NLP Project1 Report

rm879 yp344

1. Unsmoothed n-grams

For unsmoothed n-gram, we don't need to handle unknown words or unseen bigrams. We only need to calculate the probability of each seen word and the conditional probability of seen bi-words. The main idea is simple:

Given the corpus: *the students liked the assignment*

unigram probablity: $\mathrm{P(the)} = \frac{\#(the)}{N} = \frac{2}{5} = 0.4$ N is the length of word tokens

bigram probability: $\mathrm{P(the|liked)} = \frac{\#(liked\ the)}{\#(the)} = \frac{1}{1} = 1$

For programming, we use Hashmap store each word and their count. So we can easily get probability of each word $\#(w_i)$. As for bigram, we first use Arraylist to store every two continuous words and then put them into another Hashmap. So we can get $\#(w_i w_{i-1})$ easily. Then calculate the conditional probability.

1) Dataset

A collection of news articles from the 20 newsgroups for different topics (atheism, autos, religion, medicine etc.).

2) Prepossessing

We use java to read the collection of news articles from the newsgroups dataset and form a string combination for each topic folder as a single corpus. Some rare sentence tokens like "[", "{", "|" and some unique string like Email address are removed from the corpus. We assume the tokens such as ".", "…", "!", "?" are the ending mark and put a </s> behind them. Each first word in the first sentence in each text file and the word behind the ending mark (mentioned before) are treated as the starting mark and put a <s> before them. After testing the sentencing meaning, we finally get the modified and preprocessed dataset.

2. Random sentence generation

1) Develop a random sentence:

Start from <s>, we choose the next word randomly according to its probability

($P(W_i)$ for unigram and $P(W_i|W_{i-1})$ for bigram). W_{i-1} is the last word of the current string. When the next word is </s>, we stop. Then we have the sentence starts from <s> and ends at </s>.

2) Develop with seeding:

It is similar with the random sentence development. Now we don't need to give <s> to start. Instead, we use the last word of the seeding sentence to start generation.

3) Generated sentences (Examples):

Random Sentence Development:

Unigram: I meaning a kempmp

Bigram: Yes , i don't believe that doesn't make into energy as a sweeping statement that theism is legal is fairly clear that moral acts when a mark on your door .

Unigram: Supposed top philosophy milk gun kt to better to exist . g seeing , st

Bigram: So what business of that has also on the book of us to discuss the bottom line of the addresses : 1915 and haven't listened to penetrate such thing that ?

Random Development with Seedings:

Seedings: "They are"

Unigram: They are long (consensus the world . places than near you the get them of think is

Bigram: They are wrong to contact me .

Unigram: They are ?) for : the qualifier reason have it have su relationship the it make no the

Bigram: They are to purely zoroastrian tradition is born with the father : re : (brian the nazis never took a direct witness is eternal damnation an uncountably large means invincible) subject to look .

4) Analysis of the random sentence:

We find for the unsmoothed language model, a random sentence generated from the unigram is more difficult to understand than that from the bigram. That is to say, bigram can generate more real sentences. We can predict that trigram will do better. That is because of as n gets bigger, it takes more contextual consideration. For unigram, it picks each word individually; but for bigram, it generates the next word according to the current one.

We can also see from the sentences above that sentences generated by bigram is more likely to end with a punctuation than unigram. This is because in bigram model, </s> often comes after

punctuation because of our preprocessing. So if we get the punctuation as our current word, based on bigram model, it is very likely we get an </s> for next word and ends. While for unigram, each word is independently picked so we can't guarantee a punctuation when sentence ends.

We can also see the impact of corpus. Sentences generated from corpus with different topics have very different words type.

3. Smoothing and unknown words

To get rid of the 0 probability, we programmed to handle unknown words and smooth as below:

Unigram:

1. After preprocessing the corpus, we replace every first occurrence word as <unk>;
2. For each word, we store its counts and counts of counts (N_c) in Hashmap to prepare for GT smoothing;
3. GT smoothing:

$$c^* = (c+1) * \frac{N_{(c+1)}}{N_c} \quad (\mathrm{c} < 5)$$

$$c^* = c \quad (\mathrm{c} \geq 5)$$

   Then we update the map which stores the word with its counts. The counts turns to c^* form c.

4. Get the probability of each word based on the new map. $\mathrm{p}(w_i) = \frac{c*(wi)}{N} = \frac{c*(wi)}{\sum c*(wi)}$

Bigram:

Handling unknown words and smooth for bigram model is similar with unigram. The difference is we need to store two continuous words and its counts and counts of counts. Then we will meet a new problem: **unseen bigram**. Which means the two words both belong to the corpus but we never see them together. To give the unseen bigram a reasonable c*, we need to calculate N_0.

$$N_0 = |V|^2 - \#(ever\ seen\ bigram)$$

Then we can get smoothed bigram probability: $P(wi|wi\text{-}1) = Count(wi-1wi)/\sum \mathrm{Count}(\mathrm{wi}-1\mathrm{wj})$

What we should notice here is that to calculate the sum, we also need to add c* for all unseen bigram which can be implemented easily by multiplying its count and c*.

Reasons:

Smoothing and unknown words handling is important because of the following reasons. We will

catch numeric error because of a certain $P(w_i)$or $P(w_i|w_{i-1})$ equals to 0. Then the probability of whole sentence will be 0. For those unknown words and unseen bigrams, the unknown word handling and smooth take other words(bi-words) probability to increase their probability. In this case, smoothing act as a role of preprocessing of numeric error and probability re-allocator.

4. Perplexity

We calculate perplexity as below:

$$PP = \sqrt[N]{\frac{1}{P^*(w_1 w_2 \cdots w_n)}}$$

In programming, we use log space to avoid underflow and faster speed.

The perplexity of certain file under different corpus is like below:

	atheism	autos	graphics	medicine	motorcycles	religion	space
Unigram	248.089	164.918	134.008	187.706	123.434	197.436	233.511
Bigram	46.300	61.027	53.587	66.697	44.520	55.238	95.227
Trigram	21.084	34.634	30.325	37.334	28.634	30.549	56.354
Four-gram	18.837	32.054	28.784	34.520	29.007	29.313	54.497

We can draw the conclusion from the table: for the N-gram language model, the value of perplexity is basically smaller when the value of N is bigger. It presents a better language model and a tighter fit to the test data for the model of a smaller perplexity value. The reason for it is because N increases, the prediction in language model has a bigger range of the context and make the prediction more accurate leading to the decrease of the value of perplexity.

5. Topic classification

For each corpus related to the topics, we divided each into a train and a development set by a certain percentage. We change the percentage for many times and find some basic trends. We set the percentage as 80% in the following analysis.

For each topic, the number of the files multiples the percentage equals to the number of the files we are going to use and form the train set. The rest files form the development set. We pick every file from the development set and calculate the perplexity value between this file and the train set in each topic with a same percentage. The topic with minimum perplexity will be considered as “right topic”. If the topic name is the same with the topic name of the original file folder, the

number of files which match the topic adds 1. The accuracy will be get by the percentage of right prediction. If a model can give higher accuracy, we assume it is more suitable to classify files.

在此处键入公式。

A example: (corpus atheism)

	unigram	bigram	trigram	fourthgram
accuracy	0.0	0.35	0.5667	0.5667

We can see from the example: for a specific, the evaluation value, which has a positive relation with the classification accuracy, is higher when the value of N for N-gram is bigger. It may imply a positive relation with the range of the context in language model and the sentence meaning identification & the topic classification. For Kaggle, we use 3-gram and get 49% while 4-gram 55.2% accuracy.

6. Context-aware spell checker

For spell checking, we programmed as below:

1. Use a Hashmap to store each confusion word and another Hashmap to store each pair of confusion words. A word may have more than one confusion word. Like “a”, the confusion word for it is “an”, “and”. We use Arraylist to store them;
2. Use a corpus to train the 4-gram model;
3. Split the input String into a String array and traverse each word of it. If the word belongs to confusion word, we extract it along with its preceding 4 words and subsequent 4 words;
4. For this extracted substring, we detect each confusion word in it. Then we substitute it with its confusion words to make new substitute string. For each new string and original string, we calculate its perplexity based in the 4-gram model. The one with the minimum perplexity is the correct string;
5. Then we put the updated string back to its original location. Continue extract strings till the end.

Evaluate method:

1. To evaluate the accuracy of the spell checking method, we use 80% txt files in train_docs as

training set and 20% as development set. The 80% txt files are connected and form the corpus to train the model. Then we use each of the 20% txt file as input string;

2. For the input string now is actually "correct", ideally, each time it encounter a confusion word, the program will scan all its possible substitute word but still choose the original one. We use *confusew* to store the number of confusion words in the string and *correctw* to store the number of choosing the original one. Then the accuracy will be *correctw / confuse.*

Parts of the result are as below (corpus: space):

data \ file	f45	f46	f47	f48	f49	f50	f51	f52	f53	f54
confusew	7	15	24	41	73	122	145	208	218	339
correctw	7	14	23	40	63	97	116	169	178	270
accuracy	1.00	.933	.958	.976	.863	.795	.800	.812	.817	.796

The accuracy is aroud 80%~100%. So it is safe to use it on spell checking.

7. Open-ended extension

For open-ended extension, we choose trigram and fourthgram. The method to implement them are very similar to bigram. Just $\mathrm{N0} = |V|^3 - \#(ever\ seen\ trigram)$for trigram. And $\mathrm{N0} = |V|^4 - \#(ever\ seen\ fourthgram)$ for fourthgram. From the perplexity and classification above, we can see the advantages of using them.

We also use stupid back-off to handle unseen n-gram. Like for trigram,

$$\mathrm{p}(w_i|w_{i-2}w_{i-1}) = \begin{cases} \mathrm{p}(w_i|w_{i-2}w_{i-1}), if\ c(w_{i-2}w_{i-1}w_i) > 0 \\ \mu 1 * p(w_i|w_{i-1}), if\ c(w_{i-2}w_{i-1}w_i) = 0\ and\ \ c(w_{i-1}w_i) > 0 \\ \mu 2 * p(w_i), otherwise \end{cases}$$

After evaluation, we choose $\mu 1 = 1\ and\ \mu 2 = 1$ for higher performance.