Findings

May 5, 2024

1 Report

2 Chosen representation & data preprocessing

For data preprocessing, I implemented a simple text processing function to clean the abstracts extracted from research papers. This function removes non-hyphenated numbers and unnecessary spacing from the text. The motivation behind this preprocessing step is to ensure that the text data is uniform and free from irrelevant characters that could potentially introduce noise into the model. By removing non-hyphenated numbers and extra spacing, we focus on preserving the relevant contents of the abstracts.

For data representation, I chose to use occurrence frequency of words in a vector format. In this representation, each abstract is transformed into a vector where each element corresponds to the frequency count of a specific word. The reason to choosing frequency count vectorization rather than other forms such as binary count vectorization is because frequency captures the importance of each word based on its frequency of occurrence in the abstract. This is good for this dataset as certain technical terms or domain-specific vocabulary may appear frequently, by considering the frequency of words the model can give higher importance to words that occur more frequently, allowing it to capture the patterns and relationships in the data more effectively.

3 Model improvements and their implementation

The implementation of Naive Bayes is a good idea in this case because it is a good choice for documentation classificastion, I trained the model by calculating the prior probability, then the conditional probability and used them to predict the class label for the test and validation. During the conditional probility calculation, laplace smoothing was used so that if there was missing attribute in the data it is just ommited and ignored. During the prediction, the calculation was performed in log space because otherwise tiny numbers could cause potential computation issues.

I have found out from observing the dataset that the class is actually imbalanced, I have printed out what the amount classfications and we can see that initially 'E': 2144, 'B': 1602, 'A': 128, 'V': 126, this could lead to biases in the trianing process this is why I have implemented the oversampling technique using the module 'from imblearn.over_sampling import RandomOverSampler' to oversample the minority class so that the training set becomes more evenly distrubuted.

There are words in the dataset that often have sequential relationships for example: homo is often followed by sapiens, escherichia coli, human immunodeficiency virus and etc. This is the motivation for using n-grams because it can capture these relationships and allows the model to consider not only unigram words but also bigrams, trigrams and such. I have found out during my implementation of n-grams that when I didnt use the oversampling method the n-grams function made a worse prediction and that it made a better predicton with oversampling. This is probabiliy because when combining words the distrubution becomes more imbalanced. However, oversampling solves this by balancing out the dataset and therefore reducing the bias which is why n-grams got better results working with oversampling.

I used train and validation data to test out what hypermeter for n-grams is the best by looking at the accuracies, when I set the range to include only unigram (ngram_range=(1, 1)), both train and validation was the lowest with accuracies of 99.48% and 98.25%, when the range is unigram and bigram (ngram_range=(1, 2)) the train and validation accuracy increased to 99.90% and 98.72%. It should be also noted that the as n-gram value become higher the potential of overfitting occuring becomes higher due to the fact that it captures noise in the training process and not generalizing well enough for new and unseen data so for hyper perameter tuning I have decided that ngram_range=(1, 2) rather than going even higher comprimising between accuracy and overfitting.

4 Evaluation procedure

First, test preprocessing was implemented to remove unnecessary instances and ensure the format, then oversampling was used in the improved model to evenly distribute the data. I have choosen to use the train/validation split for my dataset using the train_test_split fucntion from scikit-learn so I can use 90% of the training dataset to train my naive bayes model while using 10% as validation data to validate the performance as well as determing the best functions to implement for accuracy improvement. Then I converted my data to vector form to then pass into my naive bayes classifer. I trained the naice bayes classifier using the training data and computed the accuracy scores for both the training and validation sets using the accuracy_score function from scikit-learn. These accuracy scores are metrics for evaluating the performance of the model on both the training and validation data, so that I can asses the general performance and areas to improve on.

5 Training/validation results for the standard and improved Naive Bayes model.

For the standard naive bayes model the accuracy I got for train is 98.06% and validation of 95%. For the standard model, the different between the trian and validation accuracies are quite high, this tells us that there is overfitting in play. I suspect that because of the uneven distrubution of data (‘E’: 2144, ‘B’: 1602, ‘A’: 128, ‘V’: 126) the model has overemphasized patterns from the majority class ‘E’ and ‘B’, and because of the limited dataset for ‘A’ and ‘V’. So as a result, the models performance on unseen data is poor.

For the improved naice bayes model I got train of 99.90% and validation of 98.72%. While in the improved model we can see that not only did accuracy imporved overall, the difference between train and validation also reduced with the introduction of oversampling to handle imbalanced data and n-grams allowing the model to capture word relations.

Finally, using the improved model in kaggle gives me a accuracy of 97% which is not that much different from my validation. The improvements to the model have resulted in better predictions on unseen data in comparison to the standard model.

[10]:
```
# model without any improvements
import pandas as pd
import re
import numpy as np
import math
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics import accuracy_score
from collections import Counter

df_train = pd.read_csv('trg.csv')
df_test = pd.read_csv('tst.csv')

df_train = df_train.rename(columns={'class': 'class_labels'})
train_labels = df_train['class_labels']

#text preprocessing
def text_processor(text):
    text = re.sub(r'\b(?<!-)\d+\b(?!-)', '', text)
    text = re.sub(r'\s+', ' ', text).strip()
    return text

train_inputs = df_train['abstract'].apply(text_processor)

test_inputs = df_test['abstract'].apply(text_processor)

#spliting the data into train and validation set
X_train, X_val, Y_train, Y_val = train_test_split(train_inputs, train_labels,␣
 ↪test_size=0.1, random_state=42)



vectorizer = CountVectorizer()

X_train = vectorizer.fit_transform(X_train)

X_val = vectorizer.transform(X_val)

X_test = vectorizer.transform(test_inputs)

counts = Counter(train_labels)
print(counts)

class NaiveBayes:

    #calculates the prior_probability
    def prior_probability(self,X,Y):
        rows,features = X.shape
        unique_labels = list(Y.unique())
        prior = []

        for labels in unique_labels:
            X_labels = X[Y==labels]
            prior.append(X_labels.shape[0]/rows)
        return prior

    #calculates the conditional_probability
    def conditional_probability(self,X,Y):
        rows,features = X.shape
        unique_labels = list(Y.unique())
        conditional = []

        for labels in unique_labels:
            X_labels = X[Y==labels]
            frequency_sum = X_labels.sum(axis=0)
            words = X_labels.sum()
            conditional_prob = []

            for i in range(features):
                count = frequency_sum[0,i]
                #laplace smoothing
                prob = (count+1)/(words+features)
                conditional_prob.append(prob)

            conditional.append(conditional_prob)

        return conditional

    #makes a predction
    def predict(self,X,Y,test):
        prior = self.prior_probability(X,Y)
        conditional = self.conditional_probability(X,Y)
        unique_labels = list(Y.unique())
        predictions = []

        for i in range(test.shape[0]):
            test_prob = []

            for j in range(len(prior)):
                prediction_prob = math.log(prior[j])

                for k in range(test.indptr[i], test.indptr[i+1]):
                    row = j
                    column = test.indices[k]
                    freq = test.data[k]
                    prediction_prob+=math.log(math.
↪pow(conditional[row][column],freq))

                test_prob.append(prediction_prob)

            prediction = unique_labels[test_prob.index(max(test_prob))]
            predictions.append(prediction)

        return predictions

bayes = NaiveBayes()
```

```
Counter({'E': 2144, 'B': 1602, 'A': 128, 'V': 126})
```

[11]:
```
predictions = bayes.predict(X_train,Y_train,X_train)

#computes the accuracy
accuracy = accuracy_score(Y_train,predictions)
print(accuracy)
```

```
0.9805555555555555
```

[12]:
```
predictions = bayes.predict(X_train,Y_train,X_val)

#computes the accuracy
accuracy = accuracy_score(Y_val,predictions)
print(accuracy)
```

```
0.95
```

[13]:
```
predictions = bayes.predict(X_train,Y_train,X_test)

counts = Counter(predictions)
print(counts)
```

```
Counter({'E': 573, 'B': 386, 'A': 30, 'V': 11})
```

[18]:
```
#improved model
import pandas as pd
import re
import numpy as np
import math
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics import accuracy_score
from imblearn.over_sampling import RandomOverSampler
from collections import Counter

df_train = pd.read_csv('trg.csv')
```

```python
df_test = pd.read_csv('tst.csv')

df_train = df_train.rename(columns={'class': 'class_labels'})
train_labels = df_train['class_labels']

#text preprocessing
def text_processor(text):
    text = re.sub(r'\b(?<!-)\d+\b(?!-)', '', text)
    text = re.sub(r'\s+', ' ', text).strip()
    return text

train_inputs = df_train['abstract'].apply(text_processor)

test_inputs = df_test['abstract'].apply(text_processor)

# Oversampling the data
ros = RandomOverSampler(random_state=42)
train_inputs, train_labels = ros.fit_resample(np.array(train_inputs).
 ↪reshape(-1, 1), train_labels)

# Flatten resampled data
train_inputs = train_inputs.flatten()

#spliting the data into train and validation set
X_train, X_val, Y_train, Y_val = train_test_split(train_inputs, train_labels,␣
 ↪test_size=0.1, random_state=42)


vectorizer = CountVectorizer(ngram_range=(1, 2))

X_train = vectorizer.fit_transform(X_train)

X_val = vectorizer.transform(X_val)

X_test = vectorizer.transform(test_inputs)

counts = Counter(train_labels)
print(counts)

class NaiveBayes:

    #calculates the prior_probability
    def prior_probability(self,X,Y):
        rows,features = X.shape
        unique_labels = list(Y.unique())
        prior = []
        for labels in unique_labels:
            X_labels = X[Y==labels]
            prior.append(X_labels.shape[0]/rows)
        return prior

    #calculates the conditional_probability
    def conditional_probability(self,X,Y):
        rows,features = X.shape
        unique_labels = list(Y.unique())
        conditional = []

        for labels in unique_labels:
            X_labels = X[Y==labels]
            frequency_sum = X_labels.sum(axis=0)
            words = X_labels.sum()
            conditional_prob = []

            for i in range(features):
                count = frequency_sum[0,i]
                #laplace smoothing
                prob = (count+1)/(words+features)
                conditional_prob.append(prob)

            conditional.append(conditional_prob)

        return conditional

    #makes a predction
    def predict(self,X,Y,test):
        prior = self.prior_probability(X,Y)
        conditional = self.conditional_probability(X,Y)
        unique_labels = list(Y.unique())
        predictions = []

        for i in range(test.shape[0]):
            test_prob = []

            for j in range(len(prior)):
                prediction_prob = math.log(prior[j])

                for k in range(test.indptr[i], test.indptr[i+1]):
                    row = j
                    column = test.indices[k]
                    freq = test.data[k]
                    prediction_prob+=math.log(math.
↪pow(conditional[row][column],freq))
```

```
                test_prob.append(prediction_prob)

            prediction = unique_labels[test_prob.index(max(test_prob))]
            predictions.append(prediction)

        return predictions

bayes = NaiveBayes()
```

```
Counter({'B': 2144, 'A': 2144, 'E': 2144, 'V': 2144})
```

[19]:
```
predictions = bayes.predict(X_train,Y_train,X_train)

#computes the accuracy
accuracy = accuracy_score(Y_train,predictions)
print(accuracy)
```

```
0.9989634620367971
```

[20]:
```
predictions = bayes.predict(X_train,Y_train,X_val)

#computes the accuracy
accuracy = accuracy_score(Y_val,predictions)
print(accuracy)
```

```
0.9871794871794872
```

[17]:
```
predictions = bayes.predict(X_train,Y_train,X_test)
counts = Counter(predictions)
print(counts)
df_predictions = pd.DataFrame({'id': df_test['id'], 'class': predictions})
df_predictions.to_csv('predictions.csv', index=False)
```

```
Counter({'E': 563, 'B': 374, 'V': 33, 'A': 30})
```