Midterm 2: Solutions

Midterm 2: Human migration — where is everyone going?

Version 1.1 (minor documentation edits from 1.0; no code changes)

This problem will exercise your knowledge of pandas, SQL, Numpy, and basic Python. It has **8** exercises, numbered 0 to 7. There are **18 availa**
However, to earn 100%, the threshold is just **15 points.** (Therefore, once you hit 15, you can stop. There is no extra credit for earning more thar

Each exercise builds logically on the previous one. **However, they may be completed independently.** That is, if you can't solve an exercise, y
move on and try the next one.

The point values of individual exercises are as follows:

- Exercise 0: 2 points
- Exercise 1: 2 point
- Exercise 2: 2 points
- Exercise 3: 2 points
- Exercise 4: 3 points
- Exercise 5: 3 points
- Exercise 6: 1 point
- Exercise 7: 3 points

Pro-tips.

- All test cells use **randomly generated inputs.** Therefore, try your best to write solutions that do not assume too much. To help you debug, test cell does fail, it will often tell you exactly what inputs it was using and what output it expected, compared to yours.
- If you need a complex SQL query, remember that you can define one using a triple-quoted (multiline) string (https://docs.python.org/3.7/tutorial/introduction.html#strings).
- If your program behavior seem strange, try resetting the kernel and rerunning everything.
- If you mess up this notebook or just want to start from scratch, save copies of all your partial responses and use `Actions → Reset Ass` to get a fresh, original copy of this notebook. *(Resetting will wipe out any answers you've written so far, so be sure to stash those somewhe you intend to keep or reuse them!)*
- If you generate excessive output (e.g., from an ill-placed `print` statement) that causes the notebook to load slowly or not at all, use `Acti Clear Notebook Output` to get a clean copy. The clean copy will retain your code but remove any generated output. **However**, it will a **rename** the notebook to `clean.xxx.ipynb`. Since the autograder expects a notebook file with the original name, you'll need to rename t notebook accordingly.

Good luck!

Background

In this problem, you'll analyze how people move from year-to-year within the United States. The main source of data is the US national tax coll known as the Internal Revenue Service (IRS).

When Americans pay their taxes, they are required to tell the IRS when they move. The IRS, in-turn, publishes this information as aggregated s can be used to help study migration patterns.

Let's use these data to try to predict where Americans will live fifty years from now, in the year 2070. You might care about this question becau thinking about where to expand your business, or because you are a policy planner concerned about how the natural pattern of human movem interact with, say, the changing climate.

In this notebook, you'll combine data from two sources:

- The IRS's Tax Migration Data
- Population data, including numbers of births and deaths, as collected by the US Census

Your overall workflow will be as follows:

1. You will use the tax migration data to model the flow of people year-after-year. Your model is a first-order Markov chain, just like PageRank (Notebook 11). The result of this analysis will be a probability transition matrix, which predicts what fraction of people living in one place wi another.
2. You will determine the population in different parts of the US today, using US Census data. The relative populations in each part of the US become the "initial distribution" vector for PageRank.
3. Combining (1) and (2) above, you can run PageRank to determine the "steady-state distribution" of who lives where in a future year (say, 2(

This analysis will allow you to compare the most populous places in the US today with those predicted for the year 2070.

Part 0: Setup

Run the following code cell to preload some standard modules you may find useful for this notebook.

In [1]:
```
import sys
print(f"* Python version: {sys.version}")

import pandas as pd
import numpy as np
import scipy as sp
```

```
* Python version: 3.7.5 (default, Dec 18 2019, 06:24:58)
[GCC 5.5.0 20171010]
```

Run this code cell, which will load some tools needed by the test cells.

In [2]:
```
### BEGIN HIDDEN TESTS
%load_ext autoreload
%autoreload 2
### END HIDDEN TESTS

from testing_tools import load_db, get_db_schema, data_fn
```

Dataset: IRS Tax Migration Data

The first dataset you'll need is a SQLite database containing the tax migration data produced by the IRS. The following cell opens a connectior database, which will be stored in a variable named `conn`.

In [3]:
```
conn = load_db('irs-migration/irs-migration.db')
conn
```

```
Opening database, './resource/asnlib/publicdata/irs-migration/irs-migration.db` ...
        [connection string: 'file:./resource/asnlib/publicdata/irs-migration/irs-migration.db?mo
```

Out[3]: `<sqlite3.Connection at 0x7f1e748e9030>`

The database has three tables, which were created using the following SQL commands:

In [4]:
```
for k, table in enumerate(get_db_schema(conn)):
    print(f'* [Table #{k}]', table[0])
```

```
* [Table #0] CREATE TABLE States (id INTEGER, name TEXT)
* [Table #1] CREATE TABLE Counties (id INTEGER, name TEXT)
* [Table #2] CREATE TABLE Flows (source INTEGER, dest INTEGER, year INTEGER,
                                 num_returns INTEGER, income_thousands INTEGER)
```

Let's inspect the contents of each of these tables.

`States` table

The first table, `States`, is a list of the US's fifty states (provinces). Each has a unique integer ID (`States.id`) and an abbreviated two-letter na (`States.name`). Here are the first few rows of this table:

In [5]:
```
pd.read_sql_query('SELECT * FROM States LIMIT 5', conn)
```

Out[5]:

	id	name
0	1	AL
1	13	GA
2	48	TX
3	12	FL
4	51	VA

For example, the US state of Georgia has an ID of 13 and a two-letter abbreviated name, `"GA"`. (You don't need to know the names, only that t

> This table includes the District of Columbia (`"DC"`), which is technically not a state. However, for the purpose of this notebook, let's pre (https://boundarystones.weta.org/2020/02/12/washington-taxation-without-representation-history).

`Counties` table

Each US state is further subdivided into many counties. These are stored in the table named `Counties`, whose first few rows are as follows:

```
In [6]: pd.read_sql_query('SELECT * FROM Counties LIMIT 5', conn)
```

Out[6]:

	id	name
0	1001	Autauga County
1	1051	Elmore County
2	1101	Montgomery County
3	1021	Chilton County
4	1073	Jefferson County

Each county has a unique integer ID and a name. The names are *not* unique. For instance, there are 8 counties named `"Fulton County"`:

```
In [7]: pd.read_sql_query('SELECT * FROM Counties WHERE name="Fulton County"', conn)
```

Out[7]:

	id	name
0	13121	Fulton County
1	5049	Fulton County
2	17057	Fulton County
3	18049	Fulton County
4	21075	Fulton County
5	42057	Fulton County
6	39051	Fulton County
7	36035	Fulton County

To figure out to which state a given county belongs, you can do the following calculation. Let i be the county ID. Then its state ID is $\left\lfloor \frac{i}{10^3} \right\rfloor$. Tha
county ID, divide it by 1,000, and keep the integer part. For instance, consider the Fulton County whose ID is `13121`. Its state is (13,121 / 1,00
integer part is 13. Recall that 13 is the state ID of `"GA"` (Georgia).

Evidently, Georgia has 159 counties:

```
In [8]: pd.read_sql_query('SELECT * FROM Counties WHERE id >= 13000 AND id < 14000', conn)
```

Out[8]:

	id	name
0	13215	Muscogee County
1	13121	Fulton County
2	13063	Clayton County
3	13067	Cobb County
4	13089	DeKalb County
...	...	...
154	13301	Warren County
155	13197	Marion County
156	13209	Montgomery County
157	13249	Schley County
158	13307	Webster County

159 rows × 2 columns

Exercise 0 (2 points): `count_counties`

Let `conn` be a connection to a database having tables named `States` and `Counties`, similar to the ones from the IRS database. Complete th
`count_counties(conn)`, so that it does the following:

- For each state, count how many counties it has
- Return a *Python dictionary* whose keys are states (taken from `States.name`) and whose values are the number of counties it contains.

For example, if we ran `count_counties(conn)` on the connection object for the tax migration data, the dictionary it returns would include a
of `"GA": 159` (along with all the other states).

Note: The test cell generates a database with *random* data in it. Therefore, you should depend only on the structure of the tables and tl of unique state IDs and unique county IDs, and not on any specific contents or values from the examples above.

In [9]:
```
def count_counties(conn):
    ### BEGIN SOLUTION
    result = count_counties__soln(conn)
    if False: # Set to `True` to test the test code (how meta)
        from random import random, choice
        if random() < 0.1:
            print("*** Perturbation 0: Deleting a random key ***")
            del result[choice(list(result.keys()))]
        if random() < 0.1:
            print("*** Perturbation 1: Changing a (numeric) value ***")
            result[choice(list(result.keys()))] += choice([-1, 1])
    return result

def count_counties__soln(conn):
    from pandas import read_sql_query
    query = """SELECT States.name AS s, COUNT(*) AS k
               FROM States, Counties
               WHERE States.id=(Counties.id/1000)
               GROUP BY States.id
               """
    counts_df = read_sql_query(query, conn)
    return dict(zip(counts_df['s'], counts_df['k']))
    ### END SOLUTION
```

In [10]:
```
# Demo cell
demo_counts = count_counties(conn)
print("Found", len(demo_counts), "states.")
print("The state of 'GA' has", demo_counts['GA'], "counties.")
```

```
Found 51 states.
The state of 'GA' has 159 counties.
```

In [11]:
```
## Test cell: mt2_ex0__count_counties (2 points)

### BEGIN HIDDEN TESTS
def mt2_ex0__gen_soln(fn="state_county_counts.json", overwrite=False):
    from testing_tools import load_db, file_exists, save_json
    if file_exists(fn) and not overwrite:
        print(f"'{fn}' exists; skipping ...")
    else:
        print(f"Generating '{fn}' ...")
        conn = load_db('irs-migration/irs-migration.db')
        counts = count_counties(conn) # assume it works
        conn.close()
        save_json(counts, fn)

mt2_ex0__gen_soln(overwrite=False)
### END HIDDEN TESTS

from testing_tools import mt2_ex0__check
print("Testing...")
for trial in range(250):
    mt2_ex0__check(count_counties)

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

```
'state_county_counts.json' exists; skipping ...
Testing...

(Passed!)
```

`Flows` table: Migration flows

The third table of the IRS data is the "main attraction." It is named `Flows`, and here is a sample:

In [12]:
```
pd.read_sql_query('SELECT * FROM Flows LIMIT 10', conn)
```

Out[12]:

	source	dest	year	num_returns	income_thousands
0	1001	1001	2011	17696	971428
1	1001	1001	2012	17690	1036293
2	1001	1001	2013	17611	1022862
3	1001	1001	2014	18142	1087911
4	1001	1001	2015	17914	1098515
5	1001	1001	2016	17484	1106647
6	1001	1001	2017	18028	1143227

7	1001	1051	2011	445	17682
8	1001	1051	2012	369	15836
9	1001	1051	2013	403	15691

For each year (the `'year'` column), it indicates how many addresses (`'num_returns'`) moved from one county (`'source'`) to another (`'des` example, the last row above shows that in 2013 there were 403 address changes from county 1001 to 1051. But, these source and destination be the same, too, indicating that the address did not change or remained in the same county. For instance, row 5 above shows that in 2016 the 17,484 addresses that remained in county 1001.

If a year is missing for a particular (source, destination) pair, assume the number of returns that year is zero (0).

Part 1: A Markov-chain model of migration

Let's suppose that a reasonable model of how people move follows a first-order Markov process, similar to the one we saw in the PageRank al Topic 11.

That is, let i and j be two counties. We model migration by saying that a person who lives in county i will, each year, decide to move to county probability $p_{i,j}$.

To estimate the $p_{i,j}$ values, let's use the tax migration data stored in the `Flows` table. We'll then store these transition probabilities in a sparse r following four exercises, 1-4, will walk you through this process.

Exercise 1 (2 points): `sum_outflows`

Let `conn` be a connection to a database having a table named `Flows`, just like the one above from the IRS database. Complete the function, `sum_outflows(conn)`, so that it does the following:

- For each source (`Flows.source`), it sums the number of returns (`Flows.num_returns`) over all destinations and years.
- It returns a pandas `DataFrame` with exactly two columns, one named `source` holding the source county ID, and another named `total_r` holding the sum of all returns.

For example, suppose the entire `Flows` table has just two sources, 13125 and 27077, with these entries:

	source	dest	year	num_returns	income_thousands
0	13125	13125	2011	846	34655
1	13125	13125	2012	846	36317
2	13125	13125	2013	847	36034
3	13125	13125	2014	845	38124
4	13125	13125	2015	851	40282
5	13125	13125	2016	801	40094
6	13125	13125	2017	808	40933
7	13125	13163	2011	16	466
8	13125	13163	2012	11	361
9	27077	27077	2011	1586	71766
10	27077	27077	2012	1574	95614
11	27077	27077	2013	1592	81399
12	27077	27077	2014	1639	81974
13	27077	27077	2015	1567	83778
14	27077	27077	2016	1518	80534
15	27077	27077	2017	1567	89557
16	27077	27135	2011	17	532
17	27077	27135	2012	19	622
18	27077	27135	2015	24	1008
19	27077	27135	2017	23	865

Your function would return a data frame that looks like

	source	total_returns
0	13125	5871
1	27077	11126

where the totals are taken over all destinations and years for each of the two sources.

> *Note 0:* The returned columns should have integer dtype.
>
> *Note 1:* The test cell compares your result to the expected one using a function similar to `tibbles_are_equivalent`. Therefore, the of returned sources does *not* matter, but the counts must match exactly (since they are integers).
>
> *Note 2:* Like Exercise 0, the test cell will use a randomly generated database as input.

In [13]:
```
def sum_outflows(conn):
    ### BEGIN SOLUTION
    result = sum_outflows__soln(conn)
    if False: # Set to `True` to test the test code (how meta)
        from random import random, choice, randint
        from numpy import zeros
        if random() < 0.05:
            print("*** Perturbation 0 (deleting random row) ***")
            result = result.sample(len(result)-1)
        elif random() < 0.05:
            print("*** Perturbation 1 (changing one value) ***")
            delta = zeros(len(result), dtype=int)
            delta[randint(0, len(result)-1)] = choice([-1, 1])
            result['total_returns'] += delta
        elif random() < 0.05:
            print("*** Perturbation 2 (changing column type) ***")
            result['total_returns'] = result['total_returns'].astype(float)
    return result

def sum_outflows__soln(conn):
    from pandas import read_sql_query
    query = """
    SELECT source, SUM(num_returns) AS total_returns
    FROM Flows
    GROUP BY source
    """
    return read_sql_query(query, conn)
    ### END SOLUTION
```

In [14]:
```
# Demo cell
demo_sum_outflows = sum_outflows(conn)
demo_sum_outflows[demo_sum_outflows['source'].isin({13125, 27077})]
```

Out[14]:

	source	total_returns
449	13125	5871
1352	27077	11126

In [15]:
```
# Test cell: mt2_ex1__sum_outflows (1 point)

from testing_tools import mt2_ex1__check
print("Testing...")
for trial in range(250):
    mt2_ex1__check(sum_outflows)

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

```
Testing...

(Passed!)
```

Transition probabilities. Let's estimate the transition probability of moving from county i to county j by

$$\frac{\text{total number of returns going from } i \text{ to } j}{\text{total number of returns leaving } i}.$$

Here, "total number" means summed over all years. For instance, consider source 13125. Recall from Exercise 1 that it has a total number of re 5,871. The `Flows` data for 13125 has just two destinations, 13125 (itself) and 13163, as a query for `source=13125` shows:

	source	dest	year	num_returns	income_thousands
0	13125	13125	2011	846	34655
1	13125	13125	2012	846	36317
2	13125	13125	2013	847	36034
3	13125	13125	2014	845	38124
4	13125	13125	2015	851	40282
5	13125	13125	2016	801	40094
6	13125	13125	2017	808	40933
7	13125	13163	2011	16	466
8	13125	13163	2012	11	361

The total number of returns from 13125 to 13125 (i.e., itself), summed over all years, is 846+846+847+845+851+801+808=5,844. Therefore, its probability is (5,844 / 5,871) ≈ 0.995. The total number of (13125, 13163) returns is just 16+11=27. Therefore, its transition probability is (27 / 5,

Exercise 2 (2 points): `estimate_probs`

Let `conn` be a connection to a database having a table named `Flows` like the one from the IRS database. Complete the function, `estimate_p` below. This function should return a pandas `DataFrame` with three columns: `source` (taken from `Flows.source`), `dest` (taken from `Flows.c` `prob`, which is the transition probability going from `source` to `dest`.

From our earlier discussion, recall that the formula for the transition probability is

$$\frac{\text{total number of returns going from } i \text{ to } j}{\text{total number of returns leaving } i}.$$

For the example above, your function would return

source	dest	prob
13125	13125	0.995401
13125	13163	0.00459888

Note: Your function should only return rows containing (`source`, `dest`) pairs that exist in the `Flows` table of the given `conn` database connection. As in previous exercises, the `conn` your function is given will contain randomly generated data.

Hint: If you use SQL to compute this table, note that dividing one integer by another produces a (truncated) integer result. Since probal should be floating-point values, you may need to cast your result explicitly, per this Stackoverflow post (https://stackoverflow.com/questions/8305613/converting-int-to-real-in-sqlite).

In [16]:
```
def estimate_probs(conn):
    ### BEGIN SOLUTION
    result = estimate_probs__v0(conn)
    if False: # Set to `True` to test the test code (how meta)
        from random import random, choice, randint
        from numpy import ones
        if random() < 0.05:
            print("*** Perturbation 0 (deleting random row) ***")
            result = result.sample(len(result)-1)
        elif random() < 0.05:
            print("*** Perturbation 1 (changing one value) ***")
            delta = ones(len(result))
            k = randint(0, len(result)-1)
            delta[k] = choice([0.99, 1.01])
            result['prob'] *= delta
        elif random() < 0.05:
            print("*** Perturbation 2 (changing column type) ***")
            result['prob'] = result['prob'].astype(int)
    return result

# This solution mixes pandas and SQL
def estimate_probs__v0(conn):
    from pandas import read_sql_query
    query = """
    SELECT source, dest, SUM(num_returns) as num_returns
    FROM Flows
    GROUP BY source, dest
    """
    pairs_counts = read_sql_query(query, conn)
    totals = sum_outflows(conn)
    result = pairs_counts.merge(totals, on='source')
    result['prob'] = result['num_returns'] / result['total_returns']
    result = result.drop(['num_returns', 'total_returns'], axis=1)
    return result

# This solution uses only SQL
def estimate_probs__v1(conn):
    from pandas import read_sql_query
    query = '''
    SELECT Flows.source, Flows.dest, CAST(SUM(Flows.num_returns) AS REAL)/Totals.total AS prob
        FROM Flows, (SELECT source, SUM(num_returns) AS total
                     FROM Flows
                     GROUP BY source) AS Totals
        WHERE Flows.source=Totals.source
        GROUP BY Flows.source, Flows.dest
        ORDER BY prob DESC
    '''
    return pd.read_sql_query(query, conn)
    ### END SOLUTION
```

In [17]:
```
# Demo cell
demo_probs = estimate_probs(conn)
demo_probs[demo_probs['source'] == 13125]
```

Out[17]:

	source	dest	prob
28433	13125	13125	0.995401
28434	13125	13163	0.004599

In [18]:
```
# Test cell: mt2_ex2__estimate_probs (2 points)

### BEGIN HIDDEN TESTS
def mt2_ex2__gen_soln(fn="probs.csv", overwrite=False):
    from testing_tools import load_db, file_exists, save_df
    if file_exists(fn) and not overwrite:
        print(f"'{fn}' exists; skipping ...")
    else:
        print(f"Generating '{fn}' ...")
        conn = load_db('irs-migration/irs-migration.db')
        probs = estimate_probs(conn) # assume it works
        conn.close()
        save_df(probs, fn)

mt2_ex2__gen_soln(overwrite=False)
### END HIDDEN TESTS

from testing_tools import mt2_ex2__check
print("Testing...")
for trial in range(250):
    mt2_ex2__check(estimate_probs)

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

```
'probs.csv' exists; skipping ...
Testing...

(Passed!)
```

Converting logical county IDs to "physical" indices. Recall that to construct a sparse matrix using Numpy/Scipy, we will need to convert "lo IDs, which might be arbitrary, into "physical" indices that lie in the range [0, n-1] (inclusive), where `n` is the number of unique county IDs.

In our case, the `Counties` table gives us a natural way to do that. Suppose we run the query,

```
SELECT * FROM Counties ORDER BY id
```

The output might look like the following:

	id	name
0	1001	Autauga County
1	1003	Baldwin County
2	1005	Barbour County
...	...	...
3141	56041	Uinta County
3142	56043	Washakie County
3143	56045	Weston County

Observe that the *index* values are numbered sequentially, from 0 to 3143. Thus, there are 3,144 unique county IDs. So, we can map the logical `1001` to the physical integer index `0`, `1003` to `1`, ..., `56043` to `3142`, and `56045` to `3143`.

Exercise 3 (2 points): `map_counties`

Let `conn` be a connection to a database having a table named `Counties` like the one from the IRS database. Complete the function, `map_counties(conn)`, so that it does the following.

- Runs a query that orders the county IDs in ascending order, obtaining a pandas `DataFrame` like the one shown above.
- Returns a *Python dictionary* where each key is a logical integer county ID and the corresponding value is the physical integer index.

For instance, when run on the IRS database, the output dictionary would contain the key-value pairs, `1001: 0`, `1003: 1`, ..., `56045: 3143`.

> *Note:* The test cell generates a database with *random* data in it. Therefore, you should depend only on the structure of the tables and tl of unique state IDs and unique county IDs, and not on any specific contents or values from the examples above.

In [19]:
```
def map_counties(conn):
    ### BEGIN SOLUTION
    result = map_counties__soln(conn)
    if False: # Set to `True` to test the test code (how meta)
        from random import random, choice
        if random() < 0.1:
```

```
            print("*** Perturbation 0: Deleting a random key ***")
            del result[choice(list(result.keys()))]
        if random() < 0.1:
            print("*** Perturbation 1: Changing a (numeric) value ***")
            result[choice(list(result.keys()))] += choice([-1, 1])
    return result

def map_counties__soln(conn):
    from pandas import read_sql_query
    counties = read_sql_query('SELECT * FROM Counties ORDER BY id', conn)
    return dict(zip(counties['id'], counties.index))
    ### END SOLUTION
```

In [20]:
```
# Demo cell

demo_map_counties = map_counties(conn)

for i in [1001, 1003, 1005, None, 56041, 56043, 56045]:
    if i is None:
        print("...")
        continue
    print(i, "==>", demo_map_counties[i])
```

```
1001 ==> 0
1003 ==> 1
1005 ==> 2
...
56041 ==> 3141
56043 ==> 3142
56045 ==> 3143
```

In [21]:
```
# Test cell: mt2_ex3__map_counties (2 points)

### BEGIN HIDDEN TESTS
def mt2_ex3__gen_soln(fn="map_counties.json", overwrite=False):
    from testing_tools import load_db, file_exists, save_json
    if file_exists(fn) and not overwrite:
        print(f"'{fn}' exists; skipping ...")
    else:
        print(f"Generating '{fn}' ...")
        conn = load_db('irs-migration/irs-migration.db')
        counties = map_counties(conn) # assume it works
        conn.close()
        save_json(counties, fn)

mt2_ex3__gen_soln(overwrite=False)
### END HIDDEN TESTS

from testing_tools import mt2_ex3__check
print("Testing...")
for trial in range(250):
    mt2_ex3__check(map_counties)

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

```
'map_counties.json' exists; skipping ...
Testing...

(Passed!)
```

Exercise 4 (3 points)

Suppose you are given the following two inputs:

- `probs`: A data frame produced by Exercise 2, `estimate_probs`. This data frame has three columns, `source`, `dest`, and `prob`, where ea the transition probability (`prob`) for a particular source-destination pair (`source`, `dest`).
- `county_map`: A Python dictionary that maps logical county IDs to physical indices, per Exercise 3.

Complete the function, `make_matrix(probs, county_map)`, so that it returns a probability transition matrix in Scipy's sparse COO format. should use Scipy's coo_matrix (https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.coo_matrix.html) function to construct this r matrix should be `n`-by-`n`, where `n` is the number of unique county IDs, and it should only have nonzero entries where `probs` has an entry.

In [22]:
```
def make_matrix(probs, county_map):
    from scipy.sparse import coo_matrix
    assert isinstance(probs, pd.DataFrame)
    assert isinstance(county_map, dict)
    ### BEGIN SOLUTION
    result = make_matrix__soln(probs, county_map)
    if False: # Set to `True` to test the test code (how meta)
        from random import random, randint, choice
        if random() < 0.1:
            k = randint(0, len(result.data)-1)
            print(f"*** Perturbation 0: Changing a random nonzero value ({k}) ***")
            result.data[k] *= choice([0.99, 1.01])
        elif random() < 0.1:
```

```
            k = randint(0, len(result.row)-1)
            print(f"*** Perturbation 1: Changing a row index ({k}) ***")
            result.row[k] += choice([-1, 1])
        elif random() < 0.1:
            k = randint(0, len(result.col)-1)
            print(f"*** Perturbation 2: Changing a column index ({k}) ***")
            result.col[k] += choice([-1, 1])
    return result

def make_matrix__soln(probs, county_map):
    from scipy.sparse import coo_matrix
    num_counties = len(county_map)
    rows = probs['source'].map(county_map)
    cols = probs['dest'].map(county_map)
    vals = probs['prob']
    return coo_matrix((vals, (rows, cols)), shape=(num_counties, num_counties))
    ### END SOLUTION
```

In [23]:

```
# Demo cell
demo_P = make_matrix(demo_probs, demo_map_counties)
demo_n = max(demo_map_counties.values())+1
print("* Shape:", demo_P.shape, "should equal", (demo_n, demo_n))
print("* Number of nonzeros:", demo_P.nnz, "should equal", len(demo_probs))
```

```
* Shape: (3144, 3144) should equal (3144, 3144)
* Number of nonzeros: 110295 should equal 110295
```

In [24]:

```
# Test cell: mt2_ex4__make_matrix (3 points)

### BEGIN HIDDEN TESTS
def mt2_ex4__gen_soln(fn="matrix.pickle", overwrite=False):
    from testing_tools import load_db, file_exists, save_pickle
    if file_exists(fn) and not overwrite:
        print(f"'{fn}' exists; skipping ...")
    else:
        print(f"Generating '{fn}' ...")
        conn = load_db('irs-migration/irs-migration.db')
        P = make_matrix(estimate_probs(conn), map_counties(conn))
        conn.close()
        save_pickle(P, fn)

mt2_ex4__gen_soln(overwrite=False)
### END HIDDEN TESTS

from testing_tools import mt2_ex4__check
print("Testing...")
for trial in range(250):
    mt2_ex4__check(make_matrix)

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

```
'matrix.pickle' exists; skipping ...
Testing...

(Passed!)
```

Part 2: Calculating the initial distribution

Recall that to run a PageRank-style model, we need an initial probability distribution. In our case, we want to know what is the probability that a
US lives in a particular location (county ID). Exercise 5 estimates that probability using a new data source: the US Census Bureau's population

The following code cell loads this data into a pandas `DataFrame` named `population` and inspects its contents.

In [25]:

```
def load_pop_data(fn='census/co-est2019-alldata.csv'):
    pop = pd.read_csv(data_fn(fn), encoding='latin_1')
    pop = pop[['STATE', 'COUNTY', 'POPESTIMATE2019', 'BIRTHS2019', 'DEATHS2019']]
    pop = pop[pop['COUNTY'] > 0]
    pop = pop[(pop['STATE'] != 15) & (pop['COUNTY'] != 5)]
    return pop

population = load_pop_data()
population.sample(5) # Show 5 randomly selected rows
```

Out[25]:

	STATE	COUNTY	POPESTIMATE2019	BIRTHS2019	DEATHS2019
96	2	282	579	4	1
1513	29	9	35789	410	403
2708	48	283	7520	92	41
2062	38	75	2327	17	26
967	20	125	31829	322	356

This dataframe has one row per county. The county ID is split into two separate columns, one holding the state ID (`STATE`) and another holding specific county sub-ID (`COUNTY`). So if the IRS county ID is 13125, you would see `13` for `STATE` and `125` for `COUNTY`.

The remaining three columns show

- the estimated number of people living in that county in 2019 (`POPESTIMATE2019`);
- the number of births in that county in 2019 (`BIRTHS2019`);
- and the number of deaths in that county in 2019 (`DEATHS2019`).

Exercise 5 (3 points): `normalize_pop`

Let `population` be a pandas `DataFrame` similar to the one defined above. That is, it has one row per county, and the columns `STATE`, `COUN` `POPESTIMATE2019`.

Let `county_map` be a mapping of logical county IDs to physical indices, per Exercise 3.

Complete the function, `normalize_pop(population)`, so that it returns a 1-D Numpy array such that

1. there is one entry per county; and
2. each element is the county's population divided by the *total* population (sum of the `POPESTIMATE2019` column), stored as a floating-poin

For example, suppose `population` had the following five rows,

	STATE	COUNTY	POPESTIMATE2019	BIRTHS2019	DEATHS2019
0	47	69	25050	218	289
1	50	1	36777	299	341
2	26	117	63888	728	613
3	55	23	16131	151	181
4	22	99	53431	663	537

Further suppose that `county_map == {47069: 2, 50001: 3, 26117: 1, 55023: 4, 22099: 0}`. The total population is 25050+36777+63888+16131+53431 = 195,277. Thus, `normalize_pop(population, county_map)` should return,

```
array([0.27361645, 0.32716603, 0.12827932, 0.18833247, 0.08260573])
```

For instance, county 22099 is assigned the `0` index according to `county_map`. And since its estimated population in 2019 is 53431, its normali population is 53431/195277 ≈ 0.2736...

> *Note:* Your function must *not* modify the `population` data frame! If it needs to manipulate the input, then it should make a copy.

In [26]:
```python
def normalize_pop(population, county_map):
    assert isinstance(population, pd.DataFrame)
    assert set(population.columns) == {'BIRTHS2019', 'COUNTY', 'DEATHS2019', 'POPESTIMATE2019',
    ### BEGIN SOLUTION
    pop = population[['STATE', 'COUNTY', 'POPESTIMATE2019']].copy()
    result = normalize_pop__soln_inplace(pop, county_map)
    if False:
        from random import random, randint, choice
        from numpy import reshape
        from testing_tools import prime_factors
        if random() <= 0.05 and len(result) > 3:
            print("*** Perturbation 0: Changing the shape ***")
            result = reshape(result, prime_factors(len(result)))
        elif random() <= 0.05:
            print("*** Perturbation 1: Removing an element ***")
            result = result[:-1]
        elif random() <= 0.05:
            k = randint(0, len(result)-1)
            print("*** Perturbation 2: Changing a value ({k}) ***")
            result[k] *= choice([0.99, 1.01])
    return result

def normalize_pop__soln_inplace(pop, county_map):
    from numpy import empty
    pop['i'] = (pop['STATE']*1000 + pop['COUNTY']).map(county_map)
    total = pop['POPESTIMATE2019'].sum()
    pop['x'] = pop['POPESTIMATE2019'] / total
    dist0 = empty(len(county_map))
    dist0[pop['i']] = pop['x']
    return dist0
    ### END SOLUTION
```

In [27]:
```python
# Demo cell
demo_pop = population[population.apply(lambda row: (row['STATE'], row['COUNTY']) in [(47, 69), (
6, 117), (55, 23), (22, 99)], axis=1)]
demo_map = {47069: 2, 50001: 3, 26117: 1, 55023: 4, 22099: 0}
normalize_pop(demo_pop, demo_map)
```

```
Out[27]: array([0.27361645, 0.32716603, 0.12827932, 0.18833247, 0.08260573])
```

In [28]:
```python
# Test cell: mt2_ex5__normalize_pop (3 points)

### BEGIN HIDDEN TESTS
def mt2_ex5__gen_soln(fn="dist0.pickle", overwrite=False):
    from testing_tools import data_fn, load_db, file_exists, save_pickle
    from pandas import read_csv
    if file_exists(fn) and not overwrite:
        print(f"'{fn}' exists; skipping ...")
    else:
        print(f"Generating '{fn}' ...")
        conn = load_db('irs-migration/irs-migration.db')
        county_map = map_counties(conn)
        conn.close()
        pop = load_pop_data()
        x0 = normalize_pop(pop, county_map)
        save_pickle(x0, fn)

mt2_ex5__gen_soln(overwrite=False)
### END HIDDEN TESTS

from testing_tools import mt2_ex5__check
print("Testing...")
for trial in range(250):
    mt2_ex5__check(normalize_pop)

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

```
'dist0.pickle' exists; skipping ...
Testing...

(Passed!)
```

Exercise 6 (1 point): Estimating the total future population

The `population` dataframe also includes birth and death information. From that, let's try to estimate the *overall* total population in future years following procedure:

- From the data frame, calculate the total number of people, the total number of births, and the total number of deaths in 2019. That is, we c about these values by location, but rather their overall sums.
- Let n_0 be the total population in 2019, as calculated above.
- Let b_0 be the total births in 2019. Define the *overall birth rate* as $\beta \equiv \frac{b_0}{n_0}$.
- Let d_0 be the total deaths in 2019. Define the *overall death rate* as $\delta \equiv \frac{d_0}{n_0}$.
- Assume that, overall, the birth and death rates remain constant over time. Then to estimate the total population t years from now, calculate $n_t \equiv n_0(1 + \beta - \delta)^t$.

Implement this procedure as the function, `estimate_pop(population, t)`, below. That is, it should take as input the population data (`pop` similar to Exercise 5) and target years from now (`t`). It should then return the corresponding value of n_t as a floating-point number.

For example, suppose `population` had exactly the following five rows:

	STATE	COUNTY	POPESTIMATE2019	BIRTHS2019	DEATHS2019
0	47	69	25050	218	289
1	50	1	36777	299	341
2	26	117	63888	728	613
3	55	23	16131	151	181
4	22	99	53431	663	537

In this case, the total population is 195,277 people. If you follow the above procedure, you should get an estimated population at `t=50` years la `200237.73486678504`. *(You do not need to round your result explicitly.)*

In [29]:
```python
def estimate_pop(population, t):
    assert isinstance(population, pd.DataFrame)
    assert set(population.columns) == {'BIRTHS2019', 'COUNTY', 'DEATHS2019', 'POPESTIMATE2019',
    assert isinstance(t, int) and t >= 0
    ### BEGIN SOLUTION
    result = estimate_pop__soln(population, t)
    if False:
        from random import random, randint
        if random() <= 0.05:
            print("*** Perturbation: Changing answer ***")
            result *= 1 + randint(1, 9)/10
        elif random() <= 0.05:
            print("*** Perturbation: Changing type of result ***")
            result = int(result)
    return result

def estimate_pop__soln(population, t):
    n0 = population['POPESTIMATE2019'].sum()
```

```
    b0 = population['BIRTHS2019'].sum()
    d0 = population['DEATHS2019'].sum()
    beta = b0 / n0
    delta = d0 / n0
    return n0 * ((1 + beta - delta)**t)
    ### END SOLUTION
```

In [30]:
```
# Demo cell
demo_pop = population[population.apply(lambda row: (row['STATE'], row['COUNTY']) in [(47, 69), (
6, 117), (55, 23), (22, 99)], axis=1)]
estimate_pop(demo_pop, 50)
```

Out[30]: 200237.73386678504

In [31]:
```
# Test cell: mt2_ex6__estimate_pop (1 point)

from testing_tools import mt2_ex6__check
print("Testing...")
for trial in range(1000):
    mt2_ex6__check(estimate_pop)

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

```
Testing...

(Passed!)
```

Part 3: Richest (per capita) counties

The IRS tax data set includes information about income. While not a direct representation of "wealth," let's treat it as an indicator and rank the this reported income.

"Income per return" by county. Quickly recall the structure of the `Flows` table:

	source	dest	year	num_returns	income_thousands
			...		
4	13125	13125	2015	851	40282
5	13125	13125	2016	801	40094
6	13125	13125	2017	808	40933
7	13125	13163	2011	16	466
8	13125	13163	2012	11	361
			...		

The column named `"income_thousands"` is the total reported income in thousands of US dollars for all of the returns. For instance, in row 4 total income in 2015 across all 851 returns filed (and staying within) county 13125 was 40,282,000 USD, which is about 47,334.90 USD per retu

When the `"source"` and `"dest"` are the same, all of this income "belongs" to a given county. But what happens when they differ? For examp county 2068:

	source	dest	year	num_returns	income_thousands
0	2020	2068	2011	14	683
1	2020	2068	2012	10	318
2	2068	2068	2011	854	58637
3	2068	2068	2012	842	59815
4	2068	2068	2013	832	60200
5	2068	2068	2014	855	67174
6	2068	2068	2015	866	66860
7	2068	2068	2016	837	61617
8	2068	2068	2017	858	65833
9	2068	2170	2011	10	600
10	2068	2170	2012	11	793
11	2068	2090	2012	24	1433
12	2068	2090	2015	21	1970
13	2068	2090	2016	20	1661
14	2068	2090	2017	23	1223

15	2068	2020	2012	20	890
16	2090	2068	2011	20	794
17	2090	2068	2012	18	1098
18	2122	2068	2011	10	325

Rows 2-8 have the same values for `"source"` and `"dest"`. Let's call these **self-flows.** But rows 0-1 and 16-18 have only `"dest"` values as 2 them **inflows** from other counties into 2068; and rows 9-15 are have only `"source"` equal to 2068, making them **outflows** from 2068 to other

Thus, to estimate the total income per return in a given county, let's use the following formula. It keeps all self-flow income, but "splits the diffe inflow and outflow income:

$$(\text{total income per return}) = \frac{(\text{total self-flow income}) + \frac{1}{2}[(\text{total inflow income}) + (\text{total outflow income})]}{(\text{total self-flow returns}) + \frac{1}{2}[(\text{total inflow returns}) + (\text{total outflow returns})]}$$

For the county 2068 example, this value is

$$\frac{440\,136\,000 + 4\,285\,000 + 1\,609\,000}{5\,944 + 36 + 64.5} \approx 73\,791.05 \text{ USD.}$$

Exercise 7 (3 points): Ranking counties by income per return

Given a connection `conn` to a database containing a tax `Flows` table, complete the function `calc_ipr(conn)` to calculate the income per ret county.

In particular, it should return a pandas `DataFrame` with one row per unique county ID and the following columns:

- `'county_id'`: County IDs
- `'ipr'`: Income per return, in *dollars*, as floating-point values.

> *Note 0:* Regarding the `'ipr'` column, recall that the `'income_thousands'` column of the `Flows` table uses *thousands* of dollars. So has the value, `123`, that should be treated and converted to `123,000`.
>
> *Note 1:* Note that some counties might not have inflows or outflows. So, you'll need to be able to handle that case.

In [32]:
```
def calc_ipr(conn):
    ### BEGIN SOLUTION
    self_income = get_income(conn, 'self')
    out_income = get_income(conn, 'inflow')
    in_income = get_income(conn, 'outflow')
    income = self_income.merge(out_income, how='outer', on='county_id').merge(in_income, how='ou
'county_id')
    income = income.fillna(0)
    income['total_income'] = income['self_income'] + 0.5*(income['inflow_income'] + income['out
e'])
    income['total_returns'] = income['self_returns'] + 0.5*(income['inflow_returns'] + income['
urns'])
    income['ipr'] = 1000.0 * income['total_income'] / income['total_returns']
    return income[['county_id', 'ipr']]

def get_income(conn, endpoint):
    from pandas import read_sql_query
    assert endpoint in {'self', 'inflow', 'outflow'}

    if endpoint == 'self':
        operator = '='
        group_by = 'source'
    else:
        operator = '<>'
        group_by = 'source' if endpoint == 'outflow' else 'dest'

    query = f'''
    SELECT {group_by} AS county_id,
           SUM(income_thousands) AS {endpoint}_income,
           SUM(num_returns) AS {endpoint}_returns
        FROM Flows
        WHERE source {operator} dest
        GROUP BY {group_by}
    '''
    return read_sql_query(query, conn)
### END SOLUTION
```

In [33]:
```
# Demo cell 0: Should get approximately 73791.05 for county 2068
income = calc_ipr(conn)
income[income['county_id'] == 2068]
```

Out[33]:

	county_id	ipr

	county_id	ipr
72	2068	73791.049715

In [34]:
```
# Demo cell 1: print top 5 counties by `ipr`
income.sort_values(by='ipr', ascending=False).head(5)
```

Out[34]:

	county_id	ipr
3140	56039	292092.540613
2679	48311	219276.753597
1858	36061	218857.822032
2610	48173	193962.756645
309	9001	176689.884842

In [35]:
```
# Test cell: mt2_ex7__calc_ipr (3 points)

### BEGIN HIDDEN TESTS
def mt2_ex7__gen_soln(fn="incomes.csv", overwrite=False):
    from testing_tools import data_fn, load_db, file_exists, save_pickle
    if file_exists(fn) and not overwrite:
        print(f"'{fn}' exists; skipping ...")
    else:
        print(f"Generating '{fn}' ...")
        conn = load_db('irs-migration/irs-migration.db')
        ipr = calc_ipr(conn).sort_values(by='ipr', ascending=False)
        conn.close()
        ipr.to_csv(data_fn(fn), index=False)

mt2_ex7__gen_soln(overwrite=False)
### END HIDDEN TESTS

from testing_tools import mt2_ex7__check
print("Testing...")
for trial in range(5):
    mt2_ex7__check(calc_ipr)

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

```
'incomes.csv' exists; skipping ...
Testing...

(Passed!)
```

Part 4: Putting it all together

We are now ready to put it all together, and see how migration might affect which parts of the US are most populous.

Running "PageRank." Your initial distribution (Exercise 5) provides one ranking of the cities. If we run PageRank using the probability transitior models migration (Exercises 2-4), we'll get a new ranking.

This code cell runs PageRank. (You should recognize it from Notebook 11!) It gives you the final results in a `DataFrame` named `rankings`.

In [36]:
```
from testing_tools import load_pickle, load_json

# Run PageRank
P = load_pickle('matrix.pickle')
x_0 = load_pickle('dist0.pickle') # initial distribution
x = x_0.copy() # holds final distribution
for _ in range(50):
    x = P.T.dot(x)
x_final = x.copy()

# Build DataFrame
def get_ranking(x):
    k = np.argsort(x)
    r = np.empty(len(x), dtype=int)
    r[k] = np.arange(len(x))
    return r

# Get population ranking
county_map = load_json('map_counties.json') # county IDs -> physical indices
inv_county_map = {v: k for k, v in county_map.items()} # physical indices -> county IDs

rankings = pd.DataFrame({'county_id': [inv_county_map[k] for k in range(len(county_map))],
                         'rank_2019': get_ranking(-x_0), 'x_2019': x_0,
                         'rank_2070': get_ranking(-x_final), 'x_2070': x_final})
rankings['county_id'] = rankings['county_id'].astype(int)

# Add income data
top_incomes = pd.read_csv(data_fn('incomes.csv'))
```

```
top_incomes['rank_ipr'] = top_incomes.index

# Construct location metadata
locations = pd.read_sql_query("""SELECT Counties.id AS county_id,
                                        Counties.name||', '||States.name AS name
                                 FROM Counties, States
                                 WHERE Counties.id/1000 == States.id""", conn)

# Merge
rankings = rankings.merge(locations, how='left', on='county_id') \
                   .merge(top_incomes, how='left', on='county_id') \
                   [['county_id', 'name', 'rank_2019', 'rank_2070', 'x_2019', 'x_2070', 'ipr',
]]
rankings.head()
```

```
'./resource/asnlib/publicdata/map_counties.json': 3144
```

Out[36]:

	county_id	name	rank_2019	rank_2070	x_2019	x_2070	ipr	rank_ipr
0	1001	Autauga County, AL	893	960	0.000175	0.000160	58862.401810	1096
1	1003	Baldwin County, AL	297	251	0.000698	0.000869	65972.008699	625
2	1005	Barbour County, AL	3143	2136	0.000000	0.000040	42138.228093	2849
3	1007	Bibb County, AL	1669	1808	0.000070	0.000058	50647.404527	1943
4	1009	Blount County, AL	869	905	0.000181	0.000174	52721.911220	1706

For each county (`county_id`), it shows the initial ranking by population in the year 2019 (`rank_2019`), as well as the predicted ranking in the y
also merges in the income-per-return measure, and adds a new column, called `rank_ipr`, that includes the rank of the county by income-per-
of 0 would mean that county has the highest income-per-return in the entire country.)

Let's take a look at the top 10 counties by their Year 2070 ranking:

In [37]:

```
# View Top 10 according to their year 2070 rankings:
rankings.sort_values(by='rank_2070').head(10)
```

Out[37]:

	county_id	name	rank_2019	rank_2070	x_2019	x_2070	ipr	rank_ipr
205	6037	Los Angeles County, CA	0	0	0.031396	0.027006	77724.713930	271
104	4013	Maricopa County, AZ	3	1	0.014027	0.015782	70382.137085	438
2624	48201	Harris County, TX	2	2	0.014740	0.015218	81912.760833	199
610	17031	Cook County, IL	1	3	0.016107	0.013073	79794.982548	230
223	6073	San Diego County, CA	4	4	0.010440	0.010454	79569.686413	234
216	6059	Orange County, CA	5	5	0.009931	0.009351	92588.799219	120
2580	48113	Dallas County, TX	7	6	0.008242	0.009090	77802.364011	269
1748	32003	Clark County, NV	10	7	0.007089	0.008654	64569.871861	699
2971	53033	King County, WA	12	8	0.007045	0.008134	112279.144526	52
2743	48439	Tarrant County, TX	14	9	0.006575	0.007622	76316.221563	295

You should observe that counties in the top 6 (ranks 0-5) remain there, but the ordering changes; and three counties that previously were not in
now join those ranks (Clark County, NV; King County, WA; and Tarrant County, TX).

Fin! You've reached the end of Midterm 2. Don't forget to restart and run all cells again to make sure it's all working when run in sequence; and
your work passes the submission process. Good luck!

Epilogue. The analysis in this notebook is very rough, as there are many caveats in how to interpret the IRS's data. But think of it as a starting
serious exploration of human migration.

For instance, recall the ranking by income-per-return (IPR) in Exercise 7. In the final rankings, the counties in the top 10 are relatively rich (recall
are in the low hundreds, whereas there are over 3,000 counties). However, they are not "too rich." Perhaps it is easier to move to regions where
higher than where one comes from, but not so high that one cannot afford to move at all. A deeper analysis of income and mobility would certa

The phenomenon of migration has even deeper implications. Indeed, this problem was inspired by this article from the New York Times on hum
and climate change (https://www.nytimes.com/interactive/2020/07/23/magazine/climate-migration.html), which you might enjoy reading now th
is over.

Data sources for this notebook:

- US Census Bureau: Population data (https://www.census.gov/data/datasets/time-series/demo/popest/2010s-counties-total.html)
- US Internal Revenue Service Tax migration data (https://www.irs.gov/statistics/soi-tax-stats-migration-data)

< Previous

Next Up: Practice Problems for Final Exam
1 min + 1 activity >