Training and Testing (Validating) the Classification Models

**Models to be used: **

Part1

- Logistic Regression

- Logistic Lasso

- Logistic Ridge

Part2

- Support Vector Machine (SVM)

- Random Forest: RandomForestClassifier

- Gradient Boosting': GradientBoostingClassifier(),

- KNN: KNeighborsClassifier(), and

- Naive Bayes': GaussianNB()

```python
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split

label=employee_df['LeaveOrNot']

# Create the interaction feature
employee_df['GenderEverBenched'] = employee_df['Gender'] * employee_df['EverBenched']

# Separate the features and target variable
features = employee_df.drop('LeaveOrNot', axis=1)
target = employee_df['LeaveOrNot']

# Split the dataset into training and validation (testing) sets
X_train, X_val, label_train, label_val = train_test_split(features, target, test_size=0.20, random_state=42)

# Check the shapes of the resulting datasets
print("Training set shapes:", X_train.shape, label_train.shape)
print("Validation set shapes:", X_val.shape, label_val.shape)

# Instantiate the StandardScaler
scaler = StandardScaler()

# Fit the scaler to the training data and transform it
X_train_scaled = scaler.fit_transform(X_train)

# Transform the validation (testing) data using the same scaler
X_val_scaled = scaler.transform(X_val)

# Now, X_train_scaled and X_val_scaled are the standardized versions of
# the training and validation datasets
```

```
Training set shapes: (3722, 11) (3722,)
Validation set shapes: (931, 11) (931,)
```

This code performs several preprocessing steps to prepare the dataset for machine learning. Each step plays an essential role in ensuring the data is in the right format, and the features are appropriately scaled for the machine learning algorithm.

1. **from sklearn.preprocessing import LabelEncoder, StandardScaler**: Imports the necessary preprocessing utilities from Scikit-learn. **LabelEncoder** is used to encode categorical features, and **StandardScaler** is used to standardize the features by scaling them to a mean of 0 and a standard deviation of 1.

2. **label=employee_df['LeaveOrNot']**: Extracts the target variable (whether an employee leaves or not) and stores it in a separate variable called **label**.

3. **employee_df['GenderEverBenched']=employee_df['Gender']*employee_df['EverBenched']**: Creates a new interaction feature called **GenderEverBenched** by multiplying the **Gender** and **EverBenched** features. Interaction features can capture relationships between different features that might not be apparent when they are considered individually.

4. **features = employee_df.drop('LeaveOrNot', axis=1)**: Removes the target variable 'LeaveOrNot' from the dataset to create a separate feature set.

5. **target = employee_df['LeaveOrNot']**: Stores the target variable in a separate variable called **target**.

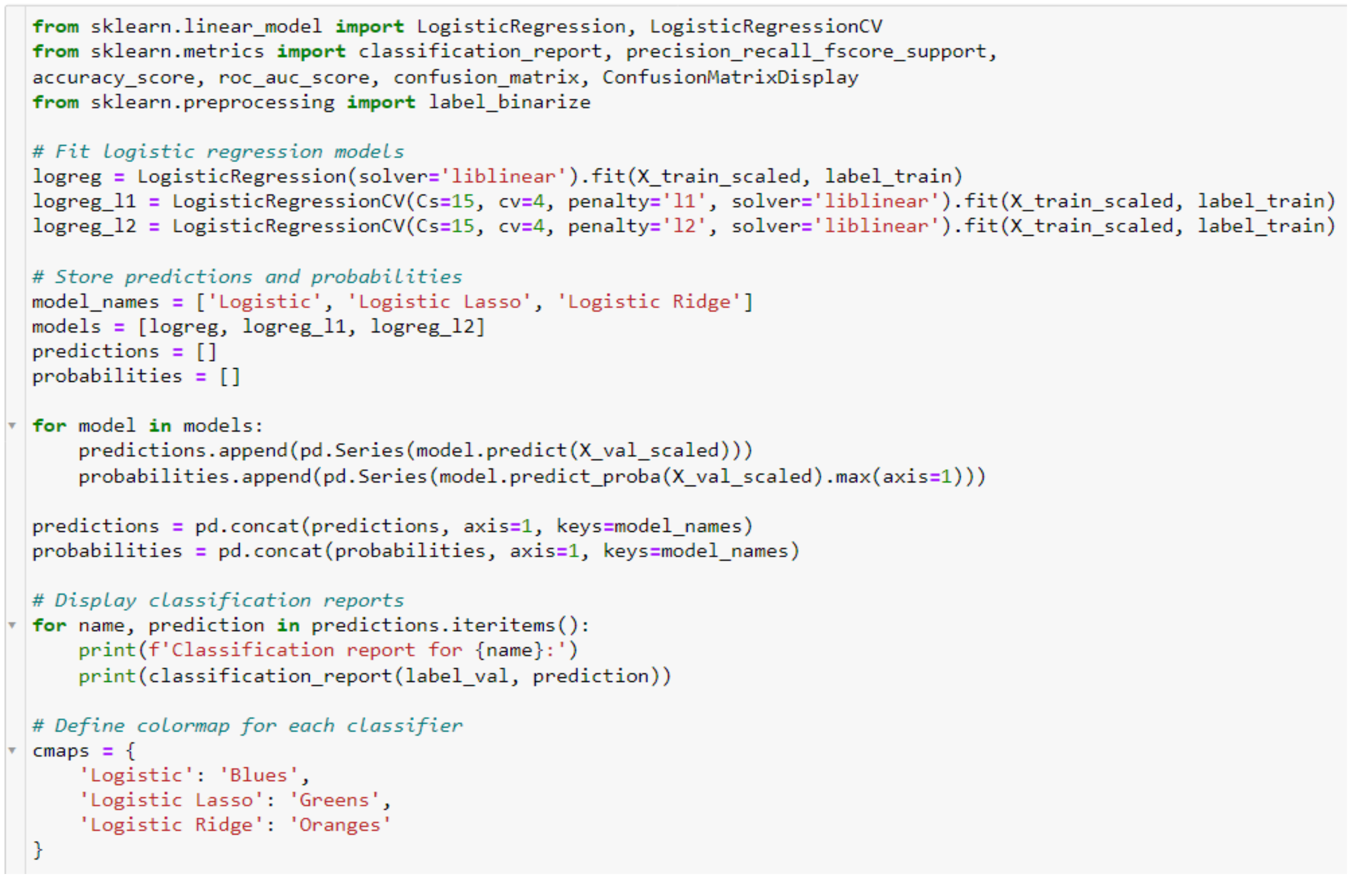
6. **train_test_split(features, target, test_size=0.20, random_state=42)**: Splits the dataset into a training set and a validation set with an 80-20% ratio. This allows us to train the model on one subset of the data and evaluate its performance on a separate, unseen subset.

7. **scaler = StandardScaler()**: Instantiates the **StandardScaler** object, which will be used to standardize the features.

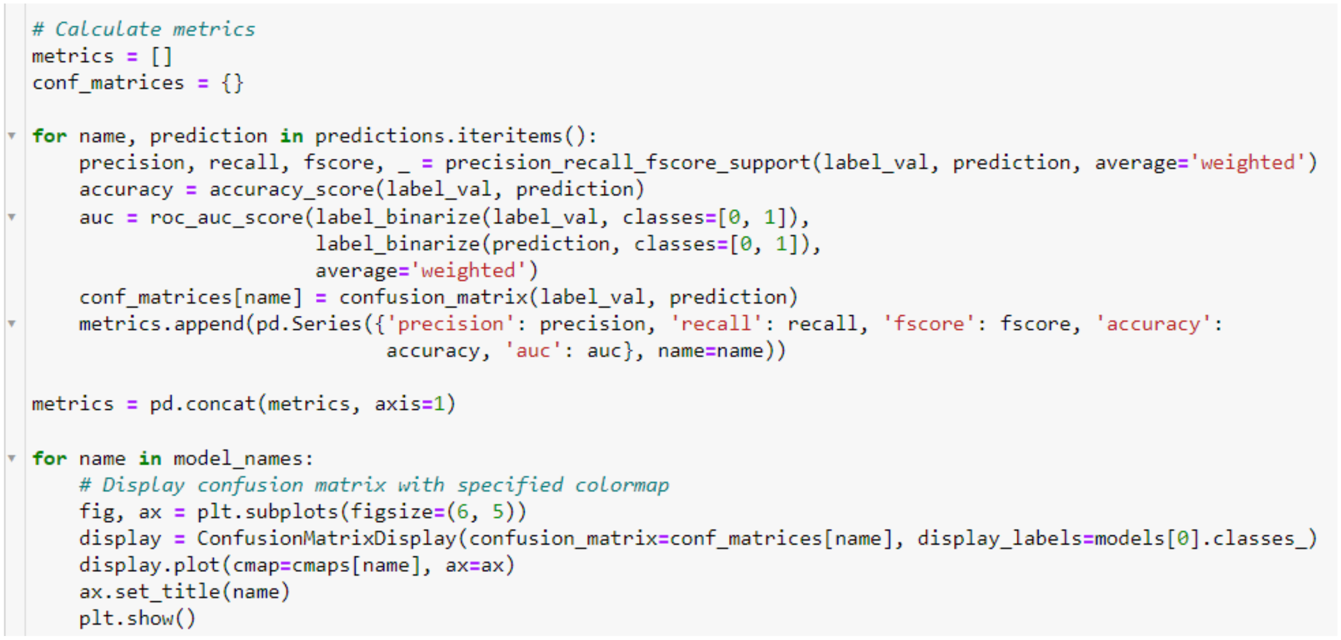
8. **X_train_scaled = scaler.fit_transform(X_train)**: Fits the **StandardScaler** object to the training data and then transforms it. The training data is now standardized, with a mean of 0 and a standard deviation of 1.

9. **X_val_scaled = scaler.transform(X_val)**: Transforms the validation data using the same **StandardScaler** object that was fit on the training data. This ensures the validation data is standardized in the same way as the training data.

The importance of these preprocessing steps lies in preparing the data for machine learning algorithms. Standardizing the features helps ensure that the algorithm converges more quickly and provides more accurate results. Splitting the data into training and validation sets allows us to evaluate the model's performance on unseen data and helps prevent overfitting. Creating interaction features can help capture more complex relationships between features that can lead to better model performance.



```python
from sklearn.linear_model import LogisticRegression, LogisticRegressionCV
from sklearn.metrics import classification_report, precision_recall_fscore_support,
accuracy_score, roc_auc_score, confusion_matrix, ConfusionMatrixDisplay
from sklearn.preprocessing import label_binarize

# Fit logistic regression models
logreg = LogisticRegression(solver='liblinear').fit(X_train_scaled, label_train)
logreg_l1 = LogisticRegressionCV(Cs=15, cv=4, penalty='l1', solver='liblinear').fit(X_train_scaled, label_train)
logreg_l2 = LogisticRegressionCV(Cs=15, cv=4, penalty='l2', solver='liblinear').fit(X_train_scaled, label_train)

# Store predictions and probabilities
model_names = ['Logistic', 'Logistic Lasso', 'Logistic Ridge']
models = [logreg, logreg_l1, logreg_l2]
predictions = []
probabilities = []

for model in models:
    predictions.append(pd.Series(model.predict(X_val_scaled)))
    probabilities.append(pd.Series(model.predict_proba(X_val_scaled).max(axis=1)))

predictions = pd.concat(predictions, axis=1, keys=model_names)
probabilities = pd.concat(probabilities, axis=1, keys=model_names)

# Display classification reports
for name, prediction in predictions.iteritems():
    print(f'Classification report for {name}:')
    print(classification_report(label_val, prediction))

# Define colormap for each classifier
cmaps = {
    'Logistic': 'Blues',
    'Logistic Lasso': 'Greens',
    'Logistic Ridge': 'Oranges'
}
```



```python
# Calculate metrics
metrics = []
conf_matrices = {}

for name, prediction in predictions.iteritems():
    precision, recall, fscore, _ = precision_recall_fscore_support(label_val, prediction, average='weighted')
    accuracy = accuracy_score(label_val, prediction)
    auc = roc_auc_score(label_binarize(label_val, classes=[0, 1]),
                        label_binarize(prediction, classes=[0, 1]),
                        average='weighted')
    conf_matrices[name] = confusion_matrix(label_val, prediction)
    metrics.append(pd.Series({'precision': precision, 'recall': recall, 'fscore': fscore, 'accuracy':
                              accuracy, 'auc': auc}, name=name))

metrics = pd.concat(metrics, axis=1)

for name in model_names:
    # Display confusion matrix with specified colormap
    fig, ax = plt.subplots(figsize=(6, 5))
    display = ConfusionMatrixDisplay(confusion_matrix=conf_matrices[name], display_labels=models[0].classes_)
    display.plot(cmap=cmaps[name], ax=ax)
    ax.set_title(name)
    plt.show()
```

This code trains logistic regression models, evaluates their performance, and visualizes the confusion matrices. Each step helps in understanding the performance of the models and their ability to predict the target variable accurately.

1. **from sklearn.linear_model import LogisticRegression, LogisticRegressionCV**: Imports the necessary classes for logistic regression and logistic regression with cross-validation (CV).
2. **from sklearn.metrics import classification_report, ...**: Imports various performance evaluation metrics and utilities such as **classification_report**, **confusion_matrix**, and **ConfusionMatrixDisplay**.
3. **from sklearn.preprocessing import label_binarize**: Imports the utility for binarizing labels. This is useful for computing the ROC AUC score for multi-class classification problems.
4. **logreg = LogisticRegression(...).fit(...)**: Trains a logistic regression model with the 'liblinear' solver on the standardized training dataset.
5. **logreg_l1 = LogisticRegressionCV(...).fit(...)**: Trains a logistic regression model with L1 regularization (Lasso) using cross-validation to select the best hyperparameters.
6. **logreg_l2 = LogisticRegressionCV(...).fit(...)**: Trains a logistic regression model with L2 regularization (Ridge) using cross-validation to select the best hyperparameters.
7. The loop stores predictions and probabilities for each model in lists, which are later concatenated into DataFrames for further processing and visualization.
8. The loop prints the classification report for each model, which includes various performance metrics such as precision, recall, and F1-score.
9. The **cmaps** dictionary assigns a colormap to each classifier for visualization purposes.
10. The loop calculates performance metrics (precision, recall, F-score, accuracy, and AUC) for each model and stores them in a list. These metrics are later concatenated into a DataFrame called **metrics**.
11. The loop displays a confusion matrix for each classifier using the specified colormap, giving a visual representation of the classifier's performance.

The importance of this code lies in training and evaluating different logistic regression models and comparing their performance. This allows you to understand how regularization affects the model performance and identify the best model. Visualizing confusion matrices helps in identifying the specific types of errors the models make, which can inform further model improvements.

****

This is a comparative analysis of three classification models applied to the employee_df dataset: Logistic Regression, Lasso Regression (L1 Regularization), and Ridge Regression (L2 Regularization). The performance of each model is evaluated using their confusion matrices, which display the number of true positives, true negatives, false positives, and false negatives.

**When comparing the three models, we can observe the following insights: **

All three models have the same accuracy of 0.74.

****

**The Lasso Regression model has the highest number of True Negatives (553) ** and the lowest number of False Positives (57), which indicates that **it is slightly better at predicting class 0 compared to the other models. **

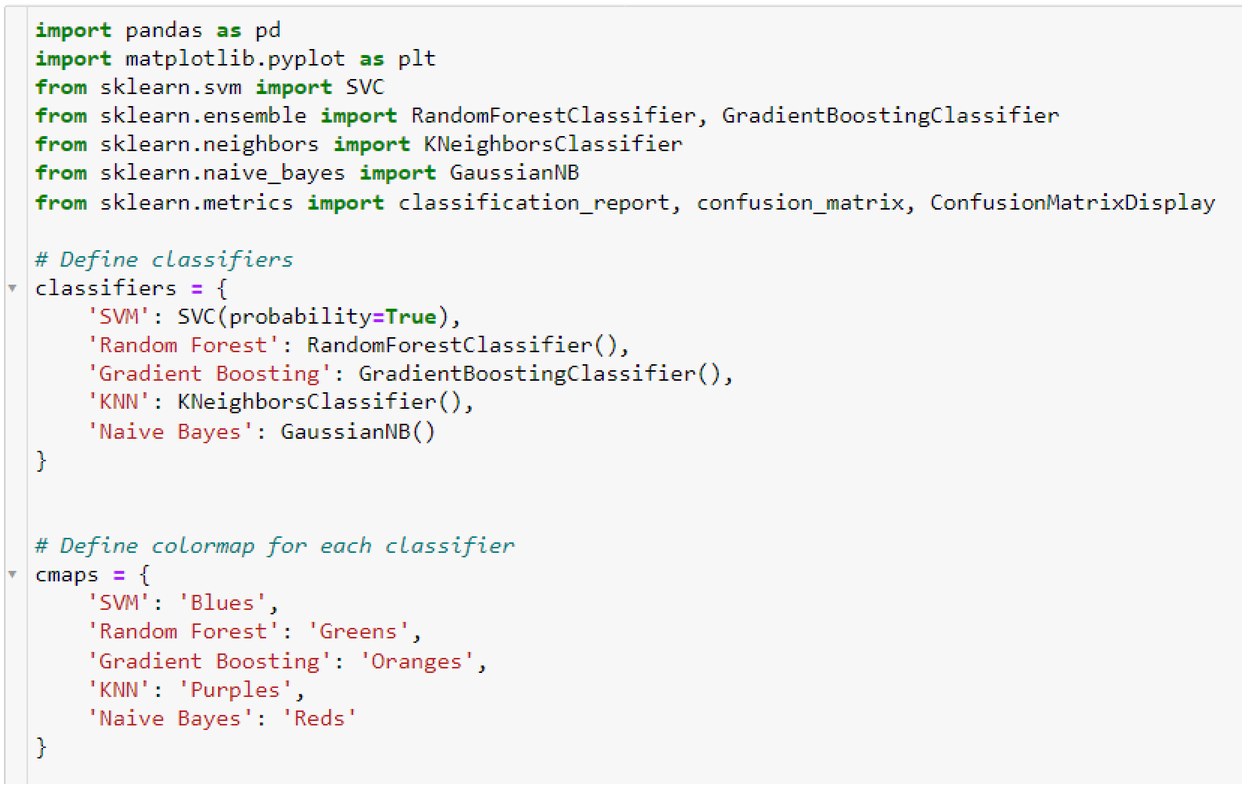
The **Logistic Regression and Lasso Regression models both have the same number of True Positives (139) and False Negatives (182), indicating that they have similar performance in predicting class 1. **

****

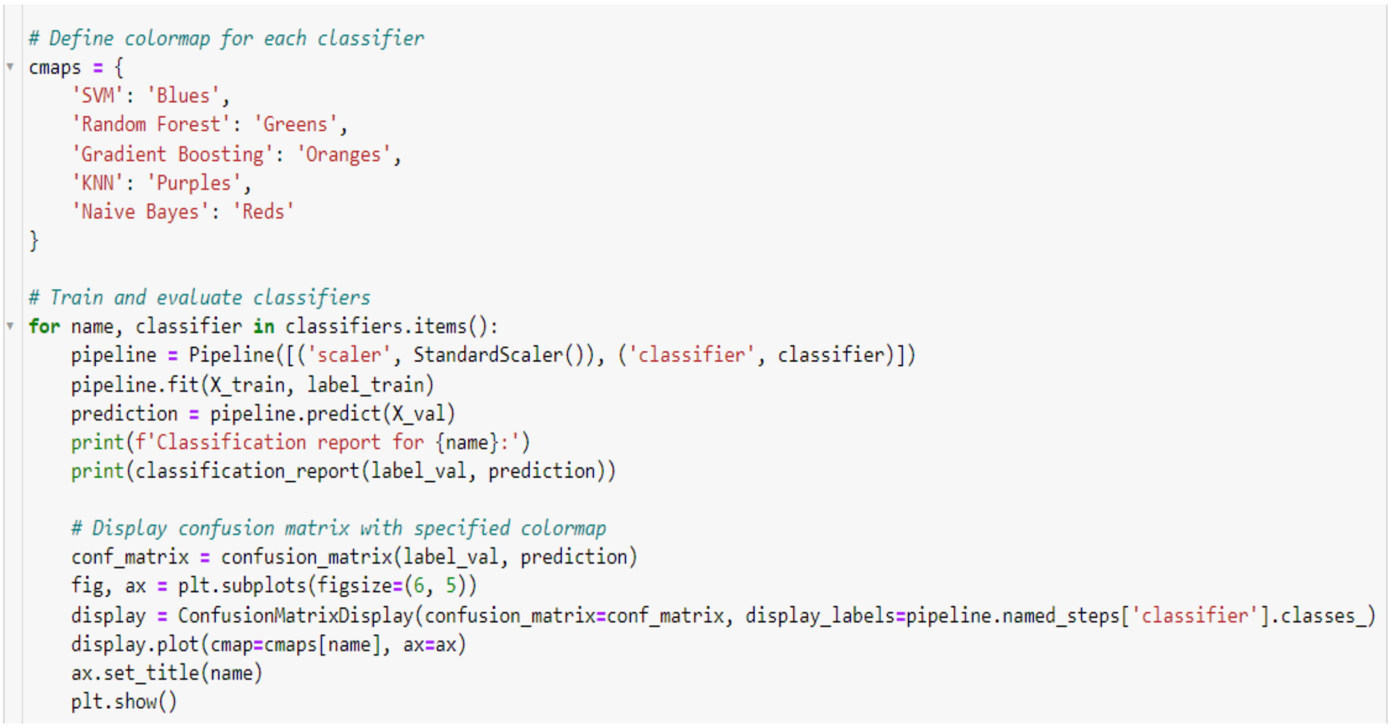
The Ridge Regression model has the lowest number of True Positives (138) and the highest number of False Negatives (183), which suggests that it is slightly worse at predicting class 1 compared to the other models.

Conclusion:

Based on the analysis of the confusion matrices, the Logistic Lasso Regression model appears to have the best overall performance in terms of true predictions, closely followed by the Logistic Regression model. The Logistic Ridge Regression model demonstrates slightly lower performance in predicting class 1.



```python
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import classification_report, confusion_matrix, ConfusionMatrixDisplay

# Define classifiers
classifiers = {
    'SVM': SVC(probability=True),
    'Random Forest': RandomForestClassifier(),
    'Gradient Boosting': GradientBoostingClassifier(),
    'KNN': KNeighborsClassifier(),
    'Naive Bayes': GaussianNB()
}


# Define colormap for each classifier
cmaps = {
    'SVM': 'Blues',
    'Random Forest': 'Greens',
    'Gradient Boosting': 'Oranges',
    'KNN': 'Purples',
    'Naive Bayes': 'Reds'
}
```



```python
# Define colormap for each classifier
cmaps = {
    'SVM': 'Blues',
    'Random Forest': 'Greens',
    'Gradient Boosting': 'Oranges',
    'KNN': 'Purples',
    'Naive Bayes': 'Reds'
}

# Train and evaluate classifiers
for name, classifier in classifiers.items():
    pipeline = Pipeline([('scaler', StandardScaler()), ('classifier', classifier)])
    pipeline.fit(X_train, label_train)
    prediction = pipeline.predict(X_val)
    print(f'Classification report for {name}:')
    print(classification_report(label_val, prediction))

    # Display confusion matrix with specified colormap
    conf_matrix = confusion_matrix(label_val, prediction)
    fig, ax = plt.subplots(figsize=(6, 5))
    display = ConfusionMatrixDisplay(confusion_matrix=conf_matrix, display_labels=pipeline.named_steps['classifier'].classes_)
    display.plot(cmap=cmaps[name], ax=ax)
    ax.set_title(name)
    plt.show()
```

This code trains and evaluates multiple classification algorithms, including SVM, Random Forest, Gradient Boosting, KNN, and Naive Bayes. It also visualizes the confusion matrices for each classifier. Each step helps in understanding the performance of different models and their ability to predict the target variable accurately.

1. **from sklearn.svm import SVC**: Imports the Support Vector Machine (SVM) classifier class.
2. **from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier**: Imports the RandomForest and GradientBoosting classifiers from the ensemble module.
3. **from sklearn.neighbors import KNeighborsClassifier**: Imports the K-Nearest Neighbors (KNN) classifier.
4. **from sklearn.naive_bayes import GaussianNB**: Imports the Gaussian Naive Bayes classifier.
5. **from sklearn.metrics import classification_report, confusion_matrix, ConfusionMatrixDisplay**: Imports various performance evaluation metrics and utilities.
6. **classifiers** is a dictionary that contains instances of the classifiers we want to train and evaluate.
7. **cmaps** is a dictionary that assigns a colormap to each classifier for visualization purposes.
8. The loop iterates through the classifiers, creating a pipeline for each classifier. It trains the classifier on the training dataset (**X_train**, **label_train**) and makes predictions on the validation dataset (**X_val**).
9. The classification report for each classifier is printed, which includes various performance metrics such as precision, recall, and F1-score.
10. The confusion matrix is displayed for each classifier using the specified colormap, giving a visual representation of the classifier's performance.

The importance of this code lies in training, evaluating, and comparing different classification models, which allows you to understand their performance and identify the best model for the task at hand. Visualizing confusion matrices helps in identifying the specific types of errors the models make, which can inform further model improvements.

When comparing the five models based on precision, recall, and F1-score, we observe that the Support Vector Machines (SVM), Random Forest, and Gradient Boosting models perform similarly, with an accuracy of 0.85. These three models also exhibit the highest weighted average F1-scores of 0.84, 0.84, and 0.85, respectively.

The K-Nearest Neighbors (KNN) model performs slightly worse than the top three models, with an accuracy of 0.84 and a weighted average F1-score of 0.83. However, it still outperforms the Naive Bayes model, which has an accuracy of 0.71 and a weighted average F1-score of 0.71.

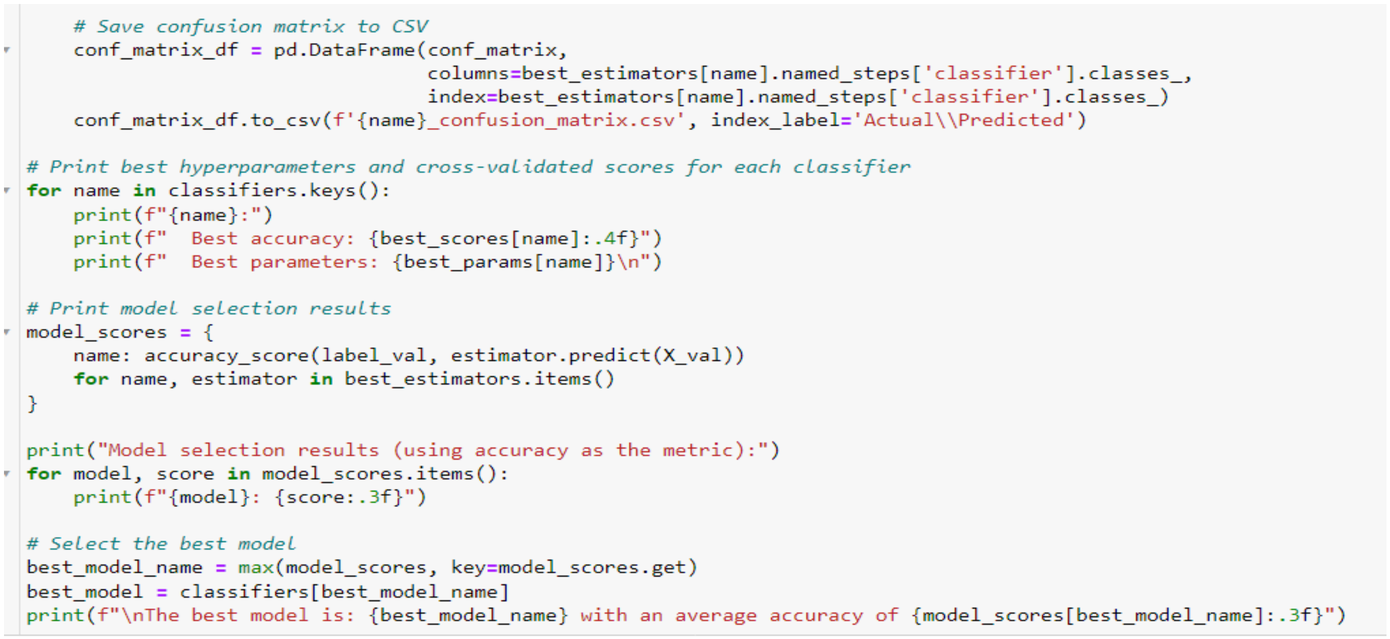
In terms of class-specific performance, the SVM model has the highest precision for class 1 (0.92), while the Gradient Boosting model has the highest precision for class 0 (0.85). Random Forest has the highest recall for class 1 (0.72), while SVM has the highest recall for class 0 (0.97).

Conclusion:

Based on the analysis, the Support Vector Machines (SVM), Random Forest, and Gradient Boosting models exhibit the best performance in this classification task. Each model has its own strengths in terms of precision and recall, depending on the class of interest. However, it is important to consider the specific use case and the trade-offs between precision and recall when selecting the best model for a given application. **In cases where a balance between precision and recall is desired, the Gradient Boosting model may be preferred due to its higher macro average F1-score (0.83) compared to SVM and Random Forest models.**

Hyperparameter Tuning for Classification Models

```python
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix

# Define colormap for each classifier
cmaps = {
    'SVM': 'Blues',
    'Random Forest': 'Greens',
    'Gradient Boosting': 'Oranges',
    'KNN': 'Purples',
    'Naive Bayes': 'Reds'
}

# Train and evaluate classifiers with cross-validated grid search
best_estimators = {}
best_scores = {}
best_params = {}

for name, classifier in classifiers.items():
    pipeline = Pipeline([('scaler', StandardScaler()), ('classifier', classifier)])
    grid_search = GridSearchCV(pipeline, param_grid=param_grids[name], scoring='accuracy', cv=5, n_jobs=-1, verbose=1)
    grid_search.fit(X_train, label_train)

    best_estimators[name] = grid_search.best_estimator_
    best_scores[name] = grid_search.best_score_
    best_params[name] = grid_search.best_params_

    # Evaluate the classifiers on the validation set
    prediction = best_estimators[name].predict(X_val)
    print(f'Classification report for {name}:')
    print(classification_report(label_val, prediction))

    # Display confusion matrix with specified colormap
    conf_matrix = confusion_matrix(label_val, prediction)
    fig, ax = plt.subplots(figsize=(6, 5))
    display = ConfusionMatrixDisplay(confusion_matrix=conf_matrix,
                                     display_labels=best_estimators[name].named_steps['classifier'].classes_)
    display.plot(cmap=cmaps[name], ax=ax)
    ax.set_title(name)
    plt.show()
```



```python
    # Save confusion matrix to CSV
    conf_matrix_df = pd.DataFrame(conf_matrix,
                                  columns=best_estimators[name].named_steps['classifier'].classes_,
                                  index=best_estimators[name].named_steps['classifier'].classes_)
    conf_matrix_df.to_csv(f'{name}_confusion_matrix.csv', index_label='Actual\\Predicted')

# Print best hyperparameters and cross-validated scores for each classifier
for name in classifiers.keys():
    print(f"{name}:")
    print(f"  Best accuracy: {best_scores[name]:.4f}")
    print(f"  Best parameters: {best_params[name]}\n")

# Print model selection results
model_scores = {
    name: accuracy_score(label_val, estimator.predict(X_val))
    for name, estimator in best_estimators.items()
}

print("Model selection results (using accuracy as the metric):")
for model, score in model_scores.items():
    print(f"{model}: {score:.3f}")

# Select the best model
best_model_name = max(model_scores, key=model_scores.get)
best_model = classifiers[best_model_name]
print(f"\nThe best model is: {best_model_name} with an average accuracy of {model_scores[best_model_name]:.3f}")
```

This code performs hyperparameter tuning for the classification models using cross-validated grid search. It trains, evaluates, and compares the performance of the classifiers with different hyperparameter configurations to find the best hyperparameters for each model.

1. **cmaps** is a dictionary that assigns a colormap to each classifier for visualization purposes.
2. **best_estimators**, **best_scores**, and **best_params** are dictionaries that will store the best estimator, the best cross-validated score, and the best hyperparameters for each classifier, respectively.
3. The loop iterates through the classifiers, creating a pipeline for each classifier and performing grid search cross-validation (**GridSearchCV**). It fits the grid search object to the training data (**X_train**, **label_train**).
4. The best estimator, best cross-validated score, and best hyperparameters for each classifier are stored in the corresponding dictionaries.
5. Each classifier's performance is evaluated on the validation dataset (**X_val**), and the classification report is printed.
6. The confusion matrix is displayed for each classifier using the specified colormap, giving a visual representation of the classifier's performance.
7. The confusion matrix is saved as a CSV file for each classifier.
8. The best hyperparameters and cross-validated scores for each classifier are printed.
9. **model_scores** is a dictionary that stores the accuracy of the best estimator for each classifier on the validation set.
10. The model selection results are printed using accuracy as the metric.
11. The best model is selected based on the highest accuracy on the validation set.

The importance of this code lies in tuning the hyperparameters of different classification models, which helps in improving their performance. By comparing the performance of models with different hyperparameters, you can identify the best configuration for the task at hand. Additionally, visualizing the confusion matrices helps in identifying the specific types of errors the models make, which can inform further model improvements. Finally, selecting the best model based on validation set performance allows you to choose the most accurate classifier for your problem.

****

Now let's evaluate the information above:

Here, I will evaluate the performance of five different classification models: Support Vector Machines (SVM), Random Forest, Gradient Boosting, K-Nearest Neighbors (KNN), and Naive Bayes. We use grid search cross-validation with five folds to find the optimal hyperparameters for each model. The total number of fits for each model is as follows:

- SVM: 80 fits
- Random Forest: 720 fits
- Gradient Boosting: 1620 fits
- KNN: 200 fits
- Naive Bayes: 20 fits<br>

We assess the performance of each model on the validation set using precision, recall, F1-score, and accuracy. The confusion matrix results are also provided for each model.

****

SVM achieves an accuracy of 0.860, with the best parameters being C=10, gamma='scale', and kernel='rbf'. The classification report shows high precision and recall scores for both classes, with an F1-score of 0.90 for class 0 and 0.76 for class 1.

****

Random Forest achieves an accuracy of 0.868, with the best parameters being max_depth=20, min_samples_leaf=2, min_samples_split=10, and n_estimators=50. The classification report demonstrates high precision and recall scores for both classes, with an F1-score of 0.91 for class 0 and 0.78 for class 1.

****

Gradient Boosting achieves the highest accuracy of 0.869, with the best parameters being learning_rate=0.1, max_depth=5, min_samples_leaf=4, min_samples_split=2, and n_estimators=50. The classification report reveals high precision and recall scores for both classes, with an F1-score of 0.91 for class 0 and 0.78 for class 1.

****

KNN achieves an accuracy of 0.837, with the best parameters being algorithm='brute', n_neighbors=11, and weights='distance'. The classification report displays relatively high precision and recall scores for both classes, with an F1-score of 0.88 for class 0 and 0.74 for class 1.

****

Naive Bayes achieves the lowest accuracy of 0.714, with the best parameter being var_smoothing=1e-09. The classification report shows moderate precision and recall scores for both classes, with an F1-score of 0.78 for class 0 and 0.58 for class 1.

****

Conclusion

The best model among the five tested classifiers is Gradient Boosting, with an average accuracy of 0.869. It outperforms other models in terms of accuracy, and its F1-scores for both classes are among the highest. This suggests that Gradient Boosting is the most suitable model for this classification task.