DOCUMENTATION FOR IRIS SPECIES PREDICTION USING MACHINE LEARNING AND DJANGO

In the field of data science and machine learning, classification problems are pivotal for making informed decisions based on data. One of the classic datasets used for teaching and demonstrating these concepts is the **Iris dataset**, which contains measurements of various features of iris flowers specifically, sepal length, sepal width, petal length, and petal width across three different species: Iris Setosa, Iris Versicolor, and Iris Virginica.

The primary objective of this project is to develop a web application that predicts the species of iris flowers based on their measurements using machine learning algorithms. By leveraging the powerful **k-nearest neighbours (k-NN)** classification technique, we aim to create an accurate model capable of distinguishing between the different species of iris flowers.

The project is built using **Django**, a high-level Python web framework that allows for the rapid development of secure and maintainable web applications. By integrating machine learning with Django, we provide users with an interactive platform where they can input flower measurements and receive instant predictions about the iris species.

Steps to follow how to predict the iris dataset using machine learning algorithms and DJANGO:

Step1: Set Up the Project Environment

→ Install Python and Django

→ Create a Django Project

Step2: Prepare the Machine Learning Model

→ Install the necessary packages

→ Create and Train the Model (k-NN Algorithm Example)

→ Run the Script

Step3: Integrate the Model into Django

→ Create a View to Predict Iris Species

→ Create Templates for User Input and Result Display

→ Configure URLs for the App

→ Include App URLs in the Project URL Configuration

Step4: Run the Django Server

→ Run the Django development server

→ Visit the following URL in your browser.

CODE EXPLANATION:

Training and Saving an Iris Species Prediction Model Using k-NN and Pickle (ip.ipynb file code)

```
import os
```

→This imports the **os** module, which provides tools for interacting with the operating system.

```
import pickle
```

→This imports the **pickle** module, which is used for **serialization** (saving objects to files) and **deserialization** (loading objects from files).

```
from sklearn.datasets import load_iris
```

→This imports the **load_iris** function from **scikit-learn**.
→The **Iris dataset** is a popular dataset used in machine learning, containing measurements of three different species of iris flowers.

```
from sklearn.neighbors import KNeighborsClassifier
```

→This imports the **K-Nearest Neighbors (k-NN)** classifier from scikit-learn's neighbors module.
→**k-NN** is a supervised machine learning algorithm used for **classification and regression** tasks.

```
from sklearn.model_selection import train_test_split
```

→This imports the **train_test_split** function, which is used to **split a dataset** into training and testing subsets.
→This is important to evaluate the performance of a machine learning model, ensuring it performs well on **unseen data**.

```
iris=load_iris()
```

Loading Iris Dataset
→Loads the Iris dataset from scikit-learn's datasets module
→load_iris() returns a Bunch object containing the dataset

```
x=iris.data
y=iris.target
```

→ iris.data contains the feature values (4 features: sepal length, sepal width, petal length, petal width).
→iris.target contains the target values (3 classes: Setosa, Versicolor, Virginica)

```
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.2,random_state=42)
```
→Splits the data into training and testing sets using train_test_split from scikit-learn's model_selection module.
→test_size=0.2 means 20% of the data will be used for testing.
→random_state=42 ensures reproducibility of the split.

```
model=KNeighborsClassifier(n_neighbors=3)
```
→ Creates a K-Nearest Neighbors (KNN) classifier with 3 neighbors.
→ KNeighborsClassifier is from scikit-learn's neighbors module.

```
model.fit(x_train,y_train)
```
→Trains the KNN model on the training data (x_train and y_train).

```
with open("model.pkl","wb") as f:
    pickle.dump(model,f)
```
→ Saves the trained model to a file named model.pkl using Python's pickle module.
→wb mode indicates writing in binary format.
→pickle.dump serializes the model object and writes it to the file.

Django URL Patterns(mliris→urls.py file)

```
from django.contrib import admin
```
→Imports the admin module from Django's contrib package, which provides the admin interface

```
from django.urls import path,include
```
→Imports two functions from Django's urls module:
→path: used to define URL patterns.
→include: used to include other URL configuration files.

```
urlpatterns = [
```
→Defines a list called urlpatterns, which contains URL patterns for the project.

```
path('admin/', admin.site.urls),
```
→Defines a URL pattern for the admin interface.
→admin/ is the URL prefix.
→ admin.site.urls is the view function that handles admin URLs.
```
path("",include("app.urls"))
```
→Defines a URL pattern that includes another URL configuration file

Django URL Patterns(app→urls.py file)

```
from django.urls import path,include
```
→imports the path and include functions for defining URL patterns.

```
from .views import predict
```
→imports the predict view function from the current app's file.

```
urlpatterns= [
```
→a list of URL patterns for the app

```
path("",predict)
```
→ Defines a single URL pattern:

→ "": empty string, matches the root URL of the app.

→ predict: the view function to handle requests to this URL

Django View Function for Predictive Modeling(app→views.py file)

Importing modules:

```
from django.shortcuts import render
```
→imports the render function for rendering HTML templates.

```
import os
```
→ imports the os module for file path manipulation.

```
import pickle
```
→ imports the pickle module for loading the saved model.

```
from sklearn.neighbors import KNeighborsClassifier
```
→ imports the KNeighborsClassifier class.

Loading the Model:

```
dir = os.path.dirname(__file__)
model_path = os.path.join(dir, "model.pkl")
try:
    with open(model_path, "rb") as f:
        model = pickle.load(f)
except FileNotFoundError:
    model = None
```
→Gets the directory of the current file using os.path.dirname(__file__).

→ Constructs the path to the saved model file (model.pkl) using os.path.join.

→Tries to load the model from the file using pickle.load. If successful, assigns it to the model variable.

→ If the file is missing, sets model to None.

Prediction View:

```
def predict(request):
    prediction = None  # Initialize prediction
```

```
    if request.method == "POST":
        try:
            # Get data from the POST request
            sl = float(request.POST.get("sl", 0))
            sw = float(request.POST.get("sw", 0))
            pl = float(request.POST.get("pl", 0))
            pw = float(request.POST.get("pw", 0))

            # Check if model was loaded correctly
            if model is None:
                raise ValueError("Model not found or failed to load.")

            # Prepare the data for prediction
            parameters = [[sl, sw, pl, pw]]

            # Make the prediction using the loaded model
            prediction_index = model.predict(parameters)[0]
            classes = ["setosa", "versicolor", "virginica"]
            prediction = classes[prediction_index]

        except ValueError as e:
            prediction = f"Input error: {str(e)}"
        except Exception as e:
            prediction = f"Error in prediction: {str(e)}"

    return render(request, "predict.html", {"prediction": prediction})
```

→ Defines a view function predict that handles POST requests.

→ Extracts four float values from the POST request (sl, sw, pl, pw).

→ Checks if the model is loaded. If not, raises a ValueError.

→Prepares the data for prediction by creating a 2D list (parameters).

→Makes a prediction using the loaded model and stores the result in prediction_index.

→ Maps the prediction index to a class label (setosa, versicolor, or virginica).

→Catches any exceptions and sets prediction to an error message.

→Renders the predict.html template with the prediction variable.

Run Django Commands with the below Script(template→manage.py)

Importing Modules:

```
import os
import sys
```

→imports the os module for interacting with the operating system.

→imports the sys module for accessing system-specific variables.

Main Function

```
def main():
    """Run administrative tasks."""
```

→ Defines a main function that serves as the entry point for the script.

Setting Django Settings

```
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'mliris.settings')
```

→Environment variable to mliris.settings, which tells Django where to find the project's settings file.

Checking Django Installation

```
try:
        from django.core.management import execute_from_command_line
    except ImportError as exc:
        raise ImportError(

        ) from exc
    execute_from_command_line(sys.argv)
```

→ Tries to import Django's execute_from_command_line function.

→If the import fails, raises an ImportError with a helpful error message.

Entry point

```
if __name__ == '__main__':
    main()
```

→ Checks if the script is being run directly (not being imported).

→Calls the main function if so.

Iris Flower Prediction Web Interface(template→predict.html)

```
{% load static %}
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{% block title %}Iris Flower Prediction{% endblock %}</title>

    <!-- Bootstrap CSS CDN -->
    <link
href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css
" rel="stylesheet">
    <style>
        body {
```

```
            background-color: aquamarine;
        }
        .form-container {
            margin-top: 50px;
            padding: 30px;
            background-color: white;
            border-radius: 10px;
            box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
        }
    </style>
</head>
<body>

    <div class="container">
        <div class="text-center my-4">
            <img src="{% static 'total.png' %}" class="img-fluid" alt="Banner"
width="1500" height="300">
            <h1 class="my-4">Predict the Iris Flower Species</h1>
        </div>

        <div class="form-container">
            <form method="POST" class="form-group">
                {% csrf_token %}
                <div class="mb-3">
                    <label for="sl">Sepal Length:</label>
                    <input id="sl" name="sl" type="number" step="0.1" min="0"
max="8" class="form-control" required>
                </div>

                <div class="mb-3">
                    <label for="sw">Sepal Width:</label>
                    <input id="sw" name="sw" type="number" step="0.1" min="0"
max="8" class="form-control" required>
                </div>

                <div class="mb-3">
                    <label for="pl">Petal Length:</label>
                    <input id="pl" name="pl" type="number" step="0.1" min="0"
max="8" class="form-control" required>
                </div>

                <div class="mb-3">
                    <label for="pw">Petal Width:</label>
                    <input id="pw" name="pw" type="number" step="0.1" min="0"
max="8" class="form-control" required>
                </div>
```

```
                <button type="submit" class="btn btn-primary btn-
block">Predict</button>
            </form>

            {% if prediction %}
                <div class="alert alert-info mt-4">
                    <h4>Prediction: {{ prediction }}</h4>
                </div>

                {% if prediction == "setosa" %}
                    <img src="{% static 'setosa.jpg' %}" class="img-fluid"
width="200" height="200" alt="Setosa">
                {% elif prediction == "versicolor" %}
                    <img src="{% static 'versicolor.jpg' %}" class="img-fluid"
width="200" height="200" alt="Versicolor">
                {% elif prediction == "virginica" %}
                    <img src="{% static 'verginica.jpg' %}" class="img-fluid"
width="200" height="200" alt="Virginica">
                {% endif %}
            {% endif %}
        </div>
    </div>

    <!-- Bootstrap JS and dependencies -->
    <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>
    <script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.9.3/dist/umd/popper.min.js"
></script>
    <script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"><
/script>
</body>
</html>
```

Loading Static Files and HTML Structure

→{% load static %}: Loads Django's static template tag.

→ <!DOCTYPE html>: Declares the document type.

→ <html lang="en">: Starts the HTML document with English language attribute.

→<head>: Begins the document's head section.

→<meta charset="UTF-8">: Specifies the character encoding.

Meta Tags and Title

→ <meta name="viewport" content="width=device-width, initial-scale=1.0">: Sets the viewport settings.

→<title>{% block title %}Iris Flower Prediction{% endblock %}</title>: Sets the page title.

CSS Styles

→ <!-- Bootstrap CSS CDN -->: Links to Bootstrap's CSS CDN.

→<link href="(link unavailable)" rel="stylesheet">: Links to Bootstrap CSS.

→ <style>...</style>: Defines custom CSS styles.

HTML Body and Container

→ </head>: Closes the <head> section.

→ <body>: Starts the HTML body.

→ : Creates a container div.

→ : Creates a centered div.

→<img src="{% static 'total.png' %}" ...>: Displays an image.

Form for Prediction

→ : Creates a form container div.

→ <form method="POST" class="form-group">: Defines a form with POST method.

→{% csrf_token %}: Includes Django's CSRF token.

Prediction Result and Image

→{% if prediction %}: Checks if prediction variable is set.

Conclusion

This project successfully demonstrates the application of machine learning techniques for predicting the species of iris flowers based on sepal and petal measurements. By employing popular machine learning models such as Logistic Regression, Decision Tree, or K-Nearest Neighbors (KNN), the system achieved good accuracy in classifying the three iris species: **Setosa, Versicolor, and Virginica**.Integrating the machine learning model with Django facilitated the creation of a user-friendly web application, where users can input flower attributes and instantly receive predictions. This seamless connection between backend machine learning logic and frontend user interaction showcases the potential of combining **data science** with **web development** to solve real-world problems.

----------KUNCHAPU GOWTHAMI----------