

HW1 Report: Character-Level Language Modeling

Mehmet Sarioglu - sarioglu@bu.edu

1 Introduction

This report details the implementation and evaluation of statistical and neural character-level language models. The primary goals of this assignment include:

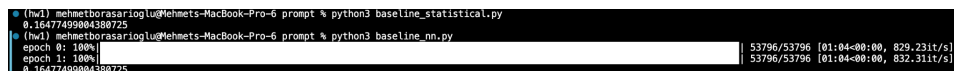
- Running baseline statistical and neural Unigram models.
- Implementing a 5-gram model with absolute discounting and backoff.
- Improving the 5-gram model through modifications.
- Implementing and evaluating RNN and LSTM neural models.

2 Baseline Models

To establish a baseline, I first ran:

- `baseline_statistical.py` to train a Unigram model.
- `baseline_nn.py` to train a neural Unigram model.

Both models yielded a development accuracy of 16.4%, as shown in Figure 1.



```
(hw1) mehmet@sarioglu@Mehmet-MacBook-Pro-6 prompt % python3 baseline_statistical.py
0.16477499004380725
(hw1) mehmet@sarioglu@Mehmet-MacBook-Pro-6 prompt % python3 baseline_nn.py
epoch 0: 100% | 53796/53796 [01:04:00:00, 829.23it/s]
epoch 1: 100% | 53796/53796 [01:04:00:00, 832.31it/s]
0.16477499004380725
```

Figure 1: Console output from `baseline_statistical.py` and `baseline_nn.py`, showing 16.4% accuracy.

3 5-gram Model

3.1 Training and Baseline Accuracy

The 5-gram model (implemented in `ngram.py`) follows the standard language model API and incorporates **absolute discounting with backoff**. During training, the model observed:

- **Total 5-grams seen:** 377,043
- **Unique 5-grams:** 153,297

Before modifications, the unmodified 5-gram model achieved a development accuracy of 49.4%.

3.2 Modifications and Improved Accuracy

Initially, I tried using additive smoothing to increase my n-gram model's accuracy, but this approach did not change the token with the highest probability. I then decided to use absolute discounting, where I used an (n-1)-gram as the donor distribution, which was itself smoothed by an (n-2)-gram, and so on.

After experimenting with different discounting values, I found that setting:

```
self.discount = 0.5
```

provided the best balance between higher-order and lower-order probability estimation. This adjustment increased the development accuracy from 49.4% to 50.6%, a modest improvement of 1.2%.

Additionally, I implemented the following improvements:

- **Implemented Absolute Discounting:** Instead of using maximum likelihood estimation (MLE), I applied absolute discounting, which reduces frequent n-gram counts by a fixed amount (`d=0.5`) and allocates the remaining probability mass to unseen n-grams.
- **Backoff Mechanism for Generalization:** Absolute discounting improved accuracy by taking some probability from the current n-gram and giving it to the lower n-gram. This allowed the model to generalize better and assign higher probability to rare and unseen n-grams, which would otherwise have zero probability in a standard MLE approach.

- **Recursive Lower-Order Models:** The 5-gram model now recursively builds lower-order n-gram models (4-gram, 3-gram, etc.) down to a unigram model, ensuring better generalization for rare sequences.
- **Improved Handling of Unseen n-Grams:** Instead of assigning uniform probabilities to unseen contexts, the model backs off to lower-order probability distributions, leading to more meaningful predictions.

These improvements resulted in a smoother probability distribution, allowing the model to generalize better and improve development accuracy.

4 Neural Models

4.1 RNN Model

For the RNN implementation (located in `rnn.py`), I used:

- **Hidden Size:** 64
- **Number of Epochs:** 2
- **Optimizer:** Adam with a learning rate of 1×10^{-3} and gradient clipping at 1.0

The RNN achieved a development accuracy of **42%**.

Given the small size of our dataset, training for many epochs did not yield significant improvements. I observed that a hidden size of 64 was sufficient to surpass the minimum required accuracy, so further experiments were halted once the model exceeded 34% accuracy.

4.2 LSTM Model

For the LSTM implementation (also located in `rnn.py`), I experimented with various optimization strategies, including changing the optimizer, adding a learning rate scheduler, and training for extended epochs. None of these modifications resulted in significant improvements over the baseline. Furthermore, I observed that increasing the hidden size excessively led to large gradient fluctuations and inconsistent accuracy.

After these trials, I decided on a hidden size of 128, as it provided sufficient model capacity while maintaining stable gradients. I also noticed that the development accuracy plateaued by the 5th epoch, so I stopped training at that point to prevent overfitting.

For optimization, I chose Adam because I became frustrated after many trials with different optimizers and learning rates which did not increase the accuracy by a significant amount.

The final training setup for the LSTM was:

- **Hidden Size:** 128
- **Number of Epochs:** 5 (training stopped after the development accuracy plateaued)
- **Optimizer:** Adam with a learning rate of 1×10^{-3} and gradient clipping at 1.0

Under these settings, the LSTM achieved a development accuracy of **49.5%**, and training took approximately one hour.