VERDICKT_JULIA-PS4

March 8, 2024

1 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 2.1) will be difficult, but it is the lynchpin 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 done 2.1, parts 2.3 and 4.3 will be relatively painless or incredibly painful.
- Part 4 (especially 4.3) 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.

1.1 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.

[ ]:
```
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
```

```
[ ]: # 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"␣
↪split is the same "random" split used by the rest of the class
np.random.seed(seed=1948)
```

---

2 Part 1: Getting oriented

2.0.1 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.

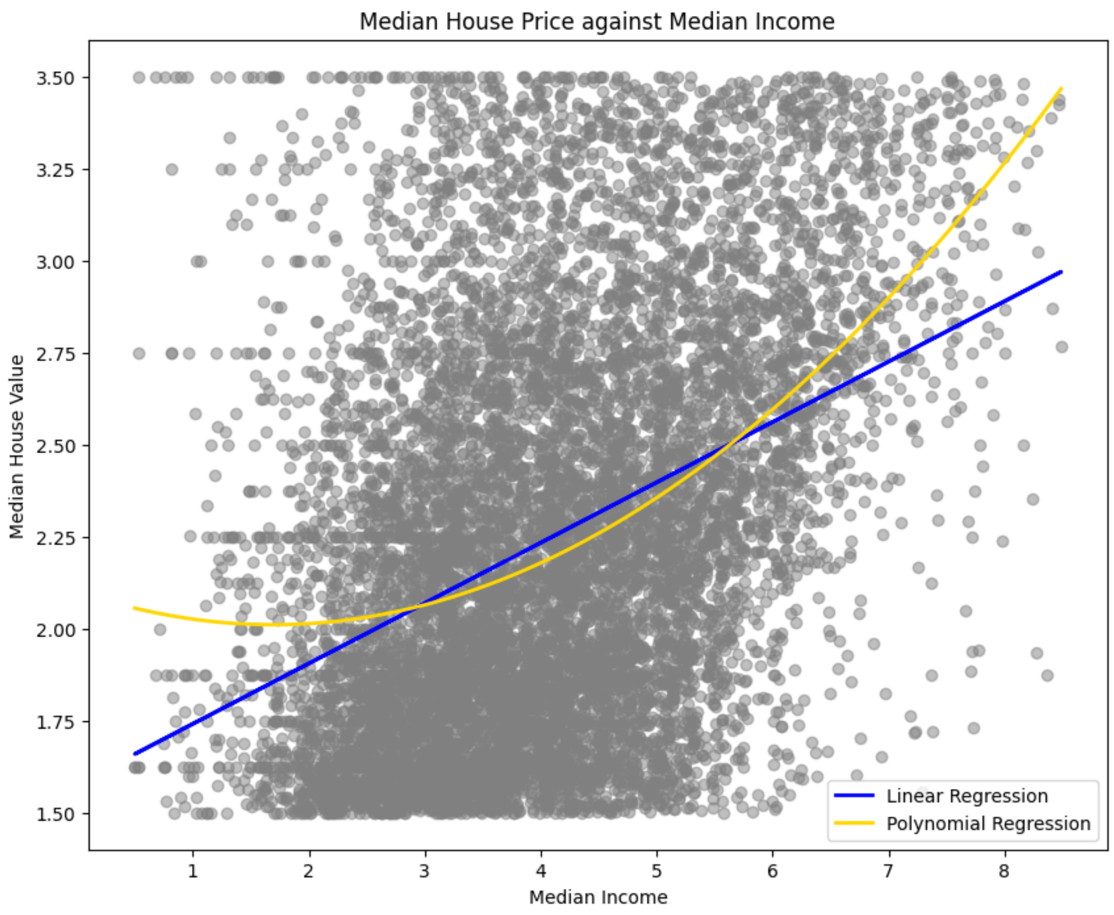
(a) Regress the median housing value `MedHouseVal` on the median income `MedInc`. Draw a scatter plot of housing price (y-axis) against income (x-axis), and draw the regression line in blue. You might want to make the dots semi-transparent if it improves the presentation of the figure.

(b) Regress the median housing value on median income and median income squared. Plot this new (curved) regression line in gold, on the same axes used for part (a).

(c) Interpret your results.

```
[ ]: # Your code here

# Step (a): Simple linear regression
X_linear = cal_df[['MedInc']]
y = cal_df['MedHouseVal']

# Fit the model
linear_model = LinearRegression()
linear_model.fit(X_linear, y)

# Predict the values for plotting
y_pred_linear = linear_model.predict(X_linear)

# Step (b): Polynomial regression (median income and median income squared)
# Generate the squared term
cal_df['MedIncSquared'] = cal_df['MedInc'] ** 2
```

```
X_poly = cal_df[['MedInc', 'MedIncSquared']]

# Fit the polynomial model
poly_model = LinearRegression()
poly_model.fit(X_poly, y)

# Predict the values for plotting
# Sort the values for a smooth curve
X_poly_sorted = X_poly.sort_values(by='MedInc')
y_pred_poly = poly_model.predict(X_poly_sorted)

# Plotting
plt.figure(figsize=(10, 8))

# Scatter plot of housing price against income
plt.scatter(cal_df['MedInc'], y, alpha=0.5, color = 'grey')

# Regression line in blue for the simple linear regression
plt.plot(X_linear, y_pred_linear, color='blue', linewidth=2, label='Linear␣
 ↪Regression')

# Regression curve in gold for the polynomial regression
plt.plot(X_poly_sorted['MedInc'], y_pred_poly, color='gold', linewidth=2,␣
 ↪label='Polynomial Regression')

# Labeling
plt.xlabel('Median Income')
plt.ylabel('Median House Value')
plt.title('Median House Price against Median Income')
plt.legend()

# Show the plot
plt.show()
```



Enter your observations here

Visually, I don't either regression line truly matches the shape of the data. I don't think the polynomial regression is very appropriate considering that the scatterplot seems to have a more linear shape if anything. However, in general a clear linear shape is not aparent in the scatterplot.

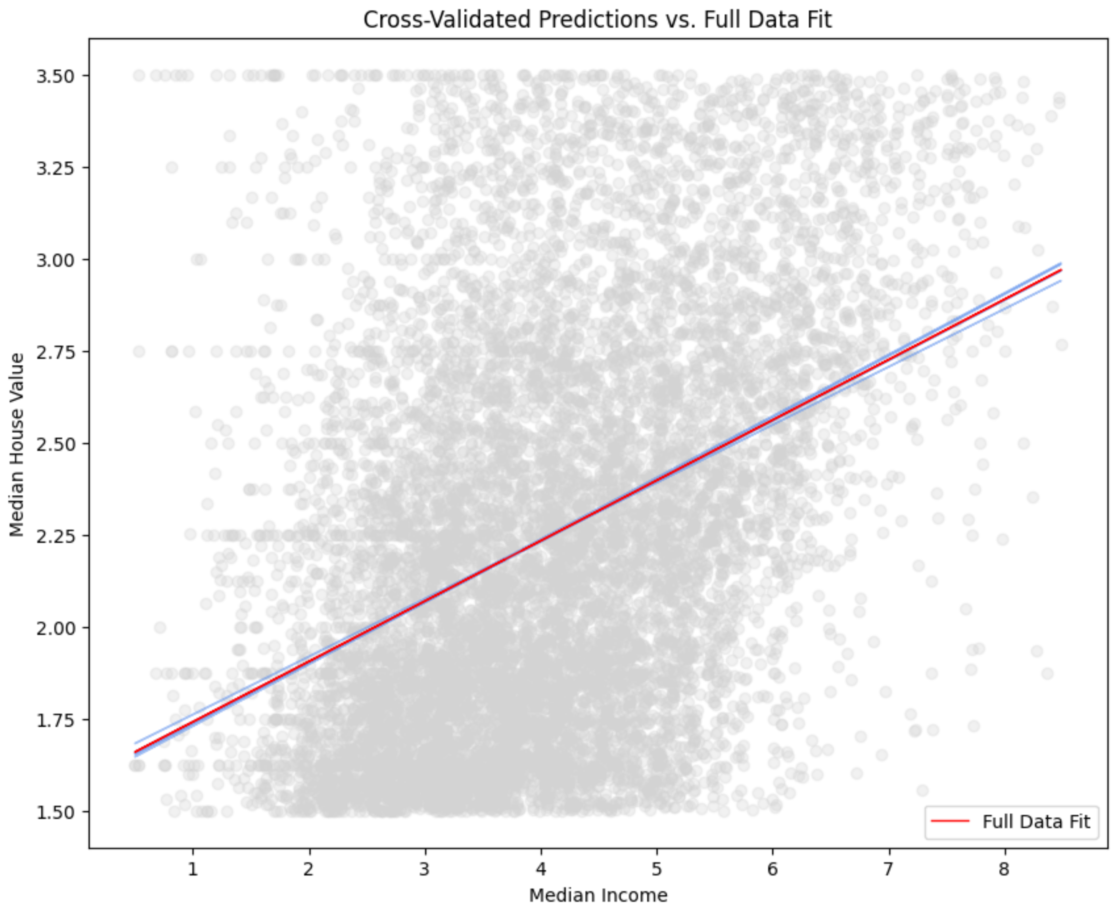
2.0.2 1.2 Training and testing

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.

Using the cross-validation functions from scikit-learn, use 5-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 income, and draw the five 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?

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

X = cal_df[['MedInc']]
y = cal_df['MedHouseVal']


# Initialize KFold
kf = KFold(n_splits=5, shuffle=True, random_state=1948)

# Prepare DataFrame to store predictions
predictions_df = pd.DataFrame(index=X.index, columns=[f"Fold_{i+1}" for i in␣
 ↪range(5)])

# Perform 5-fold cross-validation
fold = 0
for train_index, test_index in kf.split(X):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Train the model using the training subsets
    reg = LinearRegression()
    reg.fit(X_train, y_train)

    # Predict using the model and store in DataFrame
    predictions_df.iloc[:, fold] = reg.predict(X)
    fold += 1

# Plot settings
plt.figure(figsize=(10, 8))
plt.scatter(X['MedInc'], y, alpha=0.3, color = 'lightgrey')

# Plot the predictions from each fold
for column in predictions_df.columns:
    plt.plot(X['MedInc'], predictions_df[column], color='cornflowerblue',␣
 ↪linewidth=1, alpha=0.6)

# Fit the model using all data for comparison
reg_full = LinearRegression()
reg_full.fit(X, y)

# Plot the regression line using all data
plt.plot(X['MedInc'], reg_full.predict(X), color='red', linewidth=1,␣
 ↪label='Full Data Fit')

# Labeling
```

```
plt.xlabel('Median Income')
plt.ylabel('Median House Value')
plt.title('Cross-Validated Predictions vs. Full Data Fit')
plt.legend()

# Show the plot
plt.show()
```



Enter your observations here

Some of the blue lines differ slightly from the red line in slope but overall, the variation appears minimal. One can see some of the models produced during cross validation has higher or lower slopes and thus slightly different coefficients than the red line produced with all of the data.

3 Part 2: Gradient descent: Linear Regression

This is where it gets fun!

3.0.1 2.1 Implement gradient descent with one independent variable (median income)

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 3-4 different values of the learning rate R, 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.*

[ ]:
```
import time

"""
Function
--------
bivariate_ols
    Gradient Decent to minimize OLS. Used to find coefficients of bivariate OLS␣
 ↪Linear regression

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
"""

def compute_gradients(x, y, alpha, beta):
    predictions = alpha + beta * x
    error = predictions - y
    d_alpha = np.mean(error)
    d_beta = np.mean(error * x)
    return d_alpha, d_beta

def bivariate_ols(xvalues, yvalues, R=0.01, MaxIterations=1000, epsilon = 1e-6):
    start_time = time.time()
    #your code here
    alpha, beta = 0, 0  # Initialize alpha and beta
    for iteration in range(MaxIterations):
        d_alpha, d_beta = compute_gradients(xvalues, yvalues, alpha, beta)
        alpha_new = alpha - R * d_alpha
        beta_new = beta - R * d_beta

        # Check for convergence
        if np.abs(alpha_new - alpha) < epsilon and np.abs(beta_new - beta) <␣
↪epsilon:
            print(f"Converged in {iteration + 1} iterations")
            break

        alpha, beta = alpha_new, beta_new

    print("Time taken: {:.2f} seconds".format(time.time() - start_time))
    if iteration == (MaxIterations-1):
         print(f"Max Iteration Reached:{iteration+1}")
    return alpha, beta
```

[ ]:
```
full_coef = reg_full.coef_
full_inter = reg_full.intercept_

print("These are the coefficients generated by standard libraries:")
print(full_coef[0])
print(full_inter)
```

```
These are the coefficients generated by standard libraries:
0.16400595617582836
1.5772617051685813
```

```
[ ]: import warnings
     warnings.filterwarnings('ignore')

     learning_rates = np.array([1 / (10 ** i) for i in range(2,5)])
     for r in learning_rates:
         alpha, beta = bivariate_ols(cal_df['MedInc'], cal_df["MedHouseVal"], R = r,␣
      ↪MaxIterations=105000)
         print(f'Results for Learning Rate: {r}')
         print(f"Intercept: {alpha} (difference with sklearn: {np.abs(full_inter -␣
      ↪alpha)})")
         print(f"Coefficient: {beta} (difference with sklearn: {np.abs(full_coef[0]␣
      ↪- beta)})")
         print("==============================================")
```

```
Converged in 7821 iterations
Time taken: 3.24 seconds
Results for Learning Rate: 0.01
Intercept: 1.5761770605835874 (difference with sklearn: 0.0010846445849939101)
Coefficient: 0.1642474923692163 (difference with sklearn: 0.0002415361933879312)
==============================================
Converged in 53247 iterations
Time taken: 21.63 seconds
Results for Learning Rate: 0.001
Intercept: 1.5664118484371357 (difference with sklearn: 0.010849856731445673)
Coefficient: 0.1664220776685033 (difference with sklearn: 0.0024161214926749497)
==============================================
Time taken: 45.17 seconds
Max Iteration Reached:105000
Results for Learning Rate: 0.0001
Intercept: 1.0195084274609827 (difference with sklearn: 0.5577532777075986)
Coefficient: 0.288210331886478 (difference with sklearn: 0.12420437571064963)
==============================================
```

Enter your observations here

The solutions generated from gradient descent were very very close to the solutions generated by standard libraries. The lower learning rate didn't necessarily decreases the distance between the solutions generated by gradient descent and the solution generated by the standard library. This may be because, even with a smaller R, the stopping criteria "epsilon" that I have set will end the gradient descent maybe before the algorithm can reach the solutions generated by standard libraries. As a results, the larger learning rate produced the most similar results in this case.

3.0.2 2.2 Data normalization (done for you!)

Soon, you will implement a version of gradient descent that can use an arbitrary number of independent variables. Before doing this, we want to give you some code to standardize your features.

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.

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

     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)
```

3.0.3 2.3 Implement gradient descent with an arbitrary number of independent variables

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 0.001).

- *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*

```
[ ]: """
     Function
     --------
```

```
multivariate_ols
    Gradient Decent to minimize OLS. Used to find coefficients of bivariate OLS␣
 ↪Linear regression

Parameters
----------
xvalue_matrix, yvalues : narray
    xvalue_matrix: independent variable
    yvalues: dependent variable

R: float
    Learning rate

MaxIterations: Int
    maximum number of iterations


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

beta_array: array[float]
    coefficient
"""

def compute_gradients(X, y, alpha, betas):
    predictions = alpha + np.dot(X, betas)
    error = predictions - y
    d_alpha = np.mean(error)
    d_betas = np.dot(error, X) / len(X)
    return d_alpha, d_betas


def multivariate_ols(xvalue_matrix, yvalues, R=0.01, MaxIterations=1000,␣
 ↪epsilon = 1e-6):
    start_time = time.time()
    #your code here
    alpha, beta_array = 0, np.zeros(xvalue_matrix.shape[1])  # Initialize alpha␣
 ↪and beta
    for iteration in range(MaxIterations):
        d_alpha, d_betas = compute_gradients(xvalue_matrix, yvalues, alpha,␣
 ↪beta_array)
        alpha_new = alpha - R * d_alpha
        betas_new = beta_array - R * d_betas

        # Using norm for convergence check
        if np.abs(alpha_new - alpha) < epsilon and np.linalg.norm((betas_new -␣
 ↪beta_array), ord = 1) < epsilon:
            print(f"Converged in {iteration + 1} iterations")
            break

        alpha, beta_array = alpha_new, betas_new

    print("Time taken: {:.2f} seconds".format(time.time() - start_time))
    if iteration == (MaxIterations-1):
         print(f"Max Iteration Reached:{iteration+1}")
    return alpha, beta_array
```

[ ]:
```
X_mat = standardize(cal_df[['MedInc', 'HouseAge',␣
 ↪'AveRooms']],cal_df[['MedInc', 'HouseAge', 'AveRooms']] )
```

[ ]:
```
reg_test = LinearRegression(fit_intercept=True).fit(X_mat,cal_df["MedHouseVal"])
print("These are the coefficients generated by standard libraries:")
print(reg_test.coef_)
print(reg_test.intercept_)
```

```
These are the coefficients generated by standard libraries:
[ 0.2544861   0.08671268 -0.0308764 ]
2.2458724723388017
```

[ ]:
```
import warnings
warnings.filterwarnings('ignore')

learning_rates = np.array([0.1, 0.01, 0.001])
for r in learning_rates:
    alpha, betas = multivariate_ols(X_mat, cal_df["MedHouseVal"], R = r,␣
 ↪MaxIterations=100000)
    print(f'Results for Learning Rate: {r}')
    print(f"Intercept: {alpha}")
    print(f"Coefficients: {betas} ")
    print("===============================================")
```

```
Converged in 163 iterations
Time taken: 0.13 seconds
Results for Learning Rate: 0.1
Intercept: 2.2458723855084375
Coefficients: [ 0.25447878  0.08671025 -0.03087055]
===============================================
Converged in 1278 iterations
Time taken: 0.72 seconds
Results for Learning Rate: 0.01
Intercept: 2.2458664809349664
Coefficients: [ 0.25440883  0.08668579 -0.03081554]
```

```
===============================================
Converged in 9009 iterations
Time taken: 5.02 seconds
Results for Learning Rate: 0.001
Intercept: 2.2455987545444933
Coefficients: [ 0.25371636  0.08642013 -0.03029116]
===============================================
```

Enter your observations here

All of the coefficients generated by gradient descent with standardized data are close to those generated by standard libraries. Standardizing the data also drastically decreases the run-time of the gradient descent. Interestingly, the results for the largest R match mostly closely with the results from the standard libraries.

3.0.4 2.4 Compare standardized vs. non-standardized results

Repeat the analysis from 2.3, but this time do not standardize your variables - i.e., use the original data. Use the same three values of R (0.1, 0.01, and 0.001). What do you notice about the running time and convergence properties of your algorithm? Compare to the results you would obtain using standard libraries.

[ ]:
```
reg_test = LinearRegression(fit_intercept=True).fit(cal_df[['MedInc',␣
↪'HouseAge', 'AveRooms']],cal_df["MedHouseVal"])
print("These are the coefficients generated by standard libraries:")
print(reg_test.coef_)
print(reg_test.intercept_)
```

```
These are the coefficients generated by standard libraries:
[ 0.19077902  0.00698801 -0.01350345]
1.342003006912614
```

[ ]:
```
# Your code here

learning_rates = np.array([0.1, 0.01, 0.001])
for r in learning_rates:
    alpha, betas = multivariate_ols(cal_df[['MedInc', 'HouseAge', 'AveRooms']],␣
↪cal_df["MedHouseVal"], R = r, MaxIterations=100000)
    print(f'Results for Learning Rate: {r}')
    print(f"Intercept: {alpha}")
    print(f"Coefficients: {betas} ")
    print("===============================================")
```

```
Time taken: 61.05 seconds
Max Iteration Reached:100000
Results for Learning Rate: 0.1
Intercept: nan
Coefficients: [nan nan nan]
===============================================
Time taken: 62.77 seconds
Max Iteration Reached:100000
Results for Learning Rate: 0.01
Intercept: nan
Coefficients: [nan nan nan]
===============================================
Converged in 95094 iterations
Time taken: 54.68 seconds
Results for Learning Rate: 0.001
Intercept: 1.3182671271202793
Coefficients: [ 0.19339132  0.00727499 -0.01276734]
===============================================
```

Enter your observations here

Without standardizing the variables, the gradient descent does not converge unless the step size is small. Even when it is small enough, it still reaches the the max iterations I set. This results in coefficients that are not incredibly similar to the ones we expect as seen in the results from standard libraries. Overall, the run-time is longer and the results not as good.

4 3. Prediction

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

4.0.1 3.1 Cross-Validation

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` and `AveRoomsNorm`, `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.

[ ]:
```
def compute_rmse(predictions, yvalues):
    P = np.array(predictions)
    Y = np.array(yvalues)
    rmse = ((P-Y)**2).sum()*1.0 / len(P)
    rmse = np.sqrt(rmse)
    return rmse

# Your code here



kf = KFold(n_splits=5, shuffle=True, random_state=1948)
coefs = []
rmses = []
```

```
X = cal_df[['MedInc', 'HouseAge', 'AveRooms']]
y = cal_df['MedHouseVal']

for train_index, test_index in kf.split(X):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y[train_index], y[test_index]

    X_train_s = standardize(X_train, X_train)
    X_test_s = standardize(X_train, X_test)

    # Training the model
    alpha, betas = multivariate_ols(X_train_s, y_train, R = 0.001,␣
 ↪MaxIterations=10000)
    predictions = alpha + np.dot(X_test_s, betas)
    fold_rmse = compute_rmse(predictions, y_test)
    rmses.append(fold_rmse)
    # Storing the coefficients
    coefs.append([alpha] + list(betas))
```

```
Converged in 9007 iterations
Time taken: 4.33 seconds
Converged in 8783 iterations
Time taken: 4.16 seconds
Converged in 8976 iterations
Time taken: 4.50 seconds
Converged in 9643 iterations
Time taken: 5.80 seconds
Converged in 8898 iterations
Time taken: 5.31 seconds
```

```
[ ]: print("Coefficients from 5-fold Cross-Validation:")
table = pd.DataFrame(coefs)
table.columns = np.append(np.array(["intercept"]), np.array(X.columns))
table = table.reset_index(names="Fold")
table["Fold"] =table["Fold"] + 1
table['rmse'] = rmses
table
```

```
Coefficients from 5-fold Cross-Validation:
```

```
[ ]:    Fold  intercept    MedInc  HouseAge  AveRooms      rmse
     0     1   2.252266  0.260311  0.093810 -0.027443  0.501277
     1     2   2.244810  0.255553  0.082760 -0.028670  0.497259
     2     3   2.244345  0.252639  0.085742 -0.033158  0.485772
     3     4   2.242270  0.257770  0.086098 -0.034168  0.490594
     4     5   2.244319  0.243904  0.083369 -0.029943  0.487039
```

Discuss your results here

4.0.2 3.2 Predicted values and RMSE

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%. Repeate this for the remaining folds. Report the RMSE for each of the 5-folds, and the average (mean) RMSE across the five folds. How does this average RMSE compare to the performance of your nearest neighbor algorithm from the last problem set?

```
[ ]: # Your code here

     #I calculated the RMSE's in the previous step
     print("Average RMSE across 5 folds:", np.mean(rmses))
     print("RMSEs for each fold:\n")
     table[['Fold', 'rmse']]
```

```
Average RMSE across 5 folds: 0.49238821582955633
RMSEs for each fold:
```

```
[ ]:    Fold      rmse
     0     1  0.501277
     1     2  0.497259
     2     3  0.485772
     3     4  0.490594
     4     5  0.487039
```

Discuss your results here

Overall, the RMSE's are all lower thant those generated with the kNN algorithms in the previous problem set. In the previous problem set, the minimizing RMSE was about .743900948208243 (problem 2.6). Thus the results here are notably lower.

4.1 4 Regularization

4.1.1 4.1 Get prepped

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 of sklearn.preprocessing make sure you drop the constant polynomial feature (degree 0). You should have a total of 19 polynomial features.

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

```
[ ]: from sklearn.preprocessing import PolynomialFeatures
     from sklearn.model_selection import train_test_split

     # Your code here
     # Assuming features is your original DataFrame with the columns ['MedInc',␣
     ↪'HouseAge', 'AveRooms']
     poly = PolynomialFeatures(degree=3, include_bias=False)
     features = cal_df[['MedInc', 'HouseAge', 'AveRooms']]
     poly_features = poly.fit_transform(features)

     # Generate column names for the polynomial features
     feature_names = poly.get_feature_names_out(features.columns)

     # Create a DataFrame for the polynomial features with named columns
     poly_features_df = pd.DataFrame(poly_features, columns=feature_names)
     print("The dimensions show 19 columns:")
     print(poly_features.shape)
     X_train, X_test, y_train, y_test = train_test_split(poly_features,␣
     ↪cal_df['MedHouseVal'], test_size=0.2, random_state=1948)
```

```
The dimensions show 19 columns:
(10484, 19)
```

4.1.2 4.2 Complexity and overfitting?

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?

```
[ ]: # Your code here
X_train_s = standardize(X_train, X_train)
X_test_s = standardize(X_train, X_test)

# Training the model
alpha, betas = multivariate_ols(X_train_s, y_train, R = 0.01,␣
↪MaxIterations=10000)
predictions_train = alpha + np.dot(X_train_s, betas)
train_rmse = compute_rmse(predictions_train, y_train)
predictions_test = alpha + np.dot(X_test_s, betas)
test_rmse = compute_rmse(predictions_test, y_test)

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

```
Time taken: 5.21 seconds
Max Iteration Reached:10000
Train RMSE: 0.4801196083380557
Test RMSE: 0.4934757452838727
```

Discuss your results here

The test RMSE is slightly higher, indicating some level of over-fitting. However, the difference is rather small, so likely not too worrisome. Honestly, this result is not surprising considering that including all polynomial feature will increase over-fitting. In 3.2, the average of the testing RMSE's for 5 folds was 0.49250724640767907, this is similar to the test RMSE generated above but lower. This implies that the polynomial regression had more overfitting than the non-polynomial regression.

Now to compare to the results for the kNN algorithm in the previous problem set, The testing RMSE and cross-validated RMSE for the kNN algorithm were much higher at 0.7439009482082435 and 0.7367691310268063 respectively. Thus there was still over-fitting and the RMSE's were higher overall when using kNN.

4.1.3 4.3 Ridge regularization (basic)

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.

```
[ ]: def multivariate_regularized_ols(xvalue_matrix, yvalues,
                                      R=0.01, MaxIterations=1000,lmbda=1,
                                      epsilon = 1e-6, verbose = True):
         start_time = time.time()
         # Your code here
         alpha, beta_array = 0, np.zeros(xvalue_matrix.shape[1])  # Initialize alpha␣
      ↪and beta
         for iteration in range(MaxIterations):
             d_alpha, d_betas = compute_gradients(xvalue_matrix, yvalues, alpha,␣
      ↪beta_array)
             alpha_new = alpha - R * d_alpha
             betas_new = beta_array * (1- R*(lmbda/len(X))) - R * d_betas

             # Using norm for convergence check
             if np.abs(alpha_new - alpha) < epsilon and np.linalg.norm((betas_new -␣
      ↪beta_array), ord = 1) < epsilon:
                 print(f"Converged in {iteration + 1} iterations")
                 break

             alpha, beta_array = alpha_new, betas_new
         if verbose:
             print("Time taken: {:.2f} seconds".format(time.time() - start_time))
             if iteration == (MaxIterations-1):
                 print(f"Max Iteration Reached:{iteration+1}")
         return alpha, beta_array
```

```
[ ]: # Training the model
     alpha, betas = multivariate_regularized_ols(X_train_s, y_train, R = 0.01, lmbda␣
      ↪= 10**4, MaxIterations=10000)
     predictions_train = alpha + np.dot(X_train_s, betas)
     train_rmse = compute_rmse(predictions_train, y_train)
     predictions_test = alpha + np.dot(X_test_s, betas)
     test_rmse = compute_rmse(predictions_test, y_test)

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

```
Converged in 999 iterations
Time taken: 0.85 seconds
Train RMSE: 0.4865094619049543
Test RMSE: 0.49687864314191615
```

Discuss your results here

Now the RMSE's are higher across the board, and still show evidence of over-fitting. However, the magnitudes of the RMSEs are still much less than those found using kNN in the previous problem set. The increased RMSE are likely due to the regularization penalty.

4.1.4 4.4: Cross-validate lambda

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?

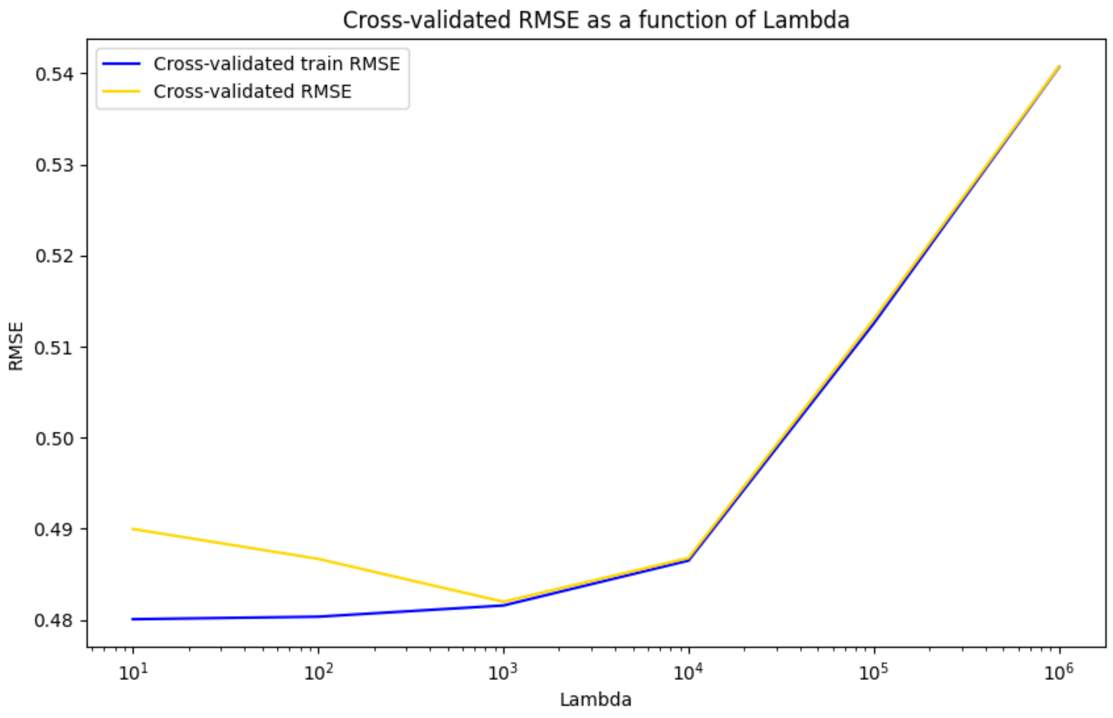
Go brag to your friends about how you just implemented cross-validated ridge-regularized multivariate regression using gradient descent optimization, from scratch!

```
[ ]: # Your code here

     # Define the range of lambda values
     lambda_values = np.array([10**exp for exp in range(1, 7)])
     kf = KFold(n_splits=5, shuffle=True, random_state=1948)
```

```python
# Prepare lists to store the results
cv_train_rmses = []
cv_val_rmses = []

# Iterate over lambda values
for lmbda in lambda_values:
    fold_train_rmses = []
    fold_val_rmses = []

    # Perform k-fold cross-validation
    for train_index, val_index in kf.split(X_train_s):
        # Split data
        X_train_fold, X_val_fold = X_train_s[train_index], X_train_s[val_index]
        y_train_fold, y_val_fold = y_train.iloc[train_index], y_train.
 ↪iloc[val_index]

        # Train the model
        alpha, betas = multivariate_regularized_ols(X_train_fold, y_train_fold,
                                                    lmbda=lmbda,␣
 ↪MaxIterations=10000,
                                                    verbose = False)

        # Compute RMSE on training and validation sets
        train_rmse = compute_rmse(alpha + np.dot(X_train_fold, betas),␣
 ↪y_train_fold)
        val_rmse = compute_rmse(alpha + np.dot(X_val_fold, betas), y_val_fold)

        fold_train_rmses.append(train_rmse)
        fold_val_rmses.append(val_rmse)

    # Record the average RMSE for this lambda value
    cv_train_rmses.append(np.mean(fold_train_rmses))
    cv_val_rmses.append(np.mean(fold_val_rmses))
```

```
Converged in 4511 iterations
Converged in 4616 iterations
Converged in 4577 iterations
Converged in 4841 iterations
Converged in 4512 iterations
Converged in 999 iterations
Converged in 999 iterations
Converged in 999 iterations
Converged in 1001 iterations
Converged in 999 iterations
Converged in 999 iterations
Converged in 999 iterations
Converged in 999 iterations
Converged in 999 iterations
Converged in 999 iterations
Converged in 999 iterations
Converged in 999 iterations
Converged in 999 iterations
Converged in 999 iterations
Converged in 999 iterations
```

[ ]:
```
# Plotting the RMSE as a function of lambda
plt.figure(figsize=(10, 6))
plt.plot(lambda_values, cv_train_rmses, color='blue', label='Cross-validated␣
  ↪train RMSE')
plt.plot(lambda_values, cv_val_rmses, color='gold', label='Cross-validated␣
  ↪RMSE')
plt.xscale('log')
plt.xlabel('Lambda')
plt.ylabel('RMSE')
plt.title('Cross-validated RMSE as a function of Lambda')
plt.legend()
plt.show()
```



[ ]:
```
# Find the optimal lambda value
optimal_lambda = lambda_values[np.argmin(cv_val_rmses)]
```

```
print(f"Optimal Lambda: {optimal_lambda}")

# Train a new model using the optimal lambda on all training data
alpha, betas = multivariate_regularized_ols(X_train_s, y_train,␣
↪lmbda=optimal_lambda, MaxIterations=10000)

# Compute RMSE on the "true" test data
test_rmse = compute_rmse(alpha + np.dot(X_test_s, betas), y_test)
print(f"Test RMSE with Optimal Lambda: {test_rmse}")
```

```
Optimal Lambda: 1000
Converged in 4526 iterations
Time taken: 2.44 seconds
Test RMSE with Optimal Lambda: 0.49432049637053577
```

Discuss your results here

A lambda of 10^3 minimizes the cross-validated RMSE. At this lambda, the test RMSE is about 0.4943186925009971. This is lower than the RMSE for 4.3 but higher than the RMSE's from 3.2, 4.2. Also, it is lower than the RMSE from the nearest neighbors algorithm. This implies that while regularization definitely added a penalty that leads to higher test RMSE's realtive to the non-regularized results from 3.2 and 4.2. However, using cross-validation we can find an "optimal" lower RMSE under regularization. This still allows us to decrease our RMSE under some constraint such that we are still penalized relative to the non-regularized models, but optimal relative to the non-crossvalidated model.

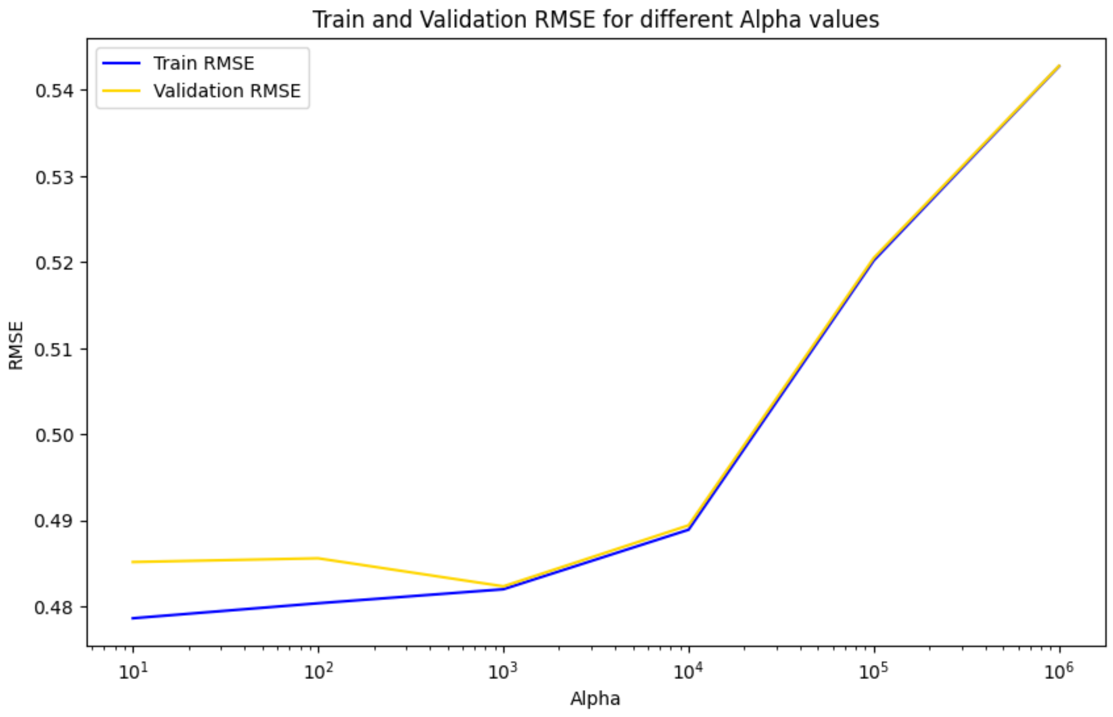
4.1.5 4.5: Compare your results to sklearn ridge [extra-credit]

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?

[ ]:
```
# Your code here

import numpy as np
from sklearn.linear_model import Ridge
from sklearn.model_selection import KFold, cross_val_score
from sklearn.metrics import mean_squared_error
from math import sqrt
import matplotlib.pyplot as plt

# Define the range of alpha values (equivalent to lambda in Ridge regression)
alpha_values = [10**exp for exp in range(1, 7)]

# Prepare lists to store the results
cv_train_rmses = []
cv_val_rmses = []
```

```python
kf = KFold(n_splits=5, shuffle=True, random_state=1948)

# Iterate over alpha values
for alpha in alpha_values:
    model = Ridge(alpha=alpha, max_iter=10000)

    # Lists to store RMSE for each fold
    fold_train_rmses = []
    fold_val_rmses = []

    for train_index, val_index in kf.split(X_train_s):
        # Split the data
        X_train_fold, X_val_fold = X_train_s[train_index], X_train_s[val_index]
        y_train_fold, y_val_fold = y_train.iloc[train_index], y_train.
 ↪iloc[val_index]

        # Fit the model
        model.fit(X_train_fold, y_train_fold)

        # Predict and calculate RMSE on training fold
        train_pred = model.predict(X_train_fold)
        train_rmse = sqrt(mean_squared_error(y_train_fold, train_pred))
        fold_train_rmses.append(train_rmse)

        # Predict and calculate RMSE on validation fold
        val_pred = model.predict(X_val_fold)
        val_rmse = sqrt(mean_squared_error(y_val_fold, val_pred))
        fold_val_rmses.append(val_rmse)

    # Record the average RMSE for this alpha value
    cv_train_rmses.append(np.mean(fold_train_rmses))
    cv_val_rmses.append(np.mean(fold_val_rmses))

# Plotting RMSE as function of alpha
plt.figure(figsize=(10, 6))
plt.plot(alpha_values, cv_train_rmses, 'b', label='Train RMSE')
plt.plot(alpha_values, cv_val_rmses, 'gold', label='Validation RMSE')
plt.xscale('log')
plt.xlabel('Alpha')
plt.ylabel('RMSE')
plt.title('Train and Validation RMSE for different Alpha values')
plt.legend()
plt.show()

# Select the optimal alpha value
optimal_alpha = alpha_values[np.argmin(cv_val_rmses)]
print(f"Optimal Alpha: {optimal_alpha}")
```

```python
# Train a new model on the full training set with the optimal alpha
final_model = Ridge(alpha=optimal_alpha, max_iter=10000)
final_model.fit(X_train_s, y_train)

# Compute RMSE on the test data
test_pred = final_model.predict(X_test_s)
test_rmse = sqrt(mean_squared_error(y_test, test_pred))
print(f"Test RMSE with Optimal Alpha: {test_rmse}")
```



```
Optimal Alpha: 1000
Test RMSE with Optimal Alpha: 0.4944292368487741
```

Discuss your results here

The optimal Alphas are identical. The test RMSE's are not indentical but very very close. Differences might be caused by variation in the cross-validation splits.

4.1.6 4.6: AdaGrad [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:

$$4\beta_j = \beta_j - \frac{R}{\sqrt{G_j^{(k)}}} \frac{\partial J(\alpha, \beta_1, ...)}{\partial \beta_j}$$

where $G_j^{(k)} = \sum_{i=1}^{k} (\frac{\partial J^{(i)}(\alpha,\beta_1,...)}{\partial \beta_j})^2$ and R is your learning rate. The notation $\frac{\partial J^{(i)}(\alpha,\beta_1,...)}{\partial \beta_j}$ 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.

[ ]:
```
X_mat = standardize(cal_df[['MedInc', 'HouseAge',␣
↪'AveRooms']],cal_df[['MedInc', 'HouseAge', 'AveRooms']] )
X = X_mat.values
y = cal_df["MedHouseVal"].values.reshape(-1, 1)

# Add a column of ones to X to account for the intercept term
X = np.hstack([np.ones((X.shape[0], 1)), X])

# Initialize parameters
n_features = X.shape[1]
beta = np.zeros((n_features, 1))  # Regression coefficients (including␣
↪intercept)
G = np.zeros((n_features, 1))  # Accumulated squared gradients for each␣
↪coefficient
R = 0.01  # Learning rate
tolerance = 1e-1
max_iterations = 10000
batch_size = 32

# Function to compute gradients
def compute_gradients(X_batch, y_batch, beta):
    predictions = X_batch @ beta
    errors = predictions - y_batch
    gradients = 2 / X_batch.shape[0] * X_batch.T @ errors
    return gradients

# Mini-batch gradient descent with AdaGrad and convergence check
converged = False
for iteration in range(max_iterations):
    if converged:
        break

    indices = np.random.permutation(len(X))
    X_shuffled = X[indices]
    y_shuffled = y[indices]

    for i in range(0, len(X), batch_size):
        X_batch = X_shuffled[i:i+batch_size]
        y_batch = y_shuffled[i:i+batch_size]

        gradients = compute_gradients(X_batch, y_batch, beta)
        G += gradients ** 2
        adjusted_gradients = gradients / (np.sqrt(G) + 1e-8)  # Add epsilon to␣
↪avoid division by zero
        beta -= R * adjusted_gradients

        # Convergence check
        if np.sum(np.abs(adjusted_gradients)) < tolerance:
            converged = True
            break  # Breaks out of the inner loop

print("Estimated regression coefficients:", beta.flatten())
if converged:
    print("Convergence achieved.")
else:
    print("Maximum iterations reached without convergence.")
```

```
Estimated regression coefficients: [0.29835161 0.08720678 0.01392191 0.02609584]
Convergence achieved.
```

[ ]:
```
def compute_gradients(X, y, beta):
    predictions = np.dot(X, beta)
    errors = predictions - y
    gradient = 2 / X.shape[0] * np.dot(X.T, errors)
    return gradient

def adagrad_multivariate_ols(xvalue_matrix, yvalues, R=0.01,␣
↪MaxIterations=1000, epsilon=1e-6):
    start_time = time.time()
    # Adding intercept term
    intercept = np.ones((xvalue_matrix.shape[0], 1))
    X = np.hstack((intercept, xvalue_matrix))
    y = yvalues.reshape(-1, 1)

    beta = np.zeros((X.shape[1], 1))
    G = np.zeros((X.shape[1], 1))  # Accumulated squared gradients
    epsilon = 1e-8  # Smoothing term to prevent division by zero

    for iteration in range(MaxIterations):
        gradient = compute_gradients(X, y, beta)
        G += gradient ** 2
        adjusted_gradient = gradient / (np.sqrt(G) + epsilon)
        beta_prev = beta.copy()
        beta -= R * adjusted_gradient

        # Check for convergence
        if np.linalg.norm(beta - beta_prev, ord=1) < epsilon:
            print(f"Converged in {iteration + 1} iterations")
            break

    print("Time taken: {:.2f} seconds".format(time.time() - start_time))
    if iteration == MaxIterations - 1:
        print(f"Max Iterations Reached: {iteration+1}")

    return beta[0][0], beta[1:]
```

```
[ ]: X_mat = standardize(cal_df[['MedInc', 'HouseAge',␣
      ↪'AveRooms']],cal_df[['MedInc', 'HouseAge', 'AveRooms']] )
     X = X_mat.values
     y = cal_df["MedHouseVal"].values

     learning_rates = [0.1, 0.01, 0.001]
     for R in learning_rates:
          print(f"Learning Rate: {R}")
          alpha, betas = adagrad_multivariate_ols(X, y, R=R, MaxIterations=10000)
          print(f"Intercept: {alpha}")
          print(f"Coefficients: {betas.flatten()}")
          print("===============================================")
```

```
Learning Rate: 0.1
Converged in 1953 iterations
Time taken: 0.22 seconds
Intercept: 2.2458710760532736
Coefficients: [ 0.2544861   0.08671268 -0.0308764 ]
===============================================
Learning Rate: 0.01
Time taken: 0.83 seconds
Max Iterations Reached: 10000
Intercept: 1.5926195756156571
Coefficients: [ 0.2544861   0.08671268 -0.0308764 ]
===============================================
Learning Rate: 0.001
Time taken: 1.03 seconds
Max Iterations Reached: 10000
Intercept: 0.1954908542947251
Coefficients: [0.16358083 0.06709047 0.00129863]
===============================================
```

Discuss your results here

For AdaGrad with higher R the results are somehow more accurate. Overall the speed is much faster and with a R of 0.1, and the results are still quite accurate. At the same time, for lower R, the results are pretty inaccruate. A main advantage here appears to me speed.