VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Machine Learning (23CS6PCMAL)

Submitted by

Rishabh Kumar (1BM22CS221)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2024 to Jan-2025

B.M.S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Machine Learning (23CS6PCMAL)" carried out by **Rishabh Kumar (1BM22CS221),** who is a bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Machine Learning (23CS6PCMAL) work prescribed for the said degree.

Lab Faculty Incharge Name: **Ms. Saritha A N** Assistant Professor Department of CSE, BMSCE	**Dr. Kavitha Sooda** Professor & HOD Department of CSE, BMSCE

Index

Github Link: **https://github.com/rishabh-agr/ML-Lab**

Program 1

Write a python program to import and export data using Pandas library functions

Code:

```
import pandas as pd


# Import data

df = pd.read_csv('data.csv')


# Export data

df.to_csv('processed_data.csv', index=False)
```

```
# Loading Data from sklearn

import pandas as pd
from sklearn.dataset import load_iris
iris = load_iris()

df = pd.DataFrame (iris.data)
df.head()
```

```
import pandas as pd

df = pd.read_csv ('sales_data.csv')
print ( df.head ())

regional = df.groupby ('Region') ['Sales'].sum()
print (regional)
```

Program 2

Demonstrate various data pre-processing techniques for a given dataset

Code:

```
# Handle missing values

df.fillna(df.mean(), inplace=True)



# Encode categorical variables using one-hot encoding

df = pd.get_dummies(df, columns=['category_column'])



# Normalize numerical features

df['normalized'] = (df['numeric_column'] - df['numeric_column'].min()) / (df['numeric_column'].max() -
df['numeric_column'].min())



# Standardize numerical features

df['standardized'] = (df['numeric_column'] - df['numeric_column'].mean()) / df['numeric_column'].std()



# Feature scaling using Min-Max

df['scaled'] = (df['numeric_column'] - df['numeric_column'].min()) / (df['numeric_column'].max() -
df['numeric_column'].min())
```

Handling "Housing Dataset"

```
import pandas as pd

df = pd.read_csv('housing.csv')

print(" All Columns" )
print( df.info() )

print(" Descriptive Analysis")
print( df.describe())

print ("Count of unique labels")

print(df['Ocean Proximity'].value_counts()]

print("Column with missing value")
missing_values = df.isnull().sum()

column_missing = missing_values [missing_values >0]

print(columns_with_missing_value)
```

Program 3

Use an appropriate data set for building the decision tree (ID3) and apply this knowledge to classify a new sample.

Code:

```
import numpy as np


# Prepare data

X = df[['feature1', 'feature2']].values

y = df['target'].values.reshape(-1, 1)


# Add intercept term

X_b = np.c_[np.ones((X.shape[0], 1)), X]


# Compute theta using Normal Equation

theta = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)


# Make predictions

predictions = X_b.dot(theta)
```

ID 3 Algorithm (Play football or ...

```
import math
import pandas as pd

def entropy(data):
    counts = data.value_counts()
    prob = counts / len(data)
    return -sum(prob * prob.apply
        (lambda p: math.log2(p)
            if p>0 else 0))

def info_gain (data, feature, target):
    total_entropy = entropy (data[target])
    values = data[feature].unique()

    weighted_entropy = 0
    for value in values:
        subset = data[data[feature]==value]
        weighted_entropy += (len(subset)/
                             len(data))
                   * entropy (subset[target])

    return total_entropy - weighted_entropy
```

Program 4

Implement Linear and Multi-Linear Regression algorithm using appropriate dataset

Code:

```
import numpy as np


# Sigmoid function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))


# Prepare data
X = df[['feature1', 'feature2']].values
y = df['target'].values.reshape(-1, 1)


# Add intercept term
X_b = np.c_[np.ones((X.shape[0], 1)), X]


# Initialize parameters
theta = np.zeros((X_b.shape[1], 1))
learning_rate = 0.01
iterations = 1000


# Gradient Descent
for i in range(iterations):
    gradients = X_b.T.dot(sigmoid(X_b.dot(theta)) - y) / X_b.shape[0]
    theta -= learning_rate * gradients
```

Make predictions

predictions = sigmoid(X_b.dot(theta)) >= 0.5

Linear Regression

```
import numpy as np
import pandas as pd
from sklearn.datasets import make_Regression

X, y = make_

def fit (x , y):
    n = len (x)

    if n != len(y)
        raise ValueError (" length must be Same")

    mean_x = sum(X) /n
    mean_y = sum (y) /n

    num = sum ((x[i] - mean_x) * (y[i] - mean_y)
                for i in range(n))

    den = sum( x[i] - mean_x) ** 2 for i in
                range(n))

    slope = num / den
    intercept = mean_y - (slope * mean_x)

    return slope , intercept
```

Program 5

Build Logistic Regression Model for a given dataset

Code:

```
import numpy as np


# Sigmoid function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))


# Prepare data
X = df[['feature1', 'feature2']].values
y = df['target'].values.reshape(-1, 1)


# Add intercept term
X_b = np.c_[np.ones((X.shape[0], 1)), X]


# Initialize parameters
theta = np.zeros((X_b.shape[1], 1))
learning_rate = 0.01
iterations = 1000


# Gradient Descent
for i in range(iterations):
    gradients = X_b.T.dot(sigmoid(X_b.dot(theta)) - y) / X_b.shape[0]
    theta -= learning_rate * gradients
```

Make predictions

predictions = sigmoid(X_b.dot(theta)) >= 0.5

Logistics Regression

```
import numpy as np

def sigmoid(z):
    ~~return~~ 1 / (1 + np.exp(-z))

def fit(x, y, learning_rate = 0.01, iter = 1000):

    n_samples, n_features = X.shape
    weights = np.zeros(n_features)
    bias = 0

    for _ in range(iter):

        linear_model = np.dot(X, weights) +
        y_predicted = sigmoid(linear_model)

        dw = (1 / n_samples) * np.dot(X.T, (y
        db = (1/n_samples) * np.sum (ypre - y

        weights -= learning_rate * dw
        bias -= learning_rate * db

    return weights, bias

if "__name__" == "__main__"
```

Program 6

Build KNN Classification model for a given dataset.

Code:

```
import numpy as np


# Function to calculate entropy

def entropy(y):

    unique, counts = np.unique(y, return_counts=True)

    probabilities = counts / len(y)

    return -np.sum(probabilities * np.log2(probabilities))


# Function to calculate information gain

def information_gain(X_column, y, threshold):

    left_mask = X_column <= threshold

    right_mask = ~left_mask

    left_y, right_y = y[left_mask], y[right_mask]

    return entropy(y) - (len(left_y) / len(y)) * entropy(left_y) - (len(right_y) / len(y)) * entropy(right_y)


# Function to find the best split

def best_split(X, y):

    best_gain = -1

    best_split = None

    for feature_index in range(X.shape[1]):

        thresholds = np.unique(X[:, feature_index])

        for threshold in thresholds:

            gain = information_gain(X[:, feature_index], y, threshold)
```

```
            if gain > best_gain:
                best_gain = gain
                best_split = (feature_index, threshold)
    return best_split

# Function to build the tree
def build_tree(X, y):
    if len(np.unique(y)) == 1:
        return {'label': np.unique(y)[0]}
    feature_index, threshold = best_split(X, y)
    left_mask = X[:, feature_index] <= threshold
    right_mask = ~left_mask
    left_tree = build_tree(X[left_mask], y[left_mask])
    right_tree = build_tree(X[right_mask], y[right_mask])
    return {'feature_index': feature_index, 'threshold': threshold, 'left': left_tree, 'right': right_tree}

# Function to predict using the tree
def predict_tree(tree, X):
    if 'label' in tree:
        return tree['label']
    if X[tree['feature_index']] <= tree['threshold']:
        return predict_tree(tree['left'], X)
    else:
        return predict_tree(tree['right'], X)
```

Prepare data

X = df[['feature1', 'feature2']].values

y = df['target'].values

Build the tree

tree = build_tree(X, y)

Make predictions

predictions = [predict_tree(tree, x) for x in X]

Support Vector Machine :-

```
import numpy as np

class SVM :
    def __init__(self, learning_rate = 0.001) :
        self.lr = learning_rate
        self.lambda_param = lambda_param
        self.n_iters = n_iters
        self.w = None
        self.b = None

    def fit (self, X, y) :
        n_samples, n_features = X.shape
        y_ = np.where (y <= 0, -1, 1)
        self.w = np.zeros(n_features)
        self.b = 0

    def predict (self, X) :
        approx = np.dot (X, self.w) - self.b
        return np.where (approx >= 0, -1, 1)
```

Program 7

Build Support vector machine model for a given dataset

Code:

```
import numpy as np

from collections import Counter


# Euclidean distance function

def euclidean_distance(x1, x2):

    return np.sqrt(np.sum((x1 - x2) ** 2))


# KNN classifier

def knn(X_train, y_train, X_test, k=3):

    predictions = []

    for x_test in X_test:

        distances = [euclidean_distance(x_test, x_train) for x_train in X_train]

        k_indices = np.argsort(distances)[:k]

        k_nearest_labels = [y_train[i] for i in k_indices]

        most_common = Counter(k_nearest_labels).most_common(1)

        predictions.append(most_common[0][0])

    return predictions


# Prepare data

X_train = df[['feature1', 'feature2']].values

y_train = df['target'].values

X_test = np.array([[value1, value2], [value3, value4]])
```

Make predictions

predictions = knn(X_train, y_train, X_test)

ML lab - 5

KNN Algorithm :-

```
import numpy as np
from collections import Counter
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

class KNN:
    def __init__(self, k=3):
        self.k = k

    def fit(self, x, y):
        self.x_train = x
        self.y_train = y

    def predict(self, x):
        predictions = [self.__predict(x) for x in
        return np.array(prediction)

data = load_iris()
x, y = data.data, data.target

x_train, x_test, y_train, y_test = train_test_split()

model = KNN(k=3)
model.fit(x_train, y_train)
```

Program 8

Implement Random forest ensemble method on a given dataset.

Code:

```
import numpy as np

import random


# Reuse entropy and tree logic from previous Decision Tree implementation

def entropy(y):

    unique, counts = np.unique(y, return_counts=True)

    probs = counts / len(y)

    return -np.sum(probs * np.log2(probs + 1e-9))


def information_gain(X_col, y, thresh):

    left_mask = X_col <= thresh

    right_mask = ~left_mask

    if len(y[left_mask]) == 0 or len(y[right_mask]) == 0:

        return 0

    return entropy(y) - (len(y[left_mask]) / len(y)) * entropy(y[left_mask]) - (len(y[right_mask]) / len(y)) *
entropy(y[right_mask])


def best_split(X, y):

    best_gain = -1

    split = None

    n_features = X.shape[1]

    for feature_index in random.sample(range(n_features), int(np.sqrt(n_features))):

        for threshold in np.unique(X[:, feature_index]):
```

```
            gain = information_gain(X[:, feature_index], y, threshold)
            if gain > best_gain:
                best_gain = gain
                split = (feature_index, threshold)
    return split


def build_tree(X, y, depth=0, max_depth=5):
    if len(np.unique(y)) == 1 or depth >= max_depth:
        return {'label': np.bincount(y).argmax()}
    feat, thresh = best_split(X, y)
    if feat is None:
        return {'label': np.bincount(y).argmax()}
    left_mask = X[:, feat] <= thresh
    right_mask = ~left_mask
    return {
        'feature': feat,
        'threshold': thresh,
        'left': build_tree(X[left_mask], y[left_mask], depth+1, max_depth),
        'right': build_tree(X[right_mask], y[right_mask], depth+1, max_depth)
    }


def predict_tree(tree, x):
    if 'label' in tree:
        return tree['label']
    if x[tree['feature']] <= tree['threshold']:
```

```
        return predict_tree(tree['left'], x)
    return predict_tree(tree['right'], x)


# Random Forest
class RandomForest:
    def __init__(self, n_trees=5, max_depth=5):
        self.n_trees = n_trees
        self.max_depth = max_depth
        self.trees = []

    def fit(self, X, y):
        self.trees = []
        for _ in range(self.n_trees):
            idxs = np.random.choice(len(X), len(X), replace=True)
            X_sample = X[idxs]
            y_sample = y[idxs]
            tree = build_tree(X_sample, y_sample, max_depth=self.max_depth)
            self.trees.append(tree)

    def predict(self, X):
        tree_preds = np.array([[predict_tree(tree, x) for tree in self.trees] for x in X])
        return [np.bincount(row).argmax() for row in tree_preds]


# Example usage
# X = df[['f1', 'f2']].values
```

y = df['target'].values

model = RandomForest()

model.fit(X, y)

preds = model.predict(X)

Random Forest

```
import numpy as np
from collections import Counter
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

class RandomForest:
    def __init__(self, sample_size = None):
        self.n_estimators = n_estimators
        self.sample_size = sample_size
        self.trees = []

    def predict(self, x):
        tree_preds = np.array([tree.predict(x) for tree
                              in self.trees])
        return np.array([Counter(tree_preds[:, i])


data = load_iris()
X = data.data[data.target != 2]
y = data.target[data.target != 2]

X_train, X_test, y_train, y_test = train_test_split(X,
                                   y, test_size = 0.2,
                                   random_state = 42)
```

Program 9

Implement Boosting ensemble method on a given dataset.

Code:

```
import numpy as np


class AdaBoost:
    def __init__(self, n_clf=5):
        self.n_clf = n_clf


    def fit(self, X, y):
        n = len(X)
        w = np.ones(n) / n
        self.models = []
        self.alphas = []


        for _ in range(self.n_clf):
            stump = self._build_stump(X, y, w)
            preds = stump['pred']
            err = np.sum(w * (preds != y))
            if err == 0:
                break
            alpha = 0.5 * np.log((1 - err) / (err + 1e-10))
            w *= np.exp(-alpha * y * preds)
            w /= np.sum(w)
            self.models.append(stump)
            self.alphas.append(alpha)
```

```
    def _build_stump(self, X, y, w):
        m, n = X.shape
        best_stump = {}
        best_pred = None
        min_error = float('inf')
        for feat in range(n):
            thresholds = np.unique(X[:, feat])
            for t in thresholds:
                for polarity in [1, -1]:
                    pred = np.ones(m)
                    pred[polarity * X[:, feat] < polarity * t] = -1
                    error = np.sum(w * (pred != y))
                    if error < min_error:
                        best_stump = {'feat': feat, 'thresh': t, 'polarity': polarity, 'pred': pred}
                        min_error = error
        return best_stump

    def predict(self, X):
        clf_preds = [alpha * np.where(clf['polarity'] * X[:, clf['feat']] >= clf['polarity'] * clf['thresh'], 1, -1)
                     for clf, alpha in zip(self.models, self.alphas)]
        return np.sign(np.sum(clf_preds, axis=0))

# Example usage
# X = df[['f1', 'f2']].values
```

y = df['target'].values

y = np.where(y == 1, 1, -1) # convert to -1, +1

model = AdaBoost()

model.fit(X, y)

preds = model.predict(X)

Boosting Ensemble Algorithm

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

class AdaBoost :
    def __init__(self, n_estimators = 5) :
        self.n_estimators = n_estimators
        self.alphas = []
        self.stumps = []

    def fit(self, x, y) :

        n_samples, n_features = x.shape
        y = np.where(y == 0, -1, 1)

        for _ in range(self.n_estimators) :
            stump = self.build_stump(x, y, w)
            preds = stump['pred']
            err = np.sum(w * (pred != y))
            if err == 0 :
                break

    def predict(self, x) :
        clf_preds = [alpha * np.where(clf['polarity']
            * x[:, clf['feat']] >= clf['polarity']]
```

Program 10

Build k-Means algorithm to cluster a set of data stored in a .CSV file.

Code:

```
import numpy as np

import pandas as pd


def kmeans(X, k=3, max_iters=100):

    centroids = X[np.random.choice(len(X), k, replace=False)]

    for _ in range(max_iters):

        distances = np.linalg.norm(X[:, np.newaxis] - centroids, axis=2)

        labels = np.argmin(distances, axis=1)

        new_centroids = np.array([X[labels == i].mean(axis=0) for i in range(k)])

        if np.allclose(centroids, new_centroids):

            break

        centroids = new_centroids

    return labels, centroids


# Example

df = pd.read_csv("data.csv")

X = df[['f1', 'f2']].values

labels, centroids = kmeans(X, k=3)
```

K-means Clustering :

```
import numpy as np
import pandas as pd

def kmeans (x, k=3, max_iter = 100):
    centroids = x [np.random.choice (len(x), k)]
    for _ in range (max_iter) :
        dis = np.linalg.norm (x[:, np.newaxis]
                  - cent, axis=2)

    labels = np.argmin (distances , axis=1)
    new_cent = np.array ([x[labels == i].mean
                 for i in range (k)])

    if np.allclose (cent, new_cent) :
        break
    cent = new_cent

    return cent


df = pd.read_csv ("data.csv")

x = df [[f1, f2]] . values

labels , centroids = kmeans ( x, k=3)
```

Program 11

Implement Dimensionality reduction using Principle Component Analysis (PCA) method.

Code:

```
import numpy as np

import pandas as pd


def pca(X, n_components):

    # Standardize

    X_meaned = X - np.mean(X, axis=0)

    cov_mat = np.cov(X_meaned, rowvar=False)

    eigen_vals, eigen_vecs = np.linalg.eigh(cov_mat)

    # Sort eigenvalues

    idxs = np.argsort(eigen_vals)[::-1]

    eigen_vecs = eigen_vecs[:, idxs]

    eigen_vals = eigen_vals[idxs]

    # Select top components

    eigen_vecs = eigen_vecs[:, :n_components]

    return np.dot(X_meaned, eigen_vecs)


# Example

df = pd.read_csv("data.csv")

X = df[['f1', 'f2', 'f3']].values

X_pca = pca(X, n_components=2)
```

Principal Component Analysis

```
import numpy as np
import pandas as pd

def pca (x, n):

    X_mean = x - np.mean (x, axis = 0)
    cov_mat = np.cov (x_mean, rowvar = F)
    eigen_val, eigen_vec = np.linalg.eigh

    idxs = np.argsort (eigen_val) [::-1]

    eigen_vec = eigen_vec [:, idxs]
    eigen_val = eigen_val [idxs]

    return np.dot (x_mean, eigen_vec)


df = pd.read_csv ("data.csv")

x = df [['f1', 'f2', 'f3']].values

x_pca = pca (x, n)
```