Week 4: Interpretable Machine Learning for Data Science

Problem: You have been mandated by a large wine-making company in Valais to discover the key chemical factors that determine the quality of wine and build an interpretable model that will help their cellar masters make decisions daily.

Fetch the data

Here we have a very nice package that can do everything for us (aka `ucimlrepo`). Let's use it!

Take a look at the website for details.

```
from ucimlrepo import fetch_ucirepo
```

Fetch dataset.

```
wine_quality = fetch_ucirepo(id=186)
```

Parse data into pandas data frames.

```
X = wine_quality.data.features
y = wine_quality.data.targets
# added color separate variable
color = wine_quality.data.original[["color"]]
```

Display metadata of the dataset.

```
wine_quality.metadata
```

```
{'uci_id': 186,
 'name': 'Wine Quality',
 'repository_url': 'https://archive.ics.uci.edu/dataset/186/wine+quality',
 'data_url': 'https://archive.ics.uci.edu/static/public/186/data.csv',
 'abstract': 'Two datasets are included, related to red and white vinho verde wine samples,
from the north of Portugal. The goal is to model wine quality based on physicochemical tests
(see [Cortez et al., 2009], http://www3.dsi.uminho.pt/pcortez/wine/).',
 'area': 'Business',
 'tasks': ['Classification', 'Regression'],
 'characteristics': ['Multivariate'],
 'num_instances': 4898,
 'num_features': 11,
 'feature_types': ['Real'],
 'demographics': [],
 'target_col': ['quality'],
 'index_col': None,
 'has_missing_values': 'no',
 'missing_values_symbol': None,
 'year_of_dataset_creation': 2009,
 'last_updated': 'Wed Nov 15 2023',
 'dataset_doi': '10.24432/C56S3T',
 'creators': ['Paulo Cortez',
  'A. Cerdeira',
  'F. Almeida',
  'T. Matos',
  'J. Reis'],
 'intro_paper': {'ID': 252,
  'type': 'NATIVE',
  'title': 'Modeling wine preferences by data mining from physicochemical properties',
  'authors': 'P. Cortez, A. Cerdeira, Fernando Almeida, Telmo Matos, J. Reis',
  'venue': 'Decision Support Systems',
  'year': 2009,
  'journal': None,
  'DOI': None,
  'URL': 'https://www.semanticscholar.org/paper/Modeling-wine-preferences-by-data-mining-from-
Cortez-Cerdeira/bf15a0ccc14ac1deb5cea570c870389c16be019c',
  'sha': None,
  'corpus': None,
  'arxiv': None,
  'mag': None,
  'acl': None,
  'pmid': None,
  'pmcid': None},
 'additional_info': {'summary': 'The two datasets are related to red and white variants of the
Portuguese "Vinho Verde" wine. For more details, consult: http://www.vinhoverde.pt/en/ or the
reference [Cortez et al., 2009].  Due to privacy and logistic issues, only physicochemical
(inputs) and sensory (the output) variables are available (e.g. there is no data about grape
types, wine brand, wine selling price, etc.).\n\nThese datasets can be viewed as
classification or regression tasks.  The classes are ordered and not balanced (e.g. there are
many more normal wines than excellent or poor ones). Outlier detection algorithms could be
used to detect the few excellent or poor wines. Also, we are not sure if all input variables
are relevant. So it could be interesting to test feature selection methods.\n',
  'purpose': None,
  'funded_by': None,
  'instances_represent': None,
  'recommended_data_splits': None,
  'sensitive_data': None,
  'preprocessing_description': None,
  'variable_info': 'For more information, read [Cortez et al., 2009].\r\nInput variables
(based on physicochemical tests):\r\n   1 - fixed acidity\r\n   2 - volatile acidity\r\n   3 -
citric acid\r\n   4 - residual sugar\r\n   5 - chlorides\r\n   6 - free sulfur dioxide\r\n   7
- total sulfur dioxide\r\n   8 - density\r\n   9 - pH\r\n   10 - sulphates\r\n   11 -
alcohol\r\nOutput variable (based on sensory data): \r\n   12 - quality (score between 0 and
10)',
  'citation': None}}
```

As we can see, there are 11 features and none require any imputation as there are no missing values.

Moreover, all the features are numerical as the are continuous variables.

Now, let's check that the data have the correct shape to ensure they have been loaded as expected.

Calculate how many samples and features we have in total, how many are red or white wines, how many are good or bad wines, etc.

```
X.shape
```

```
(6497, 11)
```

```
y.shape
```

```
(6497, 1)
```

There are 6497 samples inside the dataset.

```
X.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6497 entries, 0 to 6496
Data columns (total 11 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   fixed_acidity         6497 non-null   float64
 1   volatile_acidity      6497 non-null   float64
 2   citric_acid           6497 non-null   float64
 3   residual_sugar        6497 non-null   float64
 4   chlorides             6497 non-null   float64
 5   free_sulfur_dioxide   6497 non-null   float64
 6   total_sulfur_dioxide  6497 non-null   float64
 7   density               6497 non-null   float64
 8   pH                    6497 non-null   float64
 9   sulphates             6497 non-null   float64
 10  alcohol               6497 non-null   float64
dtypes: float64(11)
memory usage: 558.5 KB
```

All the features are defined as `float64` programmatically.

```
y.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6497 entries, 0 to 6496
Data columns (total 1 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   quality  6497 non-null   int64
dtypes: int64(1)
memory usage: 50.9 KB
```

```
color.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6497 entries, 0 to 6496
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   color   6497 non-null   object
dtypes: object(1)
memory usage: 50.9+ KB
```

The `quality` of the wine is defined as an integer and works so but the `color` variable may be further encoded as it is a string, as of now.

```
y.value_counts()
```

```
quality
6          2836
5          2138
7          1079
4           216
8           193
3            30
9             5
Name: count, dtype: int64
```

As we can see, even though the `quality` features takes value in the set $[0, 10]$, there are no wines whose `quality` is lower than 6 and higher than 9.

The different qualities aren't well represented as the number of wines of higher quality decreases greatly.

```
color.value_counts(normalize=True)
```

```
color
white    0.753886
red      0.246114
Name: proportion, dtype: float64
```

The different wine color aren't well distributed as well as there are more white wines (~75% of the dataset) than red ones (~25%).

Data Exploration

We now will inspect the features one-by-one, and try to understand their dynamics, especially between white and red wines.

- Use `Dataframe.describe` to display statistics on each feature. Do the same for red wines only, and white wines only. Do you notice any clear difference?
- Compute the effect size by computing the strictly standardized mean difference (SSMD) between the red and white wines for each feature.

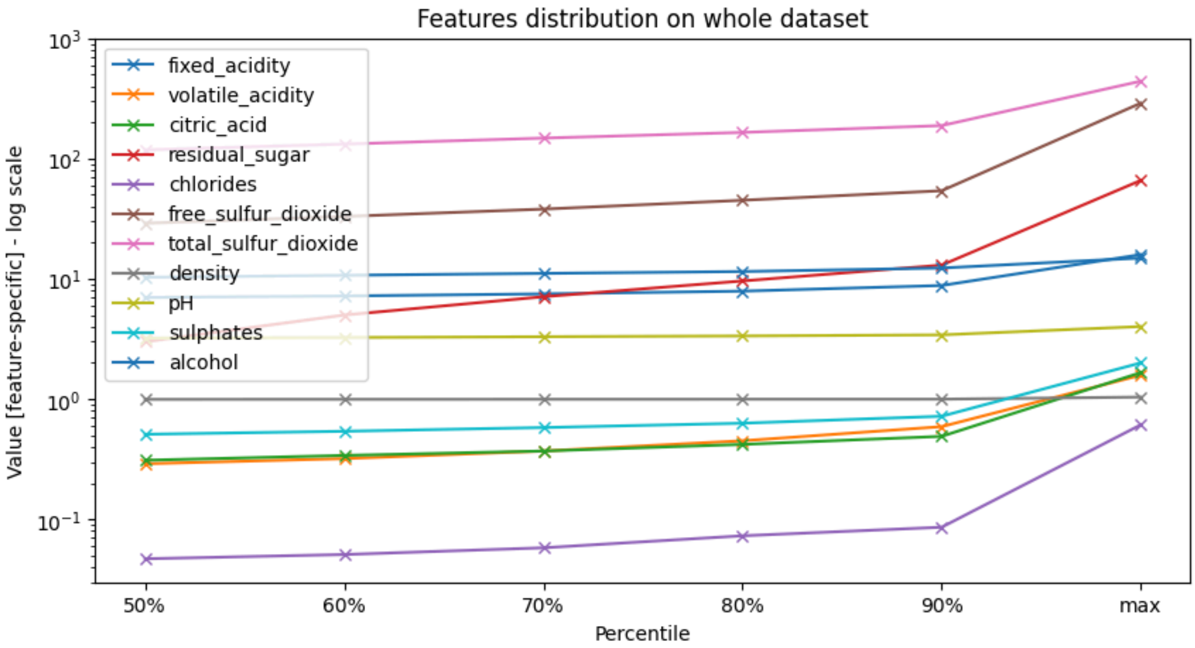
```
X.describe()
```

```
       fixed_acidity  volatile_acidity  citric_acid  residual_sugar  \
count    6497.000000       6497.000000  6497.000000     6497.000000   
mean        7.215307          0.339666     0.318633        5.443235   
std         1.296434          0.164636     0.145318        4.757804   
min         3.800000          0.080000     0.000000        0.600000   
25%         6.400000          0.230000     0.250000        1.800000   
50%         7.000000          0.290000     0.310000        3.000000   
75%         7.700000          0.400000     0.390000        8.100000   
max        15.900000          1.580000     1.660000       65.800000   

         chlorides  free_sulfur_dioxide  total_sulfur_dioxide      density  \
count  6497.000000          6497.000000           6497.000000  6497.000000   
mean      0.056034            30.525319            115.744574     0.994697   
std       0.035034            17.749400             56.521855     0.002999   
min       0.009000             1.000000              6.000000     0.987110   
25%       0.038000            17.000000             77.000000     0.992340   
50%       0.047000            29.000000            118.000000     0.994890   
75%       0.065000            41.000000            156.000000     0.996990   
max       0.611000           289.000000            440.000000     1.038980   

                pH    sulphates      alcohol  
count  6497.000000  6497.000000  6497.000000  
mean      3.218501     0.531268    10.491801  
std       0.160787     0.148806     1.192712  
min       2.720000     0.220000     8.000000  
25%       3.110000     0.430000     9.500000  
50%       3.210000     0.510000    10.300000  
75%       3.320000     0.600000    11.300000  
max       4.010000     2.000000    14.900000  
```

As we can see when observing the global wine distribution, there might be outliers when considering different features as the `max` value seems to differ greatly from the distribution as it differs a lot from the 75th percentile (seen in `fixed_acidity`, `volatile_acidity`, `citric_acid`, `chlorides`, `free_sulfur_dioxide`, `total_sulfur_dioxide` and `sulphates`).

We can also visualize it.

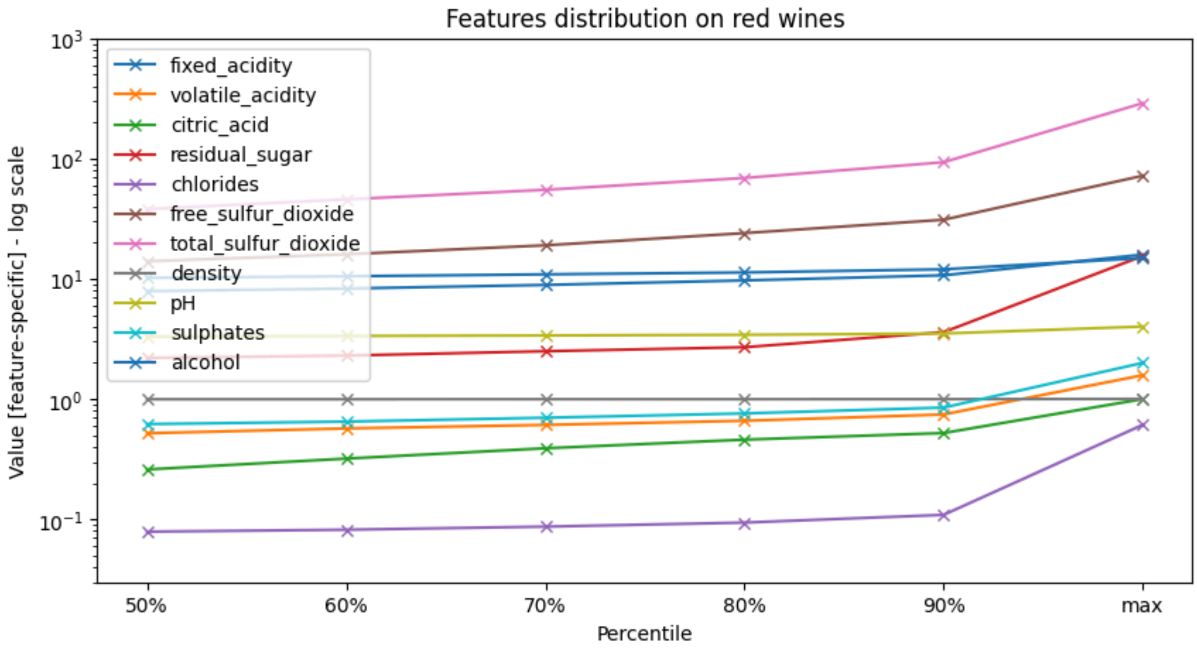
```
plt.figure()
X.describe(percentiles=[0.5, 0.6, 0.7, 0.8, 0.9]).loc[
    ["50%", "60%", "70%", "80%", "90%", "max"]
].plot(marker="x", figsize=(10, 5))
plt.title("Features distribution on whole dataset")
plt.yscale("log")
plt.ylim((3e-2, 1e3))
plt.xlabel("Percentile")
plt.ylabel("Value [feature-specific] - log scale")
```



The log-scale allows us to better visualize the magnitude changes in different percentiles for the variables.

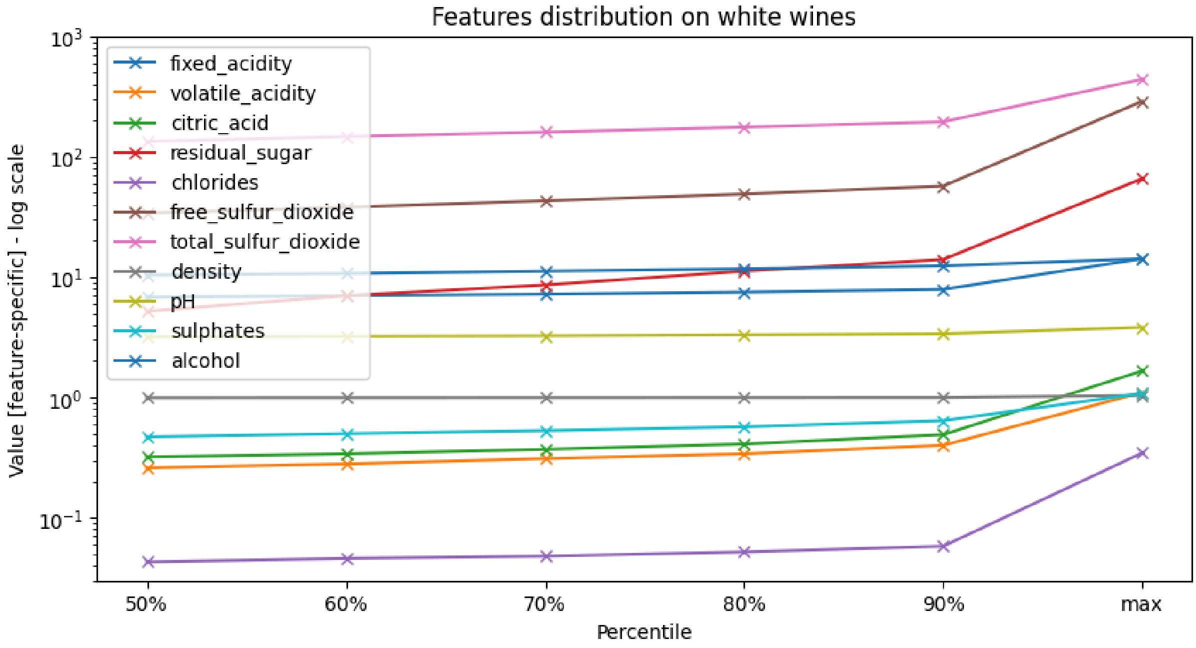
We can now analyze the wine distributions per color.

```
X_color = pd.concat([X, color], axis=1)
X_color_red = X_color[X_color["color"] == "red"]
X_color_white = X_color[X_color["color"] == "white"]

plt.figure()
X_color_red.describe(percentiles=[0.5, 0.6, 0.7, 0.8, 0.9]).loc[
    ["50%", "60%", "70%", "80%", "90%", "max"]
].plot(marker="x", figsize=(10, 5))
plt.title("Features distribution on red wines")
plt.yscale("log")
plt.ylim((3e-2, 1e3))
plt.xlabel("Percentile")
plt.ylabel("Value [feature-specific] - log scale")
```



Visualizing the distribution of features of red wines clearly displays a shift. The `residual_sugar` feature clearly shows that the order of magnitude of residual sugars is smaller when considering only red wines. We can also assess that the `total_sulfur_dioxide` and `free_sulfur_dioxide` are of lower magnitude.

```
plt.figure()
X_color_white.describe(percentiles=[0.5, 0.6, 0.7, 0.8, 0.9]).loc[
    ["50%", "60%", "70%", "80%", "90%", "max"]
].plot(marker="x", figsize=(10, 5))
plt.title("Features distribution on white wines")
plt.yscale("log")
plt.ylim((3e-2, 1e3))
plt.xlabel("Percentile")
plt.ylabel("Value [feature-specific] - log scale")
```



The white wines features distribution better fits the global distribution as they are more samples of such wine inside the dataset. The distribution of red wine features are hence "drowned" when considered together with white ones.

We can now compute the effect size between red and white wines.

```
def compute_SSMD(xs1: np.array, xs2: np.array) -> float:
    """
    Compute SSMD of two features `xs1` and `xs2`.
    """

    mean_delta = xs1.mean() - xs2.mean()
    var_sum = xs1.var() + xs2.var()

    return mean_delta / np.sqrt(var_sum)
```

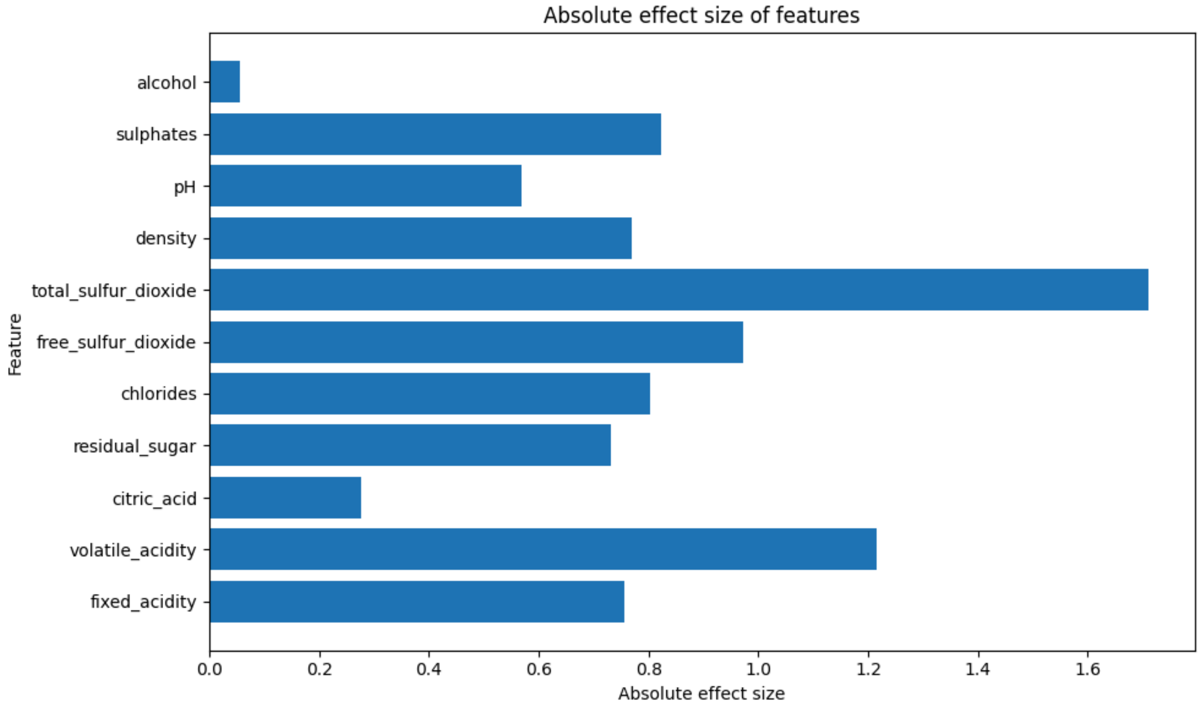
```
X_color_white.columns
```

```
Index(['fixed_acidity', 'volatile_acidity', 'citric_acid', 'residual_sugar',
       'chlorides', 'free_sulfur_dioxide', 'total_sulfur_dioxide', 'density',
       'pH', 'sulphates', 'alcohol', 'color'],
      dtype='object')
```

```
features_effect_size = [
    (c, compute_SSMD(X_color_white[c], X_color_red[c]))
    for c in X_color_white.columns
    if c != "color"
]
```

Per definition, a smaller absolute effect size indicates a bigger similarity between two distributions. We can hence plot the absolute effect size per feature.

```
features = [item[0] for item in features_effect_size]
values = [abs(item[1]) for item in features_effect_size]

plt.figure(figsize=(10, 6))
plt.barh(features, values)
plt.xlabel("Absolute effect size")
plt.ylabel("Feature")
plt.title("Absolute effect size of features")
plt.tight_layout()
plt.show()
```



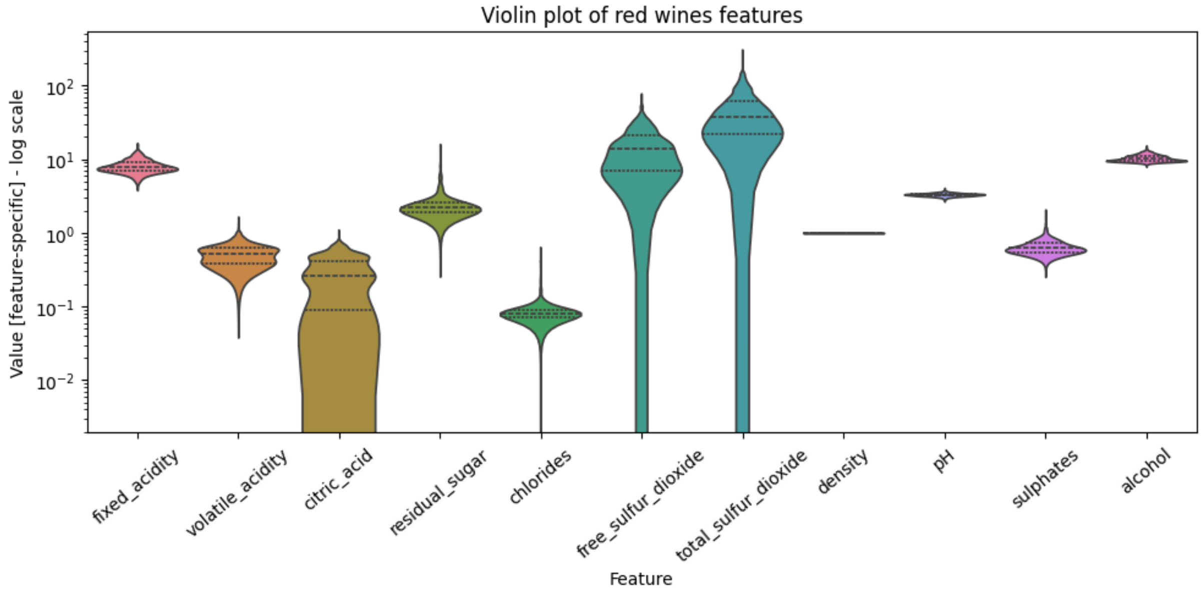
As we can see, the lowest effect size between red and white wines is the `alcohol` feature. Most interestingly, the features that might have the most predictive quality are `total_sulfur_dioxide` and `volatile_acidity`.

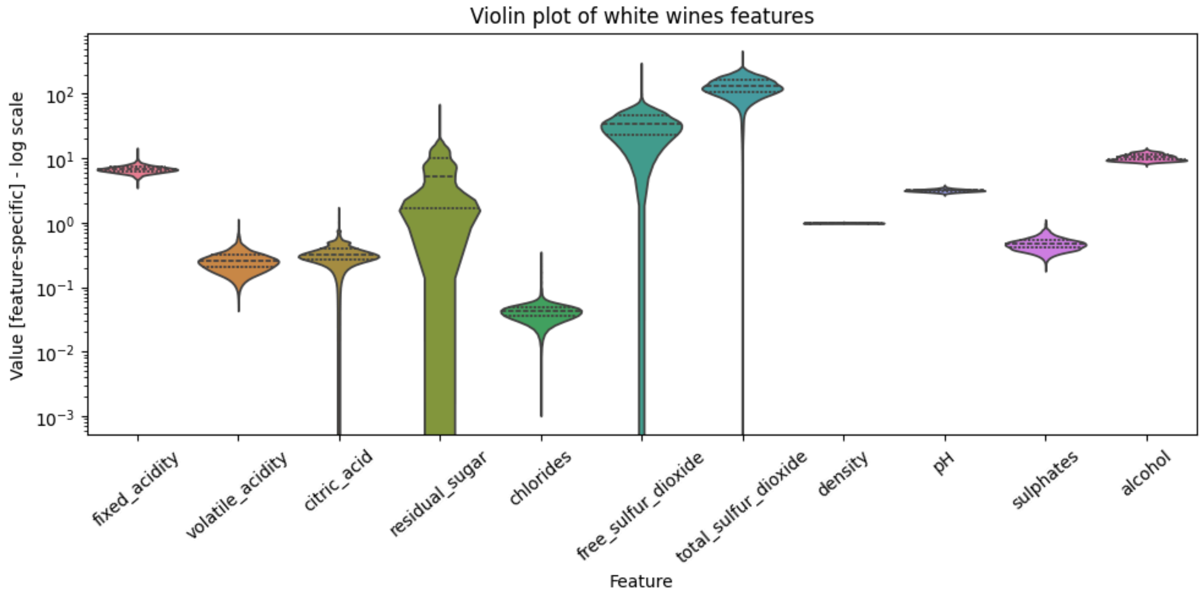
Now let's go a bit deeper into the same analysis, using more visual tools:

- For every feature, plot box plots, violin plots or histograms for red and white wines. What can you infer? **If you feel a bit more adventurous**, plot the Cumulative Distribution Function (CDF) of the feature for white and red wines, and compute the Kullback-Leibler divergence (or entropy) between them. Explain why this might be useful.
- Plot the correlation matrix of all features as heat maps, one for red and one for white wines. How do they differ? What can you infer?

We may first display a violin plot for the red and respectively white wines.

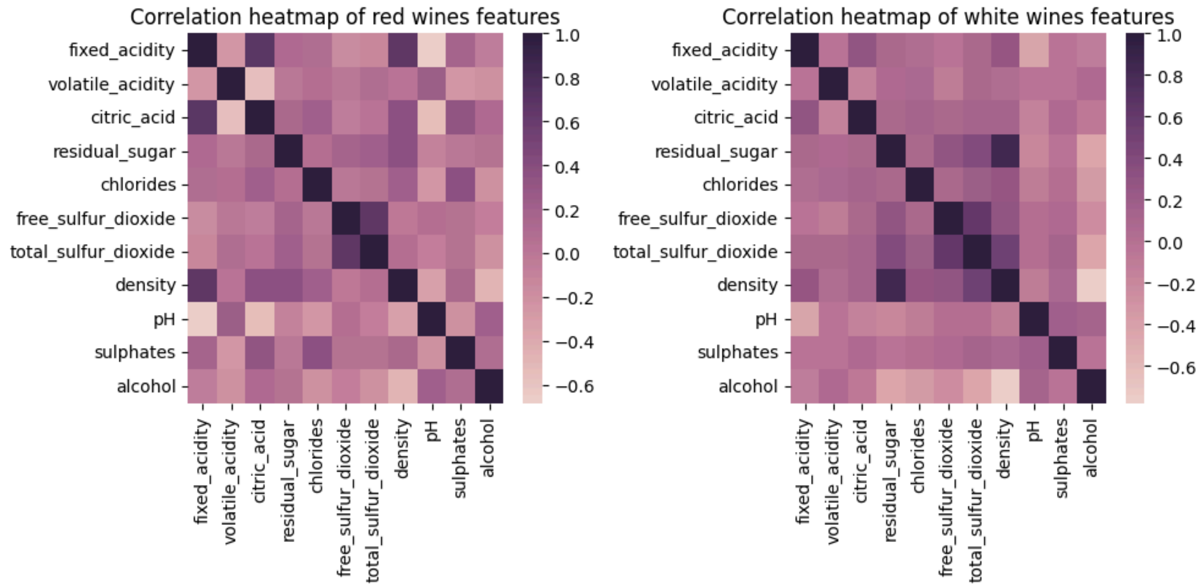
```
fig, axes = plt.subplots(2, 1, figsize=(10, 10))

# plot red wines
sns.violinplot(X_color_red, inner="quart", ax=axes[0])
axes[0].set_title("Violin plot of red wines features")
axes[0].set_xlabel("Feature")
axes[0].set_ylabel("Value [feature-specific] - log scale")
axes[0].set_yscale("log")
axes[0].tick_params(axis="x", rotation=40)

# plot white wines
sns.violinplot(X_color_white, inner="quart", ax=axes[1])
axes[1].set_title("Violin plot of white wines features")
axes[1].set_xlabel("Feature")
axes[1].set_ylabel("Value [feature-specific] - log scale")
axes[1].set_yscale("log")
axes[1].tick_params(axis="x", rotation=40)

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





The two violin plots display, per feature, their quartiles (25-50-75) and their probability density (in color). As such, we can assess key differences between red and white ones (as previously first visualized). The distribution of the continuous variable `citric_acid` is of lower magnitude in red wines. The `residual_sugar` distribution is of lower magnitude in white wines. As previously stated, the distribution of `free_sulfur_dioxide` and `total_sulfur_dioxide` is more distributed in red wines when not considering outliers.

```
plt.figure(figsize=(10, 5))

# display red wines
plt.subplot(1, 2, 1)
sns.heatmap(
    X_color_red.select_dtypes(include=["float64"]).corr(),
    cmap=sns.cubehelix_palette(as_cmap=True),
)
plt.title("Correlation heatmap of red wines features")

# display white wines
plt.subplot(1, 2, 2)
sns.heatmap(
    X_color_white.select_dtypes(include=["float64"]).corr(),
    cmap=sns.cubehelix_palette(as_cmap=True),
)
plt.title("Correlation heatmap of white wines features")

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



As we can assess, the correlation is the greatest between `citric_acid` and `fixed_acidity`, `density` and `fixed_acidity`, `total_sulfur_dioxide` and `free_sulfur_dioxide` in red wines and between `density` and `residual_sugar` in white wines.

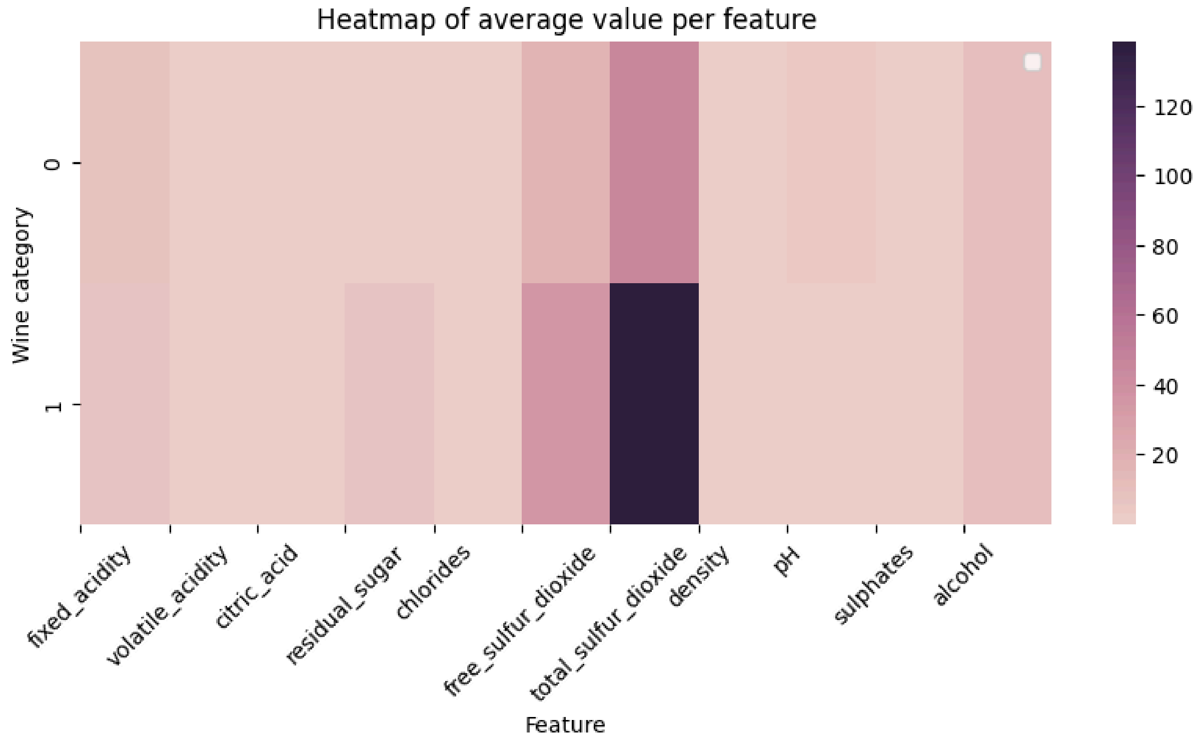
As such, these different correlations may indicate the predictive property of the different variables as a model may make use of it.

Data Exploration using Unsupervised Learning

We first explore the data in an unsupervised fashion. Start by creating a heat map of the average feature value for red and white wines. Can you spot an easy way to differentiate between reds and whites?

```
X_color_red_averaged = X_color_red.select_dtypes(include=["float64"]).agg("mean")
X_color_white_averaged = X_color_white.select_dtypes(include=["float64"]).agg("mean")

plt.figure(figsize=(10, 4))
sns.heatmap(
    np.vstack((X_color_red_averaged, X_color_white_averaged)),
    cmap=sns.cubehelix_palette(as_cmap=True),
)
plt.xlabel("Feature")
plt.ylabel("Wine category")
plt.xticks(
    ticks=np.arange(len(X_color_red_averaged.index)),
    labels=X_color_red_averaged.index.array,
    rotation=45,
)
plt.title("Heatmap of average value per feature")
plt.legend()
plt.show()
```

```
C:\Users\Dion\AppData\Local\Temp\ipykernel_15456\103545337.py:14: UserWarning: No artists with
labels found to put in legend.  Note that artists whose label start with an underscore are
ignored when legend() is called with no argument.
  plt.legend()
```



We may say that red wines have a greater `free_sulfur_dioxide` than white wines and so goes on with the `total_sulfur_dioxide` (consequence).

Using PCA to reduce the dimensionality

Use PCA to reduce the dimensionality of data. Do not forget that it requires data normalization (centering on the mean and scaling to unit variance). Plot the whole dataset onto the two principal components and color it by wine color. What does it tell you?

Project the unit vectors that correspond to each vector onto the principal components, using the same transformation. What does it tell you about the relative feature importance? Does it match the observations you made previously?

We may first standardize the different features.

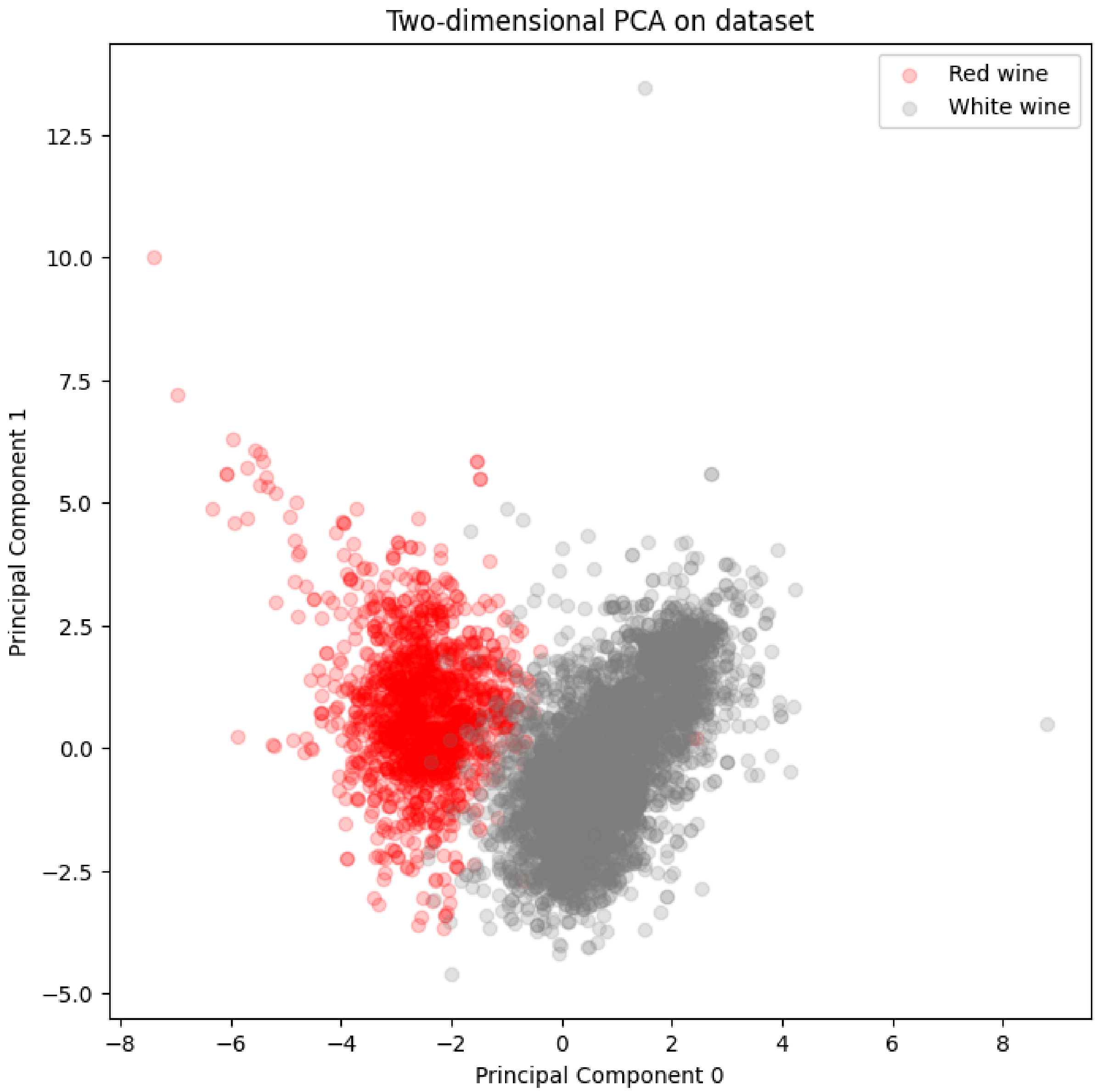
```
standard_scaler = StandardScaler().set_output(transform="pandas")

X_standardized = standard_scaler.fit_transform(X)

X_standardized.describe().loc[["mean", "std"]]
```

```
      fixed_acidity  volatile_acidity   citric_acid  residual_sugar  \
mean  -3.849639e-16      1.049902e-16  2.187295e-17    3.499672e-17   
std    1.000077e+00      1.000077e+00  1.000077e+00    1.000077e+00   

         chlorides  free_sulfur_dioxide  total_sulfur_dioxide       density  \
mean  1.749836e-17        -8.749179e-17         -6.999344e-17 -3.552167e-15   
std   1.000077e+00         1.000077e+00          1.000077e+00  1.000077e+00   

                pH     sulphates       alcohol  
mean  2.729744e-15 -5.424491e-16  6.561885e-16  
std   1.000077e+00  1.000077e+00  1.000077e+00  
```

As we can notice, the different features are standardized as the mean $\mu = 0$ and standard deviation is $\sigma = 1$.

We may now go on and apply and then display a PCA on the data.

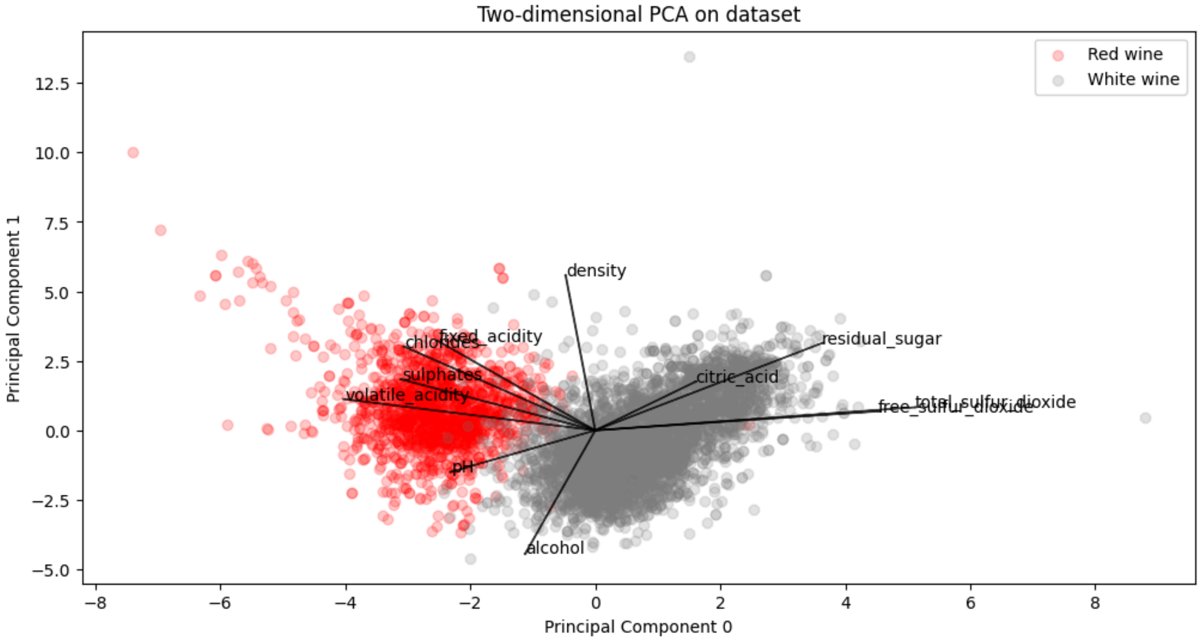
```
pca_2 = PCA(n_components=2, random_state=42).set_output(transform="pandas")

# attention, must do PCA on whole
# dataset and then split on color
# to keep distributions intact
X_pca = pca_2.fit_transform(X_standardized)

X_color_red_pca = pd.concat([X_pca, color], axis=1)
X_color_red_pca = X_color_red_pca[X_color_red_pca["color"] == "red"].drop(
    columns=["color"]
)

X_color_white_pca = pd.concat([X_pca, color], axis=1)
X_color_white_pca = X_color_white_pca[X_color_white_pca["color"] == "white"].drop(
    columns=["color"]
)

plt.figure(figsize=(8, 8))

plt.scatter(
    X_color_red_pca[["pca0"]],
    X_color_red_pca[["pca1"]],
    color="red",
    alpha=0.2,
    label="Red wine",
)
plt.scatter(
    X_color_white_pca[["pca0"]],
    X_color_white_pca[["pca1"]],
    color="gray",
    alpha=0.2,
    label="White wine",
)

plt.xlabel("Principal Component 0")
plt.ylabel("Principal Component 1")
plt.legend()
plt.title("Two-dimensional PCA on dataset")
plt.show()
```



As we can notice, reducing the data to a two-dimensional plane yields visual difference between red and white wine.

We can now display the different unit vectors that compose the higher-dimensional space by projecting them onto our plane.

```
# multiply by sqrt(variance) := std
loadings = pca_2.components_.T * np.sqrt(pca_2.explained_variance_ratio_) * 20

plt.figure(figsize=(12, 6))
plt.scatter(
    X_color_red_pca[["pca0"]],
    X_color_red_pca[["pca1"]],
    color="red",
    alpha=0.2,
    label="Red wine",
)
plt.scatter(
    X_color_white_pca[["pca0"]],
    X_color_white_pca[["pca1"]],
    color="gray",
    alpha=0.2,
    label="White wine",
)

for index, feature in enumerate(X.columns):
    plt.annotate(feature, (loadings[index, 0], loadings[index, 1]), color="black")

    plt.arrow(
        0,
        0,
        loadings[index, 0],
        loadings[index, 1],
        color="black",
        alpha=0.7,
        width=0.01,
    )


plt.xlabel("Principal Component 0")
plt.ylabel("Principal Component 1")
plt.legend()
plt.title("Two-dimensional PCA on dataset")
plt.show()
```



Displaying the projected components allows to visualize the most contributing axes when projecting into the plane (`free_sulfur_dioxide` and `total_sulfur_dioxide`).

Cluster the data in 2-dimensional space

Use k-means to cluster the data into 2 clusters and plot the same view as before, but with a coloring that corresponds to the cluster memberships.

Assuming that the cluster assignments are predictions of a model, what is the performance you can achieve in terms of mutual information score, accuracy, and f1 score?

```
color_encoded = [0 if c == "red" else 1 for _, c in color["color"].items()]
kmeans = KMeans(n_clusters=2, random_state=42)

y_clustered_11 = kmeans.fit_predict(X)
y_clustered_2 = kmeans.fit_predict(X_pca)

def print_metrics(y_true, y_pred):
    print(f"Mutual Information Score -> {mutual_info_score(y_true, y_pred)}")
    print(f"Accuracy Score -> {accuracy_score(y_true, y_pred)}")
    print(f"F1-Score -> {f1_score(y_true, y_pred)}")

print("Indicators on clustering of high-dimensional data :")
print_metrics(np.ravel(color_encoded), y_clustered_11)
```

```
Indicators on clustering of high-dimensional data :
Mutual Information Score -> 0.19652932172065912
Accuracy Score -> 0.7854394335847315
F1-Score -> 0.8373015873015873
```

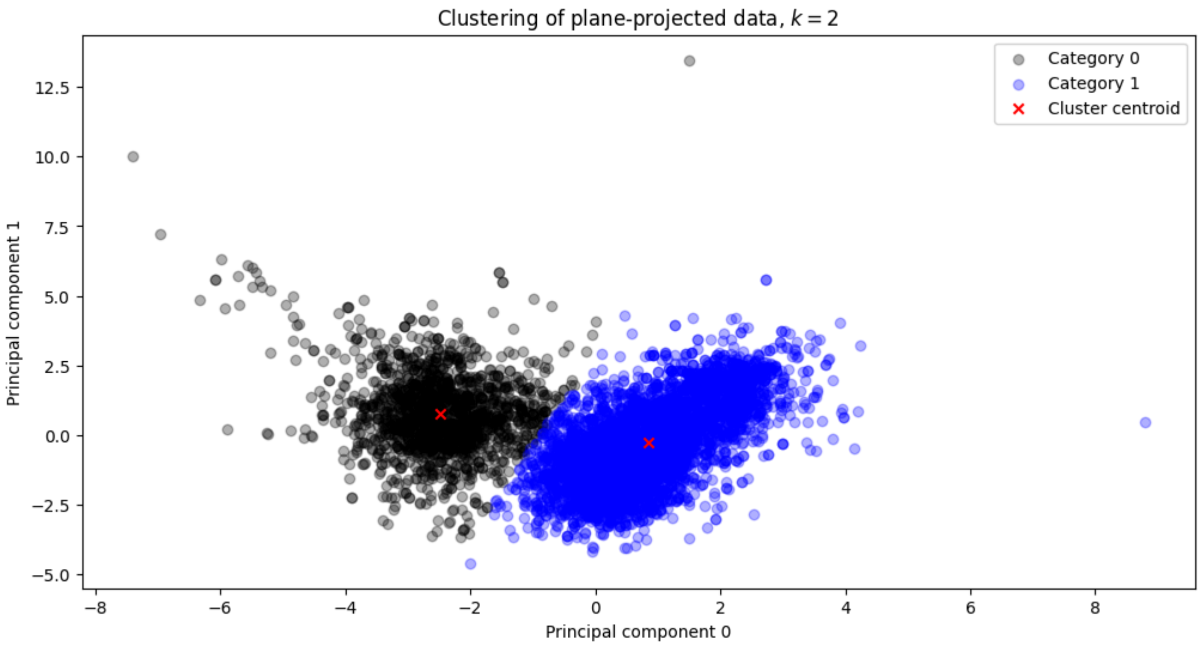
```
print("Indicators on clustering of high-dimensional data :")
print_metrics(np.ravel(color_encoded), y_clustered_2)
```

```
Indicators on clustering of high-dimensional data :
Mutual Information Score -> 0.480717971115607
Accuracy Score -> 0.9827612744343543
F1-Score -> 0.9884986650236188
```

As we can see, the indicators show greater performance when clustering the already plane-projected data. We can now display it to assess wether it clustered on wine color of other underlying properties.

```
X_pca_cluster_0 = X_pca[kmeans.labels_ == 0]
X_pca_cluster_1 = X_pca[kmeans.labels_ == 1]

plt.figure(figsize=(12, 6))

plt.scatter(
    X_pca_cluster_0[["pca0"]],
    X_pca_cluster_0[["pca1"]],
    color="black",
    alpha=0.3,
    label="Category 0",
)
plt.scatter(
    X_pca_cluster_1[["pca0"]],
    X_pca_cluster_1[["pca1"]],
    color="blue",
    alpha=0.3,
    label="Category 1",
)
plt.scatter(
    kmeans.cluster_centers_[0, 0],
    kmeans.cluster_centers_[0, 1],
    color="red",
    marker="x",
)
```

```
plt.scatter(
    kmeans.cluster_centers_[1, 0],
    kmeans.cluster_centers_[1, 1],
    color="red",
    marker="x",
    label="Cluster centroid",
)
plt.title("Clustering of plane-projected data, $k=2$")
plt.xlabel("Principal component 0")
plt.ylabel("Principal component 1")
plt.legend()

plt.show()
```



KMeans seems to have clustered by color based on the graph.

Now, we are going to train a **supervised** linear classification model using sklearn, and compare the results with the approach using clustering.

- Set up a train/test dataset using sklearn.model_selection.train_test_split.
- Use GridSearchCV to perform a cross-validation of the model's regularization C.
- Compare the test and train performance at the end. Does the model suffer from any overfitting?
- Analyze the test performance specifically. What can you conclude about this general problem of recognizing white vs red wines?

We must first encode the output labels so that the logistic regression is possible.

```
label_encoder = LabelEncoder()
```

```
color_encoded = label_encoder.fit_transform(color["color"])
```

The splitting of the data will be done with stratifying to make sure that the distribution of the training labels is the same (in terms of balance) as the one of the training split.

```
X_train, X_test, y_train, y_test = train_test_split(
    X, color_encoded, train_size=0.8, stratify=color_encoded
)
```

We must now standardize the different features so that the distributions in training and testing set are comparable.

```
X_train = standard_scaler.fit_transform(X_train)
X_test = standard_scaler.transform(X_test)
```

```
X_train.describe().loc[["mean", "std"]].agg("mean", axis=1)
```

```
mean   -2.975096e-15
std     1.000096e+00
dtype: float64
```

```
X_test.describe().loc[["mean", "std"]].agg("mean", axis=1)
```

```
mean    0.004291
std     0.986543
dtype: float64
```

As we can see, the numerical features of the training and testing split are normalized.

```
logistic_regression = GridSearchCV(
    LogisticRegression(max_iter=10000, random_state=42),
    param_grid={"C": list(map(lambda _: np.random.rand() * 10, range(0, 50)))},
    scoring="accuracy",
    n_jobs=-1,
)
```

```
logistic_regression.fit(X_train, np.ravel(y_train))
```

```
GridSearchCV(estimator=LogisticRegression(max_iter=10000, random_state=42),
             n_jobs=-1,
             param_grid={'C': [2.192504089540226, 2.1112527626569157,
                               1.1909746719728664, 9.29497571595147,
                               0.6188054900281048, 2.205880515704407,
                               2.5361950440599235, 8.392228845841336,
                               4.996378371347932, 4.9045445091369055,
                               2.555544126510876, 2.4665079955067313,
                               4.435714132005498, 3.447882579023479,
                               4.698904593767099, 8.916203754486114,
                               3.407318984655493, 5.044846055787974,
                               4.593729037472669, 2.1714257400007417,
                               7.806171172798244, 6.501470827428209,
                               6.949973706215864, 7.587627472333844,
                               5.723592542233947, 5.1631293086966545,
                               8.981232524237564, 5.681840824519467,
                               3.977412099259566, 4.544776207050836, ...]},
             scoring='accuracy')
```

```
logistic_regression = logistic_regression.best_estimator_
```

```
y_train_predicted = logistic_regression.predict(X_train)
y_test_predicted = logistic_regression.predict(X_test)
```

```
print("Performance on training split :")
print_metrics(np.ravel(y_train), y_train_predicted)
```

```
Performance on training split :
Mutual Information Score -> 0.5210778265513913
Accuracy Score -> 0.9938426015008659
F1-Score -> 0.9959183673469387
```

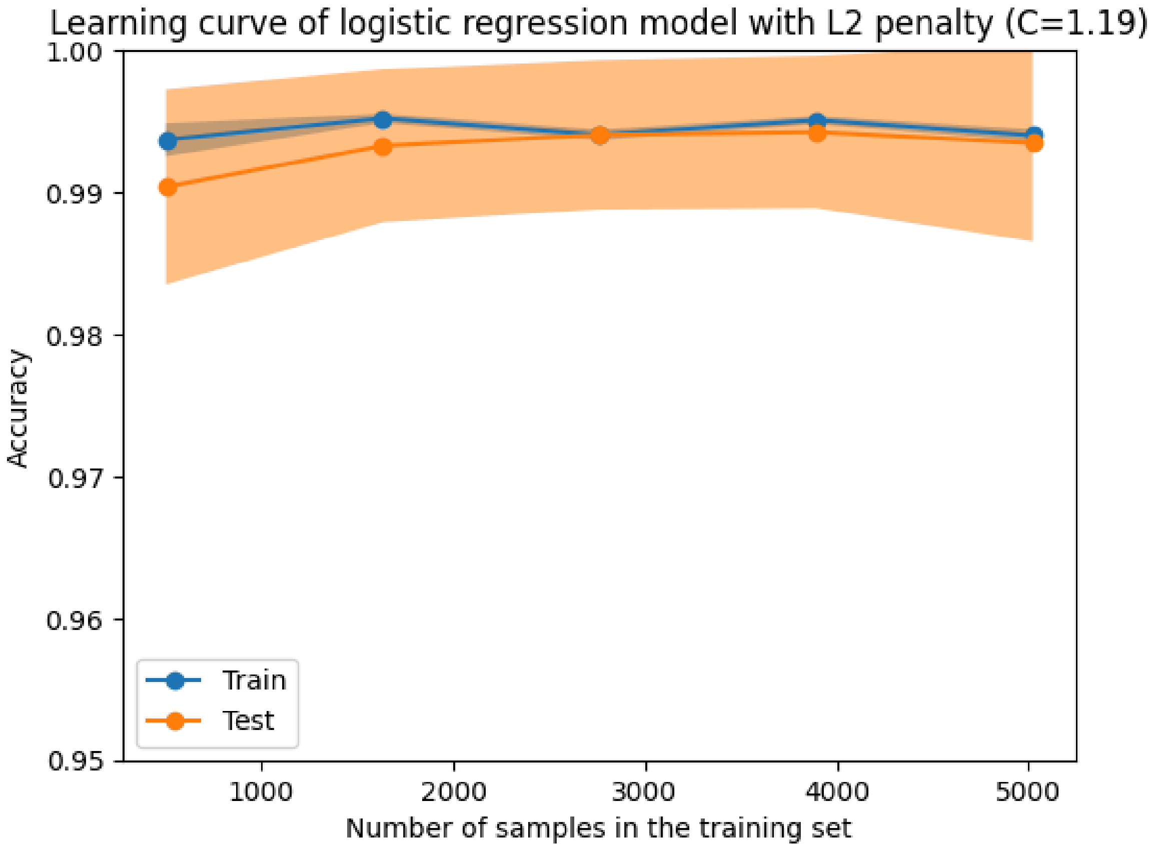
```
print("Performance on testing split :")
print_metrics(np.ravel(y_test), y_test_predicted)
```

```
Performance on testing split :
Mutual Information Score -> 0.5290096749774467
```

```
Accuracy Score -> 0.9946153846153846
F1-Score -> 0.9964194373401535
```

At first glance, the accuracy score on testing split is greater than the one computed on training split. It may indicate good fitting quality. We can assess it by displaying a learning curve.

```
plt.figure()
LearningCurveDisplay.from_estimator(
    logistic_regression,
    X_train,
    y_train,
    scoring="accuracy",
    score_name="Accuracy",
    line_kw={"marker": "o"},
    cv=30,
    n_jobs=-1,
)
plt.title(
    f"Learning curve of logistic regression model with L2 penalty
(C={logistic_regression.C:.3})"
)
plt.ylim((0.95, 1))
plt.show()
```



The learning curve helps us to assess the great fitting quality. We can now look into the details of the training split predictions performance.

```
print(classification_report(y_test, y_test_predicted))
```

```
              precision    recall  f1-score   support

           0       0.98      1.00      0.99       320
           1       1.00      0.99      1.00       980
```

```
    accuracy                           0.99      1300
   macro avg       0.99      1.00      0.99      1300
weighted avg       0.99      0.99      0.99      1300
```

As we can see, the distribution of the testing split is stratified from the distribution of the training split (75-25 between red and white wines). We can conclude from this matrix that the model is better able to predict the white wines (by an insignificant difference).

Basic model interpretability: inspecting the model

As a first step towards interpretability of the model predictions, let's take a look at the coefficients of the model. What is the most important feature from this perspective? How do you interpret positive or negative coefficients?

Is it compatible with what you have seen so far? Do you have an explanation why that might be?

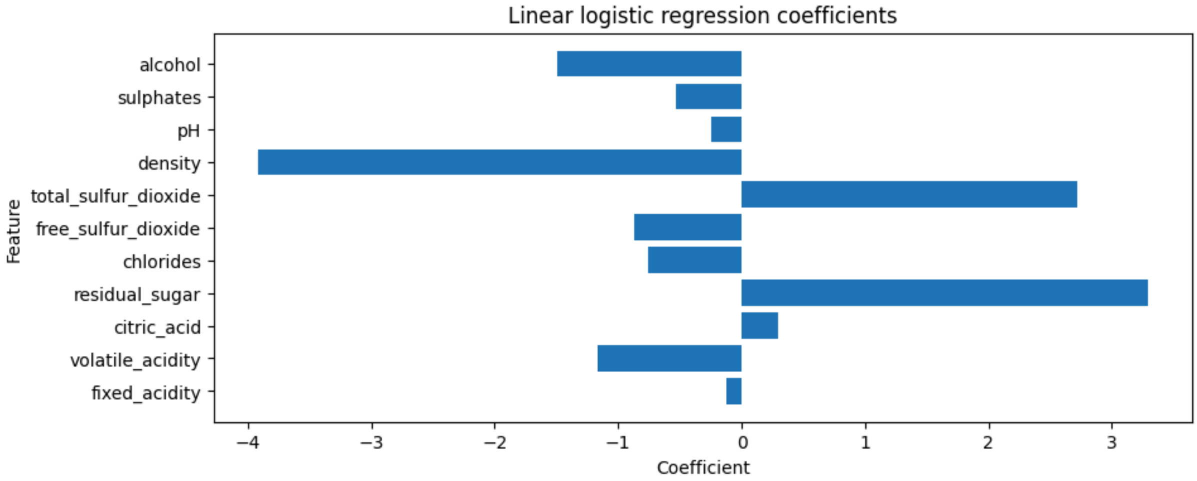
```
logistic_regression.coef_
```

```
array([[-0.12613291, -1.16420938,  0.29300419,  3.29014438, -0.75988363,
        -0.86621772,  2.725762  , -3.91647199, -0.24342432, -0.52701713,
        -1.49414518]])
```

Considering the fact that the model is linear and predicts the logarithm of the odds of having said target, the odds can be expressed as the exponential of the coefficients β_i. Hence, a positive coefficient indicates that a growth in said direction increases the odds of the outcome happening (multiplies by e^{β}), reciprocal being true.

Hence, a negative coefficient multiplies the odds of said outcome by $\frac{1}{e^{\beta}}$ which barely accounts for any ratio of said outcome happening.

```
coefficients = logistic_regression.coef_.flatten()

plt.figure(figsize=(10, 4))
plt.barh(X_train.columns, coefficients)

plt.xlabel("Coefficient")
plt.ylabel("Feature")
plt.title("Linear logistic regression coefficients")

plt.show()
```



As such, we can display the features whose coefficient is positive. The `citric_acid` and `total_sulfur_dioxide` features distribution shift from a wine color to another, for example, was clearly visible in the violin plot.

Removing features to test their importance

- What happens if you re-train a model, but remove the most important feature in the list?
- What happens if you re-train the model with a `l1` penalty and you use more regularization?
- Interpret the results you obtained above from the perspective of the business problem. What does it tell you about the key differences between a red and white wine?

```
X_train_reduced, X_test_reduced, y_train, y_test = train_test_split(
    X.drop(columns=["residual_sugar"]),
    color_encoded,
    train_size=0.8,
    stratify=color_encoded,
)
```

We can first standardize the different features.

```
X_train_reduced = standard_scaler.fit_transform(X_train_reduced)
X_test_reduced = standard_scaler.transform(X_test_reduced)
```

We can now train the model by assessing quality of accuracy on different L1 penalty parameter value. For this matter, the `liblinear` optimization solving algorithm will be used as it supports L1 penalty. Only stronger regularization than the one determined by the previous `GridSearch` will be tested.

```
logistic_regression_l1 = GridSearchCV(
    LogisticRegression(
        max_iter=10000, random_state=42, solver="liblinear", penalty="l1"
    ),
    param_grid={"C": list(map(lambda _: np.random.rand() * 5, range(0, 50)))},
    scoring="accuracy",
    n_jobs=-1,
)
```

```
logistic_regression_l1.fit(X_train_reduced, y_train)
```

```
GridSearchCV(estimator=LogisticRegression(max_iter=10000, penalty='l1',
                                          random_state=42, solver='liblinear'),
             n_jobs=-1,
             param_grid={'C': [3.8904168952084577, 3.5610752315867726,
                               3.0619756980200643, 4.993372197458522,
                               2.5102235992389623, 0.37035379145314185,
                               0.7822486168872739, 2.706228124966488,
                               3.357127826385529, 1.0234978908099068,
                               1.477259516364351, 1.092935389...
                               1.3033829944498416, 0.9621966158530004,
                               3.151975450327687, 1.61454875436708,
                               2.8261360941415097, 3.2646737847896308,
                               2.809391369810314, 0.9178576187981152,
                               3.6300177198906973, 1.4987196544348396,
                               2.2053205129976083, 1.7142245335383155,
                               0.6783836632889079, 4.409235435743836,
                               0.7182410828347668, 1.4842820386797357,
                               0.5620644145580178, 4.373945586933106, ...]},
             scoring='accuracy')
```

```
logistic_regression_l1 = logistic_regression_l1.best_estimator_
```

```
y_train_reduced_predicted = logistic_regression_l1.predict(X_train_reduced)
y_test_reduced_predicted = logistic_regression_l1.predict(X_test_reduced)
```

```
print("Performance on training split :")
print_metrics(y_train, y_train_reduced_predicted)
```

```
Performance on training split :
Mutual Information Score -> 0.5046404733257166
Accuracy Score -> 0.990379064845103
F1-Score -> 0.9936240754909462
```

```
print("Performance on testing split :")
print_metrics(y_test, y_test_reduced_predicted)
```

```
Performance on testing split :
Mutual Information Score -> 0.48994278009996195
Accuracy Score -> 0.9869230769230769
F1-Score -> 0.9913309535951046
```

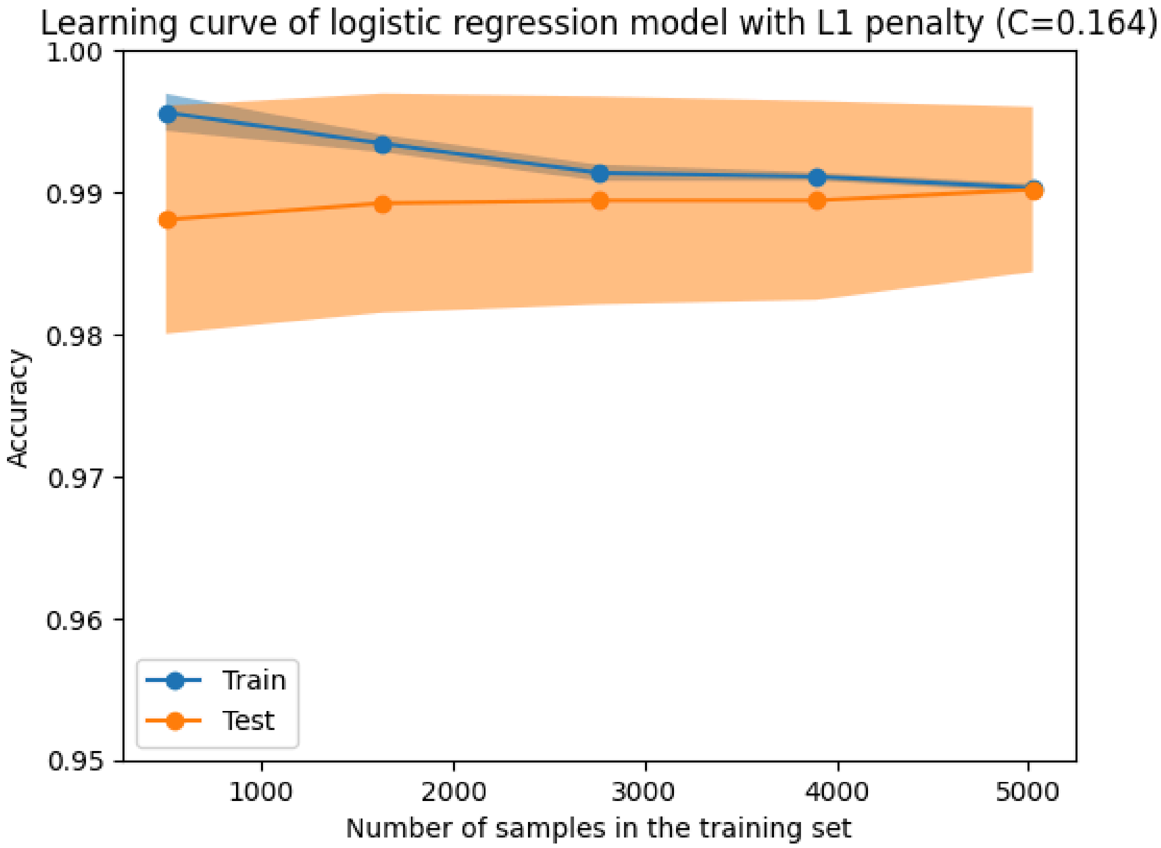
```
print(classification_report(y_test, y_test_reduced_predicted))
```

```
              precision    recall  f1-score   support

           0       0.97      0.97      0.97       320
           1       0.99      0.99      0.99       980

    accuracy                           0.99      1300
   macro avg       0.98      0.98      0.98      1300
weighted avg       0.99      0.99      0.99      1300
```

The different metrics, again, indicate greater performance of the model on testing split. We must now assess the fitting quality with multiple folds. The performance of the model, overall, is lower (by a negligeable amount).

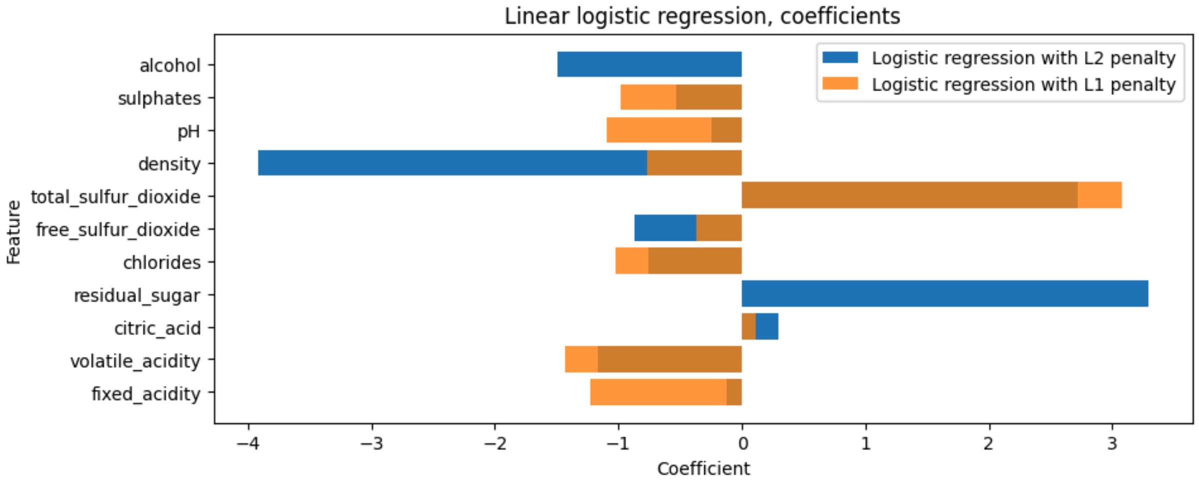
```
plt.figure()
LearningCurveDisplay.from_estimator(
    logistic_regression_l1,
    X_train_reduced,
    y_train,
    scoring="accuracy",
    score_name="Accuracy",
    line_kw={"marker": "o"},
    cv=30,
    n_jobs=-1,
)
plt.title(
    f"Learning curve of logistic regression model with L1 penalty
(C={logistic_regression_l1.C:.3})"
)
plt.ylim((0.95, 1))
plt.show()
```

```
<Figure size 640x480 with 0 Axes>
```



As we can see, the accuracy on training split seems to be better than the one computed on the testing split. However, the variance of the accuracy metric on testing split is greater. We can hence consider that the fitting quality is great.

```
coefficients_l2 = logistic_regression.coef_.flatten()
coefficients_l1 = logistic_regression_l1.coef_.flatten()

plt.figure(figsize=(10, 4))
plt.barh(
    X_train.columns,
    coefficients_l2,
    label="Logistic regression with L2 penalty",
)
plt.barh(
    X_train_reduced.columns,
    coefficients_l1,
    alpha=0.8,
    label="Logistic regression with L1 penalty",
)

plt.xlabel("Coefficient")
plt.ylabel("Feature")
plt.title("Linear logistic regression, coefficients")
plt.legend()

plt.show()
```



As we now notice, the greatest coefficient in the new model is the `total_sulfur_dioxide` feature ; the feature that previously ranked 2nd in the features coefficients ranking from greatest to smallest. In contrario, some features now have a greater coefficient (`density` for eg.).

In terms of business, it indicates us that the greatest features' coefficients indicate a bigger odds ratio and hence better help distinct red wines from white ones. Those features are mostly `residual_sugar` and `total_sulfur_dioxide`.

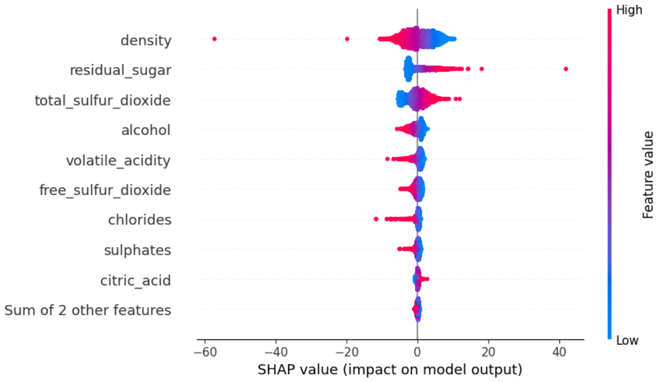
Using Shapely values

Now, use SHAP to explore how the model perceives a 'red' and 'white' wine.

- Use a `beeswarm` plot to analyze the influence of each feature on the model's output.
- What does the plot tell us about what makes a white wine 'white' and a red wine 'red'?

```
explainer = shap.Explainer(logistic_regression, X_train)
shap_values_logistic_regression = explainer(X_train)
```

```
shap.plots.beeswarm(shap_values_logistic_regression)
```

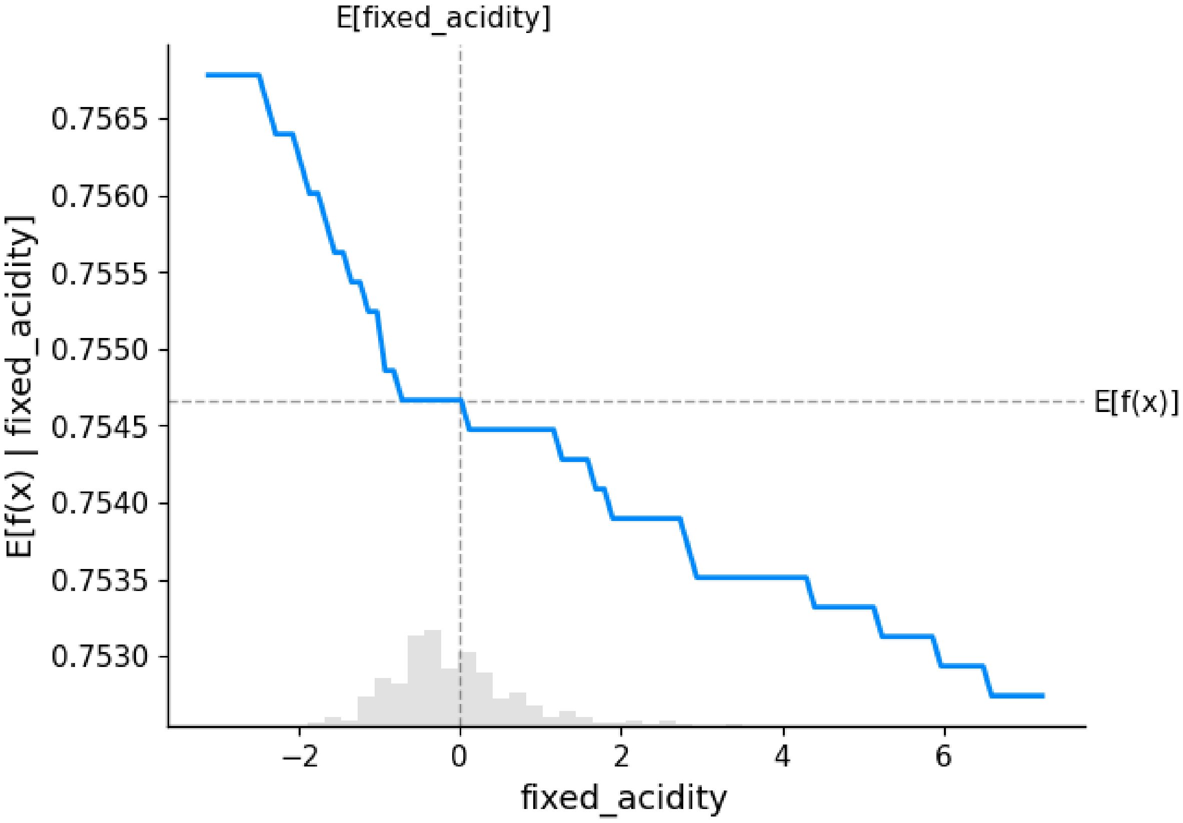


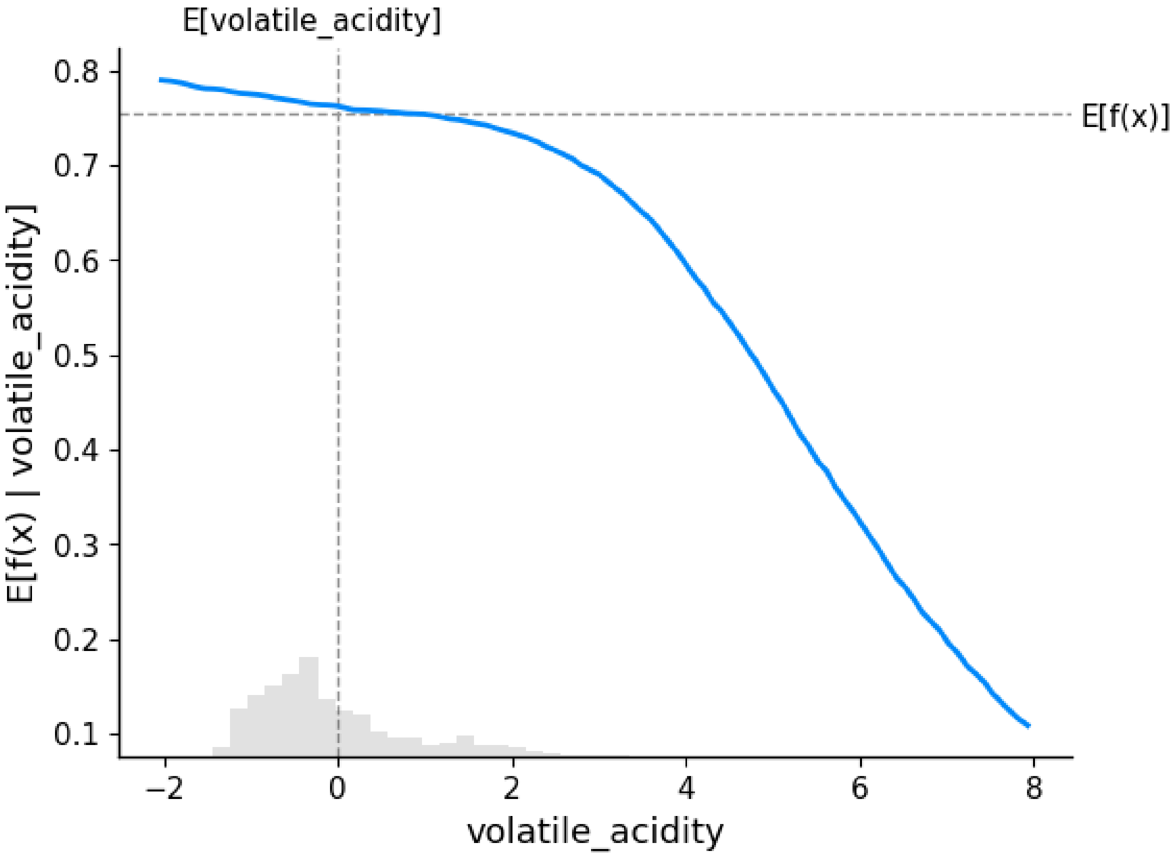
The plot shows that as certain features value increases respectively decreases, it has a certain impat on the model's output. The `residual_sugar` feature, for eg., has a bigger impact on model output as it increases whereas the `density` feature acts the other way around.

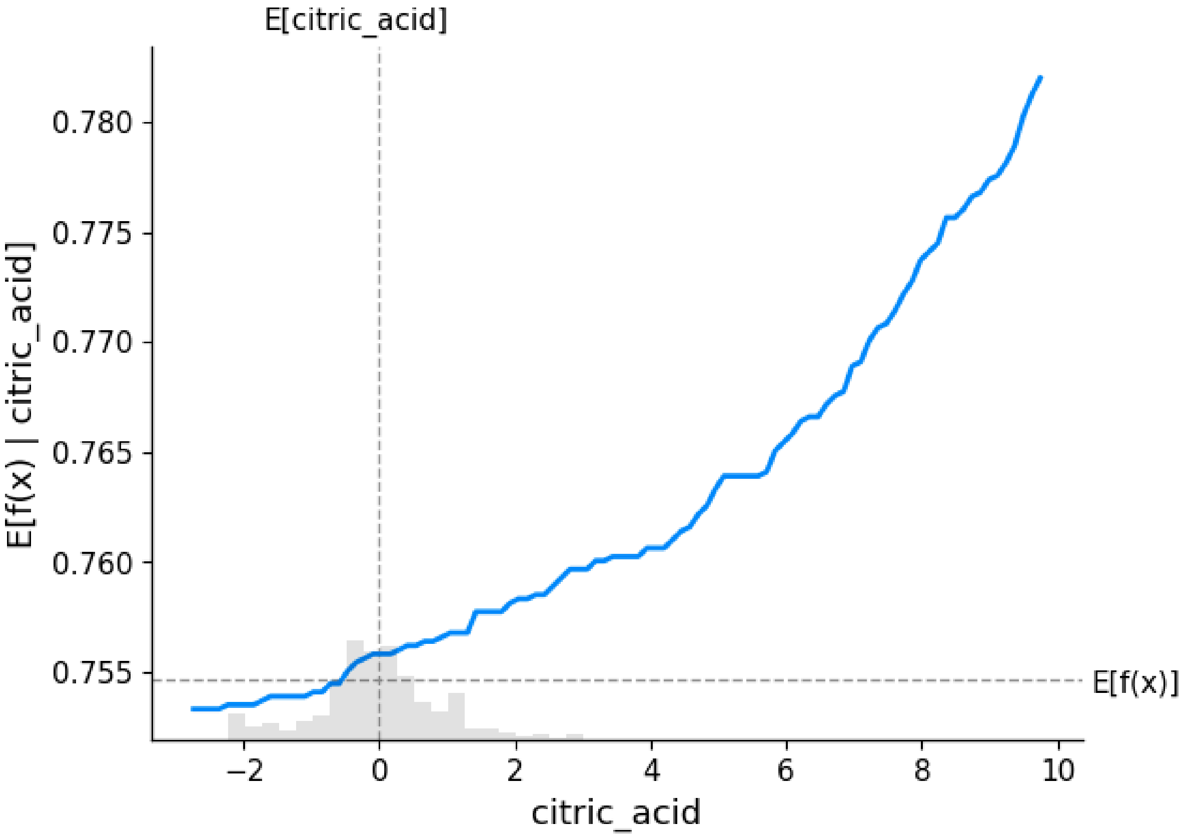
The positive impact indicates that when said feature increases respectively decreases, it has a greater impact (hence decision between red and white wine) output.

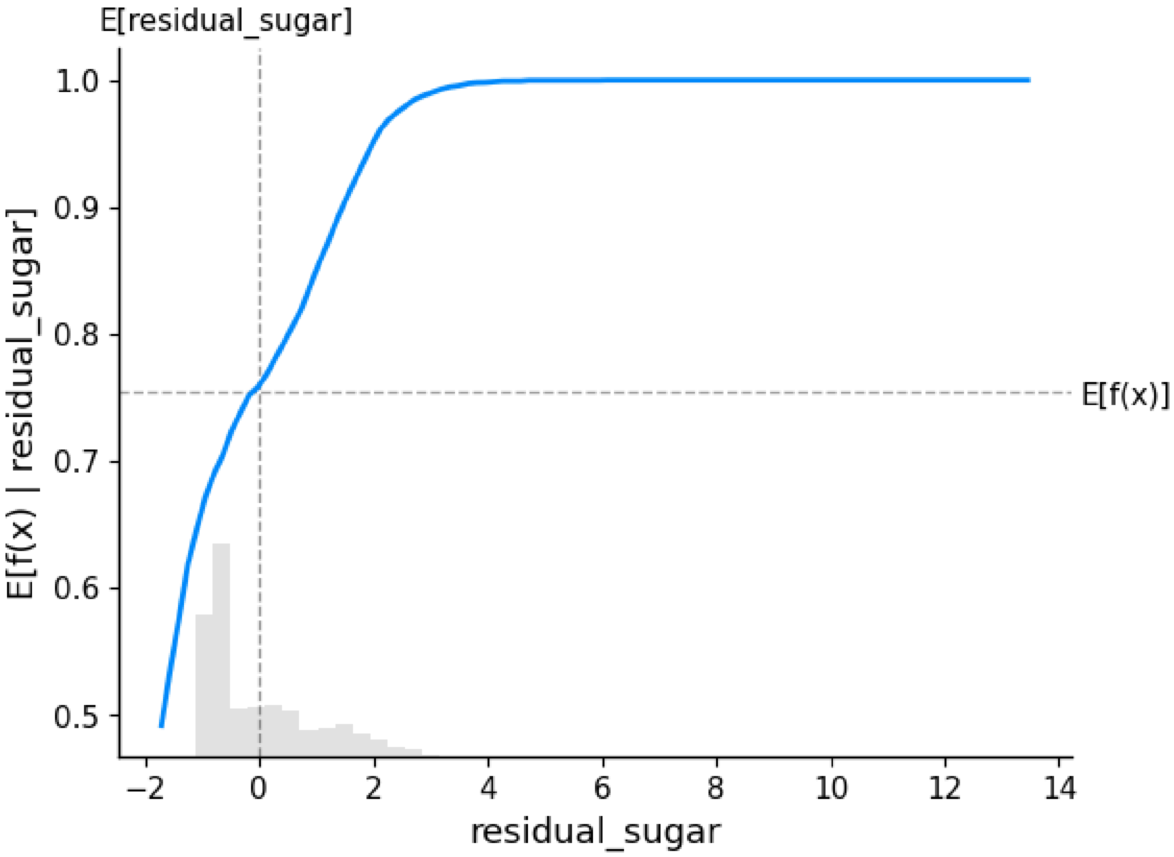
- Now use Partial Dependence Plots to see how the expected model output varies with the variation of each feature.

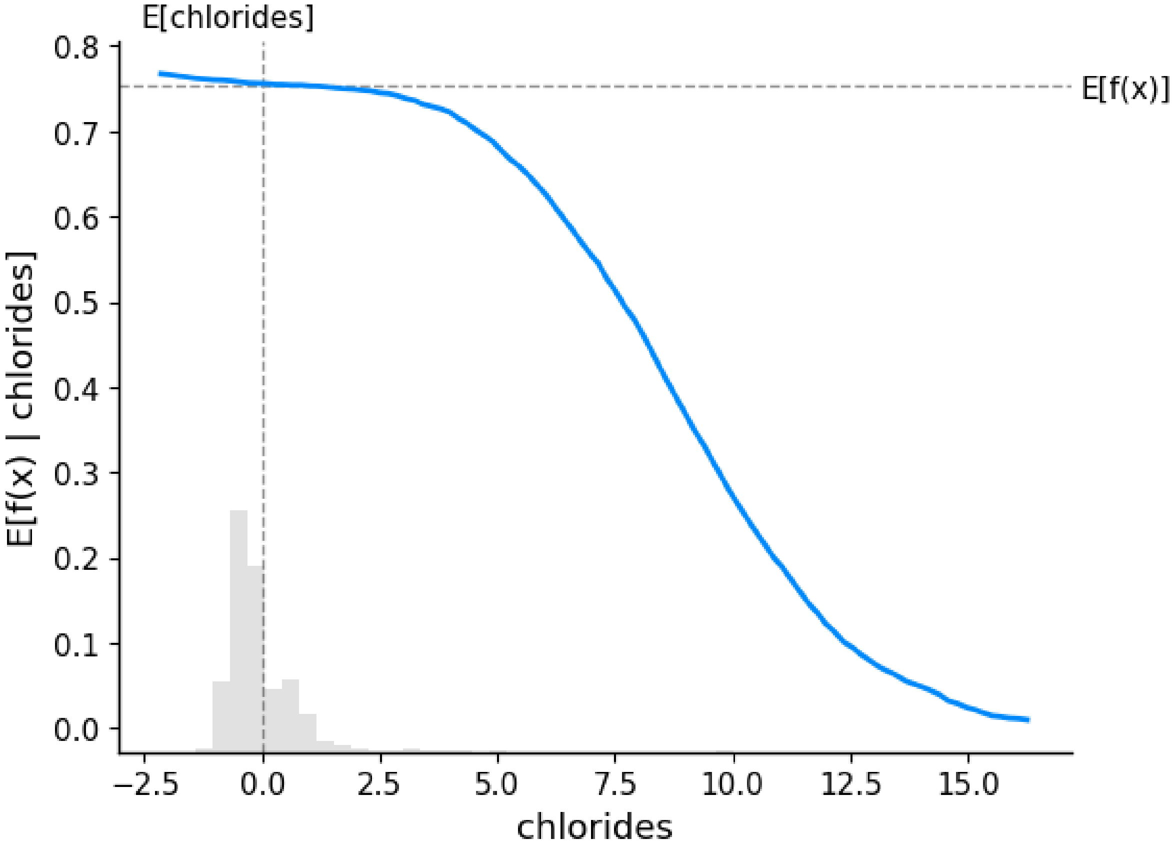
```python
for column in X_train.columns:
    shap.partial_dependence_plot(
        column,
        logistic_regression.predict,
        X_train,
        ice=False,  # when said feature is related to other features
        model_expected_value=True,
        feature_expected_value=True,
    )
```

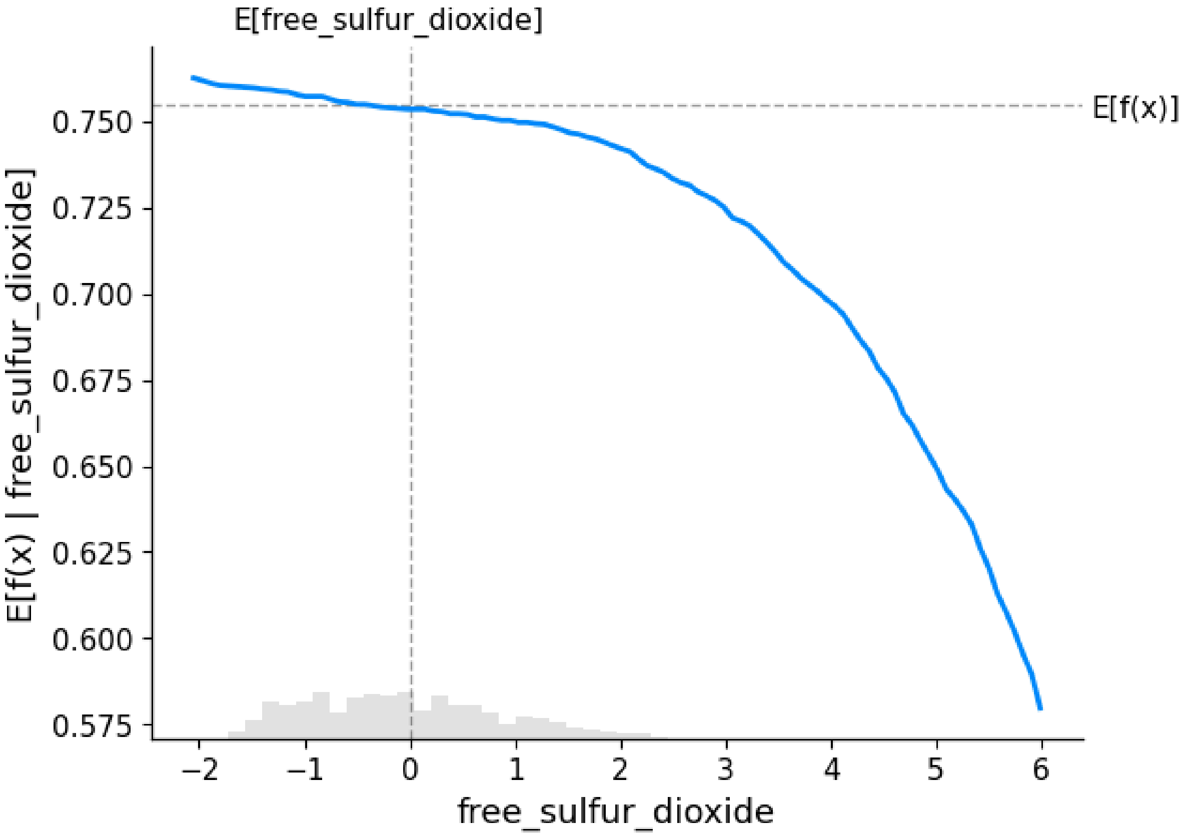


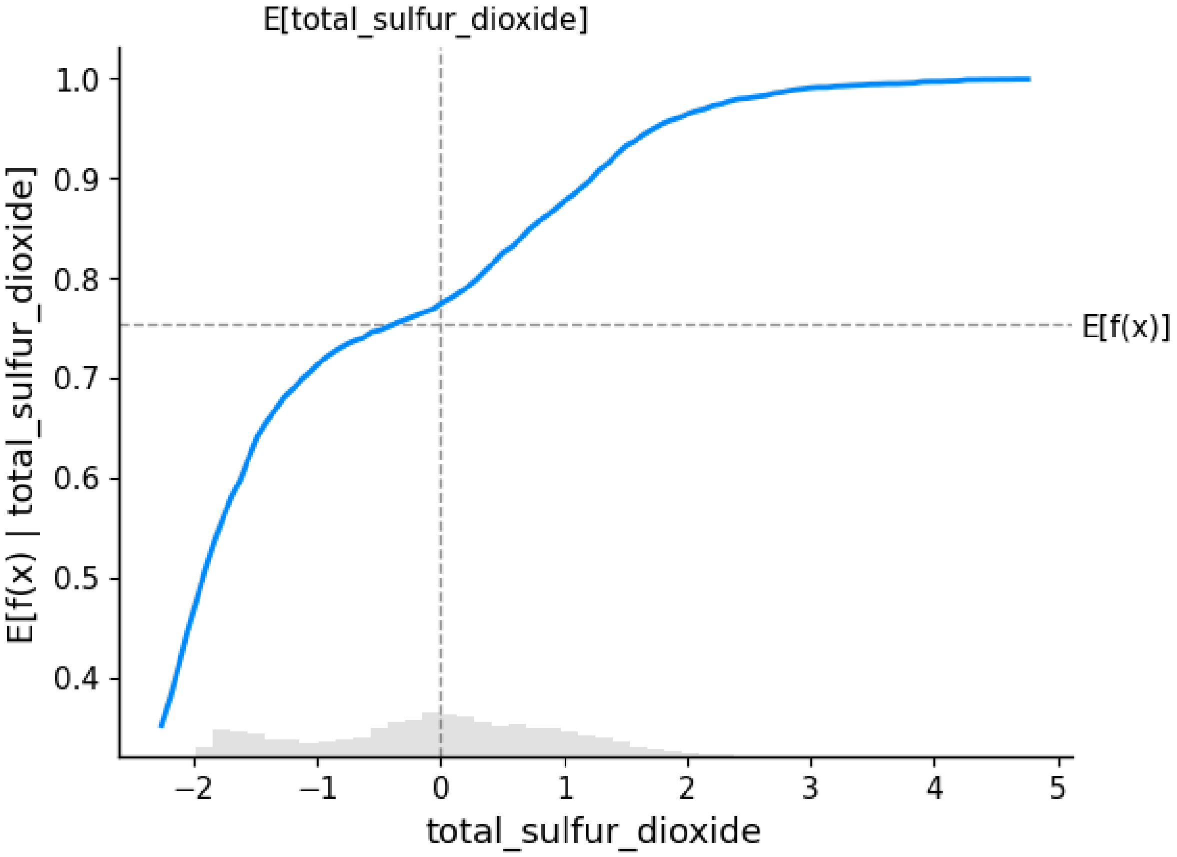

E[volatile_acidity]
E[f(x)]
E[f(x) | volatile_acidity]
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
−2
0
2
4
6
8
volatile_acidity

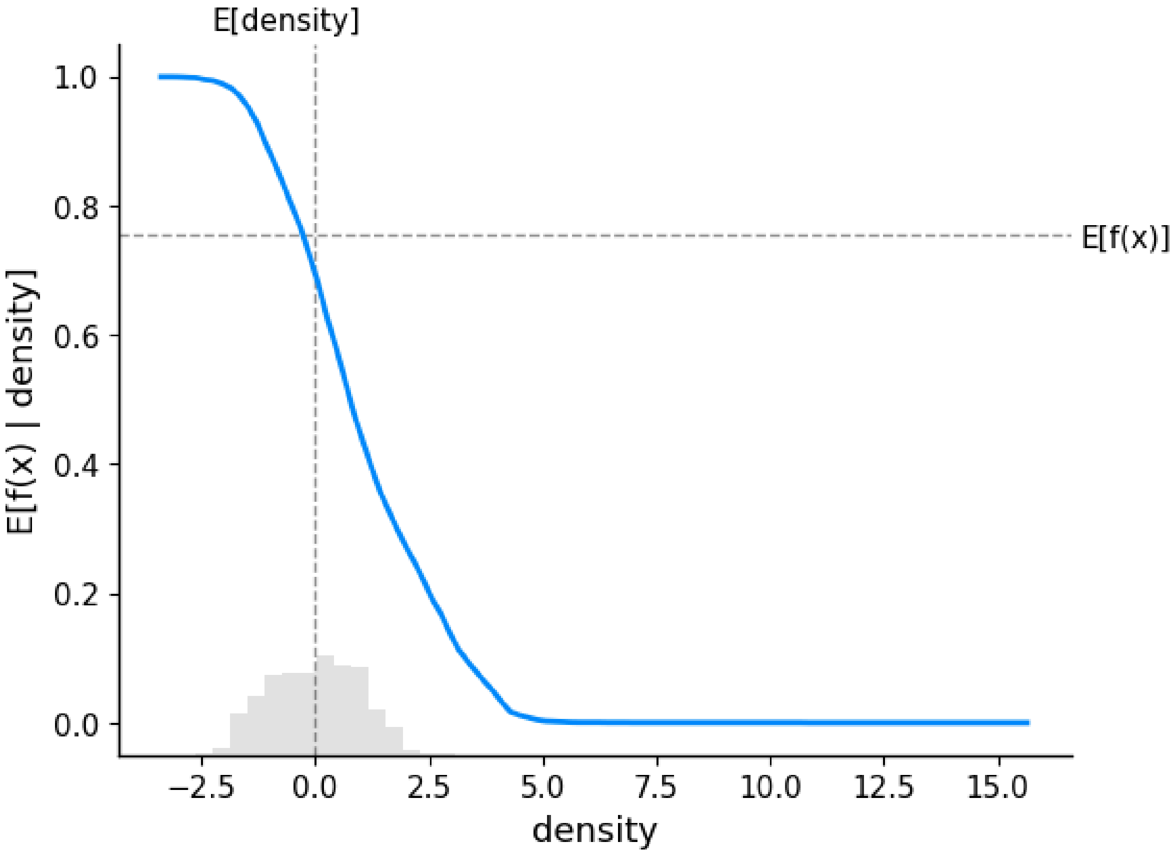

E[citric_acid]
E[f(x)]
E[f(x) | citric_acid]
0.780
0.775
0.770
0.765
0.760
0.755
−2
0
2
4
6
8
10
citric_acid

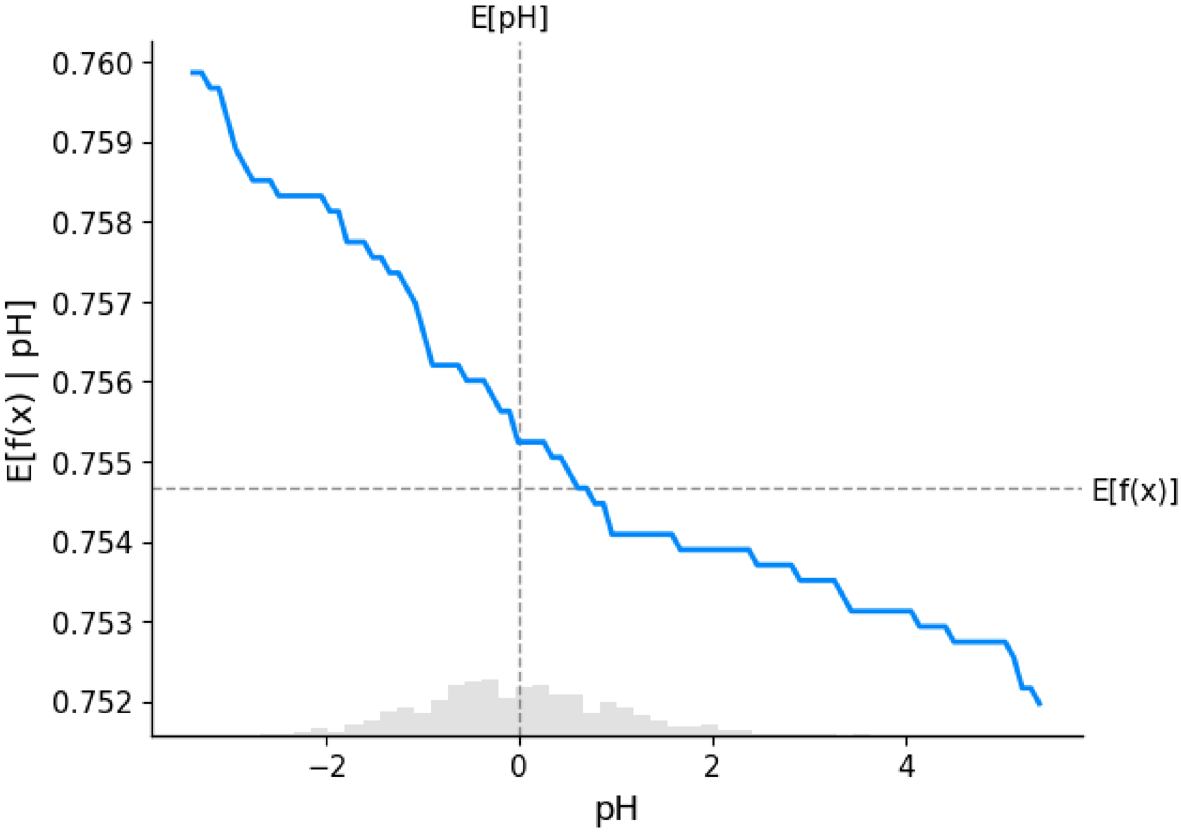

E[residual_sugar]
1.0
0.9
0.8
0.7
0.6
0.5
E[f(x) | residual_sugar]
E[f(x)]
−2
0
2
4
6
8
10
12
14
residual_sugar

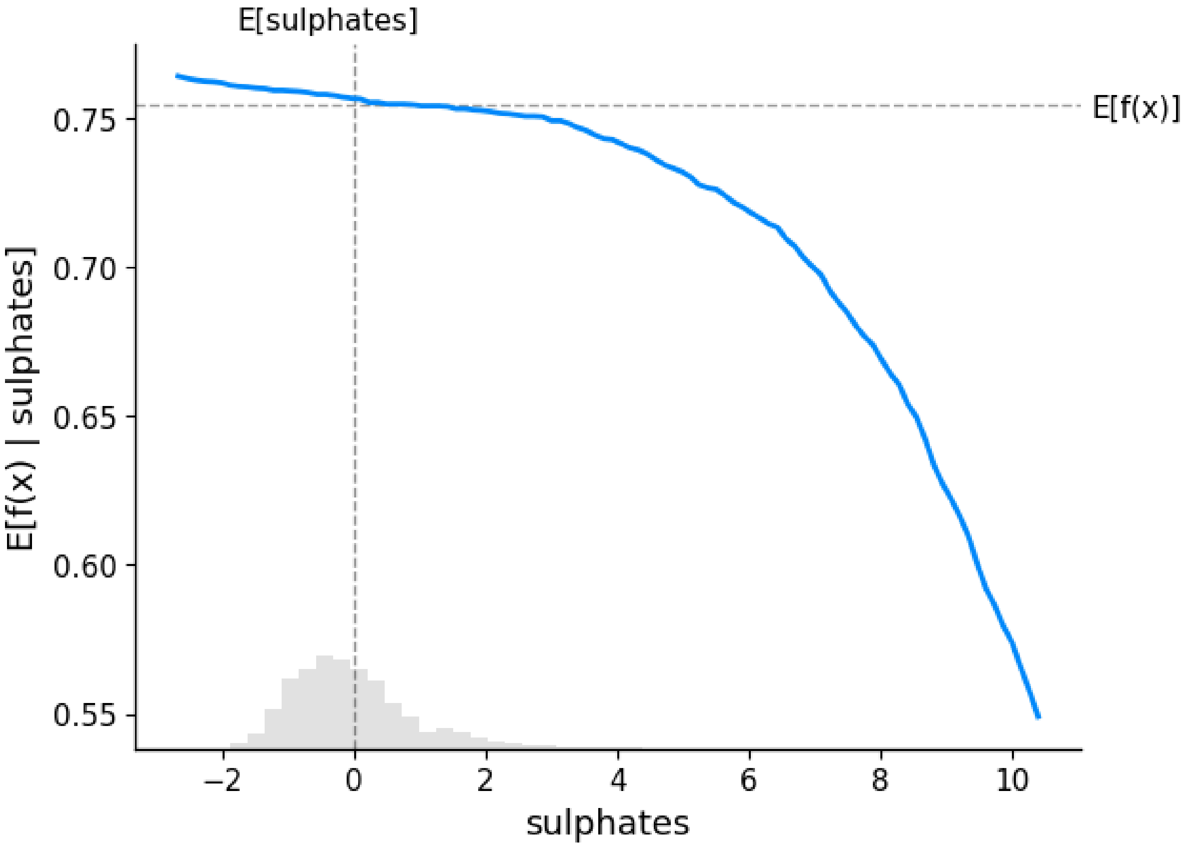

E[chlorides]
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
E[f(x) | chlorides]
E[f(x)]
−2.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
chlorides

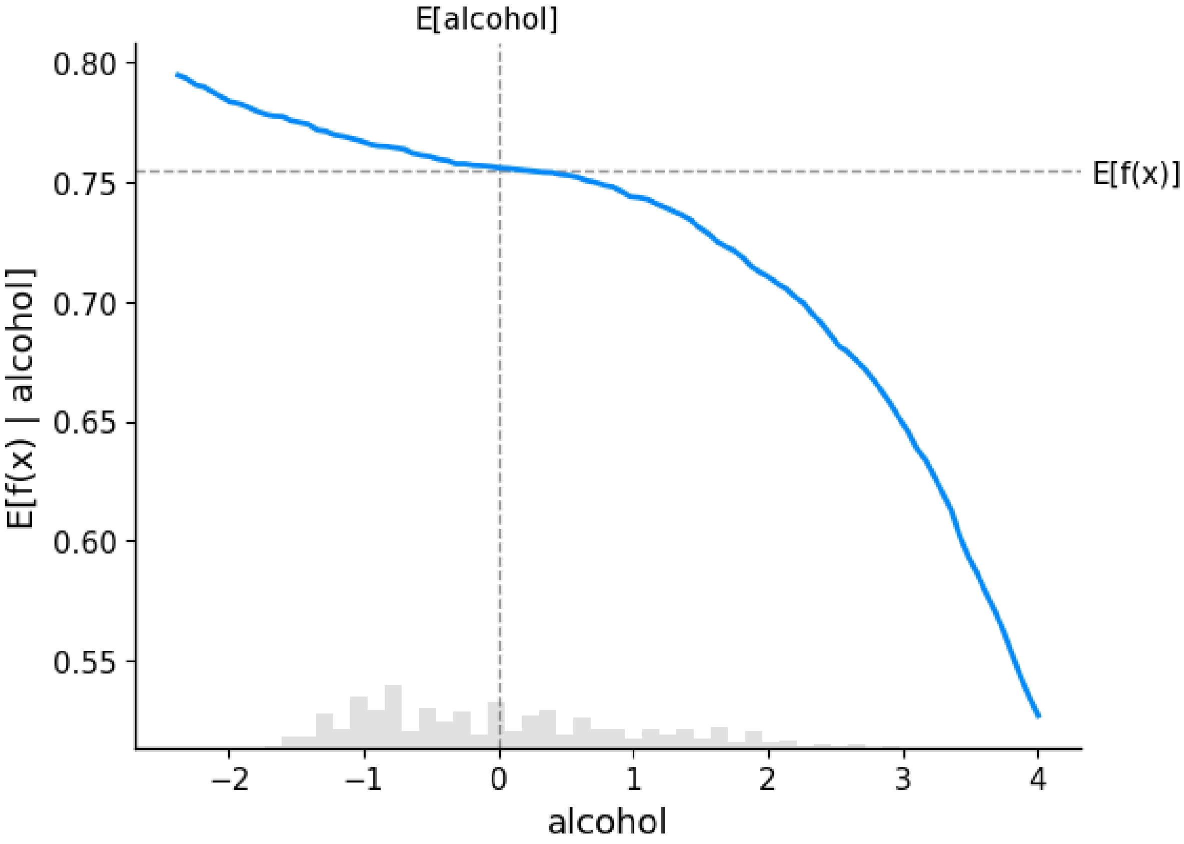

E[free_sulfur_dioxide]
E[f(x)]
E[f(x) | free_sulfur_dioxide]
0.750
0.725
0.700
0.675
0.650
0.625
0.600
0.575
−2
−1
0
1
2
3
4
5
6
free_sulfur_dioxide


E[total_sulfur_dioxide]
E[f(x)]
E[f(x) | total_sulfur_dioxide]
1.0
0.9
0.8
0.7
0.6
0.5
0.4
−2
−1
0
1
2
3
4
5
total_sulfur_dioxide


E[density]
E[f(x)]
E[f(x) | density]
1.0
0.8
0.6
0.4
0.2
0.0
−2.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
density


E[pH]
E[f(x)]
E[f(x) | pH]
0.760
0.759
0.758
0.757
0.756
0.755
0.754
0.753
0.752
−2
0
2
4
pH


E[sulphates]
E[f(x)]
E[f(x) | sulphates]
0.75
0.70
0.65
0.60
0.55
−2
0
2
4
6
8
10
sulphates


E[alcohol]
E[f(x)]
E[f(x) | alcohol]
0.80
0.75
0.70
0.65
0.60
0.55
−2
−1
0
1
2
3
4
alcohol

The different partial dependence plots display the model's output expectancy when changing the value of said feature.

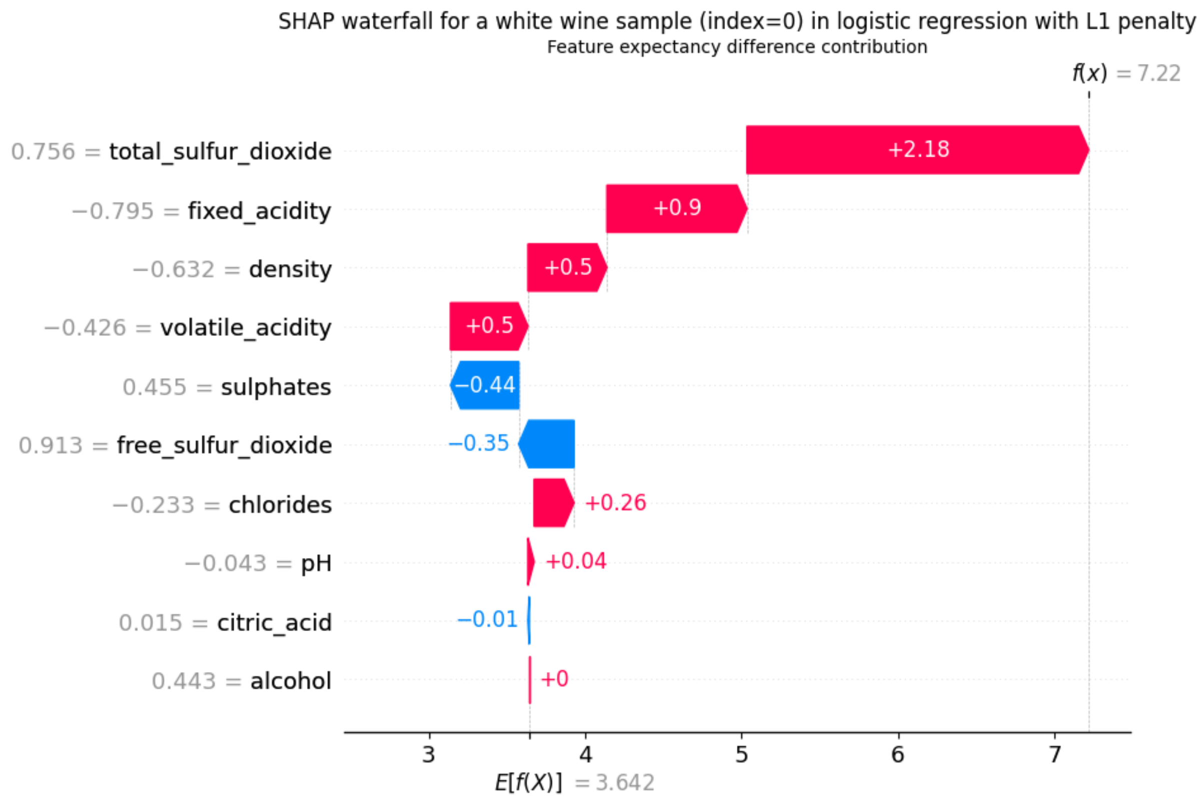
- Now use a waterfall diagram on a specific red and white wine and see how the model has made this specific prediction.

```
red_sample_index = -1
white_sample_index = -1

for index, (true, predicted) in enumerate(
    zip(
        label_encoder.inverse_transform(y_test),
        label_encoder.inverse_transform(y_test_reduced_predicted),
    )
):
    if (true == predicted) and (true == "red") and (red_sample_index == -1):
        red_sample_index = index

    if (true == predicted) and (true == "white") and (white_sample_index == -1):
        white_sample_index = index

    if red_sample_index != -1 and white_sample_index != -1:
        break

print(f"Correctly predicted red sample index: {red_sample_index}")
print(f"Correctly predicted white sample index: {white_sample_index}")
```

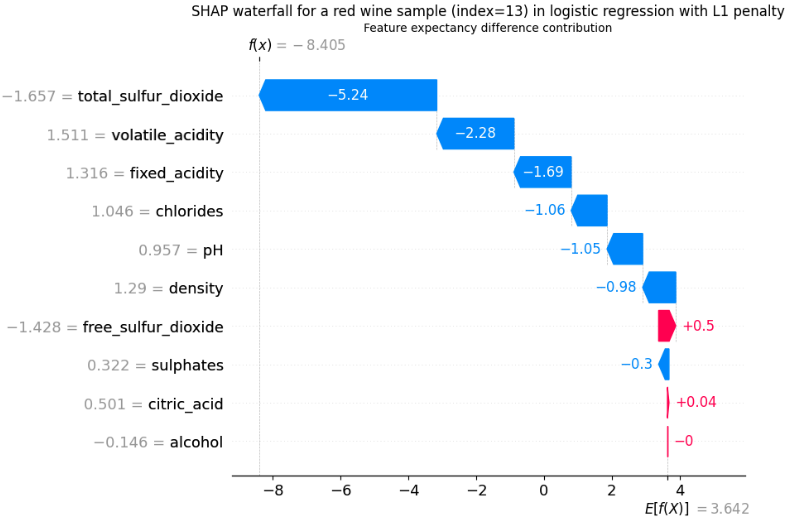
```
Correctly predicted red sample index: 13
Correctly predicted white sample index: 0
```

As we can see, the samples 0 and 1 of the testing split were both accurately predicted to respectively white and red. We can hence use them as specific samples.

```
explainer = shap.Explainer(logistic_regression_l1, X_test_reduced)
shap_values_logistic_regression_l1 = explainer(X_test_reduced)

plt.figure()
shap.plots.waterfall(shap_values_logistic_regression_l1[white_sample_index], show=False)
plt.title(
    f"SHAP waterfall for a white wine sample (index={white_sample_index}) in logistic
regression with L1 penalty"
)
plt.xlabel("Feature expectancy difference contribution")
plt.show()
```



```
plt.figure()
shap.plots.waterfall(shap_values_logistic_regression_l1[red_sample_index], show=False)
plt.title(
    f"SHAP waterfall for a red wine sample (index={red_sample_index}) in logistic regression
with L1 penalty"
)
plt.xlabel("Feature expectancy difference contribution")
plt.show()
```



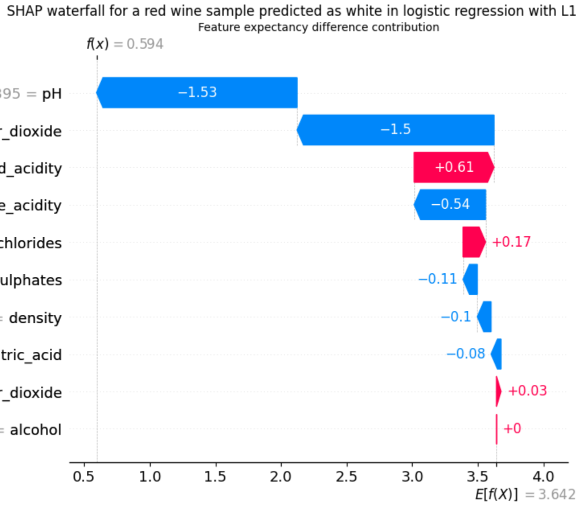
As we can see, in both cases, the `total_sulfur_dioxide` greatly contributed to make the prediction of the model differ from its expectancy. The `pH` feature when considering the white wine sample clearly shifted towards a `red` prediction as it did too in the red wine sample. In the red wine sample, the value of the `total_sulfur_dioxide` feature greatly contributed to predict the wine being red as its value greatly differs from the distribution of the white wines one.

- Now, let's take an example where the model has made an incorrect prediction, and see how it made this prediction.

```
falsely_predicted_index = 0

true_color = None
predicted_color = None

for index in range(y_test.shape[0]):
    if y_test[index] != y_test_reduced_predicted[index]:

        print(f"Sample index whose color is incorrectly predicted -> {index}")
        print(
            f"True: {label_encoder.inverse_transform(y_test[[index]])}, Predicted:
{label_encoder.inverse_transform(y_test_reduced_predicted[[index]])}"
        )
        falsely_predicted_index = index

        true_color = label_encoder.inverse_transform(y_test[[index]])[0]
        predicted_color = label_encoder.inverse_transform(
            y_test_reduced_predicted[[index]]
        )[0]

        break
```

```
Sample index whose color is incorrectly predicted -> 41
True: ['red'], Predicted: ['white']
```

```
plt.figure()
shap.plots.waterfall(
    shap_values_logistic_regression_l1[falsely_predicted_index], show=False
)
plt.title(
    f"SHAP waterfall for a {true_color} wine sample predicted as {predicted_color} in logistic
regression with L1 penalty"
)
plt.xlabel("Feature expectancy difference contribution")
plt.show()
```



As we can see, the `volatile_acidity` of our wine sample wrongly directed the model into a bad prediction of its color. We can consider such a sample to be an outlier in said distribution.

Good vs Bad classification

We are going to work on a binary classification problem, where all wines with a quality higher than 6 are considered as "good" and other are considered as "bad".

- Prepare a dataset with a new column `binary_quality` that corresponds to the above definition.

One question that we might ask right away is:

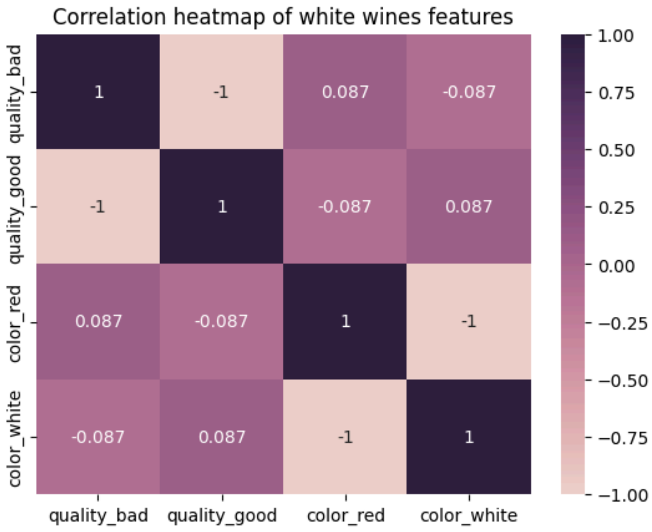
- Is there any correlation of the quality and the color of the wine?

Ideally, there should be almost none. Why could it be a problem otherwise?

```
binary_quality = pd.DataFrame(y["quality"].apply(lambda x: "good" if x > 6 else "bad"))
```

To compute correlation between both series, we need to first encode them.

```
one_hot_encoder_binary_quality = OneHotEncoder(sparse_output=False).set_output(
    transform="pandas"
)
binary_quality_encoded = one_hot_encoder_binary_quality.fit_transform(binary_quality)

one_hot_encoder_binary_color = OneHotEncoder(sparse_output=False).set_output(
    transform="pandas"
)
binary_color = one_hot_encoder_binary_color.fit_transform(color)

X_color_quality = pd.concat([X, binary_quality_encoded, binary_color], axis=1)

plt.figure()
sns.heatmap(
    X_color_quality[["quality_bad", "quality_good", "color_red", "color_white"]].corr(),
    cmap=sns.cubehelix_palette(as_cmap=True),
    annot=True,
)
plt.title("Correlation heatmap of white wines features")
plt.show()
```



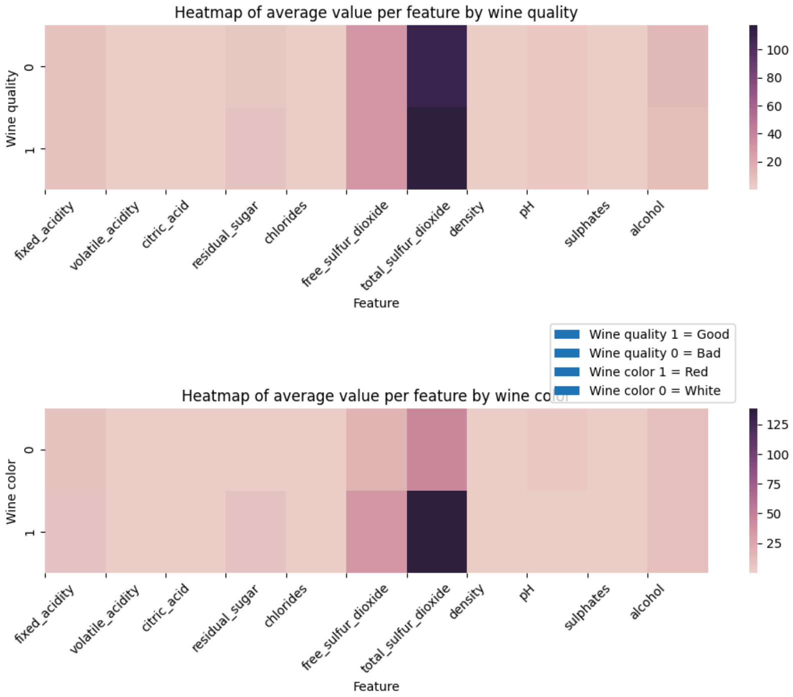
As seen on the heat map, the correlation between a good wine and white wines is positively correlated while a bad wine quality is positively correlated with red wines. The correlation being so small, we can assume that the color of a wine hasn't any predictive impact on its quality, by itself at least. If it were to be so, one could simply tell if a wine is of good quality by assessing its color.

If it turns out that there are significantly more bad red wines than bad white wines or vice versa, what are the implications for your analysis?

- Plot a heat map of the mean feature value for bad and good wines, like we did before for red and white wines.
- Plot two heat maps, one for red and white wines. How do they differ? What kind of issue can it cause?

```python
X_color_quality_good = X_color_quality[X_color_quality["quality_good"] == 1.0].drop(
    columns=["quality_good", "quality_bad", "color_red", "color_white"]
)
X_color_quality_bad = X_color_quality[X_color_quality["quality_good"] == 0.0].drop(
    columns=["quality_good", "quality_bad", "color_red", "color_white"]
)

X_color_quality_good_averaged = X_color_quality_good.select_dtypes(
    include=["float64"]
).agg("mean")
X_color_quality_bad_averaged = X_color_quality_bad.select_dtypes(
    include=["float64"]
).agg("mean")

X_color_quality_red = X_color_quality[X_color_quality["color_red"] == 1.0].drop(
    columns=["quality_good", "quality_bad", "color_red", "color_white"]
)
X_color_quality_white = X_color_quality[X_color_quality["color_red"] == 0.0].drop(
    columns=["quality_good", "quality_bad", "color_red", "color_white"]
)

X_color_quality_red_averaged = X_color_quality_red.select_dtypes(
    include=["float64"]
).agg("mean")
X_color_quality_white_averaged = X_color_quality_white.select_dtypes(
    include=["float64"]
).agg("mean")

fig, axes = plt.subplots(2, 1, figsize=(10, 8))

sns.heatmap(
    np.vstack((X_color_quality_good_averaged, X_color_quality_bad_averaged)),
    cmap=sns.cubehelix_palette(as_cmap=True),
    ax=axes[0],
)
axes[0].set_xlabel("Feature")
axes[0].set_ylabel("Wine quality")
axes[0].set_xticks(np.arange(len(X_color_quality_good_averaged.index)))
axes[0].set_xticklabels(X_color_quality_good_averaged.index.array, rotation=45)
axes[0].set_title("Heatmap of average value per feature by wine quality")

sns.heatmap(
    np.vstack((X_color_quality_red_averaged, X_color_quality_white_averaged)),
    cmap=sns.cubehelix_palette(as_cmap=True),
    ax=axes[1],
)
axes[1].set_xlabel("Feature")
axes[1].set_ylabel("Wine color")
axes[1].set_xticks(np.arange(len(X_color_quality_red_averaged.index)))
axes[1].set_xticklabels(X_color_quality_red_averaged.index.array, rotation=45)
axes[1].set_title("Heatmap of average value per feature by wine color")

legend_patches = [
    mpatches.Patch(label="Wine quality 1 = Good"),
    mpatches.Patch(label="Wine quality 0 = Bad"),
    mpatches.Patch(label="Wine color 1 = Red"),
    mpatches.Patch(label="Wine color 0 = White"),
]
```

```
plt.legend(handles=legend_patches, bbox_to_anchor=(1.05, 1), loc="lower right")

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



At first glance, the good wines (marked as `1`) have a greater mean `total_sulfur_dioxide` and so do the red wines (marked as `1`). The average value for the distribution of the good wines features and red wines features show similarity. This similarity due to the greater mean `total_sulfur_dioxide`, however, does not enable predicting a wine's quality based on this single feature as it won't yield great results. If a good wine was associated with its color, then, businesses would only buy wines based on their color.

It is a lot more difficult now to tell apart good from bad wines. Let's turn to a more complex model, which is a Gradient Boosting Trees. For the sake of interpretability, design your notebook so that you can easily filter on only white and red wines and perform again the entire procedure.

Let's first train a XGBClassifier model to distinguish between good and bad wines. Make sure to use the same best-practices (train/test split, cross-validation) as we did before. Note that the regularization of the GBTs is a lot more complex than for Logistic Regression. Test the following parameters:

```
param_grid = {
  "max_depth": [3, 4, 5],  # Focus on shallow trees to reduce complexity
  "learning_rate": [0.01, 0.05, 0.1],  # Slower learning rates
  "n_estimators": [50, 100],  # More trees but keep it reasonable
  "min_child_weight": [1, 3],  # Regularization to control split thresholds
  "subsample": [0.7, 0.9],  # Sampling rate for boosting
  "colsample_bytree": [0.7, 1.0],  # Sampling rate for columns
  "gamma": [0, 0.1],  # Regularization to penalize complex trees
}
```

Split the data.

```
X_train, X_test, y_train, y_test = train_test_split(
    pd.concat([X, color], axis=1),
    binary_quality,
    train_size=0.8,
    stratify=binary_quality,
)
```

We now must encode the categorical feature `color` and standardize the numerical features.

```
X_train = pd.concat(
    [
        standard_scaler.fit_transform(X_train.drop(columns=["color"])),
        X_train["color"],
    ],
    axis=1,
)

X_test = pd.concat(
    [
        standard_scaler.transform(X_test.drop(columns=["color"])),
        X_test["color"],
    ],
    axis=1,
)
```

```
encoded_binary_color = one_hot_encoder_binary_color.fit_transform(X_train[["color"]])
encoded_binary_color_test = one_hot_encoder_binary_color.transform(X_test[["color"]])
```

```
X_train = pd.concat([X_train, encoded_binary_color], axis=1).drop(columns=["color"])
X_test = pd.concat([X_test, encoded_binary_color_test], axis=1).drop(columns=["color"])
```

```
X_train.describe().loc[["mean", "std"]].agg("mean", axis=1)
```

```
mean    0.076923
std     0.912174
dtype: float64
```

```
X_test.describe().loc[["mean", "std"]].agg("mean", axis=1)
```

```
mean    0.086788
std     0.934484
dtype: float64
```

```
y_train = label_encoder.fit_transform(np.ravel(y_train))
label_encoder.inverse_transform([0, 1])
```

```
array(['bad', 'good'], dtype=object)
```

Bad wines are encoded as 0 and good wines as 1.

We can now train the model.

```
y_test = label_encoder.transform(np.ravel(y_test))
```

```
gbt = GridSearchCV(
    XGBClassifier(random_state=42),
    param_grid={
        "max_depth": [3, 4, 5],  # Focus on shallow trees to reduce complexity
        "learning_rate": [0.01, 0.05, 0.1],  # Slower learning rates
        "n_estimators": [50, 100],  # More trees but keep it reasonable
        "min_child_weight": [1, 3],  # Regularization to control split thresholds
        "subsample": [0.7, 0.9],  # Sampling rate for boosting
        "colsample_bytree": [0.7, 1.0],  # Sampling rate for columns
        "gamma": [0, 0.1],  # Regularization to penalize complex trees
    },
    scoring="f1",
    n_jobs=-1,
)
```

```
gbt.fit(X_train, y_train)
```

```
GridSearchCV(estimator=XGBClassifier(base_score=None, booster=None,
                                     callbacks=None, colsample_bylevel=None,
                                     colsample_bynode=None,
                                     colsample_bytree=None, device=None,
                                     early_stopping_rounds=None,
                                     enable_categorical=False, eval_metric=None,
                                     feature_types=None, gamma=None,
                                     grow_policy=None, importance_type=None,
                                     interaction_constraints=None,
                                     learning_rate=None, max_b...
                                     max_leaves=None, min_child_weight=None,
                                     missing=nan, monotone_constraints=None,
                                     multi_strategy=None, n_estimators=None,
                                     n_jobs=None, num_parallel_tree=None,
                                     random_state=42, ...),
             n_jobs=-1,
             param_grid={'colsample_bytree': [0.7, 1.0], 'gamma': [0, 0.1],
                         'learning_rate': [0.01, 0.05, 0.1],
                         'max_depth': [3, 4, 5], 'min_child_weight': [1, 3],
                         'n_estimators': [50, 100], 'subsample': [0.7, 0.9]},
             scoring='f1')
```

```
gbt = gbt.best_estimator_
```

- Analyze the results (test and train), validate whether there is overfitting.

```
y_train_predicted = gbt.predict(X_train)
y_test_predicted = gbt.predict(X_test)
```

```
print("Performance of GBT on training split (quality classification) :")
print_metrics(y_train, y_train_predicted)
```

```
Performance of GBT on training split (quality classification) :
Mutual Information Score -> 0.22776000074812197
Accuracy Score -> 0.9241870309794112
F1-Score -> 0.7818383167220376
```

```
print("Performance of GBT on testing split (quality classification) :")
print_metrics(y_test, y_test_predicted)
```

```
Performance of GBT on testing split (quality classification) :
Mutual Information Score -> 0.11007382227893486
Accuracy Score -> 0.8630769230769231
F1-Score -> 0.5954545454545455
```

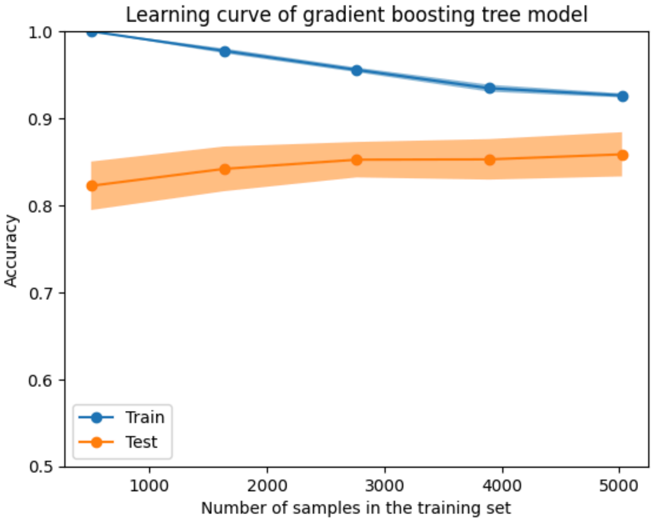
At first glance, the accuracy score of testing split seems no too far off from the training split accuracy but could still indicate potential overfitting as the mutual information scores are a bit further from training split to testing split. The true labels and predictions don't share much information. The F1-Score, too, is very bad.

```
print(classification_report(y_test, y_test_predicted))
```

```
              precision    recall  f1-score   support

           0       0.89      0.95      0.92      1044
           1       0.71      0.51      0.60       256

    accuracy                           0.86      1300
   macro avg       0.80      0.73      0.76      1300
weighted avg       0.85      0.86      0.85      1300
```

As we can see, the model is better able to retrieve and predict bad wines whereas it is worse (especially for the retrieval) at doing so on good wines. Even though the model's hyper parameters were chosen to optimize the F1-score metric, it is still very bad on the good wines.

```
plt.figure()
LearningCurveDisplay.from_estimator(
    gbt,
    X_train,
    y_train,
    scoring="accuracy",
    score_name="Accuracy",
    line_kw={"marker": "o"},
    cv=30,
    n_jobs=-1,
)
plt.title(f"Learning curve of gradient boosting tree model")
plt.ylim((0.5, 1))
plt.show()
```

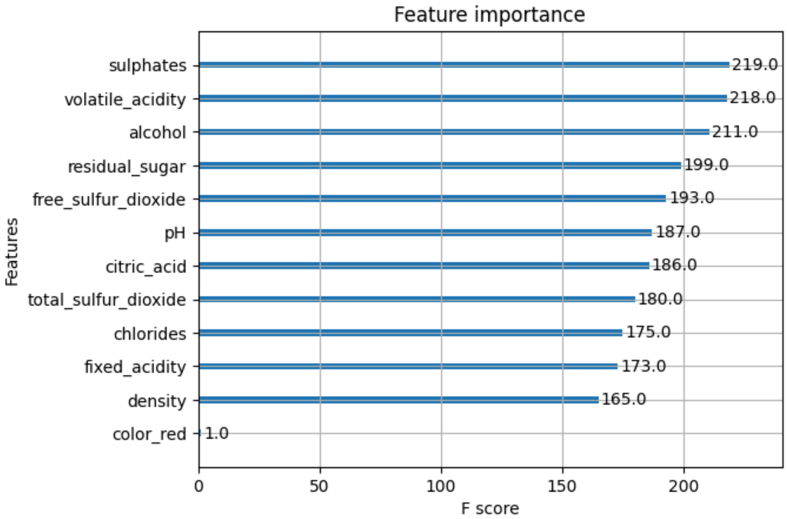


As seen on the plot, the testing curve seems to stabilize and so does the training curve. There doesn't seem to be any (significant) overfitting as the testing score is ~8% off from the training score and both curves stabilize.

Interpretability with SHAP

- Plot the feature importance (gain and cover) from the XGBoost model. What can you conclude?

```
plt.figure(figsize=(10, 6))

plot_importance(gbt)

plt.show()
```

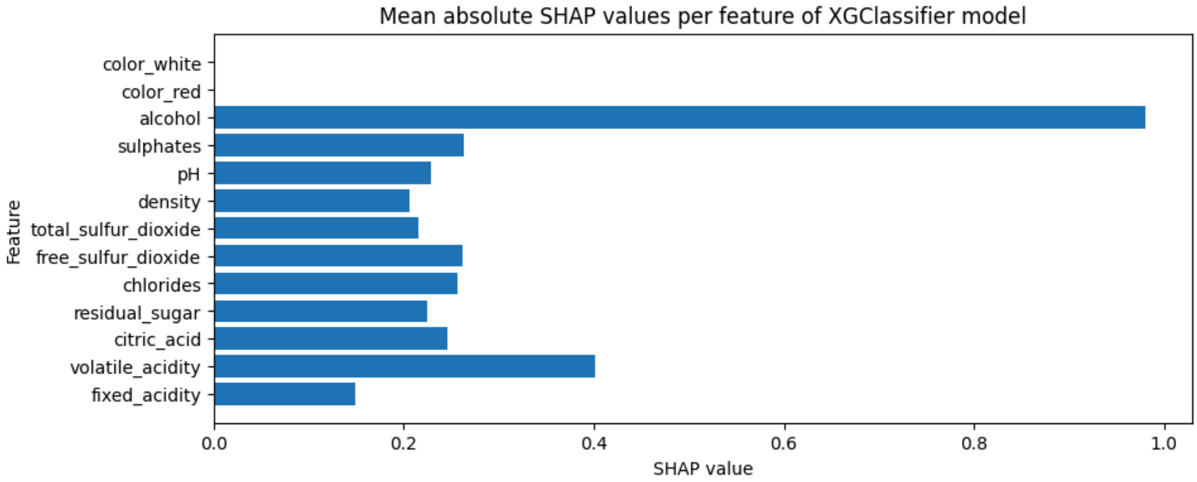


The following plot allows us to assess the importance of the different features as the `XGClassifier` model was trained. The importance, here called `F score`, is a count of how many times splitting the said feature enabled each individual tree, in average, to reduce its impurity. In our model, the `sulphates`, `volatile_acidity` and `alcohol` features allowed for most splits inside the underlying model's decision trees.

- Use SHAP's `TreeExplainer` to compute feature importance (Shapley values). Do you see any difference with XGBoost's feature importances?
- Produce different plots to analyze Shapley values:
  - A bar plot that summarizes the mean absolute value of each feature.
  - A beeswarm plot that shows the shapley value for every sample and every feature.
  - A heat map plot that indicates how different feature patterns influence the model's output.
- Based on the above results, what makes a wine 'good' or 'bad'?

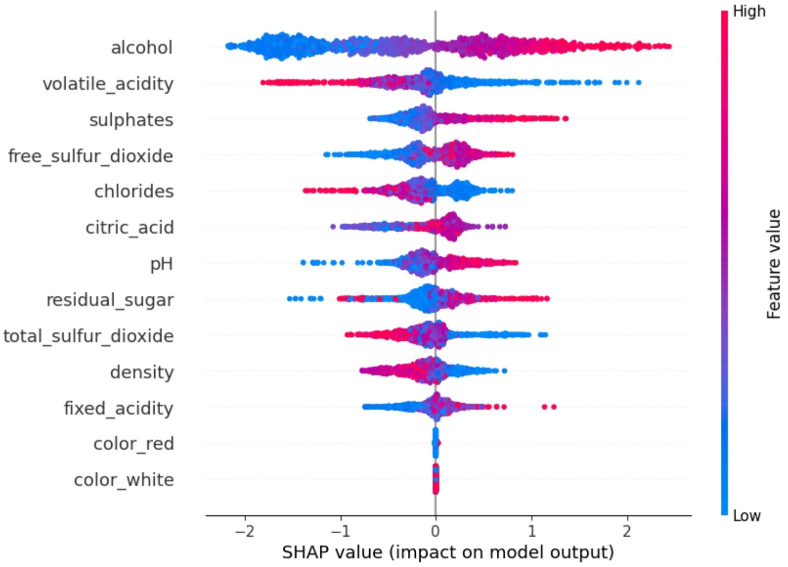
We can first compute the SHAP values of our dataset.

```
explainer = shap.TreeExplainer(gbt)
shap_gbt_explainer = explainer(X_test)
shap_values_gbt = shap_gbt_explainer.values

plt.figure(figsize=(10, 4))

plt.barh(gbt.feature_names_in_, np.abs(shap_values_gbt).mean(axis=0))

plt.xlabel("SHAP value")
plt.ylabel("Feature")

plt.title("Mean absolute SHAP values per feature of XGClassifier model")
plt.show()
```



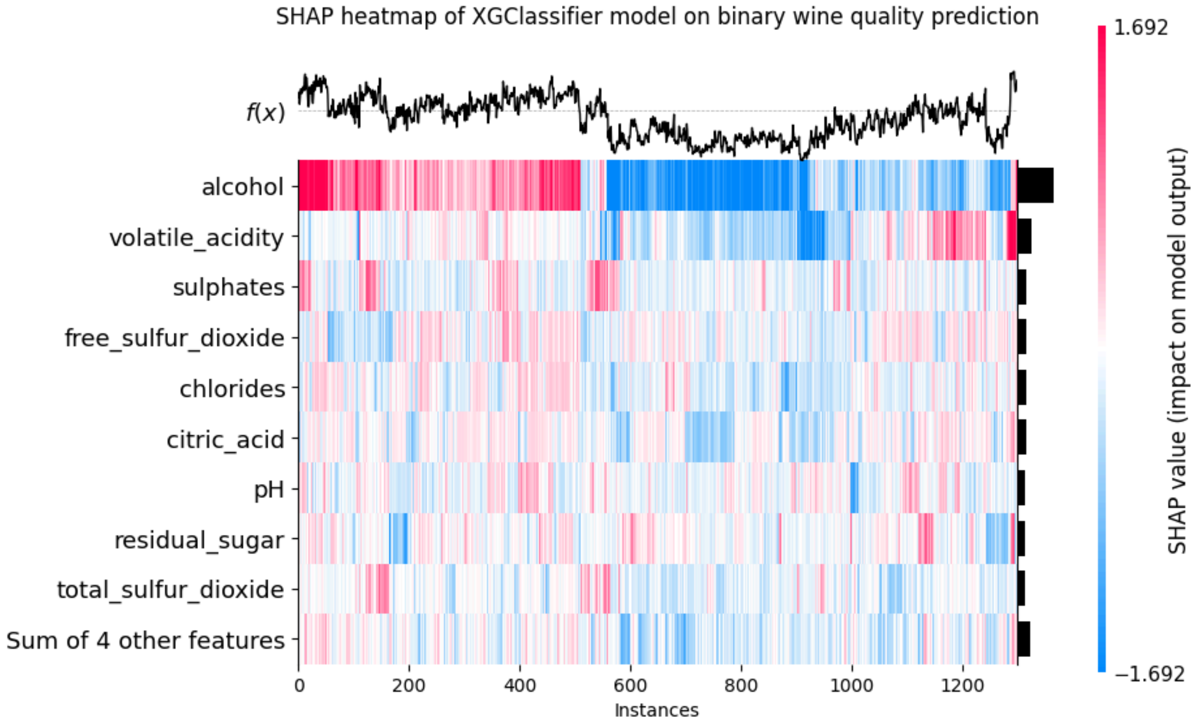
As we can see, the feature that prevails is the `alcohol` feature, which was top 3 in the feature importance plot. Its effect is even greater when observing the SHAP values. We can conclude that the `alcohol` feature is the one that has the greatest impact on a model's prediction.

```
shap.plots.beeswarm(shap_gbt_explainer, max_display=20)
```



As we can notice, the `alcohol` feature, obviously, has the greatest impact on the model's prediction based on its value. We can conclude that the greater its absolute value is, the greater its impact on the prediction will be.

```
shap.plots.heatmap(shap_gbt_explainer, show=False)

plt.title("SHAP heatmap of XGClassifier model on binary wine quality prediction")
plt.show()
```



Displaying the heat map of SHAP values can help us to identify different patterns that might tend to greatly impact the model's prediction. We now know for a fact that tuning the `alcohol` parameter, for eg., can greatly impact the model's decision in both ways. Modifying the `volatile_acidity` of our wine can also make the model predict a better quality (less frequently though). Some features such as the `citric_acid` clearly make the model tend towards a specific output. Displaying a heat map per wine color can also help to assess these patterns. Hence, new models per color will now be trained.

```
X_train_color = pd.concat([X, color], axis=1)
X_train_red = X_train_color[X_train_color["color"] == "red"]
X_train_white = X_train_color[X_train_color["color"] == "white"]

y_train_red = binary_quality.loc[X_train_red.index]
y_train_white = binary_quality.loc[X_train_white.index]

X_train_red, _, y_train_red, _ = train_test_split(X_train_red, y_train_red)
X_train_white, _, y_train_white, _ = train_test_split(X_train_white, y_train_white)

X_train_red = standard_scaler.fit_transform(X_train_red.drop(columns=["color"]))
X_train_white = standard_scaler.fit_transform(X_train_white.drop(columns=["color"]))

y_train_red = label_encoder.fit_transform(np.ravel(y_train_red))
label_encoder.inverse_transform([0, 1])
```

```
array(['bad', 'good'], dtype=object)
```

```
y_train_white = label_encoder.transform(np.ravel(y_train_white))

gbt_red = GridSearchCV(
    XGBClassifier(),
    param_grid={
        "max_depth": [3, 4, 5],  # Focus on shallow trees to reduce complexity
        "learning_rate": [0.01, 0.05, 0.1],  # Slower learning rates
        "n_estimators": [50, 100],  # More trees but keep it reasonable
        "min_child_weight": [1, 3],  # Regularization to control split thresholds
        "subsample": [0.7, 0.9],  # Sampling rate for boosting
        "colsample_bytree": [0.7, 1.0],  # Sampling rate for columns
        "gamma": [0, 0.1],  # Regularization to penalize complex trees
    },
    scoring="f1",
    n_jobs=-1,
)

gbt_white = GridSearchCV(
    XGBClassifier(),
    param_grid={
        "max_depth": [3, 4, 5],  # Focus on shallow trees to reduce complexity
        "learning_rate": [0.01, 0.05, 0.1],  # Slower learning rates
        "n_estimators": [50, 100],  # More trees but keep it reasonable
        "min_child_weight": [1, 3],  # Regularization to control split thresholds
        "subsample": [0.7, 0.9],  # Sampling rate for boosting
        "colsample_bytree": [0.7, 1.0],  # Sampling rate for columns
        "gamma": [0, 0.1],  # Regularization to penalize complex trees
    },
    scoring="f1",
    n_jobs=-1,
)
```

```
gbt_red.fit(X_train_red, y_train_red)
gbt_white.fit(X_train_white, y_train_white)
```

```
c:\dev\hesso\isc\python\ml\lab\w4-isc-ml-dij0s\.venv\lib\site-packages\numpy\ma\core.py:2820:
RuntimeWarning: invalid value encountered in cast
  _data = np.array(data, dtype=dtype, copy=copy,
c:\dev\hesso\isc\python\ml\lab\w4-isc-ml-dij0s\.venv\lib\site-packages\numpy\ma\core.py:2820:
RuntimeWarning: invalid value encountered in cast
  _data = np.array(data, dtype=dtype, copy=copy,
```

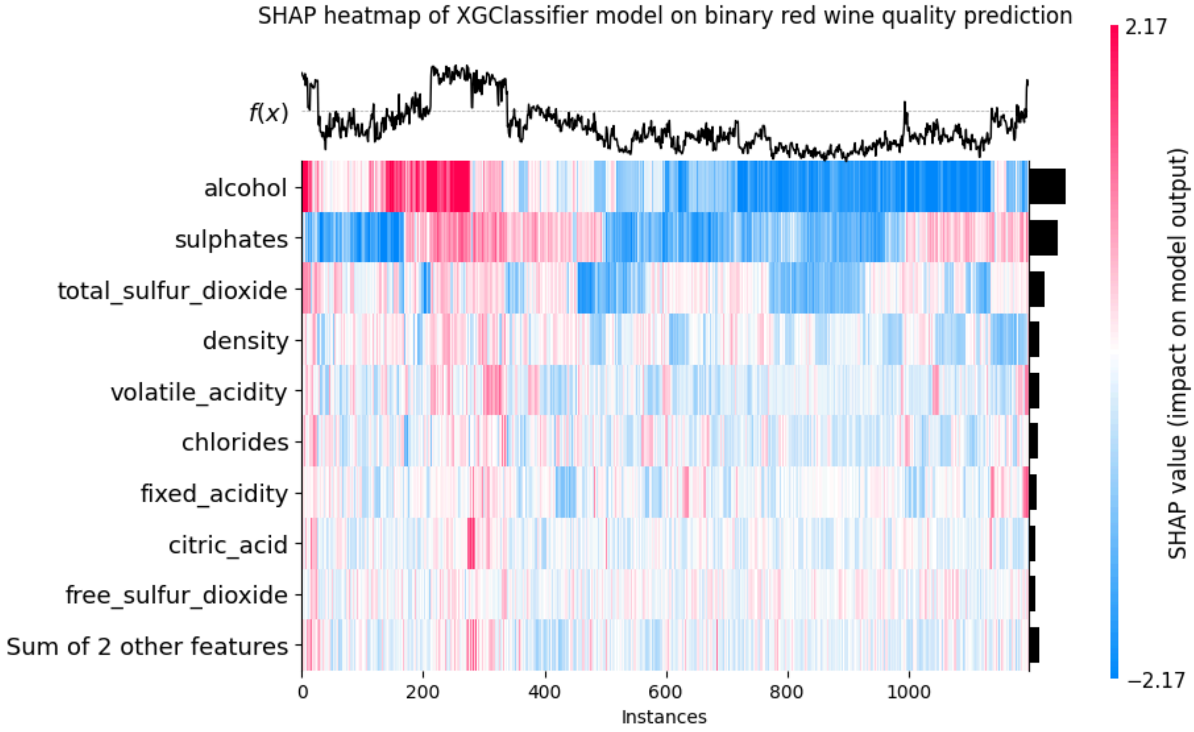
```
GridSearchCV(estimator=XGBClassifier(base_score=None, booster=None,
                                     callbacks=None, colsample_bylevel=None,
                                     colsample_bynode=None,
                                     colsample_bytree=None, device=None,
                                     early_stopping_rounds=None,
                                     enable_categorical=False, eval_metric=None,
                                     feature_types=None, gamma=None,
                                     grow_policy=None, importance_type=None,
                                     interaction_constraints=None,
                                     learning_rate=None, max_b...
                                     max_leaves=None, min_child_weight=None,
                                     missing=nan, monotone_constraints=None,
                                     multi_strategy=None, n_estimators=None,
                                     n_jobs=None, num_parallel_tree=None,
                                     random_state=None, ...),
             n_jobs=-1,
             param_grid={'colsample_bytree': [0.7, 1.0], 'gamma': [0, 0.1],
                         'learning_rate': [0.01, 0.05, 0.1],
                         'max_depth': [3, 4, 5], 'min_child_weight': [1, 3],
                         'n_estimators': [50, 100], 'subsample': [0.7, 0.9]},
             scoring='f1')
```

```
gbt_red = gbt_red.best_estimator_
gbt_white = gbt_white.best_estimator_
```

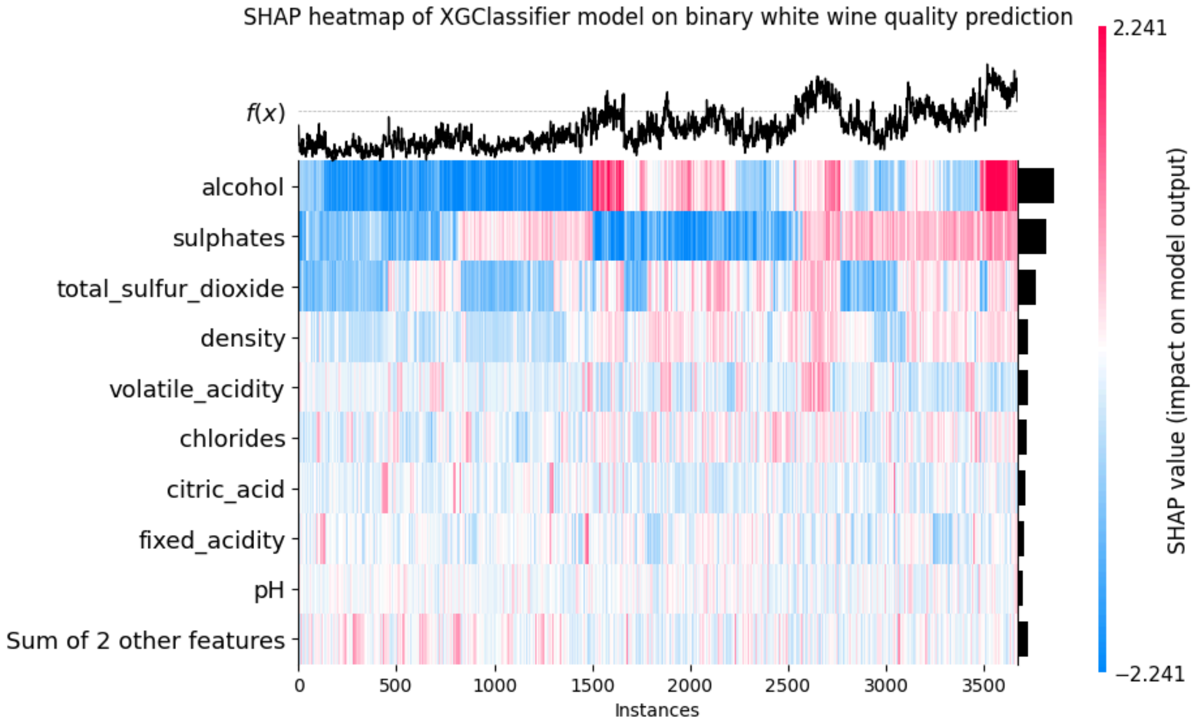
```
explainer_red = shap.TreeExplainer(gbt_red)
shap_gbt_explainer_red = explainer_red(X_train_red)
shap_values_gbt_red = shap_gbt_explainer_red.values
```

```
explainer_white = shap.TreeExplainer(gbt_white)
shap_gbt_explainer_white = explainer_red(X_train_white)
shap_values_gbt_white = shap_gbt_explainer_white.values
```

```
shap.plots.heatmap(shap_gbt_explainer_red, show=False)

plt.title("SHAP heatmap of XGClassifier model on binary red wine quality prediction")
plt.show()
```



```
shap.plots.heatmap(shap_gbt_explainer_white[:1300], show=False)

plt.title("SHAP heatmap of XGClassifier model on binary white wine quality prediction")
plt.show()
```

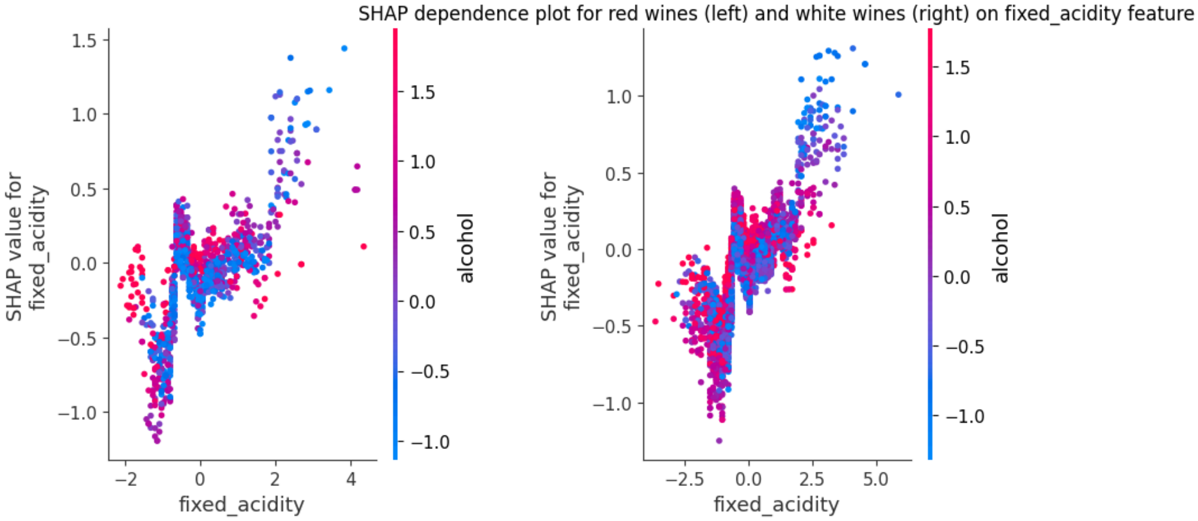


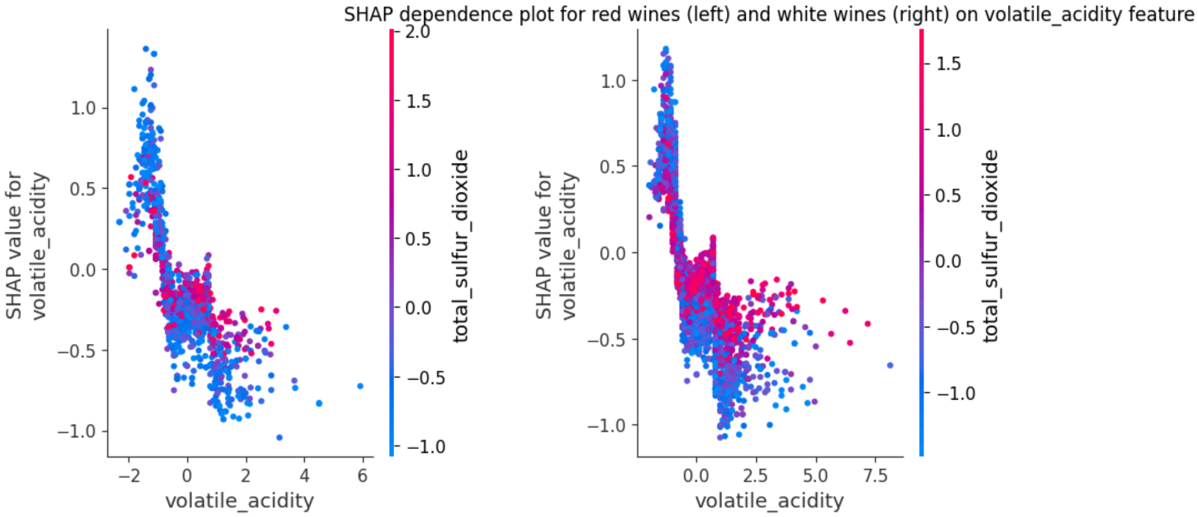
When comparing the heat maps of red and white wines, one can assess that the `alcohol` feature can better contribute to the model's "good" quality prediction on a white wine. It is double-edged as it can also greatly contribute to a negative prediction. The `sulphates` have more impact on a model's positive prediction on red wines. The `volatile_acidity` for eg. has barely any positive impact on the model's prediction when considering white wines whereas it has a lot when considering red wines. We can hence observe that the same feature, depending on the wine's color, do or don't yield any positive impact on the model's prediction.

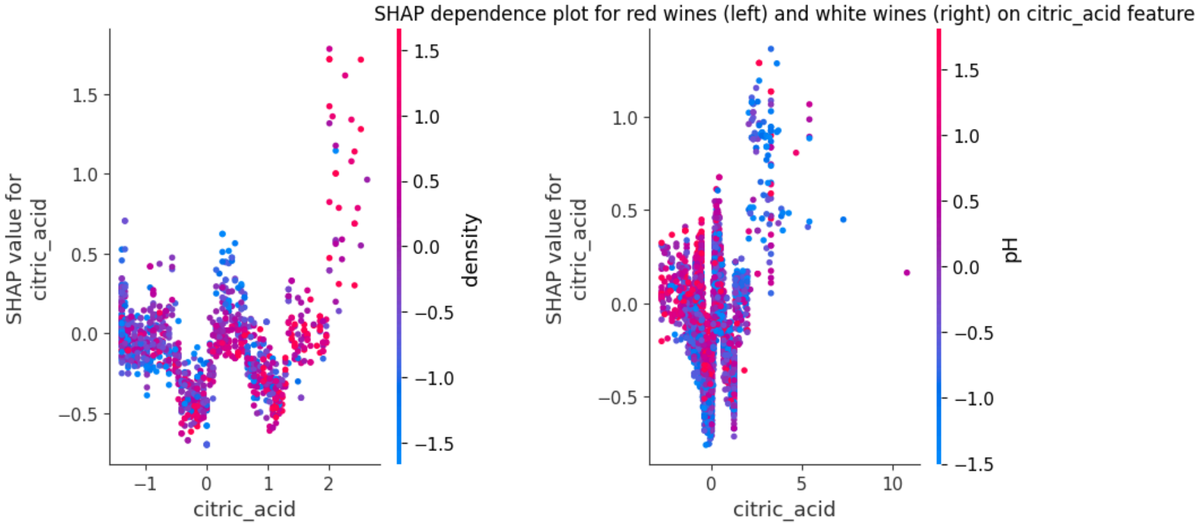
- Now use Partial Dependence Plots to see how the expected model output varies with the variation of each feature.
- How does that modify your perspective on what makes a good or bad wine?

```
for column in X_train_red.columns:
    fig, axes = plt.subplots(1, 2, figsize=(10, 5))

    plt.subplot(1, 2, 1)
    shap.dependence_plot(
        column,
        shap_values_gbt_red,
        X_train_red,
        show=False,
        ax=axes[0],
    )

    plt.subplot(1, 2, 2)
    shap.dependence_plot(
        column, shap_values_gbt_white, X_train_white, show=False, ax=axes[1], color=None
    )

    plt.title(
        f"SHAP dependence plot for red wines (left) and white wines (right) on {column}
feature"
    )
```

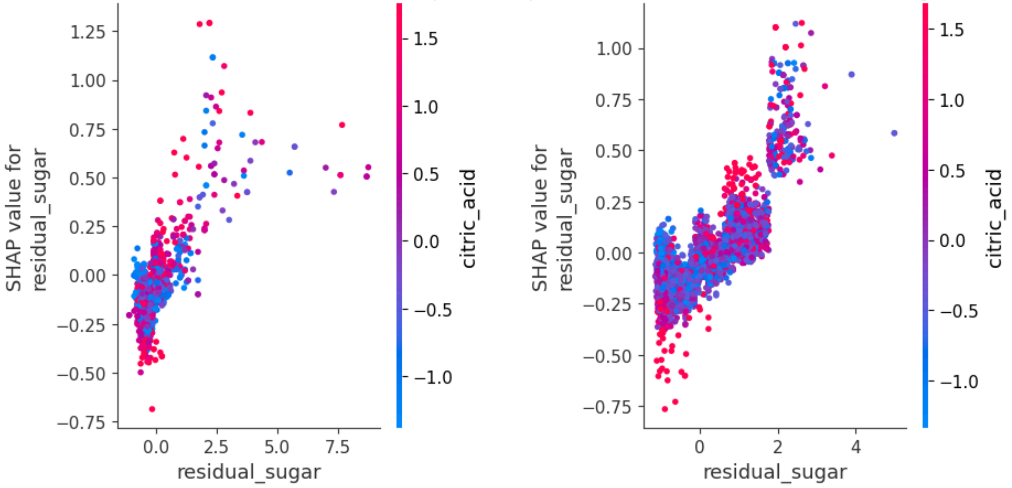
```
plt.subplots_adjust(wspace=0.6)
plt.show()
```



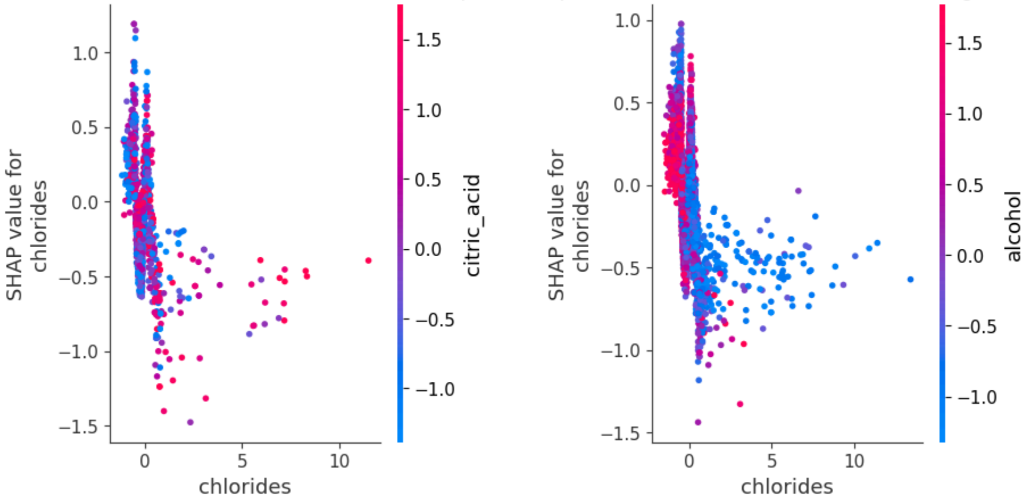




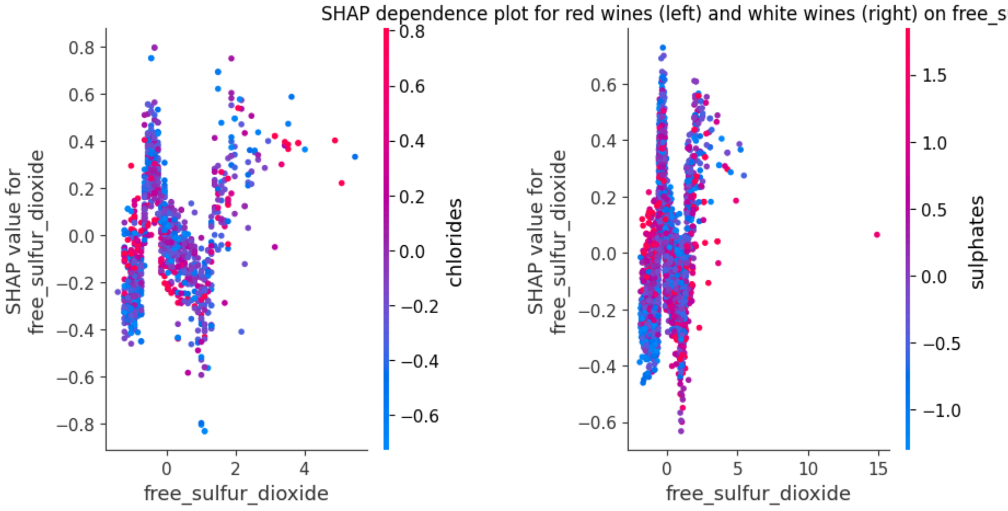
SHAP dependence plot for red wines (left) and white wines (right) on residual_sugar feature



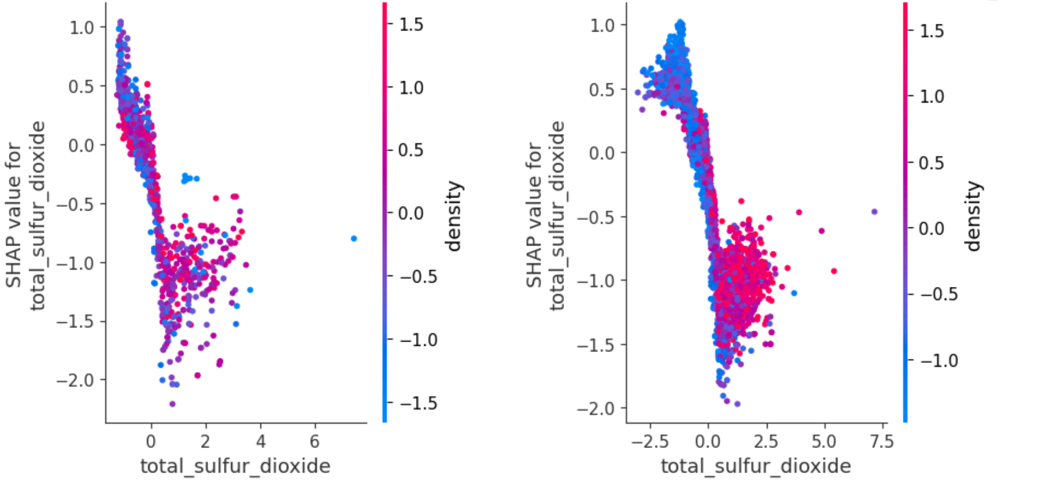
SHAP dependence plot for red wines (left) and white wines (right) on chlorides feature



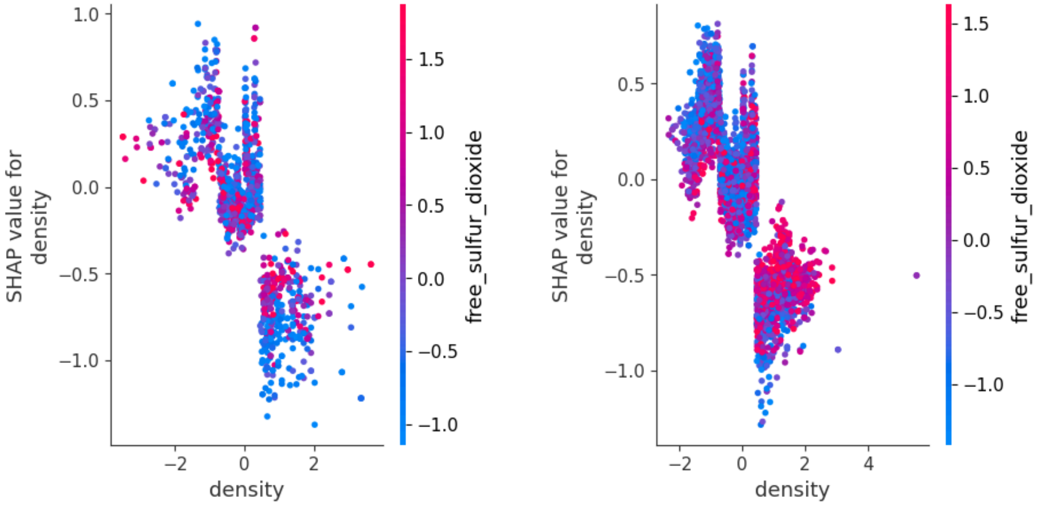
SHAP dependence plot for red wines (left) and white wines (right) on free_sulfur_dioxide feature



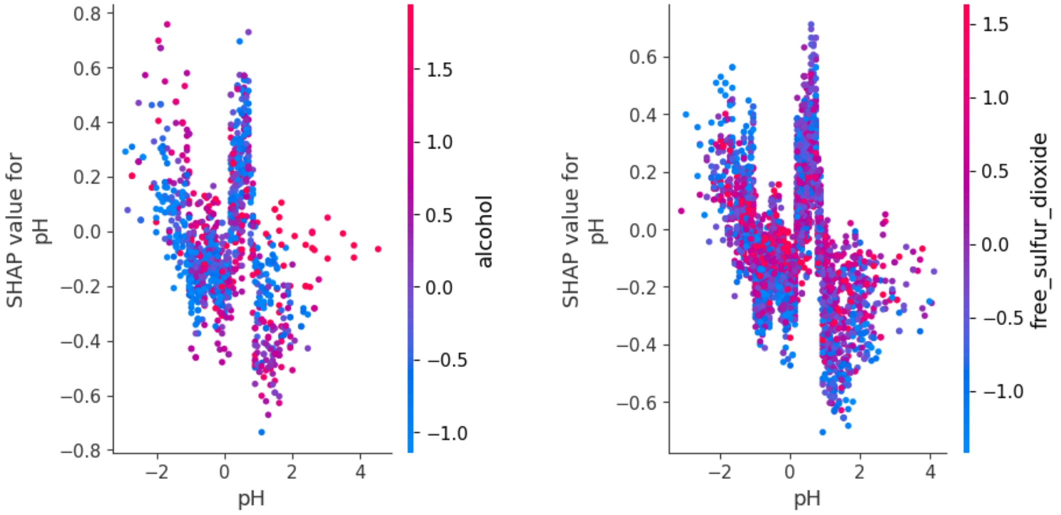
SHAP dependence plot for red wines (left) and white wines (right) on total_sulfur_dioxide feature

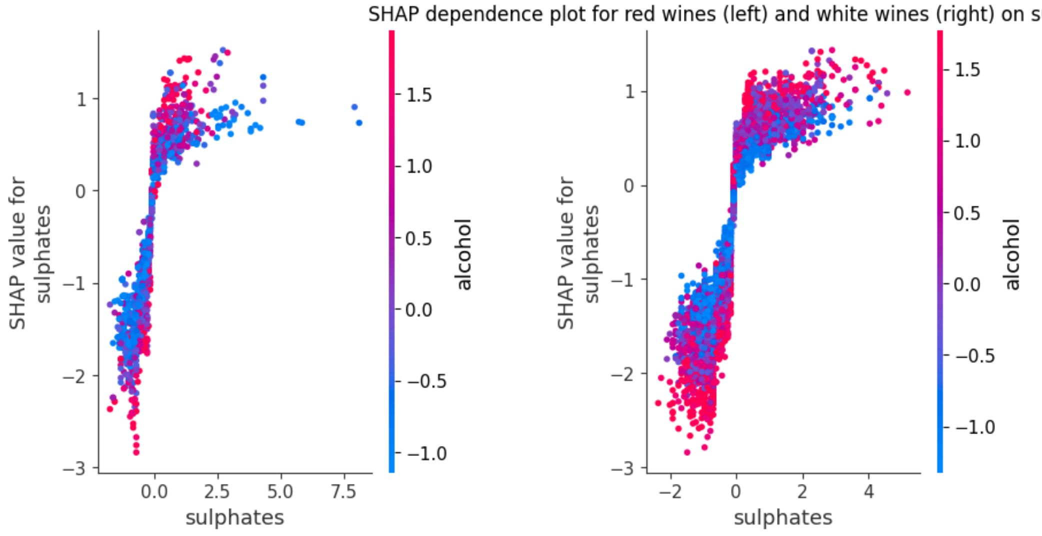


SHAP dependence plot for red wines (left) and white wines (right) on density feature

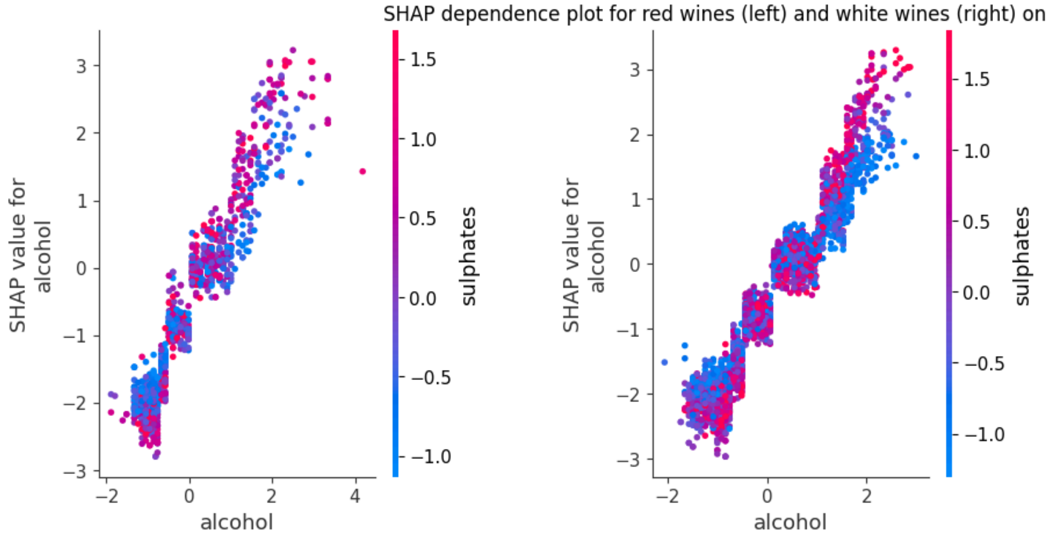


SHAP dependence plot for red wines (left) and white wines (right) on pH feature





SHAP dependence plot for red wines (left) and white wines (right) on sulphates feature



SHAP dependence plot for red wines (left) and white wines (right) on alcohol feature

On these following plots, we observe features SHAP values per wine color. The most interesting thing to extract from these are the patterns that yield the maximal SHAP value. The `alcohol` value of a sample, for eg., yields maximal SHAP value when the `sulphates` of said sample when they are both low, both around 0 or both high. The main difference between both red and white wines SHAP distributions is seen when considering the `total_sulfur_dioxide`. The SHAP value is maximized when the value of `total_sulfur_dioxide` and `density` are both small in both wines.

- Search for literature or resources that provide indications of the chemical structure of good or poor wines. Do your findings match these resources?

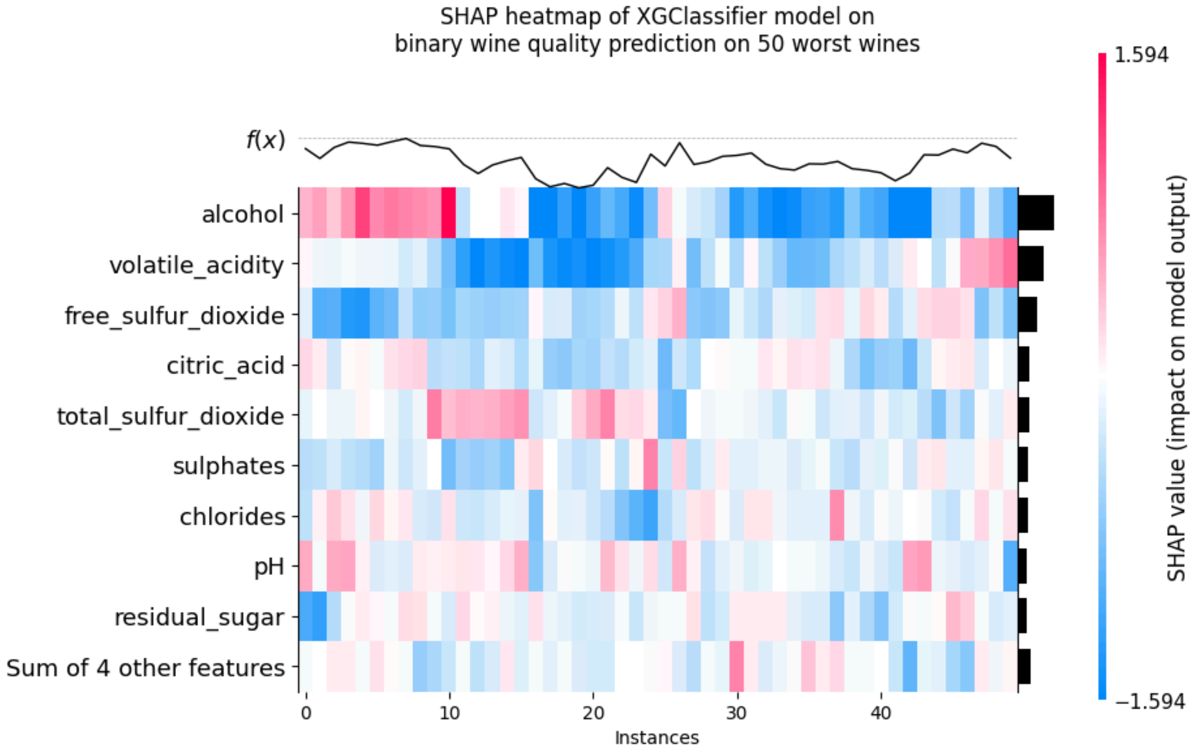
Analyze a few bad wines, and try to see how to make them better

Pick some of the worst wines, and try to see what make them so bad. Check out `shap.plots.heatmap` for some visual tool to do this.

How would you go about improving them?

```
X_train_quality = pd.concat(
    [X_train, pd.DataFrame(y["quality"]).loc[X_train.index]],
    axis=1,
)

worst_wines = X_train_quality.sort_values(by=["quality"])[:50].drop(columns=["quality"])
```

```
shap_gbt_explainer_worst = explainer(worst_wines)
shap_values_worst = shap_gbt_explainer_worst.values

shap.plots.heatmap(shap_gbt_explainer_worst, show=False)

plt.title(
    "SHAP heatmap of XGClassifier model on\nbinary wine quality prediction on 50 worst wines"
)
plt.show()
```



The worst wines can be improved by tweaking the `alcohol`, `volatile_acidity` and `total_sulfur_dioxide` as they yield great SHAP values depending on their value. These features can also yield negative impact on the model's prediction. It is hence necessary to tweak these parameters with great attention. Some features perform very bad when looking for great SHAP values (`sulphates` or `residual_sugar`) for example so these should't be tweaked much as they could potentially deteriorate the predicted quality. There might be overfitting on the model on the `alcohol` feature as it yielded a near-good quality (when observing $f(x)$) because of the `alcohol` value. Hence, high `alcohol` could bias the model into predicting a great wine.

Wrap-up and conclusion

As wrap-up, explain what are your key findings, and make 3 recommendations to the wine maker on how to improve the wines for next year. How confident are you that making these changes will lead to better wines? Explain in simple terms to the winemaker the limitations of your approach in terms of capturing causality.

Scientific facts :

- **Fixed Acidity**: Higher levels enhance freshness and balance but excessive acidity can negatively affect perceived quality source.
- **Volatile Acidity**: Typically includes acetic acid; lower levels are preferred as high concentrations can impart a vinegar-like aroma source.
- **Citric Acid**: Contributes to freshness and overall flavor profile, though present in smaller amounts source.
- **Residual Sugar**: Affects sweetness; moderate levels can improve perceived quality, especially in dessert wines, while high levels in dry wines may be unfavorable source.
- **Chlorides**: Influence salinity and mouthfeel; lower levels are generally associated with higher quality source.
- **Free and Total Sulfur Dioxide**: Acts as a preservative; higher free sulfur levels can prevent oxidation but excessive amounts may negatively impact taste source.
- **Density**: Correlates with sugar and alcohol content; affects mouthfeel and texture source.
- **pH**: Balanced pH levels (between 3 and 4) are crucial for stability and taste; lower pH levels often indicate better preservation source.
- **Sulfates**: Higher levels can enhance flavor and aroma, contributing positively to overall quality source.
- **Alcohol**: Moderate alcohol levels can enhance body and complexity, while excessive levels may detract from quality source.

Source: ChatGPT

Conclusion

In this part, we concluded different things when comparing a good wine from a bad wine.

We first observated that there isn't any direct correlation between a wine color and its quality. However, the underlying components of such wines (chemicals) may define a better wine.

We then assessed that if we were to average the chemical contents of red wines, white wines, good wines and bad wines, there is a similarity between the contents of red wines and good wines. It especially true when considering the amount of total sulfur dioxide inside the wine which, is prevalent in red wines.

When it comes to our prediction, it seems that the amount of sulfate, alcohol and volatile acidity in the wine greatly impacts what is defined as a good wine by the model. We further assessed that the alcohol ratio greatly impacts the model’s prediction as a higher alcohol ratio is, for the machine, associated with a better quality. As stated before, a moderate alcohol level can enhance body and complexity of a wine while excessive levels may detract from quality source.

It so happens that a higher alcohol ratio in white wines yields a much better quality score whereas it does not have such an impact in red wines. The higher amount of sulfates in a red wine contribute to enhancing its flavor and aroma.

In summary, if I were to cook the new wine recipe for success, I would make sure that the alcohol ratio of my wine is greater than average without reaching extravagant amounts so that I don’t detract the quality. A higher amount of total sulfur dioxide will preserve the quality of my wine and prevent it from oxidation. Finally, I would make sure to up the amount of sulfates in order to enhance flavor and aroma.