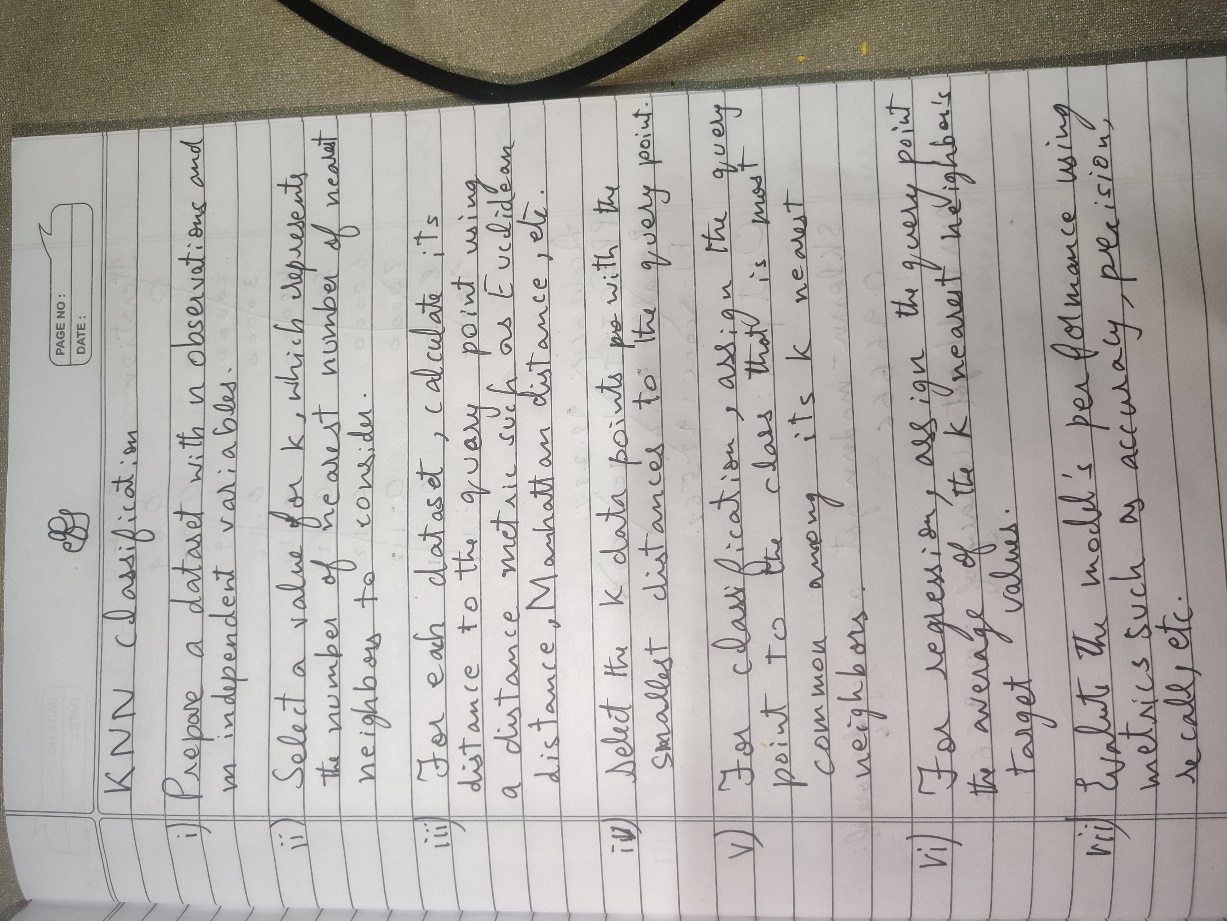
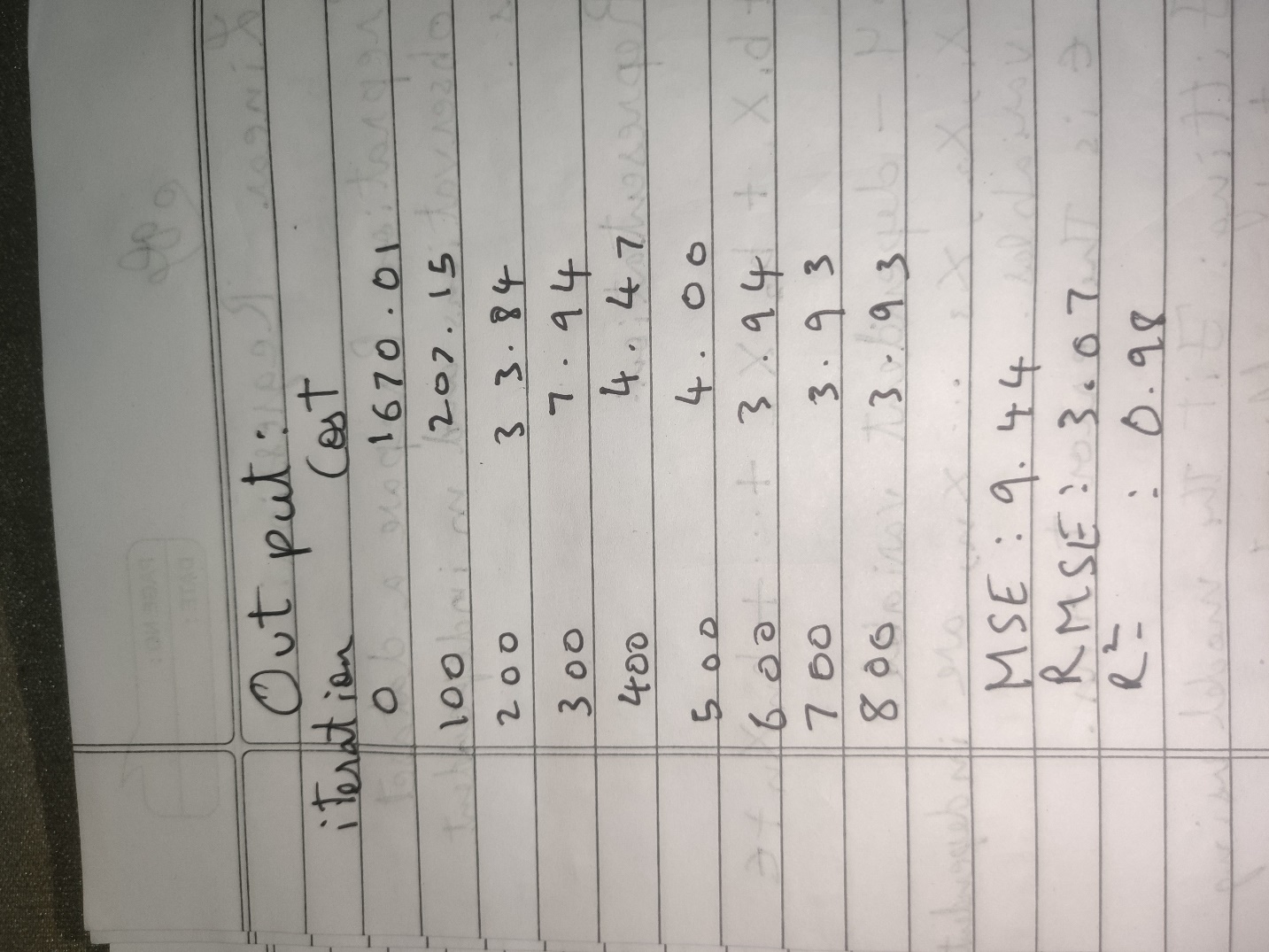
**LAB-5**

**Build Logistic Regression Model for a given dataset.**

**OBSERVATION:**





**CODE:**

import math

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

import plotly.express as px

import pprint

import pickle

In [4]:

df = pd.read\_csv('breast-cancer.csv')

In [5]:

df.head()

In [6]:

df.drop('id', axis=1, inplace=True) *#drop redundant columns*

In [7]:

df['diagnosis'] = (df['diagnosis'] == 'M').astype(int) *#encode the label into 1/0*

In [8]:

corr = df.corr()

In [9]:

plt.figure(figsize=(20,20))

sns.heatmap(corr, cmap='mako\_r',annot=True)

plt.show()

In [12]:

*# Get the absolute value of the correlation*

cor\_target = abs(corr["diagnosis"])

*# Select highly correlated features (thresold = 0.2)*

relevant\_features = cor\_target[cor\_target>0.2]

*# Collect the names of the features*

names = [index for index, value in relevant\_features.items()]

*# Drop the target variable from the results*

names.remove('diagnosis')

*# Display the results*

pprint.pprint(names)

['radius\_mean',

'texture\_mean',

'perimeter\_mean',

'area\_mean',

'smoothness\_mean',

'compactness\_mean',

'concavity\_mean',

'concave points\_mean',

'symmetry\_mean',

'radius\_se',

'perimeter\_se',

'area\_se',

'compactness\_se',

'concavity\_se',

'concave points\_se',

'radius\_worst',

'texture\_worst',

'perimeter\_worst',

'area\_worst',

'smoothness\_worst',

'compactness\_worst',

'concavity\_worst',

'concave points\_worst',

'symmetry\_worst',

'fractal\_dimension\_worst']

In [13]:

X = df[names].values

y = df['diagnosis'].values

In [14]:

def train\_test\_split(X, y, random\_state=42, test\_size=0.2):

"""

Splits the data into training and testing sets.

Parameters:

X (numpy.ndarray): Features array of shape (n\_samples, n\_features).

y (numpy.ndarray): Target array of shape (n\_samples,).

random\_state (int): Seed for the random number generator. Default is 42.

test\_size (float): Proportion of samples to include in the test set. Default is 0.2.

Returns:

Tuple[numpy.ndarray]: A tuple containing X\_train, X\_test, y\_train, y\_test.

"""

*# Get number of samples*

n\_samples = X.shape[0]

*# Set the seed for the random number generator*

np.random.seed(random\_state)

*# Shuffle the indices*

shuffled\_indices = np.random.permutation(np.arange(n\_samples))

*# Determine the size of the test set*

test\_size = int(n\_samples \* test\_size)

*# Split the indices into test and train*

test\_indices = shuffled\_indices[:test\_size]

train\_indices = shuffled\_indices[test\_size:]

*# Split the features and target arrays into test and train*

X\_train, X\_test = X[train\_indices], X[test\_indices]

y\_train, y\_test = y[train\_indices], y[test\_indices]

return X\_train, X\_test, y\_train, y\_test

In [15]:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y)

In [16]:

def standardize\_data(X\_train, X\_test):

"""

Standardizes the input data using mean and standard deviation.

Parameters:

X\_train (numpy.ndarray): Training data.

X\_test (numpy.ndarray): Testing data.

Returns:

Tuple of standardized training and testing data.

"""

*# Calculate the mean and standard deviation using the training data*

mean = np.mean(X\_train, axis=0)

std = np.std(X\_train, axis=0)

*# Standardize the data*

X\_train = (X\_train - mean) / std

X\_test = (X\_test - mean) / std

return X\_train, X\_test

X\_train, X\_test = standardize\_data(X\_train, X\_test)

In [17]:

def sigmoid(z):

"""

Compute the sigmoid function for a given input.

The sigmoid function is a mathematical function used in logistic regression and neural networks

to map any real-valued number to a value between 0 and 1.

Parameters:

z (float or numpy.ndarray): The input value(s) for which to compute the sigmoid.

Returns:

float or numpy.ndarray: The sigmoid of the input value(s).

Example:

>>> sigmoid(0)

0.5

"""

*# Compute the sigmoid function using the formula: 1 / (1 + e^(-z)).*

sigmoid\_result = 1 / (1 + np.exp(-z))

*# Return the computed sigmoid value.*

return sigmoid\_result

In [18]:

z = np.linspace(-12, 12, 200)

fig = px.line(x=z, y=sigmoid(z),title='Logistic Function',template="plotly\_dark")

fig.update\_layout(

title\_font\_color="#41BEE9",

xaxis=dict(color="#41BEE9"),

yaxis=dict(color="#41BEE9")

)

fig.show()

In [19]:

class LogisticRegression:

"""

Logistic Regression model.

Parameters:

learning\_rate (float): Learning rate for the model.

Methods:

initialize\_parameter(): Initializes the parameters of the model.

sigmoid(z): Computes the sigmoid activation function for given input z.

forward(X): Computes forward propagation for given input X.

compute\_cost(predictions): Computes the cost function for given predictions.

compute\_gradient(predictions): Computes the gradients for the model using given predictions.

fit(X, y, iterations, plot\_cost): Trains the model on given input X and labels y for specified iterations.

predict(X): Predicts the labels for given input X.

"""

def \_\_init\_\_(self, learning\_rate=0.0001):

np.random.seed(1)

self.learning\_rate = learning\_rate

def initialize\_parameter(self):

"""

Initializes the parameters of the model.

"""

self.W = np.zeros(self.X.shape[1])

self.b = 0.0

def forward(self, X):

"""

Computes forward propagation for given input X.

Parameters:

X (numpy.ndarray): Input array.

Returns:

numpy.ndarray: Output array.

"""

*# print(X.shape, self.W.shape)*

Z = np.matmul(X, self.W) + self.b

A = sigmoid(Z)

return A

def compute\_cost(self, predictions):

"""

Computes the cost function for given predictions.

Parameters:

predictions (numpy.ndarray): Predictions of the model.

Returns:

float: Cost of the model.

"""

m = self.X.shape[0] *# number of training examples*

*# compute the cost*

cost = np.sum((-np.log(predictions + 1e-8) \* self.y) + (-np.log(1 - predictions + 1e-8)) \* (

1 - self.y)) *# we are adding small value epsilon to avoid log of 0*

cost = cost / m

return cost

def compute\_gradient(self, predictions):

"""

Computes the gradients for the model using given predictions.

Parameters:

predictions (numpy.ndarray): Predictions of the model.

"""

*# get training shape*

m = self.X.shape[0]

*# compute gradients*

self.dW = np.matmul(self.X.T, (predictions - self.y))

self.dW = np.array([np.mean(grad) for grad in self.dW])

self.db = np.sum(np.subtract(predictions, self.y))

*# scale gradients*

self.dW = self.dW \* 1 / m

self.db = self.db \* 1 / m

def fit(self, X, y, iterations, plot\_cost=True):

"""

Trains the model on given input X and labels y for specified iterations.

Parameters:

X (numpy.ndarray): Input features array of shape (n\_samples, n )

y (numpy.ndarray): Labels array of shape (n\_samples, 1)

iterations (int): Number of iterations for training.

plot\_cost (bool): Whether to plot cost over iterations or not.

Returns:

None.

"""

self.X = X

self.y = y

self.initialize\_parameter()

costs = []

for i in range(iterations):

*# forward propagation*

predictions = self.forward(self.X)

*# compute cost*

cost = self.compute\_cost(predictions)

costs.append(cost)

*# compute gradients*

self.compute\_gradient(predictions)

*# update parameters*

self.W = self.W - self.learning\_rate \* self.dW

self.b = self.b - self.learning\_rate \* self.db

*# print cost every 100 iterations*

if i % 10000 == 0:

print("Cost after iteration {}: {}".format(i, cost))

if plot\_cost:

fig = px.line(y=costs,title="Cost vs Iteration",template="plotly\_dark")

fig.update\_layout(

title\_font\_color="#41BEE9",

xaxis=dict(color="#41BEE9",title="Iterations"),

yaxis=dict(color="#41BEE9",title="cost")

)

fig.show()

def predict(self, X):

"""

Predicts the labels for given input X.

Parameters:

X (numpy.ndarray): Input features array.

Returns:

numpy.ndarray: Predicted labels.

"""

predictions = self.forward(X)

return np.round(predictions)

def save\_model(self, filename=None):

"""

Save the trained model to a file using pickle.

Parameters:

filename (str): The name of the file to save the model to.

"""

model\_data = {

'learning\_rate': self.learning\_rate,

'W': self.W,

'b': self.b

}

with open(filename, 'wb') as file:

pickle.dump(model\_data, file)

@classmethod

def load\_model(cls, filename):

"""

Load a trained model from a file using pickle.

Parameters:

filename (str): The name of the file to load the model from.

Returns:

LogisticRegression: An instance of the LogisticRegression class with loaded parameters.

"""

with open(filename, 'rb') as file:

model\_data = pickle.load(file)

*# Create a new instance of the class and initialize it with the loaded parameters*

loaded\_model = cls(model\_data['learning\_rate'])

loaded\_model.W = model\_data['W']

loaded\_model.b = model\_data['b']

return loaded\_model

In [21]:

lg = LogisticRegression()

lg.fit(X\_train, y\_train,100000)

In [22]:

lg.save\_model("model.pkl")

In [23]:

class ClassificationMetrics:

@staticmethod

def accuracy(y\_true, y\_pred):

"""

Computes the accuracy of a classification model.

Parameters:

y\_true (numpy array): A numpy array of true labels for each data point.

y\_pred (numpy array): A numpy array of predicted labels for each data point.

Returns:

float: The accuracy of the model, expressed as a percentage.

"""

y\_true = y\_true.flatten()

total\_samples = len(y\_true)

correct\_predictions = np.sum(y\_true == y\_pred)

return (correct\_predictions / total\_samples)

@staticmethod

def precision(y\_true, y\_pred):

"""

Computes the precision of a classification model.

Parameters:

y\_true (numpy array): A numpy array of true labels for each data point.

y\_pred (numpy array): A numpy array of predicted labels for each data point.

Returns:

float: The precision of the model, which measures the proportion of true positive predictions

out of all positive predictions made by the model.

"""

true\_positives = np.sum((y\_true == 1) & (y\_pred == 1))

false\_positives = np.sum((y\_true == 0) & (y\_pred == 1))

return true\_positives / (true\_positives + false\_positives)

@staticmethod

def recall(y\_true, y\_pred):

"""

Computes the recall (sensitivity) of a classification model.

Parameters:

y\_true (numpy array): A numpy array of true labels for each data point.

y\_pred (numpy array): A numpy array of predicted labels for each data point.

Returns:

float: The recall of the model, which measures the proportion of true positive predictions

out of all actual positive instances in the dataset.

"""

true\_positives = np.sum((y\_true == 1) & (y\_pred == 1))

false\_negatives = np.sum((y\_true == 1) & (y\_pred == 0))

return true\_positives / (true\_positives + false\_negatives)

@staticmethod

def f1\_score(y\_true, y\_pred):

"""

Computes the F1-score of a classification model.

Parameters:

y\_true (numpy array): A numpy array of true labels for each data point.

y\_pred (numpy array): A numpy array of predicted labels for each data point.

Returns:

float: The F1-score of the model, which is the harmonic mean of precision and recall.

"""

precision\_value = ClassificationMetrics.precision(y\_true, y\_pred)

recall\_value = ClassificationMetrics.recall(y\_true, y\_pred)

return 2 \* (precision\_value \* recall\_value) / (precision\_value + recall\_value)

In [24]:

model = LogisticRegression.load\_model("model.pkl")

In [25]:

y\_pred = model.predict(X\_test)

accuracy = ClassificationMetrics.accuracy(y\_test, y\_pred)

precision = ClassificationMetrics.precision(y\_test, y\_pred)

recall = ClassificationMetrics.recall(y\_test, y\_pred)

f1\_score = ClassificationMetrics.f1\_score(y\_test, y\_pred)

print(f"Accuracy: {accuracy:.2%}")

print(f"Precision: {precision:.2%}")

print(f"Recall: {recall:.2%}")

print(f"F1-Score: {f1\_score:.2%}")

**OUTPUT:**

