Assignment 8 - Language Modeling With an RNN

MSDS 422 - SEC 57 THURSDAY

FERDYNAND HEBAL - 8/18/2019

In this assignment, language modeling with a recurrent neural network is explored within a 2x2 full factorial/crossed benchmark experiment on the Dogs vs. Cats problem on Kaggle.com. Model performance accuracy and processing times are assessed using Keras and TensorFlow. Due to the time required to fit each model only 5000 of the 25000 available images are used. Results are presented in Table 1.

Data preparation, exploration, visualization

Starter code is used to download pretrained vectors and employ an RNN model

In [26]:
```
# coding: utf-8

# Gather embeddings via chakin
# Following methods described in
#    https://github.com/chakki-works/chakin

# As originally configured, this program downloads four
# pre-trained GloVe embeddings, saves them in a zip archive,
# and then unzips the archive to create the four word-to-embeddings
# text files for use in language models.

# Note that the downloading process can take about 10 minutes to complet
e.

import numpy as np
import tensorflow as tf

# Python chakin package previously installed by
#    pip install chakin
import chakin

import json
import os
from collections import defaultdict

chakin.search(lang='English')  # lists available indices in English

# Specify English embeddings file to download and install
# by index number, number of dimensions, and subfoder name
# Note that GloVe 50-, 100-, 200-, and 300-dimensional folders
# are downloaded with a single zip download
CHAKIN_INDEX = 11
NUMBER_OF_DIMENSIONS = 50
SUBFOLDER_NAME = "gloVe.6B"

DATA_FOLDER = "embeddings"
ZIP_FILE = os.path.join(DATA_FOLDER, "{}.zip".format(SUBFOLDER_NAME))
ZIP_FILE_ALT = "glove" + ZIP_FILE[5:]  # sometimes it's lowercase onl
y...
UNZIP_FOLDER = os.path.join(DATA_FOLDER, SUBFOLDER_NAME)
if SUBFOLDER_NAME[-1] == "d":
    GLOVE_FILENAME = os.path.join(
        UNZIP_FOLDER, "{}.txt".format(SUBFOLDER_NAME))
else:
    GLOVE_FILENAME = os.path.join(UNZIP_FOLDER, "{}.{}d.txt".format(
        SUBFOLDER_NAME, NUMBER_OF_DIMENSIONS))


if not os.path.exists(ZIP_FILE) and not os.path.exists(UNZIP_FOLDER):
    # GloVe by Stanford is licensed Apache 2.0:
    #     https://github.com/stanfordnlp/GloVe/blob/master/LICENSE
    #     http://nlp.stanford.edu/data/glove.twitter.27B.zip
    #     Copyright 2014 The Board of Trustees of The Leland Stanford Ju
nior University
    print("Downloading embeddings to '{}'".format(ZIP_FILE))
    chakin.download(number=CHAKIN_INDEX, save_dir='./{}'.format(DATA_FOL
```

```
DER))
else:
    print("Embeddings already downloaded.")

if not os.path.exists(UNZIP_FOLDER):
    import zipfile
    if not os.path.exists(ZIP_FILE) and os.path.exists(ZIP_FILE_ALT):
        ZIP_FILE = ZIP_FILE_ALT
    with zipfile.ZipFile(ZIP_FILE, "r") as zip_ref:
        print("Extracting embeddings to '{}'".format(UNZIP_FOLDER))
        zip_ref.extractall(UNZIP_FOLDER)
else:
    print("Embeddings already extracted.")

print('\nRun complete')
```

```
                   Name  Dimension                      Corpus Vocabular
ySize  \
2          fastText(en)        300                   Wikipedia
2.5M
11          GloVe.6B.50d         50  Wikipedia+Gigaword 5 (6B)
400K
12         GloVe.6B.100d        100  Wikipedia+Gigaword 5 (6B)
400K
13         GloVe.6B.200d        200  Wikipedia+Gigaword 5 (6B)
400K
14         GloVe.6B.300d        300  Wikipedia+Gigaword 5 (6B)
400K
15        GloVe.42B.300d        300          Common Crawl(42B)
1.9M
16       GloVe.840B.300d        300         Common Crawl(840B)
2.2M
17    GloVe.Twitter.25d         25                Twitter(27B)
1.2M
18    GloVe.Twitter.50d         50                Twitter(27B)
1.2M
19   GloVe.Twitter.100d        100                Twitter(27B)
1.2M
20   GloVe.Twitter.200d        200                Twitter(27B)
1.2M
21  word2vec.GoogleNews        300           Google News(100B)
3.0M

      Method Language    Author
2   fastText  English  Facebook
11     GloVe  English  Stanford
12     GloVe  English  Stanford
13     GloVe  English  Stanford
14     GloVe  English  Stanford
15     GloVe  English  Stanford
16     GloVe  English  Stanford
17     GloVe  English  Stanford
18     GloVe  English  Stanford
19     GloVe  English  Stanford
20     GloVe  English  Stanford
21  word2vec  English    Google
Downloading embeddings to 'embeddings/gloVe.6B.zip'

Test: 100% ||                                    | Time:  0:01:38
8.3 MiB/s

Extracting embeddings to 'embeddings/gloVe.6B'

Run complete
```

In [31]:
```
# coding: utf-8

# Program by Thomas W. Miller, August 16, 2018

# Previous work involved gathering embeddings via chakin
# Following methods described in
#    https://github.com/chakki-works/chakin
# The previous program, run-chakin-to-get-embeddings-v001.py
# downloaded pre-trained GloVe embeddings, saved them in a zip archive,
# and unzipped that archive to create the four word-to-embeddings
# text files for use in language models.

# This program sets uses word embeddings to set up defaultdict
# dictionary data structures, that can them be employed in language
# models. This is demonstrated with a simple RNN model for predicting
# sentiment (thumbs-down versus thumbs-up) for movie reviews.

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import numpy as np

import os  # operating system functions
import os.path  # for manipulation of file path names

import re  # regular expressions

from collections import defaultdict

import nltk
from nltk.tokenize import TreebankWordTokenizer

import tensorflow as tf

RANDOM_SEED = 9999

# To make output stable across runs
def reset_graph(seed= RANDOM_SEED):
    tf.reset_default_graph()
    tf.set_random_seed(seed)
    np.random.seed(seed)

REMOVE_STOPWORDS = False  # no stopword removal

EVOCABSIZE = 10000  # specify desired size of pre-defined embedding voca
bulary

# -------------------------------------------------------------
# Select the pre-defined embeddings source
# Define vocabulary size for the language model
# Create a word_to_embedding_dict for GloVe.6B.50d
embeddings_directory = 'embeddings/gloVe.6B'
filename = 'glove.6B.50d.txt'
embeddings_filename = os.path.join(embeddings_directory, filename)
# -------------------------------------------------------------
```

```
def miller_model(EVOCABSIZE,embeddings_filename):
    # Utility function for loading embeddings follows methods described
 in
    # https://github.com/guillaume-chevalier/GloVe-as-a-TensorFlow-Embed
ding-Layer
    # Creates the Python defaultdict dictionary word_to_embedding_dict
    # for the requested pre-trained word embeddings
    #
    # Note the use of defaultdict data structure from the Python Standar
d Library
    # collections_defaultdict.py lets the caller specify a default value
up front
    # The default value will be retuned if the key is not a known dictio
nary key
    # That is, unknown words are represented by a vector of zeros
    # For word embeddings, this default value is a vector of zeros
    # Documentation for the Python standard library:
    #   Hellmann, D. 2017. The Python 3 Standard Library by Example. Bos
ton:
    #     Addison-Wesley. [ISBN-13: 978-0-13-429105-5]
    def load_embedding_from_disks(embeddings_filename, with_indexes=True
):
        """
        Read a embeddings txt file. If `with_indexes=True`,
        we return a tuple of two dictionnaries
        `(word_to_index_dict, index_to_embedding_array)`,
        otherwise we return only a direct
        `word_to_embedding_dict` dictionnary mapping
        from a string to a numpy array.
        """
        if with_indexes:
            word_to_index_dict = dict()
            index_to_embedding_array = []

        else:
            word_to_embedding_dict = dict()

        with open(embeddings_filename, 'r', encoding='utf-8') as embeddi
ngs_file:
            for (i, line) in enumerate(embeddings_file):

                split = line.split(' ')

                word = split[0]

                representation = split[1:]
                representation = np.array(
                    [float(val) for val in representation]
                )

                if with_indexes:
                    word_to_index_dict[word] = i
                    index_to_embedding_array.append(representation)
                else:
                    word_to_embedding_dict[word] = representation
```

```
        # Empty representation for unknown words.
        _WORD_NOT_FOUND = [0.0] * len(representation)
        if with_indexes:
            _LAST_INDEX = i + 1
            word_to_index_dict = defaultdict(
                lambda: _LAST_INDEX, word_to_index_dict)
            index_to_embedding_array = np.array(
                index_to_embedding_array + [_WORD_NOT_FOUND])
            return word_to_index_dict, index_to_embedding_array
        else:
            word_to_embedding_dict = defaultdict(lambda: _WORD_NOT_FOUND
)
            return word_to_embedding_dict

    print('\nLoading embeddings from', embeddings_filename)
    word_to_index, index_to_embedding = \
        load_embedding_from_disks(embeddings_filename, with_indexes=True
)
    print("Embedding loaded from disks.")

    # Note: unknown words have representations with values [0, 0, ...,
 0]

    # Additional background code from
    # https://github.com/guillaume-chevalier/GloVe-as-a-TensorFlow-Embed
ding-Layer
    # shows the general structure of the data structures for word embedd
ings
    # This code is modified for our purposes in language modeling
    vocab_size, embedding_dim = index_to_embedding.shape
    print("Embedding is of shape: {}".format(index_to_embedding.shape))
    print("This means (number of words, number of dimensions per word)\n
")
    print("The first words are words that tend occur more often.")

    print("Note: for unknown words, the representation is an empty vecto
r,\n"
          "and the index is the last one. The dictionnary has a limit:")
    print("    {} --> {} --> {}".format("A word", "Index in embedding",
          "Representation"))
    word = "worsdfkljsdf"  # a word obviously not in the vocabulary
    idx = word_to_index[word] # index for word obviously not in the voca
bulary
    complete_vocabulary_size = idx
    embd = list(np.array(index_to_embedding[idx], dtype=int)) # "int" co
mpact print
    print("    {} --> {} --> {}".format(word, idx, embd))
    word = "the"
    idx = word_to_index[word]
    embd = list(index_to_embedding[idx])  # "int" for compact print onl
y.
    print("    {} --> {} --> {}".format(word, idx, embd))

    # Show how to use embeddings dictionaries with a test sentence
    # This is a famous typing exercise with all letters of the alphabet
    # https://en.wikipedia.org/wiki/The_quick_brown_fox_jumps_over_the_l
```

```
azy_dog
    a_typing_test_sentence = 'The quick brown fox jumps over the lazy do
g'
    print('\nTest sentence: ', a_typing_test_sentence, '\n')
    words_in_test_sentence = a_typing_test_sentence.split()

    print('Test sentence embeddings from complete vocabulary of',
          complete_vocabulary_size, 'words:\n')
    for word in words_in_test_sentence:
        word_ = word.lower()
        embedding = index_to_embedding[word_to_index[word_]]
        print(word_ + ": ", embedding)

    # ------------------------------------------------------------
    # Define vocabulary size for the language model
    # To reduce the size of the vocabulary to the n most frequently used
words

    def default_factory():
        return EVOCABSIZE  # last/unknown-word row in limited_index_to_e
mbedding
    # dictionary has the items() function, returns list of (key, value)
 tuples
    limited_word_to_index = defaultdict(default_factory, \
        {k: v for k, v in word_to_index.items() if v < EVOCABSIZE})

    # Select the first EVOCABSIZE rows to the index_to_embedding
    limited_index_to_embedding = index_to_embedding[0:EVOCABSIZE,:]
    # Set the unknown-word row to be all zeros as previously
    limited_index_to_embedding = np.append(limited_index_to_embedding,
        index_to_embedding[index_to_embedding.shape[0] - 1, :].\
            reshape(1,embedding_dim),
        axis = 0)

    # Delete large numpy array to clear some CPU RAM
    del index_to_embedding

    # Verify the new vocabulary: should get same embeddings for test sen
tence
    # Note that a small EVOCABSIZE may yield some zero vectors for embed
dings
    print('\nTest sentence embeddings from vocabulary of', EVOCABSIZE,
'words:\n')
    for word in words_in_test_sentence:
        word_ = word.lower()
        embedding = limited_index_to_embedding[limited_word_to_index[wor
d_]]
        print(word_ + ": ", embedding)

    # ------------------------------------------------------------
    # code for working with movie reviews data
    # Source: Miller, T. W. (2016). Web and Network Data Science.
    #    Upper Saddle River, N.J.: Pearson Education.
    #    ISBN-13: 978-0-13-388644-3
    # This original study used a simple bag-of-words approach
    # to sentiment analysis, along with pre-defined lists of
    # negative and positive words.
```

```
    # Code available at:  https://github.com/mtpa/wnds
    # ------------------------------------------------------------
    # Utility function to get file names within a directory
    def listdir_no_hidden(path):
        start_list = os.listdir(path)
        end_list = []
        for file in start_list:
            if (not file.startswith('.')):
                end_list.append(file)
        return(end_list)

    # define list of codes to be dropped from document
    # carriage-returns, line-feeds, tabs
    codelist = ['\r', '\n', '\t']

    # We will not remove stopwords in this exercise because they are
    # important to keeping sentences intact
    if REMOVE_STOPWORDS:
        print(nltk.corpus.stopwords.words('english'))

    # previous analysis of a list of top terms showed a number of words,
along
    # with contractions and other word strings to drop from further anal
ysis, add
    # these to the usual English stopwords to be dropped from a document
collection
        more_stop_words = ['cant','didnt','doesnt','dont','goes','isnt',
'hes',\
            'shes','thats','theres','theyre','wont','youll','youre','you
ve', 'br'\
            've', 're', 'vs']

        some_proper_nouns_to_remove = ['dick','ginger','hollywood','jac
k',\
            'jill','john','karloff','kudrow','orson','peter','tcm','tom'
,\
            'toni','welles','william','wolheim','nikita']

        # start with the initial list and add to it for movie text work
        stoplist = nltk.corpus.stopwords.words('english') + more_stop_wo
rds +\
            some_proper_nouns_to_remove

    # text parsing function for creating text documents
    # there is more we could do for data preparation
    # stemming... looking for contractions... possessives...
    # but we will work with what we have in this parsing function
    # if we want to do stemming at a later time, we can use
    #     porter = nltk.PorterStemmer()
    # in a construction like this
    #     words_stemmed =  [porter.stem(word) for word in initial_words]
    def text_parse(string):
        # replace non-alphanumeric with space
        temp_string = re.sub('[^a-zA-Z]', '  ', string)
        # replace codes with space
        for i in range(len(codelist)):
            stopstring = ' ' + codelist[i] + '  '
```

```
            temp_string = re.sub(stopstring, '  ', temp_string)
        # replace single-character words with space
        temp_string = re.sub('\s.\s', ' ', temp_string)
        # convert uppercase to lowercase
        temp_string = temp_string.lower()
        if REMOVE_STOPWORDS:
            # replace selected character strings/stop-words with space
            for i in range(len(stoplist)):
                stopstring = ' ' + str(stoplist[i]) + ' '
                temp_string = re.sub(stopstring, ' ', temp_string)
        # replace multiple blank characters with one blank character
        temp_string = re.sub('\s+', ' ', temp_string)
        return(temp_string)

    # ------------------------------------------------
    # gather data for 500 negative movie reviews
    # ------------------------------------------------
    dir_name = 'movie-reviews-negative'

    filenames = listdir_no_hidden(path=dir_name)
    num_files = len(filenames)

    for i in range(len(filenames)):
        file_exists = os.path.isfile(os.path.join(dir_name, filenames[i
]))
        assert file_exists
    print('\nDirectory:',dir_name)
    print('%d files found' % len(filenames))

    # Read data for negative movie reviews
    # Data will be stored in a list of lists where the each list represe
nts
    # a document and document is a list of words.
    # We then break the text into words.

    def read_data(filename):
        with open(filename, encoding='utf-8') as f:
            data = tf.compat.as_str(f.read())
            data = data.lower()
            data = text_parse(data)
            data = TreebankWordTokenizer().tokenize(data)  # The Penn Tr
eebank
        return data

    negative_documents = []

    print('\nProcessing document files under', dir_name)
    for i in range(num_files):
        ## print(' ', filenames[i])

        words = read_data(os.path.join(dir_name, filenames[i]))

        negative_documents.append(words)
        # print('Data size (Characters) (Document %d) %d' %(i,len(word
s)))
        # print('Sample string (Document %d) %s'%(i,words[:50]))
```

```
    # ------------------------------------------------
    # gather data for 500 positive movie reviews
    # ------------------------------------------------
    dir_name = 'movie-reviews-positive'
    filenames = listdir_no_hidden(path=dir_name)
    num_files = len(filenames)

    for i in range(len(filenames)):
        file_exists = os.path.isfile(os.path.join(dir_name, filenames[i
]))
        assert file_exists
    print('\nDirectory:',dir_name)
    print('%d files found' % len(filenames))

    # Read data for positive movie reviews
    # Data will be stored in a list of lists where the each list
    # represents a document and document is a list of words.
    # We then break the text into words.

    def read_data(filename):
        with open(filename, encoding='utf-8') as f:
            data = tf.compat.as_str(f.read())
            data = data.lower()
            data = text_parse(data)
            data = TreebankWordTokenizer().tokenize(data)  # The Penn Tr
eebank
        return data

    positive_documents = []

    print('\nProcessing document files under', dir_name)
    for i in range(num_files):
        ## print(' ', filenames[i])

        words = read_data(os.path.join(dir_name, filenames[i]))

        positive_documents.append(words)
        # print('Data size (Characters) (Document %d) %d' %(i,len(word
s)))
        # print('Sample string (Document %d) %s'%(i,words[:50]))

    # ------------------------------------------------------
    # convert positive/negative documents into numpy array
    # note that reviews vary from 22 to 1052 words
    # so we use the first 20 and last 20 words of each review
    # as our word sequences for analysis
    # ------------------------------------------------------
    max_review_length = 0  # initialize
    for doc in negative_documents:
        max_review_length = max(max_review_length, len(doc))
    for doc in positive_documents:
        max_review_length = max(max_review_length, len(doc))
    print('max_review_length:', max_review_length)

    min_review_length = max_review_length  # initialize
    for doc in negative_documents:
```

```
        min_review_length = min(min_review_length, len(doc))
    for doc in positive_documents:
        min_review_length = min(min_review_length, len(doc))
    print('min_review_length:', min_review_length)

    # construct list of 1000 lists with 40 words in each list
    from itertools import chain
    documents = []
    for doc in negative_documents:
        doc_begin = doc[0:20]
        doc_end = doc[len(doc) - 20: len(doc)]
        documents.append(list(chain(*[doc_begin, doc_end])))
    for doc in positive_documents:
        doc_begin = doc[0:20]
        doc_end = doc[len(doc) - 20: len(doc)]
        documents.append(list(chain(*[doc_begin, doc_end])))

    # create list of lists of lists for embeddings
    embeddings = []
    for doc in documents:
        embedding = []
        for word in doc:
            embedding.append(limited_index_to_embedding[limited_word_to_
index[word]])
        embeddings.append(embedding)

    # -----------------------------------------------------
    # Check on the embeddings list of list of lists
    # -----------------------------------------------------
    # Show the first word in the first document
    test_word = documents[0][0]
    print('First word in first document:', test_word)
    print('Embedding for this word:\n',
          limited_index_to_embedding[limited_word_to_index[test_word]])
    print('Corresponding embedding from embeddings list of list of lists
\n',
          embeddings[0][0][:])

    # Show the seventh word in the tenth document
    test_word = documents[6][9]
    print('First word in first document:', test_word)
    print('Embedding for this word:\n',
          limited_index_to_embedding[limited_word_to_index[test_word]])
    print('Corresponding embedding from embeddings list of list of lists
\n',
          embeddings[6][9][:])

    # Show the last word in the last document
    test_word = documents[999][39]
    print('First word in first document:', test_word)
    print('Embedding for this word:\n',
          limited_index_to_embedding[limited_word_to_index[test_word]])
    print('Corresponding embedding from embeddings list of list of lists
\n',
          embeddings[999][39][:])

    # -----------------------------------------------------
```

```
    # Make embeddings a numpy array for use in an RNN
    # Create training and test sets with Scikit Learn
    # -----------------------------------------------------
    embeddings_array = np.array(embeddings)

    # Define the labels to be used 500 negative (0) and 500 positive (1)
    thumbs_down_up = np.concatenate((np.zeros((500), dtype = np.int32),
                          np.ones((500), dtype = np.int32)), axis = 0)

    # Scikit Learn for random splitting of the data
    from sklearn.model_selection import train_test_split

    # Random splitting of the data in to training (80%) and test (20%)
    X_train, X_test, y_train, y_test = \
        train_test_split(embeddings_array, thumbs_down_up, test_size=0.2
0,
                         random_state = RANDOM_SEED)

    # ------------------------------------------------------------------
--------
    # We use a very simple Recurrent Neural Network for this assignment
    # Géron, A. 2017. Hands-On Machine Learning with Scikit-Learn & Tens
orFlow:
    #    Concepts, Tools, and Techniques to Build Intelligent Systems.
    #    Sebastopol, Calif.: O'Reilly. [ISBN-13 978-1-491-96229-9]
    #    Chapter 14 Recurrent Neural Networks, pages 390-391
    #    Source code available at https://github.com/ageron/handson-ml
    #    Jupyter notebook file 14_recurrent_neural_networks.ipynb
    #    See section on Training an sequence Classifier, # In [34]:
    #    which uses the MNIST case data...  we revise to accommodate
    #    the movie review data in this assignment
    # ------------------------------------------------------------------
--------
    reset_graph()

    n_steps = embeddings_array.shape[1]  # number of words per document
    n_inputs = embeddings_array.shape[2]  # dimension of  pre-trained em
beddings
    n_neurons = 20  # analyst specified number of neurons
    n_outputs = 2  # thumbs-down or thumbs-up

    learning_rate = 0.001

    X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
    y = tf.placeholder(tf.int32, [None])

    basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
    outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)

    logits = tf.layers.dense(states, n_outputs)
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y,
                                                              logits=log
its)
    loss = tf.reduce_mean(xentropy)
    optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
    training_op = optimizer.minimize(loss)
    correct = tf.nn.in_top_k(logits, y, 1)
```

```
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

    init = tf.global_variables_initializer()

    n_epochs = 50
    batch_size = 100

    with tf.Session() as sess:
        init.run()
        for epoch in range(n_epochs):
            print('\n  ---- Epoch ', epoch, ' ----\n')
            for iteration in range(y_train.shape[0] // batch_size):
                X_batch = X_train[iteration*batch_size:(iteration + 1)*b
atch_size,:]
                y_batch = y_train[iteration*batch_size:(iteration + 1)*b
atch_size]
                print('  Batch ', iteration, ' training observations fro
m ',
                      iteration*batch_size, ' to ', (iteration + 1)*batc
h_size-1,)
                sess.run(training_op, feed_dict={X: X_batch, y: y_batch
})
            acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch
})
            acc_test = accuracy.eval(feed_dict={X: X_test, y: y_test})
            print('\n  Train accuracy:', acc_train, 'Test accuracy:', ac
c_test)
    return acc_train, acc_test
```

For this experiment, the following model structures are evaluated:

RNN using vocabulary size of 10,000 and the 50 dimension glove.6B pretrained vector

RNN using vocabulary size of 10,000 and the 50 dimension glove.6B pretrained vector

RNN using vocabulary size of 50,000 and the 300 dimension glove.6B pretrained vector

RNN using vocabulary size of 50,000 and the 300 dimension glove.6B pretrained vector

In [59]:
```
import time
import pandas as pd
from IPython.display import clear_output

sizes = [10000,50000]
filenames = ['glove.6B.50d.txt','glove.6B.300d.txt']
experiments = [(x,y) for x in sizes for y in filenames] #cartesion produ
ct of sizes and filenames
embeddings_directory = 'embeddings/gloVe.6B'


results = []
for experiment in experiments:
    embeddings_filename = os.path.join(embeddings_directory, experiment[
1])
    start = time.time()
    acc_train, acc_test = miller_model(experiment[0],embeddings_filename
)
    done = time.time()
    processing_time = done - start
    results.append([experiment[0],experiment[1], acc_train, acc_test,pro
cessing_time, acc_train - acc_test])
clear_output(wait=True)
table = pd.DataFrame(results)
table.columns = ['vocab_size','word_vector','train_acc','test_acc','proc
essing_time','acc_diff']
print('Table 1. Tensor Flow Language modeling with a Recurrent Neural Ne
twork')
table
```

```
Table 1. Tensor Flow Language modeling with a Recurrent Neural Network
```

Out[59]:

	vocab_size	word_vector	train_acc	test_acc	processing_time	acc_diff
0	10000	glove.6B.50d.txt	0.86	0.675	8.583361	0.185
1	10000	glove.6B.300d.txt	0.98	0.625	39.536008	0.355
2	50000	glove.6B.50d.txt	0.86	0.660	8.315508	0.200
3	50000	glove.6B.300d.txt	0.98	0.625	36.101784	0.355

Summary

Regarding the management problem, these results suggest that language modeling using recurrent neural networks with a larger vocabulary, no improvement is seen given the same number of dimensions in the pretrained vector used, and performance actually worsens in test accuracy with a larger number of dimensions given the same size vocabulary. Accuracy in training does improve wiht a greated number of dimensions but since the gap between test and train accuracy wideneds this suggests overfitting. For a vocabulary size of 10000 and the 300d vector, the difference in accuracy between test and train data are more than twice that of the experiment using a vocabulary size of 10000 and a 50d vector. Interestingly, increased vocabulary size does not impact the difference between train and test accuracy much given the same number of dimensions.

The increase in number of dimensions seems to incur a much greater cost in processing time than does the vocabulary size. Given these results, it is clear that a simple recurrent neural network structure cannot achieve a high performance model in test data and manipulating vocabulary size and dimension hyperparameters only risks overfitting and increased processing times.

Considering the results from this benchmark study I would conclude that RNN classifier needs more extensive testing and experimentation with different pretrained word vectors. Dimensionality of the word vector seems to incur cost in runtime with little improvement to model accuracy.

From the perspective of senior management thinking about using a language model to classify written customer reviews and call and complaint logs I would recommend additional testing to improve the model, and if only slightly more accuracy can be achieved with machine learning comapared to using manual or other methods then the time saved in achieving some improved accuracy should be weighed against the time classification woudl take using non machine learning methods, or the cost of not doing any classification at all. Sometimes a model that performs only a little better than guess work and still trains and classifies faster is more feasible than manual classification and can result in net profit or net improvement in customer satisfcation (depending on your desired outcome).

Data scientists can work to test many hyperparameters and different pretrained vectora, perhaps even develop their own vectors. Optimizers can be employed to mitigate processing times, e.g methods like stocahstic gradient descent to avoid overfitting. NLP is a difficult arena though there are known successes and a great deal of improvement may come in time. In the meantime if the costs do not outweigh the benefits, perhaps abstaining from investment may be the prudent course of action at this time.