LONDON METROPOLITAN UNIVERSITY

islington college
(इस्लिङ्टन कलेज)

Module Code & Module Title

CU6051NI Artificial Intelligence

75% Individual Coursework

Submission: Milestone 1

Academic Semester: Autumn Semester 2025

Credit: 15 credit semester long module

Student Name: Digdarshan Bhattarai

London Met ID: 23049051

College ID: NP01CP4A230320

Assignment Due Date: 07/01/2026.

Assignment Submission Date: 07/01/2026

Submitted To: Er. Roshan Shrestha

GitHub Link	*https://github.com/DigdarshanB/mushroom-edibility-classification*

I confirm that I understand my coursework needs to be submitted online via MST Classroom under the relevant module page before the deadline for my assignment to be accepted and marked. I am fully aware that late submissions will be treated as non-submission and a mark of zero will be awarded.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

Mushrooms are widely consumed across the globe, loved for being a delicious, healthy, food source. However, determining which ones to eat and which ones to avoid is a big hassle. Many varieties of poisonous species appear nearly the same shape, color, and size as the ones that can be consumed, and thus cannot be detected by merely visual inspection. Even a small mistake during identification can result into severe illness or even death.

Figure 1: Visual similarity between ***(a)*** **Chlorophyllum molybdites** *(poisonous) and* ***(b)*** **Macrolepiota procera** *(edible), highlighting the difficulty of identification based on appearance alone. (ResearchGate, 2025)*

The conventional methods of identifying mushrooms have relied on human experience, reference books, or visual examination of physical characteristics. Such approaches are generally subjective, slow and inaccurate. Fortunately, there is an increase in the availability of structured biological data, which opens the prospects of applying computational methods in enhancing accuracy and consistency in the process of classifying mushrooms.

As the amount of structured biological data increases, there is an excellent chance to use AI methods to assist in classifying mushrooms. AI systems are ideal because they can compare data in large amounts in a timely and objective manner and are therefore suitable in safety-sensitive decision-making tasks, such as determining the edibility of mushrooms.

1.1 Aim and Objectives of the Project

The overall goal of the project is to use and assess the suitability of supervised machine learning algorithms to determine whether mushrooms are edible or poisonous, based on an already existing dataset and the objectives are:

- To prepare and investigate the mushroom dataset to learn its features and adaptability to supervised classification.
- To use chosen supervised machine learning algorithms to the mushroom data to make the edibility classification.
- To compare the performance of the algorithms on the standard classification metrics to figure out which approach is most appropriate in this issue.

1.2 Overview of the Dataset and Problem Formulation

In this project we are addressing the problem of mushroom edibility and solving it as a binary classification problem - each mushroom is assigned with either edible or poisonous tag

The data is taken out of the UCI Machine Learning Repository a famous go-to resource in AI research benchmark data, which currently has and maintains 688 datasets, serving students of machine learning and AI. (UCI, 2025)

The dataset consists of 8,124 rows which consists various kinds of mushrooms and 22 columns, having attributes such as:

- Cap shape
- Cap colour
- Odour
- Gill size
- Gill colour
- Habitat

Each mushroom has been classified as either as edible or poisonous, hence the data samples are excellent at supervised learning. The attributes are diverse and rich which enables machine-learning models to identify subtle patterns.

1.3 AI Concepts Used

a) Supervised Machine Learning

Supervised ML, simply put means training a model on known dataset to predict outcomes for unknown data. First step is to train a model, then it can be used to predict new inputs (Verma, 2025).

Supervised learning is especially suitable to this problem as the correct classifications are already present in historical data. The model learns the relationship of feature and outcome and predicts correctly on new samples of the mushrooms, which are not visible.

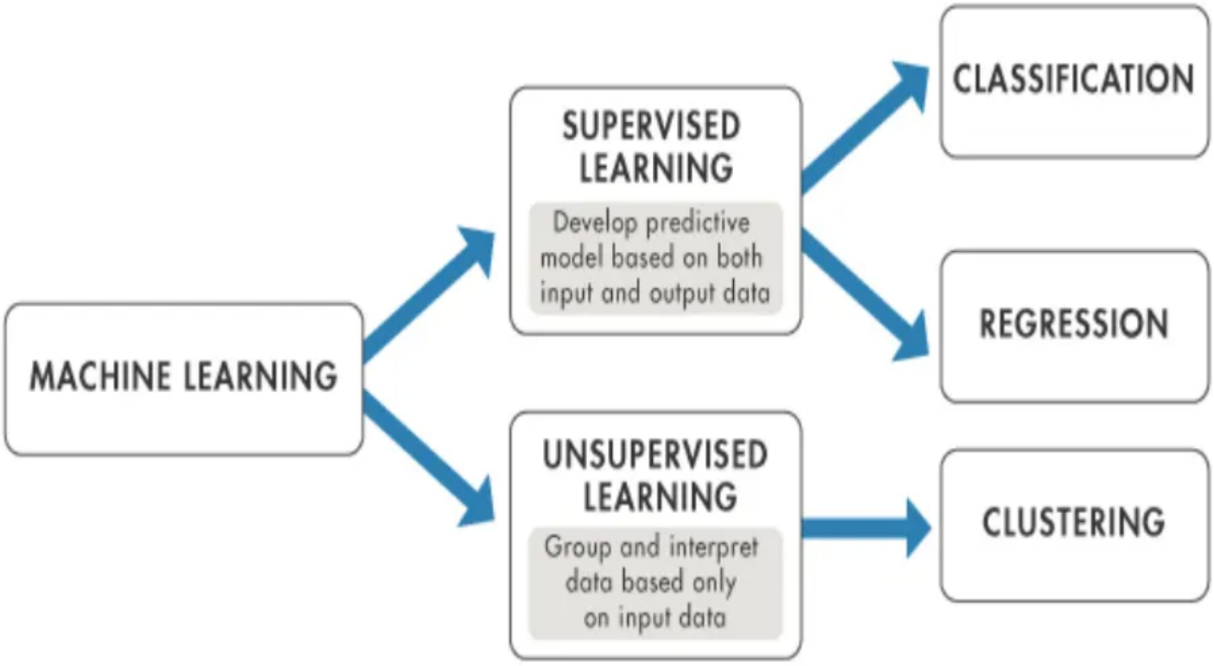

Figure 2: Types of Machine Learning (Verma, 2025)

1.4 Classification Algorithms

In this experiment, we have used and compared various supervised classification algorithms to determine which one is best used to classify a mushroom as being edible. The algorithms are:

a) Logistic Regression

Logistic Regression is used to predict binary outcomes.

b) Naïve Bayes

Naive Bayes predicts probability distribution over a set of classes.

c) Random Forest

Random Forest combines a collection of decision trees and is used to boost accuracy and reduce overfitting.

2. Background

2.1 Problem Context: Why Mushroom Identification is Hard

Correct identification of mushroom is crucial task because some deadly species resemble edible ones in shape, colour, and overall appearance closely just like the example above. People used to rely on expert knowledge, books and manuals which is both time consuming and difficult to apply consistently, especially for beginners.

The difficulty of this issue is especially the fact that this property is not defined by one obvious characteristic that makes the product edible. The own description of the dataset (and its UCI description) clearly points out that no single simple rule can be considered reliable to determine the mushroom edibility: rather, the choice relies on the combination of various traits. UCI machine learning repository.

This is precisely what kind of environment supervised learning might come in: a model can gain the ability to learn many relations between many attributes at once and generalize them.

2.2 Dataset Background and Origin

The dataset used in this coursework is the popular UCI mushroom dataset (Agaricus and Lepiota). It includes the description of hypothetical Herbs which represent 23 species of the gilled mushrooms of the family Agaricus and Lepiota which had been first present in The Audubon Society Field Guide to North American Mushrooms (1981).

Instances: 8,124

The predictor attributes include: 22 (all nominal/categorical).

Label: target: It is edible (e) or poisonous (p).

Distribution of class:

- edible = 4,208 (51.8%)
- poisonous = 3,916 (48.2%)

In the data set, it is also mentioned that species may be definitely edible, definitely poisonous, or unknown/not recommended, and the unknown/not recommended category is grouped in with the toxic category. It becomes significant as it influences us to perceive the label of poisonous in a certain way (this also involves cases of do not eat, but not necessarily being medically poisonous).

2.3 Research on Mushroom Classification

Mushroom Edibility prediction has been studied by researchers mainly as supervised classification problem, with the goal to predict edible vs unsafe mushroom from recorded traits. The UCI mushroom dataset is used for this purpose because it is stated that there is not any particular rule for finding out edibility, meaning many features should be considered together (UCI ML Repository, 1987) .

2.3.1 How mushrooms are identified usually

Traditionally, mushrooms have been characterised by observable characteristics including cap shape/colour, gill features, stalk features, spore print colour, habitat and smell. This method is effective with experts, though may be challenging with beginners since numerous edible and toxic mushrooms are similar. This is why scientists have considered the application of AI to help the process of the decision making based on learning patterns of the recorded mushroom characteristics.

2.3.2 Why AI is used for this problem

The edibility of mushrooms is an AI study that focuses on mushroom edibility as a supervised classification problem: in the training procedure, examples that specify the correct label (edible or poisonous) are provided. This is helpful since it is a combination of traits that classify edibility rather than an individual property. The description of the UCI data also indicates that it cannot be determined whether the item is edible using a single simple rule and because of this, multi-feature machine learning models can be used.

2.4 Review of Existing Work on Mushroom Edibility Prediction

a) Prior Use of Random Forest/Decision Trees on Mushroom Datasets

Tree-based methods are widely used because they represent step by step decisions using categorical features. To put it into perspective, if the odour of mushroom is x, then y, and so on.

In a study which compares Random Forest and REP Tree on the UCI mushroom dataset, the study reports high performance with Random Forest achieving 100% accuracy and REP Tree achieving slightly lover but still high results.

This suggests that for this UCI dataset, tree-based models often perform well because they capture interactions between traits. That said, the scores should still be interpreted carefully as real-world identification can involve uncertain or missing data too (Nepal Journals Online, 2022).

Nepal Journal of Mathematical Sciences (NJMS)
ISSN: 2738-9928 (online), 2738-9812 (print)
Vol. 3, No. 1, 2022 (February): 111-116
DOI: https://doi.org/10.3126/njmathsci.v3i1.44130

Research Article
Received Date: December 25, 2021
Accepted Date: February 24, 2022
Published Date: February 28, 2022

Mushroom Classification using Random Forest and REP Tree Classifiers

Nawaraj Paudel[1] and Jagdish Bhatta[2]

[1,2]*Central Department of Computer Science and IT, Tribhuvan University, Kathmandu, Nepal*
Email: [1]*nawarajpaudel@cdcsit.edu.np*, [2]*jagdish@cdcsit.edu.np*
Corresponding Author: Jagdish Bhatta

Abstract: *Mushroom is a reproductive structure produced by some fungi that has a high level of protein and a rich source of vitamin B. It aids in the prevention of cancer, weight loss, and immune system enhancement. There are numerous thousands of mushroom species within the world and a few are edible and a few are noxious due to noteworthy poisons on them. Hence, it is a vital errand to distinguish between edible and harmful mushrooms. This paper focuses on comparing the performance of two tree-based classification algorithms, Random Forest and Reduced Error Pruning (REP) Tree, for the classification of edible and poisonous mushrooms. In this paper, mushroom dataset from UCI machine learning repository has been classified using Random Forest and REP Tree classifiers. The evaluation of these two algorithms using accuracy, precision, recall and F-measure shows that the Random Forest outperforms REP Tree algorithm with value of 100% for accuracy, precision, recall and F- measure. The performance of Random Forest is 100% and is better with respect to REP Tree classifier.*

Keywords: *Mushroom Dataset, Random Forest, REP Tree, 10-fold cross-validation Confusion Matrix.*

Figure 3: Random Forest vs REP Tree (Poudel, Bhatta, 2022)

4. Experiments and Results

The two classification algorithms were executed on the mushroom dataset using 10-folds cross-validation for the classification of mushrooms based on their class labels. The table below shows confusion matrix of the classification report that has been obtained after testing Random Forest algorithm.

Table 1 Confusion Matrix of Random Forest Algorithm

		Predicted class		
		poisonous	edible	Total
Actual Class	Poisonous	3916	0	3916
	Edible	0	4208	4208
	Total	3916	4208	8124

The table below shows confusion matrix of the classification report that has been obtained after testing REPT Tree algorithm.

Table 2 Confusion Matrix of REP Tree Algorithm

		Predicted class		
		poisonous	edible	Total
Actual Class	Poisonous	3916	0	3916
	Edible	2	4206	4208
	Total	3918	4206	8124

Based on the classification reports shown in Table 1 and Table 2, the calculated summary performance result for the comparison of two algorithms applied on mushroom dataset is shown in the table below. The accuracy, precision, recall and F-measure value are shown is the average of precision, recall and F-measure for both categories.

Table 3 Performance result of two algorithms

Algorithm	Accuracy	Precision	Recall	F-measure
Random Forest	100%	100%	100%	100%
REP Tree	99.98%	99.95%	100%	99.97%

It is clearly seen that the accuracy, precision, recall, and F-measure values of Random Forest is 100% and that of REP Tree is 99.98%, 99.95%, 100%, and 99.97% respectively.

Figure 4: Result of the study (Paudel, Bhatta, 2022)

Advantages:

- Can interact with categorical features well.
- Can often attain extremely high benchmark accuracy on this dataset.
- Offer feature-importance style methods of information (in particular, with Random Forest).

Disadvantages:

- Random Forest is less interpretable than a single decision tree.
- Model behaviour may seem too good to be true on benchmark data and should be carefully examined.
- The behaviour of the models can vary.

b) Prior Use of Naïve Bayes/ Logistic Regression on Mushroom Datasets

Some works use Naïve Bayes and Logistic Regression, and these models are fast, easy to compare and provide a 'reference level' of performance which is why these models are very useful.

A study by Sulistianingsih and Martono (2025) that explains several supervised learning models that are used to classify mushrooms edibility using the UCI Mushroom dataset (8,124 instances and 22 attributes). Both Random Forest and one of their Stacking models are found to have 100% accuracy whereas Naive Bayes has significantly lower-performance (59.8% accuracy) due to the strong assumption of conditional independence by Naivete Bayes that the authors claim is very poor in capturing the interactional nature of mushroom traits.

In general, this work justifies the adoption of Logistic Regression and Naive Bayes as benchmarks and recommends the adoption of tree based / ensemble models to this field since they can represent the interactions between features. (ResearchGate, 2025)

J-INTECH (Journal of Information and Technology)
Accredited Sinta 4 Ministry of Higher Education, Science and Technology Republic of Indonesia SK No. 10/C/C3/DT.05.00/2025
E-ISSN: 2580-720X || P-ISSN: 2303-1425

J-INTECH Journal of Information and Technology

Analysis of the Effectiveness of Traditional and Ensemble Machine Learning Models for Mushroom Classification

Neny Sulistianingsih[1*], Galih Hendo Martono[2]

[1,2]*Departement of Computer Science, Master Program, Universitas Bumigora, Ismail Marzuki St. Mataram, Indonesia*

Keywords
Bagging; Ensemble Learning; K-Nearest Neighbors; Mushroom Classification; Random Forest; Stacking; Voting Classifier

***Corresponding Author:**
neny.sulistianingsih@universitasbumigora.ac.id

Abstract
The classification of edible versus poisonous mushrooms presents a critical challenge in the domains of applied biology and public health, particularly due to the serious implications of misidentification. This research employs the UCI Mushroom Dataset to evaluate and compare the effectiveness of several machine learning models, including traditional algorithms like Logistic Regression, Decision Tree, Random Forest, Support Vector Machine, K-Nearest Neighbors and Naïve Bayes, as well as advanced ensemble techniques such as Stacking and Voting Classifier. Notably, both Random Forest and Stacking achieved flawless accuracy, reaching 100%, underscoring the high predictive capacity of these models in complex categorical scenarios. Conversely, Naïve Bayes exhibited significantly weaker performance—achieving only 59.8% accuracy—likely due to its underlying assumption of feature independence, which does not hold for this dataset. The ensemble learning approaches, including the combination of Stacking and Bagging, not only preserved but also enhanced model robustness and generalization. These methods effectively leverage the complementary strengths of individual learners to yield more accurate and stable predictions while mitigating overfitting risks. Comparative analysis with previous research confirms the consistency of these findings and reinforces the viability of ensemble strategies for handling intricate classification tasks. Overall, this study highlights the importance of algorithm selection tailored to data characteristics and supports the use of ensemble learning to boost predictive reliability.

Figure 5: Traditional vs Ensemble models (Sulistianingsih & Martono, 2025)

Table 5. Classification Results with Traditional Methods

Model	Accuracy	Precision	Recall	F1 Score	AUC
Logistic Regression	0.841	0.842	0.841	0.842	0.913
Decision Tree	0.998	0.998	0.998	0.998	0.998
Support Vector Machine	0.995	0.995	0.995	0.995	0.999
K-Nearest Neighbors	1.000	1.000	1.000	1.000	1.000
Naive Bayes	0.598	0.787	0.598	0.550	0.835

Figure 6: Result of Baselines vs ensembles (Sulistianingsih & Martono, 2025)

Advantages:

- Logistic Regression is easy and easily explained as a baseline.
- Naive Bayes is fast and is usually used to form the basis of categorical classification.
- Both yield a clear baseline with which the ensembles can be evaluated as adding meaningful contributions.

Disadvantages:

- A Logistic Regression can fare poorly when the boundary of the decision is far from linear.
- Naive Bayes can fail badly when the independence properties are broken.
- It has been found that this is the case in this dataset.

c) Prior Use of Multiple Classifiers on Mushroom Datasets

Besides individual model comparisons, several studies also compare multiple supervised classifiers on the UCI mushroom dataset. The feature selection employed by Tank (2021) is the Principal Component Analysis (PCA) and various models such as Logistic Regression, Decision Tree, KNN, SVM, Naive Bayes, and Random Forest were tested. The paper provides a comparison of model performance in ROC-based performance measure, which justifies the notion that prediction of mushroom edibility is generally considered a benchmark classification task where many algorithms are assumed to be tested simultaneously and then one algorithm decided to be applied.

MINIMAL DECISION RULES BASED ON THE APRIORI ALGORITHM †

María C. FERNÁNDEZ*, Ernestina MENASALVAS*, Óscar MARBÁN*
José M. PEÑA**, Socorro MILLÁN***

Based on rough set theory many algorithms for rules extraction from data have been proposed. Decision rules can be obtained directly from a database. Some condition values may be unnecessary in a decision rule produced directly from the database. Such values can then be eliminated to create a more comprehensible (minimal) rule. Most of the algorithms that have been proposed to calculate minimal rules are based on rough set theory or machine learning. In our approach, in a post-processing stage, we apply the Apriori algorithm to reduce the decision rules obtained through rough sets. The set of dependencies thus obtained will help us discover irrelevant attribute values.

Keywords: rough sets, rough dependencies, association rules, Apriori algorithm, minimal decision rules

Figure 7:Use of Multiple Classifiers on Mushroom Datasets (Fernandez, 2001)

Advantages:

- A fair big picture comparison.
- Facilitates easier justification of results due to benchmarking of performance both against simple control baselines and stronger models
- Allows to understand what algorithms are most consistent across evaluation measures.

Disadvantages:

- Preprocessing options can bias results (e.g. the choice of encoding algorithm and the use of PCA)
- Comparison can be unreliable with models not tuned equally
- Interpretability may be compromised by dimensionality-reduction techniques such as PCA,

2.5 Key Takeaways from Existing Works

Much of the previous research indicates that models of mushroom edibility can acquire clear patterns of choices based on categorical characteristics. Various studies identify a consistent theme which is that tree-based methods especially the random forest are frequently able to perform to an extremely high benchmark score on the UCI mushroom data. An example by Paudel and Bhatta (2022), where they contrasted Random Forest with a pruned decision tree (REP Tree) had 100 percent accuracy with Random Forest with a REP Tree only slightly lower, indicating that tree-based logic is well-polarized with the interactions between the mushroom properties and the forecast of edibility.

Simultaneously, Logistic Regression and Naive Bayes are often used as a baseline model in studies. These baselines assist in interpreting the results, in that, they indicate the presence of meaningful improvement in the more sophisticated procedures. According to Sulistianingsih and Martono, (2025) nothing can be more accurate than Random Forest/stacking, and much less accurate than Naïve Bayes, which confirms the fact that mushroom characteristics tend to be interactive and that models based on simplistic assumptions are possibly not appropriate.

3. Solutions

3.1 Proposed Solution

The proposed solution for the project is a supervised ML pipeline that predicts whether a mushroom is edible or not based on recorded categorical attributes. The solution is designed to be implemented in Jupyter Notebook using Python as the programming language. The solution follows a structured workflow to evaluate every step fairly.

Three classification algorithms have been used – Logistic Regression, Naïve Bayes, and Random Forest for comparative analysis. These models present different levels of complexity. Logistic Regression provides a basic linear baseline for the project. Naïve Bayes is probability-based baseline which is best suited for categorical data and Random Forest is an ensemble model that can capture feature interactions. The final model recommendation will be based on evaluations of the train/test data.

3.2 System Workflow

1. **Load and Structure Data**: .data file is loaded into a table and feature names are applied from .names file.
2. **Data Preprocessing**: Categorial attributes are converted into machine-readable form and unknown values are handled.
3. **Dividing the dataset**: Dataset is divided into training and testing sets.
4. **Model Training**: Train Logistic Regression, Naïve Bayes, and Random Forest on the training dataset.
5. **Prediction**: Use trained models to predict class labels on the test set.
6. **Evaluation and Comparison**: Compare models using standard classification metrics and justify the most suitable model.

3.3 Pseudocode for the Overall Project

INPUT:

- Raw mushroom dataset (agaricus-lepiota.data)

- Column name list / metadata (agaricus-lepiota.names or predefined COLUMNS)

OUTPUT:

- Saved processed datasets (X_train, X_test, y_train, y_test, feature_columns)

- Trained models (LR, NB, RF)

- Evaluation results (metrics tables + plots)

START

1) IMPORT required libraries (pandas, numpy, sklearn metrics, matplotlib)

2) DATA PREPARATION

2.1 LOAD dataset (.data)

2.2 ASSIGN column names (COLUMNS)

2.3 CREATE clean working copy of dataset

2.4 REPLACE '?' with NA

2.5 FILL missing values with "missing" label

2.6 IDENTIFY non-informative columns

DROP constant columns

2.7 SPLIT into:

X = df_clean without target column "class"

y = df_clean["class"]

2.8 ONE-HOT ENCODE X → X_encoded = get_dummies(X)

2.9 TRAIN/TEST SPLIT with stratification:

2.10 VALIDATE:

- no missing values in X_train/X_test
- numeric types only

2.11 SAVE processed outputs to disk:

- X_train.csv, X_test.csv, y_train.csv, y_test.csv
- feature_columns.csv

3) MODEL TRAINING + EVALUATION (repeat for every model)

For each model in {Logistic Regression, Naive Bayes (BernoulliNB), Random Forest}:

3.1 LOAD processed datasets (X_train, X_test, y_train, y_test)

3.2 INITIALIZE model

3.3 FIT model on (X_train, y_train)

3.4 PREDICT labels: y_pred = model.predict(X_test)

3.5 PREDICT probabilities: y_score = model.predict_proba(X_test)[:, index_of_poisonous]

3.6 COMPUTE evaluation metrics:

- accuracy
- precision/recall/F1 for poisonous class
- confusion matrix
- ROC-AUC
- Average Precision (AP)

3.7 CREATE metrics summary table for comparison

3.8 PLOT:

- confusion matrix heatmap
- ROC curve
- precision-recall curve

4) COMPARE models using summary tables + safety-critical errors

END

3.4 Pseudocode for Each Algorithm

3.4.1 Pseudocode – Logistic Regression

INPUT: processed X_train, X_test, y_train, y_test

OUTPUT: trained LR model + predictions + metrics + plots

START

1) LOAD X_train, X_test, y_train, y_test from pre-processed CSV files

2) INITIALIZE LogisticRegression classifier

3) FIT model on (X_train, y_train)

4) PREDICT labels: y_pred_lr = predict(X_test)

5) PREDICT probabilities: y_score_lr = predict_proba(X_test) for poisonous class

6) EVALUATE:

- accuracy_score(y_test, y_pred_lr)
- classification_report(y_test, y_pred_lr)
- confusion_matrix(y_test, y_pred_lr)
- ROC curve + ROC-AUC using y_score_lr
- Precision-Recall curve + AP using y_score_lr

7) PLOT confusion heatmap, ROC curve, Precision-Recall curve

END

3.4.2 Pseudocode – Naïve Bayes

INPUT: processed X_train, X_test, y_train, y_test

OUTPUT: trained NB model + predictions + metrics + plots

START

1) LOAD X_train, X_test, y_train, y_test from pre-processed CSV files

2) INITIALIZE BernoulliNB classifier (binary features from one-hot encoding)

3) FIT model on (X_train, y_train)

4) PREDICT labels: y_pred_nb = predict(X_test)

5) PREDICT probabilities: y_score_nb = predict_proba(X_test) for poisonous class

6) EVALUATE:

- accuracy_score(y_test, y_pred_nb)
- classification_report(y_test, y_pred_nb)
- confusion_matrix(y_test, y_pred_nb)
- ROC curve + ROC-AUC using y_score_nb
- Precision-Recall curve + AP using y_score_nb

7) PLOT confusion heatmap, ROC curve, Precision-Recall curve

END

3.4.3 Pseudocode – Random Forest

INPUT: processed X_train, X_test, y_train, y_test

OUTPUT: trained RF model + predictions + metrics + plots

START

1) LOAD X_train, X_test, y_train, y_test from pre-processed CSV files

2) INITIALIZE RandomForestClassifier

3) FIT model on (X_train, y_train)

4) PREDICT labels: y_pred_rf = predict(X_test)

5) PREDICT probabilities: y_score_rf = predict_proba(X_test) for poisonous class

6) EVALUATE:

- accuracy_score(y_test, y_pred_rf)
- classification_report(y_test, y_pred_rf)
- confusion_matrix(y_test, y_pred_rf)
- ROC curve + ROC-AUC using y_score_rf
- Precision-Recall curve + AP using y_score_rf

7) PLOT confusion heatmap, ROC curve, Precision-Recall curve

END

3.5 Flowchart – Overall System

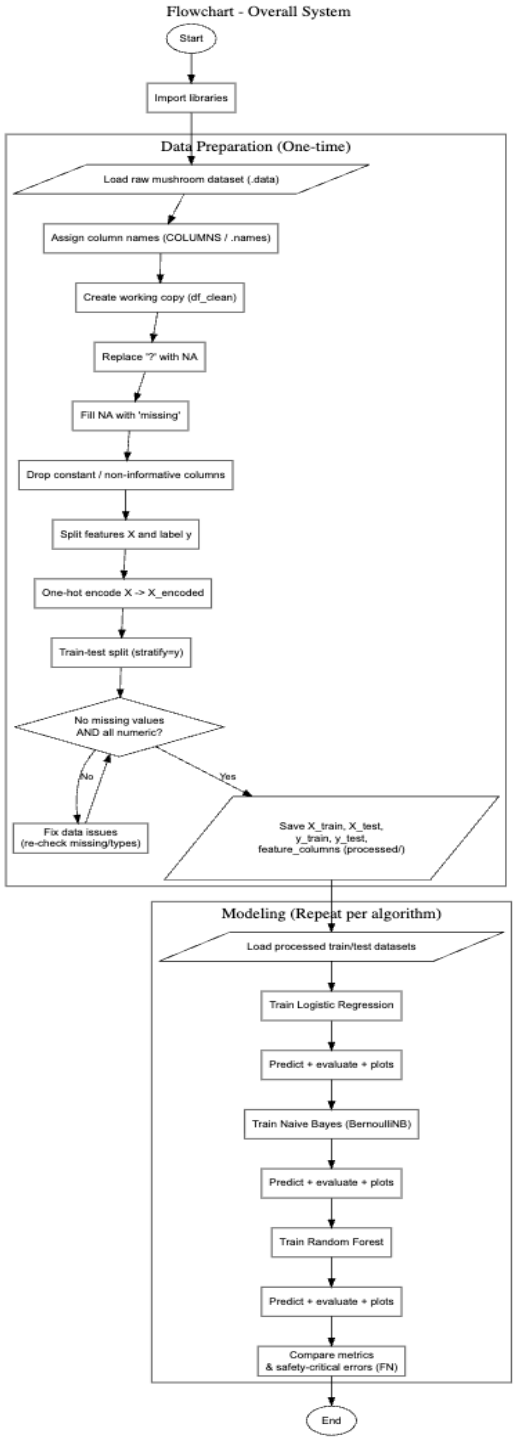

Figure 8: Flowchart of the Overall System

3.5.1 Flowchart - Logistic Regression

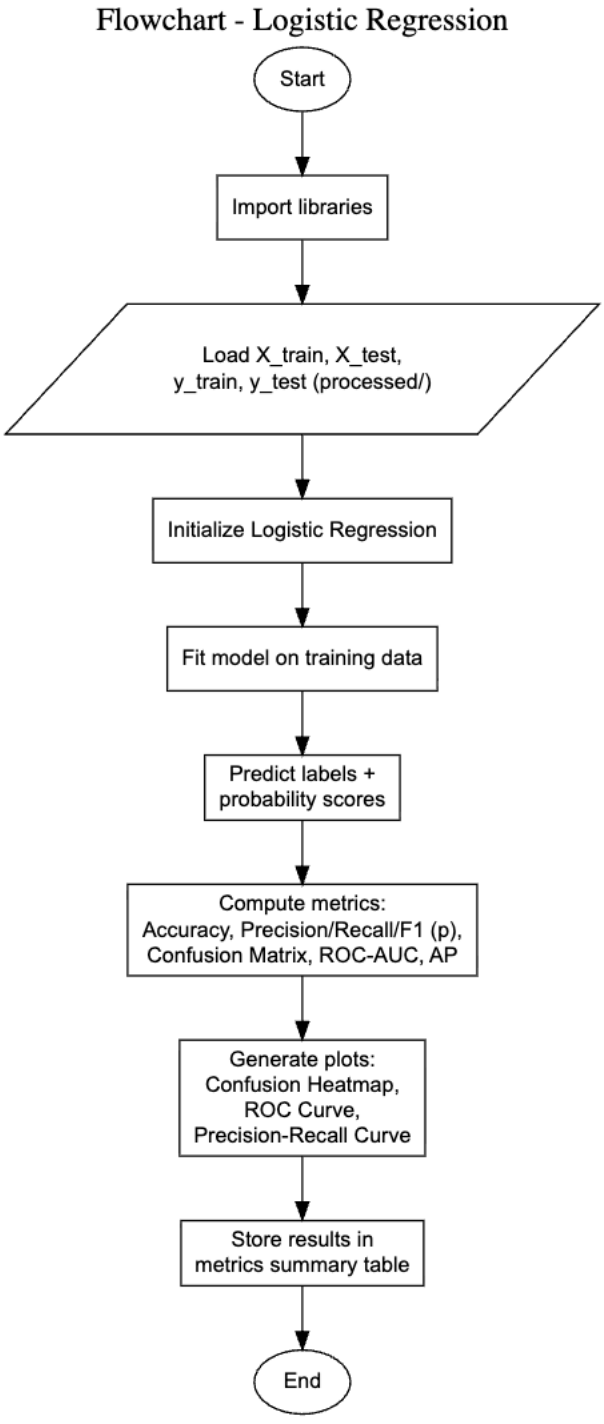

Figure 9: Flowchart for Logistic Regression

3.5.2 Flowchart – Naïve Bayes

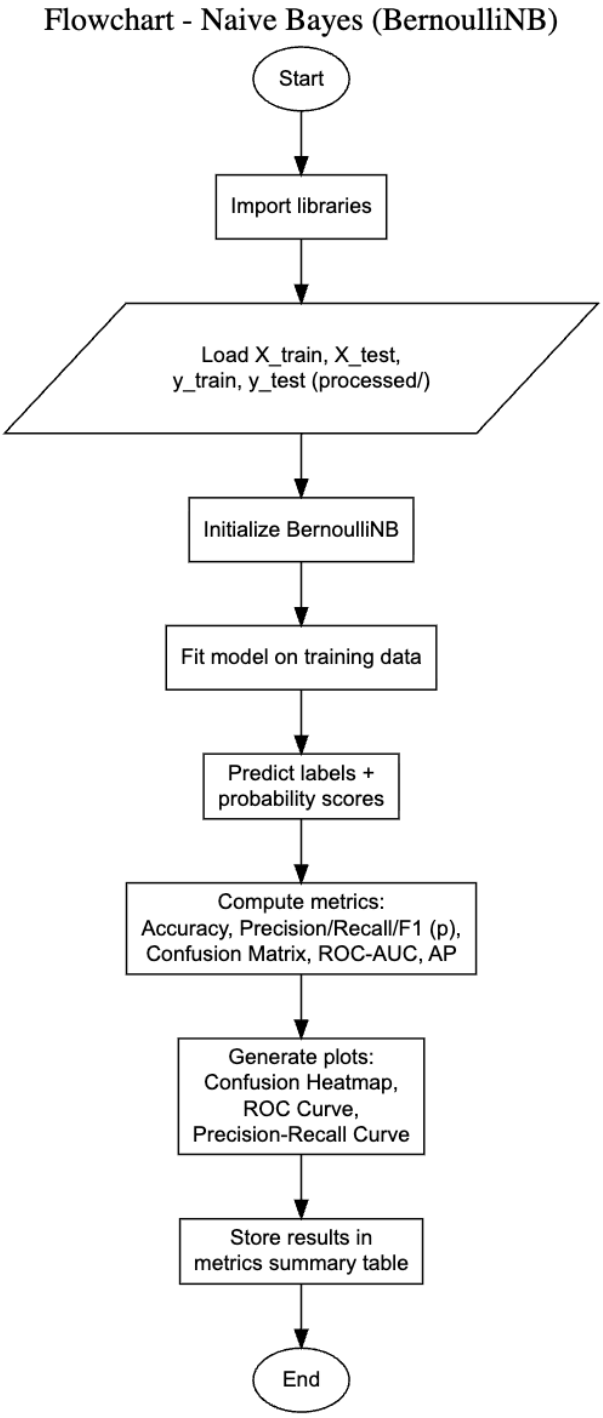

Figure 10: Flowchart for Naïve Bayes

3.5.3 Flowchart – Random Forest

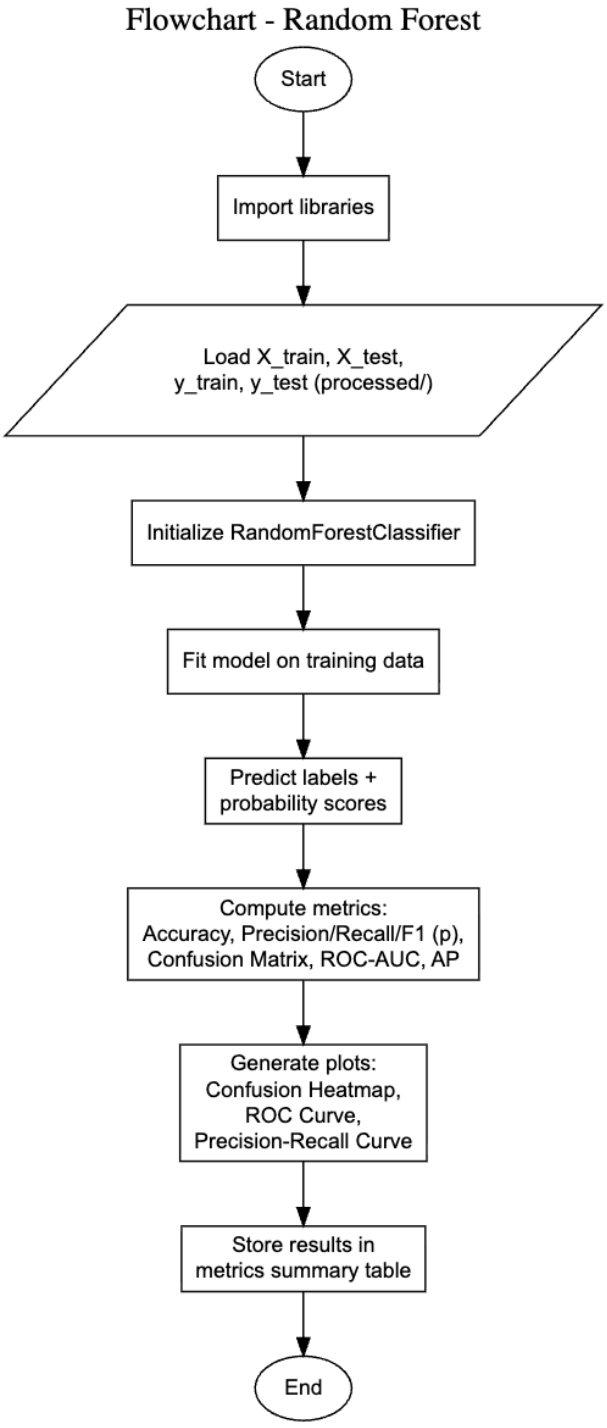

Figure 11: Flowchart for Random Forest

3.6 Data Exploration

3.6.1 Understanding the Dataset

Understanding of the dataset is essential because through it we can identify:

- Dataset size, structure, and datatypes.
- Missing or inconsistent values.
- Irrelevant columns.
- Preparation steps before

3.6.2 Observations from the dataset

- 'agaricus-lepiota.data' is the actual dataset containing one mushroom per row.
- 'agaricus-lepiota.names' is the metadata which describes what the dataset is about.
- All the attributes are categorical data (Object Datatype)
- Missing values are stored as '?', not NaN.
- Column (veil-type) has only a single unique value. As it contains no meaningful data for training models, the column can be removed.

3.6.3 Tools and Libraries Used

Python

Python was chosen as the primary development language due to the abundance of scientific computing and machine learning tools, its large and vibrant population, and usage in research in the fields of data science and high-energy physics.

Numpy

NumPy was used to process numbers with high speed and to operate with arrays effectively. It facilitates the storage of large feature matrices and label vectors and high-dimensional calculations in a compact amount of time.

Pandas

Pandas was employed in the arrangement and searching of data. It allows data loading, finding out types and structure of rows and columns, and addressing missing values during data pre-processing. Pandas is crucial for data cleaning and preparation of our project.

Matplotlib

To analyse the data and compare the results of the models Matplotlib was used to draw clear and fixed images. It was applied to plots like class balance charts, feature distribution histograms and evaluation curves like ROC and Precision Recall.

Scikit-learn

Scikit-learn served as the main library of the machine learning to create and test models. It was applied to train the classifiers of Logistic Regression and Random Forest and also to obtain the performance measures such as Accuracy, Precision, Recall, F1-score, ROC-AUC, and others.

Assigning pre-defined column names to the dataset

```python
import pandas as pd

"""
This section loads the UCI Mushroom dataset correctly.

- The file 'agaricus-lepiota.data' contains the raw data, but it does not include a header row.
- Therefore, we manually define a list of column names (COLUMNS) based on the documentation
  provided in 'agaricus-lepiota.names'.
- We use header=None so pandas treats the first row as data (not as column headers).
- We pass names=COLUMNS so the DataFrame has meaningful attribute names, making the dataset
  readable and suitable for data understanding and preparation steps.
"""

COLUMNS = [
    "class",  # edible=e, poisonous=p
    "cap-shape",
    "cap-surface",
    "cap-color",
    "bruises",
    "odor",
    "gill-attachment",
    "gill-spacing",
    "gill-size",
    "gill-color",
    "stalk-shape",
    "stalk-root",
    "stalk-surface-above-ring",
    "stalk-surface-below-ring",
    "stalk-color-above-ring",
    "stalk-color-below-ring",
    "veil-type",
    "veil-color",
    "ring-number",
    "ring-type",
    "spore-print-color",
    "population",
    "habitat",
]
```

Figure 12: Mushroom dataset loaded with predefined column names for analysis and preparation.

Loading the dataset

```python
df = pd.read_csv("agaricus-lepiota.data", header=None, names=COLUMNS)
df.head()
```

	class	cap-shape	cap-surface	cap-color	bruises	odor	gill-attachment	gill-spacing	gill-size	gill-color	...	stalk-surface-below-ring	stalk-color-above-ring	stalk-color-below-ring	veil-type	veil-color	ring-number	ring-type	spore-print-color	population
0	p	x	s	n	t	p	f	c	n	k	...	s	w	w	p	w	o	p	k	s
1	e	x	s	y	t	a	f	c	b	k	...	s	w	w	p	w	o	p	n	n
2	e	b	s	w	t	l	f	c	b	n	...	s	w	w	p	w	o	p	n	n
3	p	x	y	w	t	p	f	c	n	n	...	s	w	w	p	w	o	p	k	s
4	e	x	s	g	f	n	f	w	b	k	...	s	w	w	p	w	o	e	n	a

5 rows × 23 columns

Figure 13: Loading mushroom dataset and assigning meaningful column names from COLUMNS list

```
Signature:
pd.read_csv(
    filepath_or_buffer: 'FilePath | ReadCsvBuffer[bytes] | ReadCsvBuffer[str]',
    *,
    sep: 'str | None | lib.NoDefault' = <no_default>,
    delimiter: 'str | None | lib.NoDefault' = None,
    header: "int | Sequence[int] | None | Literal['infer']" = 'infer',
    names: 'Sequence[Hashable] | None | lib.NoDefault' = <no_default>,
    index_col: 'IndexLabel | Literal[False] | None' = None,
    usecols: 'UsecolsArgType' = None,
    dtype: 'DtypeArg | None' = None,
    engine: 'CSVEngine | None' = None,
    converters: 'Mapping[Hashable, Callable] | None' = None,
    true_values: 'list | None' = None,
    false_values: 'list | None' = None,
    skipinitialspace: 'bool' = False,
    skiprows: 'list[int] | int | Callable[[Hashable], bool] | None' = None,
    skipfooter: 'int' = 0,
    nrows: 'int | None' = None,
    na_values: 'Hashable | Iterable[Hashable] | Mapping[Hashable, Iterable[Hashable]] | None'
```

Figure 14: Signature for read_csv method

Showing the shape of the dataset

```
df.shape
```

```
(8124, 23)
```

Figure 15: Displaying the dataset's size (rows, columns)

```
Type:        property
String form: <property object at 0x126459620>
Docstring:
Return a tuple representing the dimensionality of the DataFrame.

See Also
--------
ndarray.shape : Tuple of array dimensions.

Examples
--------
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.shape
(2, 2)

>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4],
...                    'col3': [5, 6]})
>>> df.shape
(2, 3)
```

Figure 16: Signature for shape()

Statistical summary of the dataset

```
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
```

Figure 17: Showing datatypes, non-null counts, and categorical summary statistics using info()

```
Signature:
df.info(
    verbose: 'bool | None' = None,
    buf: 'WriteBuffer[str] | None' = None,
    max_cols: 'int | None' = None,
    memory_usage: 'bool | str | None' = None,
    show_counts: 'bool | None' = None,
) -> 'None'
Docstring:
Print a concise summary of a DataFrame.

This method prints information about a DataFrame including
the index dtype and columns, non-null values and memory usage.

Parameters
----------
verbose : bool, optional
    Whether to print the full summary. By default, the setting in
    ``pandas.options.display.max_info_columns`` is followed.
buf : writable buffer, defaults to sys.stdout
```

Figure 18: Signatutre for info() method

Description of the dataset

```
df.describe(include = "all")
```

	class	cap-shape	cap-surface	cap-color	bruises	odor	gill-attachment	gill-spacing	gill-size	gill-color	...	stalk-surface-below-ring	stalk-color-above-ring	stalk-color-below-ring	veil-type	veil-color	ring-number	ring-type	spore-print-color
count	8124	8124	8124	8124	8124	8124	8124	8124	8124	8124	...	8124	8124	8124	8124	8124	8124	8124	8124
unique	2	6	4	10	2	9	2	2	2	12	...	4	9	9	1	4	3	5	9
top	e	x	y	n	f	n	f	c	b	b	...	s	w	w	p	w	o	p	w
freq	4208	3656	3244	2284	4748	3528	7914	6812	5612	1728	...	4936	4464	4384	8124	7924	7488	3968	2388

4 rows × 23 columns

Figure 19: Summarizing all columns to show counts, unique values and most frequent categories using describe() method

```
Signature: df.describe(percentiles=None, include=None, exclude=None) -> 'Self'
Docstring:
Generate descriptive statistics.

Descriptive statistics include those that summarize the central
tendency, dispersion and shape of a
dataset's distribution, excluding ``NaN`` values.

Analyzes both numeric and object series, as well
as ``DataFrame`` column sets of mixed data types. The output
will vary depending on what is provided. Refer to the notes
below for more detail.

Parameters
----------
percentiles : list-like of numbers, optional
    The percentiles to include in the output. All should
    fall between 0 and 1. The default is
    ``[.25, .5, .75]``, which returns the 25th, 50th, and
    75th percentiles.
```

Figure 20: Signature for describe() method

Count of missing value in each column

```
df.isnull().sum()
```

```
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
```

Figure 21: Counting missing NaN values in each column

```
Signature: df.isnull() -> 'DataFrame'
Docstring:
DataFrame.isnull is an alias for DataFrame.isna.

Detect missing values.

Return a boolean same-sized object indicating if the values are NA.
NA values, such as None or :attr:`numpy.NaN`, gets mapped to True
values.
Everything else gets mapped to False values. Characters such as empty
strings ``''`` or :attr:`numpy.inf` are not considered NA values
(unless you set ``pandas.options.mode.use_inf_as_na = True``).

Returns
-------
DataFrame
    Mask of bool values for each element in DataFrame that
    indicates whether an element is an NA value.
```

Figure 22: Signature for isnull() method

'?' values per column

```
(df == "?").sum().sort_values(ascending=False)
```

```
stalk-root                  2480
stalk-surface-above-ring       0
population                     0
spore-print-color              0
ring-type                      0
ring-number                    0
veil-color                     0
veil-type                      0
stalk-color-below-ring         0
stalk-color-above-ring         0
stalk-surface-below-ring       0
class                          0
cap-shape                      0
stalk-shape                    0
gill-color                     0
gill-size                      0
gill-spacing                   0
gill-attachment                0
odor                           0
bruises                        0
cap-color                      0
cap-surface                    0
habitat                        0
dtype: int64
```

Figure 23: Counting '?' values per column

Target (class) Distribution

```
print(df["class"].value_counts())
print(df["class"].value_counts(normalize=True))
```

```
class
e    4208
p    3916
Name: count, dtype: int64
class
e    0.517971
p    0.482029
Name: proportion, dtype: float64
```

Figure 24: Showing distribution and class balance of target variable

3.7 Data Preparation

The dataset is now prepared for machine learning by performing data quality checks, handling missing values, removing irrelevant columns, and converting categorical values into numeric format (encoding) before applying supervised learning algorithms.

3.7.0 Creating a clean working copy of the dataset

A separate working copy is created to ensure that the original dataset is unchanged. It prevents accidental modifications during the cleaning process and makes the workflow clearer.

Method used: df_clean = df_copy()

Expected outcome: A new data frame same as the original is created for cleaning operations without affecting the original dataset.

```
# Making a clean working copy of the original dataset
df_clean = df.copy() #df.copy() -> makes duplicate df of the original df
df_clean
```

	class	cap-shape	cap-surface	cap-color	bruises	odor	gill-attachment	gill-spacing	gill-size	gill-color	...	stalk-surface-below-ring	stalk-color-above-ring	stalk-color-below-ring	veil-type	veil-color	ring-number	ring-type	spore-print-color	populatic
0	p	x	s	n	t	p	f	c	n	k	...	s	w	w	p	w	o	p	k	
1	e	x	s	y	t	a	f	c	b	k	...	s	w	w	p	w	o	p	n	
2	e	b	s	w	t	l	f	c	b	n	...	s	w	w	p	w	o	p	n	
3	p	x	y	w	t	p	f	c	n	n	...	s	w	w	p	w	o	p	k	
4	e	x	s	g	f	n	f	w	b	k	...	s	w	w	p	w	o	e	n	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
8119	e	k	s	n	f	n	a	c	b	y	...	s	o	o	p	o	o	p	b	
8120	e	x	s	n	f	n	a	c	b	y	...	s	o	o	p	n	o	p	b	
8121	e	f	s	n	f	n	a	c	b	n	...	s	o	o	p	o	o	p	b	
8122	p	k	y	n	f	y	f	c	n	b	...	k	w	w	p	w	o	e	w	
8123	e	x	s	n	f	n	a	c	b	y	...	s	o	o	p	o	o	p	o	

8124 rows × 23 columns

Figure 25: Creating a copy of original dataset to perform preparation without altering original data set using df.copy()

```
Signature: df.copy(deep: 'bool_t | None' = True) -> 'Self'
Docstring:
Make a copy of this object's indices and data.

When ``deep=True`` (default), a new object will be created with a
copy of the calling object's data and indices. Modifications to
the data or indices of the copy will not be reflected in the
original object (see notes below).

When ``deep=False``, a new object will be created without copying
the calling object's data or index (only references to the data
and index are copied). Any changes to the data of the original
will be reflected in the shallow copy (and vice versa).

.. note::
    The ``deep=False`` behaviour as described above will change
    in pandas 3.0. `Copy-on-Write
    <https://pandas.pydata.org/docs/dev/user_guide/copy_on_write.html>`__
    will be enabled by default, which means that the "shallow" copy
    is that is returned with ``deep=False`` will still avoid making
```

Figure 26: Signature for df.copy()

3.7.1 Converting '?' values into standard missing values

In the data set, missing values are not stored as NaN and instead represented by the character '?'. So, it is necessary to change the missing information into standard missing values before data cleaning.

Method used: df_clean = df_clean.replace('?',pd.NA)

Expected outcome: All occurences of '?' are replaced with NA.

```
df_clean = df_clean.replace("?", pd.NA)
df_clean
```

	class	cap-shape	cap-surface	cap-color	bruises	odor	gill-attachment	gill-spacing	gill-size	gill-color	...	stalk-surface-below-ring	stalk-color-above-ring	stalk-color-below-ring	veil-type	veil-color	ring-number	ring-type	spore-print-color	populatic
0	p	x	s	n	t	p	f	c	n	k	...	s	w	w	p	w	o	p	k	
1	e	x	s	y	t	a	f	c	b	k	...	s	w	w	p	w	o	p	n	
2	e	b	s	w	t	l	f	c	b	n	...	s	w	w	p	w	o	p	n	
3	p	x	y	w	t	p	f	c	n	n	...	s	w	w	p	w	o	p	k	
4	e	x	s	g	f	n	f	w	b	k	...	s	w	w	p	w	o	e	n	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
8119	e	k	s	n	f	n	a	c	b	y	...	s	o	o	p	o	o	p	b	
8120	e	x	s	n	f	n	a	c	b	y	...	s	o	o	p	n	o	p	b	
8121	e	f	s	n	f	n	a	c	b	n	...	s	o	o	p	o	o	p	b	
8122	p	k	y	n	f	y	f	c	n	b	...	k	w	w	p	w	o	e	w	
8123	e	x	s	n	f	n	a	c	b	y	...	s	o	o	p	o	o	p	o	

8124 rows × 23 columns

Figure 27: Replacing '?' values with standard NA missing value

3.7.2 Verifying that all '?' values are removed

The rreplacement needs to be verified after the conversion. Counting '?' values confirms that there are no '?' values remaining in the dataset.

Method used: (df_clean == "?").sum().sort_values(ascending=False)

Expected outcome: All columns show 0 for the '?' values.

```
(df_clean == "?").sum().sort_values(ascending=False)
```

```
class                       0
stalk-surface-above-ring    0
population                  0
spore-print-color           0
ring-type                   0
ring-number                 0
veil-color                  0
veil-type                   0
stalk-color-below-ring      0
stalk-color-above-ring      0
stalk-surface-below-ring    0
stalk-root                  0
cap-shape                   0
stalk-shape                 0
gill-color                  0
gill-size                   0
gill-spacing                0
gill-attachment             0
odor                        0
bruises                     0
cap-color                   0
cap-surface                 0
habitat                     0
dtype: int64
```

Figure 28: Counting '?' missing values to identify features with missing data

3.7.3 Handling missing values in the dataset

The missing '?' values are converted into NaN values. The number and percentage of missing values in each row is shown. The dataset is categorical and dropping rows can remove large samples so missing values are handled by labelling them as "missing" which preserves all rows and does not cause biasness.

Methods used:

- Df_clean.replace('?', pd.NA)
- df_clean.isnull().sum() and (df_clean.isnull().mean() to show missing counts and percentage.
- df_clean.fillna to replace missing values with label 'missing'.

Expected outcomes:

- Counts and percentage of missing values is shown.
- 0 missing counts are shown while keeping the data size unchanged.

```
missing_count = df_clean.isnull().sum()
missing_count
```

```
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                2480
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
```

Figure 29: Showing the count of missing values per row.

```
missing_percent = (df_clean.isnull().mean() * 100).round(2)
missing_percent
```

```
class                        0.00
cap-shape                    0.00
cap-surface                  0.00
cap-color                    0.00
bruises                      0.00
odor                         0.00
gill-attachment              0.00
gill-spacing                 0.00
gill-size                    0.00
gill-color                   0.00
stalk-shape                  0.00
stalk-root                  30.53
stalk-surface-above-ring     0.00
stalk-surface-below-ring     0.00
stalk-color-above-ring       0.00
stalk-color-below-ring       0.00
veil-type                    0.00
veil-color                   0.00
ring-number                  0.00
ring-type                    0.00
spore-print-color            0.00
population                   0.00
habitat                      0.00
dtype: float64
```

Figure 30: Showing the percentage of missing values per row.

```
# Fill missing categorical values with a dedicated category label
df_clean = df_clean.fillna("missing")

# Confirm that no missing values remain
df_clean.isnull().sum().sort_values(ascending=False)
```

```
class                        0
stalk-surface-above-ring     0
population                   0
spore-print-color            0
ring-type                    0
ring-number                  0
veil-color                   0
veil-type                    0
stalk-color-below-ring       0
stalk-color-above-ring       0
stalk-surface-below-ring     0
stalk-root                   0
cap-shape                    0
stalk-shape                  0
gill-color                   0
gill-size                    0
gill-spacing                 0
gill-attachment              0
odor                         0
bruises                      0
cap-color                    0
cap-surface                  0
habitat                      0
dtype: int64
```

Figure 31: Filling missing values with label 'missing' and confirming that no missing values remain

3.7.4 Identifying and removing non-informative features

A feature with only one unique value does not help to differenciate whether the mushroom is poisonous or edible. Such features are identified and dropped. The column ‘veil-type’ has only one unique value, so it needs to be removed.

Methods used:

- df_clean.nunique() to list number of unique values.
- df_clean.drop()

Expected outcomes:

- List showing unique values per column.
- Columns with only one unique value will be dropped.

```
df_clean.nunique().sort_values()
```

```
veil-type                  1
class                      2
bruises                    2
gill-attachment            2
gill-spacing               2
gill-size                  2
stalk-shape                2
ring-number                3
cap-surface                4
veil-color                 4
stalk-surface-below-ring   4
stalk-surface-above-ring   4
ring-type                  5
stalk-root                 5
cap-shape                  6
population                 6
habitat                    7
stalk-color-above-ring     9
stalk-color-below-ring     9
odor                       9
spore-print-color          9
cap-color                 10
gill-color                12
dtype: int64
```

Figure 32: Using nunique() to identify constant features per column.

```
# Drop the constant (non-informative) feature
df_clean = df_clean.drop(columns=["veil-type"])

# Verify it is removed
df_clean.shape, ("veil-type" in df_clean.columns)
```

```
((8124, 22), False)
```

Figure 33: Dropping the constant column and confirming it is removed from the dataset.

```
Signature:
df_clean.fillna(
    value: 'Hashable | Mapping | Series | DataFrame | None' = None,
    *,
    method: 'FillnaOptions | None' = None,
    axis: 'Axis | None' = None,
    inplace: 'bool_t' = False,
    limit: 'int | None' = None,
    downcast: 'dict | None | lib.NoDefault' = <no_default>,
) -> 'Self | None'
Docstring:
Fill NA/NaN values using the specified method.

Parameters
----------
value : scalar, dict, Series, or DataFrame
    Value to use to fill holes (e.g. 0), alternately a
    dict/Series/DataFrame of values specifying which value to use for
    each index (for a Series) or column (for a DataFrame).  Values not
    in the dict/Series/DataFrame will not be filled. This value cannot
```

Figure 34: Signature for df_clean.fillna()

```
Signature:
df_clean.drop(
    labels: 'IndexLabel | None' = None,
    *,
    axis: 'Axis' = 0,
    index: 'IndexLabel | None' = None,
    columns: 'IndexLabel | None' = None,
    level: 'Level | None' = None,
    inplace: 'bool' = False,
    errors: 'IgnoreRaise' = 'raise',
) -> 'DataFrame | None'
Docstring:
Drop specified labels from rows or columns.

Remove rows or columns by specifying label names and corresponding
axis, or by directly specifying index or column names. When using a
multi-index, labels on different levels can be removed by specifying
the level. See the :ref:`user guide <advanced.shown_levels>`
for more information about the now unused levels.
```

Figure 35: Signature for df_clean.drop

3.7.5 Separating features and target variable

Supervised learning needs separation between input variables (features) and output variables (targets). In our dataset, 'class' is the target label that indicates if a mushroom is edible or poisonous. All other columns are treated as categorical input features.

Methods used:

- X = df_clean.drop(columns=["class"])
- y = df_clean["class"]

Expected outcome:

- X should contain only feature columns and y should contain class labels.
- Shapes confirm the correct split of features and target.

```
# Separate input features and target label
X = df_clean.drop(columns=["class"])
y = df_clean["class"]

X.shape, y.shape
```

```
((8124, 21), (8124,))
```

Figure 36: Separating predictor features (X) and target label (y) by isolating 'class' column for supervised learning.

3.7.6 One-hot encoding categorical features

The dataset contains categorical features encoded as letters. But ML algorithms need numeric values which is possible using one-hot encoding. It converts each category into machine understandable bianary number (0 or 1).

Method used:

- X_encoded = pd.get_dummies(X)

Expected outcome:

- The number of rows remain unchanged.
- The number of columns increase as each categorical level is expanded into separate bianary feature.
- Resulting matrix is suitable to train machine learning classifiers.

```
# Convert categorical features into numeric features using one-hot encoding
X_encoded = pd.get_dummies(X)

X_encoded.shape
```

```
(8124, 116)
```

Figure 37: One-hot encoding using pd.get_dummies()

Observed Outcome:

- In the encoded dataset, all 8124 rows are retained and 21 columns are expended into 116 numeric one-hot encoded columns.

```
Signature:
pd.get_dummies(
    data,
    prefix=None,
    prefix_sep: 'str | Iterable[str] | dict[str, str]' = '_',
    dummy_na: 'bool' = False,
    columns=None,
    sparse: 'bool' = False,
    drop_first: 'bool' = False,
    dtype: 'NpDtype | None' = None,
) -> 'DataFrame'
Docstring:
Convert categorical variable into dummy/indicator variables.

Each variable is converted in as many 0/1 variables as there are different
values. Columns in the output are each named after a value; if the input is
a DataFrame, the name of the original variable is prepended to the value.

Parameters
----------
```

Figure 38: Signature for pd.get_dummies()

3.7.7 Splitting the dataset into training and testing sets

The mushroom dataset is split into training and testing sets to evaluate model performance impartially. Patters are learned by the model from the training set, and it is evaluated on test data. Stratified splitting is used to maintain the class distribution of edible/poisonous across both data sets.

Method used:

- X_train, X_test, y_train, y_test = train_test_split(X_encoded, y, test_size=0.20, random_state=42, stratify=y)

Expected outcome:

- 80% of data is reparated for training and 20% is separated for testing.
- Class distribution in both sets remain similar.
- Resulting shapes show correct split into training and testing inputs and training and testing labels.

```
"""
- test_size=0.20: uses 20% of data for testing and 80% for training.
- random_state=42: fixes the random seed so the split is reproducible.
- stratify=y: keeps the same class (e/p) proportions in both train and test sets.
"""
from sklearn.model_selection import train_test_split

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X_encoded, y,
    test_size=0.20,
    random_state=42,
    stratify=y
)

X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

```
((6499, 116), (1625, 116), (6499,), (1625,))
```

Figure 39: Splitting one-hot encoded dataset into training and testing sets.

3.7.8 Verifying class balance after stratified splitting

Stratified splitting ensures that class proportions are consistent in training and testing sets. Verifying the class balance makes sures that the split is fair and results will not be biased.

Method used:

- y_train.value_counts(normalize=True)
- y_test.value_counts(normalize=True)

Expected outcome:

- The proportions of edible and poisonous mushroom in y_train and y)test should be very close which confirms that stratify = y has worked.

```
: # Check class distribution in training and test sets (proportions)
  y_train.value_counts(normalize=True), y_test.value_counts(normalize=True)
```

```
: (class
   e    0.517926
   p    0.482074
   Name: proportion, dtype: float64,
   class
   e    0.518154
   p    0.481846
   Name: proportion, dtype: float64)
```

Figure 40: Comparing class proportions in training and testing datasets.

3.7.9 Ensuring numeric data types and no missing values

Before training the models, the datasets must be validated so that there is no missing values and all the features are numeric data. This step helps prevent training errors and ensures that the data is compatible with scikit-learn classifiers.

Methods used:

- X_train.isnull().sum().sum()
- X_test.isnull().sum().sum()
- X_train.dtypes.value_counts()

Expected outcome:

- The count of missing values for X_train and X_test should be 0.
- Datatype of the features should be numeric only (integer).

```
# Check for any remaining missing values in training and test sets
X_train.isnull().sum().sum(), X_test.isnull().sum().sum()
```

```
(0, 0)
```

```
# Confirm all features are numeric after one-hot encoding
X_train.dtypes.value_counts()
```

```
bool    116
Name: count, dtype: int64
```

Figure 41: Ensuring dataset has no missing values and the data are all numeric.

3.7.10 Ensuring numeric data types and no missing values

All the models need to be trained and evaluated on the same data sets. To ensure that, the prepared and processed datasets are saved to disk. This helps prevent inconsistent preprocessing across multiple different notebooks, makes the workflow cleaner and improved reproducibility. One-hot encoded feature lists are also saved so as the future predictions can be aligned to the same feature space.

Methods used:

- to_csv() to save X_train, X_test, y_train, y_test in the 'processed' folder.
- Saving feature_columns from X_encoded.columns to preserve the encoded feature set.

Expected outcome:

- A folder 'processed' is created and in the folder there are saved datasets for X_train, X_test, y_train, y_test and feature columns so that they can be loaded in other notebooks without the need of data preparation again and again.

```
from pathlib import Path
import pandas as pd

# Create an output folder inside your current project directory
out_dir = Path("processed")
out_dir.mkdir(exist_ok=True)

# Save train/test splits (features + labels)
X_train.to_csv(out_dir / "X_train.csv", index=False)
X_test.to_csv(out_dir / "X_test.csv", index=False)
y_train.to_csv(out_dir / "y_train.csv", index=False)
y_test.to_csv(out_dir / "y_test.csv", index=False)

# Save the one-hot encoded feature names (important for consistent model input later)
pd.Series(X_encoded.columns, name="feature").to_csv(out_dir / "feature_columns.csv", index=False)
```

Figure 42: Saving processed train/test data sets and encoded feature columns in 'processed' folder for reuse.

3.8 Description of AI Algorithms used

3.8.1 Logistic Regression

Logistic Regression The algorithm is supervised learning algorithm that is used in the binary classification. It will learn a set of weights of the input features and create a probability score that a sample is of a particular class (e.g., poisonous). The purpose of using Logistic Regression as a baseline model in this project is that it is easy, quick to train and can be used as a reference point to other more complicated methods. It is also useful in demonstrating the level of improvement that is achieved in switching it to non-linear relationship models.

Advantages:

- Easy to describe and simple to understand making it a robust baseline model.
- Trains quickly and will be useful in cases where the classes are separable in a relatively linear manner.

Disadvantages:

- Fail to perform well when the relationship between features and class is either non-linear or feature interactions.
- Does not work well when the decisions involve a combination of traits, compared to tree-based models.

Importing Essential libraries

```
# Import libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (
    accuracy_score, classification_report, confusion_matrix,
    precision_score, recall_score, f1_score,
    roc_curve, roc_auc_score,
    precision_recall_curve, average_precision_score
)
```

Figure 43: Importing essential libraries: Logistic Regression

Loading prepared train/test datasets

Logistic regression is trained and evaluated using the pre-processed dataset to ensure fair comparision, consistency and reproducibility.

Methods used:

- pd.read_csv() to load pre-processed datasets
- .squeeze() to convert columns into 1D series which is preferred by scikit-learn classifiers.

Expected outcome:

- The datasets should load and match the dimensions for X_train and X_test, and match counts for y_train and y_test

```
"""
Load processed train/test files created in DataPreparation.ipynb
"""
X_train = pd.read_csv("processed/X_train.csv")
X_test  = pd.read_csv("processed/X_test.csv")

y_train = pd.read_csv("processed/y_train.csv").squeeze()
y_test  = pd.read_csv("processed/y_test.csv").squeeze()

X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

```
((6499, 116), (1625, 116), (6499,), (1625,))
```

Figure 44: Loading pre-processed train/test dataset for Logistic Regression.

Training Logistic Regression Classifier

Logistic Regression is trained as a baseline model on the encoded dataset.

Methods:

- LogisticRegression()
- .fit(X_train, y_train)

Expected outcome:

- Trained model ready to classify mushrooms as e or p.

```
"""
Train Logistic Regression model.
- max_iter is increased to ensure convergence.
- solver='liblinear' is commonly used for binary classification.
"""
lr_model = LogisticRegression(max_iter=1000, solver="liblinear")
lr_model.fit(X_train, y_train)

lr_model
```

```
LogisticRegression(max_iter=1000, solver='liblinear')
```

Figure 45: Training Logistic Regression Model.

```
Signature: lr_model.fit(X, y, sample_weight=None)
Docstring:
Fit the model according to the given training data.

Parameters
----------
X : {array-like, sparse matrix} of shape (n_samples, n_features)
    Training vector, where `n_samples` is the number of samples and
    `n_features` is the number of features.

y : array-like of shape (n_samples,)
    Target vector relative to X.

sample_weight : array-like of shape (n_samples,) default=None
    Array of weights that are assigned to individual samples.
    If not provided, then each sample is given unit weight.

    .. versionadded:: 0.17
       *sample_weight* support to LogisticRegression.
```

Figure 46: Signature for .fit()

Predicting on the test set

Predictions are made on unseen test to check generalisation. Probability scores are extracted for ROC and Precision-Recall Curves.

Methods used:

- .predict(X_test) to generate predicted labels
- .predict_proba(X_test) to generate probability scores

Expected outcomes:

- y_pred_lr contains labels ‘e/p’ for each test sample.
- y_score_lr contains probability for poisonous mushrooms.

```
y_pred = lr_model.predict(X_test)

proba = lr_model.predict_proba(X_test)
p_index = list(lr_model.classes_).index("p")   # get column index for poisonous
y_score = proba[:, p_index]                    # probability of poisonous

y_pred[:10], y_score[:10]
```

```
(array(['p', 'p', 'e', 'p', 'p', 'e', 'e', 'e', 'e', 'p'], dtype=object),
 array([9.98625400e-01, 9.99229277e-01, 6.06164001e-05, 9.99956883e-01,
        9.98990661e-01, 3.54398214e-02, 1.12563159e-03, 2.07995935e-05,
        3.39553482e-04, 9.96553276e-01]))
```

Figure 47: Predicting test lables and poisonous-class probability scores using Logistic Regression.

Calculating model accuracy

Accuracy is calculated to measure the correctness of predictions on the test data set.

Methods used:

- accuracy_score(y_test, y_pred_lr)

Expected outcome:

- Value indicating overall performance of the test.

```
acc = accuracy_score(y_test, y_pred)
acc
```

```
0.9993846153846154
```

Figure 48: Calculating test accuracy for Logistic Regression.

```
Signature: accuracy_score(y_true, y_pred, *, normalize=True, sample_weight=None)
Docstring:
Accuracy classification score.

In multilabel classification, this function computes subset accuracy:
the set of labels predicted for a sample must *exactly* match the
corresponding set of labels in y_true.

Read more in the :ref:`User Guide <accuracy_score>`.

Parameters
----------
y_true : 1d array-like, or label indicator array / sparse matrix
    Ground truth (correct) labels.

y_pred : 1d array-like, or label indicator array / sparse matrix
    Predicted labels, as returned by a classifier.

normalize : bool, default=True
    If ``False``, return the number of correctly classified samples.
```

Figure 49: Signature for accuracy_score()

Confusion Matrix

Confusion matrix shows the summary of correct and incorrect predictions..False negative is the most safety-critical error which means a poisonous mushroom is predicted as edible.

Methods used:

- confusion_matrix(y_test, y_pred_lr)

Expected outcome:

- Matrix showing counts of TN, FP, FN, TP and error distribution.

```
print(classification_report(y_test, y_pred, target_names=["Edible (e)", "Poisonous (p)"]))

cm = confusion_matrix(y_test, y_pred, labels=["e", "p"])
cm_df = pd.DataFrame(cm, index=["Actual_e", "Actual_p"], columns=["Pred_e", "Pred_p"])
cm_df
```

```
                precision    recall  f1-score   support

   Edible (e)       1.00      1.00      1.00       842
Poisonous (p)       1.00      1.00      1.00       783

     accuracy                           1.00      1625
    macro avg       1.00      1.00      1.00      1625
 weighted avg       1.00      1.00      1.00      1625
```

	Pred_e	Pred_p
Actual_e	842	0
Actual_p	1	782

Figure 50: Summary of correct and incorrect Logistic Regression Predictions using confusion matrix.

```
Signature:
confusion_matrix(
    y_true,
    y_pred,
    *,
    labels=None,
    sample_weight=None,
    normalize=None,
)
Docstring:
Compute confusion matrix to evaluate the accuracy of a classification.

By definition a confusion matrix :math:`C` is such that :math:`C_{i, j}`
is equal to the number of observations known to be in group :math:`i` and
predicted to be in group :math:`j`.

Thus in binary classification, the count of true negatives is
:math:`C_{0,0}`, false negatives is :math:`C_{1,0}`, true positives is
:math:`C_{1,1}` and false positives is :math:`C_{0,1}`.
```

Figure 51: Signature for confusion_matrix.

Evaluation of metrics summary table

A table is made for easy comparison with Naïve Bayes and Random Forest.

Methods used:

- precision_score(pos_label="p")
- recall_score(pos_label="p")
- f1_score(pos_label="p")
- roc_auc_score()

Expected outcome:

- A table containing Accuracy, Precision, Recall, F1, and ROC-AUC

Table 1: Evaluation metrics summary table for Logistic Regression

```
"""
Metrics table
Poisonous (p) is treated as the positive class.
"""
roc_auc = roc_auc_score((y_test == "p").astype(int), y_score)

metrics_table = pd.DataFrame({
    "METRIC": ["ACCURACY", "PRECISION (p)", "RECALL (p)", "F1-SCORE (p)", "ROC-AUC"],
    "VALUE": [
        accuracy_score(y_test, y_pred),
        precision_score(y_test, y_pred, pos_label="p"),
        recall_score(y_test, y_pred, pos_label="p"),
        f1_score(y_test, y_pred, pos_label="p"),
        roc_auc
    ]
})
metrics_table["VALUE"] = metrics_table["VALUE"].round(4)
metrics_table
```

	METRIC	VALUE
0	ACCURACY	0.9994
1	PRECISION (p)	1.0000
2	RECALL (p)	0.9987
3	F1-SCORE (p)	0.9994
4	ROC-AUC	1.0000

Evaluation plots (Confusion Matrix Heatmap, ROC, Precision-Recall)

Error patterns and threshold-based performance are interpreted using visual plots.

Methods used:

- confusion_matrix()
- roc_curve()
- roc_auc_score()
- precision_recall_curve()
- average_precision_score()

Expected outcomes:

- Confusion heatmap showing misclassification distribution
- ROC curve with AUC value (separability across thresholds)
- Precision-Recall curve with AP value (poisonous detection quality)

```
[40]: # Confusion Matrix Heatmap
fig, ax = plt.subplots()
im = ax.imshow(cm, interpolation="nearest")
ax.set_title("Confusion Matrix — Logistic Regression")
ax.set_xticks([0, 1]); ax.set_yticks([0, 1])
ax.set_xticklabels(["Edible (e)", "Poisonous (p)"])
ax.set_yticklabels(["Edible (e)", "Poisonous (p)"])
ax.set_xlabel("Predicted Label")
ax.set_ylabel("True Label")

for i in range(cm.shape[0]):
    for j in range(cm.shape[1]):
        ax.text(j, i, cm[i, j], ha="center", va="center")

fig.colorbar(im, ax=ax)
plt.show()
```

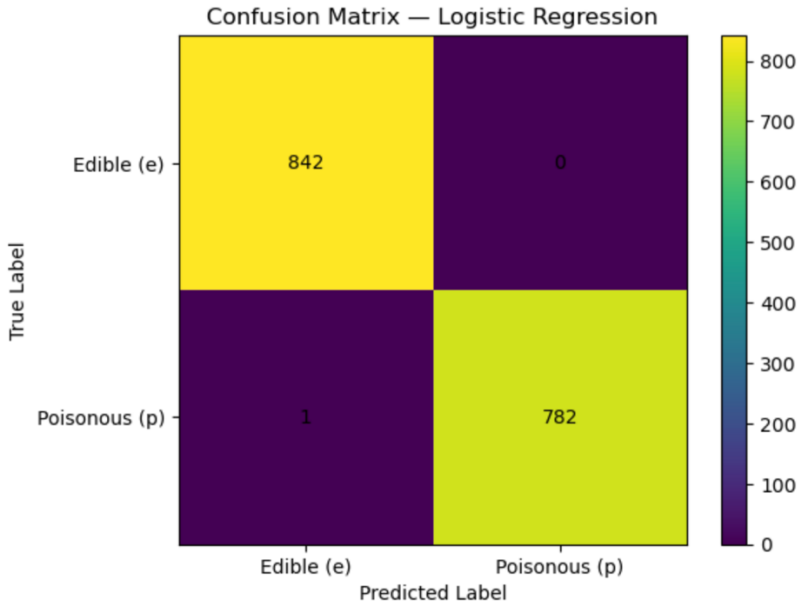

Figure 52: Confusion Matrix Heatmap for Logistic Regression.

Key Takeaways:

- Logistic Regression made 0 false positives i.e. 0 edible mushrooms were predicted as poisonous.
- It made only 1 false negative i.e. 1 poisonous mushroom was predicted as edible which is a costly mistake.

```
#ROC Curve
fpr, tpr, _ = roc_curve(y_test, y_score, pos_label="p")
plt.figure()
plt.plot(fpr, tpr, label=f"AUC = {roc_auc:.3f}")
plt.plot([0, 1], [0, 1], linestyle="--")
plt.title("ROC Curve – Logistic Regression")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.legend()
plt.show()
```

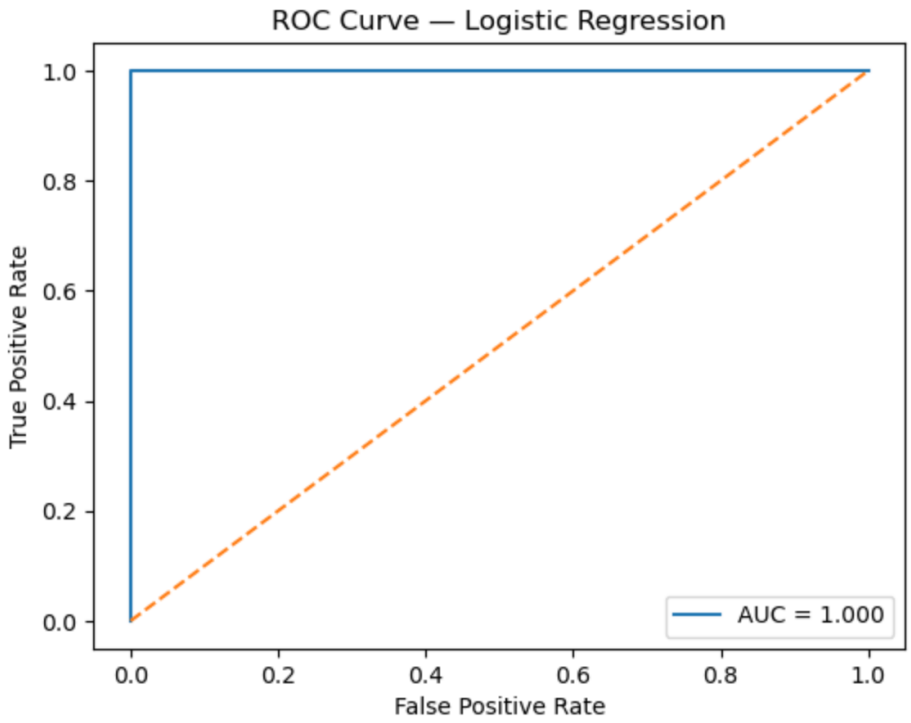

Figure 53: ROC Curve for Logistic Regression.

Key Takeaways:

- AUC = 1.000 which means it separates edible vs poisonous almost perfectly across all thresholds.
- The ROC curve is near top left which means extremely high true positive rate with almost no false positives.

```python
#Precision–Recall Curve
precision, recall, _ = precision_recall_curve(y_test, y_score, pos_label="p")
ap = average_precision_score((y_test == "p").astype(int), y_score)

plt.figure()
plt.plot(recall, precision, label=f"AP = {ap:.3f}")
plt.title("Precision–Recall Curve – Logistic Regression")
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.legend()
plt.show()
```

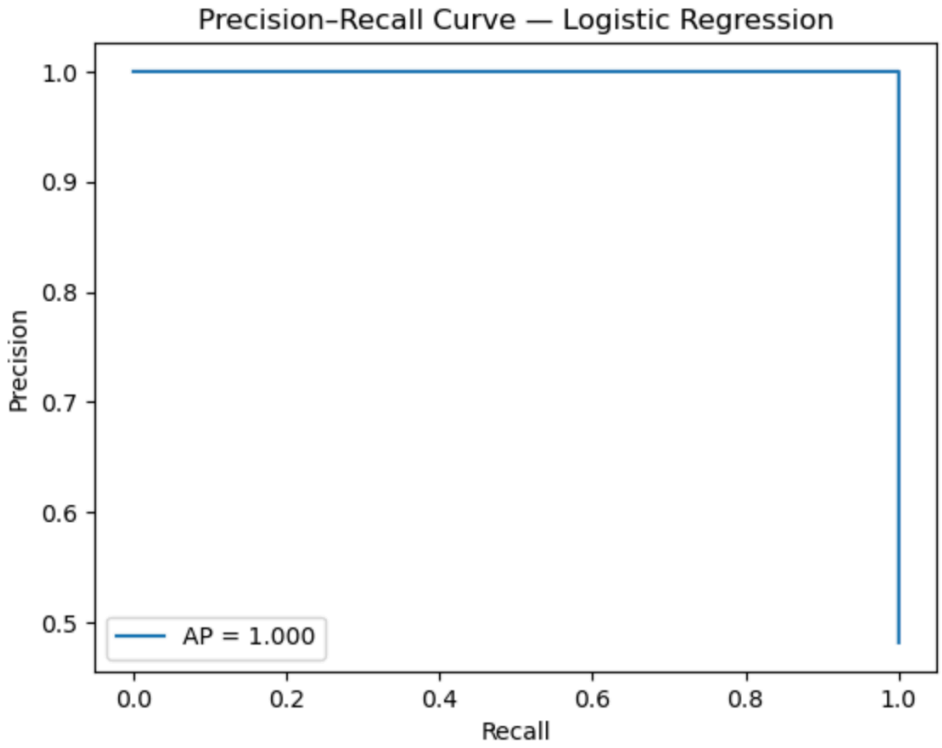

Figure 54: Precision-Recall Curve for Logistic Regression.

Key Takeaways:

- AP = 1.000 which means Logistic Regression has almost perfect precision and recall for poisonous class across thresholds.
- The curve is at top-right meaning there is almost no trade-off between finding poisonous mushrooms and avoiding false alarms

3.8.2 Naïve Bayes

Naive bayes is a probabilistic classifier which operates under the Bayes theorem. It uses the estimates of the probability of observed values of features to each class and determines the most probable class. The naive component involves the fact that features play a separate role in the eventual decision given the knowledge of the class. Although this does not necessarily apply to real data, Naive Bayes can be effective with categorical data that are structured and is computationally economical. It has been made part of this report since it is a common foundation on which the classification activity can be carried out and offers a handy comparison to Logistic Regression and Random Forest.

Advantages:

- It is extensively fast to train and predict, which is why it is helpful in the creation of a fast baseline.
- Works fairly well with limited training information and gives prediction in terms of probability.

Disadvantages:

- Usually difficult to assume, that features are independent over given the class.
- As feature interactions are important (as is the case with real classification tasks), performance may also fail.

Loading prepared train/test datasets

Naïve Bayes model is trained and evaluated using our pre-processed dataset to ensure consistent and reproducible workflow. Loading the pre-processed datasets helps avoid repeating data preparation again and again and ensures that all models are compared fairly using the same data.

Methods used:

- pd.read_csv to load the processed datasets.
- .squeeze() converts columns into 1D series (preferred format for scikit-learn classifiers)

Expected outcome:

- The datasets should load and match the dimensions for X_train and X_test, and match counts for y_train and y_test

```
"""
Load the prepared train/test datasets created during DataPreparation.ipynb.

- pd.read_csv(): loads the saved split files from disk.
- squeeze(): converts a single-column DataFrame into a 1D Series (expected by scikit-learn).
"""

import pandas as pd

X_train = pd.read_csv("processed/X_train.csv")
X_test  = pd.read_csv("processed/X_test.csv")

y_train = pd.read_csv("processed/y_train.csv").squeeze()
y_test  = pd.read_csv("processed/y_test.csv").squeeze()

X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

```
((6499, 116), (1625, 116), (6499,), (1625,))
```

Figure 55: Loading pre-processed datasets.

Importing required Libraries

```
#Importing Libraries
import pandas as pd

from sklearn.naive_bayes import BernoulliNB
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
```

Figure 56: Importing required libraries: Naïve Bayes.

Training Naïve Bayes Classifier

After one-hot encoding, all the features from the dataset are binary, therefore BernoulliNB is selected. Training estimates class priors and conditional probabilities of each bianary feature given the class.

Methods used:

- BernoulliNB() to initialize Naïve Bayes classifier for bianary inputs.
- fit(X_train, y_train) to learn model parameters from training data

Expected outcome:

- Trained Naïve Bayes model capable of predicting edible and poisonous mushrooms from unseen data.

```
"""
Train a Bernoulli Naïve Bayes classifier.

Why BernoulliNB?
- Bernoulli Naïve Bayes is designed for binary features (0/1), matching one-hot encoded inputs.
"""
nb_model = BernoulliNB()
nb_model.fit(X_train, y_train)

nb_model
```

```
BernoulliNB
BernoulliNB()
```

Figure 57: Fitting Bernoulli Naive Bayes model using encoded training set.

Predicting on the test set

Predictions must be generated on the test set which are unseen during the training of the dataset in order to evaluate how well the model is able to generalize.

Method used:

- predict(X_test) to produce predicted class labels for each test sample

Expected outcome:

- Prediction vector 'y_pred_nb' containing lables 'e or p' for every test instance.

```
"""
Generate predictions for the test set.

- predict(): returns the predicted class label for each test sample.
"""
y_pred_nb = nb_model.predict(X_test)

y_pred_nb
```

```
array(['p', 'e', 'e', ..., 'e', 'e', 'p'], dtype='<U1')
```

Figure 58: Predicting e/p classes using trained Naive Bayes model.

```
Signature: nb_model.predict(X)
Docstring:
Perform classification on an array of test vectors X.

Parameters
----------
X : array-like of shape (n_samples, n_features)
    The input samples.

Returns
-------
C : ndarray of shape (n_samples,)
    Predicted target values for X.
File:      /opt/anaconda3/lib/python3.12/site-
packages/sklearn/naive_bayes.py
Type:      method
```

Figure 59: Signature for .predict()

Model Evaluation

Performance on unseen data is quantified by model evaluation. Class-wise metrics are reported as accidentally misclassifying mushrooms can be disastrous.

Methods used:

- accuracy_score() to calculate the overall correctness
- confusion_matrix() to show true/false predictions per class
- classification_report() to report precision, recall and F1-score

Expected outcomes:

- Accuracy value for the overall performance of the model.
- A confusion matrix showing where misclassifications occur.
- A classification report showing precision/recall trade-offs.

```
"""
Compute overall accuracy.

- accuracy_score(): proportion of correct predictions on the test set.
"""
acc_nb = accuracy_score(y_test, y_pred_nb)
acc_nb
```

```
[24]:
0.932923076923077
```

Figure 60: Calculating test accuracy to measure the correctness of prediction.

```
Signature: accuracy_score(y_true, y_pred, *,
normalize=True, sample_weight=None)
Docstring:
Accuracy classification score.

In multilabel classification, this function computes
subset accuracy:
the set of labels predicted for a sample must
*exactly* match the
corresponding set of labels in y_true.

Read more in the :ref:`User Guide <accuracy_score>`.

Parameters
----------
y_true : 1d array-like, or label indicator array /
sparse matrix
    Ground truth (correct) labels.

y_pred : 1d array-like, or label indicator array /
```

Figure 61: Signature for accuracy_score

```
"""
Create a labelled confusion matrix for clearer interpretation.

Rows = actual class, Columns = predicted class
"""
cm_nb = confusion_matrix(y_test, y_pred_nb, labels=["e", "p"])

cm_nb_df = pd.DataFrame(
    cm_nb,
    index=["Actual_e", "Actual_p"],
    columns=["Pred_e", "Pred_p"]
)

cm_nb_df
```

	Pred_e	Pred_p
Actual_e	829	13
Actual_p	96	687

Figure 62: Confusion matrix to summarise correct and incorrect predictions for e/p classes.

```
Signature:
confusion_matrix(
    y_true,
    y_pred,
    *,
    labels=None,
    sample_weight=None,
    normalize=None,
)
Docstring:
Compute confusion matrix to evaluate the accuracy of a classification.

By definition a confusion matrix :math:`C` is such that :math:`C_{i, j}`
is equal to the number of observations known to be in group :math:`i` and
predicted to be in group :math:`j`.

Thus in binary classification, the count of true negatives is
:math:`C_{0,0}`, false negatives is :math:`C_{1,0}`, true positives is
:math:`C_{1,1}` and false positives is :math:`C_{0,1}`.
```

Figure 63: Signature for confusion_matrix

```
"""
Generate a detailed classification report.

- precision: correctness of positive predictions
- recall: ability to find all samples of a class
- f1-score: balance of precision and recall
"""
print(classification_report(y_test, y_pred_nb, target_names=["Edible (e)", "Poisonous (p)"]))
```

```
              precision    recall  f1-score   support

   Edible (e)       0.90      0.98      0.94       842
Poisonous (p)       0.98      0.88      0.93       783

     accuracy                           0.93      1625
    macro avg       0.94      0.93      0.93      1625
 weighted avg       0.94      0.93      0.93      1625
```

Figure 64: Class-wise precision, recall and F1-score for e/p predictions

Evaluation metrics summary table

A table is created containing key evaluation metrics for Naïve Bayes model to easily compare the results with Logistic Regression and Random Forest.

Methods used:

- accuracy_score()
- precision_score(pos_label="p"), recall_score(pos_label="p")
- f1_score(pos_label="p")
- roc_auc_score()
- average_precision_score()

Expected outcome:

- A table showing Accuracy, Precision, Recall, F1, ROC-AUC, and AP.

Table 2: Evaluation metrics summary table for Naive Bayes.

```
import pandas as pd
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

accuracy_nb  = accuracy_score(y_test, y_pred_nb)
precision_nb = precision_score(y_test, y_pred_nb, pos_label="p")
recall_nb    = recall_score(y_test, y_pred_nb, pos_label="p")
f1_nb        = f1_score(y_test, y_pred_nb, pos_label="p")

roc_auc_score_nb = auc_score
ap_score_nb      = ap_score

metrics_nb_table = pd.DataFrame({
    "METRIC": ["ACCURACY", "PRECISION (p)", "RECALL (p)", "F1-SCORE (p)", "ROC-AUC", "AP"],
    "VALUE":  [accuracy_nb, precision_nb, recall_nb, f1_nb, roc_auc_score_nb, ap_score_nb]
})

metrics_nb_table["VALUE"] = metrics_nb_table["VALUE"].round(4)
metrics_nb_table
```

	METRIC	VALUE
0	ACCURACY	0.9329
1	PRECISION (p)	0.9814
2	RECALL (p)	0.8774
3	F1-SCORE (p)	0.9265
4	ROC-AUC	0.9954
5	AP	0.9952

Evaluation plots for Confusion Matrix, ROC, and Precision-Recall

- Confusion Matrix heatmap shows how often are classes misclassified, visualises correct and incorrect predictions for each class.
- ROC Curve shows the trade-off between true positive rate and false positive rate across thressholds.
- Precision Recall Curve focuses on positive class performance.

Methods used:

- confusion_matrix()
- roc_curve()
- roc_auc_score()
- precision_recall_curve()
- average_precision_score()

Expected outcomes:

- Heatmap showing correct vs incorrect predictions
- ROC Curve with curve ideally close to top-left and AUC Value close to 1
- PR Curve showing trafe-off between precision and recall for poisonous class with AP score close to 1.

```python
import matplotlib.pyplot as plt
import numpy as np

from sklearn.metrics import (
    confusion_matrix,
    roc_curve,
    roc_auc_score,
    precision_recall_curve,
    average_precision_score
)

cm = confusion_matrix(y_test, y_pred_nb, labels=["e", "p"])

fig, ax = plt.subplots()
im = ax.imshow(cm, interpolation="nearest")
ax.set_title("Confusion Matrix — Naïve Bayes")

ax.set_xticks([0, 1])
ax.set_yticks([0, 1])
ax.set_xticklabels(["Edible (e)", "Poisonous (p)"])
ax.set_yticklabels(["Edible (e)", "Poisonous (p)"])
ax.set_xlabel("Predicted Label")
ax.set_ylabel("True Label")

# Add counts on the heatmap cells
for i in range(cm.shape[0]):
    for j in range(cm.shape[1]):
        ax.text(j, i, cm[i, j], ha="center", va="center")

fig.colorbar(im, ax=ax)
plt.show()

# Probability scores for curves (positive class = 'p')
# BernoulliNB supports predict_proba(), which we use for ROC and PR curves.
proba = nb_model.predict_proba(X_test)

# Identify the column index for class 'p'
p_index = list(nb_model.classes_).index("p")
y_score = proba[:, p_index]
```

Figure 65: Confusion Matrix code snapshot

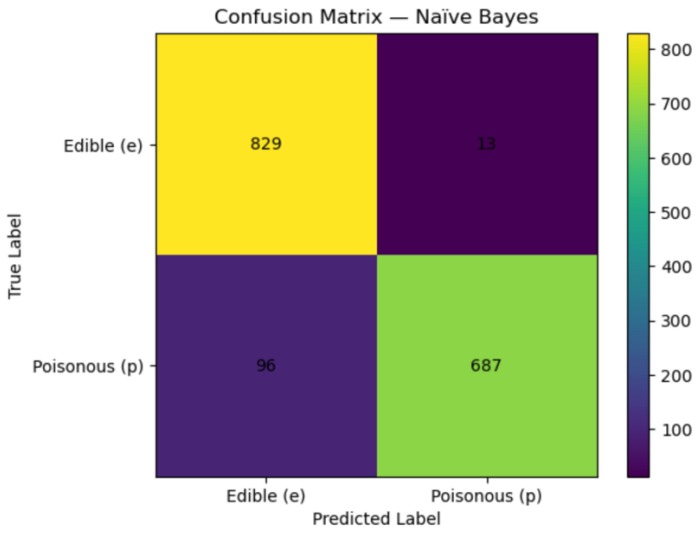

Figure 66: Evaluation of Naïve Bayes performance using Confusion Matrix Heat Map

Key Takeaways:

- The model is good at not accusing edible mushrooms to be poisonous as only 13 edible mushrooms were marked as poisonous.
- But it missed 96 poisonous mushrooms and predicted them as edible which is a serious safety risk.

```
#ROC Curve
fpr, tpr, _ = roc_curve(y_test, y_score, pos_label="p")
auc_score = roc_auc_score((y_test == "p").astype(int), y_score)

plt.figure()
plt.plot(fpr, tpr, label=f"AUC = {auc_score:.3f}")
plt.plot([0, 1], [0, 1], linestyle="--")
plt.title("ROC Curve – Naïve Bayes")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.legend()
plt.show()
```

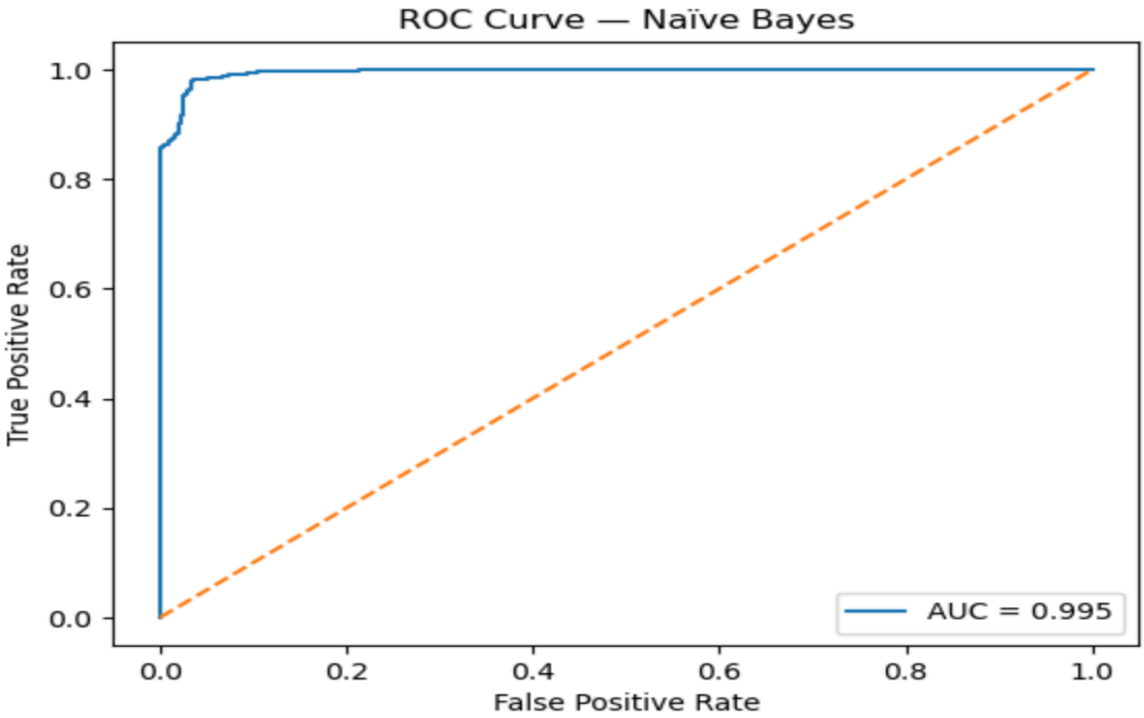

Figure 67: Evaluation of Naïve Bayes performance using ROC Curve.

Key Takeaways:

- Here, AUC = 0.995 which indicates great class separability. The model's score rank edible vs poisonous effectively across thresholds.

```
#Precision–Recall Curve
precision, recall, _ = precision_recall_curve(y_test, y_score, pos_label="p")
ap_score = average_precision_score((y_test == "p").astype(int), y_score)

plt.figure()
plt.plot(recall, precision, label=f"AP = {ap_score:.3f}")
plt.title("Precision–Recall Curve – Naïve Bayes")
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.legend()
plt.show()
```

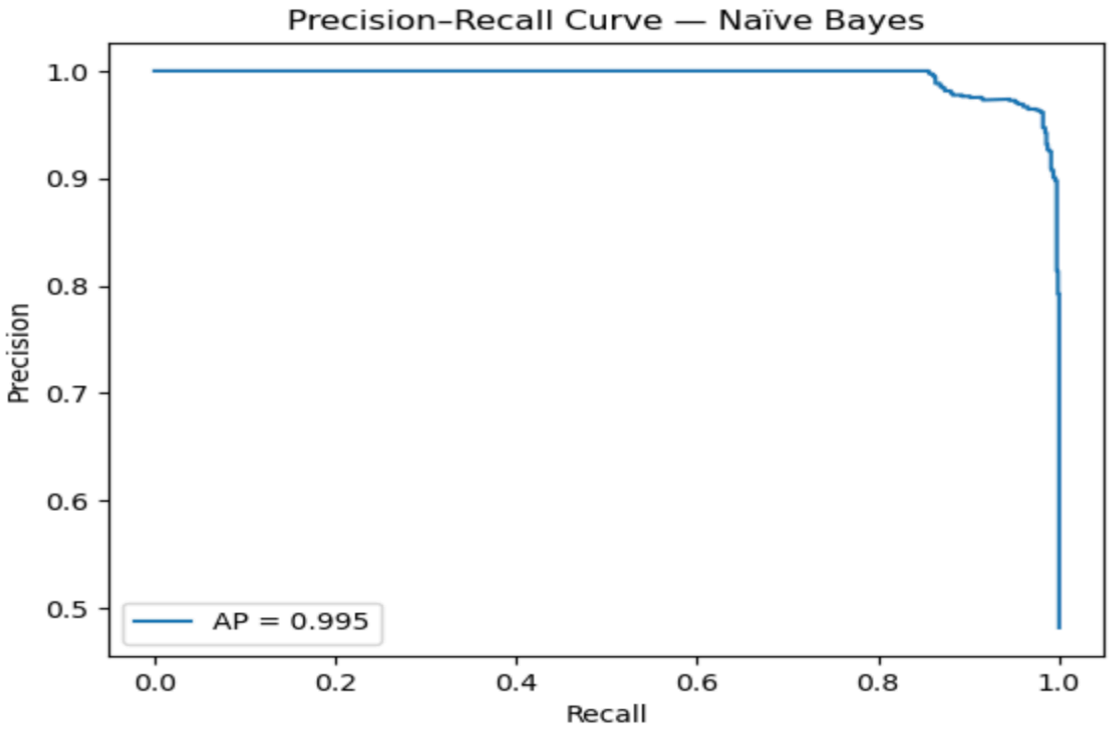

Figure 68: Evaluation of Naïve Bayes performance using Precision-Recall Curve.

Key Takeaways:

- The value of AP = 0.095 which means our model maintains high precision across most recall levels for the poisonous class, but the precision is low when trying to get identify every poisonous mushroom.

3.8.3 Random Forest

Random Forest is an ensemble machine learning procedure, which creates a variety of decision trees and integrates the results of these trees to provide a resulting prediction. A tree comes up with a set of decision rules by partitioning the data according to the feature values, and the forest gets the predictions (usually by a majority vote) to enhance reliability. Random Forest is also included, as it can deal with complex patterns and interactions between features, as the situation in mushrooms identification is expected to be (edibility is often a combination of factors).

Advantages:

- Can interact with categorical features well.
- Can often attain extremely high benchmark accuracy on this dataset.

Disadvantages:

- Random Forest is less interpretable than a single decision tree.
- Model behaviour may seem too good to be true on benchmark data and should be carefully examined.

Importing essential libraries

```
# Importing Libraries

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import (
    accuracy_score, confusion_matrix, classification_report,
    precision_score, recall_score, f1_score,
    roc_curve, roc_auc_score,
    precision_recall_curve, average_precision_score
)
```

Figure 69: Importing essential libraries for Random Forest.

Loading train/test datasets

The pre-processed dataset is loaded to ensure a consistent, reproducible and fair comparisons across models.

Methods used:

- read_csv()
- .squeeze()

Expected outcomes:

- `X_train/X_test` are loaded as feature matrices and `y_train/y_test` are loaded as 1D label vectors.

```
"""
Load the prepared train/test datasets created in DataPreparation.ipynb.

- pd.read_csv(): loads processed feature matrices and labels.
- squeeze(): converts a single-column label DataFrame into a 1D Series (preferred by scikit-learn).
"""
X_train = pd.read_csv("processed/X_train.csv")
X_test  = pd.read_csv("processed/X_test.csv")

y_train = pd.read_csv("processed/y_train.csv").squeeze()
y_test  = pd.read_csv("processed/y_test.csv").squeeze()

X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

```
((6499, 116), (1625, 116), (6499,), (1625,))
```

Figure 70: Loading train/test dataset for random forest

Training random forest classifier

The encoding of the mushroom features in the form of a random forest is then trained to learn the non-linear decision rules.

Methods used:

- RandomForestClassifier()
- .fit()

Expected outcomes:

- A trained random forest model capable of predicting edible vs poisonous mushrooms.

```
"""
Train a Random Forest classifier.

Key parameters:
- n_estimators: number of trees in the forest.
- random_state: ensures reproducible training.
- n_jobs=-1: uses all CPU cores (faster training).
"""
rf_model = RandomForestClassifier(
    n_estimators=200,
    random_state=42,
    n_jobs=-1
)
rf_model.fit(X_train, y_train)

rf_model
```

```
▾ RandomForestClassifier
RandomForestClassifier(n_estimators=200, n_jobs=-1, random_state=42)
```

Figure 71: Training Random Forest model using pre-prepared dataset.

Predicting on the test set

Predictions are made on unseen test data to assess generalisation. Scores are extracted to support ROC and Precision-Recall Evaluation.

Methods used:

- .predict(X_test)
- .predict_proba(X_test)

Expected outcome:

- 2 Arrays containing labels for e/p and probability of the mushrooms being e/p.

```
"""
Generate test predictions and probability scores.

We treat poisonous ('p') as the positive class.
"""
y_pred_rf = rf_model.predict(X_test)

proba_rf = rf_model.predict_proba(X_test)
p_index_rf = list(rf_model.classes_).index("p")
y_score_rf = proba_rf[:, p_index_rf]

y_pred_rf[:10], y_score_rf[:10]
```

```
(array(['p', 'p', 'e', 'p', 'p', 'e', 'e', 'e', 'e', 'p'], dtype=object),
 array([1., 1., 0., 1., 1., 0., 0., 0., 0., 1.]))
```

Figure 72: Generating random forest predictions and poisonous-class probability scores on test dataset

Model Evaluation

Overall correctness and class wise performance of the dataset is evaluated.

Methods used:

- accuracy_score()
- classification_report()

Expected outcomes:

- Accuracy score of the overall performance.
- Precision, recall, F1 values for e and p classes.

```
acc_rf = accuracy_score(y_test, y_pred_rf)
acc_rf
```

```
1.0
```

Figure 73: Random Forest Accuracy

```
print(classification_report(y_test, y_pred_rf, target_names=["Edible (e)", "Poisonous (p)"]))
```

```
              precision    recall  f1-score   support

   Edible (e)       1.00      1.00      1.00       842
Poisonous (p)       1.00      1.00      1.00       783

     accuracy                           1.00      1625
    macro avg       1.00      1.00      1.00      1625
 weighted avg       1.00      1.00      1.00      1625
```

Figure 74: Random Forest classification report on test dataset

Confusion Matrix

Summary of correct and incorrect prediction is shown in which the most harmful error is false negatives which means predicting poisonous mushrooms as edible.

Methods used:

- confusion_matrix()

Expected outcome:

- Confusion matrix showing TN, FP, FN, TP distribution.

```
"""
Confusion matrix (rows=actual, columns=predicted).

Most safety-critical error: poisonous predicted edible (FN).
"""
cm_rf = confusion_matrix(y_test, y_pred_rf, labels=["e", "p"])

cm_rf_df = pd.DataFrame(
    cm_rf,
    index=["Actual_e", "Actual_p"],
    columns=["Pred_e", "Pred_p"]
)
cm_rf_df
```

	Pred_e	Pred_p
Actual_e	842	0
Actual_p	0	783

Figure 75:Confusion Matrix summarising Random Forest Predictions

Evaluation metrics summary table

A compact summary table is created to compare Random Forest with Naïve Bayes and Logistic Regression. ‘p’ is treates as positive class.

Methods used:

- accuracy_score()
- precision_score(pos_label="p")
- recall_score(pos_label="p")
- f1_score(pos_label="p")
- roc_auc_score()
- average_precision_score()

Expected Outcomes:

- A table having summary for Accuracy, Precision, Recal;, F1, ROC-AUC, and AP

Table 3: Evaluation metrics summary table for Random Forest

```
"""
Metrics table (poisonous 'p' as positive class).
"""
roc_auc_rf = roc_auc_score((y_test == "p").astype(int), y_score_rf)

precision_rf = precision_score(y_test, y_pred_rf, pos_label="p")
recall_rf    = recall_score(y_test, y_pred_rf, pos_label="p")
f1_rf        = f1_score(y_test, y_pred_rf, pos_label="p")

ap_rf = average_precision_score((y_test == "p").astype(int), y_score_rf)

metrics_rf = pd.DataFrame({
    "METRIC": ["ACCURACY", "PRECISION (p)", "RECALL (p)", "F1-SCORE (p)", "ROC-AUC", "AP"],
    "VALUE":  [acc_rf, precision_rf, recall_rf, f1_rf, roc_auc_rf, ap_rf]
})
metrics_rf["VALUE"] = metrics_rf["VALUE"].round(4)
metrics_rf
```

	METRIC	VALUE
0	ACCURACY	1.0
1	PRECISION (p)	1.0
2	RECALL (p)	1.0
3	F1-SCORE (p)	1.0
4	ROC-AUC	1.0
5	AP	1.0

Evaluation plots for Confusion Matrix, ROC, and Precision-Recall

These plots provide interpretation beyond raw metrics and show misclassification patterns and threshold-based performance.

Methods used:

- confusion_matrix()
- roc_curve() + roc_auc_score()
- precision_recall_curve() + average_precision_score()

Expected outcomes:

- Confusion matrix heatmap which shows where errors occur.
- ROC Curve which shows separability across thresholds.
- Precision-Recall curve that focuses on poisonous-class detection quality.

```
# Confusion Matrix Heatmap
fig, ax = plt.subplots()
im = ax.imshow(cm_rf, interpolation="nearest")
ax.set_title("Confusion Matrix — Random Forest")
ax.set_xticks([0, 1])
ax.set_yticks([0, 1])
ax.set_xticklabels(["Edible (e)", "Poisonous (p)"])
ax.set_yticklabels(["Edible (e)", "Poisonous (p)"])
ax.set_xlabel("Predicted Label")
ax.set_ylabel("True Label")

for i in range(cm_rf.shape[0]):
    for j in range(cm_rf.shape[1]):
        ax.text(j, i, cm_rf[i, j], ha="center", va="center")
fig.colorbar(im, ax=ax)
plt.show()
```

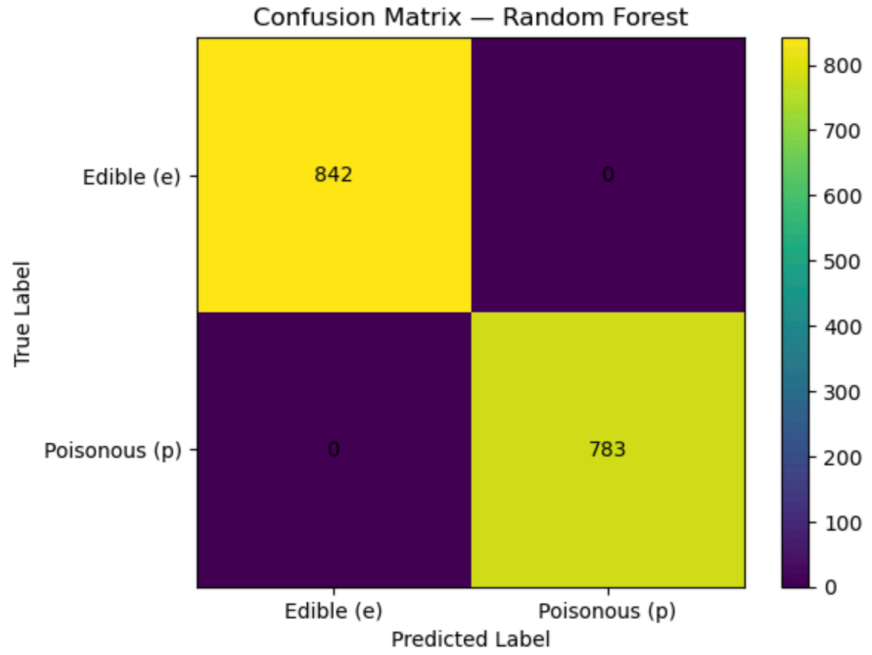

Figure 76: Confusion Matrix Heatmap to visualise Random Forest Misclassifications

Key Takeaways:

- Heatmap shows that mushrooms are correctly classified and there are no false negatives.

```
#ROC Curve

fpr_rf, tpr_rf, _ = roc_curve(y_test, y_score_rf, pos_label="p")

plt.figure()
plt.plot(fpr_rf, tpr_rf, label=f"AUC = {roc_auc_rf:.3f}")
plt.plot([0, 1], [0, 1], linestyle="--")
plt.title("ROC Curve – Random Forest")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.legend()
plt.show()
```

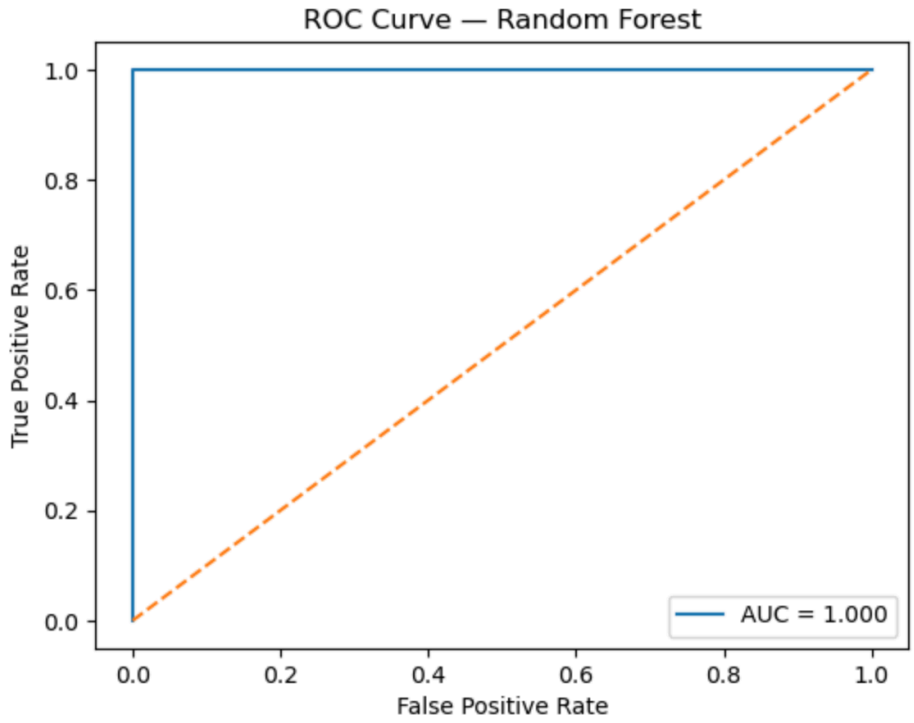

Figure 77:ROC Curve showing Random Forest class separability.

Key Takeaways:

- The value of AUC is 1 which means there is strong precision while achieving high recall for detecting poisonous mushrooms.

```
#Precision–Recall Curve

prec_curve_rf, rec_curve_rf, _ = precision_recall_curve(y_test, y_score_rf, pos_label="p")

plt.figure()
plt.plot(rec_curve_rf, prec_curve_rf, label=f"AP = {ap_rf:.3f}")
plt.title("Precision–Recall Curve – Random Forest")
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.legend()
plt.show()
```

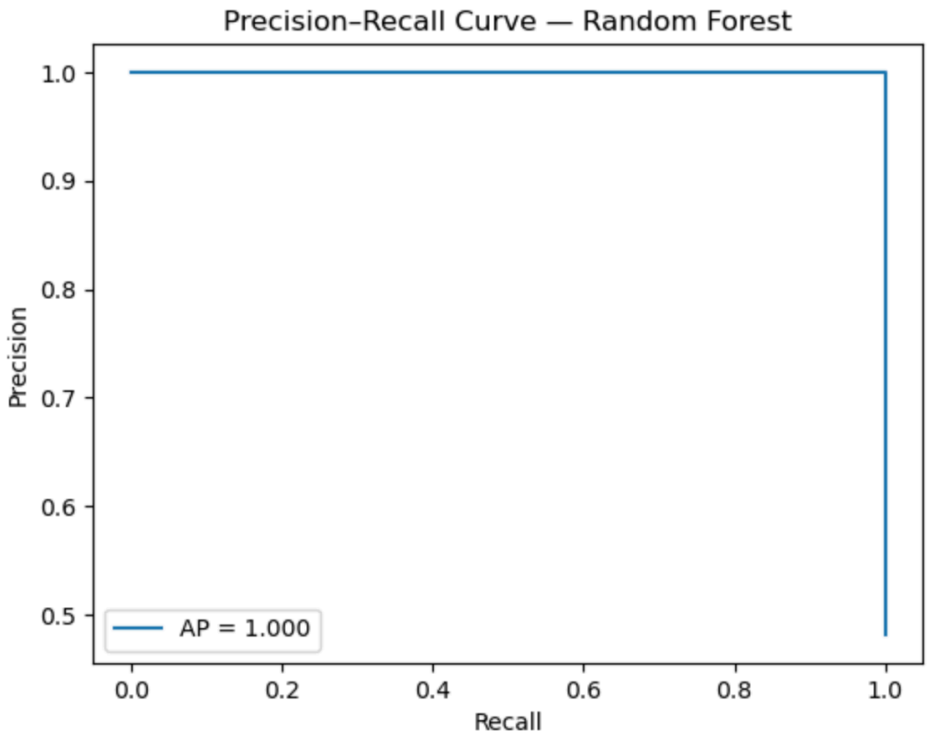

Figure 78: Precision-Recall curve showing poisonous-class performance of Random Forest.

Key Takeaways:

- AP = 1 which means Random Forest achieves near-perfect poisonous-class detection and keeps very high precision at high recall.

4. Conclusion

4.1 Analysis of the Work

In the given work, mushroom edibility prediction was a supervised binary classification task that was carried out with a complete, reproducible pipeline that was developed in various Python notebook files. It started with Data Exploration, in which the structure of the data (number of rows/ columns, type of categorical features, and the name of targets e/p) was explored to verify it was good enough to perform supervised learning. This step has helped to make sure the data is entirely categoric and by contrasting to a single attribute, that edibility is determined by a combination of interacting attributes, which justifies safety-first modelling.

Thereupon, Data Preparation constructed a clean and homogenous modelling dataset. The following steps during the workflow were followed: work copy, duplicate detection, missing/unknown feature check, non-informative constant feature removal, one-hot encoding to transform categorical features to numerical binary features, and train /test set division with a fixed random seed and stratification to maintain the balance of classes. The ready-made dataset has been stored in such a way that every model is trained and evaluated based on the same division.

Three algorithms, which included Naive Bayes, Logistic Regression, and Random Forest were then trained and tested. Naive Bayes produced high levels of separability (high ROC -AUC) but generated more false negatives (poisonous predicted as edible) which is the safest error in this problem. Logistic Regression achieved high classification accuracy with almost zero misclassification as well as minimal error hence it proves that a linear boundary of decision may already distinguish the space of one-hot encoded features. The best and most consistent overall performance was presented by Random Forest, where threshold behaviour (ROC and Precision Recall curves) is close to perfection, and safety-critical errors are minimised, by considering non-linear interactions between features. Random Forest is chosen as the most suitable Netflix model among the three of them to be deployed to the final due to its accuracy, AUC/AP behaviour and the level of risk shown in the confusion-matrix, with a significant reference to Logistic Regression.

4.2 How the Solution Helps to Address Real-World Problems

Mushroom identification is safety-critical in the real world, since errors may prove disastrous, and very often identifications are made by non-experts in the state of uncertainty. In the suggested AI-based solution, safer decision-making is facilitated through learning trends on a range of observable characteristics and implementing such reasoning continuously, instead of utilizing trial-of-thumb opinion only. It can best be considered decision support, and the comparative design of the report can guarantee that the ultimate recommendation is not based on assumptions but is evidence based.

The main approaches the solution may contribute to solve the real-life problem:

Consistency: The logic of feature-combination is used consistently, minimizing variations.

Multi-trait thinking: The model does not rely on a single (rule) but rather a combination of evaluations, as is the case with the determination of edibility.

Evidence-based Model Choice: The Logistic Regression, Naïve Bayes and the Random Forest will be compared and assessed under an equal preprocessing and train/test split assuring that the comparison remains fair.

Safety-oriented Assessment: The inspection cannot be limited to overall performance but to identify high-risk errors (such as unsafe mushrooms as to be consumed).

Conservative Meaning of the Unsafe: Since the dataset is extended to include the unknown/not recommended category together with the poisonous category, the classification is consistent with a conservative do not eat unless you are sure attitude.

4.3 Further Work

• Once the project has been developed, the system might be tested on new mushroom data to determine whether it can generalise beyond the UCI benchmark data.

• The models could be generalized to deal with real world uncertainty, including missing feature information, noisy inputs, or missing attribute values as ideal identification is hardly ever the case in practice.

• More sophisticated models might be included in the comparison to determine whether they might provide any benefits over the three models applied in this project.

• An actual system that would allow users to input their mushroom attributes and give feedback in a clear manner as it is decision support would be handy.

• Lastly, the usability can be enhanced by displaying a confidence indicator and a warning-oriented design that can minimize the interpretation of predictions in an unsafe manner.

5. References

Bibliography

GeeksForGeeks, 2025. *Machine Learning.* [Online] Available at: https://www.geeksforgeeks.org/machine-learning/understanding-logistic-regression/ [Accessed 11 December 2025].

Nepal Journals Online, 2022. *Article.* [Online] Available at: https://www.nepjol.info/index.php/njmathsci/article/view/44130 [Accessed 11 December 2025].

ResearchGate, 2025. *Figure.* [Online] Available at: https://www.researchgate.net/figure/Similar-looking-mushrooms-are-hard-to-differentiate-a-A-picture-of-Chlorophyllum_fig1_382904445 [Accessed 10 December 2025].

ResearchGate, 2025. *Publication.* [Online] Available at: https://www.researchgate.net/publication/393698893_Analysis_of_the_Effectiveness_of_Traditional_and_Ensemble_Machine_Learning_Models_for_Mushroom_Classification [Accessed 12 December 2025].

UCI ML Repository, 1987. *dataset.* [Online] Available at: https://archive.ics.uci.edu/dataset/73/mushroom [Accessed 11 December 2025].

UCI, 2025. *Home.* [Online] Available at: https://archive.ics.uci.edu/ [Accessed 10 December 2025].

Verma, A., 2025. *Solutions.* [Online] Available at: https://medium.com/@amitvsolutions/supervised-machine-learning-decoded-from-forecasts-to-decisions-4e57c6b1a0c4 [Accessed 11 December 2025].

Verma, A., 2025. *Solutions.* [Online] Available at: https://medium.com/@amitvsolutions/supervised-machine-learning-decoded-from-forecasts-to-decisions-4e57c6b1a0c4
[Accessed 11 December 2025].