Machine Learning Final Exam, Aug Semester, 2020

In this exam, you will demonstrate your understanding of the material from the lectures and labs.

For each question, insert your answer directly in this sheet. When complete, export the sheet as a PDF and upload to Gradescope. Note that you have **2.5 hours** to do the exam. Also note that there are some short answer questions that you may be able to answer faster than the coding questions. You might consider answering those questions first to get as much credit as possible!

Question 1 (10 points)

Download the CSV dataset for the exam. Note that the data are two dimensional with labels '0' and '1'. Provide a scatter plot for the data with the two classes shown in different colors.

In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from torch.autograd import Variable
```

In [ ]:
```
# Place code to load the data and plot the scatterplot here

df = pd.read_csv('final-exam-data.csv')
```

In [ ]:
```
df.head()
```

In [ ]:
```
df = df.rename(columns={'# x1': 'x1'})
```

In [ ]:
```
df.head()
```

In [ ]:
```
plt.figure(figsize=(15, 5))
plt.plot(df['x1'][df['y']==0], df['x2'][df['y']==0], 'ro')
plt.plot(df['x1'][df['y']==1],df['x2'][df['y']==1], 'bo')
plt.show()
```

Question 2 (30 points)

Do the following:

1. Split the dataset from Question 1 into a training set (80%) and validation set (20%).
2. Using the logistic regression code we developed in lab, fit a logistic regression model to the data in the training set.

3. Indicate the training set accuracy and validation set accuracy of the logistic regression model.
4. Make two separate plots, one for the training set and one for the validation set, with three colors: one for correctly classified points from class 1, one for correctly classified points from class 2, and one for incorrectly classified points.

In [ ]:
```
def normalize_data(data):

    means = np.mean(data, axis=0) # calculate mean
    stds = np.std(data, axis=0) # calculate std
    data_norm = (data - means) / stds # from formula = (x-mean)/std

    return data_norm
```

In [ ]:
```
# Place code to split the data, fit the LR model, output accuracy, and plot the res

import random
def partition(X, y, percent_train=0.8):
    m = y.shape[0]
    idx = np.arange(0,m)
    random.seed(1412)
    random.shuffle(idx)
    #percent_train = 0.6
    m_train = int(m * percent_train)
    train_idx = idx[:m_train]
    test_idx = idx[m_train:]
    X_train = X[train_idx]
    y_train = y[train_idx]
    X_test = X[test_idx]
    y_test = y[test_idx]
    return X_train, X_test,y_train, y_test

X = df[['x1', 'x2']].to_numpy()
y = df['y'].to_numpy()

X_norm = normalize_data(X)

X_train_norm, X_test_norm, y_train, y_test = partition(X_norm, y)

print(X_train_norm.shape, X_test_norm.shape, y_train.shape, y_test.shape)
```

In [ ]:
```
y_train = y_train.reshape(-1, 1)
y_test = y_test.reshape(-1, 1)
```

In [ ]:
```
# Reshape, good y is 320,1


print(X_train_norm.shape, X_test_norm.shape, y_train.shape, y_test.shape)
```

In [ ]:
```
# np.ones((shape))
intercept = np.ones((X_train_norm.shape[0], 1))

# concatenate the intercept based on axis=1
```

```
X_train_norm = np.concatenate((intercept, X_train_norm), axis=1)

# np.ones((shape))
intercept = np.ones((X_test_norm.shape[0], 1))

# concatenate the intercept based on axis=1
X_test_norm = np.concatenate((intercept, X_test_norm), axis=1)

print(X_train_norm.shape, X_test_norm.shape, y_train.shape, y_test.shape)
```

In [ ]:
```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def h(X, theta):
    return sigmoid(X @ theta)

def grad_j(X, y, y_pred):
    return X.T @ (y - y_pred) / X.shape[0]

def j_ep(theta, X, y):
    epsilon = 1e-13
    y_pred = h(X, theta)
    error = (-y * np.log(y_pred+epsilon)) - ((1 - y) * np.log(1 - y_pred + epsilon))
    cost = sum(error) / X.shape[0]
    grad = grad_j(X, y, y_pred)
    return cost[0], grad

def j(theta, X, y):

    y_pred = h(X, theta)
    error = (-y * np.log(y_pred)) - ((1 - y) * np.log(1 - y_pred))
    cost = sum(error) / X.shape[0]
    grad = grad_j(X, y, y_pred)
    return cost[0], grad

def train(X, y, theta_initial, alpha, num_iters):
    theta = theta_initial
    j_history = []
    for i in range(num_iters):
        cost, grad = j(theta, X, y)
        theta = theta + alpha * grad
        j_history.append(cost)
    return theta, j_history
```

In [ ]:
```
np.zeros((X_train_norm.shape[1], 1))
```

In [ ]:
```
theta_initial = np.zeros((X_train_norm.shape[1], 1))
alpha = 0.001
num_iters = 2000
theta, j_history = train(X_train_norm, y_train, theta_initial, alpha, num_iters)

print("Theta optimized:", theta)
print("Cost with optimized theta:", j_history[-1])
```

In [ ]:
```
j_history_list = []
```

```
theta_list = []
alpha_list = [0.01]
theta_initial1 = np.zeros((X_train_norm.shape[1], 1))
theta_initial_list = [theta_initial1]

num_iters = 5000

for alpha in alpha_list:
    for theta_initial in theta_initial_list:

        # YOUR CODE HERE

        theta_i, j_history_i = train(X_train_norm, y_train, theta_initial, alpha, n

        #raise NotImplementedError()
        #theta_i, j_history_i = None, None

        j_history_list.append(j_history_i)
        theta_list.append(theta_i)
```

In [ ]:
```
plt.plot(j_history_i)
plt.xlabel("Iteration")
plt.ylabel("$J(\theta)$")
plt.title("Training cost over time with batch gradient descent (no normalization)")
plt.show()
```

In [ ]:
```
def r_squared(y, y_pred):
    return 1 - (np.square(y - y_pred).sum() / np.square(y - y.mean()).sum())
```

In [ ]:
```
y_test_pred_soft_norm = h(X_test, theta)
y_test_pred_hard_norm = (y_test_pred_soft_norm > 0.5).astype(int)

test_rsq_soft_norm = r_squared(y_test, y_test_pred_soft_norm)
test_rsq_hard_norm = r_squared(y_test, y_test_pred_hard_norm)
test_acc_norm = (y_test_pred_hard_norm == y_test).astype(int).sum() / y_test.shape[

print('Got test set soft R^2 %0.4f, hard R^2 %0.4f, accuracy %0.2f from Normalized
```

In [ ]:
```
def boundary_points(X, theta):
    v_orthogonal = np.array([[theta[1,0]],[theta[2,0]]])
    v_ortho_length = np.sqrt(v_orthogonal.T @ v_orthogonal)
    dist_ortho = theta[0,0] / v_ortho_length
    v_orthogonal = v_orthogonal / v_ortho_length
    v_parallel = np.array([[-v_orthogonal[1,0]],[v_orthogonal[0,0]]])
    projections = X @ v_parallel
    proj_1 = min(projections)
    proj_2 = max(projections)
    point_1 = proj_1 * v_parallel - dist_ortho * v_orthogonal
    point_2 = proj_2 * v_parallel - dist_ortho * v_orthogonal
    return point_1, point_2
```

In [ ]:
```
idx_0 = np.where(y == 0)
idx_1 = np.where(y == 1)

fig1 = plt.figure(figsize=(5,5))
```

```
ax = plt.axes()
ax.set_aspect(aspect = 'equal', adjustable = 'box')
plt.title('Logistic regression boundary')
plt.xlabel('Exam 1')
plt.ylabel('Exam 2')
plt.grid(axis='both', alpha=.25)
ax.scatter(X_norm[:,0][idx_0], X_norm[:,1][idx_0], s=50, c='r', marker='*', label='
ax.scatter(X_norm[:,0][idx_1], X_norm[:,1][idx_1], s=50, c='b', marker='o', label='
point_1, point_2 = boundary_points(X_norm, theta)
plt.plot([point_1[0,0], point_2[0,0]],[point_1[1,0], point_2[1,0]], 'g-')
plt.legend(loc=0)
plt.show()
```

Question 3 (20 points)

Repeat Question 2 using the PyTorch neural network library. Your PyTorch model should have a single linear layer with two inputs and a single output, a logistic sigmoid activation function, binary cross entropy loss function, and stochastic gradient descent for the optimizer. Show the same plots you showed for Question 2.

In [ ]:
```
# Place code to build, train, and evaluate your PyTorch model here

import torch   #general pytorch
import torch.nn as nn  #neural network module
import torch.nn.functional as F  #useful functions like softmax, or relu

#pip install torchvision; conda install torchvision
from torchvision import datasets, transforms  #transforms for image processing
from torch.utils.data import DataLoader       #dataloader for preparing batch

import numpy as np
import pandas as pd
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt
```

In [ ]:
```
import random
def partition(X, y, percent_train=0.8):
    m = y.shape[0]
    idx = np.arange(0,m)
    random.seed(1412)
    random.shuffle(idx)
    #percent_train = 0.6
    m_train = int(m * percent_train)
    train_idx = idx[:m_train]
    test_idx = idx[m_train:]
    X_train = X[train_idx]
    y_train = y[train_idx]
    X_test = X[test_idx]
    y_test = y[test_idx]
    return X_train, X_test,y_train, y_test

X = df[['x1', 'x2']].to_numpy()
y = df['y'].to_numpy()
```

```
X_train, X_test, y_train, y_test = partition(X, y)
```

In [ ]:
```
X_train.shape
```

In [ ]:
```
X_train_tensor = torch.tensor(X_train)
y_train_tensor = torch.tensor(y_train)

print(type(X_train_tensor), type(y_train_tensor))
```

In [ ]:
```
class Network(nn.Module):

    def __init__(self, input_size, output_size):
        super().__init__()  #inherit everything from nn.Module
        self.layer1 = nn.Linear(input_size, output_size)  #(784, 89)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        out = self.layer1(x)
        out = self.sigmoid(out)

        return out
```

In [ ]:
```
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using {device} device")
```

In [ ]:
```
model = Network(2, 2).to(device)
model
```

In [ ]:
```
J_fn = nn.BCELoss()
#define the optimizer
optimizer = torch.optim.SGD(model.parameters(), lr = 0.00001)
```

In [ ]:
```
X_train_tensor = X_train_tensor.to(device)
y_train_tensor = y_train_tensor.to(device)
```

In [ ]:
```
X_train_tensor.shape[0]
```

In [ ]:
```
epochs = 100000
batch_size = 500
loss_log = []

for e in range(epochs):
    for i in range(0, X_train_tensor.shape[0], batch_size):
        x_mini = X_train_tensor[i:i + batch_size]
        y_mini = y_train_tensor[i:i + batch_size]

        x_var = Variable(x_mini, requires_grad=True).to(torch.float32)
        y_var = Variable(y_mini, requires_grad=True).to(torch.float32)
        #print(x_var.dtype)
        #print(y_var.dtype)
        optimizer.zero_grad()
```

```
        net_out = model(x_var)
        _, y_pred = torch.max(net_out, 1)
        #print(y_pred.dtype)
        y_pred = y_pred.to(torch.float32)
        loss = J_fn(y_pred, y_var)



        loss.backward()
        optimizer.step()

        #print(loss)

        if i % 1000 == 0:
            loss_log.append(loss.item())

    print('Epoch: {} - Loss: {:.6f}'.format(e, loss.item()))
```

Question 4 (20 points)

Add a 10-unit hidden layer with ReLU activation to the PyTorch model from Question 3. Plot training loss and validation loss as function of epoch of training. Do you see any evidence of overfitting?

In [ ]:
```
# Place code to build, train, and evaluate your PyTorch model here
```

Discuss whether you observe overfitting here.

Question 5 (10 points)

In fact, the data from Question 1 were generated from a Gaussian mixture model. If you were given the data without the labels, for unsupervised learning, do you think the EM algorithm for Gaussian mixtures with $k = 2$ components would recover the two classes? Why or why not?

Discuss here.

Question 6 (10 points)

Consider the following problem: Students taking classes in Room TC 103 often feel the room is too cold or too hot. To solve the problem, we connect three air conditioners' on/off switches to a computer, place a temperature sensor outdoors next to the room, and put a button box on each student's desk with 3 buttons, labeled "I'm freezing," "I'm sweating", and "I feel good." Students can press any of the three buttons to indicate their comfort level. Every 10 minutes, the outdoor temperature is measured in degrees Celsius, and the current comfort level of each student (1, 2, or 3) is measured. The system gets the measurements as

input and then has to set the control for each of the three air conditioners to high, low, or off for the next 10 minutes.

Suppose you would like to build a reinforcement learning agent to optmize the air conditioning control for the room. Do the following:

1. Briefly explain the state space, action space, reward function, and discount factor you think should be used for this problem.
2. Briefly explain the reason it is difficult to know the transition probabilities for this problem.
3. Briefly describe how you could use a neural network to learn the state transition probabilities.

Write your explanations here.

In [ ]: