ΕΛΛΗΝΙΚΟ
ΑΝΟΙΚΤΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ

Panagiotis Paltsokas
std163861@ac.eap.gr
#ΗΛΕ-48

DAMA 61 : Written Assignment 1

Problem 1

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
from sklearn.linear_model import LinearRegression
from pathlib import Path
import zipfile
import urllib.request
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score, mean_absolute_error, mean_absolute_percentage_error, mean_squared_error, accuracy_score
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split, cross_val_score, KFold
```

In [2]:
```
# extra code – code to save the figures as high-res PNGs for the book

IMAGES_PATH = Path() / "images"
IMAGES_PATH.mkdir(parents=True, exist_ok=True)

def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=300):
    path = IMAGES_PATH / f"{fig_id}.{fig_extension}"
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
```

1.1

Open a Jupyter-notebook. Download the wines dataset and load the data of the “red” wines.

Define a function that downloads and extract the required dataframe

In [3]:
```
def load_wine_quality_data():
    '''This function downloads and unzips the wine_quality.zip file,
    containing the winequality-white.csv and winequality-red.csv. After the extraction,
    the function returns the winequality-red.csv as a Pandas dataframe'''

    zip_path = Path('datasets/wine_quality.zip')  # Path to the ZIP file
    dataset_folder = Path('datasets/wine_quality_dataset')  # Path to the extracted dataset folder

    if not dataset_folder.exists():  # Check if dataset is already downloaded
        Path('datasets').mkdir(parents=True, exist_ok=True)  # In case it doesn't exist, create the datasets folder

        url = 'https://archive.ics.uci.edu/static/public/186/wine+quality.zip'  # URL of the dataset
        urllib.request.urlretrieve(url, zip_path)  # Download the ZIP file to the path given earlier

        with zipfile.ZipFile(zip_path) as zip_ref:  # Open the ZIP file
            zip_ref.extractall(dataset_folder)  # Extract all files to the dataset folder
        print('Dataset downloaded and extracted successfully.')

    return pd.read_csv(dataset_folder / 'winequality-red.csv')  # Return the DataFramea
wine_quality_data = load_wine_quality_data()
```

Load the Data set

In [4]:
```
red_wine_df = pd.read_csv('datasets/wine_quality_dataset/winequality-red.csv', sep=';')
```

Print the dataset's first rows

In [5]:
```
red_wine_df.head()
```

Out[5]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8	5
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8	5
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8	6
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5

In [6]:
```
print(f'The dataset has {red_wine_df.shape[0]} rows and {red_wine_df.shape[1]} columns')
```

```
The dataset has 1599 rows and 12 columns
```

1.2

What are the features describing the quality of the wines?

Overview of the red wine dataframe

In [7]:
```
red_wine_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1599 entries, 0 to 1598
Data columns (total 12 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   fixed acidity         1599 non-null   float64
 1   volatile acidity      1599 non-null   float64
 2   citric acid           1599 non-null   float64
 3   residual sugar        1599 non-null   float64
 4   chlorides             1599 non-null   float64
 5   free sulfur dioxide   1599 non-null   float64
 6   total sulfur dioxide  1599 non-null   float64
 7   density               1599 non-null   float64
 8   pH                    1599 non-null   float64
 9   sulphates             1599 non-null   float64
 10  alcohol               1599 non-null   float64
 11  quality               1599 non-null   int64  
dtypes: float64(11), int64(1)
memory usage: 150.0 KB
```

We notice there are 12 different features, 1599 non-null values out of 1599 entries, which means there are no missing values. 11 columns are of type float64 and describe continuous numerical features, while 1 column is of type int64, and describes a discrete numerical feature.

Let's store the feature names in a list and print them out.

In [8]:
```python
red_wine_df_col = list(red_wine_df.columns)
print(f'The features of the dataset are: {", ".join(red_wine_df_col)}.')
```

```
The features of the dataset are: fixed acidity, volatile acidity, citric acid, residual sugar, chlorides, free sulfur dioxide, tota
l sulfur dioxide, density, pH, sulphates, alcohol, quality.
```

1.3

Compute the descriptive statistics of the dataset features and discuss about their types, ranges and completeness.

Check the dataset features' descriptive statistics

In [9]:
```python
red_wine_df.describe()
```

Out[9]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	
count	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	159
mean	8.319637	0.527821	0.270976	2.538806	0.087467	15.874922	46.467792	0.996747	3.311113	0.658149	1
std	1.741096	0.179060	0.194801	1.409928	0.047065	10.460157	32.895324	0.001887	0.154386	0.169507	
min	4.600000	0.120000	0.000000	0.900000	0.012000	1.000000	6.000000	0.990070	2.740000	0.330000	
25%	7.100000	0.390000	0.090000	1.900000	0.070000	7.000000	22.000000	0.995600	3.210000	0.550000	
50%	7.900000	0.520000	0.260000	2.200000	0.079000	14.000000	38.000000	0.996750	3.310000	0.620000	1
75%	9.200000	0.640000	0.420000	2.600000	0.090000	21.000000	62.000000	0.997835	3.400000	0.730000	1
max	15.900000	1.580000	1.000000	15.500000	0.611000	72.000000	289.000000	1.003690	4.010000	2.000000	1

Inspect the features data types and completness

In [10]:
```python
red_wine_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1599 entries, 0 to 1598
Data columns (total 12 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   fixed acidity         1599 non-null   float64
 1   volatile acidity      1599 non-null   float64
 2   citric acid           1599 non-null   float64
 3   residual sugar        1599 non-null   float64
 4   chlorides             1599 non-null   float64
 5   free sulfur dioxide   1599 non-null   float64
 6   total sulfur dioxide  1599 non-null   float64
 7   density               1599 non-null   float64
 8   pH                    1599 non-null   float64
 9   sulphates             1599 non-null   float64
 10  alcohol               1599 non-null   float64
 11  quality               1599 non-null   int64  
dtypes: float64(11), int64(1)
memory usage: 150.0 KB
```

In [11]:
```
red_wine_df.isnull().sum()
```

Out[11]:
```
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
dtype: int64
```

In [12]:
```
def feature_type(column_name):
    '''
    Determines the type of a given feature.
    '''
    if column_name == 'quality':
        return 'Ordinal Categorical' # Checks if the column name is 'qualtiy', which is of ordinal categorical type.
    else:
        return 'Continuous Numerical' # Every other column is of continuous numerical type.
```

Present the type, range and completeness of the dataset features

In [13]:
```
descriptive_stats = red_wine_df.describe()

for column in descriptive_stats.columns: # Loop through the columns to retrieve its descriptive statistics and store their type, m
    f_type = feature_type(column)
    min_val = descriptive_stats.loc['min', column]
    max_val = descriptive_stats.loc['max', column]
    count_val = descriptive_stats.loc['count', column]

    # Print the features' details
    print(f'Feature: {column}')
    print(f' - Type: {f_type}')
    print(f' - Range: [{min_val}, {max_val}]')
    print(f' - Completeness: {int(count_val)} / {len(red_wine_df)} values present')
    print('-' * 40) # Separator
```

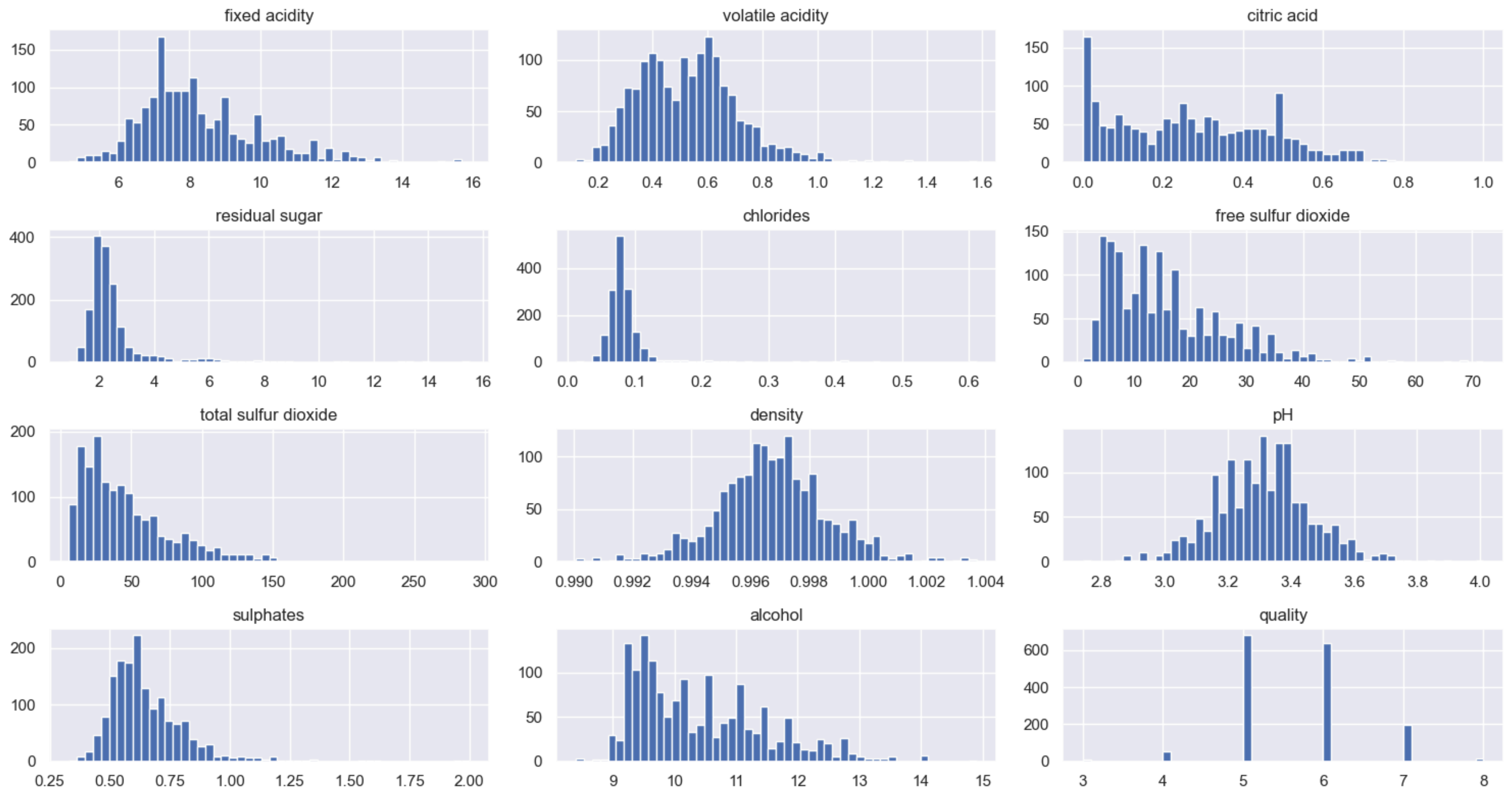
```
Feature: fixed acidity
 - Type: Continuous Numerical
 - Range: [4.6, 15.9]
 - Completeness: 1599 / 1599 values present
----------------------------------------
Feature: volatile acidity
 - Type: Continuous Numerical
 - Range: [0.12, 1.58]
 - Completeness: 1599 / 1599 values present
----------------------------------------
Feature: citric acid
 - Type: Continuous Numerical
 - Range: [0.0, 1.0]
 - Completeness: 1599 / 1599 values present
----------------------------------------
Feature: residual sugar
 - Type: Continuous Numerical
 - Range: [0.9, 15.5]
 - Completeness: 1599 / 1599 values present
----------------------------------------
Feature: chlorides
 - Type: Continuous Numerical
 - Range: [0.012, 0.611]
 - Completeness: 1599 / 1599 values present
----------------------------------------
Feature: free sulfur dioxide
 - Type: Continuous Numerical
 - Range: [1.0, 72.0]
 - Completeness: 1599 / 1599 values present
----------------------------------------
Feature: total sulfur dioxide
 - Type: Continuous Numerical
 - Range: [6.0, 289.0]
 - Completeness: 1599 / 1599 values present
----------------------------------------
Feature: density
 - Type: Continuous Numerical
 - Range: [0.99007, 1.00369]
 - Completeness: 1599 / 1599 values present
----------------------------------------
Feature: pH
 - Type: Continuous Numerical
 - Range: [2.74, 4.01]
 - Completeness: 1599 / 1599 values present
----------------------------------------
Feature: sulphates
 - Type: Continuous Numerical
 - Range: [0.33, 2.0]
 - Completeness: 1599 / 1599 values present
----------------------------------------
Feature: alcohol
 - Type: Continuous Numerical
 - Range: [8.4, 14.9]
 - Completeness: 1599 / 1599 values present
----------------------------------------
Feature: quality
 - Type: Ordinal Categorical
 - Range: [3.0, 8.0]
 - Completeness: 1599 / 1599 values present
----------------------------------------
```

Domain knowledge observations: The wines in this dataset represent a diverse range. Most of the wines fall within expected ranges for acidity, alcohol content, and pH. A wide range of values for Residual Sugar, shows that the wines include both dry and sweet wines. The quality ratings indicate that most wines are average, with no wines rated extremely poorly (min rating is 3) or excellently (max rating is 8).

1.4

Form the histograms of the features and discuss their distribution. Can the distribution of some features be improved (tending more towards the Gaussian) and how?

In [14]:
```
red_wine_df.hist(bins=50, figsize=(15,8)) # Create the histograms, with 50 bins
plt.tight_layout() # Use tight_layout so that the histogram title does not collide with the x-axis ticks/values.
save_fig('attribute_histogram_plots')
plt.show()
```



Setting aside the Quality feature, which is a categorical variable with distinct values, we observe that most distributions are right-skewed. Others have a strong right skew (Total Sulfur Dioxide, Chlorides, Residual Sugar) while others have a slight right skew (Alcohol, Fixed Acidity, Volatile Acidity). We also notice that Density and pH are nearly symmetrical, colse to normal distribution. For the right-skewed features, we should apply a logarithmic transformation to reduce the skewness and make their distributions closer to Gaussian followed by scaling techniques such as standardization, to achieve a mean of 0 and a standard deviation of 1, or min-max scaling to bring the values into a fixed range, like [0,1] or [-1,1].

1.5

Which are the features that mostly affect quality and which are those that affect it less? Provide evidence through correlation and discuss accordingly.

```
In [15]: red_wine_df.corr()
```

Out[15]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
fixed acidity	1.000000	-0.256131	0.671703	0.114777	0.093705	-0.153794	-0.113181	0.668047	-0.682978	0.183006	-0.061668	0.124052
volatile acidity	-0.256131	1.000000	-0.552496	0.001918	0.061298	-0.010504	0.076470	0.022026	0.234937	-0.260987	-0.202288	-0.390558
citric acid	0.671703	-0.552496	1.000000	0.143577	0.203823	-0.060978	0.035533	0.364947	-0.541904	0.312770	0.109903	0.226373
residual sugar	0.114777	0.001918	0.143577	1.000000	0.055610	0.187049	0.203028	0.355283	-0.085652	0.005527	0.042075	0.013732
chlorides	0.093705	0.061298	0.203823	0.055610	1.000000	0.005562	0.047400	0.200632	-0.265026	0.371260	-0.221141	-0.128907
free sulfur dioxide	-0.153794	-0.010504	-0.060978	0.187049	0.005562	1.000000	0.667666	-0.021946	0.070377	0.051658	-0.069408	-0.050656
total sulfur dioxide	-0.113181	0.076470	0.035533	0.203028	0.047400	0.667666	1.000000	0.071269	-0.066495	0.042947	-0.205654	-0.185100
density	0.668047	0.022026	0.364947	0.355283	0.200632	-0.021946	0.071269	1.000000	-0.341699	0.148506	-0.496180	-0.174919
pH	-0.682978	0.234937	-0.541904	-0.085652	-0.265026	0.070377	-0.066495	-0.341699	1.000000	-0.196648	0.205633	-0.057731
sulphates	0.183006	-0.260987	0.312770	0.005527	0.371260	0.051658	0.042947	0.148506	-0.196648	1.000000	0.093595	0.251397
alcohol	-0.061668	-0.202288	0.109903	0.042075	-0.221141	-0.069408	-0.205654	-0.496180	0.205633	0.093595	1.000000	0.476166
quality	0.124052	-0.390558	0.226373	0.013732	-0.128907	-0.050656	-0.185100	-0.174919	-0.057731	0.251397	0.476166	1.000000

In [16]:

```
corr_matrix = red_wine_df.corr()
corr_matrix['quality'].sort_values(ascending=False)
```

Out[16]:

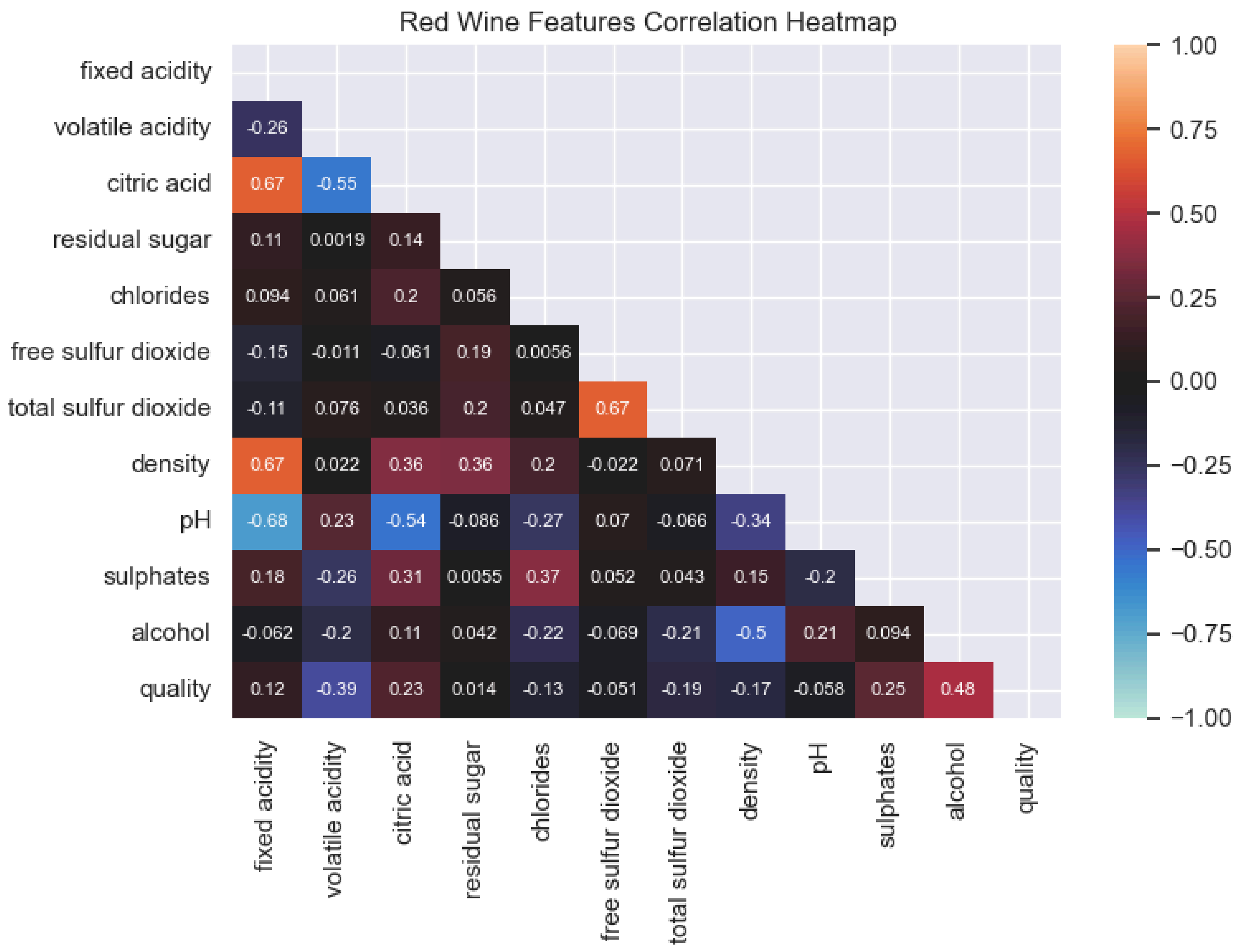
```
quality                 1.000000
alcohol                 0.476166
sulphates               0.251397
citric acid             0.226373
fixed acidity           0.124052
residual sugar          0.013732
free sulfur dioxide    -0.050656
pH                     -0.057731
chlorides              -0.128907
density                -0.174919
total sulfur dioxide   -0.185100
volatile acidity       -0.390558
Name: quality, dtype: float64
```

Create the heatmap with mask applied

In [17]:

```python
mask = np.triu(np.ones_like(red_wine_df.corr(), dtype=bool)) # Create a mask for the upper triangle

plt.figure(figsize=(8, 6))
sns.heatmap(red_wine_df.corr(), annot=True, cmap='icefire', vmin=-1, vmax=1, mask=mask, annot_kws={"size": 8})
plt.title('Red Wine Features Correlation Heatmap')
save_fig('features_correlation_heatmap')
plt.show()
```



1.6

Split the dataset into a training and a testing set retaining 80% and 20% of the total number of samples, respectively, using random shuffling and splitting that retains the statistical properties of the input data (stratified) with respect to quality.

Split the dataset into training and test sets, with a ratio of 80%-20%. I use stratified sampling to ensure that the distribution of the 'quality' target variable, is retained in the new sets.

```
In [18]: strat_train_set, strat_test_set = train_test_split(red_wine_df, test_size=0.2, stratify=red_wine_df['quality'], random_state=42)
```

Confirm with manual calculations, that the stratification maintained a similar distribution. Pass the results into a dataframe and print them.

```
In [19]: test_strat_ratio = (strat_test_set['quality'].value_counts() / len(strat_test_set)).sort_index()
         train_strat_ratio = (strat_train_set['quality'].value_counts() / len(strat_train_set)).sort_index()
         original_strat_ratio = (red_wine_df['quality'].value_counts() / len(red_wine_df)).sort_index()

         ratios_df = pd.DataFrame({
             'Original %': (original_strat_ratio * 100).round(2),
             'Train %': (train_strat_ratio * 100).round(2),
             'Test %': (test_strat_ratio * 100).round(2)
         })
```

```
In [20]: ratios_df
```

Out[20]:

	Original %	Train %	Test %
3	0.63	0.63	0.62
4	3.31	3.28	3.44
5	42.59	42.61	42.50
6	39.90	39.87	40.00
7	12.45	12.43	12.50
8	1.13	1.17	0.94

The quality distribution is retained in both training and test set. This was expected, since we used stratified splitting which ensures exactly that.

1.7

Scale the data with a Standard scaler and train a linear regression model. Evaluate the performance of the model, using the test set, with respect to metrics such as R2 -score, Mean Absolute Error, Mean Absolute Percentage Error, Mean Squared Error and Accuracy. Comment on the accuracy of predictions by plotting Actuals vs Predicted diagram.

Reset the indices for the train and the test set.

```
In [21]: strat_train_set.reset_index(drop=True, inplace=True)
         strat_test_set.reset_index(drop=True, inplace=True)
```

Separate the features and the target variable for both the train and the test set.

```
In [22]: X_train = strat_train_set.drop('quality',axis=1)
         y_train = strat_train_set['quality']

         X_test = strat_test_set.drop('quality', axis=1)
         y_test = strat_test_set['quality']
```

Initialize the StandardScaler() to scale the features to a mean of 0 and standard deviation of 1. Then, fit the scaler, **only on the training set features**, and tranform them. Finally, apply the same transformation to the test set features.

```
In [23]: target_scaler = StandardScaler()

         X_train_scaled = target_scaler.fit_transform(X_train)
         X_test_scaled = target_scaler.transform(X_test)
```

Initialize the Linear Regression model, using the scaled training features and the training target variable. Then, use the trained model to predict the 'quality' of the wines in the test set.

```
In [24]: model = LinearRegression()
         model.fit(X_train_scaled, y_train)

         y_pred = model.predict(X_test_scaled)
```

Calculate the requested metrics, to evaluate the performance of our model.

```
In [25]: # R² Score
         r2 = r2_score(y_test, y_pred)

         # Mean Absolute Error
         mae = mean_absolute_error(y_test, y_pred)

         # Mean Absolute Percentage Error
         mape = mean_absolute_percentage_error(y_test, y_pred)

         # Mean Squared Error
         mse = mean_squared_error(y_test, y_pred)
```

The target variable 'quality' in the dataset, takes integer values in range $[3,8]$. The linear regression model, outputs continuous values, that is float numbers. Since the actual wine quality is always an integer, we round those predictions, and then we keep only the ones that fall in the valid range of our dataset's wine quality. Accuracy measures how well the model performed in predicting the actual quality values. Had we not rounded the predictions, it would be virtually impossible to achieve an exact match and we would get a misleadingly low evaluation score.

In [26]:
```
y_pred_rounded = np.rint(y_pred) # Round predictions to the nearest integer

y_pred_rounded = np.clip(y_pred_rounded, y_test.min(), y_test.max()) # Ensure predictions are within the valid 'quality' range

accuracy = accuracy_score(y_test, y_pred_rounded)
```

Create a dictionary for the metrics and pass it into a dataframe.

In [27]:
```
metrics = {'Metric': ['R² Score', 'Mean Absolute Error (MAE)', 'Mean Absolute Pct Error (MPAE)', 'Mean Squared Error (MSE)', 'Accu
           'Value': [round(r2,3), round(mae,3), round(mape,3), round(mse,3), round(accuracy,3)]}
```

In [28]:
```
metrics_df = pd.DataFrame(metrics)
metrics_df
```

Out[28]:

	Metric	Value
0	R^2 Score	0.370
1	Mean Absolute Error (MAE)	0.495
2	Mean Absolute Pct Error (MPAE)	0.091
3	Mean Squared Error (MSE)	0.406
4	Accuracy	0.597

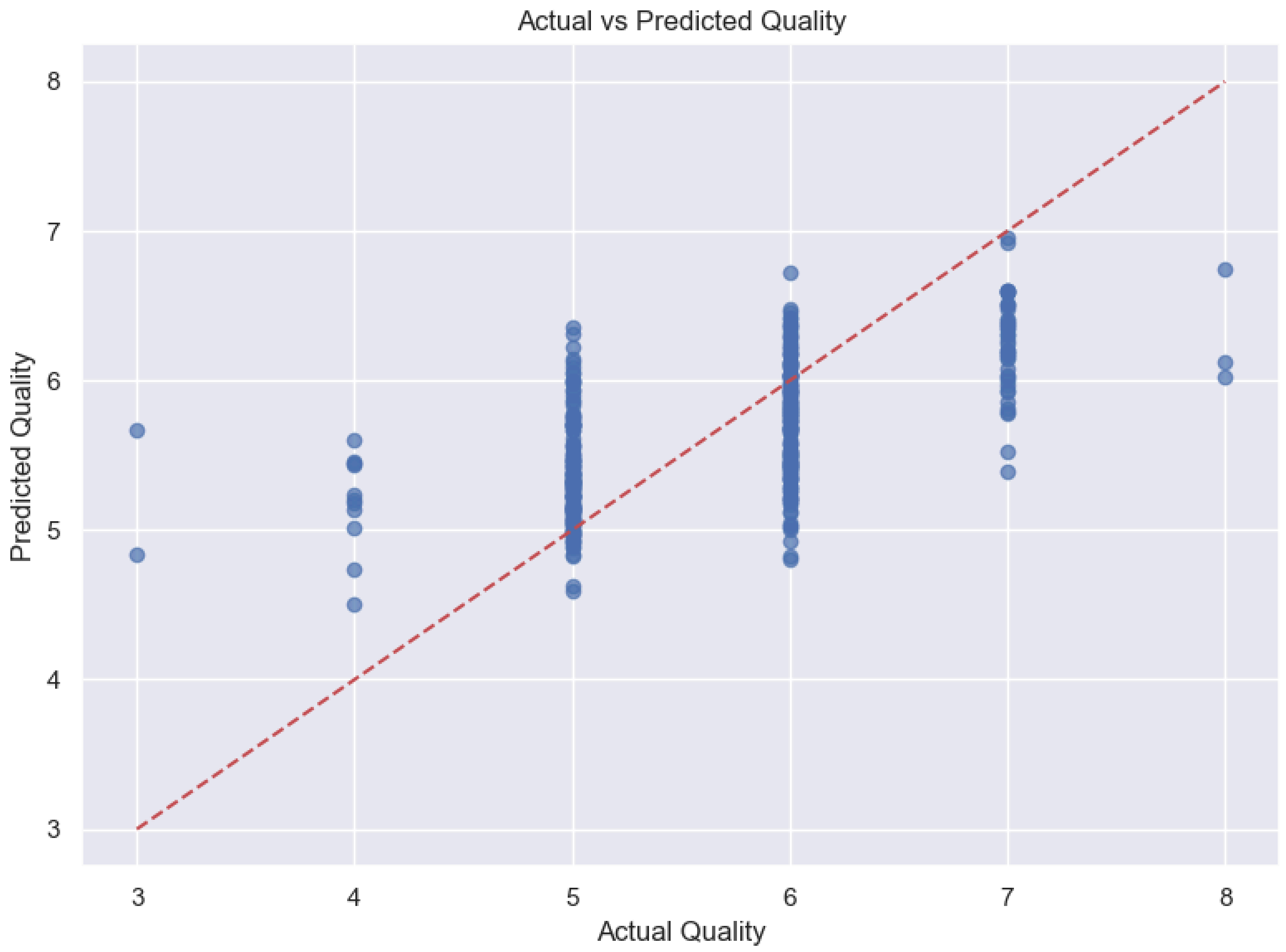
The model's R^2 score of 0.37, suggests that only 37% of the variance in the 'quality' scores is being explained by the model and the model's Accuracy score of 0.597, suggests that the model correctly predicts the wine quality around 59.7% of the time. Our model is not performing particularly well, which might mean that a linear regression model might not be the best fit for this problem.

This can also be seen in the visualization below.

Plot Actual vs Predicted values

In [29]:
```
plt.figure(figsize=(8,6))
plt.scatter(y_test, y_pred, alpha=0.7, color='b')
plt.xlabel('Actual Quality')
plt.ylabel('Predicted Quality')
plt.title('Actual vs Predicted Quality')
diagonal = np.linspace(y_test.min(), y_test.max(), 100)  # Generate 100 values between the minimum value of the y_test and
```

```
                                                # the maximum value of y_test
plt.plot(diagonal, diagonal, 'r--') # Draw a diagonal line, where x and y are equal, hence the y=x line. That represents a
                                    # diagonal reference line that indicates the perfect predictions.
save_fig('actual_VS_predicted_quality')
plt.show()
```



We observe that the points fall far off from the reference prediction reference line.

1.8

Perform 10-fold cross validation and compute the mean and standard deviation of the scores over the folds. Is the model's R2-score within the limits defined by the 10-fold cross validation?

Initialize the linear regression model, and define the 10-fold CV using KFold. Perform cross validation, computing the R^2 for each fold, and store the results in cv_scores.

In [30]:
```
linear_model = LinearRegression()

kf = KFold(n_splits=10, shuffle=True, random_state=42)

cv_scores = cross_val_score(linear_model, X_train_scaled, y_train, cv=kf, scoring='r2')
print("Cross-Validation R² Scores:")
print(cv_scores)
```

```
Cross-Validation R² Scores:
[0.43091694 0.29793909 0.41938328 0.27331003 0.39947422 0.19204422
 0.38719947 0.25085842 0.34651846 0.26805618]
```

Calculate the mean and standard deviation of R^2 scores

In [31]:
```
mean_r2 = cv_scores.mean()
std_r2 = cv_scores.std()

print(f"\nMean R² Score: {mean_r2:.4f}")
print(f"Standard Deviation of R² Scores: {std_r2:.4f}")
```

```
Mean R² Score: 0.3266
Standard Deviation of R² Scores: 0.0773
```

Train the model on the entire training set

In [32]:
```
linear_model.fit(X_train_scaled, y_train)
```

Out[32]:
```
▾ LinearRegression

LinearRegression()
```

Predict the target variable for the test data

In [33]:
```
y_pred = linear_model.predict(X_test_scaled)
```

Calculate R^2 score on the test set

In [34]:
```
test_r2 = r2_score(y_test, y_pred)

print(f"\nTest Set R² Score: {test_r2:.4f}")
```

```
Test Set R² Score: 0.3703
```

Calculate the limits

In [35]:
```
lower_limit = mean_r2 - std_r2
upper_limit = mean_r2 + std_r2

print(f"\nCross-Validation R² Score Range: [{lower_limit:.4f}, {upper_limit:.4f}]")
```

```
Cross-Validation R² Score Range: [0.2492, 0.4039]
```

Check if test R^2 score is within the limits

In [36]:
```
if lower_limit <= test_r2 <= upper_limit:
    print(f"The model's test R² score: {test_r2:.4f} is within the limits defined by the 10-fold cross-validation.")
else:
    print(f"The model's test R² score: {test_r2:.4f} is NOT within the limits defined by the 10-fold cross-validation.")
```

```
The model's test R² score: 0.3703 is within the limits defined by the 10-fold cross-validation.
```

Problem 2

In [37]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import random
from sklearn.model_selection import train_test_split, cross_val_score, StratifiedKFold, cross_val_predict
from sklearn.linear_model import SGDClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.metrics import confusion_matrix, recall_score, precision_score, accuracy_score
from sklearn.dummy import DummyClassifier
```

2.1

Load the data as arrays and split them into training and test sets with the next ratio: 85-15. Verify that all the classes have the adequate number of instances.

Load the MNIST data

```
In [38]: from sklearn.datasets import fetch_openml

         mnist = fetch_openml('mnist_784', as_frame = False)
```

Inspect the data

```
In [40]: X,y = mnist.data, mnist.target
         print(f'{"-" * 30}\ndata array:\n {X}, \n\nshape : {X.shape}')
         print(f'{"-" * 30}\ntarget array:\n {y}, \n\nshape : {y.shape}\n{"-" * 30}')
```

```
------------------------------
data array:
 [[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]],

shape : (70000, 784)
------------------------------
target array:
 ['5' '0' '4' ... '4' '5' '6'],

shape : (70000,)
------------------------------
```

Split into training and test sets

```
In [41]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15, stratify=y, random_state=42)
```

Check the distribution of labels in both training and test sets

```
In [42]: unique_labels, train_counts = np.unique(y_train, return_counts=True) # Get the unique labels and their counts in the train set
         train_dict = dict(zip(unique_labels, train_counts)) # Map each label to its count and store them as a dictionary

         unique_labels, test_counts = np.unique(y_test, return_counts=True)
         test_dict = dict(zip(unique_labels, test_counts))

         print(f'{train_dict}\n{test_dict}')
```

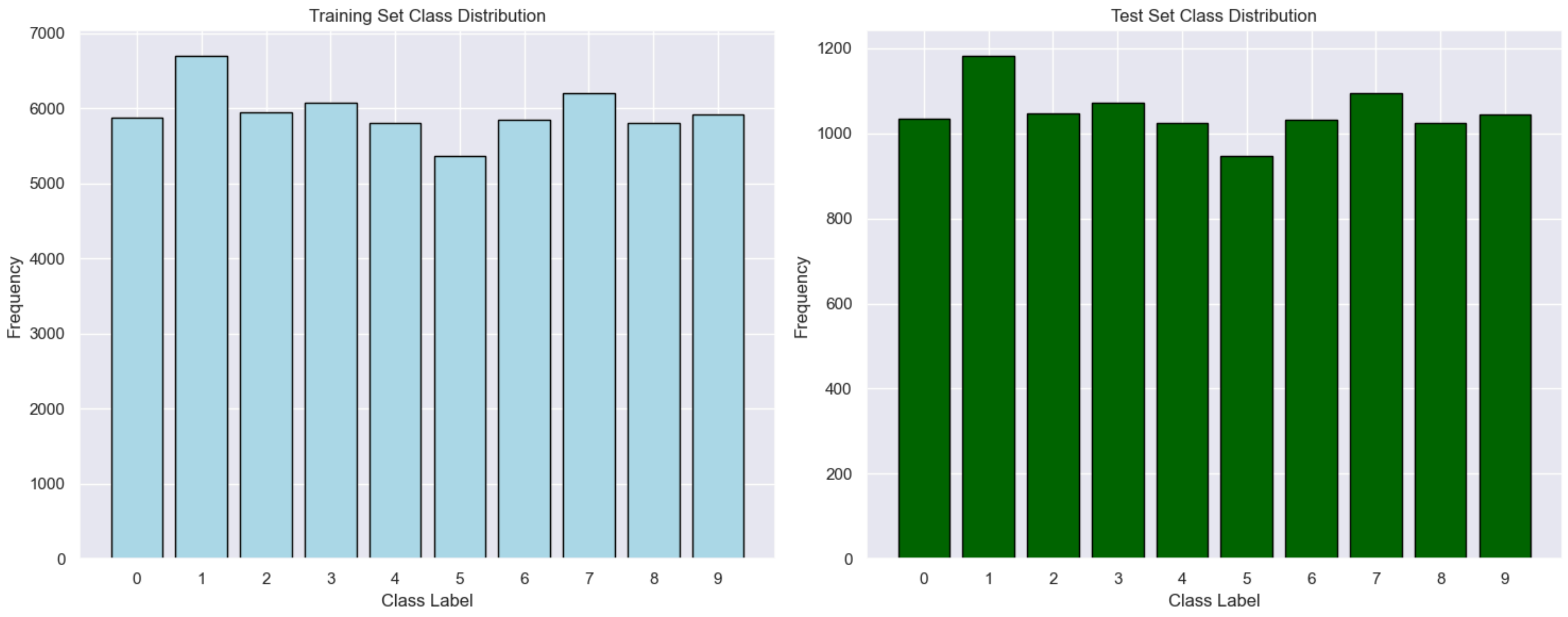
```
{'0': 5868, '1': 6695, '2': 5942, '3': 6070, '4': 5800, '5': 5366, '6': 5845, '7': 6199, '8': 5801, '9': 5914}
{'0': 1035, '1': 1182, '2': 1048, '3': 1071, '4': 1024, '5': 947, '6': 1031, '7': 1094, '8': 1024, '9': 1044}
```

Create a visualization to confirm that all classes have the adequate number of instances.

In [43]:
```
fig, axs = plt.subplots(1, 2, figsize=(15, 6)) # Set up the layout
axs[0].bar(train_dict.keys(), train_dict.values(), color='lightblue', edgecolor='black')
axs[0].set_xticks(range(10))
axs[0].set_title('Training Set Class Distribution')
axs[0].set_xlabel('Class Label')
axs[0].set_ylabel('Frequency')

axs[1].bar(test_dict.keys(), test_dict.values(), color='darkgreen', edgecolor='black')
axs[1].set_xticks(range(10))
axs[1].set_title('Test Set Class Distribution')
axs[1].set_xlabel('Class Label')
axs[1].set_ylabel('Frequency')

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



We observe that there is no class with very few or way too many instances, and our data is well balanced in both the training and the test set.

2.2

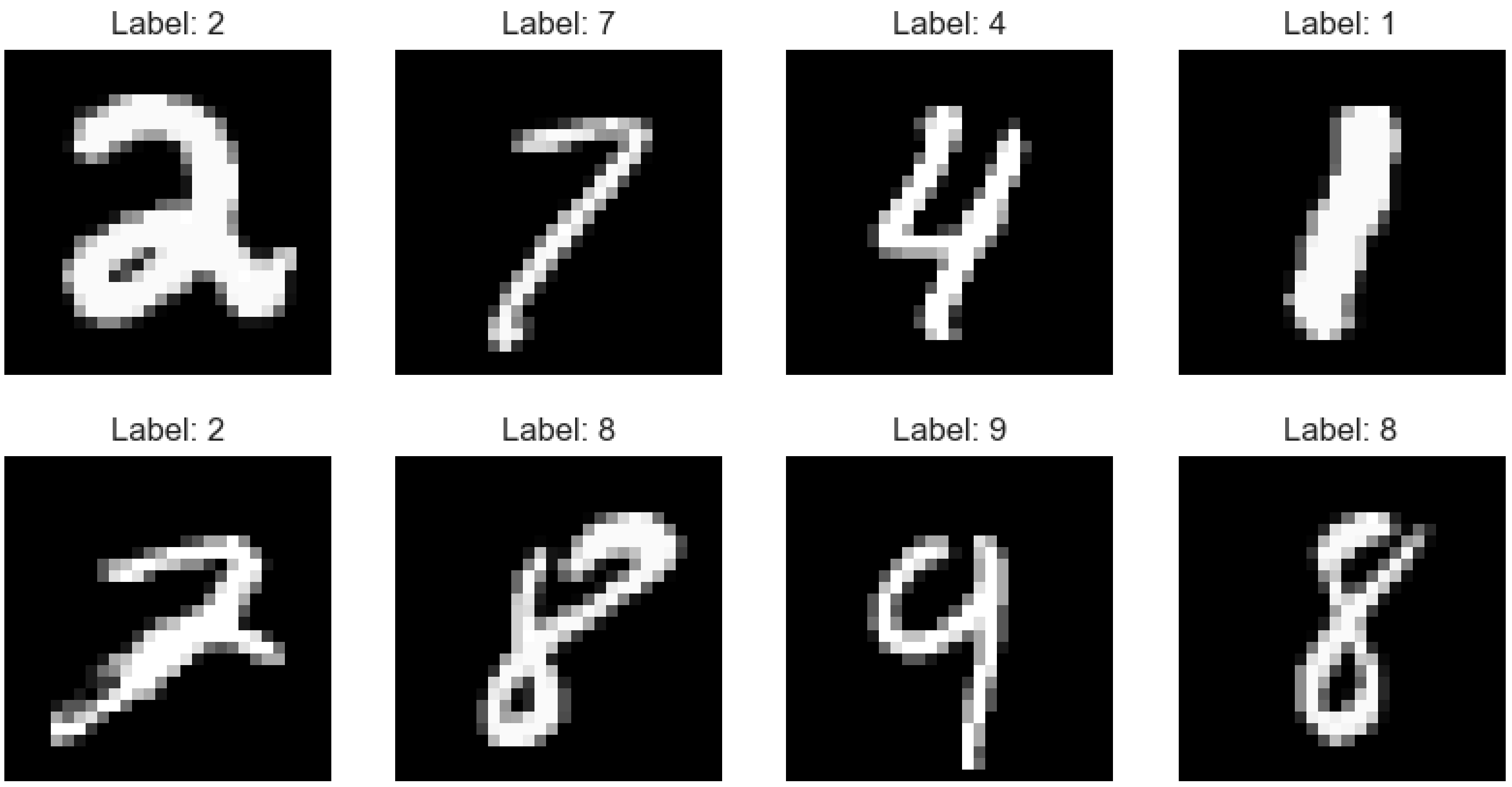
Depict the first 8 images of the created training and test sets using different subplots in a 2 by 4 frame, with their labels as titles.

Plot the first 8 images from the training set

In [44]:
```
fig, axs = plt.subplots(2, 4, figsize=(10, 5))

index = 0
for row in range(2):
    for col in range(4):
        axs[row, col].imshow(X_train[index].reshape(28, 28), cmap='gray')
        axs[row, col].set_title(f"Label: {y_train[index]}")
        axs[row, col].axis('off')
        index += 1

plt.show()
```

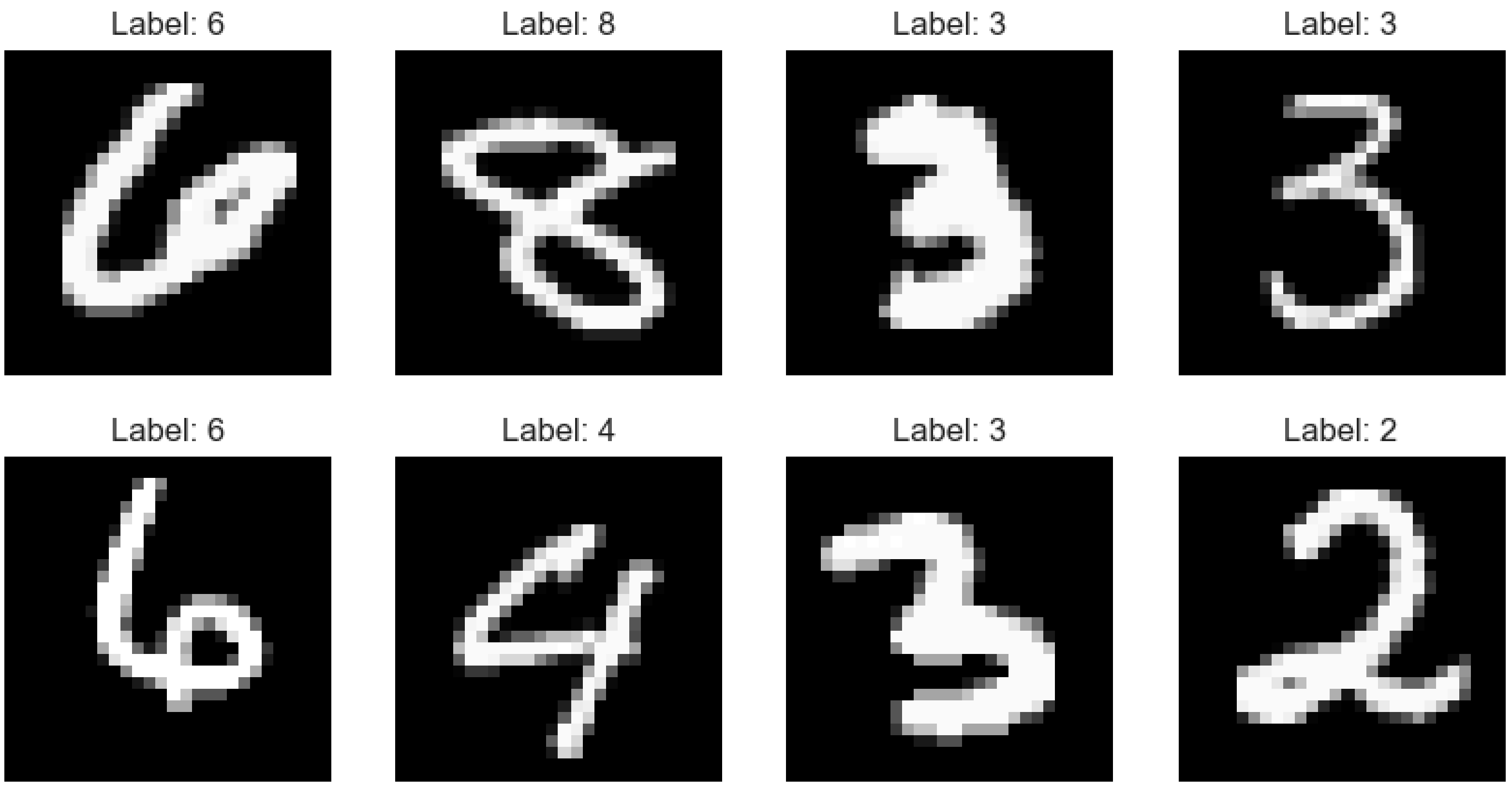


Plot the first 8 images from the test set

```
In [45]: fig, axs = plt.subplots(2, 4, figsize=(10, 5))

index = 0
for row in range(2):
    for col in range(4):
        axs[row, col].imshow(X_test[index].reshape(28, 28), cmap='gray')
        axs[row, col].set_title(f"Label: {y_test[index]}")
        axs[row, col].axis('off')
        index += 1

plt.show()
```



2.3

We need to handle a classification problem of distinguishing between two classes: even and odd numbers. First, create the training and test subsets for each class. Then, choose a binary classifier

and a normalization technique of your choice, before wrapping them into a scikit-learn pipeline. Fit your pipeline to observe the created diagram.

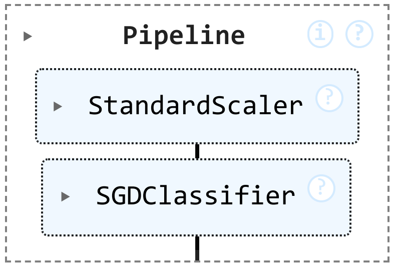
Create binary labels for even and odd numbers

In [46]:
```
y_train_binary = (y_train.astype(int) % 2 == 0) # If the remainder of the division with 2, is 0, then the value is an even
                                                # number (returns 1). If the remainder is 1, then it is an odd
                                                # number (returns 0).
y_test_binary = (y_test.astype(int) % 2 == 0)
```

Create a pipeline with StandardScaler and SGDClassifier. Then fit it on the training data

In [47]:
```
pipeline = Pipeline([('scaler', StandardScaler()), ('sgd_clf', SGDClassifier(random_state=42))])

pipeline.fit(X_train, y_train_binary)
```

Out[47]:


2.4

Use 3-fold cross validation and evaluate your classification pipeline by calculating the next metrics: accuracy, recall, and precision. Compare the predictive performance of your model against a dummy model that always guesses that an image belongs to the even category.

Evaluate accuracy, recall, and precision using 3-fold cross-validation

In [48]:
```
accuracy = cross_val_score(pipeline, X_train, y_train_binary, cv=3, scoring='accuracy')
recall = cross_val_score(pipeline, X_train, y_train_binary, cv=3, scoring='recall')
precision = cross_val_score(pipeline, X_train, y_train_binary, cv=3, scoring='precision')

print(f'Model Accuracy: {round(accuracy.mean()*100,2)}%')
print(f'Mean Recall: {round(recall.mean()*100,2)}%')
print(f'Model Precision: {round(precision.mean()*100,2)}%')
```

```
Model Accuracy: 88.33%
Model Recall: 87.67%
Model Precision: 88.48%
```

Create a dummy classifier that always predicts "even"

```
In [49]: dummy_clf = DummyClassifier(strategy = 'constant', constant = True) # We want the model to always predict even (1), therefore,
                                                                               # we use strategy='constant' and set the constant value to
                                                                               # True = 1 = even
         dummy_clf.fit(X_train, y_train_binary) # Fit the dummy classifier to the training data
         dummy_accuracy = cross_val_score(dummy_clf, X_train, y_train_binary, cv=3, scoring='accuracy') # Calculate its accuracy.

         print(f'Dummy Model Accuracy: {round(dummy_accuracy.mean()*100,2)}%')
```

```
Dummy Model Accuracy: 49.17%
```

The Dummy Classifier always predicts 'even'. Given the dataset is split into $N(even)$ even and $N(odd) = n - N(even)$ odd values, it will predict:
TP=$N(even)$ , **FP**=$N(odd)$, **TN=FN=0**, with $N(even) + N(odd) = n$ .
This means that:
$\;precision =\frac{TP}{TP+FP} = \frac{N(even)}{N(even)+N(odd)} = \frac{N(even)}{n} = accuracy$
and also
$\;recall=\frac{TP}{TP+FN}=\frac{N(even)}{N(even)+0}=1$
Therefore, examining precision and recall for this dummy model adds no extra insight beyond what the accuracy tells us.

Comparing the performance of the two models

The trained pipeline exhibits much better predictive power compared to the dummy model. Its accuracy (88.33%) is far above random guessing. This tells us that the model is succesfully learning to differentiate between even and odd digits in contrast to the dummy model, that lacks the ability to make informed predictions.

2.5

Calculate the confusion matrix for the training set, following the same 3-fold cross validation protocol. Record the kind and the amount of the predictions based on that.

Generate predictions using 3-fold cross-validation for the training set

```
In [50]: y_train_pred = cross_val_predict(pipeline, X_train, y_train_binary, cv = 3)
```

Calculate and print the confusion matrix

In [51]:
```
cm = confusion_matrix(y_train_binary, y_train_pred)
print(f'Confusion Matrix for the Training Set:\n{cm}')
```

```
Confusion Matrix for the Training Set:
[[26905  3339]
 [ 3607 25649]]
```

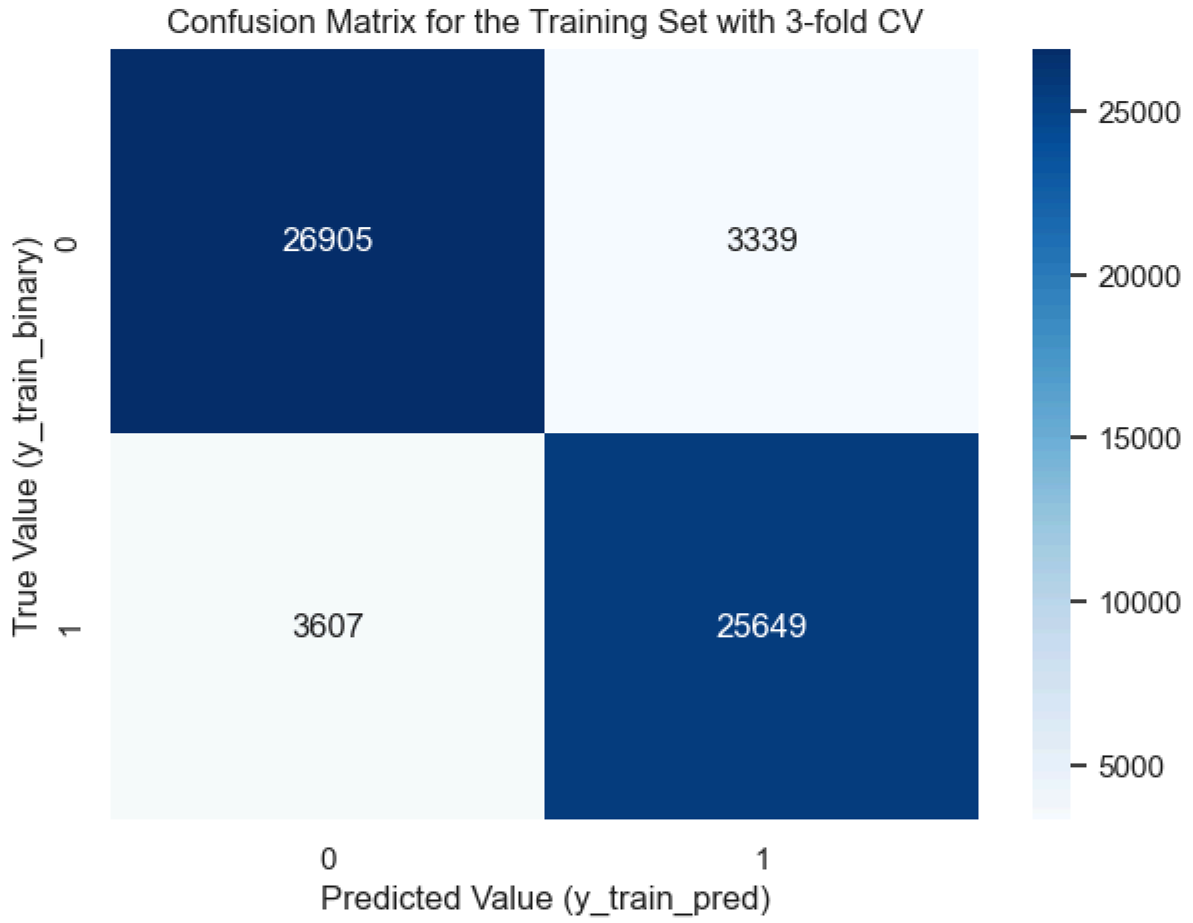
Extract and print the confusion matrix components

In [52]:
```
tn, fp, fn, tp = cm.ravel()
print(f"True Negatives (TN): {tn}, False Positives (FP): {fp}, False Negatives (FN): {fn}, True Positives (TP): {tp}")
```

```
True Negatives (TN): 26905, False Positives (FP): 3339, False Negatives (FN): 3607, True Positives (TP): 25649
```

Visualize the confusion matrix for better readability

In [53]:
```
plt.figure(figsize=(7, 5))
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g')
plt.title('Confusion Matrix for the Training Set with 3-fold CV')
plt.xlabel('Predicted Value (y_train_pred)')
plt.ylabel('True Value (y_train_binary)')
plt.show()
```



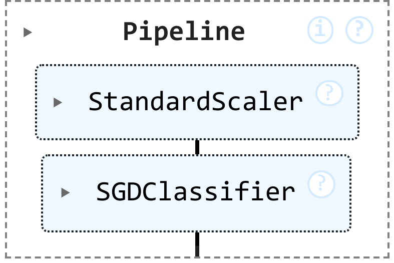
2.6

Train the same pipeline over all the training set, and apply that on the test set for getting your predictions. Extract again the confusion matrix, and comment any great changes in the behavior of your model.

Train the classifier on the entire training set

```
In [54]: pipeline.fit(X_train, y_train_binary)
```

Out[54]:



Predict the labels on the test set

In [55]:
```
y_test_pred = pipeline.predict(X_test)
```

Calculate and print the confusion matrix for the test set

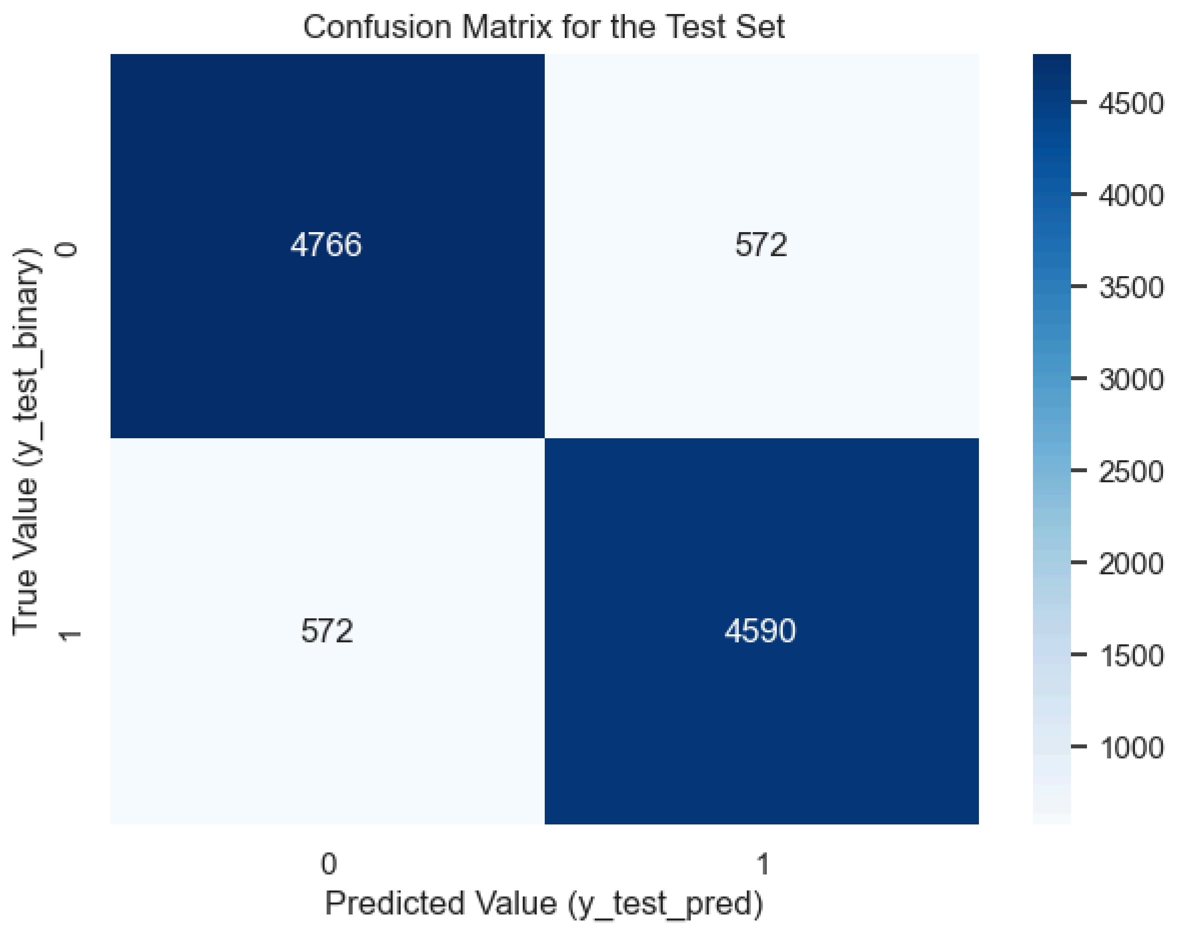
In [56]:
```
cm_test = confusion_matrix(y_test_binary, y_test_pred)

print(f'Confusion Matrix for the Test Set:\n{cm_test}')
```

```
Confusion Matrix for the Test Set:
[[4766  572]
 [ 572 4590]]
```

Visualize the confusion matrix for better readability

In [57]:
```
plt.figure(figsize=(7, 5))
sns.heatmap(cm_test, annot=True, cmap='Blues', fmt='g')
plt.title('Confusion Matrix for the Test Set')
plt.xlabel('Predicted Value (y_test_pred)')
plt.ylabel('True Value (y_test_binary)')
plt.show()
```



2.7

Pick one random instance from those that belong to false positives and false negatives from the test set, and depict their original images in separate figures.

Find the indices of false positives and false negatives

In [58]:
```
false_positives = np.where((y_test_binary == 0) & (y_test_pred == 1))[0]
false_negatives = np.where((y_test_binary == 1) & (y_test_pred == 0))[0]
```

Select one random instance from false positives and false negatives

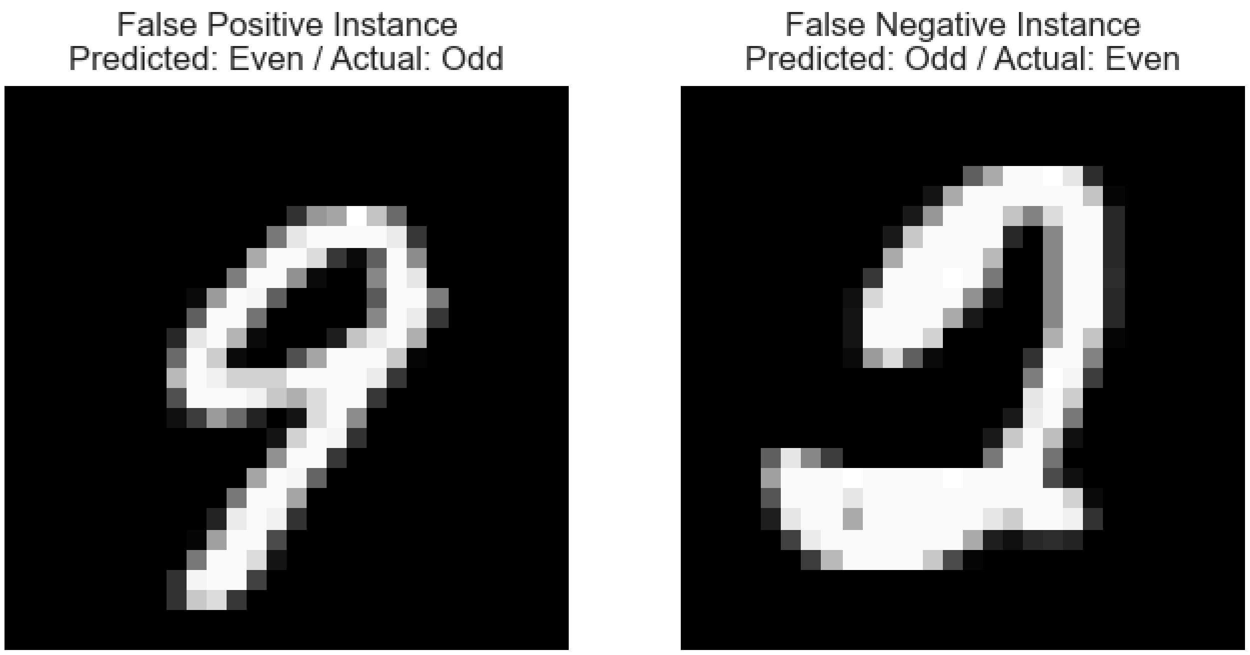
In [59]:
```
random_fp_index = random.choice(false_positives.tolist())
random_fn_index = random.choice(false_negatives.tolist())
```

Depict the selected false positive and false negative instances

```
In [60]: fig, axs = plt.subplots(1, 2, figsize=(8, 8))
         axs[0].imshow(X_test[random_fp_index].reshape(28, 28), cmap='gray')
         axs[0].set_title(f"False Positive Instance\nPredicted: Even / Actual: Odd")
         axs[0].axis('off')

         axs[1].imshow(X_test[random_fn_index].reshape(28, 28), cmap='gray')
         axs[1].set_title(f"False Negative Instance\nPredicted: Odd / Actual: Even")
         axs[1].axis('off')

         plt.show()
```



```
In [ ]:
```