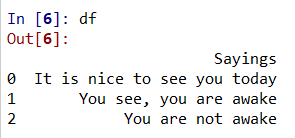
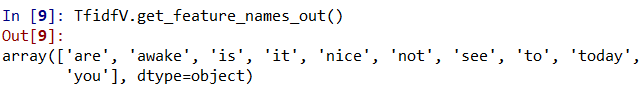
**Word Vectorizing, Embedding**

**Word Vectorizing**

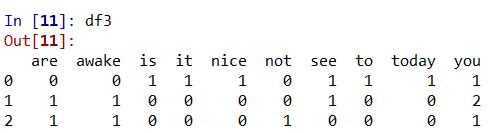
Sometimes, when doing Natural Language Processing for example, we need to encode phrases, sentences, or documents, etc. One way to do this is with sklearn’s Term Frequency Inverse Document Frequency Vectorizer. So say you have a column of ‘sayings’, Xi. I’ll use ‘sayings’ and ‘documents’ interchangeably.



We would like to encode these sayings as numbers, as vectors really. So we run through the entire list of sayings and pick out all the unique words. There are 10.



Then we could in principle treat these words as ‘basis vectors’ from which to construct our sayings. Our sayings would be:

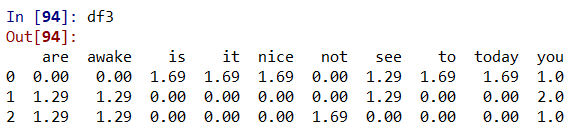


where each entry is the so-called Term Frequency (TF) = f(t) of the respective word in the saying/document. But TFIDF does things a little differently. It multiplies the term frequency by an Inverse Document Frequency (IDF) value which lowers the weight of the terms that appear frequently in all documents.

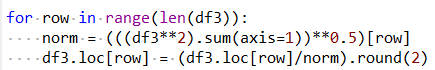


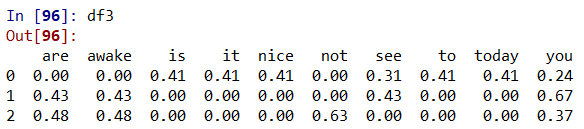
nw(t) is the number of words in the document that are equal to the term. nd is the number of documents/saying. In my example this is nd = 3. And nd(t) is the number of documents that contain that term. We can see that if nd(t) = nd, then IDF(t) = 1, the smallest value. If nd(t) = 1, then we get IDF(t) = 1 + ln((nd+1)/2), the largest value. So this word would be weighted heavily because it is unique to that document and thus can serve to identify it. Well if we do this, then we get:



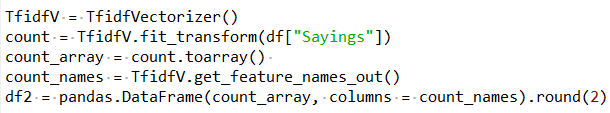


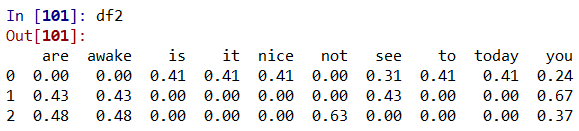
Often, we go one step further, however. We normalize the rows to unit magnitude in the usual L2 sense. So we have now,





This is what sci-kit learn does, as can see,





TFIDF is the basic word-vectorizer I think, and it apparently works pretty well. One of its drawbacks is that it treats all words as being essentially orthogonal to one another. But words like, ‘fast’ and ‘rapid’ have similar meanings, and could/should be treated synonymously. Word Embeddings using Neural Networks can do this.

**Word Embedding**

So as we saw, one way to process words, one thing we can do is basically one-hot-encoding for all O(104) words in a vocabulary. But training on a dataframe with O(104) columns would be crazy. Our ML model would have O(104) inputs. And this would ignore the fact that words are not entirely unrelated / independent. Synonyms should be more closely related than antoynyms. Words of the same part of speech are more closely related than words of different parts of speech. So instead of treating the words as O(104) dimensional independent vectors, which is what one-hot-encoding basically does, it makes more sense to treat them as O(1) or O(10), etc. dimensional vectors (which has the consequence that words can be near or far from other words, and that words can be linear combinations of other words). Treating words as being related like this proffers the additional advantage of making the ML model automatically work for synonyms of words it already works for.

Now we could try to do this number assignment by hand. For instance we could go into the dictionary and assign all nouns one set of numbers, verbs another set, adjectives another set, etc. But there’s a better way, using Neural Networks (see notes way down below). We feed, say, a book’s worth of sentences into a dataframe, and we extract all the, say, N = 5 for illustration’s sake, different words. And we also work out the most probable word to follow each of these N words. Then we create a dataframe: X, y. Xi would be an N dimensional one-hot-encoded table delineating all the N different words, and yi would be an N dimensional one-hot-encoded table specifying the most probable word to follow after each Xi. Then we feed the data into a neural network. Let’s say we’re going to represent a word as a single number (so our vector space is d = 1). Then we’d use a hidden layer with just one node.

A diagram of a structure

Description automatically generated

The input is an N-dimensional column vector representing a given word in the training data. So for instance, word\_1 would be represented as (1 0 0 0 0)T, and word\_3 as (0 0 1 0 0)T, etc. These inputs are connected to the network via weights of course. These input weights will ultimately be the numbers we associate with the words. It seems there is typically no bias applied to the input? The activation node seems to typically consists of just a linear activation function A(x) = x. Not sure. And then there are weights and biases along each of the output lines. We ultimately want the output to be probabilities. So we’ll have to do that thing where we map each of the values fi → ef\_i/Σief\_i, via the softmax layer. Okay then we train it, using the cross-entropy loss function, to take in a word vector, say word\_2 = (0 1 0 0 0), and spit out a vector of probabilities on the other side, with the word that most likely follows word\_2 being the most probable. For instance, if this is word\_3, then we should get something like:

A diagram of a network

Description automatically generated

Another variant on this is to set the inputs to be a given pair of words, and the output to be the most likely word to appear between them. This is called **Continuous Bag of Words**. So if word\_1 is the most likely word to appear between word\_4 and word\_5, then we should get something like,

A diagram of a diagram

Description automatically generated

And another option is **Skip Gram**, which is basically the reverse. We set the input to be a given word, and the output to be the most likely words on either side. So if the most likely words on either side of word\_5 are word\_1 and word\_3, then should get something like,

A diagram of a diagram

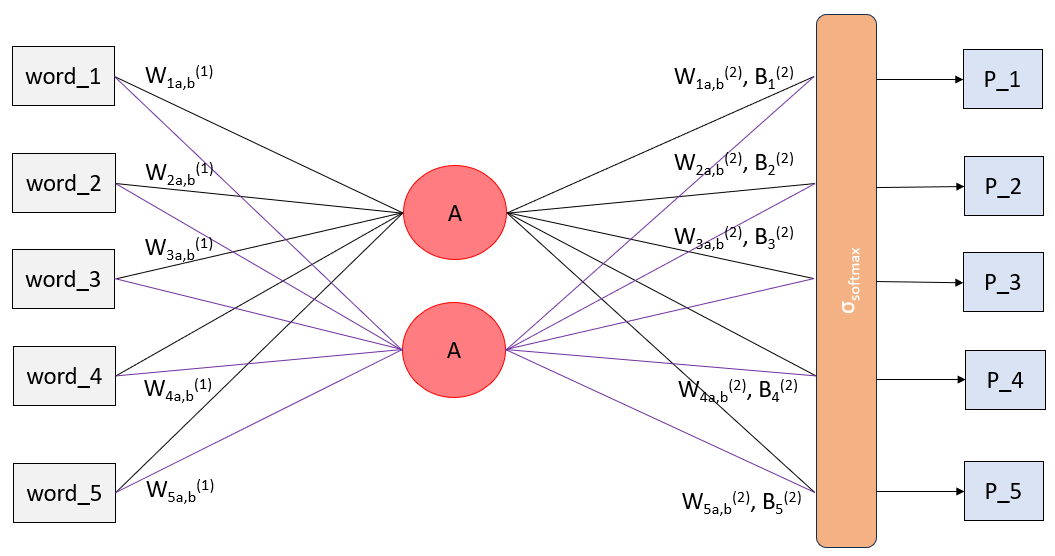
Description automatically generated

In this case, do we use the following loss function:



which is capable of handling multiple target probabilities? Or do we use regression?

As discussed above, it’s possible that we want to assign multiple values to our words, i.e., a vector to our words. I think if we assign, say, two values to a word, we have to assign two values to all of them. Let’s say we want to assign our words to a 2 dimensional vector (so vector space is d = 2). If we were to do so, then we’d use a hidden layer with two nodes. Again the weights flowing out of any word would constitute the vector we associate with it. And so for instance W3a,b(1) would be the two-element vector (W3a(1), W3b(1))T describing word\_3.



And we could do similarly if we wanted to represent our words as a 3 dimensional or higher vector. We’d just run our words through 3 activation nodes. I think d = 50 dimensions are often used. Representing as a vector allows greater subtlety in representing words. So when words are input into a neural network, I think it would look like this:

A diagram of a brain function

Description automatically generated

On the left is all the words in our vocubulary. And all the pretrained weights attached/leading to the d (=2 in this case) linear activation functions. And then we’d have whatever arbitrary weights (to be trained as part of the normal training process) going from the linear activation functions to the neural network. Specifically, if we wanted to input, say, word 2, then we’d do,

A diagram of a network

Description automatically generated

This way, only the two weights W2a,b(1) will get sent to the activation functions. W2a(1) will go to the top one, and W2b(1) will go to the bottom one. So observe how we will then have effectively replaced a five dimensional neural network vector input [in this case (0 1 0 0 0)] with a two dimensional vector input [in this case (W2a(1) W2b(1))]. And in practical applications, when we represent words with a d-dimensional vector, we’ll have replaced an O(104) dimensional vector input with a d-dimensional one. For instance, without the word embedding layer, the neural network input would be:

A green object with black lines

Description automatically generated

(wrote superscript 2 just to make it obvious which weights we should be comparing to in the previous illustration) (and I guess I should have biases here too, as well as in layer 2 of the previous illustration)

**Cosine Similarity**

We can write a metric for how similar two phrases are. Consider:

P1: Hello World.

P2: Hello World, Hello.

To start, we would represent P1 and P2 as vectors. The ‘unit axes’ are the words in the phrases. And the component of the phrase along that axis is the number of times the that word appears. We can draw a table to illustrate.

|  |  |  |
| --- | --- | --- |
|  | Hello | World |
| P1 | 1 | 1 |
| P2 | 2 | 1 |

Leaning more heavily on vector parlance, we could write,



Anyway, our metric for similarity will just be the angle between the two vectors. So,



**Cleaning Textual Data**

Words in a text are often turned into basis vectors. But this results in a lot of orthogonal vectors which should be basically the same. There are a few things we can do to clean this up.

1. remove punctuation. Can do with by changing all non alpha-numeric characters to empty space via. series.str.replace(“r[regex]”, “”, regex = True).
2. then should change all to lower case since, upper case letters/words mean the same as lower case letters/words and shouldn’t be treated differently. series.str.lower().
3. might want to change all plural nouns to singular.
4. also remove ‘stop’ words (conjunctions I guess) like ‘and’, ‘so’, etc. These don’t convey a lot of information, and so are just taking up space.
5. You can try stemming words: running 🡪 run, faster 🡪 fast. This reduces related words to a single word. But this also has problems in that applied uncritically it might reduce the word to a non-word, or unrelated word: better 🡪 bet.
6. Another option is lemmatization. This reduces similar words to a single related word: running 🡪 run, better 🡪 correct.
7. then you might use the series.str.split() method to split each text into a list of words.
8. then could feed into sklearn’s CountVectorizer method CV = CountVectorizer(), or TFIDF to create a text vector. Can adjust min\_df, max\_df to specify the a lower and upper bound for what the word frequency should be to be considered useful enough to be encoded.

I think another possibility (more common) is to use a word-vectorizer program like GenSim, which represents words on a lower-dimensional space and accounts for their similarities.