11-411/611 Natural Language Processing

Assignment 3: Language Modeling

Due on October 1, 2020, 11:59 PM EDT

Background

Language modeling plays a key role in many successful natural language processing applications, including but not limited to, machine translation, automatic speech recognition, handwriting recognition, spelling correction, etc. Many of the cool applications you use on a daily basis actually have a language modeling component working at the backend. For example, Amazon Alexa, Apple Siri, Google Translate, Nuance's Dragon Natural Speaking, etc.

Tasks

Task 1: Building Language Model (70 points)

In this assignment, you will be building your own language model (yay!). In particular, we offer you an exciting new field which has been rarely explored in the past –privacy policies of the online websites. Armed with the language modeling techniques learned from class, would you be able to model the "special language" used by the lawyers when drafting these privacy policies?

We provide you with a collection of privacy policies, each corresponds to a popular online website. The policies are grouped into 5 categories according to the functionality of the websites. The categories and representative websites are shown below:

- News - e.g., www.cnn.com, www.nbcnews.com, www.nytimes.com, etc. (news.txt)
- Shopping - e.g., www.amazon.com, www.ebay.com, bestbuy.com, etc. (shopping.txt)
- Games - e.g., blizzard.com, www.nintendo.com, etc. (games.txt)
- Sports - e.g., espn.go.com, sports.yahoo.com, etc. (sports.txt)
- Health - e.g., www.nih.gov, www.mayoclinic.org, etc. (health.txt)

Each `*.txt` file contains 50 lines, each line represents one privacy policy. We have preprocessed the text so that the file contains only white-space-separated tokens. Your task is to build and test language models using the provided `.txt` files.

More specifically, you will write a program to implement a linearly interpolated language model with N up to 3 (see Section 4.6 in the S&LP textbook). Your model should interpolate the N-gram probabilities of the following models:

- a trigram model
- a bigram model

- a unigram model
- a uniform distribution over words - a model where all words get probability $1/V$, where V is the size of the vocabulary (number of distinct types, including punctuation and the `UNK` symbol).

Your program must take the following command-line arguments in the following order:

1. `coef_unif` - coefficient for the uniform model
2. `coef_uni` - coefficient for the unigram model
3. `coef_bi` - coefficient for the bigram model
4. `coef_tri` - coefficient for the trigram model
5. `min_freq` - minimum frequency threshold for substitute with `UNK` token (strictly less than). Set to value of 1 for no substitution.
6. `testfile` - file to calculate perplexity on
7. `trainingfile` - file(s) to train the four language models

An example command is provided below.

```
$ python lm.py 0.25 0.25 0.25 0.25 5 test.txt train.txt
```

This program should then print the perplexity of `test.txt` under a language model trained on `train.txt`, with all coefficients for the four language models set to 0.25, with words appearing less than 5 times substituted with the `UNK` token.

Details:

- Make sure you have completed the required reading, chapters 4.3 - 4.8 in the textbook.
- Skeleton code has been provided. The autograder will instantiate your LanguageModel class and invoke the `most_common_words` and `calculate_perplexity` functions, so be sure to implement them and read their specifications carefully (such as sorting order).
- You will be graded in part based on whether the perplexity scores of your uniform, unigram, bigram, trigram, and interpolated models meet reference thresholds on hidden training/test files.
- When combining several language models, calculate log probabilities of each model first, then combine them with corresponding coefficients.
- Start with simpler models first, such as Bigram with Laplace smoothing.
- Use the training data to calculate the vocabulary size V.

- Before training your model, it is often good to identify some infrequent words in the training file (say they appear less than 5 times in the training file) and replace them with a hand- picked symbol (e.g. `UNK`). This will often help you greatly reduce the vocabulary size. Similarly, during testing, you will replace any word that does not appear in the vocabulary or occurs fewer than `min_freq` times with the same symbol (e.g., `UNK`). Then you may proceed with the training/test as usual.
- For the purpose of this assignment, it is not necessary to add start and end tokens, but it is good practice in general and most real-world NLP applications expect it.
- You can use the bare-bones tokenizer provider in `utils.py`, and don't have to further worry about punctuation, etc.
- Optional techniques that may boost your score include backoff, Laplace Smoothing, and Good-Turing Discounting. See the textbook details.

Notes:

- Make sure your program name is `lm.py`, you only need to modify this file.
- Your `handin.zip` should consist of `lm.py` and `utils.py` (don't include the five `.txt` files given to you)

Task 2: Analysis (Submit this part in GradeScope as a PDF) (30 points)

Subtask 1 (15 points)

Having completed Task 1, it is often important to do so in order to answer the question: "Does my model make sense?"

Using the `sports.txt` file, train the unigram/bigram/trigram models respectively using this file. Examine the unigrams, bigrams, and trigrams with the highest probabilities. Answer the following questions:

1. How many unique N-gram types are there in the unigram, bigram, and trigram models, respectively?
2. What are the most common N-grams for each model? Would you expect them to appear frequently in the privacy policies of that category?
3. What are a few interesting N-grams that are related to this specific category?
4. Do you have any other interesting observations?

By interesting examples we mean any informative ngrams or ones that stand out as being surprising from the big pool of words that you generate.

Subtask 2 (15 points)

Providing different weights to each of the language models would result in changes to the perplexity values. We would like to find weights that provide us a better result. This is an example of hyperparameter tuning. In practice, one would automate the search for optimal hyperparameters, rather than searching for it manually. In this subtask, we want you to manually find better weights for the language models.

Using your model from *Subtask 1* and `games.txt`, manually tune the values of the lambda coefficients and test its effect on the perplexity score. Display your findings in a tabular form or a chart if necessary. Report the best perplexity you were able to obtain for the test file and what lambda values were used (remember: lower perplexity numbers are better!). Also clearly explain the approach/process you took while changing these parameters, and explain why you think it improved/did not improve perplexity.

Please note we are not asking you to find the best possible lambdas that would result in the lowest perplexity score. You will not be graded on the magnitude of improvement you managed to obtain, but it should be a positive value.