**Final Project Report**

**Name: Siva Rama Rohan Sunkarapalli Email:** ssunkarapalli@crimson.ua.edu

**Statement:** For this assignment's preparation, I have utilized ChatGPT, a language model created by OpenAI. Within this assignment, the ChatGPT was used for purposes such as brainstorming and grammatical corrections.

# **Introduction**

In the data science community, it is astonishing that less than 20% of models or projects are used in the real world. That means over 80% of data science engineers create models unrelated to an application. There are myriad reasons for this phenomenon.

In this project, we took the opportunity to deploy the ML model we created on a local server powered by uvicorn using FastAPI endpoints. To begin with, we selected the Wine Quality dataset from UCI. We prepared a predictive model with gradient boosting and tuned hyperparameters, resulting in the lowest mean square error (MSE) of 0.38 on 5-fold cross-validation and 0.36 on the test dataset. This model was deployed and tested with a sample with no errors.

The report followed a general structure, beginning with a description of the dataset, model selection, and hyperparameter tuning, followed by model deployment and testing. In the end, we discussed the limitations of the framework. We presented solutions that fit real-world deployment, where multiple instances of the model need deploying and batch prediction is warranted.

# **Dataset Selection & Model Training**

We selected the wine quality dataset from the UCI website. The dataset aims to predict wine quality, given its attributes derived from physicochemical tests. The authors recommended the choice to define the problem either as a classification or a regression. We chose the regression problem because a muscular imbalance exists in the quality label.

## **2.1. Dataset Description**

The wine quality dataset consists of data for both red and white wine samples in two separate data files with 1599 and 4898 rows, respectively. We combined the two and added type variables to the input data to separate red and white wine by 0 and 1, respectively, to take advantage of all the samples in the dataset and improve the predictive power of a model.

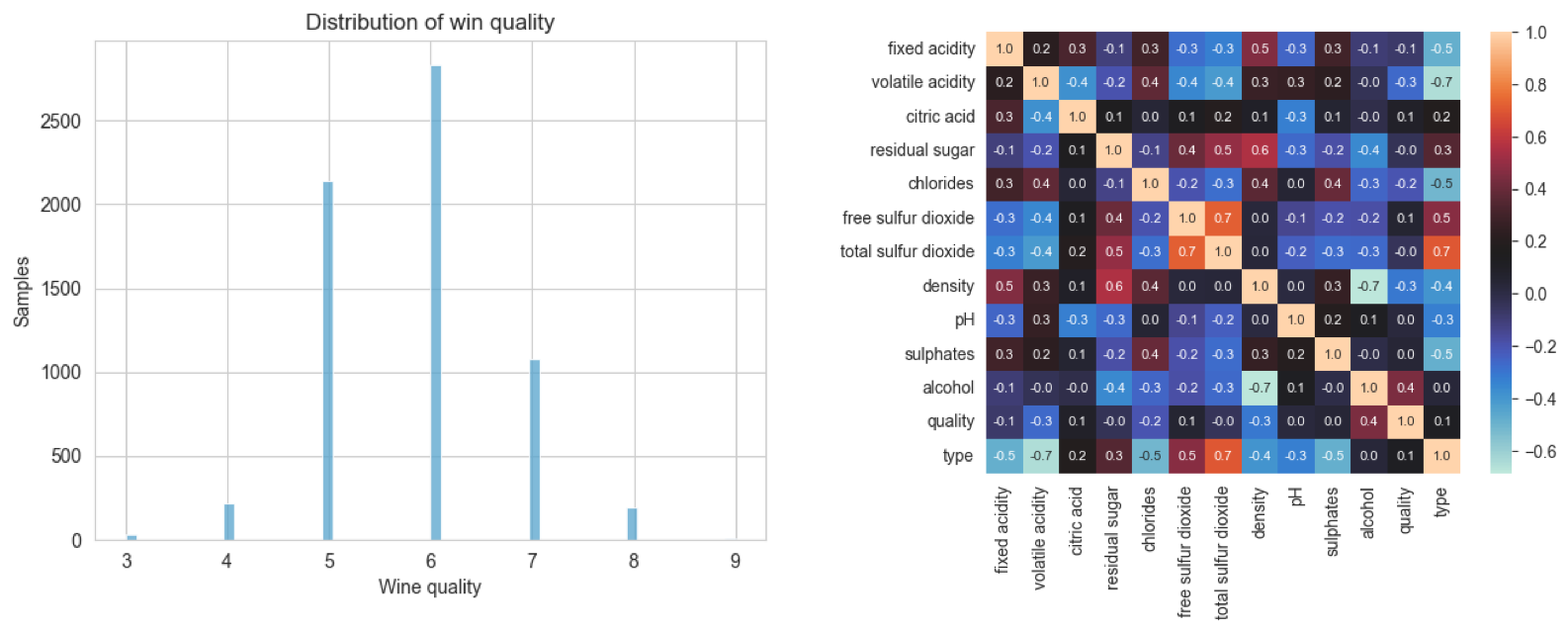
The list of attributes included in the dataset is fixed acidity, volatile acidity, citric acid, residual sugar, chlorides, free sulfur dioxide, total sulfur dioxide, density, pH, sulphates, alcohol, and type. Note that all the features in the dataset are continuous variables, except the synthetically created *type* variable is boolean. The *type* variable is treated continuously in training a predictive model. The dependent variable is the wine quality represented by the column name quality.

A screenshot of a cell phone

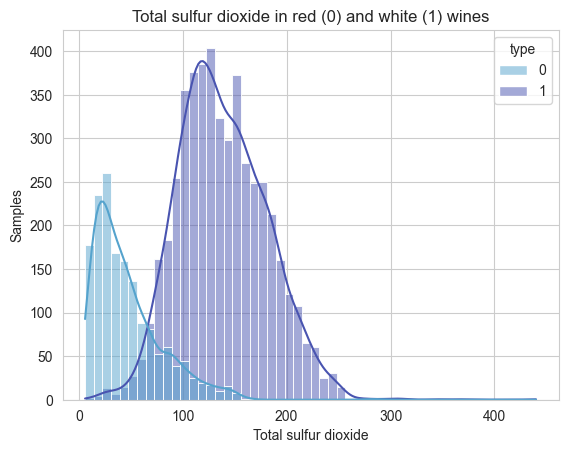
Description automatically generated

*Table - 1 Samples from the data set*

The distribution of the wine quality variable is shown in Figure 2a. The quality follows a normal distribution with 5.8 and a standard deviation of 0.873. Outliers exist in the dataset, which motivated us to frame the problem as regression rather than classification. The correlation between all the variables in the dataset is shown in Figure 2b. A strong correlation exists between free sulfur dioxide and total sulfur dioxide variables. Initially, this relationship is expected because the total sulfur dioxide is the combination of its free form and its molecular components in sulfates. We also noticed a strong relationship between total sulfur dioxide and wine type. The distribution of the compound in the respective wine category is shown in Figure 3. The trend follows a bimodal distribution, with white wine having higher sulfuric content than red wine.



*Fig 2a. Distribution of wine quality (left), 2b. Correlation between variables (right)*



*Figure 3 - Comparison of total sulfur dioxide in red and white wine*

No other variables correlate strongly with other features and the target variable - quality. The data is noisy, with strong outliers and an extended right-skewed distribution. In the next section, we will discuss training an ML model and storing the trained model.

## **2.2. Model Training and Serialization**

As discussed in the previous section, we combined red and white datasets to take advantage of all available samples. The training process is divided into two steps: (1) model selection and hyperparameter tuning, and (2) training all samples with optimized hyperparameters and checking performance on the test samples.

* **Data split for cross-validation:**

We utilized sklearn’s StratifiedKFold to split the dataset into six folds on quality. We used folds from 1 to 5 for training and cross-validation and reserved fold 6 for testing to evaluate the final performance of the predictive model.

A computer screen shot of text

Description automatically generated

*Figure 4 - Split k-fold*

As we discussed earlier of strong outlier influence on the dataset, we applied standardization for features and target (quality), using RobustScaler in the sklearn—preprocessing module. The method standardizes the values while taking into account their respective interquartile ranges.

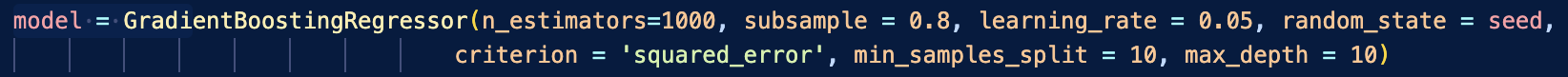
A computer screen shot of text

Description automatically generated

*Figure 5 - sklearn.preprocessing import RobustScaler*

### **Model selection and hyperparameter tuning**

We tried various regression algorithms and gradient boosting algorithms to produce better performance. The selection was done manually and iteratively by selecting algorithms between logistic regression, random forest regression, and gradient boosting regression. The hyperparameters that yield the lowest mean square error across the 5-fold cross-validation are as follows:



*Figure 6 - hyperparamters that yield the lowest mean square*

The mean of 5-fold MSE values for training and validation are 0 and 0.3809, respectively.

### **Final model training and serialization**

With the optimized hyperparameters, we trained the model with all the samples in the training dataset, from fold 1 to 5, and tested performance on fold 6, resulting in MSE with 2.54e-6 and 0.3650, respectively.

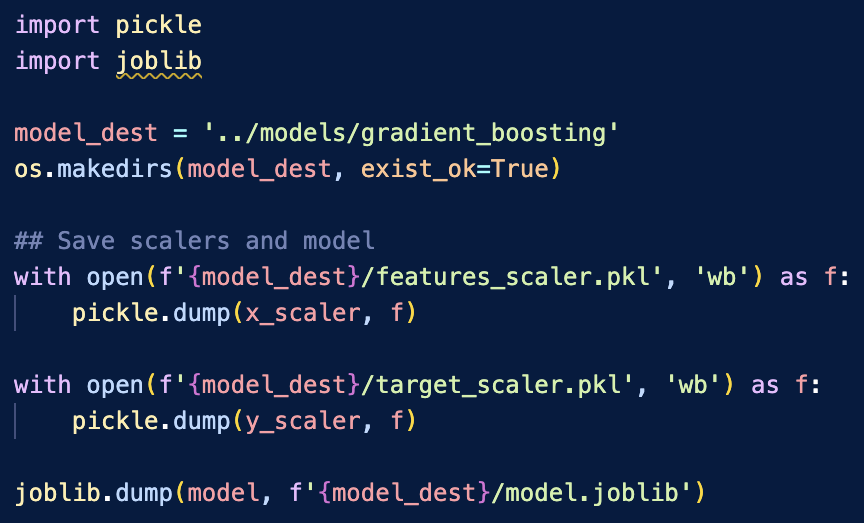
A computer screen shot of text

Description automatically generated

*Figure 7 - Training full dataset*

We overfitted the model for cross-validation. However, it resulted in the lowest MSE error in both setups.

The trained gradient boosting regression model was formatted using the joblib library so that it can be stored without loss of information and better reconstruction during deployment and testing. We will discuss this in the next section. Aligning with the model, we also saved the robust scalers of features and targets with the pickle library. The code is shown below.



*Figure 8 - Saving results*

# **Deployment Preparation**

Until now, we have trained the model, which has resulted in the lowest error between the ground truth and prediction. In this part of the paper, we will discuss utilizing the model with an interface to make predictions about using inputs. We will select the deployment tool and then the code to create endpoints.

**3.1. Selection of Deployment Tool/Platform**

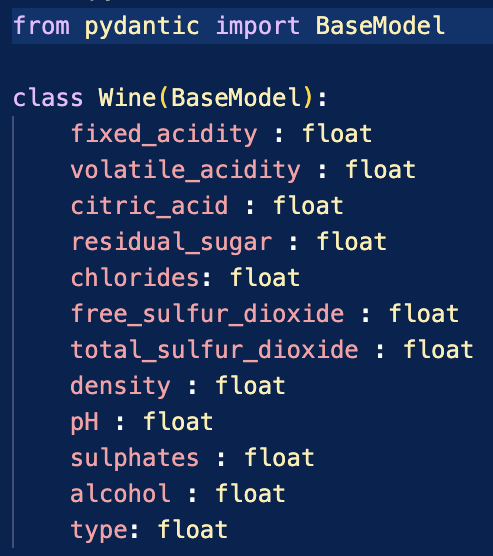
Many tools and platforms are available for users to deploy an ML model. Some popular tools and platforms are Flask and FastAPI for local deployment, AWS Lambda, Google Cloud Run, and Heroku for cloud deployment. In this project, we selected FastAPI because it is accessible and scalable. The tool is built on the Pydantic Library and leverages its speed and efficiency. Other tools mentioned above have their advantages. Google Cloud run and AWS lambda are more suitable for production deployment.

**3.2. Web/API Endpoint Development**

In this section, we will discuss the API endpoint development of FastAPI. We will begin with creating a data model and post API endpoint. The following section will discuss the deployment process, testing method, and results.

### **Data model with Pydantic**

The data model built on Pydantic is shown below. The model aims to define an input structure for the user to follow while making a prediction request.



*Figure 9 - Code for Wine data model (src/wine.py)*

The base model wrapper of the Pydantic library enables us to assign values to the required variable from a dictionary. We will discuss how the class is utilized next and more thoroughly while discussing the test method in the next section.

### **API Endpoints**

We have two endpoints in the deployment module: a static message and a prediction. The static message could be of more use in the present case. The predict function extends FastAPI’s post method and is of greater interest.

We will explain the module (src/deploy.py) functionality and process flow here. Initially, we import the required libraries, including uvicorn, for local server implementation, joblib, and pickle to load and reconstruct the trained gradient boosting model and robust scalers (saved in section 2.2). We imported FastAPI for deployment with endpoints. Lastly, import the Wine data model class from the wine.py module.

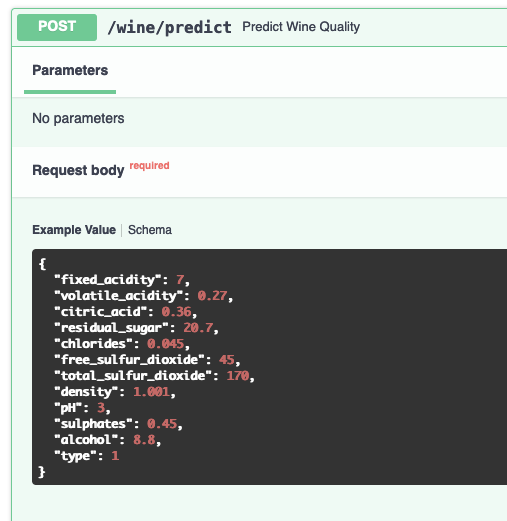
Next, we instantiate FastAPI for deployment, load model, and robust scaler. We will use them in the predict function. The magic of Pydantic can be observed here. While the Pythonic framework expects to be a data class, the FastAPI framework implements a dictionary.

A screen shot of a computer code

Description automatically generated

*Figure 10 - Copy for FastAPI endpoints (src/deploy.py)*

The first few lines of the predict\_wine\_quality function begin with converting the data model into a dictionary and extracting individual values with the respective key pair. Then, the values are stacked in a list and passed into features\_scaler to transform into a space in which the gradient boosting model is trained. We pass the transformed inputs to the model to make a prediction. The estimated value in the standardized form and by applying inverse transformation using target\_scaler, we get the quality in the original range. Finally, the prediction is saved in a dictionary, the format expected by the FastAPI, and the user will see this value on the deployed framework.



*Figure 11 - Input schema for the post endpoint taking advantage of the Pydantic data model functionality*

# **Deployment and Testing**

So far, we have done all the heavy lifting, and deploying the endpoints and testing with some examples is relatively straightforward. We begin by introducing the framework's deployment process and testing it with an example.

**4.1. Deployment Process**

The predictive model deployment process is shown in the deploy\_test.ipynb notebook. We can run the same with the terminal. However, it is slightly earlier in the notebook (from our experience.) The process contains two steps:

1. Change the directory where the deploy.py module exists. In our case, the module is in the src folder.
2. Run the module on top of the local server created by uvicorn

**4.2. Testing Methodology**

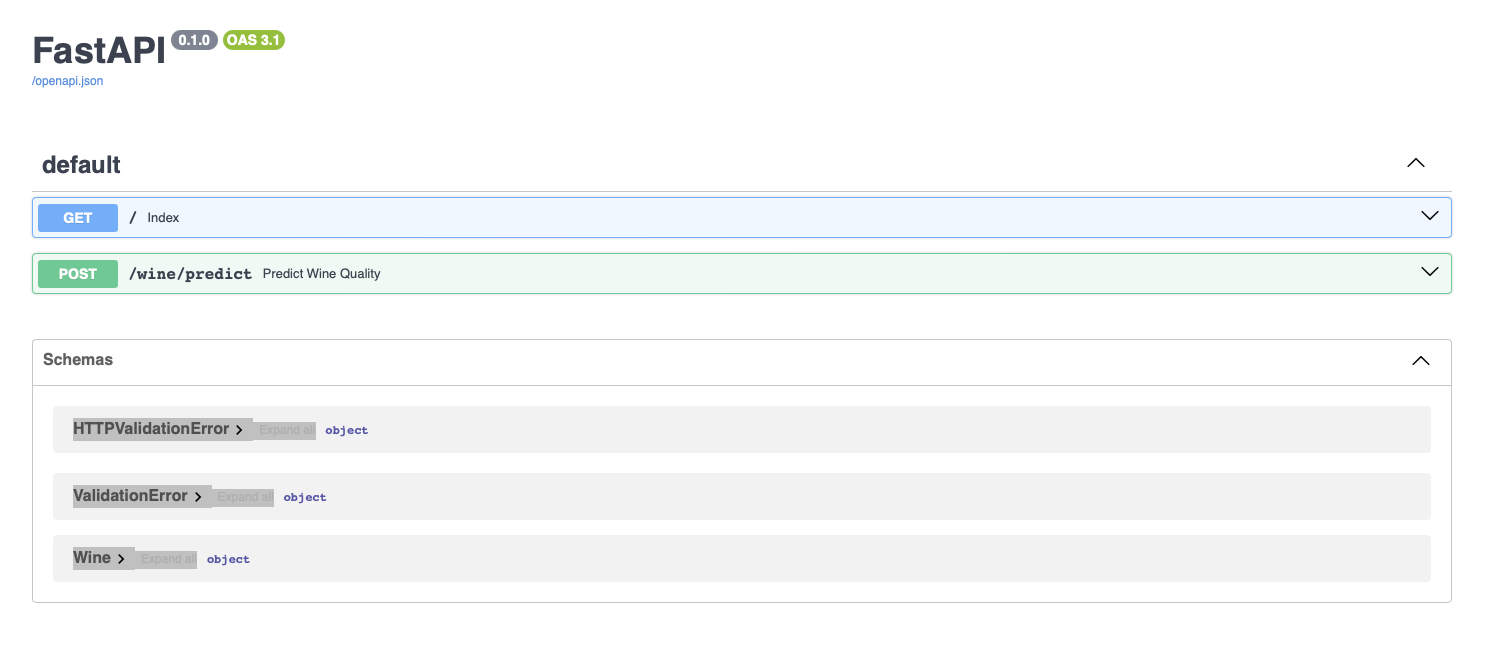
The uvicorn-created server local is <http://127.0.0.1:8000/>, and opening it in a browser results in a message from the index function ({"message": "Wine quality predictor API"}). To use the FastAPI endpoint services, we have to open <http://127.0.0.1:8000/docs>. We will discuss the usage and testing of the model with an example in the next section.

A screenshot of a computer program

Description automatically generated

*Figure 12 - Code to deploy from running the deploy.py (src/deploy.py) module on the local server with uvicorn*

We have deployed the FastAPI endpoints and will test them with an example. The example we will use is shown earlier in the figure 10.



*Figure 13 - FastAPI endpoints deployed on the local Uvicorn server*

## **4.3. Test Results and Observations**

Testing with an example is relatively straightforward. Upon selecting POST, we have to click the "Try It Out" button on the right to make the inputs editable. Replace all the zero (default) values with those shown in the example (figure 10), then click the Execute button. The post request should run with no errors and return the result - the predicted quality of wine is 5.64 (actual 6), as shown below.

A screenshot of a computer

Description automatically generated

*Figure 14 - Test Result with predicted wine quality is 5.64, while the ground truth is 6.*

The key observation in the test setup is manually inputting the attributes in the wine data class. For experimental purposes, the setup is acceptable; however, in the real world, we either feed the data from a database or feed the data in a local directory. Overall, the setup demonstrates the deployment of the ML model and testing with an example well.

# **Reflection**

In this section, we will examine the challenges of the setup, including the ML pipeline, and the solutions to overcome them.

**5.1. Challenges and Solutions**

The primary challenge of the model deployment is the simplicity and the limited practical use of the framework. Calling the server link (<http://127.0.0.1:8000/docs>) for every new prediction is not efficient. As discussed in the previous section, updating the inputs manually or programmatically, one at a time, is not helpful and needs to be more time-consuming. The solution to the first challenge is to resort to the AWS Lambda or Google Cloud run, where we can create multiple instances to manage user requests in batches (load factor) with more reliability. For the second solution, we have to work ground-up and make the data input in a form that can be processed batch-wise from an external source. This way, we can process multiple predictions seamlessly.

## **5.2. Lessons Learned**

We learned to build an ML pipeline to create a wine quality prediction model in the project. Then we created a deployment framework with FastAPI and ran it on the local server created by the uvicorn library. We tested our model and deployment with an example, and the result was very close to ground truth and ran with no error. We also discussed the challenges of the framework and how to overcome them if we had to deploy it in the real world.

# **Conclusion**

We selected wine quality data from UCI to prepare a predictive model. Among several regression models, the gradient-boosting regression model was selected for its superior performance. The model's performance was measured with mean square error and achieved a mean of 0.38 on 5-fold cross-validation and 0.36 on the test dataset. Later, the model was deployed and tested on a local server using the uvicorn package and FastAPI endpoints. We successfully tested the framework with a sample input. Ultimately, we discussed the limitations of the chosen deployment tool and package. We suggested utilizing better frameworks such as AWS Lamba and Google Cloud run, where prediction can be managed in a batch, with multiple instances in parallel.

# **References**

* <https://hdsr.mitpress.mit.edu/pub/2fu65ujf/release/2>
* UCI Wine Quality dataset: <https://archive.ics.uci.edu/dataset/186/wine+quality>
* Uvicorn: <https://www.uvicorn.org/>
* FastAPI: <https://fastapi.tiangolo.com/>