

WHITE WINE QUALITY PREDICTION

Overview

” Wine is the most healthful and most hygienic of beverages “

– Louis Pasteur.

We definitely came across the fruit **grapes**, which is so sweet on the test but grapes are not just to eat, they are used to make different types of things. Wine is one of them. **Wine is an alcoholic drink that is made up of fermented grapes**. If you have come across wine, then you will notice that wine has also their type; they are **red and white wine** this was because of different varieties of grapes.

According to experts, the wine is differentiated according to its **smell, flavor, and color**, but we are not a wine expert to say that wine is good or bad. What will we do then? Here's we use of **Machine Learning and Deep Learning**. These datasets can be viewed as classification or regression tasks. The classes are ordered and not balanced (e.g. there are much more normal wines than excellent or poor ones). we are using machine learning and deep learning to check wine quality.

Dataset and its Description

White wine Dataset: <https://www.kaggle.com/abdullah0a/wine-quality-red-white-analysis-dataset>

From the dataset, there are several features will be used to classify the quality of wine, many of them are chemical, so we need to have a basic understanding of such chemicals. let us see what each chemical terms indicates;

- **Volatile Acidity:** Volatile acidity is the gaseous acids present in wine.
- **Fixed Acidity:** Primary **fixed acids** found in wine are **tartaric, succinic, citric, and malic**.
- **Residual Sugar:** Amount of sugar left after fermentation.
- **Citric Acid:** It is weak organic acid, found in citrus fruits naturally.
- **Chlorides:** Amount of salt present in wine.
- **Free Sulphur-dioxide:** SO_2 is used for prevention of wine by oxidation and microbial spoilage.

- **pH:** In wine, pH is used for checking acidity
- **Sulphates:** Added sulphites preserve freshness and protect wine from oxidation, and bacteria.
- **Alcohol:** Percent of alcohol present in wine.

Importing Modulus

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
```

Let's we take brief about these libraries, **pandas** are used for data analysis; **NumPy** is for n-dimensional array; **seaborn** and **matplotlib** both have similar functionalities which are used for visualization.

Scikit-learn is a **Python library** that provides simple tools for machine learning, including **data preprocessing, model training, and evaluation**. It supports tasks like **classification, regression, clustering, and dimensionality reduction**.

Train_test_split is a function in Scikit-learn used to split your dataset into Training part and testing part.

RandomForestClassifier is a machine learning algorithm in Scikit-learn used for **classification** tasks. It builds an ensemble of decision trees, and combines their outputs to make more **accurate** and **stable predictions**.

Accuracy_score calculates the **ratio of correct predictions** to the total number of predictions.

```
# Loading the dataset
White_wine = pd.read_excel("/content/winequality_white.xlsx")
```

In the Google Collab, I upload the dataset which is downloaded from the Kaggle.

Observations from Dataset

The Observation is based on the technical information contained in the data.

```
# Checking the rows and columns in the dataframe
White_wine.shape
```

(4898, 12)

The **Shape** function is used here to find the **numbers of rows and columns** present in our dataset.

Here we have **4898 rows and 12 Columns** which is shown under the code as the output.

```
# First 5 rows of the dataset
White_wine.head()
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.0	0.27	0.36	20.7	0.045	45.0	170.0	1.0010	3.00	0.45	8.8	6
1	6.3	0.30	0.34	1.6	0.049	14.0	132.0	0.9940	3.30	0.49	9.5	6
2	8.1	0.28	0.40	6.9	0.050	30.0	97.0	0.9951	3.26	0.44	10.1	6
3	7.2	0.23	0.32	8.5	0.058	47.0	186.0	0.9956	3.19	0.40	9.9	6
4	7.2	0.23	0.32	8.5	0.058	47.0	186.0	0.9956	3.19	0.40	9.9	6

We can't be able to see the features of the entire dataset, that's y for our convenience, we are using this **head()** function in order to view **the first 5 rows of our dataset**.

The **isnull().sum()** function is used to check how many missing values (nulls) exist in each column of the dataset. This is a common step in **data cleaning** to decide if you need to handle missing data.

```
# Checking for the missing values
white_wine.isnull().sum()
```

	0
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 output is listed below the code which has a **0** at each column name, which indicates that the dataset is in good manner and all the data's are present without being blank.

```
#statistical measures of the dataset
white_wine.describe()
```

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

Describe function is used to understand the **range and distribution** of data and to quickly get a feel for the **scale of each feature**.

```
White_wine.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
```

Info() is the function used to see **data types** before preprocessing (e.g. encoding categorical variables). It is helpful in finding whether the data's are in **categorical values** or **the integers**.

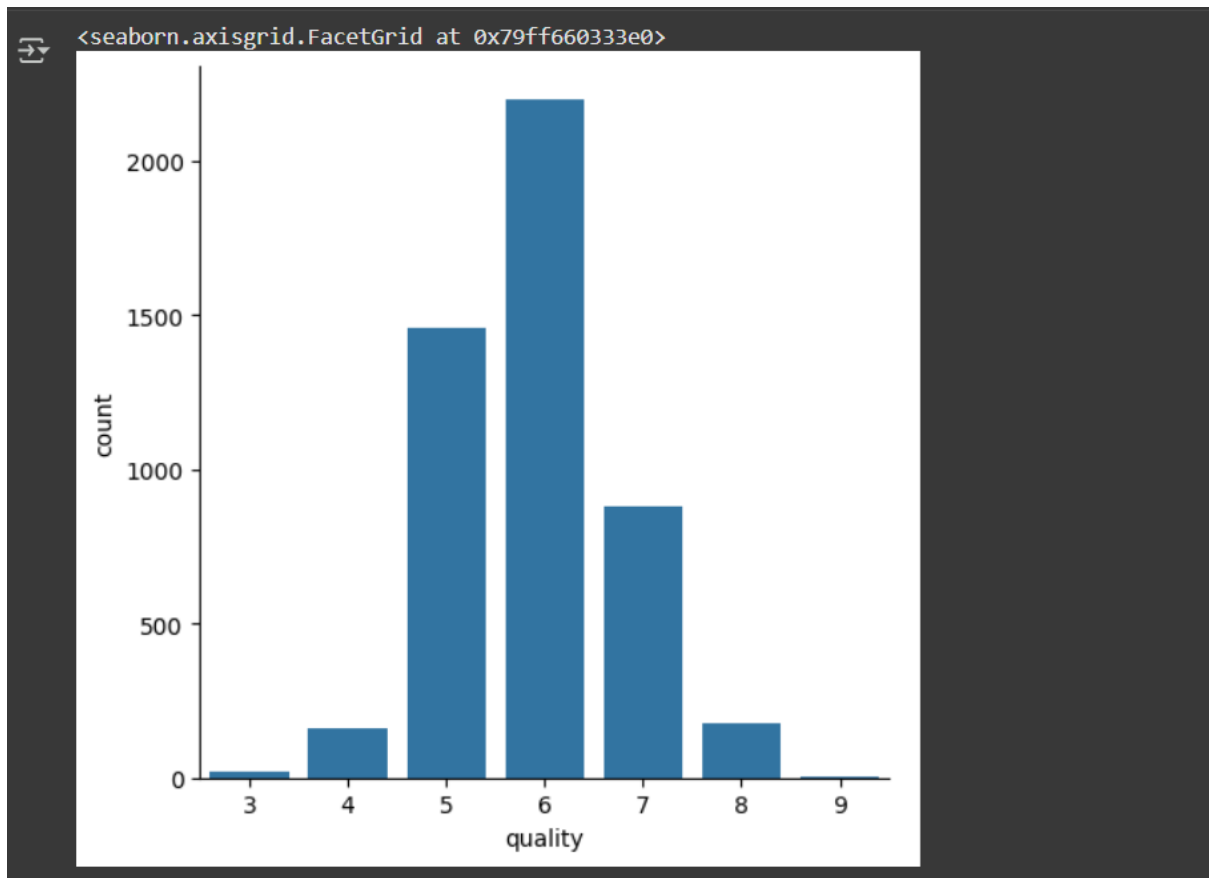
Data Visualization

We know that the “image speaks everything” , here the visualization came into the work, we use visualization for explaining the data. For Visualization we use **Matplotlib.pyplot** and **the seaborn**.

```
# Number of values for each quality
sns.catplot(x='quality', data=White_wine, kind='count')
```

The above code is the **categorical plot** that shows the **count of wine samples** for each **quality rating** in the dataset.

OUTPUT PLOT:



A **bar chart** with:

- **x-axis:** different **quality scores** (e.g., 3 to 9).
- **y-axis:** number of wine samples for each quality score.

The **height** of each bar is equal to how many wines have that specific quality rating.

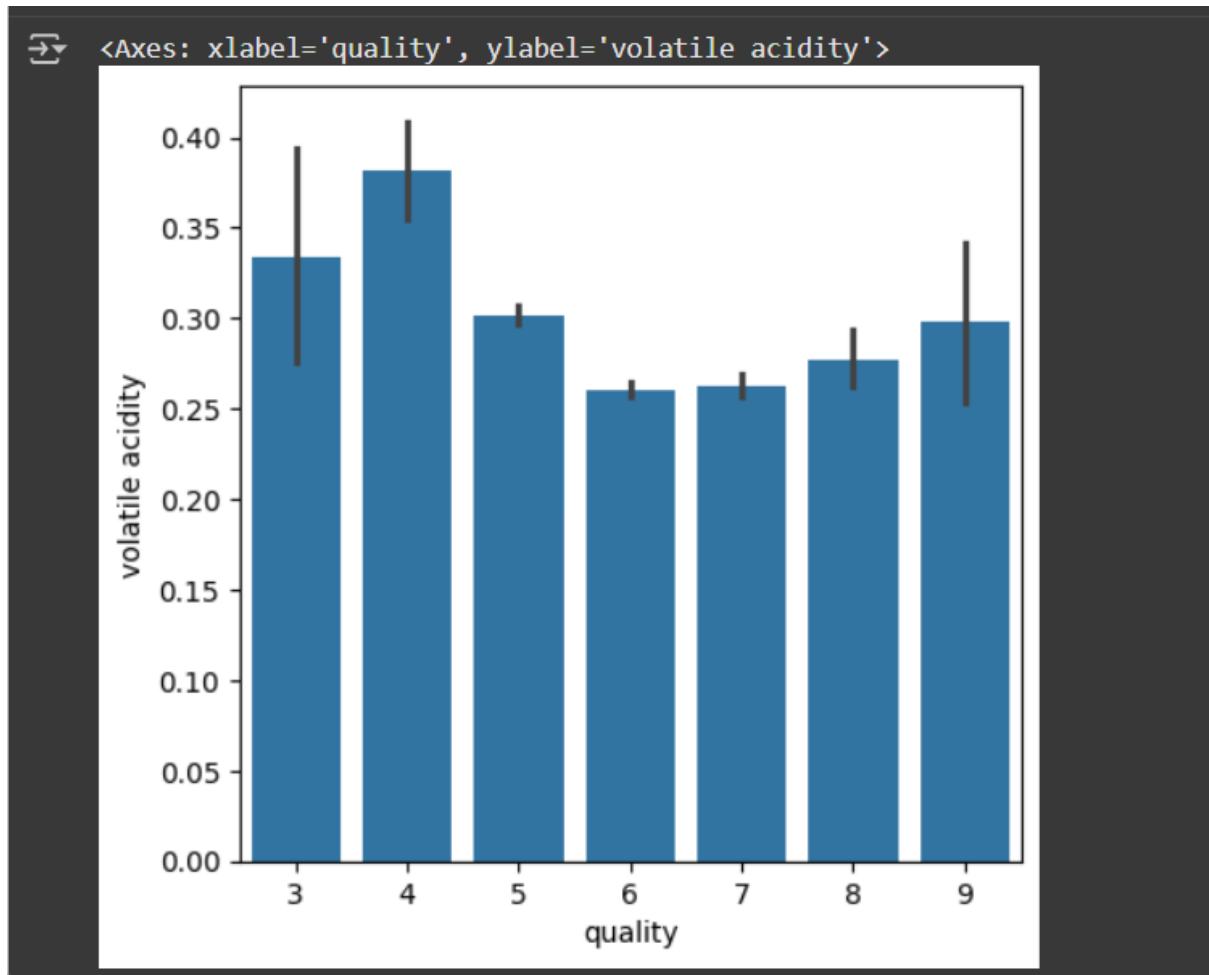
This plot is useful for **exploratory data analysis (EDA)** to:

- To see how **balanced** or **imbalanced** the quality labels are.
- It helps to understand if classification task might suffer from **class imbalance**.

```
# Volatile acidity vs quality
plot = plt.figure(figsize=(5,5))
sns.barplot(x='quality', y='volatile acidity', data=White_wine)
```

A new figure has been plotted using **Matplotlib**. Here the **figsize=(5,5)** sets the size of the plot window to **5 inches by 5 inches**. This plot holds a reference to the figure.

OUTPUT PLOT:



If bars **decrease** as quality increases, it may suggest that **lower volatile acidity** is associated with **higher-quality wine**. When we performing any machine learning operations then we have to study the data features deep, there are many ways by which we can differentiate each of the features easily. Now, we will perform a correlation on the data to see how many features are there they correlated to each other.

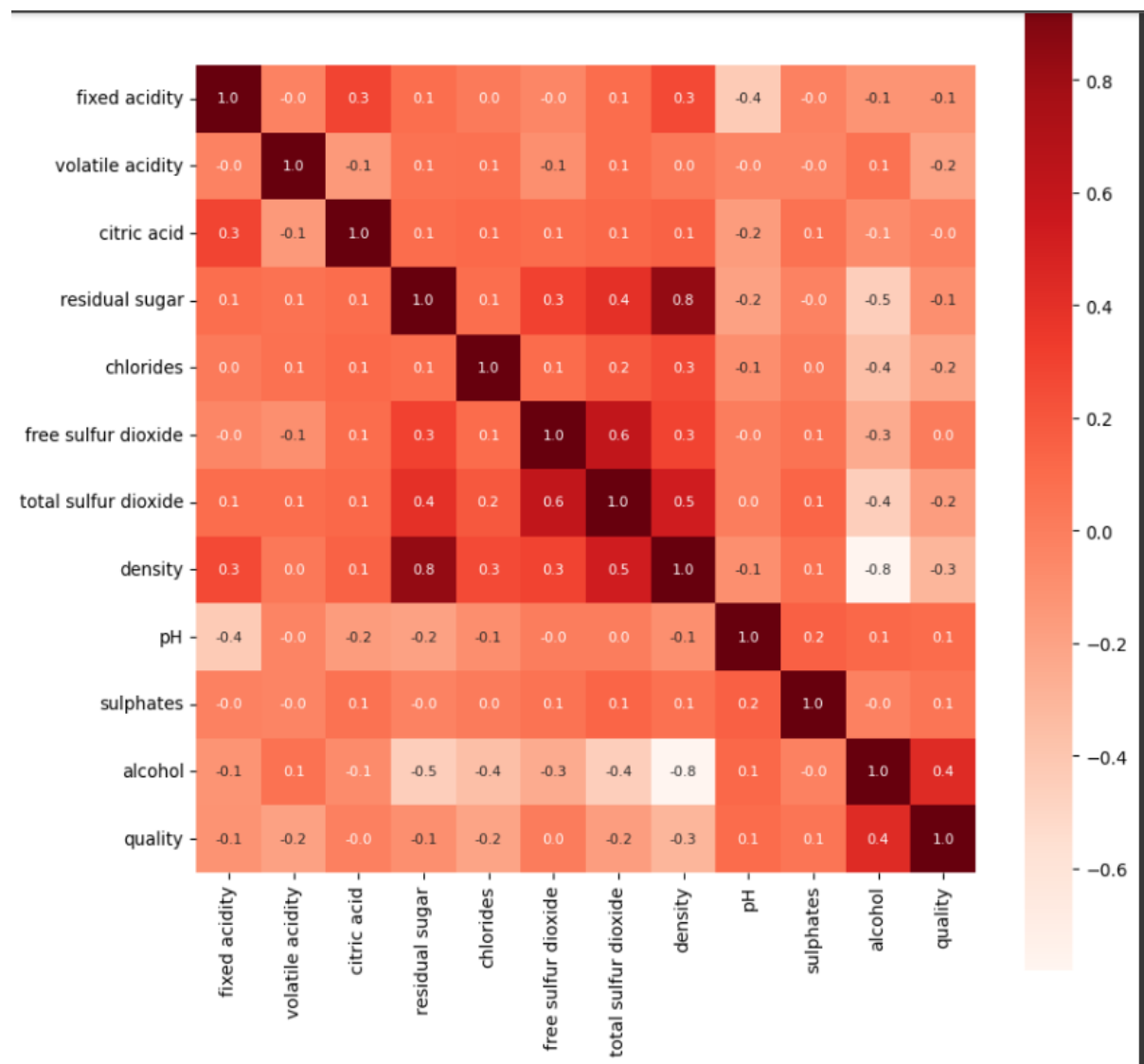
Correlation

For checking correlation, we use a statistical method that finds the bonding and relationship between two features.

```
# Construction of heatmap  
plt.figure(figsize=(10,10))  
sns.heatmap(corelation, cbar=True, square=True, fmt='.1f', annot=True, annot_kws={'size':8}, cmap='Reds')
```

This matrix shows how strongly each pair of numerical features in the White wine dataset is related.

OUTPUT PLOT:



This plot is used to identify:

- **Highly correlated features** (which may be redundant).
- **Negative correlations** (inverse relationships),
- Features that may impact the **target variable** (like quality).

Data Pre-processing

Data preprocessing is the essential step in a machine learning workflow. It involves **cleaning**, **transforming**, and **organizing** raw data into a format that can be effectively used by machine learning models.

```
# Separate features and target
x = White_wine.drop('quality', axis=1)
y = White_wine['quality']
```

To Train any model, the **feature variables (X)** and a **target variable (y)** must be separated. **X** will contain **input features** the model uses to learn. **Y** is what the model tries to **predict**.

Label Binarization

Label binarization is a technique used in preprocessing to **convert categorical labels (classes) into a binary format** and often needed for algorithms that expect binary or multi-label target variables.

```
# Label Binarization
from sklearn.preprocessing import LabelBinarizer
encoder = LabelBinarizer()
y = encoder.fit_transform(y)
```

```

y= White_wine['quality'].apply(lambda y: 1 if y>=7 else 0)
print(y)

```

```

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

```

After doing the **binarizing** the target variable:

- Wines with quality **7 or more** become **1**, consider them “**good/high quality**”
- Wines with quality **less than 7** become **0**, “**not good / lower quality**”.

Splitting the dataset

Now we perform a split operation on our dataset; that is splitting our data into training data and the test data.

```

# Train/test split
X_train, X_test, Y_train, Y_test = train_test_split(x,y, test_size=0.2, random_state=3)

```

```

print(y.shape, Y_train.shape, Y_test.shape)

```

```

(4898,) (3918,) (980,)

```

The explanation of each terms follows below;

1. X_{train} → 80% of the rows of x , to train your model
2. X_{test} → 20% of the rows of x , to evaluate the model later
3. Y_{train} → the corresponding target values (0 or 1) for X_{train}
4. Y_{test} → the corresponding target values for X_{test}

Model Training

This is the last step where we apply any suitable model which will give more accuracy, here we will use **Random Forest Classifier**.

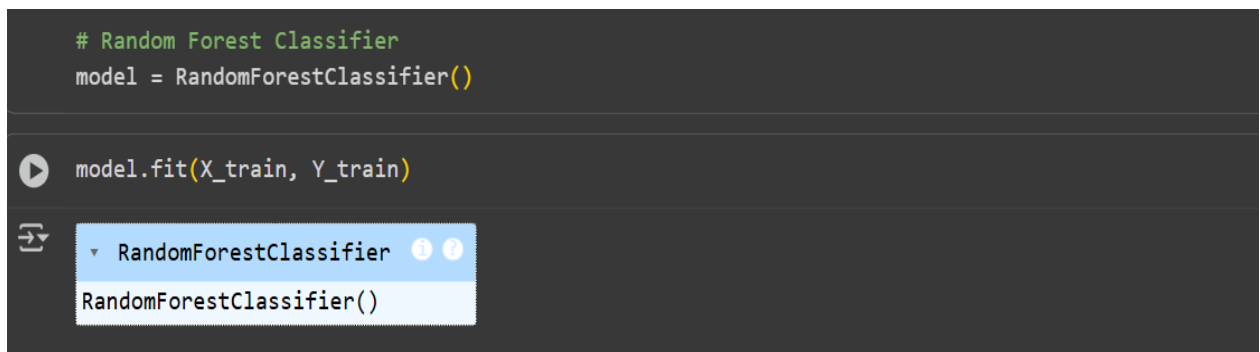
RANDOM FOREST CLASSIFIER:

<: It's an ensemble learning method for classification tasks.

<: It builds many decision trees (each a weak learner) on random subsets of the data and features, then aggregates their predictions (usually by majority vote) to produce a final prediction. This helps reduce overfitting and increases generalization.

```
# Random Forest Classifier
model = RandomForestClassifier()

model.fit(X_train, Y_train)
```

A screenshot of a Jupyter Notebook interface. The top cell contains the code to import the RandomForestClassifier and fit it to training data. The bottom cell shows the execution of the fit method, with a dropdown menu displaying the class name RandomForestClassifier and its constructor signature RandomForestClassifier().

```
# Random Forest Classifier
model = RandomForestClassifier()

model.fit(X_train, Y_train)
```

RandomForestClassifier

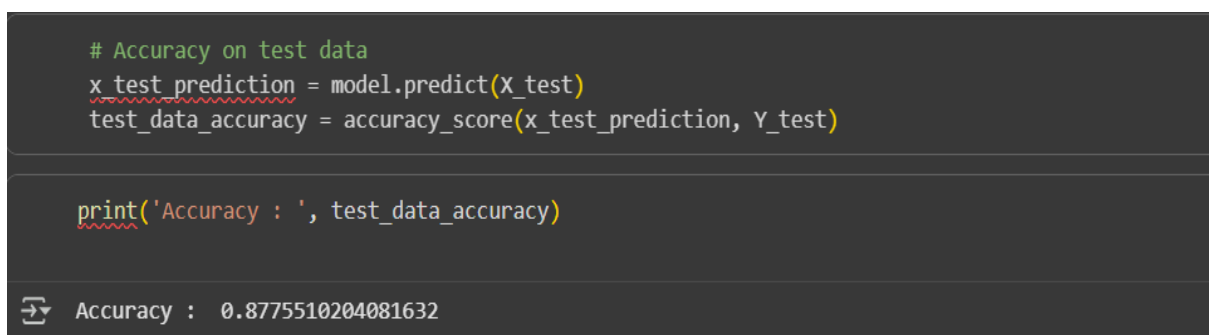
RandomForestClassifier()

Model Evaluation

“Model evaluation” in the context of using a classifier like Random Forest means checking how well the model’s predictions match the real outcomes.

```
# Accuracy on test data
x_test_prediction = model.predict(X_test)
test_data_accuracy = accuracy_score(x_test_prediction, Y_test)

print('Accuracy : ', test_data_accuracy)
```

A screenshot of a Jupyter Notebook interface. The top cell contains the code to calculate the accuracy of the model on test data. The bottom cell shows the execution of the print statement, displaying the accuracy value 0.8775510204081632.

```
# Accuracy on test data
x_test_prediction = model.predict(X_test)
test_data_accuracy = accuracy_score(x_test_prediction, Y_test)

print('Accuracy : ', test_data_accuracy)
```

Accuracy : 0.8775510204081632

By using the Random Forest Classifier, we got the **accuracy** of **0.8775** which is the good prediction of quality of white wine.

Building a predictive system

```
Building a predictive system

input_data = (7.3,0.65,0.0,1.2,0.068,15.0,21.0,0.9946,3.39,0.47,10.0)

# Changing the input data into numpy array
input_data_as_numpy_array = np.asarray(input_data)

# Reshaping the data
input_data_reshaped = input_data_as_numpy_array.reshape(1,-1)
prediction = model.predict(input_data_reshaped)
print(prediction)
if (prediction[0]==1):
    print('Good Quality Wine')
else:
    print('Bad Quality Wine')
```

[0]
Bad Quality Wine

Here the input data is the datas taken from the Kaggle dataset.

For example; we take the 1st line from the dataset as input data as numpy array for the prediction. The code is trying to reshape input data and use a trained model to predict the quality of wine. It Converts a 1D array of features into the correct shape ((1, n_features)) for prediction with Random Forest Classifier.

Each line of dataset is tested within this code and if the output is **Zero (0)**, it predicts that the quality of wine is **Bad**.

If the predicted output is **one (1)**, then it shows that the quality of wine is **Good**.

So, at this step, our machine learning prediction is over.

Deep Neural Network regression

A **Deep Neural Network (DNN) for regression** is a type of neural network specifically designed to **predict continuous numeric values** rather than categories. Unlike classification, where the output is a class label (e.g., "High" or "Low"), regression predicts a **quantitative outcome** (e.g., wine quality score, house price, or temperature).

A typical DNN regression model consists of:

- <: Input Layer
- <: Hidden Layer
- <: Output Layer

1. The network takes input features and passes them through hidden layers.
2. Each hidden layer applies weights, biases, and activation functions to extract **non-linear patterns**.
3. The final output layer produces the **predicted numeric value**.

The network learns by minimizing a **loss function** that measures the difference between predicted and actual values.

- Common loss functions for regression:
 - **Mean Squared Error (MSE)**
 - **Mean Absolute Error (MAE)**

Now, I am, going to build the **DNN Regression using tensor flow Keras API**. Using multiple hidden layers and nonlinear activations (like ReLU), the network can **learn these complex patterns** automatically from data.

Modelling and Training

```
# Building a multiclass classification
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
x_scaled = scaler.fit_transform(x)
x = pd.DataFrame(x_scaled, columns=x.columns)
print(x)
```

Here I used the standardscaler models (especially neural networks, SVM, KNN, logistic regression) to perform better when features are on a similar scale. It Helps gradients converge faster in deep learning models and avoids bias due to differing feature magnitudes.

Preparing data for Multiclass classification (Deep Learning)

```
df_train = White_wine.sample(frac=0.7, random_state=0)
df_valid = White_wine.drop(df_train.index)
display(df_train.head(4))
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
2762	7.3	0.32	0.35	1.4	0.050	8.0	163.0	0.99244	3.24	0.42	10.7	5
42	7.0	0.31	0.26	7.4	0.069	28.0	160.0	0.99540	3.13	0.46	9.8	6
1419	7.6	0.14	0.74	1.6	0.040	27.0	103.0	0.99160	3.07	0.40	10.8	7
3664	5.0	0.29	0.54	5.7	0.035	54.0	155.0	0.98976	3.27	0.34	12.9	8

After running this code:

- df_train: 70% of White_wine → used for **training**.
- df_valid: 30% of White_wine → used for **validation/testing**.
- You have a **clean split** of your data without overlap.

```
# Scale to [0,1]

max_ = df_train.max(axis=0)
min_ = df_train.min(axis=0)
df_train = (df_train - min_) / (max_ - min_)
df_valid = (df_valid - min_) / (max_ - min_)

# Split features and target
X_train = df_train.drop('quality', axis=1)
X_valid = df_valid.drop('quality', axis=1)
y_train = df_train['quality']
y_valid = df_valid['quality']

print(X_train.shape)
```

(3429, 11)

Step 1 — Compute min and max for each column

- `df_train.max(axis=0)` → gets the **maximum value** in each column (feature).
- `df_train.min(axis=0)` → gets the **minimum value** in each column (feature).
- These are stored in `max_` and `min` - each is a Pandas Series with one value per column.
-

Step 2 — Split features and target

<: `drop('quality', axis=1)` → removes the column quality (the target variable) from the dataset.

- The result is the feature matrix (`X_train` and `X_valid`).
- <: The column 'quality' itself is extracted separately:
- `y_train` and `y_valid` → these are your target vectors for training and validation respectively.

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    layers.Dense(units=512, activation='relu', input_shape=[11]),
    layers.Dense(units=512, activation='relu'),
    layers.Dense(units=512, activation='relu'),
    layers.Dense(units=1)
])

model.compile(optimizer='adam', loss='mae')
history = model.fit(
    X_train, y_train,
    validation_data = (X_valid, y_valid),
    batch_size=256,
    epochs=10,
)
```

First, the Sequential model is defined as a stack of fully connected (Dense) layers. The network begins with an input layer that expects **11 features** (since the dataset has 11 chemical properties). It then has **three hidden layers**, each **with 512 neurons and the ReLU (Rectified Linear Unit) activation function**,

which helps the model learn **complex non-linear patterns** by introducing non-linearity and reducing vanishing gradient issues. The final layer has one neuron with no activation, which outputs a single continuous value that is suitable for regression tasks like predicting wine quality.

Next, the model is **compiled** using the **Adam optimizer**, an adaptive version of gradient descent that efficiently updates weights during training. The **loss function** used is **MAE (Mean Absolute Error)**, which measures the average absolute difference between predicted and actual values, making it robust and easy to interpret.

Finally, the model is **trained** using the `.fit()` method with the normalized training data (`X_train`, `y_train`) and validated against `X_valid`, `y_valid` after each epoch. The **batch size of 256** means the model updates weights after processing 256 samples at a time, and **10 epochs** indicate it will iterate over the entire dataset 10 times. The training progress and validation performance are tracked and stored in the history object, which can later be used to visualize learning curves (loss over epochs).

OUTPUT EPOCH'S

```
Epoch 1/10
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/dense.py:93: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequ
super().__init__(activity_regularizer=activity_regularizer, **kwargs)
14/14 ━━━━━━━━━━━ 2s 38ms/step - loss: 0.2663 - val_loss: 0.1184
Epoch 2/10
14/14 ━━━━━━━━━━━ 0s 27ms/step - loss: 0.1129 - val_loss: 0.1020
Epoch 3/10
14/14 ━━━━━━━━━━━ 0s 28ms/step - loss: 0.1014 - val_loss: 0.0988
Epoch 4/10
14/14 ━━━━━━━━━━━ 1s 28ms/step - loss: 0.1009 - val_loss: 0.0976
Epoch 5/10
14/14 ━━━━━━━━━━━ 0s 28ms/step - loss: 0.0978 - val_loss: 0.0962
Epoch 6/10
14/14 ━━━━━━━━━━━ 0s 28ms/step - loss: 0.0968 - val_loss: 0.0947
Epoch 7/10
14/14 ━━━━━━━━━━━ 0s 27ms/step - loss: 0.0962 - val_loss: 0.0952
Epoch 8/10
14/14 ━━━━━━━━━━━ 0s 29ms/step - loss: 0.0960 - val_loss: 0.0925
Epoch 9/10
14/14 ━━━━━━━━━━━ 1s 28ms/step - loss: 0.0930 - val_loss: 0.0969
Epoch 10/10
14/14 ━━━━━━━━━━━ 0s 29ms/step - loss: 0.0942 - val_loss: 0.0919
```


We can see that Keras will keep you updated on the loss as the model trains. Often, a better way to view the loss though is to plot it. The `fit` method in fact keeps a record of the loss produced during training in a History object. We'll convert the data to a Pandas dataframe, which makes the plotting easy.

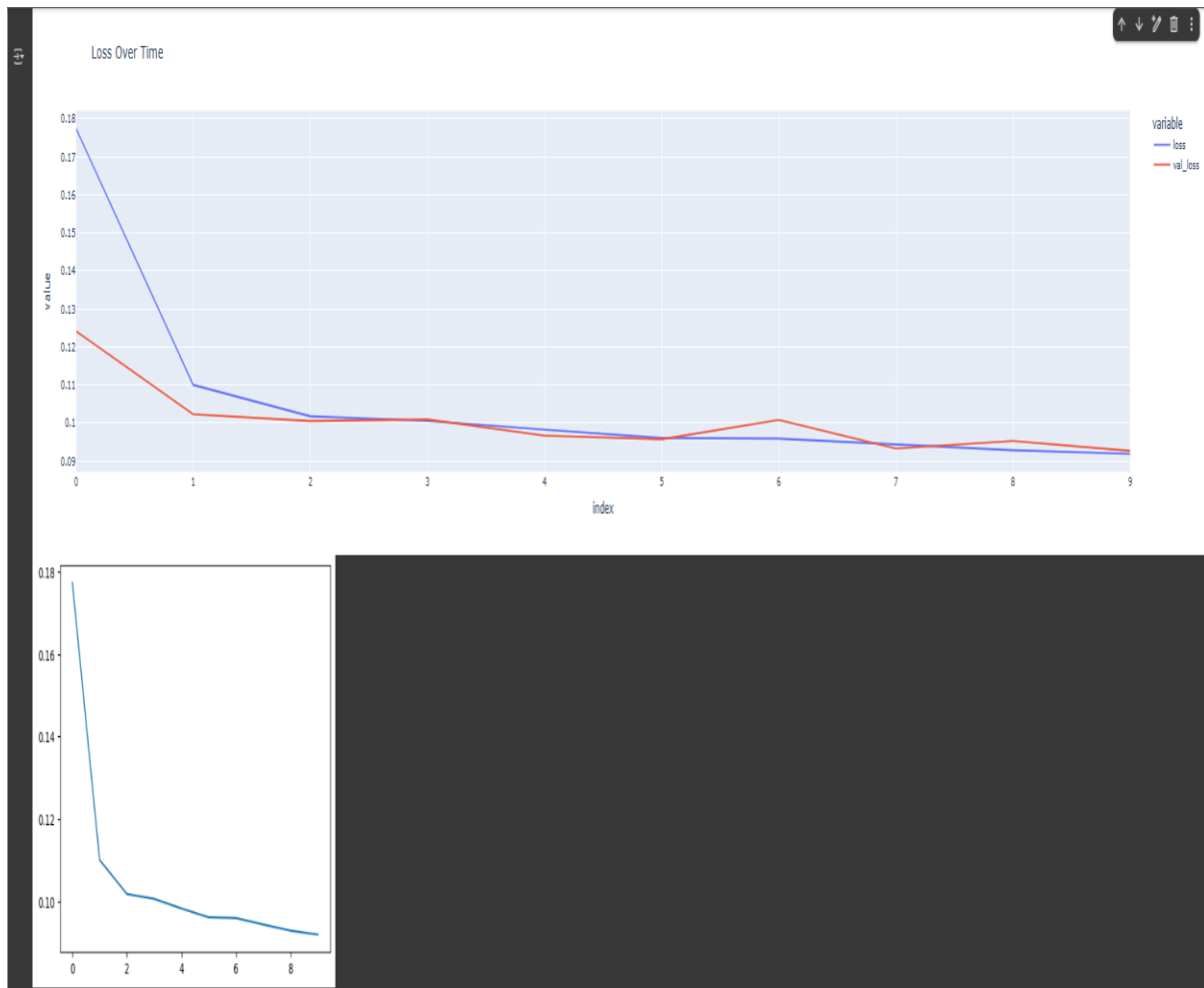
```
history_df = pd.DataFrame(history.history)
history_df['loss'].plot()

fig.show()
```

Training a model with `model.fit()`, Keras returns a history object that records the loss (and any metrics) for each epoch — both for the **training** and **validation** sets. The line `pd.DataFrame(history.history)` converts this information into a **DataFrame** called `history_df`, where each column represents a tracked metric and each row corresponds to an epoch.

Then, the line `history_df['loss'].plot()` creates a simple line plot of the training loss over time (across epochs). This graph helps you **visualize the learning progress** — we can see whether the model's loss is decreasing (indicating learning) or flattening (indicating convergence). A smooth, steadily decreasing curve means the model is learning well, while irregular or rising loss values may signal overfitting or learning issues.

Plotted figure



Notice how the loss levels off as the epochs go by. When the loss curve becomes horizontal like that, it means the model has learned all it can and there would be no reason continue for additional epochs.

Train a Model with Early Stopping

Now let's increase the capacity of the network. We'll go for a fairly large network, but rely on the callback to halt the training once the validation loss shows signs of increasing.

```

from tensorflow.keras import callbacks

early_stopping = callbacks.EarlyStopping(
    ... min_delta=0.001, # minimum amount of change to count as an improvement
    ... patience=20, # how many epochs to wait before stopping
    ... restore_best_weights=True,
)

model = keras.Sequential([
    layers.Dense(units=512, activation='relu', input_shape=[11]),
    layers.Dense(units=512, activation='relu'),
    layers.Dense(units=512, activation='relu'),
    layers.Dense(units=1)
])

model.compile(optimizer='adam', loss='mae')

history = model.fit(
    X_train, y_train,
    validation_data=(X_valid, y_valid),
    batch_size=256,
    epochs=500,
    callbacks=[early_stopping],
    verbose=0,
)

history_df = pd.DataFrame(history.history)
history_df.loc[:, ['loss', 'val_loss']].plot();
print("Minimum validation loss: {}".format(history_df['val_loss'].min()))

```

The model is built using Keras' Sequential API, consisting of **three hidden layers**, each with 512 neurons and the **ReLU (Rectified Linear Unit)** activation function, which helps the network learn complex non-linear relationships efficiently. The **input layer** expects 11 features (from the wine dataset), and the **output layer** has a single neuron since the task is regression (predicting one continuous value — wine quality).

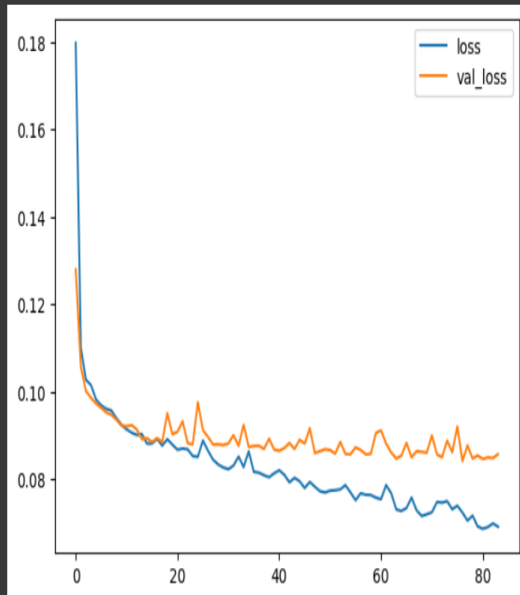
During training (`model.fit()`), the data is split into training (`X_train, y_train`) and validation (`X_valid, y_valid`) sets. The model trains for up to **50 epochs** with a **batch size of 256**, but training can stop early if the **early_stopping callback** detects that the validation loss no longer improves — this helps prevent overfitting and saves computation time. The parameter `verbose=0` hides the detailed training logs for a cleaner output. After training, the history object stores the loss and validation loss for each epoch.

OUTPUT PLOT

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/dense.py:93: UserWarning:
```

```
Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
```

```
Minimum validation loss: 0.08421353995800018
```



And sure enough, Keras stopped the training well before the full 500 epochs!

Using Dropout and Batch Normalization

We'll increase the capacity even more, but add dropout to control overfitting and batch normalization to speed up optimization. This time, we'll also leave off standardizing the data, to demonstrate how batch normalization can stabilize the training. When adding dropout, you may need to increase the number of units in your Dense layers.

```

model = keras.Sequential([
    layers.Dense(1024, activation='relu', input_shape=[11]),
    layers.Dropout(0.3),
    layers.BatchNormalization(),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.3),
    layers.BatchNormalization(),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.3),
    layers.BatchNormalization(),
    layers.Dense(1),
])

model.compile(optimizer='adam', loss='mae')

history = model.fit(
    X_train, y_train,
    validation_data=(X_valid, y_valid),
    batch_size=256,
    epochs=100,
    verbose=0
)

history_df = pd.DataFrame(history.history)
history_df.loc[:, ['loss', 'val_loss']].plot();

```

This code defines, trains, and evaluates an **advanced deep neural network (DNN) regression model** to predict a continuous target variable (wine quality) and then visualizes the learning progress over training epochs.

The model is built using Keras' Sequential API and consists of **three hidden layers**, each with **1024 neurons** and **ReLU activation**, which allows the network to learn complex non-linear relationships from the input features. To improve **generalization** and reduce overfitting, the model includes:

- **Dropout layers (0.3):** Randomly deactivate 30% of neurons during training to prevent the network from memorizing training data.
- **Batch Normalization layers:** Normalize the activations in each mini-batch, which stabilizes and speeds up training.
-

The **output layer** has a single neuron with no activation function, suitable for **regression**, predicting a continuous numeric value.

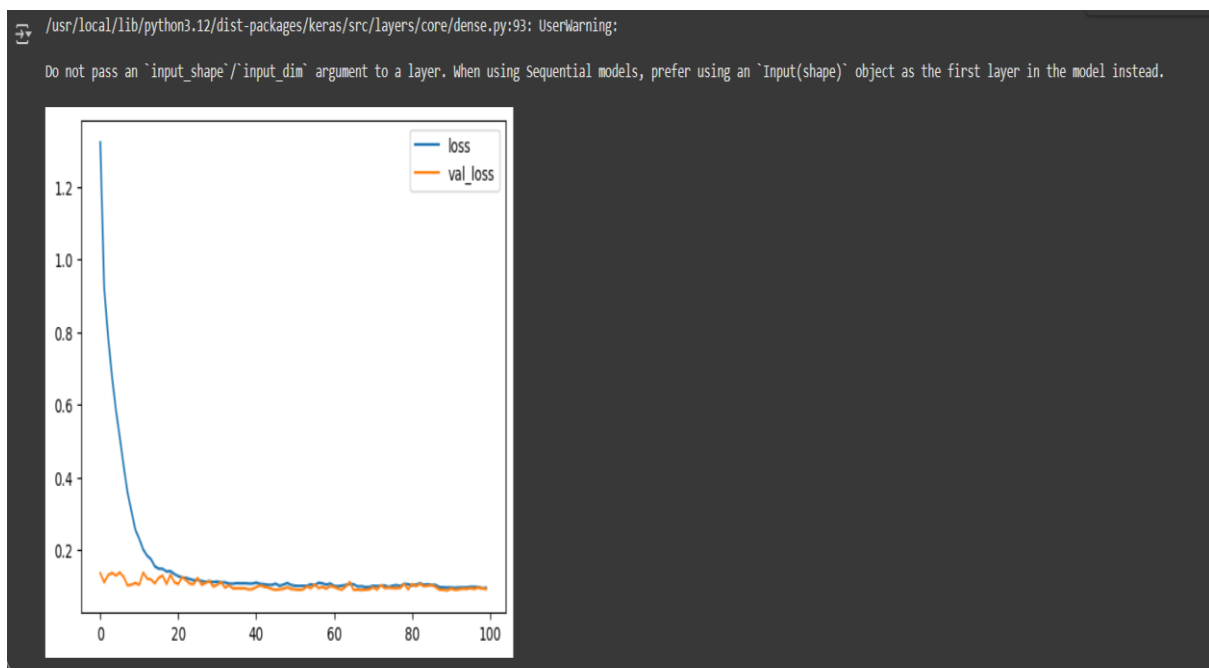
The model is compiled with the **Adam optimizer**, which adapts learning rates for faster convergence, and the **Mean Absolute Error (MAE)** loss function,

which measures the average absolute difference between predicted and actual values.

Training is performed using `.fit()` with a **batch size of 256** and for **100 epochs**, using `X_train` and `y_train` for learning and `X_valid` and `y_valid` for validation. The `verbose=0` parameter hides detailed logs, keeping the output clean.

After training, the history object stores the loss values for each epoch. By converting it into a **pandas DataFrame** (`history_df`) and plotting the columns `loss` (training loss) and `val_loss` (validation loss), we get a **visual representation of the model's learning curves**.

OUTPUT VISUAL



1. The model is **well-trained**: it captures the patterns in the data without overfitting.
2. MAE (Mean Absolute Error) around **0.1** indicates **high accuracy** in predicting wine quality.
3. The slight oscillation in validation loss is normal and shows minor fluctuations due to the batch updates during training.

Result Summary

In this project, we explored the prediction of white wine quality using both machine learning classification and DNN regression approaches.

Model	Type	Accuracy / MAE	Remarks
Random Forest	Classification	87.75%	Stable baseline
DNN	Regression	MAE = 0.10	Captures nonlinear patterns

The classifier provided a quick and interpretable way to categorize wines, but it struggled with borderline quality scores. By contrast, the DNN regression model captured subtle patterns in the data, allowing for more precise, continuous predictions of wine quality. Overall, combining these approaches demonstrates how data-driven methods can provide valuable insights for improving wine quality and guiding winemaking decisions.