Youssef Mahmoud 905854027

In [81]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import sklearn
from sklearn.model_selection import GridSearchCV
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, LeaveOneOut, KFold, cross_val_score
from sklearn.pipeline import Pipeline
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import PolynomialFeatures
import statsmodels.api as sm
import statsmodels.formula.api as smf
from patsy import dmatrix
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import scale
from sklearn import model_selection
from sklearn.linear_model import LinearRegression, Ridge, RidgeCV, Lasso, LassoCV, ElasticNet, ElasticNetCV
from sklearn.decomposition import PCA
from sklearn.cross_decomposition import PLSRegression

from sklearn.metrics import mean_squared_error
```

1

In a linear model with Gaussian errors, the likelihood function is given by:

$L(\beta) = (2\pi)^{-n/2} * \det(\Sigma)^{-1/2} * \exp(-1/2 * (y - X\beta)^T \Sigma^{-1} (y - X\beta))$

then we take the log of the function:

$\ln(L(\beta)) = -n/2 * \ln(2\pi) - 1/2 * \ln(\det(\Sigma)) - 1/2 * (y - X\beta)^T \Sigma^{-1} (y - X\beta)$

The first 2 variables on the right-hand side are constants with respect to Beta, therefore, they do no affect the Beta. The third term can be written as:

$1/2 * (y - X\beta)^T \Sigma^{-1} (y - X\beta) = - 1/2 * ||y - X\beta||^2$

So the log-likelihood function is proportional to the SSR. This means that the maximum likelihood estimator of Beta is equivalent to the least squares estimator.

Cp and AIC are both used as tools for determining the best model. Both penalize the log-likelihood function for adding more predictors. In a linear model with Gaussian errors, both Cp and AIC are proportional to the SSR plus a penalty term. The difference between Cp and AIC is in the magnitude of the penalty term.

6.6.1

A

For best subset selection, the model with k predictors has the smallest RSS among all the Ckp models with k predictors.

For forward stepwise selection, the model with k predictors has the smallest RSS among the (p - k) models which add the predictors in Matrix k−1 with 1 extra predictor.

For backward stepwise selection, the model with k predictors has the smallest RSS among the k models which contains all except 1 of the predictors in Matrix of k+1.

As a result, best subset selection has the lowest training RSS since it includes all k predictor models.

B

Usually, best subset selection has the smallest test RSS since it includes more models than forward and backward stepwise selections.

i

True.

ii

True.

iii

False.

iv

False.

v

False.

6.6.3

A

Steadily decrease because as s initially inceases, Betas will increase to the values with least squares estimates, reducing the training RSS.

B

Test RSS will initially decrease, then it will start to inrease forming a U shape. This is because as s initially increases Betas will increase to the values with least squares estimates, reducing the Test RSS, but as the difference between the training RSS and test RSS gets bigger, Test RSS will start increasing, implying that we are overfitting our model.

C

Steadily increase. As s initially increases, Betas will increase to the values with least squares estimates, indicating that the model is becoming more flexible which leads to a steady increase in variance.

D

Steadily decrease. As s initially increases, Betas will increase to the values with least squares estimates, indicating that the model is becoming more flexible which leads to a steady decrease in bias.

E

The irreducible error is always constant since it is not related to the value of s.

6.6.4

A

Steadily increase because as λ initially inceases, Betas will deviate from the values with least squares estimates, making the model less flexible and as a result steadily increase training RSS.

B

Test RSS will initially decrease, then it will start to inrease forming a U shape. This is because as λ initially increases Betas will deviate from the values with least squares estimates, making the model less flexible and as a result the test RSS will initially decrease then start to increase forming a U shape.

C

Steadily decrease. As λ initially increases, Betas will deviate from the values with least squares estimates, indicating that the model is becoming less flexible which leads to a steady decrease in variance.

D

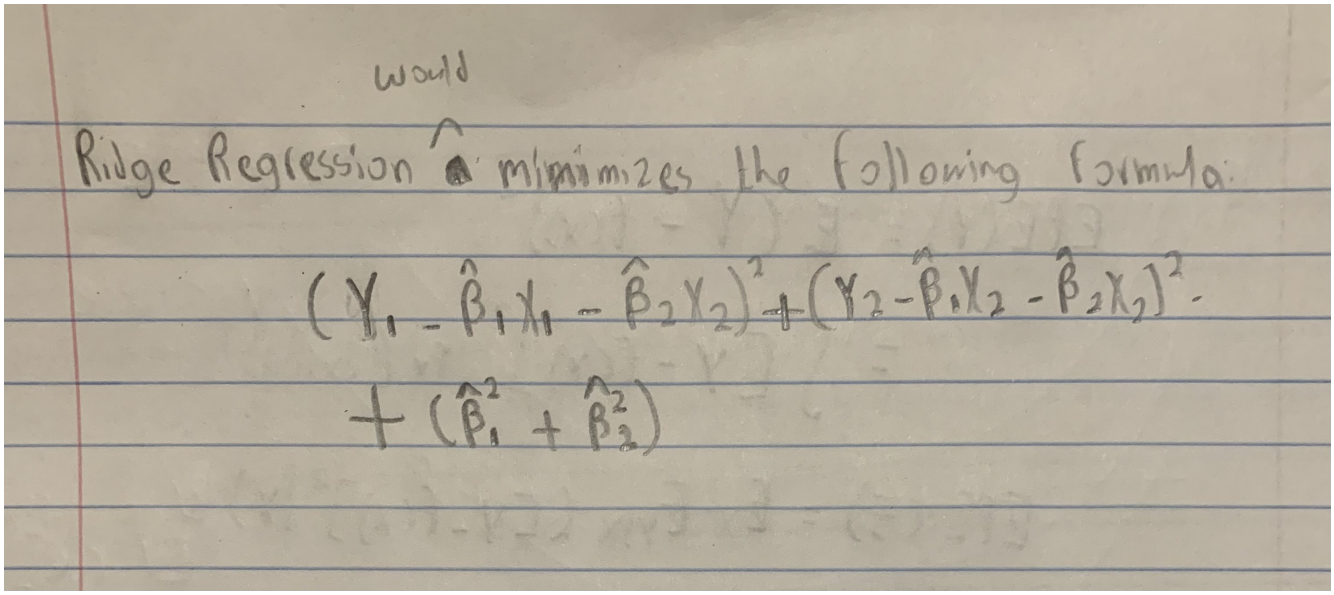
Steadily increase. As λ initially increases, Betas will deviate from the values with least squares estimates, indicating that the model is becoming less flexible which leads to a steady increase in bias.

E

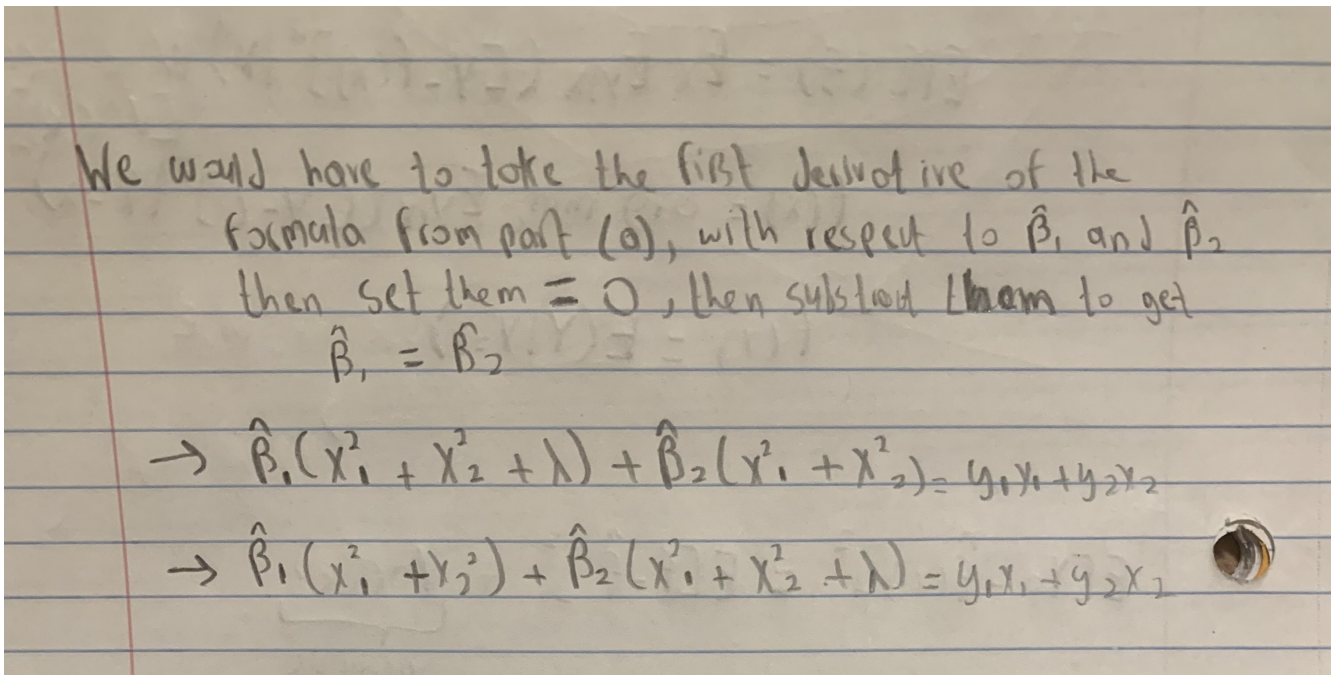
The irreducible error is always constant since it is not related to the value of λ.

6.6.5

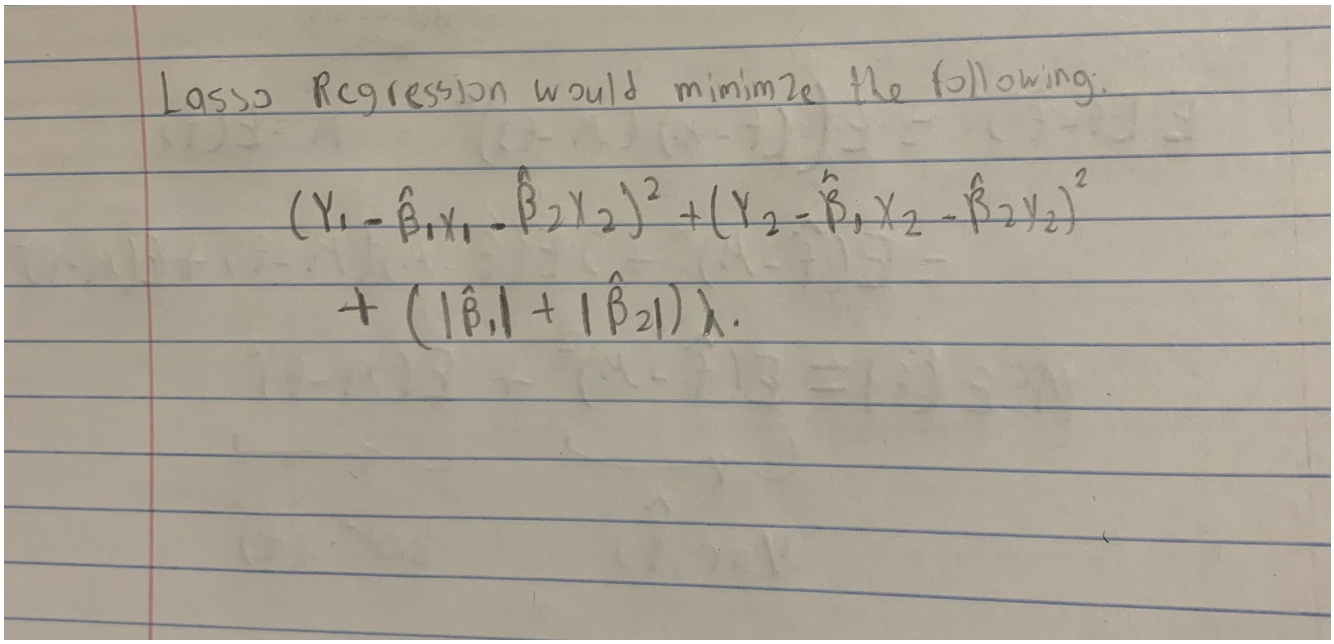
A



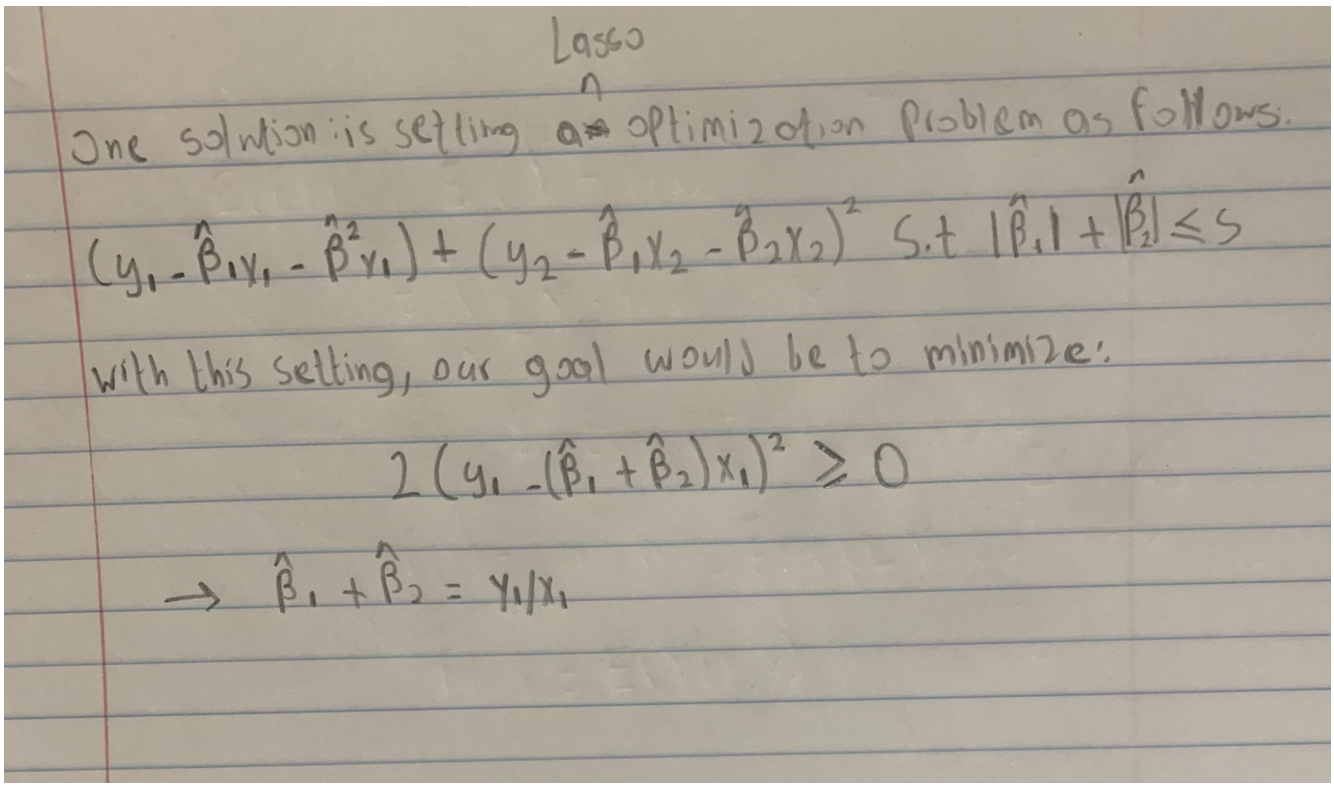
B


We would have to take the first derivative of the formula from part (a), with respect to $\hat{\beta}_1$ and $\hat{\beta}_2$ then set them = 0, then substitute them to get
$\hat{\beta}_1 = \hat{\beta}_2$
$\rightarrow \hat{\beta}_1(x_1^2 + x_2^2 + \lambda) + \hat{\beta}_2(x_1^2 + x_2^2) = y_1x_1 + y_2x_2$
$\rightarrow \hat{\beta}_1(x_1^2 + x_2^2) + \hat{\beta}_2(x_1^2 + x_2^2 + \lambda) = y_1x_1 + y_2x_2$

C


Lasso Regression would minimize the following.
$(Y_1 - \hat{\beta}_1x_1 - \hat{\beta}_2x_2)^2 + (Y_2 - \hat{\beta}_1x_2 - \hat{\beta}_2y_2)^2$
$+ (|\hat{\beta}_1| + |\hat{\beta}_2|)\lambda.$

D



6.6.11

A

In [2]:
```
df = pd.read_csv("desktop/boston.csv")
```

In [3]:
```
df
```

Out[3]:

	Unnamed: 0	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	lstat	medv
0	1	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	4.98	24.0
1	2	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	9.14	21.6
2	3	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	4.03	34.7
3	4	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	2.94	33.4
4	5	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	5.33	36.2
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
501	502	0.06263	0.0	11.93	0	0.573	6.593	69.1	2.4786	1	273	21.0	9.67	22.4
502	503	0.04527	0.0	11.93	0	0.573	6.120	76.7	2.2875	1	273	21.0	9.08	20.6
503	504	0.06076	0.0	11.93	0	0.573	6.976	91.0	2.1675	1	273	21.0	5.64	23.9
504	505	0.10959	0.0	11.93	0	0.573	6.794	89.3	2.3889	1	273	21.0	6.48	22.0
505	506	0.04741	0.0	11.93	0	0.573	6.030	80.8	2.5050	1	273	21.0	7.88	11.9

506 rows × 14 columns

In [4]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 14 columns):
 #   Column      Non-Null Count  Dtype  
---  ------      --------------  -----  
 0   Unnamed: 0  506 non-null    int64  
 1   crim        506 non-null    float64
 2   zn          506 non-null    float64
 3   indus       506 non-null    float64
 4   chas        506 non-null    int64  
 5   nox         506 non-null    float64
 6   rm          506 non-null    float64
 7   age         506 non-null    float64
 8   dis         506 non-null    float64
 9   rad         506 non-null    int64  
 10  tax         506 non-null    int64  
 11  ptratio     506 non-null    float64
 12  lstat       506 non-null    float64
 13  medv        506 non-null    float64
dtypes: float64(10), int64(4)
memory usage: 55.5 KB
```

In [5]:
```
df1 = df.iloc[:,1:]
```

In [6]:
```
df1
```

Out[6]:

	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	lstat	medv
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	5.33	36.2
...	...	...	...	...	...	...	...	...	...	...	...	...	...
501	0.06263	0.0	11.93	0	0.573	6.593	69.1	2.4786	1	273	21.0	9.67	22.4
502	0.04527	0.0	11.93	0	0.573	6.120	76.7	2.2875	1	273	21.0	9.08	20.6
503	0.06076	0.0	11.93	0	0.573	6.976	91.0	2.1675	1	273	21.0	5.64	23.9
504	0.10959	0.0	11.93	0	0.573	6.794	89.3	2.3889	1	273	21.0	6.48	22.0
505	0.04741	0.0	11.93	0	0.573	6.030	80.8	2.5050	1	273	21.0	7.88	11.9

506 rows × 13 columns

In [7]:
```
X1 = df1.drop(['crim'], axis = 1)
Y1 = df1["crim"]
```

In [8]:
```
X1_train,X1_test,Y1_train,Y1_test = sklearn.model_selection.train_test_split( X1, Y1, test_size=0.3, random_state = 10
```

In [9]:
```
scaler = StandardScaler().fit(X1_train)
```

In [71]:
```
lr = LinearRegression()
```

In [72]:
```
pipe = Pipeline(steps = [
  ("std_tf", scaler),
  ("model", lasso)
  ])
pipe

pipe2 = Pipeline(steps = [
  ("std_tf", scaler),
  ("model", ridge)
  ])
pipe2

pipe3 = Pipeline(steps = [
  ("std_tf", scaler),
  ("model", lr)
  ])
pipe3
```

Out[72]:
```
Pipeline(steps=[('std_tf', StandardScaler()), ('model', LinearRegression())])
```

In [79]:
```

alphas = np.logspace(start = -3, stop = 2, base = 10, num = 100)
tuned_parameters = {"model__alpha": alphas}


tuned_parameters2 = {'alpha': alphas}


tuned_parameters3 = {'fit_intercept': [True,False]}
```

In [82]:
```

n_folds=10
mod1 = GridSearchCV(
    pipe,
    tuned_parameters,
    cv = n_folds,
    scoring = 'neg_root_mean_squared_error',

    refit = True
)


n_folds=10
mod2 = GridSearchCV(
    ridge,
    tuned_parameters2,
    cv = n_folds,
    scoring = 'neg_root_mean_squared_error',

    refit = True
)



n_folds=10
mod3 = GridSearchCV(
    lr,
    tuned_parameters3,
    cv = n_folds,
    scoring = 'neg_root_mean_squared_error',

    refit = True
)
```

In [83]:
```

#Lasso
mod1.fit(X1_train, Y1_train)

#Ridge
mod2.fit(X1_train,Y1_train)

#LinearRegression
mod3.fit(X1_train, Y1_train)
```

Out[83]:
```
GridSearchCV(cv=10, estimator=LinearRegression(),
             param_grid={'fit_intercept': [True, False]},
             scoring='neg_root_mean_squared_error')
```

In [104]:
```
lassocv1 = LassoCV(alphas=None, cv=10, max_iter=10000)
lassocv1.fit(scale(X1_train), Y1_train.values.ravel())
```

Out[104]:
```
LassoCV(cv=10, max_iter=10000)
```

In [105]:
```
pd.Series(lasso.coef_, index=X1.columns)
```

Out[105]:
```
zn         0.979760
indus     -0.472106
chas      -0.253493
nox       -0.995798
rm         0.451638
age        0.200572
dis       -1.982333
rad        4.590747
tax       -0.193801
ptratio   -0.552956
lstat      0.276747
medv      -2.161447
dtype: float64
```

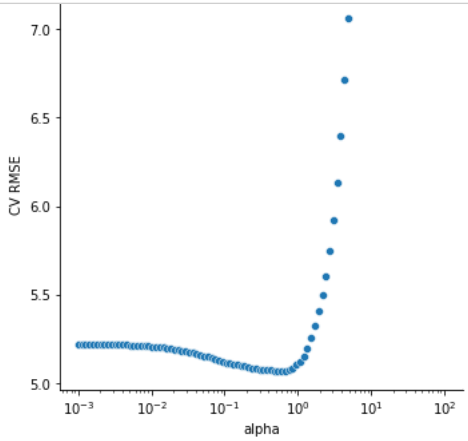
In [106]:
```
ridgecv = RidgeCV(alphas=alphas, scoring='neg_mean_squared_error')
ridgecv.fit(scale(X1_train), Y1_train)
print('Ridge alpha =', ridgecv.alpha_)
```

```
Ridge alpha = 6.892612104349695

/Users/youssefmahmoud/opt/anaconda3/lib/python3.9/site-packages/sklearn/linear_model/_ridge.py:1791: RuntimeWarnin
g: invalid value encountered in reciprocal
  w = ((singvals_sq + alpha) ** -1) - (alpha ** -1)
/Users/youssefmahmoud/opt/anaconda3/lib/python3.9/site-packages/sklearn/linear_model/_ridge.py:1791: RuntimeWarnin
g: invalid value encountered in reciprocal
  w = ((singvals_sq + alpha) ** -1) - (alpha ** -1)
/Users/youssefmahmoud/opt/anaconda3/lib/python3.9/site-packages/sklearn/linear_model/_ridge.py:1791: RuntimeWarnin
g: invalid value encountered in reciprocal
  w = ((singvals_sq + alpha) ** -1) - (alpha ** -1)
/Users/youssefmahmoud/opt/anaconda3/lib/python3.9/site-packages/sklearn/linear_model/_ridge.py:1791: RuntimeWarnin
g: invalid value encountered in reciprocal
  w = ((singvals_sq + alpha) ** -1) - (alpha ** -1)
/Users/youssefmahmoud/opt/anaconda3/lib/python3.9/site-packages/sklearn/linear_model/_ridge.py:1791: RuntimeWarnin
g: invalid value encountered in reciprocal
  w = ((singvals_sq + alpha) ** -1) - (alpha ** -1)
/Users/youssefmahmoud/opt/anaconda3/lib/python3.9/site-packages/sklearn/linear_model/_ridge.py:1791: RuntimeWarnin
g: invalid value encountered in reciprocal
  w = ((singvals_sq + alpha) ** -1) - (alpha ** -1)
```

In [108]:
```
ridge2 = Ridge(alpha=6.892612104349695)
ridge2.set_params(alpha=6.892612104349695)
ridge2.fit(scale(X1_train), Y1_train)
pd.Series(ridge2.coef_.flatten(), index=X1.columns)
```

Out[108]:
```
zn         1.035541
indus     -0.529389
chas      -0.296739
nox       -1.101103
rm         0.527524
age        0.308720
dis       -2.058418
rad        4.568684
tax       -0.223321
ptratio   -0.584646
lstat      0.374966
medv      -2.207996
dtype: float64
```

In [85]:
```
cv_res = pd.DataFrame({
  "alpha": alphas,
  "rmse": -mod1.cv_results_["mean_test_score"]
  })

plt.figure()
sns.relplot(
  data = cv_res,
  x = "alpha",
  y = "rmse"
  ).set(
    xlabel = "alpha",
    ylabel = "CV RMSE",
    xscale = "log"
);
plt.show()
```



In [88]:
```
print("Lasso Best CV Score:" + str(-mod1.best_score_))
print("Ridge Best CV Score:" + str(-mod2.best_score_))
print("LinearRegression Best CV Score:" + str(-mod3.best_score_))
```

```
Lasso Best CV Score:5.068552647540818
Ridge Best CV Score:5.126717575548153
LinearRegression Best CV Score:5.2245807076024375
```

In [90]:
```
print("Lasso Test RMSE: " + str(mean_squared_error(Y1_test, mod1.best_estimator_.predict(X1_test), squared = False)))
print("Ridge Test RMSE: " + str(mean_squared_error(Y1_test, mod2.best_estimator_.predict(X1_test), squared = False)))
print("LinearRegression Test RMSE: " +  str(mean_squared_error(Y1_test, mod3.best_estimator_.predict(X1_test), squared
```

```
Lasso Test RMSE: 7.243122587129746
Ridge Test RMSE: 7.031118161888018
LinearRegression Test RMSE: 7.059508256989245
```

B

In [114]:
```
enetcv = ElasticNetCV(cv=10, max_iter=10000)
enetcv.fit(scale(X1_train), Y1_train.values.ravel())
enet_best = ElasticNet(alpha=enetcv.alpha_)
enet_best.fit(scale(X1_train), Y1_train)
print(list(zip(enet_best.coef_, X1)))
```

```
[(0.9464809301263242, 'zn'), (-0.5060977386580996, 'indus'), (-0.27605012428931563, 'chas'), (-0.9239946467756586, 'n
ox'), (0.49260478958818826, 'rm'), (0.24259618528139576, 'age'), (-1.883123640154616, 'dis'), (4.319574860194326, 'ra
d'), (-0.0, 'tax'), (-0.5085866124691402, 'ptratio'), (0.3890931594901931, 'lstat'), (-2.0533589931578597, 'medv')]
```

In [110]:
```
pipe4 = Pipeline(steps = [
  ("std_tf", scaler),
  ("model", enet_best)
  ])
pipe4
```

Out[110]:
```
Pipeline(steps=[('std_tf', StandardScaler()),
                ('model', ElasticNet(alpha=0.042942897381866435))])
```

In [111]:
```
n_folds=10
mod4 = GridSearchCV(
    pipe4,
    tuned_parameters,
    cv = n_folds,
    scoring = 'neg_root_mean_squared_error',

    refit = True
)
```

In [112]:
```
mod4.fit(X1_train, Y1_train)
```

Out[112]:
```
GridSearchCV(cv=10,
             estimator=Pipeline(steps=[('std_tf', StandardScaler()),
                                       ('model',
                                        ElasticNet(alpha=0.042942897381866435))]),
             param_grid={'model__alpha': array([1.00000000e-03, 1.12332403e-03, 1.26185688e-03, 1.41747416e-03,
       1.59228279e-03, 1.78864953e-03, 2.00923300e-03, 2.25701972e-03,
       2.53536449e-03, 2.84803587e-03, 3.19926714e-03, 3.59381366e-03,
       4.03701726e-...
       6.89261210e+00, 7.74263683e+00, 8.69749003e+00, 9.77009957e+00,
       1.09749877e+01, 1.23284674e+01, 1.38488637e+01, 1.55567614e+01,
       1.74752840e+01, 1.96304065e+01, 2.20513074e+01, 2.47707636e+01,
       2.78255940e+01, 3.12571585e+01, 3.51119173e+01, 3.94420606e+01,
       4.43062146e+01, 4.97702356e+01, 5.59081018e+01, 6.28029144e+01,
       7.05480231e+01, 7.92482898e+01, 8.90215085e+01, 1.00000000e+02])},
             scoring='neg_root_mean_squared_error')
```

In [116]:
```
print("ElasticNet Best CV Score:" + str(-mod4.best_score_))
print("ElasticNet Test RMSE: " + str(mean_squared_error(Y1_test, mod4.best_estimator_.predict(X1_test), squared = False
```

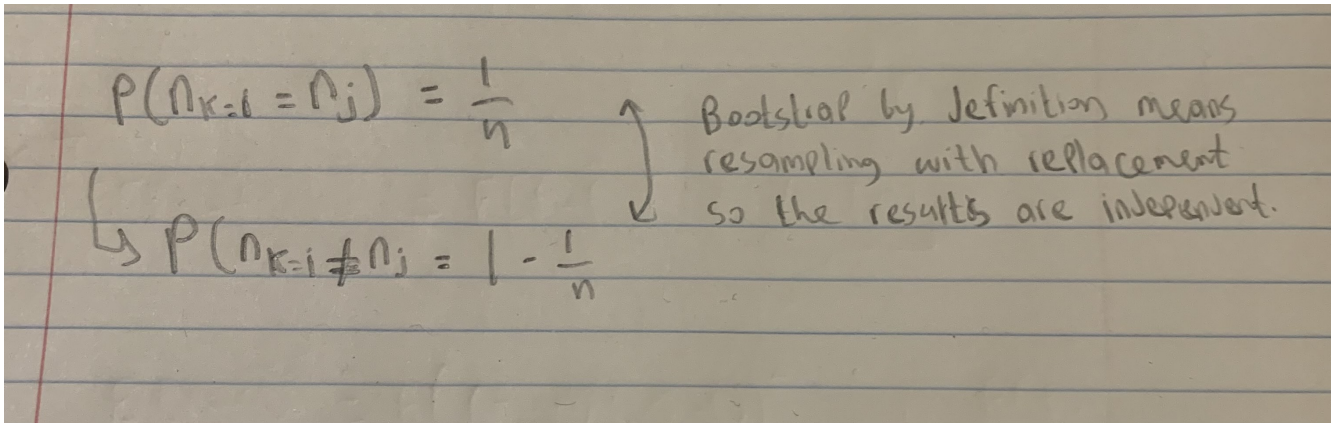
```
ElasticNet Best CV Score:5.075725057532109
ElasticNet Test RMSE: 7.197091738521994
```

C

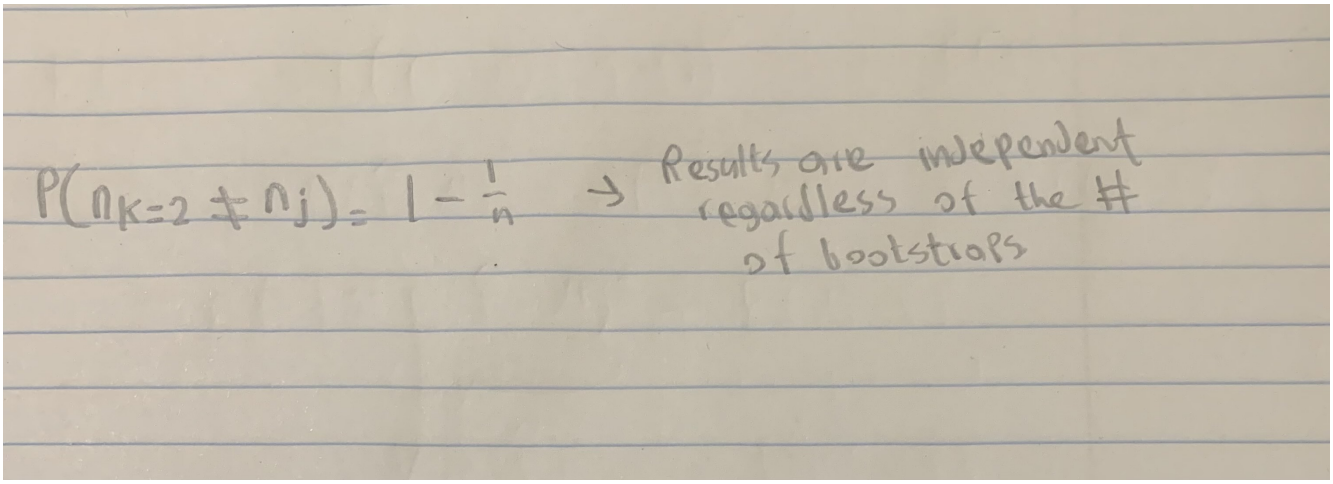
Elastic Net, a combination of ridge and lasso, did not include tax in the features since it shirnked it to 0, implying it does not have an affect on CRIM rate per capita.

5.4.2

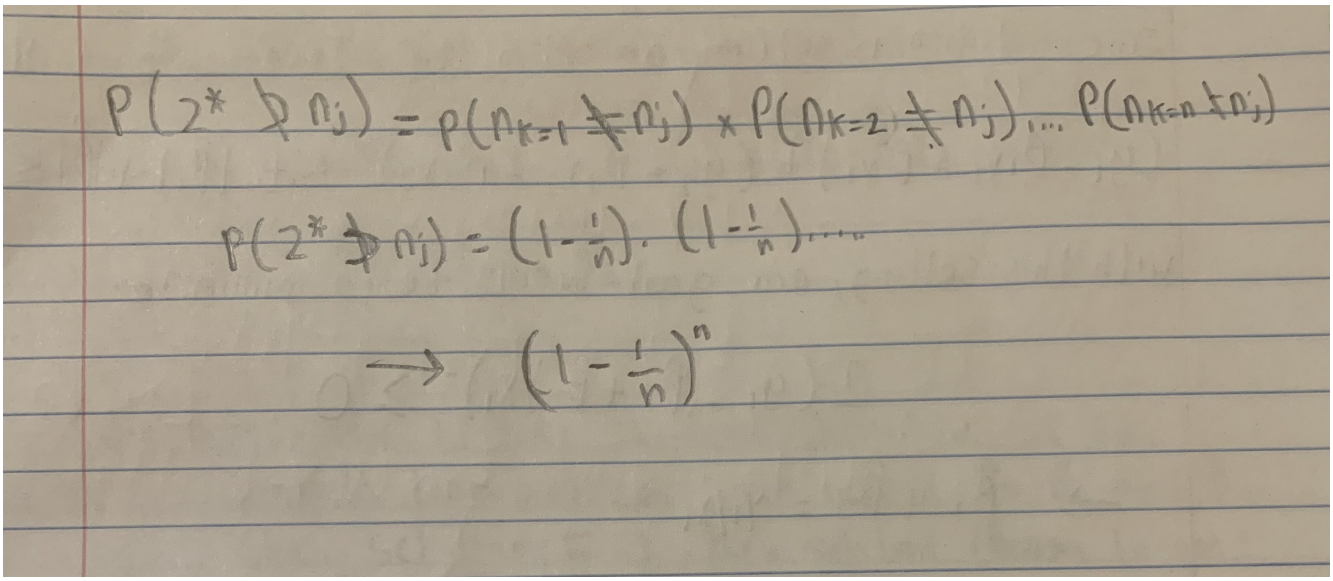
A



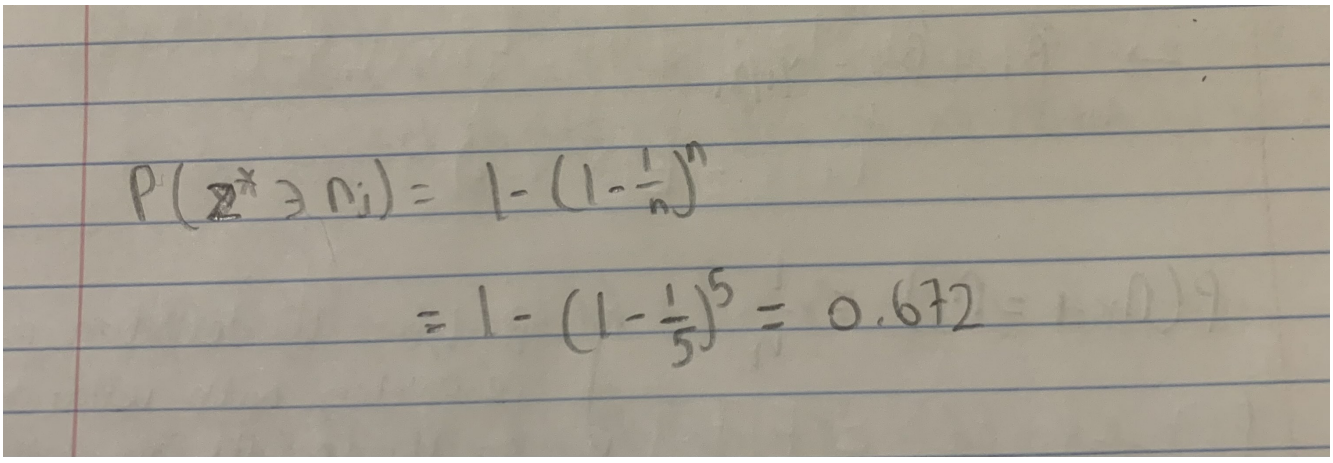
B


$P(n_{k=2} \neq n_j) = 1 - \frac{1}{n}$
→ Results are independent regardless of the # of bootstraps

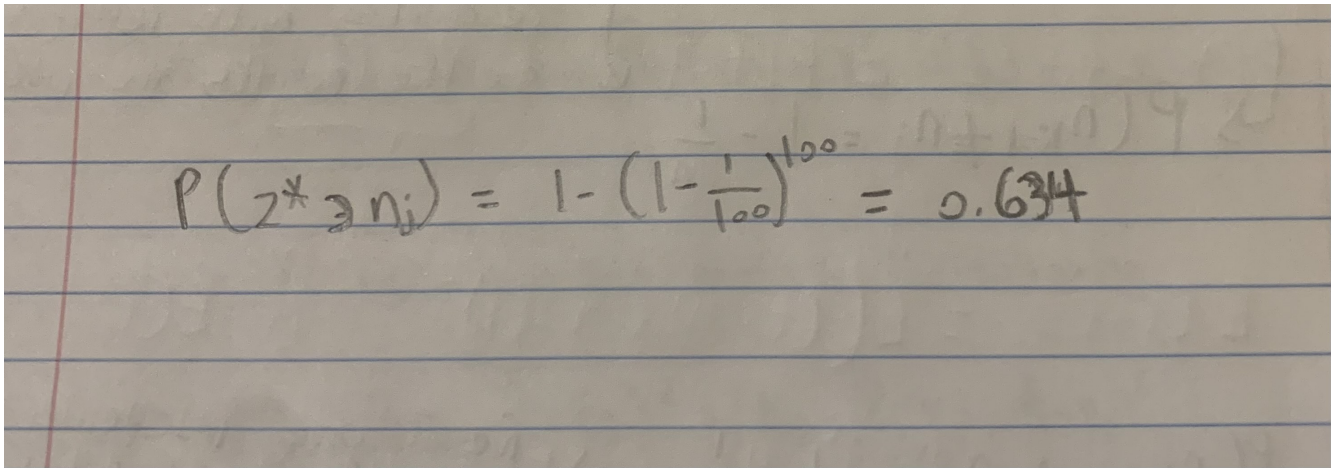
C


$P(2^* \ni n_j) = P(n_{k=1} \neq n_j) \times P(n_{k=2} \neq n_j) \ldots P(n_{k=n} \neq n_j)$
$P(2^* \ni n_j) = (1 - \frac{1}{n}) \cdot (1 - \frac{1}{n}) \ldots$
$\rightarrow (1 - \frac{1}{n})^n$

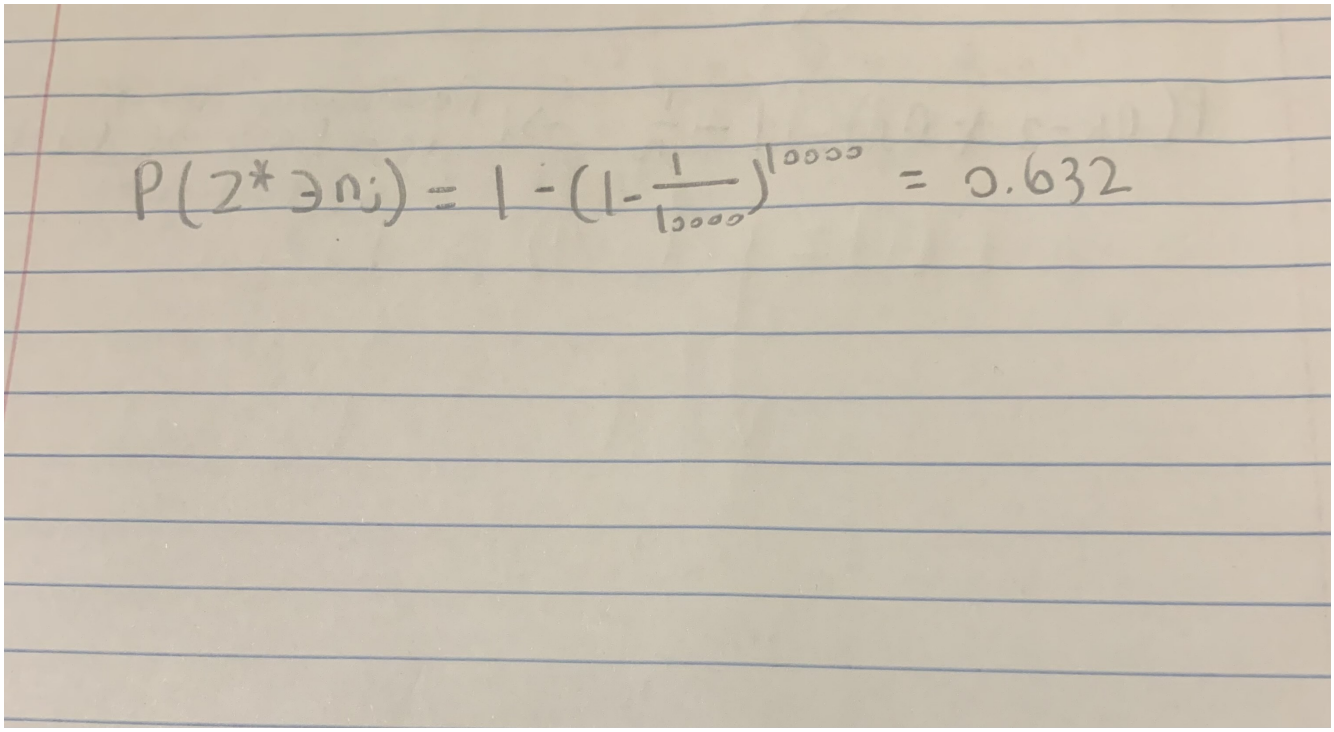
D


$P(2^* \ni n_j) = 1 - (1 - \frac{1}{n})^n$
$= 1 - (1 - \frac{1}{5})^5 = 0.672$

E


$P(2^* \ni n_j) = 1 - (1 - \frac{1}{100})^{100} = 0.634$

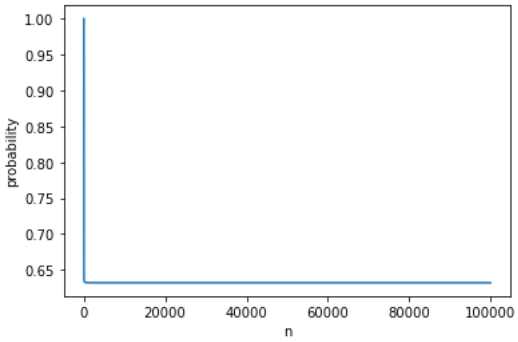
F


$P(2^* \ni n_j) = 1 - (1 - \frac{1}{10000})^{10000} = 0.632$

G

In [61]:
```
def prob_j_in_sample(n):
    return 1 - (1 - 1/n)**n

x = np.arange(1, 100000)
y = np.array([prob_j_in_sample(n) for n in x])

ax = sns.lineplot(x=x, y=prob_j_in_sample(x))
plt.xlabel('n')
plt.ylabel('probability')
```

Out[61]:
```
Text(0, 0.5, 'probability')
```



H

In [62]:
```
store = []
for i in np.arange(1, 10000):
    store += [np.sum((np.random.randint(low=1, high=101, size=100) == 4)) > 0]

np.mean(store)
```

Out[62]:
```
0.6317631763176318
```

5.4.9

A

In [39]:
```
medv_hat = df1['medv'].mean()
medv_hat
```

Out[39]:
```
22.532806324110698
```

B

In [40]:
```
def standard_error_mean(df1):
    medv = np.array(df1)
    SE   = np.std(medv) / np.sqrt(len(medv))
    return SE
standard_error_mean(df1['medv'])
```

Out[40]:
```
0.4084569346972866
```

C

In [43]:
```
def medv_mean_boot(df1, idx):
    Z = np.array(df1.loc[idx])
    return np.mean(Z)

def boot_idx(n):

    return np.random.randint(low=0, high=n, size=n)

def boot(fn, data_df1, samples):

    results = []
    for s in range(samples):
        Z = fn(data_df1, boot_idx(data_df1.shape[0]))
        results += [Z]
    return np.array(results)

B = 1000
medv_mean_boot  = boot(medv_mean_boot, df1['medv'], samples=B)
SE_prediction    = np.std(medv_mean_boot)

print('SE: ' + str(SE_prediction))
```

```
SE: 0.405930477648764
```

The bootstrap method gave us a lower SE.

D

In [45]:
```
medv_hat1   = np.mean(df1['medv'])
lower_bound = medv_hat1 - (2*SE_prediction)
upper_bound  = medv_hat1 + (2*SE_prediction)

pd.Series({'mu': medv_hat1,
           'SE': SE_prediction,
           '0.025': lower_bound ,
           '0.975': upper_bound})
```

Out[45]:
```
mu       22.532806
SE        0.405930
0.025    21.720945
0.975    23.344667
dtype: float64
```

E

In [46]:
```
medv_median_hat = np.median(df1['medv'])
print('median: ' + str(medv_median_hat))
```

```
median: 21.2
```

F

In [58]:
```
def medv_median_boot(df1, idx):
    Z = np.array(df1.loc[idx])
    return np.median(Z)

def boot_idx(n):
    """Return index for bootstrap sample of size n
    e.g. generate array in range 0 to n, with replacement"""
    return np.random.randint(low=0, high=n, size=n)

def boot(fn, data_df1, samples):
    """Perform bootstrap for B number of samples"""
    results = []
    for s in range(samples):
        Z = fn(data_df1, boot_idx(data_df1.shape[0]))
        results += [Z]
    return np.array(results)

B = 1000
medv_boot_median   = boot(medv_median_boot, df1['medv'], samples=B)
SE_prediction1    = np.std(medv_boot_median)

print('SE: ' + str(SE_prediction1))
```

```
SE: 0.38386325690276707
```

The bootstrapped SE of the median is lower than that of the mean.

G

In [49]:
```
tenth_percentile = np.percentile(df1['medv'], 10)
print('tenth_percentile: ' + str(tenth_percentile))
```

```
tenth_percentile: 12.75
```

H

In [59]:
```
def tenth_percentile(df1, idx):
    Z = np.array(df1.loc[idx])
    return np.percentile(Z, 10)

def boot_idx(n):

    return np.random.randint(low=0, high=n, size=n)

def boot(fn, data_df1, samples):

    results = []
    for s in range(samples):
        Z = fn(data_df1, boot_idx(data_df1.shape[0]))
        results += [Z]
    return np.array(results)

B = 1000
tenth_percentile_boot   = boot(tenth_percentile, df1['medv'], samples=B)
SE_prediction2    = np.std(tenth_percentile_boot)

print('SE: ' + str(SE_prediction2))
```

```
SE: 0.4961387885461083
```

The bootstrapped SE of the 10th percentile is the highest among the bootstrapped SEs of the mean and median.