandas as pd

df =
pd.read_csv("/kaggle/input/emotion-final-fixed/Emotion_final.csv")

df.head(5)
```

2. Partimos el conjunto en entrenamiento y prueba

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(df.Text,
                                                    df.Emotion,
                                                    test_size=0.30,
                                                    random_state=25,
                                                    shuffle=True)
```


#Exploramos los datos de entrenamiento

```
X_train.head()
```

3. Importamos ahora el modelo de Bayes y le pasamos los datos de entrenamiento

```
from sklearn.naive_bayes import MultinomialNB

bayes = MultinomialNB()
bayes.fit(X_train, y_train)
```

 You have categorical data, but your model needs something numerical. See our [one hot encoding tutorial](#) for a solution.

```
ValueError                                Traceback (most recent call last)
/tmp/ipykernel_33/4000940138.py in <module>
      1 bayes = MultinomialNB()
----> 2 bayes.fit(X_train, y_train)

/opt/conda/lib/python3.7/site-packages/sklearn/naive_bayes.py in fit(self, X, y, sample_weight)
    661         """ Returns the instance itself.
    662
--> 663         X, y = self._check_X_y(X, y)
    664         _, n_features = X.shape
    665

/opt/conda/lib/python3.7/site-packages/sklearn/naive_bayes.py in _check_X_y(self, X, y, reset)
    521     def _check_X_y(self, X, y, reset=True):
    522         """Validate X and y in fit methods."""
--> 523         return self._validate_data(X, y, accept_sparse="csr", reset=reset)
    524
    525     def _update_class_log_prior(self, class_prior=None):
```

Oh no!!! ¿Qué pasó? Todo voló por los aires



4. Recordar lo que vimos en clase. Bayes no sabe qué hacer con las palabras, no es un modelo que trabaje con manipulación de signos, necesita valores numéricos. Por ello debemos convertir las palabras, pero sobre todo los documentos de entrada en vectores, eso se hace utilizando un modelo de bolsa de palabras (*bag of words*). Veamos cómo funciona en scikit-learn:

```
#supongamos que este es mi corpus de documentos
corpus = [
    'This is the first document.',
    'This document is the second document.',
    'And this is the amazing third one.',
    'Is this the first document?',
]

#Construyo entonces un vocabulario, es decir un conjunto de
palabras en donde cada
#palabra aparece una sola vez
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(corpus)
print(vectorizer.get_feature_names_out())
```

Output:

```
['amazing' 'and' 'document' 'first' 'is' 'one' 'second'
'the' 'third' 'this']
```

Se trata de un array de palabras, creado con las palabras del corpus.

```
#Imprimimos nuestra variable X, para ver como quedan los
documentos ahora
print(X.toarray())
```

Output:

```
[[0 0 1 1 1 0 0 1 0 1]
 [0 0 2 0 1 0 1 1 0 1]
 [1 1 0 0 1 1 0 1 1 1]
 [0 0 1 1 1 0 0 1 0 1]]
```

Cada documento quedó representado como un vector, de igual tamaño que el **vocabulario**, en cada posición se pone la cantidad de veces que dicha palabra aparece en el vector.

Este enfoque está bien pero puede ser mejorado, podemos computar la frecuencia inversa de cada término, en donde un término (palabra) tiene un valor bajo si aparece

muchas veces en todos los documentos, pero este valor aumenta si aparece varias veces en el mismo documento. Este valor normalizado se puede calcular con:

TfidfTransformer

```
from sklearn.feature_extraction.text import TfidfTransformer
transformer = TfidfTransformer()
transformer.fit_transform(X.toarray()).toarray()
```

Podemos calcular estos valores usando smooth, para sumar uno en el denominador

```
transformer = TfidfTransformer(smooth_idf=True)
transformer.fit_transform(X.toarray()).toarray()
```

Finalmente podemos combinar estas dos operaciones: el conteo de términos y el cálculo de la frecuencia invertida con este método: **TfidfVectorizer**

```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer()
vectorizer.fit_transform(corpus)
```

5. Luego de haber repasado los métodos de Scikit-Learn para convertir en vectores las palabras estamos listos para seguir:

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import make_pipeline
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.feature_extraction.text import TfidfVectorizer

#Usando el make_pipeline, creamos un modelo que
# automáticamente aplica TfidfVectorizer y luego lo pasa a Bayes.
# Entonces este modelo "es como si" trabajase directamente con el
# texto en lenguaje natural.

model = make_pipeline(TfidfVectorizer(), MultinomialNB())

#Entrenamos el modelo
model.fit(X_train, y_train)

#hacemos predicciones en el conjunto de prueba
predicted_categories = model.predict(X_test)
```

6. Calculamos la precisión

```
print("La precision es {}".format(accuracy_score(y_test,  
predicted_categories)))
```

Árboles

Iris - Scikit-learn

1. Entrenar un árbol de decisión para el conjunto de datos de Iris.
2. Partir el conjunto Iris en dos: “entrenamiento” y “prueba”, dejando un 25% del conjunto original para pruebas.
3. Imprimir la precisión (*accuracy_score*), para ambos conjuntos.
4. Graficar el árbol de decisión.
5. Volver a calcular el árbol cambiando el hiperparámetro: *criterion* para que use la entropía en lugar de la impureza de Gini.
 - a. ¿Cuál da mejor resultado?
 - b. ¿Cambió el tiempo de entrenamiento?
6. Entrenar un modelo **RandomForest**
 - a. Calcular la precision
 - b. Calcular la matriz de confusión
 - c. Comparar con los árboles anteriores
 - d. Cambiar el hiperparámetro *n_estimators* dejarlo igual 10.
 - i. ¿Mejoró o empeoró el modelo?
 - ii. ¿Cambió el tiempo de entrenamiento?

Algunas ayudas:

- El dataset Iris viene incorporado en la biblioteca `sklearn`

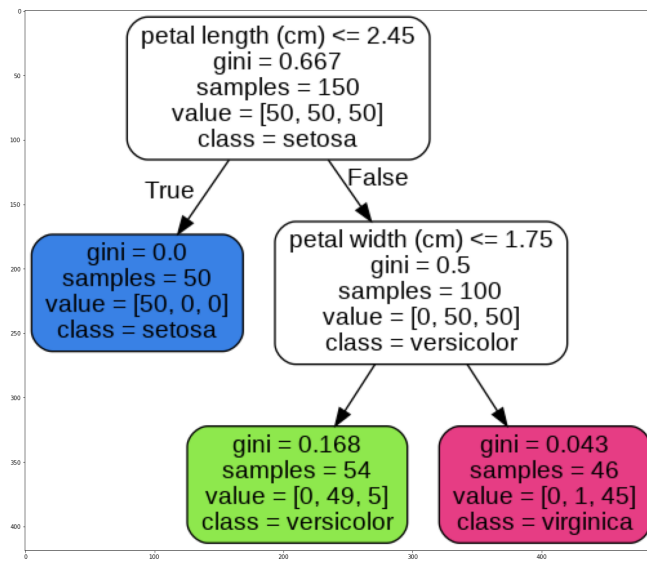
```
from sklearn import datasets
iris = datasets.load_iris()
list(iris.keys())
```
- La función “`train_test_split`” permite partir el conjunto en 2 (investigar)
- El árbol que incluye `sklearn` se llama: “`DecisionTreeClassifier`”
- Para imprimir el Score, de la medida accuracy utilizar:

```
from sklearn.metrics import accuracy_score
```
- Para graficar:

```
from sklearn.tree import export_graphviz
export_graphviz(model, 'tree.dot', feature_names = FEATURE_NAMES)
! dot -Tpng iris_tree.dot -o tree.png

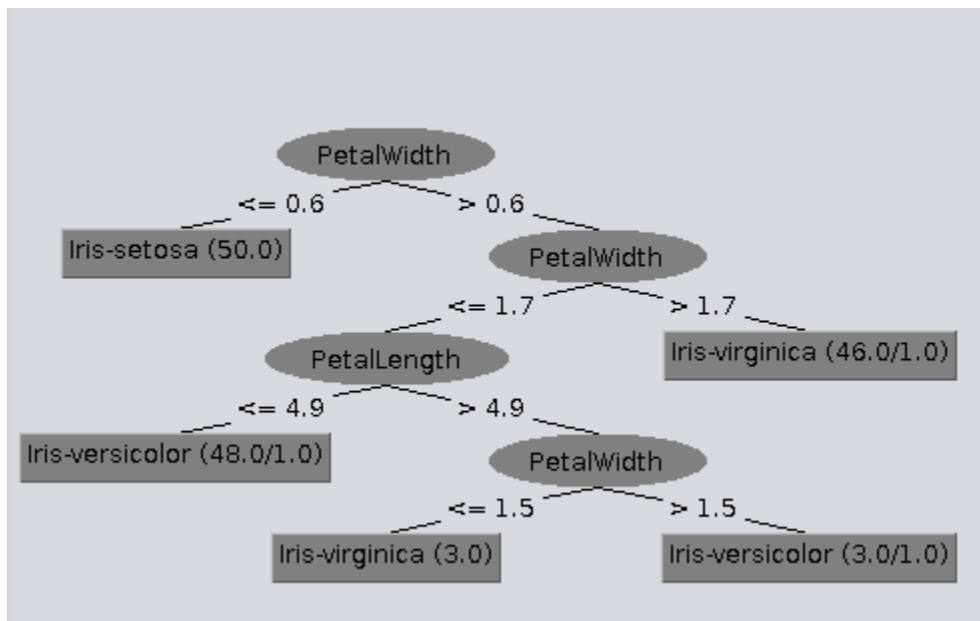
import matplotlib.pyplot as plt
import cv2
%matplotlib inline
img = cv2.imread('tree.png')
plt.figure(figsize = (20, 20))
plt.imshow(img)
```


- Resultado esperado:



Iris - Weka

1. Importar el dataset de Iris en [Weka](#), y entrenar un árbol J48. Luego visualizarlo. Deberían ver algo similar a esto:



2. Verificar la precisión obtenida, el recall y la medida F1. Comparar con el caso anterior.

3. Entrenar un clasificador **RandomForest**. Obtener la precisión, el recall y la medida F1.
¿Mejora al anterior?

Ensamblados

Boston Housing dataset

Entrenar estos 3 modelos de regresión, basados en ensambles, para el dataset propuesto:

- [sklearn.ensemble.AdaBoostRegressor](#)
- [sklearn.ensemble.GradientBoostingRegressor](#)
- [XGBRegressor](#)

NOTA: XGBoost no está implementado en sklearn, tiene su propia implementación, como biblioteca independiente, en Python.

1. Cargar el conjunto de datos de entrenamiento y explorar los datos:

```
from sklearn.datasets import load_boston
boston = load_boston()

print(boston.keys())

print(boston.feature_names)

print(boston.DESCR)
```

2. Partir los datos en entrenamiento y prueba.

```
#Pasamos los datos a Pandas
import pandas as pd
data = pd.DataFrame(boston.data)
data.columns = boston.feature_names

#agregamos en data la columna target
data['PRICE'] = boston.target

#Definimos nuestro X e y
X, y = data.iloc[:, :-1], data.iloc[:, -1]

#partimos los datos en entrenamiento y prueba
from sklearn.model_selection import train_test_split

#30% para test, y el random_state nos asegura que la partición
#sea siempre igual
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=777)
```

3. Entrenamos los modelos, abajo mostramos el código solo para XGBoost

```
import xgboost as xgb

xg_reg = xgb.XGBRegressor()

xg_reg.fit(X_train,y_train)
```

NOTA: Si falla al importar xgboost hay que instalarlo como librería:
`!pip3 install xgboost`

4. Calculamos el error cuadrático medio

```
from sklearn.metrics import mean_squared_error
preds = xg_reg.predict(X_test)
mse = mean_squared_error(y_test, preds)

print("El error cuadrático medio (MSE) en el conjunto de
prueba: {:.4f}".format(mse))
```

5. Entrenar con el mismo conjunto de datos los otros modelos pedidos y calcular su error cuadrático medio.

[sklearn.ensemble.AdaBoostRegressor](#)
[sklearn.ensemble.GradientBoostingRegressor](#)

6. Probar XGBoost con otros *hyperparameters*, distintos de los que usa por defecto:

```
xg_reg = xgb.XGBRegressor(
    objective='reg:linear',
    colsample_bytree = 0.3,
    learning_rate = 0.1,
    max_depth = 5,
    alpha = 10,
    n_estimators = 10)

xg_reg.fit(X_train,y_train)
```

7. Entrenar XGBoost con cross-validation (K-fold)

```
#convertir los datos de entrada a un formato optimizado para
XGBoost (una Dmatrix)
data_dmatrix = xgb.DMatrix(data=X,label=y)
```

```
#nfold: entrenamos con 3 particiones
#num_boost_round: indica el número de árboles que construirá
#               (análogo a n_estimators)
#as_pandas: devuelve los resultados en un DataFrame de pandas.
#early_stopping_rounds: finaliza el entrenamiento del modelo
# antes de tiempo si la métrica usada ("rmse" en este caso)
# no mejora para un número determinado de ciclos.
```

```
cv_results = xgb.cv(dtrain=data_dmatrix, params={}, nfold=3,
                    num_boost_round=50, early_stopping_rounds=10, metrics="rmse",
                    as_pandas=True, seed=777)
```

Nota: el *hyperparámetro* **params**, es para setear los hyperparámetros del modelo, si lo dejamos vacío toma los valores por defecto, pero podríamos definir los nuestros:

```
params = {"objective": "reg:squarederror", 'colsample_bytree':
0.3, 'learning_rate': 0.1, 'max_depth': 5, 'alpha': 10}
```

```
cv_results = xgb.cv(dtrain=data_dmatrix, params=params, ...)
```

8. Visualizar los datos obtenidos con k-folds

```
#cv_results contiene métricas RMSE de entrenamiento y prueba
para cada ciclo
cv_results.head()
```

```
print((cv_results["test-rmse-mean"]).tail(1))
```

9. Comparar los resultados obtenidos por XGBoost contra los otros dos métodos propuestos.

- ¿Cuál ha resultado tener en un menor error cuadrático medio?
- ¿Qué significa el error cuadrático medio en este conjunto de datos? ¿Cuán precisa es la estimación del precio de una vivienda? Es decir, si calculo el precio de una casa con uno de los modelos, sabiendo que el RMSE es de **X**, ¿por cuánto le estoy errando (en promedio) al que puede resultar finalmente el valor de venta?

Sobrevivientes del Titanic

1. Utilizando nuevamente el dataset de sobrevivientes del Titanic, entrenar estos 3 modelos de ensamble, para problemas de regresión:
 - a. [sklearn.ensemble.AdaBoostClassifier](#)
 - b. [sklearn.ensemble.GradientBoostingClassifier](#)
 - c. [XGBClassifier](#)
2. Verificar las métricas:
 - a. precisión,
 - b. *recall*
 - c. F1.
3. Comparar los resultados con los obtenidos entre estos 3 modelos y para el caso de regresión logística.

Iris con Ensamblados híbridos

Para entrar en calor, realizar el ejemplo de ensambles híbridos propuesto en la documentación de **Scikit-Learn** con el conjunto de **Iris** visto en el punto de árboles.

<https://scikit-learn.org/stable/modules/ensemble.html#voting-classifier>

Nosotros lo replicamos acá:

```
from sklearn import datasets
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier

#Cargamos el dataset, y lo partimos en X e Y
iris = datasets.load_iris()
X, y = iris.data[:, 1:3], iris.target

#Definimos 3 métodos de clasificación distintos
clf1 = LogisticRegression(random_state=1)
clf2 = RandomForestClassifier(n_estimators=50, random_state=1)
clf3 = GaussianNB()

#definimos un modelo de ensamble híbrido, con los 3 modelos, este modelo
#es de tipo Voting (o sea define por voto)
#Cada modelo es agregado con un label para identificarlo: lr, rf o gnb
```

```

#Además la votación puede ser dura: hard o blanda: soft, si es dura cada
#voto vale 1, si es blanda el voto es ponderado por la probabilidad de
#certeza con la que el modelo generó el resultado. Hay que tener en cuenta
#que no todos los modelos devuelven una probabilidad asociada al
resultado.
eclf = VotingClassifier(
    estimators=[('lr', clf1), ('rf', clf2), ('gnb', clf3)],
    voting='hard')

#Para cada uno de los modelos, por separado primero y luego para el modelo
de ensamble, entrenamos con cross validation. Con 5 particiones y le
decimos que use la métrica 'accuracy' como forma de medir el rendimiento
en cada partición.

for clf, label in zip([clf1, clf2, clf3, eclf], ['Logistic Regression',
'Random Forest', 'naive Bayes', 'Ensemble']):
    scores = cross_val_score(clf, X, y, scoring='accuracy', cv=5)

    #imprimimos el score calculado y su desviación estándar, con el label
    #correspondiente a cada modelo
    print("Accuracy: %0.2f (+/- %0.2f) [%s]" % (scores.mean(),
scores.std(), label))

```

Análisis de sentimientos con Ensambls híbridos

Para este ejercicio hay que haber completado el siguiente ejercicio de Análisis de sentimientos:

[Emociones - Bayes Naïve in Python](#)

1. Crear un clasificador híbrido, como en el caso anterior pero para mejorar la detección de emociones en el texto. A partir del ejercicio de Emociones con Bayes Naive en Python, agregar dos clasificadores más: Random Forest y SVM y calcular la nueva precisión. ¿Mejoró?
2. Reproducir el punto anterior pero con un método de ensamble de tipo: StackingClassifier

Redes neuronales

Iris y Scikit-learn: Perceptron

Primero hacemos lo mismo que hacemos siempre, cargamos el *dataset* de Iris

```
from sklearn.datasets import load_iris
iris = load_iris()
```

Luego lo que siempre hacemos como paso 2, partir el conjunto de datos en entrenamiento y prueba:

```
from sklearn.model_selection import train_test_split
X = iris.data
y = iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=1815)
```

Ahora importamos el Perceptrón (simple) de Sckit-Learn y lo entrenamos... a ver que pasa

```
from sklearn.linear_model import Perceptron
per_clf = Perceptron(max_iter=1000, tol=1e-3, random_state=42)
per_clf.fit(X_train, y_train)
```

Por último calculamos la precisión:

```
from sklearn.metrics import accuracy_score
y_pred = per_clf.predict(X_test)
print("La precision es {}".format(accuracy_score(y_test, y_pred)))
```

1. ¿Cuál fue el resultado obtenido? ¿A qué se debe el resultado anterior?
2. Posiblemente el problema es que tenemos un problema que no es linealmente separable, tengo 3 posibles conjuntos de salida y no se puede partir el espacio en 3 con un hiperplano.

En lugar de querer saber si un ejemplo es: 'setosa' 'versicolor' 'virginica', veamos si es **setosa** o **no-setosa**. Reemplacemos el "y" que era "iris.target" por:

```
y = (iris.target == 0).astype(int)
```

Calcular la nueva precisión. ¿Mejóro?

3. Estaría bueno poder graficar ese hiperplano que parte el espacio en 2, pero lamentablemente tenemos 4 coordenadas ('sepal length (cm)', 'sepal width

(cm)', 'petal length (cm)', 'petal width (cm)') y no podemos visualizar un espacio de 4D.

Usemos solo 2 coordenadas: 'petal length (cm)' y 'petal width (cm)' y veamos qué es lo que hace el Perceptrón. Reemplazamos X por lo siguiente:

```
X = iris.data[:, (2, 3)]
```

- ¿Cuál es la nueva precisión?
- Ejecutar el siguiente código para obtener una representación del espacio clasificado:

```
import matplotlib.pyplot as plt
a = -per_clf.coef_[0][0] / per_clf.coef_[0][1]
b = -per_clf.intercept_ / per_clf.coef_[0][1]

axes = [0, 5, 0, 2]

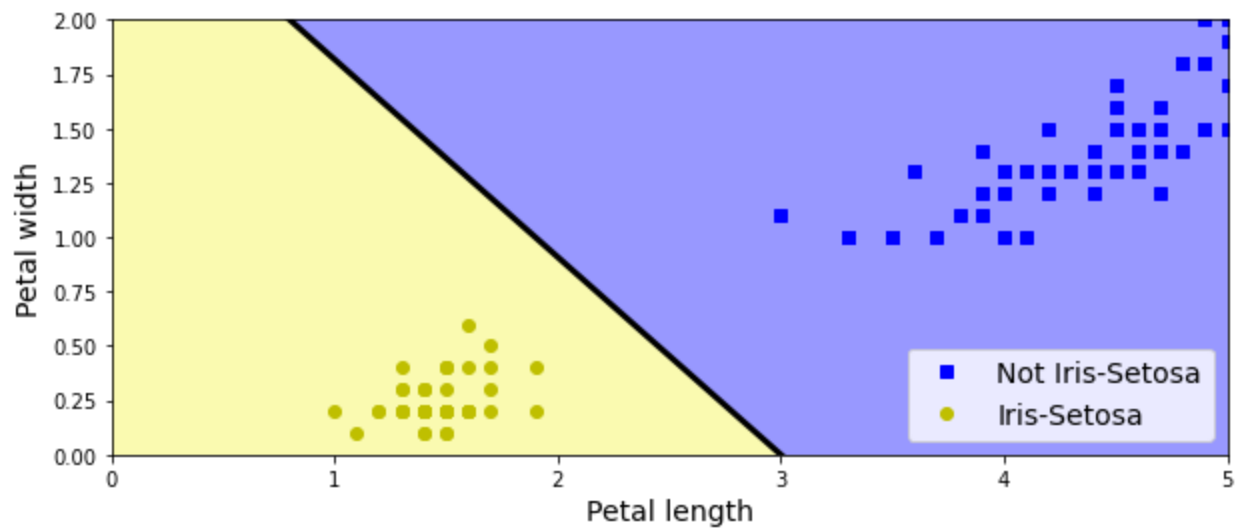
x0, x1 = np.meshgrid(
    np.linspace(axes[0], axes[1], 500).reshape(-1, 1),
    np.linspace(axes[2], axes[3], 200).reshape(-1, 1),
)
X_new = np.c_[x0.ravel(), x1.ravel()]
y_predict = per_clf.predict(X_new)
zz = y_predict.reshape(x0.shape)

plt.figure(figsize=(10, 4))
plt.plot(X[y==0, 0], X[y==0, 1], "bs", label="Not Iris-Setosa")
plt.plot(X[y==1, 0], X[y==1, 1], "yo", label="Iris-Setosa")

plt.plot([axes[0], axes[1]], [a * axes[0] + b, a * axes[1] + b],
        "k-", linewidth=3)

from matplotlib.colors import ListedColormap
custom_cmap = ListedColormap(['#9898ff', '#fafab0'])

plt.contourf(x0, x1, zz, cmap=custom_cmap)
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.legend(loc="lower right", fontsize=14)
plt.axis(axes)
plt.show()
```



Iris y Keras: Perceptrón Multicapa

Intentemos resolver el problema anterior con Perceptrón Multicapa.

Importamos Iris, seteamos el X y el Y, y calculamos cuantas neuronas necesitamos de entrada y cuántas de salida:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

iris = load_iris()
X = iris['data']
y = iris['target']
names = iris['target_names']

n_features = X.shape[1]
n_classes = len(names)

print("Neuronas de entrada:", n_features)
print("Neuronas de salida:", n_classes)
```

Ahora creamos un modelo de red neuronal con dos capas Densas (Fully connected), es decir que cada neurona de una capa se conecta con todas las neuronas de la capa siguiente.

```

from keras.models import Sequential
from keras.layers import Dense

# Creamos un modelo
model = Sequential(name='modelo_1') #un modelo secuencial es donde
"apilamos" capas
model.add(Dense(8, input_dim=n_features, activation='relu')) #capa 1,
entradas=4 (n_feature), salidas 8
model.add(Dense(n_classes, activation='softmax'))

# Compile model
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

model.summary()

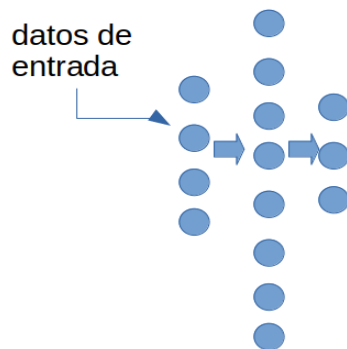
```

El resultado debería ser este:

Model: "modelo_1"

Layer (type)	Output Shape	Param #
dense_14 (Dense)	(None, 8)	40
dense_15 (Dense)	(None, 3)	27
Total params: 67		
Trainable params: 67		
Non-trainable params: 0		

NOTA: Cada capa tiene una entrada y una salida, la primera capa tiene **4 neuronas de entrada**, pero tiene **8 de salida**, la segunda capa tiene **8 de entrada y 3 de salida**. Es decir, solo las neuronas de salida se consideran neuronas de la capa, la entrada es siempre cualquier cosa que venga de la capa anterior, o bien los datos de entrada. Si la representamos sería:



Ahora partimos los datos en entrenamiento y prueba

```
from sklearn.model_selection import train_test_split
```

```
X_train, X_test, Y_train, Y_test = train_test_split(X, y, test_size=0.5,
random_state=2)
```

Y entrenamos el modelo:

```
history_callback = model.fit(X_train, Y_train,
batch_size=5, epochs=50, verbose=0)
```

Salida:

ValueError: Shapes (5, 1) and (5, 3) are incompatible

¿Que paso acá? Por que no anduvo, dice que no le gustan los datos de entrada....

Nosotros le dijimos que la salida era el vector y, el vector y tiene esta forma: [0, 0, 0, 1, 1, 1, 2, 2....]

Es decir, cada elemento del vector representa un valor de salida: 0, 1 o 2 (según la clase de flor), pero la red que armamos tiene 3 neuronas de salida, está esperando algo así: [0,0,1], [0,0,1], [0,0,1], [0,1,0],[0,1,0],[0,1,0],[0,1,0], [1,0,0],[1,0,0],[1,0,0]]

Es decir, que neurona de salida se debe activar, según la clase, así que tenemos que reconvertir la salida:

```

from sklearn.preprocessing import OneHotEncoder
import numpy as np
enc = OneHotEncoder()
Y_transformed = enc.fit_transform(y[:, np.newaxis]).toarray()
print(Y_transformed) # solo para ver qué es lo que hizo

```

Y ahora, sí, volvemos a partir los datos y entrenamos el modelo:

```

X_train, X_test, Y_train, Y_test = train_test_split(X, Y_transformed,
test_size=0.5, random_state=2)

```

```

history_callback = model.fit(X_train, Y_train,
batch_size=5, epochs=50, verbose=0)

```

Imprimimos el resultado de evaluar el conjunto de pruebas:

```

score = model.evaluate(X_test, Y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

```

1. Aumentar el número de epochs a 100, ¿mejoró el resultado?
2. Normalizar los valores de entrada entre 0 y 1 y volver a particionar el conjunto:

```

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
X_train, X_test, Y_train, Y_test = train_test_split(X_scaled,
Y_transformed, test_size=0.5, random_state=2)

```

¿Mejóro o empeoró el resultado?

3. Agregar una capa intermedia al modelo:

```

model.add(Dense(8, activation='relu'))
¿Mejóro en algo?

```

4. Teniendo la capa adicional, y los datos de entrada sin normalizar, entrenar con 100 epochs. ¿Cómo es el resultado ahora?

5. Para el mejor modelo, graficar la curva de accuracy y de loss:

```
import matplotlib.pyplot as plt

plt.style.use('ggplot')
fig, (ax1, ax2) = plt.subplots(2, figsize=(8, 6))

val_accurady = history_callback.history['accuracy']
val_loss = history_callback.history['loss']
ax1.plot(val_accurady, label='modelo_5')
ax2.plot(val_loss, label='modelo_5')

ax1.set_ylabel('validation accuracy')
ax2.set_ylabel('validation loss')
ax2.set_xlabel('epochs')
ax1.legend()
ax2.legend();
```

6. Volver a entrenar el mismo modelo, esta vez con los datos de entrada normalizados y graficar. ¿Qué se observa en los gráficos de ambos modelos?

Keras: Perceptrón Multicapa - clasificador de imágenes

Importamos Keras y Tensor Flow, verificamos las versiones

```
import tensorflow as tf
from tensorflow import keras

print(keras.__version__)
print(tf.__version__)
```

Comencemos cargando el conjunto de datos **MNIST** de moda. **Keras** tiene una serie de funciones para cargar conjuntos de datos populares en **keras.datasets**. El conjunto de datos ya está dividido entre un conjunto de entrenamiento y un conjunto de prueba, pero puede ser útil dividir aún más el conjunto de entrenamiento para tener un conjunto de validación:

```
fashion_mnist = keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()
```

El conjunto de entrenamiento contiene 60.000 imágenes en escala de grises, cada una de 28x28 píxeles:

```
X_train_full.shape
```

Salida:

(60000, 28, 28)

Cada intensidad de píxel se representa como un byte (0 a 255):

```
X_train_full.dtype
```

Salida:

dtype('uint8')

Dividamos el conjunto de entrenamiento completo en un conjunto de validación y un conjunto de entrenamiento (más pequeño). También escalamos las intensidades de los píxeles hasta el rango 0-1 y las convertimos en flotantes, dividiéndolas por 255.

```
X_valid, X_train = X_train_full[:5000] / 255., X_train_full[5000:] / 255.  
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]  
X_test = X_test / 255.
```

Podemos graficar una imagen usando la función `imshow()` de **Matplotlib**, con un mapa de color 'binario':

```
plt.imshow(X_train[0], cmap="binary")  
plt.axis('off')  
plt.show()
```

Las etiquetas son los ID de clase (representados como uint8), del 0 al 9:

```
print(y_train)
```

Aquí seteamos los nombres de clase correspondientes (en inglés):

```
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",  
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

Si las prefieren en castellano, bueno aqui estan:

```
class_names = ["Camiseta/top", "Pantalon", "Pullover", "Vestido",  
               "Abrigo", "Sandalia", "Camisa", "Zapatillas", "Bolso", "Bota corta"]
```

El conjunto de validación contiene 5000 imágenes y el conjunto de prueba contiene 10 000 imágenes:

```
print(X_valid.shape)  
print(X_test.shape)
```

Echemos un vistazo a una muestra de las imágenes en el conjunto de datos:

```
n_rows = 4  
n_cols = 10  
plt.figure(figsize=(n_cols * 1.2, n_rows * 1.2))  
for row in range(n_rows):  
    for col in range(n_cols):  
        index = n_cols * row + col  
        plt.subplot(n_rows, n_cols, index + 1)  
        plt.imshow(X_train[index], cmap="binary", interpolation="nearest")  
        plt.axis('off')  
        plt.title(class_names[y_train[index]], fontsize=12)  
plt.subplots_adjust(wspace=0.2, hspace=0.5)
```



```
plt.show()
```

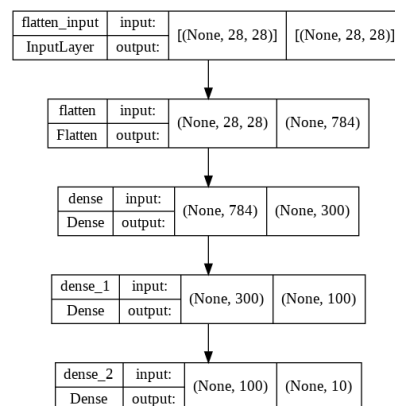


Creemos el modelo:

```
model = keras.models.Sequential()
model.add(keras.layers.Flatten(input_shape=[28, 28]))
model.add(keras.layers.Dense(300, activation="relu"))
model.add(keras.layers.Dense(100, activation="relu"))
model.add(keras.layers.Dense(10, activation="softmax"))
print(model.summary())
```

Imprimamos una representación del mismo:

```
keras.utils.plot_model(model, "my_fashion_mnist_model.png",
show_shapes=True)
```



Solo para explorar, veamos una de las capas y sus pesos:

```
hidden1 = model.layers[1]
print(hidden1.name)
```

¿Es una capa oculta realmente?

```
model.get_layer(hidden1.name) is hidden1
```

¿Cuáles son los pesos y umbrales de esta capa?

```
weights, biases = hidden1.get_weights()
```

¿Cuál es su forma?

```
weights.shape
```

¿Cómo son los umbrales (bias)?

```
print(biases)
print(biases.shape)
```

Compilamos el modelo:

```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd", metrics=["accuracy"])
```

Es equivalente a:

```
model.compile(loss=keras.losses.sparse_categorical_crossentropy,
              optimizer=keras.optimizers.SGD(),
              metrics=[keras.metrics.sparse_categorical_accuracy])
```

Entrenamos:

```
history = model.fit(X_train, y_train, epochs=30, validation_data=(X_valid,
y_valid))
```

Exploremos un poco el **history** devuelto:

```
print(history.params)
print(history.history.keys())
```

Graficamos la precisión y la pérdida o error lo obtenido:

```
import pandas as pd
```

```
pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
plt.show()
```

Evaluemos el modelo:

```
model.evaluate(X_test, y_test)
```

Hagamos una pequeña prueba, tomemos los primeros 3 valores, del conjunto de prueba y veamos que dice que son:

```
X_new = X_test[:3]
y_proba = model.predict(X_new)
y_proba.round(2)
```

Ahora en **y_proba**, están los valores de salida (los redondeamos para quea fácil visualizarlos)

Cómo un array con la forma: [0,0,0,0,0,0,0,0,0.96] es medio difícil de visualizar vamos a convertirlo en directamente el ID, correspondiente, según la neurona que se activó.

```
#el argmax, nos garantiza que se tome la neurona con el mayor valor
y_pred = np.argmax(model.predict(X_new), axis=-1)
Y_pred
```

Y según las clases que habíamos definido al inicio en **class_names**, se trata de....:

```
np.array(class_names)[y_pred]
```

Si revisamos **y_test**, veremos que coincide con lo predicho:

```
y_new = y_test[:3]
Y_new
```

Pero, ¿podemos estar seguros de que esas eran las tres primeras imágenes? Grafiquemoslas, veamos que hay en **X_new**:

```
plt.figure(figsize=(7.2, 2.4))
for index, image in enumerate(X_new):
    plt.subplot(1, 3, index + 1)
    plt.imshow(image, cmap="binary", interpolation="nearest")
    plt.axis('off')
    plt.title(class_names[y_test[index]], fontsize=12)
plt.subplots_adjust(wspace=0.2, hspace=0.5)
plt.show()
```

Keras: Perceptrón Multicapa - Regresor para la estimación de precios de viviendas

Utilizaremos para este ejercicio, un conjunto de datos que viene con Keras:

`california_housing`

```
from sklearn.datasets import fetch_california_housing
housing = fetch_california_housing()
```

En este ejercicio esperamos que seas capaz de entrenar la red por tu cuenta, no te guiaremos paso a paso, solo te dejamos algunas recomendaciones o simplemente cosas a tener en cuenta:

1. Los datos de entrada deben estar normalizados para que no oscile la función de *loss* y la red pueda converger a un valor.
2. Tendrás una sola neurona de salida ya que el valor buscado es un número de punto flotante en un rango continuo.
3. La función de *loss*, ya no es la precisión (*accuracy*), porque estamos en regresión, debemos usar el error cuadrático medio: `mean_squared_error`
4. Comienza tu modelo con 2 capas, entrada y salida, debería ser suficiente y si no logras un resultado adecuado, entonces puedes ir aumentando las capas.
5. Comienza con 20 epochs y un *learning rate* muy pequeño

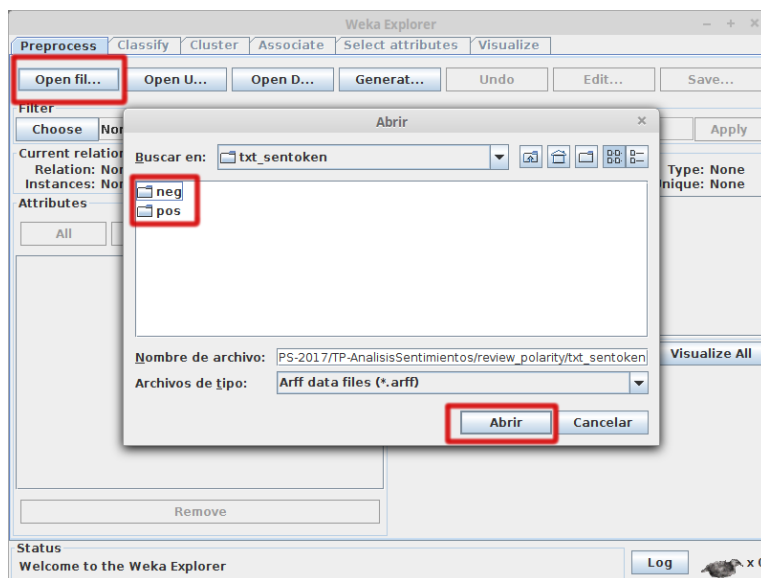
ANEXO A: Cómo entrenar un clasificador de texto utilizando WEKA

Si se tienen diferentes archivos de texto, separarlos en distintos directorios, un directorio por cada categoría. Ejemplo, si tenemos dos categorías: “positivo” y “negativo”, copiar los textos que fueron manualmente identificados como positivos en el primer directorio y los negativos en el otro.

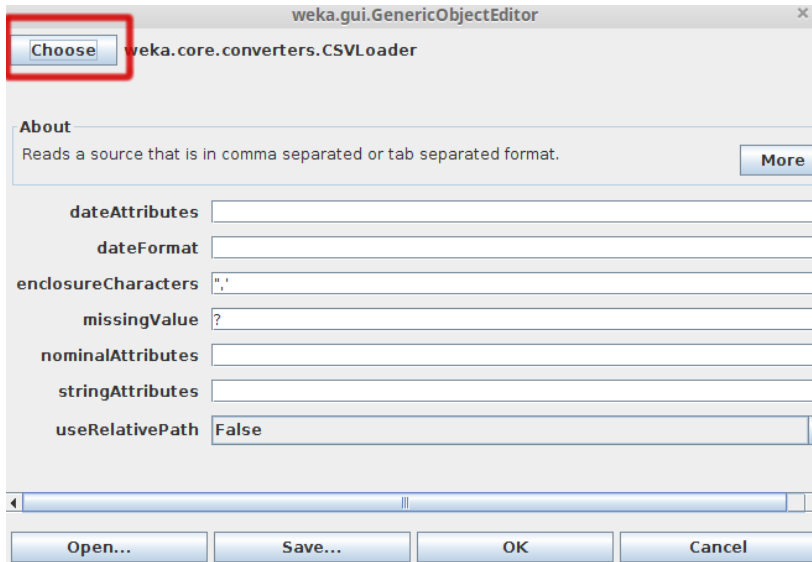
1. Ejecutamos WEKA con el comando: “java -jar weka.jar” y escogemos la opción “Explorer”



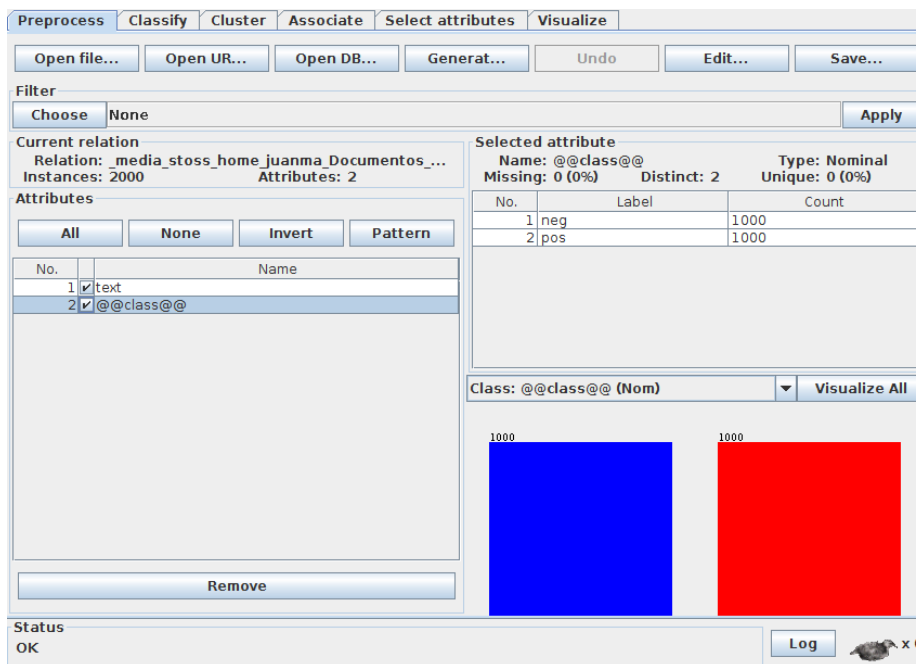
2. En la nueva ventana que aparece hacemos click en “Open File” y navegamos hasta el directorio en el cual tenemos los subdirectorios por categorías. Luego hacemos click en “Abrir”



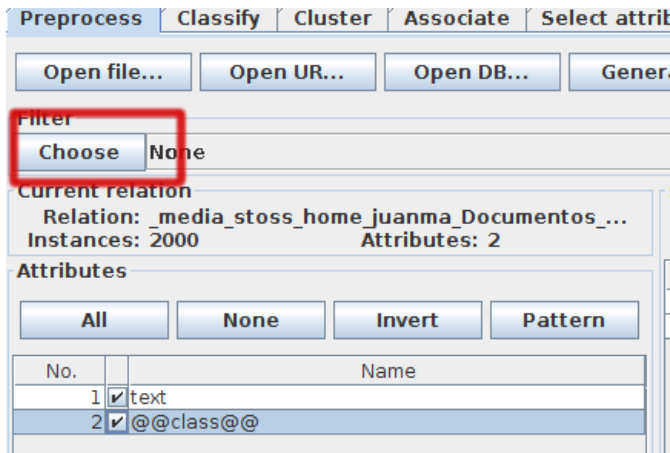
3. WEKA dirá que no sabe cómo interpretar ese tipo de archivo, le damos click al botón “Aceptar” y nos mostrará un diálogo para que le indiquemos cómo deben ser leídos esos directorios. Hacemos click en **Choose**:



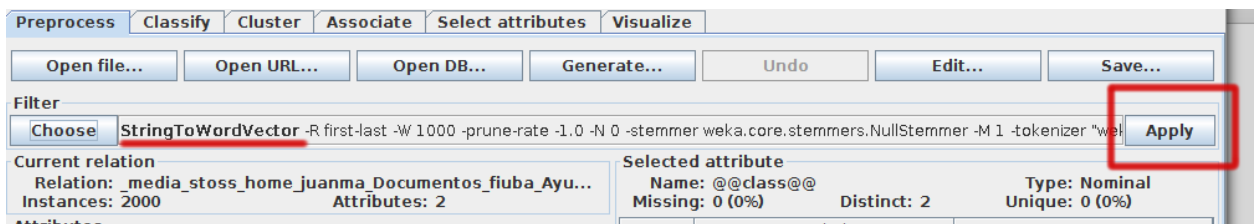
5. Escogemos la opción: “**TextDirectoryLoader**”, aparecerá otro diálogo y le damos click en “Ok”. Debería cargar todo. Deberíamos ver algo como esto:



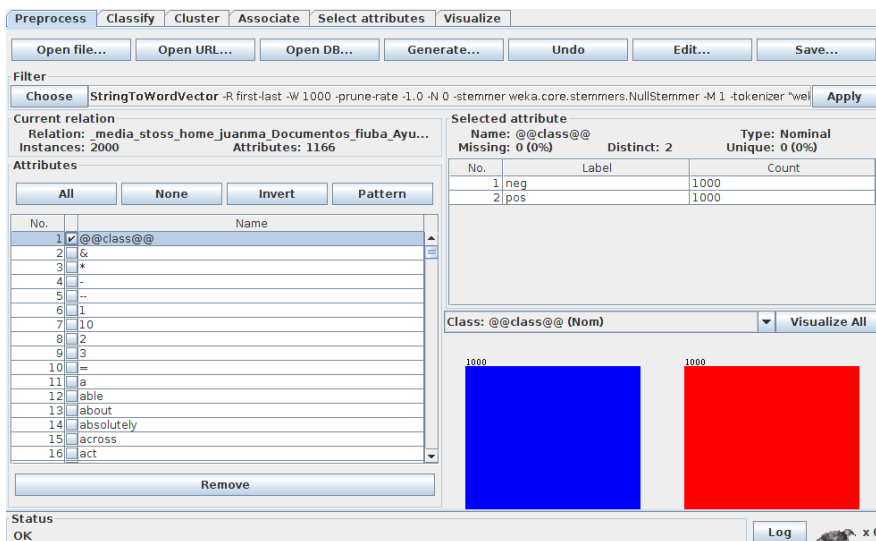
6. Ahora tenemos que **tokenizar** los textos, es decir partarlos en palabras. Para ello seleccionaremos un filtro, haciendo click en el botón **choose**:



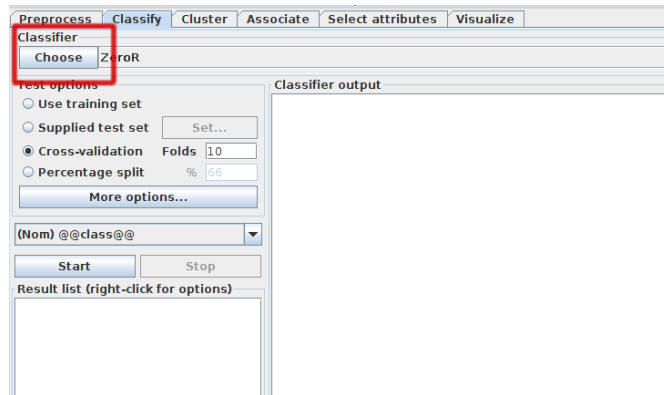
- Escogemos “filters=>unsupervised=>attribute=>StringToWordVector” y le damos click al botón “Apply”



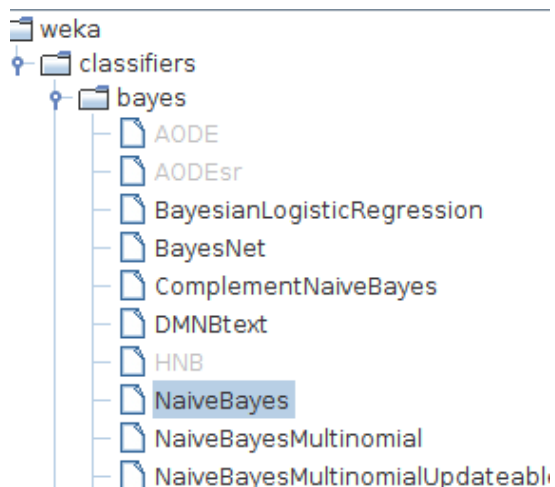
- Deberíamos ver ahora todas las palabras encontradas:



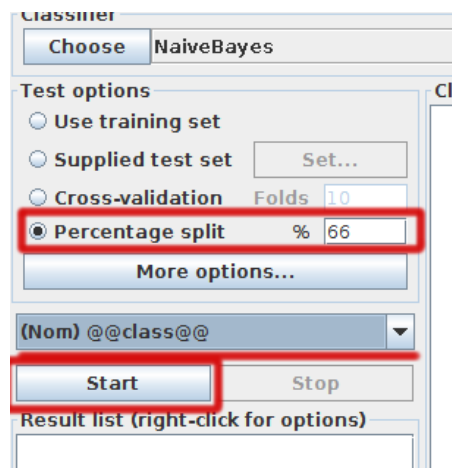
- Ahora vamos a la solapa: “**Classify**” de las solapas que están arriba de todo y hacemos click en “**Choose**” para escoger un clasificador:



10. Escogemos el clasificador, por ejemplo: “Bayes Naive”



11. Seleccionamos luego un porcentaje del 66% para entrenar. Verificamos que el selector de abajo esté en “(Nom) @@class@@” y hacemos click en **Start**



12. Listo, en el cuadro de la derecha verán todos los resultados que arroja WEKA al clasificar!!