VIRGINIA POLYTECHNIC INSTITUTE AND STATE UNIVERSITY

ISE 5406 - OPTIMIZATION – II

Mini Project – 2 (Report)

01 May 2018

Honor Code:

"On my honor, as a Hokie, I have neither given nor received unauthorized aid on this academic work".

Rohan Joseph
Saurabh Mehra

I. Literature Review

Types of solvers in R:

There are several libraries available in R that can perform convex optimization over both constrained and unconstrained problems. A few of the important ones are discussed below:

i) ***"nloptr"*** : A free/open-source library in R for nonlinear optimization started by Steven G. Johnson, providing a common interface for a number of different free optimization routines available online as well as original implementations of various other algorithms.

NLopt addresses general nonlinear optimization problems of the form:

$$\begin{aligned} & \min_{x \in R^n} f(x) \\ s.t. \quad & g(x) \leq 0 \\ & h(x) = 0 \\ & x_L \leq x \leq x_U \end{aligned}$$

where f(·) is the objective function and x represents the n optimization parameters. This problem may optionally be subject to the bound constraints (also called box constraints), xL and xU . For partially or totally unconstrained problems the bounds can take values −∞ or ∞. One may also optionally have m nonlinear inequality constraints (sometimes called a nonlinear programming problem), which can be specified in g(·), and equality constraints that can be specified in h(·).

ii) ***"rsolnp" :*** A free/open-source library in R for General Non-linear Optimization Using Augmented Lagrange Multiplier Method

The solnp function is based on the solver by Yinyu Ye which solves the general nonlinear programming problem:

$$\begin{aligned} & \min f(x) \\ & \text{s.t.} \\ & g(x) = 0 \\ & l_h \leq h(x) \leq u_h \\ & l_x \leq x \leq u_x \end{aligned}$$

where, $f(x)$, $g(x)$ and $h(x)$ are smooth functions.

Types of solvers in Python:

Similar to R, there are several libraries available in Python that performs a variety of optimization algorithms over constrained and unconstrained problems. The **scipy.optimize** is one of the most popular package that provides several commonly used optimization algorithms.

The module contains:
Unconstrained and constrained minimization of multivariate scalar functions (minimize) using a variety of algorithms (e.g. BFGS, Nelder-Mead simplex, Newton Conjugate Gradient, COBYLA or SLSQP), Global (brute-force) optimization routines (e.g. basinhopping, differential_evolution) Least-squares minimization (least_squares) and curve fitting (curve_fit) algorithms, Scalar univariate functions minimizers (minimize_scalar) and root finders (newton), Multivariate equation system solvers (root) using a variety of algorithms (e.g. hybrid Powell, Levenberg-Marquardt or large-scale methods such as Newton-Krylov).

A few key unconstrained optimizations algorithms provided in scipy.optimize are:

- **Method *Powell*** is a modification of Powell's method which is a conjugate direction method. It performs sequential one-dimensional minimizations along each vector of the directions set (*direc* field in *options* and *info*), which is updated at each iteration of the main minimization loop. The function need not be differentiable, and no derivatives are taken.
- **Method *CG*** uses a nonlinear conjugate gradient algorithm by Polak and Ribiere, a variant of the Fletcher-Reeves method. Only the first derivatives are used.
- **Method *BFGS*** uses the quasi-Newton method of Broyden, Fletcher, Goldfarb, and Shanno (BFGS). It uses the first derivatives only. BFGS has proven good performance even for non-smooth optimizations. This method also returns an approximation of the Hessian inverse, stored as *hess_inv* in the OptimizeResult object.
- **Method *Newton-CG*** uses a Newton-CG algorithm (also known as the truncated Newton method). It uses a CG method to the compute the search direction. *TNC* method for a box-constrained minimization with a similar algorithm.
- **Batch Gradient Descent**

$$\theta_j := \theta_j + \alpha \sum_{i=1}^{m} \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)} \qquad (\text{for every } j).$$

The quantity in the summation in the update rule above is just ∂J(θ)/∂θj (for the original definition of J). So, this is simply gradient descent on the original cost function J. This method looks at every example in the entire training set on every step, and is called **batch gradient descent.** Note that, while gradient descent can be susceptible to local minima in general, the optimization problem we have posed here for linear regression has only one global, and no other local, optima; thus gradient descent always converges (assuming the learning rate α is not too large) to the global minimum.

II. Problem Definition

a. Problem 1: **Definition**

Moneyball in NBA

The objective here is to determine whether a NBA team will make it to playoffs in a season based on the point difference between points scored and points conceded.

Train a model to determine the points difference of a NBA team in a season based on historical data of **~30 teams from 1980 to 2011 seasons**. Use this trained model to predict the points difference of a team in **2013 season**.

Based on historical data, if a team has a point difference of greater than **31 points**, it has **98% chance** of making it to playoffs.

Feature set:

FieldGoals	FieldGoalsA	2P	2PA	3P	3PA	FreeThrows
FreeThrowsA	OffenseRebound	DefenseRebound	Assists	Steals	Blocks	Turnovers

b. Problem 1: **Methodology**

Ridge Regression

Ridge regression performs multiple linear regression with the regularization term to prevent over-fitting over the training data when a large set of features are present. The cost function for ridge regression can be written as:

$$\text{minimize} \sum_{i=1}^{n}(y_i - \beta^\top \mathbf{z}_i)^2 \text{ s.t. } \sum_{j=1}^{p}\beta_j^2 \leq t$$

$$\Leftrightarrow \text{ minimize } (y - \mathbf{Z}\beta)^\top (y - \mathbf{Z}\beta) \text{ s.t. } \sum_{j=1}^{p}\beta_j^2 \leq t$$

We can write the ridge constraint as the following penalized residual sum of squares (PRSS):

$$\begin{aligned} PRSS(\beta)_{\ell_2} &= \sum_{i=1}^{n}(y_i - \mathbf{z}_i^\top \beta)^2 + \lambda \sum_{j=1}^{p}\beta_j^2 \\ &= (\mathbf{y} - \mathbf{Z}\beta)^\top (\mathbf{y} - \mathbf{Z}\beta) + \lambda ||\beta||_2^2 \end{aligned}$$

- Its solution may have smaller average prediction error than β^ ls
- PRSS(β)ℓ2 is convex, and hence has a unique solution

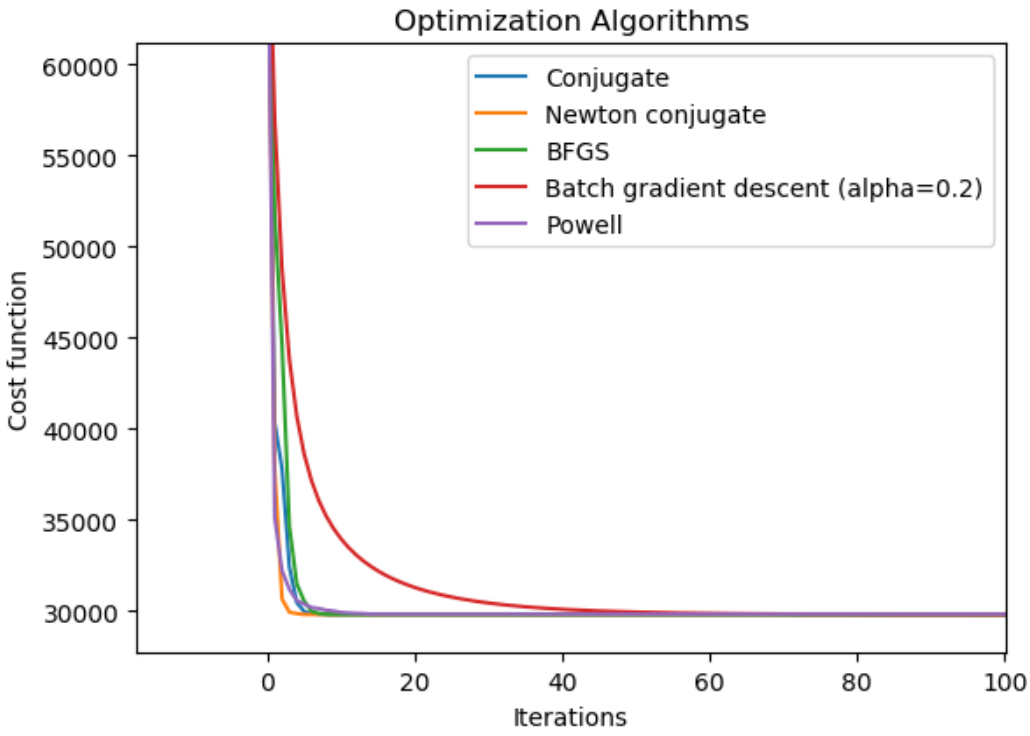
Taking derivatives, we obtain:

$$\frac{\partial PRSS(\beta)_{\ell_2}}{\partial \beta} = -2\mathbf{Z}^\top (y - \mathbf{Z}\beta) + 2\lambda\beta$$

c. Problem 1: **Computational results**

Unconstrained Optimization Algorithms used:

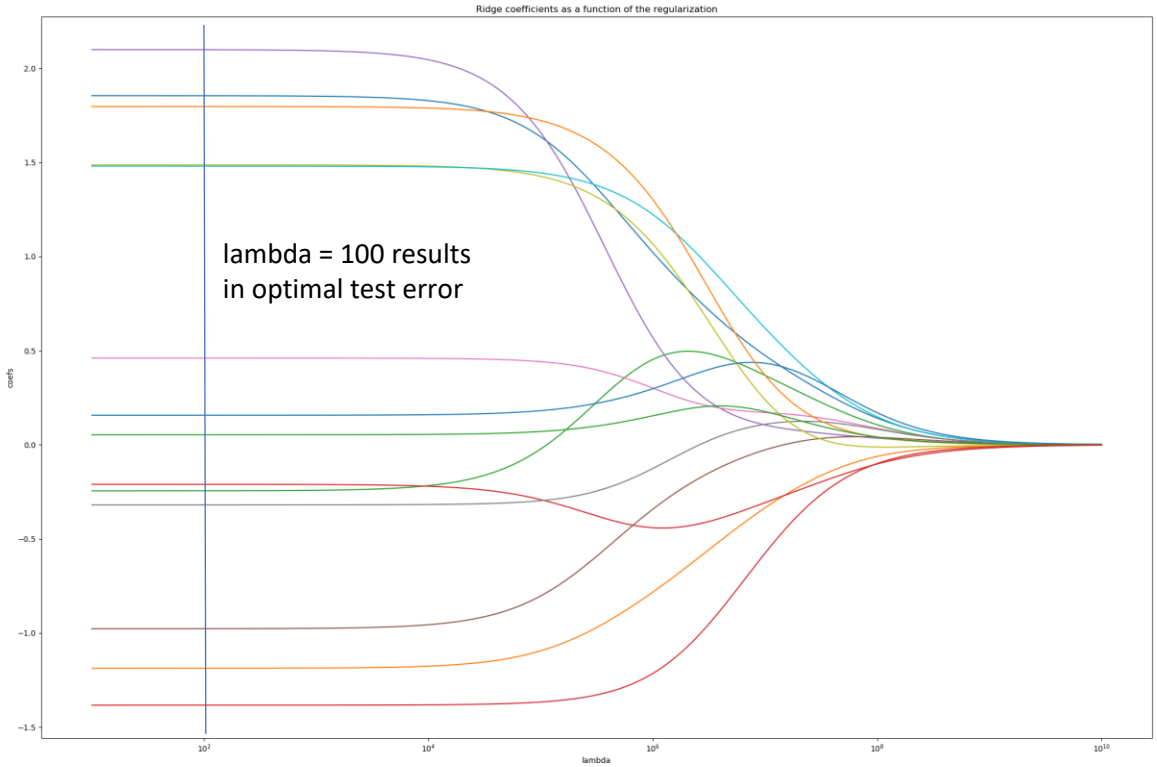
1) **Conjugate gradient descent:** Polak - Ribiere modification. Converges to global minimum in ~10 iterations.
2) **Newton conjugate gradient:** Fastest to converge (~6 iterations). Converges to global minimum.
3) **Quasi Newton method (BFGS):** Converges to global minimum in ~12 iterations.
4) **Batch gradient descent:** Takes longest to converge but converges to global minimum in ~90 iterations. For alpha>0.2, the method does not converge.
5) **Powell conjugate direction method:** Converges in ~15 iterations but do not converge to global minimum.



Ridge regression results accuracy:	
Optimal Objective value	29817
RMSE test data	174.73
R-2 coefficient for test data	0.8714

Of the 14 teams that qualified for the playoffs in the 2013 season, we accurately predicted 12 of them based on the point difference figures predicted from the model.

λ is the Langrangian dual parameter. λ controls the size of the coefficients. As λ ↓ 0, we obtain the least squares solutions As λ ↑ ∞, we have βˆ ridge = 0 (intercept-only model)



Support Vector Machine (SVM) is a supervised machine learning algorithm which can be used for both classification or regression challenges. However, it is mostly used in classification problems. In this algorithm, we plot each data item as a point in n-dimensional space (where n is number of features you have) with the value of each feature being the value of a particular coordinate. Then, we perform classification by finding the hyper-plane that differentiate the two classes very well (Figure below).

d. Problem 2: **Definition**

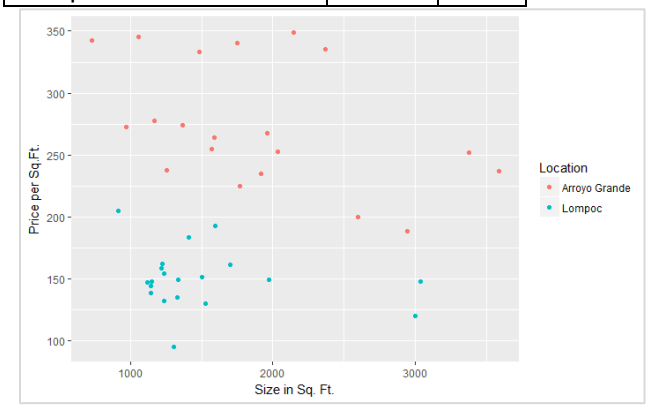
Classify the location of a house based on two attributes – Size (Square feet) and Price per square feet. The class is 'Location'.

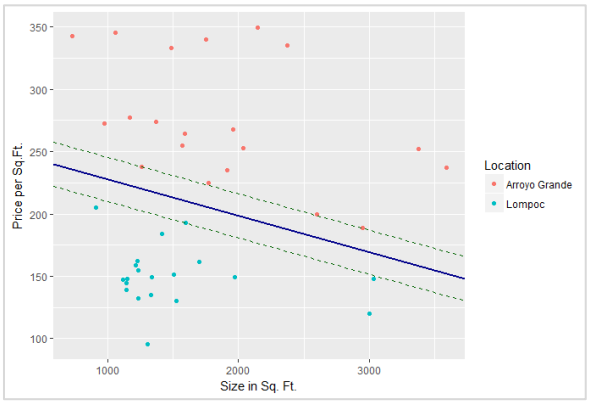
Dataset :

- Location : City in California (Arroyo Grande/Lompoc)
- Price per Square Feet
- Size in Square Feet

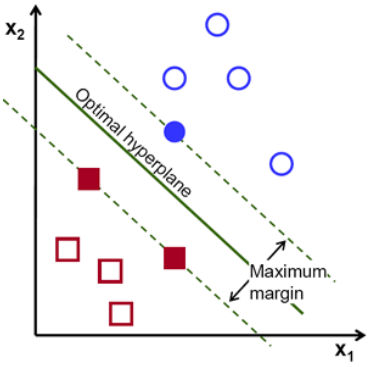
Sample :

Location	Price	Size
Arroyo Grande	335.3	2371
Arroyo Grande	237.87	1257
Arroyo Grande	340	1750
Arroyo Grande	333.11	1486
Arroyo Grande	349.18	2145
Arroyo Grande	188.63	2947
Arroyo Grande	240.63	827
Lompoc	130.32	1527
Arroyo Grande	220.44	6800
Lompoc	161.76	1700
Arroyo Grande	351.14	5411
Lompoc	204.93	912





e. Problem 2: **Methodology**
Support Vector Machines



How is the optimal hyperplane computed?

Formally define a hyperplane :

$$f(x) = \beta_0 + \beta^T x \qquad (1)$$

Where, β = weight vector and β_0 = bias

Optimal hyperplane can be represented in an infinite number of different ways by scaling of β and β_0. The conventional representation is :

$$|\beta_0 + \beta^T x| = 1 \quad (2)$$

x symbolized the training examples closest to the hyperplane and in general they are called the support vectors. The distance between a point x and a hyperplane (β, β_0) :

$$\text{distance} = \frac{|\beta_0 + \beta^T x|}{||\beta||} \quad (3)$$

For the canonical hyperplane, the numerator is equal to 1 and the distance to the support vectors is :

$$distance_{support\ vectors} = \frac{|\beta_0 + \beta^T x|}{||\beta||} = \frac{1}{||\beta||} \quad (4)$$

The margin (M) defined in the figure, is twice the distance to the closest examples :

$$M = \frac{2}{||\beta||} \quad (5)$$

Maximizing the margin M, is equivalent to minimizing a function L(β) subject to some constraints:

$$\min L(\beta) = \frac{1}{2}\left|\left|\beta^2\right|\right| \quad (6)$$

subject to, $y_i(\beta^T x_i + \beta_0) \geq 1$ (7)

where, $y_i = labels\ of\ the\ training\ examples$

This is a problem of Lagrangian optimization that can be solved using Lagrange multipliers to obtain the weight vector β and the bias β_0

To solve this optimization problem, we are using ***Augmented Lagrangier method***

Let us say we are solving the following constrained problem:

$$\min f(\mathbf{x})$$

subject to

$$c_i(\mathbf{x}) = 0 \ \forall i \in I.$$

The augmented Lagrangian method uses the following unconstrained objective:

$$\min \Phi_k(\mathbf{x}) = f(\mathbf{x}) + \frac{\mu_k}{2} \sum_{i \in I} c_i(\mathbf{x})^2 - \sum_{i \in I} \lambda_i c_i(\mathbf{x})$$

and after each iteration, in addition to updating μ_k, the variable λ is also updated according to the rule

$$\lambda_i \leftarrow \lambda_i - \mu_k c_i(\mathbf{x}_k)$$

where $\mathbf{x}_k$ is the solution to the unconstrained problem at the *k*th step, i.e. $\mathbf{x}_k = \operatorname{argmin} \Phi_k(\mathbf{x})$

The variable is an estimate of the Lagrange multiplier, and the accuracy of this estimate improves at every step. The major advantage of the method is that unlike the penalty method, it is not necessary to take μ -> ∞ in order to solve the original constrained problem. Instead, because of the presence of the Lagrange multiplier term, μ can stay much smaller, thus avoiding ill-conditioning.

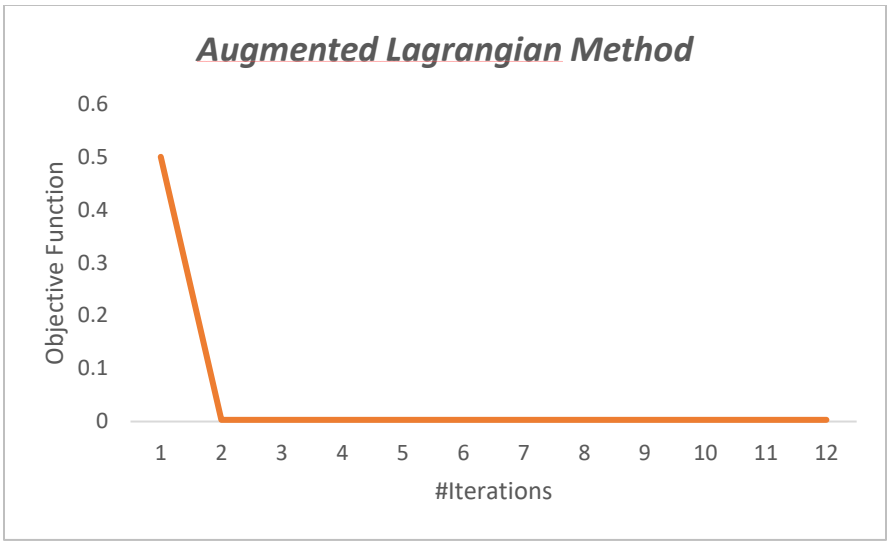
f. Problem 2: **Computational results**

Optimal hyperplane equation: $-0.05639002x_1-0.001643451x_2+14.48 = 0$

Training Error = 0%			
	Predicted		
Actual		***Arroyo Grande***	***Lompoc***
	Arroyo Grande	***20***	***0***
	Lompoc	***0***	***20***

Testing Error = 9.09%			
	Predicted		
Actual		**Arroyo Grande**	**Lompoc**
	Arroyo Grande	**7**	**1**
	Lompoc	**0**	**3**

Also, the constrained optimization problem has been solved using Augmented Lagrangier method and the objective function converges within 3 iterations:



III. References

1. *https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/*
2. *https://data-flair.training/blogs/applications-of-svm/*
3. *https://www.kaggle.com/c/house-prices-advanced-regression-techniques*
4. *https://cran.r-project.org/web/packages/nloptr/nloptr.pdf*
5. *https://cran.r-project.org/web/packages/Rsolnp/Rsolnp.pdf*
6. *https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html*

IV. Code

Problem 1: Solution using Ridge Regression in Python :

```
# -*- coding: utf-8 -*-
"""
Created on Sat Apr 28 16:52:44 2018
@author: Saurabh Mehra
"""
# Import required libraries
import os # used for manipulating directory paths
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import csv
from sklearn import datasets, linear_model
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.linear_model import Ridge
from scipy import optimize
```

```
from sklearn.metrics import mean_squared_error
from math import sqrt
import matplotlib.pyplot as plt

#  training data stored in arrays X, y
nba_train = pd.read_csv('./Data/NBA_train.csv')
nba_test = pd.read_csv('./Data/NBA_test.csv')

def to_matrix(df,start,end):
    X = df.as_matrix(columns = df.columns[start-1:end])
    return X

def normalize(X):
    m, n = X.shape
    X_mean = np.zeros((1,n))
    X_std = np.zeros((1,n))
    X_norm = X

    X_mean = np.mean(X, axis=0)
    X_std = np.std(X, axis=0)
    X_norm = (X - X_mean)/X_std

    return X_norm, X_mean, X_std

def CostFunc(Theta, X, Y, num_rows, num_features, lamda):
    Theta = Theta.reshape(num_features,1)
    J = (1/(2*num_rows))*np.sum((X.dot(Theta) - Y)**2) + (lamda/(2*num_rows))*np.sum(Theta[1:]**2)

    return J

def GradFunc(Theta, X, Y, num_rows, num_features, lamda):
    Theta = Theta.reshape(num_features,1)
    Theta_grad = np.zeros(Theta.shape)
    Theta_grad[0] = (1/num_rows)*np.sum(X.dot(Theta) - Y)
    Theta_grad[1:] = (1/num_rows)*((X[:,1:].T).dot(X.dot(Theta) - Y)) + (lamda/num_rows)*Theta[1:]
    Theta_grad = Theta_grad.flatten()

    return Theta_grad

def batch_gradient_descent(X, Y, Theta, alpha, lamda, num_rows, num_features, num_iters):
    m = Y.size
    j_history = np.zeros((num_iters+1))
    Theta = Theta.reshape(num_features,1)
    Theta_grad = np.zeros(Theta.shape)
    j_history[0] = (1/(2*m))*np.sum((X.dot(Theta) - Y)**2) + (lamda/(2*num_rows))*np.sum(Theta[1:]**2)

    for i in range(num_iters):
        Theta_grad[0] = (1/num_rows)*np.sum(X.dot(Theta) - Y)
```

```
        Theta_grad[1:] = (1/num_rows)*((X[:,1:].T).dot(X.dot(Theta) - Y)) + (lamda/num_rows)*Theta[1:]
        Theta -= alpha * Theta_grad
        j_history[i+1] = (1/(2*m))*np.sum((X.dot(Theta) - Y)**2) + (lamda/(2*num_rows))*np.sum(Theta[1:]**2)

    return Theta, j_history

# Input data and convert to matrix form
X = to_matrix(nba_train,8,21)
Y = to_matrix(nba_train,7,7)
X_test = to_matrix(nba_test,8,21)
Y_test = to_matrix(nba_test,7,7)

m,n = X.shape
m_test, n_test = X_test.shape

# Normalize data
X_norm, X_mean, X_std = normalize(X)
X_test_norm, X_test_mean, X_test_std = normalize(X_test)

## Add intercept term to X
X_0 = np.concatenate((np.ones((m, 1)), X_norm), axis=1)
X_test_0 = np.concatenate((np.ones((m_test, 1)), X_test_norm), axis=1)

# Initial value of the Theta parameter
Theta_0 = np.zeros((n+1))

# Defining system parameter values
lamda = 100
args = (X_0, Y, m, n+1, lamda)  # arguments values

# Batch gradient descent algorithm
theta_, j_batch_gd = batch_gradient_descent(X_0, Y, Theta_0, alpha=0.2, lamda=lamda, num_rows = m, num_features = n+1, num_iters=500)
Theta_0 = np.zeros((n+1))

n_iters = 501

# Modified Powell's optimization algorithm
j_powell = np.zeros((n_iters))
j_powell[0] = CostFunc(Theta_0, *args)

for i in range(1, n_iters):
    res = optimize.fmin_powell(CostFunc, Theta_0, args=args, maxiter = i, disp = 0)
    j_powell[i] = CostFunc(res, *args)

# BFGS optimization algorithm
j_bfgs = np.zeros((n_iters))
j_bfgs[0] = CostFunc(Theta_0, *args)
```

```
for i in range(1, n_iters):
    res = optimize.fmin_bfgs(CostFunc, Theta_0, fprime=GradFunc, args=args, maxiter = i, disp = 0)
    j_bfgs[i] = CostFunc(res, *args)

# Newton Conjugate gradient optimization algorithm
j_ncg = np.zeros((n_iters))
j_ncg[0] = CostFunc(Theta_0, *args)

for i in range(1, n_iters):
    res = optimize.fmin_ncg(CostFunc, Theta_0, fprime=GradFunc, args=args, maxiter = i, disp = 0)
    j_ncg[i] = CostFunc(res, *args)

# Conjugate gradient optimization algorithm
j_cg = np.zeros((n_iters))
j_cg[0] = CostFunc(Theta_0, *args)

for i in range(1, n_iters):
    res = optimize.fmin_cg(CostFunc, Theta_0, fprime=GradFunc, args=args, maxiter = i, disp = 0)
    j_cg[i] = CostFunc(res, *args)

# Plot the cost function for each model
X_ax = np.arange(0, 501, 1)
plt.plot(X_ax, j_cg, X_ax, j_ncg, X_ax, j_bfgs, X_ax, j_batch_gd, X_ax, j_powell)
plt.title('Optimization Algorithms')
plt.xlabel('Iterations')
plt.ylabel('Cost function')
plt.legend()
plt.show()

# Check model accuracy
J = CostFunc(res, *args)
Y_pred_train = X_0.dot(res.reshape(n+1,1))
Y_pred_test = X_test_0.dot(res.reshape(n_test+1,1))

RMSE_cg_train = sqrt(mean_squared_error(Y, Y_pred_train))
RMSE_cg_test = sqrt(mean_squared_error(Y_test, Y_pred_test))

############################ Scikit Ridge regression model ############################
# Create linear regression object
regr = linear_model.Ridge(normalize=True, fit_intercept=False, solver = 'sparse_cg')

# Train the model using the training sets
regr.fit(X, Y)

# Make predictions using the testing set
y_pred = regr.predict(X_test)
```

```
RMSE_lr = sqrt(mean_squared_error(Y_test, y_pred))
r2_score_lr = r2_score(Y_test, y_pred)

clf = Ridge()
coefs = []
errors = []

alphas = np.logspace(1, 10, 200)

# Train the model with different regularisation strengths
for a in alphas:
    clf.set_params(alpha=a)
    clf.fit(X, Y)
    coefs.append(clf.coef_.flatten())

# Display results
ax = plt.gca()
plt.plot(alphas, coefs)
ax.set_xscale('log')
plt.xlabel('lambda')
plt.ylabel('coefs')
plt.title('Ridge coefficients as a function of the regularization')
plt.axis('tight')
```

Problem 2: Solution using SVM in R :

```
#Set Working Directory
setwd("C:\\Users\\rohan\\Downloads")

#Import real estate dataset
real <- read.csv('RealEstate.csv',header=T,stringsAsFactors=F)

#EDA to choose the house locations
library(data.table)
real <- data.table(real)
eda <- real[,.(Avg_price = mean(Price),House_count = length(MLS)),by=.(Location)]
real <- data.frame(real)

#Subset the data for SVM
real <- real[which(real$Location %in% c('Arroyo Grande','Lompoc')),]
real <- real[,c('Location','Price.SQ.Ft','Size')]

#Create training and test dataset
## 75% of the sample size
smp_size <- floor(0.80 * nrow(real))

## set the seed to make your partition reproductible
set.seed(123)
train_ind <- sample(seq_len(nrow(real)), size = smp_size)

train <- real[train_ind, ]
test <- real[-train_ind, ]

train <-subset(train,!(Location=="Lompoc" & Price.SQ.Ft>210))
train <- subset(train,!(Location=="Arroyo Grande" & Price.SQ.Ft<175))

#Import library
```

```
library(ggplot2)

#Scatter Plot
ggplot(train,aes(y=Price.SQ.Ft,x=Size,color=Location)) +
  geom_point() +  labs(x = 'Size in Sq. Ft.',y='Price per Sq.Ft.')



#Apply SVM on training set
library(e1071)
class.f <- train[,'Location']
model <- svm(Location~.,data=train,kernel='linear',type='C-classification',scale='FALSE')

#Test accuracy on test set
pred.svm <- predict(model,train)
as.data.frame(pred.svm)
check <- table(pred.svm,class.f)

#Send the test data to model for prediction
final.result <- predict(model,test)
check2 <- as.data.frame(final.result)
class.f <- test[,'Location']
check3 <- table(final.result,class.f)

#Plot
#Get parameters of hyperplane
beta.1 <- sum(model$coefs*train$Size[model$index])
beta.2 <- sum(model$coefs*train$Price.SQ.Ft[model$index])
print(paste(beta.1,beta.2))
beta.0 <- -model$rho
print(beta.0)

#Equation of hyperplane => w[1,2]*x + w[1,1]*y + b = 0
#Equation of support vectors =>

#Plot the data with the separating hyperplane
ggplot(train,aes(y=Price.SQ.Ft,x=Size,color=Location)) +
  geom_point() +
  geom_abline(intercept=-beta.0/beta.2,slope=-beta.1/beta.2,colour='darkblue',size=1)+
  geom_abline(intercept=(-beta.0-1)/beta.2,slope=-beta.1/beta.2,colour='darkgreen',linetype='dashed')+
  geom_abline(intercept=(-beta.0+1)/beta.2,slope=-beta.1/beta.2,colour='darkgreen',linetype='dashed')+
  labs(x = 'Size in Sq. Ft.',y='Price per Sq.Ft.')

#Objective Function formulation
#Min w1^2+w2^2
#237.87*w[1,1]+1257*w[1,2]+b=-1
#192.40*w[1,1]+1594*w[1,2]+b=1
#188.63*w[1,1]+2947*w[1,2]+b=-1
```