01-Word-Embeddings

November 20, 2017

```
In [1]: import matplotlib.pyplot as plt
        %matplotlib inline
        plt.style.use('ggplot')
        from datascience import *
        import numpy as np
        from scipy.spatial.distance import cosine
        import gensim
        import nltk
        from string import punctuation
```

```
Using TensorFlow backend.
```

1 Word Embedding

This lesson is designed to explore features of word embeddings described by Ben Schmidt in his blog post "Rejecting the Gender Binary".

The primary corpus we use consists of the 150 English-language novels made available by the .txtLab at McGill. We also look at a Word2Vec model trained on the ECCO-TCP corpus of 2,350 eighteenth-century literary texts made available by Ryan Heuser. (I have shortened the number of terms in the model by half in order to conserve memory.)

For background on Word2Vec's mechanics, I suggest this brief tutorial by Google, especially the sections "Motivation," "Skip-Gram Model," and "Visualizing."

We'll read in Andrew Piper's corpus we used in our Topic Modeling notebook:

```
In [2]: metadata_tb = Table.read_table('../09-Topic-Modeling/data/txtlab_Novel150_English.csv'

        fiction_path = '../09-Topic-Modeling/data/txtlab_Novel150_English/'

        novel_list = []

        # Iterate through filenames in metadata table
        for filename in metadata_tb['filename']:

            # Read in novel text as single string, make lowercase
            with open(fiction_path + filename, 'r') as file_in:
                novel = file_in.read()
            # Add novel text as single string to master list
            novel_list.append(novel)
```

---

2 Pre-Processing

Word2Vec learns about the relationships among words by observing them in context. We'll need to tokenize the words in our corpus while retaining sentence boundaries. Since novels were imported as single strings, we'll first use sent_tokenize to divide them into sentences, and second, we'll split each sentence into its own list of words.

We'll use `nltk`'s sentence tokenizer:

```
In [3]: from nltk.tokenize import sent_tokenize
```

Due to memory and time constraints we'll use our quick and dirty tokenizer:

```
In [4]: def fast_tokenize(text):

            # Iterate through text removing punctuation characters
            no_punct = "".join([char for char in text if char not in punctuation])

            # Split text over whitespace into list of words
            tokens = no_punct.split()

            return tokens
```

First get the sentences:

```
In [5]: sentences = [sentence for novel in novel_list for sentence in sent_tokenize(novel)]
```

Now the words:

```
In [6]: words_by_sentence = [fast_tokenize(sentence.lower()) for sentence in sentences]
```

We'll double check that we don't have any empty sentences:

```
In [7]: words_by_sentence = [sentence for sentence in words_by_sentence if sentence != []]
```

We should now have a `list` of `lists` with sentences and words:

```
In [8]: words_by_sentence[:2]

Out[8]: [['authors',
          'introduction',
          'my',
          'dog',
          'had',
          'made',
 'a',
 'point',
 'on',
 'a',
 'piece',
 'of',
 'fallowground',
 'and',
 'led',
 'the',
 'curate',
 'and',
 'me',
 'two',
 'or',
 'three',
 'hundred',
 'yards',
 'over',
 'that',
 'and',
 'some',
 'stubble',
 'adjoining',
 'in',
 'a',
 'breathless',
 'state',
 'of',
 'expectation',
 'on',
 'a',
 'burning',
 'first',
 'of',
 'september'],
['it',
 'was',
 'a',
 'false',
 'point',
 'and',
 'our',
 'labour',
 'was',
 'vain',
 'yet',
 'to',
'do',
'rover',
'justice',
'for',
'hes',
'an',
'excellent',
'dog',
'though',
'i',
'have',
'lost',
'his',
'pedigree',
'the',
'fault',
'was',
'none',
'of',
'his',
'the',
'birds',
'were',
'gone',
'the',
'curate',
'showed',
'me',
'the',
'spot',
'where',
'they',
'had',
'lain',
'basking',
'at',
'the',
'root',
'of',
'an',
'old',
'hedge']]
```

---

3 Word2Vec

3.0.1 Word Embeddings

Word2Vec is the most prominent word embedding algorithm. Word embedding generally attempts to identify semantic relationships between words by observing them in context.

Imagine that each word in a novel has its meaning determined by the ones that surround it in a limited window. For example, in Moby Dick's first sentence, "me" is paired on either side by "Call" and "Ishmael." After observing the windows around every word in the novel (or many novels), the computer will notice a pattern in which "me" falls between similar pairs of words to "her," "him," or "them." Of course, the computer had gone through a similar process over the words "Call" and "Ishmael," for which "me" is reciprocally part of their contexts. This chaining of signifiers to one another mirrors some of humanists' most sophisticated interpretative frameworks of language.

The two main flavors of Word2Vec are CBOW (Continuous Bag of Words) and Skip-Gram, which can be distinguished partly by their input and output during training. Skip-Gram takes a word of interest as its input (e.g. "me") and tries to learn how to predict its context words ("Call","Ishmael"). CBOW does the opposite, taking the context words ("Call","Ishmael") as a single input and tries to predict the word of interest ("me").

In general, CBOW is is faster and does well with frequent words, while Skip-Gram potentially represents rare words better.

3.0.2 Word2Vec Features

`size`: Number of dimensions for word embedding model
`window`: Number of context words to observe in each direction
`min_count`: Minimum frequency for words included in model
`sg` (Skip-Gram): '0' indicates CBOW model; '1' indicates Skip-Gram
`alpha`: Learning rate (initial); prevents model from over-correcting, enables finer tuning
`iterations`: Number of passes through dataset
`batch_words`: Number of words to sample from data during each pass
Note: cell below uses default value for each argument

3.1 Training

We've gotten accustomed to training powerful models in Python with one line of code, why stop now?

```
In [9]: model = gensim.models.Word2Vec(words_by_sentence, size=100, window=5, \
                                       min_count=5, sg=0, alpha=0.025, iter=5, batch_words=100
```

3.2 Embeddings

We can return the actual high-dimensional vector by simply indexing the model with the word as the key:

```
In [10]: model['whale']
```

```
Out[10]: array([-0.2095125 , -0.53014708,  2.68686366,  1.08366656, -0.40812227,
               -0.68002445,  1.1633867 ,  0.71124995, -0.38245955, -1.12927985,
                0.48150864,  1.28086412, -0.26883012,  0.51484323,  0.82005334,
               -0.82178885, -0.74048662,  0.48547563, -1.53553164,  0.91338807,
               -1.05285025, -0.77587211,  0.26850548, -0.89948153, -2.09574533,
               -1.8680023 ,  0.82816815,  0.06940582, -0.47608867,  1.08366251,
               -1.56296813,  0.57904035,  0.17262411,  2.03236961,  0.45003211,
                0.05243169,  0.55305344, -0.476307  ,  0.28638583, -0.33054   ,
                1.88581407,  0.18594833, -0.73652244,  1.84361017, -0.06023399,
               -1.48566055, -0.50887358, -1.25192571, -0.89620918, -0.2445336 ,
               -0.3286128 , -1.53668356, -0.47105476, -0.65326619,  0.74715048,
                1.39061999, -1.86718798,  0.76457733, -0.81341851, -0.84378022,
               -0.20838548, -0.96371949,  0.54132009,  1.53895736, -1.92489934,
               -0.60539699, -1.32342911, -0.23234639,  0.28665653, -1.2314707 ,
               -1.08295381,  0.23786834, -0.41666359, -0.18274072, -2.03216934,
               -2.32612777,  2.4315834 ,  0.5398944 , -0.62745667,  0.64861274,
               -0.06423267,  0.18305713, -0.01736131,  2.09066939, -2.09267831,
               -1.20219624,  2.63767886, -2.04860115, -0.47418815, -0.09252869,
                1.29749274, -0.06021469,  0.25439468, -1.55340648, -0.33380491,
                0.3910937 ,  1.08162844, -0.43912572, -0.50281495, -0.28908011], dtype=float3
```

`gensim` comes with some handy methods to analyze word relationships. `similarity` will give us a number from 0-1 based on how similar two words are. If this sounds like cosine similarity for words, you'd be right! It just takes the cosine similarity of these high dimensional vectors:

```
In [11]: model.similarity('sense','sensibility')
```

```
Out[11]: 0.67510716623808897
```

We can also find cosine distance between two clusters of word vectors. Each cluster is measured as the mean of its words:

```
In [12]: model.n_similarity(['sense','sensibility'],['whale','harpoon'])
```

```
Out[12]: 0.16773429058301756
```

We can find words that don't belong with `doesnt_match`. It finds the mean vector of the words in the `list`, and identifies the furthest away:

```
In [13]: model.doesnt_match(['pride','prejudice', 'harpoon'])
```

```
Out[13]: 'harpoon'
```

The most famous implementation of this vector math is semantics. What happens if we take:

$$King - Man + Woman =$$

```
In [14]: model.most_similar(positive=['woman', 'king'], negative=['man'])
```

```
Out[14]: [('princess', 0.760888934135437),
          ('queen', 0.7292253971099854),
          ('saint', 0.6846663355827332),
          ('priest', 0.666690468788147),
          ('duke', 0.6635293960571289),
          ('duchess', 0.6611680388450623),
          ('chevalier', 0.653782844543457),
          ('solomon', 0.6347128748893738),
          ('marquis', 0.6335281133651733),
          ('abbot', 0.6312479972839355)]
```

Schmidt looked at words associated with male and female pronouns to investigate gender. Let's try take all the female pronouns and subtracting the male pronouns:

```
In [15]: model.most_similar(positive=['she','her','hers','herself'], negative=['he','him','his
```

```
Out[15]: [('lovely', 0.5040115118026733),
          ('beautiful', 0.48882389068603516),
          ('beauty', 0.46054551005363464),
          ('miss', 0.45758432149887085),
          ('sweet', 0.4536876380443573),
          ('maiden', 0.4244999289512634),
          ('jane', 0.41977018117904663),
          ('anne', 0.41554996371269226),
          ('charms', 0.41379350423812866),
          ('girlish', 0.41347140073776245)]
```

And the opposite:

```
In [16]: model.most_similar(positive=['he','him','his','himself'], negative=['she','her','hers
```

```
Out[16]: [('horse', 0.4631490111351013),
          ('bill', 0.43658673763275146),
          ('strobik', 0.4301014244556427),
          ('moby', 0.42721283435821533),
          ('mate', 0.4215654134750366),
          ('mulligan', 0.4171442985534668),
          ('gun', 0.41092386841773987),
          ('steward', 0.409824013710022),
          ('captain', 0.4036237597465515),
          ('buck', 0.4029727578163147)]
```

How about together (*genderless* in Schmidt's sense)?

```
In [17]: model.most_similar(positive=['she','her','hers','herself','he','him','his','himself']
```

```
Out[17]: [('edgar', 0.5963000655174255),
          ('antonia', 0.5927170515060425),
          ('eugenia', 0.5885695219039917),
          ('them', 0.586503267288208),
          ('eleanor', 0.5812600255012512),
          ('camilla', 0.5807881951332092),
          ('itself', 0.5786113739013672),
          ('it', 0.5713871121406555),
          ('carrie', 0.555054247379303),
          ('illtemper', 0.554614782333374),
          ('elvira', 0.5455446243286133),
          ('margaret', 0.5434470176696777),
          ('algernon', 0.5411058068275452),
          ('cora', 0.5316364765167236),
          ('erica', 0.5300624370574951),
          ('leonora', 0.5258767008781433),
          ('me', 0.5244317054748535),
          ('kim', 0.522552490234375),
          ('rhoda', 0.5216361284255981),
          ('indiana', 0.5197480916976929),
          ('magua', 0.5186641216278076),
          ('adeline', 0.5177769660949707),
          ('rebecca', 0.5078281164169312),
          ('maggie', 0.5049735903739929),
          ('hippolitus', 0.5038728713989258),
          ('amy', 0.49874913692474365),
          ('heyward', 0.49584975838661194),
          ('latter', 0.4914630055427551),
          ('aileen', 0.4899047613143921),
          ('emily', 0.4894411563873291),
          ('elinor', 0.4894157946109772),
          ('bathsheba', 0.4884398579597473),
          ('valancourt', 0.4882795810699463),
          ('isabel', 0.48250851035118103),
          ('catherine', 0.481886088848114),
          ('edward', 0.4803752899169922),
          ('marianne', 0.4788520336151123),
          ('constraint', 0.4769141674041748),
          ('cecilia', 0.47649067640304565),
          ('myself', 0.47621768712997437),
          ('elizabeth', 0.47514021396636963),
          ('pleyel', 0.4746926426887512),
          ('vent', 0.4737328290939331),
          ('strether', 0.46890759468078613),
          ('glenmurray', 0.46876823902130127),
          ('lily', 0.46821171045303345),
          ('tess', 0.4661927819252014),
          ('clermont', 0.4655090868473053),
          ('anxiety', 0.46460625529289246),
          ('carwin', 0.46370792388916016)]
```

4 Homework

Use the `most_similar` method to find the tokens nearest to 'car' in our model. Do the same for 'motorcar'.

```
In [18]: model.most_similar(['car'])
```

```
Out[18]: [('coach', 0.832050621509552),
          ('wagon', 0.8295395970344543),
          ('cart', 0.8095526695251465),
          ('cab', 0.8068643808364868),
          ('hansom', 0.794998049736023),
          ('boat', 0.7902927994728088),
          ('chaise', 0.7885570526123047),
          ('carriage', 0.7708592414855957),
          ('buggy', 0.7703739404678345),
          ('vehicle', 0.769843339920044)]
```

```
In [19]: model.most_similar(['motorcar'])
```

```
Out[19]: [('majordomo', 0.7437359690666199),
          ('armourer', 0.7359848022460938),
          ('bagdad', 0.735150933265686),
          ('officeboy', 0.7277700304985046),
          ('poster', 0.7246631383895874),
          ('marcus', 0.7222787141799927),
          ('screwdriver', 0.7189463376998901),
          ('codfish', 0.7186081409454346),
          ('woodpile', 0.7178428173065186),
          ('grazier', 0.7153677344322205)]
```

What characterizes each word in our corpus? Does this make sense?

In the first corpus, our words are characterized by meaning. Each word has similar meaning to car. In the second corpus, our words are characterized by being conjuction words, or words made up of two other words.

Vector addition and subtraction can be thought of in terms of analogy. From the example above: 'man' is to 'king' as 'woman' is to '???'. Use the `most_similar` method to find: 'paris' is to 'france' as 'london' is to '???'

```
In [20]: model.most_similar(positive=['france', 'london'], negative=['paris'])
```

```
Out[20]: [('england', 0.7864130735397339),
          ('america', 0.7678409814834595),
          ('ireland', 0.7583222389221191),
          ('spain', 0.7514252662658691),
          ('scotland', 0.7499324083328247),
          ('germany', 0.7444795966148376),
          ('italy', 0.7273027896881104),
          ('india', 0.7217679023742676),
          ('europe', 0.7205799221992493),
          ('philadelphia', 0.6627640724182129)]
```

What has our model learned about nation-states?

Our model has learned that nation-states are similar to France!

Perform the canonic Word2Vec addition again but leave out a term. Try 'king' - 'man', 'woman' - 'man', 'woman' + 'king'.

```
In [21]: model.most_similar(positive=['king'], negative=['man'])

Out[21]: [('karl', 0.4981400966644287),
          ('casimir', 0.49626171588897705),
          ('kings', 0.49196910858154297),
          ('', 0.48961499333381653),
          ('hunters', 0.48672062158584595),
          ('ruritania', 0.4803735613822937),
          ('abraham', 0.4775375425815582),
          ('royal', 0.476759135723114),
          ('macadams', 0.47644904255867004),
          ('caligula', 0.4750182628631592)]

In [22]: model.most_similar(positive=['woman'], negative=['man'])

Out[22]: [('jane', 0.579742968082428),
          ('maiden', 0.5059601068496704),
          ('lovely', 0.48367786407470703),
          ('madeleine', 0.47847461700439453),
          ('bloofer', 0.47527503967285156),
          ('louisa', 0.4749048948287964),
          ('beautiful', 0.47352397441864014),
          ('baldock', 0.46967563033103943),
          ('maid', 0.46785688400268555),
          ('glencora', 0.4667515456676483)]

In [23]: model.most_similar(positive=['king', 'woman'])

Out[23]: [('priest', 0.8045262694358826),
          ('jew', 0.77991783618927),
          ('man', 0.7674248218536377),
          ('clergyman', 0.7513748407363892),
          ('poet', 0.7482427358627319),
          ('nobleman', 0.7457173466682434),
          ('frenchman', 0.7437664866447449),
          ('soldier', 0.7334054112434387),
          ('minister', 0.731255054473877),
          ('princess', 0.7222050428390503)]
```

What do these indicate semantically?

Semantically a lot of words are connected to the actual word 'man'. So when you subtract 'man' it tends to just take out the words with 'man' in them even if they are not connected to a gender.

4.1 Visualization

We can use multi-dimensional scaling to visualize this space just like we did with the documents before. But there are a lot of words here, so let's limit it to 50 words from our female gendered subset:

```
In [25]: her_tokens = [token for token,weight in model.most_similar(positive=['she','her','hers
                                                                    negative=['he','him','his','hi
```

We need to get the vector from each word, just like above, and add that to a list:

```
In [26]: vectors = [model[word] for word in her_tokens]
```

We can then calculate pairwise the cosine distance:

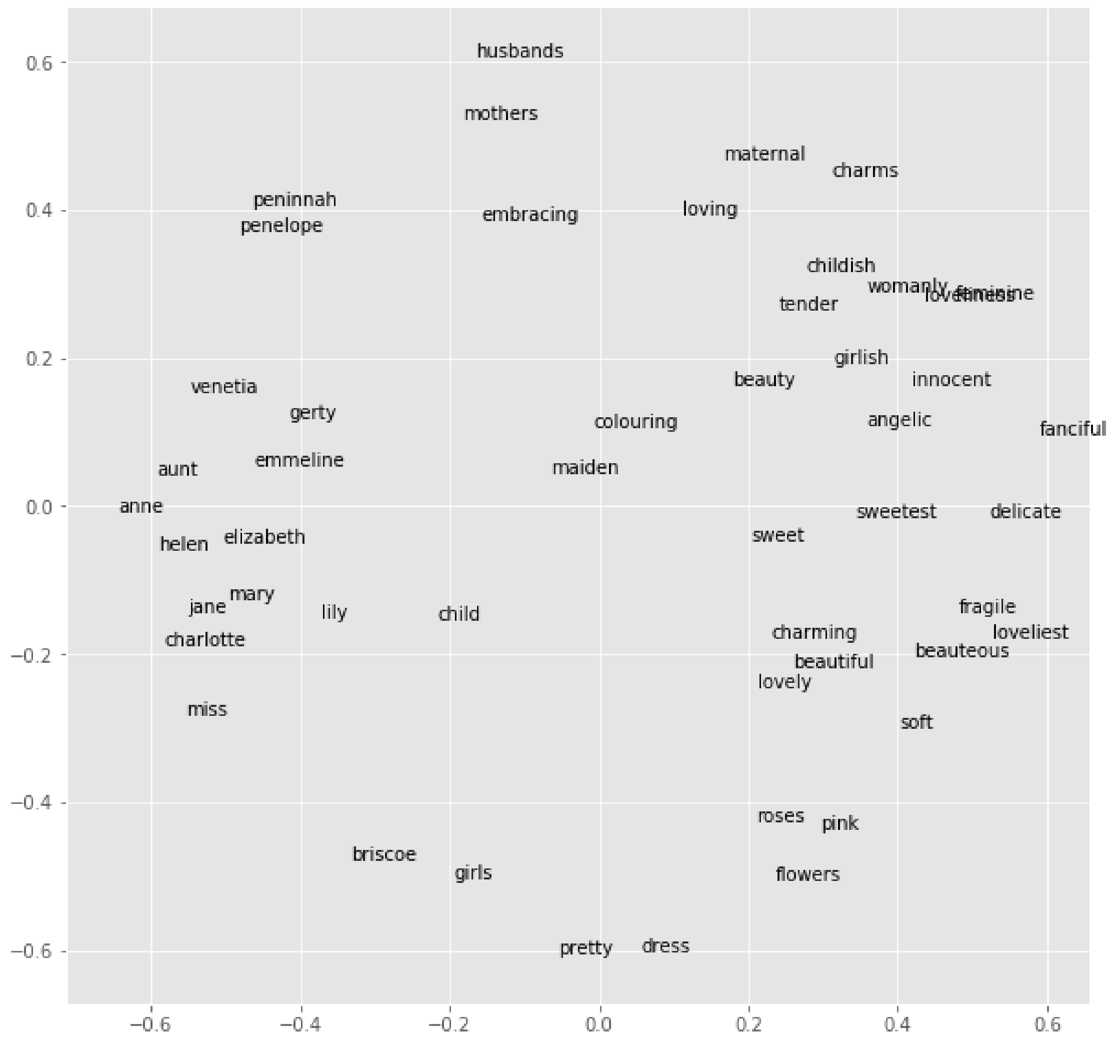
```
In [27]: from sklearn.metrics import pairwise
         dist_matrix = pairwise.pairwise_distances(vectors, metric='cosine')
```

We'll use `MDS` to reduce the dimensions to two:

```
In [28]: from sklearn.manifold import MDS
         mds = MDS(n_components = 2, dissimilarity='precomputed')
         embeddings = mds.fit_transform(dist_matrix)
```

Some fancy `matplotlib` code...

```
In [29]: _, ax = plt.subplots(figsize=(10,10))
         ax.scatter(embeddings[:,0], embeddings[:,1], alpha=0)
         for i in range(len(vectors)):
             ax.annotate(her_tokens[i], ((embeddings[i,0], embeddings[i,1])))
```



What kinds of semantic relationships exist in the diagram above? Are there any words that seem out of place? How do you think they go there?

Adjectives tend to be on the right while names are on the left!

---

5 Saving/Loading Models

We can save the model as a `.txt` file with the `save_word2vec_format` method:

```
In [55]: model.wv.save_word2vec_format('word2vec.txtalb_Novel150_English.txt')
```

To load up a model, we just ask `gensim`. Here's a model trained on Eighteenth Century Collections Online corpus (~2500 texts) made available by Ryan Heuser: http://ryanheuser.org/word-vectors-1/

```
In [56]: ecco_model = gensim.models.KeyedVectors.load_word2vec_format('data/word2vec.ECCO-TCP.
```

```
In [57]: ecco_model.most_similar(positive=['woman', 'king'], negative=['man'])
```

```
Out[57]: [('queen', 0.7854657173156738),
          ('emperor', 0.7523162364959717),
          ('prince', 0.7436755895614624),
          ('princess', 0.713316798210144),
          ('conqueror', 0.7111818194389343),
          ('regent', 0.7088087797164917),
          ('empress', 0.6977599263191223),
          ('sultan', 0.6729022264480591),
          ('confessor', 0.6569845676422119),
          ('duke', 0.6366889476776123)]
```

```
In [58]: ecco_model.most_similar(positive=['she','her','hers','herself'], negative=['he','him'
```

```
Out[58]: [("harriet's", 0.5708541870117188),
          ('softness', 0.5513930320739746),
          ('maiden', 0.5411286354064941),
          ("chloe's", 0.5403314828872681),
          ('lovely', 0.5320479869842529),
          ('coy', 0.5259038209915161),
          ('bewitching', 0.5255858898162842),
          ('soft', 0.5217857956886292),
          ('blushing', 0.5112706422805786),
          ('virgin', 0.5070083141326904)]
```

How does this differ from our novels model?

The words we get returned are much more old english-y. In the original novels model, we got more modern words but here we get classier words.

---

6 Homework

Heuser's blog post explores an analogy in eighteenth-century thought that Riches are to Virtue what Learning is to Genius. How true is this in the ECCO-trained Word2Vec model? Is it true in the one we trained?

How might we compare word2vec models more generally?

```
In [59]: ecco_model.most_similar(positive=['virtue','genius'], negative=['riches'])
```

```
Out[59]: [('piety', 0.6667044162750244),
          ('sense', 0.6573824882507324),
          ('spirit', 0.6482032537460327),
          ('nature', 0.6441324353218079),
          ('sentiment', 0.6437416076660156),
          ('prudence', 0.6216112375259399),
          ('eloquence', 0.6208864450454712),
          ('passion', 0.6205054521560669),
          ('science', 0.6176974177360535),
          ('moralist', 0.6142812371253967)]
```

```
In [60]: model.most_similar(positive=['virtue','genius'], negative=['riches'])
```

```
Out[60]: [('sentiment', 0.7178503274917603),
          ('principle', 0.7096417546272278),
          ('intellect', 0.6723524332046509),
          ('purity', 0.6720075607299805),
          ('religion', 0.6612764596939087),
          ('vice', 0.6489874124526978),
          ('piety', 0.6488131880760193),
          ('wisdom', 0.6478359699249268),
          ('christianity', 0.6471842527389526),
          ('philosophy', 0.6466567516326904)]
```

Both models did an okay job, but in my opinion our model did better because it picked out the word 'intellect'. We might compare word2vec using .similarity of words so that we can get specific numbers to compare to eachother.

---

7 Alternative features for a classification model

This is really cool but what implications does this have for our model of language? Well, word embeddings are simply more precise features of what we've been trying to get at already. That means we can use them in the machine learning models we've been building.

Recall our DTM bag of words classifier:

```
In [61]: import nltk
         nltk.download('movie_reviews')
```

```
[nltk_data] Downloading package movie_reviews to /srv/app/nltk_data...
[nltk_data]   Package movie_reviews is already up-to-date!
```

```
Out[61]: True
```

```
In [62]: from nltk.corpus import movie_reviews
         from sklearn.pipeline import Pipeline
         from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer, TfidfTra
         from sklearn.linear_model import LogisticRegression
         from sklearn.model_selection import cross_val_score, train_test_split
         from sklearn.utils import shuffle

         reviews = [movie_reviews.raw(fileid) for fileid in movie_reviews.fileids()]
         judgements = [movie_reviews.categories(fileid)[0] for fileid in movie_reviews.fileids
```

```
        np.random.seed(0)

        X, y = shuffle(reviews, judgements, random_state=0)
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=

        # get tfidf values
        tfidf = TfidfVectorizer()
        tfidf.fit(X)
        X_train_transformed = tfidf.transform(X_train)
        X_test_transformed = tfidf.transform(X_test)

        # build and test logit
        logit_class = LogisticRegression(penalty='l2', C=1000)
        logit_model = logit_class.fit(X_train_transformed, y_train)
        logit_model.score(X_test_transformed, y_test)
```

```
Out[62]: 0.87250000000000005
```

```
In [63]: print(X_train_transformed)
```

```
  (0, 39482)	0.0149841414947
  (0, 39474)	0.0345030768233
  (0, 39422)	0.0213437436018
  (0, 39396)	0.0144544442222
  (0, 39269)	0.0181623328864
  (0, 39203)	0.020693033426
  (0, 39200)	0.0327929894208
  (0, 39165)	0.0435264547455
  (0, 39031)	0.0294531298824
  (0, 39013)	0.0135878605544
  (0, 38952)	0.0266019615532
  (0, 38849)	0.0172920457952
  (0, 38835)	0.0409828918483
  (0, 38811)	0.017847241143
  (0, 38781)	0.0374593599495
  (0, 38711)	0.0373990816522
  (0, 38707)	0.0219887830607
  (0, 38699)	0.0116541610458
  (0, 38696)	0.0174684430988
  (0, 38614)	0.272537175332
  (0, 38610)	0.0213487428512
  (0, 38508)	0.0304715119971
  (0, 38426)	0.0162050803548
  (0, 38405)	0.0255704404464
  (0, 38285)	0.0734220193703
  :	:
  (1599, 3209)	0.0596711477613
  (1599, 2954)        0.0190221047383
  (1599, 2746)        0.102125517201
  (1599, 2662)        0.0424659765486
  (1599, 2514)        0.0501023995268
  (1599, 2440)        0.036887758569
  (1599, 2396)        0.0101538207029
  (1599, 2351)        0.0344622523885
  (1599, 2217)        0.0435462900877
  (1599, 2013)        0.0232850244456
  (1599, 2006)        0.0169516220772
  (1599, 1810)        0.11718635374
  (1599, 1615)        0.0244658469867
  (1599, 1599)        0.0454699813425
  (1599, 1579)        0.0158445568749
  (1599, 1559)        0.0216703030391
  (1599, 1501)        0.0380565119638
  (1599, 1256)        0.0158718967537
  (1599, 982)       0.0229023960436
  (1599, 966)       0.045249923964
  (1599, 719)       0.0366545417173
  (1599, 291)       0.0581672886126
  (1599, 229)       0.0771511476748
  (1599, 208)       0.0469956369909
  (1599, 194)       0.105021421167
```

So how can we use word embeddings as features? Believe it or not, one of the most effective ways is to simply average each dimension of our embedding across all the words for a given document. Recall our w2v model for novels was trained for 100 dimensions. Creating the features for a specific document would entail first extracting the 100 dimensions for each word, then average each dimension across all words:

```
In [64]: np.mean([model[w] for w in fast_tokenize(X[0]) if w in model], axis=0)
```

```
Out[64]: array([ -4.88956600e-01,   3.50693166e-01,   4.88219857e-02,
                 -2.01642349e-01,   4.13894802e-02,  -4.57355887e-01,
                  8.61368924e-02,  -3.69746119e-01,  -3.84137958e-01,
                 -3.11927825e-01,  -1.80752143e-01,   6.50197923e-01,
                 -4.75103021e-01,   5.88600039e-01,  -3.70605409e-01,
                  7.86905587e-01,   1.80690259e-01,  -4.75590900e-02,
                 -1.15721188e-01,  -4.11623746e-01,  -4.35276419e-01,
                 -3.81389946e-01,   2.50657499e-01,   2.26446465e-01,
                 -6.64307415e-01,  -3.83929640e-01,   5.55162489e-01,
                  1.37895912e-01,  -3.31337680e-04,   2.71445662e-01,
                 -1.43806458e-01,   3.68238725e-02,  -2.83314437e-01,
                  3.32695156e-01,   9.69328284e-01,   2.06312969e-01,
                 -9.73563790e-02,   2.05128402e-01,  -1.42255470e-01,
                 -1.26398085e-02,   2.71465927e-01,   9.83860493e-02,
                 -2.63273448e-01,   3.42925549e-01,   1.04312897e-01,
                 -2.76611716e-01,  -5.39441586e-01,   1.11765377e-01,
                  4.89544630e-01,   4.58817393e-01,   2.02680230e-01,
                 -1.50811523e-01,  -2.69931167e-01,  -8.60979557e-01,
                  2.02418327e-01,   3.96760702e-01,   2.70340353e-01,
                 -6.44397140e-02,  -5.94188012e-02,  -2.74296343e-01,
                 -2.21501574e-01,  -1.75771773e-01,   9.31525156e-02,
                  2.27200300e-01,  -2.11204514e-01,   2.33712137e-01,
                  2.53273219e-01,   3.12077165e-01,   3.49662721e-01,
                 -1.83667839e-01,  -7.20427215e-01,   4.41816419e-01,
                 -3.47270995e-01,  -1.05391607e-01,   6.57563210e-02,
                  1.04039751e-01,   6.55269384e-01,   2.71891207e-01,
                 -6.38460442e-02,   2.65860915e-01,  -5.90726025e-02,
                 -2.79114604e-01,  -3.14846218e-01,   3.15906167e-01,
                 -5.18213093e-01,  -1.73602514e-02,  -1.69169202e-01,
                  4.46759969e-01,   1.20431028e-01,   6.58601522e-02,
                  3.20888221e-01,  -4.83734697e-01,  -1.63129941e-01,
                 -3.69859695e-01,   3.33805442e-01,   2.14833498e-01,
                  5.18838763e-01,   1.23788089e-01,   6.10141277e-01,
                 -2.36541718e-01], dtype=float32)
```

This gives us a set X array with 100 features. We can write a function to do this for us for any given string:

```
In [65]: def w2v_featurize(document, model):
             return np.mean([model[w] for w in fast_tokenize(document) if w in model], axis=0)
```

We can then featurize all of our documents:

```
In [66]: X_train_w2v = [w2v_featurize(d, model) for d in X_train]
         X_test_w2v = [w2v_featurize(d, model) for d in X_test]
```

We can fit and score the machine learning modle just as before:

```
In [67]: logit_class = LogisticRegression(random_state=0, penalty='l2', C=1000)
         logit_model = logit_class.fit(X_train_w2v, y_train)
         logit_model.score(X_test_w2v, y_test)

Out[67]: 0.77749999999999997
```

What about Heuser's model?

```
In [68]: X_train_w2v = [w2v_featurize(d, ecco_model) for d in X_train]
         X_test_w2v = [w2v_featurize(d, ecco_model) for d in X_test]
         logit_class = LogisticRegression(random_state=0, penalty='l2', C=1000)
         logit_model = logit_class.fit(X_train_w2v, y_train)
         logit_model.score(X_test_w2v, y_test)

Out[68]: 0.77249999999999996
```

Cool! But wait, what if we wanted to know *why* the model was making decisions. If we ask for the most postive coefficients:

```
In [69]: np.argsort(logit_model.coef_[0])[-10:]
```

```
Out[69]: array([77, 67, 88, 81, 15, 47, 57, 36, 90, 70])
```

And the negative:

```
In [70]: np.argsort(logit_model.coef_[0])[:10]
```

```
Out[70]: array([97, 96, 49, 87,  0, 52, 84, 42, 43,  1])
```

These are the *indices* for the important features. ***But what are these features now?***

Note that using our novels w2v model was not as accurate as a BoW tfidf method. That should be expected given movie review language is likely ***VERY*** different from our novel corpus. And our novel corpus likely didn't even have entries for a lot of the words used in our movie reviews corpus.

For modern English, most people look for Stanford's GloVe model. This was trained on 6 billion tokens from Wikipedia and Gigaword! Quite a step up from 150 novels. Even the smallest model is a bit large to be working with on our cloud server, but using this model and the code below, you can see it's power:

```
>>> os.system('python -m gensim.scripts.glove2word2vec -i glove.6B.100d.txt -o glove.6B.100d.w
>>> glove = gensim.models.KeyedVectors.load_word2vec_format('glove.6B.100d.w2v.txt')

>>> X_train_glove = [w2v_featurize(d, glove) for d in X_train]
>>> X_test_glove = [w2v_featurize(d, glove) for d in X_test]

>>> logit_class = LogisticRegression(random_state=0, penalty='l2', C=1000)
>>> logit_model = logit_class.fit(X_train_glove, y_train)
>>> logit_model.score(X_test_glove, y_test)

.8125
```

While this is not as accurate as our BoW *tfidf* method, there have been several applications and transformations of word embeddings that have proven to be more accurate than a BoW *tfidf* on general modern text corpora. And keep in mind, one of the most interesting parts of this is that it only uses 100 dimensions, i.e., we can get ~81% accuracy by reducing a movie review to only 100 different features (our BoW model had over 39000!).

```
In [71]: X_train_transformed
```

```
Out[71]: <1600x39659 sparse matrix of type '<class 'numpy.float64'>'
                 with 536995 stored elements in Compressed Sparse Row format>
```

```
In [72]: !pip install --no-cache-dir textblob
```

```
Requirement already satisfied: textblob in /srv/app/venv/lib/python3.6/site-packages
Requirement already satisfied: nltk>=3.1 in /srv/app/venv/lib/python3.6/site-packages (from te
Requirement already satisfied: six in /srv/app/venv/lib/python3.6/site-packages (from nltk>=3.
```

```
In [73]: from textblob import TextBlob

In [74]: my_blob = '''this movie sucked. I hate it when stupid people get the starring role in
         This film was awesome. The cinematography was impeccible. This movie was rad'''

In [75]: blob_object = TextBlob(my_blob)

In [76]: [s for s in blob_object.sentences]

Out[76]: [Sentence("this movie sucked."),
          Sentence("I hate it when stupid people get the starring role in films like this."),
          Sentence("This film was awesome."),
          Sentence("The cinematography was impeccible."),
          Sentence("This movie was rad")]

In [77]: for s in blob_object.sentences:
             print(s)
             print(s.sentiment.polarity)
```

```
this movie sucked.
0.0
I hate it when stupid people get the starring role in films like this.
-0.8
This film was awesome.
1.0
The cinematography was impeccible.
0.0
This movie was rad
0.0
```