Project1: Language Modeling and Word Embeddings

Mengyao Lyu (ml2559), Ruoyang Sun (rs2385), Qianhao Yu (qy99)

Part 1

Operations:

1. Raw text processing:
   - Break raw text into tokens.
   - Common practice: remove all the single quotation marks '\`'.
   - Add start token (**<\start>**) after each end token, $i.e.$ "A new sentence." Change into "**<\start>** A new sentence."
2. Build Frequency Dictionaries & Tables:
   - Loop through the tokens by each word and each 2 consecutive words, store the counts of each word and word combination in 2 dictionaries, respectively.
   - Use dictionaries obtained above to build 2 frequency tables, making it easier to get the count of word or word combinations.
3. Train bigram and unigram based on conditional probability.
   - For bigram, $\mathrm{P(Wn|Wn-1)} = \frac{\mathrm{Count(Wn-1Wn)}}{\mathrm{Count(Wn-1)}}$
   - For unigram, $\mathrm{P(W)} = \frac{\mathrm{Count(W)}}{\text{Total Word Count}}$
4. Generate Random Sentence:
   - In this case, we ask the user to give the starting word of each sentence and generate random following words according to its probability or conditional probability.
   - To avoid getting the same phases, we sampled the next word according to the current conditional distribution insdead of choosing the word which has the max probability.
   - Generating process stop criterias:
     1. If an end token (we assumed as **'.'**, **'...'**, **'?'**, **'!'** and **';'**) is generated, the algorithm stops generating words.
     2. Set up cut-off length of the sentence: once the length of a generated sentence reaches the cut-off length set by the user, the algorithm stops generating words.
5. Algorithms Parameters
   - path: the path of training dataset.
   - n_gram: takes value "bigram" or "unigram"
   - cut_off: cut off length of the generated sentences.
   - start_word: the initial word of each sentence chosen by the user.

Result Analysis：

(randomly generated sentences for each sentiment and datasets are given as example)

- Dataset: ***pos.txt***
  Method: unigram
  Start word: '<\start>'

```
<\start> part profits along have me shocking story .

<\start> and some long not as the your <\start> movie in of trust <\start> heights like has it <\start> reveal seem

<\start> and the , of the , .

<\start> the that storyline classics .

<\start> striking and , .

<\start> performance certain a as mind both and but apollo deliver acting the unsung funny that a this performances .
```

  According to the randomly sentences generated by unigram and positive datasets, we observe that the generated sentences sometimes have many grammatical errors. For example, in the 3rd sentence, start token is followed by 'and' and no noun follows the article 'the', which makes it very hard to convey information as a sentence. The reason that this happens is that every following word, is independent of the previous words and simply generated randomly according to the probability the word appearing in the dataset. Therefore, in this case it is not ideal to generate random sentence with unigram.

- Dataset: ***pos.txt***
  Method: bigram
  Start word: '<\start>'

```
<\start> it 's back-stabbing , fidgeted or without cheesy fun .

<\start> while remaining one , intermittently hilarious and , that sneaks up against the music , by diane lane works
 the

<\start> an artist who normally could have baffled the mood , which you have worked wonders with conspicuous success
 .

<\start> judging by suggestion , but against practically any movie may also served cold comfort that would have any w
ay ,

<\start> sparse but especially sorvino glides gracefully from new thriller .
```

  Now if we generate random sentences using bigram with the same positive dataset, we observe that the quality of sentences is greatly improved compare to the result of unigram: the first word of sentence is more appropriate, there are fewer grammatical errors and the sentence can roughly some information. The reason of such improvement is that the first word of each sentence is generated according to the conditional probability with the start token, so are all the following words. Bigram model makes the meaning of sentences more fluent. Plus, the model generated by bigram shows that the random generated sentences based on ***pos.txt*** are also very positive.

- Dataset: ***neg.txt***
  Method: unigram
  Start word: '<\start> i'

```
<\start> i a n't and that zero movies proceeds pandora get warmth the king fluffy his between the a lacked ,

<\start> i goes becoming that comes roberts hate is they nymphette dragon unbridled hollywood and too your own rometh
ing spin ,

<\start> i than curmudgeon .

<\start> i a the by movies some jar-jar the the in director .

<\start> i it the left the .

<\start> i cliched until to how so support , be louiso , it casting possibility a sputters -- pandemonium too they
```

Similar to the random sentences generated in the positive dataset, in this case the random sentences also have numerous grammatical problems and fail to convey complete information. The sentiment of the generated sentences is consistent with the training set ***neg.txt***.

- Dataset: ***neg.txt***
  Method: bigram
  Start word: '**<\start> i**'

```
<\start> i found its one-sidedness ...

<\start> i do everything about as a lump of how things that craven of the 1920 's remake of art imitating

<\start> i 'm the 37-minute santa gives movies ever to set ups , lust , defined by men is the cast

<\start> i realized that its effective moments .

<\start> i walked the inherent humor , you 're clueless film is just rather than the future in a former wrestler

<\start> i did n't think about vicarious redemption by 86 minutes too upbeat ending for tapping into hokum .
```

As we expected, the performance of bigram is much better than unigram in both grammar and information convey.

Part 2

Operation:

Section 4. Smoothing and Unknown words

- Unknown words:
  - Add unknown words before smoothing.
  - Represent unknown words by '***<UNK>***'.
  - In the unigram and bigram models, replace words with count 1 by '***<UNK>***'.
- Smoothing
  - Smooth bigram by adding ***k*** (***k*** is trained to make optimal prediction) to each bigram count. Then compute the probability of each bigram $P(W_i|W_{i-1})$ by dividing $\text{Count}(W_{i-1}W_i) + k$ by $(\text{Count}(W_{i-1}) + k * V)$, where V is total number of word type.
  - No smoothing is applied on unigram.

Section 5. Perplexity

- Interpolation: chosen from two modes
  - If bigram probability of current word is too small (smaller than preset threshold, $e.g.$ 1e-8), we simply use the probability of unigram.
  - If probability of current word is large enough, we set the probability of current word as $(\lambda * \text{P(unigram)} + (1 - \lambda) * \text{P(bigram)})$.

    - ***λ*** is found using held-out method.
- Perplexity
    - Clarification of operation: In our project, we use ***interpolate_perplexity()*** to calculate the sum of negative log probability of a sentence in sentiment classification and ***perplexity()*** to measure the total perplexity of the whole corpse.
    - In this project, we are trying to minimize the perplexity of both unigram and bigram models.

Result: Perplexity of the ***dev***

Perplexity	Unigram	Bigram
pos.txt	312.99	309.20
neg.txt	283.60	258.43

According to the perplexity of the whole corpse calculated above, we observe that perplexity is lower for ***neg.txt*** than for ***pos.txt.*** In general, the performance of bigram is better than unigram since perplexity of bigram of both ***pos.txt*** and ***neg.txt*** is lower. However, as for ***pos.txt*** specifically, the performance of unigram is very close to bigram. For ***neg.txt***, the performance of bigram if better due to its lower perplexity.

Section 6. Sentiment classification

- Our classification method is based on perplexity of the input sentences. We have two ngram models, positive and negative. If the perplexity of one certain model is smaller than the other one, we label the sentence to the former class, and vice visa.
    - We used "**telescope search**" to find the optimal ***k*** and ***λ***. We set them to a reseanable range of values, and narrow it down by seleting the best accuracy models.

Accuracy	k = 0.001	k = 0.002	k = 0.003	k = 0.004	k = 0.005
λ = 0.	0.51636364	0.51757576	0.52	0.51636364	0.51636364
λ = 0.333	0.49333333	0.48848485	0.48969697	0.48727273	0.48363636
λ = 0.667	0.45090909	0.45090909	0.44848485	0.44363636	0.43878788

According to the table above, we find that when ***k*** = 0.003 and ***λ*** = 0, we obtain the highest accuracy.

Section 8. Sentiment classification with word embeddings

- In this section, the procedure was divided into two parts: different methods of word embedding and machine learning algorithms for classification.
- Word embedding method
    - Using pre-trained word embedding (word2vec and glove separately, both have d = 300). To compute the vector representation of a sentence, we simply averaged the all the word vectors in the given sentence.
    - TF-IDF weighting scheme
- If the word doesn't appear in the pre-trained word embedding dictionary, we omitted the word and didn't considered it in the sentence aggregation. We decided to handle unseen

word this way because we believed if that word was not in the training dictionary, it might be too rare and not very informative.

- After constructing the design matrix with each method above, we would try different machine learning methods in order to do sentiment classification below.
  - Neural Network
  - Support Vector Machine (SVM)
  - Naïve Bayes

  Notice that we didn't use multinomial Naïve Bayes, we use Gaussian NB.
- To choose the optimal parameters for each method, we took the held-out validation approach: trained different models on the data set in Training folder and tested the model performance on the data set in **Dev** folder. We compared the classification accuracy of each model on the **Dev** data.
- Accuracy on Kaggle: 81.33%, which is based on mutinomial Naïve Bayes method with alpha = 0.71
- Analysis: We found that the key of this task is to choose the best feature, rather than the classifier. The TF-IDF feature is the best among the features we tried.

Prediction accuracy on the Dev set:

	Word2vec	GloVe	TF-IDF
Neural Network	0.592	0.598	0.784
SVM	0.565	0.571	0.784
Naïve Bayes	0.566	0.575	0.796

Workflow:
Mengyao Lyu (ml2559):

1. Created probability table, dictionary of unigram, bigram.
2. Performed random sentence generation.
3. Interpolation function.
4. Researched word pre-trained word embedding.
5. Improved Sentiment classification model using Neural Network.

Ruoyang Sun (rs2385):

1. Perplexity of corpse
2. Created model (classification function)
3. Performed random sentence generation.
4. Improved the sentiment classification model using SVM, Naïve Bayes
5. Improved the sentiment classification model using ensemble method.
6. Project write-up.

Qianhao Yu (qy99):

1. Created probability table, dictionary of unigram, bigram.
2. Performed smoothing technique and add unknown words.
3. Found smoothing parameter k and interpolation parameter lambda
4. Researched word representation method.

5. Project report write-up.

Reference

Pre-trained corpus

Word2vec https://code.google.com/archive/p/word2vec/

Glove https://nlp.stanford.edu/projects/glove/

TF-IDF package

sklearn.feature_extraction.text.TfidfVectorizer

Other machine learning method
sklearn