CS 4740 Project 1 Part 1 report

Rui Liang, Gi Yoon Han, Xiaoyun Quan

September 11, 2016

1 Introduction

In the first part of Project 1, we investigated into the mechanism of generating random sentences from datasets. Specifically, we started with preprocessing the dataset to keep only relevant information in our corpus, based on which we build up our unigram and bigram models to generate random sentences.

1.1 Instruction to run the code:

For section 4 (smoothing), section 5 (perplexity), and section 8 (interpolation), run script `smoothing_perplexity.py`. When prompted, please enter the name of the target folder to run, such as `autos` or `graphics`; then enter the λ value (from 0 to 1) that would be assigned to bigram probability. The result will be saved as `CSV` files in folder `smoothing_perplexity`. If folder doesn???t exist, please create one before running the script.

For section 6 (classification), run script `classification_training.py` first. When prompted, please enter the name of the target folder to run, such as autos or graphics. Run the script for ALL 7 categories. The script will create training data results and save as CSV files in folder `classification_training_results`. If folder doesn???t exist, please create one before running the script. Then, run script `classification_test.py`, and the result will be saved in a CSV file called `prediction_result.csv`.

2 Unsmoothed n-grams

To get rid of all irrelevant information in the training dataset, we first processed the data following such steps:

- remove the sender's email addresses in each single file under the topic folder by deleting any segement of sentence starting with "From" and ending up with a space follwing an email address
- remove the first "Subject :" in each single file
- remove all linebreakers " >"

By doing so, we clean up our data so that irrelevant info such as line breakers in the text files will not take up tokens. Note that we did not remove the contents of subject because they might be relevant to the topic. Also, the email addresses other than sender's ones are kept to retain information integrity. The next thing to do is merge all files to obtain the corpus and tokenize the corpus. Specifically, the merged file is scanned for sentence ending indicators including "." "!" "?" or multiples of them, and at the beginning and ending of each sentence we add "<s>" and "</s>" respectively, which is stored as `corpus_sentence`. However, we did not consider "<s>" and "</s>" for tokenization, where results were stored in `corpus_word`.

Now that the corpus is built up, we are able to compute our unigram and bigram probablities. The probabilities are stored in two dictionary named `unigram_probabilities` and `bigram_probabilities` corresponding to unigram and bigram models respectively.

3 Random sentence generation

With our unigram and bigram language models ready, we are now able to develop a random sentence generator. The results are as expected: some are comprehensible and some are not. For example, IN the 'religion' topic corpus, a sentence " So you do wrong with gunfire . " was produced with bigram model and "Lafayette H well perhaps _____ Rosicrucians ." was generated from unigram model. While the bigram model is giving a more comprehensible sentence because it considers the preceeding word, the unigram model is less reliable by its nature. Also note that even though symbols such as "_" make the sentence even more confusing, we insisted on keeping all special symbols because they might be meaningful in certain context such as 'graphics' topic.

Another noteworthy example is that in the 'atheism' model, we obtained a random sentence "." which consists of only one period and one space. This is not surprising since most periods "." are followed by a space, and in our case since we randomly selected "." as the first word, it is hence expected to give such a sentence. A similar sentence is generated in 'space' topic under unigram model : "moon." Incomplete sentence fragments can also be generated. For example, we get " On the Reagan administration . " in the 'space' topic. It is not surprising if we consider the construction of bigram model: the probability $P(\texttt{"."}|\texttt{"administration"})$ is found to be as high as 0.5 which explains why this sentence is very likely to happen.

We can have a better comparison between unigram and bigram models by looking at the results from seeding. For example in the 'atheism' topic, we set the beginning of sentences as 'I', and what we got from unigram model is 'I keith my strong] the the ?', compared to the one from bigram model ' I guarantee that was the theists be aware of the motto to defend , what that it must be a fallacy . '. Apparently the bigram result makes more sense because in the corpus 'I' is very likely to be followed by 'guarantee' and likewise 'guarantee' will be followed by 'that' with a high probability. However in unigram model, the preceeding word 'I' is not making a difference, so it will continue to generate succeeding words independently and that's why the unigram sentence is less readable.

4 Smoothing and unknown words

Now let's consider testing our training dataset on some test files. The first question arose here is that what if we encounter some new words in the test file that are not in corpus? One way to solve this is by including unknown words in our corpus. By doing so, the probability of new words in test file will be nonzero. We learned two ways to realize this in class: one is to replace the first occurrence of each word with "<unk>", and the other is by replacing all words that only appear once in the corpus with "<unk>". We believe the latter will outperform for the reason that the gap between word types and corpus length is not that wide for all topics otherwise we would get a fairly large amount of pseudo-words which is not desired (see table below). For example, under 'Autos' topic, we have 9555 word types and 74288 tokens, and the first method will make $\frac{1}{8}$ of the corpus to be "<unk>", which will mislead the unigram and bigram probabilities.

	Autos	Atheism	Graphics	Medicine	Motorcycles	Religion	Space
Token	9555	10917	11583	12178	9537	11866	12427
Word Type	74288	120085	89140	93990	69167	117811	96024

Now that we have set up an open vocabulary, it is likely that this new corpus still has many zero bigram probabilities which can be problematic for training on test set. The strategy here will be smoothing for our language models. In fact, Good-Turing is implemented for both unigram and bigram models to get a smoothed count for each word using this formula:

$$c^* = (c+1)\frac{N_{c+1}}{N_c}$$

where c is the original count and $N_c = \sum_{\text{x: count(x)}=c} 1$. However it's noteworthy that for the unigram model,

N_0 is 0 by unigram nature, and N_1 is 0 by incorporating unknown words, so that we only did Good-Turing smoothing for $c = 2, 3, 4$. On the other hand, smoothing for $c = 1$ is not necessary since incorporating "<unk>"'s already smooths $c = 1$ down.

5 Perplexity

In this section, we would like to find perplexities for each test file under all 7 topics for each model. Before starting, we realized that the formula for perplexity is just product of $1/P(W_i|W_{i-1}, ..., W_{i-n+1})$ which implies that we do need smoothing to make sure the denomniator is not 0. So here we only considered smoothed unigrams and bigrams models (smoothing threshold $c < 5$), both with open vocabulary as done in previous sections.

To start with, we calculate the count of each word type in the corpus (preprocessed and incorporated with "<unk>"'s and smoothed) corresponding to the topic. Specifically, the count of unigrams and bigrams are found respectively for our smoothed unigram and bigram models where every word type has a nonzero probability. Next, we preprocessed the test file following the same steps as in Section 2 in order to clean up the test data. Then we are able to calculate the perplexity for that test file $t_1, t_2, ..., t_N$ by using

$$PP = \exp[\frac{1}{N}\sum_{i}^{N} -\log P(t_i|t_{i-1}, ..., t_{i-n+1})].$$

Note that

$$PP_{unigram} = \exp[\frac{1}{N}\sum_{i}^{N} -\log P(t_i)]$$

and

$$PP_{bigram} = \exp[\frac{1}{N}\sum_{i}^{N} -\log P(t_i|t_{i-1})].$$

By using log-scale, we will effectively avoid floating error issues, since many probabilities will be extremely small which is challenging the computing system. Then we repeat this process for each of the 250 test files in `test_for_classification` folder. This is then repeated for other topics.

In summary, we will have a 250×7 table for each of the two models.

6 Topic classification

In this section, we would like to select the best model for topic classification task. Perplexity is employed as the judging criterion here so we will pick the category which gives the lowest perplexity. With this in mind, we would like to obtain higher probabilities of the test data. One way is by varying the threshold of Good-Turing smoothing so that the product of probabilities will be amplified. So we decided to compare four models here: unigram with smoothing at $c < 5$; unigram with smoothing at $c < 10$; bigram with smoothing at $c < 5$; bigram with smoothing at $c < 10$. Our goal here is to find the best model for each topic, which will be used to predict the topic of the test data files.

For each topic, we repeat the following:

- Divide the training data into two sets: training set for training language models (contains 240 files) and development set (contains 60 files) for testing the performance of current training model;
- use training set to get corpus
- calculate perplexities for each file in the development set for 4 models respectively
- repeat this process until all files in training data has been in the development set once and only once

- for each model, calculate the average of all 300 perplexities
- pick up the model with lowest mean perplexity as the best model for the current topic

As last, we decided to use bigram with smoothing at $c < 5$ model for all topics.

7 Context-aware spell checker

Now we would like to further modify our model with spell-checking, for the reason that spell errors might be misleading hence mess up the predictions.

We considered 3 models here: unsmoothed unigram model; unsmoothed bigram model; and smoothed bigram model (smoothed for $c < 5$). For best performance in correction, we would like to test the 3 models at first. For each model, we find its correction ratio following these steps:

- for each topic, divide the training data (300 pairs of files from `train_docs` and `train_modified_docs` folders) into training set and development set (20% of the training data): note that we consider the data pairwise (one correct file from `train_docs` and its corresponding file with typos in `train_modified_docs`)
- obtain the corpus from training set (only correct files were involved here)
- calculate the probabilities under the current training model
- use the probabilities to decide whether to replace the confusing word or not
- repeat this process until all training data have been in the development set once and only once
- compare our output files with the corresponding correct files in `train_docs` to get the count of errors where we failed to correct and/or should not have corrected.

After we obtain the error counts, we found that the unsmoothed bigram model has the lowest error counts hence the best model for spell checking.

There are a few things to clarify here:

- in the forth step above, unigram and bigram models differ:
  - in unigram model, if for example $P(\text{went}) > P(\text{want})$, we will replace all "want" with "went";
  - in bigram model, we will consider bigram probabilities, so the context of "want"/"went" matters: for example if the test text is "I went to", we will see if $P(\text{went}|\text{I}) \cdot P(\text{to}|\text{went}) < P(\text{want}|\text{I}) \cdot P(\text{to}|\text{want})$. If yes, we will replace the "went" with "want", vice versa.
- if there is a tie (e.g. $P(\text{went}) = P(\text{want})$), we just leave the text as it is.
- we are being case insensitive here: "Went" and "went" are treated in the same way when calculating the probabilities and spell correction
- in the `confusion_set` there is a case where "maybe" might be mixed up with "may be", however "may be" is a two-word combination which may mess up with probabilities calculations in unigram/bigram models. So we decided to insert "xyz" to make it "mayxyzbe" as one single word.

Here is the table of error counts we obtained:

	bigram	bigram_GT	unigram
atheism	445	449	609
autos	292	292	324
graphics	249	255	255
medicine	322	322	415
motorcycles	282	290	252
religion	486	479	651
space	552	546	715

From the table we can see that bigram with Good-Turing smoothing at $c < 5$ is working almost the same as unsmoothed bigram. Whilst unigram is doing really bad, just as expected, because unigram fails to take context into account and therefore is expected to do a much less accurate job.

8 Open-ended extension

Our results on *Kaggle* indicate that there is space to make improvements on topic classification. After careful consideration of all the available choices in part 8.1, we decided to implement linear interpolation method in the hope of better prediction performance.

We interpolate the probabilities by

$$P(W_i|W_{i-1}) = \lambda_1 P(W_i|W_{i-1}) + \lambda_2 P(W_i)$$

where $\lambda_1 + \lambda_2 = 1$. So essentially we interpolate the bigram probabilities with unigram probabilities.

To find good combinations of (λ_1, λ_2), we conducted 3 experiments with choices of (λ_1, λ_2): $(0.8, 0.2)$; $(0.5, 0.5)$; and $(0.2, 0.8)$. In this way, we will see if heavier or lighter weights for bigram probabilities will give a better result.

Consequently, we have the output perplexities for each combination (a 250×7 table for each choice of lambda's), and this is further compared to the original perplexity table.

It turns out that the interpolation will give a higher perplexity compared to the bigram model with smoothing at $c < 5$. This disadvantage happens in each cell of the table. One reason for this disappointment might be that this interpolation method is only conducted up to bigram probablilities, but interpolation with trigram and even higher n-model might have a better result.

9 Workflow

Throughout this project, work was distributed to each member fairly and voluntarily. For each section, before coding it up, we would have a thorough discussion about what to do and how to implement. In part 1 of this project, Rui offered to do the preprocessing part. Upon completion of preprocessing, Gi Yoon offered to do the n-gram probabilities calculation and random sentence generator part. Shortly after, we together saw the experiments with random sentence generator working well, and Xiaoyun undertook to summarize all of our work in this report. In part 2, we did all work together, including analyzing the structure of code elaborately, and writing up the report at the same time when coding up the script. Brainstorms have been going on through our entire work, especially that we had a heated discussion about extension ideas and came to the conclusion that linear interpolation will be the best choice. Every time upon agreement, we divided the work into coding and writing the report. We are glad that communication among members has been efficient and work has been done in a timely manner.