**Assignment 2 Report**

*Yue Dai,* [*yud42@pitt.edu*](mailto:yud42@pitt.edu)

1. **Vanilla N-gram Model (Without smoothing)**

The vanilla N-gram Model is implemented in file “**NgramVanilla.py**” and can be tested with “**testMain.py**”. The design simply counts the frequency of character groups in N length and get probability based on that without any smoothing modification. The detailed functions within the class included in README.

I tried 1-9 grams with this unsmoothed N-gram model, and evaluated the performance based on accuracy, which is the percentage of correct prediction across the “**dev**” data. The results are shown below:

|  |  |
| --- | --- |
| N-gram | Final Accuracy |
| 1 | 0.1564571199035951 |
| 2 | 0.22176675548268088 |
| 3 | 0.3667154505540456 |
| 4 | 0.5050138681459355 |
| 5 | 0.569810498802004 |
| 6 | 0.5666295884315906 |
| 7 | 0.5548988406456012 |
| 8 | 0.520910780669145 |
| 9 | 0.4795724465558195 |

*Table 1. Accuracy of Vanilla N-gram Model*

Based on the result we can see, that the unsmoothed N-gram model performed best on 5-gram, and after that, the accuracy started to drop. The reason is probably the smoothing became a problem, in which case that more and more unseen group appeared and without smoothing the model would be unable to deal with that.

1. **N-gram Model with smoothing**

To deal with the smoothing problem, I implemented a smoothed version of previous N-gram model in file “**NgramSmoothing.py**” and can be tested with “**testMain2.py**”. The smoothing method is Jelinek-Mercer smoothing (**Interpolation**), which combine N-gram model with N-1….1-gram models by getting the probability based on weighted sum of each probability, as below:

**…..**

In test I set , and tested from 6-9 grams, which were the numbers when smoothing became a problem, and the result tested with **“test”** data shown below:

Hyper parameter = 0.9

|  |  |  |
| --- | --- | --- |
| N-gram | Unsmoothed | Smoothed (Interpolation) |
| 6 | 0.5725848114344756 | 0.5717237816428449 |
| 7 | 0.5535714285714286 | 0.5579481792717087 |
| 8 | 0.5198504539789923 | 0.521630763752893 |
| 9 | 0.46224877783813145 | 0.4664131812420786 |

*Table 2. Comparison in accuracy of Unsmoothed/Smoothed N-gram Model*

Based on the result it can be observed that the smoothing method effectively increased the accuracy in higher N-grams.

1. **RNN**

I tried 3 RNN designs with TensorFlow based on GRU for the task.

* 1. One Hidden Layer between embedding and output, the architecture shown below:

Embedding

Output

Init

GRU

GRU

GRU

Hidden Layer

[BATCH\_SIZE = 64, Units = 512]

…..

Embedding Layer

[BATCH\_SIZE = 64, SEQ\_LENGTH = 100, Embedding size = 128]

There is 1 hidden layer with 512 GRUs in it, the embedding size is 128 and sequence length is 100. The input to each GRU is result from previous one and correct last character.

The RNN was trained 5 EPOCHs, each EPOCH cost about 1600 sec and got **﻿0.491770496846639 accuracy** in the end. The file for train this model is “**RNN.py**” and can run “**RNN\_test.py**” to load pre-trained weights in “﻿./training\_checkpoints” to test.

* 1. Two Hidden Layer between embedding and output, the architecture shown below:

Output

Hidden Layer 2

[BATCH\_SIZE = 64, Units = 256]

GRU

GRU

GRU

Init

Embedding

Init

GRU

GRU

GRU

….

Hidden Layer 1

[BATCH\_SIZE = 64, Units = 256]

…..

Embedding Layer

[BATCH\_SIZE = 64, SEQ\_LENGTH = 100, Embedding size = 128]

There are 2 hidden layers with 256 GRUs in each, the embedding size is 128 and sequence length is 100. The input of layer 1 GRUs are as before, the input of layer 2 GRUs are outputs from layer 1 and previous units.

The RNN was trained 30 EPOCHs, each EPOCH cost about 1600 sec and got **﻿﻿0.48854022458083374 accuracy** in the end. The file for train this model is “**RNN\_Mod.py**” and can run “**RNN\_Mod\_test.py**” to load pre-trained weights in “﻿./training\_checkpoints\_mod” to test. The reason that the accuracy didn’t increase might because each layer squeezed even though there were 2 layers.

* 1. Another try with larger hidden layer

Same structure as 3.2, 2 hidden layers yet 512 embedding dimensions with 1028 units per layer

1. **Comparison**
2. **Chinese Predictor**

In Chinese predictor I just implemented N-gram models, both smoothed and unsmoothed. The classes are contained in **“N﻿gramChinese.py”** and **“﻿NgramChineseSmoothing.py”**, the predictor can be executed with **“ChinesePred.py”**

**Steps for N-gram:**

* Count groups in length N within train data to get frequency for the groups
* Create dictionary contains (pinyin: list Of Chinese) pairs based on “./chinese/charmap” document
* Each time when find a pinyin as input from “test.pin”, first get the candidate list by accessing dictionary above, get possible Chinese characters
* Got context from “test.han”, then concatenate with candidates to get candidate groups
* Based on Frequency got from train data set, calculate probability (either smoothed or not) for each candidate groups
* The smoothed version also used interpolation as English one with λ = 0.9
* Pick the highest probable group, then return the last character as predicted one

Since there’s no enough time training RNN for it, the way to do it is creating a new mapping between Chinese and character ids, then mapping probabilities for candidate list elements and get the highest probable character id the get back.

1. **Chinese Predictor Results**

The results shown below (N from 2-4).

|  |  |  |
| --- | --- | --- |
| N | Unsmoothed | Smoothed |
| 2 | 0.8310679611650486 | 0.7915857605177994 |
| 3 | 0.6211139896373057 | 0.8115284974093264 |
| 4 | 0.47893713545042127 | 0.813350615683733 |

*Table 3. Comparison in accuracy of Unsmoothed/Smoothed N-gram Model*

According to results, when N=3 smoothing became a problem, with smoothing modification N-gram performed better in 3 and 4 grams. However, bi-gram without smoothing performed best.