Seventh Practice ML

In this practice, we will learn **Regularizations** and how to perform **Hyper-Parameters Search**. We will also learn how use **Ensembles** to get predictions from more than one model, and how to use the models **KNN** and **LWLR (In the next practice)**. We will learn how to get code from other developers from **PyPi** (https://pypi.org/) or **GitHub** (https://github.com/) and use it in our work.
We will also learn how to use **AutoViz (https://github.com/AutoViML/AutoViz)** to show fast DataFrame graph reports.

PyPi

The Python Package Index (PyPI) (https://pypi.org/) is a repository of software for the Python programming language.
PyPI helps you find and install software developed and shared by the Python community.
Package authors use PyPI to distribute their software.

GitHub

GitHub (https://github.com/) is a code hosting platform for version control and collaboration.
It lets you and others work together on projects from anywhere.
It is also a common way to share the code you wrote with other developers.
Sometimes, the regular packages we are using are not enough, and we want to use things that are not officially implemented yet.
We can write them down ourselves, or search for implementations written by other developers.
These implementations will mostly be hosted at GitHub.

If we want to use them and their developers did not upload them to PyPi (to be downloaded easily with `pip`), we need to download them directly from their repositories in GitHub.

Downloads, Imports, and Definitions

We update packages that their Colab version is too old.

In [ ]:

```
!pip install --upgrade plotly
!pip install autoviz
```

```
Collecting plotly
  Downloading https://files.pythonhosted.org/packages/c9/09/315462259ab7b6
0a3d4b7159233ed700733c87d889755bdc00a9fb46d692/plotly-4.14.1-py2.py3-none-
any.whl (13.2MB)
     |████████████████████████████████| 13.2MB 5.0MB/s
Requirement already satisfied, skipping upgrade: six in /usr/local/lib/pyt
hon3.6/dist-packages (from plotly) (1.15.0)
Requirement already satisfied, skipping upgrade: retrying>=1.3.3 in /usr/l
ocal/lib/python3.6/dist-packages (from plotly) (1.3.3)
Installing collected packages: plotly
  Found existing installation: plotly 4.4.1
    Uninstalling plotly-4.4.1:
      Successfully uninstalled plotly-4.4.1
Successfully installed plotly-4.14.1
Collecting autoviz
  Downloading https://files.pythonhosted.org/packages/89/20/8c8c64d5221cfc
bc54679f4f048a08292a16dbad178af7c78541aa3af730/autoviz-0.0.81-py3-none-an
y.whl
Requirement already satisfied: jupyter in /usr/local/lib/python3.6/dist-pa
ckages (from autoviz) (1.0.0)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.6/dist
-packages (from autoviz) (3.2.2)
Requirement already satisfied: seaborn in /usr/local/lib/python3.6/dist-pa
ckages (from autoviz) (0.11.0)
Requirement already satisfied: ipython in /usr/local/lib/python3.6/dist-pa
ckages (from autoviz) (5.5.0)
Requirement already satisfied: pandas in /usr/local/lib/python3.6/dist-pac
kages (from autoviz) (1.1.5)
Requirement already satisfied: xgboost in /usr/local/lib/python3.6/dist-pa
ckages (from autoviz) (0.90)
Requirement already satisfied: statsmodels in /usr/local/lib/python3.6/dis
t-packages (from autoviz) (0.10.2)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.6/di
st-packages (from autoviz) (0.22.2.post1)
Requirement already satisfied: ipywidgets in /usr/local/lib/python3.6/dist
-packages (from jupyter->autoviz) (7.5.1)
Requirement already satisfied: ipykernel in /usr/local/lib/python3.6/dist-
packages (from jupyter->autoviz) (4.10.1)
Requirement already satisfied: nbconvert in /usr/local/lib/python3.6/dist-
packages (from jupyter->autoviz) (5.6.1)
Requirement already satisfied: notebook in /usr/local/lib/python3.6/dist-p
ackages (from jupyter->autoviz) (5.3.1)
Requirement already satisfied: jupyter-console in /usr/local/lib/python3.
6/dist-packages (from jupyter->autoviz) (5.2.0)
Requirement already satisfied: qtconsole in /usr/local/lib/python3.6/dist-
packages (from jupyter->autoviz) (5.0.1)
Requirement already satisfied: numpy>=1.11 in /usr/local/lib/python3.6/dis
t-packages (from matplotlib->autoviz) (1.19.4)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python
3.6/dist-packages (from matplotlib->autoviz) (1.3.1)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.6/di
st-packages (from matplotlib->autoviz) (0.10.0)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in
/usr/local/lib/python3.6/dist-packages (from matplotlib->autoviz) (2.4.7)
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/pyth
on3.6/dist-packages (from matplotlib->autoviz) (2.8.1)
Requirement already satisfied: scipy>=1.0 in /usr/local/lib/python3.6/dist
-packages (from seaborn->autoviz) (1.4.1)
Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.
6/dist-packages (from ipython->autoviz) (50.3.2)
Requirement already satisfied: pickleshare in /usr/local/lib/python3.6/dis
```

```
t-packages (from ipython->autoviz) (0.7.5)
Requirement already satisfied: simplegeneric>0.8 in /usr/local/lib/python
3.6/dist-packages (from ipython->autoviz) (0.8.1)
Requirement already satisfied: traitlets>=4.2 in /usr/local/lib/python3.6/
dist-packages (from ipython->autoviz) (4.3.3)
Requirement already satisfied: pygments in /usr/local/lib/python3.6/dist-p
ackages (from ipython->autoviz) (2.6.1)
Requirement already satisfied: decorator in /usr/local/lib/python3.6/dist-
packages (from ipython->autoviz) (4.4.2)
Requirement already satisfied: pexpect; sys_platform != "win32" in /usr/lo
cal/lib/python3.6/dist-packages (from ipython->autoviz) (4.8.0)
Requirement already satisfied: prompt-toolkit<2.0.0,>=1.0.4 in /usr/local/
lib/python3.6/dist-packages (from ipython->autoviz) (1.0.18)
Requirement already satisfied: pytz>=2017.2 in /usr/local/lib/python3.6/di
st-packages (from pandas->autoviz) (2018.9)
Requirement already satisfied: patsy>=0.4.0 in /usr/local/lib/python3.6/di
st-packages (from statsmodels->autoviz) (0.5.1)
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.6/di
st-packages (from scikit-learn->autoviz) (1.0.0)
Requirement already satisfied: nbformat>=4.2.0 in /usr/local/lib/python3.
6/dist-packages (from ipywidgets->jupyter->autoviz) (5.0.8)
Requirement already satisfied: widgetsnbextension~=3.5.0 in /usr/local/li
b/python3.6/dist-packages (from ipywidgets->jupyter->autoviz) (3.5.1)
Requirement already satisfied: jupyter-client in /usr/local/lib/python3.6/
dist-packages (from ipykernel->jupyter->autoviz) (5.3.5)
Requirement already satisfied: tornado>=4.0 in /usr/local/lib/python3.6/di
st-packages (from ipykernel->jupyter->autoviz) (5.1.1)
Requirement already satisfied: testpath in /usr/local/lib/python3.6/dist-p
ackages (from nbconvert->jupyter->autoviz) (0.4.4)
Requirement already satisfied: entrypoints>=0.2.2 in /usr/local/lib/python
3.6/dist-packages (from nbconvert->jupyter->autoviz) (0.3)
Requirement already satisfied: mistune<2,>=0.8.1 in /usr/local/lib/python
3.6/dist-packages (from nbconvert->jupyter->autoviz) (0.8.4)
Requirement already satisfied: jinja2>=2.4 in /usr/local/lib/python3.6/dis
t-packages (from nbconvert->jupyter->autoviz) (2.11.2)
Requirement already satisfied: bleach in /usr/local/lib/python3.6/dist-pac
kages (from nbconvert->jupyter->autoviz) (3.2.1)
Requirement already satisfied: defusedxml in /usr/local/lib/python3.6/dist
-packages (from nbconvert->jupyter->autoviz) (0.6.0)
Requirement already satisfied: jupyter-core in /usr/local/lib/python3.6/di
st-packages (from nbconvert->jupyter->autoviz) (4.7.0)
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/local/lib/pyth
on3.6/dist-packages (from nbconvert->jupyter->autoviz) (1.4.3)
Requirement already satisfied: Send2Trash in /usr/local/lib/python3.6/dist
-packages (from notebook->jupyter->autoviz) (1.5.0)
Requirement already satisfied: terminado>=0.8.1 in /usr/local/lib/python3.
6/dist-packages (from notebook->jupyter->autoviz) (0.9.1)
Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.
6/dist-packages (from notebook->jupyter->autoviz) (0.2.0)
Requirement already satisfied: pyzmq>=17.1 in /usr/local/lib/python3.6/dis
t-packages (from qtconsole->jupyter->autoviz) (20.0.0)
Requirement already satisfied: qtpy in /usr/local/lib/python3.6/dist-packa
ges (from qtconsole->jupyter->autoviz) (1.9.0)
Requirement already satisfied: six in /usr/local/lib/python3.6/dist-packag
es (from cycler>=0.10->matplotlib->autoviz) (1.15.0)
Requirement already satisfied: ptyprocess>=0.5 in /usr/local/lib/python3.
6/dist-packages (from pexpect; sys_platform != "win32"->ipython->autoviz)
(0.6.0)
Requirement already satisfied: wcwidth in /usr/local/lib/python3.6/dist-pa
ckages (from prompt-toolkit<2.0.0,>=1.0.4->ipython->autoviz) (0.2.5)
Requirement already satisfied: jsonschema!=2.5.0,>=2.4 in /usr/local/lib/p
```

```
ython3.6/dist-packages (from nbformat>=4.2.0->ipywidgets->jupyter->autovi
z) (2.6.0)
Requirement already satisfied: MarkupSafe>=0.23 in /usr/local/lib/python3.
6/dist-packages (from jinja2>=2.4->nbconvert->jupyter->autoviz) (1.1.1)
Requirement already satisfied: packaging in /usr/local/lib/python3.6/dist-
packages (from bleach->nbconvert->jupyter->autoviz) (20.8)
Requirement already satisfied: webencodings in /usr/local/lib/python3.6/di
st-packages (from bleach->nbconvert->jupyter->autoviz) (0.5.1)
Installing collected packages: autoviz
Successfully installed autoviz-0.0.81
```

We import our regular packages.

In [ ]:

```
# import numpy, matplotlib, etc.
import numpy as np
import pandas as pd
import seaborn as sns
import plotly.express as px
import matplotlib.pyplot as plt
import plotly.graph_objects as go

# sklearn imports
from sklearn import metrics
from sklearn import pipeline
from sklearn import linear_model
from sklearn import preprocessing
from sklearn import neural_network
from sklearn import model_selection
from sklearn.pipeline import Pipeline
from sklearn.model_selection import KFold
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import LeavePOut
from sklearn.linear_model import SGDRegressor
from sklearn.linear_model import SGDClassifier
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import OrdinalEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import train_test_split
```

Data Exploration

We use the Vinho Verde's White Wines (https://www.kaggle.com/danielpanizzo/wine-quality) dataset.

Explanation

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

In the above reference, two datasets were created, using red and white wine samples. The inputs include objective tests (e.g. PH values) and the output is based on sensory data (median of at least 3 evaluations made by wine experts).
Each expert graded the wine quality between 0 (very bad) and 10 (very excellent).

Relevant Information:
The dataset is related to white variants of the Portuguese "Vinho Verde" wine.
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.).
These datasets can be viewed as a classification or regression task.
The classes are ordered and not balanced (e.g. there are many more normal wines than excellent or poor ones).

white wine samples - 4898.

Number of Attributes - 11 + output attribute

Attribute information

Input variables (based on physicochemical tests):

1. **fixed acidity** (tartaric acid - g / dm^3) - most acids involved with wine or fixed or nonvolatile (do not evaporate readily).
2. **volatile acidity** (acetic acid - g / dm^3) - the amount of acetic acid in wine, which at too high of levels can lead to an unpleasant, vinegar taste.
3. **citric acid** (g / dm^3) - found in small quantities, citric acid can add 'freshness' and flavor to wines.
4. **residual sugar** (g / dm^3) - the amount of sugar remaining after fermentation stops, it's rare to find wines with less than 1 gram/liter, and wines with greater than 45 grams/liter are considered sweet.
5. **chlorides** (sodium chloride - g / dm^3) - the amount of salt in the wine.
6. **free sulfur dioxide** (mg / dm^3) - the free form of SO2 exists in equilibrium between molecular SO2 (as a dissolved gas) and bisulfite ion; it prevents microbial growth and the oxidation of the wine.
7. **total sulfur dioxide** (mg / dm^3) - the amount of free and bound forms of S02; in low concentrations, SO2 is mostly undetectable in wine, but at free SO2 concentrations over 50 ppm, SO2 becomes evident in the nose and taste of wine.
8. **density** (g / cm^3) - the density of water is close to that of water depending on the percent alcohol and sugar content.
9. **pH** - describes how acidic or basic a wine is on a scale from 0 (very acidic) to 14 (very basic); most wines are between 3-4 on the pH scale.
10. **sulphates** (potassium sulphate - g / dm3) - a wine additive that can contribute to sulfur dioxide gas (S02) levels, which acts as an antimicrobial and antioxidant.
11. **alcohol** (% by volume) - the percent alcohol content of the wine.

Output variable (based on sensory data):

1. **quality** (score between 0 and 10)

Let's download the dataset from Github and explore it with Pandas tools.

In [ ]:

```
# download whitewines.csv file from Github
!wget https://raw.githubusercontent.com/stedy/Machine-Learning-with-R-datasets/master/whitewines.csv
```

```
--2020-12-21 01:13:46--  https://raw.githubusercontent.com/stedy/Machine-Learning-with-R-datasets/master/whitewines.csv
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 151.101.0.133, 151.101.64.133, 151.101.128.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|151.101.0.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 264426 (258K) [text/plain]
Saving to: ‘whitewines.csv’

whitewines.csv      100%[===================>] 258.23K  --.-KB/s    in 0.04s

2020-12-21 01:13:47 (6.52 MB/s) - ‘whitewines.csv’ saved [264426/264426]
```

In [ ]:

```
# load the whitewines csv file
whitewines_df = pd.read_csv('/content/whitewines.csv')
whitewines_df
```

Out[ ]:

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

4898 rows × 12 columns

In [ ]:

```
# show whitewines_df info
whitewines_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4898 entries, 0 to 4897
Data columns (total 12 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   fixed acidity         4898 non-null   float64
 1   volatile acidity      4898 non-null   float64
 2   citric acid           4898 non-null   float64
 3   residual sugar        4898 non-null   float64
 4   chlorides             4898 non-null   float64
 5   free sulfur dioxide   4898 non-null   float64
 6   total sulfur dioxide  4898 non-null   float64
 7   density               4898 non-null   float64
 8   pH                    4898 non-null   float64
 9   sulphates             4898 non-null   float64
 10  alcohol               4898 non-null   float64
 11  quality               4898 non-null   int64  
dtypes: float64(11), int64(1)
memory usage: 459.3 KB
```

In [ ]:

```
# show whitewines_df description
whitewines_df.describe()
```

Out[ ]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total
count	4898.000000	4898.000000	4898.000000	4898.000000	4898.000000	4898.000000	4898.(
mean	6.854788	0.278241	0.334192	6.391415	0.045772	35.308085	138.3
std	0.843868	0.100795	0.121020	5.072058	0.021848	17.007137	42.4
min	3.800000	0.080000	0.000000	0.600000	0.009000	2.000000	9.(
25%	6.300000	0.210000	0.270000	1.700000	0.036000	23.000000	108.(
50%	6.800000	0.260000	0.320000	5.200000	0.043000	34.000000	134.(
75%	7.300000	0.320000	0.390000	9.900000	0.050000	46.000000	167.(
max	14.200000	1.100000	1.660000	65.800000	0.346000	289.000000	440.(

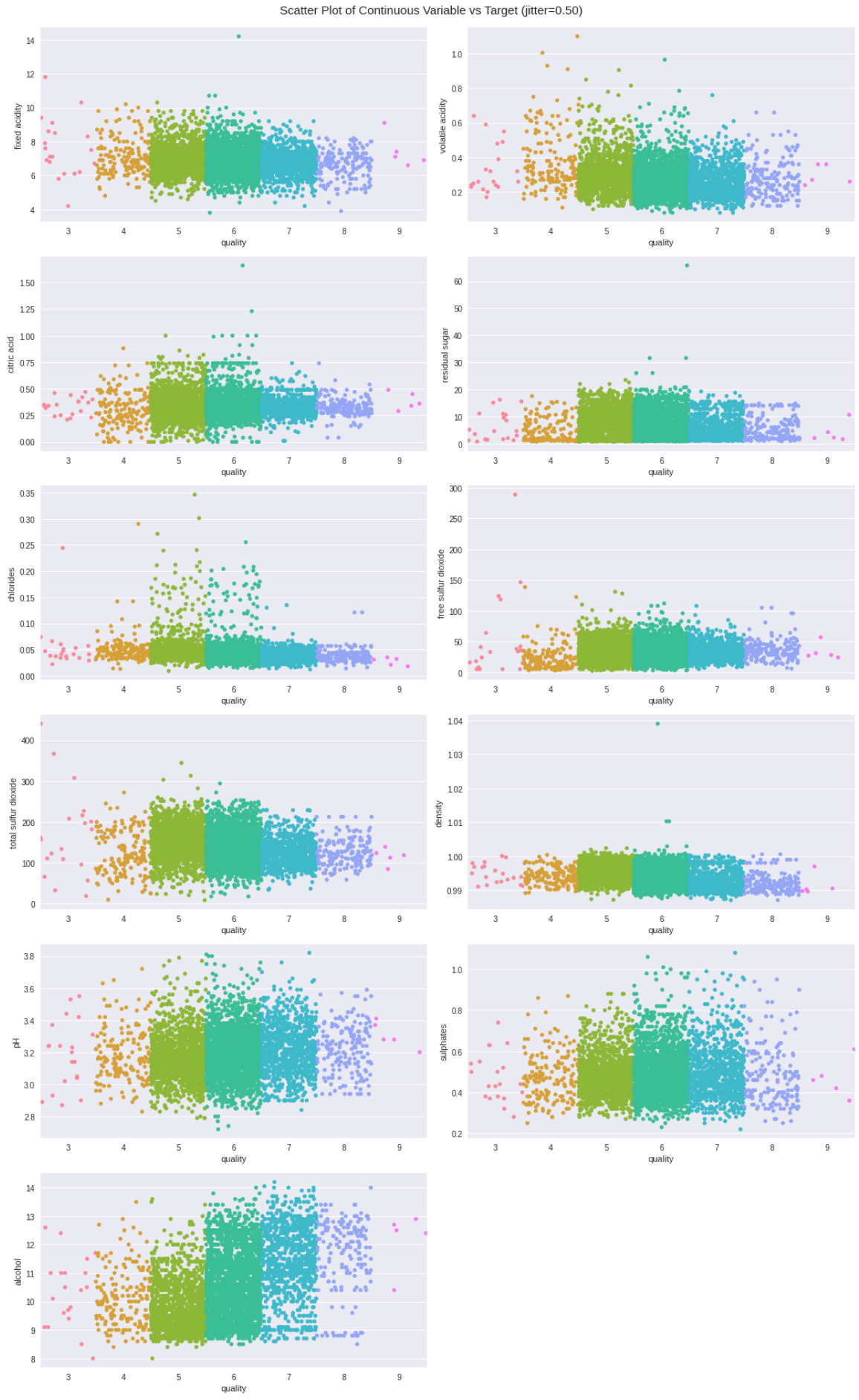
We can also use `autoviz` to show a report on the data.
This report is based on graphs other than text.

In [ ]:

```
#  import autoviz and show report on usedcars_df
from autoviz.AutoViz_Class import AutoViz_Class

AV = AutoViz_Class()
dft = AV.AutoViz("", depVar='quality', dfte=whitewines_df, verbose=1)
```

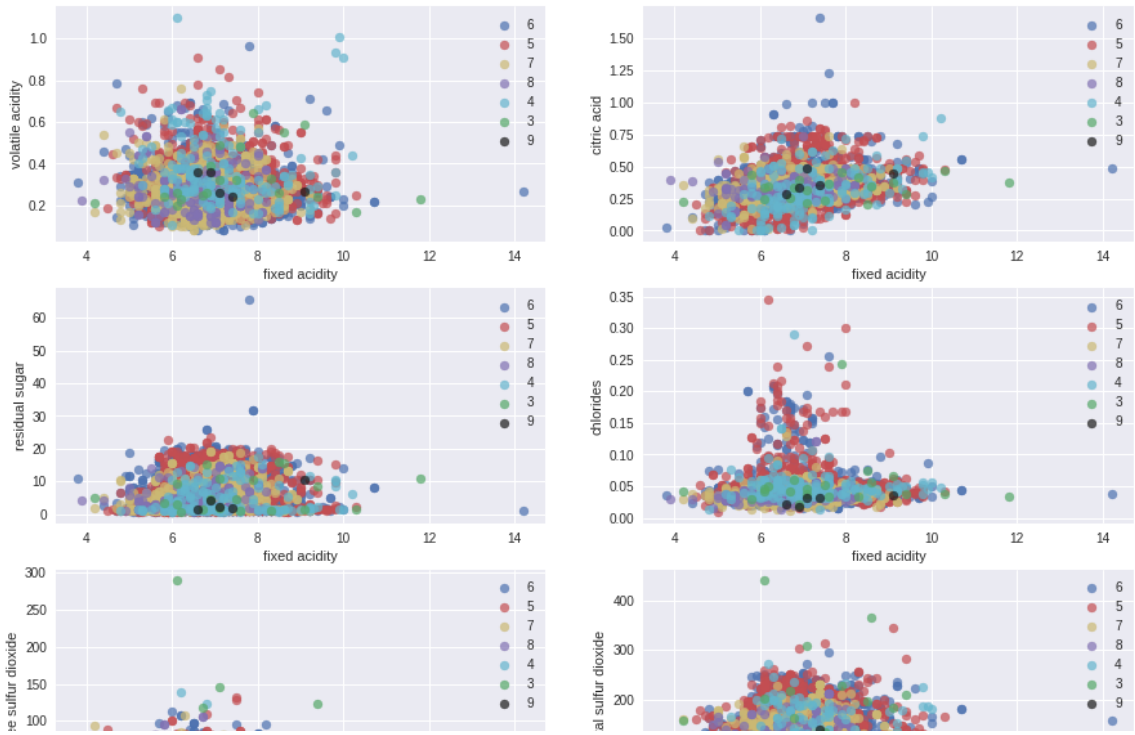
```
Imported AutoViz_Class version: 0.0.81. Call using:
    from autoviz.AutoViz_Class import AutoViz_Class
    AV = AutoViz_Class()
    AV.AutoViz(filename, sep=',', depVar='', dfte=None, header=0, verbose=
0,
                            lowess=False,chart_format='svg',max_rows_analy
zed=150000,max_cols_analyzed=30)
Note: verbose=0 or 1 generates charts and displays them in your local Jupy
ter notebook.
      verbose=2 saves plots in your local machine under AutoViz_Plots dire
ctory and does not display charts.
Shape of your Data Set: (4898, 12)
############### C L A S S I F Y I N G  V A R I A B L E S  #################
###
Classifying variables in data set...
    Number of Numeric Columns =  11
    Number of Integer-Categorical Columns =  0
    Number of String-Categorical Columns =  0
    Number of Factor-Categorical Columns =  0
    Number of String-Boolean Columns =  0
    Number of Numeric-Boolean Columns =  0
    Number of Discrete String Columns =  0
    Number of NLP String Columns =  0
    Number of Date Time Columns =  0
    Number of ID Columns =  0
    Number of Columns to Delete =  0
    11 Predictors classified...
        This does not include the Target column(s)
        No variables removed since no ID or low-information variables foun
d in data set

################ Multi_Classification VISUALIZATION Started ##############
#######
Data Set Shape: 4898 rows, 12 cols
Data Set columns info:
* fixed acidity: 0 nulls, 68 unique vals, most common: {6.8: 308, 6.6: 29
0}
* volatile acidity: 0 nulls, 125 unique vals, most common: {0.28: 263, 0.2
4: 253}
* citric acid: 0 nulls, 87 unique vals, most common: {0.3: 307, 0.28: 282}
* residual sugar: 0 nulls, 310 unique vals, most common: {1.2: 187, 1.4: 1
84}
* chlorides: 0 nulls, 160 unique vals, most common: {0.044000000000000004:
201, 0.036000000000000004: 200}
* free sulfur dioxide: 0 nulls, 132 unique vals, most common: {29.0: 160,
31.0: 132}
* total sulfur dioxide: 0 nulls, 251 unique vals, most common: {111.0: 69,
113.0: 61}
* density: 0 nulls, 890 unique vals, most common: {0.992: 64, 0.9928: 61}
* pH: 0 nulls, 103 unique vals, most common: {3.14: 172, 3.16: 164}
* sulphates: 0 nulls, 79 unique vals, most common: {0.5: 249, 0.46: 225}
* alcohol: 0 nulls, 103 unique vals, most common: {9.4: 229, 9.5: 228}
* quality: 0 nulls, 7 unique vals, most common: {6: 2198, 5: 1457}
----------------------------------------------------------------------
   Columns to delete:
'   []'
   Boolean variables %s
'   []'
   Categorical variables %s
'   []'
   Continuous variables %s
```

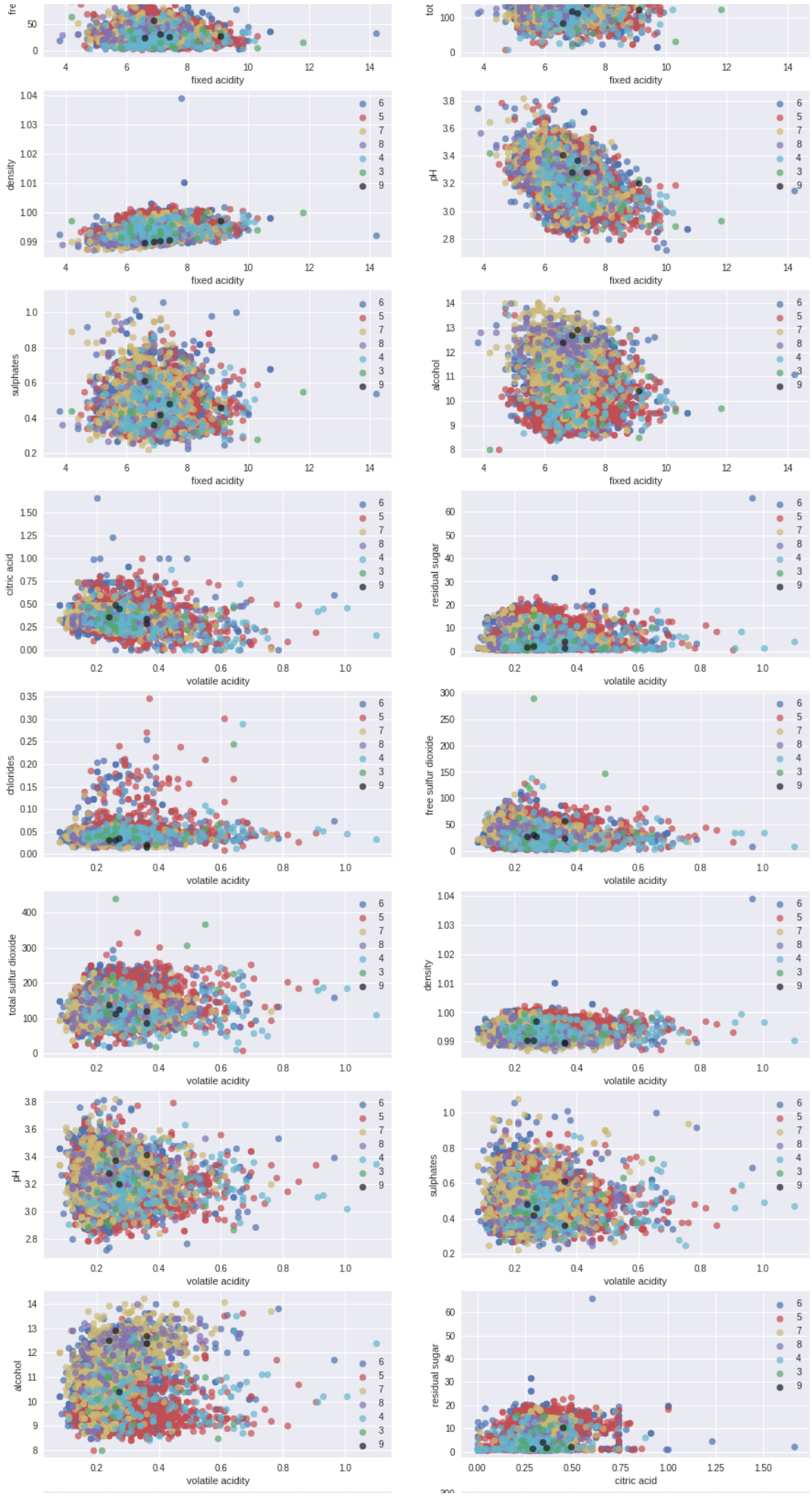
```
("   ['fixed acidity', 'volatile acidity', 'citric acid', 'residual suga
r', "
 "'chlorides', 'free sulfur dioxide', 'total sulfur dioxide', 'density',
'pH', "
 "'sulphates', 'alcohol']")
   Discrete string variables %s
'   []'
   Date and time variables %s
'   []'
   ID variables %s
'   []'
   Target variable %s
'   quality'
```

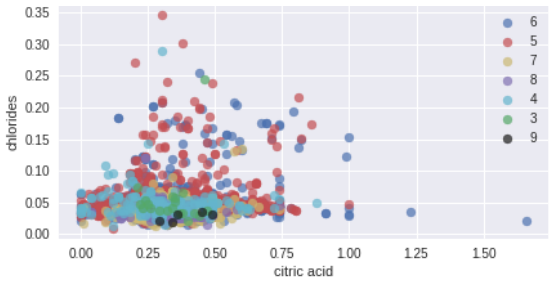

Scatter Plot of Continuous Variable vs Target (jitter=0.50)
fixed acidity
volatile acidity
citric acid
residual sugar
chlorides
free sulfur dioxide
total sulfur dioxide
density
pH
sulphates
alcohol
quality

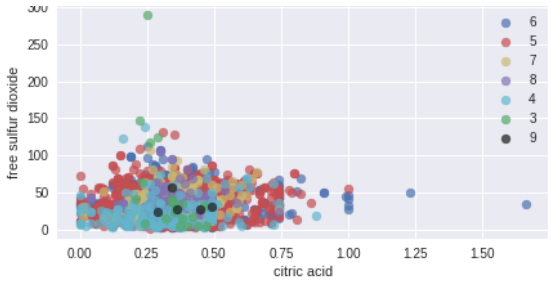
```
Total Number of Scatter Plots = 66
```

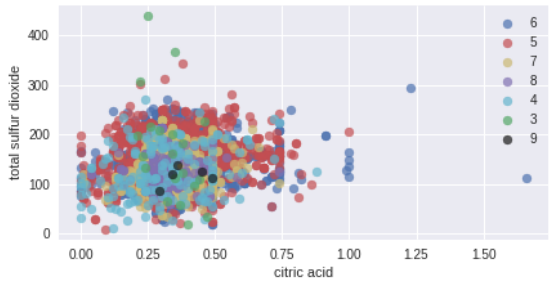
Pair-wise Scatter Plot of all Continuous Variables

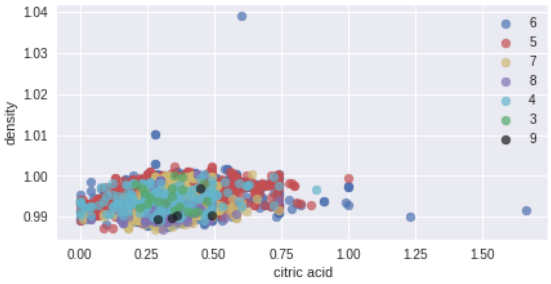


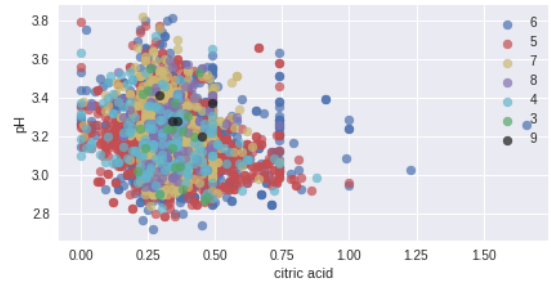

fixed acidity
density
pH
sulphates
alcohol
citric acid
residual sugar
volatile acidity
chlorides
free sulfur dioxide
total sulfur dioxide
6
5
7
8
4
3
9

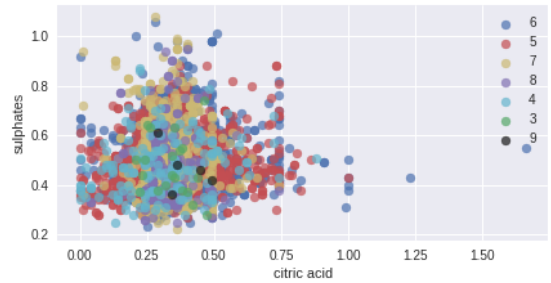

chlorides
citric acid
6
5
7
8
4
3
9

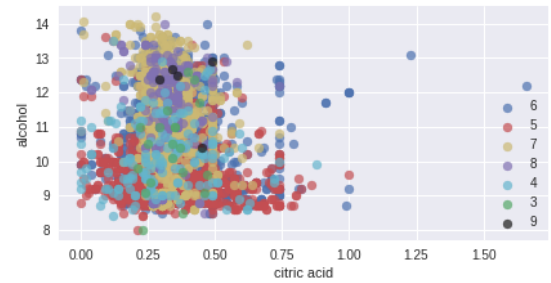

free sulfur dioxide
citric acid
6
5
7
8
4
3
9

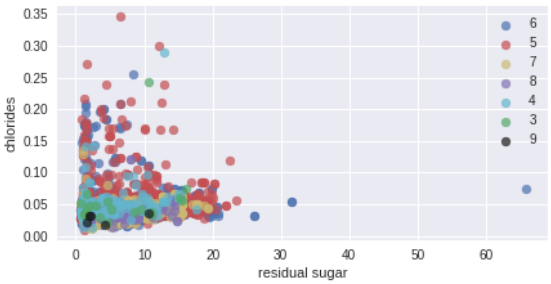

total sulfur dioxide
citric acid
6
5
7
8
4
3
9

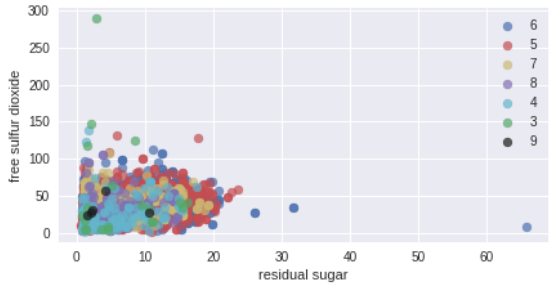

density
citric acid
6
5
7
8
4
3
9

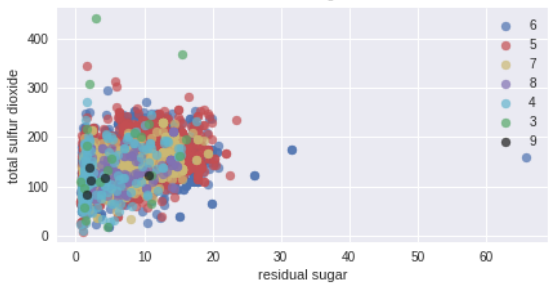

pH
citric acid
6
5
7
8
4
3
9

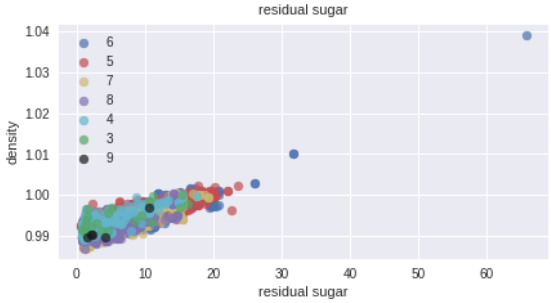

sulphates
citric acid
6
5
7
8
4
3
9


alcohol
citric acid
6
5
7
8
4
3
9


chlorides
residual sugar
6
5
7
8
4
3
9

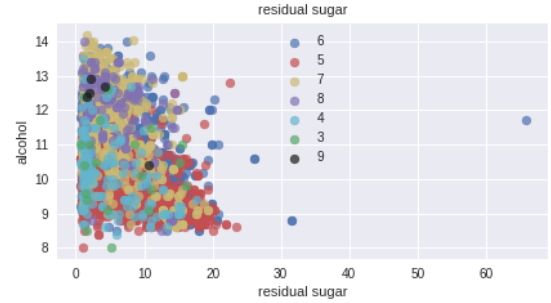

free sulfur dioxide
residual sugar
6
5
7
8
4
3
9


total sulfur dioxide
residual sugar
6
5
7
8
4
3
9


density
residual sugar
6
5
7
8
4
3
9

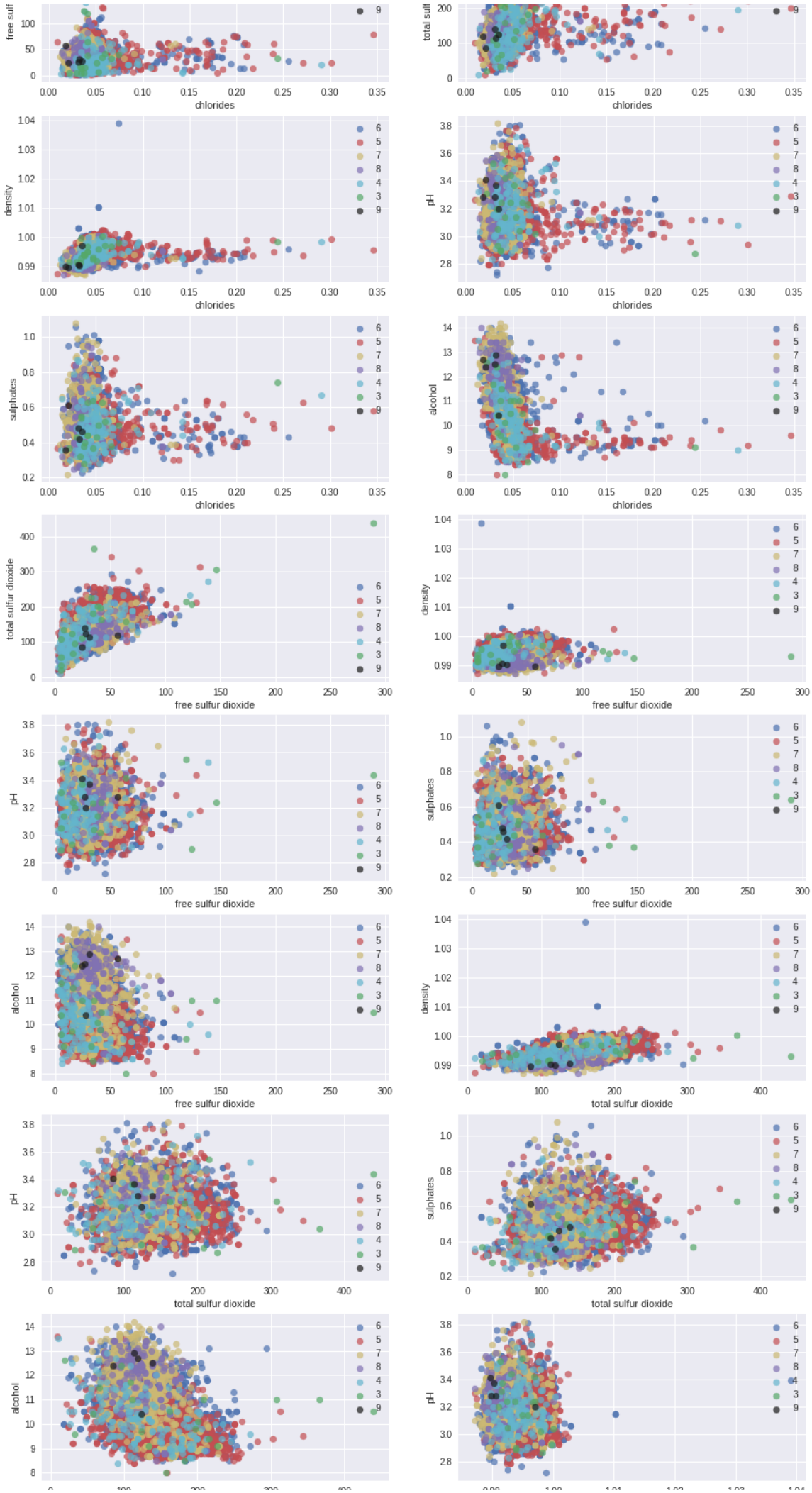

pH
residual sugar
6
5
7
8
4
3
9

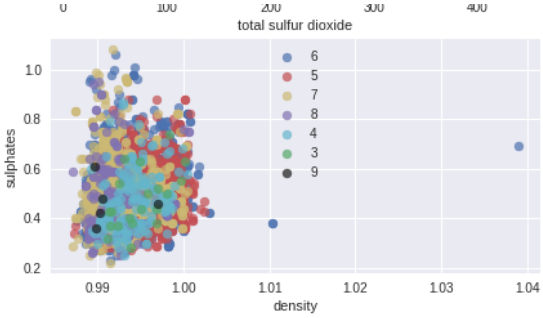

sulphates
residual sugar
6
5
7
8
4
3
9

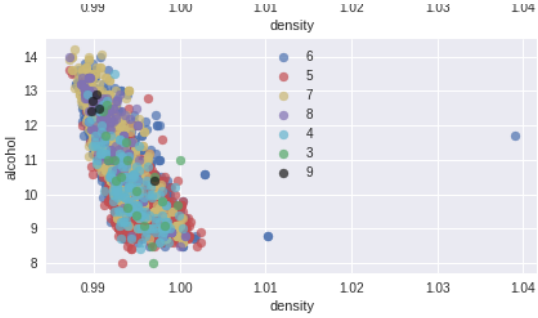

alcohol
residual sugar
6
5
7
8
4
3
9

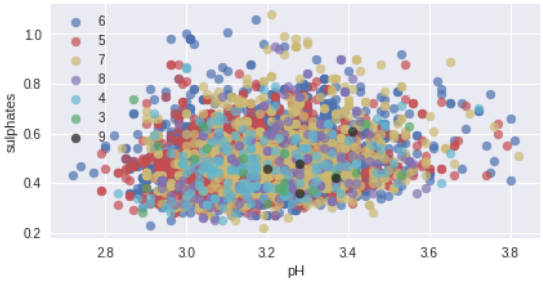

6
5
7
8
4
3

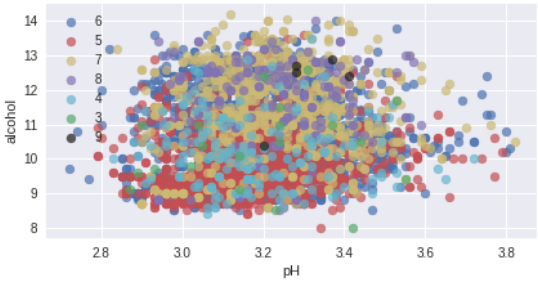

6
5
7
8
4
3

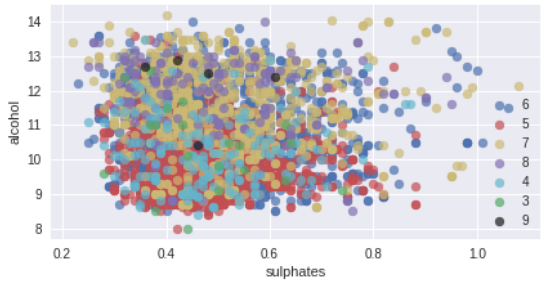

free sulf
chlorides
total sulf
chlorides
density
chlorides
pH
chlorides
sulphates
chlorides
alcohol
chlorides
total sulfur dioxide
free sulfur dioxide
density
free sulfur dioxide
pH
free sulfur dioxide
sulphates
free sulfur dioxide
alcohol
free sulfur dioxide
density
total sulfur dioxide
pH
total sulfur dioxide
sulphates
total sulfur dioxide
alcohol
pH
6
5
7
8
4
3
9

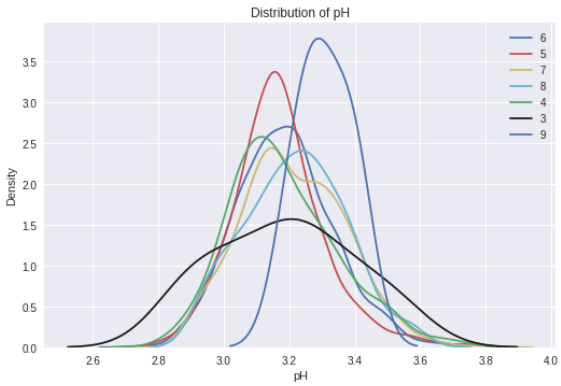

total sulfur dioxide
sulphates
density
6
5
7
8
4
3
9

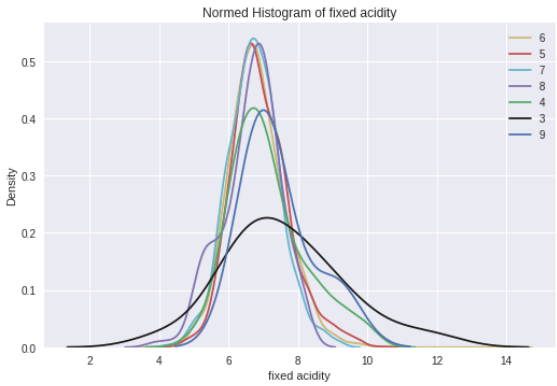

density
alcohol
density
6
5
7
8
4
3
9

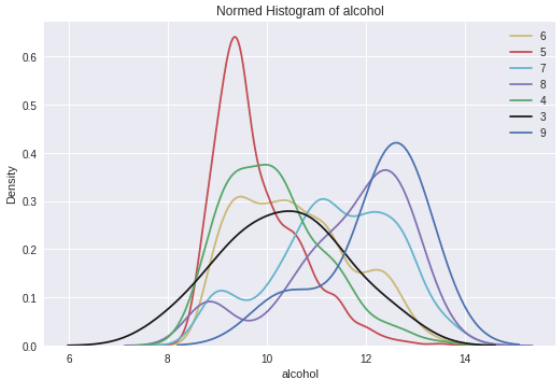

sulphates
pH
6
5
7
8
4
3
9

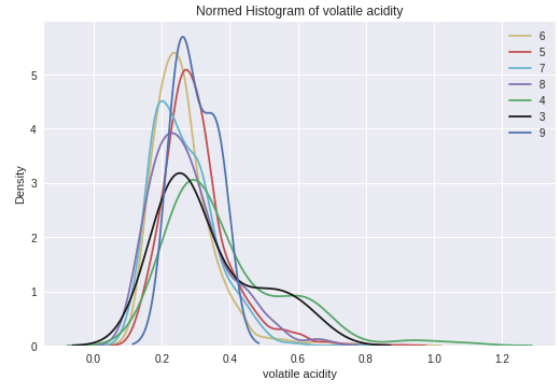

alcohol
pH
6
5
7
8
4
3
9

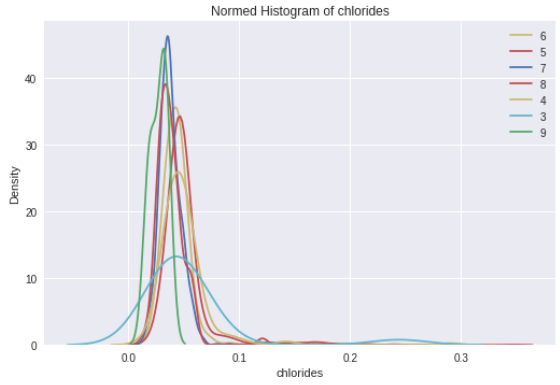

alcohol
sulphates
6
5
7
8
4
3
9

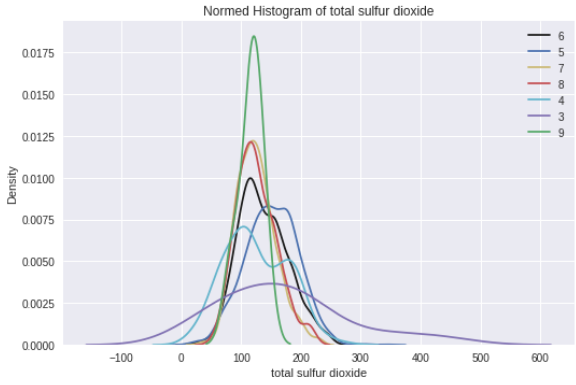

Distribution of pH
Density
pH
6
5
7
8
4
3
9

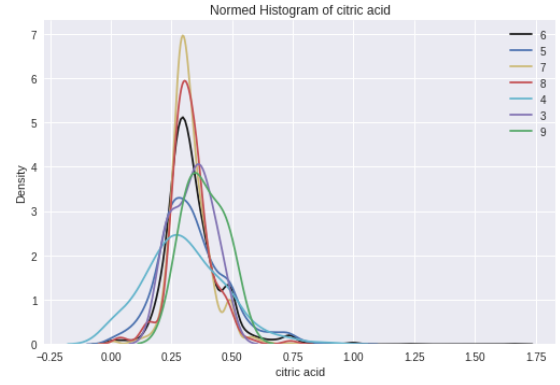

Normed Histogram of fixed acidity
Density
fixed acidity
6
5
7
8
4
3
9

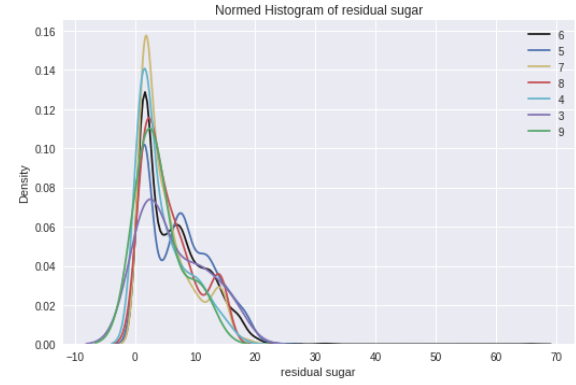

Normed Histogram of alcohol
Density
alcohol
6
5
7
8
4
3
9

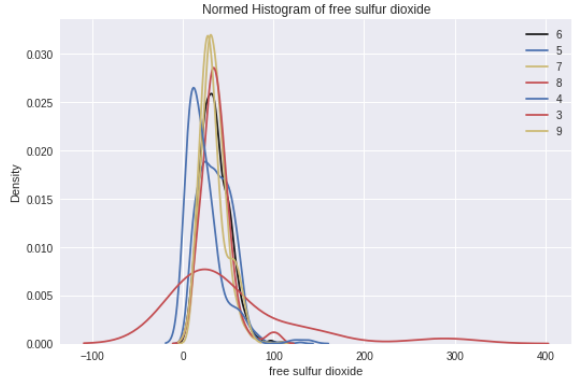

Normed Histogram of volatile acidity
Density
volatile acidity
6
5
7
8
4
3
9

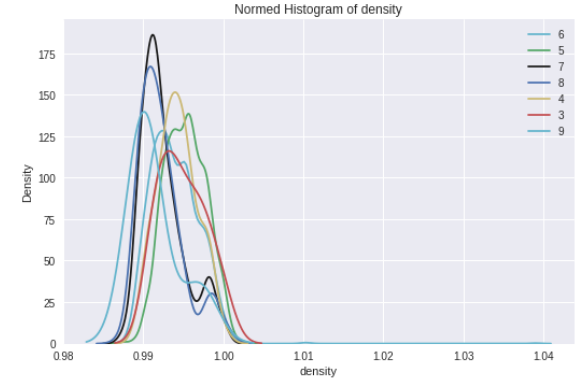

Normed Histogram of chlorides
Density
chlorides
6
5
7
8
4
3
9

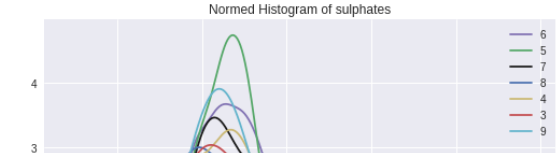

Normed Histogram of total sulfur dioxide
Density
total sulfur dioxide
6
5
7
8
4
3
9

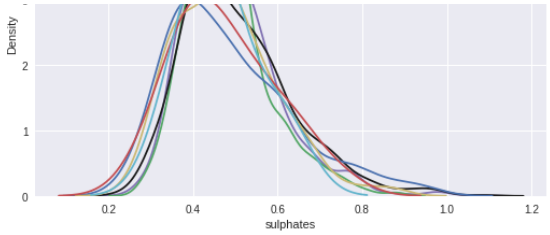

Normed Histogram of citric acid
Density
citric acid
6
5
7
8
4
3
9


Normed Histogram of residual sugar
Density
residual sugar
6
5
7
8
4
3
9

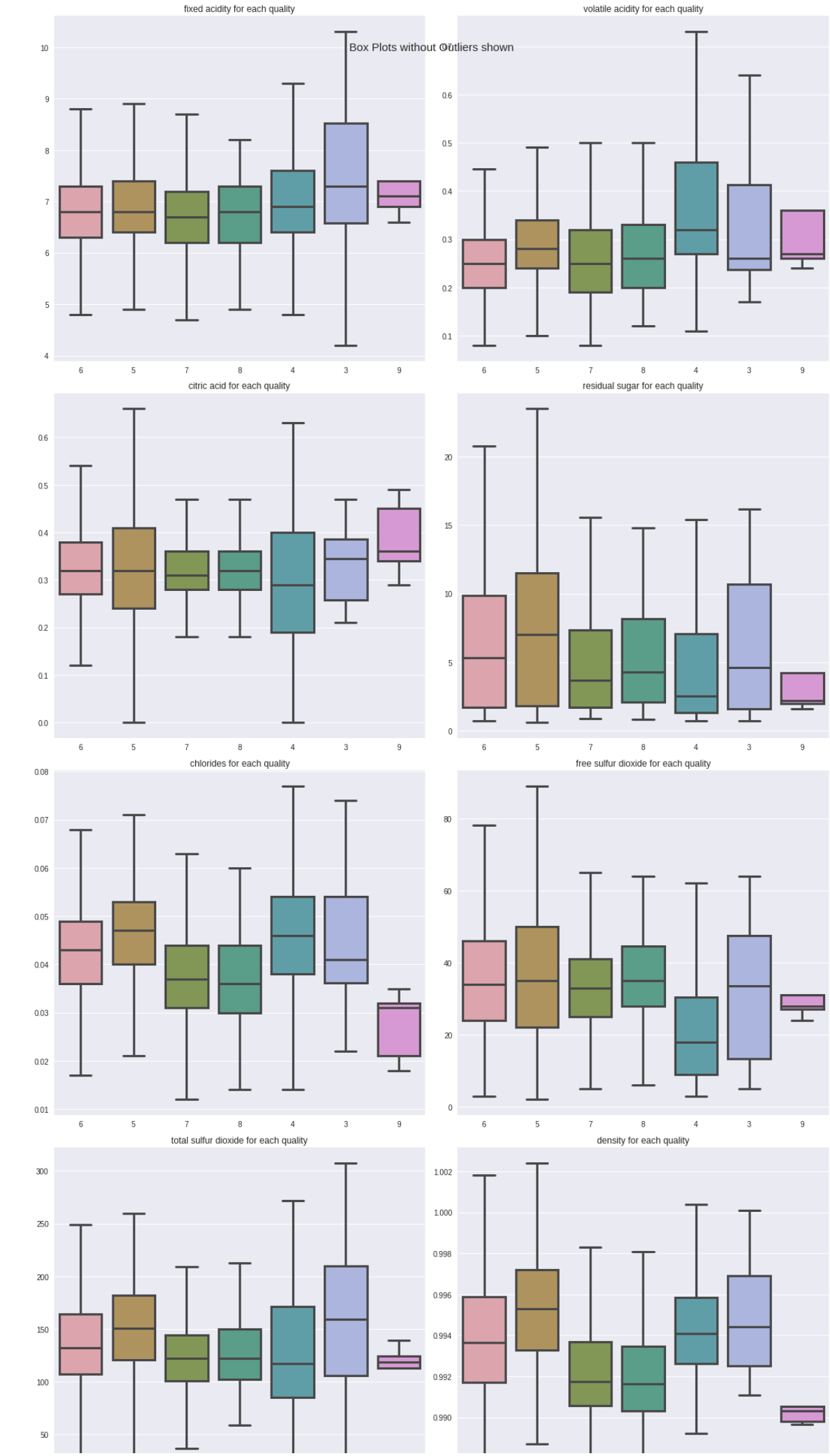

Normed Histogram of free sulfur dioxide
Density
free sulfur dioxide
6
5
7
8
4
3
9


Normed Histogram of density
Density
density
6
5
7
8
4
3
9

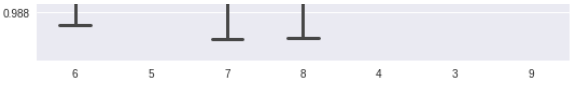

Normed Histogram of sulphates
6
5
7
8
4
3
9

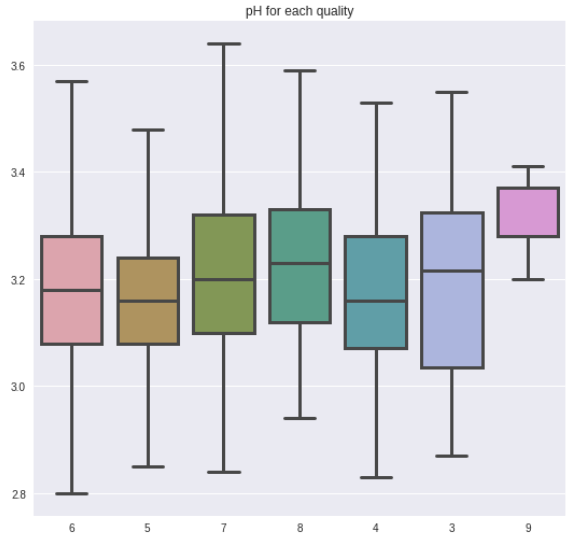

Density
sulphates
0
1
2
0.2
0.4
0.6
0.8
1.0
1.2

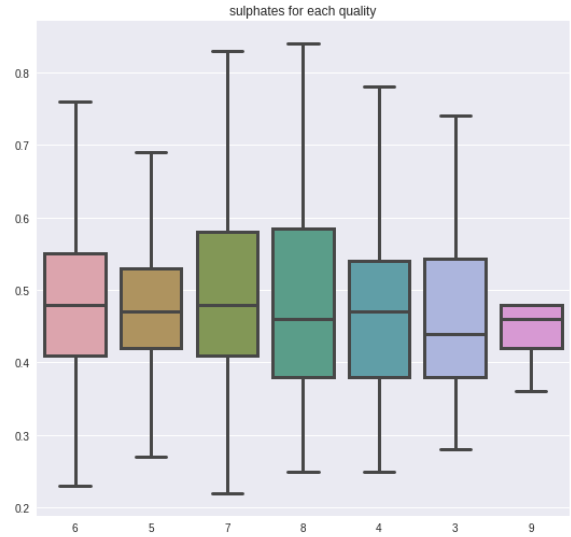

quality : Distribution of Target Variable
Percentage Distribution of Target = quality
0.45
0.3
0.18
0.04
0.03
0.0
0.0
Freq Distribution of Target Variable = quality
2198
1457
880
175
163
20
5

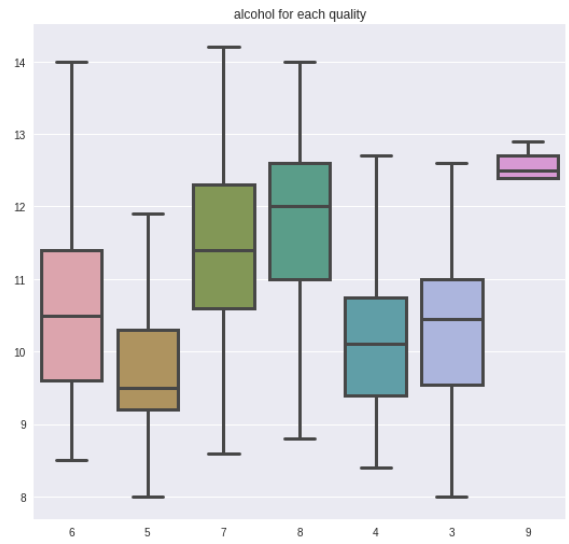

Box Plots without Outliers shown
fixed acidity for each quality
volatile acidity for each quality
citric acid for each quality
residual sugar for each quality
chlorides for each quality
free sulfur dioxide for each quality
total sulfur dioxide for each quality
density for each quality

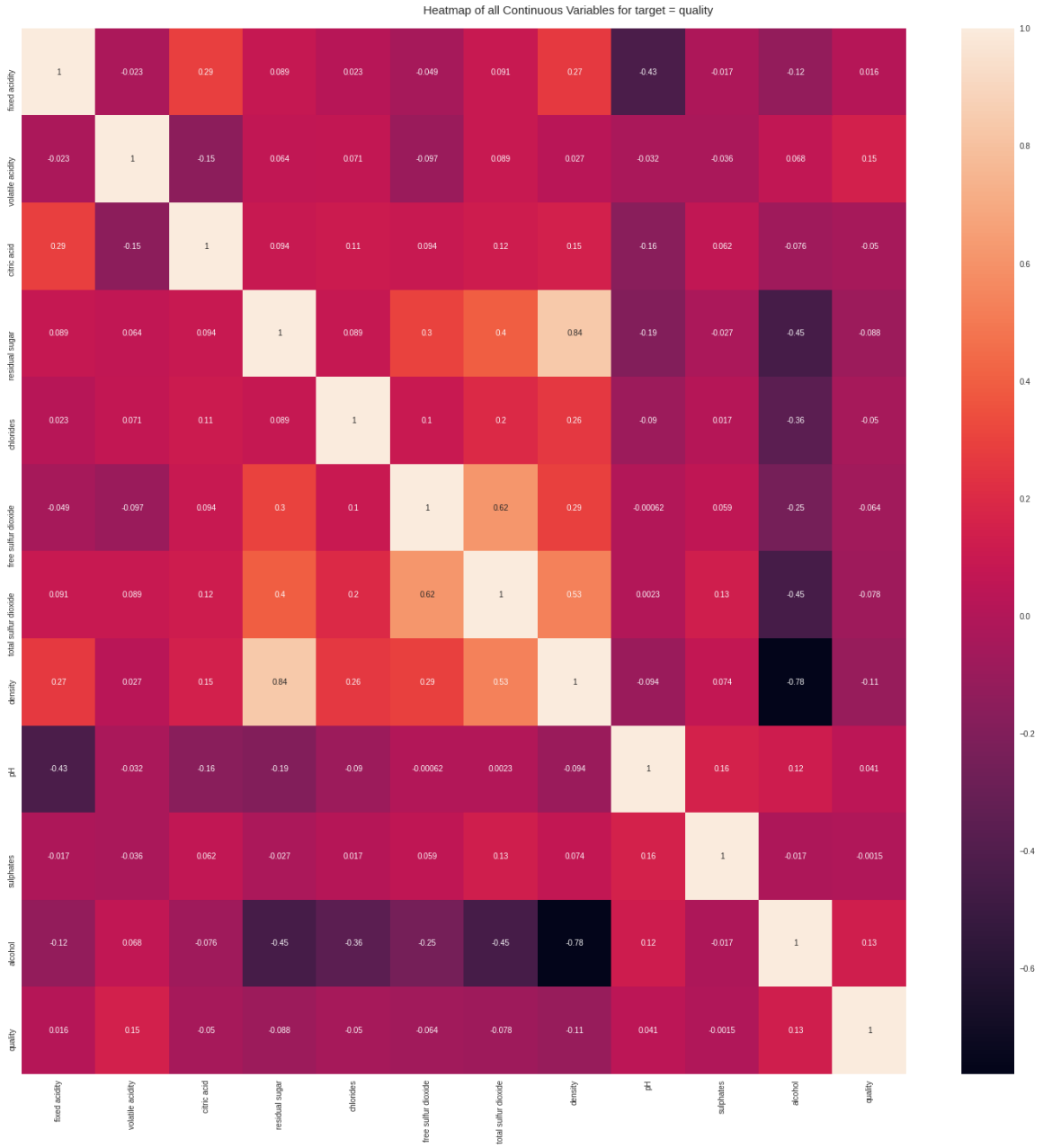

0
6
5
7
8
4
3
9


0.988
6
5
7
8
4
3
9


pH for each quality
3.6
3.4
3.2
3.0
2.8
6
5
7
8
4
3
9


sulphates for each quality
0.8
0.7
0.6
0.5
0.4
0.3
0.2
6
5
7
8
4
3
9


alcohol for each quality
14
13
12
11
10
9
8
6
5
7
8
4
3
9


Heatmap of all Continuous Variables for target = quality
fixed acidity
volatile acidity
citric acid
residual sugar
chlorides
free sulfur dioxide
total sulfur dioxide
density
pH
sulphates
alcohol
quality
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6

```
No categorical or boolean vars in data set. Hence no pivot plots...
No categorical or numeric vars in data set. Hence no bar charts.
Time to run AutoViz (in seconds) = 41.473

 ###################### VISUALIZATION Completed ########################
```

In [ ]:

```
# divide the data to features and target
t = whitewines_df['quality'].copy()
X = whitewines_df.drop(['quality'], axis=1)
print('t')
display(t)
print()
print('X')
display(X)
```

t

```
0       6
1       6
2       6
3       6
4       6
       ..
4893    6
4894    5
4895    6
4896    7
4897    6
Name: quality, Length: 4898, dtype: int64
```

X

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

4898 rows × 11 columns

Let's see how many different scores we have in the target.

In [ ]:

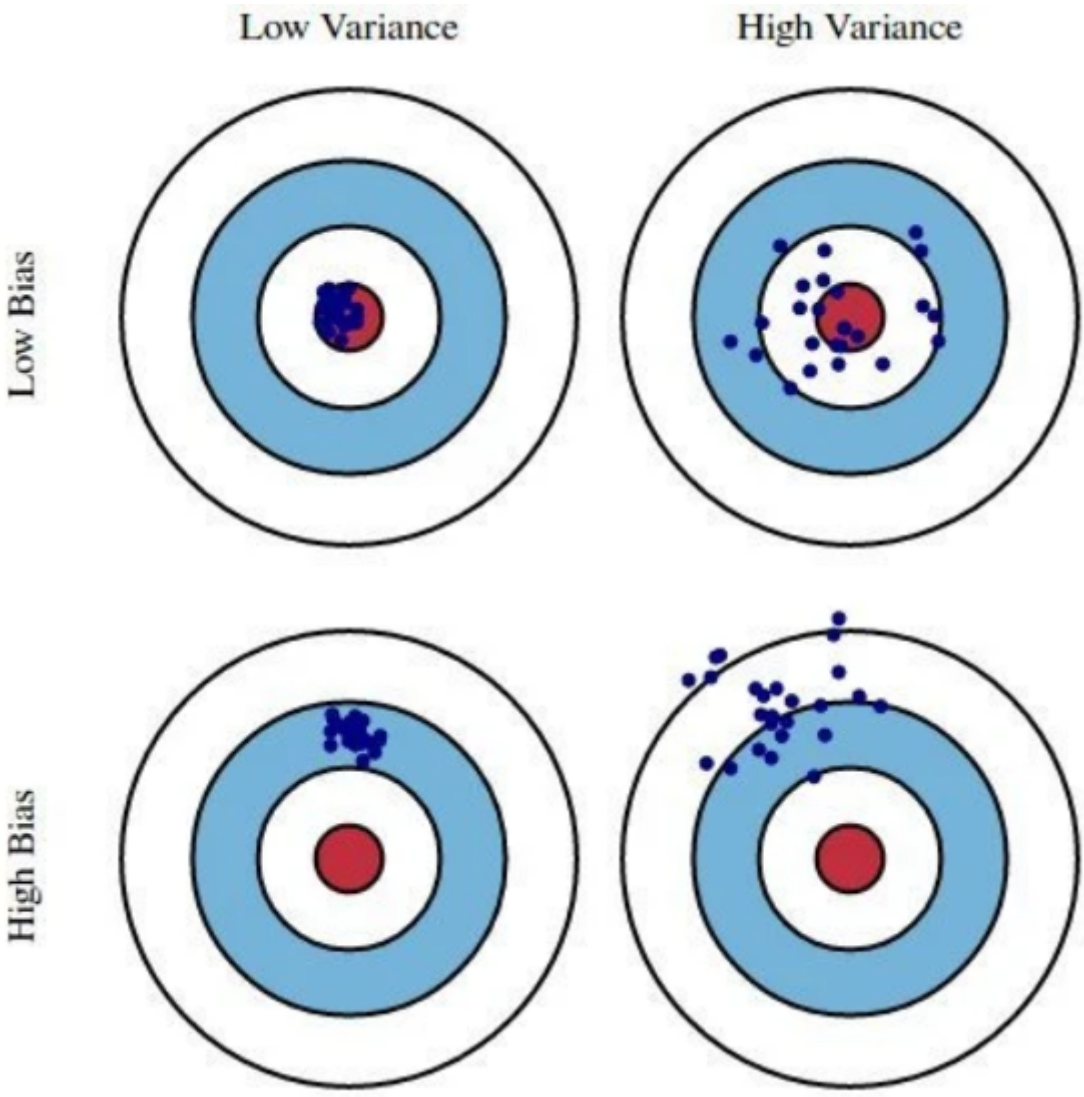
```
t.unique()
```

Out[ ]:

```
array([6, 5, 7, 8, 4, 3, 9])
```

We can see that we have only 7 different scores (ordinal target).
We can try to use regression models and classification models on this dataset.

Regularization



When we have a case of High-Variance, we can use Regularizations, to reduce it.
With regularization, we can control the size of the model weights and thus control the variance.
We control the size of the model weights by adding a term to the loss function.
This term is called a *penalty* to high weights.

We have learned about three regularization techniques:

1. L1 (also called Lasso).
2. L2 (also called Ridge).
3. Elastic Net (a combination of Lasso and Ridge).

L1 - Lasso

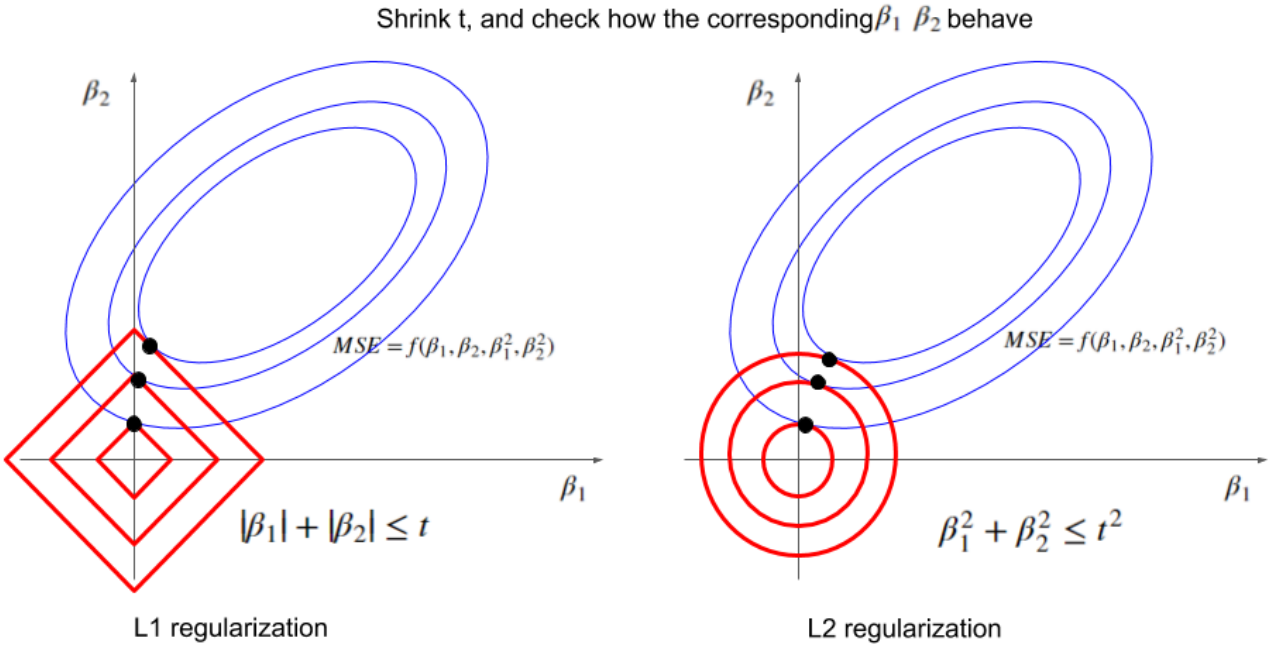
$$\sum_{i=1}^{n}(Y_i - \sum_{j=1}^{p} X_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

The penalty is fixed for every weight.
It is going up or down with fixed-size steps.
The unnecessary weights tend to get to zero (it is practically feature selection).

L2 - Ridge

$$\sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

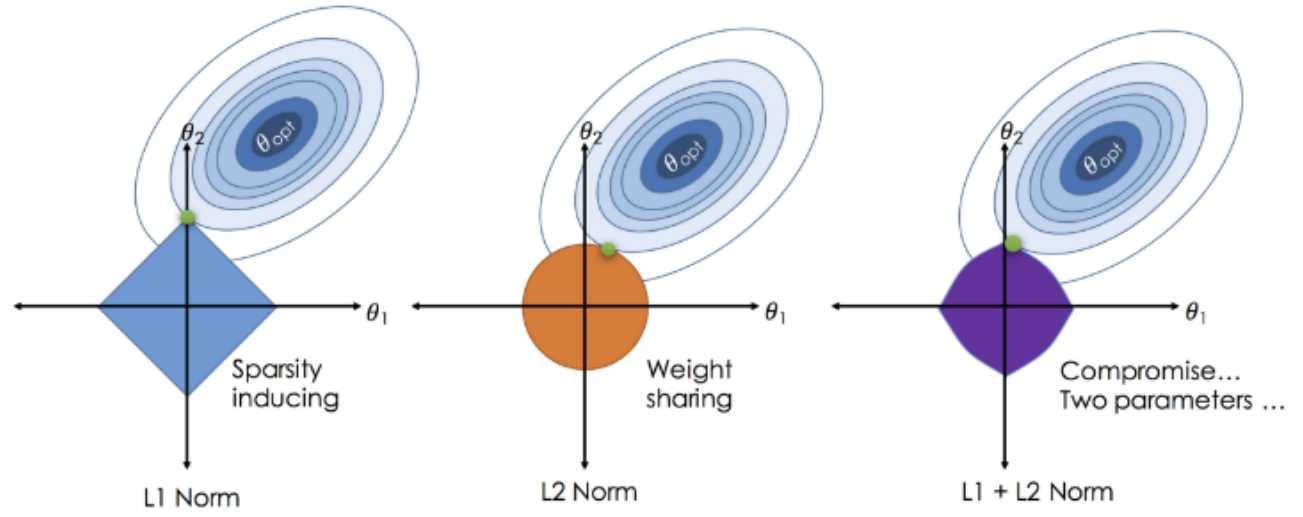
When the weights are big, this penalty adds big numbers to the loss function and thus reducing the weights by a lot.
When the weights are small, this penalty adds small numbers to the loss function and thus reducing the weights a little bit.
The weights that are unnecessary still stays in the model (but with small values).



ElasticNet

$$\frac{\sum_{i=1}^{n}(y_i - x_i^J \hat{\beta})^2}{2n} + \lambda \left(\frac{1-\alpha}{2} \sum_{j=1}^{m} \hat{\beta}_j^2 + \alpha \sum_{j=1}^{m} |\hat{\beta}_j| \right)$$

Combination of L1 (Lasso) and L2 (Ridge).



We can use these regularizations as Hyper-Parameters in Scikit-learn SGDRegressor (https://scikit-

In [ ]:

```
# print lasso, ridge and elasticnet scores as regression
from sklearn.model_selection import cross_val_score

sgd_lasso_reg = SGDRegressor(penalty='l1', random_state=1)
sgd_ridge_reg = SGDRegressor(penalty='l2', random_state=1)
sgd_elastic_reg = SGDRegressor(penalty='elasticnet', random_state=1)

print("R2 score for regression:")
print('sgd_lasso', cross_val_score(make_pipeline(StandardScaler(), sgd_lasso_reg), X, t
, cv=15).mean())
print('sgd_ridge', cross_val_score(make_pipeline(StandardScaler(), sgd_ridge_reg), X, t
, cv=15).mean())
print('sgd_elastic', cross_val_score(make_pipeline(StandardScaler(), sgd_elastic_reg),
X, t, cv=15).mean())
```

```
R2 score for regression:
sgd_lasso 0.24338626485383555
sgd_ridge 0.24338463447414035
sgd_elastic 0.24339760603675203
```

Let's check the accuracy score of the regression models.
We can do it with Scikit-learn make_scorer (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.make_scorer.html).

In [ ]:

```
# create accuracy score for ordinal predictions
from sklearn.metrics import make_scorer, accuracy_score

def get_accurate_ordinal_preds_from_numeric_preds(preds, min=None, max=None):
    if min is None:
        min = round(min(preds))
    if max is None:
        max = round(max(preds))
    preds = np.asarray(preds).ravel()
    return np.array([round(p) if min <= p and p <= max else min if p < min else max for
p in preds])

def accuracy_for_ordinal(t, y):
    min_ord = min(t)
    max_ord = max(t)
    y_ord = get_accurate_ordinal_preds_from_numeric_preds(y, min=min_ord, max=max_ord)
    return accuracy_score(t, y_ord)

print("Accuracy score for regression:")
print('sgd_lasso', cross_val_score(make_pipeline(StandardScaler(), sgd_lasso_reg), X, t
, cv=15, scoring=make_scorer(accuracy_for_ordinal)).mean())
print('sgd_ridge', cross_val_score(make_pipeline(StandardScaler(), sgd_ridge_reg), X, t
, cv=15, scoring=make_scorer(accuracy_for_ordinal)).mean())
print('sgd_elastic', cross_val_score(make_pipeline(StandardScaler(), sgd_elastic_reg),
X, t, cv=15, scoring=make_scorer(accuracy_for_ordinal)).mean())
```

```
Accuracy score for regression:
sgd_lasso 0.5188189089635591
sgd_ridge 0.518818283584423
sgd_elastic 0.5190227825619281
```

Let's try the classification approach and use SGDClassifier.

In [ ]:

```
# print lasso, ridge and elasticnet scores as classification
sgd_lasso_cls =SGDClassifier(penalty='l1', random_state=1)
sgd_ridge_cls = SGDClassifier(penalty='l2', random_state=1)
sgd_elastic_cls = SGDClassifier(penalty='elasticnet', random_state=1)

print("Accuracy score for classification:")
print('sgd_lasso', cross_val_score(make_pipeline(StandardScaler(), sgd_lasso_cls), X, t
, cv=15).mean())
print('sgd_ridge', cross_val_score(make_pipeline(StandardScaler(), sgd_ridge_cls), X, t
, cv=15).mean())
print('sgd_elastic', cross_val_score(make_pipeline(StandardScaler(), sgd_elastic_cls),
X, t, cv=15).mean())
```

```
Accuracy score for classification:
sgd_lasso 0.4877738378892203
sgd_ridge 0.4798208914154206
sgd_elastic 0.4949325528601715
```

We can see that the classifiers predicted here worse than the regressors.
We can try and create [ordinal classifiers (https://towardsdatascience.com/simple-trick-to-train-an-ordinal-regression-with-any-classifier-6911183d2a3c)](https://towardsdatascience.com/simple-trick-to-train-an-ordinal-regression-with-any-classifier-6911183d2a3c) that will use the fact that there is an order to the labels.

In [ ]:

```
from sklearn.base import clone
import sklearn.metrics

class OrdinalClassifier():

    def __init__(self, clf, class_opt=None):
        self.clf = clf
        self.clfs = {}
        self.class_opt = class_opt

    def fit(self, X, y):
        self.unique_class = np.sort(np.unique(y))
        if self.unique_class.shape[0] > 2:
            for i in range(self.unique_class.shape[0]-1):
                # for each k - 1 ordinal value we fit a binary classification problem
                binary_y = (y > self.unique_class[i]).astype(np.uint8)
                clf = clone(self.clf)
                clf.fit(X, binary_y)
                self.clfs[i] = clf

    def predict_proba(self, X):
        clfs_predict = {k:self.clfs[k].predict_proba(X) for k in self.clfs}
        norm_unique = self.unique_class - min(self.unique_class) # Added this variable
 to make the model work with the wine data
        predicted = []
        for i,y in enumerate(norm_unique):
            if i == 0:
                # V1 = 1 - Pr(y > V1)
                predicted.append(1 - clfs_predict[y][:,1])
            elif y in clfs_predict:
                # Vi = Pr(y > Vi-1) - Pr(y > Vi)
                 predicted.append(clfs_predict[y-1][:,1] - clfs_predict[y][:,1])
            else:
                # Vk = Pr(y > Vk-1)
                predicted.append(clfs_predict[y-1][:,1])
        return np.vstack(predicted).T

    def predict(self, X):
        return (np.argmax(self.predict_proba(X), axis=1) + min(self.unique_class)) # Ad
ding back the min value to align the predictions to the data

    def score(self, X, t):
      y = self.predict(X)
      return np.array(metrics.accuracy_score(np.array(t), y))
```

In [ ]:

```
from sklearn.model_selection import cross_val_score

ord_classifier_ridge = OrdinalClassifier(SGDClassifier(penalty='l2', random_state=1, lo
ss='log'), np.array([i for i in range(11)]))
ord_classifier_lasso = OrdinalClassifier(SGDClassifier(penalty='l1', random_state=1, lo
ss='log'), np.array([i for i in range(11)]))
ord_classifier_elastic = OrdinalClassifier(SGDClassifier(penalty='elasticnet', random_s
tate=1, loss='log'), np.array([i for i in range(11)]))


print("Accuracy score for:")
print("ord_classifier_ridge:", cross_val_score(make_pipeline(StandardScaler(), ord_clas
sifier_ridge), X, t, cv=15, scoring=make_scorer(accuracy_score)).mean())
print("ord_classifier_lasso:", cross_val_score(make_pipeline(StandardScaler(), ord_clas
sifier_lasso), X, t, cv=15, scoring=make_scorer(accuracy_score)).mean())
print("ord_classifier_elastic:", cross_val_score(make_pipeline(StandardScaler(), ord_cl
assifier_elastic), X, t, cv=15, scoring=make_scorer(accuracy_score)).mean())
```

```
Accuracy score for:
ord_classifier_ridge: 0.5167851760129578
ord_classifier_lasso: 0.5104388285398023
ord_classifier_elastic: 0.4991982639475181
```

So we have seen two methods to create an ordinal model.
The first method is to create it out of a regression model, and the second one is by using a classifier (The second method is explained in the above article).

We can see that the first method works a little better on this data.

Hyper-Parameters Search

Most of our models have a lot of parameters that can be adjusted.
Each parameter value can make our model better (or worse).
We want to be able to find the best hyperparameters for our models.
We have two approaches:

1. Grid Search
2. Random Search

Grid Search

When we want to check every parameter possible, we will use Grid Search.
We will try all combinations of parameters and find the best one, that gives us the best score.
This may be a little exhaustive, especially when we want to check a lot of parameters and values.

Random Search

We can choose to get random combinations of parameters and check the score on them.
This will not be as accurate as Grid Search, but it will take less time.

Let's use Scikit-learn GridSearchCV (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html).

In [ ]:

```
# train with grid search and get best parameters
from sklearn.model_selection import GridSearchCV

X_normalized = StandardScaler().fit_transform(X)
hyper_parameters = {'penalty': ('l2', 'l1', 'elasticnet'), 'alpha':[0.0001, 0.001, 0.01
, 0.1]}

gs_model = GridSearchCV(SGDClassifier(random_state=1), hyper_parameters).fit(X_normaliz
ed, t)
print('Accuracy score for classification:')
print('gs_model', gs_model.best_score_)
print('best params', gs_model.best_params_)
```

```
Accuracy score for classification:
gs_model 0.5149144274665944
best params {'alpha': 0.001, 'penalty': 'elasticnet'}
```

We can see that the best parameters on this model (obtained with Grid Search) were `penalty=elasticnet` and `alpha=0.001`.
It may change with a different `random_state`.

Now let's try Scikit-learn RandomizedSearchCV (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html).
We will use Scipy stats.uniform (https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.uniform.html) to get uniformly randomize values of `alpha`.
We will use Numpy random.seed (https://numpy.org/doc/stable/reference/random/generated/numpy.random.seed.html) to make sure that we get the same result each time we run this cell.

In [ ]:

```
# train with random search and get best parameters
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import uniform

np.random.seed(1)
distributions = dict(alpha=uniform(loc=0, scale=1), penalty=['l2', 'l1', 'elasticnet'])

rs_model = RandomizedSearchCV(SGDClassifier(), distributions, random_state=1).fit(X_nor
malized, t)
print('Accuracy score for classification:')
print('rs_model', rs_model.best_score_)
print('best params', rs_model.best_params_)
```

```
Accuracy score for classification:
rs_model 0.4971491109211815
best params {'alpha': 0.417022004702574, 'penalty': 'l2'}
```

The best parameters on this model (obtained with Random Search) were `penalty=l2` and `alpha=0.417022004702574`.
The Accuracy score of the Randomized Search is a little less than the score of the Grid Search, but it may change with a different random seed.

Ensembles

We can use a collection of models to make more accurate predictions and lower the variance.
If we use regression, we can take the mean of all the predictions of the models.
If we use classification, we can take the mean of all the probabilities of the model or choose the class that most of the models chose for some sample.
It is like "The wisdom of the crowd".
One model may be wrong, but a lot of different models are less prone to errors.

We are going to use two types of ensembles:

1. Bagging (with NFold or with Bootstrap).
2. Boosting.

Bagging

We create a few bags of samples from the original dataset.
We train a model on each of the bags of samples, and we return the combined score.

The bags can be created using NFold (same as KFold CV).
we simply divide the data into N parts and use KFold CV (where `K=N`).
We save all the N (or K) models and use them as an ensemble.

The bags can be created using Bootstrap.
We draw samples out of the dataset (with replacement) and train the model on each group of samples.
We save all the models and use them as an ensemble.

Boosting

We create a model and train it on the data.
We take the samples that the model predicted incorrectly and multiply them (thus giving them more weight in the next training).
We do this until we have few models, each of them is an expert on some type of samples.
We combine all the model's predictions and return a combined score.
There are few boosting algorithems (AdaBoost (https://www.youtube.com/watch?v=LsK-xG1cLYA&ab_channel=StatQuestwithJoshStarmer), GradientBoost (https://www.youtube.com/watch?v=3CC4N4z3GJc&ab_channel=StatQuestwithJoshStarmer), etc.).

Let's start with Scikit-learn BaggingClassifier (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html).
When we use it with `bootstrap=False` we will address it as NFold, and when we use it with `bootstrap=True` we will address it as Bootstrap.
Let's start with NFold Bagging.

In [ ]:

```
# get score with nfold bagging
from sklearn.ensemble import BaggingClassifier

bag_fold_model = BaggingClassifier(base_estimator=SGDClassifier(), n_estimators=20, ran
dom_state=1, bootstrap=False).fit(X_normalized, t)
print('Accuracy score for classification:')
print('bag_fold_model', bag_fold_model.score(X_normalized, t).mean())
```

```
Accuracy score for classification:
bag_fold_model 0.5253164556962026
```

Let's try the Bootstrap Bagging.

In [ ]:

```
# get score with bootstrap bagging
bag_boot_model = BaggingClassifier(base_estimator=SGDClassifier(), n_estimators=20, ran
dom_state=1, bootstrap=True).fit(X_normalized, t)
print('Accuracy score for classification:')
print('bag_boot_model', bag_boot_model.score(X_normalized, t).mean())
```

```
Accuracy score for classification:
bag_boot_model 0.5236831359738668
```

In our case, NFold Bagging got better results but the difference is small and it may change if we use bigger `n_estimators` .

Let's try AdaBoosting.

In [ ]:

```
# get score with ada boosting
from sklearn.ensemble import AdaBoostClassifier

ada_boost_model = AdaBoostClassifier(n_estimators=100, random_state=1).fit(X_normalized
, t)
print('Accuracy score for classification:')
print('ada_boost_model', ada_boost_model.score(X_normalized, t).mean())
```

```
Accuracy score for classification:
ada_boost_model 0.4328297264189465
```

In our case, the bagging ensemble performs best.

LWLR

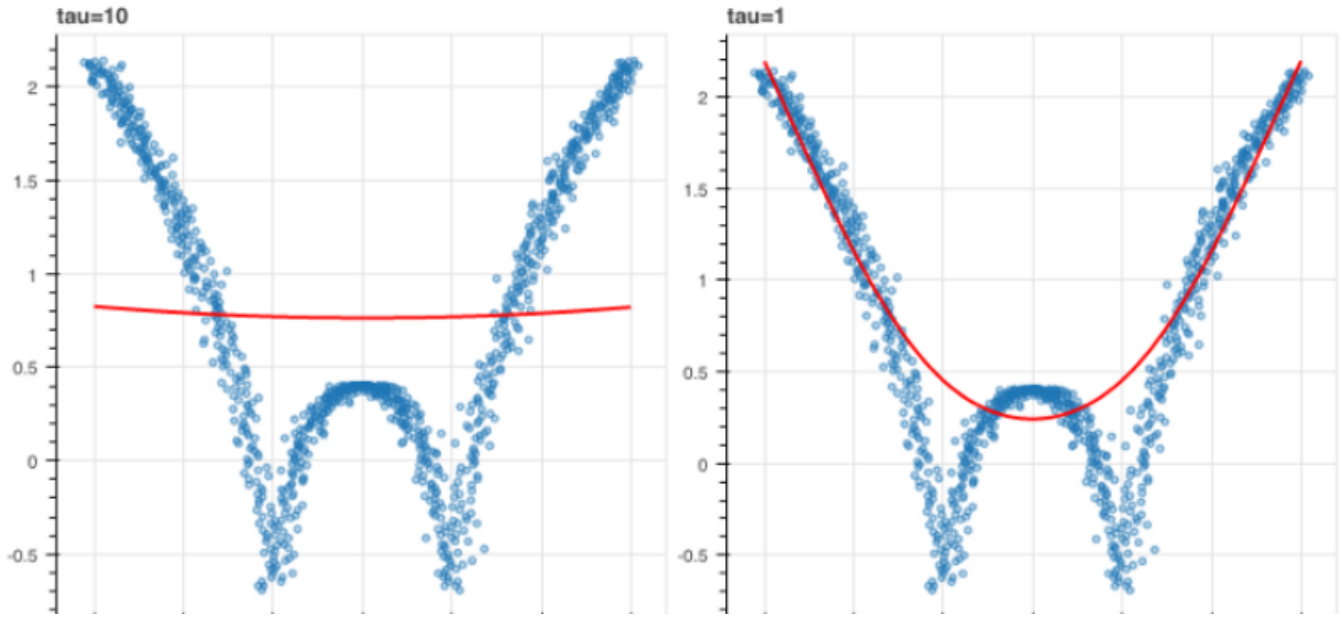
We can use a special form of Linear Regression.
This form is called *Locally Weighted Linear Regression*.
This is the equation:

$$WMSE_w = \frac{1}{2m}\sum_i \beta_i (\mathrm{wx_i} - t_i)^2$$

We can see that we added a weight (beta) for every sample.
This weight can help us emphasize the importance of the samples that are similar to our test sample.
For every test sample, we train the model from scratch and give each training sample a weight that is corresponding to the distance from the test sample.
We can use the Gaussian function as weight:

$$\beta_i = e^{\frac{-||x_i - x||^2}{2\tau^2}}$$

If tao is small, only closer samples are taking into consideration in the WMSE (big distances get small weight (beta)).
If tao is big, we get our regular MSE (big distances affect as much as small distances).



In [ ]:

```
# clone the lwlr repo from github
!git clone https://github.com/qiaochen/CourseExercises
```

```
Cloning into 'CourseExercises'...
remote: Enumerating objects: 24, done.
remote: Counting objects: 100% (24/24), done.
remote: Compressing objects: 100% (24/24), done.
remote: Total 24 (delta 7), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (24/24), done.
```

Let's check the LWLR model with k=1 .

In [ ]:

```
# get cv score for lwlr with k=1
from CourseExercises.lwlr import LWLR

arr_X_normalized = np.asarray(X_normalized)
print('R2 score for regression:')
print('lwlr', cross_val_score(LWLR(k=1), arr_X_normalized, t, cv=5, scoring='r2').mean
())
print()
print('Accuracy score for regression:')
print('lwlr', cross_val_score(LWLR(k=1), arr_X_normalized, t, cv=5, scoring=make_scorer
(accuracy_for_ordinal)).mean())
```

```
R2 score for regression:
lwlr 0.20461186304463705

Accuracy score for regression:
lwlr 0.5271624523149402
```

Let's get the best k for the LWLR model.
It may take some time (we are building the model from scratch for every test sample), so let's check how long it takes.

In [ ]:

```
%%time
# get best k for lwlr (show the calculation of this cell)
hyper_parameters = {'k': list(range(1, 10))}

gs_lw_model = GridSearchCV(LWLR(k=1), hyper_parameters, scoring='r2').fit(arr_X_normali
zed, t)
print('R2 score for regression:')
print('gs_lw_model', gs_lw_model.best_score_)
print('best params', gs_lw_model.best_params_)
print()
gs_lw_model = GridSearchCV(LWLR(k=1), hyper_parameters, scoring=make_scorer(accuracy_fo
r_ordinal)).fit(arr_X_normalized, t)
print('Accuracy score for regression:')
print('gs_lw_model', gs_lw_model.best_score_)
print('best params', gs_lw_model.best_params_)
```

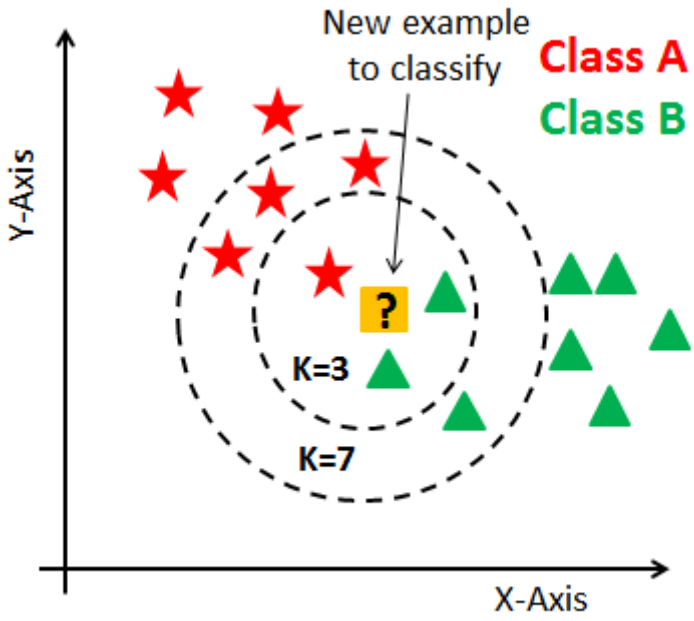
```
R2 score for regression:
gs_lw_model 0.32081021947494615
best params {'k': 3}

Accuracy score for regression:
gs_lw_model 0.5306318400700423
best params {'k': 3}
CPU times: user 6min 32s, sys: 4min 55s, total: 11min 27s
Wall time: 5min 55s
```

KNN

We can take the idea we saw in LWLR to the extreme level and create a model that predicts only based on the closest training samples to a test sample.
This model is called *K Nearest Neighbors*.
We can choose the k and the model will calculate the prediction for each test sample, based on the closest k training samples to the test sample.
We need to determine what is the meaning of *close*.
We need to use some sort of distance function to determine the closeness of each training sample to the test samples.
We can use the Euclidean distance (https://en.wikipedia.org/wiki/Euclidean_distance):

$$d(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_i - q_i)^2 + \cdots + (p_n - q_n)^2}.$$

We can use KNN in classification tasks or regression tasks.
When we use it in a regression task, we can take the mean of the target of all the neighbors.
When we use it in a classification task, we can take the mean of the probability of all the neighbors, or we can use voting, and choose the label that got the most votes from the neighbors.
We can choose to give all the neighbors that are in the decision group, the same weight in the vote, or we can give the closest neighbors higher weight than the farthest.



Let's use Scikit-learn KNeighborsClassifier (https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html).

In [ ]:

```
# run KNN on the dataset and find best K by accuracy
from sklearn.neighbors import KNeighborsClassifier

hyper_parameters = {'n_neighbors': list(range(1, 20))}

gs_neigh_model = GridSearchCV(KNeighborsClassifier(n_neighbors=5), hyper_parameters).fi
t(arr_X_normalized, t)
print('Accuracy score for classification:')
print('gs_neigh_model', gs_neigh_model.best_score_)
print('best params', gs_neigh_model.best_params_)
```

```
Accuracy score for classification:
gs_neigh_model 0.5044916720518647
best params {'n_neighbors': 16}
```

We can see that the best `n_neighbors` is `16`.
let's try the Scikit-learn KNeighborsRegressor (https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html).

In [ ]:

```
# run KNN on the dataset and find best K by R2 and accuracy
from sklearn.neighbors import KNeighborsRegressor
hyper_parameters = {'n_neighbors': list(range(1, 20))}

gs_neigh_model = GridSearchCV(KNeighborsRegressor(n_neighbors=5, weights='distance'), h
yper_parameters).fit(arr_X_normalized, t)
print('R2 score for regression:')
print('gs_neigh_model', gs_neigh_model.best_score_)
print('best params', gs_neigh_model.best_params_)
print()
gs_neigh_model = GridSearchCV(KNeighborsRegressor(n_neighbors=5, weights='distance'), h
yper_parameters, scoring=make_scorer(accuracy_for_ordinal)).fit(arr_X_normalized, t)
print('Accuracy score for regression:')
print('gs_neigh_model', gs_neigh_model.best_score_)
print('best params', gs_neigh_model.best_params_)
```

```
R2 score for regression:
gs_neigh_model 0.27277425388148835
best params {'n_neighbors': 19}

Accuracy score for regression:
gs_neigh_model 0.5259358779262471
best params {'n_neighbors': 16}
```

The KNeighborsRegressor did better than the KNeighborsClassifier on this dataset.
It might be due to the additional information it has on the order of the classes.

We also used `weights='distance'` instead of the default `weights='uniform'`, which makes this KNN model act more similarly to the LWLR model we have seen previously.

More Information

Guide on how to upload python packeges to PyPi:
How to upload your python package to PyPi (https://medium.com/@joel.barmettler/how-to-upload-your-python-package-to-pypi-65edc5fe9c56)

Explanation of few EDA libraries in python:
4 Libraries that can perform EDA in one line of python code (https://towardsdatascience.com/4-libraries-that-can-perform-eda-in-one-line-of-python-code-b13938a06ae)

Explanation of Vinho Verde wines:
Portuguese Vinho Verde wine: everything you need to know (https://www.olivemagazine.com/drink/portuguese-vinho-verde-wine-guide/)

Kaggle notebook on the white wine dataset:
KNN for classifying wine quality (https://www.kaggle.com/raultrevino/knn-for-classify-wine-quality)

Kaggle notebook on the white wine dataset:
Predicting White Wine Quality (https://www.kaggle.com/indra90/predicting-white-wine-quality)

Explanation on Lowess smoothing:
Lowess Smoothing: Overview (https://www.statisticshowto.com/lowess-smoothing/)

Explanation on regularization:
REGULARIZATION: An important concept in Machine Learning (https://towardsdatascience.com/regularization-an-important-concept-in-machine-learning-5891628907ea)

Article about the geometry of Ridge and Lasso regularizations:
Regularization and Geometry (https://towardsdatascience.com/regularization-and-geometry-c69a2365de19)

Explanation on Ridge, Lasso, and Elastic Net regularizations:
An Introduction to Ridge, Lasso, and Elastic Net Regression (https://hackernoon.com/an-introduction-to-ridge-lasso-and-elastic-net-regression-cca60b4b934f)

Explanation of the differences between Ridge and Lasso regularizations:
Intuitive and Visual Explanation on the differences between L1 and L2 regularization (https://www.linkedin.com/pulse/intuitive-visual-explanation-differences-between-l1-l2-xiaoli-chen/)

Guide on how to use regular classifier as an ordinal classifier:
Simple Trick to Train an Ordinal Regression with any Classifier (https://towardsdatascience.com/simple-trick-to-train-an-ordinal-regression-with-any-classifier-6911183d2a3c)

A list of predefined scores for Sciking-learn:
Common cases: predefined scores (https://scikit-learn.org/stable/modules/model_evaluation.html#common-cases-predefined-values)

Explanation on LWLR and a code sample:
Linear Regression: How to overcome underfitting with Locally Weighted Linear Regression (LWLR) (https://itnext.io/linear-regression-how-to-overcome-underfitting-with-locally-weight-linear-regression-lwlr-e867f0cde4a4)

Example of LWLR in python:
Locally Weighted Linear Regression in Python (https://www.codespeedy.com/locally-weighted-linear-regression-in-python/)

Short Explanation on LWLR:
ML | Locally weighted Linear Regression (https://www.geeksforgeeks.org/ml-locally-weighted-linear-regression/)