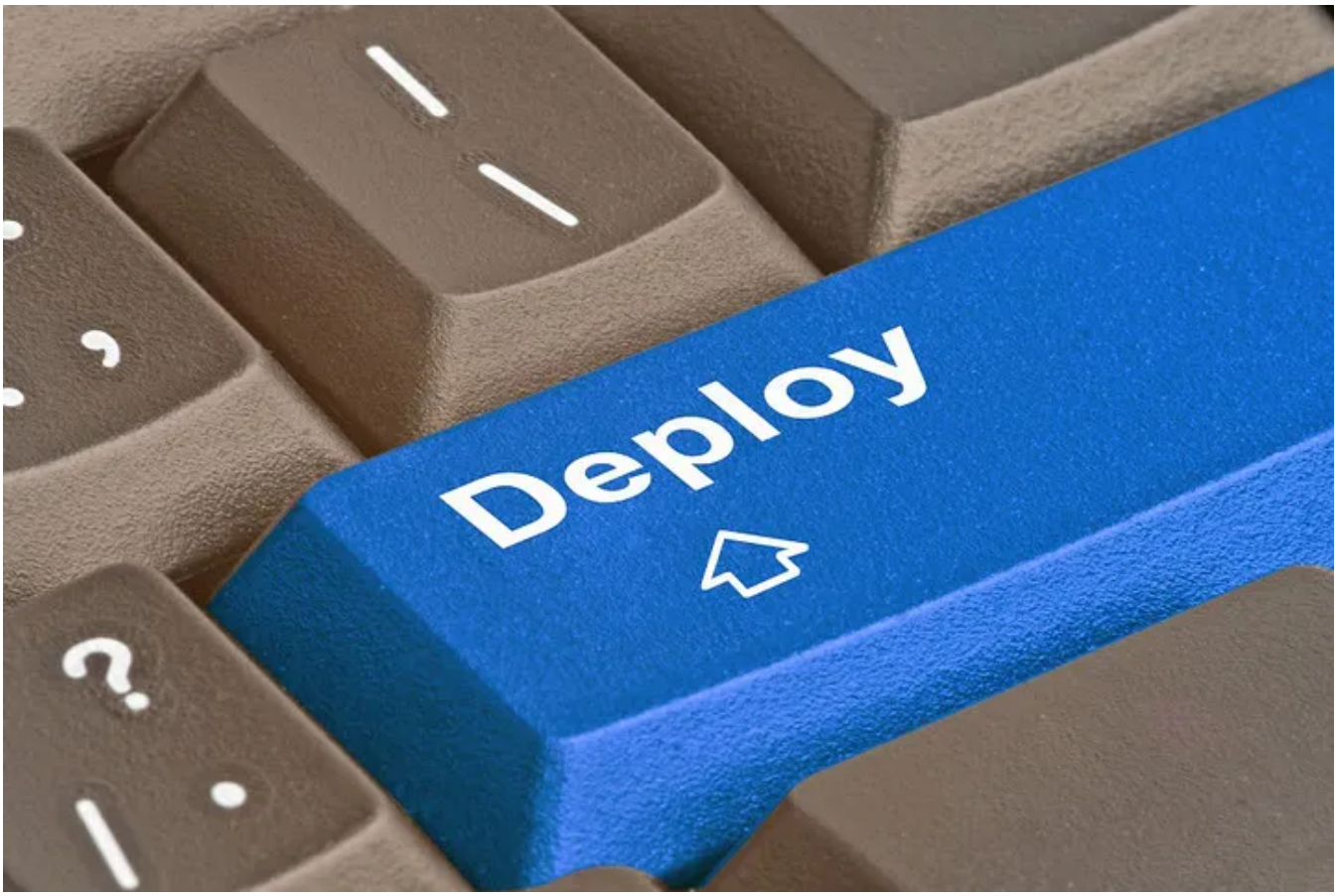
How I Deployed a Machine Learning Model for the First Time

Going beyond Jupyter Notebooks 🚀

Luís Fernando Torres · Follow

13 min read · Just now



Introduction

For as long as I've started with machine learning, Jupyter Notebooks have been my most loyal sidekick. From data preprocessing to model training, fine-tuning, and testing, Jupyter Notebooks have been there at every step to support me. However, I

always knew that there is an entire world beyond these digital pages — a world of ***deployment*** and ***application***.

Taking the leap from training a model to actually deploying it might seem intimidating. However, it's a critical step that transforms a data science project from a theoretical experiment into a practical, real-world application. And I knew I had to take that extra step!

In this article, we will embark on my journey of building a classification model for a Kaggle competition. We start from a typical EDA and pipeline building until reaching new-unexplored territory — at least for me — bringing my machine learning model to life, enabling it to interact and offer insights to users globally.

Let's brace ourselves as we step outside the comfort of our Jupyter Notebooks, because we're about to go on a deployment journey. Grab your coding cap, fasten your seatbelt, and let's get ready for a thrilling ride into the world of machine learning deployment!

Playground Series Episode 5 Season 3: Ordinal Regression with a Tabular Wine Quality Dataset

Our journey starts with the fifth episode of Kaggle's Playground Series' third season. This series, promoted by Kaggle, presents a variety of machine learning challenges, inviting users to boost their skills in data analysis, feature engineering, data cleansing, and machine learning pipeline construction.

Episode five, ***Ordinal Regression with a Tabular Wine Quality Dataset***, invited *Kagglers* to analyze a synthetically-generated dataset based on the ***Wine Quality Dataset***. This dataset consisted of 12 attributes — including the target variable — which tell us more about each wine, like how acidic it is, its pH, the amount of chlorides, alcohol levels, and so on.

The evaluation metric for this competition is the *Quadratic-Weighted Kappa,* given by the following equation:

$$\kappa = 1 - \frac{\sum_{i,j} w_{ij} O_{ij}}{\sum_{i,j} w_{ij} E_{ij}}$$

Quadratic-Weighted Kappa formula

Where:

. O_{ij} is the actual confusion matrix.

. E_{ij} is the expected confusion matrix under randomness.

. W_{ij} is the weighted matrix, which can be calculated as $(i - j)^2$, where i and j are the ratings.

A score equal to **1** suggests a perfect agreement between the raters, while a score equal to **0** indicates that the agreement is no better than what would be expected by chance.

The file submitted for evaluation had to be a *.csv* file consisting of columns **ID** and **quality**, where quality referred to the predictions of the final model for wines in the test dataset. **Quality** was a value, ranging from *3* to *8*, where 8 implied the best quality and 3 implied the worst quality.

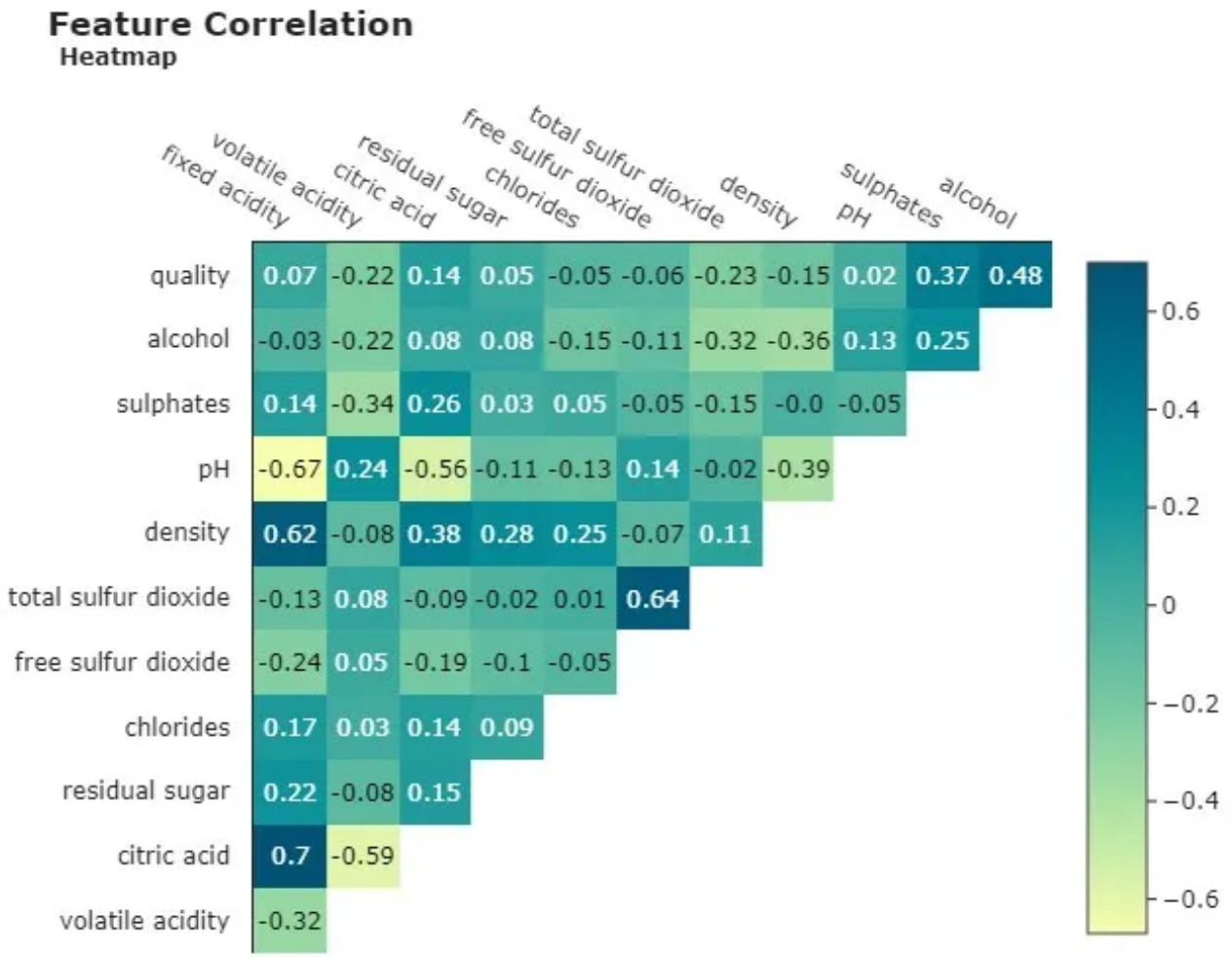
Exploratory Data Analysis

To enhance my exploratory analysis, I turned to **Plotly** — hands down, my go-to library for data visualization in Python. I kicked things off by taking a peek at some key descriptive statistics. Then, I dived straight into checking out the correlation heatmap.

Descriptive Statistics of the Dataframe
(Mean, Standard Deviation, 25%, Median, and 75%)

Feature Name	Mean	Standard Deviation	25%	Median	75%
fixed acidity	8.37	1.71	7.20	7.95	9.20
volatile acidity	0.53	0.17	0.39	0.52	0.64
citric acid	0.27	0.19	0.09	0.25	0.42
residual sugar	2.40	0.86	1.90	2.20	2.60
chlorides	0.08	0.02	0.07	0.08	0.09
free sulfur dioxide	16.96	10.01	8.00	16.00	24.00
total sulfur dioxide	49.24	32.96	22.00	44.00	65.00
density	1.00	0.00	1.00	1.00	1.00
pH	3.31	0.14	3.20	3.31	3.39
sulphates	0.64	0.14	0.55	0.61	0.72
alcohol	10.41	1.03	9.50	10.10	11.00
quality	5.72	0.85	5.00	6.00	6.00

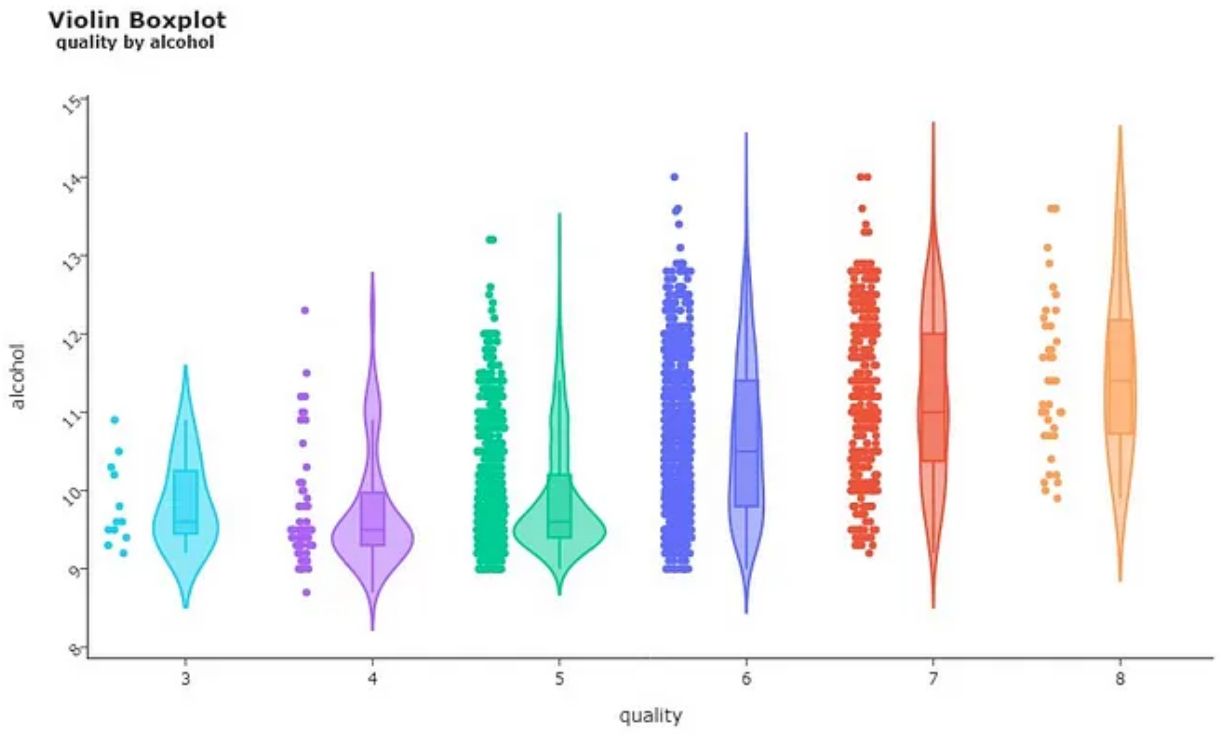
Descriptive Statistics on the Training Data



Pearson's R Correlation Across Features

The first insight I extracted from the **descriptive statistics** table was the difference in scales across the features. Something that we would deal with later on.

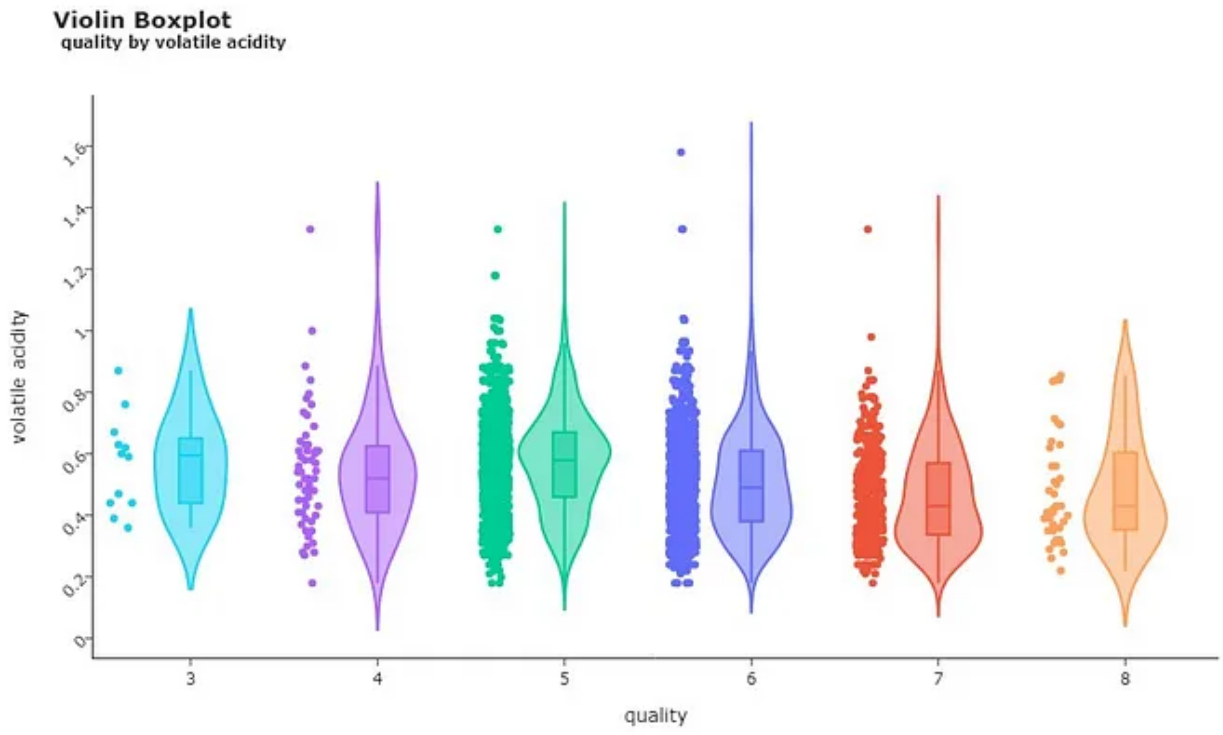
By analyzing the **feature correlation map**, I couldn't help but notice that **alcohol** was the most positively correlated feature with the target variable **quality**. This can be confirmed by the following violin boxplot.



Violin Boxplot Between Alcohol (**Y**) and Quality (**X**)

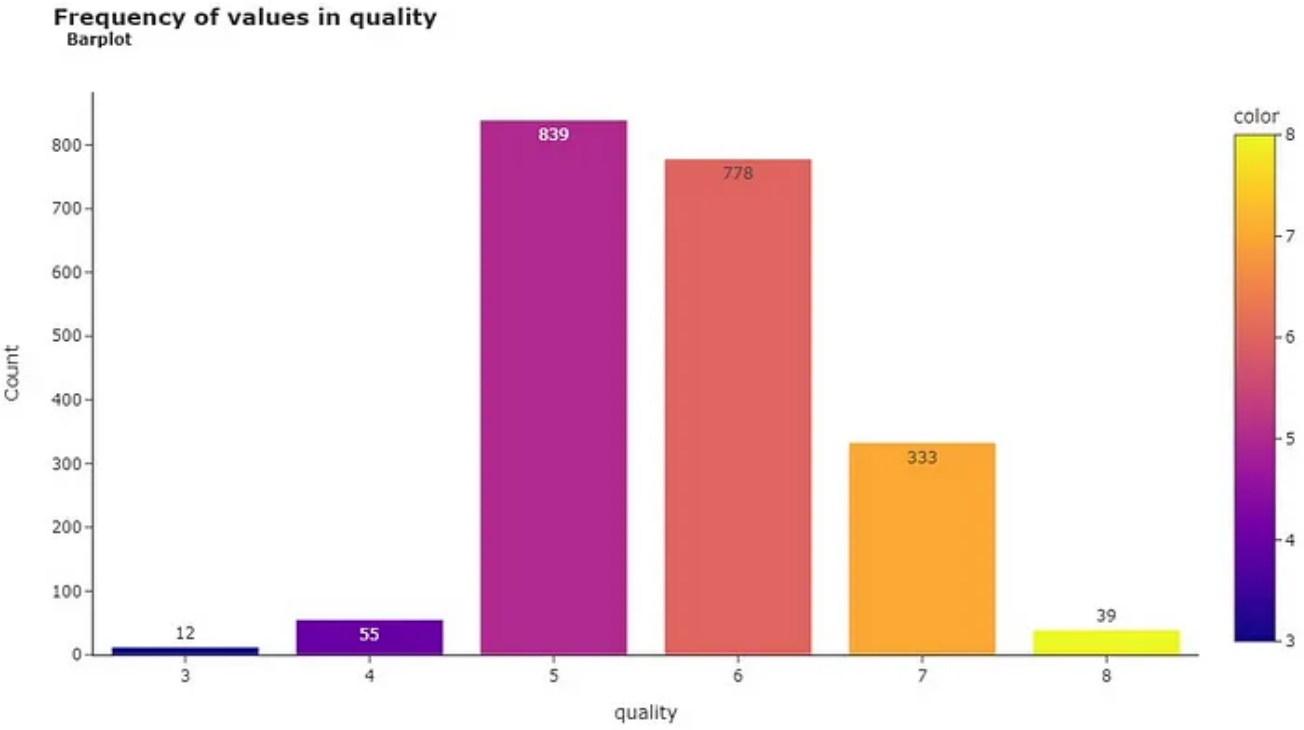
You can easily spot the trend in the image above. Starting from quality 5, we see a noticeable upward trend in the median value of the alcohol level. This pattern hints that wines with higher alcohol levels tend to also be of a higher quality. It's a clue that this feature might be pretty good at predicting wine quality.

An inverse pattern can be spotted when analyzing the relationship between **quality** and **volatile acidity**. The higher the quality of wines, the lower the median value of volatile acidity.



Violin Boxplot Between Volatile Acidity (**Y**) and Quality (**X**)

And speaking of wine quality levels, it's also helpful to check out how these qualities are distributed across the wines in our dataset.

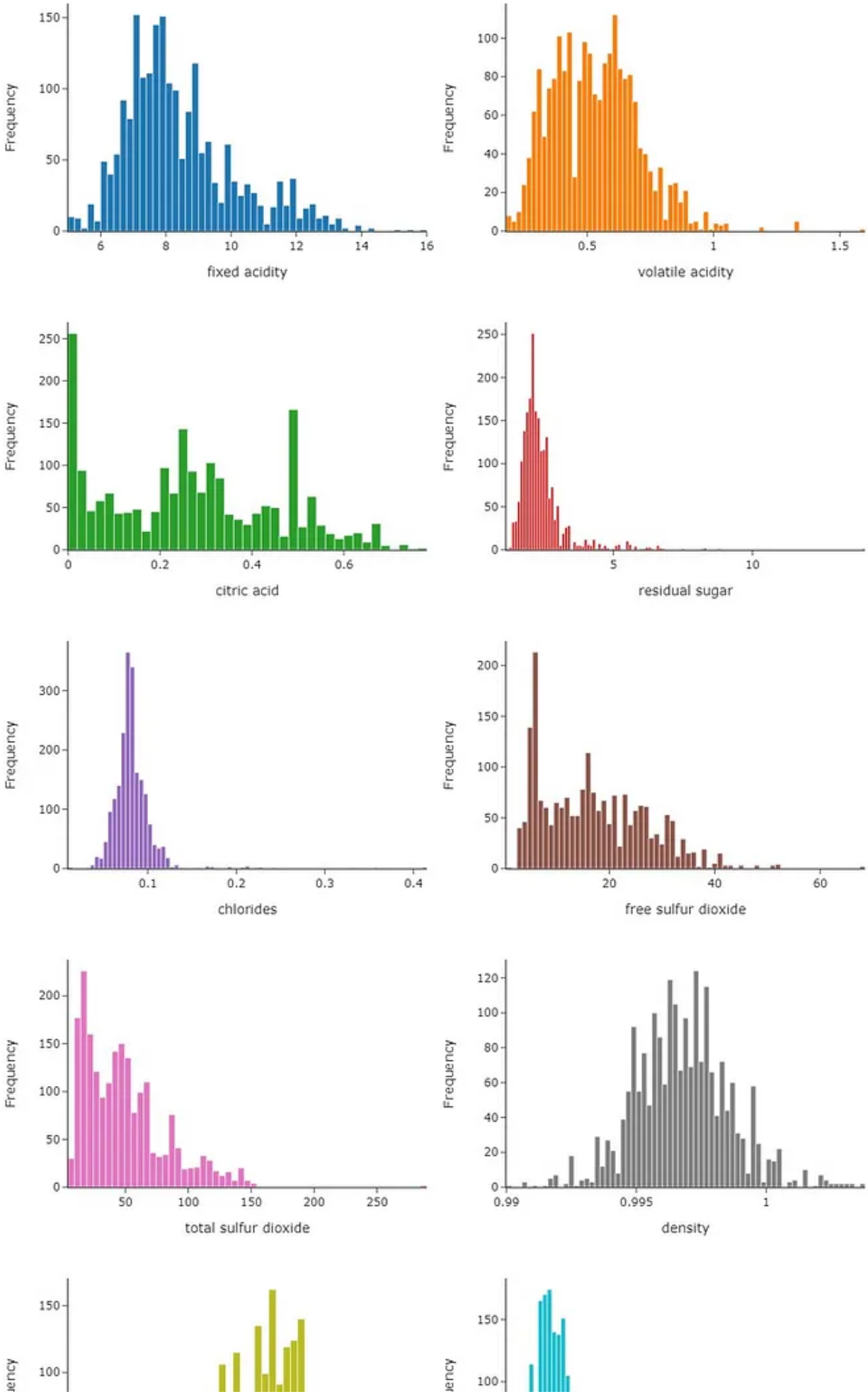


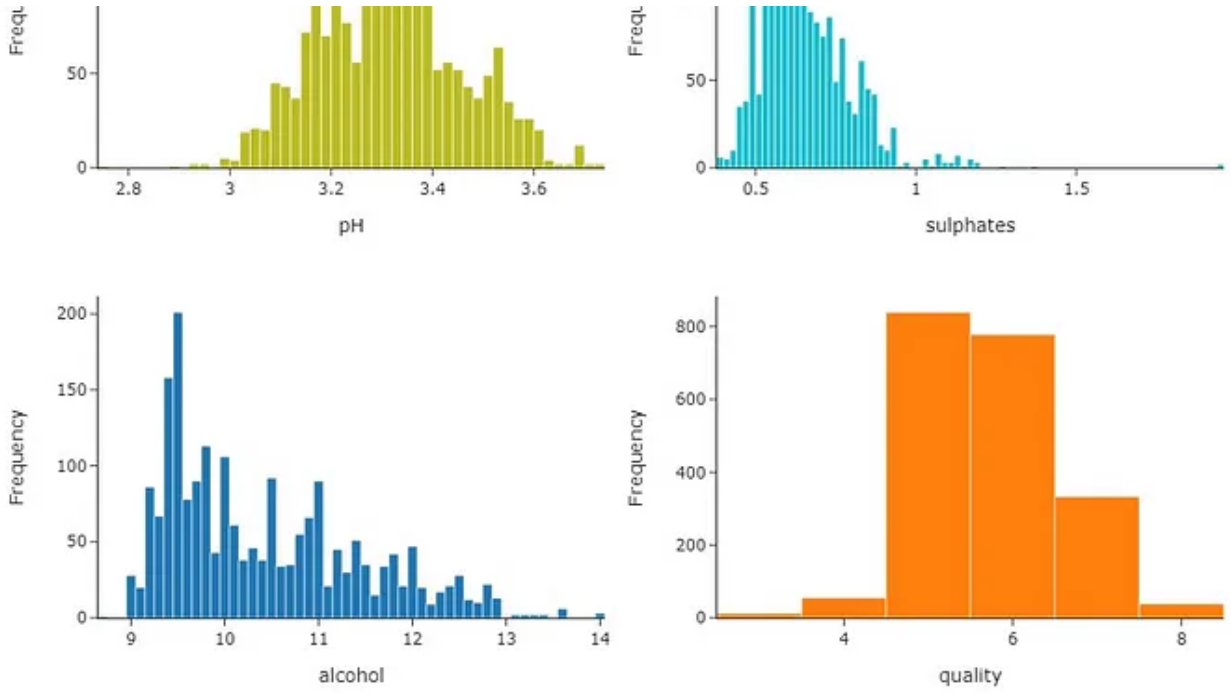
Wine quality across the dataset

Most wines scored 5 or 6, suggesting they fall in the 'average' category. Only a small portion of 12 wines received a low rating of 3, while 39 of them reached the 'golden' status with a score of 8. This kind of distribution of classes may introduce ***bias*** to our model, which makes it more susceptible to classifying wines as either 5 or 6, since that's the range most wines fall into.

And since we're talking about ***distribution***, why not take a look at how all the other features are distributed across the data?

Histogram Matrix
Continuous Features





Histogram Matrix

The histogram matrix above gives another valuable insight! Our features are **not** normally-distributed, meaning that their histogram do not resemble a bell curve. This information is relevant, because we **could** have better performances by transforming feature distributions into a Gaussian distribution — i.e., a normal distribution.

That sums up the key takeaways from our exploratory analysis for now. But don't let the fun stop here! Be sure to check out these **beautiful** plots crafted with Plotly. For a deeper dive, click here to check my full notebook on Kaggle!

Preprocessing

Before we dive into the modeling phase, we need to tidy up our data! I kicked things off by crafting new features, aiming to boost the predictive power of our independent variables.

Remember when we dug into the correlations between features? These relationships can come in handy now that we are in the feature-creation stage. We can engineer new features from attributes that share a stronger correlation with the target variable — like alcohol, for instance.

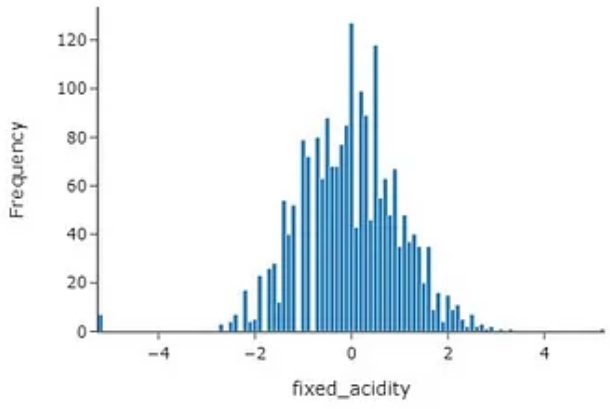
These are the following features I've added to our training dataset:

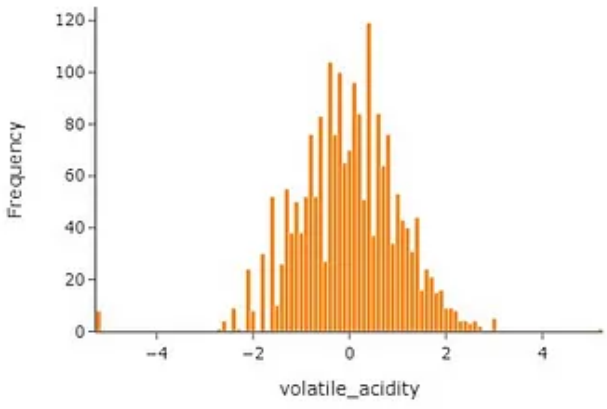
- **Total Acidity**: Fixed Acidity + Volatile Acidity + Citric Acid;

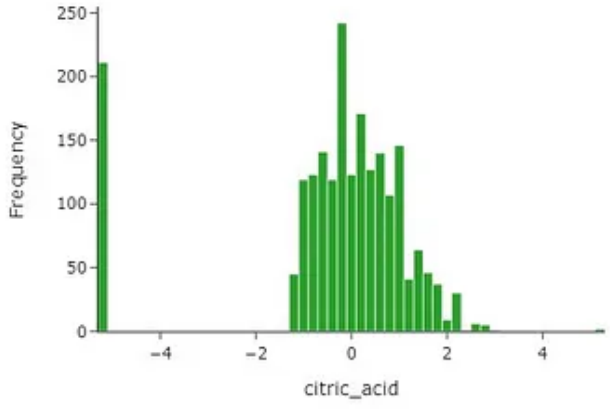
- **Acidity-to-pH ratio**: Total Acidity ÷ pH;
- **Free Sulfur Dioxide-to-Total Sulfur Dioxide ratio**: Free Sulfur Dioxide ÷ Total Sulfur Dioxide;
- **Alcohol-to-Acidity ratio**: Alcohol ÷ Total Acidity;
- **Residual Sugar-to-Citric Acid ratio**: Residual Sugar ÷ Citric Acid;
- **Total Acidity**: Fixed Acidity + Volatile Acidity + Citric Acid;
- **Alcohol-to-Density ratio**: Alcohol ÷ Density;
- **Total Alkalinity**: pH + Alcohol;
- **Total Minerals: C**hlorides + Sulfates + Residual Sugar.

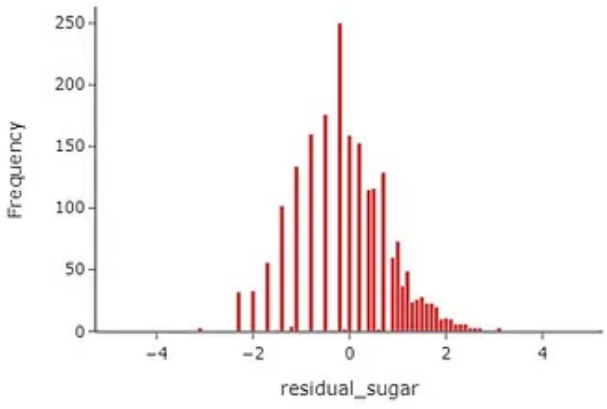
In addition to this, I used Scikit-learn's **QuantileTransformer**. This helped me mold the distribution of our independent features into something closer to the Gaussian distribution. As a result, we got that tidy bell curve shape we were aiming for, which you can spot in the following histogram matrix.

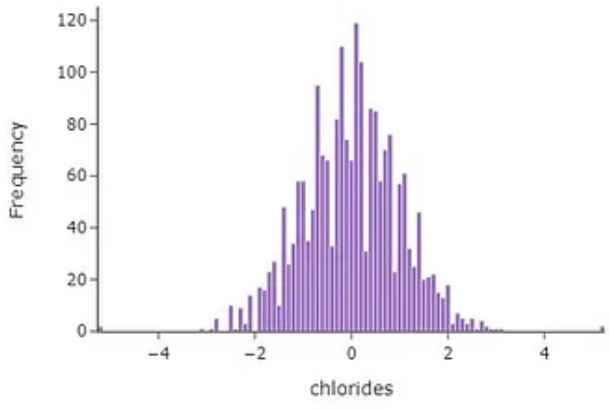
Histogram Matrix
Continuous Features

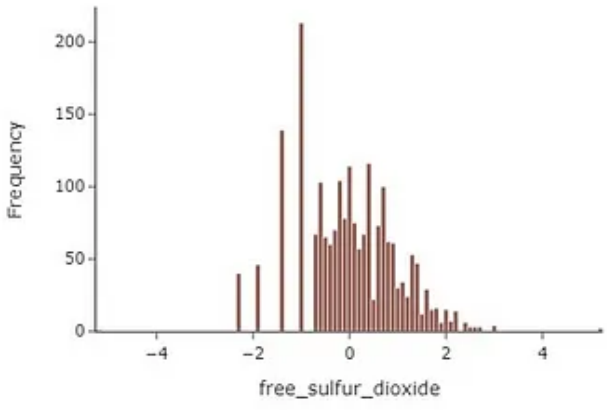


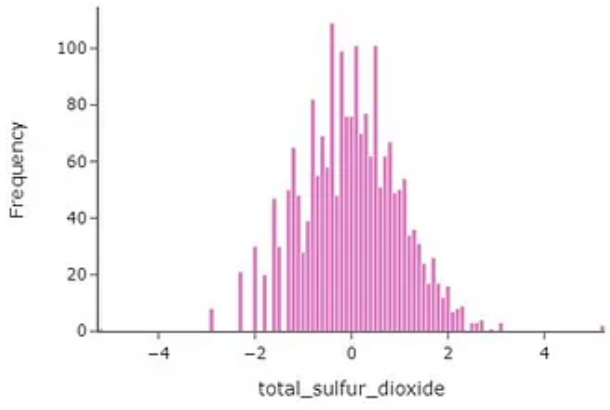


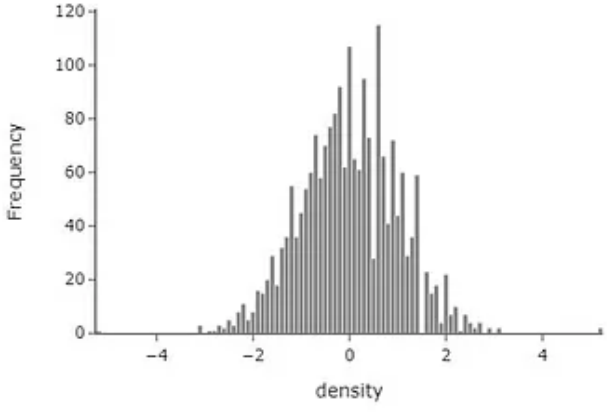


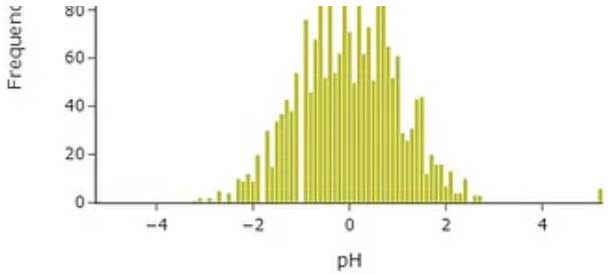


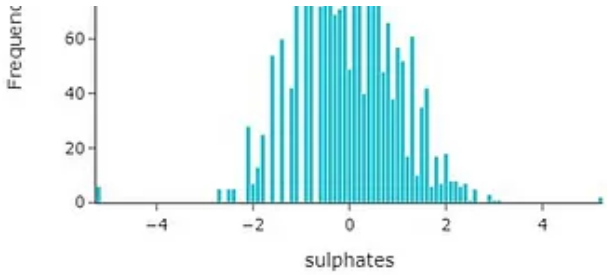


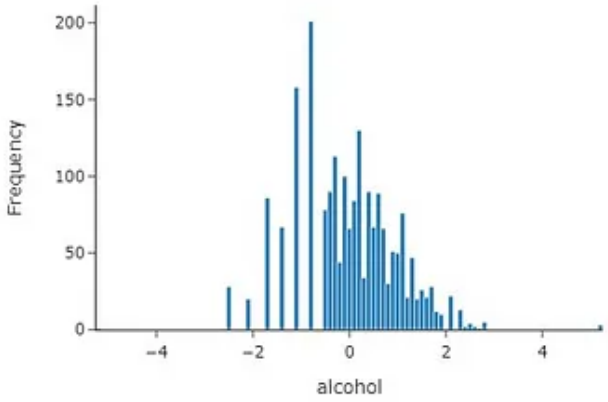


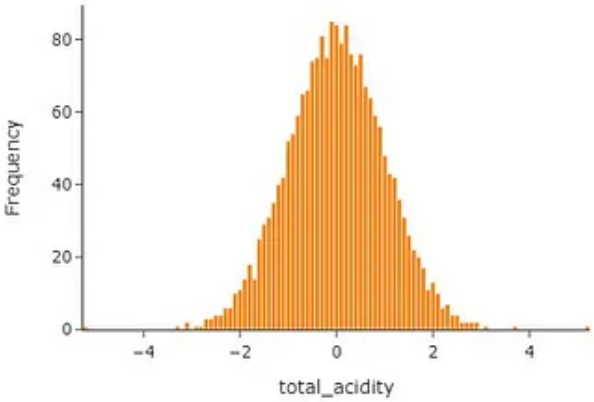


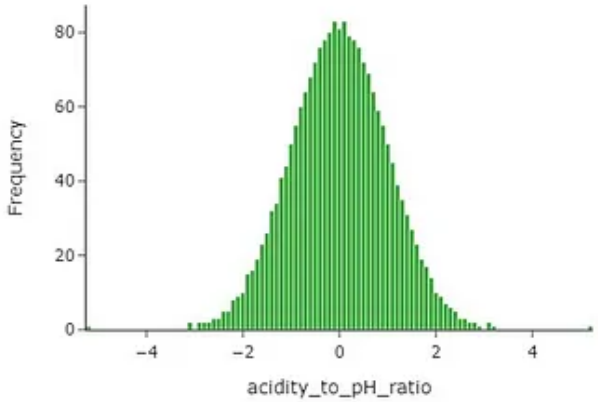


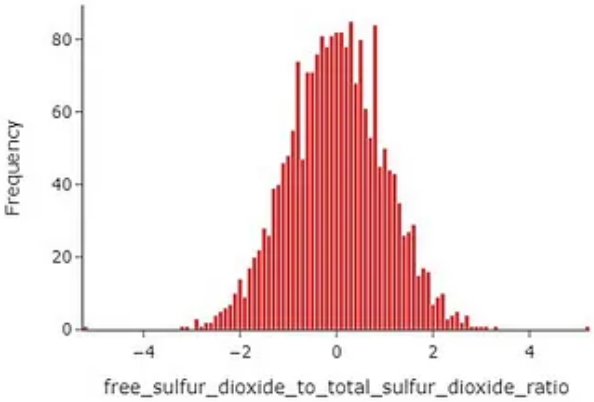

Frequenc
80
60
40
20
0
−4
−2
0
2
4
pH

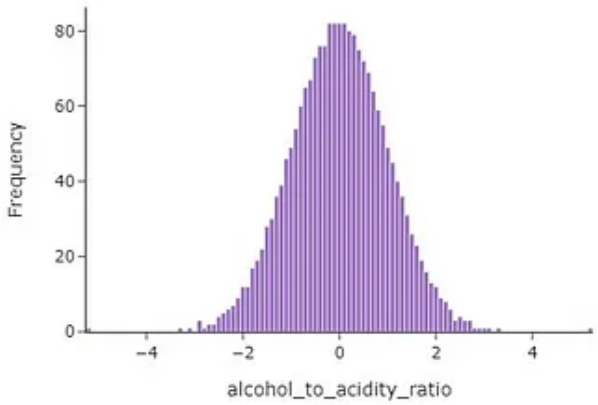

Frequenc
60
40
20
0
−4
−2
0
2
4
sulphates

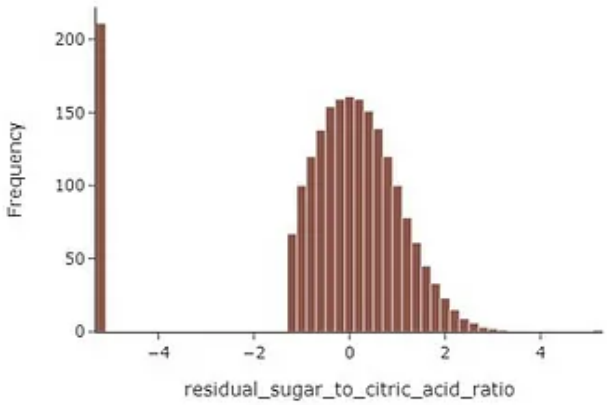

Frequency
200
150
100
50
0
−4
−2
0
2
4
alcohol

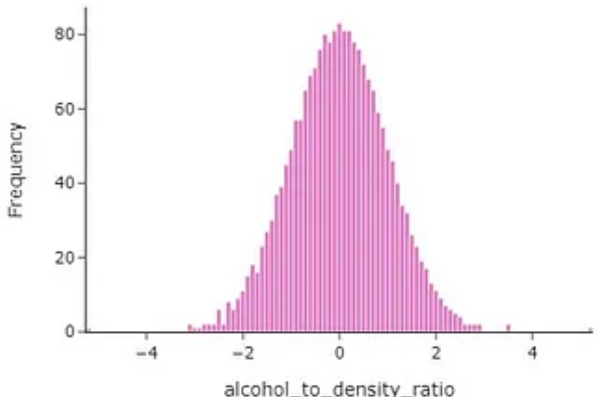

Frequency
80
60
40
20
0
−4
−2
0
2
4
total_acidity

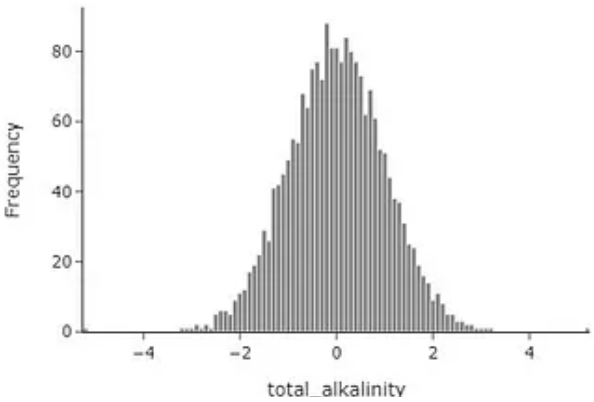

Frequency
80
60
40
20
0
−4
−2
0
2
4
acidity_to_pH_ratio

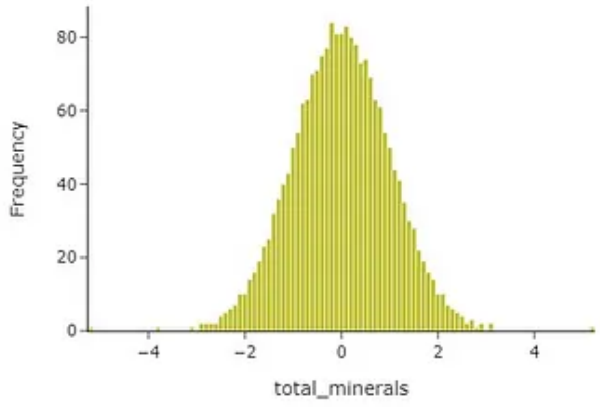

Frequency
80
60
40
20
0
−4
−2
0
2
4
free_sulfur_dioxide_to_total_sulfur_dioxide_ratio


Frequency
80
60
40
20
0
−4
−2
0
2
4
alcohol_to_acidity_ratio


Frequency
200
150
100
50
0
−4
−2
0
2
4
residual_sugar_to_citric_acid_ratio


Frequency
80
60
40
20
0
−4
−2
0
2
4
alcohol_to_density_ratio


Frequency
80
60
40
20
0
−4
−2
0
2
4
total_alkalinity



Transformed distributions. You can also see the new features.

Furthermore, I've used **StandardScaler** to tweak the mean and standard deviation values for each feature. Thanks to this tool, we have a mean (μ) of 0 and a standard deviation (σ) of 1, which you can see in the table below.

Descriptive Statistics of the Dataframe
(Mean, Standard Deviation, 25%, Median, and 75%)

Feature Name	Mean	Standard Deviation	25%	Median	75%
fixed_acidity	-0.00	1.00	-0.60	0.00	0.67
volatile_acidity	-0.00	1.00	-0.65	0.02	0.67
citric_acid	0.00	1.00	-0.18	0.19	0.57
residual_sugar	0.00	1.00	-0.78	0.01	0.71
chlorides	0.00	1.00	-0.66	-0.03	0.68
free_sulfur_dioxide	-0.00	1.00	-0.64	0.01	0.65
total_sulfur_dioxide	0.00	1.00	-0.67	0.00	0.66
density	-0.00	1.00	-0.65	0.00	0.66
pH	0.00	1.00	-0.71	0.03	0.64
sulphates	-0.00	1.00	-0.60	-0.01	0.69
alcohol	0.00	1.00	-0.75	-0.01	0.60
total_acidity	-0.00	1.00	-0.67	0.00	0.67
acidity_to_pH_ratio	0.00	1.00	-0.67	0.00	0.67
ioxide_to_total_sulfur	0.00	1.00	-0.67	0.00	0.67
lcohol_to_acidity_rati	0.00	1.00	-0.67	-0.00	0.67
al_sugar_to_citric_aci	-0.00	1.00	-0.18	0.19	0.56
lcohol_to_density_rati	0.00	1.00	-0.66	-0.00	0.67
total_alkalinity	-0.00	1.00	-0.67	0.00	0.67
total_minerals	-0.00	1.00	-0.67	0.00	0.67

Descriptive Statistics of Preprocessed Data

With all these data transformations completed, it was time to jump into feature selection, to identify the most impactful features for predicting wine quality. This

is where **RFECV**, a handy tool from Scikit-learn, came into play.

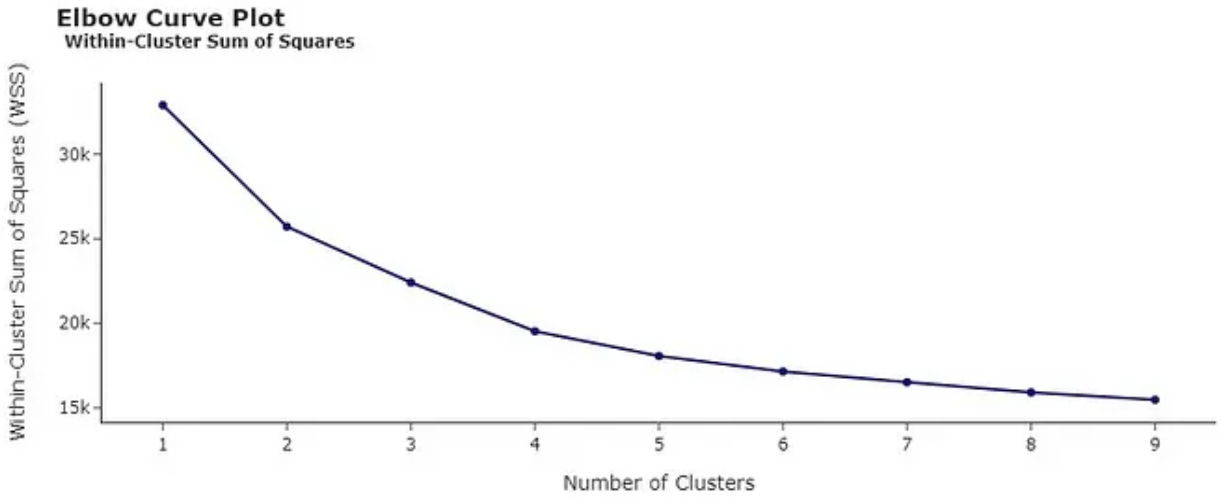
RFECV stands for ***Recursive Feature Elimination with Cross-Validation***. Recursive Feature Elimination is a technique that works by fitting an estimator and removing the weakest features — those with the smallest absolute coefficients —, and then repeating this process until all the least important features are eliminated. By using RFECV, we can have a set of features that bring genuine value to predictions, which in turn boosts accuracy and efficiency.

After running RFECV, we have a list of features that were selected as the most relevant ones for **quality** prediction.

```
volatile_acidity
citric_acid
chlorides
total_sulfur_dioxide
density
pH
sulphates
alcohol
total_acidity
acidity_to_pH_ratio
free_sulfur_dioxide_to_total_sulfur_dioxide_ratio
alcohol_to_acidity_ratio
residual_sugar_to_citric_acid_ratio
alcohol_to_density_ratio
total_alkalinity
total_minerals
```

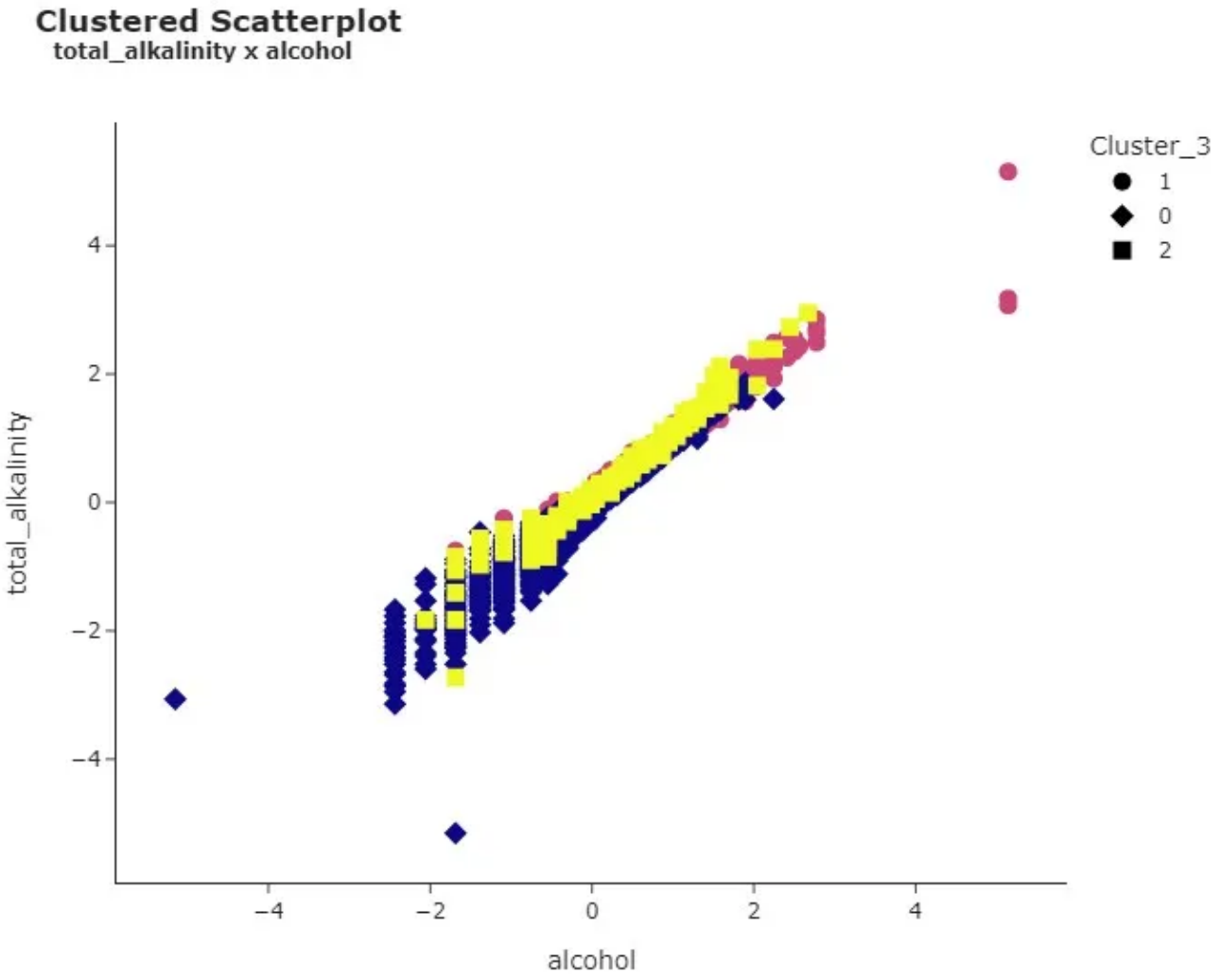
The final step in our preprocessing session involved ***clustering***. Here, I employed **KMeans** to bundle the data into distinct groups. These groups represent underlying patterns or relationships that may not be immediately apparent from, say, a Correlation Plot. It's a clever way of unveiling and capitalizing on hidden structures within our dataset.

By analyzing an **Elbow Curve Plot**, it was decided that the optimal number of clusters was $K = 3$, which grouped the data into three different groups.



Elbow Curve Plot

Furthermore, I created a scatterplot to illustrate the relationship between total alkalinity and alcohol, which provided more insight into the makeup of each cluster. For example, Cluster 0 comprises wines with lower levels of both alcohol and total alkalinity. On the other hand, Cluster 1 is dominated by wines leaning towards higher values for these same two features.



Modeling

We've finally made it to the modeling stage! We'll start by using Scikit-learn's **Pipeline**. This useful tool allows us to specify a series of functions and classes that will carry out all the preprocessing steps we've worked through earlier. This is done to ensure that each fold during cross-validation, as well as any data inputted into the app hosting our model, undergoes all the necessary preprocessing. It's an easy and simple way to keep the process consistent.

We start by preparing all the steps that will then be performed in the pipeline.

```
# Cleaning and creating features
def feat_eng(df):
    df.columns = df.columns.str.replace(' ', '_')
    df['total_acidity'] = df['fixed_acidity'] + df['volatile_acidity'] + df['c
    df['acidity_to_pH_ratio'] = df['total_acidity'] / df['pH']
    df['free_sulfur_dioxide_to_total_sulfur_dioxide_ratio'] = df['free_sulfur_
```

```
    df['alcohol_to_acidity_ratio'] = df['alcohol'] / df['total_acidity']
    df['residual_sugar_to_citric_acid_ratio'] = df['residual_sugar'] / df['cit
    df['alcohol_to_density_ratio'] = df['alcohol'] / df['density']
    df['total_alkalinity'] = df['pH'] + df['alcohol']
    df['total_minerals'] = df['chlorides'] + df['sulphates'] + df['residual_su

    df = df.replace([np.inf, -np.inf], 0)
    df = df.dropna()

    df = df[selected_features]

    return df
```

```
# Applying QuantileTransformer to change the distribution to a gaussian-like d
class CustomQuantileTransformer(BaseEstimator, TransformerMixin):
    def __init__(self, random_state=None):
        self.random_state = random_state
        self.quantile_transformer = QuantileTransformer(output_distribution='n

    def fit(self, X_train, y=None):
        self.quantile_transformer.fit(X_train)
        return self

    def transform(self, X):
        X_transformed = self.quantile_transformer.transform(X)
        X = pd.DataFrame(X_transformed, columns=X.columns)
        return X
```

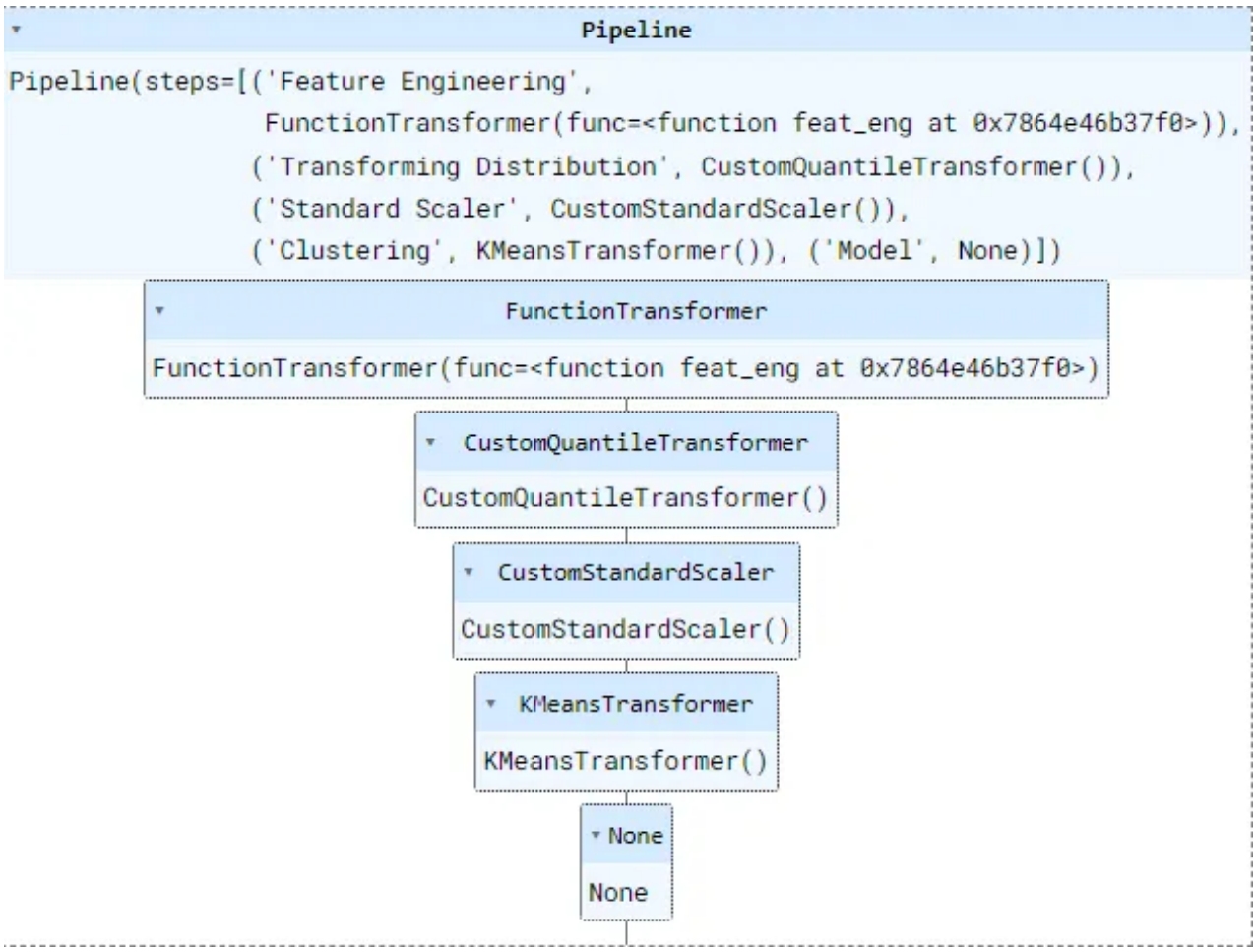
```
# Applying StandardScaler to bring every feature to the same scale
class CustomStandardScaler(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.scaler = StandardScaler()

    def fit(self, X_train, y=None):
        self.scaler.fit(X_train)
        return self

    def transform(self, X):
        X_transformed = self.scaler.transform(X)
        X = pd.DataFrame(X_transformed, columns=X.columns)
        return X
```

```
# Applying KMeans clustering with n_clusters = 3

class KMeansTransformer(BaseEstimator, TransformerMixin):

    def __init__(self, n_clusters=3, random_state=seed):
        self.n_clusters = n_clusters
        self.random_state = random_state
        self.kmeans = KMeans(n_clusters=self.n_clusters, random_state=self.rar

    def fit(self, X_train, y=None):
        self.kmeans.fit(X_train)
        return self

    def transform(self, X):
        X_clustered = pd.DataFrame(X.copy())
        cluster_labels = self.kmeans.predict(X)
        X_clustered['Cluster'] = cluster_labels
        return X_clustered
```

After that, all we need to do is use sklearn's pipeline class to create a list of tuples containing all the steps above.

```
pipeline = Pipeline([
    ('Feature Engineering', FunctionTransformer(feat_eng)),
    ('Transforming Distribution', CustomQuantileTransformer()),
    ('Standard Scaler', CustomStandardScaler()),
    ('Clustering', KMeansTransformer()),
    ('Model', None)
])
```

This is what our pipeline looks like for now:



Initial Pipeline

You can see how each preprocessing step follows a specific sequence. We start it off by creating and selecting the features, which are then transformed, scaled, and clustered. You might also notice that the final section, ***Model***, is currently empty. This is where we are going to add our machine learning model to the pipeline.

I performed cross-validation on many models, fine-tuned them, and tried different ensemble methods such as **VotingClassifier** and **StackingClassifier**. To avoid having an overly-extensive article, let's jump right into the final model used for this pipeline, but I reinforce that you can read the whole process here.

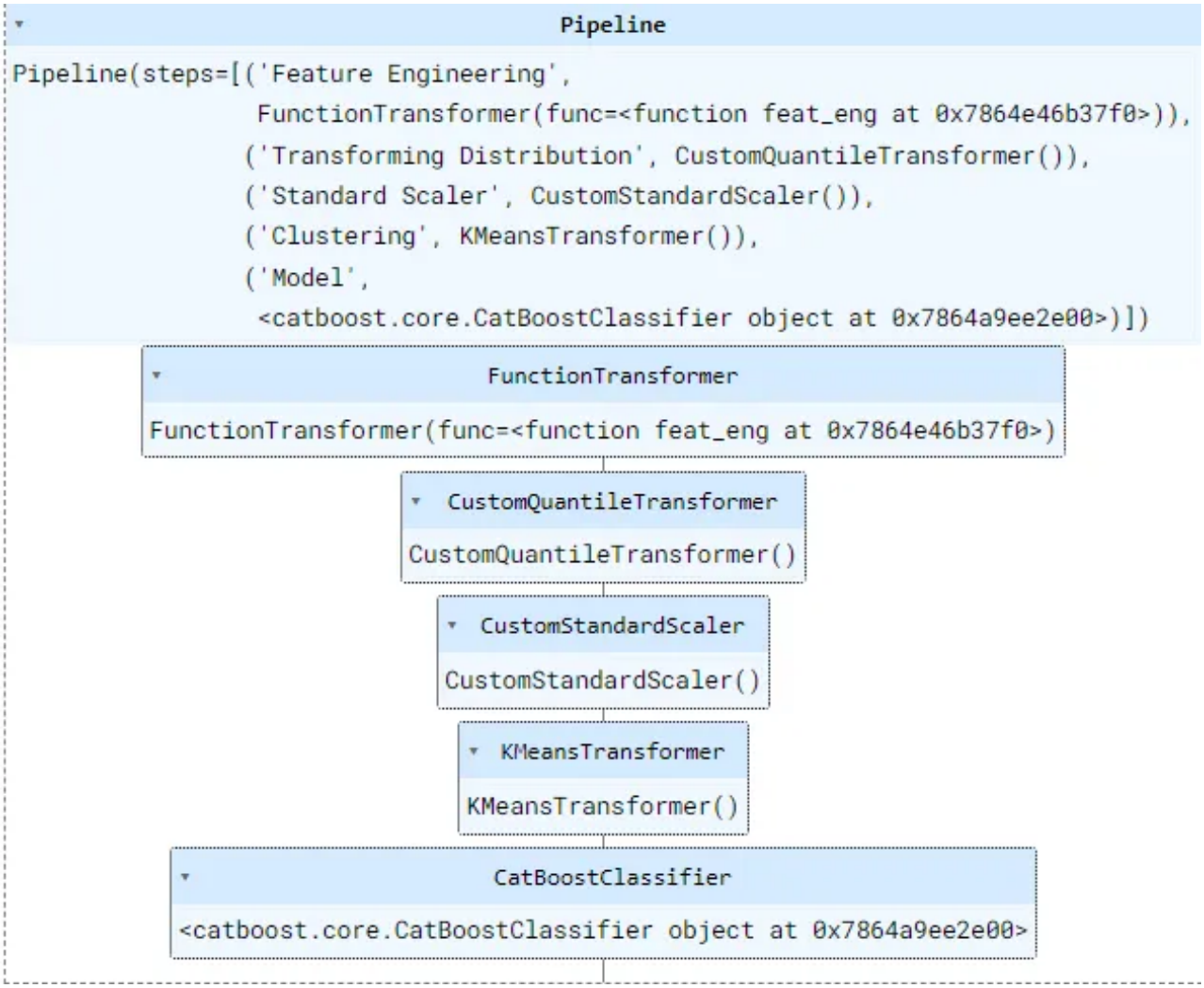
The best-performing model was the fine-tuned **CatBoostClassifier** with the following parameters:

```
Best Quadratic-Weighted Kappa score = 0.5227513447061922


 Best Params = {'learning_rate': 0.16,
                'iterations': 150,
                'max_depth': 6,
```

```
                'subsample': 0.6,
                'l2_leaf_reg': 3.7,
                'min_data_in_leaf': 45,
                'random_strength': 3.9000000000000004,
                'bootstrap_type': 'Bernoulli',
                'grow_policy': 'Depthwise',
                'leaf_estimation_method': 'Gradient'}
```

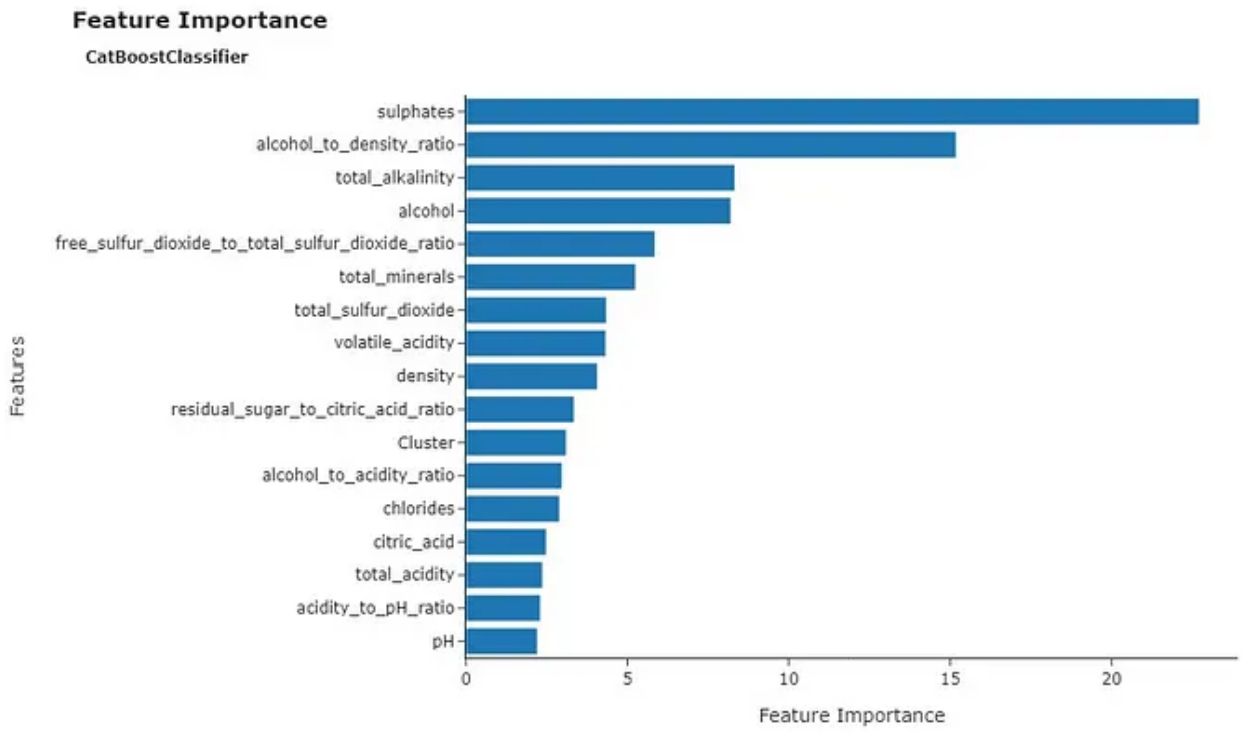
After having decided on the best model and the best parameters, all we need to do is use **Pipeline**'s *.set_params()* and add our model to the last step of the pipeline.

```
# Setting pipeline with Tuned CatBoost Model
pipeline.set_params(Model = CatBoostClassifier(**best_params2,
                    random_state = seed, verbose = False))
```



Final Structure of the Pipeline

We can also take a look at a *Feature Importance* plot, displaying the most relevant features in predicting **quality** during cross validation.



Most Important Features in Cross Validation

Sulphates earned the title of the most important feature during cross validation. It's also worth noting that **alcohol**, and related features like the **alcohol-to-density ratio** and **total alkalinity**, were also significantly important for predicting **quality**. This aligns with the positive correlation we observed earlier between **alcohol** and wine **quality**.

After training and validating the model, I've used Joblib to save my pipeline

```
import joblib

joblib.dump(pipeline, 'wine_quality_prediction.pkl')
```

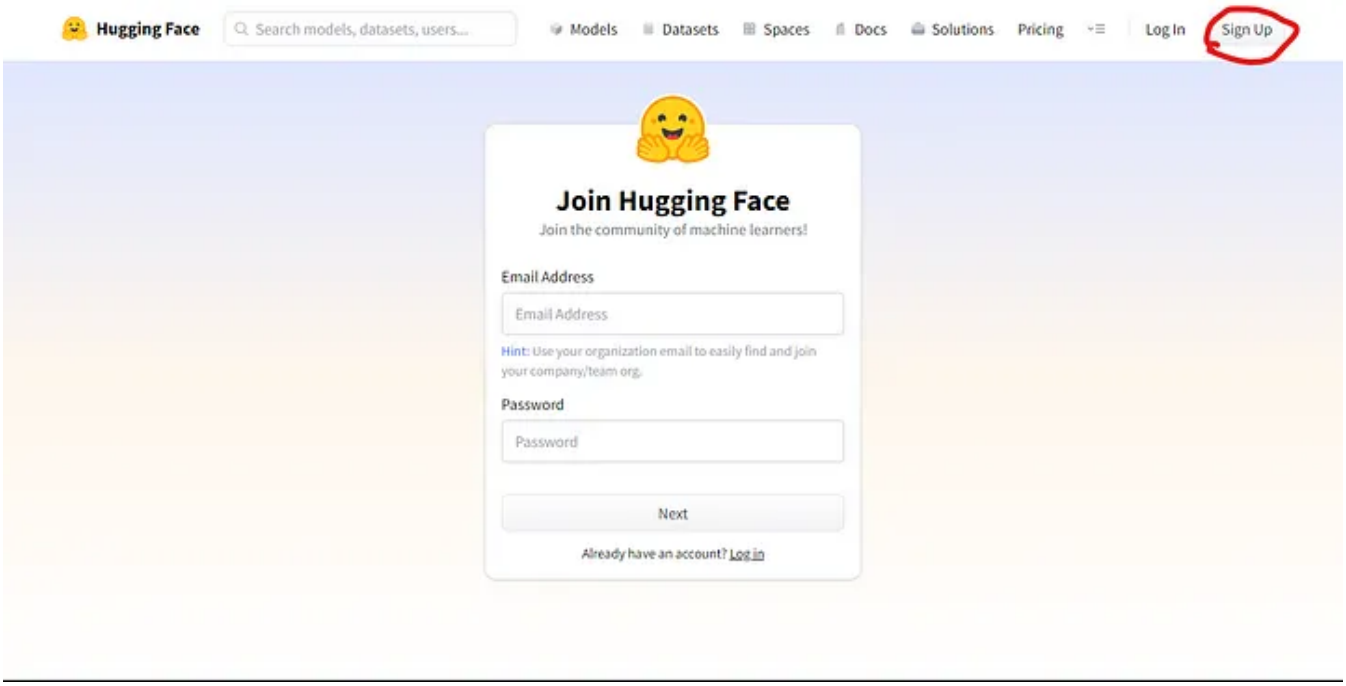
After running the code above, my pipeline — ***wine_quality_prediction.pkl*** — was saved to the same folder I was running my notebook on. With this file, I am now able to load my pipeline on another notebook or environment to run my model on.

Building a Streamlit app on Hugging Face

Alright, now that we have a saved pipeline and a validated model ready to roll, we're onto the next step— deployment!

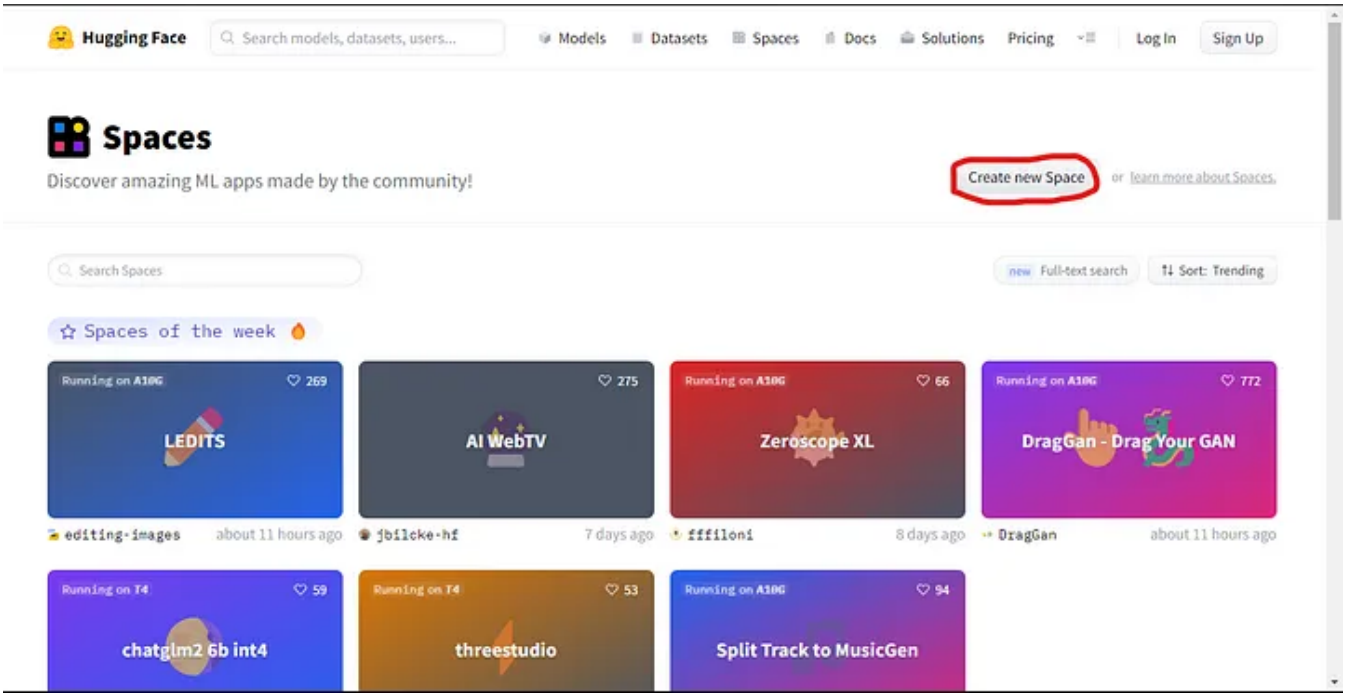
I'll be honest and say that I was a bit clueless about how to tackle this at the start. But after watching a few YouTube videos, I found out about **Hugging Face Spaces**, which allows us to easily build an app that can host our machine learning model.

If you don't have an account, just click on the **Sign Up** button located at the top right of the screen, write in your email address, set a password, and you're all set. This shouldn't be difficult at all.

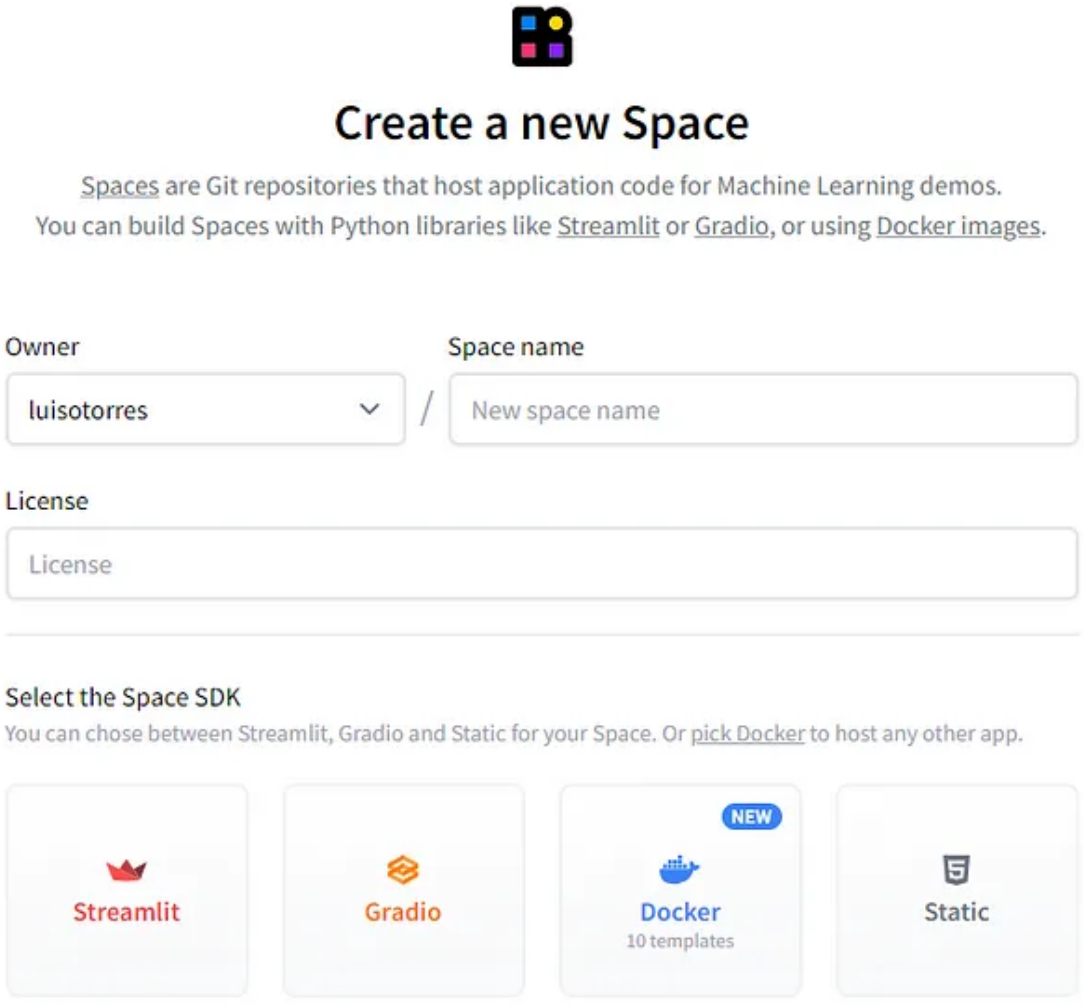


Sign Up Page for Hugging Face

After signing up, you can go to **Spaces**, on the top menu, and click on **Create new Space**

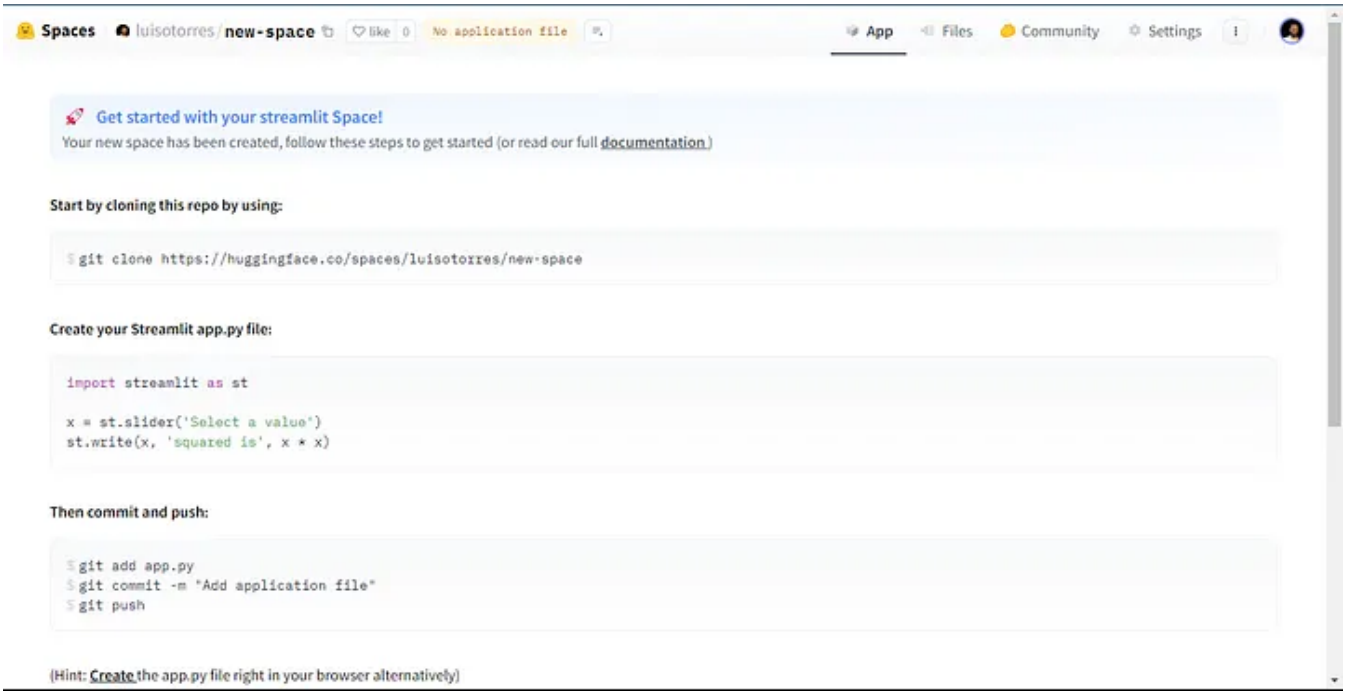


This takes us directly to another page, where we are able to give our Space a name, a license, a hardware, and a Space SDK. The SDK is what's going to allow us for creating the interface of our app.



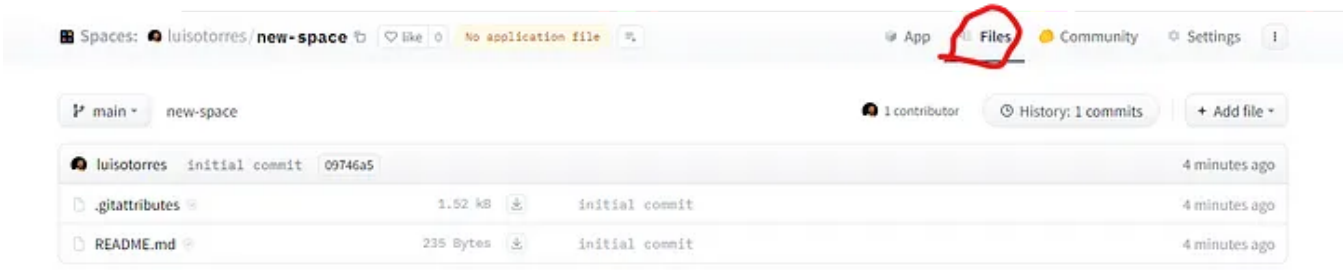
For this task, I have used **Streamlit**. According to its website, Streamlit allows for afaster way to build and share data web apps in minutes. All in pure Python, with nofront-end experience required. I'm not a specialist with Streamlit, but it was intutive and pretty easy to create my first app with it. You can have access to its documentation by clicking here.

After creating a new space, you will be met by a page similar to the one below. You can use git to clone the repository or you can create the necessary files for running your app directly from your browser.



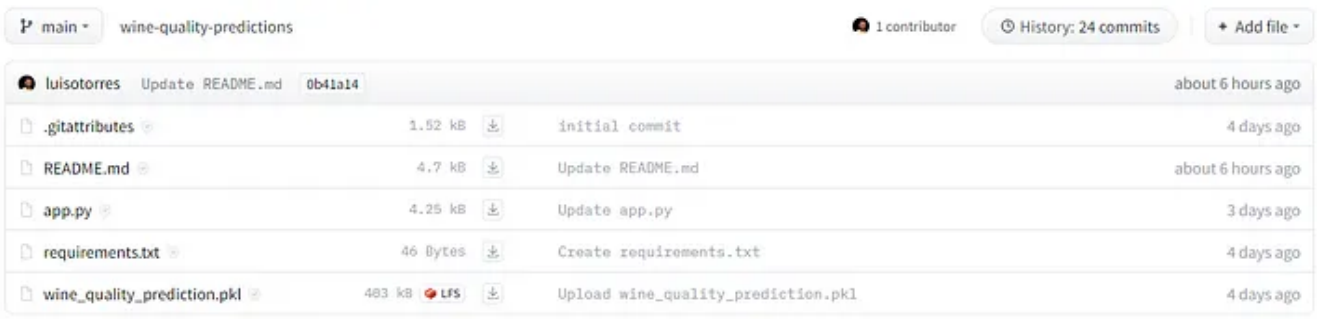
New Space Page

By clicking on **Files**, in the top menu, you'll have access to the files in your repo. Initially, all you will have here is a *.gitattributes* file and a *README.md* file.



Files page

In my ***Wine Quality Prediction*** app, I first added the pipeline file, ***wine_quality_prediction.pkl***. I then added a *requirements.txt* file, which simply contains a list of all the libraries necessary for running my app. You can see my **Files** page below.



Files Page For My App

The ***app.py*** file is a Python script where we load our pipeline and shape our app's interface using Streamlit. Even though this was my first time using Streamlit, I found it relatively easy to make a simple interface to host my model. It's not perfect, and there's definitely more for me to learn about Streamlit, but I managed to get my model up and running just fine. And by the way, do not hesitate to use all these *Large Language Models* available online to lend a hand with the coding whenever you feel it's necessary.

After having the ***app.py*** file all set up, you'll be able to run your app with no further problems by clicking on the ***App*** option in the top menu.

If you wish to test my app, you can access it at **Wine Quality Predictor Model**. By clicking on this link, you'll also be able to see the files I've uploaded to make this app run, including the coding behing the ***app.py*** script. Feel free to explore!

Also, if you want to check my Kaggle Notebook, where I've worked with the ***Wine Quality Dataset***, performed some Exploratory Data Analysis with Plotly and used tons of machine learning libraries, such as Scikit-learn and Optuna, to train and tune my machine learning models, you can check it at 🍷 ***Wine Quality: EDA, Prediction and Deploy***. Feel free to leave a feedback and an upvote, if you find this notebook useful!

Conclusion

We finally finish this journey from data preprocessing to deployment. This was, without a doubt, an enriching experience. It allowed me to explore a part of machine learning — and a bit of its real-world applications — in a way I hadn't explored yet. The process of building a classification model for a Kaggle competition, and then deploying it for public access on Hugging Face, was both quite challenging and rewarding. I hope this article has provided valuable insights into the process and encourages others to take the leap from theory to practice. Remember, the world of machine learning is vast and full of opportunities for those willing to step outside their comfort zone.

If you have any questions or suggestions, please don't hesitate to reach out.

Thank you so much for reading!

Luis Fernando Torres

Let's connect! 🔗
LinkedIn • Kaggle • HuggingFace

Machine Learning | Data Science | Hugging Face | Streamlit

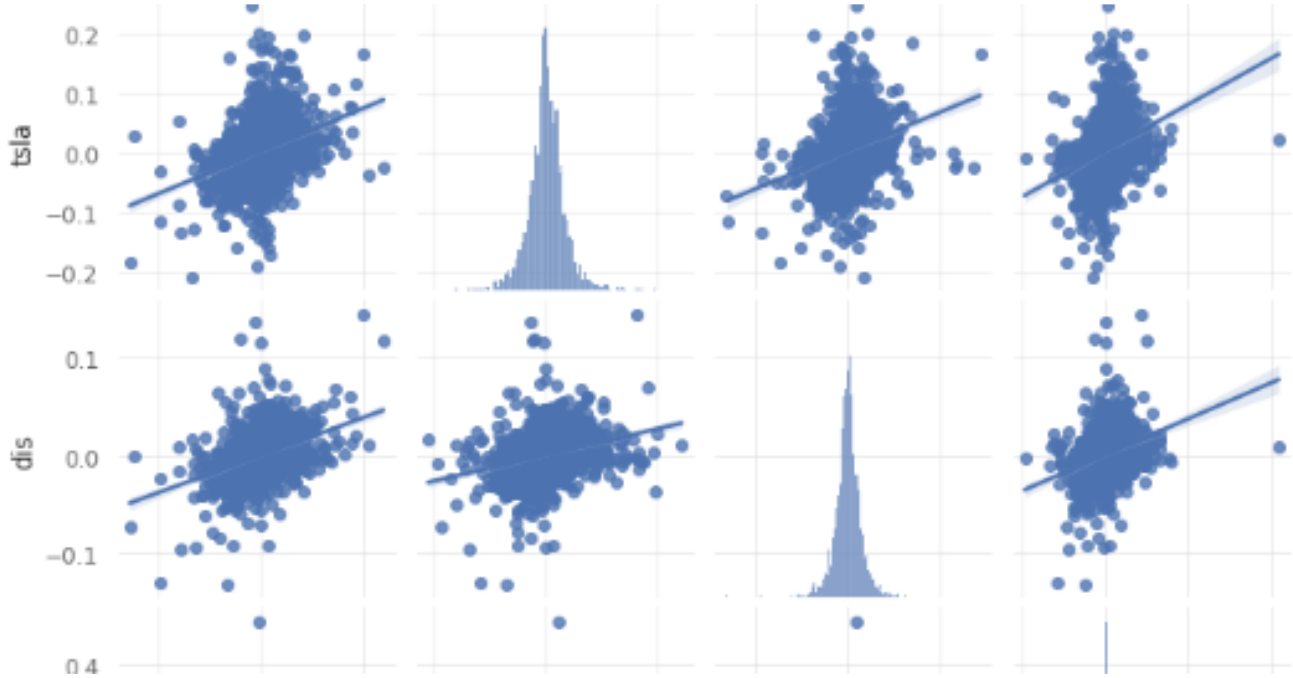
Follow

Written by Luís Fernando Torres

306 Followers

Data Scientist | Machine Learning | Commodities Trader & Investor

More from Luís Fernando Torres

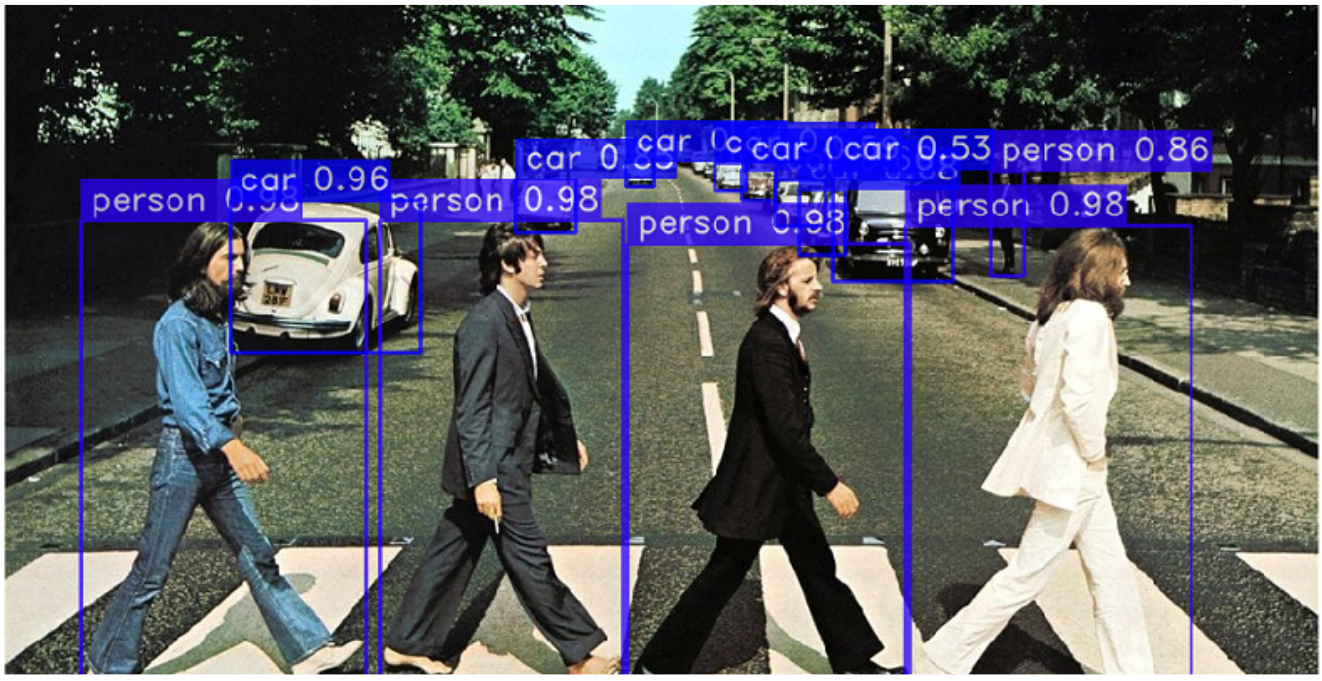


Luís Fernando Torres in The Modern Scientist

Introduction to Quant Investing with Python

Introduction

15 min read · Mar 30



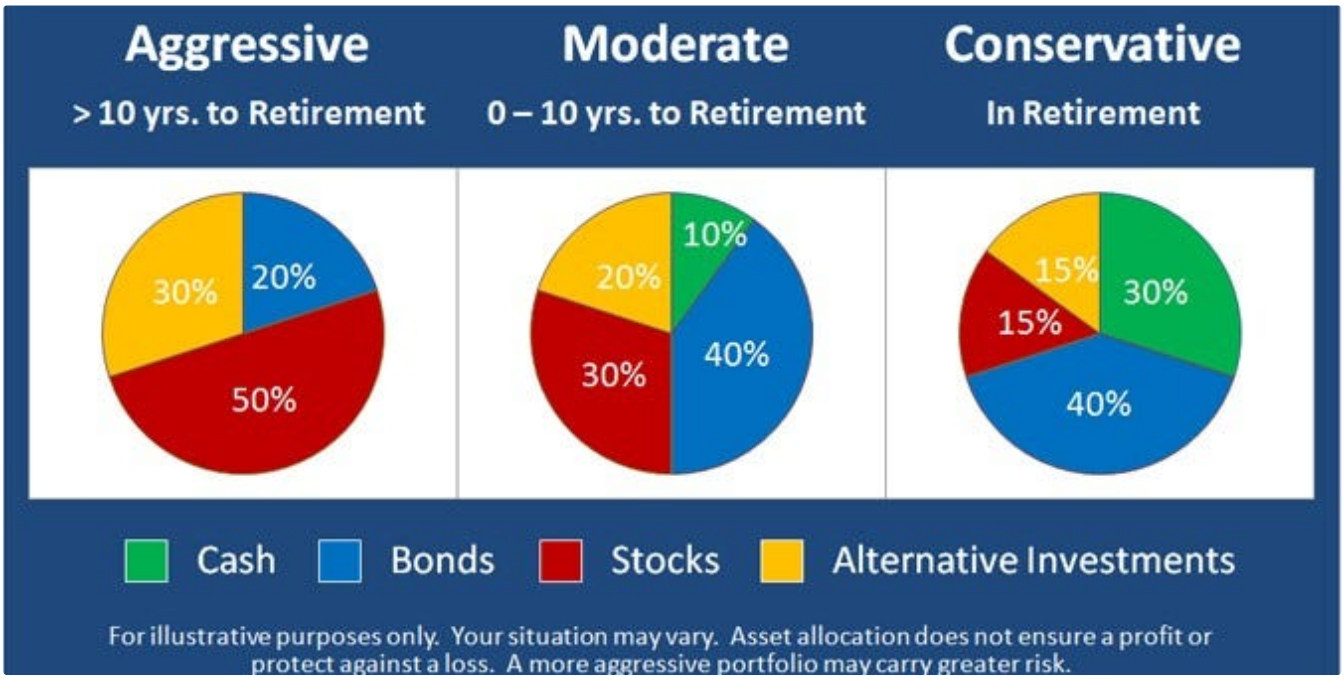
Luís Fernando Torres in LatinXinAI

Introducing YOLO-NAS: One of The Most Efficient Object Detection Algorithms

There's a new state-of-the-art model in town for real-time object detection

9 min read · May 7

147 2



Luís Fernando Torres

The Science of Smart Investing: Portfolio Evaluation with Python

This is the second part of my series on Quant Investing with Python. If you haven't had the chance to read the first part, I highly...

11 min read · Mar 30

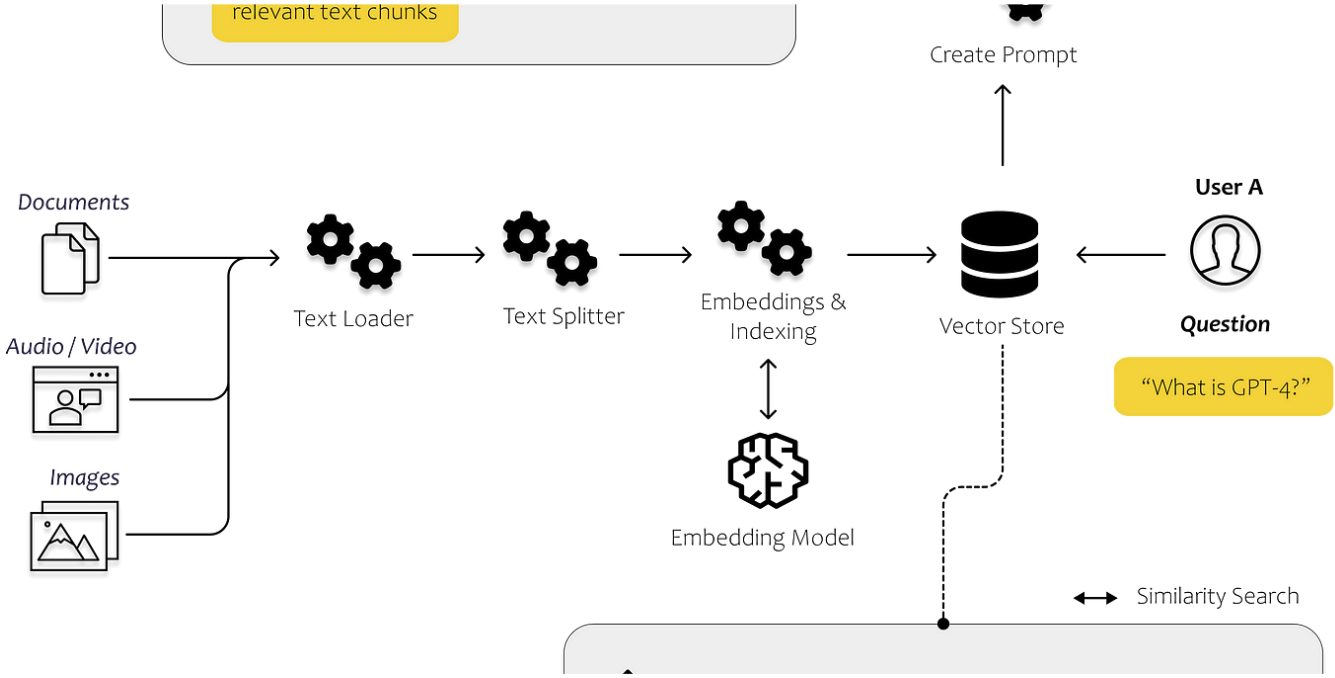
56

Luís Fernando Torres in LatinXinAI

S&P500 Volatility: ARCH vs GARCH Models

Deciding the ideal model for volatility forecasting

11 min read · Apr 17



Dominik Polzer in Towards Data Science

All You Need to Know to Build Your First LLM App

A step-by-step tutorial to document loaders, embeddings, vector stores and prompt templates

· 26 min read · Jun 22

2.3K 22

Sure, here is a blog post about the wine in the image:

Kiarna Cabernet Sauvignon: A Bold and Complex Wine from California

The Kiarna Cabernet Sauvignon is a full-bodied red wine from California. It is made from grapes grown in the Monterey AVA, which is known for its warm climate and rocky soil. The wine is aged in oak barrels for 18 months, which gives it a complex flavor profile.

The Kiarna Cabernet Sauvignon has a deep ruby color. The aroma is intense, with notes of blackberries, dark cherries, and cassis. The palate is full-bodied and well-balanced, with flavors of black fruits, oak, and spices. The tannins are present but not overwhelming.

This wine is best paired with grilled meats, roasted red meats, and game. It can also be enjoyed on its own. The Kiarna Cabernet Sauvignon is a bold and complex wine that is sure to please wine lovers of all levels.

Here are some more details about the wine:

- Region: Monterey AVA, California

Thomas Smith in The Generator

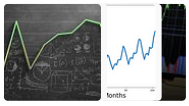
Google Bard's New Visual Feature is a Game Changer

Chatbots can officially see the world

✦ · 6 min read · 5 days ago

5K 106

Lists

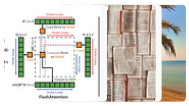


Predictive Modeling w/ Python

18 stories · 128 saves

Practical Guides to Machine Learning

10 stories · 137 saves

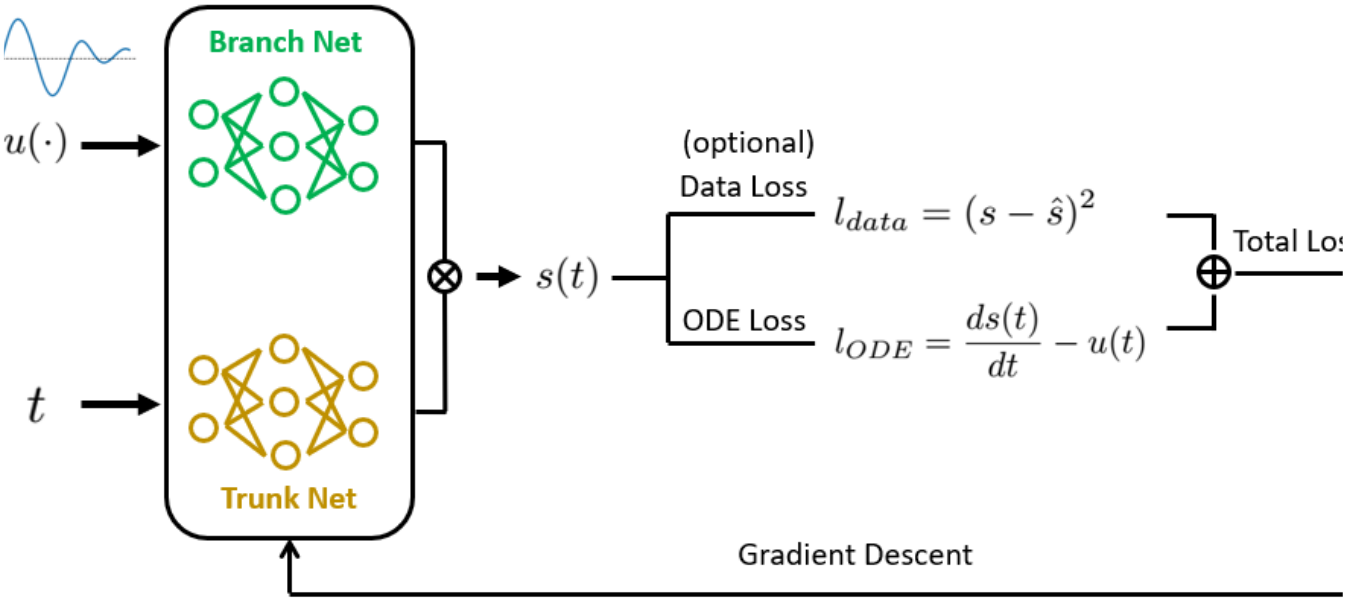


Natural Language Processing

408 stories · 55 saves

New_Reading_List

174 stories · 23 saves



Shuai Guo in Towards Data Science

Operator Learning via Physics-Informed DeepONet: Let's Implement It From Scratch

A deep dive into the DeepONets, physics-informed neural networks, and physics-informed DeepONets

· 23 min read · Jul 7

119

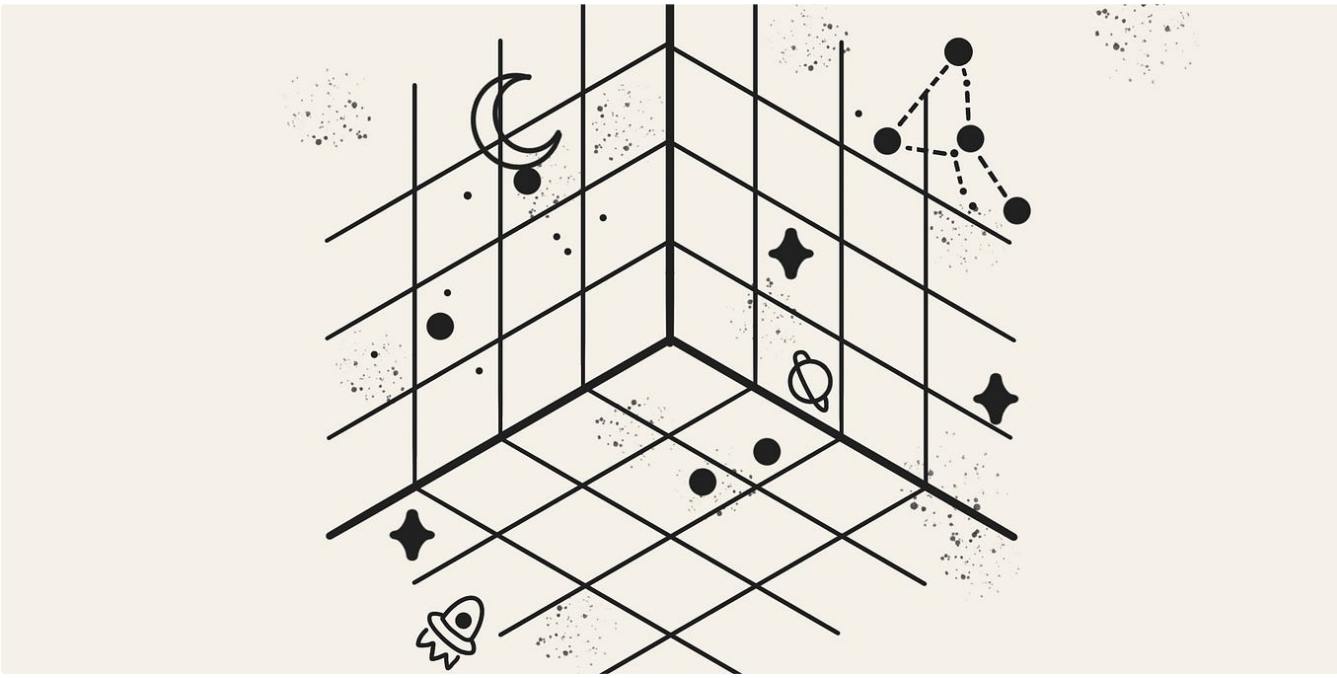
Jari Roomer in Better Humans

How I Eliminated Procrastination From My Life (Using Neuroscience)

Keep this part of the brain in optimal condition if you want to stop procrastinating.

· 6 min read · Jun 22

9.4K 103



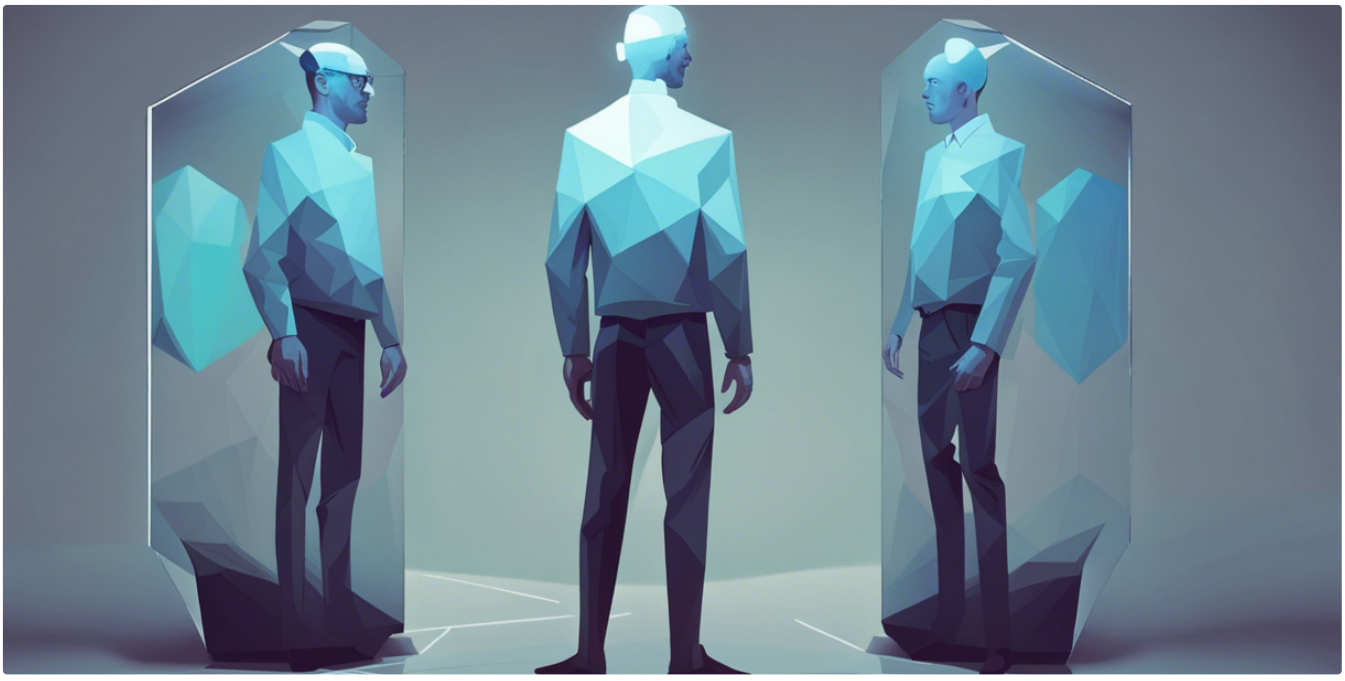
Leonie Monigatti in Towards Data Science

Explaining Vector Databases in 3 Levels of Difficulty

From noob to expert: Demystifying vector databases across different backgrounds

✨ · 8 min read · Jul 4

1.4K 15



Sergei Savvov in Better Programming

Create a Clone of Yourself With a Fine-tuned LLM

Unleash your digital twin

11 min read · 1 day ago