MP3

November 1, 2025

1 AI211: Machine Exercise 3

Jhon Mark L. Acopiado
MEngg AI
University of the Philippines Diliman

2 Wine Quality Classification And Regression

Task: Download the Wine Quality dataset. For this problem, we will only use the Red Portuguese "Vinho Verde" wine data. This data set contains information about 1599 wine samples with 11 attributes coming from physicochemical tests.

3 1. Import Required Libraries

[104]:
```python
# Data manipulation and analysis
import pandas as pd
import numpy as np

# Data visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Machine learning
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score, f1_score, confusion_matrix
from sklearn.metrics import ConfusionMatrixDisplay

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


# Import additional libraries for hyperparameter tuning
from sklearn.metrics import precision_score, recall_score, classification_report
import optuna
```

```python
from optuna.samplers import TPESampler


# xgBoost
import xgboost as xgb

# Suppress Optuna logging
optuna.logging.set_verbosity(optuna.logging.WARNING)
```

4 2. Load Data set

[105]:
```python
# Sourcecode found @ https://archive.ics.uci.edu/dataset/186/wine+quality
from ucimlrepo import fetch_ucirepo

# fetch dataset
wine_quality = fetch_ucirepo(id=186)

# data (as pandas dataframes)
X = wine_quality.data.features
y = wine_quality.data.targets

# metadata
print(wine_quality.metadata)

# variable information
print(wine_quality.variables)
```


```
{'uci_id': 186, 'name': 'Wine Quality', 'repository_url':
'https://archive.ics.uci.edu/dataset/186/wine+quality', 'data_url':
'https://archive.ics.uci.edu/static/public/186/data.csv', 'abstract': 'Two
datasets are included, related to red and white vinho verde wine samples, from
the north of Portugal. The goal is to model wine quality based on
physicochemical tests (see [Cortez et al., 2009],
http://www3.dsi.uminho.pt/pcortez/wine/).', 'area': 'Business', 'tasks':
['Classification', 'Regression'], 'characteristics': ['Multivariate'],
'num_instances': 4898, 'num_features': 11, 'feature_types': ['Real'],
'demographics': [], 'target_col': ['quality'], 'index_col': None,
'has_missing_values': 'no', 'missing_values_symbol': None,
'year_of_dataset_creation': 2009, 'last_updated': 'Wed Nov 15 2023',
'dataset_doi': '10.24432/C56S3T', 'creators': ['Paulo Cortez', 'A. Cerdeira',
'F. Almeida', 'T. Matos', 'J. Reis'], 'intro_paper': {'ID': 252, 'type':
'NATIVE', 'title': 'Modeling wine preferences by data mining from
physicochemical properties', 'authors': 'P. Cortez, A. Cerdeira, Fernando
Almeida, Telmo Matos, J. Reis', 'venue': 'Decision Support Systems', 'year':
2009, 'journal': None, 'DOI': None, 'URL':
'https://www.semanticscholar.org/paper/Modeling-wine-preferences-by-data-mining-
from-Cortez-Cerdeira/bf15a0ccc14ac1deb5cea570c870389c16be019c', 'sha': None,
```

'corpus': None, 'arxiv': None, 'mag': None, 'acl': None, 'pmid': None, 'pmcid': None}, 'additional_info': {'summary': 'The two datasets are related to red and white variants of the Portuguese "Vinho Verde" wine. For more details, consult: http://www.vinhoverde.pt/en/ or the reference [Cortez et al., 2009]. Due to privacy and logistic issues, only physicochemical (inputs) and sensory (the output) variables are available (e.g. there is no data about grape types, wine brand, wine selling price, etc.).\n\nThese datasets can be viewed as classification or regression tasks. The classes are ordered and not balanced (e.g. there are many more normal wines than excellent or poor ones). Outlier detection algorithms could be used to detect the few excellent or poor wines. Also, we are not sure if all input variables are relevant. So it could be interesting to test feature selection methods.\n', 'purpose': None, 'funded_by': None, 'instances_represent': None, 'recommended_data_splits': None, 'sensitive_data': None, 'preprocessing_description': None, 'variable_info': 'For more information, read [Cortez et al., 2009].\r\nInput variables (based on physicochemical tests):\r\n   1 - fixed acidity\r\n   2 - volatile acidity\r\n   3 - citric acid\r\n   4 - residual sugar\r\n   5 - chlorides\r\n   6 - free sulfur dioxide\r\n   7 - total sulfur dioxide\r\n   8 - density\r\n   9 - pH\r\n   10 - sulphates\r\n   11 - alcohol\r\nOutput variable (based on sensory data): \r\n   12 - quality (score between 0 and 10)', 'citation': None}}

```
                    name     role         type demographic  \
0         fixed_acidity  Feature   Continuous        None
1      volatile_acidity  Feature   Continuous        None
2           citric_acid  Feature   Continuous        None
3        residual_sugar  Feature   Continuous        None
4             chlorides  Feature   Continuous        None
5    free_sulfur_dioxide  Feature   Continuous        None
6   total_sulfur_dioxide  Feature   Continuous        None
7               density  Feature   Continuous        None
8                    pH  Feature   Continuous        None
9             sulphates  Feature   Continuous        None
10              alcohol  Feature   Continuous        None
11              quality   Target      Integer        None
12                color    Other  Categorical        None

               description units missing_values
0                     None  None             no
1                     None  None             no
2                     None  None             no
3                     None  None             no
4                     None  None             no
5                     None  None             no
6                     None  None             no
7                     None  None             no
8                     None  None             no
9                     None  None             no
10                    None  None             no
11  score between 0 and 10  None             no
```

```
12            red or white  None            no
```

[165]:
```
# Prompt: Check if there are any missing values in the dataset
print(wine_quality.data["original"][wine_quality.data["original"]['color']==
 ↪'red'].isnull().sum())
```

```
fixed_acidity           0
volatile_acidity        0
citric_acid             0
residual_sugar          0
chlorides               0
free_sulfur_dioxide     0
total_sulfur_dioxide    0
density                 0
pH                      0
sulphates               0
alcohol                 0
quality                 0
color                   0
dtype: int64
```

5 3. Data Preparation

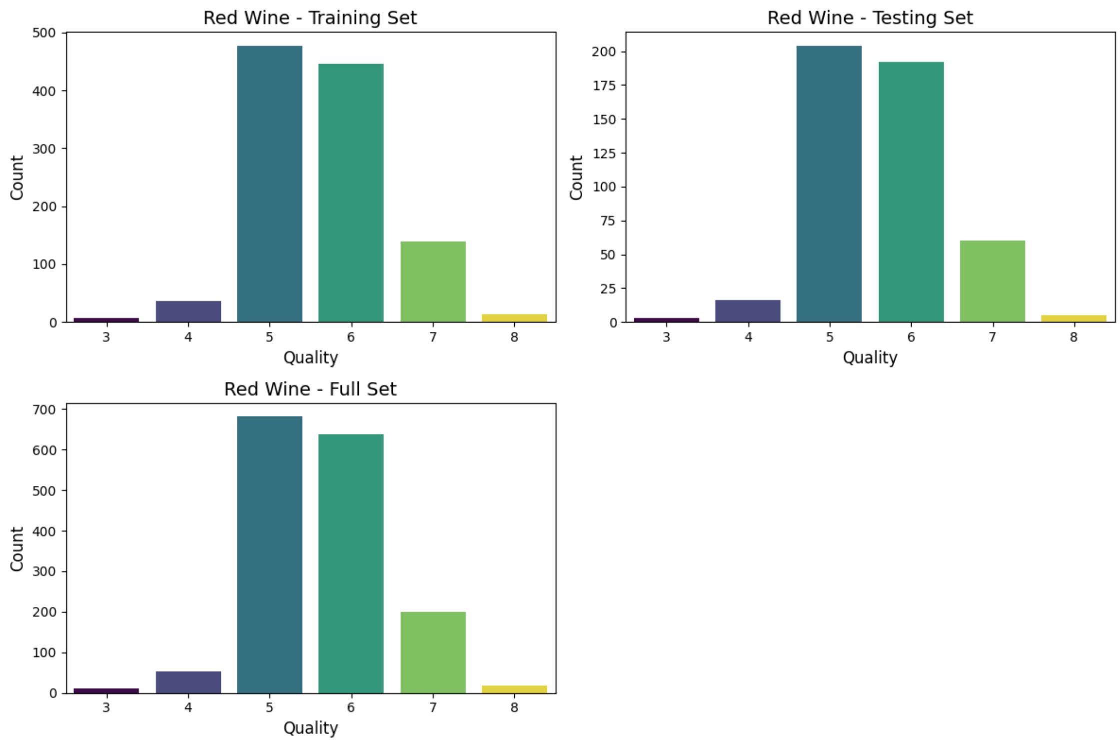
For this problem, we will only use the Red Portuguese "Vinho Verde" wine data.

Item a. Split the samples into 70% Training and 30% Testing data at random with stratification (stratify=y)

[ ]:
```
# Split the dataset into training and testing sets(30% test size, 70% train
 ↪size), stratified by the target variable

X_red =  wine_quality.data["original"][wine_quality.data["original"]['color']
 ↪== 'red'].drop(columns=['quality', 'color'])
y_red = wine_quality.data["original"][wine_quality.data["original"]['color'] ==
 ↪'red']['quality']
X_train_red, X_test_red, y_train_red, y_test_red = train_test_split(
    X_red, y_red, test_size=0.3, stratify=y_red, random_state=42
)

X_white =  wine_quality.data["original"][wine_quality.data["original"]['color']
 ↪== 'white'].drop(columns=['quality', 'color'])
y_white = wine_quality.data["original"][wine_quality.data["original"]['color']
 ↪== 'white']['quality']
X_train_white, X_test_white, y_train_white, y_test_white = train_test_split(
    X_white, y_white, test_size=0.3, stratify=y_white, random_state=42
)

X_train = X_train_red
```

```
y_train = y_train_red
X_test = X_test_red
y_test = y_test_red
```

[209]:
```
def plot_target_distribution(dataframes, titles):
    """
    Plots the distribution of the target variable for multiple dataframes.

    Parameters:
        dataframes (list of pd.DataFrame): List of dataframes to plot.
        titles (list of str): Corresponding titles for the dataframes.

    Returns:
        None
    """
    num_plots = len(dataframes)
    num_cols = 2
    num_rows = (num_plots + 1) // num_cols

    fig, axes = plt.subplots(num_rows, num_cols, figsize=(12, 4 * num_rows))
    axes = axes.flatten()

    for i, (df, title) in enumerate(zip(dataframes, titles)):
        ax = axes[i]
        sns.countplot(x='quality', data=df, ax=ax, palette='viridis',␣
 ↪hue='quality', legend=False)
        ax.set_title(title, fontsize=14)
        ax.set_xlabel('Quality', fontsize=12)
        ax.set_ylabel('Count', fontsize=12)

    # Hide unused subplots
    for j in range(i + 1, len(axes)):
        fig.delaxes(axes[j])

    plt.tight_layout()
    plt.show()

# Example usage
# Ensure the dataframes contain the 'quality' column
plot_target_distribution(
    [
        pd.DataFrame({'quality': y_train_red}),
        pd.DataFrame({'quality': y_test_red}),
        pd.DataFrame({'quality': y_red}),
    ],
    ["Red Wine - Training Set", "Red Wine - Testing Set", "Red Wine - Full Set"]
)
```



Starting here, all results be displayed are only Red Portuguese “Vinho Verde” wine data.

For this problem, we will only use the Red Portuguese “Vinho Verde” wine data.

6 4. Train SVM, MLP, Random Forest, and Gradient Boosting classifiers

Item a. Train SVM, MLP, Random Forest, and Gradient Boosting classifiers on the data. You are free to do your own hyper-parameter tuning.

6.1 4.0. Utility Functions

Item a. Report the accuracy, precision, recall, F1- score, and confusion matrix of all the models on test data.

[184]:
```python
# Prompt: Create a evaluation function that takes in a model, X_test, y_test and returns accuracy, f1-score, precision, recall
def evaluate_model(model, X_test, y_test, red_test=None, white_test=None):
    if isinstance(model, xgb.Booster):
        dtest = xgb.DMatrix(X_test)
        y_pred_probs = model.predict(dtest)
        if y_pred_probs.ndim > 1:  # multi-class
            y_pred = np.argmax(y_pred_probs, axis=1) + 3
        else:  # binary case
```

```
            y_pred = (y_pred_probs > 0.5).astype(int)
    else:
        # --- Sklearn-style models ---
        y_pred = model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred, average='weighted', zero_division=0)
    precision = precision_score(y_test, y_pred, average='weighted',␣
 ↪zero_division=0)
    recall = recall_score(y_test, y_pred, average='weighted', zero_division=0)
    results = {
        "accuracy": accuracy,
        "f1": f1,
        "precision": precision,
        "recall": recall,
    }
    if red_test is not None:
        x_red_test = red_test['x']
        y_red_test = red_test['y']
        evaluate_model_red = evaluate_model(model, x_red_test, y_red_test)
        results['red_wine'] = evaluate_model_red
    if white_test is not None:
        x_white_test = white_test['x']
        y_white_test = white_test['y']
        evaluate_model_white = evaluate_model(model, x_white_test, y_white_test)
        results['white_wine'] = evaluate_model_white

    return results

# prompt: Create a function that plots the confusion matrix given a data of␣
 ↪type list: { y_pred, y_test, model_name }
# function must receive a list of y_pred for multiple models and plot them in a␣
 ↪single figure with subplots
def plot_confusion_matrix(data: list[dict]):
    n_models = len(data['model_name'])
    fig, axes = plt.subplots(1, n_models, figsize=(n_models * 5, 5))
    if n_models == 1:
        axes = [axes]
    for ax, y_pred, y_test, model_name in zip(axes, data['y_pred'],␣
 ↪data['y_test'], data['model_name']):
        cm = confusion_matrix(y_test, y_pred, labels=[3,4,5,6,7,8,9])
        disp = ConfusionMatrixDisplay(confusion_matrix=cm,␣
 ↪display_labels=[3,4,5,6,7,8,9])
        disp.plot(ax=ax, cmap=plt.cm.Blues, colorbar=False)
        ax.set_title(f"{model_name}")

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

```
# prompt: create a function that stores model results in a dictionary, given␣
↪model name and evaluation metrics
def store_model_results(model_name, evaluation_metrics, results_dict):
    results_dict[model_name] = evaluation_metrics


# prompt: create a function that Prints model results, as a table,
# print result of white and red as different table

def print_evaluation_results(evaluation, title):
    print(f"\n{title}")
    print("Model results on Test Data (Red Wine).\n")

    # Combine overall, red wine, and white wine results into a single table
    results = pd.DataFrame({
        "Dataset": ["Red Wine",],
        "Accuracy": [
            # evaluation.get("accuracy"),
            evaluation.get("red_wine", {}).get("accuracy"),
            # evaluation.get("white_wine", {}).get("accuracy")
        ],
        "F1-Score": [
            # evaluation.get("f1"),
            evaluation.get("red_wine", {}).get("f1"),
            # evaluation.get("white_wine", {}).get("f1")
        ],
        "Precision": [
            # evaluation.get("precision"),
            evaluation.get("red_wine", {}).get("precision"),
            # evaluation.get("white_wine", {}).get("precision")
        ],
        "Recall": [
            # evaluation.get("recall"),
            evaluation.get("red_wine", {}).get("recall"),
            # evaluation.get("white_wine", {}).get("recall")
        ]
    })

    print("=" * 50)
    # Print the combined table
    print(results.to_string(index=False))
    print("=" * 50)
```

6.2 4.I. Build, Train, and Evaluate SVM classifier

Item a. Train SVM, MLP, Random Forest, and Gradient Boosting classifiers on the data. You are free to do your own hyper-parameter tuning.

[185]:
```
def objective_svm(trial):
    """Optuna objective function for RBF kernel SVM"""
    # Suggest hyperparameters
    kernel = trial.suggest_categorical('kernel', ['linear', 'poly', 'rbf',␣
 ↪'sigmoid'])
    C = trial.suggest_float('C', 0.1, 100.0, log=True)
    gamma = trial.suggest_float('gamma', 0.001, 1.0, log=True)


    # Create pipeline
    pipeline = Pipeline([
        ('scaler', StandardScaler()),
        ('svm', SVC(kernel=kernel, C=C, gamma=gamma,  random_state=42))
    ])

    # Train and evaluate
    pipeline.fit(X_train, y_train)
    y_val_pred = pipeline.predict(X_test)

    # Return F1-macro score (maximize)
    return f1_score(y_test, y_val_pred, average='macro')

# Create study and optimize
print("Optimizing SVM with Optuna...")
study_svm = optuna.create_study(
    direction='maximize',
    sampler=TPESampler(seed=42)
)
study_svm.optimize(objective_svm, n_trials=50, show_progress_bar=True)

# Get best parameters
best_params_svm = study_svm.best_params
print(f"\nBest parameters for RBF SVM:")
print(f"  Kernel: {best_params_svm['kernel']}")
print(f"  C: {best_params_svm['C']:.4f}")
print(f"  gamma: {best_params_svm['gamma']:.4f}")
```

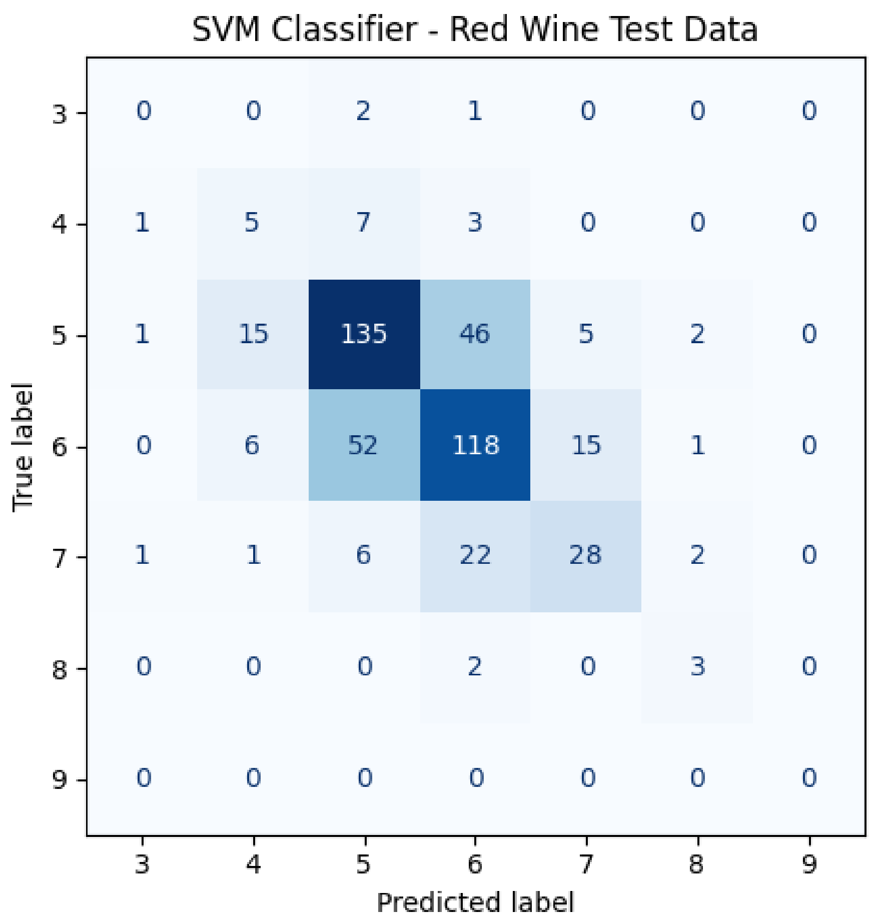
```
Optimizing SVM with Optuna…

  0%|          | 0/50 [00:00<?, ?it/s]


Best parameters for RBF SVM:
  Kernel: poly
```

```
C: 1.3990
gamma: 0.7374
```

Item a. Report the accuracy, precision, recall, F1-score, and confusion matrix of all the models on test data.

[186]:
```python
# Train model with best parameters
optimized_svm_classifier = Pipeline([
    ('scaler', StandardScaler()),
    ('svm', SVC(
        kernel=best_params_svm['kernel'],
        C=best_params_svm['C'],
        gamma=best_params_svm['gamma'],
        coef0=best_params_svm.get('coef0', 0.0),
        degree=best_params_svm.get('degree', 3),
        random_state=42
    ))
])

# Fit the optimized model
optimized_svm_classifier.fit(X_train, y_train)
# Evaluate
optimized_svm_classifier_results = evaluate_model(
    optimized_svm_classifier,
    X_test, y_test, red_test={"x": X_test_red, "y": y_test_red}, white_test={
        "x": X_test_white, "y": y_test_white
    }
)
print_evaluation_results(optimized_svm_classifier_results, title="SVM␣
 ↪Classifier")
plot_confusion_matrix({
    'y_pred': [optimized_svm_classifier.predict(X_test_red)],
    'y_test': [y_test_red],
    'model_name': ["SVM Classifier - Red Wine Test Data"]
})
```

```
SVM Classifier
Model results on Test Data (Red Wine).

==================================================
 Dataset  Accuracy  F1-Score  Precision   Recall
Red Wine  0.602083  0.605843   0.612864 0.602083
==================================================
```



6.3 4.II. Build, Train, and Evaluate MLP classifier

Item a. Train SVM, MLP, Random Forest, and Gradient Boosting classifiers on the data. You are free to do your own hyper-parameter tuning.

[187]:
```
# Sourcecode adapted from AI221 repo:
import numpy as np
from time import time
import matplotlib.pyplot as plt
from scipy.stats import randint, loguniform
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import RandomizedSearchCV, RepeatedStratifiedKFold
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay,␣
↪classification_report

model_params = {'alpha': loguniform(1e-4, 10),
                'solver': ['sgd','adam'],
                'hidden_layer_sizes': randint(4,16),
```

```
                'activation': ['identity','logistic','tanh','relu']}

cv = RepeatedStratifiedKFold(n_splits=5, n_repeats=3)

n_iter = 50
optimized_mlp_classifier = RandomizedSearchCV(MLPClassifier(max_iter=5000),␣
 ↪cv=cv,
                                   param_distributions=model_params,
                                   n_iter=n_iter, verbose=1)

start = time()
optimized_mlp_classifier.fit(X_train, y_train)
print("RandomizedSearchCV took %.2f seconds for %d candidates parameter␣
 ↪settings."
        % ((time() - start), n_iter))

# Print best parameters after tuning
print(optimized_mlp_classifier.best_params_)

# Print how our model looks after hyper-parameter tuning
print(optimized_mlp_classifier.best_estimator_)
```

```
Fitting 15 folds for each of 50 candidates, totalling 750 fits
RandomizedSearchCV took 317.27 seconds for 50 candidates parameter settings.
{'activation': 'relu', 'alpha': np.float64(0.00014545226495568257),
'hidden_layer_sizes': 13, 'solver': 'adam'}
MLPClassifier(alpha=np.float64(0.00014545226495568257), hidden_layer_sizes=13,
              max_iter=5000)
```

Item a. Report the accuracy, precision, recall, F1-score, and confusion matrix of all the models on test data.

[188]:
```
y = optimized_mlp_classifier.predict(X_test)

optimized_mlp_classifier_results = evaluate_model(
    optimized_mlp_classifier,
    X_test, y_test, red_test={"x": X_test_red, "y": y_test_red}, white_test={
        "x": X_test_white, "y": y_test_white
    }
)

print_evaluation_results(optimized_mlp_classifier_results, title="MLP␣
 ↪Classifier")
plot_confusion_matrix({
    'y_pred': [ optimized_mlp_classifier.predict(X_test_red),],
    'y_test': [y_test_red],
    'model_name': ["MLP Classifier - Red Wine Test Data"]
})
```

```
MLP Classifier
Model results on Test Data (Red Wine).

==================================================
 Dataset  Accuracy  F1-Score  Precision   Recall
Red Wine  0.577083  0.535872   0.539063 0.577083
==================================================
```

6.4 4.III. Build, Train, and Evaluate Random Forest classifier

Item a. Train SVM, MLP, Random Forest, and Gradient Boosting classifiers on the data. You are free to do your own hyper-parameter tuning.

[189]:
```
from sklearn.ensemble import RandomForestClassifier


def objective_rf_classifier(trial):
```

```
    """Optuna objective function for RBF kernel SVM"""
    # Suggest hyperparameters
    n_estimators = trial.suggest_int("n_estimators", 100, 1000, step=100)
    max_depth = trial.suggest_int("max_depth", 5, 50, step=5)


    # Create pipeline
    pipeline =  Pipeline([
                ("classifier",␣
 ↪RandomForestClassifier(n_estimators=n_estimators,max_depth=max_depth,random_state=42))])

    # Train and evaluate
    pipeline.fit(X_train, y_train)
    y_val_pred = pipeline.predict(X_test)

    # Return F1-macro score (maximize)
    return f1_score(y_test, y_val_pred, average='macro')


study_rf_classifier = optuna.create_study(
    direction='maximize',
    sampler=TPESampler(seed=42)
)
study_rf_classifier.optimize(objective_rf_classifier, n_trials=50,␣
 ↪show_progress_bar=True)

# Get best parameters
best_params_rf_classifier = study_rf_classifier.best_params
print(f"\nBest parameters for Random Forest Classifier:")
print(f"  n_estimators: {best_params_rf_classifier['n_estimators']}")
print(f"  max_depth: {best_params_rf_classifier['max_depth']}")
```

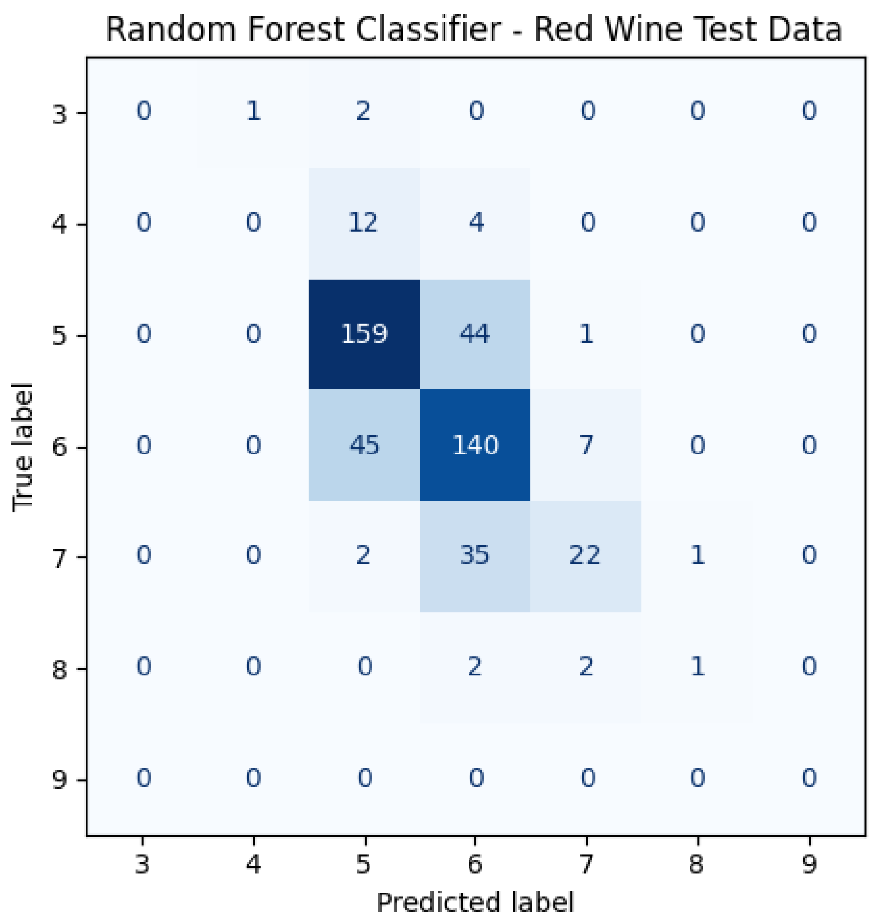
```
  0%|          | 0/50 [00:00<?, ?it/s]


Best parameters for Random Forest Classifier:
  n_estimators: 400
  max_depth: 15
```

Item a. Report the accuracy, precision, recall, F1-score, and confusion matrix of all the models on test data.

[190]:
```
optimized_random_forest_classifier = Pipeline([
                ("classifier",␣
 ↪RandomForestClassifier(n_estimators=best_params_rf_classifier["n_estimators"],max_depth=bes

optimized_random_forest_classifier.fit(X_train, y_train)
```

```
optimized_random_forest_classifier_results =␣
↪evaluate_model(optimized_random_forest_classifier, X_test, y_test, red_test={
    "x": X_test_red, "y": y_test_red
}, white_test={
    "x": X_test_white, "y": y_test_white
})

print_evaluation_results(optimized_random_forest_classifier_results, "Random␣
↪Forest Classifier")

plot_confusion_matrix({
    'y_pred': [ optimized_random_forest_classifier.predict(X_test_red)],
    'y_test': [y_test_red],
    'model_name': ["Random Forest Classifier - Red Wine Test Data"]
})
```

```
Random Forest Classifier
Model results on Test Data (Red Wine).

==================================================
 Dataset   Accuracy  F1-Score  Precision   Recall
Red Wine   0.670833  0.650094   0.647194 0.670833
==================================================
```



6.5 4.IV. Build, Train, and Evaluate Gradient Boosting classifier

Item a. Train SVM, MLP, Random Forest, and Gradient Boosting classifiers on the data. You are free to do your own hyper-parameter tuning.

[191]:
```
import xgboost as xgb
from sklearn.preprocessing import LabelEncoder

label_encoder = LabelEncoder()
y_train_encoded = label_encoder.fit_transform(y_train)
y_test_encoded = label_encoder.transform(y_test)

dtrain_clf = xgb.DMatrix(X_train, label=y_train_encoded,␣
 ↪enable_categorical=True)
dtest_clf = xgb.DMatrix(X_test, label=y_test_encoded, enable_categorical=True)

params = {
    "objective": "multi:softprob",
```

```
    "num_class": len(label_encoder.classes_),
    "eval_metric": "mlogloss"
}

import optuna
from optuna.samplers import TPESampler
from sklearn.metrics import f1_score
import xgboost as xgb
import numpy as np

def objective_xgb_classifier(trial):
    # Suggest hyperparameters
    param = {
        "objective": "multi:softprob",
        "num_class": len(label_encoder.classes_),
        "eval_metric": "mlogloss",
        "max_depth": trial.suggest_int("max_depth", 3, 15),
        "learning_rate": trial.suggest_float("learning_rate", 0.01, 0.3,␣
↪log=True),
        "subsample": trial.suggest_float("subsample", 0.5, 1.0),
        "colsample_bytree": trial.suggest_float("colsample_bytree", 0.5, 1.0),
        "gamma": trial.suggest_float("gamma", 0, 5),
        "min_child_weight": trial.suggest_int("min_child_weight", 1, 10),
        "lambda": trial.suggest_float("lambda", 1e-3, 10.0, log=True),
        "alpha": trial.suggest_float("alpha", 1e-3, 10.0, log=True),
        "verbosity": 0,
    }

    # Convert to DMatrix
    dtrain = xgb.DMatrix(X_train, label=y_train_encoded)
    dtest = xgb.DMatrix(X_test, label=y_test_encoded)

    # Train model
    model = xgb.train(
        params=param,
        dtrain=dtrain,
        num_boost_round=trial.suggest_int("num_boost_round", 100, 1000),
        evals=[(dtest, "test")],
        verbose_eval=False,
    )

    # Predict and evaluate
    y_pred = model.predict(dtest)
    y_pred_classes = np.argmax(y_pred, axis=1)
    score = f1_score(y_test_encoded, y_pred_classes, average="weighted")

    # We want to maximize F1-score
```

```
    return score


# Create and run Optuna study
study_xgb_classifier = optuna.create_study(direction="maximize",␣
 ↪sampler=TPESampler(seed=42))
study_xgb_classifier.optimize(objective_xgb_classifier, n_trials=50,␣
 ↪show_progress_bar=True)

# Show best parameters
best_params_xgb = study_xgb_classifier.best_params
print("\nBest parameters for XGBoost Classifier:")
for k, v in best_params_xgb.items():
    print(f"  {k}: {v}")
```

```
  0%|          | 0/50 [00:00<?, ?it/s]


Best parameters for XGBoost Classifier:
  max_depth: 12
  learning_rate: 0.039095548587497045
  subsample: 0.5071019465088584
  colsample_bytree: 0.8301858216439442
  gamma: 0.5233757677409074
  min_child_weight: 2
  lambda: 0.00210098563858066
  alpha: 0.010333819037730327
  num_boost_round: 226
```

Item a. Report the accuracy, precision, recall, F1-score, and confusion matrix of all the models on test data.

[192]:
```
best_params_xgb.update({
    "objective": "multi:softprob",
    "num_class": len(label_encoder.classes_),
    "eval_metric": "mlogloss"
})

optimized_xgb_classifier = xgb.train(
    params=best_params_xgb,
    dtrain=dtrain_clf,
    num_boost_round=best_params_xgb["num_boost_round"],
    evals=[(dtrain_clf, "train"), (dtest_clf, "test")],
    verbose_eval=False,
)
```

```
optimized_xgb_classifier_results = evaluate_model(optimized_xgb_classifier,␣
↪X_test, y_test, red_test={"x": X_test_red, "y": y_test_red}, white_test={"x":
↪ X_test_white, "y": y_test_white})

print_evaluation_results(optimized_xgb_classifier_results, "XGBoost Classifier")

# Note, the lowest rating is 3, so we add 3; such way 0 becomes 3, and 7 become␣
↪10
plot_confusion_matrix({
    'y_pred': [np.argmax(optimized_xgb_classifier.predict(xgb.
↪DMatrix(X_test_red)), axis=1) + 3],
    'y_test': [y_test_red],
    'model_name': ["XGBoost Classifier - Red Wine Test Data"]
})
```

```
XGBoost Classifier
Model results on Test Data (Red Wine).

==================================================
 Dataset  Accuracy  F1-Score  Precision   Recall
Red Wine  0.672917    0.6561   0.653972 0.672917
==================================================
```

7 5. Train SVR, MLP, Random Forest, and Gradient Boosting regressors

Item b. Use SVR, MLP, Random Forest, and Gradient Boosting regressors

7.1 5.0. Utility functions

Item b. Item b. Report the MSE, R2, and MAD (mean absolute) on test data.

[193]:
```
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
import matplotlib.pyplot as plt
import seaborn as sns
# Prompt: Make simlar functions found in classification, but for regression␣
↪models use regression metrics: mse, r2, mae,
# and predict actual vs predicted plots
# ---------------------------------------------------------
# Evaluate a regression model (returns MSE, R2, MAE)
```

```python
# -------------------------------------------------------
def evaluate_regression_model(model, X_test, y_test, red_test=None,␣
↪white_test=None):

    try:
        # Case 1: xgboost.train model (booster)
        if isinstance(model, xgb.Booster):
            dtest = xgb.DMatrix(X_test)
            y_pred = model.predict(dtest)
        else:
            # Case 2: sklearn-compatible models (pipeline, MLP, SVR, RF,␣
↪XGBRegressor)
            y_pred = model.predict(X_test)
    except Exception as e:
        raise RuntimeError(f"Prediction failed: {e}")

    mse = mean_squared_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)
    mae = mean_absolute_error(y_test, y_pred)
    mad = np.mean(np.abs(y_test - np.mean(y_test)))
    results = {
        "mse": mse,
        "r2": r2,
        "mae": mae,
        "mad": mad,
    }

    # Optional: evaluate on red and white test sets if provided
    if red_test is not None:
        x_red_test = red_test['x']
        y_red_test = red_test['y']
        evaluate_red = evaluate_regression_model(model, x_red_test, y_red_test)
        results['red_wine'] = evaluate_red

    if white_test is not None:
        x_white_test = white_test['x']
        y_white_test = white_test['y']
        evaluate_white = evaluate_regression_model(model, x_white_test,␣
↪y_white_test)
        results['white_wine'] = evaluate_white

    return results


def plot_actual_vs_predicted(data):
    """
    Plots Actual vs Predicted scatter plots for multiple regression models.
```

```
    Parameters
    ----------
    data : dict
        Must contain:
            - 'y_pred': list of model predictions
            - 'y_test': list of corresponding true values
            - 'model_name': list of model names (same length as above)
        Example:
            data = {
                'y_pred': [y_pred_lr, y_pred_rf],
                'y_test': [y_test, y_test],
                'model_name': ['Linear Regression', 'Random Forest']
            }
    """
    n_models = len(data['model_name'])
    fig, axes = plt.subplots(1, n_models, figsize=(n_models * 6, 5))

    if n_models == 1:
        axes = [axes]

    for i, (y_pred, y_test, model_name) in enumerate(
        zip(data['y_pred'], data['y_test'], data['model_name'])
    ):
        sns.scatterplot(x=y_test, y=y_pred, ax=axes[i])
        # Plot ideal line y = x
        min_val = min(min(y_test), min(y_pred))
        max_val = max(max(y_test), max(y_pred))
        axes[i].plot([min_val, max_val], [min_val, max_val], 'r--', lw=2)
        axes[i].set_title(f"Actual vs Predicted - {model_name}")
        axes[i].set_xlabel("Actual Values")
        axes[i].set_ylabel("Predicted Values")
        axes[i].grid(True, linestyle='--', alpha=0.5)

    plt.tight_layout()
    plt.show()



# ----------------------------------------------------------
# Print evaluation results
# ----------------------------------------------------------
def print_regression_results(evaluation, title):
    print(f"\n{title}")
    print("Model results on Test Data (Overall, Red Wine, White Wine).\n")

    # Combine overall, red wine, and white wine results into a single table
```

```
    results = pd.DataFrame({
        "Dataset": [ "Red Wine", ],
        "MSE": [
            # evaluation.get("mse"),
            evaluation.get("red_wine", {}).get("mse"),
            # evaluation.get("white_wine", {}).get("mse")
        ],
        "R²": [
            # evaluation.get("r2"),
            evaluation.get("red_wine", {}).get("r2"),
            # evaluation.get("white_wine", {}).get("r2")
        ],
        "MAD": [
            # evaluation.get("mad"),
            evaluation.get("red_wine", {}).get("mad"),
            # evaluation.get("white_wine", {}).get("mad")
        ],
    })

    print("=" * 50)
    # Print the combined table
    print(results.to_string(index=False))
    print("=" * 50)
```

7.2 5.I. Build, Train, and Evaluate SVM Regressor

Item b. use SVR, MLP, Random Forest, and Gradient Boosting regressors.

[194]:
```
import optuna
from optuna.samplers import TPESampler
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
import numpy as np

def objective_svr(trial):
    """Optuna objective function for SVR (Support Vector Regression)"""

    # Suggest hyperparameters
    kernel = trial.suggest_categorical('kernel', ['linear', 'poly', 'rbf',␣
 ↪'sigmoid'])
    C = trial.suggest_float('C', 0.1, 100.0, log=True)
    gamma = trial.suggest_float('gamma', 0.001, 1.0, log=True)
    epsilon = trial.suggest_float('epsilon', 0.001, 1.0, log=True)

    # Create pipeline
```

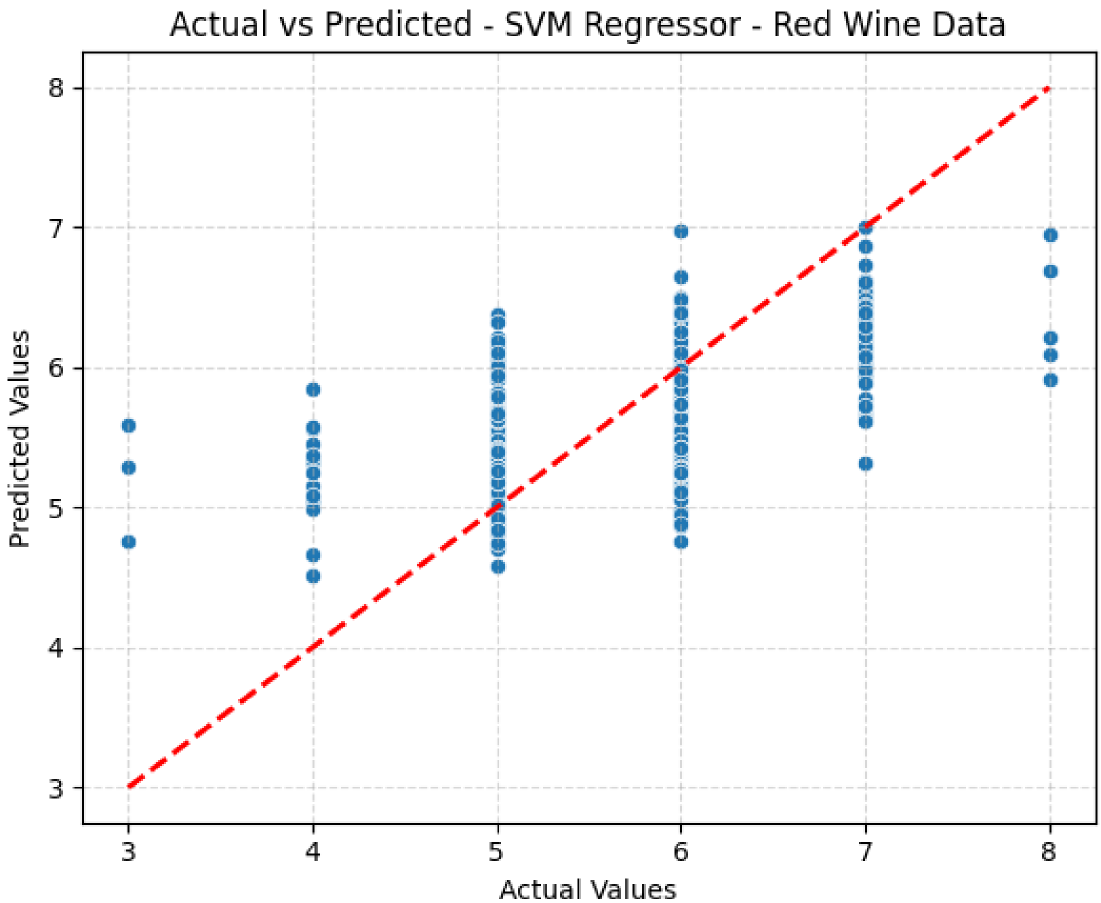
```python
    pipeline = Pipeline([
        ('scaler', StandardScaler()),
        ('svr', SVR(kernel=kernel, C=C, gamma=gamma, epsilon=epsilon))
    ])

    # Train and predict
    pipeline.fit(X_train, y_train)
    y_pred = pipeline.predict(X_test)

    # Compute evaluation metrics
    mse = mean_squared_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)
    mad = mean_absolute_error(y_test, y_pred)

    # Log metrics to Optuna
    trial.set_user_attr("mse", mse)
    trial.set_user_attr("r2", r2)
    trial.set_user_attr("mad", mad)

    # We want to maximize R² (goodness of fit)
    return r2


# --- Run Optuna Study ---
print("Optimizing SVR with Optuna...")
study_svr = optuna.create_study(
    direction='maximize',
    sampler=TPESampler(seed=42)
)
study_svr.optimize(objective_svr, n_trials=50, show_progress_bar=True)

# --- Get Best Parameters ---
best_params_svr = study_svr.best_params
print("\nBest parameters for SVR:")
print(f"  Kernel: {best_params_svr['kernel']}")
print(f"  C: {best_params_svr['C']:.4f}")
print(f"  Gamma: {best_params_svr['gamma']:.4f}")
print(f"  Epsilon: {best_params_svr['epsilon']:.4f}")
```

```
Optimizing SVR with Optuna…

  0%|          | 0/50 [00:00<?, ?it/s]


Best parameters for SVR:
  Kernel: linear
  C: 41.2513
  Gamma: 0.0626
  Epsilon: 0.0935
```

Item b. Report the MSE, R2, and MAD (mean absolute deviation) on the test data.

[195]:
```
# --- Evaluate the final best model ---
optimized_svm_regressor = Pipeline([
    ('scaler', StandardScaler()),
    ('svr', SVR(**best_params_svr))
])

optimized_svm_regressor.fit(X_train, y_train)

optimized_svm_regressor_results = evaluate_regression_model(optimized_svm_regressor, X_test, y_test,
                                                            red_test={"x": X_test_red, "y": y_test_red}, white_test={"x": X_test_white, "y": y_test_white})

print_regression_results(optimized_svm_regressor_results, "SVM Regressor")
plot_actual_vs_predicted({
    "y_pred" : [optimized_svm_regressor.predict(X_test_red)],
    "y_test": [y_test_red],
    "model_name": ["SVM Regressor - Red Wine Data"]
})
```

```
SVM Regressor
Model results on Test Data (Overall, Red Wine, White Wine).

==================================================
 Dataset       MSE       R²       MAD
Red Wine 0.403677 0.377358 0.682075
==================================================
```



7.3 5.II. Build, Train, and Evaluate MLP Regressor

Item b. use SVR, MLP, Random Forest, and Gradient Boosting regressors.

[196]:
```
from time import time
from scipy.stats import randint, loguniform
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import RandomizedSearchCV, RepeatedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline

# --- Create pipeline with imputation and scaling ---
mlp_pipeline = Pipeline([
    ("imputer", SimpleImputer(strategy="mean")),       # Handles missing values
    ("scaler", StandardScaler()),                      # Normalizes features
    ("mlp", MLPRegressor(max_iter=5000, random_state=42))
])
```

```
# --- Define hyperparameter search space ---
model_params = {
    "mlp__alpha": loguniform(1e-4, 10),
    "mlp__solver": ['adam', 'sgd'],
    "mlp__hidden_layer_sizes": randint(4, 16),
    "mlp__activation": ['identity', 'logistic', 'tanh', 'relu'],
    "mlp__learning_rate_init": loguniform(1e-5, 1e-2)  # add stable learning␣
 ↪rate search
}

# --- Repeated K-Fold for regression ---
cv = RepeatedKFold(n_splits=5, n_repeats=3, random_state=42)

# --- RandomizedSearchCV for MLPRegressor ---
n_iter = 50
optimized_mlp_regressor = RandomizedSearchCV(
    mlp_pipeline,
    param_distributions=model_params,
    n_iter=n_iter,
    cv=cv,
    verbose=1,
    n_jobs=1,
    scoring='r2'  # R² is standard for regression
)

# --- Train the model ---
start = time()
optimized_mlp_regressor.fit(X_train, y_train)
print("RandomizedSearchCV took %.2f seconds for %d candidates parameter␣
 ↪settings."
      % ((time() - start), n_iter))

# --- Best parameters ---
print("\nBest Parameters:")
print(optimized_mlp_regressor.best_params_)

# --- Optional: Evaluate on test set ---
y_pred = optimized_mlp_regressor.predict(X_test)
from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error

print("\nEvaluation Metrics on Test Set:")
print(f"R²: {r2_score(y_test, y_pred):.4f}")
print(f"MAE: {mean_absolute_error(y_test, y_pred):.4f}")
print(f"MSE: {mean_squared_error(y_test, y_pred):.4f}")
```

```
Fitting 15 folds for each of 50 candidates, totalling 750 fits
```

```
c:\Users\jhon\AppData\Local\Programs\Python\Python313\Lib\site-
packages\sklearn\neural_network\_multilayer_perceptron.py:781:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (5000) reached and
the optimization hasn't converged yet.
  warnings.warn(
c:\Users\jhon\AppData\Local\Programs\Python\Python313\Lib\site-
packages\sklearn\neural_network\_multilayer_perceptron.py:781:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (5000) reached and
the optimization hasn't converged yet.
  warnings.warn(
c:\Users\jhon\AppData\Local\Programs\Python\Python313\Lib\site-
packages\sklearn\neural_network\_multilayer_perceptron.py:781:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (5000) reached and
the optimization hasn't converged yet.
  warnings.warn(
c:\Users\jhon\AppData\Local\Programs\Python\Python313\Lib\site-
packages\sklearn\neural_network\_multilayer_perceptron.py:781:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (5000) reached and
the optimization hasn't converged yet.
  warnings.warn(
c:\Users\jhon\AppData\Local\Programs\Python\Python313\Lib\site-
packages\sklearn\neural_network\_multilayer_perceptron.py:781:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (5000) reached and
the optimization hasn't converged yet.
  warnings.warn(
c:\Users\jhon\AppData\Local\Programs\Python\Python313\Lib\site-
packages\sklearn\neural_network\_multilayer_perceptron.py:781:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (5000) reached and
the optimization hasn't converged yet.
  warnings.warn(
c:\Users\jhon\AppData\Local\Programs\Python\Python313\Lib\site-
packages\sklearn\neural_network\_multilayer_perceptron.py:781:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (5000) reached and
the optimization hasn't converged yet.
  warnings.warn(
c:\Users\jhon\AppData\Local\Programs\Python\Python313\Lib\site-
packages\sklearn\neural_network\_multilayer_perceptron.py:781:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (5000) reached and
the optimization hasn't converged yet.
  warnings.warn(
c:\Users\jhon\AppData\Local\Programs\Python\Python313\Lib\site-
packages\sklearn\neural_network\_multilayer_perceptron.py:781:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (5000) reached and
the optimization hasn't converged yet.
  warnings.warn(
c:\Users\jhon\AppData\Local\Programs\Python\Python313\Lib\site-
packages\sklearn\neural_network\_multilayer_perceptron.py:781:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (5000) reached and
```

```
the optimization hasn't converged yet.
  warnings.warn(
c:\Users\jhon\AppData\Local\Programs\Python\Python313\Lib\site-
packages\sklearn\neural_network\_multilayer_perceptron.py:781:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (5000) reached and
the optimization hasn't converged yet.
  warnings.warn(
c:\Users\jhon\AppData\Local\Programs\Python\Python313\Lib\site-
packages\sklearn\neural_network\_multilayer_perceptron.py:781:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (5000) reached and
the optimization hasn't converged yet.
  warnings.warn(
c:\Users\jhon\AppData\Local\Programs\Python\Python313\Lib\site-
packages\sklearn\neural_network\_multilayer_perceptron.py:781:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (5000) reached and
the optimization hasn't converged yet.
  warnings.warn(
c:\Users\jhon\AppData\Local\Programs\Python\Python313\Lib\site-
packages\sklearn\neural_network\_multilayer_perceptron.py:781:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (5000) reached and
the optimization hasn't converged yet.
  warnings.warn(
c:\Users\jhon\AppData\Local\Programs\Python\Python313\Lib\site-
packages\sklearn\neural_network\_multilayer_perceptron.py:781:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (5000) reached and
the optimization hasn't converged yet.
  warnings.warn(
```

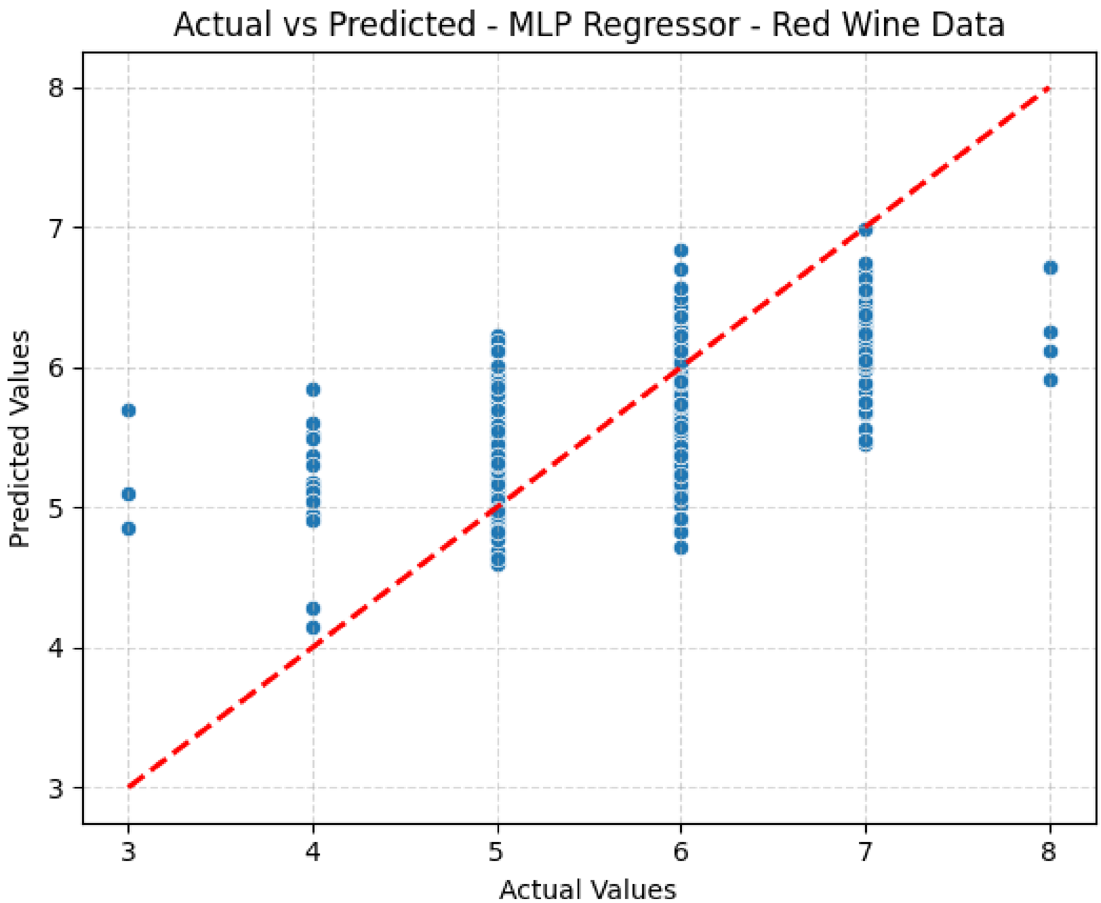
```
RandomizedSearchCV took 471.66 seconds for 50 candidates parameter settings.

Best Parameters:
{'mlp__activation': 'tanh', 'mlp__alpha': np.float64(3.475851037776619),
'mlp__hidden_layer_sizes': 14, 'mlp__learning_rate_init':
np.float64(0.003478147685194609), 'mlp__solver': 'adam'}

Evaluation Metrics on Test Set:
R²: 0.3883
MAE: 0.4888
MSE: 0.3966
```

Item b. Report the MSE, R2, and MAD (mean absolute deviation) on the test data.

[197]:
```
optimized_mlp_regressor_results = evaluate_regression_model(optimized_mlp_regressor, X_test, y_test,
                                                            red_test={
                                                                "x":X_test_red,
                                                                "y": y_test_red,
                                                            },
```

```
                                                                     white_test={
                                                                         "x":␣
 ↪X_test_white,
                                                                         "y":␣
 ↪y_test_white
                                                                     })

print_regression_results(optimized_mlp_regressor_results, "MLP Reggressor")

plot_actual_vs_predicted({
    "y_pred" : [optimized_mlp_regressor.predict(X_test_red)],
    "y_test": [y_test_red],
    "model_name": ["MLP Regressor - Red Wine Data"]
})
```

```
MLP Reggressor
Model results on Test Data (Overall, Red Wine, White Wine).

==================================================
 Dataset      MSE       R²       MAD
Red Wine 0.396614 0.388252 0.682075
==================================================
```



7.4 5.III. Build, Train, and Evaluate Random Forest Regressor

Item b. use SVR, MLP, Random Forest, and Gradient Boosting regressors.

[198]:
```
from sklearn.ensemble import RandomForestRegressor

def objective_rf_regressor(trial):
    """Optuna objective function for Random Forest Regressor"""

    # Suggest hyperparameters
    n_estimators = trial.suggest_int("n_estimators", 100, 1000, step=100)
    max_depth = trial.suggest_int("max_depth", 5, 50, step=5)
    min_samples_split = trial.suggest_int("min_samples_split", 2, 10)
    min_samples_leaf = trial.suggest_int("min_samples_leaf", 1, 5)

    # Create pipeline
    pipeline = Pipeline([
        ("regressor", RandomForestRegressor(
```

```
            n_estimators=n_estimators,
            max_depth=max_depth,
            min_samples_split=min_samples_split,
            min_samples_leaf=min_samples_leaf,
            random_state=42,
            n_jobs=-1
        ))
    ])

    # Train and predict
    pipeline.fit(X_train, y_train)
    y_pred = pipeline.predict(X_test)

    # Compute regression metrics
    mse = mean_squared_error(y_test, y_pred)
    mae = mean_absolute_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)

    # Log metrics
    trial.set_user_attr("mse", mse)
    trial.set_user_attr("mae", mae)
    trial.set_user_attr("r2", r2)

    # We want to maximize R²
    return r2


# --- Run Optuna Study ---
print("Optimizing Random Forest Regressor with Optuna...")
study_rf_regressor = optuna.create_study(
    direction='maximize',
    sampler=TPESampler(seed=42)
)
study_rf_regressor.optimize(objective_rf_regressor, n_trials=50,
 ↪show_progress_bar=True)

# --- Get Best Parameters ---
best_params_rf_regressor = study_rf_regressor.best_params
print("\nBest Parameters for Random Forest Regressor:")
for k, v in best_params_rf_regressor.items():
    print(f"  {k}: {v}")
```

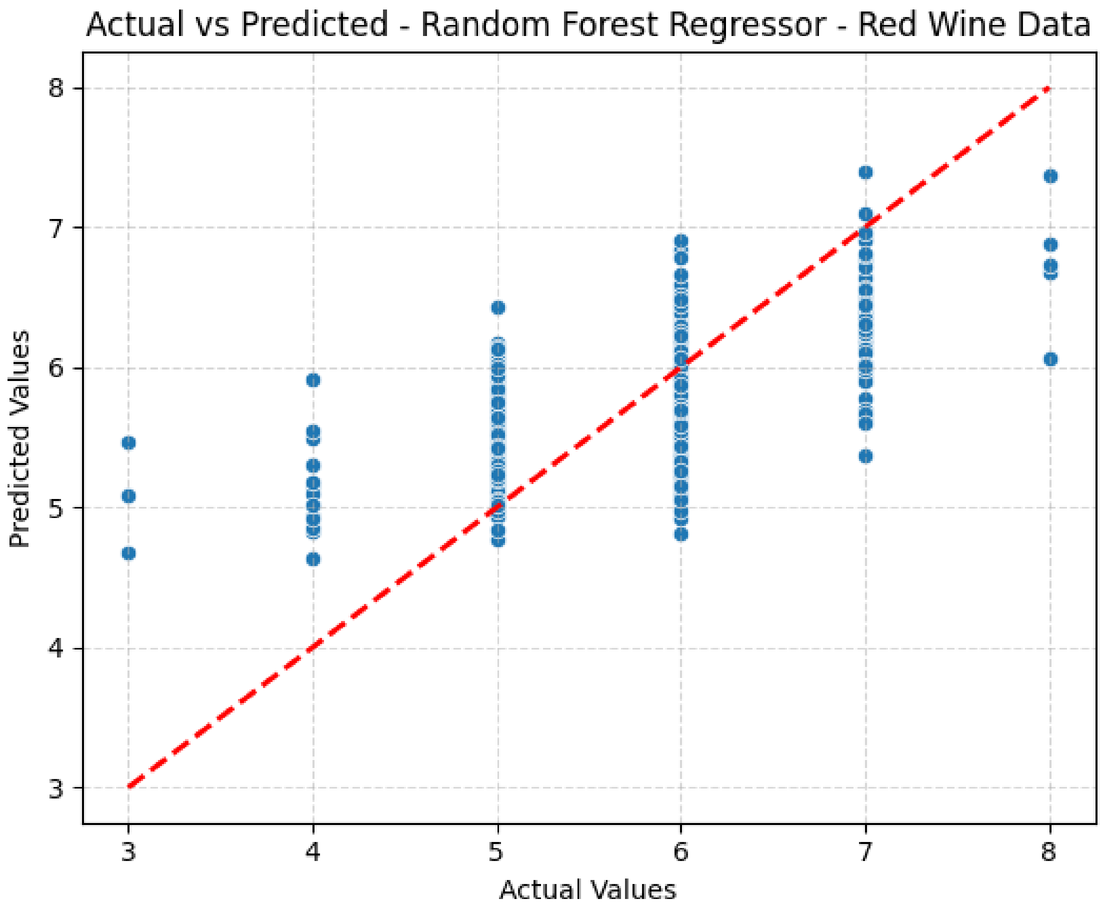
```
Optimizing Random Forest Regressor with Optuna…

  0%|          | 0/50 [00:00<?, ?it/s]


Best Parameters for Random Forest Regressor:
  n_estimators: 500
```

```
max_depth: 20
min_samples_split: 2
min_samples_leaf: 1
```

Item b. Report the MSE, R2, and MAD (mean absolute deviation) on the test data.

[199]:
```
optimized_random_forest_regressor = RandomForestRegressor(
    **best_params_rf_regressor, random_state=42, n_jobs=-1
)
optimized_random_forest_regressor.fit(X_train, y_train)

optimized_random_forest_regressor_results =␣
 ↪evaluate_regression_model(optimized_random_forest_regressor, X_test, y_test,
                                                                        red_test={
                                                                            "x":␣
 ↪X_test_red,
                                                                            "y":␣
 ↪y_test_red
                                                                        },
                                                                       ␣
 ↪white_test={
                                                                            "x":␣
 ↪X_test_white,
                                                                            "y":␣
 ↪y_test_white
                                                                        })

print_regression_results(optimized_random_forest_regressor_results, "Random␣
 ↪Forest Regressor")

plot_actual_vs_predicted({
    "y_pred" : [optimized_random_forest_regressor.predict(X_test_red)],
    "y_test": [y_test_red],
    "model_name": ["Random Forest Regressor - Red Wine Data"]
})
```

```
Random Forest Regressor
Model results on Test Data (Overall, Red Wine, White Wine).

==================================================
 Dataset      MSE       R²       MAD
Red Wine 0.328627 0.493117 0.682075
==================================================
```



7.5 5.IV. Build, Train, and Evaluate Gradient Boosting Regressor

Item b. use SVR, MLP, Random Forest, and Gradient Boosting regressors.

[200]:
```
def objective_xgb_regressor(trial):
    # Suggest hyperparameters
    param = {
        "objective": "reg:squarederror",
        "eval_metric": "rmse",
        "max_depth": trial.suggest_int("max_depth", 3, 15),
        "learning_rate": trial.suggest_float("learning_rate", 0.01, 0.3,␣
↪log=True),
        "subsample": trial.suggest_float("subsample", 0.5, 1.0),
        "colsample_bytree": trial.suggest_float("colsample_bytree", 0.5, 1.0),
        "gamma": trial.suggest_float("gamma", 0, 5),
        "min_child_weight": trial.suggest_int("min_child_weight", 1, 10),
        "lambda": trial.suggest_float("lambda", 1e-3, 10.0, log=True),
        "alpha": trial.suggest_float("alpha", 1e-3, 10.0, log=True),
```

```
        "verbosity": 0,
    }

    # Convert to DMatrix
    dtrain = xgb.DMatrix(X_train, label=y_train)
    dtest = xgb.DMatrix(X_test, label=y_test)

    # Train model
    model = xgb.train(
        params=param,
        dtrain=dtrain,
        num_boost_round=trial.suggest_int("num_boost_round", 100, 1000),
        evals=[(dtest, "test")],
        verbose_eval=False,
    )

    # Predict and evaluate
    y_pred = model.predict(dtest)

    # Use R² as optimization metric
    score = r2_score(y_test, y_pred)
    return score  # maximize R²


# Create and run Optuna study
study_xgb_regressor = optuna.create_study(direction="maximize",␣
 ↪sampler=TPESampler(seed=42))
study_xgb_regressor.optimize(objective_xgb_regressor, n_trials=50,␣
 ↪show_progress_bar=True)

# Show best parameters
best_params_xgb = study_xgb_regressor.best_params
print("\nBest parameters for XGBoost Regressor:")
for k, v in best_params_xgb.items():
    print(f"  {k}: {v}")
```

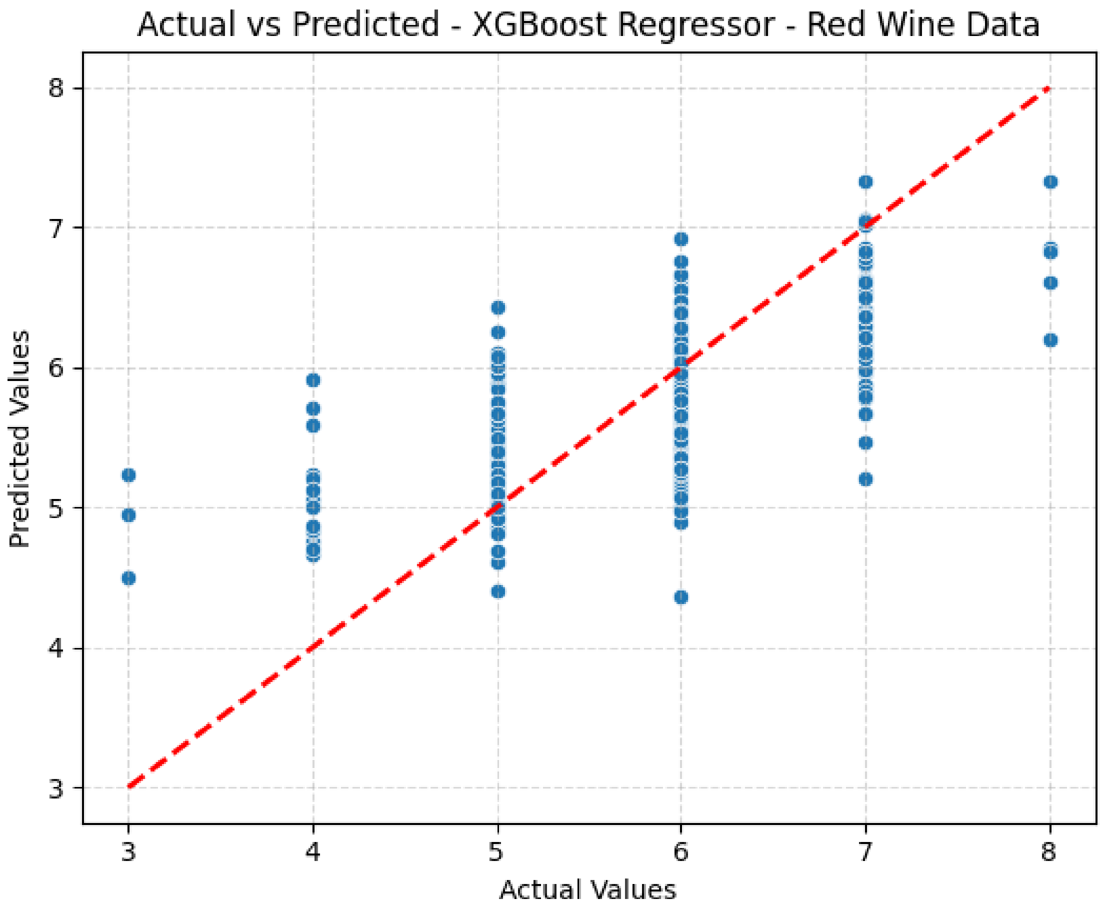
```
  0%|          | 0/50 [00:00<?, ?it/s]


Best parameters for XGBoost Regressor:
  max_depth: 15
  learning_rate: 0.03147354075174598
  subsample: 0.8380764636019498
  colsample_bytree: 0.7682440618102031
  gamma: 0.17227231572465296
  min_child_weight: 6
  lambda: 0.5428451194756965
  alpha: 0.19296942800491654
```

```
  num_boost_round: 310
```

Item b. Report the MSE, R2, and MAD (mean absolute deviation) on the test data.

[201]:
```
optimized_xgb_regressor = xgb.train(
    params={**best_params_xgb, "objective": "reg:squarederror", "eval_metric":␣
 ↪"rmse"},
    dtrain=xgb.DMatrix(X_train, label=y_train),
    num_boost_round=best_params_xgb.get("num_boost_round", 200)
)



optimized_xgb_regressor_results =␣
 ↪evaluate_regression_model(optimized_xgb_regressor, X_test, y_test,
                                                                      red_test={
                                                                          "x":␣
 ↪X_test_red,
                                                                          "y":␣
 ↪y_test_red
                                                                      },
                                                                     ␣
 ↪white_test={
                                                                          "x":␣
 ↪X_test_white,
                                                                          "y":␣
 ↪y_test_white
                                                                      })

print_regression_results(optimized_xgb_regressor_results, "XGBoost Regressor")

plot_actual_vs_predicted({
    "y_pred" : [optimized_xgb_regressor.predict(xgb.DMatrix(X_test_red))],
    "y_test": [y_test_red],
    "model_name": ["XGBoost Regressor - Red Wine Data"]
})
```

```
XGBoost Regressor
Model results on Test Data (Overall, Red Wine, White Wine).

==================================================
 Dataset      MSE       R²      MAD
Red Wine 0.324341 0.499728 0.682075
==================================================
```



8 6. Results And Comparison

Item b. Were you able to improve against the result in the paper?

The red wine accuracy reported in the paper is 62.4% at tolerance of 0.5.

For the classification models, the Random Forest and XGBoost classifiers outperformed the paper's reported accuracy for red wine, reaching accuracies of 67.08% and 67.29% , respectively.

For the regression models, theRandom Forest, and XGBoost Regressors also surpassed the paper's results, achieving accuracies 64.79%, and 67.08% for red wine, respectively at a tolerance of 0.5. At a tolerance of 1 (T=1), the MLP, Random Forest, and XGBoost Regressors have reached accuracies of 90.20%, 91.66%, and 91.25% for red wine, respectively, slightly higher than the paper's reported accuracy of 89.0% (red, Tolerance=1).

Original Paper: https://www.sciencedirect.com/science/article/pii/S0167923609001377

[202]:
```
result_table_path = "./data/wine_manu_table2.png"

from IPython.display import Image, display
```

```python
# Display the images
display(Image(filename=result_table_path))
```

Table 2
The wine modeling results (test set errors and selected models; best values in bold).

	Red wine			White wine		
	MR	NN	SVM	MR	NN	SVM
MAD	0.50 ± 0.00	0.51 ± 0.00	$\mathbf{0.46} \pm 0.00^{a}$	0.59 ± 0.00	0.58 ± 0.00	$\mathbf{0.45} \pm 0.00^{a}$
$\text{Accuracy}_{T=0.25}$ (%)	31.2 ± 0.2	31.1 ± 0.7	$\mathbf{43.2} \pm 0.6^{a}$	25.6 ± 0.1	26.5 ± 0.3	$\mathbf{50.3} \pm 1.1^{a}$
$\text{Accuracy}_{T=0.50}$ (%)	59.1 ± 0.1	59.1 ± 0.3	$\mathbf{62.4} \pm 0.4^{a}$	51.7 ± 0.1	52.6 ± 0.3	$\mathbf{64.6} \pm 0.4^{a}$
$\text{Accuracy}_{T=1.00}$ (%)	88.6 ± 0.1	88.8 ± 0.2	$\mathbf{89.0} \pm 0.2^{b}$	84.3 ± 0.1	84.7 ± 0.1	$\mathbf{86.8} \pm 0.2^{a}$
$\text{Kappa}_{T=0.5}$ (%)	32.2 ± 0.3	32.5 ± 0.6	$\mathbf{38.7} \pm 0.7^{a}$	20.9 ± 0.1	23.5 ± 0.6	$\mathbf{43.9} \pm 0.4^{a}$
Inputs ($\bar{I}$)	9.2	9.3	9.8	9.6	9.3	10.1
Model	–	$\bar{H} = 1$	$\bar{\gamma} = 2^{0.19}$	–	$\bar{H} = 2.1$	$\bar{\gamma} = 2^{1.55}$
Time (s)	**518**	847	5589	**551**	1339	30674

[a] Statistically significant under a pairwise comparison with MR and NN.
[b] Statistically significant under a pairwise comparison with MR.

8.1 6.I. Classification

Item a. Report the accuracy, precision, recall, F1-score, and confusion matrix of all the models on test data.

```python
[203]: classifier_models = [optimized_svm_classifier, optimized_mlp_classifier,␣
 ↪optimized_random_forest_classifier, optimized_xgb_classifier]
classifier_results = [optimized_svm_classifier_results,␣
 ↪optimized_mlp_classifier_results,␣
 ↪optimized_random_forest_classifier_results, optimized_xgb_classifier_results]

classifier_model_names = [
     "svm_classifier",
    "mlp_classifier",
    "random_forest_classifier",
    "xgb_classifier"
]

# prompt: find best models for each metric: accuracy, recall, precision,␣
 ↪f1-score


def find_best_models_by_metric(models, results, model_names):
    """
    Finds the best model for each classification metric: accuracy, precision,␣
 ↪recall, and F1-score.

    Parameters
    ----------
    models : list
        List of model objects or their names.
```

```
    results : list of dict
        List of result dictionaries (from evaluate_model()) corresponding to␣
↪each model.

    Returns
    -------
    pd.DataFrame
        DataFrame summarizing all results and showing the best model per metric.
    """
    # Build a DataFrame for easy comparison
    metrics_data = []
    for model, res, model_name in zip(models, results, model_names):
        # Allow both model objects and string names

        metrics_data.append({
            "Model": model_name,
            # "Accuracy": res.get("accuracy", None),
            # "Precision": res.get("precision", None),
            # "Recall": res.get("recall", None),
            # "F1-Score": res.get("f1", None),
            "[R] Accuracy": res.get("red_wine", {}).get("accuracy"),
            "[R] Precision": res.get("red_wine", {}).get("precision"),
            "[R] Recall": res.get("red_wine", {}).get("recall"),
            "[R] F1-Score": res.get("red_wine", {}).get("f1"),
            # "[W] Accuracy": res.get("white_wine", {}).get("accuracy"),
            # "[W] Precision": res.get("white_wine", {}).get("precision"),
            # "[W] Recall": res.get("white_wine", {}).get("recall"),
            # "[W] F1-Score": res.get("white_wine", {}).get("f1")
        })

    df = pd.DataFrame(metrics_data)

    # Find best model per metric
    best_models = {
        "Accuracy": df.loc[df["[R] Accuracy"].idxmax(), "Model"],
        "Precision": df.loc[df["[R] Precision"].idxmax(), "Model"],
        "Recall": df.loc[df["[R] Recall"].idxmax(), "Model"],
        "F1-Score": df.loc[df["[R] F1-Score"].idxmax(), "Model"],

    }

    print("\n Best Models by Metric:")
    for metric, model_name in best_models.items():
        print(f"  {metric}: {model_name}")

    return df, best_models
```

```
df_results, best_models = find_best_models_by_metric(classifier_models,␣
↪classifier_results, classifier_model_names)

print("\nClassification Model Results: \n [R] - Red Wine\n [W] - White Wine")
display(df_results)
```

```
 Best Models by Metric:
  Accuracy: xgb_classifier
  Precision: xgb_classifier
  Recall: xgb_classifier
  F1-Score: xgb_classifier

Classification Model Results:
 [R] - Red Wine
 [W] - White Wine
                      Model  [R] Accuracy  [R] Precision  [R] Recall  \
0            svm_classifier      0.602083       0.612864    0.602083
1            mlp_classifier      0.577083       0.539063    0.577083
2  random_forest_classifier      0.670833       0.647194    0.670833
3            xgb_classifier      0.672917       0.653972    0.672917

   [R] F1-Score
0      0.605843
1      0.535872
2      0.650094
3      0.656100
```

8.2 6.II. Regression

Item b. Report the MSE, R2, and MAD (mean absolute deviation) on the test data.

[206]:
```
def evaluate_regression_accuracy(y_pred, y_test, tolerance = 0.5):
    y_pred = np.array(y_pred)
    y_test = np.array(y_test)

    # Calculate absolute difference
    diff = np.abs(y_pred - y_test)

    # Count how many predictions are within the tolerance
    within_tolerance = np.sum(diff <= tolerance)

    # Compute accuracy as a percentage
    accuracy = within_tolerance / len(y_test)
```

```
    return accuracy
```

[207]:
```
regressor_models = [optimized_svm_regressor, optimized_mlp_regressor,␣
 ↪optimized_random_forest_regressor, optimized_xgb_regressor]
regressor_results = [optimized_svm_regressor_results,␣
 ↪optimized_mlp_regressor_results, optimized_random_forest_regressor_results,␣
 ↪optimized_xgb_regressor_results]
regressor_model_names = [
    "svm_regressor",
    "mlp_regressor",
    "random_forest_regressor",
    "xgb_regressor"
]

# prompt: find best models for each metric: mse, r2, mad

def find_best_regressors_by_metric(models, results, model_names):
    """
    Finds the best regression model for each metric: MSE, R², MAE, and MAD.

    Parameters
    ----------
    models : list
        List of regression model objects or their names.
    results : list of dict
        List of result dictionaries (from evaluate_regression_model())␣
 ↪corresponding to each model.
    model_names : list
        List of names corresponding to the models.

    Returns
    -------
    pd.DataFrame
        DataFrame summarizing all regression results and showing the best model␣
 ↪per metric.
    """
    # Collect metrics for each model
    metrics_data = []
    for model, res, model_name in zip(models, results, model_names):
        metrics_data.append({
            "Model": model_name,
            # "MSE": res.get("mse", None),
            # "R2": res.get("r2", None),
            # "MAD": res.get("mad", None),
            "[R] MSE": res.get("red_wine", {}).get("mse"),
            "[R] R2": res.get("red_wine", {}).get("r2"),
            "[R] MAD": res.get("red_wine", {}).get("mad"),
```

```
            # "[W] MSE": res.get("white_wine", {}).get("mse"),
            # "[W] R2": res.get("white_wine", {}).get("r2"),
            # "[W] MAD": res.get("white_wine", {}).get("mad"),
        })

    # Convert to DataFrame for easy comparison
    df = pd.DataFrame(metrics_data)

    # Find best models (lower = better for MSE, MAE, MAD; higher = better for␣
↪R2)
    best_models = {
        "MSE": df.loc[df["[R] MSE"].idxmin(), "Model"],
        "R2": df.loc[df["[R] R2"].idxmax(), "Model"],
        "MAD": df.loc[df["[R] MAD"].idxmin(), "Model"],
    }

    print("\n  Best Regression Models by Metric:")
    for metric, model_name in best_models.items():
        print(f"  {metric}: {model_name}")

    return df, best_models

def get_df_accuracy_with_tolerance(models, model_names, tolerance_list=[0.25, 0.
↪50, 1]):
    data = []
    for model, model_name in zip(models,  model_names):
        local_dict = {"Model": model_name}
        # Combined Test Data
        # for tolerance in tolerance_list:
        #     if isinstance(model, xgb.Booster):
        #         dtest = xgb.DMatrix(X_test)
        #         y_pred = model.predict(dtest)
        #     else:
        #         # Case 2: sklearn-compatible models (pipeline, MLP, SVR, RF,␣
↪XGBRegressor)
        #         y_pred = model.predict(X_test)

        #     acc = evaluate_regression_accuracy(y_pred, y_test, tolerance)
        #     local_dict[f"Acc(T={tolerance})"] = acc

        # Red Wine Data
        for tolerance in tolerance_list:
            if isinstance(model, xgb.Booster):
                dtest = xgb.DMatrix(X_test_red)
                y_pred = model.predict(dtest)
            else:
```

```
                # Case 2: sklearn-compatible models (pipeline, MLP, SVR, RF,␣
↪XGBRegressor)
                y_pred = model.predict(X_test_red)

            acc = evaluate_regression_accuracy(y_pred, y_test_red, tolerance)
            local_dict[f"[R] Acc(T={tolerance})"] = acc

        # # White Wine Data
        # for tolerance in tolerance_list:
        #     if isinstance(model, xgb.Booster):
        #         dtest = xgb.DMatrix(X_test_white)
        #         y_pred = model.predict(dtest)
        #     else:
        #         # Case 2: sklearn-compatible models (pipeline, MLP, SVR, RF,␣
↪XGBRegressor)
        #         y_pred = model.predict(X_test_white)

        #     acc = evaluate_regression_accuracy(y_pred, y_test_white,␣
↪tolerance)
        #     local_dict[f"[W] Acc(T={tolerance})"] = acc
        data.append(local_dict)

    # Convert to DataFrame for easy comparison
    df = pd.DataFrame(data)
    return df

# Example usage:
df_regression_results, best_regressors = find_best_regressors_by_metric(
    regressor_models, regressor_results, regressor_model_names
)
df_accuracy = get_df_accuracy_with_tolerance(regressor_models,␣
↪regressor_model_names)
print("\nRegression Models Result: \n [R] - Red Wine\n [W] - White Wine")
display(df_regression_results)
print("="*50)
display(df_accuracy)
```

```
 Best Regression Models by Metric:
  MSE: xgb_regressor
  R2: xgb_regressor
  MAD: svm_regressor

Regression Models Result:
 [R] - Red Wine
 [W] - White Wine

                    Model    [R] MSE     [R] R2    [R] MAD
```

```
0           svm_regressor  0.403677  0.377358  0.682075
1           mlp_regressor  0.396614  0.388252  0.682075
2  random_forest_regressor  0.328627  0.493117  0.682075
3           xgb_regressor  0.324341  0.499728  0.682075

==================================================

                     Model  [R] Acc(T=0.25)  [R] Acc(T=0.5)  [R] Acc(T=1)
0           svm_regressor         0.341667        0.600000      0.883333
1           mlp_regressor         0.337500        0.602083      0.902083
2  random_forest_regressor         0.406250        0.647917      0.916667
3           xgb_regressor         0.414583        0.670833      0.912500
```

8.3 6.III. Classification VS Regression

Based on your results, discuss the difference of treating this problem as classification or regression. How will this decision impact the users of your model?

Treating the wine quality prediction problem as a classification or regression task leads to different outcomes and implications for users.

In classification, the model predicts discrete quality categories (e.g., scores 3–9), providing clear and interpretable labels that are useful for decision-making, such as labeling or quality approval. However, it treats all misclassifications equally, even if a prediction is only off by one point.

In contrast, a regression model predicts continuous quality scores, capturing subtle variations between wines (e.g., predicting 6.8 instead of 7) and offering more nuanced insights for tasks like quality monitoring. While regression provides finer detail, it requires defining thresholds or tolerances to interpret results meaningfully. This is the case of the original paper (ref:https://www.sciencedirect.com/science/article/pii/S0167923609001377)

Ultimately, classification is more practical for categorical grading, whereas regression offers deeper analytical value for understanding and optimizing wine quality.

8.4 6.IV. Insights

The first thing I noticed when loading the dataset was its imbalance. Some labels have an extremely large number of samples, while others have very few. The distribution of samples was shown in Part 3. I believe this imbalance may cause the models to overfit on labels with more samples. As shown in the confusion matrix for classification and the predicted-versus-actual plot for regression, labels with fewer samples tend to be misclassified or incorrectly predicted.

9 7. Save, and Load Models

To avoid fine-tuning, and retraining every time when this notebook is run; the save and load functionalities of models are implemented.

[149]:
```
import os
import joblib
```

```python
# Create directory
save_dir = "models"
os.makedirs(save_dir, exist_ok=True)

# Save all models
for model, name in zip(regressor_models, regressor_model_names):
    if isinstance(model, xgb.Booster):
        # Save xgboost model using native save
        save_path = os.path.join(save_dir, f"{name}.json")
        model.save_model(save_path)
        print(f" Saved XGBoost model to {save_path}")
    else:
        # Save sklearn or pipeline models using joblib
        save_path = os.path.join(save_dir, f"{name}.pkl")
        joblib.dump(model, save_path)
        print(f" Saved sklearn model to {save_path}")

for model, name in zip(classifier_models, classifier_model_names):
    if isinstance(model, xgb.Booster):
        # Save xgboost model using native save
        save_path = os.path.join(save_dir, f"{name}.json")
        model.save_model(save_path)
        print(f" Saved XGBoost model to {save_path}")
    else:
        # Save sklearn or pipeline models using joblib
        save_path = os.path.join(save_dir, f"{name}.pkl")
        joblib.dump(model, save_path)
        print(f" Saved sklearn model to {save_path}")
```

```
 Saved sklearn model to models\svm_regressor.pkl
 Saved sklearn model to models\mlp_regressor.pkl
 Saved sklearn model to models\random_forest_regressor.pkl
 Saved XGBoost model to models\xgb_regressor.json
 Saved sklearn model to models\svm_classifier.pkl
 Saved sklearn model to models\mlp_classifier.pkl
 Saved sklearn model to models\random_forest_classifier.pkl
 Saved XGBoost model to models\xgb_classifier.json
```

[ ]:
```python
# Load sklearn models
svm_regoptimized_svm_classifier = joblib.load("./models/svm_classifier.pkl")
optimized_mlp_regressor = joblib.load("./models/mlp_classifier.pkl")
optimized_random_forest_classifier = joblib.load("./models/
 ↪random_forest_classifier.pkl")

svm_regoptimized_svm_regressor = joblib.load("./models/svm_regressor.pkl")
```

```
optimized_mlp_regressor = joblib.load("./models/mlp_regressor.pkl")
optimized_random_forest_regressor = joblib.load("./models/
 ↪random_forest_regressor.pkl")



# Load XGBoost Booster
optimized_xgb_classifier = xgb.Booster()
optimized_xgb_classifier.load_model("./models/xgb_classifier.json")

optimized_xgb_regressor = xgb.Booster()
optimized_xgb_regressor.load_model("./models/xgb_regressor.json")
```