1. Binary Classification on Text Data.

In this problem, you will implement several machine learning techniques from the class to perform classification on text data. Throughout the problem, we will be working on the NLP with Disaster Tweets Kaggle competition, where the task is to predict whether or not a tweet is about a real disaster.

In [ ]:
```
#Hrudai Battini HW 2, Part 2 Applied Machine Learning
import numpy as np
import seaborn as sns
import os
import pandas as pd
import nltk
import string
import re
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import f1_score
from matplotlib import pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages
from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords
import nltk
nltk.download('wordnet')
nltk.download("omw-1.4")
nltk.download('stopwords')
```

```
[nltk_data] Downloading package wordnet to
[nltk_data]     C:\Users\hruda\AppData\Roaming\nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package omw-1.4 to
[nltk_data]     C:\Users\hruda\AppData\Roaming\nltk_data...
[nltk_data]   Package omw-1.4 is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\hruda\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

Out[ ]: True

(a) Download the data.

Download the training and test data from Kaggle, and answer the follow- ing questions: (1) how many training and test data points are there? and (2) what percentage of the training tweets are of real disasters, and what percentage is not? Note that the meaning of each column is explained in the data description on Kaggle.

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

#1
```

```
print(len(X_Train),'training points.')
print(len(X_Test),'testing points')
#2
num_Tweets = X_Train['target'].value_counts()
print(round(num_Tweets[0]/len(X_Train)*100,2),"% are not of real Disasters.")
print(round(num_Tweets[1]/len(X_Train)*100,2),"% are of real Disasters.")
```

```
7613 training points.
3263 testing points
57.03 % are not of real Disasters.
42.97 % are of real Disasters.
```

- ## (b) Split the training data.

Since we do not know the correct values of labels in the test data, we will split the training data from Kaggle into a training set and a development set (a de- velopment set is a held out subset of the labeled data that we set aside in order to fine-tune models, before evaluating the best model(s) on the test 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. Throughout the rest of this problem we will keep these two sets fixed. The idea is that we will train different models on the training set, and compare their performance on the development set, in order to decide what to submit to Kaggle.

In [ ]:
```
#Splitting data set to 70% Training, 30% Development randomly
X_train,X_dev = train_test_split(X_Train,train_size=0.7)

#Combined split datasets
df = pd.concat([X_train,X_dev])
lenx = len(X_train)
len2x = len(df)
df = pd.concat([df,X_Test])
```

- ## (c) Preprocess the data.

Since the data consists of tweets, they may contain significant amounts of noise and unprocessed content. You may or may not want to do one or all of the following. Explain the reasons for each of your decision (why or why not).

- • Convert all the words to lowercase.
- • Lemmatize all the words (i.e., convert every word to its root so that all of “running,” “run,”

and “runs” are converted to “run” and and all of “good,” “well,” “better,” and “best” are converted to “good”; this is easily done using nltk.stem).

- • Strip punctuation.
- • Strip the stop words, e.g., “the”, “and”, “or”.
- • Strip @ and urls. (It’s Twitter.)
- • Something else? Tell us about it

In [ ]:
```
l = WordNetLemmatizer()
```

```
t = nltk.tokenize.WhitespaceTokenizer()
def lemat(text):
    st =  [l.lemmatize(x,pos='a') for x in t.tokenize(text)]
    num = 0
    for x in st:
        st[num] = x.translate(str.maketrans('','',string.punctuation))
        num+=1

    return " ".join(st)
```

In [ ]:
```
import enchant

t = nltk.tokenize.WhitespaceTokenizer()

stop_words = set(stopwords.words('english'))
eng_dict = enchant.Dict("en")

def stopw(text):
    out = [w for w in t.tokenize(text) if not w in stop_words]
    return " ".join(out)

def stripURLS(text):
    out2 = [l for l in t.tokenize(text) if not re.search(r'http\S+',l) or re.search(r'
    return " ".join(out2)

def isEnglish(text):
    out = [w for w in t.tokenize(text) if eng_dict.check(w)]
    return " ".join(out)
```

In [ ]:
```
#Convert to Lowercase to not confuse same letters that are differentiated by Case.
df["text"] = df['text'].str.lower()
#Lemmatize to simplify comparisons of words that have the same root.
#Punctuation to simplify comparisons as well, avoiding cases where punctuation at the
df["text"] = df["text"].apply(lemat)
#Removes Stopwords to directly target keywords.
df['text'] = df['text'].apply(stopw)
#Removes Urls of the form Https and WWW and @s from strings as they are irrelevant to
df['text'] = df['text'].apply(stripURLS)
#Remove non english words, to reduce size of bag of words and inocrrect comparisons.
df['text'] = df['text'].apply(isEnglish)

Y_train = X_train.loc[:,'target']
Y_dev = X_dev.loc[:,'target']


X_t = df.iloc[:lenx,:]
X_d = df.iloc[lenx:len2x,:]
X_test = df.iloc[len2x:,:]
```

- ## (d) Bag of words model.

The next task is to extract features in order to represent each tweet using the binary “bag of words” model, as discussed in lectures. The idea is to build a vocabulary of the words appearing in the dataset, and then to represent each tweet by a feature vector x whose length is the same

as the size of the vocabulary, where xi =1 if the i'th vocabulary word appears in that tweet, and xi =0 otherwise. In order to build the vocabulary, you should choose some threshold M, and only include words that appear in at least k different tweets; this is important both to avoid run-time and memory issues, and to avoid noisy/unreliable features that can hurt learning. Decide on an appropriate threshold M, and discuss how you made this decision. Then, build the bag of words feature vectors for both the training and development sets, and report the total number of features in these vectors.

In order to construct these features, we suggest using the CountVectorizer class in sklearn. A couple of notes on using this function: (1) you should set the option "binary=True" in order to ensure that the feature vectors are binary; and (2) you can use the option "min_df=M" in order to only include in the vocabulary words that appear in at least M different tweets. Finally,make sure you fit CountVectorizer only once on your training set and use the same instance to process both your training and development sets (don't refit it on your development set a second time).

Important: at this point you should only be constructing feature vectors for each data point using the text in the "text" column. You should ignore the "keyword" and "location" columns for now.

In [ ]:
```
#Threshold M Decision
from sklearn.feature_extraction.text import CountVectorizer

# Vectorize the training set and transform the development set.
# Threshold ranges, 5,10,15,20,50. Future optimality arises from values greater than 1
#The range of applicable words narrows too far with values greater than 20, therefore

count_vect = CountVectorizer(binary=True,min_df=15)
X_train = count_vect.fit_transform(X_t["text"]).toarray()
X_dev = count_vect.transform(X_d['text']).toarray()
names = count_vect.get_feature_names_out()

print(X_train.shape)
print(X_dev.shape)

colLen = len(df.columns)
```

```
(5329, 627)
(2284, 627)
```

- ## (e) Logistic regression.

In this question, we will be training logistic regression models using bag of words feature vectors obtained in part (d). We will use the F1-score as the evaluation metric.

Note that the F1-score, also known as F-score, is the harmonic mean of precision and recall. Recall that precision and recall are:

precision = $\frac{truepositives}{truepositives+falsepositives}$ $recall = \frac{truepositives}{truepositives+falsenegatives}$.

F1-score is the harmonic mean (or, see it as a weighted average) of precision and recall:

$$F1 = \frac{2}{precision^{-1}+recall^{-1}} = 2\frac{precision \cdot recall}{precision+recall}$$

We use F1-score because it gives a more comprehensive view of classifier performance than accuracy. For more information on this metric see F1-score.

We ask you to train the following classifiers. We suggest using the LogisticRegression implementation in sklearn.

1. Train a logistic regression model without regularization terms. You will notice that the default sklearn logistic regression utilizes L2 regularization. You can turn off L2 regularization by changing the penalty parameter. Report the F1 score in your training and in your development set. Comment on whether you observe any issues with overfitting or underfitting.

In [ ]:
```
regr = LogisticRegression(penalty='none',solver='saga') # No regularization term, Saga

regr.fit(X_train,Y_train)


Y_xdev_hat = pd.DataFrame()
Y_xtrain_hat = pd.DataFrame()

Y_xtrain_hat["accuracy"] = regr.predict(X_train)
Y_xdev_hat["accuracy"] = regr.predict(X_dev)

F1_xt_1 = f1_score(Y_train,Y_xtrain_hat)
F1_xd_1 = f1_score(Y_dev,Y_xdev_hat)

#The Development set is underfitting slightly compared to the training set.
print("F1 Score for Training Set: ",F1_xt_1)
print("F1 Score for Development Set: ", F1_xd_1)
```

```
F1 Score for Training Set:  0.7895454545454546
F1 Score for Development Set:  0.7190889370932755
```

```
c:\Users\hruda\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\line
ar_model\_sag.py:350: ConvergenceWarning: The max_iter was reached which means the co
ef_ did not converge
  warnings.warn(
```

2. Train a logistic regression model with L1 regularization. Sklearn provides some good examples for implementation. Report the performance on both the training and the development sets.

In [ ]:
```
regr = LogisticRegression(penalty='l1',solver='saga') #L1 regularization and Saga solv
regr.fit(X_train,Y_train)


Y_xdev_hat = pd.DataFrame()
Y_xtrain_hat = pd.DataFrame()
```

```
theta = pd.DataFrame()
theta['words'] = pd.DataFrame(data=regr.coef_[0],index=names)
Y_xtrain_hat["accuracy"] = regr.predict(X_train)
Y_xdev_hat["accuracy"] = regr.predict(X_dev)

F1_xt_2 = f1_score(Y_train,Y_xtrain_hat)
F1_xd_2 = f1_score(Y_dev,Y_xdev_hat)

#The Development set is underfitting slightly compared to the training set.
print("F1 Score for Training Set: ",F1_xt_2)
print("F1 Score for Development Set: ", F1_xd_2)
```

```
F1 Score for Training Set:  0.7785016286644951
F1 Score for Development Set:  0.7246376811594204
```

```
c:\Users\hruda\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\line
ar_model\_sag.py:350: ConvergenceWarning: The max_iter was reached which means the co
ef_ did not converge
  warnings.warn(
```

3. Similarly, train a logistic regression model with L2 regularization. Report the perfor-mance on the training and the development sets.

In [ ]:
```
regr = LogisticRegression(penalty='l2',solver='saga') #L2 Regularization and Saga solv
regr.fit(X_train,Y_train)


Y_xdev_hat = pd.DataFrame()
Y_xtrain_hat = pd.DataFrame()

Y_xtrain_hat["accuracy"] = regr.predict(X_train)
Y_xdev_hat["accuracy"] = regr.predict(X_dev)

F1_xt_3 = f1_score(Y_train,Y_xtrain_hat)
F1_xd_3 = f1_score(Y_dev,Y_xdev_hat)

#The Development set is underfitting slightly compared to the training set.
print("F1 Score for Training Set: ",F1_xt_3)
print("F1 Score for Development Set: ", F1_xd_3)
```

```
F1 Score for Training Set:  0.7809965237543453
F1 Score for Development Set:  0.7225592939878654
```

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

In [ ]:
```
#Of the three classifiers, having no penalty performed the best but marginally. I obse
#The values I got for training and development sets respectively for None, L1 and L2 r
print("No Regularization Training and Development Set: ", F1_xt_1,F1_xd_1)
print("L1 Regularization Training and Development Set: ", F1_xt_2,F1_xd_2)
print("L2 Regularication Training and Development Set: ", F1_xt_3,F1_xd_3)
```

```
No Regularization Training and Development Set:  0.7895454545454546 0.719088937093275
5
L1 Regularization Training and Development Set:  0.7785016286644951 0.724637681159420
4
L2 Regularication Training and Development Set:  0.7809965237543453 0.722559293987865
4
```

5. Inspect the weight vector of the classifier with L1 regularization (in other words, look at the θ you got after training). You can access the weight vector of the trained model using the coef_ attribute of a LogisticRegression instance. What are the most important words for deciding whether a tweet is about a real disaster or not?

In [ ]:
```
#print(theta.columns)
theta = theta.sort_values(by='words',ascending=False)
print("The 5 most important words weighted by the algorithm deciding if a tweet is abo
print(theta.loc[:,'words'][:5])
#5 Most important words deciding if a tweet is about a real disaster
```

```
The 5 most important words weighted by the algorithm deciding if a tweet is about a r
eal disaster are:
bombing       3.538424
debris        3.524337
airport       3.178497
typhoon       3.074675
derailment    3.031157
Name: words, dtype: float64
```

- ## (f) Bernoulli Naive Bayes.

Implement a Bernoulli Naive Bayes classifier to predict the probability of whether each tweet is about a real disaster. Train this classifier on the training set, and report its F1-score on the development set.

Important: For this question you should implement the classifier yourself similar to what was shown in class, without using any existing machine learning libraries such as sklearn. You may only use basic libraries such as numpy.

As you work on this problem, you may find that some words in the vocabulary occur in the development set but are not in the training set. As a result, the standard Naive Bayes model learns to assign them an occurrence probability of zero, which becomes problematic when we observe this "zero probability" event on our development set.

The solution to this problem is a form of regularization called Laplace smoothing or additive smoothing. The idea is to use "pseudo-counts", i.e. to increment the number of times we have seen each word or document by some number of "virtual" occurrences α. Thus, the Naive Bayes model will behave as if every word or document has been seen at least αtimes.

More formally, the ψjk parameter of Bernoulli Naive Bayes is the probability of observing word j within class k. Its normal maximum likelihood estimate is ψjk =njk nk ,

where nk is the number of documents of class k and njk is the number of documents of class k that contain word j. In Laplace smoothing, we increment each counter njk by α(thus we count each word an extra αtimes), and the resulting estimate for ψjk becomes:

ψjk = njk +α nk +2α. It's normal to take α=1.

In [ ]:
```
#Bernoulli Naive Bayes Implementation
#Followed the inclass example exactly to implement BNB
n = X_train.shape[0]
d = X_train.shape[1]
K = 2 #Binary classes = 2
a = 1 #Alpha for smoothing
psis = np.zeros([K,d])
phis = np.zeros([K])

for k in range(K):
    X_l = X_train[Y_train == k]
    #Laplace Smoothing to avoid 0 probability division
    psis[k] = (np.sum(X_l, axis=0)+a)/(np.sum(X_l)+2*a)
    phis[k] = X_l.shape[0] / float(n)
```

In [ ]:
```
#Naive_Bayes Prediction function from Lecture
def nb_predictions(x, psis, phis):
    n,d = x.shape
    x = np.reshape(x, (1, n, d))
    psis = np.reshape(psis, (K, 1, d))

    psis = psis.clip(1e-14, 1-1e-14)

    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])

idx, logpyx = nb_predictions(X_dev, psis, phis)
print("Proportions:", phis)
F1_BNB = f1_score(Y_dev,idx)

print("F1 Score of Development set using Bernoulli Naive Bayes: ",F1_BNB)
```

```
Proportions: [0.56708576 0.43291424]
F1 Score of Development set using Bernoulli Naive Bayes:  0.7161572052401747
```

- ## (g) Model comparison.

You just implemented a generative classifier and a discriminative classifier. Reflect on the following:

- • 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.

- • 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.

1. The Logistic Regression Model outperforms the Naive Bayes Model in prediciting whether a tweet is of a real disaster. The difference between the F1 scores is minimal, but Logistic outperforms Naive-Bayes. Logistic Regression and Naive Bayes are listed below. The pros and cons of using generative vs discriminative models are numerous. The pros of using the discriminative model, Logistic Regression, is that we get a score for each instance of a word based on frequency in the twitter text dataset. Furthremore we are able to interpret the models value as the conditional probability of finding y given x. Cons with the discriminative appraoch are that words that are poorly classifying data in the context of missclassifying a point. Using the generative model, Naive Bayes, for this dataset of text classification is good because we are able to filter out a lot of unncessary spam in the text. The pros of this method is generating values and dealing with other texts. The main con is that given an outlier in the dataset, the data will be skewed significantly.

In [ ]:
```
#Logistic Regression L2, Naive Bayes
print("Logistic Regression F1 Score:", F1_xd_3)
print("Bernoulli Naive Bayes F1 Score: ",F1_BNB)
```

```
Logistic Regression F1 Score: 0.7225592939878654
Bernoulli Naive Bayes F1 Score:  0.7161572052401747
```

2. The assumptions that naive bayes makes are that it assumes every event is independent, so words in the text are independent of other words. This leads to over and under confidence in the accuracy of the models. The values for each feature in Naive bayes is assumed to be Binary and will not work with non-binary data. Comparitvely a logistic regressions assumption is that there is little correlation between explanatory variables. They can lead the model to incorect interpretation. Furthermore, Logistic regression assumes independent variables are linearly related with log odds. With the assumptions taken by Bernoulli Naive Bayes it is a valid classifier for Natural Language Texts but it is not efficient as text independancy could be a limiting factor in emphasizing the connection between texts.

• (h) N-gram model.

The N-gram model is similar to the bag of words model, but instead of using individual words we use N-grams, which are contiguous sequences of words. For example, using N =2, we would says that the text "Alice fell down the rabbit hole" consists of the sequence of 2-grams: ["Alice fell", "fell down", "down the", "the rabbit", "rabbit hole"], and the following sequence of 1-grams: ["Alice", "fell", "down", "the", "rabbit", "hole"]. All eleven of these symbols may be included in the vocabulary, and the feature vector x is defined according to xi = 1 if the i'th

vocabulary symbol occurs in the tweet, and xi = 0 otherwise. Using N =2, construct feature representations of the tweets in the training and development tweets. Again, you should choose a threshold M, and only include symbols in the vocabulary that occur in at least M different tweets in the training set. Discuss how you chose the threshold M, and report the total number of 1-grams and 2-grams in your vocabulary. In addition, take 10 2-grams from your vocabulary, and print them out.

Then, implement a logistic regression and a Bernoulli classifier to train on 2-grams. You may reuse the code in (e) and (f). You may also choose to use or not use a regularization term, depending on what you got from (e). Report your results on training and development set.Do these results differ significantly from those using the bag of words model? Discuss what this implies about the task.

Again, we suggest using CountVectorizer to construct these features. In order to include both 1-gram and 2-gram features, you can set ngram_range=(1,2). Note also that in this case, since there are probably many different 2-grams in the dataset, it is especially important carefully set min_df in order to avoid run-time and memory issues.

In [ ]:
```
#N-Gram Model

#Minimum number of words is determined manually using iterative processing to determin
#the Bag of Words model and N-gram model with 2-grams present in the new word count. M
count_vect2 = CountVectorizer(ngram_range=(1,2),binary=True,min_df=15)
Xg_train = count_vect2.fit_transform(X_t["text"]).toarray()
Xg_dev = count_vect2.transform(X_d['text']).toarray()
print("Number of 1-gram and 2-gram phrases in N-Gram Bag: ", Xg_train.shape[1])

#10 2 gram from vocabulary to find specifically the 2-gram words.
model = CountVectorizer(ngram_range=(2,2),binary=True,min_df=15)
m = model.fit_transform(X_t['text'])
names2 = model.get_feature_names_out()
print("10, 2-gram words from the N-Gram Bag:")
print(names2[5:15])



#Logistic Regression due to being marginally superior to Naive-Bayes based on my prior
#L2 Regularization is used as the difference between all were minimal.
regr = LogisticRegression(penalty='l2',solver='saga')
regr.fit(Xg_train,Y_train)


Y_xdev_hat = pd.DataFrame()
Y_xtrain_hat = pd.DataFrame()

Y_xtrain_hat["accuracy"] = regr.predict(Xg_train)
Y_xdev_hat["accuracy"] = regr.predict(Xg_dev)

F1_xt = f1_score(Y_train,Y_xtrain_hat)
F1_xd = f1_score(Y_dev,Y_xdev_hat)

print("Logistic Training Score and Dev F1 Score:",F1_xt,F1_xd)
```

```
#Bernoulli Naive Bayes
n2 = Xg_train.shape[0]
d2 = Xg_train.shape[1]
K = 2 #Binary classes = 2
a = 1 #Alpha for smoothing

psis2 = np.zeros([K,d2])
phis2 = np.zeros([K])

for k in range(K):
    X_k = Xg_train[Y_train == k]

    psis2[k] = (np.sum(X_k, axis=0)+a)/(np.sum(X_k)+2*a)
    phis2[k] = X_k.shape[0] / float(n)

idx2, logpyx2 = nb_predictions(Xg_dev, psis2, phis2)
idx3, logpyx3 = nb_predictions(Xg_train,psis2, phis2)
#print("Proportions:", phis)
F1_BNB2 = f1_score(Y_dev,idx2)
F1_BNB3 = f1_score(Y_train,idx3)
print("Bernoulli Naive Bayes Training and Dev F1 Score: ",F1_BNB3,F1_BNB2)

#These results are very similar to my values for Logistic Regression and Bernoulli Nai
#This implies that the task is accomplishable by looking at both individualist words a
#Furthermore the threshold is very important to understanding how many 2-gram models a
#of 2-gram phrases but overfits the data.
```

```
Number of 1-gram and 2-gram phrases in N-Gram Bag:  704
10, 2-gram words from the N-Gram Bag:
['army trench' 'big projected' 'black hat' 'body bag' 'body bagging'
 'body bags' 'bomb turkey' 'bomber detonated' 'bomber kills'
 'burning buildings']
Logistic Training Score and Dev F1 Score: 0.783495595734817 0.7240618101545254
Bernoulli Naive Bayes Training and Dev F1 Score:  0.7400468384074942 0.70819304152637
49
```

- ## (i) Determine performance with the test set

Re-build your feature vectors and re-train your preferred classifier (either bag of word or n-gram using either logistic regression or Bernoulli naive bayes) using the entire Kaggle training data (i.e. using all of the data in both the training and development sets). Then, test it on the Kaggle test data. Submit your results to Kaggle, and report the resulting F1-score on the test data, as reported by Kaggle. Was this lower or higher than you expected? Discuss why it might be lower or higher than your expectation.

In [ ]:
```
#Refitting Feature Vectors, Classifier using Bag Of Words using entire Training Data s
count_vect3 = CountVectorizer(binary=True,min_df=15)
X_TrainBOW = count_vect2.fit_transform(X_Train["text"]).toarray()
X_TestBOW = count_vect2.transform(X_test["text"]).toarray()
Y_Train = X_Train.loc[:,"target"]

regr4 = LogisticRegression(penalty='l2',solver='saga')
regr4.fit(X_TrainBOW,Y_Train)
Y_hat = pd.DataFrame()
```

```
Y_hat['id'] = X_test.loc[:,"id"]
Y_hat["target"] = regr4.predict(X_TestBOW)

Y_hat.to_csv(path_or_buf="HW2.csv", sep =',',index=False)

#Insert Kaggle Score below
print("F1 Score according to Kaggle: 0.74655")
#This value was slightly higher than my expectations, given my previous L2 function wa
#difference could be attributed to minor variations in how the difference between the
#than the logistic score, taking the differences in the combined data I believe this a
```

```
F1 Score according to Kaggle: 0.74655
```