



# Word Predictor

DD1418 Language Engineering with Introduction to  
Machine Learning

Almir Aljic & Alexander Jakobsen  
2019-12-13



# Task

- Predict words based on previous/historical user input. Word prediction software is most commonly used on smartphones, which enables the user to finish their text messages or other inputs quickly.
- A good word predictor also...
- checks for spelling errors, and
- learns new words the user is typing.

**It is particularly important to update the completion words with each new keystroke the user makes! More on this shortly.**



## Background: Introduction

- Two primary domains of interest: 1) n-gram probabilities and 2) correcting spelling errors. Let's start with n-gram probabilities!
- Google has a very large data set containing n-gram probabilities (up to 5-grams available in different languages/dialects.
- We did not use any of Google's data for this project due to its vast size, which requires time and computing power. Unfortunately, we did not discover an API to Google's N-gram Viewer either. More on the n-gram viewer and our own implementation shortly.
- Google's vast data set has been made public and is available for download [here](#).



## Background: N-gram probabilities (Markov Assumption)

- We can use n-gram probabilities to determine the most probable continuation of a sentence. Assumption: a word only depends on its n preceding words.
- $P(w_i | w_{i-n}, \dots, w_{i-1})$  is the conditional probability that word  $w_i$  will appear after the sequence given by  $w_{i-n}, \dots, w_{i-1}$ . **We call this the n-gram probability that word  $w_i$  will succeed the given sequence.**
- Example:  $P(\text{"you"} | \text{"how are"})$  denotes the conditional probability of “you” appearing after “how are”.
- How do we compute these probabilities, given a large training data set (training corpus)?



## Background: Maximum Likelihood Estimation (MLE)

- By counting the number of occurrences of the two sequences given by  $w_1, \dots, w_{i-1}$  and  $w_1, \dots, w_i$ , we can estimate the aforementioned conditional probability as follows:
- $P(w_i | w_1, \dots, w_{i-1}) = \frac{c(w_1, \dots, w_i)}{c(w_1, \dots, w_{i-1})}$ , where  $c(\text{<sequence of words>})$  denotes a count() function.
- Example: Applying MLE to the sequence “How are you” using a hypothetical training corpus could yield:
- $P(\text{“you”} | \text{“how are”}) = \frac{c(\text{“how are you”})}{c(\text{“how are”})} = 2/4 = 0.5$
- In other words, “you” appeared after “how are” 50% of the time.



## Background: Counting trigrams & take-away

Given a large training corpus, containing many lines;

“They are here to ask questions about his disappearance ...”

First, we collect all the bigram counts:

$c(\text{“They are”})$ ,  $c(\text{“are here”})$ , ...,  $c(\text{“his disappearance”})$ .

Then, all the trigram counts:

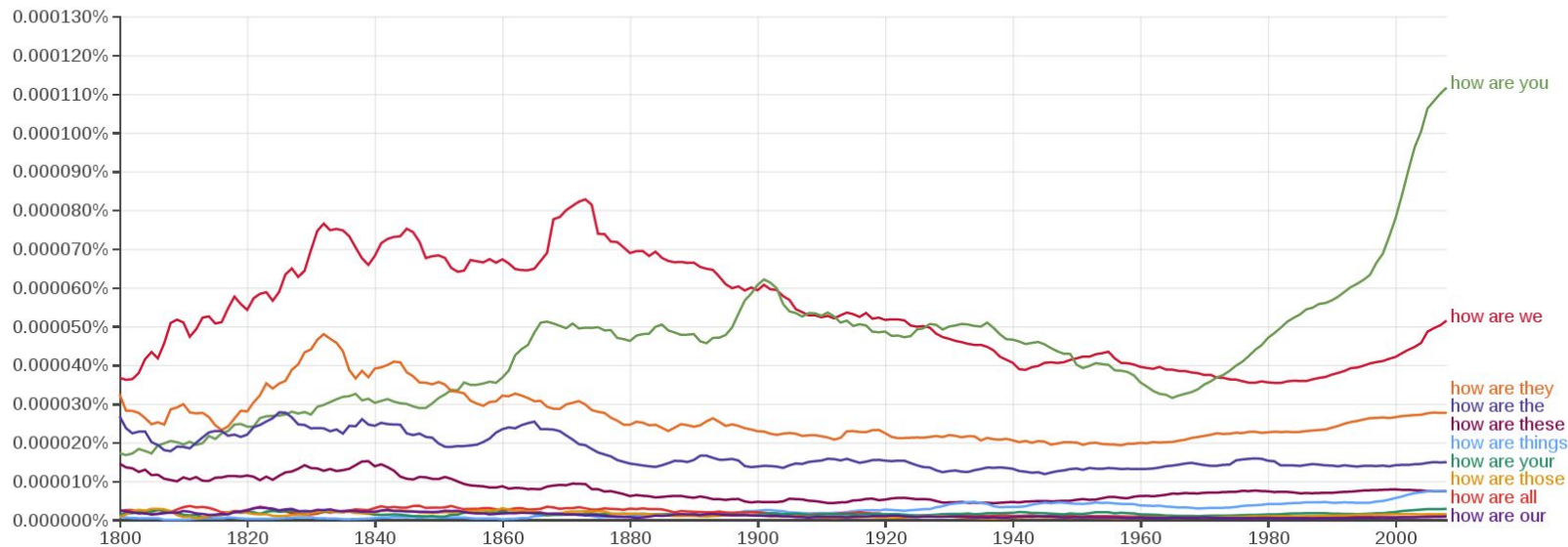
$c(\text{“They are here”})$ ,  $c(\text{“are here to”})$ , ...,  $c(\text{“about his disappearance”})$ .

Let’s say  $c(\text{“They are”}) = 1000$ ,  $c(\text{“They are here”}) = 300$  and  $c(\text{“They are not”}) = 700$ .

Then,  $P(\text{“here”} | \text{“They are”}) = 0.3$ ,  $P(\text{“not”} | \text{“They are”}) = 0.7$ . **So, if the user types “They are”, we should predict that the user wants to write “not” next.**

# Background: Google's N-gram Viewer

Graph these comma-separated phrases:  ☐ case-insensitive  
between  and  from the corpus  with smoothing of  [Search lots of books](#)





## Background: Correcting spelling errors

Given some user input, how do we determine that the input is a

- A) misspelled word and
- B) which word the user meant to type?





## Background: Correcting spelling errors

Assume the user has typed “how are thye”. We note that “thye” is a non-existent word, meaning it does not exist in our **vocabulary** (which was generated from our training corpus by recording all unique words). In other words, we have no recommended words to complete the sentence with. If the user had written “how is your mo” we might recommend the words “mom” and “mother” as completion words, assuming that both “mom” and “mother” appear in our vocabulary.

Since we have no words to recommend given the user input “thye”, we can deduce that the user has either misspelled or has written a new word we have not heard of before (word is not in our system’s vocabulary).

## Background: Correcting spelling errors

In essence, if the user inputs a sequence of characters (which are meant to constitute a word) that do not exist in our vocabulary, we suspect the user might have misspelled. How do we deal with this?





## Background: Correcting spelling errors

Premise: User has typed a non-existent word  $w$ . Algorithm:

for each word with edit distance less than or equal to  $m$  from  $w$   
    if word in vocabulary  
        recommend word to user



## Background: Correcting spelling errors

Premise: User has typed a non-existent word  $w$ . Algorithm:

For each word with edit distance  $m$  from  $w$   
    if word in vocabulary  
        recommend word to user

**You might be thinking:** how do we define the edit distance, and how do we choose  $m$ ?



## Background: Correcting spelling errors

You might be thinking: how do we define the edit distance, and how do we choose  $m$ ?

The edit distance may be defined as the Levenshtein distance, also known as the Minimal edit distance. Most commonly, a substitution/insertion/deletion has a cost of one. In other words, the minimal edit distance from “thye” to “the” is 1: the ‘y’ in “they” has to be removed.

According to Damerau (1964), a majority of spelling errors (in text written using a keyboard) stem from: 1) two neighboring characters being confounded 2) one character being left out from the word 3) an additional character that does not belong being inserted 4) one correct character being replaced by an incorrect character.



## Background: Correcting spelling errors

In other words, if  $m = 2$ , all of the aforementioned reasons for the occurrence of spelling errors can be amended.

Back to our example: User has written “thye”. Possible permutations of the word using a maximal edit distance of 2 ( $m = 2$ ) are given by

[“thy”, “the”, “tye”, “hye”, “they”, “tyhe”, ...]

From the list above, we see that “the” and “they” are the only words that are actual words, and which ought to be in our vocabulary. **But how do we rank the words in our recommendation to the user?**



## Background: Correcting spelling errors

But how do we rank the words in our recommendation to the user?

There are, of course, different ways of doing this. One way is to check which word is most likely to appear at any given time. In other words, if  $P(\text{"the"}) > P(\text{"they"})$ , and we can only recommend one word to the user, then we should recommend the word "the".

Note that

$$P(\text{"the"}) = c(\text{"the"}) \div N$$

$$P(\text{"they"}) = c(\text{"they"}) \div N$$

where  $N$  = total number of words recorded in the training corpus.



# Hypothesis

Our program will predict words quite accurately, so long as we use a large training corpus containing text on different topics/from different sources/authors and an adequate maximal edit distance is chosen for the spelling correction algorithm.





# Implementation

From assignment 2, we extended BigramTrainer.py to output a file containing not only bigram probabilities, but also trigram probabilities, in the way described previously. This file will be known as containing our **language model**. The structure of the file is given below:

<Number of unique words in training corpus> <number of total words>

<i> <word<sub>i</sub>> <frequency of occurrence of word<sub>i</sub>>

...

<i> <j> <P(word<sub>j</sub> | word<sub>i</sub>)>

...

<i> <j> <k> <P(word<sub>k</sub> | word<sub>i</sub>, word<sub>j</sub>)>



# Implementation

The main class, found in WordPredictor.py, reads the language model as follows:

- Reads all unique words, and stores them in two class variables: `self.index` and `self.word`.
- Example: `self.word[0]` might return “i” if “i” was the first unique word found in our training corpus. `self.index[“i”]` would then return a 0. These variables are dictionaries, which enables efficient look-up. `self.unigram_count[“i”]` would store the number of times “i” appeared in the training corpus.
- All bigram/trigram probabilities are stored in `self.bigram_prob` and `self.trigram_prob` respectively. An example of retrieving  $P(\text{“to”} | \text{“i like”})$  is: `self.trigram_prob[“i”][“like”][“to”]`. The variables are nested dictionaries.



# Implementation

Our implementation illustrated using an example:

Let's say the user has already written "How are", so we have a class variable `self.words = ["How", "are"]`.

Assume the class variable `self.num_words_to_recommend = 3`. Then, we recommend the top 3 words with the highest probability, based on  $P(\text{word} | \text{"How are"})$ . So, perhaps, "you", "they", "we", "some" are most likely to appear after "How are", in the given order. Then, we recommend the first three words, so we exclude "some".

For each keystroke, we update the words we recommend. If the user types ["How", "are", "t"], then we only recommend words that begin with 't'. So, in this case, "they".



# Implementation

Assume, however, that the user actually intends to type “How are termites...”. So the next word the user wants to write is “termites”. Additionally, assume that our language model has not recorded any trigram probabilities of the form  $P(\text{“termites”} | \text{“How are”})$ .

In other words, say the user has written [“How”, “are”, “te”]. We have no words to recommend from our trigram probabilities, because no word in the dictionary `self.trigram_prob[“How”][“are”]` starts with “te” (remember, we didn’t find any occurrences of “How are termites” in our training corpus, so that trigram probability does not exist in our language model - this is a problem of **sparse data**).



# Implementation

We solve this problem by applying a method known as **backoff**:

So, because our language model says there are no trigram probabilities, we simply use (n-1)-grams instead, so in this case we check our bigram probabilities instead. In practice, we check all words following “are”, which is given by the dictionary `self.bigram_prob[“are”]`, that begin with “te”. The word with the highest conditional probability of appearing is returned as the top recommendation.

If there are also no bigram probabilities, we simply check which word (that begins with “te”) has the highest unigram count.



# Implementation

When the user inputs a space, or chooses from the list of recommended completion words, we know it's the end of the word.

**In some cases, there won't be any words that follow the words the user has written thus far. What do we do then?**

Say the user has written `self.words = ["How", "are", "thye"]`. There are no completion words to recommend, i.e. "thye" is not the beginning of a word nor is it a word in and of itself. Then, we apply the spelling correction algorithm, and produce a list of all possible words with edit distance less than or equal to 2 ( $m = 2$ ) from the word "thye". We then filter out the nonsense words from the words that exist in our vocabulary. Based on this list, we then select the top alternatives with highest frequency of occurrence to recommend as completion words.



# Implementation

Let's say the user writes ["How", "are", "thye"], and then inputs a space. Remember, "thye" is not in our vocabulary, so according to our system of word prediction, "thye" is not a word. What do we do?

Well, we simply add "thye" to `self.index`, `self.word` and `self.unigram_count`, essentially adding a new word to our vocabulary. For the sake of simplicity, we do not alter the bigram and trigram probabilities, nor do we save this information between sessions (so if you quit our program, new words you have typed will not be saved). Although in a real world implementation, this is easily accomplished by writing a new text file/storing the information in a local database and then being able to fetch the information as the user types.



# Data

For our implementation, we used guardian\_training.txt from assignment 2 to generate the language model. This is a relatively large set of data, containing close to 9 million total words/tokens and almost 200 000 unique words.

*Short note on cross-entropy: we used our results from problem 3 in assignment 2 to assume that the language model would “learn well” from the training corpus. Note that this is not completely certain, because the results from that assignment were specific to bigrams. We have not computed the cross-entropy with respect to trigrams! We did not do this because this project was mainly about demonstrating the concept of how a word predictor could be implemented, and not necessarily about determining the validity of a language model. Although the two are interrelated, we did not consider the latter to be a necessity.*





## Results: Demo

Our results are best illustrated through a demo! First, generate the language model file by running

```
python3 TrigramTrainer.py -f guardian_training.txt -d language_model.txt
```

Then, run the program itself

```
python3 WordPredictor.py -f language_model.txt
```



## Results: Demo

You will now be greeted by the following:

```
aljica@debian:~/Desktop/spraktek/proj$ python3 WordPredictor.py -f
```

```
1 - The
```

```
2 - I
```

```
3 - It
```

```
Enter your letter (or choice): 
```



## Results: Demo

Recommended words can be selected by inputting “1-”, “2-”, “3-” etc. Let’s type “How many” and see which recommended words we get:

```
How many _  
1 - times  
2 - people  
3 - of  
Enter your letter (or choice): 
```



## Results: Demo

When we input a 'p', notice how only words that start with that letter appear among the recommended words:

```
How many p_  
1 - people  
2 - places  
3 - parents  
Enter your letter (or choice): 
```



## Results: Demo

And this is an example of how the spelling correction works (note: the sentence is not coherent, but that does not matter for the sake of merely illustrating this):

```
How many people. How many yaers_  
1 - years  
2 - year  
3 - yes  
Enter your letter (or choice): 
```



## Results: Demo

Demonstration of how the system learns new words:

```
I have a $100_  
1 - 10  
2 - 100  
3 - 100g  
Enter your letter (or choice): 
```



## Results: Demo

Demonstration of how the system learns new words:

```
I have a $100. Do you have $_  
1 - $  
2 - $100  
Enter your letter (or choice): 
```



## Results

In general, the results are difficult to measure quantitatively. What we can do, given a large data set (corpus), is split it into a training and a test set. Then, we can compute the cross-entropy of the test set on the language model generated from the training set. The lower the entropy, the better the language model learned from the training corpus. This is known as *intrinsic evaluation*.

Other than that, a decent way of determining the results is through real world use. This is known as *extrinsic evaluation* and is the primary evaluation method used in this project.




# Conclusions - data sparsity & backoff



There are problems associated with the methods used in this project. One of those is the problem of **data sparsity** which comes with n-gram models (and MLE). Data sparsity means that many sensible word sequences will have zero probabilities. Fortunately, there are ways of dealing with this, some perhaps better than others. In this project, **backoff** was used to handle zero probabilities, mostly for its simple nature and thus easy implementation.

One important problem with backoff is that it may assign too high a probability to an inappropriate completion of a given sequence. Say the user has written “We like jum”. Possible words that start with “jum” are [“jump”, “jumping”]. Assume, however, that there are no occurrences of the trigrams “We like jump/jumping” or bigrams “like jump/jumping”. Given the context, “jumping” should be recommended, but perhaps  $P(\text{“jump”}) > P(\text{“jumping”})$  and so “jump” would be inappropriately recommended.

# Conclusions - final thoughts



Other flaws/things to think about:

- Are trigram probabilities really appropriate given the size of guardian\_training.txt? Should we have a larger training corpus?
- The code is not perfect and has some inefficiencies which can be amended.
- Perhaps it is better to store the language model in a local database? A local database also makes it easier to add new words to the vocabulary and to update bigram/trigram probabilities.

In practice, however, using basic extrinsic evaluation, the word predictor appears to work quite well. If you would like to try and experiment with it, we are more than happy to share it with you!



**Thanks!**