

NATURAL LANGUAGE PROCESSING

COURSE WORK

2023-2024

Full Name: Lahari Mullaguru

URN: 6843990

Email: lm02027@surrey.ac.uk

Table of Contents:

1. Analyse and visualise the dataset.....	1
2. Experimentation with four different experimental setups.....	4
2.1. Data Pre-Processing techniques	4
2.2. Comparing different Vectorization methods.....	7
2.2.1. Word2Vec and FFNN.....	7
2.2.2. TF-IDF and FFNN	9
2.2.3. Fast Text and FFNN.....	10
2.2.4. Hashing Vectorizer and FFNN.....	10
2.3. Comparing Algorithms.....	11
2.3.1. Word2Vec and FFNN.....	11
2.3.2. Word2Vec and Random Forest Classifier	11
2.3.3. Word2Vec and SimpleRNN	12
2.4. Comparing loss functions and optimizers.....	12
3. Analyse testing for each of the four experiment variations.....	15
4. Discussion of best results from the testing.....	24
5. Evaluation of overall attempt and outcome	27
6. References	28

AIM: Aim is to perform the sequence classification for identifying abbreviations and long forms, with 'AC' tagging for the abbreviations and 'LF' tags denoting the long forms and 'B-O' denoting all other tokens.

1 Analyse and visualise the dataset

1.1 Imports

Install all the required libraries and import them in the notebook. For this task, the PLOD dataset, which consists of 50,000 labeled tokens, is used from the Hugging Face repository.

Load the dataset from the Hugging Face,

```
dataset = load_dataset("surrey-nlp/PLOD-CW")
```

1.2 Analyzing the dataset

After downloading the dataset, analyze the dataset to perform the experiments.

```
Available splits: dict_keys(['train', 'validation', 'test'])
```

The available splits in the given dataset are 'train', 'validation' and 'test'.

The description of dataset is,

```
Total number of rows and columns are: (1072, 3)
```

```
Information about dataset:
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1072 entries, 0 to 1071
Data columns (total 3 columns):
#   Column      Non-Null Count  Dtype
---  -
0   tokens      1072 non-null   object
1   pos_tags    1072 non-null   object
2   ner_tags    1072 non-null   object
dtypes: object(3)
memory usage: 25.3+ KB
None
```

The train dataset contains 1072 sequences and there are no null values in the dataset. Tokens column contains list of sequences where each list contains string of tokens. The corresponding pos tags and ner tags of the tokens in the sequence are listed in 'pos_tags' column and 'ner_tag' column respectively.

Description of Dataset:

	tokens	pos_tags
count	1072	1072
unique	1071	1064
top	[), .]	[PROPN, PUNCT, ADJ, NOUN, NOUN, PUNCT]
freq	2	5

	ner_tags
count	1072
unique	978
top	[B-AC, B-O, B-LF, I-LF, I-LF, B-O]
freq	8

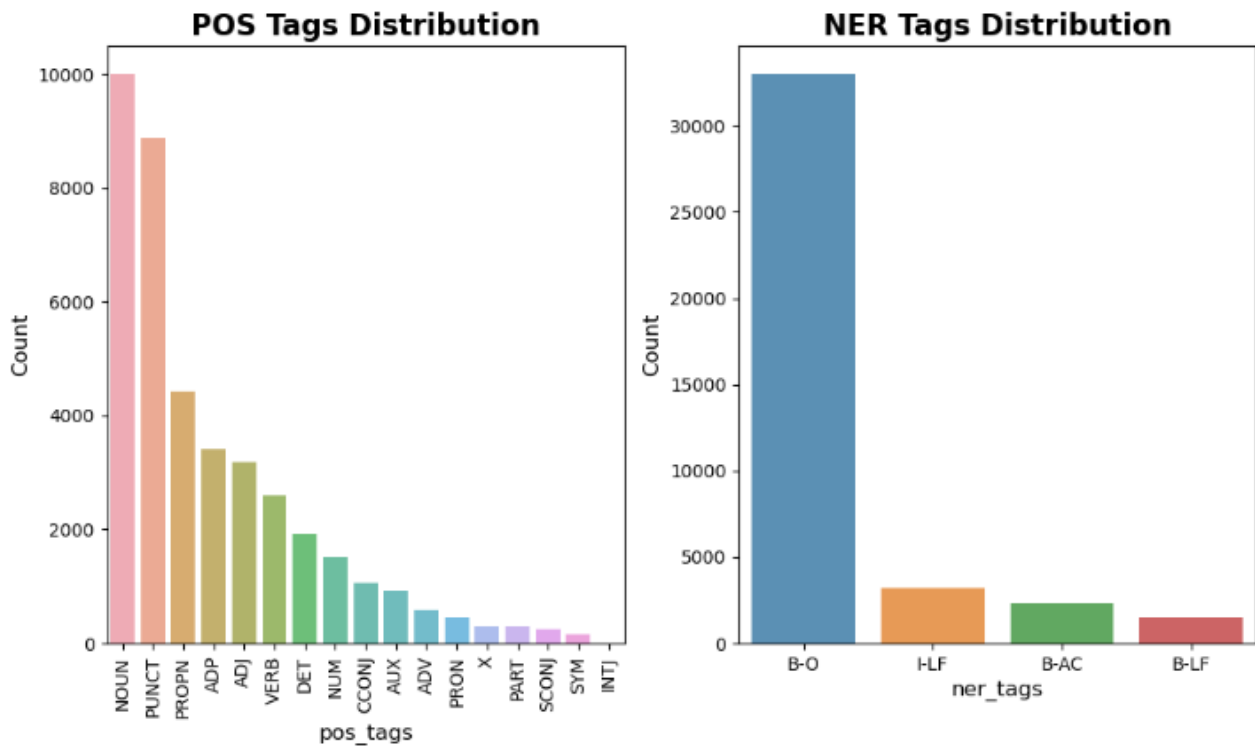
	tokens	pos_tags	ner_tags
0	[For, this, purpose, the, Gothenburg, Young, P...	[ADP, DET, NOUN, DET, PROPN, PROPN, PROPN, PRO...	[B-O, B-O, B-O, B-O, B-LF, I-LF, I-LF, I-LF, I...
1	[The, following, physiological, traits, were, ...	[DET, ADJ, ADJ, NOUN, AUX, VERB, PUNCT, ADJ, N...	[B-O, B-O, B-O, B-O, B-O, B-O, B-O, B-LF, I-LF...
2	[Minor, H, antigen, alloimmune, responses, rea...	[ADJ, PROPN, NOUN, ADJ, NOUN, ADV, VERB, ADP, ...	[B-O, B-AC, B-O, B-O, B-O, B-O, B-O, B-O, B-O,...

Total number of rows and columns in train data are: (1072, 3)

Total number of rows and columns in validation data are: (126, 3)

Total number of rows and columns in test data are: (153, 3)

1.3 Visualizing the dataset



From the above visualization of NER tags, it can be observed that:

- The class imbalance is very high in the given dataset
- The 'B-O' tags are more numerous compared to all other labels, with approximately 35,000 'B-O' tags in the dataset.
- The I-LF, B-AC, B-LF tags in the dataset are fewer than 5000.
- Pre-processing of data is required before training due to the high class imbalance.
- Hence, during the training phase, the class imbalance should be considered, and a methodology should be designed to effectively process and fit the data into a model.

1.4 Observations from the Dataset Analysis

Total Number of Tokens:

- The train dataset comprises 40,000 tokens. The most frequent tokens include commas, parentheses '()', and the word 'the'.

Pre-processing and Cleaning Requirement:

- Given the presence of frequent punctuation marks and common stop words like 'the', it is evident that preprocessing and cleaning of the data are necessary before performing sequence classification for abbreviation and long-form detection.

Distribution of POS Tags:

- The most common POS tags observed in the dataset are NOUN and PUNCT, with frequencies of approximately 10,000 and 9,000 respectively. This suggests that nouns and punctuation marks are predominant in the dataset.

Distribution of NER Tags:

- The most prevalent NER tag is B-O, occurring over 30,000 times. In contrast, NER tags such as I-LF, B-AC, and B-LF appear fewer than 5,000 times each.
- This indicates a class imbalance among NER tags, which could impact the performance of sequence classification.

With a focus on abbreviation and long-form detection, the dataset's characteristics provide valuable insights into potential challenges and strategies for model development. Preprocessing steps should address tokenization, stopwords removal, and normalization to facilitate accurate classification.

In summary, the dataset analysis underscores the importance of preprocessing and cleaning steps to address token frequency variations and class imbalances. These observations inform the approach towards sequence classification for abbreviation and long-form detection.

2 Experimentation with four different experimental setups

2.1 Data Pre-Processing techniques

The pre-processing steps I have performed to the given dataset are:

- Converting the sequences into tokens
- Lowercasing
- Normalization- Removing extra spaces and punctuations
- StopWords removal
- Removing Null values and Duplicates
- Stemming and Lemmatization

Converting Sequences into tokens using Unigram:

The sequences can be converted into tokens using **n-grams** or a word tokenizer. Here, I have used n-grams, and the value n=1 specifies that it is a '**Unigram**', which takes only one word at a time. The ngram is imported from nltk library.

```
[('For',), ('this',), ('purpose',), ('the',), ('Gothenburg',)]
```

Tokenization can also be performed using list comprehension.

	tokens	pos_tags	ner_tags
0	For	ADP	B-O
1	this	DET	B-O
2	purpose	NOUN	B-O
3	the	DET	B-O
4	Gothenburg	PROPN	B-LF

Total Number of words: 40000
Vocabulary size: 9133

Lowercasing the tokens:

```
wf['tokens'] = wf['tokens'].str.lower()
```

Generally, the tokens are lowercased to decrease the vocabulary size. If 'For' and 'for' words are treated differently then model might not learn full semantic context of word.

2	purpose	NOUN	B-O
3	the	DET	B-O
4	gothenburg	PROPN	B-LF

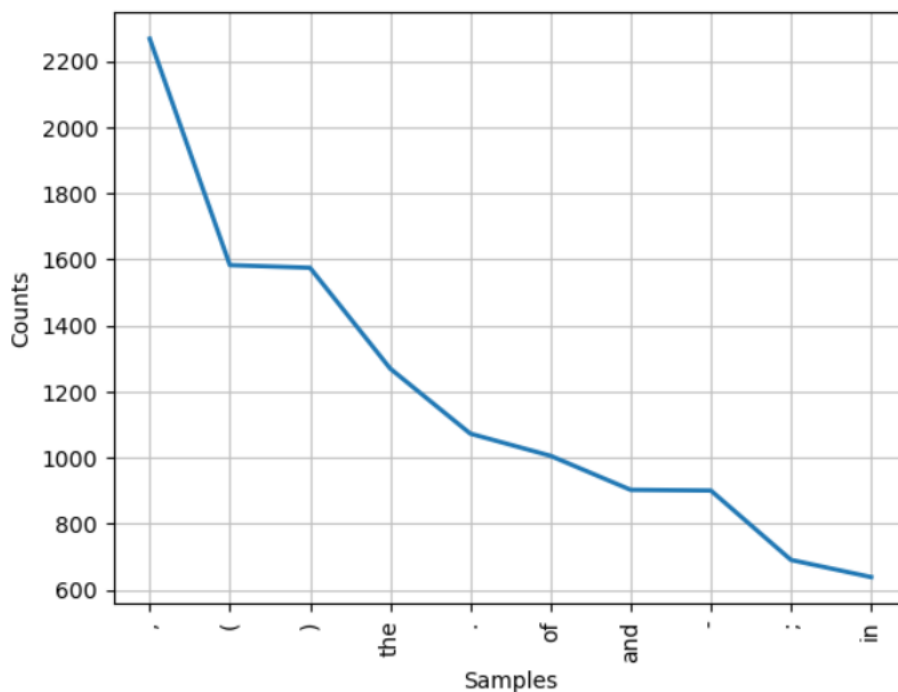
Lowercasing guarantees that the meaning of the word is more accurately represented.

Frequency of Tokens:

```
PLOD_DATA
[(',', 2268), ('(', 1583), (')', 1575), ('the', 1270), ('.', 1073), ('of', 1006), ('and', 903), ('-', 901), (';', 691), ('i
n', 639)]
```

Here, the data is not pre-processed yet. So, the most frequently occurring tokens are punctuations and stop words.

Distribution of most frequent tokens in plot:



Normalization:

Normalization is the process of removing all punctuations and extra spaces in a text. Here, I performed normalization to remove all punctuations in the given dataset.

After normalization the distribution of frequent tokens are:

```
[('the', 1271), ('of', 1006), ('and', 904), ('in', 639), ('to', 476), ('a', 394), ('with', 297), ('for', 235), ('were', 201), ('was', 196)]
```

Stop Words removal:

Eliminating stop words isn't always essential or beneficial. Stop words can sometimes convey significant context. For example, "not" may be first regarded by sentiment analysis as a stop word, but it can also change the meaning of a statement. The NLP task at hand determines whether stop words should be removed or not. In this task, stop words are considered as B-O tag, so while implementing the model stop words shouldn't be removed.

	tokens	pos_tags	ner_tags
0		ADP	B-O
1		DET	B-O
2	purpose	NOUN	B-O

From the above result, it can be observed that stop words are removed from the dataset.

Remove Null values and Duplicates:

Null values indicate that a data point contains missing information. Including null values can distort your analysis's findings. The term "duplicates" refers to data points that appear more than once. As a result, your dataset will get larger, and your analysis may become less effective.

```
wf.drop_duplicates(subset=['tokens', 'pos_tags', 'ner_tags'], inplace=True)
print('After removing duplicates:', len(wf))
```

```
After removing duplicates: 10181
```

The above implementation, drop all the duplicates in the dataset.

Stemming and Lemmatization:

- Stemming involves reducing a set of words to their base form by cutting off prefixes or suffixes, without considering the "grammatical meaning" of the resulting stem.
- Lemmatization is similar to stemming; in that it reduces words to their base form. However, unlike stemming, lemmatization considers the actual meaning of the base word.
- Stemming is done using the PorterStemmer and Lemmatization by WordNet Lemmatizer which are imported from the nltk library.
- Lemmatization requires downloading specific language resources because it involves complex linguistic processing. It uses detailed linguistic rules and dictionaries to accurately determine the base form of a word based on its part of speech and context. So, we use "nltk.download('wordnet')".

	tokens	pos_tags	ner_tags	stemmed_tokens	lemmatized_tokens
0	purpose	NOUN	B-O	purpos	purpose
1	gothenburg	PROPN	B-LF	gothenburg	gothenburg
2	young	PROPN	I-LF	young	young
3	persons	PROPN	I-LF	person	person

2.2 Comparing different Vectorization methods

2.2.1 Word2Vec and FFNN

Word2Vec Vectorization:

The Word2Vec methodology is based on two primary algorithms: the Continuous Bag of Words (CBOW) and skip-gram models. These algorithms can be implemented using two different training techniques: hierarchical softmax or negative sampling. The skip-gram architecture predicts neighboring words from a given central word, while the CBOW approach estimates a target word from the surrounding context words.

I have chosen Word2Vec model because, it considers word order and context and treats words as separate entities. Words are encoded in a lower dimensional space as dense vectors using Word2vec. Comparing this compressed version to sparse BOW matrices, computations are performed more efficiently. However, it can't handle the Out of Vocabulary (OOV) words. So, I also experimented with Fast Text Vectorization which handles the OOV words.

Implementing word2vec vectorization:

- I concatenated train, validation, and test datasets which is not pre-processed, into a single dataset.
- Encoded the NER tags into labels since the model can only be trained with numerical or vector representations of words.

```
tag_encoder = {'B-O': 0, 'B-AC': 1, 'B-LF': 2, 'I-LF': 3}
```

- Word2Vec model is imported from gensim library.
- The model is trained with sequences of data from the PLOD dataset.
- While defining the model hyperparameters, Initially I gave the vector dimensions as 20 and minimum count as one and window size as six.

```
model_x = Word2Vec(sequences, vector_size=20, window=6, min_count=1, workers=4)
```


- Here, I iterated through each sequence in the dataset and appended the vectors of a word into a list. If the word is not present in the word2Vec model, I append zero vectors of same dimensions.
- Converted the word vectors and tags into an array to prepare the data for training.
- During the visualization, we have seen that the class imbalance is very high i.e., B-O tags are very high in the dataset. So, I performed an over sampling technique '**SMOTE**' which is used for balancing the classes in the dataset.

```
#upsampling technique to balance the tags distribution

from imblearn.over_sampling import SMOTE
smote=SMOTE()
```

```
#Resampling the data to avoid class-imbalance in dataset
X_smote, y_smote = smote.fit_resample(X,y)
```

- SMOTE creates synthetic samples by interpolating between instances of the minority class and their nearest neighbors, in contrast to traditional oversampling techniques that replicate examples from the minority class. The algorithm identifies k-nearest neighbors (kNN) of minority samples. In order to guarantee that new data points are generated within the appropriate class space, these neighbors ought to be members of the same minority class.
- Now, the input and output features can be fit into the FFNN model for training.

Implementation of FFNN model:

A Keras feed-forward neural network is a form of artificial neural network (ANN) where data flows linearly from the input layer, passing through one or several intermediary/hidden layers, to the final output layer. without forming any cycles. This linear processing differentiates it from recurrent neural networks (RNNs), which have feedback connections, and convolutional neural networks (CNNs), which incorporate filters and spatial hierarchies in processing. So, for the ease of implementation I have chosen FFNN model.

Defining the FFNN model,

Layer (type)	Output Shape	Param #
dense_8 (Dense)	(None, 128)	2,688
dense_9 (Dense)	(None, 64)	8,256
dense_10 (Dense)	(None, 32)	2,080
dense_11 (Dense)	(None, 4)	132

Compile and train the model,

```
model_ffn.compile(optimizer='adam',  
                  loss='sparse_categorical_crossentropy',  
                  metrics=['accuracy'])
```

Here, I used 'sparse categorical cross-entropy' loss function and 'adam' optimizer for the better training of data.

In this methodology, I have trained the input features and output features with word2vec vectorization and FFNN model.

2.2.2 TF-IDF and FFNN

TF-IDF is an extension of the Bag of Words (BoW) concept by assessing a word's significance in a document by combining the Term Frequency (TF) with Inverse Document Frequency (IDF).

- TF-IDF can be effective for simpler classification tasks.
- In the TF-IDF vectorization, instead of sequences I gave tokens of data to create vectors.
- It is easy to implement but for complex models, it may produce high dimensional feature vectors that are computationally costly.

```
tokens=train_data['tokens'].tolist()  
vectorizer = TfidfVectorizer()  
X = vectorizer.fit_transform(tokens)
```

```
X2_smote, y2_smote = smote.fit_resample(X,y)
```

- Here, I again resampled the data using the SMOTE technique to balance the four classes so that the model can be trained effectively.

FFNN model for TF-IDF:

```
model2_ffn = models.Sequential([  
    layers.Dense(128, activation='relu', input_shape=(9021,)),
```

I changed the input shape of FFNN model to the dimensions according to TF-IDF vectors.

```
history = model2_ffn.fit(trainX2, testX2, epochs=10, batch_size=64)
```

Here I reduced the number of epochs for training when compared to word2vec, because the model trained effectively for the smaller number of epochs. I used the same optimizer and loss function and observed that it takes more amount of time for training compared to Word2Vec.

2.2.3 Fast Text and FFNN

In this experiment, I used FastText for vectorization because it handles out-of-vocabulary words unlike the Word2Vec model, where I would need to append zeros for such words.

```
model3 = FastText(sequences, vector_size=10, window=5, min_count=1, workers=4)
```

I reduced the vector dimensions compared to word2vec model, inorder to experiment with input features of less vector dimensions and to observe whether it produce good results or not.

```
X3_smote, y3_smote = smote.fit_resample(X3_train,y3_train)
```

I used the same resampling technique in this vectorization.

FFNN model for Fast Text:

```
#define the FFNN model for Fast Text  
model3_ffn = models.Sequential([  
    layers.Dense(128, activation='relu', input_shape=(10,)),
```

Here, I changed the input shape according to the Fast Text input vector dimensions.

```
history = model3_ffn.fit(trainX3, testX3, epochs=20, batch_size=64)
```

I adjusted the number of epochs to improve the performance of the model and used the same optimizer and loss function.

2.2.4 Hashing Vectorizer and FFNN

The Hashing Vectorizer employs a hashing technique to map token string names to feature integer indices, enhancing memory efficiency and scalability for handling large datasets.

- It can be used in scenarios where there are large number of unique terms. Since it employs hashing, it eliminates the need to store the complete vocabulary.
- Unlike the TF-IDF vectorizer, it does not encode word frequency.

```
#vectorize the tokens  
tokens=train_data['tokens'].tolist()  
vectorizer = HashingVectorizer(n_features=800)  
X3 = vectorizer.fit_transform(tokens)
```

```
X4_smote, y4_smote = smote.fit_resample(X3,y3)
```

- I used the smote resampling technique for the class imbalance.

FFNN model for Hashing Vectors:

```
#define the FFNN model for Hashing Vectorizer
model4_ffn = models.Sequential([
    layers.Dense(128, activation='relu', input_shape=(800,)),
```

Here, I changed the input shape according to hashing vectors input dimensions.

```
history = model4_ffn.fit(trainX4, testX4, epochs=10, batch_size=64)
```

I used the same loss function and optimizer and trained the model with 10 epochs to get better accuracy and f1-score.

Hence, the training of the FFNN model using various vectorization methods has been completed. I have detailed the accuracy, F1-score, confusion matrix, and accuracy curve for each experiment in the 'Analysis of Testing' section.

2.3 Comparing Algorithms

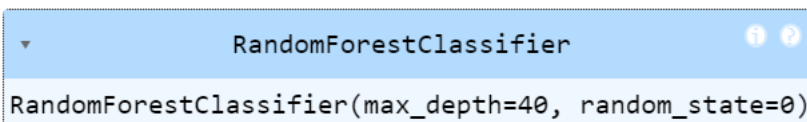
2.3.1 Word2Vec and FFNN

I have described the implementation of the Word2Vec and FFNN model in the previous section.

2.3.2 Word2Vec and Random Forest Classifier

A Random Forest Classifier employs an ensemble of decision trees to tackle classification problems. It builds multiple decision trees on various subsets of the dataset and combines their outputs to enhance predictive accuracy and prevent overfitting.

Better generalization on previously undiscovered sequences is achieved by averaging predictions from several trees, which lessens the effect of overfitting on a particular training dataset.



```
RandomForestClassifier
RandomForestClassifier(max_depth=40, random_state=0)
```

The input and output features are upsampled and then trained using a Random Forest Classifier. I have set the **max_depth** to 40, as I observed improved prediction results with an increased depth in the Random Forest.

2.3.3 Word2Vec and SimpleRNN

A Simple Recurrent Neural Network (RNN) is typically employed for sequence processing. This type of RNN, known as a fully-connected RNN, passes the output from one timestep as the input to the next timestep. In RNNs, the output is influenced by previous elements in the sequence.

- Unlike feedforward networks, where each node may have different weights, RNNs maintain consistent weights across each layer within the network.
- Here I have resampled the data and then trained using Simple RNN model.
- RNNs are built to handle sequential data, they learn relations between elements in a sequence.

```
#Define the model for simple RNN
model_rnn = models.Sequential([
    layers.SimpleRNN(128, activation='relu', input_shape=(20, 1),
        return_sequences=False),
```

I used same loss function and optimizer to train the model. Here I observed that the time required for training of the model is high compared to all other models.

I noticed that as the number of epochs increased, the model trained more effectively and yielded improved results.

For all the neural network models, I chose the **Adam optimizer** for its efficiency with large datasets and its adaptive learning rate, which helps in faster convergence. The **sparse categorical cross-entropy** loss function was chosen because it is well-suited for problems involving classification into multiple categories, optimizing computational efficiency. I used the **ReLU** activation function because it introduces necessary non-linearity, facilitating quicker and more effective learning in deep networks. These choices collectively ensure robust performance and efficient training of the model.

2.4 Comparing loss functions and optimizers

For comparing different loss functions and optimizers, I have used the word2vec and FFNN model. The different loss functions I have compared are:

Categorical Crossentropy loss function

This function is suitable for multiclass classification scenarios involving more than two class labels. Here, output labels are one hot encoded to test with categorical cross entropy.

```
testX_one_hot = tf.keras.utils.to_categorical(testX,4)

model_ffn5.compile(optimizer='adam',
    loss='categorical_crossentropy',
```

Poisson Loss Function

The Poisson loss is calculated between the predicted value and the actual value. This loss function is typically used with datasets that follows a Poisson distribution.

Though Poisson loss function is not suitable for sequence classification tasks, I just want to experiment whether the model predicts the labels or not after training.

```
loss=tf.keras.losses.Poisson(name="poisson")
model_ffn5.compile(optimizer='adam',
                  loss=loss,
```

KL Divergence Loss Function:

The KL Divergence, also known as Kullback-Leibler Divergence Loss function, is calculated between the predicted value and the actual value when dealing with continuous distributions.

It can be used in sequence classification to compare the true distribution (one-hot encoded vector representing the correct class) with the probability distribution predicted by the model (the SoftMax output).

```
loss=tf.keras.losses.KLDivergence(name="kl_divergence")
model_ffn5.compile(optimizer='adam',
                  loss=loss,
```

Loss Estimation of different Loss Functions:

The losses incurred during model training using various loss functions are computed in the following manner:

```
CategoricalCrossentropy loss is: 523973.88
```

```
Poisson loss is: 2.0877752
```

```
KLDivergence loss is: 19035.469
```

Each loss function evaluates model accuracy under different assumptions and data characteristics, with the Categorical Crossentropy showing the greatest loss, indicating potential overfitting or misclassification issues.

Comparing Optimizers:

I experimented with different optimizers for the FFNN model.

Stochastic Gradient Descent:

The SGD optimizer employs gradient descent combined with momentum.

Instead of updating the model parameters all at once, it repeatedly iterates through the dataset using small, random batches of data. Because of this, it is computationally efficient for big text data sets.

In this experiment, I set the learning rate at 0.01 and the optimizer's momentum at 0.9 for the parameter values.

```
opt = SGD(learning_rate=0.01, momentum=0.9)
model_ffn6.compile(optimizer=opt,
```

Adagrad Optimizer:

The Adagrad optimizer adjusts learning rates based on individual parameters, with the adjustment depending on how frequently a parameter is updated. This results in different learning rates for various weights, allowing the optimizer to adapt the learning rate for each feature uniquely.

The learning rate of the optimizer is 0.5.

Adagrad uses previous gradients to modify the learning rate for every parameter. For sequence classification tasks, where certain features may be less frequent, this can be advantageous. It may facilitate the model's learning of these less frequent features.

```
model_ffn6.compile(optimizer=tf.keras.optimizers.Adagrad(learning_rate=0.5),
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
history_Adagrad = model_ffn6.fit(trainX, testX, epochs=10, batch_size=64)
```

Adadelata Optimizer:

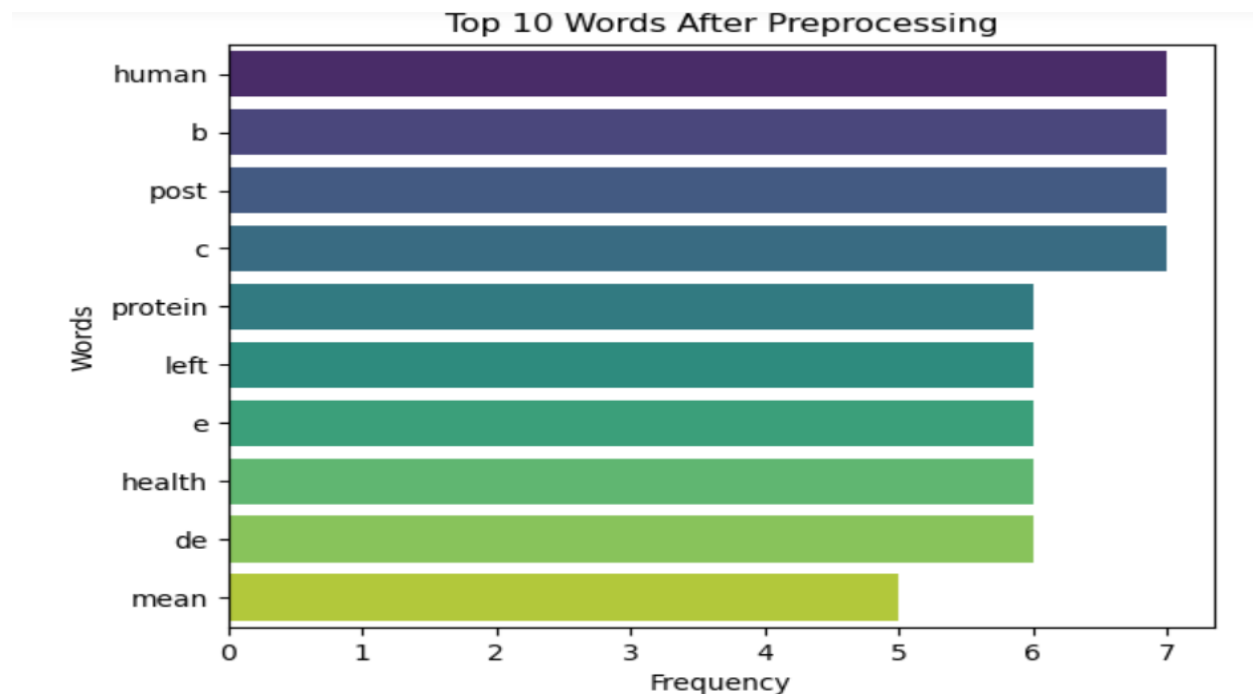
The Adadelata optimizer utilizes an adaptive learning rate approach combined with the stochastic gradient descent method. This optimizer addresses the issue of ongoing reduction of the learning rate throughout training.

Adadelata, in contrast to SGD, gathers data about previous gradients, which aids in resolving the vanishing gradient issue that might arise in sequence models.

The learning rate of the optimizer is 0.1.

```
opt = tf.keras.optimizers.Adadelata(learning_rate=0.1)
model_ffn6.compile(optimizer=opt,
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
history_Adadelata = model_ffn6.fit(trainX, testX, epochs=10, batch_size=64)
```


After pre-processing the data, bar plot of top 10 tokens is,



3.2 Comparing different Vectorization methods

Accuracy, F1-score, and Confusion matrix of different vectorization methods are as follows,

	ACCURACY	F1-SCORE
Word2Vec and FFNN	0.808	0.804
TF-IDF and FFNN	0.850	0.849
Fast Text and FFNN	0.658	0.645
Hash Vectorizer andFFNN	0.659	0.656

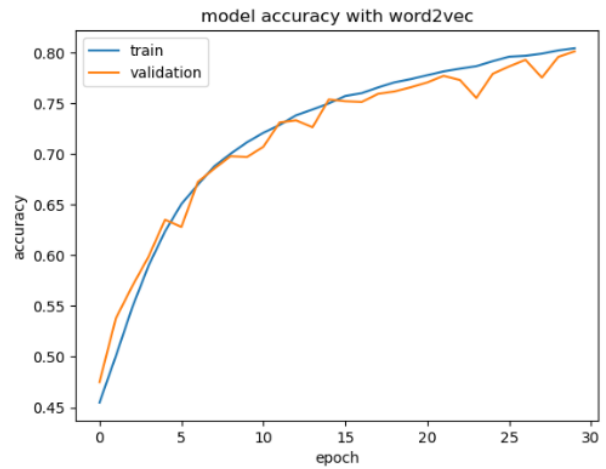
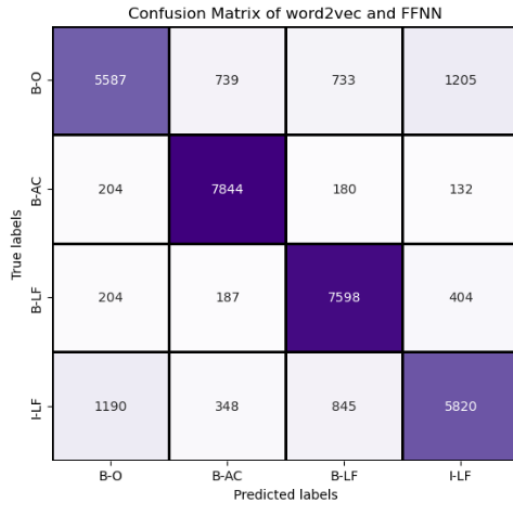
Confusion Matrix and accuracy curve:

Word2vec and FFNN:

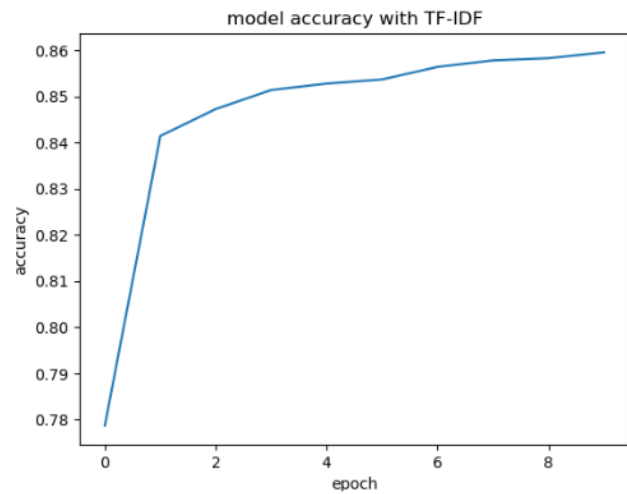
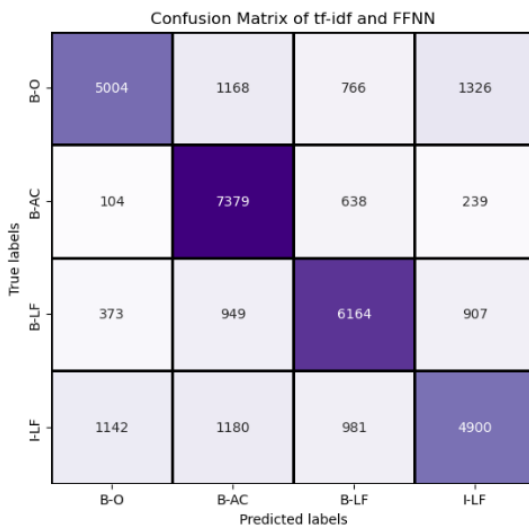
```
#predict the labels
y_pred = model_ffn.predict(trainY)
y_pred_labels = np.argmax(y_pred, axis=1)

# calculate the accuracy
accuracy = accuracy_score(testY, y_pred_labels)

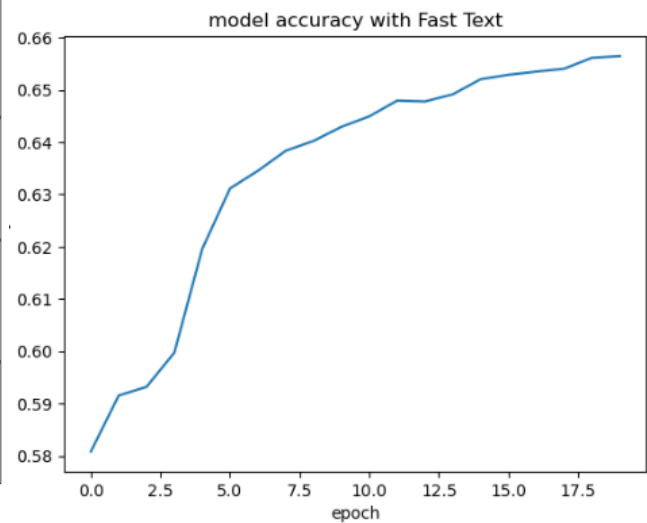
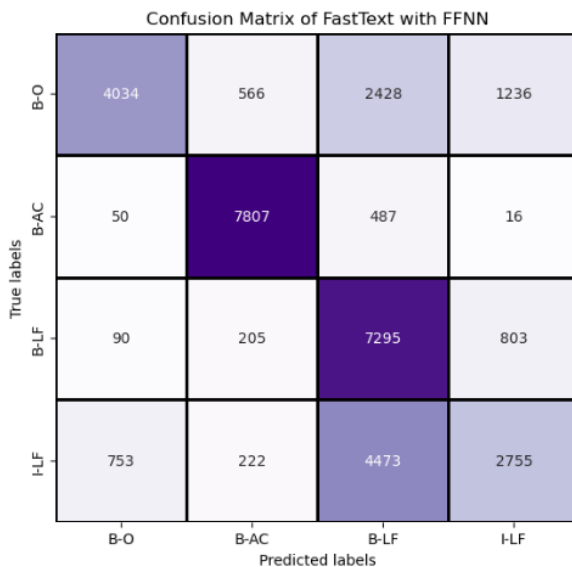
conf_matrix = confusion_matrix(testY, y_pred_labels)
```



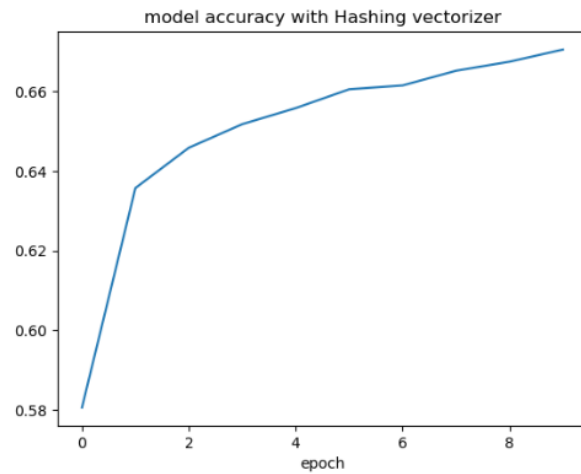
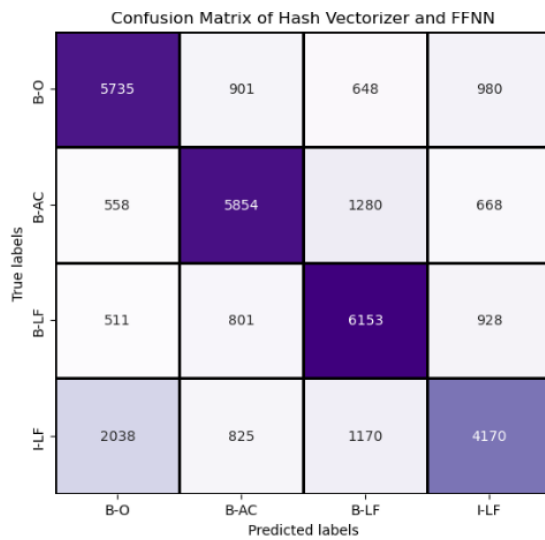
TF-IDF and FFNN:



Fast Text and FFNN:



Hashing Vectorizer and FFNN:

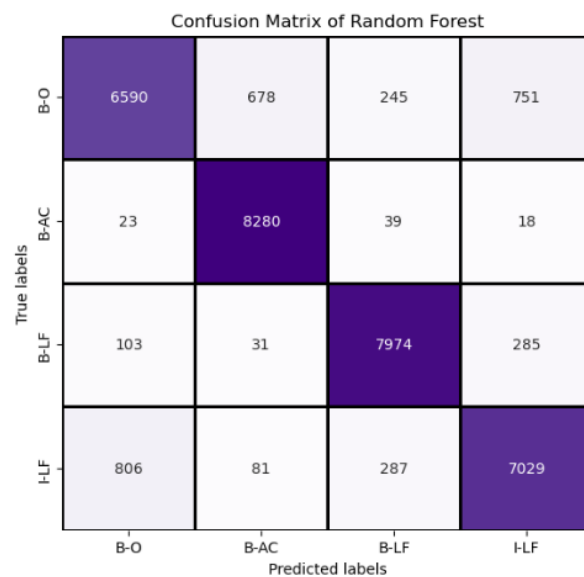


3.3 Comparing Algorithms

The accuracy and F1-score of different algorithms that I experimented with are:

	ACCURACY	F1-SCORE
Word2vec and FFNN	0.808	0.804
Word2vec and Random Forest Classifier	0.899	0.897
Word2vec and Simple RNN	0.705	0.701

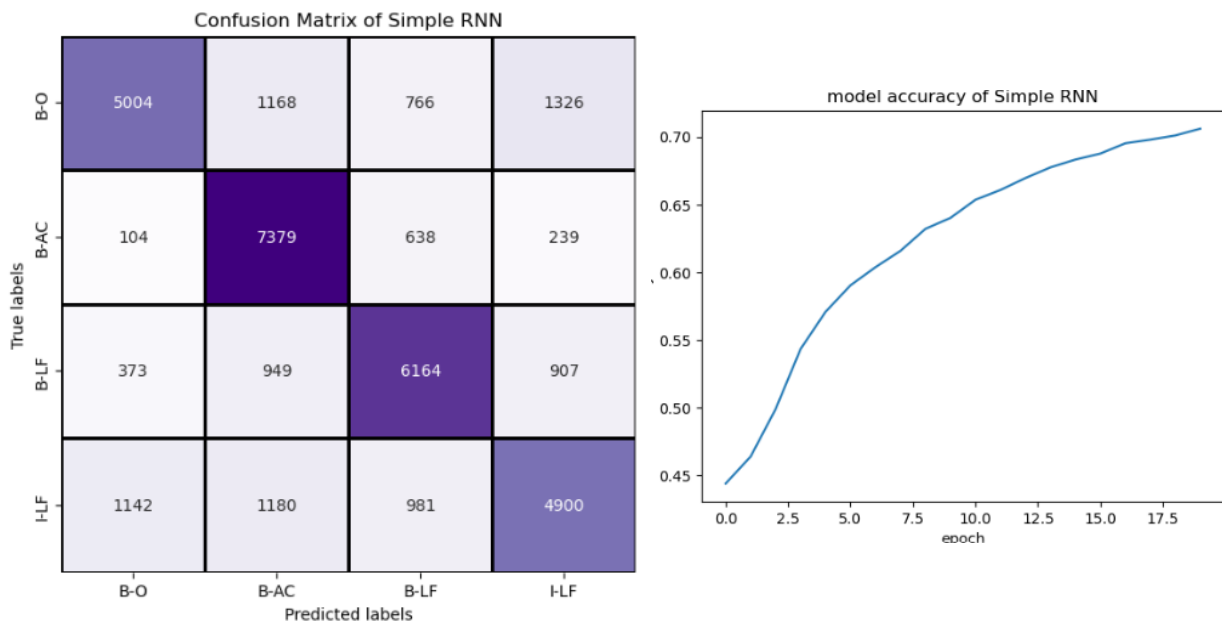
Word2Vec and Random Forest Classifier:



Classification Report of Random Forest:

	precision	recall	f1-score	support
B-O	0.88	0.80	0.83	8264
B-AC	0.91	0.99	0.95	8360
B-LF	0.93	0.95	0.94	8393
I-LF	0.87	0.86	0.86	8203
accuracy			0.90	33220
macro avg	0.90	0.90	0.90	33220
weighted avg	0.90	0.90	0.90	33220

Word2Vec and Simple RNN:



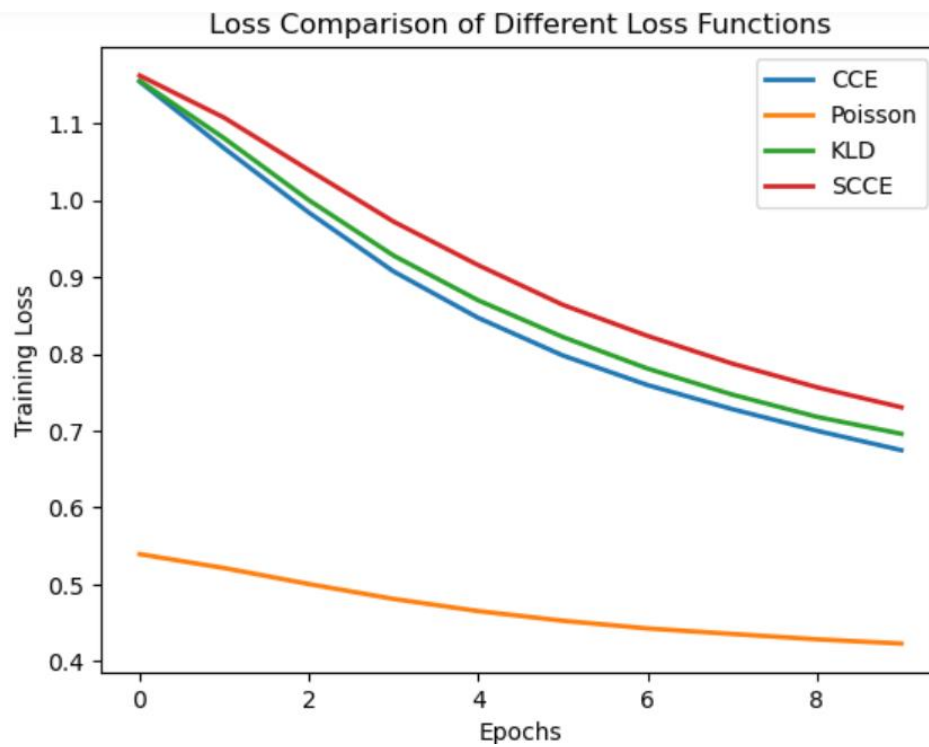
3.4 Comparing loss functions and optimizers

Comparing Loss Functions:

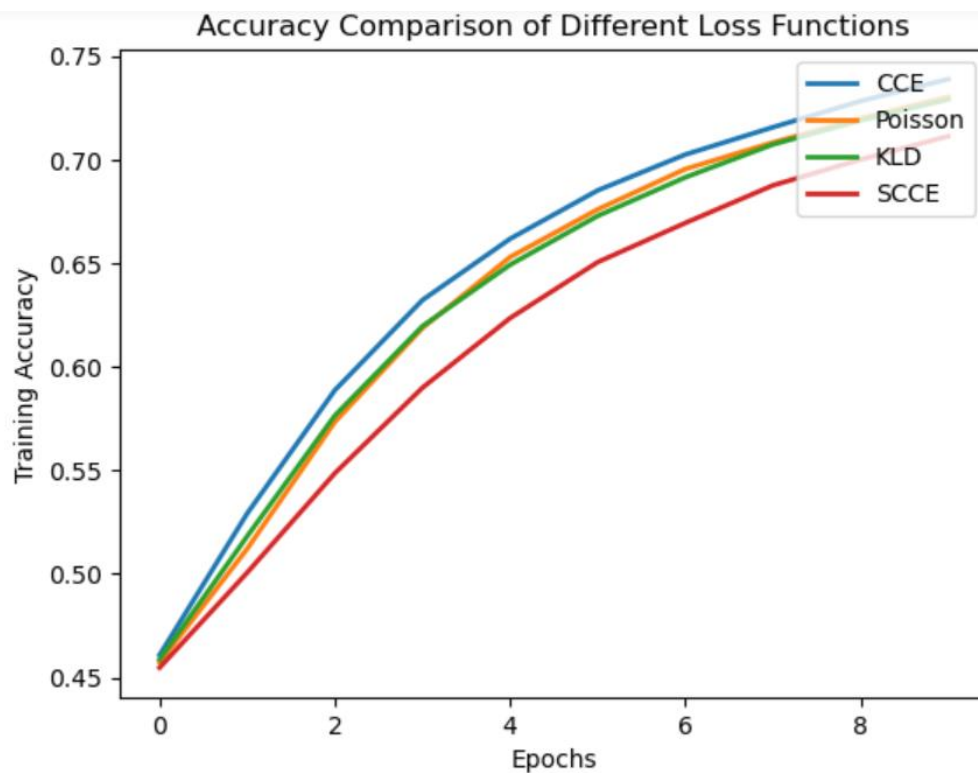
The accuracy and F1-score of Word2Vec and FFNN model with different loss functions are:

	ACCURACY	F1-SCORE
Sparse Categorical Crossentropy	0.808	0.804
Categorical Crossentropy	0.724	0.720
Poisson Loss Function	0.725	0.719
KL Divergence Loss	0.780	0.775

Loss Comparison of different Loss functions:



Accuracy Comparison of different Loss functions:

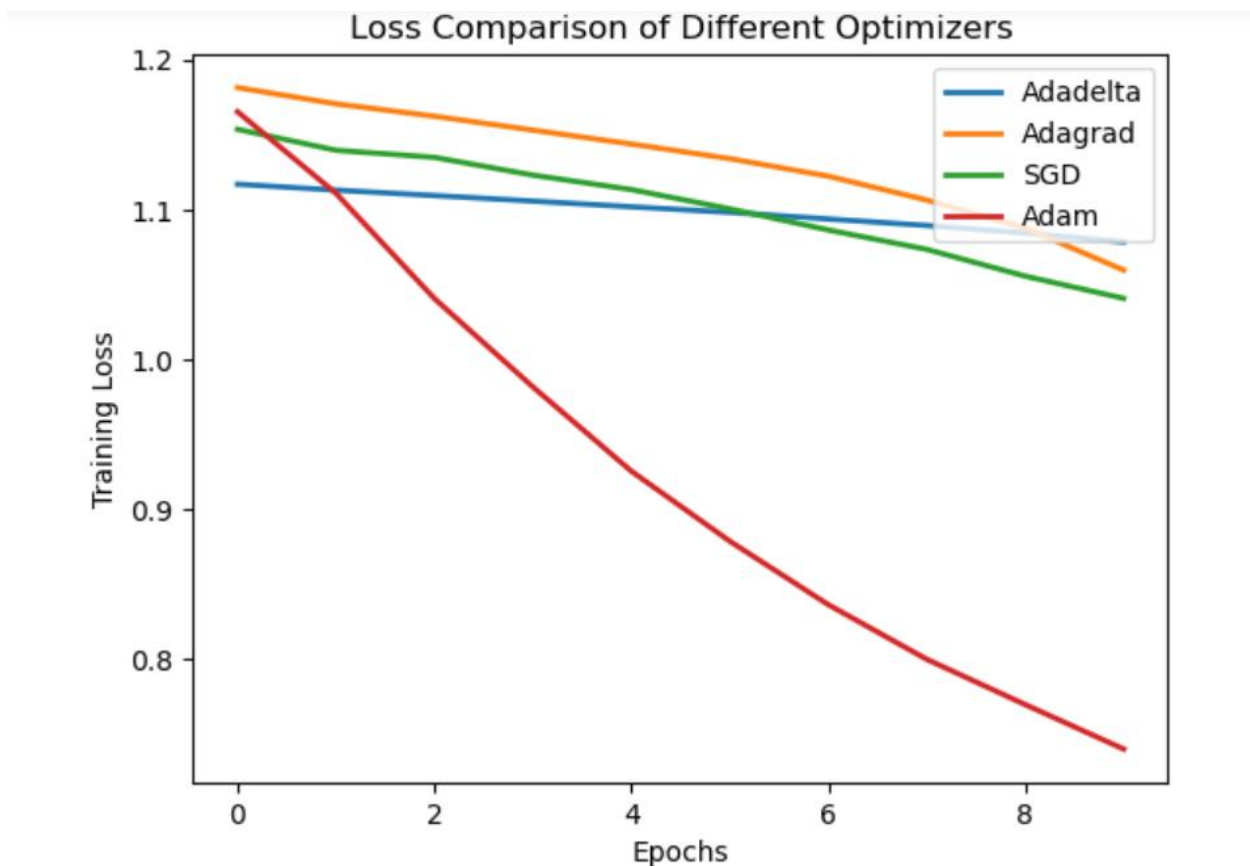


Comparing Optimizers:

The accuracy and F1-score of Word2Vec and FFNN model with different optimizers are:

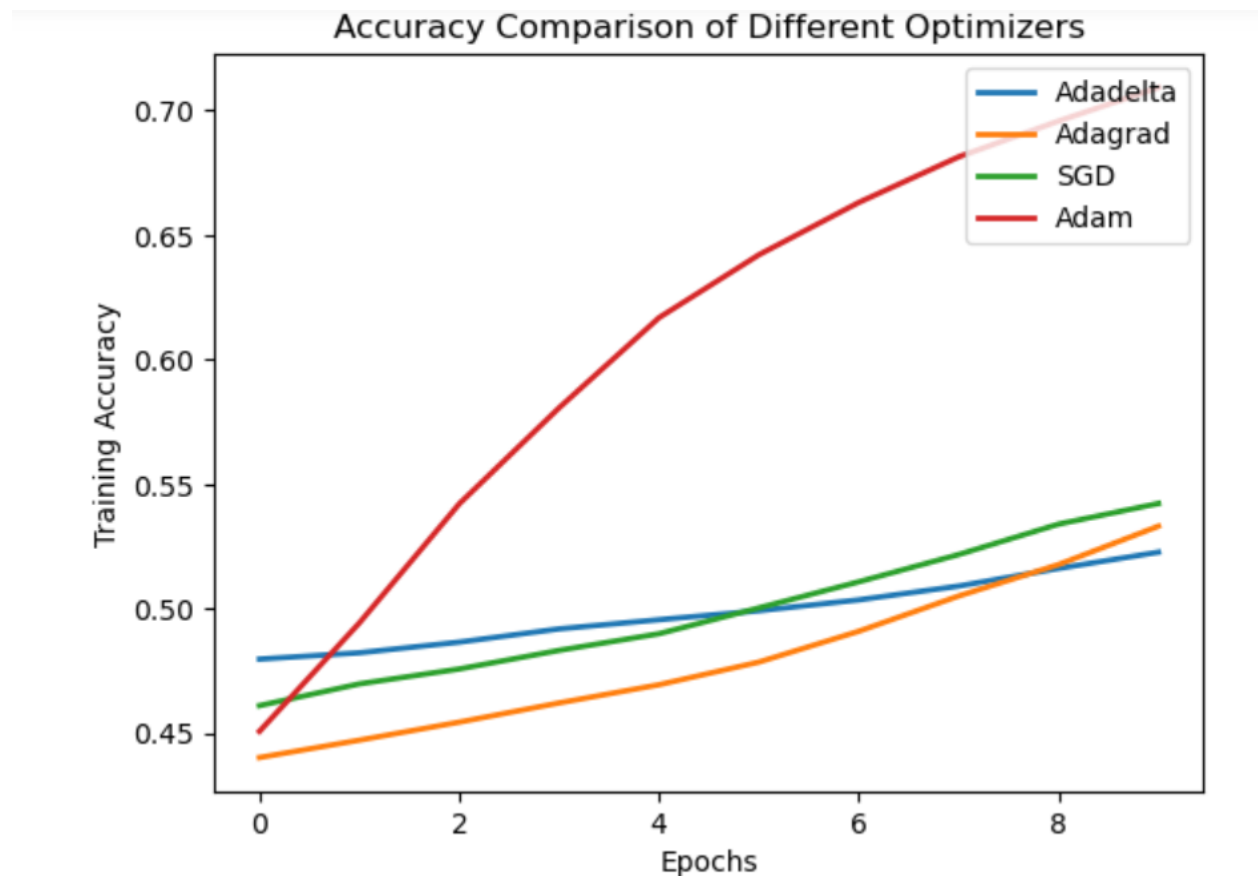
	ACCURACY	F1-SCORE
Adam Optimizer	0.808	0.804
Stochastic Gradient Descent	0.563	0.559
Adagrad Optimizer	0.529	0.510
Adadelata Optimizer	0.517	0.509

Loss Comparison of different optimizers:



It can be inferred that using the 'adam' optimizer results in minimal loss during the training of the model. Therefore, for sequence classification tasks, the 'adam' optimizer is the optimal choice.

Accuracy Comparison of different optimizers:



Error Analysis on the predictions:

I have created a pipeline to get predictions of the given input to perform the error analysis. This involved evaluating the predictions for different algorithms, including a Feedforward Neural Network (FFNN) model, Random Forest, and Simple RNN, and comparing them with the true labels. To perform the analysis, I have taken the sequences from the PLOD dataset and passed them as input to the pipeline to get predictions.

```
train_data_sequences.loc[3]
```

```
tokens      [EPI, =, Echo, planar, imaging, .]
pos_tags     [PROPN, PUNCT, NOUN, NOUN, NOUN, PUNCT]
ner_tags     [B-AC, B-O, B-LF, I-LF, I-LF, B-O]
Name: 3, dtype: object
```

The given input sequence contains all the four ner tags in it. So, I passed it as an input sequence to the pipeline to get predictions from the FFNN, Random Forest and Simple RNN. The obtained predictions are compared with true labels i.e., the 'ner_tags' that are listed above.

The predictions obtained are compared with true labels to analyse which tags are predicted wrong. From the experiments, I observed that accuracy of FFNN and Random Forest is high compared to all other models. So, I expected these models to produce better results.

True labels:

```
['B-AC', 'B-O', 'B-LF', 'I-LF', 'I-LF', 'B-O']
```

predictions of word2vec and FFNN:

```
['B-AC', 'B-O', 'B-LF', 'I-LF', 'I-LF', 'B-O']
```

predictions of word2vec and Random Forest:

```
['B-AC', 'B-O', 'B-LF', 'B-LF', 'B-O', 'B-O']
```

predictions of word2vec and Simple RNN:

```
['B-AC', 'B-O', 'B-LF', 'I-LF', 'I-LF', 'B-O']
```

Error Analysis:

- FFNN model gave accurate results and predicted every label correctly.
- Random Forest model predicted only 2 labels wrong. It predicted I-LF as B-LF and I-LF as B-O. These wrong predictions might be contextually similar words in the training data. Here, it predicted 'Echo' and 'planar' as B-LF because 'Echo-planar-imaging (EPI)' is a frequently used medical term, so the model classified these terms as same tags.
- Simple RNN model also predicted every label correctly.

Since, these algorithms produced better results, I gave another sentence as input which contains only one tag more predominantly.

True labels:

```
['B-O', 'B-O', 'B-O', 'B-O', 'B-O', 'B-O', 'B-O', 'B-LF']
```

predictions of word2vec and FFNN:

```
['B-O', 'B-O', 'B-O', 'B-O', 'I-LF', 'B-O', 'B-O', 'I-LF']
```

predictions of word2vec and Random Forest:

```
['B-O', 'B-O', 'B-O', 'B-O', 'I-LF', 'B-O', 'B-O', 'B-O']
```

predictions of word2vec and Simple RNN:

```
['I-LF', 'B-O', 'B-O', 'B-O', 'I-LF', 'B-O', 'B-O', 'B-LF']
```

Here, the given input sequence contains B-O tags predominantly and error analysis for these predictions are:

- FFNN model gave 2 wrong predictions B-O as I-LF. Here, considering the two words 'Fig 1', it suggests that the model is confusing the context, or the word has appeared inside similar entities during the training phase.

- Random Forest also gave 2 wrong predictions. As explained above, model is assuming B-O tags as I-LF for the words 'Fig 1' since it is considering token '1' as I-LF tag because its preceding word is 'Fig'.
- Simple RNN gave 2 wrong predictions. Here also it assumed B-O tag as I-LF tag for the tokens 'Fig 1'.

I observed, the models are predicting most of the I-LF tags wrongly as B-O or B-LF. So, I gave the next input sequence that contain I-LF tags predominantly to observe the results.

True labels:

```
['B-AC', 'B-O', 'B-LF', 'I-LF', 'B-AC', 'I-LF', 'I-LF', 'I-LF']
```

predictions of word2vec and FFNN:

```
['B-AC', 'B-O', 'B-AC', 'I-LF', 'B-AC', 'B-O', 'I-LF', 'I-LF']
```

predictions of word2vec and Random Forest:

```
['B-AC', 'B-O', 'B-AC', 'I-LF', 'B-AC', 'B-O', 'I-LF', 'I-LF']
```

predictions of word2vec and Simple RNN:

```
['B-LF', 'B-O', 'B-AC', 'I-LF', 'B-AC', 'B-O', 'I-LF', 'B-O']
```

Error analysis:

- As expected, FFNN model predicted I-LF as B-O and B-LF as B-AC.
- Random Forest model gave same predictions as FFNN model.
- Simple RNN model gave 4 wrong predictions. It predicted B-AC as B-LF and B-LF as B-AC and I-LF as B-O.

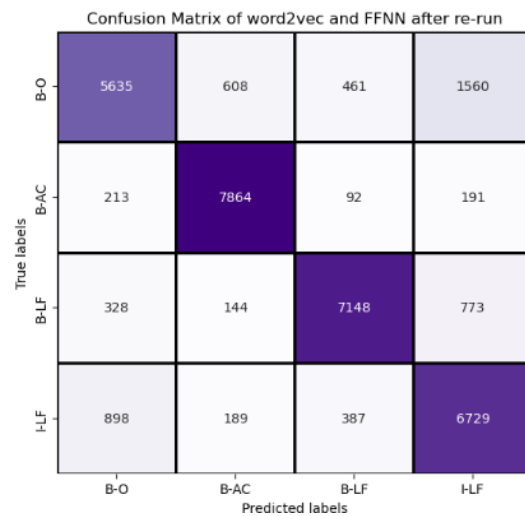
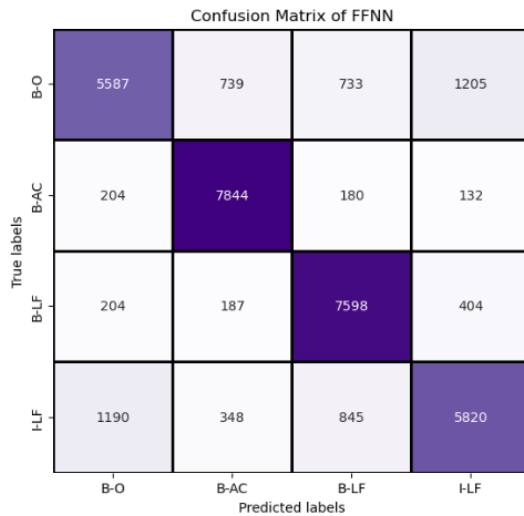
4 Discussion of best results from the testing

From the above experiments I have done, FFNN model and Random Forest Classifier with Word2Vec vectorization produced best results and gave better predictions for the input sequences. So, I did some variations and re-run the experiments.

First Variation: I have increased the number of epochs of training in FFNN model and expected to get better f1-score than previous model.

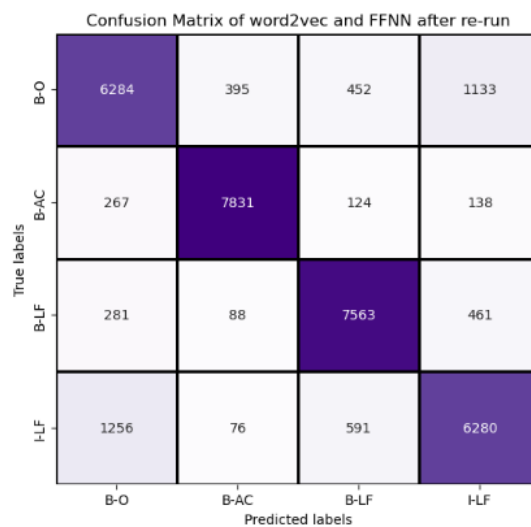
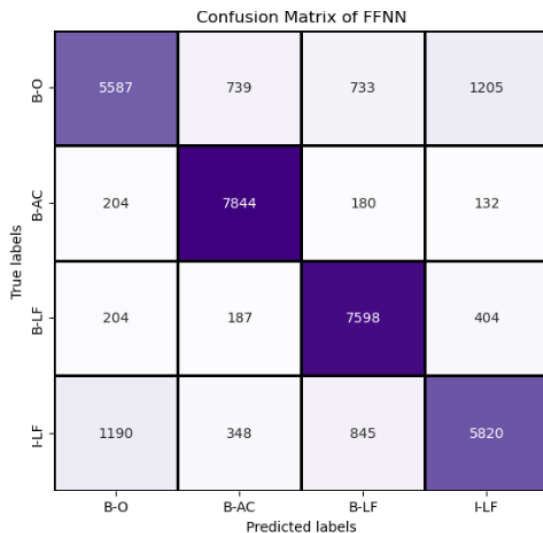
	Word2Vec and FFNN
F1-Score before	0.804
F1-Score After	0.822

After re-run of the Word2Vec and FFNN model, the true prediction values in the confusion matrix are increased for all the four labels and the model produced better predictions.



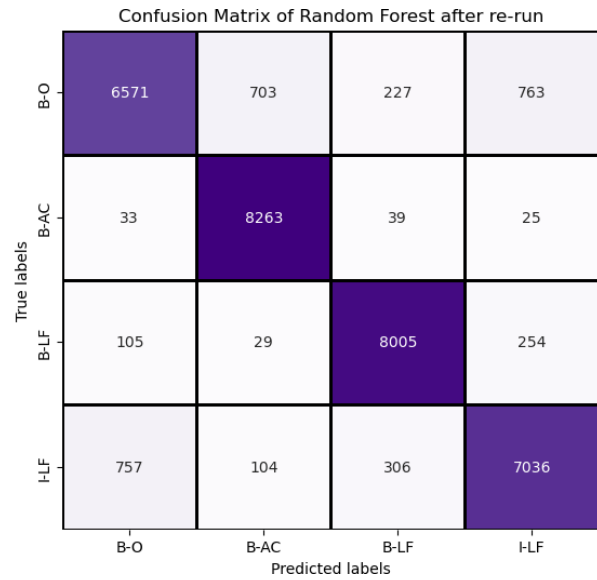
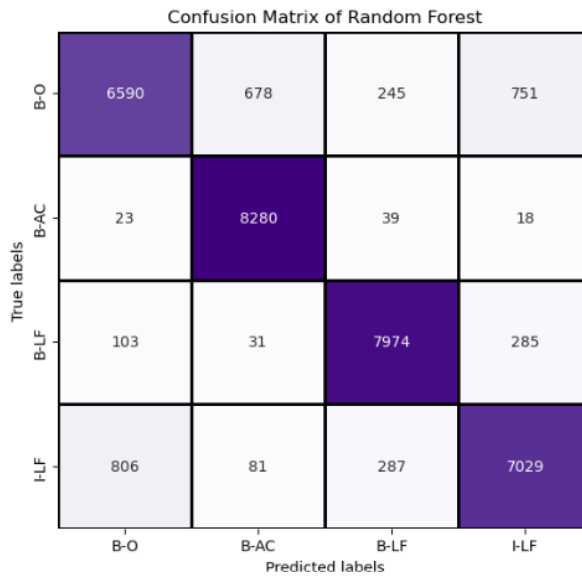
Second Variation: I increased the vector dimensions from 20 to vector size 100 in FFNN model and trained with 70 epochs and observed the results.

	Word2Vec and FFNN
F1-Score Before	0.804
F1-Score After	0.840



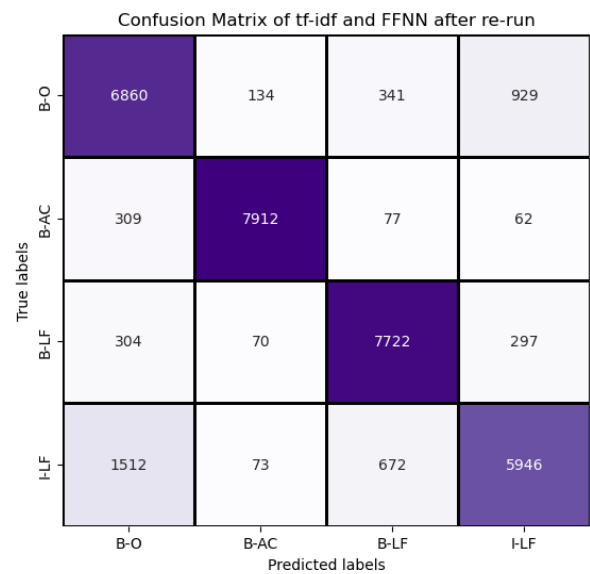
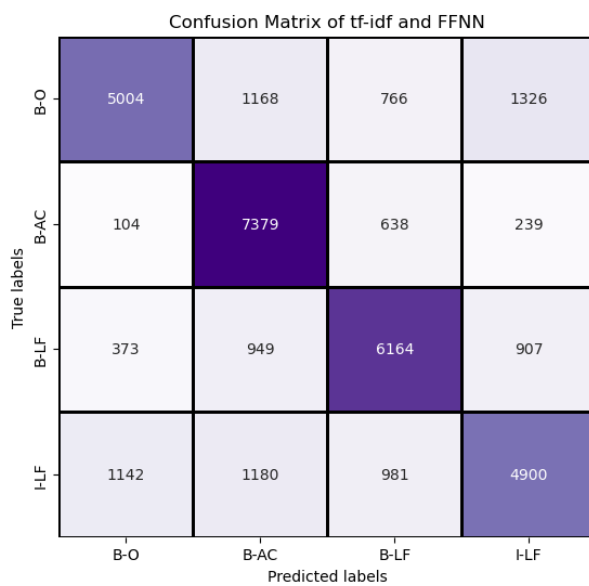
Third Variation: Increasing the maximum depth of Random Forest Classifier and re-run the experiment.

The **f1-score** remained same even after re-run of the experiment i.e., **0.897**. But there are changes in the confusion matrix. After running the experiment again, the true predictions of the model has increased compared to the previous model.



Fourth Variation: Increasing the number of training epochs for TF-IDF vectorization with FFNN model and there is slight increase in the f1-score and changes can be observed in confusion matrix.

	TF-TDF and FFNN
F1-Score Before	0.849
F1-Score After	0.855



5 Evaluation of overall attempt and outcome

5.a. In the task of sequence classification for detecting abbreviations and long forms, the models I tested served their purpose well. Notably, some models, such as the Feedforward Neural Network (FFNN) and the Random Forest Classifier using Word2Vec for vectorization, outperformed the other models in terms of predictions.

5.b. Accuracy is a measure of how many predictions a model got right, including both true positives and true negatives. It's a good measure when the class distribution is similar.

Conversely, the **F1-score** represents the harmonic mean between precision and recall. The F1-score is particularly useful when the class distribution is imbalanced. So, for this classification task F1-score is a good evaluation metric because,

- Balances Precision and Recall
- Handles imbalanced data

5.c To improve the performance of model:

- In the experimentation of comparing loss functions and optimizers, I observed that the model f1-score is very low when the optimizers were changed. So, 'Adam' optimizer is the good solution to perform classification tasks.
- Even though the f1-score for Random Forest and TF-IDF is high, the models did not produce better results as Word2Vec and FFNN model.
- Random Forest with Word2Vec vectorization has the high accuracy but it didn't perform well in the predictions. So, I had re-run the experiment by increasing the maximum depth and I observed that, even though the accuracy is same as before, the confusion matrix of model has improved but it didn't produce accurate predictions when compared to Word2Vec and FFNN model.

5.d As I mentioned before, '**Word2Vec and FFNN**' model performed well and gave good predictions with a smaller number of wrong predictions. I would try to make it more efficient by re-run the experiment again by changing the hyper-parameters. So, I changed the vector dimensions and trained the model with more number of epochs to give better results.

Before optimization, it predicted 2 labels wrong.

```
true predictions:
['B-O', 'B-O', 'B-AC', 'B-O', 'B-O', 'B-LF', 'I-LF', 'I-LF']

predictions of word2vec and FFNN:
['B-AC', 'B-AC', 'B-AC', 'B-O', 'B-O', 'B-LF', 'I-LF', 'I-LF']
```

After optimization, it predicted only one label wrong.

```
predictions of word2vec and FFNN:  
['B-O', 'B-O', 'B-O', 'B-O', 'B-O', 'B-LF', 'I-LF', 'I-LF']
```

Hence, I would choose the most effective model Word2Vec and FFNN compared to most accurate model Word2Vec and Random Forest.

6 References

[How to choose loss functions when training deep learning neural network](#)

machinelearningknowledge.ai/keras-optimizers

scikit-learn.org documentation

radimrehurek.com/gensim/examples

[nlp-preprocessing-steps-in-easy-way](#)

[multi-class-imbalanced-classification](#)

[vectorization-techniques-in-nlp-guide](#)

[confusion-matrix-in-machine-learning](#)

[implementing-feedforward-neural-networks-with-keras-and-tensorflow](#)

[what-are-n-grams-and-how-to-implement-them-in-python](#)