

CS4740: Intro to NLP

Project1: Language Modeling and Word Embeddings

Name: XINGYU CHEN NetID: xc374

Name: BYRAN MIN NetID: km567

Part1. Random Sentence Generation

In this part, we have tested random sentence generation tasks on several n-gram language models using both Obama and Trump datasets. The experiment involves unigram, bigram, trigram and quadgram models to generate sentences based on weighted sample distributions.

Section2.Unsmoothed n-grams

We decided to use python for the project, because it was the language that we were both most familiar with. In this section, we have compute unsmoothed unigram and bigram probabilities. To do so, we did preprocessing of text dataset from training corpora for Trump's and Obama's speeches.

Preprocessing

The raw input file needs to be preprocessed for better tokenization, below steps are taken:

1. Delete leading and trailing white space and new line denotation '\n' and tab space '\t'
2. Re-assemble words like 'there ' s', "they ' ve", "do n't", "don 't" into consistent form. Delete space around single quotes, e.g. "do n't", "don 't" become 'don't'.
3. Add start of sentence token <s> and end of sentence token </s> for each sentence. Replace all the '!?.' Symbols, since they are the equivalent of end sentence token and we made an assumption that every sentence should end with </s> token and start with <s> token
4. Replace string of numbers like "2 percent" "100 percent" into English characters of words, "two percent" and "one hundred percent". Therefore, we have aggregate words like 'one', 'two' if a speech gives lot of statistical information. This modification might be able to reflect some distinct speech features for the later speech classification task, e.g. Trump's speech might contain more statistical demonstrations.
5. Separate concatenation of punctuation and word like 'and,' by adding space. Then apply tokenization by using white space as the delimiter.
- 6.

N-gram generation

We created dictionaries to map n-gram key and its associated count using the collected tokens. The dictionaries are then sorted to generate a probability dictionary to map the key to its associated probability. The unigram probability is obtained by using the count of specific tokens

divided by the entire number of tokens in the corpus. To calculate the n-gram($n \geq 2$) probability, it follows the formula below:

$$\text{prob}(w_k | w_{k-1}^{k-n+1}) = \frac{\text{count}(w_k^{k-n+1})}{\text{count}(w_{k-1}^{k-n+1})}$$

where w_{k-1}^{k-n+1} represents the preceding sequence of $n-1$ gram tokens before word $w(k)$

An example of bigram probability dictionary is shown below:

```
(',', '---'): 0.16666666666666666,
(',', ' '): 0.16666666666666666,
(',', 'so'): 0.16666666666666666,
(',', 'thank'): 0.16666666666666666,
(',', 'there's'): 0.16666666666666666,
(',', 'we're'): 0.16666666666666666,
(',', '101st'): 9.256687957048968e-05,
(',', '1st'): 9.256687957048968e-05,
(',', '2d'): 9.256687957048968e-05,
(',', '503d'): 9.256687957048968e-05,
(',', '5:00'): 9.256687957048968e-05,
(',', '8th'): 0.00018513375914097936,
(',', '['): 9.256687957048968e-05,
(',', '~'): 0.003795242062390077,
(',', 'a'): 0.012033694344163659,
(',', 'abided'): 9.256687957048968e-05,
(',', 'aboard'): 9.256687957048968e-05,
(',', 'about'): 0.0006479681569934278,
(',', 'above'): 9.256687957048968e-05,
(',', 'abraham'): 9.256687957048968e-05,
```

Figure 1. Part of bigram probability dictionary

For the unigram probability, it is obtained by using the count of specific tokens divided by the entire number of tokens in the corpus.

We also have verified that for the unigram, the probabilities of all tokens sum up to 1. Also, for all bigrams that has the same preceding word in the key, their probabilities sum up to 1, e.g. like ',' in the Figure 1.

Section3.Random sentence generation

For this section, evaluations and comparisons would be made on models generated by Obama datasets and Trump dataset.

The words are generated individually step by step. The next word generation is sampled from the word or n-gram probability distribution. Basically, a word has a higher probability given by its preceding words will be more likely to be generated and vice versa.

The specific method of generating next word is shown in the code snippet below:

```
def weighted_choice(choices):
    #choice is set of the all the possible words given by a preceding sequence of words
    #for bigram that is all the possible words followed by a specific word
    if choices==[]:
        return [None]
    #sum all the count in the choices set
    total = sum(w for c, w in choices)
    #gerate a random number in the range of (0, total_count)
    r = random.uniform(0, total)
    upto = 0
    # iterate over all the choices of words
    for c, w in choices:
        if upto + w > r:
            #once the cumulative prob exceeds the generated count r, we return the word
            return c
        upto += w
```

Also, the sentence generation will stop when the next word to be generated is the end of sequence

token symbol '</s>'.

Unigram, bigram, trigram and quadgram models from both data sets are all experimented.

Evaluation:

For the evaluations, there are basically two tasks:

1. Seeded sentence generation where the several words in the start of sentences are predefined
2. Unseeded sentence generation where no words are given, sentences are generated from scratch.

Seeded sentence generation: sentences are generated based on given tokens 'so that's why' and 'we're going to'.

| Model | Dataset | Start with "so that's why" | Start with 'we're going to' |
|----------|---------|---|--|
| Unigram | Obama | <s> so that's why we </s> | we're going to they from this month , of american states those skills health i small continue were </s> |
| | Trump | <s> so that's why we decades that veterans suspects drug those actually on </s> | <s> we're going to with day is , he our killing see number new together means we </s> |
| Bigram | Obama | <s> so that is why i want to run , i will never been lifted -- present ; the scenes in brazil , or turn , that measure of medford </s> | <s> we're going to harness domestic spending and assume the images of us resolve , you represent american foreign oil ; of guns purchased through this recession , listening across borders are hurting </s> |
| | Trump | <s> so that is why i also confronting some of business </s> | <s> we're going to detroit , make , if they may god bless all gun </s> |
| Trigram | Obama | so that's why we relentlessly gather the facts </s> | <s> we're going to blow up the street and wall off america from the tyranny of drug cartels , dictators and sham elections </s> |
| | Trump | <s> so that's why we stand together , hunting or sport shooting , but joy cometh in the broader goal of using old tensions to scapegoat other countries at the dawn of history poses challenges that threaten to wreck the entire middle east and north africa </s> | we're going to spark new ones </s> |
| Quadgram | Obama | <s> so that's why we have to build the capacity of afghans , partnering | <s> we're going to support efforts to build healthy |

| | | | |
|--|-------|--|---|
| | | with communities and police and security forces will be prepared to stand up and accept our gratitude for your remarkable service , especially during the holidays </s> | relationships between parents as well -- - every family , every business , because it is right for us to get even more involved </s> |
| | Trump | <s> so that's why we welcome efforts like saudi arabian king abdullah's interfaith dialogue and turkey's leadership in the world can not now stand by and wait for america to solve the problem </s> | <s> we're going to lift up our educational system , but we will emerge from our tests and trials and tribulations stronger than before </s> |

Table1. Seeded sentence generate on language models

| Model | Dataset | Generated sentence from scratch |
|----------|---------|---|
| Unigram | Obama | <s> we're the kind of the road is all to help , i recognize this a new foundation of freezing water and privacy , they come down the people to all </s> |
| | Trump | <s> and they didn't they take our support me </s> |
| Bigram | Obama | <s> that's my reality </s> |
| | Trump | <s> i watched television – you know , we're going to make speeches without having to do it </s> |
| Trigram | Obama | <s> and we're not going to be smart and efficient and more just , and to protect ourselves and for them and what will happen if we have always loved what they did when i first arrived on this goal </s> |
| | Trump | <s> but i don't want them in jail for eight years and have to put in a rainy rough day </s> |
| Quadgram | Obama | <s> instead , she chose a life of ease </s> |
| | Trump | <s> we have no idea where they come from , wherever they were born , i don't know </s> |

Table1. Unseeded sentence generate on language models

Summary:

The generated sentences for unigram model does not have coherence and some of the sentences are not even complete; the bigram model becomes better that it contains correct word combinations and is somewhat coherent. It is observed that bigram is current for generated sentences that are short in length. Trigram and quadgram generated sentences in most the cases are complete, coherent and understandable

Section 4. Smoothing and unknown words

This part is to prepare for the speech classification. Consider that later we will use perplexity as our evaluation metric for both datasets, we decided to choose a vocabulary set that is fixed in advance for both datasets. To do this, we followed the steps below:

1. Combine the train and dev file to get more data and do it for both Obama and Trump datasets
2. Find the common vocab in both datasets, which contains 4160 tokens
3. Find the word that at least occur two times in the Obama dataset but not shown up in the Trump datasets, which contains 3392 tokens
4. Find the word that at least occur two times in the Trump dataset but not shown up in the Obama datasets, which contains 735 tokens
5. Combine the tokens obtained in 2,3,4 to get around 10000 tokens as our predefined vocab

To deal with unknown words:

1. Convert in the training set any word that is not in vocab set as unknown words and map a token of "UNK"
2. Do the same thing for the imported text corpus, replace the unknown word by "UNK"
3. Update the probability dictionary for the unigram model, adding an "UNK" key
4. Map the unknown word probability based on the count of "UNK" in the training file

To deal with unseen bigram:

1. Recollect the bigram tokens and update the dictionary for bigram, after mapping the 'UNK' key
2. The probability of unseen bigram is calculated by add-1 smoothing with the formula below:
Prob(unseen_biagram(a,b))=1/(count(a)+Vocab_size), count(a) is the number of count of bigrams that starts with token a,

Note for the step 2, we didn't explicitly create a dictionary for every unseen bigram, because they are essentially same for all the unseen bigrams if they start with same word token, we calculate it on the fly and use it while we meet a specific unseen bigram during testing. The dictionary we created is only recording the probability of seen bigrams.

Section 5. Perplexity

Perplexity is given by the formula below:

$$PP(W) = \sqrt[N]{\prod_{i=1}^N \frac{1}{P(w_i|w_{i-1})}}$$

To calculate the perplexity efficiently, we took a log of probability and so we can apply additions over each word instead of multiplication. The generated perplexity result using add-1 smoothing on the development corpus is given by table below.

For the unigram perplexity, we include end of sentence token but not take account the beginning sentence token in total number of vocabs N. This is done by setting probability of <s> in unigram gram model should be mapped to 1 explicitly after smoothing.

For the bigram perplexity, we excluded the count of bigram (</s>, <s>) in the total count of bigram keys N, since the probability of this key is always 1. This is done by not apply smoothing on bigram that starts with </s>.

Validation of perplexity on dev set

We test the prediction accuracy on the dev set using the unigram models and bigram models. The dev set contains 200 samples which are mixture of Obama's speech and Trump's speech.

| | Obama's dataset | Trump's dataset |
|--------------------------|-----------------|-----------------|
| Unigram model from Obama | 548.49 | 541.58 |
| Unigram model from Trump | 775.41 | 338.60 |
| Bigram model from Obama | 796.93 | 1111.51 |
| Bigram model from Trump | 1921.515 | 500.2 |

Table 3. Perplexity results for different models on two dev datasets

We can see from table 3, lower perplexity of the model on a text file, then the text file is more likely to be derived from the same context with which we built the model. From the table above, we can conclude that both unigram and bigram is able to classify the correct corpus. Also, there are more distinct different for the bigram model over different datasets. Thus, the bigram classifier outperforms the unigram classifier.

The result of having very large perplexity is because the vocab size we chose is large.

Part2. Classification

Section6. Speech Classification with Language models

To verify the method of calculating perplexity shown in the last section. We trained the language model and tested the speech classification accuracy over the dev set. The speech classification task is to classify a part of speech sequences as Obama's speech or Trump's speech.

Add-1 smoothing

The unigram model and bigram models are implemented with add-1 smoothing. The validation accuracy over the development set is shown in table 4. The dev set contains 100 sample of trumps speech and 100 samples of Obama speech. In this validation experiment, only the bigram models are considered.

| Models | Obama dev set | Trump dev set | overall accuracy |
|---------------|---------------|---------------|------------------|
| Unigram model | 0.965 | 0.92 | 0.935 |
| Bigram model | 0.98 | 0.95 | 0.965 |

Table 4. Validation accuracy for unigram and bigram models

Add k smoothing

We also explored the add-k smoothing as comparison. We choose the best k by validating the prediction accuracy over the dev set.

The validation accuracy in the dev set for different Ks is shown the table below

| K | Obama dev set | Trump dev set | overall accuracy |
|---|---------------|---------------|------------------|
| 1 | 0.98 | 0.95 | 0.965 |
| 2 | 0.97 | 0.93 | 0.94 |
| 3 | 0.93 | 0.92 | 0.925 |
| 4 | 0.86 | 0.92 | 0.89 |
| 5 | 0.73 | 0.91 | 0.82 |

Table 5. Validation accuracy for different k in Laplacian smoothing over the dev set

As the k increases, the accuracy keeps decreasing. Thus, the best k is set as 1. For curiosity, we picked k as numbers smaller than 1 when calculating the probability and surprisingly get better result.

| K | accuracy of Obama dev set | accuracy of Trump dev set | overall accuracy |
|-----|---------------------------|---------------------------|------------------|
| 0.1 | 1 | 0.95 | 0.975 |
| 0.2 | 1 | 0.94 | 0.97 |
| 0.5 | 0.99 | 0.94 | 0.965 |

Table 6. Validation accuracy for decimal numbers of k in Laplacian smoothing over the dev set

From table6, improved accuracies using decimal numbers of k (<1) suggests that the unseen bigram

probability should be lower than the probability achieved by add-1 smoothing. A decimal number of K will result in higher probability of seen bigram and lower probability of unseen bigram. It reflects that some of the unseen bigram combination is highly unlikely and should not be smoothed to have 1 count same as others. Therefore, considering add- k smoothing, even if $k=1$ is best of choice yet is still limited in this regard. This finding also encouraged us to use a more advanced smoothing method.

Prediction on test set and Kaggle score

For this part, we combined the training and develop set to get more data. We computed perplexity for data in each line in the test file to compare model performances. We used add-1 smoothing consistently, the bigram models we built got us the currently highest ranking on Kaggle, with an accuracy score of **0.98**.

To seek some improvements, we added more words into vocab set could reduce the probability of 'UNK' in the training set making a total number of vocab being around 11700. This is done by selecting half of words that occurred only once in the train and dev sets into the vocab making a total number of vocab being around 12000. With this trick, we improve the accuracy score to **0.985**.

Section 7. Evaluate Word Embeddings

The task in this section is to identify words based on the idea that the words occur in similar contexts are similar in terms of word embeddings.

For example, given an analogy sets and an incomplete set:

| | |
|------------------|------------|
| Mother - Mothers | Father - ? |
| a b | c d |

Find the word “?” that corresponds in the same way as the word ‘Mother’ corresponds to the word ‘Mothers’. We implemented the model to find the most matching word to an incomplete analogy word pair given a type of relationship defined in a complete analogy word pairs. This is done by computing the cosine similarity with respect to all the words in the vocabulary. Theoretically, the cosine similarity is computed between vector $(b-a+c)$ and entire set of word embeddings of the stored pretrained word embedding file to find the closest one. It will be mentioned that during some modification during implementation considering the running time of the algorithm.

For this part, we didn’t rely on the help of external library like gensim, we implemented it naively for the search algorithm based on cosine similarity using matrix operations.

Dataset

Two different types of dataset are chosen for performing the analogy task

A. Glove:

1. “glove.6B.300d.txt”, small set, which contains 400,000 tokens. This file is used for creating our vocabulary for loading embedding and it defines the search range of most similar word embeddings. We also used it as comparison to the embedding trained with ‘840B’ version.
2. “glove.840B.300d.txt”, large set, which contains around 2.2 million (2,195,875) tokens

B. Dependency-based embeddings: 174015 tokens in total, relatively small compared to Glove

Analog Task 1

Initially, we loaded the vector file “glove.840B.300d.txt” to construct a dictionary to store embeddings for all the 2.2 million words. The text file contains 13 It turns out computing the most similar words over the entire analogy dataset is timing consuming since we will end up computing 2.2 million times of cosine similarity calculation. This task is not easy without the help of external library. Potentially, we can break the file into trunks, use Spark library to process them in parallel. Yet, we take a different approach. Instead, we took a different approach: to narrow down the search range and define a smaller size of embedding set that contains only 400,000 million rather than 2.2 million words

Instead use all the words in the Glove file, we predefined some vocabs to narrow down the search range of the algorithm. We load vocab stored in vector file “glove.6B.300d.txt” to construct a smaller vocabulary. We also extract the words occurred in the analogy sets. The final vocab is obtained by taking the union of them, consisting of 4,00,379 words.

Also, to make the algorithm to operate faster, we used matrix operations of computing cosine distance and finding the k most similar vector for the entire analogy data set instead of using for-loops.

A single matrix operation would be still very slow even if we already reduced the embedding set size, so we also break the training matrix set into submatrices to alleviate the CPU load on our laptop during operation.

Task1

A portion of correct/false prediction result of using Glove and Dependency-based embeddings are shown below:

10 correct predictions based on Glove vector:

| | | | |
|--|-----------------|----------------|-----------------|
| data:['Netherlands' 'Dutch' 'Switzerland'] | label:Swiss | pred:Swiss | result: correct |
| data:['Macedonia' 'Macedonian' 'Colombia'] | label:Colombian | pred:Colombian | result: correct |
| data:['Mogadishu' 'Somalia' 'Tripoli'] | label:Libya | pred:Libya | result: correct |
| data:['father' 'mother' 'uncle'] | label:aunt | pred:aunt | result: correct |
| data:['Rabat' 'Morocco' 'Algiers'] | label:Algeria | pred:Algeria | result: correct |
| data:['Vientiane' 'Laos' 'Accra'] | label:Ghana | pred:Ghana | result: correct |
| data:['Italy' 'Italian' 'Peru'] | label:Peruvian | pred:Peruvian | result: correct |
| data:['Colombia' 'Colombian' 'Poland'] | label:Polish | pred:Polish | result: correct |
| data:['Russia' 'Russian' 'Peru'] | label:Peruvian | pred:Peruvian | result: correct |
| data:['Caracas' 'Venezuela' 'Kiev'] | label:Ukraine | pred:Ukraine | result: correct |

10 false predictions based on Glove vector:

| | | | |
|---|--------------------|-------------------|-------------------|
| data:['furious' 'furiously' 'rare'] | label:rarely | pred:rare | result: incorrect |
| data:['large' 'larger' 'high'] | label:higher | pred:high | result: incorrect |
| data:['large' 'larger' 'simple'] | label:simpler | pred:simple | result: incorrect |
| data:['happy' 'happily' 'quick'] | label:quickly | pred:quick | result: incorrect |
| data:['Angola' 'kwanza' 'Armenia'] | label:dram | pred:kwanza | result: incorrect |
| data:['melon' 'melons' 'cloud'] | label:clouds | pred:cloud | result: incorrect |
| data:['precise' 'precisely' 'occasional'] | label:occasionally | pred:occasional | result: incorrect |
| data:['Lexington' 'Kentucky' 'Chicago'] | label:Illinois | pred:Chicago | result: incorrect |
| data:['Irvine' 'California' 'Philadelphia'] | label:Pennsylvania | pred:Philadelphia | result: incorrect |
| data:['lion' 'lions' 'cloud'] | label:clouds | pred:cloud | result: incorrect |

Summary of performance for glove

1. Glove vectors performs good on the object/location related analogy task, few examples failed.
2. Glove vectors performs poorly on mapping word of single form to its plural form, and most failed.
3. Glove vector performs poorly on mapping adjective word to adverb word, most failed. For example, precise-precisely, occasion-occasionally

10 correct predictions based on Dependency vector:

| | | | |
|-------------------------------------|-------------------|------------------|-----------------|
| data:['car' 'cars' 'computer'] | label:computers | pred:computers | result: correct |
| data:['small' 'smaller' 'cheap'] | label:cheaper | pred:cheaper | result: correct |
| data:['donkey' 'donkeys' 'horse'] | label:horses | pred:horses | result: correct |
| data:['bottle' 'bottles' 'machine'] | label:machines | pred:machines | result: correct |
| data:['monkey' 'monkeys' 'dog'] | label:dogs | pred:dogs | result: correct |
| data:['large' 'larger' 'heavy'] | label:heavier | pred:heavier | result: correct |
| data:['uncle' 'aunt' 'grandfather'] | label:grandmother | pred:grandmother | result: correct |
| data:['cow' 'cows' 'dog'] | label:dogs | pred:dogs | result: correct |
| data:['bright' 'brighter' 'small'] | label:smaller | pred:smaller | result: correct |
| data:['young' 'younger' 'tough'] | label:tougher | pred:tougher | result: correct |

10 false prediction based on Dependency vector

| | | | |
|---|---------------------|------------------|-------------------|
| data:['dad' 'mom' 'grandson'] | label:granddaughter | pred:grandson | result: incorrect |
| data:['brother' 'sister' 'man'] | label:woman | pred:man | result: incorrect |
| data:['groom' 'bride' 'grandpa'] | label:grandma | pred:grandpa | result: incorrect |
| data:['brothers' 'sisters' 'policeman'] | label:policewoman | pred:policeman | result: incorrect |
| data:['brothers' 'sisters' 'stepbrother'] | label:stepsister | pred:stepbrother | result: incorrect |
| data:['brothers' 'sisters' 'brother'] | label:sister | pred:brother | result: incorrect |
| data:['his' 'her' 'uncle'] | label:aunt | pred:uncle | result: incorrect |
| data:['grandpa' 'grandma' 'grandfather'] | label:grandmother | pred:grandfather | result: incorrect |
| data:['grandson' 'granddaughter' 'boy'] | label:girl | pred:boy | result: incorrect |
| data:['groom' 'bride' 'policeman'] | label:policewoman | pred:policeman | result: incorrect |

Summary of performance for dependency vector

1. It performs better on comparative analogy set, single-plural analog compared to glove
2. It performs poorly on object related analogy set compared to glove

Also, we found most of false prediction give exact same word vector as the third vector in the training sample. For example, the prediction result for ['brother', 'sister', 'man'] is still 'man' rather than the current word 'woman'. Given the analogy set: lion-lions, cloud-?', the model usually predicts the most similar word of "?" is 'cloud' instead of clouds. A lot of similar cases failed and give the same type of error.

Surprisingly, the correct prediction in most cases are the second most similar word if we search for the second closest word embedding. Therefore, we can include that prior knowledge into modelling and make prediction upon it that if the most similar word is same as the third column word in the training file, then output the second most similar word. The improved accuracy could be viewed in the table below:

| Models | Accuracy | Improved Accuracy |
|-----------------------------|----------|-------------------|
| Glove (6B) | 25.4% | 54.8% |
| Glove (840B) | 37.5% | 75.7% |
| Dependency-based embeddings | 1.65% | 16.7% |

Table7. Overall accuracy of analogy task for 3 different pretrained embeddings. 'Accuracy' refers to the original implementation of finding the most similar word. The improved implementation's accuracy is shown in the last column.

| Vocab | Task | Running Time |
|---|--------------------------------|--------------|
| Vocabs from the text file (size:659) | Going over 1 example | ~0.0002s |
| | Going over entire training set | ~0.5s |
| Vocabs of 6B glove embedding set (size:400386) | Going over 1 example | ~0.03s |
| | Going over entire training set | 560s |

Table 8. running time of implemented algorithm

From the running time analysis, we can see the time cost increases hugely with the increase of pre-defined vocabulary set. With huge vocab, time cost of going over the entire dataset with our naïve implementation of matrix operation might still be significant. Yet, for small vocabs, the algorithm performs a lot better, and time cost is acceptable.

Task2

Two types of analogy tasks are customized.

A. Antonyms: 3 examples are given below

| Task | Label | Prediction Glove(300B) | of Dependency data Prediction |
|-----------------------|-------|---------------------------|-------------------------------------|
| small-big short-? | tall | big | short |
| Young-old beautiful-? | ugly | beautiful | beautiful |
| Sharp-blunt sweet-? | Sour | Sweet | Spicy |

B. Intensity related: 3 examples are given below

| Task | Label | Prediction Glove(300B) | of Dependency data Prediction |
|----------------------------|---------|---------------------------|-------------------------------------|
| irritation-anger mist-? | rain | clouds | anger |
| Starving-hungry drenched-? | damp | dripping | transfixed |
| Like-fancy dislike-? | despise | dislike | dread |

For both tasks, two embeddings gave no correct prediction. For the antonyms, both datasets perform poorly, and the predictions make no connection between the input analogy. Yet, for the intensity related task, the prediction gave by the Glove vector is closer to the actual label compared to the dependency dataset.

Task3

10 most similar words for verbs are shown below by running the Glove model
(from left to right from most similar to less similar)

word: cat

10 most similar words from Glove:

cats, kitten, dog, kitty, pet, puppy, kittens, feline, dogs, kitties

word: enter

10 most similar words from Glove:

entering, entered, re-enter, must, proceed, to, leave, go, submit, able

Word: increase

10 most similar words from Glove:

increased, decrease, increasing, increases, reduce, decreased, significantly, higher, decreasing, reduced

It can be shown that in the 10 nearest neighbors, for a given word, its antonym always exists. One effective way to find antonym is to apply word stemmer to these 10 neighbors. The first most frequent word stemmer might be originated from the word itself, yet the second most frequent word is always the desired result – the input word's antonym.

Section 8. Speech classification with word embeddings

This section focuses on speech classification using word embeddings. Each speech is represented by an average word embedding of words in that set of speeches. We tried two different embeddings, one is the glove(300b) vector and the other is ELMO vector from Allennlp.

ELMO vector bin outputs adaptive word embedding for a word based on the sentence context, which means same word will have different word embedding for different sentences. In this regard, the computing of ELMO vector takes a comparatively longer time and the RAM in my computer is not able to convert and load the entire training data in memory all in once. Therefore, we broken the corpus into trumps, and aggerating the word vector for each chunk and finally diving the total number of word vectors for a given speech.

We trained two models using only the training corpus and computed the accuracy by running the model on the dev set.

The result is shown in the table below

| Dataset | Classification accuracy |
|-------------|-------------------------|
| Glove(6B) | 78% |
| Glove(840B) | 80.5% |
| Elmo | 85% |

Table 9. Results of speech classification using word embeddings

8.1 Machine Learning

We also ran machine learning approaches, specifically neural network to classify sentences.

The training corpus is preprocessed to remove stop word. For the input to the model, rather than an average vector of speeches embeddings, instead we fed sequences of word embeddings for each sentence (each sentences are padded into same length) and so the LSTM networks are able to learn the context. Also, a layer of CNN is used prior to the LSTM layer for feature extraction and dimension reduction. The result achieved by Neural Network is not optimized, potentially, we can have a better accuracy if adding learning rate optimizer, regulation, etc.

Also, an improvement has been made by making the model to learn the word embeddings in the same time it optimizes the weights (make the embedding layer trainable). In this way, the trained word embedding is more suitable for the speech context compared to untrained (pretrained) word embeddings

The result is shown in the table below:

| Dataset | Classification accuracy |
|--|-------------------------|
| Glove(6B) | 79% |
| Glove(840B) | 80.5% |
| Elmo | 84.5% |
| Neural Network with untrained word embedding | 91.5% |
| Neural Network with trainable word embedding | 94.5% |


Table 10. Comparison between different models

The results proved that neural network perform better compared to our previous implementations. Also, making the embedding trainable will further improve the result.


Kaggle submission

Name: XIAOXINGXING

Language model:

| # | △1w | Team Name | Kernel | Team Members | Score 🏆 | Entries | Last |
|---|-----|--------------|--------|--|---------|---------|------|
| 1 | — | XIAOXINGXING | |  | 0.98500 | 16 | 2d |
| Your Best Entry ↑ Your submission scored 0.98000, which is not an improvement of your best score. Keep trying! | | | | | | | |

Word embedding: (the 0.975 is a false submission, the one we got is the newest submission which is 0.945)

| | | | | | | | |
|---|-----|--------------|--|---|---------|----|-----|
| 2 | new | XIAOXINGXING | |  | 0.97500 | 13 | 10h |
| Your Best Entry ↑ Your submission scored 0.94500, which is not an improvement of your best score. Keep trying! | | | | | | | |

Workflow

We participate all the required work listed in this report, we did not explicitly divide work. Xingyu is responsible for the glove vector, and Byran is responsible for the dependency dataset. Xingyu did extra work on Machine Learning and Elmo vectors.