hw07

July 29, 2022

```
[ ]: # Initialize Otter
import otter
grader = otter.Notebook("hw07.ipynb")
```

1 Homework 7: IMDb and U.S. Presidential Elections

1.1 SQL + PCA

1.2 Due Date: Monday, August 1, 11:59 PM PDT

Collaboration Policy

Data science is a collaborative activity. While you may talk with others about the homework, we ask that you **write your solutions individually**. If you do discuss the assignments with others please **include their names** below.

Collaborators: *list collaborators here*

1.3 Introduction

This homework has two disjoint parts:

Part 1: We will use SQL to dive deep into the Internet Movie Database (IMDb) and answer different questions involving movies, actors, and movie ratings. Click here to jump to Part 1.

Part 2: We will use Principal Component Analysis to understand a high-dimensional dataset: U.S. Presidential Elections by State. Click here to jump to Part 2.

1.4 Grading

Grading is broken down into autograded answers and free response. For autograded answers, the results of your code are compared to provided and/or hidden tests. For free response, readers will evaluate how well you answered the question and/or fulfilled the requirements of the question.

Question	Points
Q1a	2
Q1b	2
Q2	3
Q3	3
Q4	4

Question	Points
Q5	4
Q6a	1
Q6b	1
Q6c	1
Q6d	1
Q6e	2
Q7a	2
Q7b	2
Q8a	1
Q8b	2
Q8c	2
Total	33

[1]:
```
# Run this cell to set up your notebook

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import plotly.express as px
import seaborn as sns
import sqlalchemy
from pathlib import Path

plt.style.use('fivethirtyeight') # Use plt.style.available to see more styles
sns.set()
sns.set_context("talk")
np.set_printoptions(threshold=5) # avoid printing out big matrices
%matplotlib inline
%load_ext sql
```

Part 1: The IMDB (mini) Dataset

(Click here to jump back to the top of this notebook.)

We will explore a miniature version of the IMDb Dataset. This is the same dataset that we used for this week's lab. The remainder of this overview section is copied from this week's lab.

Let's load in the database in two ways (using both Python and cell magic) so that we can flexibly explore the SQL database.

A few reminders: * **Only SQL code written with `pd.read_sql` will be graded.** You should feel free to create `%%sql` cells **after** your Python answer + autograder cells to reduce debugging headaches, but you will still need to copy over any SQL to the Python answer cells. **Do not** add new cells betwen the question and the grading cells; it will cause errors when we run the autograder, and it will sometimes cause an error in generating the PDF file.

- **Caution: Be careful with large SQL queries!!** You may need to reboot your Jupyter

Hub instance if it stops responding. **Use the LIMIT keyword** to avoid printing out 100k-sized tables (but remember to remove it).

[2]:
```
# run this cell and the next one
engine = sqlalchemy.create_engine("sqlite:///data/imdbmini.db")
connection = engine.connect()
```

[3]:
```
%sql sqlite:///data/imdbmini.db
```

Let's take a look at the table schemas:

[4]:
```
%%sql
-- just run this cell --
SELECT * FROM sqlite_master WHERE type='table';
```

```
 * sqlite:///data/imdbmini.db
Done.
```

[4]:
```
[('table', 'Title', 'Title', 2, 'CREATE TABLE "Title" (\n"tconst" INTEGER,\n
"titleType" TEXT,\n  "primaryTitle" TEXT,\n  "originalTitle" TEXT,\n  "isAdult"
TEXT,\n  "startYear" TEXT,\n  "endYear" TEXT,\n  "runtimeMinutes" TEXT,\n
"genres" TEXT\n)'),
 ('table', 'Name', 'Name', 12, 'CREATE TABLE "Name" (\n"nconst" INTEGER,\n
"primaryName" TEXT,\n  "birthYear" TEXT,\n  "deathYear" TEXT,\n
"primaryProfession" TEXT\n)'),
 ('table', 'Role', 'Role', 70, 'CREATE TABLE "Role" (\ntconst INTEGER,\nordering
TEXT,\nnconst INTEGER,\ncategory TEXT,\njob TEXT,\ncharacters TEXT\n)'),
 ('table', 'Rating', 'Rating', 41, 'CREATE TABLE "Rating" (\ntconst
INTEGER,\naverageRating TEXT,\nnumVotes TEXT\n)')]
```

From running the above cell, we see the database has 4 tables: `Name`, `Role`, `Rating`, and `Title`.

[Click to Expand] See descriptions of each table's schema.

`Name` – Contains the following information for names of people.

- nconst (text) - alphanumeric unique identifier of the name/person
- primaryName (text)– name by which the person is most often credited
- birthYear (integer) – in YYYY format
- deathYear (integer) – in YYYY format

`Role` – Contains the principal cast/crew for titles.

- tconst (text) - alphanumeric unique identifier of the title
- ordering (integer) – a number to uniquely identify rows for a given tconst
- nconst (text) - alphanumeric unique identifier of the name/person
- category (text) - the category of job that person was in
- characters (text) - the name of the character played if applicable, else '\N'

`Rating` – Contains the IMDb rating and votes information for titles.

- tconst (integer) - alphanumeric unique identifier of the title

- averageRating (text) – weighted average of all the individual user ratings
- numVotes (text) - number of votes (i.e., ratings) the title has received

`Title` - Contains the following information for titles.

- tconst (text) - alphanumeric unique identifier of the title
- titleType (text) - the type/format of the title
- primaryTitle (text) - the more popular title / the title used by the filmmakers on promotional materials at the point of release
- isAdult (text) - 0: non-adult title; 1: adult title
- startYear (text) – represents the release year of a title.
- runtimeMinutes (integer) – primary runtime of the title, in minutes

From the above descriptions, we can conclude the following: * `Name.nconst` and `Title.tconst` are primary keys of the `Name` and `Title` tables, respectively. * `Role.nconst` and `Role.tconst` are **foreign keys** that point to `Name.nconst` and `Title.tconst`, respectively.

1.5 Question 1

1.5.1 Question 1a

How far back does our data go? Does it only include recent data, or do we have information about older movies and movie stars as well?

List the **10 oldest `movie` titles** by `startYear` and then `primaryTitle` both in **ascending** order. Do not include films where the `startYear` is `NULL`. The output should contain the `startYear`, `primaryTitle`, and `titleType`.

Remember, you can create a `%%sql` cell **after** the grader cell as scratch work. Just be sure to copy the query back into the Python cell to run the autograder.

[5]:
```
query_q1a = """
...       # replace this with
...;      # your multi-line solution
"""

# BEGIN SOLUTION NO PROMPT
query_q1a = """
SELECT
    startYear, primaryTitle, titleType
FROM Title
WHERE startYear IS NOT NULL
AND titleType = "movie"
ORDER BY startYear, primaryTitle
LIMIT 10;
"""
# END SOLUTION

res_q1a = pd.read_sql(query_q1a, engine)
res_q1a
```

[5]:

	startYear	primaryTitle	titleType
0	1915	The Birth of a Nation	movie
1	1920	The Cabinet of Dr. Caligari	movie
2	1921	The Kid	movie
3	1922	Nosferatu	movie
4	1924	Sherlock Jr.	movie
5	1925	Battleship Potemkin	movie
6	1925	The Gold Rush	movie
7	1926	The General	movie
8	1927	Metropolis	movie
9	1927	Sunrise	movie

[ ]:
```
grader.check("q1a")
```

1.5.2 Question 1b

Next, let's calculate the distribution of films by year. Write a query that returns the **total** `movie` titles for each `startYear` in the `Title` table as `total`. Keep in mind that some entries may not have a `startYear` listed – you should filter those out. Order your final results by the `startYear` in **ascending** order.

The first few records of the table should look like the following (but you should compute the entire table).

	startYear	total
0	1902	1
1	1915	1
2	1920	1
3	1921	1
4	1922	1
...	...	...

[10]:
```
query_q1b = """
...       # replace this with
...;      # your multi-line solution
"""

# BEGIN SOLUTION NO PROMPT
query_q1b = """
SELECT
    startYear,
    COUNT(*) AS total
FROM Title
WHERE startYear IS NOT NULL
AND titleType = "movie"
GROUP BY startYear
```

```
ORDER BY startYear; -- ascending for plot --
"""
# END SOLUTION

res_q1b = pd.read_sql(query_q1b, engine)
res_q1b
```

[10]:

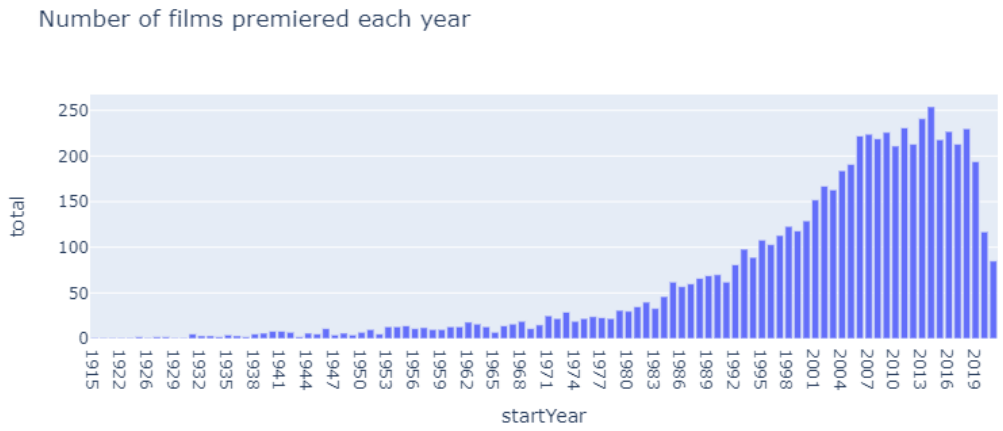
	startYear	total
0	1915	1
1	1920	1
2	1921	1
3	1922	1
4	1924	1
..	…	…
97	2017	213
98	2018	230
99	2019	194
100	2020	117
101	2021	85

[102 rows x 2 columns]

[ ]:
```
grader.check("q1b")
```

The following should generate an interesting plot of the number of films that premiered each year. Notice there is a dip between the 1920s and late 1940s. Why might that be? *This question is rhetorical; you do not need to write your answer anywhere.*

[14]:
```
# just run this cell
px.bar(res_q1b, x="startYear", y="total",
       title="Number of films premiered each year")
```



1.6 Question 2

Who are the **top 10 most prolific movie actors**?

Define the term "movie actor" is defined as anyone with an `actor` or `actress` job category role in a `movie` title.

Your SQL query should output exactly two fields named `name` (the movie actor name) and `total` (the number of movies the movie actor appears in). Order the records by `total` in descending order, and break ties by ordering by `name` in ascending order.

Your result should look something like the following, but without `????`:

	name	total
0	????	64
1	????	54
2	????	53
3	????	49
4	????	46
5	????	43
6	????	41
7	????	40
8	????	40
9	????	39

Some hints:

- ***The query should take < 2 minutes to run.***
- Google the top of the list and see if it makes sense.
- If you want to include a non-aggregate field in the `SELECT` clause, it must also be included in the `GROUP BY` clause.

[15]:
```
query_q2 = """
...       # replace this with
...;      # your multi-line solution
"""

# BEGIN SOLUTION NO PROMPT
query_q2 = """
SELECT
    Name.primaryName AS name,
    COUNT(*) AS total
FROM Name
JOIN Role ON Name.nconst = Role.nconst
JOIN Title ON Role.tconst = Title.tconst
WHERE Title.titleType = 'movie' AND Role.category IN ('actor', 'actress')
GROUP BY Name.primaryName
ORDER BY total DESC, name ASC
LIMIT 10;
```

```
"""

# -- Alternate solution --
# SELECT
#     primaryName AS name,
#     COUNT(*) AS total
# FROM Name AS n, Role AS ro, Title AS t
# WHERE n.nconst = ro.nconst AND ro.tconst = t.tconst
#     AND ro.category == 'actor' OR ro.category == 'actress'
#     AND t.titleType = 'movie'
# GROUP BY n.nconst, name
# ORDER BY total DESC, name ASC
# LIMIT 10;


# END SOLUTION

res_q2 = pd.read_sql(query_q2, engine)
res_q2
```

[15]:

	name	total
0	Robert De Niro	64
1	Samuel L. Jackson	54
2	Nicolas Cage	53
3	Bruce Willis	49
4	Tom Hanks	46
5	Johnny Depp	43
6	Mark Wahlberg	41
7	Liam Neeson	40
8	Morgan Freeman	40
9	Adam Sandler	39

[ ]:
```
grader.check("q2")
```

[20]:
```
sorted(list(res_q2['name']))
```

[20]:
```
['Adam Sandler',
 'Bruce Willis',
 'Johnny Depp',
 'Liam Neeson',
 'Mark Wahlberg',
 'Morgan Freeman',
 'Nicolas Cage',
 'Robert De Niro',
 'Samuel L. Jackson',
 'Tom Hanks']
```

1.7 Question 3: The `CASE` Keyword

The `Rating` table has the `numVotes` and the `averageRating` for each title. Which `movie` titles were **"big hits"**, defined as a movie with over 100,000 votes? Construct the following table:

	isBigHit	total
0	no	????
1	yes	????

Where `????` is replaced with the correct values. The row with `no` should have the count for how many movies **are not** big hits, and the row with `yes` should have the count of how many movies **are** big hits.

- Rating.numVotes currently consists of string objects, use `CAST(Rating.numVotes AS int)` to convert them to integer.
- You will need to use some type of `JOIN`.
- You may also consider using a `CASE WHEN ... IS ... THEN 'yes' ... ELSE ... END` statement. `CASE` statements are the SQL-equivalent of Python `if... elif... else` statements. To read up on `CASE`, take a look at the following links:
  - https://mode.com/sql-tutorial/sql-case/
  - https://www.w3schools.com/sql/sql_ref_case.asp

[21]:
```
query_q3 = """
...       # replace this with
...;      # your multi-line solution
"""

# BEGIN SOLUTION NO PROMPT
query_q3 = """
SELECT
    CASE
        WHEN CAST(Rating.numVotes AS int) > 100000 THEN "yes"
        ELSE "no"
    END AS isBigHit,
    COUNT(*) as total
FROM Title
JOIN Rating
    ON Rating.tconst = Title.tconst
WHERE Title.titleType == 'movie'
GROUP BY isBigHit
;
"""
# END SOLUTION

res_q3 = pd.read_sql(query_q3, engine)
res_q3
```

[21]:

	isBigHit	total
0	no	4318
1	yes	2041

[ ]:
```
grader.check("q3")
```

1.8 Question 4

How does film length relate to ratings? To answer this question we want to bin `movie` titles by length and compute the average of the average ratings within each length bin. We will group movies by 10-minute increments – that is, one bin for movies [0, 10) minutes long, another for [10, 20) minutes, another for [20, 30) minutes, and so on. Use the following code snippet to help construct 10-minute bins:

```
ROUND(runtimeMinutes / 10.0 + 0.5) * 10 AS runtimeBin
```

Construct a table containing the **`runtimeBin`**, the **average** of the **average ratings** (as `averageRating`), the **average number of votes** (as `averageNumVotes`), and the number of `titles` in that **runtimeBin** (as `total`). Only include movies with **at least 10000 votes**. Order the final results by the value of `runtimeBin`.

[26]:
```
query_q4 = """
...       # replace this with
...;      # your multi-line solution
"""

# BEGIN SOLUTION NO PROMPT
query_q4 = """
    SELECT ROUND(runtimeMinutes / 10.0 + 0.5) * 10 AS runtimeBin,
        AVG(averageRating) AS averageRating,
        AVG(numVotes) AS averageNumVotes,
        COUNT(*) AS total
    FROM Title
    JOIN Rating
         ON Title.tconst = Rating.tconst
    WHERE Rating.numVotes >= 10000
        AND Title.titleType = 'movie'
    GROUP BY runtimeBin
    ORDER BY runtimeBin;
"""
# END SOLUTION

res_q4 = pd.read_sql(query_q4, engine)
res_q4.head()
```

[26]:

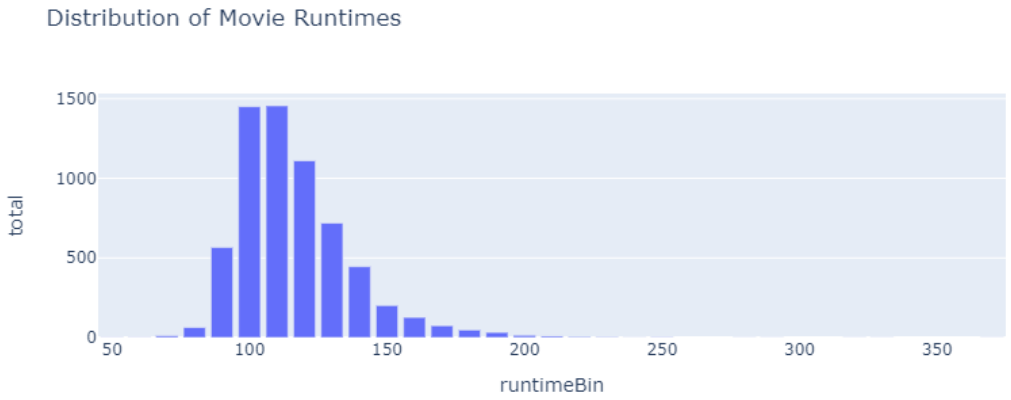
	runtimeBin	averageRating	averageNumVotes	total
0	50.0	7.850000	42535.000000	2
1	60.0	6.400000	30668.500000	2

```
2        70.0        7.600000      59822.000000      13
3        80.0        6.860937      67896.187500      64
4        90.0        6.283951      76907.608466     567
```

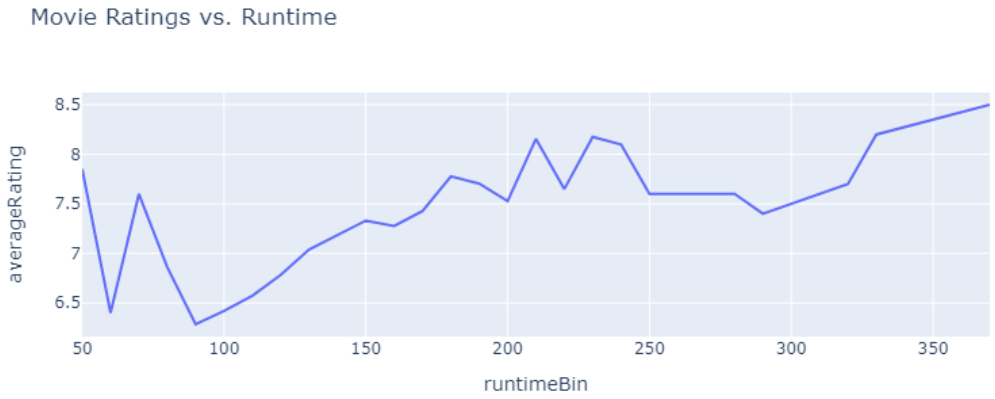
[ ]:
```
grader.check("q4")
```

If your SQL query is correct you should get some interesting plots below. This might explain why directors keep going a particular direction with film lengths.

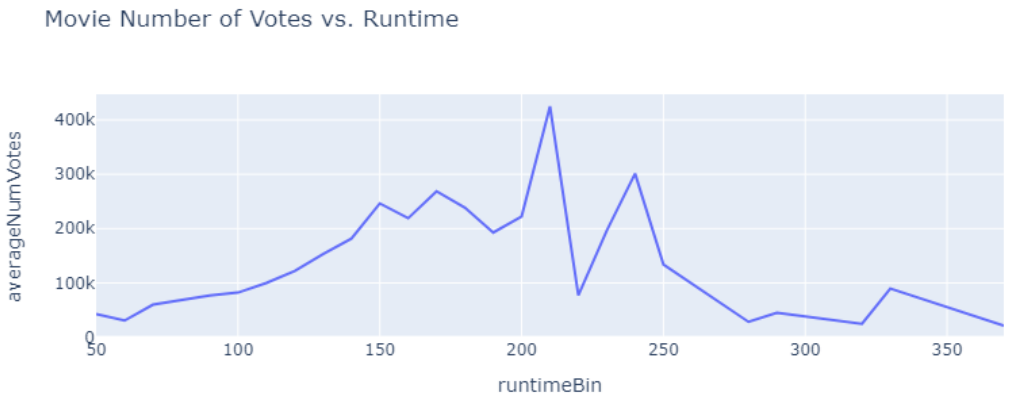
[34]:
```
# just run this cell
px.bar(res_q4, x="runtimeBin", y="total",
       title="Distribution of Movie Runtimes")
```



[35]:
```
# just run this cell
px.line(res_q4, x="runtimeBin", y="averageRating",
        title="Movie Ratings vs. Runtime")
```



[36]:
```
px.line(res_q4, x="runtimeBin", y="averageNumVotes",
        title="Movie Number of Votes vs. Runtime")
```



1.9 Question 5

Which **movie actors** have the highest average ratings across all the **movies** in which they star? Again, define "movie actor" as anyone with an `actor` or `actress` job category role in a `movie` title.

Construct a table consisting of the **movie actor's name** (as `name`) and their **average actor rating** (as `actorRating`) computed by rescaling ratings for movies in which they had a role:

$$\text{actorRating} = \frac{\sum_m \text{averageRating}[m] * \text{numVotes}[m]}{\sum_m \text{numVotes}[m]}$$

Some notes: * Note that if an actor/actress has multiple `role` listings for a film then that film will have a bigger impact in the overall average (this is desired). * ***The query should take < 3 minutes to run.*** * Only consider ratings where there are **at least 1000** votes and only consider movie actors that have **at least 20 rated performances**. Present the movie actors with the **top 10** `actorRating` in descending order and break ties alphabetically using the movie actor's name.

The results should look something like this but without the `????`, and with higher rating precision.

	name	actorRating
0	????	8.4413...
1	????	8.2473...
2	????	8.1383...
3	????	8.1339...
4	????	8.0349...
5	????	7.9898...

	name	actorRating
6	????	7.9464...
7	????	7.9330...
8	????	7.9261...
9	????	7.8668...

[37]:
```
query_q5 = """
...       # replace this with
...;      # your multi-line solution
"""

# BEGIN SOLUTION NO PROMPT
query_q5 = """
SELECT primaryName as name,
    SUM(Rating.averageRating * Rating.numVotes) / SUM(Rating.numVotes) AS␣
 ↪actorRating
FROM Name
JOIN Role
    ON Name.nconst == Role.nconst
JOIN Title
    ON Role.tconst == Title.tconst
JOIN Rating
    ON Rating.tconst == Title.tconst
WHERE Role.category IN ('actor', 'actress')
    AND Title.titleType = 'movie'
    AND Rating.numVotes >= 1000
GROUP BY Name.nconst, name
HAVING COUNT(*) >= 20
ORDER BY actorRating DESC, name ASC
LIMIT 10;
"""
# END SOLUTION

res_q5 = pd.read_sql(query_q5, engine)
res_q5
```

[37]:

	name	actorRating
0	Diane Keaton	8.441302
1	Tim Robbins	8.247318
2	Al Pacino	8.138361
3	Michael Caine	8.133915
4	Leonardo DiCaprio	8.034961
5	Christian Bale	7.989825
6	Robert Duvall	7.946483
7	Jack Nicholson	7.933034
8	Kevin Spacey	7.926158

```
9     Clint Eastwood     7.866839
```

[ ]:
```
grader.check("q5")
```

Part 2: U.S. Presidential Elections By State

(Click here to jump back to the top of this notebook.)

The second part of this homework focuses on a new dataset. If you haven't worked with Principal Component Analysis before, we highly encourage you to take a look at the PCA lab first. PCA really shines on data where you have reason to believe that the data is relatively low in rank.

We'll look at how states voted in presidential elections between 1972 and 2016. **Our ultimate goal in this part is to show how 2D PCA scatterplots can allow us to identify clusters in a high dimensional dataset.** For this example, that means finding groups of states that vote similarly by plotting their 1st and 2nd principal components.

1.10 Question 6

We explore a dataset on U.S. Presidential Elections by State since 1789, as taken from Wikipedia.

[42]:
```
df = pd.read_csv("data/presidential_elections.csv")
df.head(5)
```

[42]:

	State	1789	1792	1796	1800 †	Unnamed: 5	1804	1808	1812	1816	…	1992	\
0	Alabama	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	…	R	
1	Alaska	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	…	R	
2	Arizona	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	…	R	
3	Arkansas	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	…	D	
4	California	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	…	D	

	1996	2000 ‡	Unnamed: 60	2004	2008	2012	2016 ‡	2020	State.1
0	R	R	NaN	R	R	R	R	R	Alabama
1	R	R	NaN	R	R	R	R	R	Alaska
2	D	R	NaN	R	R	R	R	D	Arizona
3	D	R	NaN	R	R	R	R	R	Arkansas
4	D	D	NaN	D	D	D	D	D	California

[5 rows x 67 columns]

The data in this table is pretty messy (missing records, inconsistent field naming, etc.), so let's create a clean version.

Running the cell below will produce a clean table, which contains exactly 51 rows (corresponding to the 50 states plus Washington DC) and 13 columns (one for each of the election years from 1972 to 2020). The index of this dataframe is the state name.

[43]:
```
# just run this cell
df_clean = (
        df.iloc[:, -15:]
```

```
        .drop(['Unnamed: 60'], axis = 1)
        .rename(columns = {"2000 ‡": "2000", "2016 ‡": "2016", "State.1":␣
 ↪"State"})
        .drop([51])
        .set_index("State")
)
df_clean
```

[43]:

State	1972	1976	1980	1984	1988	1992	1996	2000	2004	2008	2012	2016
Alabama	R	D	R	R	R	R	R	R	R	R	R	R
Alaska	R	R	R	R	R	R	R	R	R	R	R	R
Arizona	R	R	R	R	R	R	D	R	R	R	R	R
Arkansas	R	D	R	R	R	D	D	R	R	R	R	R
California	R	R	R	R	R	D	D	D	D	D	D	D
Colorado	R	R	R	R	R	D	R	R	R	D	D	D
Connecticut	R	R	R	R	R	D	D	D	D	D	D	D
Delaware	R	D	R	R	R	D	D	D	D	D	D	D
D.C.	D	D	D	D	D	D	D	D	D	D	D	D
Florida	R	D	R	R	R	R	D	R	R	D	D	R
Georgia	R	D	D	R	R	D	R	R	R	R	R	R
Hawaii	R	D	D	R	D	D	D	D	D	D	D	D
Idaho	R	R	R	R	R	R	R	R	R	R	R	R
Illinois	R	R	R	R	R	D	D	D	D	D	D	D
Indiana	R	R	R	R	R	R	R	R	R	D	R	R
Iowa	R	R	R	R	D	D	D	D	R	D	D	R
Kansas	R	R	R	R	R	R	R	R	R	R	R	R
Kentucky	R	D	R	R	R	D	D	R	R	R	R	R
Louisiana	R	D	R	R	R	D	D	R	R	R	R	R
Maine	R	R	R	R	R	D	D	D	D	D	D	D
Maryland	R	D	D	R	R	D	D	D	D	D	D	D
Massachusetts	D	D	R	R	D	D	D	D	D	D	D	D
Michigan	R	R	R	R	R	D	D	D	D	D	D	R
Minnesota	R	D	D	D	D	D	D	D	D	D	D	D
Mississippi	R	D	R	R	R	R	R	R	R	R	R	R
Missouri	R	D	R	R	R	D	D	R	R	R	R	R
Montana	R	R	R	R	R	D	R	R	R	R	R	R
Nebraska	R	R	R	R	R	R	R	R	R	R	R	R
Nevada	R	R	R	R	R	D	D	R	R	D	D	D
New Hampshire	R	R	R	R	R	D	D	R	D	D	D	D
New Jersey	R	R	R	R	R	D	D	D	D	D	D	D
New Mexico	R	R	R	R	R	D	D	D	R	D	D	D
New York	R	D	R	R	D	D	D	D	D	D	D	D
North Carolina	R	D	R	R	R	R	R	R	R	D	R	R
North Dakota	R	R	R	R	R	R	R	R	R	R	R	R
Ohio	R	D	R	R	R	D	D	R	R	D	D	R
Oklahoma	R	R	R	R	R	R	R	R	R	R	R	R

Oregon	R	R	R	R	D	D	D	D	D	D	D	D
Pennsylvania	R	D	R	R	R	D	D	D	D	D	D	R
Rhode Island	R	D	D	R	D	D	D	D	D	D	D	D
South Carolina	R	D	R	R	R	R	R	R	R	R	R	R
South Dakota	R	R	R	R	R	R	R	R	R	R	R	R
Tennessee	R	D	R	R	R	D	D	R	R	R	R	R
Texas	R	D	R	R	R	R	R	R	R	R	R	R
Utah	R	R	R	R	R	R	R	R	R	R	R	R
Vermont	R	R	R	R	R	D	D	D	D	D	D	D
Virginia	R	R	R	R	R	R	R	R	R	D	D	D
Washington	R	R	R	R	D	D	D	D	D	D	D	D
West Virginia	R	D	D	R	D	D	D	R	R	R	R	R
Wisconsin	R	D	R	R	D	D	D	D	D	D	D	R
Wyoming	R	R	R	R	R	R	R	R	R	R	R	R

	2020
State	
Alabama	R
Alaska	R
Arizona	D
Arkansas	R
California	D
Colorado	D
Connecticut	D
Delaware	D
D.C.	D
Florida	R
Georgia	D
Hawaii	D
Idaho	R
Illinois	D
Indiana	R
Iowa	R
Kansas	R
Kentucky	R
Louisiana	R
Maine	D
Maryland	D
Massachusetts	D
Michigan	D
Minnesota	D
Mississippi	R
Missouri	R
Montana	R
Nebraska	R
Nevada	D
New Hampshire	D

```
New Jersey        D
New Mexico        D
New York          D
North Carolina    R
North Dakota      R
Ohio              R
Oklahoma          R
Oregon            D
Pennsylvania      D
Rhode Island      D
South Carolina    R
South Dakota      R
Tennessee         R
Texas             R
Utah              R
Vermont           D
Virginia          D
Washington        D
West Virginia     R
Wisconsin         D
Wyoming           R
```

Side note: We produced the data cleaning function chain above by inspecting the CSV file. In your personal projects, you may be tempted to use Excel or Google Sheets (What You See Is What You Get, or WYSIWYG) to clean data. While sometimes more convenient, the downside of this is that you have no record of what you did—and if you have to redownload the data, you have to redo the manual data cleaning process.

1.10.1 Question 6a

What does each row in `df_clean` represent?

Type your answer here, replacing this text.

SOLUTION: Each row represents the voting record of a particular state for each presidential election between 1972 and 2020.

1.10.2 Question 6b

To perform PCA, we need to convert our data into numerical values. Create a `df_numerical` dataframe that replaces all of the "D" characters in `df_clean` with the number 0, and all of the "R" characters with the number 1.

Hint: Use `df.replace` (documentation).

[44]:
```
df_numerical = ...
# BEGIN SOLUTION
df_numerical = df_clean.replace({'D': 0, 'R': 1})
# END SOLUTION
```

[ ]:
```
grader.check("q6b")
```

1.10.3 Question 6c

Now **standardize the data**: Center the data so that each column's mean is 0, and scale the data so that each column's variance is 1. Store your result in `df_standardized`.

[47]:
```
df_standardized = ...
# BEGIN SOLUTION
df_centered = df_numerical - np.mean(df_numerical, axis = 0)
df_standardized = df_centered / np.std(df_centered, axis = 0)
# END SOLUTION
```

[ ]:
```
grader.check("q6c")
```

We now have our data in a nice and tidy centered and scaled format. Phew! We are now ready to do PCA.

1.10.4 Question 6d: SVD

In the following cell, compute the SVD of `df_standardized`:

$$\texttt{df\_standardized} = U\Sigma V^T$$

Store the U, Σ, and V^T matrices in `u`, `s`, and `vt` respectively. This is one line of simple code (exactly like what we saw in lecture and what you did in lab) using the `np.linalg.svd` function with the `full_matrices` argument set to `False`.

[50]:
```
u, s, vt = np.linalg.svd(df_standardized, full_matrices=False) # SOLUTION
u, s, vt
```

[50]:
```
(array([[-0.16256283,  0.07765483,  0.00122229, …, -0.0134597 ,
          0.08820748,  0.08217091],
        [-0.16706061, -0.0162679 , -0.12628238, …, -0.04426333,
         -0.01355807,  0.07427217],
        [-0.09532553, -0.04755254, -0.03905914, …,  0.45011822,
          0.13814776, -0.20684122],
        …,
        [-0.03836168,  0.28942847,  0.23831932, …, -0.08179345,
         -0.12580385, -0.02809131],
        [ 0.13046354,  0.02585922,  0.16202611, …,  0.42873974,
         -0.07025696,  0.0487704 ],
        [-0.16706061, -0.0162679 , -0.12628238, …, -0.04426333,
         -0.01355807,  0.07427217]]),
 array([18.07691752, 10.25583887,  7.64572873, …,  2.80257708,
         2.26892781,  1.70403666]),
 array([[-0.136002  , -0.04058259, -0.15155429, …, -0.35596474,
         -0.32690835, -0.33987784],
```

```
[-0.3492229 , -0.48079419, -0.4675855 , …,  0.18624495,
  0.15138213,  0.15659938],
[ 0.41610347, -0.48658899, -0.02929042, …,  0.03473676,
  0.22737918,  0.113887  ],
…,
[ 0.04435877, -0.04309004,  0.18776949, …, -0.04299605,
  0.49573543, -0.77479348],
[ 0.01144521, -0.11524943,  0.04285712, …, -0.05855334,
 -0.11179211, -0.16276473],
[-0.01003386, -0.00671822, -0.03591355, …, -0.80093148,
  0.07451528,  0.13458479]]))
```

[ ]:
```
grader.check("q6d")
```

1.10.5 Question 6e: Get Principal Components

Using your results from the previous part, create a new `first_2_pcs` **dataframe** (documentation) that contains exactly the first two columns of the principal components matrix. The first column should be labeled `pc1` and the second column should be labeled `pc2`. Store your result in `first_2_pcs`, and make sure to set the index to be the default numerical range index (i.e. 0, 1, 2, …).

[55]:
```
first_2_pcs = pd.DataFrame((u*s)[:, 0:2], columns = ["pc1", "pc2"]) # SOLUTION
first_2_pcs.head()
```

[55]:

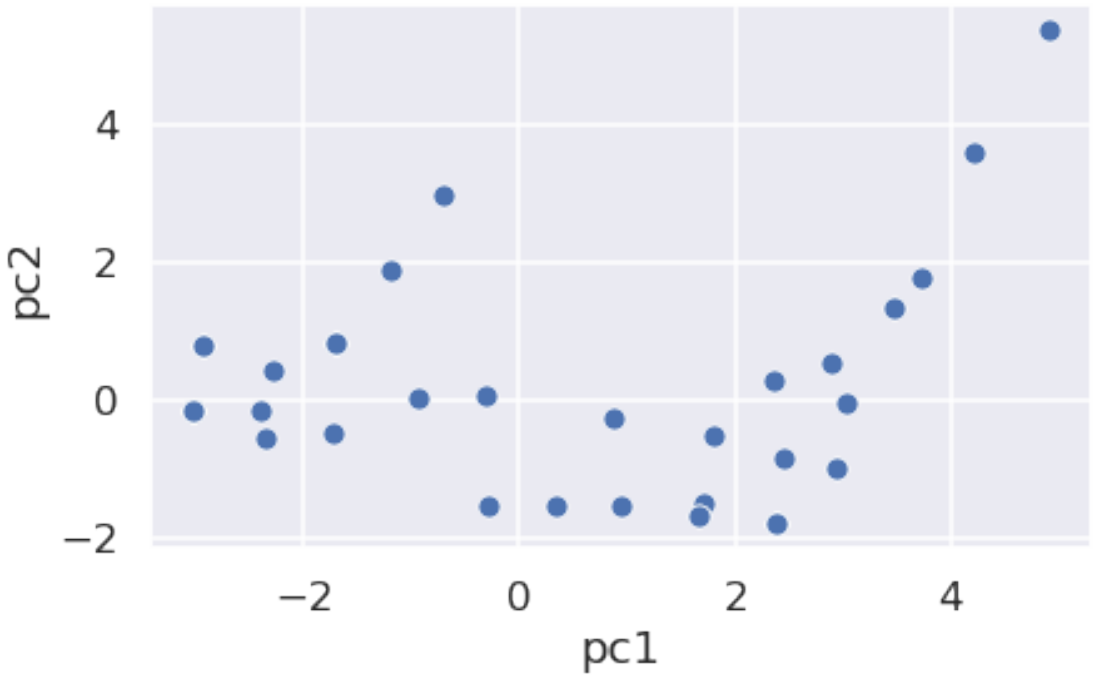
	pc1	pc2
0	-2.938635	0.796415
1	-3.019941	-0.166841
2	-1.723192	-0.487691
3	-1.698397	0.807836
4	2.376794	-1.807335

[ ]:
```
grader.check("q6e")
```

1.11 Question 7

The cell below plots the 1st and 2nd principal components of our 50 states + Washington DC.

[62]:
```
# just run this cell
sns.scatterplot(data = first_2_pcs, x = "pc1", y = "pc2");
```



Unfortunately, we have two problems:

1. There is a lot of overplotting, with only 28 distinct dots (out of 104 points). This means that at least some states voted exactly alike in these elections.
2. We don't know which state is which because the points are unlabeled.

1.11.1 Question 7a: Jitter

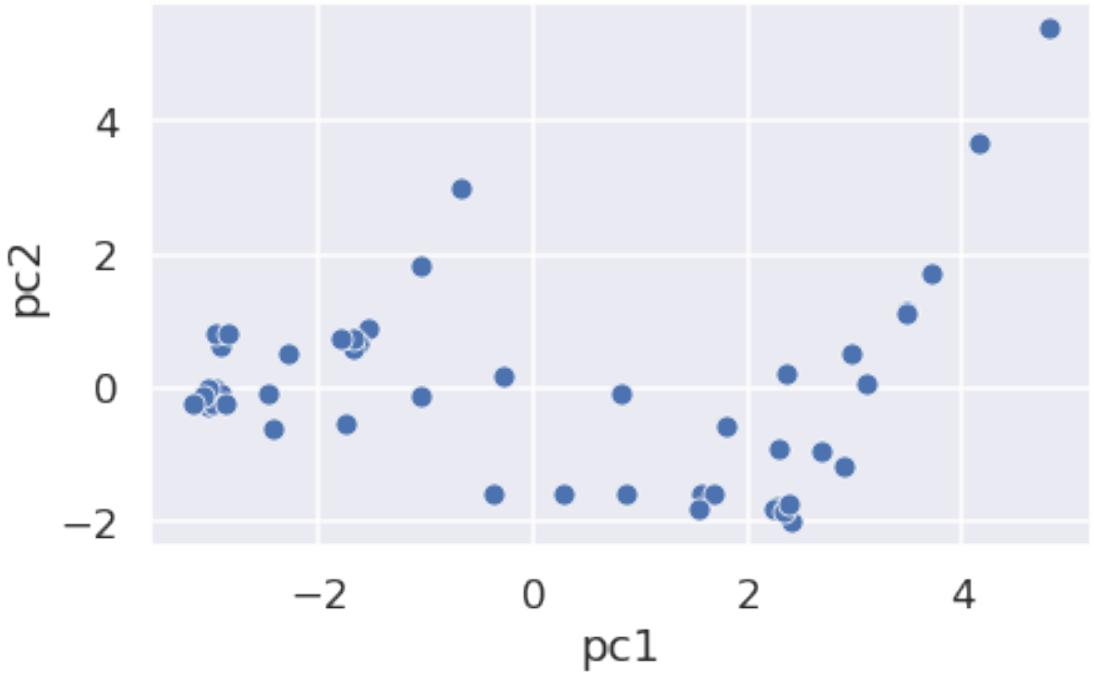
Let's start by addressing problem 1.

In the cell below, create a new dataframe `first_2_pcs_jittered` with a small amount of random noise added to each principal component. In this same cell, create a scatterplot.

To reduce overplotting, we **jitter** the first two principal components: * Add a small amount of random, unbiased Gaussian noise to each value using `np.random.normal` (documentation) with mean 0 and standard deviation less than 1. * Don't get caught up on the exact details of your noise generation; it's fine as long as your plot looks roughly the same as the original scatterplot, but without overplotting. * The amount of noise you add *should not significantly affect* the appearance of the plot; it should simply serve to separate overlapping observations.

[63]:
```
np.random.seed(42)
# first, jitter the data
first_2_pcs_jittered = first_2_pcs + np.random.normal(0, 0.1, first_2_pcs.
 ↪shape) # SOLUTION

# then, create a scatter plot
```

```
# BEGIN SOLUTION
sns.scatterplot(data = first_2_pcs_jittered, x = "pc1", y = "pc2");
# END SOLUTION
```



[64]:
```
first_2_pcs_jittered
```

[64]:

	pc1	pc2
0	-2.888963	0.782589
1	-2.955172	-0.014538
2	-1.746607	-0.511105
3	-1.540476	0.884579
4	2.329846	-1.753079
5	0.281022	-1.572768
6	2.400990	-1.998663
7	2.285608	-0.900307
8	4.801990	5.404664
9	-1.032772	-0.109922
10	-1.048393	1.838392
11	3.469789	1.173943
12	-3.074379	-0.155749
13	2.261694	-1.769765
14	-2.413367	-0.580390
15	0.820609	-0.066840
16	-3.021291	-0.272612

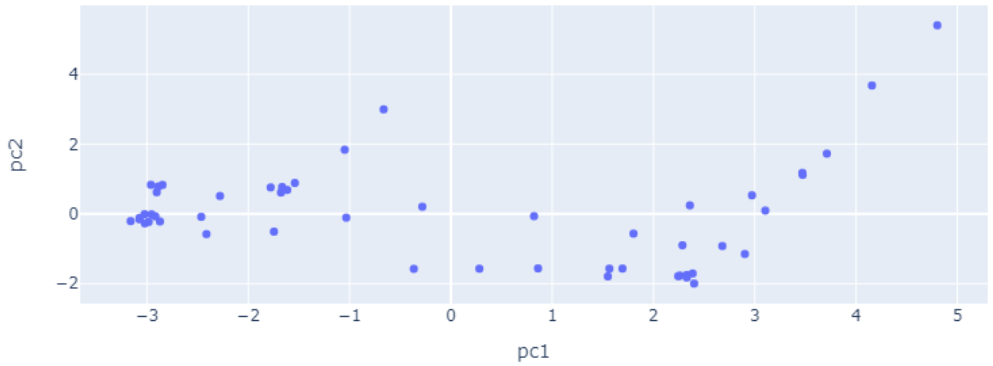
```
17 -1.616143  0.685752
18 -1.677511  0.611869
19  2.243975 -1.787649
20  2.972362  0.531859
21  3.711707  1.726625
22  1.564700 -1.571728
23  4.156557  3.679832
24 -2.904273  0.620111
25 -1.665989  0.769328
26 -2.464258 -0.086662
27 -2.916841 -0.073713
28  0.860305 -1.564708
29  1.693760 -1.568953
30  2.328876 -1.825901
31  1.549753 -1.794234
32  3.103873  0.093240
33 -2.279198  0.512389
34 -2.983777 -0.231353
35 -0.282455  0.204124
36 -3.023523 -0.010377
37  2.679340 -0.923450
38  1.802563 -0.566388
39  3.472212  1.117660
40 -2.960602  0.832127
41 -2.872151 -0.218668
42 -1.779247  0.757660
43 -2.847095  0.829291
44 -3.072917 -0.115514
45  2.386501 -1.710470
46 -0.366216 -1.577973
47  2.902103 -1.151991
48 -0.663849  2.994437
49  2.358890  0.241749
50 -3.161478 -0.208905
```

1.11.2 Question 7b

To address Problem 2, we can turn to Plotly. The below cell uses these Plotly guides (example 1, example 2) to create a scatter plot of the jittered .

[65]:
```
# just run this cell
import plotly.express as px

# get the state names from the standardized dataframe's index
first_2_pcs_jittered['state'] = df_standardized.index

# show state names on hover
```

```
fig = px.scatter(first_2_pcs_jittered, x="pc1", y="pc2",
                 hover_data={"state": True});

fig.show();
```



Analyze the above plot. In the below cell, address the following two points: 1. Give an example of a cluster of states that vote a similar way. Does the composition of this cluster surprise you? If you're not familiar with U.S. politics, it's fine to just say "No, I'm not surprised because I don't know anything about U.S. politics." 1. Include anything interesting that you observe. You will get credit for this as long as you write something reasonable that you can takeaway from the plot.

Type your answer here, replacing this text.

SOLUTION: 1. Any response is fine here. 2. Any observation with some thought and effort should receive full credit.

1.12 Question 8

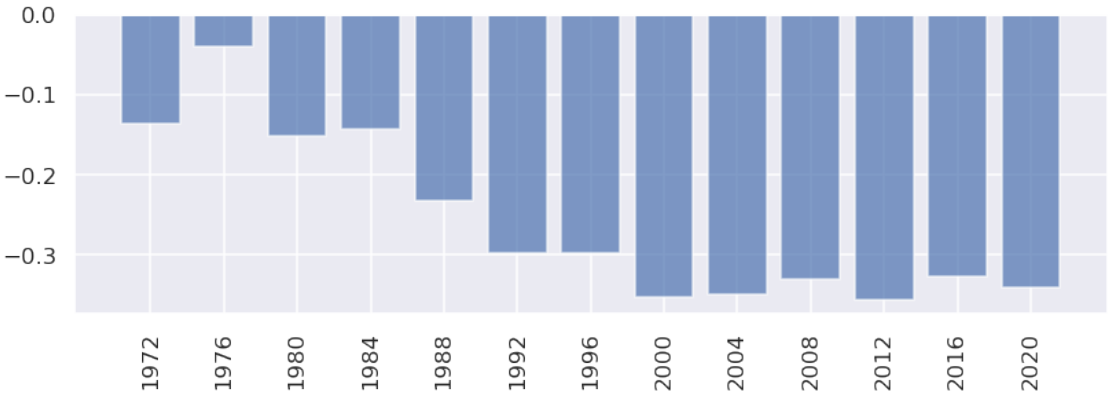
We can also look at the contributions of each year's elections results on the values for our principal components.

Below, we define the `plot_pc` function that plots and labels the rows of V^T. We then call this function to plot the 1st row of V^T, i.e., the row of V^T that corresponds to `pc1`.

Note: If you get an error when running this cell, make sure you are properly assigning the `vt` variable from Question 6.

[66]:
```
# just run this cell

def plot_pc(col_names, row_mat_vt, k):
    plt.bar(col_names, row_mat_vt[k, :], alpha=0.7)
    plt.xticks(col_names, rotation=90);
```

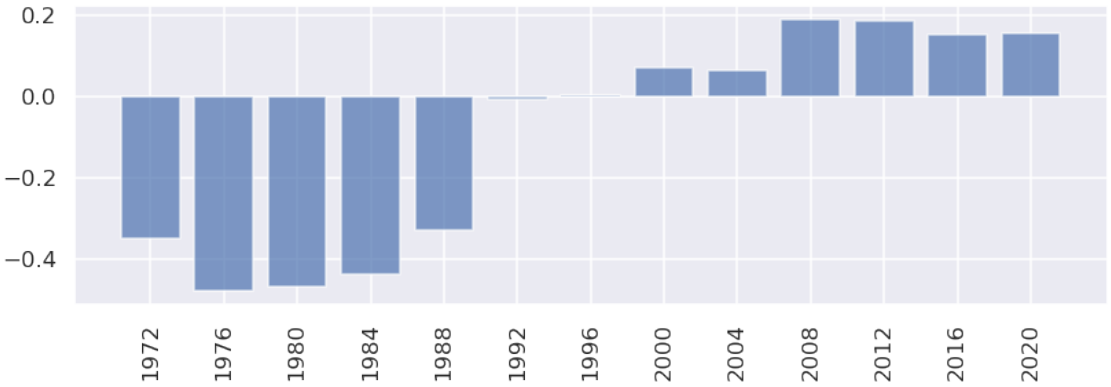
```
plt.figure(figsize=(12, 4)) # adjusts size of plot
plot_pc(list(df_standardized.columns), vt, 0);
```



1.12.1 Question 8a

In the cell below, plot the the 2nd row of V^T, i.e., the row of V^T that correpsonds to `pc2`.

[67]:
```
# BEGIN SOLUTION
plt.figure(figsize=(12, 4))
plot_pc(list(df_standardized.columns), vt, 1);
# END SOLUTION
```



1.12.2 Question 8b

Using the two above plots of the rows of V^T as well as the original table, give a description of what it means for a point to be in the top-right quadrant of the 2-D scatter plot from Question 7.

In other words, what is generally true about a state with relatively large positive value for `pc1` (right side of 2-D scatter plot)? For a large positive value for `pc2` (top side of 2-D scatter plot)?

Notes: * `pc2` is pretty hard to interpret, and the staff doesn't really have a consensus on what it means either. We'll be very nice when grading as long as your answer is reasonable - there is no correct answer necessarily. * Principal components beyond the first are often hard to interpret (but not always; see the lab).

Type your answer here, replacing this text.

SOLUTION: States with relatively large positive values for pc1 tend to vote more democratically. Having a larger pc2 values suggests that states had more variance within their pc1 grouping. In other words, they switched between voting for democratic/republican candidates more often than other states with the same voting trend.

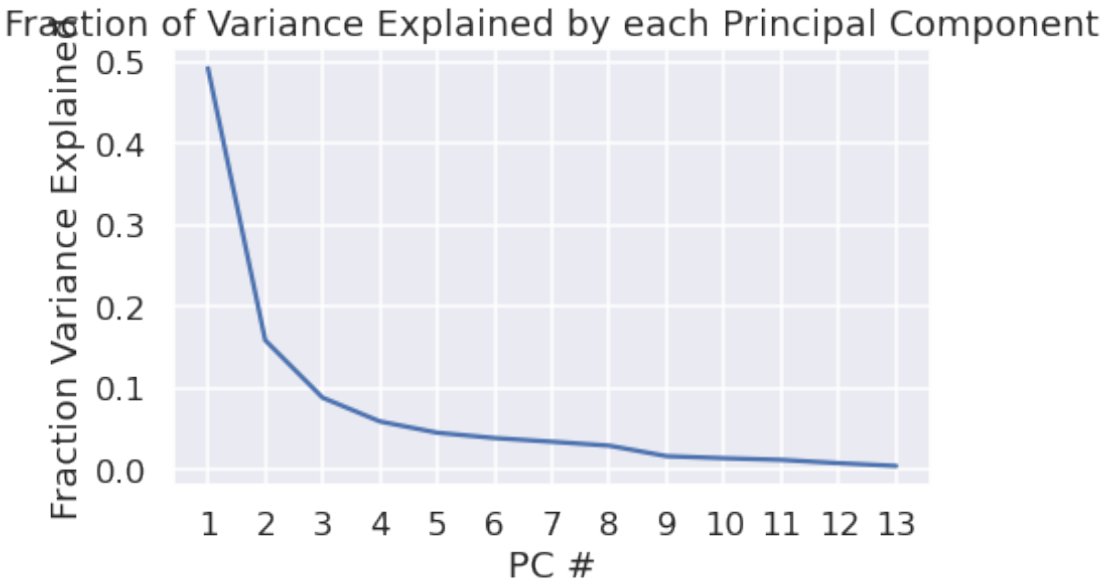
[68]:
```
# feel free to use this cell for scratch work. If you need more scratch space,␣
↪add cells *below* this one.

# Make sure to put your actual answer in the cell above where it says "Type␣
↪your answer here, replacing this text"
```

1.12.3 Question 8c

To get a better sense of whether our 2D scatterplot captures the whole story, create a **scree plot** for this data. In other words, plot the fraction of the total variance (y-axis) captured by the ith principal component (x-axis).

Hint: Be sure to label your axes appropriately! You may find `plt.xticks()` (documentation) helpful for formatting. Also check out the lab for more on scree plots.

[69]:
```
# BEGIN SOLUTION
pc_nums = range(1, len(s) + 1)
plt.plot(pc_nums, s**2 / np.sum(s**2))
plt.xlabel('PC #')
plt.xticks(pc_nums, pc_nums)
plt.ylabel('Fraction Variance Explained');
plt.title('Fraction of Variance Explained by each Principal Component');
# END SOLUTION
```



From your scree plot above, you should see that the first two principal components capture a large portion of the variance in our cleaned data. It is partially for this reason that the 2D scatter plot of principal components was easy to interpret.

1.13 Congratulations!

Congrats! You are finished with this homework assignment.

[ ]:

To double-check your work, the cell below will rerun all of the autograder tests.

[ ]:
```
grader.check_all()
```

1.14 Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit. **Please save before exporting!**

[ ]:
```
# Save your notebook first, then run this cell to export your submission.
grader.export()
```