Práctica 3.1

Procesamiento del lenguaje natural

Creación de un filtro de SPAM[1]

Enunciado

Actualmente, una de las aplicaciones principales de *Machine Learning* es la detección de spam. Casi todos los servicios de email más importantes proporcionan un detector de spam que clasifica el spam automáticamente y lo envía al buzón de "correo no deseado".

En esta práctica, desarrollaremos un modelo ***Naive Bayes*** que clasifica los mensajes SMS como spam o no spam (referido como 'ham' aquí). Partiremos de un conjunto de mensajes que alimentarán a modo de entrenamiento a nuestro modelo.

Haciendo una investigación previa, encontramos que en los mensajes de *spam* se cumple usualmente lo siguiente:

- Contienen palabras como: 'gratis', 'gana', 'ganador', 'dinero' y 'premio'.
- Tienden a contener palabras escritas con todas las letras mayúsculas y tienden al uso de muchos signos de exclamación.

Esto puede manejarse como un problema de **clasificación binaria supervisada**, ya que los mensajes son o 'Spam' o 'No spam'.

Realizaremos los siguientes pasos:

1. Entender el conjunto de datos
2. Procesar los datos
3. Revisión del "Saco de palabras" ("Bag of Words", o BoW) y su implementación en la librería *Scikit Learn*
4. División del conjunto de datos (dataset) en los grupos de entrenamiento y pruebas
5. Aplicar BoW para procesar nuestro conjunto de datos
6. Implementación de *Naive Bayes con Scikit Learn*
7. Evaluación del modelo
8. Conclusión

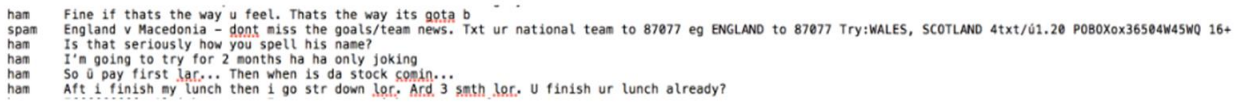
[1] Fuente: https://medium.com/datos-y-ciencia/algoritmos-naive-bayes-fudamentos-e-implementaci%C3%B3n-4bcb24b307f

Pasos a seguir

1) Entender el conjunto de datos

Utilizaremos para entrenar nuestro modelo un conjunto de datos del repositorio UCI Machine Learning.

Un primer vistazo a los datos:



Las columnas no se han nombrado, pero como podemos imaginar al leerlas:

- La primera columna determina la clase del mensaje, o 'spam' o 'ham' (no spam).
- La segunda columna corresponde al contenido del mensaje

Importamos el conjunto de datos y cambiaremos los nombres de las columnas.

```
# Importar la librería Pandas
import pandas as pd
# Dataset de https://archive.ics.uci.edu/ml/datasets/SMS+Spam+Collection

ruta_archivo_sms = # indica aquí la ruta al archivo
df = pd.read_table( ruta_archivo_sms + 'SMSSpamCollection',
          sep='\t', names=['label','sms_message'])
# Visualización de las 5 primeras filas
df.head()
```

Haciendo una exploración previa, vemos que el conjunto de datos está separado en dos partes, tomando como separador '\t'.

	label	sms_message
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...

2) Preprocesamiento de datos

Scikit-Learn solo maneja valores numéricos como entradas, así que convertiremos las etiquetas en variables binarias, donde o representará 'ham' y 1 representará 'spam'.

Para realizar la conversión:

```
# Conversión
df['label'] = df.label.map({'ham':0, 'spam':1})
# Visualizar las dimensiones de los datos
df.shape
```

3) Introducción al Bow y *Scikit Learn*

El conjunto de datos de partida es un texto relativamente grande (5572 filas). Como nuestro modelo solo aceptará datos numéricos como entrada, cabe procesar los mensajes de texto previamente. Aquí es donde el modelo de *Saco de Palabras* ("Bag of Words") entra en juego.

Revisemos brevemente cómo funciona BoW.

BoW toma un fragmento de texto y cuenta la frecuencia de las diferentes palabras que lo componen. BoW trata cada palabra independientemente, donde el orden resulta irrelevante. Podemos convertir un conjunto de documentos en una matriz, siendo cada documento una fila y cada palabra (token) una columna, y los valores correspondientes (fila, columna) son la frecuencia de ocurrencia de cada palabra (token) en el documento.

Como ejemplo, si tenemos los siguientes cuatro documentos:

['Hello, how are you!', 'Win money, win from home.', 'Call me now', 'Hello, Call you tomorrow?']

Convertimos el texto en una matriz de frecuencia de distribución como la siguiente:

	are	call	from	hello	home	how	me	money	now	tomorrow	win	you
0	1	0	0	1	0	1	0	0	0	0	0	1
1	0	0	1	0	1	0	0	1	0	0	2	0
2	0	1	0	0	0	0	1	0	1	0	0	0
3	0	1	0	1	0	0	0	0	0	1	0	1

Los documentos se numeran por filas, y cada palabra es un nombre de columna, siendo el valor correspondiente la frecuencia de la palabra en el documento.

Usaremos el método contador de vectorización **(CountVectorizer)** de *Scikit Learn*, que funciona de la siguiente manera:

- Fragmenta y valora la cadena (separa la cadena en palabras individuales) y asigna un ID entero a cada fragmento (palabra).
- Cuenta la ocurrencia de cada uno de los fragmentos (palabras).
- Convierte automáticamente todas las palabras valoradas en minúsculas para no tratar de forma diferente palabras como "el" y "El".
- También ignora los signos de puntuación para no tratar de forma distinta palabras seguidas de un signo de puntuación de aquellas que no lo poseen (por ejemplo "¡hola!" y "hola").
- El tercer parámetro a tener en cuenta es el parámetro *stop_words*. Este parámetro se refiere a las palabras más comúnmente usadas en el lenguaje. Incluye palabras como "el", "uno", "y", "soy", etc. Estableciendo el valor de este parámetro, por ejemplo, en "english", *CountVectorizer* automáticamente ignorará todas las palabras (de nuestro texto de entrada) que se encuentran en la lista de *stop words* del idioma inglés.

La implementación en *Sci-kit Learn* sería la siguiente:

```
# Definimos los documentos
documents = ['Hello, how are you!',
             'Win money, win from home.',
             'Call me now.',
             'Hello, Call hello you tomorrow?']

# Importamos el contador de vectorización e inicializarlo
from sklearn.feature_extraction.text import CountVectorizer
```

```
count_vector = CountVectorizer()

# Visualizamos del objeto'count_vector' que es una instancia de 'CountVectorizer()'
print(count_vector)
```

Para ajustar el conjunto de datos del documento al objeto *CountVectorizer* creado, usaremos el método *fit()* para obtener la lista de palabras que han sido clasificadas como características usando el método *get_feature_names_out()*. Este método devuelve el diccionario de características para este conjunto de datos, que es el conjunto de palabras que componen nuestro vocabulario en nuestro corpus (i.e. *documents)*.

```
count_vector.fit(documents)
names = count_vector.get_feature_names_out()
names
```

A continuación, queremos crear una matriz cuyas filas corresponderán a los cuatro documentos, y las columnas a cada palabra del diccionario de caraterísticas. El valor correspondiente (fila, columna) será la frecuencia de ocurrencia de esa palabra (en la columna) en un documento particular (en la fila).

Podemos hacer esto usando el método *transform()* y pasando como argumento en el conjunto de datos del documento. El método *transform()* devuelve una matriz de enteros, que se puede convertir en tabla de datos usando *toarray()*.

```
doc_array = count_vector.transform(documents).toarray()
doc_array
```

El paso siguiente es convertir esta tabla en una estructura de datos y nombrar las columnas adecuadamente.

```
frequency_matrix = pd.DataFrame(data=doc_array, columns=names)
frequency_matrix
```

Con esto, hemos implementado con éxito el *Bag of Words* para un conjunto de datos (documentos) que hemos creado. Podemos además aprovechar para mostrar las palabras con más presencia usando una nube de palabras (librería "**WordCloud**").

¿Qué se asigna concretamente a la variable *spam_words*?

```
from wordcloud import WordCloud
import matplotlib.pyplot as plt
# Creamos una lista con las palabras que se consideran 'spam'
spam_words = ' '.join(list(df[df['label'] == 1]['sms_message']))
spam_wc = WordCloud(width = 600, height = 512,
                    backgroundcolor = "grey").generate(spam_words)

# Representamos la nube de palabras
plt.figure(figsize = (12, 8), facecolor = 'k')
plt.imshow(spam_wc)
plt.axis('off')
plt.tight_layout(pad = 0)
plt.show()
```

Un problema potencial que puede surgir al usar este método tiene que ver con el tamaño del conjunto de datos, ya que habrá ciertos valores que son más comunes que otros simplemente debido a la estructura del propio idioma. Así, por ejemplo, palabras como 'es', 'el', 'a', pronombres, construcciones gramaticales, etc. podrían sesgar nuestra matriz y afectar nuestro análisis. Para minimizar este inconveniente, usaremos el parámetro *stop_words* de la clase *CountVectorizer* y estableceremos su valor en inglés.

4) Dividiendo el conjunto de datos en conjuntos de entrenamiento y pruebas

Pasemos ahora al ejemplo que nos ocupa relacionado con la detección de *spam*.

Comenzaremos por dividir nuestros datos para que tengan la siguiente forma:

- **X_train** son nuestros datos de entrenamiento para la columna 'sms_message'
- **y_train** son nuestros datos de entrenamiento para la columna 'label'
- **X_test** son nuestros datos de prueba para la columna 'sms_message'
- **y_test** son nuestros datos de prueba para la columna 'label'.

Mostramos el número de filas que tenemos en nuestros datos de entrenamiento y pruebas

```
# Dividimos los datos en dos partes: entrenamiento y prueba

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(df['sms_message'], df['label'],
random_state=1)

print('Number of rows in the total set: {}'.format(df.shape[0]))
print('Number of rows in the training set: {}'.format(X_train.shape[0]))
print('Number of rows in the test set: {}'.format(X_test.shape[0]))
```

5) Aplicar BoW para procesar los datos de prueba

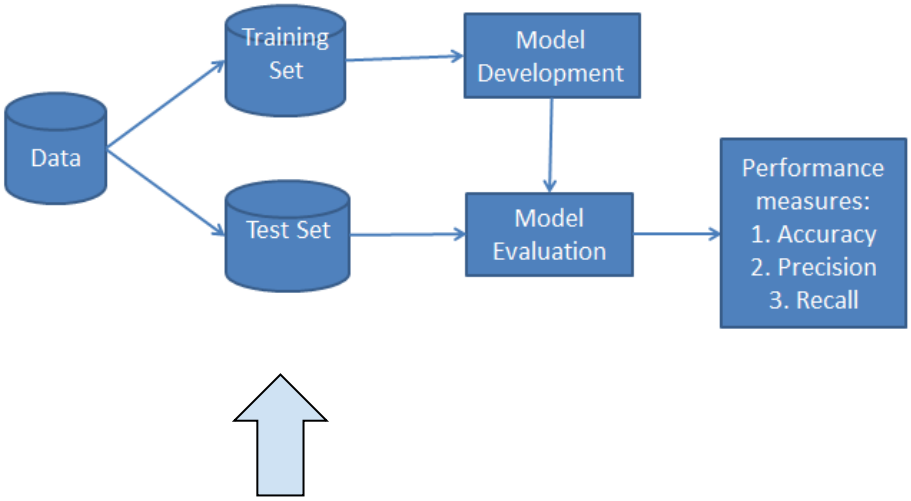
Una vez divididos los datos, el próximo objetivo es convertir nuestros datos al formato de la matriz buscada. Para realizar esto, utilizaremos *CountVectorizer()* como hicimos antes.

Tenemos que considerar dos casos:

1) ajustar nuestros datos de entrenamiento (**X_train**) en *CountVectorizer()*, devolviendo una matriz.
2) transformar nuestros datos de prueba (**X_test**) devolviendo una matriz.

Hay que tener en cuenta que:

- **X_train** son los datos de entrenamiento de nuestro modelo para la columna 'sms_message' en nuestro conjunto de datos.
- **X_test** son nuestros datos de prueba para la columna 'sms_message', y son los datos que utilizaremos (después de transformarlos en una matriz) para realizar predicciones. Compararemos luego esas predicciones con *y_test* en un paso posterior.



El código para este segmento en el que implementamos **BoW** está dividido en 2 partes:

1) generamos un diccionario con el vocabulario para los datos de entrenamiento y luego transformamos los datos en una matriz de documentos
2) para los datos de prueba, solo transformamos los datos en una matriz de documentos, puesto que ya contamos con el diccionario de características (términos).

```
# Creamos un objeto CountVectorizer
count_vector = CountVectorizer()

# Generamos la matriz que a partir de los datos de entrenamiento
training_data = count_vector.fit_transform(X_train)

# Transformamos los datos de prueba y devolvemos la matriz (sin generar el diccionario de características)
testing_data = count_vector.transform(X_test)
```

6) Implementación Naive Bayes con *Scikit-Learn*

Algoritmo *Naive Bayes* Supervisado

A continuación, se listan los pasos que hay que realizar para poder utilizar el algoritmo *Naive Bayes* en problemas de clasificación:

1. Convertir el conjunto de datos en una tabla de frecuencias.
2. Crear una tabla de probabilidad calculando las frecuencias correspondientes a que ocurran los diversos eventos.
3. La ecuación *Naive Bayes* se usa para calcular la probabilidad posterior de cada clase.
4. La clase con la probabilidad posterior más alta es el resultado de la predicción.

Naive Bayes aplicado al análisis de texto

Como decíamos anteriormente usaremos la implementación *Naive Bayes* "multinomial". Esta variante del clasificador es adecuada para la clasificación de características discretas (como en nuestro caso, contador de apariciones de palabras para la clasificación de texto), y toma como entrada el conteo completo de palabras. Por otro lado, el *Naive Bayes gaussiano* es más adecuado para datos continuos ya que asume que los datos de entrada tienen una distribución de curva de Gauss (normal)[2].

Retomamos la codificación en Python de nuestro modelo, importando el clasificador "MultinomialNB" y ajustando los datos de entrenamiento para el clasificador usando **fit().**

```
from sklearn.naive_bayes import MultinomialNB
naive_bayes = MultinomialNB()
naive_bayes.fit(training_data, y_train)
```

Ahora que nuestro algoritmo ha sido entrenado usando el conjunto de datos de entrenamiento, podemos hacer algunas predicciones en los datos de prueba almacenados en 'testing_data' usando **predict().**

```
predictions = naive_bayes.predict(testing_data)
```

Una vez realizadas las predicciones para el conjunto de pruebas, necesitamos comprobar la exactitud de las mismas.

[2] Existe además el clasificador de Bernoulli (Binomial). En este enlace viene una descripción sencilla adicional: https://www.quora.com/What-is-the-difference-between-the-the-Gaussian-Bernoulli-Multinomial-and-the-regular-Naive-Bayes-algorithms

7) Evaluación del modelo

Hay varios mecanismos para hacerlo, pero para entenderlo debemos hacer referencia en primer lugar al concepto de *matriz de confusión*.

La **matriz de confusión** es una forma de representar el conjunto de posibilidades entre la clase correcta de un evento, y su predicción.

Prediction \ Class	True	False
True	True Positive	False Positive
False	False Negative	True Negative

En base a la matriz de confusión, Bayes cuenta con diferentes **métricas**:

- **Exactitud** ("accuracy"): mide con qué frecuencia el clasificador realiza la predicción correcta. Es el ratio de número de predicciones correctas respecto del número total de predicciones (el número de puntos de datos de prueba).

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

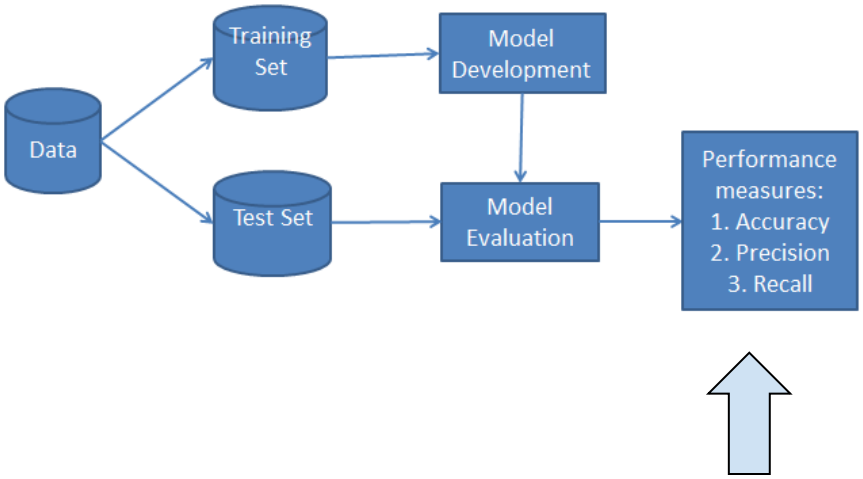
- **Precisión** ("precision"): nos dice la proporción de mensajes que clasificamos como spam. Es el ratio entre positivos "verdaderos" (palabras clasificadas como *spam* que son realmente *spam*) y todos los positivos (palabras clasificadas como *spam*, lo sean realmente o no).

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

- **Sensibilidad** ("recall"): Nos dice la proporción de mensajes que realmente eran spam y que fueron clasificados por nosotros como spam. Es el ratio de positivos "verdaderos" (palabras clasificadas como spam, que son realmente spam) y todas las palabras que fueron realmente spam.

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

Respecto de la **exactitud**, imaginemos, por ejemplo, que tenemos 100 mensajes de texto donde solo 2 fueron spam y los 98 restantes no spam. Podríamos clasificar 90 mensajes como no spam (incluyendo los 2 que eran spam, pero que clasificamos como "no spam", y por tanto, como *verdaderos negativos*) y 10 como spam (siendo 8 de los mismos *falsos positivos*) y todavía conseguir un valor de "exactitud" razonablemente bueno. Sin embargo, en casos como este, la **precisión** y la **sensibilidad** son preferibles. Estas dos métricas pueden ser combinadas para conseguir la puntuación **F1**, que es el "peso" medio de las puntuaciones de precisión y sensibilidad. Esta puntuación puede ir en el rango de 0 a 1, siendo 1 la mejor puntuación posible F1.



Usaremos las cuatro métricas para estar seguros de que nuestro modelo se comporta correctamente. Para todas estas métricas cuyo rango es de 0 a 1, **tener una puntuación lo más cercana posible a 1 es un buen indicador de la bondad del modelo** (es decir, de su fiabilidad en las predicciones que elabora).

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

print('Accuracy score: ', format(accuracy_score(y_test, predictions)))
print('Precision score: ', format(precision_score(y_test, predictions)))
print('Recall score: ', format(recall_score(y_test, predictions)))
print('F1 score: ', format(f1_score(y_test, predictions)))
```

Finalmente, indaga e implementa cómo:

1) cómo mostrar por pantalla la matriz de confusión señalada previamente (preferiblemente como "mapa de calor").
2) cómo mostrar una tabla con las métricas anteriores (existe un método para ello).

Y responde razonadamente a esta pregunta en el cuaderno:

¿Qué podemos decir del modelo?, ¿es fiable cómo se comporta?

Recursos extra

Naive Bayes Classification Tutorial using Scikit-learn

- https://www.datacamp.com/community/tutorials/naive-bayes-scikit-learn

A practical explanation of a Naive Bayes classifier

- https://monkeylearn.com/blog/practical-explanation-naive-bayes-classifier/

Bayes' Theorem, Simply Explained

- https://towardsdatascience.com/bayes-theorem-simply-explained-17217ebc39ff