Pontificia Universidad Católica del Perú

INF656, Minería Web
Trabajo práctico No. n
Segundo semestre 2016, Master en Informática

Clasificación de documentos textuales

Importante: Los scripts y capturas de pantalla mostrando los resultados de los dos problemas propuestos en este TP, deben ser colocados en el repositorio creado en Paideia al terminar la clase o en su defecto, enviarlo al correo `halatrista@pucp.pe`. Si su respuesta involucra 2 o más archivos, comprímalos en un solo archivo ZIP y envíelo con su nombre. El tiempo estimado para solucionar este trabajo práctico es de 2.5 horas.

1. Detección de Spam

La clasificación de documentos, es una tarea fundamental de aprendizaje automático. Se utiliza en todo tipo de aplicaciones, como el filtrado de correos no deseados, la detección del idioma, la clasificación de género, análisis de sentimientos, y mucho más. En este trabajo práctico, utilizaremos el módulo de Python *scikit-learn* para realizar la tarea de clasificación sobre documentos textuales. El objetivo de este trabajo práctico es construir un filtro de spams, poniendo en práctica lo que se aprendió en el curso teórico y programar un filtro simple. Nuestra intención no es crear filtros complejos, como los creados para la producción de servicios como Gmail.

El filtrado de mensajes spams (mails, sms, etc.), es una tarea básica en la clasificación de documentos (es comparado como el "Hello word" en programación). El problema de clasificación de spams es un problema de clasificación binaria: documento deseado (*ham*) o no deseado (*spam*). Sin embargo, debemos tener en cuenta que otros problemas (como la detección de polaridad) no está limitado a dos clases.

1.1. Preparación del corpus

Como en la práctica pasada, utilizaremos la extension llamada *scikit-learn* (`http://scikit-learn.org/stable/index.html`), que es un módulo de Python creado para realizar una gran cantidad de tareas de *machine learning*, que incluye tratamiento de texto y algoritmos de aprendizaje supervisado (clasificación).

El corpus que analizaremos esta compuesto de 5574 *sms* en lengua inglesa. Nuestro corpus se encuentra en la página `https://archive.ics.uci.edu/ml/datasets/SMS+Spam+Collection`. Descargar el archivo *.zip* y descomprimirlo en el disco duro. El archivo descomprimido contiene dos archivos, uno que describe el corpus y otro que contiene los *sms* (recomendamos agregar la extensión *.txt* a los documentos).

Para realizar nuestros experimentos en este trabajo práctico, primeramente, creamos un script que lo llamaremos "spam.py", que luego podrá ser ejecutado con `python spam.py` desde la línea de comandos de Linux.

Como en todas las prácticas, lo primero que necesitaremos es llamar a las extensiones de Python, las cuales ya las hemos utilizado más de una vez, excepto por la extensión *pandas*, la cual, nos brinda algunas herramientas para el manejo de archivos.

```
import os
import pandas
```

Luego, creamos una variable para guardar la ruta donde se encuentra nuestro archivo. Esta línea debe cambiarse por la ruta donde se encuentra guardado el archivo *SMSSpamCollection.txt*. Luego, leemos el archivo completo línea por línea (*line.rstrip for line*)

```
smsFile = "/miRuta/SMSSpamCollection.txt"
sms_content = [line.rstrip() for line in open(smsFile)]
```

Para ver la cantidad de líneas que hemos recuperado en la variable *sms_content*, podemos escribir `print len(sms_content)`. Para ver el contenido de nuestro corpus podemos escribir `print sms`.

En la variable *sms_content*, cada *sms* es se encuentra separado por comas y entre comillas (simples o dobles). Además, vemos que los *sms* ya están anotados (con clases). Las dos clases existentes son *ham* y *spam* y se encuentran al principio de cada línea. Como se puede observar, las clases y los mensajes están separados por $\backslash t$ (tabulador). Entonces, la idea es separarlos en un vector de talla 2 y colocarles un título ("class" y "message"). Esto lo haremos gracias a la extensión *pandas*.

```
sms_content = pandas.read_csv(smsFile, sep='\t', names=["class", "message"])
```

Antes de que podamos entrenar un clasificador, tenemos que cargar datos en un formato que nos permita utilizarlos en un algoritmo de aprendizaje (bag-of-words). Como vimos en el trabajo práctico anterior, *scikit-learn* utiliza estructuras de datos como la matriz *scipy-sparce*, la cual es adecuada para ser utilizada con un clasificador de *sms*. Esto lo hacemos gracias al módulo *CountVectorizer* propuesto por *scikit-learn*, a partir del cual, podremos crear una matriz de términos-documentos. En esta parte del código, debemos tener mucho cuidado, ya que lo que nos interesa es obtener un vector de características a partir de los mensajes (*message*) mas no de las etiquetas (*class*), ambas almacenadas en la variable *sms_content*.

```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
sms_vector = vectorizer.fit_transform(sms_content['message'].values)
```

La instancia *CountVectorizer* creada anteriormente, nos brinda la matriz de términos-documentos (*sms_vector*) y un vocabulario de términos, ambos almacenados en una sola lista. Por otro lado, tenemos nuestras clases que no han sido aún utilizadas, las cuales se pueden ver mediante `print sms_content['class'].values`.

Problema 1

- Si muestra en pantalla la matriz *sms_vector*, puede notar que la representación de cada palabra en el texto se realiza mediante su frecuencia. ¿Qué desventajas tiene este modelo frente a otros, como la representación *tf-idf*? Respalde su respuesta mediante un ejemplo sencillo y pequeño.

1.2. Representación *tf-idf*

Confío en que respondieron correctamente a la pregunta anterior. Entonces, en este trabajo práctico utilizaremos una representación *tf-idf* para cada palabra en el corpus. Para ello, agregamos las líneas de código siguientes a nuestro script.

```
from sklearn.feature_extraction.text import TfidfTransformer
tfidf_transformer = TfidfTransformer().fit(sms_vector)
sms_tfidf = tfidf_transformer.transform(sms_vector)
```

En el código anterior, importamos la herramienta *TfidfTransformer* que nos permite obtener una representación *tf-idf*. Luego, utilizamos la función *TfidfTransformer()* sobre nuestra matriz de términos-documentos *sms_vector*. Finalmente obtenemos una nueva matriz de términos-documentos llamada *sms_tfidf* con una representación de los términos (tokens) en el modelo *tf-idf*. Podemos visualizar la matriz utilizando un `print`.

2. Creación de un modelo

Una vez construida nuestra matriz, solo nos queda seleccionar un algoritmo de clasificación entre los muchos que ofrece *scikit-learn*. Gracias a una rápida lectura en el estado del arte, nos daremos cuenta que el mejor algoritmo de clasificación para datos textuales es el *SVM*. Sin embargo, esta vez, utilizaremos un algoritmo básico, como *Naive Bayes*. Para ello, importamos *MultinomialNB*. Entendemos por multinomial, un algoritmo que permite analizar muchos atributos. Posteriormente almacenamos las clases dentro de una variable (*targets*) para luego crear nuestro modelo gracias a la línea `classifier.fit(sms_vector, targets)`. Este comando utiliza dos parámetros: el vector con la representación *tf-idf* y las clases (una por cada *sms*).

```
from sklearn.naive_bayes import MultinomialNB
classifier = MultinomialNB()
targets = sms_content['class'].values
clf = classifier.fit(sms_tfidf, targets)
```

2.1. Predicciones de *spam*

Una vez nuestro modelo creado, no nos queda más que probar si funciona. Para ello, creamos un pequeño vector con un par de textos "sugestivos", los cuales nos permitan fácilmente probar nuestro modelo. Luego, hacemos lo que siempre hacemos, las vectorizamos y predecimos su clase con *classifier.predict()*. Todo esto se hace en 4 simples líneas en Python y podemos ver las clases de nuestros *sms* gracias al famoso `print`.

```
examples = ['Free Viagra call today!', 'Hello my friend']
example_vector = vectorizer.transform(examples)
predictions = classifier.predict(example_vector)
print predictions
```

2.2. Evaluación del modelo

Cuando creamos un modelo, debemos saber qué tan correcta se realizó la clasificación. Existen varios métodos para realizar una validación, de entre las cuales, la validación mediante la técnica de *cross validation* es una de las mejores. En este trabajo práctico, haremos algo más simple. Vamos a predecir las clases de nuestro corpus (5574 *sms*) con el modelo creado anteriormente y veremos cuántas veces nuestro modelo se equivocó, comparándola con sus "verdaderas" clases. Para ello, utilizamos las líneas siguientes.

```
from sklearn.metrics import classification_report, accuracy_score, confusion_matrix
all_predictions = clf.predict(sms_tfidf)
accuracy = accuracy_score(sms_content['class'], all_predictions)
cm = confusion_matrix(sms_content['class'], all_predictions)
statistics = classification_report(sms_content['class'], all_predictions)
print accuracy
print cm
print statistics
```

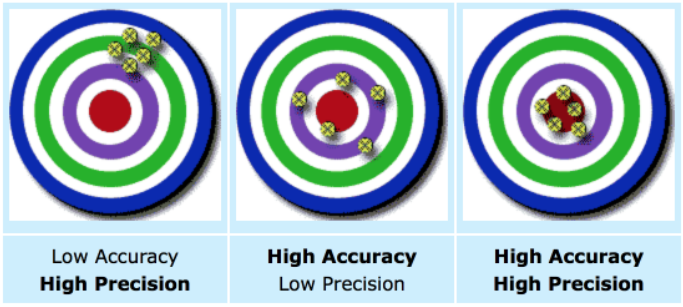
Primeramente, importamos lo que necesitamos, luego, la línea `all_predictions = clf.predict(sms_tfidf)` nos permite predecir las clases de nuestro corpus. Luego, simplemente calculamos la *accuracy*, la *matriz de confusión* y algunas *estadísticas* como la *precisión*, el F1, etc. y las mostramos.

Problema 2

Repita los pasos anteriores, pero esta vez, utilize un corpus más difícil a analizar: correos electrónicos. Para ello, vaya a sitio `http://www.aueb.gr/users/ion/data/enron-spam/` y descargue el archivo *Enron4*,

el cual tiene 2 directorios (ham y spam). La idea es construir una matriz *mail - class* para luego realizar todos los pasos realizados en este trabajo práctico. Puede mejorar este trabajo práctico utilizando el algoritmo *SVM* y una evaluación utilizando *cross-validation*.

NOTA: con respecto a la validación de nuestro modelo, descrito en la Sección 2.2, se muestran dos medidas diferentes: *accuracy* y *presicion*. Ambas son diferentes y esta diferencia puede mostrarse en la figura siguiente.



http://www.mathsisfun.com/accuracy-precision.html

–
Hugo Alatrista-Salas
PUCP