**SOURCE CODE**

app.py

# Author: DANAIAH, NAVAYA University JNTUK KAKAINADA

# Date: 03/4/2022

# Description: This is a Flask App that uses SQLite3 to

# execute (C)reate, (R)ead, (U)pdate, (D)elete operations

from flask import Flask

from flask import render\_template

from flask import request,session, flash

import sqlite3

app = Flask(\_\_name\_\_)

# Home Page route

@app.route("/")

def home():

return render\_template("home.html")

def userhome():

return render\_template('user/userhome')

# Route to form used to add a new student to the database

@app.route("/enternew")

def enternew():

return render\_template("student.html")

# Route to add a new record (INSERT) student data to the database

@app.route("/addrec", methods = ['POST', 'GET'])

def addrec():

# Data will be available from POST submitted by the form

if request.method == 'POST':

try:

nm = request.form['nm']

addr = request.form['add']

city = request.form['city']

zip = request.form['zip']

loginid = request.form['loginid']

email = request.form['email']

password = request.form['password']

# Connect to SQLite3 database and execute the INSERT

with sqlite3.connect('database.db') as con:

cur = con.cursor()

cur.execute("INSERT INTO students (name, loginid, email, password, addr, city, zip) VALUES (?,?,?,?,?,?,?)",(nm, loginid, email, password, addr, city, zip))

con.commit()

msg = "Record successfully added to database"

except:

con.rollback()

msg = "Error in the INSERT"

finally:

con.close()

# Send the transaction message to result.html

return render\_template('result.html',msg=msg)

# Route to SELECT all data from the database and display in a table

@app.route('/list')

def list():

# Connect to the SQLite3 datatabase and

# SELECT rowid and all Rows from the students table.

con = sqlite3.connect("database.db")

con.row\_factory = sqlite3.Row

cur = con.cursor()

cur.execute("SELECT rowid, \* FROM students")

rows = cur.fetchall()

con.close()

# Send the results of the SELECT to the list.html page

return render\_template("list.html",rows=rows)

# Route that will SELECT a specific row in the database then load an Edit form

@app.route("/edit", methods=['POST','GET'])

def edit():

if request.method == 'POST':

try:

# Use the hidden input value of id from the form to get the rowid

id = request.form['id']

# Connect to the database and SELECT a specific rowid

con = sqlite3.connect("database.db")

con.row\_factory = sqlite3.Row

cur = con.cursor()

cur.execute("SELECT rowid, \* FROM students WHERE rowid = " + id)

rows = cur.fetchall()

except:

id=None

finally:

con.close()

# Send the specific record of data to edit.html

return render\_template("edit.html",rows=rows)

# Route used to execute the UPDATE statement on a specific record in the database

@app.route("/editrec", methods=['POST','GET'])

def editrec():

# Data will be available from POST submitted by the form

if request.method == 'POST':

try:

# Use the hidden input value of id from the form to get the rowid

rowid = request.form['rowid']

nm = request.form['nm']

addr = request.form['add']

city = request.form['city']

zip = request.form['zip']

loginid = request.form['loginid']

email = request.form['email']

password = request.form['password']

# UPDATE a specific record in the database based on the rowid

with sqlite3.connect('database.db') as con:

cur = con.cursor()

cur.execute("UPDATE students SET name='"+nm+"', addr='"+addr+"', city='"+city+"', zip='"+zip+"', loginid='"+loginid+"',email='"+email+"', password='"+password+"' WHERE rowid="+rowid)

con.commit()

msg = "Record successfully edited in the database"

except:

con.rollback()

msg = "Error in the Edit: UPDATE students SET name="+nm+", addr="+addr+", city="+city+", zip="+zip+" loginid="+loginid+",email="+email+", password="+password+" WHERE rowid="+rowid

finally:

con.close()

# Send the transaction message to result.html

return render\_template('result.html',msg=msg)

# Route used to DELETE a specific record in the database

@app.route("/delete", methods=['POST','GET'])

def delete():

if request.method == 'POST':

try:

# Use the hidden input value of id from the form to get the rowid

rowid = request.form['id']

# Connect to the database and DELETE a specific record based on rowid

with sqlite3.connect('database.db') as con:

cur = con.cursor()

cur.execute("DELETE FROM students WHERE rowid="+rowid)

con.commit()

msg = "Record successfully deleted from the database"

except:

con.rollback()

msg = "Error in the DELETE"

finally:

con.close()

# Send the transaction message to result.html

return render\_template('result.html',msg=msg)

@app.route('/login', methods=['GET', 'POST'])

def login():

if request.method == 'POST':

connection = sqlite3.connect('database.db')

cursor = connection.cursor()

loginid = request.form['loginid']

password = request.form['password']

print(loginid, password)

query = "SELECT loginid, password FROM students WHERE loginid=? AND password=?"

cursor.execute(query, (loginid, password))

results = cursor.fetchall()

if not results:

error\_message = 'Invalid login credentials. Please try again.'

return render\_template('login.html', error=error\_message)

else:

return render\_template('user/userhome.html')

return render\_template('login.html')

import pandas as pd

import numpy as np

import os

from flask import current\_app

from sklearn.ensemble import RandomForestRegressor

from sklearn.model\_selection import RandomizedSearchCV

import matplotlib.pyplot as plt

import seaborn as sns

# path = os.path.join(current\_app.root\_path, 'media', 'ElectricCarData\_Clean\_Me.csv')

# path1 = pd.read\_csv(path)

@app.route('/dataset')

def dataset():

path = os.path.join(current\_app.root\_path, 'media', 'ElectricCarData\_Clean\_Me.csv')

path1 = pd.read\_csv(path)

data = path1.to\_html()

return render\_template('user/dataset.html',data=data)

@app.route('/training')

def training():

path = os.path.join(current\_app.root\_path, 'media', 'ElectricCarData\_Clean\_Me.csv')

df = pd.read\_csv(path)

df['FullName'] = df['Brand'] + '-' + df['Model']

df\_1 = df.loc[df['PriceEuro'] <= 50000]

df\_2 = df.loc[df['PriceEuro'] > 50000]

t1 = 'Less than 50,000 Euros'

t2 = 'More than 50,000 Euros'

def power\_train(dataframe):

sns.countplot(x= dataframe['PowerTrain'])

plt.title('Count Plot of Powertrain', fontsize = 20)

plt.xlabel('Power Train', fontsize = 15)

plt.ylabel('Count', fontsize = 15)

power\_train(df)

def bodystyle(dataframe):

plt.figure(figsize=(10, 5))

sns.countplot(x= 'BodyStyle', data= dataframe, hue='PowerTrain')

plt.title('Count plot of Body Style', fontsize= 20)

plt.xlabel('Body Style', fontsize= 15)

plt.ylabel('Count', fontsize= 15)

plt.show()

bodystyle(df)

def range(dataframe, price):

plt.figure(figsize=(20,5))

sns.set\_theme(style="whitegrid")

sns.barplot(x='FullName', y='Range\_Km', data=dataframe, hue='PowerTrain')

plt.title('''Range(Km) of EV's costing {} '''.format(price), fontsize=20)

plt.ylabel('Range (Km)', fontsize=15)

plt.xlabel('Model', fontsize=15)

plt.xticks(rotation=90)

plt.show()

# Bar Graphs

range(df\_1, t1)

range(df\_2, t2)

def range\_batterypack(dataframe, text):

fig = plt.figure(figsize=(20,5))

ax1 = plt.subplot()

ax1.bar(dataframe['FullName'], dataframe['Range\_Km'],label= 'Range (Km)', color= 'steelblue')

plt.legend(loc= 'upper left', bbox\_to\_anchor = (0, 1.105))

ax2 = ax1.twinx()

ax2.scatter(dataframe['FullName'], dataframe['Battery\_Pack Kwh'], label= 'Battery Pack', color = 'black')

plt.title('''RANGE (Km) vs BATTERY PACK CAPACITY (KwH) of EV's costing {}'''.format(text), fontsize= 20)

ax1.set\_xlabel('Models', size = 20)

ax1.set\_ylabel('Range (Km)', color = 'steelblue', size = 20)

ax2.set\_ylabel('Battery Pack Capacity (Kwh)', color= 'black', size= 20)

plt.legend(loc= 'upper left', bbox\_to\_anchor = (0, 1))

ax1.set\_xticklabels(df\_1['FullName'], rotation = 'vertical')

plt.show()

range\_batterypack(df\_1, t1)

range\_batterypack(df\_2, t2)

def acc(dataframe, text):

plt.figure(figsize=(20,5))

sns.set\_theme(style="whitegrid")

sns.barplot(x='FullName', y='AccelSec', data=dataframe, hue='PowerTrain')

plt.title('''Acceleration 0-100 Km of EV's costing {}'''.format(text), fontsize=20)

plt.ylabel('Acceleration (seconds)')

plt.xlabel('Model')

plt.xticks(rotation=90)

plt.show()

# Acceleration

acc(df\_1, t1)

acc(df\_2, t2)

def range\_price(dataframe, text):

fig = plt.figure(figsize=(20,5))

ax1 = plt.subplot()

ax1.bar(dataframe['FullName'], dataframe['Range\_Km'],label= 'Range (Km)', color= 'steelblue')

plt.legend(loc= 'upper left', bbox\_to\_anchor = (0, 1.1))

ax2 = ax1.twinx()

ax2.scatter(dataframe['FullName'], dataframe['PriceEuro'], label= 'Price', color = 'black')

plt.title('''RANGE (Km) vs PRICE (Euros)of EV's costing {}'''.format(text), fontsize= 20)

ax1.set\_xlabel('Models', size = 20)

ax1.set\_ylabel('Range (Km)', color = 'steelblue', size = 20)

ax2.set\_ylabel('Price (Euros)', color= 'black', size= 20)

plt.legend(loc= 'upper left', bbox\_to\_anchor = (0, 1))

ax1.set\_xticklabels(df\_1['FullName'], rotation = 'vertical')

plt.show()

# price vs acceleration

range\_price(df\_1, t1)

range\_price(df\_2, t2)

def range\_efficiency(dataframe, text):

fig = plt.figure(figsize=(20,5))

ax1 = plt.subplot()

ax1.bar(dataframe['FullName'], dataframe['Range\_Km'],label= 'Range (Km)', color= 'darkseagreen')

plt.legend(loc= 'upper left', bbox\_to\_anchor = (0, 1.1))

ax2 = ax1.twinx()

ax2.scatter(dataframe['FullName'], dataframe['Efficiency\_WhKm'], label= 'Price', color = 'black')

plt.title('''RANGE (Km) vs Efficiency (Wh/km)of EV's costing {}'''.format(text), fontsize= 20)

ax1.set\_xlabel('Models', size = 20)

ax1.set\_ylabel('Range (Km)', color = 'darkseagreen', size = 20)

ax2.set\_ylabel('Efficiency (Wh/Km)', color= 'black', size= 20)

plt.legend(loc= 'upper left', bbox\_to\_anchor = (0, 1))

ax1.set\_xticklabels(df\_1['FullName'], rotation = 'vertical')

plt.show()

# range vs efficiency

range\_efficiency(df\_1, t1)

range\_efficiency(df\_2, t2)

def fastcharge(dataframe, price):

plt.figure(figsize=(20, 5))

sns.set\_theme(style="whitegrid")

sns.barplot(x='FullName', y='FastCharge\_KmH', data=dataframe, color='lightslategrey')

plt.title('''Fast Charging of EV's costing {} '''.format(price), fontsize=20)

plt.ylabel('Charging Capacity (kmH)', fontsize=15)

plt.xlabel('Model', fontsize=15)

plt.xticks(rotation=90)

plt.show()

# Fast charge Data

fastcharge(df\_1, t1)

fastcharge(df\_2, t2)

# df

df = df.drop(['Brand','Model','FullName'],axis=1)

# df

X = df[[x for x in df.columns if x!='Range\_Km']]

X = pd.get\_dummies(X)

# X

Y = df['Range\_Km']

# Y

# Number of trees in random forest

n\_estimators = [int(x) for x in np.linspace(start = 200, stop = 2000, num = 10)]

# Number of features to consider at every split

max\_features = ['auto', 'sqrt']

# Maximum number of levels in tree

max\_depth = [int(x) for x in np.linspace(10, 110, num = 11)]

max\_depth.append(None)

# Minimum number of samples required to split a node

min\_samples\_split = [2, 5, 10]

# Minimum number of samples required at each leaf node

min\_samples\_leaf = [1, 2, 4]

# Method of selecting samples for training each tree

bootstrap = [True, False]

# Create the random grid

random\_grid = {'n\_estimators': n\_estimators,

'max\_features': max\_features,

'max\_depth': max\_depth,

'min\_samples\_split': min\_samples\_split,

'min\_samples\_leaf': min\_samples\_leaf,

'bootstrap': bootstrap}

print(random\_grid)

{'bootstrap': [True, False],

'max\_depth': [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, None],

'max\_features': ['auto', 'sqrt'],

'min\_samples\_leaf': [1, 2, 4],

'min\_samples\_split': [2, 5, 10],

'n\_estimators': [200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000]}

rf = RandomForestRegressor()

rf\_random = RandomizedSearchCV(estimator = rf, param\_distributions = random\_grid, n\_iter = 100, cv = 3, verbose=2, random\_state=42, n\_jobs = -1)

def evaluate(model, test\_features, test\_labels):

predictions = model.predict(test\_features)

errors = abs(predictions - test\_labels)

mape = 100 \* np.mean(errors / test\_labels)

accuracy = 100 - mape

print('Model Performance')

print('Average Error: {:0.4f} degrees.'.format(np.mean(errors)))

print('Accuracy = {:0.2f}%.'.format(accuracy))

base\_model = RandomForestRegressor(n\_estimators = 1600, random\_state = 42)

base\_model.fit(X, Y)

base\_accuracy = evaluate(base\_model, X, Y)

# Perform evaluation directly without using the evaluate function

predictions = base\_model.predict(X)

errors = abs(predictions - Y)

mape = 100 \* np.mean(errors / Y)

accuracy = 100 - mape

# Save the trained model

import joblib

joblib.dump(base\_model, 'trained\_model.joblib')

# Print model performance metrics

print('Model Performance')

print('Average Error: {:0.4f} degrees.'.format(np.mean(errors)))

print('Accuracy = {:0.2f}%.'.format(accuracy))

return render\_template('user/training.html',accuracy=accuracy,errors=mape)

# @app.route('/predication',methods=['GET', 'POST'])

# def predication():

# if request.method == 'POST':

# AccelSec = request.form.get("AccelSec")

# TopSpeed\_KmH = request.form.get('TopSpeed\_KmH')

# Range\_Km = request.form.get('Range\_km')

# Battery\_Pack\_Kwh = request.form.get('Battery\_Pack\_Kwh')

# Efficiency\_WhKm = request.form.get('Efficiency\_WhKm')

# FastCharge\_KmH = request.form.get('FastCharge\_KmH')

# RapidCharge = request.form.get('RapidCharge')

# PowerTrain = request.form.get('PowerTrain')

# PlugType = request.form.get('PlugType')

# BodyStyle = request.form.get('BodyStyle')

# Segment = request.form.get('Segment')

# Seats = request.form.get('Seats')

# path = os.path.join(current\_app.root\_path, 'media','ElectricCarData\_Modified.csv')

# df = pd.read\_csv(path)

# column\_mappings = {

# 'RapidCharge':{'Yes':1,'No':0},

# 'PlugType':{'Type 2 CCS':1,'Type 2':2,'Type 2 CHAdeMO':3},

# 'PowerTrain':{'AWD':1,'FWD':2,'RWD':3},

# 'BodyStyle':{'SUV':1,'Hatchback':2,'Sedan':3,'Liftback':4,'Pickup':5,'Cabrio':6,'SPV':7},

# 'Segment':{'C':1,'B':2,'D':3,'F':4,'E':5,'A':6,'N':7}

# }

# df.replace(column\_mappings, inplace=True)

# print(df)

# # Compute the correlation matrix

# X = df[[x for x in df.columns if x!='PriceEuro']]

# print(X)

# # X = pd.get\_dummies(X)

# Y = df['PriceEuro']

# # Importing necessary libraries

# from sklearn.impute import SimpleImputer

# from sklearn.ensemble import RandomForestRegressor

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

# # Impute missing values

# imputer = SimpleImputer(strategy='mean')

# X\_imputed = imputer.fit\_transform(X)

# # Splitting the data into training and testing sets

# X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_imputed, Y, test\_size=0.2, random\_state=42)

# # Initializing the Random Forest regressor

# rf\_regressor = RandomForestRegressor(n\_estimators=100, random\_state=42)

# # Training the model

# rf\_regressor.fit(X\_train, y\_train)

# X\_test = [[AccelSec,TopSpeed\_KmH,Range\_Km,Battery\_Pack\_Kwh,Efficiency\_WhKm, FastCharge\_KmH, RapidCharge, PowerTrain, PlugType, BodyStyle, Segment, Seats]]

# # Making predictions on the test set

# predictions = rf\_regressor.predict(X\_test)

# #new features

# return render\_template('user/predication.html',y\_pred=predictions)

# return render\_template('user/predication.html')

# Importing necessary libraries

from sklearn.impute import SimpleImputer

from sklearn.ensemble import RandomForestRegressor

from flask import request

@app.route('/predication', methods=['GET', 'POST'])

def predication():

if request.method == 'POST':

# Function to convert form inputs to float or return None if empty

def to\_float\_or\_none(value):

return float(value) if value is not None and value.strip() else None

# Getting form inputs

AccelSec = to\_float\_or\_none(request.form.get("AccelSec"))

TopSpeed\_KmH = to\_float\_or\_none(request.form.get('TopSpeed\_KmH'))

Range\_Km = to\_float\_or\_none(request.form.get('Range\_km'))

Battery\_Pack\_Kwh = to\_float\_or\_none(request.form.get('Battery\_Pack\_Kwh'))

Efficiency\_WhKm = to\_float\_or\_none(request.form.get('Efficiency\_WhKm'))

FastCharge\_KmH = to\_float\_or\_none(request.form.get('FastCharge\_KmH'))

RapidCharge = to\_float\_or\_none(request.form.get('RapidCharge'))

PowerTrain = to\_float\_or\_none(request.form.get('PowerTrain'))

PlugType = to\_float\_or\_none(request.form.get('PlugType'))

BodyStyle = to\_float\_or\_none(request.form.get('BodyStyle'))

Segment = to\_float\_or\_none(request.form.get('Segment'))

Seats = to\_float\_or\_none(request.form.get('Seats'))

path = os.path.join(current\_app.root\_path, 'media', 'ElectricCarData\_Modified.csv')

df = pd.read\_csv(path)

column\_mappings = {

'RapidCharge': {'Yes': 1, 'No': 0},

'PlugType': {'Type 2 CCS': 1, 'Type 2': 2, 'Type 2 CHAdeMO': 3},

'PowerTrain': {'AWD': 1, 'FWD': 2, 'RWD': 3},

'BodyStyle': {'SUV': 1, 'Hatchback': 2, 'Sedan': 3, 'Liftback': 4, 'Pickup': 5, 'Cabrio': 6, 'SPV': 7},

'Segment': {'C': 1, 'B': 2, 'D': 3, 'F': 4, 'E': 5, 'A': 6, 'N': 7}

}

df.replace(column\_mappings, inplace=True)

# Compute the correlation matrix

X = df[[x for x in df.columns if x != 'PriceEuro']]

Y = df['PriceEuro']

# Impute missing values

imputer = SimpleImputer(strategy='mean')

X\_imputed = imputer.fit\_transform(X)

# Initializing the Random Forest regressor

rf\_regressor = RandomForestRegressor(n\_estimators=100, random\_state=42)

# Training the model

rf\_regressor.fit(X\_imputed, Y)

X\_test = np.array([[AccelSec, TopSpeed\_KmH, Range\_Km, Battery\_Pack\_Kwh, Efficiency\_WhKm, FastCharge\_KmH,

RapidCharge, PowerTrain, PlugType, BodyStyle, Segment, Seats]])

X\_test\_imputed = imputer.transform(X\_test)

predictions = rf\_regressor.predict(X\_test\_imputed)

return render\_template('user/predication.html', y\_pred=predictions[0])

return render\_template('user/predication.html')

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True,port=8000)

create\_table.db

# Author: DANAIAH,NAVAYA University of JNTUK KAKINADA

# Date: 03/04/2022

# Description: This python script assumes that you already have

import sqlite3

# Connect to the database

def connect\_db():

conn = sqlite3.connect('database.db')

print("Connected to database successfully")

# Check if the students table exists

cursor = conn.cursor()

cursor.execute("SELECT name FROM sqlite\_master WHERE type='table' AND name='students'")

table\_exists = cursor.fetchone()

if not table\_exists:

# Create the students table if it doesn't exist

conn.execute('CREATE TABLE students (name TEXT, addr TEXT, city TEXT, zip TEXT, loginid TEXT, email TEXT, password TEXT)')

print("Created table successfully!")

else:

print("Table 'students' already exists.")

# Close the connection

conn.close()

connect\_db()