Sample solutions (Fall 2016)

Bookmark this page

problem1 (Score: 8.0 / 8.0)

1. Test cell (Score: 1.0 / 1.0)
2. Test cell (Score: 3.0 / 3.0)
3. Test cell (Score: 2.0 / 2.0)
4. Test cell (Score: 2.0 / 2.0)

Important note!

Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

In [1]:
```
YOUR_ID = "rvuduc3" # Please enter your GT login, e.g., "rvuduc3" or "gtg911x"
COLLABORATORS = ["skaramati3"] # list of strings of your collaborators' IDs
```

In [2]: Grade cell: `who__test` Score: 1.0 / 1.0 (Top)
```
import re

RE_CHECK_ID = re.compile (r'''[a-zA-Z]+\d+|[gG][tT][gG]\d+[a-zA-Z]''')
assert RE_CHECK_ID.match (YOUR_ID) is not None

collab_check = [RE_CHECK_ID.match (i) is not None for i in COLLABORATORS]
assert all (collab_check)

del collab_check
del RE_CHECK_ID
del re
```

Jupyter / IPython version check. The following code cell verifies that you are using the correct version of Jupyter/IPython.

In [3]:
```
import IPython
assert IPython.version_info[0] >= 3, "Your version of IPython is too old, please update i
t."
```

Problem 1: Warm-up! [7 points]

Winter is coming (or rather, is here). So let's do some warm-up exercises to get you in the right mindset for the rest of the exam.

The main point of this problem is to make sure you can quickly read some Python documentation and apply it.

This problem has three (3) parts or "exercises" and is worth a total of seven (7) points.

Setup

This problem depends on the following modules. Make sure these are available on your system before beginning.

In [4]:
```
import pandas as pd

from IPython.display import display

import seaborn as sns
%matplotlib inline

from scipy.cluster.vq import kmeans2

%reload_ext autoreload
%autoreload 2

from cse6040utilsfa16 import make_scatter_plot

import urllib
import gzip
```

Download and unpack a compressed file

Exercise 1 (3 points). Write a function that

- downloads a compressed gzip (`xxx.gz`) file from the interwebs, given its URL; and
- returns a file-like handle to it.

In particular, find and read the documentation for `urllib.request.urlopen()` (https://docs.python.org/3/library/urllib.request.html) and `gzip.open()` (https://docs.python.org/3/library/gzip.html), and use them to implement the desired function.

> Note: `gzip.open()` accepts an optional argument named `mode`, which specifies how to interpret the contents of the data when decompressing it. The `url_open_gz()` function you implement should simply pass its `mode` argument onto `gzip.open()`.

The test code will check your implementation and uses it to download a dataset, which you will need for the rest of this problem.

In [5]: Student's answer (Top)

```
def open_url_gz (url_gz, mode='rt'):
    """
    Given a URL to a compressed gzip (.gz) file, downloads it to a
    temporary location, unpacks it, and returns a file-like handle
    for reading its uncompressed contents.
    """
    print ("Downloading", url_gz, "...")

    return gzip.open (urllib.request.urlopen (url_gz), mode=mode)
```

In [6]: Grade cell: `open_url_gz_test` Score: 3.0 / 3.0 (Top)

```
# Test code for Part 1-A

# First, check a small message.
url_msg_gz = 'http://cse6040.gatech.edu/datasets/message_in_a_bottle.txt.gz'
with open_url_gz (url_msg_gz) as fp_msg:
    line = fp_msg.readline ()
    msg = line.strip ()
    print ("\nDownloaded the message: '%s'" % msg)

assert msg == 'Good luck, kiddos!'
print ("\n(Passed check 1 of 2! Sarah P., this 'pass' message is for you!)\n")

# Next, use this function to download a dataset.
```

```
url_data_gz = 'http://cse6040.gatech.edu/datasets/faithful.dat.gz'
with open_url_gz (url_data_gz) as fp:
    fp_local = open ('faithful.dat', 'wt')
    fp_local.write (fp.read ())
    fp_local.close ()
with open ('faithful.dat', 'rt') as fp_faithful:
    assert fp_faithful.readline () == 'Old Faithful Geyser Data\n'

print ("\n(Passed check 2 of 2!)")
```

```
Downloading http://cse6040.gatech.edu/datasets/message_in_a_bottle.txt.gz ...

Downloaded the message: 'Good luck, kiddos!'

(Passed check 1 of 2! Sarah P., this 'pass' message is for you!)

Downloading http://cse6040.gatech.edu/datasets/faithful.dat.gz ...

(Passed check 2 of 2!)
```

Load the "Old Faithful" dataset

The test code above, assuming you implemented `open_url_gz()` correctly, should have downloaded and unpacked a file in the local directory called `faithful.dat`.

If you did not manage to get a working implementation that does that, then manually download a copy of this file now from here: http://www.stat.cmu.edu/~larry/all-of-statistics/=data/faithful.dat (http://www.stat.cmu.edu/~larry/all-of-statistics/=data/faithful.dat)

(In either case, you should probably click on the above URL to see what is in the file, as you'll be working with this data.)

This particular dataset comes from a study of the Old Faithful geyser (https://en.wikipedia.org/wiki/Old_Faithful) in Yellowstone National Park. Amazingly, this geyser erupts very regularly, hence the name! The dataset contains a bunch of observations, where each observation consists of a) the duration of an eruption (in minutes), and b) the time until the next eruption (again in minutes).

Exercise 2 (2 points). Use the Pandas function, `pd.read_table()` (http://pandas.pydata.org/pandas-docs/version/0.18.1/generated/pandas.read_table.html), to read this dataset from `faithful.dat` and store it in a data frame with two columns, one named `eruptions` and one named `waiting`.

> Hint: There is a one-line solution, which requires only that you set the right arguments to `pd.read_table()`.

In [7]: Student's answer (Top)

```
faithful = pd.read_table ('faithful.dat',
                          sep='\s+',
                          skiprows=26,
                          names=['eruptions', 'waiting'])

# Quickly inspect this data
display (faithful.head ())
print ("...")
display (faithful.tail ())
```

	eruptions	waiting
1	3.600	79
2	1.800	54
3	3.333	74
4	2.283	62
5	4.533	85

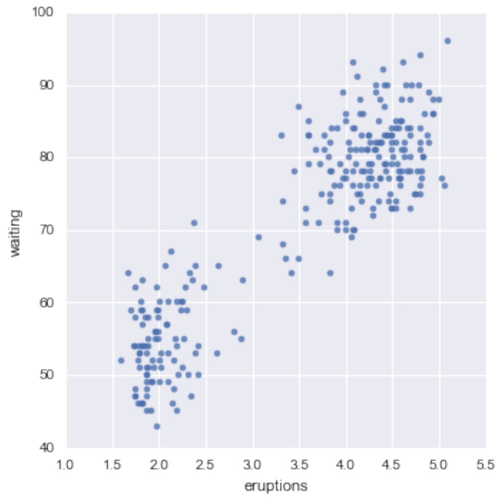
...

	eruptions	waiting
268	4.117	81
269	2.150	46
270	4.417	90
271	1.817	46
272	4.467	74

In [8]: Grade cell: `load_old_faithful_test` Score: 2.0 / 2.0 (Top)

```
# Test code

make_scatter_plot (faithful, x='eruptions', y='waiting', hue=None)

# Sanity check a few values
assert set (faithful.columns) == set (['eruptions', 'waiting'])
assert sum (faithful['waiting'] == 54) == 9
assert sum (faithful['waiting'] == 90) == 6
print ("\n(Passed!)")
```

```
(Passed!)
```



Cluster this data

Exercise 3 (2 points). The plot of the data suggests that there is a relationship between how long an eruption lasts and the time between eruptions. Use Scipy's `kmeans2()` (https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.cluster.vq.kmeans2.html) to estimate this cluster structure, specifically by doing the following.

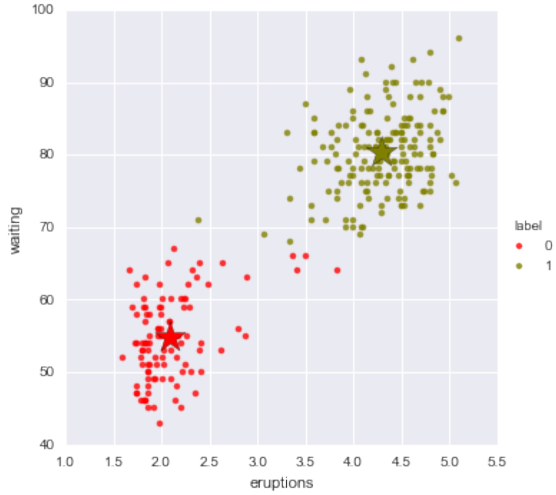
- Assign each point of `faithful` an integer cluster label that is either 0 or 1. Store these labels in a new column of the data frame named `label`.
- Compute the centers of the two clusters, storing them in a 2×2 Numpy array named `centers`, where each row `centers[i, :]` is a center whose columns contain its coordinates.

In [9]: Student's answer (Top)

```
centers, labels = kmeans2 (faithful[['eruptions', 'waiting']], k=2)
faithful['label'] = labels

num_clusters = 2
class_sizes = [sum (labels==c) for c in range (num_clusters)]
print ("\n=== Clusters ===")
for c in range (num_clusters):
    print ("- Cluster {}: {} points centered at {}".format (c,
                                                           class_sizes[c],
```

```
                                                        centers[c, :2]))

make_scatter_plot (faithful, x='eruptions', y='waiting', centers=centers)
```

```
=== Clusters ===
- Cluster 0: 100 points centered at [   2.09433   54.75    ]
- Cluster 1: 172 points centered at [   4.29793023   80.28488372]
```



In [10]: Grade cell: `kmeans2_test` Score: 2.0 / 2.0 (Top)

```
assert 90 <= min (class_sizes) <= max (class_sizes) <= 180
print ("\n(Passed!)")
```

```
(Passed!)
```

In [11]:

Problem 2 was adapted into one of your homework assignments (Notebook 15). As such, we have omitted it from these sample solutions to the practice final exam.

problem3 (Score: 13.0 / 13.0)

1. Test cell (Score: 1.0 / 1.0)
2. Test cell (Score: 1.0 / 1.0)
3. Test cell (Score: 4.0 / 4.0)
4. Test cell (Score: 4.0 / 4.0)
5. Test cell (Score: 3.0 / 3.0)

Important note!

Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

In [1]:
```
YOUR_ID = "rvuduc3" # Please enter your GT login, e.g., "rvuduc3" or "gtg911x"
COLLABORATORS = ["skarmati3"] # list of strings of your collaborators' IDs
```

In [2]: Grade cell: `who__test` Score: 1.0 / 1.0 (Top)

```
import re

RE_CHECK_ID = re.compile (r'''[a-zA-Z]+\d+|[gG][tT][gG]\d+[a-zA-Z]''')
assert RE_CHECK_ID.match (YOUR_ID) is not None

collab_check = [RE_CHECK_ID.match (i) is not None for i in COLLABORATORS]
assert all (collab_check)

del collab_check
del RE_CHECK_ID
del re
```

Jupyter / IPython version check. The following code cell verifies that you are using the correct version of Jupyter/IPython.

In [3]:
```
import IPython
assert IPython.version_info[0] >= 3, "Your version of IPython is too old, please update i
t."
```

Problem 3: HRC's Email [12 points]

In this problem, you'll show your SQL and Pandas chops on the dataset consisting of Hilary Rodham Clinton's emails!

This problem has four (4) parts or "exercises."

Setup

Start by downloading an SQLite database containing about 7,900 of HRC's messages:

https://t-square.gatech.edu/access/content/group/gtc-3bd6-e221-5b9f-b047-31c7564358b7/hrc.db (https://t-square.gatech.edu/access/content/group/gtc-3bd6-e221-5b9f-b047-31c7564358b7/hrc.db)

> **Do not share this file outside of this class!** We downloaded this database from Kaggle and have posted it on T-Square for your convenience. If anyone outside this class is interested in getting a copy of this database, please point them directly to the Kaggle site: https://www.kaggle.com/kaggle/hillary-clinton-emails (https://www.kaggle.com/kaggle/hillary-clinton-emails)

Next, let's run some setup code, which will load the modules you'll need for this problem

In [4]:
```
import sqlite3 as db

from IPython.display import display
import pandas as pd

%reload_ext autoreload
%autoreload 2
from cse6040utilsfa16 import peek_table, list_tables
from cse6040utilsfa16 import tibbles_are_equivalent

import numpy as np
```

In [5]:
```
conn = db.connect ('hrc.db')

print ("List of tables in the database:", list_tables (conn))
```

```
List of tables in the database: ['Emails', 'Persons', 'Aliases', 'EmailReceivers']
```

In [6]:
```
peek_table (conn, 'Emails')
peek_table (conn, 'EmailReceivers', num=3)
peek_table (conn, 'Persons')
```

```
Total number of records: 7945

First 5 entries:
```

	Id	DocNumber	MetadataSubject	MetadataTo	MetadataFrom	SenderPersonId	MetadataDateSent	Metadat
0	1	C05739545	WOW	H	Sullivan, Jacob J	87	2012-09-12T04:00:00+00:00	2015-05-22T04:0C
1	2	C05739546	H: LATEST: HOW SYRIA IS AIDING QADDAFI AND MOR...	H			2011-03-03T05:00:00+00:00	2015-05-22T04:0C
2	3	C05739547	CHRIS STEVENS	;H	Mills, Cheryl D	32	2012-09-12T04:00:00+00:00	2015-05-22T04:0C
3	4	C05739550	CAIRO CONDEMNATION - FINAL	H	Mills, Cheryl D	32	2012-09-12T04:00:00+00:00	2015-05-22T04:0C
4	5	C05739554	H: LATEST: HOW SYRIA IS AIDING QADDAFI AND MOR...	Abedin, Huma	H	80	2011-03-11T05:00:00+00:00	2015-05-22T04:0C

5 rows × 22 columns

```
Total number of records: 9306

First 3 entries:
```

	Id	EmailId	PersonId
0	1	1	80
1	2	2	80
2	3	3	228

```
Total number of records: 513

First 5 entries:
```

	Id	Name
0	1	111th Congress
1	2	AGNA USEMB Kabul Afghanistan
2	3	AP
3	4	ASUNCION
4	5	Alec

Exercise 1 (1 point). Extract the `Persons` table from the database and store it as a Pandas data frame with two columns: `Id` and `Name`.

In [7]: Student's answer (Top)

```
Persons = pd.read_sql_query ('SELECT * FROM Persons', conn)
```

In [8]: Grade cell: `Persons_test` Score: 1.0 / 1.0 (Top)

```
assert 'Persons' in globals ()
assert type (Persons) is type (pd.DataFrame ())
assert len (Persons) == 513

print ("Five random people from the `Persons` table:")
display (Persons.iloc[np.random.choice (len (Persons), 5)])

print ("\n(Passed!)")
```

Five random people from the `Persons` table:

	Id	Name
235	236	russoiv@state.gov
389	390	steinberg james
470	471	laurenjiloty jilotylc@state.gov
396	397	abedinh@state.govr
200	201	Thomas Nides

(Passed!)

Exercise 2 (4 points). Query the database to determine how frequently particular pairs of people communicate. Store the results in a Pandas data frame named `CommEdges` having the following three columns:

- `Sender`: The ID of the sender (taken from the `Emails` table).
- `Receiver`: The ID of the receiver (taken from the `EmailReceivers` table).
- `Frequency`: The number of times this particular (`Sender`, `Receiver`) pair occurs.

Order the results in *descending* order of `Frequency`.

There is one corner case that you should also handle: sometimes the `Sender` field is empty (unknown). You can filter these cases by checking that the sender ID is not the empty string.

In [9]: Student's answer (Top)

```
query = """
  SELECT
      SenderPersonId AS Sender,
      PersonId AS Receiver,
      COUNT (*) AS Frequency
  FROM
      Emails, EmailReceivers
  WHERE
      Emails.Id = EmailReceivers.EmailId AND Sender <> ''
  GROUP BY
      Sender, Receiver
  ORDER BY
      -Frequency
"""
CommEdges = pd.read_sql_query (query, conn)
```

In [10]: Grade cell: `CommEdges_test` Score: 4.0 / 4.0 (Top)

```
# Read what we believe is the exact result (up to permutations)
CommEdges_soln = pd.read_csv ('CommEdges_soln.csv')

# Check that we got a data frame of the expected shape:
assert 'CommEdges' in globals ()
assert type (CommEdges) is type (pd.DataFrame ())
assert len (CommEdges) == len (CommEdges_soln)
assert set (CommEdges.columns) == set (['Sender', 'Receiver', 'Frequency'])

# Check that the results are sorted:
non_increasing = (CommEdges['Frequency'].iloc[:-1] >= CommEdges['Frequency'].iloc[1:])
assert non_increasing.all ()
```

```
print ("Top 5 communicating pairs:")
display (CommEdges.head ())

from cse6040utilsfa16 import canonicalize_tibble
assert tibbles_are_equivalent (CommEdges, CommEdges_soln)
print ("\n(Passed!)")
```

Top 5 communicating pairs:

	Sender	Receiver	Frequency
0	81	80	1406
1	32	80	1262
2	87	80	857
3	80	81	529
4	80	32	372

(Passed!)

Exercise 3 (4 points). Consider any pair of people, a and b. Suppose we don't care whether person a sends and person b receives or whether person b sends and person a receives. Rather, we only care that $\{a, b\}$ have exchanged messages.

That is, the previous exercise computed a *directed* graph, $G = \left(g_{a,b}\right)$, where $g_{a,b}$ is the number of times (or "frequency") that person a was the sender and person b was the receiver. Instead, suppose we wish to compute its *symmetrized* or *undirected* version, $H = G + G^T$.

Write some code that computes H and stores it in a Pandas data frame named `CommPairs` with the columns, `A`, `B`, and `Frequency`. Per the definition of H, the `Frequency` column should combine frequencies from G and G^T accordingly.

In [11]: Student's answer (Top)

```
G = CommEdges.rename (columns={'Sender': 'A', 'Receiver': 'B'})
GT = CommEdges.rename (columns={'Sender': 'B', 'Receiver': 'A'})
H = pd.merge (G, GT, on=['A', 'B'], suffixes=('_G', '_GT'))
H['Frequency'] = H['Frequency_G'] + H['Frequency_GT']
del H['Frequency_G']
del H['Frequency_GT']
CommPairs = H
```

In [12]: Grade cell: `CommPairs_test` Score: 4.0 / 4.0 (Top)

```
CommPairs_soln = pd.read_csv ('CommPairs_soln.csv')

assert 'CommPairs' in globals ()
assert type (CommPairs) is type (pd.DataFrame ())
assert len (CommPairs) == len (CommPairs_soln)

print ("Most frequently communicating pairs:")
display (CommPairs.sort_values (by='Frequency', ascending=False).head (10))

assert tibbles_are_equivalent (CommPairs, CommPairs_soln)
print ("\n(Passed!)")
```

Most frequently communicating pairs:

	A	B	Frequency
0	81	80	1935
3	80	81	1935
4	80	32	1634
1	32	80	1634
2	87	80	1206

6	80	87	1206
7	116	80	580
8	80	116	580
5	194	80	413
19	80	194	413

```
(Passed!)
```

Exercise 4 (3 points). Starting with a copy of `CommPairs`, named `CommPairsNamed`, add two additional columns that contain the names of the communicators. Place these values in columns named `A_name` and `B_name` in `CommPairsNamed`.

In [13]: Student's answer (Top)

```
CommPairsNamed = CommPairs.copy ()

CommPairsNamed = pd.merge (CommPairsNamed, Persons, left_on=['A'], right_on=['Id'])
CommPairsNamed.rename (columns={'Name': 'A_name'}, inplace=True)
del CommPairsNamed['Id']

CommPairsNamed = pd.merge (CommPairsNamed, Persons, left_on=['B'], right_on=['Id'])
CommPairsNamed.rename (columns={'Name': 'B_name'}, inplace=True)
del CommPairsNamed['Id']
```

In [14]: Grade cell: `add_names_test` Score: 3.0 / 3.0 (Top)

```
CommPairsNamed_soln = pd.read_csv ('CommPairsNamed_soln.csv')

assert 'CommPairsNamed' in globals ()
assert type (CommPairsNamed) is type (pd.DataFrame ())
assert set (CommPairsNamed.columns) == set (['A', 'A_name', 'B', 'B_name', 'Frequency']
)

print ("Top few entries:")
CommPairsNamed.sort_values (by=['Frequency', 'A', 'B'], ascending=False, inplace=True)
display (CommPairsNamed.head (10))

assert tibbles_are_equivalent (CommPairsNamed, CommPairsNamed_soln)
print ("\n(Passed!)")
```

```
Top few entries:
```

	A	B	Frequency	A_name	B_name
0	81	80	1935	Huma Abedin	Hillary Clinton
137	80	81	1935	Hillary Clinton	Huma Abedin
75	80	32	1634	Hillary Clinton	Cheryl Mills
1	32	80	1634	Cheryl Mills	Hillary Clinton
2	87	80	1206	Jake Sullivan	Hillary Clinton
61	80	87	1206	Hillary Clinton	Jake Sullivan
4	116	80	580	Lauren Jiloty	Hillary Clinton
95	80	116	580	Hillary Clinton	Lauren Jiloty
3	194	80	413	Sidney Blumenthal	Hillary Clinton
168	80	194	413	Hillary Clinton	Sidney Blumenthal

```
(Passed!)
```

When you are all done, it's good practice to close the database. The following will do that for you.

In [15]:
```
conn.close ()
```

In [16]:

problem4 (Score: 15.0 / 15.0)

1. Test cell (Score: 1.0 / 1.0)
2. Test cell (Score: 3.0 / 3.0)
3. Test cell (Score: 5.0 / 5.0)
4. Test cell (Score: 2.0 / 2.0)
5. Test cell (Score: 2.0 / 2.0)
6. Test cell (Score: 2.0 / 2.0)

Important note!

Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

In [1]:
```
YOUR_ID = "rvuduc3" # Please enter your GT login, e.g., "rvuduc3" or "gtg911x"
COLLABORATORS = ["skaramati3"] # list of strings of your collaborators' IDs
```

In [2]: Grade cell: `who__test` Score: 1.0 / 1.0 (Top)

```
import re

RE_CHECK_ID = re.compile (r'''[a-zA-Z]+\d+|[gG][tT][gG]\d+[a-zA-Z]''')
assert RE_CHECK_ID.match (YOUR_ID) is not None

collab_check = [RE_CHECK_ID.match (i) is not None for i in COLLABORATORS]
assert all (collab_check)

del collab_check
del RE_CHECK_ID
del re
```

Jupyter / IPython version check. The following code cell verifies that you are using the correct version of Jupyter/IPython.

In [3]:
```
import IPython
assert IPython.version_info[0] >= 3, "Your version of IPython is too old, please update i
t."
```

Problem 4: Tracking population movement [14 points]

This problem checks that you can perform some basic data cleaning and analysis. You'll work with what we think is a pretty interesting dataset, which can tell us something about how people move within the United States.

This problem has five (5) parts or "exercises" and is worth a total of fourteen (14) points.

Setup: IRS Tax Migration Data

The data for this problem comes from the IRS, which can tell where many households move from or to in any given year based on their tax returns.

For your convenience, we've placed the data files you'll need at the links below. Download them now. They are split by year among four consecutive years (2011-2015).

- 2011-2012 data: http://cse6040.gatech.edu/datasets/stateoutflow1112.csv (http://cse6040.gatech.edu/datasets/stateoutflow1112.csv)
- 2012-2013 data: http://cse6040.gatech.edu/datasets/stateoutflow1213.csv (http://cse6040.gatech.edu/datasets/stateoutflow1213.csv)
- 2013-2014 data: http://cse6040.gatech.edu/datasets/stateoutflow1314.csv (http://cse6040.gatech.edu/datasets/stateoutflow1314.csv)
- 2014-2015 data: http://cse6040.gatech.edu/datasets/stateoutflow1415.csv (http://cse6040.gatech.edu/datasets/stateoutflow1415.csv)

These data files reference states by their FIPS codes. So, we'll need some additional data to translate state FIPS numbers to "friendly" names.

- FIPS data: http://cse6040.gatech.edu/datasets/fips-state-2010-census.txt (http://cse6040.gatech.edu/datasets/fips-state-2010-census.txt)

> These are state-level data though county-level data also exist elsewhere. If you ever need that, you'll find it at the IRS website: https://www.irs.gov/uac/soi-tax-stats-migration-data (https://www.irs.gov/uac/soi-tax-stats-migration-data). And if you ever need the original FIPS codes data, see the Census Bureau website: https://www.census.gov/geo/reference/codes/cou.html (https://www.census.gov/geo/reference/codes/cou.html).

Beyond the data, you'll also need the following Python modules.

In [4]:
```
from IPython.display import display
import pandas as pd

%reload_ext autoreload
%autoreload 2

from cse6040utilsfa16 import tibbles_are_equivalent
```

Here is a sneak peek of what one of the data files looks like. Note the encoding specification, which may be needed to get Pandas to parse it.

In [5]:
```
print ("First few rows...")
display (pd.read_csv ('stateoutflow1112.csv', encoding='latin-1').head (3))

print ("\n...and some from the middle somewhere...")
display (pd.read_csv ('stateoutflow1112.csv', encoding='latin-1').head (1000).tail (3))
```

```
First few rows...
```

	y1_statefips	y2_statefips	y2_state	y2_state_name	n1	n2	AGI
0	1	96	AL	AL Total Migration US and Foreign	51971	107304	2109108
1	1	97	AL	AL Total Migration US	50940	105006	2059642
2	1	98	AL	AL Total Migration Foreign	1031	2298	49465

```
...and some from the middle somewhere...
```

	y1_statefips	y2_statefips	y2_state	y2_state_name	n1	n2	AGI
997	22	13	GA	GEORGIA	2526	4984	83544
998	22	6	CA	CALIFORNIA	2267	3974	89566
999	22	5	AR	ARKANSAS	1355	2851	52356

The `y1_.*` fields describe the state in which the household originated (the "source" vertices) and the `y2_.*` fields describe the state into which the household moved (the "destination"). Column `n1` is the number of such households for the given (source, destination) locations. Notice that there are some special FIPS designators as well, e.g., in the first three rows. These show total outflows, which you can use to normalize counts.

Exercise 1 (3 points). The data files are separated by year. Write some code to merge all of the data into a single Pandas data frame called `StateOutFlows`. It should have the same columns as the original data (e.g., `y1_statefips`, `y2_statefips`), plus an additional `year` column to hold the year.

> Represent the year by a 4-digit value, e.g., `2011` rather than just `11`. Also, use the starting year for the file. That is, if the file is the `1314` file, use `2013` as the year.

In [6]: Student's answer (Top)

```
all_df = []
for yy in range (11, 15):
    filename = "stateoutflow{}{}.csv".format (yy, yy+1)
    df = pd.read_csv (filename, encoding='latin-1')
    df['year'] = 2000 + yy
    all_df.append (df)

StateOutFlows = pd.concat (all_df)
```

In [7]: Grade cell: `StateOutFlows_test` Score: 3.0 / 3.0 (Top)

```
assert 'StateOutFlows' in globals ()
assert type (StateOutFlows) is type (pd.DataFrame ())

print ("Found {} outflow records between 2011-2015.".format (len (StateOutFlows)))
print ("First few rows...")
display (StateOutFlows.head ())

StateOutFlows_soln = pd.read_csv ('StateOutFlows_soln.csv')
assert tibbles_are_equivalent (StateOutFlows, StateOutFlows_soln)

print ("\n(Passed!)")
```

```
Found 11320 outflow records between 2011-2015.
First few rows...
```

	y1_statefips	y2_statefips	y2_state	y2_state_name	n1	n2	AGI	year
0	1	96	AL	AL Total Migration US and Foreign	51971	107304	2109108	2011
1	1	97	AL	AL Total Migration US	50940	105006	2059642	2011
2	1	98	AL	AL Total Migration Foreign	1031	2298	49465	2011
3	1	1	AL	AL Non-migrants	1584665	3603439	87222478	2011
4	1	13	GA	GEORGIA	9920	19470	329213	2011

```
(Passed!)
```

Observe that the `y2_state_name` column has some special values.

For instance, suppose you want to know the *total* number of households that filed returns within the state of Alabama. Evidently, there is a row in each year with `AL Total Migration US and Foreign` as well as an `AL Non-migrants`, the sum of which is presumably the total number of returns.

Exercise 2 (5 points). Create a new Pandas data frame named `Totals` with one row for each state and the following five (5) columns:

- `st`: The two-letter state abbreviation
- `year`: The year of the observation
- `migrated`: The state's `Total Migration US and Foreign` value during that year
- `stayed`: The state's `Non-migrants` value that year
- `all`: The sum of `migrated` and `stayed` columns

> *Hint:* Before proceeding, run the cell below and observe how the strings marking total migrations appear.

In [8]: Student's answer (Top)

```
print ("=== HINT! Observe this hint before proceeding with your solution... ===\n")
print (list (StateOutFlows[StateOutFlows['y2_state'] == 'GA']['y2_state_name'].unique (
)))

def ends_in (pattern, s):
    import re
    return re.match ("^.*{}$".format (pattern), s) is not None

def ends_in_total_migration (s):
    return ends_in ('Total Migration[ -]US and Foreign', s)

def ends_in_non_migrants (s):
    return ends_in ('Non-migrants', s)

migrants = StateOutFlows['y2_state_name'].apply (ends_in_total_migration)
stayed = StateOutFlows['y2_state_name'].apply (ends_in_non_migrants)

Migrated = StateOutFlows[migrants][['y2_state', 'year', 'n1']] \
           .rename (columns={'y2_state': 'st', 'n1': 'migrated'})

Stayed = StateOutFlows[stayed][['y2_state', 'year', 'n1']] \
         .rename (columns={'y2_state': 'st', 'n1': 'stayed'})

Totals = pd.merge (Migrated, Stayed, on=['st', 'year'])
Totals['all'] = Totals['migrated'] + Totals['stayed']
```

```
=== HINT! Observe this hint before proceeding with your solution... ===

['GEORGIA', 'GA Total Migration US and Foreign', 'GA Total Migration US', 'GA Total Migra
tion Foreign', 'GA Non-migrants', 'Georgia', 'GA Total Migration-US and Foreign', 'GA Tot
al Migration-US', 'GA Total Migration-Foreign', 'GA Total Migration-Same State']
```

In [9]: Grade cell: `tidy_totals` Score: 5.0 / 5.0 (Top)

```
Totals_soln = pd.read_csv ('Totals_soln.csv')

assert 'Totals' in globals ()
assert type (Totals) is type (Totals_soln)
assert set (Totals.columns) == set (['st', 'year', 'migrated', 'stayed', 'all'])

print ("Some rows of Totals:")
print (Totals.head ())
print ("...")
print (Totals.tail ())

print ("\n({} rows total.)".format (len (Totals)))

assert tibbles_are_equivalent (Totals, Totals_soln)
```

```
Some rows of Totals:
   st  year  migrated    stayed       all
```

```
0  AL  2011     51971   1584665   1636636
1  AK  2011     19446    258223    277669
2  AZ  2011     91135   2121852   2212987
3  AR  2011     33258    944195    977453
4  CA  2011    266673  13084530  13351203
...
     st  year  migrated   stayed      all
199  VA  2014     91471  2964636  3056107
200  WA  2014     62280  2615285  2677565
201  WV  2014     14869   631644   646513
202  WI  2014     36700  2252810  2289510
203  WY  2014      9834   216928   226762

(204 rows total.)
```

Exercise 3 (2 points). Load the FIPS codes from `fips-state-2010-census.txt`. Store them in a Pandas data frame named `FIPS`. Use the original column names from the input file: `STATE`, `STUSAB`, `STATE_NAME`, `STATENS`.

> Hint: You can use Pandas's `read_csv()` (http://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html) function to read the file. However, be sure to take a look at the file before you try to load it, so you know how to parse by setting the arguments of `read_csv()` appropriately.

In [10]: Student's answer (Top)

```
FIPS = pd.read_csv ('fips-state-2010-census.txt', sep='|')
```

In [11]: Grade cell: `FIPS_test` Score: 2.0 / 2.0 (Top)

```
assert 'FIPS' in globals ()
assert type (FIPS) is type (pd.DataFrame ())
assert len (FIPS) == 57

print ("FIPS data frame, at location 10:\n")
print (FIPS.loc[10])
assert FIPS.loc[10, 'STATE_NAME'] == 'Georgia'

print ("\n(Passed!)")
```

```
FIPS data frame, at location 10:

STATE              13
STUSAB             GA
STATE_NAME    Georgia
STATENS       1705317
Name: 10, dtype: object

(Passed!)
```

Inspect the test code above. Notice that the FIPS code for Georgia is 13, which is located at index position 10 of the data frame (i.e., at `FIPS.loc[10]`).

It would help if the index of the data frame were also the same as the FIPS state code (`STATE`). That way, you could use `FIPS.loc[13]` to get the state code for Georgia; in effect, converting the data frame into something similar to a Python dictionary.

Exercise 4 (2 points). Convert the `STATE` column into an index. To do so, use the Pandas method, `FIPS.set_index()` (http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.set_index.html). Set the arguments to `set_index()` so that the change is made in-place.

In [12]: Student's answer (Top)

```
FIPS.set_index ('STATE', inplace=True)
```

In [13]: Grade cell: `set_index_test` Score: 2.0 / 2.0 (Top)

```
display (FIPS[10:15])

assert set (FIPS.columns) == set (['STUSAB', 'STATE_NAME', 'STATENS'])
assert FIPS.loc[13, 'STATE_NAME'] == 'Georgia'
assert FIPS.loc[15, 'STATE_NAME'] == 'Hawaii'
print ("\n(Passed!)")
```

	STUSAB	STATE_NAME	STATENS
STATE			
13	GA	Georgia	1705317
15	HI	Hawaii	1779782
16	ID	Idaho	1779783
17	IL	Illinois	1779784
18	IN	Indiana	448508

```
(Passed!)
```

Migration edges

Using the code you've set up above, we can build a table of *migration edges*, that is, a succinct summary of the number of households that moved from one state to another, broken down by year. The following code cell does that, leaving the result in a Pandas data frame called `MigrationEdges`.

In [14]:
```
Edges = StateOutFlows[['y1_statefips', 'y2_state', 'year', 'n1']]
Edges = pd.merge (Edges, FIPS[['STUSAB']],
                  left_on='y1_statefips', right_index=True)
Edges.rename (columns={'STUSAB': 'from', 'y2_state': 'to', 'n1': 'moved'}, inplace=True)
del Edges['y1_statefips']

MigrationEdges = Edges[Edges['from'] != Edges['to']]
MigrationEdges.head ()
```

Out[14]:

	to	year	moved	from
4	GA	2011	9920	AL
5	FL	2011	7550	AL
6	TN	2011	4237	AL
7	TX	2011	4121	AL
8	MS	2011	2868	AL

Using the `MigrationEdges` data frame, we can (relatively) easily determine the top 5 states whose households moved to the state of Georgia over all years. Here is one way to do so:

1. Filter rows keeping only those containing `'GA'` as the destination.
2. Group the results by originating state.
3. Sum the results over all years.
4. Sort these results in descending order.
5. Emit just the top 5 results.

In [15]:
```
# Steps 1 and 2
ToGA = (MigrationEdges['to'] == 'GA')
MovedToGA = MigrationEdges[ToGA].groupby ('from')

# Step 3
MovedToGA_counts_by_state = MovedToGA['moved'].sum ()
MovedToGA_counts_by_state[:10]
```

Out[15]: from

```
AK     1978
AL    34691
AR     3321
AZ     5808
CA    25857
CO     6674
CT     4713
DC     1794
DE     1619
FL    89736
Name: moved, dtype: int64
```

In [16]:
```
# Steps 4 and 5: Sort and report the top 5
MovedToGA_counts_by_state.sort_values (ascending=False)[:5]
```

Out[16]:
```
from
FL    89736
TX    36614
AL    34691
NC    29759
SC    27938
Name: moved, dtype: int64
```

Exercise 5 (2 points). Following a similar procedure, determine the top 5 states that Georgians moved to. Store the resulting names and counts in a variable named `GAExodus`.

In [17]: Student's answer (Top)

```
# Steps 1 and 2
FromGA = (MigrationEdges['from'] == 'GA')
MovedFromGA = MigrationEdges[FromGA].groupby ('to')

# Step 3
MovedFromGA_counts_by_state = MovedFromGA['moved'].sum ()

# Steps 4 and 5: Sort and report the top 5
GAExodus = MovedFromGA_counts_by_state.sort_values (ascending=False)[:5]
```

In [18]: Grade cell: `GAExodus_test` Score: 2.0 / 2.0 (Top)

```
assert 'GAExodus' in globals ()
assert type (GAExodus) is type (pd.Series ())
assert len (GAExodus) == 5

print ("=== The exodus from Georgia ===")
assert set (GAExodus.index) == set (['FL', 'TX', 'AL', 'NC', 'SC'])
assert (GAExodus.values == [86178, 50467, 32970, 30352, 30141]).all ()
print (GAExodus)

print ("\n(Passed!)")
```

```
=== The exodus from Georgia ===
to
FL    86178
TX    50467
AL    32970
NC    30352
SC    30141
Name: moved, dtype: int64

(Passed!)
```

In [19]:

problem5 (Score: 17.0 / 17.0)

1. Test cell (Score: 1.0 / 1.0)
2. Test cell (Score: 1.0 / 1.0)
3. Test cell (Score: 2.0 / 2.0)
4. Test cell (Score: 2.0 / 2.0)
5. Test cell (Score: 2.0 / 2.0)
6. Test cell (Score: 3.0 / 3.0)
7. Test cell (Score: 6.0 / 6.0)

Important note!

Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

In [1]:
```
YOUR_ID = "rvuduc3" # Please enter your GT login, e.g., "rvuduc3" or "gtg911x"
COLLABORATORS = ["skaramati3"] # list of strings of your collaborators' IDs
```

In [2]: Grade cell: `who__test` Score: 1.0 / 1.0 (Top)

```
import re

RE_CHECK_ID = re.compile (r'''[a-zA-Z]+\d+|[gG][tT][gG]\d+[a-zA-Z]''')
assert RE_CHECK_ID.match (YOUR_ID) is not None

collab_check = [RE_CHECK_ID.match (i) is not None for i in COLLABORATORS]
assert all (collab_check)

del collab_check
del RE_CHECK_ID
del re
```

Jupyter / IPython version check. The following code cell verifies that you are using the correct version of Jupyter/IPython.

In [3]:
```
import IPython
assert IPython.version_info[0] >= 3, "Your version of IPython is too old, please update it."
```

Problem 5: Density-based Clustering via DBSCAN [16 points]

This problem tests whether you can read an abstract description of a "new" algorithm (or rather, one which most of you might not have seen before) and implement it.

> This problem has six (6) parts or "exercises" and is worth a total of sixteen (16) points.

The algorithm is called DBSCAN (https://en.wikipedia.org/wiki/DBSCAN), which is short for *density-based spatial clustering for applications with noise*. It addresses a limitation of k-means clustering, as described below.

Although there are existing implementations for Python (e.g., see here (http://scikit-learn.org/stable/modules/generated/sklearn.cluster.dbscan.html)), in this notebook we are asking you to build it from scratch, albeit using a lot of scaffolding that we have provided.

Setup

Here are the modules you will need for this problem.

In [4]:
```
from IPython.display import display

import numpy as np
import pandas as pd

%matplotlib inline

%reload_ext autoreload
%autoreload 2
from cse6040utilsfa16 import make_crater, make_scatter_plot, make_scatter_plot2
```

Loading the data

We will work work with a synthetic data set that is an especially bad case for k-means. It's sometimes called the "crater" data because of its shape.

Exercise 1 (1 point). Start by reading the data into a Pandas data frame. The data is stored locally within this assignment in a file called `crater.csv`. Name your data frame `crater`.

In [5]: Student's answer (Top)

```
crater = pd.read_csv ('crater.csv')

display (crater.head (3))
print ("...")
display (crater.tail (3))
```

	x_1	x_2	kmeans_label
0	-1.719159	-2.500154	1
1	-3.002477	-4.801853	1
2	0.469441	1.591334	0

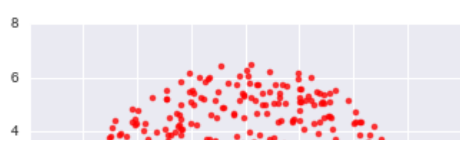
...

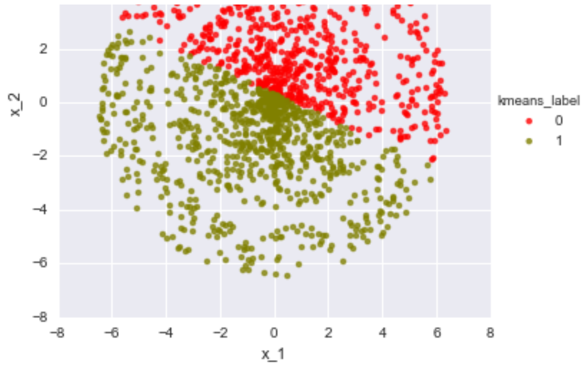
	x_1	x_2	kmeans_label
1497	0.278229	0.162659	1
1498	-0.939762	3.127528	0
1499	-1.506346	5.179035	0

In [6]: Grade cell: `load_crater_test` Score: 1.0 / 1.0 (Top)

```
assert len (crater) == 1500
assert set (crater.columns) == set (['x_1', 'x_2', 'kmeans_label'])

with open ('crater_counts.txt', 'rt') as fp:
    true_counts = [int (c) for c in fp.read ().split (',')]
    assert sum (crater['kmeans_label'] == 0) == true_counts[0]
    assert sum (crater['kmeans_label'] == 1) == true_counts[1]

make_scatter_plot (crater, hue='kmeans_label')

print ("\n(Passed!)")
```

```
(Passed!)
```





The testing code plots these data points, which are 2-D. The colors show the clusters computed by k-means for $k = 2$. Notice that the "natural" structure is, arguably, a dense ball in the middle and a ring (donut) on the outside. However, k-means instead split the points about an arbitrary line that cuts through the middle of the points.

Indeed, this fact is one of the limitations of k-means: it works well when you know the value of k and the k clusters come from Gaussian distributions of similar shape and size (and, therefore, density). However, if you don't know k or there is non-uniform shape and density among the clusters---or some other grouping, as above---then k-means does not work well (qualitatively).

Elements of DBSCAN

The DBSCAN algorithm takes a different approach. Rather than having to provide the number of clusters, k, you define parameters related to *neighborhoods* and *target density*. Let's see how DBSCAN works by building it from the ground up.

Neighborhoods

The first important concept in DBSCAN is that of an ϵ-*neighborhood*.

Consider any point p. The ϵ-neighborhood of p is the set of all points within a distance of ϵ from p. That is, if $\{\hat{x_0}, \hat{x_1}, \ldots, \hat{x_{m-1}}\}$ is a collection of m data points, then the ϵ-neighborhood centered at p is

$$N_\epsilon(p) = \{\hat{x_i} : \|\hat{x_i} - p\|_2 \leq \epsilon\},$$

where the measure of distance is Euclidean (i.e., the two-norm). Notice that this definition would *include* the point p if p is one of the data points.

Exercise 2 (2 points). Implement a function that computes the ϵ-neighborhood of p for a data matrix of points, X, defined by our usual convention as

$$X = \begin{pmatrix} x_0^T \\ x_1^T \\ \vdots \\ x_{m-1}^T \end{pmatrix}.$$

In particular, complete the function named `region_query(p, eps, X)` below. Its inputs are:

- `p[:d]`: The query point, of dimension `d`.
- `eps`: The size of the ball around `p` to search.
- `X[:m, :d]`: The set of points, i.e., data matrix.

It should return a boolean Numpy array `adj[:m]` with one entry per point (i.e., per row of `X`). The entry `adj[i]` should be `True` only if `X[i, :]` lies within the `eps`-sized ball centered at `p`.

> *Hint:* There is a one-line solution of the form, `return (boolean array expression)`.

In [7]: Student's answer (Top)

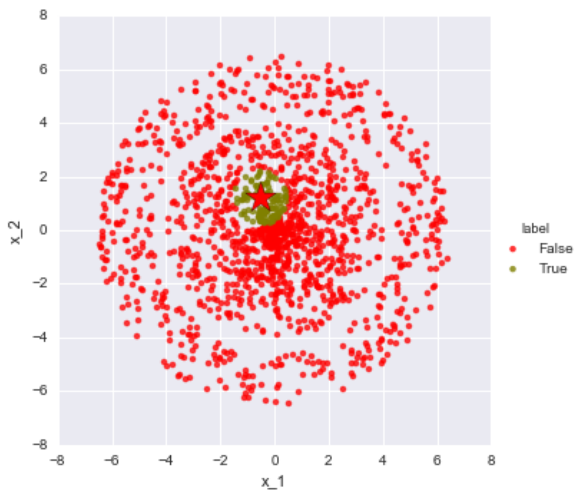
```
def region_query (p, eps, X):
    # These lines check that the inputs `p` and `X` have
    # the right shape.
    _, dim = X.shape
    assert (p.shape == (dim,)) or (p.shape == (1, dim)) or (p.shape == (dim, 1))

    return np.linalg.norm (p - X, axis=1) <= eps
```

Here is the test code for `region_query()`. In addition to sanity-checking your solution, it plots the original points, a query point (marked by a red star), and highlights all points in an ϵ-neighborhood computed by your function so you can visually verify the result. (In this test, $p = (-0.5, 1.2)$ and $\epsilon = 1.0$.)

In [8]: Grade cell: `region_query_test` Score: 2.0 / 2.0 (Top)

```
X = crater[['x_1', 'x_2']].as_matrix ()
p = np.array ([-0.5, 1.2])
in_region = region_query (p, 1.0, X)

crater_ball = crater.copy ()
crater_ball['label'] = in_region
make_scatter_plot (crater_ball, centers=p[np.newaxis])

with open ('region_query_soln.txt', 'rt') as fp:
    assert int (fp.read ()) == sum (in_region)

print ("\n(Passed!)")
```

```
(Passed!)
```



Exercise 3 (2 points). Suppose you are given a vector `y[:]` of boolean (`True` and `False`) values, such as the one computed above. Write a function named `index_set(y)` that returns the index locations of all of `y`'s `True` elements. **Your function must return these index values as a Python set.**

In [9]: Student's answer (Top)

```
def index_set (y):
    """
    Given a boolean vector, this function returns
    the indices of all True elements.
    """
    assert len (y.shape) == 1

    return set (np.where (y)[0])
```

In [10]: Grade cell: `indices_test` Score: 2.0 / 2.0 (Top)

```
y_test = np.array ([True, False, False, True, False, True, True, True, False])
i_soln = set ([0, 3, 5, 6, 7])

i_test = index_set (y_test)
assert type (i_test) is set
assert len (i_test) == len (i_soln)
assert i_test == i_soln

print ("\n(Passed!)")
```

```
(Passed!)
```

Exercise 4 (2 points). Given a value for ϵ and a data matrix X of points, complete the function below so that it determines the neighborhood of each point.

Your function,

```
def find_neighbors(eps, X[:m, :]):
        ...
```

should return a Python list `neighbors[:m]` such that `neighbors[i]` is the index set of neighbors of point `X[i, :]`.

In [11]: Student's answer (Top)

```
def find_neighbors (eps, X):
    m, d = X.shape
    neighbors = [] # Empty list to start
    for i in range (len (X)):
        n_i = index_set (region_query (X[i, :], eps, X))
        neighbors.append (n_i)
    assert len (neighbors) == m
    return neighbors
```

In [12]: Grade cell: `find_neighbors_test` Score: 2.0 / 2.0 (Top)

```
with open ('find_neighbors_soln.csv', 'rt') as fp:
    neighbors = find_neighbors (0.25, X)
    for i, n_i in enumerate (neighbors):
        j_, c_j_ = fp.readline ().split (',')
        assert (i == int (j_)) and (len (n_i) == int (c_j_))

print ("\n(Passed!)")
```

```
(Passed!)
```

Density

The next important concept in DBSCAN is that of the *density* of a neighborhood. Intuitively, the DBSCAN algorithm will try to "grow" clusters around points whose neighborhoods are sufficiently dense.

Let's make this idea more precise.

Definition: *core points*. A point p is a *core point* if its ϵ-neighborhood has at least s points.

In other words, the algorithm now has two user-defined parameters: the neighborhood size, ϵ, and the minimum density, specified using a threshold s on the number of points in such a neighborhood.

Exercise 5 (3 points). Complete the function, `find_core_points(s, neighbors)`, below. It takes as input a minimum-points threshold, `s`, and a list of point neighborhoods, `neighbors[:]`, such that `neighbors[i]` is the (index) set of neighbors of point `i`. It should return a Python set, `core_set`, such that an index `j in core_set` only if the size of the neighborhood at `j` is at least `s`.

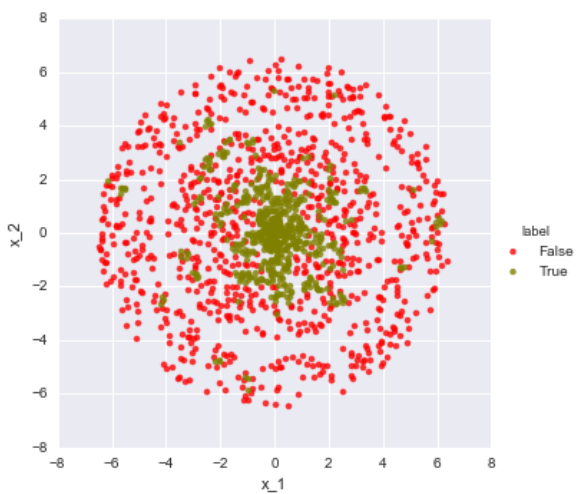
In [13]: Student's answer (Top)

```
def find_core_points (s, neighbors):
    assert type (neighbors) is list
    assert all ([type (n) is set for n in neighbors])

    core_set = set ()
    for i, n_i in enumerate (neighbors):
        if len (n_i) >= s:
            core_set.add (i)
    return core_set
```

In [14]: Grade cell: `find_core_points_test` Score: 3.0 / 3.0 (Top)

```
core_set = find_core_points (5, neighbors)
print ("Found {} core points.".format (len (core_set)))

def plot_core_set (df, core_set):
    df_labeled = df.copy ()
    df_labeled['label'] = False
    df_labeled.ix[list (core_set), 'label'] = True
    make_scatter_plot (df_labeled)

plot_core_set (crater, core_set)

with open ('find_core_points_soln.txt', 'rt') as fp:
    core_set_soln = set ([int (i) for i in fp.read ().split ()])
    assert core_set == core_set_soln

print ("\n(Passed!)")
```

```
Found 623 core points.

(Passed!)
```



Growing clusters via "reachable" points

The last concept needed for DBSCAN is the idea of *growing* a cluster around a core point. It depends on the notion of *reachability*.

Definition: Reachability. A point q is *reachable* from another point p if there exists a sequence of points $p = p_1, p_2, \ldots, p_k = q$ such that every p_i is a core point, possibly except for $p_k = q$, and $p_i \in N_\epsilon(p_{i-1})$ for all $1 < i < k$.

This procedure works as follows.

"Expand Cluster" procedure:

1. Consider any point p that is not yet assigned to a cluster.
2. If p is a core point, then start a new cluster for it.

3. Maintain a "reachable" set, which will be used to hold points that are reachable from p as they are encountered. Initially, the reachable points are just p's ϵ-neighbors.
4. Remove any point q from the reachable set.
5. If q has not yet been visited, then mark it as being visited.
6. If q is also a core point, then add all of its neighbors to the reachable set, per the definition of "reachability" above.
7. If q is not yet assigned to any cluster, then add it to p's cluster.

Notice how this procedure explores the points reachable from p (Step 6). Intuitively, it is trying to join all neighborhoods whose core points are mutually contained.

Here is a brief illustration of these concepts:

In this picture, suppose the minimum density parameter is $s = 3$ points. Thus, only the ϵ-neighborhoods centered at 1, 3, and 6 are core points, since these are the only points that include at least $s = 3$ points. For instance, $N_\epsilon(1) = \{0, 1, 3, 7\}$, making it a core point since its neighborhood has four (4) points, whereas $N_\epsilon(4) = \{3, 4\}$ is not a core point since its neighborhood has just two (2) points.

Exercise 6 (6 points). Implement the "cluster growth" procedure described above in the function, `expand_cluster()`, below.

To simplify your task, we will **give you all the lines of code that you need.** However, you need to figure out in what order these lines must execute, as well as how to indent them!

The signature of the function is:

```
def expand_cluster (p, neighbors, core_set, visited, assignment):
    ...
```

Its parameters are:

- `p` is the *index* of a starting core point. The caller must guarantee that it is indeed a core point, and furthermore, that it has been assigned to some cluster. (See below.)
- `neighbors[:]` is a list of ϵ-neighborhoods, given as Python sets. For instance, `neighbors[p]` is a set of indices of all points in the neighborhood of `p`. It will have been computed from `find_neighbors()` above.
- `core_set` is a Python set containing the indices of all core points. That is, the expression, **`i in core_set`**, is true only if `i` is indeed a core point.
- `visited` is another Python set containing the indices of all points that have already been visited. That is, the expression **`i in visited`** should be `True` only if `i` has been visited. Thus, your `expand_cluster()` function should update this set when visiting any previously unvisited point.
- `assignment` is a Python dictionary. The key is the index of a point; the value is the cluster label to which that point has been assigned. Consequently, if a point `i` does not yet have any cluster assignment, then the expression, **`i in assignment`**, will be **`False`**. Your `expand_cluster()` function should update cluster assignments by updating this dictionary.

The skeleton of `expand_cluster()` does everything up to and including Step 4 of the procedure above. It first initializes the reachable set as a Python set, `reachable`, containing the neighbors of `p`. It then removes one of those reachable points, storing it in `q`. You just need to perform steps 5-7. In fact, we will even give you all of the lines of code that you need! But you have to to incorporate them into the skeleton, ordered and indented correctly.

```
assignment[q] = assignment[p]
if q in core_set:
if q not in assignment:
if q not in visited:
reachable |= neighbors[q]
visited.add (q)
```

In [15]: Student's answer (Top)

```
def expand_cluster (p, neighbors, core_set, visited, assignment):
    # Assume the caller performs Steps 1 and 2 of the procedure.
    # That means 'p' must be a core point that is part of a cluster.
    assert (p in core_set) and (p in visited) and (p in assignment)

    reachable = set (neighbors[p])  # Step 3
    while reachable:
        q = reachable.pop () # Step 4
```

```
        # Put your reordered and correctly indented statements here:
        if q not in visited:
            visited.add (q) # Mark q as visited
            if q in core_set:
                reachable |= neighbors[q]
        if q not in assignment:
            assignment[q] = assignment[p]

    # This procedure does not return anything
    # except via updates to `visited` and
    # `assignment`.
```

In [16]: Grade cell: `expand_cluster_test` Score: 6.0 / 6.0 (Top)

```
# This test is based on the illustration above.
p_test = 3
neighbors_test = [set ([0, 1]),
                  set ([0, 1, 3, 7]),
                  set ([2, 3]),
                  set ([1, 2, 3, 4, 6]),
                  set ([3, 4]),
                  set ([5]),
                  set ([3, 6, 7]),
                  set ([1, 7])
                 ]
core_set_test = set ([1, 3, 6])
visited_test = set ([p_test])
assignment_test = {p_test: 0}
expand_cluster (p_test, neighbors_test, core_set_test,
                visited_test, assignment_test)
assert visited_test == set ([0, 1, 2, 3, 4, 6, 7]) # All but 5
assert set (assignment_test.keys ()) == visited_test
assert set (assignment_test.values ()) == set ([0])

print ("\n(Passed!)")
```

```
(Passed!)
```

Putting it all together

If you've successfully completed all steps above, then you have everything you need to run the final DBSCAN algorithm, which we've provided below. The second code cell below shows a picture of the clusters discovered for a particular setting of neighborhood size, ϵ, and density threshold, s.

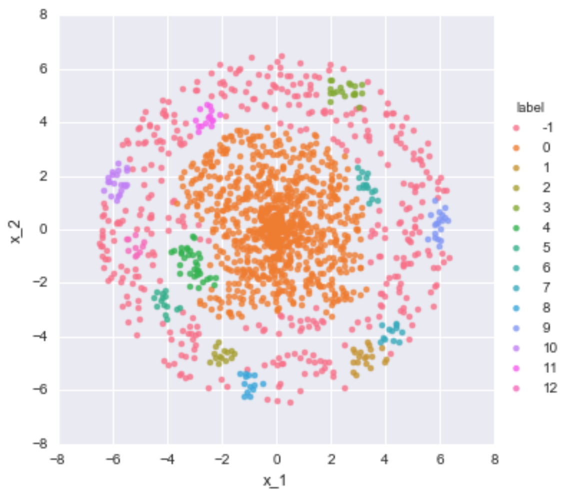
And there is **no additional code for you to write below**! However, you should make sure the remaining cells execute without error.

In [17]:

```
def dbscan (eps, s, X):
    clusters = []
    point_to_cluster = {}

    neighbors = find_neighbors (eps, X)
    core_set = find_core_points (s, neighbors)

    assignment = {}
    next_cluster_id = 0

    visited = set ()
    for i in core_set: # for each core point i
        if i not in visited:
            visited.add (i) # Mark i as visited
            assignment[i] = next_cluster_id
            expand_cluster (i, neighbors, core_set,
                            visited, assignment)
            next_cluster_id += 1

    return assignment, core_set
```

```
In [18]: assignment, core_set = dbscan (0.5, 10, X)

print ("Number of core points:", len (core_set))
print ("Number of clusters:", max (assignment.values ()))
print ("Number of unclassified points:", len (X) - len (assignment))

def plot_labels (df, labels):
    df_labeled = df.copy ()
    df_labeled['label'] = labels
    make_scatter_plot2 (df_labeled)

labels = [-1] * len (X)
for i, c in assignment.items ():
    labels[i] = c
plot_labels (crater, labels)

with open ('dbscan_soln.csv', 'rt') as fp:
    num_cores, num_clusters, num_outliers = fp.read ().split (',')
    assert int (num_cores) == len (core_set)
    assert int (num_clusters) == max (assignment.values ())
    assert int (num_outliers) == (len (X) - len (assignment))
print ("\n(Passed!)")
```

```
Number of core points: 904
Number of clusters: 12
Number of unclassified points: 394

(Passed!)
```




```
In [19]:
```

English