Assignment 1

02807 Computational Tools for Big Data

ANONYMOUS AUTHOR

Contents

Unix

I will present first the whole pipeline, and then go through the output and explain the individual commands that make up the pipeline.

Exercise 1.1

The solution

I chose Alice in Wonderland as the textfile. Find the 10 most common words is solved with the pipeline

```
tail -n +31 alice.txt | tr '[:blank:]' '\n' | tr -cd '[:alpha:]\n' | tr '[:
    upper:]' '[:lower:]' | sort | uniq -c | sort -nr | head -11
```

Output

```
   2319
   1670 the
    840 and
    791 to
    667 a
    598 of
    499 she
    492 it
    456 said
    409 in
    400 I
```

Explanation

I got Alice in Wonderland from Project Gutenberg. It has a header that needs to be removed first.

```
tail -n +31
```

Move every word to its own line.

```
tr '[:blank:]' '\n'
```

Remove punctuation and everything else that isn't a letter, or a newline, by selecting and then negating and deleting.

```
tr -cd '[:alpha:]'\n'
```

Convert to lower case.

```
tr '[:upper:]' '[:lower:]'
```

Sort the list to prepare it for uniq.

```
sort
```

Count lines of the same word. Ignore case.

```
uniq -c
```

Sort by number of occurences (in reverse order).

```
sort -nr
```

Print 10 most common words (-11 because the first line is actually empty lines being counted, and that shows up in the output).

```
head -11
```

Exercise 1.2

The solution

This exercise is solved with `awk`. It is self-explanatory. Display lines in cars.txt with price less than 10000:

```
awk '$5 < 10000 {print $n}' cars.txt
```

Output

```
plym    fury    77      73      2500
chevy   nova    79      60      3000
volvo   gl      78      102     9850
Chevy   nova    80      50      3500
fiat    600     65      115     450
honda   accord  81      30      6000
toyota  tercel  82      180     750
chevy   impala  65      85      1550
ford    bronco  83      25      9525
```

Exercise 1.3

The solution

Spell check: Count the number of unique words in shakespeare.txt that do not appear in the dict. This exercise is solved with the following pipelines:

```
cat dict | tr '[:upper:]' '[:lower:]' | sort > dictSorted
cat shakespeare.txt | tr '[:upper:]' '[:lower:]' | tr '[:blank:]' '\n' | sed
    "/['-]/d" | tr -cd "[:alnum:]\n" | sort | uniq > shakespeareProcessed.
   txt
comm shakespeareProcessed.txt dictSorted -23 | wc -w
```

This is a conservative spell-check. It removes words with dashes and apostrophes. It marks 686 words as incorrectly spelled.

Output

```
686
```

Explanation

Convert dict to lower case. This is used both for dict and shakespeare.txt.

```
tr '[:upper:]' '[:lower:]'
```

Apparantly the dict is not already sorted according to the rules followed by sort, so sort it.

```
sort dict
```

Split words to their own lines.

```
tr '[:blank:]' '\n'
```

Remove entire words if they include dashes or apostrophes. In effect, this is a conservative spell-check.

```
sed "/['-]/d"
```

Remove punctuation from words.

```
tr -cd "[:alnum:]\n"
```

Sort the list of words in shakespeare.txt and remove duplicates.

```
sort
uniq
```

Compare and output the number of words not found in the dict.

```
comm shakespeareProcessed.txt dictSorted -23
```

Count number of words.

```
wc -w
```

Git & Amazon EC2

Exercise 1.4 & 1.5

I went to Github and created a new repo for this course. I cloned the repo into a folder on my own PC, created some files and pushed it back to Github. In summary I used the commands

```
git clone https://github.com/<username_removed>/CompToolsForBigData
cd CompToolsForBigData
cat > index.html
Hello 02807!
git add index.html
git commit -m "Added index.html"
git push origin master
```

Then I created an instance located on the AWS server in Frankfurt. I generated key-pairs and opened for SSH and HTTP access. I logged into the instance using the SSH command

```
ssh -i "myKeyPair.pem" ec2-user@52.28.133.138
```

I updated pre-installed packages using

```
sudo yum -y update
```

I installed Python-3.4.3, Git and Apache using

```
sudo yum install python34
sudo yum install git
sudo yum install httpd
```

Then I cloned the github repo into my `home/ec2-user` folder, again using

```
git clone https://github.com/<username_removed>/CompToolsForBigData
```

For the fun of it, I want to host a webserver with that file. Rather than try to point Apache to the CompToolsForBigData folder, I copy the index.html file manually this time.

```
sudo cp index.html /var/www/http/index.html
```

The Apache server is launched by

```
sudo service httpd start
```

Finally, my simple website is now hosted at http://52.28.133.138 !

Python

Exercise 2.1

My python script takes the filename of the matrix file as the first argument on the command line. I use the matrix 3x5 matrix from the link in the exercise description. There is no magic happening in this script. There is some sting operations like trimming and splitting, and a conversion from string to int (the matrix holds ints). I did use a list comprehension for that conversion. A `map` would probably do as well.

Output

The output is:

```
The matrix was saved internally as:
[[0, 1, 1, 3, 0], [0, 2, 3, 4, 10], [8, 2, 2, 0, 7]]
Printing the matrix to file exercise-2.1-OUTPUT
```

Reading the matrix from exercise-2.1-OUTPUT using `cat` we get:

```
0 1 1 3 0
0 2 3 4 10
8 2 2 0 7
```

Which is identical to the file containing the input matrix.

The code

```
""" Exercise 2.1 """

import sys


def main():
    # Get filenames from command line
    n_arg = len(sys.argv)
    if n_arg > 1:
        inputFile = sys.argv[1]
    else:
        print('Missing input filename argument.')
        sys.exit()
    outputFile = 'exercise-2.1-OUTPUT'

    # Read in the matrix:
    M = readMatrixFromFile(inputFile)

    # Prove that we actually got M by printing it:
    print('The matrix was saved internally as:')
    print(M)

    # Store the matrix to disk in original format:
    print('Printing the matrix to file', outputFile)
    saveMatrixToFile(M, outputFile)


def readMatrixFromFile(inputFile):
    # Reads in a matrix from a file, assuming that the elements are ints.
    # Returns the matrix as an array of arrays.
    M = []
    with open(inputFile, 'r') as f:
        for line in f.readlines():
            stringlist = line.strip().split(' ')
            intlist = [int(s) for s in stringlist]
            M.append(intlist)
    return M


def saveMatrixToFile(M, outputFile):
    # Write the matrix to a file
    with open(outputFile, 'w') as f:
        for row in M:
            for column in row:
                f.write(str(column) + ' ')
            f.write('\n')

if __name__ == '__main__':
    main()
```

Exercise 2.2

Realizing that there are 2^n possible permutations, I perform 2^n iterations with a counter starting from 0. Then I convert that counter to binary, pad it with some zeros in front so that it has length n (using the method `zfill`), and convert it to a string separated by commas, and wrap it in `[` and `]`. Then I iterate through the 2^n possibilities. I write the output to a file (as we go).

Afterthought: I know that this can be accomplished by recursive code. However I typed up all this stuff before the exercise was changed to rule out `bin()`, and I don't feel like spending time changing it now.

Output

Calling the script with $n = 3$, we get:

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

I didn't bother trimming the trailing comma.

The code

```
import sys

n = int(sys.argv[1])
outputFile = 'exercise-2.2-OUTPUT'

with open(outputFile, 'w') as f:
    for i in range(2 ** n):
        i_binary = bin(i)[2:].zfill(n)
        s = '[' + ','.join(i_binary) + '],'
        f.write(s)
    f.write('\n')
```

Exercise 2.3

I convert to lower-case, use RegEx to select words consisting of characters a-z and apostrophes. Those are then put into a dictionary using the word itself as the key, and the number of appearances as the value.

Finally I sort it (using a lambda function for the sort key), and print the 10 most used words using the unix `head` command. See output below. The word pizza just barely makes it into the top 10!

Output

```
i 12516
a 9899
to 9881
and 9843
the 7307
my 6592
of 4892
for 4868
in 4303
pizza 3720
```

The code

```
import json
import re

with open('data/pizza-train.json', 'r') as f:
    data = json.load(f)

myDict = {}

for item in data:
    req_text = item['request_text'].lower()
    p = re.compile("[a-z][a-z']*")
    for match in p.finditer(req_text):
        word = match.group()
        if word in myDict:
            myDict[word] += 1
        else:
            myDict[word] = 1

# Time to sort it
myList = [(key, myDict[key]) for key in myDict]
myList.sort(key=lambda x: x[1])
myList.reverse()

with open('exercise-2.3-OUTPUT', 'w') as f:
    for entry in myList:
        f.write(entry[0] + ' ' + str(entry[1]) + '\n')
```

Python Libraries

Exercise 3.1 (numpy):

I re-used some code from the second week 2 to load the matrix. It is then converted to a numpy array using `np.array()` and subsequently split into A and b. I solve the system using `np.linalg.solve()` as suggested. Finally I verify the solution by using `np.dot()` to multiply A and x.

Output

```
A =
 [[ 1  2  3]
 [ 6  9 12]
 [ 2  0  9]]
b =
 [ 4  7 10]
x =
 [-5.09090909  1.18181818  2.24242424]
A dot x =
 [  4.   7.  10.]
```

Code

My full code for this exercise is

```
import sys
import numpy as np


def readMatrixFromFile(inputFile):
    # Reads in a matrix from a file, assuming that the elements are ints.
    # Returns the matrix as an array of arrays.
    M = []
    with open(inputFile, 'r') as f:
        for line in f.readlines():
            stringlist = line.strip().split(' ')
            intlist = [int(s) for s in stringlist]
            M.append(intlist)
    return M

# Read the matrix from a file and put it in a list of lists format
matrix = readMatrixFromFile('data/matrix')

# Convert to a numpy array
npmatrix = np.array(matrix)

# A is everything except the last column
A = npmatrix[:, 0:-1]
print('A =\n', A)

# A is the last column
b = npmatrix[:, -1]
print('b =\n', b)

# Solve the linear system to find x
x = np.linalg.solve(A, b)
print('x =\n', x)

# Verify solution
b_ver = np.dot(A, x)
print('A dot x =\n', b_ver)
```

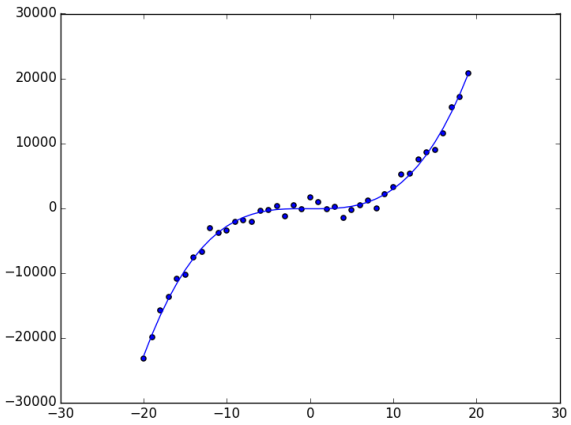

Figure 1: $2.999x^3 + 1.001x^2 - 2.007x + 3.918$. Note: While working on this exercise I manually introduced some noise into the data. This plot shows that noise and a fit to the data with noise. It makes it a little easier to see that the fit actually works, because otherwise the line simply goes through all the points.

Exercise 3.2 (scipy):

I solve this exercise using `curve_fit()`. This is a bit more of a general approach, since I know that there are specific functions for finding polynomial coefficients. Learning to use this function however, allows me to fit any function of my choosing in the future. As such, I am finding parameters `a,b,c,d` to fit `a*x**3 + b*x**2 + c*x + d`.

Then in order to find the roots, I simply use SciPy's `fsolve`.

Output

I find coefficients and roots:

```
Coefficients: [ 2.99999264  1.00106185 -2.00736185  3.91800201]
Roots: [ 0.3740314   0.37403139  0.37403139]
```

corresponding to the polynomial $2.999x^3 + 1.001x^2 - 2.007x + 3.918$. We can plot the function using `MatplotLib` too. See Figure 1. The code follows below.

Code

```
import numpy as np
from matplotlib import pyplot as plt
from scipy.optimize import curve_fit
from scipy.optimize import fsolve

x_data = []
y_data = []

with open('xypoints.txt', 'r') as f:
    for line in f.readlines():
        (x, y) = line.strip().split(' ')
        x_data.append(float(x))
        y_data.append(float(y))

def f(x, a, b, c, d):
    return a*x**3 + b*x**2 + c*x + d

x_data = np.array(x_data)
y_data = np.array(y_data)

# Introduce some noise into y_data to make it a bit more fun
# y_data = y_data + np.random.randn(len(y_data)) * 1000

popt, pcov  = curve_fit(f, x_data, y_data)
print('Coefficients:', popt)

roots = fsolve(f, [0,0,0], args=tuple(popt))
print('Roots:', roots)

# scatter plot the data
plt.scatter(x_data, y_data)

# plot the fitted function
plt.plot(x_data, f(x_data, *popt))

plt.show()
```

Exercise 3.3 (pandas):

First we import

```
import heapq
import pandas as pd
```

Read in data from .dat files. The column separator is ::. I manually set the column headers.

```
users = pd.read_csv("data/users.dat", header=None, sep='::', engine='python')
ratings = pd.read_csv("data/ratings.dat", header=None, sep='::', engine='python')
movies = pd.read_csv("data/movies.dat", header=None, sep='::', engine='python')

users.columns = ['userid', 'gender', 'age', 'occupationcode', 'zip']
ratings.columns = ['userid', 'movieid', 'rating', 'timestamp']
movies.columns = ['movieid', 'title', 'genre']
```

Merge into one DataFrame. First we merge on user id, and then on movie id.

```
merged = pd.merge(users, ratings, on='userid')
merged = pd.merge(merged, movies, on='movieid')
```

Top 5 movies with most ratings. I count the number of ratings for each movie by counting unique entries under `title`. It might be more correct to use `movie id` for this, since two movies could theoretically have the same name, but since this is a training exercise, it is more fun to work with titles. This is achieved with the `value_counts()` function.

```
merged['title'].value_counts()[:5]
```

```
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
dtype: int64
```

Now I use a lambda function to filter out movies with less than 250 ratings. Again I group by `title`.

```
active_titles = merged.groupby("title").filter(lambda x: len(x) >= 250)
```

I use a boolean mask to separate female and male ratings. And then it is easy to find the average ratings using `groupby()` and `mean()`:

```
female = active_titles[active_titles['gender'] == 'F']
male = active_titles[active_titles['gender'] == 'M']
female_means = female.groupby('title')['rating'].mean()
male_means = male.groupby('title')['rating'].mean()
```

Top 3 movies by female rating:

```
female_means.order(ascending=False)[:3]
```

```
title
Close Shave, A (1995)                               4.644444
```

```
Wrong Trousers, The (1993)                          4.588235
Sunset Blvd. (a.k.a. Sunset Boulevard) (1950)       4.572650
Name: rating, dtype: float64
```

Top 3 movies by male rating:

```
male_means.order(ascending=False)[:3]
```

```
title
Godfather, The (1972)                                                    4.583333
Seven Samurai (The Magnificent Seven) (Shichinin no samurai) (1954)
    4.576628
Shawshank Redemption, The (1994)                                         4.560625
Name: rating, dtype: float64
```

In order to compute the differences in average rating, I resort to some standard Python code. It is possible to loop through the entries in the pandas series by calling `iteritems()`.

```
diff_ratings = []
for title, female_rating in female_means.iteritems():
    if title in male_means:
        male_rating = male_means[title]
        diff = float(female_rating) - float(male_rating)
        diff_ratings.append((title, diff))
```

Top 10 movies favored by females:

```
heapq.nlargest(10, diff_ratings, key = lambda x: x[1])
```

```
[('Dirty Dancing (1987)', 0.8307820472768923),
 ("Jumpin' Jack Flash (1986)", 0.6763587721768514),
 ('Grease (1978)', 0.6082238191659717),
 ('Little Women (1994)', 0.5488491048593351),
 ('Steel Magnolias (1989)', 0.5357766572377325),
 ('Anastasia (1997)', 0.518390804597701),
 ('Rocky Horror Picture Show, The (1975)', 0.5128851540616246),
 ('Color Purple, The (1985)', 0.4988514310548209),
 ('Age of Innocence, The (1993)', 0.4875614963334258),
 ('Free Willy (1993)', 0.4825728044026598)]
```

Top 10 movies favored by males:

```
heapq.nlargest(10, diff_ratings, key = lambda x: -x[1])
```

```
[('Good, The Bad and The Ugly, The (1966)', -0.7263506433630917),
 ('Kentucky Fried Movie, The (1977)', -0.6763591800356505),
 ('Dumb & Dumber (1994)', -0.63860833475617),
 ('Longest Day, The (1962)', -0.6196818349981501),
 ('Cable Guy, The (1996)', -0.6137873754152823),
 ('Evil Dead II (Dead By Dawn) (1987)', -0.6119854031246437),
```

```
 ('Hidden, The (1987)', -0.6071670047329278),
 ('Rocky III (1982)', -0.5818006971991823),
 ('Caddyshack (1980)', -0.5736015764047799),
 ('For a Few Dollars More (1965)', -0.5447044704470447)]
```

Top 5 controversial movies (greatest standard deviation):

```
active_titles.groupby('title')['rating'].std().order(ascending=False)[:5]
```

```
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
Name: rating, dtype: float64
```

Exercise 3.4 (scikit-learn):

I solve this using the `sklearn.linear_model.LogisticRegression` module from SciKit-Learn. I had to modify the code from week 2 quite a lot, because in week 2 I understood the bag of words to just one huge dictionary with the number of occurences of each word, across all the request texts. So I altered the code and now it works roughly in the following way:

1. Load the data from an external file.
2. Iterate through the request texts and build a set of unique words, using RegEx to filter valid words.
3. Associate a unique ID or index to each word in the set while converting it into a dictionary.
4. Iterate through request texts again, this time building a vector of how many times a word occures in the request text. Each index in the vector corresponds to a word in the set of unique words by the index given in the previous step.
5. Fit the logistic regression model using 90 percent of the dataset.
6. Predict whether a person received pizza for the remaining 10 percent of the data set.
7. Compute success rate.

Output

```
Rows in bag:    4040
Columns in bag: 12379
Built bag_of_words in: 4.428589105606079 seconds

Fitted logistic regression model in: 15.016097068786621 seconds
Predicted on validation data set in: 13.19341492652893 seconds
Results of predictions:
Number of True predictions: 744
```

```
Number of False predictions: 2892
Comparing predictions of the model to reality in the validation data set:
Good predictions:    3394
Bad predictions:     242
Success rate:        93.34 %
```

A success rate of 93.3 percent in the prediction of the validation data set is quite good! It is certainly significantly better than a random 50 percent.

For the fun of it I took the validation set and looked at which request had the highest probability of failure, and which had the highest probablity of success. This was the result:

The following request was the WORST out of the validation data, with a predicted 99.9990972073 percent probability of failure:

> So... this is finals week, and I figured a responsible way of bracing myself for exams would be.....catching up on Futurama/Adventure Time/Fullmetal Alchemist episodes while EATING PIZZA !! Isn't it an awesome idea ? Also my battleplan involves a blanket and tea, and extreme denial of the imminence of exams ;)
>
> The only erm, slight problem I encountered in my GENIUS project. Is that I'm kinda broke, and fishfingers and cereals only last you so long before you are too broke to have pizza but too proud/ashamed to beg your family to send you enough to last the month :/ Anyone ever experienced this ?
>
> So if anyone feels like sending a bit of cheesy food to Dublin, Ireland, it would really brighten up the whole *week* :D of a broke but happy female Economics and Politics college student. (not sure if there's a rule to specify stuff about myself to prove I'm real but, yeah, here ou go haha)
>
> If infos are needed or something to prove I'm real PM me please, also I'm not sure how to give my adress or...(?) (sorry about the awkwardness, it's my first time creating a topic instead of just commenting. Stomach made me do it)
>
> PS: Sorry for any second-language slips or grammatical atrocities. English isn't my native language. (France yaaay)

The following request was the BEST out of the validation data, with a predicted 99.9999273381 percent probability of success:

> EDIT! Thanks to jetboyterp! He provided us with some nice grub for the night. It amazes me to no end to see the kindness still left in some people, and pushes me forward to keep providing that same kindness to others whenever I am able. I'll follow up with some pictures of some pizza!
>
> Hello fellow Redditors. I'm probably gonna rant and make a bigger deal of this than I should, and I'll try to keep it short and simple. But let me mention, I am NOT one to ask for assistance. But there comes times when we must swallow our pride.
>
> I have a small family of three, just my daughter, my wife, and myself. I'm a mechanic by trade, and make an acceptable amount of money...normally. The last month has been very rough, with work being slow and hours being cut. We are in an apartment that costs us $325/wk, and that is killing us. We've had to cut off both phones, and recently our car took a turn for the worst and left us without transportation.

I've been able to keep food in the house, up until the last week or so. Mostly thanks to foodstamps, but those funds have run out. This last week I've had to pay back some small debts to my boss and father, and it's really hurt.

Also, our stove at this apartment doesn't work, so we've had to basically resort to microwave, cold prep foods, and a single burner hotplate. My cupboards and fridge are empty, short of some month old ice cream and leftovers that we've been too lazy to throw away. My family is hungry, but just for the day.

So here is my request, which I will counter with an offer. In exchange for a pizza, I will buy 2 other people in need a pizza in the near future. This is my way of setting aside the guilt of asking, and expanding the assistance of RAOP. I can provide any and all proof and verification needed. And whatever happens, I'd still like to thank RAOP for all their awesomeness.

NOTE: Come Monday, I can get my paycheck advanced to me. Yesterday we just barely slid by, and all will be well from Moday on. I've got a second job that I start Tuesday at. It's just rather...depressing(maybe even desperate) feeling to be hungry, have an empty wallet, and feel like there is nothing I can do. Thanks again Reddit. :)

EDIT: I'm so sorry, I completely forgot to put my location, and I kept saying it in my head. :P Anyways, I'm in South Florida, right by Ft. Lauderdale.

Code

```
from sklearn import linear_model
import json
import re
import time

t0 = time.time()

#  Use regex to only allow words with letters and apostrophes
p = re.compile("[a-z][a-z']*")

with open('data/pizza-train.json', 'r') as f:
    data = json.load(f)

# First I find every unique word in the request texts. I use a set for this
unique_words = set()

for item in data:
    # Convert to lower case
    req_text = item['request_text'].lower()

    # Add words to the set of unique words
    for match in p.finditer(req_text):
        word = match.group()
        unique_words.add(word)

n_words = len(unique_words)

# Now I need to associate an index with every word in this set, and it should be
↪  quick to search in it too, so let's build a dict with an index as a key..
unique_words_indexed = {word: i for i, word in enumerate(unique_words)}

bag_of_words = []
received_pizza = []

# Iterate through the request texts again, building a bag of words for each entry
for item in data:
    # Initialize an empty vector of zeros to hold the multiplicity of words for
↪  this request text
    vector = [0]*n_words

    req_text = item['request_text'].lower()
    for match in p.finditer(req_text):
        word = match.group()
        index = unique_words_indexed[word]
        vector[index] += 1

    # append the results of this request text into the bag of words
    bag_of_words.append(vector)

    # Did he/she receive pizza?
    received_pizza.append(item['requester_received_pizza'])

print('Rows in bag:    ', len(bag_of_words))
print('Columns in bag:', n_words)

# PAGE BREAK
```

```
t1 = time.time()
print('Built bag_of_words in:', t1-t0, 'seconds\n')

t0 = time.time()
# Fit the logistic regression model using 90 % as training data
X = bag_of_words[:int(0.9*len(bag_of_words))]
Y = received_pizza[:int(0.9*len(bag_of_words))]
logreg = linear_model.LogisticRegression()
logreg.fit(X, Y)

t1 = time.time()
print('Fitted logistic regression model in:', t1-t0, 'seconds')

t0 = time.time()
# Validation data set:
Xval = bag_of_words[int(0.1*len(bag_of_words)):]
Yval = received_pizza[int(0.1*len(bag_of_words)):]
prediction = logreg.predict(Xval)
t1 = time.time()
print('Predicted on validation data set in:', t1-t0, 'seconds')

print('Results of predictions:')
print('Number of True predictions:', list(prediction).count(True))
print('Number of False predictions:', list(prediction).count(False))

# Let us see how closely the prediction fits the validation data set:
successes = 0
failures = 0
for val, pred in zip(Yval, prediction):
    if val == pred:
        successes += 1
    else:
        failures += 1
print('Comparing predictions of the model to reality in the validation data set:')
print('Good predictions:   ', successes)
print('Bad predictions:    ', failures)
print('Success rate:       ', round(successes / (successes + failures) * 100, 2),
↪  '%')

# For the fun of it, let's look again at the validation data set and find the parts
↪  of the
data_val = data[int(0.1*len(bag_of_words)):]
proba_val = logreg.predict_proba(Xval)
max_false_prob = max(proba_val[:, 0]) # the first element is the prob of false
max_true_prob = max(proba_val[:, 1])
for item, prob_false, prob_true in zip(data_val, proba_val[:, 0], proba_val[:, 1]):
    if prob_false == max_false_prob:
        worst_request = item['request_text']
    if prob_true == max_true_prob:
        best_request = item['request_text']

print('The following request was the WORST out of the validation data, with a
↪  predicted', max_false_prob*100, 'percent probability of failure:\n')
print(worst_request)
print('\nThe following request was the BEST out of the validation data, with a
↪  predicted', max_true_prob*100, 'percent probability of success:\n')
print(best_request)
```

Exercise 3.5 (cython):

At first I tried to use some nice Python syntax, using generators. However Cython was not good at optimizing this, so I resorted to using loops. The python code for the interpreted version looks like:

```
def s():
    tot = 0
    for i in range(1, 10001):
        tot += 1/(i*i)
```

And the code I compile with Cython looks like

```
def s():
    cdef double tot
    cdef int i
    tot = 0
    for i in range(1, 10001):
        tot += 1/(i*i)
```

Here I have declared the running sum `tot` as a C type double, and the iteration variable `i` as a C type int. Then I wrote some wrapper code to call these functions and time it. 500 times was not really enough to get reliable results on my desktop PC, so I called it 5000 times and averaged it. The result is a speedup of 231 times:

```
Running interpreted code in... 6.473860025405884 seconds
Running compiled code in... 0.028018951416015625 seconds
Speedup factor: 231.05290163376446
```

The command I use to build with Cython is:

```
python setup.py build_ext --inplace
```

DBSCAN

I managed to solve the 100000×100000 problem in about 12 seconds running on my PC at home. See the results in Table 1.

Size	Classes incl noise	Points in largest class	Running time in seconds
10	4	4	0.14
100	6	24	0.16
1000	9	289	0.44
10000	394	2847	3.15
100000	1692	28470	12.81

Table 1: Results for all the test data sets. Points in largest class is not counting the noise class.

First let me introduce how I parse the data and some other simple observations and choices in the implementation. This is the work needed to just get it running, but with horrible performance, because we will be calling `regionQuery()` n times and `jaccardDistance()` n^2 times.

Then I will introduce two ways in which I very significantly reduced the number of calls made to `jaccardDistance()`, but also the number of calls made to `regionQuery()`.

Basic implementation

Here are some notes on what I did to just get it running.

- The data is boolean, it is either 1 or non-existant in the sparse matrix structure. That means we do not need to store the 1. All we need is a set of indices for where there is data. This will speed up the program somewhat, and more importantly it makes the data simpler and easier to work with.
- I do not use the Scipy sparse matrix for the actual work. I parse all the data into native Python structures.
- I use a class for the points. So while parsing the points, I can initiate it with its indices and some other fields like `self.visited = False` and `self.cluster = None`. I am not sure that using a class is the most efficient, but it is extremely convenient and makes the whole thing easier to code.
- For every point, I store the indices indicating where it has data as a Python `Set()`, because then I can compute the intersection needed for the jaccardDistance by the built-in `.intersection()` method of Python sets. This is rather efficient.

Reducing complexity

Now let us talk about how to reduce the number of calls made to `jaccardDistance()` and `regionQuery()`. Without this, I was not in any way able to come near the 100000×100000 problem. The ideas presented here are for pre-processing the data in certain ways and then implementing DBSCAN in a way that can take advantage of it.

The first idea is to reduce the number of calls made to `jaccardDistance()` by producing a dictionary that quickly tell us which points are worth considering as potential neighbors. The observation is that if two points have no dimensions in common at all, then certainly they cannot be neighbors (by the definition of Jaccard Distance). Therefore:

Make a dictionary (that I call a `similarityDict` that uses dimensions as keys (every key in the dictionary corresponds to a dimension). Then as values, store a set of all the points that has this dimension. This dictionary can be built very quickly – it takes next to no time to run.

Now use the dictionary in the following way: If we call `regionQuery()` to find neighbors of the point P, then instead of iterating through every single other point, we instead iterate through only the

dimensions that P itself has. For every dimension that P has, we use the `similarityDict` to look up the other points that shares this dimension, and add them to a set of potential neigbors. At the end, we only have to call `jaccardDistance()` on exactly those points that share at least one dimension with P. Using this trick, I was able to solve the 100000×100000 problem in about 2 hours.

The second idea is to reduce the number of calls made to `regionQuery()` itself, and indirectly also the number of calls to `jaccardDistance()`. I do this by realizing that these data sets have a huge amount of redundancy. If we analyze the 100000×100000 dataset, then there are only 2978 unique points out of 100000! In other words, there are only 2978 unique rows in the sparse SciPy matrix. This means that on average, there are 33 duplicates of every point – they literally lie on top of each other. Now there are certainly no reason for us to call `regionQuery(P1)` and later `regionQuery(P2)` if we know that `P1` and `P2` actually lie on top of each other! There are perhaps multiple ways to optimize DBSCAN for duplicate points. I did it the following way: I simply removed duplicate points and instead stored a multiplicity with every point, `self.multiplicity`. Then in DBSCAN when we have found a neighbor that has a Jaccard Distance less than epsilon, we weigh it with its multiplicity when comparing the number of neighbors to `MinPts`, hereby accounting for all the duplicate points we have removed. When we store the points using their multiplicity, in order not to lose the information on which row in the original SciPy matrix the points correspond to, we add a list in the `Point` class: `self.org_row_numbers`. Then when we parse a row from the SciPy matrix, we simply add the row number to this list to keep track of which rows go into which instance of the `Point` class.

This allowed me to reduce the running time from about 2 hours to 45 seconds. Then I did some further optimizing by looking at the output from `cProfile` and got it down to 12 seconds.

Note that i have NOT altered the data set. I have NOT lost any information by doing things this way! I have simply stored the data in a data structure that takes advantage of the characteristics of the data in question. I would be able to convert the data back into the same format that I received it in, without loss, if I wanted to. The output from DBSCAN is not affected either. I am able to tell the number of clusters, easily count the number of points in every cluster and output the exact points that lie in a cluster.

Finally, remember to take note of the number of points found in the largest cluster, see Table 1. As noted on Wikipedia, DBSCAN is not actually deterministic – some points might end up in one cluster on one run, but another cluster on a second run. So the numbers I found might vary slightly with the "correct" numbers posted for evaluation.

Output

Here is actual sample output from the 100000×100000 problem:

```
Size 100000 x 100000

Load data using Pickle in:        0.14 seconds
Prepare data structure in:        8.61 seconds
Running DBSCAN in:                4.07 seconds
Total running time:               12.81 seconds

Classes incl noise class:         1692
Unique points:                    2978 out of 100000
Elements in the largest cluster: 28470
```

Code for DBSCAN (158 lines)

```
from collections import deque
import pickle
import time
import sys

class Point():
    def __init__(self, features, row_number):
        self.cluster = None
        self.visited = False
        self.features = features
        self.multiplicity = 1
        # Add a list that keeps track of which row these points had in the original
↪   SciPy array (zero-indexed):
        self.org_row_numbers = [row_number]

def main():
    if len(sys.argv) > 1 and sys.argv[1] == '10':
        # Testdata 10x10
        testrun(size=10, eps=0.4, MinPts=2)
    elif sys.argv[1] == '100':
        # Testdata 100x100
        testrun(size=100, eps=0.3, MinPts=2)
    elif sys.argv[1] == '1000':
        # Testdata 1000x1000
        testrun(size=1000, eps=0.15, MinPts=2)
    elif sys.argv[1] == '10000':
        # Testdata 10000x10000
        testrun(size=10000, eps=0.15, MinPts=2)
    elif sys.argv[1] == '100000':
        # Testdata 100000x100000
        testrun(size=100000, eps=0.15, MinPts=2)
    else:
        print('Error in input argument')
        sys.exit()

def sparse_to_python(sparse_mat):
    # Convert the Scipy sparse matrix to a pure native Python structure, using my
↪   Point class.
    # And remove duplicate points, instead storing a multiplicity with the
↪   remaining point.
    multiplicityDict = {}
    D = []
    for row_number, row in enumerate(sparse_mat):
        indices = tuple(row.indices)
        if indices in multiplicityDict:
            point = multiplicityDict[indices]
            point.multiplicity += 1
            point.org_row_numbers.append(row_number)
        else:
            point = Point(set(indices), row_number)
            D.append(point)
            multiplicityDict[indices] = point
    return D

## PAGE BREAK ##
```

```
def similarities(D):
    # Make a dict where each key is a feature, and the value being stored is a list
↪   of the points that have this feature.
    similarityDict = {}
    for point in D:
        for feature in point.features:
            if feature in similarityDict:
                similarityDict[feature].append(point)
            else:
                similarityDict[feature] = [point]
    return similarityDict

def neighborUnpack(neighbors):
    # Returns the actual number of neigbors in 'neigbors' by looking at
↪   multiplicity
    return sum(n.multiplicity for n in neighbors)

def DBSCAN(D, eps, MinPts, similarityDict):
    C = 1
    for P in D:
        if P.visited:
            continue
        P.visited = True
        NeighborPts = regionQuery(D, P, eps, similarityDict)
        if neighborUnpack(NeighborPts) >= MinPts:
            # Make a new cluster
            expandCluster(D, P, NeighborPts, C, eps, MinPts, similarityDict)
            C += 1
    # return number of clusters found including None class (noise class)
    return C

def expandCluster(D, P, NeighborPts, C, eps, MinPts, similarityDict):
    queue = deque(NeighborPts)
    P.cluster = C
    while queue:
        P2 = queue.popleft()
        if not P2.visited:
            P2.visited = True
            NeighborPts2 = regionQuery(D, P2, eps, similarityDict)
            if neighborUnpack(NeighborPts2) >= MinPts:
                queue.extend(NeighborPts2)
        if P2.cluster == None:
            P2.cluster = C

def jaccardDistance(p1, p2):
    intersection = len(p1.features.intersection(p2.features))
    jDis = 1 - intersection / ( len(p1.features) + len(p2.features) - intersection)
    return jDis

## PAGE BREAK ##
```

```
def regionQuery(D, P, eps, similarityDict):
    # return all points within P's eps-neighborhood (including P)
    neighbors = []
    potential_neighbors = set()
    for feature in P.features:
        for point in similarityDict[feature]:
            potential_neighbors.add(point)
    for point in potential_neighbors:
        jDis = jaccardDistance(P, point)
        if jDis <= eps:
            neighbors.append(point)
    return neighbors

def getTestdata(s_point_count):
    s_point_count = str(s_point_count)
    data = pickle.load(open('test_files/data_%spoints_%sdims.dat' % (s_point_count,
↪   s_point_count), 'rb'), encoding='latin1')
    return data

def largestCluster(D):
    # Return the number of elements in the largest cluster
    clusters = {}
    for point in D:
        if point.cluster in clusters:
            clusters[point.cluster] += point.multiplicity
        else:
            clusters[point.cluster] = point.multiplicity
    # Remove the None class (noise class)
    clusters.pop(None, None)
    largest_cluster = max(clusters, key = lambda x: clusters[x])
    nElements = clusters[largest_cluster]
    return nElements

## PAGE BREAK ##
```

```
def testrun(size=None, eps=None, MinPts=None):
    print('Size', size, 'x', size, '\n')

    t0 = time.time()
    D = getTestdata(size)
    t1 = time.time()
    tot = t1 - t0
    print('Load data using Pickle in:      ', round(t1-t0, 2), 'seconds')

    t0 = time.time()
    D = sparse_to_python(D)
    similarityDict = similarities(D)
    t1 = time.time()
    tot += t1 - t0
    print('Prepare data structure in:      ', round(t1-t0, 2), 'seconds')

    t0 = time.time()
    C = DBSCAN(D, eps, MinPts, similarityDict)
    t1 = time.time()
    tot += t1 - t0
    print('Running DBSCAN in:              ', round(t1-t0, 2), 'seconds')
    print('Total running time:             ', round(tot, 2), 'seconds')

    print('\nClasses incl noise class:       ', C,)
    print('Unique points:                  ', len(D), 'out of', size)
    nElements = largestCluster(D)
    print('Elements in the largest cluster:', nElements)

if __name__ == '__main__':
    main()
```