Sentiment Analysis of Restaurant Reviews Using Naive Bayes Classification

Ye-eun Kim 22098692
dept. School of Physics, Engineering and Computer Science
University of Hertfordshire
Hatfield, Hertfordshire
july215215@gmail.com

I. Introduction

This project explores Natural Language Processing (NLP) by developing a Naive Bayes classifier from scratch for sentiment analysis of restaurant reviews. Sourced from Kaggle, the dataset contains 1,000 reviews, each labeled as positive or negative. (https://www.kaggle.com/datasets/nanuprasad/restaurant-reviews) The aim is to categorize these reviews, enhancing customer experience analysis. The approach involves comprehensive data preprocessing using NLTK for tokenization, stopwords removal, and stemming, followed by feature extraction through TF-IDF vectorization. This process, building upon the Bag of Words model, prepares the text for a custom-built Naive Bayes classifier that incorporates Laplace smoothing and log likelihood calculations to predict sentiments

II. Methodology

This project employs a practical approach to NLP by constructing a Naive Bayes classifier from the ground up, specifically for sentiment analysis of restaurant reviews. The methodology starts with data preprocessing, where the Bag of Words model is implicitly utilized as a foundation for feature extraction through TF-IDF vectorization.

III. Data Analysis

A meticulous examination of the dataset was conducted to ascertain the distribution of sentiments within the corpus. The dataset consists of a total of 1,000 entries, each entry being a customer review of a restaurant. These reviews are classified into two distinct categories of sentiment: positive and negative. The distribution of these sentiments is pivotal to the subsequent analytical procedures, as it informs the implementation strategy of the Naive Bayes classifier.

The dataset was found to be balanced with an equal number of positive and negative sentiments, comprising 500 instances of each category. This balance is crucial as it allows for the assumption that the prior probabilities of each class are equal, thus simplifying the computation of posterior probabilities. Figure 2 shows the output of the code utilized to verify the balance of the dataset, where variable ‘pos‘ represents the count of positive sentiments and ‘neg‘ the count of negative sentiments.

```
dataset.info()
```
✓ 0.0s
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Review  1000 non-null   object
 1   Liked   1000 non-null   int64
dtypes: int64(1), object(1)
memory usage: 15.8+ KB
```

Fig. 1. Dataset Information

```
pos = 0
neg = 0
for val in dataset['Liked']:
    if val == 0:
        neg+=1
    else:
        pos+=1

print(f'positive: {pos}, negative: {neg}')
```
✓ 0.0s
```
positive: 500, negative: 500
```

Fig. 2. Code Output of Counts of Positive and Negative Sentiments.

The integrity of the dataset’s balance is further corroborated by the DataFrame’s information output, which denotes a total of 1,000 non-null entries across two columns, as depicted in Figure 1. The ‘Liked‘ column, which contains binary sentiment indicators, possesses an equal split of 0 and 1 values, representing negative and positive sentiments, respectively. This uniform distribution is integral to the fair assessment of the classifier’s performance during the testing phase.

A. Data Preprocessing

The dataset, comprising 1,000 restaurant reviews, underwent a series of preprocessing steps designed to standardize the text and reduce noise that could potentially skew the analysis. The first step involved tokenization, where the text was split into individual words, or "tokens," using the `word_tokenize` function from the NLTK library, which is proficient in handling the nuances of natural language.

Subsequently, all tokens were converted to lowercase to maintain consistency, as text data is case-sensitive, and the presence of both uppercase and lowercase versions of the same word would be treated as distinct entities otherwise. The process continued with the removal of non-alphabetic characters to exclude any numbers and symbols that do not contribute to sentiment analysis.

A critical aspect of the preprocessing phase was the exclusion of stopwords, which are commonly used words in a language that carry minimal informative weight, such as "the," "is," and "in." A curated list of stopwords from the NLTK corpus was utilized, with the exception of negations like "not," "don't," and "isn't," as these can significantly alter the sentiment of a phrase. The retention of these negative words is essential in sentiment analysis, as they can invert the sentiment expressed in a sentence.

Following the removal of stopwords, the Porter Stemming algorithm, also from NLTK, was applied to reduce words to their root form, or "stem," thereby consolidating different grammatical variations of a word into a single representation.

```
# Prepare stop words from NLTK library, excluding negative words
# as they may carry sentiment meaning
stop_words = set(stopwords.words("english"))
negative_words = {'no', 'not', "don't", "doesn't", "didn't", "isn't",
                  "aren't", "won't"}
stop_words = stop_words - negative_words

# Data preprocessing: Tokenize, remove stop words, stem, and rejoin
# to form the final corpus
corpus = []
for i in range(0, len(dataset)):
    tokens = word_tokenize(dataset['Review'][i].lower())
    ps = PorterStemmer()
    review = [ps.stem(word) for word in tokens if word
              not in stop_words and word.isalpha()]
    corpus.append(' '.join(review))
```

Fig. 3. Data Preprocessing

B. Feature Extraction

The preprocessed data undergoes feature extraction through TF-IDF vectorization, a process that initially aligns with the Bag of Words (BoW) model. This method, executed by the `TfidfVectorizer`, begins by counting the occurrence of each word within documents, akin to the BoW approach. It then extends this model by applying the Term Frequency-Inverse Document Frequency (TF-IDF) measure, which quantifies text by emphasizing words that uniquely characterize each document. This dual approach captures both the frequency of words (BoW) and their relative importance (TF-IDF), thus creating a more nuanced feature set for subsequent sentiment analysis.

```
# Feature extraction using TF-IDF
tfidf = TfidfVectorizer(max_features = 900)
X = tfidf.fit_transform(corpus).toarray()
y = dataset.iloc[:, -1].values
```

Fig. 4. TF-IDF Feature Extraction

C. Custom Naive Bayes Classifier

The Custom Naive Bayes Classifier constructed for this investigation operationalizes the Naive Bayes algorithm's core tenets. Commencing with the computation of conditional probabilities for each word within a given class, the classifier employs the Laplace smoothing formula as presented in Equation (1):

$$P(\text{word}|\text{class}) = \frac{\text{frequency of the word in the class} + 1}{\text{class size} + \text{size of the vocabulary}} \tag{1}$$

Laplace smoothing adeptly rectifies the zero-frequency predicament by adjusting the frequency counts. During the model's fitting routine, these probabilities are logarithmically scaled to alleviate the computational intricacies associated with the high-dimensional textual data.

```
class NaiveBayesClassifier:
    def __init__(self, laplace=1):
        self.laplace = laplace

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self._classes = np.unique(y)
        n_classes = len(self._classes)

        # Initialize probabilities
        self._log_prior = np.zeros(n_classes)
        self._word_count = np.zeros((n_classes, n_features))
        self._word_sum = np.zeros(n_classes)

        for c in self._classes:
            X_c = X[y == c]
            self._word_count[c, :] = X_c.sum(axis=0)
            self._word_sum[c] = X_c.sum()
            self._log_prior[c] = np.log(X_c.shape[0] / n_samples)

        self._word_prob = np.log(self._word_count + self.laplace)
        - np.log(self._word_sum[:, np.newaxis]
                 + n_features * self.laplace)

    def predict(self, X):
        return [self._predict(x) for x in X]

    def _predict(self, x):
        posteriors = []

        for idx, c in enumerate(self._classes):
            posterior = self._log_prior[idx]
            posterior += (self._word_prob[idx, :] * x).sum()
            posteriors.append(posterior)

        return self._classes[np.argmax(posteriors)]
```

Fig. 5. Custom Naive Bayes Classifier

The classifier's next stride involves the aggregation of these log probabilities across the vocabulary for each class, forming the foundation for inference. To determine the overall likelihood of a document belonging to a particular class, Equation (2) is used, which aggregates the log probabilities of all the words in the document, adjusted by the prior probabilities of the classes:

$$\log \frac{P(\text{pos}|\text{data})}{P(\text{neg}|\text{data})} = \log \frac{P(\text{pos})}{P(\text{neg})} + \sum_{i=1}^{n} \log \frac{P(w_i|\text{pos})}{P(w_i|\text{neg})} \quad (2)$$

The classification decision hinges on the class with the superior posterior probability. Notwithstanding the dataset's balanced constitution, the train-test split's inherent randomness could skew the class distribution. This contingency necessitates the use of the full Bayesian equation over its simplified counterpart. Such meticulous adherence to the comprehensive formula bolsters the classifier's robustness, circumventing potential bias from disproportionate class representation post-split. The culmination of these methodological steps ensures enhanced precision and dependability in the sentiment predictions derived from textual data.

D. Model Training and Prediction

The dataset was initially partitioned into training and testing subsets, with 80% allocated for training and the remaining 20% designated for evaluation. To ensure the reproducibility of results and maintain consistency across different runs, a fixed seed value was used during the train-test splitting process. This practice guarantees that the same data partitioning is achieved each time the code is executed, allowing for a fair comparison of model performance over iterative refinements. Post-training, the model's performance was meticulously gauged using the test set. Comparisons of predictions against actual sentiments were conducted, leading to the construction of a confusion matrix. This matrix was instrumental in deriving various performance metrics, such as accuracy, precision, recall, and the F1 score. These metrics collectively offer insights into the classifier's proficiency in sentiment analysis.

```
# fix a seed to make the constant result
SEED = 215

# Split dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    random_state=SEED)

# Training using Multinomial Naive Bayes classifier,
# which includes log-likelihood and Laplace smoothing by default
classifier = NaiveBayesClassifier()
classifier.fit(X_train, y_train)

# Prediction on the test dataset
y_test_pred = classifier.predict(X_test)
```

Fig. 6. Model Training and Prediction

IV. Results and Discussion

The efficacy of the custom Naive Bayes classifier was appraised through a variety of performance metrics extrapolated from the confusion matrix. Following the adjustment of the maximum features parameter to tailor the model's complexity, the classifier evidenced an accuracy of 79%, precision of 76%, recall of 82%, and an F1 score of 79%. The adjusted confusion matrix is elucidated below:

$$\begin{bmatrix} 76 & 25 \\ 18 & 81 \end{bmatrix}$$

The matrix reflects the classifier's proficient capability in discerning both positive and negative sentiments. The accuracy rate denotes that approximately 79% of the total predictions were correct. Precision, at 76%, intimates that when the classifier prognosticates a review as positive, it is reliable three-quarters of the time. Conversely, the recall rate implies that of all the actual positive reviews, the classifier successfully detects 82% of them. The F1 score, a weighted average of precision and recall, intimates a harmonized balance between the two metrics, corroborating the classifier's efficacy in maintaining a calibrated performance amidst varied sentiment expressions. This harmonization is pivotal in scenarios where both the minimization of false positives and the maximization of true positives are of paramount importance.

V. Conclusion

In conclusion, the custom Naive Bayes classifier developed for this project demonstrates considerable efficacy in the sentiment analysis of restaurant reviews. Despite the challenges inherent in text classification, the classifier exhibits a commendable balance of precision and recall, as reflected in the F1 score. The insights garnered from the performance metrics underscore the potential of such tailored classifiers in NLP applications. While there is room for enhancement, particularly in reducing false positives, the foundation established by this study provides a robust basis for future explorations. Further research could explore more sophisticated natural language processing techniques and the integration of additional contextual factors to refine the classifier's performance.