databricks hw4_Workbook

HW 4 - Supervised Learning at Scale.

`MIDS w261: Machine Learning at Scale | UC Berkeley School of Information |`

In the first three homeworks you became familiar with the Map-Reduce programming paradigm as manifested in the Hadoop Streaming and Spark frameworks. We explored how different data structures and design patterns can help us manage the computational complexity of an algorithm. As part of this process you implemented both a supervised learning alogorithm (Naive Bayes) and an unsupervised learning algorithm (synonym detection via cosine similarity). In both of these tasks parallelization helped us manage calculations involving a large number of features. However a large feature space isn't the only situation that might prompt us to want to parallelize a machine learning algorithm. In the final two assignments we'll look at cases where the iterative nature of an algorithm is the main driver of its computational complexity (and the reason we might want to parallelize it).

In this week's assignment we'll perform 3 kinds of linear regression: OLS, Ridge and Lasso. As in previous assignments you will implement the core calculations using Spark RDDs... though we've provided more of a code base than before since the focus of the latter half of the course is more on general machine learning concepts. By the end of this homework you should be able to:

- ... **define** the loss functions for OLS, Ridge and Lasso regression.
- ... **calculate** the gradient for each of these loss functions.
- ... **identify** which parts of the gradient descent algorithm can be parallelized.
- ... **implement** parallelized gradient descent with cross-validation and regularization.
- ... **compare/contrast** how L1 and L2 regularization impact model parameters & performance.

Additional Reference: Spark 2.2 Documentation - RDD programming guide (https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html)

Please refer to the `README` for homework submission instructions and additional resources.

Notebook Set-Up

Before starting your homework run the following cells to confirm your setup.

```
# imports
import re
import time
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import ast
import os
```

Run the next cell to create your directory in dbfs

You do not need to understand this scala snippet. It simply dynamically fetches your user directory name so that any files you write can be saved in your own directory.

```
# RUN THIS CELL AS IS
# This code snippet reads the user directory name, and stores is in a python
variable.
# Next, it creates a folder inside your home folder, which you will use for
files which you save inside this notebook.
username =
dbutils.notebook.entry_point.getDbutils().notebook().getContext().tags().apply(
'user')
userhome = 'dbfs:/user/' + username
print(userhome)
hw4_path = userhome + "/HW4/"
hw4_path_open = '/dbfs' + hw4_path.split(':')[-1] # for use with python open()
dbutils.fs.mkdirs(hw4_path)
```

```
dbfs:/user/tonydisera@ischool.berkeley.edu
Out[248]: True
```

```
# RUN THIS CELL AS IS. You should see multiple google-eng-all-5gram-* files in
the results. If you do not see these, please let an Instructor or TA know.
sum = 0
DATA_PATH = 'dbfs:/mnt/mids-w261/data/HW4/'
for item in dbutils.fs.ls(DATA_PATH):
  sum = sum+item.size
sum
```

```
Out[249]: 348625
```

```
# RUN THIS CELL AS IS - A test to make sure your directory is working as
expected.
# You should see a result like:
# dbfs:/user/youremail@ischool.berkeley.edu/hw4/sample_docs.txt
dbutils.fs.put(hw4_path+'test.txt',"hello world",True)
display(dbutils.fs.ls(hw4_path))
```

path
dbfs:/user/tonydisera@ischool.berkeley.edu/HW4/OLSloss.csv
dbfs:/user/tonydisera@ischool.berkeley.edu/HW4/OLSmodels.csv
dbfs:/user/tonydisera@ischool.berkeley.edu/HW4/lasso_loss.txt
dbfs:/user/tonydisera@ischool.berkeley.edu/HW4/lasso_models.txt
dbfs:/user/tonydisera@ischool.berkeley.edu/HW4/ols_loss.txt
dbfs:/user/tonydisera@ischool.berkeley.edu/HW4/ols_models.txt
dbfs:/user/tonydisera@ischool.berkeley.edu/HW4/ridge_loss.txt
dbfs:/user/tonydisera@ischool.berkeley.edu/HW4/ridge_models.txt
dbfs:/user/tonydisera@ischool.berkeley.edu/HW4/test.txt

```
sc = spark.sparkContext
spark
```

```
Out[251]:
```

SparkSession - hive

SparkContext

Spark UI

Version
```
v2.4.4
```
Master
```
spark://10.74.247.18:7077
```
AppName
```
Databricks Shell
```

Question 1: Opimization Theory

As you know from w207, Gradient Descent is an iterative process that seeks to find the optimal parameters for a model given a particular training data set. It does this by using the vector of partial derivatives of a loss function to strategically update parameters in a way that will reduce the loss. In live session 6 you discussed some of the theory behnid why gradient descent works and looked at a small example of gradient descent in the context of linear regression.

Q1 Tasks:

- **a) short response:** What are the first and second order conditions for convexity and why do we care about them when performing Gradient Descent?
- **b) short response:** Explain the relationship between problem domain space and model parameter space in the context of Gradient Descent. In practice, why can't we find the optimal model by simply looking at the error surface in model parameter space?
- **c) short response:** In the context of Gradient Descent, what is the 'learning rate' and what are the tradeoffs associated with setting this hyperparameter?
- **d) BONUS:** In the context of OLS, what do we mean by a 'closed form solution' and why is it not scalable?

Q1 Student Answers:

a) The first order condition for convexity states that the objective function is above the tangent line at every point. Another way of stating this, is that the function is differentiable (we have a tangent line at every point) and that because the objective function is above its tangent line it is concave up. Expanding on this, there exists a the point where the derivative of the function (the tangent line) = 0. For the function to be convex, this critical point is a minimum. For the second order condition, this critical point is a minimum if and only if the second order derivative is greater than zero (positive). If the second derivative is positive across the entire domain, the function is strictly convex. (A strictly convex function with have only 1 global mininum.) When the first and second order conditions of of the objective functions are met, we know that gradient descent to perform optimization on the loss function. With these conditions in place, we are guaranteed that there is a way to optimize the loss function such that we will arrive at a minimum by adjusting the weights by the gradient at every iteration through the data. So the gradient calculation will tell us how much to adjust each weight and if we know that the function is convex (the first and second order conditions are met), we know we can subtract the gradient from the weights because the gradient represents the 'positive gradient'; hence we want to travel in the opposite direction to march toward the minimum.

b) The problem domain space represents how closely a model matches the data; whereas the parameter space represents how closely the model matches the optimal model. In Gradient descent, we are adjusting the parameters to get as close to the optimal model. So we can think of the parameter space as the weight space and we can think of the problem domain space adjusting the model to fit the data. With the parameter space, we are adjusting the weights through gradient descent to try to reach the the optimal target. We cannot find the optimal model by looking at the error surface in the parameter space because that space is unknown; thus we can only approximate a target; a target that is based on (and limited by) the data that we have trained the model on.

c) The learning rate alters the degree to which the weights are adjusted after calculating the gradient based on the loss function. If the learning rate is too big, we risk oscillating around the mininum and never converging on the minumum; if the learning rate is too small; we risk running too many iterations without converging on the minimum because the steps taken are too small.

d) OLS is defined as

$$y_j = w_0 + w_1 x_1 ... + w_k x_k \ = \sum_{j=0}^{k} w^T x$$

In vector notation, this can be written:

$$y_j = \mathbf{w}^T \mathbf{x}_j + b$$

Which in the closed form solution the weights can be computed as follows:

$$w = (X^T X)^{-1} X^T y$$

This calculation requires the inverse of the transposed matrix, which is computationally very expensive, especially as the number of predictors and number of observations increase. On the other hand, with Gradient Descent, we can parallelize much of the process (all of the loss calculations on each observations) and avoid the expensive calculation (inverse of the transposed matrix) altogether.

About the Data

For the main task in this portion of the homework you will use data about red and white Portuguese wines. This data (http://archive.ics.uci.edu/ml/datasets/Wine+Quality) was made available to the UC Irvine public repository of Machine Learning datasets by researchers at the University of Minho in association with this paper (https://www.sciencedirect.com/science/article/pii/S0167923609001377?via%3Dihub):

> P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis. Modeling wine preferences by data mining from physicochemical properties. In Decision Support Systems, Elsevier, 47(4):547-553, 2009.

The dataset includes 12 fields:

```
fixed acidity
volatile acidity
citric acid
residual sugar
chlorides
free sulfur dioxide
total sulfur dioxide
density
pH
sulphates
alcohol
quality
```
-- (*a score between 0 and 10*)

`IMPORTANT NOTE:` The outcome variable in our data is a human assigned score ranging from 0 to 10. Since the scores are integers this is actually an ordinal and not numerical outcome varaible. However for the purposes of this assignment we'll treat it as a numerical quantity.

The data are in two files: one containing red wines and another containing white wines. Use the following cells to download the data, add a field for red/white, and split it into a test and train set.

```
headers = dbutils.fs.head(DATA_PATH + '/winequality-red.csv',200)
headers = headers.split('\n')[0]
FIELDS = ['color'] + re.sub('"', '',headers).split(';')
```

```
[Truncated to first 200 bytes]
```

```
# RUN THIS CELL AS IS. You should see winequality-red.csv and winequality-
white.csv. Please contact an Instructor or TA if you do not see these two
files.
dbutils.fs.ls(DATA_PATH)
```

```
Out[588]: [FileInfo(path='dbfs:/mnt/mids-w261/data/HW4/titanic/', name='titani
c/', size=0),
 FileInfo(path='dbfs:/mnt/mids-w261/data/HW4/winequality-red.csv', name='winequ
ality-red.csv', size=84199),
 FileInfo(path='dbfs:/mnt/mids-w261/data/HW4/winequality-white.csv', name='wine
quality-white.csv', size=264426)]
```

```
# load the raw data into an RDD - RUN THIS CELL AS IS
redsRDD = sc.textFile(DATA_PATH + '/winequality-red.csv')\
            .filter(lambda x: x != headers)\
            .map(lambda x: '1;' + x) # set first field 1 to indicate red wine
whitesRDD = sc.textFile(DATA_PATH + '/winequality-white.csv')\
              .filter(lambda x: x != headers)\
              .map(lambda x: '0;' + x) # set first field 0 to indicate white
wine
```

```
redsRDD.take(5)
```

```
Out[590]: ['1;7.4;0.7;0;1.9;0.076;11;34;0.9978;3.51;0.56;9.4;5',
 '1;7.8;0.88;0;2.6;0.098;25;67;0.9968;3.2;0.68;9.8;5',
 '1;7.8;0.76;0.04;2.3;0.092;15;54;0.997;3.26;0.65;9.8;5',
 '1;11.2;0.28;0.56;1.9;0.075;17;60;0.998;3.16;0.58;9.8;6',
 '1;7.4;0.7;0;1.9;0.076;11;34;0.9978;3.51;0.56;9.4;5']
```

```
whitesRDD.take(5)
```

```
Out[591]: ['0;7;0.27;0.36;20.7;0.045;45;170;1.001;3;0.45;8.8;6',
 '0;6.3;0.3;0.34;1.6;0.049;14;132;0.994;3.3;0.49;9.5;6',
 '0;8.1;0.28;0.4;6.9;0.05;30;97;0.9951;3.26;0.44;10.1;6',
 '0;7.2;0.23;0.32;8.5;0.058;47;186;0.9956;3.19;0.4;9.9;6',
 '0;7.2;0.23;0.32;8.5;0.058;47;186;0.9956;3.19;0.4;9.9;6']
```

```
# Generate 80/20 (pseudo)random train/test split - RUN THIS CELL AS IS
trainRDD, heldOutRDD = redsRDD.union(whitesRDD).randomSplit([0.8,0.2], seed =
1)
print(f"... held out {heldOutRDD.count()} records for evaluation and assigned
{trainRDD.count()} for training.")
```

```
... held out 1316 records for evaluation and assigned 5181 for training.
```

```
# helper function - RUN THIS CELL AS IS
def parse(line):
    """
    Map record_csv_string --> (tuple,of,fields)
    """
    fields = np.array(line.split(';'), dtype = 'float')
    features,quality = fields[:-1], fields[-1]
    return(features, quality)
```

```
# cache the training set - RUN THIS CELL AS IS
trainRDDCached = trainRDD.map(parse).cache()
```

Question 2: EDA

A statistician's approach to Linear Regression typically involves a series of EDA steps to examine each feature in the data and then a series of steps to test assumptions about their potential contribution to a multi-feature linear model. In particular, we'd want to look for a set of features that exhibit a likely linear relationship with the outcome variable and that are *not* highy correlated with each other. In the context of machine learning, these considerations remain important techniques for improving model generalizability despite the common practice to use model evaluation techniques (and large data sets) to get the final word on feature selection.

In this question we'll briefly look at the features in our data set. To mimic an 'at scale' analysis we'll start by sampling from our Spark RDD training set so that we have a manageable amount of data to work with in our visuals.

Q2 Tasks:

- **a) short response:** Run the provided code to sample 1000 points and visualize histograms of each feature. Comment on the distributions you observe (eg. *Which features appear normaly distributed, which don't? Which features vary most/least?*) How is the varaible `color` different than the other features & what does that mean about how we interpret its regression coefficient?

- **b) short response:** Run the provided code to create boxplots of each feature. Which, if any, appear to have a positive linear relationship with `quality` ? Which if any appear to have a negative linear relationship with `quality` ?
- **c) short response:** Run the provided code to plot the correlations matrix. Which pairs of features are most *strongly* (postively or negatively) associated with each other? What implications would that have for our feature selection?

Q2 Student Answers:

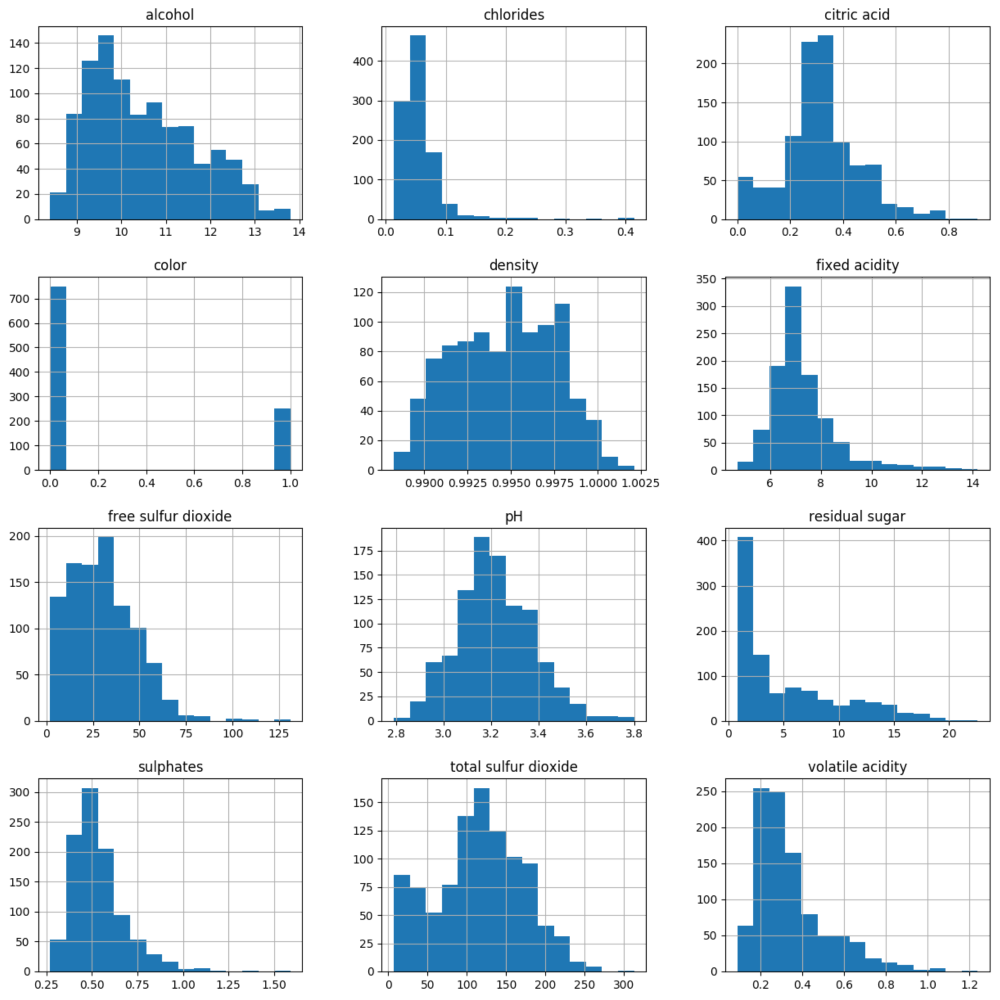
a) pH appears to follow a normal distribution. Most of the predictor variables appear to be positively skewed and total sulfur dioxide appears to be a bimodal distribution. Color must be a dummy indicator, since there are only 2 values with any frequency in the distribution. As such, we should transform the color variable into a dummy variable (0 for white, 1 for red), which would make interpretation of its coefficient simpler -- a positive coefficient would indicate higher quality is associated with the color red; a negative coefficient would indicate that higher quality is associated with the color white. Some of the variables have tight distributions (low variance), including chlorides, sulfates. In contrast, some variables vary more, like alcohol, and pH.

b) Fixed acidity and volatile acidity appear to be negatively correlated to quality, but there are many outliers, so we would need to perform more EDA to verify this. Citric acid appears to have a slightly positive correlation to quality; but we don't have enough data points to state this confidence.

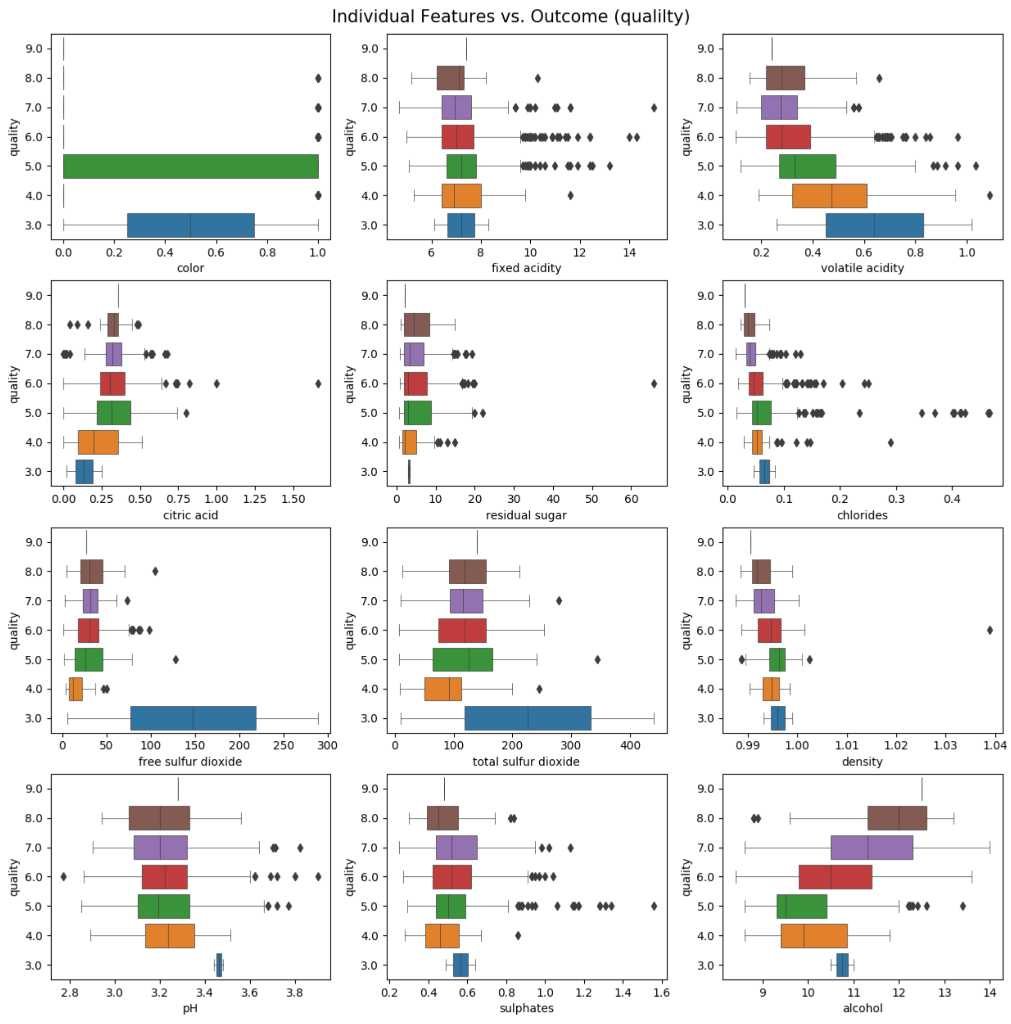
c) Positive correlations between predictor variable include density and alchol, as well as color and total sulfur dioxide. Negative correlations include: free sulfur dioxide and total sulfur dioxide, color and volatile acidity. In multiple linear regression, multicollinearity will throw off the coefficients for the correlated variables. This increases the variance of the coefficients, making them unstable. The more variance, the higher the standard error, the lower the statistical signficance for these coefficients. So if we can drop variables during feature selection such that none of the remaining are highly correlated (negatively or positively), we will have a better model.

```
# part a - take a 1000 point sample for EDA (RUN THIS CELL AS IS)
sample = np.array(trainRDDCached.map(lambda x: np.append(x[0], [x[1]]))
                                .takeSample(False, 1000))
sample_df = pd.DataFrame(np.array(sample), columns = FIELDS)
```

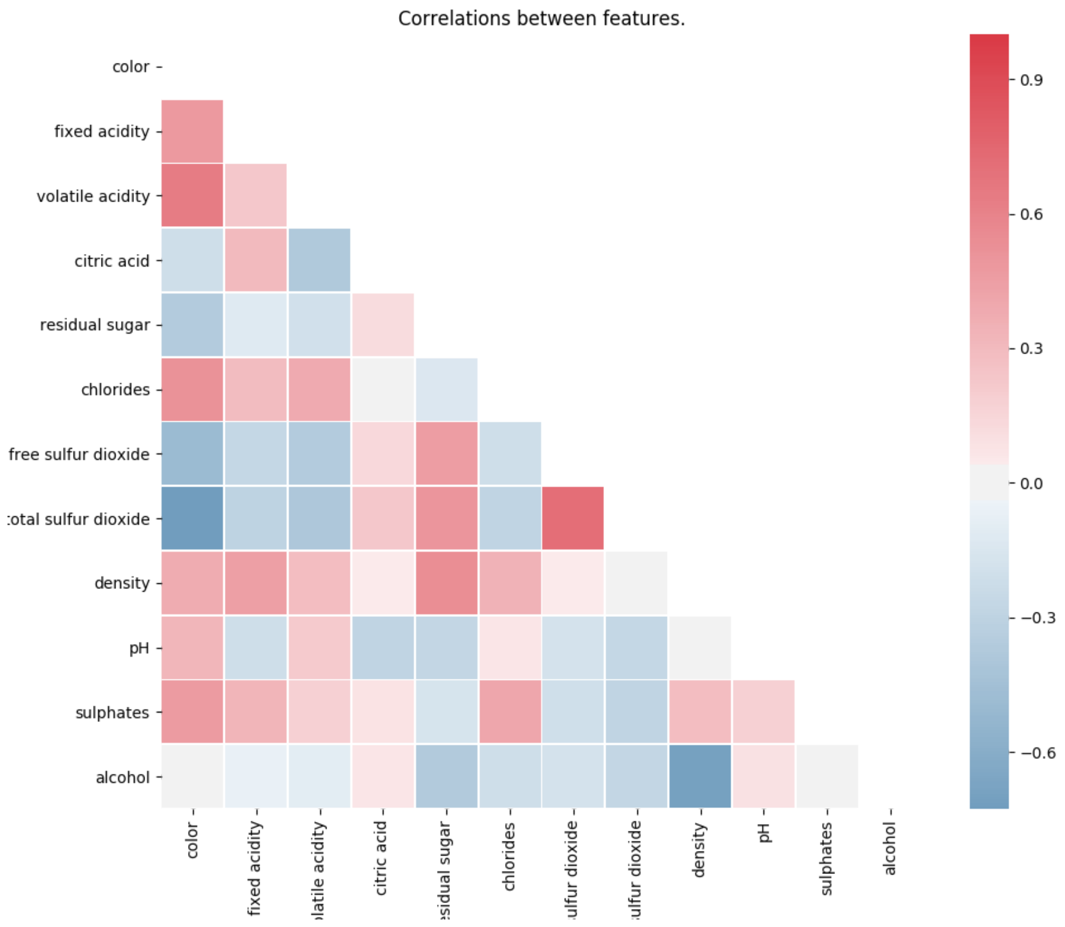
```
# part a - take a look at histograms for each feature (RUN THIS CELL AS IS)
sample_df[FIELDS[:-1]].hist(figsize=(15,15), bins=15)
display(plt.show())
```



```
# part b -  plot boxplots of each feature vs. the outcome (RUN THIS CELL AS IS)
fig, ax_grid = plt.subplots(4, 3, figsize=(15,15))
y = sample_df['quality']
for idx, feature in enumerate(FIELDS[:-1]):
    x = sample_df[feature]
    sns.boxplot(x, y, ax=ax_grid[idx//3][idx%3], orient='h', linewidth=.5)
    ax_grid[idx//3][idx%3].invert_yaxis()
fig.suptitle("Individual Features vs. Outcome (qualilty)", fontsize=15, y=0.9)
display(plt.show())
```



```
# plot heatmap for correlations matrix - RUN THIS CELL AS IS
corr = sample_df[FIELDS[:-1]].corr()
fig, ax = plt.subplots(figsize=(11, 9))
mask = np.zeros_like(corr, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True
cmap = sns.diverging_palette(240, 10, as_cmap=True)
sns.heatmap(corr, mask=mask, cmap=cmap, center=0, linewidths=.5)
plt.title("Correlations between features.")
display(plt.show())
```



Question 3: OLS Loss

For a parametric model, the key factor that will impact how easy it is to optimize is your choice of how to define the loss function. In Ordinary Least Squares (OLS) Regression our loss function is just about as convenient as you will get: not only is it convex, its also very easy to interpret.

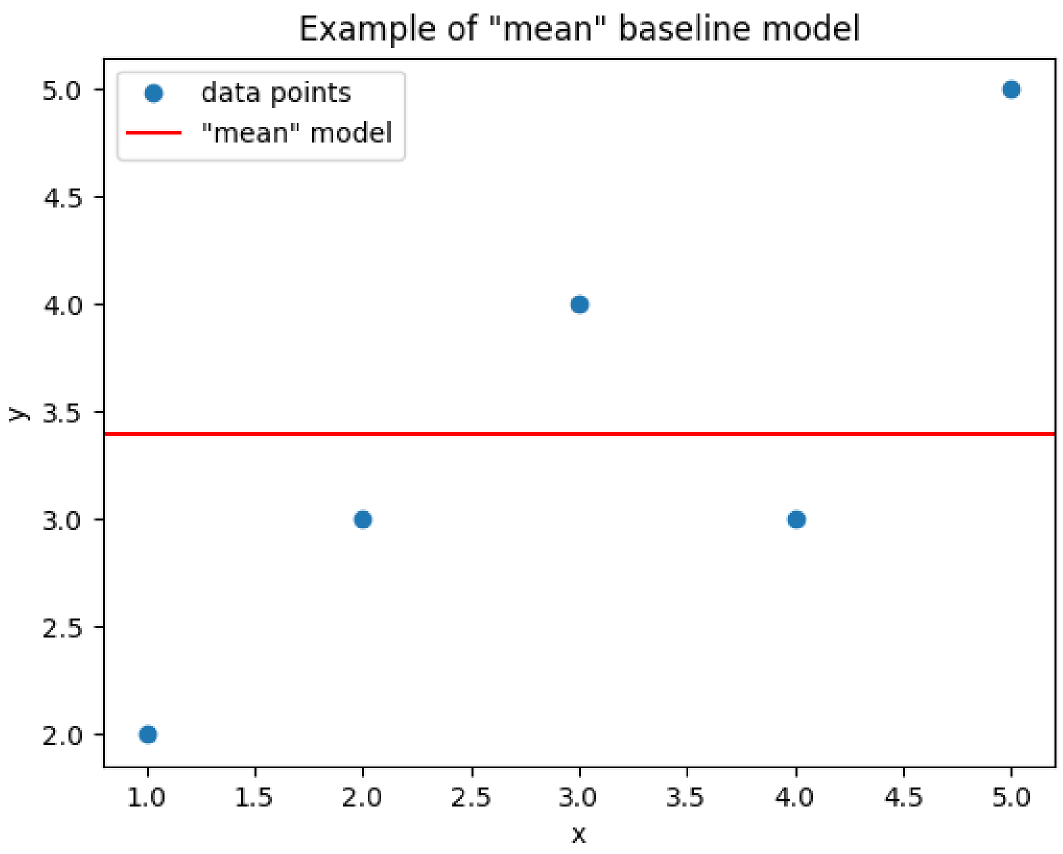
When doing supervised learning, a simple sanity check consists of comparing one's estimator against simple rules of thumb. It is useful as a simple baseline to compare with other (real) regressors. Examples of regression baselines include:

- "mean": always predicts the mean of the training set
- "median": always predicts the median of the training set
- "quantile": always predicts a specified quantile of the training set,provided with the quantile parameter.
- "constant": always predicts a constant value that is provided by the user.

In this question you'll "choose" a baseline model and then write a function to compute the loss of a linear model in Spark. You'll reuse this function in Q4 when you implement gradient descent.

Baseline example illustrated:

```
# points from our mini example from the demo 6 notebook
points = np.array([[1,2],[3,4],[5,5],[4,3],[2,3]])
x = points[:,0]
y = points[:,1]

plt.figure()
plt.plot(x, y,'o', label='data points')
plt.axhline(np.mean(y),c='r', label='"mean" model')
plt.title('Example of "mean" baseline model')
plt.ylabel("y")
plt.xlabel("x")
plt.legend()
display(plt.show())
```



Q3 Tasks:

- **a) code:** Fill in the code below to compute the mean and variance of your outcome variable. [**`HINT:`** *use `trainRDDCached` as the input & feel free to use Spark built-in functions.*]
- **b) short response:** Write the formula for the OLS loss function and explain how to interpret it graphically.
- **c) short response:** In the context of linear models & vector computations what does it mean to 'augment' a data point and why do we do this?
- **d) code + short response:** Fill in the missing code to complete the `OLSLoss` function. Is computing the loss "embarassingly parallel'? Explain. [**`TIP:`** Use `augmentedData` as your input when computing the loss.]

- **e) code + short response:** Fill in the missing code to define a baseline model for this data set that has a bias term equal to the mean of your outcome variable and `0.0` for all coefficients. Note that in the docstring for `OLSLoss` we specified that the model should be a numpy array with the bias in the first position. Once you've defined your model, run the provided cells to check that your model has the correct dimensions and then compute the loss for your baseline model. Compare your results to the result you got in `part a` and explain what you see.

Q3 Student Answers:

b)

$$f(\mathbf{w}, b) = \frac{1}{n}\sum_{j=1}^{n}\left[(\mathbf{w}^T\mathbf{x}_j + b) - y_i\right]^2$$

This function can also be expressed such that the bias is included in the weights:

$$\mathbf{x}^{'} := \begin{bmatrix}\mathbf{x}\\1\end{bmatrix}, \quad \boldsymbol{\theta} := \begin{bmatrix}\mathbf{w}\\b\end{bmatrix}$$

$$f(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\left[\boldsymbol{\theta}^T \cdot \mathbf{x}_i^{'} - y_i\right]^2$$

This loss function is the Mean Squared Loss. To optimize this objective (loss) function, we want to minimize it, so we will want to make the residuals (the difference between the predicted and the actual) as small as possible. In a simple 2D graphical form, where we have x and y, we are trying to find the slope so that the distance between the actual data points and the line (prediction) are as small as possible. The closer this line (our prediction) matches the data points, the smaller the loss. In mutliple dimensions, this becomes a hyperplane in which we are calculating the residuals.

c) We augment the data points by adding the integer 1 to our vector of data points. We do this so that we don't have to treat the bias separately when calculating the loss (or gradient). Instead, the the '1' at the end (or beginning) of the vector of data points will go through the matrix calculation and since the augmented datapoint will have an integer 1, this essentially adds the bias (unweighted).

d) Yes. The loss calculation for each observation can be run in parallel, and at the end, a reduceBy step can compute the average loss by dividing the number of observations (count) by the sum of the residuals (e.g. loss) squared.

e) Our loss calculation equals the variance of y. We see this because we are setting the bias to the mean and the weights to 0, so we start with a line across the mean of y. Each residual is the difference between the data point and the predicted y (e.g.the point on the line). So in this case, our loss function is the definition of variance because the 'predicted y' is set at the mean (having a zero slope).

```
# part a - mean and variance of the outcome variable
meanQuality =  trainRDDCached.map(lambda x: x[1]).mean()
varQuality = trainRDDCached.map(lambda x: x[1]).variance()
print(f"Mean: {meanQuality}")
print(f"Variance: {varQuality}")
```

```
Mean: 5.811040339702759
Variance: 0.760086564865664
```

```
# part d - write function to compute loss (FILL IN MISSING CODE BELOW)
def OLSLoss(dataRDD, W):
    """
    Compute mean squared error.
    Args:
        dataRDD - each record is a tuple of (features_array, y)
        W       - (array) model coefficients with bias at index 0
    """
    augmentedData = dataRDD.map(lambda x: (np.append([1.0], x[0]), x[1]))
    ################## YOUR CODE HERE ##################
    loss_agg = augmentedData.map(lambda x: ((W.dot(x[0]) - x[1])**2, 1)) \
                            .reduce(lambda x,y: (x[0]+y[0], x[1]+y[1]))
    loss = loss_agg[0] / float(loss_agg[1])
    ################## (END) YOUR CODE ##################
    return loss
```

```
# part e - define your baseline model here
BASELINE = np.zeros(13)
BASELINE[0] = 5.811040339702759
```

```
# part e - compute the loss for your baseline model (RUN THIS CELL AS IS)
assert len(BASELINE) == len(trainRDDCached.take(1)[0][0]) + 1, "Double check
model dimensions"
print(f"Baseline model loss: {OLSLoss(trainRDDCached, BASELINE)}")
```

```
Baseline model loss: 0.7600865648656668
```

Question 4: Vanilla Gradient Descent

Performing Gradient Descent technically only requires two steps: 1) *use the current model to calculate the gradient*; 2) *use the gradient to update the current model parameters*. In practice though, we'll want to add a third step which is to compute the loss for our new model so that we can see if its working. In this question you'll implement gradient descent for OLS regression and take a look at a few update steps.

Q4 Tasks:

- **a) short response:** Jimi describes the main part of the gradient calculation for OLS Regression using a short mantra: '*the mean of the data weighted by the errors*'. . Write the formula for the gradient and explain how it reflects this phrase.
- **b) short response:** Looking at the formula you wrote in `part a`, what parts of this calculation can be parallelized and what has to happen after reducing?
- **c) code:** Fill in the missing lines in `GDUpdate` to compute the gradient and perform a single update of the model parameters.
  - **`TIP 1:`** *remember that the gradient is a vector of partial derivatives, `grad` should be a numpy array*
  - **`TIP 2:`** *Spark's built in `mean()` function may help you here*
- **d) short response:** Run the provided code to perform 5 steps of Gradient Descent on our data. What is wrong with these results?
- **e) code + short response:** Fill in the missing code in `normalize` so that this function scales each feature and centers it at 0. Then use the provide code block to rerun your same gradient descent code on the scaled data. Use these results to explain what the problem was in 'd'.
  - **`TIP:`** *You may find this brief illustration (https://www.coursera.org/lecture/machine-learning/gradient-descent-in-practice-i-feature-scaling-xx3Da) from Andrew Ng's Coursera helpful.*

Q4 Student Answers:

a) The OLS gradient calculation is the following:

$$\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}) = \frac{2}{n} \sum_{i=1}^{n} \left[\boldsymbol{\theta}^T \cdot \mathbf{x}_i' - y_i\right] \cdot \mathbf{x}_i'$$

Which is the rate of change (the derivative) of the OLS loss function. In this loss function, we are calculating how far off the predictions are from the data. If the weights are zero, this "average" squared error represents how different each data observation is from its mean. With Gradient Descent, we start adjusting these weights to minimize the loss, essentially getting us as close as possible to the mean for each variable in an optimized fashion such that in total, we find the mimimum total loss (or error). Looking at the key part of the gradient function, we see that it is in fact the mean of the errors (times 2). And we are adjusting this by the second part of the equation, the weights.

b) We can parallelize the gradient calculation (in a map step) for each observation. Then in the reduce step (not paralellized), we need to sum these gradients, multiply by 2 and divide by the number of observations. This will give us the deltas of each coefficient, which we will mutiply by the learning rate to adjust the weight vector for our next iteration.

c) *complete the coding portions of this question before answering d & e*

d) We did not scale the features in the data, so the gradient steps didn't result in much improvement in the model with each iteration.

e) When the data is normalized, we are scaling all of the features between -1 and 1. This will allow the contours of the loss function to be closer to circular, which will result in a more direct path toward convergence in Gradient Descent. The problem we observed with unscaled features is that some of the variables have a much larger range of values which result if loss contours that are skewed with much greater distance in the dimensions with the variables of large ranges. With these skewed contours, we risk oscillating or taking too many steps to make

> improvements and we will never converge on a set of adjust weights that are close to an optimal solution. In addition, we want to scale the data such that the the regularization term λ has a similar effect across all of the weights, regardless of their range of values.

```
# part b - function to perform a single GD step
def GDUpdate(dataRDD, W, learningRate = 0.1):
    """
    Perform one OLS gradient descent step/update.
    Args:
        dataRDD - records are tuples of (features_array, y)
        W       - (array) model coefficients with bias at index 0
    Returns:
        new_model - (array) updated coefficients, bias at index 0
    """
    # add a bias 'feature' of 1 at index 0
    augmentedData = dataRDD.map(lambda x: (np.append([1.0], x[0]),
x[1])).cache()

    ################## YOUR CODE HERE #################
    grad_agg = augmentedData.map(lambda x: ((W.dot(x[0]) - x[1]) * x[0],  1)) \
                              .reduce(lambda x,y: (x[0]+y[0],  x[1]+y[1]))
    grad = 2 * grad_agg[0] / grad_agg[1]

    update = np.multiply(grad,learningRate)

    new_model = W - update
    ################## (END) YOUR CODE #################

    return new_model
```

```
%%time
# part c - take a look at a few Gradient Descent steps (RUN THIS CELL AS IS)

nSteps = 5
model = BASELINE
print(f"BASELINE:  Loss = {OLSLoss(trainRDDCached,model)}")
for idx in range(nSteps):
    print("----------")
    print(f"STEP: {idx+1}")
    model = GDUpdate(trainRDDCached, model)
    loss = OLSLoss(trainRDDCached, model)
    print(f"Loss: {loss}")
    print(f"Model: {[round(w,3) for w in model]}")
```

```
BASELINE:  Loss = 0.7600865648656668
----------
STEP: 1
Loss: 3255.0397706648027
Model: [5.811, -0.009, -0.017, -0.007, 0.002, -0.035, -0.001, 0.167, -0.489, -
0.0, 0.0, 0.001, 0.094]
----------
STEP: 2
Loss: 41538896320.900345
Model: [15.998, 0.946, 71.311, 3.109, 3.394, 67.104, 0.517, 372.213, 1469.681,
10.133, 32.587, 5.197, 105.407]
----------
STEP: 3
Loss: 5.3215162542469235e+17
Model: [-36732.224, -3790.819, -257459.013, -11294.195, -12193.785, -240557.88
6, -1880.28, -1359608.579, -5252902.327, -36544.479, -117571.505, -18781.008, -
379946.295]
----------
STEP: 4
Loss: 6.817355374095379e+24
Model: [131498780.95, 13574790.866, 921540888.277, 40426173.604, 43645471.549,
861027822.897, 6730358.366, 4867148463.848, 18801163135.814, 130805962.538, 420
832007.136, 67225003.981, 1359973545.811]
----------
STEP: 5
Loss: 8.733663880159573e+31
Model: [-470665198626.59, -48587550191.149, -3298412689019.922, -144694834630.2
6, -156217444459.899, -3081821742976.577, -24089547803.278, -17420693064591.16
8, -67293783870862.45, -468185458548.501, -1506257249668.984, -240614204453.62
4, -4867666956412.445]
CPU times: user 92 ms, sys: 0 ns, total: 92 ms
Wall time: 1.16 s
```

```
# part d - helper function to normalize the data (FILL IN THE MISSING CODE
BELOW)
def normalize(dataRDD):
    """
    Scale and center data round mean of each feature.
    Args:
        dataRDD - records are tuples of (features_array, y)
    Returns:
        normedRDD - records are tuples of (features_array, y)
    """
    featureMeans = dataRDD.map(lambda x: x[0]).mean()
    featureStdev = np.sqrt(dataRDD.map(lambda x: x[0]).variance())

    ################ YOUR CODE HERE #############
    normedRDD = dataRDD.map(lambda x: ((x[0] - featureMeans) / featureStdev,
x[1]))
    ################ FILL IN YOUR CODE HERE #############

    return normedRDD
```

```
# part d - cache normalized data (RUN THIS CELL AS IS)
normedRDD = normalize(trainRDDCached).cache()
```

```
%%time
# part e - take a look at a few GD steps w/ normalized data  (RUN THIS CELL AS
IS)
nSteps = 5
model = BASELINE
print(f"BASELINE:  Loss = {OLSLoss(trainRDDCached,model)}")
for idx in range(nSteps):
    print("----------")
    print(f"STEP: {idx+1}")
    model = GDUpdate(normedRDD, model)
    loss = OLSLoss(normedRDD, model)
    print(f"Loss: {loss}")
    print(f"Model: {[round(w,3) for w in model]}")
```

```
BASELINE:  Loss = 0.7600865648656668
----------
STEP: 1
Loss: 0.6555276691703882
Model: [5.811, -0.02, -0.014, -0.045, 0.014, -0.007, -0.035, 0.009, -0.009, -0.
054, 0.003, 0.008, 0.078]
----------
STEP: 2
```

```
Loss: 0.6150592390442609
Model: [5.811, -0.022, -0.014, -0.071, 0.023, -0.006, -0.05, 0.014, -0.021, -0.
079, 0.007, 0.024, 0.131]
----------
STEP: 3
Loss: 0.5935522582531457
Model: [5.811, -0.018, -0.01, -0.09, 0.03, 0.002, -0.057, 0.02, -0.032, -0.09,
0.011, 0.04, 0.167]
----------
STEP: 4
Loss: 0.5798853360580276
Model: [5.811, -0.014, -0.006, -0.105, 0.034, 0.012, -0.061, 0.026, -0.04, -0.0
94, 0.013, 0.054, 0.195]
----------
STEP: 5
Loss: 0.5703196233260648
Model: [5.811, -0.009, -0.003, -0.117, 0.036, 0.024, -0.063, 0.032, -0.047, -0.
096, 0.016, 0.066, 0.217]
CPU times: user 96 ms, sys: 0 ns, total: 96 ms
Wall time: 1.14 s
```

Question 5: Assessing the performance of your model.

Printing out the loss as we perform each gradient descent step allows us to confirm that our Gradient Descent code appears to be working, but this number doesn't accurately reflect "how good" our model is. In this question you'll plot error curves for a test and training set in order to discuss model performance. Note that although we split out a test & train set when we first loaded the data... in the spirit of keeping that 20% truly 'held out' until then end of the assignment, we'll make an additional split for the purposes of this question dividing the existing training set into two smaller RDDs.

Q5 Tasks:

- **a) short response:** Why doesn't the loss that we printed in Question 4 accurately reflect "how good" our model is?
- **b) code:** Since we're going to be running Gradient Descent a number of times let's package it into a function for convenience. Fill in the missing code in

`GradientDescent()` , note that the missing code is going to look a lot like the provided code blocks in Q5 -- feel free to use those as a starting point.

- **c) short response:** Use the provided code to split the normalized data into a test and train set, then run 50 iterations of gradient descent and plot the MSE curves for each. Describe what you see and speculate about why this might be happening.
- **d) short response:** Note that passing the optional parameter `seed` to the Spark method `randomSplit` allows us to pseudo randomize our test/train split in a way that is replicable. Re-run the code for part 'c but this time in the line where we perform the `normedRDD.randomSplit` change the seed to *`seed = 5`* . What changes in the plot? Repeat for *`seed = 4`* . How does this change your interpret the results you saw in 'c'. What is the more likely explanation?

Q5 Student Answers:

a) We only ran 5 iterations, so we have probably haven't performed enough iterations to converge. But more importantly, we haven't tried running our model on unseen data, so we don't know if the model sufficiently captures the underlying patterns to do well at predicting the quality of wine on new data. This is where cross-validation would help. Not only would it give us a slightly different set of training data each fold to fully capture the model, but we can simulate 'unseen' data with each fold. This cross-validation step would also allow us to determine an appropriate learning rate by trying diffent values to see which offers the best overall performance.

c) The test error is higher than the train error across all of the iterations. The most improvement in the model occurs within the first 20 iterations; after than we see little improvement and we don't see the train and test error curves get any closer. This might be because our training data doesn't adequately capture the problem domain space, so there will always be a significant gap between these error plots.

d) When we run with a seed = 4, we see the train and test error lines swap, with the plot showing the test having a lower error rate than the training data. Then when we runthe split with seed of 4, we see very close errors between training and test across all iterations. This hints that the way we split the data (which observations are trained vs which are held out for test) has larger impact on how how closely the test errors will track with the train errors. So this randomness in the split will result in a different model that deviates slightly. That is why it is important to perform sufficient cross-fold validations to understand the "average" predicted (test) error. In other words, we can't read too much into the performance of the model based on any given split of training and test data.

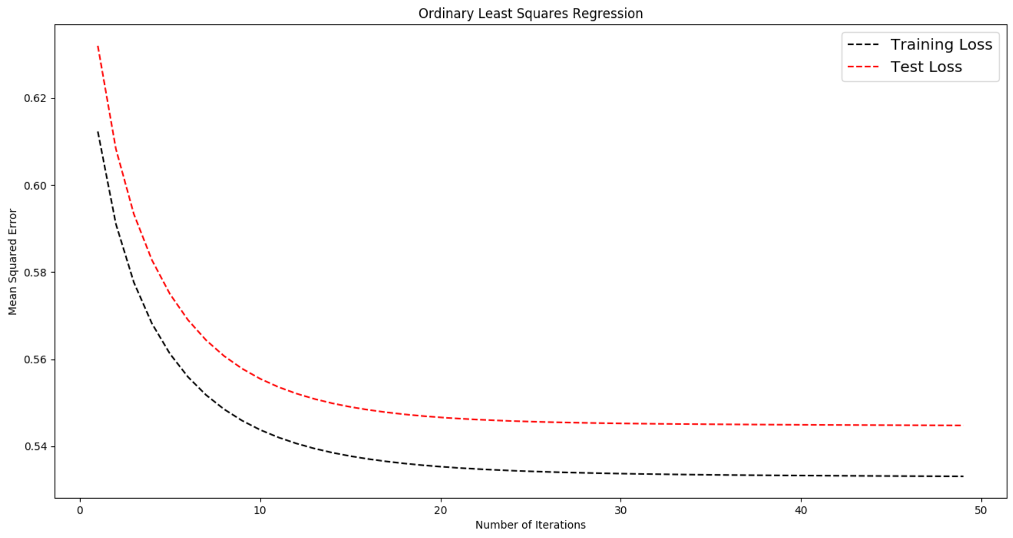
```
# part b - OLS gradient descent function
def GradientDescent(trainRDD, testRDD, wInit, nSteps = 20,
                    learningRate = 0.1, verbose = False):
    """
    Perform nSteps iterations of OLS gradient descent and
    track loss on a test and train set. Return lists of
    test/train loss and the models themselves.
    """
    # initialize lists to track model performance
    train_history, test_history, model_history = [], [], []

    # perform n updates & compute test and train loss after each
    model = wInit
    for idx in range(nSteps):

        ############## YOUR CODE HERE #############
        model = GDUpdate(trainRDD, model)
        training_loss = OLSLoss(trainRDD, model)
        test_loss = OLSLoss(testRDD, model)
        ############## (END) YOUR CODE #############

        # keep track of test/train loss for plotting
        train_history.append(training_loss)
        test_history.append(test_loss)
        model_history.append(model)

        # console output if desired
        if verbose:
            print("----------")
            print(f"STEP: {idx+1}")
            print(f"training loss: {training_loss}")
            print(f"test loss: {test_loss}")
            print(f"Model: {[round(w,3) for w in model]}")
    return train_history, test_history, model_history
```

```
# plot error curves - RUN THIS CELL AS IS
def plotErrorCurves(trainLoss, testLoss, title = None):
    """
    Helper function for plotting.
    Args: trainLoss (list of MSE) , testLoss (list of MSE)
    """
    fig, ax = plt.subplots(1,1,figsize = (16,8))
    x = list(range(len(trainLoss)))[1:]
    ax.plot(x, trainLoss[1:], 'k--', label='Training Loss')
    ax.plot(x, testLoss[1:], 'r--', label='Test Loss')
    ax.legend(loc='upper right', fontsize='x-large')
    plt.xlabel('Number of Iterations')
    plt.ylabel('Mean Squared Error')
    if title:
        plt.title(title)
    display(plt.show())
```

```
# run 50 iterations (RUN THIS CELL AS IS)
wInit = BASELINE
trainRDD, testRDD = normedRDD.randomSplit([0.8,0.2], seed = 2018)
start = time.time()
MSEtrain, MSEtest, models = GradientDescent(trainRDD, testRDD, wInit, nSteps =
50)
print(f"\n... trained {len(models)} iterations in {time.time() - start}
seconds")
```

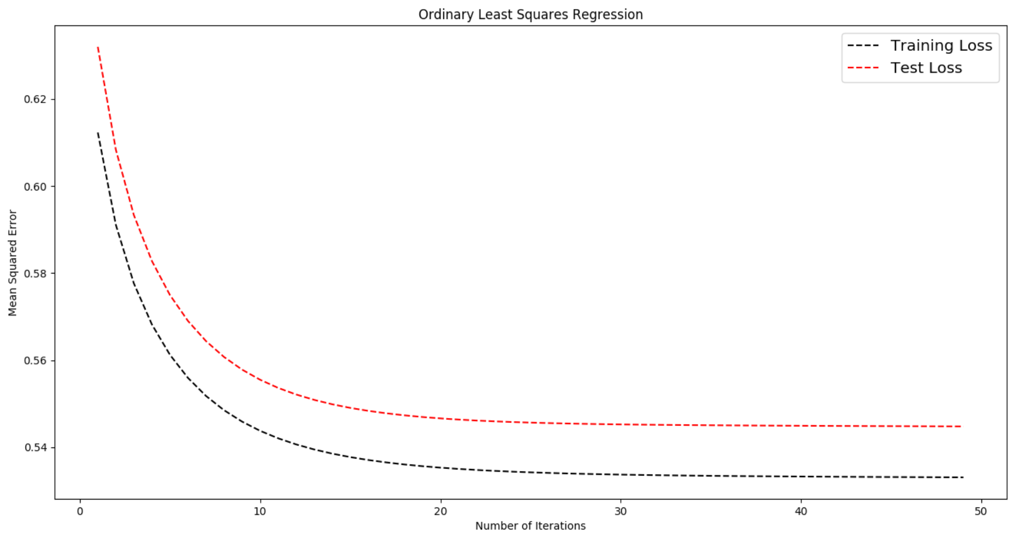
```
... trained 50 iterations in 16.36422085762024 seconds
```

```
# take a look (RUN THIS CELL AS IS)
plotErrorCurves(MSEtrain, MSEtest, title = 'Ordinary Least Squares Regression'
)
```



```
# save the models & their performance for comparison later (RUN THIS CELL AS
IS)
np.savetxt(hw4_path_open + 'OLSmodels.csv', np.array(models), delimiter=',')
np.savetxt(hw4_path_open + 'OLSloss.csv', np.array([MSEtrain, MSEtest]),
delimiter=',')
```

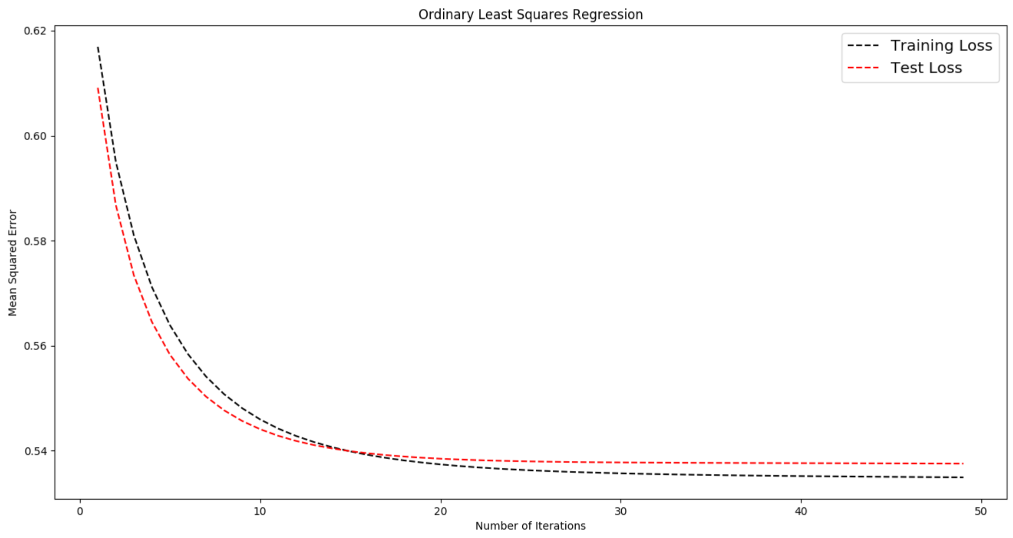
```
# take a look (RUN THIS CELL AS IS)
dbutils.fs.put(hw4_path + 'ols_models.txt', str(models), True)
dbutils.fs.put(hw4_path + 'ols_loss.txt', str([MSEtrain, MSEtest]), True)
plotErrorCurves(MSEtrain, MSEtest, title = 'Ordinary Least Squares Regression'
)
```



```
# run 50 iterations (RUN THIS CELL AS IS)
wInit = BASELINE
trainRDD, testRDD = normedRDD.randomSplit([0.8,0.2], seed = 5)
start = time.time()
MSEtrain, MSEtest, models = GradientDescent(trainRDD, testRDD, wInit, nSteps =
50)
print(f"\n... trained {len(models)} iterations in {time.time() - start}
seconds")
```

```
... trained 50 iterations in 13.241607189178467 seconds
```

```
# take a look (RUN THIS CELL AS IS)
plotErrorCurves(MSEtrain, MSEtest, title = 'Ordinary Least Squares Regression'
)
```



Question 6: Cross Validation

In question 5 we mentioned that computing the loss after each iteration is not strictly a part of Gradient Descent, its just convenient for visualizing our progress. This "third step" however comes with a tradeoff: it requires an extra pass through the data. Normally this would cause us to cringe except for the fact that both the loss computation and the gradient computation are very easy to parallelize - lots of the work can be done in place no shuffle needed for the aggregation.

Cross-validation (https://en.wikipedia.org/wiki/Cross-validation_(statistics)), sometimes called rotation estimation, or out-of-sample testing, is a model validation technique for assessing how well the model will generalize to an independent data set. The goal of cross-validation is to test the model's ability to predict new data.

Cross validation, which will solve the problem of the unreliable test-loss that we saw in question 5, presents a bit more of a scalability challenge. To avoid over-dependence on a particulary good or bad test/train split we divide the data into `k` roughly equal size parts and train `k` models. The `k-th` model is trained on all the data *except* the `k-th` split which is used as a test set for that model. Finally we compute the loss by averaging together the test/train loss for each model. In this question we've provided a code base to perform gradient descent and cross validation in parallel. You'll fill in some of the key details based on your understanding from questions 1-5.

From ISLR Chapter 5.1 - Cross Validation

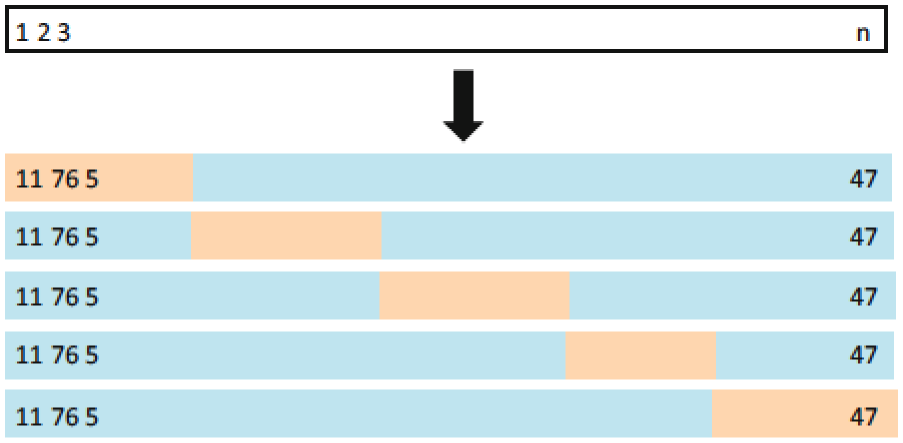


***FIGURE 5.5.** A schematic display of 5-fold CV. A set of n observations is randomly split into five non-overlapping groups. Each of these fifths acts as a validation set (shown in beige), and the remainder as a training set (shown in blue). The test error is estimated by averaging the five resulting MSE estimates.*

Q6 Tasks:

- **a) short response:** A naive approach to training an OLS Regression model with cross validation might be to simply perform Gradient Descent on each of the 5 models in sequence. In this naive approach, how many total passes would be made over the data? [**`HINT:`** *it will depend on factors that you should be able to name.*]
- **b) short response:** Read through the provided helper function `kResiduals()` and note where it gets used in the subsequent function `CVLoss()` . For each record in the original dataset, how many tuples does `kResiduals()` emit? What are the keys of these newly emitted records? How will these keys help us compute cross validated loss?
- **c) code:** Complete the missing Spark code in `CVLoss()` so that this function returns the test/train cross validated error for a given set of data splits and their corresponding models. [**`TIP:`** *your goal is to start from `partialLossRDD` and*

compute the test & train loss for each model so that the provided code can take the final average].

- **d) code:** Read through the provided functions `partialGradients()` and `CVUpdate()` . These should have a familiar feel. Fill in the missing line in `CVUpdate()` to update each model and add the (new) array of coefficients to the `new_models` list.
- **e) short response:** Read `GradientDescent_withCV()` and then run the provided code to perform 50 iterations and plot the error curves. What can you conclude from this graph?

Q6 Student Answers:

a) The total passes made over the data will depend on the number of observations in the training split, the learning rate, and the number of iterations. The number of iterations may vary depending on stopping criteria, which may be expressed as a max iterations or the cutoff where the deltas on the weights between iterations is sufficiently small. So let's take the most extreme case where we provided the perfect random starting weights for the model. In this case, there would be only 1 training iteration over the data and one 'prediction' run of the test data for each of the splits, so we would pass over the data 5 times. But in actuality, for each 'split', it will take a number of iterations over the training observations to converge on an acceptable model and then we will run once against the test observations based on this model (for each fold).

b) In this case, we are performing 5-fold cross-validation, so kResiduals will emit 5 tuples for each record in the original dataset. The key of each tuple identifies the record as either part of the 'train' or 'test' set for a particular 'fold'. These keys will allow us to calculate the training and test loss across each fold, which will then be averaged. The 'test' vs 'train' part of the key will determine whether the record goes through the training iterations or is held out for running predictions on the model after it is trained.

e) The training and test errors plot very similarly across the iterations, with a lockstep error reduction, that looks to converge around iteration 30. The 5-fold cross-validation was sufficient to give enough training examples to acheive a "simulated" test performance very close to the "training" model performance. This would give us confidence that this OLS model will have low variance as new data is encountered; although we can only guess that our 5-fold cross validation was sufficient to fully capture the patterns that we are likely to see in the real world. Another positive indicator of model performance is that the test and training errors don't tend seem to diverge over more iterations, indicating that we haven't overfit the model.

```
# part b - helper function to emit residuals (RUN THIS CELL AS IS)
def kResiduals(dataPoint, models, splitNum):
    """
    Compute the (squared) residuals for a data point given k different models.
    Note that points from the k-th split are part of the test set for model
number k
    and part of the training set for all other models. We'll emit a key to
track this.
    Args:
        dataPoint - tuple of (features_array, y)
        models    - list of arrays representing model weights (bias at index 0)
    Returns:
        (stringFormattedKey, squared_error)
    """
    # augment the data point with a bias term at index 0
    X = np.append([1.0], dataPoint[0])
    y = dataPoint[1]
    # emit squared residuals for each model
    for modelNum, W in enumerate(models):
        if modelNum == splitNum:
            yield(f"{modelNum}-test", (W.dot(X) - y)**2)
        else:
            yield(f"{modelNum}-train", (W.dot(X) - y)**2)
```

```
# part d -  run 50 iterations (RUN THIS CELL AS IS)
dataSplits = normedRDD.randomSplit([0.2, 0.2, 0.2, 0.2, 0.2], seed = 2018)
wInit = BASELINE
trainLoss, testLoss, models = GradientDescent_withCV(dataSplits, wInit,
learningRate=0.1, nSteps = 50, verbose = False)
```

```
... trained 50 iterations in 37.4231276512146 seconds
```

```
# part c - fill in the missing code below
def CVLoss(dataSplits, models):
    """
    Compute the k-fold cross-validated test and train loss.
    Args:
        dataSplits - list of RDDs corresponding to the k test splits.
        models     - list of k arrays representing model weights (bias at index
0)
    Returns:
        tuple of floats: (training_loss, test_loss)
    """
    # compute k residuals for each dataPoint (one for each model)
    partialLossRDD = sc.parallelize([])
    for splitNum, splitRDD in enumerate(dataSplits):
        residuals = splitRDD.flatMap(lambda x: kResiduals(x, models, splitNum))
        partialLossRDD = sc.union([partialLossRDD, residuals])

    ################ YOUR CODE HERE #################
    loss = partialLossRDD.map( lambda x: (x[0], (x[1], 1)) ) \
                         .reduceByKey( lambda x,y: [x[0] + y[0], x[1] + y[1]] )
\
                         .mapValues( lambda x: x[0]/x[1] ) \
                         .collect()
    ################ (END) YOUR CODE ################

    test_loss = np.mean([x[1] for x in loss if x[0].split('-')[1] == 'test'])
    training_loss = np.mean([x[1] for x in loss if x[0].split('-')[1] ==
'train'])
    return training_loss, test_loss


# part d - helper function RUN THIS CELL AS IS
def partialGradients(splitNum, dataPoint, models):
    """
    Emit partial gradient for this data point for each model.
    NOTE: a data point from split-number k is in the test set for
    model-k so we don't compute a partial gradient for that model.
    """
    # augment the data point
    X = np.append([1.0], dataPoint[0])
    y = dataPoint[1]
    # emit partial gradients for each model with a counter for averaging later
    for modelNum, W in enumerate(models):
        if modelNum != splitNum:
            yield (modelNum, [(W.dot(X) - y)*X, 1])
```

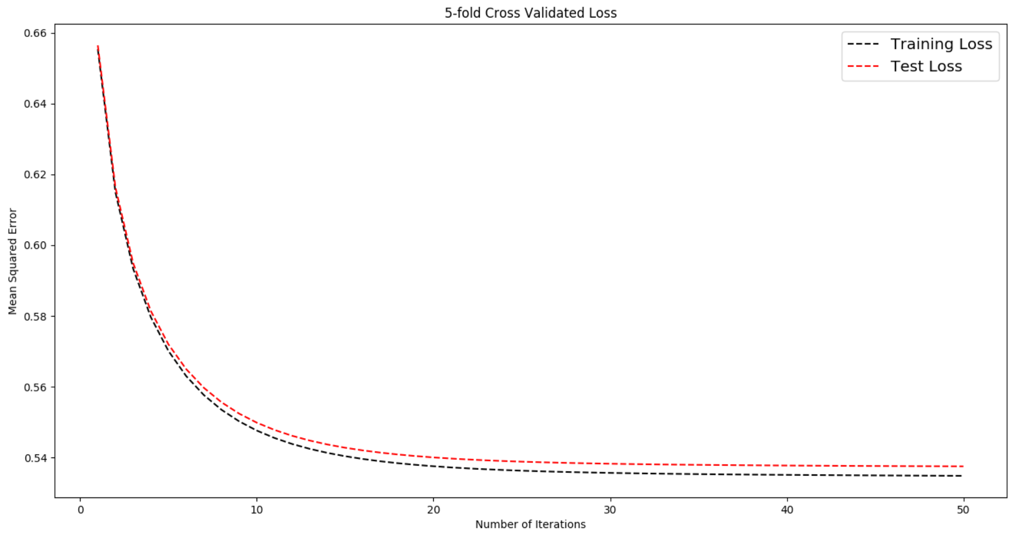
```
# part d - perform GD updates for all k models (FILL IN MISSING CODE BELOW)
def CVUpdate(dataSplits, models, learningRate = 0.1):
    """
    Compute gradients for k models given k corresponding dataSplits.
    NOTE: the training set for model-k is all records EXCEPT those in the k-th
split.
    """
    # compute partial gradient k-1 times for each fold
    partialsRDD = sc.parallelize([])
    for splitNum, splitRDD in enumerate(dataSplits):
        thisFoldPartialGrads = splitRDD.flatMap(lambda x:
partialGradients(splitNum, x, models))
        partialsRDD = sc.union([partialsRDD, thisFoldPartialGrads])

    # compute gradients by taking the average partialGrad for each fold
    gradients = partialsRDD.reduceByKey(lambda a, b: (a[0] + b[0], a[1] +
b[1]))\
                           .mapValues(lambda x: x[0]/x[1])\
                           .map(lambda x: x[1])\
                           .collect()

    # update all k models & return them in a list
    new_models = []
    for W, grad in zip(models, gradients):
        ############# YOUR CODE HERE ############
        update = np.multiply(grad*2,learningRate)
        new_model = W - update
        new_models.append(new_model)
        ############# (END) YOUR CODE ###########
    return new_models
```

```
# part e - RUN THIS CELL AS IS
def GradientDescent_withCV(dataSplits, wInit, learningRate=0.1, nSteps = 5,
verbose = False):
    """
    Train k models in parallel and track cross validated test/train loss.
    Returns:
        train_hist - (list) of floats
        test_hist - (list) of floats
        model_hist - (list) of arrays representing model coefficients (bias at
index 0)
    """
    # broadcast initial models (one for each fold)
    bModels = sc.broadcast([wInit] * len(dataSplits))



    # initialize lists to track performance
    train_loss_0, test_loss_0 = CVLoss(dataSplits, bModels.value)
    train_hist, test_hist, model_hist = [train_loss_0], [test_loss_0], [wInit]

    # perform k gradient updates at a time (one for each fold)
    start = time.time()
    for step in range(nSteps):
        new_models = CVUpdate(dataSplits, bModels.value, learningRate)

        bModels = sc.broadcast(new_models)

        # log progress
        train_loss, test_loss = CVLoss(dataSplits, bModels.value)
        train_hist.append(train_loss)
        test_hist.append(test_loss)
        model_hist.append(new_models[0])

        if verbose:
            print("-------------------")
            print(f"STEP {step}: ")
            print(f"model 1: {[round(w,4) for w in new_models[0]]}")
            print(f" train loss: {round(train_loss,4)}")
            print(f" test loss: {round(test_loss,4)}")

    print(f"\n... trained {nSteps} iterations in {time.time() - start}
seconds")
    return train_hist, test_hist, model_hist
```

```
# part d -  run 50 iterations (RUN THIS CELL AS IS)
dataSplits = normedRDD.randomSplit([0.2, 0.2, 0.2, 0.2, 0.2], seed = 2018)
wInit = BASELINE
trainLoss, testLoss, models = GradientDescent_withCV(dataSplits, wInit,
learningRate=0.1, nSteps = 50, verbose = False)
```

```
... trained 50 iterations in 37.02153205871582 seconds
```

```
# part d - take a look (RUN THIS CELL AS IS)
plotErrorCurves(trainLoss, testLoss, title = '5-fold Cross Validated Loss' )
```



Question 7: Regularization.

Our goal, as always, is to build a linear model that will extend well to unseen data. Chosing the right combination of features to optimize generalizability can be extremely computationally costly given that there are $2^{p}$ potential models that can be built from $p$ features. Traditional methods like forward selection would involve iteratively testing these options to asses which combinations of features achieve a statistically significant prediction.

Ridge Regression and Lasso Regression are two popular alternatives to OLS, which enable us to train generalizable models without the trouble of forward selection and/or manual feature selection. Both methods take advantage of the bias-variance tradeoff by *shrinking* the model coefficients towards 0 which reduces the variance of our model with little increase in bias. In practice this 'shrinkage' is achieved by adding a penalty (a.k.a. 'regularization') term to the means squared error loss function. In this question you will implement Gradient Descent with ridge and lasso regularization.

`IMPORTANT NOTE:` When performing regularization *do not* include the bias in your regularization term calcultion (Recall, that throughout this assignment we've included the bias at index 0 in the vector of weights that is your model).

Q7 Tasks:

- **a) short response:** The regularization term for ridge regression is the square of the $L2$ norm of the weights vector (i.e. the sum of squares of the coefficients) times the regularization parameter, λ. Write the formulas for both the loss function and the gradient for Ridge Regularization and explain what extra step this will add to our gradient descent algorithm.
- **b) short response:** The regularization term for lasso regression is the $L1$ norm of the weights vector (i.e. the sum of the absolute values of the coefficients) times the regularization parameter, λ. Write the formulas for both the loss function and the gradient for Lasso Regularization and explain how the gradient descent update in Lasso will be different than it was in Ridge.
- **c) code:** Fill in the first two missing code blocks in `GDUpdate_wReg()` so that this function will perform a single parameter update using $L2$ regularization if the parameter `regType` is set to `ridge` , $L1$ regularization if set to `lasso` and unregularized OLS otherwise.
- **d) code + short response:** Use the provided code to train 50 iterations of ridge and lasso regression and plot the test/train error. Comment on the curves you see. Does this match your expectation?

Q7 Student Answers:

a) The loss function for OLS Ridge (L2) Regularization is the following:

$$\frac{1}{n}\sum_{i=1}^{n}\left[\boldsymbol{\theta}^{T}\cdot\mathbf{x}_{i}^{'}-y_{i}\right]^{2}+\lambda\sum_{j=1}^{p}(w_{j})^{2}$$

Where the first part of the equation is the average loss squared across all observations and the second part of the equation is the sum of the coefficients (weights only) squared. The gradient function is the following:

$$\frac{2}{n}\sum_{i=1}^{n}\left[\theta^{T}\cdot x_{i}^{'}-y_{i}\right]\cdot\mathbf{x}_{i}^{'}+2\lambda w$$

Where the first part is the derivative of the loss function and the second part the derivative of the L2 regularization. So the extra step we are adding during gradient descent will occur when we are calculating the gradient. Here we will add the regularization term to each weight, which is the dot product of the weights and regularization parameter multiplied by 2. Since we are minimizing the gradient function to zero and adding on this regularization, we are bounding our minimized result by negative of this regularization term This will pull all of the coefficients (bias excluded) closer to zero. In the broader sense, we can think of the regularization term of the gradient equation as a way of 'dampening' the effect of the data on the model we are training. This dampening will increase the bias of the model, but decrease the variance, given that the data has less impact on the weights. We can also think of this as protecting against overfitting to the training data. And since the penalty is more severe as we add more variables (weights) to our model, regularization will essentially yield a simpler model.

b) The loss function for OLS Lasso (L1) Regularization is the following:

$$\frac{1}{n}\sum_{i=1}^{n}\left[\boldsymbol{\theta}^{T}\cdot\mathbf{x}_{i}^{'}-y_{i}\right]^{2}+\lambda\sum_{j=1}^{p}|w_{j}|$$

Where the first part of the equation is the average loss squared across all observations and the second part of the equation is sum of the absolute value of the coefficients (weights, excluding bias term). The gradient function is the following:

$$\frac{2}{n}\sum_{i=1}^{n}\left[\theta^{T}\cdot x_{i}^{'}-y_{i}\right]\cdot\mathbf{x}_{i}^{'}+\lambda sign(w)$$

Where the first part is the derivative of the loss function and the second part the derivative of the L1 regularization. The derivative of an absolute value is either -1 (for negative values) or 1 (for positive values), hence the sign evaluation. As with the ridge regularization, we are bounding the minimum solution to the negative of the regularization term, which will pull all of the coefficients (bias excluded) closer to zero. Where lasso regression differs is that some of the coefficients are more likely to be pulled to exactly zero, providing a computationaly reasonable way to perform feature selection. This behavior of pulling some coefficients to zero is due to the constraint region of lasso regression, which is a diamond shape (as opposed to a circle for ridge regression). Given that this shape defines the region for the optimal model, it is far more likely for a given variable to fall on the point of the diamond shape (the highest point), which intersects with zero. This is because the tangent of the contours of the loss function are must more likely to cross through the point on the diamond (thus intersecting at zero). Another way of thinking about this difference is that the weights in L2 (ridge) regularization are adjusted proportionally to the weight itself, but with L1 (lasso) regularizaion, the weight is adjusted by a constant (either positive or negative depending on the weight) across all of the weights.

d) When compared to the non-regularized OLS, the error curves for ridge regression show less variance (less of a gap in the curves) between training and test loss, especially as the number of iterations increase. And it appears that we acheived slightly better performance on the test data overall. The test error is actually lower than the train error for the first 20 iterations.

The error plots for lasso regression are similar to what we see with ridge regression, but we see more perturbation in the loss, especially in the later iterations. This was not what I expected, but that is probably because I mistakenly equated low variance with a smooth error curve. In fact, we do see lower variance with lasso regression in that the training and test errors are closer. So we know that with lasso regression, some coeffiencts will be set to zero as the weights are updated at each iteration. Perhaps the perturbations we see in the

> test data show a wider swing at those points where a coefficient drops to zero and then a subsequent iteration where another coefficient may drop to zero while a formerly zero coefficient reenters the model (with a coefficient close by not exactly at zero).

```
# part c - gradient descent with regularization
def GDUpdate_wReg(dataRDD, W, learningRate = 0.1, regType = None, regParam =
0.1):
    """
    Perform one gradient descent step/update with ridge or lasso
regularization.
    Args:
        dataRDD - tuple of (features_array, y)
        W       - (array) model coefficients with bias at index 0
        learningRate - (float) defaults to 0.1
        regType - (str) 'ridge' or 'lasso', defaults to None
        regParam - (float) regularization term coefficient
    Returns:
        model   - (array) updated coefficients, bias still at index 0
    """
    # augmented data
    augmentedData = dataRDD.map(lambda x: (np.append([1.0], x[0]), x[1]))

    new_model = None
    ################### YOUR CODE HERE ##################

    def calcGradient(x):
      return (W.dot(x[0]) - x[1]) * x[0]

    def calcRegTerm():
      modelNoBiasTerm = np.array(W[1:])
      if regType == 'ridge':
        regTerm = 2*regParam*modelNoBiasTerm
        regTerm = np.append([0], regTerm)
      elif regType == 'lasso':
        regTerm = regParam*np.sign(modelNoBiasTerm)
        regTerm = np.append([0], regTerm)
      else:
        regTerm = np.zeros(13)
      return regTerm

    regTerm = calcRegTerm()

    grad_agg = augmentedData.map(lambda x: (calcGradient(x), 1)) \
                            .reduce(lambda x,y: (x[0]+y[0],  x[1]+y[1]))

    grad = (2 * grad_agg[0] / grad_agg[1]) + regTerm

    update = np.multiply(grad,learningRate)

    new_model = W - update
```

```
        ################## (END) YOUR CODE ####################
        return new_model
```

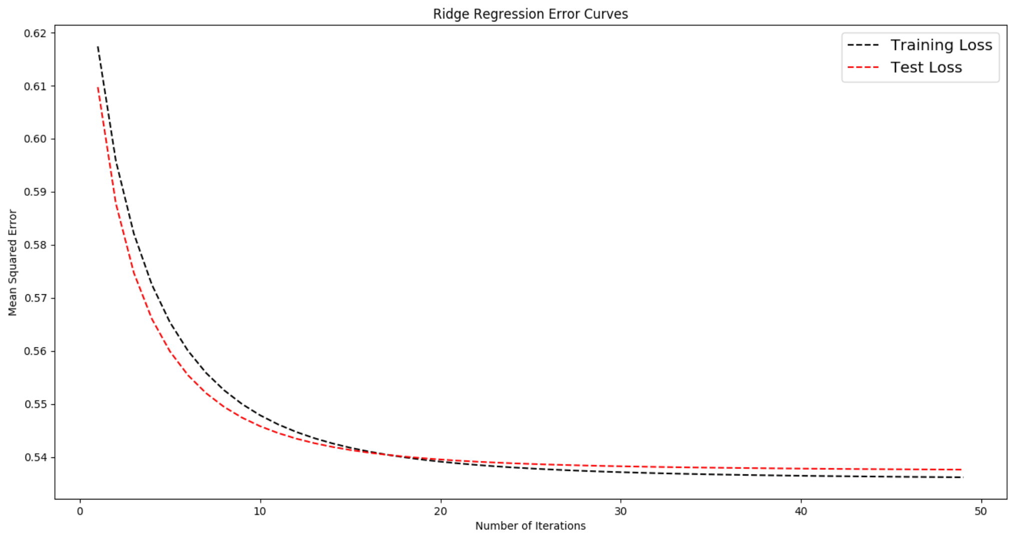
```
# part c - gradient descent with regularization
def GDUpdate_wReg1(dataRDD, W, learningRate = 0.1, regType = None, regParam =
0.1):
    """
    Perform one gradient descent step/update with ridge or lasso
regularization.
    Args:
        dataRDD - tuple of (features_array, y)
        W       - (array) model coefficients with bias at index 0
        learningRate - (float) defaults to 0.1
        regType - (str) 'ridge' or 'lasso', defaults to None
        regParam - (float) regularization term coefficient
    Returns:
        model   - (array) updated coefficients, bias still at index 0
    """
    # augmented data
    augmentedData = dataRDD.map(lambda x: (np.append([1.0], x[0]), x[1]))

    new_model = None
    ################### YOUR CODE HERE ##################

    def calcGradient(x):
      modelNoBiasTerm = np.array(W[1:])

      if regType == 'ridge':
        regTerm = 2*regParam*modelNoBiasTerm
        regTerm = np.append([0], regTerm)
      elif regType == 'lasso':
        regTerm = regParam*np.sign(modelNoBiasTerm)
        regTerm = np.append([0], regTerm)
      else:
        regTerm = np.zeros(13)

      gradient = (W.dot(x[0]) - x[1]) * x[0]

      return gradient + regTerm

    grad_agg = augmentedData.map(lambda x: (calcGradient(x),  1)) \
                            .reduce(lambda x,y: (x[0]+y[0],  x[1]+y[1]))

    grad = 2 * grad_agg[0] / grad_agg[1]

    update = np.multiply(grad,learningRate)

    new_model = W - update
    ################# (END) YOUR CODE ###################
    return new_model
```

```
# part d - ridge/lasso gradient descent function
def GradientDescent_wReg(trainRDD, testRDD, wInit, nSteps = 20, learningRate =
0.1,
                         regType = None, regParam = 0.1, verbose = False):
    """
    Perform nSteps iterations of regularized gradient descent and
    track loss on a test and train set. Return lists of
    test/train loss and the models themselves.
    """
    # initialize lists to track model performance
    train_history, test_history, model_history = [], [], []

    # perform n updates & compute test and train loss after each
    model = wInit
    for idx in range(nSteps):
        # update the model
        model = GDUpdate_wReg(trainRDD, model, learningRate, regType, regParam)

        # keep track of test/train loss for plotting
        train_history.append(OLSLoss(trainRDD, model))
        test_history.append(OLSLoss(testRDD, model))
        model_history.append(model)

        # console output if desired
        if verbose:
            print("----------")
            print(f"STEP: {idx+1}")
            print(f"training loss: {training_loss}")
            print(f"test loss: {test_loss}")
            print(f"Model: {[round(w,3) for w in model]}")
    return train_history, test_history, model_history
```

```
# run 50 iterations of ridge (RUN THIS CELL AS IS)
wInit = BASELINE
trainRDD, testRDD = normedRDD.randomSplit([0.8,0.2], seed = 5)
start = time.time()
ridge_results = GradientDescent_wReg(trainRDD, testRDD, wInit, nSteps = 50,
                                     regType='ridge', regParam = 0.05 )
print(f"\n... trained {len(ridge_results[2])} iterations in {time.time() -
start} seconds")
```

```
... trained 50 iterations in 11.070934534072876 seconds
```

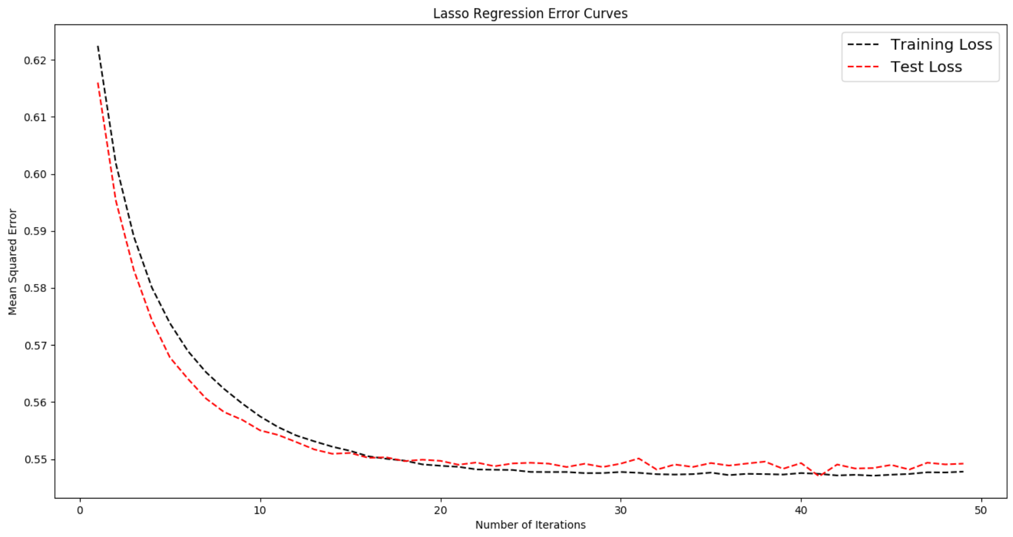
```
# part d - save and display ridge results (RUN THIS CELL AS IS)
trainLoss, testLoss, models = ridge_results
dbutils.fs.put(hw4_path + 'ridge_models.txt', str(models), True)
dbutils.fs.put(hw4_path + 'ridge_loss.txt', str([trainLoss, testLoss]), True)
plotErrorCurves(trainLoss, testLoss, title = 'Ridge Regression Error Curves' )
```



```
# run 50 iterations of lasso (RUN THIS CELL AS IS)
wInit = BASELINE
trainRDD, testRDD = normedRDD.randomSplit([0.8,0.2], seed = 5)
start = time.time()
lasso_results = GradientDescent_wReg(trainRDD, testRDD, wInit, nSteps = 50,
                                     regType='lasso', regParam = 0.05)
print(f"\n... trained {len(lasso_results[2])} iterations in {time.time() -
start} seconds")
```

```
... trained 50 iterations in 10.7092604637146 seconds
```

```
# part d - save and display lasso results (RUN THIS CELL AS IS)
trainLoss, testLoss, models = lasso_results
dbutils.fs.put(hw4_path + 'lasso_models.txt', str(models), True)
dbutils.fs.put(hw4_path + 'lasso_loss.txt', str([trainLoss, testLoss]), True)
plotErrorCurves(trainLoss, testLoss, title = 'Lasso Regression Error Curves' )
```



Question 8: Results

In this final question we'll use a few different plots to help us compare the OLS, Ridge and Lasso models that we have trained. Use the provided code to load the training history from file and retrieve the best (i.e. last) model from each method.

Q8 Tasks:

- **a) code:** Use the provided code to load the training history from file and retrieve the best (i.e. last) model from each method. Then compute the mean squared error on the held out dataset for each of the three models. [**`TIP:`** *the held out data is in it's raw form, don't forget to parse and normalize before applying your calculations, you should also be careful to normalize using the same scaling parameters that you used for the training data.*]

- **b) short response:** Which model performed best? Discuss how you interpret these results and what you would want to try next.

- **c) short response:** Use the provided code to plot side by side boxplots of the residuals vs. the outcome (i.e. `quality`). What can you observe about our model performance? [**`TIP:`** *note that the heldout data set is plenty small enough to fit in* *memory so no need to sample. Feel free to do your plotting in pandas or any other comfortable python package.*]

- **d) short response:** Run the provided code to visualize the model coefficients for the first 50 iterations of training. What do you observe about how the OLS, ridge and lasso coefficients change over the course of the training process. Please be sure to discuss all three in your response.

Q8 Student Answers:

b) OLS regression performed best, with the the lowest Mean Squared error, followed by OLS with Ridge Regularization, followed by OLS with Lasso Regularization. It could be that the regularization parameter wasn't tuned properly but looking at the test vs train error curves for the three methods shows that overfitting was not a big problem to start with. Interestingly, we see that the test and train error cross at iteration 20, so perhaps one also considered model training time, one could argue that the performance difference is negligle and that regularization is still justified to a) shorten training time b) opt for a simpler model that will likely generalize better with new 'unseen' data. So I would like to try tuning the regularization parameter to see if the performance for regularized models can be improved. Then, if convinced that that the regularized model will never yield a lower MSE, I would want to play with the stopping criteria to assess how the MSE's compare at the point were training MSE and test MSE cross over. I would also like to eliminate multicollinearity. So I would focus on eliminating some highly correlated (negative or positive) variables. Also, the color variable should be converted to a dummy variable.

c) First, prediction seems to be worse for low and high quality wines, and prediction is best for 'average' quality wines. This warrants further investigation and suggests that even those observations that we do predict correctly may have less to do with the model and more to do with the distribution of quality over the observations. So plotting a histogram of quality (labels) shows that we have a normal distribution with most quality values landing around the center value of '6'. Here we see that we have very few observations for quality lower than 4 or higher than 7. Next to this, we show the distribution predicted quality and see that predictions are bounded to a tighter range. The next plot further illustrates this as we see a band of predictions between quality 4 and 6, even though there are observations for lower and higher quality wines. Although we could explain part of this prediction problem due to lack of enough observations at the lower and higher ends of quality, it doesn't entirely explain why these predictions were missed. It could be that we don't have adequate variables to fully describe quality at these upper and lower bounds or it could be that we have multicollinerity that our regularization is failing to eliminate. Or it could be that the a linear model isn't the best model for describing wine quality. For example, how do we know that the differences between each quality step is linear?

For the last plot, I wanted to look at the coefficients across OLS alone and the 2 different regularizations. Here we see that Lasso regularization eliminated a number of the variables in the model, while Ridge regularization only modestly dampened the coefficients. But in both cases, we didn't acheive a better performing model, so there is not supporting evidence to suggest that we have a problem of overfitting or multicollinearity.

d) First, all coefficients seem to stabilize around iteration 20. Before that, we see quite a bit of fluctuation, across all three models. Some coefficients have a larger (positive) coefficient across all models and all iterations and similarly, some have a larger negative coefficient across all models and iterations. We see more divergence across the three models for the smaller coefficients (e.g. alcohol, sulfur, pH), which we would expect, as we will see a more pronounced effect of regularization on these smaller coefficients. In the case of ridge regression, we see more fluctuation around zero for these smaller coefficients. We observe some variables dropping out and then back in (0 to non-zero coefficient) across the iterations but as we get closer to iteration 50, we see more of these coefficients settle at or very close to 0. This behavior is different than ridge regression, in that we see a smoother pull toward zero for these smaller coefficients across the iterations.

```
# part a - load the models from file (RUN THIS CELL AS IS)
ridge_models = open(hw4_path_open + 'ridge_models.txt', 'r').read()
lasso_models = open(hw4_path_open + 'lasso_models.txt', 'r').read()
ols_models = open(hw4_path_open + 'ols_models.txt', 'r').read()
ridge_models = np.array(ast.literal_eval(re.sub(r'[()\n]','',re.sub('array',
'', ridge_models))))
lasso_models = np.array(ast.literal_eval(re.sub(r'[()\n]','',re.sub('array',
'', lasso_models))))
ols_models = np.array(ast.literal_eval(re.sub(r'[()\n]','',re.sub('array', '',
ols_models))))
best_ols = ols_models[-1,:]
best_ridge = ridge_models[-1,:]
best_lasso = lasso_models[-1,:]
```

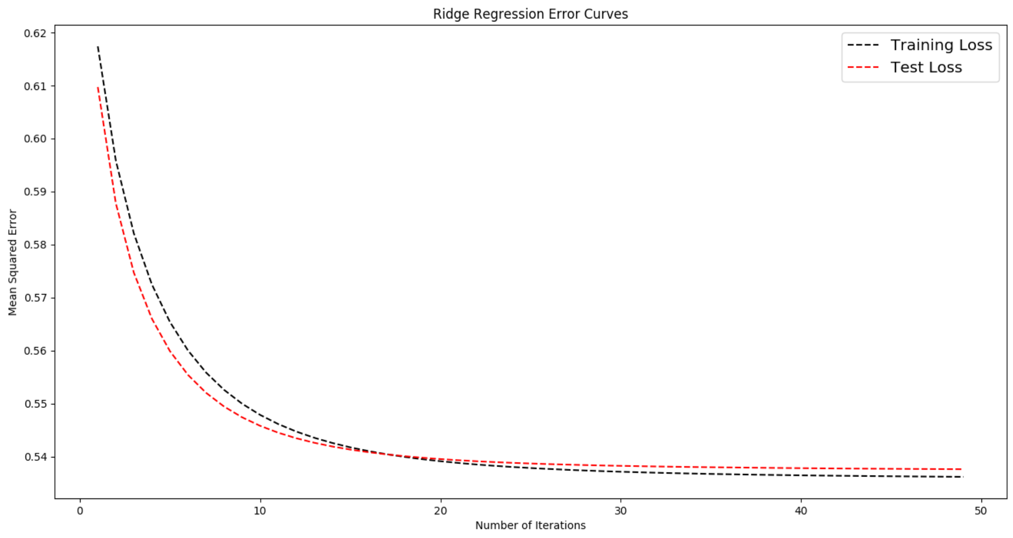
```
# part a - compute MSE on the held out data for all three 'best' models
olsMSE, ridgeMSE, lassoMSE = None, None, None
############### YOUR CODE HERE #################
heldOutRDDCached = heldOutRDD.map(parse).cache()
validationRDD = normalize(heldOutRDDCached).cache()

olsMSE = OLSLoss(validationRDD, best_ols)
ridgeMSE = OLSLoss(validationRDD, best_ridge)
lassoMSE = OLSLoss(validationRDD, best_lasso)
############### YOUR CODE HERE #################

print(f"OLS Mean Squared Error: {olsMSE}")
print(f"Ridge Mean Squared Error: {ridgeMSE}")
print(f"Lasso Mean Squared Error: {lassoMSE}")
```

```
OLS Mean Squared Error: 0.552893063319935
Ridge Mean Squared Error: 0.5537596776782171
Lasso Mean Squared Error: 0.561038670486936
```

```
# part c - helper function (RUN THIS CELL AS IS)
def get_residuals(dataRDD, model):
    """
    Return a collected list of tuples (residual, quality_score)
    """
    augmentedData = dataRDD.map(lambda x: (np.append([1.0], x[0]), x[1]))
    residuals = augmentedData.map(lambda x: (x[1] - model.dot(x[0]), x[1]))
    return residuals.collect()
```

```
# part c - compute residuals for all three models (RUN THIS CELL AS IS)
ols_resid = np.array(get_residuals(validationRDD, best_ols))
ridge_resid = np.array(get_residuals(validationRDD, best_ridge))
lasso_resid = np.array(get_residuals(validationRDD, best_lasso))
```

```
# part d - save and display ridge results (RUN THIS CELL AS IS)
trainLoss, testLoss, models = ridge_results
dbutils.fs.put(hw4_path + 'ridge_models.txt', str(models), True)
dbutils.fs.put(hw4_path + 'ridge_loss.txt', str([trainLoss, testLoss]), True)
plotErrorCurves(trainLoss, testLoss, title = 'Ridge Regression Error Curves' )
```



```
colors = {'OLS': '#8fb599', 'Ridge': '#cfb586', 'Lasso': '#8f91b5'}

def showPredictors():
  raw_data = {'variable': FIELDS[1:-1],
          'ols': best_ols[1:-1],
          'ridge': best_ridge[1:-1],
          'lasso': best_lasso[1:-1]}
  raw_data
  df = pd.DataFrame(raw_data, columns = ['variable', 'ols', 'ridge', 'lasso'])
  df['ols']

  pos = list(range(len(df['ols'])))
  width = 0.25
  fig, ax = plt.subplots(figsize=(20,5))
  # Create a bar with pre_score data,
  # in position pos,
  plt.bar(pos,
          df['ols'],
          width,
          color=colors['OLS'])

  plt.bar([p + width for p in pos],
          df['ridge'],
          width,
          color=colors['Ridge'])

  plt.bar([p + width*2 for p in pos],
          df['lasso'],
          width,
          color=colors['Lasso'])

  ax.set_xticks([p + 1.5 * width for p in pos])
  ax.set_xticklabels(df['variable'])


  plt.legend(['OLS', 'Ridge', 'Lasso'], loc='upper left')
  plt.grid()
  display(plt.show())
```

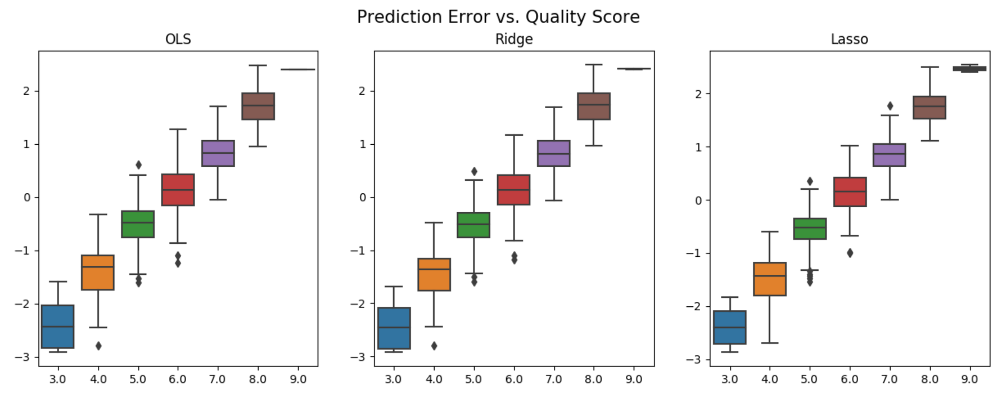
```
# part c - boxplots of residuals for all three models (RUN THIS CELL AS IS)
fig, axes = plt.subplots(1, 3, figsize=(15,5))
stuff_to_plot = zip(axes, ["OLS", "Ridge", "Lasso"], [ols_resid, ridge_resid,
lasso_resid])
for ax, title, data in stuff_to_plot:
    ax.set_title(title)
    y = data[:, 0]
    x = data[:, 1]
    sns.boxplot(x, y, ax=ax)
fig.suptitle("Prediction Error vs. Quality Score", fontsize=15, y=0.98)
display(plt.show())
```

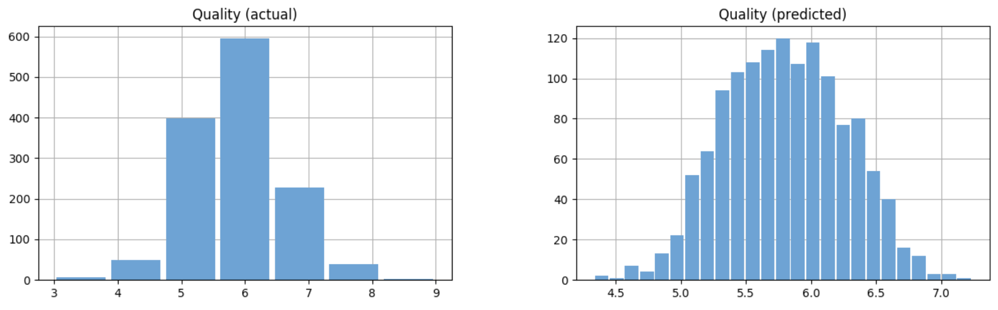
```
def get_predicted(dataRDD, model):
    """
    Return a collected list of tuples (residual, quality_score)
    """
    augmentedData = dataRDD.map(lambda x: np.append([1.0], x[0]) )
    predicted = augmentedData.map(lambda x: model.dot(x) )

    return predicted.collect()
```

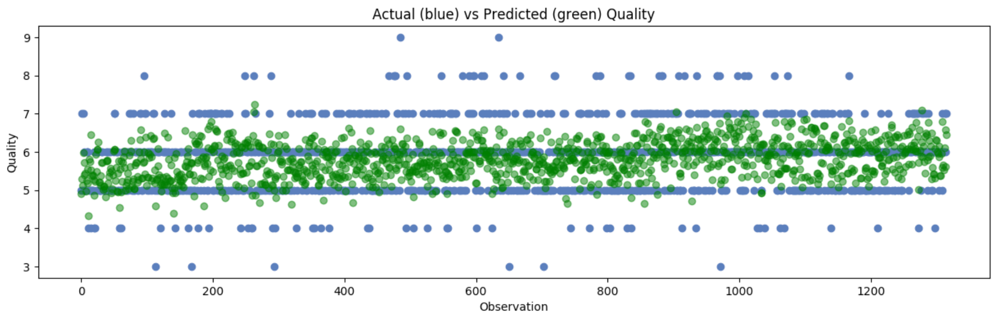
```
ols_predicted = np.array(get_predicted(validationRDD, best_ols))
```

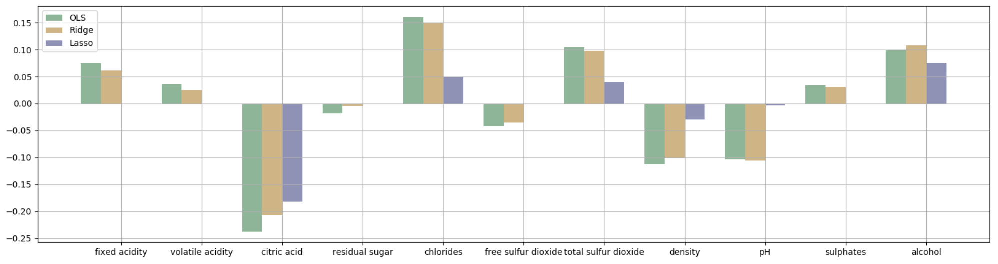
```
fig, axes = plt.subplots(1, 2, figsize=(15,4))

qual_df = pd.DataFrame(ols_resid, columns = ['error', 'Quality (actual)'])
qual_df[['Quality (actual)']].hist(ax=axes[0], bins=7,  color='#6ea3d4',
zorder=2, rwidth=0.9)

qual_pred_df = pd.DataFrame(ols_predicted, columns = ['Quality (predicted)'])
qual_pred_df[['Quality (predicted)']].hist(ax=axes[1], bins=25,
color='#6ea3d4', zorder=2, rwidth=0.9)
display(plt.show())

fig, axes = plt.subplots(1, 1, figsize=(15,4))
axes.plot(qual_df[['Quality (actual)']], 'o', color="#5b7fbd")
axes.plot(qual_pred_df[['Quality (predicted)']], 'go', alpha=.5)
plt.xlabel('Observation')
plt.ylabel('Quality')
plt.title('Actual (blue) vs Predicted (green) Quality')
display(plt.show())
```

```
showPredictors()
```



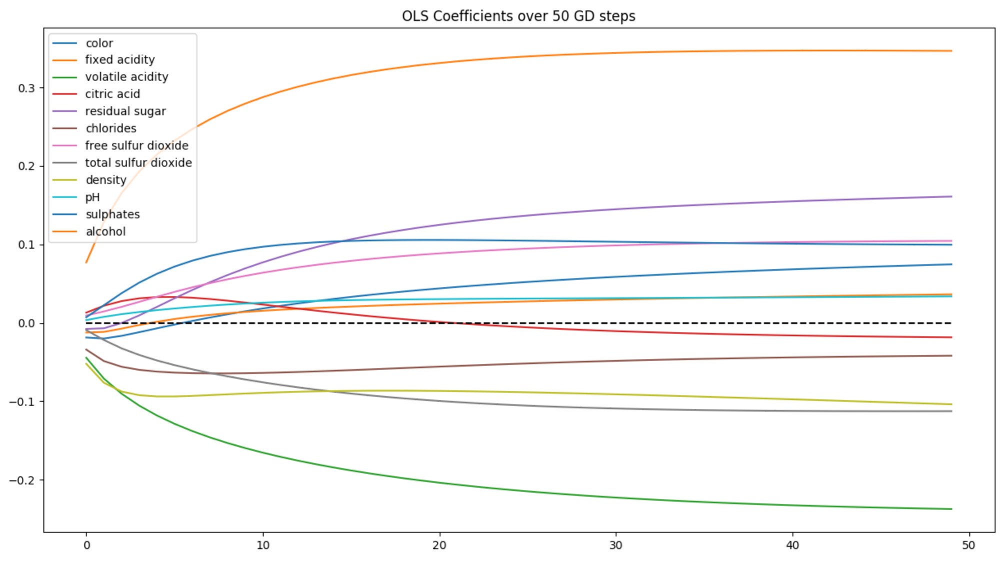




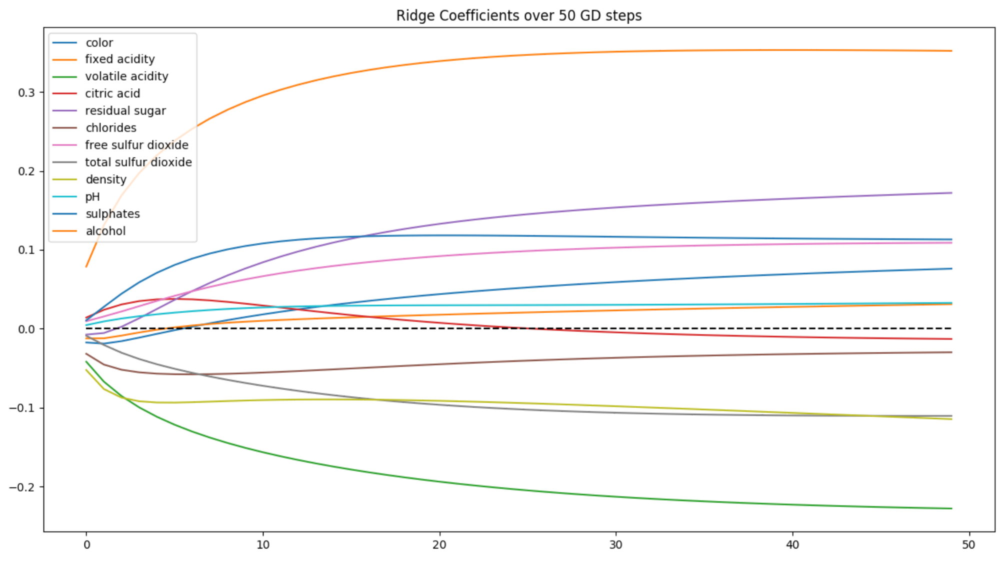


```
# part d - plotting function (RUN THIS CELL AS IS)
def plotCoeffs(models, featureNames, title):
    """
    Helper Function to show how coefficients change as we train.
    """
    fig, ax = plt.subplots(figsize = (15,8))
    X = list(range(len(models)))
    for data, name in zip(models.T, featureNames):
        if name == "Bias":
            continue
        ax.plot(X, data, label=name)
    ax.plot(X,[0]*len(X), 'k--')
    plt.title(title)
    plt.legend()
    display(plt.show())
```

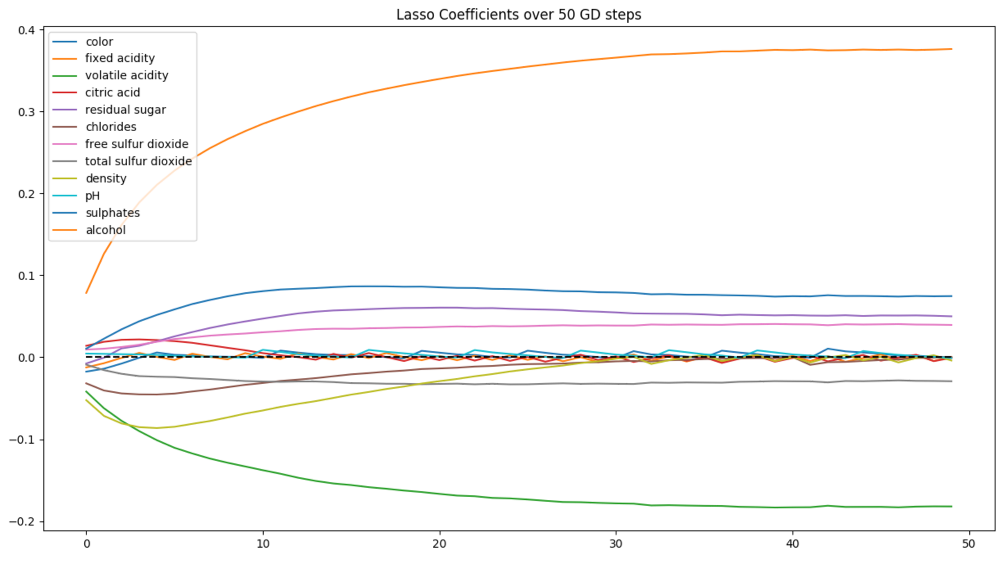
```
# take a look (RUN THIS CELL AS IS)
plotCoeffs(ols_models, ['Bias'] + FIELDS, "OLS Coefficients over 50 GD steps")
```



```
# take a look (RUN THIS CELL AS IS)
plotCoeffs(ridge_models, ['Bias'] + FIELDS, "Ridge Coefficients over 50 GD
steps")
```



```
# take a look (RUN THIS CELL AS IS)
plotCoeffs(lasso_models, ['Bias'] + FIELDS, "Lasso Coefficients over 50 GD
steps")
```



Congratulations, you have completed HW4! Please refer to the readme for submission instructions.

If you would like to provide feedback regarding this homework, please use the survey at:
https://docs.google.com/forms/d/e/1FAIpQLScgIz4laP2JHChStLZx8MO0jGvrGyrOyQ
usp=sf_link
(https://docs.google.com/forms/d/e/1FAIpQLScgIz4laP2JHChStLZx8MO0jGvrGyrOyC
usp=sf_link)