Implementing and Analyzing an N-Gram Language using Smoothing, Backoff, and Interpolation.

CS437 – Machine Learning and Soft Computing

Adam Brazda

12/08/2025

Introduction

Language modeling is a type of machine learning algorithm that aims to predict and generate natural sounding text. This, however, is a difficult task; natural language is difficult to capture algorithmically. This is due to the complicated and irregular behavior of language. There are many different approaches used to try and solve the issue of translating natural language into an algorithm such as n-grams, neural networks, and vector embeddings. However, for this paper we will focus on the n-gram model.

Why N-grams?

The n-gram model is the foundation and simplest form of natural language processing (NLP). Compared to the alternatives, an n-gram model does not require much tuning, only expansions of the logic. An N-gram model is a predictive text model; it tries to predict the Nth word based on the preceding words. As this model is more probability based, it struggles with some aspects of NLPs like text generation, instead finding its strength in text prediction.

Project Goal

The goal of this project was to implement a N-gram natural language processing model from scratch without using any machine learning libraries. After implementing a basic bigram and trigram model, further refinement and experimentation was done through the implementation of add-k smoothing, simple backoff, and linear interpolation.

Methodology

Dataset

Before any implementation can be done, a dataset needs to be selected. For this project, the goal was to find a dataset that was decently sized with options for expansion, wasn't tedious to parse, and was free to download. After some searching, the 1 Billion Word Dataset was selected. The dataset is large and split into subfiles allowing for easy expansion, the words were per-tokenized simplifying the remaining cleaning, and it was free to download and use. This project used the first eight subfiles with files, one, two, four, and five making up the training dataset; files seven and eight made up the validation dataset, and files three and six made up the test dataset.

Cleaning the Dataset

While the data was tokenized and separated by spaces when downloaded, the text was not ready to be analyzed and used for n-gram modeling. The text still had some messy characters and duplications that would need to be cleaned up before analysis.

```
There have been some exceptions -- such as Medicare in 1965 .
The government guidance will be reviewed early next year after a period of public comment .
It wasn 't the most seaworthy of prizes .
```

The cleaning process for the text corpus was simple but tedious. Numbers such as the 1965 above were converted to <num>. Instances of repeated punctuation were simplified to one character. A sentence start and end tokens <s> and </s> were added, turning the above into:

```
<s> there have been some exceptions - such as medicare in <num> . </s>
<s> the government guidance will be reviewed early next year after a period of public comment . </s>
<s> it wasn 't the most seaworthy of prizes . </s>
```

This process was done using regex strings in Python. They combed the text and when true turned the associated characters into the correct output. Below are two examples of the lines used, the first one turns all numbers into the <num> token, and the second replaces any unexpected characters with an empty space.

```
cleaned = re.sub(r'(?<![a-zA-Z])[0-9]+(\.[0-9]+)?', '<num>', line)
cleaned = re.sub(r'[^a-z0-9\s.,?!\'"():;\-<>]', '', cleaned)
```

With the dataset finally clean, it was able to be counted and set up for the n-gram model.

Vocabulary

With the clean dataset, a script was ran to count the instances of each word splitting the words by the spaces. This count would show the entire vocabulary in the training dataset, so a cutoff could be chosen. While every word in the training set could be used, it would have been inefficient, and many words only appeared once or twice. When all the words were counted by using a dictionary, the results were placed into a table and a file. The table shows the actual counts while the graph (See Figure 1) shows how the frequency of a word goes down exponentially.

Token	Count
the	3,336,222
,	2,839,096
.	2,450,414
<s>	2,450,288
</s>	2,450,288
to	1,476,812
of	1,407,382
a	1,331,100
and	1,308,068
in	1,195,886

Table 1

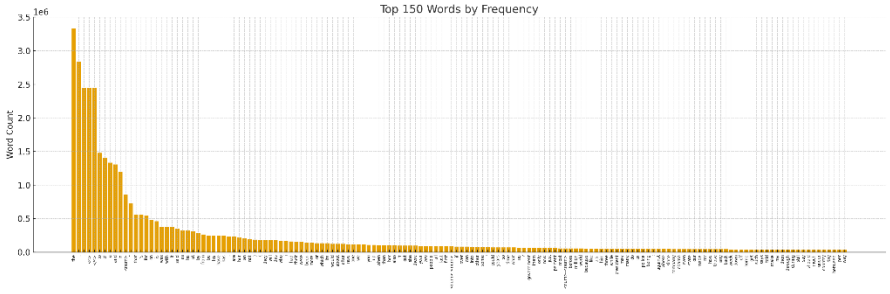

Figure 1

The total number of words in the vocabulary list started at 312,432 words, with majority appearing less than 5 times. To simplify the vocabulary list the cutoff was set to remove any words with less than appearances in the training set. The cut brought the vocabulary size down to 108,589 words.

Unknown Mapping

Once the cutoff was finalized there was one last step before model implementation could begin. In its current state the model wouldn't be able to handle words not in the final vocabulary list, so a script was ran to map all missing words to <unk>. This mapping made sure that all unexpected words could be handled predictably by the model.

Motivation

N-Grams

An N-gram model is a model that aims to predict the next word in a sentence by using the history of words before it. This can be explained in terms of the probability that the word *am* will follow *I*.

This can be expressed as $P(\text{am}|\text{I})$ and more generally as $P(w_n|w_{n-1})$.

In order to compute this probability, we can use the following formula: $P(w_n|w_{n-1}) = \frac{C(w_n w_{n-1})}{C(w_{n-1})}$.
Now both of these formulas can be used to find the probability of a bigram (2-gram), but the formulas can be expanded using the probability chain rule to become:

$$P(w_n|w_{1:n-1}) = \frac{C(w_n : w_1)}{C(w_{n-1} : w_1)}$$

This formula generalizes the process and works for any N grams. For this model due to the limited size of the subset, did not go higher than a trigram (3-gram) model (Jurafsky & Martin, 2025).

Counting N-grams

While there are differences in counting unigrams, bigrams, and trigrams, the method for storing them remains the same. The script went through each file in the training dataset and went token group by token group through the file and counted the respective N-gram. A dictionary was used to store the results with a tuple of the tokens as the key.

Unigrams

Unigrams were the easiest to count as it was just counting how many times each token appeared in the dataset. This means that the total count of unique unigrams is equal to the size of the vocabulary list and therefore 108,589.

Bigrams

Bigrams complicate the process slightly requiring a sliding window to be used. It starts from the second token on the line and updates each bigram by making a tuple out of the two tokens. The Bigram model is where the vocabulary list cutoff really starts to show its benefits and the count of unique bigrams is roughly 40 times larger than the unigrams. The count of unique bigrams is 4,296,741.

Trigrams

The trigram process is very similar to the bigram process just counting one extra preceding token. While the jump in size is not quite to the same extent as unigrams to bigrams, there is still an increase of roughly 3 times between two, making trigrams roughly 120 times larger than unigrams. The count of unique trigrams is 14,437,460.

How to Measure Model Accuracy?

With the counts of the N-grams we can now predict how likely a word is to follow another or a set of other words. To revisit an earlier example, the probability that **am** follows **I** for this training set is ≈48%. While this is good for spot testing, it does not give us a way to check the overall accuracy of the model. Realistically, the best way to test a model's accuracy is to use it and see the results, however, it is ill advised to put out a model without verifying its accuracy first. Enter Perplexity.

Perplexity

Perplexity is the measure of a model's ability to predict words in a dataset. Compared to extrinsic measures of a model's ability to predict, perplexity provides an intrinsic measurement. In simpler terms perplexity is how well the model predicts the next word continually for a text set, where a perfect model would predict with 100% accuracy and thus provide 1 for each word. As such, Perplexity can be explain as the roughly the inverse of the combined multiplication of all probabilities for a dataset to the square root of N where N is the total count of word tokens, or by the following formula:

$$perplexity(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i)}}$$

However, in practice, using this definition of perplexity does not work. When computing decimal numbers on a computer, they are stored as a floating-point number, and multiplying these floating points together can lead to errors due to how they're stored. The solution, move the operation into log-space. Perplexity can then be represented by this formula:

$$PP = e^{\left(\frac{-1}{N}\sum \ln P(w_i)\right)}$$

This makes it safe to calculate with computer operations. Despite the operation being safe, there is one edge case that does need to be handled: when the probability is 0. The solution is luckily very simple, simply take the max of the probability and a very small number; for this project 1e-12 was the low value used. This prevents log(0) errors while not heavily swaying perplexity outcomes.

```
def test_perplexity(path, d=0.0):
    tot_tokens = 0
    bi_sum = 0
    tri_sum = 0
    for file in read_directory(path):
        for line in read_file(file):
            tokens = line.split()
            t_len = len(tokens)
            tot_tokens += t_len
            if t_len < 2:
                continue
            for t in range(1, t_len):
                x = max(ngram_probability(2, tokens[t-1:t+1],delta=d), 1e-12)
                bi_sum += math.log(x)
            if t_len > 2:
                for t in range(2, t_len):
                    x = max(ngram_probability(3, tokens[t-2:t+1], delta=d), 1e-12)
                    tri_sum += math.log(x)
    return math.exp(-bi_sum/tot_tokens), math.exp(-tri_sum/tot_tokens)
```

Basic Model

A basic bigram or trigram model is quite simple to implement once the dataset is cleaned, it is only a matter of going through the text corpus calculating the probability of each word and creating a summation of their log sums, while tracking each token predicted. Then use the perplexity formula above to get the perplexity.

Add-K/Laplace Smoothing

A basic model has one big drawback, a probability of 0 means log of 0 which means a perplexity of infinity. One of the solutions to this is add-k or Laplace smoothing. This smoothing method aims to give each token a chance to follow every other token. Of course, this would naturally boost the most frequent words even more, so the probability is normalized against the vocabulary size. Add-k smoothing can be expressed as:

$$P(w_i) = \frac{C + k}{N + kV}$$

Where C is the count of the w_i following the preceding words, and N is the count of the preceding words up to the Nth word w_i. V is the vocabulary list size and k is the added probability (Hockenmaier, 2018).

```
def ngram_probability(n, string:tuple, delta=0.0):
    if n == 1:
        s = tuple(t.lower() for t in string)
        num = unigrams.get(s, 0)
        den = VOCAB_SIZE
        return num / den
    if n == 2:
        ng = bigrams
        lg = unigrams
    elif n == 3:
        ng = trigrams
        lg = bigrams
    else:
        return 0 #beyond model scope
    s = tuple(t.lower() for t in string)
    num = ng.get(s, 0) + delta
    den = lg.get(s[0:n-1], 0) + (VOCAB_SIZE * delta)
    if den <= 0:
        return 0 #avoid divide by zero
    return num / den
```

Basic Backoff

While add-k smoothing does improve overall model accuracy and reduces perplexity scores, it can struggle to truly capture the relationship between words. One solution to this problem is stupid/basic backoff. Instead of adding a chance of each word following every other word like add-k, backoff instead decreases model complexity by reducing N until a relationship is found. In practice this is implemented

as: test the trigram probability, if that probability is zero, try the bigram probability. If the bigram probability is also zero, default to the unigram and return that probability.

```
def backoff_perplexity(path):
    predicted = 0
    log_sum = 0.0
    for file in read_directory(path):
        for line in read_file(file):
            tokens = line.split()
            t_len = len(tokens)
            if t_len == 0:
                continue
            for t in range(t_len):
                if tokens[t] == '<s>': #avoids predicting <s> token
                    continue
                if t >=2:
                    x = ngram_probability(3, tokens[t-2:t+1])
                else:
                    x = 0.0
                if x == 0.0 and t >=1:
                    x = ngram_probability(2, tokens[t-1:t+1])
                if x == 0.0:
                    x = ngram_probability(1, tokens[t:t+1])
                x = max(x, 1e-12)
                log_sum += math.log(x)
                predicted += 1
    if predicted == 0:
        return float("inf")
    return math.exp(-log_sum/predicted)
```

Linear Interpolation

The alternative approach to backoff is linear interpolation. The core idea is that values from a lesser context can help determine the next word, or more clearly, the lower N-grams can be combined/interpolated with higher N-grams to help determine the Nth word. In practice, this can be represented by the following equation:

$$P(w_i) = \lambda_1 P_1 + \lambda_2 P_2 + \lambda_3 P_3$$

For linear interpolation in particular, the values of each lambda should add up to one in total. When implementing interpolation, to guarantee this condition, a normalization was added in case the lambdas added to more than one.

```
def linear_interpolation(window:tuple, l1, l2, l3):
    w = tuple(window)
    L = len(w)
    p1 = ngram_probability(1, (w[-1],))
    p2 = ngram_probability(2, w[-2:]) if L >= 2 else 0.0
    p3 = ngram_probability(3, w[-3:]) if L >= 3 else 0.0
    return l1 * p1 + l2 * p2 + l3 * p3
```

Implementing interpolation required writing a new probability handler and text parser to properly apply the weights to each probability, and to pass in a window so the probabilities could be computed for each function call.

```
def linear_interpolation_perplexity(path, l1=0.1, l2=0.3, l3=0.6):
    predicted = 0
    log_sum = 0.0
    total_l = l1 + l2 + l3
    l1, l2, l3 = (l1 / total_l, l2 / total_l, l3 / total_l)

    for file in read_directory(path):
        for line in read_file(file):
            tokens = line.split()
            t_len = len(tokens)
            if t_len == 0:
                continue
            for t in range(t_len):
                if tokens[t] == '<s>': #avoids predicting <s> token
                    continue

                window = tokens[max(0, t-2):t+1]

                x = linear_interpolation(window, l1, l2, l3)
                x = max(x, 1e-12)      # avoid log(0)
                log_sum += math.log(x)
                predicted += 1

    if predicted == 0:
        return float("inf")
    return math.exp(-log_sum/predicted)
```

Tuning Parameters

For add-k smoothing, and linear interpolation, there were parameters that could be tuned. The goal was to find the value that produced the lowest perplexity. Each function had an initial value which was lowered until the perplexity was higher than the previous result. Then, a couple numbers were chosen at random in the same ballpark as the lowest perplexity score number, to see if it could be lowered further. The best performing value was then saved.

Result Analysis

Basic Model and Add-k Smoothing

Add-k Performance on Validation Set

Smoothing (k)	Bigram Perplexity	Trigram Perplexity
0 (No Smoothing)	629.04	90,993.68
0.01	256.58	989.26
0.003	236.87	736.96
0.0015	234.15	659.75
0.001	234.69	631.19
0.0001	258.36	643.83

Table 2

Table 2 shows how various k values affect the perplexity of the validation set. The trigram and bigram models do diverge slightly with their lowest results, bigram being k=0.0015 and trigram being k=0.001. The overall winner is k=0.001 as the difference between 234.15 and 234.69 is minimal compared to the difference between 659.75 and 631.19.

Add-k Smoothing on Test Set

Smoothing (k)	Bigram Perplexity	Trigram Perplexity
0	630.45	91,339.75
0.001	234.89	631.85

Table 3

When the best k=0.001 is applied to the test set, the same noticeable decrease appears with both sets, the most noticeable being the over 90 times decrease in trigram perplexity (See Table 3). While these perplexity values are slightly higher than the validation set, they are reasonably in the same range. It is likely that the small differences come from minor differences between the two sets.

Add-K Smoothing Training Set

Smoothing (k)	Bigram Perplexity	Trigram Perplexity
0	114.74	15.18
0.001	128.67	51.76

Table 4

Looking at the training set numbers shows how the model overfits (See Table 4). The k=0.001 value decreases model accuracy and overfitting which matches the improved generalization on unseen data shown by the test and validation set.

Basic Backoff

Dataset	Perplexity
Training Set	21.14
Validation Set	65.94
Test Set	65.94

Table 5

Table 5 represents the perplexity values calculated for each dataset with the backoff model. This model overfits more on the training set compared to the bigram model but generalizes better than the trigram model (See Table 4). For the Validation and Test sets, the model performs distinctly better when compared to Add-k smoothing (See Table 2 and Table 3).

Linear Interpolation

λ_1	λ_2	λ_3	Train Perplexity	Validation Perplexity	Test Perplexity
0.10	0.30	0.60	9.08	20.18	20.20
0.05	0.30	0.65	12.14	32.1	32.13
0.10	0.35	0.55	9.34	20.27	20.28

Table 6

Tuning lambda for interpolation is more difficult than for add-k smoothing. The first result in table 6 shows the default lambda values and the best perplexity results. While attempts were made to find better values, no direction of movement was found to decrease the perplexity below the initial values.

Analyzing the Results

Best Perplexity Values

Model	Train	Valid	Test
Bigram	114.74	629.04	630.45
Trigram	15.18	90,993.68	91,3339.75
Add-k (k=0.001) Bigram | Trigram	128.67 | 51.76	234.69 | 631.19	234.89 | 631.85
Backoff	21.14	65.94	65.94
Interpolation (0.1, 0.3, 0.6)	9.08	20.18	20.20

Table 7

When comparing all the models implemented and tested for this project, interpolation stands out as the model with the lowest perplexity and therefore highest accuracy (See Table 7). Compared to the basic trigram model, interpolation saw a roughly 4500 times decrease in perplexity when comparing the test set results. Interpolation likely performed best as it allowed lower context probabilities to help fill spaces where trigrams were otherwise sparse, additionally bigrams likely reinforced the trigrams that were found increasing overall accuracy. Backoff, the runner-up struggles in this regard as it can only fall back to lower contexts and reinforce correct trigram sequences.

Discussion

Why Interpolation is the most Accurate

As seen in table 7, Interpolation provided the most accurate results with the lowest perplexity. This can be attributed to a couple factors. First, it has the benefit of combining multiple context lengths to reinforce correct predictions. Secondly, sparsity is mitigated by that same lower context allowing for approximations when a trigram is not found in the trigram list. These results also happen to line up with typical NLP research and findings, where the added context is proven to increase interpolation accuracy (Jurafsky & Martin, 2025).

Trigrams Crazy Perplexity

When looking at the initial perplexity for trigrams across the validation and test sets without any smoothing, the results are incredibly high. However, the training set shows a very small perplexity in direct contradiction. This can be explained by trigram sparsity in the validation and test data and overfitting in the training set. The model only has experience with trigrams that exist in the training data, however there are far more trigrams that exist compared to what is in the training set. When the trigram model fails to find the trigram in its set, the probability is set to zero. When the zero is used in the log functions, it produces an error and log of zero approaches infinity; the infinity value balloons the perplexity leading to a very high value. This explosion of perplexity shows why some form of smoothing is needed for generalizing NLP models (Jurafsky & Martin, 2025).

Smoothing Strength vs. Model Bias

Anyone who has studied machine learning or AI is likely familiar with the bias/variance trade-off. In this model, that trade off takes the form of the smoothing value k. A low k value can increase model complexity but increases the variance as less likely words are more likely to crop up due to the normalization of the k value; meanwhile, a high k value leads to model bias, and the model is over smoothed providing a too simplistic representation of each word's probability.

Why Backoff did not Perform Best

The backoff model provided notable improvements when compared to the add-k model, however it did perform better than interpolation. This can be explained simply by comparing how the two models handle lesser contexts. Backoff uses lower contexts only when a higher context does not exist, otherwise throwing them out, while interpolation uses lower contexts to reinforce its probabilities. This one difference explains why interpolation performed better than backoff. However, backoff still proved stable and consistent for calculating rough probabilities as it does have means for finding the Nth word when the trigram does not exist.

N-Gram Limitations

N-grams are one of the core models of NPL. However, N-grams do not come without their own weaknesses. N-grams have a limited view of a sentence, only having context as far back as N words, which is often not enough for most sentences. Additionally, the higher the value of N, the more N-grams you have to store, for example the just from just over 100k unigrams to 4 million bigrams for this small subset of data making large models spatially expensive. On top of that expansion, the higher the value of N the sparser some of the N-grams can be, leading to the same issue that can be found with the vocabulary where a large number of N-grams only have one or two occurrences throughout the whole dataset.

Generative Limitations of N-grams

This project attempted to explore basic sentence generation using bigrams, trigrams, and interpolation. These attempts showcased a major limitation of N-gram models, generation strength. While N-grams can be used to generate sentences that sound vaguely natural, they cannot create an understandable sentence without a large amount of luck. For generative purposes, the trigram and bigram actually outperformed the interpolation model, making far more understandable sentences. This can be explained by the interpolation process which increases the chances of unigram words unnaturally representing the more common words used in English; these words often lack the meaning that drives a sentence leaving instead a collection of words that does not sound natural at all.

Bigram Generator: quip from sole-source contracts requiring followup column

Trigram Generator: mini-bus packed with ball state cardinals stayed unbeaten by wearing

Interpolation Generator: from of to about is of for at to are and be of his with an in out <num> had

Conclusion

N-gram models are a foundational part of natural language processing and can provide great insight into how language can be reconstructed and predicted using probability. An N-gram model helps build the foundation required for more advanced natural language processing and improves understanding of how language is constructed by analyzing generative outputs. N-grams also show how data sparsity can affect computational results and shows remedies that can limit their impact.

This project focused on constructing an N-gram from scratch without the use of external libraries. In the process, multiple models were built including a basic bigram and trigram model; those models were then expanded with add-k/Laplace smoothing to improve their results. The project then expanded to include interpolation and backoff which were compared against the earlier models where interpolation was found to be the best model.

Should this project be continued further, expansions could be made to add neural components, as well as vector embeddings, to improve semantic connections and context retention.

References

Chelba, C., Mikolov, T., Schuster, M., Ge, Q., Brants, T., Koehn, P., & Robinson, T. (2013). *One billion word benchmark for measuring progress in statistical language modeling*. Google Research. https://www.statmt.org/lm-benchmark/

Hockenmaier, J. (2018). *Lecture 4: N-gram language models* [Lecture slides]. CS 447: Natural Language Processing, University of Illinois at Urbana–Champaign. https://courses.grainger.illinois.edu/cs447/fa2018/Slides/Lecture04.pdf

Jurafsky, D., & Martin, J. H. (2025). *Speech and language processing* (3rd ed., Draft of Chapter 3). Stanford University. https://web.stanford.edu/~jurafsky/slp3/3.pdf