logistic-regression-implementation

May 21, 2024

Exploratory Data Analysis (EDA) and Implemention of Logistic Regression on Iris Dataset The Iris dataset is a classic dataset in machine learning and contains features of iris flowers along with their species.

Libraries Used:

- NumPy (np): A library for numerical computations in Python.
- Pandas (pd): A powerful data analysis and manipulation library.
- Seaborn (sns): A statistical data visualization library based on Matplotlib.
- Matplotlib.pyplot (plt): A plotting library for creating static, interactive, and animated visualizations.
- Sklearn.datasets: Provides utilities to load datasets from scikit-learn.

Steps:

Load the Iris Dataset: Load the Iris dataset using scikit-learn's datasets.load_iris() function.

Create a DataFrame: Convert the dataset into a Pandas DataFrame for easier analysis and manipulation.

Basic Statistics: Print basic statistics (count, mean, std, min, max, etc.) for each feature using DataFrame.describe().

Visualizations: Visualize the data using pair plots to explore relationships between features, categorized by species.

Analyze Species Distribution: Print the distribution of species in the dataset.

Correlation Analysis: Calculate the correlation matrix between features and plot it as a heatmap to identify relationships between variables.

[8]:
```
pip install numpy pandas matplotlib seaborn scikit-learn
```

```
Requirement already satisfied: numpy in
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
(1.26.4)
Requirement already satisfied: pandas in
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
(2.2.2)
Requirement already satisfied: matplotlib in
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
(3.9.0)
Requirement already satisfied: seaborn in
```

```
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
(0.13.2)
Collecting scikit-learn
  Downloading scikit_learn-1.5.0-cp312-cp312-macosx_12_0_arm64.whl.metadata (11
kB)
Requirement already satisfied: python-dateutil>=2.8.2 in
/Users/varun/Library/Python/3.12/lib/python/site-packages (from pandas)
(2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
(from pandas) (2024.1)
Requirement already satisfied: tzdata>=2022.7 in
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
(from pandas) (2024.1)
Requirement already satisfied: contourpy>=1.0.1 in
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
(from matplotlib) (1.2.1)
Requirement already satisfied: cycler>=0.10 in
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
(from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
(from matplotlib) (4.51.0)
Requirement already satisfied: kiwisolver>=1.3.1 in
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
(from matplotlib) (1.4.5)
Requirement already satisfied: packaging>=20.0 in
/Users/varun/Library/Python/3.12/lib/python/site-packages (from matplotlib)
(24.0)
Requirement already satisfied: pillow>=8 in
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
(from matplotlib) (10.3.0)
Requirement already satisfied: pyparsing>=2.3.1 in
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
(from matplotlib) (3.1.2)
Collecting scipy>=1.6.0 (from scikit-learn)
  Downloading scipy-1.13.0-cp312-cp312-macosx_12_0_arm64.whl.metadata (60 kB)
                           60.6/60.6 kB
695.8 kB/s eta 0:00:00a 0:00:01
Collecting joblib>=1.2.0 (from scikit-learn)
  Downloading joblib-1.4.2-py3-none-any.whl.metadata (5.4 kB)
Collecting threadpoolctl>=3.1.0 (from scikit-learn)
  Downloading threadpoolctl-3.5.0-py3-none-any.whl.metadata (13 kB)
Requirement already satisfied: six>=1.5 in
/Users/varun/Library/Python/3.12/lib/python/site-packages (from python-
dateutil>=2.8.2->pandas) (1.16.0)
Downloading scikit_learn-1.5.0-cp312-cp312-macosx_12_0_arm64.whl (11.0 MB)
                         11.0/11.0 MB
```

```
1.2 MB/s eta 0:00:0000:0100:01m
Downloading joblib-1.4.2-py3-none-any.whl (301 kB)
                          301.8/301.8 kB
6.4 MB/s eta 0:00:00a 0:00:01
Downloading scipy-1.13.0-cp312-cp312-macosx_12_0_arm64.whl (30.4 MB)
                          30.4/30.4 MB
14.2 MB/s eta 0:00:0000:0100:01
Downloading threadpoolctl-3.5.0-py3-none-any.whl (18 kB)
Installing collected packages: threadpoolctl, scipy, joblib, scikit-learn
Successfully installed joblib-1.4.2 scikit-learn-1.5.0 scipy-1.13.0
threadpoolctl-3.5.0
Note: you may need to restart the kernel to use updated packages.
```

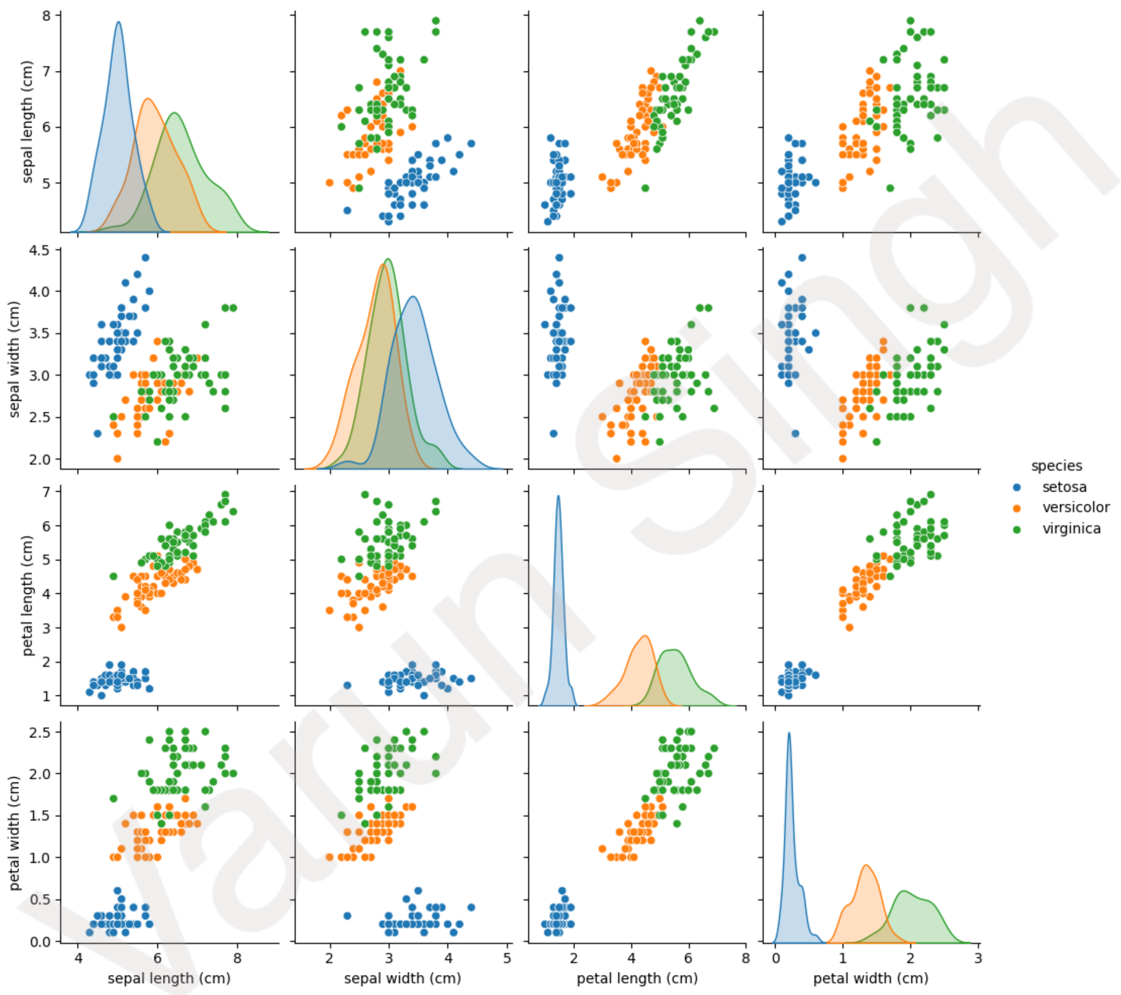
[9]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Step 1: Load the Iris Dataset
iris = datasets.load_iris()
X = iris.data
y = iris.target
feature_names = iris.feature_names
target_names = iris.target_names
```

[ ]:
```
# Step 2: Create a DataFrame for easier analysis
df = pd.DataFrame(data=np.c_[X, y], columns=feature_names + ['species'])
df['species'] = df['species'].map({i: target_names[i] for i in␣
 ↪range(len(target_names))})

# Step 3: Basic Statistics
print(df.describe())
```

```
       sepal length (cm)  sepal width (cm)  petal length (cm)  \
count         150.000000        150.000000         150.000000
mean            5.843333          3.057333           3.758000
std             0.828066          0.435866           1.765298
min             4.300000          2.000000           1.000000
25%             5.100000          2.800000           1.600000
50%             5.800000          3.000000           4.350000
75%             6.400000          3.300000           5.100000
max             7.900000          4.400000           6.900000

       petal width (cm)
count        150.000000
```

```
mean          1.199333
std           0.762238
min           0.100000
25%           0.300000
50%           1.300000
75%           1.800000
max           2.500000
```

[ ]:
```
# Step 4: Visualizations
sns.pairplot(df, hue='species')
plt.show()
```

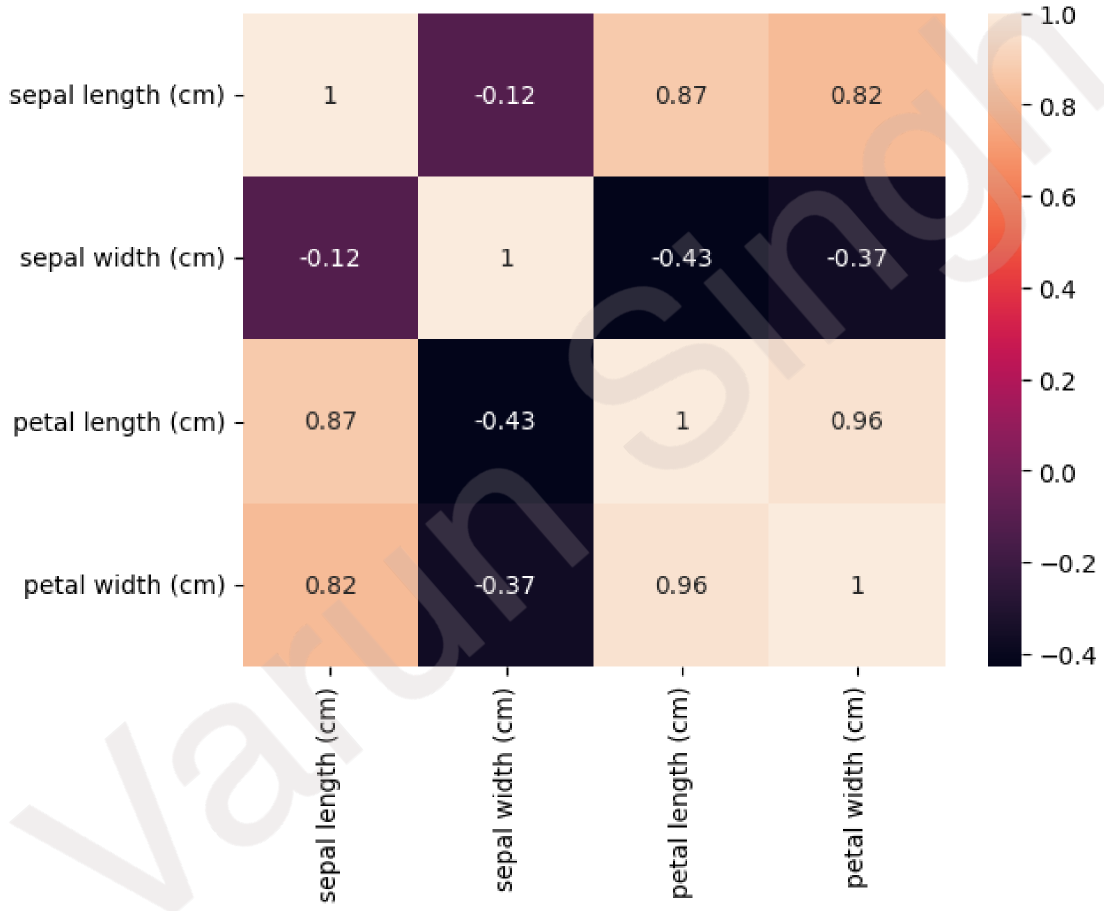


[ ]:
```
# Step 5: Analyze Species Distribution
species_counts = df['species'].value_counts()
print(species_counts)
```

```
species
```

```
setosa        50
versicolor    50
virginica     50
Name: count, dtype: int64
```

[ ]:
```python
# Step 6: Correlation Analysis
# Exclude non-numeric column ('species') before calculating correlation
numeric_df = df.drop('species', axis=1)
correlation_matrix = numeric_df.corr()

# Plot the correlation matrix as a heatmap
sns.heatmap(correlation_matrix, annot=True)
plt.show()
```



Insight of the EDA 1. Based on the pair plots and correlation analysis, it appears that the features 'petal length' and 'petal width' exhibit significant separability between different species.

2. Setosa species tends to have smaller petal lengths and widths compared to Versicolor and Virginica.

3. Versicolor and Virginica species exhibit more overlap in their distributions, but still, there are discernible differences, especially in petal length and width.
4. Based on these insights, one could consider building a classification model using 'petal length' and 'petal width' as features for distinguishing between species.
5. Additional feature engineering or dimensionality reduction techniques could be explored to enhance model performance.

0.0.1 Binary Classification

##Methodology: 1. **Loading the Dataset**: The Iris dataset is loaded using the `datasets.load_iris()` function provided by scikit-learn. 2. **Subset Selection**: Since the goal is to perform binary classification, only a subset of the dataset is used. Specifically, the first 100 samples are selected. 3. **Data Assignment**: - `X`: Contains the feature data for the selected subset. - `y`: Contains the target labels corresponding to the selected subset.

0.0.2 Parameters:

- **iris**: This variable holds the loaded Iris dataset, which includes both feature data and target labels.
- **X**: Represents the feature data, which is a 2D array of shape (100, n_features), where n_features is the number of features.
- **y**: Represents the target labels, which is a 1D array of shape (100,) containing integers representing the class labels.

0.0.3 Notes:

- The Iris dataset consists of 150 samples, each representing a different iris flower. There are 3 classes (species) in total: Setosa, Versicolor, and Virginica.
- By selecting the first 100 samples, only two species (Setosa and Versicolor) are considered for binary classification, simplifying the problem.

[ ]:
```
# Load the Iris Dataset
iris = datasets.load_iris()
X = iris.data[:100]  # Using only two species for binary classification
y = iris.target[:100]
```

0.1 Preprocess the Data

0.1.1 Purpose:

The purpose of this step is to preprocess the dataset for binary classification, specifically distinguishing between the Setosa and Versicolour species.

0.1.2 Methodology:

1. **Data Splitting**: The dataset is split into training and testing sets using the `train_test_split()` function from scikit-learn. This step ensures that the model's performance can be evaluated on unseen data.

2. **Feature Scaling**: Standardization is applied to the feature data using the `StandardScaler()` from scikit-learn. This step ensures that all features have a mean of 0 and a standard deviation of 1, which can improve the performance of certain machine learning algorithms.

0.1.3 Parameters:

- **X_train, X_test**: Feature data for the training and testing sets, respectively.
- **y_train, y_test**: Target labels for the training and testing sets, respectively.
- **test_size**: Proportion of the dataset to include in the testing set. Here, 20% of the data is used for testing.
- **random_state**: Seed used by the random number generator for reproducibility.

0.1.4 Notes:

- Binary classification is performed to distinguish between the Setosa (0) and Versicolour (1) species.
- The dataset is split into training and testing sets to evaluate the model's performance.
- Feature scaling ensures that all features have the same scale, which can be beneficial for certain machine learning algorithms, such as SVMs and KNN.

[ ]:
```
# Preprocess the Data
# Binary classification: Setosa (0) or Versicolour (1)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Feature scaling
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

0.2 Implement Logistic Regression

0.2.1 Purpose:

The purpose of this step is to implement logistic regression, a popular machine learning algorithm used for binary classification tasks.

0.2.2 Methodology:

1. **Sigmoid Function**: The `sigmoid()` function is implemented to map the output of the linear combination of features to the range [0, 1], representing the probability of belonging to the positive class.
2. **Cost Function**: The `cost_function()` calculates the logistic regression cost (or loss) for a given set of weights, by computing the cross-entropy loss between the predicted probabilities and the actual labels.
3. **Gradient Descent**: The `gradient_descent()` function performs gradient descent optimization to minimize the cost function. It updates the weights iteratively in the direction that reduces the cost.

0.2.3 Parameters:

- **X**: Feature matrix (input data).
- **y**: Target labels.
- **weights**: Initial weights for logistic regression.
- **learning_rate**: Hyperparameter controlling the step size in the gradient descent algorithm.
- **iterations**: Number of iterations for gradient descent.

0.2.4 Functions:

1. **sigmoid(z)**:
    - **Input**: z (linear combination of features)
    - **Output**: Sigmoid transformation of z
2. **cost_function(X, y, weights)**:
    - **Input**: Feature matrix X, target labels y, weights
    - **Output**: Logistic regression cost
3. **gradient_descent(X, y, weights, learning_rate, iterations)**:
    - **Input**: Feature matrix X, target labels y, initial weights, learning rate, number of iterations
    - **Output**: Updated weights after gradient descent, cost history over iterations

0.2.5 Notes:

- Logistic regression is a linear classifier that models the probability of the positive class.
- Gradient descent is used to optimize the weights by minimizing the logistic regression cost.

[ ]:
```python
# Step 3: Implement Logistic Regression
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def cost_function(X, y, weights):
    m = len(y)
    h = sigmoid(X.dot(weights))
    cost = (-y).dot(np.log(h)) - (1 - y).dot(np.log(1 - h))
    return cost / m

def gradient_descent(X, y, weights, learning_rate, iterations):
    m = len(y)
    cost_history = np.zeros(iterations)

    for i in range(iterations):
        weights = weights - (learning_rate/m) * X.T.dot(sigmoid(X.dot(weights))␣
↪- y)
        cost_history[i] = cost_function(X, y, weights)

    return weights, cost_history
```

0.3 Train the Model

0.3.1 Purpose:

The purpose of this step is to train the logistic regression model using gradient descent optimization.

0.3.2 Methodology:

1. **Data Preparation**:
   - **Add Bias Term**: A bias term is added to the feature matrix to account for the intercept term in the linear equation.
2. **Initialization**:
   - **Weights Initialization**: Initial weights are set to zero for all features, including the bias term.
3. **Training**:
   - **Gradient Descent**: The logistic regression model is trained using the `gradient_descent()` function, which iteratively updates the weights to minimize the cost function.

0.3.3 Parameters:

- **X_train_scaled**: Feature matrix for the training set after feature scaling.
- **y_train**: Target labels for the training set.
- **weights**: Initial weights for logistic regression, including the bias term.
- **iterations**: Number of iterations for gradient descent.
- **learning_rate**: Hyperparameter controlling the step size in gradient descent.

0.3.4 Notes:

- The logistic regression model is trained using gradient descent optimization to minimize the logistic regression cost.
- Hyperparameters such as the number of iterations and learning rate need to be tuned to achieve optimal performance.

[ ]:
```
# Step 4: Train the Model
m, n = X_train_scaled.shape
X_train_with_bias = np.append(np.ones((m, 1)), X_train_scaled, axis=1)  # Add␣
↪bias term
weights = np.zeros(n + 1)
iterations = 2000
learning_rate = 0.1

weights, cost_history = gradient_descent(X_train_with_bias, y_train, weights,␣
↪learning_rate, iterations)
```

0.4 Predict and Evaluate the Model

0.4.1 Purpose:

The purpose of this step is to use the trained logistic regression model to make predictions on the test data and evaluate its performance.

0.4.2 Methodology:

1. **Prediction**:
   - The `predict()` function is used to predict the target labels for the test data based on the learned weights from the trained model.
   - Predictions are made by applying the sigmoid function to the linear combination of features and weights. If the predicted probability is greater than 0.5, the predicted label is set to 1; otherwise, it's set to 0.
2. **Evaluation**:
   - Accuracy is calculated by comparing the predicted labels with the true labels from the test set.

0.4.3 Parameters:

- **X_test_scaled**: Feature matrix for the test set after feature scaling.
- **weights**: Learned weights from the trained logistic regression model.

0.4.4 Functions:

- **predict(X, weights)**:
  - **Input**: Feature matrix X, learned weights
  - **Output**: Predicted target labels based on the logistic regression model.

0.4.5 Notes:

- The logistic regression model predicts binary labels (0 or 1) for the test set based on learned weights.
- Accuracy is a commonly used metric to evaluate the performance of binary classification models, representing the proportion of correctly predicted labels.

[ ]:
```
def predict(X, weights):
    X_with_bias = np.append(np.ones((X.shape[0], 1)), X, axis=1)
    return [1 if i > 0.5 else 0 for i in sigmoid(X_with_bias.dot(weights))]

y_pred = predict(X_test_scaled, weights)

# Calculate accuracy
accuracy = np.mean(y_pred == y_test)
print(f'Accuracy: {accuracy * 100:.2f}%')
```

```
Accuracy: 100.00%
```