NLP220 HW1 Report

Yifei Gan

1 Part A

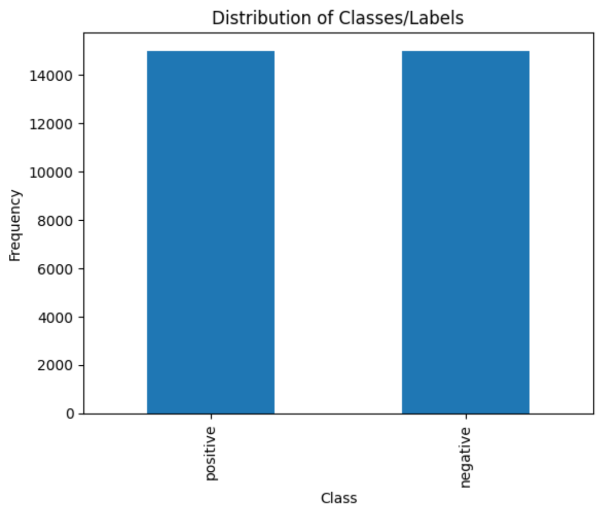

Figure 1: Distribution of Classes/Lables

1.1 Count Vectorizer

Count vectorizer converts text into a matrix of token counts, where each word or token is represented by its frequency in the document, capturing basic term occurrences.

Decision Tree:

$$\begin{bmatrix} 1578 & 696 \\ 661 & 1565 \end{bmatrix}$$

Decision Tree Training Accuracy: 0.99996
Decision Tree Testing Accuracy: 0.69844

Class	Precision	Recall	F1-Score	Support
0	0.70	0.69	0.70	2274
1	0.69	0.70	0.70	2226
Accuracy		0.70		
Macro avg	0.70	0.70	0.70	4500
Weighted avg	0.70	0.70	0.70	4500

Table 1: Classification report for the Decision Tree model with CountVectorizer.

Naive Bayes:

$$\begin{bmatrix} 1918 & 356 \\ 571 & 1655 \end{bmatrix}$$

Training Accuracy: 0.79286
Testing Accuracy: 0.794

Class	Precision	Recall	F1-Score	Support
0	0.77	0.84	0.81	2274
1	0.82	0.74	0.78	2226
Accuracy			0.79	
Macro avg	0.80	0.79	0.79	4500
Weighted avg	0.80	0.79	0.79	4500

Table 2: Naive Bayes with CountVectorizer.

Support Vritual Machine:

$$\begin{bmatrix} 1867 & 407 \\ 317 & 1909 \end{bmatrix}$$

SVC Training Accuracy: 0.8532
SVC Testing Accuracy: 0.83911

Class	Precision	Recall	F1-Score	Support
0	0.85	0.82	0.84	2274
1	0.82	0.86	0.84	2226
Accuracy			0.84	
Macro avg	0.84	0.84	0.84	4500
Weighted avg	0.84	0.84	0.84	4500

Table 3: SVC with CountVectorizer.

1.2 Count Vectorizer by characters

Count vectorizer by character is Similar to Count Vectorizer, but breaks text into sequences of characters rather than words, which helps capture word morphology or structural patterns.

Decision Tree:

$$\begin{bmatrix} 1479 & 795 \\ 724 & 1502 \end{bmatrix}$$

Decision Tree Training Accuracy: 0.99996
Decision Tree Testing Accuracy: 0.66244

Class	Precision	Recall	F1-Score	Support
0	0.67	0.65	0.66	2274
1	0.65	0.67	0.66	2226
Accuracy		0.66		
Macro avg	0.66	0.66	0.66	4500
Weighted avg	0.66	0.66	0.66	4500

Table 4: Decision Tree with CountVectorizer by characters.

Naive Bayes:

$$\begin{bmatrix} 1701 & 573 \\ 840 & 1386 \end{bmatrix}$$

Training Accuracy: 0.68188
Testing Accuracy: 0.686

Class	Precision	Recall	F1-Score	Support
0	0.67	0.75	0.71	2274
1	0.71	0.62	0.66	2226
Accuracy		0.69		
Macro avg	0.69	0.69	0.68	4500
Weighted avg	0.69	0.69	0.68	4500

Table 5: Naive Bayse with CountVectorizer by characters.

Support Virtual Machine:

$$\begin{bmatrix} 1830 & 444 \\ 441 & 1785 \end{bmatrix}$$

SVC Training Accuracy: 0.823294
SVC Testing Accuracy: 0.80333

Class	Precision	Recall	F1-Score	Support
0	0.81	0.80	0.81	2274
1	0.80	0.80	0.80	2226
Accuracy		0.80		
Macro avg	0.80	0.80	0.80	4500
Weighted avg	0.80	0.80	0.80	4500

Table 6: SVC with CountVectorizer by characters.

1.3 TF-IDF

TF-IDF Weighs terms by their frequency in a document relative to their frequency across the entire corpus, highlighting important terms while downplaying common ones.

Decision Tree:

$$\begin{bmatrix} 1593 & 681 \\ 645 & 1581 \end{bmatrix}$$

Decision Tree Training Accuracy: 0.99996
Decision Tree Testing Accuracy: 0.70533

Class	Precision	Recall	F1-Score	Support
0	0.71	0.70	0.71	2274
1	0.70	0.71	0.70	2226
Accuracy		0.71		
Macro avg	0.71	0.71	0.71	4500
Weighted avg	0.71	0.71	0.71	4500

Table 7: Decision Tree with TF-IDF.

Naive Bayes:

$$\begin{bmatrix} 1954 & 320 \\ 337 & 1889 \end{bmatrix}$$

Training Accuracy: 0.86403
Testing Accuracy: 0.854

Class	Precision	Recall	F1-Score	Support
0	0.85	0.86	0.86	2274
1	0.86	0.85	0.85	2226
Accuracy		0.85		
Macro avg	0.85	0.85	0.85	4500
Weighted avg	0.85	0.85	0.85	4500

Table 8: Naive Bayes with TF-IDF.

Support Virtual Machine:

$$\begin{bmatrix} 1976 & 298 \\ 267 & 1959 \end{bmatrix}$$

SVC Training Accuracy: 0.9353
SVC Testing Accuracy: 0.87444

Class	Precision	Recall	F1-Score	Support
0	0.88	0.87	0.87	2274
1	0.87	0.88	0.87	2226
Accuracy		0.87		
Macro avg	0.87	0.87	0.87	4500
Weighted avg	0.87	0.87	0.87	4500

Table 9: SVC with TF-IDF.

Here's a summary of all the model performances. SVM is really good at handling complex data, especially when there are lots of features like word

Accuracy Rate	Count	Count by Character	TF-IDF
Decision Tree	0.698	0.66	0.705
Naive Bayes	0.794	0.686	0.854
SVM	0.839	0.80	0.87

Table 10: Model Performance Metrics

counts or TF-IDF (a way of scoring important words). It looks for the best line (or boundary) that separates the data into classes, even if the data has many layers of complexity. It's also good at not getting "confused" by noise in the data, so it can make better predictions overall.Naive Bayes works well with text data because it's fast and easy. It assumes that all features are independent of each other, which isn't always true, but often works well enough. It does a decent job but isn't as flexible as SVM, which means it can miss some patterns in the data that SVM picks up on.Decision Trees struggle with more complicated data like text features (TF-IDF, word counts, etc.). They tend to "overthink" by splitting the data too much, which leads to poor predictions when you test it on new data. Also, they don't handle a lot of features as well as SVM, so they end up performing the worst in this case.

2 Part B

I've used SVC, naive bayes, logistic regression, random forest classifier, and K-Nearest Neighbors. The text data was vectorized using CountVectorizer with an ngram range of (1, 2). This means that both unigrams (single words) and bigrams (pairs of consecutive words) were extracted as features. The Support Vector Classifier works by finding the optimal hyperplane that separates classes in the feature space. SVC is particularly effective in high-dimensional spaces, making it well-suited for text classification, where feature sets can be large due to the nature of textual data. The Naive Bayes classifier is a probabilistic model based on Bayes' Theorem with the assumption that features are conditionally independent. Logistic Regression is a linear model used for binary classification tasks. It predicts the probability of a sample belonging to a particular class and assigns it based on a decision threshold. The Random Forest Classifier is an ensemble learning method that constructs multiple decision trees during training and outputs the mode of their predictions. It is robust to overfitting and provides high accuracy but can struggle with sparse, high-dimensional data like text because of its reliance on decision trees, which may not handle such data effectively. K-Nearest Neighbors is a non-parametric, distance-based algorithm that classifies samples based on the majority class of their nearest neighbors. While KNN is conceptually simple and can be effective for some types of classification, it is computationally expensive for large datasets and performs poorly when faced with high-dimensional data like text, due to the curse of dimensionality.
To locate which measure of hyperparameters is the most accurate, optuna has been set and used in order to help.

2.1 Naive Bayes

```
alpha = trial.suggest_loguniform('alpha', 1e-4, 1e2)
```

Best hyperparameters for Naive Bayes: alpha: 0.08561959327899799.
Naive Bayes Accuracy with Optuna: 0.8872.

2.2 Logistic Regression

```
C = trial.suggest_loguniform('C', 1e-4, 1e2)
solver = trial.suggest_categorical('solver', ['liblinear', 'saga'])
```

Best hyperparameters for Logistic Regression: C: 0.07546343057088854; solver: liblinear.
Logistic Regression Accuracy with Optuna: 0.9060.

2.3 Random Forest Classifier

```
n_estimators = trial.suggest_int('n_estimators', 10, 200)
max_depth = trial.suggest_int('max_depth', 2, 32)
min_samples_split = trial.suggest_int('min_samples_split', 2, 10)
min_samples_leaf = trial.suggest_int('min_samples_leaf', 1, 10)
```

Best hyperparameters for Random Forest: n_estimators: 179, max_depth: 30; min_samples_split: 3; min_samples_leaf: 2.
Random Forest Accuracy with Optuna: 0.8772.

2.4 K-Nearest Neighbors

```
n_neighbors = trial.suggest_int('n_neighbors', 1, 30)
weights = trial.suggest_categorical('weights',
    ['uniform', 'distance'])
algorithm = trial.suggest_categorical('algorithm',
    ['auto', 'ball_tree', 'kd_tree', 'brute'])
```

Best hyperparameters for K-Nearest Neighbors: n_neighbors: 30; weights: uniform; algorithm: brute.

K-Nearest Neighbors Accuracy with Optuna: 0.6472.

2.5 SVC

I used both SVC and Linear SVC, which resulted in completely different time of running the code. For LinearSVC, it only cost me several seconds, however, the regular SVC kernel linear took me more than 1100 hours to find the best parameters. Additionally, the accuracy rate for SVC is even lower than LinearSVC.

2.5.1 SVC

```
C = trial.suggest_loguniform('C', 1e-4, 1e2)
kernel = trial.suggest_categorical('kernel', ['linear',
    'poly', 'rbf', 'sigmoid'])
gamma = trial.suggest_categorical('gamma', ['scale', 'auto'])
```

Best hyperparameters for Support Vector Classifier: C: 0.007482667477667498; kernel: linear; gamma: scale.
Support Vector Classifier Accuracy with Optuna: 0.8992.

2.5.2 LinearSVC

```
C = trial.suggest_loguniform('C', 1e-4, 1e2)
```

Best hyperparameters for Support Vector Classifier: C: 0.002863754188529008.
Support Vector Classifier Accuracy with Optuna: 0.9028.

3 Part C

After loading the data, I preprocess the data, eliminating all the review_score = 3.0, and assign binary_labels for them.

```
df = df[df['review/score'] != 3]
df['binary_label'] = df['review/score'].apply
    (lambda x: 'positive' if x > 3 else 'negative')
```

Following I perform lemmatization using wordnet in nltk in order to help saving unnecessary computational overhead in deciphering entire words since most words' meanings are well-expressed by their separate lemmas. The text is first broken into individual tokens using the WhitespaceTokenizer() from nltk.

```
import nltk
nltk.download('wordnet')
w_tokenizer = nltk.tokenize.WhitespaceTokenizer()
lemmatizer = nltk.stem.WordNetLemmatizer()
def lemmatize_text(text):
    st = ""
    for w in w_tokenizer.tokenize(text):
        st = st + lemmatizer.lemmatize(w) + " "
    return st
df['review/text'] = df['review/text']
    .apply(lemmatize_text)
```

The dataset is then split into 80% train and 20% test parts using train_test_split.

```
train_sentences, test_sentences, train_labels,
test_labels =train_test_split(
df['review/text'].values,
encoded_labels,
stratify=encoded_labels, test_size=0.15)
```

After this I use CountVectorizer to get the frequency of each word appearing in the training set. I store them in a dictionary called 'word_counts'. All the unique words in the corpus are stored in 'vocab.'

```
vec = CountVectorizer(max_features = 3000)
X = vec.fit_transform(train_sentences)
vocab = vec.get_feature_names_out()
X = X.toarray()
word_counts = {}
for l in range(2):
    word_counts[l] = defaultdict(lambda: 0)
for i in range(X.shape[0]):
    l = train_labels[i]
    for j in range(len(vocab)):
        word_counts[l][vocab[j]] += X[i][j]
```

Then the fit and predict function is defined. The 'fit' function takes reviews and labels values to be fitted on and returns the number of reviews with each label and the apriori conditional probabilities.

```
def fit(x, y, labels):
    n_label_items = {}
    log_label_priors = {}
    n = len(x)
    grouped_data = group_by_label(x, y, labels)
    for l, data in grouped_data.items():
        n_label_items[l] = len(data)
        log_label_priors[l] = math.log
            (n_label_items[l] / n)
    return n_label_items, log_label_priors

def predict(n_label_items, vocab, word_counts,
    log_label_priors, labels, x):
    result = []
    for text in x:
        label_scores = {l: log_label_priors[l]
            for l in labels}
        words = set(w_tokenizer.tokenize(text))
        for word in words:
            if word not in vocab: continue
            for l in labels:
                log_w_given_l = laplace_smoothing
                    (n_label_items, vocab, word_counts, word, l)
                label_scores[l] += log_w_given_l
        result.append(max(label_scores, key=label_scores.get))
    return result
```

At last get the accuracy of the model.

```
labels = [0,1]
n_label_items, log_label_priors = fit
    (train_sentences,train_labels,labels)
pred = predict(n_label_items, vocab,
    word_counts, log_label_priors, labels, test_sentences)
print("Accuracy of prediction on test set :
    ", accuracy_score(test_labels,pred))
```

The Bayes Therom:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

where: $P(A|B)$ is the probability of event A occurring given that B is true. $P(B|A)$ is the probability of event B occurring given that A is true. $P(A)$ is

the probability of event A. $P(B)$ is the probability of event B.
For a given $X\{a_1, a_2, a_3, ..., a_n\}$ to be classified, calculate the probability of each X occurring under the condition y_i that this item appears. The item is assigned to the class with the highest $P(y_i \mid X)$.