UNIVERSITÀ DEGLI STUDI DI CATANIA
DIPARTIMENTO DI ECONOMIA E IMPRESA
CORSO DI LAUREA IN DATA SCIENCE FOR MANAGEMENT

REPORT ON "RED WINE QUALITY" DATASET

Casella Bruno
1000014143

Neural Computing

ACADEMIC YEAR 2020 – 2021

INTRODUCTION

Once viewed as a luxury good, nowadays wine is increasingly enjoyed by a wider range of consumers. Wine is an alcoholic drink typically made from fermented grapes. The earliest known traces of wine are from China (7000 BC) and Georgia (6000 BC). Throughout history, wine has been consumed for its intoxicating effects.

To support its growth, the wine industry is investing in new technologies for both wine making and selling processes. Wine certification and quality assessment are key elements within this context. Quality evaluation is often part of the certification process and can be used to improve wine making (by identifying the most influential factors) and to stratify wines such as premium brands (useful for setting prices). When you know what influences and signifies wine quality, you will be in a better position to make good purchases. You will also begin to recognize your preferences and how your favorite wines can change with each harvest. Your appreciation for wines will deepen once you are familiar with wine quality levels and how wines vary in taste from region to region.

In this report, I present a case study for modeling wine quality based on a dataset related to the red variant of the Portuguese “Vinho Verde” wine.
The report is organized as follows: the first chapter describes the dataset; in the second chapter, I discuss Exploratory Data Analysis (EDA) including data processing, missing values, correlation plots, outliers and so on; the third chapter presents Deep Learning experiments and conclusions.

"RED WINE QUALITY" DATASET

The dataset used in this report is the "Red Wine Quality" available at the following link:
https://archive.ics.uci.edu/ml/datasets/wine+quality
Two datasets are included, related to red and white variants of the Portuguese "Vinho Verde" wine, from north Portugal, but I will use only the Red version. Due to privacy and logistic issues, only physicochemical (inputs) and sensory (output) variables are available (e.g. there is no data about grape types, wine brand, wine selling price, etc.).
The Red Wine dataset contains 1599 observations and 12 variables.
Content of the dataset:

- **Input** Variables (based on physicochemical tests):
    1. Fixed acidity, the fixed acids involved with wine that do not evaporate readily
    2. Volatile acidity, the amount of acetic acids in wine, which at too high of levels can lead to an unpleasant, vinegar taste
    3. Citric acid, which is found in small quantities but can add "freshness" and flavor to wines
    4. Residual sugar, the amount of sugar remaining after fermentation stops
    5. Chlorides, the amount of salt in the wine
    6. Free sulfur dioxide, the free form of SO2 exists in equilibrium between molecular SO2 (as a dissolved gas) and bisulfite ion, which prevents microbial growth and the oxidation of wine
    7. Total sulfur dioxide, amount of free and bound forms of SO2
    8. Density, 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 o (very acidic) to 14 (very basic)
    10. Sulphates, a wine additive which can contribute to sulfur dioxide gas SO2 levels, which acts as an antimicrobial and antioxidant
    11. Alcohol, the percent alcohol content of the wine
- **Output** Variable:
    12. Quality, the sensory data, median of at least 3 evaluations graded by wine experts between 0 (very bad) and 10 (excellent). However, in the dataset there are only values ranging from 3 to 8.

Let's see some data points present in the data.

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8	5
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8	5
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8	6
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5

This dataset can be viewed as a classification or regression task.

EXPLORATORY DATA ANALYSIS

First of all, I look at the descriptive statistics.

	mean	std	min	25%	50%	75%	max
fixed acidity	8.319637	1.741096	4.60000	7.1000	7.90000	9.200000	15.90000
volatile acidity	0.527821	0.179060	0.12000	0.3900	0.52000	0.640000	1.58000
citric acid	0.270976	0.194801	0.00000	0.0900	0.26000	0.420000	1.00000
residual sugar	2.538806	1.409928	0.90000	1.9000	2.20000	2.600000	15.50000
chlorides	0.087467	0.047065	0.01200	0.0700	0.07900	0.090000	0.61100
free sulfur dioxide	15.874922	10.460157	1.00000	7.0000	14.00000	21.000000	72.00000
total sulfur dioxide	46.467792	32.895324	6.00000	22.0000	38.00000	62.000000	289.00000
density	0.996747	0.001887	0.99007	0.9956	0.99675	0.997835	1.00369
pH	3.311113	0.154386	2.74000	3.2100	3.31000	3.400000	4.01000
sulphates	0.658149	0.169507	0.33000	0.5500	0.62000	0.730000	2.00000
alcohol	10.422983	1.065668	8.40000	9.5000	10.20000	11.100000	14.90000
quality	5.636023	0.807569	3.00000	5.0000	6.00000	6.000000	8.00000

We find that the mean of quality score is about 5.64 and we can see using the interquartile range (IQR), calculated as the difference between the 75^{th} and 25^{th} percentiles, that most variables have outliers. Moreover, for example, we can see that there is a large difference between 75^{th} percentile and max values of Residual Sugar, Free Sulfur Dioxide and Total Sulfur Dioxide. Thus observation suggests that there are extreme values or outliers in the dataset. However, I will identify outliers later using visualization techniques.
From this picture we can see also that mean value is higher than the median value of each column.

Now let's check if there is any missing value in the data. Missing values are always a problem while analyzing the data and also building the models using that data.

```
fixed acidity           0
volatile acidity        0
citric acid             0
residual sugar          0
chlorides               0
free sulfur dioxide     0
total sulfur dioxide    0
density                 0
pH                      0
sulphates               0
alcohol                 0
quality                 0
dtype: int64
```

The result shows that we have the data intact 100% and there is no missing data in our dataset.

I ask information about the data frame, including the data types of each column and memory usage of the entire data

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1599 entries, 0 to 1598
Data columns (total 12 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   fixed acidity         1599 non-null   float64
 1   volatile acidity      1599 non-null   float64
 2   citric acid           1599 non-null   float64
 3   residual sugar        1599 non-null   float64
 4   chlorides             1599 non-null   float64
 5   free sulfur dioxide   1599 non-null   float64
 6   total sulfur dioxide  1599 non-null   float64
 7   density               1599 non-null   float64
 8   pH                    1599 non-null   float64
 9   sulphates             1599 non-null   float64
 10  alcohol               1599 non-null   float64
 11  quality               1599 non-null   int64  
dtypes: float64(11), int64(1)
memory usage: 150.0 KB
```

Data has only float and integer values.

The below-shown picture will print the number of unique values in each of the features.

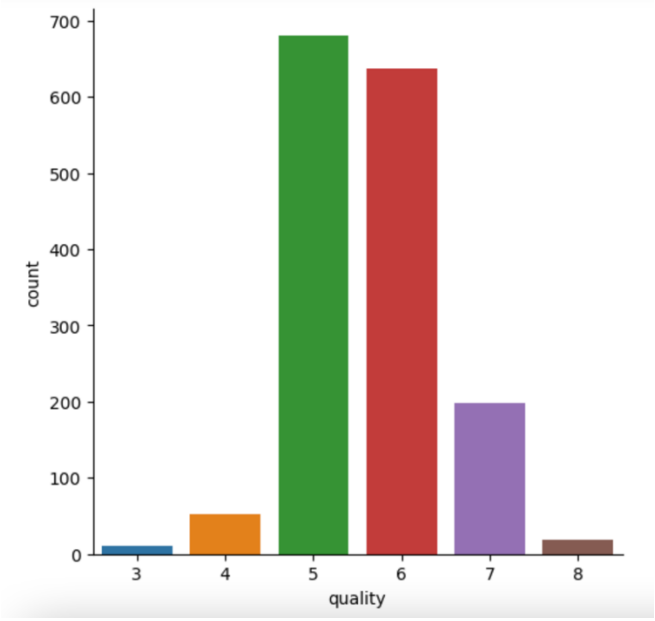
```
Number of unique values of fixed acidity: 96
Number of unique values of volatile acidity: 143
Number of unique values of citric acid: 80
Number of unique values of residual sugar: 91
Number of unique values of chlorides: 153
Number of unique values of free sulfur dioxide: 60
Number of unique values of total sulfur dioxide: 144
Number of unique values of density: 436
Number of unique values of pH: 89
Number of unique values of sulphates: 96
Number of unique values of alcohol: 65
Number of unique values of quality: 6
```

We can see that the feature with the maximum unique value is density, and the feature that has the minimum unique value is quality, so let's see these unique values of quality.

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

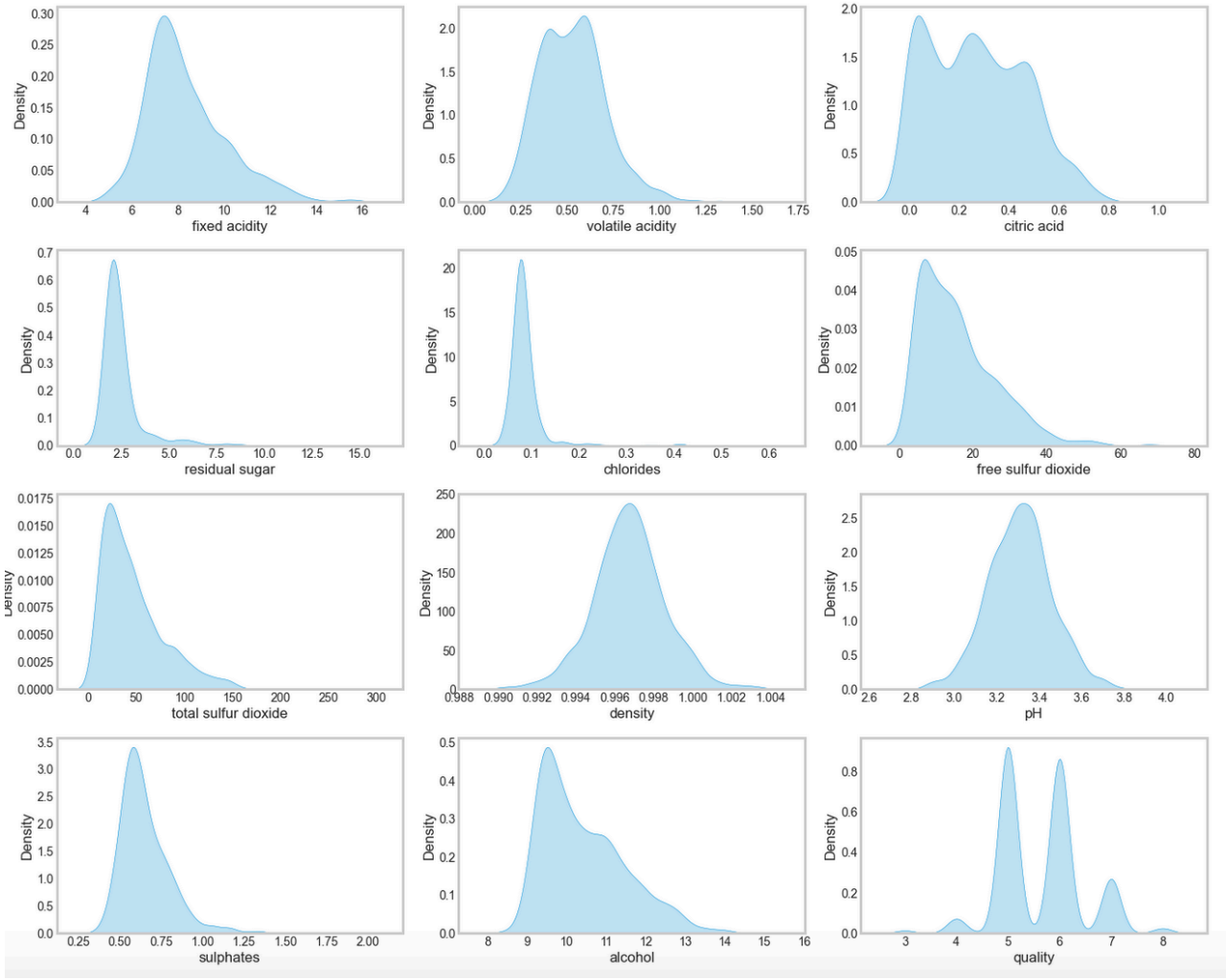
This tells us vote count of each quality score in descending order. Quality has most values concentrated in the categories 5,6 and 7. Only a few observations made for the categories 3 and 8.

```
5    681
6    638
7    199
4     53
8     18
3     10
Name: quality, dtype: int64
```



From these insights we can identify class imbalance which can help you understand and fix classification errors at a later stage. Here the classes are ordered and not balanced (there are much more normal wine than excellent or poor ones).

Now, I will show the PDF plots visualizing the spread of the data.

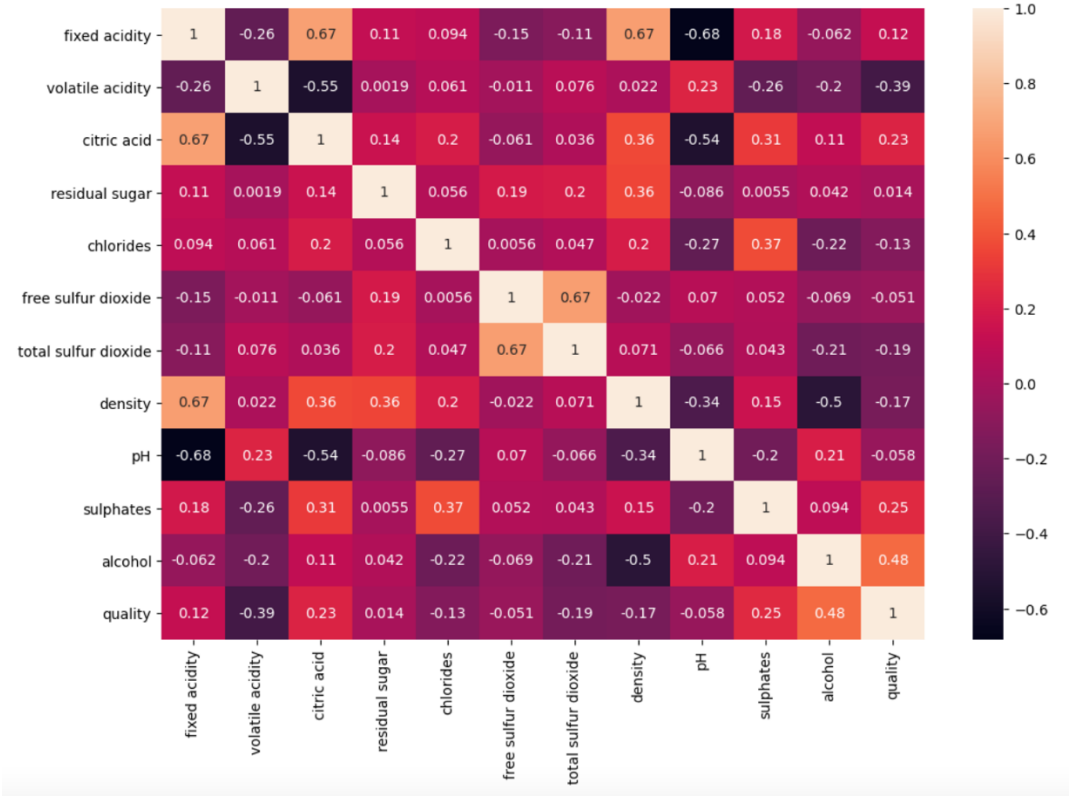


It can be seen that most wines' pH levels are between 3 and 3.6, and that chlorides is most prevalent at level 0.1. Some distributions are bimodal or trimodal, so they possibly have 2 or 3 classes, like volatile acidity, citric acid and alcohol.
Moreover, we can see that pH and density are approximately normally distributed, and the remaining features are positively skewed. Skewness is a measure of the asymmetry of the probability distribution of a real-valued random variable about its mean. Finally, all the features except citric acid are leptokurtic.

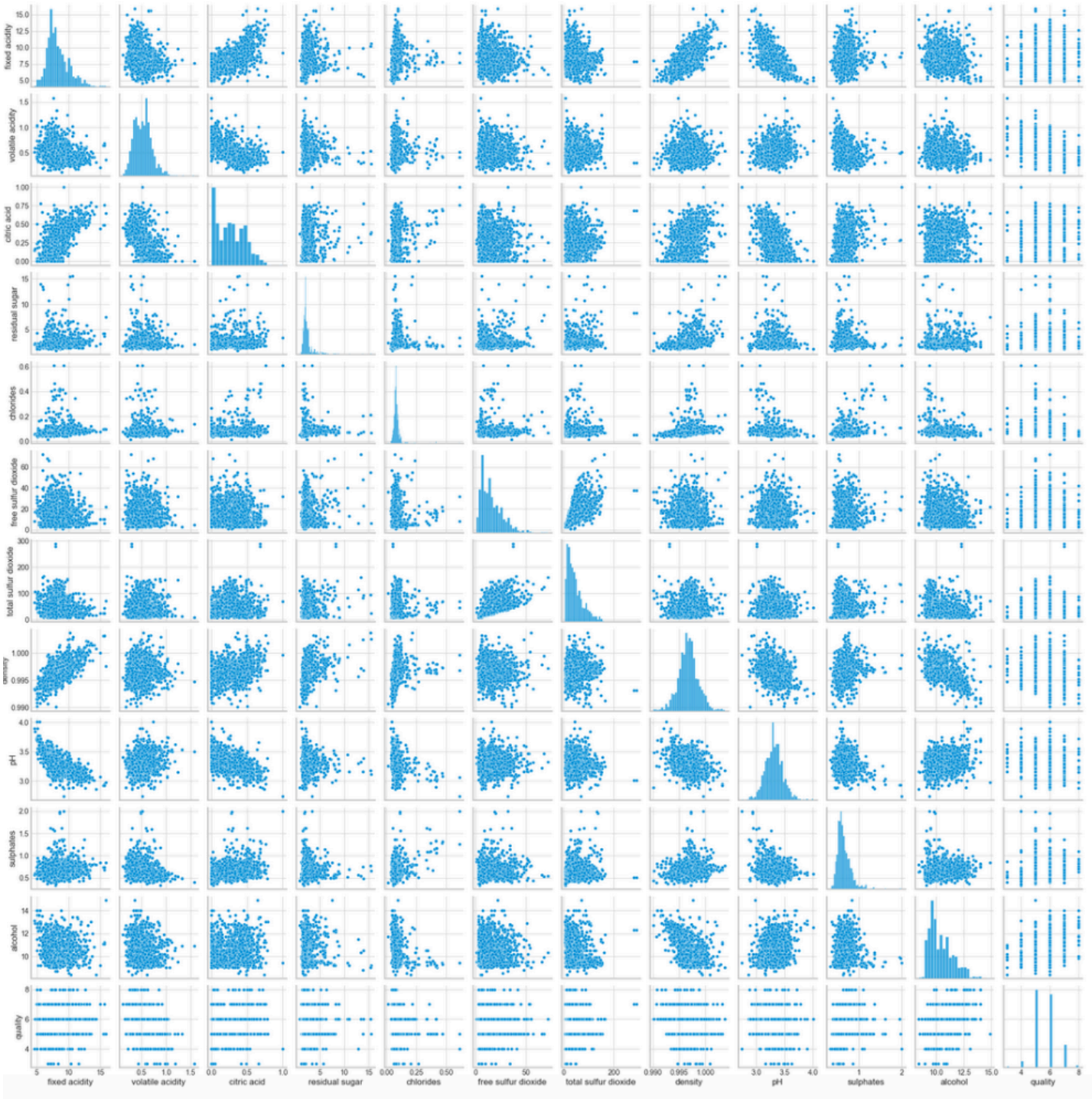
```
Skewness
 fixed acidity           0.982751
volatile acidity         0.671593
citric acid              0.318337
residual sugar           4.540655
chlorides                5.680347
free sulfur dioxide      1.250567
total sulfur dioxide     1.515531
density                  0.071288
pH                       0.193683
sulphates                2.428672
alcohol                  0.860829
quality                  0.217802
dtype: float64
```

```
Kurtosis
  fixed acidity          1.132143
volatile acidity         1.225542
citric acid             -0.788998
residual sugar          28.617595
chlorides               41.715787
free sulfur dioxide      2.023562
total sulfur dioxide     3.809824
density                  0.934079
pH                       0.806943
sulphates               11.720251
alcohol                  0.200029
quality                  0.296708
dtype: float64
```

To see which variables are likely to affect the quality of wine the most, I ran a correlation analysis.

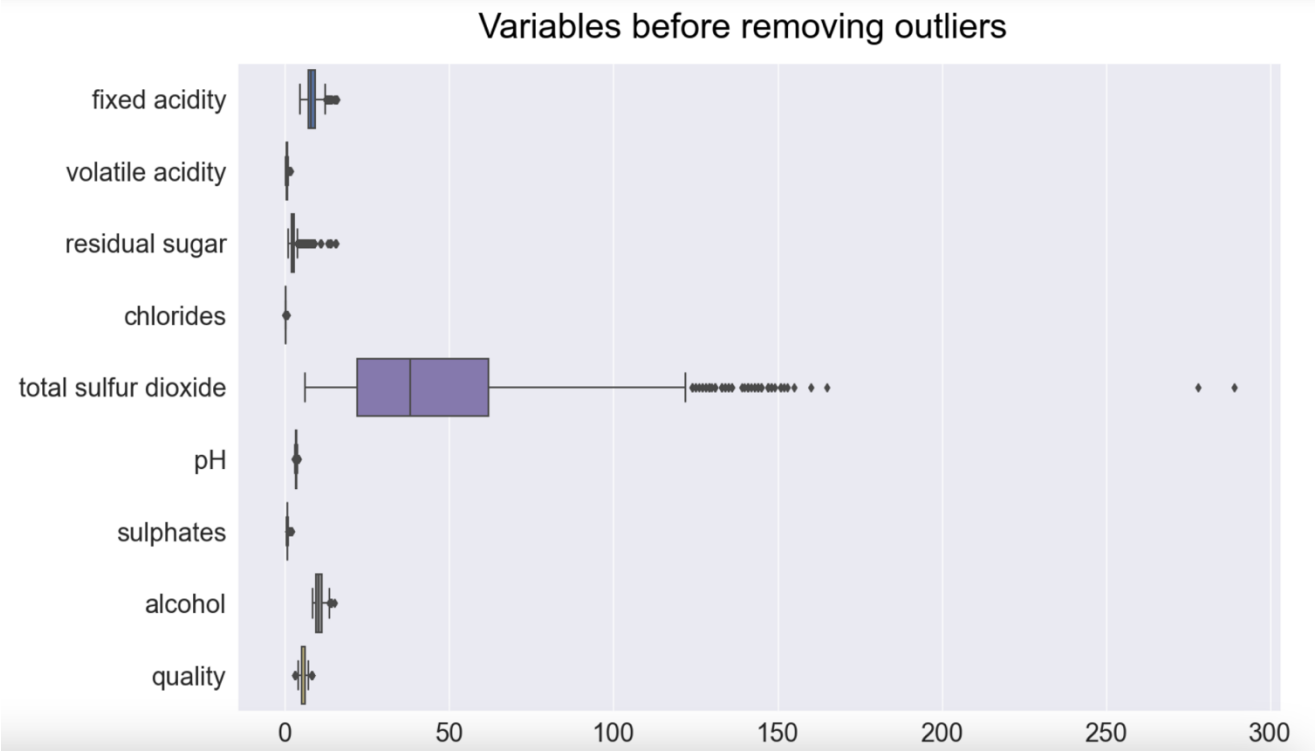


In order of highest correlation: alcohol, volatile acidity, sulphates, citric acid, total sulfur dioxide, density, chlorides, fixed acidity, pH, free sulfur dioxide, residual sugar.
And now I plot pairwise relationships in the dataset.

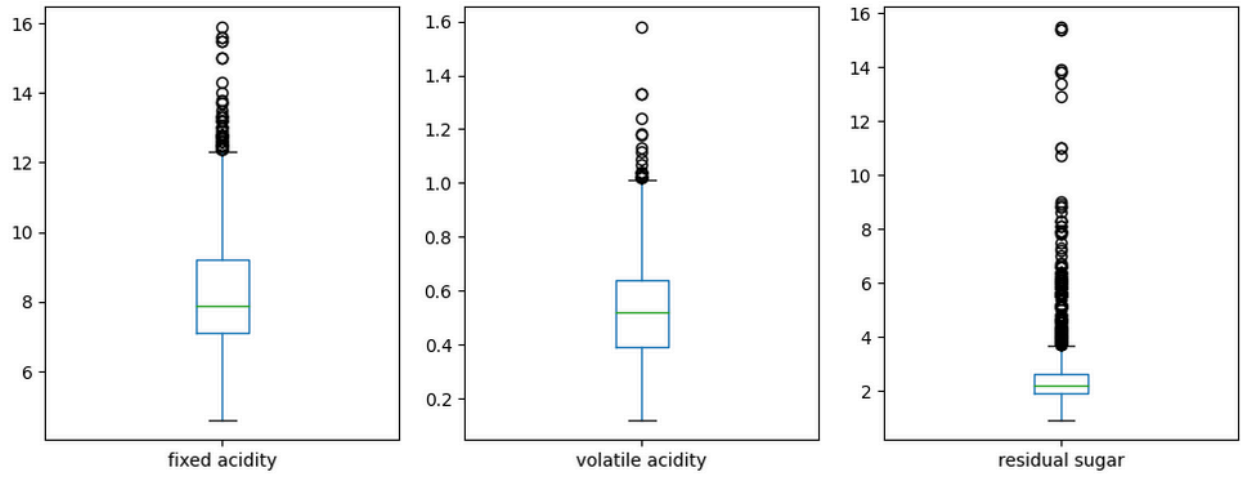


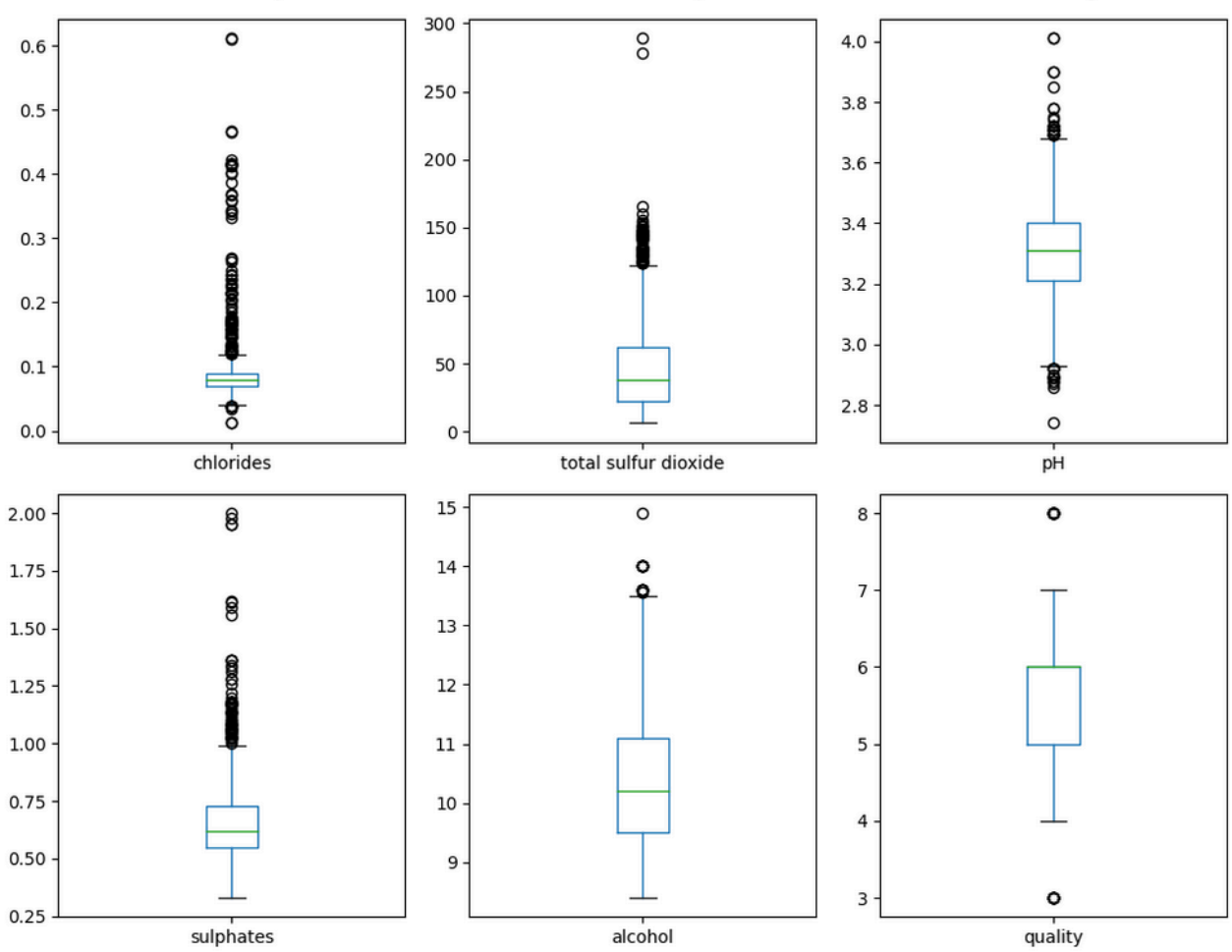
From the above heatmap and correlation plot on red wine, we have the following inference: 1. It looks like we have pH and fixed acidity with inverse relationship between them, and also for citric acid and volatile acidity. 2. There is a strong positive relationship between total sulfur dioxide and free sulfur dioxide.
Moreover, fixed acidity, citric acidity and density are highly correlated, likewise free sulfur dioxide and total sulfur dioxide. So, I can drop the

variables citric acidity, density and free sulfur dioxide to reduce multicollinearity.
Going back to the previous argument of outliers: in statistics, an outlier is a data point that differs significantly from other observations. An outlier may be due to variability in the measurement or it may indicate experimental error. And because the quality of the input of our model decides the quality of the output, detecting and treating/removing outliers is an important step in the data exploration stage. I used boxplots that display the five-number summary of a set of data: minimum, 1st quartile, median, 3rd quartile, and maximum. In a boxplot we draw a box from the 1st quartile to the 3rd quartile. A vertical line goes through the box at the median. The whiskers go from each quartile to the minimum or maximum.



The picture above is an overview of all the boxplots before removing outliers. Now I show the boxplot for each variable in a different graph, for better visualization.





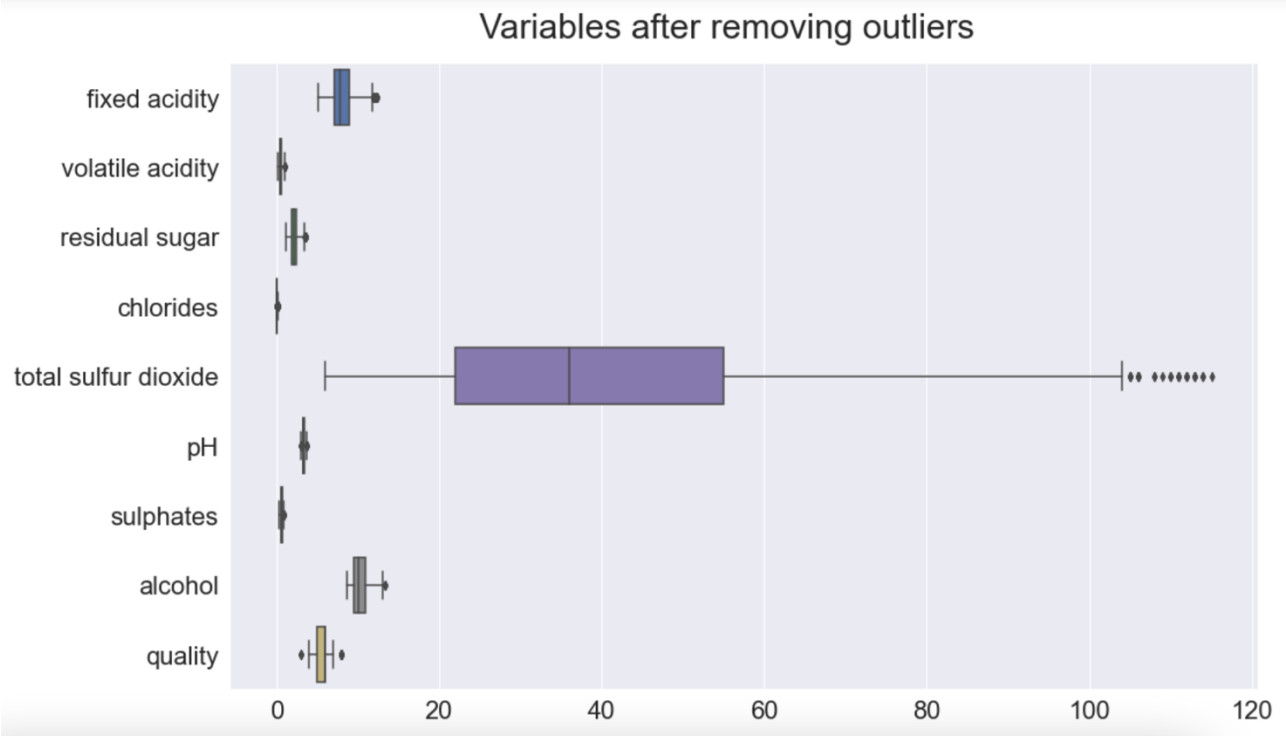
All the features have outliers. Most of them present a lot of outliers. Instead, quality has only 2 outliers, alcohol 3 and citric acid 1. Other methods to detect outliers could be using statistical tests like Peirce's Criterion, Chauvenet's Criterion, Grubb's test or Dixon's test.

Now, there are different ways to proceed in case you have outliers. If it is obvious that the outlier is due to incorrectly enter or measured data, or if you have a lot of data so your sample won't be hurt by dropping a questionable outlier, or if you can go back and recollect the questionable data point, you should drop the outlier.

Instead, if your results are critical, so even small changes will matter a lot, or if there are a lot of outliers, you should not drop the outlier.

In practice, I will remove most of the outliers using the following method: I calculate the Inter Quartile Range that is IQR=Q3-Q1 and then, any value below Q1-1.5*IQR and Q3+1.5*IQR is an outlier.

It can be seen from the picture below that we have successfully removed most of the outliers for all variables.



A Box Cox transformation is a way to transform non-normal data distribution into a normal shape. Why does it matter?
Because if you are performing a regression or any statistical modeling, this asymmetrical behavior may lead to a bias in the model. If a factor has a significant effect on the average, because the variability is much larger, many factors will seem to have a stronger effect when the mean is larger. This is not due, however, to a true factor effect but rather to an increased amount of variability that affects all factor effect estimates when the mean gets larger. This will probably generate spurious interactions due to a non-constant variation, resulting in a very complex model with many spurious and unrealistic interactions.
Moreover, normality is an important assumption for many statistical techniques. One solution to this is to transform your data into normality using a Box-Cox transformation.
So, after removing outliers, skewness of variables changed. Here the new values:

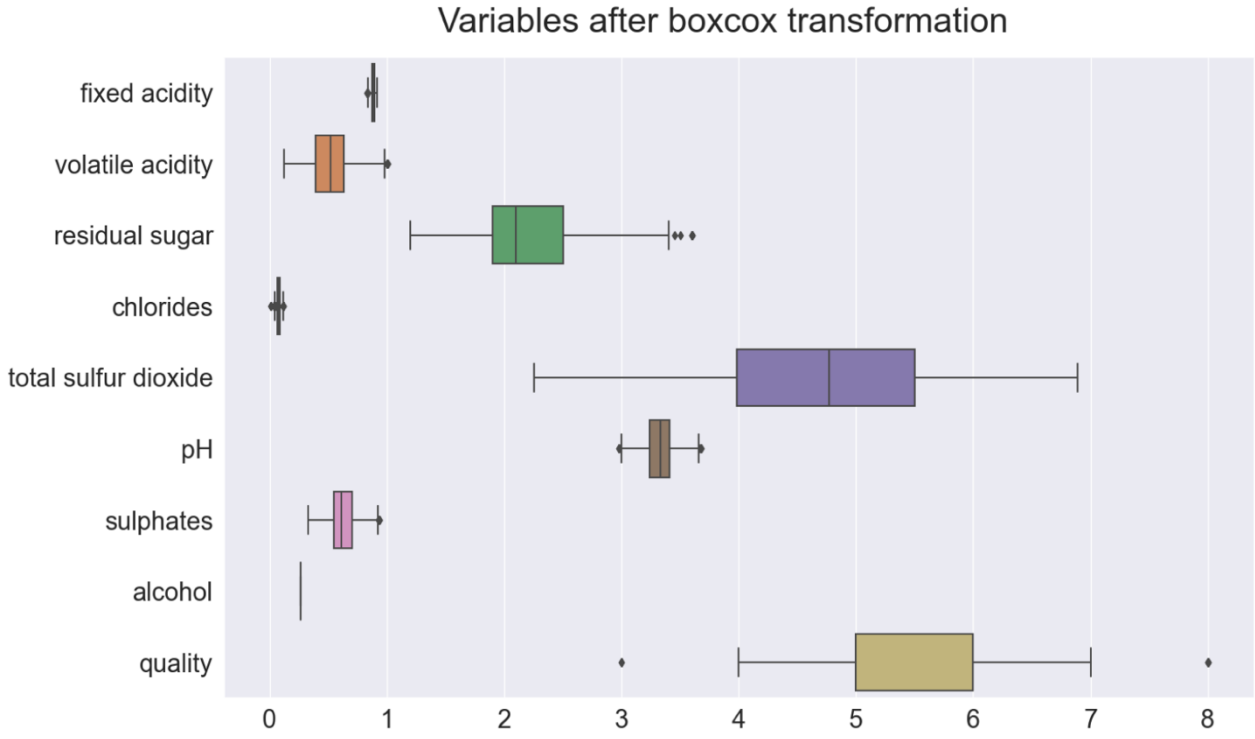
```
Skewness
 fixed acidity           0.748003
volatile acidity         0.282093
residual sugar           0.625386
chlorides               -0.064911
total sulfur dioxide     0.892078
pH                       0.079758
sulphates                0.513739
alcohol                  0.744075
quality                  0.321122
dtype: float64
```

It can be seen that there are some variables highly skewed, on which we can apply Box Cox transformation:

```
There are 3 highest skewed numerical features to box cox transform
```

	Skew	New Skew
total sulfur dioxide	0.890950	-0.022689
fixed acidity	0.747056	0.018871
alcohol	0.743133	0.117989

And see how change the boxplot graph:



At the end of these data processing steps, the dataset has 1186 rows and 6 columns.

DATA MODELING

My input features are Fixed Acidity, Volatile Acidity, Residual Sugar, Chlorides, Total Sulfur Dioxide, pH, Sulphates and Alcohol. My target feature is Quality.
First of all, I converted my dataframe into NumPy arrays. In the definition of targets_array there is a -3 of the value of the outputs. This is due to the fact that indexing starts at 0. In the sense that, the classes of quality go from 3 to 8, but if we use these labels with the loss function, it will be matched with total 0 labels (class0 to class8) as maximum class labels is 8. So, I needed to transform 3 to 8 in 0 to 5.

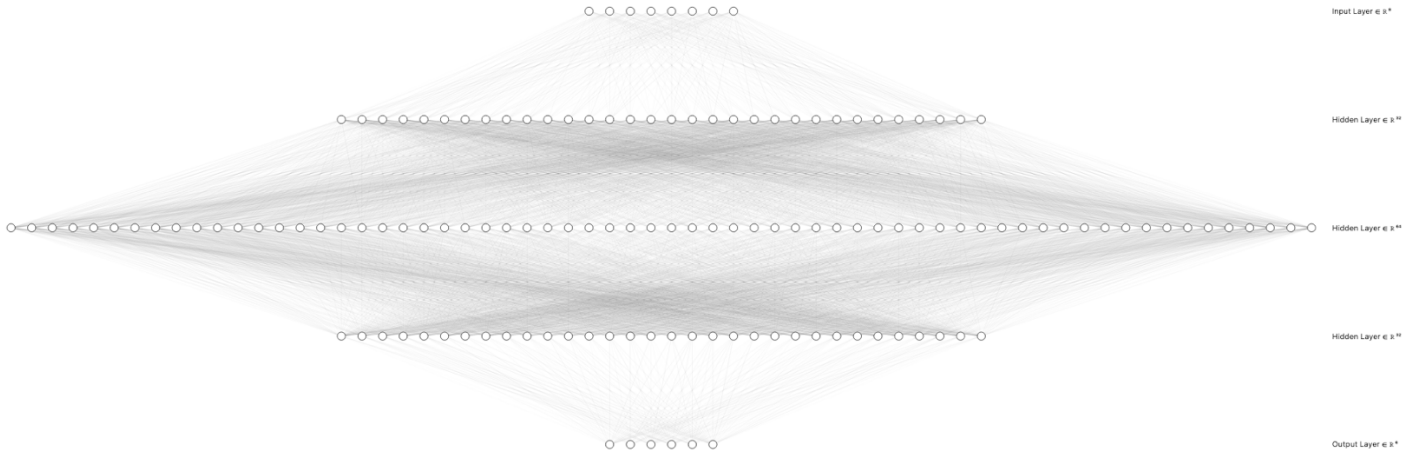
```
inputs_array = wine_copy[input_columns].to_numpy()
targets_array = (wine_copy[output_columns]-3).to_numpy().squeeze().astype(np.long)
```

Then I flattened the targets_array because the loss function will need a 1D vector.
After that, I converted these arrays into Pytorch Tensors; inputs as float and outputs as long.
Next, I converted into a TensorDataset my inputs and outputs, and now that I have a single dataset, I could split in training set and validation set. I used the 20% of the entire dataset as validation, so at the end I had 949 examples for the training set and 237 for the validation set. To ensure that we always create the same validation set, we also set a seed for the random number generator.
In the era of Big Data, it is common to divide into training, validation and test sets, with percentages like 98-1-1%. But in this case, the dataset was small, so I decided to split only in 2 parts with 80-20%. Moreover, the dataset is imbalanced, so there was the possibility that all the minor classes would go in the validation set, giving us bad results at the end of the training.
Next, I initialized batch size to 64. The batch size is a hyperparameter that defines the number of samples to work through before updating the internal model parameters. During my work I tried with different batch sizes value like 32 or 128, but at the end I used 64. Then I used the DataLoader class, that represent a Python iterable over a dataset. It combines a dataset and a sampler, and yields data as a batch for every epoch. The parameter shuffle = True shuffle the data while training, so that inputs and outputs are collected in a rearranged or intermixed manner from the dataset for each batch. This randomization helps and generalize and speed up the training process.
Finally, I defined a DeviceDataLoader class to wrap our existing data loaders and move batches of data to the selected device. Tensors moved to the GPU have a device property which includes that word cuda.
The loss measure used is the Cross-Entropy Loss. Cross-entropy is a commonly used loss function for classification tasks. The prediction is a probability vector, meaning it represents predicted probabilities of all classes, summing up to 1. This method combines the Softmax activation function principle and the negative log likelihood loss. In a neural network, you typically achieve this prediction by having the last layer activated by a softmax function, but anything goes — it just must be a probability vector. Loss is a measure of performance of a model. The lower, the better. When learning, the model aims to get the lowest loss possible. The target represents probabilities for all classes — 3 to 8 quality. The target for multi-class

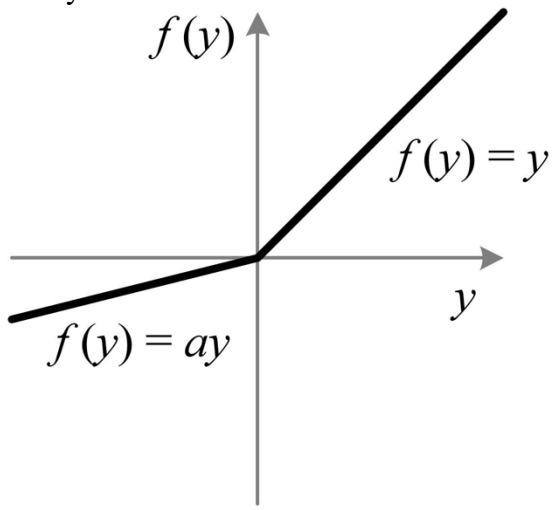
classification is a one-hot vector, meaning it has 1 on a single position and 0's everywhere else. For example, for the 3rd class, we want the probability to be 1. For other classes, we want it to be 0.
The loss can be described as:

$$\text{loss}(x, class) = -\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right) = -x[class] + \log\left(\sum_j \exp(x[j])\right)$$

Now, in the picture below you can see the architecture of my Neural Network (in vertical, from top to bottom).



Information move from the initial 8 input nodes, that are the features, to the last layer with 6 output nodes, that are the predicted classes of quality. In between I put 3 hidden layers, the first one with 32 neurons, the second one with 64 and the last one with 32. I chose this architecture after doing several trials. In each hidden layer I used the non-linear activation function PReLU, that is similar to the ReLU, but with the advantage that his derivative is different from zero when the variable is negative. We need activation functions otherwise each layer will be only the linear function of its input features, and a linear hidden layer is more or less useless. In the picture below you can see the PReLU function, with a law of f(y) = max(ay, y)



All the elements of this NN are contained in the class WineQuality, that contains several methods.
Then, in 2 hidden layers I used Dropout, that is a regularization technique to prevent overfitting. With Dropout I set a probability of eliminating (zeroing out) a node in that layer of NN.
Everything which contains weights which you want to be trained during the training process should be defined in your __init__ method. Then, I use these things in the forward method for doing forward propagation. The other methods are used to generate predictions and to calculate loss of training and validation steps, and to combine losses and accuracies and print them.
So, the model is this:

```
WineQuality(
  (net): Sequential(
    (0): Linear(in_features=8, out_features=32, bias=True)
    (1): PReLU(num_parameters=1)
    (2): Linear(in_features=32, out_features=64, bias=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): PReLU(num_parameters=1)
    (5): Linear(in_features=64, out_features=32, bias=True)
    (6): Dropout(p=0.5, inplace=False)
    (7): PReLU(num_parameters=1)
    (8): Linear(in_features=32, out_features=6, bias=True)
  )
)
```

Now I'll define an evaluate function, which will perform the validation phase, a fit function which will perform the entire training process, and a get_lr function that will be used inside of the fit function. Before training the model, let's see the fit function. It contains different techniques to do hyperparameter optimization, to prevent overfitting, and to avoid exploding gradients.

Learning rate scheduling: Instead of using a fixed learning rate, I will use a learning rate scheduler, which will change the learning rate after every batch of training. There are many strategies for varying the learning rate during training, and the one I'll use is called the "One Cycle Learning Rate Policy", which involves starting with a low learning rate, gradually increasing it batch-by-batch to a high learning rate for about 30% of epochs, then gradually decreasing it to a very low value for the remaining epochs.

Weight decay: I also use weight decay, which is yet another regularization technique which prevents the weights from becoming too large by adding an additional term to the loss function.

Gradient clipping: Apart from the layer weights and outputs, it is also helpful to limit the values of gradients to a small range to prevent undesirable changes in parameters due to large gradient values.

Before we begin training, let's see how the model performs on the validation set with the initial set of parameters.

```
{'val_loss': 1.7637394666671753, 'val_acc': 0.13593749701976776}
```

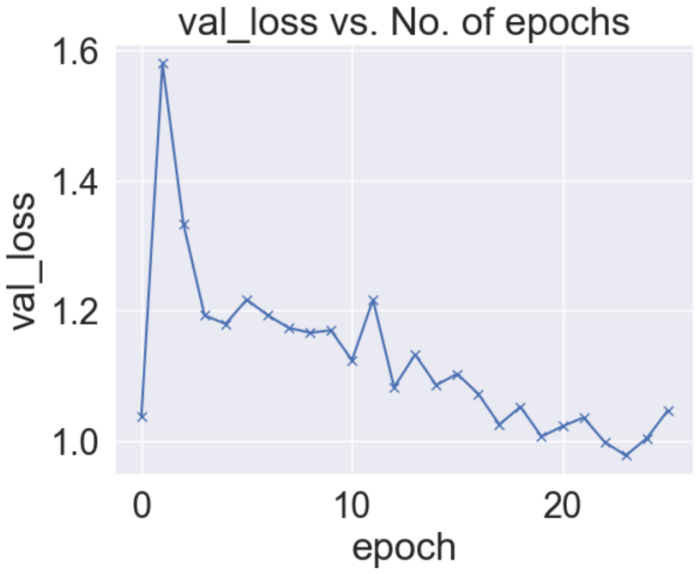
The initial accuracy is low, which is what one might expect from a randomly intialized model.

We'll use the following hyperparmeters (max learning rate, no. of epochs, grad_clip, weight decay, batch_size etc.) to train our model. Instead of SGD (stochastic gradient descent), we'll use the Adam optimizer which uses techniques like momentum and adaptive learning rates (RMSProp) for faster training.

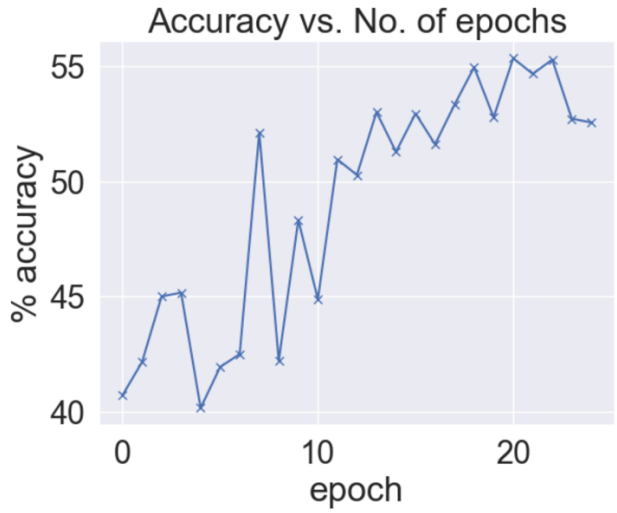
```
epochs = 25
max_lr = 0.01
grad_clip = 0.1
weight_decay = 1e-4
opt_func = torch.optim.Adam
```

```
Epoch [0], last_lr: 0.00077, train_loss: 1.6840, val_loss: 1.5795, val_acc: 0.4069
Epoch [1], last_lr: 0.00192, train_loss: 1.4170, val_loss: 1.3326, val_acc: 0.4214
Epoch [2], last_lr: 0.00364, train_loss: 1.2421, val_loss: 1.1925, val_acc: 0.4499
Epoch [3], last_lr: 0.00564, train_loss: 1.2065, val_loss: 1.1795, val_acc: 0.4516
Epoch [4], last_lr: 0.00756, train_loss: 1.1795, val_loss: 1.2169, val_acc: 0.4014
Epoch [5], last_lr: 0.00907, train_loss: 1.1738, val_loss: 1.1929, val_acc: 0.4193
Epoch [6], last_lr: 0.00989, train_loss: 1.1700, val_loss: 1.1735, val_acc: 0.4248
Epoch [7], last_lr: 0.00998, train_loss: 1.1519, val_loss: 1.1662, val_acc: 0.5213
Epoch [8], last_lr: 0.00982, train_loss: 1.1436, val_loss: 1.1696, val_acc: 0.4220
Epoch [9], last_lr: 0.00950, train_loss: 1.1385, val_loss: 1.1238, val_acc: 0.4832
Epoch [10], last_lr: 0.00905, train_loss: 1.1189, val_loss: 1.2166, val_acc: 0.4487
Epoch [11], last_lr: 0.00846, train_loss: 1.1388, val_loss: 1.0824, val_acc: 0.5095
Epoch [12], last_lr: 0.00775, train_loss: 1.0880, val_loss: 1.1325, val_acc: 0.5028
Epoch [13], last_lr: 0.00697, train_loss: 1.1029, val_loss: 1.0855, val_acc: 0.5301
Epoch [14], last_lr: 0.00611, train_loss: 1.0804, val_loss: 1.1028, val_acc: 0.5128
Epoch [15], last_lr: 0.00522, train_loss: 1.0811, val_loss: 1.0722, val_acc: 0.5295
Epoch [16], last_lr: 0.00433, train_loss: 1.0634, val_loss: 1.0251, val_acc: 0.5161
Epoch [17], last_lr: 0.00345, train_loss: 1.0608, val_loss: 1.0523, val_acc: 0.5334
Epoch [18], last_lr: 0.00263, train_loss: 1.0235, val_loss: 1.0066, val_acc: 0.5497
Epoch [19], last_lr: 0.00188, train_loss: 1.0554, val_loss: 1.0223, val_acc: 0.5279
Epoch [20], last_lr: 0.00123, train_loss: 1.0409, val_loss: 1.0359, val_acc: 0.5536
Epoch [21], last_lr: 0.00071, train_loss: 1.0392, val_loss: 0.9982, val_acc: 0.5468
Epoch [22], last_lr: 0.00032, train_loss: 1.0299, val_loss: 0.9775, val_acc: 0.5530
Epoch [23], last_lr: 0.00008, train_loss: 1.0232, val_loss: 1.0038, val_acc: 0.5273
Epoch [24], last_lr: 0.00000, train_loss: 1.0490, val_loss: 1.0467, val_acc: 0.5256
CPU times: user 2.72 s, sys: 57.1 ms, total: 2.78 s
Wall time: 1.58 s
```

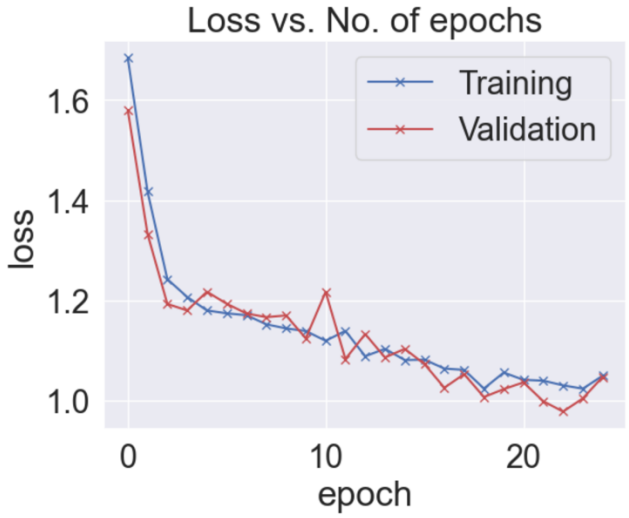
We can now plot the losses & accuracies to study how the model improves over time.



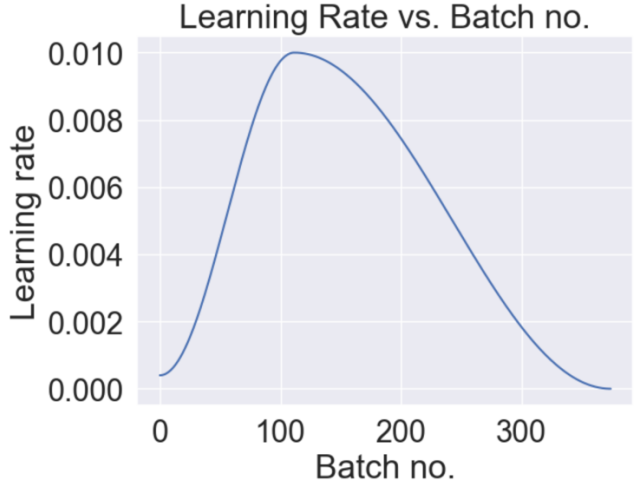
We can see that the validation loss goes down, after 20 epochs, to about 1.0. While the accuracy does continue to increase as we train for more epochs, the improvements get smaller with every epoch. Let's visualize this using a line graph.



We can also plot the training and validation losses to study the trend.



And finally, how changes the learning rate using the method OneCycleLR:



After these graphs we can understand that the model doesn't learn nicely after such operations, but it started to output the class that happens to be the most

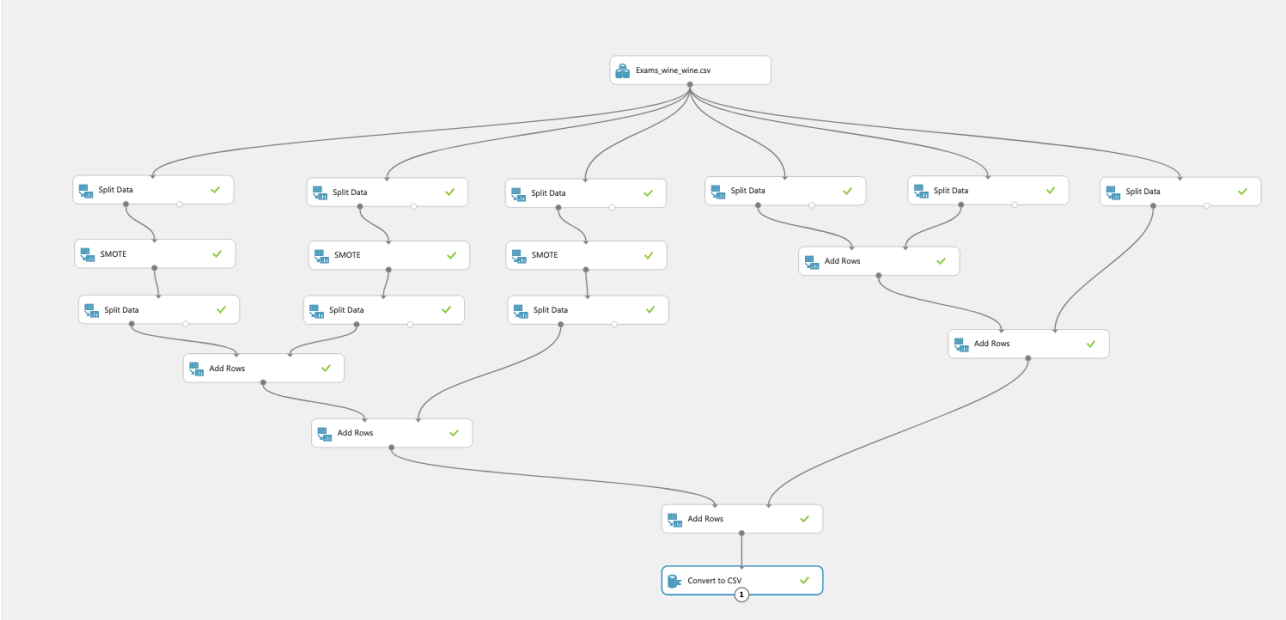
frequent one in the dataset. This is because our dataset is too much unbalanced, so the model does not learn well. Now we can do some predictions on a single example, and then on a batch size. We see again that the model will predict the most present classes in the dataset.
First, a single example:

```
Input: tensor([0.8626, 0.5700, 2.1000, 0.1150, 3.5177, 3.3800, 0.6900, 0.2610])
Target: tensor(5)
Prediction (logits): tensor([-4.1822, -1.9769,  1.1256,  1.7021,  0.9571, -2.1787])
Prediction (probabilities): tensor([0.0013, 0.0121, 0.2694, 0.4796, 0.2277, 0.0099])
Predicted class (-3): tensor([3])
Predicted class: 6
```

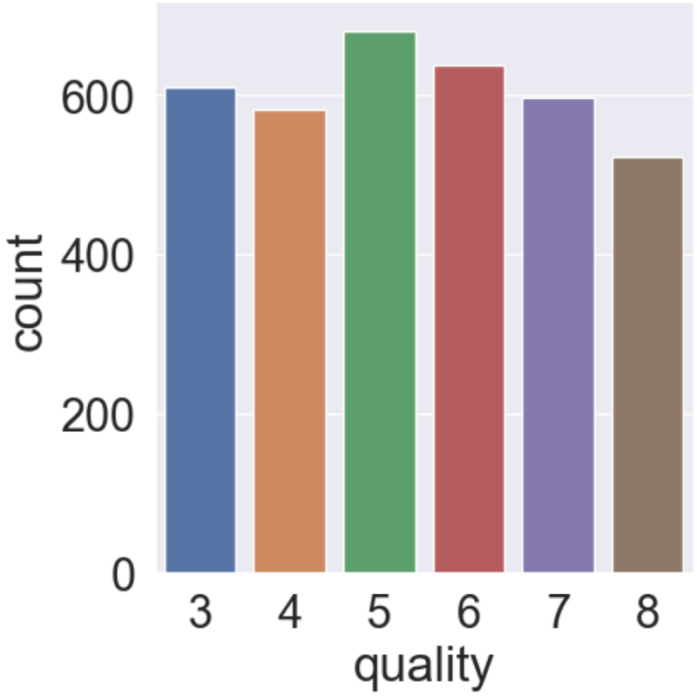
And now, a batch prediction:

```
Targets: tensor([2, 2, 3, 2, 3, 2, 2, 3, 3, 4, 2, 1, 3, 3, 1, 3, 4, 3, 2, 3, 3, 2, 2, 2,
        4, 2, 2, 2, 4, 3, 3, 2, 3, 3, 2, 2, 2, 4, 2, 3, 3, 2, 3, 4, 2, 2, 3, 3,
        3, 3, 4, 3, 2, 3, 2, 4, 2, 4, 3, 2, 3, 3, 2, 3])
Predicted class: tensor([6, 6, 6, 5, 6, 5, 5, 5, 7, 6, 5, 5, 5, 6, 6, 6, 7, 6, 5, 6, 6, 6, 5, 5,
       6, 6, 6, 5, 6, 6, 5, 5, 6, 6, 5, 5, 5, 6, 5, 6, 6, 5, 6, 6, 5, 5, 6, 6,
       5, 5, 6, 6, 5, 6, 5, 7, 6, 7, 6, 5, 6, 5, 5, 6])
```

We have understood that this model reaches only about 50% of accuracy, and we know that the dataset is imbalanced. So, I used the oversampling technique of SMOTE to try to obtain better results. I performed SMOTE on the platform Microsoft Azure Machine Learning. Another option, equivalent to oversampling, was the class weighting, which consists in passing as argument to the Cross Entropy Loss a tensor of weights, in which the weight of class c is equal to the size of the largest class divided by the size of class c. Weighting classes is also better from storage and computational point of view since it avoids working with a larger data-set, but I used SMOTE because in this case the dataset was very small. So, I will read the new dataset and perform the data prep in a single cell.
Here is the workflow of my experiment on Microsoft Azure:

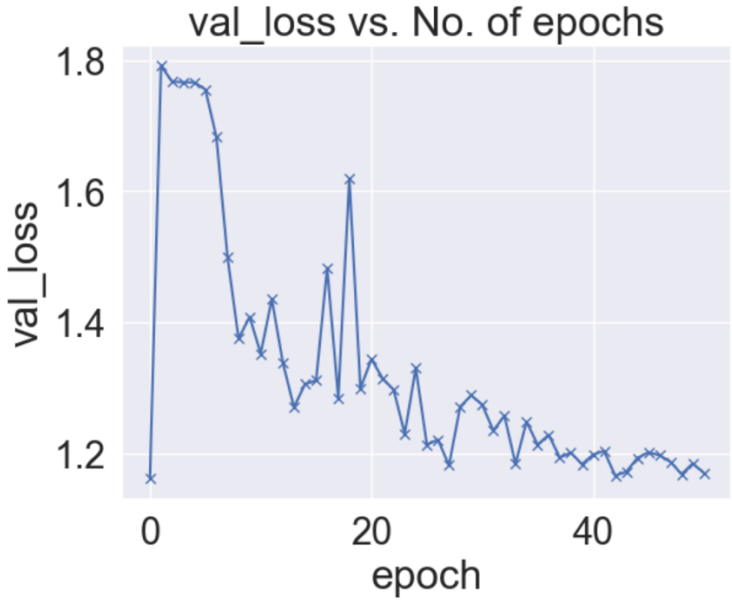


Now, the dataset is more balanced:

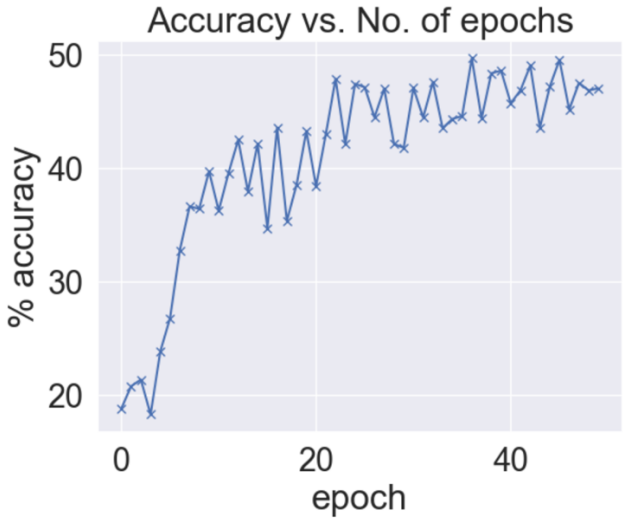


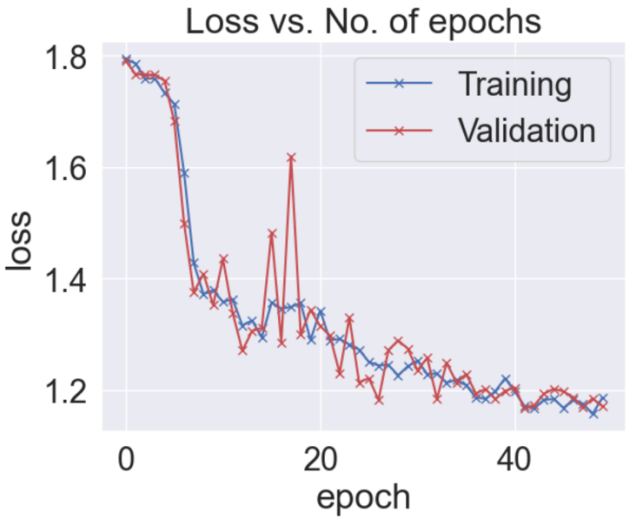
I used the same parameters of before. Let's evaluate the model before the training and then start with training:

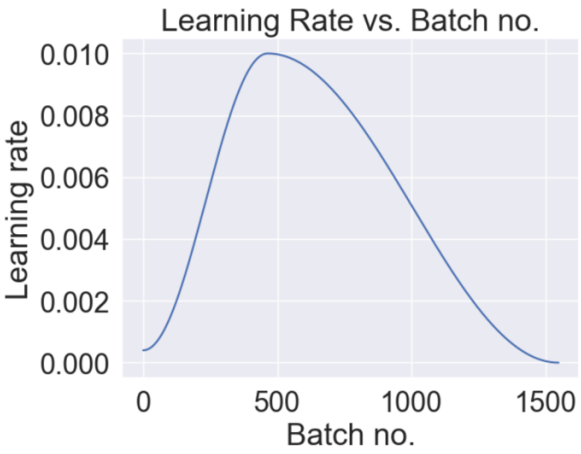
```
{'val_loss': 1.8258535861968994, 'val_acc': 0.1868106573820114}
```




```
Input: tensor([ 7.5828,  0.3362,  1.7692,  0.0668,  2.6050,  3.2250,  0.8903, 11.6655])
Target: tensor(8)
Prediction (logits): tensor([-8.4125, -5.5694, -1.2872,  1.6220,  4.6969,  5.2533])
Prediction (probabilities): tensor([7.2541e-07, 1.2454e-05, 9.0163e-04, 1.6539e-02, 3.5801e-01, 6.2454e-01])
Predicted class (-3): tensor([5])
Predicted class: 8
```



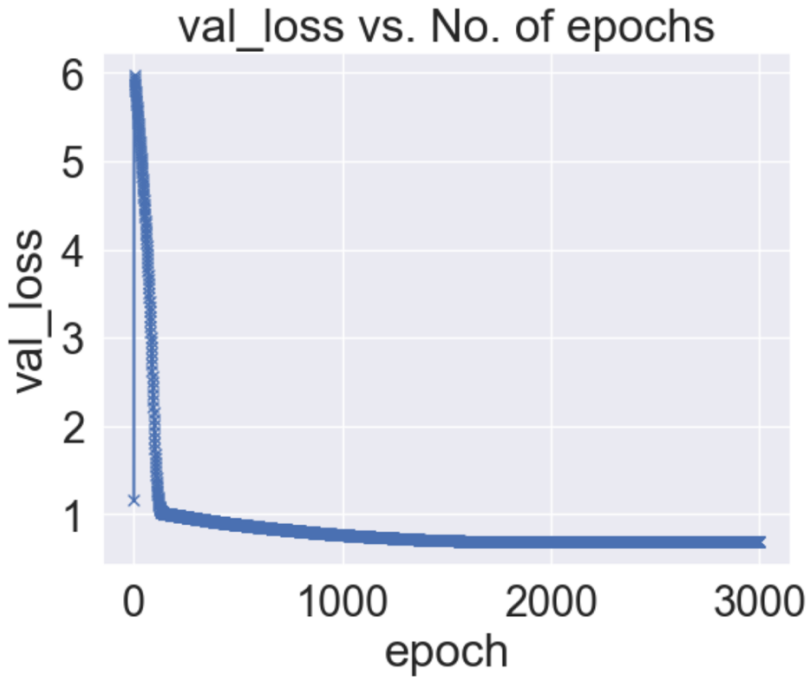




We can see that also with a balanced dataset, the results in terms of accuracy and loss do not improve so much. I tried also with weighting classes, but both these solutions failed. So now I will do the last experiment in order to obtain better results.

Now, I wanna treat my problem as a regression problem. So I will create a Linear Regression Model, using the L1 Loss, and 1 as output_size. I used also a simpler model and less evaluation terms, to go faster.

```
WineQuality(
  (net): Sequential(
    (0): Linear(in_features=8, out_features=14, bias=True)
    (1): PReLU(num_parameters=1)
    (2): Linear(in_features=14, out_features=1, bias=True)
  )
)
```




```
Input: tensor([0.8626, 0.5700, 2.1000, 0.1150, 3.5177, 3.3800, 0.6900, 0.2610])
Target: tensor([5.])
Prediction: tensor([5.4170])
```

Here you can see quality of wine is 5. and our model is predicting 5.4170 which is pretty close. We can try other samples and get the accuracy of prediction.

For further works, it can be nice to continue working on the problem of classification, instead of regression, because it is more appropriate with this dataset.
It could be useful try to find better values for the hyperparameters, or a better architecture of the neural network.
We could also build new classes upon them (something like bad/medium/good quality wine), and try to use SMOTE to fix the imbalance also in this case.
We'll loose a bit of "resolution" but we may end up with better predictions.