BERT, Word2Vec and the Microsoft Sentence Completion Challenge: A Comparison in Performance.

Abstract

With natural language processing modelling advancing at a high pace, it is essential to have standards to which both current and future models can be compared against. The Microsoft Sentence Completion Challenge provides exactly that. With the provided questions and answers, this paper investigates the usefulness of both the Word2Vec and the BERT models in attempting this challenge. We hypothesize the deeper BERT model, with its ability to encode both left and right contextual information, will significantly outperform the Word2Vec model. This is shown to be true, with the BERT model's performance measuring ~20% higher.

1. Introduction

1.1 Problem Statement

This paper aims to design, refine, and implement working codebase for two different approaches to The Microsoft Research Sentence Completion Challenge. The challenge in question required systems to be able to predict the most likely word missing from a sentence. There are five possible choices of missing words for each sentence, with 1040 sentences provided. Some possible options provided include:

- Tri-gram models
- Word Similarity methods
- A combination of N-gram Methods and Word Similarity methods
- Neural Language models

It should be stated that although performance was not the main aim of this task, increasing it was still vital to measuring how successful a process, and any refinements made to said process, may be.

The first approach chosen for this paper makes use of the Word2Vec word similarity method, using GoogleNews vectors. The second method makes use of the Bert Neural Language model. Firstly, for both approaches a basic working model was implemented and evaluated. After this, modifications were made to improve performance, evaluated, and compared to the previous measurements.

It is prudent then, to first cover relevant background material associated with each approach. This will allow us to understand the strengths and weaknesses of each method, and to gain an idea of how to improve both.

1.2 Word2Vec and the Pretrained GoogleNews Model

Word2Vec is a neural network model which takes as input, a text corpus and outputs sets of vectors. These vectors are a representation of each word in the text. It is trained on an even larger text corpus, with some pretrained models readily available, such as the GoogleNews300 model. Being able to represent words as vectors is effective for word similarity tasks, as it has been shown similar words are closer to each other, on average, in the vector-space [1]. To show an example of this, let us take the vector representations of 'king', 'man' and 'woman' as v(king), v(man) and v(woman). Then v(king) – v(man) + v(woman) equates to a vector close to v(queen) [1]. This is seen in numerous other examples [2]. Word2Vec models are trained using either a Continuous Bag of Words (CBOW) architecture or Skip-Gram architecture, both proposed in the seminal paper [3]. With the CBOW model, the order of words does not affect the predicted word, and the context representation is modelled as a continuous distribution [3]. For the Skip-gram model, a word is taken as input, and using a log linear classifier, the surrounding words within a range either side of the input are predicted [3]. After training, or by using a pretrained model, the similarity between two words can be quickly

and efficiently calculated by using an in-built Word2Vec method. This method uses cosine similarity as the measure of similarity between two words.

1.3 Language Model and their Neural Counterparts

Language models compute the probability distribution of a text corpus. This distribution represents the frequency with which words in the corpus are seen [4]. A basic language model such as an N-gram, can predict the next word in a sequence by computing the conditional probability of a word, given the words before it. Unfortunately, due to sparsity of data, this method can often lead to poor estimates [5]. Neural Language models (NLMs) can circumvent this issue, using word embeddings in conjunction with neural networks [5]. NLMs have been shown to outperform N-grams significantly [6]. A disadvantage of NLMs is that they do not deal well with the morphological information that words can convey [5]. This is a problem when dealing with morphologically rich languages, as rare words specifically, can be poorly estimated. Early investigations show Recurrent Neural Networks (RNNs) as the more promising choice of model architecture [5][7], however, more recent studies prove that a Transformer network architecture [8] to outperform even RNNs.

1.4 Transformer Network Architecture and the BERT Model

While RNNs use sequence alignment to learn word embeddings and make predictions [8][9], Transformers focus solely on attention mechanisms, mainly self-attention, to learn dependencies between input and output [8]. Attention put simply, is the focus on certain details at the expense of others. Self-attention allows a representation of a sequence to be created, relating to the varying positions of the sequence [8]. The BERT (Bidirectional Encoder Representations from Transformers) model, is a pretrained deep transformer network, which allows inexpensive finetuning to specific datasets. Whereas previously pretrained transformer models have been unidirectional, BERTs bidirectionality allows right and left context to be fused together when creating its representations [10]. This is a result of the architecture's pretraining objective, the 'masked language model'. The BERT model has improved upon many benchmark natural language processing test scores, including the absolute accuracy of the Glue Test, by 7.7% [10].

1.5 Report Outline

This paper will be tackling each approach's methodology and their results separately, with the comparison of the two dealt with afterwards. Part 2 will deal with the Word2Vec approach and any modifications made with associated justifications. Part 3 is dedicated to the BERT model method. In part 4, we will draw comparisons between the two. Part 5 will contain the conclusion, with part 6 describing further areas of work for both approaches.

2 Word2Vec

2.1 Methodology

The design and implementation of the code base to make use of the Word2Vec model is simple. Firstly, we loaded the KeyedVectors module with the 'GoogleNews vectors negative 300' data.

```
filename = r'C:\Users\t_p_c\AI\Term
2\ANLP\Labs\GoogleNews-vectors-negative300.bin.gz'
mymodel = KeyedVectors.load_word2vec_format(filename,binary=True)
```

Figure 1: The code used to load the Word2Vec module with the GoogleNews vectors.

Once this was competed, functions to pass the data through Word2Vec were writen. It should be said that if Word2Vec is given a word which was not present in its training, it results in a Key error. This is the main issue faced when trying to use the model. It seemed shrewd to inspect the data before writing code to deal with it, if only to see which type of words would bring up this error.

As can be seen in figure 2, the most common tokens present in the question data, but not in the model's vocabulary, are punctuation and stopwords. It also shows they appear often. As a whole, both punctuation and stopwords do not contribute to context. There are 56 hyphenated words in the question sentences and 81 in the answers which are not in the model's vocabulary.

In the initial codebase design, words which are not present in the model's vocabulary are ignored.

	Words not in Vocab	Count
0	,	1160
1	.	1041
2	and	673
3	of	542
4	a	522
...	...	...
87	practised	1
88	trap-door	1
89	broad-shouldered	1
90	sick-room	1
91	programme	1

Figure 2: The table shows the 5 most common/least common tokens present in the questions but not the Word2Vec model, and their count.

Hence, to make full use of Word2Vec, the code base must deal with:

- Checking whether or not the question words are present in the model's vocab
- Checking whether or not the answer words are present in the model's vocab
- Computing the similarity between questions and possible answers
- Evaluating the accuracy of the model

To check if the question and answer sentences contained any words not in the model's vocab, the function checkQuestion() and checkAnswer() were written. As an answer is a single word, the checkAnswer() function simply checks whether the word is in the vocabulary or not. It then returns 'True' if it is and 'False' if it isn't.

```
def checkAnswer(answer):
    if answer in mymodel.vocab:
        return True
    else:
        return False
```

Figure 3: The checkAnswer() function. An 'if/else' statement is used to indicate whether the argument passed through it is in the Word2Vec vocabulary or not.

```
def checkQuestion(question):
    # List containing punctuation to remove from questions
    punc = ['.', ',', '?', '!', "'", '_____']
    # Call NLTK method to tokenise sentence
    tokenised = tokenize(question)
    # List to add words which are present in the vocab
    sentence = []
    # For loop to check each word in order
    for word in tokenised:
        # If the word is punctuation, ignore it
        if word in punc:
            continue
        # If the word in the vocab add it to sentence
        elif word in mymodel.vocab:
            sentence.append(word)

    return sentence
```

Figure 4: The checkQuestion() function. Tokenisation was performed on the argument passed through it.

The checkQuestion() function required the use of the NLTK libraries tokeniser. This is because the questions had to be checked word by word. If a token is not in the model's vocabulary, it is removed from the sentence. This new sentence is returned.

To compute the similarity of a question and its corresponding possible answers, the getSim() function was designed. The function calls on the Word2Vec similarity() method, and sums the similarity between each word in the question and a possible answer. It then calculates the average of this similarity which is used to score the answer. The rationale behind this, is that the answer which is more similar to a greater number of words will give a better average. The getSim() function is used in the maxSim() function, which loops through the 5 possible answers to choose the highest scoring answer for the associated question. The testAllAQuestions() function calls maxSim() for all questions.

```
def getSim(question, answer):
    # Set similarity score to be
    sim_score = 0
    # Loop through question to get similarity with answer
    for word in question:
        # Increment similarity score with Word2Vec similarity function
        sim_score += mymodel.similarity(word, answer)
    # Get average score for question and answer
    average_score = sim_score / len(quiestion)
    return average_score
```

Figure 5: The getSim() function call on the Word2Vec similarity() method to compute similarity of question and corresponding answers.

2.2 Initial Evaluation

To evaluate the accuracy of our Word2Vec approach, the evaluate() function simply compares the answers our model obtains with the correct ones. It then returns the percentage of those which are the same. Our initial design scores 36.06 2.d.p. This score will be the baseline for future modifications to be compared against.

2.3 Modifications

In order to see what improvements can be made to our approach, we must look through the data we chose to ignore in our first design. Our code base will still ignore punctuation and stopwords. To begin with, there are 14 words from the questions which appear in the model's vocabulary with their first letter capitalised. Replacing these words for their titled counterparts, and therefore giving them a vector representaion actually lowers the accuracy of the model by 2% from ~36% to ~34%. This is presumably because the titled counterparts are more than likely headlines, given that our training set is made up of news articles. Therefore, they probably do not occur too often in the training data and giving them more weight in our approach is skewing the similarity calculations. As a result, we will continue to ignore these 14 words.

Our next step is to deal with the 137 compositional words which are hyphentated. One method for this is to separate the words into their individual constituents and simply take their additive vector. This is justified through words which are similar to each other, also being closer to each other in the vector space. If an individual word is not in the vocabulary, it is ignored. The only two words from this set are "humoured" and "to". Because of the computational cost, the compositional words in the questions were dealt with first. This accounted for 56 unknown words. After the vectors had been added to the model, performance increased from 36.06% to 36.64% 2.d.p. This is an insignificant result; however, justification could be made for tackling the compositional words in the answers due to this increase.

After adding vector representations for 81 compositional answer words, the performance of our approach dropped by ~0.11% to 36.53% 2.d.p.

Model Version	Model Accuracy, %
Initial Design	36.06
Modified to Include Titled Words	34.00
Modified to include Compositional Questions	36.64
Modified to include Compositional Q/A	36.53

Figure 6: Table showing the accuracy of each modified version of the Word2Vec model

3 Bert Model

3.1 Methodology

For this approach, the BERT base model was used. This version of BERT uses a stack of 12 transformer networks [8]. To make full use of the pretrained model, the question data required some preprocessing. Firstly, the word the challenge asked to be predicted, identified by the '_____' token, is replaced by a '[MASK]' token. Also at the start of every word, a special classifier token, '[CLS]', is inserted. At the end of every sentence, a '[SEP]' token is concatenated. This preprocessing formatted the questions into the same format with which the model was pretrained with for the 'Masked Language Model' pretraining objective.

In designing a function, tokeniseSentence(), to handle this, the function was made to return the '[MASK]' token's index. This would be used later. The BERT model also has its own tokeniser method in the BertTokenizer module, which was used in our approach's preprocessing function.

```
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
```

```
def tokeniseSentence(sentence):
    sentence = sentence.replace('_____', '[MASK]')
    tokenised = ['[CLS]'] + tokenizer.tokenize(sentence) + ['[SEP]']
    mask_ind = tokenised.index('[MASK]')
    return tokenised, mask_ind
```

Figure 7: Above, the BertTokenizer module. Below this the tokeniseSentence() function makes use of the Bert tokenizer to pre-process the question data.

Next came the prediction function. This generates a prediction for the masked word. To begin with, the question tokens need to be converted into a sequence of ids. These are calculated from the models vocabulary. After this, both the token ids and segement id are converted into tensors. This is to exploit the PyTorch library. The token ids are then passed through the BERT model, with the output being token ids of the whole vocabulary. The token id which maximises the prediction is then chosen and converted back to a token. This is then returned.

The problem that arose from this method, is the predicted word was often not included in the 5 possible answers to choose from. To combat this, the intial code base would select an answer at random when this was the case. Like the Word2Vec approach, the questions and answers were assessed by the model in turn.

3.2 Initial Evaluation

Our method of evaluation is the same as in our Word2Vec approach. However, as there is a random element to our model answer choice, we take the average of 10 runs. The average performance of our model measures at 31.83% 2.d.p. This is only ~11% better than what would be expected if the answer selection was completely random. This is because so few of the provided answers, roughly 140, are being generated by the model. This could be because either the BERT model does not lend itself well to masked word prediction, or the training set and testing set are very different. Considering the pretraining objective is the 'Masked Language Model', the former reason can be ignored. When we look at the origin of the question set, Sherlock Holmes books, we can see how this may affect the accuracy. As these books are old, the frequency distribution of a word and its context may have changed. This drastically affects how a word is represented in a vector space.

3.3 Modifications

If the training and testing data sets' difference in age is causing the predicted word to be different from the majority of the possible answers, we need to overcome this. Just because the predicted word is not in the answers, does not mean that it isn't similar to the answers. Following this line of thinking, replacing the '[MASK]' token with the generated word, it may be viable to compute the sentence similarity with the '[MASK]' replaced with each answer in turn. To do this, each sentence was encoded into a vector representation. The cosine similarity between sentences was then calculated, with the answer being chosen from the sentence with maximum similarity. This method was done only when the predicted word was not a possible answer.

After evaluation, the model's performance significantly increased to 54.9% 2.d.p. This confirms that although the model's generated word is largely incorrect, it is still similar to the correct answer a notable proportion of the time. This likely because the context surrounding the answer and predicted word in the training and testing sets are very similar, giving the sentence with the correct answer a better similarity score. This effect is further augmented by the bidirectionality of the BERT model, which allows the words' left and right contexts to be fused together, allowing for a more complete representation.

Model Version	Model Accuracy, %
Initial Design	31.83
Modified for Sentence Similarity	54.90

Figure 8: Table showing accuracy of the BERT model approach.

4 Comparison

When comparing the two approaches taken, figure 9 shows the BERT model clearly outperformes the Word2Vec model in the Microsoft Sentence Completion Challenge. What isn't so clear is why. If we

Model Version	Model Accuracy, %
Word2Vec Initial Design	36.06
BERT Initial Design	31.83
Modified Word2Vec	36.64
Modified BERT	54.90

Figure 9: Table showing results of the two approaches for comparison.

look at each model's intial performance, the Word2Vec model scores ~4% higher than BERT. This is

likely a result of the difference in contextual information between the GoogleNews training set and the Sherlock Holmes testing set. Word2Vec seems to be able to better represent the words as vectors, allowing the distribution to be better captured on the vector space. As the BERT model is more suited to word prediction rather than similarity, the base version of our approach does not generalise well to the task. Where it pulls ahead however, is in its ability to better circumvent words unseen in the training data, and predict words similar to the possible answer. Word2Vec however, is ill-equipped to deal with unseen words, as any modifications made to account for them either lowered the accuracy or raised it by a miniscule percentage. This is evident in the ~18% difference in the modified models. It should also be noted that the computational cost of dealing with unseen words with Word2Vec is large compared to BERT.

5 Conclusion

To conclude, this paper investigated two differing approaches to the Microsoft Sentence Completion Challenge, with the aim of maximising the performance of both. In doing so, multiple modifications were made to the models in question in order to deal with issues that arose. This paper confirms the BERT model is better suited to the task as a result of it greater dexterity and adaptiveness in the face of common NLP challenges. Word2Vec seems to struggle with unseen words.

6 Further Work

For Word2Vec, some avenues of furture work should include its use in combination with N-grams in order better represent rare and infrequent words. Better representations for compositional words, like hyphenated words which occurred frequently in the challenge, are needed. It may be possible to deal with these by better fusing the context of individual components. For the BERT model, although not perfect, it does peform well. Appropriate pretraining data is vital for its peformance. Work should be done to make the model more generalisable to differing contexts.

7 References

[1] Mikolov, T., Yih, W.T. and Zweig, G., 2013, June. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 conference of the north american chapter of the association for computational linguistics: Human language technologies* (pp. 746-751).

[2] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. and Dean, J., 2013. Distributed representations of words and phrases and their compositionality. *arXiv preprint arXiv:1310.4546*.

[3] Mikolov, T., Chen, K., Corrado, G. and Dean, J., 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

[4] stanley F. Chen, Goodman, J., 1998. An Empirical study of smoothing Techniques for Language Modeling.

[5] Kim, Y., Jernite, Y., Sontag, D. and Rush, A., 2016, March. Character-aware neural language models. In *Proceedings of the AAAI conference on artificial intelligence* (Vol. 30, No. 1).

[6] Bengio, Y., Ducharme, R., Vincent, P. and Janvin, C., 2003. A neural probabilistic language model. *The journal of machine learning research*, *3*, pp.1137-1155.

[7] Mikolov, T., Sutskever, I., Deoras, A., Le, H.S., Kombrink, S. and Cernocky, J., 2012. Subword language modeling with neural networks. *preprint (http://www. fit. vutbr. cz/imikolov/rnnlm/char. pdf)*, *8*, p.67.

[8] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L. and Polosukhin, I., 2017. Attention is all you need. *arXiv preprint arXiv:1706.03762*.

[9] Bahdanau, D., Cho, K. and Bengio, Y., 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

[10] Devlin, J., Chang, M.W., Lee, K. and Toutanova, K., 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.