N-Gram Word Prediction Model

Braden Becker and Bryan Bailey
Williams College
CSCI 375 Natural Language Processing, Prof. Johan Boye

Introduction:

In the field of natural language processing (NLP), word prediction is a data-intensive problem that has a variety of effective implementations depending on the availability of data and the particular goal of each model. N-grams in probabilistic predictive word models are a common approach in both research[1] and consumer products, and these models target data from a context that is consistent with its target platform. For example, the iOS prediction models on the Apple iMessage app have a dataset based upon vast amounts of user input, which generally represents a more colloquial, abbreviated, and grammatically nonadherent set of n-grams due to the medium of text messages.

By contrast, some word predictive educational tools in the 'Assistive Technology' domain are aimed at helping dyslexic, visually impaired, and physically impaired people to communicate more clearly and efficiently through digital mediums[2], and so these data sets are based upon grammatically conformant and regularly occurring n-grams. Implementation method aside, empirical observation from user input, whether it be from literature, web-crawlers[3], or conversational communication, is regarded as the most effective way to produce reliable word-prediction results. In this paper, we will present our approach to creating an n-gram word prediction model using an amalgamation of NLP tools, including n-gram backoff, viterbi decoding as a spell checker, and letter-level backoff prediction.

[1] Examples include Gerald R. Gendron on behalf of Confido Consulting: "Natural Language Processing: A Model to Predict a Series of Words" (2015), Tinniam V Ganesh at R-Bloggers (2015), and Bickel et al. "Predicting Sentences using N-Gram Language Models"
[2] See Penfriend Ltd as well as Don Johnston word prediction software
[3] Buck et al. "N-gram Counts and Language Models from the Common Crawl" (2014)

Method and Data Selection:

While n-gram models are a common way to address the word prediction problem, more complex machine learning and neural network approaches are also conventional[4,5]. As with many NLP problems, n-gram modeling requires a large amount of data in order to be effective. Our corpus of choice is the Corpus of Contemporary American English, providing over 1 million of each 2, 3, and 4-gram probabilities. Being a corpus of American English, our target platform is most likely a word processor or e-mail, given the grammatical and lexical 'correctness' of the grams. N-gram models capture innate grammatical structures that naturally emerge from the empirical observations of language as it's used in practice. A prominent drawback of n-gram models are that they lack grammatical structure, meaning sentences from concatenations of predicted words might be grammatically incorrect. The model also does not capture distant lexical relations, such as relating a plural noun to its possessive later on in the text, e.g. an n-gram model could predict the phrase "the doors of the house is painted red", with these agreement disturbances being mitigated by the n-level in the model. As our implementation filtered probability based upon n-gram count, the equation to predict the next word in a phrase is given by:

$$\underset{w_n}{\operatorname{argmax}}(w_n \mid w_{n-1} \dots w_{n-l})$$

Where w_n is the predicted word and $w_{n-1} \dots w_{n-l}$ are the preceding l words, with l being the gram-level of the model. The idea that a future event can be predicted using a short history of observed word sequences is called a Markov assumption[6]. We collected data on the effectiveness of different Markov assumptions, or l, in order to minimize data initialization time and maximize the number of letters saved by each word predicted (See **Data** below).

[4] Yair Even-Zohar & Dan Roth "Classification Approach to Word Prediction" (2000)
[5] Sachin Agarwal & Shilpa Arora "Context Based Word Prediction for Texting Language" (2007)
[6] Stanford Professor emeritus Eric Roberts' page on word prediction in NLP

Implementation:

Word- and Letter-Level Backoff Prediction

Our program uses HashMaps and PriorityQueues as the primary data structures for their constant-time access and manipulable sorting capabilities, respectively. In order to easily facilitate the use of corpus data, we used the Java Serializeable interface[7] to store nested HashMaps prefilled with 2, 3, and 4-gram data that are each loaded into the model upon program startup. Our GUI uses a KeyListener in order to receive income letters. Depending on the running word count of the sentence and the gram-level given at startup, a gram-level predictor class will input the array of words and, using the 'n'-nested HashMaps to parse out possible continuations, output a PriorityQueue, with the top (string, integer) tuple being the most likely continuation of the phrase given by the corpus data. We implemented a recursive backoff so that if either the word count of the input was not long enough or the highest order gram level had no predictions, the model used the $n-1$ gram level until a word was predicted or unigram models could be implemented (see **Possible Extensions** below). Letter level backoff prediction is used to dramatically increase the number of letters saved. As a user types in a key, the list of predictions is updated to only include words that begin with the given stem, so that the predictions become more refined as more letters of a word are typed by the user.

Viterbi Decoding

If a user inputs a word that is not recognized in the program's serialized dictionary of over 100,000 words, a Viterbi Algorithm iterates over the word and decodes it until it either exhausts the program (preset limit) or the decoded word is found in the dictionary. The Viterbi Algorithm uses a Hidden Markov Model (HMM) in a trellis data structure to calculate the most probable path of a series of states given a set of observation and transition probabilities. Our transition probabilities are

[7] Java Official Documentation

2- and 3-gram letter probabilities taken from a training set of data based on Guardian News textfiles, while the observation probabilities are the likelihoods that the user accidentally hits a neighboring key instead of the intended key (0.1 probability for each neighboring key, remaining probability given to observed key)[8]. As an example, our implementation decodes “indubitqble” to “indubitable”, “cimpyrter” to “computer” and “heklo wodld” to “hello world” using 2- and 3-gram viterbi decoding. While Viterbi decoding was not the specific focus of this word prediction model, it certainly aids in helping the predictions operate at optimum level if there is no interruption in gram-level due to a mistyped word.

Procedure:

In order to determine the effectiveness of the model, an objective evaluatory metric was needed. As the model was being implemented into a real world application it was important to find a quantification that would directly measure the usefulness to our future users. The purpose of text prediction software is to speed up the pace at which one can type; therefore we evaluated each model by the number of keystrokes it saved the user compared to no predictive software. Pressing the “enter” key allows for a user to select the highest ranked word our model suggests. For example if the user had typed the sentence “at the very” and the highest ranked item was the word “beginning”, the selection of “beginning” by the user would result in 9 saved keystrokes as “beginning” is 9 characters long. In contrast if the user had typed “at the very bo” and the top recommendation for the partially completed word “bo” was “bottom”, the user could press enter and a keystroke saving of 4 would be recorded.

A series of sample texts were randomly selected from Wikipedia articles[9]. The excerpts varied in subject, from an entry concerning Donald Trump to a page detailing the field of Natural Language Processing, and in size, the smallest being 81 characters in length and the biggest almost

[8] Thank you to Prof. Boye for both probability sets
[9] See Appendix

9 times as large at 719 characters long. By selecting a cross-section of sample texts we hoped to collect more reasonable data on our model's performance in the variety of situations it would be expected to perform in.

In order to make our data comparable across different sized text samples we calculated the percentage of saved keystrokes by dividing the saved characters by the total characters. A model found to have a 50% predicted character rating would have, by way of the user pressing "enter" to select the top suggestion, typed half of the total text for the user.

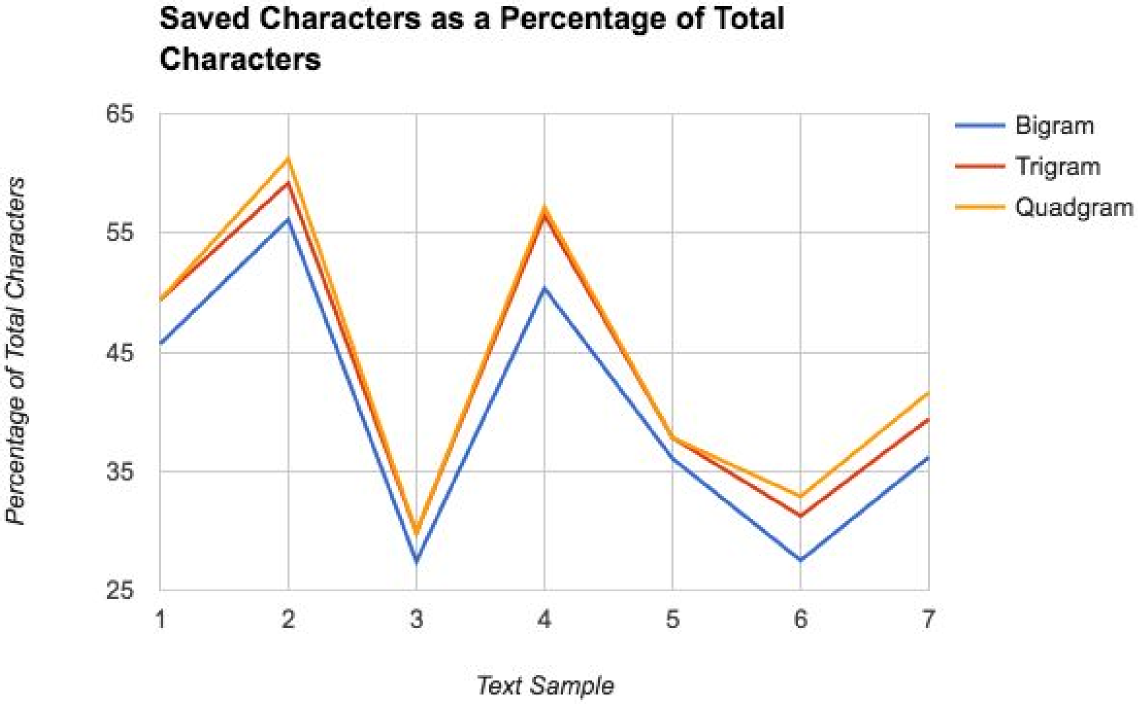
For each of the 7 sample texts 3 trials were conducted. The first predicted words only utilizing 2-gram probabilities, the second using up to 3-gram probabilities, but with the ability to backoff to 2-gram probabilities if no 3-gram existed, and third using up to 4-gram probabilities with the ability to backoff first to 3-gram, and then to 2-gram. Because of the backoff it was unlikely that a higher level n-gram model would ever score worse than a lower level n-gram. This was acceptable as we were interested in the relative increase in saved keystrokes as n-gram layers were added, not in the capability of a single n-gram level.

Hypothesis:

The most accurate prediction model will be the 4-gram level followed by the 3-gram level and finally the 2-gram level. Additionally, the longer the sample text the greater discrepancy will be between the smaller and larger n-grams. We believe this as due to backoff the higher level n-grams include all lower level grams and the larger n-grams will gain more and more context as the size of a text increases.

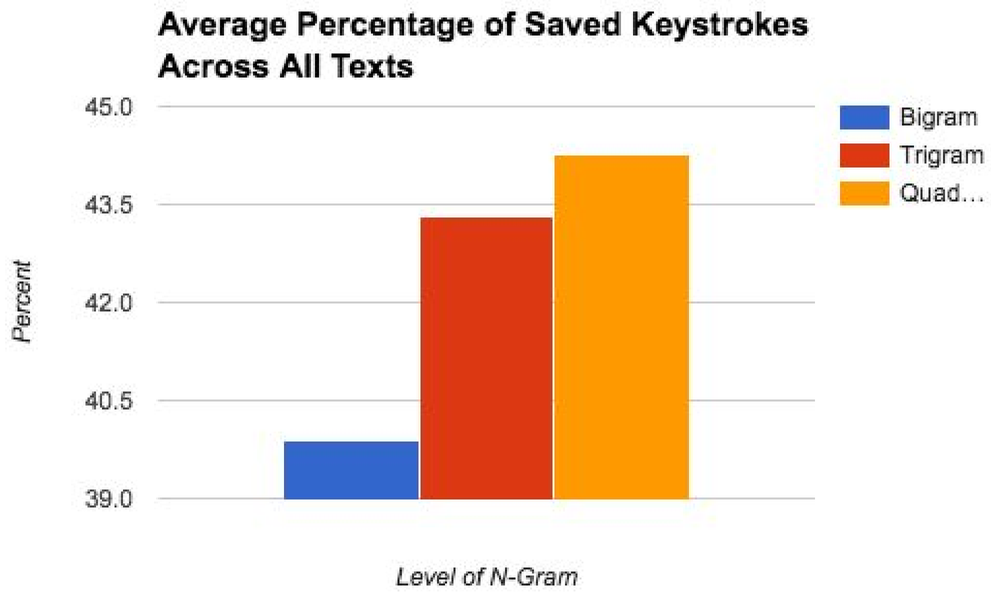
Data:

Across 7 different text samples each additional level of n-gram prediction was tested[10]. Calculating the percentage of characters predicted yielded a more useful metric for comparing across models.

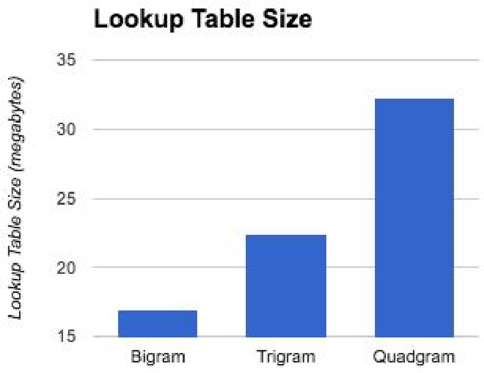


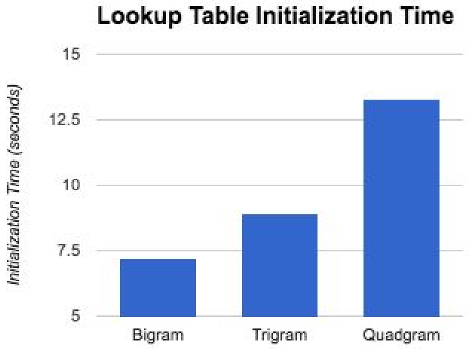
The text with the highest spread between n-gram levels was sample 4, with the 2-gram model saving 50.3% of characters and the 4-gram saving 57.1%, a 6.8% difference. The smallest spread between models was sample 5, with the 2-gram model saving 36% of characters and the 4-gram model saving only 37.8%, a mere 1.8% improvement. Across all models the lowest and highest scores were on text 3 and 2. The 2-gram model for test sample number 2 only saved 27.4% of characters, compared to the all time high score on sample number 2 with the 4-gram model saving 61.2% of characters, in effect typing over half of the text for the user. Averaging the percentages across all texts makes the differences between the models clear.

[10] See Appendix, Table 1



On average the 2-gram model saved the user 39.9% of the total keystrokes required to write a given text. The 3-gram model represented a significant increase in accuracy with an average prediction of 43.3%, a 3.4% increase over the 2-gram model. Finally the 4-gram model further improved on 3-gram with an average keystroke saving of 44.3%. However the improvement a 4-gram provided over a 3-gram was much smaller than 3-gram compared to a 2-gram with an increase in savings of only 1%. Framing this diminishing increase within the scope of the resources used at higher level n-gram models reveals an unsustainable path towards better accuracy.





The serialized HashMap lookup table object for the 2-gram probabilities is 16.9 megabytes in size. This increases to 22.4 megabytes for the 3-gram probabilities, and finally to 32.2 megabytes for the full 4-gram HashMap object. Additionally the time it takes for the Java ObjectInputStream class to initialize a given binary file at runtime is not trivial. It takes nearly double the time to initialize the 4-gram HashMap than it does to initialize the 2-gram HashMap, at 13.3 seconds versus 7.2 seconds. At runtime, with a 4-gram level selected, the program takes 29.42 seconds to initialize all of the serialized n-gram data structures. Seeing as the addition of a 4-gram level only enhances the performance of the program by 1% while increasing the required memory by 83% and startup time by 81%, it is difficult to justify the necessity of 4-gram level prediction, let alone adding 5-gram probabilities. For a formal word processor running in a desktop setting, the slightly increased accuracy may prove its worth. More complex documents may increase the likelihood of fully utilizing higher level n-grams while startup time and size are of little concern. However in a mobile application, where size and speed are crucial, the addition of n-gram levels above a 3-gram to our model is not practical.

Conclusions:

A correlation was expected between the effectiveness of higher level n-gram compared to lower level n-grams as the text sample size grew larger. As a sentence becomes longer there is more data for a higher level n-gram to use in order to make a more accurate prediction. However no such correlation appears in the data. The longest samples did not show a larger spread between models as the text size increased. The 4-gram predicted 5.1% more than the 2-gram when the text only contained 98 characters. When the text size was increased to 719 characters the 4-gram was only predicted 5.4% more than the 2-gram. The size of the larger file is nearly 9 times that of the smaller, yet the spread between a 2-gram model and a 4-gram model has not increased anywhere near the same rate. From this data it appears that the file size does not have an significant impact on the efficiency of each model. In order to prove this more trials would need to be conducted.

There was a large variation in the accuracy of our model depending on the text sample. Take for example the highest and lowest scoring texts, 2 and 3. Text 2 concerns Hillary Clinton and contains many phrases easily ranked by n-grams such as “the first lady”. Text 3 does not contain as many common phrases and additionally has obscure 2-grams such as “Ohio farm” and “canal boat” neither of which are in our corpus. In order to smooth the data a larger sample size would need to be picked to normalize the impact of strange wording and uncommon phrases.

Comparing our results to other text prediction platforms is difficult. Most software is proprietary and unavailable for data collection. However as a rough experiment to determine the effectiveness of our software, we compared some of our trials to an Apple iPhone 6 running iOS 10. We inputted the texts with the highest and lowest scores when run on our program into the messaging app and evaluated the letters saved using the procedure outlined above. Text sample 3, concerning James A. Garfield, had the worst percent accuracy of the samples we tested, saving only 37 characters out of the total 124 for an accuracy of 29.8% at the 4-gram level[11]. In comparison when run on iOS messenger, there were savings of 51 characters for an accuracy of 41.1%. Text sample 2, a snippet from Hillary Clinton’s wikipedia page, generated the highest score for our program with the 4-gram level saving 60 of the 98 characters for 61.2% accuracy score. Apple’s iOS messenger saved 57 characters for an accuracy of 58.1%, slightly below our programs’ achievement.

While impressive performance considering our program’s lack of unigrams, it is likely the corpus selection has drastically impacted the results. The underlying probabilities used in the iOS text prediction software are no doubt generated from a corpus of short text messages and therefore innately optimized for dealing with more informal lexical constructions. Our corpus, on the other

[11] See Appendix, Text Samples

hand, represents a much broader cross-section of English language. With the caveat of corpus optimization in mind, it remains impressive that using publicly available data we are able to build software with accuracy comparable to that of a proprietary commoditized version.

Possible Extensions:

Unigram Dataset

A unigram dataset could be added to improve the efficiency of letters saved upon user input. Our model did not include a unigram dataset dually because we were unable to find a dataset of proper size and format, and because we estimated that because of the nature of our backoff prediction model, all prediction levels would be able to utilize the unigram structure, thus uniformly smoothing the data across prediction levels.

Dealing with Novel Words

Currently, if an input word is not found in the program's dictionary, our program only attempts to decode it to a dictionary word using the Viterbi decoding algorithm. However, many prominent implementations of word prediction models store custom user data, as it is highly likely that a user will want to use their specific lexicon and vernacular again for convenience sake. To avoid having to store stock messaging phrases, esoteric proper nouns, and location-based words and phrases, our model could be extended to add previously unobserved n-grams to the data set and assign a probability according to a relative factor that it will be used again, implementation dependant.

Multiple Selection Option

Currently our program allows the user to only select the top item from the PriorityQueue. This was selected as speed was the primary concern, and the top item can always be selected by pressing a single key. Other options would require using either the mouse, arrows keys, or a multi key combination to select which word in the Queue should be added before actually adding it. Implementing the option to select which item in the Queue is straightforward and while it would certainly increase the letters saved, it would complicate the user experience.

Kneser-Ney Smoothing[12]

Evolving from absolute-discount interpolation, a discount value and normalizing constant are applied to each n-gram model in order to smooth probabilities across improperly weighted gram pairings. When predicting the probability of a word given a set of preceding words, we want to take into account the number of sets that predicted word is a part of as well as its given probability in the data set. For example, ‘Francisco’ is hardly ever observed outside of the preceding context of ‘San Francisco’, and Kneser-Ney smoothing takes this into account by dividing the number of unique unigrams (novel words) that the predicted word follows by all n-grams. This type of smoothing allows a model to predict out how likely a novel word is to appear in an unfamiliar n-gram context.

$$P_{abs}(w_i \mid w_{i-1}) = \frac{\max(c(w_{i-1} w_i) - \delta, 0)}{\sum_{w'} c(w_{i-1} w')} + \alpha \, p_{abs}(w_i)$$

Kneser-Ney Smoothing applied to a bigram language model

[12] Thanks to Stanford researcher Jon Gauthier for the explanation

Appendix:

Text Samples:

Sample 1: The United States is a highly developed country, with the world's largest economy by nominal GDP.

Source: https://en.wikipedia.org/wiki/United_States

Sample 2: As First Lady of the United States, Clinton led the unsuccessful effort to enact the Clinton health care plan of 1993.

From: https://en.wikipedia.org/wiki/Hillary_Clinton

Sample 3: “Garfield was raised in humble circumstances on an Ohio farm by his widowed mother. He worked at various jobs, including on a canal boat, in his youth.”

From: https://en.wikipedia.org/wiki/James_A._Garfield

Sample 4: Trump was born and raised in the Queens borough of New York City and received a bachelor's degree in economics from the Wharton School of the University of Pennsylvania in 1968.

From: https://en.wikipedia.org/wiki/Donald_Trump

Sample 5: Natural language processing is a field of computer science, artificial intelligence, and computational linguistics concerned with the interactions between computers and human (natural) languages.

From: https://en.wikipedia.org/wiki/Natural_language_processing

Sample 6: As a result, the country underwent a period of rapid industrialization and collectivization which laid the foundation for its victory in World War II and post-war dominance of Eastern Europe. Stalin also fomented political paranoia, and conducted the Great Purgeto remove opponents of his from the Communist Party through the mass arbitrary arrest of many people (military leaders, Communist Party members, and ordinary citizens alike) who were then sent to correctional labor camps or sentenced to death.

From: https://en.wikipedia.org/wiki/Soviet_Union

Sample 7: During the 1940s, as new and more powerful computing machines were developed, the term *computer* came to refer to the machines rather than their human predecessors. As it became clear that computers could be used for more than just mathematical calculations, the field of computer science broadened to study computation in general. Computer science began to be established as a distinct academic discipline in the 1950s and early 1960s. The world's first computer science degree program, the Cambridge Diploma in Computer Science, began at the University of Cambridge Computer Laboratory in 1953. The first computer science degree program in the United States was formed at Purdue University in 1962. Since practical computers became available, many applications of computing have become distinct areas of study in their own rights.

From: https://en.wikipedia.org/wiki/Computer_science

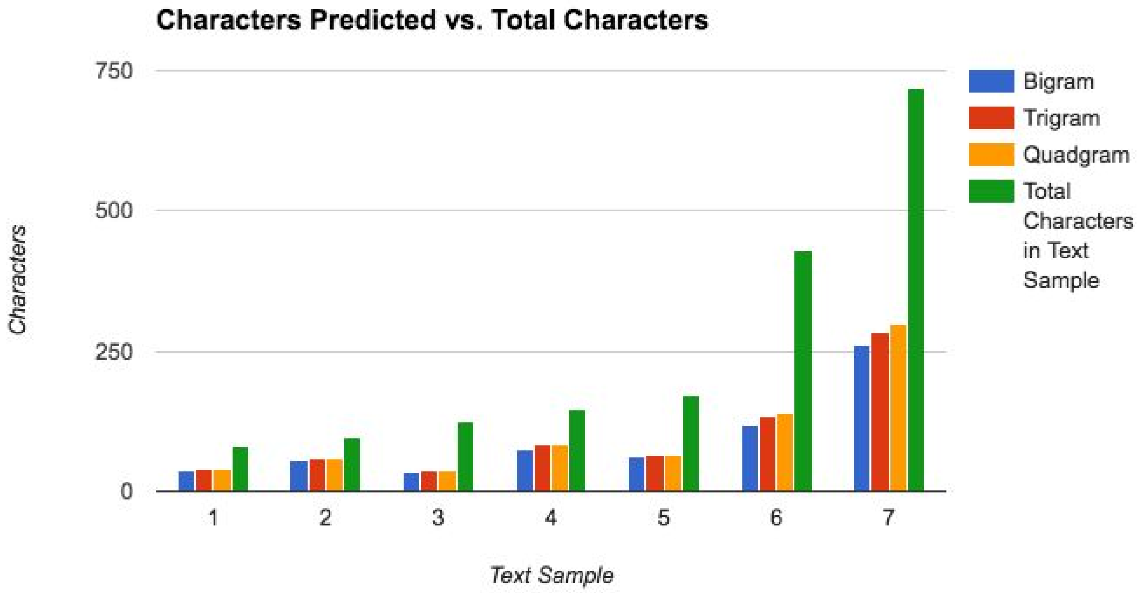
Table 1:

Text	Text Sample	2-gram	3-gram	4-gram	Total Characters in Text Sample	2-gram Percent	3-gram Percent	4-gram Percent
United States	1	37	40	40	81	45.7	49.4	49.4
Hillary Clinton	2	55	58	60	98	56.1	59.2	61.2
Garfield	3	34	37	37	124	27.4	29.8	29.8
Donald Trump	4	74	83	84	147	50.3	56.5	57.1
NLP	5	62	65	65	172	36.0	37.8	37.8
Soviet Union	6	118	134	141	429	27.5	31.2	32.9
Computer Science	7	260	283	299	719	36.2	39.4	41.6

Table 2:

	2-gram	3-gram	4-gram
Lookup table size(mb)	16.9	22.4	32.2
Average %	39.9	43.3	44.3
Initialization time (seconds)	7.23	8.92	13.27

Graph 1:


Characters Predicted vs. Total Characters
750
500
250
0
Characters
1
2
3
4
5
6
7
Text Sample
Bigram
Trigram
Quadgram
Total Characters in Text Sample