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.

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 [ ]:
```
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 [ ]:
```
# 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=1948)
```

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.

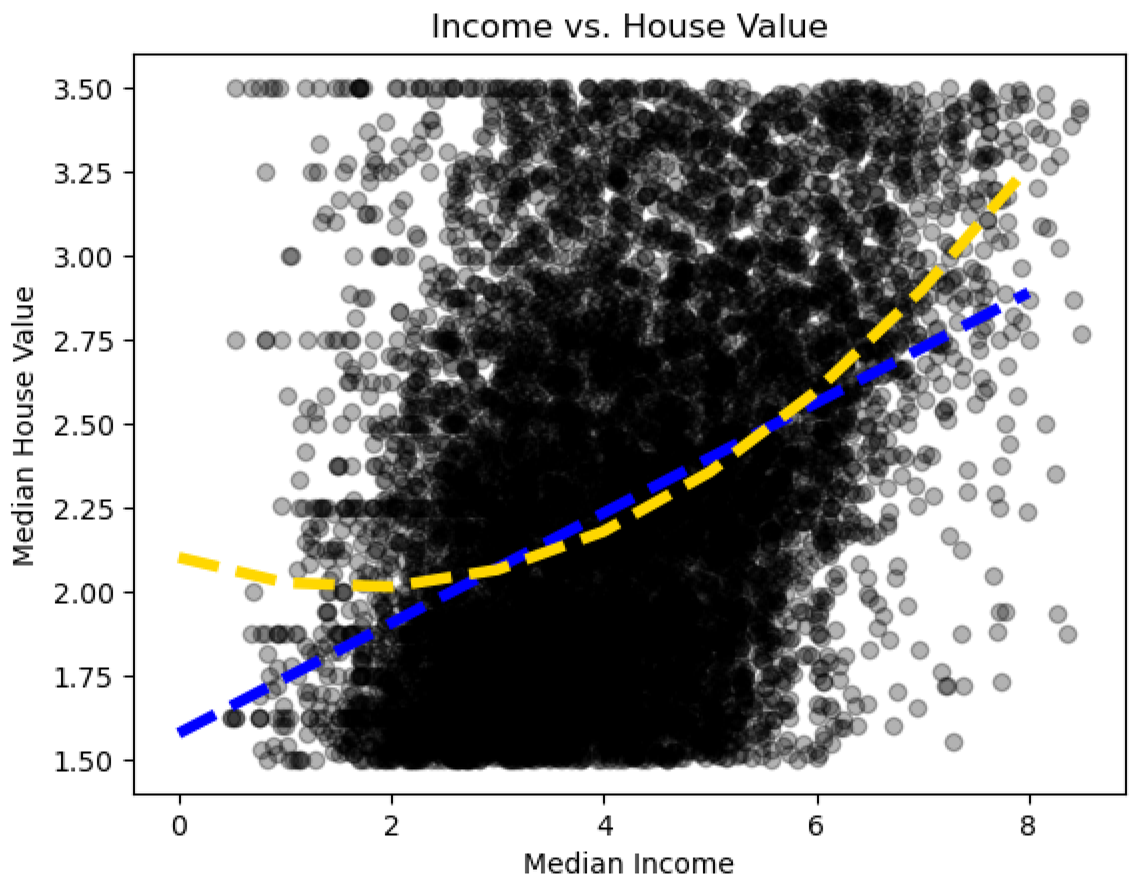
Use the Linear Regression class from sklearn or the OLS class from SciPy to explore the relationship between median housing value and median income in California's census block groups.

(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.

In [ ]:
```
# Your code here
model = LinearRegression()

# a) Regress the median housing value `MedHouseVal` on the median income `Me
    # (You might want to make the dots semi-transparent if it improves the p

cal_df['MedInc'].values.shape # (10484,)
cal_df['MedHouseVal'].values.shape # (10484,)

model.fit(cal_df['MedInc'].values.reshape(-1,1), cal_df['MedHouseVal'].value
intercept = model.intercept_ # 1.577
slope = model.coef_[0] # 0.164

pred = model.predict(cal_df['MedInc'].values.reshape(-1,1))
#plt.scatter(pred, cal_df['MedHouseVal'])

# Scatter Plot
plt.scatter(cal_df['MedInc'], cal_df['MedHouseVal'], alpha = .3, c='black')
X_sample = np.arange(0, np.ceil(np.max(cal_df['MedInc'])))
y_sample = intercept + X_sample * slope
plt.plot(X_sample, y_sample, color='blue', lw = 4, ls = 'dashed') # add fit

plt.title('Income vs. House Value')
plt.xlabel('Median Income')
plt.ylabel('Median House Value')

# b) Regress the median housing value on median income and median income squ
model2 = LinearRegression()

# Set X, Y (let's make this iteration a bit more explicit)
X = np.column_stack((cal_df['MedInc'].values, cal_df['MedInc'].values ** 2))
Y = cal_df['MedHouseVal'].values

# Fit model
model2.fit(X, Y)

# Look at output
coefs = model2.coef_
intercept = model2.intercept_

# Predict
pred2 = model2.predict(X)
pred2

# Plot again
y2_sample = intercept + coefs[0] * X_sample + coefs[1] * X_sample * X_sample
plt.plot(X_sample, y2_sample, color = 'gold', lw = 4, ls = 'dashed')

plt.show()
```



```
In [ ]: # c) Interpret your results

        # Let's look at error and r2
        from sklearn.metrics import mean_squared_error, r2_score
        mse_1 = mean_squared_error(cal_df['MedHouseVal'], pred)
        r2_1 = r2_score(cal_df['MedHouseVal'], pred)
        mse_2 = mean_squared_error(cal_df['MedHouseVal'], pred2)
        r2_2 = r2_score(cal_df['MedHouseVal'], pred2)


        print(f'Linear Model:\t\tmse: {mse_1: .3f}, r2: {r2_1: .3f}')
        print(f'Interaction Model:\tmse: {mse_2: .3f}, r2: {r2_2: .3f}')
```

```
Linear Model:           mse:  0.250, r2:  0.161
Interaction Model:      mse:  0.245, r2:  0.179
```

Enter your observations here

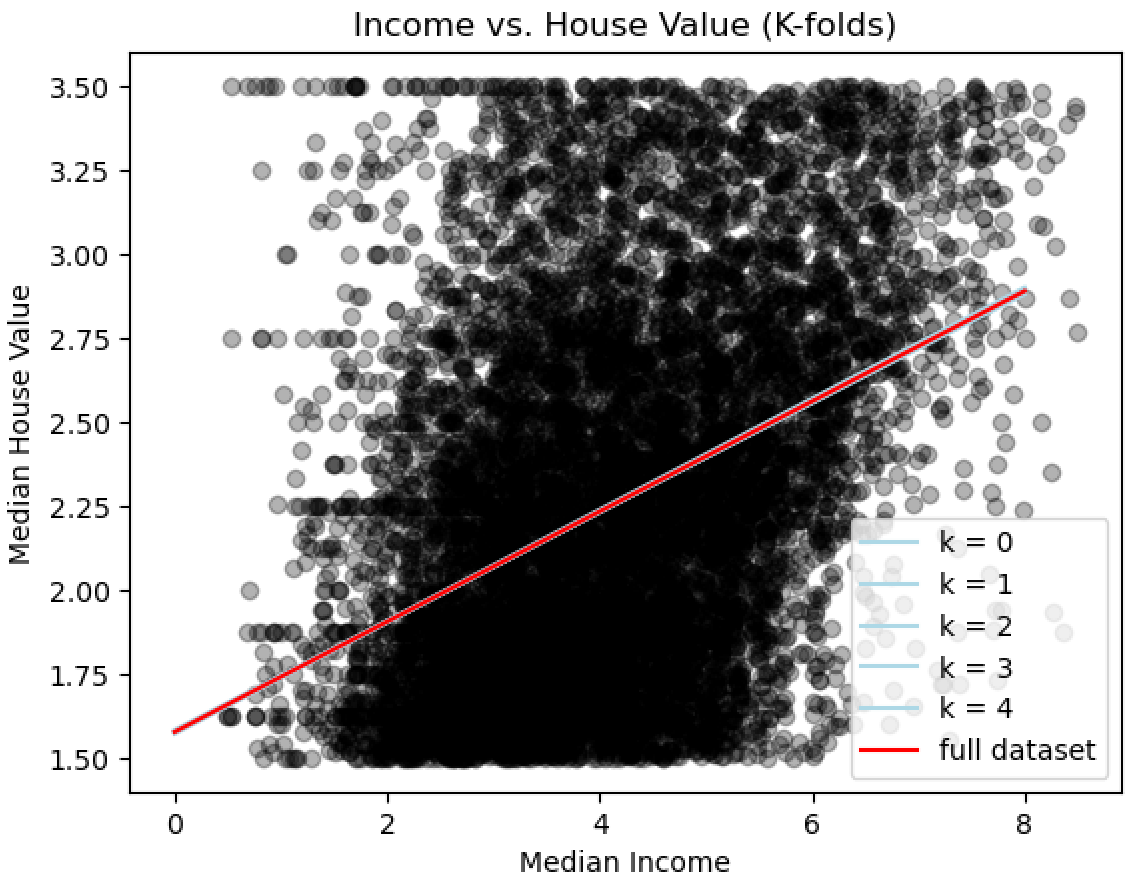
Based on R2 and MSE, the Interaction model appears to be a slightly better fit, however it doesn't appear to be a big difference

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?

In [ ]:
```python
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score

# your code here
k = 5

# a) Linear Model using K-folds
model_kf_1 = LinearRegression()

X = cal_df['MedInc'].values.reshape(-1,1)
y = cal_df['MedHouseVal'].values

# K-folds
kf = KFold(n_splits=k, shuffle=True, random_state=1)

# Store outputs
coefs = []
intercepts = []

# Loop thru KFolds split
for idx_train, idx_test in kf.split(X):
    # Split the data into training and test sets for this fold
    X_train, X_test = X[idx_train], X[idx_test]
    y_train, y_test = y[idx_train], y[idx_test]

    # Fit the model on the training data
    model_kf_1.fit(X_train, y_train)

    # Store the coefficients and intercept for this fold
    coefs.append(model_kf_1.coef_)
    intercepts.append(model_kf_1.intercept_)


# Now Plot it
plt.scatter(cal_df['MedInc'], cal_df['MedHouseVal'], c='black', alpha = .3)

for i, (a, b) in enumerate(zip(intercepts, coefs)):
    y_kf_sample = a + b * X_sample
    plt.plot(X_sample, y_kf_sample, label = f'k = {i}', c = 'lightblue')

plt.plot(X_sample, y_sample, c='red', label = 'full dataset')
plt.title('Income vs. House Value (K-folds)')
plt.xlabel('Median Income')
plt.ylabel('Median House Value')
plt.legend()

plt.show()
```



Enter your observations here

They all seem about the same, to the point we can't even really see the the k-folds ones

Part 2: Gradient descent: Linear Regression

This is where it gets fun!

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.*

In [ ]:
```
import time

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

Parameters
----------
x_values, y_values : narray
    x_values: independent variable
    y_values: dependent variable

R: float
    Learning rate

MaxIterations: Int
    maximum number of iterations

epsilon = .0001: float
    convergence criteria

alpha_init=0, beta_init=1:
    initial values for


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

```
beta: float
    coefficient
"""
def bivariate_ols(x_values, y_values, R=0.01, MaxIterations=1000, epsilon=.0
    start_time = time.time()

    # Checks
    assert x_values.shape[0] == np.size(y_values)
    n = np.size(y_values)

    # Intialize storage arrays
    cost_storage = np.zeros(MaxIterations)

    # Initialize default params
    alpha = alpha_init
    beta = beta_init

    i = 0
    while(i < MaxIterations):
        # Predict
        y_pred = alpha + beta * x_values

        # Calculate Gradients
        alpha_grad = np.sum(y_pred - y_values) / n
        beta_grad = np.sum((y_pred - y_values) * x_values) / n

        # Update params
        alpha = alpha - R * alpha_grad
        beta = beta - R * beta_grad

        # Store costs
        y_pred_new = alpha + beta * x_values
        cost_storage[i] = np.sum((y_pred_new - y)**2)/(2 * n)

        # Stop Condition
        if(i > 0 and np.abs(cost_storage[i] - cost_storage[i-1]) < epsilon):
            print(f'Breaking after {i} iterations')
            break
        else:
            i = i + 1

    # if don't converge, find cost-minimizing id?
    lowest_cost_idx = np.argmin(cost_storage)

    print("Time taken: {:.2f} seconds".format(time.time() - start_time))

    return alpha, beta

# Implement the batch gradient descent algorithm that we discussed in class.
# Use the version you implement to regress the median house value on the med
# Experiment with 3-4 different values of the learning rate *R*, and do the 
bols_r_0001 = bivariate_ols(cal_df['MedInc'].values, cal_df['MedHouseVal'].v
bols_r_01 = bivariate_ols(cal_df['MedInc'].values, cal_df['MedHouseVal'].val
bols_r_05 = bivariate_ols(cal_df['MedInc'].values, cal_df['MedHouseVal'].val
bols_r_2 = bivariate_ols(cal_df['MedInc'].values, cal_df['MedHouseVal'].valu
print(f"R=.0001 --\talpha: {bols_r_0001[0]:.4f}\tbeta: {bols_r_0001[1]:.4f}"
print(f"R=.01 --\talpha: {bols_r_01[0]:.4f}\tbeta: {bols_r_01[1]:.4f}")
print(f"R=.05 --\talpha: {bols_r_05[0]:.4f}\tbeta: {bols_r_05[1]:.4f}")
print(f"R=.2 --\talpha: {bols_r_2[0]:.4f}\tbeta: {bols_r_2[1]:.4f}")

coefs
```

```
Time taken: 0.10 seconds
Breaking after 501 iterations
Time taken: 0.04 seconds
Breaking after 274 iterations
Time taken: 0.02 seconds
Breaking after 667 iterations
Time taken: 0.05 seconds
R=.0001 --      alpha: -0.0725  beta: 0.6016
R=.01 --        alpha: 0.5197   beta: 0.3995
R=.05 --        alpha: 1.1055   beta: 0.2691
R=.2 -- alpha: nan      beta: nan
```

```
/Users/jon/anaconda3/envs/aml/lib/python3.12/site-packages/numpy/core/_metho
ds.py:49: RuntimeWarning: overflow encountered in reduce
  return umr_sum(a, axis, dtype, out, keepdims, initial, where)
/var/folders/t8/dm9l8xy95mv0d_75b2m8r5nw0000gn/T/ipykernel_77025/1307543738.
py:68: RuntimeWarning: invalid value encountered in scalar subtract
  if(i > 0 and np.abs(cost_storage[i] - cost_storage[i-1]) < epsilon):
/Users/jon/anaconda3/envs/aml/lib/python3.12/site-packages/numpy/core/fromnu
meric.py:88: RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/var/folders/t8/dm9l8xy95mv0d_75b2m8r5nw0000gn/T/ipykernel_77025/1307543738.
py:61: RuntimeWarning: invalid value encountered in scalar subtract
  beta = beta - R * beta_grad
```

```
Out[ ]: array([-0.10678359,  0.03153349])
```

Enter your observations here

- Report the values of alpha and beta that minimize the loss function

(see above)

- Report the number of iterations it takes for your algorithm to converge (for each value of *R*)

(see print output)

- Report the total running time of your algorithm, in seconds

(see print output)

- How do your coefficients compare to the ones estimated through standard libraries in 1.1? Does this depend on *R*?

In 1.1, we saw alpha=-.1067 and beta=.0315. These values are closest to the iteration from R=.0001, which implies that the optimal learning rate may be on the lower end here.

The coefficients are still different, so we can't say whether it is too large or too small or even that close, but it clearly seems to imply that a learning rate lower than .01 might be best

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.

In [ ]:
```
'''
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)
```

2.3 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*

In [ ]:
```
"""
Function
--------
multivariate_ols
    Gradient Decent to minimize OLS. Used to find coefficients of bivariate

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

R: float
    Learning rate

MaxIterations: Int
    maximum number of iterations


Returns
-------
beta: array[float]
    coefficients including intercept as first value
"""

def multivariate_ols(xvalue_matrix, yvalues, R=0.01, MaxIterations=1000, eps
    start_time = time.time()
    # your code here

    # Get Size
    assert xvalue_matrix.shape[0] == np.size(yvalues)
    n = np.size(yvalues)
    n_betas = xvalue_matrix.shape[1] + 1

    # Add Intercept
    xvalue_matrix = np.column_stack((np.ones(n), xvalue_matrix))

    # Initialize betas
    beta = np.repeat(beta_init_value, n_betas)

    # Store values
    cost_storage = np.zeros(MaxIterations)
    beta_storage = np.zeros((MaxIterations, n_betas))

    # Run
    i = 0
    while(i < MaxIterations):
        # Predict
        y_hat = np.dot(xvalue_matrix, beta)

        # Calculate Gradient
        gradient = np.dot(xvalue_matrix.T, y_hat - yvalues)/n

        # Update
        beta = beta - R * gradient
        beta_storage[i] = beta

        # Store Cost
        cost_storage[i] = np.sum(np.dot(xvalue_matrix, beta)**2) /(2*n)

        # Check stop condition
        if(i > 0 and np.abs(cost_storage[i] - cost_storage[i-1]) < epsilon):
            print(f"Iterations Required: {i}\nTime taken: {time.time() - sta
            return beta

        # Update i
        i = i+1

    # if it made it through MaxIterations, then choose whichever has the low
    min_idx = np.argmin(cost_storage,)
    beta = beta_storage[min_idx]

    print(f"Iterations Required: {i}\nTime taken: {time.time() - start_time:
    return beta


#multivariate_ols(cal_df[['MedInc', 'AveRooms']].values, cal_df['MedHouseVal
#np.ones(np.size(cal_df['MedHouseVal']))
#cal_df[['MedInc', 'AveRooms']].values

# Set Test Cols
test_cols = ['MedInc', 'HouseAge','AveRooms']
target_col = 'MedHouseVal'

# Separate X, y
X = cal_df[test_cols]
y = cal_df[target_col]

# Normalize Cols
X_norm = standardize(X, X)

output = multivariate_ols(X_norm.values, y.values, R=.01, MaxIterations=1000

print(f'output\n\tintrcpt: {output[0]}\n\tcoefs: \t{output[1:]}')
```

```
Iterations Required: 1374
Time taken: 0.20 seconds
output
        intrcpt: 2.245870234763479
        coefs:  [ 0.25444345  0.08669804 -0.03084263]
```

In [ ]:
```
# Now lets redo it with the sm
X_norm2 = X_norm
X_norm2 = sm.add_constant(X_norm2)

est = sm.OLS(y.values, X_norm2.values).fit()

est.summary()
```

Out[ ]:

OLS Regression Results

Dep. Variable:	y	**R-squared:**	0.188
Model:	OLS	**Adj. R-squared:**	0.188
Method:	Least Squares	**F-statistic:**	809.2
Date:	Fri, 08 Mar 2024	**Prob (F-statistic):**	0.00
Time:	14:17:56	**Log-Likelihood:**	-7436.9
No. Observations:	10484	**AIC:**	1.488e+04
Df Residuals:	10480	**BIC:**	1.491e+04
Df Model:	3		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	2.2459	0.005	467.446	0.000	2.236	2.255
x1	0.2545	0.005	47.713	0.000	0.244	0.265
x2	0.0867	0.005	17.276	0.000	0.077	0.097
x3	-0.0309	0.005	-5.922	0.000	-0.041	-0.021

Omnibus:	587.391	**Durbin-Watson:**	2.027
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	691.299
Skew:	0.629	**Prob(JB):**	7.70e-151
Kurtosis:	2.954	**Cond. No.**	1.61

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Enter your observations here

After a lot of fiddling with the parameters in my implementation, I was able to get the values to line up almost exactly.

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.

In [ ]:
```
# Your code here

# Add Intercept
X2 = X
X2 = sm.add_constant(X2)
sm_model_nonnorm = sm.OLS(y.values, X2.values).fit()

#display(sm_model_nonnorm.summary())

coefs__sm = sm_model_nonnorm.params

# R = .1 - go down to a more reasonable epsilon and MaxIterations
coefs__gd_R1 = multivariate_ols(X.values, y.values, R=.1, MaxIterations=1000
# goes to inf

# R = .01
coefs__gd_R01 = multivariate_ols(X.values, y.values, R=.01, MaxIterations=10

# R = .001
coefs__gd_R001 = multivariate_ols(X.values, y.values, R=.001, MaxIterations=

print(coefs__sm)
print(coefs__gd_R1)
print(coefs__gd_R01)
print(coefs__gd_R001)
```

```
/Users/jon/anaconda3/envs/aml/lib/python3.12/site-packages/numpy/core/fromnu
meric.py:88: RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/var/folders/t8/dm9l8xy95mv0d_75b2m8r5nw0000gn/T/ipykernel_77025/144152375.p
y:59: RuntimeWarning: overflow encountered in square
  cost_storage[i] = np.sum(np.dot(xvalue_matrix, beta)**2) /(2*n)
/var/folders/t8/dm9l8xy95mv0d_75b2m8r5nw0000gn/T/ipykernel_77025/144152375.p
y:62: RuntimeWarning: invalid value encountered in scalar subtract
  if(i > 0 and np.abs(cost_storage[i] - cost_storage[i-1]) < epsilon):
/var/folders/t8/dm9l8xy95mv0d_75b2m8r5nw0000gn/T/ipykernel_77025/144152375.p
y:55: RuntimeWarning: invalid value encountered in subtract
  beta = beta - R * gradient
Iterations Required: 10000
Time taken: 1.24 seconds
Iterations Required: 10000
Time taken: 1.21 seconds
Iterations Required: 1736
Time taken: 0.23 seconds
[ 1.34200301  0.19077902  0.00698801 -0.01350345]
[-inf -inf  nan -inf]
[-inf -inf  nan -inf]
[0.12905922 0.31453696 0.02185527 0.03016742]
```

Enter your observations here

They take much longer to converge and are much less likely to do so in a standard amount of time. When it did converge, the results are fairly different which suggests it needs many more iterations to find the convergence point, and that the function does not follow as much of a linear relationship as the standardized versions

3. Prediction

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

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.

In [ ]:
```
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
K = 5

kf = KFold(n_splits=K)

# Instantiate Model
models_kf = LinearRegression()

# create storage
coefs__LR = np.zeros((K, X_norm.shape[1] + 1)) # For the built-in Linear Reg
coefs__GD = np.zeros((K, X_norm.shape[1] + 1)) # For my implementation of Gr

i = 0
for idx_train, idx_test in kf.split(X_norm.values):
    # Split the data into training and test sets for this fold
    X_train, X_test = X_norm.values[idx_train], X_norm.values[idx_test]
    y_train, y_test = y.values[idx_train], y.values[idx_test]

    # Linear Regression
    # Fit the model on the training data
    models_kf.fit(X_train, y_train)
    # Store the coefficients and intercept for this fold
    LR_coefs = np.append(models_kf.intercept_, models_kf.coef_)
    coefs__LR[i] = LR_coefs

    # Gradient Descent
    GD_coefs = multivariate_ols(X_train, y_train, R=.01, MaxIterations=10000
    coefs__GD[i] = GD_coefs

    i=i+1

display(pd.DataFrame(coefs__GD).rename(columns = lambda x: f'b{x}'))
```

```
Iterations Required: 1382
Time taken: 0.20 seconds
Iterations Required: 1507
Time taken: 0.21 seconds
Iterations Required: 1360
Time taken: 0.19 seconds
Iterations Required: 1363
Time taken: 0.19 seconds
Iterations Required: 1369
Time taken: 0.19 seconds
```

	b0	b1	b2	b3
0	2.247110	0.252201	0.087007	-0.030311
1	2.246911	0.260782	0.086502	-0.052491
2	2.247057	0.253123	0.084969	-0.027762
3	2.246057	0.253713	0.086914	-0.028335
4	2.242012	0.256149	0.087180	-0.026035

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?

In [ ]:
```
# Your code here
rmse_save = []

for i, idxs in enumerate(kf.split(X_norm.values)):
    # Get idx
    idx_train, idx_test = idxs

    # Set X/y
    X_train, X_test = X_norm.values[idx_train], X_norm.values[idx_test]
    y_train, y_test = y.values[idx_train], y.values[idx_test]

    assert X_test.shape[0] == np.size(y_test)
    n = np.size(y_test)

    # Get Coefs
    beta = coefs__GD[i]

    # Add Intercept
    X_test = np.column_stack((np.ones(n), X_test))

    # Predict
    y_hat = np.dot(X_test, beta)

    # Calculate Error
    err = compute_rmse(y_hat, y_test)

    rmse_save.append(err)

print(f'RMSEs {rmse_save}')
print(f'mean: {np.mean(rmse_save)}')
```

```
RMSEs [0.4812479053332355, 0.49877227651856076, 0.4880438908463683, 0.492001
96661073703, 0.5025827423080966]
mean: 0.4925297563233997
```

Discuss your results here

the RMSE of .4925 is significantly lower than the .768 from PS3.

4 Regularization

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.

In [ ]:
```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import train_test_split
# Your code here

# Step 1:

# Degree > 0, <= 3, means MedInc, HouseAge, AveRooms, MedInc * HouseAge, Mec
pf = PolynomialFeatures(degree=(1,3), include_bias=False)
```

```
X_pf = pf.fit_transform(X)
X_pf_norm = standardize(X_pf, X_pf)
X_pf_norm__df = pd.DataFrame(X_pf_norm, columns=pf.get_feature_names_out())


# Step 2:
X_pf_norm__train, X_pf_norm__test, y__train, y__test = train_test_split(X_pf
```

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?

In [ ]:
```
# Your code here
# X_pf_norm__df = pd.DataFrame(X_pf_norm, columns=pf.get_feature_names_out()

coefs = multivariate_ols(X_pf_norm__train, y__train, MaxIterations=10000, be

# Add intercept
X_pf_norm__train_plus_int = np.column_stack((np.ones(X_pf_norm__train.shape[

y_hat__train = np.dot(X_pf_norm__train_plus_int, coefs)

print(f"coefs: {coefs}")
print(f"train RMSE: {compute_rmse(y_hat__train, y__train):.4f}") # .4923

# Test Set
# Add intercept
X_pf_norm__test_plus_int = np.column_stack((np.ones(X_pf_norm__test.shape[0]

y_hat__test = np.dot(X_pf_norm__test_plus_int, coefs)

print(f"coefs: {coefs}")
print(f"test RMSE: {compute_rmse(y_hat__test, y__test):.4f}") # .5053
```

```
Iterations Required: 1349
Time taken: 0.25 seconds
coefs: [ 2.24714555  0.02523061  0.04252389  0.04252389  0.04196147  0.02807
716
  0.04196147  0.02807716  0.00446484 -0.02543229]
train RMSE: 0.4923
coefs: [ 2.24714555  0.02523061  0.04252389  0.04252389  0.04196147  0.02807
716
  0.04196147  0.02807716  0.00446484 -0.02543229]
test RMSE: 0.5053
```

Discuss your results here

This result doesn't seem unexpected to me. The RMSE on the train set (`.4923`) is slightly better than the test set (`.5053`). I guess I could have expected a bit more overfitting with the increased complexity of the terms allowing for more tailoring of the model to the idosyncracies of the training set, but that doesn't really seem to be happening here as far as I can tell

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.

In [ ]:
```
def multivariate_regularized_ols(xvalue_matrix, yvalues, R=0.01, MaxIteratio
    start_time = time.time()
    # Your code here

    # Get Size
    assert xvalue_matrix.shape[0] == np.size(yvalues)
    n = np.size(yvalues)
    n_betas = xvalue_matrix.shape[1]

    # Add Intercept
    if(add_intercept):
        xvalue_matrix = np.column_stack((np.ones(n), xvalue_matrix))
        n_betas = n_betas + 1

    # Initialize betas
    beta = np.repeat(beta_init_value, n_betas)

    # Store values
    cost_storage = np.zeros(MaxIterations)
    beta_storage = np.zeros((MaxIterations, n_betas))

    # Run
    i = 1
    while(i < MaxIterations):
        # Predict
        y_hat = np.dot(xvalue_matrix, beta)

        # Calculate Gradient
        gradient = (np.dot(xvalue_matrix.T, (y_hat - yvalues)) + lmbda * np.

        # Update
        beta = beta - R * gradient
        beta_storage[i] = beta

        # Store Cost
        cost_storage[i] = (np.sum(np.dot(xvalue_matrix, beta)**2) + (lmbda *

        # Check stop condition
        if(i > 0 and np.abs(cost_storage[i] - cost_storage[i-1]) < epsilon):
            print(f"Iterations Required: {i}\nTime taken: {time.time() - sta
            return beta

        # Update i
        i = i+1

    # if it made it through MaxIterations, then choose whichever has the low
    min_idx = np.argmin(cost_storage)
    beta = beta_storage[min_idx]

    print("\nMax iterations required -- Time taken: {:.2f} seconds".format(t
    return beta[0], beta[1:]

# Run it
coefs_ridge = multivariate_regularized_ols(X_pf_norm__train, y__train, MaxIt
coefs_ridge

# Predict
y_hat_ridge__train = np.dot(X_pf_norm__train_plus_int, coefs_ridge)

# Compute RMSE
rmse_train = compute_rmse(y_hat_ridge__train, y__train) # .4839

# Predict test set
y_hat_ridge__test = np.dot(X_pf_norm__test_plus_int, coefs_ridge)

# Compute RMSE
rmse_test = compute_rmse(y_hat_ridge__test, y__test) # .4839


print(f"train RMSE: {rmse_train:.4f}")
print(f"test RMSE:  {rmse_test:.4f}")
```

```
Iterations Required: 850
Time taken: 0.17 seconds
train RMSE: 0.4939
test RMSE:  0.5069
```

Discuss your results here

The results seem about the same. The Ridge regularization doesn't seem to be doing too much here

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!

In [ ]:
```
# Your code here

# Set of possible lambdas
lmbdas = 2.0**np.arange(-3,13)
print(lmbdas)

k = 5
#model_kf = LinearRegression()
kf44 = KFold(n_splits=k, shuffle=True, random_state=1)

# Using the 80% of the data set aside for training, iterate thru the values
lmbdas_storage = {}

for l in lmbdas:
    # for each value of lambda, use k-fold cross-validation to compute the a
    # computed as the average across the held-out folds.

    rmses_substorage = {}
    rmses_substorage['train'] = []
    rmses_substorage['test'] = []
    for idx_train, idx_test in kf44.split(X_pf_norm__train):
        # Split the data into training and test sets for this fold
        X_train_44, X_test_44 = X_pf_norm__train[idx_train], X_pf_norm__trai
        y_train_44, y_test_44 = y__train[idx_train], y__train[idx_test]

        # Fit Model
        coefs_44 = multivariate_regularized_ols(X_train_44, y_train_44, MaxI

        # Predict Train
        X_train_44_plus_int = np.column_stack((np.ones(X_train_44.shape[0]),
        y_hat_train = np.dot(X_train_44_plus_int, coefs_44)

        # Compute RMSE
        rmses_substorage['train'].append(compute_rmse(y_hat_train, y_train_4
        # Predict Test
        X_test_44_plus_int = np.column_stack((np.ones(X_test_44.shape[0]), X
        y_hat_test = np.dot(X_test_44_plus_int, coefs_44)

        # Compute RMSE
        rmses_substorage['test'].append(compute_rmse(y_hat_test, y_test_44))

    lmbdas_storage[l] = rmses_substorage
```

```
[1.250e-01 2.500e-01 5.000e-01 1.000e+00 2.000e+00 4.000e+00 8.000e+00
 1.600e+01 3.200e+01 6.400e+01 1.280e+02 2.560e+02 5.120e+02 1.024e+03
 2.048e+03 4.096e+03]
Iterations Required: 853
Time taken: 0.23 seconds
Iterations Required: 851
Time taken: 0.17 seconds
Iterations Required: 851
Time taken: 0.25 seconds
Iterations Required: 852
Time taken: 0.18 seconds
Iterations Required: 850
Time taken: 0.18 seconds
Iterations Required: 853
Time taken: 0.18 seconds
Iterations Required: 851
Time taken: 0.17 seconds
Iterations Required: 851
Time taken: 0.18 seconds
Iterations Required: 852
Time taken: 0.25 seconds
Iterations Required: 850
Time taken: 0.19 seconds
Iterations Required: 853
Time taken: 0.19 seconds
Iterations Required: 851
Time taken: 0.17 seconds
Iterations Required: 851
Time taken: 0.18 seconds
Iterations Required: 852
Time taken: 0.21 seconds
Iterations Required: 850
Time taken: 0.18 seconds
Iterations Required: 853
Time taken: 0.25 seconds
Iterations Required: 851
Time taken: 0.19 seconds
Iterations Required: 851
Time taken: 0.18 seconds
Iterations Required: 852
Time taken: 0.21 seconds
Iterations Required: 850
Time taken: 0.30 seconds
Iterations Required: 853
Time taken: 0.19 seconds
Iterations Required: 851
Time taken: 0.16 seconds
Iterations Required: 851
Time taken: 0.27 seconds
Iterations Required: 852
Time taken: 0.17 seconds
Iterations Required: 850
Time taken: 0.20 seconds
Iterations Required: 853
Time taken: 0.20 seconds
Iterations Required: 851
```

```
Time taken: 0.27 seconds
Iterations Required: 851
Time taken: 0.17 seconds
Iterations Required: 852
Time taken: 0.20 seconds
Iterations Required: 850
Time taken: 0.21 seconds
Iterations Required: 853
Time taken: 0.19 seconds
Iterations Required: 851
Time taken: 0.26 seconds
Iterations Required: 851
Time taken: 0.21 seconds
Iterations Required: 852
Time taken: 0.21 seconds
Iterations Required: 850
Time taken: 0.19 seconds
Iterations Required: 853
Time taken: 0.27 seconds
Iterations Required: 851
Time taken: 0.19 seconds
Iterations Required: 851
Time taken: 0.20 seconds
Iterations Required: 852
Time taken: 0.25 seconds
Iterations Required: 850
Time taken: 0.20 seconds
Iterations Required: 853
Time taken: 0.17 seconds
Iterations Required: 851
Time taken: 0.17 seconds
Iterations Required: 851
Time taken: 0.26 seconds
Iterations Required: 852
Time taken: 0.17 seconds
Iterations Required: 850
Time taken: 0.17 seconds
Iterations Required: 853
Time taken: 0.19 seconds
Iterations Required: 851
Time taken: 0.28 seconds
Iterations Required: 851
Time taken: 0.18 seconds
Iterations Required: 852
Time taken: 0.20 seconds
Iterations Required: 850
Time taken: 0.22 seconds
Iterations Required: 853
Time taken: 0.28 seconds
Iterations Required: 851
Time taken: 0.19 seconds
Iterations Required: 851
Time taken: 0.21 seconds
Iterations Required: 852
Time taken: 0.24 seconds
Iterations Required: 850
```

```
Time taken: 0.18 seconds
Iterations Required: 853
Time taken: 0.17 seconds
Iterations Required: 851
Time taken: 0.26 seconds
Iterations Required: 851
Time taken: 0.23 seconds
Iterations Required: 851
Time taken: 0.18 seconds
Iterations Required: 850
Time taken: 0.18 seconds
Iterations Required: 852
Time taken: 0.29 seconds
Iterations Required: 851
Time taken: 0.16 seconds
Iterations Required: 850
Time taken: 0.18 seconds
Iterations Required: 851
Time taken: 0.28 seconds
Iterations Required: 850
Time taken: 0.17 seconds
Iterations Required: 851
Time taken: 0.22 seconds
Iterations Required: 851
Time taken: 0.21 seconds
Iterations Required: 850
Time taken: 0.27 seconds
Iterations Required: 851
Time taken: 0.17 seconds
Iterations Required: 850
Time taken: 0.17 seconds
Iterations Required: 851
Time taken: 0.28 seconds
Iterations Required: 850
Time taken: 0.19 seconds
Iterations Required: 850
Time taken: 0.20 seconds
Iterations Required: 850
Time taken: 0.30 seconds
Iterations Required: 850
Time taken: 0.19 seconds
Iterations Required: 850
Time taken: 0.18 seconds
Iterations Required: 850
Time taken: 0.20 seconds
Iterations Required: 850
Time taken: 0.26 seconds
Iterations Required: 850
Time taken: 0.18 seconds
Iterations Required: 850
Time taken: 0.19 seconds
```

In [ ]:
```
#pd.DataFrame(lmbdas_storage[.125])
pd.DataFrame(np.stack([pd.DataFrame(lmbdas_storage[d]).mean() for d in lmbda
# They're all basically identical, regardless of lambda value
# # Lets try again with values between 1 and 4?
```

Out[ ]:

	train	test
0.125	0.492377	0.492620
0.250	0.492377	0.492620
0.500	0.492377	0.492620
1.000	0.492377	0.492620
2.000	0.492377	0.492620
4.000	0.492377	0.492620
8.000	0.492378	0.492621
16.000	0.492379	0.492622
32.000	0.492381	0.492623
64.000	0.492385	0.492627
128.000	0.492394	0.492634
256.000	0.492413	0.492649
512.000	0.492453	0.492684
1024.000	0.492540	0.492761
2048.000	0.492721	0.492927
4096.000	0.493084	0.493273

In [ ]:

```
lmbdas = np.arange(1,4, .1)
print(lmbdas)

# Using the 80% of the data set aside for training, iterate thru the values

for l in lmbdas:
    # for each value of lambda, use k-fold cross-validation to compute the a
    # computed as the average across the held-out folds.

    rmses_substorage = {}
    rmses_substorage['train'] = []
    rmses_substorage['test'] = []
    for idx_train, idx_test in kf44.split(X_pf_norm__train):
        # Split the data into training and test sets for this fold
        X_train_44, X_test_44 = X_pf_norm__train[idx_train], X_pf_norm__trai
        y_train_44, y_test_44 = y__train[idx_train], y__train[idx_test]

        # Fit Model
        coefs_44 = multivariate_regularized_ols(X_train_44, y_train_44, MaxI

        # Predict Train
        X_train_44_plus_int = np.column_stack((np.ones(X_train_44.shape[0]),
        y_hat_train = np.dot(X_train_44_plus_int, coefs_44)
        # Compute RMSE
        rmses_substorage['train'].append(compute_rmse(y_hat_train, y_train_4

        # Predict Test
        X_test_44_plus_int = np.column_stack((np.ones(X_test_44.shape[0]), X
        y_hat_test = np.dot(X_test_44_plus_int, coefs_44)

        # Compute RMSE
        rmses_substorage['test'].append(compute_rmse(y_hat_test, y_test_44))

    lmbdas_storage[l] = rmses_substorage
```

```
[1.  1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.  2.1 2.2 2.3 2.4 2.5 2.6 2.7
 2.8 2.9 3.  3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9]
Iterations Required: 853
Time taken: 0.26 seconds
Iterations Required: 851
Time taken: 0.17 seconds
Iterations Required: 851
Time taken: 0.19 seconds
Iterations Required: 852
Time taken: 0.25 seconds
Iterations Required: 850
Time taken: 0.18 seconds
Iterations Required: 853
Time taken: 0.18 seconds
Iterations Required: 851
Time taken: 0.18 seconds
Iterations Required: 851
Time taken: 0.21 seconds
Iterations Required: 852
Time taken: 0.28 seconds
Iterations Required: 850
Time taken: 0.18 seconds
Iterations Required: 853
Time taken: 0.19 seconds
Iterations Required: 851
Time taken: 0.16 seconds
Iterations Required: 851
Time taken: 0.22 seconds
Iterations Required: 852
Time taken: 0.21 seconds
Iterations Required: 850
Time taken: 0.19 seconds
Iterations Required: 853
Time taken: 0.26 seconds
Iterations Required: 851
Time taken: 0.19 seconds
Iterations Required: 851
Time taken: 0.21 seconds
Iterations Required: 852
Time taken: 0.19 seconds
Iterations Required: 850
Time taken: 0.25 seconds
Iterations Required: 853
Time taken: 0.19 seconds
Iterations Required: 851
Time taken: 0.17 seconds
Iterations Required: 851
Time taken: 0.24 seconds
Iterations Required: 852
Time taken: 0.18 seconds
Iterations Required: 850
Time taken: 0.17 seconds
Iterations Required: 853
Time taken: 0.17 seconds
Iterations Required: 851
Time taken: 0.24 seconds
```

```
Iterations Required: 851
Time taken: 0.17 seconds
Iterations Required: 852
Time taken: 0.16 seconds
Iterations Required: 850
Time taken: 0.19 seconds
Iterations Required: 853
Time taken: 0.18 seconds
Iterations Required: 851
Time taken: 0.23 seconds
Iterations Required: 851
Time taken: 0.16 seconds
Iterations Required: 852
Time taken: 0.17 seconds
Iterations Required: 850
Time taken: 0.18 seconds
Iterations Required: 853
Time taken: 0.23 seconds
Iterations Required: 851
Time taken: 0.18 seconds
Iterations Required: 851
Time taken: 0.18 seconds
Iterations Required: 852
Time taken: 0.24 seconds
Iterations Required: 850
Time taken: 0.22 seconds
Iterations Required: 853
Time taken: 0.18 seconds
Iterations Required: 851
Time taken: 0.18 seconds
Iterations Required: 851
Time taken: 0.17 seconds
Iterations Required: 852
Time taken: 0.22 seconds
Iterations Required: 850
Time taken: 0.29 seconds
Iterations Required: 853
Time taken: 0.27 seconds
Iterations Required: 851
Time taken: 0.25 seconds
Iterations Required: 851
Time taken: 0.24 seconds
Iterations Required: 852
Time taken: 0.24 seconds
Iterations Required: 850
Time taken: 0.19 seconds
Iterations Required: 853
Time taken: 0.27 seconds
Iterations Required: 851
Time taken: 0.20 seconds
Iterations Required: 851
Time taken: 0.21 seconds
Iterations Required: 852
Time taken: 0.31 seconds
Iterations Required: 850
Time taken: 0.20 seconds
```

```
Iterations Required: 853
Time taken: 0.21 seconds
Iterations Required: 851
Time taken: 0.19 seconds
Iterations Required: 851
Time taken: 0.29 seconds
Iterations Required: 852
Time taken: 0.18 seconds
Iterations Required: 850
Time taken: 0.19 seconds
Iterations Required: 853
Time taken: 0.26 seconds
Iterations Required: 851
Time taken: 0.17 seconds
Iterations Required: 851
Time taken: 0.24 seconds
Iterations Required: 852
Time taken: 0.27 seconds
Iterations Required: 850
Time taken: 0.18 seconds
Iterations Required: 853
Time taken: 0.17 seconds
Iterations Required: 851
Time taken: 0.22 seconds
Iterations Required: 851
Time taken: 0.21 seconds
Iterations Required: 852
Time taken: 0.18 seconds
Iterations Required: 850
Time taken: 0.16 seconds
Iterations Required: 853
Time taken: 0.26 seconds
Iterations Required: 851
Time taken: 0.17 seconds
Iterations Required: 851
Time taken: 0.17 seconds
Iterations Required: 852
Time taken: 0.16 seconds
Iterations Required: 850
Time taken: 0.23 seconds
Iterations Required: 853
Time taken: 0.17 seconds
Iterations Required: 851
Time taken: 0.18 seconds
Iterations Required: 851
Time taken: 0.17 seconds
Iterations Required: 852
Time taken: 0.24 seconds
Iterations Required: 850
Time taken: 0.17 seconds
Iterations Required: 853
Time taken: 0.18 seconds
Iterations Required: 851
Time taken: 0.25 seconds
Iterations Required: 851
Time taken: 0.17 seconds
```

```
Iterations Required: 852
Time taken: 0.16 seconds
Iterations Required: 850
Time taken: 0.20 seconds
Iterations Required: 853
Time taken: 0.21 seconds
Iterations Required: 851
Time taken: 0.16 seconds
Iterations Required: 851
Time taken: 0.17 seconds
Iterations Required: 852
Time taken: 0.22 seconds
Iterations Required: 850
Time taken: 0.18 seconds
Iterations Required: 853
Time taken: 0.19 seconds
Iterations Required: 851
Time taken: 0.17 seconds
Iterations Required: 851
Time taken: 0.24 seconds
Iterations Required: 852
Time taken: 0.17 seconds
Iterations Required: 850
Time taken: 0.17 seconds
Iterations Required: 853
Time taken: 0.26 seconds
Iterations Required: 851
Time taken: 0.17 seconds
Iterations Required: 851
Time taken: 0.18 seconds
Iterations Required: 852
Time taken: 0.19 seconds
Iterations Required: 850
Time taken: 0.29 seconds
Iterations Required: 853
Time taken: 0.18 seconds
Iterations Required: 851
Time taken: 0.18 seconds
Iterations Required: 851
Time taken: 0.25 seconds
Iterations Required: 852
Time taken: 0.17 seconds
Iterations Required: 850
Time taken: 0.21 seconds
Iterations Required: 853
Time taken: 0.26 seconds
Iterations Required: 851
Time taken: 0.20 seconds
Iterations Required: 851
Time taken: 0.18 seconds
Iterations Required: 852
Time taken: 0.26 seconds
Iterations Required: 850
Time taken: 0.23 seconds
Iterations Required: 853
Time taken: 0.19 seconds
```

```
Iterations Required: 851
Time taken: 0.19 seconds
Iterations Required: 851
Time taken: 0.30 seconds
Iterations Required: 852
Time taken: 0.17 seconds
Iterations Required: 850
Time taken: 0.17 seconds
Iterations Required: 853
Time taken: 0.26 seconds
Iterations Required: 851
Time taken: 0.22 seconds
Iterations Required: 851
Time taken: 0.19 seconds
Iterations Required: 852
Time taken: 0.18 seconds
Iterations Required: 850
Time taken: 0.24 seconds
Iterations Required: 853
Time taken: 0.18 seconds
Iterations Required: 851
Time taken: 0.17 seconds
Iterations Required: 851
Time taken: 0.27 seconds
Iterations Required: 852
Time taken: 0.17 seconds
Iterations Required: 850
Time taken: 0.19 seconds
Iterations Required: 853
Time taken: 0.18 seconds
Iterations Required: 851
Time taken: 0.28 seconds
Iterations Required: 851
Time taken: 0.19 seconds
Iterations Required: 852
Time taken: 0.19 seconds
Iterations Required: 850
Time taken: 0.24 seconds
Iterations Required: 853
Time taken: 0.17 seconds
Iterations Required: 851
Time taken: 0.18 seconds
Iterations Required: 851
Time taken: 0.18 seconds
Iterations Required: 852
Time taken: 0.25 seconds
Iterations Required: 850
Time taken: 0.17 seconds
Iterations Required: 853
Time taken: 0.18 seconds
Iterations Required: 851
Time taken: 0.25 seconds
Iterations Required: 851
Time taken: 0.19 seconds
Iterations Required: 852
Time taken: 0.16 seconds
```

```
Iterations Required: 850
Time taken: 0.15 seconds
Iterations Required: 853
Time taken: 0.28 seconds
Iterations Required: 851
Time taken: 0.27 seconds
Iterations Required: 851
Time taken: 0.26 seconds
Iterations Required: 852
Time taken: 0.31 seconds
Iterations Required: 850
Time taken: 0.27 seconds
Iterations Required: 853
Time taken: 0.28 seconds
Iterations Required: 851
Time taken: 0.31 seconds
Iterations Required: 851
Time taken: 0.30 seconds
Iterations Required: 852
Time taken: 0.29 seconds
Iterations Required: 850
Time taken: 0.27 seconds
```

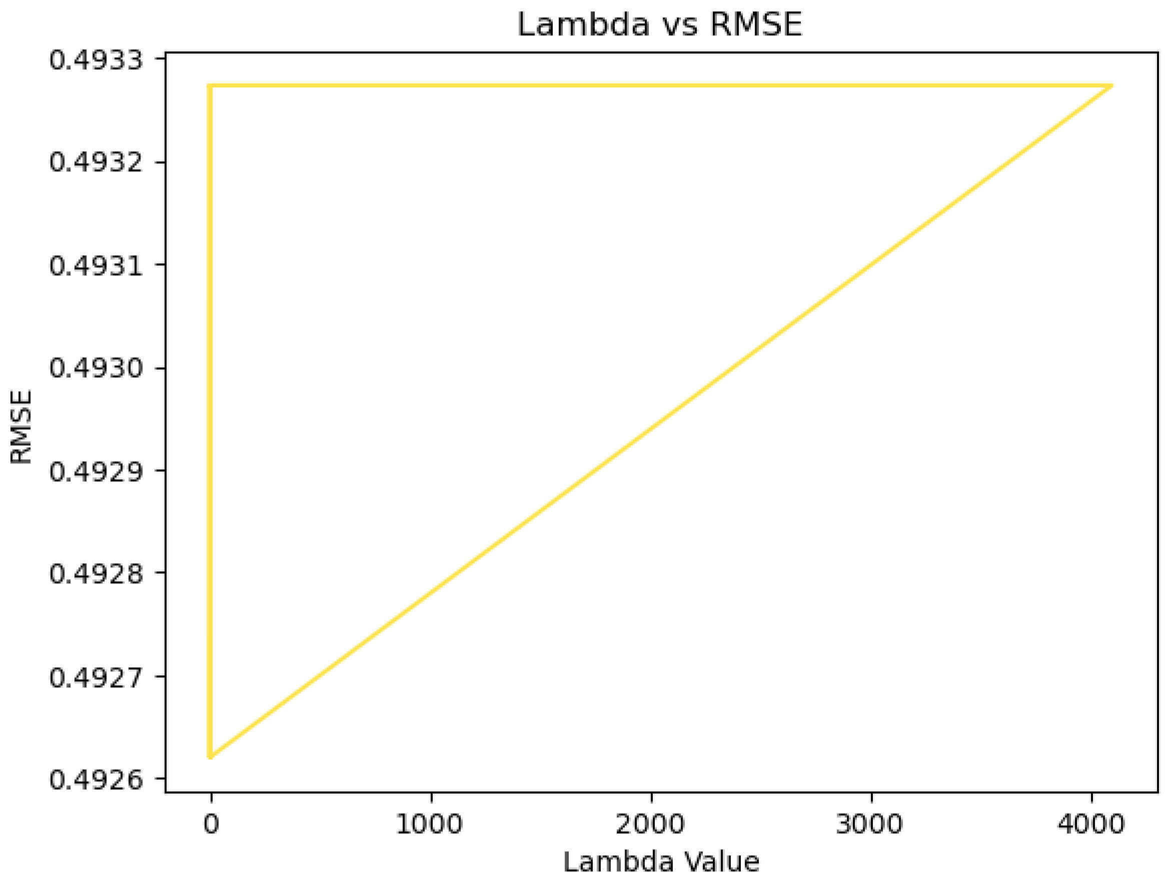
In [ ]:
```
rmses_output = pd.DataFrame(np.stack([pd.DataFrame(lmbdas_storage[d]).mean()

rmses_output
# Just really all identical, maybe I'm doing something wrong in my implement
```

Out[ ]:

	train	test
0.125	0.493084	0.493273
0.250	0.493084	0.493273
0.500	0.493084	0.493273
1.000	0.492377	0.492620
2.000	0.493084	0.493273
4.000	0.493084	0.493273
8.000	0.493084	0.493273
16.000	0.493084	0.493273
32.000	0.493084	0.493273
64.000	0.493084	0.493273
128.000	0.493084	0.493273
256.000	0.493084	0.493273
512.000	0.493084	0.493273
1024.000	0.493084	0.493273
2048.000	0.493084	0.493273
4096.000	0.493084	0.493273
1.100	0.492377	0.492620
1.200	0.492377	0.492620
1.300	0.492377	0.492620
1.400	0.492377	0.492620
1.500	0.492377	0.492620
1.600	0.492377	0.492620
1.700	0.492377	0.492620
1.800	0.492377	0.492620
1.900	0.492377	0.492620
2.000	0.492377	0.492620
2.100	0.492377	0.492620
2.200	0.492377	0.492620
2.300	0.492377	0.492620
2.400	0.492377	0.492620
2.500	0.492377	0.492620
2.600	0.492377	0.492620

	train	test
2.700	0.492377	0.492620
2.800	0.492377	0.492620
2.900	0.492377	0.492620
3.000	0.492377	0.492620
3.100	0.492377	0.492620
3.200	0.492377	0.492620
3.300	0.492377	0.492620
3.400	0.492377	0.492620
3.500	0.492377	0.492620
3.600	0.492377	0.492620
3.700	0.492377	0.492620
3.800	0.492377	0.492620
3.900	0.492377	0.492620

In [ ]:
```
# The scatter plot should have two lines: a gold line showing the cross-vali
# 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.

plt.plot(rmses_output.index.values, rmses_output['test'].values, c='gold', a

plt.title('Lambda vs RMSE')
plt.xlabel('Lambda Value')
plt.ylabel('RMSE')

plt.show()

# What value of lambda minimizes your cross-validated RMSE?
#  Fix that value of lambda, and train a new model using all of your trainir
# 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
```



Discuss your results here

This is identical to what I did above with value of lambda = 1, its the same as above. Maybe I'm implementing something wrong but I checked in office hours. As such, I'd say there is not much risk of overtraining in this model as currently set up.

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?

```
In [ ]: # Your code here
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[230], line 3
      1 # Your code here
----> 3 from sklearn.linearmodel import Ridge
      4 model = Ridge()

ModuleNotFoundError: No module named 'sklearn.linearmodel'
```

Discuss your results here

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:

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

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 [ ]:
```
# Your code here
```

Discuss your results here