TEXT CONVOLUTIONAL NEURAL NETWORK FOR ARTIFICIALLY GENERATED TEXT DETECTION

Calvin Leighton
University of Technology Sydney
Sydney, Australia
ORCID: 0009-0005-1004-5442

***Abstract*— Artificially generated text is a prevalent issue in todays academic world, where the syllabus is not properly prepared to handle students using generative models in order to complete assignments for them. Here, I demonstrate the application of a text convolutional neural network to identify patterns throughout the "AI vs Human Text" dataset which encompasses 500,000 human and artificially generated essays. The network achieves a 97% detection rate, outperforming state of the art models like SciBERT and baseline models such as support vector machines. It's conclusive that with more refined and regularly updated data for academic texts, a text convolutional neural network is a viable strategy for academics to identify what is, and what isn't artificially generated. All the code for this project can be found here: https://github.com/nekovin/AI_Text_Detection_Using_TCNN. The data is publicly available here: https://www.kaggle.com/datasets/shanegerami/ai-vs-human-text/data.**

Keywords—Generated Text Detection, Text-Convolutional-Neural-Network

I. Introduction

ChatGPT and other large-language models have become extremely prominent in recent years. Generative models such as these are hard to monitor in schools, since they go against ethical principles of academic frameworks. Australia provides the "Australian Framework for Generative Artificial Intelligence in Schools" [1] which essentially covers what students should do, however, as students do, this can be easily disregarded. The need for reliable methods to detect artificially generated content is necessary as nobody can solely rely on the academic integrity of students.

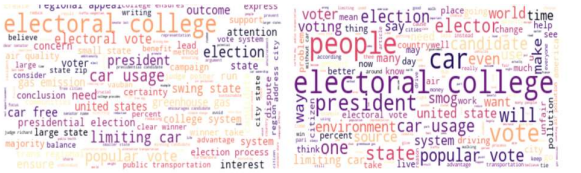
Throughout a brief literature review, I found that most papers on the detection of AI text was primarily focused on the current tools performance; with a distinct lack of research on how the actual machine learning model is performing. Therefore, this project is to explore the performance of a text convolutional neural network (TCNN) and training it to understand the nuances of how a human writes, and how an artificially generated model writes.

II. Challenges with Detecting Artificially Generated Text

The task of detecting artificially generated text comes with various challenges.

A. Nuance

Figure 1. Depicts two word-clouds of human and artificially generated text- immediately, it is difficult to discern what is human generated and artificially generated. When looking at the frequencies of the words, there is no indication that one is immediately recognisable as human or artificially generated. This is a challenge that text classification models must overcome by not looking at the isolated words, but instead the patterns in which they are used. This is a very unrealistic task for a human to solve, however, the TCNN is made for just this.



(a) Word-cloud of human generated text (b) Word-cloud of artificially generated text

Figure 1: **Word-cloud of human and AI generated text**. Word-cloud of corpus. Human (a) and AI (b) most commonly used words.

B. Variance among generative models

One of the least obvious but also most realistically challenging task to address for future research, is the fact that there are so many different text-generative models (Chat-GPT, Claude, Llamma-3 to name a few), each with their own distinctive writing patterns as they have been trained on similar but still different corpus. It is unclear how the artificial data was collected in the "AI vs Human Text" dataset, this is something for future research.

C. Computational Complexity

A challenge which becomes immediately apparent once training begins, is that the computational complexity scales proportionally to the dataset size.

$$O(L \cdot d) \tag{1}$$

In (1) L represents the length of the input and d representing the dimension of the embedding. This happens because the initial embedding layer maps the sequence of length L to a $L \cdot d$ matrix. Implicitly, the kernel function then must convolve over L – C + 1 positions (C being a constant), each requiring $O(C \cdot d)$ operations; since computational complexity disregards constant values, the entire operation is $O(L \cdot d)$.

III. Text Convolutional Neural Networks

Convolutional neural networks (CNNs) have been traditionally used for image classification, segmentation and detection tasks. First proposed in "Convolutional Neural Networks for Sentence Classification" [2], the principles of an image-CNN can be expanded onto a text-CNN.

A. Capabilities of Capturing Patterns

A traditional CNN takes in a matrix representation of an image where it continues to throughout the network in order to make a prediction.

The most defining feature of CNNs is the convolution layers. These are layers which use a kernel/filter which parse

over a 2D matrix representation of the data and captures the most important details (features) by using pooling. These captured features are then passed

through an activation function (commonly ReLU) in order to reduce linearity and finally to a simple feed-forward network in order to learn the patterns from the features of the input. These patterns are then passed through to an task-appropriate activation function (in this case, sigmoid) where a final class prediction is then made.

This is only half the process though, the other half involves back propagation in order to learn which nodes in the network need to be adjusted in order to allow class defining features to be passed to the correct output node.

The only difference between a traditional CNN and TCNN in this case, is the initial layer, which in the case of a TCNN, is an embedding layer. Therefore, arguably the most important step for a TCNN is the proper preparation of the data in order to mimic traditional, CNNs for classification.

B. Strong Support from Python Libraries

Another reason to use a TCNN, is that there is much support for the implementation using well established Python libraries such as Pytorch, SciPY, Numpy and Pandas. These libraries allow for easy integration of neural networks and the manipulation of large data corpuses. Trying to implement more niche classification models may not have the same level of technical support that a TCNN does.

C. Adaptability to Scale

As mentioned in the challenges, the computational complexity rises as the scale also increases. This is a challenge, however, it is also a positive, as this means that the model can adapt to different sizes, making implementation easier and safe (but still time-consuming).

D. Weaknesses

Now that the TCNN has been justified, it is important to acknowledge the apparent weaknesses of this approach.

a) Computationally heavy

The double-edged positive is also a weakness. The more data that the model gets, the better it can perform, however the more data, means more computational lifting.

IV. Corpus Details

A. Corpus

AI-Human Text (AHT): AHT is a dataset of 487,235 pieces of text which are classified as either human generated or artificially generated (see Figure 2). 181,438 is artificially generated and 305,797 is human generated. Due to hardware constraints, I limited my research to a sample size of 10,000 with a vocabulary size is 32,689 as this is computationally viable while being representative of the data.

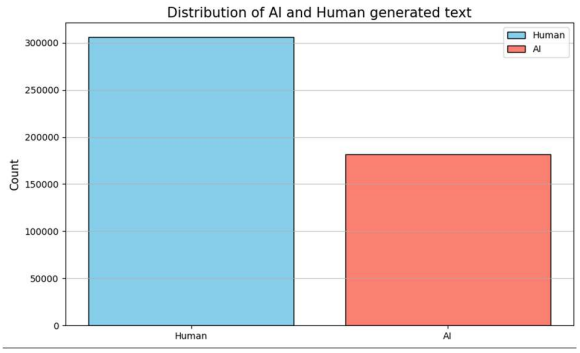


Figure 2: **Distribution of corpus**. There is an imbalance of data, however, I elected not to balance them out because I believe this is more similar to a real-world distribution of human to AI text. That is, more human written essays are submitted than AI text essays in real-life.

V. Implementation Details

A. Corpus Preprocessing

The AHT corpus was pre-processed in a systematic way, in order to avoid any bias or misrepresentation of the data.

1) Cleaning

All the text was initial cleaned. This entailed lowercasing all words and removing stop-words including punctuation marks and full stops. See Appendix C, Figure 8, (b) for some of this code implementation.

2) Tokenisation

Each sequence of text was tokenised. This was to break the work up for processing into manageable 'bites' for the computer to handle. See Appendix C, Figure 8, (c) for this code implementation.

3) Vocabulary Building and Word Indexing

The vocabulary was built using the data. This is essentially allocating a value to each unique word. These values were then used to replace the tokenised word with their unique value from the vocabulary.

4) Padding

The final pre-processing step was padding. TCNNs require all the input data to be of the same length, therefore, the shorter sequences had to be padded out with the following formula (2):

$$Padding\ count(x) = Max(L) - L_x \qquad (2)$$

Where x is the current sequence, $Max(L)$ is the sequence from the data which has the most text, and L_x being the current sequence length. This value indicates how many zeros will need to be added to the sequence in order to ensure that all sequences are of the same length.

Once these steps were completed, they were packed into batches of 32, ready for training.

B. Algorithmic Details

The implementation of the TCNN was trivial when compared to the corpus pre-processing steps. This was a simple convolutional neural network, with the inclusion of an initial embedding layer. See Appendix C, Figure 8 (e) for the code.

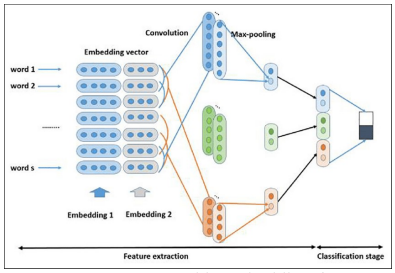


Figure 3: **Embedding layer**. This embedding framework was depicted by [5]. This takes in the tokenised sequence and transforms it into a dense vector. This is adjusted alongside the feed forward network during back-propagation.

1) Embedding layer

Figure 3 depicts the embedding layer. This layer takes in the tokenized sequences and transforms them into dense vectors. This layer is learned throughout training.

2) Convolutional layers and pooling

The convolutional layers are the layers of the network which learn the patterns. It does this by parsing through the input data (represented as a 2D matrix) by using a filter/kernel. This kernel sweeps across the matrix and applies pooling. This is depicted in Figure 4 (a).

3) ReLU activation

The ReLU activation function introduces non-linearity in the network. It's easier to explain what happens to the network when this is omitted, than what it actually does:

$$ReLU(x) = \max(0, x) \tag{3}$$

Without this function, the network would be completely linear, essentially reducing itself into a single linear regression model, nullifying any benefits that the convolution network brings by only being able to learn linear functions. Therefore, this is what is allowing the convolutional layer to learn the patterns from the input data.

4) Flattening layer

This converts 2D text into 1D text which is then passed to the linear layer. This is depicted in Figure 4 (b).

5) Linear layer

This layer takes in the flattened features from the convolution layers and learns the patterns. This is depicted in Figure 4 (c).

6) Sigmoid activation

This commonly used activation function takes the output value from the linear layer and transforms it into a squished value between zero and one.

$$Sigmoid = S(x) = \frac{1}{1+e^{-x}} \tag{4}$$

Where x is the input vector. This is required for binary classification tasks which is exactly what this research is aiming for.

C. Experimental Settings

1) Hardware

A NVIDIA GTX 2080 was used for training with 32GB of memory. Although on the lower end of GPUs for machine learning, it gets the job done.

2) Hyperparameters

a) Architecture

There are different kinds of pooling, however, I elected to use max-pooling. After pooling, an activation function is applied. The number of convolution layers can be varied from model to model. The settled-on number was three, satisfying a middle ground between performance and computational efficiency. Due to the hardware constraints of this research, the size of the model had to be heavily considered. Two linear layers were used.

b) Learning rate

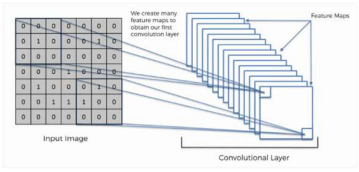
The learning rate is that which the weights are adjusted throughout the network during back propagation. This was set to 0.0001 and iteratively adjusted by the optimization function.

c) Optimisation function

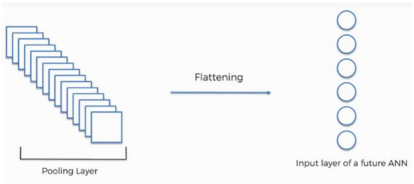
Adaptive Moment Estimation (Adam) was used to incrementally adjusting the learning rates for individual parameters in the neural network [3]. This has shown to improve convergence instead of using a global learning rate for all parameters.

d) Dropout rate

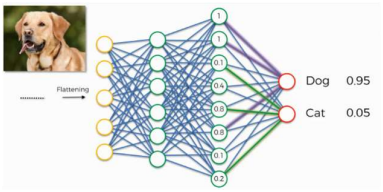
A dropout rate of 0.5 was used, that is, half the nodes throughout the training were dropped. This is in order to increase generalizability by not relying on the most obvious



(a) Pooling layer



(b) Flattening layer



(c) Linear layer / feed-forward network

Figure 4. **Traditional Convolutional Neural Network Process**. In an image classification task, the image (e.g. a dog) is passed in as a 1-3 channel, 2D matrix representation. This matrix is initially passed into a convolution stage which is comprised of convolution and pooling layers and activation functions. After the final step in the convolution stage, the output is flattened into a 1-dimensional vector, which is now ready to be passed into a feed-forward network. The feed-forward network takes the cumulation of the previous steps in order to learn the patterns which details what class it should be sorted into. After each pass through the entire network, back-propagation occurs, in which the network reverses itself and optimises the node weights in order to increase the probabilistic confidence in any one prediction. This is evaluated by how much the loss/optimisation-function is minimised through a gradient descent. Images sourced by [6].

learned connections in the network and is instead forced to find new paths.

Table I. Comparative Study Table

Model	Precision		Recall		F1-Score		Acc
M/W	M	W	M	W	M	W	
TCNN	0.97	0.97	0.97	0.97	0.97	0.97	0.97
SciBERT	**0.98**	**0.99**	**0.98**	**0.99**	**0.99**	**0.99**	**0.99**
SVC	0.94	0.92	0.89	0.91	0.9	0.91	0.91

In order to evaluate how well the TCNN did, the state-of-the-art, SciBERT [4] classification model and baseline support-vector-classifier (SVC) were employed. The evaluation metrics were precision, recall, F1-score and accuracy; each had a support of 1500 and were written as macro (M) and weighted (W) averages. Boldened figures represent the best score.

VI. Experimental Results

A. Results

TCNN was heavily influenced by the number of available datapoints throughout training. The complete dataset could not be trained on due to the hardware constraints, however, it was trained on 10,000 datapoints which is still a sufficient dataset for hypothesis testing. This required more data to get substantial results, however, provided marginally diminished returns after using the complete dataset. The accuracy after training and validating on the 10,000 datapoints of AHT was 97.13%.

B. Comparitive Study

Shown in Table I, the TCNN has very competitive scores when compared to the state of the art SciBERT model and managed to perform better than the SVC. Considering the fact that the SciBERT model has far more parameters than the TCNN, the TCNN was only a single percent off of performing just as well. This suggests that for detecting artificially generated text, the TCNN is a viable approach for hardware limitations. See Appendix E and F for the implementation of these alternate models and their outputs.

C. Ablation Study

Two ablation studies were conducted; alteration of the dataset size and random word ablations. The results of these are explored in the discussion section.

a) Dataset Size

The models were trained on different dataset sizes. It was clear that as the dataset size increased to a sufficient count, so too did the accuracy of the models. See Appendix A for results.

b) Random word ablation

There was actually a slight uptick when removing 25% of the words. See Appendix B for results.

VII. Discussion

In the comparative study, the TCNN performed almost as well as the state-of-the-art, with far less parameters. Considering this, there is a viable approach for academic landscapes: A potential application is to train a TCNN on each students writing (tests, essays, etc) in order to have a personalised model which ensures academic integrity. This could be possible, solely due to the low hardware requirements for training such a model.

As the performance increased proportionally with dataset sizes, an application could possibly be sitting regular, non-assessed, in-class essays to feed more data into the models.

Finally, removing words from the dataset only marginally reduced the accuracy of the models. This suggests that the model is not learning the specific words but the certain patterns and dialects that the words are being used with. In other words, it can safely predict artificial text and human text based on the structure of the sentences instead of isolated word inclusion. Using this, perhaps these ablated datasets could be refed into the model for increased dataset sizes.

Conclusion

In conclusion, TCNN has shown to have better learning capabilities than the baseline, and very competitive results compared to state-of-the-art models. Future research may wish to look into contrastive learning, as this has shown to have great generalisability in image discrimination tasks, perhaps the same is for text. The real-world implications of this research is improved academic integrity for students and academics alike.

VIII. Acknowledgment

Thank you to Dr Wei Liu and Dr Rafiqul Islam for helping me facilitate my learning of natural language processing throughout the first semester at the University of Technology Sydney, 2024.

References

[1] Australian Government, "Australian Framework for Generative Artificial Intelligence (AI) in Schools," 2024. [Online]. Available: https://www.education.gov.au/schooling/resources/australian-framework-generative-artificial-intelligence-ai-schools

[2] Y. Kim, "Convolutional Neural Networks for Sentence Classification," 2014. [Online]. Available: https://doi.org/10.48550/arXiv.1408.5882

[3] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," 2014. [Online]. Available: https://doi.org/10.48550/arXiv.1412.6980

[4] I. Beltagy, K. Lo, and A. Cohan, "SciBERT: A Pretrained Language Model for Scientific Text," in Proc. 2019 Conf. Empirical Methods in Natural Language Processing and the 9th Int. Joint Conf. on Natural Language Processing (EMNLP-IJCNLP), Association for Computational Linguistics, 2019, pp. 3615-3620. [Online]. Available: https://doi.org/10.18653/v1/D19-1371

[5] V. Q. Nguyen, N. Anh, and H.-J. Yang, "Real-time event detection using recurrent neural network in social sensors," Int. J. Distrib. Sensor Networks, vol. 15, no. 6, 2019. doi: 10.1177/1550147719856492

[6] SuperDataScience, "The Ultimate Guide to Convolutional Neural Networks (CNN)," SuperDataScience.com, 2021. [Online]. Available: https://www.superdatascience.com/blogs/the-ultimate-guide-to-convolutional-neural-networks-cnn

IX. APPENDIXES

Appendix A: Dataset Size Ablation Figures

Table II. Dataset Size Ablation Results			
Model	Accuracy		
Datapoints	100	1000	10,000
TCNN	46.67%	**97.27%**	96.40%
SciBERT	**95%**	95%	**99.3%**
SVC	73%	93%	90%

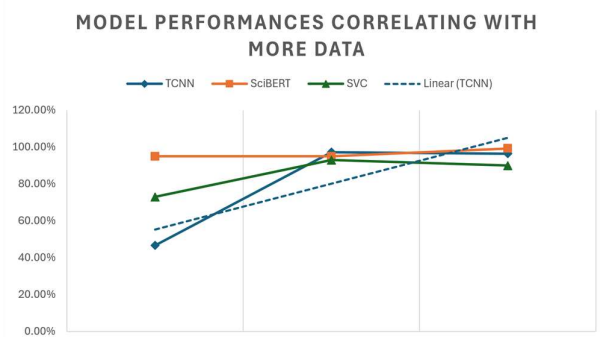


Figure 5: Regression chart for the performances for the different models on increasing dataset sizes.

Appendix B: Word Ablation Study Results

Table III. Word Ablation Results	
Words Ablated	Accuracy
0%	96.4%
25%	96.6%
50%	94.2%

Appendix C: Loss and test visualisation

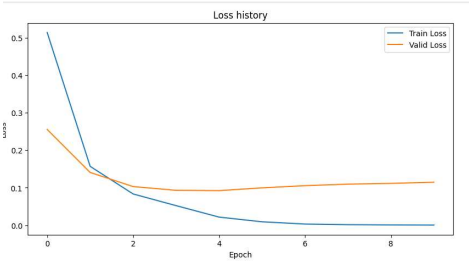


Figure 6: Train and validation loss visualisation from the latest training run.

```
print(classification_report(y_val, y_pred))

100%|██████████| 47/47 [00:00<00:00, 351.82it/s]
Test Loss: 0.1149, Accuracy: 0.9713
Predictions: [1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0
1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0,
0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1,
0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0,
0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0,
1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0,
1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0,
1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0,
0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1,
1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0,
0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1,
1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0,
0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0,
1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0,
0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1,
0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0,
0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0,
1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0,
0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1,
0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0,
0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0,
1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0,
0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0,
0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0,
1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0,
0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1,
1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1,
0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0,
0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
              precision    recall  f1-score   support

           0       0.98      0.98      0.98       936
           1       0.96      0.96      0.96       564

    accuracy                           0.97      1500
   macro avg       0.97      0.97      0.97      1500
weighted avg       0.97      0.97      0.97      1500
```

Figure 7: Predictions for some batches. This is demonstrating that the code is actually making diverse predictions, accurately and there isn't some sort of one sides estimation.

Appendix C: Workflow

```
df_ai = df[df['generated'] == 1]
df_human = df[df['generated'] == 0]

df = pd.concat([df_ai, df_human])

df['generated'] = df['generated'].apply(lambda x: int(x))
df['text'] = df['text'].apply(lambda x: str(x).lower())

print(df)
```

```
                                                     text  generated
704     this essay will analyze, discuss and prove one...          1
740     i strongly believe that the electoral college ...          1
1262    limiting car use causes pollution, increases c...          1
1378    car-free cities have become a subject of incre...          1
1379    car free cities  car-free cities, a concept ga...          1
...                                                   ...        ...
487229  in "the challenge of exploring venus" the auth...          0
487230  tie face on mars is really just a big misunder...          0
487231  the whole purpose of democracy is to create a ...          0
487233  i dfn't agree with this decision because a lft...          0
487234  richard non, jimmy carter, and bob dole and ot...          0

[487235 rows x 2 columns]
```

(a) Data frame prep.

```
class PreprocessedDataFrame(pd.DataFrame):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self._remove_tags()
        self._remove_punctuation()
        self._remove_stopwords()

    def _remove_tags(self):
        tags = ['\n', '\'']
        for tag in tags:
            self['text'] = self['text'].replace(tag, '')

    def _remove_punctuation(self):
        self['text'] = self['text'].apply(lambda text: ''.join([x for x in text if x not in string.punctuation]))

    def _remove_stopwords(self):
        stop_words = set(stopwords.words('english'))
        self['text'] = self['text'].apply(lambda text: ' '.join([word for word in nltk.word_tokenize(text) if word.lower()

# Usage:
df = PreprocessedDataFrame(df.sample(10000)) # sampling 10000, change this if you would like to use more/less
```

```
df.head()
```

	text	generated
34639	facial action coding system facs method analyz...	1
177014	think important de churchill important place m...	0
110071	according article unmasking faze mars twentyfi...	0
34784	venus named roman goddess beauty second planet...	1
228937	online schooling failing students examining hu...	0

(b) Cleaning.

```
class SimpleTokenizer:
    def __init__(self):
        self.word_index = {'[UNK]': 0}
        self.index_word = {0: '[UNK]'}
        self.vocab_size = 1

    def fit_on_texts(self, texts):
        word_count = Counter(word for text in texts for word in text.split())
        for word, _ in word_count.items():
            if word not in self.word_index:
                self.word_index[word] = self.vocab_size
                self.index_word[self.vocab_size] = word
                self.vocab_size += 1

    def texts_to_sequences(self, texts):
        return [[self.word_index.get(word, 0) for word in text.split()] if text.split() else [0] for text in texts]

def prepare_data(sequences, word_features, max_length):
    padded_sequences = np.zeros((len(sequences), max_length), dtype=int)
    for i, seq in enumerate(sequences):
        end = min(len(seq), max_length)
        padded_sequences[i, :end] = seq[:end]
    return (
        torch.tensor(padded_sequences, dtype=torch.long),
        torch.tensor(word_features, dtype=torch.float32),
        torch.tensor([min(len(seq), max_length) for seq in sequences], dtype=torch.long)
    )

def tokenize_text(text):
    tokens = word_tokenize(text.lower())
    stop_words = set(stopwords.words('english'))
    return [token for token in tokens if token.isalpha() and token not in stop_words]
```

(c) Tokeniser.

```
Sample padded sequences: tensor([[  1,   2,   3, ...,   0,   0,   0],
        [174, 175, 174, ...,   0,   0,   0],
        [264, 265, 266, ...,   0,   0,   0],
        [548, 549, 550, ...,   0,   0,   0],
        [117, 646, 258, ...,   0,   0,   0]])
```

(d) A sample of some padded data.

```
class TextCNN(nn.Module):
    def __init__(self, vocab_size, embed_size, num_classes=1, max_seq_length=MAX_SEQUENCE_LENGTH):
        super(TextCNN, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.conv1 = nn.Conv1d(embed_size, 128, kernel_size=4, padding='same')
        self.pool1 = nn.MaxPool1d(kernel_size=2)
        self.conv2 = nn.Conv1d(128, 64, kernel_size=4, padding='same')
        self.pool2 = nn.MaxPool1d(kernel_size=2)
        self.conv3 = nn.Conv1d(64, 32, kernel_size=4, padding='same')
        self.pool3 = nn.MaxPool1d(kernel_size=2)
        self.flatten = nn.flatten()

        output_length = max_seq_length // (2**3)
        flattened_output_size = 32 * output_length

        self.fc1 = nn.Linear(flattened_output_size, 256)
        self.fc2 = nn.Linear(256, num_classes)
        self.output_act = nn.Sigmoid()

    def forward(self, x):
        x = self.embedding(x)
        x = x.permute(0, 2, 1)
        x = F.relu(self.conv1(x))
        x = self.pool1(x)
        x = F.relu(self.conv2(x))
        x = self.pool2(x)
        x = F.relu(self.conv3(x))
        x = self.pool3(x)
        x = self.flatten(x)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        if num_classes == 1:
            x = self.output_act(x)
        return x
```

(e) TextCNN code implementation.

Figure 8: The above depict the main workflow of this project.

Appendix E: SVC implementation

SVC

```
pipeline = Pipeline([
    ('count_vectorizer', CountVectorizer()),  # Step 1: CountVectorizer
    ('tfidf_transformer', TfidfTransformer()),  # Step 2: TF-IDF Transformer
    ('naive_bayes', MultinomialNB())])

pipeline.fit(X_train, y_train)

y_pred= pipeline.predict(X_test)


print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

           0       0.83      1.00      0.91       896
           1       0.99      0.70      0.82       604

    accuracy                           0.88      1500
   macro avg       0.91      0.85      0.86      1500
weighted avg       0.90      0.88      0.87      1500
```

Figure 9: The pipeline implementation and classification report of the SVC.

Appendix F: SciBERT model implementation

```
BertForSequenceClassification(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(31090, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0-11): 12 x BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
    )
  )
  (dropout): Dropout(p=0.1, inplace=False)
  (classifier): Linear(in_features=768, out_features=2, bias=True)
)
```

(a) SciBERT model display.

```
              precision    recall  f1-score   support

           0       0.93      0.93      0.93        15
           1       0.80      0.80      0.80         5

    accuracy                           0.90        20
   macro avg       0.87      0.87      0.87        20
weighted avg       0.90      0.90      0.90        20
```

(b) Classification report from SciBERT.

Figure 10: (a) is the printed model of the SciBERT model. (b) depicts the outputs of the classification report from it.