COM3029 - Natural Language Processing

Individual Coursework

Group 6

Oliver Krieger
URN:6664919

April 2024

Contents

1 Introduction

1.1 Group 6 Members

1. Eleanor Lurgio
2. Oliver Krieger
3. Seo(Seoeun) Lee
4. Thiri(Alice) Thu

1.2 Individual Tasks

Individual Tasks				
Tasks	Eleanor	Oliver	Seoeun	Alice
Data Pre-processing		✓	✓	
NLP Algorithms			✓	✓
Text Encoding / Transformation		✓	✓	
Dataset Splitting	✓			✓
Loss Functions and Optimisers	✓			✓
Hyper Parameters	✓	✓	✓	
Pre-trained models	✓	✓		✓

My tasks include:

1. Data Pre-processing
2. Text Encoding / Transformation
3. Hyper Parameters (Fine Tuning)
4. Pre-trained models (DistilBERT and SpaCy)

2 Data Analysis

2.1 Loading The Dataset

For this coursework, we are exploring the PLOD Dataset which is an English-language dataset of abbreviations and their long-forms tagged in text. The data is gathered from research for PLOS journals. The dataset is loaded using the hugging face datasets module. This is done within jupyter notebook environment.

```python
# Import dataset import function for hugging face
from datasets import load_dataset, DatasetDict, Dataset

# import the coursework dataset from
dataset_dict:DatasetDict = load_dataset("surrey-nlp/PLOD-CW")
```

Listing 1: Load Dataset

This creates a dataset dictionary with 3 datasets - train, validation and test. Each dictionary item has dataset definitions of ”tokens”, ”pos tags” and ”ner tags”. These are represented as lists of lists of strings (list[list[str]]), the sublists being a representation of rows.

```
DatasetDict({
    train: Dataset({
        features: ['tokens', 'pos_tags', 'ner_tags'],
        num_rows: 1072
    })
    validation: Dataset({
        features: ['tokens', 'pos_tags', 'ner_tags'],
        num_rows: 126
    })
    test: Dataset({
        features: ['tokens', 'pos_tags', 'ner_tags'],
        num_rows: 153
    })
})
```

Listing 2: Dataset Features

Each of these datasets can be accessed as normal python dictionaries by calling the dataset name followed by the features name. For instance:

```
dataset["train"]["tokens"]
```

Would return the list of tokens from the train dataset. As the arrays can become slow to iterate, it was sensible to load them into data collections where that contain data items. The collections allow for easy access to lists and reusable functionality.

2.2 Data Samples

After loading the dataset, we can test to see how many samples we have to train on, as well as what our split between labels are. To begin observing this, we will want to write some helper functions to make it easier to read the data. First we will write a ”flatten list” function that will help us take a list of lists and make it into a singular list:

```
def flatten_list(given_list:list[list[any]]) -> list[any]:
    return [element for inner_list in given_list for element in
    inner_list]
```

Listing 3: Flatten List Function

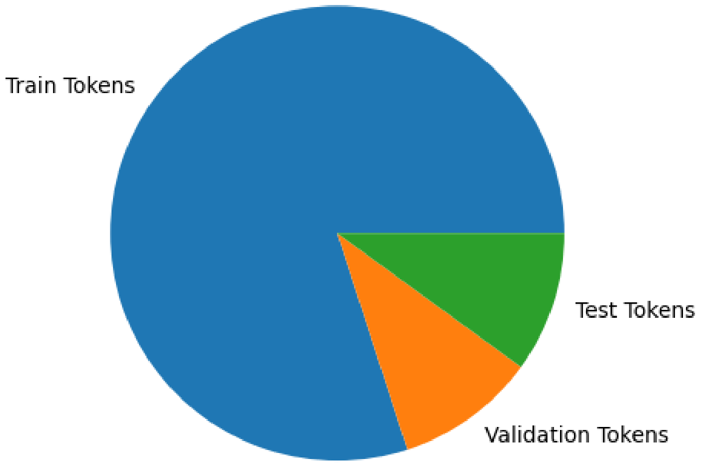
This way we can combine all our rows into one single list and perform calculations on it. Since every token has to have a pos and ner tag to it, we can assume that all lists are of the same size at the end. To more easily access our data and to speed up our queries, we will create our own DataRow and DataCollection items, as to more easily manipulate the data.

```
class DataRow:
    def __init__(self, tokens, pos, ner, row_idx=0):
```

```python
        self.idx:int = row_idx
        self.tokens:list[str] = tokens
        self.pos:list[str] = pos
        self.ner:list = ner

class DataCollection:
    def __init__(self, list_collection:list[DataRow], max_token_length=512):
        self.max_token_length:int = max_token_length # max token length (if we tokenize inputs)
        self.collection:list[DataRow] = list_collection # list of rows in the collection

    # get a list of token rows
    def get_token_list(self) -> list[list[str]]:
        return [data_item.tokens for data_item in self.collection]

    # get a list of pos rows
    def get_pos_list(self) -> list[list[str]]:
        return [data_item.pos for data_item in self.collection]

    # get a list of ner rows
    def get_ner_list(self) -> list[list[str]]:
        return [data_item.ner for data_item in self.collection]
```

Listing 4: Data Row and Data Collection

By getting each of the lists, we discover that the train dataset has 40,000 tokens, and validation and test sets both have 5000 tokens each. This leads us to 50000 total samples, from which 40000 we use to train on and another 5000 to test while training.



As we will be training on the train dataset, we will want to know how many of the 40000 tokens are actually unique. To do this, we will build a function to make a dictionary of unique tokens, as well as to get the frequency of each token.

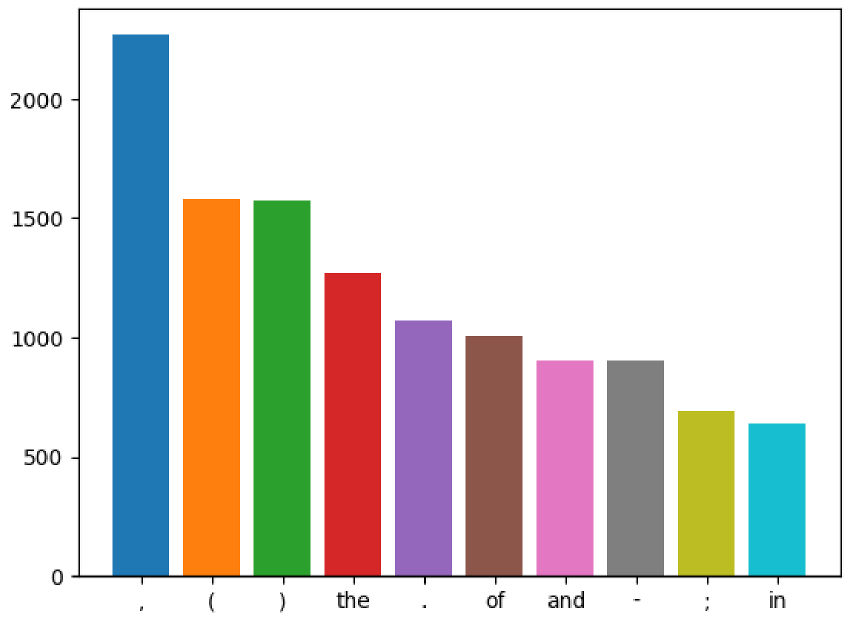
```python
def get_word_frequency(data_list:list[str]) -> dict:
    word_frequency:dict = {"total_tokens":0, "unique_tokens":0, "unique_token_frequency":{}}
    for value in data_list:
        if value not in word_frequency["unique_token_frequency"].keys():
            word_frequency["unique_token_frequency"][value] = 1
            word_frequency["unique_tokens"] += 1
        else:
            word_frequency["unique_token_frequency"][value] += 1
        word_frequency["total_tokens"] += 1
    return word_frequency
```

Listing 5: Getting Token Frequency

By looking at the unique tokens in the train set, we find that out of the 40000 samples, we actually only have 9133 unique tokens. This number is actually smaller as we will discuss in the pre-processing step, but for now these will be our unique tokens. From the frequency we can also get which are the most common tokens.

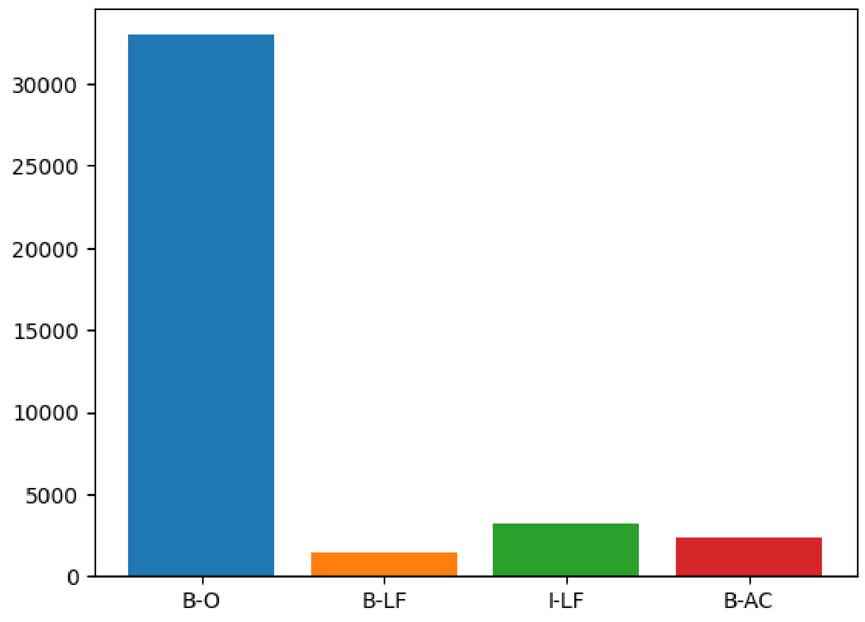
```python
train_frequency:dict = get_word_frequency(flatten_list(train_collection.get_token_list()))
sorted_list = sorted(train_frequency["unique_token_frequency"].items(), key=lambda t: t[1], reverse=True)
sorted_list = sorted_list[:10]
sorted_dict = {item_tuple[0]:item_tuple[1] for item_tuple in sorted_list}
plt.bar(sorted_dict.keys(), sorted_dict.values(), color=colors)
plt.show() # frequency of classes / labels
```

Listing 6: Getting Token Frequency



From plotting the top 10 most frequent tokens, we can clearly see that the most common tokens are stopwords and punctuations. In order to both balance our classes and focus learning only on abbreviations and long forms, we might benefit from removing them, though we also want to learn what are words that should be labelled as neither, so we will experiment with both.

Next we will want to test how many labels we have available. To do this, we can get use the word frequency function we have already written. This will show us a distribution on the training set of 'B-O': 32971, 'B-LF': 1462, 'I-LF': 3231, 'B-AC': 2336. In a plot it would look as following:



We can see that we have 4 classes, B-O standing for not an abbreviation or long form, B-AC being an abbreviation, B-LF is long form beginning and I-LF is the continuation of the long form. All long forms will be represented as starting with B-LF followed by I-LF if any. This creates a sequence.

From the plot, we can see that the dataset is imbalanced towards the B-O class. Data imbalance is usually resolved by either oversampling or undersampling the dataset. In the case of oversampling, we would add more values, either from the test or validation set (while removing them from their corresponding set) and adding them to the training set to increase the values of the B-LF, I-LF and B-AC classes. For undersampling we would remove values of B-O. This would be most easily done by removing stop words and their corresponding POS and NER tag values.

3 Experiments

3.1 Experiment 1 - Data-Pre processing

As the first experiment, we will explore pre-processing the data, by using different techniques. To test, we will use the pre-trained DistilBert model from hugging face transformers. Techniques to be tested will be as follows:

1. Clean Stopwords and Punctuations
2. Stemming
3. Lemmatization
4. Removing Special Characters

3.1.1 Letter Casing

A common first step in any NLP training is to pre-process the text as the original collection might contain noise. As we have seen from the data analysis, this is true in terms of containing stop words and punctuations.

Let us look at the first 5 words in the first 3 rows:

First 5 words Per Row
['For', 'this', 'purpose', 'the', 'Gothenburg']
['The', 'following', 'physiological', 'traits', 'were']
['Minor', 'H', 'antigen', 'alloimmune', 'responses']

As our first step, we should make sure that the data is uniform throughout. When we analysed our training tokens, we found that we had 9133 unique tokens. However, this also includes same words with different cased letters. Firstly, let us write a function to lowercase all the tokens:

```python
def data_to_lower(data:list[list[str]]) -> list[list[str]]:
    return [[token.lower() for token in tokens] for tokens in data]
```

Listing 7: Lowercase strings function

Using the "get word frequency" function we created before, we can now test our original tokens and lower case tokens for unique token count. What we will find is that after lowercasing all of the training dataset tokens, we are left with 8339 unique tokens.

3.1.2 Removing punctuations and stopwords

With our texts being uniform, the next step we can do is to remove all of the punctuations from the text. As they are not words, they increase the B-O count, as well as are one of the most common tokens within the dataset and therefore lean the model to potentially learn more about punctuations than about abbreviations and long forms.

```python
def remove_values(data_collection:DataCollection, remove_values:
    list|str) -> DataCollection:
    collection:list[DataRow] = []
    for data_row in data_collection.collection:
        new_row:DataRow = DataRow([], [], [])
        for item_idx, token in enumerate(data_row.tokens):
            if token not in remove_values:
                new_row.tokens.append(data_row.tokens[item_idx])
                new_row.pos.append(data_row.pos[item_idx])
                new_row.ner.append(data_row.ner[item_idx])
        if len(new_row.tokens) > 0:
            collection.append(new_row)
    new_collection = DataCollection(collection)
    new_collection.set_unique_ner_tags(tag_list)
    return new_collection
```

Listing 8: Code to remove any characters within given list

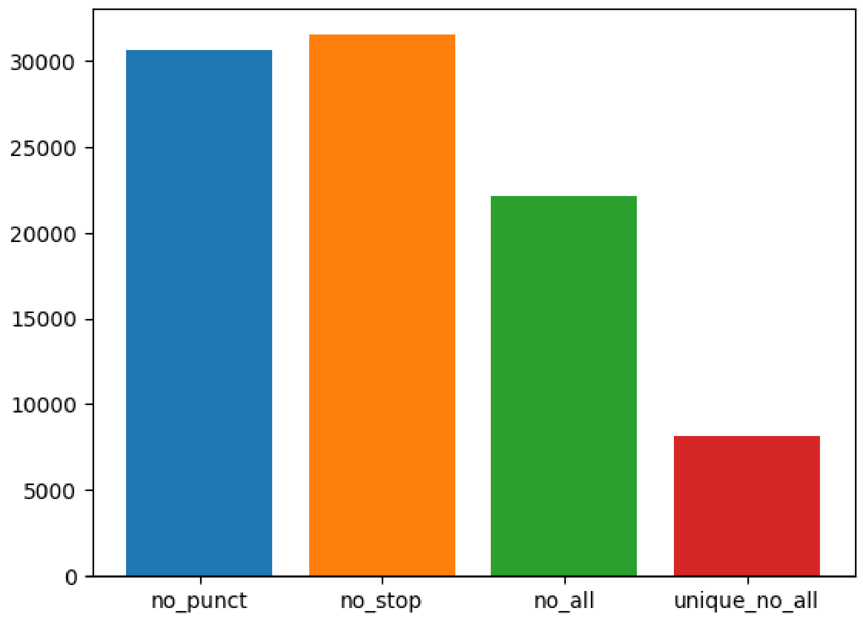
3.1.3 Removing unique words

With the above code, we can also create a collection where we remove stopwords and a collection where we remove both stopwords and punctuations. If we remove the stopwords we are left with 31497 tokens. Removing both the stopwords and punctuations leaves us with 22079 words. But what if we were to also only keep unique tokens? We would lose sequential data (as this would not always guarantee B-LF to be followed by I-LF) and we would expect to have worse results in recognition, but to test this out and prove this theory, let us only keep unique words and then also remove stopwords and punctuations.

```python
def unique_collection(data_collection:DataCollection) ->
    DataCollection:
    collection:list[DataRow] = []
    unique_tokens:list[str] = []
    for data_row in data_collection.collection:
        new_row:DataRow = DataRow([], [], [])
        for item_idx, token in enumerate(data_row.tokens):
            if token not in unique_tokens:
                new_row.tokens.append(data_row.tokens[item_idx])
                new_row.pos.append(data_row.pos[item_idx])
                new_row.ner.append(data_row.ner[item_idx])
                unique_tokens.append(token)
        if len(new_row.tokens) > 0:
            collection.append(new_row)
    new_collection = DataCollection(collection)
    new_collection.set_unique_ner_tags(tag_list)
    return new_collection
```

Listing 9: Code to make tokens unique only

When running the unique token collection with stop words and punctuations removed, we are left with 8198 tokens. When we plot all of these collections together, we can view their sizes:



3.1.4 Stemming and Lemmatisation

Next step is to create 2 more collections for our experiments - stemmed and lemmatisation collections.

Stemming is a text pre-processing technique used to reduce words to their root or base form. The goal of stemming is to simplify and standardize words, which helps improve the performance of text classification. In lemmatisation we also try to reduce a given word to its root word.

The difference between the two is that in stemming we remove last few characters from a word, often leading to incorrect meanings and spelling. However in lemmatisation, we consider the context and convert the word to its meaningful base form, which is called Lemma.

The issue with stemming can be that the word loses its original meaning completely. However, with lemmatising, we might not scale the word back enough to correlate 2 words to be the same. Both have their pros and cons.

In order to do this, we will import the appropriate NLTK libraries

```python
from nltk.stem import PorterStemmer
ps = PorterStemmer()

from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
```

Listing 10: NLTK libraries for stemming and lemmatising

To stem or lemmatise tokens, we can do the following:

```python
stemmed_tokens = [[ps.stem(token) for token in token_list]for
    token_list in train_collection.get_token_list()]
```

```
lemmatised_tokens = [[lemmatizer.lemmatize(token) for token in
    token_list]for token_list in train_collection.get_token_list()]
```

Listing 11: Tokens to stemmed and lemma

The NLTK lemmatiser can also take an additional pos tag value: ”n” for nouns, ”v” for verbs, ”a” for adjectives, ”r” for adverbs and ”s” for satellite adjectives.

With this, we can build a slightly more sophisticated lemmatiser and see if it has a more different effect.

```
def collection_to_lemma(data_collection:DataCollection) ->
    DataCollection:
    collection:list[DataRow] = []
    for data_row in data_collection.collection:
        new_row:DataRow = DataRow([], [], [])
        for item_idx, pos in enumerate(data_row.pos):
            token = data_row.tokens[item_idx]
            new_token = ""
            if pos == "NOUN":
                new_token = lemmatizer.lemmatize(token, "n")
            elif pos == "VERB":
                new_token = lemmatizer.lemmatize(token, "v")
            elif pos == "ADJ":
                new_token = lemmatizer.lemmatize(token, "a")
            elif pos == "ADV":
                new_token = lemmatizer.lemmatize(token, "r")
            else:
                new_token = lemmatizer.lemmatize(token)
            new_row.tokens.append(new_token)
            new_row.pos.append(data_row.pos[item_idx])
            new_row.ner.append(data_row.ner[item_idx])
        if len(new_row.tokens) > 0:
            collection.append(new_row)
    new_collection = DataCollection(collection)
    new_collection.set_unique_ner_tags(tag_list)
    return new_collection
```

Listing 12: Lemma with POS handling

3.1.5 Removing special characters

One last thing we can do, is to remove special characters. Sometimes learning these characters can result in the model accuracy dropping and thus we can see if there are any within our dataset and remove them.

```
def is_special_character(token):
    if len(token) > 1:
        return False
    else:
        return ord(token) < 32 or ord(token) > 127 # true if not
    special character


special_chars = []
for token in flatten_list(train_collection.get_token_list()):
```

```python
    if is_special_character(token):
        special_chars.append(token)

train_no_special_collection = remove_values(train_collection, list(
    set(special_chars)))
```

Listing 13: Special character removal

3.1.6 Evaluation

Now that we have set up our data for testing, we require a model to test the accuracy on. As such, we will build the Hugging Face Token Classification model for DistilBERT and test our data processed data on that. Since our data is already in collections, it will be easy to add it as we go.

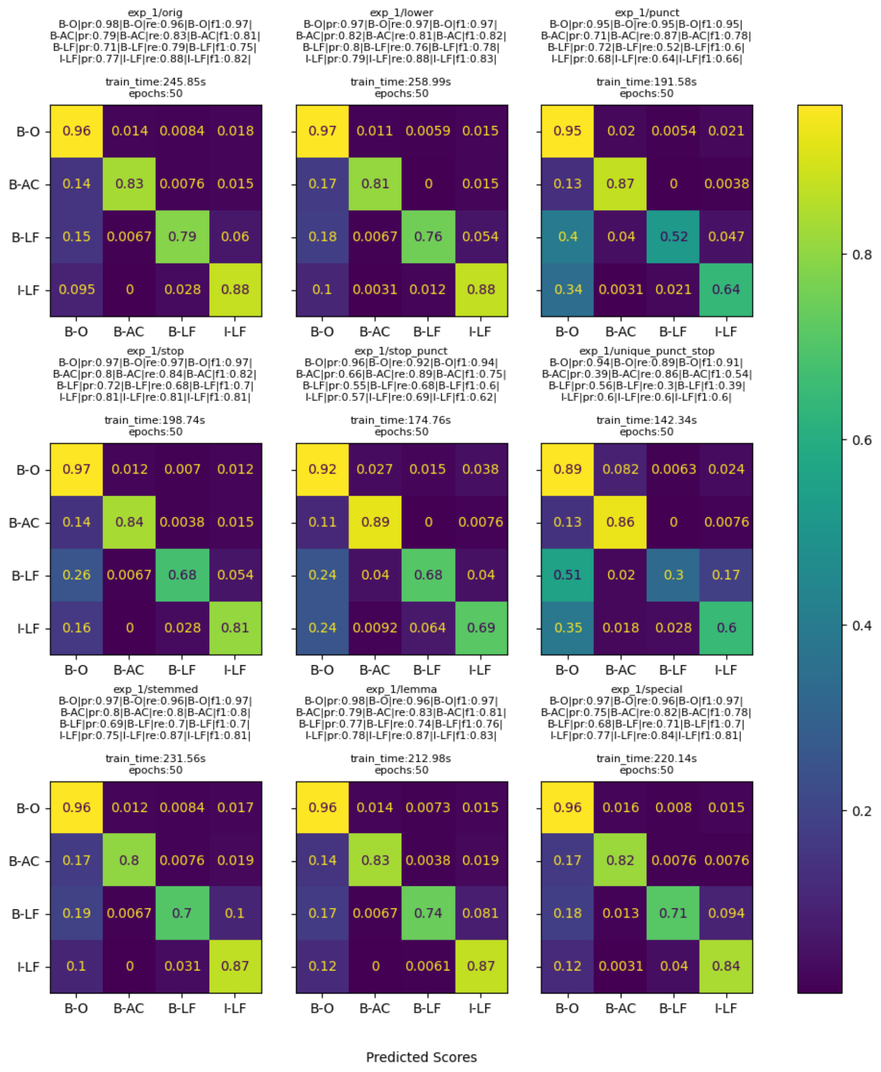
For the model, we will have to build an auto tokeniser, which will both encode and pad our tokens and align labels. It will also give us access to hyper parameterisation. However, we will explain those in more detail in the coming experiments. For now we will focus on only the evaluation of data processing.

The model is trained with the following arguments:

Training Arguments	
Epochs	50
Learning Rate	2e-5
Batch Size	16
Weight Decay	0.01
Evaluation Strategy	Epoch
Save Strategy	Epoch

As we can see from our results below, training over 50 epochs that our best results come from the orginial text and lemmatisation. We can also see that worst results are from only using unique words while removing stop words and punctuations. This makes sense, as we are removing a lot of data as well as breaking sequential order of B-LF and I-LF, resulting in worse capability of predicting, which lines up with our predictions before.

What is more interesting is that removing stopwords actually does not affect the f1 score too much, but removing punctuations does. This is potentially down to the fact that we have more punctuations than stopwords (as seen in the data analysis section), meaning that we lose more data when removing the punctuations and therefore causing a loss in score. On the other hand, as we do not have many special characters in our training set, removing them barely changes the scoring.


exp_1/orig
B-O|pr:0.98|B-O|re:0.96|B-O|f1:0.97|
B-AC|pr:0.79|B-AC|re:0.83|B-AC|f1:0.81|
B-LF|pr:0.71|B-LF|re:0.79|B-LF|f1:0.75|
I-LF|pr:0.77|I-LF|re:0.88|I-LF|f1:0.82|
train_time:245.85s
epochs:50
B-O 0.96 0.014 0.0084 0.018
B-AC 0.14 0.83 0.0076 0.015
B-LF 0.15 0.0067 0.79 0.06
I-LF 0.095 0 0.028 0.88
B-O B-AC B-LF I-LF
exp_1/lower
B-O|pr:0.97|B-O|re:0.97|B-O|f1:0.97|
B-AC|pr:0.82|B-AC|re:0.81|B-AC|f1:0.82|
B-LF|pr:0.8|B-LF|re:0.76|B-LF|f1:0.78|
I-LF|pr:0.79|I-LF|re:0.88|I-LF|f1:0.83|
train_time:258.99s
epochs:50
0.97 0.011 0.0059 0.015
0.17 0.81 0 0.015
0.18 0.0067 0.76 0.054
0.1 0.0031 0.012 0.88
B-O B-AC B-LF I-LF
exp_1/punct
B-O|pr:0.95|B-O|re:0.95|B-O|f1:0.95|
B-AC|pr:0.71|B-AC|re:0.87|B-AC|f1:0.78|
B-LF|pr:0.72|B-LF|re:0.52|B-LF|f1:0.6|
I-LF|pr:0.68|I-LF|re:0.64|I-LF|f1:0.66|
train_time:191.58s
epochs:50
0.95 0.02 0.0054 0.021
0.13 0.87 0 0.0038
0.4 0.04 0.52 0.047
0.34 0.0031 0.021 0.64
B-O B-AC B-LF I-LF
exp_1/stop
B-O|pr:0.97|B-O|re:0.97|B-O|f1:0.97|
B-AC|pr:0.8|B-AC|re:0.84|B-AC|f1:0.82|
B-LF|pr:0.72|B-LF|re:0.68|B-LF|f1:0.7|
I-LF|pr:0.81|I-LF|re:0.81|I-LF|f1:0.81|
train_time:198.74s
epochs:50
B-O 0.97 0.012 0.007 0.012
B-AC 0.14 0.84 0.0038 0.015
B-LF 0.26 0.0067 0.68 0.054
I-LF 0.16 0 0.028 0.81
B-O B-AC B-LF I-LF
exp_1/stop_punct
B-O|pr:0.96|B-O|re:0.92|B-O|f1:0.94|
B-AC|pr:0.66|B-AC|re:0.89|B-AC|f1:0.75|
B-LF|pr:0.55|B-LF|re:0.68|B-LF|f1:0.6|
I-LF|pr:0.57|I-LF|re:0.69|I-LF|f1:0.62|
train_time:174.76s
epochs:50
0.92 0.027 0.015 0.038
0.11 0.89 0 0.0076
0.24 0.04 0.68 0.04
0.24 0.0092 0.064 0.69
B-O B-AC B-LF I-LF
exp_1/unique_punct_stop
B-O|pr:0.94|B-O|re:0.89|B-O|f1:0.91|
B-AC|pr:0.39|B-AC|re:0.86|B-AC|f1:0.54|
B-LF|pr:0.56|B-LF|re:0.3|B-LF|f1:0.39|
I-LF|pr:0.6|I-LF|re:0.6|I-LF|f1:0.6|
train_time:142.34s
epochs:50
0.89 0.082 0.0063 0.024
0.13 0.86 0 0.0076
0.51 0.02 0.3 0.17
0.35 0.018 0.028 0.6
B-O B-AC B-LF I-LF
exp_1/stemmed
B-O|pr:0.97|B-O|re:0.96|B-O|f1:0.97|
B-AC|pr:0.8|B-AC|re:0.8|B-AC|f1:0.8|
B-LF|pr:0.69|B-LF|re:0.7|B-LF|f1:0.7|
I-LF|pr:0.75|I-LF|re:0.87|I-LF|f1:0.81|
train_time:231.56s
epochs:50
B-O 0.96 0.012 0.0084 0.017
B-AC 0.17 0.8 0.0076 0.019
B-LF 0.19 0.0067 0.7 0.1
I-LF 0.1 0 0.031 0.87
B-O B-AC B-LF I-LF
exp_1/lemma
B-O|pr:0.98|B-O|re:0.96|B-O|f1:0.97|
B-AC|pr:0.79|B-AC|re:0.83|B-AC|f1:0.81|
B-LF|pr:0.77|B-LF|re:0.74|B-LF|f1:0.76|
I-LF|pr:0.78|I-LF|re:0.87|I-LF|f1:0.83|
train_time:212.98s
epochs:50
0.96 0.014 0.0073 0.015
0.14 0.83 0.0038 0.019
0.17 0.0067 0.74 0.081
0.12 0 0.0061 0.87
B-O B-AC B-LF I-LF
exp_1/special
B-O|pr:0.97|B-O|re:0.96|B-O|f1:0.97|
B-AC|pr:0.75|B-AC|re:0.82|B-AC|f1:0.78|
B-LF|pr:0.68|B-LF|re:0.71|B-LF|f1:0.7|
I-LF|pr:0.77|I-LF|re:0.84|I-LF|f1:0.81|
train_time:220.14s
epochs:50
0.96 0.016 0.008 0.015
0.17 0.82 0.0076 0.0076
0.18 0.013 0.71 0.094
0.12 0.0031 0.04 0.84
B-O B-AC B-LF I-LF
0.8
0.6
0.4
0.2
Predicted Scores

Experiment 1 Results		
Test	F1 Score	Time
Original	O:97%, AC:81%, BLF:75%, ILF:82%	245.85s
Lowered	O:97%, AC:82%, BLF:78%, ILF:83%	258.99s
Removed Punct	O:95%, AC:78%, BLF:60%, ILF:66%	191.58s
Removed Stop	O:97%, AC:82%, BLF:70%, ILF:81%	198.74s
Removed Punct/Stop	O:94%, AC:75%, BLF:60%, ILF:62%	174.76s
Unique Words, Removed Punct/Stop	O:91%, AC:54%, BLF:39%, ILF:60%	142.34s
Stemmed	O:97%, AC:80%, BLF:70%, ILF:81%	231.56s
Lemmatised	O:97%, AC:81%, BLF:76%, ILF:83%	212.98s
Removed Special	O:97%, AC:78%, BLF:70%, ILF:81%	220.14s

3.2 Experiment 2 - Text Encoding / Transformation

For the second experiment we will test out text vectorisation and see how that will affect the outcome of training. For testing, we will build our own model using a CNN model.

Tokenisers are used to convert raw text into numerical tokens which makes it possible for machine learning models to be understood. When using hugging face models, the AutoTokenizer class makes it easy to select and load appropriate tokenisers for pre-trained models, where they encode words and vectorise them. This simplifies the process of selecting the correct tokeniser associated with model, streamlining the workflow.

However, to test different vectorisation techniques and see how it works, we will be vectorising the tokens ourselves, using different methods to see how it improves (or unimproves) the training process. The methods we will use for vectorisation are as following:

1. Word2vec
2. Fasttext

In name entity recognition, using word vectors can give extra context to the models to improve the identification of entities (such as names and locations). This is done by vectorising words and setting similar words near each other.

3.2.1 Word2Vec

The way that the word2vec works is by generating vectors for words that are supposed to be close to each other. For instance "man" and "boy" would be close to each other, while "woman" would be further from them, but all would be close to "person". To get started with word2vec, we must import it from the gensim models.

```python
from gensim.models import Word2Vec
```

Listing 14: Word2Vec gensim import

Methods often used in Word2Vec is CBOW or Skip Gram [3]. Both algorithms use nearby words in order to extract the semantics of the words. In Skip-Gram, we try to predict the context words using the main word. In Continuous Bag of Words (CBOW), we want to use context words to predict the main words. Essentially we are doing reverse of each other.

Before we can use vectorisation in Word2Vec, we need to create a collection of all tokens that the Word2Vec will use to create a "library" of words it can create vectors for. It also uses it to position tokens that should be close to each other, as it has train method to position these tokens as well.

To do this, let us write some code to collect our tokens into one:

```python
def create_token_library(collections:list[DataCollection],
    collect_all:bool=True) -> list[list[str]]:
    unique_tokens:list[str] = []
    token_lists:list[list[str]] = []
    for collection in collections:
        tokens:list[list[str]] = collection.get_token_list()
        for token_row in tokens:
            token_rows:list[str] = []
            for token in token_row:
                if token not in unique_tokens or collect_all:
                    token_rows.append(token)
                    unique_tokens.append(token)
            if len(token_rows) > 0:
                token_lists.append(token_rows)
    return token_lists
```
Listing 15: Creating token library

This creates us a list of 50000 tokens, which is the entire set of all of our tokens between our 3 datasets - train, test and validation. With our tokens added to lists, we put these words through the word2vec and it auto indexes it for us, creating embeddings. We can view these the word2vector model to get those embedding numbers.

```python
w2v_model_CBOW:Word2Vec = Word2Vec(token_lib, min_count=1,
    vector_size=2, window=5)
```
Listing 16: Word2Vec model generated

From this, there are few values we should understand when generating the embeddings and vectors:

1. vector_size (int, optional) – Dimensionality of the word vectors.
2. window (int, optional) – Maximum distance between the current and predicted word within a sentence.
3. min_count (int, optional) – Ignores all words with total frequency lower than this.
4. sg (0, 1, optional) – 1 for skip-gram; otherwise CBOW.

To get a word embedding for the word "picture" we can run the following code:

```python
w2v_model_CBOW.wv.key_to_index["picture"]
```
Listing 17: Get Word2Vec embedding

Looking at the first 5 embeddings, we can see how it generates as a dictionary:

First 5 embeddings
',': 0, '(': 1, ')': 2, 'the': 3, '.': 4, 'of': 5

However, in order to get the vector representations of words, we have to pass it to the get vector function. This will return us a vector of a size that we give it when we construct our word2vec model. For instance, a representation of a vector size 2 for the word "picture" could be:

```
array([0.0559599 , 0.07942823], dtype=float32)
```

Listing 18: Vector of size 2 representation of word "picture"

The vector representation also allows us to call similarity between words, as well as get top n words similar to that word.

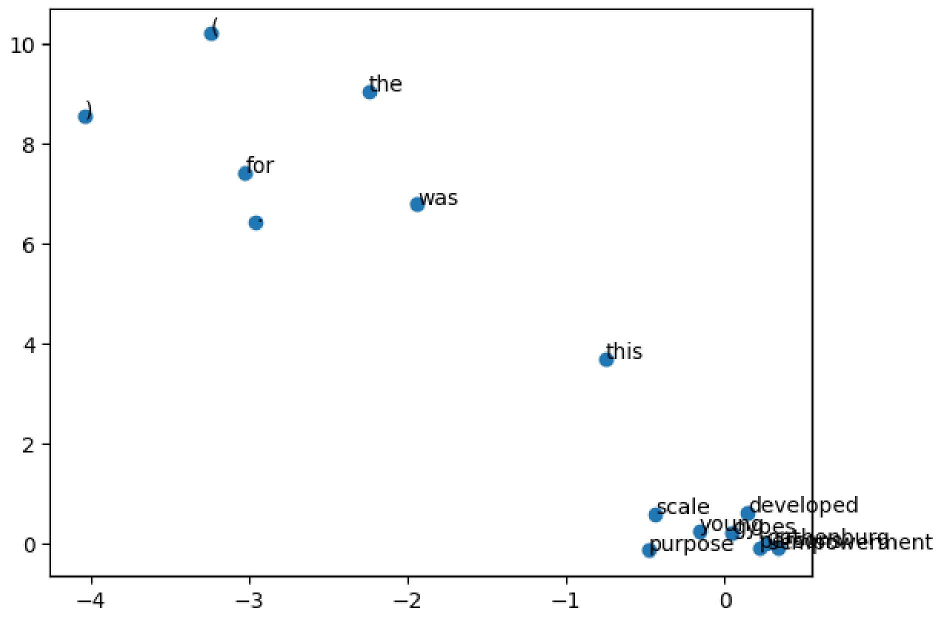
```
w2v_model_CBOW.wv.similarity('walk', 'bed')
w2v_model_CBOW.wv.most_similar('bed', topn=5)
w2v_model_CBOW.wv.most_similar('hour', topn=5)
```

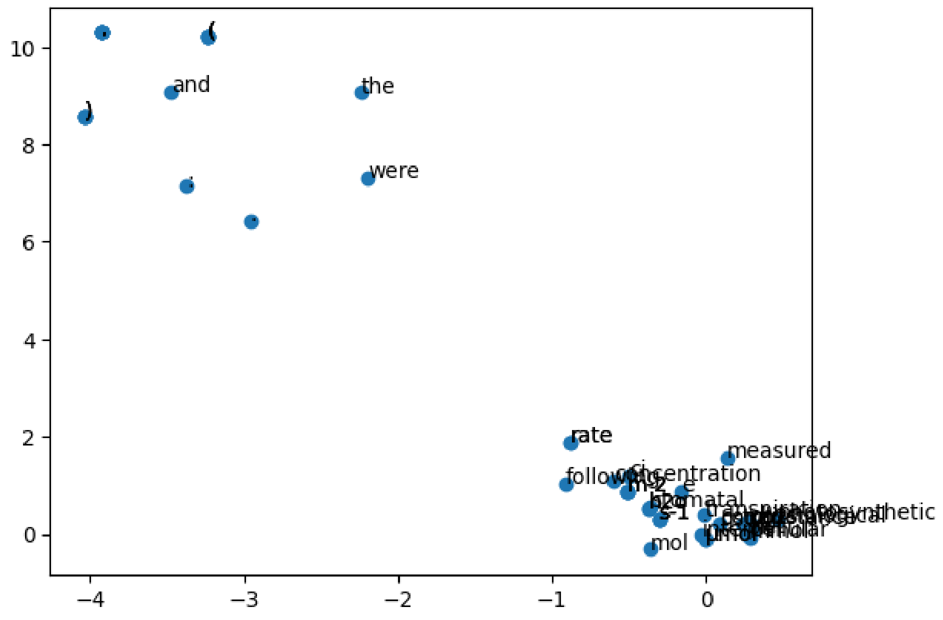
Listing 19: Similarity calculations

Now, usually CNNs are trained on images by moving a kernel around an image. However, in our case, we do not have an image, but instead text. However, we know that each of the tokens that we have in a row essentially forms a "sentence". if we vectorise the words, especially with 2, we can imagine a graph, where the words are plotted around and thus generate images that we can go across with a kernel. To visualise, we will plot the first two sentances from the train collection:

```
v1:list[tuple[any, str]] = [(w2v_model_CBOW.wv.get_vector(token),
    token) for token in train_collection.get_token_list()[0]]
v2:list[tuple[any, str]] = [(w2v_model_CBOW.wv.get_vector(token),
    token) for token in train_collection.get_token_list()[1]]

def plot_vec(vector:list[tuple[any, str]]):
    x = []
    y = []


    for vec, _ in vector:
        x.append(vec[0])
        y.append(vec[1])

    fig, ax = plt.subplots()
    ax.scatter(x, y)

    for idx, (_, token) in enumerate(vector):
        ax.annotate(token, (x[idx], y[idx]))

plot_vec(v1)
plot_vec(v2)
```

Listing 20: Similarity calculations





We can now actually visually see that stopwords and punctuations actually draw quite far away from the rest of our words.

However, in order to do this, we would need the images to all be the same size. This is why we require padding. We calculate the max sentence length and then pad all the sentences with lesser size to be the same.

```python
def find_max_token_row(tokens:list[list[any]]) -> int:
    max_token_row_size = 0
    for token_row in tokens:
        if len(token_row) > max_token_row_size:
            max_token_row_size = len(token_row)
    return max_token_row_size

def pad_vectors(vector_row:list, max_row_value:int, pad_token:any) -> None:
    padding_size = max_row_value - len(vector_row) # how many numbers do we need to pad with

    if padding_size == 0:
        return # vector does not required padding, so return early
    elif padding_size < 0:
        print("ERROR - max row value is somehow smaller than the given vector row length! This is not allowed!")
        return

    for _ in range(padding_size):
        vector_row.append(pad_token)

def vectorise_tokens(data_collection:DataCollection, vectorisation_model:FastText|Word2Vec, max_token_row_size:int, pad_token:str, pad_label_token:int) -> list[dict]:
    tokens = data_collection.get_token_list()
    ner_tags = data_collection.ner_as_idx

    vectorised_tokens = []
    vectorised_ner = []

    for token_row, ner_row in zip(tokens, ner_tags):
        ner_rows = []
        token_rows = []
        for token, ner in zip(token_row, ner_row):
            if vectorisation_model.wv.has_index_for(token):
                token_rows.append(vectorisation_model.wv.get_vector(token))
                ner_rows.append(ner)
        if len(token_rows) > 0:
            vectorised_tokens.append(token_rows)
            vectorised_ner.append(ner_rows)

    for vec_token_row, vec_ner_row in zip (vectorised_tokens, vectorised_ner):
        pad_vectors(vec_token_row, max_token_row_size, vectorisation_model.wv.get_vector(pad_token))
        pad_vectors(vec_ner_row, max_token_row_size, pad_label_token)

    return vectorised_tokens, vectorised_ner
```

Listing 21: Vectorise and pad sentences

While in concept, this might seem a great way to solve this issue, we might note that in our train datasets case, the longest sentence is 323, but the average sentence length is only 38. This means that we will have a majority of sentences that have been padded by an extreme amount, making the sentence consist mostly of padding, rather than of the sentence.

```python
def get_avg_sent_size(collection:DataCollection):
    sent_token_sum = 0
    for token_row in collection.get_token_list():
        sent_token_sum += len(token_row)
    return (sent_token_sum/len(collection.get_token_list()))

def split_list(row:list, max_sent_size:int, split_into:int) -> list
    [list[str]]:
    new_list = []
    for idx in range(split_into):
        if idx != split_into:
            new_list.append(row[idx:idx+max_sent_size])
        else:
            new_list.append(row[idx:]) # append the remaining
    values!
    return new_list

def split_sentances_to_max_size(max_sent_size:int, collection:
    DataCollection, tag_list:dict) -> DataCollection:
    data_row_list:list[DataRow] = []
    for idx, data_row in enumerate(collection.collection):
        if len(data_row.tokens) > max_sent_size:
            split_into = math.ceil(len(data_row.tokens)/
    max_sent_size)
            token_lists = split_list(data_row.tokens, max_sent_size
    , split_into)
            pos_lists = split_list(data_row.pos, max_sent_size,
    split_into)
            ner_lists = split_list(data_row.ner, max_sent_size,
    split_into)
            for row_idx in range(split_into):
                data_row_list.append(DataRow(token_lists[row_idx],
    pos_lists[row_idx], ner_lists[row_idx]))
        else:
            data_row_list.append(data_row)
    new_collection = DataCollection(data_row_list)
    new_collection.set_unique_ner_tags(tag_list)
    return new_collection
```

Listing 22: Split sentences to max size

To avoid this issue, let's split the sentences up to the average sentence length - this will introduce issues in our semantics of B-LF and I-LF as they will potentially be broken across two sentences. However, this is still more optimal than having a majority padding solution.

3.2.2 FastText

Similarly to Word2Vec, FastText can be used to vectorise words. In fact both of them can be imported from gensim and interacted with almost identically. However in FastText, each word is represented as the average of the vector representation of its character n-grams along with the word itself [7].

Consider the word ”wilds” and n-gram = 3, the representative n-grams would be:
wi, wil, ild, lds, ds and wilds

Because of this, the vector representation would be different per token and thus yield a different training result.

3.2.3 CNN Model

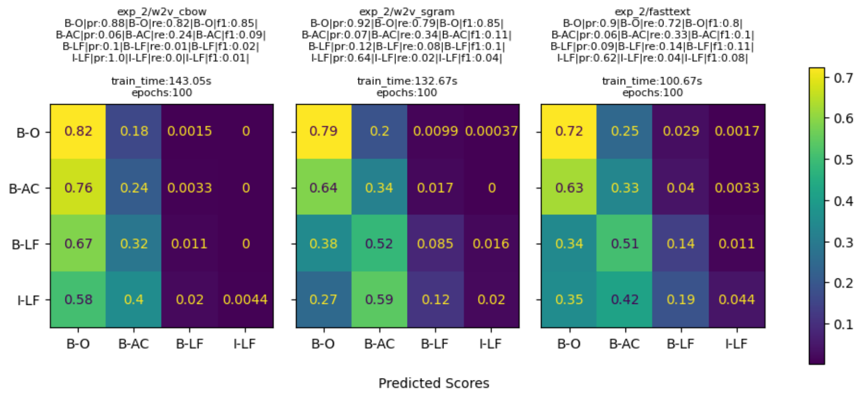
The CNN model itself is fairly simplistic - it uses 2 convolution layers, linear layers and some pooling [1]. The loss function is MSELoss and Optimiser Adam. Vector sizes for words has been set to 25.

```python
import torch
import torch.nn.functional as F

class CNNModel(torch.nn.Module):

    def __init__(self):
        super(CNNModel, self).__init__()
        self.conv1 = torch.nn.Conv2d(1, 6, 1)
        self.conv2 = torch.nn.Conv2d(6, 16, 1)

        self.fc1 = torch.nn.Linear(864, 648)
        self.fc2 = torch.nn.Linear(648, 432)
        self.fc3 = torch.nn.Linear(432, 216)
        self.fc4 = torch.nn.Linear(216, 38) # 38 is the max
    sentance size

    def forward(self, x: torch.Tensor):
        x = x.unsqueeze(-1)
        x = torch.permute(x, (0, 3, 2, 1))
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = self.fc4(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]
        num_features = 1
        for s in size:
            num_features *= s
        return num_features

class WindowDataset(torch.utils.data.Dataset):
    def __init__(self, data_inputs:list[list[list[float]]],
    data_labels:list[list[list[float]]]):
        self.inputs = data_inputs
        self.labels = data_labels
        self.length = len(data_inputs)

    def __getitem__(self, idx):
        inputs = torch.from_numpy(np.array(self.inputs[idx]))
        labels = torch.from_numpy(np.array(self.labels[idx]))
        return inputs, labels

    def __len__(self):
        return self.length
```

Listing 23: CNN and Dataloader

The training argument for the model are as follows:

Training Arguments	
Epochs	100
Learning Rate	5e-5
Loss Function	MSELoss
Optimiser	Adam



Experiment 1 Results		
Test	F1 Score	Time
Word2Vec CBOW	O:85%, AC:9%, BLF:2%, ILF:1%	143.05s
Word2Vec Skip Gram	O:85%, AC:11%, BLF:1%, ILF:4%	132.67s
FastText	O:80%, AC:10%, BLF:11%, ILF:8%	100.67s

From our end results, while they are not great, we can see that FastText seem to perform better than Word2Vec. This is most likely due to the fact how the vectors are calculated in FastText - Word2Vec looks at words individually for CBOW, while Fast Text looks at the words as n-grams.

What is surprising is that Skip Gram is supposed to look at the surrounding words when generating the vectors as well, but there is potentially too much noise added from this or the imbalance in the dataset is confusing the data.

The model itself seems to learn towards the negative padding, as it thinks it needs to go from positive numbers to negative numbers. There is also a heavy imbalance towards the B-O label in the dataset, which skews the results.

What could be done to avoid the negative padding is to one-hot encode all the labels in the future, as well as use ”windowing” - essentially turning each token into a collection of tokens that surround the word itself.

3.3 Experiment 3 - Pre-Trained Models

In this section we will explore two pre-trained models to label our tokens - DistilBERT and spaCy. We have been using DistilBERT throughout the experiments, but now we will explain how to build it, as well as with spaCy.

3.3.1 DistilBert

Since our data is already in collections, it will be easy to add it as we go. First thing we have to do is to take our tokens and encode them. Hugging face offers transformers that help us achieve this by calling the "AutoTokenizer" class. This will return our tokens as a series of input IDs (numbers) and an attention mask. These keys might vary depending on the model, but for DistilBERT, the two returned keys are input ids and attention mask [4].

```
from transformers import AutoTokenizer
from transformers import BatchEncoding
tokenizer = AutoTokenizer.from_pretrained("distilbert/distilbert-
    base-uncased")

tokenized_input:BatchEncoding = tokenizer(train_collection.
    get_token_list()[0], is_split_into_words=True) # convert tokens
     to numbers (input_ids) and attention_mask
tokenized_words = tokenizer.convert_ids_to_tokens(tokenized_input["
    input_ids"]) # converts inputs numbers to words again
```

Listing 24: Auto Tokeniser

This will yield a result as follows:

```
Original Tokens : ['for', 'this', 'purpose', 'the', 'gothenburg']
Tokenised Inputs  : [101, 2005, 2023, 3800, 1996, 22836]
Tokenised Inputs to Words : ['[CLS]', 'for', 'this', 'purpose', '
    the', 'gothenburg']
Original Token Len  : 15
Tokenised Input Len : 19
```

Listing 25: Auto Tokeniser output

There are two things that we will notice from here. The first is that when encoding happens with the Auto Tokeniser, it sometimes decides on its own to split the words even more. For instance, the word "gypes" has been split into "g", "ype" and "s". Secondly, there is a CLS and SEP added to the list of tokens. These two tags mean start and end of tokens (or a sentence) for the model to know.

Because of this, when we print the length of the original tokens and the new tokenised inputs, we notice that the lengths no longer match up. This will be problem for our labels, as the NER tags are set up to match the length of the tokens. In order to fix this issue, we have to write a function that will "pad" the tokens that have been added with -100.

Helpfully, the tokenizer has a function called "word_ids" that will return the ids of the words sequentially and mark special tokens as "none":

```
print(tokenized_input.word_ids())
[None, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 11, 12, 13, 14, None]
```

Listing 26: Auto Tokeniser word ids

As we can see, where the word ”gypes” is, it is represented by 3 sequential ”10”s that note that the word has been split into three. There is however 1 more issue - DistilBERT supports tokens of only size 512. Because the tokeniser splits our tokens into extra tokens, we need to make sure that we do not let the tokens become longer than 512. However, luckily this has already been handled by the tokeniser function by giving it ”max token length”.

With this, we are ready to perform processing our tokens into input ids and padding the ner tags:

```python
def tokenize_and_align_labels(data_collection:DataCollection) -> BatchEncoding:
    tokenized_inputs = tokenizer(data_collection.get_token_list(), truncation=True, is_split_into_words=True, max_length=512) # tokenise inputs

    labels = [] # create empty labels list to later matchs with tokenised inputs

    for i, label in enumerate(data_collection.ner_as_idx): # enumerate ner tags that we have converted to
        word_ids = tokenized_inputs.word_ids(batch_index=i)  # get word ids
        previous_word_idx = None # previous word index to check if same
        label_ids = [] # create current label ids list
        for word_idx in word_ids:  # for each index
            if word_idx is None:  # if index is none must be special token
                label_ids.append(-100) # append -100
            elif word_idx != previous_word_idx:  # Only label the first token of a given word.
                label_ids.append(label[word_idx]) # if does not equal previous word idx, append the label
            else:
                label_ids.append(-100) # if it does, the word has split so we add -100 again
            previous_word_idx = word_idx # set the current index as previous index for next check
        labels.append(label_ids) # on all processed, add to labels list

    tokenized_inputs["labels"] = labels # add to dictionary, will be input_ids, labels and attention mask
    return tokenized_inputs

def batch_list(batch:BatchEncoding):
    return [{"input_ids": inputs, "labels": labels} for labels, inputs in zip(batch["labels"], batch["input_ids"])]
```

Listing 27: Tokenise and align with Auto Tokeniser

Because the DistilBERT model requires the inputs to be in a dictionary list, the batch list function helps return that. Now that we have tokenised all of our collections that we will test on, we will be able to generate a metric function to run when we are training our dataset. The metric function is important to progressively test our dataset and see how well it is doing as some models are capable of adjusting rates and values depending on progress.

```python
from datasets import load_metric
import evaluate
seqeval = evaluate.load("seqeval")
metric = load_metric("seqeval")

def compute_metrics(p):
    predictions, labels = p
    predictions = np.argmax(predictions, axis=2)

    true_predictions = [
        [tag_list[p] for (p, l) in zip(prediction, label) if l !=
    -100]
        for prediction, label in zip(predictions, labels)
    ]
    true_labels = [
        [tag_list[l] for (p, l) in zip(prediction, label) if l !=
    -100]
        for prediction, label in zip(predictions, labels)
    ]

    results = seqeval.compute(predictions=true_predictions,
    references=true_labels)
    return {
        "precision": results["overall_precision"],
        "recall": results["overall_recall"],
        "f1": results["overall_f1"],
        "accuracy": results["overall_accuracy"],
    }
```

Listing 28: Metric Function

The metric function uses the labels to check if something is to be ignored. If not, it uses the predictions and labels it gets and adds them to a general list where evaluations can be made. Finally, we load the model and the dataloader and train:

```python
from transformers import AutoModelForTokenClassification,
    TrainingArguments, Trainer, DataCollatorForTokenClassification
data_collator = DataCollatorForTokenClassification(tokenizer=
    tokenizer)
task = "ner" # Should be one of "ner", "pos" or "chunk"
model_checkpoint = "distilbert-base-uncased"

def get_training_args(
        out_dir:str,
        learning_rate:float=2e-5,
        batch_size:int=16,
        epochs:int=2,
        weight_decay:float=0.01,
        evaluation_strategy:str="epoch",
        save_strategy:str="epoch",
        lr_scheduler_type="linear") -> TrainingArguments:
    return TrainingArguments(
        output_dir=out_dir,
        learning_rate=learning_rate,
        per_device_train_batch_size=batch_size,
        per_device_eval_batch_size=batch_size,
        lr_scheduler_type=lr_scheduler_type,
        num_train_epochs=epochs, # number of epochs to train, can be overriden by max steps
        weight_decay=weight_decay, # The weight decay to apply (if not zero) to all layers except all bias and LayerNorm weights
        evaluation_strategy=evaluation_strategy, # evaluate at the end of each epoch
        save_strategy=save_strategy, # can save by epoch, steps or not at all
        save_total_limit=1, # how many checkpoints to keep before overriding (set to 1, so latest checkpoint is only kept)!
        load_best_model_at_end=True,
        report_to=['none'], # REQUIRED because otherwise keeps asking to log into "wandb",
        overwrite_output_dir=True,
    )

def get_trainer(model, training_args:TrainingArguments, tokenised_train:list[dict], tokenised_eval:list[dict]) -> Trainer:
    return Trainer(
        model=model, # model we use for training
        args=training_args, # model arguments
        train_dataset=tokenised_train, # train dataset tokenised
        eval_dataset=tokenised_eval, # testing dataset tokenised for model training evaluation
        tokenizer=tokenizer, # which tokeniser are we using
        data_collator=data_collator, # data collector pads the tokens along with the labels
        compute_metrics=compute_metrics, # function to compute metrics on how well we are scoring
    )
```

Listing 29: Train Distil BERT

When doing the predictions, we use the same method as with computing metrics to get the labels, then using ConfusionMatrixDisplay from sklearn, we plot the dataframes and display them.

3.3.2 spaCy

For spacy, we will need to setup the required libraries and paths first, following the training guidelines [6]. Once that is complete, we will start to create a vocabulary. In order to do so, we have to load our train and test dataset into the ”doc”. This is spaCys way to index words and labels. We create spaces where the words are separated and then add both words and ner labels to the vocabulary:

```python
train_docbin:DocBin = DocBin()
```

```
test_docbin:DocBin = DocBin()

def dataset_to_vocab(collection:DataCollection, doc_bin:DocBin) ->
    dict:
    for data_item in collection.collection:
        spaces = [True if token not in string.punctuation else
    False for token in data_item.tokens]
        doc = Doc(nlp.vocab, words=data_item.tokens, spaces=spaces,
     ents=data_item.ner)
        doc_bin.add(doc)

dataset_to_vocab(train_collection, train_docbin)
dataset_to_vocab(test_collection, test_docbin)

train_docbin.to_disk(train_vocab_path)
test_docbin.to_disk(dev_vocab_path)
```

Listing 30: spaCy DocBin

With the docbin vocabulary built, we can train our model. SpaCy trains its epochs until the loss function begins to start to stop if the epochs are left at 0.

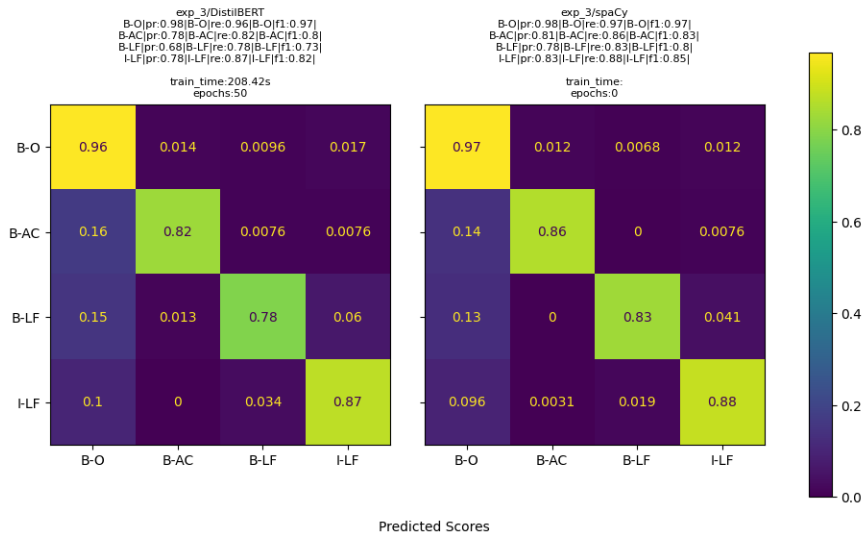
```
if retrain_exp_3:
    from spacy.cli.train import train
    train(config_path=config_path, output_path=output_dir,
    overrides={"paths.train": train_vocab_path, "paths.dev":
    dev_vocab_path}, use_gpu=0)
```

Listing 31: spaCy DocBin

Once the training is finished, we have to evaluate our model. The general issue here is that spacy, like the autotokeniser in BERT, likes to break words into more subsections. This means that our predicted labels and true labels will not match up. Furthermore, because spacy is a bit more clever, it will automatically combine our longform labels into a singular ”LF” tag, but that does not make it easy for us to recognise prediction accuracy between B-LF and I-LF. However, because we know that I-LF follow B-LF, then we can re-split the predictions to match.

To fix that we will have to pad the tokens with invalid tags if they are shorter and otherwise pad labels if they are longer. However, once we do it successfully, we can plot our results.

Training Arguments	
Learning Rate	2e-5
Optimiser	Adam



Experiment 1 Results		
Test	F1 Score	Time
DistilBERT	O:97%, AC:80%, BLF:73%, ILF:82%	208.42s
spaCy	O:97%, AC:83%, BLF:80%, ILF:85%	1104.01s

Both models perform almost indentically well - spaCy manages to be slightly better at at all of the labels. but DistilBERT is not far behind in calculations. However, spaCy does reach an f1 score of over 80% for each label, meaning it did perform better. This is however the case that spaCy has been allowed to train as longer, meaning that it will be more accurate in it's resolution.

However, DistilBERT has trained very close results in 1/4 of the time and thus is a better contender for getting accurate results quickly.

3.4 Experiment 4 - Hyper Parameters

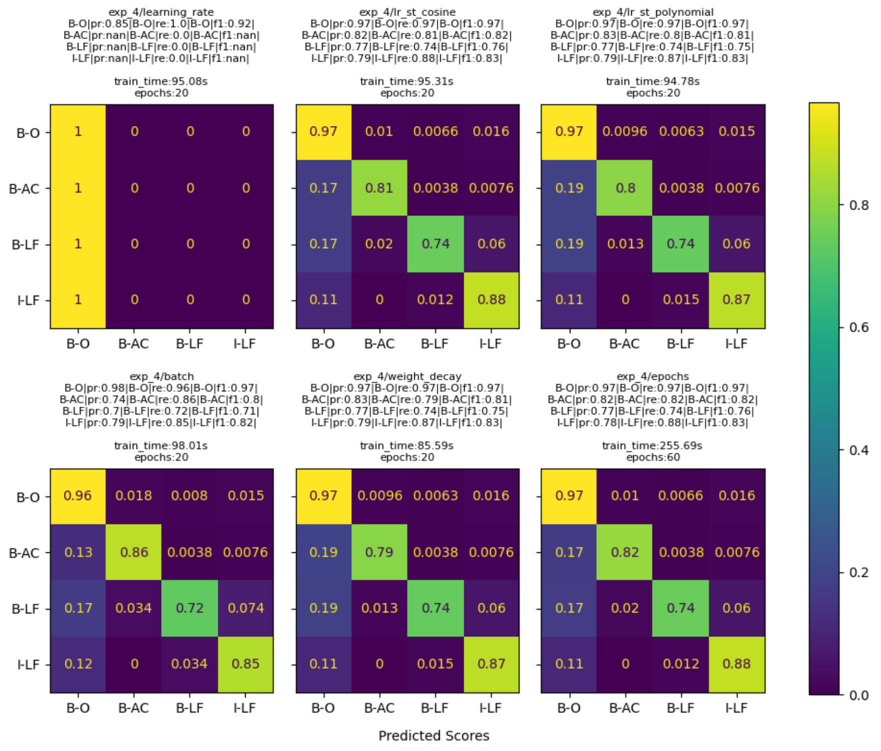
In this section we are finally exploring different parameters and how they affect our training and testing, as well as what the results of each of these tests will be.

The 6 final tests with the hyper parameters will be as following:

1. Learning Rate
2. Learning Rate Scheduler - Cosine
3. Learning Rate Scheduler - Polynomial
4. Batch Size
5. Weight Decay
6. Epochs

As we are using the DistilBERT model to hyper parameterise we can rerun our pipeline setup before and change arguments as we go. As the pipeline has an easy way to make training arguments [5] we will explore the hyper parameters made available to us [2].

The results of the testing are as follows:



```
exp_4/learning_rate
B-O|pr:0.85|B-O|re:1.0|B-O|f1:0.92|
B-AC|pr:nan|B-AC|re:0.0|B-AC|f1:nan|
B-LF|pr:nan|B-LF|re:0.0|B-LF|f1:nan|
I-LF|pr:nan|I-LF|re:0.0|I-LF|f1:nan|

train_time:95.08s
epochs:20

exp_4/lr_st_cosine
B-O|pr:0.97|B-O|re:0.97|B-O|f1:0.97|
B-AC|pr:0.82|B-AC|re:0.81|B-AC|f1:0.82|
B-LF|pr:0.77|B-LF|re:0.74|B-LF|f1:0.76|
I-LF|pr:0.79|I-LF|re:0.88|I-LF|f1:0.83|

train_time:95.31s
epochs:20

exp_4/lr_st_polynomial
B-O|pr:0.97|B-O|re:0.97|B-O|f1:0.97|
B-AC|pr:0.83|B-AC|re:0.8|B-AC|f1:0.81|
B-LF|pr:0.77|B-LF|re:0.74|B-LF|f1:0.75|
I-LF|pr:0.79|I-LF|re:0.87|I-LF|f1:0.83|

train_time:94.78s
epochs:20

exp_4/batch
B-O|pr:0.98|B-O|re:0.96|B-O|f1:0.97|
B-AC|pr:0.74|B-AC|re:0.86|B-AC|f1:0.8|
B-LF|pr:0.7|B-LF|re:0.72|B-LF|f1:0.71|
I-LF|pr:0.79|I-LF|re:0.85|I-LF|f1:0.82|

train_time:98.01s
epochs:20

exp_4/weight_decay
B-O|pr:0.97|B-O|re:0.97|B-O|f1:0.97|
B-AC|pr:0.83|B-AC|re:0.79|B-AC|f1:0.81|
B-LF|pr:0.77|B-LF|re:0.74|B-LF|f1:0.75|
I-LF|pr:0.79|I-LF|re:0.87|I-LF|f1:0.83|

train_time:85.59s
epochs:20

exp_4/epochs
B-O|pr:0.97|B-O|re:0.97|B-O|f1:0.97|
B-AC|pr:0.82|B-AC|re:0.82|B-AC|f1:0.82|
B-LF|pr:0.77|B-LF|re:0.74|B-LF|f1:0.76|
I-LF|pr:0.78|I-LF|re:0.88|I-LF|f1:0.83|

train_time:255.69s
epochs:60
```

Experiment 1 Results		
Test	F1 Score	Time
Learning Rate	O:92%, AC:0%, BLF:0%, ILF:0%	95.08s
Scheduler - Cosine	O:97%, AC:82%, BLF:76%, ILF:83%	95.31s
Scheduler - Polynomial	O:97%, AC:81%, BLF:75%, ILF:83%	94.78s
Batch Size	O:97%, AC:80%, BLF:71%, ILF:82%	98.01s
Weight Decay	O:97%, AC:81%, BLF:75%, ILF:83%	85.59s
Epochs	O:97%, AC:82%, BLF:76%, ILF:83%	255.69s

Most of the results are as expected, the only exception being the learning rate, as the original hypothesis would have been that too large or a learning rate would decrease the score, but it has made the model completely unable to learn. However, on further query, this does make sense as it is most likely only able to learn the largest imbalance in the dataset, which is B-O. This further proves that the dataset imbalance makes it difficult for the model to learn.

It also makes sense that the best hyper parameter result is having more epochs to train, as it gives the model more time to learn. We can also see that slightly adjusting hyper parameters makes relatively no difference to the model.

4 Conclusion

4.1 Experiment Result

In experiment 1, the best performing data results were the original tokens and lemmatisation, while the worst was creating unique words with stopwords and punctuations removed. This makes sense however, as making all the words unique removes too much context around the words, while having more data, means that pre-trained models are more capable of using that data.

Experiment 2 holds overall the worst result, as the CNN model only reachs a maximum of 1% for some of the label f1 scores. It proves that an untrained model is not able to perform well on an unbalanced dataset.

As mentioned above, there are a multitude of things that could be further attempted - balancing the dataset, windowing and one-hot encoding - to avoid negative padding issues. Between the tests themselves however, the best result seemed to be FastText, most likely due to the ability of being able to generate n-grams and thus creating more meaning per word.

Experiment 3 produced overall some of the best results as both DistilBERT and spaCy came above 80% in almost all the f1 score. Both of them out perform other models simply due to the fact that they have already been built to capture the semantics of sentences before fine tuning it to the dataset. Thus, it does not need to start from scratch, unlike the CNN model.

While the DistilBERT model itself is slightly better, given more time to train, potentially spaCy could surpass that. Where DistilBERT does excel however, is time. It takes far less time to train the model than with spaCy and both of them achieve relatively similar results. Not only this, but they are far easier to configure and run than building from scratch and achieve much greater results.

In experiment 4, almost all the experiments reached a scored highly, the only exception to this was changing the learning rate to 2e-3 from 2e-5 which make the model only able to recognise everything as a B-O label, being unable to recognise anything. This makes also sense as the learning rate controls how ambitious the model is in changing its weights. Having too high a learning rate would cause it to be unable to predict anything, as has happened here.

4.2 Outcome

4.2.1 Can the models you built fulfil their purpose?

The overall experiments achieved the tasks that they were set out to do. Testing with data processing, the outcome of the results was mostly as anticipated - more data, even if imbalanced, is better for models that have already learned how to recognise and train on semantics of sentences, while the data being unbalanced causes issues with untrained models.

4.2.2 What is good enough F1/accuracy?

The highest achieved result was achieved by the spaCy model getting f1 scores over 80% for all. This is excellent accuracy, considering how heavily the dataset is imbalanced. B-O makes up over 82% of the dataset, while B-AC makes up about 6%, B-LF about 4% and I-LF about 8%.

This means that even with only 4% label frequency, spaCy manages to handle most cases and predict with a degree of confidence a correct label. To give the models some room for inaccuracy, for any label to be with an accuracy of above 70% would be excellent and with the imbalance, it would be good for any long from or abbreviation to be predicted above 50%.

4.2.3 If any of the models did not perform well, what is needed to improve?

Looking at the CNN model, definitely the issue here lies in the imbalanced data, which could be easily remedied by having more samples of the abbreviation and long-form tags. However, semantics of a sentence might be difficult to construct as such.

For the CNN model that did not perform well at all, the further tests should include trying to balance the dataset, building one hot encoding into the labels and trying to play around with different vector sizes and potentially trying windowing. All of these might give more context to the model to learn from.

Another option to improve the model in terms of vectorisation is to "cluster" words close to each other. Clustering is a method where words close to each other are assigned to "n" clusters. While this could also be detramental for indvidual word learning, the model could learn generalisation of the words and thus perform better.

4.2.4 If any of the models performed really well, could/would you make it more efficient and sacrifice some quality?

As the DistilBERT model performed well with the un-cased model, it would be interesting to see if a larger model or even just BERT could out perform it in terms of accuracy. However, we also need to account for speed of the model, as being slightly more accurate is not worth a massive time increase cost.

Another interesting test could be to hyper parameterise the spaCy model to see if that would yield better results than DistilBERT.

5 References

References

[1] *Defining a Neural Network in PyTorch — PyTorch Tutorials 2.3.0+cu121 documentation*. `https://pytorch.org/tutorials/recipes/recipes/defining_a_neural_network.html`. (Accessed on 04/26/2024).

[2] *Hyperparameter Optimization*. `https://huggingface.co/docs/setfit/en/how_to/hyperparameter_optimization`. (Accessed on 04/26/2024).

[3] *NLP 101: Word2Vec — Skip-gram and CBOW — by Ria Kulshrestha — Towards Data Science*. `https://towardsdatascience.com/nlp-101-word2vec-skip-gram-and-cbow-93512ee24314`. (Accessed on 04/26/2024).

[4] *Token Classification - Colab*. `https://colab.research.google.com/github/huggingface/notebooks/blob/main/examples/token_classification.ipynb`. (Accessed on 04/26/2024).

[5] *Trainer*. `https://huggingface.co/docs/transformers/en/main_classes/trainer`. (Accessed on 04/26/2024).

[6] *Training Pipelines & Models · spaCy Usage Documentation*. `https://spacy.io/usage/training`. (Accessed on 04/26/2024).

[7] *Word representations · fastText*. `https://fasttext.cc/docs/en/unsupervised-tutorial.html`. (Accessed on 04/26/2024).