PS4: Gradient descent and regularization

This is a fun but challenging problem set. It will test your python skills, as well as your understanding of the material in class and in the readings. Start early and debug often! Some notes:

- Part 1 is meant to be easy, so get through it quickly.
- Part 2 (especially Question 9) will be difficult, but it is the linchpin of this problem set so make sure to do it well and understand what you've done. If you find your gradient descent algorithm is taking more than a few minutes to complete, debug more, compare notes with others, and go to the TA sessions (especially the sections on vectorized computation and computational efficiency).
- Depending on how well you've implemented Question 9's coding portion, Questions 10, 11, and 16 will be relatively painless or incredibly painful.
- Part 4 (especially Questions 16 and 17) will be computationally intensive. Don't leave this until the last minute, otherwise your code might be running when the deadline arrives.
- Do the extra credit problems last.

Part 0

Question 1 - upload your ipynb to bcourses

Question 2 - upload your pdf to bcourses

Question 3 - please provide a summary of any resources consulted, and people with whom you worked in the completion of the problem set.

Highlight any specific LLM prompts/resources used at the point of use.

your answer here

Question 4 - answer the quiz question about which submission option you prefer

---

Introduction to the assignment

As with the last assignment, you will be using a modified version of the California Housing Prices Dataset. Please download the csv file from bCourses (`'cal_housing_data_clean_ps4.csv'`).

To perform any randomized operation, only use functions in the `numpy` library (`np.random`). Do not use other packages for random functions.

In [2]:
```
import IPython
import numpy as np
import scipy as sp
import pandas as pd
import matplotlib
import sklearn

%matplotlib inline
import matplotlib.pyplot as plt
import statsmodels.api as sm
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import statsmodels.formula.api as smf

from sklearn.linear_model import Ridge
```

In [5]:
```
# Load the California Housing Dataset
cal_df = pd.read_csv('cal_housing_data_clean_ps4.csv')

# leave the following line untouched, it will help ensure that your "random"
np.random.seed(seed=94611)
```

In [3]:
```
cal_df.head()
```

Out[3]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	DistCoast	Inlar
0	3.6885	49	5.184569	1.052950	2363	3.574887	4205.460788	
1	3.1630	26	4.267241	0.961207	719	3.099138	28060.624020	
2	2.8042	35	3.895018	1.080071	1193	2.122776	20913.168450	
3	4.2305	32	5.891775	1.235931	2541	5.500000	21878.282810	
4	4.7663	38	5.566038	1.015094	827	3.120755	16863.074990	

Part 1: Getting oriented

1.1 Use existing libraries

Soon, you will write your own gradient descent algorithm, which you will then use to minimize the squared error cost function. First, however, let's use the canned versions that come with Python, to make sure we understand what we're aiming to achieve.

Use the Linear Regression class from sklearn or the OLS class from SciPy to explore the relationship between median housing value and median income in California's census block groups.

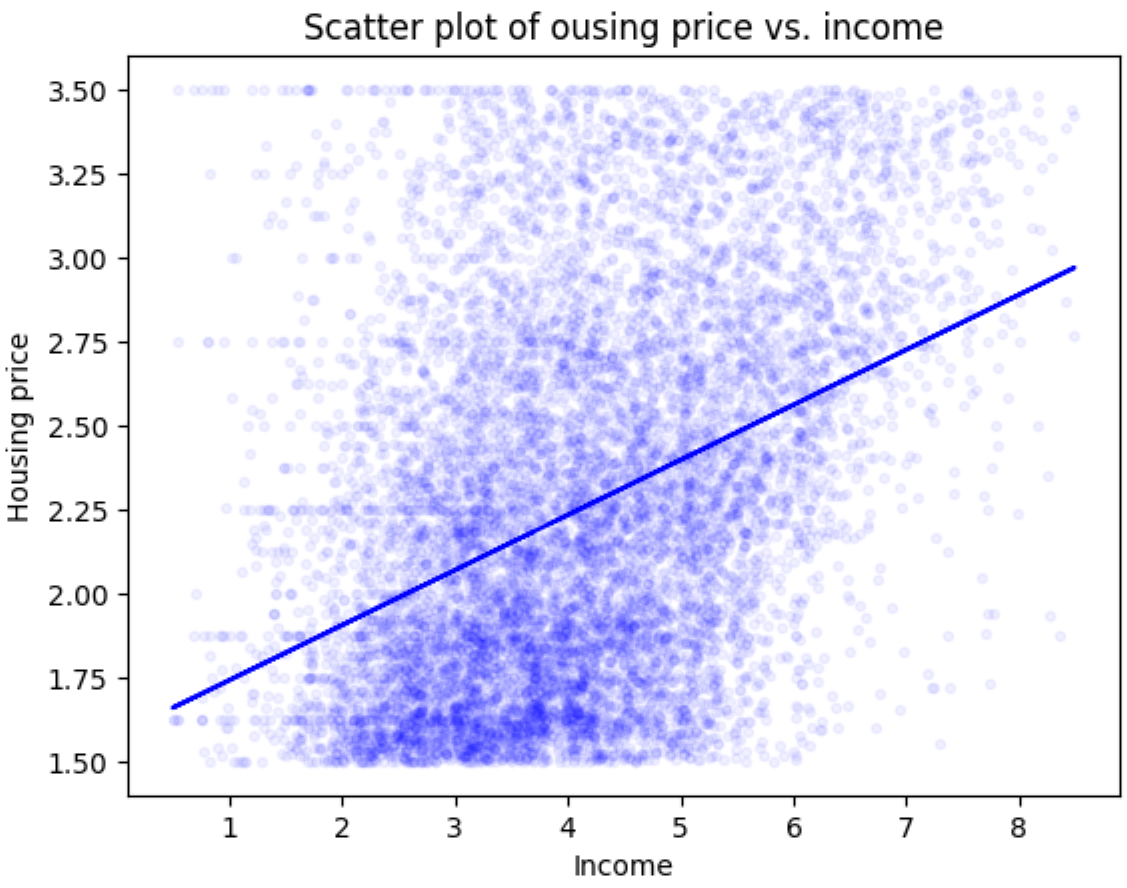
Question 5

Regress the median housing value `MedHouseVal` on the median income `MedInc` and a constant. Report the coefficients and R². Draw a scatter plot of housing price (y-axis) against income (x-axis), and draw the regression line in blue. You might want to make the dots semi-transparent if it improves the presentation of the figure.

In [95]:
```
X = np.array(cal_df['MedInc']).reshape(-1, 1)
y = np.array(cal_df['MedHouseVal'])

reg = LinearRegression(fit_intercept=True).fit(X, y)
R_2 = reg.score(X, y)
print(f"R squared is: {R_2}")
print(f"Coefficient: {reg.coef_[0]}")
print(f"Intercept: {reg.intercept_}")

fig, ax = plt.subplots(1)
ax.scatter(X, y, color='blue', s=10, alpha=0.05)
ax.plot(X, reg.predict(X), color='blue')
ax.set_xlabel('Income')
ax.set_ylabel('Housing price')
ax.set_title('Scatter plot of ousing price vs. income')
```

```
R squared is: 0.16063166767339787
Coefficient: 0.16400595617582836
Intercept: 1.5772617051685813
```

Out[95]:
```
Text(0.5, 1.0, 'Scatter plot of ousing price vs. income')
```



your answers here

Question 6

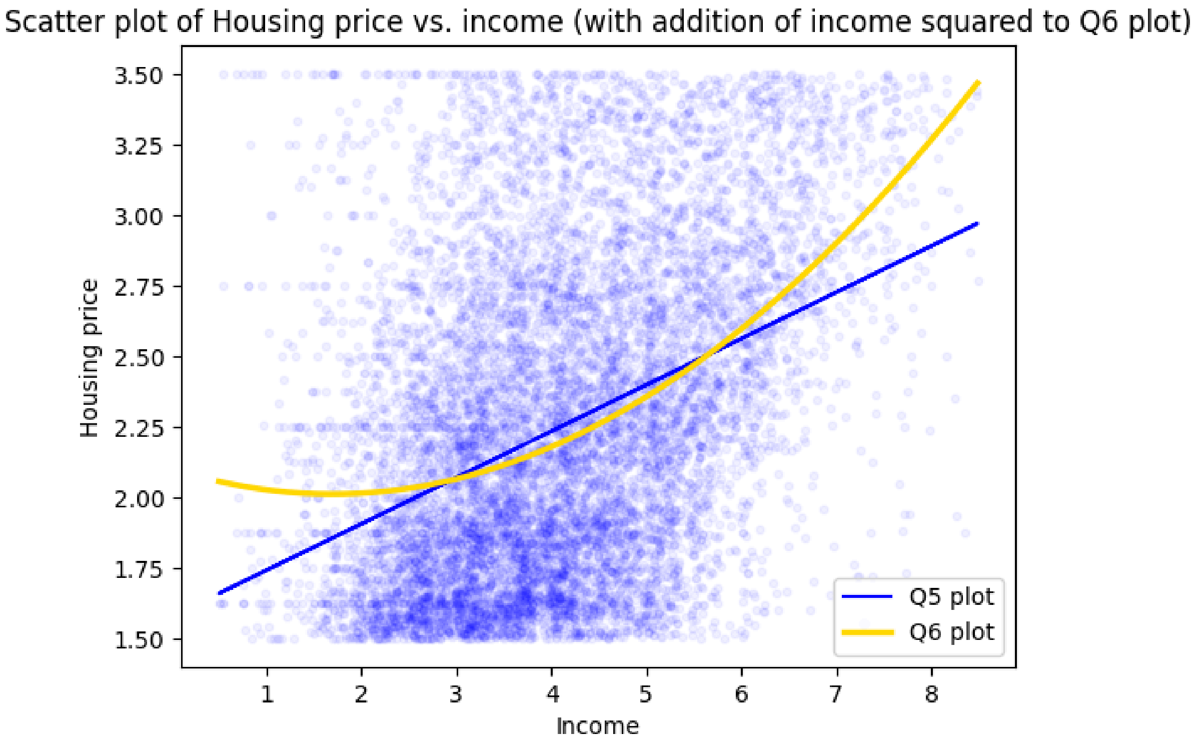
Regress the median housing value on median income, median income squared, and a constant. Report the coefficients and R². Plot the data and linear prediction line from Question 1 in blue, and then plot this new (curved) regression line in gold, on the same axes.

In [71]:
```
X = np.array(cal_df['MedInc']).reshape(-1, 1)

X2 = X**2
X_income = np.hstack((X, X2))
y = np.array(cal_df['MedHouseVal'])

reg_q6 = LinearRegression(fit_intercept=True).fit(X_income, y)
R_2_q6 = reg_q6.score(X_income, y)
print(f"R squared is: {R_2_q6}")
print(f"Coefficient: {reg_q6.coef_}")
print(f"Intercept: {reg_q6.intercept_}")

fig, ax = plt.subplots(1)
ax.scatter(X, y, color='blue', s=10, alpha=0.05)
ax.plot(X, reg.predict(X), color='blue', label='Q5 plot')

X_sorted = np.sort(X, axis=0) #to avoide zigzag line: before plotting we sor
```

```
ax.plot(X_sorted, reg_q6.predict(np.hstack((X_sorted, X_sorted**2))), color=
ax.set_xlabel('Income')
ax.set_ylabel('Housing price')
ax.set_title('Scatter plot of Housing price vs. income (with addition of inc
ax.legend()
```

```
R squared is: 0.1787757868918639
Coefficient: [-0.10678359  0.03153349]
Intercept: 2.101009985286182
```

```
Out[71]: <matplotlib.legend.Legend at 0x76517e53d6f0>
```



your answer here

Question 7

Intepret your results from Questions 5 and 6.

your answer here

1.2 Training and testing

Chances are, for the above problem you used all of your data to fit the regression line. In some circumstances this is a reasonable thing to do, but if your primary objective is prediction, you should be careful about overfitting. Let's redo the above results the ML way, using careful cross-validation. Since you are now experts in cross-validation, and have written your own cross-validation algorithm from scratch, you can now take a shortcut and use the libraries that others have built for you.

Question 8

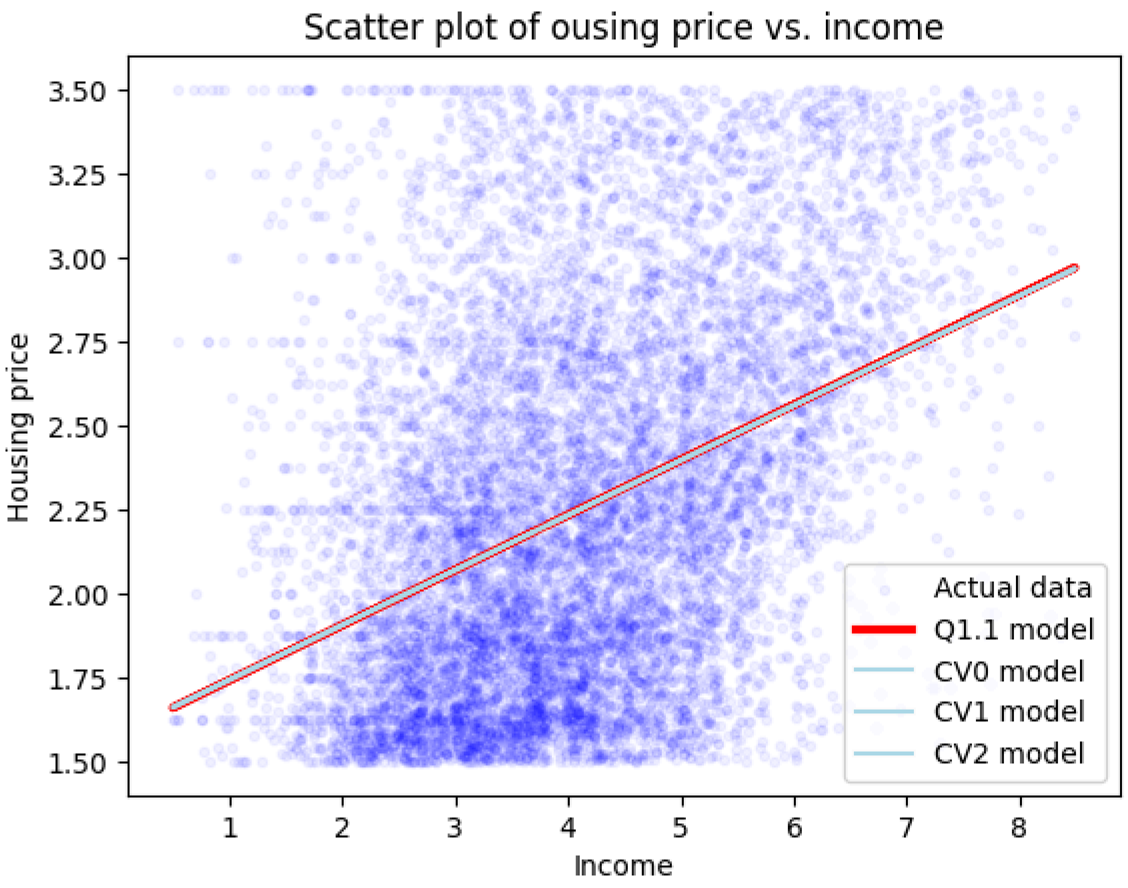
Using the cross-validation functions from scikit-learn, use 3-fold cross-validation to fit the regression model (a) from 1.1, i.e. the linear fit of median housing value on median income. Each fold of cross-validation will give you one slope coefficient and one intercept coefficient. Create a new scatterplot of housing price against rooms, and draw the three different regression lines in light blue, and the original regression line from 1.1 in red (which was estimated using the full dataset). What do you notice?

In [96]:
```
from sklearn.model_selection import KFold

kf = KFold(n_splits=3, shuffle=True, random_state=42)
```

In [111…
```
X = np.array(cal_df['MedInc']).reshape(-1, 1)
y = np.array(cal_df['MedHouseVal'])

intercepts = []
coefs = []
regressions = []

linreg = LinearRegression(fit_intercept=True)
for train_id, test_id in kf.split(X):
    X_train, X_test = X[train_id], X[test_id]
    y_train, y_test = y[train_id], y[test_id]

    linreg.fit(X_train, y_train)

    regressions.append(linreg)
    intercepts.append(linreg.intercept_)
    coefs.append(linreg.coef_)
print(coefs)
print(intercepts)

fig, ax = plt.subplots(1)
ax.scatter(X, y, color='blue', s=10, alpha=0.05, label='Actual data')
ax.plot(X, reg.predict(X), color='red', lw=3, label='Q1.1 model')
for i in range(len(coefs)):
    ax.plot(X, regressions[i].predict(X), color='lightblue', lw=1.5, label=f
    # ax.plot(X, coefs[i]*X+intercepts[i])

ax.set_xlabel('Income')
ax.set_ylabel('Housing price')
ax.set_title('Scatter plot of ousing price vs. income')
ax.legend()
```

```
[array([0.16538021]), array([0.16313235]), array([0.16348076])]
[np.float64(1.5721739070853897), np.float64(1.5808431238118916), np.float64
(1.5788526466142028)]
```

Out[111…
```
<matplotlib.legend.Legend at 0x76517c7ce470>
```



your answer here

Part 2: Gradient descent: Linear Regression

This is where it gets fun!

2.0 Data normalization (done for you!)

For all the following questions, unless explicitly asked otherwise, you are expected to standardize appropriately. Recall that in settings where you are using holdout data for validation or testing purposes, this involves substracting the average and dividing by the standard deviation of your training data.

In [6]:
```
'''
Function
--------
standardize
    Column-wise standardization of a target dataframe using the mean and std

Parameters
----------
ref,tar : pd.DataFrame
    ref: reference dataframe
    tar: target dataframe

Returns
-------
tar_norm: pd.DataFrame
    Standardized target dataframe
'''
def standardize(ref,tar):
    tar_norm = ((tar - np.mean(ref, axis = 0)) / np.std(ref, axis = 0))
    return tar_norm

# Examples
# Standardize train: standardize(ref=x_train,tar=x_train)
# Standardize test: standardize(ref=x_train,tar=x_test)
```

2.1 Implement gradient descent with one independent variable (median income)

Question 9

Implement the batch gradient descent algorithm that we discussed in class. Use the version you implement to regress the median house value on the median income.

Experiment with 3 different values of the learning rate R(0.001, 0.01, 0.05), and do the following:

- Report the values of alpha and beta that minimize the loss function
- Report the number of iterations it takes for your algorithm to converge (for each value of R)
- Report the total running time of your algorithm, in seconds
- How do your coefficients compare to the ones estimated through standard libraries in 1.1? Does this depend on R?

Some skeleton code is provided below, but you should feel free to delete this code and start from scratch if you prefer.

- *Hint 1: Don't forget to implement a stopping condition, so that at every iteration you check whether your results have converged. Common approaches to this are to (a) check to see if the loss has stopped decreasing; and (b) check if both your current parameter esimates are close to the estimates from the previous iteration. In both cases, "close" should not be ==0, it should be <=epsilon, where epsilon is something very small (like 0.0001).*
- *Hint 2: We recommend including a MaxIterations parameter in their gradient descent algorithm, to make sure things don't go off the rails, i.e., as a safeguard in case your algorithm isn't converging as it should.*

In [7]:
```
import time

"""
Function
--------
bivariate_ols
    Gradient Decent to minimize OLS. Used to find coefficients of bivariate

Parameters
----------
xvalues, yvalues : narray
    xvalues: independent variable
    yvalues: dependent variable
```

```
R: float
    Learning rate

MaxIterations: Int
    maximum number of iterations


Returns
-------
alpha: float
    intercept

beta: float
    coefficient
n: number of features
m: number of sampples
"""
def bivariate_ols(xvalues, yvalues, R=0.01, MaxIterations=1000):
    # initialize the parameters
    m, n  = xvalues.shape
    beta = np.random.rand(n, 1)
    alpha  = np.zeros((1, 1))
    eps   = 1e-4
    loss  = []

    start_time = time.time()

    for iters in range(MaxIterations):
        y_hat = xvalues @ beta + alpha

        d_beta = (1/m) * xvalues.T @ (y_hat - yvalues)
        d_alpha  = (1/m) * np.sum(y_hat - yvalues)

        beta  -= R*d_beta
        alpha -= R*d_alpha

        ls = np.mean((y_hat-yvalues)**2)
        loss.append(ls)
        # if iter % 50 == 0: print(f"iter: {iter} >> loss: {ls}")

        if np.abs(d_alpha) < eps and np.all(np.abs(d_beta) < eps):
            print(f"Converged in {iters} iterations")
            print("Time taken: {:.2f} seconds".format(time.time() - start_ti
            return alpha, beta, iters

    print("Time taken: {:.2f} seconds".format(time.time() - start_time))
    return alpha, beta, iters
```

In [9]:
```
MaxIterations=50000

X = np.array(cal_df['MedInc']).reshape(-1, 1)
y = np.array(cal_df['MedHouseVal']).reshape(-1, 1)
X_train = standardize(X, X)
y_train = y
# print(X.shape, y.shape)
```

```
for R in [0.001, 0.01, 0.05]:
    alpha, beta, iters = bivariate_ols(X, y, R, MaxIterations)
    print(f"for learning rate: {R} >> alpha is {alpha}, beta is {beta}, and
#     plt.plot(loss, label=f"LR={R}")

# plt.xlabel("Iterations")
# plt.ylabel("Loss (MSE)")
# # plt.title("")
# plt.legend()
# plt.show()
```

```
Time taken: 8.29 seconds
for learning rate: 0.001 >> alpha is [[1.56223294]], beta is [[0.16735267]],
and iteration number is 49999
Converged in 7873 iterations
Time taken: 1.21 seconds
for learning rate: 0.01 >> alpha is [[1.57617844]], beta is [[0.16424719]],
and iteration number is 7873
Converged in 1580 iterations
Time taken: 0.25 seconds
for learning rate: 0.05 >> alpha is [[1.57618286]], beta is [[0.1642462]], a
nd iteration number is 1580
```

2.2 Implement gradient descent with an arbitrary number of independent variables

Question 10

Now that you have a simple version of gradient descent working, create a version of gradient descent that can take more than one independent variable. Assume all independent variables will be continuous. Test your algorithm using `MedInc`, `HouseAge`, and `AveRooms` as independent variables. Remember to standardize appropriately before inputting them to the gradient descent algorithm. How do your coefficients compare to the ones estimated through standard libraries?

As before, report and interpret your estimated coefficients, the number of iterations before convergence, and the total running time of your algorithm. Experiment with three values of R (0.1, 0.01, and 0.05).

- *Hint 1: Be careful to implement this efficiently, otherwise it might take a long time for your code to run. Commands like `np.dot` can be a good friend to you on this problem*

In [3]:
```
cal_df = pd.read_csv('cal_housing_data_clean_ps4.csv')
```

In [10]:
```
MaxIterations=10000
```

```
X = np.array(cal_df[['MedInc', 'HouseAge', 'AveRooms']])#.reshape(-1, 1)
y = np.array(cal_df['MedHouseVal']).reshape(-1, 1)
X_train = standardize(X, X)
print(X.shape, y.shape)
print("any NaNs in X:", np.isnan(X_train).sum())
print("any NaNs in y:", np.isnan(y_train).sum())
print("any Inf in X:", np.isinf(X_train).sum())
print("any Inf in y:", np.isinf(y_train).sum())

for R in [0.1, 0.01, 0.05]:
    alpha, beta, iters = bivariate_ols(X_train, y, R, MaxIterations)
    print(f"for learning rate: {R} >> alpha is {alpha}, beta is {beta.T}, ar
#     plt.plot(loss, label=f"LR={R}")

# plt.xlabel("Iterations")
# plt.ylabel("Loss (MSE)")
# # plt.title("")
# plt.legend()
# plt.show()
```

```
(10484, 3) (10484, 1)
any NaNs in X: 0
any NaNs in y: 0
any Inf in X: 0
any Inf in y: 0
Converged in 126 iterations
Time taken: 0.01 seconds
for learning rate: 0.1 >> alpha is [[2.245869]], beta is [[ 0.25463676  0.08
677189 -0.03098932]], and iteration number is 126
Converged in 1105 iterations
Time taken: 0.07 seconds
for learning rate: 0.01 >> alpha is [[2.24583906]], beta is [[ 0.25440852
0.08677862 -0.03073616]], and iteration number is 1105
Converged in 226 iterations
Time taken: 0.01 seconds
for learning rate: 0.05 >> alpha is [[2.24585276]], beta is [[ 0.25432928
0.08668795 -0.03072745]], and iteration number is 226
```

In [ ]:

your answer here

2.3 Implement mini-batch gradient descent

Question 11

Now, let's extend our code to implement mini-batch gradient descent. From the lectures, recall that in mini-batch gradient descent, parameters are updated (in each epoch / iteration) after computing gradients of the error with respect to a *subset of the training set*.

In order to implement mini-batch gradient descent, first implement the function *create_mini_batches*, which splits some given data into batches of size *batch_size*.

Then, take your code from Question 10, and make the necessary modifications to implement mini-batch gradient descent. To be specific, you need to make two sets of changes:

1. For each epoch / iteration, you will need to split your training data into batches (using the create_mini_batches function). You will need to calculate the cost / error, the gradients, and implement the parameter updates for each batch.
2. You need to modify your stopping condition -- think carefully about this!

Write out what the change to the stopping condition needs to be. Report and interpret your estimated coefficients, the number of iterations before convergence, and the total running time of your algorithm. For this experiment, let R = 0.05).

In [11]:
```
def create_mini_batches(X, y, batch_size):
    dataSet = np.hstack((X, y))
    np.random.shuffle(dataSet)
    m = X.shape[0]

    miniBatches = []

    for i in range(0, m, batch_size):
        X_batch = dataSet[i:i+batch_size, :-1]
        y_batch = dataSet[i:i+batch_size, -1:]
        miniBatches.append((X_batch, y_batch))

    return miniBatches
```

In [12]:
```
def MBGD(xvalues, yvalues, R=0.01, batch_size=20, MaxIterations=1000, eps=1e
    # initialize the parameters
    m, n  = xvalues.shape
    beta = np.random.rand(n, 1)
    alpha  = np.zeros((1, 1))

    loss  = []

    start_time = time.time()

    for iters in range(MaxIterations):
        mini_batches = create_mini_batches(xvalues, yvalues, batch_size)
        d_beta_m  = np.zeros_like(beta)
        d_alpha_m = np.zeros((1, 1))

        for X_batch, y_batch in mini_batches:
            y_hat = X_batch @ beta + alpha

            d_beta = (1/batch_size) * X_batch.T @ (y_hat - y_batch)
            d_alpha  = (1/batch_size) * np.sum(y_hat - y_batch)

            beta  -= R*d_beta
            alpha -= R*d_alpha

            d_beta_m += d_beta/len(mini_batches)
            d_alpha_m += d_alpha/len(mini_batches)
            # print(d_alpha_m.shape)

            # ls = np.mean((X_batch @ beta + alpha -y_batch)**2)
            # loss.append(ls)
            # if iter % 50 == 0: print(f"iter: {iter} >> loss: {ls}")

            if np.linalg.norm(d_beta_m) < eps and abs(d_alpha_m) < eps:
                print(f"Converged in {iters} iterations")
                print("Time taken: {:.2f} seconds".format(time.time() - star
                return alpha, beta, iters

    print("Time taken: {:.2f} seconds".format(time.time() - start_time))
    return alpha, beta, iters
```

In [14]:
```
print('Experiment 1:')
batch_siz = 20
R=0.05
MaxIterations=100000
eps=1e-4

alpha, beta, iters = MBGD(X_train, y_train, R, batch_siz, MaxIterations, eps
print(f"for learning rate: {R} >> alpha is {alpha}, beta is {beta.T}, and it
```

```
Experiment 1:
Converged in 22 iterations
Time taken: 0.47 seconds
for learning rate: 0.05 >> alpha is [[2.21280749]], beta is [[0.23921492 0.0
8745226 0.00338775]], and iteration number is 22
```

your answer here

Part 3: Prediction

Let's use our fitted model to make predictions about housing prices.

3.1 Cross-Validation

Question 12

Unless you were careful above, you probably overfit your data again. Let's fix that. Use 5-fold cross-validation to re-fit the multivariate regression from 2.3 above, and report your estimated coefficients (there should be four, corresponding to the intercept and the three coefficients for `MedInc`, `AveRoomsNorm`, and `HouseAgeNorm`). Since there are 5 folds, there will be 5 sets of four coefficients -- report them all in a 5x4 table.

Note: You can use KFold to perform the cross-validation.

In [18]:
```
from sklearn.model_selection import KFold
```

In [20]:
```
def cross_val(xvalues, yvalues, R, batch_size, MaxIterations, eps, k=5):
    kf = KFold(n_splits=k, shuffle=True, random_state=42)

    err_t = []
    alphas = []
    betas = []

    for X_index, y_index in kf.split(xvalues):
        X_train, X_test = xvalues[X_index], xvalues[y_index]
        y_train, y_test = yvalues[X_index], yvalues[y_index]

        alpha, beta, _ = MBGD(X_train, y_train, R, batch_size, MaxIterations

        y_hat = X_test @ beta + alpha

        rsme = np.sqrt(mean_squared_error(y_test, y_hat))

        err_t.append(rsme)
        alphas.append(alpha)
        betas.append(beta)

    print(f" Mean RSME across all folds is: {np.mean(err_t)}")

    return alphas, betas, err_t
```

In [21]:
```
X = np.array(cal_df[['MedInc', 'HouseAge', 'AveRooms']])#.reshape(-1, 1)
y = np.array(cal_df['MedHouseVal']).reshape(-1, 1)
X_train = standardize(X, X)

batch_size = 20
R=0.05
MaxIterations=100000
eps=1e-4
k=5

alphas, betas, err_t = cross_val(X_train, y, R, batch_size, MaxIterations, e
```

```
Converged in 34 iterations
Time taken: 0.56 seconds
Converged in 6 iterations
Time taken: 0.10 seconds
Converged in 41 iterations
Time taken: 0.66 seconds
Converged in 311 iterations
Time taken: 5.03 seconds
Converged in 35 iterations
Time taken: 0.59 seconds
 Mean RSME across all folds is: 0.49268074358955777
```

In [170…
```
Table12 = pd.DataFrame({"RMSE": err_t, "alpha": [a[0][0] for a in alphas], "
    "HouseAge": [b[1][0] for b in betas], "AveRooms": [b[2][0] for b in beta

mean_row = Table12.mean(numeric_only=True)
```

```
Table12.loc["Mean"] = mean_row

Table12.index = [f"Fold {i+1}" for i in range(len(Table12))]
Table12
```

Out[170…

	RMSE	alpha	MedInc	HouseAge	AveRooms
Fold 1	0.507006	2.247212	0.252530	0.115456	-0.015473
Fold 2	0.491011	2.235001	0.262868	0.110693	-0.039710
Fold 3	0.480452	2.227409	0.275793	0.067447	-0.050133
Fold 4	0.497460	2.251209	0.256995	0.053062	-0.056503
Fold 5	0.492560	2.209979	0.265303	0.108222	0.017676
Fold 6	0.493698	2.234162	0.262698	0.090976	-0.028829

your answer here

3.2 Predicted values and RMSE

Question 13

Let's figure out how accurate this predictive model turned out to be. Compute the cross-validated RMSE for each of the 5 folds above. In other words, in fold 1, use the parameters estimated on the 80% of the data to make predictions for the 20%, and calculate the RMSE for those 20%. Repeat this for the remaining folds. Report the RMSE for each of the 5 folds, and the average (mean) RMSE across all 5 folds. How does this average RMSE compare to the performance of your nearest neighbor algorithm from the last problem set?

In [ ]:
```
#the code for this part was integrated to the code done for Question 12. The
```

your answer here

Part 4: Regularization

4.1 Get prepped

Question 14

Step 1: Generate features consisting of all polynomial combinations of degree greater than 0 and less than or equal to 3 of the following features: `MedInc`, `HouseAge`, and `AveRooms`. If you are using `PolynomialFeatures` from `sklearn.preprocessing` make sure you drop the constant polynomial feature (degree 0). You should have a total of 19 polynomial features. Display a portion of your matrix, and print the shape of the matrix.

Step 2: Randomly sample 80% of your data and call this the training set, and set aside the remaining 20% as your test set.

In [174…

```
from sklearn.preprocessing import PolynomialFeatures

# leave the following line untouched, it will help ensure that your "random"
# resetting the seed here means that the number of times you called numpy's
np.random.seed(seed=94611)

X = np.array(cal_df[['MedInc', 'HouseAge', 'AveRooms']])#.reshape(-1, 1)
y = np.array(cal_df['MedHouseVal']).reshape(-1, 1)


model_polynom = PolynomialFeatures(degree=3, include_bias=False)
X_poly = model_polynom.fit_transform(X)

feats_polynom = model_polynom.get_feature_names_out(input_features=['MedInc'
df_poly = pd.DataFrame(X_poly, columns=feats_polynom)
print("Shape of the polynomial features matrix:", df_poly.shape)
df_poly.head()
```

```
Shape of the polynomial features matrix: (10484, 19)
```

Out[174…

	MedInc	HouseAge	AveRooms	MedInc^2	MedInc HouseAge	MedInc AveRooms	HouseAge^2	House AveRo
0	3.6885	49.0	5.184569	13.605032	180.7365	19.123282	2401.0	254.04
1	3.1630	26.0	4.267241	10.004569	82.2380	13.497284	676.0	110.94
2	2.8042	35.0	3.895018	7.863538	98.1470	10.922409	1225.0	136.32
3	4.2305	32.0	5.891775	17.897130	135.3760	24.925154	1024.0	188.53
4	4.7663	38.0	5.566038	22.717616	181.1194	26.529406	1444.0	211.50

In [175…

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X_poly, y, test_size=0.2

print("Shape of X_train:", X_train.shape)
print("Shape of X_test:", X_test.shape)
print("Shape of y_train:", y_train.shape)
print("Shape of y_test:", y_test.shape)
```

```
Shape of X_train: (8387, 19)
Shape of X_test: (2097, 19)
Shape of y_train: (8387, 1)
Shape of y_test: (2097, 1)
```

In [176…

```
df_poly
```

Out[176…

	MedInc	HouseAge	AveRooms	MedInc^2	MedInc HouseAge	MedInc AveRooms	HouseAge^2	H A
0	3.6885	49.0	5.184569	13.605032	180.7365	19.123282	2401.0	25
1	3.1630	26.0	4.267241	10.004569	82.2380	13.497284	676.0	11
2	2.8042	35.0	3.895018	7.863538	98.1470	10.922409	1225.0	13
3	4.2305	32.0	5.891775	17.897130	135.3760	24.925154	1024.0	18
4	4.7663	38.0	5.566038	22.717616	181.1194	26.529406	1444.0	21
...	...	...	...	...	...	...	...	
10479	3.4453	37.0	4.834667	11.870092	127.4761	16.656877	1369.0	17
10480	6.1504	19.0	5.710452	37.827420	116.8576	35.121564	361.0	10
10481	5.0526	36.0	6.088496	25.528767	181.8936	30.762733	1296.0	21
10482	5.0081	13.0	4.717890	25.081066	65.1053	23.627664	169.0	6
10483	3.8426	16.0	4.706941	14.765575	61.4816	18.086891	256.0	7

10484 rows × 19 columns

4.2 Complexity and overfitting?

Question 15

Now, using your version of multivariate regression from 2.3, let's try to build a more complex model. **Remember to standardize appropriately!** Using the training set, regress the median house value on the polynomial features using your multivariate ols algorithm. Calculate train and test RMSE. Is this the result that you were expecting? How do these numbers compare to each other, and to the RMSE from 3.2 and nearest neighbors?

In [177…

```
X_train_st = standardize(X_train, X_train)
X_test_st  = standardize(X_train, X_test)
# X_train_st = X_train_st.to_numpy()
# X_test_st  = X_test_st.to_numpy()

alpha, beta, _ = MBGD(X_train_st, y_train, R=0.001, batch_size=20, MaxIterat

y_train_pred = X_train_st @ beta + alpha
y_test_pred = X_test_st @ beta + alpha

# RMSE Calculation
train_rmse = np.sqrt(mean_squared_error(y_train, y_train_pred))
test_rmse = np.sqrt(mean_squared_error(y_test, y_test_pred))

print(f"Train RMSE: {train_rmse}")
print(f"Test RMSE: {test_rmse}")
```

```
Converged in 1230 iterations
Time taken: 20.35 seconds
Train RMSE: 0.47889933902113196
Test RMSE: 0.49097613296161957
```

In [178…

```
print(alpha,beta,_)
```

```
[[2.24766736]] [[-0.27000093]
 [-0.20956942]
 [ 0.01344352]
 [ 0.53630596]
 [ 0.45803308]
 [-0.37143965]
 [-0.03769479]
 [-0.13687314]
 [ 0.31736774]
 [-0.09942099]
 [-0.22390783]
 [ 0.25676086]
 [-0.17150504]
 [ 0.21124886]
 [-0.06213372]
 [ 0.23619518]
 [-0.00729521]
 [-0.10788198]
 [-0.09901244]] 1230
```

4.3 Ridge regularization (basic)

Question 16

Incorporate L2 (Ridge) regularization into your multivariate_ols regression. Write a new version of your gradient descent algorithm that includes a regularization term "lambda" to penalize excessive complexity.

Use your regularized regression to re-fit the model using all the polynomial features on your training data and using the value lambda = 10^4. Report the RMSE obtained for your training data, and the RMSE obtained for your testing data.

In [181…

```
def multivariate_regularized_ols(xvalues, yvalues, R=0.01, batch_size=20, _l
    # initialize the parameters
    m, n  = xvalues.shape
    beta = np.random.rand(n, 1)
    alpha  = np.zeros((1, 1))

    loss  = []

    start_time = time.time()

    for iters in range(MaxIterations):
        mini_batches = create_mini_batches(xvalues, yvalues, batch_size)
        d_beta_m  = np.zeros_like(beta)
        d_alpha_m = np.zeros((1, 1))

        for X_batch, y_batch in mini_batches:
            y_hat = X_batch @ beta + alpha

            d_beta = (1/batch_size) * X_batch.T @ (y_hat - y_batch) + (_lamb
            d_alpha = (1/batch_size) * np.sum(y_hat - y_batch)

            beta  -= R*d_beta
            alpha -= R*d_alpha

            d_beta_m += d_beta/len(mini_batches)
            d_alpha_m += d_alpha/len(mini_batches)

            if np.linalg.norm(d_beta_m) < eps and abs(d_alpha_m) < eps:
                print(f"Converged in {iters} iterations")
                print("Time taken: {:.2f} seconds".format(time.time() - star
                return alpha, beta, iters

    print("Time taken: {:.2f} seconds".format(time.time() - start_time))
    return alpha, beta, iters
```

In [183…

```
_lambda_=10^4

alpha, beta, _ = multivariate_regularized_ols(X_train_st, y_train, R=0.001,

y_train_pred = X_train_st @ beta + alpha
y_test_pred = X_test_st @ beta + alpha

# RMSE Calculation
train_rmse = np.sqrt(mean_squared_error(y_train, y_train_pred))
test_rmse = np.sqrt(mean_squared_error(y_test, y_test_pred))

print(f"Train RMSE: {train_rmse}")
print(f"Test RMSE: {test_rmse}")
```

```
Converged in 5583 iterations
Time taken: 96.14 seconds
Train RMSE: 0.478929444726325 65
Test RMSE: 0.4918030229674391
```

your answer here

4.4: Cross-validate lambda

Question 17

This is where it all comes together! Use k-fold cross-validation to select the optimal value of lambda in a regression using all the polynomial features. In other words, define a set of different values of lambda. Then, using the 80% of your data that you set aside for training, iterate through the values of lambda one at a time. For each value of lambda, use k-fold cross-validation to compute the average cross-validated RMSE for that lambda value, computed as the average across the held-out folds. You should also record the average cross-validated train RMSE, computed as the average across the folds used for training. Create a scatter plot that shows RMSE as a function of lambda. The scatter plot should have two lines: a gold line showing the cross-validated RMSE, and a blue line showing the cross-validated train RMSE. At this point, you should not have touched your held-out 20% of "true" test data.

What value of lambda minimizes your cross-validated RMSE? Fix that value of lambda, and train a new model using all of your training data with that value of lambda (i.e., use the entire 80% of the data that you set aside in 4.1). Calculate the RMSE for this model on the 20% of "true" test data. How does your test RMSE compare to the RMSE from 3.2, 4.2, 4.3 and to the RMSE from nearest neighbors? What do you make of these results?

Go brag to your friends about how you just implemented cross-validated ridge-regularized multivariate regression using gradient descent optimization, from scratch!

In [214…
```
def cross_val_reg(xvalues, yvalues, R, batch_size, MaxIterations, eps, lambc
    kf = KFold(n_splits=k, shuffle=True, random_state=42)

    results = []

    for _lambda_ in lambdas:
        train_rmse = []
        test_rmse  = []
        print(f"Running model with lambda = {_lambda_}")
        for X_index, y_index in kf.split(xvalues):
            X_train, X_test = xvalues[X_index], xvalues[y_index]
            y_train, y_test = yvalues[X_index], yvalues[y_index]

            alpha, beta, _ = multivariate_regularized_ols(X_train, y_train,

            y_hat_train = X_train @ beta + alpha
            y_hat       = X_test  @ beta + alpha

            train_rmse.append(np.sqrt(mean_squared_error(y_train, y_hat_trai
            test_rmse.append(np.sqrt(mean_squared_error(y_test, y_hat)))

        results.append((_lambda_, np.mean(train_rmse), np.mean(test_rmse)))

    return results
```

In [222…
```
MaxIterations=1000
lambdas = np.logspace(2, 5, 10)

res = cross_val_reg(X_train_st, y_train, R=0.001, batch_size=20, MaxIteratic
_lambdas, train_RMSEs, test_RMSEs = zip(*res)

_lambdas = np.array(_lambdas)
test_RMSEs = np.array(test_RMSEs)
```

```
best_lambda = _lambdas[np.argmin(test_RMSEs)]
print(f"Optimal lambda: {best_lambda}")
```

```
Running model with lambda = 100.0
Converged in 52434 iterations
Time taken: 717.77 seconds
Converged in 24860 iterations
Time taken: 339.55 seconds
Converged in 9659 iterations
Time taken: 131.62 seconds
Converged in 9122 iterations
Time taken: 124.56 seconds
Converged in 36479 iterations
Time taken: 499.14 seconds
Running model with lambda = 215.44346900318845
Converged in 7597 iterations
Time taken: 103.46 seconds
Converged in 8961 iterations
Time taken: 122.42 seconds
Converged in 1467 iterations
Time taken: 20.02 seconds
Converged in 19851 iterations
Time taken: 270.61 seconds
Converged in 19255 iterations
Time taken: 261.71 seconds
Running model with lambda = 464.15888336127773
Converged in 2850 iterations
Time taken: 38.79 seconds
Converged in 557 iterations
Time taken: 7.64 seconds
Converged in 74731 iterations
Time taken: 1047.89 seconds
Converged in 5522 iterations
Time taken: 75.48 seconds
Converged in 6873 iterations
Time taken: 94.24 seconds
Running model with lambda = 1000.0
Converged in 4670 iterations
Time taken: 63.74 seconds
Converged in 58979 iterations
Time taken: 817.99 seconds
Converged in 41885 iterations
Time taken: 589.54 seconds
Converged in 5002 iterations
Time taken: 70.88 seconds
Converged in 34823 iterations
Time taken: 489.70 seconds
Running model with lambda = 2154.4346900318824
Converged in 9565 iterations
Time taken: 135.72 seconds
Converged in 14734 iterations
Time taken: 208.62 seconds
Converged in 8975 iterations
Time taken: 125.32 seconds
Converged in 27467 iterations
Time taken: 388.25 seconds
Converged in 56902 iterations
Time taken: 796.07 seconds
Running model with lambda = 4641.588833612777
```

```
Converged in 40349 iterations
Time taken: 568.73 seconds
Converged in 7336 iterations
Time taken: 104.37 seconds
Converged in 30581 iterations
Time taken: 429.70 seconds
Converged in 36709 iterations
Time taken: 513.64 seconds
Converged in 12768 iterations
Time taken: 176.56 seconds
Running model with lambda = 10000.0
Time taken: 1410.63 seconds
Converged in 51679 iterations
Time taken: 726.54 seconds
Time taken: 1363.97 seconds
Converged in 1915 iterations
Time taken: 26.01 seconds
Time taken: 1364.55 seconds
Running model with lambda = 21544.346900318822
Time taken: 1361.45 seconds
Converged in 70459 iterations
Time taken: 968.52 seconds
Time taken: 1371.07 seconds
Time taken: 1373.78 seconds
Time taken: 1366.46 seconds
Running model with lambda = 46415.888336127726
Time taken: 1366.11 seconds
Converged in 32241 iterations
Time taken: 440.81 seconds
Time taken: 1365.38 seconds
Time taken: 1364.26 seconds
Time taken: 1363.96 seconds
Running model with lambda = 100000.0
Time taken: 1367.19 seconds
Time taken: 1364.17 seconds
Time taken: 1365.04 seconds
Time taken: 1369.84 seconds
Time taken: 1373.12 seconds
Optimal lambda: 1000.0
```

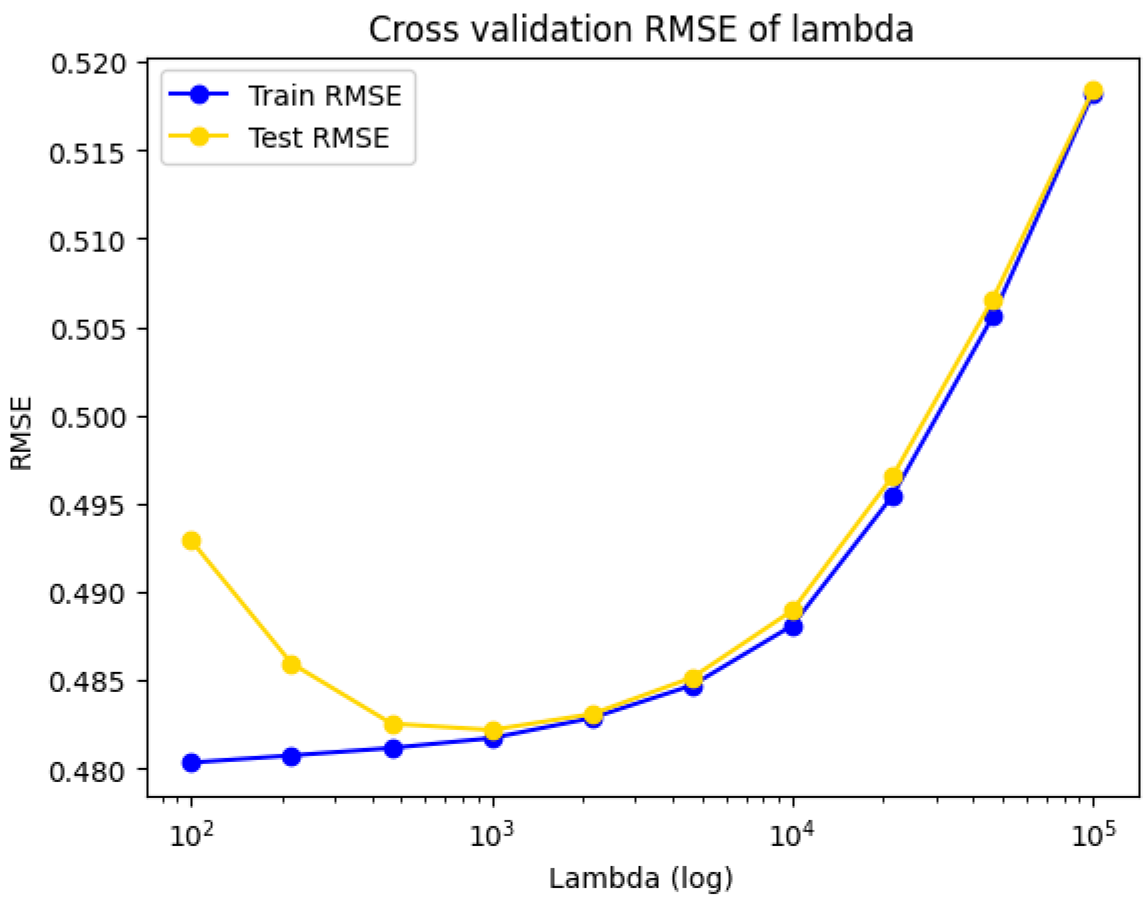
In [231…

```
print(_lambdas[3])
print(test_RMSEs[3])
print(train_RMSEs[3])
```

```
1000.0
0.48218707558953966
0.48172648395282025
```

In [223…

```
plt.plot(_lambdas, train_RMSEs, label="Train RMSE", marker='o', color='blue'
plt.plot(_lambdas, test_RMSEs, label="Test RMSE", marker='o', color='gold')
plt.xscale('log')
plt.xlabel("Lambda (log)")
plt.ylabel("RMSE")
plt.title("Cross validation RMSE of lambda")
plt.legend()
plt.show()
```



In [235…
```
print(best_lambda)
```

```
1000.0
```

In [236…
```
#training the model with best lambda on the entire 80% training data
alpha_final, beta_final, _ = multivariate_regularized_ols(X_train_st, y_trai
                                                    _lambda_=best_lambda, MaxIterati
```

```
Converged in 4106 iterations
Time taken: 72.24 seconds
```

In [237…
```
# Predict on the 20% held-out test data
y_hat_test = X_test_st @ beta_final + alpha_final
test_rmse_final = np.sqrt(mean_squared_error(y_test, y_hat_test))

print(f"Test RMSE with best lambda: {test_rmse_final}")
```

```
Test RMSE with best lambda: 0.4964626650882523
```

your answer here

4.5: Compare your results to sklearn ridge

Question 18 [extra-credit]

Repeat your analysis in 4.4, but this time use the sklearn implementation of ridge regression (sklearn.linearmodel.Ridge). Are the results similar? How would you explain the differences, if any?

In [234…
```
from sklearn.linear_model import Ridge

ridge_func = Ridge(alpha=best_lambda, solver='auto')
ridge_func.fit(X_train_st, y_train)

y_pred = ridge_func.predict(X_test_st)

test_rmse = np.sqrt(mean_squared_error(y_test, y_pred))
print(f"Test RMSE on Ridge: {test_rmse}")
```

```
Test RMSE on Ridge: 0.4963940112525463
```

In [240…
```
from sklearn.model_selection import cross_val_score
train_rmse_list = []
test_rmse_list = []
for _lambda_ in lambdas:
    ridge = Ridge(alpha=_lambda_)

    test_rmse = -cross_val_score(ridge, X_train_st, y_train, scoring="neg_rc

    avg_test_rmse = np.mean(test_rmse)
    test_rmse_list.append(avg_test_rmse)

best_lambda_sklearn = lambdas[np.argmin(test_rmse_list)]
print(f"Optimal lambda (sklearn Ridge): {best_lambda_sklearn}")
```

```
Optimal lambda (sklearn Ridge): 1000.0
```

In [241…
```
ridge_final = Ridge(alpha=best_lambda_sklearn)
ridge_final.fit(X_train_st, y_train)

y_hat_test_sklearn = ridge_final.predict(X_test_st)

test_rmse_final_sklearn = np.sqrt(mean_squared_error(y_test, y_hat_test_skle
print(f"Test RMSE with best lambda (sklearn Ridge): {test_rmse_final_sklearn
```

```
Test RMSE with best lambda (sklearn Ridge): 0.4963940112525463
```

The results are largely aligned, the differences are probaby explained by (i) the fact that sklearn uses different solvers that do not correspond exactly to our GD implementation and (ii) sklearn implementation is in all likelihood more effcient and careful in dealing with numeric issues.

4.6: AdaGrad

Question 19 [extra-credit]

AdaGrad is a method to implement gradient descent with different learning rates for each feature. Adaptive algorithms like this one are being extensively used especially in neural network training. Implement AdaGrad on 2.3 using `MedInc`, `HouseAge` and `AveRooms` as independent variables. Standardize these variables before inputting them to the gradient descent algorithm. Tune the algorithm until you estimate the regression coefficients within a tolerance of 1e-1. Use mini-batch gradient descent in this implementation. In summary: for each parameter (in our case one intercept and three slopes) the update step of the gradient (in this example β_j) at iteration k of the GD algorithm becomes:

$$\beta_j = \beta_j - \frac{R}{\sqrt{G_j^{(k)}}} \frac{\partial J(\alpha, \beta_1, \ldots)}{\partial \beta_j}$$

where $G_j^{(k)} = \sum_{i=1}^{k} (\frac{\partial J^{(i)}(\alpha,\beta_1,\ldots)}{\partial \beta_j})^2$ and R is your learning rate. The notation $\frac{\partial J^{(i)}(\alpha,\beta_1,\ldots)}{\partial \beta_j}$ corresponds to the value of the gradient at iteration (i). Essentially we are "storing" information about previous iteration gradients. Doing that we effectively decrease the learning rate slower when a feature x_i is sparse (i.e. has many zero values which would lead to zero gradients). Although this method is not necessary for our regression problem, it is good to be familiar with these methods as they are widely used in neural network training.

In [227…

```
def MBGD_AdaGrad(xvalues, yvalues, R=0.00001, batch_size=20, MaxIterations=1

    m, n = xvalues.shape
    beta = np.random.rand(n, 1)
    alpha = np.zeros((1, 1))

    ada_beta = np.zeros_like(beta)
    ada_alpha = np.zeros((1, 1))

    start_time = time.time()

    for iters in range(MaxIterations):
        mini_batches = create_mini_batches(xvalues, yvalues, batch_size)
        d_beta_m = np.zeros_like(beta)
        d_alpha_m = np.zeros((1, 1))

        for X_batch, y_batch in mini_batches:
            y_hat = X_batch @ beta + alpha

            d_beta = (1/batch_size) * X_batch.T @ (y_hat - y_batch)
            d_alpha = (1/batch_size) * np.sum(y_hat - y_batch)

            ada_beta += d_beta ** 2
            ada_alpha += d_alpha ** 2

            beta -= (R/(np.sqrt(ada_beta) + epsilon)) * d_beta
            alpha -= (R/(np.sqrt(ada_alpha) + epsilon)) * d_alpha

            d_beta_m += d_beta / len(mini_batches)
            d_alpha_m += d_alpha / len(mini_batches)

            if np.linalg.norm(d_beta_m) < eps and abs(d_alpha_m) < eps:
                print(f"Converged in {iters} iterations")
                print("Time taken: {:.2f} seconds".format(time.time() - star
                return alpha, beta, iters

    print("Time taken: {:.2f} seconds".format(time.time() - start_time))
    return alpha, beta, iters
```

In [232…

```
alpha, beta, iters = MBGD_AdaGrad(X_train_st, y_train, R=0.01, batch_size=20

print(f"alpha: {alpha} \nand betas: \n{beta}")
```

```
Converged in 720 iterations
Time taken: 15.73 seconds
alpha: [[2.2472522]]
and betas:
[[-0.24249091]
 [ 0.2744174 ]
 [ 0.10607189]
 [-0.22292635]
 [ 0.00293687]
 [ 0.28896675]
 [-0.38025788]
 [-0.20277245]
 [-0.14984484]
 [ 0.21296749]
 [ 0.21730346]
 [ 0.09452535]
 [-0.00127596]
 [ 0.01576663]
 [ 0.52828538]
 [ 0.31207965]
 [-0.06605153]
 [-0.1276462 ]
 [-0.16345318]]
```

In [ ]: