Assignment 1: UNIX, Python and Fast Data
02807 Computational Tools for Big Data

Anonymous authors

September 28^{th} 2015

1 Exercise 1.1

Following is a command that finds the most popular words in the text file `randomtext.txt`

remove all non-alpha char

```
$ tr -c '[:alnum:]' '[\n*]' < randomtext.txt | sort -f| uniq -c | sort -nr | head
-10
```

Gives the output:

```
10 an
9 it
8 to
8 in
7 no
6 out
6 or
6 her
5 you
```

First the `tr` command translates all non-alphanumeric characters to a new line, such that the output of `randomtext.txt` becomes an array of words. Then the `sort` command sorts the lines of the new array (the words) alphabetically where the `-f` flag is for case ignorance. The `uniq` command then filters out repeated lines (words) in the document and the `-c` flag is for keeping track of the number of times the line occurred in the input. As each line now starts with the number of occurrences of the corresponding word, we use the `sort` command again to sort the lines numerically in reverse order, hence the `-nr` flag. Finally we only display the 10 most frequent words in the document with the `head` command.

2 Exercise 1.2

We use the `awk` command to remove the all rows where the price is more than 10,000$ in `data12.txt` and save the result in `expesiveCars.txt`.

```
$ awk '$5>=10000' data12.txt > expensiveCars.txt
```

If we do not save it to the file `expensiveCars.txt` the output is

```
ford mustang 65 45 17000
ford ltd 83 15 10500
ford thundbd 84 10 17000
```

3 Exercise 1.3

The file `shakespeare.txt` includes a screenplay for a Shakespeare drama. In order to be able to check for misspelled words we first translate the document to a document called `shakespearelist.txt` where each line contains a single uniquely represented word from the original document. This is just like in Exercise 1.1, but now we don't care about the number of occurrences and use case insensitive comparison of lines `-i`, such that the words *you* and *You* are considered to be identical.

```
$ tr -c '[:alnum:]'  '[\n*]' < shakespeare.txt | sort -f | uniq -i > shakespearelist.txt
```

Which yields the following output (head) if we do not save it to the file `shakespearlist.txt`

```
A
abandon
abed
Abhor
able
abominable
about
above
Abruptly
```

The following spellchecker takes input from the file `shakespearelist.txt` and outputs a list of words from that file that are not in the dictionary `dict`.

```
$ comm -13i dict shakespearelist.txt
```

By including the option `-13i` we state that we only are interested in the words (case insensitive) that are present in `shakespearelist.txt` and absent in the dictionary but not vice versa. We pipe the previous command with the word count command `wc` to count the number of unique words not represented by the dictionary.

```
$ comm -13i dict shakespearelist.txt | wc
```

This yields the following output

```
722 721 5550
```

And we therefore see that 721 lines are only present in the `shakespearelist.txt`, which is expected.

4 Exercise 1.4

A t2.micro instance was launched on Amazon EC2, files created and git installed.

5 Exercise 1.5

We created files locally on our machine, pushed them to the Git-hub repository, cloned that repository to EC2 instance, made changes, pushed the changes to Git-hub and pulled finally the changes on our local machine.

6 Exercise 2.1

We start by defining the `filetolist` function which takes `filename` as input file and returns the list of lists called `super_list`

```
def filetolist(filename):
    with open(filename) as f:
        super_list = []
        for line in f:
            line = line.split()
            if line:
                line = [int(i) for i in line]
                super_list.append(line)
    return super_list
```

The function simply splits the file up into lines and converts each line to list, which then is appended to the `super_list`. The output of `matrix_in = filetolist('matrix.txt')` is following

```
[[0, 1, 1, 3, 0], [0, 2, 3, 4, 10], [8, 2, 2, 0, 7]]
```

Now we define the function `listtofile` that takes a list of lists `array` and saves it to the file `filename`.

```
def listtofile(array,filename):
    s = open(filename, 'w+')
    string = ''
    for k in range(len(array)):
        string = string + " ".join(map(str, array[k])) + '\n
            '
    s.write("%s\n" % string)
```

By using the function call: `listtofile(matrix_in,'matrix_copy'` the previously generated list of list is saved to the previous format.

7 Exercise 2.2

Following is the function `allbinarycomb(N)` that takes an integer `N`, and outputs all bit-strings of length `N` as lists, without using the bin function.

```
def allbinarycomb(N):
    binni = []
    for i in range(2**N):
        binstr = '{0:0'+str(N)+'b}'
        binni.append(map(int,list(binstr.format(i))))
    return binni
```

The function declares a super list `binni` and constructs a sub list `binstr` that is defined as a list of integers that for a corresponding binary number. The function call `allbinarycomb(3)` then yields the following results

```
[[0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1], [1, 0, 0], [1,
    0, 1], [1, 1, 0], [1, 1, 1]]
```

This looks like the expected result, where the number of lists int the list is $2^N = 2^3 = 8$.

8 Exercise 2.3

We started by importing the data as **data** from the .json file to python and printed the first field of the data for validation

```
import json
from pprint import pprint

with open('pizza-train.json') as json_file:
    data = json.load(json_file)
pprint(data[0]['request_text'])
```

Which gave the following output

```
u'Hi I am in need of food for my 4 children we are a
    military family that has really hit hard times and we
    have exahusted all means of help just to be able to feed
    my family and make it through another night is all i ask
```

```
    i know our blessing is coming so whatever u can find in
    your heart to give is greatly appreciated'
```

That matched the expected string, but we noticed that the data was unicode, so that we converted it to string that we then converted to list of sub-strings(words) for each object

```
import unicodedata
import re
dic = []
wordList = []
for i in range(len(data)):
    dic.append(unicodedata.normalize('NFKD', data[i]['
        request_text']).encode('ascii','ignore'))
    wordList.append(re.sub("[^\w]", " ",  dic[i]).split())
```

The `wordList` is therefore a list of lists that includes a list of the words for each object. Now that we have a list for each object, we need to count how many times each word occurs in each object.

```
from collections import Counter
bag =[] # bag holding a dictionary for each object
counts = Counter()
words = re.compile(r'\w+')
for i in range(len(data)):
    temp = Counter()
    for sentence in wordList[i]:
        counts.update(words.findall(sentence.lower()))
        temp.update(words.findall(sentence.lower()))
    bag.append(temp)
```

Here the `counts` includes all the individual words of all the `'request_text'` fields and their corresponding occurrence. The `bag` is a list of counters for each object that includes its words and corresponding occurrence. We now convert the keys from the counter object `counts` to a dictionary object that we call `dictionary`, as we do not care about the overall occurrence of each word.

```
a = dict()
a.update(counts)
dictionary = a.keys()
```

Finally we construct a bag of words representation of each string by constructing the zero matrix `M` and for each string we fill in the values of the corresponding indexes of the matrix

```
import numpy
M = numpy.zeros(shape=(len(bag),len(dictionary)))

for j in range(len(bag)):
    for i in range(len(bag[j])):
        M[j,dictionary.index(bag[j].keys()[i])] = bag[j].
            values()[i]
M.shape
```

which gives

```
(4040, 12627)
```

M has therefore `rows = len(bag) = 4040` and `columns = len(dictionary) = 12627` which is expected. For further validation we check if the sum of values adds up

```
M.sum() == sum(counts.values())
```

which yields

```
True
```

Finally we check if the values of `M` are as expected. Locate a string:

```
bag[2].keys()[1]
```

out:

```
'taxi'
```

The corresponding value

```
bag[2].values()[1]
```

out:

```
1
```

and the corresponding index in dictionary is therefore

```
dictionary.index(bag[2].keys()[1])
```

out:

```
1913
```

We check if the value of matches

```
M[2,1913]
```

out

```
1.0
```

It does, so we are pretty happy.

9 Exercise 3.1

We start by importing `numpy` and import the `loadtxt` package. Then we use the package to load `XAhwshXe.txt` to pyton as the `numpy.ndarray` called `H`.

```
import numpy as np
from numpy import loadtxt
H = loadtxt("XAhwshXe.txt", delimiter=",", unpack=False)
```

As `b` is the last column of the input-matrix `H` and `A` the rest of the columns we define those as

```
A = H[:,:-1]
b = H[:,-1]
x = np.linalg.solve(A,b)
```

and use the solve function from numpy.linalg to find the solution x. This numpy.linalg to find the solution x. This results in the numpy.ndarray

```
array([-5.09090909,  1.18181818,  2.24242424])
```

10 Exercise 3.2

First we define the function getarray that simply takes each line of the filename and converts to a list of the coordinates, which then are appended to the list coordinates. Finally the list of lists is converted to an numpy.array and returned.

```
def getarray(filename):
    with open(filename) as f:
        coordinates = []
        for line in f:
            line = line.split() # to deal with blank
            if line:            # lines (ie skip them)
                line = [float(i) for i in line]
                coordinates.append(line)
    return np.asarray(coordinates)
```

In order to be able to fit these coordinates with a polynomial of 3^{rd} degree we import interpolate packages from scipy

```
import scipy
from scipy import interpolate
from scipy.interpolate import interp1d
```

Then we call the getarray function and define the first column as x and the second as y

```
dotsArray = getarray('ENyYffaq.txt')
x =dotsArray[:,0]
y =dotsArray[:,1]
```

Then we use the interp1d package to interpolate the coordinates

```
f2 = interp1d(x, y, kind=3)
```

In order to visualize the function we plot it as illustrated in Figure 1.

```
xnew = np.linspace(-20, 19, num=411, endpoint=True)
import matplotlib.pyplot as plt
plt.plot(x, y, 'o', xnew, f2(xnew), '--')
plt.legend(['data', 'cubic'], loc='best')
plt.savefig('ex321.eps')
```

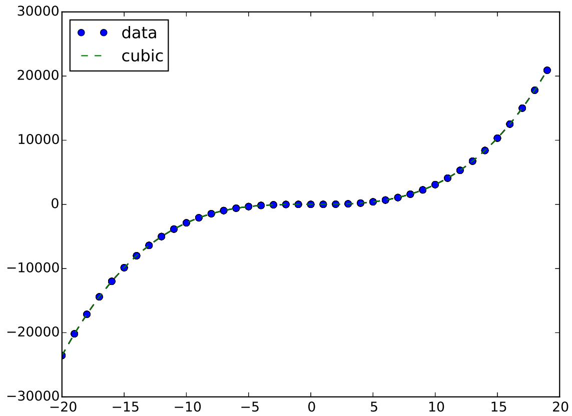

Figure 1:

Finally we find the real root of the polynomial numerically using Scipy's optimization function `brentq`

```
from scipy import optimize
from scipy.optimize import brentq
r = brentq(f2,-20,19)
```

Which yields the output

```
print("%.4f" % r)
-1.4134
```

11 Exercise 3.3

We first imported `pandas` and the used its $read_table$ function to read in the three different `.dat` files.

```
movies = pandas.read_table('ml-1m/movies.dat',sep='::',names
    =['movie id', 'title', 'genre'])
ratings = pandas.read_table('ml-1m/ratings.dat',sep='::',
    names = ['user id', 'movie id', 'rating', 'timestamp'])
users = pandas.read_table('ml-1m/users.dat',sep='::',names =
     ['user id', 'gender', 'age', 'occupation code', 'zip'])
```

Then we use the data combining tool `merge` to combine the three objects from above

```
movie_data = pandas.merge(pandas.merge(users,ratings),movies
    );
```

Now we use this merged object to find the 5 movies with the most number of ratings

```
counts = pandas.DataFrame(movie_data['title'].value_counts()
    ,columns = ['nRatings'])
counts['title']=counts.index
counts.head(5)
```

which yields the following output

```
                                                              nRatings

American Beauty (1999)                                        3428
Star Wars: Episode IV - A New Hope (1977)                     2991
Star Wars: Episode V - The Empire Strikes Back (1980)         2990
Star Wars: Episode VI - Return of the Jedi (1983)             2883
Jurassic Park (1993)                                          2672
```

Now we construct an object called `active titles` that is a subset of the `movie data` that only includes movies having at least 250 ratings.

```
active_titles = pandas.merge(counts,movie_data)
active_titles = active_titles[active_titles.nRatings>=250]
active_titles.tail(5)
```

Now we use the `pivot` method for finding the 3 movies with the highest average rating for females and males respectively.

```
table = pandas.pivot_table(active_titles, values='rating',
    index=['movie id'], columns=['gender'], aggfunc=np.mean)

topF = table.sort(['F'], ascending = 0).head(3)
topM = table.sort(['M'], ascending = 0).head(3)
print topF.F
print topM.M
```

The highest average for female is

```
Close Shave, A (1995)                              4.644444
Wrong Trousers, The (1993)                         4.588235
Sunset Blvd. (a.k.a. Sunset Boulevard) (1950)      4.572650
```

The highest average for male is

```
Godfather, The (1972)                                4.583333
Seven Samurai (The Magnificent Seven) (1954)         4.576628
Shawshank Redemption, The (1994)                     4.560625
```

We then found the 10 movies women liked much more than men

```
maxdiffF = table.sort(['diff'], ascending = 1).head(10)
maxdiffF.F
```

```
Dirty Dancing (1987)                       3.790378
Jumpin Jack Flash (1986)                   3.254717
Grease (1978)                              3.975265
Little Women (1994)                        3.870588
Steel Magnolias (1989)                     3.901734
Anastasia (1997)                           3.800000
Rocky Horror Picture Show, The (1975)      3.673016
Color Purple, The (1985)                   4.158192
Age of Innocence, The (1993)               3.827068
Free Willy (1993)                          2.921348
```

and the 10 movies men liked more than women

```
maxdiffM = table.sort(['diff'], ascending = 0).head(10)
maxdiffM.M
```

```
Good, The Bad and The Ugly, The (1966)      4.221300
Kentucky Fried Movie, The (1977)            3.555147
Dumb & Dumber (1994)                        3.336595
Longest Day, The (1962)                     4.031447
Cable Guy, The (1996)                       2.863787
Evil Dead II (Dead By Dawn) (1987)          3.909283
Hidden, The (1987)                          3.745098
Rocky III (1982)                            2.943503
Caddyshack (1980)                           3.969737
For a Few Dollars More (1965)               3.953795
```

One could imagine that girls tend to like Dirty Dancing much better than guys as well as guys can probably relate to the plot of Dumb and Dumber in much higher degree.
The 5 movies that had the highest standard deviation in rating were found by the following pivot table, where we use the `std` function from `numpy` for the ratings of the active titles

```
tableStd = pandas.pivot_table(active_titles, values='rating'
    , index=['title'], aggfunc=np.std)
tableStd.nlargest(5)
```

that yields the following output

```
title
Dumb & Dumber (1994)                       1.321333
Blair Witch Project, The (1999)            1.316368
Natural Born Killers (1994)                1.307198
Tank Girl (1995)                           1.277695
Rocky Horror Picture Show, The (1975)      1.260177
```

```
'Would be cool to validate!'
```

12 Exercise 3.4

We start by getting the data from the json file

```
import json
with open('pizza-train.json') as json_file:
    data = json.load(json_file)
```

Then we visualize a value of the request_text field for one of the objects to make sure that we are returning the right field.

```
from pprint import pprint
pprint(data[1]['request_text'])
```

Which it does:

```
u'I spent the last money I had on gas today. Im broke until
   next Thursday :('
```

Now we introduce clean_train_reviews as a list that includes all the request_text fields from the .json file.

```
clean_train_reviews = []
for x in range(0,len(data)):
    clean_train_reviews.append(data[x]['request_text'] )
```

Now we import the CountVectorizer function from scikit-learn as vectorizer function to convert the list of fields to a sparse feature matrix of words called train_data_features

```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer(analyzer = "word",   \
                             tokenizer = None,    \
                             preprocessor = None, \
                             stop_words = None,   \
                             max_features = 50000)
train_data_features = vectorizer.fit_transform(
    clean_train_reviews)
```

We check the dimensionality of this sparse matrix

```
train_data_features.shape
(4040, 12593)
```

And therefore we see that the data includes 4040 objects and 12593 distinct words. Now we convert the train_data_features to array

```
train_data_features = train_data_features.toarray()
```

and as the training set should only be 90% of the data we calculate that amount and define the first 90% of the data as training_features.

```
size_training_set = np.multiply(0.9,len(data)).astype(int)
training_features = train_data_features[:size_training_set,:
    ]
training_features.shape
```

Which yields the output

```
(3636, 12593)
```

Now that we have the training features ready, we get the corresponding boolean labels from the data and define as train_labels

```
clean_train_labels = []
for x in range(0,len(data)):
    clean_train_labels.append(data[x]['
        requester_received_pizza'] )
train_data_labels = np.asarray(clean_train_labels)
train_labels = train_data_labels[:size_training_set]
```

Now we have both the bag of words as well as the lables corresponding to each request text we are ready to fit a logistic regression model. Therefore we import the linear_model from scikit-learn and define the fit as logreg

```
from sklearn import linear_model, datasets

logreg = linear_model.LogisticRegression(C=1e5)
logreg.fit(training_features,train_labels)
```

Now that the fit is ready we need to define the test features in order to validate the model/fit.

```
test_features = train_data_features[size_training_set+1:,:]
```

Now we use the model logreg to predict the boolean value of requester_received_pizza for the 'test' features

```
predict_labels = logreg.predict(test_features)
```

predicted_labels are therefore the predicted labels for test_features based on the training data. Now we check how good the prediction is by comparing the true values of requester_received_pizza of the test data to the predicted ones

```
true_labels = train_data_labels[size_training_set+1:]
accuracy = np.true_divide(sum(predict_labels==true_labels),
   len(predict_labels))
print("%.4f" % accuracy)
```

Which yields

```
0.6576
```

This means that by using the request_text from the data we can predict if the user received pizza with approximately 66% accuracy, which is better than the default.

13 Exercise 3.5

We made a simple function that calculated the sum of a given function 500 times

```
import numpy as np
def pyfun():
    for j in range(500):
        sum_ = 0
        for i in range(100000):
            sum_ += np.divide(1,float(np.power(float(i+1),2)
                ))
```

The execution time was calculated to be 235.8 seconds.

Then we compiled the code with Cython as

```
def cyfun():
    cdef float value
    cdef int j,i
    for j in range(500):
        value = 0
        for i in range(1,10000):
            value += 1.0/(i*i)
    return value
```

It's execution time was calculated as 0.0368 seconds. Huge differnece.

14 Exercise 4.1

A lot of ways where tried to make the code run faster. Some matrix calculations and list comparisons. However through trial and error we found out that dictionaries had the fastest indexing so we started by transferring all the indices to a dictionary format and then calculated all the length of indices for every point. This decreased the time that took to calculate the Jacardian length dramatically. To the point where running the code for the 10.000 point dataset took only two minutes. However we did not have the time to run it on the biggest dataset due to time constraints. We finished the code too late. Due to arrays where pretty slow indexing, especially big sparce matrixes we used lists to keep track of visited points and unchecked neighbours. The set(a).instersect(b) proved to be the fastest way we found to compare list/arrays.

In the results we print out the time it took as time1, maxval as the total number of clusters. The max(clustercount) says the total number of points in the biggest cluster:

14.1 dataset 1:

time1
0.2395930290222168
maxval
3
max(clustercount)

4.0

14.2 dataset 2:

time1
0.2626159191131592
maxval
5
max(clustercount)
30.0

14.3 dataset 3:

time1
1.513556957244873
maxval
8
max(clustercount)
289.0

14.4 dataset 4:

time1
133.0414900779724
maxval
393
max(clustercount)
2847.0

15 Appendix

Following is the code for those exercises that are not gone through thoroughly in the assignment.

15.1 Exercise 2.1 and 2.2

```
"Exercise 2.1"
def filetolist(filename):
    with open(filename) as f:
        super_list = []
        for line in f:
            line = line.split()
```

```
            if line:
                line = [int(i) for i in line]
                super_list.append(line)
    return super_list

def listtofile(array,filename):
    s = open(filename, 'w+')
    string = ''
    for k in range(len(array)):
        string = string + " ".join(map(str, array[k])) + '\n
            '
    s.write("%s\n" % string)

matrix_in = filetolist('matrix.txt')
matrix_out = listtofile(matrix_in,'matrix_copy.txt')


"Exercise 2.2"
def allbinarycomb(N):
    binni = []
    for i in range(2**N):
        binstr = '{0:0'+str(N)+'b}'
        binni.append(map(int,list(binstr.format(i))))
    return binni
```

15.2 Exercise 4

```
import cPickle as pickle
from scipy.sparse import *
from scipy import *
import numpy as np
import time
import querycalculate
from multiprocessing import Process


X1 = pickle.load(open('data_10points_10dims.dat', 'r'))
X2 = pickle.load(open('data_100points_100dims.dat', 'r'))
X3 = pickle.load(open('data_1000points_1000dims.dat', 'r'))
X4 = pickle.load(open('data_10000points_10000dims.dat', 'r')
    )
X5 = pickle.load(open('data_100000points_100000dims.dat', 'r
    '))


Y1 = csr_matrix(X1)
Y2 = csr_matrix(X2)
Y3 = csr_matrix(X3)
Y4 = csr_matrix(X4)
Y5 = csr_matrix(X5)
```

```
def expandCluster(P, neighbours, C, eps, M, checked):
    cluster[P] = C
    while True:
        #uncheckedneighbours = querycalculate.diff_arr(
            neighbours,checked)
        uncheckedneighbours = list(set(neighbours) - set(set
            (checked).intersection(neighbours)))
        if len(uncheckedneighbours) == 0:
            break
        P = uncheckedneighbours[0]
        checked.append(P)
        neighboursm = calN(pos,lenp,P,lenY,eps)
        if len(neighboursm) >= M:
            neighbours = list(neighbours) + list(set(
                neighboursm) - set(neighbours))
        cluster[P] = C



def calN(pos,lenp,p,lenY,eps):
    N = []
    for i in range(lenY):
        inter = len(set(pos[i]).intersection(pos[p]))# set(
            pos[i]+pos[p]).__len__()*2 + lenp[p]+lenp[i]
        Jlen = 1.0 - float(inter)/(lenp[p]+lenp[i]-inter)
        if Jlen <= eps:
            N.append(i)
    return N



# Main run

t1 = time.time()
Y = Y4
Yt = Y4t
cluster = {}
checked = []
lenY = shape(Y)[0]
C = 0
M = 2
eps = 0.15
pos = {}
for i in range(lenY):
    pos[i] = list(Y[i].indices)
lenp = {}
```

```
for i in range(lenY):
    lenp[i] = shape(Y[i].indices)[0]


while True:
    #Pleft = querycalculate.diff_arr(range(lenY),checked)
    Pleft = list(set(range(lenY)) - set(set(checked).
        intersection(range(lenY))))
    if len(Pleft) == 0:
        t3 = time.time()
        break
    P = Pleft[0]
    checked.append(P)
    neighbours = calN(pos,lenp,P,lenY,eps)
    if len(neighbours) < M:
        cluster[P] = -1
    else:
        C = C + 1
        expandCluster(P, neighbours, C, eps,M, checked)

t2 = time.time()

time1 = t2-t1
time1
```