# NLP Assignment 1

Muhammad Usman Shahid

## Generating the 80/20% Split

The first question required splitting the data into 80%/20% training and testing datasets. The training split was created by generating a random list of 6253 numbers between 0 and between 7815. These numbers were used as list indexes to pick out 6253 random data values from the pre-processed training data.

The remaining numbers between 0 and 7815 that were not present in the testing index list were extracted and used to pick out the rows to be used for the 20% testing dataset from the preprocessed training data.

## False Positive Error Analysis

The classes with the lowest precision were `B-Opinion`, `I-Opinion`, and `B-Quote`. Most of the `O` words were incorrectly classified as `B-Opinion` and `I-Opinion` which hints at the classifier needed more context about these words. Also, for these classes, the total population sample size was very low, so even a small number of misclassifications resulted in a drastically low precision.

False Negative Error Analysis  
  
The classes with the lowest recall were `O`, `I-Plot`, `B-Plot`, `B-Origin` and `B-Character`. It is evident from the confusion matrix that a significant number of `O` tags were misclassified. A significant number of `B-Plots` were misclassified as `I-Plot` which shows the tagger had difficulty understanding the beginning of the Plot class.  
  
Incorporating POS tags as Features

Incorporating POS tags as features required making changes to the Preprocessing and Feature Extraction functions.

In the preprocessing function, we are extracting every word in the training data from the word-tag tuple and storing it in a list of words. The list of words is then passed through the POS tagger using the tag() method of the tagger.

The POS tag for each word is the extracted, concatenated to each original word with the `@` separator to make splitting later on easier. This `word@tag` data point needs to be updated in the original list of tuples that will later be used as an input to the CRF tagger. Unfortunately, this turned out to be a bit of a problem since the tuple data type is immutable and does not support item assignment. This limitation was overcome by casting each tuple element to a list, updating the required information and then casting it back to a tuple.

The POS tag information was extracted from the `word@tag` structure by splitting on the `@` delimiter and the POS tag was added to the feature list.

Incorporating the POS Tag, however, did not have a significant impact on the performance of the tagger.

The Macro Average F1 score for this tagger was 58.25% as compared to 57.16% with the default feature set.

## Feature extraction and optimization

Since the addition of the POS tags didn’t lead to a decent improvement in performance, we looked at some other features that could be used.

The first idea was to increase the length of Suffixes from 3 to 5 in the feature list and add 3 Prefixes. The logic behind this idea was to give the tagger some more context about the word that might help it understand better.

This worked better than just adding the POS Tag feature as the Macro Average F1 score went up to 59.93%.

To take the approach further, we tried to increase the suffix length to 5 which further improved the performance of the tagger. It showed an improvement of 0.79% as the macro average F1 score went up to 60.72%. This clearly showed that increasing the length of suffixes and prefixes would be good features for the CRF tagger.

Another possible feature idea was to add the length the token as a feature. The length of a word can provide a hint to its tag as for example, actor names will generally be longer than other words. Unfortunately, implementing this feature brought down the macro average F1 score to 60.43% which indicated that this might not be a very good feature. One possible issue with this, however, could be that the word-length feature was being passed to the tagger as a string in the feature\_list. This feature might be more effective if the tagger interprets it as numerical.

The next feature engineering we tried was to provide a sliding window of the previous and next token to the word in question to the tagger. For example to tag the word `likes` in `John likes Bill and Mary`, the tagger would be forwarded the words `John` and `Bill` as well. The idea here was that having some context behind a word might help improve tagging performance.

This ended up being true because the model showed a 2.2% improvement as compared to the previous models. This was also the highest increase in performance we had seen in all the feature engineering techniques used.

Just to test if the performance improvement was consistent, we increased the sliding window size to 2 (2 previous and 2 next words). This increased the performance but not as drastically as the first time.

We could also tell that the sequence window was a good feature because it drastically brought down the number of False Positives and False Negatives in classes which had the most misclassifications. For the class `I-Director`, for example, the false positives were brought down from 109 to 59 and for class `O`, this was even more drastic with a decrease in false positives of 232. False negatives for both these classes went down as well but not as drastically (they had relatively lower number of false negatives anyway).

The results for all the feature engineering techniques used are summarized in the following table:

|  |  |  |
| --- | --- | --- |
| **Data** | **Feature Engineering** | **Macro Avg. F1 Score (%)** |
| 20%  Test data | Default | 57.16 |
| 20%  Test data | POS Tags | 58.25 |
| 20%  Test data | Added 5 suffixes and 3 prefixes | 59.93 |
| 20%  Test data | Increased prefixes to 5 | 60.72 |
| 20%  Test data | Added word length | 60.43 |
| 20%  Test data | Added Next + Prev word window | 62.84 |
| 20%  Test data | Window size increased to 2 | 64.84 |
| Holdout data | Final feature engineered model | 63.76 |

## Conclusion

We tried several feature engineering techniques to improve the overall performance of the CRF tagger. The technique that worked the best was adding information about the next and previous words as features for the tagger.

The model can be further improved through more sophisticated feature engineering as well as by optimizing it for the data through hyperparameter tuning. All the models during this exercise were trained on default hyperparameters except for the minimum feature frequency which was set at 2.