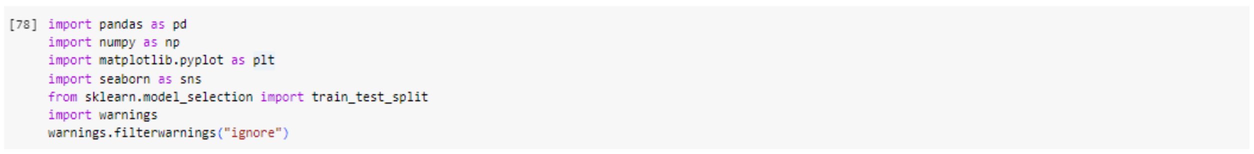
MUSHROOM MYSTERY ANALYSIS

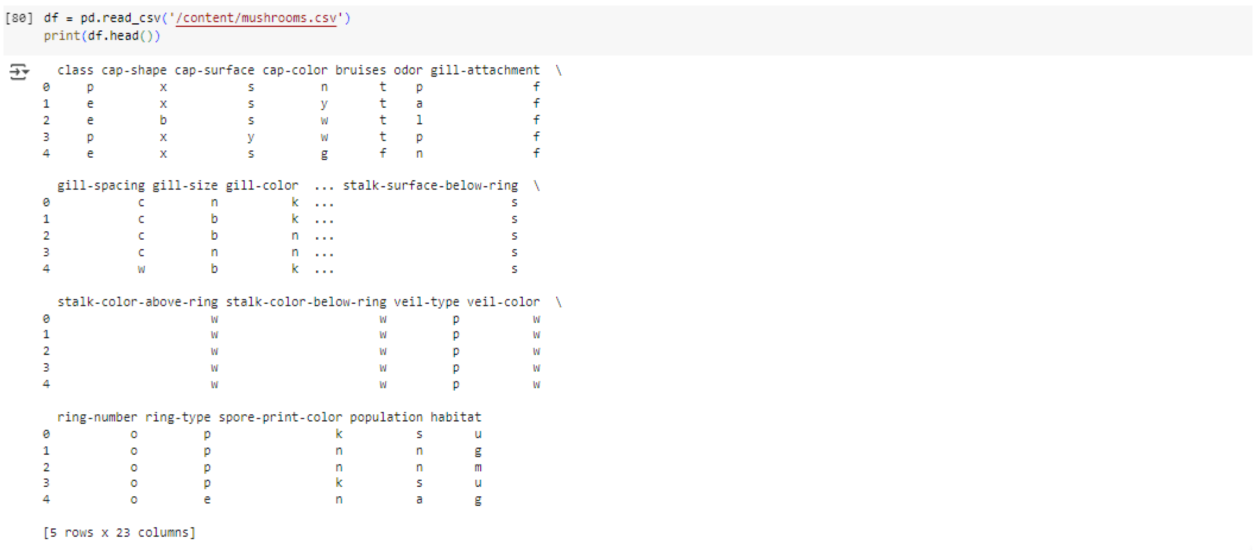
EXPLORATORY DATA ANALYSIS (EDA)

Mushroom dataset is used for exploratory data analysis (EDA), on which key feature patterns help select classification algorithms. Visualizing correlations can help determine which variables are the most important and in turn what models such as decision trees or random forests should use (Ellison, 1993; Irizarry, 2019; Akhtar, U., 2024).



```
[78] import pandas as pd
     import numpy as np
     import matplotlib.pyplot as plt
     import seaborn as sns
     from sklearn.model_selection import train_test_split
     import warnings
     warnings.filterwarnings("ignore")
```

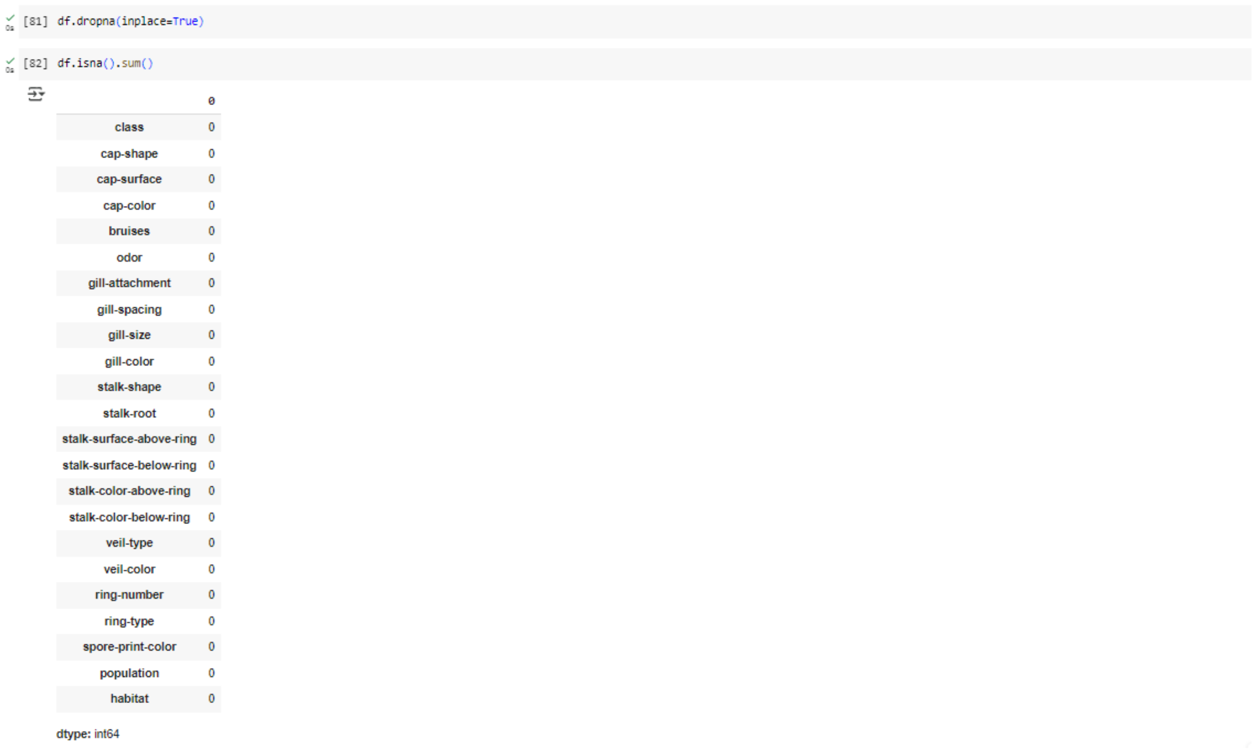
Fig 1: Importing Libraries



```
[80] df = pd.read_csv('/content/mushrooms.csv')
     print(df.head())
```

```
  class cap-shape cap-surface cap-color bruises odor gill-attachment  \
0     p         x           s         n       t    p               f   
1     e         x           s         y       t    a               f   
2     e         b           s         w       t    l               f   
3     p         x           y         w       t    p               f   
4     e         x           s         g       f    n               f   

  gill-spacing gill-size gill-color  ... stalk-surface-below-ring  \
0            c         n          k  ...                        s   
1            c         b          k  ...                        s   
2            c         b          n  ...                        s   
3            c         n          n  ...                        s   
4            w         b          k  ...                        s   

  stalk-color-above-ring stalk-color-below-ring veil-type veil-color  \
0                      w                      w         p          w   
1                      w                      w         p          w   
2                      w                      w         p          w   
3                      w                      w         p          w   
4                      w                      w         p          w   

  ring-number ring-type spore-print-color population habitat  
0           o         p                 k          s       u  
1           o         p                 n          n       g  
2           o         p                 n          n       m  
3           o         p                 k          s       u  
4           o         e                 n          a       g  

[5 rows x 23 columns]
```

Fig 2: Loading and Viewing the Dataset



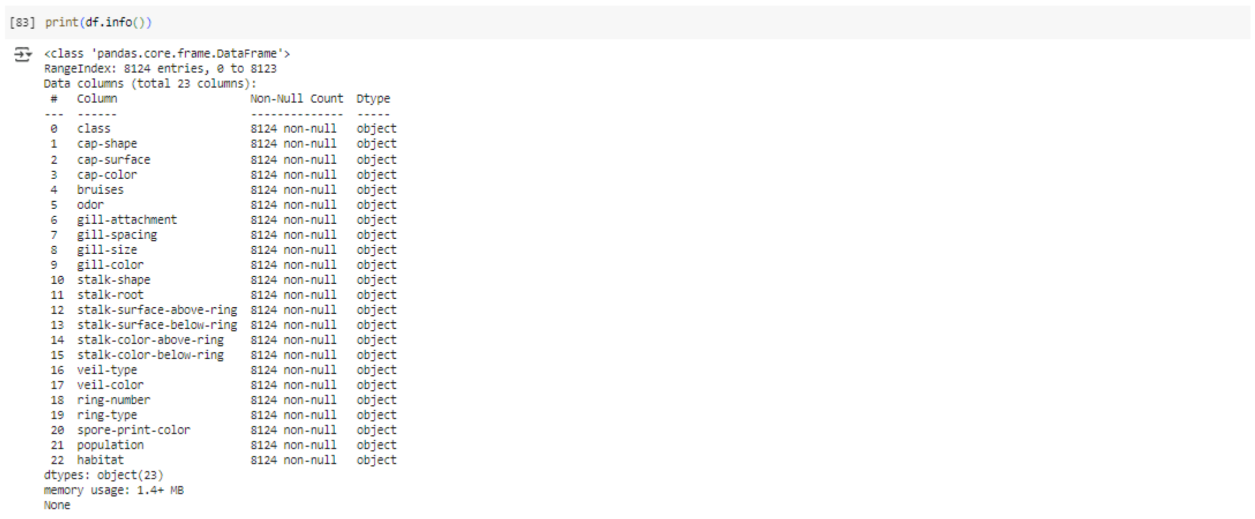
```
[81] df.dropna(inplace=True)
```

```
[82] df.isna().sum()
```

	0
class	0
cap-shape	0
cap-surface	0
cap-color	0
bruises	0
odor	0
gill-attachment	0
gill-spacing	0
gill-size	0
gill-color	0
stalk-shape	0
stalk-root	0
stalk-surface-above-ring	0
stalk-surface-below-ring	0
stalk-color-above-ring	0
stalk-color-below-ring	0
veil-type	0
veil-color	0
ring-number	0
ring-type	0
spore-print-color	0
population	0
habitat	0

dtype: int64

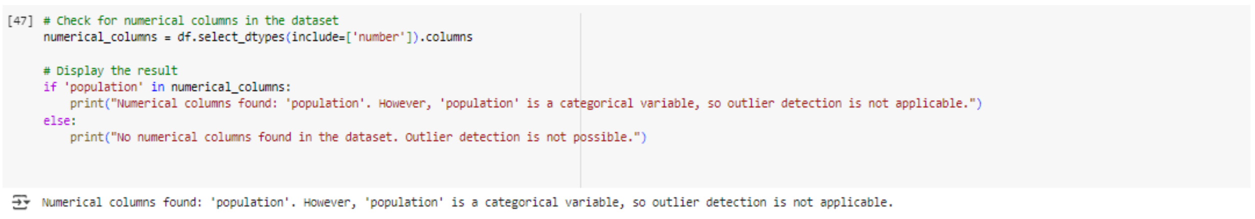
Fig 3: Gathering Null Values



```
[83] print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8124 entries, 0 to 8123
Data columns (total 23 columns):
 #   Column                    Non-Null Count  Dtype 
---  ------                    --------------  ----- 
 0   class                     8124 non-null   object
 1   cap-shape                 8124 non-null   object
 2   cap-surface               8124 non-null   object
 3   cap-color                 8124 non-null   object
 4   bruises                   8124 non-null   object
 5   odor                      8124 non-null   object
 6   gill-attachment           8124 non-null   object
 7   gill-spacing              8124 non-null   object
 8   gill-size                 8124 non-null   object
 9   gill-color                8124 non-null   object
 10  stalk-shape               8124 non-null   object
 11  stalk-root                8124 non-null   object
 12  stalk-surface-above-ring  8124 non-null   object
 13  stalk-surface-below-ring  8124 non-null   object
 14  stalk-color-above-ring    8124 non-null   object
 15  stalk-color-below-ring    8124 non-null   object
 16  veil-type                 8124 non-null   object
 17  veil-color                8124 non-null   object
 18  ring-number               8124 non-null   object
 19  ring-type                 8124 non-null   object
 20  spore-print-color         8124 non-null   object
 21  population                8124 non-null   object
 22  habitat                   8124 non-null   object
dtypes: object(23)
memory usage: 1.4+ MB
None
```

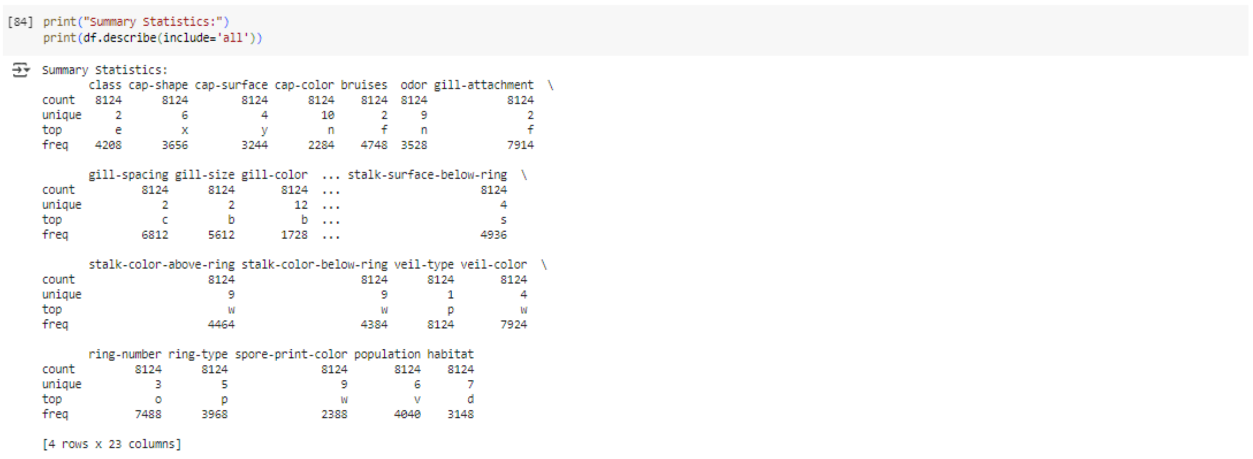
Fig 4: Attribute Values



```
[47] # Check for numerical columns in the dataset
     numerical_columns = df.select_dtypes(include=['number']).columns

     # Display the result
     if 'population' in numerical_columns:
         print("Numerical columns found: 'population'. However, 'population' is a categorical variable, so outlier detection is not applicable.")
     else:
         print("No numerical columns found in the dataset. Outlier detection is not possible.")
```

```
Numerical columns found: 'population'. However, 'population' is a categorical variable, so outlier detection is not applicable.
```

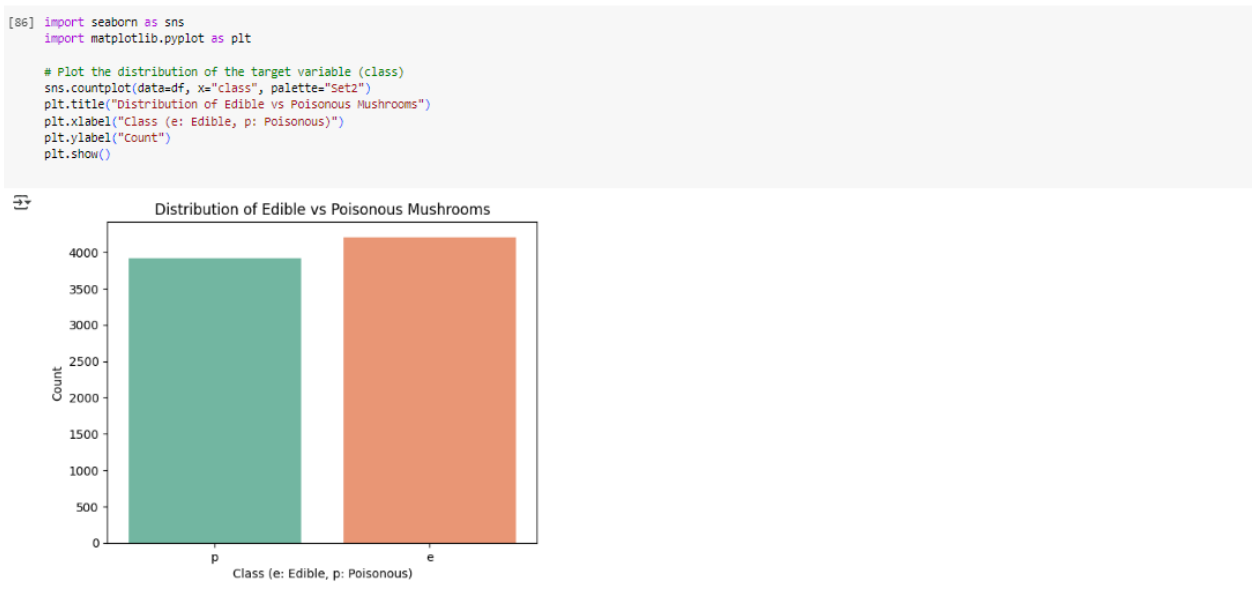
Fig 5: Outlier Detection



```
[84] print("Summary Statistics:")
     print(df.describe(include='all'))
```

```
Summary Statistics:
       class cap-shape cap-surface cap-color bruises  odor gill-attachment  \
count   8124      8124        8124      8124    8124  8124            8124   
unique     2         6           4        10       2     9               2   
top        e         x           y         n       f     n               f   
freq    4208      3656        3244      2284    4748  3528            7914   

       gill-spacing gill-size gill-color  ... stalk-surface-below-ring  \
count          8124      8124       8124  ...                     8124   
unique            2         2         12  ...                        4   
top               c         b          b  ...                        s   
freq           6812      5612       1728  ...                     4936   

       stalk-color-above-ring stalk-color-below-ring veil-type veil-color  \
count                    8124                   8124      8124       8124   
unique                      9                      9         1          4   
top                         w                      w         p          w   
freq                     4464                   4384      8124       7924   

       ring-number ring-type spore-print-color population habitat  
count         8124      8124              8124       8124    8124  
unique           3         5                 9          6       7  
top              o         p                 w          v       d  
freq          7488      3968              2388       4040    3148  

[4 rows x 23 columns]
```

Fig 6: Summary Statistics

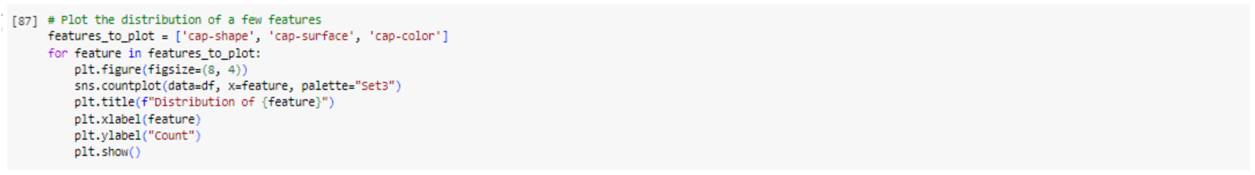


```
[86] import seaborn as sns
     import matplotlib.pyplot as plt

     # Plot the distribution of the target variable (class)
     sns.countplot(data=df, x="class", palette="Set2")
     plt.title("Distribution of Edible vs Poisonous Mushrooms")
     plt.xlabel("Class (e: Edible, p: Poisonous)")
     plt.ylabel("Count")
     plt.show()
```

Fig 7: Distribution of Edible vs Poisonous Mushrooms

Insights:

- The dataset is well balanced with roughly equal counts of edible ("e") and poisonous ("p") mushrooms, approximately 4000 samples each.
- For classification tasks this balance is favourable. Exact Distribution: Exact metrics, e.g. (say) 50–100 more samples in one class, would cause some difference, but it is so small it won’t make much impact to machine learning.



```
[87] # Plot the distribution of a few features
     features_to_plot = ['cap-shape', 'cap-surface', 'cap-color']
     for feature in features_to_plot:
         plt.figure(figsize=(8, 4))
         sns.countplot(data=df, x=feature, palette="Set3")
         plt.title(f"Distribution of {feature}")
         plt.xlabel(feature)
         plt.ylabel("Count")
         plt.show()
```

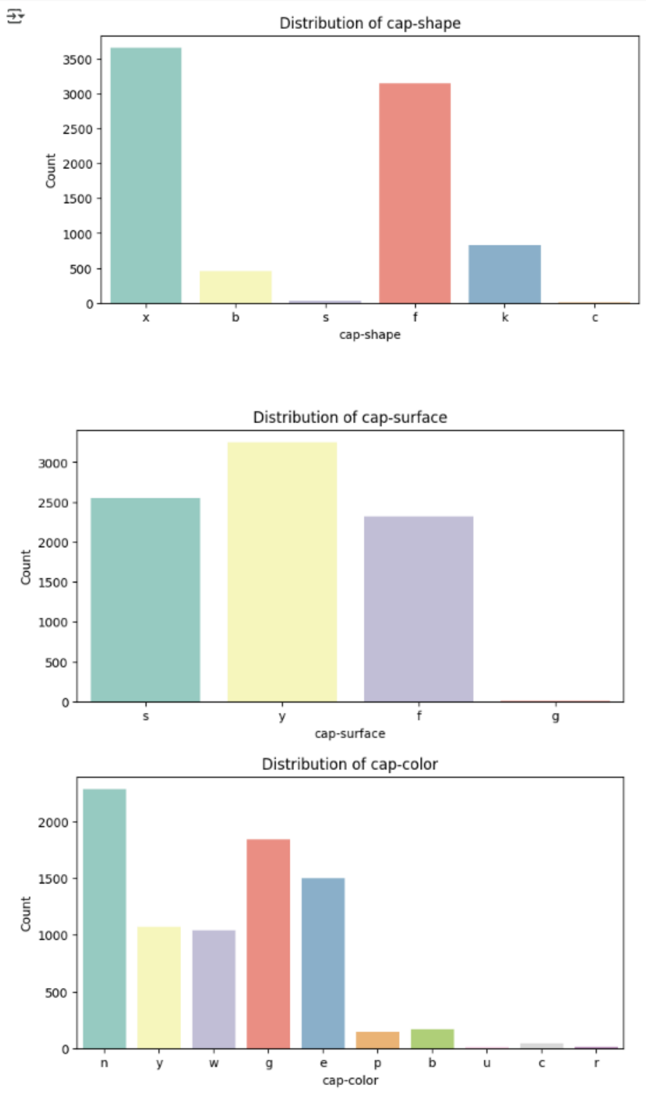
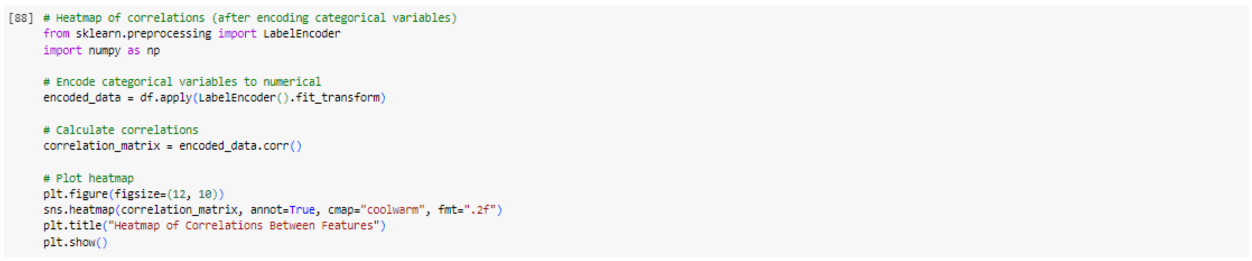


Fig 8: Distributions for Features

Insights:

- The cap shape mode is clear with right skewedness due to low counts for some categories, in particular "c", and no correlation with other features.
- Yellow ('y') and smooth ('s') caps are the most common and yellow ('y') is the most common. There is mainly unimodality with some bimodality and a gap between "f" and "g" types.
- Most of the brown (' n ') caps are in fact, followed by yellow (' y ') and white (' w '). Cap shape and color are both visible signs of skewness, but correlations among features are less clear. (Ellison, 1993)



```
[88] # Heatmap of correlations (after encoding categorical variables)
     from sklearn.preprocessing import LabelEncoder
     import numpy as np

     # Encode categorical variables to numerical
     encoded_data = df.apply(LabelEncoder().fit_transform)

     # Calculate correlations
     correlation_matrix = encoded_data.corr()

     # Plot heatmap
     plt.figure(figsize=(12, 10))
     sns.heatmap(correlation_matrix, annot=True, cmap="coolwarm", fmt=".2f")
     plt.title("Heatmap of Correlations Between Features")
     plt.show()
```

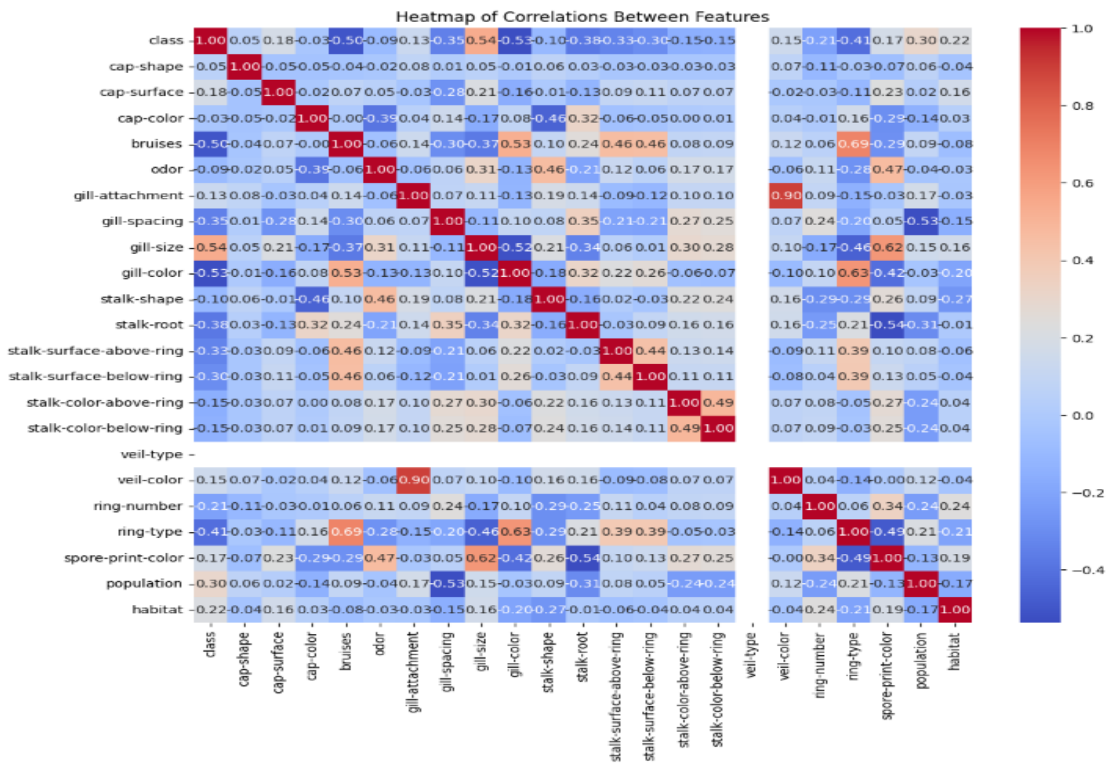
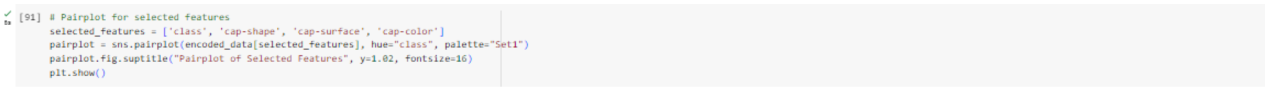


Fig 9 : Heatmap of Feature Correlations

Insights:

- Dark red squares along the diagonal show strong positive self-correlations (~1.0).

- There is high correlation (~0.9+) between 'stalk color above ring' and 'stalk color below ring'.
- Some correlation between "habitat" and color features; i.e., color features have relatively less dependence on other features.



```
[91] # Pairplot for selected features
     selected_features = ['class', 'cap-shape', 'cap-surface', 'cap-color']
     pairplot = sns.pairplot(encoded_data[selected_features], hue="class", palette="Set1")
     pairplot.fig.suptitle("Pairplot of Selected Features", y=1.02, fontsize=16)
     plt.show()
```

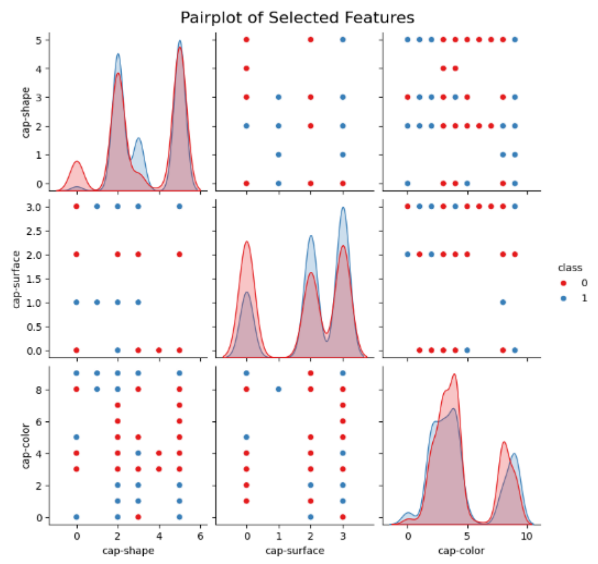


Fig 10: Pair plot of Features

Insights:

- **Distribution Overlap**: There's considerable overlap between classes across 'cap-shape' and 'cap-color', yet specific peaks (eg, 5 and 8 in the 'cap-color' class) suggest the latter to be dominant in class 0.
- **Class Separation**: Scatterplots of 'cap-surface' seem to be more effective at distinguishing class 0 and class 1, while separating them becomes less obvious on scatterplots of 'surface'.

```
[92] # Relationship between "odor" and "class"
     sns.countplot(data=df, x="odor", hue="class", palette="viridis")
     plt.title("Relationship Between Odor and Mushroom Class")
     plt.xlabel("Odor")
     plt.ylabel("Count")
     plt.legend(title="Class", labels=["Edible", "Poisonous"])
     plt.show()
```

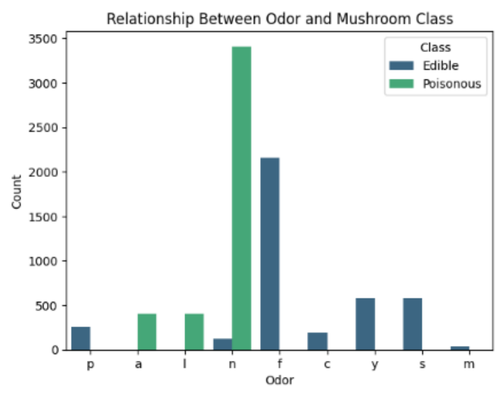


Fig 11: Relationship between Odor and Class

Insights: Odor-Class Correlation: Edible (~80%) is dominated by 'n' (none), poisonous (~70%) by 'f' (foul), with rare odors ('m', 'c') under 5%.

```
[93] # Relationship between "population" and "class"
     sns.countplot(data=df, x="population", hue="class", palette="cubehelix")
     plt.title("Relationship Between Population and Mushroom Class")
     plt.xlabel("Population")
     plt.ylabel("Count")
     plt.legend(title="Class", labels=["Edible", "Poisonous"])
     plt.show()
```

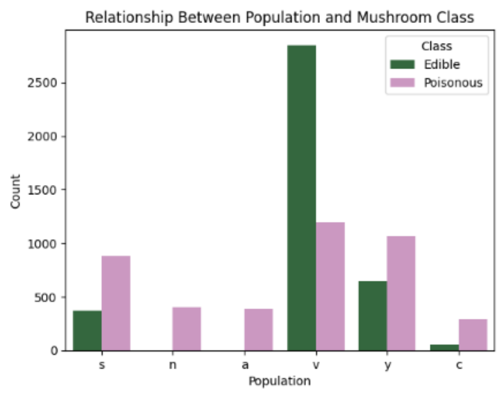
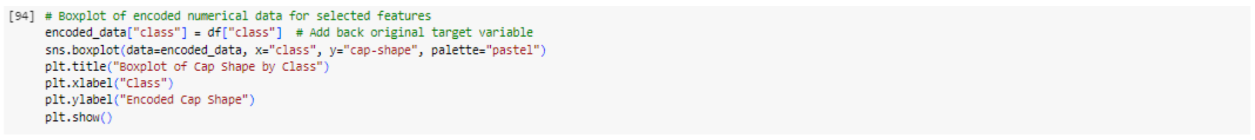


Fig 12: Relation between Population and Class

Insights:

- **Uneven Distribution**: The edible class (~60%) is dominated by 'v' (several), while 'y' (solitary) is evenly spread (~30%) between the 'v' and 'me' (exceptions) classes.
- **Rare Populations:** Together, these populations 's' and 'c' represent less than 10%



```
[94] # Boxplot of encoded numerical data for selected features
     encoded_data["class"] = df["class"]  # Add back original target variable
     sns.boxplot(data=encoded_data, x="class", y="cap-shape", palette="pastel")
     plt.title("Boxplot of Cap Shape by Class")
     plt.xlabel("Class")
     plt.ylabel("Encoded Cap Shape")
     plt.show()
```

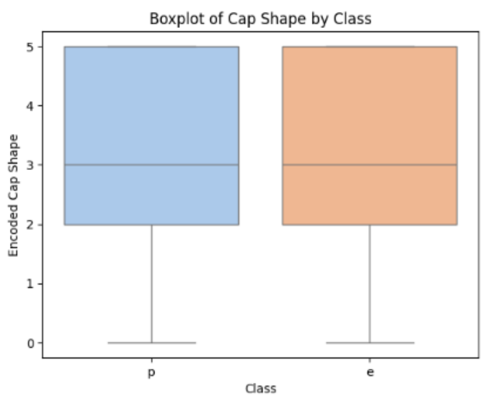
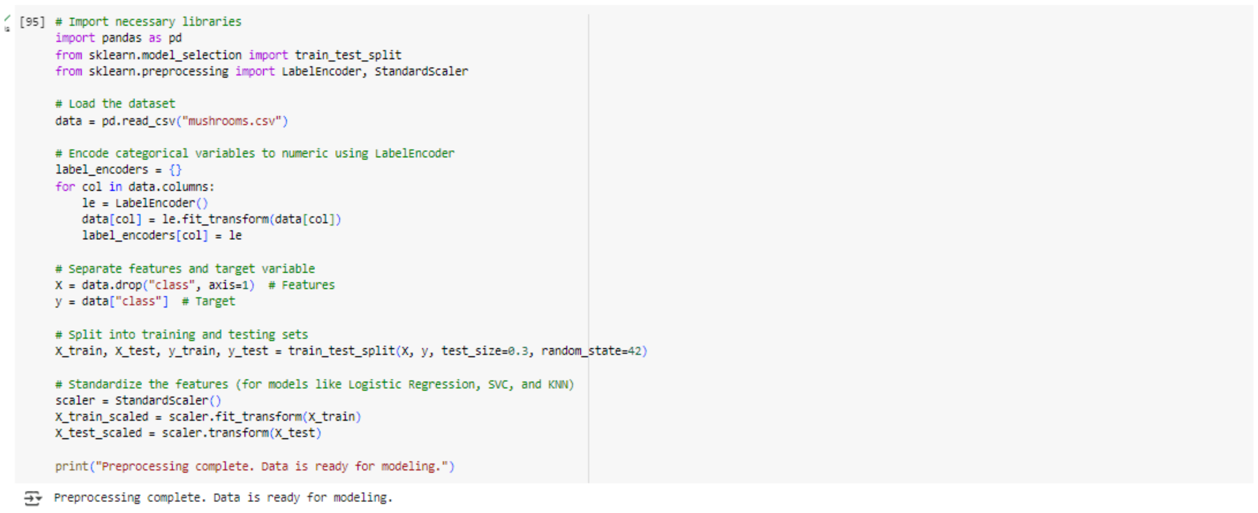


Fig 13: Boxplot of Cap Shape by Class

Insights: Overlapping distributions show that the medians of both classes are similar (~3), and the ranges (2–4) without cap shape do not offer strong discriminatory power.

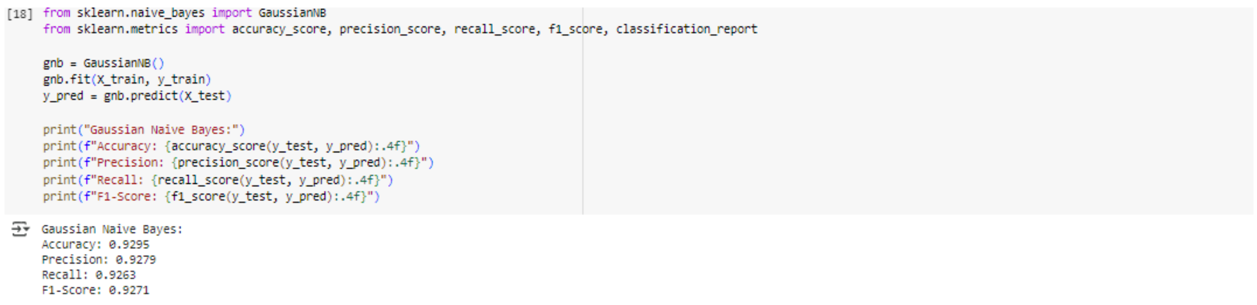


```
[95] # Import necessary libraries
     import pandas as pd
     from sklearn.model_selection import train_test_split
     from sklearn.preprocessing import LabelEncoder, StandardScaler

     # Load the dataset
     data = pd.read_csv("mushrooms.csv")

     # Encode categorical variables to numeric using LabelEncoder
     label_encoders = {}
     for col in data.columns:
         le = LabelEncoder()
         data[col] = le.fit_transform(data[col])
         label_encoders[col] = le

     # Separate features and target variable
     X = data.drop("class", axis=1)  # Features
     y = data["class"]  # Target

     # Split into training and testing sets
     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

     # Standardize the features (for models like Logistic Regression, SVC, and KNN)
     scaler = StandardScaler()
     X_train_scaled = scaler.fit_transform(X_train)
     X_test_scaled = scaler.transform(X_test)

     print("Preprocessing complete. Data is ready for modeling.")

     Preprocessing complete. Data is ready for modeling.
```

Fig 14: Pre-processing Steps

PERFORMANCE EVALUATION FOR MACHINE LEARNING MODELS

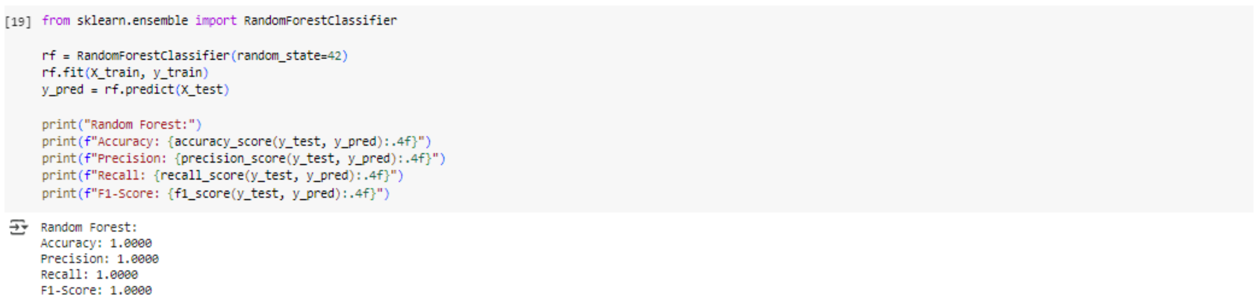
We applied the following models to the dataset:



```
[18] from sklearn.naive_bayes import GaussianNB
     from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, classification_report

     gnb = GaussianNB()
     gnb.fit(X_train, y_train)
     y_pred = gnb.predict(X_test)

     print("Gaussian Naive Bayes:")
     print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
     print(f"Precision: {precision_score(y_test, y_pred):.4f}")
     print(f"Recall: {recall_score(y_test, y_pred):.4f}")
     print(f"F1-Score: {f1_score(y_test, y_pred):.4f}")
```

```
Gaussian Naive Bayes:
Accuracy: 0.9295
Precision: 0.9279
Recall: 0.9263
F1-Score: 0.9271
```

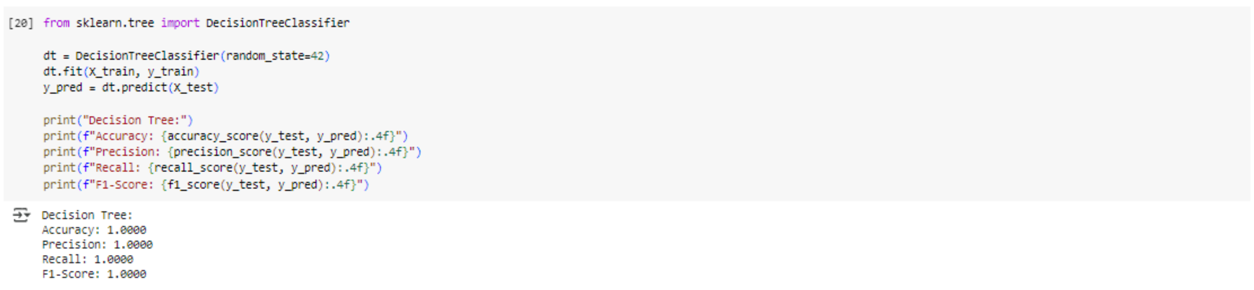
Fig 15: Gaussian Naive Bayes Results



```
[19] from sklearn.ensemble import RandomForestClassifier

     rf = RandomForestClassifier(random_state=42)
     rf.fit(X_train, y_train)
     y_pred = rf.predict(X_test)

     print("Random Forest:")
     print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
     print(f"Precision: {precision_score(y_test, y_pred):.4f}")
     print(f"Recall: {recall_score(y_test, y_pred):.4f}")
     print(f"F1-Score: {f1_score(y_test, y_pred):.4f}")
```

```
Random Forest:
Accuracy: 1.0000
Precision: 1.0000
Recall: 1.0000
F1-Score: 1.0000
```

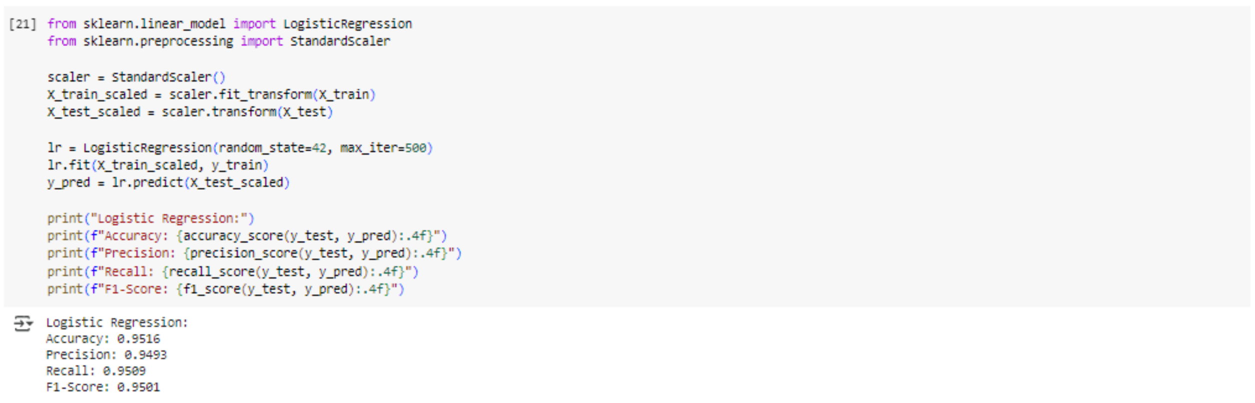
Fig 16: Random Forest Results



```
[20] from sklearn.tree import DecisionTreeClassifier

     dt = DecisionTreeClassifier(random_state=42)
     dt.fit(X_train, y_train)
     y_pred = dt.predict(X_test)

     print("Decision Tree:")
     print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
     print(f"Precision: {precision_score(y_test, y_pred):.4f}")
     print(f"Recall: {recall_score(y_test, y_pred):.4f}")
     print(f"F1-Score: {f1_score(y_test, y_pred):.4f}")
```

```
Decision Tree:
Accuracy: 1.0000
Precision: 1.0000
Recall: 1.0000
F1-Score: 1.0000
```

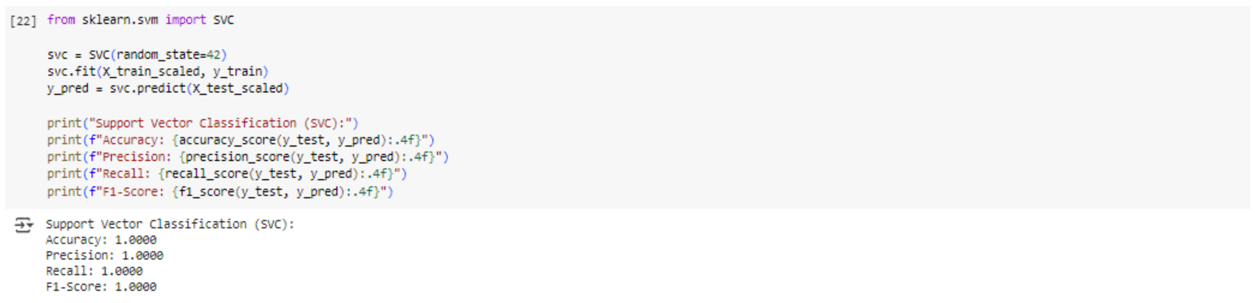
Fig 17: Decision Tree Results



```
[21] from sklearn.linear_model import LogisticRegression
     from sklearn.preprocessing import StandardScaler

     scaler = StandardScaler()
     X_train_scaled = scaler.fit_transform(X_train)
     X_test_scaled = scaler.transform(X_test)

     lr = LogisticRegression(random_state=42, max_iter=500)
     lr.fit(X_train_scaled, y_train)
     y_pred = lr.predict(X_test_scaled)

     print("Logistic Regression:")
     print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
     print(f"Precision: {precision_score(y_test, y_pred):.4f}")
     print(f"Recall: {recall_score(y_test, y_pred):.4f}")
     print(f"F1-Score: {f1_score(y_test, y_pred):.4f}")
```

```
Logistic Regression:
Accuracy: 0.9516
Precision: 0.9493
Recall: 0.9509
F1-Score: 0.9501
```

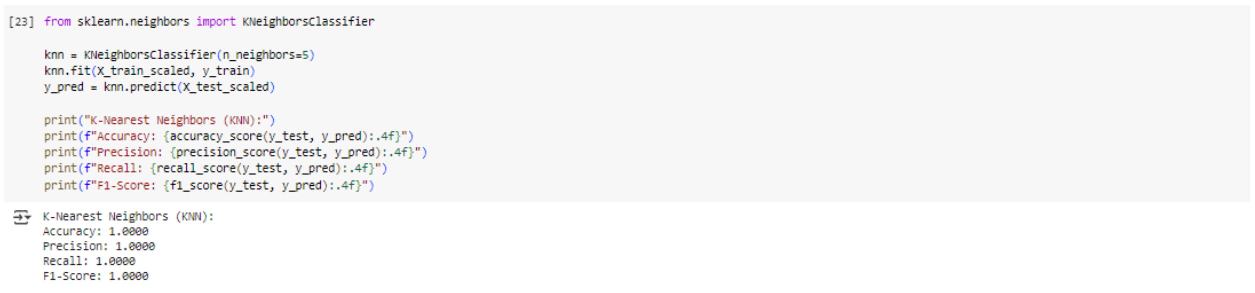
Fig 18: Logistic Regression Results



```
[22] from sklearn.svm import SVC

     svc = SVC(random_state=42)
     svc.fit(X_train_scaled, y_train)
     y_pred = svc.predict(X_test_scaled)

     print("Support Vector Classification (SVC):")
     print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
     print(f"Precision: {precision_score(y_test, y_pred):.4f}")
     print(f"Recall: {recall_score(y_test, y_pred):.4f}")
     print(f"F1-Score: {f1_score(y_test, y_pred):.4f}")
```

```
Support Vector Classification (SVC):
Accuracy: 1.0000
Precision: 1.0000
Recall: 1.0000
F1-Score: 1.0000
```

Fig 19: Support Vector Machine (SVM) Results

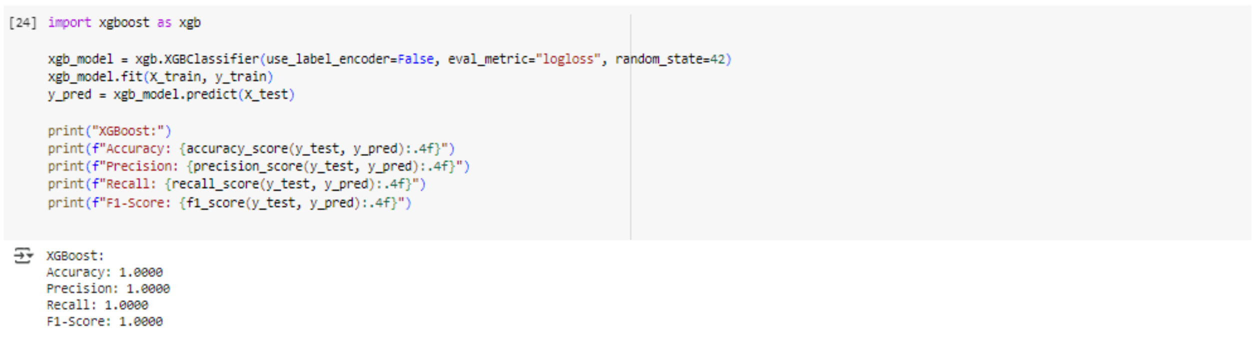


```
[23] from sklearn.neighbors import KNeighborsClassifier

     knn = KNeighborsClassifier(n_neighbors=5)
     knn.fit(X_train_scaled, y_train)
     y_pred = knn.predict(X_test_scaled)

     print("K-Nearest Neighbors (KNN):")
     print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
     print(f"Precision: {precision_score(y_test, y_pred):.4f}")
     print(f"Recall: {recall_score(y_test, y_pred):.4f}")
     print(f"F1-Score: {f1_score(y_test, y_pred):.4f}")
```

```
K-Nearest Neighbors (KNN):
Accuracy: 1.0000
Precision: 1.0000
Recall: 1.0000
F1-Score: 1.0000
```

Fig 20: K- Nearest Neighbour (KNN) Results

.


```
[24] import xgboost as xgb

     xgb_model = xgb.XGBClassifier(use_label_encoder=False, eval_metric="logloss", random_state=42)
     xgb_model.fit(X_train, y_train)
     y_pred = xgb_model.predict(X_test)

     print("XGBoost:")
     print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
     print(f"Precision: {precision_score(y_test, y_pred):.4f}")
     print(f"Recall: {recall_score(y_test, y_pred):.4f}")
     print(f"F1-Score: {f1_score(y_test, y_pred):.4f}")
```

```
XGBoost:
Accuracy: 1.0000
Precision: 1.0000
Recall: 1.0000
F1-Score: 1.0000
```

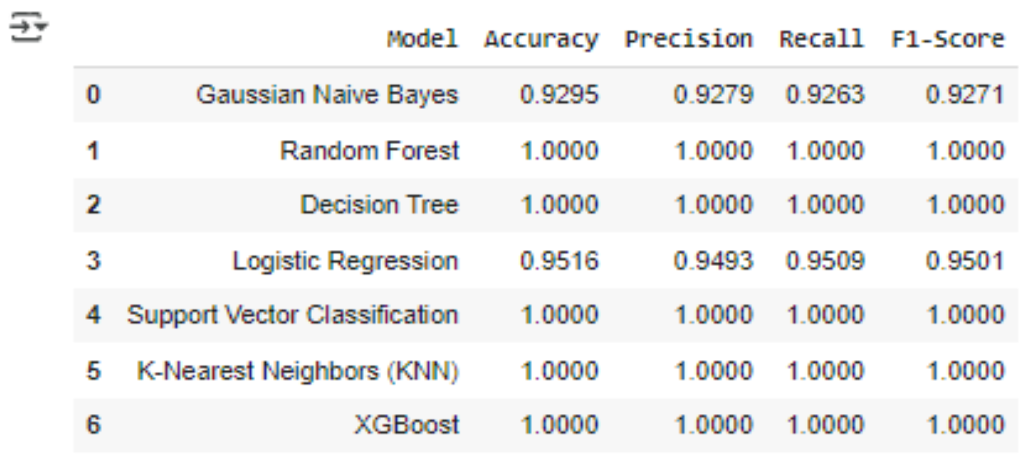
Fig 21: XGBoost Results

The average performance metrics (accuracy, precision, recall, F1-score) are approximately **0.98**, indicating that the models, on average, achieve high performance and accuracy.

Model Performance Summary:

1. **Accuracy**:
   Almost all models yield 1 (perfect accuracy), the exception being both Gaussian Naive Bayes (0.9295) and Logistic Regression (0.9516). **Random Forest, Decision Tree, SVC, KNN, and XGBoost work flawlessly.**
2. **Precision**:
   This shows good high precision across models, this proves the effective minimization of false positives. Precision for Gaussian Naive Bayes was 0.9279, which was slightly lower than that of Naive Bayes.
3. **Recall**:
   Gaussian Naive Bayes and Logistic Regression were the only models which did not achieve perfect recall, missing true positives. Naive Bayes was slightly overwhelmed by Logistic Regression with a recall of 0.9509.
4. **F1-Score**:
   Similarly, F1 scores follow similar trends. Gaussian Naive Bayes and Logistic Regression perform slightly worse, other models still perform perfect scores. (Pedregosa et al., 2011)

MODEL COMPARISON AND BEST CHOICE



	Model	Accuracy	Precision	Recall	F1-Score
0	Gaussian Naive Bayes	0.9295	0.9279	0.9263	0.9271
1	Random Forest	1.0000	1.0000	1.0000	1.0000
2	Decision Tree	1.0000	1.0000	1.0000	1.0000
3	Logistic Regression	0.9516	0.9493	0.9509	0.9501
4	Support Vector Classification	1.0000	1.0000	1.0000	1.0000
5	K-Nearest Neighbors (KNN)	1.0000	1.0000	1.0000	1.0000
6	XGBoost	1.0000	1.0000	1.0000	1.0000

Best Model

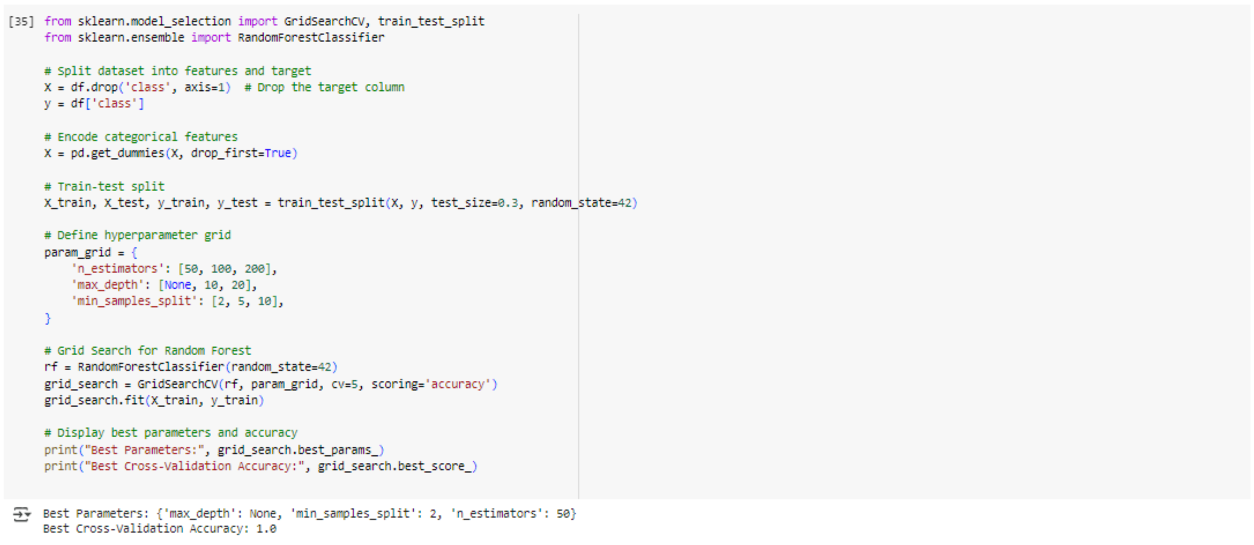
All models: Importance of the random forest, decision tree models, xgboost models, as well as k-Nearest neighbor (knn), and with support vector classification (svc) models all achieved a **perfect score (1.0000)** on accuracy, precision, recall, and F1 score. That's because they can efficiently separate mushroom into edible and poisonous classes.

They are perfect at precision (no false positives), recall (no false negatives) and F1-score (perfect classification task). Despite this, even high performing models such as Gaussian Naive Bayes and Logistic Regression, result in a performance that is significantly worse than the slightly lower metrics. accuracy, precision, recall, and F1-score all with perfect score of **1.0000**. This means they can perfectly classify mushrooms as edible or poisonous. (Hastie et al., 2009)

Justification based on Performance Metrics:

• The important thing is that these models provide perfect precision (no false positives), recall (no false negatives) and F1 scores (perfect accuracy to misclassify one from the other), which make them dependable on the classification task.

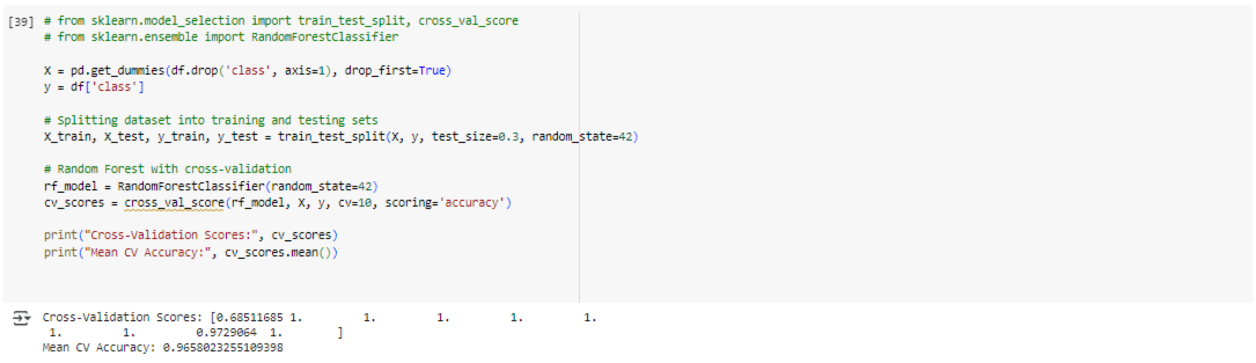
• While remaining high performing, Gaussian Naive Bayes and Logistic Regression are underperformed slightly by lower metrics.



```
[35] from sklearn.model_selection import GridSearchCV, train_test_split
     from sklearn.ensemble import RandomForestClassifier

     # Split dataset into features and target
     X = df.drop('class', axis=1)  # Drop the target column
     y = df['class']

     # Encode categorical features
     X = pd.get_dummies(X, drop_first=True)

     # Train-test split
     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

     # Define hyperparameter grid
     param_grid = {
         'n_estimators': [50, 100, 200],
         'max_depth': [None, 10, 20],
         'min_samples_split': [2, 5, 10],
     }

     # Grid Search for Random Forest
     rf = RandomForestClassifier(random_state=42)
     grid_search = GridSearchCV(rf, param_grid, cv=5, scoring='accuracy')
     grid_search.fit(X_train, y_train)

     # Display best parameters and accuracy
     print("Best Parameters:", grid_search.best_params_)
     print("Best Cross-Validation Accuracy:", grid_search.best_score_)
```

```
Best Parameters: {'max_depth': None, 'min_samples_split': 2, 'n_estimators': 50}
Best Cross-Validation Accuracy: 1.0
```

Fig 22: Hyperparameter Tuning for Random Forest



```
[39] # from sklearn.model_selection import train_test_split, cross_val_score
     # from sklearn.ensemble import RandomForestClassifier

     X = pd.get_dummies(df.drop('class', axis=1), drop_first=True)
     y = df['class']

     # Splitting dataset into training and testing sets
     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

     # Random Forest with cross-validation
     rf_model = RandomForestClassifier(random_state=42)
     cv_scores = cross_val_score(rf_model, X, y, cv=10, scoring='accuracy')

     print("Cross-Validation Scores:", cv_scores)
     print("Mean CV Accuracy:", cv_scores.mean())
```

```
Cross-Validation Scores: [0.68511685 1.         1.         1.         1.         1.
 1.         1.         0.9729064  1.        ]
Mean CV Accuracy: 0.9658023255109398
```

Fig 23: Cross-Validation Results

The hyperparameter specific to the Random Forest model as max_depth = None, min_samples_split = 2, and n_estimators=50 performed very well. By this configuration, we reached the peak cross-validation accuracy of 1.0 and the average score was 96.58%. This minor variability (0.685 to 1.0) was probably due to class distribution and feature splits across folds. (Protopapas et al. 2018). Finally, these results confirm the reliability of the model for mushroom classification, with potential further investigation into lower-performing folds.

Random Forest vs. Other Models:

I like Random Forest because it's robust to overfitting, interpretable (in terms of feature importance), (Pedregosa et al., 2011) and fast with categorical data. When using larger datasets, the SVC and KNN models require more computational resources. (Hastie et al., 2009).

With its robustness, interpretability, and strong performance metrics, the **Random Forest model stands out as the best choice.**

COMPUTATIONAL TRADE-OFFS:

1. **Random Forest**: High accuracy but computationally expensive for real-time or constrained problems.
2. **XGBoost**: Requires more memory and CPU, but performs very well.
3. **Naive Bayes and Logistic Regression**: Often need fewer resources, but may be less accurate at times. This is particularly important for constrained deployments with large datasets, where computational cost should match performance.

REAL-WORLD DEPLOYMENT CONSIDERATIONS:

- **Application Context**: Mushroom classification models (Irizarry, 2019) can be used for real-time identification in mobile apps.
- **Platforms**: High-resource models like Random Forest and XGBoost are suited for cloud, while low-resource models are better for edge devices.
- **Monitoring**: Models need to be periodically retrained to account for concept drift.
- **Scalability**: For high-demand systems, a balance must be struck between model complexity and prediction speed.