Supervised Learning II

Learning Objectives

1. Implement and evaluate classification models (Logistic Regression, K-Nearest Neighbors, Support Vector Machines, and Decision Trees) using scikit-learn to predict binary outcomes.

2. Visualize model predictions, decision boundaries, and performance metrics to understand how each algorithm separates classes and performs in classification tasks.

Introduction

This lab focuses on applying supervised learning for binary classification. Using the breast cancer dataset, you will preprocess the data, explore its characteristics, and train four classification models: Logistic Regression, K-Nearest Neighbors (KNN), Support Vector Machines (SVM), and Decision Trees. You will evaluate model performance using metrics such as accuracy, precision, recall, F1 score, confusion matrix, and ROC AUC, and visualize decision boundaries to understand class separation. By the end of this lab, you will get hands-on experience in implementation, evaluation, and behavior analysis to identify suitable algorithms for classification problems.

Dataset

- **Source:** sklearn.datasets.load_breast_cancer()
- **Description:** 569 samples, 30 numerical features (e.g., mean radius, texture), binary target (0: malignant, 1: benign).

Instructions: Lab sheet focuses on applying supervised learning for binary classification using the breast cancer dataset. It consists of 2 main tasks and 4 sub-tasks under Task 2**.**

- Sub-tasks:
    - Task 2A (Logistic Regression),
    - Task 2B (K-Nearest Neighbors)
    - Task 2C (Support Vector Machines)

- Task 2D (Decision Trees).

Each sub-task involves training a model, evaluating its performance, visualizing predictions, and analyzing behavior. Some steps include provided code segments, while others require code based on prior tasks or standard practices. You can reuse evaluation and visualization code across sub-tasks to reinforce modularity.

Task 1: Dataset Preparation

Step 1: Import Libraries

Load pandas, numpy, matplotlib, seaborn, and other relevant scikit-learn libraries to support dataset analysis and preparation for classification.

```
import warnings
warnings.filterwarnings('ignore')
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
```

Step 2: Load the Dataset

- Load the dataset

```
data = load_breast_cancer()
X = pd.DataFrame(data.data, columns=data.feature_names)
y = pd.Series(data.target)
```

- Print dataset shape (rows, columns)
- Print feature names and target names (malignant, benign)
- Visualize the top 5 rows of features and target

Step 3: Preprocess the dataset

- Check for missing values and handle them if present by dropping rows.
- Scale features using StandardScaler to standardize
- Display the top 5 rows of scaled features.

Step 4: Explore Data

- Plot class distribution to show malignant vs. benign proportions.

```
# EDA: Class distribution
print("\nClass distribution (0: malignant, 1: benign):")
print(y.value_counts(normalize=True))
plt.figure(figsize=(6, 4))
sns.countplot(x=y)
plt.title("Class Distribution (0: Malignant, 1: Benign)")
plt.show()
```

- Create a correlation heatmap for all features and target.

```
# EDA: Correlation heatmap
data_temp = X.copy()
data_temp['Target'] = y
corr_matrix = data_temp.corr()
plt.figure(figsize=(10, 8))
sns.heatmap(corr_matrix, annot=False, cmap='coolwarm', square=True)
plt.title("Correlation Heatmap")
plt.show()
```

Step 5: Split the Data in to training and testing

- Split data into 80% training and 20% testing sets with random_state=42.
- Print the number of training and testing samples.

Task 2 : Classification Models

Task 2 A: Logistic Regression

Step 6: Train Logistic Regression Model

- Fits a logistic regression model using a sigmoid function to predict binary classes (malignant/benign) based on all 30 features.
- Print the model's coefficients and intercept to examine feature contributions.

```
from sklearn.linear_model import LogisticRegression
model_lr = LogisticRegression(random_state=42)
model_lr.fit(X_train, y_train)
print("Coefficients:", model_lr.coef_)
print("Intercept:", model_lr.intercept_)
```

Step 7: Evaluate Model Performance

- Import metrics from sklearn.metrics.
- Generate predictions on X_test using the trained model.
- Compute and print accuracy, precision, recall, F1 score, and ROC AUC.

```
from sklearn.metrics import (accuracy_score, precision_score, recall_score, f1_score,
                             confusion_matrix, roc_auc_score, roc_curve)
y_pred_lr = model_lr.predict(X_test)
print("Logistic Regression Evaluation Metrics:")
print(f"Accuracy: {accuracy_score(y_test, y_pred_lr):.2f}")
print(f"Precision: {precision_score(y_test, y_pred_lr):.2f}")
print(f"Recall: {recall_score(y_test, y_pred_lr):.2f}")
print(f"F1 Score: {f1_score(y_test, y_pred_lr):.2f}")
print(f"ROC AUC: {roc_auc_score(y_test, model_lr.predict_proba(X_test)[:, 1]):.2f}")
```

Step 8: Visualize Predictions

Shows the model's linear decision boundary using two highly correlated simplifying visualization compared to PCA-based methods while highlighting class separation.

- Import required libraries for PCA, and confusion matrix visualization.
- Reduce the 30-feature dataset to 2D using PCA for visualization.

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn.decomposition import PCA

# Section 1: Reduce Dimensionality with PCA
pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)
```

- Train a logistic regression model on the 2D PCA-transformed training data to create a decision boundary.

```
model_lr_pca = LogisticRegression().fit(X_train_pca, y_train)
```

- Create a mesh grid to predict class labels across the 2D PCA space for the decision boundary.

```
x_min, x_max = X_test_pca[:, 0].min() - 1, X_test_pca[:, 0].max() + 1
y_min, y_max = X_test_pca[:, 1].min() - 1, X_test_pca[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1), np.arange(y_min, y_max, 0.1))
Z = model_lr_pca.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)
```

- Plot the decision boundary and test data points to show class separation.

```
plt.figure(figsize=(6, 4))
plt.contourf(xx, yy, Z, alpha=0.3, cmap='coolwarm')
plt.scatter(X_test_pca[:, 0], X_test_pca[:, 1], c=y_test, cmap='coolwarm', edgecolors='k')
plt.title("Logistic Regression Decision Boundary (PCA)")
plt.xlabel("PCA Component 1")
plt.ylabel("PCA Component 2")
plt.show()
```

- Plot the confusion matrix to visualize classification performance.

```
cm = confusion_matrix(y_test, y_pred_lr)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=data.target_names)
disp.plot(cmap=plt.cm.plasma)
plt.title("Confusion Matrix")
plt.show()
```

Task 2 B: K-Nearest Neighbors

Step 9: Train KNN Model

- Import KNeighborsClassifier from sklearn.neighbors.
- Initialize and fit the model with n_neighbors=5 using X_train and y_train.

```
from sklearn.neighbors import KNeighborsClassifier
model_knn = KNeighborsClassifier(n_neighbors=5)
model_knn.fit(X_train, y_train)
```

Step 10: Evaluate Model Performance

- Generate predictions on X_test using the trained model.
- Compute and print accuracy, precision, recall, F1 score, and ROC AUC.

You can reuse the code segment in Step 7. However, **adjust the variables names accordingly to avoid getting errors.

(Replace the model variable *model_lr* with *model_knn* and replace the prediction variable *y_pred_lr* with *y_pred_knn* to avoid errors)

```
y_pred_knn = model_knn.predict(X_test)
print("\nKNN Metrics:")
print(f"Accuracy: {accuracy_score(y_test, y_pred_knn):.2f}")
print(f"Precision: {precision_score(y_test, y_pred_knn):.2f}")
print(f"Recall: {recall_score(y_test, y_pred_knn):.2f}")
print(f"F1 Score: {f1_score(y_test, y_pred_knn):.2f}")
print(f"ROC AUC: {roc_auc_score(y_test, model_knn.predict_proba(X_test)[:, 1]):.2f}")
```

Step 11: Visualize Predictions

Shows the model's linear decision boundary using two highly correlated simplifying visualization compared to PCA-based methods while highlighting class separation.

You can reuse the code segments in Step 8. However, **adjust the variables names accordingly to avoid getting errors.

- Reduce the 30-feature dataset to 2D using PCA for visualization.

```
pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)
```

- Train KNN model on the 2D PCA-transformed training data to create a decision boundary.

```
model_knn_pca = KNeighborsClassifier(n_neighbors=5).fit(X_train_pca, y_train)
```

- Create a mesh grid to predict class labels across the 2D PCA space for the decision boundary.

```
x_min, x_max = X_test_pca[:, 0].min() - 1, X_test_pca[:, 0].max() + 1
y_min, y_max = X_test_pca[:, 1].min() - 1, X_test_pca[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1), np.arange(y_min, y_max, 0.1))
Z = model_knn_pca.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)
```

- Plot the decision boundary and test data points to show class separation.

```
plt.figure(figsize=(6, 4))
plt.contourf(xx, yy, Z, alpha=0.3, cmap='coolwarm')
plt.scatter(X_test_pca[:, 0], X_test_pca[:, 1], c=y_test, cmap='coolwarm', edgecolors='k')
plt.title("KNN Decision Boundary (PCA, K=5)")
plt.xlabel("PCA Component 1")
plt.ylabel("PCA Component 2")
plt.show()
```

- Plot the confusion matrix to visualize classification performance.

```
cm = confusion_matrix(y_test, y_pred_knn)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=data.target_names)
disp.plot(cmap=plt.cm.plasma)
plt.title("Confusion Matrix")
plt.show()
```

Step 12: Experiment with different K values

Experiment the behavior of KNN algorithm with different K values (1, 5, 10) and compare performance metrics.

- Loop over K values (1, 5, 10).
- For each K, train a KNN model on X_train and y_train.
- Predict on X_test and compute accuracy and F1 score.
- Print metrics for each K to compare.

```
for k in [1, 5, 10]:
    model_knn = KNeighborsClassifier(n_neighbors=k).fit(X_train, y_train)
    y_pred_knn = model_knn.predict(X_test)
    print(f"KNN (K={k}) Metrics:")
    print(f"Accuracy: {accuracy_score(y_test, y_pred_knn):.2f}, F1: {f1_score(y_test, y_pred_knn):.2f}")
```

Task 2 C: Support Vector Machines

Step 13: Train SVM Model

Train SVM models with **linear** and **RBF** kernels and display the number of support vectors.

- Import SVC from sklearn.svm.
- Loop over kernels (linear, rbf).
- For each kernel, initialize and fit an SVM model with kernel, probability=True, random_state=42 using X_train and y_train.
- Print the number of support vectors for each model.

```
from sklearn.svm import SVC
for kernel in ['linear', 'rbf']:
    model_svm = SVC(kernel=kernel, probability=True, random_state=42)
    model_svm.fit(X_train, y_train)
    print(f"\n SVM ({kernel}) Support Vectors:", len(model_svm.support_vectors_))
```

Step 14: Evaluate Model Performance

You can reuse the code segment in Step 7. However, **adjust the variables names/functions accordingly to avoid getting errors.

Note that confusion matrices are drawn at this step for SVM.

```
for kernel in ['linear', 'rbf']:
    model_svm = SVC(kernel=kernel, probability=True, random_state=42).fit(X_train, y_train)
    y_pred_svm = model_svm.predict(X_test)
    print(f"\nSVM ({kernel}) Metrics:")
    print(f"Accuracy: {accuracy_score(y_test, y_pred_svm):.2f}")
    print(f"Precision: {precision_score(y_test, y_pred_svm):.2f}")
    print(f"Recall: {recall_score(y_test, y_pred_svm):.2f}")
    print(f"F1 Score: {f1_score(y_test, y_pred_svm):.2f}")
    print(f"ROC AUC: {roc_auc_score(y_test, model_svm.predict_proba(X_test)[:, 1]):.2f}")

    cm = confusion_matrix(y_test, y_pred_svm)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=data.target_names)
    disp.plot(cmap=plt.cm.plasma)
    plt.title("Confusion Matrix")
    plt.show()
```

Step 15: Visualize Predictions

You can reuse the code segments in Step 8. However, **adjust the variables names accordingly to avoid getting errors.

- Reduce the 30-feature dataset to 2D using PCA for visualization.

```
pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)
```

- Train SVC model on the 2D PCA-transformed training data to create a decision boundary.
- Create a mesh grid to predict class labels across the 2D PCA space for the decision boundary.
- Plot the decision boundary and test data points to show class separation.

```
for kernel in ['linear', 'rbf']:

    model_svm_pca = SVC(kernel=kernel).fit(X_train_pca, y_train)

    x_min, x_max = X_test_pca[:, 0].min() - 1, X_test_pca[:, 0].max() + 1
    y_min, y_max = X_test_pca[:, 1].min() - 1, X_test_pca[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1), np.arange(y_min, y_max, 0.1))
    Z = model_svm_pca.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

    plt.figure(figsize=(6, 4))
    plt.contourf(xx, yy, Z, alpha=0.3, cmap='coolwarm')
    plt.scatter(X_test_pca[:, 0], X_test_pca[:, 1], c=y_test, cmap='coolwarm', edgecolors='k')
    plt.title(f"SVM Decision Boundary ({kernel}, PCA)")
    plt.xlabel("PCA Component 1")
    plt.ylabel("PCA Component 2")
    plt.show()
```

Step 16: Experiment with Parameters and Analyze Behavior

Experiment with the regularization parameter **C** for the RBF kernel and reflect on SVM's behavior.

```
for C in [0.1, 1, 10]:
    model_svm = SVC(kernel='rbf', C=C, probability=True, random_state=42).fit(X_train, y_train)
    y_pred_svm = model_svm.predict(X_test)
    print(f"SVM (RBF, C={C}) Metrics:")
    print(f"Accuracy: {accuracy_score(y_test, y_pred_svm):.2f}, F1: {f1_score(y_test, y_pred_svm):.2f}")
```

Task 2 D: Decision Trees

Step 17: Train Decision Tree Model

- Import DecisionTreeClassifier from sklearn.tree.
- Initialize and fit the model with random_state=42 using X_train and y_train.

```
from sklearn.tree import DecisionTreeClassifier
model_dt = DecisionTreeClassifier(random_state=42)
model_dt.fit(X_train, y_train)
print("\nTree Depth:", model_dt.get_depth())
print("Feature Importance:", pd.Series(model_dt.feature_importances_, index=X.columns))
```

Step 18: Evaluate Model Performance

You can reuse the code segment in Step 7. However, **adjust the variables names/functions accordingly to avoid getting errors

```
y_pred_dt = model_dt.predict(X_test)
print("\nDecision Tree Metrics:")
print(f"Accuracy: {accuracy_score(y_test, y_pred_dt):.2f}")
print(f"Precision: {precision_score(y_test, y_pred_dt):.2f}")
print(f"Recall: {recall_score(y_test, y_pred_dt):.2f}")
print(f"F1 Score: {f1_score(y_test, y_pred_dt):.2f}")
print(f"ROC AUC: {roc_auc_score(y_test, model_dt.predict_proba(X_test)[:, 1]):.2f}")
```

Step 19: Visualize Predictions

You can reuse the code segments in Step 8. However, **adjust the variables names accordingly to avoid getting errors.

- Plot the Decision Tree structure to visualize its hierarchical splits

```
from sklearn.tree import plot_tree
plt.figure(figsize=(12, 8))
plot_tree(model_dt, feature_names=X.columns,
          class_names=['malignant', 'benign'],
          filled=True)
plt.title("Decision Tree Structure")
plt.show()
```

- Reduce the 30-feature dataset to 2D using PCA for the decision boundary visualization

```
pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)
```

- Train a Decision Tree model on the 2D PCA-transformed training data to create a decision boundary

```
model_dt_pca = DecisionTreeClassifier(random_state=42).fit(X_train_pca, y_train)
```

- Create a mesh grid to predict class labels across the 2D PCA space for the decision boundary.

```
x_min, x_max = X_test_pca[:, 0].min() - 1, X_test_pca[:, 0].max() + 1
y_min, y_max = X_test_pca[:, 1].min() - 1, X_test_pca[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1), np.arange(y_min, y_max, 0.1))
Z = model_dt_pca.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)
```

- Plot the decision boundary and test data points to show class separation.

```
plt.figure(figsize=(6, 4))
plt.contourf(xx, yy, Z, alpha=0.3, cmap='coolwarm')
plt.scatter(X_test_pca[:, 0], X_test_pca[:, 1], c=y_test, cmap='coolwarm', edgecolors='k')
plt.title("Decision Tree Decision Boundary (PCA)")
plt.xlabel("PCA Component 1")
plt.ylabel("PCA Component 2")
plt.show()
```

- Plot the confusion matrix to visualize classification performance

```
cm = confusion_matrix(y_test, y_pred_dt)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=data.target_names)
disp.plot(cmap=plt.cm.plasma)
plt.title("Confusion Matrix")
plt.show()
```

Step 20: Experiment with Parameters and Analyze Behavior

Experiment with the max_depth parameter and reflect on Decision Tree's behavior.

- Loop over max_depth values (1, 3, 10) for the Decision Tree model.
- For each max_depth, train a model, predict on X_test, and compute accuracy and F1 score.
- Print metrics for each max_depth to compare.

```
for depth in [1, 3, 10]:
    model_dt = DecisionTreeClassifier(max_depth=depth, random_state=42).fit(X_train, y_train)
    y_pred_dt = model_dt.predict(X_test)
    print(f"Decision Tree (max_depth={depth}) Metrics:")
    print(f"Accuracy:{accuracy_score(y_test, y_pred_dt):.2f},F1: {f1_score(y_test, y_pred_dt):.2f}")
```

Question

Analyze the performance matrices accuracy, F1 score, ROC AUC and suitability of Logistic Regression, KNN, SVM, and Decision Trees for the breast cancer dataset, considering class imbalance and separability. Evaluate which model is best for this binary classification task, explaining why based on generalization and overfitting risks.

Recommend one improvement for the weakest model.