Capstone Report

Kobe Taylor

Project Summary:

This project explores how predictive text systems work by building a hybrid next-word prediction tool using both traditional N-gram language modeling and a neural GRU-based model. My goal was to understand the complete underlying structure and fundamentals that are behind modern typing suggestions—how text becomes data, how probabilities are computed, how neural architectures learn patterns, and how these models are deployed in an interactive application. Throughout the semester, I designed a full workflow that included text preprocessing, database construction in SQLite, N-gram extraction, vocabulary building, neural network training in PyTorch, and a final Streamlit interface that allows users to compare prediction methods in real time.

The early history of this project traces back to my first submission around Halloween, which was essentially the bare bones of an AI system. At that point, I knew almost nothing about training loops, optimization, epochs, neural layers, or even the basic philosophy behind designing an AI. Instead, I started with something simple: predicting words based purely on frequency. From there, I learned how to tokenize text properly, how to decide which data actually mattered, how to store information efficiently with upsert operations, and how bigrams and trigrams form the backbone of statistical prediction. Each step made the next one possible. By progressing gradually—understanding one layer of the system before moving to the next—I was able to build confidence and eventually transition into neural modeling. This project became both a technical challenge and a personal journey of understanding how modern AI systems actually function.

Project Motivation:

When I first proposed this project, I didn't fully understand how predictive text systems worked—I only knew that they fascinated me. Every day, typing into my phone or writing assignments on my laptop, I would see suggested words appear before I finished typing. I wanted to understand the mechanism behind that experience, not from a high-level explanation but by building it myself from scratch. My motivation was curiosity mixed with a desire to finally work on something involving real AI concepts instead of just reading about them. To keep the project manageable, I separated my goals into four levels: basic, advanced, and smoothing of my results/data, and wildly optimistic. The basic features represented the bare minimum I needed to say I had built a functional predictive text system. This included constructing a database of typed sentences, tokenizing them, generating N-grams, calculating probabilities, and producing next-word suggestions based on statistical frequency alone. At the time, this already felt like a good milestone, because even simple N-gram models involve understanding text preprocessing, probability smoothing, and how to structure a database to support fast lookups. The advanced features reflected where I wanted the project to go once I gained enough comfort with the basics. This list included implementing a neural network—specifically a GRU—to compare against my N-gram model. These goals mattered because they pushed me out of purely theoretical learning and into practical engineering: I wanted to experience firsthand what it's like to train a model, deal with its mistakes, and refine it. Finally, the smoothing of my data was the fine-tuning of making sure my predictions were not just mambo-jumbo, spitting out very unrelated words based on whatever the user typed, and this was quite easily one of the most fixable in how to fine-tune these predictions but annoyingly tenuous in the practicality of having to insert thousands of data into my database to make the predciitons more accurate.

Lastly, the wildly optimistic goal wasn’t fleshed out, and it feels like something I will expand upon more in my conclusion, but I didn’t have much time to do so. Of course, in my mind, this whole project felt optimistic knowing that as someone who has never worked with coding anything AI related, let alone how they function at the most basic level, to making an AI that predicts the next word that a user may type was such a fascinating experience in of itself. So this goal of of wild optimism is something ongoing for the project.

Project Walkthrough and What I Learned During It

1. The Fundamentals and Building Blocks of Machine Learning

a. At the beginning of the project, my focus was on building the foundation the entire predictive text system would rely on: a structured database. I created a local SQLite database entirely through Python, which was an important milestone because it showed me how language-processing systems store and organize text data behind the scenes.

b. I implemented two primary tables:

   i. sentences – used to store complete user-entered text for training or testing

   ii. ngrams – used to store word-sequence patterns (“context → next word”), the core of next-word prediction

c. I also learned how SQLite seamlessly fits into a Python workflow — no special setup, no servers — making it perfect for rapid language model experimentation.

d. Most importantly, I began forming a deeper understanding of n-grams themselves:

i. Unigrams - single word, Bigrams - two-words, Trigrams - three words and etc

ii. It also deepened my understanding of how predictive AI's function, with two key values: context and next_word.

    1. Context being perhaps the most important aspect in telling our machine the most likely word to come next, and in early history of my project, it happened to be frequency-based.

2. Focus on Clarity, Cleanliness, and User-Experience

a. Tokenization and Regex

i. By this point in the project, I had moved beyond simply storing text and counting n-grams—I had to ensure that every sentence entering the system was cleaned and standardized. This led me into the world of tokenization, one of the most fundamental steps in natural language processing (NLP).

ii. I learned how raw text is full of inconsistencies: uppercase letters, punctuation, emojis, symbols, and spacing irregularities. These variations lead to fragmented or misleading n-gram statistics unless cleaned properly. My tokenization function addressed this by:

    1. Lowercasing text so "Pizza" and "pizza" are recognized as the same token
    2. Using a regular expression to remove any character that isn't a letter, number, space, or apostrophe
    3. Splitting the sentence into meaningful tokens
    4. Removing empty strings caused by multiple spaces

        iii. This experience taught me that clean data is the backbone of any predictive model. Even minor inconsistencies—such as treating “pizza!” as a different word than “pizza”—would distort frequency counts and weaken prediction accuracy.

   b. After weeks of designing backend logic, I finally reached a major turning point: building a user interface. I chose Streamlit, a Python-based web framework that allowed me to turn my scripts into an interactive app
   c. This UI allowed a user to:
      i. Type any sentence into an input field
      ii. Store the sentence directly into the SQLite database
      iii. Instantly view predicted next words based on n-gram statistics
      iv. Real-Time Prediction Testing
   d. It transformed all of my backend logic into something visible and interactive.
   e. Streamlit essentially connected my entire system:
      i. user → UI → database → model → prediction → UI output.
   f. Made it easy to test in real-time and for the first time in my project’s history gave me something tangible.

3. The Mathematical Structure

   a. Exploring Smoothing
      i. As my predictive system grew more complex, I reached a new stage of understanding: probabilities in language models are fragile. Without careful handling, a single unseen phrase can collapse the entire prediction pipeline. This led me into the world of smoothing techniques and backoff

strategies—two pillars of classical NLP that ensure reliability even when data is sparse. (This did not work in the basic model, but did later in my neural model)

ii. I discovered that without smoothing, any unseen word or phrase receives a probability of zero. This is disastrous for predictive text, because language is creative—people constantly write phrases that were never observed during training. Smoothing prevents the model from concluding that something is "impossible."

iii. Laplace Smoothing

    1. My simplest approach was Laplace smoothing, which adds +1 to every count before computing probabilities. This artificially boosts the frequency of rare or unseen words just enough to keep the system functional.
    2. It's mathematically simple and especially useful for small datasets like mine, where many word combinations appear only once or not at all. Implementing it gave me insight into how early language models survived long before modern neural networks.

iv. Kneser-Ney Smoothing

    1. Though I didn't implement Kneser-Ney (it's more complex and used in production systems), researching it helped expand my conceptual understanding. Its big idea is subtle but important:
        a. Instead of just counting how often a word appears,

i. It considers how many different contexts the word appears in.

b. This prevents misleading situations where frequent but context-specific words (like “San” from “San Francisco”) dominate predictions when they shouldn’t. Learning about this technique helped me see the limitations of my own simpler model and set the stage for my shift toward neural networks later on.

b. Embeddings and Vectors

   i. I learned that embeddings transform words from text into vectors—lists of floating-point numbers representing meaning. Words that appear in similar contexts end up near each other in vector space:

      1. cat and dog → close
      2. cat and economy → far apart

   ii. For example, with embeddings:

      1. Even if “I enjoy pizza” never appears in training,

         a. The model may still predict “pizza” after “I enjoy” because “enjoy” is vector-close to “like,” which was in the training data.

      2. This concept planted the seeds for the later GRU model—it helped me understand why neural networks outperform n-grams and why modern predictive text feels smarter and more flexible

4. Transition into Neural Networks

   a. By mid-October, my project had reached a natural turning point. I had spent weeks working with N-grams—explicitly counting word sequences, applying smoothing, tokenizing text, storing frequencies in SQLite. N-grams gave me a concrete foundation, but they also revealed their limitations: they struggle with long-range dependencies, unseen phrases, or anything more complex than rigid patterns.
   b. This realization pushed me into the world of neural networks—specifically GRUs (Gated Recurrent Units).
   c. Learning What a GRU Really Is
      i. A GRU is a type of recurrent neural network designed for handling sequences like sentences. Unlike N-grams, which only store frequencies, a GRU learns patterns through repetition and memory.
   d. As I studied GRU design, I also encountered the idea of layering in deep learning.
   e. Each layer adds a different level of abstraction:
      i. Embedding Layer: converts words into numeric vectors
      ii. GRU Layer(s): interpret sequences, learning grammar, flow, and contextual meaning across time.
      iii. Output Layer: converts the GRU's final hidden state into a probability distribution over possible next words.
   f. PyTorch

i. With the decision to implement a GRU, I had to learn PyTorch, the deep learning framework that powers neural networks in real-world applications.

ii. Understanding Tensors and Computation Graphs

1. At its core, PyTorch revolves around tensors—multi-dimensional arrays similar to matrices.
2. But what makes PyTorch powerful is that every tensor operation contributes to a dynamic computation graph:
    a. When you add, multiply, or pass data through a layer, PyTorch remembers the operation.
    b. When training time comes, PyTorch uses this graph to compute gradients automatically.
3. This is the engine behind neural learning: gradients indicate how much a weight should change to reduce prediction error.

iii. Learning What Weights Really Are

1. Weights had been abstract to me before this point. But through PyTorch, I learned:
    a. Every neuron connection has a numeric weight.
    b. These weights determine how influential each input is.
    c. Training is the process of slowly adjusting these weights until predictions improve.

g. Optimization Loops, Backpropagation, and Autograd

i. I learned that training a neural network is a repeated cycle called an optimization loop or a training loop
    1. Forward Pass – feed input through the GRU to get predictions.
    2. Loss Calculation – measure how wrong the prediction was.
    3. Backward Pass (Backpropagation) – compute gradients for every weight.
    4. Optimizer Step – adjust weights based on gradients.
    5. Repeat – thousands of times.

ii. This iterative refinement is what allows the network to learn language patterns.

iii. My n-gram model could only count; the GRU could now improve.

iv. Backpropagation
    1. Tracing how errors move backward through the network was eye-opening.
    2. Backpropagation answers two questions:
        a. Which weights caused the mistake?
        b. How should each weight change to reduce that mistake next time?

v. Autograd
    1. Thankfully, I didn't have to calculate the derivatives manually for the backpropagation
    2. PyTorch's autograd engine handles:
        a. Building the computation graph

        b. Tracking operations

        c. Calculating gradients automatically

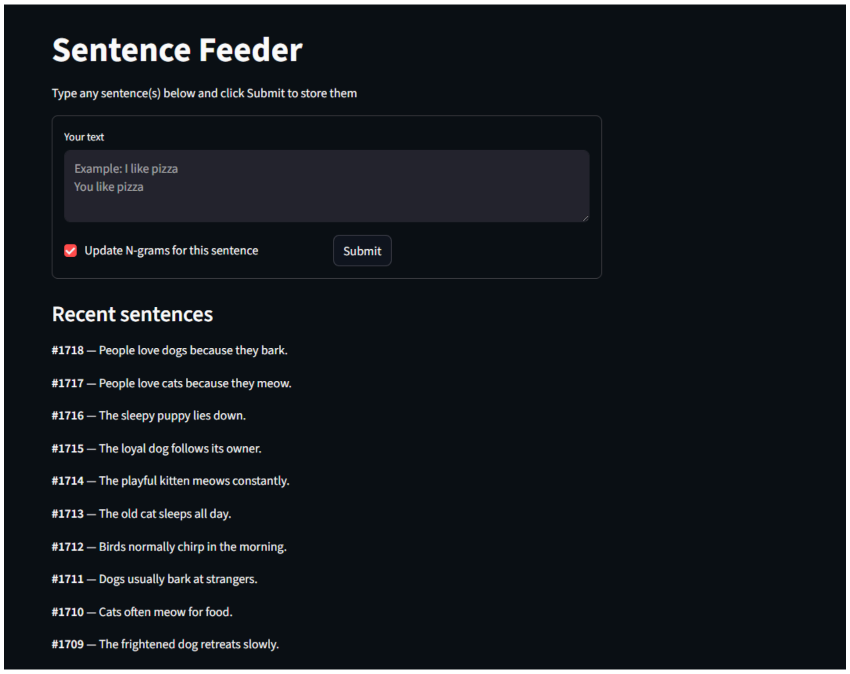
        d. Resetting gradients each step

    vi. This simplified the learning process massively. Where manually computing derivatives would have been so hard, PyTorch reduced it to just a line or two of code.

5. The Finish Line: Training/ Special Tokens/ More UI

a. In this phase, I fine-tuned my training, dealt with thousands of datasets, and spent more time ensuring my predictions were more accurate. I also added another app to make things cleaner and more streamlined. At this phase, one of the biggest hurdles was not hard conceptually to fix, but in practicality was a total slog to feed more data into my neural network to make my predictions better and to make the vectors of my words more similar. The problem to this involves several things; one, the English vocabulary is so vast, diverse, and potentially unlimited in the combinations it can be used, where even a thousand data sets are not comparable and are in fact minuscule in the scheme of making a really good predictive AI. Early on, I saw success, cat and dog were grouped together and words such as "is", "the", and "that" seemed to proceed a noun. It was also interesting to see how the evolution of my predictions matured during epochs of training after inserting more data over time. I am not at the point where I want to be in having an auto-complete feature such as the one on your phone, but it is incrementally on its way there with every new data set.

Tutorial - How to Run it and Go Through It

1. Get these packages installed - via terminal
    a. Have Python installed
        i. python --version or python3 --version
    b. Install Streamlit for our UI
        i. pip install streamlit
    c. Install PyTorch
        i. pip install torch
2. Initialize the Database
    a. Run the init_schema.py file
3. Run our first App - Our "Sentence Feeder."
    a. streamlit run app_textbox.py
4. Walk through the Sentence Feeder
    a. 

Sentence Feeder

Type any sentence(s) below and click Submit to store them

Your text

Example: I like pizza
You like pizza

☑ Update N-grams for this sentence

Submit

Recent sentences

#1718 — People love dogs because they bark.

#1717 — People love cats because they meow.

#1716 — The sleepy puppy lies down.

#1715 — The loyal dog follows its owner.

#1714 — The playful kitten meows constantly.

#1713 — The old cat sleeps all day.

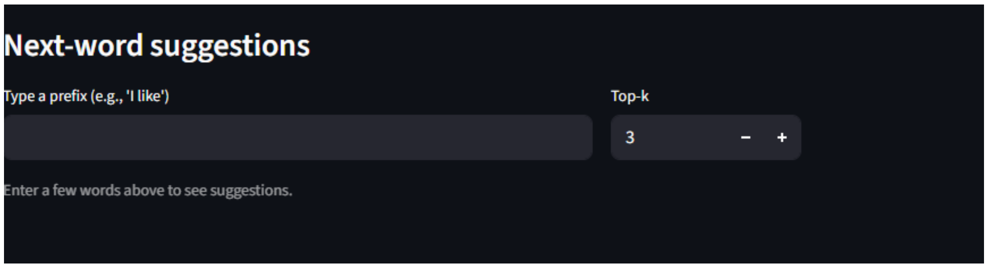
#1712 — Birds normally chirp in the morning.

#1711 — Dogs usually bark at strangers.

#1710 — Cats often meow for food.

#1709 — The frightened dog retreats slowly.

b. Type some sentences into the textbox above, anything separated by a line will be read as its own sentence, and watch recent Sentences become Updated

c. 

Next-word suggestions

Type a prefix (e.g., 'I like')

Top-k

3

Enter a few words above to see suggestions.

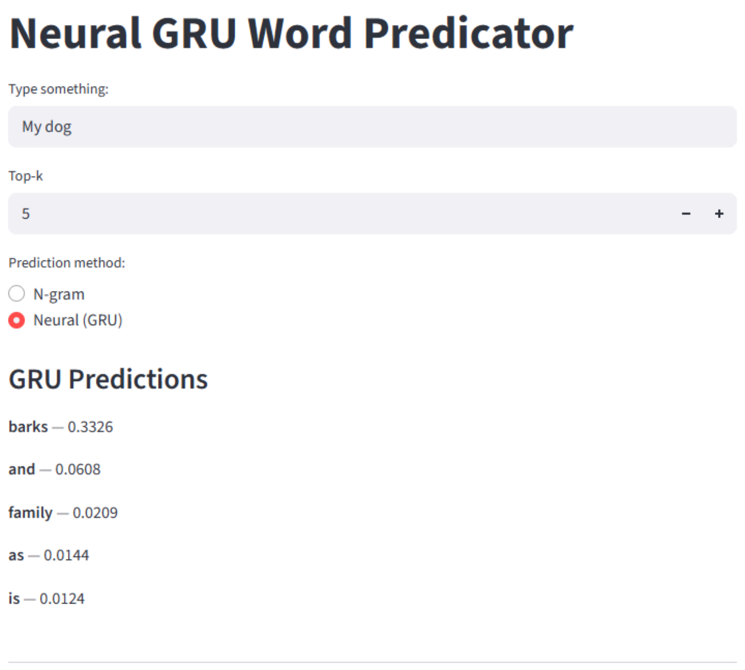
d. Try the bottom text-box, which primarily uses N-Grams to see if the new sentence you typed allows certain predictions
    i. Example:
        1. If you typed; “A Cat Meows”
            a. Simply typing “A Cat” ashould result in the predictions of “Meows”

e. After this, we now run our build_ngrams file to store those n-grams

f. We now run our train_gru.py to load our vocabulary, create our GRU model, and perform our training on our data

g. We now in our terminal, open our second app via
    i. streamlit run neural_gru.appy.py

h. We should now see this screen



Neural GRU Word Predicator

Type something: My dog

Top-k: 5

Prediction method: N-gram [✓ Neural (GRU)]

GRU Predictions

barks — 0.3326

and — 0.0608

family — 0.0209

as — 0.0144

is — 0.0124

i.

ii. Here you can now see the predictions of words given whatever you type, and you can seek and compare the two different predictive AI models, n-gram and neural network-based.