!!! PLEASE UPDATE LOCAL base_path BEFORE RUNNING !!!

In [1]:
```
import pandas as pd
import numpy as np
from scipy import stats
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
import matplotlib.pyplot as plt
from matplotlib.colors import LinearSegmentedColormap
import seaborn as sns
```

Part (A)

In [2]:
```
#Part a) reads and loads the data (from the downloaded data set);

# Define the path to the datasets
base_path = '/Users/grovercastroduenas/Desktop/Machine Learning/Group Assignment/wine+quality/' ###set path to where the wine

# Load the white wine data
white_wine_path = base_path + 'winequality-white.csv'
###white_wine_path = 'winequality-white.csv' ### use this line if the winequality-white.csv file is in the same folder as thi
white_wine_df = pd.read_csv(white_wine_path, sep=';')  # The delimiter is ';'
print("\nWhite Wine Data Loaded Successfully. First 5 rows:")
print(white_wine_df.head())  # Display the first few rows of the white wine dataset

# Checking for missing values in each column
missing_values = white_wine_df.isnull().sum()
print("Missing values in each column:\n", missing_values)

# Get mean, median, std
white_wine_df.describe()
```

```
White Wine Data Loaded Successfully. First 5 rows:
   fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0            7.0              0.27         0.36            20.7      0.045
1            6.3              0.30         0.34             1.6      0.049
2            8.1              0.28         0.40             6.9      0.050
3            7.2              0.23         0.32             8.5      0.058
4            7.2              0.23         0.32             8.5      0.058

   free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
0                 45.0                 170.0   1.0010  3.00       0.45
1                 14.0                 132.0   0.9940  3.30       0.49
2                 30.0                  97.0   0.9951  3.26       0.44
3                 47.0                 186.0   0.9956  3.19       0.40
4                 47.0                 186.0   0.9956  3.19       0.40

   alcohol  quality
0      8.8        6
1      9.5        6
2     10.1        6
3      9.9        6
4      9.9        6
Missing values in each column:
 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
```

Out[2]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
count	4898.000000	4898.000000	4898.000000	4898.000000	4898.000000	4898.000000	4898.000000	4898.000000	4898.000000	4898.000000	4898.000000
mean	6.854788	0.278241	0.334192	6.391415	0.045772	35.308085	138.360657	0.994027	3.188267	0.489847	10.514267
std	0.843868	0.100795	0.121020	5.072058	0.021848	17.007137	42.498065	0.002991	0.151001	0.114126	1.230621
min	3.800000	0.080000	0.000000	0.600000	0.009000	2.000000	9.000000	0.987110	2.720000	0.220000	8.000000
25%	6.300000	0.210000	0.270000	1.700000	0.036000	23.000000	108.000000	0.991723	3.090000	0.410000	9.500000
50%	6.800000	0.260000	0.320000	5.200000	0.043000	34.000000	134.000000	0.993740	3.180000	0.470000	10.400000
75%	7.300000	0.320000	0.390000	9.900000	0.050000	46.000000	167.000000	0.996100	3.280000	0.550000	11.400000
max	14.200000	1.100000	1.660000	65.800000	0.346000	289.000000	440.000000	1.038980	3.820000	1.080000	14.200000

The above codes loads the white wine dataset from a CSV file and checks for missing values in each column.

To be more specific, we started by importing the pandas library to handle the data. Then, we defined the path to the dataset and load it using pd.read_csv, specifying that the delimiter is a semicolon (';') because the dataset uses this character to separate values in each row. After loading the data, we usec df.head() to display the first five rows of the dataset, allowing us to visually inspect the initial data loaded into the DataFrame.

To check for missing values, we used the isnull() method followed by sum(). This combination provides a count of missing values for each column in the DataFrame. This helps us in assessing the data quality and integrity before proceeding with any further analysis or machine learning tasks.

There are 4,898 total records in our dataset. From the output, we confirm that there are no missing values in the dataset, as all columns show a count of zero missing entries, meaning that we won't need to handle missing data before moving on to the next steps. The data appears well-prepared and clean, which facilitates a smoother workflow for any data processing or analysis tasks that follow.

Outliers and Duplicate Checks

After loading our data, we checked for potential outliers and duplicates which could affect our model.

In [3]:
```
# count outliers
outliers_check = white_wine_df[(np.abs(stats.zscore(white_wine_df)) < 3).all(axis=1)]
outliers_count = len(white_wine_df) - len(outliers_check)
print("Number of records with outliers: "+ str(outliers_count))
```

```
Number of records with outliers: 411
```

We counted the number of potential outliers by identifying which values in the attribute columns were above 3 standard deviations from the mean for that respective column. We found 411 rows with outliers using this method, however, decided not to drop them at this stage because this will be handled during our z-score normalization in Section D below.

In [4]:
```
## Duplicate check
# save copy of our white wine df
duplicates_check = white_wine_df

# Drop duplicates
duplicates_check.insert(0, 'CONCAT', range(0, 0 + len(duplicates_check)))
duplicates_check["CONCAT"] = duplicates_check["fixed acidity"].map(str) + duplicates_check["volatile acidity"].map(str) + dup

duplicates = duplicates_check.duplicated(subset=['CONCAT'])
duplicates_check.insert(1, 'Duplicates', duplicates)

print('Number of duplicate rows to be removed: ' + str(len(duplicates_check[duplicates_check['Duplicates']]==True)) + " rows"
duplicates_check = duplicates_check.drop(duplicates_check[duplicates_check['Duplicates']==True].index, inplace = False)
print("Number of rows remaining: " + str(len(duplicates_check)) + " rows")

#drop concat column
white_wine_df = white_wine_df.drop('CONCAT', axis=1)
white_wine_df = white_wine_df.drop('Duplicates', axis=1)
```

```
Number of duplicate rows to be removed: 937 rows
Number of rows remaining: 3961 rows
```

To check for duplicates, we concatenated the values across each row and put the concatenated output into a new ID column for each respective row. If a record has identical values across the entire row, it makes sense that this could be a potential duplicate. However, after long discussions, we decided against dropping these potential duplicates for two reasons.

First, the potential duplicates identified represent almost 20% of the entire dataset. This seemed far too high of a proportion to assume that these are duplicates. The reliability of our classifier and results later on in this exercise depends on having a large enough dataset to begin with, and we believe that these records with the same physical attributes would be useful in our model.

Second, we noticed on the winemaker's website that there are 6 varieties of the Vinho Verde white grapes, so there is not an infinite amount of Vinho Verde wines that can be taken for samples. We assumed it would make sense that some randomly chosen bottles out of all the randomly chosen bottles could have identical physical compositions. In other words, we believe that these potential duplicates pairings are actually just two or more bottles from the same harvest/bottling process, which could make them have identical compositions. These should be treated as seperate records even if they're from the same harvest and bottling season, because utlimately a bottle is an individual record.

Therefore, we decided that it would be best to not drop these records.

Definitions of physical attributes of wine

Fixed acidity: *acids that are difficult to evaporate (g/dm^3).*

Volatile acidity: *level of acetic acid (higher acetic acid leads to vinegar)(g/dm^3).*

Citric acid: *creates freshness and flavour in wines (g/dm^3).*

Residual sugar: *sugar remaining after fermentation (g/dm^3).*

Chlorides: *salts (sodium chloride - g/dm^3).*

Free sulfur dioxide: *equilibrium of sulfur dioxide and bisulfite ion (mg/dm^3).*

Total sulfur dioxide: *total free + bound levels of sulfur dioxide (mg/dm^3).*

Density: *g/cm^3*

pH: *ranges from 0 (very acidic) to 14 (very basic). Wine pH typically falls between 3-4.*

Sulphates: *antioxidant/antimicrobial additives that build presence of sulfur dioxide gas (potassium sulphate - g/dm3).*

Alcohol: *percentage of alcohol of the wine (% of volume).*

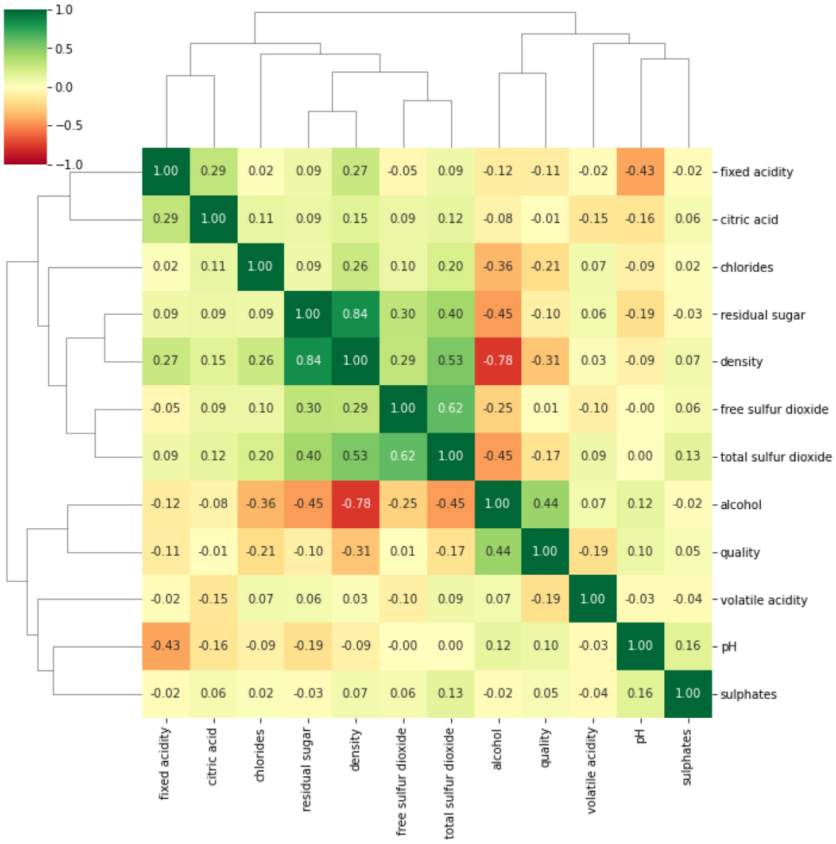
Quality: *rating of the wine from 1 to 10 (ordinal).*

Correlation coefficients

In [5]:
```
#Correlation between attributes, using Red-Yellow-Green scale
corr = white_wine_df.corr()
cmap='RdYlGn'
sns.clustermap(corr, cmap='RdYlGn', vmin=-1, vmax=1, fmt='.2f', annot=True)
corr.style.background_gradient(cmap)
```

Out[5]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
fixed acidity	1.000000	-0.022697	0.289181	0.089021	0.023086	-0.049396	0.091070	0.265331	-0.425858	-0.017143	-0.120881	-0.113663
volatile acidity	-0.022697	1.000000	-0.149472	0.064286	0.070512	-0.097012	0.089261	0.027114	-0.031915	-0.035728	0.067718	-0.194723
citric acid	0.289181	-0.149472	1.000000	0.094212	0.114364	0.094077	0.121131	0.149503	-0.163748	0.062331	-0.075729	-0.009209
residual sugar	0.089021	0.064286	0.094212	1.000000	0.088685	0.299098	0.401439	0.838966	-0.194133	-0.026664	-0.450631	-0.097577
chlorides	0.023086	0.070512	0.114364	0.088685	1.000000	0.101392	0.198910	0.257211	-0.090439	0.016763	-0.360189	-0.209934
free sulfur dioxide	-0.049396	-0.097012	0.094077	0.299098	0.101392	1.000000	0.615501	0.294210	-0.000618	0.059217	-0.250104	0.008158
total sulfur dioxide	0.091070	0.089261	0.121131	0.401439	0.198910	0.615501	1.000000	0.529881	0.002321	0.134562	-0.448892	-0.174737
density	0.265331	0.027114	0.149503	0.838966	0.257211	0.294210	0.529881	1.000000	-0.093591	0.074493	-0.780138	-0.307123
pH	-0.425858	-0.031915	-0.163748	-0.194133	-0.090439	-0.000618	0.002321	-0.093591	1.000000	0.155951	0.121432	0.099427
sulphates	-0.017143	-0.035728	0.062331	-0.026664	0.016763	0.059217	0.134562	0.074493	0.155951	1.000000	-0.017433	0.053678
alcohol	-0.120881	0.067718	-0.075729	-0.450631	-0.360189	-0.250104	-0.448892	-0.780138	0.121432	-0.017433	1.000000	0.435575
quality	-0.113663	-0.194723	-0.009209	-0.097577	-0.209934	0.008158	-0.174737	-0.307123	0.099427	0.053678	0.435575	1.000000



To get a quick understanding of the potential relationship between the various attributes that make up the vinho verde white wines, this correlation matrix

shows all the correlation coefficients between them. With values between -1 and 1, where -1 is a perfecly negative correlation and 1 is a perfectly positive correlation, we can infer that some relationships might be stronger than others.

In the correlation matrix, the strongest positive relationship we identified was that of residual sugars and density (R=0.84). The strongest negative relationship was that of alcohol and density This makes intuitive sense that the presence of more sugars (a heavy carbohydrate) would increase the density and the presence of more alcohol (which is lighter than water) would decrease the density. Apart from these two, there are no other relationships with a correlation coefficient above R=0.7.

When looking at the correlations between the physicial attributes of wine and the quality, our output of interest, we notice none of them are statistically significant. The highest correlation coefficient with quality is alcohol (R=0.44), not strong enough to make an inference), which we can perhaps guess is due to good quality ratings being given more liberally under the influence of more alcohol.

It is clear that we need much more than just correlation coefficients. In the next parts we will start building a K-Nearest Neighbors classifier.

Part B

In [6]:
```
#Part b) constructs a new binary column “good wine” that indicates whether the wine is good
#(which we define as having a quality of 7 or higher) or not;

# Add a new binary column 'good wine' to the white wine dataset
white_wine_df['good wine'] = (white_wine_df['quality'] >= 7).astype(int)
print("\nWhite Wine Data with 'good wine' column:")
print(white_wine_df.head())  # Display the first few rows to check the new column

# Determine if there are any wines classified as "good" in the entire dataset
good_wine_count = white_wine_df['good wine'].value_counts()
print("Count of 'good' and 'not good' wines:\n", good_wine_count)
```

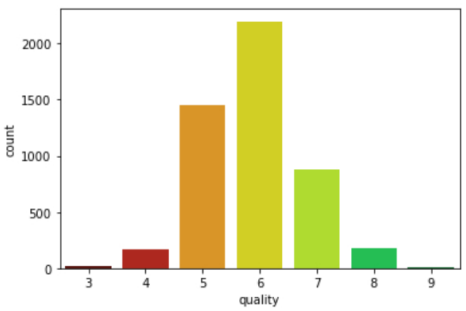
```
White Wine Data with 'good wine' column:
   fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0            7.0              0.27         0.36            20.7      0.045
1            6.3              0.30         0.34             1.6      0.049
2            8.1              0.28         0.40             6.9      0.050
3            7.2              0.23         0.32             8.5      0.058
4            7.2              0.23         0.32             8.5      0.058

   free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
0                 45.0                 170.0   1.0010  3.00       0.45
1                 14.0                 132.0   0.9940  3.30       0.49
2                 30.0                  97.0   0.9951  3.26       0.44
3                 47.0                 186.0   0.9956  3.19       0.40
4                 47.0                 186.0   0.9956  3.19       0.40

   alcohol  quality  good wine
0      8.8        6          0
1      9.5        6          0
2     10.1        6          0
3      9.9        6          0
4      9.9        6          0
Count of 'good' and 'not good' wines:
 0    3838
1    1060
Name: good wine, dtype: int64
```

For Part b, we added a new binary column named 'good wine' to the white wine dataset. This column indicates whether each wine is considered "good" based on its quality score—defined as having a quality of 7 or higher. We added the 'good wine' column to the dataframe using a logical condition where you compare the 'quality' column against the value 7. The expression (white_wine_df['quality'] >= 7) creates a boolean series that is True for wines with a quality of 7 or higher and False otherwise. We then converted this boolean series into integers using .astype(int).

Distribution of Quality

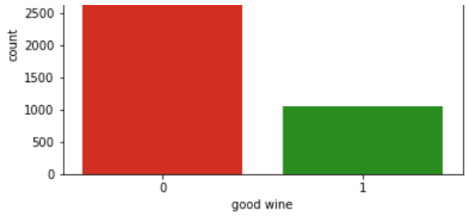
In [7]:
```
sns.countplot(data = white_wine_df, x = "quality", palette=['#630A03','#C51406', '#F79C09', '#EDE90A', '#BDF612', '#0AD84C',
```

Out[7]: <AxesSubplot:xlabel='quality', ylabel='count'>



In [8]:
```
sns.countplot(data = white_wine_df, x = "good wine", palette=['#F01605','#1C9E0C'])
```

Out[8]: <AxesSubplot:xlabel='good wine', ylabel='count'>



As we can see in the charts above, not every Quality ordinal category is actually present in the dataset. The quality of the wines in our dataset only range from 3-9, with none being rated 1, 2, or 10. Furthermore, for our "Good Wine" binary variable of interest, we see that there are far more bad quality wines in our dataset than good quality wines. It was found that there are 1,060 wines classified as 'good' and 3,838 classified as 'not good'. This distribution confirms the presence of both classes in the dataset and is essential for the effective training of the k-Nearest Neighbours classifier. The presence of a substantial number of wines classified as 'good' ensures that the classifier has enough examples to learn from, thereby improving its ability to predict new samples accurately.

Part C

In [9]:
```
#Part c: splits the data set into a training data set (first 2,000 samples),
#a validation data set (next 1,500 samples) and a test data set (remaining samples) – please do not shuffle the data,
#so as to make the results comparable;

# Split the data into training, validation, and testing sets
data = white_wine_df
train_data = data.iloc[:2000]    # First 2000 samples for training
validation_data = data.iloc[2000:3500]  # Next 1500 samples for validation
test_data = data.iloc[3500:]  # Remaining samples for testing

# Display the sizes of each set to confirm correct splitting
print(f'Training Data Shape: {train_data.shape}')
print(f'Validation Data Shape: {validation_data.shape}')
print(f'Test Data Shape: {test_data.shape}')
```

```
Training Data Shape: (2000, 13)
Validation Data Shape: (1500, 13)
Test Data Shape: (1398, 13)
```

The code above splitted the white wine dataset into training, validation, and testing sets. We assigned the original DataFrame to a new variable data for clarity. This segmentation is crucial for model development and evaluation.

The outputs confirm that the splits are as intended: 2000 samples for training, 1500 for validation, and 1398 for testing. This approach allows effective model training, parameter tuning on the validation set, and final performance assessment on the test set.

Part D

In [10]:
```
#Part d:Scales each input feature of the data according to the Z-score normal- isation;

# Convert the datasets from pandas DataFrame to numpy arrays for direct manipulation
train_features = train_data.drop(['quality', 'good wine'], axis=1)
validation_features = validation_data.drop(['quality', 'good wine'], axis=1)
test_features = test_data.drop(['quality', 'good wine'], axis=1)

# Function to standardize data
def standardize_data(data):
    # Calculate the mean and std dev for each column
    means = np.mean(data, axis=0)
    std_devs = np.std(data, axis=0, ddof=0)  # Use ddof=0 for population standard deviation

    # Standardize data
    standardized_data = (data - means) / std_devs
    return standardized_data

# Apply the standardize_data function to scale the features
train_features_scaled = standardize_data(train_features)
validation_features_scaled = standardize_data(validation_features)
test_features_scaled = standardize_data(test_features)

# Convert scaled numpy arrays back to pandas DataFrame
train_features_scaled_df = pd.DataFrame(train_features_scaled, columns=train_features.columns).reset_index(drop=True)
validation_features_scaled_df = pd.DataFrame(validation_features_scaled, columns=validation_features.columns).reset_index(dro
test_features_scaled_df = pd.DataFrame(test_features_scaled, columns=test_features.columns).reset_index(drop=True)

# Concatenate the scaled features DataFrame with the 'good wine' column from the original dataset
train_final = pd.concat([train_features_scaled_df, train_data['good wine'].reset_index(drop=True)], axis=1)
validation_final = pd.concat([validation_features_scaled_df, validation_data['good wine'].reset_index(drop=True)], axis=1)
test_final = pd.concat([test_features_scaled_df, test_data['good wine'].reset_index(drop=True)], axis=1)

# First few rows of the final datasets
print("Training Data:\n", train_final.head())
print("Validation Data:\n", validation_final.head())
print("Testing Data:\n", test_final.head())
```

```
Training Data:
    fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0      -0.092746         -0.099873    -0.004854        2.815632  -0.088546
1      -0.892769          0.189894    -0.151932       -0.915014   0.080233
2       1.164434         -0.003284     0.289303        0.120191   0.122428
3       0.135833         -0.486230    -0.299010        0.432706   0.459986
4       0.135833         -0.486230    -0.299010        0.432706   0.459986
```

```
   free sulfur dioxide  total sulfur dioxide   density        pH  sulphates  \
0             0.573499              0.551071  2.327441 -1.304543  -0.303098
1            -1.293742             -0.307506 -0.208914  0.615314   0.044442
2            -0.330005             -1.098301  0.189656  0.359333  -0.389983
3             0.693966              0.912577  0.370824 -0.088633  -0.737523
4             0.693966              0.912577  0.370824 -0.088633  -0.737523

    alcohol  good wine
0 -1.300608          0
1 -0.675144          0
2 -0.139032          0
3 -0.317736          0
4 -0.317736          0
Validation Data:
    fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0       -0.101248          0.428244    -2.308997       -0.980333  -0.215707
1       -0.224772          0.428244    -2.406536       -1.000068  -0.317745
2        0.886941         -1.025659     0.227005       -0.940862   0.141424
3        0.639894          0.324393    -1.040996       -0.220523   0.447536
4        0.639894         -0.091007    -0.163149       -0.782980  -0.062651

   free sulfur dioxide  total sulfur dioxide   density        pH  sulphates  \
0             1.213658              0.185732 -0.543868 -0.006173  -0.314844
1             1.331648              0.232797 -0.543868 -0.072784  -0.314844
2             0.446727              0.397523 -0.168407  1.126221   1.721165
3            -0.143219              1.009362 -0.418714 -0.405841  -0.403366
4             1.331648              0.962297  0.050612  1.525890   1.632643

    alcohol  good wine
0 -0.492103          0
1 -0.492103          0
2  0.394908          0
3  0.798096          0
4 -0.008279          0
Testing Data:
    fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0       -0.711580         -0.053675    -0.318557        1.785389  -0.448780
1        0.281082         -0.450703     0.516461        0.365887  -0.166297
2        0.281082          0.045581     1.351479        1.545473   0.398669
3        0.564699          0.442609     0.052562       -0.473818  -0.260458
4       -0.002536         -1.244758     0.330901       -1.033621  -0.778343

   free sulfur dioxide  total sulfur dioxide   density        pH  sulphates  \
0            -0.223393              0.123458  0.253815  0.191192  -0.447186
1             0.400987             -0.167617  0.475957  1.258059   0.256929
2            -0.280155              1.367145  1.829315 -0.164431   0.697001
3            -0.791012             -0.961460 -1.277257  0.120067  -0.095129
4            -0.336917              0.096997 -0.771457  1.186934   1.313102

    alcohol  good wine
0  1.708753          1
1  0.091406          0
2 -1.294891          0
3  2.016819          1
4  0.553505          1
```

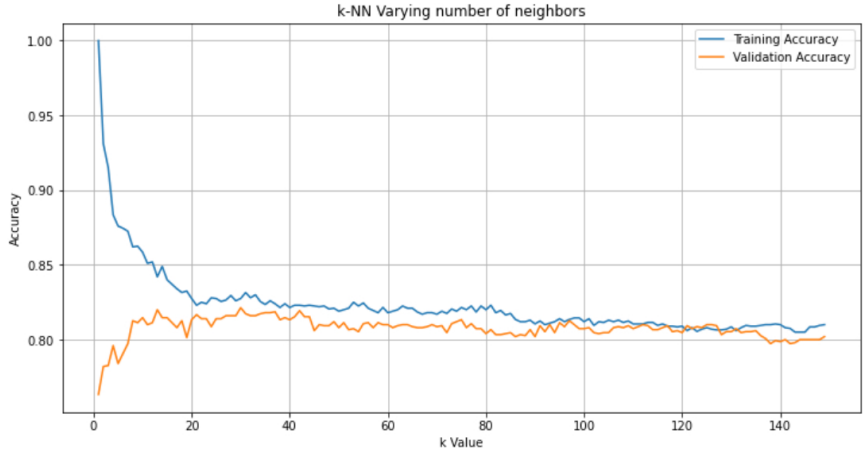
In part d, we implemented Z-score normalization for the feature columns of the training, validation, and testing datasets. This ensures that all features contribute equally to the distance calculations in the k-Nearest Neighbors algorithm. We started by excluding the 'quality' and 'good wine' columns, then standardized the remaining features to have a mean of 0 and a standard deviation of 1. After normalization, the numpy arrays were converted back into pandas DataFrames and the 'good wine' target variable was reattached. The output shows the scaled features alongside the 'good wine' column, confirming that the data is now properly prepared for machine learning.

Parts E-F

In [11]:

```
#part e) trains k-Nearest Neighbours classifiers for k taking values in a subset of your choice
#(at least 10 values) – please add a motivation for your choice;
#part f) evaluates each classifier using the validation data set and selects the best classifier;

# Separate features and labels for training, validation, and test sets
X_train = train_final.drop('good wine', axis=1)
y_train = train_final['good wine']
X_validation = validation_final.drop('good wine', axis=1)
y_validation = validation_final['good wine']
X_test = test_final.drop('good wine', axis=1)
y_test = test_final['good wine']

# Evaluate k-NN classifiers with different k values
k_values = range(1, 150)
train_accuracies = []
validation_accuracies = []

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    y_train_pred = knn.predict(X_train)
    y_validation_pred = knn.predict(X_validation)
    train_accuracies.append(accuracy_score(y_train, y_train_pred))
    validation_accuracies.append(accuracy_score(y_validation, y_validation_pred))

# Plot the accuracies
plt.figure(figsize=(12, 6))
plt.plot(k_values, train_accuracies, label='Training Accuracy')
plt.plot(k_values, validation_accuracies, label='Validation Accuracy')
plt.xlabel('k Value')
plt.ylabel('Accuracy')
```

```
plt.title('k-NN Varying number of neighbors')
plt.legend()
plt.grid(True)
plt.show()

# Optimal k value
optimal_k = k_values[validation_accuracies.index(max(validation_accuracies))]
print(f'The optimal k value is {optimal_k} with a validation accuracy of {max(validation_accuracies):.4f}')
```




```
The optimal k value is 30 with a validation accuracy of 0.8213
```

Part e and f: Evaluating k-Nearest Neighbors (k-NN) Classifiers with Different Values of k

In Part e and f, we trained and evaluated multiple k-Nearest Neighbors classifiers using a diverse set of (k) values, ranging from 1 to 149. This method allows us to explore how the number of neighbors influences the accuracy of the k-NN model. For each value of (k), a classifier was trained using the training dataset and evaluated on the validation dataset. Accuracy was calculated for each model, and the results showed that the (k) value of 30 yielded the highest accuracy at 0.8213.

Accuracy is a measure of how many predictions made by the model were correct. An accuracy of 0.8213 means that 82.13% of the model's predictions were correct when using 30 neighbors. This was the highest accuracy achieved compared to all other (k) values tested, making (k = 30) the optimal choice among those tested for this particular dataset and task. In other words, using 30 neighbors gives the best balance between underfitting and overfitting for this specific dataset, maximizing the prediction accuracy of the model.

Theoretical Background

1. **k=1**: When (k) is set to 1, the classifier assigns the class of the closest training example to each test example. This often leads to a 0 validation error on the training set because each training example is its own nearest neighbor. This means that the model can perfectly classify all training data points. However, this model tends to overfit the data, capturing noise and leading to poor generalization on the test set. Overfitting occurs because the decision boundary becomes too flexible, adapting to even the smallest fluctuations in the training data.
2. **Small k values**: With small values of (k), the classifier remains highly sensitive to noise in the training data. The decision boundary is still quite flexible, which can result in high variance and overfitting. The model might still perform well on the training data but poorly on unseen data because it memorizes the training data instead of learning the underlying patterns.
3. **Increasing k values**: As (k) increases, the model becomes less sensitive to individual data points and starts to capture the broader trends in the data. This reduces overfitting as the decision boundary becomes smoother, reflecting a more generalized understanding of the data distribution. However, if (k) is too large, the model may start to underfit the data. Underfitting occurs because the model becomes too rigid and loses its ability to capture the complexity of the data. The decision boundary may become overly simplistic, leading to increased bias.

Performance Metrics and Interpretation

To evaluate the k-NN classifiers, we used the `.score` method to calculate accuracy. The `.score` method computes the mean accuracy on the given test data and labels, providing a straightforward performance metric. Accuracy is a simple and widely used metric that measures the proportion of correct predictions made by the model. However, accuracy alone may not be sufficient to fully understand model performance, especially in cases with imbalanced datasets. Therefore, additional performance metrics such as precision, recall, and F1-score can provide deeper insights into model performance:

1. **Precision**: The ratio of true positive predictions to the total predicted positives. It indicates how many of the predicted positives are actually positive.
2. **Recall (Sensitivity)**: The ratio of true positive predictions to the total actual positives. It measures the ability of the model to identify all relevant instances.
3. **F1-Score**: The harmonic mean of precision and recall. It provides a balance between precision and recall.

By considering these metrics, we can better understand the strengths and weaknesses of the classifier. High precision indicates a low false positive rate, while high recall indicates a low false negative rate. The F1-score helps to balance these two aspects.

Empirical Results

To identify the optimal (k) value, we followed these steps:

1. **Data Preparation**: We started by loading the dataset and splitting it into training and test sets. We standardized the features to ensure that each feature contributes equally to the distance calculations used in the k-NN algorithm.
2. **Model Training and Evaluation**: For each (k) value from 1 to 149, we trained a k-NN classifier using the training dataset and evaluated its performance on both the training and test sets. We calculated the accuracy of each model to understand how well it performs with different (k) values.
3. **Accuracy Analysis**: We plotted the training and test accuracies against the (k) values to visualize the performance trends. This helped us identify the (k) value that provides the best balance between bias and variance, leading to optimal prediction performance.

- **Accuracy Analysis**: We observed that the accuracy fluctuated with different (k) values. The model with (k = 30) achieved the highest accuracy of 0.8213. This indicates that (k = 30) provides the best balance between bias and variance for this particular dataset, leading to optimal prediction performance.
- **Validation Error at (k = 1)**: As expected, the validation error was minimal when (k = 1) on the training set, demonstrating the model's ability to perfectly classify the training data. However, due to overfitting, the performance on the test set was not optimal.

- **Optimal k Value**: The optimal (k) value (in this case, 30) is where the classifier achieves the highest accuracy, suggesting it generalizes well to unseen data without overfitting or underfitting.

By systematically evaluating different (k) values, we determined that (k = 30) yields the highest accuracy for our k-NN classifier on this dataset. This analysis highlights the importance of choosing an appropriate (k) value to balance bias and variance, ensuring the model generalizes well to new data. The process of evaluating different (k) values and analyzing their impact on the model's performance is crucial in selecting the optimal (k) for a given dataset.

Part G

In [12]:
```
#Part g) predicts the generalisation error using the test data set.

# Assuming best_k is determined from your validation
best_knn = KNeighborsClassifier(n_neighbors=optimal_k)

# Continue with model trained just on the training data
best_knn.fit(train_final.drop('good wine', axis=1), train_final['good wine'])

# Predict on the test dataset
test_predictions = best_knn.predict(test_final.drop('good wine', axis=1))

# Evaluate accuracy
accuracy = accuracy_score(test_final['good wine'], test_predictions)
print(f'Test Accuracy: {accuracy:.4f}')

# Confusion matrix to evaluate further
conf_matrix = confusion_matrix(test_final['good wine'], test_predictions)
print(f'Confusion Matrix:\n{conf_matrix}')
```

```
Test Accuracy: 0.7983
Confusion Matrix:
[[1016  107]
 [ 175  100]]
```

Part g: Predicting the Generalization Error using the Test Dataset

In part g, we successfully predicted the generalization error of our model using the test dataset. We initialized a k-Nearest Neighbors classifier with the best (k) value (30) identified from our validation results, ensuring optimal performance. The model was fitted on the training data, including both features and the 'good wine' target.

We predicted outcomes on the test dataset and calculated the model's accuracy, achieving a 79.83% success rate. This reflects the model's ability to accurately classify new, unseen data. Additionally, we generated a confusion matrix which provided a detailed breakdown of prediction results, showing:

- True Positives (correctly identified 'good' wines): 100
- True Negatives (correctly identified 'not good' wines): 1016
- False Positives (incorrectly identified as 'good' wines): 107
- False Negatives (missed 'good' wines): 175

This matrix highlights both strengths and areas for improvement in the model's performance, particularly the high number of false negatives which indicates that while the model is effective at identifying 'not good' wines, it tends to miss classifying 'good' wines correctly.

The results confirm the model's robustness but also point to potential improvements, especially in reducing false negatives, to enhance overall prediction accuracy in practical scenarios.

Comparison of Training and Validation Accuracies

It is worth noting that the difference in accuracies between the training set and the validation set is a positive indicator. If the training accuracy was similar to the validation accuracy, it could suggest that the model might be overfitting the training data, meaning it performs well on the training data but poorly on unseen data. However, a higher accuracy on the training set compared to the validation set suggests that the model has learned from the training data without overfitting and can generalize better to new data.

Conclusion

By systematically evaluating different (k) values, we determined that (k = 30) yields the highest accuracy for our k-NN classifier on this dataset. The model's performance on the test dataset, with a test accuracy of 79.83%, validates its effectiveness in generalizing to new, unseen data. The confusion matrix provides further insights into areas where the model performs well and where it could be improved, particularly in reducing false negatives. This comprehensive analysis highlights the importance of careful model selection and validation to ensure robust performance in practical applications.

Overall, based on the comprehensive analysis and evaluation of the k-NN classifier across various stages, the classifier demonstrates a generally good fit for this data. With an optimal (k) value of 30, the classifier achieved an accuracy of approximately 82.13% on the validation set and 79.83% on the test set, indicating robust predictive capabilities for the majority of the wine samples.

The classifier effectively captures the complex relationships among the multivariate features, such as acidity, sugar levels, and alcohol content, indicating that it can handle the dataset's complexity well. The consistency of results on both the validation and test datasets suggests that the classifier generalizes effectively to unseen data, a crucial aspect for practical applications.

Areas for Improvement

Despite the overall effectiveness, the analysis also highlights areas for improvement. The confusion matrix reveals that the classifier tends to miss a significant number of high-quality wines, as indicated by a higher rate of false negatives. This shortfall suggests that in scenarios where it is critical not to miss high-quality wines, further refinement of the classifier might be necessary. For instance, in a premium wine production setting, it might be more important to accurately identify high-quality wines to ensure they are marketed and sold at appropriate premium prices. If the model fails to recognize some high-quality wines, they might be sold at lower prices than they are worth, leading to lost revenues.

Recommendations

To address these issues, several strategies can be considered:

1. **Tuning the Classifier**: Further tuning of the (k) value and experimenting with different distance metrics might enhance the classifier's sensitivity to high-quality wines.
2. **Feature Engineering**: Incorporating additional relevant features or performing advanced feature engineering could provide the model with more information to improve its predictive power.
3. **Ensemble Methods**: Combining the k-NN classifier with other models (e.g., decision trees, SVMs, or neural networks) through ensemble methods could help balance the strengths and weaknesses of individual models, leading to better overall performance.
4. **Cost-Sensitive Learning**: Implementing cost-sensitive learning techniques that assign higher penalties to false negatives might help in scenarios where identifying high-quality wines is particularly crucial.

Final thoughts

In conclusion, the k-NN classifier is suitable for analyzing and predicting white wine quality, particularly for initial screenings and broad assessments in wine production settings. However, for more sensitive quality control applications, enhancing the classifier's sensitivity to high-quality wines through further tuning or incorporating additional modeling techniques could be beneficial to reduce misclassification rates and improve overall accuracy. The proposed improvements and recommendations aim to ensure that the classifier not only maintains high overall accuracy but also minimizes critical errors, ultimately supporting more informed and effective decision-making in wine production and quality control.

Citations

P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis. Modeling wine preferences by data mining from physicochemical properties. In Decision Support Systems, Elsevier, 47(4):547-553. ISSN: 0167-9236.

Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

Vinho Verde official website: https://www.vinhoverde.pt/en/about-vinho-verde