Students are expected to:

1- Implement natural language processing solutions to build accurate NLP classifiers in order to predict the topic of a given test example.

2- Implement text generation model to generate text for a chosen topic and then use the generated data to test the top performing NLP classifier.

The report is accomplished from the findings on the aspects below:

1.

During the preprocessing process, this program gets a set for training with 40000 rows, a testing set with 4000 rows, and a vocabulary of 49750 tokens. The set for training is split into a training set and a validation set.

After that, this program adopts TfidfVectorizer to extract texts' features. TfidfVectorizer is a bag-of-words vectorization technique. It adopts unigrams here and transforms input texts into a matrix of TF-IDF features. Each word in input texts is expressed as {the number of times a word appears in one text / [log (number of texts+1/the number of texts that contains this word+1) +1]}. Each input text is normalized by the Euclidean norm. Therefore, the set for training is encoded as a 40000*49750 matrix while the testing set is encoded as a 4000*49750 matrix.

TfidfVectorizer's major advantage is that the encoded text data can be directly passed to most machine learning models. Moreover, it works better than the CountVectorizer to reflect one word's frequency in that it utilizes IDF -- 1/ [log (number of texts+1/ the number of texts that contains this word+1) +1] to

diminish the weight of terms that only occur very frequently in one or few texts. In general, it allows people to calculate two encoded texts' similarity efficiently.

However, TfidfVectorizer ignores the order of tokens in texts. The encoded words in each row of the matrix are not ordered by their original order in texts.

This program also applies Word2vec to extract texts' features. Word2vec is one kind of pretrained word embeddings. It embeds words on a n-dimension plane (n>=2), so words turn to be points on the plane. This technique can extract semantic relationships between two words. For instance, the vector "goes from cat to dog" may equal to the one "goes from tiger to wolf", as both resemble the vector "goes from canine to feline". This program applies "CBOW" to compute word representation and each input word will be represented by a vector of length 100. Hence, a ((vocabulary size+1) * dimensionality of the word vectors) embedding matrix (Figure 1) can be developed to represent all input words. For Random Forest, a sentence vector is generated for each input text using a weighted average of word vectors. The matrix consisting of sentences vectors can be the input to Random Forest. For the Deep Learning model, this program tokenizes and pads texts through Tokenizer() and pad_sequences() as the input to model and the embedding matrix is passed to the Embedding layer as weights matrix where each encoded word can map to the correct vector.

Although Word2vec helps remain original order and semantic relationships of texts, it is more time-consuming than TfidfVectorizer.

```
# get vectors in the right order. vocab_size = len(tokenizer.word_index) + 1 was defined above.
embedding_vectors = get_weight_matrix(raw_embedding, tokenizer.word_index, vocab_size)
# do a sanity check
pd.DataFrame(embedding_vectors)
```

	0	1	2	3	4	5	6	7	8	9	...	90	91	92	93
0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.000000	0.000000	0.000000
1	-0.500554	0.366605	0.465124	-0.029959	-0.331847	-0.152977	-0.265990	1.139357	0.274801	-0.147491	...	0.750396	-0.800082	-0.350218	0.114742
2	0.023091	0.347684	2.160541	0.413702	1.167261	-0.046568	1.129993	-0.540587	0.309240	-1.096298	...	-0.181062	0.019054	0.187391	0.409127
3	1.276698	-1.742334	0.184628	1.155431	0.092625	-0.059228	0.216021	1.541625	-0.384224	-1.017697	...	1.387728	0.033509	-1.090336	-0.320322
4	-0.372234	0.462172	1.083369	0.453856	0.730952	-0.212748	-0.880524	0.173341	0.322810	-0.573779	...	0.643976	-0.367109	0.580967	0.269638
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
49746	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.000000	0.000000	0.000000
49747	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.000000	0.000000	0.000000
49748	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.000000	0.000000	0.000000
49749	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.000000	0.000000	0.000000
49750	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.000000	0.000000	0.000000

49751 rows × 100 columns

Figure 1 Embedding Matrix

2.

In both Random Forest models, GridSearchCV is utilized to find the best parameter. The best estimator of Random Forest (with Tfidf) is neither overfitting nor underfitting, while that of Random Forest (with Word2vec) is slightly overfitting. However, they share a similar testing accuracy (Figure 2&3). There may be little difference between preparing text data with TfidfVectorizer and with Word2vec for a standard Machine Learning algorithm.

```
# show the accuracy score on the testing dataset
acc_t = accuracy_score(y_test, clfRF_t.predict(X_test_t)) * 100

print('Testing Accuracy: %0.2f%%' % acc_t)

testing_acc = pd.DataFrame(columns = ["Classifier", "Testing accuracy(percent)"])
testing_acc.loc[0] = ["Random Forest(with Tfidf)", acc_t]
```

```
Testing Accuracy: 83.80%
```

Figure 2 Testing Accuracy of Random Forest (with Tfidf)

```
# show the accuracy score on the testing dataset
acc_w = accuracy_score(y_test, clfRF_w.predict(X_test_w_RF)) * 100
print('Test Accuracy: %0.2f%%' % acc_w)

testing_acc.loc[1] = ["Random Forest(with Word2vec)", acc_w]
```

```
Test Accuracy: 84.17%
```

Figure 3 Testing Accuracy of Random Forest (with Word2vec)

3.

This program builds a Deep learning model with 10 layers (Figure 4&5).

The Embedding layer adopts the embedding matrix prepared in step 1 as weights matrix. This layer can look up word index in the weights matrix and returns the corresponding word vectors. The input to the layer is a 2D tensor with the shape of (samples, sequence_length) and returns a 3D tensor with the shape of (samples, sequence_length, embedding_dimensionality). To prevent overfitting, a dropout layer is added after that.

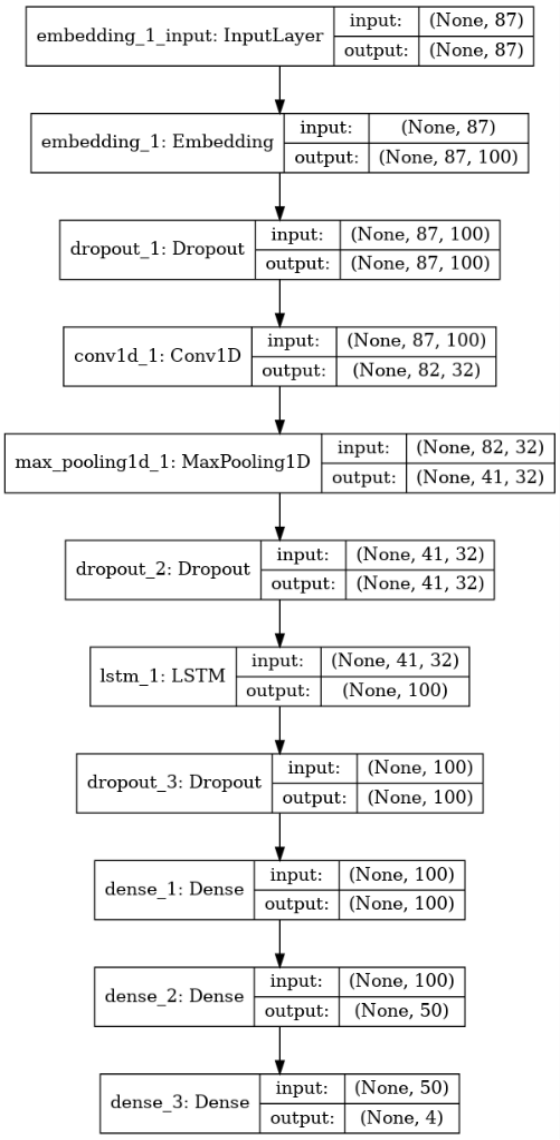

Figure 4 Architecture of the Deep Learning model

Then, a CNN layer is added. The input is a 3D tensor: (samples, sequence_length, embedding_dimensionality). The kernel size is 6, so the convolution works by sliding a window of size 1*6 on each input sample of length 87 to learn samples' features. The number of filters is 32, which is the output_depth as each filter encodes one specific aspect of the input. Each filter saves a feature map of 1*82 from the convolution on one sample (feature maps are not 1*87 because of border effects). The output is a 3D tensor of shape (samples, the length of feature map, output_depth). The kernel size 6 is neither too small to be inefficient nor too large to lose many features. The

filters' number 32 is also suitable for the input length 100. Next comes a MaxPooling1D layer, which downsamples the feature maps by a factor of 2 to make sliding windows contain more information from the initial input and prevent overfitting. After that, a dropout layer is added.

A LSTM layer is added followingly to learn each inputs' labels based on their context. The 100 units are set for learning and flattening 41*32 coefficients from CNN per sample. A dropout layer is added again.

Finally, fully connected layers are added to learn 100 neurons progressively. The output layer consists of one neuron for each label (4 in all) and uses a softmax activation function to give a probability distribution of labels.

For the multi-label classification, the loss function can be "categorical_crossentropy". Additionally, the optimizer "adam" with high efficiency is utilized and the metrics of this model should be "accuracy" as the metrics of this project is "accuracy". The model is trained with 10 epochs, which is enough here.

```
def define_model(vocab_size, max_length):
    model = Sequential()
    # use Pre-trained Embedding (Word2vec)
    model.add(Embedding(vocab_size, 100, weights=[embedding_vectors], input_length=max_length, trainable=False))
    model.add(Dropout(0.2))
    model.add(Conv1D(32, 6, activation='relu'))
    model.add(MaxPooling1D())
    model.add(Dropout(0.2))
    model.add(LSTM(100))
    model.add(Dropout(0.2))
    model.add(Dense(100, activation='relu'))
    model.add(Dense(50, activation='relu'))
    model.add(Dense(4, activation='softmax'))

    # compile network
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    # summarize defined model
    model.summary()
    plot_model(model, to_file='model_dl.png', show_shapes=True)
    return model
```

Figure 5

4.

As every model takes much time to run once and this is easy to compare, this program subtracts the time.time() at the beginning from the time.time() in the end to get execution time in each model-fitting process (For instance: Figure 6).

```
start_time_RF_t =time.time()

clfRF_t = clfRF(X_train_t, y_train)

trainingtime = pd.DataFrame(columns = ["Classifier", "Training Time(Seconds)"])
trainingtime.loc[0] = ["Random Forest(with Tfidf)", round((time.time()-start_time_RF_t), 2)]
```

Figure 6

As is shown in Table 1, Random Forest(with Word2vec)'s running time is about 250% more than that of other models (962.47 seconds/6*3*5 = 90 combinations of parameters). Running time of Random Forest with Tfidf (331.36 seconds/6*3*5 = 90 combinations of parameters) is close to that of the Deep Learning model with Word2vec (377.97 seconds/10 epochs).

Table 1 Trainingtime

```
trainingtime = trainingtime.sort_values(by=['Training Time(Seconds)'])
trainingtime.index = trainingtime.index + 1
trainingtime
```

	Classifier	Training Time(Seconds)
3	Deep Learning Model	331.36
1	Random Forest(with Tfidf)	377.97
2	Random Forest(with Word2vec)	962.47

In terms of testing accuracy, all models perform well (Table 2). The Deep Learning model (88.17%) performs slightly better than two Random Forest models (around 83%-84%).

It is worthy to pointing out that Tfidf encoding is adopted to 49750 words

while Word2vec is only applied to 14168 words (min_counts = 5 set in Word2vec filters the words occurring less than 5 times). However, Random Forest with Word2vec still takes long time to run while the Deep Learning model takes much less time and gets a better result. Therefore, the Deep Learning model is the best one among three models.

Table 2 Testing accuracy

```
testing_acc = testing_acc.sort_values(by=['Testing accuracy(percent)'])
testing_acc.index = testing_acc.index + 1
testing_acc
```

	Classifier	Testing accuracy(percent)
1	Random Forest(with Tfidf)	83.800000
2	Random Forest(with Word2vec)	84.175000
3	Deep Learning Model	88.174999

5.

To prepare text data, the program loads the training data which has been cleaned in the preprocessing of step 1 (saved in ‘train_clean.csv’) and extract texts of the topic ‘Sports’. As the maximum length of the texts is not very large, there is no need to truncate any text.

The language model has an Embedding layer to learn the representation of words and a LSTM layer to learn to predict words based on their context. As the ‘One-word-in, one-word-out’ model may cause ambiguity, this program picks 2 as the length of context and transforms the tokens of each text into sequences of 2 input words and 1 output word. Then, this program encodes and pads these sequences with tokenizer and pad_sequences(), and split into input and output to train the model.

The randint() function can select a random line of the training sequences of text as a input text when the model works. In the self-defined generate_seq() function (Figure 7), input texts are encoded by the same tokenizer as used during training and pad_sequences(), and then be passed to the model. The predict_classes() function predicts the output – the next word of input. As the expected length of one generated sample is 87, which is the maximum length of “article text” among training data, the process above is repeated 87 times to build up a sample with 87 words. In each loop, output word is added to next loop’s input text and pad_sequences() removes the first word of input text to build a new input of length 2 when next loop begins. Through the self-defined generate_seq_samples() function, 100 randomly selected input text can be fed into generate_seq() respectively and thus 100 samples can be generated. In general, the model calculates the probability distribution of output words and compares to return the index of the word with the highest probability. The predicted word will be fed in as input in turn generate the next word.

```
# generate a sequence from a language model
def generate_seq(model, tokenizer, seq_length, seed_text, n_words):
    result = list()
    in_text = seed_text
    # generate a fixed number of words
    for _ in range(n_words):
        # encode the text as integer
        encoded = tokenizer.texts_to_sequences([in_text])[0]
        # truncate sequences to a fixed length
        encoded = pad_sequences([encoded], maxlen = seq_length, truncating = 'pre')
        # predict probabilities for each word
        yhat = model.predict_classes(encoded, verbose = 0)
        # map predicted word index to word
        out_word = ''
        for word, index in tokenizer.word_index.items():
            if index == yhat:
                out_word = word
                break
            # append to input
        in_text += ' ' + out_word
        result.append(out_word)
    return ' '.join(result)


def generate_seq_samples(model, tokenizer, seq_length, n_words, seed_text_list):
    samples = []
    for seed_text in seed_text_list:
        generated = generate_seq(model, tokenizer, seq_length, seed_text, n_words)
        samples.append(generated)
    return samples


# load cleaned text sequences
in_filename = 'train_Sports_seq.txt'
doc = load_doc(in_filename)
lines = doc.split('\n')
# seq_length = len(X)
seq_length = len(lines[0].split()) - 1
# load the model
model = load_model('model_tg.h5')
# load the tokenizer
tokenizer = load(open('tokenizer.pkl', 'rb'))

samples_count = 100
seed_text_list = []
for _ in range(samples_count):
    # select a seed text from lines
    seed_text = lines[randint(0, len(lines)-1)]
    seed_text_list.append(seed_text)
seed_text_list

# generate new text: generate 100 new words
samples_100 = generate_seq_samples(model, tokenizer, seq_length, 87, seed_text_list)
```

Figure 7 The process of generating 100 samples

6.

To test performance of the three models developed in step 2 and step3, these 100 samples are experienced the corresponding feature extraction techniques adopted in step 1 and then be put into each model. All three models get a 100% classification accuracy (Table 3). However, among multiple trials behind, sometimes some of the models cannot get a 100% accuracy, especially the two random forest models. This results from the stochastic nature of random forest and neural networks. Moreover, the number of samples may be too small to get a stable accuracy. However, all models can still get an accuracy around 99% or 98% when they cannot correctly predict all samples' topic. It can be concluded that all models are acceptable.

Table 3 Testing accuracy on 100 generated samples

```
testing_acc = testing_acc.sort_values(by=['Testing accuracy(percent)'])
testing_acc.index = testing_acc.index + 1
testing_acc
```

	Classifier	Testing accuracy(percent)
1	Random Forest(with Tfidf)	100.0
2	Random Forest(with word2vec)	100.0
3	Deep Learning Model	100.0

Reference List

François Chollet (2018) Deep Learning with Python.

H Yao and et al (2018) Modeling Spatial-Temporal Dynamics for Traffic Prediction. In Proceedings of. ACM, New York, NY, USA, Article 4, 9 pages. https://doi.org/10.475/123_4

Jason Brownlee (2019) How to Use Word Embedding Layers for Deep Learning with Keras. Available at: https://machinelearningmastery.com/use-word-embedding-layers-deep-learning-keras/

Kotharishruti (2018). Sentiment-Analysis-of-Movie-Reviews Available at: https://github.com/kotharishruti/Sentiment-Analysis-of-Movie-Reviews/blob/master/Sentiment_Analysis.ipynb

Marwan Torki (2018) A Text Classifier Using Weighted Average Word Embedding. Conference: International Japan-Africa Conference on Electronics, Communications and Computations (JAC-ECC)

Scikit-learn documentation (2019) Text-feature-extraction. Available at: https://scikit-learn.org/stable/modules/feature_extraction.html#text-feature-extraction

Wangle (2018) kaggle_Quora_Insincere_Questions_Classification. Available at: https://github.com/wangle1218/NLP-competition-baseline/blob/master/kaggle_Quora_Insincere_Questions_Classification.ipynb

Wangyue (2019) Text-classification. Available at: https://github.com/wavewangyue/text-classification