Numeric Representations of Words

Now that we have a trained model with some semantic understanding of words, how should we use it? If you look beneath the hood, the Word2Vec model trained in Part 2 consists of a feature vector for each word in the vocabulary, stored in a [numpy](http://www.numpy.org/) array called "syn0":

>>> # Load the model that we created in Part 2  
>>> from gensim.models import Word2Vec  
>>> model = Word2Vec.load("300features\_40minwords\_10context")  
2014-08-03 14:50:15,126 : INFO : loading Word2Vec object from 300features\_40min\_word\_count\_10context  
2014-08-03 14:50:15,777 : INFO : setting ignored attribute syn0norm to None  
  
>>> type(model.syn0)  
<type 'numpy.ndarray'>  
  
>>> model.syn0.shape  
(16492, 300)

The number of rows in syn0 is the number of words in the model's vocabulary, and the number of columns corresponds to the size of the feature vector, which we set in Part 2.  Setting the minimum word count to 40 gave us a total vocabulary of 16,492 words with 300 features apiece. Individual word vectors can be accessed in the following way:

>>> model["flower"]

... which returns a 1x300 numpy array.

From Words To Paragraphs, Attempt 1: Vector Averaging

One challenge with the IMDB dataset is the variable-length reviews. We need to find a way to take individual word vectors and transform them into a feature set that is the same length for every review.

Since each word is a vector in 300-dimensional space, we can use vector operations to combine the words in each review. One method we tried was to simply average the word vectors in a given review (for this purpose, we removed stop words, which would just add noise).

The following code averages the feature vectors, building on our code from Part 2.

import numpy as np # Make sure that numpy is imported  
  
def makeFeatureVec(words, model, num\_features):  
 # Function to average all of the word vectors in a given  
 # paragraph  
 #  
 # Pre-initialize an empty numpy array (for speed)  
 featureVec = np.zeros((num\_features,),dtype="float32")  
 #  
 nwords = 0.  
 #   
 # Index2word is a list that contains the names of the words in   
 # the model's vocabulary. Convert it to a set, for speed   
 index2word\_set = set(model.index2word)  
 #  
 # Loop over each word in the review and, if it is in the model's  
 # vocaublary, add its feature vector to the total  
 for word in words:  
 if word in index2word\_set:   
 nwords = nwords + 1.  
 featureVec = np.add(featureVec,model[word])  
 #   
 # Divide the result by the number of words to get the average  
 featureVec = np.divide(featureVec,nwords)  
 return featureVec  
  
  
def getAvgFeatureVecs(reviews, model, num\_features):  
 # Given a set of reviews (each one a list of words), calculate   
 # the average feature vector for each one and return a 2D numpy array   
 #   
 # Initialize a counter  
 counter = 0.  
 #   
 # Preallocate a 2D numpy array, for speed  
 reviewFeatureVecs = np.zeros((len(reviews),num\_features),dtype="float32")  
 #   
 # Loop through the reviews  
 for review in reviews:  
 #  
 # Print a status message every 1000th review  
 if counter%1000. == 0.:  
 print "Review %d of %d" % (counter, len(reviews))  
 #   
 # Call the function (defined above) that makes average feature vectors  
 reviewFeatureVecs[counter] = makeFeatureVec(review, model, \  
 num\_features)  
 #  
 # Increment the counter  
 counter = counter + 1.  
 return reviewFeatureVecs

Now, we can call these functions to create average vectors for each paragraph. The following operations will take a few minutes:

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
# Calculate average feature vectors for training and testing sets,  
# using the functions we defined above. Notice that we now use stop word  
# removal.  
  
clean\_train\_reviews = []  
for review in train["review"]:  
 clean\_train\_reviews.append( review\_to\_wordlist( review, \  
 remove\_stopwords=True ))  
  
trainDataVecs = getAvgFeatureVecs( clean\_train\_reviews, model, num\_features )  
  
print "Creating average feature vecs for test reviews"  
clean\_test\_reviews = []  
for review in test["review"]:  
 clean\_test\_reviews.append( review\_to\_wordlist( review, \  
 remove\_stopwords=True ))  
  
testDataVecs = getAvgFeatureVecs( clean\_test\_reviews, model, num\_features )

Next, use the average paragraph vectors to train a random forest. Note that, as in Part 1, we can only use the labeled training reviews to train the model.

# Fit a random forest to the training data, using 100 trees  
from sklearn.ensemble import RandomForestClassifier  
forest = RandomForestClassifier( n\_estimators = 100 )  
  
print "Fitting a random forest to labeled training data..."  
forest = forest.fit( trainDataVecs, train["sentiment"] )  
  
# Test & extract results   
result = forest.predict( testDataVecs )  
  
# Write the test results   
output = pd.DataFrame( data={"id":test["id"], "sentiment":result} )  
output.to\_csv( "Word2Vec\_AverageVectors.csv", index=False, quoting=3 )

We found that this produced results much better than chance, but underperformed Bag of Words by a few percentage points.

Since the element-wise average of the vectors didn't produce spectacular results, perhaps we could do it in a more intelligent way? A standard way of weighting word vectors is to apply "[tf-idf](http://en.wikipedia.org/wiki/Tf%E2%80%93idf)" weights, which measure how important a given word is within a given set of documents. One way to extract tf-idf weights in Python is by using scikit-learn's [TfidfVectorizer](http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html), which has an interface similar to the CountVectorizer that we used in Part 1. However, when we tried weighting our word vectors in this way, we found no substantial improvement in performance.

From Words to Paragraphs, Attempt 2: Clustering

Word2Vec creates clusters of semantically related words, so another possible approach is to exploit the similarity of words within a cluster. Grouping vectors in this way is known as "vector quantization." To accomplish this, we first need to find the centers of the word clusters, which we can do by using a [clustering algorithm](http://scikit-learn.org/stable/modules/clustering.html) such as [K-Means](http://en.wikipedia.org/wiki/K-means_clustering).

In K-Means, the one parameter we need to set is "K," or the number of clusters. How should we decide how many clusters to create? Trial and error suggested that small clusters, with an average of only 5 words or so per cluster, gave better results than large clusters with many words. Clustering code is given below. We [use scikit-learn to perform our K-Means](http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html).

K-Means clustering with large K can be very slow; the following code took more than 40 minutes on my computer. Below, we set a timer around the K-Means function to see how long it takes.

from sklearn.cluster import KMeans  
import time  
  
start = time.time() # Start time  
  
# Set "k" (num\_clusters) to be 1/5th of the vocabulary size, or an  
# average of 5 words per cluster  
word\_vectors = model.syn0

num\_clusters = word\_vectors.shape[0] / 5  
  
# Initalize a k-means object and use it to extract centroids  
kmeans\_clustering = KMeans( n\_clusters = num\_clusters )  
idx = kmeans\_clustering.fit\_predict( word\_vectors )  
  
# Get the end time and print how long the process took

end = time.time()  
elapsed = end - start  
print "Time taken for K Means clustering: ", elapsed, "seconds."

The cluster assignment for each word is now stored in idx, and the vocabulary from our original Word2Vec model is still stored in model.index2word. For convenience, we zip these into one dictionary as follows:

# Create a Word / Index dictionary, mapping each vocabulary word to  
# a cluster number

word\_centroid\_map = dict(zip( model.index2word, idx ))

This is a little abstract, so let's take a closer look at what our clusters contain. Your clusters may differ, as Word2Vec relies on a random number seed. Here is a loop that prints out the words for clusters 0 through 9:

# For the first 10 clusters  
for cluster in xrange(0,10):  
 #  
 # Print the cluster number   
 print "\nCluster %d" % cluster  
 #  
 # Find all of the words for that cluster number, and print them out  
 words = []  
 for i in xrange(0,len(word\_centroid\_map.values())):  
 if( word\_centroid\_map.values()[i] == cluster ):  
 words.append(word\_centroid\_map.keys()[i])  
 print words

The results are very interesting:

Cluster 0  
[u'passport', u'penthouse', u'suite', u'seattle', u'apple']

Cluster 1  
[u'unnoticed']

Cluster 2  
[u'midst', u'forming', u'forefront', u'feud', u'bonds', u'merge', u'collide', u'dispute', u'rivalry', u'hostile', u'torn', u'advancing', u'aftermath', u'clans', u'ongoing', u'paths', u'opposing', u'sexes', u'factions', u'journeys']

Cluster 3  
[u'lori', u'denholm', u'sheffer', u'howell', u'elton', u'gladys', u'menjou', u'caroline', u'polly', u'isabella', u'rossi', u'nora', u'bailey', u'mackenzie', u'bobbie', u'kathleen', u'bianca', u'jacqueline', u'reid', u'joyce', u'bennett', u'fay', u'alexis', u'jayne', u'roland', u'davenport', u'linden', u'trevor', u'seymour', u'craig', u'windsor', u'fletcher', u'barrie', u'deborah', u'hayward', u'samantha', u'debra', u'frances', u'hildy', u'rhonda', u'archer', u'lesley', u'dolores', u'elsie', u'harper', u'carlson', u'ella', u'preston', u'allison', u'sutton', u'yvonne', u'jo', u'bellamy', u'conte', u'stella', u'edmund', u'cuthbert', u'maude', u'ellen', u'hilary', u'phyllis', u'wray', u'darren', u'morton', u'withers', u'bain', u'keller', u'martha', u'henderson', u'madeline', u'kay', u'lacey', u'topper', u'wilding', u'jessie', u'theresa', u'auteuil', u'dane', u'jeanne', u'kathryn', u'bentley', u'valerie', u'suzanne', u'abigail']

Cluster 4  
[u'fest', u'flick']

Cluster 5  
[u'lobster', u'deer']

Cluster 6  
[u'humorless', u'dopey', u'limp']

Cluster 7  
[u'enlightening', u'truthful']

Cluster 8  
[u'dominates', u'showcases', u'electrifying', u'powerhouse', u'standout', u'versatility', u'astounding']

Cluster 9  
[u'succumbs', u'comatose', u'humiliating', u'temper', u'looses', u'leans']

We can see that the clusters are of varying quality. Some make sense - Cluster 3 mostly contains names, and Clusters 6-8 contain related adjectives (Cluster 6 is my favorite). On the other hand, Cluster 5 is a little mystifying: What do a lobster and a deer have in common (besides being two animals)? Cluster 0 is even worse: Penthouses and suites seem to belong together, but they don't seem to belong with apples and passports. Cluster 2 contains ... maybe war-related words? Perhaps our algorithm works best on adjectives.

At any rate, now we have a cluster (or "centroid") assignment for each word, and we can define a function to convert reviews into bags-of-centroids. This works just like Bag of Words but uses semantically related clusters instead of individual words:

def create\_bag\_of\_centroids( wordlist, word\_centroid\_map ):  
 #  
 # The number of clusters is equal to the highest cluster index  
 # in the word / centroid map  
 num\_centroids = max( word\_centroid\_map.values() ) + 1  
 #  
 # Pre-allocate the bag of centroids vector (for speed)  
 bag\_of\_centroids = np.zeros( num\_centroids, dtype="float32" )  
 #  
 # Loop over the words in the review. If the word is in the vocabulary,  
 # find which cluster it belongs to, and increment that cluster count   
 # by one  
 for word in wordlist:  
 if word in word\_centroid\_map:  
 index = word\_centroid\_map[word]  
 bag\_of\_centroids[index] += 1  
 #  
 # Return the "bag of centroids"  
 return bag\_of\_centroids

The function above will give us a numpy array for each review, each with a number of features equal to the number of clusters. Finally, we create bags of centroids for our training and test set, then train a random forest and extract results:

# Pre-allocate an array for the training set bags of centroids (for speed)  
train\_centroids = np.zeros( (train["review"].size, num\_clusters), \  
 dtype="float32" )  
  
# Transform the training set reviews into bags of centroids  
counter = 0  
for review in clean\_train\_reviews:  
 train\_centroids[counter] = create\_bag\_of\_centroids( review, \  
 word\_centroid\_map )  
 counter += 1  
  
# Repeat for test reviews   
test\_centroids = np.zeros(( test["review"].size, num\_clusters), \  
 dtype="float32" )  
  
counter = 0  
for review in clean\_test\_reviews:  
 test\_centroids[counter] = create\_bag\_of\_centroids( review, \  
 word\_centroid\_map )  
 counter += 1

# Fit a random forest and extract predictions   
forest = RandomForestClassifier(n\_estimators = 100)  
  
# Fitting the forest may take a few minutes  
print "Fitting a random forest to labeled training data..."  
forest = forest.fit(train\_centroids,train["sentiment"])  
result = forest.predict(test\_centroids)  
  
# Write the test results   
output = pd.DataFrame(data={"id":test["id"], "sentiment":result})  
output.to\_csv( "BagOfCentroids.csv", index=False, quoting=3 )

We found that the code above gives about the same (or slightly worse) results compared to the Bag of Words in Part 1.