Mid-Semester

Problem 1 We have four points , lets name them as

1. A (0.3 , 0.8)
2. B (0.4, 0.3)
3. C (0.2 , 0.4)
4. D(0.6,0.56)

We can see that D is the only classifier whose FPR is greater than TPR , so D turn out to be bad classifier, If someone aims to reduce only false predicted cases C will be the best choice , point A is the best choice in terms of ROC curve

Problem 2 We have given $f(x_i) = w_i$ for all i = 1,2,3,4,5,........n

Then the empirical distribution is given by

$$F_n(x) = \sum_{i=1}^{n} f(x_i) 1_{(X_i \leq x)} = \sum_{i=1}^{n} w_i 1_{(X_i \leq x)}$$

and the α^{th} quantile is given by

$$F^{-1}(\alpha) = \{x_i : \sum_{j=0}^{i-1} w_i < \alpha \leq \sum_{j=0}^{i} w_i\}$$

Problem 3 We have

$$y_1, y_2, y_3 ... y_n | \sigma^2 \sim N(0, \sigma^2) p(\sigma^2) \propto (\sigma^2)^{-\frac{5}{2}-1} \exp(-\frac{1}{2\sigma^2})$$

Now the posterior will be given by

$$p(\sigma^2|y_1, y_2...y_n) \propto p(y_1, y_2...y_n|\sigma^2) \cdot p(\sigma^2) p(\sigma^2|y_1, y_2...y_n) \propto (\sigma^2)^{-\frac{5}{2}-1} \exp(-\frac{1}{2\sigma^2}) \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}(\frac{y_i}{\sigma})^2} p(\sigma^2|y_1, y_2$$

So the kernel here is of inverse gamma with

$$\alpha = \frac{n+5}{2} = 52.5, \quad \beta = \frac{\sum_{i=1}^{n} y_i^2 + 1}{2}$$

Approximate Bayes Estimate

```
x <- rnorm(100 , 0 , sqrt(5))
y <- rinvgamma(30, 52.5 , rate = (sum(x^2)+1)/2)
mean(y)
```

Credible interval for σ^2

Credible interval for the σ^2 is given by , lets say

$$p(\sigma^2|y_1, y_2...y_n) = \frac{(\sigma^2)^{-\frac{n}{2}-\frac{5}{2}-1}\exp(-\frac{1}{2\sigma^2}\bigg(\sum_{i=1}^{n} y_i^2 + 1\bigg))}{K}$$

Where K is a normalising constant, assume F as cumulative distribution for the given posterior then the credible interval will be given by

$$F(\sigma_L^2|y_1, y_2....y_n) = \alpha/2 \quad and \quad F(\sigma_U^2|y_1, y_2....y_n) = 1 - \alpha/2$$

```
x <- rnorm(100 , 0 , sqrt(5))
y <- rinvgamma(30, 52.5 , rate = (sum(x^2)+1)/2)
mean(y)
shaped = 52.5
rated = (sum(x^2)+1)/2
mode = rated / ( (shaped) + 1 )
ruler1 <- seq(0.001, mode , length = 1000)
ruler2 <- seq(mode , 5 , length = 1000)
target = 0.95
tolerance = 0.0005
done <- FALSE
for(i in ruler1){
  for(j in ruler2){
    if(round(dinvgamma(i ,shaped , rate = rated),3) == round(dinvgamma(j, shaped , rate =rat
      L <- pinvgamma(i ,shape , rate = rate)
      H <- pinvgamma(i ,shape , rate = rate)
      if((((H-L)< target+tolerance)) & (((H-L) > target-tolerance))){
        done <- TRUE
        break
      }
    }
  }
  if(done){break}
}
HPD.L <- i;HPD.U <- j
print(paste(target*100 , "% HPD interval:",HPD.L, "to", HPD.U))
```

Problem 4

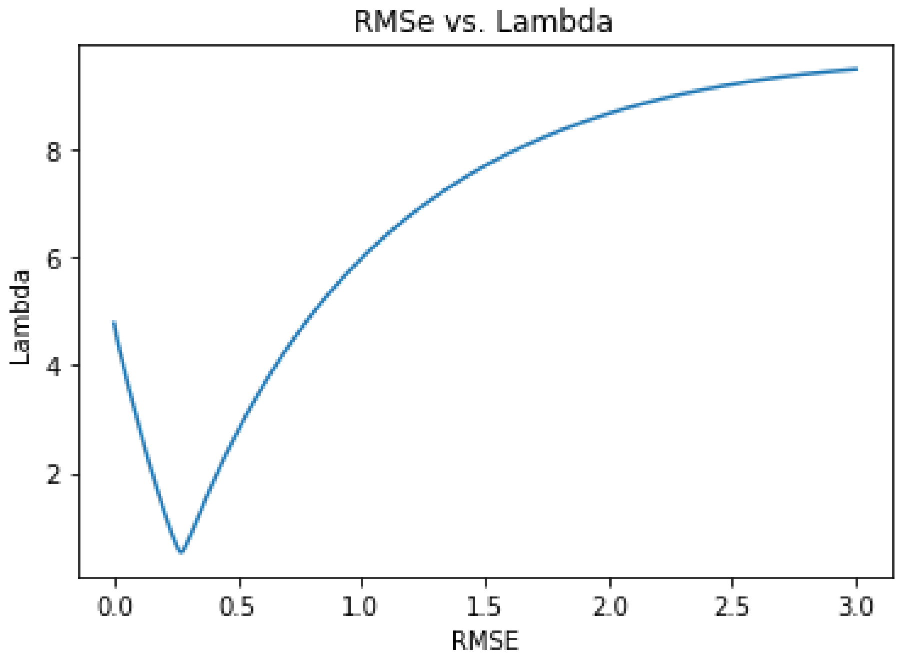
```python
import numpy as np
from sklearn.model_selection import train_test_split,KFold

X = np.squeeze(np.arange(1,37)).astype(np.float64)
y = np.loadtxt("FRWRD.txt", delimiter="\n", unpack=False)
X = np.c_[X , X**2 , X**3 ,X**4 ,X**5 ,X**6 ,X**7 ,X**8]
data = np.c_[X , y]


class cv_model:
    def __init__(self,model,data,fold,lambda_=None):
        self.model = model
        self.X  = data[:,:-1]
        self.y  = data[:,-1]
        self.fold  = fold
        self.lambda_ = lambda_

    def compute(self):
        kf = KFold(n_splits=self.fold)
        regressor = self.model(self.lambda_)
        a = np.empty(self.fold,dtype=np.float64)
        b = np.empty(self.fold,dtype=np.float64)
        count = -1
        for train_index, test_index in kf.split(self.X):
            count = count +1
            X_train, X_test = self.X[train_index], self.X[test_index]
            y_train, y_test = self.y[train_index], self.y[test_index]
            regressor.fit(X_train,y_train)
            prediction = regressor.predict(X_test)
            rmse_ =  metrics.mean_squared_error(y_test,prediction,squared=False)
            r2_ = metrics.r2_score(y_test,prediction)
            a[count]=rmse_
            b[count]=r2_
        self.rmse = a
        self.r2 = b
        return self

 class ridge:
    def __init__(self, lambda_):
        self.lambda_ = lambda_

    def fit(self,x,y):
        x = np.c_[np.ones(x.shape[0]),x]
        x_transpose = x.transpose()    # Transpose of x_train
        x_t_x = np.matmul(x_transpose,x)
        x_t_x_l_inv = np.linalg.inv(x_t_x+self.lambda_ * np.identity(x_t_x.shape[1]))
```

```
        self.beta = np.matmul(np.matmul(x_t_x_l_inv,x_transpose),y)
        return self

    def predict(self,x_test):
        x_test = np.c_[np.ones(x_test.shape[0]),x_test]
        return np.matmul(x_test,self.beta)


# Finding best lambda
diff_values = np.linspace(0,3 ,num = 1000)
rmse = []
opt_rmse = 5
opt_lambda = 0
for i in diff_values:
  cv_ridge = cv_model(ridge,data,5,lambda_=i)
  cv_ridge.compute()
  mean = cv_ridge.rmse.mean()
  rmse.append(mean)
  if mean < opt_rmse :
    opt_rmse = mean
    opt_lambda = i
print("Optimal RMSE is "+ str(opt_rmse)+" for regularizer constant " + str(opt_lambda))
```

Output : Optimal RMSE is 0.5331170378761044 for regularizer constant 0.2732732732732733

```
# importing the required module
import matplotlib.pyplot as plt

# x axis values
x = diff_values
# corresponding y axis values
y_ = rmse

# plotting the points
plt.plot(x, y_)


# naming the x axis
plt.xlabel('RMSE')
# naming the y axis
plt.ylabel('Lambda')

# giving a title to my graph
```

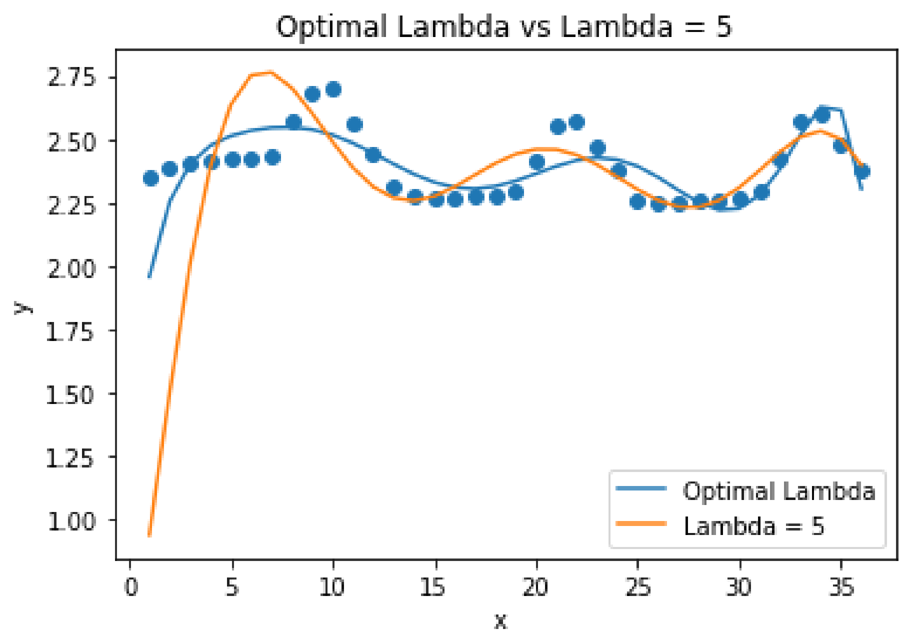
```
plt.title('RMSe vs. Lambda')
```



Output :

```
# Plotting fit with Optimal Regularizer Vs. Arbitary Regularizer


x1 =  np.squeeze(np.arange(1,37)).astype(np.float64)
estimator1 = ridge(lambda_ = 0.27327)
estimator1.fit(X,y)
y1 = estimator1.predict(X)
plt.plot(x1, y1, label = "Optimal Lambda")

# line 2 points

estimator2 = ridge(lambda_ = 5.0)
estimator2.fit(X,y)
y2 = estimator2.predict(X)
plt.plot(x1, y2, label = "Lambda = 5")

# naming the x axis
plt.xlabel('x ')
# naming the y axis
plt.ylabel('y')
# giving a title to my graph
plt.title('Optimal Lambda vs Lambda = 5')
```

```
# show a legend on the plot
plt.legend()
plt.scatter(x1 , y)
# function to show the plot
plt.show()
```



Output : (1).png

Problem 5

```
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import pandas as pd
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np

class SoftmaxRegression(nn.Module):
    def __init__(self, n_input_features, num_classes):
        super(SoftmaxRegression, self).__init__()
        self.linear = nn.Linear(n_input_features, num_classes)
```

```
    def forward(self, x):
        y_pred = torch.softmax(self.linear(x), dim=1)
        return y_pred


def get_accuracy(X_test, y_test, model, rev_map):
    correct,total = 0,0
    confusion_matrix = np.zeros((8,8))

    with torch.no_grad():
        y_predicted = model(X_test)
        _,y_predicted_cls = torch.max(y_predicted.data, 1)
        acc = y_predicted_cls.eq(y_test).sum() / float(y_test.shape[0])

        for i in range(y_predicted_cls.shape[0]):
            confusion_matrix[int(y_test[i].item())][int(y_predicted_cls[i].item())]+=1

        print("-------- Accuracy --------")
        print(f'accuracy: {acc.item():.4f}')
        print("-------- Confusion Matrix --------")
        print(confusion_matrix)
        print_precision_recall(confusion_matrix,rev_map)

def print_precision_recall(confusion_matrix,rev_map):
    total_precision = 0
    total_recall = 0

    print("---------- Precision -----------")
    for i in range(confusion_matrix.shape[0]):
        if sum(confusion_matrix[i,:]) != 0:
            print(f'class {rev_map[i]}: {confusion_matrix[i,i]/sum(confusion_matrix[i,:])}')
            total_precision += confusion_matrix[i,i]/sum(confusion_matrix[i,:])
        else:
            print(f'class {rev_map[i]}: 0')

    print("--------- Recall ----------")
    for i in range(confusion_matrix.shape[0]):
        if sum(confusion_matrix[:,i]) != 0:
            print(f'class {rev_map[i]}: {confusion_matrix[i,i]/sum(confusion_matrix[:,i])}')
            total_recall += confusion_matrix[i,i]/sum(confusion_matrix[:,i])
        else:
            print(f'class {rev_map[i]}: 0')

    print("------- Macro Precision --------")
    print(total_precision/8)
```

```
    print("--------- Macro Recall ---------")
    print(total_recall/8)

def normalize(X,y):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state
    sc = StandardScaler()
    X_train = sc.fit_transform(X_train)
    X_test = sc.transform(X_test)
    X_train = torch.from_numpy(X_train.astype(np.float32))
    X_test = torch.from_numpy(X_test.astype(np.float32))
    y_train = torch.from_numpy(y_train.astype(np.float32)).long()
    y_test = torch.from_numpy(y_test.astype(np.float32)).long()
    return X_train, X_test, y_train, y_test

def softmax_regression(X, y, num_classes, num_epochs, learning_rate, rev_map):
    n_samples, n_features = X.shape
    X_train, X_test, y_train, y_test = normalize(X,y)
    model = SoftmaxRegression(n_features,num_classes)
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

    for epoch in range(num_epochs):
        y_pred = model(X_train)
        loss = criterion(y_pred, y_train)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
        if (epoch+1) % 100 == 0:
            print(f'epoch: {epoch+1}, loss = {loss.item():.4f}')

    get_accuracy(X_test, y_test, model,rev_map)

ecoli = pd.read_csv('ecoli.data', delim_whitespace=True, header=None)

data_ecoli = ecoli.iloc[:,1:7].values
ecoli_target = ecoli.iloc[:,8].values

map_hash={}
rev_map={}
k=0
for i in range(len(ecoli_target)):
    if ecoli_target[i] not in map_hash:
        map_hash[ecoli_target[i]] = k
        rev_map[k] = ecoli_target[i]
        k+=1
    else:
```

```
        map_hash[ecoli_target[i]] = map_hash[ecoli_target[i]]

ecoli_target = np.array([map_hash[i] for i in ecoli_target])

print("======== Ecoli dataset =========")
softmax_regression(data_ecoli, ecoli_target, num_classes=8, num_epochs=10000, learning_rate=
```

Output :

-------- Accuracy --------

accuracy: 0.8382

-------- Confusion Matrix --------

[[29. 0. 0. 0. 0. 0. 0. 0.]

[0. 12. 0. 0. 1. 0. 0. 0.]

[0. 0. 0. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 0. 0. 0. 0.]

[1. 5. 0. 0. 3. 0. 0. 0.]

[0. 0. 0. 0. 0. 2. 0. 3.]

[0. 0. 0. 0. 0. 0. 1. 0.]

[0. 1. 0. 0. 0. 0. 0. 10.]]

---------- Precision -----------

class cp: 1.0

class im: 0.9230769230769231

class imS: 0

class imL: 0

class imU: 0.3333333333333333

class om: 0.4

class omL: 1.0

class pp: 0.9090909090909091

--------- Recall ----------

class cp: 0.9666666666666667

class im: 0.6666666666666666

class imS: 0

class imL: 0

class imU: 0.75

class om: 1.0

class omL: 1.0

class pp: 0.7692307692307693

------- Macro Precision --------

0.5706876456876456

--------- Macro Recall ---------

0.6440705128205128

Problem 6

```
import numpy as np
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import torch
import torch.nn as nn
import torch.nn.functional as F


class Model(nn.Module):
    def __init__(self, n_input_features):
        super(Model, self).__init__()
        self.linear = nn.Linear(n_input_features, 1)


    def forward(self, x):
        y_pred = torch.sigmoid(self.linear(x))
        return y_pred




def logistic_regression(X, y, num_epochs, learning_rate):


    n_samples, n_features = X.shape
```

```python
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state


sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)


X_train = torch.from_numpy(X_train.astype(np.float32))
X_test = torch.from_numpy(X_test.astype(np.float32))
y_train = torch.from_numpy(y_train.astype(np.float32)).long()
y_test = torch.from_numpy(y_test.astype(np.float32)).long()




y_train = y_train.view(y_train.shape[0], 1).type(torch.FloatTensor)
y_test = y_test.view(y_test.shape[0], 1).type(torch.FloatTensor)


model = Model(n_features)


criterion = nn.BCELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)


for epoch in range(num_epochs):
    y_pred = model(X_train)
    loss = criterion(y_pred, y_train)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
    if (epoch+1) % 100 == 0:
        print(f'epoch: {epoch+1}, loss = {loss.item():.4f}')


correct,total = 0,0

confusion_matrix = np.zeros((2,2))


with torch.no_grad():
    y_predicted = model(X_test)
    y_predicted_class = y_predicted.round()
```

```
        acc = y_predicted_class.eq(y_test).sum() / float(y_test.shape[0])

        for i in range(y_predicted_class.shape[0]):
            confusion_matrix[1 - int(y_test[i].item())][1-int(y_predicted_class[i].item())]

        print(f'accuracy: {acc.item():.4f}')

    cost_matrix = np.array([[0,4],[1,0]])

    print("Confusion Matrix")
    print(confusion_matrix)

    print("Cost Matrix")
    print(cost_matrix)

    cost = np.sum(confusion_matrix*cost_matrix)
    print(f'cost: {cost.item():.4f}')

# Load breast cancer dataset
bc_dataset = datasets.load_breast_cancer()

print("=============== Breast cancer dataset ================")
logistic_regression(bc_dataset.data, bc_dataset.target, num_epochs = 1000, learning_rate =
```

Output:

================ Breast cancer dataset =================

epoch: 100, loss = 0.2501

epoch: 200, loss = 0.1871

epoch: 300, loss = 0.1584

epoch: 400, loss = 0.1412

epoch: 500, loss = 0.1295

epoch: 600, loss = 0.1210

epoch: 700, loss = 0.1145

epoch: 800, loss = 0.1093

epoch: 900, loss = 0.1050

epoch: 1000, loss = 0.1014

accuracy: 0.9737

Confusion Matrix

[[73. 1.]

[2. 38.]]

Cost Matrix

[[0 4]

[1 0]]

cost: 6.0000