Assignment 2 Report
CSE 143: Intro to Natural Language Processing

University of California Santa Cruz
Tommaso Framba, tframba@ucsc.edu, ID: 1815342
Thursday May 5, 2022

This assignment was done in Python 3 without use of any NLP libraries (such as NLTK).

Abstract

In this second assignment I implemented N-gram language models such as unigrams, bigrams, and trigrams with optional additive smoothing or linear interpolation smoothing.

1 Programming: n-gram lanugage modeling

Model Description In this first problem, I implemented unigram, bigram, and trigram language models to build and evaluate perplexity in the data as discussed in the sections and lectures. To handle out of vocabulary words the tokens that occurred less than three times were converted to "UNK". First the model grabs unigrams from the given data file by tokenizing sentences and adding a start and end token to the start and end of each sentence. Then the model computes a maximum likelihood estimate for unigrams and stores it into a probability dictionary. If calculating perplexity for bigrams and trigrams the model similarly creates the tokens from the unigram data in each sentence and applies an appropriate maximum likelihood calculation in order to calculate perplexity. Perplexity was calculated by going through each sentence in the given data and breaking it up into tokens. Bigrams and Trigrams were created by creating combinations of the current element in a loop and the next two or three elements and adding each combination into a list. The probabilities of mle were then added together by their math.log value. The math.exp value was then calculated by the sum of mle probabilities over the sentence length to provide the perplexity.

Maximum Likelihood Estimate for models was calculated as:
Unigrams: $P(w_i) = c(w_i)/\Sigma_w c(w)$
Bigrams: $P(w_i, w_{i-1}) = c(w_{i-1}, w_i)/\Sigma_w c(w_{i-1}, w)$
Trigrams: $P(w_i, w_{i-1}, w_{i-2}) = c(w_{i-2}, w_{i-1}, w_i)/\Sigma_w c(w_{i-2}, w_{i-1}, w)$

Perplexity for models was calculated as:
Unigrams: $Per(unigrams) = \Sigma_s unigramMLE(s)/sentenceCount$
Bigrams: $Per(bigrams) = \Sigma_s bigramMLE(s)/sentenceCount$
Trigrams: $Per(trigrams) = \Sigma_s trigramMLE(s)/sentenceCount$

Performance Overall the performance for all three models is decent and the execution of six perplexity requests and all data queries for train, test, and dev files runs in under six seconds. I elected to utilize dictionaries because in python dictionary lookup takes amortized constant time versus linear time for lists. As well the perplexity calculations and data queries from files takes $O(n^2)$ runtime.

Experimental Procedure In order to implement all three natural language processing models and perplexity calculation I began by extracting the features from the given data. I ensured that the amount of unique tokens with removed words replaced by "UNK" for unigrams was equal to 22602. The unique token count for bigrams and trigrams came out to 510391 and 1116131 respectively. Happy with those results I then created an experiment file with the data "HDTV ." and confirmed perplexity results that equaled "658.0445066285453", "63.707573620519", and "39.47865107091443" for unigram, bigram, and trigram models. With these results I knew that I met the given specs for the models and began to work on providing deliverable data.

Deliverables

Perplexity of Unigrams:
Train: 976.5437422200694
Dev: 892.2466475122606
Test: 896.499491434349

Perplexity of Bigrams:
Train: 77.07346595629372
Dev: 28.290360699463918
Test: 28.312808836092298

Perplexity of Trigrams:
Train: 7.872967947054013
Dev: 2.9944740517539046
Test: 2.9948349871756643

Experimental Results: From these results one can clearly see that the perplexity drastically decreases when increasing previously seen word count in the n gram feature. The trigram feature indicated the best generalization performance out of the three and resulted in an average 89.5% drop in perplexity across all three pieces of data when compared to the bigram feature and a whopping 99.51% average drop in perplexity when compared to the unigram feature. The bigram feature also did well and reported a 95.26% average drop in perplexity when compared to the unigram feature. This showcases that n-grams are much stronger generalization devices than single unigram features when classifying text in NLP and increasing the amount of words coupled into a n-gram feature drastically reduces the overall perplexity of the model.

2 Programming: additive smoothing

Model Description In this second problem the models were the exact same as the first problem and only differed in how the maximum likelihood was calculated. The Perplexity and data gathering attributes of all three n-gram models remained exactly the same.

Maximum Likelihood Estimate for models was calculated as:
Unigrams: $P(w_i) = (c(w_i) + 1) * (N/(N + V))$
Bigrams: $P(w_i) = (c(w_{i-1} * w_i) + 1)/(c(w_{i-1} + V))$
Trigrams: $P(w_i) = (c(w_{i-2}^{i-1} * w_i) + 1)/(c(w_{i-2}^{i-1} + V)$
Where V is the total number of unique tokens and N is the number of specific tokens.

Performance Overall the performance of the models with additive smoothing remains the same as before as the only computation that changed was the MLE. The remaining perplexity computations remained the same with the exception of calling the appropriate additive smoothing methods. In the end the model computes in $O(n^2)$ runtime.

Experimental Procedure The experimental procedure was also similar to the previous model and deliverable assessments. I began by researching lecture and section notes and deriving the additive smoothing maximum likelihood calculations then verifying if the perplexity numbers added up to provide optimal results.

Deliverables

1. With alpha = 1

Perplexity of unigram: Train(972.6323766506046), Dev(890.2700616149332)
Perplexity of bigram: Train(66.7176482165485), Dev(27.25610618012612)
Perplexity of Trigram: Train(6.540010930482069), Dev(2.934982551214741).

2. With alpha = 0.75

Perplexity of Unigram: Train(734.818422162674), Dev(672.4577402964736)
Perplexity of Bigram: Train(66.7176482165485), Dev(27.25610618012612)
Perplexity of Trigram: Train(5.813880752785232), Dev(2.726699169051844)

With alpha = 0.70

Perplexity of Unigram: Train(687.052983466531), Dev(628.7257928698918)
Perplexity of Bigram: Train(66.7176482165485), Dev(27.25610618012612)
Perplexity of Trigram: Train(5.6492304115434875), Dev(2.6789887203789444).

3. With test set alpha = 0.70

Perplexity of unigram: 631.650373805156
Perplexity of bigram: 27.28230707467201
Perplexity of trigram: 2.678693307114576

Results: From these results one can conclude that additive smoothing is decently useful re-purposing the counts and creating new generated data that the training model can see. Larger values are penalized and values that were previously smaller gain a small bump in order to balance out the overall counts of the unigram and n-gram features. In the end the alpha value of 0.7 led to the best results and provided a strong perplexity.

3 Programming: smoothing with linear interpolation

Model Description For this third problem the calculations remained the same for MLE unigram, bigram, and trigram models with the exception of new calculations for perplexity of trigrams. The new trigram perplexity used the given linear interpolation smoothing equation of three lambdas as follows:

Linear Interpolation smoothing:

$$\theta'_{x_j|x_{j-2,j-1}} = \lambda_1\theta_{x_j} + \lambda_2\theta_{x_j|x_{j-1}} + \lambda_3\theta_{x_j|x_{j-2},x_{j-1}}$$

Inside of the linear interpolation method the weights were applied to the old trigram perplexity method with respect to each part of the unigram, bigram, and trigram portions.

Performance The performance of the linear interpolation smoothing aspect of the updated model remained exactly the same as the calculations inside the inner for loop were the only piece that was updated from the original perplexity calculation. This resulted in a previously seen $O(n^2)$ runtime.

Experimental Procedure As given specifications for the debug model were included for this problem I elected to start by deriving the given linear interpolation model into my trigram perplexity method and began working to match the given 48.1 perplexity value with debug data for lambdas equal to 0.1, 0.3, and 0.6. Once I had hit the given specs with a perplexity value of: 48.11351783080278 I began to work on deliverables. For analyzing half of the data I used the linux $split$ command on the training data and ran my program again to record results.

Deliverables

1. Results for three different sets of $\lambda's$:

$\lambda_1 : 0.1, \lambda_2 : 0.2, \lambda_3 : 0.3$
Train: 21.218894800819708
Dev: 8.332137412673413

$\lambda_1 : 0.3, \lambda_2 : 0.4, \lambda_3 : 0.5$
Train: 12.195756319181998
Dev: 4.822638133719282

$\lambda_1 : 0.5, \lambda_2 : 0.6, \lambda_3 : 0.7$
Train: 8.56180867914766
Dev: 3.3956203686128967

Required Specs: $\lambda_1 : 0.1, \lambda_2 : 0.3, \lambda_3 : 0.6$
Train: 11.151493208689091
Dev: 4.343292286827112

2. Perplexity on test set with best set of hyperparameters from development set:

$\lambda_1 : 0.4, \lambda_2 : 0.5, \lambda_3 : 0.6$
Test: 3.9883099177167036

3. Increase or decrease perplexity using half of the training data:

Evidence:
Model Vocabulary Unique Tokens: 17537
Train Unigram Perplexity: 816.4896195723179

Train Bigram Perplexity: 63.08454352221167
Train Trigram Perplexity: 6.5508791756376095

Train Linear Smoothing Perplexity using $\lambda_1 : 0.1, \lambda_2 : 0.2, \lambda_3 : 0.3$: 17.720619475123197

Explanation: As the evidence showcases the perplexity decreases for train data with and without smoothing when using half of the training data. Perplexity is a measure of how well the models can generalize sentences in given data. The evidence showcases that increase in data creates a direct effect on the model failing to generalize more accurately. In all four cases of evidence the perplexity when using half of the training data dropped an average 17.11% when compared to the original values. As previously stated this makes sense as perplexity is a measure of generalization and too little data can cause increase in n-gram model perplexity as it fails to generalize for too little data and a large increase in data also increases perplexity because the model struggles to generalize for too large amounts of data.

4. Convert all tokens appeared less than 5 times to "unk". Increase or decrease perplexity when compared to an approach that converts only a fraction of words that appeared just once to "unk"?

Converting less than 5 times to "unk":
Train Unigram Perplexity: 1128.3632479535158
Dev Unigram Perplexity: 1001.1737807925177
Test Unigram Perplexity: 1003.6887059145532

Explanation: Perplexity increases in the case of converting words that occur less than 5 times to "unk" when compared to converting a fraction of words because there are more words that occur less than five times than words that occur once and converting a fraction. This in turn increases because now too many words have been converted to "unk" and the models have a harder time trying to generalize. As the evidence showcases each of the models perplexity increases by an average of 12.1% and thus the model has a poor time generalizing as too many words have been converted to the unknown token.