Text document encodings:

Bag of words (BoW) encoding

BoW(w,d)=(number of times word w occurs in d)

For example we have a document:

D1: A support vector machine is the best linear classifier. For non-linear we want to use the support vector machine again but in a transformed feature space.

D2: The linear regression classifier is probably the first classification ever introduced but the support vector machine outperforms it almost all the time on real data. The reason for this is that it is less sensitive to outliers than linear regression.

D3: Speaking of outliers the 01 loss is least sensitive and can handle outliers better than any linear classifier. But the 01 loss is very hard to optimize.

We can make a word vector that counts the number of occurrences of words in documents.

Word vector = (support, vector, machine, linear, non-linear, classifier, loss, optimize, outliers)

BoW(w,D1) = (2, 2, 2, 1, 1, 1, 0, 0, 0)
BoW(w,D2) = (1, 1, 1, 2, 0, 1, 0, 0, 1)

And in this way we can also do BoW(w,D3). Now we can compare documents and use their vector form to classify them as well.

Term frequency (TF) encoding

Here we normalize the BoW given above to account for different numbers of words across documents.

tf(w,d)=(number of times word w occurs in d)/(total words in d)

Inverse document frequency (IDF) encoding

Words that occur frequently in all documents may not be so helpful in classification. We downweight such words and upweight less frequent words with the idf encoding below.

idf(w,D)=log((number of documents in D)/(number of documents in D that contain the word w))

TF.IDF encoding

tf(w,d,D)=tf(w,d)*idf(w,D)

Regular expressions

Sometimes we want to find text patterns in the data. For example searching for specific words in a string. This can be done with string matching (such as .find() method) but we won’t be able to find patterns with this. For patterns we use regular expressions. We have special keywords and commands to find specific patterns. For example:

\s single white space
\s+ one of more consecutive single white space
\d any decimal character
[ACGT] any character in the string
. any character
\w any alphanumeric character
\D any non-decimal character
\W any non-alphanumeric character

We can search for patterns, search and replace patterns, and split a string by pattern.

Word tagging and nltk

Nltk is the Python natural language toolkit.

Word encodings and sequence to sequence models:

Above we have seen methods to encode an entire document or a paragraph or even a sentence. That allows us to classify and compare documents, paras, and sentences. For example in a search engine (like Google) our query becomes a document vector that we then compare against document vectors of internet web pages.

Now we look at word encodings. Let us discuss this for the problem of context prediction.

Word prediction

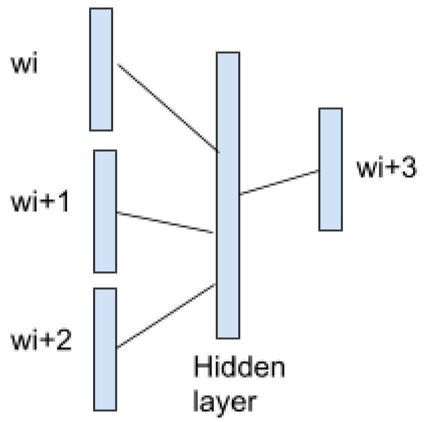
Given a set of words in a sentence can we predict the next word? In order to solve this problem, we need a model that takes context into consideration. A simple context model would be to predict the fifth word from the first four. For example a model would be given “The cat ate the” and the prediction is “mouse”. To build such a model we need a training dataset (xi,yi) where each xi are four words of a sentence and the yi is the fifth predicted word.

How do we encode the words so we can perform machine learning? Three choices:

1. One hot encoding: high dimensional vectors where each dimension corresponds to a word
2. Label encoding: each word maps to a unique number
3. Word2vec: Use a neural network to the word from a previous one. Use the hidden layer to represent the input word.

We can see from above that the input and output to the model are entire vectors. This brings us to sequence to sequence models.

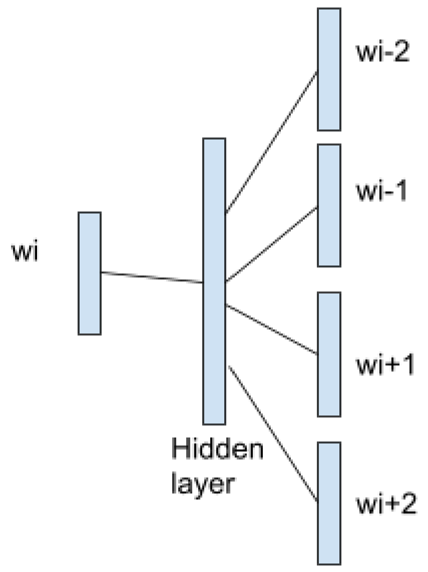
In the example shown below we see a model designed to predict the fourth word given a sequence of three preceding words. We can choose more than one hidden layer if it works better.



Word2Vec

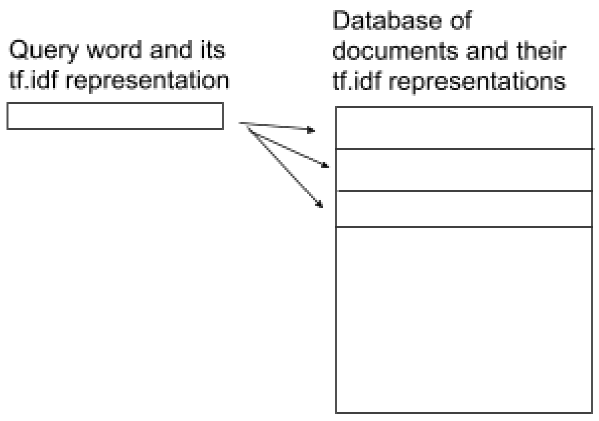
So far we have looked at methods to encode an entire document or a set of words into a vector. These are based upon basic statistics of counting frequency of words in the document. We don’t consider context or meaning in creating the document vectors. In fact we can even represent documents with single words. They would just be a one-hot vector.

In Word2Vec we try to create more meaningful representations of words. We achieve this with a network that maps a word to its neighboring words. In the network below we represent each word with its one hot encoded vector. After training the network below the hidden gives us the vector representation for word wi. The hidden layer is typically 300 nodes as seen in previous studies.



Searching text documents:

Text document search is common in many places. For example searching the web with a search engine. Or searching legal, medical, and tax documents for example. Typically we perform search by encoding documents and words into vectors with tf.idf representations. We then encode the query also as a tf.idf vector and perform cosine similarity against the target database.


Query word and its tf.idf representation
Database of documents and their tf.idf representations

Alternatively we can search documents with word2vec learned representations. Here we would use word similarity. For example suppose we have a query word w and its word2vec representation is v. We then look for the most similar word vector v1, v2, … vi that we have learned from some dataset. This dataset can be domain specific. For example if we want to build a legal document search program then we would learn word2vec on a set of legal documents.

After finding the top similar words we would identify their location in the document and output the sentence or paragraph that they occur in. In this method we are searching for documents with vectors that encode context and meaning whereas in tf.idf it is based mostly upon frequency of words in documents. The order of words and their context is not captured in tf.idf representations.