def compute\_cost(x, y, w, b):

"""

Computes the cost function for linear regression.

Args:

x (ndarray): Shape (m,) Input to the model (Population of cities)

y (ndarray): Shape (m,) Label (Actual profits for the cities)

w, b (scalar): Parameters of the model

Returns

total\_cost (float): The cost of using w,b as the parameters for linear regression

to fit the data points in x and y

"""

# number of training examples

m = x.shape[0]

# You need to return this variable correctly

total\_cost = 0

### START CODE HERE ###

i=0

for i in range(0,m):

total\_cost = total\_cost + ( x[i]\*w +b - y[i] )\*\*2

total\_cost = 1/(2\*m)\*total\_cost

### END CODE HERE ###

return total\_cost

practical 2

# UNQ\_C2

# GRADED FUNCTION: compute\_gradient

def compute\_gradient(x, y, w, b):

"""

Computes the gradient for linear regression

Args:

x (ndarray): Shape (m,) Input to the model (Population of cities)

y (ndarray): Shape (m,) Label (Actual profits for the cities)

w, b (scalar): Parameters of the model

Returns

dj\_dw (scalar): The gradient of the cost w.r.t. the parameters w

dj\_db (scalar): The gradient of the cost w.r.t. the parameter b

"""

# Number of training examples

m = x.shape[0]

# You need to return the following variables correctly

dj\_dw = 0

dj\_db = 0

delta = 0

### START CODE HERE ###

for i in range(0,m):

delta = ((w\*x[i]+b) - y[i])

dj\_dw = dj\_dw + delta\*x[i]

dj\_db = dj\_db + delta

dj\_dw = 1/m\*dj\_dw

dj\_db = 1/m\*dj\_db

### END CODE HERE ###

return dj\_dw, dj\_db

practical lab 2

if np.isscalar(z):

g = 1/(1 +math.exp(-z))

else:

if np.ndim (z) == 1:

g = np.empty(z.shape)

for i in range(0,z.shape[0]):

g[i] = 1/(1 +math.exp(-z[i]))

else:

g = np.empty(z.shape)

rows = z.shape[0]

cols = z.shape[1]

for j in range(0,cols):

for i in range (0,rows):

g[j,i] = 1/(1 +math.exp(-z[j,i]))

### START CODE HERE ###

total\_cost = 0

loss = 0

for i in range(0,m):

temp\_z = 0

for j in range(0,n):

temp\_z = temp\_z + X[i,j]\*w[j]

temp\_z = temp\_z+ b

z = sigmoid(temp\_z)

loss = -y[i]\*np.log(z) - (1-y[i])\*np.log(1-z)

total\_cost = total\_cost + loss

total\_cost = total\_cost/m

### END CODE HERE ###

m, n = X.shape

dj\_dw = np.zeros(w.shape)

dj\_db = 0.

### START CODE HERE ###

for i in range(m):

z\_wb = 0

for j in range(n):

z\_wb += X[i,j]\*w[j]

z\_wb += b

f\_wb = sigmoid(z\_wb)

dj\_db\_i = f\_wb - y[i]

dj\_db += dj\_db\_i

for j in range(n):

dj\_dw\_ij = (f\_wb - y[i])\*X[i,j]

dj\_dw[j] += dj\_dw\_ij

dj\_dw = (1/m)\*dj\_dw

dj\_db = (1/m)\*dj\_db

### END CODE HERE ###

### START CODE HERE ###

# Loop over each example

for i in range(m):

z\_wb = 0

# Loop over each feature

for j in range(n):

# Add the corresponding term to z\_wb

z\_wb += X[i,j]\*w[j]

# Add bias term

z\_wb += b

# Calculate the prediction for this example

f\_wb = sigmoid(z\_wb)

# Apply the threshold

if f\_wb >= 0.5:

p[i] = 1

else:

p[i] = 0

### END CODE HERE ###

return p

m, n = X.shape

dj\_db, dj\_dw = compute\_gradient(X, y, w, b)

### START CODE HERE ###

grad\_reg =0

for j in range(0,n):

grad\_reg\_j = (1/m)\*lambda\_\*w[j]

dj\_dw[j] += grad\_reg\_j

### END CODE HERE ###

return dj\_db, dj\_dw

week 2 – lab

model = Sequential(

[

tf.keras.Input(shape=(400,)), #specify input size

### START CODE HERE ###

tf.keras.Sequential(

[

Dense(25,activation='sigmoid'),

Dense(15,activation='sigmoid'),

Dense(1,activation='sigmoid'),

])

### END CODE HERE ###

], name = "my\_model"

### START CODE HERE ###

for j in range(units):

w=W[:,j]

z=np.dot(w,a\_in)+b[j]

a\_out[j] = g(z)

### END CODE HERE ###

return(a\_out)

### START CODE HERE ###

z=np.matmul(A\_in,W) + b

A\_out=g(z)

### START CODE HERE ###

n= z.shape[0]

a = np.ndarray(n)

#calculate denominator

totalexp1 = 0

for j in range(0,n):

totalexp1 += np.exp(z[j])

for i in range(0,n):

a[i]=np.exp(z[i])/totalexp1

### END CODE HERE ###

return a

# GRADED CELL: Sequential model

tf.random.set\_seed(1234) # for consistent results

model = Sequential(

[

### START CODE HERE ###

Dense(25,activation='relu'),

Dense(15,activation='relu'),

Dense(10,activation='linear')

### END CODE HERE ###

], name = "my\_model"

)

ef eval\_mse(y, yhat):

"""

Calculate the mean squared error on a data set.

Args:

y : (ndarray Shape (m,) or (m,1)) target value of each example

yhat : (ndarray Shape (m,) or (m,1)) predicted value of each example

Returns:

err: (scalar)

"""

m = len(y)

err = 0.0

for i in range(m):

### START CODE HERE ###

err += (yhat[i] - y[i])\*\*2

err = err/(2\*m)

### END CODE HERE ###

return(err)

# UNQ\_C2

# GRADED CELL: eval\_cat\_err

def eval\_cat\_err(y, yhat):

"""

Calculate the categorization error

Args:

y : (ndarray Shape (m,) or (m,1)) target value of each example

yhat : (ndarray Shape (m,) or (m,1)) predicted value of each example

Returns:|

cerr: (scalar)

"""

m = len(y)

incorrect = 0

for i in range(m):

### START CODE HERE ###

if yhat[i] != y[i]:

incorrect = incorrect + 1

cerr = incorrect/m

### END CODE HERE ###

return(cerr)

# UNQ\_C3

# GRADED CELL: model

import logging

logging.getLogger("tensorflow").setLevel(logging.ERROR)

tf.random.set\_seed(1234)

model = Sequential(

[

### START CODE HERE ###

Dense(120,activation='relu'),

Dense(40,activation='relu'),

Dense(6,activation='linear'),

### END CODE HERE ###

], name="Complex"

)

model.compile(

### START CODE HERE ###

loss=SparseCategoricalCrossentropy(from\_logits=True),

optimizer=Adam(learning\_rate=1e-2),

### END CODE HERE ###

)

tf.random.set\_seed(1234)

model\_s = Sequential(

[

### START CODE HERE ###

Dense(6,activation='relu'),

Dense(6,activation='linear'),

### END CODE HERE ###

], name = "Simple"

)

model\_s.compile(

### START CODE HERE ###

loss=SparseCategoricalCrossentropy(SparseCategoricalCrossentropy),

optimizer=Adam(learning\_rate=1e-2),

### START CODE HERE ###

# UNQ\_C5

# GRADED CELL: model\_r

tf.random.set\_seed(1234)

model\_r = Sequential(

[

### START CODE HERE ###

Dense(120,activation='relu',kernel\_regularizer=tf.keras.regularizers.l2(0.1)),

Dense(40,activation='relu',kernel\_regularizer=tf.keras.regularizers.l2(0.1)),

Dense(6,activation='linear'),

### START CODE HERE ###

], name= None

)

model\_r.compile(

### START CODE HERE ###

loss=SparseCategoricalCrossentropy(from\_logits=True),

optimizer=Adam(learning\_rate=1e-2),

### START CODE HERE ###

)

def compute\_entropy(y):

"""

Computes the entropy for

Args:

y (ndarray): Numpy array indicating whether each example at a node is

edible (`1`) or poisonous (`0`)

Returns:

entropy (float): Entropy at that node

"""

# You need to return the following variables correctly

entropy = 0.

### START CODE HERE ###

n=len(y)

if n != 0:

p1=0.

for i in range(0,n):

p1 = p1 + y[i]

p1 = p1/n

if p1 == 0 or p1 == 1:

entropy =0.

else:

entropy = -p1 \*np.log2(p1) - (1-p1)\*np.log2(1-p1)

else:

entropy=0.

### END CODE HERE ###

return entropy

# UNQ\_C2

# GRADED FUNCTION: split\_dataset

def split\_dataset(X, node\_indices, feature):

"""

Splits the data at the given node into

left and right branches

Args:

X (ndarray): Data matrix of shape(n\_samples, n\_features)

node\_indices (list): List containing the active indices. I.e, the samples being considered at this step.

feature (int): Index of feature to split on

Returns:

left\_indices (list): Indices with feature value == 1

right\_indices (list): Indices with feature value == 0

"""

# You need to return the following variables correctly

left\_indices = []

right\_indices = []

### START CODE HERE ###

n=len(node\_indices)

for i in range(n):

if X[node\_indices[i]][feature] == 1:

left\_indices.append(node\_indices[i])

else:

right\_indices.append(node\_indices[i])

### END CODE HERE ###

return left\_indices, right\_indice

# UNQ\_C3

# GRADED FUNCTION: compute\_information\_gain

def compute\_information\_gain(X, y, node\_indices, feature):

"""

Compute the information of splitting the node on a given feature

Args:

X (ndarray): Data matrix of shape(n\_samples, n\_features)

y (array like): list or ndarray with n\_samples containing the target variable

node\_indices (ndarray): List containing the active indices. I.e, the samples being considered in this step.

Returns:

cost (float): Cost computed

"""

# Split dataset

left\_indices, right\_indices = split\_dataset(X, node\_indices, feature)

# Some useful variables

X\_node, y\_node = X[node\_indices], y[node\_indices]

X\_left, y\_left = X[left\_indices], y[left\_indices]

X\_right, y\_right = X[right\_indices], y[right\_indices]

# You need to return the following variables correctly

information\_gain = 0

### START CODE HERE ###

SamplesinStep = len(node\_indices)

LeftSamples= len(left\_indices)

RightSamples= len(right\_indices)

TargetvariablesinLeft = np.empty(LeftSamples)

for i in range (LeftSamples):

value2= left\_indices[i]

TargetvariablesinLeft[i]=y[value2]

TargetvariablesinRight = np.empty(RightSamples)

for i in range (RightSamples):

value1 =right\_indices[i]

TargetvariablesinRight[i]=(y[value1])

TargetvariablesinSample = np.empty(SamplesinStep)

for i in range (SamplesinStep):

TargetvariablesinSample[i]=y[node\_indices[i]]

information\_gain = compute\_entropy(TargetvariablesinSample) - ( RightSamples/SamplesinStep\*compute\_entropy(TargetvariablesinRight) + LeftSamples/SamplesinStep\*compute\_entropy(TargetvariablesinLeft))

### END CODE HERE ###

return information\_gain

# UNQ\_C4

# GRADED FUNCTION: get\_best\_split

def get\_best\_split(X, y, node\_indices):

"""

Returns the optimal feature and threshold value

to split the node data

Args:

X (ndarray): Data matrix of shape(n\_samples, n\_features)

y (array like): list or ndarray with n\_samples containing the target variable

node\_indices (ndarray): List containing the active indices. I.e, the samples being considered in this step.

Returns:

best\_feature (int): The index of the best feature to split

"""

# Some useful variables

num\_features = X.shape[1]

# You need to return the following variables correctly

best\_feature = -1

### START CODE HERE ###

IGlist= np.empty(num\_features)

MaxIG =0.

for i in range(num\_features):

IGlist[i]=compute\_information\_gain(X,y,node\_indices,i)

if IGlist[i] > MaxIG:

best\_feature=i

MaxIG =IGlist[i]

### END CODE HERE ##

return best\_feature

# UNQ\_C1

# GRADED FUNCTION: find\_closest\_centroids

def find\_closest\_centroids(X, centroids):

"""

Computes the centroid memberships for every example

Args:

X (ndarray): (m, n) Input values

centroids (ndarray): k centroids

Returns:

idx (array\_like): (m,) closest centroids

"""

# Set K

K = centroids.shape[0]

# You need to return the following variables correctly

idx = np.zeros(X.shape[0], dtype=int)

### START CODE HERE ###

m = X.shape[0]

n = X.shape[1]

#for every point

for j in range (m):

# for every centroid

distance = 0

IndexCent = 0

distance = []

for i in range(K):

distanceP = np.linalg.norm(X[j] - centroids[i])

distance.append(distanceP)

idx[j] = np.argmin(distance)

### END CODE HERE ###

return idx

UNQ\_C2

# GRADED FUNCTION: compute\_centpods

def compute\_centroids(X, idx, K):

"""

Returns the new centroids by computing the means of the

data points assigned to each centroid.

Args:

X (ndarray): (m, n) Data points

idx (ndarray): (m,) Array containing index of closest centroid for each

example in X. Concretely, idx[i] contains the index of

the centroid closest to example i

K (int): number of centroids

Returns:

centroids (ndarray): (K, n) New centroids computed

"""

# Useful variables

m, n = X.shape

# You need to return the following variables correctly

centroids = np.zeros((K, n))

### START CODE HERE ###

for i in range(K):

NewCen = np.zeros(n)

elementsCen = 0

for j in range(m):

if idx[j]== i:

NewCen = NewCen + X[j]

elementsCen += 1

NewCen = 1/elementsCen\*NewCen

centroids[i] = NewCen

### END CODE HERE ##

return centroids

# UNQ\_C1

# GRADED FUNCTION: estimate\_gaussian

def estimate\_gaussian(X):

"""

Calculates mean and variance of all features

in the dataset

Args:

X (ndarray): (m, n) Data matrix

Returns:

mu (ndarray): (n,) Mean of all features

var (ndarray): (n,) Variance of all features

"""

m, n = X.shape

### START CODE HERE ###

mu= np.zeros(n)

var = np.zeros(n)

#start looping through all the features

for i in range(n):

#loop through all the samples to calculate mu\_i

for j in range(m):

mu[i] = mu[i] + X[j][i]

mu[i] = 1/m\*mu[i]

#loop again to calculate variance

for j in range(m):

var[i] = var [i] + (X[j][i] - mu[i])\*\*2

var[i] = 1/m\*var[i]

### END CODE HERE ###

return mu, var

ef select\_threshold(y\_val, p\_val):

"""

Finds the best threshold to use for selecting outliers

based on the results from a validation set (p\_val)

and the ground truth (y\_val)

Args:

y\_val (ndarray): Ground truth on validation set

p\_val (ndarray): Results on validation set

Returns:

epsilon (float): Threshold chosen

F1 (float): F1 score by choosing epsilon as threshold

"""

best\_epsilon = 0

best\_F1 = 0

F1 = 0

step\_size = (max(p\_val) - min(p\_val)) / 1000

for epsilon in np.arange(min(p\_val), max(p\_val), step\_size):

### START CODE HERE ###

n = p\_val.shape[0]

tp =0

fn =0

fp=0

prec = 0

rec = 0

for i in range(n):

prob = p\_val[i]

#is current element an anomaly based on current epsilon?

if prob > epsilon:

#compared to ground truth?

if y\_val[i] == 1:

fn += 1

if prob < epsilon:

#compared to ground truth?

if y\_val[i] == 1:

tp += 1

else:

fp += 1

if(tp+fp)>0:

prec = tp/(tp+fp)

if (tp+fn)>0:

rec = tp/(tp+fn)

if (prec+rec)>0:

F1 = 2\* prec \*rec/(prec+rec)

### END CODE HERE ###

if F1 > best\_F1:

best\_F1 = F1

best\_epsilon = epsilon

return best\_epsilon, best\_F1

# GRADED FUNCTION: cofi\_cost\_func

# UNQ\_C1

def cofi\_cost\_func(X, W, b, Y, R, lambda\_):

"""

Returns the cost for the content-based filtering

Args:

X (ndarray (num\_movies,num\_features)): matrix of item features

W (ndarray (num\_users,num\_features)) : matrix of user parameters

b (ndarray (1, num\_users) : vector of user parameters

Y (ndarray (num\_movies,num\_users) : matrix of user ratings of movies

R (ndarray (num\_movies,num\_users) : matrix, where R(i, j) = 1 if the i-th movies was rated by the j-th user

lambda\_ (float): regularization parameter

Returns:

J (float) : Cost

"""

nm, nu = Y.shape

J = 0.

### START CODE HERE ###

#first without regularization

for j in range(nu):

w = W[j,:]

b\_j = b[0,j]

for i in range(nm):

x = X[i,:]

y = Y[i][j]

if R[i][j] == 1:

J += (np.dot(w,x) + b\_j - y)\*\*2

J = 0.5\*J

### END CODE HERE ###

return J

# GRADED FUNCTION: cofi\_cost\_func

# UNQ\_C1

def cofi\_cost\_func(X, W, b, Y, R, lambda\_):

"""

Returns the cost for the content-based filtering

Args:

X (ndarray (num\_movies,num\_features)): matrix of item features

W (ndarray (num\_users,num\_features)) : matrix of user parameters

b (ndarray (1, num\_users) : vector of user parameters

Y (ndarray (num\_movies,num\_users) : matrix of user ratings of movies

R (ndarray (num\_movies,num\_users) : matrix, where R(i, j) = 1 if the i-th movies was rated by the j-th user

lambda\_ (float): regularization parameter

Returns:

J (float) : Cost

"""

nm, nu = Y.shape

J = 0.

### START CODE HERE ###

#first without regularization

for j in range(nu):

w = W[j,:]

b\_j = b[0,j]

for i in range(nm):

x = X[i,:]

y = Y[i][j]

if R[i][j] == 1:

J += (np.dot(w,x) + b\_j - y)\*\*2

J = 0.5\*J

reg = 0.

n = X.shape[1]

# adding regularitation parameter 1

for j in range(nu):

for k in range(n):

reg += W[j][k]\*\*2

# adding regularitation parameter 2

for i in range(nm):

for k in range(n):

reg += X[i][k]\*\*2

reg= reg\*lambda\_/2

J+=reg

### END CODE HERE ###

return J

# GRADED\_CELL

# UNQ\_C1

num\_outputs = 32

tf.random.set\_seed(1)

user\_NN = tf.keras.models.Sequential([

### START CODE HERE ###

tf.keras.layers.Dense(256,activation='relu'),

tf.keras.layers.Dense(128,activation='relu'),

tf.keras.layers.Dense(num\_outputs,activation='linear')

### END CODE HERE ###

])

item\_NN = tf.keras.models.Sequential([

### START CODE HERE ###

tf.keras.layers.Dense(256,activation='relu'),

tf.keras.layers.Dense(128,activation='relu'),

tf.keras.layers.Dense(num\_outputs,activation='linear')

### END CODE HERE ###

])

# create the user input and point to the base network

input\_user = tf.keras.layers.Input(shape=(num\_user\_features))

vu = user\_NN(input\_user)

vu = tf.linalg.l2\_normalize(vu, axis=1)

# create the item input and point to the base network

input\_item = tf.keras.layers.Input(shape=(num\_item\_features))

vm = item\_NN(input\_item)

vm = tf.linalg.l2\_normalize(vm, axis=1)

# compute the dot product of the two vectors vu and vm

output = tf.keras.layers.Dot(axes=1)([vu, vm])

# specify the inputs and output of the model

model = tf.keras.Model([input\_user, input\_item], output)

model.summary()

# GRADED\_FUNCTION: sq\_dist

# UNQ\_C2

def sq\_dist(a,b):

"""

Returns the squared distance between two vectors

Args:

a (ndarray (n,)): vector with n features

b (ndarray (n,)): vector with n features

Returns:

d (float) : distance

"""

### START CODE HERE ###

d=0.

n=a.shape[0]

for i in range(n):

d += (a[i]-b[i])\*\*2

### END CODE HERE ###

return d

# UNQ\_C1

# GRADED CELL

# Create the Q-Network.

q\_network = Sequential([

### START CODE HERE ###

Input(state\_size),

Dense(64,activation='relu'),

Dense(64,activation='relu'),

Dense(num\_actions,activation='linear')

### END CODE HERE ###

])

# Create the target Q^-Network.

target\_q\_network = Sequential([

### START CODE HERE ###

Input(state\_size),

Dense(64,activation='relu'),

Dense(64,activation='relu'),

Dense(num\_actions,activation='linear')

### END CODE HERE ###

])

### START CODE HERE ###

optimizer=Adam(learning\_rate=ALPHA)

### END CODE HERE ###

UNQ\_C2

# GRADED FUNCTION: calculate\_loss

def compute\_loss(experiences, gamma, q\_network, target\_q\_network):

"""

Calculates the loss.

Args:

experiences: (tuple) tuple of ["state", "action", "reward", "next\_state", "done"] namedtuples

gamma: (float) The discount factor.

q\_network: (tf.keras.Sequential) Keras model for predicting the q\_values

target\_q\_network: (tf.keras.Sequential) Keras model for predicting the targets

Returns:

loss: (TensorFlow Tensor(shape=(0,), dtype=int32)) the Mean-Squared Error between

the y targets and the Q(s,a) values.

"""

# Unpack the mini-batch of experience tuples.

states, actions, rewards, next\_states, done\_vals = experiences

# Compute max Q^(s,a).

max\_qsa = tf.reduce\_max(target\_q\_network(next\_states), axis=-1)

# Set y = R if episode terminates, otherwise set y = R + γ max Q^(s,a).

### START CODE HERE ###

y\_targets = rewards + gamma\*max\_qsa\*(1 - done\_vals)

### END CODE HERE ###

# Get the q\_values.

q\_values = q\_network(states)

q\_values = tf.gather\_nd(q\_values, tf.stack([tf.range(q\_values.shape[0]),

tf.cast(actions, tf.int32)], axis=1))

# Compute the loss.

### START CODE HERE ###

loss = MSE(y\_targets,q\_values)

### END CODE HERE ###

return loss