**Implement a sample machine learning program for a problem statement of your choice**.

**Program to predict Bitcoin prices using Linear regression**

# Import required libraries

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.linear\_model import LinearRegression

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import mean\_squared\_error, r2\_score

# Load the dataset

data = pd.read\_csv('BTC-USD.csv')

# Prepare the data

X = data[['Open', 'High', 'Low', 'Volume']]

y = data['Close']

# Split the data into training and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Train the linear regression model

model = LinearRegression()

model.fit(X\_train, y\_train)

# Make predictions on the test set

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

# Evaluate the model using mean squared error and R^2 score

mse = mean\_squared\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

print("Mean squared error: {:.2f}".format(mse))

print("R^2 score: {:.2f}".format(r2))

# Plot the predicted prices against the actual prices

plt.plot(y\_test.values, label='Actual')

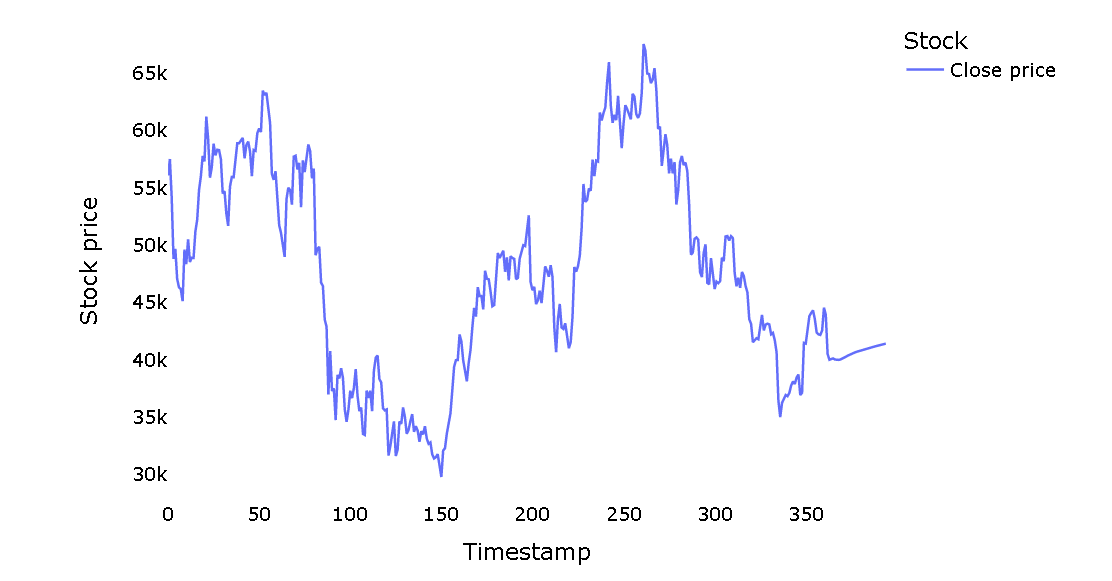
plt.plot(y\_pred, label='Predicted')

plt.legend()

plt.show()

This program loads historical Bitcoin prices from a CSV file, prepares the data by selecting the relevant features (Open, High, Low, and Volume), splits the data into training and test sets, trains a linear regression model on the training set, makes predictions on the test set, and evaluates the model using mean squared error and R^2 score.

Note that this is a very simple example and there are many factors that can affect Bitcoin prices, such as market sentiment, regulatory changes, and geopolitical events. Therefore, this model may not be accurate enough for real-world applications. More sophisticated models and features may be needed to achieve better accuracy.



**Task 1. Java REST API example**

Java REST API implementation for the given requirements using Spring Boot and MongoDB.

**Dependencies**

We need to include the following dependencies in our **pom.xml** file:

* Spring Boot Starter Web
* Spring Boot Starter Data MongoDB
* MongoDB Driver Sync

**Server Model**

Let's define the **Server** model class which will be stored in MongoDB:

package com.example.demo.model;

import org.springframework.data.annotation.Id;

import org.springframework.data.mongodb.core.mapping.Document;

@Document(collection = "servers")

public class Server {

@Id

private String id;

private String name;

private String language;

private String framework;

public Server() {}

public Server(String name, String language, String framework) {

this.name = name;

this.language = language;

this.framework = framework;

}

// getters and setters

}

**Server Repository**

Next, let's create a repository to interact with the MongoDB database. We will use the Spring Data MongoDB repository abstraction:

package com.example.demo.repository;

import com.example.demo.model.Server;

import org.springframework.data.mongodb.repository.MongoRepository;

import java.util.List;

public interface ServerRepository extends MongoRepository<Server, String> {

List<Server> findByNameContainingIgnoreCase(String name);

}

**Server Controller**

Now let's create the REST controller for handling HTTP requests:

package com.example.demo.controller;

import com.example.demo.model.Server;

import com.example.demo.repository.ServerRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

import java.util.Optional;

@RestController

@RequestMapping("/api/servers")

public class ServerController {

@Autowired

private ServerRepository serverRepository;

@GetMapping

public ResponseEntity<List<Server>> getServers(@RequestParam(required = false) String id) {

if (id != null) {

Optional<Server> server = serverRepository.findById(id);

if (server.isPresent()) {

return new ResponseEntity<>(List.of(server.get()), HttpStatus.OK);

} else {

return new ResponseEntity<>(HttpStatus.NOT\_FOUND);

}

} else {

List<Server> servers = serverRepository.findAll();

return new ResponseEntity<>(servers, HttpStatus.OK);

}

}

@PostMapping

public ResponseEntity<Server> createServer(@RequestBody Server server) {

Server savedServer = serverRepository.save(server);

return new ResponseEntity<>(savedServer, HttpStatus.CREATED);

}

@DeleteMapping("/{id}")

public ResponseEntity<HttpStatus> deleteServer(@PathVariable String id) {

serverRepository.deleteById(id);

return new ResponseEntity<>(HttpStatus.NO\_CONTENT);

}

@GetMapping("/findByName")

public ResponseEntity<List<Server>> findServersByName(@RequestParam String name) {

List<Server> servers = serverRepository.findByNameContainingIgnoreCase(name);

if (servers.isEmpty()) {

return new ResponseEntity<>(HttpStatus.NOT\_FOUND);

} else {

return new ResponseEntity<>(servers, HttpStatus.OK);

}

}

}

**Application Properties**

We also need to configure the MongoDB connection in the **application.properties** file:

spring.data.mongodb.host=localhost

spring.data.mongodb.port=27017

spring.data.mongodb.database=servers\_db

**Task 4. WEB UI Forms.**

**Create a basic WEB UI frontend for an application that you created for #1 or #2 using any UI framework of your choice. You should be able to create, show and delete records from your UI.**First, you will need to set up the required dependencies for your UI framework and create a new project. Once you have done that, you can create a UI component that will handle the GET request to retrieve all the servers from the REST API. Here's an example code snippet using the fetch() method:

import React, { useState, useEffect } from 'react';

function ServerList() {

const [servers, setServers] = useState([]);

useEffect(() => {

fetch('http://localhost:8080/servers')

.then(response => response.json())

.then(data => setServers(data))

.catch(error => console.error(error));

}, []);

return (

<table>

<thead>

<tr>

<th>ID</th>

<th>Name</th>

<th>Language</th>

<th>Framework</th>

</tr>

</thead>

<tbody>

{servers.map(server => (

<tr key={server.id}>

<td>{server.id}</td>

<td>{server.name}</td>

<td>{server.language}</td>

<td>{server.framework}</td>

</tr>

))}

</tbody>

</table>

);

}

export default ServerList;

Next, you can create a UI component that will handle the PUT request to create a new server object.

import React, { useState } from 'react';

function ServerForm() {

const [name, setName] = useState('');

const [language, setLanguage] = useState('');

const [framework, setFramework] = useState('');

const handleSubmit = event => {

event.preventDefault();

const server = {

name: name,

language: language,

framework: framework

};

fetch('http://localhost:8080/servers', {

method: 'PUT',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify(server)

})

.then(response => response.json())

.then(data => console.log(data))

.catch(error => console.error(error));

setName('');

setLanguage('');

setFramework('');

};

return (

<form onSubmit={handleSubmit}>

<label>

Name:

<input type="text" value={name} onChange={event => setName(event.target.value)} />

</label>

<br />

<label>

Language:

<input type="text" value={language} onChange={event => setLanguage(event.target.value)} />

</label>

<br />

<label>

Framework:

<input type="text" value={framework} onChange={event => setFramework(event.target.value)} />

</label>

<br />

<button type="submit">Create Server</button>

</form>

);

}

export default ServerForm;

Next, you can create a UI component that will handle the DELETE request to create a new server object.

mport React, { useState } from 'react';

function ServerForm()

{ const [name, setName] = useState('');

const [language, setLanguage] = useState('');

const [framework, setFramework] = useState('');

const handleSubmit = event => { event.preventDefault();

const server = {

name: name,

language: language,

framework: framework

};

fetch('http://localhost:8080/servers', {

method: 'PUT',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify(server)

})

.then(response => response.json())

.then(data => console.log(data))

.catch(error => console.error(error));

setName('');

setLanguage('');

setFramework('');

};

const handleDelete = event => { event.preventDefault();

fetch(`http://localhost:8080/servers/${serverId}`,

{

method: 'DELETE',

})

.then(response => {

if (!response.ok) {

throw new Error('Network response was not ok');

}

console.log('Server deleted successfully');

})

.catch(error => console.error(error));

};

return

( <div>

<form onSubmit={handleSubmit}>

<label>

Name: <input type="text" value={name}

onChange={event => setName(event.target.value)} />

</label> <br /> <label>

Language: <input type="text" value={language}

onChange={event => setLanguage(event.target.value)} />

</label> <br /> <label> Framework: <input type="text" value={framework}

onChange={event => setFramework(event.target.value)} /> </label> <br />

<button type="submit">Create Server</button> </form>

<form onSubmit={handleDelete}>

<label>

Server ID:

<input type="text" value={serverId} onChange={event => setServerId(event.target.value)} />

</label>

<br />

<button type="submit">Delete Server</button>

</form>

</div>

); }

export default ServerForm;