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

In [ ]:
```
import numpy as np
import pandas as pd
import seaborn as sns
SEED = 3383
```

0. Data preparation

We will load a dataset from voteview.com, which tabulates the roll-call vote of every session of the U.S. Congress. Here, we load all of the votes cast by senators between the 91st and 111th sessions of Congress.

In [ ]:
```
senate = pd.read_csv("senate_votes.csv")
senate.head(6)
```

The `rollnumber` is a serial number of the bill or resolution that was voted on, and `icpsr` is a unique identifier for everyone who has ever served in Congress.

Without going into parliamentary details, there are three codes that mean a "yea" vote and three that mean "nay." We will encode these as +1 and -1, respectively, and anything else is a 0.

In [ ]:
```
def vote_type(v):
    if v in [1,2,3]: return 1
    elif v in [4,5,6]: return -1
    else: return 0

senate["vote"] = senate["cast_code"].apply(vote_type)
senate.head(6)
```

In addition to the voting records, we will also access to the party affiliations of the individual senators. This information is loaded from a separate file here.

In [ ]:
```
members = pd.read_csv("HSall_members.csv").set_index("icpsr")
members.tail(6)
```

There have been over 50 political parties for senators over U.S. history, as encoded by `party_code` above. For modern times, we want to identify the two major parties and label all the others as independent. By indexing a series with default value *Ind* on all the unique party codes, we only need to set the two values we care about and then make the replacement.

In [ ]:
```
party_name = pd.Series("Ind",index=members["party_code"].unique())   # all codes are set
party_name[100] = "Dem"
party_name[200] = "Rep"
members["party"] = members["party_code"].replace(party_name)

print(members["party"].value_counts())
```

We now write a function that returns the data we need for a single session. One output is a table having the senator ID as the index, the individual roll calls as the columns, and values that are the individual votes. This is a job for the `pivot` method. Any senator with a `NaN` in the record did not serve the full

session, and their row is dropped. The other output is the list of party affiliations for the senators in the table.

In [ ]:
```
def session(number,senate,members):
    votes = senate.loc[senate["congress"]==number].pivot(index="icpsr",columns="rollnumb
    party = members.loc[members["congress"]==number,"party"][votes.index]
    # For more than one affiliation, use only the last one:
    party = party.loc[np.bitwise_not(party.index.duplicated(keep="last"))]
    return votes,party
```

So, for example, in the 100th Congress we have

In [ ]:
```
X100,y100 = session(100,senate,members)
print("party counts:")
print(y100.value_counts())
print("feature matrix:")
X100
```

It's interesting to view this feature matrix as a *heat map*.

In [ ]:
```
sns.heatmap(X100);
```

Each row is the voting record of a senator. You can definitely see some dominant motifs in the rows, implying party lines. For the rest of the lab, you will investigate the extent of polarization in the Senate over time.

1. Case studies

1.1 For the 95th Congress, compute the distance matrix (matrix of pairwise distances between voting records) using the 2-norm. Make a histogram showing the distribution of all the distances. (Hint: use `A.flatten()` to transform any array `A` into a vector.)

Type your answer here, replacing this text.

In [ ]:
```

```

1.2 Repeat problem 1.1 for the 110th Congress.

Type your answer here, replacing this text.

In [ ]:
```

```

1.3 Party affiliations effectively define a clustering for each session. For the 95th Congress, make a violin plot showing the distribution of silhouette values for each cluster as defined by party affiliation. (You want one violin per party, although there is only one Independent senator.)

In [ ]:
```

```

1.4 Repeat problem 1.3 for the 110th Congress.

In [ ]:
```

```

The distributions plotted above hint that there may be clusters within a session that have differing

characteristics, so we may want to track the silhouette scores by cluster. They also suggest that we might want to use medians rather than means as central values, since there can be significant asymmetric outliers in the distributions.

2. K-means

2.1 Use k-means with $k = 2$ clusters to fit the voting data for session number 95. **Make sure to initialize the clusterer with a random state equal to `SEED`.** Compute the silhouette values for all the samples; then, compute the median silhouette value in each cluster. (One option is to create a frame with all the values, and use `groupby` to get the medians.)

In [ ]:
```
# The following value should be a vector (1D array).
median95 = ...

print("Medians for k-means, k=2, session 95:")
print(median95)
```

In [ ]:
```
grader.check("kmeans-95")
```

2.2 Repeat the calculation you did in problem 2.1 for all sessions from 91 to 111 (inclusive). For each session, add the median scores for both clusters to a frame, so that the final frame has 42 rows and 2 columns.

In [ ]:
```
# Use the following column names in your response.
results_km2 = pd.DataFrame({"session":[],"silhouette":[]})


print("K-means for k=2:")
print(results_km2.head(10))
```

In [ ]:
```
grader.check("kmeans-k=2")
```

2.3 Plot the median silhouette scores with dots as a function of session number.

In [ ]:
```

```

The conclusion from the k-means results is that there has been a substantial increase over time in the formation of two distinct clusters in the voting records.

3. Agglomerative clustering

3.1 Repeat problem 2.2, but using agglomerative clustering with average linkage and both 2 and 3 clusters. (Note: This type of clustering does not use a random seed.) You will get 5 median values for each session.

In [ ]:
```
# Use the following column names in your response.
results_agg = pd.DataFrame({"session":[],"clusters":[],"silhouette":[]})


print("Agglomerative:")
print(results_agg.head(10))
```

In [ ]:
```
grader.check("agglom-results")
```

3.2 Make a plot like the one in problem 2.3, but add color to the dots to designate the number of clusters.

In [ ]:
```

```

The conclusion from the above results is that in every session, interpreting the records as 2 clusters is more justifiable than interpreting them as 3 clusters.

4. DBSCAN

In this dataset we can estimate a good value for ϵ in the DBSCAN algorithm. The goal is to pick a value larger than the closest neighbors but smaller than cases you would not consider to be similar. Let's use a benchmark of similarity as voting identically at least 80% of the time. Then the difference in voting records can have at most $0.2V$ nonzeros, where V is the number of votes taken in the session. The nonzero values are all ± 2, so the 2-norm distance between similar senators is bounded by

$$2\sqrt{0.2V} \approx 0.894\sqrt{V}.$$

Our strategy will therefore be to try the values $\epsilon = \gamma\sqrt{V}$, where $\gamma = 0.5, 0.55, 0.6, \ldots, 1.2$.

4.1 For session 110, set $N_{\min} = 4$ and iterate over the values of ϵ above to fit the voting data using DBSCAN. For each fit, record the number of non-noise clusters and the total number of noise samples.

In [ ]:
```
# Use the following column names in your response.
fits_110 = pd.DataFrame({"eps":[],"clusters":[],"noise":[]})


print("DBSCAN for session 110:")
print(fits_110)
```

In [ ]:
```
grader.check("dbscan-110")
```

4.2 For each session, use the strategy of problem 4.1 to apply DBSCAN. Among all the fits with two non-noise clusters, choose the fit that minimizes the number of noise samples, and compute the adjusted Rand index compared to the party affiliations.

In [ ]:
```
# Use the following column names in your response.
results_db = pd.DataFrame({"session":[],"noise":[],"ARI":[]})


print("DBSCAN for all sessions:")
print(results_db.head(10))
```

In [ ]:
```
grader.check("dbscan-ARI")
```

4.3 In two separate graphs, plot the ARI as a function of session number and the number of noise samples as a function of session number.

In [ ]:
```

```

In [ ]:
```

```

From the above we conclude that party affiliations have become increasingly reflective of the natural clustering behavior in voting records, and that the number of senators who do not fit well into either cluster has dropped from a substantial minority of the chamber to nearly zero.

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

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

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.

Select *Kernel/Restart & Run All*, then save, then run this export cell again. Submit by pushing the resulting zip file to your GitHub assignment repo.

In [ ]:
```
grader.export(pdf=False, force_save=True)
```