In this document, I continue to strive for high accuracy in predicting the classification of documents based on their anonymized words

However, in contrast to the previous effort, I take the order of the words into account here and employ a Recurrent Neural Network (specifically an LSTM). This ends up being important, because my resultant accuracy is 95.5%

```
import numpy as np
import tensorflow as tf
import pandas as pd
from sklearn.decomposition import TruncatedSVD
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import Normalizer
from sklearn.neighbors import KNeighborsClassifier
from sklearn.feature_extraction.text import TfidfVectorizer
import pickle
import random
df = pd.read_csv("shuffled-full-set-hashed.csv", header=None)
```

It's worth pointing out that my TF-IDF vectorizer here is much smaller, and that's because I re-trained it using only single words, not n-grams of up to 3. I did this for the size and speed of the vectorizer, but also because, we are taking word order into account for the LSTM beyond the n-gram length.

```
encoder = pickle.load(open('../server/webservice/pickles/encoder
.pkl', 'rb'))
smaller_vectorizer = pickle.load(open('./smaller_vectorizer.pkl'
, 'rb'))
new_lsa = pickle.load(open('./new_lsa.pkl', 'rb'))
```

I begin by encoding the label for each row as a one-hot vector based on the encoded integer

```
def get_label_vect(label):
    vect = [0] * output_size
    vect[label - 1] = 1
    return vect

output_size = len(encoder.classes_)
sample = df[0:10000]
sample.dropna(inplace=True)
sample_y = sample[0]
encoded_y = np.array([get_label_vect(l) for l in encoder.transfo
rm(sample_y)])
```

```
def chunks(l, n):
    """ Break a list into sized sub-lists """
    for i in range(0, len(l), n):
        new_chunk = l[i:i+n]
        if len(new_chunk) < n:
            new_chunk = (['''] * (n - len(new_chunk))) + new_chun
k
        yield new_chunk

def word_2_vect(word):
    """ For a word, transform into a wordvec """
    return new_lsa.transform(smaller_vectorizer.transform(pd.Ser
ies([word])))[0]

def get_batches(x, y, batch_size=300):
    """ Break up input into batch-sized tensors """
    n_batches = len(x)//batch_size
    x, y = x[:n_batches*batch_size], y[:n_batches*batch_size]
    for ii in range(0, len(x), batch_size):
        yield x[ii:ii+batch_size], y[ii:ii+batch_size]
```

Break the documents up into 20 word sized chunks, each associated with the correct label vector

```
chunk_size = 20
training_set = []
training_vects = []

doc_arr = [x for x in sample[1]]
doc_arr[0].split(' ')
doc_words = [doc.split(' ') for doc in doc_arr]

for i in range(len(doc_words)):
    for chunk in chunks(doc_words[i], chunk_size):
        training_set.append((encoded_y[i], chunk))

random.shuffle(training_set)
```

Now, with our previously saved vectorizers, convert each of those words into 100-dimension vectors. Don't forget to save!

```
i = 0
for label, chunk in training_set:
    training_vects.append((label, [word_2_vect(word) for word in
chunk]))
    if i%500 == 0:
        print('{} of {}'.format(i, len(training_set)))
    i += 1


i = 0
for ch in chunks(training_vects, 80000):
    pickle.dump(ch, open('training_vects_big_{}.pkl'.format(i),
'wb'))
    i += 1
```

```
lstm_size = 256
lstm_layers = 2
batch_size = 500
learning_rate = 0.001
```

```
# Create the graph object
graph = tf.Graph()
# Add nodes to the graph
with graph.as_default():
    inputs_ = tf.placeholder("float", [None, chunk_size, 100])
    labels_ = tf.placeholder(tf.int32, [None, output_size])
    keep_prob = tf.placeholder(tf.float32, name='keep_prob')
```

Now we begin building our LSTM, First we create our two layers, each wrapped in a dropout layer

```
with graph.as_default():
    with tf.name_scope("RNN_layers"):
        def lstm_cell():
            lstm = tf.contrib.rnn.BasicLSTMCell(lstm_size, reuse
=tf.get_variable_scope().reuse)
            return tf.contrib.rnn.DropoutWrapper(lstm, output_ke
ep_prob=keep_prob)

        cell = tf.contrib.rnn.MultiRNNCell([lstm_cell() for _ in
range(lstm_layers)])
        initial_state = cell.zero_state(batch_size, tf.float32)

    outputs, final_state = tf.nn.dynamic_rnn(cell, inputs_,
                                             initial_state=initi
al_state)
```

Now we generate predictions, an output vector based on a logistic activation. The cost is the mean squared error with our one-hot label vectors, and our optimization algorithm, ADAM.

```
with graph.as_default():
    predictions = tf.contrib.layers.fully_connected(outputs[:, -
1], output_size, activation_fn=tf.sigmoid)
    cost = tf.losses.mean_squared_error(labels_, predictions)
    optimizer = tf.train.AdamOptimizer(learning_rate).minimize(c
ost)
```

```
with graph.as_default():
    correct_pred = tf.equal(tf.cast(tf.round(predictions), tf.in
t32), labels_)
    accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

Now we break up our cleaned data set into a training group, a validation group, and a testing group.

```
train_prop = 0.8

set_size = x.shape[0]

split_idx = int(set_size*0.8)
train_x, rest_x = x[:split_idx], x[split_idx:]
train_y, rest_y = y[:split_idx], y[split_idx:]

val_prop = 0.5
rest_size = rest_x.shape[0]
val_idx = int(val_prop * rest_size)
val_x, test_x = rest_x[:val_idx], rest_x[val_idx:]
val_y, test_y = rest_y[:val_idx], rest_y[val_idx:]
```

And run it through our LSTM

```
epochs = 10

with graph.as_default():
    saver = tf.train.Saver()

with tf.Session(graph=graph) as sess:
    sess.run(tf.global_variables_initializer())
    iteration = 1
    for e in range(epochs):
        state = sess.run(initial_state)

        for ii, (batch_x, batch_y) in enumerate(get_batches(trai
n_x, train_y, batch_size), 1):

            feed = {inputs_: batch_x,
```

```
                    labels_: batch_y,
                    keep_prob: 0.5,
                    initial_state: state}
            loss, state, _ = sess.run([cost, final_state, optimi
zer], feed_dict=feed)

            if iteration%5==0:
                print("Epoch: {}/{}".format(e, epochs),
                      "Iteration: {}".format(iteration),
                      "Train loss: {:.3f}".format(loss))

            if iteration%25==0:
                val_acc = []
                val_state = sess.run(cell.zero_state(batch_size,
tf.float32))

                for batch_val_x, batch_val_y in get_batches(val_
x, val_y, batch_size):
                    feed = {inputs_: batch_val_x,
                            labels_: batch_val_y,
                            keep_prob: 1,
                            initial_state: val_state}
                    batch_acc, val_state = sess.run([accuracy, f
inal_state], feed_dict=feed)

                    val_acc.append(batch_acc)
                print("Val acc: {:.3f}".format(np.mean(val_acc))
)


            iteration +=1
            saver.save(sess, "checkpoints/final_sentiment.ckpt")
    saver.save(sess, "checkpoints/final_sentiment.ckpt")
```

```
Epoch: 0/10 Iteration: 5 Train loss: 0.195
Epoch: 0/10 Iteration: 10 Train loss: 0.061
Epoch: 0/10 Iteration: 15 Train loss: 0.062
Epoch: 0/10 Iteration: 20 Train loss: 0.061
Epoch: 0/10 Iteration: 25 Train loss: 0.058
Val acc: 0.929
Epoch: 9/10 Iteration: 2655 Train loss: 0.035
Epoch: 9/10 Iteration: 2660 Train loss: 0.038
Epoch: 9/10 Iteration: 2665 Train loss: 0.036
Epoch: 9/10 Iteration: 2670 Train loss: 0.037
Epoch: 9/10 Iteration: 2675 Train loss: 0.037
...
Epoch: 9/10 Iteration: 2680 Train loss: 0.036
Epoch: 9/10 Iteration: 2685 Train loss: 0.037
Epoch: 9/10 Iteration: 2690 Train loss: 0.037
Epoch: 9/10 Iteration: 2695 Train loss: 0.037
Epoch: 9/10 Iteration: 2700 Train loss: 0.038
Val acc: 0.953
```

And run the test data set through it...

```
test_acc = []
with tf.Session(graph=graph) as sess:
    saver.restore(sess, "checkpoints/final_sentiment.ckpt")
    test_state = sess.run(cell.zero_state(batch_size, tf.float32
))
    for ii, (bat_test_x, bat_test_y) in enumerate(get_batches(te
st_x, test_y, batch_size), 1):
        feed = {inputs_: bat_test_x,
                labels_: bat_test_y,
                keep_prob: 1,
                initial_state: test_state}
        batch_acc, test_state = sess.run([accuracy, final_state]
, feed_dict=feed)
        test_acc.append(batch_acc)
    print("Test accuracy: {:.3f}".format(np.mean(test_acc)))
```

```
Test accuracy: 0.954
```

95.4% Accuracy! Good time to push this live.