Chinese Pinyins to Characters Decoding

Jingchen Hu Wenxi Lu

1. Background and Introduction:

This project attempts to convert Mandarin pinyin utterances into the corresponding orthographic Chinese character sequences. Pinyin is the official romanization system for Standard Mandarin. Different from English, French, Spanish, etc, which use phonemic orthography to represent words or phrases, written Chinese is a logographic language whose characters don't carry sounds. Pinyin was invented in the 20th century as a phonetic transcription system to describe Chinese using Latin alphabet. Due to the existence of homonyms, a single pinyin can usually represent more than one characters. In standard Mandarin, there are about 500 legal pinyin syllables, which map to over 6000 common characters. Pinyin-to-character conversion is a core step in a pinyin IME (input method engine) pipeline.

Given a pinyin sequence, our approach attempts to select the best character sequence with the highest joint probability estimation. Ngram language model is widely used for such decoding problems. Since an average modern chinese word contains 2.4 characters, a bigram language model is sufficient for our purpose. Note that each token/unigram need to be a (pinyin, character) pair. For instance, we store the bigram for "你好(hello)" as "你#ni3 好#hao3". We need to include the pinyin information together with the characters because Chinese characters can also have multiple pronunciations.

In the first stage, we mostly focused on parsing data, and implementing the baseline and unigram-based algorithms. In the second stage, we implemented bigram with three smoothing algorithms, built the web demo, and added the option to accept pinyins without tones. We coded everything ourselves since the NLTK n-gram and smoothing modules has documented bugs.

About the web demo

A demo website for this project is running at https://pinyins2chars.herokuapp.com/. Note that the sample test sets generated in the demo are random small subsets of the actual test set we used in testing, which makes the demo faster to run. **Note that the first few queries may take longer since the web service needs to wake from its idle state**. The website used a python Flask server deployed on Heroku. The server exposes a set of HTTP API endpoints for the client to query.

Guide to running the code

The website above should be sufficient to demonstrate all the configurations we support. We recommend using this for grading purposes, since we already configured the database, and the test sets generated are relative small to be able to finish in reasonable amount of time. To run the code yourself, please refer to README.md at the root directory of the project reposirtory. **Note that the server cannot run on Patas since Flask is not installed.**

2. Methodology

2.1 Input / Output

Our algorithm accepts pinyin strings representing utterances as input. We assume that the pinyin tokens are already segmented (separated by spaces), and all the pinyins are spelled correctly. The algorithm is configurable to accept pinyins both with and without tones. That is, each pinyin token should match the regex /[a-z]+[1-5]?/. Here the numbers represent the diacritics used to characterize the tones. An exception is that the pinyins for some special symbols including Arabic numerals are represented by themselves. This is because there aren't any pinyin labels for these symbols in the corpus, but we need to keep them to make the utterances complete. In the end we only output the one most probable decoding as a list of characters. Additionally, the training scripts pinyin2chars.py writes the bitexts, unigram and bigram counts, and Good-Turing smoothed counts to files on disk in Json format so that clients such as the server can query the trained model directly.

3. Dataset and Parsing

We chose to use the Lancaster Corpus of Mandarin Chinese (LCMC) as our corpus (McEnery & Xiao, 2004). It is used for both the pinyin-to-character mapping, and as the source of our training/test sets. The native form of the dataset is XML, which is not easy to work with, Therefore we ported the dataset into SQLite for convenience, following a guide online by Chad Redman (Redman, 2012). To build the bitext used for training and testing, we need to join the character table and pinyin table on the character location identifications. Therefore we added indexes to the database to make it more efficient. When parsing the data, we treated each contiguous chunk of text between punctuations as an utterance, and added start/end symbols for each of them. These utterances are the units of input for our algorithm.

There are 16 text subsets in total in the database and we choose 11 of them to be the training data, and the remaining 5 to be the test data. This is roughly an 80-20 division, where there are 120700 utterances and 4702 unique characters in the training set, and 29146 utterances and 3774 unique characters in the test set. Not all characters in the test set are seen in the training set.

2.2 Algorithm and Implementation

2.2.1 Preselection

As a preparation step we first query the database to build a mapping from pinyins to candidate characters. For each pinyin syllable s_i, build a vector of characters w_i, such that each character w_i,j has the same pinyin s_i.

2.2.2 Baseline:

The algorithm for evaluating baseline is just randomly choosing a candidate character for each input pinyin. Since a pinyin represents multiple characters, the random guess wouldn't generate a very promising result.

2.2.3 Unigram:

For each pinyin pick a matching character with highest frequency / unigram counts (or frequencies sum of all 5 tones for the version without pinyin tones).

2.2.4 Bigram:

We denote the character sequence as W, and the pinyin sequence as S. Here we want to find the character sequence W* with highest joint probability P(W, S) = P(W | S) P(S) (Chen, 2015),

$$
\begin{aligned}
W^* &= \arg\max_{W} P(W|S) \\
&= \arg\max_{W} \frac{P(W)P(S|W)}{P(S)} \\
&= \arg\max_{W} P(W)P(S|W) \\
&= \arg\max_{w_1, w_2, \ldots, w_M} \prod_{w_i} P(w_i|w_{i-1}) \prod_{w_i} P(s_i|w_i)
\end{aligned}
$$

In our task, since the the probability of the pinyin (s_i) given a character (w_i) is always 1 (we pre-selected the characters for each pinyin), $\prod P(s_i|w_i)$ is always 1. Therefore, we only need to consider about $P(w_i|w_{i-1})$. Applying the markov assumption for bigram and take the log gives us::

$$W* = argmax \sum logP(wi|wi-1)$$

where P(wi|wi-1) is the MLE probability: $P(wi|wi-1) = \frac{C(wi-1, wi)}{C(wi-1)}$

We now want to maximize the sum of the log conditional probabilities . We can view the characters as nodes in a graph -- the nodes are in layers, where layer i contains all nodes/characters with the pinyin s_i. The edge between any two nodes w_i-1,j and w_i-1,k in two neighboring layers have the weight equals the log conditional probability: log P(wi | wi-1). The problem thus becomes equivalent to finding a longest path from start node to end node (boundary symbols we added) on a DAG graph. Since the graph is acyclic and layered, it's perfect for dynamic programming. Let f(i, cur) be the best sum up to pinyin s_i, where the last unigram "w_i#s_i" = cur. Keeping track of the last unigram makes sure that each pinyin s_i are used with the same tone in the two consecutive bigrams. The DP formula is therefore:

$$f(i, cur) = \max(f(i-1, prev) + log\_cond\_prob(prev, cur))$$

While doing DP, we keep a backtrack pointer of the previous character which gives the highest total sum so far. In the end we can trace back to get the most probable sequence of characters. The papers we read described this decoding approach as Viterbi algorithm on a Hidden Markov Model (Zheng, Sun & Li, 2011), but it's essentially the same algorithm. For the version without tones, the same algorithm applies, except that it will run much slower since we need to enumerate all tones to be used for each pinyin. This change effectively causes each layer in the graph has roughly 5 times as many nodes,

2.2.5 Smoothing:

In the first stage, we used Laplace (add-one) smoothing to deal with unseen words. In this method, we add one count to each original bigram counts such that no zero counts will ever occur. It's easy to implement but might distribute too much probability mass to unseen bigrams.

In the second stage, we implemented Good-Turing and Witten-Bell smoothing. For Good-Turing, we only discount for counts that's less than 5 to avoid gaps between higher counts. Witten-Bell smoothing estimates the unseen events based on the events with counts 1. We placed all three smoothing algorithms in smoothing.py, and passed a flag in the code to determine which one to apply.

4. Evaluation

We use extrinsic evaluation for this task in which we would run the entire program with test data and compare the result with our golden standard (correct Chinese characters for testing pinyin text). As mentioned before, we used disjoint subsets of LCMC for training and test sets. There is no need for development set because we don't need to tune any parameters. The gold standard for output is the real character sequences in the test bitext. The metric we used is the accuracy of the result, where *Accuracy = #correct_characters / #total_characters*. Calculating precision and recall doesn't apply well in this task. The common concept of recall would be "percentage of characters in the gold standard that the model got correctly". It fails to capture the order of outputs, which matters a lot in our context. For instance, we can get a 100% recall but 0% precision by shuffling a correct output, which impossible in a classification task.

5. Results

	Accuracies with pinyin tones (training, test)	Accuracies without pinyin tones (training, test)
Baseline (random guess)	0.314804537302, 0.329181669885	0.097634001309, 0.104692982456
Unigram (most frequent candidates)	0.747885305373, 0.715787310381	0.540888229291, 0.538609649123

Bigram with Laplace * (highest joint probability)	0.959594514501, 0.843993646199	N/A, 0.744558748076
Bigram with Witten-Bell *	0.981769502749, 0.853759831479	N/A, 0.737685992817
Bigram with Good-Turing *	0.971373486753, 0.848450725935	N/A, 0.759235505387

*Note bigram evaluations run much slower than the rest since the algorithm is polynomial time. For bigram with pinyin tones, we did the evaluation on the same test set but only on roughly 1/5 of the training data. Bigram without tones runs even slower (since the number of nodes increases), and we did the evaluation on much smaller test sets. N/A: takes too long to run (>20min) even on the smallest sub-training set we chose in LCMC.

The accuracy of baseline is about 1/3, which implies that in our training data, there are in average 3 characters homonyms per pinyin. In all n-gram models, the accuracy of training data result is higher than that of test data result because characters and bigrams in the training data has already been seen. Predictions for pinyin with tones in the input are much more accurate than without tones in all models. This is because the extra information carried by the tones greatly reduce the number of candidate characters for each pinyin. Unigram gives a fair accuracy compared to the baseline, because Chinese reuses some characters a lot, and the frequency distribution of the characters is greatly left-skewed, with a long tail of small counts. Bigram based algorithms are clearly superior to the rest, with around 85% test set accuracies and greater than 95% training set accuracies. We didn't anticipate the model to get such good results. Looking at the three smoothing methods, the differences of results are not very prominent. However, Witten-Bell gets slightly better accuracy than the other two and Laplace tends to be the least accurate (with tone in input pinyin string). The difference is small probably because the algorithm already reaches its bottleneck, where small probability mass differences in unseen bigrams will not make a lot of difference.

Finally, it's important to note that 1) We included some special symbols in the utterances such as Arabic numerals, which make the accuracy higher than it should be because these symbols' pinyin forms and characters have one-to-one match. 2) There are actually quite a few pinyin labeling errors in the LCMC dataset which cause the accuracy lower than it should be. For example, the character "像" is indeed pronounced as "xiang4" and is also decoded correctly in the output, but the gold standard marks it incorrect.

6. Discussions

We found some common mistakes and observations in the predictions by the bigram based algorithms. Firstly, a really large portion of the incorrect predictions are for named entities, such as "Hilary" and "Bai Su". Named entities impose a bottleneck to our algorithm, since they tend to be arbitrary combination of characters very unlikely to be seen earlier in the training set. We can imagine that, when building a real input method engine, one might need to feed a large set of named entities into the system separated from the training corpus. Furthermore, some Chinese pronounces have the same pronunciations, which confuses the algorithm. For instance, "他(he)", "她(she)", and "它(it)" all have the same pinyin "ta1". It's interesting to see that for subject, "he" is almost always chosen, and for object, "she" is always chosen. For problems like this, context information would be necessary for the system to make the correct choice.

From the above examples we can see that this algorithm we use is far from perfect. In a real Chinese IME, more complex machine learning based models might be used to improve the accuracy of predictions. For instance, Chen et. al. described a neural network based language model on top of n-grams. Furthermore, we learned that in machine learning literature this type of decoding task can be viewed as a sequence classification problem, which is a type of structured learning, and could be approached using the structured perceptron algorithm (Colin, 2002). Nevertheless, the bigram based algorithm we implemented is easy to implement, efficient, and accurate in most common cases.

Division of work

- Jingchen Hu
  - Database setup; data parsing
  - Unigram/bigram algorithms implementations (with tones)
  - Laplace and Witten-Bell smoothing
  - Web demo implementation
  - Report outline and corresponding parts
- Wenxi Lu
  - Good-Turing smoothing;
  - Evaluation; Accuracy calculation
  - Versions without tones
  - Bulk of the report

Bibliography

http://www.lancaster.ac.uk/fass/projects/corpus/LCMC/

http://www.zhtoolkit.com/posts/2012/10/lcmc-as-sql-database/

Chen, S., Wang, R., & Zhao, H. (2015). Neural Network Language Model for Chinese Pinyin Input Retrieved June 6, 2016, from http://www.aclweb.org/anthology/Y/Y15/Y15-1052.pdf

Collins, M. (2002). Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms. Retrieved from http://www.aclweb.org/anthology/W02-1001

Maeta, H., & Mori, S. (2016). Statistical Input Method based on a Phrase Class n-gram Model. Retrieved from http://www.ar.media.kyoto-u.ac.jp/publications/maeta-WTIM12.pdf

Zheng, Y., Sun, M., & Li, C. (2011). CHIME: An efficient error-tolerant Chinese Pinyin input method. IJCAI International Joint Conference on Artificial Intelligence, 2551-2556.