Natural Language Processing COMM061

Name – Pallav Kumar Bhardwaj

Student No -6846581

Stream -MSc Data Science

Group -36

Type- Individual Assignment

Analyse and visualize the Dataset

Analysing the Data

The PLOW Dataset has tokens and their short forms. These short forms are also called Acronyms. These Short forms can help facilitate text understanding.
These Dataset have been used in Biomedical Named Entity Recognition for handling Bio-Medical terms such as RNA, Protein, cell type, and DNA.For example we can use finding city names to find gene name.
NER recognition has a huge impact on the field of NLP and AI. Named Entity Recognition (NER) technique empowers machines not just to comprehend text but also to identify and categorize specific entities within it.
The Dataset contains POS and NER representation of tokens , NER forms has been abbreviated as B-O , B-LF ,B-AC and I-LF .
B-O (or 'O') indicates other tokens which are neither abbreviations nor long forms. B-AC signifies that the token is an abbreviation/acronym, while B-LF signifies that a long-form 'begins' with this token. The I-LF label signifies that the token is 'inside' of a long form.

I have Plotted the different datasets under the PLOW Dataset namely train, test, and validate against one another to analyse the different distribution of ner_tags.

Visualizing the Dataset

Train Dataset

```
df.head()
```

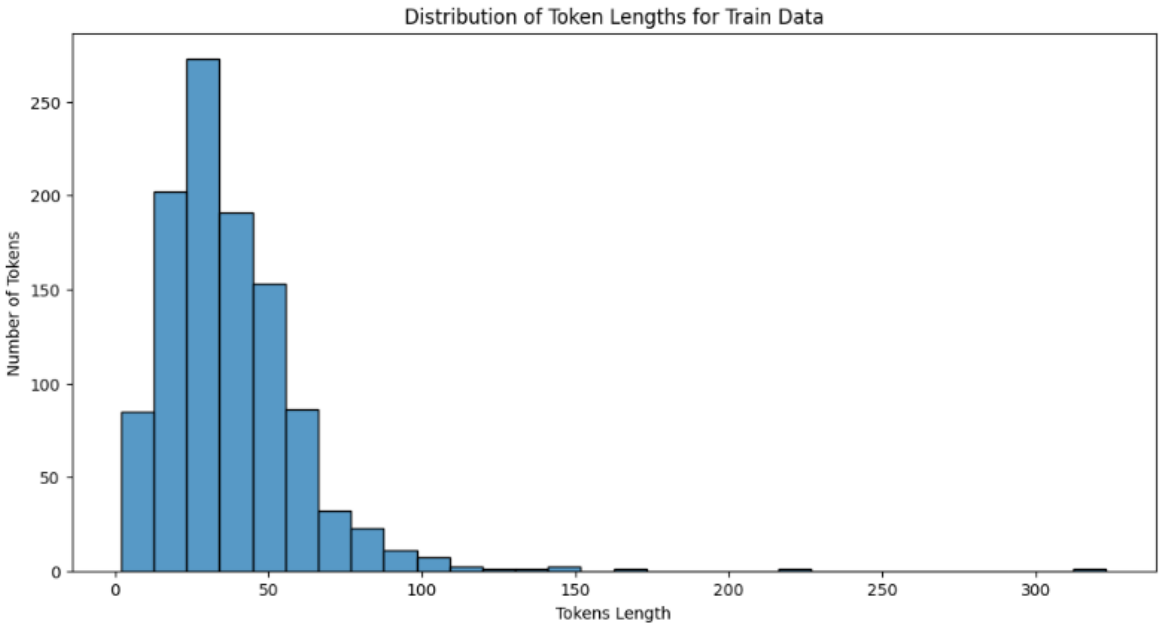
	tokens	pos_tags	ner_tags
0	[For, this, purpose, the, Gothenburg, Young, P...	[ADP, DET, NOUN, DET, PROPN, PROPN, PROPN, PRO...	[B-O, B-O, B-O, B-O, B-LF, I-LF, I-LF, I-LF, I...
1	[The, following, physiological, traits, were, ...	[DET, ADJ, ADJ, NOUN, AUX, VERB, PUNCT, ADJ, N...	[B-O, B-O, B-O, B-O, B-O, 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,...
3	[EPI, =, Echo, planar, imaging, .]	[PROPN, PUNCT, NOUN, NOUN, NOUN, PUNCT]	[B-AC, B-O, B-LF, I-LF, I-LF, B-O]

Plotting Data for better Visualization

Plot 1

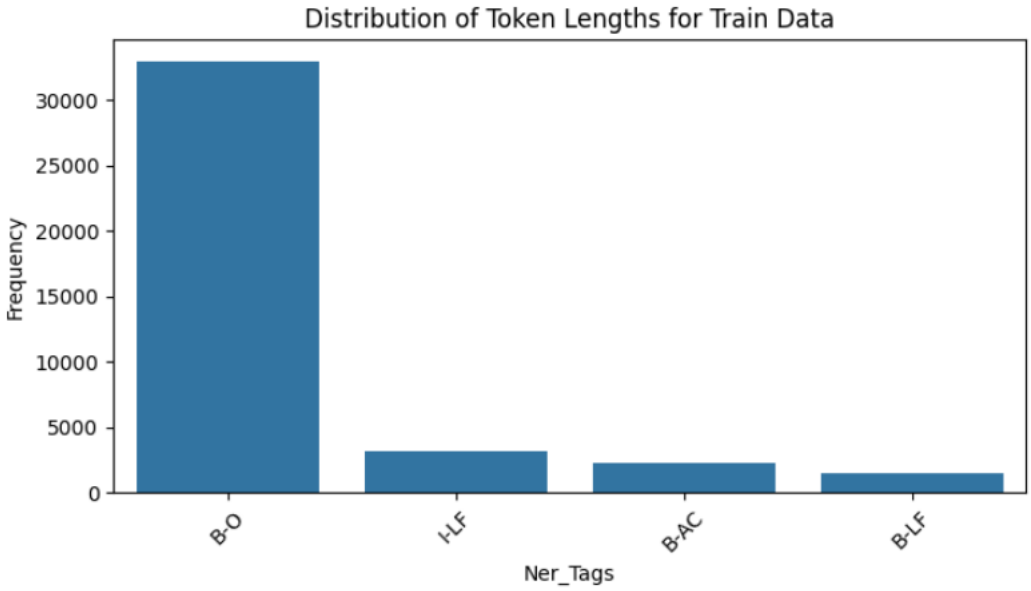
-We deploy histograms for all the three datasets to analyse the distribution of tokens as per their length. The Y-axis has the Number of tokens and X-axis has tokens length. For the Train data, we have most of the token sentence length in the range of 20-100, with maximum token sentence length in between 20-30. The distribution is skewed to the right , its means that there are more sentences in the token column with shorter lengths. There are few outliers that have token lengths greater than 100. This helps us to understand the nature of the dataset that we are going to work on.

For the test dataset, the is in the range of 80 which means the distribution is more homogenous as compared to the train dataset with no outliers. The validation set contains a few outliers with token lengths greater than 100.



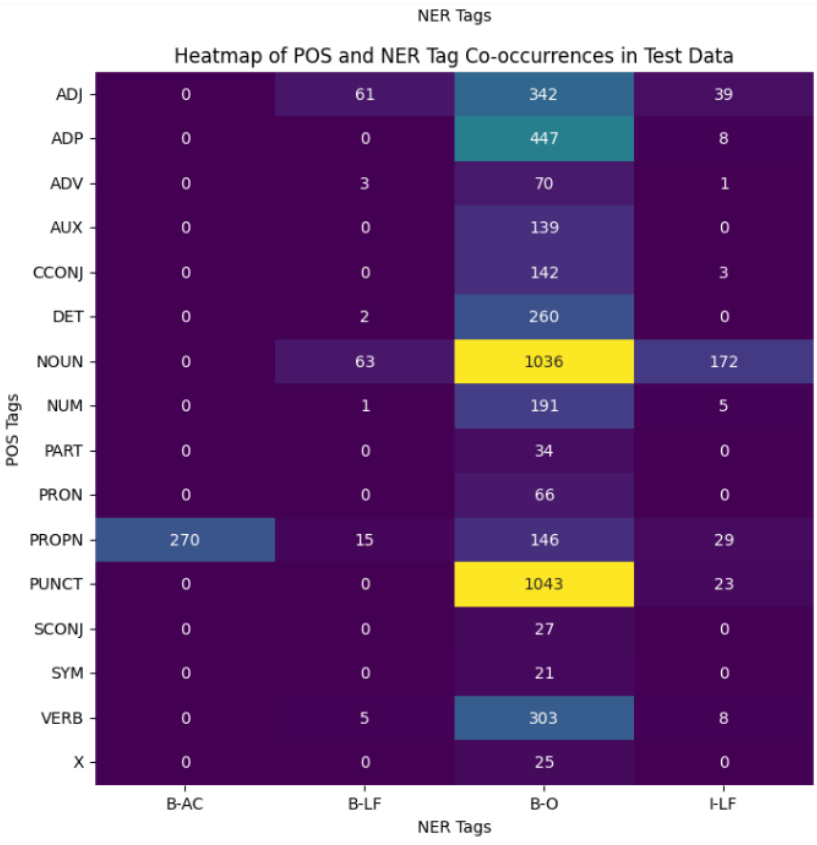
Plot2

- In this, we have created a bar chart for all three sub-datasets. In the Y-axis we have the frequency of tags and in the X-axis we the different short forms for the tokens. In all 3 datasets namely Train ,test and Validation B-O tag has highest frequency which means it has frequent occurrence as compared to I-LF , B-AC and B-LF. Due to this the dataset looks disbalanced.



Plot3

-In this plot we have implemented a heatmap of POS tag and NER Tag occurrences in 3 different dataset. For Training Data the intersection of NOUN and B-O has very high value which means that the dataset has huge B-O occurrence which and these are mostly Nouns. The most frequent co-occurrence in this dataset is between PUNCT (punctuation) and B-O, as indicated by the cell with the highest number (8618) and the brightest colour on the scale.



2. Experimentation with Four Different Experimental Setups

➢ 2.1 Experiment - Data pre-processing techniques

-Setup - We installed and loaded the dataset surrey-nlp/PLOD-CW from HuggingFace. Installed necessary libraries such as pandas, matplotlib, Nltk, Spacy, and Sklearn. Imported different classes such as TfidfVectorizer, Porter Stemmer and SVC .

As the PLOD-CW has three different datasets namely Train, Test and Validation, we load them in different dataset that we will be using in creating different model/systems.

For the first two systems/models, I have also flattened different datasets to get one value per row that will be easy to analyze and encode. Shapes of the dataset after it has flattened changes to Train 40000 x2 , Test 5000x2and Validation 5000x2.

For the 3rd System, I have used the dataset as given because Bigram needs to create pairs, which cannot be done in the flattened dataset.

```
flat_df.head()
```

	Token	Ner_Tag
0	For	B-O
1	this	B-O
2	purpose	B-O
3	the	B-O
4	Gothenburg	B-LF

```
flat_df_test.head()
```

	Token	Ner_Tag
0	Abbreviations	B-O
1	:	B-O
2	GEMS	B-AC
3	,	B-O
4	Global	B-LF

```
flat_df_valid.head()
```

	Token	Ner_Tag
0	=	B-O
1	Manual	B-LF
2	Ability	I-LF
3	Classification	I-LF
4	System	I-LF

As we are going to preprocess data we have loaded Spacy English model "en_core_web_sm" and also created an instance of the class PorterStemmer that will help us in Stemming.

```
print(flat_df.shape)
print(flat_df_test.shape)
print(flat_df_valid.shape)
```

```
(40000, 2)
(5000, 2)
(5000, 2)
```

System 1: Preprocessing (Stemming and Lemmatization), Tfidf Vectorization, KNN

Data-Preprocessing - We have defined a function Preprocess that takes a token and applies Lemmatization using lemma_ under spacy pipeline and Stemming using PorterStemmer, it also returns an NA value if the returned token is empty.

We will use these NA values to remove the drop the rows that has NA values in them , which will eventually lead to a Smaller dataset than before.

Train Data

```
Shape Before (40000, 2)
Shape After (30630, 3)
```

Test Data

```
Shape Before (5000, 2)
Shape After (3884, 3)
```

Validation Data

```
Shape Before (5000, 2)
Shape After (3875, 3)
```

```
flat_df1_clean.head()
```

	Token	Ner_Tag	text_for_tfidf
0	For	B-O	for
1	this	B-O	thi
2	purpose	B-O	purpos
3	the	B-O	the
4	Gothenburg	B-LF	gothenburg

```
flat_df1_test_clean.head()
```

	Token	Ner_Tag	text_for_tfidf
0	Abbreviations	B-O	abbrevi
2	GEMS	B-AC	gem
4	Global	B-LF	global
5	Enteric	I-LF	enter
6	Multicenter	I-LF	multicent

Text Encoding - We are using Tfidf Vectorization method to vectorize the token all of the three Dataset after Cleaning. This vectorization method helps vectorizing the tokens using Term Frequency and Inverse Document frequency. Tfidf works good with this dataset as its values increases proportionally with no of times occurrence of the word. Tfidf creates a huge vector space as it uses one dimension per term in the vocabulary. After cleaning of Data and vectorization we got 3 sparse matrix namely X_train_tfidf, X_test_idf, and X_valid_tfidf.

```
X_train_tfidf
```

```
<30630x6333 sparse matrix of type '<class 'numpy.float64'>'
```

```
X_test_tfidf
```

```
<3884x6333 sparse matrix of type '<class 'numpy.float64'>'
```

```
X_valid_tfidf
```

```
<3875x6333 sparse matrix of type '<class 'numpy.float64'>'
```

Algorithm Used - In this system we are using the KNN (K-Nearest Neighbour) Algorithm . It's a hugely popular algorithm in Supervised Machine learning as it does analysis and training based on nearest neighbour of the data point. To classify a new data point it measures the distance from this new instance to all other instances in the training set. For measuring distance it uses Euclidean distance, Manhattan, and Hamming distance. KNN usually doesn't make any assumption based on underlying data so it can work with unseen data better than other algorithms. Output produced during testing is y_pred, which a list and to evaluate our model we will compare y_test to y_pred.

```
[237] from sklearn.neighbors import KNeighborsClassifier
      knn_model = KNeighborsClassifier(n_neighbors=10)
```

```
[238] knn_model.fit(X_train_tfidf, y)
```

```
KNeighborsClassifier
KNeighborsClassifier(n_neighbors=10)
```

```
print(y_pred)
```

```
['B-O' 'B-O' 'B-O' ... 'B-O' 'B-O' 'B-O']
```

```
df_predicted.head(50)
```

	X_test	y_test	y_pred
0	abbrevi	B-O	B-O
2	gem	B-AC	B-O
4	global	B-LF	B-O
5	enter	I-LF	B-O
6	multicent	I-LF	B-O
7	studi	I-LF	B-O
9	vip	B-AC	B-O

Parameters Used - We have used no of neighbours parameter as 10, usually KNN algorithms don't work well if we choose a very high or very low no of neighbours.

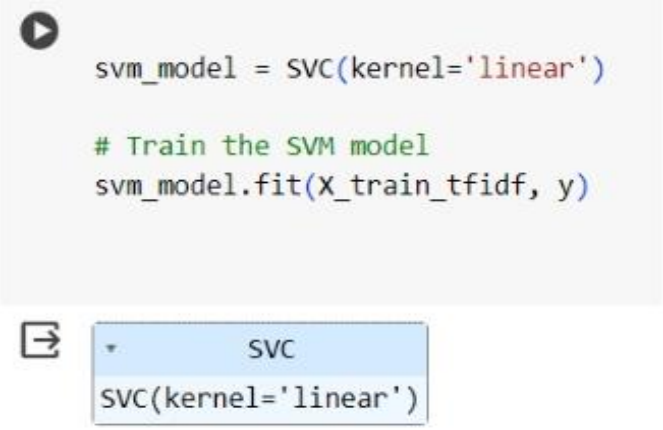
System 2: Preprocessing (Stemming and Lemmatization), Tfidf Vectorization, SVM

Data-Preprocessing - We have used the same data-preprocessing technique as above model/system using the preprocess function, which helps in stemming and lemmatization. After that, we cleaned the dataset by dropping NA values.

Text Encoding - We have vectorized the same as the above model using Tfidf, which has created a sparse matrix.

Algorithm Used - Here we have used SVM (Support Vector Machine) which is a versatile supervised machine learning algorithm . Its capable of both linear and non-linear classification. It creates a Hyperplane in N-dimensions using Support vectors. Support vectors are the data points that lie closest to the hyperplane. It's very effective when there is margin of separation between classes is clear.

SVM takes the above-mentioned X_train_tfidf and y to train the model . It does prediction on X_test and produces y_pred which is shown in the below images.



```
[182] df_predicted_SVM.head(50)
```

	X_test	y_test	y_pred
0	abbrevi	B-O	B-O
2	gem	B-AC	B-O
4	global	B-LF	B-O
5	enter	I-LF	B-O
6	multicent	I-LF	B-O
7	studi	I-LF	B-O
9	vip	B-AC	B-O
11	ventil	B-LF	B-O
12	improv	I-LF	B-O
13	pit	I-LF	B-O
15	fraction	B-O	B-O

Parameters Used - We have used a linear kernel as it finds a straight line in 2-D, a plane in 3-d, and a hyperplane for larger dimensions, which helps to separate the classes by maximum margin.

System 3: Preprocessing (Bigram), Tfidf Vectorization, KNN

Data-Preprocessing and Encoding - We haven't used the flattened dataset for this as its not suitable for bigrams. For tokens we have used Tfidf vectorizer with bigrams that vectorize the pairs of tokens together. In below image we have multiple 0's in the same column because its using bigram and for every token in the list we can have multiple vectors .

For our target label Ner_tags we have used Multibinarizer which produces a matrix of 1x4 as we have 4 different labels in our ner representation B-O ,B-AC , B-IF and B-lF , if the matrix contains [1,1 ,0,1] it means in that list B-IF is not present .We used eval function evaluates a Python expression from a string form and convert back string list.

```
print(X)
```

```
  (0, 21574)    0.2774330267347462
  (0, 8694)     0.306605398110339
  (0, 17162)    0.306605398110339
  (0, 6810)     0.306605398110339
  (0, 14951)    0.306605398110339
  (0, 22457)    0.306605398110339
  (0, 8553)     0.306605398110339
  (0, 19344)    0.306605398110339
  (0, 15951)    0.306605398110339
  (0, 20049)    0.306605398110339
```

```
print(y_train_binarized )
```

```
[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 ...
 [1 1 1 1]
 [1 1 1 1]
 [0 0 1 0]]
```

Algorithm Used -

We have used KNN algorithm for this as well. It takes the Vectorized token and Multibinarizer y_train and produces y_pred in a Multibinarizer form , which can be used to do evaluation against y_test_binarized.

```
[234] print(y_pred)
```

```
[[1 1 1 1]
 [1 0 1 0]
 [1 1 1 1]
 [1 1 1 1]
 [1 0 1 0]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 0]
 [1 0 1 0]
 [1 1 1 1]
 [1 1 1 0]]
```

```
[236] print(y_test_binarized[:10])
```

```
[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 [1 0 1 0]
 [1 1 1 0]
 [1 1 1 1]
 [1 1 1 1]
 [1 0 1 0]
 [0 0 1 0]
 [1 1 1 1]]
```

.

Parameters Used - We have used the no of neighbours parameter as 25 , as it improves the accuracy compared to System1.

➢ 2.2 Experiment - NLP algorithms/techniques

System 1: No Preprocessing, FastText Vectorization, SVM

Setup - For this model/system we will be using our flattened dataset used previously in the experiment. We have 40000 rows in Training data, and 5000 rows in each Test and Validation data.

For text encoding/vectorization we are using FastText vectorizer. FastText captures the morphological structure of words so it should sync with the SVM Algorithm later used in the model. FastText will help us analyze rare tokens due their robust nature.

We have defined a FastText model that takes tokens and creates embeddings of size 100.Then it vectorizes the token using the defined Vectorize function using the FastText model.

Algorithm used- We are using SVM here. The fundamental idea in SVM is that it creates a hyperplane that will divide the dataset into 2 classes. The SVM model position the Hyperplane in such a way that maximizes the distance from the nearest data points of any class. Different SVM kernels that can be used are Linear, Polynomial, Radial basis Functions etc.

```
[78] svm_classifier = SVC(kernel='linear')

     # Train the SVM model
     svm_classifier.fit(X, flat_df1['Encoded_Tag'])
```

```
SVC
SVC(kernel='linear')
```

In this model SVM classifier takes X which is a NumPy array generated after vectorization using FastText and Flat_df1['Encoded_Tag'] which is generated after Label encoding the ner_tags in df_falt1.

Then we use the same classifier model on X_test which is a numpy array created after vectorizing the test dataset and generates y_pred , which is used to evaluates the model on comparing it to y_test.

Parameters Used - We have used a linear kernel as it finds a straight line in 2-D, a plane in 3-d and a hyperplane for larger dimensions, which helps to separate the classes by maximum margin.

System 2 No preprocessing, Encoding, LSTM

Setup - For this model/system we will be using our flattened dataset used previously in the experiment. We have 40000 rows in Training data, and 5000 rows in each Test and Validation data.

For text encoding/vectorization we are using the tokenizer provided under Keras-preprocessing. We have also defined OOV_token (Out_of_vocabulary) as it will help us to handle words not seen in the training data. If any token that it will consider Out_of_vocabulary will be replaced with 'UNK' that is Unknown and move ahead with processing. Then we will fit it on tokens. Then we will convert these tokenized words to sequences so we use it our LSTM Algorithm. For this, we have also padded the sequence using the Pad_sequences function.

```
print(X_trainp)
```

```
[[   9    0    0 ...    0    0    0]
 [  31    0    0 ...    0    0    0]
 [3417    0    0 ...    0    0    0]
 ...
 [ 663    0    0 ...    0    0    0]
 [  17    0    0 ...    0    0    0]
 [ 207    0    0 ...    0    0    0]]
```

For our ner_tags we use Label Encoder to encode them and using the to_categorical function on the encoded tags. This is a function that takes integer-encoded labels as input and converts each integer into a binary vector of a length equal to the number of classes in the data.

```
print(y_trainc)
```

```
[[0. 0. 1. 0.]
 [0. 0. 1. 0.]
 [0. 0. 1. 0.]
 ...
 [0. 0. 1. 0.]
 [0. 0. 1. 0.]
 [0. 0. 1. 0.]]
```

Algorithm used - In this model/system we are using LSTM(Long Short_Term Memory) which comes under RNN. It is capable of learning long-term dependencies in Data sequences. LSTM can perform better than conventional RNN as it can model sequence and Time series Data more effectively.

For the below LSTM sequential model we set the input dimension as size the size of vocabulary. Output_dim as 64 which means each vector will be mapped to a vector of size 64. input_length as 50 which implies that each input sequence to the model will

consists of 50 tokens. For LSTM layer we have 64 units. y_trainc.shape[1] sets the no of output neurons to match the no of classes in target data.

The activation function used here is SoftMax which is appropriate for multi-classification problems.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense

model = Sequential()
model.add(Embedding(input_dim=len(tokenizer.word_index) + 1, output_dim=64, input
model.add(LSTM(64, return_sequences=False))
model.add(Dense(y_trainc.shape[1], activation='softmax'))

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accura
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_2 (Embedding)     (None, 50, 64)            515968

 lstm_2 (LSTM)               (None, 64)                33024

 dense_2 (Dense)             (None, 4)                 260

=================================================================
Total params: 549252 (2.10 MB)
Trainable params: 549252 (2.10 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Then we fit the data using X_trainp which is a padded sequence and y_trainc which is encoded ner_tags to create a model using epoch as 5 and batch size as 20 . With this model, we will predict y_predc and evaluate it against y_testc.

➢ 2.3 Experiment - text encoding/transformation into numerical vectors

➢

Setup -We installed the dataset `surrey-nlp/PLOD-CW` from HuggingFace and loaded it into our environment. We also set up the necessary libraries, including pandas, matplotlib, NLTK, spaCy, and scikit-learn. To process the data, we imported specific classes such as `TfidfVectorizer`, `PorterStemmer `and `SVC`.

The `PLOD-CW` dataset consists of three subsets: Train, Test, and Validation, which we loaded into separate datasets for use in building various models or systems. For the first two systems, I flattened the datasets to achieve a single value per row, simplifying analysis and encoding. As a result of this flattening, the dimensions of the datasets changed to 40,000 x 2 for Train, 5,000 x 2 for Test, and 5,000 x 2 for Validation.

System 1 No preprocessing, Tfidf Vectorization, KNN

Data-preprocessing - No Data preprocessing technique has been used in this model .

Text Encoding/transformation

We are using tfidf vectorization in model, for that we will create a instance of the TfidfVectorizer class and use it to convert it into numerical vectors. Tfidf works by calculating Term frequency and inverse document frequency. Term frequency measures the no of times the word appears in the document.

Inverse document frequency measures the importance or the weight of term in corpus. Mathematically they can be represented below

$$\mathrm{TF}(t,d) = \frac{\text{Number of times term } t \text{ appears in document } d}{\text{Total number of terms in document } d}$$

$$\mathrm{IDF}(t,D) = \log\left(\frac{N}{|\{d \in D : t \in d\}|}\right)$$

$$\text{TF-IDF}(t,d,D) = \mathrm{TF}(t,d) \times \mathrm{IDF}(t,D)$$

We can use different parameters such as max_df which ignore terms that have a document frequency higher that the given threshold. Other parameters that can be used with Tfidf is min_df, max_features, norm etc.

We take the df_flat1['tokens'] and vectorize it using instance of TfidfVectorizer class .Similarly we do it for test and validation data. It creates a sparse matrix as shown below.

```
[18] X_train_tfidf

     <40000x7869 sparse matrix of type '<class 'numpy.float64'>'
```

```
[19] X_test_tfidf

<5000x7869 sparse matrix of type '<class 'numpy.float64'>'

[ ] X_valid_tfidf

<5000x7869 sparse matrix of type '<class 'numpy.float64'>'

print(X_train_tfidf)

  (0, 2931)     1.0
  (1, 7229)     1.0
  (2, 5920)     1.0
  (3, 7206)     1.0
  (4, 3181)     1.0
  (5, 7810)     1.0
  (6, 5430)     1.0
  (7, 2490)     1.0
  (8, 6451)     1.0
  (10, 3246)    1.0
  (12, 7689)    1.0
  (13, 2077)    1.0

[ ] print(y[:10])

0     B-O
1     B-O
2     B-O
3     B-O
4    B-LF
5    I-LF
6    I-LF
7    I-LF
8    I-LF
9     B-O
```

For ex if (21, 3772) 1.0 is output above that 21st word is represented by index 3772 that has a Tfidf score of 1.

Algorithm used

KNN model takes X_train_tfidf and y as input and fits the model. It tests on X_test produces y_pred and then we evaluate it with comparison to y_test.

```
df_predicted_KNN.head()
```

	X_test	y_test	y_pred
0	Abbreviations	B-O	B-O
1	:	B-O	B-O
2	GEMS	B-AC	B-O
3	,	B-O	B-O
4	Global	B-LF	B-O

Parameters Used - We have used no of neighbours paarmeter as 25 , usually KNN algorithms doesn't work well if we choose very high or very low no of neighbours.

System 2 No preprocessing , FastText Vectorization , KNN

Data-preprocessing - No Data preprocessing technique has been used in this model .

Text Encoding/transformation

In this model FastText is used as a Vectorization method . FastText is popular vectorization technique as it enhances the Word2vec idea, rather than learning vectors only for whole words, FastText represents each word as a bag of character n-grams. FastText can also better technique when it comes to representing array words as it decomposes word into smaller units.

FastText is available under Genism models, we import it from there and use for our model. To use Fasttext, we need to create a model that helps us to vectorize our tokens.

```
# FastText model
Fasttext_model= FastText(tokens, vector_size=100, window=5, min_count=1, workers=4)
```

Here the model takes in tokens , vector_size as 100 is the embedding size that is defined. Window =5 defines the maximum distance that can be predicted and the current token. Min_count=1 sets the minimum frequency count of words. It helps in ignoring rare words. Workers=4 helps speed up the training process by inducing parallelization.

We use our tokens in flat_df1 and model to create vectors. The below image shows converted vectors.

```
[85] print(X)

[[ 1.04379340e-03 -1.88685150e-03 -1.92178669e-03 ...  1.18104217e-03
  -3.36294505e-03  2.44610314e-03]
 [ 1.14076433e-03  3.34119657e-03 -1.33493869e-03 ... -2.37689601e-05
   3.40386271e-03 -1.15793686e-04]
 [ 3.21004121e-03  8.01503717e-04 -3.43976752e-03 ...  2.50496785e-03
  -1.58503361e-03 -1.72686053e-03]
 ...
 [-3.34261102e-04  4.70247032e-04  6.65784115e-04 ... -8.79905419e-04
   3.83111270e-04  1.59678119e-03]
 [-8.05429241e-04  9.27802525e-04 -2.65061011e-04 ...  7.88405712e-04
   3.02777207e-03  2.59733293e-03]
 [-3.35044321e-03  3.15338722e-03  5.54983877e-03 ... -6.92472607e-03
  -5.12847537e-03 -1.01963058e-03]]
```

We have use LabelEncoder to encode our ner_tags as y and train them using KNN model.

Algorithm used

KNN model takes X and y as input and fit the model . It test on X_test produces y_pred and then we evaluate it with comparison to y_test.

Parameters Used - We have used no of neighbours parameter as 25 , to compare both vectorization models later.

- **Experiment 2.6 Hyper-parameter optimization**

System1 No preprocessing, Tfidf, SVM (Hyperparameter Optimization)

Setup - Here we are using the given df_train, df_test and df_valid dataset as loaded.

Data-preprocessing - No Data pre-processing technique such as lemmatization or stemming used here. We join the token list in df_train, df_test, and df_valid and apply Vectorization.

Vectorization- I have used Tfidf vectorization in this model with max_features as 5000. We save the vectorized data as X_train, X_test and Valid respectively. Below is the vectorized X_train.

We have used ner_tags as target value and saved the ner_tags in Y_train, Y_test and Y_valid. We have encoded it using Multibinarizer that give us Y_train as 4x1 vectors that indicates the presence of certain labels in our data.

```
[49]: print(y_train_binarized[:10])

      [[1 1 1 1]
       [1 1 1 1]
       [1 1 1 1]
       [1 1 1 1]
       [1 1 1 1]
       [1 0 1 0]
       [1 0 1 0]
       [1 1 1 1]
       [1 1 1 1]
       [1 1 1 1]]
```

Below is Vectorized X_train

```
[48]: print(X_train[:10])

  (0, 1511)     0.35337378350180654
  (0, 1875)     0.17526881289553137
  (0, 2922)     0.4157488441830654
  (0, 3361)     0.47556360652815105
  (0, 3892)     0.3773785775769426
  (0, 4345)     0.09952399408663466
  (0, 4361)     0.2398456304763198
  (0, 4821)     0.18403251985445085
  (0, 4942)     0.44909515253527005
  (1, 532)      0.04967714296321862
  (1, 1227)     0.13634789692625202
  (1, 1254)     0.3955763810768054
  (1, 1304)     0.16482472055295558
  (1, 1873)     0.15437390238837814
  (1, 1988)     0.20641998854251215
  (1, 1999)     0.4128399770850243
  (1, 2518)     0.15103661686701095
  (1, 2616)     0.20641998854251215
  (1, 2696)     0.19109284784829908
  (1, 2950)     0.20641998854251215
  (1, 3023)     0.21858582453318098
```

Algorithm and Hyperparameters- We have used SVM algorithm with hyper-parameter optimization and with one-vs -Rest strategy with Linear kernel.

We have created a Parameter Grid with estimator_c value as [1] these are ‘C’ values that will be used during GridSearchCV. The lower value of C will create a wider margin but will allow more misclassification i.e. underfitting but a larger C will cause the issue of overfitting. Another parameter we defined here is "estimator__kernel” which defines a list of the kernel that will be used to create the best model. "estimator__gamma" is kernel coefficient for ‘RBF’ kernel.

We have created a GridSearchCV object is created with the defined model, parameter grid, 5-fold cross-validation (cv=5), and accuracy as the scoring metric. We have trained the Grid_search with above defined X_train and y_train_binarized.

In our model we used the estimator_c as 1, estimator__gamma as scale and estimator kernel as RBF. Hyperparameters which are used here, we found out by rerunning the model. The best Cross validation score for our model is 0.75.

```
Best parameters: {'estimator__C': 1, 'estimator__gamma': 'scale', 'estimator__kernel': 'rbf'}
Best cross-validation score: 0.75
```

System 2 - No preprocessing, Tfidf , KNN(Hyperparameter Optimization)

Setup -

We have same setup as above system 1 and used the same tfidf vectorizer with max_features =5000. We applied the eval function on the dataset to convert it to string values which will be easy for further analysis.

Algorithm and Hyperparameters- We have used KNN algorithm here which works on find K -nearest neighbour for our data points. We have created a Parameter grid for our model where n_neighbours range will be (0,100) , for weights is uses distance and metric it had make choice between Euclidean , Manhattan and Minkowski. We have created a GridSearchCV object is created with the defined model, parameter grid, 5-fold cross-validation (cv=5), verbose =1 and accuracy as the scoring metric. We have trained the Grid_search with above defined X_train and y_train_binarized.

In our model, we used the metric as 'Euclidean' ,'n_neighbours ' =51 and weights as Distance. Hyperparameter which is used here, we found out by rerunning the model. The best Cross validation score for our model is 0.75.

```
Fitting 5 folds for each of 72 candidates, totalling 360 fits
Best parameters: {'metric': 'euclidean', 'n_neighbors': 51, 'weights': 'distance'}
Best cross-validation score: 0.75
```

3. Analyse Testing for Each Experiment Variation

3.1 Testing Results:

Experiment2.1 Data pre-processing techniques

System 1: Preprocessing (Stemming and Lemmatization), Tfidf Vectorization, KNN

Classification Report

```
[40] y_pred = knn_model.predict(X_test_tfidf)

     # Generate a classification report to evaluate the model
     report = classification_report(y_test, y_pred)

     print(report)
```

```
                precision    recall  f1-score   support

         B-AC       0.80      0.12      0.21       270
         B-LF       0.35      0.05      0.09       150
          B-O       0.84      0.99      0.91      3201
         I-LF       0.62      0.14      0.22       263

     accuracy                           0.83      3884
    macro avg       0.65      0.33      0.36      3884
 weighted avg       0.80      0.83      0.78      3884
```

Model Metrics

```
Accuracy: 0.8344490216271885
Precision: 0.6536089474615397
Recall: 0.3252196992463936
F1 Score: 0.3594517351618455
Hamming Loss: 0.16555097837281155
```

System 2: Preprocessing (Stemming and Lemmatization), Tfidf Vectorization, SVM

Classification Report

```
y_pred1 = svm_model.predict(X_test_tfidf)


report = classification_report(y_test, y_pred1)

print(report)
```

```
              precision    recall  f1-score   support

        B-AC       0.78      0.35      0.48       270
        B-LF       0.28      0.06      0.10       150
         B-O       0.86      0.96      0.91      3201
        I-LF       0.37      0.19      0.25       263

    accuracy                           0.83      3884
   macro avg       0.57      0.39      0.44      3884
weighted avg       0.80      0.83      0.80      3884
```

Model Metrices

```
Accuracy: 0.8341915550978373
Precision: 0.57329222075495
Recall: 0.3906620864809489
F1 Score: 0.4351042397293068
Hamming Loss: 0.16580844490216273
```

System 3: Preprocessing (Bigram), Tfidf Vectorization, KNN

Classification Report

```
[72]

    # Generate a classification report to evaluate the model
    report = classification_report(y_test_binarized, y_pred)

    print(report)
```

```
                precision    recall  f1-score   support

             0       0.86      0.98      0.91       131
             1       0.76      0.86      0.80       113
             2       1.00      1.00      1.00       153
             3       0.66      0.76      0.71       100

     micro avg       0.83      0.91      0.87       497
     macro avg       0.82      0.90      0.86       497
  weighted avg       0.84      0.91      0.87       497
   samples avg       0.84      0.93      0.84       497
```

Model Metrices

```
Accuracy: 0.6111111111111112
Precision: 0.8974867724867724
Recall: 0.9292328042328042
F1 Score: 0.8823885109599398
Hamming Loss: 0.16071428571428573
```

Experiment2.2 NLP algorithms/techniques

System 1: No Preprocessing, FastText Vectorization, SVM

Classification report

```
y_pred1 = svm_classifier.predict(X_test)


report = classification_report(y_test, y_pred1)

print(report)
```

```
              precision    recall  f1-score   support

           0       0.00      0.00      0.00       270
           1       0.00      0.00      0.00       150
           2       0.86      1.00      0.92      4292
           3       0.00      0.00      0.00       288

    accuracy                           0.86      5000
   macro avg       0.21      0.25      0.23      5000
weighted avg       0.74      0.86      0.79      5000
```

Model Metrics

```
Accuracy: 0.8584
Precision: 0.2146
Recall: 0.25
F1 Score: 0.23095135600516575
Hamming Loss: 0.1416
```

System 2 No preprocessing, Encoding, LSTM

Classification report

```
report = classification_report(y_true, y_pred, target_names=label_encoder.classes
print(report)
```

```
              precision    recall  f1-score   support

       B-AC       0.00      0.00      0.00       270
       B-LF       0.00      0.00      0.00       150
        B-O       0.86      1.00      0.92      4292
       I-LF       0.00      0.00      0.00       288

   accuracy                           0.86      5000
  macro avg       0.21      0.25      0.23      5000
weighted avg       0.74      0.86      0.79      5000
```

Model Metrics

```
Accuracy: 0.8584
Precision: 0.2146
Recall: 0.25
F1 Score: 0.23095135600516575
Hamming Loss: 0.1416
```

Experiment 2.3 - text encoding/transformation

System 1 No preprocessing, Tfidf Vectorization, KNN

Classification report

```
              precision    recall  f1-score   support

       B-AC       1.00      0.01      0.03       270
       B-LF       0.25      0.03      0.05       150
        B-O       0.86      0.99      0.92      4292
       I-LF       0.38      0.05      0.08       288

   accuracy                           0.86      5000
  macro avg       0.62      0.27      0.27      5000
weighted avg       0.82      0.86      0.80      5000
```

Model Metrics

```
Accuracy: 0.8568
Precision: 0.6235653885492733
Recall: 0.2699659034034034
F1 Score: 0.2702655583690421
Hamming Loss: 0.1432
```

System 2 No preprocessing, FastText Vectorization , KNN

Classification Report

```
print(classification_report(y_test, y_pred, target_names=label_encoder.classes_))
```

	precision	recall	f1-score	support
B-AC	0.67	0.01	0.03	270
B-LF	0.36	0.05	0.09	150
B-O	0.87	0.99	0.92	4292
I-LF	0.51	0.11	0.19	288
accuracy			0.86	5000
macro avg	0.60	0.29	0.31	5000
weighted avg	0.82	0.86	0.81	5000

Model Metrics

```
Accuracy: 0.8594
Precision: 0.601128139573218
Recall: 0.29335295424735075
F1 Score: 0.308356477796528
Hamming Loss: 0.1406
```

Experiment2.6_Hyper-Parameter Optimization

System 1 - No preprocessing, Tfidf, SVM (Hyperparameter Optimization)

Model Metrics

```
Accuracy: 0.6797385620915033
Precision: 0.8349673202614379
Recall: 0.9967320261437909
F1 Score: 0.8780578898225958
Hamming Loss: 0.16666666666666666
```

System 2 - No preprocessing, Tfidf, KNN(Hyperparameter Optimization)

Model Metrics

```
Accuracy: 0.673202614379085
Precision: 0.8306100217864923
Recall: 0.9967320261437909
F1 Score: 0.8747899159663867
Hamming Loss: 0.16993464052287582
```

3.2 analysis of test result using F1 score

The F1-score is a statistical metric utilized to assess the accuracy of classification models. It is especially valuable in scenarios where there is an uneven distribution of classes and different types of errors carry varying consequences. The F1-score is calculated as the harmonic mean of precision and recall, effectively balancing these two elements by considering both false positives and false negatives.

Experiment2.1 Data pre-processing techniques

For this experiment, F1 score of System 1 is 0.35, for System2 is 0.43 and for System 3 is 0.88 . We can say that system3 has outperformed systems 1 and 2. This result is mainly because of use of bigrams while vectorizing the Tokens. All 3 system used Tfidf vectorization but the use bigram significantly increase the F1 score for System3.

Experiment2.2 NLP algorithms/techniques

For this experiment, F1 score of System 1 is 0.23 and System 2 is 0.23. Both the system underperformed in this aspect. For System 2 increasing the epoch might result in a better F1 score.

Experiment 2.3 - text encoding/transformation

For this experiment, F1 score of System 1 is 0.27 and System 2 is 0.30. Yet these are low F1 scores for any NLP model. Despite having good Accuracy lower precision score significantly reduces the overall F1 score.

Experiment2.6_Hyper-paarmeter Optimization

For this experiment F1 score of System 1 is 0.87 and System 2 is 0.87. Using Hyperparameters improves the F1-score. while performing Hyperparameter optimization we come across better results. These system has low accuracy than above model but better precision score helped the model to achieve a better F1 score. Getting best no of neighbours for KNN algorithm which 51 gave us better result .

3.3 Error Analysis

While using Tfidf vectorization few errors occurred which led to bad results, modify the dataset gave us better results. Errors also encountered like Out of vocab for the LSTM model. The issue was with the length of matrix, once we resolved the issue the LSTM model worked fine.

Also, the Dataset has a very high no of B-O tags sometimes it produces biased results . These model were able to predict the B-O tags but weren't able to handle othe tags such as B-AC , B-LF and I-LF.

4. Best Results from Testing

We got the best result after Hyper-parameter optimization as it shown as which is the right kernel to use and how many no of neighbours are required for a good F1 score. With both model SVM and KNN, we were able to get an F1 score 0f 0.87 which is much better than all the models used in the experiments.

For Experiemnt1 using bigrams helped in improving the model F1 score to 0.88.

5. Overall Attempt and Outcome

- All the models were able to predict the results to some extent, some produced biased result and some produced good results. For the first 3 experiments, we had low F1 score but a good accuracy, for the last experiment with Hyper-parameter Optimisation we were able to get good Accuracy, precision, and F1 Score.

-Based on the experiments a good F1 score will be higher than 0.85 which few of the model were able to achieve.

-For models that has underperformed such as KNN finding the right no of neighbours is of utmost importance taking that no randomly reduces the prediction of these models. For a few models that used Tfidf vectorization a better or more complex vectorization technique should be used. For example, in Exp1 system 3 we used bigrams with Tfidf which improved the results significantly.

- For models such as Exp1 System3 using bigrams, using trigrams or more n-grams could produce a much better results.

6. References

https://www.researchgate.net/figure/Sequence-Labeling-model-for-LSTM-network_fig1_317085608

https://monkeylearn.com/blog/beginners-guide-text-vectorization/

https://arxiv.org/abs/2209.09430