Automate Sentiment Analysis of Textual Comments and Feedback
(Artificial Intelligence)

Internship Project Report

Submitted to:

Tata Consultancy Services

TCS iON RIO-125 Batch 01

Submitted by:

Sai Manvitha Nadella

B.Tech, Electronics and Communication Engineering

Sridevi Women's Engineering College, Gandipet

Project Industry Mentor: Debashis Roy

Start Date of Internship: 13th October 2020

Report Date: 17th December 2020

PREFACE

This report documents is the work done during the remote internship at Tata Consultancy Services on automate sentiment analysis of textual comments and feedback under the supervision of ***Mr. Debashis Roy****. The report give an overview of the tasks completed during the period of internship with technical details. Then the results obtained are discussed and analysed. I have tried my best to keep the report simple yet technically correct. I hope I succeed in my attempt.*

Sai Manvitha Nadella

ACKNOWLEDGEMENT

Simply, I could not have done this work without the lots of help I received cheerfully from Internship Team. The work culture at Tata Consultancy Services is really motivating. Everybody is such a friendly and cheerful companion here that work stress never comes in way. I would specially like to thank ***Mr. Debashis Roy*** *for constantly providing additional resources to complete the project. I am also very thankful to* ***Mr. Kulveer Kumar Sharma*** *for checking on us and evaluating our daily reports.*

Sai Manvitha Nadella

ABSTRACT

The report presents the tasks completed during the internship at Tata Consultancy Services.

Which are listed below:

1. *Identify and finalize a collection of English sentences or a large paragraph which will also cover contradictory statements.*
2. *Develop a deep learning model for detection and segmentation of sentiments whether positive, negative or neutral from the paragraph.*
3. *Enhance the previous algorithm to accurately predict the overall sentiment of the paragraph even if it contains contradictory statements.*
4. *Test the model for accuracy.*

All these tasks have been completed successfully and results were according to expectations.

Sai Manvitha Nadella

AUTOMATE SENTIMENT ANALYSIS OF TEXTUAL COMMENTS AND FEEDBACK

Introduction and Importing Data:

IMDB movie reviews dataset is used for this study. The dataset contains 50,000 reviews — 25,000 positive and 25,000 negative reviews. An example of a review can be seen in Fig 1, where a user gave a 10/10 rating and a written review for the Oscar-winning movie Parasite (2020). The number of stars would be a good proxy for sentiment classification. For example, we could pre-assign the following:

At least 7 out of 10 stars => positive (label=1)

At most 4 out of 10 stars => negative (label=0)

Figure 1

Here is a preview of how the dataset looks like:

	text	sentiment
0	For a movie that gets no respect there sure ar...	0
1	Bizarre horror movie filled with famous faces ...	0
2	A solid, if unremarkable film. Matthau, as Ein...	0
3	It's a strange feeling to sit alone in a theat...	0
4	You probably all already know this by now, but...	0
5	I saw the movie with two grown children. Altho...	0
6	You're using the IMDb. You've given some heft...	0
7	This was a good film with a powerful message o...	0
8	Made after QUARTET was, TRIO continued the qua...	0
9	For a mature man, to admit that he shed a tear...	0

Figure 2

EXPLORATORY TEXT ANALYSIS:

In order to proceed with exploratory text analysis, firstly the data has been separated into training and test set, and then the data is split into sub-sets.

```
***** Extract of train_string *****
After watching many of the "Next Action Star" reality TV eps TiVo taped
this gawd-awful tripe for me.

***** Extract of splits *****
['After', 'watching', 'many', 'of', 'the', '"Next', 'Action', 'Star"',
'reality', 'TV', 'eps', 'TiVo', 'taped', 'this', 'gawd-awful', 'tripe',
'for', 'me.']
```

No. of Strings Present:

print(f"Number of strings: {len(splits)}")

print(f"Number of unique strings: {len(set(splits))}")

```
Number of strings: 10388491
Number of unique strings: 410259
```

Most common strings:

freq_splits = FreqDist(splits)

*print(f"***** 10 most common strings ***** \n{freq_splits.most_common(10)}", "\n")*

```
[('the', 511118), ('a', 275901), ('and', 271298), ('of', 255015), ('to'
, 235463), ('is', 182645), ('in', 152812), ('I', 119122), ('that', 1141
24), ('this', 101978)]
```

Frequency of occurrence of html tags:

The example html tag: '

' would have been split into three strings: '<br', '/><br' and '/>' when we split the data based on white space. On side note, the
 tag seems to be used to break lines.

```
def summarise(pattern, strings, freq):
"""Summarise strings matching a pattern."""
compiled_pattern = re.compile(pattern)
matches = [s for s in strings if compiled_pattern.search(s)]
# Print volume and proportion of matches
print("{} strings, that is {:.2%} of total".format(len(matches), len(matches)/ len(strings)))
# Create list of tuples containing matches and their frequency
output = [(s, freq[s]) for s in set(matches)]
output.sort(key=lambda x:x[1], reverse=True)
return output
# Find strings possibly containing html tag
summarise(r"/?>?w*<|/>", splits, freq_splits)
```

```
271752 strings, that is 2.62% of total
```

```
('/><br', 90671),
('/>The', 12875),
('<br', 10799),
('/>I', 6609),
('/>This', 3938),
('/>In', 1732),
('/>It', 1709),
('/>If', 1645),
('it.<br', 1512),
('/>There', 1437),
('/>But', 1323),
('movie.<br', 1299),
('/>A', 1268),
('/>', 1262),
('film.<br', 1250),
('/>As', 1207),
('/>And', 1155),
("/>It's", 1115),
('/>What', 855),
('/>All', 743),
('/>My', 703),
('/>So', 658),
('/>One', 654),
('/>For', 592),
('time.<br', 548),
('/>To', 491),
('/>When', 481),
('them.<br', 447),
("/>I'm", 432),
('/>While', 408),
('/>You', 391),
('/>-', 388),
('/>Overall,', 371),
('well.<br', 363),
('/>Not', 359),
('/>However,', 357),
```

Frequency of Punctuation Marks:

```
summarise(r"\w+[_!&/](<\|}{\[\]]\w+", splits, freq_splits)
```

```
17859 strings, that is 0.17% of total
```

```
('and/or', 322),
('10/10', 229),
('8/10', 169),
('1/2', 160),
('writer/director', 155),
('4/10', 154),
('7/10', 147),
('1/10', 137),
('3/10', 124),
('9/10', 114),
('/>7/10', 90),
('/>4/10', 88),
('2/10', 86),
('*1/2', 82),
('10/10.', 78),
('/>8/10', 76),
('his/her', 73),
('/>10/10', 73),
('8/10.', 72),
('9/11', 71),
('T&A', 70),
('/>3/10', 66),
('9/10.', 65),
('7/10.', 62),
('the<br', 61),
('/>9/10', 54),
('/>2/10', 50),
('/>1/10', 50),
('4/10.', 48),
('(*1/2)', 45),
('he/she', 45),
('actors/actresses', 45),
('0/10', 42),
```

Most Frequent Stop-words:

```
stop_words = stopwords.words("english")
print(f"There are {len(stop_words)} stopwords.\n")
print(stop_words)
```

```
There are 179 stopwords.

['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "y
ou're", "you've", "you'll", "you'd", 'your', 'yours', 'yourself', 'your
selves', 'he', 'him', 'his', 'himself', 'she', "she's", 'her', 'hers',
'herself', 'it', "it's", 'its', 'itself', 'they', 'them', 'their', 'the
irs', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'that', "th
at'll", 'these', 'those', 'am', 'is', 'are', 'was', 'were', 'be', 'been
', 'being', 'have', 'has', 'had', 'having', 'do', 'does', 'did', 'doing
', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until'
, 'while', 'of', 'at', 'by', 'for', 'with', 'about', 'against', 'betwee
n', 'into', 'through', 'during', 'before', 'after', 'above', 'below', '
to', 'from', 'up', 'down', 'in', 'out', 'on', 'off', 'over', 'under', '
again', 'further', 'then', 'once', 'here', 'there', 'when', 'where', 'w
hy', 'how', 'all', 'any', 'both', 'each', 'few', 'more', 'most', 'other
', 'some', 'such', 'no', 'nor', 'not', 'only', 'own', 'same', 'so', 'th
an', 'too', 'very', 's', 't', 'can', 'will', 'just', 'don', "don't", 's
hould', "should've", 'now', 'd', 'll', 'm', 'o', 're', 've', 'y', 'ain'
, 'aren', "aren't", 'couldn', "couldn't", 'didn', "didn't", 'doesn', "d
oesn't", 'hadn', "hadn't", 'hasn', "hasn't", 'haven', "haven't", 'isn',
"isn't", 'ma', 'mightn', "mightn't", 'mustn', "mustn't", 'needn', "need
n't", 'shan', "shan't", 'shouldn', "shouldn't", 'wasn', "wasn't", 'were
n', "weren't", 'won', "won't", 'wouldn', "wouldn't"]
```

This Exploratory Text analysis will help to get an idea regarding what steps to proceed with in order to get an accurate sentiment prediction while creating or advancing with models.

In this Project report the performance of Logistic Regression and Naïve Bayes Algorithm are been accessed in determining the sentiment of a review.

LOGISTIC REGRESSION MODEL ANALYSIS AND PERFORMANCE:

For training, our feature matrix is greatly sparse, meaning there is a lot of zeros in this matrix as there are 25,000 rows and approximately 75,000 columns. So, what we're going to do is find the weight of each feature and multiply them with their corresponding TD-IDF values; sum all the values, pass it through a sigmoid activation function, and that's how we end up with the **Logistic Regression model**.

The advantages of applying a Logistic function in this case:

- The model handles sparse matrices really well, and
- The weights can be interpreted as a probability for the sentiments.

Model: Logistic regression

- $p(y=1|x)=\sigma(wTx)$p(y=1|x)=σ(wTx)
- Linear classification model
- Can handle sparse data
- Fast to train
- Weights can be interpreted

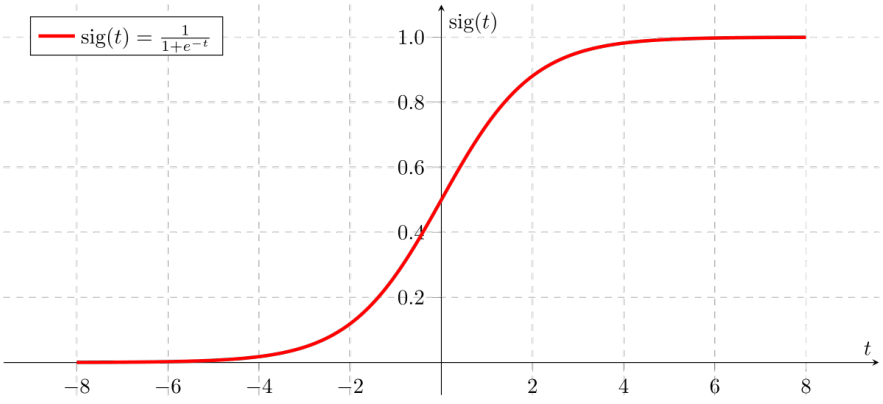


Figure 3

Transforming Documents into Feature Vectors

Below, we will call the fit_transform method on CountVectorizer. This will construct the vocabulary of the bag-of-words model and transform the sample sentence below into a sparse feature vector.

```
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer

count = CountVectorizer()
docs = np.array([
        'The sun is shining',
        'The weather is sweet',
        'The sun is shining, the weather is sweet, and one and one is two'])
bag = count.fit_transform(docs)
```

```
print(count.vocabulary_)
```

```
{'the': 6, 'sun': 4, 'is': 1, 'shining': 3, 'weather': 8, 'sweet': 5, 'and': 0, 'one': 2, 'two': 7}
```

```
print(bag.toarray())
```

```
[[0 1 0 1 1 0 1 0 0]
 [0 1 0 0 0 1 1 0 1]
 [2 3 2 1 1 1 2 1 1]]
```

Figure 4

Raw term frequencies: tf (t,d)—the number of times a term t occurs in a document d.

Word relevancy using term frequency-inverse document frequency

$$\text{tf-idf}(t, d) = \text{tf}(t,d) \times \text{idf}(t, d)$$

$$\text{idf}(t, d) = \log \frac{n_d}{1 + \text{df}(d, t)},$$

where n_d is the total number of documents, and df(d, t) is the number of documents d that contain the term t.

```
np.set_printoptions(precision=2)
```

```
from sklearn.feature_extraction.text import TfidfTransformer

tfidf = TfidfTransformer(use_idf=True, norm='l2', smooth_idf=True)
print(tfidf.fit_transform(count.fit_transform(docs)).toarray())
```

```
[[0.   0.43 0.   0.56 0.56 0.   0.43 0.   0.  ]
 [0.   0.43 0.   0.   0.   0.56 0.43 0.   0.56]
 [0.5  0.45 0.5  0.19 0.19 0.19 0.3  0.25 0.19]]
```

Figure 5

The equations for the Idf and Tf-Idf that are implemented in scikit-learn are:

$$\text{idf}(t, d) = log\frac{1 + n_d}{1 + \text{df}(d, t)}$$

The Tf-Idf equation that is implemented in scikit-learn is as follows:

$$\text{tf-idf}(t, d) = \text{tf}(t, d) \times (\text{idf}(t, d) + 1)$$

$$v_{\text{norm}} = \frac{v}{||v||_2} = \frac{v}{\sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}} = \frac{v}{\left(\sum_{i=1}^{n} v_i^2\right)^{\frac{1}{2}}}$$

Example:

$$\text{idf}(\text{" } is \text{ "}, d3) = log\frac{1 + 3}{1 + 3} = 0$$

Now in order to calculate the Tf-Idf, we simply need to add 1 to the inverse document frequency and multiply it by the term frequency:

$$\text{tf-idf}(\text{" } is \text{ "}, d3) = 3 \times (0 + 1) = 3$$

Calculation of Tf-Idf of the term "*is*":

```
tf_is = 3
n_docs = 3
idf_is = np.log((n_docs+1) / (3+1))
tfidf_is = tf_is * (idf_is + 1)
print('tf-idf of term "is" = %.2f' % tfidf_is)
```

```
tf-idf of term "is" = 3.00
```

$$\text{tfi-df}_{norm} = \frac{[3.39, 3.0, 3.39, 1.29, 1.29, 1.29, 2.0, 1.69, 1.29]}{\sqrt{[3.39^2, 3.0^2, 3.39^2, 1.29^2, 1.29^2, 1.29^2, 2.0^2, 1.69^2, 1.29^2]}}$$
$$= [0.5, 0.45, 0.5, 0.19, 0.19, 0.19, 0.3, 0.25, 0.19]$$
$$\Rightarrow \text{tfi-df}_{norm}(\text{" } is \text{ "}, d3) = 0.45$$

```
l2_tfidf = raw_tfidf / np.sqrt(np.sum(raw_tfidf**2))
l2_tfidf
```

```
array([0.5 , 0.45, 0.5 , 0.19, 0.19, 0.19, 0.3 , 0.25, 0.19])
```

```
tfidf = TfidfTransformer(use_idf=True, norm=None, smooth_idf=True)
raw_tfidf = tfidf.fit_transform(count.fit_transform(docs)).toarray()[-1]
raw_tfidf
```

```
array([3.39, 3.  , 3.39, 1.29, 1.29, 1.29, 2.  , 1.69, 1.29])
```

Data Preparation:

We will define a helper function to pre-process the text data as our text of reviews can contain special characters — html tags, emojis — that we would want to account for when training our model.

The special characters, emoji's and unwanted garbage values are removed throughout the string and added at the end of the string.

```
df.loc[0, 'review'][-50:]
```

```
'is seven.<br /><br />Title (Brazil): Not Available'
```

```
import re
def preprocessor(text):
    text = re.sub('<[^>]*>', '', text)
    emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)', text)
    text = re.sub('[\W]+', ' ', text.lower()) +\
        ' '.join(emoticons).replace('-', '')
    return text
```

```
preprocessor(df.loc[0, 'review'][-50:])
```

```
'is seven title brazil not available'
```

```
preprocessor("</a>This :) is :( a test :-)!")
```

```
'this is a test :) :( :)'
```

```
df['review'] = df['review'].apply(preprocessor)
```

As seen above, after applying the defined preprocessor function, the sample sentence was stripped of special characters; the emojis were also moved to the end of the sentence. This is so that our model can make use of the sequence of the text and also ascertain the sentiment of the emojis at the end of the sentences.

Tokenisation of Documents

The data is represented as a collection of words or tokens; and also be performing word-level pre-processing tasks such as stemming. To achieve this, we will utilize the natural language toolkit, or nltk.

Stemming is a technic that reduces the inflectional forms, and sometimes derivationally related forms, of a common word to a base form. For example, the words 'organizer' and 'organizing' stems from the base word 'organize'. So, stemming is conventionally referred to as a crude heuristic process that 'chops' off the ends of words, in the hope of achieving the goal correctly most of the time — this often includes the removal of derivational affixes.

```
from nltk.stem.porter import PorterStemmer

porter = PorterStemmer()

def tokenizer(text):
    return text.split()


def tokenizer_porter(text):
    return [porter.stem(word) for word in text.split()]
```

```
tokenizer('runners like running and thus they run')
```

```
['runners', 'like', 'running', 'and', 'thus', 'they', 'run']
```

```
tokenizer_porter('runners like running and thus they run')
```

```
['runner', 'like', 'run', 'and', 'thu', 'they', 'run']
```

```
import nltk

nltk.download('stopwords')
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\Administrator\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

```
True
```

```
from nltk.corpus import stopwords

stop = stopwords.words('english')
[w for w in tokenizer_porter('a runner likes running and runs a lot')[-10:]
if w not in stop]
```

```
['runner', 'like', 'run', 'run', 'lot']
```

Document classification via a logistic regression model:

With X and y as our feature matrices of TD-IDF values and target vector of sentiment values respectively, we are ready to split our dataset into training and test sets. Then, we will fit our training set into a Logistic Regression model.

Note that instead of manually hyperparameter tuning our model, we're using LogisticRegressionCV to specify the number of cross-validation folds we want to do to tune the hyperparameter — that is 5-fold cross-validation.

```
X_train = df.loc[:25000, 'review'].values
y_train = df.loc[:25000, 'sentiment'].values
X_test = df.loc[25000:, 'review'].values
y_test = df.loc[25000:, 'sentiment'].values
```

```
print("Review: ", X_train[1:2], "\n")
print("Sentiment: ", y_train[1])
```

```
Review:  ['ok so i really like kris kristofferson and his usual easy going delivery of lines in his movies age has helped him w
ith his soft spoken low energy style and he will steal a scene effortlessly but disappearance is his misstep holy moly this was
a bad movie i must give kudos to the cinematography and and the actors including kris for trying their darndest to make sense f
rom this goofy confusing story none of it made sense and kris probably didn t understand it either and he was just going throug
h the motions hoping someone would come up to him and tell him what it was all about i don t care that everyone on this movie w
as doing out of love for the project or some such nonsense i ve seen low budget movies that had a plot for goodness sake this h
ad none zilcho nada zippo empty of reason a complete waste of good talent scenery and celluloid i rented this piece of garbage
for a buck and i want my money back i want my 2 hours back i invested on this grade f waste of my time don t watch this movie o
r waste 1 minute of your valuable time while passing through a room where it s playing or even open up the case that is holding
the dvd believe me you ll thank me for the advice ']

Sentiment:  0
```

```
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import GridSearchCV

tfidf = TfidfVectorizer(strip_accents=None,
                        lowercase=False,
                        preprocessor=None)

param_grid = [{'vect__ngram_range': [(1, 1)],
               'vect__stop_words': [stop, None],
               'vect__tokenizer': [tokenizer, tokenizer_porter],
               'clf__penalty': ['l2'],
               'clf__C': [1.0, 10.0, 100.0]},
              {'vect__ngram_range': [(1, 1)],
               'vect__stop_words': [stop, None],
               'vect__tokenizer': [tokenizer, tokenizer_porter],
               'vect__use_idf':[False],
               'vect__norm':[None],
               'clf__penalty': ['l2'],
               'clf__C': [1.0, 10.0, 100.0]},
              ]

lr_tfidf = Pipeline([('vect', tfidf),
                     ('clf', LogisticRegression(random_state=0))])

gs_lr_tfidf = GridSearchCV(lr_tfidf, param_grid,
                           scoring='accuracy',
                           cv=5,
                           verbose=1,
                           n_jobs=-1)
```

After applying logistic regression model to the documents then the model is loaded from the disk and the accuracy is calculated using metric measures.

Model accuracy:

```
print('Best parameter set: %s' % gs_lr_tfidf.best_params_)
print('CV Accuracy:%.3f' % gs_lr_tfidf.best_score_)
```

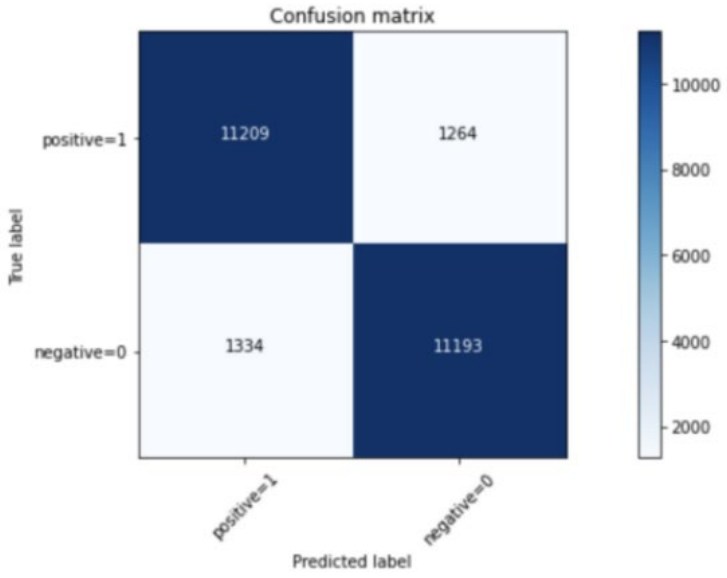
```
Best parameter set: {'clf__C': 10.0, 'clf__penalty': 'l2', 'vect__ngram_range': (1, 1), 'vect__stop_words': None, 'vect__tokeni
zer': <function tokenizer at 0x000001544E5C0488>}
CV Accuracy:0.897
```

```
clf = gs_lr_tfidf.best_estimator_
print('Test Accuracy: %.3f' % clf.score(X_test, y_test))
```

```
Test Accuracy: 0.899
```

89.9% accuracy is a pretty good score for relatively simple models.

Another way of looking at the accuracy of the classifier is through a confusion matrix.



The first row is for reviews which their actual sentiment values in the test set are 1. As you can calculate, out of 25,000 reviews, the sentiment value of 12,473 of them is 1; and out of these 12,473, the classifier correctly predicted 11,209 of them as 1.

It means, for 11,209 reviews, the actual sentiment values were 1 in the test set, and the classifier also correctly predicted those as 1. However, while the actual labels of 1,264 reviews were 1, the classifier predicted those as 0, which is pretty good.

What about the reviews with the sentiment value of 0? Let's look at the second row. It looks like there were a total 12,527 reviews which their actual sentiment values were 0.

The classifier correctly predicted 11,193 of them as 0, and 1,334 of them wrongly as 1. So, it has done a good job in predicting the reviews with sentiment values of 0.

A good thing about confusion matrix is that it shows the model's ability to correctly predict or separate the classes. In specific cases of a binary classifier, such as this example, we can interpret these numbers as the count of true positives, false positives, true negatives, and false negatives.

NAIVE BAYES MODEL ANALYSIS AND PERFORMANCE:

Naive Bayes it's a popular and easy to understand Supervised Probabilistic classification algorithm. The Naive Bayes Algorithm is based on the Bayes Rule which describes the probability of an event, based on prior knowledge of conditions that might be related to the event.

```
def pr(y_i):
    p = x[y==y_i].sum(0)
    return (p+1) / ((y==y_i).sum()+1)
```

```
x=trn_term_doc
y=trn_y

r = np.log((pr(1)/pr(0))
b = np.log((y==1).mean() / (y==0).mean())
```

```
r.shape,val_term_doc.shape
```

```
((1, 75132), (25000, 75132))
```

```
preds
```

```
matrix([[False, False, False, ...,  True,  True,  True]])
```

```
pre_preds = val_term_doc @ r.T + b
preds = pre_preds.T>0
(preds==val_y).mean()
```

```
0.81656
```

Advantages and disadvantages of Naive Bayes

Major advantages:

Simplicity

Requires small training set

Computationally fast

Scales linearly with the number of features and training examples

Disadvantages:

Strong feature independence assumption which rarely holds true in the real world. Remember, it's naive

May provide poor estimates, based on its independence assumption