HW1-CSCI544

September 8, 2022

0.1 Brief overview of approach for NLP HW1

- From the original dataset we keep only the 'star_rating' and 'review_body' columns to train/test multiple classifiers
- We convert 'star_rating' to integer and 'review_body' to string for uniformity of datatypes across these columns
- We sample 100,000 random reviews from each 'star_rating' class since it was discovered that the mininmum number of reviews for a class is around 100,000. This sampling was performed to fit the TFIDFVectorizer to a larger corpus so that it can understand the data and provide better features when 20K reviews are randomly sampled
- Following data cleaning techniques have been performed on the dataset -
  1. convert all reviews to lowercase
  2. remove html tags and urls from reviews using BeautifulSoup
  3. remove extra spaces in the reviews
  4. remove punctuations
  5. remove non-alphabetical characters
  6. perform contractions on the reviews
- Following data preprocessing techniques have been performed on the dataset -
  1. remove stop words
  2. perform lemmatization
- Once we have a fitted TFIDFVectorizer - we further sample 20K reviews for each class and transform it using the fitted TFIDFVectorizer
- Next, we create our training and testing split with an 80-20 ratio and pass it to classifiers like the Perceptron, SVM, Logistic Regression and Multinomial Naive Bayes
- Finally, we evaluate our trained models on the testing splits and present the results

Notes - It was observed that the removal of stop words from the reviews did not help the performance of the models but actually degraded it - However lemmatization helped with the performance of the models - Therefore we perform two types of preprocessing 1. Reviews with stop words removed and lemmatization performed 2. Reviews with no stop word removal but lemmatization is performed (This data is used for training and testing the model as it yielded better performances)

0.2 Imports

[169]:
```
# ! pip install bs4
# ! pip install lxml
# ! pip install contractions
```

[1]:
```python
import pandas as pd
import numpy as np
import nltk
import re
from bs4 import BeautifulSoup
nltk.download('wordnet', quiet=True)
nltk.download('stopwords', quiet=True)
nltk.download('punkt', quiet=True)
nltk.download('omw-1.4', quiet=True)
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
import string
import contractions
from sklearn.metrics import classification_report
import warnings
```

[171]:
```python
# Dataset: https://s3.amazonaws.com/amazon-reviews-pds/tsv/
↪amazon_reviews_us_Jewelry_v1_00.tsv.gz
```

0.3 Read Data

[172]:
```python
filepath = './amazon_reviews_us_Jewelry_v1_00.tsv'
reviews_df = pd.read_csv(filepath, sep='\t', on_bad_lines='skip',␣
↪dtype='unicode')
```

[173]:
```python
reviews_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1766992 entries, 0 to 1766991
Data columns (total 15 columns):
 #   Column             Dtype
---  ------             -----
 0   marketplace        object
 1   customer_id        object
 2   review_id          object
 3   product_id         object
 4   product_parent     object
 5   product_title      object
 6   product_category   object
 7   star_rating        object
 8   helpful_votes      object
 9   total_votes        object
 10  vine               object
 11  verified_purchase  object
 12  review_headline    object
 13  review_body        object
 14  review_date        object
```

```
dtypes: object(15)
memory usage: 202.2+ MB
```

0.4 Keep Reviews and Ratings

- Selecting only 'star_rating' and 'review_body'
- We use 'review_body' to develop the input features
- We use 'star_rating' as the target results which must be predicted

[174]:
```
reviews_df = reviews_df[['star_rating','review_body']]
```

[175]:
```
reviews_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1766992 entries, 0 to 1766991
Data columns (total 2 columns):
 #   Column       Dtype
---  ------       -----
 0   star_rating  object
 1   review_body  object
dtypes: object(2)
memory usage: 27.0+ MB
```

0.5 Randomly selecting reviews from each star_rating_class

[176]:
```
# Converting all 'star_rating' to integer representations
# Select all rows which have 'star_rating' and 'review_body' as existing int/
↪string values for optimal training results of the models

reviews_df['star_rating'] = pd.
↪to_numeric(reviews_df['star_rating'],errors='coerce')
reviews_df = reviews_df[reviews_df['star_rating'].notna()]
reviews_df = reviews_df[reviews_df['review_body'].notna()]

#Convert all 'star_rating' to int
reviews_df['star_rating'] = reviews_df['star_rating'].astype(int)

#Convert all reviews to string
reviews_df['review_body'] = reviews_df['review_body'].astype(str)
```

[177]:
```
reviews_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1766748 entries, 0 to 1766991
Data columns (total 2 columns):
 #   Column       Dtype
---  ------       -----
 0   star_rating  int64
 1   review_body  object
```

```
dtypes: int64(1), object(1)
memory usage: 40.4+ MB
```

0.6 Sampling 100,000 samples per class

- It was discovered that the minimum number of samples in a class (Rating - 2) was around 100K so I decided to select 100K samples per class to train the TFIDFVectorizer.
- We select 100K per class so that there is no imbalance in representation of words/reviews/classes which may affect the TFIDFVectorizer since it takes into consideration term and document frequencies which could be influenced by major class imbalances
- Once we have trained the TFIDFVectorizer on the 500K samples (100K samples for 5 rating classes)- we sample 20000 reviews randomly from each rating class to create the training and testing dataset.

[178]:
```
reviews_df['star_rating'].value_counts()
```

[178]:
```
5    1080871
4     270424
3     159654
1     155002
2     100797
Name: star_rating, dtype: int64
```

[179]:
```
rating_1 = reviews_df[reviews_df.star_rating.eq(1)].sample(100000,␣
 ↪random_state=1)
rating_2 = reviews_df[reviews_df.star_rating.eq(2)].sample(100000,␣
 ↪random_state=1)
rating_3 = reviews_df[reviews_df.star_rating.eq(3)].sample(100000,␣
 ↪random_state=1)
rating_4 = reviews_df[reviews_df.star_rating.eq(4)].sample(100000,␣
 ↪random_state=1)
rating_5 = reviews_df[reviews_df.star_rating.eq(5)].sample(100000,␣
 ↪random_state=1)

sampled_reviews_df_100000 = pd.concat([rating_1, rating_2, rating_3, rating_4,␣
 ↪rating_5])
```

[180]:
```
sampled_reviews_df_100000.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 500000 entries, 344647 to 1044904
Data columns (total 2 columns):
 #   Column       Non-Null Count   Dtype
---  ------       --------------   -----
 0   star_rating  500000 non-null  int64
 1   review_body  500000 non-null  object
dtypes: int64(1), object(1)
memory usage: 11.4+ MB
```

0.7 Utility functions

[181]:
```
def calculateAverageLength(df, columnName):
    """Calculates the average length of the given column in the dataframe␣
↪provided as an argument

    Args:
        df (DataFrame): Dataframe in which we must locate the column and␣
↪calculate it's average length
        columnName (string): Name of the column for which we calculate the␣
↪average length
    """
    total_length = 0
    for i in df[columnName].tolist():
        total_length += len(i)
    mean_length = total_length/len(df[columnName].tolist())
    return mean_length
```

[182]:
```
def URLRemoval(sentence):
    """Function to remove the HTML tags and URLs from reviews using␣
↪BeautifulSoup

    Args:
        sentence (string): Sentence from which we remove the HTML tags and URLs

    Returns:
        string: Sentence which does not contain any HTML tags and URLs
    """
    return BeautifulSoup(sentence, 'lxml').get_text()
```

[183]:
```
def nonAlphabeticRemoval(sentence):
    """Function to remove non-alphabetic characters from the sentence.
    Note - We do not remove spaces from the sentence however extra spaces are␣
↪removed in a different function

    Args:
        sentence (string): Sentence from which we remove the non-alphabetic␣
↪characters

    Returns:
        string: Sentence from which non-alphabetic characters have been removed
    """
    return re.sub(r"[^a-zA-Z ]+", "", sentence)  #This will also remove numbers.
```

[184]:
```
def removeExtraSpaces(sentence):
    """Remove extra spaces from the sentence
```

```
    Args:
        sentence (string): Sentence from which we remove extra spaces

    Returns:
        string: Sentence from which extra spaces have been removed
    """
    return ' '.join(sentence.split())
```

[185]:
```
def stopWordRemoval(sentence):
    """Function to remove stop words from the sentence

    Args:
        sentence (string): Sentence from which stop words have to be removed

    Returns:
        string: Sentence from which stop words have been removed
    """
    word_tokens = word_tokenize(sentence)
    filtered_sentence = []
    stop_words = {}
    for stop_word in stopwords.words('english'):
        if stop_words.get(stop_word) == None:
            stop_words[stop_word] = 1
    for w in word_tokens:
        if stop_words.get(w) == None:
            filtered_sentence.append(w)
    return ' '.join(filtered_sentence)
```

[186]:
```
def lemmatizeSentence(sentence):
    """Function to lemmatize a sentence

    Args:
        sentence (string): Sentence which has to be lemmatized

    Returns:
        string: Sentence which has been lemmatized
    """
    word_tokens = word_tokenize(sentence)
    lemmatized_sentence = []
    lemmatizer = WordNetLemmatizer()
    for word in word_tokens:
            lemmatized_sentence.append(lemmatizer.lemmatize(word, pos='a')) #We␣
 ↪utilize POS tag 'a' - adjective
    return ' '.join(lemmatized_sentence)
```

[187]:
```
def removePunctuation(sentence):
    """Function to remove punctuations from a sentence
```

```
    Args:
        sentence (string): Sentence from which punctuations have to be removed

    Returns:
        string: Sentence from which punctuations have been removed
    """
    for value in string.punctuation:
        if value in sentence:
            sentence = sentence.replace(value, ' ')
    return sentence.strip()
```

[188]:
```
def displayReport(actualLabels, predictedLabels, classifierName):
    """Function to display precision/recall/f1-score metrics for a classifier

    Args:
        actualLabels (_type_): True labels of the data
        predictedLabels (_type_): Labels predicted by the classifier
        classifierName (_type_): Name of the classifier which predicted the␣
↪labels
    """
    targetNames = ['Rating 1', 'Rating 2', 'Rating 3', 'Rating 4', 'Rating 5']
    report = classification_report(actualLabels, predictedLabels,␣
↪target_names=targetNames, output_dict=True)
    print(f'Precision, Recall, f1-score for Testing split for {classifierName}')
    print('============================================================')
    for targetClass in targetNames:
        print(f'{targetClass}: {report[targetClass]["precision"]},␣
↪{report[targetClass]["recall"]}, {report[targetClass]["f1-score"]}')
    print(f'Macro Average: {report["macro avg"]["precision"]}, {report["macro␣
↪avg"]["recall"]}, {report["macro avg"]["f1-score"]}')
    print('============================================================')
```

1 Data Cleaning

[189]:
```
lengthBeforeCleaning = calculateAverageLength(sampled_reviews_df_100000,␣
↪'review_body')
```

1.1 Convert all reviews into the lower case

[190]:
```
sampled_reviews_df_100000['review_body'] =␣
↪sampled_reviews_df_100000['review_body'].apply(lambda value:value.lower())
```

1.2 Remove the HTML and URLs from the reviews

[191]:
```
sampled_reviews_df_100000['review_body'] =␣
↪sampled_reviews_df_100000['review_body'].apply(URLRemoval)
```

1.3 Remove extra spaces

[192]:
```
sampled_reviews_df_100000['review_body'] =␣
↪sampled_reviews_df_100000['review_body'].apply(removeExtraSpaces)
```

1.4 Remove punctuations

[193]:
```
sampled_reviews_df_100000['review_body'] =␣
↪sampled_reviews_df_100000['review_body'].apply(removePunctuation)
```

1.5 Remove non-alphabetical characters (excluding spaces)

[194]:
```
sampled_reviews_df_100000['review_body'] =␣
↪sampled_reviews_df_100000['review_body'].apply(nonAlphabeticRemoval)
```

1.6 Perform contractions on the reviews

[195]:
```
sampled_reviews_df_100000['review_body_contracted'] =␣
↪sampled_reviews_df_100000['review_body'].apply(lambda value:[contractions.
↪fix(word) for word in value.split()])
sampled_reviews_df_100000['review_body_contracted'] =␣
↪sampled_reviews_df_100000['review_body_contracted'].apply(' '.join)
```

1.7 Average length of the reviews before & after data cleaning

[196]:
```
lengthAfterCleaning = calculateAverageLength(sampled_reviews_df_100000,␣
↪'review_body_contracted')
```

[197]:
```
print(f'Average length of reviews before and after data cleaning:␣
↪{lengthBeforeCleaning}, {lengthAfterCleaning}')
```

```
Average length of reviews before and after data cleaning: 189.71903, 182.180316
```

2 Pre-processing

[198]:
```
lengthBeforePreprocessing = calculateAverageLength(sampled_reviews_df_100000,␣
↪'review_body_contracted')
```

2.1 Remove the stop words

[199]:
```
sampled_reviews_df_100000['review_body_without_stop_words'] =␣
↪sampled_reviews_df_100000['review_body_contracted'].apply(lambda value:
↪stopWordRemoval(value))
```

2.2 Perform lemmatization

[200]:
```
# Perform lemmatization on sentences which had stop words removed from them
sampled_reviews_df_100000['review_body_with_stop_after_lemma'] =␣
↪sampled_reviews_df_100000['review_body_without_stop_words'].apply(lambda␣
↪value:lemmatizeSentence(value))
```

[201]:
```
# Perform lemmatization on sentences which did not have stop words removed from␣
↪them
sampled_reviews_df_100000['review_body_without_stop_after_lemma'] =␣
↪sampled_reviews_df_100000['review_body_contracted'].apply(lambda value:
↪lemmatizeSentence(value))
```

2.3 Average length of the reviews after preprocessing

[202]:
```
lengthAfterPreprocessing = calculateAverageLength(sampled_reviews_df_100000,␣
↪'review_body_with_stop_after_lemma')
```

[203]:
```
print(f'Average length of reviews before and after data preprocessing:␣
↪{lengthBeforePreprocessing}, {lengthAfterPreprocessing}')
```

```
Average length of reviews before and after data preprocessing: 182.180316,
108.783628
```

3 TF-IDF Feature Extraction

3.0.1 Feature Extraction using TfidfVectorizer

- Through the process of experimentation with multiple feature preprocessing and data cleaning techinques it was observed that the best results for all classifiers were obtained when a TFIDFVectorizer is fitted on the reviews data which did not have any stop word removal but lemmatization of the reviews had been performed.
- Observed a precision/recall/f1-score boost of 1-2% when utilizing this data over reviews which had stop words removed and the content was lemmatized.

[204]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
tfidfVector = TfidfVectorizer(use_idf=True, min_df=1, ngram_range=(1,3))
fittedtfidfVector = tfidfVector.
↪fit(sampled_reviews_df_100000['review_body_without_stop_after_lemma'].
↪to_list())
```

We select 20000 reviews randomly from each rating class and create the train-test split

[205]:
```
rating_1 = sampled_reviews_df_100000[sampled_reviews_df_100000.star_rating.
 ↪eq(1)].sample(20000, random_state=1)
rating_2 = sampled_reviews_df_100000[sampled_reviews_df_100000.star_rating.
 ↪eq(2)].sample(20000, random_state=1)
rating_3 = sampled_reviews_df_100000[sampled_reviews_df_100000.star_rating.
 ↪eq(3)].sample(20000, random_state=1)
rating_4 = sampled_reviews_df_100000[sampled_reviews_df_100000.star_rating.
 ↪eq(4)].sample(20000, random_state=1)
rating_5 = sampled_reviews_df_100000[sampled_reviews_df_100000.star_rating.
 ↪eq(5)].sample(20000, random_state=1)
sampled_reviews_df_20000 = pd.concat([rating_1, rating_2, rating_3, rating_4,␣
 ↪rating_5])
```

80-20 train-test split is created in the next section with stratification being performed for equal representation of classes in the training and test sets.

[206]:
```
from sklearn.model_selection import train_test_split
trainData, testData, trainLabels, testLabels =␣
 ↪train_test_split(sampled_reviews_df_20000['review_body_without_stop_after_lemma'].
 ↪to_list(), sampled_reviews_df_20000['star_rating'].to_list(), test_size=0.2,␣
 ↪random_state=42, stratify=sampled_reviews_df_20000['star_rating'].to_list())
```

[207]:
```
print(len(trainData), len(testData))
```

```
80000 20000
```

3.0.2 Transform the training and testing data using the fitted TFIDFVectorizer

[208]:
```
trainData = fittedtfidfVector.transform(trainData)
testData = fittedtfidfVector.transform(testData)
```

4 Perceptron

- After experimentation with different hyperparameter settings - best results are achieved with the configuration below.

[209]:
```
from sklearn.linear_model import Perceptron
perceptronModel = Perceptron(eta0=0.1, tol=1e-5, n_jobs=-1, max_iter=5000)
```

[210]:
```
perceptronModel.fit(trainData, trainLabels)
```

[210]:
```
Perceptron(eta0=0.1, max_iter=5000, n_jobs=-1, tol=1e-05)
```

[211]:
```
predictedLabels = perceptronModel.predict(testData)
displayReport(testLabels, predictedLabels, 'Perceptron')
```

```
Precision, Recall, f1-score for Testing split for Perceptron
============================================================
```

```
Rating 1: 0.5565935259613296, 0.6405, 0.595606183889341
Rating 2: 0.38808290155440417, 0.3745, 0.3811704834605598
Rating 3: 0.41912932952017795, 0.32975, 0.369105918567231
Rating 4: 0.45373665480427045, 0.44625, 0.44996218805142424
Rating 5: 0.6326301615798923, 0.70475, 0.6667455061494797
Macro Average: 0.4900345146840149, 0.49915000000000004, 0.4925180560236071
============================================================
```

5 SVM

- After experimentation with different hyperparameter settings - best results are achieved with the configuration below.

[212]:
```
from sklearn.svm import LinearSVC
svmModel = LinearSVC(penalty='l2', max_iter=2000, dual=True, C=0.25,␣
↪class_weight={1:1, 2:10, 3:10, 4:1, 5:3}).fit(trainData, trainLabels)
```

[213]:
```
# svmModel = LinearSVC(penalty='l2', max_iter=2000, dual=True, C=0.25,␣
↪class_weight={1:1, 2:10, 3:10, 4:1, 5:3}).fit(trainData, trainLabels)
predictedLabels = svmModel.predict(testData)
displayReport(testLabels, predictedLabels, 'SVM')
```

```
Precision, Recall, f1-score for Testing split for SVM
============================================================
Rating 1: 0.7631198844487241, 0.39625, 0.5216389665953596
Rating 2: 0.3938931297709924, 0.5805, 0.46932794340576045
Rating 3: 0.40298507462686567, 0.58725, 0.47797334418557325
Rating 4: 0.649387370405278, 0.17225, 0.27227820588816437
Rating 5: 0.6463604515375633, 0.83025, 0.7268548916611951
Macro Average: 0.5711491821578847, 0.5133, 0.4936146703472105
============================================================
```

6 Logistic Regression

- After experimentation with different hyperparameter settings - best results are achieved with the configuration below.

[221]:
```
from sklearn.linear_model import LogisticRegression
logRegModel = LogisticRegression(penalty='elasticnet', solver='saga',␣
↪l1_ratio=0.5, random_state=0, n_jobs=-1, class_weight={1:1, 2:2, 3:2, 4:1, 5:
↪1}).fit(trainData, trainLabels)
```

[222]:
```
predictedLabelsLogRegn = logRegModel.predict(testData)
displayReport(testLabels, predictedLabelsLogRegn, 'Logistic Regression')
```

```
Precision, Recall, f1-score for Testing split for Logistic Regression
============================================================
Rating 1: 0.764235294117647, 0.406, 0.5302857142857144
```

```
Rating 2: 0.39891250617894214, 0.60525, 0.480881914787963
Rating 3: 0.4165202108963093, 0.5925, 0.48916408668730654
Rating 4: 0.5695564516129032, 0.2825, 0.3776737967914438
Rating 5: 0.6965150048402711, 0.7195, 0.7078209542547959
Macro Average: 0.5691478935292146, 0.52115, 0.5171652933614447
============================================================
```

7 Naive Bayes

- After experimentation with different hyperparameter settings - best results are achieved with the configuration below.

[219]:
```python
from sklearn.naive_bayes import MultinomialNB
mnbModel = MultinomialNB().fit(trainData, trainLabels)
```

[220]:
```python
predictedLabelsMNB = mnbModel.predict(testData)
displayReport(testLabels, predictedLabelsMNB, 'Naive Bayes')
```

```
Precision, Recall, f1-score for Testing split for Naive Bayes
============================================================
Rating 1: 0.6522228474957794, 0.5795, 0.6137145882975907
Rating 2: 0.4172443674176776, 0.4815, 0.447075208913649
Rating 3: 0.4446354038792045, 0.45275, 0.44865601387340515
Rating 4: 0.49503752118131206, 0.51125, 0.5030131595129751
Rating 5: 0.7153888582460011, 0.6485, 0.6803042223970626
Macro Average: 0.5449057996439949, 0.5347, 0.5385526385989364
============================================================
```