

Master of Science (MS) in Data Science

Module: ITC6010A1 – Natural Language Processing

Instructor: Dr. Lazaros Polymenakos

Student’ name: Alkiviadis Kariotis 241735

Term: Spring Term 2023

Type: HW#2 Individual Assignment

Submission Date: Friday, May 26

Words: 556

Table of Contents

[Introduction 3](#_Toc136022203)

[Building the Language Model 3](#_Toc136022204)

[Smoothing 3](#_Toc136022205)

[Building the Spell Checker 3](#_Toc136022206)

[Results 4](#_Toc136022207)

[Conclusion 4](#_Toc136022208)

# Introduction

In this project, I built a trigram language model and a spell checker using the Reuters dataset and a collection from the Gutenberg project. The language model was trained on a corpus of sentences and used to calculate the probability of a sentence. The spell checker used the language model and an edit model to suggest corrections for misspelled words in a sentence.

# Building the Language Model

The language model was built using trigrams, which are sequences of three words. The probability of a sentence was calculated as the product of the probabilities of its trigrams. Each trigram probability was calculated as the count of the trigram in the corpus divided by the count of the bigram formed by the first two words of the trigram.

The language model was implemented in Python using the Natural Language Toolkit (NLTK) for tokenization and the collections module for counting trigrams and bigrams. The model was trained on a corpus of sentences, which were tokenized into words using NLTK's word\_tokenize function. The model was evaluated using perplexity, a common metric for language models. Perplexity is the exponentiation of the average negative log probability per word. Lower perplexity means a better model.

## Smoothing

Smoothing was applied to handle trigrams and bigrams that did not appear in the training corpus. I used Laplace smoothing (also known as add-one smoothing), which adds one to the count of each trigram and bigram. This ensures that the probability of any trigram or bigram is never zero.

## Building the Spell Checker

The spell checker was built using the Noisy Channel Approach. For each word in a sentence, if the word was not in the vocabulary of the trained model, the spell checker generated a set of candidate words by performing delete, transpose, replace, and insert operations on the word. It then selected the candidate with the highest trigram probability and lowest edit distance to the original word as the corrected word.

The spell checker was implemented in Python using the nltk and collections modules. The edit model was trained on a file of edit counts, which was used to calculate the edit distance between words.

## Results

The language model and spell checker were tested on the Reuters dataset and a collection from the Gutenberg project. The perplexity of the model was calculated on both the training and test sets.

For the Reuters dataset, the perplexity of the model on the training data was 6.020840213995279, and the perplexity on the test data was 132854.35510278275.

For the Gutenberg project, the perplexity of the model on the training data was 14860.635906644056, and the perplexity on the test data was 14461.73760696.

The spell checker was able to correct misspelled words with a reasonable degree of accuracy. However, it did not handle words that were spelled correctly but used incorrectly, and it may not generate the correct spelling if the edit distance is more than 2.

# Conclusion

In this project, I was called to build a trigram language model and a spell checker using Python and NLTK. The model and spell checker performed reasonably well on both the Reuters dataset and a collection from the Gutenberg project. Future work could involve using more sophisticated methods for smoothing and candidate generation, and extending the spell checker to handle context beyond trigrams and words that are spelled correctly but used incorrectly.