COMPSCI 671D Fall 2019
Homework 2
Vinayak Gupta
vg101

1 Convexity (25 points)

Dataset: $\{X, \mathbf{y}\} = \{\mathbf{x}_i, y_i\}_{i=1}^n$, where $\mathbf{x}_i \in \mathbb{R}^p$ is a feature vector, and $y_i \in \{-1, +1\}$ is a binary label.

$\mathbf{w} = [w_1, ..., w_p]^T \in \mathbb{R}^p, b \in \mathbb{R}$ are the weight and bias.

(a) Logistic Regression (5 points)

$$L(\mathbf{w}, b) = \sum_{i=1}^{n} \log(1 + \exp(-y_i(\mathbf{w}^T\mathbf{x}_i + b)))$$

Let $\mathbf{w} = [w_1, ..., w_p, b]$ and $\mathbf{x}_i = [x_{i1}, ..., x_{ip}, 1]$,then above loss function can be written as:

$$L(\mathbf{w}) = \sum_{i=1}^{n} \log(1 + \exp(-y_i(\mathbf{w}^T\mathbf{x}_i)))$$

Now, first consider the function $f(w) = \log(1 + \exp(-w))$

Since the $\exp(-w)$ has a range from (0,∞), the domain for $\log(1 + \exp(-w))$ is (1,∞). A logarithmic function is contnuosly differentiable in this domain so we can just prove its convexity by checking whether the second derivative of the function is positive or not.

$f'(w)$ be the first derivative w.r.t. $\mathbf{w}$

$$f'(w) = \frac{-\exp(-\mathbf{w})}{1 + \exp(-\mathbf{w})}$$

Dividing both numerator and denominator by $\exp(-w)$,

$$f'(w) = \frac{-1}{\exp(\mathbf{w}) + 1}$$

$$f''(w) = \frac{(\exp(\mathbf{w}) + 1)0 - (-1)(1)(\exp(\mathbf{w}))}{(\exp(\mathbf{w}) + 1)^2}$$

$$f''(w) = \frac{\exp(\mathbf{w})}{(\exp(\mathbf{w}) + 1)^2} > 0$$

Therefore, $f(w)$ is a convex function

Now, the composition of a convex and affine function is convex which can be shown as below:

Suppose $f(x) = g(h(x))$ where g is a convex function but f is an affine function, then:

Let $\theta \in [0, 1]$ and x_1 , x_2 be in the domain of f and h:

$$f(\theta x_1 + (1 - \theta)x_2) = g(\theta h(x_1) + (1 - \theta)h(x_2))$$

$$\leq \theta g(h(x_1)) + (1 - \theta)g(h(x_2)) = \theta f(x_1) + (1 - \theta)f(x_2)$$

Therefore f(x) is convex.

We will use this property to say that $f(w) = \log(1+\exp(-y_i(\mathbf{w}^T\mathbf{x}_i)))$ is a convex function since its a composition of a convex and an affine function where:

$$f(w) = g(h(w))$$

where $h(w) = -y_i(\mathbf{w}^T\mathbf{x}_i)$ (an affine function) and $g(w) = \log(1+\exp(-\mathbf{w}))$ (a convex function)

Hence, $f(w) = \log(1 + \exp(-y_i(\mathbf{w}^T\mathbf{x}_i)))$ is a convex function and since sum of convex functions is convex, $L(\mathbf{w}, b) = \sum_{i=1}^{n} \log(1 + \exp(-y_i(\mathbf{w}^T\mathbf{x}_i + b)))$ is also a **CONVEX** function.

(b) SVM (5 points)

$$L(\mathbf{w}, b; C) = \frac{1}{2}||\mathbf{w}||_2^2 + C\sum_{i=1}^{n} \max(0, 1 - y_i(\mathbf{w}^T\mathbf{x}_i + b)), \quad C \geq 0$$

where C is the penalty parameter for errors.

The convexity of this loss function can be proved by looking at the component functions. First, look at $\frac{1}{2}||\mathbf{w}||_2^2$

Let $f(w) = \frac{1}{2}||\mathbf{w}||_2^2 = \frac{1}{2}\sum_{i=1}^{n} {w_i}^2$

$f'(w) = w$ where $f'(w)$ is first derivative w.r.t w

Now computing the Hessian Matrix, we see that the diagonal elements have the value 1 whereas the non diagnoal elements are 0.

$$H(w) = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

The Hessian is positive semi-definite, therefore $\frac{1}{2}||\mathbf{w}||_2^2$ is convex. For derivative w.r.t b it would be 0 as there is no term containing b, hence convex. Now look at $C\sum_{i=1}^{n} \max(0, 1 - y_i(\mathbf{w}^T\mathbf{x}_i + b))$

0 is an affine function which is a convex function and $1 - y_i(\mathbf{w}^T\mathbf{x}_i + b)$ is also an affine function (w.r.t w and b) which is convex. Moreover, **maximum** of 2 convex functions is a **convex** function. Therefore, $C\sum_{i=1}^{n} \max(0, 1 - y_i(\mathbf{w}^T\mathbf{x}_i + b))$ is also a convex function since its just the sum of functions that are convex multiplied by a constant.

Since the loss function $L(\mathbf{w}, b; C)$ is just the sum of two convex function, it is also a **CONVEX** function

(c) LASSO (5 points) Dataset: $\{X, \mathbf{z}\} = \{\mathbf{x}_i, z_i\}_{i=1}^n$, where $\mathbf{x}_i \in \mathbb{R}^p$ is a feature vector, and $z_i \in \mathbb{R}$ is a real-valued target.

$$L(\mathbf{w}, b; \lambda) = ||\mathbf{z} - (X\mathbf{w} + b\mathbf{1}_n)||_2^2 + \lambda||\mathbf{w}||_1, \quad \lambda \geq 0$$

where λ is the tuning parameter for L_1 penalty.

The convexity of this loss function can be proved by looking at the component functions. First, look at $\lambda||\mathbf{w}||_1$

Let $f(w) = \lambda||\mathbf{w}||_1$, this is an affine function and therefore is convex. Also $f''(w) = 0$, therefore it is positive semi-definite and hence a convex function.. Similarly, it is covex w.r.t b as well.

Now look at the $||\mathbf{z} - (X\mathbf{w} + b\mathbf{1}_n)||_2^2$ function:

This is a composition of a convex and an affine function, in 1(b) we proved that $\frac{1}{2}||\mathbf{w}||_2^2$ is convex, therefore $||\mathbf{w}||_2^2$ is also a convex function. Let:

$$f(w) = g(h(w))$$

where $h(w) = \mathbf{z} - (X\mathbf{w} + b\mathbf{1}_n)$ (an affine function w.r.t both w and b) and $||\mathbf{w}||_2^2$ (a convex function)

Hence, $f(w) = ||\mathbf{z} - (X\mathbf{w} + b\mathbf{1}_n)||_2^2$ is also a convex function.

Since sum of 2 convex functions is a convex function, therefore $L(\mathbf{w}, b; \lambda) = ||\mathbf{z} - (X\mathbf{w} + b\mathbf{1}_n)||_2^2 + \lambda||\mathbf{w}||_1, \quad \lambda \geq 0$ is also a **CONVEX** function

(d) Linear Regression with SCAD penalty (10 points)

$$L(\mathbf{w}, b; \lambda) = ||\mathbf{z} - (X\mathbf{w} + b\mathbf{1}_n)||_2^2 + \sum_{j=1}^{p} P(w_j; \lambda, \gamma), \quad \lambda > 0, \gamma > 2$$

where $P(x; \lambda, \gamma)$ is the penalty function

$$P(x; \lambda, \gamma) = \begin{cases} \lambda|x| & \text{if } |x| \leq \lambda \\ \frac{2\gamma\lambda|x| - x^2 - \lambda^2}{2(\gamma-1)} & \text{if } \lambda < |x| < \gamma\lambda \\ \frac{\lambda^2(\gamma+1)}{2} & \text{if } |x| \geq \gamma\lambda \end{cases}$$

This function is **not necessarily convex** as the penalty function is not convex between $|x| \leq \lambda$ and $\lambda < |x| < \gamma\lambda$. This can be shown using a counter example which will satisfy all the conditions for the penalty function:

Suppose, $\lambda = 2, \gamma = 3$

The above values of λ and γ satisfy the penalty function constraints, so we can safely use them. Now consider two points:

$x_1 = 1, x_2 = 3$

Then $P(x_1; 2, 3) = P(1, 2, 3) = \lambda|x_1| = 2$

$P(x_2; 2, 3) = P(3, 2, 3) = \frac{2\gamma\lambda|x| - x^2 - \lambda^2}{2(\gamma-1)} = \frac{2*3*3-9-4}{2*(3-1)} = 5/4$

For penalty function to be convex: $P(\theta x_1 + (1-\theta)x_2; 2, 3) \leq \theta P(x_1; 2, 3) + (1-\theta)P(x_2; 2, 3)$

Suppose $\theta = 0.5$, then $\theta x_1 + (1-\theta)x_2 = 0.5 * 1 + 0.5 * 3 = 2$

Now, P(2;2,3) = $\lambda|x_1|$ = 2*2 =4

For P to be convex, P(2;2,3) $\leq$ 0.5 * P(1;2,3) +(1-0.5)*P(3;2,3)

However, RHS equals to 0.5* 2 + 0.5* 5/4 = 13/8

But 13/8 is not $\geq$ 4, therefore the penalty function is not convex.

Actually, the penalty function is not even convex in the region $\lambda < |x| < \gamma\lambda$ as we can see the second derivative is not ≥ 0:

$$P(x; \lambda, \gamma) = \frac{2\gamma\lambda|x| - x^2 - \lambda^2}{2(\gamma-1)}; \lambda < |x| < \gamma\lambda$$

The second derivative of the above function is $\frac{-2}{2(\gamma-1)}$ which is always negative.

Also, if X in the loss function is 0 and w_j's lies in the range $\lambda < |x| < \gamma\lambda$, then the loss function reduces to:

$$L(\mathbf{w}, b; \lambda) = ||\mathbf{z} - b\mathbf{1}_n)||_2^2 + \sum_{j=1}^{p} \frac{2\gamma\lambda|w_j| - {w_j}^2 - \lambda^2}{2(\gamma - 1)}$$

Since, the second function is not convex as proved above and $||\mathbf{z} - b\mathbf{1}_n)||_2^2$ is just a constant function w.r.t w, then the entire loss function is not convex.

Hence, again we can show that the loss function is **NOT NECESSARILY CONVEX** since all the component functions are not convex

2 Support Vector Machine (35 points)

(a) Least-squares Support Vector Machine (10 points)

Given a dataset $\{x_i, y_i\}_{i=1}^n, x_i \in \mathbb{R}^p, y_i \in \{-1, +1\}$, the least-squares SVM learns the classifier by the following optimization problem:

$$\min L(w, b, e) = \frac{1}{2} w^T w + \frac{1}{2} C \sum_{i=1}^{n} e_i^2, \quad s.t. \quad y_i[w^T \phi(x_i) + b] = 1 - e_i.$$

where $\phi(x)$ is the feature map from the original space to the higher- or infinite-dimensional space.

Multiply $y_i[w^T \phi(x_i) + b] = 1 - e_i$ by y_i on both sides, we get

$[w^T \phi(x_i) + b] = y_i - y_i e_i$ since ${y_i}^2$ is 1

However, since in the objective function, we have e_i^2 the term $(y_i e_i)^2 = e_i^2$ and therefore we can safely say in the constraint that $y_i e_i = e_i$

The constraint now becomes $[w^T \phi(x_i) + b] = y_i - e_i$

Constructing the Lagrangian,

$$L_1(w, b, e) = \frac{1}{2} w^T w + \frac{1}{2} C \sum_{i=1}^{n} e_i^2 - \sum_{i=1}^{n} \alpha_i [w^T \phi(x_i) + b] - y_i + e_i]$$

, where $\alpha_i \in \mathbb{R}$ (Dual Feasability)

The KKT conditions can be written as:

$\frac{\partial L_1}{\partial w} = 0$ gives $w = \sum_{i=1}^n \alpha_i \phi(x_i)$ Lagrangian Stationary

$\frac{\partial L_1}{\partial b} = 0$ gives $\sum_{i=1}^n \alpha_i = 0$ Lagrangian Stationary

$\frac{\partial L_1}{\partial e_i} = 0$ gives $C e_i = \alpha_i$ Lagrangian Stationary

and finally $[w^T \phi(x_i) + b] = y_i - e_i$ Primal Feasability

Therefore, $e_i = \alpha_i / C$, replacing w and e in the last equation gives:

$y_j = \sum_{i=1}^n \alpha_i \phi(x_i) \phi(x_j) + b + \alpha_j / C$

$y_j = \sum_{i=1}^n \alpha_i \Omega_{ij} + b + \alpha_j / C$, since $\Omega_{ij} = \phi(x_i)^T \phi(x_j) = K(x_i, x_j)$

Thus, the above equation can be written as:

$$\begin{bmatrix} 0 & 1_N^T \\ 1_N & \Omega + C^{-1} I_N \end{bmatrix} \begin{bmatrix} b \\ \alpha \end{bmatrix} = \begin{bmatrix} 0 \\ y \end{bmatrix},$$

where $y = [y_1, \ldots, y_N]^T$, $1_N = [1, \ldots, 1]^T$, $\alpha = [\alpha_1, \ldots, \alpha_N]^T$ and 1_N is the N x N Identity Matrix. The classifier formula can be written as:

$f(x_j) = w^T \phi(x_j) + b = \sum_{i=1}^{n} \alpha_i \Omega_{ij} + b$, where α_i and b are calculated from the above set of linear equations and in Python, they can be solved using linalg.solve in the numpy package.

(b) Implement SVM and LS-SVM (15 points)

```
import matplotlib.pyplot as plt
import numpy as np
import scipy.io as sio
import pandas as pd
import sklearn
import math
import csv
from sklearn.datasets import fetch_california_housing
from sklearn import metrics
from sklearn.svm import SVC

# Calculates kernel value
def rbfker(x1,x2,gamma):
    return math.exp(-1*gamma*((np.linalg.norm(x1-x2))**2))

# Populates Gram matrix + 1/C* Identity matrix
def Kmatrix(df,gamma,length,C):
    kmatrix= [[rbfker(X.iloc[k,:].to_numpy(),X.iloc[j,:].to_numpy(),gamma) + 1/C
    if(k==j)else rbfker(X.iloc[k,:].to_numpy(),X.iloc[j,:].to_numpy(),gamma)  for j
     in range(length)] for k in range(length)]
    return kmatrix

#Populates the coefficient matrix for the equations
def Inmatrix(kmatrix,length):
    inmatrix=np.zeros((length+1,length+1))
    for i in range(length+1):
        for j in range(length+1):
            if(i==0 and j==0): inmatrix[i][j]=0
            elif(i==0): inmatrix[i][j]=1
            elif(j==0): inmatrix[i][j]=1
            else: inmatrix[i][j]= kmatrix[i-1][j-1]
    return inmatrix

# Calculates the Predicted value from classifier
def calculate_fx(len_test,length,solution,X,X_test,gamma):
    f_x=np.zeros((len_test))
    for i in range(len_test):
        s=0
        for j in range(length):
            s+= solution[j+1]*rbfker(X.iloc[j,:].to_numpy(),X_test.iloc[i,:].
    to_numpy(),gamma)
        f_x[i]=s+ solution[0]
    return f_x

#Classifies a point using the sign of classifier
def classify_fx(f_x):
    y_pred= [1 if i>=0 else -1 for i in f_x]
    return y_pred

#Populates the dataframe for boxplot using different SVM kernels as well as LS-SVM
```

```
def populatebox(A,boxval,gamma,C,X,y,length,X_test,y_test,len_test):
    kmatrix=Kmatrix(X,gamma,length,C)
    inmatrix=np.zeros((length+1,length+1))
    inmatrix = Inmatrix(kmatrix,length)
    rhs=[0]
    rhs.extend(y.to_numpy())
    solution = np.linalg.solve(inmatrix, rhs)

    f_x= calculate_fx(len_test,length,solution,X,X_test,gamma)
    y_pred=classify_fx(f_x)
    boxval['LS-SVM-Test'].iloc[A]= 1-metrics.accuracy_score(y_test, y_pred)

    f_x_train= calculate_fx(length,length,solution,X,X,gamma)
    y_pred_train=classify_fx(f_x_train)
    boxval['LS-SVM-Train'].iloc[A]= 1-metrics.accuracy_score(y, y_pred_train)

    classifier= SVC(kernel='rbf',random_state=0,C=C,gamma=gamma)
    classifier.fit(X,y)
    y_pred_SVM=classifier.predict(X_test)
    boxval['Rbf-SVM-Test'].iloc[A]= 1-metrics.accuracy_score(y_test, y_pred_SVM)

    y_pred_train_SVM=classifier.predict(X)
    boxval['Rbf-SVM-Train'].iloc[A]= 1-metrics.accuracy_score(y, y_pred_train_SVM)

    classifier1= SVC(kernel='linear',random_state=0,C=C,gamma=gamma)
    classifier1.fit(X,y)
    y_pred_SVM1=classifier1.predict(X_test)
    boxval['Linear-SVM-Test'].iloc[A]= 1-metrics.accuracy_score(y_test,
    y_pred_SVM1)

    y_pred_train_SVM1=classifier1.predict(X)
    boxval['Linear-SVM-Train'].iloc[A]= 1-metrics.accuracy_score(y,
    y_pred_train_SVM1)

    classifier2= SVC(kernel='poly',random_state=0,C=C,gamma=gamma)
    classifier2.fit(X,y)
    y_pred_SVM2=classifier2.predict(X_test)
    boxval['Polynomial-SVM-Test'].iloc[A]= 1-metrics.accuracy_score(y_test,
    y_pred_SVM2)

    y_pred_train_SVM2=classifier2.predict(X)
    boxval['Polynomial-SVM-Train'].iloc[A]= 1-metrics.accuracy_score(y,
    y_pred_train_SVM2)

    classifier3= SVC(kernel='sigmoid',random_state=0,C=C,gamma=gamma)
    classifier3.fit(X,y)
    y_pred_SVM3=classifier3.predict(X_test)
    boxval['Sigmoid-SVM-Test'].iloc[A]= 1-metrics.accuracy_score(y_test,
    y_pred_SVM3)

    y_pred_train_SVM3=classifier3.predict(X)
    boxval['Sigmoid-SVM-Train'].iloc[A]= 1-metrics.accuracy_score(y,
    y_pred_train_SVM3)

#Read Training data
df = pd.read_csv("train.csv",header=None)
df.columns = df.iloc[0]
df=df.drop(df.index[0])
df.x1 = pd.to_numeric(df.x1, errors='coerce')
```

```
df.x2 = pd.to_numeric(df.x2, errors='coerce')
df.y = pd.to_numeric(df.y, errors='coerce')
length= df.shape[0]
X=df.iloc[:,:-1]
y=df.iloc[:,-1]

#Read Test Data
df_test = pd.read_csv("test.csv",header=None)
df_test.columns = df_test.iloc[0]
df_test=df_test.drop(df_test.index[0])
df_test.x1 = pd.to_numeric(df_test.x1, errors='coerce')
df_test.x2 = pd.to_numeric(df_test.x2, errors='coerce')
df_test.y = pd.to_numeric(df_test.y, errors='coerce')
X_test=df_test.iloc[:,:-1]
y_test=df_test.iloc[:,-1]
len_test=df_test.shape[0]

#fix C and vary gamma
gamma = 100
C=1
N_rows=7
A=0
boxval1 = pd.DataFrame(np.zeros((N_rows, 12)),columns =['Gamma','C','LS-SVM-Train'
    , 'LS-SVM-Test', 'Rbf-SVM-Train','Rbf-SVM-Test','Linear-SVM-Train','Linear-SVM-
    Test','Polynomial-SVM-Train','Polynomial-SVM-Test','Sigmoid-SVM-Train','Sigmoid
    -SVM-Test'])
for j in range(7):
    gamma_j=gamma/(10**j)
    boxval1['C'].iloc[A]= C
    boxval1['Gamma'].iloc[A]= gamma_j
    populatebox(A,boxval1,gamma_j,C,X,y,length,X_test,y_test,len_test)
    A+=1
print(boxval1)

boxval1.boxplot(column=['LS-SVM-Train', 'LS-SVM-Test', 'Rbf-SVM-Train','Rbf-SVM-
    Test','Linear-SVM-Train','Linear-SVM-Test','Polynomial-SVM-Train','Polynomial-
    SVM-Test','Sigmoid-SVM-Train','Sigmoid-SVM-Test'])
plt.xticks(rotation=90)
plt.show()

gamma = 1
C=10000
N_rows=9
A=0
pd.set_option('precision', 5)
boxval = pd.DataFrame(np.zeros((N_rows, 12)),columns =['Gamma','C','LS-SVM-Train',
     'LS-SVM-Test', 'Rbf-SVM-Train','Rbf-SVM-Test','Linear-SVM-Train','Linear-SVM-
    Test','Polynomial-SVM-Train','Polynomial-SVM-Test','Sigmoid-SVM-Train','Sigmoid
    -SVM-Test'])
for i in range(9):
    C_i=C/(10**i)
    boxval['C'].iloc[A]= C_i
    boxval['Gamma'].iloc[A]= gamma
    populatebox(A,boxval,gamma,C_i,X,y,length,X_test,y_test,len_test)
    A+=1

print(boxval)
boxval.boxplot(column=['LS-SVM-Train', 'LS-SVM-Test', 'Rbf-SVM-Train','Rbf-SVM-
    Test','Linear-SVM-Train','Linear-SVM-Test','Polynomial-SVM-Train','Polynomial-
```

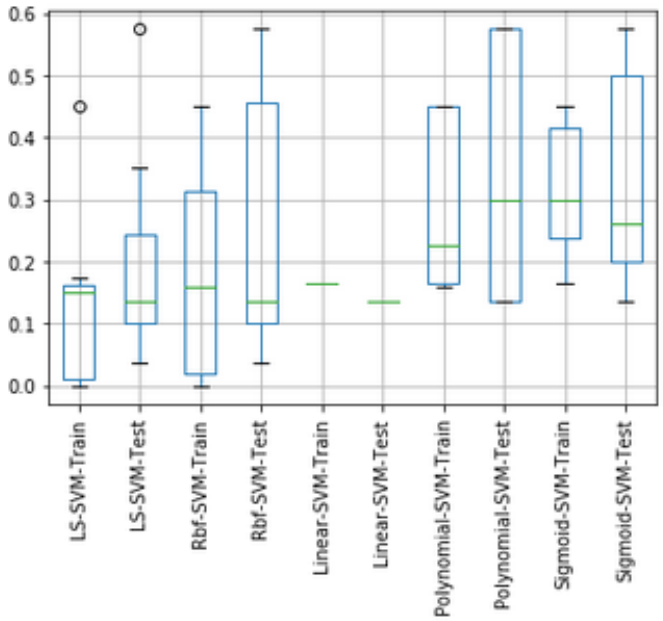
```
    SVM-Test','Sigmoid-SVM-Train','Sigmoid-SVM-Test'])
plt.xticks(rotation=90)
plt.show()
```

The above code implements LS-SVM and construct a classifier with **rbf** kernel. The code also compares against different values of C (10^4 till 10^{-4}) keeping γ constant, different values of γ (10^2 till 10^{-4}) keeping C constant and different SVM kernels (linear, polynomial, rbf, sigmoid).

The training and test error by keeping the C constant and varying the γ has been shown in the figure below. It can be seen that the training error decreases as γ increases because we start overfitting the data. While on the test set, the error decreases as γ increases but then it starts increasing again because we overfit the data in our training (moving from underfitting to overfitting).

	Gamma	C	LS-SVM-Train	LS-SVM-Test	Rbf-SVM-Train	Rbf-SVM-Test	Linear-SVM-Train	Linear-SVM-Test	Polynomial-SVM-Train	Polynomial-SVM-Test	Sigmoid-SVM-Train	Sigmoid-SVM-Test
0	100.0000	1.0	0.000	0.3500	0.000000	0.3375	0.166667	0.1375	0.166667	0.1375	0.300000	0.2625
1	10.0000	1.0	0.000	0.0625	0.008333	0.0625	0.166667	0.1375	0.166667	0.1375	0.300000	0.2500
2	1.0000	1.0	0.025	0.0375	0.033333	0.0375	0.166667	0.1375	0.158333	0.1375	0.383333	0.4250
3	0.1000	1.0	0.150	0.1375	0.158333	0.1375	0.166667	0.1375	0.225000	0.3000	0.175000	0.1375
4	0.0100	1.0	0.175	0.1375	0.175000	0.1375	0.166667	0.1375	0.450000	0.5750	0.166667	0.1500
5	0.0010	1.0	0.150	0.1375	0.450000	0.5750	0.166667	0.1375	0.450000	0.5750	0.450000	0.5750
6	0.0001	1.0	0.450	0.5750	0.450000	0.5750	0.166667	0.1375	0.450000	0.5750	0.450000	0.5750

The box plots showing the classification training and test errors for each algorithm while varying the γ and fixing the C are also shown below:

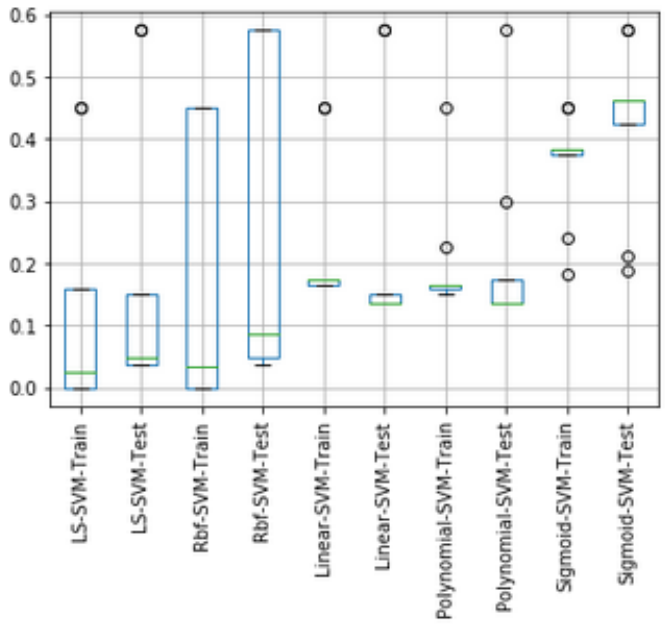


For Linear SVM, γ does not have any effect for linear SVM which makes sense as it does not depend on γ.

The training and test error by keeping the γ constant and varying the C has been shown in the figure below. In general, we can see that for higher value of C, the training error is lower because we are essentially heavily penalizing the algorithm for making errors (Overfitting occurs). When C is very small (0.0001), the algorithm just maximizes the margin as much as possible rather than trying to classify maximum number of points correctly, therefore the training error is higher (Underfitting occurs). In case of test errors, again the error decreases as C increases but after a point the error starts increasing because we start overfitting the data.

	Gamma	C	LS-SVM-Train	LS-SVM-Test	Rbf-SVM-Train	Rbf-SVM-Test	Linear-SVM-Train	Linear-SVM-Test	Polynomial-SVM-Train	Polynomial-SVM-Test	Sigmoid-SVM-Train	Sigmoid-SVM-Test
0	1.0	10000.0000	0.00000	0.0625	0.00000	0.0875	0.17500	0.1375	0.16667	0.1375	0.38333	0.4625
1	1.0	1000.0000	0.00000	0.0500	0.00000	0.0875	0.17500	0.1375	0.16667	0.1375	0.38333	0.4625
2	1.0	100.0000	0.00000	0.0500	0.00000	0.0750	0.17500	0.1375	0.16667	0.1375	0.38333	0.4625
3	1.0	10.0000	0.00833	0.0375	0.00833	0.0375	0.17500	0.1375	0.16667	0.1375	0.37500	0.4625
4	1.0	1.0000	0.02500	0.0375	0.03333	0.0375	0.16667	0.1375	0.15833	0.1375	0.38333	0.4250
5	1.0	0.1000	0.05833	0.0375	0.07500	0.0500	0.16667	0.1375	0.15000	0.1375	0.24167	0.2125
6	1.0	0.0100	0.15833	0.1500	0.45000	0.5750	0.16667	0.1500	0.15000	0.1750	0.18333	0.1875
7	1.0	0.0010	0.45000	0.5750	0.45000	0.5750	0.45000	0.5750	0.22500	0.3000	0.45000	0.5750
8	1.0	0.0001	0.45000	0.5750	0.45000	0.5750	0.45000	0.5750	0.45000	0.5750	0.45000	0.5750

The box plots showing the classification training and test errors for each algorithm while varying the C and fixing the γ are also shown below:



The LS-SVM however has the best performance among all as its classification error for test and train are both less as compared to other algorithms as well as there is lower variance in those errors with different parameters as compared to rbf SVM.

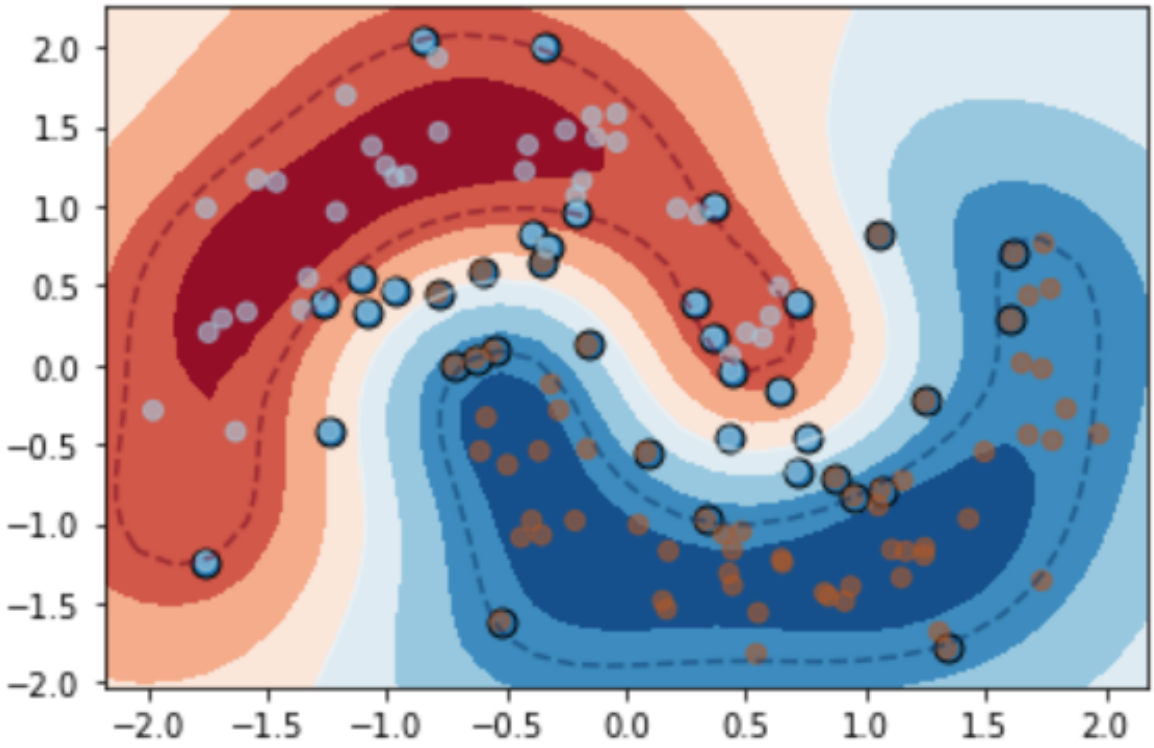
(c) Support Vectors (10 points)

(i)

```
import matplotlib.pyplot as plt
import numpy as np
import scipy.io as sio
import pandas as pd
import sklearn
import math
import csv
from sklearn.datasets import fetch_california_housing
from sklearn import metrics
from sklearn.svm import SVC
from scipy.interpolate import griddata

def visualizestuff(X,y,gamma,C,ax):
    ax.scatter(X.iloc[:, 0], X.iloc[:, 1], c=y)
    limx = ax.set_xlim()
    limy = ax.set_ylim()
    x1 = np.linspace(limx[0], limy[1],30)
    y1 = np.linspace(limy[0], limy[1],30)
    Ygrid, Xgrid = np.meshgrid(y1, x1)
    XY = np.transpose(np.vstack([Xgrid.ravel(), Ygrid.ravel()]))
    classifier= SVC(kernel='rbf',random_state=0,C=C,gamma=gamma)
    classifier.fit(X,y)
    Z = classifier.decision_function(XY).reshape(Xgrid.shape)
    ax.contourf(Xgrid,Ygrid,Z,cmap='RdBu')
    ax.contour(Xgrid, Ygrid, Z,levels=[-1, 0, 1],linestyles=['--', '-', '--'],
    alpha=0.5,cmap='RdBu')
    ax.scatter(classifier.support_vectors_[:, 0], classifier.support_vectors_[:,
    1], s=80,edgecolors='k',alpha=1)
    ax.scatter(X.iloc[:, 0], X.iloc[:, 1], c=y, cmap=plt.cm.Paired,alpha=0.6)

df = pd.read_csv("train.csv",header=None)
df.columns = df.iloc[0]
df=df.drop(df.index[0])
df.x1 = pd.to_numeric(df.x1, errors='coerce')
df.x2 = pd.to_numeric(df.x2, errors='coerce')
df.y = pd.to_numeric(df.y, errors='coerce')
X=df.iloc[:,:-1]
y=df.iloc[:,-1]

fig = plt.figure()
ax= fig.add_subplot(111)
visualizestuff(X,y,1,1,ax)
plt.show()

gamma = 1
C=10000
fig = plt.figure()
fig.subplots_adjust(hspace=0.5)
fig.set_figheight(15)
fig.set_figwidth(15)
plt.suptitle(gamma)
m=0
for i in range(9):
    C_i=C/(10**i)
    ax= fig.add_subplot(331 + m)
    visualizestuff(X,y,gamma,C_i,ax)
```

```
    m=m+1
    ax.title.set_text(C_i)
plt.show()

gamma = 10000
C=1
fig = plt.figure()
fig.subplots_adjust(hspace=0.5)
fig.set_figheight(15)
fig.set_figwidth(15)
plt.suptitle(C)
m=0
for i in range(9):
    gamma_i=gamma/(10**i)
    ax= fig.add_subplot(331 + m)
    visualizestuff(X,y,gamma_i,C,ax)
    m=m+1
    ax.title.set_text(gamma_i)
plt.show()
```

The above code results in the following figure:



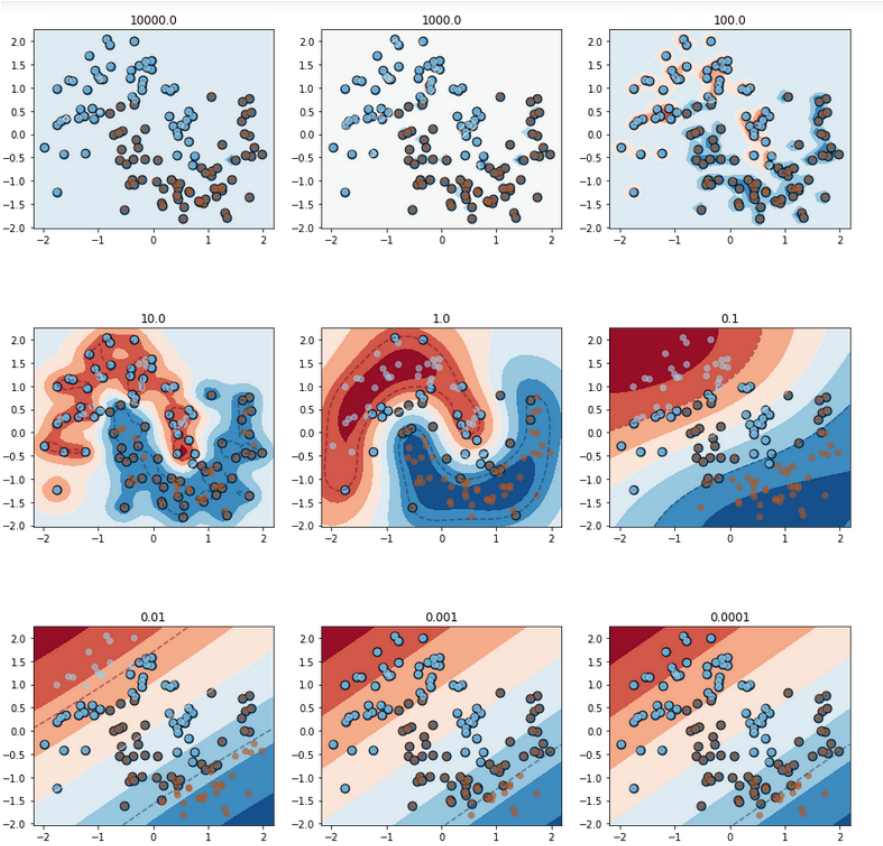
(ii) Write an expression for a region that contains your support vectors with **positive labels**. Specifically, simplify $\{x : f(x) \leq \theta\}$ for a value of θ that contains all of your support vectors.

$$\theta = max(\mathbf{1}_{[y_j==1]}(\sum_{i=1}^{n} \alpha_i y_i K(x_i, x_j) + b)) \quad \forall i, j \in \mathcal{I}$$

where n represents all the support vectors as α_i would be 0 for non support vectors.

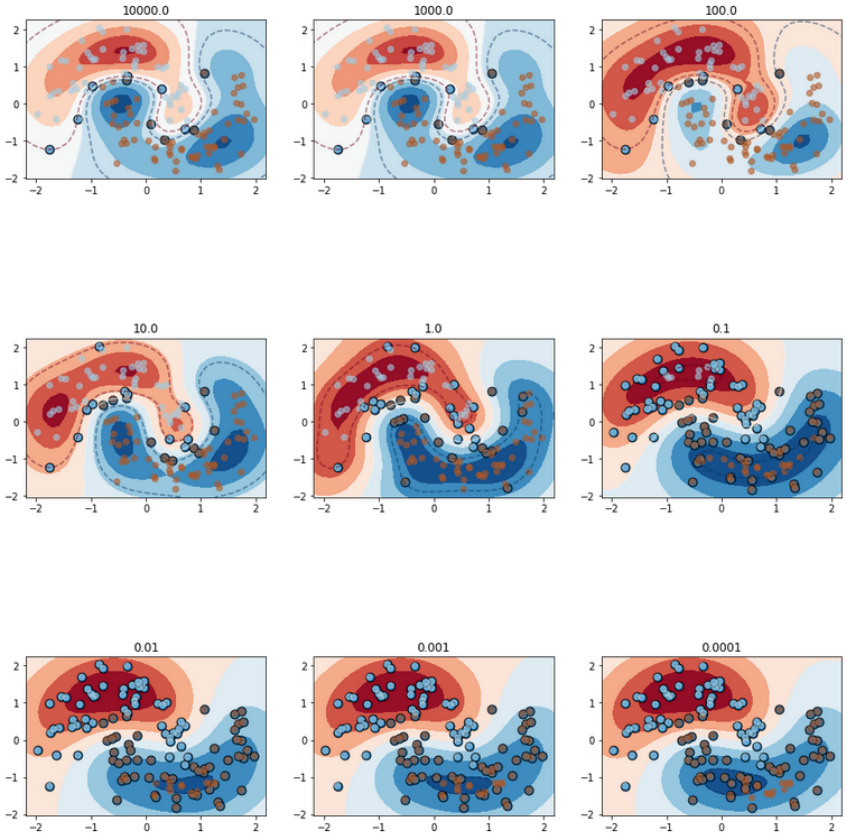
The above expression calculates the maximum margin for a positive labeled point which is what we want. The indicator function is used to filter only the positive points and $(\sum_{i=1}^{n} \alpha_i y_i K(x_i, x_j)+b)$ is used to calculate the margin for that positive point. The one with the maximum positive margin among such positive support vectors gives the value of θ that we need to calculate.

(iii)



The code in (i) also plots for varying γ value keeping C constant at 1 as shown above. The non-linearity of boundary **increases** as γ **increases** as it starts overfitting the data.

The code in (i) also plots for varying C value keeping γ constant at 1 as shown below. The Euclidean distance between 0 and 1 **decreases** as C **increases** because we start overfitting the data



3 AdaBoost with Stumps Fits an Additive Model (30 points + 5 bonus)

$\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, $\mathbf{x}_i \in \mathbb{R}^p$, $y_i \in \mathbb{R}$.

$$\hat{y}_i = f(\mathbf{x}_i) = b + \sum_{q=1}^{p} f_q(\mathbf{x}_i). \tag{1}$$

$f_q(\mathbf{x}_i)$ is a piece-wise constant function. The sum of component functions can then be written as a weighted summation of **decision stumps**: $h_j(x) = \mathbf{1}_{[x_{\cdot,q} > I_j]}$, where $\mathbf{1}_{[x_{\cdot,q} > I_j]}$ means that the value of the function is 1 if the qth component of x is greater than threshold I_j. We can add up weighted sums of these functions to produce the final combined classifier,

$$\hat{y}_i = f(x_i) = b + \sum_{j=1}^{m} \lambda_j h_j(x_i) = \mathbf{h}(\mathbf{x}_i)^T \lambda, \tag{2}$$

where $\mathbf{h}(\mathbf{x}_i) = [1, h_1(\mathbf{x}_i), \ldots, h_m(\mathbf{x}_i)]$ and $\lambda = [b, \lambda_1, \ldots, \lambda_m]$, $\lambda \in \mathbb{R}^m$ is a vector of weights.

We then define the L2 loss:

$$L(\lambda) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \mathbf{h}(\mathbf{x}_i)^T \lambda)^2. \tag{3}$$

(a) Coordinate descent with L2 loss (10 points)

(i) **To derive the optimal direction to move in for AdaBoost with L2 loss:**

To do coordinate descent, suppose we are at λ_t and we want the steepest direction:

$j_t \in argmax_j[-\frac{dR^{train}(\lambda_t + \alpha e_j)}{d\alpha}]$ and set $\alpha = 0$

where R^{train} is our loss function and $e_j = [0, \ldots, 1, \ldots, 0]$, 1 at j^{th} index

$$j_t \in argmax_j[-\frac{d}{d\alpha}(\frac{\sum_{i=1}^n (y_i - \mathbf{h}(\mathbf{x}_i)^T(\lambda_\mathbf{t} + \alpha e_j))^2}{n})]$$

$$j_t \in argmax_j[2\frac{\sum_{i=1}^n (y_i - \mathbf{h}(\mathbf{x}_i)^T(\lambda_\mathbf{t} + \alpha e_j))(\mathbf{h}(\mathbf{x}_i)^T e_j)}{n}]$$

2 gets absorbed in the argmax, set $\alpha = 0$ now, the equation becomes:

$$j_t \in argmax_j[\frac{\sum_{i=1}^n (y_i - \mathbf{h}(\mathbf{x}_i)^T \lambda_\mathbf{t})(\mathbf{h_j}(\mathbf{x}_i))}{n}]$$

n can also be absorbed in the argmax, therefore the equation becomes:

$$j_t \in argmax_j[\sum_{i=1}^{n}(y_i - \mathbf{h}(\mathbf{x}_i)^T\lambda_\mathbf{t})(\mathbf{h_j}(\mathbf{x}_i))]$$

However, here we have considered that α is in the positive direction of coordinates, and since we won't reweight the misclassified points, we need to look at the direction where the α is negative. Performing the same coordinate descent operation with alpha in the negative direction:

$j_t \in argmax_j[-\frac{dR^{train}(\lambda_t - \alpha e_j)}{d\alpha}]$ and set $\alpha = 0$

we finally get

$$j_t \in argmax_j[-\sum_{i=1}^{n}(y_i - \mathbf{h}(\mathbf{x}_i)^T\lambda_\mathbf{t})(\mathbf{h_j}(\mathbf{x}_i))]$$

Therefore, both equations together be written as:

$$j_t \in argmax_j[|\sum_{i=1}^{n}(y_i - \mathbf{h}(\mathbf{x}_i)^T\lambda_\mathbf{t})(\mathbf{h_j}(\mathbf{x}_i))|]$$

where $||$ represents the absolute value

(ii) **Derive the amount to move by in that direction.**

Now we perform linesearch along the direction j_t selected from above:

$$\frac{dR^{train}(\lambda_t + \alpha e_{jt})}{d\alpha} = 0$$

at α_t which is the amount to move by.

$$\frac{d}{d\alpha}(\frac{\sum_{i=1}^{n}(y_i - \mathbf{h}(\mathbf{x}_i)^T(\lambda + \alpha e_{jt}))^2}{n}) = 0$$

$$2\frac{\sum_{i=1}^{n}(y_i - \mathbf{h}(\mathbf{x}_i)^T(\lambda + \alpha_t e_{jt}))(\mathbf{h}(\mathbf{x}_i)^T e_{jt})}{n} = 0$$

$$\sum_{i=1}^{n}(y_i - \mathbf{h}(\mathbf{x}_i)^T(\lambda + \alpha_t e_{jt}))(\mathbf{h_j}(\mathbf{x}_i)) = 0$$

$$\sum_{i=1}^{n}(y_i - \mathbf{h}(\mathbf{x}_i)^T\lambda - \mathbf{h}(\mathbf{x}_i)^T\alpha_t e_{jt})(\mathbf{h_j}(\mathbf{x}_i)) = 0$$

$$\sum_{i=1}^{n}(y_i - \mathbf{h}(\mathbf{x}_i)^T\lambda - \mathbf{h_j}(\mathbf{x}_i)\alpha_t)(\mathbf{h_j}(\mathbf{x}_i)) = 0$$

$$\sum_{i=1}^{n}(y_i - \mathbf{h}(\mathbf{x}_i)^T\lambda)(\mathbf{h_j}(\mathbf{x}_i)) = \sum_{i=1}^{n}(\mathbf{h_j}(\mathbf{x}_i)\alpha_t)(\mathbf{h_j}(\mathbf{x}_i))$$

$$\alpha_t \sum_{i=1}^{n}(\mathbf{h_j}(\mathbf{x}_i))^2 = \sum_{i=1}^{n}(y_i - \mathbf{h}(\mathbf{x}_i)^T\lambda)(\mathbf{h_j}(\mathbf{x}_i))$$

Since, $h_j(x) = \mathbf{1}_{[x_{\cdot,q} > I_j]}$, let n' denote the number of points where $h_j(x_i) = 1$

Then the above equation becomes:

$$\alpha_t = \frac{\sum_{i=1}^{n}(y_i - \mathbf{h}(\mathbf{x}_i)^T\lambda)(\mathbf{h_j}(\mathbf{x}_i))}{n'}$$

(b) Fit Californian house prices with AdaBoost (15 points)

(i) For each feature, features = ['MedInc','AveOccup',...], plot 'HousePrice' versus each feature q. (These plots should already tell us something intuitive about the model we will produce.)

(ii) Apply AdaBoost. Sample code for downloading the dataset and for creating the decision stumps is provided.

(iii) For each feature, plot the feature value on the horizontal axis and $f_q(x)$ on the vertical axis. Repeat this for each feature q, where features = ['MedInc','AveOccup',...]. Make sure to compute $f_q(x)$ as a weighted sum of all of the stumps on that feature. Describe the trends you see.

(iv) Compute variable importance using a technique shown in class. Discuss the results from variable importance in comparison with the plots in (i) and (iii).

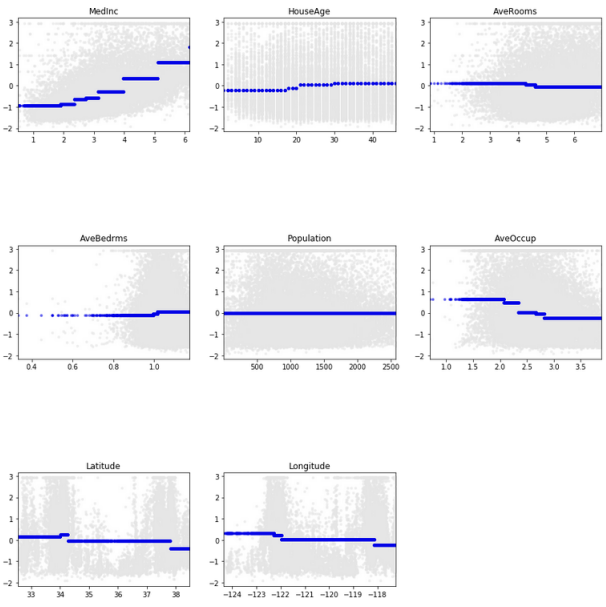
(ii)

```
import matplotlib.pyplot as plt
import numpy as np
import scipy.io as sio
import pandas as pd
import sklearn
import copy
from sklearn.datasets import fetch_california_housing

plt.rcParams['font.size'] = 14

# Download data

tmp = sklearn.datasets.fetch_california_housing()
num_samples   = tmp['data'].shape[0]
feature_names = tmp['feature_names']
y = tmp['target']
X = tmp['data']

data = {}
for n, feature in enumerate(feature_names):
    data[feature] = tmp['data'][:,n]

def varimp(X, y,num_samples,feature_names,data):
    bins = {}
    bin_idx = (np.arange(0,1.1,0.1)*num_samples).astype(np.int16)

    bin_idx[-1] = bin_idx[-1]-1
```

```
28
29      for feature in (feature_names):
30          bins[feature] = np.sort(data[feature])[bin_idx]
31  # decision stumps as weak classifiers
32  # 0 if not in bin, 1 if in bin
33      stumps = {}
34      for feature in feature_names:
35          stumps[feature] = np.zeros([num_samples,len(bins[feature])-1])
36          for n in range(len(bins[feature])-1):
37              stumps[feature][:,n] =  data[feature]>bins[feature][n]
38  # stack the weak classifiers into a matrix
39      H = np.hstack([stumps[feature] for feature in feature_names])
40      H = np.hstack([np.ones([num_samples,1]),H])
41  # prepare the vector for storing weights
42      alphas = np.zeros(H.shape[1])
43      num_iterations = 30
44      MSE = np.zeros(num_iterations)
45      for iteration in range(num_iterations):
46          f = np.dot(H,alphas)
47          r = y-f; MSE[iteration] = np.mean(r**2) # r = residual
48          idx = np.min(np.where(abs(np.dot(np.transpose(r),H))==np.max(abs(np.dot(np
    .transpose(r),H)))))
49          alphas[idx] = alphas[idx] + np.dot(np.transpose(r),H[:,idx])/np.sum(H[:,
    idx])# amount to move in optimal direction
50      print(MSE[29])
51      alphasf = {}
52      start = 1
53      for feature in feature_names:
54          alphasf[feature] = alphas[start:(start+stumps[feature].shape[1])]
55          start = start + stumps[feature].shape[1]
56      alphasf['mean'] = alphas[0]
57      i=1
58      m=1
59      fig = plt.figure()
60      fig.subplots_adjust(hspace=1)
61      fig.set_figheight(15)
62      fig.set_figwidth(15)
63      plt.suptitle('contribution to house price')
64      for feature in feature_names:
65          #plt.close("all")
66
67          ax= fig.add_subplot(330 + m)
68          ax.plot(data[feature],y-np.mean(y),'.',alpha=0.5,color=[0.9,0.9,0.9])
69
70          # plot stuff
71          ax.plot(data[feature],np.dot(H[:,i:i+10],alphasf[feature])-np.mean(np.dot(
    H[:,i:i+10],alphasf[feature])),'.',alpha=0.5,color=[0,0,0.9])
72          ax.set_xlim([bins[feature][0],bins[feature][-2]])
73          ax.title.set_text(feature)
74          i=i+10
75          m+=1
76      plt.show()
77
78  varimp(X, y,num_samples,feature_names,data)# original data
79
80  # Variable Importance
81  for features in feature_names:
82      newdata=copy.deepcopy(data)
83      newdata[features] = np.random.permutation(newdata[features])
```

```
    print(features)
    varimp(X, y,num_samples,feature_names,newdata)

#Boosted Decision Tree
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.ensemble.partial_dependence import plot_partial_dependence
clf = GradientBoostingRegressor(loss="ls")
clf.fit(X,y)
plt.close("all")
plt.figure(figsize=[10,10])
ax = plt.gca()
plot_partial_dependence(clf, X, feature_names, feature_names, n_cols=3, ax=ax)
plt.tight_layout()
plt.show()

#Linear Regression
from sklearn.linear_model import LinearRegression
clf2 = LinearRegression()
clf2.fit(X,y)


#Comparison in MSE
print(np.mean((y-clf2.predict(X))**2))
print(np.mean((y-clf.predict(X))**2))
```

(i)(iii) The following graph shows the plots required in (i) and (iii) together, the grey dots represent the 'HousePrice' versus each feature q and the blue dots represent $f_q(x)$ with the feature value on horizontal axis. The features that determine the HousePrice have some increasing/decreasing or a mix of both trends. Those variables that do not play an impotant role are mostly just a horizontal line.



```
MedInc
0.7777653040421583
HouseAge
0.5042674576086329
AveRooms
0.4929544855901924
AveBedrms
0.49694458398888214
Population
0.4953180130155925
AveOccup
0.5551221947030573
Latitude
0.5278391365335422
Longitude
0.5288313805790584
```

(iv) Variable improtance can be defined by how much the mean square error increases after perfoming scrambling of data on that feature. So higher, the mean square error,more the importance of the variable. The above image shows the MSE based on the feature we are scrambling on. Since, the original MSE is 0.4953180130155925 we can see that MedInc has the highest variable importance, then AveOccup. All others are less important as compared to these 2. Population has the minimum variable importance. The following graphs show the change in plots when testing for variable importance:

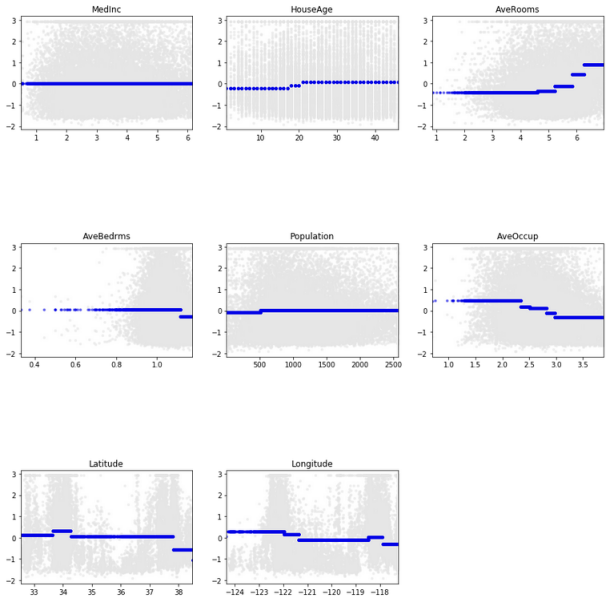


Figure 1: MedInc Variable Importance

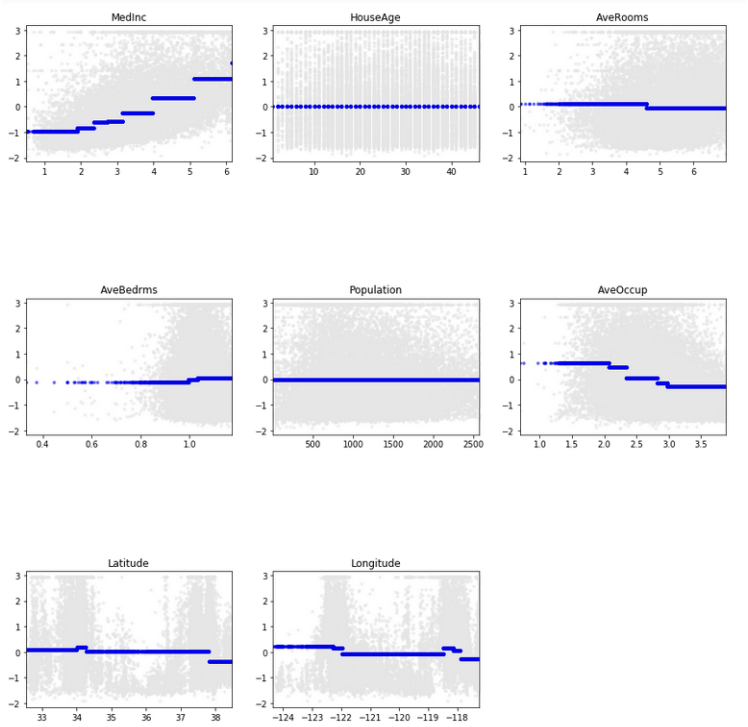


Figure 2: HouseAge Variable Importance

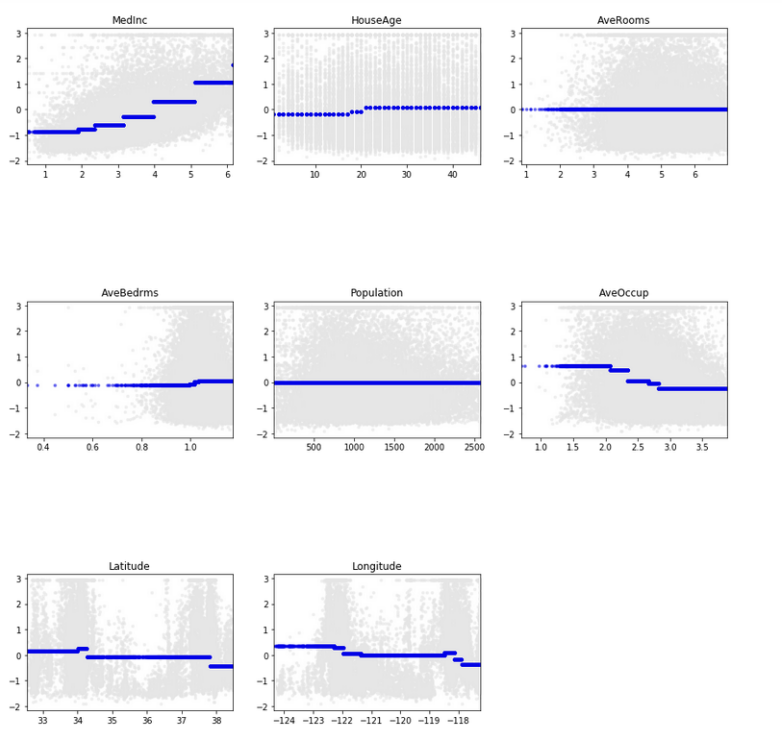


Figure 3: AveRooms Variable Importance

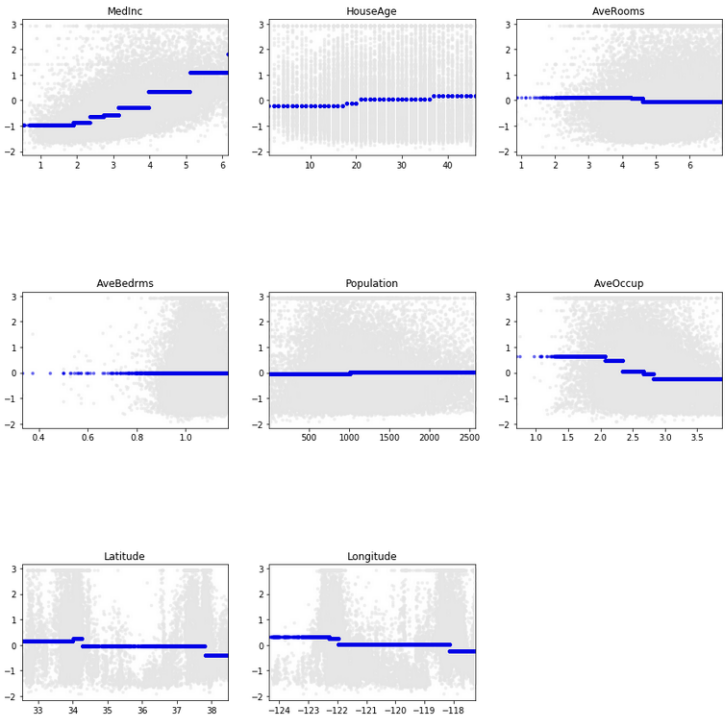


Figure 4: AveBedrms Variable Importance

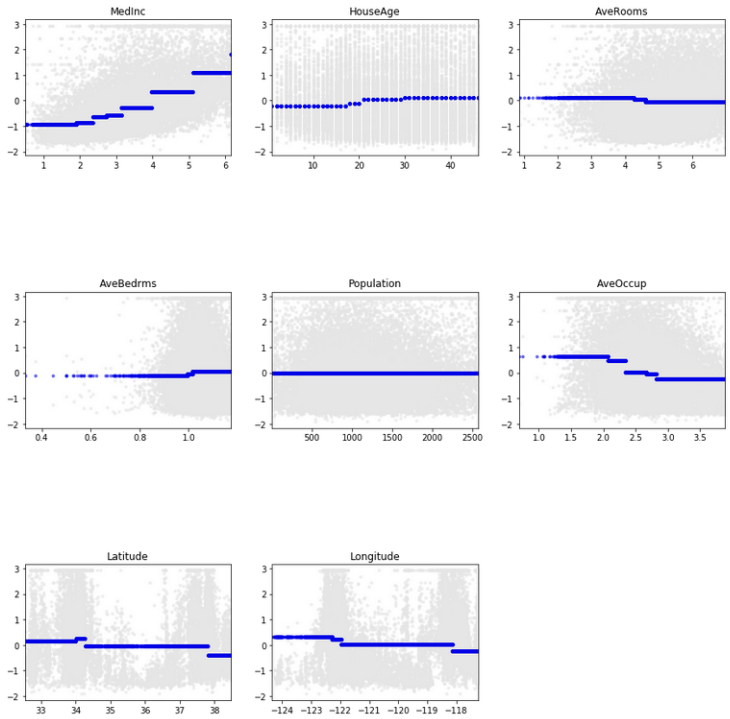


Figure 5: Population Variable Importance

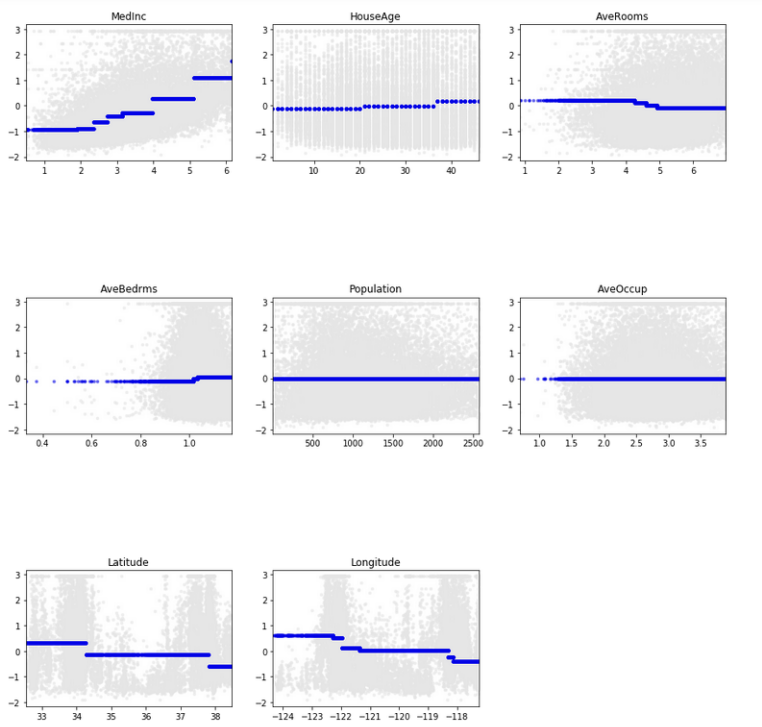


Figure 6: AveOccup Variable Importance

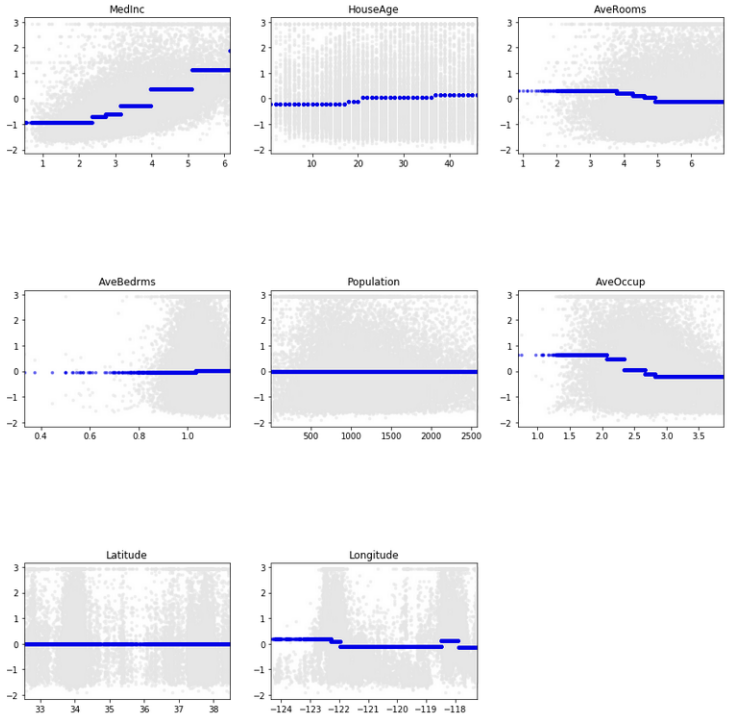


Figure 7: Latitude Variable Importance

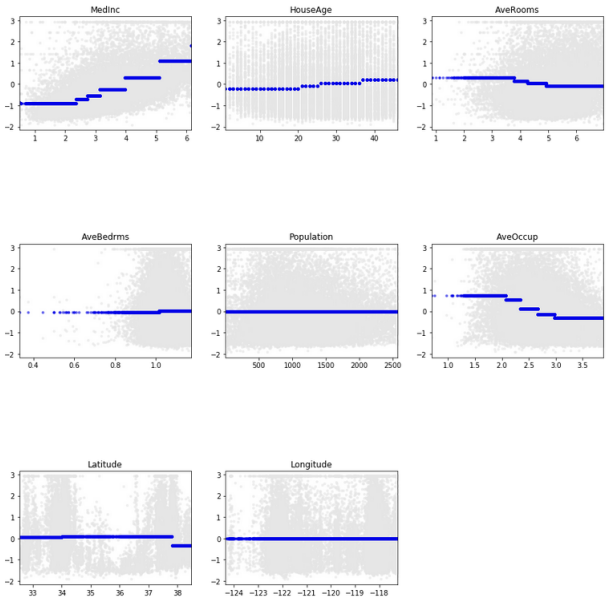


Figure 8: Longitude Variable Importance

As can be seen from the plots, while performing scrambling on the feature, the value of $f_q(x)$ becomes a horizontal straight line passing through 0. Also, the plots like MedInc and AveOccup that have trends in their graphs(when no scrambling happens) play an important role, as when their trend becomes a horizontal straight line (after scrambling), the MSE increases significantly.

(c) Compare with Linear Regression (5 points)

MSE for linear regression is 0.5243209861846071
MSE for boosted decisiom tree is 0.26188431965892933
MSE for our model is 0.4953180130155925

Boosting in general performs better than linear regression as there is no iterative optimization in linear regression. However, boosting methods reduce the error iteratively be selecting best classifier(non linear). Theoretically, they should overfit but in practical they do not.

(d) Bonus! (5 points)

If we make the weight update α at every iteration **positive**, then each $f_q(x_{\cdot,q})$ is monotonically increasing in $x_{\cdot,q}$. This will always cause $f_q(x_{\cdot,q})$ to either go up or stay constant at every iteration and the resultant graph would be monotonically increasing. If $x_{i,q} \geq x_{j,q}$ then $f_q(x_{i,q}) \geq f_q(x_{j,q})$ as all the classifier satisfying $h_k(x) = \mathbf{1}_{[x_{i,q} > I_k]}$ will automatically satisfy $h_k(x) = \mathbf{1}_{[x_{j,q} > I_k]}$ and since the weights are positive, it can only go up.

To make it monotonically decreasing, the weight update α at every iteration should be **negative**, then each $f_q(x_{\cdot,q})$ is monotonically decreasing in $x_{\cdot,q}$. This will always cause $f_q(x_{\cdot,q})$ to either go down or stay constant at every iteration and the resultant graph would be monotonically decreasing. If $x_{i,q} \geq x_{j,q}$ then $f_q(x_{i,q}) \leq f_q(x_{j,q})$ as all the classifier satisfying $h_k(x) = \mathbf{1}_{[x_{i,q} > I_k]}$ will automatically satisfy $h_k(x) = \mathbf{1}_{[x_{j,q} > I_k]}$ and since the weights are negaitive, it can only go down.

4 Random Forest (10 points)

(a) Strength of individual trees (4 points)

By allowing a tree to go to full depth, we can get a training error of 0. In other words, if we keep splitting until the entropy at each leaf node is 0, that is, no information can be gain from splitting on any feature, we can get a tree with training error 0. If the entropy at each leaf node is 0, then $P(y = 1|x)$ is either 1 or 0 as $\sum_{j=1}^{J} p_j log_2 p_j$ equals the Entropy where J are the total events. We can prove this by induction:

Base: Suppose there is only 1 point, then splitting on any feature would classify it correctly and give the training error to be 0.

Induction hypothesis: Now, assume we can split m data points correctly with training error 0.

Induction Step: With m+1 data points, the extra data point can either be classified correctly by the decision tree constructed using m data points or we can add another splitting node (feature) on the decision tree (split on the feature which makes it different from the other m data). This again gives the training error to be 0.

Therefore, we can always construct a decision tree which gives 0 training error by doing a greedy search for the best splitting point.

(b) Correlation between trees (6 points) The sample selection scheme (on its own) is known as "bagging", and the feature selection scheme is known as choosing a "random subspace".

Suppose we are performing regression, for a datapoint $\mathbf{x}$, we have a set of random variables $\{F_j\}_{j=1}^{M}$, where each $F_j = f_j(\mathbf{x}) \in \mathbb{R}$ is an output from a tree in the random forest.

(i) $\text{Var}(F_j) = \sigma^2$

Correlation $(F_i, F_j) = \frac{Covariance(F_i,F_j)}{\sqrt{Var(F_i)}\sqrt{Var(F_j)}} = \rho$

$Covariance(F_i, F_j) = \rho\sqrt{Var(F_i)}\sqrt{Var(F_j)} = \rho\sigma^2$

$\text{Var}(\frac{1}{M}\sum_{j=1}^{M} F_j) = \frac{1}{M^2} Var(\sum_{j=1}^{M} F_j)$

Now,

$$Var(\sum_{j=1}^{M} F_j) = \sum_{j=1}^{M} Var(F_i) + \sum_{i=1}^{M}\sum_{j=1}^{M} Covariance(F_i, F_j) I[i \neq j]$$

$$Var(\sum_{j=1}^{M} F_j) = M\sigma^2 + 2\binom{M}{2}\rho\sigma^2$$

$$Var(\sum_{j=1}^{M} F_j) = M\sigma^2 + M(M-1)\rho\sigma^2 = M\sigma^2(1 + \rho(M-1))$$

Therefore,

$$Var(\frac{1}{M}\sum_{j=1}^{M} F_j) = \frac{1}{M^2} M\sigma^2(1+\rho(M-1))$$

which equals

$$\frac{\sigma^2}{M}(1+\rho(M-1)).$$

(ii) As $M \to \infty$, $Var(\frac{1}{M}\sum_{j=1}^{M} F_j)$ becomes $\rho\sigma^{\mathbf{2}}$ using limit properties.

(iii) ρ can be decreased in various ways:

a) Suppose there are n data points and m trees, and n>>m, then we can construct tree using n/m points and these points are mutually exclusive for every tree. Since, every tree would be trained on a different data, we can expect ρ to decrease. Howere, this method effectively removes the fact that we are sampling with replacement and our results might not be good.

b) Suppose there are p features of the data point, and m trees, and p>>m, then we can construct trees using p/m features and these features are mutually exclusive. Since, every tree would be trained using a different set of feature we can expect the correlation to be really low. However, the variance might increase for individual tree.

c) We can also use a combination of both methods mentioned above.

d) We can do bagging first with z data points for every tree where $z < n$ and then perform a random subspacing at each split of the tree with features $m < p$ (p represent the total features). This will also reduce the overall ρ as compared to doing it without subspacing and bagging, however the process might be computationally expensive