UNIVERSITEIT•STELLENBOSCH•UNIVERSITY
jou kennisvennoot • your knowledge partner

Neural networks for language model smoothing

Werner Van der Merwe
20076223

Report submitted in partial fulfilment of the requirements of the module Project (E) 448 for the degree Baccalaureus in Engineering in the Department of Electrical and Electronic Engineering at Stellenbosch University.

Supervisor: Prof. T. R. Niesler

November 2019

Acknowledgements

Declaration

I, the undersigned, hereby declare that the work contained in this report is my own original work unless otherwise stated.

Signature: .....
Werner Van der Merwe

Date: .....

Abstract

Contents

List of Figures

List of Tables

Nomenclature

Variables and functions

$p(x)$	Probability density function with respect to variable x.
$P(A)$	Probability of event A occurring.
$N(A)$	Number of times event A occurred.
$E(x)$	Expectation of random variable x.

Acronyms and abbreviations

LM	Language Model
PP	Perplexity
NN	Neural Network
MLE	Maximum Likelihood Estimation
MSE	Mean Square Error
MAE	Mean Absolute Error
SGD	Stochastic Gradient Descent
SRILM	SRI Language Modeleling toolkit
GPU	Graphics Processing Unit

Chapter 1

Introduction

Chapter 2

Literature Study

2.1. The Language Model

2.1.1. The Need for the Statistical Language Model

Natural languages spoken by people are unlike programming languages, they are not designed, but rather emerge naturally and tend to change over time. Although they are based on a set of grammatical rules, spontaneous speech often deviates from it. Because of this, trying to program machines to interpret languages can be difficult. Rather than trying to model formal grammars and structures of a language, we find more success in basing them on statistical models. This is far less complex and still allows machines to interpret certain ambiguities, occurring often under natural circumstances.

One such ambiguity can be explained by an example. If a machine's purpose was to translate a user's speech to text, it could encounter the following problem; having to determine which of two words, that share acoustic characteristics, were said by the user. Such as "what do you see?" vs "what do you sea?". For the machine to interpret this correctly, we make use of a statistical language model. These models succeed because words do not appear in random order i.e. their neighbours provide much-needed context to them.

A statistical language model, in short, simply assigns a probability to a sequence of words. By doing so, the machine is able to choose the correct sequence of words based on their probabilities. To be clear, the language model does not assign a probability based on acoustic data, but rather the possible word sequences themselves.

2.1.2. Mathematical Preliminaries

The probability of a sequence of words can be represented as follows:

$$P(\mathbf{w}(0, L-1)) \tag{2.1}$$

with $\mathbf{w}(0, L-1) = w(0), w(1), ..., w(L-1)$ representing a sequence of L words. This probability is used by the machine in the speech-to-text example to successfully distinguish between the ambiguous words and choose the correct sequence. Much like a human uses the context of the given words.

The joint probability in Equation (2.1) can be decomposed, using the definition of conditional probabilities, into a product of conditional probabilities:

$$P(\mathbf{w}(0, L-1) = \prod_{i=0}^{L-1} P(w(i)|\mathbf{w}(0, i-1)) \tag{2.2}$$

Therefore the probability of observing the i^{th} word is dependent on knowing the preceding $i-1$ words. To reduce complexity, we approximate Equation (2.2) to only consider the preceding $n-1$ words:

$$\prod_{i=0}^{L-1} P(w(i)|\mathbf{w}(0, i-1)) \approx \prod_{i=0}^{L-1} P(w(i)|\mathbf{w}(i-n+1, i-1)) \tag{2.3}$$

We have now derived the basis for the n-gram language model using the **Markov** assumption. Markov models are described as probabilistic models that predict some future unit without looking too far into the past [2].

2.1.3. N-Gram Language Model

In this report, we will only be focusing on one type of language model, the n-gram model. The n-gram model has several reasons for its success in becoming one of the preferred models. It considered as being computationally efficient and simple to implement [3].

The next step would be to determine the value of n. This is also referred to as the tuple size. Ideally, we would want a large tuple size, because words could still strongly influence others far down the sequence. However, the number of parameters that need to be estimated grows exponentially as tuple size increases. If we take a vocabulary size of 20,000 words, which is being modest, Table 2.1 shows the number of parameters to be estimated for each tuple size [4].

Table 2.1: Increase of parameters for n-gram size

Tuple Size	Parameters
2 (bigram)	$20,000^2 = 4 \times 10^8$
3 (trigram)	$20,000^3 = 8 \times 10^{12}$
4 (four-gram)	$20,000^4 = 16 \times 10^{16}$
5 (five-gram)	$20,000^5 = 32 \times 10^{20}$

A five-gram model, although having more information available for potentially better estimation of the probabilities, is not computationally practical. Four-gram models are considered to be barely feasible. Models tend mostly to incorporate bigrams and trigrams [4].

2.1.4. Estimating N-gram Probabilities

The simplest way to estimate the probability is with a Maximum Likelihood Estimation, known as MLE. To obtain this estimate, we observe the n-gram counts from a training corpus. These counts are thereafter normalized to give [2]:

$$P_{rf}(w_n|w_1...w_{n-1}) = \frac{N(w_1...w_n)}{\sum_{\forall k} N(w_1...w_{n-1}, w_k)} \tag{2.4}$$

This equation can be further simplified to yield Equation (2.5) as the count of a given prefix is equal to the sum of all n-gram counts containing that same prefix.

$$P_{rf}(w_n|w_1...w_{n-1}) = \frac{N(w_1...w_n)}{N(w_1...w_{n-1})} \tag{2.5}$$

We now have a value that lies between 0 and 1, and will refer to this probability estimate as the relative frequency estimate.

A new problem arises because our training set and validation set have not necessarily seen the same n-grams. Even if we increase the size of the training set, the majority of words are still considered to be uncommon and n-grams containing these words are rare [4]. The validation set will most likely contain several unseen n-grams whose relative frequencies according to Equation (2.5) is considered to be 0.

The probability of occurrence for a sequence of words should never be 0, however since all legitimate word sequences will never be seen in a practical training set, some form of regularization is needed to prevent overfitting of the training data. This is where smoothing

is applied. Smoothing decreases the probability of seen n-grams and assigns this newly acquired probability mass to unseen n-grams [4]. The next section will discuss different kinds of smoothing and how they are implemented.

2.2. Smoothing

2.2.1. Good-Turing

This smoothing technique was published by I. J. Good in 1953, who credits Alan Turing with the original idea [5]. It is based on the assumption that the frequency for each, in our case, n-gram follows a binomial distribution [2]. Specifically, n-grams occurring the same number of times are considered to have the same probability estimate.

We begin with the Good-Turing probability estimate:

$$P_{GT} = \frac{r^*}{N_\sum} \tag{2.6}$$

Where $N_\sum$ is the total number of n-grams and r^* is a re-estimated frequency formulated as follows [4]:

$$r^* = (r+1)\frac{E(C_{r+1})}{E(C_r)} \tag{2.7}$$

In Equation (2.7) C_r refers to the count of n-grams that occur exactly r times in the training data, and $E(\cdot)$ refers to the expectation. By approximating the expectations as counts and combining Equations (2.6) and (2.7) we obtain:

$$q_r = \frac{(r+1)\cdot C_{r+1}}{N_\sum \cdot C_r} \tag{2.8}$$

In Equation (2.8), q_r denotes the Good-Turing estimate for the probability of occurrence of an n-gram occurring r times in the training data. For unseen n-grams, $r = 0$, giving us:

$$C_0 \cdot q_0 = \frac{C_1}{N_\sum} \tag{2.9}$$

and where $C_0 \cdot q_0$ can be considered as the probability of occurrence of any unseen n-gram.

The Good-Turing technique can be applied to a language model, in order to provide nonzero probability estimates for unseen n-grams. However, this technique has two concerns.

One is that our assumption in Equation (2.8) is only viable for values of $r < k$ (where k is a threshold, typically a value ranging from 5 to 10 [2, 4]). The MLE estimate of high

frequency words is accurate enough that they do need smoothing.

The other concern is that it is quite possible for C_r to equal 0 for some value of r. This would result in a probability estimate of zero, leading us back to our initial problem. To remedy this, one can smooth the values for C_r, replacing any zeroes with posisitive estimates before calculating the probability.

2.2.2. Backing-Off

Instead of redistributing the probability mass equally amongst unseen n-grams, S. M. Katz suggests an alternative. His solution, introduced in 1987 [2], assigns a nonzero probability estimate to unseen n-grams, by *backing off* to the $(n-1)$-gram. This backoff process continues until a n-gram with a nonzero count is observed. By doing so, the model is able to provide a better probability estimate for an unseen n-gram.

In Katz's backoff model, a discounting factor reserves the probability mass for unseen n-grams. This discounting factor could be implemented in the form of absolute discounting, linear discounting or other suitable estimators. The backoff procedure dictates the redistribution of this probability mass amongst unseen n-grams [2].

We will be combining Katz's backoff model with Good-Turing discounting. Our combined model is described as follows [6]:

$$P_{bo}(w_n|w_1...w_{n-1}) = \begin{cases} P^*(w_n|w_1...w_{n-1}) & \text{if} \quad N(w_1...w_{n-1}) > 0 \\ \alpha(w_1...w_{n-1}) \cdot P_{bo}(w_n|w_2...w_{n-1}) & \text{if} \quad N(w_1...w_{n-1}) = 0 \end{cases} \tag{2.10}$$

where P^* is the Good-Turing discounted probability estimate and $\alpha(w_1...w_{n-1})$ is the backoff weight introduced by Katz. The second case in Equation (2.10) shows how the procedure recursively backs off for unseen n-grams, thereby obtaining a nonzero probability estimate normalised by the backoff weight.

It is important to use discounted probability estimates to ensure that there is probability mass to distribute among unseen n-grams during backoff. To ensure that Equation (2.10) produces true probabilities, the backoff weight is chosen to satisfy the following requirement [2];

$$\sum_{\forall k} P_{bo}(w_k|w_1...w_{n-1}) = 1 \tag{2.11}$$

Equation (2.11) ensures that the probability estimate $P_{bo}(\cdot)$ is correctly normalised. In Equation (2.11) k represents the total number of words within a specified vocabulary.

To derive a the value for $\alpha(w_1...w_{n-1})$, we start by defining $\beta(w_1...w_{n-1})$ as the harvested probability mass obtained during discounting. β is calculated by subtracting from 1 the total discounted probability mass for all n-grams, seen in our training set, sharing the same prefix [6]:

$$\beta(w_1...w_{n-1}) = 1 - \sum_{\forall k:r>0} P^*(w_k|w_1...w_{n-1}) \tag{2.12}$$

with r indicating the exact number of occurrence for the n-gram in the training set.

We normalise Equation (2.12) to ensure each $(n-1)$-gram only receives a fraction of the total mass. The normalising factor is calculated by summing the probability estimate of all $(n-1)$-grams sharing the prefix as the unseen n-gram [6]:

$$\gamma(w_1...w_{n-1}) = \sum_{\forall k:r=0} P^*(w_k|w_2...w_{n-1}) \tag{2.13}$$

Which is more conveniently expressed as:

$$\gamma(w_1...w_{n-1}) = 1 - \sum_{\forall k:r>0} P^*(w_k|w_2...w_{n-1}) \tag{2.14}$$

where r represents the exact number of occurrence for the original n-gram (not to be confused with the $(n-1)$-gram) in the training set. Combining Equations (2.12) and (2.13) we find:

$$\alpha(w_1...w_{n-1}) = \frac{1 - \sum_{\forall k:r>0} P^*(w_k|w_1...w_{n-1})}{1 - \sum_{\forall k:r>0} P^*(w_k|w_2...w_{n-1})} \tag{2.15}$$

In our case, we substitute values for n in Equations (2.10) and (2.15), yielding our bigram and trigram models:

$$P_{bo}(w_2|w_1) = \begin{cases} P(w_2|w_1) & \text{if} \quad N(w_1,w_2) > 0 \\ \alpha(w_1) \cdot P_{MLE}(w_2) & \text{if} \quad N(w_1,w_2) = 0 \end{cases} \tag{2.16}$$

with

$$\alpha(w_1) = \frac{1 - \sum_{\forall k:r>0} P^*(w_k|w_1)}{1 - \sum_{\forall k:r>0} P_{MLE}(w_k)} \tag{2.17}$$

and

$$P_{bo}(w_3|w_1,w_2) = \begin{cases} P^*(w_3|w_1,w_2) & \text{if } N(w_1,w_2,w_3) > 0 \\ \alpha(w_1,w_2) \cdot P_{bo}(w_3|w_2) & \text{else if } N(w_2,w_3) > 0 \\ \alpha(w_1,w_2) \cdot \alpha(w_2) \cdot P_{MLE}(w_3) & \text{otherwise.} \end{cases} \tag{2.18}$$

with

$$\alpha(w_1,w_2) = \frac{1 - \sum_{\forall k:r>0} P^*(w_k|w_1,w_2)}{1 - \sum_{\forall k:r>0} P^*(w_k|w_2)} \tag{2.19}$$

Where $P_{MLE}(w)$ represents a MLE of that unigram i.e. $N(w)$ divided by the total count of unigrams in the corpus [6]. The last case in Equation (2.18) indicates a further backoff to the unigram if the bigram is also unseen.

Backoff models do exhibit a flaw under certain circumstances. Take the trigrams $w_iw_jw_k$ for example. The bigram w_iw_j and unigram w_k could both have high counts in our training set, however there could be a particular reason, possibly grammatical, for these words to not appear as a trigram. The backoff model, being very simple, is not capable of capturing this and will assign a probability estimate to the trigram that is most likely too high. Despite this, backoff models have proven to work well in practise [2, 4].

2.3. Perplexity

The unit often used in measuring the quality of a language model (LM), is perplexity. By definition, "perplexity" indicates the inability to understand something. For an LM, perplexity shows the degree of uncertainty the LM experiences in predicting a sequence of words.

Perplexity, referred to as PP, is defined by the following equation:

$$PP = [P(\mathbf{w}(0, L-1))]^{-\frac{1}{L}} \tag{2.20}$$

with $P(\mathbf{w}(0, L-1))$ representing the probability of a sequence of words, L long. Perplexity is the reciprocal per-word geometric mean probability of the given sequence [3]. In layman's terms, perplexity is the number of possibilities the LM considered for each word. The lower the perplexity, the more accurate the LM assigns probabilities to n-grams.

Decomposing perplexity, using Equation (2.2), we find a less computationally demanding formula:

$$\log(PP) = -\frac{1}{L}\sum_{i=0}^{L-1}\log[P(w(i)|\mathbf{w}(0, i-1))] \tag{2.21}$$

Substituting the LM probability estimates into the right-hand side of Equation (2.21), we obtain a perplexity value for the LM on the given sequence of words.

2.4. Feedforward Neural Networks

Neural networks, modelled after the human brain, consists of multiple layers connected via mathematical algorithms [1]. Each layer, comprising nodes that emulate biological neurons, is responsible for processing the inputs and finding correlations between them. In this report, we focus on implementing the neural network (NN) to perform regression. Specifically, taking in a set of input features and predicting an appropriate output.

In a feedforward NN, each node's output is passed on to the following layer. What makes this unique is that data only moves in one direction. Feedforward NNs have proven to work well in regression related tasks [1]. This type of NN has the benefit of being effective yet simple to implement.

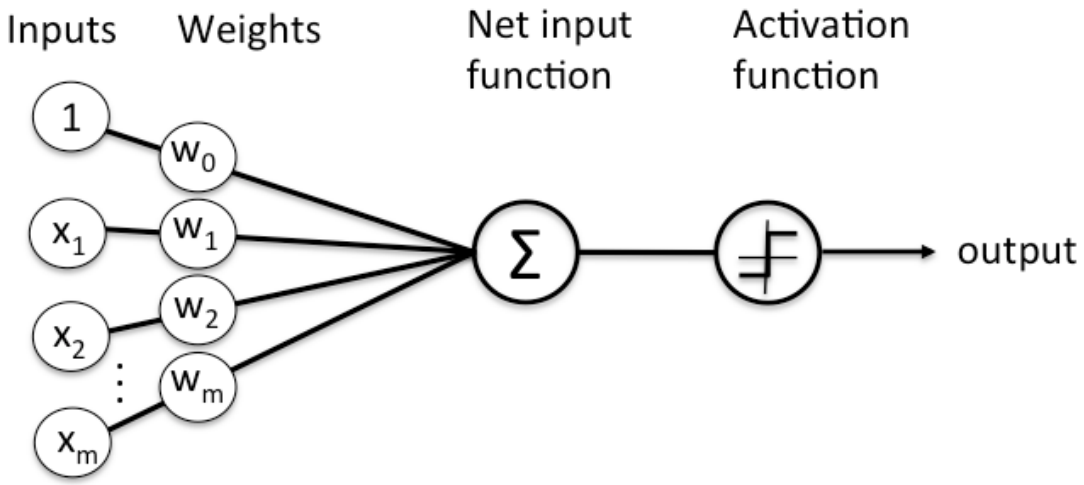

Figure 2.1: A single node, or neuron, producing an output, given multiple inputs [1].

To better understand how NNs are capable of learning a required task, we focus on a single neuron. Given a set of inputs and weights the neuron produces an output dictated by its activation function, as shown in Figure 2.1. Each input is multiplied by a weight, w_n, which is re-estimated for each training cycle. This weight indicates the significance its corresponding input has, in regards to task the NN is trying to learn. The input-weight products are summed and passed through the node's activation function [1]. The activation function determines the extent to which this signal should progress further through the network, ultimately contributing to the output.

As we are working with probabilistic values in language modelling, the sigmoid activation function is best suited for the neurons in our NN. This is because its output is bound from

0 to 1. The sigmoid activation function is defined by:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.22}$$

Let us further elaborate on how the NN re-estimates its weights. Taking the output computed by the NN and comparing it with the target output, we calculate a difference. We refer to this difference as error. A loss function dictates how error is calculated. We will be implementing and comparing the performance of two different loss functions in Chapter 4. These two loss functions are mean square error (MSE) and mean absolute error (MAE). Both effective when used for regression [7]. The formulas for MSE and MAE loss are respectively defined as:

$$error(x, y) = (x - y)^2 \tag{2.23}$$

and

$$error(x, y) = |x - y| \tag{2.24}$$

where x is the value computed by the NN and y is the target value.

Having calculated the error, we can infer the extent to which each weight needs re-estimation from an optimisation algorithm. The optimisation algorithm penalises those contributing largely to the error and reward those that do not. The optimisation algorithm used is Stochastic Gradient Descent (SGD). This is done for several iterations until the NN converges, i.e. the difference between the target- and actual output reaches local minima.

Chapter 3

Preparation

3.1. Text corpus

The text corpus used in this report is provided by the University of Stellenbosch. It comprises of text collected from several major South African newspapers [8], separated by the year it was published. Stored in 17 preproccessed text files, taken from 2000 to 2016. We partition this by selecting the last six years worth of text, containing 156 million words.

Preprocessing includes each sentence starting on a new line with start of sentence (<s>) and end of sentence (<\s>) tags appended to each line. For normalisation, all letters are presented in capital letters. Finally, whitespaces separate each token.

The six files, each containing the text accumulated for that year, are divided files between three separate sets: training, development and test sets. Figure 3.1 shows this breakdown.

The training set consists of 125 million words, making up 80% of all data provided. The training set is used to estimate a LM as well as train a neural network. Each individual file making up the training set, will have a n-gram count extracted from it. Thereafter the counts will be combined to form one final training set.

The development set is used to fine tune certain parameters in both the LM and neural network before the final testing phase. The outputs from several variations of language models and neural networks, being either perplexity scores or probabilities, are compared to choose the LM and neural network best suited for further comparison.

Finally each developed LM is evaluated on the test set to compare performance and determine final results. Our conclusion is based on this set.

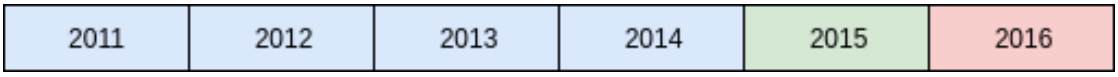

Figure 3.1: The split between Training (blue), Development (green) and Test (red) sets.

3.2. SRILM

3.2.1. Background

The SRI Language Modeling Toolkit has been in development since 1995 by SRI International, a non-profit research organisation. SRILM, comprised of a collection of C++ libraries, executable programs and helper scripts, is freely available for educational purposes [9].

SRILM enables the user to estimate and evaluate statistical language models, focusing on n-gram language modeling. SRILM is capable loading in large text corpora efficiently for producing a LM and calculating the probability of a giving test set, expressed in perplexity [9]. SRILM itself does not perform any text processing, and therefore assumes that the text has already been normalised and tokenised.

3.2.2. Implementation

To apply SRILM, it is first compiled into binary files for execution. Thereafter these binary files are executed and parsed the required parameters. This is all done from a Linux terminal.

Estimating a LM with SRILM is done in two steps. Firstly, producing a file containing the n-gram counts of a training set. Secondly, taking these counts and estimating probabilities for each n-gram. These two steps can be combined, however for large training sets, we separate them to prevent exceeding the available memory. The class used to accomplish this is *ngram-count*.

We start by feeding *ngram-count* four parameter options, a training set, tuple size, path at which to save the text file containing the n-gram counts, and finally the specified vocabulary. The vocabulary used, consists of the 60 thousand most seen words appearing in the corpus from 2000 to 2014. These counts will later be used to process inputs for our neural network as well.

To output these counts, the following command is executed in the terminal:

```
ngram-count -text [training_set]
-order [tuple_size]
-write [output_file_path]
-vocab [vocab]
```

The n-gram counts file for each input file is produced and combined using *ngram-merge.* Having produced the final n-gram counts file, SRILM can now estimate a LM by executing:

```
ngram-count -read [ngram_counts] -lm [output_file_path]
```

The LM is stored in ARPA format at the specified path [9]. ARPA files store the total count of n-grams for each tuple size. Following that, each n-gram is listed with its conditional probability (in base 10 logarithmic form) followed by its backoff weight, if applicable. Note that $\log_{10} 0$ is represented as -99.

When estimating a LM, there are two additional parameters options we will be utilising, namely:

```
-addsmooth [n]
-gt3min [n]
```

Unless otherwise specified, SRILM incorporates Good-Turing discounting together with Katz backoff to smooth the LM [9]. If a LM without any implemented smoothing technique is desired, *-addsmooth 0* should be specified. The *addsmooth* option, indicates that the LM should be smoothed by adding the count n to all n-gram counts. If $n = 0$, then SRILM effectively produces an unsmoothed LM.

By default, SRILM also discards n-grams, of tuple size 3 or higher, occurring only once in the training set. Essentially SRILM classifies these n-grams as unseen ones. This is done for optimisation reasons. The argument, *-gt3min [n]* with $n = 0$, prevents this optimisation, however it could have certain ramifications. Section 3.3 elaborates on this and evaluates whether or not implementing this would be beneficial.

Our final use for SRILM is evaluating the LM. SRILM does this by calculating perplexity, given a test set, via the *ngram* class. When calling *ngram* we parse the test set and LM:

```
ngram -ppl [test_set] -lm [LM] -debug [n]
```

SRILM calculates and outputs two different perplexity scores, referred to as *ppl* and *ppl1*. The former, indicating the perplexity score counting all tokens found in the test set and the latter, excluding end of sentence tags. Optionally, the level of detail printed is specified by the *-debug* parameter.

It is important to note that evaluating an unsmoothed LM will produce a perplexity score of infinity due to the zero probability of unseen n-grams being present in the test set. When SRILM encounters a zero probability n-gram, it sets it aside to continue calculating the perplexity. The count of zero probability n-grams is thereafter displayed in the output.

3.3. Baseline Language Model

To implement a neural network smoothed LM, we first develop as a baseline, an unsmoothed LM. Our target being to improve its perplexity scores. Together with the newly proposed neural network smoothing, traditional smoothing will be used as a reference, to compare the extent of which the perplexity scores have improved. The traditional smoothing technique used is Good-Turing discounting together with Katz backoff.

Before developing theses models we first evaluate what impact the *-gt3min 1* parameter option has on a LM produced by SRILM. We construct two separate models, both being the default LM SRILM produces, using the training set. The difference is that one contains no optimisation i.e. trained with the parameter option *-gt3min 1*. We evaluate both on our development set.

Table 3.1: Perplexity Output of each LM on development set

LM condition	Perplexity (*ppl*)	Perplexity (*ppl1*)
Optimised	152.128	199.633
Unoptimised	150.533	197.427

As seen in Table 3.1, producing an optimised version does slightly decrease its perplexity, however negligibly. The drawback of producing an unoptimised LM is that it is large (roughly twice the size of an optimised LM). This makes memory management another concern. Given the little improvement on perplexity, it is clear that working with an optimised LM is more beneficial.

We now estimate our baseline unsmoothed LM and reference smoothed LM with default SRILM parameter options. We compute and compare their test set perplexity.

Table 3.2: Perplexity output of each LM on test set

LM condition	Perplexity (*ppl*)	Perplexity (*ppl1*)
Unsmoothed	117.544	175.55
Smoothed	104.939	154.264

Table 3.2 shows a 12% to 13% improvement using the traditional smoothing technique. We now move on to training a neural network to replicate this result.

Chapter 4

Neural Network Smoothing

4.1. Python & Pytorch

Python is the chosen programming language in which our NN will be developed and implemented in. It is an open source, interpreted language, with libraries available for NN development. The environment in which our python scripts are executed in is Jupyter Notebook. All python notebooks and classes are available in the following Git repository: ...

The Pytorch library enables the user to develop a NN from scratch. It supports GPU accelerated computing, allowing cutting model training times down significantly. Pytorch includes all necessary activation functions, loss functions and optimiser algorithms required.

4.2. Input Features

The data available to the NN includes an unsmoothed LM stored in ARPA format, together with all n-gram counts which we load into a python dictionary. We then step through the ARPA file line by line, load in each n-gram and start pulling different counts from it:

- Prefix count
- N-gram count
- $(N-1)$-gram count
- $(N-2)$-gram count, if applicable

To ensure the NN converges, these counts will have to be normalised to lie between 0 and 1. To do so we simply divide 1 by the counts.

Together with those counts we add more input features. We do so by and analyzing the ngram itself. Binary is used to indicate whether or not words repeat in the n-gram.

It is important to note that these input features are only pulled from n-grams occurring seven or less times. This is because these are the only n-grams applicable to smoothing. For all n-grams accruing more than seven times, a MLE probability is used and will not be used to train our model.

4.2.1. Target Output

We use as target output, the probability estimate present in the smoothed LM ARPA file. Alternatively, we calculate it by taking the MLE probability of the n-gram from Equation (2.5) and multiplying it by the corresponding Good-Turing discount factors. Doing so is more memory efficient. These discount factors can be exported from an n-gram count using SRILM. Different discount factors are used for each occurrence count as well as tuple size.

Currently we will be using SRILM to recompute backoff probabilities.

To improve prediction results and ensure the model trains better, we will be developing two separate models for bigrams and trigrams.

4.3. Smoothing Trigrams

4.3.1. Features and Shape

4431

4.3.2. Training and Dev Set results

compare lr, and loss function maybe

4.4. Smoothing ARPA File

Chapter 5

Summary and Conclusion

Bibliography

[1] C. Nicholson, "A beginner's guide to neural networks and deep learning," 2019. [Online]. Available: https://skymind.ai/wiki/neural-network. [Accessed: 24- Aug- 2019].

[2] D. Jurafsky and J. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, ser. Pearson international edition. Prentice Hall, 2009.

[3] T. R. Niesler, "Category-based statistical language models," Ph.D. dissertation, University of Cambridge Cambridge, UK, 1997.

[4] C. Manning, C. Manning, H. Schütze, and H. SCHUTZE, *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[5] I. J. Good, "The Population Frequencies of Species and the Stimation of Population Parameters," *Biometrika*, vol. 40, no. 3-4, pp. 237–264, 1953.

[6] L. Mak, "Next word prediction using katz backoff model," 2019. [Online]. Available: https://rpubs.com/leomak/TextPrediction_KBO_Katz_Good-Turing. [Accessed: 6- Aug- 2019].

[7] P. Jha, "A brief overview of loss functions in pytorch," 2019. [Online]. Available: https://medium.com/udacity-pytorch-challengers/a-brief-overview-of-loss-functions-in-pytorch-c0ddb78068f7. [Accessed: 25- Aug- 2019].

[8] H. Kamper, F. Wet, T. Hain, and T. Niesler, "Capitalising on north american speech resources for the development of a south african english large vocabulary speech recognition system," *Computer Speech and Language*, vol. 28, pp. 1255–1268, 2014.

[9] A. Stolcke, "Srilm - an extensible language modeling toolkit," in *INTERSPEECH*, 2002.

Appendix A

SRILM License

SRILM Research Community License Version 1.1

Preamble

THE SRILM RESEARCH COMMUNITY LICENSE PERMITS USE OF THE SOFTWARE ONLY BY GOVERNMENT AGENCIES, OR BY SCHOOLS, UNIVERSITIES, AND NON-PROFIT ORGANIZATIONS FOR PROJECTS THAT DO NOT RECEIVE EXTERNAL FUNDING OTHER THAN GOVERNMENT RESEARCH GRANTS AND CONTRACTS. FOR THE SPECIFIC LIMITATIONS, SEE SECTION 1.12. ANY OTHER USE REQUIRES A COMMERCIAL LICENSE. PLEASE CONTACT US FOR DETAILS AT http://www.speechatsri.com/utils/contact.php.

1. DEFINITIONS

1.1. "Community" means the Initial Developer and each and every one of You.

1.2. "Contributor" means each individual or legal entity that creates or contributes to the creation of Modifications.

1.3. "Distribute" means to transfer, publish, or otherwise provide or make available a copy of software to any third party.

1.4. "Distributor" means each individual or legal entity exercising a right to Distribute Licensed Software under this License.

1.5. "Distributor Code" means, collectively, (a) the Original Source Code and (b) the Licensed Software that is Distributed by a particular Distributor.

1.6. "Distributor IP" means all patent claims, copyright, and trade secrets owned or licensable by a Distributor that are embodied in or necessarily practiced by the Distributor Code of that particular Distributor.

1.7. "Electronic Distribution Mechanism" means a mechanism generally accepted in the software development community for the electronic transfer of data.

1.8. "Executable" means software in any form other than Source Code.

1.9. "Initial Developer" means SRI International ("SRI"), a California nonprofit public benefit corporation having offices in Menlo Park, California.

1.10. "Initial Developer IP" means all patent claims, copyright, and trade secrets owned or licensable by the Initial Developer which are embodied in or necessarily practiced

by the unmodified Original Source Code.

1.11. "License" means this agreement, the SRILM Research Community License Version 1.1.

1.12. "Licensed Purpose" means the exercise of rights under this License: (a) by an official governmental agency, strictly for non-commercial, public benefit purposes; or (b) by a school, university or non-profit organization solely for non-commercial teaching or research purposes in exchange for which no financially valuable consideration (including but not limited to sales or license revenue, service revenue, and advertising revenue, and further including but not limited to monetary, equity, and in-kind forms of consideration) is received, except for tuition, or research funding received from an official government agency sponsor.

1.13. "Licensed Software" means any copy of the Original Source Code, the Modifications, and/or Executable versions thereof, as to which You are exercising any rights under this License.

1.14. "Modifications" means any derivative work of either the Original Source Code or of existing Modifications.Including header files or calls to Licensed Software library functions or Executable programs in an application program shall not in and of itself cause that application program to be deemed a Modification, provided that the Licensed Software so included shall in any case remain subject to the terms of this License.

1.15. "Original Source Code" means the Source Code for the SRI Language Modeling Toolkit as made available for download by SRI International from URL http://www.speech.sri.com/ or ftp://ftp.speech.sri.com/.

1.16. "Source Code" means the preferred form of software for making modifications to it, including all modules it contains, plus any associated interface definition files, scripts used to control compilation and installation of an Executable, or a list of source code differential comparisons against either the Original Source Code or another well known, available Modification of the Contributor's choice. The Source Code can be in a compressed or archival form, provided the appropriate decompression or de-archiving software is widely available for no charge.

1.17. "You" means an individual or a legal entity exercising rights under this License. For legal entities, "You" includes any entity that controls, is controlled by, or is under common control with You. For purposes of this definition, "control" means (a) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (b) ownership of fifty percent (50%) or more of the outstanding shares or beneficial ownership of such entity.

2. SOFTWARE LICENSE GRANTS

2.1. The Initial Developer Grant. The Initial Developer hereby grants the Community worldwide, royalty-free, non-exclusive rights, solely for the Licensed Purpose and subject

to the terms and conditions of this License(including without limitation the Contributor and Distributor grants set forth in Section 2 and the Distribution Obligations set forth in Section 3), as follows:

(a) A license under the Initial Developer IP to modify, copy, use, and Distribute the Original Source Code and Modifications. (b) A license to copy, use and Distribute Executable versions of the Original Source Code and Modifications.

2.2. Distributor Grant. If You are a Distributor, You hereby grant the Community a worldwide, royalty-free, non-exclusive license under the Distributor IP, solely for the Licensed Purpose and subject to the terms and conditions of this License, to modify copy, use, and Distribute the Distributor Code and Modifications.

2.3. Sublicenses. The license rights that are granted under this Section 2 may be sublicensed only by Distributing Licensed Software in compliance with all applicable terms and conditions of this License. No other sublicensing of intellectual property rights granted under this License is permitted.

2.4. No Other Rights. Except as expressly set forth in this section 2, no other license rights are implied or otherwise granted under this License.

3. DISTRIBUTION OBLIGATIONS

The right to Distribute Licensed Software set forth above, whether to a government sponsor or to any other recipient, is subject to the following obligations:

3.1. Notice of License to Recipients.Any publication or other Distribution of Licensed Software must be made expressly subject to this License.You must include a prominent copy of the notice set forth in Exhibit A with all Licensed Software that you Distribute: (a) in all notices or documentation in which You describe the origin or ownership of the software or the recipient's rights thereto, and (b) in each Source Code file. You hereby agree to indemnify the Initial Developer and every Contributor to the Licensed Software for any and all liability or damages they may incur if caused by Your violation of this Section. Except as expressly authorized by this License, no right is granted to use the name of SRI in any advertising, news release, other publication, or product documentation, without the prior express written consent of SRI.

3.2. Required Availability of Source Code. If You Distribute Licensed Software for which you are a Contributor, You must make the Source Code for all Modifications to which you contribute publicly available to the Community, under the terms of this License and at no additional restriction or cost, via an accepted Electronic Distribution Mechanism for a period of at least twelve (12) months following the date You first Distribute the Licensed Software.You must also include with the Source Code a file documenting any changes You made to create each Modification, and the dates of such changes.

3.3. Licensee Registration. Before You Distribute any Licensed Software under this License, You must first register by sending email to the Initial Developer addressed to

srilm@speech.sri.com, including a statement confirming that you accept the terms and conditions of this License and describing the specific Electronic Distribution Mechanism you are using (e.g., identify the URL) to make Source Code available under Section 3.2 where applicable.

4. DISCLAIMER OF WARRANTY

LICENSED SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, WITHOUT LIMITATION, WARRANTIES THAT THE LICENSED SOFTWARE IS FREE OF DEFECTS, MERCHANTABLE, FIT FOR A PARTICULAR PURPOSE OR NON-INFRINGING. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LICENSED SOFTWARE IS WITH YOU. SHOULD ANY LICENSED SOFTWARE PROVE DEFECTIVE IN ANY RESPECT, YOU (NOT THE INITIAL DEVELOPER OR ANY CONTRIBUTOR) ASSUME THE COST OF ANY NECESSARY SERVICING, REPAIR OR CORRECTION. THIS DISCLAIMER OF WARRANTY CONSTITUTES AN ESSENTIAL PART OF THIS LICENSE. NO USE OF ANY LICENSED SOFTWARE IS AUTHORIZED HEREUNDER EXCEPT UNDER THIS DISCLAIMER.

5. LIMITATION OF LIABILITY

UNDER NO CIRCUMSTANCES AND UNDER NO LEGAL THEORY, WHETHER TORT (INCLUDING NEGLIGENCE), CONTRACT, OR OTHERWISE, SHALL THE INITIAL DEVELOPER, ANY CONTRIBUTOR, OR ANY DISTRIBUTOR OF LICENSED SOFTWARE, OR ANY SUPPLIER OF ANY OF SUCH PARTIES, BE LIABLE TO YOU OR ANY OTHER PERSON FOR ANY INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY CHARACTER INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF GOODWILL, WORK STOPPAGE, COMPUTER FAILURE OR MALFUNCTION, OR ANY AND ALL OTHER COMMERCIAL DAMAGES OR LOSSES, EVEN IF SUCH PARTY SHALL HAVE BEEN INFORMED OF THE POSSIBILITY OF SUCH DAMAGES. THIS LIMITATION OF LIABILITY SHALL NOT APPLY TO LIABILITY FOR DEATH OR PERSONAL INJURY RESULTING FROM SUCH PARTY'S NEGLIGENCE TO THE EXTENT APPLICABLE LAW PROHIBITS SUCH LIMITATION. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THAT EXCLUSION AND LIMITATION MAY NOT APPLY TO YOU.

6. RESPONSIBILITY FOR CLAIMS.

You are responsible for damages arising, directly or indirectly, out of Your exercise of rights under this License, based on the number of copies of Licensed Software you made available, the revenues you received from utilizing such rights, and other relevant factors.

You agree to work with affected parties to distribute responsibility on an equitable basis.

7. U.S. GOVERNMENT END USERS.

The Licensed Software is a "commercial item," as that term is defined in 48 C.F.R. 2.101 (Oct. 1995), consisting of "commercial computer software" and "commercial computer software documentation," as such terms are used in 48 C.F.R. 12.212 (Sept. 1995). Consistent with 48 C.F.R. 12.212 and 48 C.F.R. 227.7202-1 through 227.7202-4 (June 1995), all U.S. Government End Users acquire Licensed Software with only those rights set forth herein.

8. MISCELLANEOUS.

This License represents the complete agreement concerning subject matter hereof. If any provision of this License is held to be unenforceable, such provision shall be reformed only to the extent necessary to make it enforceable. This License shall be governed by California law provisions (except to the extent applicable law, if any, provides otherwise), excluding its conflict-of-law provisions. With respect to disputes in which at least one party is a citizen of, or an entity chartered or registered to do business in, the United States of America any litigation relating to this Agreement shall be subject to personal jurisdiction and venue in the Federal Courts of the Northern District of California, and in the California state courts of San Mateo County, California, with the losing party responsible for costs, including without limitation, court costs and reasonable attorneys fees and expenses. The application of the United Nations Convention on Contracts for the International Sale of Goods is expressly excluded. Any law or regulation which provides that the language of a contract shall be construed against the drafter shall not apply to this License.

EXHIBIT A.

The contents of this file are subject to the SRILM Community Research License Version 1.1 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.speech.sri.com/srilm/. Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License. Portions of the software are Copyright (c) SRI International, 1995-2013. All rights reserved.

Appendix B

SRILM Class Documentation

ngram-format

NAME

ngram-format - File format for ARPA backoff N-gram models

SYNOPSIS

```
\data\
ngram 1=n1
ngram 2=n2
...
ngram N=nN

\1-grams:
p    w        [bow]
...

\2-grams:
p    w1 w2        [bow]
...

\N-grams:
p    w1 ... wN
...

\end\
```

DESCRIPTION

The so-called ARPA (or Doug Paul) format for N-gram backoff models starts with a header, introduced by the keyword **\data**, listing the number of N-grams of each length. Following that, N-grams are listed one per line, grouped into sections by length, each section starting with the keyword ***N*-gram:**, where *N* is the length of the N-grams to follow. Each N-gram line starts with the logarithm (base 10) of conditional probability *p* of that N-gram, followed by the words *w1*... *wN* making up the N-gram. These are optionally followed by the logarithm (base 10) of the backoff weight for the N-gram. The keyword **\end** concludes the model representation.

Backoff weights are required only for those N-grams that form a prefix of longer N-grams in the model. The highest-order N-grams in particular will not need backoff weights (they would be useless).

Since log(0) (minus infinity) has no portable representation, such values are mapped to a large negative number. However, the designated dummy value (-99 in SRILM) is interpreted as log(0) when read back from file into memory.

The correctness of the N-gram counts *n1*, *n2*, ... in the header is not enforced by SRILM software when reading models (although a warning is printed when an inconsistency is encountered). This allows easy textual insertion or deletion of parameters in a model file. The proper format can be recovered by passsing the model through the command

```
ngram -order N -lm input -write-lm output
```

Note that the format is self-delimiting, allowing multiple models to be stored in one file, or to be surrounded by ancillary information. Some extensions of N-gram models in SRILM store additional parameters after a basic N-gram section in the standard format.

ngram-count

NAME

ngram-count - count N-grams and estimate language models

SYNOPSIS

```
ngram-count [ -help ] option ...
```

DESCRIPTION

ngram-count generates and manipulates N-gram counts, and estimates N-gram language models from them. The program first builds an internal N-gram count set, either by reading counts from a file, or by scanning text input. Following that, the resulting counts can be output back to a file or used for building an N-gram language model in ARPA ngram-format(5). Each of these actions is triggered by corresponding options, as described below.

OPTIONS

Each filename argument can be an ASCII file, or a compressed file (name ending in .Z or .gz), or "-" to indicate stdin/stdout.

-help Print option summary.

-version Print version information.

-order *n* Set the maximal order (length) of N-grams to count. This also determines the order of the estimated LM, if any. The default order is 3.

-vocab *file* Read a vocabulary from file. Subsequently, out-of-vocabulary words in both counts or text are replaced with the unknown-word token. If this option is not specified all words found are implicitly added to the vocabulary.

-vocab-aliases *file* Reads vocabulary alias definitions from *file*, consisting of lines of the form

alias word

This causes all tokens *alias* to be mapped to *word*.

-write-vocab *file* Write the vocabulary built in the counting process to *file*.

-write-vocab-index *file* Write the internal word-to-integer mapping to *file*.

-tagged Interpret text and N-grams as consisting of word/tag pairs.

-tolower Map all vocabulary to lowercase.

-memuse Print memory usage statistics.

Counting Options

-text *textfile* Generate N-gram counts from text file. *textfile* should contain one sentence unit per line. Begin/end sentence tokens are added if not already present. Empty lines are ignored.

-text-has-weights Treat the first field in each text input line as a weight factor by which the N-gram counts for that line are to be multiplied.

-no-sos Disable the automatic insertion of start-of-sentence tokens in N-gram counting.

-no-eos Disable the automatic insertion of end-of-sentence tokens in N-gram counting.

-read *countsfile* Read N-gram counts from a file. Ascii count files contain one N-gram of words per line, followed by an integer count, all separated by whitespace. Repeated counts for the same N-gram are added. Thus several count files can be merged by using cat(1) and feeding the result to **ngram-count -read -** (but see ngram-merge(1) for merging counts that exceed available memory). Counts collected by **-text** and **-read** are additive as well. Binary count files (see below) are also recognized.

-read-google *dir* Read N-grams counts from an indexed directory structure rooted in **dir**, in a format developed by Google to store very large N-gram collections. The corresponding directory structure can be created using the script **make-google-ngrams** described in training-scripts(1).

-intersect *file* Limit reading of counts via **-read** and **-read-google** to a subset of N-grams defined by another count *file*. (The counts in *file* are ignored, only the N-grams are relevant.)

-write *file* Write total counts to *file*.

-write-binary *file* Write total counts to *file* in binary format. Binary count files cannot be compressed and are typically larger than compressed ascii count files. However, they can be loaded faster, especially when the **-limit-vocab** option is used.

-write-order *n* Order of counts to write. The default is 0, which stands for N-grams of all lengths.

-write*n* *file* where n is 1, 2, 3, 4, 5, 6, 7, 8, or 9. Writes only counts of the indicated order to *file*. This is convenient to generate counts of different orders separately in a single pass.

-sort Output counts in lexicographic order, as required for ngram-merge(1).

-recompute Regenerate lower-order counts by summing the highest-order counts for each N-gram prefix.

-limit-vocab Discard N-gram counts on reading that do not pertain to the words specified in the vocabulary. The default is that words used in the count files are automatically added to the vocabulary.

LM Options

-lm *lmfile* Estimate a language model from the total counts and write it to *lmfile*. This option applies to several LM model types (see below) but the default is to estimate a backoff N-gram model, and write it in ngram-format(5).

-write-binary-lm Output *lmfile* in binary format. If an LM class does not provide a binary format the default (text) format will be output instead.

-nonevents *file* Read a list of words from *file* that are to be considered non-events, i.e., that can only occur in the context of an N-gram. Such words are given zero probability mass in model estimation.

-float-counts Enable manipulation of fractional counts. Only certain discounting methods support non-integer counts.

-skip Estimate a "skip" N-gram model, which predicts a word by an interpolation of the immediate context and the context one word prior. This also triggers N-gram counts to be generated that are one word longer than the indicated order. The following four options control the EM estimation algorithm used for skip-N-grams.

-init-lm *lmfile* Load an LM to initialize the parameters of the skip-N-gram.

-skip-init *value* The initial skip probability for all words.

-em-iters *n* The maximum number of EM iterations.

-em-delta *d* The convergence criterion for EM: if the relative change in log likelihood falls below the given value, iteration stops.

-count-lm Estimate a count-based interpolated LM using Jelinek-Mercer smoothing (Chen & Goodman, 1998). Several of the options for skip-N-gram LMs (above) apply.

An initial count-LM in the format described in ngram(1) needs to be specified using **-init-lm**. The options **-em-iters** and **-em-delta** control termination of the EM algorithm. Note that the N-gram counts used to estimate the maximum-likelihood estimates come from the **-init-lm** model. The counts specified with **-read** or **-text** are used only to estimate the smoothing (interpolation weights).

-maxent Estimate a maximum entropy N-gram model. The model is written to the file specifed by the **-lm** option.
If **-init-lm** *file* is also specified then use an existing maxent model from *file* as prior and adapt it using the given data.

-maxent-alpha *A* Use the L1 regularization constant A for maxent estimation. The default value is 0.5.

-maxent-sigma2 *S* Use the L2 regularization constant S for maxent estimation. The default value is 6 for estimation and 0.5 for adaptation.

-maxent-convert-to-arpa Convert the maxent model to ngram-format(5) before writing it out, using the algorithm by Wu (2002).

-unk Build an "open vocabulary" LM, i.e., one that contains the unknown-word token as a regular word. The default is to remove the unknown word.

-map-unk *word* Map out-of-vocabulary words to *word*, rather than the default tag.

-trust-totals Force the lower-order counts to be used as total counts in estimating N-gram probabilities. Usually these totals are recomputed from the higher-order counts.

-prune *threshold* Prune N-gram probabilities if their removal causes (training set) perplexity of the model to increase by less than *threshold* relative.

-minprune *n* Only prune N-grams of length at least n. The default (and minimum allowed value) is 2, i.e., only unigrams are excluded from pruning.

-debug *level* Set debugging output from estimated LM at *level*. Level 0 means no debugging. Debugging messages are written to stderr.

-gt*n*min *count* where n is 1, 2, 3, 4, 5, 6, 7, 8, or 9. Set the minimal count of N-grams of order n that will be included in the LM. All N-grams with frequency lower than that will effectively be discounted to 0. If n is omitted the parameter for N-grams of order > 9 is set.
NOTE: This option affects not only the default Good-Turing discounting but the alternative discounting methods described below as well.

-gt*n*max *count* where n is 1, 2, 3, 4, 5, 6, 7, 8, or 9. Set the maximal count of N-grams of order n that are discounted under Good-Turing. All N-grams more frequent than that will receive maximum likelihood estimates. Discounting can be effectively disabled by setting this to 0. If n is omitted the parameter for N-grams of order > 9 is set.

In the following discounting parameter options, the order n may be omitted, in which case a default for all N-gram orders is set. The corresponding discounting method then becomes the default method for all orders, unless specifically overridden by an option with n. If no discounting method is specified, Good-Turing is used.

-gt*n* *gtfile* where n is 1, 2, 3, 4, 5, 6, 7, 8, or 9. Save or retrieve Good-Turing parameters (cutoffs and discounting factors) in/from *gtfile*. This is useful as GT parameters should always be determined from unlimited vocabulary counts, whereas the eventual LM may use a limited vocabulary. The parameter files may also be hand-edited. If an **-lm** option is specified the GT parameters are read from *gtfile*, otherwise they are computed from the current counts and saved in *gtfile*.

-cdiscount*n* *discount* where n is 1, 2, 3, 4, 5, 6, 7, 8, or 9. Use Ney's absolute discounting for N-grams of order n, using *discount* as the constant to subtract.

-wbdiscount*n* where n is 1, 2, 3, 4, 5, 6, 7, 8, or 9. Use Witten-Bell discounting for N-grams of order n. (This is the estimator where the first occurrence of each word is taken to be a sample for the "unseen" event.)

-ndiscount*n* where n is 1, 2, 3, 4, 5, 6, 7, 8, or 9. Use Ristad's natural discounting law for N-grams of order n.

-addsmooth*n* *delta* where n is 1, 2, 3, 4, 5, 6, 7, 8, or 9. Smooth by adding *delta* to each N-gram count. This is usually a poor smoothing method and included mainly for instructional purposes.

ngram

NAME

ngram - apply N-gram language models

SYNOPSIS

```
ngram [ -help ] option ...
```

DESCRIPTION

ngram performs various operations with N-gram-based and related language models, including sentence scoring, perplexity computation, sentences generation, and various types of model interpolation. The N-gram language models are read from files in ARPA ngram-format(5); various extended language model formats are described with the options below.

OPTIONS

Each filename argument can be an ASCII file, or a compressed file (name ending in .Z or .gz), or "-" to indicate stdin/stdout.

-help Print option summary.

-version Print version information.

-order ***n*** Set the maximal N-gram order to be used, by default 3. NOTE: The order of the model is not set automatically when a model file is read, so the same file can be used at various orders. To use models of order higher than 3 it is always necessary to specify this option.

-debug ***level*** Set the debugging output level (0 means no debugging output). Debugging messages are sent to stderr, with the exception of **-ppl** output as explained below.

-memuse Print memory usage statistics for the LM.

The following options determine the type of LM to be used.

-null

Use a 'null' LM as the main model (one that gives probability 1 to all words). This is useful in combination with mixture creation or for debugging.

-lm *file*

Read the (main) N-gram model from *file*. This option is always required, unless **-null** was chosen. Unless modified by other options, the *file* is assumed to contain an N-gram backoff language model in ngram-format(5).

-tagged

Interpret the LM as containing word/tag N-grams.

-skip

Interpret the LM as a "skip" N-gram model.

-hidden-vocab *file*

Interpret the LM as an N-gram model containing hidden events between words. The list of hidden event tags is read from *file*.

Hidden event definitions may also follow the N-gram definitions in the LM file (the argument to **-lm**). The format for such definitions is

```
event [-delete D] [-repeat R] [-insert w] [-observed] [-omit]
```

The optional flags after the event name modify the default behavior of hidden events in the model. By default events are unobserved pseudo-words of which at most one can occur between regular words, and which are added to the context to predict following words and events. (A typical use would be to model hidden sentence boundaries.) **-delete** indicates that upon encountering the event, D words are deleted from the next word's context. **-repeat** indicates that after the event the next R words from the context are to be repeated. **-insert** specifies that an (unobserved) word w is to be inserted into the history. **-observed** specifies the event tag is not hidden, but observed in the word stream. **-omit** indicates that the event tag itself is not to be added to the history for predicting the following words.

The hidden event mechanism represents a generalization of the disfluency LM enabled by **-df**.

-hidden-not

Modifies processing of hidden event N-grams for the case that the event tags are embedded in the word stream, as opposed to inferred through dynamic programming.

-vocab *file*

Initialize the vocabulary for the LM from *file*. This is especially useful if the LM itself does not specify a complete vocabulary, e.g., as with **-null**.

-vocab-aliases *file*

Reads vocabulary alias definitions from *file*, consisting of lines of the form

```
alias word
```

This causes all tokens *alias* to be mapped to *word*.

-nonevents *file*

Read a list of words from *file* that are to be considered non-events, i.e., that should only occur in LM contexts, but not as predictions. Such words are excluded from sentence generation (**-gen**) and probability summation (**-ppl -debug 3**).

-limit-vocab

Discard LM parameters on reading that do not pertain to the words specified in the vocabulary. The default is that words used in the LM are automatically added to the vocabulary. This option can be used to reduce the memory requirements for large LMs that are going to be evaluated only on a small vocabulary subset.

-unk

Indicates that the LM contains the unknown word, i.e., is an open-class LM.

-map-unk *word*

Map out-of-vocabulary words to *word*, rather than the default **<unk>** tag.

-tolower

Map all vocabulary to lowercase. Useful if case conventions for text/counts and language model differ.

-multiwords

Split input words consisting of multiwords joined by underscores into their components, before evaluating LM probabilities.

-multi-char *C*

Character used to delimit component words in multiwords (an underscore character by default).

-zeroprob-word *W*

If a word token is assigned a probability of zero by the LM, look up the word *W* instead. This is useful to avoid zero probabilities when processing input with an LM that is mismatched in vocabulary.

-context-priors file

Read context-dependent mixture weight priors from *file*. Each line in *file* should contain a context N-gram (most recent word first) followed by a vector of mixture weights whose length matches the number of LMs being interpolated. (This and the following options currently only apply to linear interpolation.)

-weight ***W*** the prior weight given to the component LM

-order ***N*** the maximal ngram order to use

-type ***T*** the LM type, one of **ARPA** (the default), **COUNTLM**, **MAXENT**, **LM-CLIENT**, or **MSWEBLM**

-classes ***C*** the word class definitions for the component LM (which must be of type ARPA)

The following options specify the operations performed on/with the LM constructed as per the options above.

-renorm Renormalize the main model by recomputing backoff weights for the given probabilities.

-prune *threshold* Prune N-gram probabilities if their removal causes (training set) perplexity of the model to increase by less than *threshold* relative.

-prune-history-lm *L* Read a separate LM from file *L* and use it to obtain the history marginal probabilities required for computing the entropy loss incurred by pruning an N-gram. The LM needs to only be of an order one less than the LM being pruned. If this option is not used the LM being pruned is used to compute history marginals. This option is useful because, as pointed out by Chelba et al. (2010), the lower-order N-gram probabilities in Kneser-Ney smoothed LMs are unsuitable for this purpose.

-prune-lowprobs Prune N-gram probabilities that are lower than the corresponding backed-off estimates. This generates N-gram models that can be correctly converted into probabilistic finite-state networks.

-minprune *n* Only prune N-grams of length at least *n*. The default (and minimum allowed value) is 2, i.e., only unigrams are excluded from pruning. This option applies to both **-prune** and **-prune-lowprobs**.

-rescore-ngram *file* Read an N-gram LM from *file* and recompute its N-gram probabilities using the LM specified by the other options; then renormalize and evaluate the resulting new N-gram LM.

-write-lm *file* Write a model back to *file*. The output will be in the same format as read by **-lm**, except if operations such as **-mix-lm** or **-expand-classes** were applied, in which case the output will contain the generated single N-gram backoff model in ARPA ngram-format(5).

-write-bin-lm *file* Write a model to *file* using a binary data format. This is only supported by certain model types, specifically, those based on N-gram backoff models and N-gram counts. Binary model files are recognized automatically by the **-read** function. If an LM class does not provide a binary format the default (text) format will be output instead.

-write-vocab *file* Write the LM's vocabulary to *file*.

-gen *number* Generate *number* random sentences from the LM.

-gen-prefixes *file* Read a list of sentence prefixes from *file* and generate random word strings conditioned on them, one per line. (Note: The start-of-sentence tag **<s>** is not automatically added to these prefixes.)

-seed *value* Initialize the random number generator used for sentence generation using seed *value*. The default is to use a seed that should be close to unique for each invocation of the program.

-ppl *textfile* Compute sentence scores (log probabilities) and perplexities from the sentences in *textfile*, which should contain one sentence per line. The **-debug** option controls the level of detail printed, even though output is to stdout (not stderr).

-debug 0 Only summary statistics for the entire corpus are printed, as well as partial statistics for each input portion delimited by escaped lines (see **-escape**). These statistics include the number of sentences, words, out-of-vocabulary words and zero-probability tokens in the input, as well as its total log probability and perplexity. Perplexity is given with two different normalizations: counting all input tokens ("ppl'') and excluding end-of-sentence tags ("ppl1").

-debug 1 Statistics for individual sentences are printed.

-debug 2 Probabilities for each word, plus LM-dependent details about backoff used etc., are printed.

-debug 3 Probabilities for all words are summed in each context, and the sum is printed. If this differs significantly from 1, a warning message to stderr will be issued.

-debug 4 Outputs ranking statistics (number of times the actual word's probability was ranked in top 1, 5, 10 among all possible words, both excluding and including end-of-sentence tokens), as well as quadratic and absolute loss averages (based on how much actual word probability differs from 1).

-text-has-weights Treat the first field on each **-ppl** input line as a weight factor by which the statistics for that sentence are to be multiplied.

-rescore *file* Similar to **-nbest**, but the input is processed as a stream of N-best hypotheses (without header). The output consists of the rescored hypotheses in SRILM format (the third of the formats described in nbest-format(5)).

-escape *string* Set an "escape string" for the **-ppl**, **-counts**, and **-rescore** computations. Input lines starting with *string* are not processed as sentences and passed unchanged to stdout instead. This allows associated information to be passed to scoring scripts etc.

-counts *countsfile* Perform a computation similar to **-ppl**, but based only on the N-gram counts found in *countsfile*. Probabilities are computed for the last word of each N-gram, using the other words as contexts, and scaling by the associated N-gram count. Summary statistics are output at the end, as well as before each escaped input line if **-debug** level 1 or higher is set.

-count-order *n* Use only counts up to order n in the **-counts** computation. The default value is the order of the LM (the value specified by **-order**).

-float-counts Allow processing of fractional counts with **-counts**.

-counts-entropy Weight the log probabilities for **-counts** processing by the join probabilities of the N-grams. This effectively computes the sum over p(w,h) log p(w|h), i.e., the entropy of the model. In debugging mode, both the conditional log probabilities and the corresponding joint probabilities are output.

-skipoovs Instruct the LM to skip over contexts that contain out-of-vocabulary words, instead of using a backoff strategy in these cases.

-no-sos Disable the automatic insertion of start-of-sentence tokens for sentence probability computation. The probability of the initial word is thus computed with an empty context.

-no-eos Disable the automatic insertion of end-of-sentence tokens for sentence probability computation. End-of-sentence is thus excluded from the total probability.