In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from dataprep.eda import plot, plot_missing, plot_correlation, create_report

from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV

from sklearn.neural_network import BernoulliRBM
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

from sklearn.metrics import (confusion_matrix, precision_score, recall_score,
                             roc_auc_score, ConfusionMatrixDisplay, f1_score,
                             accuracy_score)

import time
```

In [2]:
```
df_red = pd.read_csv("winequality-red.csv", delimiter=";")
df_red
```

Out[2]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	9.4	5
1	7.8	0.880	0.00	2.6	0.098	25.0	67.0	0.99680	3.20	0.68	9.8	5
2	7.8	0.760	0.04	2.3	0.092	15.0	54.0	0.99700	3.26	0.65	9.8	5
3	11.2	0.280	0.56	1.9	0.075	17.0	60.0	0.99800	3.16	0.58	9.8	6
4	7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	9.4	5
...	...	...	...	...	...	...	...	...	...	...	...	...
1594	6.2	0.600	0.08	2.0	0.090	32.0	44.0	0.99490	3.45	0.58	10.5	5
1595	5.9	0.550	0.10	2.2	0.062	39.0	51.0	0.99512	3.52	0.76	11.2	6
1596	6.3	0.510	0.13	2.3	0.076	29.0	40.0	0.99574	3.42	0.75	11.0	6
1597	5.9	0.645	0.12	2.0	0.075	32.0	44.0	0.99547	3.57	0.71	10.2	5
1598	6.0	0.310	0.47	3.6	0.067	18.0	42.0	0.99549	3.39	0.66	11.0	6

1599 rows × 12 columns

In [3]:
```
df_white = pd.read_csv("winequality-white.csv", delimiter=";")
df_white
```

Out[3]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.0	0.27	0.36	20.7	0.045	45.0	170.0	1.00100	3.00	0.45	8.8	6
1	6.3	0.30	0.34	1.6	0.049	14.0	132.0	0.99400	3.30	0.49	9.5	6
2	8.1	0.28	0.40	6.9	0.050	30.0	97.0	0.99510	3.26	0.44	10.1	6
3	7.2	0.23	0.32	8.5	0.058	47.0	186.0	0.99560	3.19	0.40	9.9	6
4	7.2	0.23	0.32	8.5	0.058	47.0	186.0	0.99560	3.19	0.40	9.9	6
...	...	...	...	...	...	...	...	...	...	...	...	...
4893	6.2	0.21	0.29	1.6	0.039	24.0	92.0	0.99114	3.27	0.50	11.2	6
4894	6.6	0.32	0.36	8.0	0.047	57.0	168.0	0.99490	3.15	0.46	9.6	5
4895	6.5	0.24	0.19	1.2	0.041	30.0	111.0	0.99254	2.99	0.46	9.4	6
4896	5.5	0.29	0.30	1.1	0.022	20.0	110.0	0.98869	3.34	0.38	12.8	7
4897	6.0	0.21	0.38	0.8	0.020	22.0	98.0	0.98941	3.26	0.32	11.8	6

4898 rows × 12 columns

We have two datasets for red and white wine both come with the same columns. Let's do a little bit of investigation before we decide how to move forward wit the assignment

In [4]:
```
df_red.describe()
```

Out[4]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	qualit
count	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.00000
mean	8.319637	0.527821	0.270976	2.538806	0.087467	15.874922	46.467792	0.996747	3.311113	0.658149	10.422983	5.63602
std	1.741096	0.179060	0.194801	1.409928	0.047065	10.460157	32.895324	0.001887	0.154386	0.169507	1.065668	0.80756
min	4.600000	0.120000	0.000000	0.900000	0.012000	1.000000	6.000000	0.990070	2.740000	0.330000	8.400000	3.00000
25%	7.100000	0.390000	0.090000	1.900000	0.070000	7.000000	22.000000	0.995600	3.210000	0.550000	9.500000	5.00000
50%	7.900000	0.520000	0.260000	2.200000	0.079000	14.000000	38.000000	0.996750	3.310000	0.620000	10.200000	6.00000
75%	9.200000	0.640000	0.420000	2.600000	0.090000	21.000000	62.000000	0.997835	3.400000	0.730000	11.100000	6.00000
max	15.900000	1.580000	1.000000	15.500000	0.611000	72.000000	289.000000	1.003690	4.010000	2.000000	14.900000	8.00000

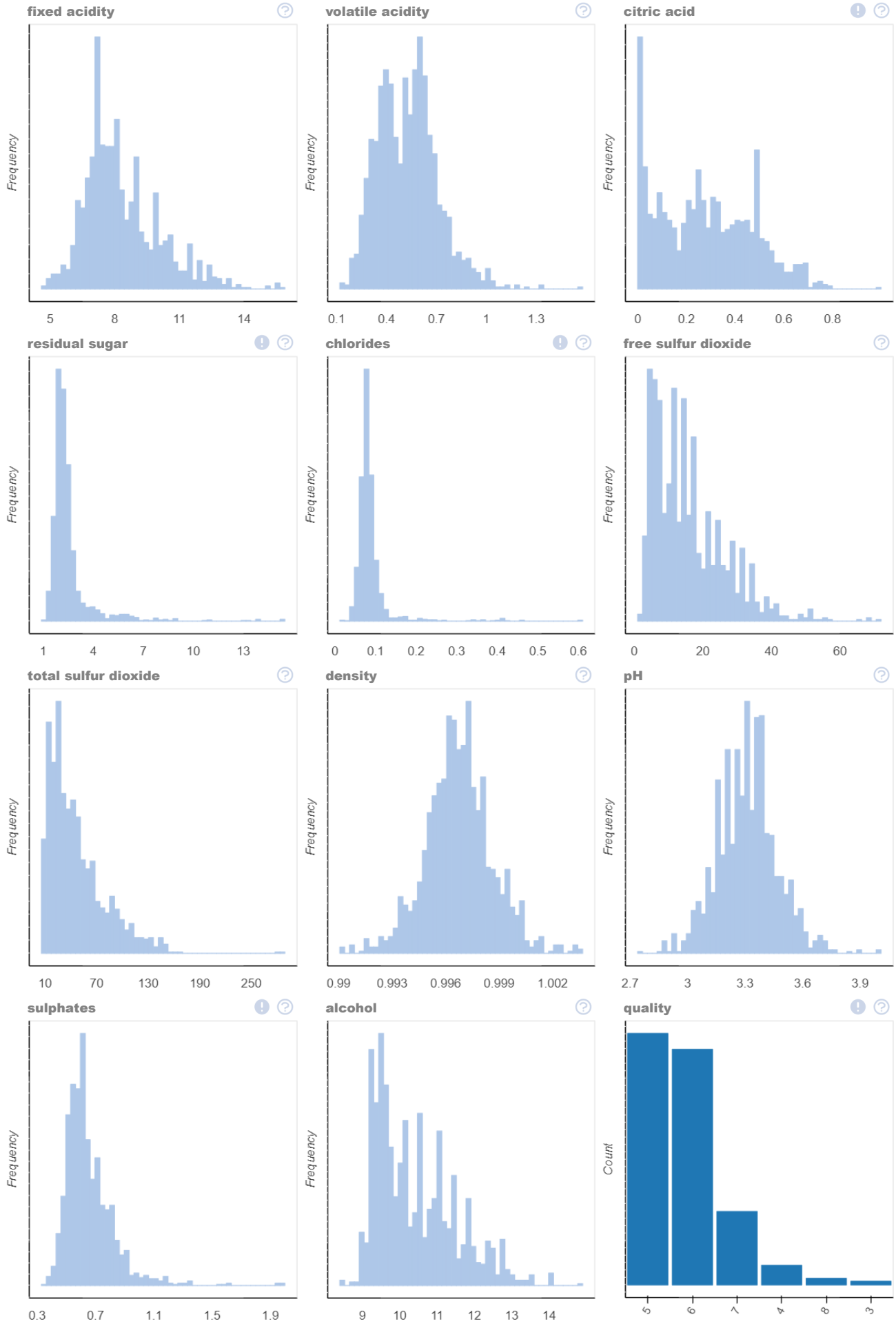
We see no nulls in the red wine set. It was indicated in the source of the dataset that there are no nulls but it's always nice to confirm. All our data is numeric. I see outliers in the max value of a lot of columns such as `fixed acidity`, `residual sugar`, `cholrides` and more. Most mean and median values for each columns are very close which tells me that the data in most columns is balanced and the spread is roughly normal around the mean which tells me that the outliers are few. We see that the range of `quality` values is between 3 and 8.
Let's make some visualizations to see things better.

In [5]:
```
plot(df_red)
```

```
  0%|                                                                                | 0/790 [00:00<…
```

Out[5]:

Show Stats and Insights



We can see that most `quality` values are around 5 and 6. We also see some repeats that we will investigate a bit further.

In [6]:
```
# keep=False to return all duplicated rows
num_duplicates = len(df_red[df_red.drop("quality", axis=1).duplicated(
    keep=False)].sort_values(by=df_red.columns.to_list())) # we sorted the values in case we want
                                                           # to view the repeated rows
print("number of duplicated columns (including repeats) in red wine dataset:", num_duplicates)

num_duplicates_originals = len(df_red[df_red.duplicated])
print("number of original duplicated columns (regardless of number of repeats) in red wine dataset:"
      , num_duplicates_originals)
```

```
number of duplicated columns (including repeats) in red wine dataset: 460
number of original duplicated columns (regardless of number of repeats) in red wine dataset: 240
```

As we can see above, we have 460 rows that have duplicates. It's best to investigate this further by reaching out to the data source but we will just keep all duplicates here as it's our suspicion that we just have wine samples with the same ingredients and factors.
Note that we compared the number of duplicates with the same columns excluding `quality` with duplicated columns including quality which tells us that the wines with the same duplicated features were given the same quality rating which is good.

Now let's take a look at the white wine dataset

In [7]:
```
df_white.describe()
```

Out[7]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
count	4898.000000	4898.000000	4898.000000	4898.000000	4898.000000	4898.000000	4898.000000	4898.000000	4898.000000	4898.000000	4898.000000	4898.000000
mean	6.854788	0.278241	0.334192	6.391415	0.045772	35.308085	138.360657	0.994027	3.188267	0.489847	10.514267	5.877909
std	0.843868	0.100795	0.121020	5.072058	0.021848	17.007137	42.498065	0.002991	0.151001	0.114126	1.230621	0.885639
min	3.800000	0.080000	0.000000	0.600000	0.009000	2.000000	9.000000	0.987110	2.720000	0.220000	8.000000	3.000000
25%	6.300000	0.210000	0.270000	1.700000	0.036000	23.000000	108.000000	0.991723	3.090000	0.410000	9.500000	5.000000
50%	6.800000	0.260000	0.320000	5.200000	0.043000	34.000000	134.000000	0.993740	3.180000	0.470000	10.400000	6.000000
75%	7.300000	0.320000	0.390000	9.900000	0.050000	46.000000	167.000000	0.996100	3.280000	0.550000	11.400000	6.000000
max	14.200000	1.100000	1.660000	65.800000	0.346000	289.000000	440.000000	1.038980	3.820000	1.080000	14.200000	9.000000

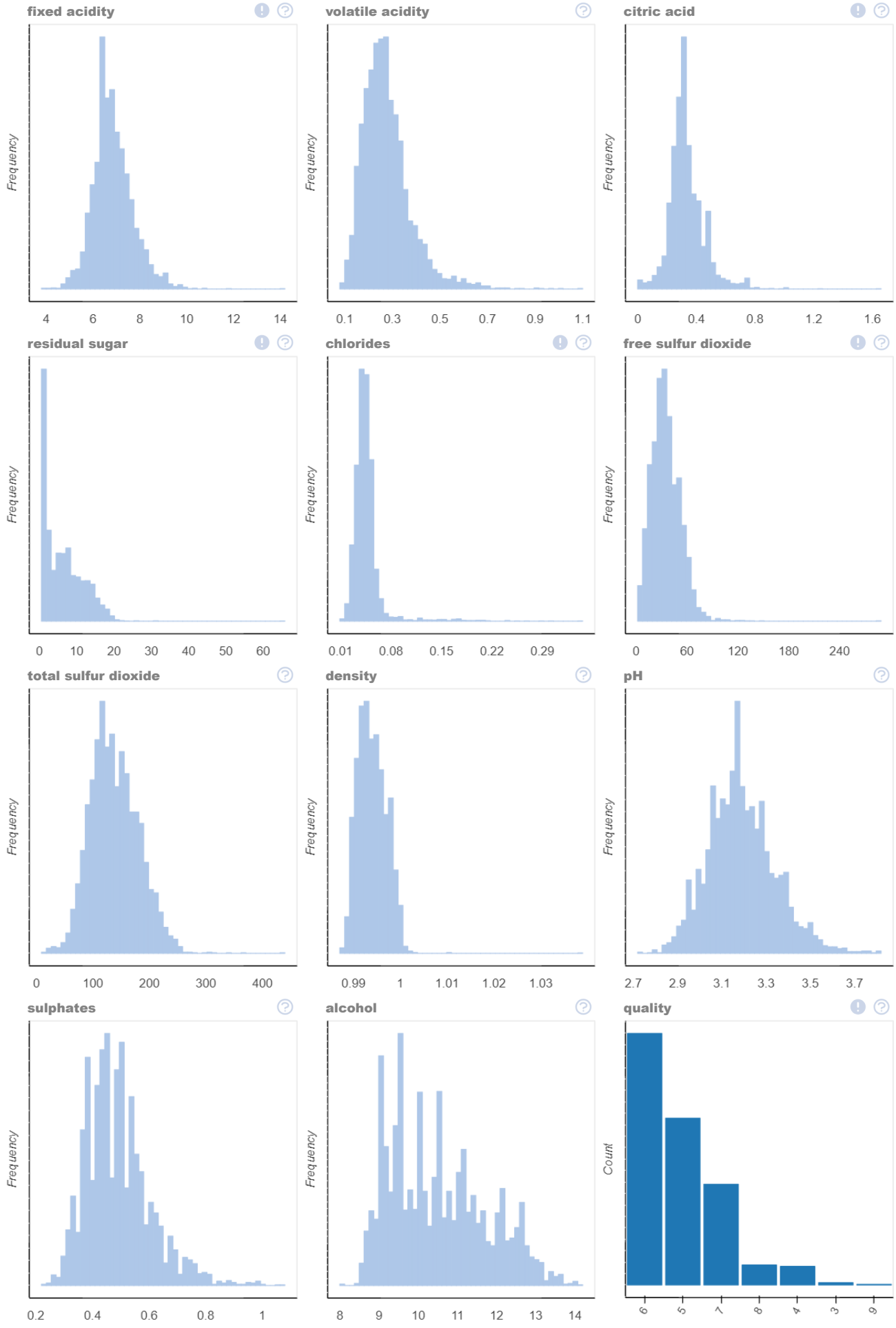
It also looks like we don't have any nulls in any column which is good. We also see some outliers in many columns especially `residual sugar`, `free sulfur dioxide`, and `total sulfur dioxide`.

In [8]:
```
plot(df_white)
```

```
  0%|                                                                                | 0/790 [00:00<…
```

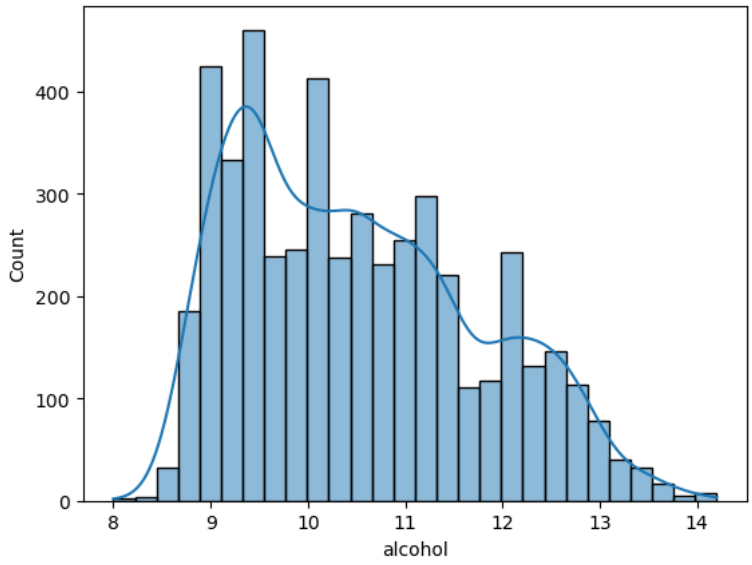
Out[8]:

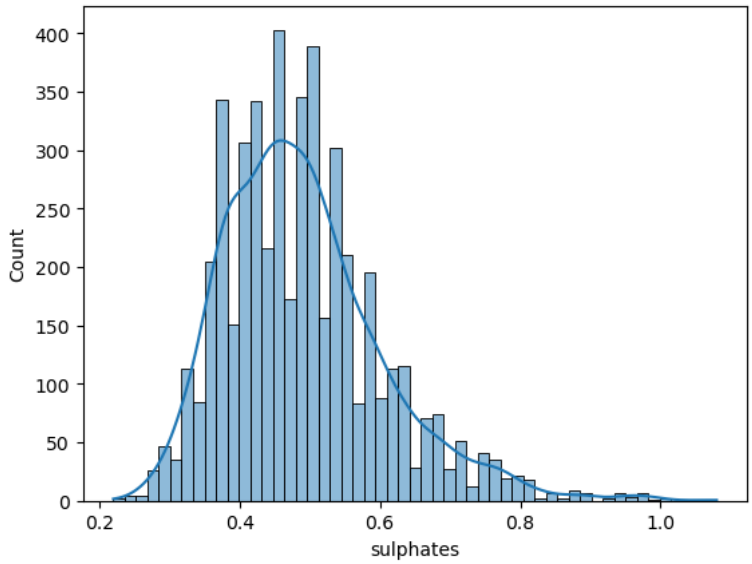
Show Stats and Insights

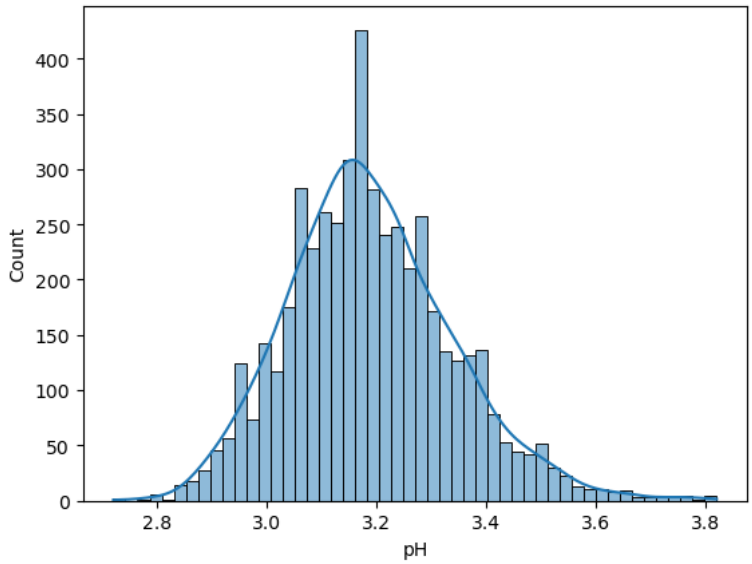


We notice that most of the values in `quality` lie between 5 and 7 similarly to red wine dataset. However, here we go up to 9 but only a small quantity.
We notice periodic peaks in both datasets in columns like `alcohol`, `sulphates`, and `pH`. Let's plot these separately to examine them.

In [9]:
```
cols_periodic_peaks = ["alcohol", "sulphates", "pH"]
for col in cols_periodic_peaks:
    sns.histplot(data=df_white, x=col, kde=True)
    plt.show()
```







The most interesting looking one to me is `sulphats`. My guess is that these values are popular among wine makers so they're used more often than neighboring values but it would be worth asking an expert in the wine-making business to investigate it further.

In [35]:
```
df_white[df_white.duplicated()]
```

Out[35]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality	quality_binary
4	7.2	0.23	0.32	8.5	0.058	47.0	186.0	0.99560	3.19	0.40	9.900000	6	
5	8.1	0.28	0.40	6.9	0.050	30.0	97.0	0.99510	3.26	0.44	10.100000	6	
7	7.0	0.27	0.36	20.7	0.045	45.0	170.0	1.00100	3.00	0.45	8.800000	6	
8	6.3	0.30	0.34	1.6	0.049	14.0	132.0	0.99400	3.30	0.49	9.500000	6	
20	6.2	0.66	0.48	1.2	0.029	29.0	75.0	0.98920	3.33	0.39	12.800000	8	
...	...	...	...	...	...	...	...	...	...	...	...	...	...
4828	6.4	0.23	0.35	10.3	0.042	54.0	140.0	0.99670	3.23	0.47	9.200000	5	
4850	7.0	0.36	0.35	2.5	0.048	67.0	161.0	0.99146	3.05	0.56	11.100000	6	
4851	6.4	0.33	0.44	8.9	0.055	52.0	164.0	0.99488	3.10	0.48	9.600000	5	
4856	7.1	0.23	0.39	13.7	0.058	26.0	172.0	0.99755	2.90	0.46	9.000000	6	
4880	6.6	0.34	0.40	8.1	0.046	68.0	170.0	0.99494	3.15	0.50	9.533333	6	

937 rows × 13 columns

In [38]:
```
# keep=False to return all duplicated rows
num_duplicates = len(df_white[df_white.duplicated(keep=False)])

print("number of duplicated columns (including repeats) in white wine dataset:", num_duplicates)

num_duplicates_originals = len(df_white[df_white.duplicated()])
print("number of original duplicated columns (regardless of number of repeats) in white wine dataset:"
      , num_duplicates_originals)
```

```
number of duplicated columns (including repeats) in white wine dataset: 1709
number of original duplicated columns (regardless of number of repeats) in white wine dataset: 937
```

Again, we see duplicates that are worth investigating but we will keep as is for the purposes of this assignment.

In [39]:
```
plot_correlation(df_white)
```

Out[39]:

Stats | Pearson | Spearman | KendallTau

	Pearson	Spearman	KendallTau
Highest Positive Correlation	0.839	0.873	0.812
Highest Negative Correlation	-0.78	-0.822	-0.635
Lowest Correlation	0.001	0.004	0.003
Mean Correlation	0.021	0.026	0.019

We don't see a lot of correlation between `quality_binary` and the rest of the columns - `quality` excluded of course. This is interesting. It's kind of like saying ingredients on their own aren't necessarily very correlated with the taste or quality but it's when they come together in the right way that determines the quality. Adding a few eggs to the cake doesn't directly translate into a better cake but it's how the eggs interact with all the other factors that make a great cake

There are always more EDA to be done, for example I believe we can and probably should plot the values of the features against the `quality` (target column) to examine more closely their effect and contribution to `quality` but we will just use all the features for now with the exception of `quality` of course.

The assignment asks us to do a binary classification of quality measure using a Boltzmann Machine (BM). We are thinking to split up the quality values into goo and bad with the values below or equal to 5 mapped to bad and 6 and above mapped to good. Also, we can pick either red or white wine to model or combine the two and include a feature indicating wheather the row or instance is red or white (perhaps we can assign 0 to red and 1 to white wine) and see how the model performs. However, since there are many differences in the measures of the ingredients for both types of wine, we thought it'd be best to just pick one and model it. I'm typically on the side that says let's define our goals to be as narrow as possible of course while still being realistic and practical in a real-world setting. Therefore, we're going with **white wine** for now as it has more data ergo, our model will have a better chance to learn the system.

In [11]:
```
df_white["quality_binary"] = df_white[["quality"]].applymap(lambda x: 0 if 0 <= x <= 5 else 1)
```

In [12]:
```
sc = StandardScaler()

# I tried MinMaxScaler and it helped bound the pseudo likelihoods closer to zero in the epochs
# but it made the overall performance of the model way worse so I went back to StandardScaler
# sc = MinMaxScaler()

X = df_white.drop(["quality", "quality_binary"], axis=1)
y = df_white[["quality_binary"]]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)

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

We will train and fit RBM, namely BernouilliRBM as implemented by the Scikit-Learn library. **We will couple it with a Logistic Regression model that will take its input from the hidden neuron values from the RBM and will be trained to perform the classification**. This is done because Boltzmann Machines are generative models and can't be used for regression directly.

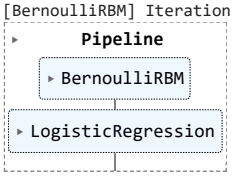
In [13]:
```
rbm = BernoulliRBM(n_components=512, learning_rate=0.001,
                   batch_size=10, n_iter=10, verbose=1,
                   random_state=0) # n_components = # hidden units, n_iter = ephocs
logistic = LogisticRegression(penalty="l2", solver='lbfgs', max_iter=500, random_state=1)
```

We will use Scikit-Learn's Pipeline to have a unified object that we can call to fit and transform both models. Also, it will help us should we want to perform grid search to find the optimal hyperparameters

In [14]:
```
classifier = Pipeline(steps=[('rbm', rbm), ('logistic', logistic)])
classifier.fit(X_train, y_train.values[:,0])
```

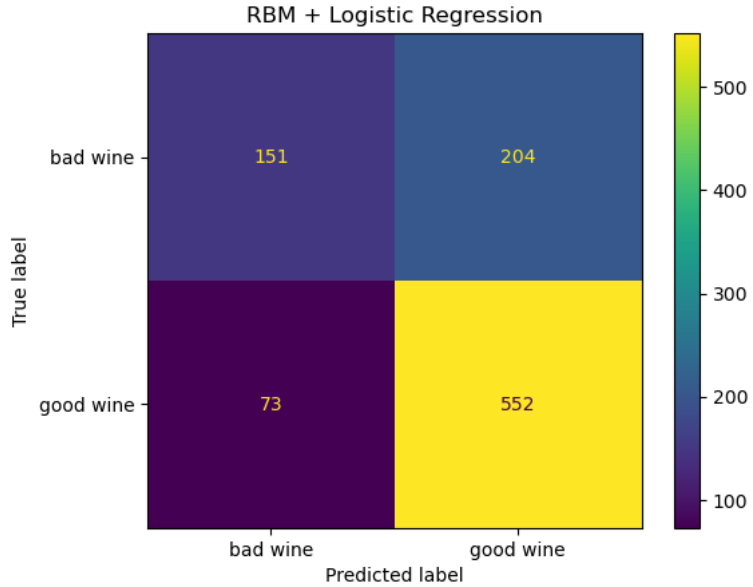
```
[BernoulliRBM] Iteration 1, pseudo-likelihood = -18.91, time = 0.46s
[BernoulliRBM] Iteration 2, pseudo-likelihood = -23.83, time = 0.72s
[BernoulliRBM] Iteration 3, pseudo-likelihood = -28.45, time = 0.61s
[BernoulliRBM] Iteration 4, pseudo-likelihood = -33.81, time = 0.61s
[BernoulliRBM] Iteration 5, pseudo-likelihood = -40.21, time = 0.59s
[BernoulliRBM] Iteration 6, pseudo-likelihood = -47.96, time = 0.69s
[BernoulliRBM] Iteration 7, pseudo-likelihood = -56.91, time = 0.80s
[BernoulliRBM] Iteration 8, pseudo-likelihood = -67.38, time = 0.67s
[BernoulliRBM] Iteration 9, pseudo-likelihood = -78.85, time = 0.78s
[BernoulliRBM] Iteration 10, pseudo-likelihood = -91.63, time = 0.87s
```

Out[14]:



In [15]:
```
predictions = classifier.predict_proba(X_test)

threshold = 0.5  # Set a threshold to determine the class
binary_predictions = [1 if prob[1] > threshold else 0 for prob in predictions]

print("accuracy:", accuracy_score(y_test, binary_predictions))
print("precision:", precision_score(y_test, binary_predictions))
print("recall:", recall_score(y_test, binary_predictions))
print("f1-score:", f1_score(y_test, binary_predictions))

cm = confusion_matrix(y_test, binary_predictions)
ConfusionMatrixDisplay(confusion_matrix=cm,
                       display_labels=["bad wine", "good wine"]).plot()

plt.title("RBM + Logistic Regression")
plt.show()
```

```
accuracy: 0.7173469387755103
precision: 0.7301587301587301
recall: 0.8832
f1-score: 0.7994207096307022
```



We tried to run the model with a different number of neurons (hidden units) and different values for the batch (e.g. 32, 64) and different number of epochs. Those didn't seem to make a considerable difference. increasing the number of neurons helped as we went up to 512, the farther up we went didn't seem to enhance the performance.

The one thing that considerably helped the model is the learning rate. The default value for it is 0.1 which I believe is too high. We decreased it to 0.01 and it performed better. We went down to 0.001 and it performed even better. We tried a few values around the latter number and it didn't seem to enhance the model in any significant way. The learning rate has been by far the best way to enhance the performance.

For logistic regression we increased *max_iter* as we were getting warnings saying that the maximum was hit before convergence. We experimented with many different *solver* options and norms for he penality but found that *lbfgs* performed slightly better than the rest.

As We've noticed from first hand experience and from the literature, it's tricky to train an RBM. We will now use grid_search to systematically try out different combinations of hyperparameters to see if we can find a good combination.

In [16]:
```
custom_params = [
    {'rbm__n_components': [100], 'rbm__learning_rate': [0.001], 'rbm__n_iter': [20],
     'logistic__C': [0.1, 1], 'logistic__solver': ['liblinear'], 'logistic__penalty': ["l1", "l2"],
     'logistic__max_iter': [500]},
    {'rbm__n_components': [128, 256, 512, 1024], 'rbm__learning_rate': [0.0005, 0.001, 0.01, 0.1],
     'rbm__n_iter': [10, 50], 'logistic__C': [0.001, 0.01, 0.1, 1], "logistic__max_iter": [500]},
]

rbm = BernoulliRBM(random_state=0)
logistic_regression = LogisticRegression(random_state=0, max_iter=500)
classifier = Pipeline(steps=[('rbm', rbm), ('logistic', logistic_regression)])

t1 = time.time()

# note that we're just looking at accuracy in the evaluations below
# as they're the default scoring metric in classification
grid_search = GridSearchCV(classifier, custom_params, cv=5) # 5-fold cross-validation
grid_search.fit(X_train, y_train.values[:,0])

t2 = time.time()
print("time to execute grid_search in minutes:", (t2-t1)/60)

# Get the best hyperparameters and the corresponding best model
best_params = grid_search.best_params_
best_model = grid_search.best_estimator_
```

```
C:\Users\yousi\anaconda3\lib\site-packages\sklearn\linear_model\_logistic.py:458: ConvergenceWarning: lbfgs failed to converge (status=1)
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-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-regression
  n_iter_i = _check_optimize_result(
C:\Users\yousi\anaconda3\lib\site-packages\sklearn\linear_model\_logistic.py:458: ConvergenceWarning: lbfgs failed to converge (status=1)
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-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-regression
  n_iter_i = _check_optimize_result(
C:\Users\yousi\anaconda3\lib\site-packages\sklearn\linear_model\_logistic.py:458: ConvergenceWarning: lbfgs failed to converge (status=1)
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-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-regression
  n_iter_i = _check_optimize_result(
C:\Users\yousi\anaconda3\lib\site-packages\sklearn\linear_model\_logistic.py:458: ConvergenceWarning: lbfgs failed to converge (status=1)
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-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-regression
  n_iter_i = _check_optimize_result(
C:\Users\yousi\anaconda3\lib\site-packages\sklearn\linear_model\_logistic.py:458: ConvergenceWarning: lbfgs failed to converge (status=1)
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-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-regression
  n_iter_i = _check_optimize_result(
C:\Users\yousi\anaconda3\lib\site-packages\sklearn\linear_model\_logistic.py:458: ConvergenceWarning: lbfgs failed to converge (status=1)
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-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-regression
  n_iter_i = _check_optimize_result(
time to execute grid_search in minutes: 88.13229154745737
```

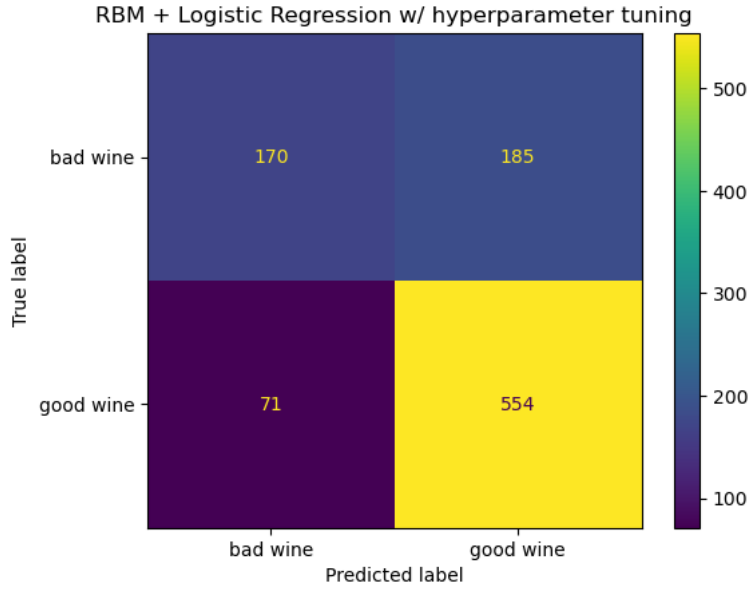
In [17]:
```
best_params
```

Out[17]:
```
{'logistic__C': 1,
 'logistic__max_iter': 500,
 'rbm__learning_rate': 0.0005,
 'rbm__n_components': 512,
 'rbm__n_iter': 50}
```

In [18]:
```
predictions = best_model.predict_proba(X_test)

threshold = 0.5  # Set a threshold to determine the class
binary_predictions = [1 if prob[1] > threshold else 0 for prob in predictions]

print("accuracy:", accuracy_score(y_test, binary_predictions))
print("precision:", precision_score(y_test, binary_predictions))
print("recall:", recall_score(y_test, binary_predictions))
```

```
print("f1-score:", f1_score(y_test, binary_predictions))

cm = confusion_matrix(y_test, binary_predictions)
ConfusionMatrixDisplay(confusion_matrix=cm,
                       display_labels=["bad wine", "good wine"]).plot()

plt.title("RBM + Logistic Regression w/ hyperparameter tuning")
plt.show()
```

```
accuracy: 0.7387755102040816
precision: 0.7496617050067659
recall: 0.8864
f1-score: 0.8123167155425219
```



After a long time of execution, we finally get our best hyperparameters. We were able to improve the model by a few percentage points at least in accuracy. I noticed that type I error got reduced considerably. It's still not a very good performing model by any means, though. I'll run it against XGBoost just to see if we can perform better

Just out of curiosity, we will train a vanilla XGBoost classifier to see how it compares against our composite model

In [19]:
```
import xgboost

xg_clf = xgboost.XGBClassifier()
xg_clf.fit(X_train, y_train)
```

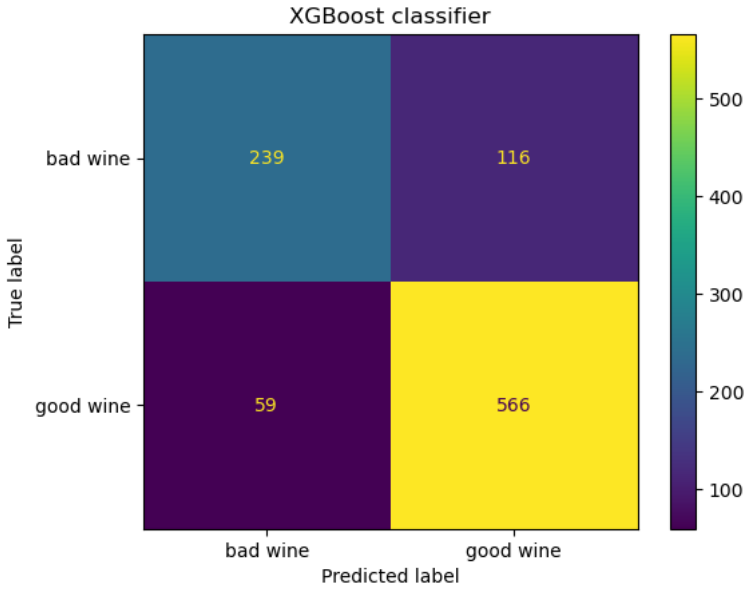
Out[19]:

XGBClassifier

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, gpu_id=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=None, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              n_estimators=100, n_jobs=None, num_parallel_tree=None,
              predictor=None, random_state=None, ...)
```

In [20]:
```
predictions = xg_clf.predict_proba(X_test)

threshold = 0.5  # Set a threshold to determine the class
binary_predictions = [1 if prob[1] > threshold else 0 for prob in predictions]

print("accuracy:", accuracy_score(y_test, binary_predictions))
print("precision:", precision_score(y_test, binary_predictions))
print("recall:", recall_score(y_test, binary_predictions))
print("f1-score:", f1_score(y_test, binary_predictions))

cm = confusion_matrix(y_test, binary_predictions)
ConfusionMatrixDisplay(confusion_matrix=cm,
                       display_labels=["bad wine", "good wine"]).plot()

plt.title("XGBoost classifier")
plt.show()
```

```
accuracy: 0.8214285714285714
precision: 0.8299120234604106
recall: 0.9056
f1-score: 0.86610558530987
```

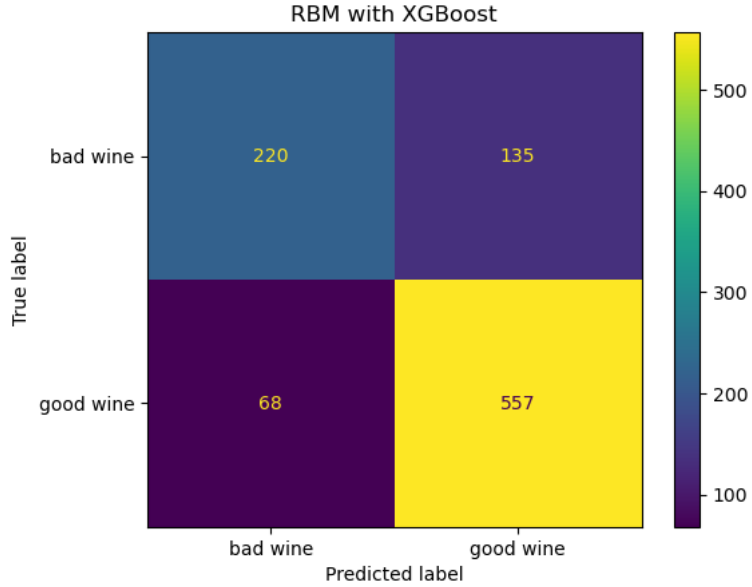


We can see that XGBoost performs way better than our model. Perhaps RBM needs more data to really learn the system. Also, we can always experiment with having a classifier different from Logistic Regression. In fact, why don't we try combining RBM with XGBoost!

In [24]:

```
rbm = BernoulliRBM(n_components=512, learning_rate=0.0005,
                   batch_size=10, n_iter=50, verbose=1,
                   random_state=0) # n_components = # hidden units, n_iter = ephocs
xgb_clf = xgboost.XGBClassifier()

classifier = Pipeline(steps=[('rbm', rbm), ('XGB', xgb_clf)])
classifier.fit(X_train, y_train.values[:,0])

predictions = classifier.predict_proba(X_test)

threshold = 0.5  # Set a threshold to determine the class
binary_predictions = [1 if prob[1] > threshold else 0 for prob in predictions]

print("accuracy:", accuracy_score(y_test, binary_predictions))
print("precision:", precision_score(y_test, binary_predictions))
print("recall:", recall_score(y_test, binary_predictions))
print("f1-score:", f1_score(y_test, binary_predictions))

cm = confusion_matrix(y_test, binary_predictions)
ConfusionMatrixDisplay(confusion_matrix=cm,
                       display_labels=["bad wine", "good wine"]).plot()

plt.title("RBM with XGBoost")
plt.show()
```

```
[BernoulliRBM] Iteration 1, pseudo-likelihood = -15.14, time = 0.24s
[BernoulliRBM] Iteration 2, pseudo-likelihood = -18.77, time = 0.51s
[BernoulliRBM] Iteration 3, pseudo-likelihood = -21.41, time = 0.61s
[BernoulliRBM] Iteration 4, pseudo-likelihood = -23.76, time = 0.81s
[BernoulliRBM] Iteration 5, pseudo-likelihood = -26.05, time = 0.78s
[BernoulliRBM] Iteration 6, pseudo-likelihood = -28.45, time = 0.74s
[BernoulliRBM] Iteration 7, pseudo-likelihood = -30.96, time = 0.66s
[BernoulliRBM] Iteration 8, pseudo-likelihood = -33.69, time = 0.63s
[BernoulliRBM] Iteration 9, pseudo-likelihood = -36.65, time = 0.58s
[BernoulliRBM] Iteration 10, pseudo-likelihood = -40.03, time = 0.71s
[BernoulliRBM] Iteration 11, pseudo-likelihood = -43.69, time = 0.65s
[BernoulliRBM] Iteration 12, pseudo-likelihood = -47.69, time = 0.77s
[BernoulliRBM] Iteration 13, pseudo-likelihood = -51.94, time = 0.69s
[BernoulliRBM] Iteration 14, pseudo-likelihood = -56.65, time = 0.74s
[BernoulliRBM] Iteration 15, pseudo-likelihood = -61.62, time = 0.61s
[BernoulliRBM] Iteration 16, pseudo-likelihood = -66.95, time = 0.68s
[BernoulliRBM] Iteration 17, pseudo-likelihood = -72.53, time = 0.82s
[BernoulliRBM] Iteration 18, pseudo-likelihood = -78.47, time = 0.75s
[BernoulliRBM] Iteration 19, pseudo-likelihood = -84.61, time = 0.64s
[BernoulliRBM] Iteration 20, pseudo-likelihood = -91.05, time = 0.69s
[BernoulliRBM] Iteration 21, pseudo-likelihood = -97.75, time = 0.52s
[BernoulliRBM] Iteration 22, pseudo-likelihood = -104.56, time = 0.66s
[BernoulliRBM] Iteration 23, pseudo-likelihood = -111.46, time = 0.72s
[BernoulliRBM] Iteration 24, pseudo-likelihood = -118.44, time = 0.71s
[BernoulliRBM] Iteration 25, pseudo-likelihood = -125.31, time = 0.64s
[BernoulliRBM] Iteration 26, pseudo-likelihood = -132.20, time = 0.58s
[BernoulliRBM] Iteration 27, pseudo-likelihood = -139.02, time = 0.70s
[BernoulliRBM] Iteration 28, pseudo-likelihood = -145.78, time = 0.65s
[BernoulliRBM] Iteration 29, pseudo-likelihood = -152.47, time = 0.59s
[BernoulliRBM] Iteration 30, pseudo-likelihood = -159.18, time = 0.73s
[BernoulliRBM] Iteration 31, pseudo-likelihood = -165.54, time = 0.58s
[BernoulliRBM] Iteration 32, pseudo-likelihood = -172.19, time = 0.57s
[BernoulliRBM] Iteration 33, pseudo-likelihood = -179.01, time = 0.57s
[BernoulliRBM] Iteration 34, pseudo-likelihood = -185.41, time = 0.56s
[BernoulliRBM] Iteration 35, pseudo-likelihood = -192.38, time = 0.52s
[BernoulliRBM] Iteration 36, pseudo-likelihood = -199.61, time = 0.64s
[BernoulliRBM] Iteration 37, pseudo-likelihood = -206.40, time = 0.72s
[BernoulliRBM] Iteration 38, pseudo-likelihood = -214.53, time = 0.85s
[BernoulliRBM] Iteration 39, pseudo-likelihood = -222.63, time = 0.93s
[BernoulliRBM] Iteration 40, pseudo-likelihood = -231.26, time = 0.81s
[BernoulliRBM] Iteration 41, pseudo-likelihood = -239.66, time = 0.55s
[BernoulliRBM] Iteration 42, pseudo-likelihood = -248.96, time = 0.61s
[BernoulliRBM] Iteration 43, pseudo-likelihood = -258.03, time = 0.61s
[BernoulliRBM] Iteration 44, pseudo-likelihood = -266.82, time = 0.68s
[BernoulliRBM] Iteration 45, pseudo-likelihood = -277.71, time = 0.81s
[BernoulliRBM] Iteration 46, pseudo-likelihood = -288.82, time = 0.74s
[BernoulliRBM] Iteration 47, pseudo-likelihood = -297.71, time = 0.65s
[BernoulliRBM] Iteration 48, pseudo-likelihood = -307.65, time = 0.63s
[BernoulliRBM] Iteration 49, pseudo-likelihood = -323.17, time = 0.73s
[BernoulliRBM] Iteration 50, pseudo-likelihood = -329.95, time = 0.67s
accuracy: 0.7928571428571428
precision: 0.8049132947976878
recall: 0.8912
f1-score: 0.8458618071374335
```



We see that we actually performed slightly worse than by just passing the data straight to the XGB model. We can mess with hyperparameter tuning here and increase our performance but, I can say that a system like this is likely too simple and can be modeled with simple classic ML models or even neural networks and it doesn't need an energy based model. Those should be reserved for more complex simples with latent features

In [ ]: