linear-regression

November 19, 2024

1 Predicting House Prices - Linear Regression

Linear Regression in machine learning is a supervised algorithm that models the relationship between a dependent variable and one or more independent variables by fitting a straight line to minimize prediction errors.

1.0.1 Importing necssary modules

We will use `numpy` for numerical operations and `pandas` for handling and manipulating data efficiently.

[2]:
```
import numpy as np
import pandas as pd
```

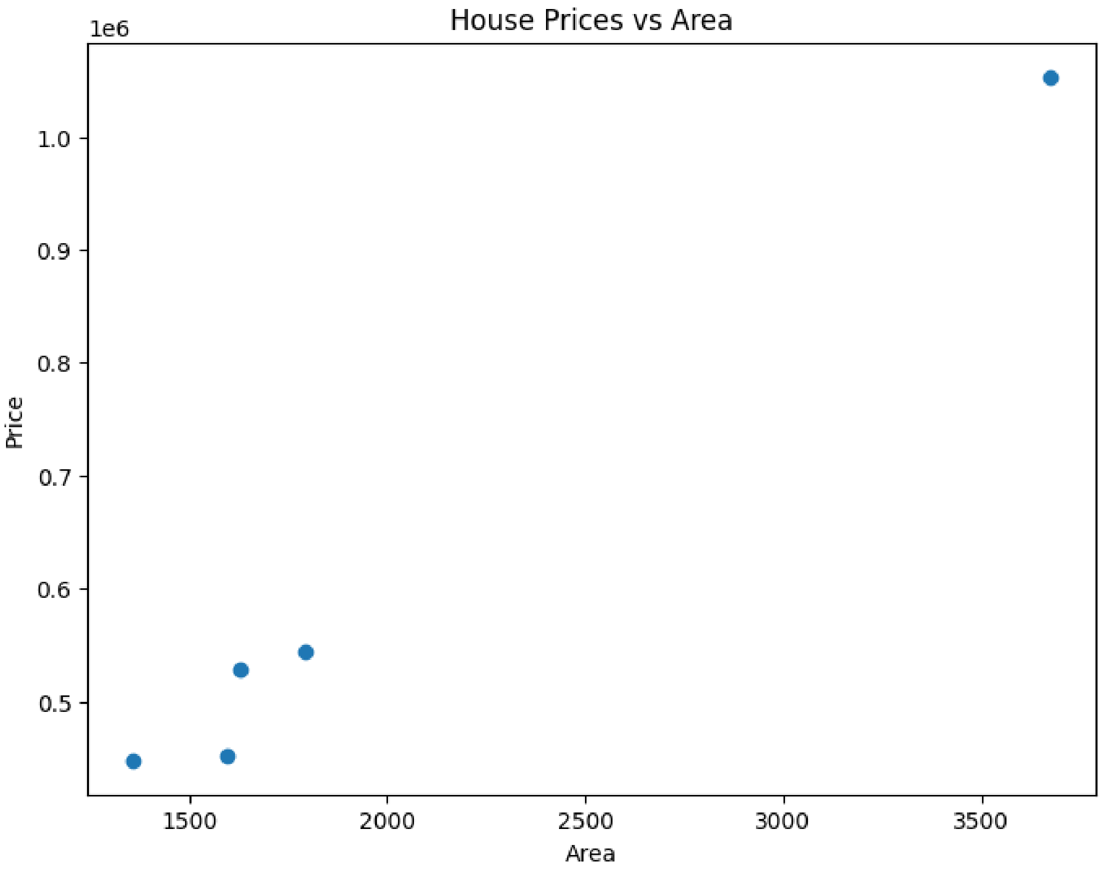
[3]:
```
data = {'area': [3674,1360,1794,1630,1595],
        'price': [1053216,447789,543791,528812,451747]}

df = pd.DataFrame(data)
df.head()
```

[3]:
```
   area    price
0  3674  1053216
1  1360   447789
2  1794   543791
3  1630   528812
4  1595   451747
```

1.0.2 Plotting Data

Creating scatter plot of house price data and Here, we will create a scatter plot to visualize the relationship between the house area and its price. This helps us understand the data before applying linear regression.

[4]:
```
import matplotlib.pyplot as plt

plt.figure(figsize=(8,6))
plt.scatter(df['area'], df['price'],label='Data Points')
plt.xlabel('Area')
plt.ylabel('Price')
plt.title('House Prices vs Area')
plt.show()
```



1.0.3 Building the Linear Regression Model

We now import the `LinearRegression` model from `sklearn` and fit the model to our dataset.

[5]:
```
from sklearn.linear_model import LinearRegression

X = df[['area']]
y = df.price

model = LinearRegression()
model.fit(X, y)
```

[5]:
```
LinearRegression()
```

[9]:
```
model.predict([[4000]])
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/base.py:493: UserWarning: X does
not have valid feature names, but LinearRegression was fitted with feature names
  warnings.warn(
```

```
[9]: array([1138867.08242013])
```

1.0.4 Model Evaluation

Now, we will check the model's performance by calculating the R^2 score (coefficient of determination) and also make predictions for the given data points.

[ ]:

```
predicted_prices = model.predict(X)
score = model.score(X, y)

print("Score:", score)
print("Predicted Prices:", predicted_prices)
```

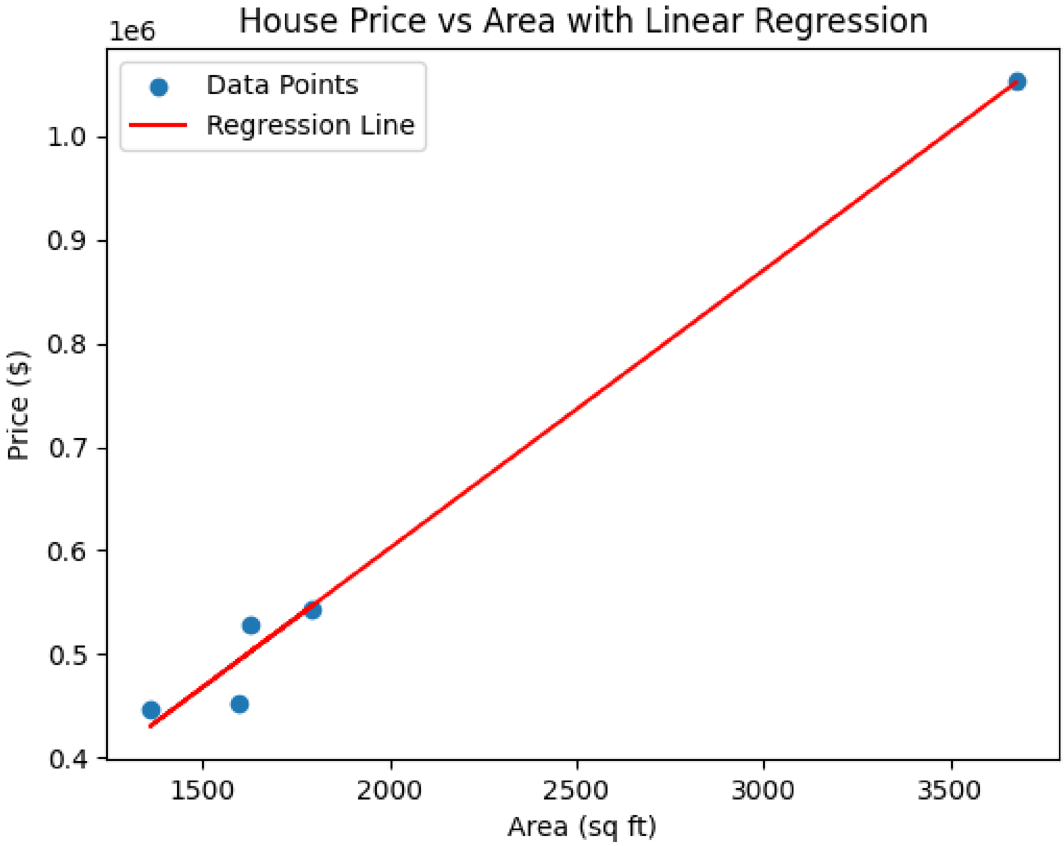
```
Score: 0.9894484081295886
Predicted Prices: [1051394.71745132  430501.91825549  546952.85812195
502948.3554996
  493557.15067166]
```

1.0.5 Plotting the Regression Line

We will now overlay the regression line on the scatter plot to visually represent the linear relationship between house area and price.

[ ]:

```
plt.scatter(df['area'], df['price'],label='Data Points')

plt.plot(df['area'], predicted_prices, color='red', label='Regression Line')

plt.xlabel('Area (sq ft)')
plt.ylabel('Price ($)')
plt.title('House Price vs Area with Linear Regression')


plt.legend()
plt.show()
```


House Price vs Area with Linear Regression
1e6
Data Points
Regression Line
1.0
0.9
0.8
0.7
0.6
0.5
0.4
Price ($)
1500
2000
2500
3000
3500
Area (sq ft)

multivariate-linear-regression

November 19, 2024

1 Multivariate Linear Regression

This notebook demonstrates multivariate linear regression using synthetic data generated with three independent variables. We'll split the data into training and testing sets, train the model, evaluate its performance, and visualize the results.

[15]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
```

1.0.1 Generate the dataset

[9]:

```
# Set random seed for reproducibility
np.random.seed(42)

# Create 100 samples with 3 independent variables
X = np.random.rand(100, 3)

# Target variable (dependent variable) with noise added
y = 4 + 3 * X[:, 0] + 2 * X[:, 1] + X[:, 2] + np.random.randn(100)
```

1.0.2 Splitting the dataset into training and testing sets

We'll use 70% of the data for training and 30% for testing.

[3]:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=42)
```

1.0.3 Train the Linear Regression model

We will fit a linear regression model using the training data.

[10]:

```
# Initialize and train the model
lr_model = LinearRegression()
lr_model.fit(X_train, y_train)
```

[10]:

```
LinearRegression()
```

1.0.4 Model Predictions

We now predict the target variable for the test set using the trained model.

[5]:

```python
# Predicting on the test set
y_pred = lr_model.predict(X_test)
```

1.0.5 Model Evaluation

We'll evaluate the model's performance using: - R^2 Score: Indicates how well the independent variables explain the variance of the target variable. - Mean Squared Error (MSE): Measures the average squared difference between actual and predicted values.

[11]:

```python
# Calculate R² score
r2_score = lr_model.score(X_test, y_test)
print(f"R² Score: {r2_score:.2f}")
```

```
R² Score: 0.45
```

[12]:

```python
# Calculate Mean Squared Error
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse:.2f}")
```

```
Mean Squared Error: 1.51
```

1.0.6 Model Coefficients

The coefficients (weights) indicate the relationship between each independent variable and the target variable.

[13]:

```python
# Print the intercept and coefficients
print("Intercept:", lr_model.intercept_)
print("Coefficients:", lr_model.coef_)
```

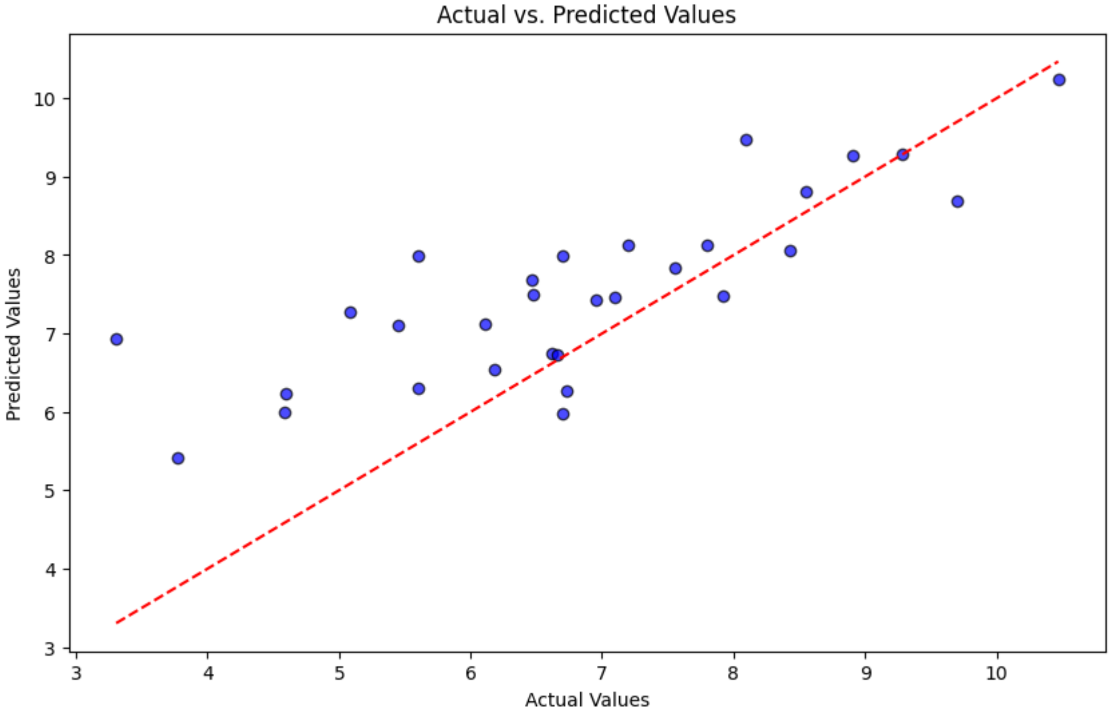
```
Intercept: 3.9203695786134714
Coefficients: [3.17059733 1.86545932 1.71097327]
```

1.0.7 Visualizing the Results

We will visualize the actual vs. predicted values to assess the model's performance.

[16]:

```python
# Plotting the Actual vs. Predicted values
plt.figure(figsize=(10,6))
plt.scatter(y_test, y_pred, color='blue', edgecolor='k', alpha=0.7)
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], color='red',␣
  ↪linestyle='--')
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Actual vs. Predicted Values')
plt.show()
```

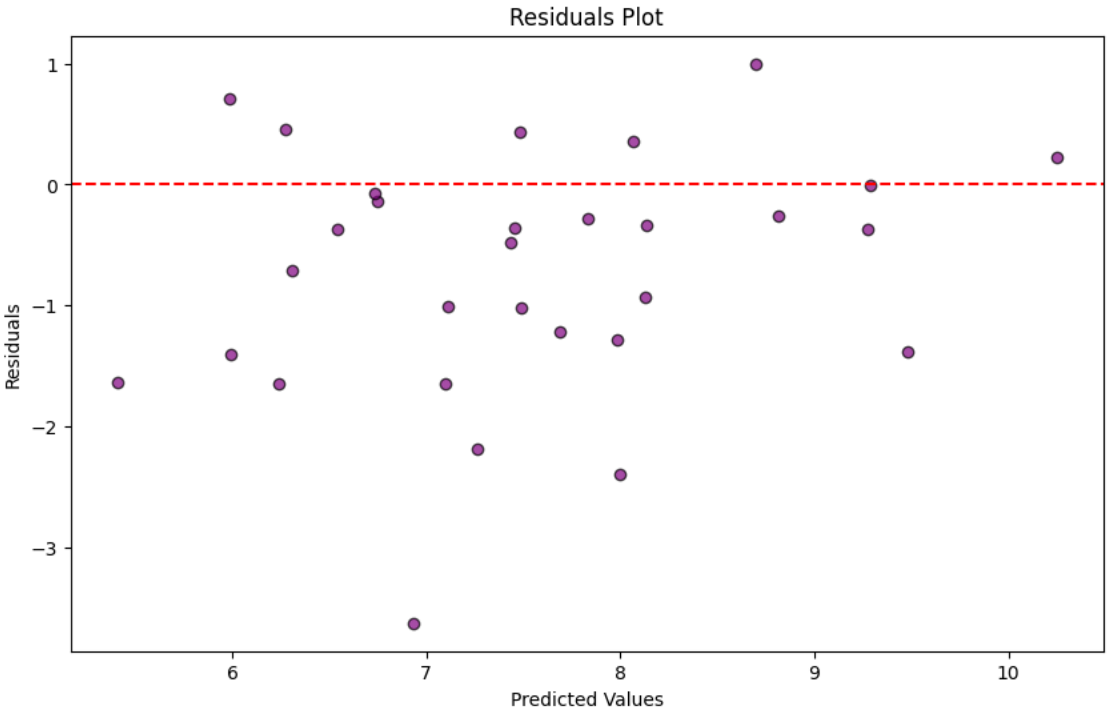


1.0.8 Residuals Plot

A residuals plot helps to diagnose issues with the model (e.g., non-linearity, outliers).

[17]:

```
# Plotting residuals
residuals = y_test - y_pred

plt.figure(figsize=(10,6))
plt.scatter(y_pred, residuals, color='purple', edgecolor='k', alpha=0.7)
plt.axhline(0, color='red', linestyle='--')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.title('Residuals Plot')
plt.show()
```


Residuals Plot
1
0
−1
−2
−3
Residuals
6
7
8
9
10
Predicted Values

logistic-regression

November 19, 2024

1 Predicting Flower Species - Logistic Regression

Logistic Regression is a supervised machine learning algorithm used for classification tasks. It models the probability that a given input belongs to a particular class.

1.0.1 Importing necessary libraries

We use `pandas` and `numpy` for data manipulation and `matplotlib` for visualization. The dataset is imported from `sklearn.datasets`.

[2]:

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix,␣
  ↪classification_report
```

[3]:

```python
# Load the iris dataset from sklearn
iris = datasets.load_iris()

# Create a DataFrame for easier handling
df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
df['target'] = iris.target
df.head()
```

[3]:

```
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

   target
0       0
1       0
2       0
```

```
3       0
4       0
```

[ ]:

```
# Filter out two of the species (versicolor and virginica), keeping only setosa (class 0) and versicolor (class 1)
df = df[df['target'] != 2]  # Remove class 2 (virginica)

# Mapping species to human-readable labels
df['target'] = df['target'].map({0: 'setosa', 1: 'versicolor'})

df.head()
```

[ ]:

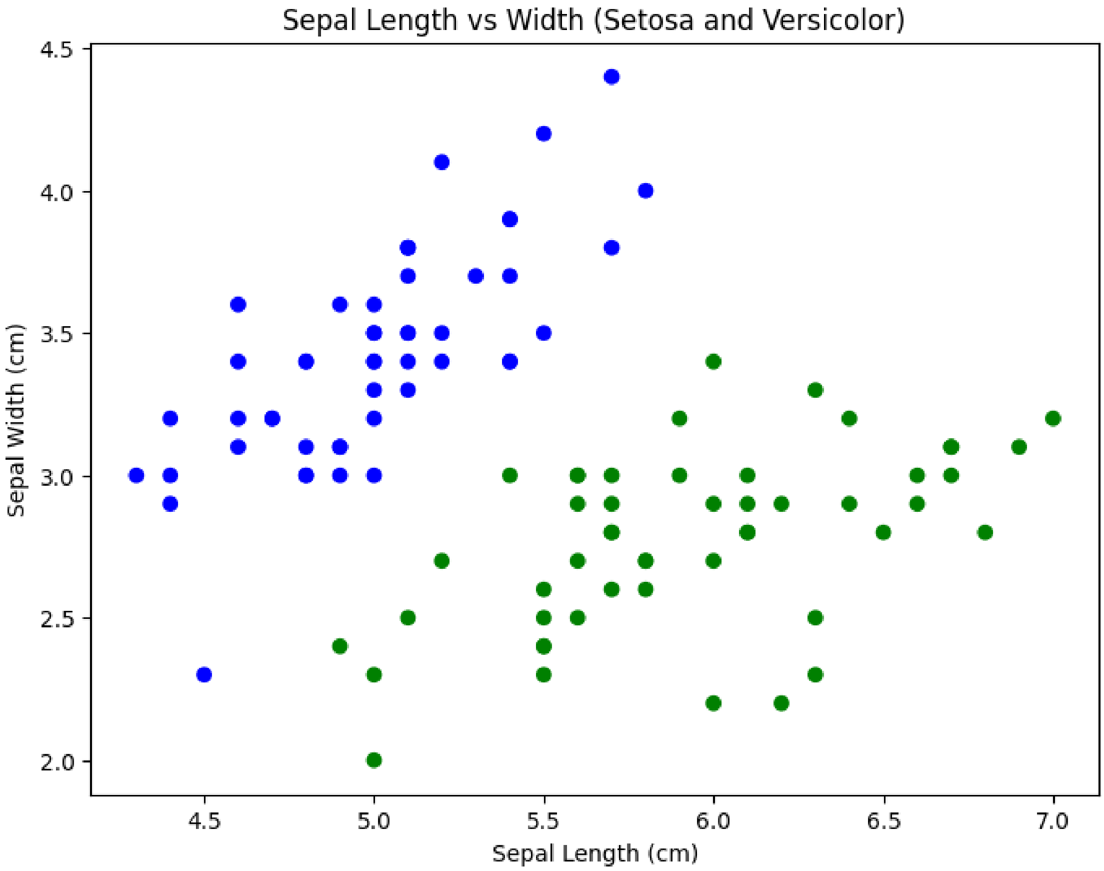
```
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

   target
0  setosa
1  setosa
2  setosa
3  setosa
4  setosa
```

1.0.2 Visualizing the data

We will create a scatter plot of two features: sepal length and sepal width to understand the relationship between them and the flower species.

[ ]:

```
# Plotting sepal length vs sepal width, color-coded by species
plt.figure(figsize=(8,6))
species_color = {'setosa': 'blue', 'versicolor': 'green'}
plt.scatter(df['sepal length (cm)'], df['sepal width (cm)'],
            c=df['target'].map(species_color), label='Data Points')
plt.xlabel('Sepal Length (cm)')
plt.ylabel('Sepal Width (cm)')
plt.title('Sepal Length vs Width (Setosa and Versicolor)')
plt.show()
```



1.0.3 Preparing the data

We will now split the data into features (X) and labels (y), and then into training and testing sets.

[ ]:
```
# Features (sepal length and width) and target (species)
X = df[['sepal length (cm)', 'sepal width (cm)']].values  # Independent␣
 ↪variables
y = df['target'].map({'setosa': 0, 'versicolor': 1}).values  # Dependent␣
 ↪variable (binary classification)

# Split the dataset into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
```

1.0.4 Building and Training the Logistic Regression Model

We will now create the logistic regression model and train it on the training data.

[ ]:

```
# Initialize the Logistic Regression model
log_model = LogisticRegression()

# Train the model using the training data
log_model.fit(X_train, y_train)
```

[ ]:

```
LogisticRegression()
```

1.0.5 Model Evaluation

We will evaluate the performance of the logistic regression model by predicting on the test data and calculating accuracy.

[ ]:

```
# Make predictions on the test set
y_pred = log_model.predict(X_test)

# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

# Generate a confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", conf_matrix)

# Detailed classification report
class_report = classification_report(y_test, y_pred)
print("Classification Report:\n", class_report)
```

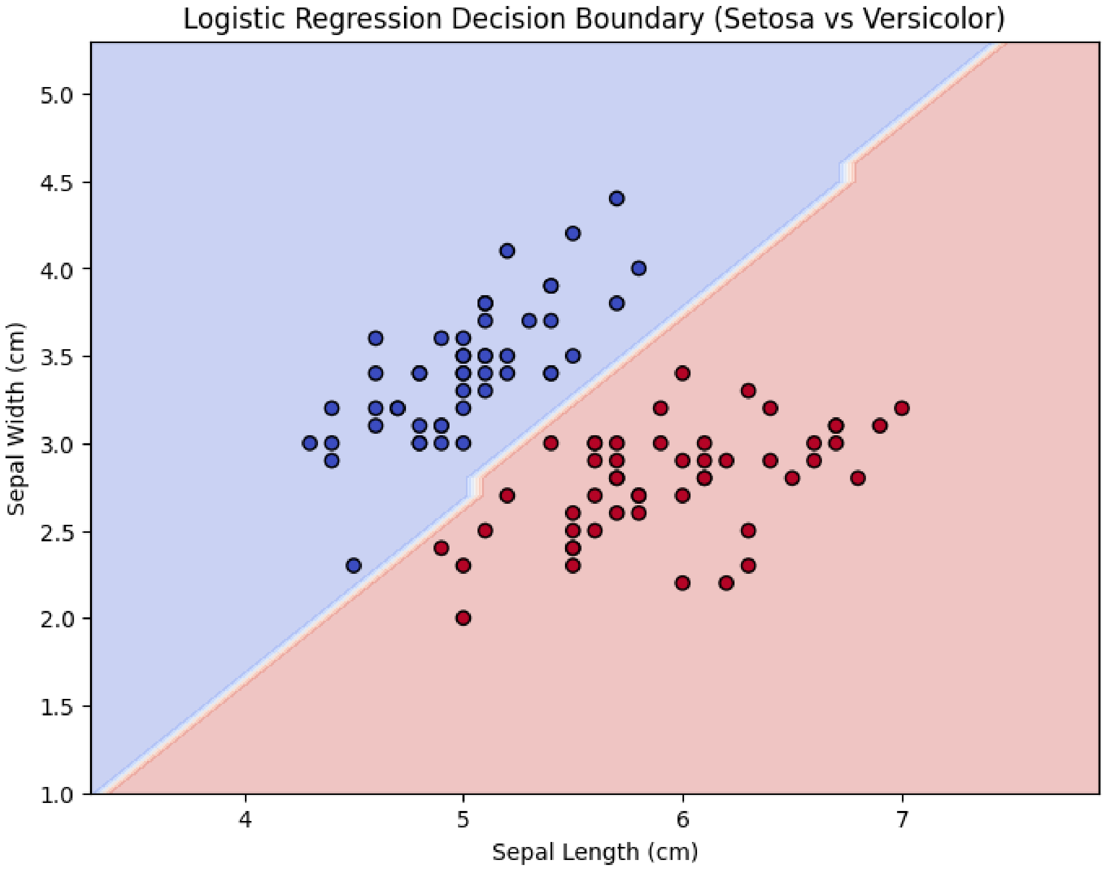
```
Accuracy: 1.0
Confusion Matrix:
 [[12  0]
 [ 0  8]]
Classification Report:
               precision    recall  f1-score   support

           0       1.00      1.00      1.00        12
           1       1.00      1.00      1.00         8

    accuracy                           1.00        20
   macro avg       1.00      1.00      1.00        20
weighted avg       1.00      1.00      1.00        20
```

1.0.6 Decision Boundary Plot

We will now visualize the decision boundary created by the logistic regression model.

[ ]:

```
# Create a mesh grid to plot the decision boundary
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))

# Use the model to predict the class for each point in the mesh grid
Z = log_model.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# Plot the decision boundary
plt.figure(figsize=(8,6))
plt.contourf(xx, yy, Z, alpha=0.3, cmap='coolwarm')
plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', cmap='coolwarm')
plt.xlabel('Sepal Length (cm)')
plt.ylabel('Sepal Width (cm)')
plt.title('Logistic Regression Decision Boundary (Setosa vs Versicolor)')
plt.show()
```



svm-irisdatset

November 19, 2024

1 SVM Classification with Iris Dataset

This notebook demonstrates the use of Support Vector Machines (SVM) to classify the Iris dataset. We will split the data, train an SVM classifier, and visualize the results, including decision boundaries and model evaluation.

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, confusion_matrix,␣
 ↪classification_report
```

1.0.1 Load the Iris dataset

The Iris dataset is a classic dataset used for classification problems. It contains 150 samples from three species of Iris flowers (Setosa, Versicolour, and Virginica) with four features: sepal length, sepal width, petal length, and petal width.

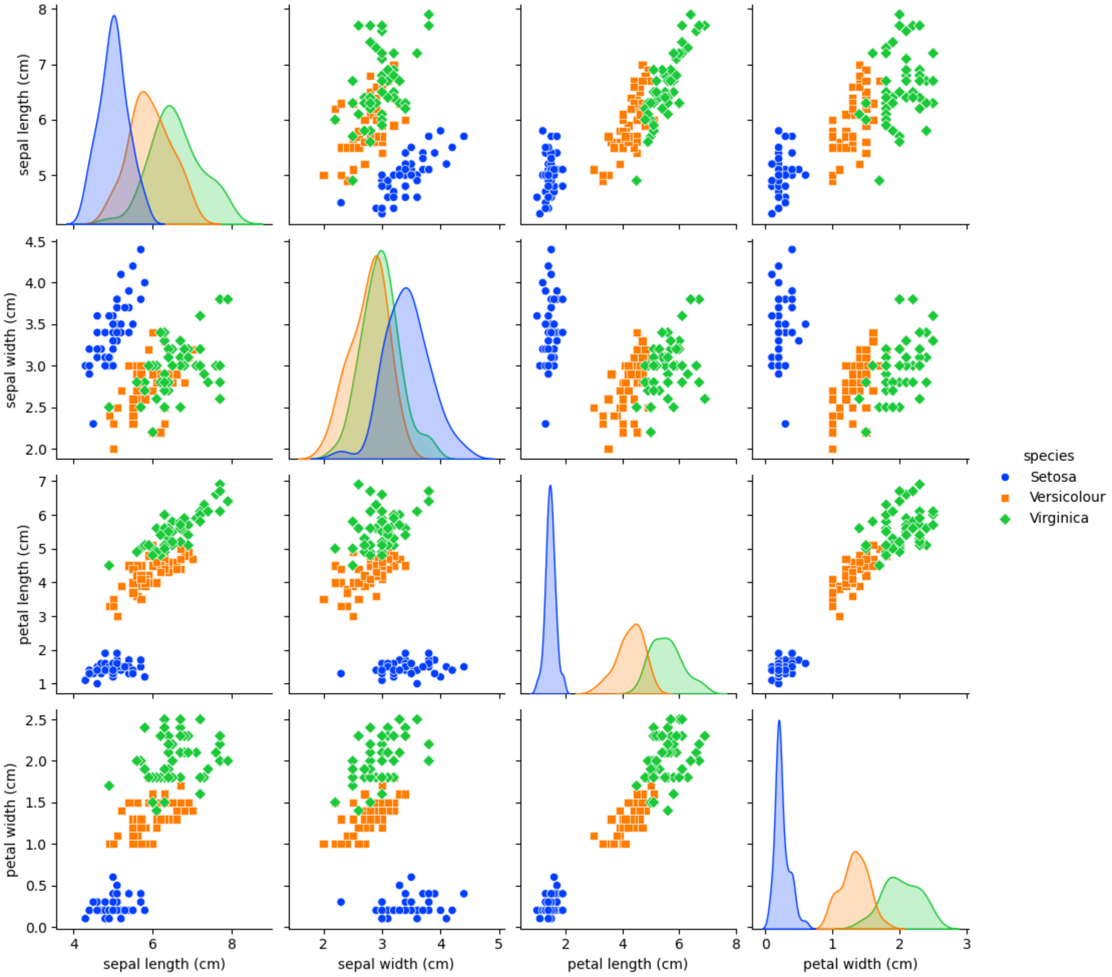
[4]:
```
# Load the iris dataset
iris = datasets.load_iris()

# Features and target labels
X = iris.data
y = iris.target

# Convert to a pandas DataFrame for better visualization
df_iris = pd.DataFrame(data=np.c_[X, y], columns=iris.feature_names +␣
 ↪['species'])
df_iris['species'] = df_iris['species'].map({0: 'Setosa', 1: 'Versicolour', 2:␣
 ↪'Virginica'})

# Display the first few rows of the dataset
df_iris.head()
```

[4]:
```
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

  species
0  Setosa
1  Setosa
2  Setosa
3  Setosa
4  Setosa
```

1.0.2 Exploratory Data Analysis (EDA)

Let's visualize the dataset with pairplots to understand how the features relate to each other and to the target variable.

[5]:
```python
# Pairplot for visualizing feature relationships
sns.pairplot(df_iris, hue='species', markers=["o", "s", "D"], palette="bright")
plt.show()
```



1.0.3 Splitting the dataset

We split the data into training and testing sets, with 70% used for training and 30% for testing.

[7]:

```
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42, stratify=y)
```

1.0.4 SVM Model Training

We use the SVM classifier with a **linear kernel** to classify the Iris dataset.

[8]:

```
# Create the SVM model with a linear kernel
svm_model = SVC(kernel='linear', random_state=42)

# Train the SVM model
svm_model.fit(X_train, y_train)
```

[8]:

```
SVC(kernel='linear', random_state=42)
```

1.0.5 Model Predictions

We predict the species of the Iris flowers in the test set using the trained SVM model.

[9]:

```
# Make predictions on the test set
y_pred = svm_model.predict(X_test)
```

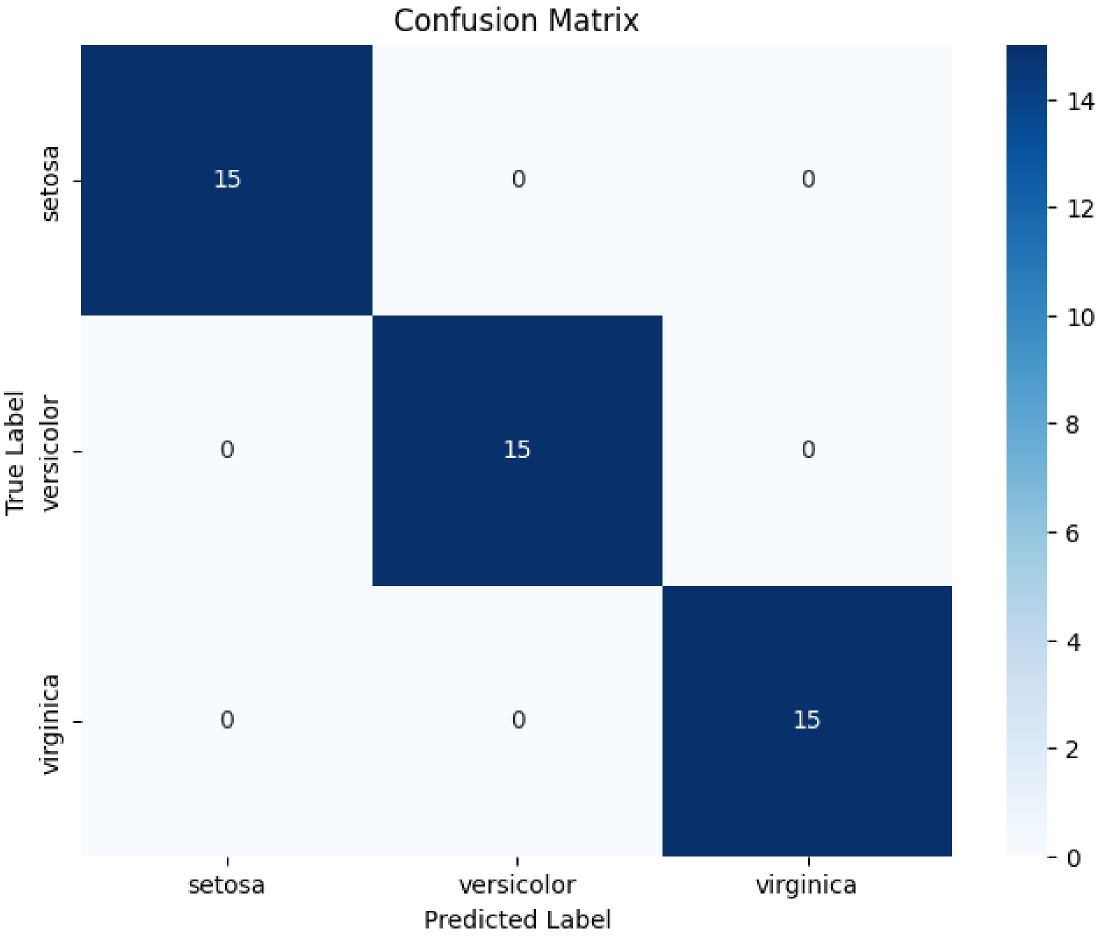
1.0.6 Model Evaluation

We evaluate the model's performance using accuracy, confusion matrix, and a classification report.

[10]:

```
# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")

# Confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)

# Display the confusion matrix using a heatmap
plt.figure(figsize=(8,6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=iris.
 ↪target_names, yticklabels=iris.target_names)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()

# Classification report
print("Classification Report:")
print(classification_report(y_test, y_pred, target_names=iris.target_names))
```

```
Accuracy: 100.00%
```



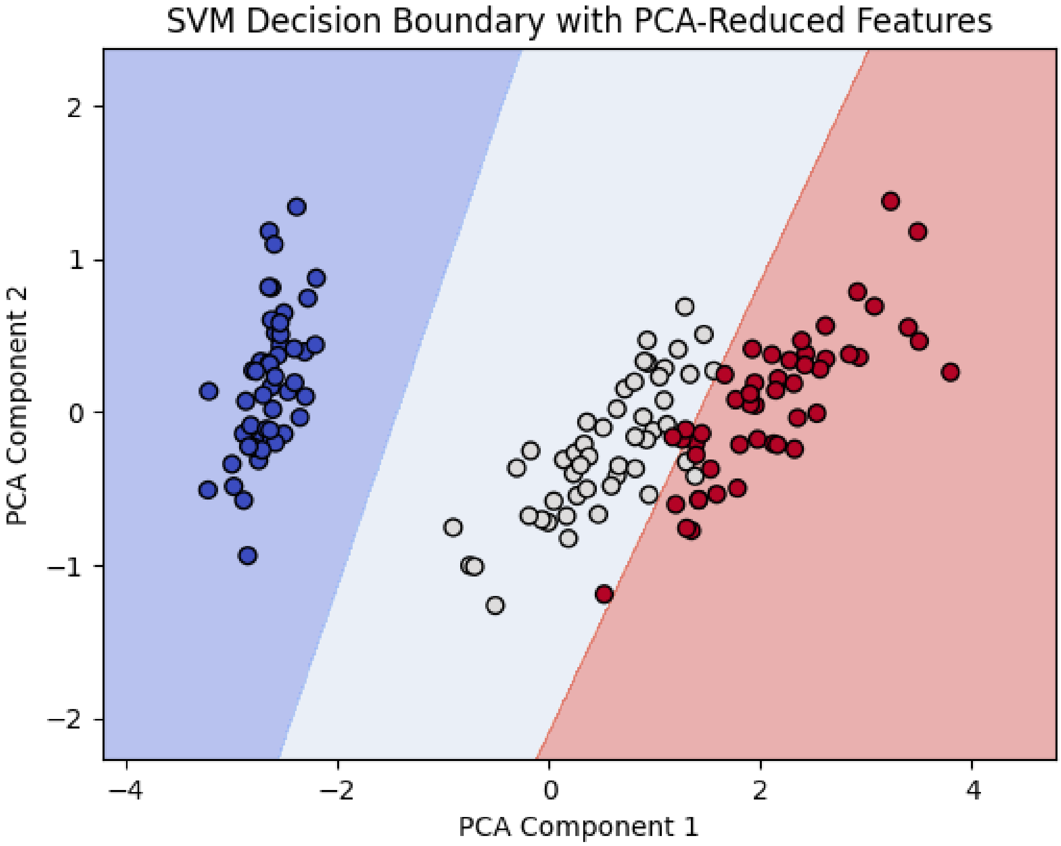
```
Classification Report:
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        15
  versicolor       1.00      1.00      1.00        15
   virginica       1.00      1.00      1.00        15

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```

1.0.7 Visualizing the Decision Boundaries

Since the Iris dataset has 4 features, it's difficult to visualize in high dimensions. However, we can reduce the dimensionality to 2 using **Principal Component Analysis (PCA)** to plot the decision boundaries of the SVM model.

[11]:

```
from sklearn.decomposition import PCA

# Reduce the features to 2 dimensions using PCA for visualization
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

# Split the transformed data into training and testing sets
X_train_pca, X_test_pca, y_train_pca, y_test_pca = train_test_split(X_pca, y,␣
 ↪test_size=0.3, random_state=42, stratify=y)

# Retrain the SVM model on the PCA-reduced data
svm_model_pca = SVC(kernel='linear', random_state=42)
svm_model_pca.fit(X_train_pca, y_train_pca)

# Plot the decision boundaries
def plot_decision_boundary(X, y, model):
    # Create a mesh grid
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
                         np.arange(y_min, y_max, 0.01))

    # Predict class using trained model
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Plot contour and training examples
    plt.contourf(xx, yy, Z, alpha=0.4, cmap='coolwarm')
    plt.scatter(X[:, 0], X[:, 1], c=y, s=40, edgecolor='k', cmap='coolwarm')
    plt.xlabel('PCA Component 1')
    plt.ylabel('PCA Component 2')
    plt.title('SVM Decision Boundary with PCA-Reduced Features')
    plt.show()

# Plot the decision boundary for the SVM model trained on PCA-reduced data
plot_decision_boundary(X_pca, y, svm_model_pca)
```


SVM Decision Boundary with PCA-Reduced Features
2
1
0
−1
−2
−4
−2
0
2
4
PCA Component 1
PCA Component 2

navie-bayes-email-classification

November 19, 2024

1 Naive Bayes Email Classification

This notebook demonstrates email classification using the Naive Bayes algorithm. We will preprocess email text, train a Naive Bayes model to classify emails as "spam" or "not spam," and evaluate the model's performance.

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, confusion_matrix,␣
 ↪classification_report
```

1.0.1 Load the dataset

We'll use a sample dataset of emails. The dataset consists of emails labeled as "spam" or "not spam" (ham).

[2]:
```
file = '/content/spam.csv'

df = pd.read_csv(file)
df.head()
```

[2]:
```
  Category                                            Message
0      ham  Go until jurong point, crazy.. Available only …
1      ham                      Ok lar… Joking wif u oni…
2     spam  Free entry in 2 a wkly comp to win FA Cup fina…
3      ham  U dun say so early hor… U c already then say…
4      ham  Nah I don't think he goes to usf, he lives aro…
```

1.0.2 Data Preprocessing

We will preprocess the email text by converting it into numerical features using **CountVectorizer**, which counts the frequency of words in each email.

[3]:

```
# Initialize CountVectorizer to convert text to numerical data (bag of words␣
 ↪model)
vectorizer = CountVectorizer()

# Transform the text data into feature vectors
X = vectorizer.fit_transform(df['Message']).toarray()

# Target variable (0 = ham, 1 = spam)
y = df['Category'].map({'ham': 0, 'spam': 1}).values

print("X-Axis")
print(X)
print("Y-Axis")
print(y)
```

```
X-Axis
[[0 0 0 … 0 0 0]
 [0 0 0 … 0 0 0]
 [0 0 0 … 0 0 0]
 …
 [0 0 0 … 0 0 0]
 [0 0 0 … 0 0 0]
 [0 0 0 … 0 0 0]]
Y-Axis
[0 0 1 … 0 0 0]
```

1.0.3 Splitting the dataset

We split the data into training and testing sets.

[4]:

```
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=42)
```

1.0.4 Training the Naive Bayes Model

We will train a **Multinomial Naive Bayes** classifier, which is well-suited for text classification.

[5]:

```
# Initialize the Naive Bayes classifier
nb_model = MultinomialNB()

# Train the model
nb_model.fit(X_train, y_train)
```

[5]:

```
MultinomialNB()
```

1.0.5 Model Predictions

We will predict whether the emails in the test set are spam or not using the trained Naive Bayes model.

[6]:

```
# Make predictions on the test set
y_pred = nb_model.predict(X_test)
```

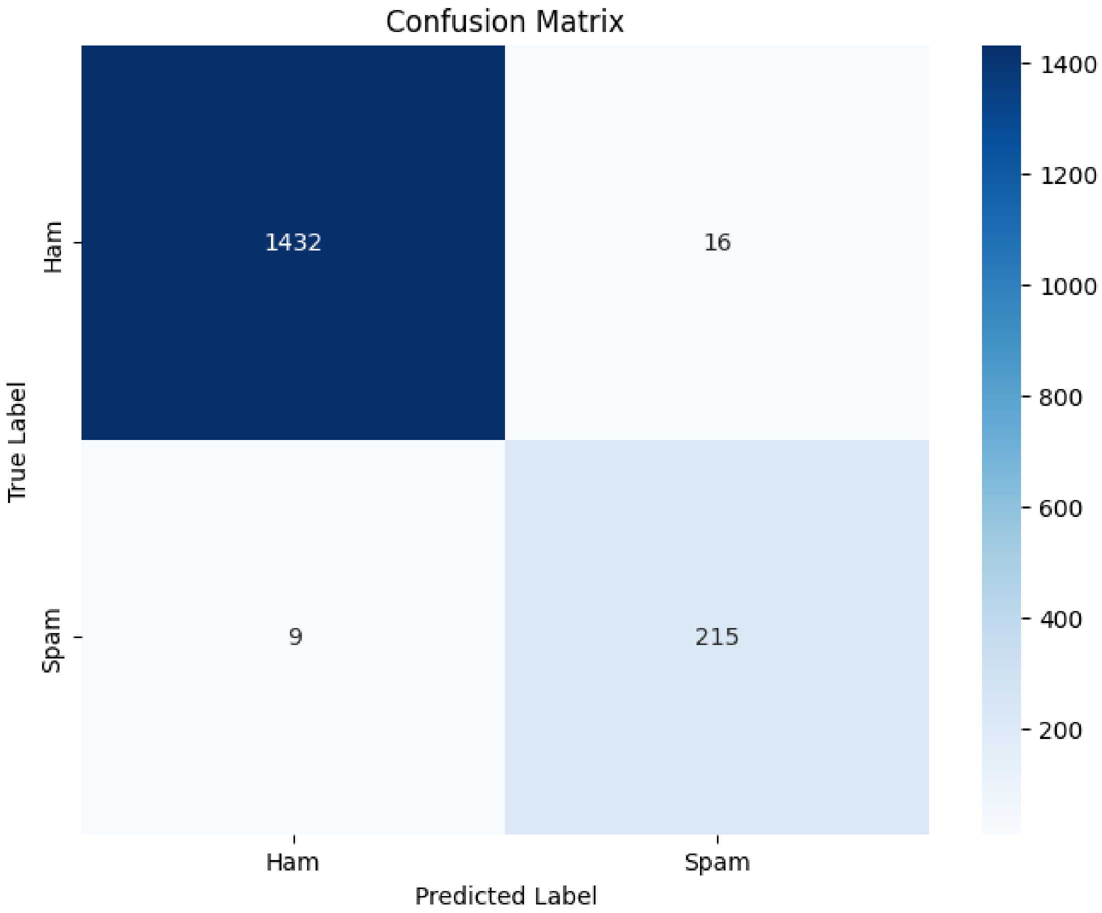
1.0.6 Model Evaluation

We evaluate the performance of the Naive Bayes model using accuracy, precision, recall, F1-score, and confusion matrix.

[7]:

```
# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")

# Confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)

# Display the confusion matrix using a heatmap
plt.figure(figsize=(8,6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Ham',␣
  ↪'Spam'], yticklabels=['Ham', 'Spam'])
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()

# Classification report
print("Classification Report:")
print(classification_report(y_test, y_pred, target_names=['Ham', 'Spam']))
```

```
Accuracy: 98.50%
```



```
Classification Report:
              precision    recall  f1-score   support

         Ham       0.99      0.99      0.99      1448
        Spam       0.93      0.96      0.95       224

    accuracy                           0.99      1672
   macro avg       0.96      0.97      0.97      1672
weighted avg       0.99      0.99      0.99      1672
```

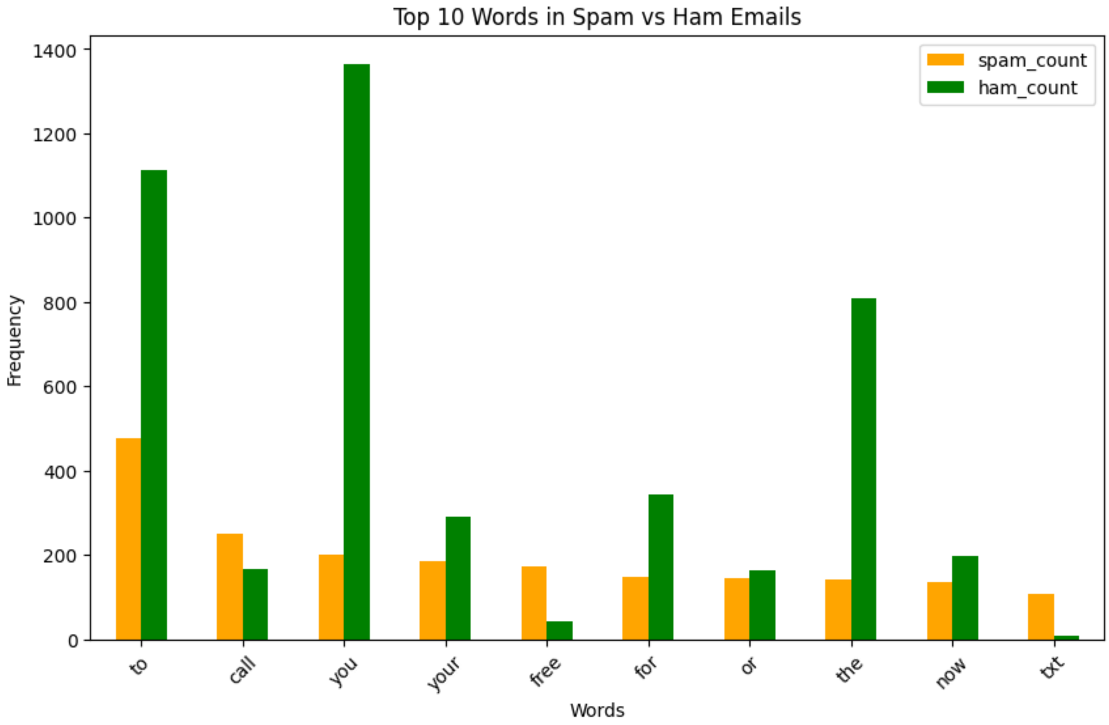
1.0.7 Visualizing Word Frequencies

We can visualize the most frequent words used in the spam and ham emails to gain insights.

[8]:
```
# Get the feature names (words) from the vectorizer
feature_names = vectorizer.get_feature_names_out()

# Visualizing word frequencies for spam and ham emails
```

```
spam_word_counts = X_train[y_train == 1].sum(axis=0)
ham_word_counts = X_train[y_train == 0].sum(axis=0)

# Create a DataFrame for word counts
word_freq_df = pd.DataFrame({'word': feature_names, 'spam_count':␣
 ↪spam_word_counts, 'ham_count': ham_word_counts})

# Sort the DataFrame by spam count
word_freq_df = word_freq_df.sort_values(by='spam_count', ascending=False).
 ↪head(10)

# Plot word frequencies for spam and ham emails
plt.figure(figsize=(10,6))
word_freq_df.plot(x='word', y=['spam_count', 'ham_count'], kind='bar',␣
 ↪figsize=(10,6), color=['orange', 'green'])
plt.title('Top 10 Words in Spam vs Ham Emails')
plt.xlabel('Words')
plt.ylabel('Frequency')
plt.xticks(rotation=45)
plt.show()
```

```
<Figure size 1000x600 with 0 Axes>
```



naive-bayes-health-dataset

November 19, 2024

1 Naive Bayes Classifier on Diabetes Dataset

This notebook uses the **Naive Bayes** classification algorithm to predict the presence of diabetes based on various health metrics. The dataset includes factors such as pregnancies, glucose levels, blood pressure, skin thickness, insulin, BMI, age, and others.

The target variable, **Outcome**, indicates whether the individual is diabetic (`1`) or not (`0`).

1.1 Importing Libraries

```
[10]: from sklearn.naive_bayes import MultinomialNB
      from sklearn.metrics import accuracy_score, confusion_matrix,␣
       ↪classification_report
      import pandas as pd
      import numpy as np
      from sklearn.model_selection import train_test_split
      import matplotlib.pyplot as plt
      import seaborn as sns
```

1.2 Dataset Overview

The dataset looks like this:

Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
6	148	72	35	0	33.6	0.627	50	1
1	85	66	29	0	26.6	0.351	31	0
8	183	64	0	0	23.3	0.672	32	1
1	89	66	23	94	28.1	0.167	21	0

1.3 Loading the Dataset

```
[11]: file = '/content/diabetes.csv'  # Replace with your dataset path if needed
      df = pd.read_csv(file)
      df.head()  # Display the first few rows of the dataset
```

```
[11]:    Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI  \
      0            6      148             72             35        0  33.6
```

```
1              1       85              66              29         0   26.6
2              8      183              64               0         0   23.3
3              1       89              66              23        94   28.1
4              0      137              40              35       168   43.1

   DiabetesPedigreeFunction  Age  Outcome
0                     0.627   50        1
1                     0.351   31        0
2                     0.672   32        1
3                     0.167   21        0
4                     2.288   33        1
```

1.4 Data Preprocessing

We will separate the features and the target variable, and then split the data into training and test sets.

[12]:

```
# Separate features and target
X = df.drop('Outcome', axis=1)
y = df['Outcome']

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
```

1.5 Training the Naive Bayes Model

We will initialize the Multinomial Naive Bayes classifier and train it on the training data.

[6]:

```
# Initialize the Naive Bayes classifier
nb_model = MultinomialNB()

# Train the model
nb_model.fit(X_train, y_train)

# Make predictions
y_pred = nb_model.predict(X_test)
```

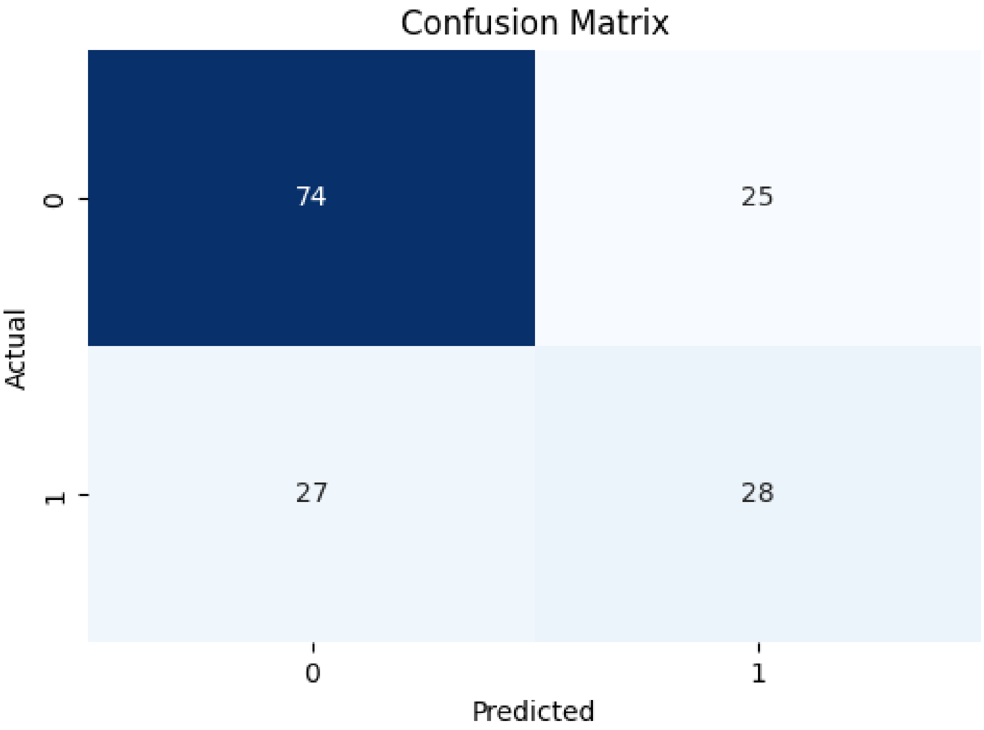
1.6 Evaluating the Model

We'll evaluate the performance of the model using accuracy, a confusion matrix, and a classification report.

[16]:

```python
# Model accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")

# Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", conf_matrix)

# Classification Report
print("Classification Report:\n", classification_report(y_test, y_pred))

# Plotting confusion matrix
plt.figure(figsize=(6, 4))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', cbar=False)
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
```

```
Accuracy: 0.66
Confusion Matrix:
 [[74 25]
 [27 28]]
Classification Report:
               precision    recall  f1-score   support

           0       0.73      0.75      0.74        99
           1       0.53      0.51      0.52        55

    accuracy                           0.66       154
   macro avg       0.63      0.63      0.63       154
weighted avg       0.66      0.66      0.66       154
```



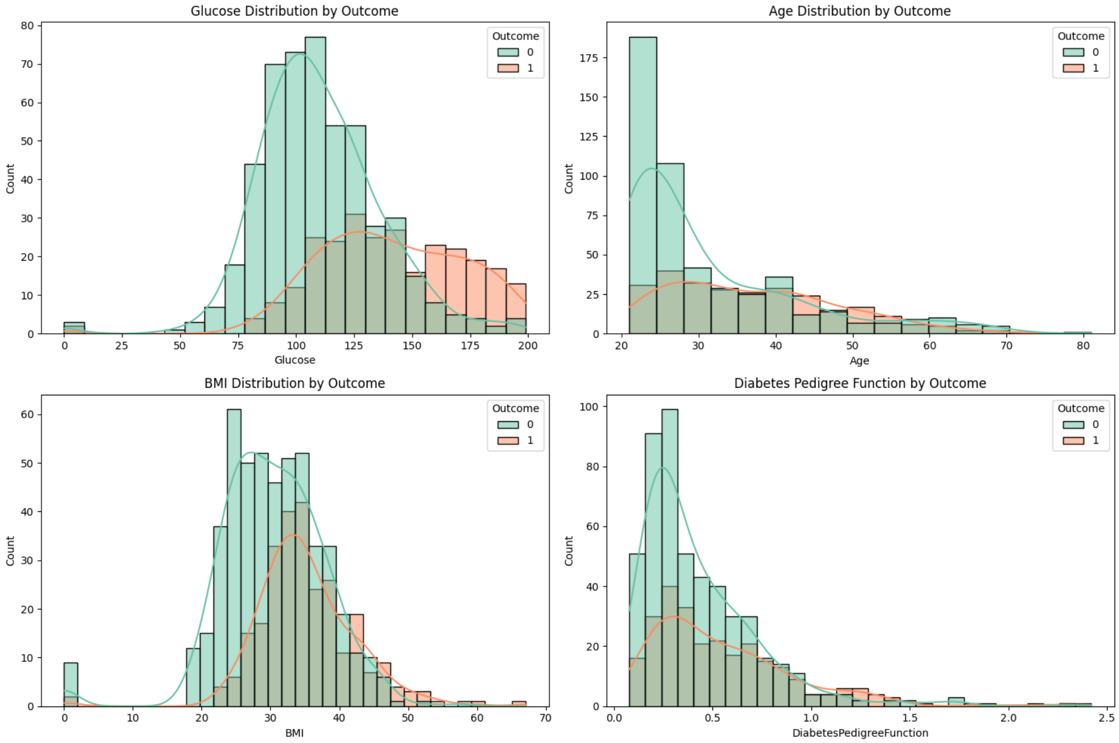
1.7 Distribution of Features by Outcome

We will visualize the distributions of key features for diabetic vs non-diabetic individuals.

[15]:

```
# Plotting distributions of important features
plt.figure(figsize=(15, 10))

# Glucose levels
plt.subplot(2, 2, 1)
sns.histplot(data=df, x='Glucose', hue='Outcome', kde=True, palette='Set2')
plt.title('Glucose Distribution by Outcome')

# Age distribution
plt.subplot(2, 2, 2)
sns.histplot(data=df, x='Age', hue='Outcome', kde=True, palette='Set2')
plt.title('Age Distribution by Outcome')

# BMI distribution
plt.subplot(2, 2, 3)
sns.histplot(data=df, x='BMI', hue='Outcome', kde=True, palette='Set2')
plt.title('BMI Distribution by Outcome')

# DiabetesPedigreeFunction
```

```
plt.subplot(2, 2, 4)
sns.histplot(data=df, x='DiabetesPedigreeFunction', hue='Outcome', kde=True,␣
 ↪palette='Set2')
plt.title('Diabetes Pedigree Function by Outcome')

plt.tight_layout()
plt.show()
```



standard-scale-with-knn

November 19, 2024

1 Standard Scale with KNN

This notebook demonstrates the application of StandardScaler with KNN on the Iris dataset. It includes data visualization, detailed steps, and evaluation metrics for better understanding.

[ ]:

```
# Importing necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score, classification_report,␣
 ↪confusion_matrix
```

1.1 Load the Dataset

The Iris dataset is a classic dataset used in machine learning. It contains three classes of iris plants, characterized by four features: sepal length, sepal width, petal length, and petal width.

[ ]:

```
# Load the Iris dataset
data = load_iris()
X = data.data  # Features
y = data.target  # Labels
feature_names = data.feature_names
class_names = data.target_names

# Convert to pandas DataFrame for better visualization
iris_df = pd.DataFrame(X, columns=feature_names)
iris_df['species'] = [class_names[label] for label in y]

# Display the first few rows of the dataset
iris_df.head()
```

[ ]:

```
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                5.1               3.5                1.4               0.2
```

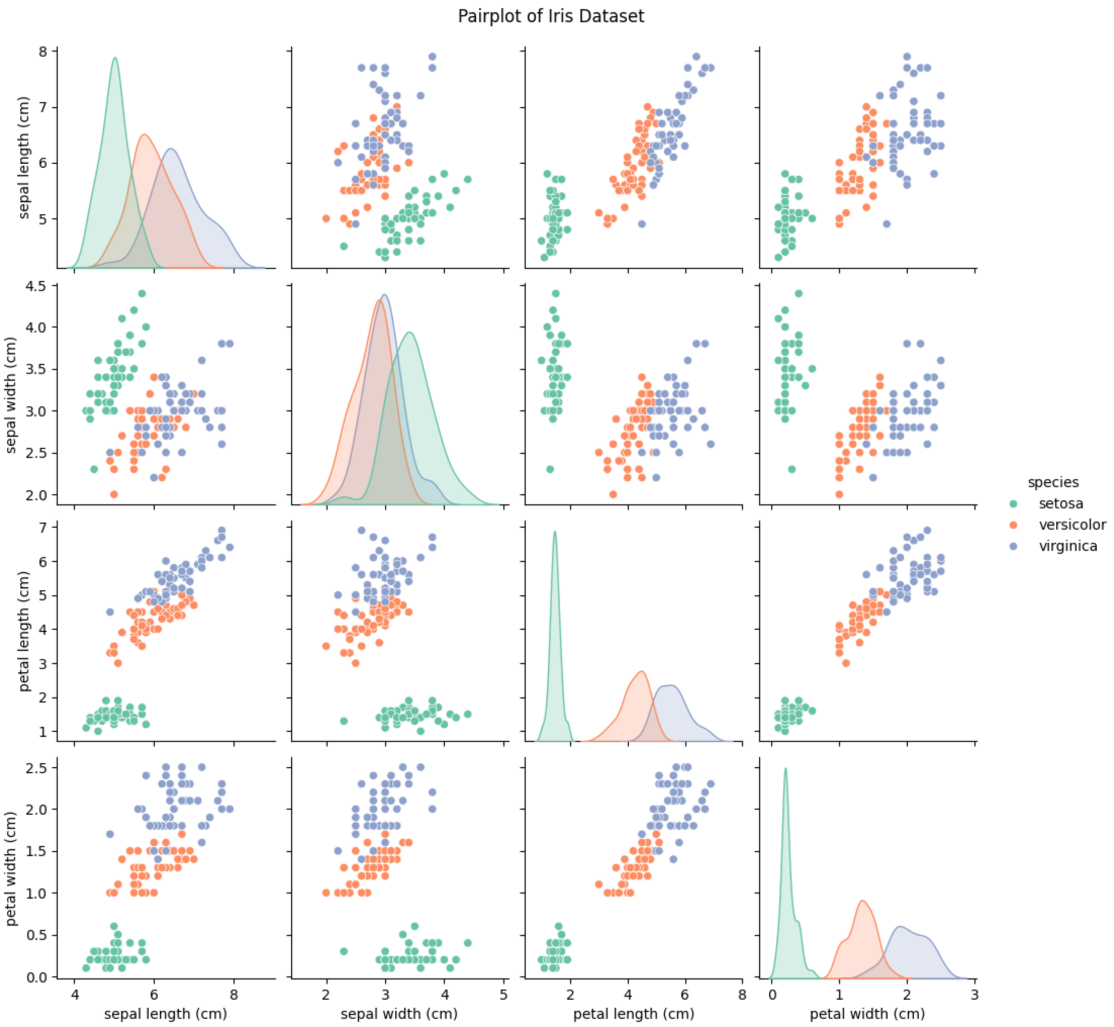
```
1                 4.9               3.0               1.4               0.2
2                 4.7               3.2               1.3               0.2
3                 4.6               3.1               1.5               0.2
4                 5.0               3.6               1.4               0.2

  species
0  setosa
1  setosa
2  setosa
3  setosa
4  setosa
```

1.2 Visualize the Dataset

Use pair plots to visualize relationships between features, colored by species.

[ ]:

```
sns.pairplot(iris_df, hue='species', diag_kind='kde', palette='Set2')
plt.suptitle("Pairplot of Iris Dataset", y=1.02)
plt.show()
```



1.3 Split the Data into Training and Testing Sets

We'll split the data into training (80%) and testing (20%) sets.

```
[ ]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

     print(f"Training set size: {X_train.shape[0]} samples")
     print(f"Testing set size: {X_test.shape[0]} samples")
```

```
Training set size: 120 samples
Testing set size: 30 samples
```

1.4 Standardize the Features

Standardizing features scales the data to have a mean of 0 and a standard deviation of 1. This is essential for algorithms like KNN to perform well.

```
[ ]: scaler = StandardScaler()
     X_train_scaled = scaler.fit_transform(X_train)
     X_test_scaled = scaler.transform(X_test)
```

1.5 Initialize and Train the KNN Model

KNN is a simple algorithm that classifies samples based on the majority vote of their neighbors.

```
[ ]: knn = KNeighborsClassifier(n_neighbors=3)
     knn.fit(X_train_scaled, y_train)
```

```
[ ]: KNeighborsClassifier(n_neighbors=3)
```

1.6 Make Predictions and Evaluate the Model

Predict on the test set and evaluate the model's performance using accuracy and classification report.

```
[ ]: y_pred = knn.predict(X_test_scaled)

     # Calculate accuracy
     accuracy = accuracy_score(y_test, y_pred)
     print(f"Accuracy: {accuracy:.2f}")

     # Display classification report
     print("\nClassification Report:\n")
     print(classification_report(y_test, y_pred, target_names=class_names))
```

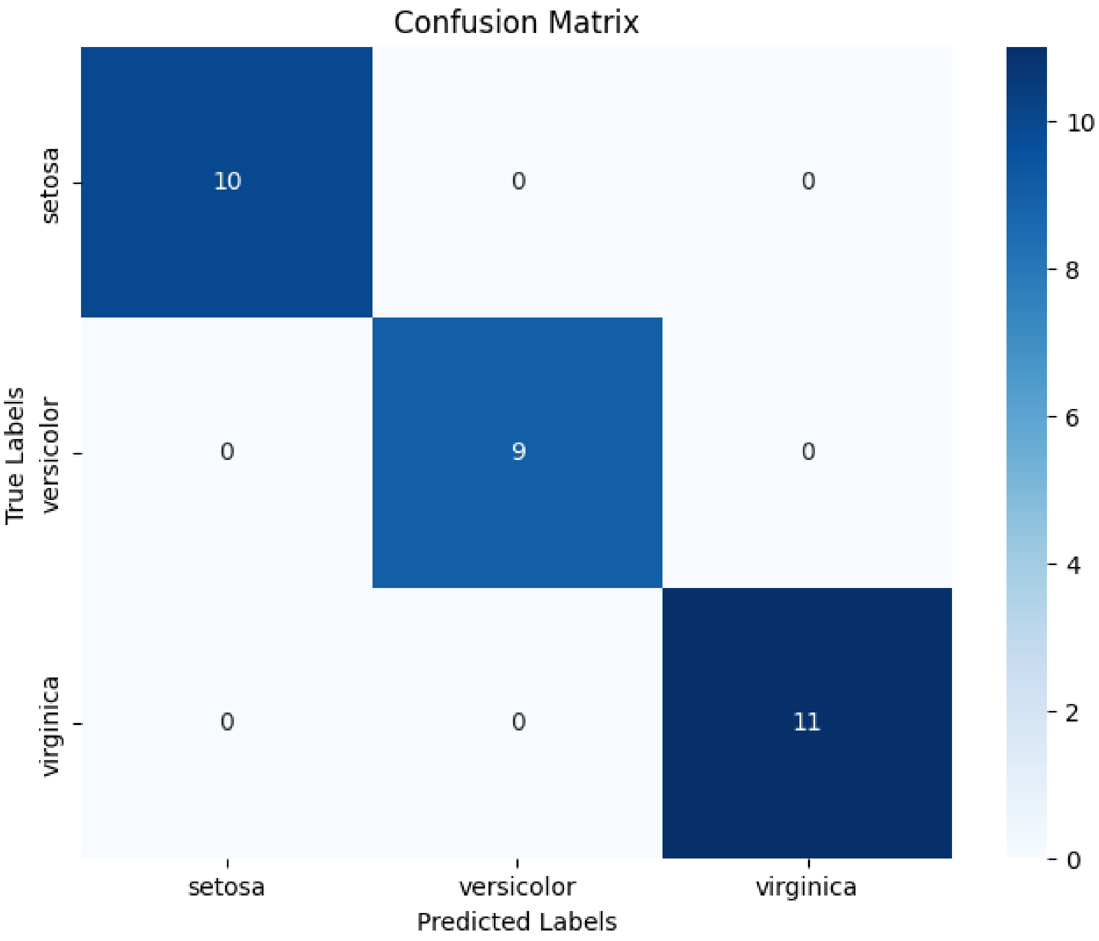
```
Accuracy: 1.00

Classification Report:

              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        10
  versicolor       1.00      1.00      1.00         9
   virginica       1.00      1.00      1.00        11

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

1.7 Visualize Confusion Matrix

A confusion matrix helps us see the breakdown of correct and incorrect predictions for each class.

[ ]:

```
conf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, cmap='Blues', fmt='d',␣
 ↪xticklabels=class_names, yticklabels=class_names)
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.title("Confusion Matrix")
plt.show()
```



knn-on-iris-dataset

November 19, 2024

1 K-Nearest Neighbors on Iris Dataset

KNN is a supervised classification algorithm that classifies a data point based on the class of its nearest neighbors.

1.0.1 Importing necessary libraries

We use `pandas` and `numpy` for data manipulation, `matplotlib` for visualizations, and `sklearn` for model building.

```
[ ]: import numpy as np
     import pandas as pd
     import matplotlib.pyplot as plt
     from sklearn import datasets
     from sklearn.neighbors import KNeighborsClassifier
     from sklearn.model_selection import train_test_split
     from sklearn.metrics import accuracy_score, confusion_matrix,␣
       ↪classification_report
```

```
[ ]: # Load the full iris dataset from sklearn
     iris = datasets.load_iris()

     # Create a DataFrame for easier handling
     df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
     df['target'] = iris.target

     # Mapping species to human-readable labels
     df['target'] = df['target'].map({0: 'setosa', 1: 'versicolor', 2: 'virginica'})

     df.head()
```

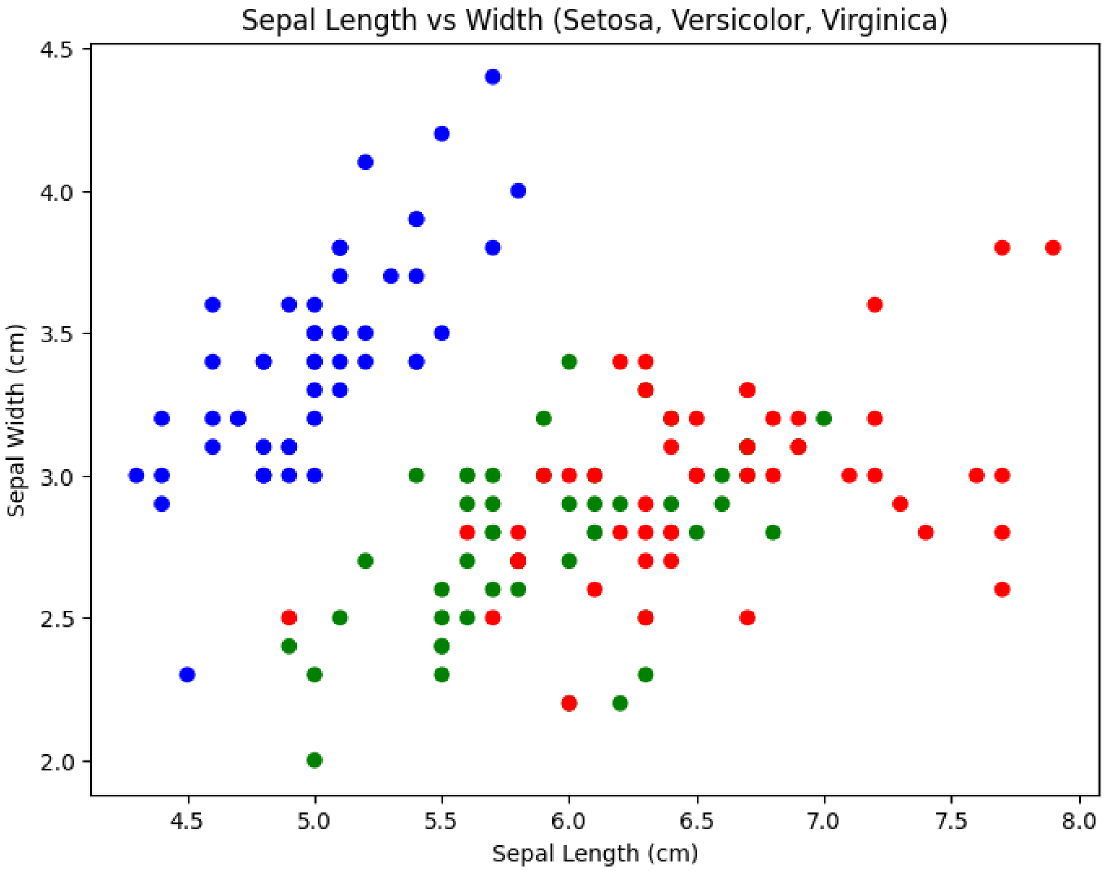
```
[ ]:    sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
     0                5.1               3.5                1.4               0.2
     1                4.9               3.0                1.4               0.2
     2                4.7               3.2                1.3               0.2
     3                4.6               3.1                1.5               0.2
     4                5.0               3.6                1.4               0.2
```

```
  target
0  setosa
1  setosa
2  setosa
3  setosa
4  setosa
```

1.0.2 Visualizing the data

We will create a scater plot of two features: sepal length tand sepal width, color-coded by species, to understand the relationship between these features and the flower species.

[ ]:
```
# Plotting sepal length vs sepal width, color-coded by species
plt.figure(figsize=(8,6))
species_color = {'setosa': 'blue', 'versicolor': 'green', 'virginica': 'red'}
plt.scatter(df['sepal length (cm)'], df['sepal width (cm)'],
            c=df['target'].map(species_color), label='Data Points')
plt.xlabel('Sepal Length (cm)')
plt.ylabel('Sepal Width (cm)')
plt.title('Sepal Length vs Width (Setosa, Versicolor, Virginica)')
plt.show()
```



1.0.3 Preparing the data

We will now split the data into features (X) and labels (y), and then into training and testing sets.

[ ]:

```
# Features (sepal length and width) and target (species)
X = df[['sepal length (cm)', 'sepal width (cm)']].values  # Independent␣
 ↪variables
y = df['target'].map({'setosa': 0, 'versicolor': 1, 'virginica': 2}).values  #␣
 ↪Dependent variable (multi-class classification)

# Split the dataset into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
```

1.0.4 Building and Training the KNN Model

We will now create the K-Nearest Neighbors (KNN) model and train it on the training data.

[ ]:

```
# Initialize the KNN model with k=3 neighbors
knn_model = KNeighborsClassifier(n_neighbors=3)

# Train the model using the training data
knn_model.fit(X_train, y_train)
```

[ ]:

```
KNeighborsClassifier(n_neighbors=3)
```

1.0.5 Model Evaluation

We will evaluate the performance of the KNN model by predicting on the test data and calculating accuracy.

[ ]:

```
# Make predictions on the test set
y_pred = knn_model.predict(X_test)

# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

# Generate a confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", conf_matrix)

# Detailed classification report
class_report = classification_report(y_test, y_pred)
print("Classification Report:\n", class_report)
```

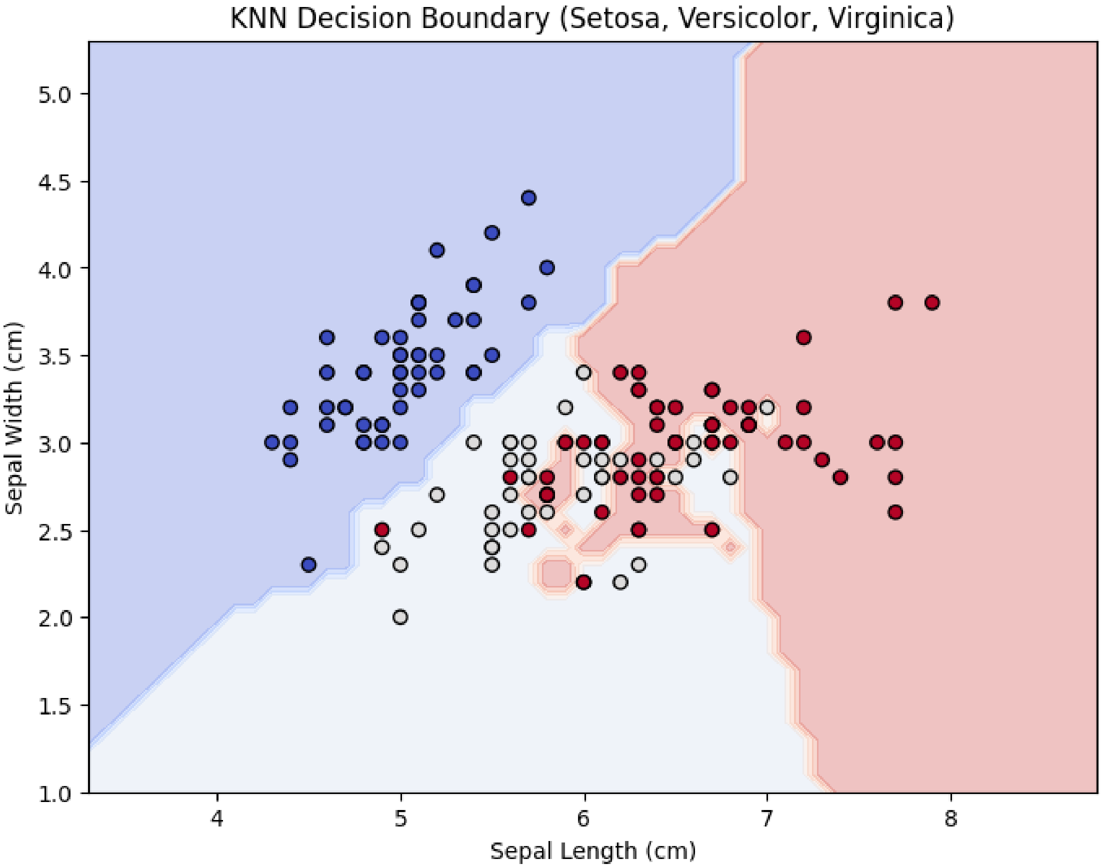
```
Accuracy: 0.8333333333333334
Confusion Matrix:
 [[10  0  0]
 [ 0  7  2]
 [ 0  3  8]]
Classification Report:
               precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       0.70      0.78      0.74         9
           2       0.80      0.73      0.76        11

    accuracy                           0.83        30
   macro avg       0.83      0.84      0.83        30
weighted avg       0.84      0.83      0.83        30
```

1.0.6 Decision Boundary Plot

We will now visualize the decision boundary created by the KNN model.

[ ]:

```
# Create a mesh grid to plot the decision boundary
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))

# Use the model to predict the class for each point in the mesh grid
Z = knn_model.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# Plot the decision boundary
plt.figure(figsize=(8,6))
plt.contourf(xx, yy, Z, alpha=0.3, cmap='coolwarm')
plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', cmap='coolwarm')
plt.xlabel('Sepal Length (cm)')
plt.ylabel('Sepal Width (cm)')
plt.title('KNN Decision Boundary (Setosa, Versicolor, Virginica)')
plt.show()
```


KNN Decision Boundary (Setosa, Versicolor, Virginica)
Sepal Width (cm)
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
4
5
6
7
8
Sepal Length (cm)

sion-tree-on-breast-cancer-dataset

November 19, 2024

1 Decision Tree Classifier on Breast Cancer Dataset

This notebook implements a Decision Tree Classifier on the Breast Cancer Dataset using K-Fold Cross-Validation for evaluation and visualizes the trained model and feature importance.

1.1 Import Libraries

[1]:

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_breast_cancer
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.model_selection import cross_val_score, KFold, train_test_split
import matplotlib.pyplot as plt
import seaborn as sns
```

1.2 Load the Breast Cancer Dataset

The breast cancer dataset is available from the `sklearn.datasets` module. It contains data on features that describe a tumor, such as texture, smoothness, area, etc., and a target variable representing whether the tumor is malignant (0) or benign (1).

[2]:

```
# Load the dataset from sklearn
data = load_breast_cancer()
X = pd.DataFrame(data.data, columns=data.feature_names)  # Features
y = pd.Series(data.target)  # Target (0 = malignant, 1 = benign)

# Display basic information about the dataset
print(f"Dataset Shape: {X.shape}")
X.head()
```

```
Dataset Shape: (569, 30)
```

[2]:

```
   mean radius  mean texture  mean perimeter  mean area  mean smoothness  \
0        17.99         10.38          122.80     1001.0          0.11840
1        20.57         17.77          132.90     1326.0          0.08474
2        19.69         21.25          130.00     1203.0          0.10960
3        11.42         20.38           77.58      386.1          0.14250
4        20.29         14.34          135.10     1297.0          0.10030
```

```
   mean compactness  mean concavity  mean concave points  mean symmetry  \
0           0.27760          0.3001              0.14710         0.2419
1           0.07864          0.0869              0.07017         0.1812
2           0.15990          0.1974              0.12790         0.2069
3           0.28390          0.2414              0.10520         0.2597
4           0.13280          0.1980              0.10430         0.1809

   mean fractal dimension  …  worst radius  worst texture  worst perimeter  \
0                 0.07871  …         25.38          17.33           184.60
1                 0.05667  …         24.99          23.41           158.80
2                 0.05999  …         23.57          25.53           152.50
3                 0.09744  …         14.91          26.50            98.87
4                 0.05883  …         22.54          16.67           152.20

   worst area  worst smoothness  worst compactness  worst concavity  \
0      2019.0            0.1622             0.6656           0.7119
1      1956.0            0.1238             0.1866           0.2416
2      1709.0            0.1444             0.4245           0.4504
3       567.7            0.2098             0.8663           0.6869
4      1575.0            0.1374             0.2050           0.4000

   worst concave points  worst symmetry  worst fractal dimension
0                0.2654          0.4601                  0.11890
1                0.1860          0.2750                  0.08902
2                0.2430          0.3613                  0.08758
3                0.2575          0.6638                  0.17300
4                0.1625          0.2364                  0.07678

[5 rows x 30 columns]
```

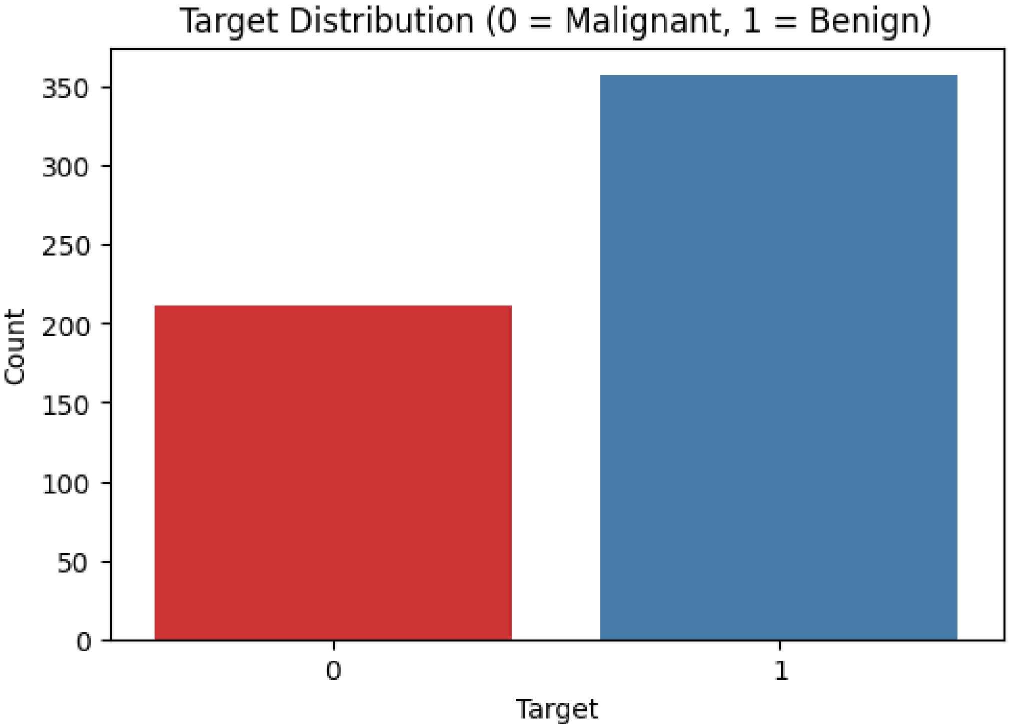
1.3 Visualize the Dataset

Before training the model, let's visualize the distribution of the target variable (malignant vs benign). This helps us understand if there is a class imbalance or if both classes are fairly represented.

[3]:
```
# Check the distribution of the target variable
plt.figure(figsize=(6, 4))
sns.countplot(x=y, palette="Set1")
plt.title("Target Distribution (0 = Malignant, 1 = Benign)")
plt.xlabel("Target")
plt.ylabel("Count")
plt.show()
```

```
<ipython-input-3-4f0049775732>:3: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in
v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same
```

```
effect.

  sns.countplot(x=y, palette="Set1")
```



1.4 Train-Test Split

To evaluate the model's performance, we first split the data into training and testing sets. This allows us to train the model on one portion of the data and test it on a separate portion to check for overfitting and generalization.

[4]:
```
# Split the data into training and testing sets for initial testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
  ↪random_state=42)
```

1.5 Build the Decision Tree Model

We initialize and fit the Decision Tree Classifier on the training data. The model uses the Gini index as the criterion to split the data, and we limit the tree depth to 4 to prevent overfitting.

[5]:
```
# Initialize the Decision Tree Classifier
model = DecisionTreeClassifier(random_state=42, criterion="gini", max_depth=4)

# Fit the model on the training data
model.fit(X_train, y_train)
```

```python

# Evaluate the model's performance on the test set
test_accuracy = model.score(X_test, y_test)
print(f"Decision Tree Test Accuracy: {test_accuracy:.2f}")
```

```
Decision Tree Test Accuracy: 0.95
```

1.6 Perform K-Fold Cross-Validation

To obtain a more reliable evaluation of the model's performance, we use K-Fold Cross-Validation. This technique splits the dataset into K subsets (or folds), training and testing the model K times. We report the average accuracy across all folds.

[6]:

```python
# Use K-Fold Cross-Validation to evaluate the model
kfold = KFold(n_splits=5, shuffle=True, random_state=42)
cv_scores = cross_val_score(model, X, y, cv=kfold)

# Display the cross-validation results
print(f"Cross-Validation Scores: {cv_scores}")
print(f"Mean Cross-Validation Accuracy: {cv_scores.mean():.2f}")
```

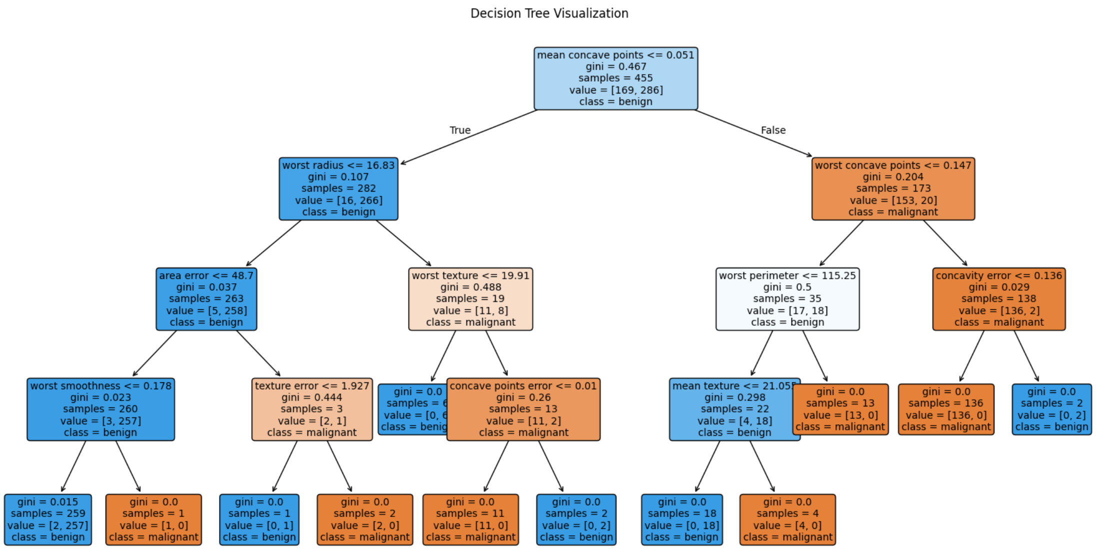
```
Cross-Validation Scores: [0.94736842 0.96491228 0.92982456 0.94736842
0.94690265]
Mean Cross-Validation Accuracy: 0.95
```

1.7 Visualize the Decision Tree

Visualizing the trained decision tree helps us understand how the model is making decisions based on the feature values. Each node represents a decision point, and the branches show the splits based on feature values.

[7]:

```python
# Plot the trained decision tree
plt.figure(figsize=(20, 10))
plot_tree(
    model,
    feature_names=data.feature_names,
    class_names=data.target_names,
    filled=True,
    rounded=True,
    fontsize=10,
)
plt.title("Decision Tree Visualization")
plt.show()
```



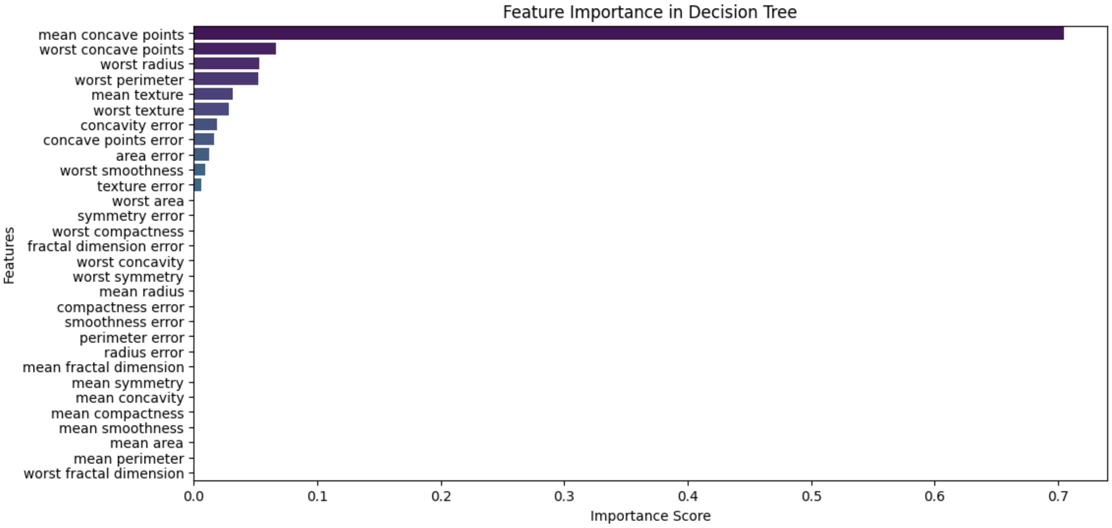
1.8 Feature Importance Visualization

A decision tree classifier assigns different importance to each feature based on how helpful they are in making decisions. This visualization shows the relative importance of each feature in the decision-making process of the tree.

[8]:

```
# Extract and visualize the importance of each feature
feature_importance = pd.Series(model.feature_importances_, index=data.
  ↪feature_names)
feature_importance = feature_importance.sort_values(ascending=False)

plt.figure(figsize=(12, 6))
sns.barplot(x=feature_importance, y=feature_importance.index, palette="viridis")
plt.title("Feature Importance in Decision Tree")
plt.xlabel("Importance Score")
plt.ylabel("Features")
plt.show()
```

```
<ipython-input-8-e30ceabadcca>:6: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in
v0.14.0. Assign the `y` variable to `hue` and set `legend=False` for the same
effect.

  sns.barplot(x=feature_importance, y=feature_importance.index,
palette="viridis")
```



1.9 Predict on New Data

Finally, we can use the trained decision tree model to make predictions on new, unseen samples. This demonstrates how the model can classify tumors as malignant or benign based on their feature values.

[21]:

```
import warnings

with warnings.catch_warnings():
    warnings.filterwarnings('ignore')
    # Code that generates warnings
    # Demonstrate prediction on new samples (manually created)

    # Example 1: Feature values of a new breast cancer sample
    sample_1 = X.iloc[0].values  # First sample from the dataset
    sample_2 = X.iloc[10].values  # Another sample from the dataset

    # Ensure samples are in the correct 2D format
    sample_1 = sample_1.reshape(1, -1)  # Reshape to (1, number_of_features)
    sample_2 = sample_2.reshape(1, -1)

    # Predict whether these samples are malignant or benign
    predicted_1 = model.predict(sample_1)
    predicted_2 = model.predict(sample_2)

    # Map predictions to their respective classes
    result_1 = "Benign" if predicted_1[0] == 1 else "Malignant"
    result_2 = "Benign" if predicted_2[0] == 1 else "Malignant"
    # Print the results
    print(f"Prediction for Sample 1: {result_1}")
    print(f"Prediction for Sample 2: {result_2}")
```

```
Prediction for Sample 1: Malignant
Prediction for Sample 2: Malignant
```

decision-tree-with-salary-dataset

November 19, 2024

1 Salary Prediction with Decision Tree

This notebook demonstrates how to use a Decision Tree Classifier to predict whether an employee earns more than 100k based on their company, job title, and degree.

1.1 Import Necessary Libraries

[3]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder
from sklearn import tree
from sklearn.metrics import accuracy_score, classification_report,␣
 ↪confusion_matrix
import seaborn as sns
```

1.2 Load and Explore the Dataset

We start by loading the dataset, which contains information about employees, including their company, job title, degree, and salary (whether it's more than 100k or not). We will explore the dataset to understand its structure and gain some insights.

[4]:

```
# Load the dataset containing information about salaries
data = pd.read_csv('/content/salaries.csv')
df = pd.DataFrame(data)
df.head()

# Display dataset information
print("Dataset Overview:")
print(df.info())
print("\nSummary Statistics:")
print(df.describe(include="all"))
```

```
Dataset Overview:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 16 entries, 0 to 15
Data columns (total 4 columns):
 #   Column                 Non-Null Count  Dtype
```

```
---  ------                 --------------  -----
 0   company                16 non-null     object
 1   job                    16 non-null     object
 2   degree                 16 non-null     object
 3   salary_more_then_100k  16 non-null     int64
dtypes: int64(1), object(3)
memory usage: 640.0+ bytes
None

Summary Statistics:
       company               job     degree  salary_more_then_100k
count       16                16         16                 16.000
unique       3                 3          2                    NaN
top     google  business manager  bachelors                    NaN
freq         6                 6          8                    NaN
mean       NaN               NaN        NaN                  0.625
std        NaN               NaN        NaN                  0.500
min        NaN               NaN        NaN                  0.000
25%        NaN               NaN        NaN                  0.000
50%        NaN               NaN        NaN                  1.000
75%        NaN               NaN        NaN                  1.000
max        NaN               NaN        NaN                  1.000
```

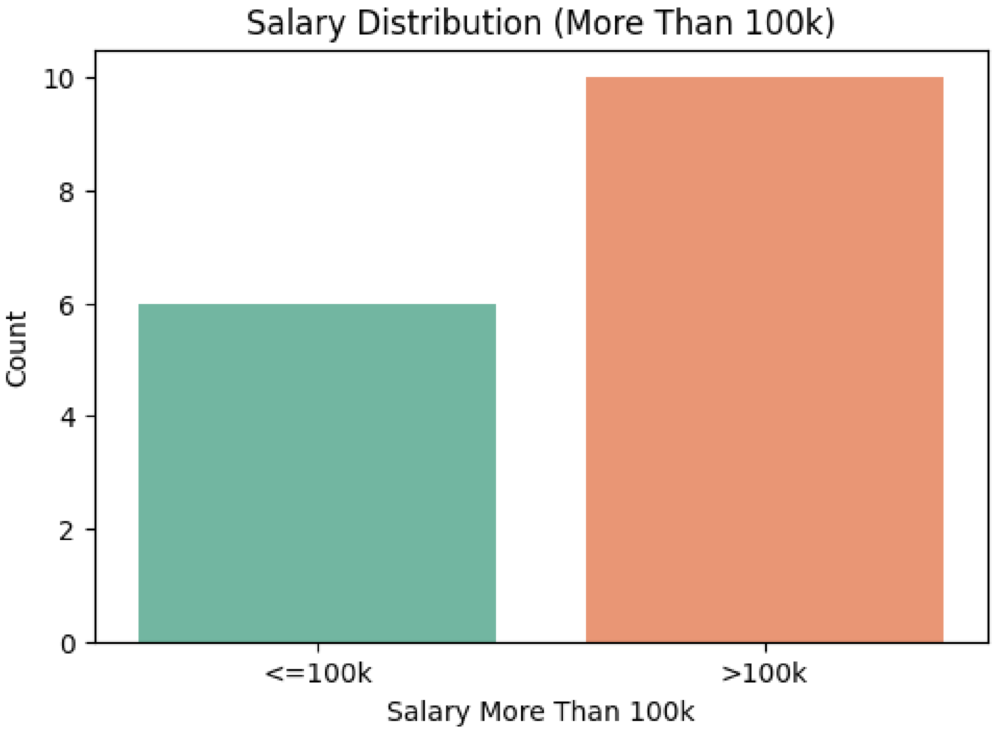
1.3 Visualize the Dataset

Before training the model, we visualize the distribution of the target variable (`salary_more_then_100k`) to understand how many employees earn more than 100k and how many earn less or equal to 100k.

[5]:

```python
# Show the distribution of the target variable
plt.figure(figsize=(6, 4))
sns.countplot(x='salary_more_then_100k', data=df, palette='Set2')
plt.title("Salary Distribution (More Than 100k)")
plt.xlabel("Salary More Than 100k")
plt.ylabel("Count")
plt.xticks(ticks=[0, 1], labels=["<=100k", ">100k"])
plt.show()
```

```
<ipython-input-5-c5ce02dba7cc>:3: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in
v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same
effect.

  sns.countplot(x='salary_more_then_100k', data=df, palette='Set2')
```



1.4 Encode Categorical Variables

The dataset contains categorical variables (`company`, `job`, and `degree`). Decision Trees require numerical input, so we will encode these categorical variables using `LabelEncoder`, which converts each category into a numerical value.

[6]:
```
inputs = df.drop('salary_more_then_100k', axis='columns')
target = df['salary_more_then_100k']

# Initialize LabelEncoders for each column
le_company = LabelEncoder()
le_job = LabelEncoder()
le_degree = LabelEncoder()

# Apply encoding
inputs['company_n'] = le_company.fit_transform(inputs['company'])
inputs['job_n'] = le_job.fit_transform(inputs['job'])
inputs['degree_n'] = le_degree.fit_transform(inputs['degree'])

# Display the transformed inputs
print("\nTransformed Inputs:")
print(inputs.head())
```

```
# Drop the original columns as they are now encoded
inputs_n = inputs.drop(['company', 'job', 'degree'], axis='columns')
```

```
Transformed Inputs:
  company                   job     degree  company_n  job_n  degree_n
0  google       sales executive  bachelors          2      2         0
1  google       sales executive    masters          2      2         1
2  google      business manager  bachelors          2      0         0
3  google      business manager    masters          2      0         1
4  google   computer programmer  bachelors          2      1         0
```

1.5 Train the Decision Tree Model

Now that the data is preprocessed, we can train a Decision Tree Classifier on the dataset. The model will learn to predict whether an employee earns more than 100k based on the encoded features (`company_n`, `job_n`, and `degree_n`).

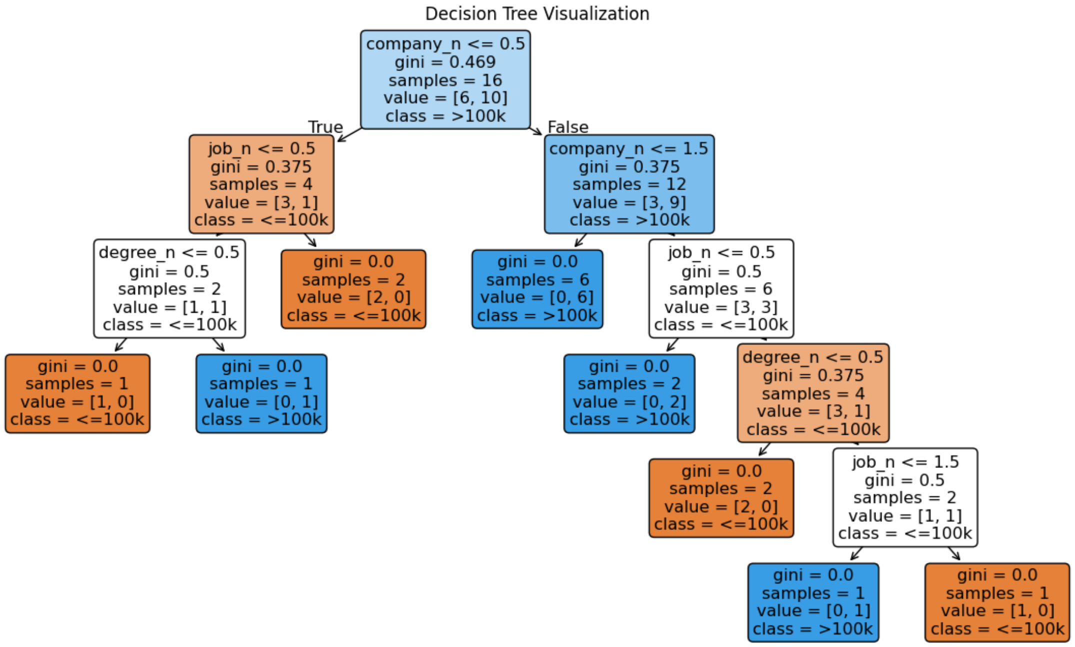
[7]:
```
# Train a Decision Tree Classifier on the dataset
model = tree.DecisionTreeClassifier()
model.fit(inputs_n, target)

# Evaluate the model on the training data
accuracy = model.score(inputs_n, target)
print(f"\nModel Accuracy on Training Data: {accuracy:.2f}")
```

```

Model Accuracy on Training Data: 1.00
```

1.6 Visualize the Decision Tree

To understand how the model makes decisions, we visualize the trained Decision Tree. This allows us to see the rules learned by the model based on the input features.

[8]:
```
# Visualize the decision tree to understand the learned rules
plt.figure(figsize=(14, 8))
tree.plot_tree(
    model,
    feature_names=['company_n', 'job_n', 'degree_n'],
    class_names=['<=100k', '>100k'],
    filled=True,
    rounded=True
)
plt.title("Decision Tree Visualization")
plt.show()
```



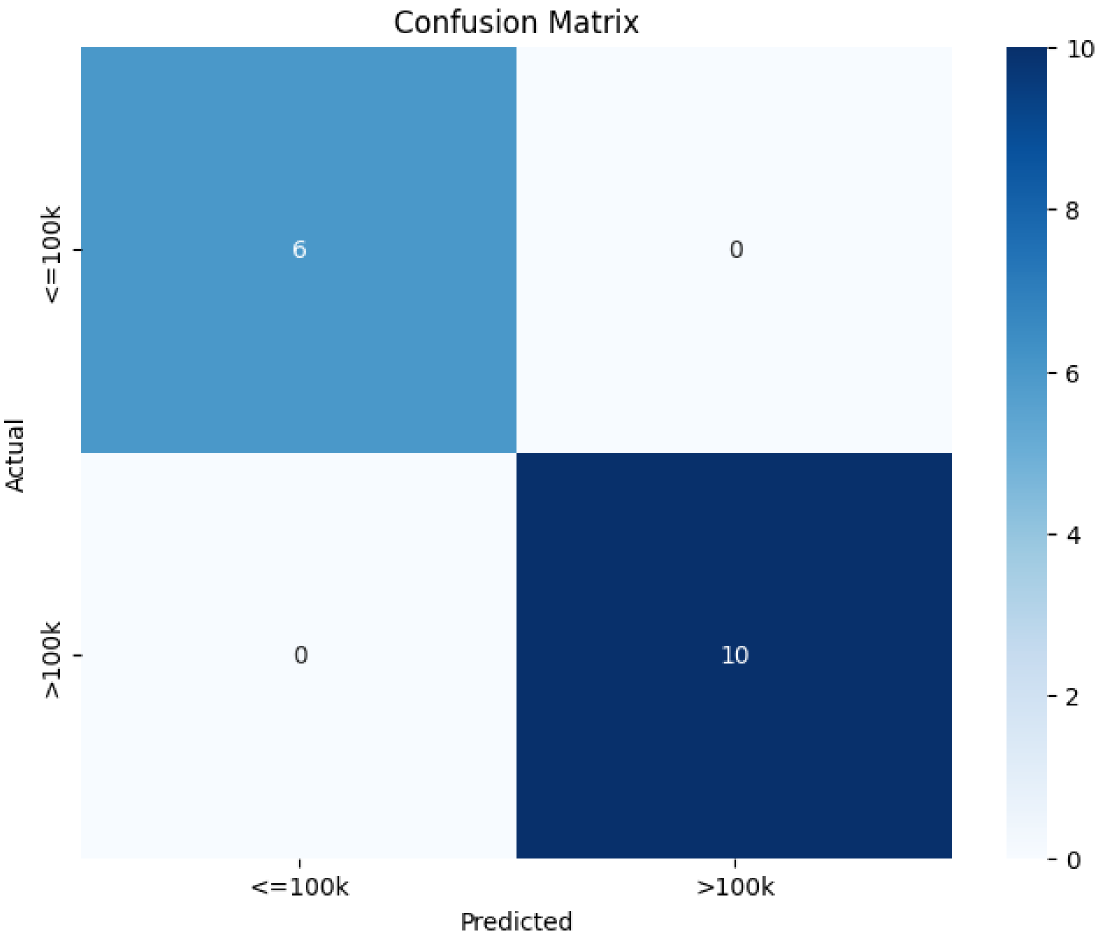
1.7 Evaluate the Model

We evaluate the model's performance by generating a classification report, which provides precision, recall, and F1-score metrics for each class. Additionally, we visualize the confusion matrix to understand how well the model is performing.

[9]:
```
# Predict and evaluate the model's performance
y_pred = model.predict(inputs_n)

print("\nClassification Report:")
print(classification_report(target, y_pred, target_names=["<=100k", ">100k"]))

# Confusion matrix visualization
conf_matrix = confusion_matrix(target, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',␣
  ↪xticklabels=["<=100k", ">100k"], yticklabels=["<=100k", ">100k"])
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix")
plt.show()
```

```
Classification Report:
              precision    recall  f1-score   support
```

```
       <=100k       1.00      1.00      1.00         6
        >100k       1.00      1.00      1.00        10

     accuracy                           1.00        16
    macro avg       1.00      1.00      1.00        16
 weighted avg       1.00      1.00      1.00        16
```



1.8 Make Predictions

Finally, we use the trained model to make predictions on new data. In this case, we provide new data points in the form of encoded values for `company`, `job`, and `degree` and predict whether the employee would earn more than 100k.

```
[10]: import warnings

      with warnings.catch_warnings():
          warnings.filterwarnings('ignore')
```

```python
# Use the trained model to make predictions on new data
new_data = [[2, 2, 0], [1, 2, 0]]  # [company_n, job_n, degree_n]
predictions = model.predict(new_data)

# Map predictions back to labels for clarity
predicted_labels = ["<=100k" if pred == 0 else ">100k" for pred in predictions]

print("\nPredictions for New Data:")
for data_point, label in zip(new_data, predicted_labels):
    print(f"Input: {data_point}, Predicted Salary: {label}")
```

```
Predictions for New Data:
Input: [2, 2, 0], Predicted Salary: <=100k
Input: [1, 2, 0], Predicted Salary: >100k
```

ical-data-heart-disease-dataset-1

November 19, 2024

1 Bayesian Network on Medical Data (Heart Disease Dataset)

This notebook demonstrates the use of Bayesian Networks on medical data. It includes structure learning, inference, and visualization of the network.

1.1 Import Libraries

[6]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from pgmpy.estimators import HillClimbSearch, BicScore
from pgmpy.models import BayesianNetwork
from pgmpy.inference import VariableElimination
from pgmpy.factors.discrete import DiscreteFactor
from pgmpy.estimators import MaximumLikelihoodEstimator
import networkx as nx # Use NetworkX for visualization
```

1.2 Load the Diabetes Dataset

This is the Pima Indians Diabetes dataset. We will use it to construct a Bayesian Network and perform inference.

You can download the dataset from OpenML or other sources.

[2]:

```
# Load the diabetes dataset (You can download the dataset from OpenML or other␣
 ↪sources)
url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/
 ↪pima-indians-diabetes.data.csv"
column_names = ["Pregnancies", "Glucose", "BloodPressure", "SkinThickness",␣
 ↪"Insulin", "BMI", "DiabetesPedigreeFunction", "Age", "Outcome"]
df = pd.read_csv(url, names=column_names)
# Drop the last 10 rows for demonstration purposes
df = df.iloc[-10:]
# Check for missing values
```

```
print("Missing values:\n", df.isnull().sum())
# Drop missing values (if any)
df = df.dropna()
# Discretize continuous features for Bayesian Network
for col in ['Glucose', 'BMI', 'Age']:
    df[col] = pd.cut(df[col], bins=4, labels=False)  # Convert continuous␣
 ↪values to discrete categories
# Check the updated dataframe
df.head()
```

```
Missing values:
 Pregnancies                 0
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

```
[2]:      Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin  BMI  \
     758            1        0             76              0        0    2
     759            6        3             92              0        0    2
     760            2        0             58             26       16    1
     761            9        3             74             31        0    3
     762            9        0             62              0        0    0

          DiabetesPedigreeFunction  Age  Outcome
     758                     0.197    0        0
     759                     0.278    3        1
     760                     0.766    0        0
     761                     0.403    1        1
     762                     0.142    0        0
```

2 Building the Bayesian Network

We will use the pgmpy library to construct the Bayesian Network. A Bayesian Network is a graphical model that represents the probabilistic relationships among variables. We will define the structure of the network based on domain knowledge and then learn the parameters from the data.

2.0.1 Create and Train the Bayesian Network

Now, let's define the structure of the network and use the pgmpy library to build the Bayesian Network.

[3]:

```python
# Define the structure of the Bayesian Network (this is based on domain␣
 ↪knowledge)
model = BayesianNetwork([
    ('Pregnancies', 'Glucose'),
    ('Glucose', 'Outcome'),
    ('BloodPressure', 'BMI'),
    ('BMI', 'Outcome'),
    ('Insulin', 'Outcome'),
    ('DiabetesPedigreeFunction', 'Outcome'),
    ('Age', 'Outcome')
])
# Learn the parameters from the data using MaximumLikelihoodEstimator
model.fit(df, estimator=MaximumLikelihoodEstimator)
# Print the learned parameters (CPDs) for each node
for cpd in model.get_cpds():
    print(cpd)
```

```
+-----------------+-----+
| Pregnancies(1)  | 0.3 |
+-----------------+-----+
| Pregnancies(2)  | 0.2 |
+-----------------+-----+
| Pregnancies(5)  | 0.1 |
+-----------------+-----+
| Pregnancies(6)  | 0.1 |
+-----------------+-----+
| Pregnancies(9)  | 0.2 |
+-----------------+-----+
| Pregnancies(10) | 0.1 |
+-----------------+-----+
+-------------+--------------------+-----+----------------+-----------------+
| Pregnancies | Pregnancies(1)     | … | Pregnancies(9) | Pregnancies(10) |
+-------------+--------------------+-----+----------------+-----------------+
| Glucose(0)  | 0.6666666666666666 | … | 0.5            | 1.0             |
+-------------+--------------------+-----+----------------+-----------------+
| Glucose(1)  | 0.3333333333333333 | … | 0.0            | 0.0             |
+-------------+--------------------+-----+----------------+-----------------+
| Glucose(3)  | 0.0                | … | 0.5            | 0.0             |
+-------------+--------------------+-----+----------------+-----------------+
+--------------------------+-----+---------------------------------+
| Age                      | … | Age(3)                          |
+--------------------------+-----+---------------------------------+
| BMI                      | … | BMI(3)                          |
+--------------------------+-----+---------------------------------+
| DiabetesPedigreeFunction | … | DiabetesPedigreeFunction(0.766) |
+--------------------------+-----+---------------------------------+
| Glucose                  | … | Glucose(3)                      |
+---------------------------+-----+----------------------------------+
| Insulin                   | … | Insulin(180)                     |
+---------------------------+-----+----------------------------------+
| Outcome(0)                | … | 0.5                              |
+---------------------------+-----+----------------------------------+
| Outcome(1)                | … | 0.5                              |
+---------------------------+-----+----------------------------------+
+-------------------+-----+
| BloodPressure(58) | 0.1 |
+-------------------+-----+
| BloodPressure(60) | 0.1 |
+-------------------+-----+
| BloodPressure(62) | 0.1 |
+-------------------+-----+
| BloodPressure(70) | 0.2 |
+-------------------+-----+
| BloodPressure(72) | 0.1 |
+-------------------+-----+
| BloodPressure(74) | 0.1 |
+-------------------+-----+
| BloodPressure(76) | 0.2 |
+-------------------+-----+
| BloodPressure(92) | 0.1 |
+-------------------+-----+
+---------------+-------------------+-----+-------------------+-----------------
--+
| BloodPressure | BloodPressure(58) | … | BloodPressure(76) |
BloodPressure(92) |
+---------------+-------------------+-----+-------------------+-----------------
--+
| BMI(0)        | 0.0               | … | 0.0               | 0.0
|
+---------------+-------------------+-----+-------------------+-----------------
--+
| BMI(1)        | 1.0               | … | 0.5               | 0.0
|
+---------------+-------------------+-----+-------------------+-----------------
--+
| BMI(2)        | 0.0               | … | 0.5               | 1.0
|
+---------------+-------------------+-----+-------------------+-----------------
--+
| BMI(3)        | 0.0               | … | 0.0               | 0.0
|
+---------------+-------------------+-----+-------------------+-----------------
--+
+--------------+-----+
| Insulin(0)   | 0.7 |
+--------------+-----+
| Insulin(16)  | 0.1 |
+--------------+-----+
| Insulin(112) | 0.1 |
+--------------+-----+
| Insulin(180) | 0.1 |
+--------------+-----+
+---------------------------------+-----+
| DiabetesPedigreeFunction(0.142) | 0.1 |
+---------------------------------+-----+
| DiabetesPedigreeFunction(0.171) | 0.1 |
+---------------------------------+-----+
| DiabetesPedigreeFunction(0.197) | 0.1 |
+---------------------------------+-----+
| DiabetesPedigreeFunction(0.245) | 0.1 |
+---------------------------------+-----+
| DiabetesPedigreeFunction(0.278) | 0.1 |
+---------------------------------+-----+
| DiabetesPedigreeFunction(0.315) | 0.1 |
+---------------------------------+-----+
| DiabetesPedigreeFunction(0.34)  | 0.1 |
+---------------------------------+-----+
| DiabetesPedigreeFunction(0.349) | 0.1 |
+---------------------------------+-----+
| DiabetesPedigreeFunction(0.403) | 0.1 |
+---------------------------------+-----+
| DiabetesPedigreeFunction(0.766) | 0.1 |
+---------------------------------+-----+
+--------+-----+
| Age(0) | 0.6 |
+--------+-----+
| Age(1) | 0.1 |
+--------+-----+
| Age(2) | 0.1 |
+--------+-----+
| Age(3) | 0.2 |
+--------+-----+
```

2.1 Visualizing the Bayesian Network

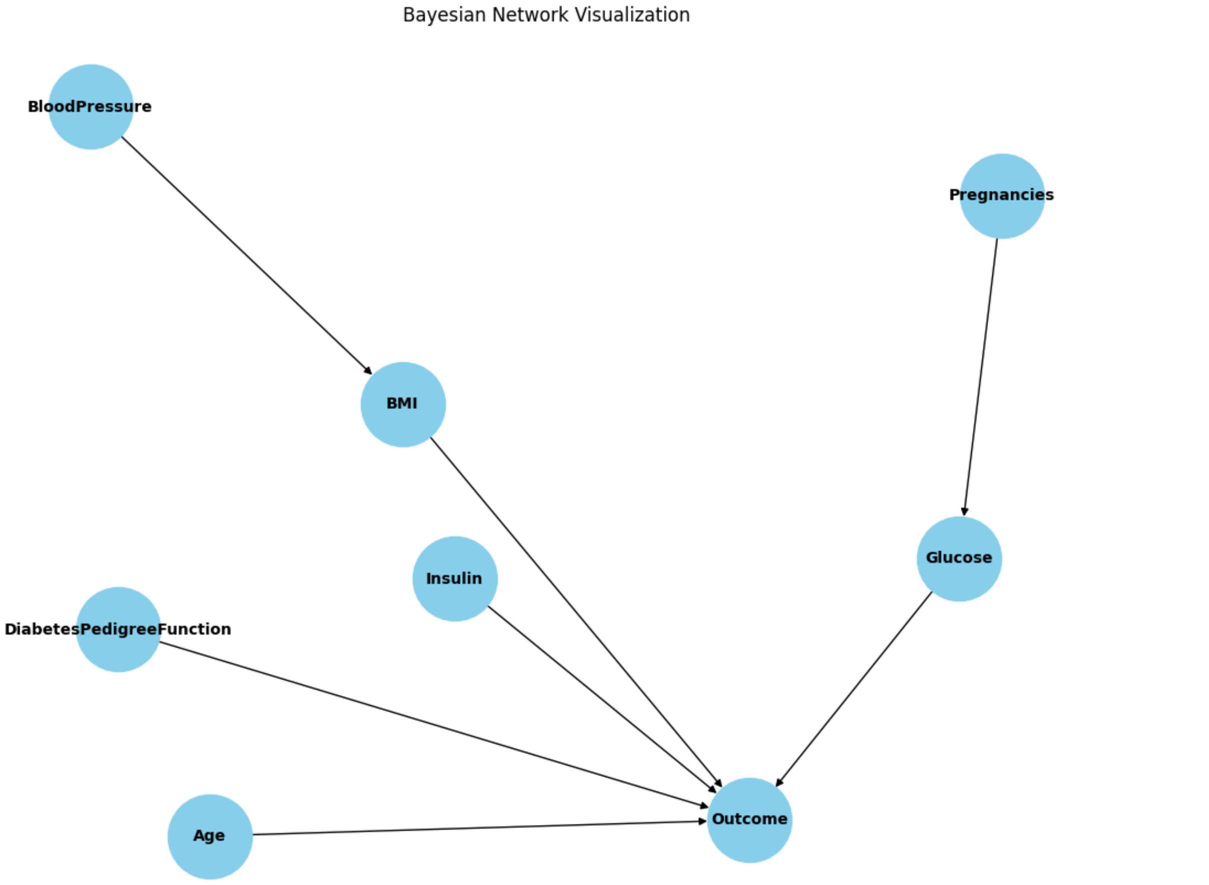
Visualization is key to understanding the structure and relationships between variables in a Bayesian Network. The pgmpy library provides a function to visualize the network.

We will use NetworkX to create and visualize the graph.

[4]:

```python
# Initialize a NetworkX graph
G = nx.DiGraph()  # Directed graph (for Bayesian Network)
# Add nodes and edges from the pgmpy model
```

```
for node in model.nodes():
    G.add_node(node)
for edge in model.edges():
    G.add_edge(edge[0], edge[1])
# Visualize the graph using Matplotlib
plt.figure(figsize=(10, 8))
nx.draw(G, with_labels=True, node_size=3000, node_color='skyblue',␣
  ↪font_size=10, font_weight='bold', arrows=True)
plt.title('Bayesian Network Visualization')
plt.show()
```



2.2 Inference in the Bayesian Network

Once the network is built and visualized, we can use it to perform inference. For example, we can calculate the probability of a patient being diabetic given certain conditions (e.g., Glucose level, Age, etc.).

2.2.1 Query the Probability of Outcome Given Certain Conditions

We will now perform inference on the trained Bayesian Network by querying the probability of `Outcome` given `Age` and `Glucose`.

Note: Ensure the evidence provided in the query corresponds to the discretized values of `Age` and `Glucose`.

[5]:

```
# Check unique values after discretization
print(f"Discretized 'Age' values: {df['Age'].unique()}")
print(f"Discretized 'Glucose' values: {df['Glucose'].unique()}")
# Perform inference using Variable Elimination
inference = VariableElimination(model)
# Query the probability of Outcome given Age and Glucose (using discretized␣
 ↪values)
query = inference.query(variables=['Outcome'], evidence={'Age': 2, 'Glucose':␣
 ↪3})  # Here, Age=2 and Glucose=3 are discretized categories
print(query)
```

```
Discretized 'Age' values: [0 3 1 2]
Discretized 'Glucose' values: [0 3 1]
+------------+----------------+
| Outcome    |   phi(Outcome) |
+============+================+
| Outcome(0) |         0.5000 |
+------------+----------------+
| Outcome(1) |         0.5000 |
+------------+----------------+
```

2.3 Results and Interpretation

2.3.1 Bayesian Network Structure

The structure of the network shows how variables are dependent on each other. For example: - `Glucose` is influenced by `Pregnancies`. - `Outcome` is influenced by `Glucose`, `BMI`, `Age`, and other factors.

2.3.2 Inference

With the trained Bayesian Network, we can query the probability of a certain outcome (diabetes, in this case) given specific conditions. This can be very useful in medical decision-making, where the likelihood of diabetes can be predicted based on a patient's medical data.

k-means-on-breast-cancer-dataset

November 19, 2024

1 K-means Clustering on Breast Cancer Dataset

This program demonstrates the use of K-means clustering on the Breast Cancer dataset. It includes preprocessing, clustering, mapping clusters to actual class labels, and visualizing the clustering performance.

1.1 Import Libraries

[2]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from sklearn.metrics import accuracy_score, confusion_matrix
from collections import Counter
```

1.2 Load and Explore the Dataset

The Breast Cancer dataset contains features derived from digitized images of breast mass biopsies, classified as either malignant (`0`) or benign (`1`).

We will: 1. Load the dataset from `sklearn`. 2. Convert it into a `DataFrame` for easier handling. 3. Inspect its structure.

[3]:

```
# Load the dataset
data = load_breast_cancer()
X = data.data  # Features
y = data.target  # True labels (0 = malignant, 1 = benign)
feature_names = data.feature_names
target_names = data.target_names

# Convert to a DataFrame
df = pd.DataFrame(X, columns=feature_names)
df['target'] = y

# Display basic information about the dataset
```

```python
print("Dataset Overview:")
print(df.info())
print("\nSample Data:")
print(df.head())
```

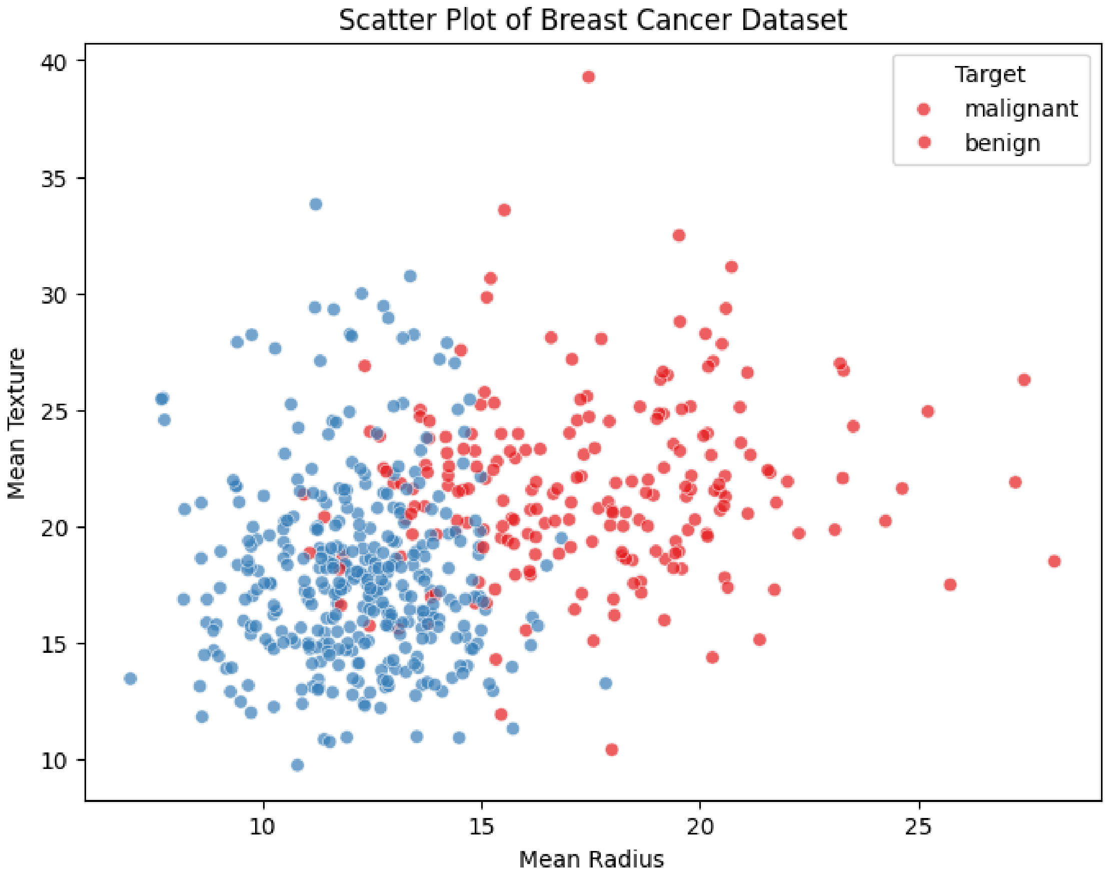
```
Dataset Overview:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 569 entries, 0 to 568
Data columns (total 31 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   mean radius              569 non-null    float64
 1   mean texture             569 non-null    float64
 2   mean perimeter           569 non-null    float64
 3   mean area                569 non-null    float64
 4   mean smoothness          569 non-null    float64
 5   mean compactness         569 non-null    float64
 6   mean concavity           569 non-null    float64
 7   mean concave points      569 non-null    float64
 8   mean symmetry            569 non-null    float64
 9   mean fractal dimension   569 non-null    float64
 10  radius error             569 non-null    float64
 11  texture error            569 non-null    float64
 12  perimeter error          569 non-null    float64
 13  area error               569 non-null    float64
 14  smoothness error         569 non-null    float64
 15  compactness error        569 non-null    float64
 16  concavity error          569 non-null    float64
 17  concave points error     569 non-null    float64
 18  symmetry error           569 non-null    float64
 19  fractal dimension error  569 non-null    float64
 20  worst radius             569 non-null    float64
 21  worst texture            569 non-null    float64
 22  worst perimeter          569 non-null    float64
 23  worst area               569 non-null    float64
 24  worst smoothness         569 non-null    float64
 25  worst compactness        569 non-null    float64
 26  worst concavity          569 non-null    float64
 27  worst concave points     569 non-null    float64
 28  worst symmetry           569 non-null    float64
 29  worst fractal dimension  569 non-null    float64
 30  target                   569 non-null    int64
dtypes: float64(30), int64(1)
memory usage: 137.9 KB
None

Sample Data:
   mean radius  mean texture  mean perimeter  mean area  mean smoothness  \
0        17.99         10.38          122.80     1001.0          0.11840   
1        20.57         17.77          132.90     1326.0          0.08474   
2        19.69         21.25          130.00     1203.0          0.10960   
3        11.42         20.38           77.58      386.1          0.14250   
4        20.29         14.34          135.10     1297.0          0.10030   

   mean compactness  mean concavity  mean concave points  mean symmetry  \
0           0.27760          0.3001              0.14710         0.2419   
1           0.07864          0.0869              0.07017         0.1812   
2           0.15990          0.1974              0.12790         0.2069   
3           0.28390          0.2414              0.10520         0.2597   
4           0.13280          0.1980              0.10430         0.1809   

   mean fractal dimension  ...  worst texture  worst perimeter  worst area  \
0                 0.07871  ...          17.33           184.60      2019.0   
1                 0.05667  ...          23.41           158.80      1956.0   
2                 0.05999  ...          25.53           152.50      1709.0   
3                 0.09744  ...          26.50            98.87       567.7   
4                 0.05883  ...          16.67           152.20      1575.0   

   worst smoothness  worst compactness  worst concavity  worst concave points  \
0            0.1622             0.6656           0.7119                0.2654   
1            0.1238             0.1866           0.2416                0.1860   
2            0.1444             0.4245           0.4504                0.2430   
3            0.2098             0.8663           0.6869                0.2575   
4            0.1374             0.2050           0.4000                0.1625   

   worst symmetry  worst fractal dimension  target  
0          0.4601                  0.11890       0  
1          0.2750                  0.08902       0  
2          0.3613                  0.08758       0  
3          0.6638                  0.17300       0  
4          0.2364                  0.07678       0  

[5 rows x 31 columns]
```

1.3 Visualize the Dataset

To understand the data distribution, let's visualize two selected features (`mean radius` and `mean texture`) using a scatter plot, color-coded by the actual class labels (malignant or benign).

[4]:

```
plt.figure(figsize=(8, 6))
sns.scatterplot(
    x=df['mean radius'],
    y=df['mean texture'],
    hue=df['target'],
    palette="Set1",
    alpha=0.7
)
plt.title("Scatter Plot of Breast Cancer Dataset")
plt.xlabel("Mean Radius")
plt.ylabel("Mean Texture")
plt.legend(title="Target", labels=target_names)
plt.show()
```



1.4 Standardize the Dataset

K-means clustering is sensitive to the scale of features. To ensure that all features contribute equally, we standardize the dataset using `StandardScaler`.

[5]:
```
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

1.5 Perform K-means Clustering

We perform K-means clustering with `k=2` (as we know there are two classes: malignant and benign) and assign cluster labels to the data.

[6]:

```python
# Perform K-means clustering
kmeans = KMeans(n_clusters=2, random_state=42)
kmeans.fit(X_scaled)
kmeans_labels = kmeans.labels_

# Display the assigned cluster labels
print("Cluster Labels Assigned by K-means:")
print(np.unique(kmeans_labels))
```

```
Cluster Labels Assigned by K-means:
[0 1]
```

1.6 Map K-means Clusters to Actual Labels

Since K-means clusters are assigned arbitrarily, we map them to the true labels using a majority vote approach. This helps evaluate the clustering results against the ground truth.

[7]:

```python
# Map clusters to actual labels
cluster_mapping = {}
for cluster in range(2):
    cluster_indices = np.where(kmeans_labels == cluster)[0]
    cluster_labels = y[cluster_indices]
    if len(cluster_labels) > 0:
        majority_label = Counter(cluster_labels).most_common(1)[0][0]
        cluster_mapping[cluster] = majority_label

# Map the cluster labels to actual labels
mapped_labels = np.array([cluster_mapping[label] for label in kmeans_labels])
```

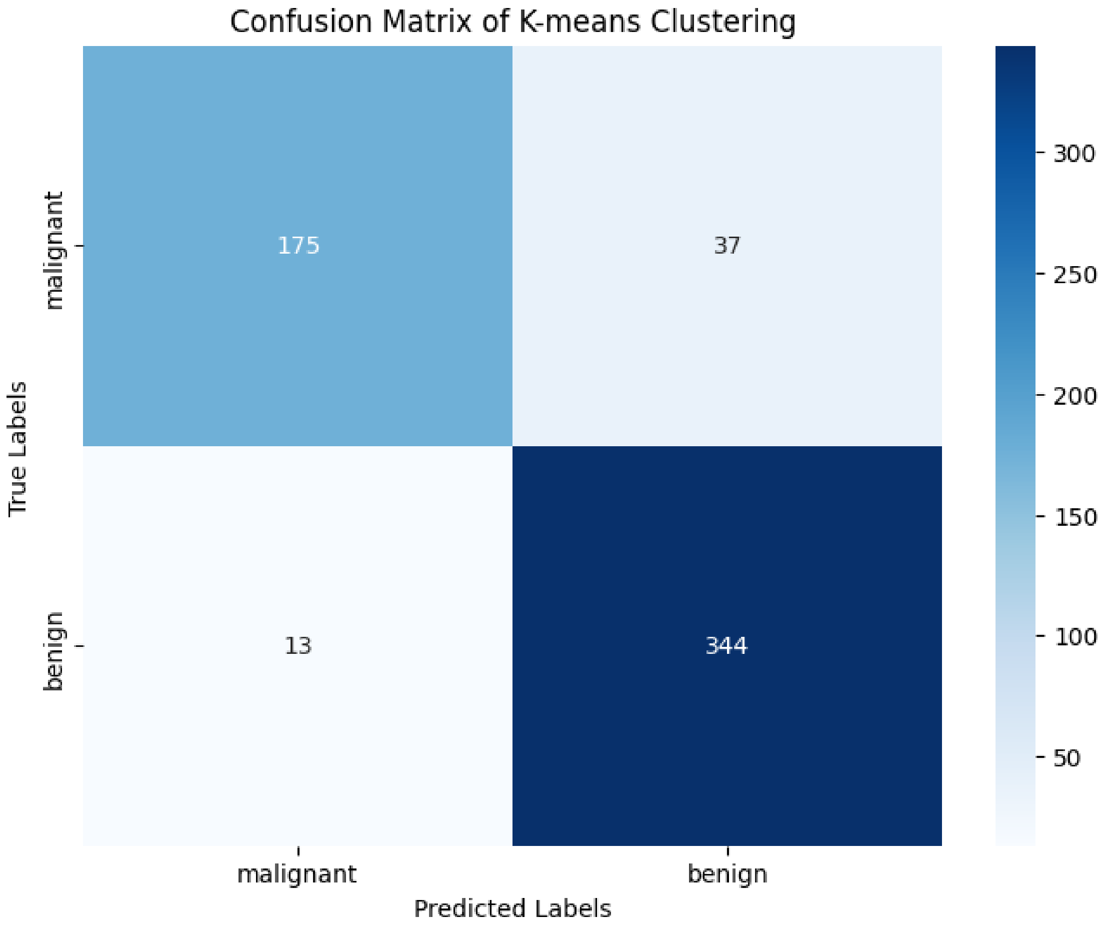
1.7 Evaluate the Clustering

We evaluate the clustering performance by calculating the accuracy and visualizing the confusion matrix.

[8]:

```python
# Calculate accuracy
accuracy = accuracy_score(y, mapped_labels)
print(f"K-means Accuracy: {accuracy:.2f}")

# Confusion matrix
conf_matrix = confusion_matrix(y, mapped_labels)

plt.figure(figsize=(8, 6))
sns.heatmap(
    conf_matrix,
    annot=True,
    fmt='d',
    cmap='Blues',
    xticklabels=target_names,
    yticklabels=target_names
)
plt.title("Confusion Matrix of K-means Clustering")
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.show()
```

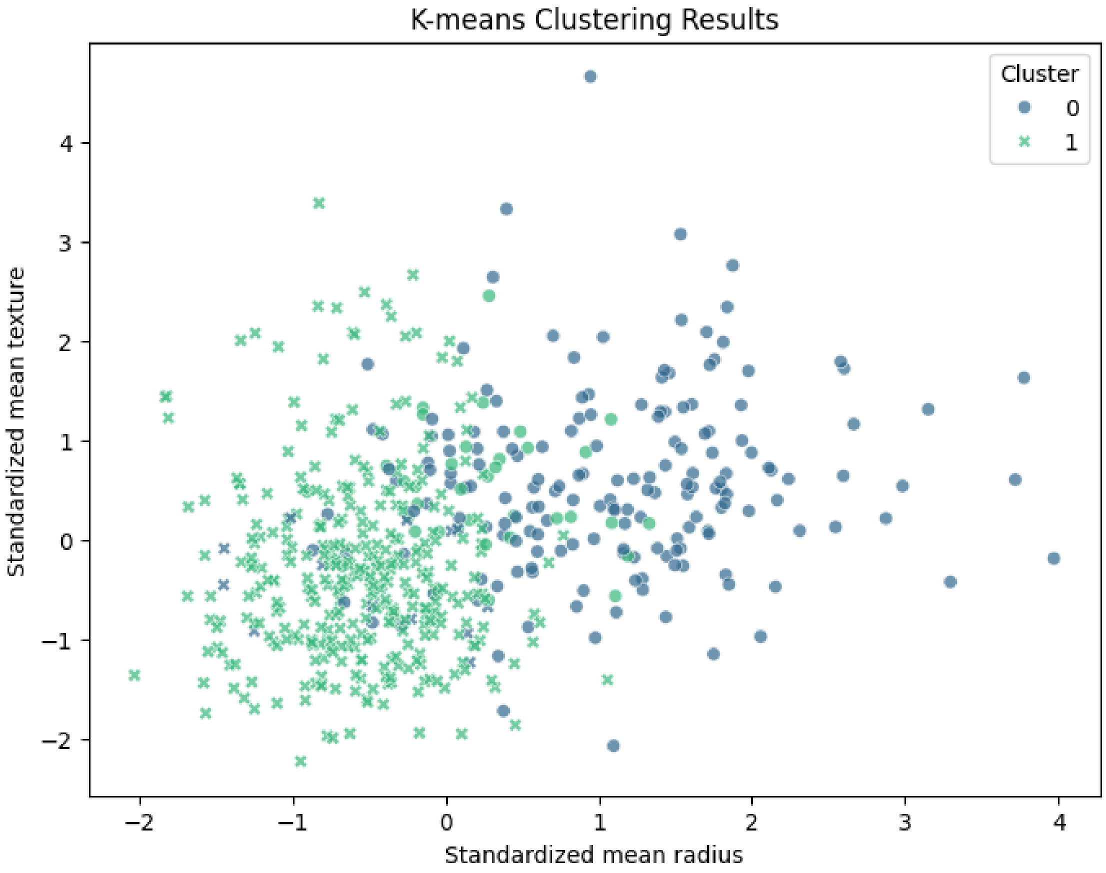
```
K-means Accuracy: 0.91
```



1.8 Visualize the Clustering Results

To understand the clusters identified by K-means, we visualize them in the feature space of `mean radius` and `mean texture`.

[9]:
```
plt.figure(figsize=(8, 6))
sns.scatterplot(
    x=X_scaled[:, 0],
    y=X_scaled[:, 1],
    hue=mapped_labels,
    palette="viridis",
    style=y,
    alpha=0.7
)
plt.title("K-means Clustering Results")
plt.xlabel("Standardized " + feature_names[0])
plt.ylabel("Standardized " + feature_names[1])
plt.legend(title="Cluster")
plt.show()
```



k-means-on-iris-dataset

November 19, 2024

1 K-means Clustering on Iris Dataset

This notebook demonstrates the use of the K-means algorithm on the Iris dataset, mapping clusters to actual class labels and evaluating the labeling error.

1.1 Import Libraries

[ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from sklearn.metrics import accuracy_score, confusion_matrix
from scipy.stats import mode
```

1.2 Load and Explore the Dataset

The Iris dataset contains measurements for three species of Iris flowers (`setosa`, `versicolor`, `virginica`). We'll explore its structure and prepare it for clustering.

[ ]:
```
# Load the Iris dataset
iris = load_iris()
X = iris.data  # Features
y = iris.target  # True labels
class_names = iris.target_names

# Convert to a DataFrame for visualization
iris_df = pd.DataFrame(X, columns=iris.feature_names)
iris_df['species'] = [class_names[label] for label in y]

# Display the first few rows of the dataset
iris_df.head()
```

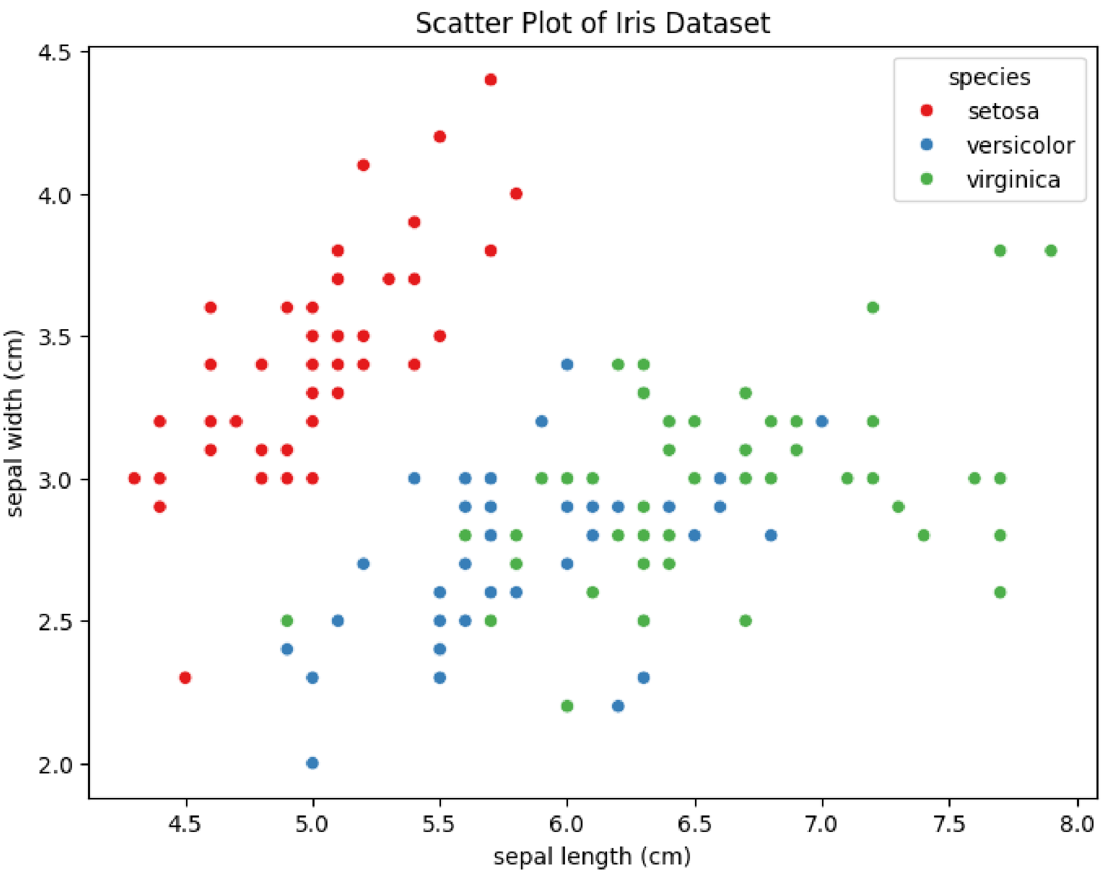
[ ]:
```
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
```

```
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

  species
0  setosa
1  setosa
2  setosa
3  setosa
4  setosa
```

1.3 Visualize the Dataset

To understand the distribution of the data, we plot the first two features (`sepal length` and `sepal width`) and color the points by their true species labels.

```
[ ]: # Scatter plot of the first two features colored by species
     plt.figure(figsize=(8, 6))
     sns.scatterplot(
         x=iris.feature_names[0],
         y=iris.feature_names[1],
         hue=iris_df['species'],
         palette='Set1',
         data=iris_df
     )
     plt.title("Scatter Plot of Iris Dataset")
     plt.xlabel(iris.feature_names[0])
     plt.ylabel(iris.feature_names[1])
     plt.show()
```



1.4 K-means Clustering

We apply the K-means algorithm, assuming the number of clusters (k=3) is known, to group the data into clusters based on their feature similarities. Before clustering, we standardize the features for better performance.

[ ]:

```
# Standardize the features for better performance
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Perform K-means clustering
kmeans = KMeans(n_clusters=3, random_state=42)
kmeans.fit(X_scaled)
kmeans_labels = kmeans.labels_
# Print K-means cluster labels
print(f"Cluster Labels Assigned by K-means: {np.unique(kmeans_labels)}")
```

1.5 Map K-means Clusters to Actual Labels

Since K-means assigns clusters arbitrarily, we map the clusters to the actual species labels by finding the most common true label in each cluster.

[ ]:

```
from collections import Counter

# Create a mapping for clusters to actual class labels
cluster_mapping = {}
for cluster in range(3):
    # Get the true labels for the data points in the current cluster
    cluster_indices = np.where(kmeans_labels == cluster)[0]
    cluster_labels = y[cluster_indices]

    if len(cluster_labels) == 0:  # Skip empty clusters
        continue

    # Find the majority label in this cluster
    majority_label = Counter(cluster_labels).most_common(1)[0][0]
    cluster_mapping[cluster] = majority_label

# Map K-means labels to actual class labels using the cluster mapping
kmeans_mapped_labels = np.array([cluster_mapping[label] for label in␣
 ↪kmeans_labels])
```

1.6 Evaluate the K-means Algorithm

We evaluate the clustering performance by calculating the accuracy and labeling error. Additionally, we compute and visualize the confusion matrix to compare the predicted and true labels.

[ ]:

```
# Calculate the accuracy of K-means clustering
accuracy = accuracy_score(y, kmeans_mapped_labels)
print(f"K-means Accuracy: {accuracy:.2f}")

# Labeling error
labeling_error = 1 - accuracy
print(f"K-means Labeling Error: {labeling_error:.2f}")
```

```
K-means Accuracy: 0.67
K-means Labeling Error: 0.33
```

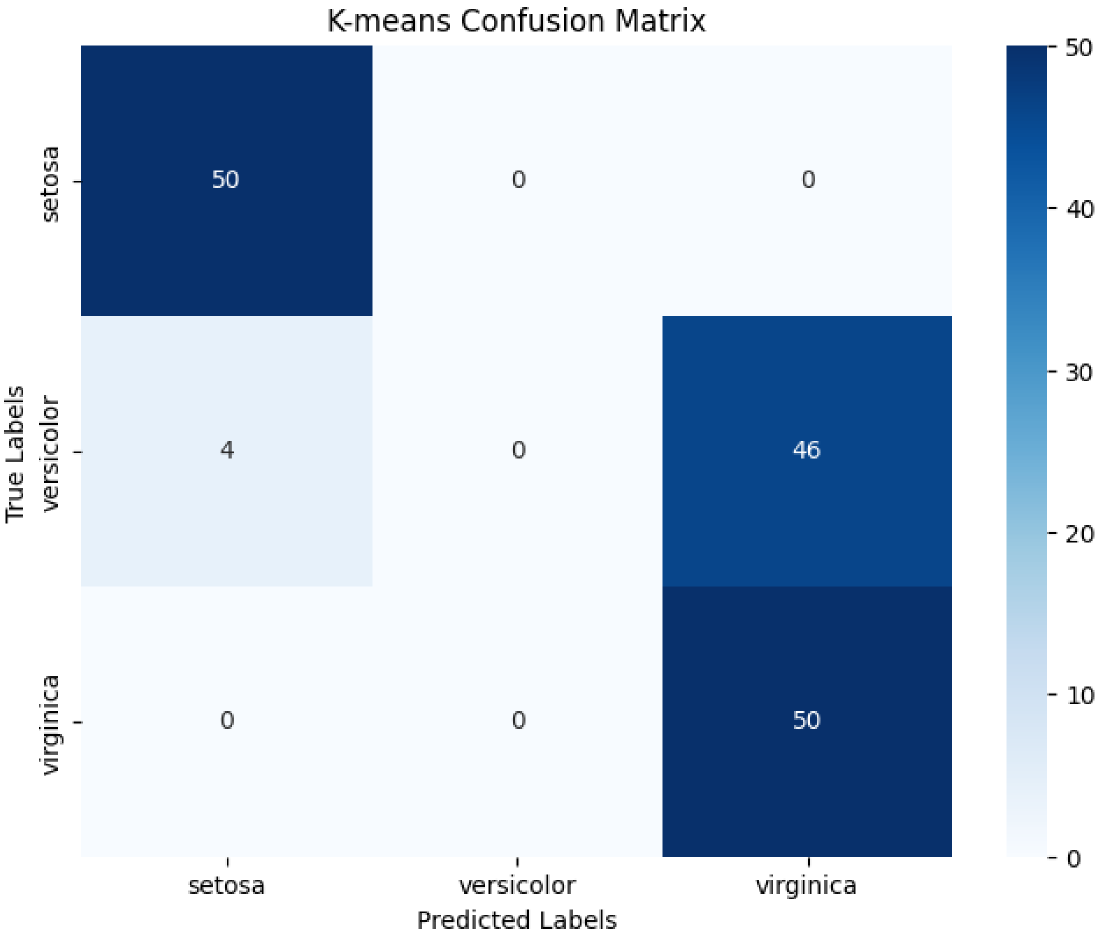
1.7 Visualize the Clustering Results

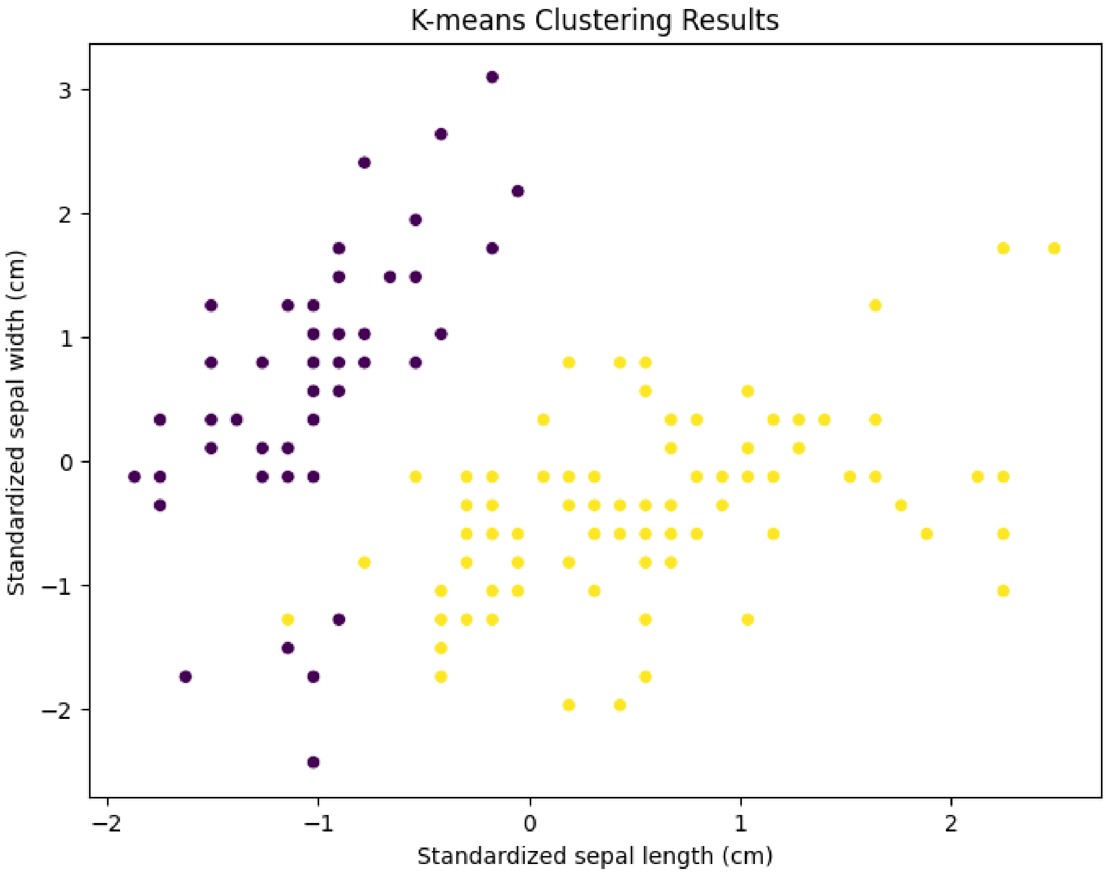
Finally, we visualize the clustering results in the first two feature dimensions. Each cluster is represented with a unique color, showcasing the separations found by K-means.

[ ]:

```
# Confusion matrix to evaluate the clustering
conf_matrix = confusion_matrix(y, kmeans_mapped_labels)
```

```
plt.figure(figsize=(8, 6))
sns.heatmap(
    conf_matrix,
    annot=True,
    fmt='d',
    cmap='Blues',
    xticklabels=class_names,
    yticklabels=class_names
)
plt.title("K-means Confusion Matrix")
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.show()
```





```
[ ]: # Visualize clusters using the first two features
plt.figure(figsize=(8, 6))
sns.scatterplot(
    x=X_scaled[:, 0],
    y=X_scaled[:, 1],
    hue=kmeans_mapped_labels,
    palette='viridis',
    legend=False
)
plt.title("K-means Clustering Results")
plt.xlabel("Standardized " + iris.feature_names[0])
plt.ylabel("Standardized " + iris.feature_names[1])
plt.show()
```