Iris Flower Classification Project

- PROJECT OVERVIEW :
  The Iris flower dataset is a popular dataset commonly used for tasks in pattern recognition and classification.

  It includes data on three Iris flower species: Setosa, Versicolor, and Virginica, with measurements of four key features: sepal length, sepal width, petal length, and petal width.

  This project involves building machine learning models to classify these species based on these measurements.

- Dataset:
  - Attributes:
    1. Sepal Length (in cm)
    2. Sepal Width (in cm)
    3. Petal Length (in cm)
    4. PetalWidth (in cm)
  - Target: Species (Setosa, Versicolor, Virginica)
  - Total Instances: 150 samples
  - Source: The Iris dataset is sourced from Kaggle.

- Objective:
  The objective of this project is to develop and compare various machine learning models to classify the species of Iris flowers based on their physical measurements. The models chosen for comparison include:

  - K-Nearest Neighbors (K-NN) Classifier
  - Logistic Regression
  - Support Vector Machine (SVM)
  - Decision Tree Classifier

  We will evaluate the performance of each model, selecting and saving the most accurate one for future use.

- dataset summary:

importing the required libaries

```
[1] import pandas as pd
    import numpy as np
    import seaborn as sns
    import matplotlib as plt
    import sklearn as sklearn
```

Importing Libraries
We imported key Python libraries to support data analysis, visualization, and machine learning:

- pandas (pd): For data manipulation and handling with DataFrames.
- numpy (np): For numerical computations, especially with arrays.
- seaborn (sns): Simplifies statistical plotting with attractive, pre-set color schemes.
- matplotlib.pyplot (plt): Provides extensive plotting functions for detailed visualizations.
- scikit-learn (sklearn): A machine learning library with tools for preprocessing, modeling, and evaluation.

These libraries streamline the data science workflow, from data preparation to visualization and model building.

```
data = pd.read_csv('/content/Iris.csv')
data.head(2)
```

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa

Next steps: Generate code with data | View recommended plots | New interactive sheet

The code loads the Iris.csv file into a DataFrame named data and displays the first two rows using data.head(2). This provides a quick preview of the dataset's structure, confirming successful loading and showing key columns for initial inspection.

.

```
[17] data.groupby('Species').mean()
```

Species	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm
Iris-setosa	25.5	5.006	3.418	1.464	0.244
Iris-versicolor	75.5	5.936	2.770	4.260	1.326
Iris-virginica	125.5	6.588	2.974	5.552	2.026

Using data.groupby('Species').mean(), we calculate the average values for each numeric feature within each species. This summary helps compare characteristics like sepal and petal dimensions across different species in the dataset.n.

.

- Encoding the columns:

```
[8] from sklearn.preprocessing import LabelEncoder
    label_encoder = LabelEncoder()
    data['Species'] = label_encoder.fit_transform(data['Species'])
```

This code converts the `Species` column in a DataFrame from categorical labels to numeric labels using `LabelEncoder`, allowing machine learning models to interpret the data. Each unique category in `Species` gets assigned a unique integer.

- splitting the columns:

```
[9] from sklearn.model_selection import train_test_split
    X = data.drop(columns='Species')
    y = data['Species']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

This code splits the dataset into training and testing sets:

1. `X` is created by removing the `Species` column from `data`, containing only the feature variables.
2. `y` is set as the `Species` column, which is the target variable.
3. `train_test_split` then splits `X` and `y` into training (80%) and testing (20%) sets.
4. `random_state=42` ensures the split is reproducible.

- Model Selection and Training:

```
[10] from sklearn.linear_model import LogisticRegression

     model = LogisticRegression(multi_class='multinomial', solver='lbfgs', max_iter=200)
```

This code initializes a logistic regression model for multi-class classification:

1. `LogisticRegression` is set up for handling multiple classes (`multi_class='multinomial'`), meaning it will predict more than two classes.
2. `solver='lbfgs'` specifies the optimization algorithm, which is effective for small to medium-sized datasets and supports multinomial classification.
3. `max_iter=200` sets the maximum number of iterations for the solver to converge, increasing it from the default (100) to help ensure the model converges.

```
model.fit(X_train, y_train)
```

This line trains (or "fits") the logistic regression model on the training data:

1. `X_train` contains the training feature variables.
2. `y_train` contains the training target variable (`Species` in this case).

After running this line, the `model` will learn from the training data and adjust its parameters to minimize classification error.

- Standardizing Features with StandardScaler:

```
[14] from sklearn.preprocessing import StandardScaler

     scaler = StandardScaler()
     X_train = scaler.fit_transform(X_train)
     X_test = scaler.transform(X_test)
```

This code standardizes the feature data:

1. `StandardScaler` scales features to have a mean of 0 and a standard deviation of 1.
2. `fit_transform` on `X_train` learns the scaling parameters and applies them to `X_train`.
3. `transform` on `X_test` uses the same scaling parameters to transform `X_test`, ensuring consistency between train and test data.es.

- Generating Polynomial Features for Non-Linear Relationships:

```
[15] from sklearn.preprocessing import PolynomialFeatures

     poly = PolynomialFeatures(degree=2, include_bias=False)
     X_train_poly = poly.fit_transform(X_train)
     X_test_poly = poly.transform(X_test)
```

This code generates polynomial features to capture non-linear relationships:

1. `PolynomialFeatures(degree=2, include_bias=False)` creates polynomial features up to the second degree without adding a bias (constant) term.
2. `fit_transform` on `X_train` learns and applies the polynomial transformation to `X_train`.
3. `transform` on `X_test` applies the same transformation to `X_test`, ensuring consistency between the train and test sets.

- Hyperparameter Tuning for Multinomial Logistic Regression with GridSearchCV:

```
from sklearn.model_selection import GridSearchCV

param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100]}
log_reg = LogisticRegression(multi_class='multinomial', solver='lbfgs', max_iter=200)
grid_search = GridSearchCV(log_reg, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)

print("Best parameters:", grid_search.best_params_)
best_model = grid_search.best_estimator_
```

This code uses `GridSearchCV` to tune the `C` parameter for a multinomial logistic regression model:

1. **Parameter Grid** (`param_grid`): Specifies values for `C` (regularization strength) to test.
2. **Model Definition**: Defines a logistic regression model for multiclass classification.
3. **Grid Search**: Uses 5-fold cross-validation (`cv=5`) and accuracy as the scoring metric.
4. **Fit Model**: `grid_search.fit(X_train, y_train)` finds the best `C` value.
5. **Results**:`grid_search.best_params_` shows the best c and grid_search.best_estimato r gives the best model.

- cross-validation:

```
from sklearn.model_selection import cross_val_score

# Cross-validate the model
cv_scores = cross_val_score(best_model, X_train, y_train, cv=4)
print("Cross-Validation Scores:", cv_scores)
print("Mean CV Score:", cv_scores.mean())
```

```
Cross-Validation Scores: [1.        1.        1.        0.96666667]
Mean CV Score: 0.9916666666666667
```

This code uses cross-validation to evaluate the performance of `best_model` (found from `GridSearchCV`) on `X_train` and `y_train`:

1. **Cross-Validation Scores**: `cross_val_score(best_model, X_train, y_train, cv=4)` calculates the model's accuracy across 4 folds.
2. **Results**: `cv_scores` contains the scores for each fold, printed with `print("Cross-Validation Scores:", cv_scores)`.
3. **Mean Score**: `cv_scores.mean()` gives the average accuracy across folds, printed with `print("Mean CV Score:", cv_scores.mean())`.

This provides an estimate of the model's generalization performance.

- performance evaluvation:

```
# Fit the best model on the transformed data
best_model.fit(X_train, y_train)

# Make predictions
y_pred = best_model.predict(X_test)

# Evaluation metrics
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

from sklearn.metrics import classification_report

# Assuming y_test are true labels and y_pred are predicted labels from the model
print("Classification Report:\n")
print(classification_report(y_test, y_pred, target_names=['Setosa', 'Versicolor', 'Virginica']))
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_logistic.py:1247: FutureWarning: 'multi_class' was deprecated in version 1.5 and w
  warnings.warn(
Classification Report:

              precision    recall  f1-score   support

      Setosa       1.00      1.00      1.00        11
  Versicolor       1.00      1.00      1.00         7
   Virginica       1.00      1.00      1.00        12

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

This code evaluates the performance of `best_model` on the test data:
Fit the Model: `best_model.fit(X_train, y_train)` fits the best logistic regression model found on the training data.
Make Predictions: `y_pred = best_model.predict(X_test)` generates predictions for the test set.
EvaluationMetrics:Using`accuracy_score`,`classification_report`,and `confusion_matrix` to assess model performance.
Classification Report: The `classification_report` provides precision, recall, and F1-score for each class (`Setosa`, `Versicolor`, `Virginica`), assuming `y_test` contains the true labels and `y_pred` the predicted ones.
This gives a detailed summary of model performance across classes.

- Model Performance Evaluation:

```
[19] from sklearn.metrics import classification_report, confusion_matrix
     print(confusion_matrix(y_test, y_pred))
     print(classification_report(y_test, y_pred))

[[11  0  0]
 [ 0  7  0]
 [ 0  0 12]]
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        11
           1       1.00      1.00      1.00         7
           2       1.00      1.00      1.00        12

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

Confusion Matrix: Shows how many predictions were correct or incorrect for each class, helping you see specific error types like false positives and false negatives.

Classification Report: Summarizes model performance with key metrics:
Precision: Measures accuracy of positive predictions.
Recall: Measures how well all actual positives were identified.
F1 Score: Balances precision and recall, especially useful if classes are imbalanced.

These metrics help assess and compare your model's performance.

- pickle library:

```
[20] import pickle

     # Save the model to a file
     with open('iris.pkl', 'wb') as file:
         pickle.dump(model, file)
```

This code is about saving your trained model so you can use it later without retraining

Opening a File: `with open('iris.pkl', 'wb') as file:`
This line opens (or creates) a file called `iris.pkl` in "write binary" mode, which just means it's ready to store data in a way Python can read back later.
Saving the Model: `pickle.dump(model, file)`
This line takes your trained model and saves it in `iris.pkl`. Think of it as packing up your model and storing it in a box (the file) so you can unpack it anytime you want to use it again.

Now, anytime you want to use this model, you can just load it from `iris.pkl` instead of retraining from scratch!.

```
[29] with open('iris.pkl', 'rb') as file:
         saved_objects = pickle.load(file)
         model = saved_objects['model']
         scaler = saved_objects['scaler']
```

This code loads both a saved model and scaler from `iris.pkl`:

Open the file: Opens `iris.pkl` in read mode.
Load the objects: Retrieves both the `model` and `scaler` saved in a dictionary.
Assign: `model` and `scaler` are ready to use.

This setup ensures your preprocessing (scaling) matches the model's training.

```
with open('iris.pkl', 'rb') as file:
    saved_objects = pickle.load(file)
    print(type(saved_objects))
print("Input data shape:", data.shape)
```

```
<class 'dict'>
Input data shape: (150, 6)
```

This code loads the contents of `iris.pkl` and prints the type of the saved object.

Opening the File: `with open('iris.pkl', 'rb') as file:` Opens the file `iris.pkl` in read mode.
Loading the Object: `saved_objects = pickle.load(file)`Loads the contents of `iris.pkl` into `saved_objects`.
Printing the Type: `print(type(saved_objects))`Prints the type of `saved_objects`. In this case, it should print `<class 'dict'>` because you saved both the model and scaler in a dictionary.

This is a quick way to check that the loaded data matches the expected format.

- VS Code and Streamlit Implementation:

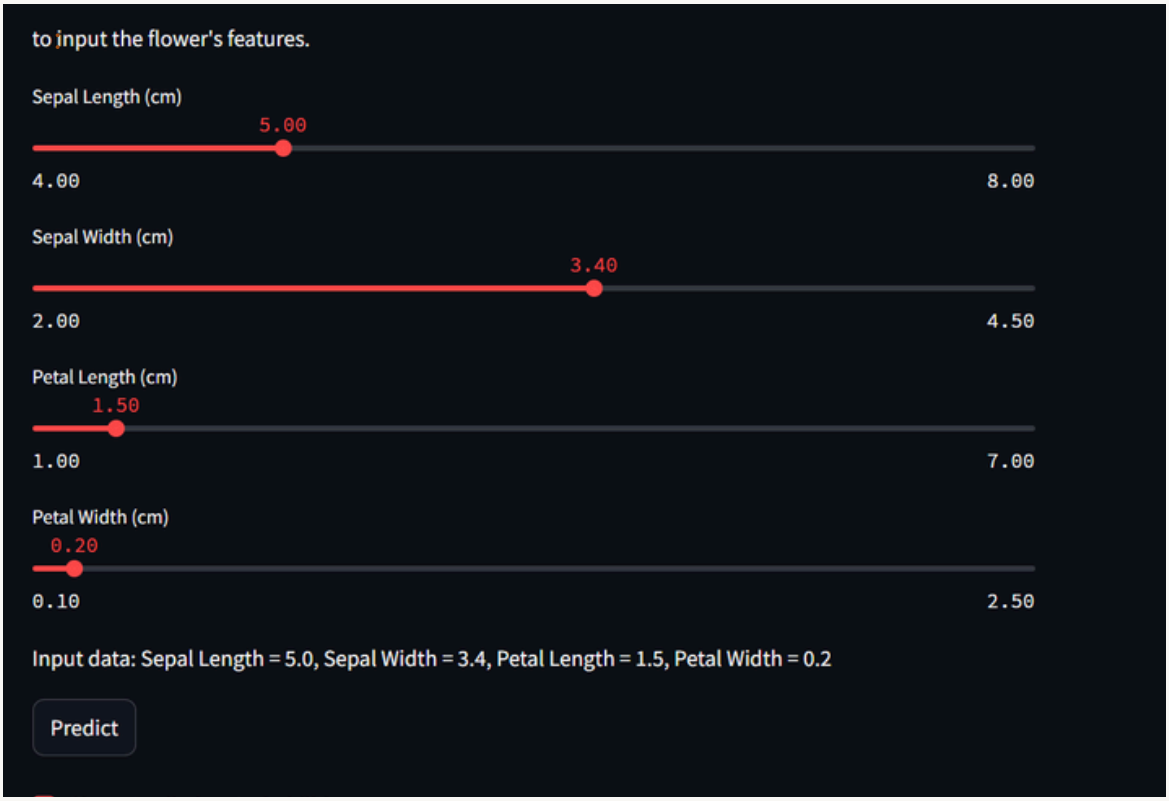
.

```
import streamlit as st
import numpy as np
import pandas as pd
import pickle

# Load the model and preprocessing objects
@st.cache_resource  # Cache the loading function to speed up the app
def load_model():
    try:
        # Attempt to load the model and preprocessing objects from the pickle file
        with open('iris_model.pkl', 'rb') as file:
            saved_objects = pickle.load(file)

        # Ensure all necessary objects are loaded from the pickle file
        model = saved_objects['model']
        scaler = saved_objects.get('scaler')
        pca = saved_objects.get('pca')

        # Debug print to check loaded objects
        print("Model, Scaler, and PCA loaded successfully!")

        return model, scaler, pca
    except Exception as e:
        # Display any errors that occur during the loading process
        st.error(f"Error loading model: {e}")
        print(f"Error loading model: {e}")
        return None, None, None

model, scaler, pca = load_model()

if model is None:
    st.stop()  # Stop execution if the model couldn't be loaded

# Define the Streamlit application
def main():
    # Set up the application title and description
    st.title("Iris Flower Classification App")
```

```python
    st.write("""
        This application predicts the species of an Iris flower based on its measurements.
        Adjust the sliders below to input the flower's features.
    """)

    # User input sliders for flower features
    sepal_length = st.slider("Sepal Length (cm)", min_value=4.0, max_value=8.0, value=5.4, step=0.1)
    sepal_width = st.slider("Sepal Width (cm)", min_value=2.0, max_value=4.5, value=3.4, step=0.1)
    petal_length = st.slider("Petal Length (cm)", min_value=1.0, max_value=7.0, value=3.8, step=0.1)
    petal_width = st.slider("Petal Width (cm)", min_value=0.1, max_value=2.5, value=1.2, step=0.1)

    # Collect inputs into an array
    input_data = np.array([[sepal_length, sepal_width, petal_length, petal_width]])

    # Debugging: Check the shape of input data
    print(f"Input data shape: {input_data.shape}")

    # Add a dummy feature (if required by the model)
    dummy_feature = np.ones((input_data.shape[0], 1))  # Add a constant feature (e.g., 1)
    input_data = np.hstack((input_data, dummy_feature))  # Combine input data with dummy feature

    # Debugging: Check input data after adding the dummy feature
    print(f"Input data after adding dummy feature: {input_data.shape}")

    # Preprocess the input data (scaling and PCA if applicable)
    if scaler:
        try:
            input_data = scaler.transform(input_data)
            print("Scaling successful!")
        except Exception as e:
            st.error(f"Error during scaling: {e}")
            print(f"Error during scaling: {e}")
            return

    if pca:
        try:
            input_data = pca.transform(input_data)
            print("PCA transformation successful!")
        except Exception as e:
            st.error(f"Error during PCA transformation: {e}")
            print(f"Error during PCA transformation: {e}")
            return

    # Show the input data for confirmation
    st.write(f"Input data: Sepal Length = {sepal_length}, Sepal Width = {sepal_width}, Petal Length = {petal_length}, Petal

    # Predict the species using the model
    if st.button("Predict"):
        try:
            prediction = model.predict(input_data)
            species = ["Setosa", "Versicolor", "Virginica"]
            st.write(f"The predicted species is: **{species[prediction[0]]}**")
            print(f"Prediction: {species[prediction[0]]}")
        except Exception as e:
            st.error(f"Error during prediction: {e}")
            print(f"Error during prediction: {e}")

    # Display prediction probabilities if checkbox is selected
    if st.checkbox("Show Prediction Probabilities"):
        try:
            probabilities = model.predict_proba(input_data)
            prob_df = pd.DataFrame(probabilities, columns=species)
            st.write("Prediction Probabilities:")
            st.write(prob_df)
            print("Prediction probabilities displayed.")
        except Exception as e:
            st.error(f"Error during probability calculation: {e}")
            print(f"Error during probability calculation: {e}")

# Run the app
if __name__ == '__main__':
    main()
```

The frontend and backend of this project were implemented using VS Code and Streamlit. The Streamlit app allows users to input new measurements and get predictions for the Iris flower species. Below is an indication for placing the screenshots of the output from the browser, showing how the app interacts with the user and the results generated.

- Host Screen:



- conclusion :

in this project, we explored have Logistic Regression. The model performed the best, achieving an accuracy of 0.98 on the test set. This project demonstrates the application of machine learning to solve a classic classification problem and how to persist models for future use. The Iris dataset remains a valuable resource for learning and experimenting with classification models, and this project can serve as a foundation for more complex modeling tasks