DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Homework Assignment No. 08:

PYTHON SCRIPTING

Submitted to:

Professor Joseph Picone
ECE 3822: Software Tools for Engineers
Temple University
College of Engineering
1947 North 12th Street
Philadelphia, Pennsylvania 19122

11/29/2015

Prepared by:

Devin Trejo
Email: devin.trejo@temple.edu

1. PROBLEM

In assignment eight we switch over to python scripting in which we will reproduce our results we found in homework assignment number one but using python. Reiterating homework one goals:

"In this first assignment of the semester we simply want to familiarize ourselves with command line tools. We start by learning how to personalize our Linux configuration by editing the .bash_profile and .bashrc files in ~/. Aliases and manipulating the environment path allow us to run commands from any directory on our machines.

1. *Edit the environment path so a 'hello world' command can be ran from any directory. Then create an alias by modifying the .bash_profile.*

We then switch gears and learn some commonly used commands. Commands like 'grep', 'find', 'wc', 'echo', etc. are powerful commands that we should know. Specifically we will work with a large data set of clinical EEGs and query for three cases:

2. *Patient Names whose first names start with R and last names start with S who had an EEG in the date range 2010-13*
3. *EEG reports that contain the word 'spike'. EEG reports that contain the word 'seizure'. We then produce a histogram of the words in these reports.*
4. *For EEG reports that contain the word 'spike' produce a histogram of bi-grams."*

So now instead of using 'grep', 'find' and 'wc' we need to find similar tools in python that accomplish the same results. The one item we will still leave in bash in generating a list of all the file in our "/data/" directory. In the end we will time and compare the results of homework one to the performance seen when using this python approach.

2. APPROACH

To begin we need to find tools that accomplish the same goals in python that we find in bash. To begin we generate a list of files in bash using the same script we used in homework one. Generating a list of file in python is also possible but we ignore this aspect of the assignment for this homework. Since we already have a list of files we are then tasked with reading this list of files into a python list or other type.

After having all the file location stored in memory from running our python script we can begin our analysis of the data. We found that one of the better data types to work with is Pandas. Pandas is similar to having excel functionality within python. Specifically a data frame is a table that can be used within the python coding language. Our approach starts with loading each filename into the first column of a dataframe. To simplify things we house this dataframe within its own class called "wordCounts". The class has functions that will manipulate our list of files within our EEG directory.

The first function in our custom class is read. Read simply reads in a list of files and stores its filename in the first column of dataframe. Read will also take the filename and read in the contents of the file into the second column which we name "contents".

A second function in our class called "dropNotContain" will drop any file whose column "contents" does not contain the "specialWord" argument used when the function is called. The second argument in our function "drop" will specify whether we drop the rows whose contents do not contain the "specialWord". If drop is false we simply only report the number of rows contain the word asked for in the "speicalWord" argument. In our case we will look for a "speicalWord" whose value equals "spike" and drop = "True".

The third function we use is "createHistogram" which will create a histogram of all words in the data

column “contents” over a span of a number word contained in the arugment “span”. So if span = 1 then we will create a one word histogram, if span = 2 then we will create a two word histogram, and so on. This functionality is similar to the “tail –n +2 words | paste words | sort | uniq –ic | sort –nr >subseta_bi.hist” command we would use in bash to create a two word histogram. However in python we use the string functions “join” to combine multiple column “contents” together. Then we split each word into a list of words. If necessary we can combine to adjoining words together again using the “join” string library function. Lastly we take advantage of the collections module to find a histogram of the words inside our resulting dataframe.

What is interesting is whether the python or bash result is faster. We will find out.

3. RESULTS

The main driver program for homework eight is very simplistic. It first reads in our list of files we generated using a bash command. The list of EEG files is loaded into our class and we call two functions. The first function is to drop all files whose “contents” column does not contain the word spike. The second function is a to generate our one word histogram. We then use the same function to generate our two word histogram.

```
#!/usr/bin/env python
import os
import sys

# Import Class
#
from foo_00 import wordCounts


def main(argv):

    # Create class object
    #
    myList = wordCounts()

    # Read in file
    #
    myList.read(argv[1])
    #myList.saveData('eegs.pkl')
    myList.dropNotContain("Spike", True)
    myList.createHistogram(1)
    myList.createHistogram(2)
    #myList.myPrint()

    # Exit Gracefully
    #
    print("Program Finished successfully.")


# Run main if tihs is ran as main function.
if __name__ == "__main__":
    main(sys.argv)
```

Figure 1: foo.py – Main driver program

The real brains of the program lies within our function class “wordCounts” contained in the file foo_00.py. As stated before we have three core functions: “read(), dropNotContain(), createHistogram()”

that we use in our main function. We also include a few additional functions that allow us to print the contents of the class main dataFrame that contains the all filenames and the contents of those files. Another function allows us to save our dataFrame to a file of type pickle. A pickle is a dataFrame saved to a file (although it can be used to save lists and dictionaries in python) so that we do not have to read in our filenames and the contents of those files every time we run our program. We will later see how the three core functions perform with disregard to the time taken to read in the filenames and their contents. The class and its functions are listed below in separate parts. First the class declaration and initialization:

```
import os
import sys
from collections import Counter
import pandas as pd
import csv

# wordCounts class
#
class wordCounts():

    # Initialize Variables
    #
    def __init__(self):
        # Define the column names
        #
        self.cols = ['Filename', 'Contents']

        # Initialize dataframe
        #
        self.massWord = pd.DataFrame(columns=self.cols)
```

Figure 2: foo_00.py – Class declaration and initialization

The above show the class declaration and initialization. Pretty standard stuff expect note we are using dataframes. We also define our two columns with the names “FileName and “Contents”.

```
def read(self, fileName):
    # Check if file exsits
    #
    try:
        file = open(fileName, 'r')

    # Run this code if the open function throws an IOError
    #
    except IOError:
        print('There was an error opening the file \"+fileName+'\")
        sys.exit(1)

    # Check if file is a pickle (already created dataFrame). Load
    # the dataFrame directly and skip reading in text file operation
    #
    if (fileName[-3:] == "pkl"):
        self.loadData(fileName)
        return

    # Build the list of files leading/lagging characters especailly
    # that pesky new line character
    #
    listFiles = []
    with open(fileName) as f:
        listFiles = [tmp.strip() for tmp in f.readlines()]

    # Open each file and store the contents of files into a column=contents
    # We also have another column that stores the filename

    # read in each file into a dataframe
    #
    df_list = []
    df_list = [pd.DataFrame({self.cols[0] : str(fName),\
            self.cols[1] : open(fName, 'r').read()},\
            columns=self.cols, index={1})\
            for fName in listFiles]

    # Combine all the dataframes in the df_list into one dataframe
    #
    self.massWord = pd.concat(df_list).reset_index()

    # The combined datframe has an outdate index column that we drop
    #
    self.massWord.drop('index',  axis=1, inplace=True)
```

Figure 3: foo_00.py – Class Read-In Function

The read in function will take in our list of files and populate the two columns “FileName” and “Contents” for us. However the caveat is if the file passed in is a pickle then our function will attempt to read in the dataFrame from the pickle and skip the entire populating “FileName” and “Contents” column file by file and instead use the dataframe generated from a previous run. This process saves computation time since we are not reading in our entire 20,000 EEG files, file by file.

```
def myPrint(self):
    print(self.massWord)
# Save the contents of the dataframe to a file
#
def saveData(self, fName):
    self.massWord.to_pickle(fName)

# Load the contents of a previous run into dataframe
#
def loadData(self,fName):
    self.massWord = pd.read_pickle(fName)
```

Figure 4: foo_00.py – Class print/save/load Function

First function prints the contents of the class dataFrame. Mostly for debugging purposes and for the curious minded. The second function "saveData" allows us to save our class dataframe to a pickle datatype. This pickle can be used to speed up our class in reading in files on sequential runs on the same data. The "loadData"function will allow us to load data from a pickle which we will use in our read function.

```
# This function looks for files that contain the whole special word
# (any case). Appends a column containing true/false.
# If drop == true it will remove any rows that don't contain the special
# word
#
def dropNotContain(self, specialWord, drop):
    # Define our search pattern using regex
    #
    searPat = r'\b'+specialWord+r'\b'

    # Append a column to dataframe containing true/false if the contents
    # column contains the search pattern defined above
    #
    self.massWord['Contains ' + specialWord]\
          = self.massWord[self.cols[1]].str.contains(\
          searPat, case=False, na=False)
    # If drop == true it will remove any rows that don't contain the
    # special word
    #
    if(drop==True):
        self.massWord = self.massWord[self.massWord['Contains ' + \
              specialWord]==True]

    # Print the number of files left
    #
    print("A total of " + str(len(self.massWord[self.massWord['Contains '+\
          specialWord]==True])) + \
         " contain the word " + specialWord + ".")
```

Figure 5: foo_00.py – Class dropNotContain Function

The dropNotcContain() function allows us to drop any rows in our dataFrame whose contents do not contain a specified word. The function works by creating a new boolean column on top of our dataFrame stating True/False whether the row contains the "specialWord". If drop=True then we will eventually drop all columns whose "specialWordContains" column = False. Then at the end of the function we print out the number of columns whose "Contents" column contained the "specialWord" specified.

```
# Create a span word histogram of all words in the dictionary
#
def createHistogram(self, span):
    # Print to console that we producing a histogram
    #
    print('Producing a ' + str(span) + \
          ' word histogram of dataset.')
    # Using collections we can create histogram.
    # We join together all entry columns containing contents. We also
    # reaplce all non-alphanumeric characters with spaces as not to ruin
    # our results using a regex. If needed we combine words over the span.
    #
    words = " ".join(self.massWord[self.cols[1]].str.lower().replace({\
          '[^0-9a-zA-Z]+': ' '}, regex=True)).split()

    # If we are not doing a histogram of single words we regroup our
    # list of words using another join
    #
    if(span != 1):
        words = [" ".join(words[i:i+span]) for i in xrange(0, \
            len(words), 1)]
    word_counts = Counter(words)

    # We convert the dictionary back to a dataframe allowing us to sort
    # by frequency
    #
    df = pd.DataFrame.from_dict(word_counts, orient='index').reset_index()
    df.sort(columns=[df.columns[1], df.columns[0]],\
          inplace=True, ascending=False)

    # Swap the counts and words columns (to obtain same formatting
    # that base would provide
    #
    df = df[[df.columns[1], df.columns[0]]]

    # Save results to file
    #
    #fileO = open('massWords_'+str(span)+'.hist', 'w')
    #fileO.write(df.to_string(index=False, header=False))
    df.to_csv('massWords_'+str(span)+'.hist', header=False,index=False,
          sep=' ', quoting=csv.QUOTE_NONE, quotechar='', escapechar='\t')
    # Print to console that histogram generation was successful
    #
    print('\tHistogram saved to "massWords_' + str(span) + '.hist".')
```

Figure 6: foo_00.py – Class createHistogram Function

The last function is a specialize class it a “createHistogram()” function whose purpose it to create a histogram from our “Contents” column. The first task is to clean up the “Contents” column so that the only characters that remain are alphanumeric characters. We accomplish this task by using a regular expression that replaces all characters that are not a-zA-z0-9 to nothing (‘’). Next we check the “span” argument to see what type of histogram we are creating. If it is a two word histogram, three word, ect. The we use the collections module to create our histogram from our dataFrame directly. We sort by the value of occurrence of each word (or word combination) and save the results to a txt file. By saving to a txt file we can compare our python results to our bash results.

Full Codebase available on my github: https://github.com/dtrejod/myece3822/blob/master/hw8

4. ANALYSIS

So the real accomplishment is seeing if the python result is faster than the bash result. So first we revist our code from homework one and time the speed to see how fast it performs. Linux has a built in function "time" that allows us to time how long it takes a script to run.

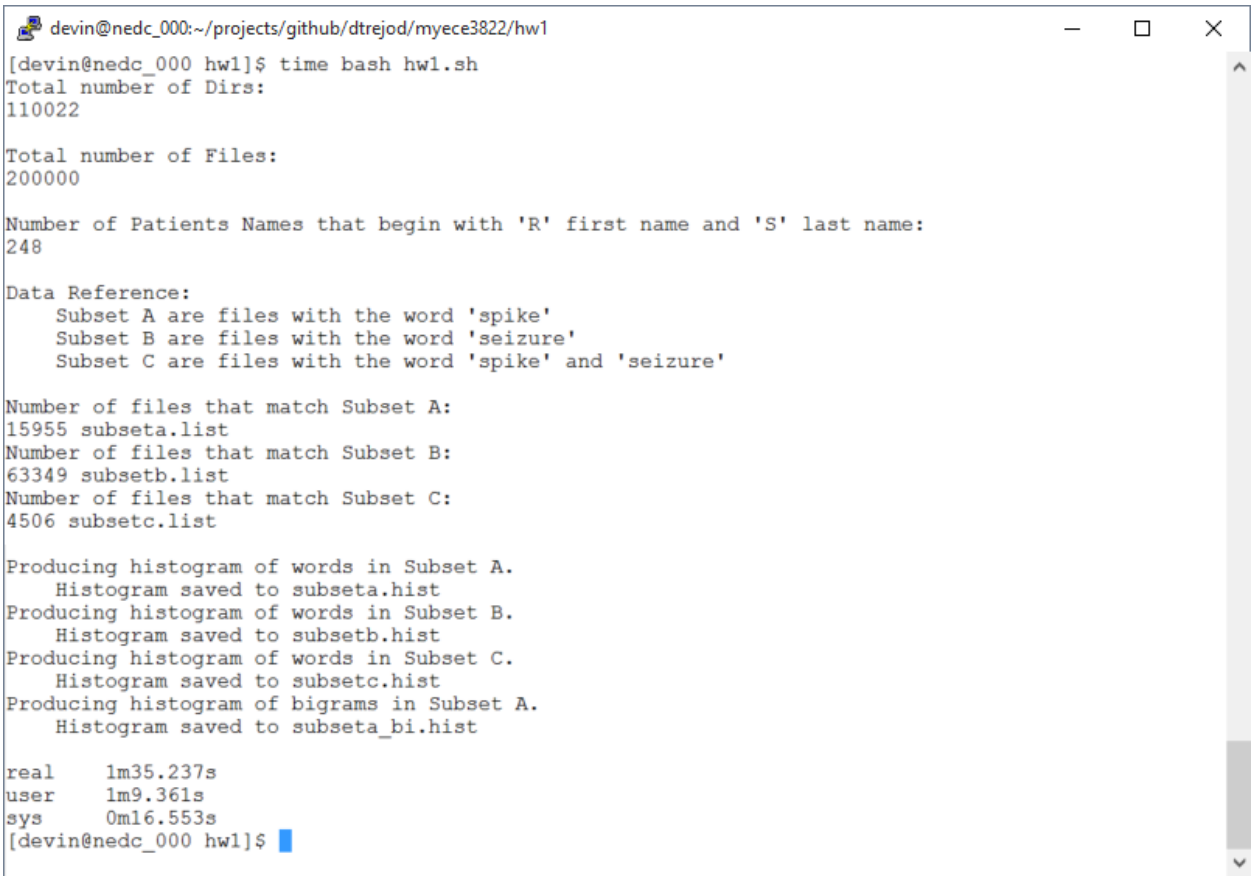

Figure 7: homework1 hw1.sh - RunTime

In the result above we observe how it took ~1min35seconds mins to read in and parse each file in our EEG database. The above result looks for how many files are in our "/data/" directory, parse each file for the work 'spike', parse each file for the word 'seizure' and finally parses each file for the combination of 'spike' and 'seizure'. Lastly it produces a histogram of each one of these file listings and outputs the contents to a file called "subseta.hist", "subset.hist" and "subseta_bi.hist". The first containing the one word histogram for files that conaintaed only the word 'spike'. The second for files that contained the word 'seizure'. The third contained a histogram for file that contain both 'spike' and 'seizure'. Finally the last containing a two word histogram for files containing 'spike'.

Now let us run our python script from reading in a raw list of fileNames that point to the locations of each EEG file. We first need to run our bash script that generates a list of all the files in our "/data/" directory.

Figure 8: createListFiles.sh – RunTime

The above shows that our generating a list of files takes no time at all clocking in at under 1 second. Next we run the main python script that takes in the output of the "createFileList.sh" script and pushes it into a dataFrame reading in each file one at a time. Note that we will save our output file to a pickle to test how it long it takes to run the script if we ignore reading in each file one by one.

Figure 9: homework8 hw1.sh – RunTime from Pickle

Note how the results for the number of files (15995) that contain the word spike match between our python and bash script. This result ensures our two approaches for finding files that contain the word spike are indeed finding the same files. The results are saved to a "*.hist" file as like our bash script. We will latter diff the result and ensure that the two indeed produce the same histograms. The timing of the reading in the raw file name shows how our python script run about 6 seconds slower than our bash script. Keep in mind however, that the bash script is doing more analysis than our python script. Our python script is only parsing files whose contents contain the word 'spike'. Our bash script is producing file listings, and creating histograms for files that contain the word 'spike' as well as 'seizure', 'spike|seizure', and a two word histogram for files that contain the word 'spike'.

The last test is to read of dataFrame from a pickle which already contains the list of "FileNames" and their "Contents". The read in time take a large percentage of the time needed for our python script to run.

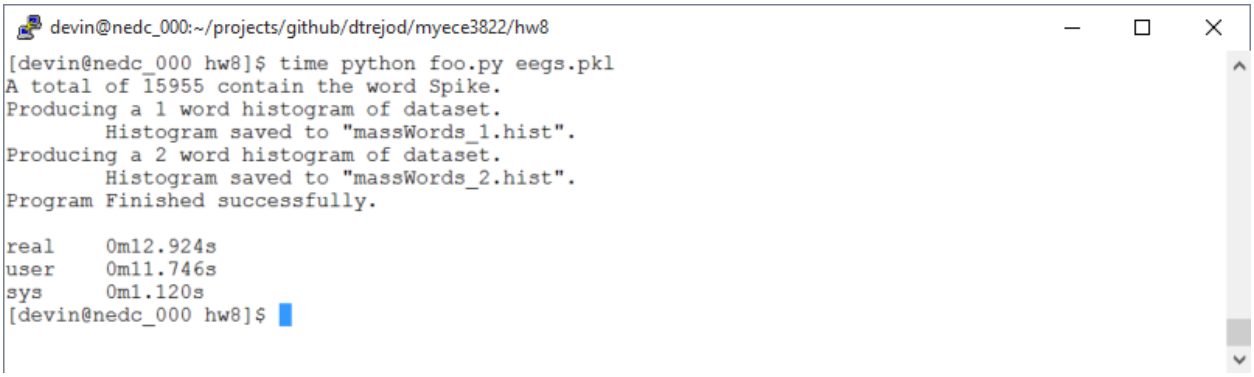

Figure 10: homework8 hw1.sh – RunTime from Pickle

As is seen above reading the files form a pickle drastically improves our runtime. Granted our python script is not finding files and generating histograms for files who contents contain the word "seizure", so we still say the bash script runs faster. However the python performance is not bad all considering we are loading each file into memory unlike in the bash script. The bash script does not load each file into memory but instead reads each file into memory one by one. This procedure is beneficial for desktops/servers who do not have a lot of memory. In my scenario however, I'm running my script on a machine that contains 64GB of memory, thus it is a non-issue.

The last step is to ensure both approaches produce the same result. To accomplish this we use the built in "diff" command in linux. Note that for our two word histogram we also need to use the "sort –nr" command on our two word python histogram results since python sort and bash sort by two difference methods. Another note is that we use the "-wE" flags with the "diff" command to ignore whitespace and tab spacing differences between our python and bash results. First we compare the first word histogram:

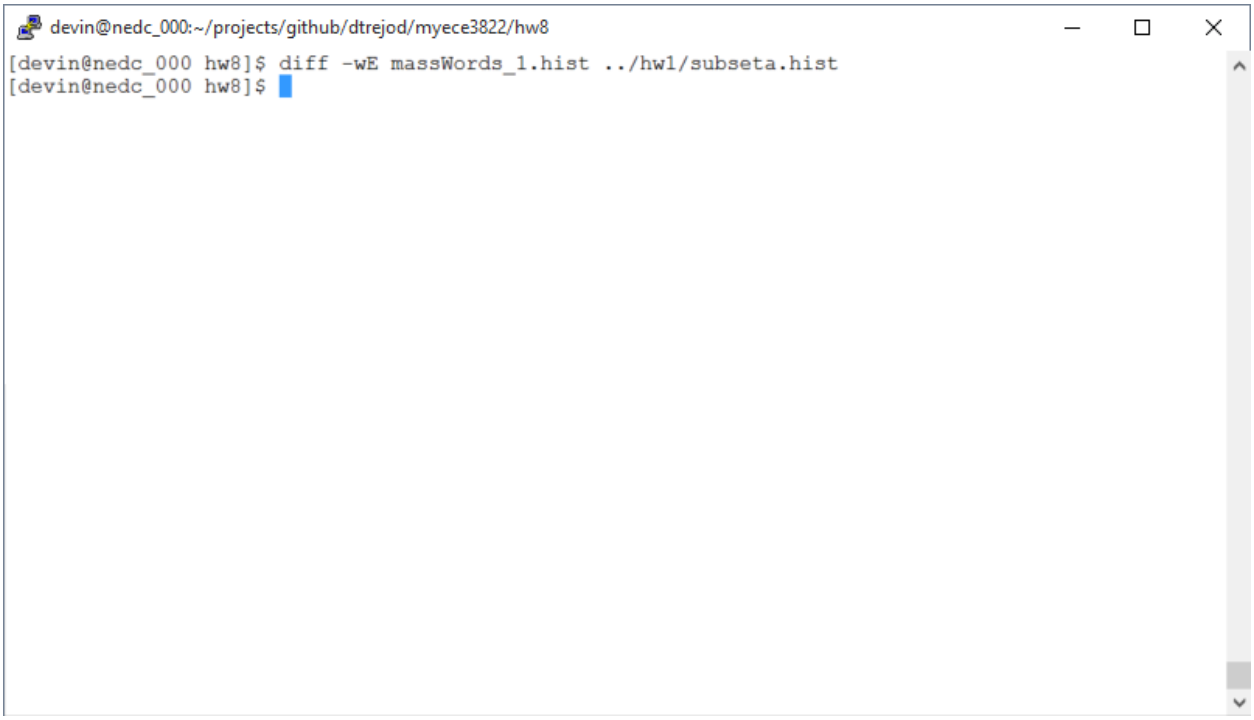

Figure 11: Linux "diff" results between homework1 and homework8 – oneword histogram

The results above produces no differences thus we show that the two have no differences.

Next we compare the two word histogram. As stated before we are require to sort using the bash sort command to ensure the two histograms are organized in the same manner (This sort alogrithum is the same sort we use to sort our 'hw1.sh' 'subseta_bi.hist' histogram):

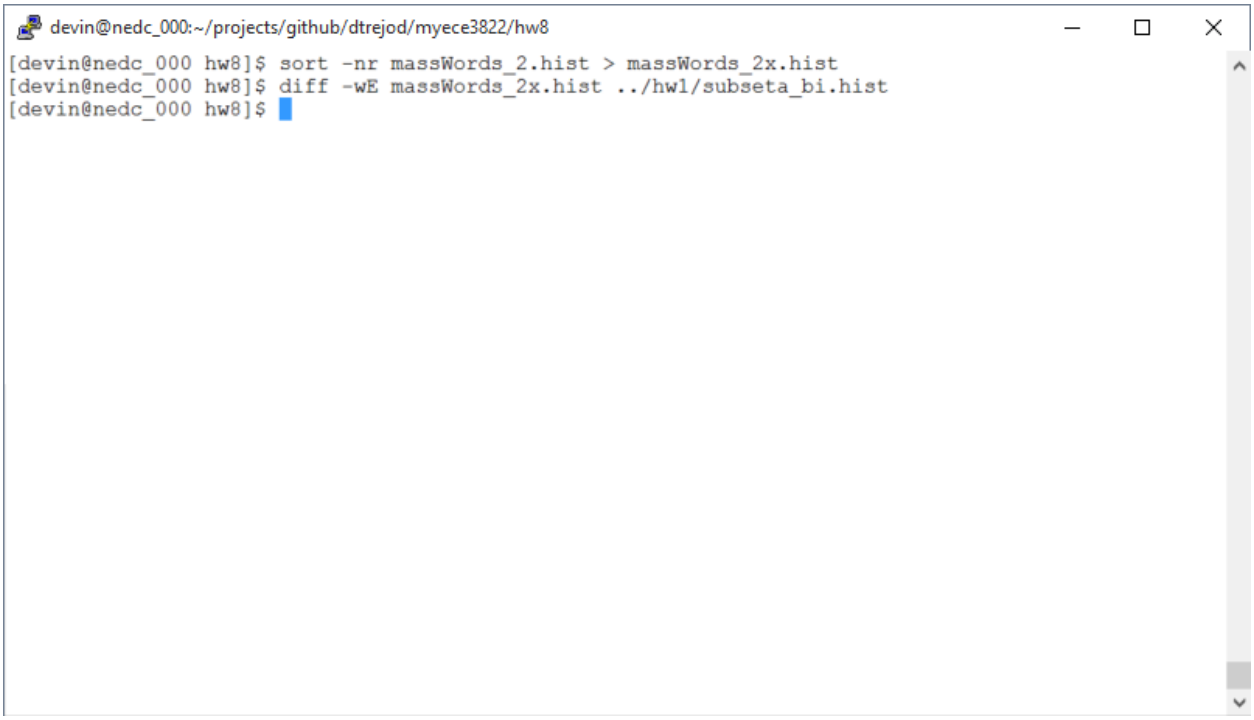

Figure 12: Linux "diff" results between homework1 and homework8 – twoword histogram

Again we see that there are no differences. Thus we conclude our bash and python scripting process produce the same results.