LATENT SEMANTIC ANALYSIS

1. Introduction

One of the big goals in the field of natural language processing is attempting to discern ideas from a corpus (body) of text. For example, the phrase "Tom and I are good friends" and "Tom and I are close companions" share the same idea - namely that Tom and I share a deep friendship. However, the two phrases use different words, "close companions" versus "good friends," in order to convey the same idea. Our task in creating an NLP model is to extract the underlying topic in a body of text despite whatever different phrasing might be used. This idea might seem intractable at first, but as we'll soon see, we can use techniques very similar to Principal Component Analysis (PCA) to reduce bodies of text to their underlying ideas. There are two big ideas in NLP that attempt to perform this task, Word2Vec and Latent Semantic Analysis and we will cover both of them.

2. Word2Vec

Word2Vec is more an algorithm than it is a technique, but the big idea is that words can be seen as weighted combinations of other words. For example, one might say a "male queen" is a "king." In mathematical terms one might assign a weight of 1 to both "male" and "queen" such that "male+queen-woman=king". Then if we heard "male" and "queen" without "woman" next to each other, we would know that the word that they were representing is "king". One can see how this allows us easy applications to software like search engines, where a person might not type exactly a phrase than exists within the relevant query result, but it still shows up because the engine is able to discern the underlying meaning of the phrase. Not only this, we might be concerned with how to map a complex vocabulary of words into a lower-dimensional subspace, in order to perform computations on said words much more efficiently.

So the question remains as to how we should approach creating such a list? Well the **distributional hypothesis** tells us that similar words will appear in similar distributions across a large body of text. That is to say, they will generally appear near each other and in similar proportions. Using this hypothesis, we should believe that for some word at position i in a text, the surrounding k words for some constant k are the most relevant contextual clues. For example, By weighting each of the words, we get

$$w[i] = w[i-k] + \cdots + w[i-1] + w[i+1] + \cdots + w[i+k]$$

This model is known as the **skip-gram** model. So now given a word at i, we have a mathematical form for how we might want to predict the words that surrounding it. Specifically, we see that this looks like a linear combination, so we are inclined to think of this as matrix multiplication. Suppose we have the one-hot encoding of a word at position i, x_i, multiply it by some $n \times m$ matrix W, then multiply it again by some W' to go back from an $m \times 1$ vector to an $n \times 1$ vector. However, training such a set of W, W' is exactly a task suited for a neural network! We just have a hidden layer of size m and the weights from the input to

Date: December 11, 2020.

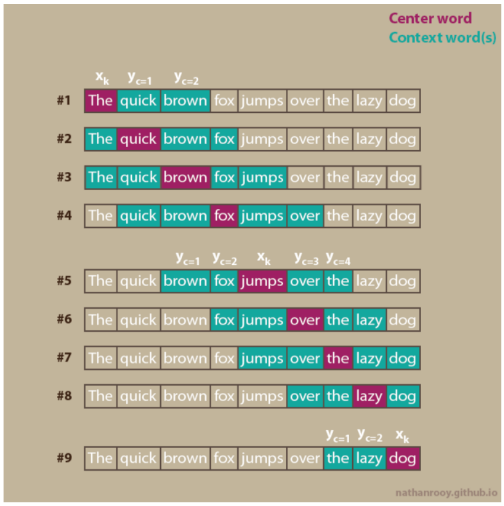

Figure 1. Using 2 context words

the hidden layer represent a $m \times n$ matrix, i.e. W, and the weights from the hidden layer to the output represent a $n \times m$ matrix, i.e. W'. Once we have our output, we then normalize this to a probability vector by applying a softmax function, meaning the vector will have ℓ_1 norm equal to 1. From here it should be create our error function by just having the correct label be a $2k$-sparse vector with $\frac{1}{2k}$ at the indices corresponding to the correct context words.

One might be wondering how such a neural net that predicts context words will allow us to generate a word embedding. Well this is the elegance of word2vec. Obviously there is no supervised way to directly generate a word embedding because we do not have labels for what a proper linear combination of words should look like. Therefore, the task of predicting context words acts almost like a fake task, where we do not really care about the output, so much as the hidden layer. The trick lies in the fact that the input is a one-hot encoding, which means Wx_i basically just looks up the i-th column of W. This first matrix W is our

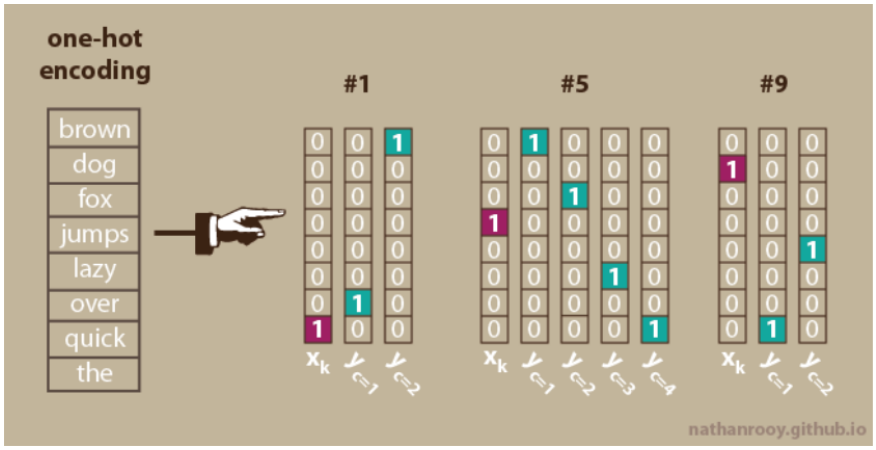

Figure 2. One-hot encoding

matrix that will perform low-dimensional embeddings.

A close relative of the skip-gram idea is the **continuous bag of words** method. The fake task that we set up for ourselves is taking $2k$ surrounding inputs of some word and trying to

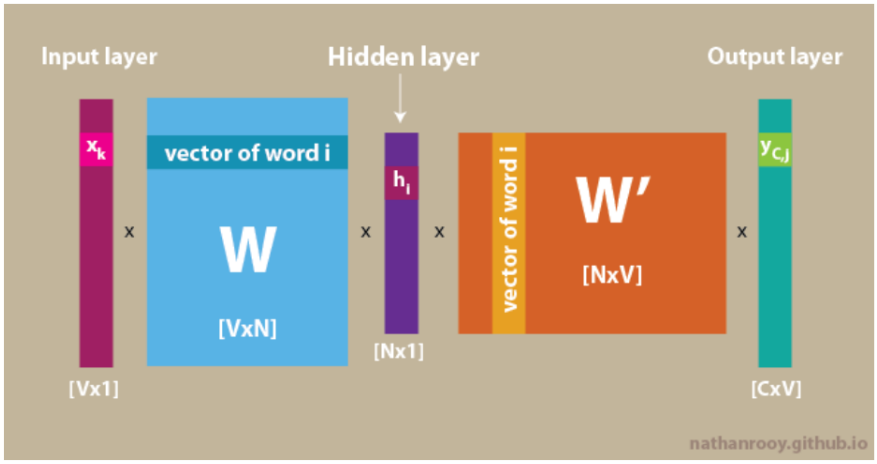

Figure 3

guess the missing word. We will similarly be able to use the matrix corresponding to weights in our neural net as a lookup table and generate low-dimensional embeddings for words.

3. Latent Semantic Analysis

How about when we are not concerned with finding an underlying idea by treating it as a linear combination of other words/categories. Suppose we just want to create a clustering algorithm that will group words based on some underlying idea. **Latent Semantic** Analysis definitionally concerns itself with finding the underlying (read: latent) logic (read: semantic analysis) that might generate some sentence/paragraph/document. Specifically, we will leverage some ideas from PCA to perform dimensionality reduction from the relatively large corpus of words, to a much smaller group of ideas.

As a quick motivating example, suppose we are working for a financial firm and are interested in having a computer trove through large amounts of financial news. Clearly this is a task that requires an immense amount of computational power, so we want to reduce the complexity of such a task. Specifically, we might be reading two different articles, with one saying in many words "the SP500 has gone up 3% on 5/17/2019" and the other (also in many words) saying "The SP500 has gone down 2% on 5/18/2001". Clearly there are three relevant parts of both articles: the SP500, the percentage, and the date. Everything else is irrelevant for our considerations. Then if we can somehow compress the article into these three relevant data points, we can easily compare the data in these articles to the data in other articles whereas otherwise it would be much harder.

So how will we leverage the techniques in PCA? Well we need to start by generating some matrix on which to perform our low-rank-approximation. A natural way to do this is by creating a **term-document matrix** D. Then $D_{i,j}$ is the number of times in document i that the word corresponding to index j is used. Then we calculate the Singular Value Decomposition of D, to be $U\Sigma V^T$ as is standard. Then U represents a matrix with the rows representing the words and the topic representing columns. Σ is then a matrix where $\Sigma_{i,i}$ represents the importance of topic i numerically. Lastly, V^T has topics representing the words and the documents representing the columns. In this manner U is a word to topic matrix, Σ is a topic importance matrix, and V^T is a topic prevalence matrix amongst our documents.

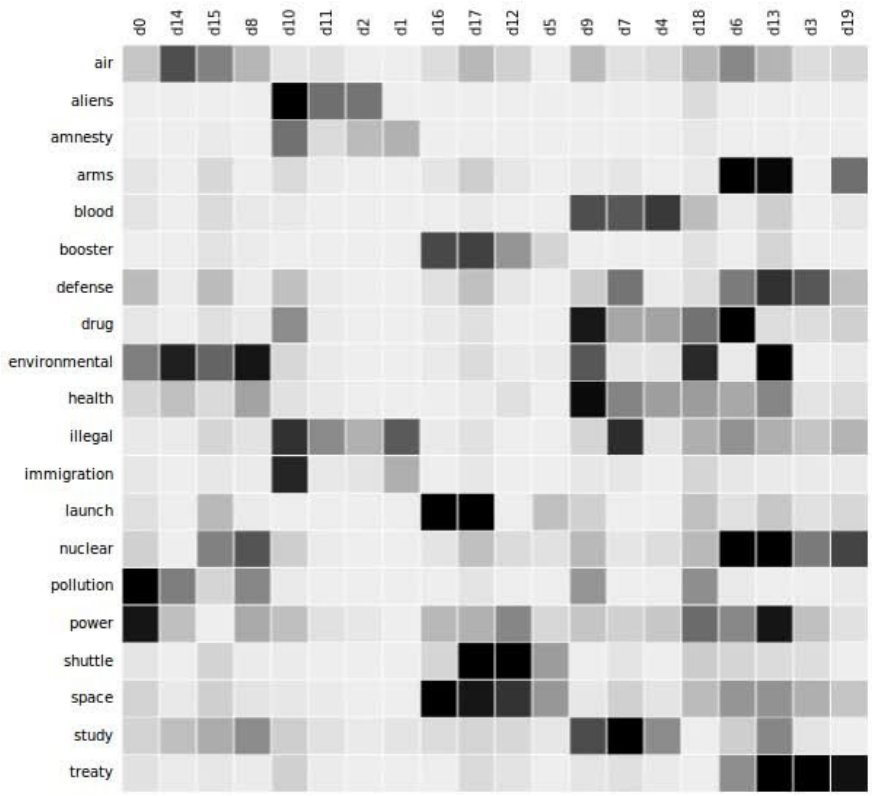

Figure 4. Term-Document Matrix

Now we can use our standard tricks from PCA to generate a low rank approximation. Suppose we want k underlying topics, then we take

$$D = U\Sigma V^T = \sum_{i=1}^{n} \sigma_i u_i v_i^T$$

which means we can truncate our summation to get a rank k approximation of D to be

$$\hat{D} = \sum_{i=1}^{k} \sigma_i u_i v_i^T$$

And that's it! We can look at each of the basis vectors $(x_1, \cdots, x_k)$ for $\hat{D} = \sum_{i=1}^{k} a_i x_i$ and interpret them as representing some underlying topic. For example we might have a basis that corresponds to 0.3*happy-0.4*sad which might be a vector corresponding to the idea of happiness. One thing to note is that the basis vectors might not be easily interpretable, i.e. we might have 0.3*happy+0.4*bottle as a basis vector which cannot easily be deciphered.