Xing Yi Chan
R00183768

Part 1

In the first part of the assignment, the training and development dataset will first undergo the data preprocessing steps. In natural language processing, the process of preparing text data is called text preprocessing. It is an important step for NLP tasks as it transforms the data into a more digestible form so that machine learning models can perform better. Below is the workflow of text preprocessing for this assignment.

- Tokenization / segmentation
- Removing diacritics
- Removing punctuation
- Removing stopwords
- Stemming / lemmatization

Preparing dataset

Before performing text preprocessing, import and read the data into dataframes. The dataset Madar Corpus 26 train and dev set were used. The training data is named as training_data and the dev data is named as testing_data.

```
# import training and testing data
training_data = pd.read_csv('/content/drive/My Drive/NLP/dataset/MADAR-Corpus-26-train.tsv', sep= '\t',
                            names=['Sentence', 'City'])
testing_data = pd.read_csv('/content/drive/My Drive/NLP/dataset/MADAR-Corpus-26-dev.tsv', sep= '\t',
                           names=['Sentence', 'City'])

# seperate training and testing data and target values
trainX = training_data['Sentence']
trainY = training_data['City']
testX = testing_data['Sentence']
testY = testing_data['City']

# perform lable encoding on both training and testing target values
enc = LabelEncoder()
trainY = enc.fit_transform(trainY)
testY = enc.fit_transform(testY)
```

Both training data and testing data are then separated into data and labels. Both labels will undergo Label Encoder() function to convert the values into integer values.

Tokenization / segmentation

It is a process of breaking up a stream of text into words, phrases, symbols or other meaningful elements called as tokens. In this assignment, a library called farasa is used to perform tokenization. Farasa is an arabic NLP toolkit that can perform tasks like segmentation, stemming, Name Entity Recognition(NER), Part Of Speech tagging(POS) tagging and diacritics removal.

```
# segmentation / tokenization
def tokenization(train, test):
    segmenter = FarasaSegmenter()
    train_output = train.apply(lambda x: segmenter.segment(x))
    test_output = test.apply(lambda x: segmenter.segment(x))
    print()
    print('Completed segmentation......')

    return train_output, test_output

trainX, testX = tokenization(trainX, testX)
```

The data were pushed through the FarasaSegmenter() function by farasa to perform segmentation / tokenization. As all the functions in Farasa only accept string input, we can only the function by looping through the dataset. Converting the whole dataset into strings and converting them back into dataframes for classification will reduce the accuracy score.

Removing diacritics

After tokenization / segmentation, the text will undergo diacritics removal. The word diacritic in arabic refers to all the markings that can appear above and below letters to alter the pronunciation.

```
# remove diacritics
def remove_diacritics(train, test):
    diacritizer = FarasaDiacritizer()
    train_output = train.apply(lambda x: diacritizer.diacritize(x))
    test_output = test.apply(lambda x: diacritizer.diacritize(x))
    print()
    print('Completed diacritization......')

    return train_output, test_output

trainX, testX = remove_diacritics(trainX, testX)
```

A function called remove_diacritics() is created and it will apply FarasaDiacritizer() from farasa to the data to remove diacritics and return the processed training and testing data.

Removing punctuation

Below is the code for removing punctuation from the text.

```
# remove punctuation
def remove_punctuation(text):
    output = re.sub(r'[^\w\s]','', text)
    output = re.sub('[؟.]','', text)
    return output

for i in range(len(trainX)):
    trainX[i] = remove_punctuation(trainX[i])

for j in range(len(testX)):
    testX[j] = remove_punctuation(testX[j])
```

A function named remove_punctuation is created to remove the punctuation in the text. It will remove the common punctuations first and move on to removing the common punctuations in the text. Loop through the training data and testing data to apply to function.

Removing stopwords

After removing punctuations, we will move on to removing the stopwords in the text.

```
# remove stopwords
from nltk.corpus import stopwords
nltk.download('stopwords')
stopwords = stopwords.words('arabic')

def remove_stopwords(text):
    words = [w for w in text if w not in stopwords]
    return "".join(words)

for i in range(len(trainX)):
    trainX[i] = remove_stopwords(trainX[i])

for j in range(len(testX)):
    testX[j] = remove_stopwords(testX[j])
```

First, download the arabic stopwords from the nltk corpus. A function is created to check every word in the input sentence. If the word is in the list of stopwords, it will be removed.

Stemming / lemmatization

The aim of these two processes are the same, which is to reduce the words to its common base or root form. However, the difference is that stemmer operates on a single word without having knowledge of the context. For example, stemming will reduce the word studies to studi while lemmatization will reduce it to study.

```
# stemming
def stemming(train, test):
    stemmer = FarasaStemmer()
    train_output = train.apply(lambda x: stemmer.stem(x))
    test_output = test.apply(lambda x: stemmer.stem(x))
    print()
    print('Completed stemming...')

    return train_output, test_output

trainX, testX = stemming(trainX, testX)
```

A function called stemming() is created to perform stemming on the data. The function will apply FarasaStemmer(), a stemming function by farasa to stem the text and return the processed training and testing data.

Converting text to integers

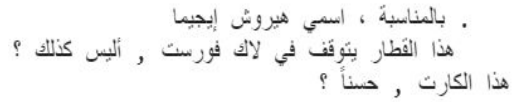
As all classification models only take in numerical values as input, the data has to be transformed into numerical values.

```
# create tf-idf vectors
vectorizer = TfidfVectorizer(max_features=1500, analyzer='word', ngram_range=(1, 2))
trainX_tfidf = vectorizer.fit_transform(trainX)
testX_tfidf = vectorizer.fit_transform(testX)
```

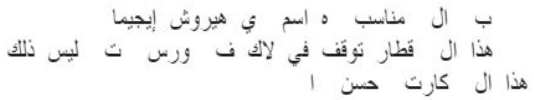
TfidfVectorizer() function is used to transform the texts into feature vectors which can be used as input for the models. For this task, the TF_IDF vectorized features are extracted based on word level with bi-grams. The reason for using bi-grams is because after comparing n-grams that range from 1 to 5, bi-gram returns the best results.

Before and after text preprocessing

Images below show a small part of the data before and after it has been processed.



Before



After

Part 2

After text preprocessing, the second part of the assignment is to perform classification. Several machine learning classification methods will be used to figure out which model will fit better and properly capture the relationships between the points and the labels. Below are the methods that are used in this task.

- Naive Bayes classifier
- Support Vector Machine classifier
- k-Nearest Neighbour classifier
- Random Forest classifier

Naive Bayes Classifier

```
1 # naive bayes classifier
2 nb = MultinomialNB()
3 nb.fit(trainX_tfidf, trainY)
4 print('Accuracy: ', nb.score(testX_tfidf, testY)*100)
5
6 y_pred = nb.predict(testX_tfidf)
```

```
Accuracy:  71.75
```

Naive Bayes classifier is a classifier which works based on Bayes's theorem which uses the probabilities of events. It is widely used in text classification as it requires less training time and less training data. To apply Naive Bayes classifier, we need to first call the classifier provided by Scikit-Learn. Next, push the training and testing data through the model and get the accuracy score of the model.

Support Vector Machine Classifier

Similar to Naive Bayes classifier, Support Vector Machine classifier is also a classifier which is commonly used for text classification problems.

```
# svm classifier
svm = SVC(kernel='linear')
svm.fit(trainX_tfidf, trainY)
print('Accuracy: ', svm.score(testX_tfidf, testY)*100)
```

```
Accuracy:  68.03
```

As SVM is sensitive to the parameter used, selecting a suitable kernel is essential. For text classification, linear kernel is the most suitable kernel to use as most of the text classification problems are linearly separable. Moreover, linear kernel performs better when the data has a lot of features and the number of instances and features are large in text classification.

Similar to Naive Bayes classifier, we need to call the SVM function and define the kernel that will be used. Next, push the training data and test data through the model and the model will return it's accuracy score.

k-Nearest Neighbour Classifier

k-Nearest Neighbour classifier is used to classify items by finding the k nearest matches in the training data and used the label of the closest matches to make predictions. This classifier uses euclidean distance to find the closest match.

```
# k-nn classifier
knn = KNeighborsClassifier()
knn.fit(trainX_tfidf, trainY)
print('Accuracy: ', knn.score(testX_tfidf, testY)*100)
```

```
Accuracy:  61.52
```

First, call the KNeighboursClassifier() function and push the training and testing data through the data to get the accuracy score of the model. By default, the k value is 3 and after comparing the performance of the classifier with different k values, the classifier performs better when the k value is 3.

Random Forest Classifier

```
# random forest classifier
rf = RandomForestClassifier()
rf.fit(trainX_tfidf, trainY)
print('Accuracy: ', rf.score(testX_tfidf, testY)*100)
```

```
Accuracy:  68.75
```

Similar to the other classifiers, we will first call the RandomForestClassifier() function. Next, feed the model with training and testing data. Lastly, print out the accuracy score of the classifier. As Random Forest classifier is not sensitive to the parameter used, the parameters will remain the same as their default state.

Part 3

Below is a table containing the results of the 4 different classifiers used.

Classifier	Accuracy (%)
Naive Bayes Classifier	71.75
Support Vector Machine Classifier	68.03
k-Nearest Neighbour Classifier	61.52
Random Forest Classifier	68.75

From the results above, it can be concluded that Naive Bayes classifier performs the best and is the most suitable classifier for this task. In contrast, the k-NN classifier gave the weakest performance among the four classifiers. We can say that Naive Bayes classifier suits the data the most.

Evaluation using MADAR-DID-SCORER

```
OVERALL SCORES:
MACRO AVERAGE PRECISION SCORE: 55.95 %
MACRO AVERAGE RECALL SCORE: 55.95 %
MACRO AVERAGE F1 SCORE: 55.95 %
OVERALL ACCURACY: 71.75 %
```

Above is the results after the predicted output and the test label are being evaluated by the MADAR-DID-SCORER. Since the Naive Bayes classifier performed better than the other classifiers, the prediction output was generated by the Naive Bayes classifier. From the results, we can see that the model achieved an overall accuracy of **71.75%**. Moreover, the results show that 55.95% of the positive patterns are correctly classified and 55.95% of positive patterns are correctly predicted from the total predicted patterns in a positive class.