Language Modelling Project

Amol Sharma

25th January 2019

To explore language models, unigram, bigram and trigram models were created in `ngram_no_smooth.java` based on a subset of the One Billion Word Language Modeling Benchmark [1]. Then, a smoothed model was created in `ngram_int_smooth.java` using linear interpolation. A neural model was also implemented using AllenNLP [2]. The perplexity of all models was compared and it was found that the n-gram model with smoothing performed best on held-out data.

1 N-Gram Language Modelling

Description of the Model

My process heavily depends on the order that the models are created. In the code, unigram, bigram, and trigram models are developed as hashmaps in that order. This allows us to store a word as a key, and its occurrence count as the value, and retrieve it quickly. For each model, we parse through an unedited training file, considering each word to be a token and adding an artificial **STOP** token once we reach the end of a line. We keep track of the last 2 words in order to be able to make bigram and trigram models for the current word.

Once the unigram model has been created, we iterate over its hashmap and UNK any word that were seen less than 3 times in training. The words that were UNKed are remembered in a set in order to be able to UNK the same ones for the bigram and trigram models in training as well. A count for the total number of tokens in the training file is also maintained.

Experimental Procedure

After each model is trained, it is tested against the training, dev, and test sets and evaluated based on perplexity. In testing, files are read in the same way as in training where the last 2 words are remembered. If we read a word in the file that is not in our unigram model, it means that we UNKed it or did not see it in training, thus the word is replaced with an UNK token. For the unigram model, we can get a probability for each word. However, for the bigram model, we only get the probability once we have one word in the history (or 2 words in history for trigrams). This means that in 'I saw a deer', 'I saw' would be the first bigram whose probability is evaluated. To calculate the p_x or θ element of perplexity for each word, the following methods were used:

Unigram # times token was seen in training divided by the number of tokens in training data

Bigram # times token was seen followed by last token divided by unigram count of last token

Trigram # times token was seen followed by last 2 tokens divided by bigram count of last 2 tokens

After getting θ, the perplexity is calculated based on the operations described in lecture. The results are shown below:

	Unigram	Bigram	Trigram
Training	976.544	77.073	7.857
Dev	892.247	∞	∞
Test	896.499	∞	∞

From this table, we see that perplexity is only comparable between these models over the training set. The bigram and trigram had infinite perplexity over held out data without smoothing likely because the model saw new bigrams and trigrams even after UNKing. For example, if we did not see a bigram in training that was ⟨UNK⟩ ⟨UNK⟩, but happen to come across two unknown words in a sequence, then we try to get the count of ⟨UNK⟩ ⟨UNK⟩ and get 0. If θ is 0, when we try to take the log of it, we will get ∞. The trigram example is even clearer if we consider the situation where we have ⟨UNK⟩ ⟨UNK⟩ ⟨The⟩ but ⟨UNK⟩ ⟨UNK⟩ is not in the bigram model. Then, we will be dividing by 0. In essence, seeing unseen bi/trigrams when we don't rely on smoothing makes our model infinitely perplexed.

Comparing over the training set, we see that perplexity decreases by an order of magnitude as the n in the n-gram increases. This makes sense intuitively, since we know that our annunciations depend on the words we spoke earlier in the same sentence.

2 Smoothing

Description of the Model

To improve on the models, interpolation smoothing was used between the unigram, bigram and trigram models. All 3 models were trained, and the probabilities from their outputs were weighted for each word. This model also needed to include ⟨START⟩ tokens to be able to make bi/trigrams for the first words in the lines. The unigram model was first constructed on its own, without ⟨START⟩. Then in a copy of the unigram model, counts for the ⟨START⟩ tokens were added. I included only one ⟨START⟩ per line in the copy of the unigram model because we would never be considering the bigram ⟨START⟩⟨START⟩. When we calculate bigram probabilities, we would only be looking at ⟨START⟩ ⟨WORD 1⟩. Then, words seen less than 3 times by the unigram model were UNKed and remembered for UNKing in bi/trigrams. The bigram and trigram models were then built, and a count for ⟨START⟩ ⟨START⟩ was added to a copy of the bigram model. When calculating θ_{x_j}, the original unigram model was used. But for $\theta_{x_j|x_{j-1}}$ (the bigram), the copy of the unigram model which included ⟨START⟩ was used for the denominator. This extends to the trigram model for which the copy of the bigram was used for the counts in the denominator.

Another aspect to consider was what happens when one of the n-grams is invalid. For unigrams, we will always have a (non-zero) result as the word we read and check the probability for will always be in the hashmap (as the word or UNK). For bigrams, it is possible for us to get a zero probability (hence the ∞ result in part 1 for held out data), but we never compute a mathematically incorrect expression. This is because we divide a number ≥ 0 by a positive number which comes

from the counts of the unigram. However, for trigrams, when we divide by the count of a bigram, it is possible that the bigram was never seen and its count is 0. Therefore, we would be trying to divide by 0. To fix this problem with held out data, I redistribute the weight λ_3 to make sure that $\lambda_1 + \lambda_2 = 1$ and the trigram gets ignored. I did this by adding $\frac{1}{2}\lambda_3$ to each of the other two λs and setting $\lambda_3 = 0$.

Experimental Procedure

The perplexity for this smoothed model was calculated in a very similar way to part 1. The big change was that p_x or θ was now a sum of the weighted θs from each of the n-gram models.

To find the best values for λ_1, λ_2 and λ_3, a grid search was used. I simply tried a few different values, optimising for low perplexity in the dev set. The optimisation was for the dev set so that the model did not overfit the training data. The test set could have also been used for hyperparameter tuning, but then the tests would have been performed on the dev set.

Perplexity Scores

The following table shows the different λ values tried and the resulting perplexities:

Trial	λ_1	λ_2	λ_3	Training Perplexity	Dev Perplexity	Test Perplexity
1	0.3	0.35	0.35	13.551	173.372	172.864
2	0.2	0.6	0.2	17.228	170.347	169.860
3	0.2	0.2	0.6	9.691	194.188	193.412
4	0.6	0.3	0.1	29.614	193.282	193.149
5	0.2	0.4	0.4	12.144	173.772	173.157
0	0.1	0.3	0.6	9.394	194.484	193.574

From the above, we see that the highest dev perplexities occur with the greatest weights for λ_3 (specifically when it is 0.6 in these cases). Swapping λ_3 and λ_1 has similar perplexity as well, meaning that more balanced weights may be more useful. Then, after λ_2 is assigned one of the larger weights, the perplexity seems to decrease as λ_3 decreases.

Chosen Hyperparameters

The hyperparameters chosen from the above experiment are $\lambda_1 = 0.2$, $\lambda_2 = 0.6$ and $\lambda_3 = 0.2$. I think that these parameters seemed to perform the best because of multiple reasons. Firstly, the unigram model was given a relatively low weight. I would expected longer n-grams to be less perplexed as they are conditioned on a history. However, I think giving λ_3 too much weight was not beneficial because of the dividing-by-zero problem mentioned earlier. To fix the problem, I had decided to split λ_3 evenly between λ_1 and λ_2. However, this makes λ_1 a relatively bigger weight than it was before if we do need to split λ_3. If there were many trigrams with unseen bigram histories, then essentially λ_1 was becoming a bigger weight than it should have been. Therefore, perhaps the model could be improved further by splitting λ_3 unevenly and weighting λ_2 more heavily in the split. This is corroborated by the fact that trigrams actually were helping in the training data,

where the lowest perplexity corresponded to the greatest λ_3, but is less helpful for held out data sets like dev and test.

The perplexity for the test set using the specified chosen hyperparameters was 169.86030067326013.

Half the Training Data

If only half of the training data is used, I think that the perplexity for previously unseen data would generally decrease because fewer words would appear more than 3 times, and therefore more words would get UNKed. The table below shows a comparison for the smoothed model:

	Training Perplexity	Dev Perplexity	Test Perplexity
All Training Data	17.228	170.347	169.860
Half Training Data	48.599	162.854	162.981

From the table, we can see that the perplexity of the held out data in both dev and test decreases when we use half the training data. To check my reasoning, I got the count of how many unique words were UNKed in both experiments. This decreased from 54,059 to 37,392, so the UNKed words actually decreased and my original hypothesis was wrong. However, I now realise that that metric is irrelevant. With half the data, I found that the new vocabulary size was 17,537 instead of 26,602. This means that although less words were UNKed in training, the model also knows less words. When it comes across new words in held out data, it UNKs more of those words because it concretely knows the counts of fewer words. Therefore, even though there is a lower count of UNK, the probability of UNK is used much more often, and there are fewer words the model gets perplexed by.

UNKing with Different Occurrences

Similarly to the last problem, if more words are UNKed, then the vocabulary size will be smaller and the model will know the counts of fewer words. Therefore, coming across new words in held out data, the model will not be able to look anywhere except for UNK. Since UNK really represents many words, it will represent a higher probability than that one word should, giving it an inflated value for θ. Then, perplexity which is $2^{-f(\theta)}$, is reduced. Intuitively, there are simply less words that the model knows, so there are less words it can be confused between. A quick test indicated that test data perplexity was 189.7365 when UNKing words that appeared once, and 144.9257 when UNKing words that appeared less than 5 times, corroborating the hypothesis.

3 Neural Language Modelling

This model for AllenNLP built off of the starter config file provided. The starter file contained start and stop tokens, and also UNKed words occurring less than 3 times. This kept the same vocabulary size which was incredibly important as seen in the 2 parts above. I used a default trainer, an LSTM contextualizer, and an embedding dimension of 32. As the program did not complain, I stuck with my configuration. To calculate perplexity, I am reporting exp(loss) where loss was a value provided by AllenNLP. The following are the perplexity results from the neural language model:

	Training Perplexity	Dev Perplexity	Test Perplexity
Loss	3.808	5.703	5.693
Perplexity	45.069	299.835	296.678

This result is somewhat surprising to me. The training perplexity is only marginally lower than the n-gram model's while the dev and test perplexity are much higher on the neural model. I had expected the neural language model to have a lower perplexity than the n-gram models. Perhaps there was an error in my configuration file and I need to run more experiments. Another reason could be that the model needs more data to recognise features more accurately from. Or perhaps that my embedding size was too large and there were not actually that many features to be found in the corpus so the data overfit to the training set. That would explain why the training perplexity is lower than n-grams but held out data's is higher.

References

[1] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, and Phillipp Koehn. "One billion word benchmark for measuring progress in statistical language modeling". In: *INTERSPEECH*. 2014.

[2] Matt Gardner, Joel Grus, Mark Neumann, Oyvind Tafjord, Pradeep Dasigi, Nelson F. Liu, Matthew Peters, Michael Schmitz, and Luke Zettlemoyer. "AllenNLP: A deep semantic natural language processing platform". In: *NLP-OSS*. 2018.