Python Code for QSS Chapter 4: Prediction

Kosuke Imai, Python code by Jeff Allen

First Printing

[ ]:
```
# import libraries with conventional aliases
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

Section 4.1: Predicting Election Outcomes

Section 4.1.1: Loops in Python

[ ]:
```
values = np.array([2, 4, 6])
n = len(values) # number of elements in values
results = np.zeros(n) # empty container vector for storing the results

# looper counter `i` will take on values 0, 1, ..., n in that order
for i in range(n):
    # store multiplication results as the ith element of `results` vector
    results[i] = values[i] * 2
    print(f"{values[i]} times 2 is equal to {results[i]}")

results
```

```
2 times 2 is equal to 4.0
4 times 2 is equal to 8.0
6 times 2 is equal to 12.0
```

[ ]:
```
array([ 4.,  8., 12.])
```

[ ]:
```
# check if the code runs when i = 0
# i = 0 represents the first element in 'values'
i = 0
x = values[i] * 2
print(f"{values[i]} times 2 is equal to {x}")
```

```
2 times 2 is equal to 4
```

Section 4.1.2: General Conditional Statements in Python

[ ]:
```
# define the operation to be executed
operation = 'add'

if operation=='add':
    print('I will perform addition 4 + 4')
    print(4 + 4)

if operation=='multiply':
    print('I will perform multiplication 4 * 4')
    print(4 * 4)
```

```
I will perform addition 4 + 4
8
```

[ ]:
```
# Note that 'operation' is redefined
operation = 'multiply'

if operation=='add':
    print('I will perform addition 4 + 4')
    print(4 + 4)
else:
    print('I will perform multiplication 4 * 4')
    print(4 * 4)
```

```
I will perform multiplication 4 * 4
16
```

[ ]:
```
# Note that 'operation' is redefined
operation = 'subtract'

if operation=='add':
    print('I will perform addition 4 + 4')
    print(4 + 4)
elif operation=='multiply':
    print('I will perform multiplication 4 * 4')
    print(4 * 4)
else:
    print(f"'{operation}' is invalid. Use either 'add' or 'multiply'.")
```

```
'subtract' is invalid. Use either 'add' or 'multiply'.
```

[ ]:
```
values = np.arange(1,6)
n = len(values)
results = np.zeros(n)

for i in range(n):
    # x and r get overwritten in each iteration
```

```
    x = values[i]
    r = x % 2 # remainder of x divided by 2 to check if x is even or odd
    if r==0: # remainder is 0
        print(f"{x} is even and I will perform addition {x} + {x}")
        results[i] = x + x
    else: # remainder is not 0
        print(f"{x} is odd and I will perform multiplication {x} * {x}")
        results[i] = x * x

results
```

```
1 is odd and I will perform multiplication 1 * 1
2 is even and I will perform addition 2 + 2
3 is odd and I will perform multiplication 3 * 3
4 is even and I will perform addition 4 + 4
5 is odd and I will perform multiplication 5 * 5
```

[ ]: array([1., 4., 9., 8., 25.])

Section 4.1.3: Poll Predictions

[ ]:
```
# import the datetime module
from datetime import datetime

# load election results, by state
pres08 = pd.read_csv('pres08.csv')

# load polling data
polls08 = pd.read_csv('polls08.csv')

# compute Obama's margin
polls08['margin'] = polls08['Obama'] - polls08['McCain']
pres08['margin'] = pres08['Obama'] - pres08['McCain']

x = datetime.strptime('2008-11-04', '%Y-%m-%d')
y = datetime.strptime('2008/9/1', '%Y/%m/%d')

# number of days between 9/1/2008 and 11/4/2008
x-y # a timedelta object
```

[ ]: datetime.timedelta(days=64)

[ ]:
```
# number of days as an integer
(x-y).days
```

[ ]: 64

```
[ ]: # convert middate to datetime object using pandas convenience function
polls08['middate'] = pd.to_datetime(polls08['middate'])

# compute the number of days to the election; use x defined above
# extract days using the .dt accessor
polls08['days_to_election'] = (x - polls08['middate']).dt.days

# extract unique state names which the loop will iterate through
st_names = polls08['state'].unique()

# initialize a container vector for storing the results as a series
poll_pred = pd.Series(index=st_names)

poll_pred.head()
```

```
[ ]: AL   NaN
AK   NaN
AZ   NaN
AR   NaN
CA   NaN
dtype: float64
```

```
[ ]: # loop across the 50 states plus DC
for i in range(len(st_names)):
    # subset the ith state
    state_data = polls08[polls08['state']==st_names[i]]
    # further subset the latest polls within the state
    latest = (state_data[state_data['days_to_election']==
                         state_data['days_to_election'].min()])
    # compute the mean of the latest polls and store it
    poll_pred[i] = latest['margin'].mean()

poll_pred.head(10)
```

```
[ ]: AL   -25.0
AK   -19.0
AZ    -2.5
AR    -7.0
CA    24.0
CO     7.0
CT    25.0
DC    69.0
DE    30.0
FL     2.0
dtype: float64
```

Because we stored the state identifier as the index, we could use states as the loop counter. In complex numeric indexing cases, looping through names can be a good alternative.

[ ]:
```python
poll_pred_alt = pd.Series(index=st_names)

# loop across the 50 states plus DC
for state in st_names:
    # subset the polls data for the current state
    state_data = polls08[polls08['state']==state]
    # subset the latest poll for the current state
    latest = (state_data[state_data['days_to_election']==
                         state_data['days_to_election'].min()])
    # compute the mean of the latest poll and store it in the results vector
    poll_pred_alt[state] = latest['margin'].mean()

# check that results are the same
poll_pred.equals(poll_pred_alt)
```

[ ]:
```
True
```

Recall from chapter 3 that if we want to perform element-wise arithmetic on two equal length vectors whose elements are sorted correctly, the indexes should be identical. Since the `poll_pred` index is state abbreviations, we can reset the `pres08` index to state abbreviations and then extract the margin column without modifying the data frame in place. Of course, we could also add `poll_pred` to the data frame, which we will illustrate later.

[ ]:
```python
# errors of latest polls
errors = pres08.set_index('state')['margin'] - poll_pred

errors.head()
```

[ ]:
```
state
AL     4.0
AK    -2.0
AZ    -6.5
AR   -13.0
CA     0.0
dtype: float64
```

[ ]:
```python
# mean prediction error
errors.mean()
```

[ ]:
```
1.0620915032679739
```

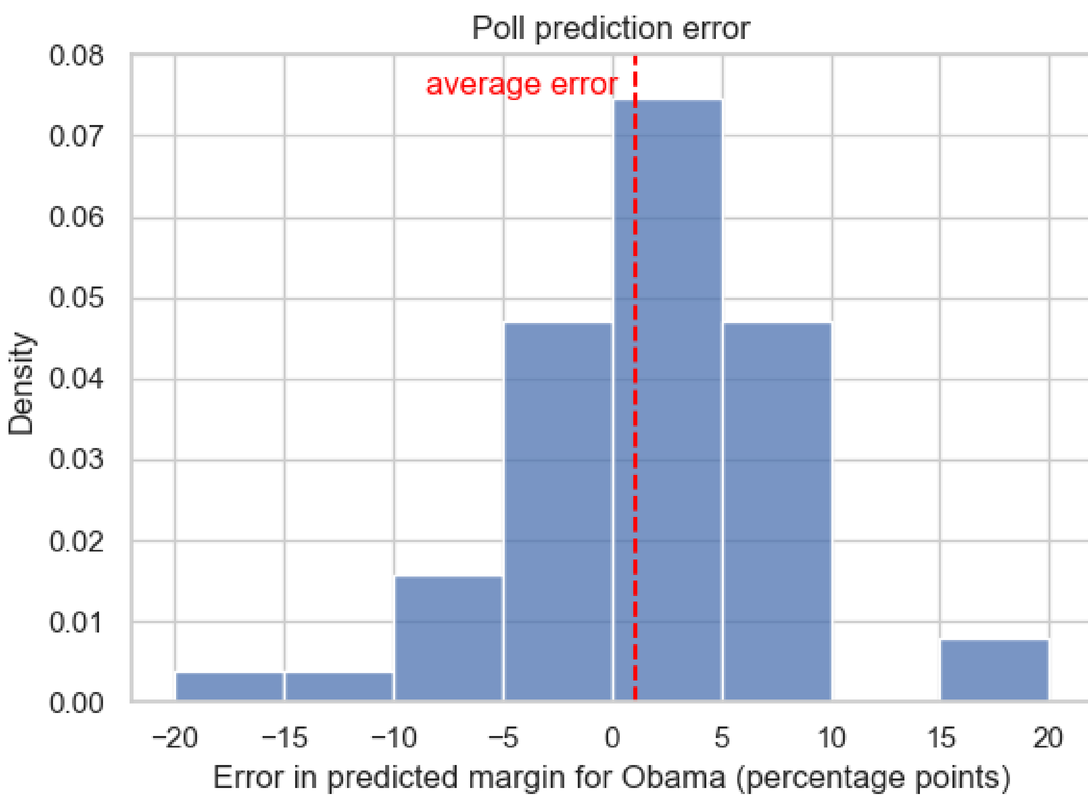
[ ]:
```python
# root mean squared prediction error
np.sqrt((errors**2).mean())
```

[ ]:
```
5.908940458495747
```

[ ]:
```python
# histogram of errors
sns.set_theme(style="whitegrid")
```

```
sns.displot(
    x=errors, stat='density', binrange=(-20, 20), binwidth=5,
    height=4, aspect=1.5,
).set(xlabel='Error in predicted margin for Obama (percentage points)',
      title='Poll prediction error',
      ylim=(0, 0.08)).despine(right=False, top=False)

# add a vertical line representing the mean
plt.axvline(x=errors.mean(), color='red', linestyle='--')

# add a text label for the median
plt.text(x=-8.5, y=0.075, s='average error', color='red')
```

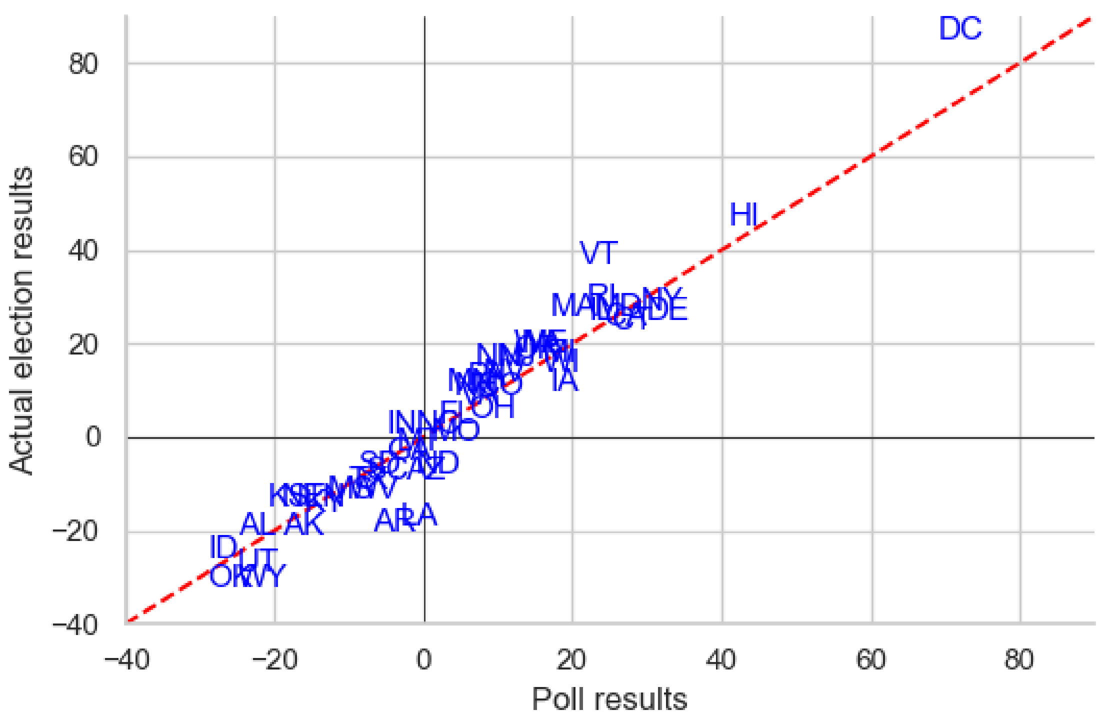
[ ]: Text(-8.5, 0.075, 'average error')



[ ]:
```
# add poll_pred to pres08 for easier plotting and analysis
# reset the index to match the index of pres08 and drop the old index
pres08['poll_pred'] = poll_pred.reset_index(drop=True)

# marker='' generates an "empty" plot
```

```python
sns.relplot(
    data=pres08, x='poll_pred', y='margin', marker='',
    height=4, aspect=1.5,
).set(xlabel='Poll results', ylabel='Actual election results',
      ylim=(-40, 90), xlim=(-40, 90))

# add state abbreviations
for i in range(len(pres08['state'])):
    plt.text(x=pres08['poll_pred'][i], y=pres08['margin'][i],
             s=pres08['state'][i], color='blue')

# add 45 degree line
plt.gca().axline((0, 0), slope=1, color='red', linestyle='--')

# add vertical and horizontal lines at 0
plt.axvline(x=0, color='black', linewidth=0.5)
plt.axhline(y=0, color='black', linewidth=0.5)
```

[ ]: <matplotlib.lines.Line2D at 0x26ee249d0f0>



[ ]:
```python
# which state polls called the election wrong?
pres08['state'][np.sign(pres08['poll_pred']) != np.sign(pres08['margin'])]
```

[ ]:
```
14    IN
25    MO
33    NC
Name: state, dtype: object
```

[ ]:
```python
# what was the actual margin for these states?
pres08['margin'][np.sign(pres08['poll_pred']) != np.sign(pres08['margin'])]
```

[ ]:
```
14    1
25   -1
33    1
Name: margin, dtype: int64
```

[ ]:
```python
# actual results: total number of electoral votes won by Obama
pres08['EV'][pres08['margin']>0].sum()
```

[ ]:
```
364
```

[ ]:
```python
# poll prediction
pres08['EV'][pres08['poll_pred']>0].sum()
```

[ ]:
```
349
```

[ ]:
```python
# load the data
pollsUS08 = pd.read_csv('pollsUS08.csv')

# compute number of days to the election as before
pollsUS08['middate'] = pd.to_datetime(pollsUS08['middate'])

pollsUS08['days_to_election'] = (x - pollsUS08['middate']).dt.days

# empty numpy vectors to store predictions for Obama and McCain
Obama_pred = np.zeros(90)
McCain_pred = np.zeros(90)
```

With zero-based indexing, the days sequence 1-90 does not match the vector index 0-89. We need to account for this somewhere. One option, among many, is to add 1 to the loop counter when working with the days sequence.

[ ]:
```python
for i in range(len(Obama_pred)):
    # take all polls conducted within the past 7 days
    week_data = (pollsUS08[(pollsUS08['days_to_election'] <= (90 - (i + 1) + 7))
                           & (pollsUS08['days_to_election'] > (90 - (i + 1)))])
    # compute the mean of the polls for Obama and McCain
    Obama_pred[i] = week_data['Obama'].mean()
    McCain_pred[i] = week_data['McCain'].mean()

# put together a data frame for plotting
```

```python
pollsUS08_avg = pd.DataFrame({'Obama': Obama_pred,
                              'McCain': McCain_pred,
                              'days_to_election': np.arange(90, 0, -1)})

pollsUS08_avg.head()
```

[ ]:

	Obama	McCain	days_to_election
0	44.538462	40.692308	90
1	45.000000	40.692308	89
2	45.230769	40.846154	88
3	45.750000	42.000000	87
4	45.888889	42.000000	86

Recall from chapter 3 that plotting groups in seaborn works best when the grouping variable is stored in its own column. In this case, the grouping variable is the candidate. To pivot the candidates into a single column, we need to reshape the data into a longer format, which can be accomplished with the `melt()` method in pandas.

[ ]:

```python
# reshape the data: pivot longer using melt
pollsUS08_avg_long = pollsUS08_avg.melt(id_vars='days_to_election',
                                        var_name='Candidate',
                                        value_name='poll_avg')

pollsUS08_avg_long.head()
```

[ ]:

	days_to_election	Candidate	poll_avg
0	90	Obama	44.538462
1	89	Obama	45.000000
2	88	Obama	45.230769
3	87	Obama	45.750000
4	86	Obama	45.888889

[ ]:

```python
pollsUS08_avg_long.tail()
```

[ ]:

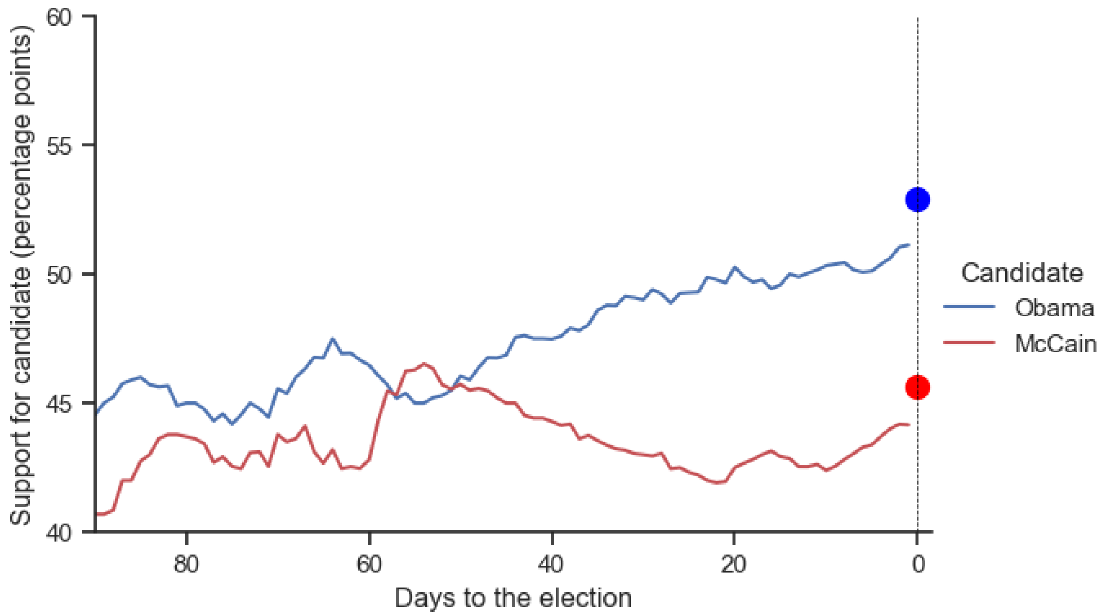
	days_to_election	Candidate	poll_avg
175	5	McCain	43.384615
176	4	McCain	43.708333
177	3	McCain	44.000000
178	2	McCain	44.185185
179	1	McCain	44.160000

[ ]:

```python
sns.set_theme(style="ticks")

# plot going from 90 days to 1 day before the election
sns.relplot(
    data=pollsUS08_avg_long, x='days_to_election', y='poll_avg',
    hue='Candidate', kind='line',
    palette=['b', 'r'], height=4, aspect=1.5
```

```
).set(ylim=(40, 60), yticks=range(40, 61, 5),
      xlim=(90, -1.5), # small buffer in right limit for aesthetics
      xlabel='Days to the election',
      ylabel='Support for candidate (percentage points)')

# line indicating election day
plt.axvline(x=0, color='black', linestyle='--', linewidth=0.5)

# actual election results
plt.scatter(0, 52.93, color='blue', s=100)
plt.scatter(0, 45.65, color='red', s=100)
```

[ ]: <matplotlib.collections.PathCollection at 0x26ee2434f40>



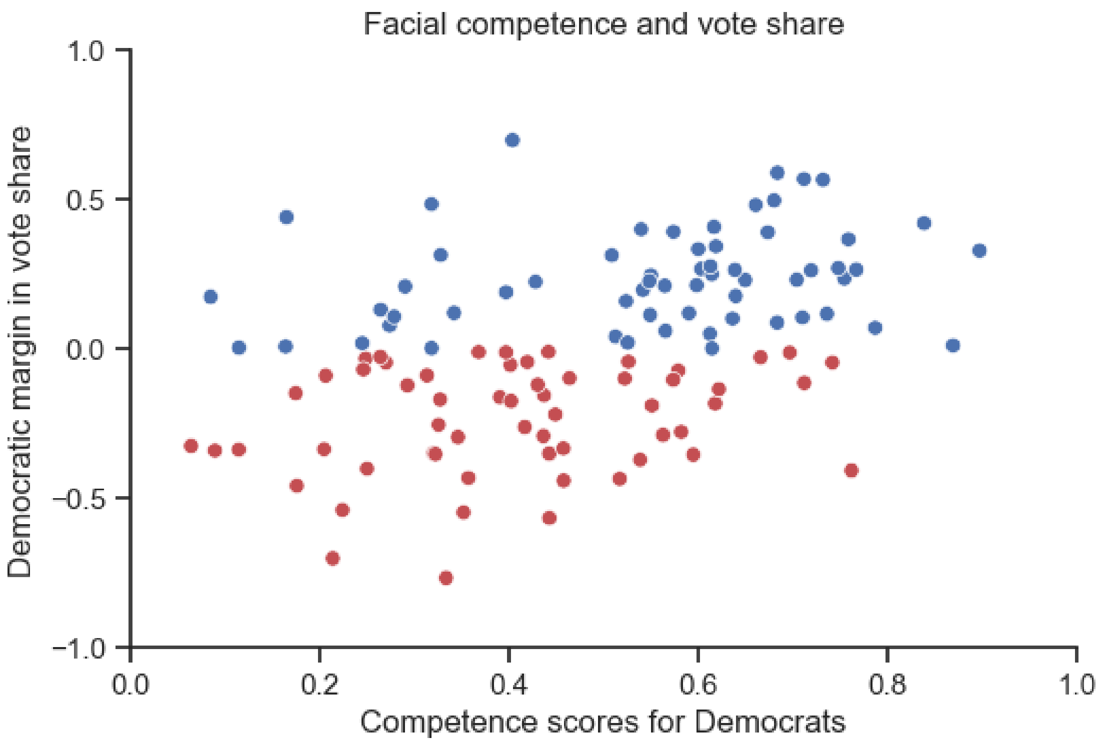
Section 4.2: Linear Regression

Section 4.2.1: Facial Appearance and Election Outcomes

[ ]:
```
# load the data
face = pd.read_csv('face.csv')

# two-party vote share for Democrats and Republicans
face['d_share'] = face['d.votes'] / (face['d.votes'] + face['r.votes'])
face['r_share'] = face['r.votes'] / (face['d.votes'] + face['r.votes'])
face['diff_share'] = face['d_share'] - face['r_share']

sns.relplot(
```

```
    data=face, x='d.comp', y='diff_share',
    hue='w.party', palette=['b','r'], legend=False, height=4, aspect=1.5
).set(xlim=(0, 1), ylim=(-1, 1), yticks=np.arange(-1.0, 1.5, 0.5),
      title='Facial competence and vote share',
      xlabel='Competence scores for Democrats',
      ylabel='Democratic margin in vote share')
```

[ ]: <seaborn.axisgrid.FacetGrid at 0x26ee24dfaf0>



Section 4.2.2: Correlation and Scatter Plots

[ ]:
```
face['d.comp'].corr(face['diff_share'])
```

[ ]: 0.43277434572761064

Section 4.3.3: Least Squares

[ ]:
```
# import the statsmodels formula API
import statsmodels.formula.api as smf
```

`statsmodels` works best when column names do not contain spaces or special characters, such as dots. The chapter appendix provides a more in-depth discussion about why this is the case and how to use the module if you want to retain special characters or spaces in variable names. For now, though, we will replace the dots in the column names with underscores to prevent any errors.

[ ]:
```
# replace dots in column names with underscores
face.columns = face.columns.str.replace('.', '_')

face.columns
```

[ ]:
```
Index(['year', 'state', 'winner', 'loser', 'w_party', 'l_party', 'd_comp',
       'r_comp', 'd_votes', 'r_votes', 'd_share', 'r_share', 'diff_share'],
      dtype='object')
```

[ ]:
```
# fit the model; the statsmodels formula API uses R-style formulas
fit = smf.ols('diff_share ~ d_comp', data=face).fit()

fit.model.formula
```

[ ]:
```
'diff_share ~ d_comp'
```

[ ]:
```
# get the estimated coefficients
fit.params
```

[ ]:
```
Intercept   -0.312226
d_comp       0.660381
dtype: float64
```

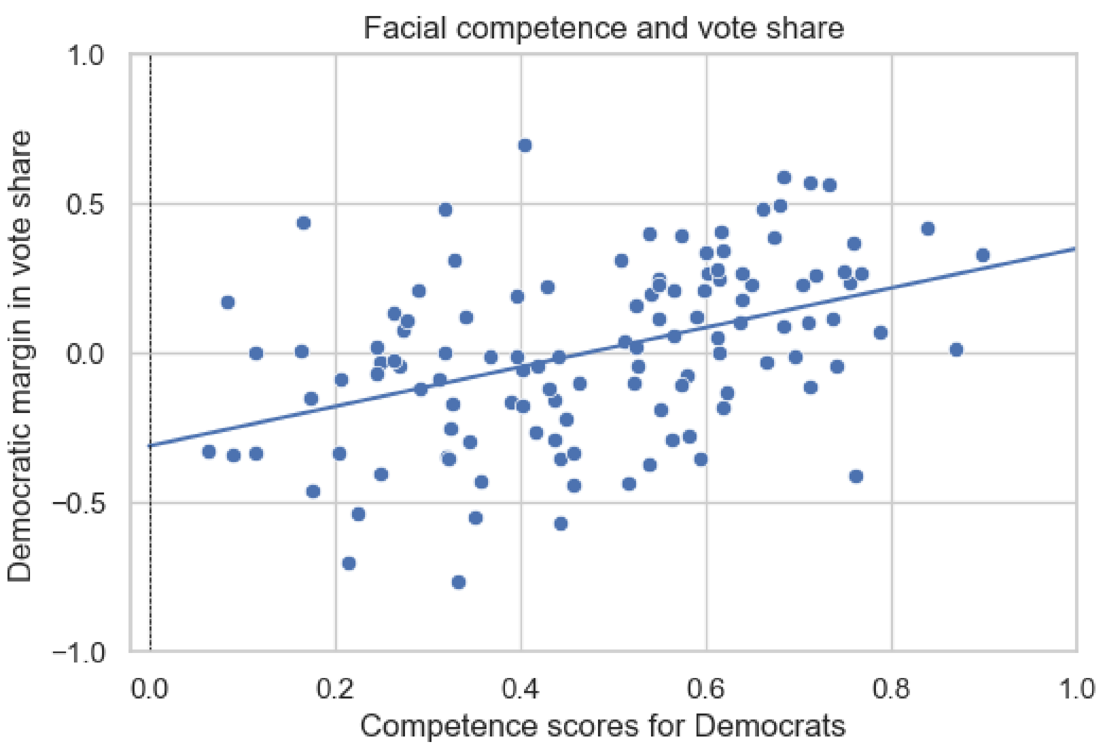
[ ]:
```
# get fitted or predicted values
fit.fittedvalues.head(n=10)
```

[ ]:
```
0    0.060604
1   -0.086433
2    0.092171
3    0.045392
4    0.136987
5   -0.100572
6   -0.045593
7    0.085994
8    0.043438
9    0.261788
dtype: float64
```

[ ]:
```
# store the intercept and slope for plotting a regression line
intercept, slope = fit.params

# generate 100 evenly spaced values between 0-1
x_values = np.linspace(0, 1, 100)

# using the slope and intercept, predict values over the range of x_values
```

```
y_values = intercept + slope * x_values

sns.set_theme(style="whitegrid")

# plot a scatterplot and overlay a regression line
sns.relplot(
    data=face, x='d_comp', y='diff_share', height=4, aspect=1.5
).set(xlim=(-0.02, 1), ylim=(-1, 1), yticks=np.arange(-1.0, 1.5, 0.5),
      title='Facial competence and vote share',
      xlabel='Competence scores for Democrats',
      ylabel='Democratic margin in vote share').despine(right=False, top=False)

plt.plot(x_values, y_values) # regression line

plt.axvline(x=0, color='black', linewidth=0.5, linestyle='--')
```

[ ]: <matplotlib.lines.Line2D at 0x26ee1a99f90>



Note that seaborn has a built-in function for plotting regression lines, which we will use later, but it is not as easy to show the regression line's intercept.

[ ]:
```
epsilon_hat = fit.resid # residuals
np.sqrt((epsilon_hat**2).mean()) # RMSE
```

[ ]: 0.2642360764039512

Section 4.2.4: Regression Towards the Mean

Section 4.2.5: Merging Datasets in Pandas

[ ]:
```
# load the 2012 data
pres12 = pd.read_csv('pres12.csv')

# remove poll_pred from pres08
pres08.drop('poll_pred', axis=1, inplace=True)

# quick look at the two data sets
pres08.head()
```

[ ]:

	state.name	state	Obama	McCain	EV	margin
0	Alabama	AL	39	60	9	-21
1	Alaska	AK	38	59	3	-21
2	Arizona	AZ	45	54	10	-9
3	Arkansas	AR	39	59	6	-20
4	California	CA	61	37	55	24

[ ]:
```
pres12.head()
```

[ ]:

	state	Obama	Romney	EV
0	AL	38	61	9
1	AK	41	55	3
2	AZ	45	54	11
3	AR	37	61	6
4	CA	60	37	55

[ ]:
```
# merge two data frames
pres = pd.merge(pres08, pres12, on='state')

pres.head()
```

[ ]:

	state.name	state	Obama_x	McCain	EV_x	margin	Obama_y	Romney	EV_y
0	Alabama	AL	39	60	9	-21	38	61	9
1	Alaska	AK	38	59	3	-21	41	55	3
2	Arizona	AZ	45	54	10	-9	45	54	11
3	Arkansas	AR	39	59	6	-20	37	61	6
4	California	CA	61	37	55	24	60	37	55

[ ]:
```
pres.describe().round(2)
```

[ ]:

	Obama_x	McCain	EV_x	margin	Obama_y	Romney	EV_y
count	51.00	51.00	51.00	51.00	51.00	51.00	51.00
mean	51.37	47.06	10.55	4.31	49.06	49.04	10.55

```
std     11.04  11.04   9.58   22.07   11.80  11.79   9.69
min     33.00   7.00   3.00  -32.00   25.00   7.00   3.00
25%     43.00  40.00   4.50  -13.00   40.50  41.00   4.50
50%     51.00  47.00   8.00    4.00   51.00  48.00   8.00
75%     57.50  56.00  11.50   17.50   56.00  58.00  11.50
max     92.00  66.00  55.00   85.00   91.00  73.00  55.00
```

[ ]:
```
# change the variable name for illustration
pres12.rename(columns={'state': 'state_abb'}, inplace=True)

pres12.head()
```

[ ]:
```
  state_abb  Obama  Romney  EV
0        AL     38      61   9
1        AK     41      55   3
2        AZ     45      54  11
3        AR     37      61   6
4        CA     60      37  55
```

[ ]:
```
# merging data sets using variable keys with different names
pres = (pd.merge(pres08, pres12, left_on='state', right_on='state_abb').
        drop('state_abb', axis=1))

pres.head()
```

[ ]:
```
   state.name state  Obama_x  McCain  EV_x  margin  Obama_y  Romney  EV_y
0     Alabama    AL       39      60     9     -21       38      61     9
1      Alaska    AK       38      59     3     -21       41      55     3
2     Arizona    AZ       45      54    10      -9       45      54    11
3    Arkansas    AR       39      59     6     -20       37      61     6
4  California    CA       61      37    55      24       60      37    55
```

[ ]:
```
pres.describe().round(2)
```

[ ]:
```
       Obama_x  McCain   EV_x  margin  Obama_y  Romney   EV_y
count    51.00   51.00  51.00   51.00    51.00   51.00  51.00
mean     51.37   47.06  10.55    4.31    49.06   49.04  10.55
std      11.04   11.04   9.58   22.07    11.80   11.79   9.69
min      33.00    7.00   3.00  -32.00    25.00    7.00   3.00
25%      43.00   40.00   4.50  -13.00    40.50   41.00   4.50
50%      51.00   47.00   8.00    4.00    51.00   48.00   8.00
75%      57.50   56.00  11.50   17.50    56.00   58.00  11.50
max      92.00   66.00  55.00   85.00    91.00   73.00  55.00
```

[ ]:
```
# concatenate two data frames
pres1 = pd.concat([pres08, pres12], axis='columns')

pres1.head()
```

[ ]:

	state.name	state	Obama	McCain	EV	margin	state_abb	Obama	Romney	EV
0	Alabama	AL	39	60	9	-21	AL	38	61	9
1	Alaska	AK	38	59	3	-21	AK	41	55	3
2	Arizona	AZ	45	54	10	-9	AZ	45	54	11
3	Arkansas	AR	39	59	6	-20	AR	37	61	6
4	California	CA	61	37	55	24	CA	60	37	55

[ ]:
```
'''
DC and DE are flipped in this alternative approach, and we have overlapping
column names.
'''
pres1.iloc[7:9]
```

[ ]:

	state.name	state	Obama	McCain	EV	margin	state_abb	Obama	Romney	EV
7	D.C.	DC	92	7	3	85	DE	59	40	3
8	Delaware	DE	62	37	3	25	DC	91	7	3

[ ]:
```
# merge() does not have this problem
pres.iloc[7:9]
```

[ ]:

	state.name	state	Obama_x	McCain	EV_x	margin	Obama_y	Romney	EV_y
7	D.C.	DC	92	7	3	85	91	7	3
8	Delaware	DE	62	37	3	25	59	40	3

If we move the state identifier to the index, then `concat()` will align the indexes correctly. We still have overlapping column names, though.

[ ]:
```
pres2 = pd.concat([pres08.set_index('state'),
                   pres12.set_index('state_abb')], axis='columns')

pres2.iloc[7:9]
```

[ ]:

	state.name	Obama	McCain	EV	margin	Obama	Romney	EV
DC	D.C.	92	7	3	85	91	7	3
DE	Delaware	62	37	3	25	59	40	3

Pandas and numpy do not have built-in z-score functions. We can either calculate the z-scores manually, use the zscore function from the scipy module, or build a simple function of our own. In this case, the final option is straightforward.

[ ]:
```
# define a function to standardize a vector (calculate z-scores)
def standardize(x):
    return (x - x.mean()) / x.std()

pres['Obama2008_z'] = standardize(pres['Obama_x'])
pres['Obama2012_z'] = standardize(pres['Obama_y'])

# estimated intercept is essentially zero
fit1 = smf.ols('Obama2012_z ~ Obama2008_z', data=pres).fit()
```

```python
fit1.params
```

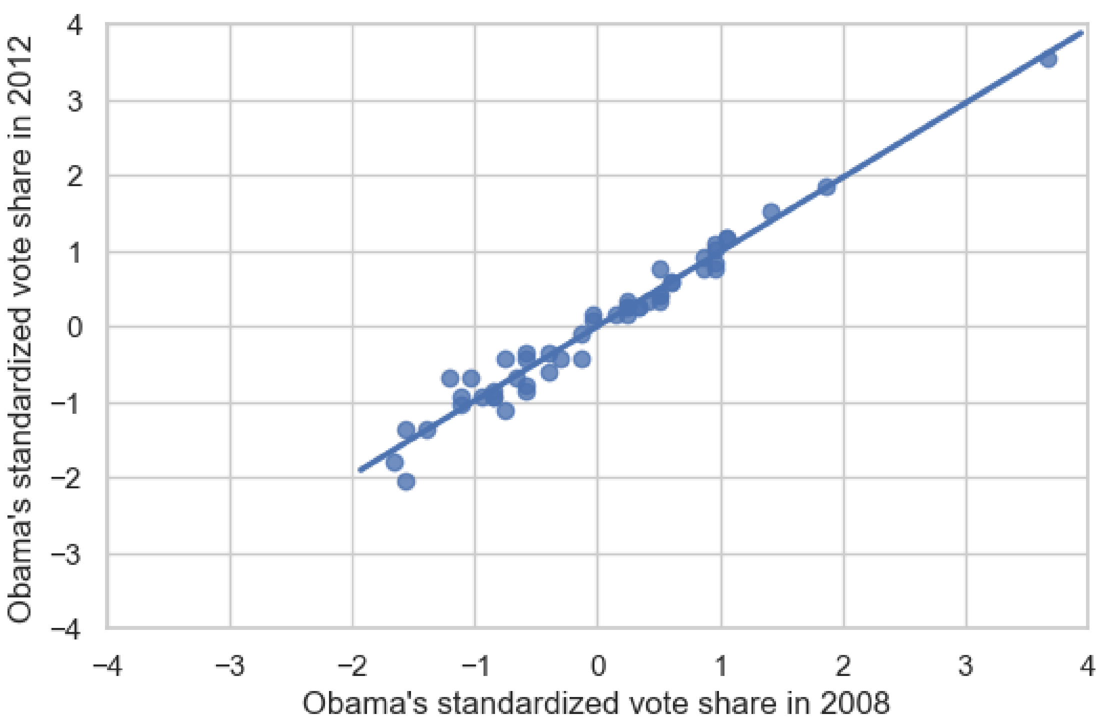
[ ]:
```
Intercept      -2.914335e-16
Obama2008_z     9.834419e-01
dtype: float64
```

[ ]:
```python
# regression without an intercept
fit1 = smf.ols('Obama2012_z ~ -1 + Obama2008_z', data=pres).fit()

# estimated slope is identical
fit1.params
```

[ ]:
```
Obama2008_z    0.983442
dtype: float64
```

[ ]:
```python
# plot using seaborn's built-in lmplot function
sns.lmplot(
    data=pres, x='Obama2008_z', y='Obama2012_z', ci=None, truncate=False,
    height=4, aspect=1.5,
).set(xlim=(-4, 4), ylim=(-4, 4),
      xlabel="Obama's standardized vote share in 2008",
      ylabel="Obama's standardized vote share in 2012").despine(
          right=False, top=False)
```

[ ]: <seaborn.axisgrid.FacetGrid at 0x26ee3f98250>



Setting `truncate=False` extends the regression line a bit past the data range, but only up to the axis limits that `lmplot()` sets internally, not to the axis limits we set manually in `.set()`.

[ ]:
```python
# bottom quartile
((pres['Obama2012_z'] > pres['Obama2008_z'])[
    (pres['Obama2008_z'] <= pres['Obama2008_z'].quantile(0.25))].mean())
```

[ ]:
```
0.5714285714285714
```

[ ]:
```python
# top quartile
((pres['Obama2012_z'] > pres['Obama2008_z'])[
    (pres['Obama2008_z'] >= pres['Obama2008_z'].quantile(0.75))].mean())
```

[ ]:
```
0.46153846153846156
```

Section 4.2.6: Model Fit

In Progress

Appendix: statsmodels considerations

This appendix addresses a few nuances to consider when using the `statsmodels` module.

Section A.1: Interaction with patsy module

Section A.2: Varibles names

Section A.3: Object oriented programming (OOP) workflow

In Progress