CS5785 Applied Machine Learning Homework 2

Yixuan (Rylee) Li yl2557

October 2021

1. *CODING EXERCISES*

(a) Download the data

I downloaded the data set and successfully imported it by implementing following code:

```
# load datasets
df_test  = pd.read_csv("test.csv");
df_train = pd.read_csv("train.csv");
# check data
df_train.head(5)
# check data
df_test.head(5)
```

(1) how many training and test data points are there?

7613 in the training set and 3263 in the testing set.

```
# basic insepction
print(df_train.shape[0], df_test.shape[0])
```

(2) what percentage of the training tweets are of real disasters, and what percentage is not?

42.97% are of real disasters, 57.03% are not.

```
# percentage of the trainning tweets are of real disasters
df_train.loc[df_train['target'] == 1].shape
percentage_real = df_train.loc[df_train['target'] == 1].shape[0] / df_train.shape[0]
print("Percentage of the training tweets are of real disasters: "
        + str(percentage_real * 100) + "%.")
# percentage of the trainning tweets are of not real disasters
percentage_fake = df_train.loc[df_train['target'] == 0].shape[0] / df_train.shape[0]
print("Percentage of the training tweets are NOT of real disasters: "
        + str(percentage_fake * 100) + "%.")
```

(b) Split the training data

```
# Randomly choose 70% of the data points in the training data as the training set
# and the remaining 30% of the data as the development set
train, dev = train_test_split(df_train, test_size=0.3)
train.shape, dev.shape
```

(c) Preprocess the data

The data sets contain significant amounts of noise and unprocessed content. Data cleaning and pre-processing methods would be applied. I explained the reasons for each of my decision (why or why not) in the following paragraph:

- Convert all the words to lowercase: Yes. I think whether the word is lowercase or uppercase doesn't contribute to the classification in the context of our problem. And users (especially Twitter users) may not strictly follow the grammar and use lowercase and uppercase arbitrarily. In this sense, building the vector based on cases is meaningless and makes our operations inefficient. I chose to convert all the words to lowercase.

- Lemmatize all the words: Yes. I think the tense of words doesn't contribute much to the classification in the context of our problem. So I lemmatized all the words based on their POS tags using WordNetLemmatizer from the nltk library. I didn't use nltk.stem since stemming may create the non-existence meaning of a word (eg. from ”causes” to ”caus”). But lemmatization always gives the dictionary meaning word while converting into root-form.

- Strip punctuation: Yes. I think punctuation barely contributes to our analysis of text. Users' choices of punctuation don't impact the meaning of their tweets. So I stripped punctuation using the Python built-in library.

- Strip the stop words, e.g., “the”, “and”, “or”: Yes. Stop words don't not add much information to the text and should be filtered out.

- Strip @ and urls. (It's Twitter.): Yes for ulrs, and No for @. I stripped urls since they are references to a location on the web, but do not provide any additional information. It's meaningless to do NLP and get conclusion from these non-semantic urls. And urls are lengthy, thus causing inefficiency while getting bag of words. I use the re library that provides regular expression matching operations to strip them. I chose not to strip @ since mention of certain users may contribute the our classification problem. For instance, it's possible that mentioning @earthquakeBot may be a feature that contribute to tweets that are about real disasters since the followers of this account closely follow this topic (this is just an assumption).

- Something else: Strip the HTML tags - I found some HTML tags like &amp in the text. They do not add any value to text data and only enable proper browser rendering. So I removed them using re library.

```python
# import packages from nltk and re
from nltk import word_tokenize, pos_tag
from nltk.corpus import wordnet
from nltk.stem import WordNetLemmatizer
import re # Regex library
nltk.download('averaged_perceptron_tagger')
nltk.download('punkt')
nltk.download('stopwords')

# Get pos tags
def get_wordnet_pos(treebank_tag):
    if treebank_tag.startswith('J'):
        return wordnet.ADJ
    elif treebank_tag.startswith('V'):
        return wordnet.VERB
    elif treebank_tag.startswith('N'):
        return wordnet.NOUN
    elif treebank_tag.startswith('R'):
        return wordnet.ADV
    else:
        return None

# Lemmatize all words
def Lemmatize_Sentence(sentence):
    res = []
    lemmatizer = WordNetLemmatizer()
    for word, pos in pos_tag(word_tokenize(sentence)):
        wordnet_pos = get_wordnet_pos(pos) or wordnet.NOUN
        res.append(lemmatizer.lemmatize(word, pos=wordnet_pos))
    return " ".join(res)

# Strip Punctuation
def Strip_Punct(text):
    table = str.maketrans('', '', string.punctuation)
    return text.translate(table)

# Strip Stop words
stopword = stopwords.words('english')
def Strip_StopWord(string):
    string_list = string.split()
    return ' '.join(i for i in string_list if i not in stopword)

# Strip Urls
def Strip_Url(string):
    return re.sub(r'(https|http)?:\/\/(\w|\.|\/|\?|\=|\&|\%|\-)*\b', '', string)

#  Strip HTML Tags
def Strip_HTML(text):
    html = re.compile(r'<.*?>|&([a-z0-9]+|#[0-9]{1,6}|#x[0-9a-f]{1,6});')
    return re.sub(html, '', text)
```

I first converted all the words to lowercase, then applied all the above processing methods:

```
# Convert all the words to lowercase, then apply all the above processing methods
train['text'] = train['text'].str.lower().apply(Lemmatize_Sentence).apply(Strip_Punct)
                .apply(Strip_StopWord).apply(Strip_Url).apply(Strip_HTML)
dev['text'] = dev['text'].str.lower().apply(Lemmatize_Sentence).apply(Strip_Punct)
                .apply(Strip_StopWord).apply(Strip_Url).apply(Strip_HTML)
```

(d) Bag of words model

(1) Choice of threshold M

I chose threshold $M = 3$ to avoid run-time and memory issues, and to avoid noisy/unreliable features that can hurt learning. Intuitively, I was thinking about $M \in [2, 10]$. If M was too small, a run-time and memory issues may occur since we may have very high-dimensional vectors. And too many features may cause overfitting and bad generalization. If M was too large, fewer words/features were selected, thus leading to a poor-performed model that cannot generalize well to new data. Then I ran a test for $M \in [2, 10]$ with the implementation of regularized logistic regression, and I decided to choose $M = 3$ as it reflects the highest F1-score.

```
# Choose threshold to avoid run-time and memory issues
# and to avoid noisy/unreliable features that can hurt learning
threshold = 3
# Build the bag of words feature vectors
def cv(data):
    count_vectorizer = CountVectorizer(min_df = threshold, binary=True)
    emb = count_vectorizer.fit_transform(data)
    return emb, count_vectorizer
# Tokenizing the pre-processed clean texts
train['tokenized'] = train["text"].tolist()
dev['tokenized'] = dev["text"].tolist()
# Build the bag of words feature vectors for both the training and development sets
train_bag_of_words, train_cv = cv(train['tokenized'])
dev_bag_of_words = train_cv.transform(dev['tokenized'])
train_words = train_cv.get_feature_names()
# Report the total number of features in these vectors
print("Total number of features in these vectors: "+ str(len(train_words)))
```

(2) Total number of features in these vectors

Total number of features in these vectors is 2917

(e) Logistic regression

(i) logistic regression model without regularization terms

- Logistic regression without regularization terms - F1 Score on training set: 0.9804177545691906.
- Logistic regression without regularization terms - F1 Score on development set: 0.68407835258664.
- I observed issues with overfitting. This is a very expressive model that fits the training dataset perfectly but doesn't generalize to fit data points outside this dataset.

```
# Train a logistic regression model without regularization terms
LR0_Model = LogisticRegression(penalty='none')
LR0_Model.fit(train_bag_of_words, train['target'])
train_predicted_LR0 = LR0_Model.predict(train_bag_of_words)
dev_predicted_LR0 = LR0_Model.predict(dev_bag_of_words)
#  Report the F1 score in the training and in the development set
print("Logistic regression without regularization terms - F1 Score on training set: "
        + str(f1_score(train_predicted_LR0, train['target'])))
print("Logistic regression without regularization terms - F1 Score on development set: "
        + str(f1_score(dev_predicted_LR0, dev['target'])))
```

(ii) logistic regression model with L1 regularization

- Logistic regression with L1 regularization - F1 Score on training set: 0.8439814814814816.
- Logistic regression with L1 regularization - F1 Score on development set: 0.7417366946778711.

```
# Train a logistic regression model with L1 regularization
LR_L1_Model = LogisticRegression(penalty='l1', solver='liblinear')
LR_L1_Model.fit(train_bag_of_words, train['target'])
train_predicted_LR_L1 = LR_L1_Model.predict(train_bag_of_words)
```

```
dev_predicted_LR_L1 = LR_L1_Model.predict(dev_bag_of_words)
#  Report the F1 score in the training and in the development set
print("Logistic regression with L1 regularization - F1 Score on training set: "
        + str(f1_score(train_predicted_LR_L1, train['target'])))
print("Logistic regression with L1 regularization - F1 Score on development set: "
        + str(f1_score(dev_predicted_LR_L1, dev['target'])))
```

(iii) logistic regression model with L2 regularization

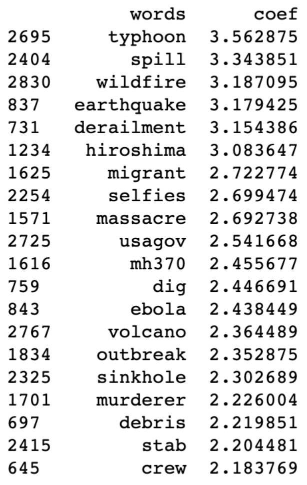
- Logistic regression with L2 regularization - F1 Score on training set: 0.8846855059252506.
- Logistic regression with L2 regularization - F1 Score on development set: 0.7472283813747228.

```
# Train a logistic regression model with L2 regularization
LR_L2_Model = LogisticRegression(penalty='l2', solver='liblinear')
LR_L2_Model.fit(train_bag_of_words, train['target'])
train_predicted_LR_L2 = LR_L2_Model.predict(train_bag_of_words)
dev_predicted_LR_L2 = LR_L2_Model.predict(dev_bag_of_words)
#  Report the F1 score in the training and in the development set
print("Logistic regression with L2 regularization - F1 Score on training set: "
        + str(f1_score(train_predicted_LR_L2, train['target'])))
print("Logistic regression with L2 regularization - F1 Score on development set: "
        + str(f1_score(dev_predicted_LR_L2, dev['target'])))
```

(iv) Which one of the three classifiers performed the best on your training and development set? Did you observe any overfitting and did regularization help reduce it? Support your answers with the classifier performance you got.

- Logistic regression model without regularization performed the best on the training set with a F1-score of 0.9804 on the training set. Logistic regression model with L2 regularization performed the best on the development set with a F1-score of 0.7472 on the development set. This model reflects the highest F1-score on the development set and reduces the problem of over-fitting with regularization.
- Yes, I observed issues of over-fitting while implementing logistic regression model without regularization terms. The model reflects a F1-score of 0.9804 on the training set and a F1-score of 0.6841 on the development set. It fits the training dataset perfectly with a high F1-score but performs poor predictions outside the training set (ie. the development set).
- Both L1 regularization and L2 regularization help reduce the over-fitting. After regularization, the F1-score on development set increases from 0.6841 to 0.7417, 0.7472 respectively.

(v) What are the most important words for deciding whether a tweet is about a real disaster or not?



```
             words      coef
2695       typhoon  3.562875
2404         spill  3.343851
2830      wildfire  3.187095
837     earthquake  3.179425
731     derailment  3.154386
1234     hiroshima  3.083647
1625       migrant  2.722774
2254       selfies  2.699474
1571      massacre  2.692738
2725        usagov  2.541668
1616         mh370  2.455677
759            dig  2.446691
843          ebola  2.438449
2767       volcano  2.364489
1834      outbreak  2.352875
2325      sinkhole  2.302689
1701      murderer  2.226004
697         debris  2.219851
2415          stab  2.204481
645           crew  2.183769
```

Figure 1.1: Top 20 Most Important Words

```python
# Inspect the weight vector of the classifier with L1 regularization
coefficients = pd.concat([pd.DataFrame(train_words, columns=['words']),
                          pd.DataFrame(np.transpose(np.abs(LR_L1_Model.coef_)),
                          columns=['coef'])], axis=1)
# Sort the coefficients
coefficients = coefficients.sort_values(by=['coef'], ascending=False)
# List the most important words for deciding whether a tweet is about areal disaster or not
print(coefficients.head(20))
```

(f) Bernoulli Naive Bayes

- I implemented a Bernoulli Naive Bayes classifier with additive smoothing to predict the probability of whether each tweet is about a real disaster.
- Bernoulli Naive Bayes - F1 Score on training set: 0.8105163429654192.
- Bernoulli Naive Bayes - F1 Score on development set: 0.7416378316032295.

```python
# Compute the maximum likelihood model parameters on our dataset
n = train_bag_of_words.shape[0] # size of the dataset
d = train_bag_of_words.shape[1] # number of features in our dataset
K = 2 # number of clases

# # these are the shapes of the parameters
psis = np.zeros([K,d])
phis = np.zeros([K])
alpha = 1 # additive smoothing

# we now compute the parameters
for k in range(K):
    X_k = train_bag_of_words[train['target'] == k]
    # psis[k] = np.mean(X_k, axis=0)
    psis[k] = (X_k.sum(axis=0) + alpha) / (2 * alpha + X_k.shape[0])
    phis[k] = (X_k.shape[0]) / (float(n))

# print out the class proportions
print(phis)

# Compute predictions using Bayes' rule and implement the model in numpy
def nb_predictions(x, psis, phis):
    """This returns class assignments and scores under the NB model.
    We compute \arg\max_y p(y|x) as \arg\max_y p(x|y)p(y)
    """
    # adjust shapes
    n, d = x.shape
    x = np.reshape(x, (1, n, d))
    psis = np.reshape(psis, (K, 1, d))

    # clip probabilities to avoid log(0)
    psis = psis.clip(1e-14, 1-1e-14)

    # compute log-probabilities
    logpy = np.log(phis).reshape([K,1])
    logpxy = x * np.log(psis) + (1-x) * np.log(1-psis)
    logpyx = logpxy.sum(axis=2) + logpy

    return logpyx.argmax(axis=0).flatten(), logpyx.reshape([K,n])

# Train this classifier on the training set
idx_train, logpyx_train = nb_predictions(train_bag_of_words.toarray(), psis, phis)
idx_dev, logpyx_dev = nb_predictions(dev_bag_of_words.toarray(), psis, phis)
# Report its F1-score on the development set
print("Bernoulli Naive Bayes - F1 Score on training set: "
        + str(f1_score(idx_train, train['target'])))
print("Bernoulli Naive Bayes - F1 Score on development set: "
        + str(f1_score(idx_dev, dev['target'])))
```

(g) Model comparison

Which model performed the best in predicting whether a tweet is of a real disaster or not? Include your performance metric in your response. Comment on the pros and cons of using generative vs discriminative models.

Model	F-1 Score on Training Set	F-1 Score on Development Set
Logistic regressio without regularization	0.9804177545691906	0.68407835258664
Logistic regression with L1 regularization	0.8439814814814816	0.7417366946778711
Logistic regression with L2 regularization	0.8846855059252506	0.7472283813747228
Bernoulli Naive Bayes	0.8105163429654192	0.7416378316032295

- According to the above table, the logistic regression model with L2 regularization in predicting whether a tweet is of a real disaster or not.
- Generative model (Bernoulli Naive Bayes classifier in our problem): Generative model learns the model

that generates the observed data. It can be used to generate new samples as it learns about the distribution. It shows strong resistance to outliers. And it has less problem of overfitting but have high bias (ie problem of underfitting) due to the generalization by assumed distribution.

- Discriminative model (Logistic Regression in our problem): It's more sensitive against outliers in the problem. Discriminative model has low bias, but it may result in over-fitting, thus requiring L1/L2 regularization. Unlike generative model, it makes no attempt to model underlying probability distributions. It only focus on optimizing a mapping from inputs to desired outputs.

Think about the assumptions that Naive Bayes makes. How are the assumptions different from logistic regressions? Discuss whether it's valid and efficient to use Bernoulli Naive Bayes classifier for natural language texts.

- Different from logistic regressions, Naive Bayes assumes independence of predictors. That is, words in the bag are uncorrelated, but in reality they are.
- Although the model of logistic regression with L2 regularization outperforms Bernoulli Naive Bayes classifier on our dataset, Bernoulli Naive Bayes classifier is still valid and efficient with a performance score that is not the best but still good.
- I think the validation and efficiency of Bernoulli Naive Bayes classifier depends on the type/scale of natural language texts problem we are looking into. But in general, it's valid and efficient as it often does well. It's based on a very simple theory and it's easy to build, with no complicated iterative parameter estimation which makes it particularly useful for very large datasets.
  - When assumption of independence holds, it performs better compare to other models (eg. logistic regression) and requires less training data. It performs exceptionally well in case of categorical input variables in comparison to numerical variables, due to the normal distribution assumption.
  - Examples of Scenarios that Bernoulli Naive Bayes classifier may be valid and efficient:
    * Dealing with very high dimensional data sets
    * Providing quick and rough solutions for classification problems
    * Dealing with data set where each category is highly differentiated
  - But when predictors are highly interdependent, the Bernoulli Naive Bayes classifier is a worse estimator compared to other complicated models.

(h) N-gram model

(1) Choice of threshold M

I chose threshold $M = 6$. If M was too small, a run-time and memory issues may occur since we may have very high-dimensional vectors, especially when building a n-gram model. And too many features may cause over-fitting and bad generalizations. If M was too large, fewer words/features (especially bi-grams) were selected, thus leading to a poor-performed model that cannot generalize well to new data. Then I ran a test for $M \in [2, 10]$. I decided to choose $M = 6$ as it obtains 2112 words in the vocabulary (1-grams and 2-grams), which is closed to the total number of words in the vocabulary when we built the bag of words for previous models (ie. 2917). As we built vocabulary sets with similar size, the comparison between models would be more insightful.

```python
# Choose threshold to avoid run-time and memory issues
# and to avoid noisy/unreliable features that can hurt learning
threshold = 6
# Build the bag of words feature vectors
def ngram_cv(data):
    count_vectorizer = CountVectorizer(min_df = threshold, binary=True, ngram_range=(1,2))
    emb = count_vectorizer.fit_transform(data)
    return emb, count_vectorizer
```

(2) Total number of 1-grams and 2-grams in the vocabulary

- Total number of 1-grams and 2-grams in the vocabulary: 2112.
- The total number of 1-gram is: 1570
- The total number of 2-grams is: 542

```
# Build the bag of words feature vectors for both the training and development sets
train_ngram_bag_of_words, train_ngram_cv = ngram_cv(train['tokenized'])
dev_ngram_bag_of_words = train_ngram_cv.transform(dev['tokenized'])
train_ngram_words = train_ngram_cv.get_feature_names()
# Report the total number of 1-grams and 2-grams in the vocabulary.
print("Total number of 1-grams and 2-grams in the vocabulary: "
        + str(train_ngram_bag_of_words.shape[1]))

# count 1-gram and 2-gram respectively
one_gram = 0
two_gram = 0

for word in train_ngram_words:
    if ' ' in word:
        two_gram += 1
    else:
        one_gram += 1
# print out the result
print('The total number of 1-gram is:', one_gram)
print('The total number of 2-grams is:', two_gram)
```

(3) Take 10 2-grams from the vocabulary

['11yearold boy', '12000 nigerian', '15 saudi', '16yr old', '1980 http', '2015 http', '3g whole', '40 family', '5km volcano', '70 year']

```
# Take 10 2-grams from your vocabulary, and print them out
k = 0
i = 0
bigram = [0 for i in range(10)]
while k < 10 and i < len(train_ngram_words):
    words = train_ngram_words[i]
    wordslist = words.split(" ")
    if len(wordslist) == 2:
        bigram[k] = words
        k += 1
    i += 1
print(bigram)
```

(4) Logistic regression with L2 regularization

- Logistic regression with L2 regularization - N gram - F1 Score on training set: 0.8554382744378155

- Logistic regression with L2 regularization - N gram - F1 Score on development set: 0.741111111111111

```
# Train a logistic regression model with L2 regularization
LR_L2_ngram_Model = LogisticRegression(penalty='l2', solver='liblinear')
LR_L2_ngram_Model.fit(train_ngram_bag_of_words, train['target'])
train_predicted_LR_L2_ngram = LR_L2_ngram_Model.predict(train_ngram_bag_of_words)
dev_predicted_LR_L2_ngram = LR_L2_ngram_Model.predict(dev_ngram_bag_of_words)
# Report the F1 score in the training and in the development set
print("Logistic regression with L2 regularization - N gram - F1 Score on training set: "
          + str(f1_score(train_predicted_LR_L2_ngram, train['target'])))
print("Logistic regression with L2 regularization - N gram - F1 Score on development set: "
          + str(f1_score(dev_predicted_LR_L2_ngram, dev['target'])))
```

(5) Bernoulli classifier

- Bernoulli Naive Bayes - N gram - F1 Score on training set: 0.7544574630667344
- Bernoulli Naive Bayes - N gram - F1 Score on development set: 0.7045596502186133

```
# Compute the maximum likelihood model parameters on our dataset
n = train_ngram_bag_of_words.shape[0] # size of the dataset
d = train_ngram_bag_of_words.shape[1] # number of features in our dataset
K = 2 # number of clases

# these are the shapes of the parameters
psis = np.zeros([K,d])
phis = np.zeros([K])
alpha = 1 # additive smoothing

# we now compute the parameters
for k in range(K):
    X_k = train_ngram_bag_of_words[train['target'] == k]
    # psis[k] = np.mean(X_k, axis=0)
    psis[k] = (X_k.sum(axis=0) + alpha) / (2 * alpha + X_k.shape[0])
    phis[k] = (X_k.shape[0]) / (float(n))

# print out the class proportions
print(phis)

# Train this classifier on the training set
idx_train_ngram, logpyx_train_ngram = nb_predictions(train_ngram_bag_of_words.toarray(), psis, phis)
idx_dev_ngram, logpyx_dev_ngram = nb_predictions(dev_ngram_bag_of_words.toarray(), psis, phis)
# Report its F1-score on the development set
print("Bernoulli Naive Bayes - N gram - F1 Score on training set: "
          + str(f1_score(idx_train_ngram, train['target'])))
print("Bernoulli Naive Bayes - N gram - F1 Score on development set: "
          + str(f1_score(idx_dev_ngram, dev['target'])))
```

(6) Do these results differ significantly from those using the bag of words model? Discuss what this implies about the task.

Model	F-1 Score on Training Set	F-1 Score on Development Set
BOW - Logistic regression (L2)	0.8846855059252506	0.7472283813747228
BOW - Bernoulli Naive Bayes	0.8105163429654192	0.7416378316032295
2gram - Logistic regression (L2)	0.8554382744378155	0.741111111111111
2gram - Bernoulli Naive Bayes	0.7544574630667344	0.7045596502186133

No, these results do not differ significantly from those using the bag of words model. The bag of words does not consider the order of the words in which they appear in a tweet, and only individual words are counted. N-grams captures the context in which the words are used together. N-gram is implemented based on the Markov assumption that the probability of a word depends on the previous word. In our case, the N-gram model performs worse than those using a bag of words, implying that based on our training set, the dependency of continuous words does not contribute much towards whether a tweet is about a real disaster or not.

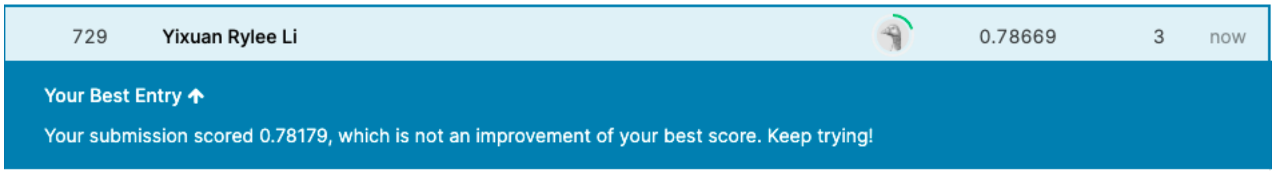
(i) Determine performance with the test set

(1) Re-build the feature vectors and re-train the preferred classifier - Logistic Regression with L2 regularization

```
# Pre-process data
df_train["text"] = df_train["text"].str.lower().apply(Lemmatize_Sentence)
                .apply(Strip_Punct).apply(Strip_StopWord).apply(Strip_Url).apply(Strip_HTML)
df_test["text"] = df_test["text"].str.lower().apply(Lemmatize_Sentence)
                .apply(Strip_Punct).apply(Strip_StopWord).apply(Strip_Url).apply(Strip_HTML)
# Tokenizing the pre-processed clean texts
df_train['tokenized'] = df_train["text"].tolist()
df_test['tokenized'] = df_test["text"].tolist()
# Re-build the feature vectors
train_BOW, data_cv = cv(df_train['tokenized'])
test_BOW = data_cv.transform(df_test['tokenized'])
# Re-train the preferred classifier - Logistic Regression with L2 regularization
LR_L2_Model = LogisticRegression(penalty='l2', solver='liblinear')
LR_L2_Model.fit(train_BOW, df_train['target'])
train_predicted = LR_L2_Model.predict(train_BOW)
test_predicted = LR_L2_Model.predict(test_BOW)
# save predictions
res=pd.DataFrame(columns = ['id', 'target'])
res['id'] = df_test['id']
res['target'] = test_predicted
res.to_csv('my_submission.csv',index=False)
```

(2) Report the resulting F1-score on the test data

The resulting F1-score on the test data is 0.78669.



729 Yixuan Rylee Li 0.78669 3 now

Your Best Entry

Your submission scored 0.78179, which is not an improvement of your best score. Keep trying!

Figure 1.2: Kaggle Submission Result

(3) Was this lower or higher than you expected? Discuss why it might be lower or higher than your expectation

It's higher than what I expected. When I fine-tuned the model, the F-1 score of using logistic regression model with L2 regularization is around 0.74. I thought the F-1 score of the Kaggle submission would be lower than that. I think the result is higher than my expectation since more data points are used as training set (only 70% of the training set are used for fine-tune).

2. *WRITTEN EXERCISES*

Question 1 Naive Bayes with Binary Features

Let A, F, C represent the events of having COVID, having fever, and coughing respectively. From the question, we have the following probabilities:

$$\mathbb{P}(A) = 0.1, \mathbb{P}(A^c) = 0.9 \tag{2.1}$$

$$\mathbb{P}(F, C|A) = 0.75, \mathbb{P}(F, C^c|A) = 0.05, \mathbb{P}(F^c, C|A) = 0.05, \mathbb{P}(F^c, C^c|A) = 0.15 \tag{2.2}$$

$$\mathbb{P}(F, C|A^c) = 0.04, \mathbb{P}(F, C^c|A^c) = 0.01, \mathbb{P}(F^c, C|A^c) = 0.01, \mathbb{P}(F^c, C^c|A^c) = 0.94 \tag{2.3}$$

(a) Suppose the hospital is presented with a patient who has both fever and coughing. According to the full data distribution presented above, what is the probability that the patient doesn't have COVID?

According to Bayes' theorem, we have:

$$\mathbb{P}(F^c|F, C) = \frac{\mathbb{P}(F, C|A^c) \cdot \mathbb{P}(A^c)}{\mathbb{P}(F, C)} = \frac{\mathbb{P}(F, C|A^c) \cdot \mathbb{P}(A^c)}{\mathbb{P}(F, C|A) \cdot \mathbb{P}(A) + \mathbb{P}(F, C|A^c) \cdot \mathbb{P}(A^c)} \tag{2.4}$$

$$= \frac{0.04 * 0.9}{0.75 * 0.1 + 0.04 * 0.9} = 0.3243 \tag{2.5}$$

(b) Next, suppose we have trained a Naive Bayes classifier using a very large dataset sampled IID from the data distribution and we have found the best Naive Bayes model that approximates this data distribution. What is the probability that the above patient doesn't have COVID according to the naive Bayes model?

$$\mathbb{P}(F^c|F, C) = \frac{\mathbb{P}(F, C|A^c) \cdot \mathbb{P}(A^c)}{\mathbb{P}(F, C)} = \frac{\mathbb{P}(F|A^c) \cdot \mathbb{P}(C|A^c) \cdot \mathbb{P}(A^c)}{\mathbb{P}(F|A) \cdot \mathbb{P}(C|A) \cdot \mathbb{P}(A) + \mathbb{P}(F|A^c) \cdot \mathbb{P}(C|A^c) \cdot \mathbb{P}(A^c)}$$

$$= \frac{(\mathbb{P}(F, C|A^c) + \mathbb{P}(F, C^c|A^c)))(\mathbb{P}(C, F|A^c) + \mathbb{P}(C, F^c|A^c))\mathbb{P}(A^c)}{(\mathbb{P}(F, C|A) + \mathbb{P}(F, C^c|A))(\mathbb{P}(C, F|A) + \mathbb{P}(C, F^c|A))\mathbb{P}(A) + (\mathbb{P}(F, C^c|A^c) + \mathbb{P}(F, C|A^c))(\mathbb{P}(C, F|A^c) + \mathbb{P}(C, F^c|A^c))\mathbb{P}(A^c)}$$

$$= \frac{(0.04 + 0.01) * (0.04 + 0.01) * 0.9}{(0.75 + 0.05) * (0.75 + 0.05) * 0.1 + (0.04 + 0.01) * (0.04 + 0.01) * 0.9} = 0.03396$$

(c) Do the above approaches give significantly different probabilities that the patient doesn't have COVID? If so, why? Which approach do you think gives a more reasonable estimate? You should relate your answer to the different assumptions made by the two approaches.

Yes, the above two approaches give significantly different probabilities that the patient doesn't have COVID. The first approach using Bayes' theorem assumes that the events can be either independent or nonindependent. The second approach assumes the independence among indicators. That is, having fever and coughing are independent events. According to the different assumptions made by the two approaches, I think the first approach gives a more reasonable estimate. The second approach assumes that the presence of having fever is unrelated to the presence of coughing. However, this assumption of independence does not hold in reality.

Question 2 Categorical Naive Bayes

(1) Show that the maximum likelihood estimate for the parameters ϕ is $\phi^* = \frac{n_k}{n}$, where n_k is the number of data points with class k.

We want to maximize the log likelihood:

$$\frac{1}{n}(\sum_{k=1}^{K} n_k \cdot log(\phi_k) + \sum_{j=1}^{d}\sum_{k=1}^{K}\sum_{l=1}^{L} n_{jkl} \cdot log(\psi_{jkl})) \tag{2.6}$$

For problem (1) the log function we are examining is

$$\sum_{k=1}^{K} n_k \cdot log(\phi_k) \tag{2.7}$$

To find the maximum likelihood estimator, we have to find the maximum of this log likelihood function. Since there is a constraint:

$$\phi_1 + \phi_2 + \cdots + \phi_k = 1 \tag{2.8}$$

I use the Lagrange multiplier method, take the derivative. and set it to zero:

$$\frac{\partial}{\partial \phi_j}\{\sum_{k=1}^{K} n_k \cdot log(\phi_k) + \lambda(\sum_{k=1}^{K} \phi_j - 1)\} = 0 \tag{2.9}$$

From (2.9) we have:

$$\frac{n_j}{\phi_j} + \lambda = 0 \text{ , for } j = 1, \dots, k \tag{2.10}$$

Then we have:

$$n_j + \lambda\phi_j = 0 \tag{2.11}$$

Summing this over j and using constraint (2.8) we get $\lambda = -n$.

Hence, we have:

$$\phi^* = \frac{n_k}{n} \tag{2.12}$$

(2) Show that the maximum likelihood estimate for the parameters ψ is $\psi^*_{jkl} = \frac{n_{jkl}}{n_k}$, where n_{jkl} is the number of data points with class k for which the j-th feature equals.

We want to maximize the log likelihood:

$$\frac{1}{n}(\sum_{k=1}^{K} n_k \cdot log(\phi_k) + \sum_{j=1}^{d}\sum_{k=1}^{K}\sum_{l=1}^{L} n_{jkl} \cdot log(\psi_{jkl})) \tag{2.13}$$

For problem (2) the log function we are examining is

$$\sum_{j=1}^{d}\sum_{k=1}^{K}\sum_{l=1}^{L} n_{jkl} \cdot log(\psi_{jkl}) \tag{2.14}$$

To find the maximum likelihood estimator, we have to find the maximum of this log likelihood function. Since there is a constraint:

$$\psi_{111} + \cdots + \psi_{dKL} = 1 \tag{2.15}$$

I use the Lagrange multiplier method, take the derivative. and set it to zero:

$$\frac{\partial}{\partial \psi_{abc}}\{\sum_{j=1}^{d}\sum_{k=1}^{K}\sum_{l=1}^{L} n_{jkl} \cdot log(\psi_{jkl}) + \lambda(\sum_{j=1}^{d}\sum_{k=1}^{K}\sum_{l=1}^{L} \psi_{abc} - 1)\} = 0 \tag{2.16}$$

From (2.16) we have:

$$\frac{n_{abc}}{\psi_{abc}} + \lambda = 0 \text{ , for } a = 1, \ldots, j; b = 1, \ldots, k; c = 1, \ldots, l \tag{2.17}$$

Then we have:

$$n_{abc} + \lambda \psi_{abc} = 0 \tag{2.18}$$

Summing this over a, b and c and using constraint (2.15) we get $\lambda = -n_k$.

Hence, we have:

$$\psi^*_{jkl} = \frac{n_{jkl}}{n_k} \tag{2.19}$$