

Brewing Chemistry

This project was completed as the final Phase 5, Capstone assessment in the Flatiron School's Data Science Bootcamp.

Analysis by Erin Wasserman, July 2024

Kaggle Overview

This dataset presents an extensive collection of data from a craft beer brewery, spanning from January 2020 to January 2024. It encapsulates a rich blend of brewing parameters, sales data, and quality assessments, providing a holistic view of the brewing process and its market implications.

Business Problem

In the beer market, demand plays a key role in future industry dynamics. The introduction of new ingredients and innovative flavors into the beer market, combined with a business model that values consumer loyalty, will increase appeal among generations.

The main goal of any brewery is to focus on producing high quality beer. **Quality beer is a key success factor.** The most important thing a brewery can do is keep producing quality beer to stay competitive. As the variety of beers on the market increases, low-quality beers will be eliminated first.

1. Determine best malt and hops types for quality beer.
2. Assess malt-to-hops ratio impact on beer quality.
3. Evaluate ML accuracy in predicting beer characteristics.
4. Visualize beer styles based on quality ratings.

Data Understanding

Dataset Description

[Kaggle Dataset](#)

Data Format and Structure:

- The dataset is structured in a tabular format, provided in a CSV file for easy integration with various data analysis tools.
- It comprises over 10 million records, each representing a unique batch with a comprehensive set of features.

Intended Audience:

This dataset is invaluable for data scientists, brewing process engineers, market analysts, supply chain experts, and quality control professionals in the brewing industry. It is also highly relevant for academic research in food technology, fermentation science, and business analytics.

Disclaimer:

- The data is synthetic and intended for educational, analytical, and simulation purposes.
- Users are advised to apply appropriate data processing and analysis techniques for meaningful insights.
- This comprehensive dataset serves as a rich resource for exploring the intricacies of brewing science, market dynamics, and operational efficiency in the craft beer industry.

Highlighted Data Features

Brewing Parameters: Includes crucial brewing factors such as fermentation time, temperature, pH level, gravity, and ingredient ratios. These parameters are pivotal in understanding the brewing process and its impact on the final product.

Beer Styles: The dataset categorizes beers into various styles like IPA, Stout, Lager, etc.

Quality Scores: Each batch is rated for its quality on a scale, offering insights into the success and consistency of different brewing approaches.

Application: Brewing Process Optimization: Ideal for analysis aiming to correlate brewing techniques with beer quality, facilitating the optimization of brewing conditions for superior product quality.

✓ Import Libraries, Packages, and Environments

```
%%capture

import time
# Start the timer
start_time = time.time()

#Install Packages
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!pip install pyspark==3.0.1 #Big data processing
!pip install tensorflow==2.8.0 #2.5.0 #machine learning
!pip install --upgrade keras #neural net/deep learning modeling
!pip install elephas==0.4.3 #intergates TensorFlow/Keras with Spark
!pip install chempy #chemistry related calculations
!pip install xgboost #machine learning for large datasets
!pip install numba #to speed up computations on heavy metrics

#Import libraries for system operations and environment management
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2' # Suppress TensorFlow warnings

import sys
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"

# To retrieve info on system usage
import psutil

import warnings
# Suppress only specific warnings
warnings.filterwarnings("ignore", category=UserWarning) # Suppress UserWarnings
warnings.filterwarnings("ignore", category=DeprecationWarning) # Suppress DeprecationWarnings

# Stop the timer
end_time = time.time()

# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f"Data transformation execution time: {elapsed_time:.2f} seconds")

#Verify version for dependencies
import tensorflow as tf
import keras
print("TensorFlow version:", tf.__version__)
print("Keras version:", keras.__version__)
print("Python version:")
print(sys.version)
print()
```

 TensorFlow version: 2.8.0
 Keras version: 2.8.0
 Python version:
 3.10.12 (main, Jul 29 2024, 16:56:48) [GCC 11.4.0]

```
# Start the timer
start_time = time.time()

# Import the necessary third party packages
from pyspark import SparkContext, SparkConf
from pyspark.sql import SparkSession
from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler
from pyspark.ml import Pipeline
from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.ml.regression import GBRegressor
from pyspark.sql.types import IntegerType, FloatType, DoubleType
from pyspark.sql import functions as F
from pyspark.sql.functions import col, when, count, split, lit, udf
from pyspark.ml.stat import Correlation
from pyspark.ml.feature import PCA, StandardScaler
from pyspark.ml.clustering import KMeans, KMeansModel
from pyspark.ml.evaluation import ClusteringEvaluator
from pyspark.sql import Window
from pyspark.sql.functions import expr, broadcast

import pandas as pd
import numpy as np
import plotly.express as px
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
from matplotlib.colors import ListedColormap
from matplotlib.colors import LinearSegmentedColormap
from google.colab import files
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import *
from sklearn.model_selection import train_test_split
from sklearn.metrics import *
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.metrics import roc_curve, roc_auc_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score
from sklearn.metrics import classification_report
from sklearn.inspection import permutation_importance
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.tree import plot_tree
from sklearn.manifold import TSNE
from sklearn.metrics import silhouette_score
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import train_test_split
from sklearn.manifold import TSNE
import xgboost as xgb
from xgboost import DMatrix
from numba import jit
import zipfile
import glob
import joblib

from chempy.kinetics.ode import get_odesys
from chempy.kinetics.rates import MassAction
from chempy import Equilibrium, Substance
from scipy.optimize import fsolve
from collections import defaultdict
from chempy.chemistry import Species
from chempy.equilibria import EqSystem
from math import log10

# TensorFlow and Keras for deep learning (for future work)
import tensorflow as tf
from tensorflow.keras import models
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.optimizers import Adam, RMSprop, SGD, serialize
from tensorflow.keras.utils import *
from tensorflow.python.keras.utils import *
```

```

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Activation
from elephas.utils.rdd_utils import to_simple_rdd
from elephas.spark_model import SparkModel
from elephas.ml_model import ElephasEstimator
from tensorflow.keras import optimizers
from tensorflow.keras.optimizers import serialize

# Set seaborn style for better visualizations
sns.set(style="whitegrid")

# Define your custom color palette using hex codes
custom_colors = ["#f7e1a1", "#e9ad3f", "#d98416", "#b74d00", "#9f3400", "#811f00", "#5e0e00", "#410500", "#2f0200", "#0d0000"]

# Create a custom colormap based on your custom color palette
custom_cmap = mcolors.ListedColormap(custom_colors)

# Set this custom palette globally for Seaborn
sns.set_palette(custom_colors)

# Set the same custom palette for Matplotlib
plt.rcParams['axes.prop_cycle'] = plt.cycler(color=custom_colors)

# Stop the timer
end_time = time.time()

# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f"Data transformation execution time: {elapsed_time:.2f} seconds")

⚠ WARNING
Data transformation execution time: 3.20 seconds

```

✓ Notebook Architecture

The following code blocks establish the general setup architecture for the notebook. The Spark session can be initialized according to the computational resources available. Also, pickle functions are set to ease the computational resources by saving the state of an object to a disk, so that it can be reused later without having to recompute or recreate it.

```

# Start the timer
start_time = time.time()

# Initialize the SparkConf object
conf = SparkConf()

# Set all the configurations you need
conf.setAppName('Elephas_App') \
    .setMaster('local[8]') \
    .set("spark.driver.maxResultSize", "18g")

#comment or uncomment the code block that suits your computing architecture

# Initialize Spark session (default)
#spark = SparkSession.builder \
#    .appName("Beer Quality Analysis") \
#    .config("spark.sql.shuffle.partitions", "50") \
#    .config("spark.executor.memory", "4g") \
#    .config("spark.executor.cores", "4") \
#    .getOrCreate()

# Initialize Spark session (optimized CPU)
#spark = SparkSession.builder \
#    .appName("Beer Quality Analysis") \
#    .config("spark.sql.shuffle.partitions", "50") \
#    .config("spark.driver.memory", "8g") \
#    .config("spark.executor.memory", "4g") \
#    .config("spark.executor.cores", "2") \
#    .config("spark.driver.maxResultSize", "4g") \
#    .getOrCreate()

# Initialize the Spark session (optimized T4_GPU)
#spark = SparkSession.builder \
#    .appName("Beer Quality Analysis") \
#    .config("spark.sql.shuffle.partitions", "200") \
#    .config("spark.driver.memory", "16g") \
#    .config("spark.executor.memory", "8g") \
#    .config("spark.executor.cores", "4") \
#    .config("spark.driver.maxResultSize", "8g") \
#    .config("spark.executor.instances", "1") \
#    .getOrCreate()

# Start Spark session with optimized settings for T4 GPU
spark = SparkSession.builder \
    .appName("BeerClustering") \
    .config("spark.sql.shuffle.partitions", "400") \
    .config("spark.driver.memory", "16g") \
    .config("spark.executor.memory", "8g") \
    .config("spark.executor.cores", "4") \
    .config("spark.executor.instances", "4") \
    .getOrCreate()

# Determine the number of cores available
total_cores = spark._jsc.sc().getExecutorMemoryStatus().keySet().size()
print(f"Total number of cores available: {total_cores}")

# Set the parallelism to a reasonable value, e.g., half or all of the available cores
parallelism = min(total_cores, 4) # Adjust this based on project specific needs and testing

# Print Spark configuration settings
spark.sparkContext.setLogLevel("DEBUG")
spark.sparkContext.setCheckpointDir("/tmp/spark-checkpoints")
print(spark.sparkContext.getConf().getAll())

# Stop the timer
end_time = time.time()

# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f"Data transformation execution time: {elapsed_time:.2f} seconds")

/usr/lib/python3.10/subprocess.py:1796: RuntimeWarning: os.fork() was called. os.fork() is incompatible with multithreaded (
  self.pid = _posixsubprocess.fork_exec(
Total number of cores available: 1
[('spark.executor.instances', '4'), ('spark.driver.port', '32795'), ('spark.app.name', 'BeerClustering'), ('spark.executor.

```

Data transformation execution time: 6.96 seconds



Explanation of Settings:

spark.sql.shuffle.partitions: This setting determines the number of partitions to use when shuffling data for operations such as joins or aggregations. A value of 50 is a recommended starting point for balancing performance and resource utilization.

spark.driver.memory: Allocates 8 GB of memory for the Spark driver, which is the main control node of your Spark application. This setting ensures that the driver has sufficient memory to manage the overall execution of the Spark jobs.

spark.executor.memory: Allocates 4 GB of memory for each Spark executor. Executors are the distributed agents that execute the tasks assigned by the driver. This setting ensures that each executor has adequate memory to perform its tasks efficiently.

spark.executor.cores: Allocates 2 CPU cores for each executor. This configuration helps in parallelizing tasks effectively, allowing the executors to handle multiple tasks concurrently.

spark.driver.maxResultSize: Sets a limit of 4 GB on the maximum size of results that can be collected to the driver. This prevents out-of-memory errors when collecting large results, ensuring the stability of the Spark application.



The following code block defines two functions to save and load models using spark. Pickle and Jiblib were considered and not needed for this project given the other resource constraints. The `save_model` function saves the model to a file, and the `load_model` function loads the model from a file. This helps save time by avoiding the need to recreate the model every time.

```

# Start the timer
start_time = time.time()

# Directory to save models and checkpoints
model_dir = "/content/Brewing_Chemistry/Models/"
os.makedirs(model_dir, exist_ok=True)

def save_spark_model(model, filename):
    """
    Save a Spark MLlib model.

    Parameters:
    model : pyspark.ml.Model
        The Spark MLlib model to be saved.
    filename : str
        The name of the directory where the model will be saved.

    Returns:
    None
    """
    filepath = os.path.join(model_dir, filename)
    model.save(filepath)
    print(f"Model saved to {filepath}")

def load_spark_model(model_class, filename):
    """
    Load a Spark MLlib model.

    Parameters:
    model_class : type
        The class of the model to be loaded (e.g., RandomForestRegressorModel).
    filename : str
        The name of the directory from which the model will be loaded.

    Returns:
    pyspark.ml.Model
        The loaded Spark MLlib model.
    """
    filepath = os.path.join(model_dir, filename)
    model = model_class.load(filepath)
    print(f"Model loaded from {filepath}")
    return model

def save_spark_dataframe(df, filename):
    """
    Save a Spark DataFrame as a Parquet file.

    Parameters:
    df : pyspark.sql.DataFrame
        The Spark DataFrame to be saved.
    filename : str
        The name of the file where the DataFrame will be saved.

    Returns:
    None
    """
    filepath = os.path.join(model_dir, filename)
    df.write.mode("overwrite").parquet(filepath)
    print(f"DataFrame saved to {filepath}")

def load_spark_dataframe(filename):
    """
    Load a Spark DataFrame from a Parquet file.

    Parameters:
    filename : str
        The name of the file from which the DataFrame will be loaded.

    Returns:
    pyspark.sql.DataFrame
        The loaded Spark DataFrame.
    """
    filepath = os.path.join(model_dir, filename)
    df = spark.read.parquet(filepath)
    print(f"DataFrame loaded from {filepath}")
    return df

```

```
def checkpoint_dataframe(df, checkpoint_dir="/tmp/spark-checkpoints"):
    """
    Checkpoint a Spark DataFrame.

    Parameters:
    df : pyspark.sql.DataFrame
        The Spark DataFrame to be checkpointed.
    checkpoint_dir : str
        The directory where the checkpoint will be saved.

    Returns:
    pyspark.sql.DataFrame
        The checkpointed DataFrame.
    """
    spark.sparkContext.setCheckpointDir(checkpoint_dir)
    df_checkpointed = df.checkpoint()
    print(f'DataFrame checkpointed to {checkpoint_dir}')
    return df_checkpointed
```

✓ Import Data

This code block is focused on importing the data using Spark.

```
# Start the timer
start_time = time.time()

# Unzip the file
!unzip /content/sample_data/brewery_dataset.zip -d /content/sample_data/

# Find the path of the unzipped file (assuming there's only one CSV file)
csv_file_path = glob.glob('/content/sample_data/**/*.csv', recursive=True)[0]

# Load the dataset using Spark
beer_sample_set = spark.read.csv(csv_file_path, header=True, inferSchema=True)

# Preview the dataset shape and first few rows
print("Dataset shape: Rows -", beer_sample_set.count(), " Columns -", len(beer_sample_set.columns))
print("Schema of the dataset:")
beer_sample_set.printSchema()

print("First few rows of the dataset:")
beer_sample_set.show(5) # Show the first 5 rows

# Stop the timer
end_time = time.time()

# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f'Data transformation execution time: {elapsed_time:.2f} seconds')

📁 Archive: /content/sample_data/brewery_dataset.zip
  inflating: /content/sample_data/brewery_data_complete_extended.csv
Dataset shape: Rows - 10000000 Columns - 20
Schema of the dataset:
root
 |-- Batch_ID: integer (nullable = true)
 |-- Brew_Date: string (nullable = true)
 |-- Beer_Style: string (nullable = true)
 |-- SKU: string (nullable = true)
 |-- Location: string (nullable = true)
 |-- Fermentation_Time: integer (nullable = true)
 |-- Temperature: double (nullable = true)
 |-- pH_Level: double (nullable = true)
 |-- Gravity: double (nullable = true)
 |-- Alcohol_Content: double (nullable = true)
 |-- Bitterness: integer (nullable = true)
 |-- Color: integer (nullable = true)
 |-- Ingredient_Ratio: string (nullable = true)
 |-- Volume_Produced: integer (nullable = true)
 |-- Total_Sales: double (nullable = true)
 |-- Quality_Score: double (nullable = true)
 |-- Brewhouse_Efficiency: double (nullable = true)
 |-- Loss_During_Brewing: double (nullable = true)
```



```
-- Loss_During_Fermentation: double (nullable = true)
-- Loss_During_Bottling_Kegging: double (nullable = true)
```

First few rows of the dataset:

Batch_ID	Brew_Date	Beer_Style	SKU	Location	Fermentation_Time	Temperature	pH_Level
7870796	2020-01-01 00:00:19	Wheat Beer	Kegs	Whitefield	16	24.204250857069873	5.2898454476095615
9810411	2020-01-01 00:00:31	Sour	Kegs	Whitefield	13	18.086762947259544	5.275643382756193
2623342	2020-01-01 00:00:40	Wheat Beer	Kegs	Malleswaram	12	15.539332669116469	4.7780156232459765
8114651	2020-01-01 00:01:37	Ale	Kegs	Rajajinagar	17	16.41848910394318	5.345260585546188
4579587	2020-01-01 00:01:43	Stout	Cans	Marathahalli	18	19.144907654338517	4.86185374113861

only showing top 5 rows

Data transformation execution time: 81.51 seconds

✓ Preprocessing

Categorical, numeric, and mixed numeric values to be processed accordingly.

```
# Start the timer
start_time = time.time()

# Create a copy of the DataFrame to not alter original
beer_copy = beer_sample_set

# Stop the timer
end_time = time.time()

# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f'Data transformation execution time: {elapsed_time:.2f} seconds')
```

↗ Data transformation execution time: 0.00 seconds

```
# Start the timer
start_time = time.time()

# Compute summary statistics
describe_df = beer_copy.describe()

# Show the summary statistics
describe_df.show()

# Stop the timer
end_time = time.time()

# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f'Data transformation execution time: {elapsed_time:.2f} seconds')
```

↗

summary	Batch_ID	Brew_Date	Beer_Style	SKU	Location	Fermentation_Time	Temperature
count	10000000	10000000	10000000	10000000	10000000	10000000	10000000
mean	4999999.5	null	null	null	null	14.500898	19.999898511018827
stddev	2886751.4902856858	null	null	null	null	2.872006096518228	2.8870297120328576
min	0	2020-01-01 00:00:19	Ale	Bottles	Electronic City	10	15.000001163771435
max	9999999	2023-12-31 23:59:54	Wheat Beer	Pints	Yelahanka	19	24.999998289887966

Data transformation execution time: 95.24 seconds



Focusing on the column subset related to the chemical properties of beer, the pH seems consistently below 7. The average temperature slightly below room standard room temperature. The bitterness has a large standard deviation relative to the mean.

The standard metric that will be used for this analysis of quality score seems to have a small standard deviation and the scores all seem to be high in a range of 6-9.99 with an average score of 7.99.

```
# Cache the DataFrame to avoid recomputation
beer_sample_set.cache()

# Start the timer
start_time = time.time()

# Select numeric columns
numeric_columns = [field.name for field in beer_copy.schema.fields if isinstance(field.dataType, (IntegerType, FloatType, DoubleType))]

# Efficiently compute the correlation matrix using Spark's built-in method
corr_matrix_spark = beer_copy.select(numeric_columns).toPandas().corr()

# Convert Spark's correlation matrix to a Pandas DataFrame
corr_matrix = pd.DataFrame(corr_matrix_spark, index=numeric_columns, columns=numeric_columns)

# Plot the heatmap
plt.figure(figsize=(12, 8))
sns.heatmap(corr_matrix, annot=True, fmt=".2f", cmap=custom_cmap)
plt.title('Correlation Matrix')

# Save the plot as an image with better layout
filename = "all_data_correlation_matrix.jpg"
plt.savefig(filename, bbox_inches='tight')

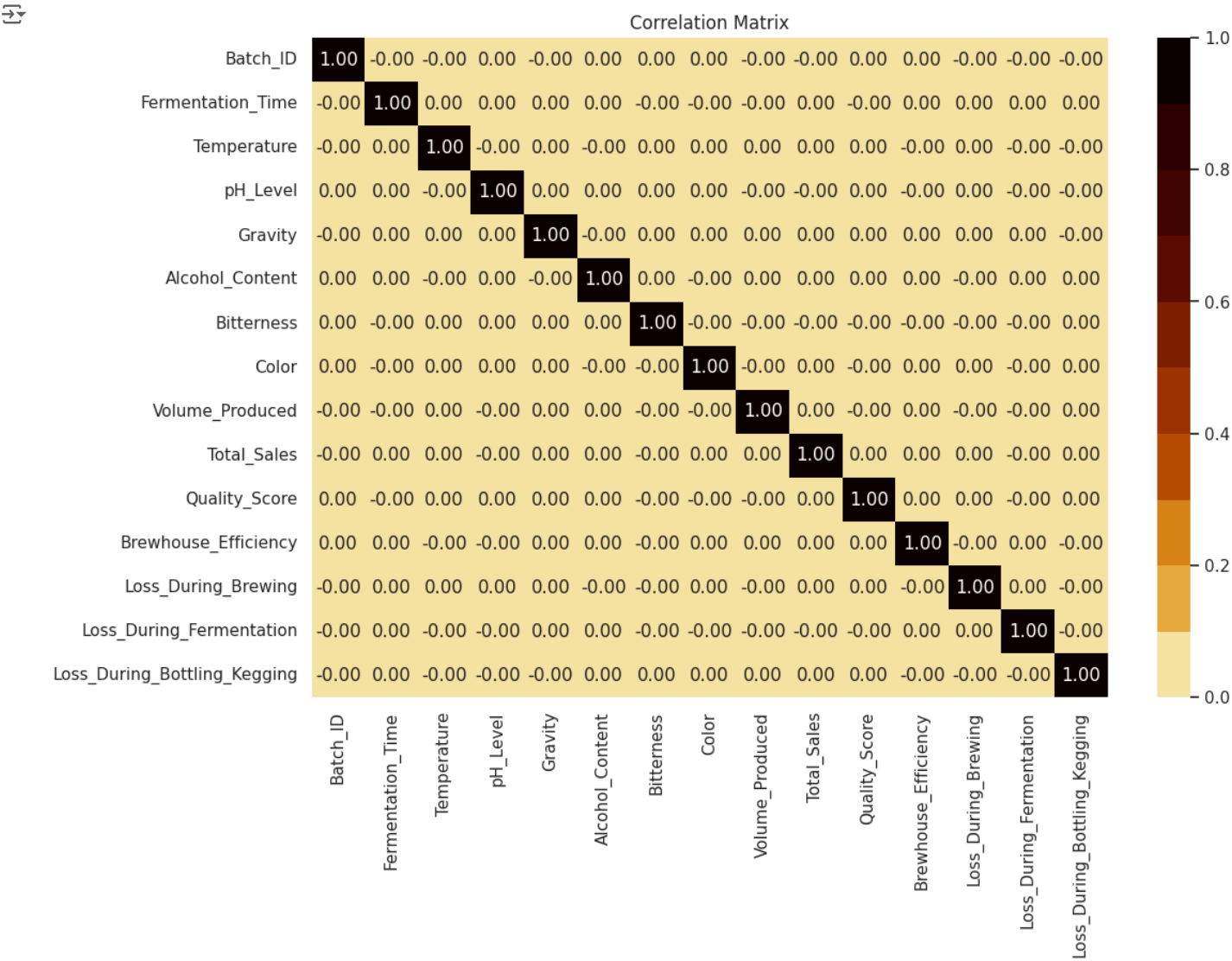
# Download the file
files.download(filename)

# Show the plot
plt.show()

# Stop the timer
end_time = time.time()

# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f>Data transformation execution time: {elapsed_time:.2f} seconds")
```



Data transformation execution time: 153.62 seconds



The most notable insight here is that the correlation plot seems to be completely uniform. Processing this data set will be challenging. To discover the relationships will require advanced techniques and deep domain knowledge.

```
# Start the timer
start_time = time.time()

# Select relevant numerical columns
numeric_columns = ['pH_Level', 'Alcohol_Content', 'Bitterness', 'Color', 'Quality_Score']

# Collect the data as a Pandas DataFrame
beer_sample_pandas = beer_copy.select(numeric_columns).toPandas()

## Plot the pair plot
plt.figure(figsize=(12, 10))
sns.pairplot(beer_sample_pandas, plot_kws={'color': '#d98416'}, diag_kws={'color': '#d98416'})

# Save the plot as an image with better layout
filename = "all_data_pair_plot.jpg"
plt.savefig(filename, bbox_inches='tight')

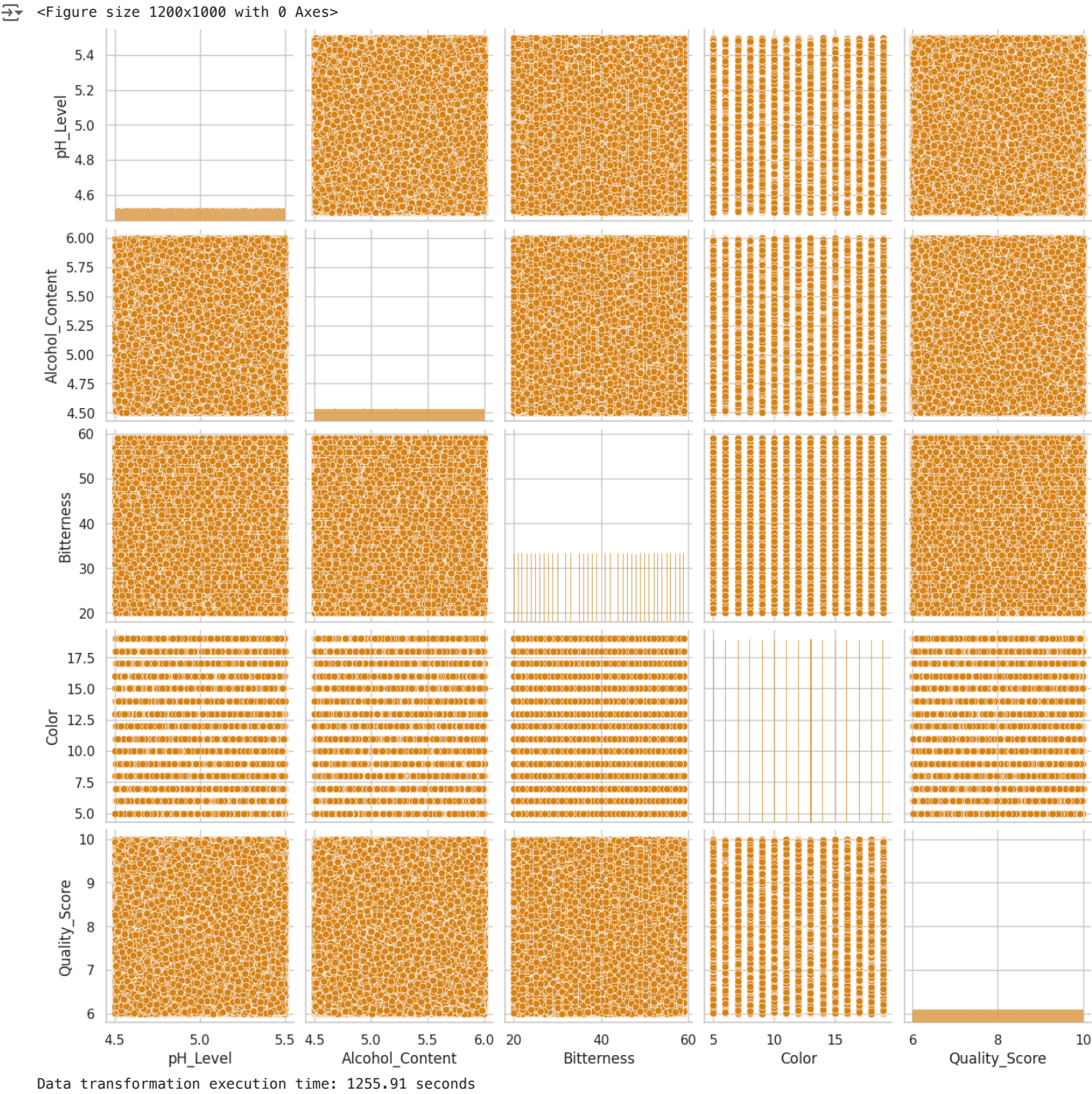
# Download the file
files.download("all_data_pair_plot.jpg")

# Show the plot
plt.show()

# Stop the timer
end_time = time.time()

# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f>Data transformation execution time: {elapsed_time:.2f} seconds")
```



This confirms the correlation plot conclusion. These pair plots show flat and uniform data. This shows how important this step in the preprocessing is because without these plots, it wouldn't be so obvious that the data set would be a challenge for finding relationships.

```
# Start the timer
start_time = time.time()

# Define the columns relevant for feature engineering (features that contain chemistry dependent properties)
relevant_columns = ['pH_Level', 'Ingredient_Ratio', 'Fermentation_Time', 'Temperature',
                    'Alcohol_Content', 'Bitterness', 'Color', 'Quality_Score']
categorical_columns = ['Beer_Style']

# Create a subset of the data with relevant columns
beer_quality_predict = beer_copy.select(relevant_columns + categorical_columns)

# Preview the subset of the data
print("Subset of the data with relevant columns:")
beer_quality_predict.show(5)

# Stop the timer
end_time = time.time()

# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f>Data transformation execution time: {elapsed_time:.2f} seconds")
```

Subset of the data with relevant columns:

	pH_Level	Ingredient_Ratio	Fermentation_Time	Temperature	Alcohol_Content	Bitterness	Color	Quality_Score
	5.2898454476095615	1:0.32:0.16	16	24.204250857069873	5.370842159553436	20	5	8.57701633109
	5.275643382756193	1:0.39:0.24	13	18.086762947259544	5.096053082797625	36	14	7.420540752553
	4.7780156232459765	1:0.35:0.16	12	15.539332669116469	4.824737120959184	30	10	8.451364886803
	5.345260585546188	1:0.35:0.15	17	16.41848910394318	5.509243080797997	48	18	9.671859404043
	4.86185374113861	1:0.46:0.11	18	19.144907654338517	5.133624684263243	57	13	7.895333676172

only showing top 5 rows

Data transformation execution time: 0.25 seconds



This analysis will focus on the chemical properties of the beer making process that lead to top quality scores (<https://www.micetgroup.com/the-business-competition-strategy-of-craft-brewery/>). So here a chemical properties data subset is formed.

```
# Start the timer
start_time = time.time()

# Show schema of the DataFrame
beer_quality_predict.printSchema()

# Show summary statistics for numerical columns
beer_quality_predict.describe().show()

# Stop the timer
end_time = time.time()

# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f>Data transformation execution time: {elapsed_time:.2f} seconds")
```

```
root
|-- pH_Level: double (nullable = true)
|-- Ingredient_Ratio: string (nullable = true)
|-- Fermentation_Time: integer (nullable = true)
|-- Temperature: double (nullable = true)
|-- Alcohol_Content: double (nullable = true)
|-- Bitterness: integer (nullable = true)
|-- Color: integer (nullable = true)
|-- Quality_Score: double (nullable = true)
|-- Beer_Style: string (nullable = true)
```

summary	pH_Level	Ingredient_Ratio	Fermentation_Time	Temperature	Alcohol_Content	Bitterness
count	10000000	10000000	10000000	10000000	10000000	10000000

mean	4.999940543893489	null	14.500898	19.999898511018827	5.249709006579308	39.4961996
stddev	0.28863762894103545	null	2.872006096518228	2.8870297120328576	0.43296144791729213	11.545572488490313
min	4.500000005935603	1:0.20:0.10	10	15.000001163771435	4.500000235642255	20
max	5.499999818305633	1:0.50:0.30	19	24.999998289887966	5.999999932506248	59

Data transformation execution time: 34.87 seconds

```
# Start the timer
start_time = time.time()

# Extract Individual Components
beer_quality_predict = beer_quality_predict.withColumn('Water_Ratio', lit(1.0))
beer_quality_predict = beer_quality_predict.withColumn('Malt_Ratio', split(col('Ingredient_Ratio'), ':').getItem(1).cast('float'))
beer_quality_predict = beer_quality_predict.withColumn('Hops_Ratio', split(col('Ingredient_Ratio'), ':').getItem(2).cast('float'))

# Assemble all relevant features into a vector
vector_col = 'features'
assembler = VectorAssembler(inputCols=['Water_Ratio', 'Malt_Ratio', 'Hops_Ratio', 'pH_Level', 'Fermentation_Time', 'Temperature', 'Alcohol_Content', 'Bitterness', 'Color', 'Quality_Score'], outputCol=vector_col)
df_vector = assembler.transform(beer_quality_predict).select(vector_col)

# Calculate Correlation Matrix
matrix = Correlation.corr(df_vector, vector_col).head()[0]
corr_matrix_indv = pd.DataFrame(matrix.toArray(), columns=['Water_Ratio', 'Malt_Ratio', 'Hops_Ratio', 'pH_Level', 'Fermentation_Time', 'Alcohol_Content', 'Bitterness', 'Color', 'Quality_Score'])

# Visualize Correlations
plt.figure(figsize=(12, 8))
sns.heatmap(corr_matrix_indv, annot=True, cmap=custom_cmap)
plt.title('Correlation Matrix for Individual Ratios')

# Save the plot as an image with better layout
filename = "property_data_correlation_matrix.jpg"
plt.savefig(filename, bbox_inches='tight')

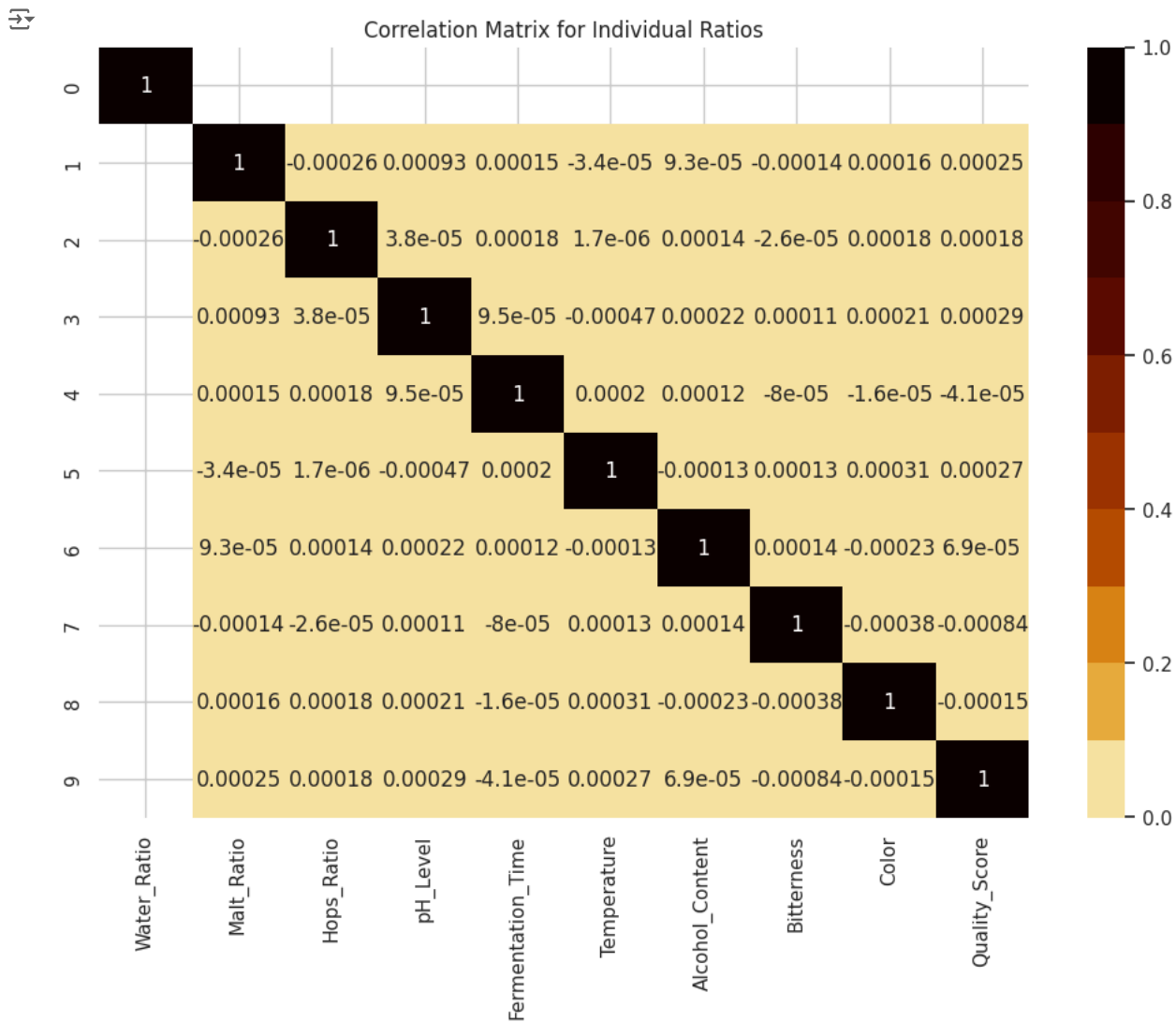
# Download the file
files.download("property_data_correlation_matrix.jpg")

# Show the plot
plt.show()

# Stop the timer
end_time = time.time()

# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f"Data transformation execution time: {elapsed_time:.2f} seconds")
```

Remarkably uniform. Traditional correlation matrices would show hints of a relationship. In this case, a deep dive into domain knowledge will be used to look closely for any relationships in the raw data.

```
# Start the timer
start_time = time.time()

# Create Combined Ratio Feature
beer_quality_predict = beer_quality_predict.withColumn('Malt_Hops_Ratio', col('Malt_Ratio') / col('Hops_Ratio'))

# Stop the timer
end_time = time.time()

# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f'Data transformation execution time: {elapsed_time:.2f} seconds')

Data transformation execution time: 0.02 seconds
```

Summary

Given the uniform structure of this data set, let's see if any insights can be learned when applying domain knowledge to the dataset.

The ingredient ratio of water, malt, and hops is crucial in brewing. Chemistry shows that water forms the base of the mixture, malt provides the sugars needed for fermentation, and hops add bitterness and aroma. Typically, water is present in the highest ratio, followed by malt, with hops being in the smallest ratio. In this next analysis, we will create separate columns for the water, malt, and hops ratios, as well as a new feature column for the malt-to-hops ratio, to better understand their individual and combined effects on beer quality.

✓ Feature Engineering

We will infer new features based on existing data. The Malt_Type column will be created based on the beer's color, and the Hops_Type column will be derived from the beer's bitterness. These engineered features will then be one-hot encoded to prepare them for machine learning models.

```
# Start the timer
start_time = time.time()

# Function to infer malt type from color
beer_quality_predict = beer_quality_predict.withColumn(
    'Malt_Type',
    when(col('Color') < 10, 'Barley')
    .when((col('Color') >= 10) & (col('Color') < 20), 'Wheat')
    .when((col('Color') >= 20) & (col('Color') < 30), 'Rye')
    .otherwise('Oats')
)

# Function to infer hops type from bitterness
beer_quality_predict = beer_quality_predict.withColumn(
    'Hops_Type',
    when(col('Bitterness') > 40, 'Bittering')
    .when((col('Bitterness') > 20) & (col('Bitterness') <= 40), 'Dual Purpose')
    .otherwise('Aroma')
)

# Show the updated DataFrame with the new engineered features
beer_quality_predict.select('Color', 'Malt_Type', 'Bitterness', 'Hops_Type').show(5)

# One-hot encode categorical variables for machine learning models
categorical_columns = ['Beer_Style', 'Malt_Type', 'Hops_Type']

# Create a list to hold the stages of the Pipeline
stages = []

# Loop through each categorical column and create a StringIndexer and OneHotEncoder for each
for colm in categorical_columns:
    indexer = StringIndexer(inputCol=colm, outputCol=colm + '_Index')
    encoder = OneHotEncoder(inputCol=colm + '_Index', outputCol=colm + '_Vec')
    stages += [indexer, encoder]

# Apply the stages of the Pipeline to the DataFrame
pipeline = Pipeline(stages=stages)
beer_encoded = pipeline.fit(beer_quality_predict).transform(beer_quality_predict)

# Show the resulting DataFrame with the one-hot encoded columns
print("One-hot encoded dataframe:")
beer_encoded.select([colm for colm in beer_encoded.columns if 'Vec' in colm]).show()

# Preview the entire DataFrame
beer_encoded.show()

# Stop the timer
end_time = time.time()

# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f"Data transformation execution time: {elapsed_time:.2f} seconds")
```



```

beer_style_vec|malt_type_vec|hops_type_vec|
+-----+-----+-----+
| (7, [], []) | (1, [], []) | (2, [], []) |
| (7, [2], [1.0]) | (1, [0], [1.0]) | (2, [0], [1.0]) |
| (7, [], []) | (1, [0], [1.0]) | (2, [0], [1.0]) |
| (7, [0], [1.0]) | (1, [0], [1.0]) | (2, [1], [1.0]) |
| (7, [3], [1.0]) | (1, [0], [1.0]) | (2, [1], [1.0]) |
| (7, [0], [1.0]) | (1, [], []) | (2, [1], [1.0]) |
| (7, [5], [1.0]) | (1, [], []) | (2, [1], [1.0]) |
| (7, [], []) | (1, [], []) | (2, [0], [1.0]) |
| (7, [3], [1.0]) | (1, [0], [1.0]) | (2, [0], [1.0]) |
| (7, [3], [1.0]) | (1, [0], [1.0]) | (2, [1], [1.0]) |
| (7, [6], [1.0]) | (1, [0], [1.0]) | (2, [0], [1.0]) |
| (7, [4], [1.0]) | (1, [], []) | (2, [1], [1.0]) |
| (7, [2], [1.0]) | (1, [0], [1.0]) | (2, [0], [1.0]) |
| (7, [0], [1.0]) | (1, [], []) | (2, [0], [1.0]) |
| (7, [1], [1.0]) | (1, [0], [1.0]) | (2, [0], [1.0]) |
| (7, [6], [1.0]) | (1, [0], [1.0]) | (2, [0], [1.0]) |
| (7, [2], [1.0]) | (1, [0], [1.0]) | (2, [1], [1.0]) |
| (7, [1], [1.0]) | (1, [], []) | (2, [0], [1.0]) |
| (7, [6], [1.0]) | (1, [], []) | (2, [0], [1.0]) |
| (7, [3], [1.0]) | (1, [0], [1.0]) | (2, [0], [1.0]) |
+-----+-----+-----+

```

only showing top 20 rows

pH_Level	Ingredient_Ratio	Fermentation_Time	Temperature	Alcohol_Content	Bitterness	Color	Quality_Score
5.2898454476095615	1:0.32:0.16	16	24.204250857069873	5.370842159553436	20	5	8.57701633109
5.275643382756193	1:0.39:0.24	13	18.086762947259544	5.096053082797625	36	14	7.420540752553
4.7780156232459765	1:0.35:0.16	12	15.539332669116469	4.824737120959184	30	10	8.451364886803
5.345260585546188	1:0.35:0.15	17	16.41848910394318	5.509243080797997	48	18	9.671859404043
4.86185374113861	1:0.46:0.11	18	19.144907654338517	5.133624684263243	57	13	7.895333676172
5.291786621908058	1:0.23:0.15	10	17.424614393375766	4.859171021614226	45	9	8.47381192497
5.3328812410204325	1:0.34:0.16	16	15.629810875902074	4.710402522822796	44	8	6.958182685507
4.507213419450749	1:0.45:0.26	13	21.605469689190052	4.83702482975061	38	9	6.295632960105
5.224481542948347	1:0.39:0.19	18	20.183530075814367	5.133596661986578	25	15	9.660185081725
4.911261716318917	1:0.43:0.17	11	17.237974924919577	5.709899253603638	48	10	9.348719159247
4.809827440806769	1:0.27:0.16	14	15.70400745122817	5.460061910956435	26	15	7.652974518762
5.312024579912181	1:0.44:0.25	19	17.597701886470652	4.661066385691683	47	6	8.549107361500
5.422808722785356	1:0.30:0.24	14	15.415524753322972	4.516939869630938	34	16	8.31540186347
4.768508394883426	1:0.24:0.14	17	20.37375557905417	5.319366431771322	32	6	9.95303003672
4.876144319732823	1:0.50:0.28	17	24.678870169235424	4.97381389280548	34	15	8.046295862216
5.241133152001924	1:0.47:0.12	10	16.24653222310524	4.535179745113896	37	17	9.730922046183
5.473563790227288	1:0.31:0.21	11	22.903972245938505	5.12916808476057	44	10	9.167866793350
4.68579566157026	1:0.32:0.28	15	24.88502400214049	5.575448433172943	36	6	8.423886929535
5.318024913782187	1:0.23:0.16	15	17.785713321899884	5.406625556398892	23	5	9.181668960937
4.771957308753228	1:0.34:0.14	16	18.53433164072429	5.402760414393702	40	13	8.428914872818

only showing top 20 rows

Data transformation execution time: 6.09 seconds



Using reference literature on beer making, hops and malt types can be inferred based on the beer's color and bitterness. In brewing, the color of the beer, which ranges from pale yellow to dark brown, is primarily influenced by the type and amount of malt used. Malt provides fermentable sugars and contributes to the beer's flavor and body.

Hops, known for their bitterness and aromatic qualities, counterbalance the sweetness of the malt. The bitterness level, often measured in International Bitterness Units (IBUs), indicates the type of hops used. Chemistry plays a crucial role in understanding these properties, as different malts and hops impart specific flavors, aromas, and colors through their unique chemical compositions.

By analyzing these characteristics, brewers can predict the type of malt and hops used in a beer, enabling them to replicate or innovate on traditional recipes.



The temperature split turned out not to be a useful metric because the category values overlap so Beer_Style cannot be clearly delineated at this point.

Reference:

- 1 Ales: 62-75 °F (17-24 °C)
- 2 Lagers: 46-58 °F (8-14 °C) #assume dark lagers for this project
- 3 Wheat and Belgian styles: 62-85 °F (17-29 °C), ales are divided into these categories

Next, let's take a look at whether or not the Beer_Style feature distribute consistently into the Fermentation_Time feature.

```
# Start the timer
start_time = time.time()

# Extract relevant columns
ferment_style_comp = beer_quality_predict.select('Beer_Style', 'Fermentation_Time')

# Convert to Pandas DataFrame for plotting
ferment_style_comp_pd = ferment_style_comp.toPandas()

# Create a box plot
plt.figure(figsize=(14, 8))
sns.boxplot(x='Beer_Style', y='Fermentation_Time', data=ferment_style_comp_pd)
plt.title('Distribution of Fermentation Time by Beer Style')
plt.xticks(rotation=45)

# Save the plot as an image with better layout
filename = "fermentation_style_box_plot.jpg"
plt.savefig(filename, bbox_inches='tight')

# Download the file
files.download("fermentation_style_box_plot.jpg")

# Show the plot
plt.show()

# Create a violin plot
plt.figure(figsize=(14, 8))
sns.violinplot(x='Beer_Style', y='Fermentation_Time', data=ferment_style_comp_pd)
plt.title('Distribution of Fermentation Time by Beer Style')
plt.xticks(rotation=45)

# Save the plot as an image with better layout
filename = "fermentation_style_violin_plot.jpg"
plt.savefig(filename, bbox_inches='tight')

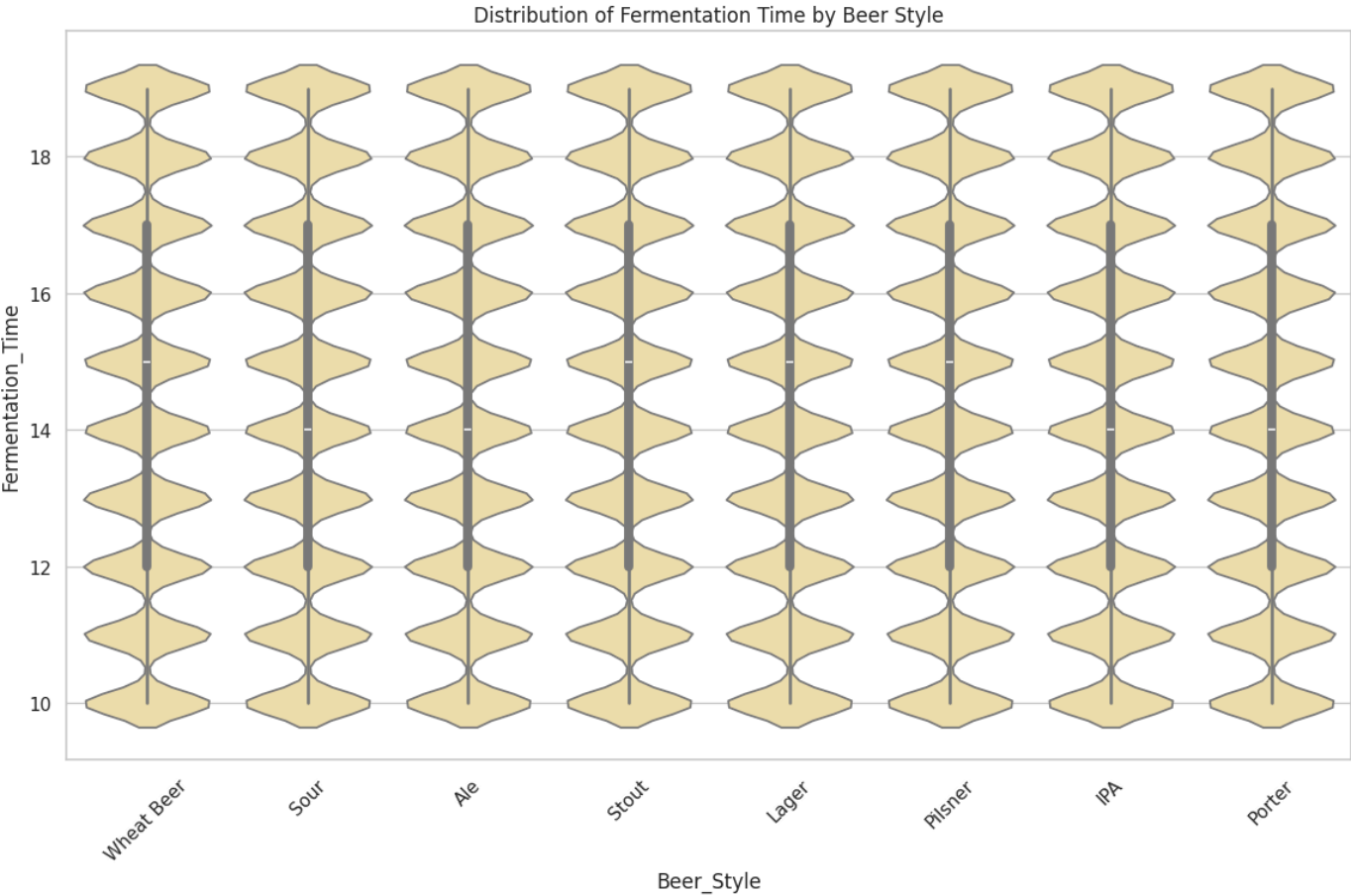
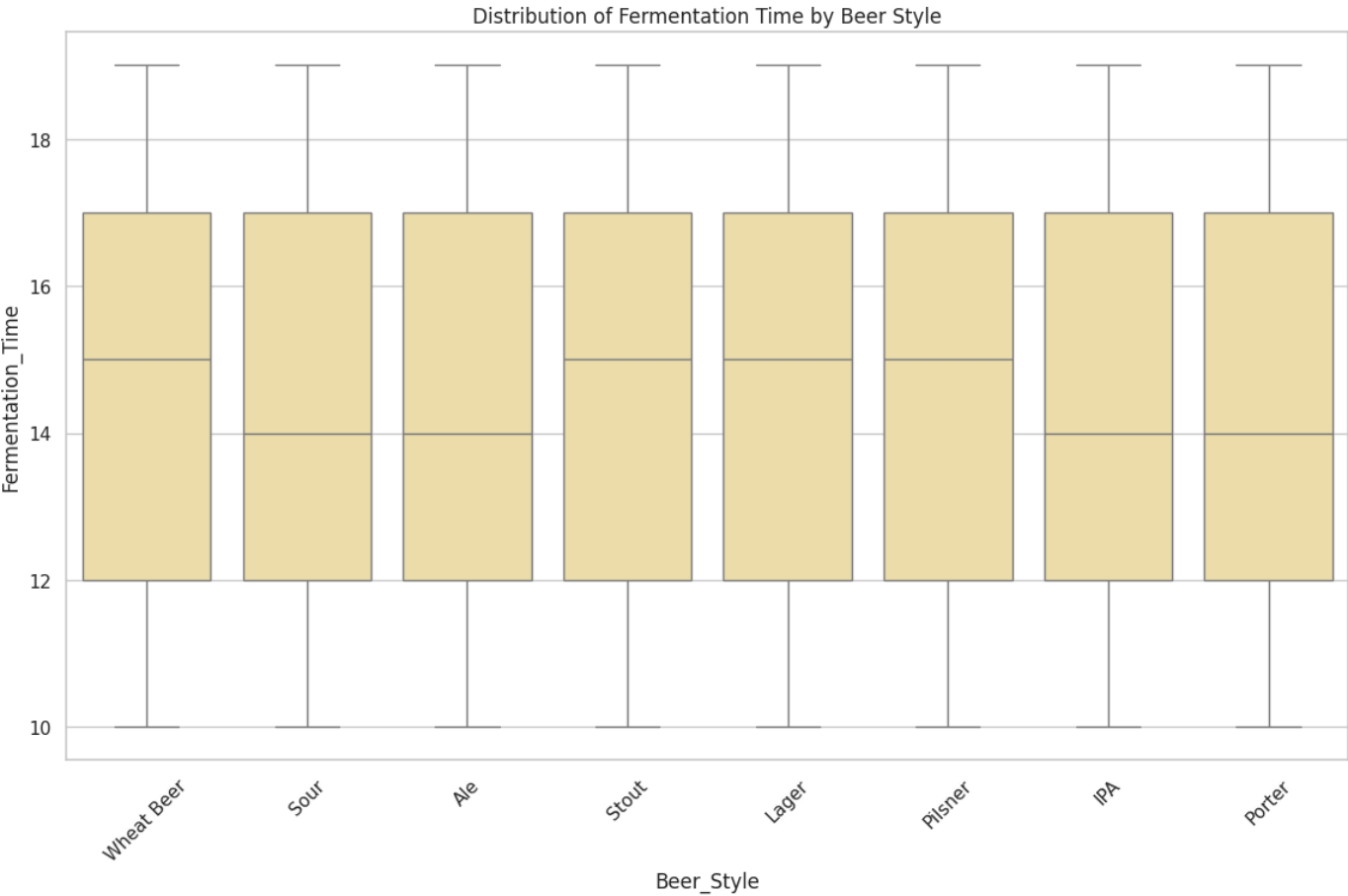
# Download the file
files.download("fermentation_style_violin_plot.jpg")

# Show the plot
plt.show()

# Stop the timer
end_time = time.time()

# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f"Data transformation execution time: {elapsed_time:.2f} seconds")
```



Data transformation execution time: 70.60 seconds



Box Plot:

This box plot shows the distribution of fermentation time across different beer styles. Box plots are useful for visualizing the spread and central tendency of the data, as well as identifying any potential outliers.

Here are the box plot key points:

X-Axis (Beer Style): Lists the different types of beer styles, such as Wheat Beer, Sour, Ale, Stout, Lager, Pilsner, IPA, and Porter.

Y-Axis (Fermentation Time): Represents the fermentation time in days.

Median Fermentation Time: The horizontal line within each box represents the median fermentation time for each beer style. Across the beer styles, the median fermentation time is relatively consistent, ranging between 14 and 16 days.

Interquartile Range (IQR): The boxes represent the middle 50% of the data (from the 25th percentile to the 75th percentile). The IQR for each beer style shows some variation, but overall, most beer styles have an IQR that spans from approximately 13 to 16.5 days.

Whiskers: The lines (whiskers) extending from the boxes indicate the range of fermentation times, excluding outliers. The whiskers suggest that the majority of the fermentation times fall between 10 and 18 days for all beer styles.

Outliers: There do not appear to be any significant outliers in the data, as no points are plotted outside the whiskers.

Conclusion: The plot shows that while there is some variation in fermentation times across different beer styles, the range is generally consistent. Most beer styles have a median fermentation time around 15 days, with fermentation times ranging from approximately 10 to 18 days. This suggests that fermentation time is relatively standardized across different styles of beer.

Violin Plot

The violin plot shows that fermentation times are relatively consistent across different beer styles, with most styles having a median fermentation time around 15 days. Violin plots combine aspects of box plots and density plots to provide more information about the distribution of data.

Here are the box plot key points:

X-Axis (Beer Style): Lists the different types of beer styles, such as Wheat Beer, Sour, Ale, Stout, Lager, Pilsner, IPA, and Porter.

Y-Axis (Fermentation Time): Represents the fermentation time in days.

Distribution Shape: The width of each violin represents the density of data points at different fermentation times for each beer style. Wider sections indicate higher density (more common fermentation times), while narrower sections indicate lower density.

Central Tendency: The white dot in the center of each violin represents the median fermentation time for that beer style. The median is relatively consistent across beer styles, around 15 days.

Range of Fermentation Time: The plots are generally symmetrical, indicating a uniform distribution of fermentation times. The range of fermentation time for most beer styles is between 10 and 18 days.

Density Peaks: Some beer styles show distinct peaks in density, indicating specific fermentation times that are more common. For example, Lager, Pilsner, and IPA show a relatively wider distribution near the median, suggesting a more even spread of fermentation times around the median value.

Summary

This discrete data is based on all 10 million data points and shows that Lager and IPA have an increased fermentation time due to the chemistry involved. Lager yeasts ferment at lower temperatures and more slowly, allowing for the development of clean, crisp flavors. IPAs, with their higher hop content, require extended fermentation to balance the bitterness and develop the complex hop aromas and flavors through biochemical reactions.

Generally, the plots show that the fermentation times for different beer styles are consistent; with median values around 15-16 days and a range between approximately 10 and 18 days. This is evident from the box plot and violin plot, which both show similar distributions across beer styles. The similar interquartile ranges indicate that the central 50% of the data points for each style lie within a comparable range. The density of data points around the median values, as shown in the violin plot, reinforces this observation.

This data suggests that while specific styles like Lager and IPA may have increased fermentation times due to their unique chemistry, the overall process duration is quite uniform across various beer styles.

✓ Unsupervised Learning

This experiment explores clustering models with a K-means experiment. To determine the optimal cluster grouping number inertia and silhouette scores will be used.

```

# Start the timer
start_time = time.time()

# Select relevant features for clustering
clustering_features = ['Temperature', 'Fermentation_Time', 'pH_Level', 'Alcohol_Content', 'Bitterness', 'Color']

# Assemble the features into a single vector column
assembler = VectorAssembler(inputCols=clustering_features, outputCol="features")
beer_assembled = assembler.transform(beer_encoded)

# Normalize the features
scaler = StandardScaler(inputCol="features", outputCol="scaled_features", withStd=True, withMean=True)
scaler_model = scaler.fit(beer_assembled)
beer_scaled = scaler_model.transform(beer_assembled)

# Cache the scaled DataFrame to speed up clustering iterations
beer_scaled.cache()

# Define the range of clusters to test; based on previous experiments
min_clusters = 2
max_clusters = 40

# Store inertia and silhouette scores for different cluster sizes
inertia_values = []
silhouette_scores = []

# ClusteringEvaluator to calculate silhouette scores
evaluator = ClusteringEvaluator(featuresCol='scaled_features', metricName='silhouette', distanceMeasure='squaredEuclidean')

# Iterate over the range of clusters
for k_clusters in range(min_clusters, max_clusters + 1):
    kmeans = KMeans(k=k_clusters, seed=42, featuresCol="scaled_features")
    kmeans_model = kmeans.fit(beer_scaled)

    # Make predictions
    predictions = kmeans_model.transform(beer_scaled)

    # Append inertia (sum of squared distances to nearest cluster center)
    inertia_values.append(kmeans_model.summary.trainingCost)

    # Calculate silhouette score
    silhouette_avg = evaluator.evaluate(predictions)
    silhouette_scores.append(silhouette_avg)

# Print the scores for this number of clusters
print(f"Number of clusters: {k_clusters}, Inertia: {inertia_values[-1]:.2f}, Silhouette Score: {silhouette_avg:.4f}")

# Plot the inertia values to visualize the optimal number of clusters
plt.figure(figsize=(10, 6))
plt.plot(range(min_clusters, max_clusters + 1), inertia_values, marker='o', linestyle='-', color='#d98416', label='Inertia')
plt.xlabel('Number of Clusters')
plt.ylabel('Inertia')
plt.title('Elbow Method for Optimal Number of Clusters')
plt.xticks(range(min_clusters, max_clusters + 1))
plt.grid(True)
plt.legend()

# Save the plot as an image with better layout
filename = "elbow_cluster_plot.jpg"
plt.savefig(filename, bbox_inches='tight')

# Show the plot
plt.show()

# Plot the silhouette scores to further validate the optimal number of clusters
plt.figure(figsize=(10, 6))
plt.plot(range(2, max_clusters + 1), silhouette_scores, marker='o', linestyle='-', color='#d98416', label='Silhouette Score')
plt.xlabel('Number of Clusters')
plt.ylabel('Silhouette Score')
plt.title('Silhouette Scores for Different Numbers of Clusters')
plt.xticks(range(2, max_clusters + 1))
plt.grid(True)
plt.legend()

# Save the plot as an image with better layout
filename = "silhouette_cluster_plot.jpg"
plt.savefig(filename, bbox_inches='tight')

```

```
# Show the plot
plt.show()

# Stop the timer
end_time = time.time()

# Calculate the elapsed time
elapsed_time = end_time - start_time

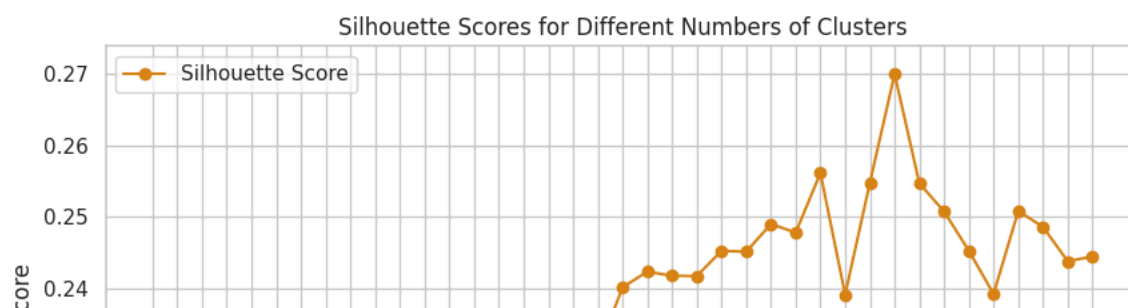
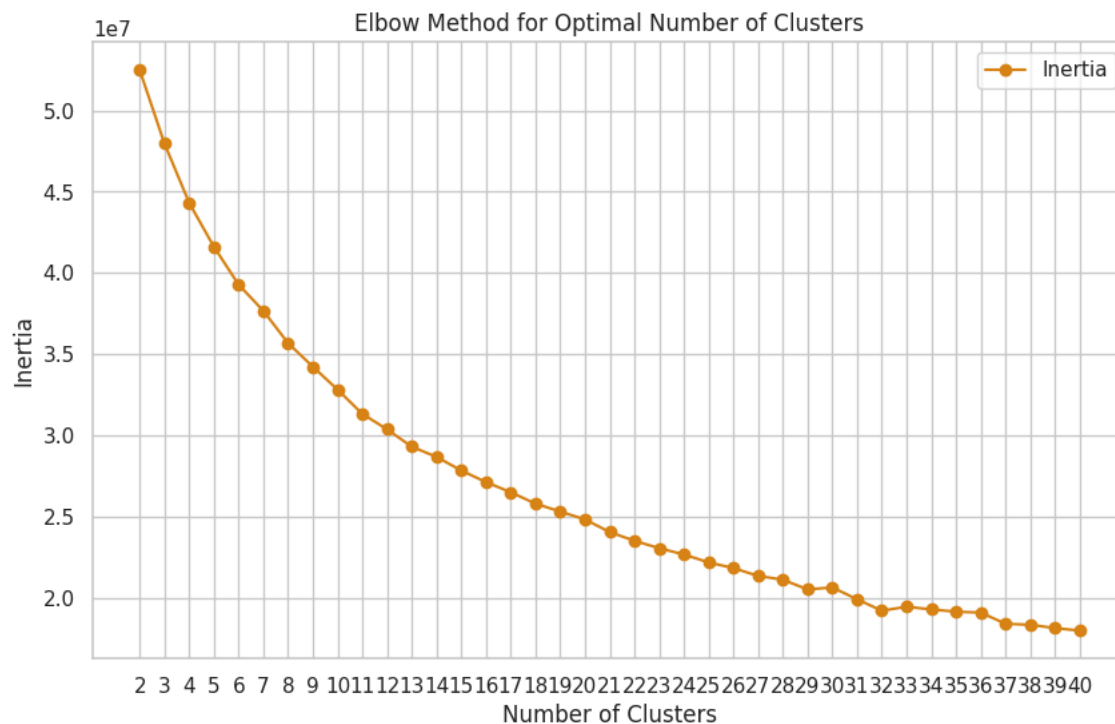
# Print the elapsed time
print(f>Data transformation execution time: {elapsed_time:.2f} seconds")
```

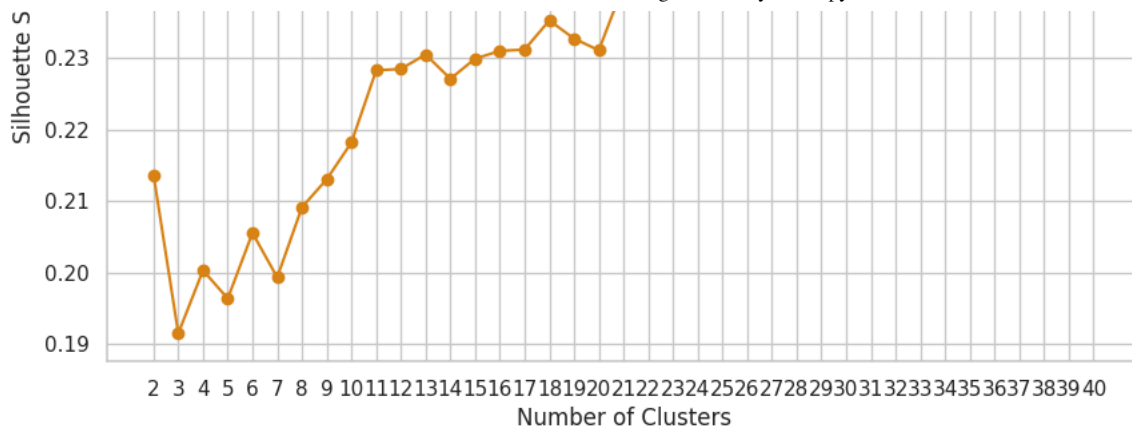


```

Number of clusters: 2, Inertia: 52501297.28, Silhouette Score: 0.2137
Number of clusters: 3, Inertia: 47974139.18, Silhouette Score: 0.1915
Number of clusters: 4, Inertia: 44320283.81, Silhouette Score: 0.2003
Number of clusters: 5, Inertia: 41614067.07, Silhouette Score: 0.1964
Number of clusters: 6, Inertia: 39281235.53, Silhouette Score: 0.2056
Number of clusters: 7, Inertia: 37664870.06, Silhouette Score: 0.1993
Number of clusters: 8, Inertia: 35677312.61, Silhouette Score: 0.2091
Number of clusters: 9, Inertia: 34228089.81, Silhouette Score: 0.2130
Number of clusters: 10, Inertia: 32813952.07, Silhouette Score: 0.2182
Number of clusters: 11, Inertia: 31321035.40, Silhouette Score: 0.2283
Number of clusters: 12, Inertia: 30369568.92, Silhouette Score: 0.2285
Number of clusters: 13, Inertia: 29299052.04, Silhouette Score: 0.2305
Number of clusters: 14, Inertia: 28670062.20, Silhouette Score: 0.2271
Number of clusters: 15, Inertia: 27833784.57, Silhouette Score: 0.2299
Number of clusters: 16, Inertia: 27117094.19, Silhouette Score: 0.2310
Number of clusters: 17, Inertia: 26495972.75, Silhouette Score: 0.2312
Number of clusters: 18, Inertia: 25790569.14, Silhouette Score: 0.2353
Number of clusters: 19, Inertia: 25306040.98, Silhouette Score: 0.2327
Number of clusters: 20, Inertia: 24817819.77, Silhouette Score: 0.2311
Number of clusters: 21, Inertia: 24041825.95, Silhouette Score: 0.2401
Number of clusters: 22, Inertia: 23504209.13, Silhouette Score: 0.2423
Number of clusters: 23, Inertia: 23046486.79, Silhouette Score: 0.2417
Number of clusters: 24, Inertia: 22659821.12, Silhouette Score: 0.2417
Number of clusters: 25, Inertia: 22171427.77, Silhouette Score: 0.2452
Number of clusters: 26, Inertia: 21822079.33, Silhouette Score: 0.2451
Number of clusters: 27, Inertia: 21341106.81, Silhouette Score: 0.2490
Number of clusters: 28, Inertia: 21106249.08, Silhouette Score: 0.2478
Number of clusters: 29, Inertia: 20507810.59, Silhouette Score: 0.2561
Number of clusters: 30, Inertia: 20629668.56, Silhouette Score: 0.2391
Number of clusters: 31, Inertia: 19903230.18, Silhouette Score: 0.2548
Number of clusters: 32, Inertia: 19196114.00, Silhouette Score: 0.2700
Number of clusters: 33, Inertia: 19442675.80, Silhouette Score: 0.2547
Number of clusters: 34, Inertia: 19276390.99, Silhouette Score: 0.2507
Number of clusters: 35, Inertia: 19134725.38, Silhouette Score: 0.2452
Number of clusters: 36, Inertia: 19084151.86, Silhouette Score: 0.2392
Number of clusters: 37, Inertia: 18393238.99, Silhouette Score: 0.2508
Number of clusters: 38, Inertia: 18329202.18, Silhouette Score: 0.2486
Number of clusters: 39, Inertia: 18124599.65, Silhouette Score: 0.2438
Number of clusters: 40, Inertia: 17972818.38, Silhouette Score: 0.2444

```





Data transformation execution time: 7630.00 seconds



Through analysis, 32 has been identified as the optimal number of clusters. The Inertia score shows a plateau, indicating minimal improvement in compactness of clusters beyond 32. Additionally, the Silhouette score peaks at 32, indicating the highest average separation between clusters.

The calculations support each other and the determination that 32 is the optimal number of clusters.

```
# Start the timer
start_time = time.time()

# Select relevant features for clustering
clustering_features = ['Temperature', 'Fermentation_Time', 'pH_Level', 'Alcohol_Content', 'Bitterness', 'Color']

# VectorAssembler to combine feature columns into a single vector column
assembler = VectorAssembler(inputCols=clustering_features, outputCol='features')
beer_assembled = assembler.transform(beer_encoded)

# Normalize the features
scaler = StandardScaler(inputCol="features", outputCol="scaled_features", withStd=True, withMean=True)
scaler_model = scaler.fit(beer_assembled)
beer_scaled = scaler_model.transform(beer_assembled)

# Apply KMeans clustering
num_clusters = 32 # Specified number of clusters based on previous experiments
kmeans = KMeans(k=num_clusters, seed=42, featuresCol='scaled_features', predictionCol='Cluster')
kmeans_model = kmeans.fit(beer_scaled)

# Transform the data using the fitted KMeans model
beer_clustered = kmeans_model.transform(beer_scaled)

# Apply PCA
pca = PCA(k=2, inputCol="scaled_features", outputCol="pcaFeatures")
pca_model = pca.fit(beer_clustered)
beer_pca = pca_model.transform(beer_clustered)

# Convert to Pandas for visualization
beer_pca_pd = beer_pca.select("pcaFeatures", "Cluster").toPandas()
beer_pca_pd[['pca1', 'pca2']] = pd.DataFrame(beer_pca_pd['pcaFeatures'].tolist(), index=beer_pca_pd.index)

# Create a continuous colormap based on your custom color palette
custom_cmap = LinearSegmentedColormap.from_list("custom_cmap", custom_colors, N=num_clusters)

# Plot the PCA results with clusters
plt.figure(figsize=(10, 6))
scatter = plt.scatter(beer_pca_pd['pca1'], beer_pca_pd['pca2'], c=beer_pca_pd['Cluster'], cmap=custom_cmap)
plt.colorbar(scatter)
plt.title('PCA of K-means Clustering of Beer Quality Data')
plt.xlabel('PCA Component 1')
plt.ylabel('PCA Component 2')

# Save the plot as an image with better layout
filename = "PCA_clustering.jpg"
plt.savefig(filename, bbox_inches='tight')

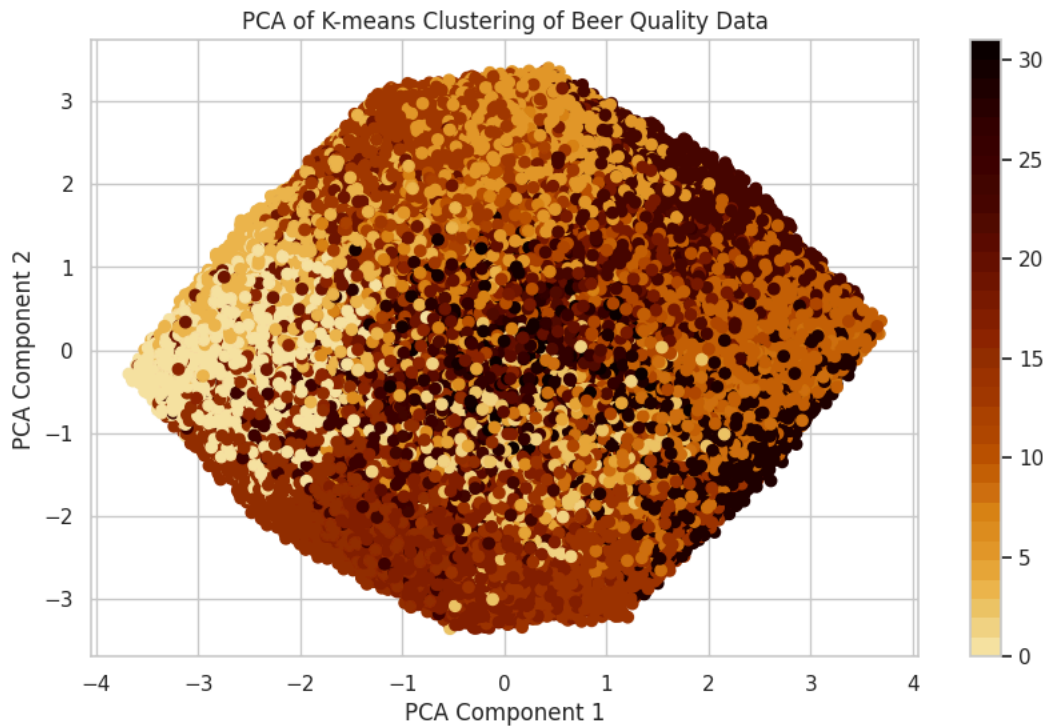
# Download the file
files.download("PCA_clustering.jpg")

# Show the plot
plt.show()

# Stop the timer
end_time = time.time()

# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f>Data transformation execution time: {elapsed_time:.2f} seconds")
```



This plot represents the Principal Component Analysis (PCA) of K-means clustering applied to the entire dataset. The PCA reduces the dimensionality of the data, projecting it onto two principal components, allowing the visualization of clustering results in a 2D space. The color gradient indicates the quality score, with lighter colors representing lower quality scores and darker colors representing higher quality scores.

X-Axis (PCA Component 1): Represents the first principal component, which captures the maximum variance in the data after dimensionality reduction.

Y-Axis (PCA Component 2): Represents the second principal component, capturing the second most significant variance.

Several key observations can be made from this visualization:

Cluster Density and Overlap:

The plot shows a dense central region with data points scattered around, forming a diamond shape. This suggests that the beer quality data has a central concentration of similar characteristics, with less variation moving towards the edges.

Quality Distribution:

The color gradient suggests that certain regions within the PCA space are associated with higher or lower values of the metric used for coloring (likely cluster assignment). This implies that the K-means clustering has identified distinct groupings or clusters within the data, which are visually separable when reduced to two dimensions.

PCA Components Interpretation:

The PCA components represent combinations of the original features that capture the most variance in the data. The use of PCA here helps to simplify the complex, high-dimensional beer quality data into a two-dimensional plot, making it easier to observe patterns, clusters, or trends.

Further analysis is needed to interpret the specific characteristics of each cluster and their implications for beer quality.



A note on the method:

When comparing PCA and t-SNE for dimensionality reduction, PCA is faster but tends to produce more discrete dots in the visualization. In contrast, t-SNE, though slower, often shows more distinct clusters. This difference arises because PCA captures the variance in the data, making it effective for linear separations, while t-SNE focuses on preserving local structures, making it better at identifying clusters.

K-Means clustering works effectively with numerical variables, as it relies on the distances between data points. Categorical variables do not imply order, which may produce unreliable or misleading information. Future research should explore strategies for integrating numerical and categorical data more effectively in clustering.

*In the context of this flat and uniform dataset, **the initial clustering results are promising** and suggest a path toward deeper insights.*

K-Means is the better choice for this project because scalability and speed are primary concerns. In the future, HAC (Hierarchical Agglomerative Clustering) could offer a deeper understanding the hierarchial relationships in the data if computational resources were available.

```
# Start the timer
start_time = time.time()

# Calculate the median quality score for each cluster
cluster_medians = beer_clustered.groupBy('Cluster').agg(expr('percentile_approx(Quality_Score, 0.5)').alias('median_Quality_Score'))
cluster_order = [row['Cluster'] for row in cluster_medians.orderBy('median_Quality_Score').collect()]

# Create a new column 'Cluster_Order' with the ordered cluster categories
cluster_mapping_expr = when(col('Cluster') == cluster_order[0], lit(0))
for i, cluster in enumerate(cluster_order[1:], 1):
    cluster_mapping_expr = cluster_mapping_expr.when(col('Cluster') == cluster, lit(i))

beer_clustered = beer_clustered.withColumn('Cluster_Order', cluster_mapping_expr)

# Convert the 'Cluster_Order' column to integer type
beer_clustered = beer_clustered.withColumn('Cluster_Order', col('Cluster_Order').cast('int'))

# Take a (1 million data) sample of the data before converting to Pandas
sample_fraction = 0.1 # Adjust this value to control the sample size (e.g., 0.1 for 10%)
beer_clustered_sample_pd = beer_clustered.sample(fraction=sample_fraction, seed=42)

# Convert the DataFrame to Pandas for plotting
beer_clustered_pd = beer_clustered_sample_pd.toPandas()

# Ensure the DataFrame is sorted by Cluster_Order
beer_clustered_pd = beer_clustered_pd.sort_values(by='Cluster_Order')

# Create a continuous colormap based on your custom color palette
num_clusters = len(beer_clustered_pd['Cluster_Order'].unique())
custom_cmap = LinearSegmentedColormap.from_list("custom_cmap", custom_colors, N=num_clusters)

# Plot the box plots using the ordered clusters
features_to_plot = ['Quality_Score', 'Temperature', 'Fermentation_Time', 'pH_Level', 'Alcohol_Content', 'Bitterness', 'Color']

for feature in features_to_plot:
    plt.figure(figsize=(14, 10))
    sns.boxplot(x='Cluster_Order', y=feature, data=beer_clustered_pd, hue='Cluster_Order', palette=custom_cmap, legend=False)
    plt.title(f'Distribution of {feature} by Cluster Ordered by Quality Score')
    plt.xlabel('Cluster')
    plt.ylabel(feature)

    # Save the plot as an image with better layout
    filename = "cluster_box_plots.jpg"
    plt.savefig(filename, bbox_inches='tight')

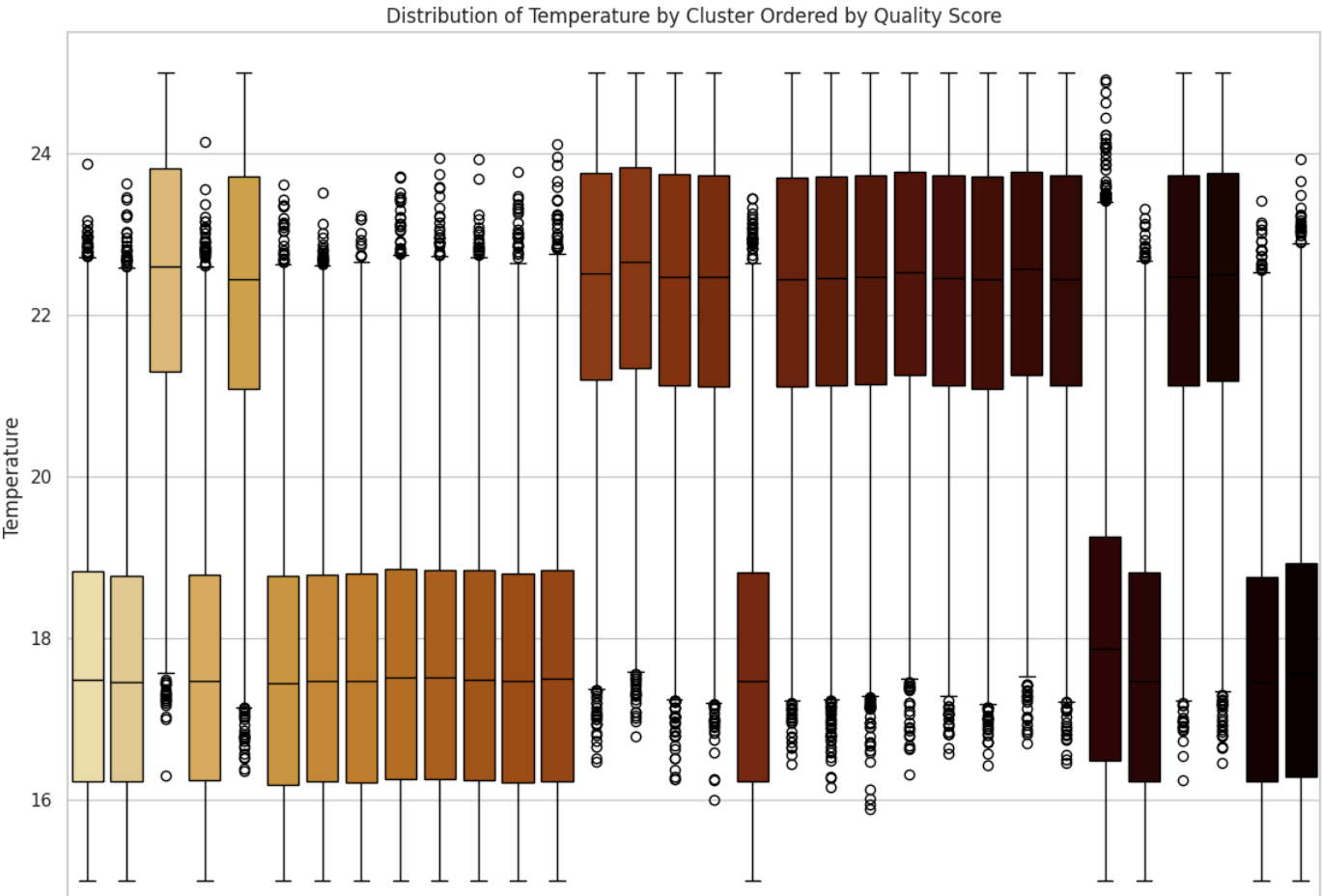
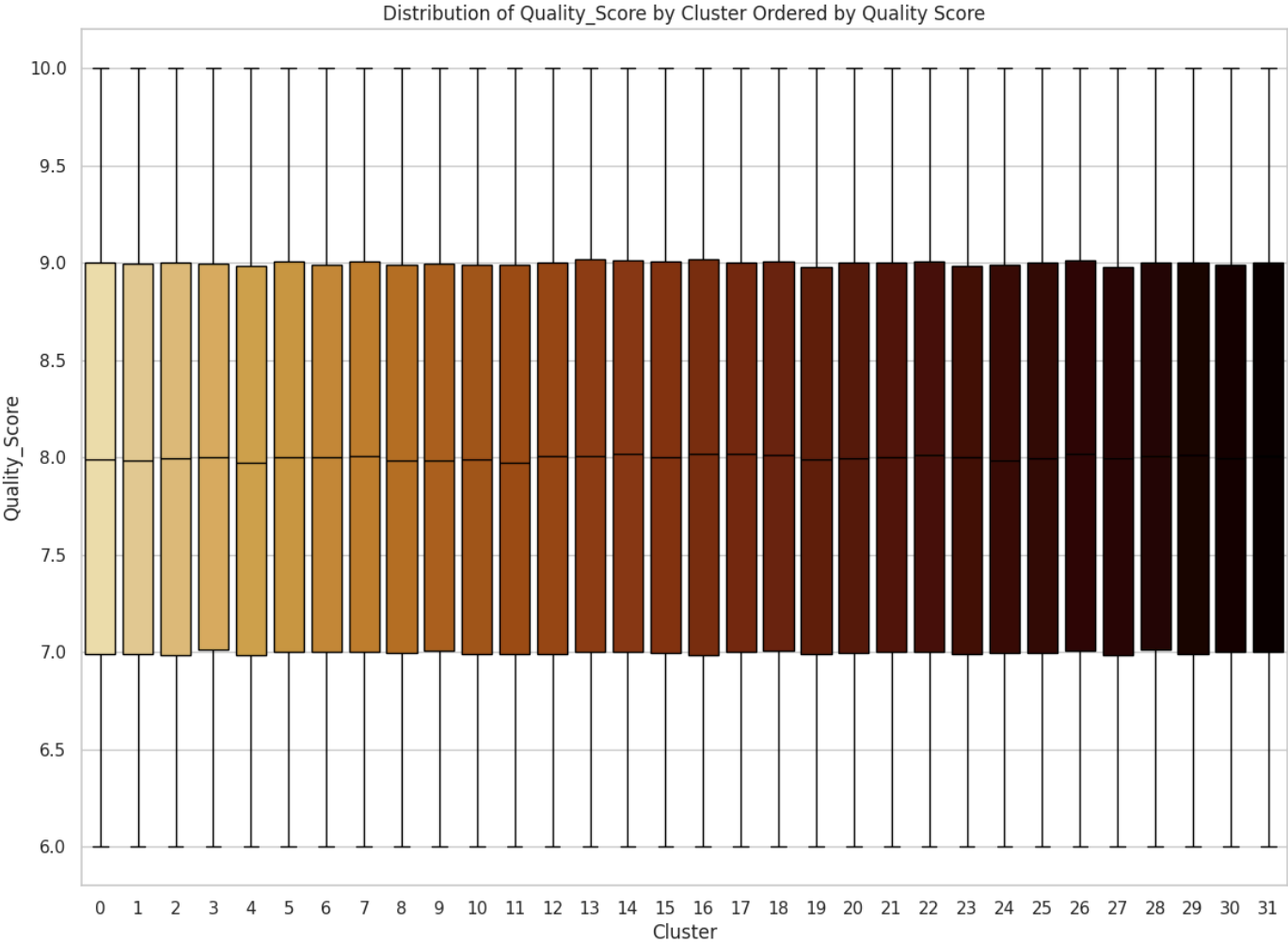
    # Download the file
    files.download("cluster_box_plots.jpg")

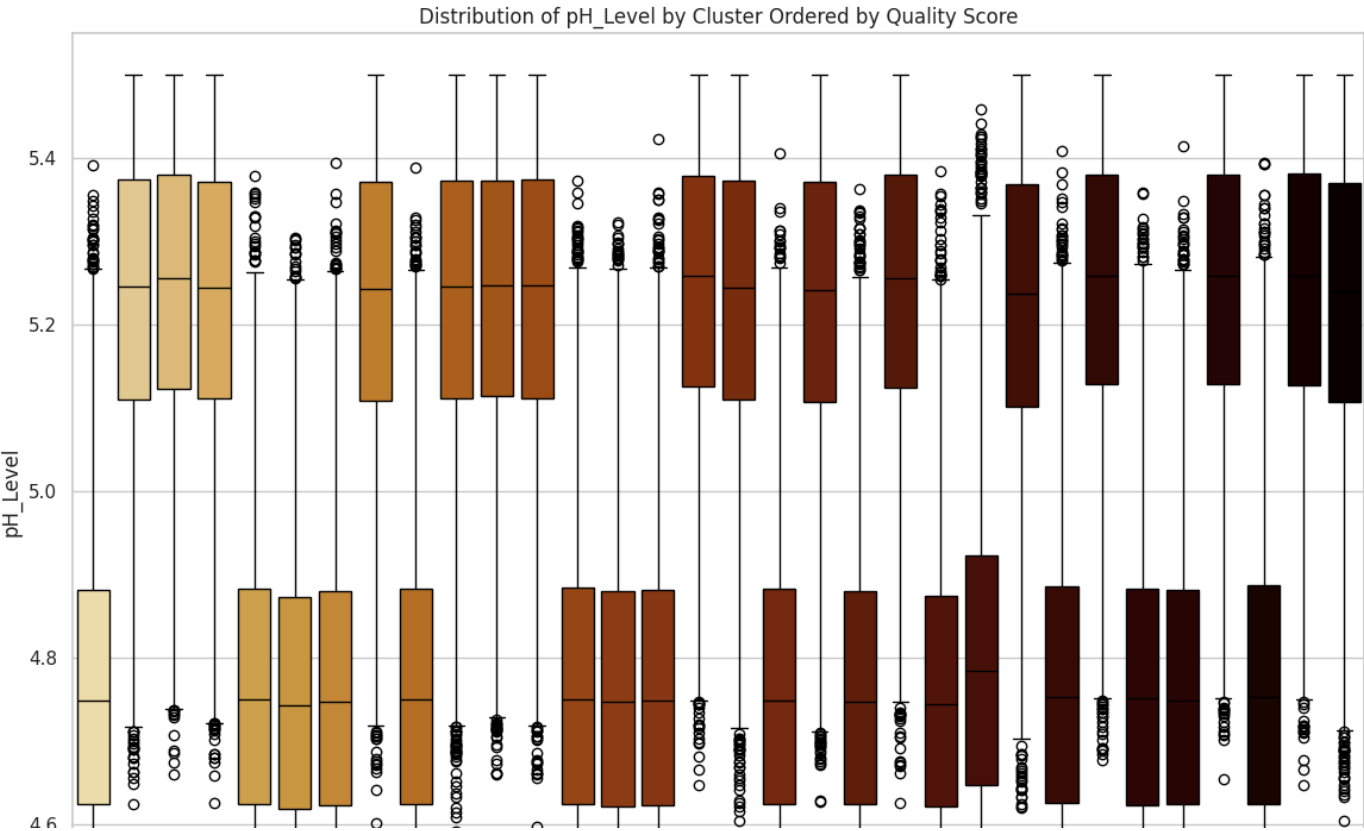
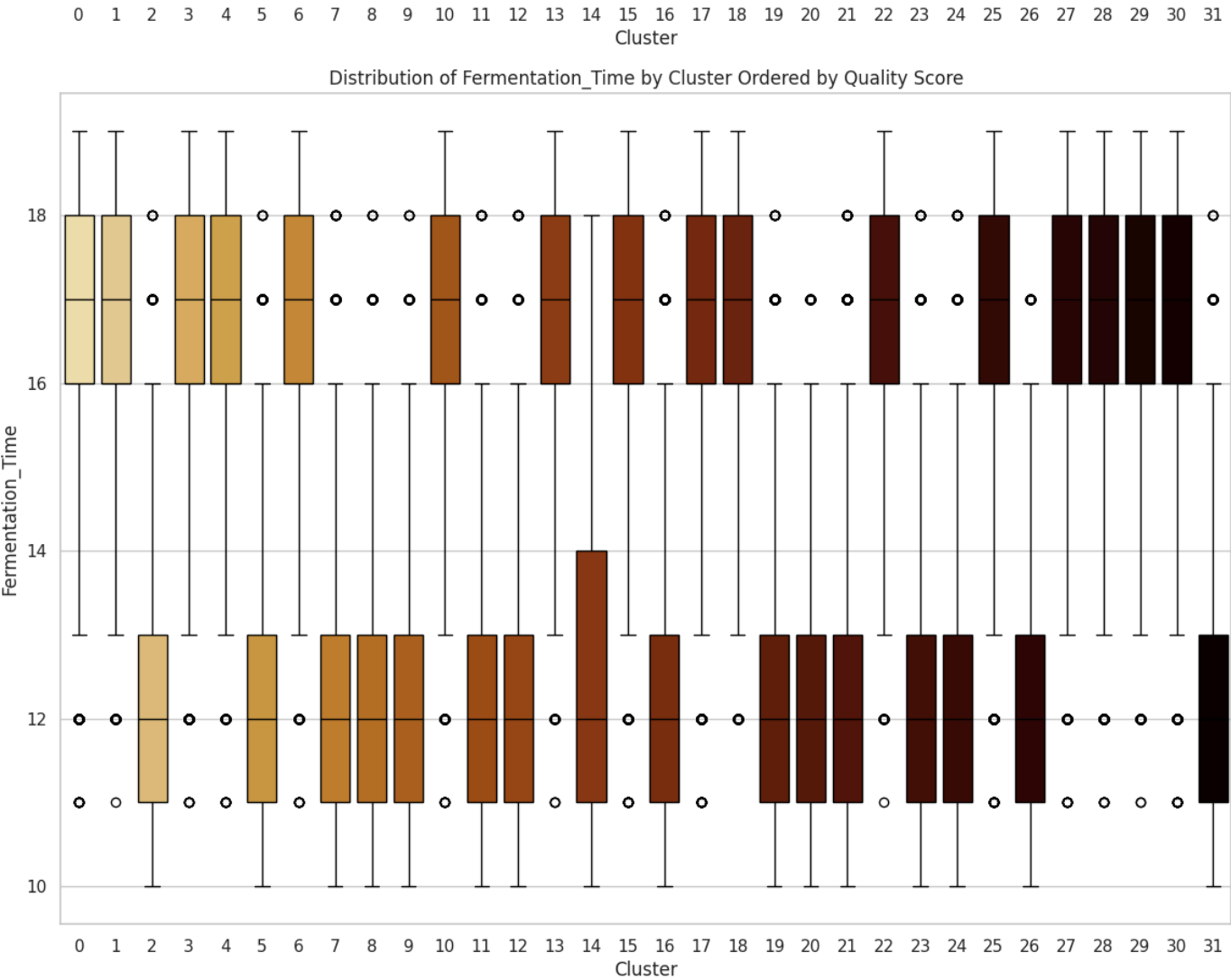
    # Show the plot
    plt.show()

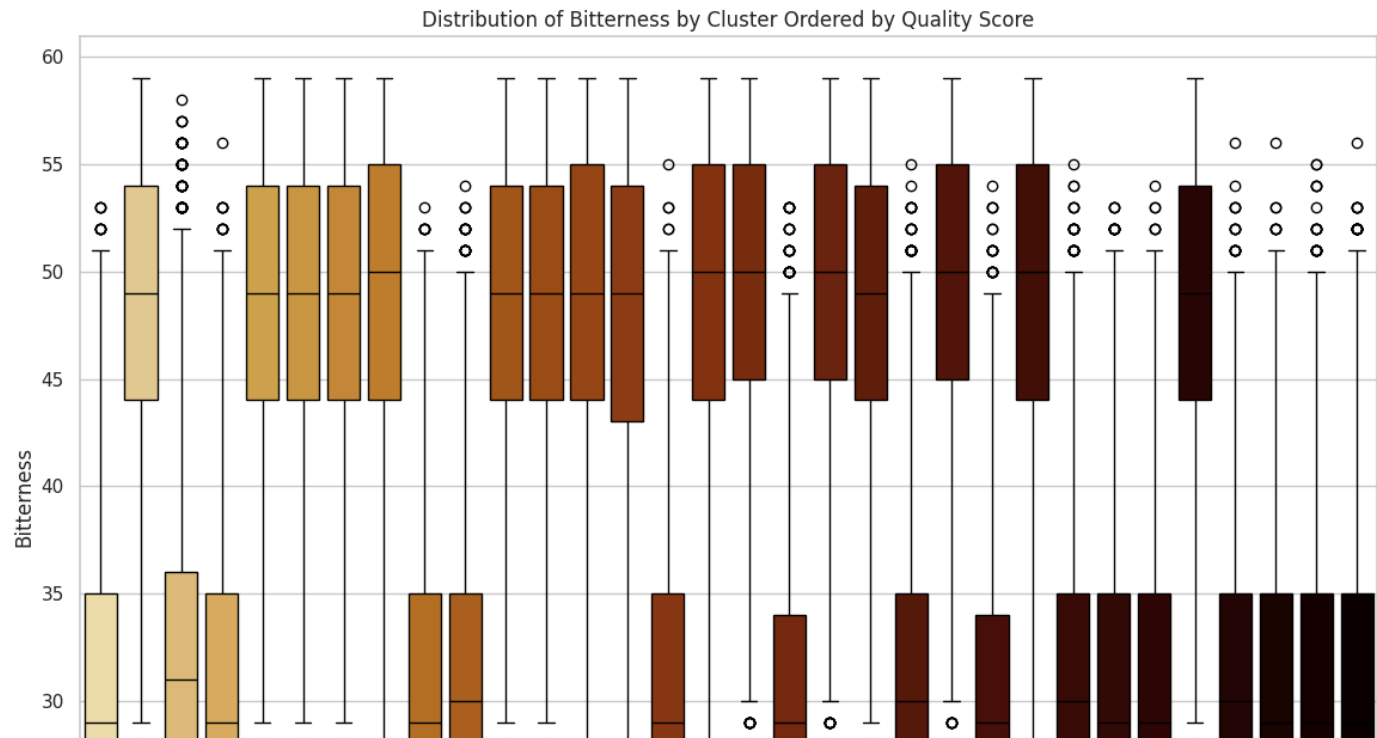
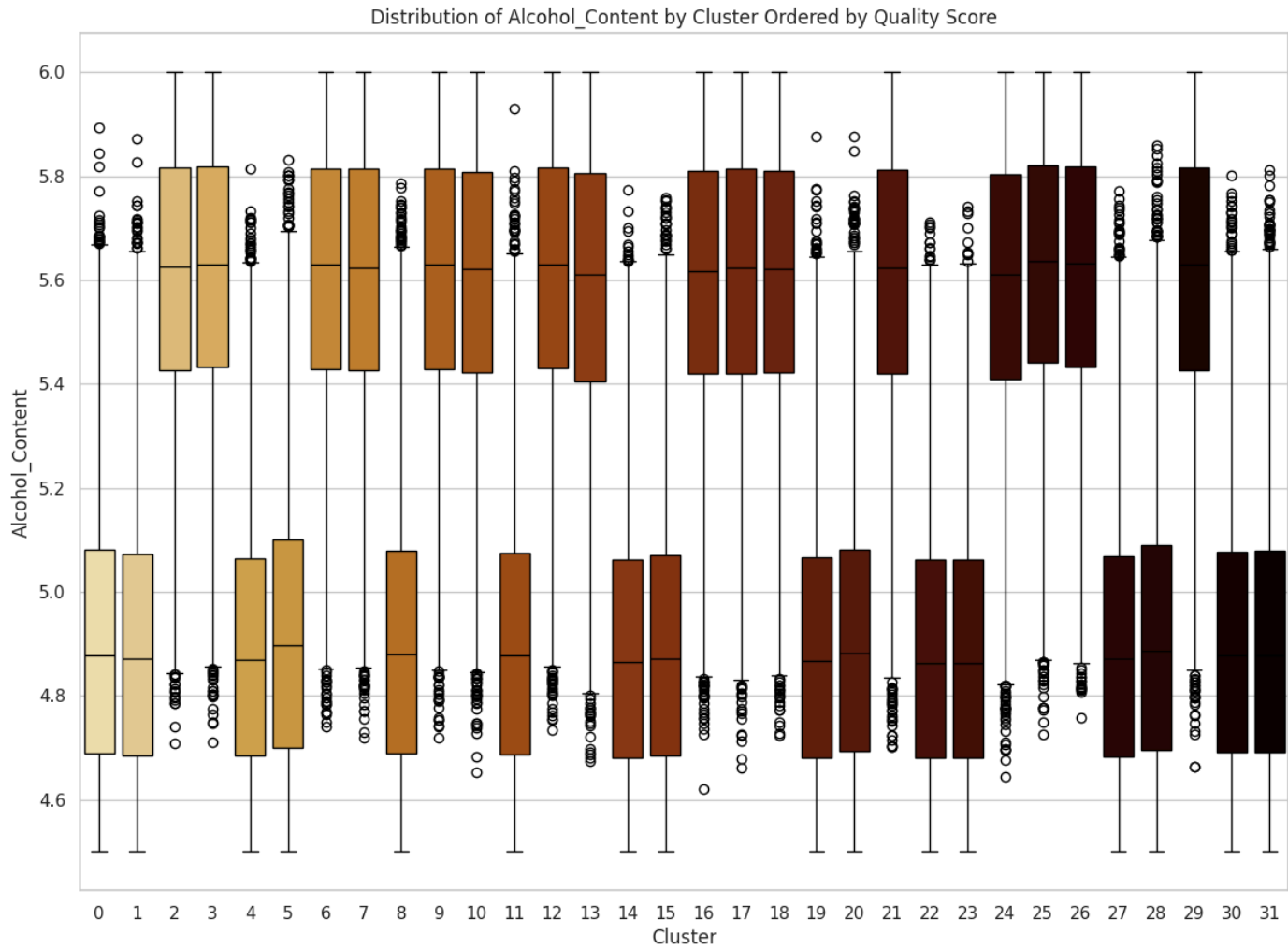
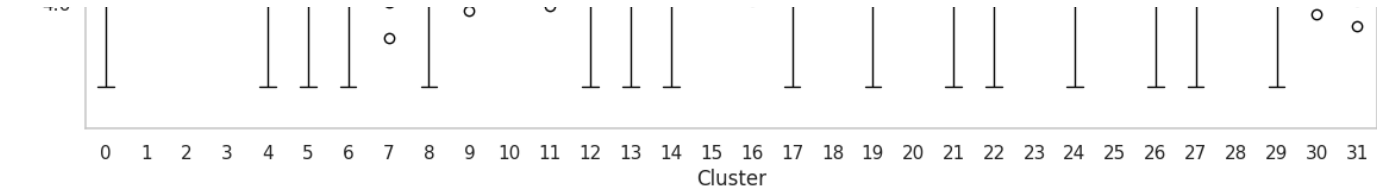
# Stop the timer
end_time = time.time()

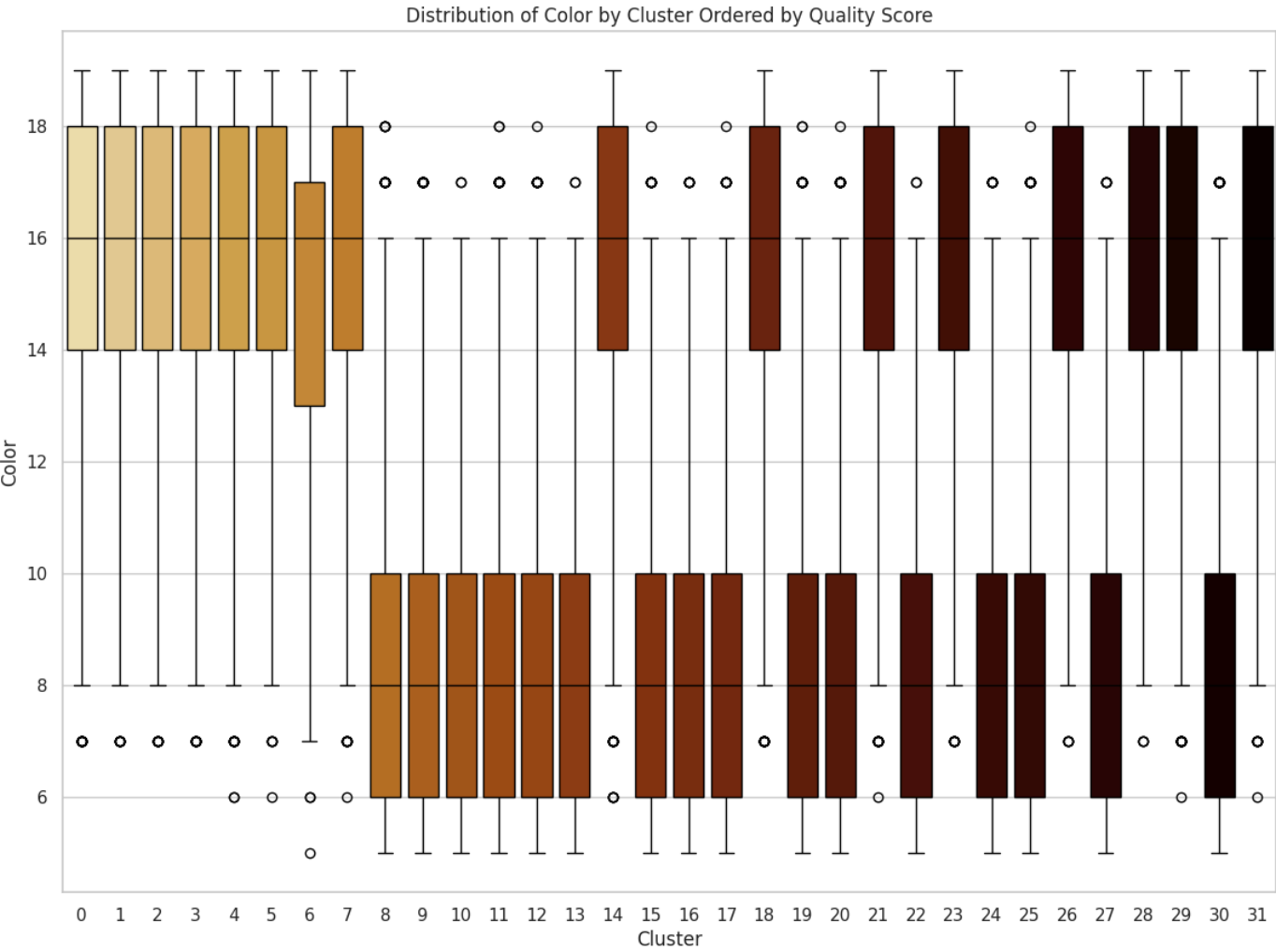
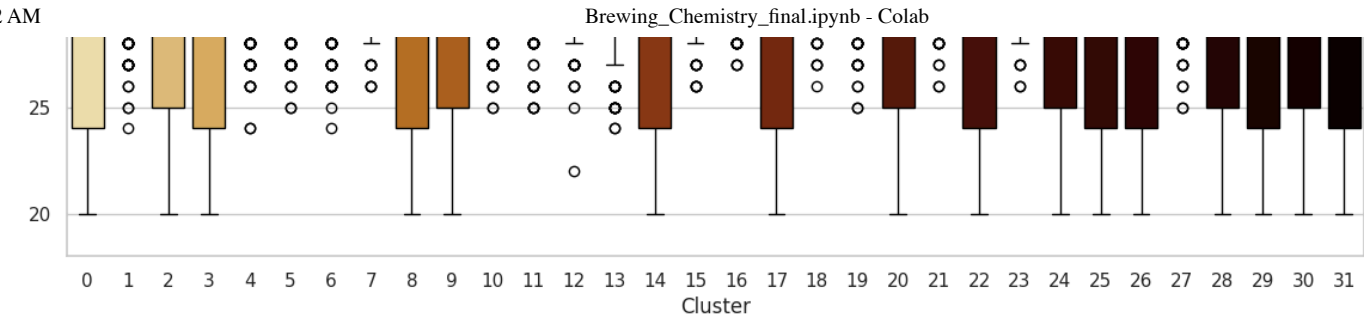
# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f"Data transformation execution time: {elapsed_time:.2f} seconds")
```









Data transformation execution time: 167.44 seconds



This box plot orders the clusters by the median beer quality to visually and comparably order the clusters to potentially separate any clusters that showed either remarkable qualities or completely undesirable qualities.

The boxes represent the interquartile range (IQR), which contains the middle 50% of the data points. The top and bottom edges of the box indicate the 75th and 25th percentiles, respectively. The vertical lines extending from the boxes (whiskers) represent the range of the data, excluding outliers. They extend to 1.5 times the IQR. The circles outside the whiskers represent outliers—data points that fall outside the expected range (more than 1.5 times the IQR from the quartiles).

Generally, the feature box plot distributions by cluster ordered by quality score showed a bimodal distribution. Indicating that some more structure in the data is becoming visible.

Observations: Quality Score Distribution: Overall, the quality scores are relatively consistent across clusters, with most clusters having median scores between 7.5 and 8.5. This consistency suggests that certain characteristics, like alcohol content and bitterness, might be influencing quality scores more significantly.

Temperature: The temperature distribution shows considerable variation between clusters. Higher quality clusters tend to have a more consistent temperature range, whereas lower quality clusters show more variability. Some clusters (e.g., 9, 15, 16) showed a higher median temperature, possibly indicating that specific temperature ranges are more conducive to achieving higher quality.

Fermentation Time: The plot shows that fermentation time is relatively consistent across clusters, with median times mostly falling between 14 and 16 days. This consistency might indicate that fermentation time has a less direct impact on quality compared to other factors like alcohol content and bitterness.

pH Level: The pH level distributions indicate that higher quality clusters maintain a slightly higher median pH level, potentially contributing to the desired flavor profile.

Alcohol Content: Higher quality clusters generally have a slightly higher alcohol content, which might be a factor in the overall beer quality.

Bitterness: The bitterness distribution has considerable variation in some clusters. Higher quality clusters tend to have higher median bitterness levels, avoiding extremes. This suggests that that bitterness may be a key quality determinant.

Color: Color intensity varies across clusters, with darker colors corresponding to higher quality scores. Clusters with higher quality scores tend to have a narrower range of color distribution, indicating consistency in color is associated with quality.

Summary

The clustering results suggest that certain features, such as fermentation time, temperature, pH level, alcohol content, and color, have noticeable patterns when associated with beer quality. These findings highlight areas for further investigation to refine the brewing process and enhance the quality of the final product. By understanding these feature distributions, the brewery can make informed decisions to optimize their recipes and brewing techniques.

✓ Deep Dive; Deriving Chemistry Data

Arguments can be made to add in the derived data earlier or later. For this project, because it's a derived and not raw data, it will be included only for the ML models. This is because derived data could influence the unsupervised kmeans clustering algorithm by putting clusters together showing relationships that don't really exist.

Some ideas for using ChemPy in this brewing analysis:

pH and Mineral Content: Use ChemPy to model how different mineral compositions in water (calcium, magnesium) affect the pH during mashing and how this influences enzyme activity and starch conversion.

Fermentation Dynamics: Model fermentation reactions to understand how different yeast strains (temperature and fermentation time) convert sugars to alcohol and CO₂, impacting the final quality and flavor profile of the beer.

Ingredient Interactions: Analyze how different ratios of malts and hops interact chemically during the brewing process and their cumulative impact on beer quality.

Integrating ChemPy with Machine Learning for more advanced analysis: (see phase 3 project for a demo of this technique) Integrate ChemPy with machine learning tools like scikit-learn to predict quality scores based on chemical composition and brewing conditions

Feature Engineering: Use ChemPy to calculate chemical properties (e.g., reaction rates, equilibrium constants) and use these as features for machine learning models.

Looking at derived pH data, can be helpful as it can provide context and information about the brewing environment that impacts the behavior of reactants in the measured data.



In this experiment, buffer capacities will be explored. Different malt and hops types have different buffering capacities, affecting the pH differently.

A note about the calculation, by incorporating malt and hops types into the buffer system calculation, the buffering capacity can be adjusted based on the specific ingredients used. The buffer capacities are essential for the pH calculation. The dictionaries used below are simple and efficient for lookup operations.

This method improves understanding of the system by providing a more details of how the initial pH and buffering capacities impact the final beer quality.

```
# Start the timer
start_time = time.time()

# Display the schema of the DataFrame
beer_encoded.printSchema()

# Stop the timer
end_time = time.time()

# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f"Data transformation execution time: {elapsed_time:.2f} seconds")
```

```
root
|-- pH_Level: double (nullable = true)
|-- Ingredient_Ratio: string (nullable = true)
|-- Fermentation_Time: integer (nullable = true)
|-- Temperature: double (nullable = true)
|-- Alcohol_Content: double (nullable = true)
|-- Bitterness: integer (nullable = true)
|-- Color: integer (nullable = true)
|-- Quality_Score: double (nullable = true)
|-- Beer_Style: string (nullable = true)
|-- Water_Ratio: double (nullable = false)
|-- Malt_Ratio: float (nullable = true)
|-- Hops_Ratio: float (nullable = true)
|-- Malt_Hops_Ratio: double (nullable = true)
|-- Malt_Type: string (nullable = false)
|-- Hops_Type: string (nullable = false)
|-- Beer_Style_Index: double (nullable = false)
|-- Beer_Style_Vec: vector (nullable = true)
|-- Malt_Type_Index: double (nullable = false)
|-- Malt_Type_Vec: vector (nullable = true)
|-- Hops_Type_Index: double (nullable = false)
|-- Hops_Type_Vec: vector (nullable = true)
```

Data transformation execution time: 0.00 seconds



Resources were sufficient to process the entire sample dataset using K-Means clustering. However, due to the resource-intensive nature of the chemistry analysis and machine learning modeling, the data was further subsampled for the subsequent analyses.

```
# Start the timer
start_time = time.time()

# Sample 19% of the rows with a seed for reproducibility (about 50,000 data points)
beer_encoded_sampled = beer_encoded.sample(withReplacement=False, fraction= 0.10, seed=42) #fraction=0.19
#save_checkpoint(beer_encoded_sampled, 'beer_encoded_sampled_checkpoint.pkl')

# Stop the timer
end_time = time.time()

# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f"Data transformation execution time: {elapsed_time:.2f} seconds")

↗ Data transformation execution time: 0.00 seconds

# Start the timer
start_time = time.time()

beer_encoded_sampled.show()

# Stop the timer
end_time = time.time()

# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f"Data transformation execution time: {elapsed_time:.2f} seconds")
```

↗

	pH_Level	Ingredient_Ratio	Fermentation_Time	Temperature	Alcohol_Content	Bitterness	Color	Quality_Score
	4.507213419450749	1:0.45:0.26	13	21.605469689190052	4.83702482975061	38	9	6.29563296010
	5.473563790227288	1:0.31:0.21	11	22.903972245938505	5.12916808476057	44	10	9.16786679335
	5.318024913782187	1:0.23:0.16	15	17.785713321899884	5.406625556398892	23	5	9.18166896093
	4.849521382421813	1:0.42:0.27	16	24.869178082298525	4.992812931715872	40	10	8.7216597606
	4.77639650569921	1:0.26:0.11	12	22.629383129373906	5.934652916760902	38	19	9.54481170565
	4.936910740031992	1:0.42:0.28	19	22.16253755912976	4.8155146142703495	21	13	6.47322438536
	4.898951275325849	1:0.25:0.18	12	21.085313067453882	5.743752260999525	50	5	8.7746891972
	4.686506092247504	1:0.33:0.29	14	24.627174195398503	5.853071743056866	38	19	6.88144480368
	4.830392434776611	1:0.32:0.30	10	19.98238616057116	5.763349903047741	24	8	9.62475291698
	4.981041360364337	1:0.48:0.12	15	15.950799547138214	4.806670861427709	34	8	9.10255600120
	5.176958540120013	1:0.34:0.19	19	21.606980566890734	5.699407080924889	42	8	7.41496913689
	5.059213040094015	1:0.24:0.22	14	24.297601254743455	5.294507072125857	44	10	6.12919023425
	5.36169705504533	1:0.38:0.27	15	15.122768597181741	5.231454844918106	24	9	7.270182014
	5.3010390412375274	1:0.40:0.14	10	24.63550929550697	4.660059729773532	43	13	9.25518069628
	4.94604033920897	1:0.29:0.22	18	18.793309849086146	5.345725971674105	38	11	7.3384618272
	4.632978412535182	1:0.30:0.11	14	21.973402759778423	5.887068567209649	59	12	8.34071473610
	4.520703452815498	1:0.22:0.17	14	21.27942480729402	4.6876777750637215	30	15	8.78875235823
	5.085276609607533	1:0.49:0.13	13	16.73400174339125	4.70563185194283	44	17	6.95840314949
	4.676128847897197	1:0.44:0.21	16	23.448759656048246	4.926426773725896	44	18	8.40158925659
	5.330018991092887	1:0.20:0.22	18	22.147875760404673	4.531304819165852	51	16	8.46728185008

only showing top 20 rows

Data transformation execution time: 0.25 seconds

```

# Start the timer
start_time = time.time()

# Define buffer capacities
malt_buffer_capacities = {
    'Barley': 0.1,
    'Wheat': 0.2,
    'Rye': 0.15,
    'Oats': 0.05
}

hops_buffer_capacities = {
    'Bittering': 0.2,
    'Dual Purpose': 0.1,
    'Aroma': 0.05
}

@jit(nopython=True)
def buffer_system_calculation(final_pH, malt_buffer, hops_buffer, malt_ratio, hops_ratio):
    """
    Simulates the buffering system in a brewing process based on the final pH,
    malt buffer, hops buffer, malt ratio, and hops ratio.
    """
    # Simplified buffer system calculation
    H_initial = 10 ** -final_pH
    return -np.log10(H_initial * (malt_buffer * malt_ratio + hops_buffer * hops_ratio))

def buffer_system_with_types_spark(final_pH, malt_type, hops_type, malt_ratio, hops_ratio):
    """
    Calculates the initial pH value of a brewing process based on the final pH, malt type, hops type, and their respective ratios.

    This function simulates the buffering system in brewing by:
    - Retrieving the buffer capacities for the given malt and hops types.
    - Using a predefined buffer system calculation function to estimate the initial pH based on these capacities and ratios.
    - Handling exceptions to ensure robustness during the Spark UDF execution.

    Parameters:
    -----
    final_pH : float
        The final pH level of the brew.
    malt_type : str
        The type of malt used in the brew (e.g., 'Barley', 'Wheat').
    hops_type : str
        The type of hops used in the brew (e.g., 'Bittering', 'Aroma').
    malt_ratio : float
        The ratio of malt in the brewing process.
    hops_ratio : float
        The ratio of hops in the brewing process.

    Returns:
    -----
    float
        The estimated initial pH value based on the buffer capacities and ratios.
    Returns None if an error occurs during the calculation.
    """
    try:
        malt_buffer = malt_buffer_capacities.get(malt_type, 0.1)
        hops_buffer = hops_buffer_capacities.get(hops_type, 0.1)
        initial_pH = buffer_system_calculation(final_pH, malt_buffer, hops_buffer, malt_ratio, hops_ratio)
        return float(initial_pH)
    except Exception as e:
        print(f"Error in buffer_system_with_types_spark: {e}")
        return None

# Define the UDF using the buffer system function
buffer_system_udf = udf(buffer_system_with_types_spark, DoubleType())

# Apply UDF to DataFrame with inferred malt and hops types
beer_encoded_sampled = beer_encoded_sampled.withColumn(
    'Derived_Initial_pH',
    buffer_system_udf(
        col('pH_Level'),
        col('Malt_Type'),
        col('Hops_Type'),
        col('Malt_Ratio'),
        col('Hops_Ratio')
    )
)

```


```
)

# Print the first 5 rows of the Derived_Initial_pH column
beer_encoded_sampled.select('Derived_Initial_pH').show(5)

# Stop the timer
end_time = time.time()

# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f"Data transformation execution time: {elapsed_time:.2f} seconds")
```



```
+-----+
|Derived_Initial_pH|
+-----+
| 5.655955083856954|
| 6.456530454413157|
| 6.726960306091945|
| 5.804198409698535|
| 5.977055969804945|
+-----+
only showing top 5 rows
```

Data transformation execution time: 1.80 seconds



This calculation might not fully capture the complexity of the actual buffering system in the brewing process. It is a placeholder or starting point, and the calculation can be adjusted to better reflect the real chemical processes.

This could involve:

- 1 Adding more detailed buffer equations.
- 2 Considering additional components or reactions in the brewing process.
- 3 Incorporating temperature and pressure effects.
- 4 Using more accurate or empirical data for buffer capacities and ratios.

Here's an example:

`@jit(nopython=True) def buffer_system_calculation(final_pH, malt_buffer, hops_buffer, malt_ratio, hops_ratio, additional_buffer, temp_factor):` """ Simulates the buffering system in a brewing process based on the final pH, malt buffer, hops buffer, malt ratio, hops ratio, and other factors. """

```
# Adjusted buffer system calculation
H_initial = 10 ** -final_pH
combined_buffer = (malt_buffer * malt_ratio + hops_buffer * hops_ratio + additional_buffer)
adjusted_buffer = combined_buffer * temp_factor # Adjusting for temperature
return -np.log10(H_initial * adjusted_buffer)
```

```

# Start the timer
start_time = time.time()

# Persist the DataFrame to avoid recomputation
beer_encoded_sampled.cache()

# Select only the necessary columns for conversion to Pandas
selected_columns = beer_encoded_sampled.select('Derived_Initial_pH', 'Quality_Score')

# Convert to Pandas DataFrame for visualization (only after cleaning and reducing the size)
beer_encoded_pH_pd = selected_columns.toPandas()

# Create a custom colormap based on custom color palette
custom_cmap = mcolors.ListedColormap(custom_colors)

# Create the hexbin plot
plt.figure(figsize=(10, 6))
hb = plt.hexbin(beer_encoded_pH_pd['Quality_Score'], beer_encoded_pH_pd['Derived_Initial_pH'], gridsize=50, reduce_C_function=np.mean)
plt.colorbar(hb, label='Mean Derived Initial pH')
plt.title('Derived Initial pH vs Quality Score')
plt.xlabel('Quality Score')
plt.ylabel('Derived Initial pH')

# Save the plot as an image with better layout
filename = "derived_initial_pH_hexbin.jpg"
plt.savefig(filename, bbox_inches='tight')

# Download the file
files.download("derived_initial_pH_hexbin.jpg")

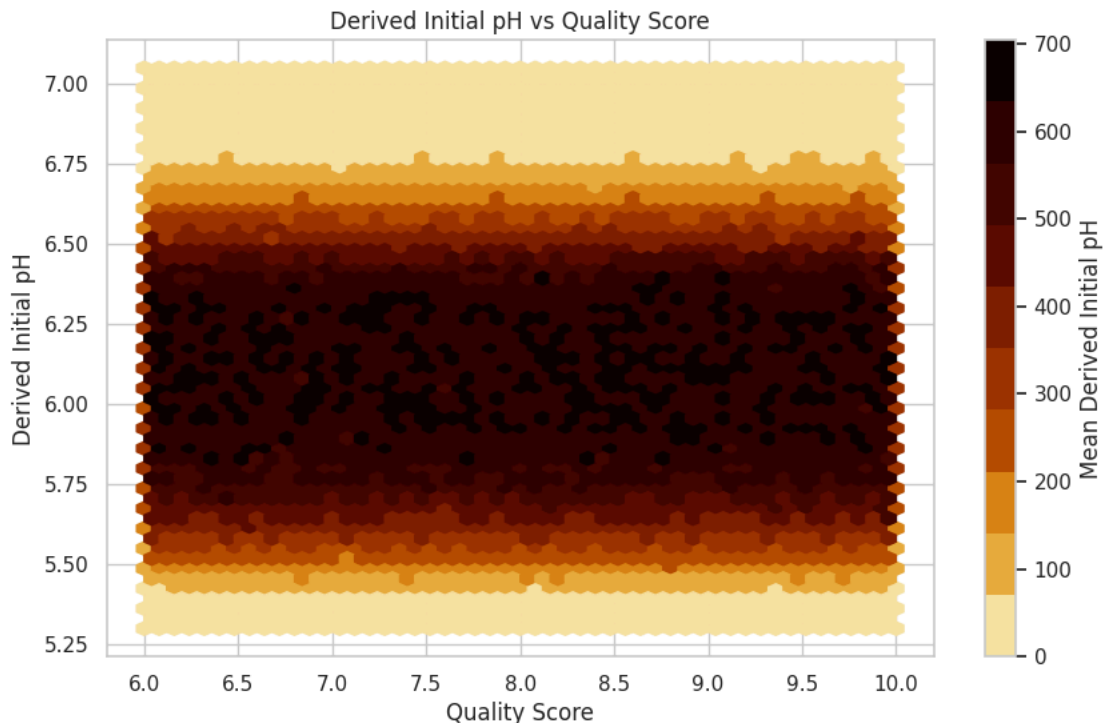
# Show the plot
plt.show()

# Stop the timer
end_time = time.time()

# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f>Data transformation execution time: {elapsed_time:.2f} seconds")

```



Data transformation execution time: 20.87 seconds