Classifying Sentences Inside Research Articles using NLP Techniques

Connor Forbes, Matthew Ellis (Dropped course)

Introduction

A Systematic Review (SR) is a literature review which is designed to combine multiple Randomized Controlled Trials (RCTs) in order to synthesize findings from all research in a given field. An example of this may be a SR which is designed to find the effectiveness of paracetamol on treating headaches. All research involving the effectiveness of paracetamol at treating headaches is analyzed and synthesized in a way that allows a stronger conclusion (compared to the individual RCTs) establishing how effective paracetamol is at treating headaches.

However, the number of Randomized Controlled Trials (RCTs) published each year is growing at an exponential rate, while the number of Systematic Reviews (SRs) to synthesise these RCT into usable clinical guidelines struggles to keep up with the growing demand (see Figure 1 below).

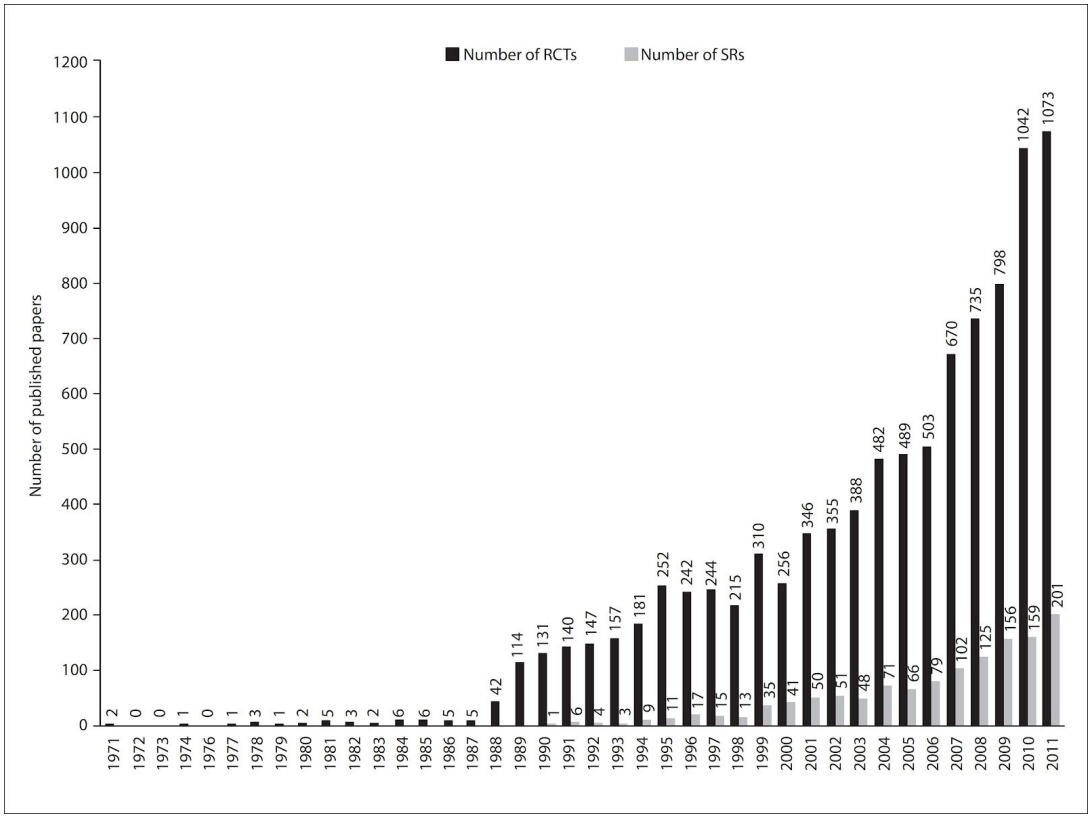


Figure 1: Year by Year Comparison of number of RCTs and SRs (Grande, et al., 2014)

In order to help with the growing demand for systematic reviews, multiple SR automation tools have been developed. However, the field of Natural Language Processing (NLP) has not been fully explored in this domain yet. The goal of this assignment is to begin implementing text classification on a freely available real world dataset (Pubmed 20k), in order to classify sentences into multiple classes (multi-class classification). These classes are ‘background’, ‘objective’, ‘methods’, ‘results’, ‘conclusions’.

The objective of the classification is to allow a large amount of RCTs to be compared at once, with the relevant sections being automatically extracted. For example, when screening multiple articles to decide if they should be included/excluded from the SR, the researcher will analyse the background of each research paper. By classifying the background within the paper, it may be possible to easily compare the background of multiple studies, eliminating the need for the researcher to manually find these sections in the article PDF.

Solution

Due to the large amount of sentences associated with the dataset (88,797 sentences in the training dataset), a subset of the data was used in the solution to keep training times reasonable due to hardware limitations. For this model, the first 2500 sentences in the dataset were selected as the training data with the following 1000 sentences selected as testing data.

```
train_data = df['scibert'][:2500].tolist()
train_label = df['code'][:2500].tolist()
test_data = df['scibert'][2500:3500].tolist()
test_labels = df['code'][2500:3500].tolist()
```

Embedding Sentences with SciBERT

Because the dataset is presented in string format, each sentence must be converted into vector representation so the k-Nearest Neighbor (kNN) algorithm can be applied. Initially in the first part of this assignment, it was explored training an embedder in keras from scratch. However, upon further investigation a transformer network known as SciBERT was discovered which was trained on a corpus of 1.14 million full-text research papers (Beltagy et al., 2019). Due to how well the pre-trained data matches the selected problem and given the size of the training data, the SciBERT transformer was selected to perform sentence embedding and convert each sentence to a vector with 768 dimensions (code below).

```
def get_embedding(model, tokenizer, text):

    # Encode with special tokens ([CLS] and [SEP], returning pytorch
tensors
    encoded_dict = tokenizer.encode_plus(
                        text,
                        add_special_tokens = True,
                        return_tensors = 'pt'
                )

    input_ids = encoded_dict['input_ids']
    # Set model to evaluation mode
    model.eval()
    # Run through BERT
    with torch.no_grad():
        outputs = model(input_ids)
        # Extract hidden states
        hidden_states = outputs[2]

    # Select the embeddings
    token_vecs = hidden_states[-2][0]

    # Calculate average of token vectors
    sentence_embedding = torch.mean(token_vecs, dim=0)
```

```
    # Convert to np array
    sentence_embedding = sentence_embedding.detach().numpy()

    return sentence_embedding
```

kNN with Scikit-Learn

The first toolkit which was used for the implementation of the kNN algorithm was scikit-learn's KNeighborsClassifier. This classifier by default will use euclidean distance as the metric to calculate the closest neighbors, however by experimentation it was discovered that using manhattan distance can improve the accuracy of the algorithm from 77.8% to 78%. Furthermore, by setting 'n_jobs = -1', all processors can be utilized, decreasing the processing time from 3.24 seconds to 0.65 seconds. Furthermore, by setting the number of neighbors used in evaluation to 8, the accuracy can be further improved to 79.3%. The resulting code for implementing the library is given below.

```
# kNN with scikit-learn
import timeit
import seaborn as sns
from sklearn.neighbors import KNeighborsClassifier

model = KNeighborsClassifier(8, metric="manhattan", n_jobs=-1)
# Fit models
start = timeit.default_timer()
model.fit(df['scibert'][:2500].tolist(), df['code'][:2500].tolist())
# Predictions for each model
pred = model.predict(df['scibert'][2500:3500].tolist())
stop = timeit.default_timer()
print("Time elapsed:", stop-start)
# Evaluate
print("F1 Score")
print(f1_score(df['code'][2500:3500].tolist(), pred,
average="macro"))
print("MCC")
print(matthews_corrcoef(df['code'][2500:3500].tolist(), pred))
print("Accuracy")
print(accuracy_score(df['code'][2500:3500].tolist(), pred))
plt.figure(figsize=(10,8))
sns.heatmap(confusion_matrix(df['code'][2500:3500].tolist(), pred),
```

```
annot=True)
plt.show()
```

kNN with Weka

The second toolkit for which an implementation for kNN was investigated is Weka. For this, the data must first be converted to a compatible CSV format and then loaded into Weka. Two CSV were generated, a training CSV and test CSV using the code below.

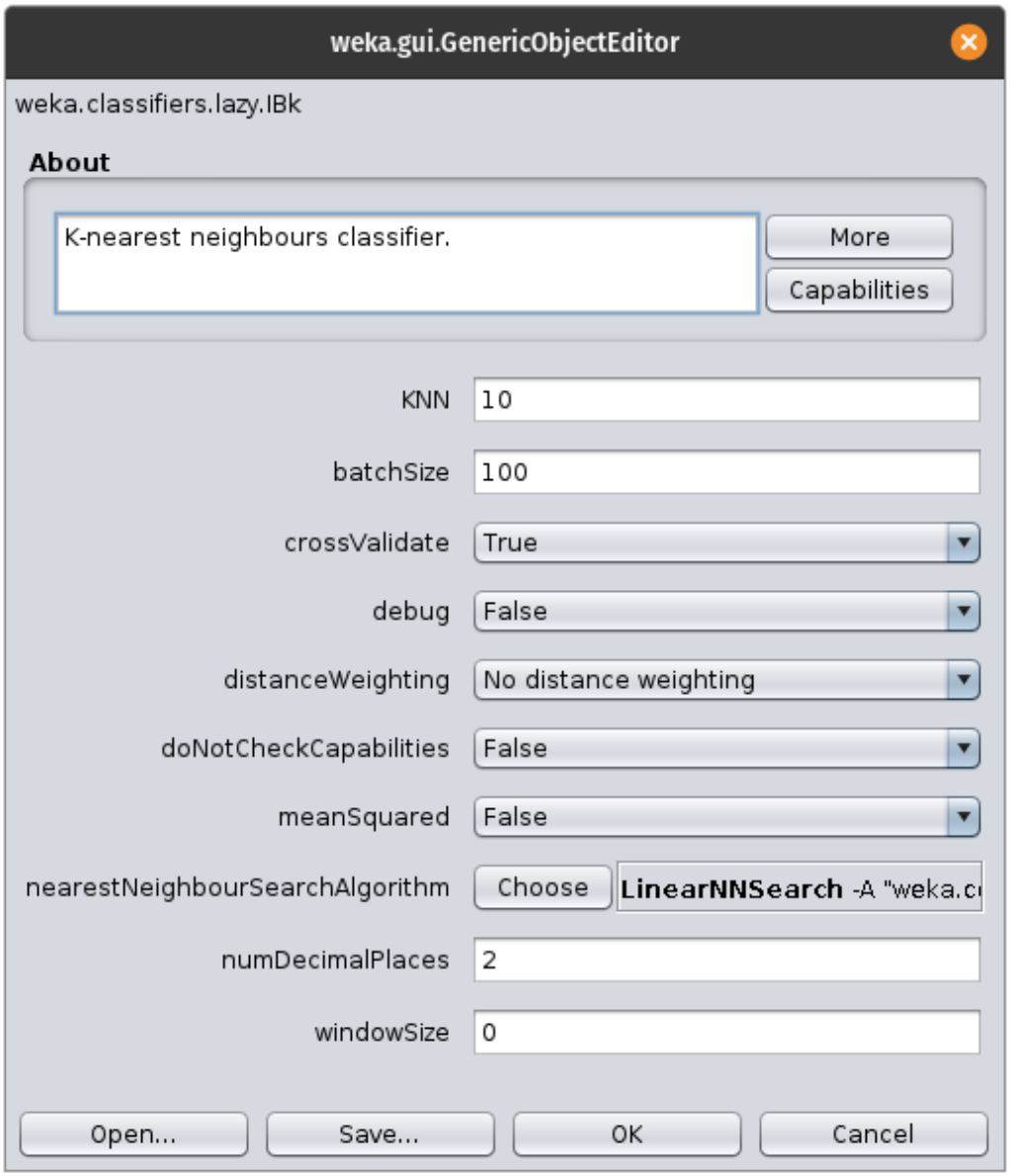
```
# Export to weka
export_df = pd.DataFrame(df['scibert'][:2500].tolist(),
index=df.index[:2500])
export_df['label'] = df['label'][:2500]

print(export_df.head())

export_df.to_csv('./weka.csv', index=False)

export_df = pd.DataFrame(df['scibert'][2500:3500].tolist(),
index=df.index[2500:3500])
export_df['label'] = df['label'][2500:3500]
print(export_df.head())
export_df.to_csv('./weka_test.csv', index=False)
```

In weka, the option for testing was to use the ‘Supplied test set’. The classifier for kNN can be found in Weka under ‘weka/classifiers/lazy/IBk’. The settings used for the classifier are below:



It was chosen to implement cross validation to pick the best k value between 1 and 10. Furthermore, as above the manhattan distance was used as the distance metric.

```
IBk -K 10 -W 0 -X -A "weka.core.neighboursearch.LinearNNSearch -A \"weka.core.ManhattanDistance -R first-last\""
```

The cross validation supported the decision with scikit-learn, as weka determined the optimal number of neighbors to use as 8 achieving an accuracy of 79.7%.

Coding kNN from scratch

When implementing the kNN from scratch, parameters selected were based on those found to be ideal above, namely setting the k-value (number of neighbors) to 8 and using the manhattan distance as the metric. In the case of a tie, the label for the first appearing neighbor will be selected. From initial implementation, testing was performed on 20 sentences, however

the time to run the algorithm on just 20 sentences was 20.34 seconds. To optimize this, it was decided to implement numpy to perform the Manhattan distance calculations in parallel and amazingly this cut the processing time from 20.34 seconds to just 0.35 seconds for 20 seconds. The algorithm was then tested on the full dataset and achieved an accuracy of 79.3%, matching the accuracy achieved with scikit-learn, suggesting the implementation is correct. The code for this implementation is below:

```
# kNN improved with numpy
import timeit
# import Numpy for more efficient manhattan distance
import numpy as np

def knn(k, test_data, train_data, train_label):
    test_arr = np.array(test_data)
    train_arr = np.array(train_data)
    pred = []
    # Iterate over each input data case
    for test_embedding in test_arr:
        # List to hold (distance, label)
        distances = []
        # Calculate distance of data point from each train_data case
        for i in range(len(train_arr)):
            manhattan_dist =
np.sum(np.abs(test_embedding-train_arr[i]))
            # Append (distance, label)
            distances.append((manhattan_dist, train_label[i]))
        # Find k-nearest neighbors
        k_smallest = heapq.nsmallest(k, distances)
        # Select just labels
        labels = [lis[1] for lis in k_smallest]
        # Find mode (in tie will select first appearing neighbor)
        label = max(set(labels), key=labels.count)
        pred.append(label)
    return pred

train_data = df['scibert'][:2500].tolist()
train_label = df['code'][:2500].tolist()
test_data = df['scibert'][2500:3500].tolist()
```

```
# Run knn
start = timeit.default_timer()
pred = knn(8, test_data, train_data, train_label)
stop = timeit.default_timer()
print("Time elapsed:", stop-start)

# Evaluate
print("F1 Score")
print(f1_score(df['code'][2500:3500].tolist(), pred,
average="macro"))
print("MCC")
print(matthews_corrcoef(df['code'][2500:3500].tolist(), pred))
print("Accuracy")
print(accuracy_score(df['code'][2500:3500].tolist(), pred))
plt.figure(figsize=(10,8))
sns.heatmap(confusion_matrix(df['code'][2500:3500].tolist(), pred),
annot=True)
plt.show()
```

Results and Metrics

The primary metrics chosen in comparison of the three solutions is accuracy, the weighted F-measure and Matthew's Correlation Coefficient (MCC). MCC was chosen as a metric for analysis due to the fact that accuracy and F-measure can often give over-optimistic, inflated results on imbalanced datasets (Chicco and Jurman, 2020). The MCC is a measure which produces a high score only if the algorithm performs well in all four confusion matrix categories.

Comparison of the Implemented Solutions

	Accuracy	F-Measure (weighted)	Matthew's Correlation Coefficient
Scikit-learn	0.793	0.791	0.721
Weka	0.797	0.794	0.743
Coded	0.793	0.791	0.721

Table 1: Accuracy Metrics for the kNN Solutions

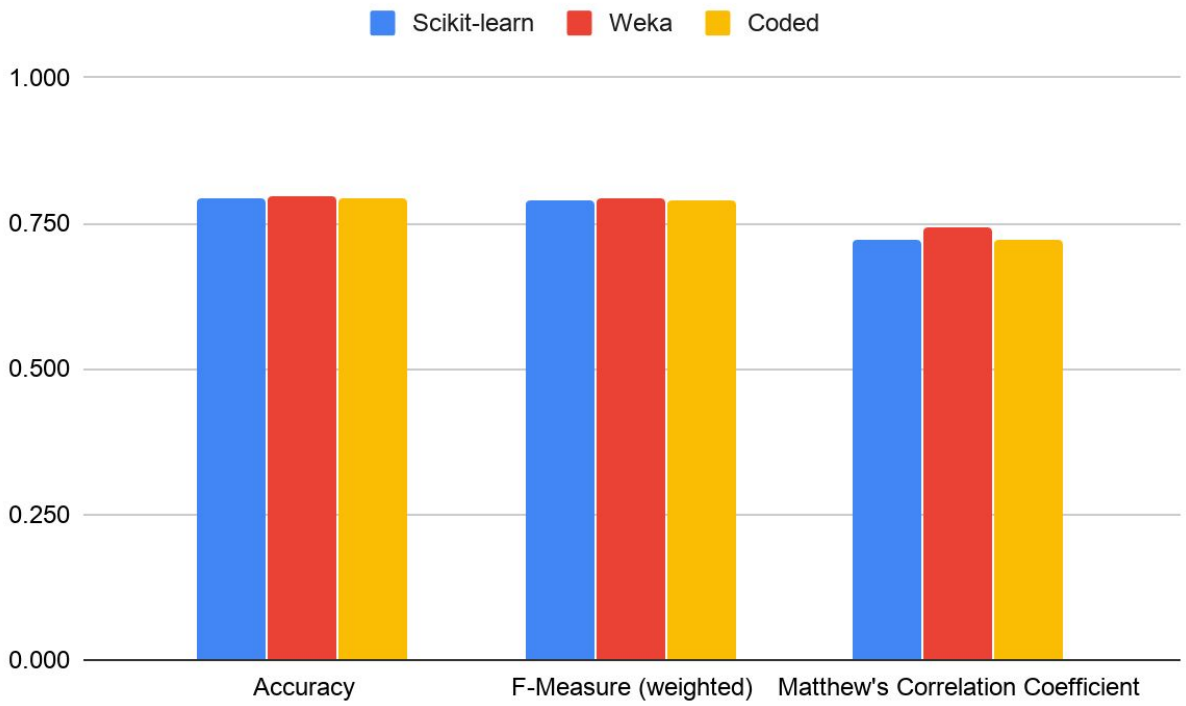


Figure 2: Accuracy, F-Measure and Matthew's Correlation Coefficient for the kNN Solutions

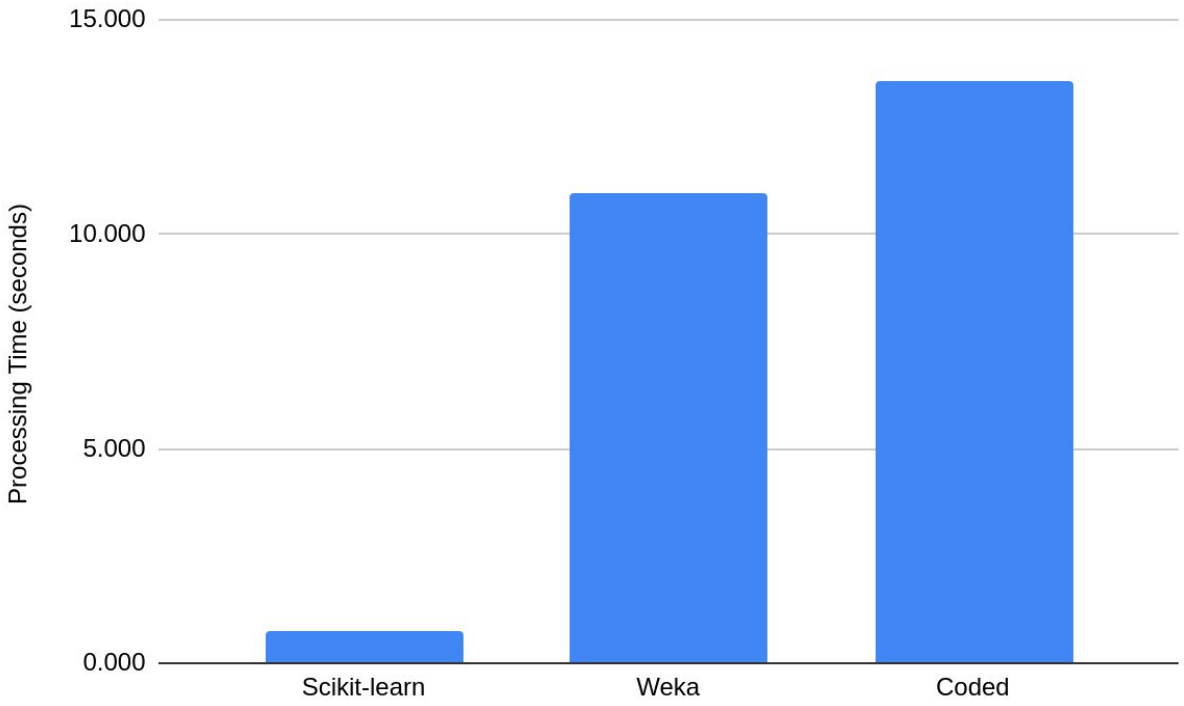


Figure 3: Processing Time for the kNN Solutions

As is shown above, in terms of accuracy all 3 solutions perform similarly with Weka performing slightly better than the scikit-learn and coded implementation. There should be no reason for this considering all parameters are the same for the three implementations, however it is estimated that Weka may use a different algorithm for determining the class in the event of a tie. The scikit-learn and coded solutions both select the first appearing class, however Weka may reduce the number of neighbors until there is no longer a tie (stopping at k=1).

Despite this, scikit-learn performs much faster in comparison to the other two solutions (see Figure 3), most likely due to the fact that it includes optimizations such as BallTree or KDTree to improve the efficiency of the algorithm in addition to better utilization of multi-threading.

In-Depth Analysis of Metrics

Scikit-Learn

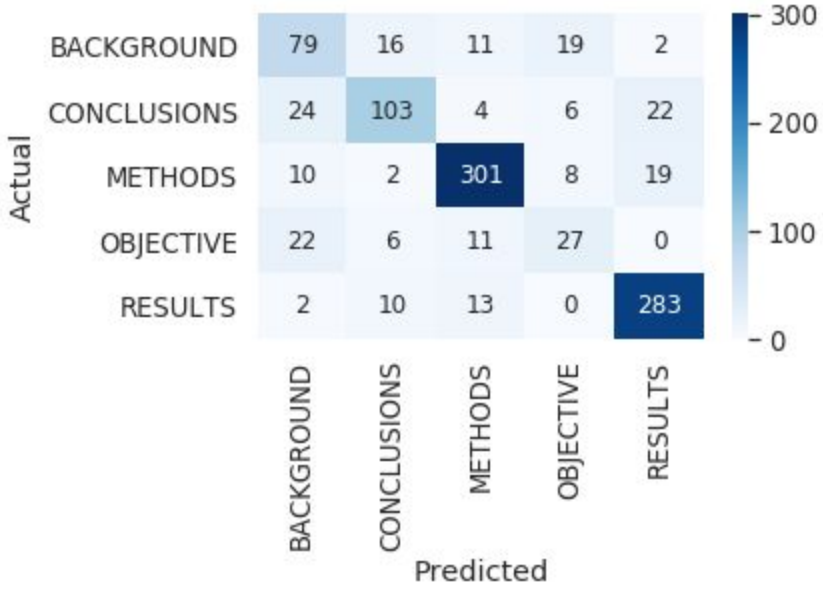


Figure 4: Confusion Matrix for Scikit-Learn Results

	precision	recall	f1-score	support
BACKGROUND	0.58	0.62	0.60	127
CONCLUSIONS	0.75	0.65	0.70	159
METHODS	0.89	0.89	0.89	340
OBJECTIVE	0.45	0.41	0.43	66
RESULTS	0.87	0.92	0.89	308
accuracy			0.79	1000
macro avg	0.71	0.70	0.70	1000
weighted avg	0.79	0.79	0.79	1000

Table 2: Scikit-Learn Precision, Recall and F1-Score for each Class

Weka

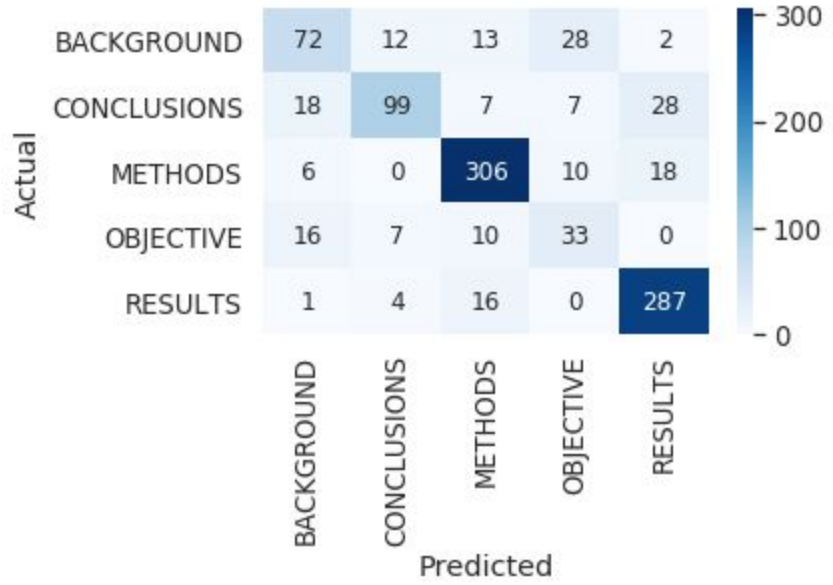


Figure 5: Confusion Matrix for Weka Results

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0.500	0.048	0.423	0.500	0.458	0.418	0.869	0.391	OBJECTIVE
	0.900	0.070	0.869	0.900	0.884	0.824	0.955	0.910	METHODS
	0.932	0.069	0.857	0.932	0.893	0.844	0.970	0.922	RESULTS
	0.567	0.047	0.637	0.567	0.600	0.547	0.898	0.567	BACKGROUND
	0.623	0.027	0.811	0.623	0.705	0.665	0.919	0.732	CONCLUSIONS
Weighted Avg.	0.797	0.059	0.797	0.797	0.794	0.743	0.941	0.808	

Table 3: Weka Precision, Recall and F1-Score for each Class

Python Coded from Scratch

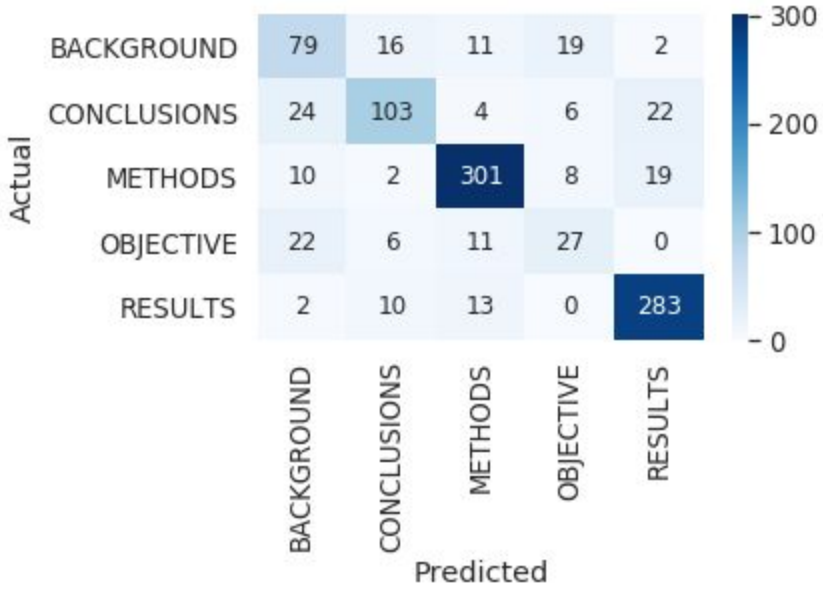


Figure 6: Confusion Matrix for Coded Results

	precision	recall	f1-score	support
BACKGROUND	0.58	0.62	0.60	127
CONCLUSIONS	0.75	0.65	0.70	159
METHODS	0.89	0.89	0.89	340
OBJECTIVE	0.45	0.41	0.43	66
RESULTS	0.87	0.92	0.89	308
accuracy			0.79	1000
macro avg	0.71	0.70	0.70	1000
weighted avg	0.79	0.79	0.79	1000

Table 4: Coded Precision, Recall and F1-Score for each Class

The in-depth analysis of the metrics for each solution confirm that the scikit-learn and hand coded solutions both perform the exact same in the classification task. Weka performs slightly better however is still comparable to the other solutions. The confusion matrix for the solution looks healthy, with minimal false readings. However, the class breakdown of precision and recall reveal that the classes with minimal support and a low number of training examples have a lower precision and recall compared to the other classes. This is most likely due to the imbalanced class training data and would explain why the MCC is lower compared to the accuracy and F-measure for each model (see Figure 2).

Summary

Overall, the three solutions implemented performed very well, managing to achieve close to 80% accuracy compared to the ~20% accuracy that would be achieved by randomly guessing the 5 classes. This is somewhat surprising considering that the SciBERT encoder was not trained for this task and was instead trained for general NLP tasks.

This task was very challenging as it was hard to find a good encoding method and then to apply the SciBERT algorithm. The encoding aspect took 2 hours and 20 minutes to run for all 88797 sentences in the training data, of which only the first 3500 sentences were used to reduce the training time for the solutions. However, despite only using a small subset of the available data, the algorithm still performed very well.

It was then tested how modifying the size of the input data would influence the results. The training data was increased by 10x (to 25000 training data samples and 10000 testing data samples), achieving an accuracy of 82%. However the same misclassification bias for smaller classes still existed that was prevalent in the initial data.

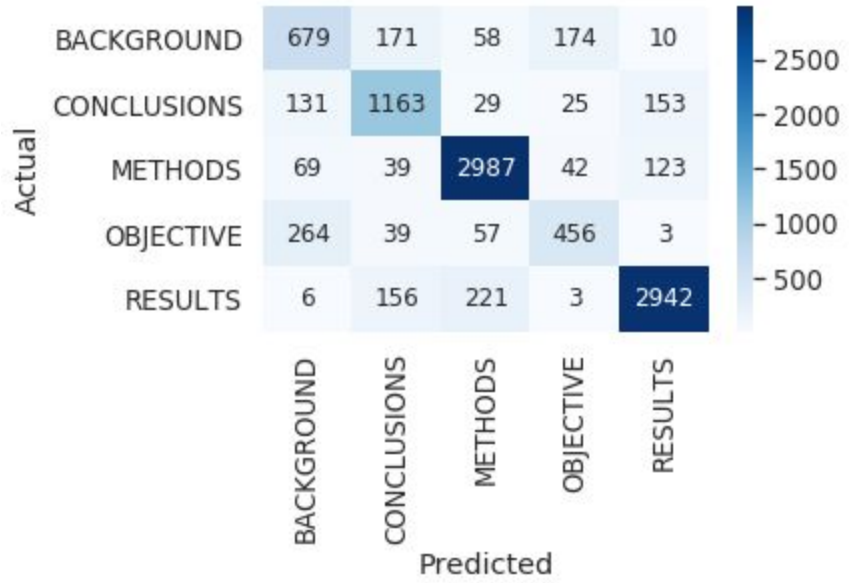


Figure 7: Confusion matrix for kNN with 25000 Training Sample Points and 10000 Test Sentences

It was experimented if balancing the class data would help improve the precision and recall with both the background and objective classes. However as is shown in table 4 below this only helped improve those two classes marginally (see Table 5). However, it is interesting to note that the accuracy of the model at 0.72 now matches what the MCC was calculated as in Table 1.

	precision	recall	f1-score	support
BACKGROUND	0.58	0.57	0.58	200
CONCLUSIONS	0.70	0.71	0.71	200
METHODS	0.86	0.80	0.83	200
OBJECTIVE	0.63	0.70	0.67	200
RESULTS	0.85	0.81	0.83	200
accuracy			0.72	1000
macro avg	0.72	0.72	0.72	1000
weighted avg	0.72	0.72	0.72	1000

Table 5: kNN Precision, Recall and F1-Score for Balanced Class Data

Finally, other models were investigated to find out if there was a better performing algorithm on this data compared to kNN. In scikit, multiple algorithms were tested experimentally and the best performing one was determined to be an SVC with an initial accuracy of 83%. By fine tuning the parameters, the accuracy was further improved to 84.2%. This was the highest accuracy achieved and is very close to the best reported accuracy reported by the SciBERT team for this data at 86.81% (Beltagy et al., 2019). This is an impressive result for a model which only uses 2.8% of the available training data compared to Beltagy (2019) results which uses 100% of the training data. By increasing SVM's training data to include 28% of the training data, the accuracy was further improved to 86.6% with an MCC of 0.82, the best metrics achieved and almost identical to those in the original paper.

It should be noted that none of these models implement fine tuning, a method of further training the SciBERT embeddings on specific training data to improve the results. It is hypothesized that with more computation, by including all available data in the dataset and fine tuning the SciBERT model, over 90% accuracy is feasible. However, with the time and computational constraints, these results are very impressive with an 86.6% accuracy being high enough to use in real-world applications.

References

Beltagy, I., Lo, K. and Cohan, A., 2019. SciBERT: A Pretrained Language Model for Scientific Text. *arXiv e-prints*.

Dernoncourt, F. and Lee, J., 2017. PubMed 200k RCT: a Dataset for Sequential Sentence Classification in Medical Abstracts. *arXiv e-prints*.

Grande, A., Hoffmann, T. and Glasziou, P., 2014. Searching for randomized controlled trials and systematic reviews on exercise. A descriptive study. *Sao Paulo Medical Journal*, 133(2), pp.109-114.

Navlani, A., 2018. *KNN Classification Using Scikit-Learn*. [online] DataCamp Community. Available at: <https://www.datacamp.com/community/tutorials/k-nearest-neighbor-classification-scikit-learn> [Accessed 28 August 2020].

Contribution

As Matthew Ellis has dropped the course before work on the assignment had begun, I have completed the assignment on my own. This was approved via email correspondence with the course convenor.