Microsoft Research
Sentence Completion Challenge
Assessed Coursework 1

CandNo. 183708
Advanced Natural Language Engineering
April 30, 2021

1 Introduction

This assignment required building and evaluating the performance of various models on the Microsoft Research (MSR) Sentence Completion Challenge (SCC) [1]. This paper and its implementations focus on giving a comparison of the models experimented with and exploring the ways in which they differ and the effect that that has on performance.

1.1 Sentence Completion Challenge

The challenge came about from the MSR SCC's authors observation that few public datasets existed for semantic modelling [1]. By filling this gap the authors present a way to evaluate the efficacy of a system in its task of modelling semantic meaning in text. The challenge consists of 1040 sentences, seeded from five of the original Sherlock Holmes novels. In each sentence a low-frequency focus word was selected and four alternatives to that word were generated using a maximum entropy n-gram model. The challenge here being to select the correct original word, not any of the impostors. This n-gram model was trained on approximately 540 texts from the Project Gutenberg collection [1], with most of the selected texts being 19th novels. The word generation process adhered to various criteria but all selected candidate impostor words have similar occurrence statistics with the original word. Each of the four impostor sentences were hand-picked from a larger set by human judges whose roles were to pick sentences where the generated word fit best, without making the original correct answer less clear.

1.2 Motivation

Implementing and evaluating the performance of various models on these test sentences means we can more easily examine the theory underpinning those implementations and identify nuances in the dataset. By documenting the changes in performance of the models, whilst attempting to maximise each one's score on the SCC, we can explore the effects of different features and hyper-parameter settings during the development process, and hope to gain insights into the way in which the different models process natural language.

2 Methods

The MSR SCC paper details benchmark results for six different approaches [1]. The highest scoring was their human baseline. The second highest scoring overall and highest computational baseline was achieved using latent semantic analysis (LSA) similarity calculations of the cosines of angles between the vector forms of different words against the candidate word - this answered 49% of the challenge questions correctly. The remaining four methods were variations of n-gram models. These n-gram models all performed in the range of 31% to 39%.

This investigation examines at two different implementations for sentence completion and tests the features and parameter settings to see their effect on accuracy. First to be examined is an n-gram language model comprised of unigram, bigram, trigram, and 4-gram statistics. The second method looks at word embedding-based methods. The n-gram and word embedding methods show progression in their accuracy on the test sentences and can sometimes provide an answer to each other's shortcomings. For example embedding models can move past the local information constraint imposed on n-grams and can provide global semantic coherence [2]. In addition an ensemble method which aggregates predictions in a voting system is experimented with.

2.1 N-Gram Language Model

N-grams approximate the probability of a word appearing in a sentence using the conditional probabilities of previous words, i.e. they are statistical models that are able to use context to estimate word probabilities. This process relies on the chain rule of probabilities, Markov assumptions and the maximum likelihood estimation. These three components relate to: computing the probabilities of word sequences, the assumption that the probability of a word depends solely on the previously seen words, and that probabilities can be estimated by normalizing the n-gram counts, as seen in the corpus. N-gram models have performed well in a range of tasks, from speech recognition and machine translation to augmentative and alternative communications systems [3, 4]. These tasks are not dissimilar to sentence completion as all can be completed by assigning probabilities to sequences of words. This was a factor in determining the suitability of an n-gram model for this challenge.

2.2 Word Embeddings Model

Modelling words as points in high dimensional space creates word embeddings, using these dense vectors to model semantics is known as Latent Semantic Analysis.[5]. This subject area was expanded on with Word2vec [6], which uses a neural network as a predictive model, and GloVe [7] which is a count based method that uses co-occurrence information. The vectorized forms of words can then be compared with one another by computing a measure between their vectors, generally using some function of the dot product such as cosine or euclidean distance. This report explores two different methods: Word2Vec and its extension fastText [8, 9]. Intrinsic evaluation methods do exist for these models which compare the model's word similarity scores to those assigned by humans, such as the WordSim-353 [10] or the TOEFL dataset [11] which has the model select the correct synonym for a target word. However, extrinsic evaluations for vector models are generally more useful as one can see directly whether there is any improvement in performance for the task [3], and so evaluation for the embedding models will take place on the MSR-SCC challenge sentences themselves and model's accuracy on the set of test sentences.

3 Experimental Results

This section details the results of experimentation with the n-gram and word embedding models, as well as an ensemble method combining the two. If models are trained on text (rather than pretrained) this will be referring to a subset of the collection of Project Gutenberg texts, more information regarding these is available in Appendix A as well as some example test sentences for clarity.

3.1 N-Gram Language Model

The first method of scoring takes an individual order of n-gram, let's say bigram, and returns the candidate token which maximizes the bigram probability of that candidate token and its context. The models here handle out of vocabulary (OOV) words by passing the model a numerical parameter - *known* - upon initialization which replaces all n-grams with occurrence statistics less than the *known* value with an unknown token. This n-gram model was trained initially on increasing numbers of Project Gutenberg texts and perplexity calculations were done on a test size 20% the size of the training set. For expositional clarity one can assume that all figures quoted relating to the number of texts processed are approximated even if not stated so due to decoding errors of some of the text files. Table 1 shows the effect of increasingly large training and testing sets on perplexity for different n-gram models - the perplexity test set is roughly 20% of the number quoted in the table. For reference the average sentence length including words and punctuation was 12.04 and the average number of sentences per document was 7511. Sentences here refer to lines of text in the Project Gutenberg files.

N-Gram	Perplexity		
	8	20	40
Unigram	107.0178	116.2954	221.4313
Bigram	51.6554	56.5050	74.3852
Trigram	37.5881	25.3478	17.0038
4-gram	12.3239	9.2456	6.4820

Table 1: Perplexity scores with increasing sizes of documents the models trained on

Table 1 shows that the model fits the data better with increasing n-gram sizes; a trend seen across all recorded documents sizes. Comparisons of perplexity across models trained on different amounts of text is not possible as they would not have used identical vocabularies [12]. Visualizing the sentences generated using the n-gram models trained on 8 and 40 texts also shows an increase in apparent coherence. This is shown below with the left most value corresponding to the size of n-gram model used:

Document Size: **8**

1. **-** the the , , the the the , ,
2. **-** and the same time , and the
3. **-** alone knew where Anne

4. **-** 'they shut or opened their gates with a trembling hand ,

Document Size: **40**

1. **-** the the
2. **-** the same as the first
3. **-** careless people should think
4. **-** had spoken its simple reason through the lips of Dejah Thoris ' prison before the long

The unigram and bigram models across the three sizes are predisposed to select the most common word sequences or words that were observed in the training data - stopwords and punctuation. Figure 1 explores this by looking at word token occurrence statistics for the model trained on 40 texts. Figure 1 also helps explain the short sentence length in the Shannon visualization [13] sentences; the unknown token, '__UNK' is the most commonly occurring token and was used as a cut off for any sentence generated. This could be due to the numerical *known* parameter passed to the the n-gram model. Across all three documents this value was initially tested at 50 and so for the 8 document model, its perplexity might have been artificially low due to the small vocabulary size and the unknown token being assigned a high probability - at $known = 50$ many trigrams and higher would be trimmed. Figure 2 looks at this possibility in the unigram and bigram cases by exploring the effects of the known parameter on a models' test set perplexity. The models shown below had fixed vocabularies as training and testing data remained unchanged.

Figure 2 shows perplexity decreasing as *known* is increased and more n-grams have probability mass redistributed towards the unknown token. Inspecting the size of the vocabulary for the models with vary-

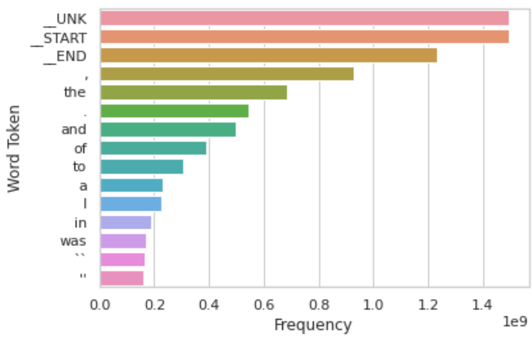

Figure 1: Word Token Occurrence Statistics

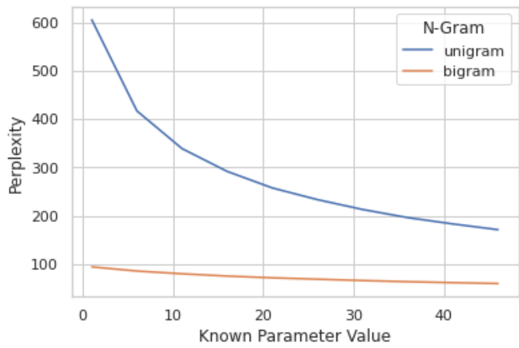

Figure 2: Perplexity against varying values of the *known* parameter

ing sizes of the *known* parameter show the largest decrease came from using $known = 1$ to $known = 6$, which cut the size of the set from 36971 to 9955. Table 2 shows the extrinsic evaluation: the scores measured in accuracy on the challenges' test sentences for models trained on 8, 20, 40, 200 texts with no pre-processing of the data or smoothing applied. The *known* parameter was cut to 5 for each. The scoring system used here looks at the context of each candidate word, calculates the probability of that n-gram with the candidate word occurring, and selects the word contained in the highest yielding n-gram as the choice for that sentence. In the trigram and 4-gram cases this method performs little better than random chance (20%) whereas some improvement is observed in the unigram and bigram cases. The * attached to the 19.90 value here and in further experiments symbolizes that the n-gram model had no information to help answer the challenge sentence and so it defaulted to selecting option A for that sentence.

N-Gram	Accuracy (%)			
	8	20	40	200
Unigram	25.87	25.29	23.75	24.90
Bigram	25.29	24.90	25.29	27.60
Trigram	18.85	19.13	19.13	19.33
4-gram	20.19	20.29	19.90*	20.77

Table 2: MSR-SCC score of n-gram models with varying sizes of training set

Using the context to the right of the target word rather than left was attempted with the highest scoring bigram model. This reduced accuracy from 27.60 to 21.54. Table 3 seeks to explain this by examining the most frequent word tokens on either side of the candidate word choices in the test sentences, as it was thought that it might be due to a greater frequency of lexically insignificant words in the set of right context tokens. A more precise measure which counted each occurrence of a stopword or a punctuation character revealed a greater number of these less meaningful tokens occurring on the right side of the candidate choice. Figure 3 looks at updated word occurrence statistics for a model trained on 40 texts with *known* cut down further to equal two and with stopwords and punctuation removed. Note this image has the sentence start and end markers removed for clarity due to their high frequency - the full image can be found in Appendix as Figure 8. Removing stopwords and punctuation dropped average sentence length from 12.04 to 5.79 tokens.

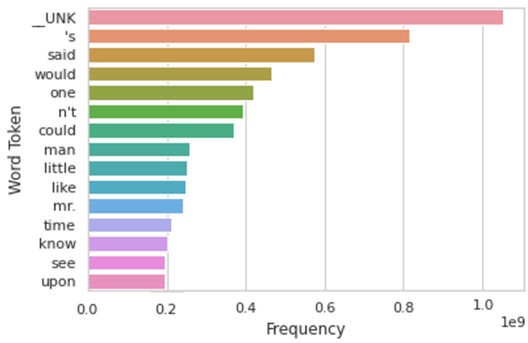

Figure 3: Word token occurrence statistics with stopwords and punctuation removed

The MSR SCC test sentences can place the target word at a point where its context only contains stopwords or punctuation. For the extrinsic evaluation to run without error, all contexts are padded with the minimum number of unknown tokens the model might require to predict the correct candidate word for the sentence. Using the same method of evaluation as Table 2 and with *known* set to 2, Table 4 shows slight improvements in the bigram cases, with a model trained on 200 documents achieving 30.1% accuracy. We observe the trigram model failing until the size of the training set is increased and that the 4-gram model never produces a match.

Left	Right
the	,
a	of
was	and
to	in
and	the
had	to

Table 3: Most frequent tokens used in left and right context bigram cases

N-Gram	Accuracy (%)			
	8	20	40	200
Unigram	26.63	24.42	24.90	23.65
Bigram	21.73	24.04	27.21	30.10
Trigram	19.90*	20.19	20.29	20.67
4-Gram	19.90*	19.90*	19.90*	19.90*

Table 4: MSR-SCC score of N-gram models with stopwords and punctuation removed

The ”Stupid Backoff” method [14] was next tested. Backoff is a way of dealing with out of vocabulary (OOV) word sequences that draws on available contextual information by using a lower order n-gram to estimate probability when the higher order n-grams do not exist. Along with handling the OOVs, using less context can help the language model by generalizing for lesser seen contexts [3]. Stupid Backoff does not generate normalized probabilities, opting instead to use the relative frequencies of the n-grams [14]. The function S describes this distribution of frequency scores:

$$S(w_i|w_{i-k+1}^{i-1}) = \begin{cases} \frac{f(w_{i-k+1}^{i}}{f(w_{i-k+1}^{i-1}} & \text{if} f(w_{i-k+1}^{i}) > 0 \\ \alpha S(w_i|w_{i-k+2}^{i-1}) & \text{otherwise} \end{cases}$$

The α term is known as the backoff factor and acts as a context independent weighting for each order of n-gram calculation. Empirical evidence from the original paper suggests using $\alpha = 0.4$. It was also noted that using multiple values depending on the order of n-grams can improve results slightly [14]. Table 5 shows the results of using Stupid Backoff on n-gram models trained on 80 texts with $known = 5$.

N-Gram	Accuracy (%)		
	Stop	Stop + Lemma	None
Unigram	24.81	24.42	25.29
Bigram	29.42	29.04	26.92
Trigram	27.12	26.54	27.12
4-Gram	25.19	24.23	27.5

Table 5: MSR-SCC Stupid Backoff score of N-gram models with and without stopword removal and lemmatization (lemma) pre-processing

Each n-gram row in Table 5 corresponds to the highest order of n-gram used - e.g. trigram Stupid Backoff will start backing off from trigram probabilities. Pre-processing the sentences appears to have little effect and if α is set to not penalize trigram weights the 4-gram model performs roughly as well as the trigram, indicating that 4-gram information is still rarely available. Word tokens were lemmatized in the hope that it would help the model generalize better however for each order of n-gram it underperformed the version with only stopwords (and punctuation) removed.

Using the same scoring system as the MSR SCC simple 4-gram baseline [2] with the same bigram, trigram, and 4-gram probabilities, Table 6 shows an improvement in the accuracy of the results. This method matches n-grams up to 4-grams of the test sentence that contain the target word. A different value is added per n-gram match (+1 bigram, +2 trigram, +3 4-gram.) and the candidate word with the highest score is selected. Building off the best score given in Table 6 - maintaining $known = 5$ and not applying any pre-processing to the training or test data - by increasing the training set size to the maximum of approximately 250 Project Gutenberg texts the simple 4-gram evaluation method yields 34.9% correct. Training on the same texts with stopwords removed yielded 34.42% correct.

N-Gram	Accuracy (%)		
	Stop	Stop + Lemma	None
Simple 4-Gram	31.15	30.87	31.25

Table 6: MSR-SCC "simple 4-gram" score with and without sentence pre-processing

3.2 Word Embeddings Model

First, pretrained embeddings (with 300 dimensions) for Word2Vec and fastText are tested. These models' base datasets are found in Appendix A. Their accuracy on the challenge sentences using vector similarity and distance measures before and after sentence pre-processing is applied gives the results seen in Table 7.

The scores in Table 7 were computed using a total similarity system [2]. In the cosine similarity case this means selecting the candidate word with the greatest average similarity to all other words in the test sentence. In the Euclidean distance case this means selecting the candidate word with the least average distance to all other words. Removing stopwords and punctuation which carry insignificant lexical meaning shows an improvement.

Embedding Method	Accuracy (%)	
	Cosine	Euclidean
Word2Vec	36.06	29.52
Word2Vec - PP	38.75	31.06
fastText	35.10	24.52
fasttext - PP	42.88	28.17

Table 7: Cosine and Euclidean total similarity test sentence accuracy with and without stopword removal (pre-processing : PP)

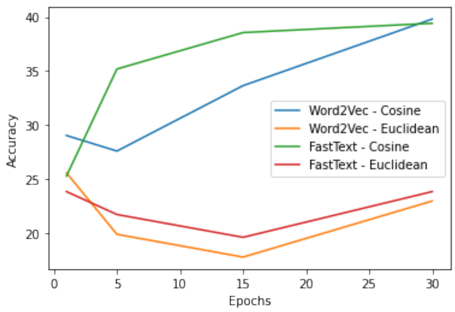

Figure 4: Accuracy of models on test sentences with a varying number of epochs trained for

These results however are still below that of the LSA baseline given in the original MSR SCC publication and so embeddings are instead generated from the Project Gutenberg training data. As well as matching the training and testing domains (19th century novels) using these embeddings means more freedom in the hyper-parameter settings when creating the models. The same sentence pre-processing in Table 7 is applied going forward. Next we explore the effect of the *epoch* parameter on Word2Vec and fastText in Figure 4. This dictates the number of iterations the algorithm (skip-gram) runs over the corpus. The models in Figure 4 were trained on 30 texts, other parameters were set as default (see

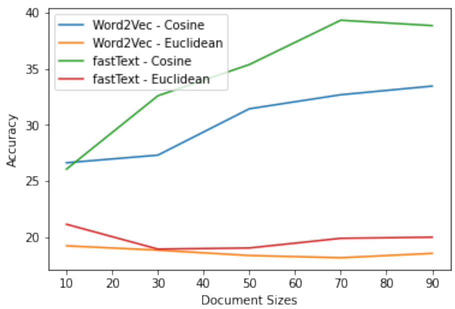

Figure 5: Accuracy of models on test sentences with increasing training documents size

Appendix A). Unless otherwise stated one can assume unmentioned parameters to be set at default values. Both models' Euclidean distance measures drop off after training for 15 epochs however past that point a steady increase is observed across all metrics. Maintaining *epoch* = 15, Figure 5 shows the effects of increasing the number of documents the model is trained on. In the cosine similarity case the model's performance increases with the size of the training set however Euclidean distance measures underperform. Exploring the *window* size parameter, with models trained on 30 documents, is seen in Figure 6. Euclidean distance measures were dropped due to continued low performance. Window size measures the maximum distance between the current and algorithm's predicted word within a sentence.

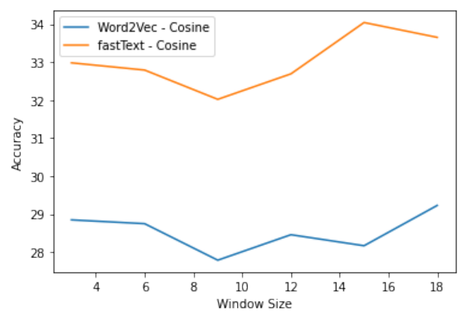

Figure 6: Accuracy of models with varying window sizes

Maintaining the parameters used for Figure 6 but reducing the desired word embedding dimensions from 300 to 100 and re-running the above window size test gives Figure 7 - this offers improvements over the previous tests in terms of accuracy and cut training time by a large factor.

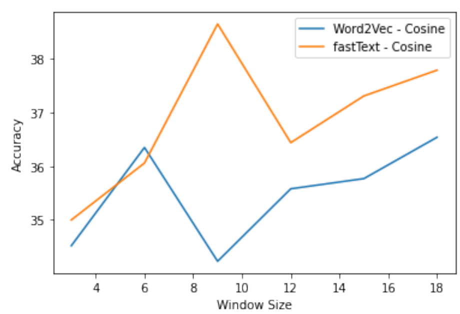

Figure 7: Accuracy of models of 100 dimensions with varying window sizes

3.3 Ensemble

Methods can be combined in machine learned systems to obtain even better results [2, 15].This ensemble method was implemented which took the scores from the two best performing models overall, pretrained fasttext embeddings and the simple-4-gram, normalized each score against the sum of those scores and added the two sets of scores together. The candidate word with the highest aggregate score was selected. This ensemble voting system improved slightly from the individual scores and scored 44.6% correct.

4 Analysis

Improvements were seen across both main methods tested as their hyperparameters and data pre-processing steps were tuned towards the task at hand. A summary of the best performing models is given in Table 8. The simple-4-gram model is the only variation listed that had no pre-processing applied and outperformed a similar model with pre-processing. Although the difference was minor (less than 1% change in accuracy) it could be the case that stopwords give relevant context to the target word such as inferring the required part of speech for the candidate choice. N-gram models are also sensitive to OOV words and so success was dependent on how the unknowns were handled via the *known* parameter - Figure 2 also showed that too high a *known* setting could potentially skip the model's process of learning by over-generalizing its predictions for text with excessive probability mass stored for the unknown token. With increased document sizes and so increased quantities of n-grams, it was observed in Table 1 that increasing the size of n would decrease perplexity however with each order of n, the effect was diminished, suggesting there is a cut-off point where adding additional n-gram complexity does not translate into an equal effect on the models' ability to predict text. Initially the trigram and 4-gram models performed worse than their unigram and bigram counterparts but the in-

Model	Accuracy (%)
Simple-4-gram	34.90
Pretrained: Word2vec	38.75
Custom: fastText	39.33
Pretrained: fastText	42.88
Ensemble (fastText + 4-gram)	44.60

Table 8: Best performing Models

troduction of Stupid Backoff allowed the language model to make use of all available information and use the rarer trigram and 4-gram information where possible. Lemmatizing the text as well as removing stopwords did not perform as well as solely removing stopwords perhaps due to the reduced number of unique n-grams and the inability to preserve context in full after lemmatization. In one of the tokenized pre-processed test sentences - ['tortured', 'tried', 'get', 'away', '_____', 'tortured'] - the pre-processed simple-4-gram model predicted the target word as [laughing] when the correct answer was [captured]. Here it is suspected the n-gram model had trained on more occurrences of unigram, bigram, and even trigram variations that contained ['tried', 'get', 'away'] and [laughing] than it had with [captured]. This does not show semantic understanding as [laughing] is clearly out of place but the n-gram model is at the mercy of its training data.

Out of all the word embedding methods implemented, the highest challenge score was given by the pretrained fastText model. Both pretrained Word2Vec and pretrained fastText's cosine similarity metrics and Euclidean distance measures were improved by removing lexically insignificant words. For the custom word embeddings generated from the Project Gutenberg texts, it was observed that increasing the value for the *epoch* parameter had a positive effect on accuracy on the test sentences. Figure 4 showed this relationship with a fixed size of 30 training documents. When compared with the effects of increasing document size, as seen in Figure 5, an increased *epoch* value can counterbalance the negative effects of a smaller sized training set. As document size increased the Euclidean distance measures' scores for Word2Vec and fastText decreased steadily. This could be due to the increase in magnitude of the vectorized words and n-grams, which would affect distance measures; cosine similarity measures are not as affected as they are only concerned with the angle between the vectors.

Reducing the dimensionality of the vector forms of words produced showed an increase in accuracy as well greatly reducing the training time for each custom model. This is in line with research that suggested that without sufficient data to be learnt, a too large dimension setting can make the model harder and slower to train [8]. At $dimensions = 100$, which is the recommended default for fastText, both fastText and Word2Vec models outperformed their $dimensions = 300$ counterparts for all window sizes tested in Figures 6 and 7. Larger window sizes have been shown to better capture topic information whereas smaller sizes capture information about the specific current word and any words that appear near it [16]. Table 9 demonstrates this using Word2Vec models trained on 30 documents with different window sizes. This highlights one of the ways that word embedding methods improve on n-gram models, which are limited by their context sizes to accessing only local information. Another improvement is seen specifically with fastText. fastText computes word embeddings by using the vectors of substrings of characters contained within the word, allowing it to model OOV occurrences by aggregating the n-grams that the OOV word is made up of. For example in the pre-processed test sentence: ['holmes', 'pulled', 'large', 'sheet', '_____', 'pocket', 'carefully', 'unfolded', 'upon', 'knee'] the simple-4-gram model selects [iron] as the candidate when the answer was [tissue-paper]. In this case pre-processing may have split the correct answer into separate tokens (if tissue-paper occurred in the training data) so the simple-4-gram model had to default to using the unknown token for [tissue-paper] whereas fastText was able to aggregate the n-grams to make a correct prediction.

Word	Window = 2	Window = 25
butcher	mastiff	baker
chicken	sweetbread	partridges

Table 9: Most similar word at different window sizes

The ensemble method which aggregated normalized candidate word scores of the pretrained fastText model and the simple-4-gram model into an equally weighted voting system did improve on either method's solo challenge accuracy however it only gave 18 correct answers to sentences that neither model individually got correct. In terms of correct answers types, the ensemble method favoured the embedding method's prediction 132 times over the n-gram prediction and the simple-4-gram method's prediction 110 times over the word embedding prediction. The remaining correct answers (204) were when simple-4-gram and word embedding methods were in agreement.

5 Discussion

This report has found that n-gram and word embedding-based implementations can return acceptable scores on the MSR SCC. Striking a balance between the size of the training set and the n-gram model's method for dealing with unknowns can decrease the perplexity reported from a test text, possibly showing an improvement in semantic coherence. This translates into improvements in accuracy on the challenge questions. Stupid Backoff as well as the simple-4-gram method allows different orders of n-gram to contribute information to further this. The word embedding models using total cosine similarity perform better than n-gram generally and can have settings tuned to the training set to improve accuracy further. The improvement in performance may be due to the lack of constraint when dealing with less frequently seen word sequences and by capturing semantic information from long span contexts. To show statistical significance, a one-tailed binomial hypothesis test was carried out on the results of the ensemble method to demonstrate the unlikelihood that its performance was due to random chance - the outcome of the test meant the null hypothesis (that the model was performing as random chance) was rejected. The full results are attached in Appendix A. Further investigations may be fruitful in exploring the effects of popular smoothing algorithms on the n-gram models as well as exploring the effects of the *known* parameter as training set size increases. Similarly investigating the effects of using Word2Vec and fastText's continuous bag of words algorithm rather than solely testing skip-gram on the domain specific word embedding methods would have been an interesting area to explore.

6 Conclusion

This report presents an investigation into methods for sentence completion. The n-gram and word embedding based methods provide a good benchmark for the task and can accurately predict the correct word for a sentence based on computationally simple and inexpensive methods. Both give scores better than random chance, with the word embedding based methods achieving slightly less than half correct. Extensions of this work would be finding more effective ways to combine local and global sentence information in ensemble methods to achieve even greater accuracy.

Bibliography

[1] G. Zweig and C. Burges, "The microsoft research sentence completion challenge," 01 2011.

[2] G. Zweig, J. Platt, C. Meek, C. Burges, A. Yessenalina, and Q. Liu, "Computational approaches to sentence completion," *50th Annual Meeting of the Association for Computational Linguistics, ACL 2012 - Proceedings of the Conference*, vol. 1, 07 2012.

[3] D. Jurafsky and J. H. Martin, *Speech and Language Processing (2nd Edition)*. USA: Prentice-Hall, Inc., 2009.

[4] K. Trnka, D. Yarrington, J. McCaw, K. F. McCoy, and C. Pennington, "The effects of word prediction on communication rate for aac," in *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Companion Volume, Short Papers*, ser. NAACL-Short '07. USA: Association for Computational Linguistics, 2007, p. 173–176.

[5] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American Society for Information Science*, vol. 41, no. 6, pp. 391–407, 1990.

[6] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *ICLR*, 2013.

[7] J. Pennington, R. Socher, and C. Manning, "GloVe: Global vectors for word representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543.

[8] T. Mikolov, E. Grave, P. Bojanowski, C. Puhrsch, and A. Joulin, "Advances in pre-training distributed word representations," in *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*, 2018.

[9] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *CoRR*, vol. abs/1607.04606, 2016. [Online]. Available: http://arxiv.org/abs/1607.04606

[10] "Placing search in context: The concept revisited," *ACM Trans. Inf. Syst.*, vol. 20, no. 1, p. 116–131, Jan. 2002. [Online]. Available: https://doi.org/10.1145/503104.503110

[11] T. Landauer and S. Dumais, "A solution to plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge." *Psychological Review*, vol. 104, pp. 211–240, 1997.

[12] C. Buck, K. Heafield, and B. V. Ooyen, "N-gram counts and language models from the common crawl," in *LREC*, 2014.

[13] C. Shannon, "Prediction and entropy of printed english," *Bell System Technical Journal*, vol. 30, pp. 50–64, 1951.

[14] T. Brants, A. C. Popat, P. Xu, F. J. Och, and J. Dean, "Large language models in machine translation," 2007.

[15] T. G. Dietterich, "Ensemble methods in machine learning," in *Proceedings of the First International Workshop on Multiple Classifier Systems*, ser. MCS '00. Berlin, Heidelberg: Springer-Verlag, 2000, p. 1–15.

[16] O. Levy and Y. Goldberg, "Dependency-based word embeddings," in *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Baltimore, Maryland: Association for Computational Linguistics, Jun. 2014, pp. 302–308. [Online]. Available: https://www.aclweb.org/anthology/P14-2050

A Appendix

Example Sentences from the MSR SCC:
Available at `https://www.microsoft.com/en-us/research/project/msr-sentence-completion-challenge/`

Two example sentences are given with their five candidate word choices listed below. The original and correct answer is in bold and the string '_____' marks the target position of the word in the sentence.

I have it from the same source that you are both an orphan and a bachelor and are _____ alone in london.				
crying	instantaneously	**residing**	matched	walking

As I descended , my old ally , the _____ , came out of the room and closed the door tightly behind him.				
gods	moon	panther	**guard**	country-dance

Image displaying the occurrence statistics of n-gram model trained on 40 texts, $known = 2$. This image includes the start and end sentence markers: "__START", "__END"

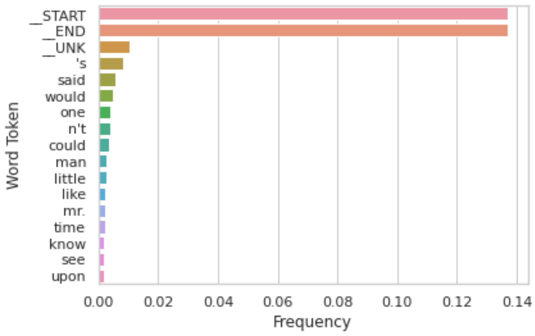

Figure 8: Word token occurrence statistics with stopwords and punctuation removed

Pretrained Word Embeddings: Licenses and additional information can be found at:
`https://github.com/RaRe-Technologies/gensim-data`

fasttext-wiki-news-subwords-30 Wikipedia 2017, UMBC webbase corpus and statmt.org news dataset (16B tokens)

word2vec-google-news-300 Google News (about 100 billion words)

Default parameters for the custom embedding models can be found at:

Word2vec `https://radimrehurek.com/gensim/models/word2vec.html#gensim.models.word2vec.Word2Vec`

fastText `https://radimrehurek.com/gensim/models/fasttext.html#gensim.models.fasttext.FastText`

Binomial Hypothesis Test using a confidence level of 95% ($\alpha = 0.05$) for a one-tailed test. Total of 1040 trials with our ensemble model successfully completing 464 of them. It is assumed that trials (questions) are mutually independent and the probability of a given outcome is the same for all. Random choice here denotes one in five chance of a correct answer and four in five chance of incorrect.

Let x equal the number of times the model answers a question successfully.
Let π equal the probability of success in any one trial.

$H_0 : \pi \leq 0.2$ i.e. due to random chance.
$H_1 : \pi > 0.2$

$P(x \geq 464) = 7.26\text{e}{-72}$ and so we reject the null hypothesis at the 5% significance value because the returned p-value is less than the critical value of 0.05.

*—————— **Source code is available overleaf** ——————*

```
# -*- coding: utf-8 -*-

import os
import random, math
import numpy as np
import pandas as pd

from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle

import operator
import nltk
from nltk import word_tokenize as tokenize
nltk.download("punkt")
nltk.download("wordnet")

from nltk.stem import   WordNetLemmatizer
wordnet_lemmatizer = WordNetLemmatizer()


import tqdm

# Download lab2 resources.
os.system("gdown --id 1H26pdLFh2cDxU-Nkf1QHzcNCYWUgHCbX")
os.system("unzip lab2resources.zip")

# Download scc resources.
os.system("gdown --id 155TLf2OdXtvfPD8VsWwI2YlHfjS8ph04")

# Stopwords
from nltk.corpus import stopwords
nltk.download("stopwords")

import string

stop = set(stopwords.words("english"))
punc = string.punctuation

# Create stopword + punctuation list.
stop_puncs = (set([x for x in punc] + list(stop)))


def get_training_testing(training_dir,split=0.5):
    """
    Get training testing files.
    """
    filenames=os.listdir(training_dir)
    n=len(filenames)
    print("There are {} files in the training directory: {}".format(n,training_dir))
    # random.seed(53)  #if you want the same random split every time
    random.shuffle(filenames)
    index=int(n*split)
    trainingfiles=filenames[:index]
    heldoutfiles=filenames[index:]
    return trainingfiles,heldoutfiles

parentdir="lab2resources/sentence-completion"
trainingdir=os.path.join(parentdir,"Holmes_Training_Data")
training,testing=get_training_testing(trainingdir)

"""## N-Gram model"""

class n_gram_language_model():

    """
    N-gram language model class that stores n-grams and their probabilties learnt from training text
    in individiual dictionaries.

    Code adapted from the original work of Dr. J Weeds, University of Sussex.

    Parameters
    ----------
    trainingdir : str
        The training directory where training data can be found.
    files : list
        List of file names to be trained on.
    test_files : list
        List of file names for the model to be tested on.
    construct_params : dict
        Stores the parameters such as known to initialize the language model with.
    Attributes
```

```
    ----------
    trainingdir : str
        The training directory where training data can be found.
    files : list
        List of file names to be trained on.
    test_files : list
        List of file names for the model to be tested on.
    construct_params : dict
        Stores the parameters such as known to initialize the language model with.
    verbose : bool
        Whether or not method calls will print progress.
    unigram : dict
        Dictionary to store unigram probabilities.
    bigram : dict
        Dictionary to store bigram probabilities.
    trigram : dict
        Dictionary to store trigram probabilities.
    4-gram : dict
        Dictionary to store 4-gram probabilities.
    """


    def __init__(self,trainingdir,files=[], test_files=[], construct_params={}):
        self.training_dir=trainingdir
        self.files=files
        self.test_files = test_files
        # Constructor Parameters.
        self.construct_params=construct_params
        self.verbose = construct_params.get("verbose", False)
        self.train()

    def train(self):
        """
        Method called in model initialization.
        Calls "private" methods which process files, make unknowns, discount and convert n-gram
    dictionaries to probabilities.
        """
        self.unigram={}
        self.bigram={}
        self.trigram={}
        self.quad_gram={}

        self._processfiles()
        self._make_unknowns(known=self.construct_params.get("known",2))
        self._discount()
        self._convert_to_probs()



    def _processline(self,line):
        """
        Method processes lines of txt files and tokenizes sentences contained within.
        Information is stored within respective n-gram dictionaries.
        """
        tokens=tokenize(line)
        if self.construct_params.get("remove_stopwords",False) == True:
          tokens = [token.lower() for token in tokens if token.lower() not in stop_puncs]
        if self.construct_params.get("lemmatize", False) == True:
          tokens = [wordnet_lemmatizer.lemmatize(token) for token in tokens]
        tokens = ["__START"] + tokens + ["__END"]
        previous="__END"
        for i, token in enumerate(tokens):
            # Unigram
            self.unigram[token]=self.unigram.get(token,0)+1
            # Bigram
            current=self.bigram.get(previous,{})
            current[token]=current.get(token,0)+1
            self.bigram[previous]=current
            previous=token
            # Trigram
            if i < len(tokens)-2:
              # Next words.
              next = tokens[i+1]
              next_next = tokens[i+2]
              # Get dictionaries.
              inner = self.trigram.get(token,{})
              innermost = inner.get(next,{})
              innermost[next_next] = innermost.get(token,0) + 1
              # Write frequencies to dictionaries.
              inner[next] = innermost
              self.trigram[token] = inner
```

```
            # 4-gram
            if i < len(tokens)-3:
              # Next words.
              next1 = tokens[i+1]
              next2 = tokens[i+2]
              next3 = tokens[i+3]
              # Get dictionaries.
              inner1 = self.quad_gram.get(token,{})
              inner2 = inner1.get(next1,{})
              inner3 = inner2.get(next2,{})
              inner3[next3] = inner3.get(token,0) + 1
              # Write frequencies to dictionaries.
              inner2[next2] = inner3
              inner1[next1] = inner2
              self.quad_gram[token] = inner1



    def _processfiles(self):
        """
        Process text files.
        """
        for afile in tqdm.tqdm(self.files):
            # print("Processing {}".format(afile))
            try:
                with open(os.path.join(self.training_dir,afile)) as instream:
                    for line in instream:
                        line=line.rstrip()
                        if len(line)>0:
                            self._processline(line)
            except UnicodeDecodeError:
              if self.verbose:
                print("UnicodeDecodeError processing {}: ignoring rest of file".format(afile))
              else:
                pass


    def _convert_to_probs(self):
        """
        Convert counts to probabilities for each n-gram dictionary.
        """
        self.unigram={k:v/sum(self.unigram.values()) for (k,v) in self.unigram.items()}
        self.bigram={key:{k:v/sum(adict.values()) for (k,v) in adict.items()} for (key,adict) in self.bigram.items()}
        self.trigram={k1:{k2:{k3:v/sum(adict2.values()) for k3, v in adict2.items()} for k2, adict2 in adict1.items()} for k1, adict1 in self.trigram.items()}
        self.quad_gram={k1:{k2:{k3:{k4:v/sum(adict3.values()) for k4, v in adict3.items()} for k3, adict3 in adict2.items()} for k2, adict2 in adict1.items()} for k1, adict1 in self.quad_gram.items()}
        self.kn={k:v/sum(self.kn.values()) for (k,v) in self.kn.items()}


    def nextlikely(self,k=1,current="",method="unigram"):
        #use probabilities according to method to generate a likely next sequence
        #choose random token from k best
        blacklist=["__START","__UNK","__DISCOUNT"]
        most_likely = []
        if method=="unigram":
            dist=self.unigram
            #sort the tokens by unigram probability
            most_likely=sorted(list(dist.items()),key=operator.itemgetter(1),reverse=True)
        elif method == "bigram":
            dist=self.bigram.get(current,self.bigram.get("__UNK",{}))
            most_likely=sorted(list(dist.items()),key=operator.itemgetter(1),reverse=True)
        elif method == "trigram":
            # Split context string for first and second context words.
            context = current.split()
            c1, c2 = context[0], context[1]
            dist = self.trigram[c1][c2]
            # Get all words with maximum value.
            most_likely = [(k, _) for k, v in dist.items() if v == max(dist.values())]
        elif method == "quad_gram":
            context = current.split(" ")
            c1,c2,c3 = context[0], context[1], context[2]
            dist = self.quad_gram[c1][c2][c3]
            most_likely = [(k, _) for k, v in dist.items() if v == max(dist.values())]
        #filter out any undesirable tokens
        filtered=[w for (w,p) in most_likely if w not in blacklist]
        #choose one randomly from the top k
        res=random.choice(filtered[:k])
        return res
```

```
    def generate(self,k=3,end="__END",limit=20,method="bigram",methodparams={}):
        """
        Example sentence generator method: Shannon Visualizations.
        k selects from the best top(k)s.
        """
        if method=="":
            method=methodparams.get("method","bigram")
        current="__START"
        tokens=[]
        try:
          # Trigram
          if method=="trigram":
            # Set current word to first context.
            context_1 = current
            # Set random choice of next word to second context.
            context_2 = random.choice([key for key, adict in self.trigram[current].items()])
            # Check end token hasnt been reached.
            while context_2 != end and len(tokens)<limit:
              # Pass current contexts to next likely method which re splits them in the tri- 4-gram cases.
              current = " ".join([context_1, context_2])
              current = self.nextlikely(k=k, current=current, method=method)
              # Append word to the list that will eventually be generated.
              tokens.append(current)
              # Set the the second context to now be first and the predicted word (current) to be next.
              context_1 = context_2
              context_2 = current
            # After loop return the tokens joined by whitespace.
            return " ".join(tokens[:-1])
          # Quad-Gram
          elif method == "quad_gram":
            # Functionality is the same as above with an additional context variable to account for 4 rather than 3 n-grams.
            context_1 = current
            context_2 = random.choice([key for key, adict in self.quad_gram[context_1].items()])
            context_3 = random.choice([key for key, adict in self.quad_gram[context_1][context_2].items()])
            while context_3 != end and len(tokens) < limit:
              current = " ".join([context_1, context_2, context_3])
              current = self.nextlikely(k=k, current=current, method=method)
              tokens.append(current)
              context_1 = context_2
              context_2 = context_3
              context_3 = current
            return " ".join(tokens[:-1])
        except:
          # If error is thrown rerun method until it generates a valid sentence.
          return self.generate(k=k,end=end,limit=limit,method=method,methodparams=methodparams)
        # Below calls the unigram and bigram versions of the method.
        while current!=end and len(tokens)<limit:
            current=self.nextlikely(k=k,current=current,method=method)
            tokens.append(current)
        return " ".join(tokens[:-1])


    def get_prob(self,token,context="",methodparams={}):
        if methodparams.get("method","unigram")=="unigram":
            return self.unigram.get(token,self.unigram.get("__UNK",0))
        else:
            if methodparams.get("smoothing","kneser-ney")=="kneser-ney":
                unidist=self.kn
            else:
                unidist=self.unigram
            bigram=self.bigram.get(context[-1],self.bigram.get("__UNK",{}))
            big_p=bigram.get(token,bigram.get("__UNK",0))
            lmbda=bigram["__DISCOUNT"]
            uni_p=unidist.get(token,unidist.get("__UNK",0))
            #print(big_p,lmbda,uni_p)
            p=big_p+lmbda*uni_p
            return p


    def compute_prob_line(self,line,methodparams={}):
        """
        Refactored method which calls get_probs() for uni- and bigram cases. Contains functionality for tri- and 4-gram cases within.
        Method is not commented as it should be self explanatory:
        Lots of if else statements to fit the contexts into a n-gram dictionary.

```

```python
        #this will add _start to the beginning of a line of text
        #compute the probability of the line according to the desired model
        #and returns probability together with number of tokens
        """
        tokens=tokenize(line)
        if self.construct_params.get("remove_stopwords",False) == True:
          tokens = [token.lower() for token in tokens if token.lower() not in stop_puncs]
        if self.construct_params.get("lemmatize", False) == True:
          tokens = [wordnet_lemmatizer.lemmatize(token) for token in tokens]
        tokens = ["__START"] + tokens + ["__END"]
        acc=0
        if methodparams.get("method", "unigram") in ["unigram", "bigram"]:
          for i,token in enumerate(tokens[1:]):
            acc+=math.log(self.get_prob(token,tokens[:i+1],methodparams))
          return acc,len(tokens[1:])
        # Trigram.
        if methodparams.get("method") == "trigram":
          try:
            for i, token in enumerate(tokens[1:]):
              if i < len(tokens[1:]) - 3 and len(tokens[1:]) >= 3:
                word1, word2, word3 = tokens[i+1], tokens[i+1+1], tokens[i+1+2]
                if word1 in self.trigram:
                  if word2 in self.trigram[word1]:
                    if word3 in self.trigram[word1][word2]:
                      acc+=math.log(self.trigram[word1][word2][word3])
                    else:
                      acc+=math.log(self.trigram[word1][word2]["__UNK"])
                  else:
                    if word3 in self.trigram[word1]["__UNK"]:
                      acc+=math.log(self.trigram[word1]["__UNK"][word3])
                    else:
                      acc+=math.log(self.trigram[word1]["__UNK"]["__UNK"])
                else:
                  if word2 in self.trigram["__UNK"]:
                    if word3 in self.trigram["__UNK"][word2]:
                      acc+=math.log(self.trigram["__UNK"][word2][word3])
                    else:
                      acc+=math.log(self.trigram["__UNK"][word2]["__UNK"])
                  else:
                    if word3 in self.trigram["__UNK"]["__UNK"]:
                      acc+=math.log(self.trigram["__UNK"]["__UNK"][word3])
                    else:
                      acc+=math.log(self.trigram["__UNK"]["__UNK"]["__UNK"])
            return acc, len(tokens[1:])
          except KeyError:
            return acc, len(tokens[1:])
        # Quad_gram - same as above. FYI - if else if statements are used rather than if elif to enhance readability.
        if methodparams.get("method") == "quad_gram":
          try:
            for i, token in enumerate(tokens[1:]):
              if i < len(tokens[1:]) - 4 and len(tokens[1:]) >= 4:
                word1, word2, word3, word4 = tokens[i+1], tokens[i+1+1], tokens[i+1+2], tokens[i+1+3]
                if word1 in self.quad_gram:
                  if word2 in self.quad_gram[word1]:
                    if word3 in self.quad_gram[word1][word2]:
                      if word4 in self.quad_gram[word1][word2][word3]:
                        acc+=math.log(self.quad_gram[word1][word2][word3][word4])
                      elif "__UNK" in self.quad_gram[word1][word2][word3]:
                        acc+=math.log(self.quad_gram[word1][word2][word3]["__UNK"])
                    else:
                      if word4 in self.quad_gram[word1][word2]["__UNK"]:
                        acc+=math.log(self.quad_gram[word1][word2]["__UNK"][word4])
                      elif "__UNK" in self.quad_gram[word1][word2]["__UNK"]:
                        acc+=math.log(self.quad_gram[word1][word2]["__UNK"]["__UNK"])
                  else:
                    if "__UNK" in self.quad_gram[word1]:
                      if word3 in self.quad_gram[word1]["__UNK"]:
                        if word4 in self.quad_gram[word1]["__UNK"][word3]:
                          acc+=math.log(self.quad_gram[word1]["__UNK"][word3][word4])
                        elif "__UNK" in self.quad_gram[word1]["__UNK"][word3]:
                          acc+=math.log(self.quad_gram[word1]["__UNK"][word3]["__UNK"])
                      else:
                        if "__UNK" in self.quad_gram[word1]["__UNK"]:
                          if word4 in self.quad_gram[word1]["__UNK"]["__UNK"]:
                            acc+=math.log(self.quad_gram[word1]["__UNK"]["__UNK"][word4])
                          elif "__UNK" in self.quad_gram[word1]["__UNK"]["__UNK"]:
                            acc+=math.log(self.quad_gram[word1]["__UNK"]["__UNK"]["__UNK"])
                else:
                  if "__UNK" in self.quad_gram:
                    if word2 in self.quad_gram["__UNK"]:
```

```python
                      if word3 in self.quad_gram["__UNK"][word2]:
                        if word4 in self.quad_gram["__UNK"][word2][word3]:
                          acc+=math.log(self.quad_gram["__UNK"][word2][word3][word4])
                        elif "__UNK" in self.quad_gram["__UNK"][word2][word3]:
                          acc+=math.log(self.quad_gram["__UNK"][word2][word3]["__UNK"])
                      else:
                        if word4 in self.quad_gram["__UNK"][word2]["__UNK"]:
                          acc+=math.log(self.quad_gram["__UNK"][word2]["__UNK"][word4])
                        elif "__UNK" in self.quad_gram["__UNK"][word2]["__UNK"]:
                          acc+=math.log(self.quad_gram["__UNK"][word2]["__UNK"]["__UNK"])
                    else:
                      if "__UNK" in self.quad_gram["__UNK"]:
                        if word3 in self.quad_gram["__UNK"]["__UNK"]:
                          if word4 in self.quad_gram["__UNK"]["__UNK"][word3]:
                            acc+=math.log(self.quad_gram["__UNK"]["__UNK"][word3][word4])
                          elif "__UNK" in self.quad_gram["__UNK"]["__UNK"][word3]:
                            acc+=math.log(self.quad_gram["__UNK"]["__UNK"][word3]["__UNK"])
                        else:
                          if "__UNK" in self.quad_gram["__UNK"]["__UNK"]:
                            if word4 in self.quad_gram["__UNK"]["__UNK"]["__UNK"]:
                              acc+=math.log(self.quad_gram["__UNK"]["__UNK"]["__UNK"][word4])
                            elif "__UNK" in self.quad_gram["__UNK"]["__UNK"]["__UNK"]:
                              acc+=math.log(self.quad_gram["__UNK"]["__UNK"]["__UNK"]["__UNK"])
            return acc, len(tokens[1:])
          except KeyError:
            return acc, len(tokens[1:])


    def compute_probability(self,filenames=[],methodparams={}):
        #computes the probability (and length) of a corpus contained in filenames
        if filenames==[]:
            filenames=self.files
        total_p=0
        total_N=0
        for i,afile in enumerate(filenames):
          if self.verbose:
            print("Processing file {}:{}".format(i,afile))
          try:
              with open(os.path.join(self.training_dir,afile)) as instream:
                  for line in instream:
                      line=line.rstrip()
                      if len(line)>0:
                          p,N=self.compute_prob_line(line,methodparams=methodparams)
                          total_p+=p
                          total_N+=N
          except UnicodeDecodeError:
            if self.verbose:
              print("UnicodeDecodeError processing file {}: ignoring rest of file".format(afile))
            else:
              pass
        return total_p,total_N

    def compute_perplexity(self,filenames=[],methodparams={"method":"bigram","smoothing":"kneser-ney"}):
        """
        compute the probability and length of the corpus
        calculate perplexity
        lower perplexity means that the model better explains the data
        """
        p,N=self.compute_probability(filenames=filenames,methodparams=methodparams)
        # print(p,N)
        if methodparams.get("method") in ["trigram", "quad_gram"]:
          rem = self.super_counter[methodparams.get("method")] - self.magic_counter[methodparams.get("method")]
          pp=math.exp(-p/N) * (self.super_counter[methodparams.get("method")]/rem)
          return pp
        pp=math.exp(-p/N)
        return pp


    def _make_unknowns(self,known=2):
        """
        Method to distribute probability mass towards the unknown token.
        param known (int): dictates cut off point where n-grams less frequent than known are pruned.
        """
        # Unigram -----------------------------------
        for (k,v) in list(self.unigram.items()):
            if v<known:
                del self.unigram[k]
                self.unigram["__UNK"]=self.unigram.get("__UNK",0)+v
        # Bigram -----------------------------------
```

```
        for (k,adict) in list(self.bigram.items()):
            for (kk,v) in list(adict.items()):
                isknown=self.unigram.get(kk,0)
                if isknown <= known:
                    # Loop into the innermost dictionary. If val is less than known then reserve that probability mass for unknown token.
                    # Delete key after saving val.
                    adict["__UNK"]=adict.get("__UNK",0)+v
                    del adict[kk]
            isknown=self.unigram.get(k,0)
            if isknown <= known:
                del self.bigram[k]
                current=self.bigram.get("__UNK",{})
                current.update(adict)
                self.bigram["__UNK"]=current
            else:
                self.bigram[k]=adict
        # Trigram -----------------------------------
        for (k1, dict1) in list(self.trigram.items()):
          for (k2, dict2) in list(dict1.items()):
            for (k3, val) in list(dict2.items()):
              isknown=self.unigram.get(k3,0)
              if isknown == 0:
                dict2["__UNK"] = dict2.get("__UNK",0) + val
                del dict2[k3]
            isknown=self.unigram.get(k2,0)
            if isknown <= known:
              del self.trigram[k1][k2]
              current=self.trigram[k1].get("__UNK",{})
              current.update(dict2)
              self.trigram[k1]["__UNK"] = current
            else:
              self.trigram[k1][k2] = dict2
          # For first token:
          isknown=self.unigram.get(k1,0)
          if isknown <= known:
            del self.trigram[k1]
            current = self.trigram.get("__UNK",{})
            current.update(dict1)
            self.trigram["__UNK"] = current
          else:
            self.trigram[k1] = dict1
        # Quad Gram -----------------------------------
        for (k1, dict1) in list(self.quad_gram.items()):
          for (k2, dict2) in list(dict1.items()):
            for (k3, dict3) in list(dict2.items()):
              for (k4, val) in list(dict3.items()):
                # Next
                isknown = self.unigram.get(k4,0)
                if isknown <= known:
                  dict3["__UNK"] = dict3.get("__UNK",0) + val
                  del dict3[k4]
              # Next
              isknown=self.unigram.get(k3,0)
              if isknown <= known:
                del self.quad_gram[k1][k2][k3]
                current = self.quad_gram[k1][k2].get("__UNK", {})
                current.update(dict3)
                self.quad_gram[k1][k2]["__UNK"] = current
              else:
                self.quad_gram[k1][k2][k3] = dict3
            # Next
            isknown=self.unigram.get(k2,0)
            if isknown <= known:
              del self.quad_gram[k1][k2]
              current = self.quad_gram[k1].get("__UNK",{})
              current.update(dict2)
              self.quad_gram[k1]["__UNK"] = current
            else:
              self.quad_gram[k1][k2] = dict2
          # Next
          isknown=self.unigram.get(k1,0)
          if isknown <= known:
            del self.quad_gram[k1]
            current = self.quad_gram.get("__UNK", {})
            current.update(dict1)
            self.quad_gram["__UNK"] = current
          else:
            self.quad_gram[k1] = dict1
```

```python
    def _discount(self,discount=0.75):
        #discount each bigram count by a small fixed amount
        self.bigram={k:{kk:value-discount for (kk,value) in adict.items()}for (k,adict) in self.bigram
    .items()}

        #for each word, store the total amount of the discount so that the total is the same
        #i.e., so we are reserving this as probability mass
        for k in self.bigram.keys():
            lamb=len(self.bigram[k])
            self.bigram[k]["__DISCOUNT"]=lamb*discount

        #work out kneser-ney unigram probabilities
        #count the number of contexts each word has been seen in
        self.kn={}
        for (k,adict) in self.bigram.items():
            for kk in adict.keys():
                self.kn[kk]=self.kn.get(kk,0)+1



class question:

    """
    Question class which stores information about a singular MSR SCC question.

    Code adapted from the original work of  Dr. J Weeds, University of Sussex.

    Parameters
    ----------
    aline : str
        The training directory where training data can be found.
    files : list
        List of file names to be trained on.
    test_files : list
        List of file names for the model to be tested on.
    construct_params : dict
        Stores the parameters such as known to initialize the language model with.
    Attributes
    ----------
    trainingdir : str
        The training directory where training data can be found.
    files : list
        List of file names to be trained on.
    test_files : list
        List of file names for the model to be tested on.
    construct_params : dict
        Stores the parameters such as known to initialize the language model with.
    verbose : bool
        Whether or not method calls will print progress.
    unigram : dict
        Dictionary to store unigram probabilities.
    bigram : dict
        Dictionary to store bigram probabilities.
    trigram : dict
        Dictionary to store trigram probabilities.
    4-gram : dict
        Dictionary to store 4-gram probabilities.
    """


    def __init__(self,aline,stop=True):
        self.fields=aline
        self.num2letter = {
            0:"a",
            1:"b",
            2:"c",
            3:"d",
            4:"e"
            }
        self.stop = stop
        if self.stop:
          self.tokenized = [token.lower() for token in tokenize(self.fields[1]) if token.lower() not
    in stop_puncs]
          # self.tokenized = [wordnet_lemmatizer.lemmatize(token) for token in self.tokenized]
        else:
          self.tokenized = tokenize(self.fields[1])
        self.options = self.fields[2:7]
        self.backoff_factor = 0.4


    def get_field(self,field):
```

```
        return self.fields[question.colnames[field]]


    def add_answer(self,fields):
        self.answer=fields[1]


    def get_context(self,window,target="_____",method="left"):
      """
      Method to return the context of a target word in question sentence.
      If not sufficient context the method returns context with unknown token padding.
      """
      for i, token in enumerate(self.tokenized):
        if token == target:
            if method=="left":
              try:
                return self.tokenized[i-window:i]
              except:
                return ["__UNK"] * window
            elif method=="right":
              return self.tokenized[i+1:i+1+window]


    def chooseA(self):
        return("a")


    def random(self):
      """
      Retrun random choice of letter.
      """
      return random.choice(self.num2letter)


    def unigram(self):
      """
      Return position of word with greatest unigram probability. 0 otherwise.
      """
      option_probs = [lm.unigram[word] if word in lm.unigram else 0 for word in self.options]
      index = option_probs.index(max(option_probs))
      return self.num2letter[index]


    #
    #
    # The following bigram, trigram, 4-gram methods are for use in stupid backoff. See further below
    for individual methods.
    #
    #

    def bigram(self, context_dir="left"): # Backoff
      """
      Return position of word-pair with greatest bigram probability. 0 otherwise.
      """
      option_probs = []
      context = self.get_context(1, method=context_dir) # [0] to delist context.
      context = ["__UNK"] + context
      if context_dir == "left":
        for word in self.options:
          # Bigram.
          if context[-1] in lm.bigram and word in lm.bigram[context[-1]]:
            option_probs.append(lm.bigram[context[-1]][word])
          # Back off to unigram
          elif word in lm.unigram:
            option_probs.append(self.backoff_factor * lm.unigram[word])
          else:
            option_probs.append(0)
      elif context_dir == "right":
        option_probs = [lm.bigram[word][context] if word in lm.bigram and context in lm.bigram[word]
    else 0 for word in self.options]
      index = option_probs.index(max(option_probs))
      return self.num2letter[index]


    def trigram(self, context_dir="left"): # Backoff
      """
      Return position of word-group with greatest trigram probability. 0 otherwise.
      """
      option_probs = []
      context = self.get_context(2, method=context_dir)
      context = ["__UNK"] * 2 + context
```

```
    if context_dir == "left":
      for word in self.options:
        if context[-2] in lm.trigram and context[-1] in lm.trigram[context[-2]] and word in lm.trigram[context[-2]][context[-1]]:
          option_probs.append(lm.trigram[context[-2]][context[-1]][word])
        # Back off to bigram.
        elif context[-1] in lm.bigram and word in lm.bigram[context[-1]]:
          option_probs.append(self.backoff_factor * lm.bigram[context[-1]][word])
        # Back off to unigram.
        elif word in lm.unigram:
          option_probs.append(self.backoff_factor * self.backoff_factor * lm.unigram[word])
        # Else 0.
        else:
          option_probs.append(0)
    index = option_probs.index(max(option_probs))
    return self.num2letter[index]


  def quad_gram(self, context_dir="left"): # Backoff
    """
    Return position of word-group with greatest trigram probability. 0 otherwise.
    """
    option_probs = []
    context = self.get_context(3, method=context_dir)
    context = ["__UNK"] * 3 + context
    for word in self.options:
      if context[-3] in lm.quad_gram and context[-2] in lm.quad_gram[context[-3]] and context[-1] in lm.quad_gram[context[-3]][context[-2]] and word in lm.quad_gram[context[-3]][context[-2]][context[-1]]:
        option_probs.append(lm.quad_gram[context[-3]][context[-2]][context[-1]][word])
      # Back off to trigram.
      elif context[-2] in lm.trigram and context[-1] in lm.trigram[context[-2]] and word in lm.trigram[context[-2]][context[-1]]:
        option_probs.append(self.backoff_factor * lm.trigram[context[-2]][context[-1]][word])
      # Back off to bigram.
      elif context[-1] in lm.bigram and word in lm.bigram[context[-1]]:
        option_probs.append((self.backoff_factor**2) *lm.bigram[context[-1]][word])
      # Back off to unigram.
      elif word in lm.unigram:
        option_probs.append((self.backoff_factor**3) * lm.unigram[word])
      # Else 0.
      else:
        option_probs.append(0)
    index = option_probs.index(max(option_probs))
    return self.num2letter[index]


  def simple_4_gram(self, additional_args={}):
    """
    Method follows implementation in original MSR SCC publication.
    For each n-gram occurrence containing the target word, increment score
    by weighted value depending on n-gram.
    """
    lm = additional_args.get("n_gram_model")
    option_probs = []
    left_context = self.get_context(3, method="left")
    left_context = ["__UNK"] * 3 + left_context
    con1 = left_context[-3]
    con2 = left_context[-2]
    con3 = left_context[-1]
    right_context = self.get_context(3, method="right")
    right_context = right_context + ["__UNK"] * 3
    r_con1 = right_context[0]
    r_con2 = right_context[1]
    r_con3 = right_context[2]
    for word in self.options:
      try:
        score = 0
        # Bigram
        if word in lm.bigram.get(con3,{}):
          score += 1
        if r_con1 in lm.bigram.get(word,{}):
          score += 1
        # Trigram
        if word in lm.trigram.get(con3,{}).get(con2,{}):
          score += 2
        if r_con1 in lm.trigram.get(con3,{}).get(word,{}):
          score += 2
        if r_con2 in lm.trigram.get(word,{}).get(r_con1,{}):
          score += 2
        # Quad_gram
```

```python
          if word in lm.quad_gram.get(con1,{}).get(con2,{}).get(con3,{}):
            score += 3
          if r_con1 in lm.quad_gram.get(con2,{}).get(con3,{}).get(word,{}):
            score += 3
          if r_con2 in lm.quad_gram.get(con3,{}).get(word,{}).get(r_con1,{}):
            score += 3
          if r_con3 in lm.quad_gram.get(word,{}).get(r_con1,{}).get(r_con2,{}):
            score += 3
          option_probs.append(score)
        except TypeError:
          print([con1,con2,con3,word,r_con1,r_con2,r_con3])
          option_probs.append(0)
      # -------------
      # Ensemble
      if additional_args.get("ensemble", False):
        return option_probs
      # -------------
      index = option_probs.index(max(option_probs))
      return self.num2letter[index]


    def embedding_similarity(self, method="cos", additional_args={}):
      """
      For use with pretrained or even custom embeddings.
      """
      model = additional_args.get("pre_emb_model")
      option_probs = []
      # Remove target string.
      sentence = self.tokenized #.remove("_____")
      # Iterate through candidate choices.
      for word in self.options:
        try:
          # If no embedding for that word exists.
          if word not in model.wv:
            option_probs.append(0)
            # Continue to next candidate word.
            continue
          # Get vectorized form of word.
          word_vector = model.wv.get_vector(word)
          # Get vectorized form of sentence tokens.
          sentence_vectors = [model.wv.get_vector(sent_token) for sent_token in sentence if sent_token in model.wv and sent_token != "_____"]
          # For euclidean distances.
          if method == "euc":
            sim_score = [np.linalg.norm(model.wv.get_vector(word) - model.wv.get_vector(sent_token)) for sent_token in sentence if sent_token in model.wv and sent_token != "_____"]
          # For cosine distances.
          else:
            sim_score = model.wv.cosine_similarities(word_vector, sentence_vectors)
          # Append average "method" similarity.
          option_probs.append(sum(sim_score)/len(sim_score))
        except (TypeError, np.AxisError, ZeroDivisionError) as e:
          print(sentence)
          option_probs.append(0)
      # ----------------
      # Ensemble - cosine:
      if additional_args.get("ensemble", False):
        return option_probs
      # ----------------
      if method == "cos":
        index = option_probs.index(max(option_probs))
      else:
        index = option_probs.index(min(option_probs))
      return self.num2letter[index]


    def ensemble(self, additional_args={}):
      """
      Ensemble method which aggregates scores of both tested models and normalizes + sums them.
      """
      n_gram_model = additional_args.get("n_gram_model")
      n_gram = self.simple_4_gram(additional_args=additional_args)
      norm_n_gram = [float(i)/sum(n_gram) if sum(n_gram) !=0 else 0 for i in n_gram]

      pre_emb_model = additional_args.get("pre_emb_model")
      pre_emb = self.embedding_similarity(additional_args=additional_args)
      norm_pre_emb = [float(i)/sum(pre_emb) if sum(pre_emb) !=0 else 0 for i in pre_emb]

      option_probs = [sum(val) for val in zip(norm_n_gram, norm_pre_emb)]
      index = option_probs.index(max(option_probs))
      return self.num2letter[index]
```

```python


# ###################################################### comment below to use stupid backoff.
  def bigram(self, context_dir="left"):
    """
    Return position of word-pair with greatest bigram probability. 0 otherwise.
    """
    try:
      context = self.get_context(1, method=context_dir)[0] # [0] to delist context.
    except:
      context = ["__START"][0]
    if context_dir == "left":
      option_probs = [lm.bigram[context][word] if context in lm.bigram and word in lm.bigram[context] else 0 for word in self.options]
    elif context_dir == "right":
      option_probs = [lm.bigram[word][context] if word in lm.bigram and context in lm.bigram[word] else 0 for word in self.options]
    index = option_probs.index(max(option_probs))
    return self.num2letter[index]


  def trigram(self, context_dir="left"):
    """
    Return position of word-group with greatest trigram probability. 0 otherwise.
    """
    option_probs = []
    try:
      context = ["__UNK"] * 2 + self.get_context(2, method=context_dir)
    except:
      context = ["__UNK"] * 2 + context
    if context_dir == "left":
      for word in self.options:
        if context[-2] in lm.trigram and context[-1] in lm.trigram[context[-2]] and word in lm.trigram[context[-2]][context[-1]]:
          option_probs.append(lm.trigram[context[-2]][context[-1]][word])
        elif context[-2] in lm.trigram and context[-1] in lm.trigram[context[-2]] and "__UNK" in lm.trigram[context[-2]][context[-1]]:
          option_probs.append(lm.trigram[context[-2]][context[-1]]["__UNK"])
        # Else 0.
        else:
          option_probs.append(0)
    index = option_probs.index(max(option_probs))
    return self.num2letter[index]


  def quad_gram(self, context_dir="left"):
    """
    Return position of word-group with greatest trigram probability. 0 otherwise.
    """
    option_probs = []
    context = ["__UNK"] * 3 + self.get_context(3, method=context_dir)
    con_len = len(context)
    for word in self.options:
      if context[-3] in lm.quad_gram and context[-2] in lm.quad_gram[context[-3]] and context[-1] in lm.quad_gram[context[-3]][context[-2]] and word in lm.quad_gram[context[-3]][context[-2]][context[-1]]:
        option_probs.append(lm.quad_gram[context[-3]][context[-2]][context[-1]][word])
      elif context[-3] in lm.quad_gram and context[-2] in lm.quad_gram[context[-3]] and context[-1] in lm.quad_gram[context[-3]][context[-2]] and "__UNK" in lm.quad_gram[context[-3]][context[-2]][context[-1]]:
        option_probs.append(lm.quad_gram[context[-3]][context[-2]][context[-1]]["__UNK"])
      else:
        option_probs.append(0)
    index = option_probs.index(max(option_probs))
    return self.num2letter[index]

  def quad_gram(self, context_dir="left"):
    """
    Return position of word-group with greatest trigram probability. 0 otherwise.
    """
    option_probs = []
    context = ["__UNK"] * 3 + self.get_context(3, method=context_dir)
    con1, con2, con3 = context[-3], context[-2], context[-1]
    try:
      if con1 not in lm.quad_gram:
        con1 = "__UNK"
      if con2 not in lm.quad_gram[con1]:
        con2 = "__UNK"
      if con3 not in lm.quad_gram[con1][con2]:
        con3 = "__UNK"
      for word in self.options:
```

```
          if con1 in lm.quad_gram and con2 in lm.quad_gram[con1] and con3 in lm.quad_gram[con1][con2]
    and word in lm.quad_gram[con1][con2][con3]:
            option_probs.append(lm.quad_gram[con1][con2][con3][word])
          elif con1 in lm.quad_gram and con2 in lm.quad_gram[con1] and con3 in lm.quad_gram[con1][con2
    ] and "__UNK" in lm.quad_gram[con1][con2][con3]:
            option_probs.append(lm.quad_gram[con1][con2][con3]["__UNK"])
          else:
            option_probs.append(0)
      except KeyError:
        option_probs.append(0)
      index = option_probs.index(max(option_probs))
      return self.num2letter[index]
  # ###################################################


    def predict(self, method="chooseA", additional_args=None):
        if method=="chooseA":
          return self.chooseA()
        elif method=="random":
          return self.random()
        elif method=="unigram":
          return self.unigram()
        elif method=="bigram":
          return self.bigram()
        elif method=="trigram":
          return self.trigram()
        elif method=="quad_gram":
          return self.quad_gram()
        elif method=="simple_4_gram":
          return self.simple_4_gram(additional_args=additional_args)
        elif method=="embedding_similarity":
          return self.embedding_similarity(additional_args=additional_args)
        elif method=="ensemble":
          return self.ensemble(additional_args=additional_args)
        elif method == "cos":
          return self.embedding_similarity(additional_args=additional_args)
        elif method == "euc":
          return self.embedding_similarity(additional_args=args,method="euc")


    def predict_and_score(self, method="chooseA", additional_args=None):
        #compare prediction according to method with the correct answer
        #return 1 or 0 accordingly
        # Method also records which questions were answered correctly by index.
        prediction=self.predict(method=method, additional_args=additional_args)
        if prediction == self.answer:
            correct_answers.get(method).append(1)
            return 1
        else:
            correct_answers.get(method).append(0)
            return 0



class scc_reader:

    """
    Sentence completion challenge reader class. Used to read project training, testing files.

    Code adapted from the original work of  Dr. J Weeds, University of Sussex.

    Parameters
    ----------
    qs : str
        The question file path.
    ans : str
        The answers file path.
    stop : bool
        Dicates whether stopwords are removed in the Question class.
    Attributes
    ----------
    qs : str
        The question file path.
    ans : str
        The answers file path.
    stop : bool
        Dicates whether stopwords are removed in the Question class.
    """

    def __init__(self,qs=questions,ans=answers,stop=False):
        self.qs=qs
```

```
        self.ans=ans
        self.stop = stop
        self.read_files()


    def read_files(self):

        #read in the question file
        with open(self.qs) as instream:
            csvreader=csv.reader(instream)
            qlines=list(csvreader)

        #store the column names as a reverse index so they can be used to reference parts of the
    question
        question.colnames={item:i for i,item in enumerate(qlines[0])}

        #create a question instance for each line of the file (other than heading line)
        self.questions=[question(qline, self.stop) for qline in qlines[1:]]

        #read in the answer file
        with open(self.ans) as instream:
            csvreader=csv.reader(instream)
            alines=list(csvreader)

        #add answers to questions so predictions can be checked
        for q,aline in zip(self.questions,alines[1:]):
            q.add_answer(aline)


    def get_field(self,field):
        return [q.get_field(field) for q in self.questions]


    def predict(self,method="chooseA"):
        return [q.predict(method=method) for q in self.questions]

    def predict_and_score(self,method="chooseA", additional_args=None):
        scores=[q.predict_and_score(method=method, additional_args=additional_args) for q in self.
    questions]
        return sum(scores)/len(scores)







"""# Word Embedding methods

### Pre-trained
"""

# Note - uses gensim version 4.0.1
import gensim.downloader as api

fasttext_model300 = api.load('fasttext-wiki-news-subwords-300')
word2vec_model300 = api.load('word2vec-google-news-300')
glove_model300 = api.load('glove-wiki-gigaword-300')

def processfiles(max_files=10):
    """
    Code adapted from n_gram_language_model class definition.
    Returns lists of preprocessed and not tokenized sentences.
    """
    sentences = []
    sentences_preprocessed = []
    for afile in tqdm.tqdm(training):
        # print("Processing {}".format(afile))
        try:
            with open(os.path.join(trainingdir,afile)) as instream:
                for line in instream:
                    line=line.rstrip()
                    if len(line)>0:
                        tokens = [token for token in tokenize(line) if token not in stop_puncs]
                        sentences_preprocessed.append(tokens)
                        tokens = [token for token in tokenize(line)]
                        sentences.append(tokens)
        except UnicodeDecodeError:
            print("\nUnicodeDecodeError processing {}: ignoring rest of file".format(afile))
    return sentences, sentences_preprocessed

```

```python
# Code to create gensim models to be trained on gutenberg text.
from gensim.models import Word2Vec, FastText

def create_word2vec_gensim(sentences, window=5, vector_size=100, skip_gram=1, negative=5, alpha=0.05,
    epochs=15, seed=1):
  return Word2Vec(sentences,
                  window = window,
                  vector_size = vector_size,
                  sg = skip_gram,
                  negative = negative,
                  alpha = alpha,
                  epochs = epochs,
                  seed = seed)


def create_fasttext_gensim(sentences, window=5, vector_size=100, skip_gram=1, negative=5, alpha=0.05,
    epochs=15, min_n=3, max_n=6, seed=1):
  return FastText(sentences,
                  window = window,
                  vector_size = vector_size,
                  sg = skip_gram,
                  negative = negative,
                  alpha = alpha,
                  epochs = epochs,
                  min_n = min_n,
                  max_n = max_n,
                  seed = seed)

"""# Ensemble

### N-gram
"""

construct_params = {
    "known" : 5,
    "verbose" : False,
    "remove_stopwords" : True
}

# Initialize n-gram language model.
lm=n_gram_language_model(trainingdir=trainingdir,files=training, test_files=[], construct_params=
    construct_params)

additional_args = {"n_gram_model" : lm, "pre_emb_model" : fasttext_model300, "ensemble" : True}

"""# Questions and Answers

"""

import pandas as pd, csv
questions=os.path.join(parentdir,"testing_data.csv")
answers=os.path.join(parentdir,"test_answer.csv")

with open(questions) as instream:
    csvreader=csv.reader(instream)
    lines=list(csvreader)
qs_df=pd.DataFrame(lines[1:],columns=lines[0])



"""## Ensemble"""

correct_answers = {"simple_4_gram" : [], "embedding_similarity" : [], "ensemble" : []}

# Simple-4-gram score
additional_args = {"n_gram_model" : lm, "pre_emb_model" : fasttext_model300, "ensemble" : False}
SCC = scc_reader(questions, answers, stop=True)
SCC.predict_and_score(method="simple_4_gram", additional_args=additional_args)

# Embedding similarity score
SCC = scc_reader(questions, answers, stop=True)
SCC.predict_and_score(method="embedding_similarity", additional_args=additional_args)

# Ensemble score
additional_args["ensemble"] = True
SCC = scc_reader(questions, answers, stop=True)
SCC.predict_and_score(method="ensemble", additional_args=additional_args)

"""## Error Analysis"""

import warnings
```

```
warnings.filterwarnings(action='ignore') #,category=DeprecationWarning,module='gensim')

#read in the answer file
holding_list = []
with open("lab2resources/sentence-completion/test_answer.csv") as instream:
    csvreader=csv.reader(instream)
    alines=list(csvreader)
    holding_list.append(alines)
answers = [holding_list[0][i][1] for i, _ in enumerate(holding_list[0]) if i != 0]

# All correct.
correct = []
# All incorrect.
incorrect = []
# N-gram incorrect
n_g_incorrect = []
# n-g over emb.
n_g_vs_emb = []
# emb over n-g.
emb_vs_n_g = []
# ensemble got right when other two didnt.
ensemble_solved = []
# Ensemble favouring n-gram.
n_g_favour = []
# Ensemble favouring word embeddings.
emb_favour = []
# Ensemble only.
ensemble_only = []

for i, (n_g, emb, ensemble) in enumerate(zip(correct_answers["simple_4_gram"], correct_answers["embedding_similarity"], correct_answers["ensemble"])):
  if n_g == emb == ensemble == 1:
    correct.append(i)
  if n_g == emb == ensemble == 0:
    incorrect.append(i)
  if n_g == 1 and emb == 0:
    n_g_vs_emb.append(i)
  if n_g == 0 and emb == 1:
    emb_vs_n_g.append(i)
  if n_g == emb == 0 and ensemble == 1:
    ensemble_solved.append(i)
  if n_g == 0:
    n_g_incorrect.append(i)

  if n_g == 0 and ensemble == 1 == emb:
    emb_favour.append(i)
  if emb == 0 and ensemble == 1 == n_g:
    n_g_favour.append(i)

  if ensemble == 1:
    ensemble_only.append(i)

def error_analysis(index):
  """
  Function to print out the answers predicted by each model stated, as well as the question (tokenized and not) and the correct answer.
  """

  q = question([SCC.get_field("id")[index], SCC.get_field("question")[index], SCC.get_field("a)")[index], SCC.get_field("b)")[index],SCC.get_field("c)")[index], SCC.get_field("d)")[index], SCC.get_field("e)")[index]])
  answer = answers[index]

  additional_args = {"n_gram_model" : lm, "pre_emb_model" : fasttext_model300, "ensemble" : False}
  pred_ngram = q.predict("simple_4_gram", additional_args=additional_args)
  answer_ngram = SCC.get_field("{})".format(pred_ngram))[index]

  pred_emb = q.predict("embedding_similarity", additional_args=additional_args)
  answer_emb = SCC.get_field("{})".format(pred_emb))[index]

  additional_args["ensemble"] = True
  pred_ensemble = q.predict("ensemble", additional_args=additional_args)
  answer_ensemble = SCC.get_field("{})".format(pred_ensemble))[index]

  answer_correct = SCC.get_field("{})".format(answers[index]))[index]

  # print()
  print("------------------------------")
  print(SCC.get_field("question")[index])
  print(q.tokenized)
  print()
```

```
  print(answer_correct)
  print()
  print("N-gram: {}".format(answer_ngram))
  print("Embedd: {}".format(answer_emb))
  print("Ensemb: {}".format(answer_ensemble))
  print("------------------------------")
  print()

[error_analysis(index) for index in ensemble_solved]

"""# Development + graphing functions"""

# Example development code for experimenting with the known parameter.
lm_known = {}
MAX_FILES=100

training_shuffled = shuffle(training)
training_shuffled = training_shuffled[:MAX_FILES]

num = 0.2
train, test = train_test_split(training_shuffled,test_size=num)

for known in [5, 10, 25, 50]:


  construct_params = {
      "known" : known,
      "verbose" : False,
      "remove_stopwords" : True
  }

  # Initialize n-gram language model.
  lm=n_gram_language_model(trainingdir=trainingdir,files=train, test_files=test, construct_params=
    construct_params)

  lm_known[known] = lm

knowns_test = [lm.compute_perplexity(filenames=lm.test_files,methodparams={"method":method}) for lm in
     lm_stop.values() for method in ["unigram", "bigram", "trigram", "quad_gram"]]

def chunker(lst, n):
    """
    Chunnk lst (list) into n chunks.
    """
    for i in range(0, len(lst), n):
        yield lst[i:i + n]

vocab_size = [len(lm.unigram) for lm in lm_stop.values()]

one = []
two = []
three = []
four = []
for i_l in list(chunker(knowns_test,4)):
  one.append(i_l[0])
  two.append(i_l[1])
  three.append(i_l[2])
  four.append(i_l[3])

data_preproc = pd.DataFrame({
    'Training Doc Size': [key for key, lm in lm_stop.items()],
    'unigram': one,
    'bigram': two,
    "trigram":three,
    '4-gram': four,
    'vocab': vocab_size,
    })

data_preproc

# Graphing function.
ax = sns.lineplot(x='knowns', y='value', hue='variable',
             data=pd.melt(data_preproc, ['knowns']))

ax.set(xlabel="Known Parameter Value", ylabel="Perplexity")
# ax._legend.set_title("N-Gram")
leg = ax.legend()
leg.set_title("N-Gram")

import seaborn as sns
sns.set_theme(style="whitegrid")
```

```

keys = [p[0] for p in gg]
val = [p[1] for p in gg]

ax = sns.barplot(y=keys, x=val, orient="horizontal")
ax.set(xlabel="Frequency", ylabel="Word Token")


## ---- Hypothesis Testing - Binomial ---- ##
from scipy import stats
# 464 successfully answered questions, total of 1040 sentences, probability due to random chance=0.2.
    (one tailed test)
stats.binom_test(464, n=1040, p=0.2, alternative="greater")
```