Facultad de Ingeniería - UBA

Organización de Datos 75.06

Trabajo Práctico: Informe Final

Fine Food Reviews

Integrantes:	*E-mails:*
Agustina Barbetta *96528*	agustina.barbetta@gmail.com
Santiago Lazzari *96735*	santilazzari1994@gmail.com
Luciano Mintrone *95463*	lucianomintrone@gmail.com
Manuel Porto *96587*	manu.porto94@gmail.com

Grupo 18

30 de noviembre de 2016

Resumen

En el presente trabajo se implementa un predictor de puntaje para un set de datos con reseñas de productos estudiado, anteriormente, en la instancia de diseño.

Índice

1. Modelo propuesto anteriormente

1.1. Pre-procesamiento

Para la solución a implementar se decidió realizar una eliminación del HTML, caracteres especiales, numéricos y pasaje a lower case sobre los sets *training* y *test*, obviando la eliminación de *stopwords*, ya que desempeñan una función importante a la hora de entender los sentimientos de los usuarios.

Se propone realizar un manejo de las negaciones. Al utilizar cada palabra como un *feature*, la palabra "good" en la frase "not good" contribuirá a un sentimiento positivo en lugar de a uno negativo, debido que la presencia de "not" no se tiene en cuenta. Para resolver este problema se escribirá un *script* que transforme una palabra seguida de un "not" o "n't" en "not_" + palabra. Las palabras a negar se detectan con una variable de estado, cuando esta es verdadera, a todas las palabras leídas se les antepondrá la negación. La variable vuelve a su valor inicial cuando se encuentra un signo de puntuación u otra negación.

1.2. Procesamiento

Previo al entrenamiento del modelo elegido, se deberá preparar el set de acuerdo a los requisitos del algoritmo. Además se determinará qué campos serán incluidos para realizar el entrenamiento, en base a lo explicado en apartados anteriores.

El procesamiento propuesto consiste en construir una matriz compuesta por los Id's de las reseñas, su puntaje y todas las palabras halladas en los campos "Summary" y "Text" de todas las reviews. Cada fila de esta matriz corresponde a un review individual. Cada una de las palabras corresponderá a un atributo y tendrán un valor entre 0 y N dependiendo de la cantidad de veces que dicha palabra aparezca en la review. A continuación se presenta un ejemplo de la matriz:

Id	Prediction	good	not_good	taste
1	4	3	0	1
2	1	0	2	2
3	2	1	3	0
4	2	0	1	0

Cuadro 1

Dado que la gran cantidad de palabras existentes tiene como consecuencia que las matrices tengan una dimensión demasiado grande, se filtrarán las palabras en base a su ocurrencia. Sabiendo que la cardinalidad de las palabras por predicción se distribuye como una ley de potencias, se podrá encontrar algún porcentaje empírico de palabras, las más relevantes, que se utilizarán para entrenar. Y siendo ésta distribución una ley de potencias, dicha relación es invariante a la escala, es decir, que para todas las predicciones (independientemente de la cantidad de reviews que tengan) se podrá usar la mismo proporcionalidad.

Otra forma de reducir éste número es aplicarle the hashing trick a las palabras y así poder reducir las dimensiones de la matriz. Se probará si hace falta aplicarle the hashing trick, y si hiciera falta, cuán costoso és.

1.3. Predicción

El modelo que se utilizará para predecir el puntaje de los reviews será SVM, como ya se explicó, SVM es un algoritmo especialmente útil para clasificación de textos. Este modelo puede

ser utilizado para problemas no lineales, como es el caso del presente trabajo [1] y se cuenta con evidencia empírica de que obtiene mejores resultados que otros algoritmos para problemas de clasificación no lineales. [2]

Según lo investigado, SVM no funciona bien para sets de datos desbalanceados[3], lo cual es nuestro caso (como fue detallado en el análisis del set de datos). Para solucionar este problema se pueden usar dos alternativas; Una solución posible es balancear el set de datos, recortándolo aleatoriamente o aprovechando a sacar las entradas repetidas. Por otro lado, si esto último no funciona, optaremos por cambiar el predictor por otro, como un perceptrón multicapa.

2. Flujo y procesamiento de la información

El dataset es procesado de la siguiente manera:

1. Se preprocesan los reviews del set con el script de PySpark, `parallel_preprocess.ipynb`. En el se realizan los siguiente cambios, que luego son guardados en un nuevo archivo.
   - Eliminación de HTML
   - Eliminación de caracteres especiales
   - Eliminación de caracteres numéricos
   - Pasaje a lower case
   - Eliminación de stopwords
   - Negador
2. Una vez pre-procesado se vectoriza el texto de cada review y se reducen las dimensiones de los vectores resultantes mediante Hashing Trick con el script `word2matrix.py`.
3. Se utiliza la totalidad del *training* set como entrenamiento para el clasificador.
4. Realizado el entrenamiento, se utiliza el *test* set para generar las predicciones.

3. Pre-procesamiento

El script parte de los sets de datos descargados de Kaggle y consiste en pasar los campos Summary y Text de una review por una función que recibe una cadena y ejecuta una serie de funciones sobre ella.

En primer lugar, tres expresiones regulares que filtran el marcado HTML, los caracteres especiales (Excepto los . y ', que se necesitarán para el manejo de negaciones) y los caracteres numéricos. Luego, se pasa toda la cadena a lower case y se la recorre para eliminar las palabras que se encuentran en nuestro `stopwords.csv` en el cual hemos puesto las palabras que no contienen un significado importante para el idioma inglés, excluyendo palabras como "good","like", "not" ya que estas pueden aportar la predicción del sentimiento del usuario. Por último, se realiza la negación de la cadena, que transforma cada palabra seguida de un "not" o "n't" en "not_" + palabra. Las palabras a negar se detectan con una variable de estado; cuando esta es verdadera, a todas las palabras leídas se les antepone la negación. La variable vuelve a su valor inicial cuando se encuentra un signo de puntuación u otra negación. Opcionalmente, se añaden los pasos de *stemming*, que lleva cada palabra de la cadena a su raíz, y *n-gram*, que realiza secuencias de n palabras con la review.

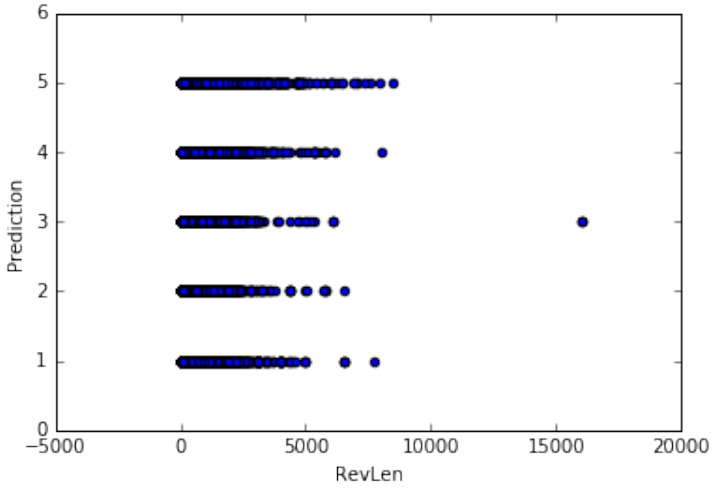

Figura 1: Longitud de reviews por puntuación

4. Procesamiento

Para este paso, se parte de los sets de datos pre-procesados y se vectoriza cada review de ambos sets por medio de **Hashing Trick**.

Luego del estudio realizado para la entrega de diseño (vimos, por ejemplo, que los usuarios suelen escribir reviews de longitudes similares independientemente del puntaje que le asignen a un producto (Figura 1) y que los *HelpfulnessNumerator* y *HelpfulnessDenominator* no suelen marcar diferencias ya que son muy pocos los votos totales por review (Figura 2)), decidimos procesar sólo los campos *Summary* y *Text* de los sets de datos.

Inicialmente se decidió convertir los textos del set a vectores con una cota de error del 1 % que, según el teorema de Johnson-Linderstrauss, son de 127 dimensiones. Más adelante fuimos probando con mayores dimensiones reduciendo, consecuentemente, la cota del error.

La salida de este script es un `.csv` con `Id,Prediction,vector` en el caso del *training* set y `Id,vector` en el caso del *test* set.

5. Predictores utilizados

5.1. Support Vector Machines

Si bien en la entrega de diseño propusimos la utilización de este algoritmo, al momento de implementarlo encontramos que las implementaciones presentes no eran óptimas para las condiciones de nuestro problema. En una de las librerías de análisis de datos de Python advertían que su implementación no escalaba bien para un numero de muestras mayor a 10000 ya que su implementación del algoritmo es de orden cuadrático. [4]

Teniendo en cuenta lo anterior consideramos que utilizar dicha librería no seria adecuado para el presente problema. Si bien podríamos haber implementado nuestra propia versión, consideramos que dado el tiempo disponible y los conocimientos que teníamos sobre el tema, la tarea no era factible por lo que decidimos no utilizar este algoritmo.

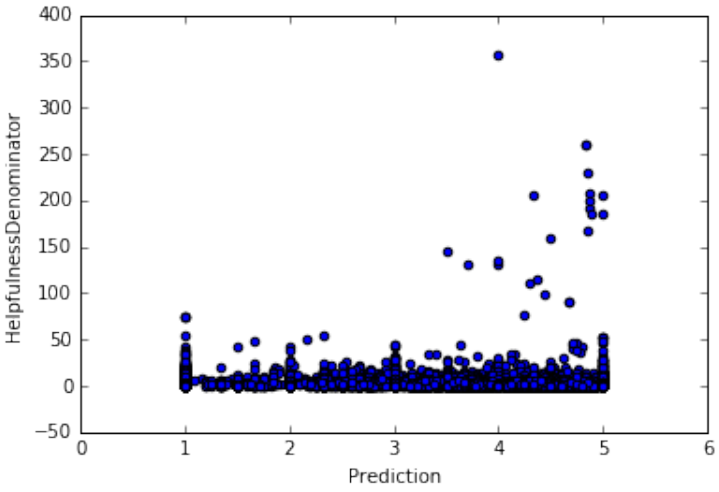

Figura 2: *HelpfulnessDenominator* por puntuación

5.2. Dynamic Vector Prediction

Inspirados en la charla de las JAIIO y de la manera en que Facebook hacia sentiment analisis, propusimos un algoritmo similar que lo llamamos Dynamic vector prediction.

El algoritmo consiste en armar una tabla de la forma palabra:cardinalidad de la palabra por cada predicción. Por ejemplo, en el caso de la predicción 1.

```
{
"tast":22508,
"br":22305,
"like":21614,
"product":21051,
"one":15308,
"food":14724
.
.
.
}
```

Esto se genera recorriendo todos los reviews y de una misma predicción y sumando 1 por cada palabra. Ésta tabla de frecuencias seria el equivalente al aprendizaje.

Para poder predecir un review por ejemplo “The food was awful”

Se hace un split lo cual queda [“The”, “food” , “was”, “awful”]. Teniendo esto que llamaremos, la referencia, se genera un vector, llamado vector dinámico con la cardinalidad de cada palabra. Es decir, si alguna palabra se repitiera, no estaría duplicada en las referencias pero si aumentaría la cardinalidad en el vector dinámico. En este caso el vector dinámico quedaría de la siguiente forma. [1, 1, 1, 1].

Teniendo las tablas de frecuencias de las 5 predicciones, se hace lo mismo con las palabras de referencia. Por ejemplo, si “The” tiene cardinalidad 10000, “food” 1500,“was” 10000 y “awful”500, en la predicción 1 entonces el vector dinámico es [10000, 1500, 1000, 500]. Esto se repite con todas las predicciones.

A esta altura se generaron 6 vectores dinámicos, el de la review + las 5 predicciones. Para

poder comprar todos los reviews de la misma forma, sabiendo que el set de datos está desbalanceado, se procede a normalizar los 6 vectores.

Finalmente se calcula la distancia coseno entre la review contra todas las predicciones y se queda con la menor. Esto define a cuál se parece más y esa es la predicción.

Los resultados de éste algoritmo se probaron con tres distancias distintas y se hizo un post en Kaggle con la mejor de ellas. Basándonos en cuánto se ajustaba al train set.

Con la distancia del coseno los resultados fueron 377 predicciones correctas con respecto a mil. Con la Euclideana fueron 345. Mientras que con Jaccard 77.

Se obtuvo una puntuación de `5.06451` en Kaggle, por lo que concluimos en que el algoritmo no es bueno para este tipo de problema.

5.3. Perceptrón

Como sabemos perceptrón es un clasificador binario lineal por lo tanto tuvimos que adaptarlo a nuestro problema que tiene mas de dos clases. Para ello propusimos perceptrones en cascada, en donde el primero de ellos divide el las reseñas es buenas (3,4 y 5 estrellas) y malas (1 y 2 estrellas). Luego aplicamos un nuevo perceptrón dentro de las reseñas malas, obteniendo una lista de reseñas de 1 estrella y otra lista de 2 estrellas. Lo mismo se realizo para las buenas reseñas. Mediante Cross Validation fuimos probando los resultados y cambiando los hiperparámetros dependiendo de estos. Los hiperparámetros a modificar son la función de activación, el numero de dimensiones de los datos, la cantidad de iteraciones para entrenar el vector de pesos. Usando la función de activación sigmoid, 127 dimensiones y 10 iteraciones obtuvimos un resultado de 50 % de resultados correctos con Cross Validation y en Kaggle de `5.13634`

5.4. Red Neuronal

Inspiradas en las redes neuronales biológicas, una red multicapa utiliza el concepto de los perceptrones donde cada perceptrón es una neurona de una red que conecta varias de estas unidades.

Dado que la red puede ser entrenada con hiperparámetros y datos procesados de distintas formas, se realizaron pruebas con algunas de las combinaciones posibles.

1. Random search de mejores hiperparámetros.
2. Separando el campo *Summary* de *Text*. se subieron entradas aplicando pesos mayores a *Summary* y con pesos idénticos para ambos.
3. Recortando los *reviews* de 5 estrellas del *training* set a 1/3.
4. Reduciendo cantidad de iteraciones para evitar *overfitting*.
5. Aumentando la cantidad de dimensiones del vector de The Hashing Trick.
6. Realizando bigramas antes de vectorizar las reviews.

5.4.1. Random search de hiperparámetros óptimos

Primero se entrenaron y validaron, con Cross Validation, redes con distintos hiperparámetros y el mismo set de datos. Los parámetros fueron los siguientes.

1. activations = ['identity', 'logistic', 'tanh']
2. solvers = ['lbfgs', 'sgd', 'adam']

3. learning_rates = ['constant', 'invscaling', 'adaptive']
4. layers : De 1 a 10 layers cada una con 10 a 100 neuronas
5. alpha : Un numero del 0 al 1
6. momentum : Un numero próximo a 0.9
7. beta_1 : Un número próximo al 0.9
8. beta_2 : Un número muy próximo al 0.9

Luego se tomaron las mejores dos redes y se utilizaron las mismas para realizar dos entregas a Kaggle. Los puntajes fueron `4.90687` para

```
Net ID 25 Correct prediction 33774 total predictions 45468 ratio
       0.742808128794

MLPClassifier(activation='relu', alpha=6.4603656567e-05, batch_size='auto',
       beta_1=0.762066045724, beta_2=0.9977827051, early_stopping=False,
       epsilon=1e-08, hidden_layer_sizes=(82, 63, 62, 82, 84, 14, 17, 94),
       learning_rate='adaptive', learning_rate_init=0.001, max_iter=200,
       momentum=0.802854594112, nesterovs_momentum=True, power_t=0.5,
       random_state=1, shuffle=True, solver='adam', tol=0.0001,
       validation_fraction=0.1, verbose=True, warm_start=False)
```

y `4.77462` para

```
Net ID 17 Correct prediction 32056 total predictions 45559 ratio
      0.703615092517

MLPClassifier(activation='relu', alpha=1.81461856718e-05, batch_size='auto',
      beta_1=0.647482014388, beta_2=0.9977827051, early_stopping=False,
      epsilon=1e-08, hidden_layer_sizes=(35, 73, 90, 24, 47, 66, 18, 38),
      learning_rate='constant', learning_rate_init=0.001, max_iter=200,
      momentum=0.648414985591, nesterovs_momentum=True, power_t=0.5,
      random_state=1, shuffle=True, solver='adam', tol=0.0001,
      validation_fraction=0.1, verbose=True, warm_start=False)
```

Para visualizar los parametros de las redes recien mencionadas y el resto de las utilizadas en el *Random Search* se debe acceder al archivo de analisis de los mismos, presente en el repositorio del trabajo.[5]

5.4.2. Pesos idénticos para Summary y Text

Con 40 iteraciones, vectorizamos las reviews utilizando el Hashing Trick explicado anteriormente, entrenamos la red y la utilizamos para predecir. Se obtuvo una puntuación de `2.04606` en Kaggle.

5.4.3. Peso doble para Summary

Manteniendo constante el número de iteraciones (40) y teniendo en cuenta que el campo *Summary* es el título de la review, se nos ocurrió ponderarlo con un peso mayor frente al campo *Text* ya que, el primero, es un resumen del segundo.

Para ello, modificamos nuestro vectorizador de textos para que incremente en dos unidades la dimensión indicada por la función de hash cuando se vectorizan palabras del campo *Summary*.

Mientras que dejamos que incremente una unidad para las dimensiones hasheadas por palabras en el campo *Text*.

Se obtuvo una puntuación de `2.21231` en Kaggle. La cual es peor que en los intentos anteriores, por lo que descartamos esta idea y continuamos ponderando los features *Summary* y *Text* de la misma forma.

5.4.4. Recorte de 2/3 de las reviews puntuadas con 5

Frente al desbalance del set de entrenamiento estudiado para la entrega de diseño (En el cual vimos que un 75 % de las reviews tienen *Prediction* 5), decidimos probar entrenando la red excluyendo 2/3 de estas reviews. Las mismas fueron vectorizadas mediante el Hashing Trick usual con un peso equitativo para los campos *Summary* y *Text*, y estos vectores utilizados para entrenar la red neuronal con 40 iteraciones. Se obtuvo una puntuación de `2.08896` en Kaggle. La cual es peor que en el intento anterior (con el set completo), por lo que se descartó esta idea.

5.4.5. Reducción de iteraciones realizadas durante entrenamiento de la red

Consideramos que las redes estaban fallando en generalizar correctamente, debido a que estaban realizando una gran cantidad de iteraciones al momento de entrenarse. Por lo tanto, decidimos reducir la cantidad de las mismas para resolver este problema (De 40 a 25). Se obtuvo una puntuación de `2.04423` en Kaggle. La puntuación de la entrega fue superior, por lo que se consiguió una mejora en la capacidad de predicción de la red. Al ver una mejora, continuamos entrenando con pocas iteraciones.

5.4.6. Texto de reseñas agrupados en bigramas

Una variante utilizada fue agrupar el texto de las reviews en bigramas durante el preprocesamiento. Se vectorizaron las reviews con 127 dimensiones y se iteró 25 veces sin embargo esto no produjo mejores resultados a los ya obtenidos. La puntuación en Kaggle fue de `2.32407`.

Se intentó pre-procesar con n-gramas mayores sin éxito por problemas de *timeout* en PySpark.

5.4.7. Utilización de tabla de hash de mayor dimensión

Para reducir el numero de colisiones y por consiguiente el error asociado al vectorizar las reseñas decidimos incrementar el tamaño de la tabla de hash utilizada de 127 a 1009. Proseguimos a entrenar la red con 25 pasadas. Este cambio logró una mejora significativa en la precisión de la red ya que se logró una puntuación de `1.02591`. Continuamos aumentando la cantidad de dimensiones, esta vez de 1009 a 1511. Entrenamos la red con 25 pasadas y obtuvimos un resultado de `0.90247` en Kaggle.

Continuamos aumentando la cantidad de dimensiones a 2003, pero la computadora utilizada no disponía de la cantidad de memoria suficiente para realizar la tarea, por lo que no fue posible entrenar la red con esta cantidad de dimensiones.

Luego, realizamos un intento modificando la cantidad de iteraciones de 25 a 40 con 1511 dimensiones, cuyo resultado fue de `0.91549`.

Finalmente, realizamos un último intento con 25 iteraciones y 1709 dimensiones, pero el resultado fue el mismo que en el caso de las 2003 dimensiones.

5.5. Naïve Bayes

Como describimos en el diseño Naïve Bayes es una extensión de Bayes para n características, las cuales se asumen independientes. Aplicamos su formula

$$P(y|x_1,..,x_n) = \frac{P(y)\prod_{i=1}^{n} P(x_i|y)}{P(x_1,..,x_n)}$$

En donde 'y' seria la puntuación y 'x' cada *feature* que en nuestro caso corresponde a cada palabra. Entonces vamos a calcular a

1. P(y) como la $\frac{review-de-clase-x}{total-de-reviews}$
2. $P(x|y)$ como la frecuencia de la palabra x en la clase y $p_{palabra} = \frac{apariciones}{total}$
3. $P(x_1,..,x_n)$ como la probabilidad de que en un review aparezcan esas palabras.

Luego vemos para cada clase cual tiene mas probabilidad de ser elegida. El resultado en Cross Validation no es muy bueno siendo al rededor del 28 % Una desventaja de este algoritmo es que no puede ser entrenado, una forma de mejorarlo es consiguiendo mas datos, que por otro lado lo hace mas lento.

Mediante algoritmo se obtuvo una puntuación de `2.68657` en Kaggle. Dicha puntuación se condice con el resultado de Cross Validation, que no era bueno.

5.6. Decision Trees

Otro de los algoritmos utilizados fue el de Arboles de Decisión. Si bien este algoritmo no fue propuesto en la entrega de diseño, al ser un algoritmo sencillo de utilizar y con un orden de ejecución logarítmico [6] consideramos interesante probar este algoritmo y compararlo con nuestras otras soluciones.

Se realizaron dos pruebas con este algoritmo, en ambas se entreno al árbol con los parámetros por defecto del mismo pero se modificaron las dimensiones utilizadas durante la vectorización de los sets de datos. En la primer prueba se utilizaron 129 dimensiones y se obtuvo un resultado de `1.77059`, mientras que en la segunda se aumento las dimensiones del mismo a 1009 y logramos un resultado de `1.32765`. Al igual que en la red neuronal, el aumento de dimensiones durante *Hashing Trick* impacta positivamente en la precisión del modelo ya que se reduce el error por colisiones de palabras distintas.

Es importante notar que este modelo es susceptible a sets de datos desbalanceados [6], como el utilizado en el presente trabajo, por lo que los resultados podrían haber sido mejores en el caso de haber entrenado al árbol con un set balanceado. Sin embargo, por limitaciones de tiempo no fue posible realizar esta prueba por lo que no podemos afirmar cuan significativa hubiese sido la mejora de la predicción.

5.7. Tabla de submits realizados en Kaggle

Predictor			**Puntaje**
Vowpal Wabbit			0.41703
Dynamic Vector Prediction			5.06451
Naive Bayes			2.68657
Perceptrón			5.13634
Decision Trees	Hashing Trick de 127 dimensiones		1.77059
	Hashing Trick de 1009 dimensiones		1.32765
Red Neuronal	Random Search de hiperparámetros		4.77462
	Pesos idénticos para Summary y Text		2.04606
	Peso doble para Summary		2.21231
	Recorte de 2/3 de las reviews puntuadas con 5		2.08896
	Reducción de iteraciones		2.04423
	Preprocesamiento con bigramas		2.32407
	Aumento de las dimensiones en Hashing Trick	1009 dim. 25 iter.	1.02591
		1511 dim. 25 iter.	0.90247
		1511 dim. 40 iter.	0.91549

Cuadro 2: Tabla de submits realizados en Kaggle

6. Algoritmo final utilizado

Como explicamos a lo largo del informe, no nos limitamos a utilizar un solo algoritmo sino que probamos prediciendo las reviews con varios. Sin embargo, al encontrar un predictor con un mejor resultado inicial, como lo fue Redes Neuronales, nos dedicamos a ahondar en el mismo modificando no sólo sus hiperparámetros sino también el previo pre-procesamiento y vectorización de los datos.

Como se puede ver en la tabla de puntuaciones, nuestros mejores resultados fueron obtenidos con pre-procesamiento sin *stemming* ni n-gramas, muchas dimensiones al realizar Hashing Trick (Entre 1000 y 1500 que fue el máximo que pudimos probar con éxito) y pocas iteraciones al entrenar la red (Alrededor de 25, para que la red no se ajuste a predecir sólo el *training* set).

Cabe destacar que luego de explotar los recursos de la red también obtuvimos muy buenos resultados, desde el principio, con Decision Trees para el cual también mejoró el puntaje significativamente al aumentar la cantidad de dimensiones de la vectorización de reviews.

7. Comentario final

En términos generales, notamos que al mantener el modelo lo más simple posible se obtuvieron los mejores resultados, es decir sin aplicar transformaciones como *stemming*, *shingles*, etcétera a los datos.

Por otro lado, pudimos observar que las pruebas que daban bien con el *training* set no necesariamente lo hacían con el set de prueba. Esto ocurría porque el algoritmo no generalizaba correctamente a causa del *over-fitting*. La solución propuesta fue iterar a la red una menor cantidad de veces durante su entrenamiento.

Al no obtener resultados satisfactorios, luego de buscar los mejores hiperparámetros posibles mediante *random search*, decidimos aumentar las dimensiones del vector resultante de Hashing

Trick. Este cambio fue el de mayor impacto en la precisión del algoritmo, ya que logramos resultados significativamente mejores que con las anteriores estrategias aplicadas. Sin embargo, esta solución se encuentra fuertemente limitada por los recursos de hardware disponibles, ya que con mas de 1700 dimensiones, no fue posible procesar los datos con ningún predictor ya que el mismo utilizaba cantidades de memoria superiores a las disponibles en nuestros equipos.

8. Apéndices

8.1. Apéndice I: Requisitos para ejecutar el código presentado

Los siguientes lenguajes y librerías fueron utilizados durante este trabajo:

- Python 2.7.12
- PySpark
- graphviz==0.5.1
- numpy==1.11.2
- pandas==0.19.1
- python-dateutil==2.6.0
- pytz==2016.7
- scikit-learn==0.18.1
- scipy==0.18.1
- six==1.10.0
- sklearn==0.0

Estos requisitos se encuentran en el archivo `requirements-python2712.txt`. Algunos notebooks de Jupyter se realizaron en Python 3 por lo que los requisitos para correrlos se encuentran en el archivo `requirements-python3.txt` del repositorio del presente trabajo.

En ambos casos, las librerías pueden instalarse fácilmente mediante `pip`. Ejecutando el siguiente comando:

```
pip install -r requirements-python2712.txt
```

O,

```
pip install -r requirements-python3.txt
```

para instalar las librerías utilizadas con Python 3.

8.2. Apéndice II: Instrucciones de uso de los algoritmos utilizados

8.2.1. Aclaración inicial sobre los clasificadores

Todos los clasificadores desarrollados aceptan únicamente sets de datos cuyos textos de review fueron previamente vectorizados y presentan un formato de la forma `Id,Prediction,0,1,2,...,n` para el *train* set o `Id,0,1,2,...,n` para el caso de *test* set.

8.2.2. parallel_preprocess.ipynb

Jupyter notebook que ejecuta todas las funciones de pre-procesamiento sobre cada campo Summary y Text de cada registro. Para utilizarlo, descomentar las funciones de la lista `pipeline` que se desean ejecutar y cambiar los archivos input y output en la llamada a la función `parallel_preprocess`. Por default, las funciones esperan un *train* set, en caso de querer pre-procesar un *test* set, asegurarse de agregar como tercer parámetro `test=True`.

8.2.3. word2matrix.py

Script de Python que vectoriza los sets de reviews por medio de Hashing Trick. El módulo contiene dos funciones, una que vectoriza los campos *Summary* y *Text* de forma equitativa y otra que le asigna un peso mayor a los miembros de *Summary*. La función a utilizar se determina cambiando el nombre de la llamada en la última línea del script. También se puede cambiar el peso a designar a *Summary* y la cantidad de dimensiones del Hashing Trick modificando las constantes `SUMMARY_W` y `prime_number` respectivamente.

Un ejemplo de uso para vectorizar un *test* set:

```
python word2matrix.py testinput.csv testoutput.csv test
```

Un ejemplo de uso para vectorizar un *train* set:

```
python word2matrix.py traininput.csv trainoutput.csv train
```

8.2.4. Perceptrón

Para ejecutar el script de Perceptrón simplemente se deben editar los parámetros presentes en el código fuente. Estos son, `train_to_read`, `prime_number` y `number_of_laps`.

El script se corre de la siguiente manera:

```
python perceptron.py
```

El resultado del mismo es un `.csv` con las predicciones realizadas.

8.2.5. Red Neuronal

Hay dos scripts, `multilayer_perseptron_fitting.py` y `multilayer_perseptron_prediction.py`. El primero sirve para entrenar la red neuronal, ahí se le determinan las constantes mencionadas anteriormente que fueron encontradas por el *random search*. Se le modifica una constante que indica el source de la matriz a entrenar. El segundo sirve para hacer las predicciones, tanto el Cross Validation contra el *train* set como para hacer un Kaggle submit contra el test.

8.2.6. Naïve Bayes

Este script se ejecuta sin argumentos pasados por linea de comandos, ya que los mismos se determinan dentro del script mismo, editando las variables `train_to_read` y `test_to_read`

Por lo tanto el script se ejecuta de la siguiente forma:

```
python bayes.py
```

El resultado de dicha ejecución es un archivo `.csv` con las predicciones realizadas por el algoritmo.

8.2.7. Decision Trees

El script que ejecuta el árbol de decisión no recibe argumentos por linea de comandos, en cambio, se deben modificar una serie de constantes ubicadas en el inicio del script. Estas son:

- `TEST_F`: Path con el archivo que se quiere predecir.
- `TRAIN_F`: Path con el archivo de entrenamiento.
- `PRED_F`: Path con el archivo en donde se guardaran las predicciones.
- `TREE_DUMP_F`: Path con el archivo en donde se persistirá el árbol entrenado.

El script produce un archivo `.csv` luego de ejecutarse de la siguiente manera:

```
python decision_tree.py
```

9. Bibliografía

Referencias

[1] Koby Crammer and Yoram Singer. On the algorithmic implementation of multiclass kernel-based vector machines. http://jmlr.csail.mit.edu/papers/volume2/crammer01a/crammer01a.pdf, 2001.

[2] Thorsten Joachims. Text categorization with support vector machines: Learning with many relevant features. https://www.cs.cornell.edu/people/tj/publications/joachims_98a.pdf, 1997.

[3] Luis Argerich. For what kind of classification problems is svm a bad approach. https://www.quora.com/For-what-kind-of-classification-problems-is-SVM-a-bad-approach/answer/Luis-Argerich?srid=kXaw.

[4] sickit learn. sickit-learn documentation. http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC.

[5] A. Barbetta S. Lazzari L. Mintrone M. Porto. Multilayer perceptron analysis. https://github.com/manuporto/food-reviews/blob/master/classifiers/multilayer_perceptron_prediction/multilayer_perseptron_analisis.

[6] sickit learn. sickit-learn documentation. http://scikit-learn.org/stable/modules/tree.html.