CS 4410 Final Project Report

Lux Zhang (yz862)

December 2021

1 Overview

Modern Chinese writing uses over 8,000 unique characters, yet most Chinese speakers use only the alphanumeric keys to type. Therefore, mapping from strings representing the pronunciation of Chinese words, or pinyin (e.g. "putonghua"), to particular Chinese characters, or hanzi (e.g. "普通话"), is a well-established problem. Software components solving this problem are called input methods or input method editors (IMEs).

One problem is that IMEs are interactive; they require user intervention to perform conversions. For instance, a user might type a few syllables, hit the space bar to begin a conversion, and then press a number key or arrow key to choose the correct conversion out of a list. This makes IMEs unsuitable for use in ML systems which, for instance, convert speech to text or which transliterate phonetic data to hanzi. In systems such as these, there is no interactive user input, and the program must attempt to find the most likely conversion without supervision.

In this work, I attempt to address this problem by designing, implementing, and evaluating the performance of a non-interactive pinyin to hanzi converter for Chinese which uses a newly built N-gram language model and beam search. I demonstrate that the system is capable of achieving approximately 93% accuracy character-by-character and 73% Top-1 accuracy for sentences from news, wiki, and internet datasets. Further, I present a detailed performance analysis for the converter which characterizes its accuracy, time efficiency, and memory efficiency across variations in beam width, N-gram order, quantity of training data, and source of training data.

2 Background

Input method editors have been in use since the first languages with non-Latin orthography were implemented in operating systems. Microsoft Windows 3.0 first added multilingual support to desktop platforms. However, due to memory restrictions, early IMEs were often inaccurate and limited in vocabulary.

It is instructive to explore some of the best extant techniques for interactive Chinese typing. While most IMEs today are closed-source, Rime [Ju] is an open-source modern IDE for Chinese which uses frequencies of dictionary entries and ad-hoc heuristics to perform interactive segmentation and hanzi candidate generation. Its dataset is in N-gram format, but only contains dictionary entries. An analysis of Rime's frequency data and source code indicates that the corpus which was used to create its language model contained only about 204 million segmented words [Ju21], which presents a challenge for us: Is it possible to surpass this performance with a larger language model?

Many more sophisticated approaches are in the literature. One of the most modern and representative of them is due to Jia and Zhao [JZ14], who generalize from N-grams to a hidden Markov model and then build a graph such that the shortest path from start to end will solve the Viterbi algorithm and find the most likely characters. Their implementation is novel because it simultaneously performs typo correction, word segmentation, and pinyin-to-character conversion in a single graph operation. Their performance is excellent for character-by-character accuracy at 96.4%, but for entire sentences, they only achieve 40% Top-1 accuracy, which means that their method may be less suitable for non-interactive use.

Finally, several researchers have experimented with incorporating neural models into pinyin input methods. In 2018, Huang et al. [Hua+18] published Moon IME, which uses an LSTM encoder with 3 layers to vectorize pinyin and a global attentional decoder, which converts the encoder's hidden states into Chinese characters. They achieve 71% Top-1 accuracy on sentences, which is better for our use case, but since these kinds of sophisticated NLP techniques are outside the scope of CS 4700, I chose not to investigate them further in this work.

3 Methods

In this project, I have chosen to take a contrarian approach to the problem of pinyin-to-character conversion. State of the art techniques are either fast, memory-light, but inaccurate, such as Rime, or very complex to implement and increasingly sophisticated, as in Jia and Zhao's joint graph model [JZ14].

Rather than creating a model with even more priors and assumptions, what I have chosen to do is to operate directly on character data with no attempt to segment words or "understand" sentence content. I have taken inspiration from recent trends in NLP research that indicate the most important parameter of a model is its size, and I have allowed the in-memory footprint and the amount of corpus data to compensate for what is likely to be a less efficient but more general technique.

In essence, this paper tests the hypothesis that priors can strengthen a model with weak data or limited storage capacity, but that every prior encodes the assumptions of the humans who build it, and that a simple model with a large memory footprint can actually perform as well as or better than a smaller, more complex system.

3.1 System Overview

Figure 1 shows a block diagram of the ETL stages of the system and its overall function at runtime.

The heart of this system, which I have affectionately named the "Pinyin Moistener", requires two main components to search for the most likely hanzi representation of a pinyin input: the N-gram table, which is detailed further below, and a pinyin-to-character mapping that gives a comprehensive list of all Chinese characters that a pronunciation can correspond to.

The ETL begins on the left with the Corpus Processor, which ingests raw corpora and writes out N-grams to a file on disk. This is then loaded into memory as the N-gram table when the runtime begins.

On the right, we take in raw hanzi data from a dictionary and output a file containing unaccented pinyin readings alongside the hanzi that have that reading.

The Moistener, then, takes in an input pinyin string and an integer β, the beam width of the beam search algorithm, in addition to the N-gram

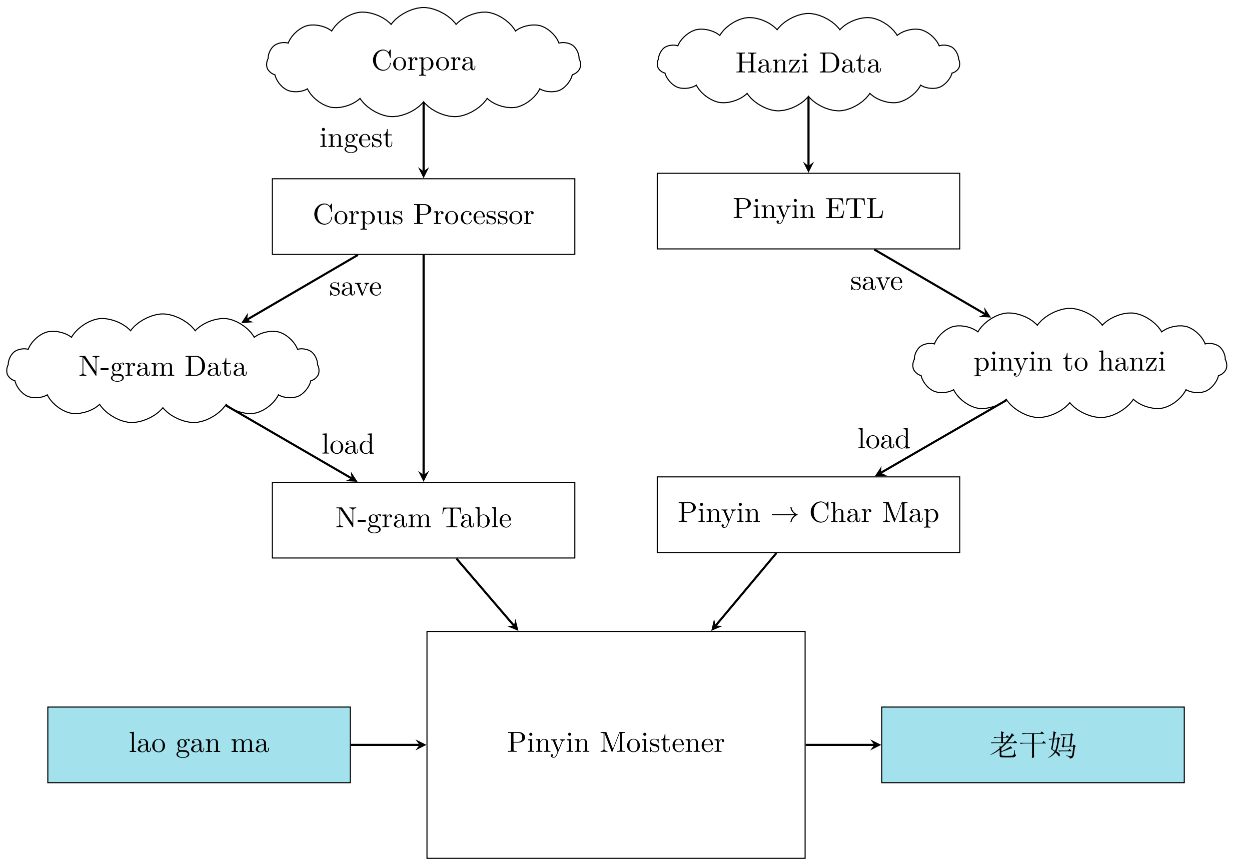

Figure 1. Block diagram showing the overall system.

table and the pinyin-to-character mapping. It keeps track of P, an array of pairs consisting of a hanzi string and its probability of being the correct guess so far. In other words, P is the pruned frontier of the search algorithm.

The Moistener then takes each input pinyin syllable and replaces the existing frontier P, which consists of strings of length n, with every possible Chinese string of length $n+1$. These are saved alongside their probabilities. At the end of each iteration, P is pruned to β length. The output of the Moistener is the contents of P after every reading has been processed, which we hope will contain the correct solution as its first entry.

These outputs are compared against the correct solution in the testing data, from which useful accuracy and efficiency metrics can be computed and used to analyze the performance of the system with varying hyperparameters, which I will discuss further in Section 4.

3.2 Data Sourcing and Preparation

Our language model is trained on 3 million sentences using corpora from the Leipzig Corpora Collection [GEQ12], with 1 million sentences each from 2018 Chinese Wikipedia, 2007–2009 Chinese news articles, and 2015 webcrawl data. Given that each sentence contains on average about 15 words, this puts our dataset at approximately 20 times the size of Rime's dataset.

Corpus data files average around 150 MB, so we ingest them via streaming using the "readline" library. Each line is a mixture of Chinese characters, alphanumerics, and punctuation. Arrays of contiguous Chinese characters within the most common Unicode code range are extracted, and traditional Chinese characters are converted to their simplified equivalents to ensure consistency.

Using the “pinyin” library for Node.js [hot21], I generate pairs of known-good pinyin readings and Chinese sentences they correspond to. There are many homonyms in the Chinese language, so this process is challenging. Word segmentation algorithms for Chinese are strong and seldom wrong, though, so the “pinyin” library can ensure that the hanzi to pinyin conversion is typically unambiguous. The pinyin output is given character by character. The resulting pairs are written out to a “pairs” file, which can then be used to generate N-grams.

3.3 Generating N-grams

My algorithm uses, by default, 1-, 2-, and 3-gram statistics from across the entire corpus. Formally, these statistics approximate the probability that a certain hanzi will occur within a string of natural language text given certain prior characters. A 2-gram, for instance, might approximate the probability that y will occur given that x preceded it. This is typically written as $P(y \mid x)$.

In memory, this N-gram data is stored as a C++ unordered_map from strings to floating point numbers. In particular, if I want to look up $P(z \mid xy)$, I look up the key “xyz” in the map. Details about memory use in practice are given in the results section below.

3.3.1 Smoothing

Since I have taken the approach of utilizing a naive N-gram model with no knowledge of word segmentation and no dictionary, I am relying on the strength of the corpus to supply sufficient statistical data for the language model. However, since our model does not use segmentation, it will happily ingest 2-grams and 3-grams that cross word boundaries, include grammatical particles, and so forth. These N-grams may only occur once in the entire dataset, but form a large proportion of the overall model.

Statistical noise of this form presents a challenge for approximating the probability of such N-grams in input strings during testing. For instance, if a 3-gram “ABC” occurs 0 times in our model, should we therefore conclude that it is impossible? Or if it occurs 1 time, should we bias future occurrences heavily in favor of the pattern we may have seen only because of chance?

The approach I use is called additive smoothing [CG98], which approximates the probability of character c_n following characters $c_1, \ldots, c_{n-1}$ as

$$Pr(c_n|c_1c_2\ldots c_{n-1}) \approx \frac{\epsilon_1 + \#(c_1c_2\ldots c_n)}{\epsilon_2 + \#(c_1c_2\ldots c_{n-1})}$$

where $\#(s)$ indicates the number of times our model has seen s. The epsilon values ϵ_1, ϵ_2 then become hyperparameters for our model.

3.4 Beam Search

The conversion of a pinyin string to its hanzi representation is accomplished by iterating over each character-by-character pinyin reading’s corresponding hanzi characters and finding a path that maximizes the probability of the resulting Chinese character sequence, which means that this process can be treated as a search problem.

Although an exhaustive search algorithm is guaranteed to find the optimal solution, it is impossible to take such an approach. With a branching factor of around 40 on average and a depth equal to the length of the input string in syllables, the search space of this problem is beyond the capacity of modern hardware. On the other hand, beam search's fixed beam size β means that only the top β nodes encountered at each depth will be kept and expanded upon, resulting in a practical memory usage and processing time. Therefore, I decided to use beam search for this application; finding the optimal path is functionally impossible.

4 Performance and Results

4.1 Beam Width

Beam width is the most important hyperparameter of the system. Higher beam widths correspond to a higher probability of retaining the optimal path through the graph of possible pinyin strings, but come with a tradeoff of a nearly linear performance penalty. In other words, for a beam width β, the Moistener's runtime is roughly $O(\beta)$.

The results are shown in Table 1, where I compare different beam widths with their respective performance metrics. Top-1, -5, and -10 accuracy indicate the percentage of testing pairs where the correct Chinese sentence is in the top 1, 5, or 10 results. Character accuracy indicates the percentage of characters in the top result that match the correct solution. We also give the average time for processing a single test sentence.

For small beam widths ($\beta \leq 10$), the accuracy is poor. Diminishing returns start to kick in around $\beta = 30$, and eventually, the accuracy metrics stabilize at $\beta = 100$. The best trade-off between execution speed and accuracy, in my opinion, is at this value. Gaining a few tenths of a percent of accuracy at the cost of doubling average evaluation time is a minuscule improvement when the algorithm is intended for non-interactive use on large inputs.

Width	Top-1 Acc	Top-5 Acc	Top-10 Acc	Char Acc	Time (ms)
5	58.0726	63.4543	63.4543	86.8209	3.94985
10	68.0851	76.4706	76.9712	90.5674	7.8149
20	72.0901	82.9787	84.2303	91.8925	13.0297
30	72.3404	84.6058	85.9825	92.1817	20.09
50	73.2165	85.8573	87.4844	92.3503	31.8671
75	73.3417	86.4831	88.6108	92.4949	45.1071
100	73.592	86.7334	88.8611	92.5431	58.2917
150	73.592	86.7334	88.8611	92.4828	84.5306
200	73.592	86.7334	88.9862	92.519	114.052

Table 1. Performance metrics by beam width.

4.2 N-gram Order

Should we use 1-, 2-, 3-, or 4-grams? The maximum length of strings in our language model is referred to as the N-gram order, and must also be optimized as a hyperparameter.

There are two main considerations here. For one, there is a tradeoff between accuracy and runtime memory usage. As Table 2 shows, 4-grams require 3 GB of memory more than 3-grams! This is a nontrivial quantity for even a powerful desktop PC.

There is a second, more subtle, reason that higher order N-gram models might be a problem. As discussed above, N-grams approximate probabilities, and many sequences of 4 characters will occur very rarely even in a huge corpus. This means that higher-order N-gram models require humongous amounts of data to smooth out the noise in their rarely-occurring character sequences. In effect, what this means is that at some point the corpus size becomes the limiting factor. This is likely why our dataset was effective up to 3-grams, but offered little additional performance with 4-grams.

The cost of a 3-gram model is justifiable; there is a huge accuracy improvement from 1- to 3-grams. However, for our system, the tiny increase in accuracy for 4-grams at a cost of 3 GB of memory indicates that the 3-gram model is likely the optimum on current hardware.

Order	Top-1 Acc	Top-5 Acc	Top-10 Acc	Char Acc	Time (ms)	Memory
1-grams	3.87985	12.015	16.3955	45.115	30.8722	0.01 GB
2-grams	46.3079	68.9612	74.8436	85.05	46.1863	0.2 GB
3-grams	73.592	86.7334	88.8611	92.5431	58.2917	2 GB
4-grams	76.3454	87.1089	89.3617	93.0008	98.4544	5 GB

Table 2. Performance metrics by N-gram order with beam width 100.

4.3 Training Data

Since my approach to pinyin-to-hanzi conversion does not rely on word or phrase dictionaries and other heuristics, the performance of my solution heavily relies on the size of the corpus.

For that reason, I investigated the effect of corpus size on accuracy. I randomly excerpted parts of the corpus and measured results in Table 3. Corpus size has no effect on time to execute (the table shows noise that resolves on repeated runs), yet it has a dramatic effect on accuracy. It is likely that continuing to increase the corpus size would give this algorithm considerable benefit, which is in part a verification of the hypothesis that informed my methods: dataset size can compensate for complexity.

Corpus Size	Top-1 Acc	Top-5 Acc	Top-10 Acc	Char Acc	Time (ms)
30,000	47.6846	65.2065	69.587	85.4355	51.7973
300,000	63.7046	77.3467	80.1001	89.8085	52.0118
1,500,000	70.8385	84.3554	87.1089	91.7118	61.4025
2,700,000	73.592	86.7334	88.8611	92.5431	58.2917

Table 3. Performance by corpus size in sentences with beam width 100.

4.4 Genre Transfer

Since the model was trained using data drawn from web, news, and wiki sources, I got curious whether transfer learning was possible.

In this experiment, the same amount of training data from all sources, web only, news only, and wiki only, were used to train new 3-gram models. They were then evaluated on the same testing data as the previous experiments.

My hypothesis was that accuracy would degrade in any of the models trained on limited data. However, Table 4 shows that while performance degrades in models trained only on news or wiki data, it actually improves when the model is trained on only web data. This outcome is surprising. Additional research will be needed to determine whether this improvement might be caused by the more general, plain style of writing in the web dataset. If so, this represents a powerful avenue for optimizing subsequent models.

Genre	Top-1 Acc	Top-5 Acc	Top-10 Acc	Char Acc	Time (ms)
900K All	69.3367	82.1026	85.2315	91.3745	55.7231
900K Web	73.4668	85.2315	87.234	92.2419	59.2902
900K News	49.562	66.2078	70.9637	86.7365	51.264
900K Wiki	43.4293	59.199	62.5782	82.0744	52.1054

Table 4. Performance by training data genre with beam width 100.

5 Conclusion

In this project, I successfully designed and implemented a system which can convert the pronunciation of Chinese sentences into written Chinese, employing artificial intelligence techniques to make the size of the search space manageable and probabilistic models to approximate the probability of sequences of characters using reasoning under priors. The accuracy of the model approaches or exceeds that of state of the art techniques employing far more sophisticated methods.

The results show that, given a corpus of modest size (3M sentences), the N-gram model hits a performance-to-accuracy sweet spot when it is of order 3. The beam search has a similar optimum at $\beta = 100$ beam width. There is strong evidence that additional research is necessary to discover whether a larger corpus would increase accuracy further and whether restricting the genre of the corpus, however counter-intuitive, might also benefit accuracy. It will be interesting to see what the next decade or two of work on this problem will bring.

References

[CG98] Stanley F. Chen and Joshua Goodman. “An Empirical Study of Smoothing Techniques for Language Modeling”. In: *Harvard Computer Science Group Technical Report TR-10-98* (1998), pp. 2–63.

[GEQ12] D. Goldhahn, T. Eckart, and U. Quasthoff. “Building Large Monolingual Dictionaries at the Leipzig Corpora Collection: From 100 to 200 Languages”. In: *Proceedings of the 8th International Language Resources and Evaluation (LREC’12)* (2012).

[JZ14] Zhongye Jia and Hai Zhao. “A Joint Graph Model for Pinyin-to-Chinese Conversion with Typo Correction”. In: *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics* (2014), pp. 1512–1523.

[Hua+18] Yafang Huang et al. “Moon IME: Neural-based Chinese Pinyin Aided Input Method with Customizable Association”. In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics-System Demonstrations* (2018), pp. 140–145. DOI: `10.18653/v1/P18-4024`.

[hot21] hotoo. *pinyin*. `https://github.com/hotoo/pinyin/blob/master/data/dict-zi.js`. 2021.

[Ju21] Rongshi Ju. *Rime Essay*. `https://github.com/rime/rime-essay/blob/master/essay.txt`. 2021.

[Ju] Rongshi Ju. *RIME / 中州韵输入法引擎*. URL: `https://rime.im/`. (accessed: 12.2.2021).