Midterm

December 21, 2021

1 Sentiment Analysis of Film Reviews

This report will document the process of developing a binary classifier for sentiment analysis across a dataset of film reviews.

1.1 Introduction

1.1.1 Overview of Sentiment Analysis for Product and Service Reviews

Nowadays it is well-known that online product and service reviews have become extremely important, not only in the e-commerce industry, but for any type of business generally. Through a variety of platforms such as Google, Amazon, Yelp, etc, customers are able to express their opinions of any product or service in the form of reviews. Users who browse through these products and services very often rely on these reviews to make their purchasing decisions.

Natural language processing allows businesses and entrepreneurs to assess unstructured product reviews via sentiment analysis, which can be useful for "showcasing examination, advertising, item surveys, net advertiser scoring, item input, and client administration". [1] Sentiment analysis is a form of text classification that aims to determine attitudes or opinions within text. It is sometimes also called *opinion mining*. [1]

Possible attributes that may be extracted from product reviews are the **polarity** of the review (i.e. whether it is positive or negative), the **subject** of the review, and the **opinion holder** or author of the review. [1] Additionally, some work has been done on extracting **use cases** from reviews (i.e. what the customer used the product for). [2]

We may strive to distinguish facts from opinions within reviews. [1] Opinions moreover can be categorised into two separate classes: direct and comparative. [1] Direct opinions are straightforward statements of opinion, e.g. "This product is bad". [1] A comparative opinion would be something like "This product is better than the one offered by [some brand]". [1]

Sentiment analysis can be applied not only at the document level (i.e. the review as a whole); we can examine the sentiment of individual sentences, or even the clauses or words within a sentence. [1] Additionally we may aim to detect emotions within the review such as happiness, anger, or disappointment. [1] A variety of classification algorithms may be used such as naive Bayes, logistic regression, support vector machines, neural networks, etc. [1]

1.1.2 Objectives

As mentioned before, the objective is the development of a classifier for film reviews. Here I will only look at the polarity of the review to determine whether it is positive or negative. Since there

will only be two classes (positive and negative), we can consider this to be a binary classification problem.

On websites such as Amazon and Google, the polarity of the review is very often already indicated by the user in the form of some kind of rating, e.g. a star rating from 1 to 5. Still, some users will inevitably leave a review without rating the product if the rating portion of the review is optional. Additionally we must consider the vast amount of *unstructured* [1] product opinions on the internet outside of these platforms, for example in blog posts or social media.

Suppose then that a company wants to know how its product is being discussed on these other unstructured mediums. They must first determine whether the social media or blog post relates to the product, i.e. the **subject** of the text. Since the dataset I will be using is comprised of only film reviews, I do not need to consider this. However, it is worth noting that this would be very important if we were opinion mining over unstructured data. The second phase is the determination of the polarity of the text, and this is what this report will focus on. To this end, I will be using the following techniques which will be discussed in more detail in future sections: stemming, regular expressions, negation handling, TF-IDF text representation, and multinomial naive Bayes.

1.1.3 Dataset

I will be using the IMDB dataset offered by Stanford University, a large dataset of 50,000 polarized movie reviews that was used in their 2011 research paper Learning Word Vectors for Sentiment Analysis. They state that the dataset is symmetrically comprised of 25,000 negative reviews and 25,000 positive reviews, and that the negative reviews were rated 1-4 stars while the positive reviews were rated 7-10 stars. Reviews with ratings of 5 or 6 stars were ignored when compiling the dataset. For any given film, at most 30 reviews are included in the dataset. This is to prevent the dataset from being biased toward particular films. [3]

In regard to its structure, the dataset contains two columns of whose data types are strings: a **review** column which is the text of the review and a **sentiment** column whose values are either the string `positive` or the string `negative`.

Let us now read in the CSV file and take a look at the first few rows of the dataset:

[16]:
```
import pandas as pd

# I will be using df as an acronym for a pandas dataframe
df_large = pd.read_csv('data/imdb-reviews.csv')
df_large.head()
```

[16]:

	review	sentiment
0	One of the other reviewers has mentioned that …	positive
1	A wonderful little production. The…	positive
2	I thought this was a wonderful way to spend ti…	positive
3	Basically there's a family where a little boy …	negative
4	Petter Mattei's "Love in the Time of Money" is…	positive

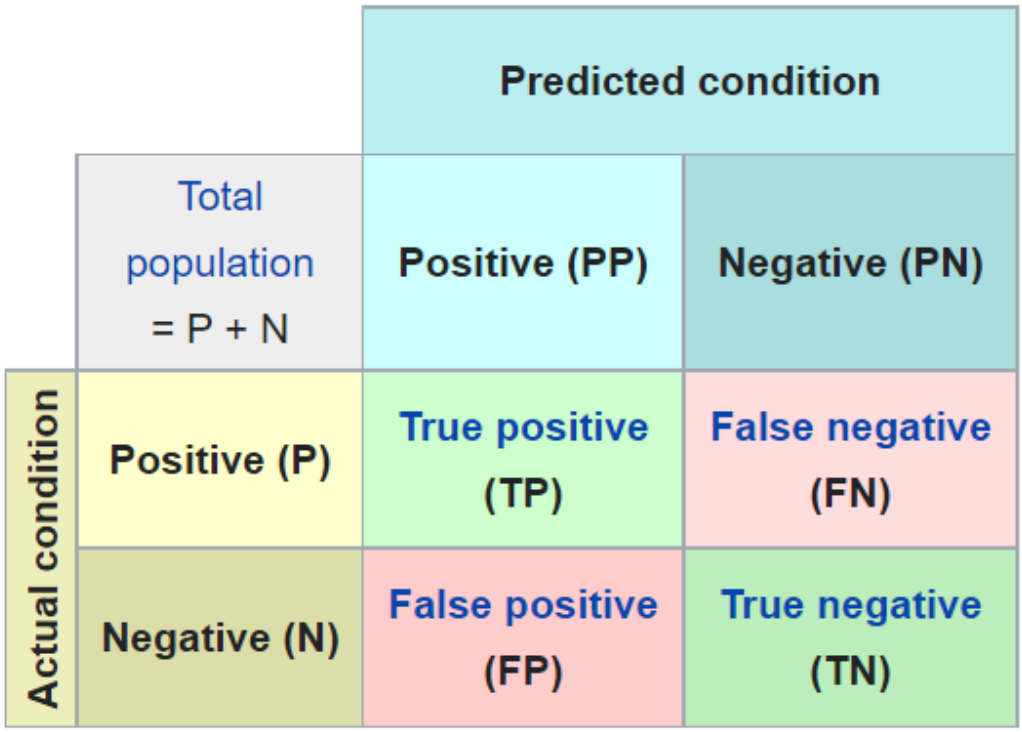
We can note from the above data that some HTML and unnecessary whitespace is present in the review data which we may need to consider when doing the preprocessing. Moreover, due to memory constraints, I will only be using 20% of the dataset, i.e. 10,000 rows.

1.1.4 Evaluation Methodology

Since I am building and training only one model that is not terribly complicated, it should be straightforward to use an extrinsic evaluation approach. If, for example, I were building a very complicated model that takes a long time to train and wanted to compare various models or factors within one model, an instrinsic evaluation approach such as perplexity might be preferable.

As extrinsic evaluation measures the performance of the model, I must decide what sort of performance metric will be used. Because I am dealing with a perfectly symmetric dataset (50% positive reviews and 50% negative reviews), using **accuracy** alone should suffice. To measure the acccuracy of the model, I can first calculate a confusion matrix like the one below.

Image source: Wikipedia



The values along the diagnal of the matrix are correct predictions: a TP is a correct prediction of `positive` where as a TN is a correct prediction of `negative`. An FP is an incorrect prediction of `positive`, and an FN is an incorrect prediction of `negative`. Using these values, the accuracy of the model can be calculated via the following formula:

(TP + TN) / (TP + TN + FP + FN)

Oftentimes, however, we will not be working with a symmetric dataset such as this one. In these cases, accuracy is usually not a favorable metric to use. Consider a case where we are searching through 10,000 social media posts and wish to identify whether or not these posts relate to a certain product. Assume further that only 500 of these posts actually relate to the product. A simple baseline classifier that predicts `negative` 100% of the time would achieve an accuracy of

95%. Under these conditions, we should consider other performance metrics such as precision, recall, and F-measure. [4]

With that being said, accuracy alone should be sufficient for our purposes.

1.2 Implementation

1.2.1 Pre-Processing

I have already read the CSV data into a Pandas dataframe stored in the variable `df_large`. First I will import the `re` and `nltk` libraries and download the stopwords. I will also confirm that the number of rows totals 50,000.

[17]:
```
import re
import nltk

# uncomment to download stopwords
# nltk.download('stopwords')

# check length
len(df_large)
```

[17]:
```
50000
```

Let's also confirm that there are 25,000 positive reviews and 25,000 negative reviews.

[18]:
```
positive_reviews = 0
negative_reviews = 0
for i in range(len(df_large)):
    if df_large['sentiment'][i] == 'positive':
        positive_reviews += 1
    elif df_large['sentiment'][i] == 'negative':
        negative_reviews += 1

positive_reviews == negative_reviews == 25000
```

[18]:
```
True
```

Confirm that there are no null values.

[19]:
```
import numpy as np
result = np.where(pd.isnull(df_large))
result[0].size == result[1].size == 0
```

[19]:
```
True
```

Confirm that there are no empty string values.

[20]:
```
result = np.where(df_large.applymap(lambda x: x == ''))
result[0].size == result[1].size == 0
```

[20]:
```
True
```

Sort the data by the `sentiment` column.

[21]:
```
df = df_large.sort_values('sentiment')
```

As mentioned before, I will only use 20% of the dataset. Take the first 5000 negative reviews and the last 5000 positive reviews and concatenate them into one dataframe.

[22]:
```
df = df[:5000].append(df[-5000:])
len(df)
```

[22]:
```
10000
```

Now reorder the indices of the resulting dataframe.

[23]:
```
df.index = pd.RangeIndex(len(df.index))
```

Let's take a look at the first review in its entirety.

[24]:
```
df['review'][0]
```

[24]:
```
'This film should have never been made. Honestly, I must admit that before I saw
it I had some serious doubts. The director is not a great actress, though she
did a lot of movies in Holland, and the young woman who took the main part is a
TV-personality with a constant smile on the face and not much self-criticism.
The actor who played the other main part I recently saw in Bride Flight and
although that film is better, he did not convince me than. To start with the the
story, I have not read the novel it is based upon, but the script that underlays
the film is something that might have been done with in mind kids having a
birthday party on a rainy Sunday afternoon, not someone of the same age as the
director who likes to watch a good movie. Something really disturbing were the
overdubbed dialogues, it was most of the time spoken out loud. My regards go to
the cameraman, at least he tried to make something out of it. It is a pity that
the film is edited lousy, if not, some scenes were certainly more credible.'
```

I will use stemming and forego lemmatization. Stemming is faster and using lemmaization often shows little to no improvement. I would use lemmaization if I needed to produce readable text. For text classification, stemming is often good enough.

Additionally I will add the string `br` to the stopwords to account for those HTML break tags, and I will moreover convert the stopwords list to a set for constant-time look-ups.

[25]:
```
from nltk.corpus import stopwords as nltk_stopwords
from nltk.stem.porter import PorterStemmer

stopwords = set(nltk_stopwords.words('english') + ['br'])
stemmer = PorterStemmer()
```

I will now define a function that will essentially process the words within clauses. It will take a list of words found within a clause as input and output a new processed list of words. This will handle stemming and negation and moreover ignore stopwords.

[26]:
```
def get_processed_words(words):
    processed_words = []
    has_not = False  # True if the clause contains the word 'not'
    for word in words:
        # if the string is 'not' then we ignore it but indicate that it was
        # found in the clause
        if word == 'not':
            has_not = True
            continue
        # otherwise if it is a stopword, we just ignore it
        elif word in stopwords:
            continue
        # at this point the string is neither 'not' nor a stopword so we stem it
        word = stemmer.stem(word)
        # if the word 'not' was found earlier, we need to prefix the stem
        # with 'NOT_'
        if has_not:
            processed_words.append('NOT_' + word)
        # otherwise just append it to the processed list of words
        else:
            processed_words.append(word)
    return processed_words
```

At the individual review level, I will process the document clause by clause. The reasoning for this is that I want to prefix words within a clause following the word `not` with `NOT_`.

[27]:
```
def get_processed_review(clauses):
    processed_review = []
    for clause in clauses:
        words = clause.split()  # split on whitespace
        has_not = False  # True if the clause contains the word 'not'
        processed_words = get_processed_words(words)
        processed_review += processed_words
    return processed_review
```

Now the main function `process_reviews` will iterate over the dataframe, removing unwanted characters which has the effect of tokenization, and splitting the review into clauses via regular expressions and then making calls to `get_processed_review` to handle further processing at the sub-sentence level.

[28]:
```
def process_reviews(dataframe):
    processed_reviews = []
    for i in range(len(dataframe)):
        # replace non-letters and non-punctuation with a space
```

```
        review = re.sub('[^a-zA-Z\.,;:]', ' ', df['review'][i])
        # convert to lowercase
        review = review.lower()
        # split on punctuation
        clauses = re.split('[\.,;:]', review)
        processed_review = get_processed_review(clauses)
        # join the list using spaces and add the resulting string to the list
        # of processed reviews
        processed_reviews.append(' '.join(processed_review))
    return processed_reviews
```

Let's now run the pre-processing algorithm and see what the first review looks like afterward.

[29]:
```
corpus = process_reviews(df)
corpus[0]
```

[29]:
```
'film never made honestli must admit saw seriou doubt director NOT_great
NOT_actress though lot movi holland young woman took main part tv person
constant smile face NOT_much NOT_self NOT_critic actor play main part recent saw
bride flight although film better NOT_convinc start stori NOT_read NOT_novel
NOT_base NOT_upon script underlay film someth might done mind kid birthday parti
raini sunday afternoon NOT_someon NOT_age NOT_director NOT_like NOT_watch
NOT_good NOT_movi someth realli disturb overdub dialogu time spoken loud regard
go cameraman least tri make someth piti film edit lousi scene certainli credibl'
```

I will be using a TF-IDF matrix for the text representation which is considered to be an improvement on the bag of words representation, for it gives more semantic meaning or importance to more frequent words in the corpus that are not stopwords.

TF stands for term frequency and IDF is inverse document frequency. The column vectors of the TF-IDF matrix represent documents or reviews in this case, where as the row vectors represent the vocabulary. TF is the frequency of a word in the document, while IDF is calculated as the log of the number of documents divided by the number of documents that contain a particular word. Each element of the TF-IDF matrix is the product of the TF and the IDF.

Note that the model may be improved by using n-grams in the TF-IDF matrix. We could for example use bigrams by passing the tuple `(1, 2)` to the `ngram_range` argument of `TfidfVectorizer`. Due to memory contraints however I will only be using unigrams.

Furthermore, because this is a binary classification task, the algorithm can be made more efficient by remapping the set of labels `{'negative', 'positive'}` to the boolean set `{0, 1}`.

[30]:
```
from sklearn.feature_extraction.text import TfidfVectorizer

# create the TF-IDF matrix
vectorizer = TfidfVectorizer(ngram_range=(1, 1))
tfidf_matrix = vectorizer.fit_transform(corpus).toarray()

# remap the sentiment labels to boolean values
```

```
labels = df['sentiment'].apply(
    lambda x: 0 if x[0] == 'n' else 1
).to_numpy(dtype=np.uint8)
```

1.2.2 Baseline Performance

I am working with a symmetric dataset that is evenly split between positive and negative reviews. Although the test set may not necessarily be evenly split, it will be evenly split on average if shuffling the documents in the corpus appropriately. Thus, a baseline classifier that randomly predicts `positive` with 50% probability and `negative` with 50% probability will achieve an accuracy of 50% on average. The accuracy of the model must therefore exceed 50%.

1.2.3 Classification Approach

I will use the naive Bayes classification approach as it is known to work well for text classification and is quite fast both for training and prediction. Naive Bayes makes the assumption that all features - i.e. words or n-grams in our case - are conditionally independent of one another, although this assumption is bound to introduce a certain degree of bias since it is never true with languages. [4] Nevertheless, in practice naive Bayes tends to work quite well.

The algorithm is an optimization problem where we iterate over the set of labels (`{0, 1}` in our case) and perform the calculation below to determine which label (or *class*) maximizes the right-hand side of the equation.

$$\hat{y} = \underset{k \in \{1,\ldots,K\}}{\operatorname{argmax}} \; p(C_k) \prod_{i=1}^{n} p(x_i \mid C_k)$$

Note that Ck represents a particular class and xi represents a particular feature.

Oftentimes the algorithm can be improved by capping the feature counts at 1 for each document. This approach is known as multinomial naive Bayes. As mentioned in the scikit-learn documentation, although theoretically multinomial naive Bayes expects discrete feature counts (i.e. bag of words without TF-IDF), in practice it does also work with continuous feature values such as those in TF-IDF.

[31]:
```
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB

reviews_train, reviews_test, labels_train, labels_test = train_test_split(
    tfidf_matrix, labels, test_size=0.2, random_state=1
)

model = MultinomialNB().fit(reviews_train, labels_train)
labels_pred = model.predict(reviews_test)
```

1.3 Conclusions

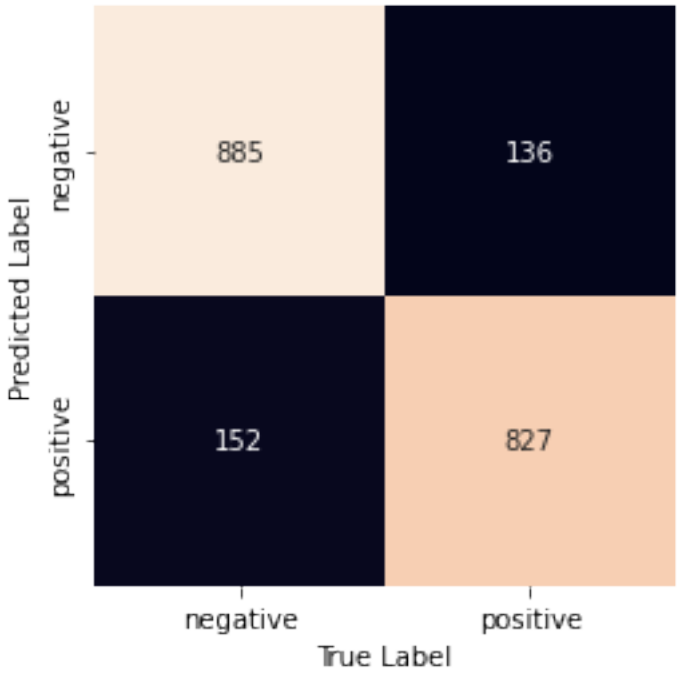
1.3.1 Evaluation

The classifier's performance may now be evaluated by comparing its predictions with the true labels in the test set. As discussed previously, calculating a confusion matrix will aid us in determining the accuracy.

[32]:
```
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix

# get confusion matrix
c_matrix = confusion_matrix(labels_test, labels_pred)

# plot confusion matrix
tick_labels = ['negative', 'positive']
sns.heatmap(
    c_matrix, square=True, annot=True, fmt='d', cbar=False,
    xticklabels=tick_labels, yticklabels=tick_labels
)
plt.xlabel('True Label')
plt.ylabel('Predicted Label')
print()
```



Here we see that the classifier predicted 885 true negatives and 827 true positives. 152 false negatives and 136 false positives were also predicted. To calculate the accuracy, we must divide the sum of the matrix's diagonal by the total sum of all elements in the matrix.

[33]:
```
# calculate accuracy
sum(np.diagonal(c_matrix)) / np.sum(c_matrix)
```

[33]: 0.856

The accuracy of the classifier is thus 85.6%, which is quite a significant improvement from our predictive baseline performance of 50% that simply guessed at random.

1.3.2 Summary and Conclusions

We have seen the effectiveness of using TF-IDF text representation in conjunction with the naive Bayes algorithm to achieve generalization power in natural language processing. Using these methods, a business or its competitors would be able to analyze web-scaped data on certain products and assess the customers' opinions of them. This model would probably also perform at least decently with any other text classification task, such as an email spam classifier.

When pre-processing text, tokenization via regular expressions, stemming, stopwords, and negation handling were all used. It might be possible that performing lemmatization along with stemming could reduce the complexity of the vocabulary further, so it may be worth exploring this idea.

The naive Bayes model could probably be improved with expanded features via n-grams. If you remember, we were unable to use n-grams due to memory constraints. It would also be worth comparing the naive Bayes model to methods such as logistic regression or support vector machine (SVM). Logistic regression often performs well with a large dataset and sparse text representation; while SVM - as pointed out here - is better at capturing the relationships between words as the algorithm does not make the same assumption of conditional independence between features that naive Bayes does make. By experimenting with n-grams and different algorithms, our result of 85.6% accuracy can likely be further improved.

These results should be reproducible using any programming environment; however, the ease with which they are reproducible will vary depending on the programming language and libraries. Python is a high-level interpreted language with a lot of processing overhead. If performance is critical, a low-level compiled language such as C or C++ could be more appropriate despite being more time-consuming to implement.

1.4 References

1. Kuncherichen, Sarath, Ebin, Neema. (2019). Sentiment Analysis in Product Reviews using Natural Language Processing and Machine Learning. [URL]
2. Wamambo, Tinashe & Luca, Cristina & Fatima, Arooj. (2019). Use Case Prediction Using Product Reviews Text Classification. 10.1007/978-3-030-33607-3_28. [URL]
3. Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. (2011). Learning Word Vectors for Sentiment Analysis. The 49th Annual Meeting of the Association for Computational Linguistics (ACL 2011). [URL]
4. Jurafsky, D. and J.H. Martin Speech and language processing. (2021; 3rd draft ed.). [URL]