Advanced Natural Language Processing (968G5)

Assessed coursework 1

Abstract:

Although work on modelling semantics in text is rapidly developing, there are currently few publicly available datasets that writers may use to compare and assess their systems. This effort contributes to the addressing of the problem. Here we are using one dataset and choosing the two different models. Project Gutenberg data was used to create this dataset. Seed phrases were chosen from five Sherlock Holmes books by Sir Arthur Conan Doyle, and then impostor words were proposed using a language model trained on over 500 nineteenth century works. Human judges chose the four best imposter words based on a set of predefined guidelines after the language model computed 30 alternative words for a specific low frequency word in a text. Although the data given here will not be modified, this is currently a work in progress, and we want to incorporate similar datasets from other sources in the future. This technical report is a dynamic resource that will be updated when new datasets are created and new results on current datasets (such as those involving human participants) are published.

Introduction:

Language is a very effective means of communication. We have the capacity to create projects from the ground up utilizing linguistic subtleties. It's what initially drew me to Natural Language Processing (NLP). Here we are experimenting data with two different models for sentence completion challenge. Here I use the language model with sherlock homes text files and the word2vec method using the google news vectors. Language modelling is a method for calculating the likelihood of any word sequence. Language modelling is utilized in a broad range of applications, including Speech Recognition and Spam Filtering. The development of many state-of-the-art Natural Language Processing models is driven by language modelling. A word embedding is a method for creating a dense vector representation of words that encapsulates some aspect of their meaning. Word embeddings operate by training a collection of fixed-length dense and continuous-valued vectors based on a huge corpus of text using an algorithm. Each word is represented in the embedding space by a point, which is learnt and moved dependent on the words that surround the target word.

Process of Creation:

The possibility of a certain word inside any sequence of words in a language is predicted by a language model. We can predict p(w | h) – what is the likelihood of encountering the word w given a history of prior words h – where the history comprises n-1 words – if we have a good language model. This probability is calculated in two steps:

* Use the probability chain rule.
* We then use a very strong simplification assumption to make it simple to compute p(w1...ws).

The chain rule of probability is:

P(w1…ws) = p(w1) . p(w2 | w1) . p(w3 | w1 w2 ) . p(w4 | w1 w2 w3 )….. p(wn | wn-1)

This is where a simplification assumption is made. For all situations, we may assume:

P(wk | w1….wk-1) = p(wk | wk-1)

By examining simply the final word of the context, we may approximate the history (context) of the term wk. The Markov assumption is the name for this assumption. (We utilised it here in a simplified setting of length 1 – which corresponds to a bigram model; in general, we might use bigger fixed-sized histories.)

Let's make a trigram language model now that we know what a language model is. To undertake a very basic analysis of the Sherlock stories, we'll utilise the Python Regular Expression and NLTK packages. The training directory contains 523 files.

Language Model Implementation:

The specialty of language modelling is determining the likelihood of a sequence of words. These are helpful in a variety of NLP applications, including machine translation, speech recognition, optical character recognition, and many more. In recent years, language models have relied on neural networks to predict a word in a phrase based on surrounding terms. However, we shall cover the most traditional of language models in this project: n-gram models.

I have implemented 3 different Language model that is unigram, Bigram and trigram. A unigram model may be thought of as a collection of finite automata with only one state.

Unigram : It divides the probability of several phrases in a context, such as from and to. Because each word's probability in this model is only determined by that word's own probability in the text, we can only use one-state finite automata as units.

Bigram: The bigram model, as the name implies, uses just the conditional probability of one preceding word to estimate the likelihood of a word given all prior words. In other words, you use probability to approximate it: (the | that) P(the | that).

Trigram: The trigram model looks back two words.

To compute a particular trigram probability of a word y given prior words x, z, we'll compute the count of the trigram C(x,z,y) and normalise by the total of all the trigrams that share the same words x and z, as shown below:

Text

Description automatically generated

We are training our model for Unigram, Bigram and Trigram using the specified methods in our class to predict the required probability of words. It's important to add markers at the start and the end of sentences in any N-gram model. Using our n-gram model, we have a way for creating a random piece of text. As a result, the overall probability of the entire language equals one. In the count of word tokens, however, all computations should include the end markers but not the start markers. Language model probabilities are always represented and computed as log probabilities. Because probabilities are (by definition) less than or equal to 1, the result shrinks as we combine more probabilities together. We utilise the log probabilities instead of the raw probabilities since multiplying enough n-grams together would result in numerical underflow. Extrinsic evaluation and Intrinsic evaluation are two ways of evaluating and comparing language models. We'll be assessing models inherently since it's a quick and efficient approach to do so. We'll use a measure called Perplexity, which is an intrinsic technique of evaluation. The performance of our model on certain test data will be used to assess its quality. The inverse probability of the test set, normalised by the number of words, is the perplexity of a language model on a test set. As a result, the lower the perplexity, the greater the conditional probability of the word sequence, thus maximising perplexity is equivalent to maximising the test set probability according to the language model. You'll have a finite vocabulary of words that you can manage unless you employ a character-based language model in some method, therefore you'll need a mechanism to handle out-of-vocabulary terms. One typical solution is to replace all out-of-vocabulary words with the unknown token, a pseudo-word that substitutes all out-of-vocabulary words. The unknown token has a number of benefits: Infrequent terms in the training corpus are unlikely to be learnt by the neural network or utilised in sentence generation. Ignoring them will result in a significant reduction in model size, as the embedding layer and SoftMax both need a large number of parameters. So we are counting the less probability word and labelling them as UNK. We have a method for creating unknown words from which the user may calculate the likelihood of occurring words. This method considers the specified number of words as unknown if they appear at least 2 or 3 times. Then there's the issue of zero probability words, which don't appear in the training set and are brand new. Then there's the issue.

It's likely that certain word sequences that were never observed during training appear in test data. When this happens, the sequence's probability equals zero. The perplexity metric grows limitless, making evaluation impossible. Giving non-zero counts to N-grams that are encountered in testing but not in training is a common solution. Because the entire probability distribution will not be normalised, we can't merely add 1 to all the zero counts. Instead, we subtract some probability mass from non-zero counts and add it to the zero counts (a process known as discounting). Smoothing is the term for the entire procedure.

Word Embedding:

For natural language processing (NLP) tasks, Word2Vec is a popular approach. Distributed Gensim is a Python topic modelling package that gives you access to Word2Vec and other word embedding methods for training, as well as the ability to load pre-trained word embeddings from the internet.embeddings for new state-of-the-art (SotA) deep neural networks provide a similar source of inspiration. The improper mix of hyper-parameters, on the other hand, might result in low-quality vectors. The goal of this study is to prove that there is an ideal combination of hyper-parameters and to assess alternative combinations. Named entity recognition (NER) and sentiment analysis (SA) were used in the intrinsic and extrinsic (downstream) assessments. The downstream tasks demonstrate that the optimal model is typically task-specific, that high analogy scores don't always correlate favourably with F1 scores, and that the focus on data alone doesn't always correlate positively with F1. After a point, increasing the vector dimension size results in poor quality or performance. When ethical concerns for time, energy, and the environment are taken into account, fairly smaller corporations may perform just as well, if not better, in some instances.

As we already seen the definition of word embedding, a pre-trained model is just a file with tokens and their corresponding word vectors. The pre-trained Google word2vec model was built using 300-dimensional word vectors and was trained on Google news data (about 100 billion words). It comprises 3 million words and phrases. In the most similar() function on the trained or loaded model, Genism provides an interface for executing these sorts of actions. The quality of a word2vec model may be considerably influenced by the use of different model parameters and corpus sizes. A variety of factors can increase accuracy, including the model architecture (CBOW or Skip-Gram), the size of the training data set, the number of vector dimensions, and the window size of words processed by the algorithm. Each of these enhancements comes at the expense of more computing complexity and, as a result, longer model generation times. We utilise Word2Vec for word embedding, but instead of using the mean of word embeddings, which assigns weights to each word in the sentence even if any of them is unimportant for semantic similarity, we use a weighted average. Every word embedding is weighted by a/(a + p(w)), where an is a parameter that is commonly set to 0.001 and p(w) represents the word's estimated frequency in a corpus.

I

Evaluating the model:

On the decreased data set, the 3-gram model was also trained. The 3-gram model, on the other hand, only got 23.5 percent of the time right, which is worse than just guessing at one of the five options. It was able to discover some patterns in the significantly reduced corpus by just counting the frequency of 3-grams. Ensemble approaches are models that are created in multiples and then combined to get better results. In most cases, ensemble approaches provide more accurate results than a single model. We have many different type of similarity measures, here in this experiment we use the cosine similarity as a default similarity method. We have got the similarity of almost 20 percent from the correct answers.

While error analysis is an excellent starting point, it can also overlook essential aspects of learner language. To begin with, concentrating solely on mistakes may cause you to overlook instances where the learner utilises the form correctly. For example, you may note that a student makes mistakes while articulating a TL sound before consonants but not when the sound is appropriately produced before vowels. The second item that a mistake analysis overlooks is prevention.

While the learnt model beats the baseline in many circumstances, it still falls short when the sentence heuristic is inaccurate. Our most recent model was successful in detecting high-signal words. However, it's extremely possible that if we use this model, we'll come across terms we haven't seen before in our training set. Even if it has seen extremely similar terms during training, the prior model will be unable to appropriately categorise these.

The accuracy level of unigram and bigram in the language model accuracy level of bigram is better than unigram. And the accuracy level of bigram and trigram is almost similar according to our experiment, as demonstrated in the graph below.

Chart, line chart

Description automatically generated

Chart, bar chart

Description automatically generated

Conclusion:

In our all experiment I have got the very less accuracy of the models, we can increase the accuracy by increasing the data set and get the average accuracy for both the models which I have used. But in the Language model we face the math domain error because of the probability zero cases. In my experiment both the models give almost similar accuracy it’s because of the size of data. In most cases, the greater the N, the better the model. However, this results in a lot of calculation overhead, which necessitates a lot of RAM. N-grams are a type of linguistic representation that is very sparse. All words not found in the training corpus will be given a probability of zero

Traditional embedding approaches like Word2Vec, as a general rule, produce good results when the job just requires the text's global meaning. This is demonstrated in their superior performance on tasks like semantic text similarity and paraphrase identification when compared to state-of-the-art deep learning algorithms. More elaborate contextual approaches, on the other hand, perform better when the job requires something more particular than just the global meaning, such as sentiment analysis or sequence labelling.

The CBOW or Skip-gram approaches can be used to train Word2vec vectors. Because the models are simpler, they may be trained on large corpora and with higher dimensional vectors, resulting in a more continuous representation of word vectors. The use of word2vec models in NLP tasks has a wide range of applications. When creating neural network models, the vectors operate as characteristics of the words. Word2vec may also be used to handle problems with word similarity and out-of-the-list words.