Bachelor's Thesis

Spelling Correction and Autocompletion for Mobile Devices

Ziang Lu

Examiner: Prof. Dr. Hannah Bast
Adviser: Matthias Hertel

University of Freiburg
Faculty of Engineering
Department of Computer Science
Chair of Algorithms and Data Structures

November 03rd, 2021

Writing Period

04. 08. 2021 – 03. 11. 2021

Examiner

Prof. Dr. Hannah Bast

Adviser

Matthias Hertel

Declaration

I hereby declare that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date	Signature

Abstract

With the ubiquity of the Internet, people are used to sending messages, writing e-mails, and querying search engines with smart mobile devices. To facilitate entering text with less keystrokes, and thereby save inputting time, ever smarter keyboards are developed. This paper aims to present how n-gram models and other relevant techniques could be used for the implementation of smart keyboards. In this paper, an English system keyboard for Android devices was developed to help in predicting, completing and correcting the next word. A detailed evaluation shows that n-gram models could contribute to the development of better keyboards and improve the user experience.

Zusammenfassung

Mit der Allgegenwart von Internet gewöhnen Leute sich schon daran, auf mobilen Geräten Nachrichten zu schicken, Email zu schreiben oder auf Suchmaschine ein Query zu machen. Um Tastendrücken zu reduzieren und Zeit zu ersparen, versucht man immer eine klugere Tastatur zu entwickeln. In diesem Paper werden wir n-gram Modell und andere relevante Theorien einführen und die entsprechende Implementierung zeigen. Und dann werden wir eine englische Systemtastatur auf Android Gerät entwickeln, die hilfreich bei Vervollständigung, Korrektur und Vorhersage sein kann. Am Ende beschreiben und analysieren wir die Evaluationsergebnisse und behaupten dass man mit der Implementierung von einem n-gram Modell eine kluge Tastatur bauen und Benutzung erlechtern kann.

Contents

1 Introduction

Virtual keyboards are becoming ubiquitous with the increasing use of mobile devices and touch screens. People interact with a keyboard by touching the screen, pressing keys, and choosing suggestions. A more effective way to input text on mobile devices could be achieved in many aspects, for example, people find the optimization of keyboard layout can improve user experiences and accelerate the speed of inputting [1]. In addition, a better inputting experience also depends on an effective "recommendation system" for the next word, namely, after individuals type a part of a word, they could choose candidates from a bar on the top of the keyboard. The earlier a user could find the appropriate candidate, the quicker he could finish his input. In this paper, we will only pay attention to the program solution: completion, correction, and prediction.

If you intend to input "something" and have inputted "som" at first, you find you can directly let "som" completed by clicking a candidate from the bar on the top of the keyboard, then you have saved at least 6 keystrokes; Imagine that after you finished typing "I will be there in five " (with a space), you find three candidates: "days", "minutes", "seconds". Here you have a chance to complete your next word with only one keystroke; Or while you are walking, you want to reply to your friend, but you can not hold your phone very well. You may type "do you need somthing" and the keyboard shows you the correct version of "something". Luckily, you do not have to come back to correct your spelling mistake. All of the work above needs an effective algorithm and precise language model.

Likelihood probabilistic models have been used for text processing for a long time [2]. Such a model is a context-based model which is built by counting the frequency of the same sequences or combinations. A typical and classical example is the application of n-gram which is widely used in many areas in Natural Language Processing, such as text categorization [3], machine translation [4], speech recognition [5] and so on. Application of the n-gram model in predicting or completing the next word has also been shown as an efficient and precise way to finish this work [6, 7].

Completing or correcting a word, we need to know which word could be selected to complete or correct our goal. Other than the use of the n-gram model, we still should quantify the similarity between two words. Church and Gale (1991) had developed a correct program that could rank suggestions according to four confusion matrices which represented probabilities of different ways of making spelling mistakes [8]. Kondrak (2005) attached importance to the role of edit distance in quantifying similarity between two strings and also referred to the combination of edit distance and n-gram model [9]. And we will implement this combination of the n-gram model and Prefix Edit Distance into an Android keyboard, which could complete, correct, and predict the next word. Since the papers before only refer to the level of theory or model's implementation about such combination, this paper succeeds in realizing algorithms into an available and function-completed app.

As a comparison, Google Keyboard (Gboard), a keyboard trained by a neural network language model, will be used to execute part of tasks for evaluation. The applied language model uses a distributed and on-device learning framework which is called Federated learning [10]. This framework focuses on the collection and analysis of data generated by users. One of the outstanding features of Federated learning is that it could train a language model using "real" data without the invasion of private and sensible information [10]. Instead of uploading users' data into the server, Federated learning uses client data to execute training tasks only in their own devices during idle time. After finishing pre-defined computation tasks, the updated model weights

will be sent in the form of vectors into the server and aggregated for the next step [10]. That guarantees the privacy of users and the quality of the language model. Through comparison with Gboard, we will prove that the traditional statistical model could still have satisfying performance in completing, correcting, and predicting next words.

2 Theoretical Background

We will firstly list the theories that we need and then talk about the application of theories in the next section.

2.1 Edit Distance

2.1.1 Levenshtein Distance

Edit distance is a way of quantifying how dissimilar two strings are to one another. It quantifies the difference between two strings by counting the number of operations needed to transfer one string to the other. Levenshtein Distance is a member of group edit distance. It is named after Soviet mathematician Vladimir Levenshtein, who first thought about this method in 1965. To calculate the Levenshtein Distance between two strings, one can insert, replace or delete letters to transfer one string to the other string [11]. The total number of such operations is the Levenshtein Distance between two strings:

insert(i, c): insert character c at position i

delete(i, c): delete character c at position i

replace(i, c): replace character c at position i

Let us take a look at an example:

A: *similar*

B: *familiar*

We intend to transfer A to B. Available operations: insert, replace and delete.

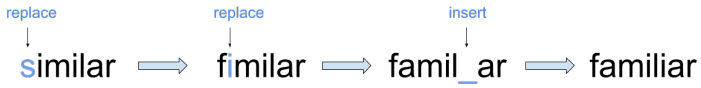

Figure 1: Edit Distance between *similar* and *familiar*

Therefore, we need two replacements and one insertion to finish the transference. And the Levenshtein Distance between "similar" and "familiar" is 3.

2.1.2 Prefix Edit Distance

Prefix Edit Distance is a variant of Levenshtein Distance.

Definition Given two strings A and B, we should count the minimal number of operations that we need to transfer A into a prefix of B. The operation could be insertion, replacement, or deletion:

$$PED(x, y) = min_{y'}(ED(x, y'))$$

where y' is a prefix of y

Then the Prefix Edit Distance between *same* and *something* should look like this:

Figure 2: Prefix Edit Distance between *same* and *something*

Hence, the Prefix Edit Distance (denoted as PED) between C and D is 1.

2.2 Q-gram Index

Q-grams of a string S are simply a set of all substrings of string S with a length q. If q = 3, then the 3-grams of "something" should be:

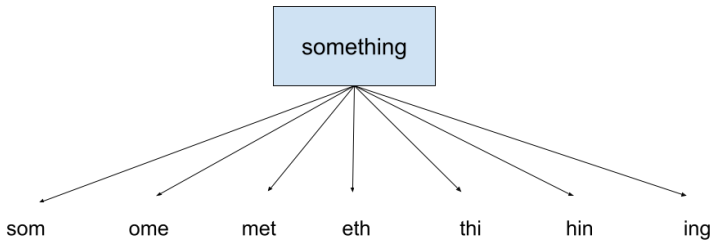

Figure 3: 3-grams of *something*

Given a string A and threshold δ, we want to find all candidates whose PED from A is smaller than δ. Because of the size of a vocabulary, calculating PED between string A and all other words from the vocabulary will take a lot of time. Generally, we could avoid some unnecessary calculations by comparing the number of the common Q-gram of two strings. Only if the number of common Q-grams between two strings

exceeds a given threshold, we will calculate the PED between them. This could save us much time.

A q-gram index maps a q-gram to all words which contain that q-gram. Given a vocabulary with a fixed size, the q-gram index could map a q-gram to tuples which consist of the index of a word that contains that q-gram and the frequency of q-gram in that word. Q-gram index could be applied to find words that have enough common q-grams with a string or the part of the inputted sentence if we have determined a threshold.

For example, the inputted string is *compu* and its q-grams are *com*, *omp* and *mpu*. Assume that the threshold is 3 and we only want to get words who have at least 3 common q-grams with *compu*. Given the q-grams indices, we retrieve the corresponding words (index, frequency) which contain the three q-grams above and get results as follows:

com	(1, 1), (2, 1), (3, 1)
omp	(1, 1), (3, 1)
mpu	(1, 1), (3, 1)

After merging the three lists, we have an obvious answer:

(1, 3), (2, 1), (3, 3)

Therefore, two words have fulfilled the requirement and they are the first word and the third word.

2.3 N-gram Model

A n-gram is a sequence of n words: a 2-gram is a two-word sequence of words like "have you", "seen my" or "my book" and a 3-gram is a three-word sequence of words like "have you seen", "you seen my" or "seen my book". A 1-gram (or unigram) is just a single word or one-word sequence. It is a type of probabilistic language model for predicting the next item based on the last n – 1 words.

Let us continue with the example before:

h: "have you seen my"
w: "book"
k is a corpus

we consider **h** as history and **w** as a possible candidate. We want to calculate the probability of **w** as the next word and denote it as:

$$P(\text{book}|\text{have you seen my})$$

Since the size of corpus **k** is limited and combinations like "have you seen my" may not appear, we will instead approximate the probability just through the last few words, for instance, "seen my" or "my". In other words, we could calculate the probability based on a 2-gram or 3-gram, namely, we look back into only one word or two words. In this case, we just need to count the frequency of sequence "my book" and frequency of "my" in the corpus **k** (here we denote that as **C**(sequence)). Then we have:

$$P(\text{book}|\text{my}) = \frac{C(\text{my book})}{C(\text{my})}$$

or

$$P(\text{book}|\text{seen my}) = \frac{C(\text{seen my book})}{C(\text{seen my})}$$

C(my book) is to count how many times the combination "my book" appears in the corpus. **C**(my) is to count how many times the word "my" in **k** appears. To extend our 2-gram model to general n-gram model, we have this formula as follow:

$$P(w_n|w_{n-N+1}^{n-1}) = \frac{C(w_{n-N+1}^{n-1}w_n)}{C(w_{n-N+1}^{n-1})}$$

(n represents the length of a whole sentence. w_J^I represents a sequence which consists of $w_I, w_{I+1}, w_{I+2}, ..., w_J$.)

2.4 Grammar Process

2.4.1 POS-Tagging

POS is short for *part of speech*. A part of speech is a category of words that have similar grammatical properties. Commonly listed English parts of speech are noun, verb, adjective, adverb, pronoun, preposition, conjunction, numeral, article, or determiner. POS-Tagging is literally to categorize every part of a sentence into a tag that denotes the grammatical property of that part.

For example:

Figure 4: POS-Tags

Annotation for a large corpus takes a lot of time if people manually do it. Nowadays,

we get the annotation or an annotated corpus by using the automatic POS tagger. In this paper, we will use the python package NLTK as the POS-tagger which supports annotation of 45 kinds of POS tags [12].

2.4.2 Hidden Markov States

Markov Chain

A Markov Chain is a mathematical system that experiences transitions from one state to another according to certain probabilistic rules. The probability of transitioning to any particular state is dependent only on the current state but not before that. We call that the Markov property.

Given a set of states of ground, e.g.

$$S = \{\mathbf{d}(\text{damp}),\ \mathbf{n}(\text{normal}),\ \mathbf{r}(\text{dry})\}$$

with initial probabilities.

$$Pr(d) = 0.1$$

$$Pr(n) = 0.7$$

$$Pr(r) = 0.2$$

and **transition probabilities** between states

$$Pr(d \to d) = 0.2, Pr(d \to n) = 0.6, Pr(d \to r) = 0.2$$

$$Pr(n \to n) = 0.6, Pr(n \to d) = 0.1, Pr(n \to r) = 0.3$$

$$Pr(r \to r) = 0.3, Pr(r \to n) = 0.6, Pr(r \to d) = 0.1$$

With this we could calculate the probability of a sequence of events, for example, we want to know the probability when the ground is firstly dumped, then becomes normal, then becomes dry and at last comes back to dump state, namely:

$$\begin{aligned} Pr(d \to n \to r \to d) &= Pr(d) \cdot Pr(d \to n) \cdot Pr(n \to r) \cdot Pr(r \to d) && (1) \\ &= 0.1 \cdot 0.6 \cdot 0.3 \cdot 0.1 && (2) \\ &= 0.0018 && (3) \end{aligned}$$

Hidden Markov States

Hidden Markov model is basically a Markov chain whose internal state cannot be observed directly but only through some probabilistic function [13]. For example, dry, damp, and normal are states which could be observed, and the weather: rain, sunny or cloudy we could call them hidden states as causes.

Sometimes we could only perceive observed states and want to know the hidden state or the cause behind the phenomenon. We could calculate the probability of a given hidden state through Hidden Markov model.

Given a set of hidden states

$$H = \{y^{-}(rainy), y^{=}(cloudy), y^{+}(sunny)\}$$

and a set of observable states

$$O = \{d(dump), n(normal), r(dry)\}$$

Like we define the probability of observed states before, we also need to determine initial probabilities and transition probabilities of those hidden states:

$$Pr(y^-) = 0.3, Pr(y^=) = 0.1, Pr(y^+) = 0.6$$

$$Pr(y^= \to y^-) = 0.1, Pr(y^= \to y^=) = 0.3, Pr(y^= \to y^-) = 0.6$$

$$Pr(y^- \to y^-) = 0.3, Pr(y^- \to y^=) = 0.1, Pr(y^- \to y^+) = 0.6$$

$$Pr(y^+ \to y^+) = 0.5, Pr(y^+ \to y^=) = 0.3, Pr(y^+ \to y^-) = 0.2$$

As explained above, observable states are usually dependent on hidden states. So we could define the conditional probability and we will call this **emission probability** here:

$$Pr(r|y^-) = 0.1 Pr(n|y^-) = 0.2 Pr(d|y^-) = 0.7$$

$$Pr(r|y^=) = 0.3 Pr(n|y^=) = 0.6 Pr(d|y^=) = 0.1$$

$$Pr(r|y^+) = 0.7 Pr(n|y^+) = 0.2 Pr(d|y^+) = 0.1$$

Now, we could think about how to get the probability of a sequence of hidden states given a sequence of observable states.

Formula:

$$\begin{aligned} Pr(o_1, ..., o_n, h_1, ..., h_n) &= Pr(o_1|h_1) \cdot Pr(o_2, h_2|o_1, h_1) \cdot Pr(o_3, h_3|o_2, h_2) \cdot ... && (4) \\ &= Pr(h_1) \cdot Pr(o_1|h_1) \cdot Pr(h_2|h_1) \cdot Pr(o_2|h2) \cdot ... && (5) \\ &= \prod_{n=1..n} Pr(h_i|h_{i-1}) \cdot Pr(o_i|h_i) && (6) \end{aligned}$$

Our goal is to maximize the formula:

$$\prod_{n=1..n} Pr(h_i|h_{i-1}) \cdot Pr(o_i|h_i)$$

so that we can find the most likely sequence of hidden states with the highest probability.

Instead of solving this optimization problem by brute force (trying all assignments to h_i), we will look at the Viterbi Algorithm.

2.4.3 Viterbi Algorithm

Viterbi Algorithm is especially for obtaining the maximum probability of the most likely sequence of hidden states. Let $Pr_n(h_n) := \prod_{n=1..n} Pr(h_i|h_{i-1}) \cdot Pr(o_i|h_i)$. And the principle of the algorithm is to express $Pr_n(h_n)$ recursively by $Pr_{n-1}(h_{n-1})$:

$$\begin{aligned} Pr_n(h_n) &= max_{h_{1,\ldots,h_{n-1}}} \prod_{i=1,..n} \{ Pr(h_i|h_{i-1}) \cdot Pr(o_i|h_i) \} \\ &= max_{h_{1,\ldots,h_{n-1}}} \left\{ Pr(h_n|h_{n-1}) \cdot Pr(o_n|h_n) \cdot \prod_{i=1,..n} Pr(h_i|h_{i-1}) \cdot Pr(o_i|h_i) \right\} \\ &= Pr(o_n|h_n) \cdot max_{h_{n-1}} \left\{ Pr(h_n|h_{n-1}) \cdot max_{h1,\ldots,h_{n-2}} \prod_{i=1,..n} Pr(h_i|h_{i-1}) \cdot Pr(o_i|h_i) \right\} \\ &= Pr(o_n|h_n) \cdot max_{h_{n-1}} \{ Pr(h_n|h_{n-1}) \cdot Pr_{n-1}(h_{n-1}) \} \end{aligned}$$

The detailed expression in the algorithm will be introduced in the next section.

2.5 SQLite

SQLite is a relational database management system. Compared with many other databases, the most outstanding feature of SQLite is that it is not a server-client database engine and is embedded into the program. For example, MySQL requires a server to run. MySQL will require a client and server architecture to interact over a network. In contrast, SQLite is self-contained [14]. And the DB engine runs as

a part of the app. A program could call functions of SQLite by its API defined in the programming language. This feature could help in reducing delays when an application or program visits data in the database.

3 Approach

In this section, we will talk about approaches to realizing the functionalities of the keyboard and the application of theories.

3.1 Find Candidates

3.1.1 Similarity calculate

We are going to build a keyboard which firstly could complete and correct our input. If the keyboard ought to complete our input into a correct word, it needs to know whether our input could be part of our intended word. In other words, the keyboard ought to judge if there is a potential candidate that is very similar to our inputted string.

The same principle could be applied to the correction of the keyboard. Spelling mistakes could be classified into two types [15]. One is context-sensitive spelling error which means our "wrong" input exits actually in the dictionary but it does not correspond to the syntax or grammar of context, such as "I having an idea". The other one is called a non-word error. In this paper, we only focus on the non-word error which can not be found in the dictionary such as "abouuut". In this case, our keyboard should know which correct word should be most possibly shown to the user. That also includes the consideration of similarity.

In the section of theoretical background, we have talked about the method to calculate the similarity between two strings: Levenshtein distance. When we compare two words with Levenshtein distance, we need to consider two complete words. However, we input a word by starting the first letter, and the second one until the last letter. Consequently, if the keyboard could show us *something* as a candidate, then we must have inputted *somethi* or maybe a little bit longer prefix. Because no matter with which prefix *some* or *somet*, the Levenshtein distance between the "something" and those prefixes would be very large.

That means the Levenshtein distance could not be used as a criterium for the keyboard to generate a candidate. Because the larger Levenshtein distance implies that the similarity between the input and an ideal candidate is very small. Hence, we have introduced a variant of Levenshtein Distance: Prefix Edit Distance. This time we still calculate the Levenshtein Distance but not between two complete words. In that way, the keyboard supplies us with candidates only considering the Levenshtein distance between the prefix of a word and what we have inputted. And if we input *som*, then *something*, *sometimes* or *someone* could also be acknowledged as possible candidates since the PED between them and *som* is only 1.

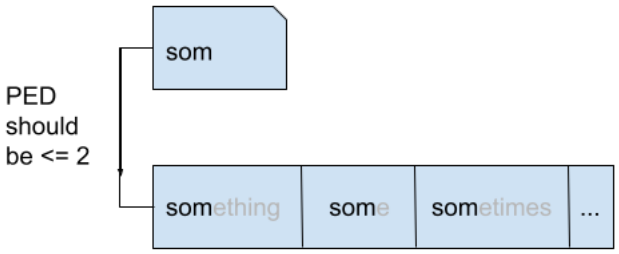

Figure 5: Judge similarity by PED

3.1.2 Accelerate calculation

A response time feels interactive until around 200ms. It takes us a lot of time if we compare a string with all other words one by one from the vocabulary. It makes calculation very inefficient and unnecessarily slow. For example, PED between "movi" and "cinema" is intuitively larger than 2 and it is unnecessary to make a calculation.

Before we talked about the role of the Q-gram in reducing the number of candidates which needed to be considered. Intuitively, if two strings are not too short and δ(threshold of PED) is not too large, they will have one or more Q-grams in common.

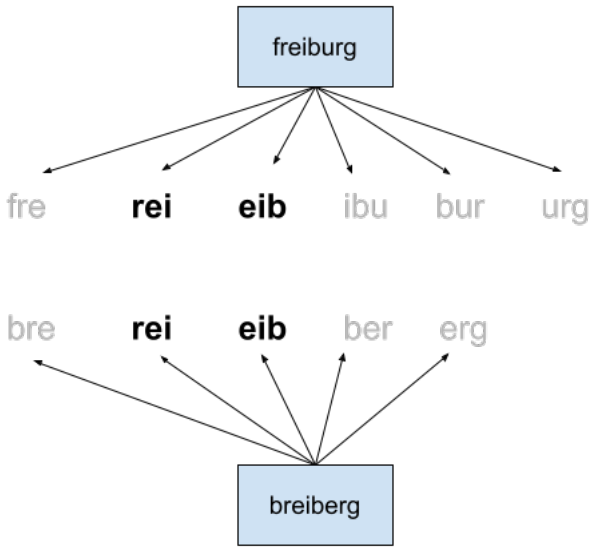

Figure 6: Q-gram works just like we quickly have a glance and judge if two strings may have small PED

That means, given a threshold, we could decide if a word deserves to be taken into consideration by comparing the number of the common Q-grams between two strings. If not, we can kick those words out and save the time of calculation.

However, if q = 3, it is impossible to divide a word whose length is less than 3 into any parts, such as "I", "if", "am" and so on. To avoid ignoring important candidates,

we will pad two special symbols "$$" into the start of every word, such as *$$I*, *$$am*, *$$sometimes*. In this way, we could get Q-gram indices of every padded word.

Given a string x and its padded version $x^{'}$, we want to find if PED between another string y (padded version $y^{'}$) and x is smaller than or equal to 2 ($\delta = 2$). For this, the number of common Q-grams must reach a value:

$$comm(x^{'}, y^{'}) \geq |x| - q \cdot \delta$$

We could apply the formula to check if a comparison should be executed:

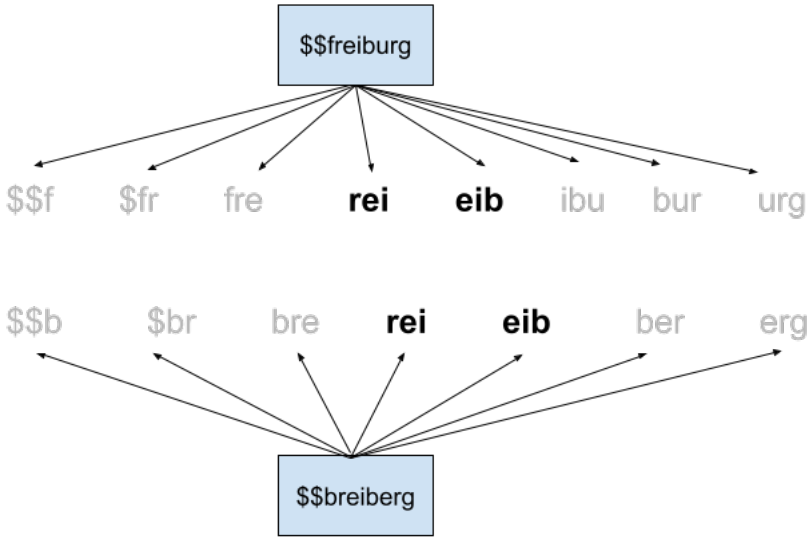

Figure 7: Pre-check

$comm(x^{'}, y^{'}) = 2 \geq |8| - 3 \cdot 2$. Hence, *breiberg* is a match.

3.2 N-gram probability

We have firstly solved the problem of similarity and its calculation speed. In the next step, we need to consider the ranking of candidates. We may see even thousands of candidates who fulfill the requirement of PED. But some candidates should be filtered according to our linguistic experience. For example, given a part of a sentence: "have you seen my b ". And you need to complete the word with the prefix "b". By the way, the PED between "b" and many words is only 0 such as "be", "being", "back", "bag", "book", "basketball". If you are a native English speaker, you will very quickly exclude words like "be", "being" and "back" without thinking of any grammatical rules. Because in your memory, you have never seen them before. It is a very simple statistical problem. You see many combinations like "my book", "my basketball" or "my baby" and you will think "book", "basketball" and "baby" will have high probabilities than others. What we think about is just how n-gram models work.

Before in section 2 we have extended our bigram model to a general N-gram model and we have this formula as follow:

$$P(w_n|w_{n-N+1}^{n-1}) = \frac{C(w_{n-N+1}^{n-1}w_n)}{C(w_{n-N+1}^{n-1})}$$

(n represents the length of a whole sentence. w_J^I represents a sequence which consists of $w_I, w_{I+1}, w_{I+2}, ..., w_J$.)

With this formula, we can usually find a corresponding bigram or trigram (if the corpus is big enough) for the candidates. If the corresponding bigram and trigram do not exist, we still could calculate the probability based on unigram. When we could find both of the corresponding bigram and trigram in models, we need to assign different weights to probability based on unigram, bigram, and trigram. The reason could be described as that the more parts of the sentence we consider, the more

precise probability we could get for the next word. Since people produce a sentence in any language is a sequential process.

For example, the inputted part is: “where are you g?”. If the whole part of “where are you” is masked, and you could only guess which words should come most likely with the prefix “g”, and it could be very hard, since no historical information is available. But as we uncover more and more parts until the last word “where”, it becomes always clearer which word you should choose. In this paper, the calculation of a n-gram probability will be so planned:

$$P(\text{N-gram}) = \sum_{i=1}^{N} P(w_n | w_{n-N+1}^{n-1}) \ \cdot \lambda_i$$

We assume that the attendance of longer sequence could improve the performance a lot since it considers more contexts. The specific values of λ_i will be determined in the evaluation.

3.3 POS-Tag probability

No matter how big a corpus is, there is still the combination of words that never appears. However, if we only see “I sleep” a lot of times in the corpus but never see the combination “she sleeps” or “he sleeps”, it does not mean that “she sleeps” or “he sleeps” is not grammatically true. The reason is that language or expression is very flexible and changeable, so a limited corpus is difficult to cover all possibilities. In this situation, you have known that after a third-person pronoun we need to change the form of a verb such as “sleep” into “sleeps”, “make” into “makes”, “take” into “takes”, and so on. The task is to find out if a candidate could be selected as the next word, which linguistic feature it should have. That means we need to know the maximum probabilities of POS-Tags for a sentence that ends with different candidates. Here we

could apply the Viterbi algorithm to calculate the most likely linguistic property of the next word.

Hidden states could be instantiated as the POS-Tags of English words and observable states could be regarded as English words.

In the second section, we have introduced a formula that maximizes our goal recursively. To calculate the grammatical property of the last word, we start always from the first left part of the sentence. Our input includes [16]:

- the observation space $\mathcal{O} = \{o_1, o_2, ..., o_N\}$ and in our case this space is composed of every word from vocabulary.

- the state space $\mathcal{S} = \{s_1, s_2, ..., S_K\}$. Here the state space is equal to the set of the POS-tags.

- an array of initial probabilities $\prod = (\pi_1, \pi_2, ..., \pi_K)$. This describes the probability of a word under every tag could be the start of a sentence.

- a sequence of observations $\mathcal{Y} = (y_1, y_2, ..., y_T)$. That is just which sentence we have observed and each y belonging to the $\mathcal{O}$.

- transition matrix A of size $K \times K$ such that A_{ij} stores the **transition probabilities** from tag to tag.

- **emission** matrix B of size $K \times N$ such that B_{ij} stores the probability of a word belongs to a specific tag.

Output:

- the most likely hidden state sequence, namely the most likely POS-tags sequence $\mathcal{X} = (x_1, x_2, ..., x_T)$

and the whole routine could be represented as the algorithm as follows [16]:

Algorithm 1 Viterbi Algorithm

function VITERBI($O, S, \prod, Y, A, B$): X
  for *each state* $i = 1, .., K$ **do**
    $T_1\,[i,\ 1] \leftarrow \pi_i \cdot B_{iy_1}$ ▷ mark current most likely tag
    $T_2\,[i,\ 1] \leftarrow 0$ ▷ mark last most likely tag
  end for
  for *each observation* $j = 2, 3, ..., T$ **do** ▷ iteratively calculate possibility
    for *each state* $i = 1, 2, ..., K$ **do**
      $T_1[i,\ j] \leftarrow \max\limits_k (T_1[k,\ j-1] \cdot A_{ki} \cdot B_{iy_j})$
      $T_2[i,\ j] \leftarrow arg\ \max\limits_k\ (T_1[k,\ j-1] \cdot A_{ki} \cdot B_{iy_j})$
    end for
  end for
  $z_T \leftarrow arg\ \max\limits_k\ (T_1[k,\ T])$
  $x_T \leftarrow s_{z_T}$
  for $j = T, T-1, ..., 2$ **do**
    $z_{j-1} \leftarrow T_2[z_j,\ j]$ ▷ find most likely tag backwards until the start
    $x_{j-1} \leftarrow S_{z_{j-1}}$
  end for
  return X
end function

Since we only want to know the most likely underlying POS-tag of the last word namely the next word, we do not have to get the whole sequence of most likely tags.

3.4 Integration

Before we assigned different weights to probabilities based on different n-grams. In this way, the more important factor plays a greater role. Hence, we also need to find a balance among the effects of PED, n-gram, and POS-Tags. If we assign a very high weight into PED, then a lot of candidates which are very likely based on n-gram model but with higher PED would be ignored; If we attach too much importance to the role of n-gram, then we may see more words which appear in the corpus in high frequency but with higher PED. In contrast, a lot of words will disappear which

match our wish better because of lower PED but they do not come so frequently. POS-Tagging plays an important role but still faces a similar problem.

Firstly, we put our attention into the case that needs less POS-Tagging. Since we expect the candidates with high probabilities based on n-gram model and smaller PED with a given input, we should let the n-gram probability be limited to the PED. Hence, the weight assigned to PED would be a penalty weight. Then we have the formula as follows:

$$\text{probability} = \text{n-gram probability} - \text{weight} \cdot \text{PED}$$

In this paper, the weight above was set to 1.0. We will prove its rationality in the section of the evaluation.

Lastly, we can think about the attendance of POS-Tagging refer to the method above. However, we could not directly add the POS-Tagging probability into the sum, because POS-Tagging probability may be very tiny and it could not bring too much. For example:

candidate1

$$\text{n-gram probability} = 0.00015 \tag{7}$$

$$\text{POS-Tagging probability} = 0.000045 \tag{8}$$

$$(7) + (8) = 0.000195 \tag{9}$$

$$(7) \cdot (8) = 6.75e - 9 \tag{10}$$

candidate2

$$\text{n-gram probability} = 0.00019 \tag{11}$$

$$\text{POS-Tagging probability} = 0.000021 \tag{12}$$

$$(11) + (12) = 0.000211 \tag{13}$$

$$(11) \cdot (12) = 3.99e - 9 \tag{14}$$

POS-Tagging probability should help improve the rank of candidate1. However, the sum of (7) and (8) is still smaller than that of (11) and (12) and it can not protrude the effect of POS-Tagging. If we multiply n-gram probability and POS-Tagging probability, the final results of (10) and (14) show that candidate1 has higher priority than candidate2, since candidate1 makes more sense from the perspective of grammatical rules. Therefore, our final formula to integrate these three factors is:

$$final\ probability = (\text{POS-Tagging probability} \cdot \text{n-gram probability}) - 1.0 \cdot \text{PED}$$

3.5 Vocabulary

Before processing the text in the corpus, we have no idea which words the texts may contain. It is necessary to estimate the n-gram model with a fixed vocabulary. On the one hand, the corpus sometimes contains a lot of content under the same topics. So we can not depend on the vocabulary collected from words in the corpus. Because of limited text cleaning, some words may make no sense but would also be gathered into the vocabulary, which will affect the performance of the model. On the other hand, adding words with relatively low frequency would enlarge the model which causes a long calculating time.

Therefore, we will firstly choose a pre-filtered vocabulary by MIT which contains 10000 words that are commonly used in English dialogue [17]. And then we choose

1000 words with the highest frequency from corpus which do not belong to the MIT words. With these 1000 words, we will replace the 10 percent of the original 10000 words which appear very rarely in the corpus. This will actually work very well because it has avoided adding nonsensical words and could consider some special usage of English words like “I’m”, “I’ll”, “we’re” and so on.

3.6 Corpus

The size of a corpus and its content are significant to build n-gram models. The effectiveness of statistical natural language processing techniques is highly susceptible to the data size. As a statistical NLP technique, the performance of the n-gram model is affected not only by the size of the corpus but also by the content of the corpus. If corpus for training consists of only one or two topics, then the trained n-gram models will have very bad adaptability and are very theme-specific. Imagine that if you have a big corpus but only consists of the topic of Brexit (British exit from EU). You may have such combinations as “Scottish people” or “Camelon decided” at a very high frequency just because this corpus included too many contents under the Brexit topic. If you type “sc” then you get the “Scottish” as the first candidate in the list. That is actually very annoyed because there are a lot of other words which are used more commonly than “Scottish” like “school”, “scenery”, “scene” or “scholar”. Mostly, you do need the word “Scottish”. In our application, we take the corpus whose content is retrieved randomly from 95000 websites that cover different themes and consists of 14 million words [18].

3.7 Training and data processing

Getting the Q-grams of the words from vocabulary will be always repeated and it takes a lot of time to calculate Q-grams of all words. The best way is to finish this

process in advance and store Q-grams Index of all words somewhere so that we do not have to do this calculation repeatably. As explained before, the number of common Q-grams decides if two strings intuitively deserve to be compared.

After we determine our vocabulary and corpus, we could train n-gram models. Since the whole corpus contains about 13 million words and occupies almost 1 GB of physical storage, we will separate the corpus into 10 small corpora so that size of the heap could fulfill the requirement. According to the length of a sequence, our n-gram models consist of 5 kinds of grams: unigram (1-gram), bigram (2-gram), trigram (3-gram), quadrigram (4-gram), and quinquegram (5-gram). When building the n-gram models, we will set a threshold of frequency so that we only store bigrams and trigrams which appear more than 30 times in the corpus. Such threshold for quadrigram is 20 and for quinquegram is 15, because there are relatively fewer identical long sequences in the corpus. Hence, we need first to build the unigram model which contains only a single word, then the bigram until quinquegram separately from every single corpus. Those n-grams or sequences with n words must be composed of words from the vocabulary. This could guarantee the adaptability of n-gram models since the words of vocabulary cover more themes and topics. At last, we combine all models and filter out n-grams that appear at a low frequency.

Type	Amount
unigram	10000
bigram	84839
trigram	42322
quadrigram	20129
quinquegram	11600

Table 1: Amount of n-grams

Again, a lot of combinations do not exist in the corpus but they follow grammatical rules. To deal with those cases, we could make the keyboard know about grammar so that it could return the right candidate. This so-called grammar is supported by

two parts: transition probabilities and emission probabilities. Firstly, we need to find out all parts of the sentence which a word could be and their emission probabilities (observable). Secondly, we explore the possibilities of all transitions from one group of words to another sort of words and their transition probabilities (hidden).

With NLTK we could calculate emission probabilities for every word from vocabulary and transition probabilities of POS-tags. After processing all sentences in corpus, the results will contain all possibilities of tag a word could be, for example, emission probabilities of word *like*:

Tag	Frequency
IN	84839
VB	10000
JJ	42322
VBP	10322
...	....
EX	1
RBR	1

Table 2: Emission probabilities of **like**

And transition probabilities of the POS-tag RB (adverb) could look like this:

Tag	Frequency
JJ	106766
NNP	10035
VBZ	28452
VBP	25094
... ...	
SYM	9

Table 3: Transition probabilities of **RB**

Actually, NLTK could not tell third-person pronouns from first-person nouns, for example, “I” would be categorized as “PRP” in the sentence “I want to have a drink” and “he” would also be categorized as “PRP” in the sentence “He wants to have a drink”. Therefore, we created a new label “PRPZ” manually which annotates “he” or “she” when they appear as third-person nouns in the sentence.

3.8 Model and Data storage

3.8.1 Motivation

The n-gram model, Q-gram index, and other data should be accessible for the keyboard when it runs, so we should think about an effective way to store data. The first idea is that we store those data as binary objects into several .txt files. When the keyboard is called, it can read those .txt files and load binary objects into the memory and use them directly. Then we will not have to read the file again and again after the first load. Because all data has been stored in the memory, then we could access it anytime. However, the disadvantage is difficult to be ignored: it takes a too long time to load all data at once and the time is highly relevant with the size of files. That means one needs to wait for a “long” time before the keyboard is ready to use. That works if people use the keyboard only to make a test or evaluation since the waiting time does not play a big role here.

However, if we use it as a system keyboard and it should appear on the screen quickly whenever we call it. No matter where we store the models and other data and no matter how we visit those files, it takes always very long to read and load them once. But is it necessary to load all data at once, especially when we still have not inputted anything? For example, our initial input is “Have ”, and the keyboard just needs to know which words would come after “Have” at high frequency. That is the only piece of data we need. Now the new logic is to load data that we need at the moment.

Reading and loading data will be carried out with every keystroke but the loading time is much shorter.

3.8.2 Database

To achieve this goal, we store the model and other data in a database. A single query will be carried out only when we need pieces of information in a table. Since a single query in SQLite could take 20 to 500 ms, we should create a proper structure in a database to store data so that the number of queries could be minimized.

Our data consists of 4 parts. emission probabilities of vocabulary, transition probabilities between two POS-tags, Q-grams of vocabulary, and n-gram models. As explained above, a response time feels interactive until around 200 ms. Therefore, we ought to execute as few queries as possible and every query should fetch as much information as possible.

	QGRAM	ENTITY	NUM
	Filter	Filter	Filter
1	cca	occasion	1
2	cca	occasional	1
3	cca	occasionally	1
4	cca	occasions	1
5	cca	rebecca	1
6	ksh	workshop	1
7	ksh	workshops	1
8	ksh	yorkshire	1
9	cce	acceleration	1
10	cce	accent	1
11	cce	accept	1
12	cce	acceptable	1
13	cce	acceptance	1
14	cce	accepted	1
15	cce	accepting	1
16	cce	accepts	1
17	cce	access	1

Figure 8: Q-gran Indices

Given a string s and delta as the threshold of PED, we want to firstly find out all candidates who deserve to be compared in the next step. That means we need to count the number of common Q-grams between string s and every word by one query, because it takes too much time to execute a separate query for every word. We will design the table as shown in Figure 8: we have three attributes for every tuple, namely, QGRAM, ENTITY, and NUM. Every tuple represents that which Q-gram appears in which word and at which frequency. In this way, we could get candidates which fulfill the requirement through one query command:

```
SELECT count(*), ENTITY from IDX WHERE QGRAM = qgram1
or QGRAM = qgram2 ..... or QGRAM = qgramn)
group by ENTITY having count(*) > threshold
```

After we find all candidates which fulfill the requirement for PED, we need to calculate the probability of every candidate based on the n-gram model. To reduce the number of queries, we could add more sequences with common words to a row as shown in Figure 9. In addition, we stored corresponding frequencies after every n-gram so that we could directly use them to make a calculation.

a	2942699	a a	894	a	2942699	null	null	null	null
a	2942699	aboard a	60	aboard	1913	null	null	null	null
a	2942699	about a	9839	about	286630	for about a	524	for about	4268
a	2942699	about a	9839	about	286630	how about a	94	how about	1598
a	2942699	about a	9839	about	286630	is about a	79	is about	6328
a	2942699	about a	9839	about	286630	story about a	57	story about	1025
a	2942699	about a	9839	about	286630	talk about a	51	talk about	7608
a	2942699	about a	9839	about	286630	talking abo...	156	talking about	7149
a	2942699	above a	263	above	31797	null	null	null	null
a	2942699	accept a	333	accept	7744	null	null	null	null
a	2942699	accepted a	67	accepted	6107	null	null	null	null
a	2942699	accepts a	31	accepts	1124	null	null	null	null
a	2942699	access a	173	access	33173	null	null	null	null
a	2942699	accommoda...	39	accommodate	2246	null	null	null	null
a	2942699	achieve a	894	achieve	9055	to achieve a	621	to achieve	5667
a	2942699	achieved a	121	achieved	4075	null	null	null	null
a	2942699	acquire a	84	acquire	1951	null	null	null	null

Figure 9: N-gram models

ENTITY	TAG1	FREQ1	TAG2	FREQ2	TAG3	FR
Filter	Filter	Filter	Filter	Filter	Filter	Filter
a	DT	0.98982934035715	NNP	0.0100110619049677	NN	0....
aa	NNP	0.959259259259259	NN	0.0271604938271605	VBP	0.013580
aaa	NNP	0.94954128440367	JJ	0.0298165137614679	NN	0.020642
aaron	NNP	0.989172467130704	NN	0.0100541376643465	VB	0....
ab	NNP	0.906086956521739	NN	0.0695652173913043	JJ	0.024347
abandoned	VBN	0.679540229885057	VBD	0.222528735632184	JJ	0.097931
abc	NNP	0.982315112540193	NN	0.0112540192926045	JJ	0....
aberdeen	NNP	0.966010042487447	JJ	0.0312862108922364	VB	0....
abilities	NNS	0.973552211582307	NNP	0.0182398540811674	NNPS	0....
ability	NN	0.96946392218955	NNP	0.0305360778104501	null	null
able	JJ	0.998142829377848	NNP	0.0018139806076835	VBP	4....
aboard	IN	0.675801749271137	RB	0.173760932944606	NN	0.150437
aboriginal	NNP	0.674326986211425	JJ	0.320420223243598	NN	0....
abortion	NN	0.972185430463576	NNP	0.0278145695364238	null	null
about	IN	0.940095762899449	RB	0.058016939082214	RP	0....
above	IN	0.802349554596024	JJ	0.119928424164624	NN	0.077722
abraham	NNP	0.996495327102804	NN	0....	VB	0....
abroad	RB	0.943704340352385	NNP	0.0550064460678986	JJ	0....

Figure 10: Emission probabilities

The structure for transition probabilities and emission probabilities is relatively simple. To simplify the calculation and avoid unnecessary POS-tags, every word only keeps three tags with the highest probabilities. Because of the flexibility of language, a word could sometimes be used in a very strange way, and such extreme cases should be excluded. Hence, only 3 tags of a word would be kept and it also follows the grammar rules in daily dialogue. Transition probabilities of POS-tags will be stored in the form of a string. The time for parsing could be ignored since NLTK only supports 45 tags.

WRB	0.08729351801724479_JJ	0.04320588265499995_NN	0.001546368308545...	
WP$	0.17819222601791154_JJ	0.4925676299510664_NN	0.0041547410211430...	
WP	0.03868126237794484_NN	0.010440334610950725_JJ	...	
WDT	0.030619751215507987_NN	0.009544105575550925_JJ	4.7988886784113...	
VBZ	0.09644259761835443_JJ	0.012052779774132535_``	...	
VBP	0.027015785739349325_NN	0.006618297657395166_``	0.0891906196702...	
VBN	0.04747668180504849_NN	0.03243966011158198_JJ	0.007288083797437...	
VBG	0.11374357635724418_NN	0.06366747902166355_JJ	...	
VBD	0.035978131503910146_NN	0.055262927312049626_JJ	...	
VB	0.0729541885730344_JJ	0.05568459508042699_NN	0.007889215872933505_``	...
UH	0.033483544995220535_NN	0.00453366106786836_JJ	0.00609040010924...	
TO	0.020011412538455293_JJ	0.0028500333805472605_``	...	
SYM	0.002190923317683881_NN	0.001564945226917058_RB	...	
RP	0.04110406722211666_NN	0.03116009205038588_JJ	0.003314658390576...	
RBS	0.7482391538761557_JJ	0.026601968476839426_NN	0.0018468809507421937_``	...
RBR	0.446108128930267_JJ	0.04715569985642175_NN	0.006403489901996588_``	...
RB	0.11109810464757414_JJ	0.0170031336507335_NN	...	
PRPZ	0.0015477015477015476_JJ	0.0019439557901096362_``	...	

Figure 11: Transition probabilities

4 Keyboard

In this section, we will talk about how our implemented keyboard on an Android device works.

4.1 Implementation and Usage

Implementation of the keyboard is based on the framework of Input Method Editor (IME) supplied by Android, which enables users to input text. After extending the framework, we could design a customized keyboard for users.

The visual components consist of two parts: input view and candidates view. In the input view, users could input by pressing or sliding keys, and the candidates view will show us options to finish our input quickly.

After the system keyboard is installed successfully, users need to choose the keyboard in the "settings" and set the keyboard as the default IME, since Android only permits one IME which runs in the system. The system keyboard could be applied in any field of input, such as in the browser, notebook, message, and so on. As explained before, our keyboard reads data only on demand. When the keyboard is called firstly, it will jump out from the bottom of the screen without observable delay. The keyboard also supports the input of special symbols and mathematical symbols.

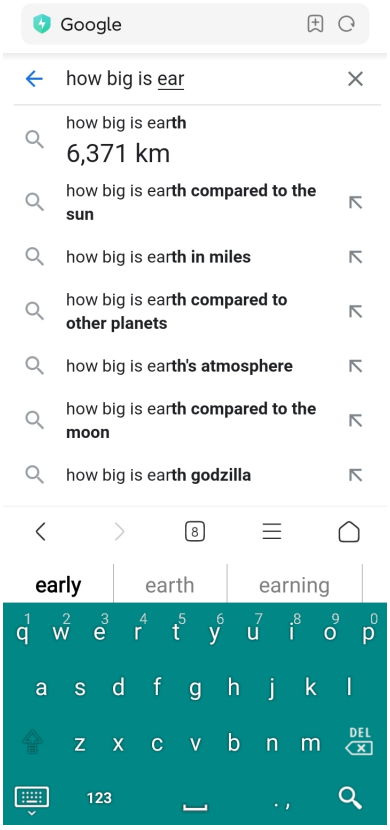

Figure 12: Usage of keyboard on Google search

4.2 Completion, correction and prediction

The whole process will run as shown in Figure 13, when any functionality of completion, prediction, and correction is involved:

- In the state represented by the red circle, the keyboard accepts the user's input or a chosen candidate by the user.

- Program splits the input by punctuation and transfers tokenized parts into the next two states.

- Through the golden rhombus the last part will be used to decide whether the user expects a completion (correction) or prediction.

- In the light orange shadow program computes PED, n-gram probability, and POS-Tagging probability. In this phase, the keyboard needs to interact with the database to load and read data.

- After ranking candidates we will return 3 candidates with the highest probabilities

- We come back to the state represented by the red circle and the user chooses candidate or continue inputting

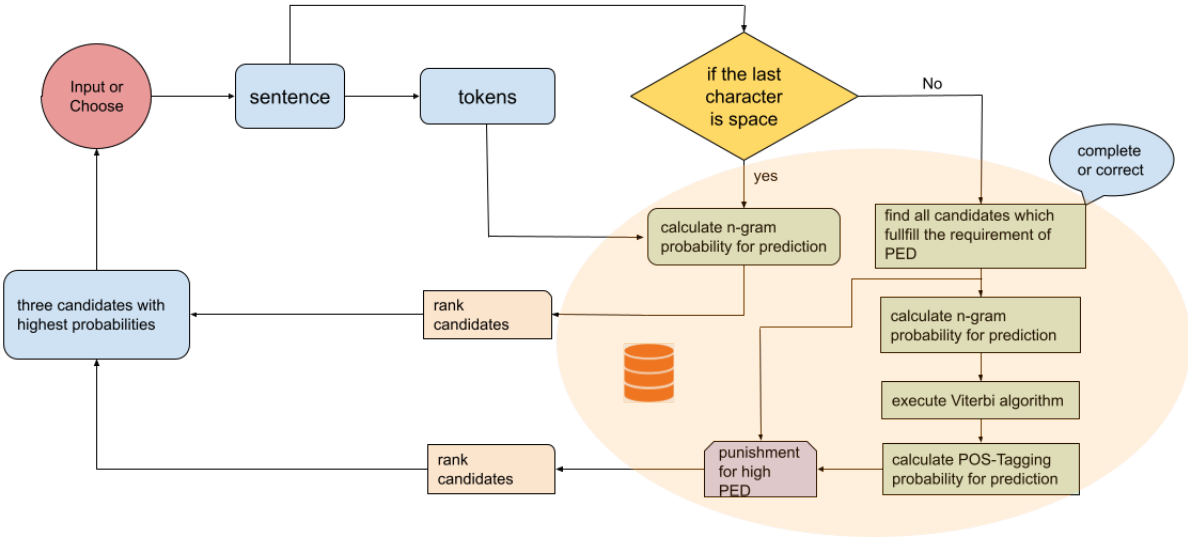

Figure 13: Keyboard interacts with users

4.3 Usage of RAM

RAM is short for Random Access Memory and it could exchange data with the CPU directly at a high speed. As a limited and expensive resource, RAM should be carefully and reasonably used. In a mobile device, RAM becomes more precious because of the physical limit. Most modern mobile devices have at least 2 GB of RAM which makes developers deal with usage of RAM carefully [19]. Since a keyboard will be mostly called and used in another app, the keyboard must not take up too much RAM, otherwise, it will cause an interrupt of the main app or the keyboard and bring users into trouble.

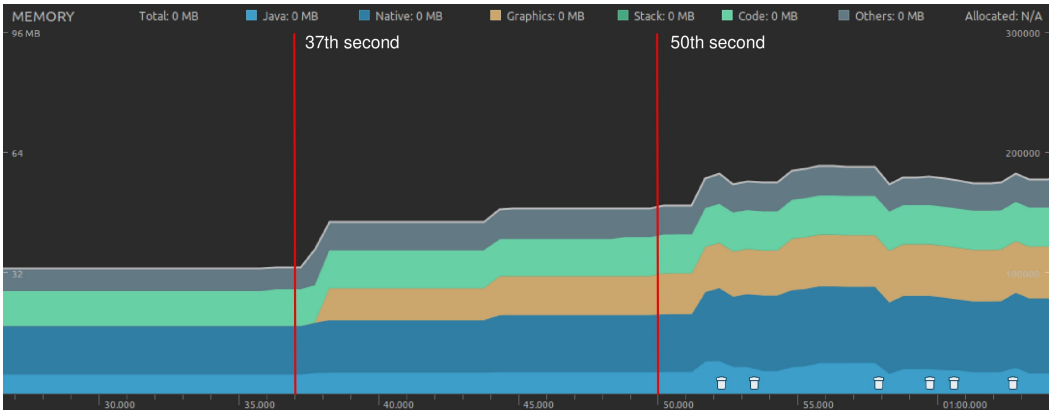

Figure 14: Usage of RAM of implemented keyboard

Our implemented keyboard has achieved the goal that it always takes up a suitable amount of memory. After installing the keyboard on an Android mobile device, we could choose this keyboard as the default IME (Input Method Editor). After this service starts, 33 megabytes of RAM will be allocated for the running service until the 37th second in Figure 14. Then, we call the keyboard in the input field of an app and more RAM will be allocated for the keyboard and now it takes up about 50 megabytes. From the 50th second, we try to enter letters, space, and choose candidates and we could see the occupied RAM varies always between 50 megabytes and 60 megabytes. Because of the event of garbage collection, the unused RAM

will be released back into the heap (see the icons at the bottom) [20]. Since the modern smartphone has at least 2 gigabytes of RAM, we think the usage of RAM is acceptable and reasonable.

5 Evaluation

In this part, we evaluate different aspects of our keyboard. We will test its performance in completion and correction. And then we will show by experiments that how factors such as POS tag, probabilities' weights could influence the performance and if they are useful to build the keyboard.

5.1 Evaluation Method

Data for the evaluation consists of two parts: 5% of the whole corpus and subset of emails generated by Enron's employees [21]. Even the corpus was retrieved from webs under many different themes, its content sometimes does not look like a daily dialogue. Hence, the usage of emails as a test set makes much sense, since emails are more similar to what we use a keyboard on a mobile device to input. The evaluation criterion is the number of saved keystrokes (percentage) by using the keyboard. For the calculation of the number of saved keystrokes, we assume that one needs to type every letter in a sentence by pressing keys without the functionalities of a keyboard. For example:

sentence: Let's have a drink.

length(including space): 19

We assume that one needs 19 keystrokes to finish the input without functionalities of a keyboard. With the functionalities of the keyboard, the number of keystrokes needed will be reduced to 10. In this case, there are two types of keystrokes that we will count into the sum of strokes needed. The first one is to press a key that represents any character, punctuation, or space. The second one is to choose a candidate which has been shown in the candidates' list on the top of the keyboard. So the percentage of saved keystrokes will be:

$$\frac{19-10}{19} \approx 47.3684\%$$

The number of keystrokes used to change capital mode or to find symbols will not be counted into the total number of strokes, since this paper will only focus on the program solution but not the layout design.

At first, we will determine the weights that are assigned to different n-gram probabilities. Through experiments and adjustments, we will apply different combinations of weights to achieve the most optimal performance. The first combination of weights as initial values for unigram, bigram and trigram is as follows [22]:

$$\lambda_1 = 0.1 \quad \lambda_2 = 0.3 \quad \lambda_3 = 0.6$$

After determining values of λ_1, λ_2 and λ_3, we assign different values to λ_4 for the weight of quadrigrams and then to λ_5 for the weight of quinquegrams. Then we could find out the best combination.

Secondly, we want to test if POS-Tagging could help improve the performance of our keyboard. On the set of 5% of the corpus, we will separately execute our test program with POS-tagging probability and without POS-tagging probability. That effect of grammar rules will be shown by the difference between the two experimental results.

And then with different penalty values based on PED, we aim to find which value would maximize the improvement by penalty weight.

In the next, we want to verify if a n-model with longer sequences will work better, for example, with trigrams or quadrigrams. In the meanwhile, we also aim to find if the 5-gram could bring more improvements.

After that, for the test of correction rate, we will replace the first letter of every word with a wrong one. It is assumed that if we find one spelling mistake in the word, we need two more keystrokes to correct it (delete and insert). As a consequence, the total number of keystrokes to input a wrong word should be two more than that of a non-error intend word. We will choose 100 sentences from the set of Enron emails and input them separately with our implemented keyboard and the newest version of Gboard which was published in May 2021. And then we can compare the performance of our keyboard with that of Gboard.

At last, we make tests for the time which the keyboard spends in making queries for candidates from the database, calculating the PED of candidates, and selecting the best candidates by probabilities. In addition, we want to know how long on average it takes from keystroke is pressed to candidates will be shown.

5.2 Evaluation Results

We first test the performance of the keyboard with different combinations of n-gram weights as shown in Table 4. The initial combination, namely $\lambda_1 = 0.1$, $\lambda_2 = 0.3$ and $\lambda_3 = 0.6$ works better than without any weights ($\lambda_1 = 1.0$, $\lambda_2 = 1.0$ and $\lambda_3 = 1.0$). When we test different combinations of weights, we will only change the value of one weight in an experiment and keep the other two weights unchanged. As we constantly increase the weight to trigram probability λ_3, the performance has been being improved until λ_3 becomes 3.0. Then we also try to increase the weight to

λ_1	λ_2	λ_3	Saved keystrokes(%)
1.0	1.0	1.0	39.7598%
0.1	0.3	0.6	39.8926%
0.1	0.3	1.0	39.9020%
0.1	0.3	2.0	39.9092%
0.1	0.3	3.0	39.9095%
0.1	0.3	4.0	39.9091%
0.1	0.5	3.0	39.9354%
0.1	0.8	3.0	39.9537%
0.3	1.0	3.0	39.9588%
0.3	1.5	3.0	39.9589%
0.3	1.8	3.0	39.9596%
0.3	2.0	3.0	39.9597%
0.3	2.5	3.0	39.9603%
0.3	3.0	3.0	39.9563%
0.5	2.5	3.0	39.9159%
0.01	2.5	3.0	39.9668%
0.005	**2.5**	**3.0**	**39.9670%**
0.001	2.5	3.0	39.9667%

Table 4: Test for n-gram weights

bigram probability λ_2 and the performance comes to the top when $\lambda_2 = 2.5$. After that, we apply the same method to tune λ_1. However, increasing the value of λ_1 brings a negative effect on the performance. Therefore, we reduce the value of λ_1 until 0.005 since there is no more improvement for performance in the trend. Hence, the combination of values of λ_1, λ_2 and λ_3 we choose is as follows:

$$\lambda_1 = 0.005,\ \lambda_2 = 2.5,\ \lambda_3 = 3.0$$

Besides, we also want to evaluate the performance of the keyboard with attendance of quadrigram and quinquegram. Therefore, based on fixed values of λ_1, λ_2 and λ_3 that

we have explored, we intend to find out the most suitable weights for quadrigram and quinquegram, namely λ_4 and λ_5. We increased the weight of $\lambda 4$ at first and find the

λ_1	λ_2	λ_3	λ_4	Saved keystrokes(%)
0.005	2.5	3.0	3.0	39.9772%
...	...	...	4.0	39.9776%
...	...	...	**5.0**	**39.9778%**
...	...	...	6.0	39.9778%

Table 5: Test for quadrigram weights

optimal value should be 5.0. To find if a small value could bring more improvements, we set the value as 3.0 (same as the value of λ_3) and it proves that the performance gets even worse in this way.

λ_1	λ_2	λ_3	λ_4	λ_5	Saved keystrokes(%)
0.005	2.5	3.0	5.0	4.0	39.9790%
...	...	...	...	6.0	39.9790%
...	...	...	...	7.0	39.9790%

Table 6: Test for quinquegram weights

Applying the same method to determine the value of λ_5, we found that the performance does not change with the different values of λ_5 as shown in Table 6. At last, we choose 6.0 as the value of λ_5.

We have before attached much importance to the role of the trigram model since it considers more contexts. However, the results of the experiment in Table 7 show that the role of trigram has been overestimated. And quadrigram (4-gram) and quinquegram(5-gram) could only bring tiny improvements. An important reason is when the trigram probability of candidate 1 is higher than that of candidate 2, mostly the bigram probability of candidate 1 is also higher than that of candidate 2. For example:

Gram	Saved keystrokes(%)
bigram	39.2942%
trigram	39.9670%
quadrigram	39.9778%
quinquegram	39.9790%

Table 7: Test for different grams

Finished part of sentence: *Maybe he is one of th*

Candidate 1: them

$$bigram\ probability : 0.0069$$

$$trigram\ probability : 0.0318$$

Candidate 2: these

$$bigram\ probability : 0.0091$$

$$trigram\ probability : 0.0360$$

No matter we take the sum of two probabilities or use the trigram probability directly as result, the trigram probability cannot influence the rank of two candidates in this case. Therefore, trigrams could help reduce the number of keystrokes only if trigram probability could greatly compensate for the disadvantage of the bigram probability of a candidate. And the results show those cases only account for a small part. The insignificance of quadrigram(4-gram) and quinquegram(5-gram) could also be owed to the same reason. In addition, longer grams can very rarely match a finished part of a sentence, especially for quinquegram.

In the next, we look into the results of the evaluation for the role of POS-tags. The improvement by POS-tags is observable but also limited like in Table 8. The reason

Mode	Saved keystrokes(%)
Without POS-Tags	39.3114%
With POS-Tags	39.9790%
+	0.6676%

Table 8: Test for the POS-tagging

is that our n-gram model has also done the same work as POS-Tagging. We have previously pointed out that the absence of some combination is the biggest motivation to add the role of POS-tagging. Therefore, if an n-gram appears repeatedly at high frequency in the corpus, it has implied such a combination follows a grammar rule. Once a combination exists, the function of POS-tagging will not stand out.

Value	Saved keystrokes(%)
0.0	25.3724%
0.0005	33.6434%
0.005	37.8342%
0.05	39.7523%
0.5	39.9719%
1.0	39.9790%

Table 9: Test for the penalty weight

We could see from Table 9 that at the very beginning the performance could be improved by the increased value of penalty value strongly. After the penalty value reaches 0.5, the performance has been very stable and will not change a lot with the penalty value. Therefore, ranking the candidates by the penalty value is reasonable and necessary to improve the performance of our keyboard. So 1.0 will be set to the value of the penalty weight.

Then we selected 100 sentences from Enron emails to evaluate the functionality of correction. Enron emails are the ones of very few collections of "real" emails that are public. Here we also make a comparison between Gboard and our keyboard (denoted

as Nboard).

Keyboard	No mistake	With mistakes
Gboard	48.8138%	19.8448%
Nboard	45.6404%	21.5652%

Table 10: Test for the correction

We could see that Gboard has performed very well in reducing the number of strokes since it has saved almost half of the keystrokes needed. And Nboard has also achieved a similar level with Gboard which is about 3.2% less than that of Gboard. Gboard was trained by a modern neural model and is being trained constantly by millions of users by the framework of Federated Learning [10]. The decentralized learning method makes it possible to train the language model of Gboard based on the user's input and language habits.

Nboard wins Gboard slightly in correcting the wrong words. Since we added spelling mistakes by replacing the first letter of every word with "a" or "b" ("A" or "B") and deactivated the functionality of prediction, the correcting work became very hard for two keyboards. Even though this design could not completely simulate the inputting habits of users, the result still shows the performance of our keyboard is acceptable in correcting mistakes, compared with the performance of Gboard on the identical experimental method.

No mistake	With mistakes
40.9789%	20.7947%

Table 11: Test with 2400 sentences from Enron emails

After that, we choose a larger set from Enron emails to test the adaptability of our keyboard. The test set from corpus was retrieved randomly from websites, so the syntax and grammar could be more complicated than that of normal emails. Therefore, the result shows the keyboard performs better on the test set of emails

than on the 5% of the corpus. Nboard could improve user experience better when it processes such dialogue in life and work.

User experience will also be affected by the running speed of the app. We tested the total and average running speed of different stages when our keyboard processes a user's input (100 keystrokes) and reacts with the corresponding calculation:

Type	Duration	Average
From keystrokes to candidates	6947 ms	69.47 ms
PED	580 ms	5.80 ms
Query	2047 ms	20.47 ms
Generating candidates by model	6370 ms	63.70 ms

Table 12: Test for the running speed at different stages

As shown in Table 12, after a user presses the key, he or she could expect to see the three best candidates after 69.47 ms on average. Every time when a user presses a key, the keyboard will judge which calculation should be executed. If the last character is not a space, then the keyboard will look for all candidates by calculating PED between the last part of the input and possible words. This process takes 5.80 ms pro keystroke as shown in the second line in Table 12. During the calculation of PED, the keyboard needs to load Q-grams of candidates from the database. In addition, queries will also be made by the keyboard when it needs to read the frequency of a word or POS-tags of candidates. All those queries take about 20.47 ms pro keystroke. Then we use a cursor as an interface to access the result set from queries, namely, to read every row in the result. After fetching necessary data, the keyboard calculates n-gram probabilities and POS-tagging probabilities of candidates. After integrating different factors and weights, our keyboard will generate the three best candidates for the user. The model generates the best candidates by reading data and executing calculations and this process takes 63.70 ms on average. Hence, a person could use our keyboard to save much work in inputting and also without time issues.

6 Current Problem and Future work

6.1 Current problems

POS-tags could help improve the performance of our keyboard in some situations and it depends mostly if we could determine extensive and precise linguistic features for vocabulary. We used NLTK to make the grammatical annotation for every word in the corpus and this tool has saved us a lot of time. However, NLTK could not always make a correct annotation and may cause a problem. For example, in our corpus there is such a sentence: "I have had Tado now for around 2 months, everything works perfectly apart from Auto app on android phone." Obviously, "everything" is a noun here, and "works" plays a role of a verb in third-person form. Unexpectedly, NLTK has taken "works" as a form of a plural noun. In the process of annotation by NLTK, this mistake appears so many times that the frequency of the combination of singular noun + plural noun is higher than that of the combination of singular noun + verb in third-person form. This causes a problem directly. When a sequence does not appear in the corpus, for instance, "a dog drinks" is never to find in the corpus and the keyboard should complete the next word based on a part of the sentence "a dog dri" with grammar rules. But the keyboard can not show the "drinks", because the probability of noun + verb in third-person form is apparently lower than other combinations.

In addition, NLTK can only annotate part of a sentence from a limited set of tags. As mentioned in section 3, the subject of a sentence could only be annotated as PRP

no matter it is a third-person form or first-person form. We have applied an interim solution so that “he” or “she” as the subject of a sentence could be annotated as “PRPZ”. There are still other similar unknown and known cases that could reduce the accuracy of the probability of POS tags.

It is assumed that a user always knows what he or she needs to input. That means when a user is trying to input a complete sentence, every part of the sentence has a close connection with each other. For example, given a sentence “he is going to a bank in downtown now because he needs money”, the existence of “now” could be due to the status of the person’s action “going”, and because “bank in downtown” refers to a financial institution but not a geographical location, “money” should be placed after “needs” but not “mussel”. Hence, no matter which method we are applying, the context we take care of is still not enough to ensure a very precise prediction or completion.

6.2 Future work

In the light of limited time and computational resources, this paper was not able to pay enough attention to the referred problems above. In the future, there should be more work to be done to extend the language model. Especially when completing or predicting the next word given the context, the keyboard should be trained to pay more attention to every part of inputted part of the sentence. The model Transformer developed by Google could be a potential solution since the “self-attention” mechanism just fulfill the requirements. Besides, people could customize existing NLP tools like NLTK on-demand and improve the performance of the tool so that it could be adapted to a specific training task. With further development of NLP technology, there will be more user-friendly and efficient keyboards coming into the view.

7 Conclusion

In this paper, we described how classical n-gram models and other relevant theories could be applied to complete, correct, and predict the next word. Based on these theories, we have implemented a system keyboard that could be used on an Android device. Then we evaluated different aspects of the keyboard, such as assignments of penalty weights, the performance of different n-gram models in saving keystrokes, and so on. In addition, we used Google Board (Gboard) to execute some same tasks as an important comparison, since it was trained by a modern neural network. Through analysis of experiment results and comparison with Gboard, we demonstrate that the application of classical n-gram models could still have a satisfying performance in completing, correcting, and predicting the next word and the implemented keyboard could bring users a good experience.

Bibliography

[1] S. Jain and S. Bhattacharya, "Virtual keyboard layout optimization," in *2010 IEEE Students Technology Symposium (TechSym)*, pp. 312–317, 2010.

[2] S. Ghosh and P. O. Kristensson, "Neural networks for text correction and completion in keyboard decoding," *arXiv preprint arXiv:1709.06429*, 2017.

[3] W. B. Cavnar, J. M. Trenkle, *et al.*, "N-gram-based text categorization," in *Proceedings of SDAIR-94, 3rd annual symposium on document analysis and information retrieval*, vol. 161175, Citeseer, 1994.

[4] J. B. Marino, R. E. Banchs, J. M. Crego, A. de Gispert, P. Lambert, J. A. Fonollosa, and M. R. Costa-jussà, "N-gram-based machine translation," *Computational linguistics*, vol. 32, no. 4, pp. 527–549, 2006.

[5] L. Bahl, P. Brown, P. de Souza, and R. Mercer, "A tree-based statistical language model for natural language speech recognition," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 37, no. 7, pp. 1001–1008, 1989.

[6] S. Mani, S. V. Gothe, S. Ghosh, A. K. Mishra, P. Kulshreshtha, M. Bhargavi, and M. Kumaran, "Real-time optimized n-gram for mobile devices," in *2019 IEEE 13th International Conference on Semantic Computing (ICSC)*, pp. 87–92, 2019.

[7] S. Bickel, P. Haider, and T. Scheffer, "Predicting sentences using n-gram language models," in *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*, pp. 193–200, 2005.

[8] K. W. Church and W. A. Gale, "Probability scoring for spelling correction," *Statistics and Computing*, vol. 1, no. 2, pp. 93–103, 1991.

[9] G. Kondrak, "N-gram similarity and distance," in *International symposium on string processing and information retrieval*, pp. 115–126, Springer, 2005.

[10] A. Hard, K. Rao, R. Mathews, S. Ramaswamy, F. Beaufays, S. Augenstein, H. Eichner, C. Kiddon, and R. Ramage, "Federated learning for mobile keyboard prediction (2018)," *arXiv preprint arXiv:1811.03604*, 1811.

[11] V. I. Levenshtein *et al.*, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, pp. 707–710, Soviet Union, 1966.

[12] B. Steven and T. Liling, "Natural Language Toolkit." `https://www.nltk.org/`. [Online; accessed 26-October-2021].

[13] H. Antti, "Hidden States." `https://www.cs.helsinki.fi/u/ahonkela/dippa/node34.html`. [Online; accessed 26-October-2021].

[14] S. Edward, "SQLite vs MySQL – What's the Difference." `https://www.hostinger.com/tutorials/sqlite-vs-mysql-whats-the-difference/`. [Online; accessed 26-October-2021].

[15] J.-H. Lee, M. Kim, and H.-C. Kwon, "Deep learning-based context-sensitive spelling typing error correction," *IEEE Access*, vol. 8, pp. 152565–152578, 2020.

[16] Wikipedia, "Viterbi algorithm — Wikipedia, the free encyclopedia." http://en.wikipedia.org/w/index.php?title=Viterbi%20algorithm&oldid=1049907873, 2021. [Online; accessed 25-October-2021].

[17] E. Price, "MIT word llist." http://www.mit.edu/~ecprice/wordlist.10000. [Online; accessed 26-October-2021].

[18] Corpusdata, "iWeb sample." https://www.corpusdata.org/iweb_samples.asp. [Online; accessed 26-October-2021].

[19] E. Collins, "How Much RAM Does a Smartphone Need?." https://www.makeuseof.com/how-much-ram-smartphone-need/. [Online; accessed 26-October-2021].

[20] Google, "Inspect your app's memory usage with Memory Profiler." https://developer.android.com/studio/profile/memory-profiler. [Online; accessed 26-October-2021].

[21] L. Kaelbling and M. Gervasio, "Enron Email Dataset." https://www.cs.cmu.edu/~enron/. [Online; accessed 26-October-2021].

[22] J. Coleman, "Trigram model calculations ." http://www.phon.ox.ac.uk/jcoleman/old_SLP/Lecture_6/trigram-modelling.html. [Online; accessed 26-October-2021].