# Project1: Smoothing Algorithms of n-gram Language Model

## 1. Construction of n-gram Language Model

In order to thoroughly compare the ability of different language models, I constructed **4 models** with different grams in this project, including **unigram, bigram, trigram and 4-gram**.

n-gram language model is **statistic model**, which means that the training procedure only includes counting the words or phrases in the corpus.

Take the construction of unigram model as an example (bigram, trigram and 4-gram models are constructed similarly). Before counting the amount of each word, we should begin with **listing all of the different words** (also called "type") that exist in the whole corpus, which is the union of train_set, dev_set and test_set. Next, the number of each type in the unigram model is **initialized as** $0$:

```
for word in corpus:
    model.unigram[word] = 0
```

Then, we traverse through the train_set and **count the number of each word**, preparing for the following discounting and testing procedure:

```
for word in train_set:
    model.unigram[word] += 1
```

After counting each word in the train_set, we notice that some of the words do not exist in train_set, making the count of them 0. This problem is also called **sparsity problem**, which is a serious problem for the following reason.

During the test procedure, the perplexity is computed by the following formula:

$$PPL = 10^{-\frac{1}{K}\sum_{k=1}^{K} log_{10} P(\omega_k | W_{k-n+1}^{k-1})}$$

Obviously, if $P = 0$, the PPL will become an **infinite number**, which is not acceptable.

Therefore, some algorithms have to be designed to solve sparsity problem. There are two kinds of solutions:

1. **Discounting**: To keep a language model from assigning zero probability to these unseen events, we shave off a bit of probability mass from some more frequent events and give it to the events we've never seen. The algorithm can be denoted as:

   $$P_{discount}(\omega|y) = d(y,\omega)\frac{C(y,\omega)}{C(y)}$$

   $d(y,\omega)$ is called "discounting coefficient".

2. **Back-off & Interpolation:** Recursively move back to lower-order n-grams, until we get a robust estimation.

In this project, $2$ discounting algorithms are designed and implemented, which is **Add-k Smoothing** and **Good-Turing Smoothing**, respectively. The detailed design and implementation are shown in the next section.

# 2. Design and Implementation of Discounting Algorithms

In this section, the Add-k Smoothing and Good-Turing Smoothing is discussed in detail.

## 2.1 Add-k Smoothing

- Basic Idea

  Since there are several unseen cases, the **intuitive way** to do smoothing is to add $1$ to all of the n-gram counts before we normalize them into probabilities. This method is also called Laplace smoothing.

- Algorithm Design

  Suppose there are $V$ words in the vocabulary (the number of different types in the corpus). We also need to adjust the denominator to take the extra $V$ observations into account:

  $$P_{Laplace}(\omega_i) = \frac{c_i + 1}{N + V}$$

  Furthermore, in the practical situation, adding $1$ to every word may be too large. Hence, we may add a fractional count $k$ instead of $1$ to each word:

  $$P_{Laplace}(\omega_i) = \frac{c_i + k}{N + kV}$$

  This is why the method is called add-k smoothing.

- Implementation

  The implementation can be denoted as the following code:

```
model.unigram = {
    word: (count+k)/(N+k*V) for word, count in model.unigram.items()}
```

## 2.2 Good-Turing Smoothing

- Basic Idea

  **Redistribute** the probability of n-grams that appear for $r + 1$ times to those that appear for $r$ times.

- Algorithm Design

  Suppose $N_r$ is the "**counts of counts**", which is the amount of n-grams that appear for $r$ times in the train_set. Hence, the amount of observed n-grams in the train_set is

  $$N = \sum_{r=0}^{\infty} rN_r$$

  Next, the new $r^*$ can be denoted as

  $$r^* = (r + 1)\frac{N_{r+1}}{N_r}$$

  Therefore, the new probability of an n-gram can be defined as:

$$P^* = \frac{r^*}{N}$$

- Implementation

  Based on the discussion above, we can implement the algorithm by the following code:

```
for k in model.unigram.keys():
        r_star = model.unigram[k] + 1
        if r_star in Nr.keys():
            model.unigram[k] = r_star * Nr[r_star] / Nr[r_star - 1]
```

# 3. Experiments & PPL Results

In this section, the effect of Add-k Smoothing and Good-Turing Smoothing is tested respectively using 4 models.

For the Add-k Smoothing, experiments using different values of $k$ ranging from $10^{-20}$ to $10$ is performed on the given dataset, while only one experiment is performed using Good-Turing Smoothing. The n-gram models are trained on *train_set.txt* and tested on *test_set.txt*. Moreover, Good-Turing Smoothing is only done on the cases with $r \leq 50$.

Notice that **the whole test_set is regarded as a single sentence** in the experiments, thus only one ending token is added to the end of the test_set.

The perplexity results are shown in the following table:

| Model\Method | Good-Turing | Add-k (k=10) | Add-k (k=1) | Add-k (k=0.01) | Add-k (k=$10^{-5}$) | Add-k (k=$10^{-20}$) |
|---|---|---|---|---|---|---|
| **unigram** | **1864.24** | 2037.71 | 1892.49 | 1986.76 | 2161.53 | 3295.87 |
| **bigram** | **130.77** | 373872.59 | 83899.55 | 5104.26 | 2071.04 | 1586870.85 |
| **trigram** | **7.72** | 4768431.84 | 2341297.87 | 353648.89 | 59004.89 | 3971517736.62 |
| **4-gram** | **1.66** | 10646985.95 | 7815486.22 | 3188502.67 | 1024714.42 | 1041193555.62 |

# 4. Conclusion

Based on the results, we can come to the following conclusions:

1. Generally speaking, **models with larger grams performs better than those with lower gram**s (suppose that you have chosen the appropriate discounting method). The reason is that models with larger grams takes more context into account, thus being more robust.

   An interesting idea: This conclusion is quite similar to the fact that **LSTMs usually outperforms Vanilla RNNs** due to its capability of considering more context!

2. When it comes to **robustness**, Good-Turing Smoothing largely outperforms Add-k Smoothing. As we can see from the table above, if inappropriate $k$ is chosen, Add-k smoothing algorithm may result in extremely large perplexity, especially for models with larger grams.

   Therefore, Good-Turing Smoothing may be a better choice in most situations.

# 5. Reference


1. Slides of SJTU-CS382
2. N-Gram Language Models | Towards Data Science
3. Additive smoothing - Wikipedia

4. n-gram - Wikipedia
5. cs224n-lecture2-language-models.ppt (stanford.edu)
6. Good–Turing frequency estimation - Wikipedia
7. good-turing-smoothing-without.pdf (ntu.edu.tw)

# 6. Supplementary

1. To **execute the code**, you may simply run the code below in the project folder:

```
cd tools
# Train the model and save model parameters to trained_model folder
python ../main.py --discounting AddAlpha --alpha 1e-5 --save 1 --train 1
# Add-k Smoothing
python ../main.py --discounting Good-Turing --save 1 --train 1 # Good-Turing Smoothing
# Load the trained model and only test perplexity
python ../main.py
```

   Packages you may need to install by simply executing `pip install package-name`:

```
numpy
tqdm
pprint
argparse
```

2. **I guarantee that everything of the code and report is done by myself without using any complex open-source tool other than basic packages such as numpy**. If you have any question for any section of this project, please feel free to contact me by the following methods:

   E-mail: douyiming@sjtu.edu.cn

   Wechat: 18017112986