**Text Mining II**

**Análisis de Sentimientos:** **Técnica** que nos permite **analizar un fragmento de texto** y **determinar qué sentimiento contiene**. Es una **combinación de aprendizaje automático** y **procesamiento de lenguaje natural (NLP)**. Se puede pensar que estamos abordando un **problema de clasificación**, donde **generamos las features** con las **técnicas** de preprocesamiento vistas en **Text Mining I**.

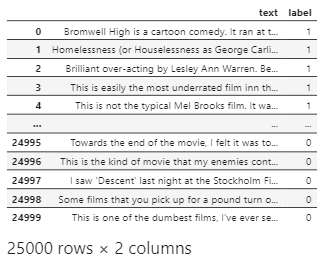
**Clasificación**: Vamos a trabajar con un dataset de reviews de imdb, usando regresión logística para predecir, a partir del texto del review, si la calificación será positiva o negativa.

**En Python:**

train\_file\_path = ‘../Data/imdb\_train.zip’

data\_train = pd.read\_csv(train\_file\_path, sep=’/t’)

data\_train



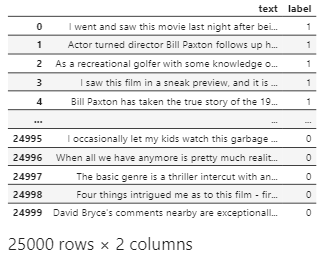
data\_train.label.value\_counts()



test\_file\_path = ‘../Data/imdb\_test.zip’

data\_test = (test\_file\_path, sep =’/t’)

data\_test



data\_test.label.value\_counts()



from nltk import RegexpTokenizer

from nltk.stem.snowball import SnowBallStemmer

from nltk.corpus import stopwords

from sklearn.feature\_extraction.text import CountVectorizer

Vamos a limpiar los datos de train y test: Con un tokennizer vamos a eliminar los signos de puntuación y tags html; con stemming vamos a obtener las raíces de las palabras en minúscula y vamos a eliminar stopwords.

**En Python:**

def clean\_review(review\_text, tokenizer, stemmer, stopwords):

#tokens (eliminamos todos los signos de puntuación)

words = tokenizer.tokenize(review\_text)

#stemming: raíz y minúsculas

stem\_words = [stemmer.stem(x) for x in words]

#print(stem\_words)

#eliminamos stopwords (ya pasaron por stem)

clean\_words = [x for x in stem\_words if x not in stopwords]

#print(clean\_words)

result = ‘ ‘,join(clean\_words)

return(result)

Ejemplo sobre un registro de train:

review\_text = data\_train.test[1]

print(‘antes: ’, review\_text)

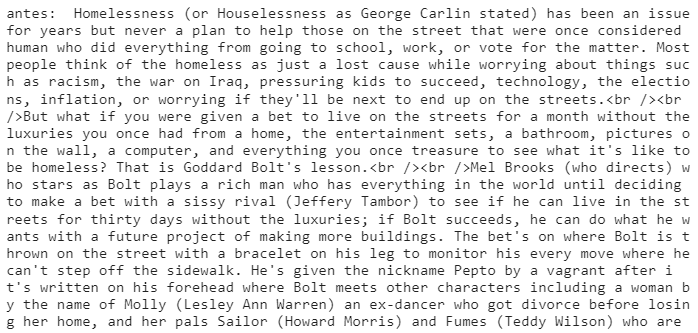
tokenizer = RegexpTokenizer(r’/w+’)

englishStemmer = SnowballStemer(‘english’)

stopwords\_en = stopwords.words(‘english’)

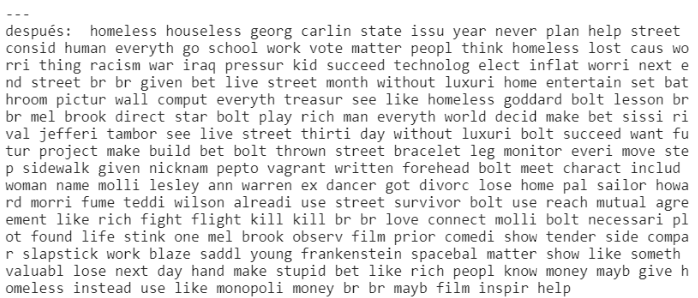
stopwords\_en\_stem = [englishStemmer.stem(x) for x in stopwords\_en]

review\_text\_clean = clean\_review(review\_text, tokenizer, englishStemmer, stopwords\_en\_stem)



print(‘---’)

print(‘después: ’, review\_text\_clean)



Vamos a procesar todos los registros en los datasets de train y test:

clean\_train = [clean\_review(x, tokenizer, englishStemmer, stopwords\_en\_stem) for x in data\_train.text]

#clean\_train[0:5]

clean\_test = [clean\_review(x, tokenizer, englishStemmer, stopwords\_en\_stem) for x in data\_test.text]

#clean\_test[0:5]

Con CountVectorizer vamos a transformar los datos de train y test:

**En Python:**

count\_vectorizer = CountVectorizer()

count\_vectorizer.fit(clean\_train)

X\_train\_sparse = count\_vectorizer.transform(clean\_train)

X\_test\_sparse = count\_vectorizer.transform(clean\_test)

X\_train = pd.DataFrame(X\_train\_spare.todense(), columns = count\_vectorizer.get\_feature\_names())

y\_train = data\_train.label

X\_test = pd.DataFrame(X\_test\_spare.todense(), columns = count\_vectorizer.get\_feature\_names())

y\_test = data\_test.label

Ahora, con la regression logística vamos a predecir el valor de label:

**En Python:**

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score, confusion\_matrix

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

Vamos a ver **para qué valor de C logramos un modelo que performee mejor**. **Valores más pequeños de C** implican **regularización más fuerte**.

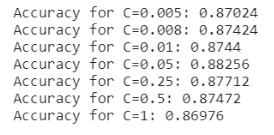
X\_train\_train, X\_train\_val, y\_train\_train, y\_train\_val = train\_test\_split(X\_train, y\_train, train\_size = 0.75, shuffle = True, random\_state = 147)

for c in [0.005, 0.008, 0.01, 0.05, 0.25, 0.5, 1]:

lr = LogisticRegression(C=c, solver = ‘newton-cg’, penalty = ‘l2’)

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

print(‘Accuracy for C=%s: %s’ % (c, accuracy\_score(y\_train\_val, lr.predict(X\_train\_val))))



Entrenamos el modelo final:

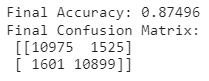
**En Python:**

final\_model = LogisticRegression(C = 0.05, solver = ‘newton-cg’, penalty = ‘l2’)

final\_model.fit(X\_train, y\_train)

print(‘Final Accuracy: %s’ % accuracy\_score(y\_test, final\_model.predict(X\_test)))

print(‘Final Confusion Matrix: /n %s’ % confusion\_matrix(y\_test, final\_model.predict(X\_test)))



Ahora vamos a ver cuáles son las 3 palabras más discriminantes tanto positiva como negativamente. Esto lo conseguimos viendo cuáles son las palabras (features) asociadas a los coeficientes máximos y mínimos de la regresión logística. Entonces, lo que vamos a hacer es construir un dataframe de 2 columnas: la columna word (cada una de las palabras del corpus), y la columna coef (el coeficiente de regresión logística asociado a esa palabra). Ordenando este dataset por sus coeficientes en forma descendente podremos obtener, mirando los primeros y los últimos registros, cuáles son las palabras que más influyen positivamente, y cuáles las que más influyen negativamente sobre el algoritmo.

**En Python:**

feature\_to\_coef = pd.DataFrame(columns = [‘word’, ‘coef’])

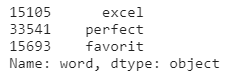
feature\_to\_coef.word = count\_vectorizer.get\_feature\_names()

feature\_to\_coef.coef = final\_model.coef\_[0]

feature\_to\_coef\_sort\_desc = feature\_to\_coef.sort\_values(by=’coef’, ascending = False)

positive\_words = feature\_to\_coef\_sort\_desc.word[0:3]

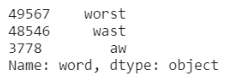
positive\_words



feature\_to\_coef\_sort\_asc = feature\_to\_coef.sort\_values(by=’coef’, ascending = True)

negative\_words = feature\_to\_coef\_sort\_asc.word[0:3]

negative\_words



Ahora vamos a graficar estos resultados para ver cómo se distribuyen sus valores:

**En Python:**

columns = np.concatenate ([positive\_words.values, negative\_words.values])

columns



data\_plot = X\_trai.loc[:, columns]

data\_plot.reset\_index(inplace = True)

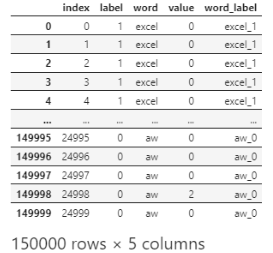
data\_plot[‘label’] = y\_train

data\_plot\_long = pd.melt(data\_plot, id\_vars = [‘index’, ‘label’], var\_name = ‘word’, value\_name = ‘value’)

# value\_vars = data\_plot.columns[1:len(data\_plot.columns)-1]

data\_plot\_long[‘word\_label’] = data\_plot\_long.word + [‘\_‘ + str(x) for x in data\_plot\_long.label]

data\_plot\_long

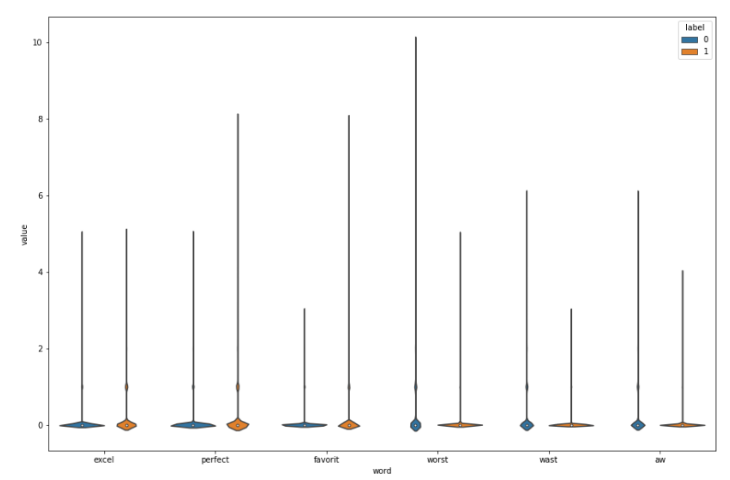


data\_plot\_long.sort\_values(by=’index’).head()



plt.figure(figsize=(15,10))

sns.violinplot(x = data\_plot\_long.word, y = data\_plot\_long.value, hue = data\_plot\_long.label)



Ahora vamos a entrenar una regresión logística que use como features el resultado de SVD.

feature\_to\_coef.shape



Venimos trabajando con un modelo que fue entrenado con más de 50.000 features. Vamos a ver qué pasa si lo reducimos a 200 features.

**En Python:**

from sklearn.decomposition import TruncatedSVD

svd = TruncatedSVD(n\_components = 200)

X\_train\_svd = svd.fit\_transform(X\_train)

X\_test\_svd = svd.transform(X\_test)

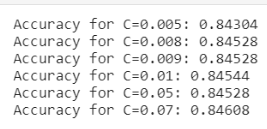
X\_train\_svd\_train, X\_train\_svd\_val, y\_train\_train, y\_train\_val = train\_test\_split(X\_train\_svd, y\_train, train\_size = 0.75, shuffle = True, random\_state = 147)

For c in [0.005, 0.008, 0.009, 0.01, 0.05, 0.07]:

lr = LogisticRegression(C=c, solver = ‘newton-cg’, penalty = ‘12’)

lr.fit(X\_train\_svd\_train, y\_train\_train)

print(‘Accuracy for C=%s: %s’ % (c, accuracy\_score(y\_train\_val, lr.predict(X\_train\_svd\_val))))

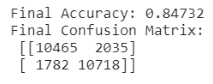


final\_model\_svd = Logisticregression(C = 0.01, solver = ‘newton-cg’, penalty = ‘12’)

final\_model\_svd.fit(X\_train\_svd, y\_train)

print(‘Final Accuracy: %s’ % accuracy\_score(y\_test, final\_model.predict(X\_test\_svd)))

print(‘Final Confusion Matrix: %s’ % confusion\_matrix(y\_test, final\_model.predict(X\_test\_svd)))



Cuando las **features** se componen **de una sola palabra**, como estuvimos trabajando hasta ahora, se las llama **1-grama o unigrama**. Podríamos agregar más poder predictivo si agregáramos secuencias de **dos** **o tres palabras** (**bigramas** o **trigramas**). Ejemplo: no es lo mismo analizar “no me gustó la película” todo junto que separado, ya que gustó va fuertemente asociada a una experiencia positiva, pero en este caso quiere decir todo lo contrario. Scikit-learn nos ayuda a construir estas features usando el argumento **ngram\_range** con cualquiera de las clases ‘vectorizador’.

**En Python:**

count\_vecotrizer\_bigram = CountVectorizer(ngram\_range = (1,2))

count\_vecotrizer\_bigram.fit(clean\_train)

X\_train\_bigram\_sparse = count\_vecotrizer\_bigram.transform(clean\_train)

X\_test\_bigram\_sparse = count\_vecotrizer\_bigram.transform(clean\_test)

# X\_train\_bigram = pd.DataFrame(X\_train\_bigram\_sparse.todense(), columns = count\_vectorizer\_bigram.get\_feature\_names())

# X\_test\_bigram = pd.DataFrame(X\_test\_bigram\_sparse.todense(), columns = count\_vectorizer\_bigram.get\_feature\_names())

# usamos las matrices esparsas porque rompe si trato de convertirlos en densas para esta cantidad de features.

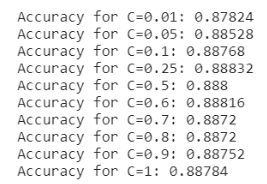
X\_train\_bigram\_train, X\_train\_bigram\_val, y\_train\_train, y\_train\_val = train\_test\_split(X\_train\_bigram\_sparse, y\_train, train\_size = 0.75, shuffle = True, random\_state = 147)

for c in [0.01, 0.05, 0.1, 0.25, 0.5, 0.6, 0.7, 0.8, 0.9, 1]:

lr = LogisticRegression(C=c, solver = ‘newton-cg’, penalty = ‘12’)

lr.fit(X\_train\_bigram\_train, y\_train\_train)

print(‘Accuracy for C=%s: %s’ % (c, accuracy\_score(y\_train\_val, lr.predict(X\_train\_bigram\_val))))

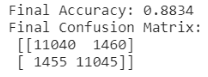


final\_model\_bigram = LogisticReegression(C = 0.25, solver = ‘newton-cg’, penalty 0 ‘12’)

final\_model\_bigram.fit(X\_train\_bigram\_sparse, y\_train)

print(‘Final Accuracy: %s’ % accuracy\_score(y\_test, final\_model\_bigram.predict(X\_test\_bigram\_sparse)))

print(‘Final Confusion Matrix: /n %s’ % confusion\_matrix(y\_test, final\_model\_bigram.predict(X\_test\_bigram\_sparse)))



**TF-IDF**: Es mejor que CountVectorizer porque además de la frecuencia de las palabras presentes en el corpus, tiene en cuenta la importancia de las mismas. Vamos a entrenar la regresión logística usando como features la representación tf-idf de unigramas, bigramas y trigramas; y vamos a ver cuáles son los n-gramas más discriminantes:

**En Python:**

from sklearn.feature\_extraction.text import TfidTransformer

count\_vectorizer = CountVectorizer(ngram\_range = (1,3))

count\_vectorizer.fit(clean\_train)

X\_train\_sparse = count\_vectorizer.transform(clean\_train)

X\_test\_sparse = count\_vectorizer.transform(clean\_test)

transformer = TfidTransformer()

X\_train\_ngram\_encoding = transformer.fit\_transform(X\_train\_sparse)

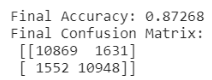
X\_test\_ngram\_encoding = transformer. transform(X\_test\_sparse)

final\_model\_tfidf = LogisticRegression(C = 1, solver = ’newton-cg’, penalty = ‘12’)

final\_model\_tfidf.fit(X\_train\_ngram\_encoding, y\_train)

print(‘Final Accuracy: %s’ % accuracy\_score(y\_test, final\_model\_tfidf.predict(X\_test\_ngram\_encoding)))

print(‘Final Confusion Matrix: /n %s’ % confusion\_matrix(y\_test, final\_model\_tfidf.predict(X\_test\_ngram\_encoding)))



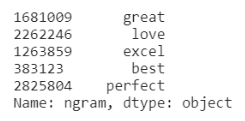
feature\_to\_coef = pd.DataFrame(columns = [‘ngram’, ‘coef’])

feature\_to\_coef.ngram = count\_vectorizer.get\_feature\_names()

feature\_to\_coef.coef = final\_model\_tfidf.coef\_[0]

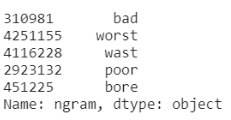
feature\_to\_coef\_sort\_desc = feature\_to\_coef.sort\_values(by = ‘coef’, ascending = False)

feature\_to\_coef\_sort\_desc.ngram[0:5]



feature\_to\_coef\_sort\_asc = feature\_to\_coef.sort\_values(by = ‘coef’, ascending = True)

feature\_to\_coef\_sort\_asc.ngram[0:5]



Los ngrams más discriminantes son unigramas.

**Conclusiones:**

* A partir de los algoritmos de Machine Learning, construimos un modelo de clasificación de textos.
* Con las técnicas de procesamiento de Text Mining I pudimos transformar los inputs (textos) para construir el conjunto de features que alimentaron el modelo.
* El modelo se evalúa de la misma manera en la que evaluamos modelos de clasificación.