Homework 2 Binary Classification on Text Data

Part a: Download the data

In [1]:
```
import pandas as pd
import numpy as np
```

In [2]:
```
train = pd.read_csv("train.csv")
test = pd.read_csv("test.csv")
```

In [3]:
```
print('Training data points:', len(train))
print('Test data points:', len(test))
```

```
Training data points: 7613
Test data points: 3263
```

1. There are 7613 training data points and 3263 test data points

In [4]:
```
len(train[train['target'] == 1]) / len(train)
```

Out[4]:
```
0.4296597924602653
```

In [5]:
```
len(train[train['target'] == 0]) / len(train)
```

Out[5]:
```
0.5703402075397347
```

2. ~43% of the training tweets are about real disasters and ~57% of the training tweets are about non-real disasters.

Part b: Split the training data

In [6]:
```
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7613 entries, 0 to 7612
Data columns (total 5 columns):
 #   Column    Non-Null Count  Dtype 
---  ------    --------------  ----- 
 0   id        7613 non-null   int64 
 1   keyword   7552 non-null   object
 2   location  5080 non-null   object
 3   text      7613 non-null   object
 4   target    7613 non-null   int64 
dtypes: int64(2), object(3)
memory usage: 297.5+ KB
```

In [7]:
```
from sklearn.model_selection import train_test_split

X = list(train.columns[:-1])

# Splits train.csv into training set (70%) and development set (30%)
X_train, X_dev, y_train, y_dev = train_test_split(train[X], train['target'], test_size=0.3, random_sta
X_train
```

Out[7]:

	id	keyword	location	text
476	686	attack	#UNITE THE BLUE	@blazerfan not everyone can see ignoranceshe i...
4854	6913	mass%20murderer	NaN	White people I know you worry tirelessly about...
4270	6066	heat%20wave	NaN	Chilli heat wave Doritos never fail!
992	1441	body%20bagging	New Your	@BroseidonRex @dapurplesharpie I skimmed throu...
4475	6365	hostages	cuba	#hot C-130 specially modified to land in a st...
...	...	...	...	...
4931	7025	mayhem	Manavadar, Gujarat	They are the real heroes... RIP Brave hearts.....
3264	4689	engulfed	USA	Car engulfed in flames backs up traffic at Par...
1653	2388	collapsed	Alexandria, Egypt.	Great British Bake Off's back and Dorret's cho...
2607	3742	destroyed	USA	Black Eye 9: A space battle occurred at Star O...
2732	3924	devastated	Dorset, UK	???????????? @MikeParrActor absolutely devasta...

5329 rows × 4 columns

Part c: Preprocess the data

To clean noise and unprocessed content, we did the following:

- Convert all the words to lowercase
- Remove all URLs
- Removes usernames (e.g. @twitter)
- Strip punctuations
- Strip stop words
- Lemmatize words based on part of speech

We converted all the words to lowercase because the first letter of the first word in a sentence is usually capitalized. Moreover, text could be in all caps.

We removed all URLs because URLs are mostly sequences of meaningless characters and do not contain valuable information.

We removed Twitter usernames because usernames do not provide valuable information about the text itself.

We stripped punctuations because not everyone uses punctuations when they tweet.

We generated our own set of stop words and stripped them from the text. Stop words are commonly used words that do not provide much information about the text.

Lastly, we lemmatized words based on the part of speech so that we don't have different variations of the same word.

In [8]:

```
import re

stop_words = ['the', 'a', 'an', 'and', 'or', 'this', 'that', 'i', 'my', 'me', 'we', 'us', 'our', 'she'
              'he', 'his', 'him', 'they', 'their', 'them', 'you', 'your', 'there', 'are', 'is', 'from'
              'will', 'can', 'cant', 'must', 'might', 'should', 'shouldnt', 'would', 'wouldnt', 'has',
              'have', 'havent', 'could', 'couldnt', 'as', 'if', 'in', 'on', 'also', 'at', 'such', 'unde
              'lmao', 'haha', 'of', 'into', 'over', 'with', 'by', 'be', 'it', 'its', 'so', 'im', 'your
              'hes', 'shes', 'were', 'was', 'not', 'but', 'no', 'never', 'with', 'really', 'do', 'does
              'doesnt', 'for', 'about', 'what', 'how', 'who', 'just', 'when', 'via', 'which', 'than',
              'yeah', 'up', 'more', 'most', 'isnt', 'arent', 'am', 'wont', 'werent', 'wasnt', 'yet']

def regex_stop_word(words):
    regex = r'\b'
    for i in range(len(words)):
        if i == len(words) - 1:
            regex += words[i] + r'\b'
        else:
```

```
            regex += words[i] + r'\b|\b'
    return regex


def preprocess_text(X):
    # Converts all the words to lowercase
    X = X.apply(lambda text: text.lower())

    # Removes URLs
    X = X.apply(lambda text: re.sub(r'http\S+', ' ', text))

    # Removes user id
    X = X.apply(lambda text: re.sub(r'@(.*?)[\s]', ' ', text))

    # Strips punctuations
    X = X.apply(lambda text: text.replace('-', ' '))
    X = X.apply(lambda text: re.sub(r'[^\w\s]', '', text))

    # Strips stop words
    X = X.apply(lambda text: re.sub(regex_stop_word(stop_words), ' ', text))
    return X

X_train['text'] = preprocess_text(X_train['text'])
X_dev['text'] = preprocess_text(X_dev['text'])
```

In [9]:

```
from nltk.stem import WordNetLemmatizer
from nltk.corpus import wordnet
from nltk.tag import pos_tag

# import nltk
# nltk.download()

# Lemmatize words based on part of speech (verbs, adjectives, and nouns)
def lemmatize(text):
    wnl = WordNetLemmatizer()
    word_tags = pos_tag(text.split())
    result_text = []
    for word_tag in word_tags:
        lemmatized_word = word_tag[0]
        # lemmatize verbs (e.g. ate -> eat)
        if 'VB' in word_tag[1]:
            lemmatized_word = wnl.lemmatize(word_tag[0], pos='v')
        # lemmatize adjectives (e.g. better -> good)
        elif 'JJ' in word_tag[1]:
            lemmatized_word = wnl.lemmatize(word_tag[0], pos='a')
        # lemmatize nouns (e.g. cookies -> cookie)
        elif 'NN' in word_tag[1]:
            lemmatized_word = wnl.lemmatize(word_tag[0], pos='n')
        result_text.append(lemmatized_word)
    return ' '.join(result_text)

X_train['text'] = X_train['text'].apply(lambda text: lemmatize(text))
X_dev['text'] = X_dev['text'].apply(lambda text: lemmatize(text))
test['text'] = test['text'].apply(lambda text: lemmatize(text))
X_train['text']
```

Out[9]:

```
476     everyone see ignoranceshe latinoand all ever b...
4854    white people know worry tirelessly black black...
4270                        chilli heat wave doritos fail
992                      skim through twitter miss body bag
4475    hot c 130 specially modified land stadium resc...
                               ...
4931                              real hero rip brave heart
3264      car engulf flames back traffic parleyûªs summit
1653    great british bake offs back dorrets chocolate...
2607    black eye 9 space battle occur star o784 invol...
2732    absolutely devastate actor miss rossbarton eve...
Name: text, Length: 5329, dtype: object
```

Part d: Bag of words model

The threshold selected for our bag of words model is 10. In other words, our model includes only the words that appear in at least 10 different tweets.

We selected 10 as our threshold by evaluating the performance of our logistic regression model on the development set with different thresholds as shown below.

In [10]:
```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import f1_score

def f1_score_dev(M, xtrain, ytrain, xdev, ydev, ngram_range=(1, 1)):
    vectorizer = CountVectorizer(binary=True, min_df=M, ngram_range=ngram_range)
    vtz = vectorizer.fit(xtrain)
    V_train = vtz.transform(xtrain).toarray()
    V_dev = vtz.transform(xdev).toarray()

    clf = LogisticRegression(solver='saga', penalty='l1', max_iter=3000).fit(V_train, ytrain)
    y_dev_predict = clf.predict(V_dev)
    f1_dev = f1_score(ydev, y_dev_predict)
    print('When M = ', M , ', F1 score = ', f1_dev, sep='')
```

In [11]:
```
f1_score_dev(10, X_train['text'], y_train, X_dev['text'], y_dev)
f1_score_dev(15, X_train['text'], y_train, X_dev['text'], y_dev)
f1_score_dev(20, X_train['text'], y_train, X_dev['text'], y_dev)
```

```
When M = 10, F1 score = 0.7383230163196398
When M = 15, F1 score = 0.7226318774815654
When M = 20, F1 score = 0.7163841807909606
```

When M < 10, we ran into a run-time problem and were forced to choose a higher threshold.

From the F1 scores above, we concluded that M = 10 prunes the tweets just enough to provide good performance.

In [12]:
```
M = 10
vectorizer = CountVectorizer(binary=True, min_df=M)
vtz = vectorizer.fit(X_train['text'])
V_train = vtz.transform(X_train['text']).toarray()
V_dev = vtz.transform(X_dev['text']).toarray()
```

Part e: Logistic Regression

In [13]:
```
def logistic_regression_f1(penalty):
    clf = LogisticRegression(penalty=penalty, solver='saga', max_iter = 3000).fit(V_train, y_train)
    y_train_predict = clf.predict(V_train)
    f1_train = f1_score(y_train, y_train_predict)

    y_dev_predict = clf.predict(V_dev)
    f1_dev = f1_score(y_dev, y_dev_predict)

    return clf, f1_train, f1_dev
```

In [14]:
```
print('F1 scores for logistic regression model without regularization')
(clf_none, f1_train, f1_dev) = logistic_regression_f1('none')
print('\tTraining data:', f1_train)
print('\tDevelopment data:', f1_dev)
```

```
F1 scores for logistic regression model without regularization
        Training data: 0.839213934792318
        Development data: 0.7184986595174263
```

The F1 score in the training data (~0.839) is significantly higher than the F1 score in the development data (~0.719). This means that our model performs significantly better in the training data than it does in the development data. Therefore, our logistic regression model without regularization terms is overfitting.

In [15]:
```
print('F1 scores for logistic regression model with L1 regularization')
(clf_l1, f1_train, f1_dev) = logistic_regression_f1('l1')
```

```
print('\tTraining data:', f1_train)
print('\tDevelopment data:', f1_dev)
```

```
F1 scores for logistic regression model with L1 regularization
	Training data: 0.8068886337543054
	Development data: 0.7383230163196398
```

In [16]:
```
print('F1 scores for logistic regression model with L2 regularization')
(clf_l2, f1_train, f1_dev) = logistic_regression_f1('l2')
print('\tTraining data:', f1_train)
print('\tDevelopment data:', f1_dev)
```

```
F1 scores for logistic regression model with L2 regularization
	Training data: 0.8101033295063145
	Development data: 0.7340067340067339
```

Among the three logistic regression models, the logistic regression model without regularization terms performed the best on the training data with an F1 score of ~0.839. However, the logistic regression model with L1 regularization performed the best on the development data with an F1 score of ~0.738.

The difference between the F1 score in the training data and the F1 score in the development data for the logistic regression model without regularization terms is ~0.12. The difference in the F1 scores for the model with L1 regularization is ~0.069. The difference in the F1 scores for the model with L2 regularization is ~0.076. For the two models with regularization, the difference seems to be smaller than that of the model without any regularization terms. Moreover, the F1 scores in the develpment data for the models with regularization are higher than that of the model without regularization. Therefore, we can conclude that regularization helped reduce overfitting in our logistic regression model.

In [17]:
```
# Inspect weight vector of classifier with L1 regularization

param_dict = dict()
words = vtz.inverse_transform(clf_l1.coef_)[0]
for i in range(len(words)):
    param_dict[words[i]] = clf_l1.coef_[0][i]

sorted_dict = sorted(param_dict.items(), key=lambda x: x[1], reverse=True)
sorted_dict[:20]
```

Out[17]:
```
[('want', 3.4756825562170186),
 ('volcano', 3.4663539281278735),
 ('israeli', 3.380726824816572),
 ('room', 3.335618090569478),
 ('hawaii', 2.586177243424469),
 ('south', 2.480811172346849),
 ('typhoon', 2.389500095522087),
 ('fukushima', 2.35492928625883),
 ('outrage', 2.2322912817082896),
 ('disaster', 2.1728721371828836),
 ('guess', 2.1592005131333725),
 ('life', 2.125643352377016),
 ('injure', 2.0971814688114643),
 ('say', 2.029047746262981),
 ('bioterror', 1.983672174905185),
 ('wound', 1.954552024898809),
 ('york', 1.8926312986755938),
 ('due', 1.787475483679735),
 ('st', 1.717247608550534),
 ('issue', 1.716380209881435)]
```

The words above are some of the most important words for deciding whether a tweet is about a real disaster or not.

Part f: Bernoulli Naive Bayes

In [18]:
```
# Bernoulli Naive Bayes

def nb_predictions(x, psis, phis, K):
    # 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])


def Bernoulli_Naive_Bayes(xtrain, ytrain, xdev, K, alpha):
    n = xtrain.shape[0]  # number of tweets
    d = xtrain.shape[1]  # number of words in dataset
    psis = np.zeros([K,d])
    phis = np.zeros([K])

    for k in range(K):
        X_k = xtrain[ytrain == k]
        psis[k] = (np.sum(X_k, axis=0) + alpha) / (X_k.shape[0] + 2 * alpha)
        phis[k] = X_k.shape[0] / float(n)

    return nb_predictions(xdev, psis, phis, K)[0]

idx = Bernoulli_Naive_Bayes(V_train, y_train, V_dev, K = 2, alpha = 1)
print(f1_score(idx, y_dev, average='micro'))
```

```
0.792907180385289
```

The F1 score on the development set for our Bernoulli Naive Bayes classifier is ~0.793

Part g: Model Comparison

The F1 scores on the development set for our four classifiers are the following:

- Bernoulli Naive Bayes: ~0.793
- Logistic Regression with L1 Regularization: ~0.738
- Logistic Regression with L2 Regularization: ~0.734
- Logistic Regression withouot Regularization: ~0.719

The Bernoulli Naive Bayes model performed the best in predicting whether a tweet is of a real disaster or not, with an F1 score of ~0.793.

One advantage of using generative models is that we do not need to worry about dealing with missing values and noisy inputs with generative models. Moreover, the maximum likelihood parameters are fairly simple for generative models. However, generative models might not perform well if the assumptions made for building the models are not true.

One advantage of using discriminative models is that they are often more accurate than generative models because they are based on fewer assumptions than generative models. On the other hand, the accuracy of the discriminative models could be low when we have inputs with missing values.

The Bernoulli Naive Bayes model assumes that each word in the text is independent, meaning that for each word and each class, the probability of observing that word within that class can be represented using a single number. However, for the logistic regression, the conditional probability is used for the words instead, meaning that some correlations in the words have a minimal effect on the performance.

Since not all the words are present in each Tweet, it is efficient to use a Bernoulli Naive Bayes classifier for natural language texts. We do not need to deal with missing words when we use a Bernoulli Naive Bayes classifier. It is also efficient as it uses simple formulas to calculate maximum likelihood parameters. However, if the words in a sentence have high correlations with each other, our initial assumption for the Naive Bayes classifier might not hold true and thus, it would not perform well.

Part h: N-gram Model

The threshold selected for the 2-gram model is 10. Our 2-gram model includes only the words that appear in at least 10 different tweets.

We selected 10 as our threshold by evaluating the performance of our logistic regression model with L1 regularization on the development set with different thresholds as shown below.

In [19]:
```
f1_score_dev(10, X_train['text'], y_train, X_dev['text'], y_dev, (1,2))
f1_score_dev(11, X_train['text'], y_train, X_dev['text'], y_dev, (1,2))
f1_score_dev(12, X_train['text'], y_train, X_dev['text'], y_dev, (1,2))
f1_score_dev(13, X_train['text'], y_train, X_dev['text'], y_dev, (1,2))
f1_score_dev(14, X_train['text'], y_train, X_dev['text'], y_dev, (1,2))
f1_score_dev(15, X_train['text'], y_train, X_dev['text'], y_dev, (1,2))
f1_score_dev(20, X_train['text'], y_train, X_dev['text'], y_dev, (1,2))
```

```
When M = 10, F1 score = 0.7320113314447593
When M = 11, F1 score = 0.7293318233295584
When M = 12, F1 score = 0.7319004524886878
When M = 13, F1 score = 0.7250996015936255
When M = 14, F1 score = 0.7241576242147345
When M = 15, F1 score = 0.7246871444823663
When M = 20, F1 score = 0.7178329571106093
```

Again, when M < 10, we ran into a run-time problem and were forced to choose a higher threshold. From the F1 scores above, we concluded that M = 10 prunes the tweets just enough to provide good performance.

In [20]:
```
# 2-gram set up

M2 = 10
vectorizer2 = CountVectorizer(binary=True, min_df=M2, ngram_range=(1,2))
vtz2 = vectorizer2.fit(X_train['text'])
V_train2 = vtz2.transform(X_train['text']).toarray()
V_dev2 = vtz2.transform(X_dev['text']).toarray()
```

In [21]:
```
# Print 10 2-grams from vocabulary
features = vectorizer2.get_feature_names_out()
counter = 0
for feature in features:
    if counter < 10:
        if ' ' in feature:
            print(feature)
            counter += 1
    else:
        break
```

```
08 05
12000 nigerian
15 saudi
16yr old
2015 prebreak
40 family
70 year
add video
affect fatal
after exchange
```

The words listed above are some of the 2-grams from the vocabulary of our 2-gram model.

In [22]:
```
# Number of 1-grams and 2-grams
onegram_counter = 0
twogram_counter = 0
for feature in features:
    if ' ' in feature:
        twogram_counter += 1
    else:
        onegram_counter += 1
```

```
print(onegram_counter, '1-grams')
print(twogram_counter, '2-grams')
```

```
1038 1-grams
244 2-grams
```

For our 2-gram model, there are 1038 1-grams and 244 2-grams.

We generated a logistic regression model with L1 regularization on our 2-gram.

Logistic regression with L1 is chosen because it had the highest F-score in development set for 2-gram.

In [23]:
```
# Logistic regression with L1 regularization on 2-gram

clf_2gram = LogisticRegression(penalty='l1', solver='saga', max_iter=3000).fit(V_train2, y_train)
y_train_predict = clf_2gram.predict(V_train2)
f1_train = f1_score(y_train, y_train_predict)
print('\tTraining data:', f1_train)

y_dev_predict = clf_2gram.predict(V_dev2)
f1_dev = f1_score(y_dev, y_dev_predict)
print('\tDevelopment data:',f1_dev)
```

```
        Training data: 0.8068234209313048
        Development data: 0.7320113314447593
```

Then, we generated a Bernoulli classifier on our 2-gram.

In [24]:
```
# Bernoulli classifier on 2-gram

idx_train = Bernoulli_Naive_Bayes(V_train2, y_train, V_train2, K = 2, alpha = 1)
print('\tTraining data:', f1_score(idx_train, y_train, average='micro'))

idx_dev = Bernoulli_Naive_Bayes(V_train2, y_train, V_dev2, K = 2, alpha = 1)
print('\tDevelopment data:', f1_score(idx_dev, y_dev, average='micro'))
```

```
        Training data: 0.8087821354850816
        Development data: 0.7946584938704028
```

The F1 score for the logistic regression model with L1 regularization using the bags of words on the development set was ~0.738 and the F1 score for the logistic regression model using the 2-grams on the development set was ~0.732. The difference in these results is only ~0.006, meaning that they do not differ significantly.

The F1 score for the Bernoulli classifier using the bags of words on the development set was ~0.793 and the F1 score for the Bernoulli classifier using the 2-grams on the development set was ~0.795. The difference in these results is only ~0.002, meaning that they do not differ significantly, again.

The small difference in these results implies that 2-grams do not play much role in our data. This might mean that consecutive words in our text data are mostly independent, resulting in similar results to the bag of words model.

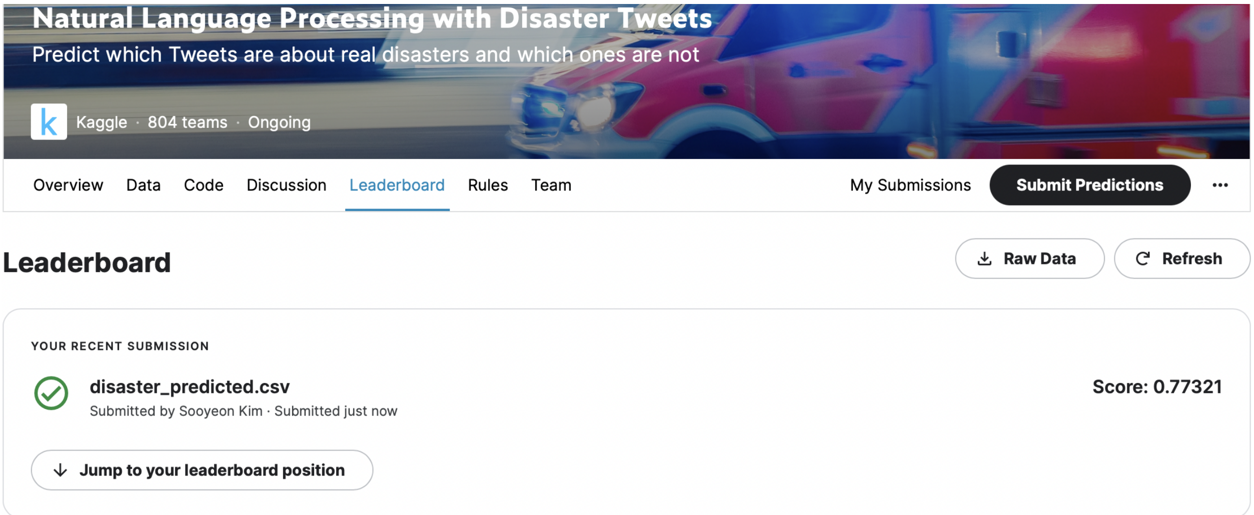
Part i: Determine performance with the test set

The final model that we settled on was the Bernoulli classifier using 2-grams. We trained this classifier on the entire training data.

In [25]:
```
test['text'] = preprocess_text(test['text'])
train['text'] = preprocess_text(train['text'])

M3 = 10
vectorizer3 = CountVectorizer(binary=True, min_df=M3, ngram_range=(1,2))
vtz3 = vectorizer3.fit(train['text'])
V_train3 = vtz3.transform(train['text']).toarray()
V_test = vtz3.transform(test['text']).toarray()
idx_test = Bernoulli_Naive_Bayes(V_train3, train['target'], V_test, K = 2, alpha = 1)

# Generates final df that is used for creating a csv file
final_df = pd.DataFrame({'id': test['id'], 'target': idx_test})
final_df.to_csv('disaster_predicted.csv', index=False)
```



The F1 score on the test data is 0.77321.

We expected the test data F1 score to be very close to the development data F1 score for the Bernoulli classifier using the 2-grams because the development data F1 score was very close to that of the training data. In other words, we expected the F1 score for the test data to be close to 0.795. The actual F1 score came out to be slightly lower than our expectation, meaning that our model could be slightly overfitting. This might be due to an increase in the volume of vocabulary used in our classifier.