1º Deliverable: Machine Learning
RED WINE QUALITY

UNIVERSIDAD DE MONTEVIDEO

Team VIII: Ariel Mordetzki and Mateo Stipaničić
Prof. Sebastián García Parra
24th September 2022

Table of Contents

1 Dataset Analysis

> **In-depth analysis of given dataset**: In this section, we will analyze each feature's statistical properties, as well as the relevant relationships between them.

1.1 Import libraries and dataset

In [1]:
```
# Libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import sklearn
import seaborn as sns

# Dataset
```

1.2 Data exploration and understanding

1.2.1 Overall analysis of dataset

In [2]:
```
# Show dataset format with first sample wines
```

Out[2]:

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

In [3]:
```
# Print dataset size
```

Out[3]:
```
(1591, 12)
```

The dataset is composed of 11 features and 1 target ('quality'), with a sample size of 1591 wines.

In [4]:
```
# Consider data type
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1591 entries, 0 to 1590
Data columns (total 12 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   fixed acidity         1591 non-null   float64
 1   volatile acidity      1591 non-null   float64
 2   citric acid           1591 non-null   float64
 3   residual sugar        1591 non-null   float64
 4   chlorides             1591 non-null   float64
 5   free sulfur dioxide   1591 non-null   float64
 6   total sulfur dioxide  1591 non-null   float64
 7   density               1591 non-null   float64
 8   pH                    1591 non-null   float64
 9   sulphates             1591 non-null   float64
 10  alcohol               1591 non-null   float64
 11  quality               1591 non-null   int64  
dtypes: float64(11), int64(1)
memory usage: 149.3 KB
```

In [5]:
```
# Check null values
```

Out[5]:
```
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
```

Confirming there are no null values, which might indicate good data quality (not in terms of bias, rather in a sense of completeness).

In [6]:
```
# Check unique values for target variable
```

```
[ 3  4  5  6  7  8 14 15 16]
```

Unique values analysis shows no sample wines of quality 9, for example. Also, certain wines were evaluated with quality [14,15,16], which certainly raises concerns on the data quality.

In [7]:
```
# Filtering outliers due to unsenseful data
wines_filter1 = wines_df[wines_df['quality'] <= 10]

# Instead, we could have chosen to filter by probability
# wines_filter1 = wines_df['quality'].quantile(0.95)

# After filtering:
```

```
[3 4 5 6 7 8]
```

Wine quality measurement ranges from 0 to 10. Unique values confirmation provides input on data that is out of that range, thus, we decide to treat these cases as **data errors**, regarding them as outliers. A wine quality beyond 10 does not make any business-wise sense. We work now with variable **wines_filter1**.

In [8]:
```
# Break down dataset in percentiles
```

Out[8]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total d
count	1588.000000	1588.000000	1588.000000	1588.000000	1588.000000	1588.000000	1588.0
mean	8.324559	0.527365	0.271770	2.539830	0.087509	15.867758	46.4
std	1.741589	0.178995	0.194921	1.413155	0.047148	10.434004	32.8
min	4.600000	0.120000	0.000000	0.900000	0.012000	1.000000	6.0
25%	7.100000	0.390000	0.090000	1.900000	0.070000	7.000000	22.0
50%	7.900000	0.520000	0.260000	2.200000	0.079000	14.000000	38.0
75%	9.200000	0.640000	0.422500	2.600000	0.090000	21.000000	62.0
max	15.900000	1.580000	1.000000	15.500000	0.611000	72.000000	289.0

This description gives deeper statistical insight of the data. Firstly, percentile data (especially, 50% percentile) is nice to have in mind. As well, minimum and maximum values of each feature.

A more exhaustive analysis could consist in calculating the percentual standard deviation as a measure of data dispersion.

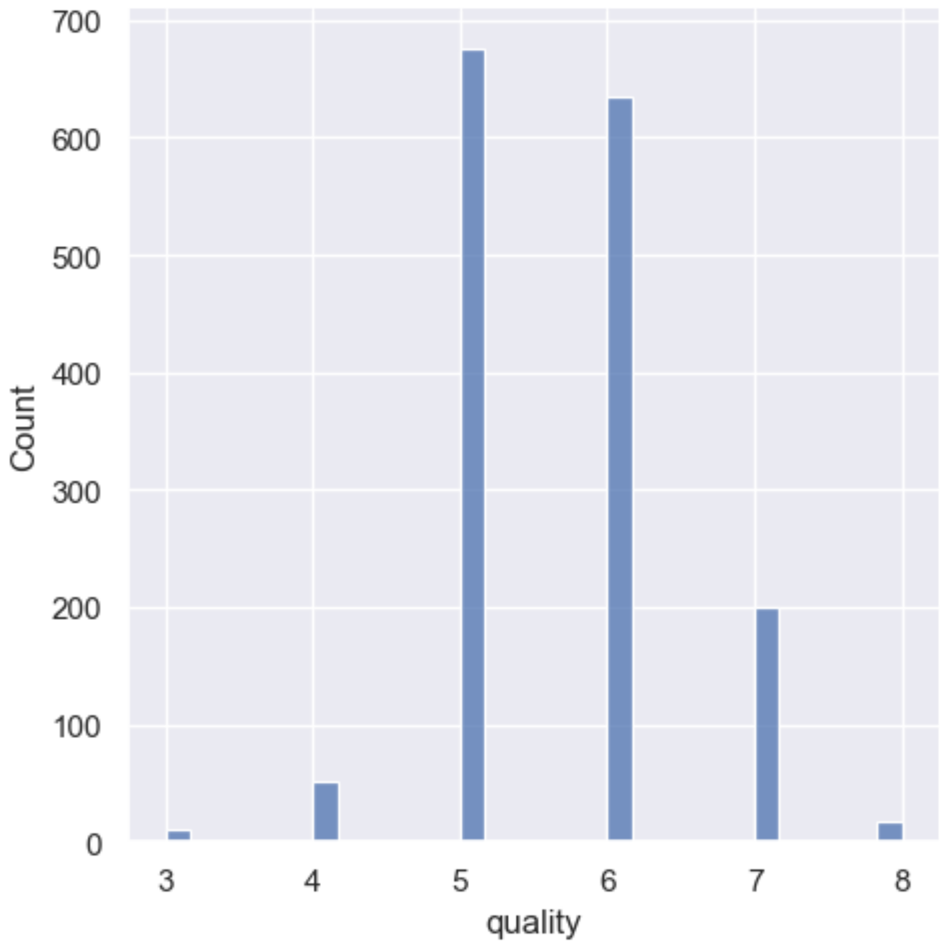
In [9]:
```
# Target variable count
```

Out[9]:
```
5    676
6    634
7    199
4     51
8     18
3     10
Name: quality, dtype: int64
```

Dataset seems to be **unbalanced**: the vast majority of samples are of quality 5 and 6. Also, there are no samples of, for example, a wine of quality 2, 9, even 10.

In [10]:
```
# Target variable distribution
sns.set(rc={'figure.figsize':(11,9)}) # sets image size
```

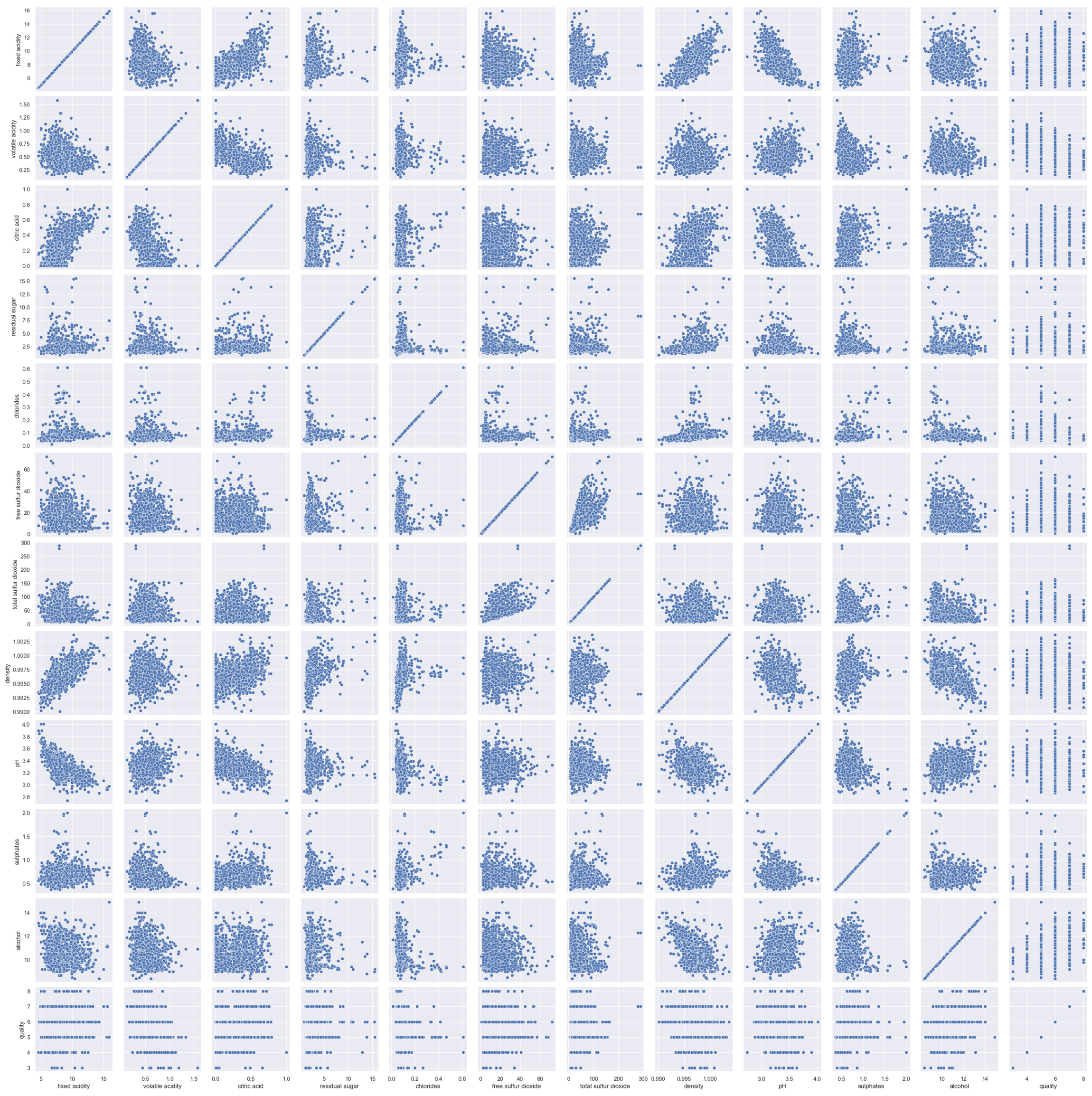
Out[10]:
```
<seaborn.axisgrid.FacetGrid at 0x195fa559700>
```



1.2.2 Feature review

In [11]:
```
# Presenting  relationships between variables
```

Out[11]:
```
<seaborn.axisgrid.PairGrid at 0x195fa5b8910>
```

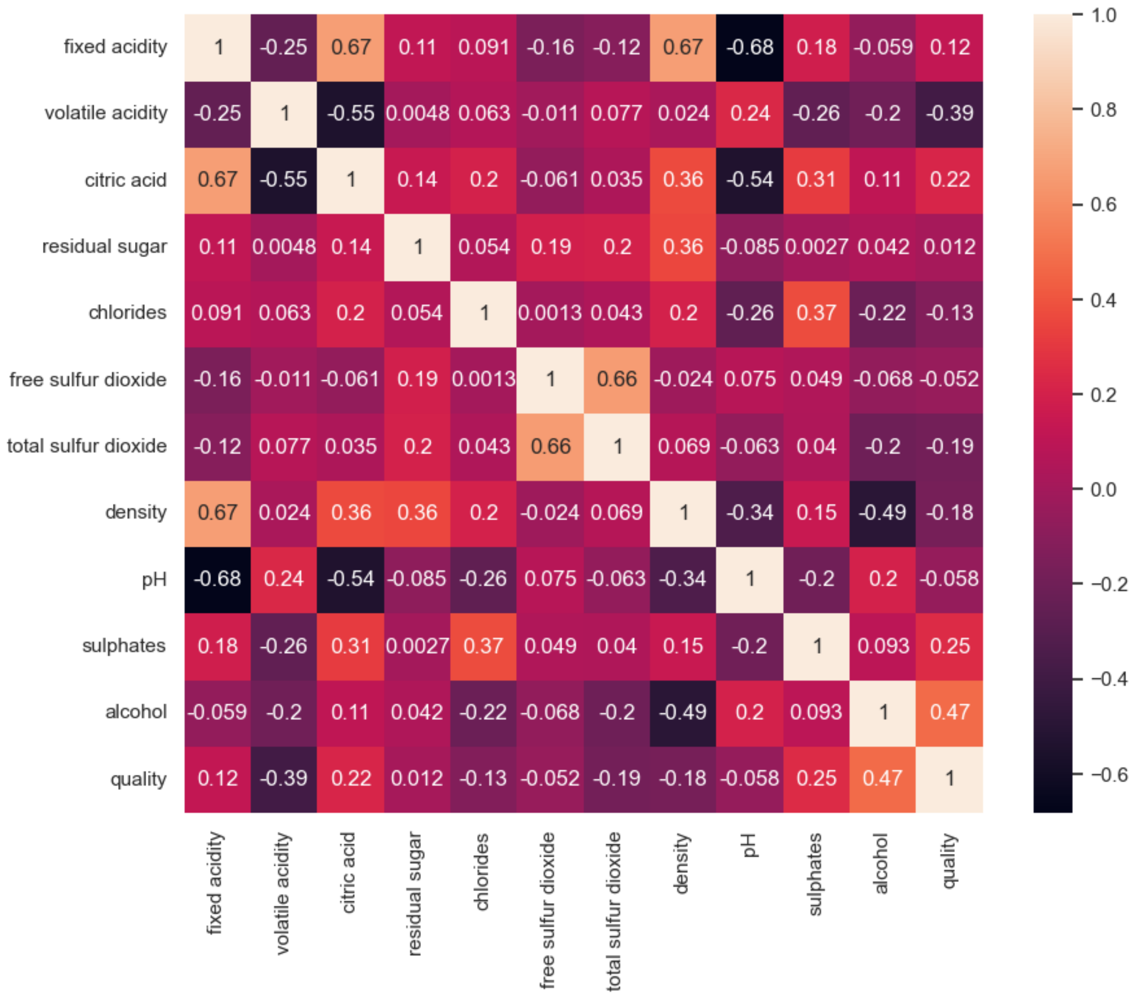


In diagonals the correlation is 1 to 1 (evident, since it is pairing x with x). Certain other pairs call to attention, such as **density** and **fixed acidity**. Looking at **quality** vs. **alcohol** we can observe a light tendency of increase in quality in greater alcohol numbers. The relationship with **quality** of each pair seems odd, but this is due to the fact that it is the only discrete variable (datatype = int).

Nevertheless, this plot can only hint at plausible correlation between variables, instead of giving a definite answer on the relationships among features: thus, we need to plot the correlation matrix to confirm our conjectures.

In [12]:
```python
# Plotting correlation matrix
f, ax = plt.subplots(figsize=(10, 8))
correlation = wines_filter1.corr()
```

Out[12]:
```
<AxesSubplot:>
```



Firstly, we must say that our guesses were partially correct: **alcohol** is the most correlated feature with the target variable and **density** and **fixed acidity** are intertwined. But, they are not enough.

We keep an eye out for two cases:

- Features strongly correlated between each other.
- Features weakly correlated with the target.

We thus check first for features weakly correlated with the target variable and decide whether to remove them in accordance to their correlation with other features (if it is correlated to other feature, then that first feature does not add relevance: we freely remove it). Therefore, from strongly correlated features, we keep those which correlate with the target in a senseful way.

In addition, it is clear to see the link between the correlation matrix and the previous plot (correlation between variables).

In [13]:
```python
# Removing unwanted features
wines_filter2 = wines_filter1.drop(['pH', 'free sulfur dioxide', 'residual
```

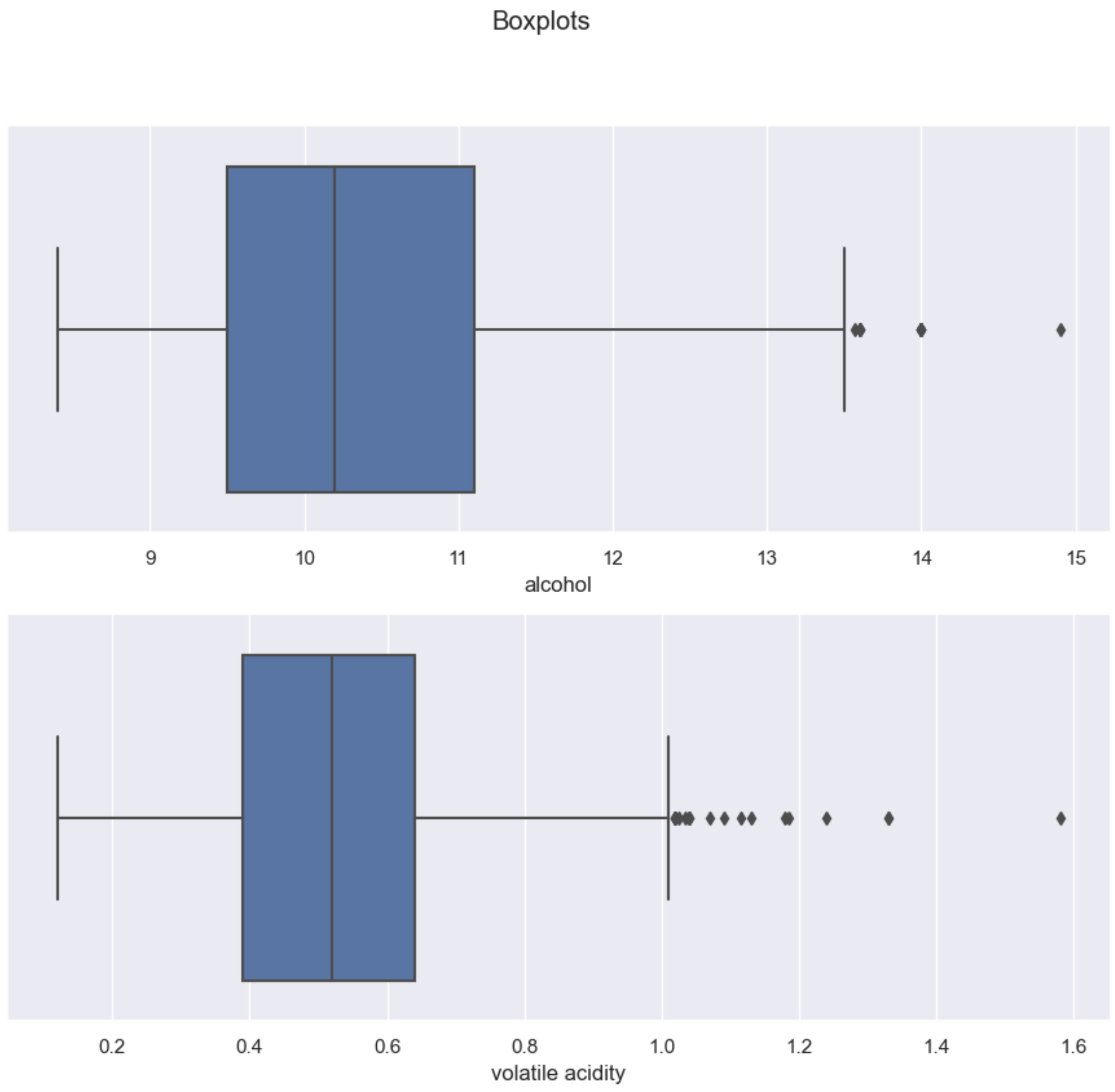
We remove **pH** because it is weakly correlated with quality (-0.058) and we do so safely, since **pH** is strongly correlated with **fixed acidity** (-0.68), and we chose not to remove this latter feature because it is sufficiently correlated with the target variable (0.12). Thus, we are certain we are not losing valuable relationships/information between data.

Likewise, we will not consider in this problem **free sulfur dioxide** based in analogus reasoning.

Although rather weakly correlated with **density**, **residual sugar** is also removed due to its low correlation with the target variable (0.012).

We considered dropping **density** or **fixed acidity** due to their strong correlation between each other (0.67); however, since **pH** was already removed from the feature outline, we decided to keep both in order to ensure sufficient dimensions in the problem.

In [15]:
```
# Relevant features' boxplots for outliers scrutiny
fig, (ax1, ax2) = plt.subplots(2)
fig.suptitle('Boxplots')
sns.boxplot(wines_filter2['alcohol'], ax=ax1)
sns.boxplot(wines_filter2['volatile acidity'], ax=ax2)

import warnings
```



In order to discuss whether certain samples should be left behind, we make boxplots for the features most strongly correlated to the target variable, meaning, **alcohol** and **volatile acidity**.

In volatile acidity's boxplot, we can see a greater amount of outliers: thus, we consider it necessary to keep them all. Instead, in alcohol's boxplot, we deem appropriate to remove the farthest point, since it is a unique sample that is a statistical anomaly.

A model should take statistical anomalies into account: this is why we leave volatile acidity's outliers and a few of alcohol's. However, this does not mean that any anomaly should be welcomed.

In [16]:
```
# Removing more outliers
```

1.3 Dataset discussion

In order to analyze the dataset's properties, we will frame the discussion into categories:

- **Volume:** The question on whether the dataset is sufficient or not for an accurate prediction should be put into context of the different models' use: if the use will be for a single wineyard, for example, then it can be said to be enough. However, out of this scenario, for a global application, this dataset is short in volume.
- **Velocity:** This aspect is not of interest in this deliverable (we are **not** getting data faster than we can process it).
- **Variety:** This relevant aspect of the dataset turned out to be less than desired. **The dataset is unbalanced**. This proposition is clear when we see the wine quality's histogram and counts (no wines of 9 or 10, or 1 or 2).
- **Veracity:** Although some points had to be removed due to their lack of business sense, all in all the dataset can be considered as true and reliable. However, we lack credentials and tools to assess the truthfulness of values of fixed acidity, pH, and similar features.
- **Value:** The business value of the dataset can raise the question of the necessity of a oenologist in the wineyard.

2 Linear Regression

> The following is a linear regression model as to **predict wine quality** in basis of a given set of features.

2.1 Data preparation

> This division into testing and training will be used for the entire modelling, ensuring we are always measuring with respect to the same data (in sight of a proper scientific method).

In [17]:
```
# Divide dataset into training and testing samples
from sklearn.model_selection import train_test_split
x = wines_filter3.loc[:,wines_filter3.columns != 'quality'] # features
y = wines_filter3 ['quality'] # target variable
```

2.2 Model definition

In [18]:
```
# Importing model
from sklearn.linear_model import LinearRegression
from sklearn import metrics

# Creating object "model"
model = LinearRegression()

# Feed training dataset into model
model.fit(x_train, y_train)

# Result of the model
coef = pd.DataFrame(model.coef_, x.columns)
coef.columns = ['LR coeffecients:']
```

Out[18]:

	LR coeffecients:
fixed acidity	0.060513
volatile acidity	-1.316590
citric acid	-0.406922
chlorides	-1.615948
total sulfur dioxide	-0.001295
density	-23.886747
sulphates	0.929126
alcohol	0.275331

This means that our result is a **hiperplane of formula**:

$$Quality = f(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) = 0.1x_1 - 1.3x_2 - 0.4x_3 - 1.6x_4 - 0.0x_5 -$$

- x_1 = fixed acidity
- x_2 = volatile acidity
- x_3 = citric acid
- x_4 = chlorides
- x_5 = total sulfur dioxide
- x_6 = density
- x_7 = sulphates
- x_8 = alcohol

Note: this hiperplane formula is not dynamic (if training dataset re-shuffled using other random state, the hiperplane will change).

It should be noted that the parameters of the model, meaning, the coefficients of the linear regression, are fairly balanced, if it not were for density, that is roughly 23 times the size of the others.

At a first glance, this raised a warning on our heads: how could it be that a feature that was not so strongly correlated to the target variable, result in such high coefficient? After looking back at the percentiles and minimum and maximum values for each feature, we concluded that it is relatively logical to have such coefficient, since density has a lesser order of magnitude, than, for example, fixed acidity, and total sulfur dioxide.

We believe an analysis of the size of the coefficients is only appropriate when comparing features with, for example, the same units of measurement: if not, we would be comparing "apples with potatos".

2.3 Error analysis & metrics

In [19]:
```
# Test set prediction
y_pred = model.predict(x_test)

# Root mean squared error from testing dataset
test_rms = (metrics.mean_squared_error(y_pred, y_test))**0.5
```

```
Root mean quadratic error (testing):
0.6866095406412581
```

We first use the RMSE to sense an overall mean of the predictive error.

This suggests that, for example, a prediction of wine quality 6, the mean error is 6 ± 0.69.

As it is mean, it implies there are smaller errors and bigger. It can be said that, as a first measure of the model, it indicates good stability of the model: using integers as wine quality, $Prediction \pm 1$.

In [20]:
```
# Root mean squared error from training dataset
train_pred = model.predict(x_train) # prediction on the train test, only fo
```

```
Root mean quadratic error (training):
0.40528705439075785
```

Useful to compare RMSE on Testing and Training: The RMSE on training is less than the RMSE on testing, suggesting that the model does not adjust appropriately to new data. The model can successfully adapt to the data it already knows, but once it is confronted to new data, it worsens.

In [21]:
```
# R-squared
from sklearn import metrics
```

```
0.31298771556864347
```

R-squared gives another perspective of the fitness of the model to the testing data: the value of the metric is too low.

In [22]:
```
# Displaying predictions
predicted_quality = np.round(y_pred)
```

```
[5. 6. 6. 6. 5. 6. 6. 7. 7. 5. 6. 5. 5. 6. 6. 5. 6. 6. 6. 5. 6. 6. 5. 5.
 5. 5. 6. 5. 5. 6. 5. 6. 5. 5. 6. 6. 5. 6. 5. 7. 6. 6. 5. 5. 6. 6. 5. 5.
 5. 6. 5. 5. 5. 6. 5. 7. 6. 5. 6. 6. 5. 6. 7. 6. 6. 5. 6. 5. 5. 6. 5. 5.
 6. 6. 5. 5. 5. 5. 5. 5. 5. 5. 7. 5. 6. 6. 6. 5. 5. 7. 5. 5. 5. 6. 6. 6.
 6. 6. 6. 6. 5. 6. 6. 5. 5. 6. 6. 5. 5. 6. 6. 5. 6. 5. 5. 7. 6. 6. 6. 5.
 6. 6. 6. 6. 6. 5. 5. 5. 6. 5. 5. 7. 6. 5. 6. 5. 6. 6. 5. 6. 6. 6. 5. 7.
 6. 5. 5. 5. 6. 6. 5. 6. 6. 6. 6. 6. 6. 6. 5. 6. 5. 5. 5. 5. 5. 6. 7. 5.
 6. 6. 5. 5. 5. 6. 6. 6. 7. 5. 6. 6. 6. 5. 5. 5. 5. 5. 6. 5. 6. 6. 6. 6.
 6. 5. 6. 5. 6. 5. 6. 6. 6. 5. 6. 6. 5. 7. 6. 6. 6. 5. 5. 7. 6. 5. 6. 5.
 5. 6. 5. 5. 5. 6. 5. 5. 5. 7. 5. 5. 6. 6. 5. 5. 5. 5. 6. 6. 5. 6. 6. 5.
 5. 5. 5. 6. 7. 5. 6. 5. 5. 5. 5. 5. 6. 5. 5. 6. 5. 6. 6. 6. 6. 5. 6. 5.
 6. 6. 5. 5. 6. 5. 6. 6. 6. 6. 5. 6. 5. 5. 5. 6. 6. 6. 6. 6. 6. 6. 6. 5.
 6. 6. 6. 7. 6. 5. 5. 5. 7. 5. 5. 6. 6. 5. 5. 5. 5. 5. 6. 6. 5. 6. 5. 7.
 6. 5. 6. 6. 5. 6.]
```

This print is vital to understand the limitations of the model, since it becomes clear how **the predictions are only 5,6 or 7**. More on this in the model discussion, but this suggests **high degree of unbalance in the dataset**.

The decision to use round numbers as the final prediction of the Linear Regression should not be left unnoticed. Although the whole idea of a regression is to produce values from the linear function, we considered inappropriate to compare the model (continuous) to the testing dataset (discrete) without this decision of rounding. Instead of round, it could also have been np.ceil or np.floor, adding yet another parameter to this discrete LR.

In [23]:
```
# Calculating if errors tend to over-estimate or under-estimate
y_testarr = y_test.tolist()
errors_LR = predicted_quality - y_testarr # difference between predicted an
count_P = 0
count_N = 0
equals = 0

for sample in errors_LR:
    if sample > 0:
        count_P += 1
    elif sample < 0:
        count_N += 1
    else:
        equals += 1

print(f'Frequency of over-estimation: {round(count_P/len(y_testarr),4)}')
print(f'Frequency of predictions: {round(equals/len(y_testarr), 4)}')
```

```
Frequency of over-estimation: 0.1981
Frequency of predictions: 0.5723
Frequency of under-estimation: 0.2296
```

2.4 Model discussion

2.4.1 Results

- The display of predictions immediately shows the tendency to predict medium-tier

qualities. Note that in percentile table, 50% percentile **quality** is equal to 75% percentile **quality**: ***thus, we affirm the dataset is unbalanced***, producing a biased regression with central tendency.
- Predictive data dispersion is low due to low RMSE, which seemed to be advantageous. But this could mean either the model is correctly adjusting to new data, or that even predictive data is unbalanced, yet another symptom of a **lack of data variety**.
- In total, the model was **correct 57% of the time** with the testing data: this is not enough. The proclivity to under-estimate the prediction is slightly greater than to over-estimate it (23% of the time it). This discussion in a classifier method is more interesting, since we are prone to choose on whether we would like more False Positives or False Negatives. In the case of wines, one might say we would desire a greater amount of under-estimation than over-estimation (better to surprise than to disappoint).

2.4.2 Potential improvements

As to achieve better results, we suggest, firstly, a **dataset expansion** as to cover a wider amount of wine qualities, looking to correct the bias in the dataset. A stratified dataset could also strengthen this, but the problem is not on the **data dispersion** (how many 1s or 5s there are) but strictly on **data variety**, since there are simply no samples of 1s or 10s or 9s. The problem of **dataset unbalance**, according to our research, could be improved through the use of *class weights* (even though this model was linear regression, the discrete values of quality could have been considered as classes).

With a greater time availability, some trials could be performed as to explore more ways to make the model more robust: **removing more outliers**, testing whether **feature selection** influenced heavily on the model or using a **LOOCV** method of validation (considering that dataset needs to be expanded) instead of a test split.

Lastly, considering that wine quality is a **qualitative feature that is quantitized**, it seems odd to make a linear regression since it does not contemplate this necessary misstep: a LR model returns exact qualities, when the notion of quality is *per se* fuzzy and indefinite. This is why a **classifier model** seems more accurate, since it broadens the notion of quality; of course, this implies a more laxer demand on the model (we expect less specificity in a classifier than in a regression).

---

3 Classification

3.1 Further data treatment

In order to implement any classification methods, we must add an *etiquette* column to the dataframe, that will act as a *qualitative* variable. This new variable will strongly depend on the "quality" feature.

For instance, we shall map the quality scores as follows, according to a chosen **threshold**:

- quality ≤ 6 is *BAD*

- quality > 6 is *GOOD*

In [24]:
```
# Creation of new target label
wines_filter3['quality'] = wines_filter3['quality'].apply(lambda x: 'bad' i

# Print dataframe
```

```
C:\Users\Usuario\AppData\Local\Temp\ipykernel_6560\2659909878.py:2: Settin
gWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-doc
s/stable/user_guide/indexing.html#returning-a-view-versus-a-copy (https://
pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-
view-versus-a-copy)
  wines_filter3['quality'] = wines_filter3['quality'].apply(lambda x: 'bad
' if x < 6.1 else 'good')
```

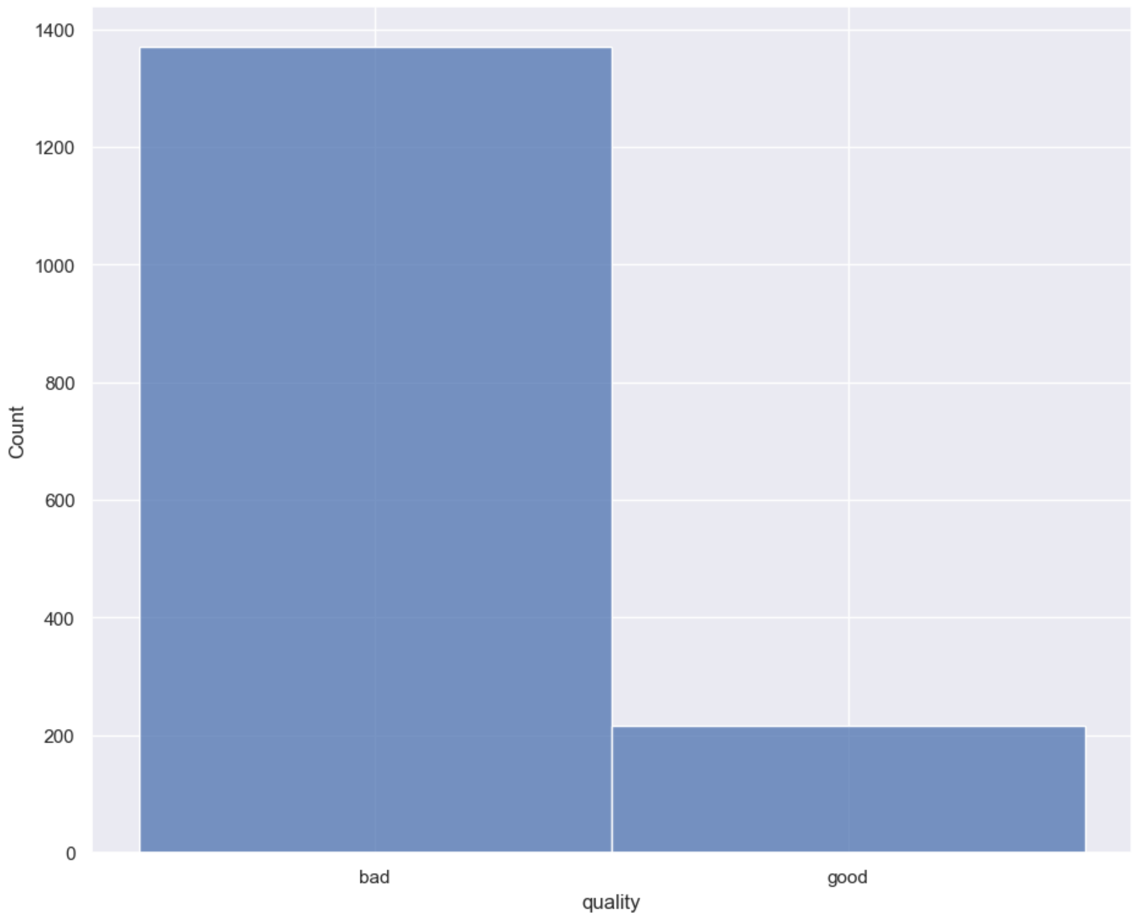
Out[24]:

	fixed acidity	volatile acidity	citric acid	chlorides	total sulfur dioxide	density	sulphates	alcohol	quality
0	7.4	0.70	0.00	0.076	34.0	0.9978	0.56	9.4	bad
1	7.8	0.88	0.00	0.098	67.0	0.9968	0.68	9.8	bad
2	7.8	0.76	0.04	0.092	54.0	0.9970	0.65	9.8	bad
3	11.2	0.28	0.56	0.075	60.0	0.9980	0.58	9.8	bad
4	7.4	0.70	0.00	0.076	34.0	0.9978	0.56	9.4	bad

In [25]:
```
# Print counts and plot histogram
print(wines_filter3['quality'].value_counts())
```

```
bad     1370
good     217
Name: quality, dtype: int64
```

Out[25]:
```
<AxesSubplot:xlabel='quality', ylabel='Count'>
```



Our threshold definition results, again, in an **imbalanced dataset in regards to the target variable**.

In [26]:
```
# Redefining y variable into classes
y = wines_filter3['quality'] # target variable

# Re-shuffling the data with our new y variable using the same 80-20 rule ar
```

3.2 Classification models: Definition

> In the following section, we create model objects and run the models of **Logistics Regression** and **Naive Bayes** as the two methods of classification to explore.

3.2.1 Logistic regression: Model definition

In [27]:
```python
# Import model
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix, Confus

# Creation of an object "model"
model_log = LogisticRegression()

# Feeding training set into model
model_log.fit(x_train, y_train)

# Printing results
y_pred_log = model_log.predict(x_test)
print(f"Number of bad wines using Logistic Regression: {np.sum(y_pred_log ==
print(f"Number of good wines usign Logistic Regression: {np.sum(y_pred_log =
```

```
Number of bad wines using Logistic Regression: 298
Number of good wines usign Logistic Regression: 20
Number of mislabeled wines out of a total 318 points (Logistics Regressio
n): 38

c:\Users\Usuario\miniconda3\envs\ML_env\lib\site-packages\sklearn\linear_m
odel\_logistic.py:444: ConvergenceWarning: lbfgs failed to converge (statu
s=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown i
n:
    https://scikit-learn.org/stable/modules/preprocessing.html (https://sc
ikit-learn.org/stable/modules/preprocessing.html)
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-reg
ression (https://scikit-learn.org/stable/modules/linear_model.html#logisti
c-regression)
  n_iter_i = _check_optimize_result(
```

3.2.2 Naive Bayes: Model definition

In [28]:
```python
# Import model
from sklearn.naive_bayes import GaussianNB

# Creation of an object "model"
model_nb = GaussianNB()

# Feeding training set into model
model_nb.fit(x_train,y_train)

# Printing results
y_pred_nb = model_nb.predict(x_test)
print(f"Number of bad wines using Naive Bayes: {np.sum(y_pred_nb =='bad')}"
print(f"Number of good wines usign Naive Bayes: {np.sum(y_pred_nb == 'good'
```

```
Number of bad wines using Naive Bayes: 261
Number of good wines usign Naive Bayes: 57
Number of mislabeled wines out of a total 318 points (Naive Bayes): 51
```

3.3 Classification models: Comparison

We first define which **metrics** will be used to compare and contrast the models, justifying their use. Afterwards, we perform the comparison and discuss the results.

3.3.1 Metrics: discussion

To reflect upon which metrics to use, it is key to remember the problem definition: we are predicting wine quality in basis of physico-chemical properties.

To begin with, the matter of **wine quality** forces us to consider whether it is best to over-estimate a wine's quality or under-estimate it. Again, thinking it is best to surprise than to disappoint (a business assumption), we ought to **prefer more false negatives than false positives**. Thus, our main metric is **precision**, since it represents the extent to which our positive predictions are close or not to the real ones. In the same line, we highlight the metric **false positive rate**, which we desire to be as low as possible (the FPR is defined as the tendency of getting false positives).

On the other hand, a metric such as recall is not so applicable, since it tends to favour false positives (since a **bad recall has more false negatives**). In sight of not ignoring false negatives, we will also consider in our analysis **accuracy**, as a general metric for the model.

In a few words, our key metrics are:

- **precision**
- **false positive rate**
- **accuracy**

3.3.2 Result presentation

In [29]:
```
bad_log = np.sum(y_pred_log =='bad')
good_log = np.sum(y_pred_log == 'good')
mislabelled_log = (y_test != y_pred_log).sum()

bad_nb = np.sum(y_pred_nb =='bad')
good_nb = np.sum(y_pred_nb == 'good')
mislabelled_nb = (y_test != y_pred_nb).sum()

good_test = np.sum(y_test == 'good')
bad_test = np.sum(y_test == 'bad')

# Result of the models
clasf_res = {'Context': ['Testing data', 'Logistic Regression', 'Naive Baye

clasf_resdf = pd.DataFrame(data=clasf_res)
clasf_resdf
```

Out[29]:

	Context	Good wines	Bad wines	Mislabelled wines
0	Testing data	38	280	0
1	Logistic Regression	20	298	38
2	Naive Bayes	57	261	51

With a first glance, logistic regression is better in terms of labelling, due to the lower amount of mislabelled wines. However, this is not the only metric to be analyzed, nor the most important.

3.3.3 Error analysis in models

In [30]:
```
# Calculating the training and testing accuracies for Logistics Regression
print("Training accuracy for Logistic Regression:", model_log.score(x_train
print("Testing accuracy for Logistic Regression:", model_log.score(x_test, 

print('------------------------------------------------------------')

# Calculating the training and testing accuracies for Naive Bayes
print("Training accuracy for Naive Bayes:", model_nb.score(x_train, y_train
print("Testing accuracy for Naive Bayes:", model_nb.score(x_test, y_test))
```

```
Training accuracy for Logistic Regression: 0.8770685579196218
Testing accuracy for Logistic Regression: 0.8805031446540881
------------------------------------------------------------
Training accuracy for Naive Bayes: 0.8502758077226162
Testing accuracy for Naive Bayes: 0.839622641509434
```

In terms of **accuracy**, the models are very similar in order of magnitude, with a slight advantage for the Logistic Regression.

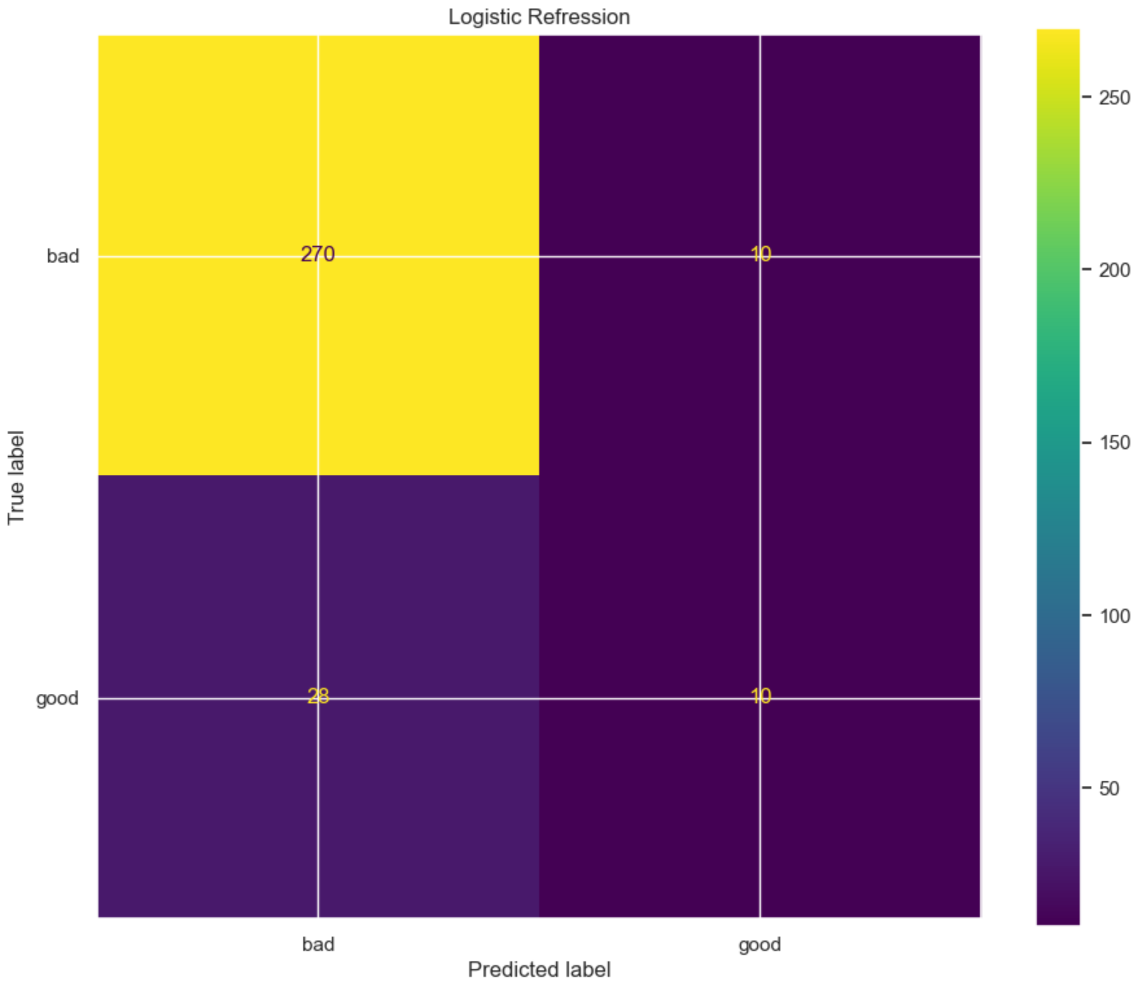
It should also be noted that in Naive Bayes the training accuracy is greater than the testing, whereas in the Logistic Regression this is reversed. The strength of a model is appreciated when it is **confronted to new data**: thus, this is also another advantageous point for the Logistic Regression.

In [31]:
```python
# Classification report for Logistic Regression
print('Logistic Regression')
print(classification_report(y_test, y_pred_log))

# Classification report for Naive Bayes
print('Naive Bayes')
```

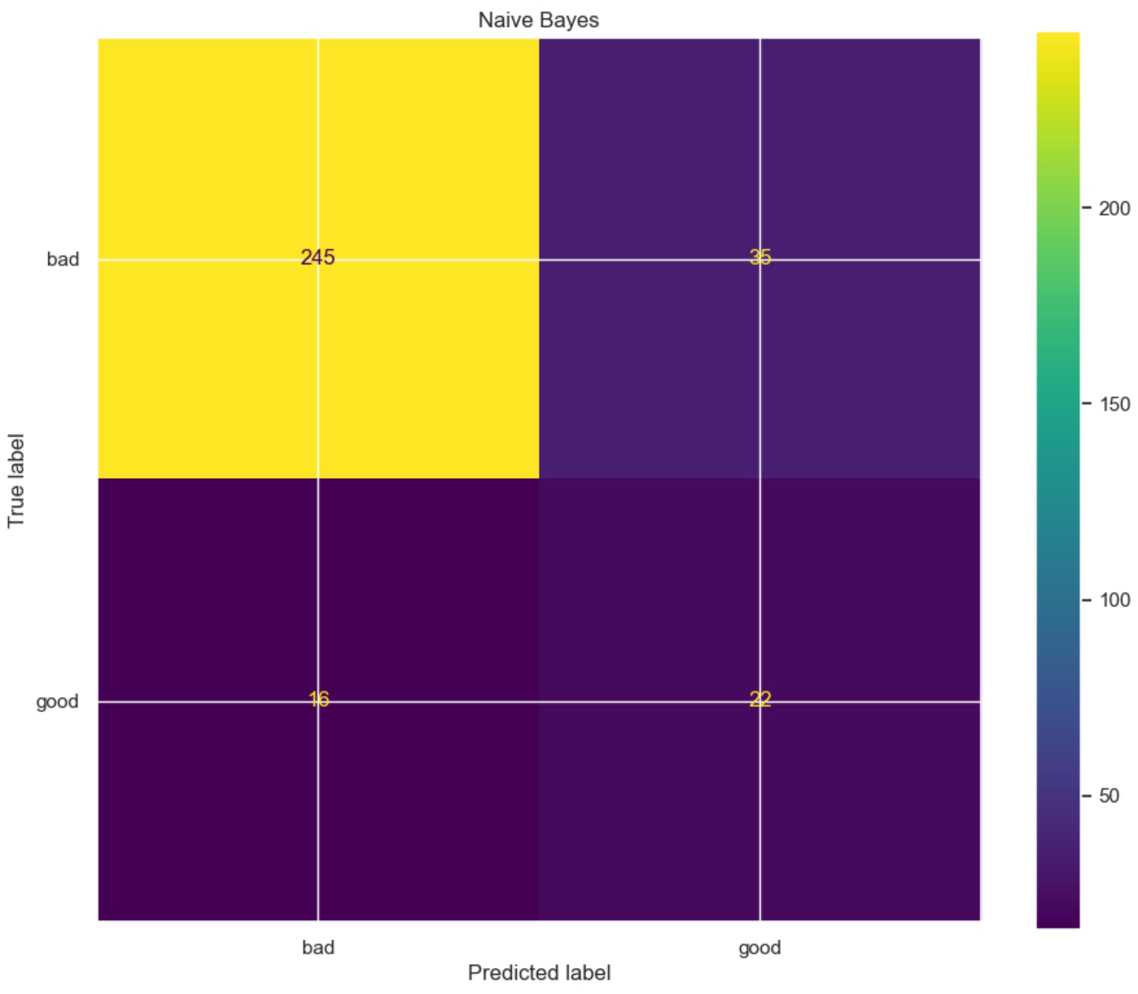
```
Logistic Regression
              precision    recall  f1-score   support

         bad       0.91      0.96      0.93       280
        good       0.50      0.26      0.34        38

    accuracy                           0.88       318
   macro avg       0.70      0.61      0.64       318
weighted avg       0.86      0.88      0.86       318

Naive Bayes
              precision    recall  f1-score   support

         bad       0.94      0.88      0.91       280
        good       0.39      0.58      0.46        38

    accuracy                           0.84       318
   macro avg       0.66      0.73      0.68       318
weighted avg       0.87      0.84      0.85       318
```

As justified earlier, the key metric here to be analyzed is the **'bad' precision**, where interestingly enough, Naive Bayes performs slightly better.

```
In [32]: # Confusion matrix for Logistic Regression

         conf_log = confusion_matrix(y_test, y_pred_log)
         disp = sklearn.metrics.ConfusionMatrixDisplay(conf_log, display_labels = ['
         disp.plot()
         plt.title('Logistic Refression')
```



In [33]:
```python
# Confusion matrix for Naive Bayes
conf_nb = confusion_matrix(y_test, y_pred_nb)
disp = sklearn.metrics.ConfusionMatrixDisplay(conf_nb, display_labels = ['ba
disp.plot()
plt.title('Naive Bayes')
```



Confusion matrices indicate the distribution of False and True Positives and Negatives. In this case, we can observe how Naive Bayes has a higher value of False Positives, whereas in Logistic Regression we see an greater amount of False Negatives. This can be linked to the corresponding recall and precision values.

Overall, Logistic Regression identifies better bad wines than Naive Bayes, and viceverse with good wines.

In [34]:
```python
# Calculating False Positive Rate for Logistic Regression
fpr_log = conf_log[0][1]/(conf_log[0][1] + conf_log[0][0])
print('FPR for Logistic Regression: ' + str(fpr_log))

# Calculating False Positive Rate for Naive Bayes
fpr_nb = conf_nb[0][1]/(conf_nb[0][1] + conf_nb[0][0])
```

```
FPR for Logistic Regression: 0.03571428571428571
FPR for Naive Bayes: 0.125
```

Yet another situation where Logistic Regression shows a strength: **we desire a low FPR**, since it suggests a lower tendency to show false positives.

All in all, the Logistic Regression shows better performance in terms of the metrics we defined.

- In terms of accuracy, the Logistic Regression performs slighly better, with a 0.87 (training) and 0.88 (testing) score, whereas in Naive Bayes the results were 0.85 (training) and 0.83 (testing).
- In relation with the FPR, the difference is much larger: the Logistic Regression is around 25% better with a 0.036 FPR.
- Finally, in regards to precision, although Naive Bayes has a higher bad precision (0.94), Logistic Regression's not that far, with a 0.91 score. Taking into account the dataset size, this difference can be discarted.

This means that if we were to choose between classifier methods, we would most definetly select the **Logistic Regression model**.

4 K-NN Algorithm

4.1 Introduction

The KNN algorithm, meaning "k- nearest neighbours" is a simple **classification** algorithm based on a supervision learning technique (because it needs a previously labeled dataset in order to properly "train").

The key concept of the algorithm is the assumption of similarity between old (training) data and new data. However, the algorithm does not actually train. The training phase consists of just storing all the data, and then performing some operation in the dataset in order to determine which class will be asigned to the new data.

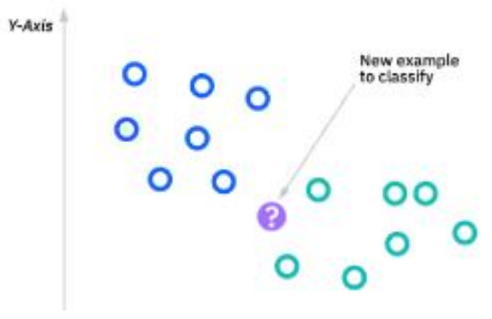
4.2 Mathematical background

First, let us consider the "1 nearest neighbour" classification, which equals to KNN when $k = 1$.

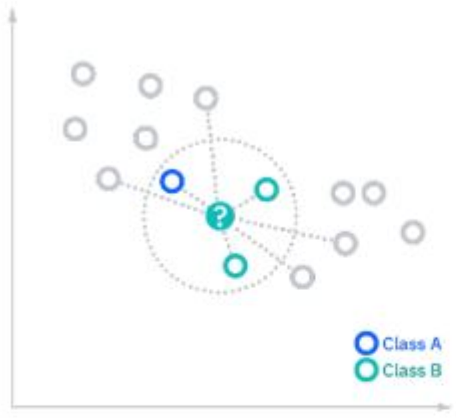
Suppose we have several classes in our dataset, and we add new data to it. the $1NN$ algorithm will asign the class of its closest neighbour, using some metric of distance. Usually, the euclidean norm is chosen.

Now, for the KNN algorithm, new data is classified by asigning the label which is most frequent among the **K NEARESTS GROUPS** in the dataset.

Graphically, it may look something like this:



The new data must be clasiffied into two existing classes, A and B. By looking at its 3 nearest neighbours, we can observe that the most frequent class is B



thus, the new data is assigned to class B.

This algorithm falls into the category of *"lazy learning"* because, instead of properly training (updating internal parameters of a certain model), it performs an operation over the readily available "training" dataset.

4.3 Parameter selection

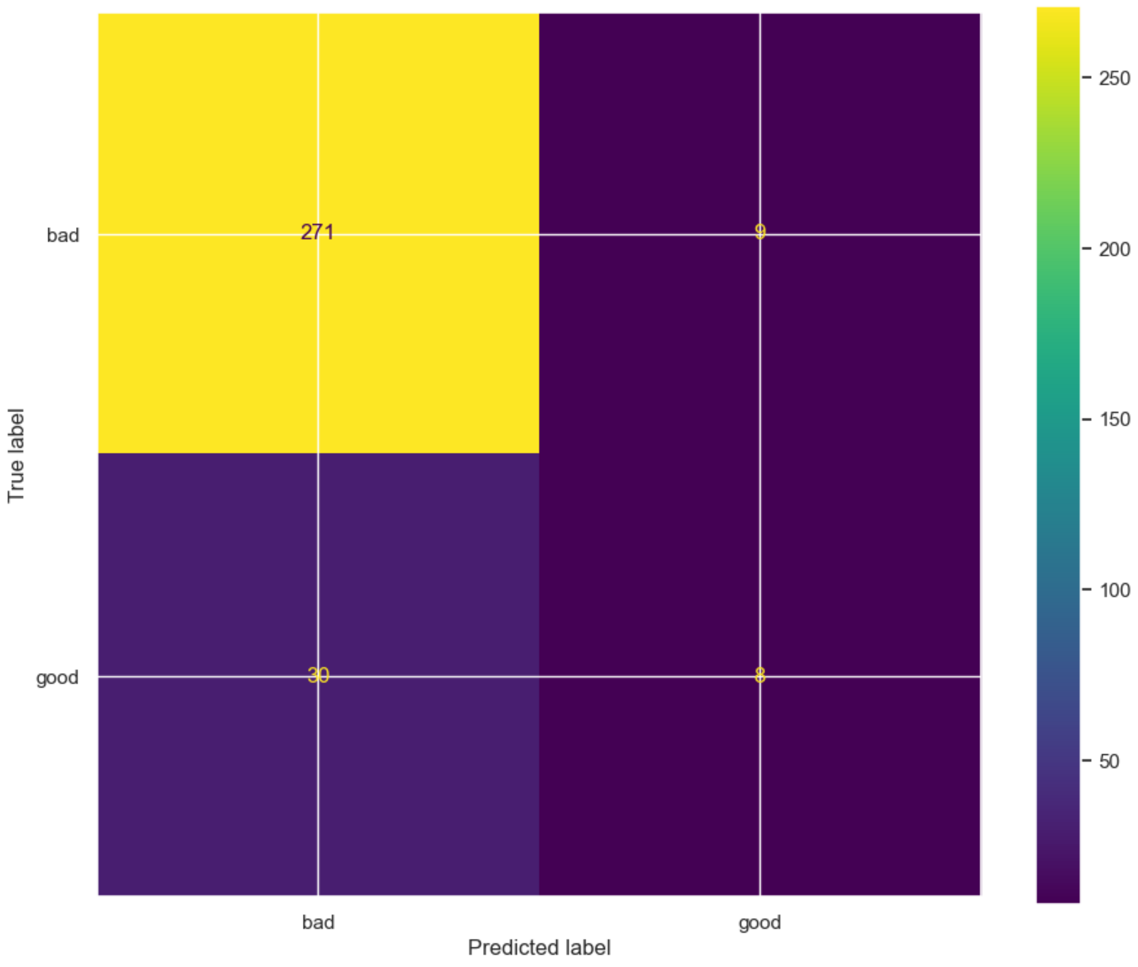
The choice of the parameter K depends strictly on data. An optimal value of K can be determined by various heuristic optimization techniques. In this document, we chose to run a grid search over K ranging from the **odd** numbers between 1 to 10, using the quality of wine as classes. The optimal parameter is the one with **least cross validated error**, according to scikit documentation.

In [35]:
```
# Imports
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score

# Specifying the parameter name and its range., usually odd numbers
parameters  = {'n_neighbors':[1, 3, 5, 7, 9]}

# Creating an object "model"
model_knn = KNeighborsClassifier()

# Define Grid Search object
optimal_Knn = GridSearchCV(model_knn, parameters)
optimal_Knn.fit(x_train,y_train) # perform the search

optimal_k = optimal_Knn.best_params_ # bestParameter = gridsearch.get_param
print(f"The optimal value for k parameter is: {optimal_k['n_neighbors']}")

# Predict over testing dataset
y_pred_knn = optimal_Knn.predict(x_test) # no need to define a #KNeighborsC
print("Accuracy Score:", accuracy_score(y_test,y_pred_knn))

# Classification report
print(classification_report(y_test, y_pred_knn))
```

```
The optimal value for k parameter is: 7
Accuracy Score: 0.8773584905660378
              precision    recall  f1-score   support

         bad       0.90      0.97      0.93       280
        good       0.47      0.21      0.29        38

    accuracy                           0.88       318
   macro avg       0.69      0.59      0.61       318
weighted avg       0.85      0.88      0.86       318
```

In [36]:
```
# Confusion matrix
conf_knn = confusion_matrix(y_test, y_pred_knn)
disp = sklearn.metrics.ConfusionMatrixDisplay(conf_knn, display_labels = ['
```

Out[36]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x19584c
fbdf0>
```



In [40]:
```
# Calculating False Positive Rate for K-NN
fpr_knn = conf_knn[0][1]/(conf_knn[0][1] + conf_knn[0][0])
```

```
FPR for K-NN (with k = 7): 0.03214285714285714
```

4.4 K-NN Model: Error analysis

Metrical analysis show solid performance from kNN algorithm.

- The FPR is as desired, showing low tendency to predict false positives.
- Precision on "bad" label is 0.9, a relatively good indicator.
- Accuracy score is 0.88, again a good metric.

However, this metrical analysis is not a huge improvement compared to the Logistic Regression from Section 3.

5 Conclusions

> In this last section, we will discuss the limits of the model and the reach of our solutions, as well as the business application of these Machine Learning

5.1 Model performances

To begin with, we must comment on **the problem's ambivalent nature**: having to select a integer value ranging from 1 to 10, implies that the solution could be modelled as a classification method **and/or** a linear regression of discrete nature. Thus, both linear and classifier models can be used. At first sight, this might seem like a serious limitation for any given model in particular, classifier or regression. However, in this context, the combination of models turns out to be an added value proposition. Thus the need to perform both a linear regression and a classifier method, models which can get feedback between each other (the tendency of one to over-estimate quality can be counterbalanced with another's tendency to under-estimate).

In addition, **the unbalance of the dataset** cannot be overlooked and is a vital issue, since it *pollutes any model we might have chosen*. Any training will be linked to the training dataset, insufficient in **data variety**. Under this circumstance, accuracy, FPR, precision of a model are not sufficient as representative metrics due to the fact that it is not symbolic for the potential real world data: the model will not be able to predict a 2-rated wine quality, since it never trained with one. This suggests that **none of the models developed are robust**. Models cannot be expected to behave appropriately if we train with an unbalanced dataset.

Moreover, potential improvements for the models (as discussed in section 2.4.2) could largely consist of a dataset expansion, the use of class weights to override the unbalance problem, a more exhaustive feature review or even the use of LOOCV given the dataset's size.

5.2 Use case discussion

Taking the latter comments into account, we must put into context the relevance of each model. In order to successfully analyze the business value of each model, we need to answer a situational question: **in which ways will the model be used?**

Will the model be used for Quality Control purposes in a wineyard? Or will its purpose relate with good wine indicators, as to select the wineyard's top products? While in the first situation the identification of bad wines is key, in the second one, even the notion of bad and good wine changes. Is a good wine defined as tasteful, or as healthily satisfactory? These open questions signify that we would need more context as to assess the limitations of the models.

Deepening this discussion, we would need to ponder what really makes a good wine. And here is where the debateful topic of the oenologist comes into picture. In our opinion, the wineyard cannot dispense with the oenologist: the determination of wine quality is an experience that cannot be simply reduced to these chemical properties (the whole is greater than the sum of its parts). The matter of taste is also involved. On top of that, and this is our most decisive argument (pragmatic, but certain), the models rest on an unbalanced dataset, so they cannot be completely trusted: the oenologist is still needed. We should even think if what we really desire is to substitute the oenologist or we are looking for the model to have another role.

Under this circumstance however, what we suggest is the shift in the dynamics of the determination of wine quality. Instead of Machine Learning being considered as a sustitution, in this context we believe it is more appropriate to consider it as a collaborative measure, **the model ought to be seen as a way in which the wineyard can provide support to the oenologist** in wine quality determination

6 Relevant bibliography

General

- “Predict Red Wine Quality.” Kaggle. Accessed September 24, 2022. https://www.kaggle.com/c/predict-red-wine-quality (https://www.kaggle.com/c/predict-red-wine-quality).
- Amidi, Shervine. “Machine Learning Tips and Tricks Cheatsheet Star.” Stanford CS 229 - Machine Learning Tips and Tricks Cheatsheet. Accessed September 24, 2022. https://stanford.edu/~shervine/teaching/cs-229/cheatsheet-machine-learning-tips-and-tricks (https://stanford.edu/~shervine/teaching/cs-229/cheatsheet-machine-learning-tips-and-tricks).

k-NN Algorithm

- Christopher, Antony. “K-Nearest Neighbor.” Medium, February 3, 2021. Accessed Septermber 24, 2022. https://medium.com/swlh/k-nearest-neighbor-ca2593d7a3c4 (https://medium.com/swlh/k-nearest-neighbor-ca2593d7a3c4).
- “What Is the K-Nearest Neighbors Algorithm?” IBM. Accessed September 24, 2022. https://www.ibm.com/topics/knn (https://www.ibm.com/topics/knn).

Related repositories:

- https://github.com/amberkakkar01/Prediction-of-Wine-Quality/blob/master/Wine_Quality.ipynb (https://github.com/amberkakkar01/Prediction-of-Wine-Quality/blob/master/Wine_Quality.ipynb)
- https://github.com/apoorva-dave/WineQualityPrediction/blob/master/predict_quality.ipynb (https://github.com/apoorva-dave/WineQualityPrediction/blob/master/predict_quality.ipynb)
- https://github.com/sharmaroshan/Wine-Quality-Predictions/blob/master/WineQuality.ipynb (https://github.com/sharmaroshan/Wine-Quality-Predictions/blob/master/WineQuality.ipynb)

Machine Learning Course at Universidad de Montevideo

Ariel Mordetzki and Mateo Stipaničić