Final Report
Text Inference Application

Jordi Castro
University of Arkansas
jc163@uark.edu

1 Problem Statement

Natural language processing (NLP) is a branch of artificial intelligence (AI) that is concerned with processing and interpreting data derived from natural language [7]. Text inference (TI), or text auto completion is a subsection of NLP that has gotten significant attention in various years. Often, TI is seen when responding to emails (notably Outlook and Gmail), Search engines giving users possible queries based on popularity and context, and GitHub Copilot being able to auto-fill the rest of your code block.
While text inference models already exist for a variety of useful tools, there isn't a specific AI-powered note taking application that assists in auto completion using text inference and information retrieval. The purpose of this paper is to introduce detailed experiments that will serve as the starting point in achieving that goal. Moreover, within this paper, experiments involving N-grams and NLP transformers will be reported, along with the full-stack application process: mock-ups, design details, front end and back end code to create and deploy a text inference application.

2 Literature Review

2.1 Brilliant

Brilliant offers a variety of interactive, online courses in mathematics and science. One of the notable courses offered is 'How LLMs [Large Language Models] Work'. This course introduces concepts of N-grams and provides Python code for further understanding. N-grams are a fundamental and simplistic technique in NLP. They are a sequence of letters in a particular order. When you process a body of text (corpus) in the format of a N-gram, you are able to get inference abilities.

2.2 Bigram

For example, say you process a body of text into bigrams, a sequence of two words (tokens). If you input one word (the key) into the processed text, the output would be the mapping of the input word to all the adjacent output words seen in the body. Hence, with a bigram model and input word "Romeo", one would expect to see the inference "and". Generalized, N-grams take in $N-1$ words as context to generate the next word ($(N-1)+1$ inference word $= N$). A bigram takes in only one word of context, so it often generates senseless strings of text.

2.3 Higher Order N-grams

On the other hand, if you choose a 10-gram with a small corpus, you might not have a nine-word sequence that exactly matches the context. And as such, higher N-grams tend to quote the corpus directly, which is useful in information retrieval but not useful if you want variable text generation.

Hence, a compromise between too much context and too little context is needed to generate a variably accurate model.

3 Methods

3.1 How N-grams Work

N-grams function primarily using a `successor_map`. The successor map is a list of key-value pairs, where the key is the context (a single string for bigrams, a tuple of strings for N-grams), and the value is the word that proceeds the contexts in the corpus. The implementation from Brilliant had multiple instances of the same value for a given key, which ultimately worked but seemed inefficient, especially as more and more duplicate key-value pairs were added to the successor map. To fix this redundancy, I changed the structure of each value of the successor map to be a tuple with the value and its frequency. That way, the selection of the next word from the key's list would remain weighted while saving memory and lookup time.

Brilliant's Bigram: `('romeo': ['and', 'and', 'or', ...])`

My Bigram: `('romeo': [('and', 7), ('or', 4), ...])`

3.2 Tokenizing the Corpora

To clean the corpus text of any unwanted characters before processing the text into the successor map, the python `.strip()` and `.lower()` methods are used. Taking a look at the cleaned corpus, there were few tokens that were not processed correctly. In the future, the use of a Python tokenization library, such as SpaCy [1], would be beneficial.

3.3 Building the Successor Map

To build the successor map, each word is cleaned of punctuation and inserted into the window. The window is a list of strings of length N. The first $N - 1$ words are the key (context), and the last word is the value (next adjacent word) of the successor map. The cleaned words are pushed onto the window until the window is full. Following a full window, the window is processed. If the key does not already exist, it's key-value pair is inserted into the map. If the key exists, but the value does not, the value is added to the list of values of that key. If the key exists and the value also exists, the frequency of the value is incremented by one. Finally, the window "slides" through the text by popping the oldest (left-most) word and continuing processing the corpus.

3.4 Saving the Successor Map

To save the successor map after processing a corpus, a custom write method was created. This method converts the object into JSON and stores it into a dynamic path given the N-gram. For example, if the successor map processed the corpus using a 4-gram, the file would be written and stored in `data/grams/4gram/shakespeare_4gram.json`.

3.5 Loading the Successor Map

Loading the successor map relies on `json.load` to load the JSON from the file path. For N-grams greater than 2, the keys need to be serialized back into tuples, which is handled gracefully in my custom method.

3.6 Adding More Corpora to the Successor Map

To add more corpora to the successor map, the previous N-gram successor map is loaded. Using the loaded model, more key-value terms are added, increasing frequencies of already existing pairs. Finally, the new successor map is saved using the write method.

3.7 Photopea [4]

Photopea is a free web-app photo and graphics editor. It has similar functionality to Adobe Photoshop. Leaning into the full-stack process, I wanted to make the website more personal by adding a logo. The logo was created using Vectorpea (similar to Adobe Illustrator) and Photopea, and it is a cartoon head of William Shakespeare, nicknamed 'Shakey'.

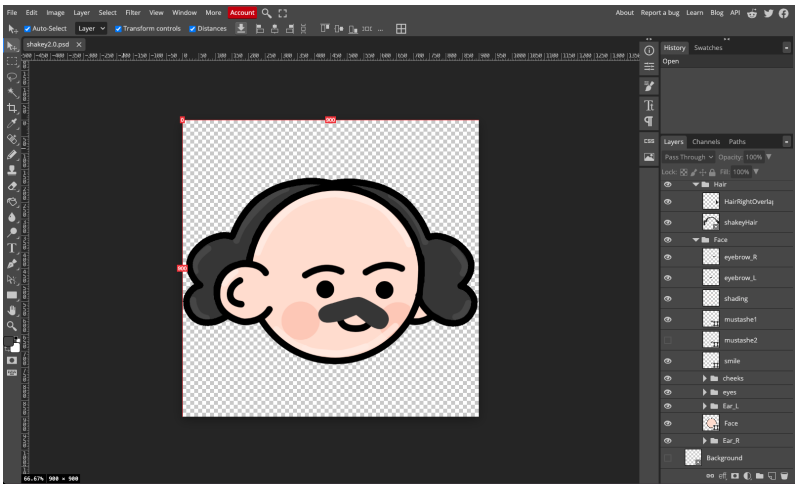

Figure 1: Using Photopea to create the logo.

3.8 Figma [2]

I began the rough outline of the website by viewing the Streamlit documentation [6] and seeing all the built-in components I had at my disposal. Next, I sketched what I wanted my web application to look like. Finally, I used my sketch to create the mock-up using Figma. This mock up was used when programming the user interface (UI) of the website to ensure a robust, intuitive user experience.

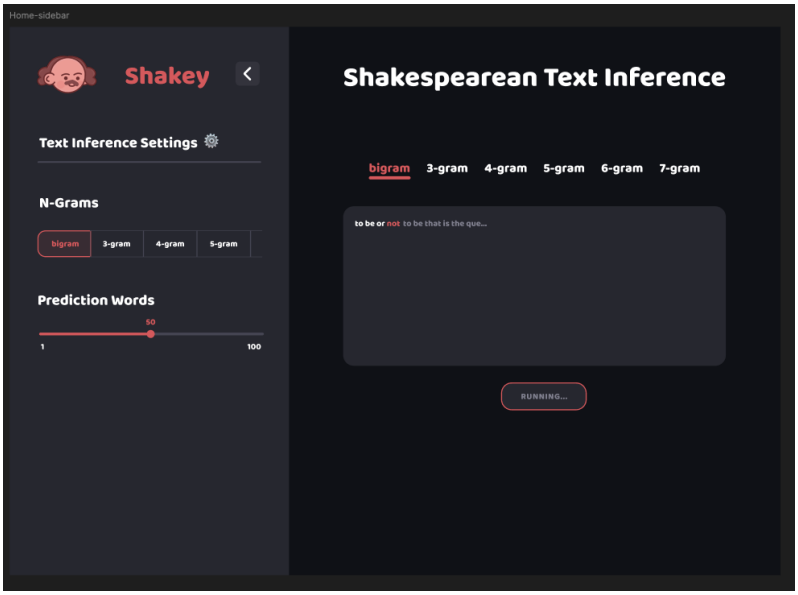

Figure 2: Figma Mockup.

4 Experiments

During the initial experimental phase of my project, I tested my N-gram code by creating successor maps of N-grams two through seven using a total of ten plays by William Shakespeare. Eventually, I added a total of 20 plays. The data was collected from MIT [3], and the plays used and their categories are listed below:

Table 1: Trained Shakespeare Plays and Their Categories

Play Title	**Category**
Romeo and Juliet	Tragedy
Hamlet	Tragedy
Macbeth	Tragedy
King Lear	Tragedy
Othello	Tragedy
Julius Caesar	Tragedy
Twelfth Night	Comedy
A Midsummer Night's Dream	Comedy
Much Ado About Nothing	Comedy
The Tempest	Comedy
As You Like It	Comedy
The Merchant of Venice	Comedy
The Winter's Tale	Comedy
The Comedy of Errors	Comedy
The Taming of the Shrew	Comedy
King Henry IV Part I	History
King Henry IV Part II	History
The Life and Death of King John	History
Richard II	History
Henry V	History

4.1 Initial Experimental Results

When testing these models, the major timing came from loading the pre-trained n-gram successor maps from the files. A general trend demonstrated that the larger the N-gram, the larger the file, and the longer it took to load the successor map from that file. The average size and load time for each N-gram model trained on ten versus twenty plays is below:

Table 2: Successor Map Size for Models Trained on 10 and 20 Plays

N-gram	**10 Plays**	**20 Plays**
Bi-gram	15,190	21,792
3-gram	121,775	211,044
4-gram	209,608	398,731
5-gram	230,626	453,011
6-gram	233,883	461,931
7-gram	234,552	436,014

Table 3: Time to Load for Models Trained on 10 and 20 Plays (Seconds)

N-gram	**10 Plays**	**20 Plays**
Bi-gram	0.03	0.06
3-gram	0.73	1.27
4-gram	1.46	2.81
5-gram	1.82	3.69
6-gram	2.08	4.27
7-gram	2.32	4.45

Loading times vary between local inference and inference on the Streamlit application

5 Evaluation

5.1 Top-k Neighbors

Top-k neighbors is a common evaluation metric in Natural Language Processing and Machine Learning. It is used to evaluate the performance of a model by checking if the correct answer is among the top k predictions made by the model.

5.2 Evaluation Setup

In this experiment, 100 quotes from ten plays not in the trained corpora were used to test the N-gram models. The selection of the plays were made while keeping an equal ratio of categories to ensure the testing encapsulated the breadth of Shakespearean language. Sourcing the quotes outside the trained body challenged the models and ensured no leakage such that the models would be tested on their ability to speak "Shakespearean English" and not recite quotes. The quotes were sourced from Royal Shakespeare Company (RSC) [5] and were chosen such that character names and locations pertinent to a particular play were omitted to ensure fairness. The tested plays are listed below.

Table 4: Tested Shakespeare Plays and Their Categories

Play Title	**Category**
Titus Andronicus	Tragedy
Timon of Athens	Tragedy
Coriolanus	Tragedy
Antony and Cleopatra	Tragedy
Two Gentlemen of Verona	Comedy
Troilus and Cressida	Comedy
Pericles, Prince of Tyre	Comedy
Richard III	History
Henry VIII	History
King Henry VI Part III	History

5.3 Logic Equations

For each quote in the ten unseen plays, each model returns a list of the top k results, sorting the successor map of the context by descending frequency. After the model makes a guess, the true positive, false positive, and false negative counters are adjusted.

```
if target in guess_words:
    TP += 1 # correct guess
else:
    FN += 1 # incorrect guess

for guess in guess_words:
    if guess != target:
        FP += 1 # false positives
```

Figure 3: Evaluation Logic for Top-K Neighbors.

As seen above, if the target word is in the list of guess words (of size k), then the model guessed correctly, and the TP variable is incremented. If the target is not in the list of guesses, a false negative

is recorded. Regardless of a hit or miss, the false positive is incremented for each guess in the guesses list that is not the target. For example, if $k = 3$, and the target is found in the guessses, the FP variable is still incremented by 2: (k-1). On the contrary, if the target is not found, every guess in the guesses list is incorrect, so the FP variable is increased by 3: (k). After the models were tested on the quotes, the precision, recall, and F1-measure (harmonic mean of precision and recall) were computed using the values of each counter. The equations can be found below:

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{TP}{TP + FN}$$

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

6 Evaluation Results

The models were evaluated for $k = 1$, $k = 3$, and $k = 5$. From the measurements, it was clear that N-grams five through seven were incapable of passing the evaluation tests. This is due largely to the fact that these higher N-gram models require more context to guess a word, and, when introduced to quotes not seen in the training body, the models struggle to find any successors for the multi-word key. Despite this drawback, the lower order N-grams (2-4) were able to perform slightly better. The accuracy of the models trended to increase from bigram to 3-gram, where it peaked on the 4-gram and hastily declined thereafter. The clear winner is the 3-gram model, with a low inference time and an 8.45% precision and an 11.50% F1 score. Evaluation metrics were measured for $k = 1$, $k = 3$, and $k = 5$.

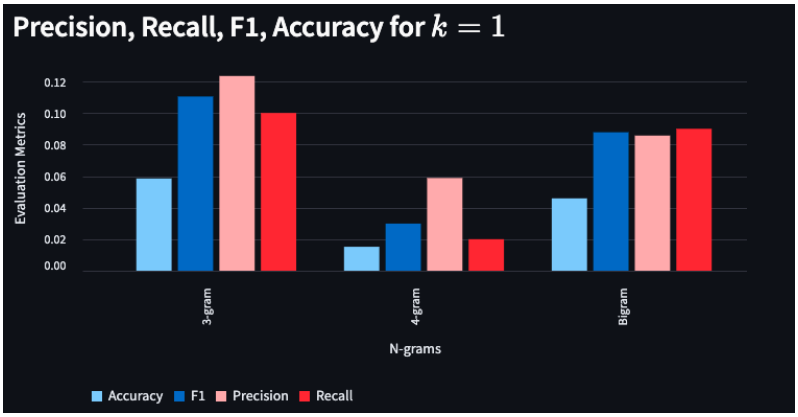

Figure 4: Evaluation Metrics for $k = 1$.

7 Streamlit[6]

Streamlit is the web-application framework used to build this application. The following subsections offer insight on the important components that make up this website.

7.1 Sidebar

The collapsible sidebar contains the text inference settings. The segmented control component allows users to seamlessly switch between N-gram models. The slider allows users to intuitively change the number of prediction words they want the model to return.

7.2 Main

The main code contains the title and a large text area box and button for the user to type and submit their query.

7.3 Session State & Running a Query

The Streamlit hook, `st.session_state` enables state to be present throughout the application. This is used when passing the values of the text inference settings from the sidebar to the main page. The submit button triggers a callback function that starts a spinner and collects the values of the N-gram type and the number of predicted words. These values are passed as arguments in the `query.sh` script. This script calls the `script.py` script, which uses the arguments to load the correct N-gram successor map and return the result with specified length to the Streamlit code. Python's `Subprocess` library is used to get the terminal output of the script and store it in a local variable. This output is processed and ran through the `write_stream` component, which generates the response in a stream animation similar to ChatGPT. Once the output finishes visually generating, the content is placed back into the text area. The loading time is passed back to the application and is displayed right of the inference button.

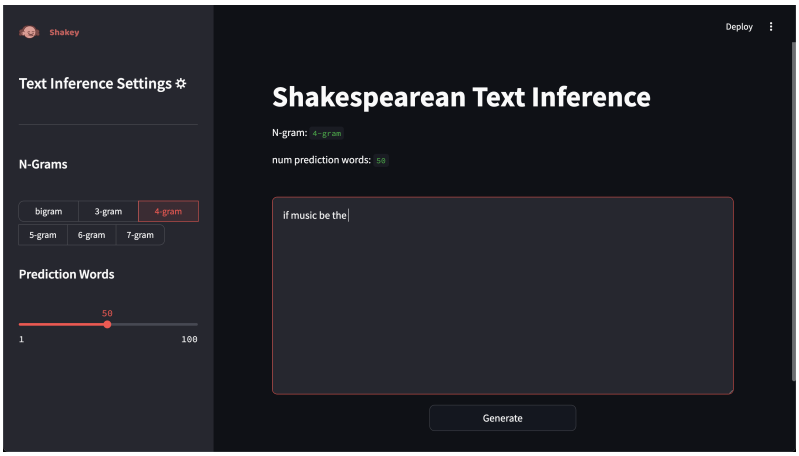

Figure 5: Streamlit UI.

8 Conclusions

While the bigram is the fastest model, it does not seem to generate proper English. Finding a middle between time efficiency and model variance, the best model for simply quote retrieval from the trained corpus is the 4-gram, with a loading time of 2.81 seconds. The best model for "speaking" in Shakespearean English is the 3-gram model. The generation time (1-100 words) is insignificant in inference calculations. Grams 5-7 quote the corpus directly most of the time and thus performed poorly on the evaluation from quotes outside the corpus. All models took a significant hit when going from training with ten to twenty plays. However, the larger models were the most affected. A two-second increase in inference time is not worth it, and I would not recommend any N-gram models above five. Because of this, it can be said that N-grams do not scale well. My corpora size of 20 plays is relatively small, and a much larger corpus size (such as those used for generative AI models such as ChatGPT) will leave the N-gram models thinking for hours.

9 Future Work

The next step involves researching NLP transformer models, seeing if there is an open-source model I can refine and make it work with my data set. I would like to add more training data to these models, enabling them to inference code (GitHub Copilot) or retrieve common data from Wikipedia. These models will be thoroughly tested before being integrated into my web application. I have deployed my N-gram models on a Streamlit application. Find it on https://shakey.streamlit.app/

10 Project Schedule

Table 5: The Project Schedule

Task	Status	Timeline
Literature Review, N-gram Experiments, Mockup UI	Done	Mar 08 - Apr 02
Building Streamlit App	Done	Apr 02 - Apr 14
Training with 20 Plays & Evaluation Metrics	Done	Apr 15 - Apr 18
Finalize Report, Final Presentation, Deployment	Done	Apr 19 - Apr 30

References

[1] Explosion AI. spaCy: Industrial-strength NLP, 2025.

[2] Figma. Figma: The collaborative interface design tool, 2025.

[3] MIT. The complete works of william shakespeare, 2025.

[4] Photopea. Photopea | online photo editor, 2025.

[5] Royal Shakespeare Company. Royal shakespeare company, 2025.

[6] Streamlit. Api reference - streamlit docs, 2025.

[7] Wikipedia contributors. Natural language processing — Wikipedia, the free encyclopedia, 2025.