Project-Based Data Analytics for Mushroom Cultivat

Training Module: Project-Based Data Analytics for Mushroom Cultivation

🌱 Module Introduction

This training is a **3-day project-based programme** designed to equip learners with practical skills in **data analytics** using both **general datasets** and a **real-world mushroom cultivation case study**.

The training emphasizes **hands-on practice**, guiding participants through the entire data analytics pipeline:

- Collecting and preparing data
- Visualizing and exploring data
- Building predictive models
- Applying prescriptive analytics (decision-making & optimization)
- Developing interactive dashboards
- Integrating image analytics for disease/quality detection

By the end of the programme, participants will deliver a **team-based project** that integrates **sensor data** (temperature, humidity, CO_2, voltage, timestamp) with **image data**, producing a functional mushroom analytics dashboard.

4MB

Exploratory Data Analysis_Kolej Komuniti.pdf
pdf

🎯 Learning Outcomes

Upon completing this module, participants will be able to:

1. Explain the principles of **descriptive, predictive, and prescriptive analytics**.
2. Apply **Python libraries** (Pandas, Numpy, Seaborn, and Scikit-learn) for data analysis.
3. Build **predictive models** for classification and regression tasks.
4. Design **prescriptive analytics solutions** using optimisation techniques.
5. Develop an **interactive dashboard** using Streamlit/Dash.
6. Integrate **sensor data** with **image analytics** to solve agricultural problems.
7. Deliver a **project presentation** that demonstrates an end-to-end analytics system.

Training Schedule

Day 1 – Foundations of Data Analytics (General Datasets)

- Introduction to analytics (descriptive → predictive → prescriptive)
- Python refresher (Pandas, Numpy, visualization)
- Predictive modeling (classification & regression)
- From prediction to prescription (optimization basics)
- Lab: Titanic / Iris dataset analysis

Day 2 – Mushroom Analytics (Domain Application)

- Mushroom dataset exploration (UCI + Kaggle)
- End-to-end analytics pipeline (sensor-style data)
- Dashboard development (Streamlit/Dash)
- Project development sessions (team-based)
- Lab: Predict mushroom edibility & growth conditions

Day 3 – Integration & Presentation

- Image analytics (mushroom disease/quality detection)
- Final project integration (sensors + images)
- Team presentations & demo
- Reflection, feedback, certification

🛠️ Tools and Technologies

- **Python Libraries**: Pandas, Numpy, Matplotlib, Seaborn, Scikit-learn
- **Dashboards**: Streamlit, Dash
- **Machine Learning**: Random Forest, Logistic Regression, Regression Models
- **Optimization**: PuLP, Scipy.optimize
- **Deep Learning for Images**: TensorFlow, Keras
- **Datasets**:
  - General: Iris, Titanic, Boston Housing
  - Mushroom: UCI Mushroom Dataset, Kaggle Mushroom Growth, Kaggle Mushroom Images

---

📦 Deliverables

Each participant/team will produce:

1. **Jupyter Notebooks** with data preprocessing, modeling, and prescriptions.
2. **Interactive Dashboard** for monitoring and decision support.
3. **Image Classification Component** (disease/quality detection).
4. **Final Project Presentation** demonstrating the system.

📚 Module Structure

The training module is organized into three main parts:

1. **Foundations of Analytics (Day 1)**
    - Topic 1: Introduction to Analytics
    - Topic 2: Python Refresher
    - Topic 3: Predictive Modeling
    - Topic 4: From Prediction to Prescription
2. **Mushroom Analytics Application (Day 2)**
    - Topic 5: Mushroom Dataset Exploration
    - Topic 6: Building the Analytics Pipeline
    - Topic 7: Dashboard Development
    - Topic 8: Project Development Sessions
3. **Integration & Presentation (Day 3)**
    - Topic 9: Image Analytics for Mushroom Quality
    - Topic 10: Final Project Integration
    - Topic 11: Project Presentations & Reflection

Foundations of Analytics

Introduction

This module lays the **foundation of analytics** by equipping participants with the essential concepts, tools, and practices required for data-driven decision-making. We begin with a broad overview of analytics, progress into a Python refresher, and then explore predictive modeling before linking it to prescriptive analytics.

To make the learning smooth, participants will first work with **general datasets** (e.g., Iris, Titanic, Boston Housing) so they can focus on techniques without domain-specific complexities. Later modules will apply these skills to **mushroom datasets**.

Learning Objectives

By the end of this module, participants will be able to:

- Differentiate between **descriptive, predictive, and prescriptive analytics**.
- Use **Python libraries** (Pandas, Numpy, Matplotlib, and Seaborn) for data preparation and visualisation.
- Apply **predictive models** (classification and regression) using Scikit-learn.
- Understand how predictions lead to **prescriptive actions** through optimisation.
- Work collaboratively in teams to discuss how analytics applies to real-world problems.

Session Flow

Morning Sessions

- **Topic 1: Introduction to Analytics**
  - Analytics cycle: descriptive → predictive → prescriptive.
  - Use cases in business, healthcare, and agriculture.
- **Topic 2: Python Refresher**
  - DataFrames with Pandas.
  - Numerical operations with Numpy.
  - Visualisations with Matplotlib and Seaborn.

Afternoon Sessions

- **Topic 3: Predictive Modeling**
  - Classification and regression basics.
  - Model evaluation (accuracy, ROC AUC, RMSE).
  - Lab: Titanic survival prediction.
- **Topic 4: From Prediction to Prescription**
  - Optimisation concepts.
  - Decision rules for prescriptive analytics.
  - Lab: Resource allocation with `scipy.optimize`.

Hands-On Labs

This module is highly practical. Example labs include:

- Loading the **Iris dataset** and performing basic exploration.
- Visualising the Titanic dataset survival rates.
- Building a **Random Forest model** to classify passenger survival.
- Applying **linear regression** to predict house prices (Boston Housing dataset).
- Writing a simple **optimisation problem** (e.g., maximise farm yield given limited water resources).

Tools Used

- **Pandas** → Data cleaning & manipulation.
- **Numpy** → Numerical computations.
- **Matplotlib & Seaborn** → Visualisation.
- **Scikit-learn** → Machine learning models.
- **Scipy.optimize** → Prescriptive analytics (optimisation).

Expected Outcomes

At the end of this module, participants will:

- Have a clear understanding of the **analytics workflow**.
- Be able to explore and visualise datasets independently.
- Build and evaluate their **first machine learning models**.
- Translate model results into **basic prescriptive recommendations**.

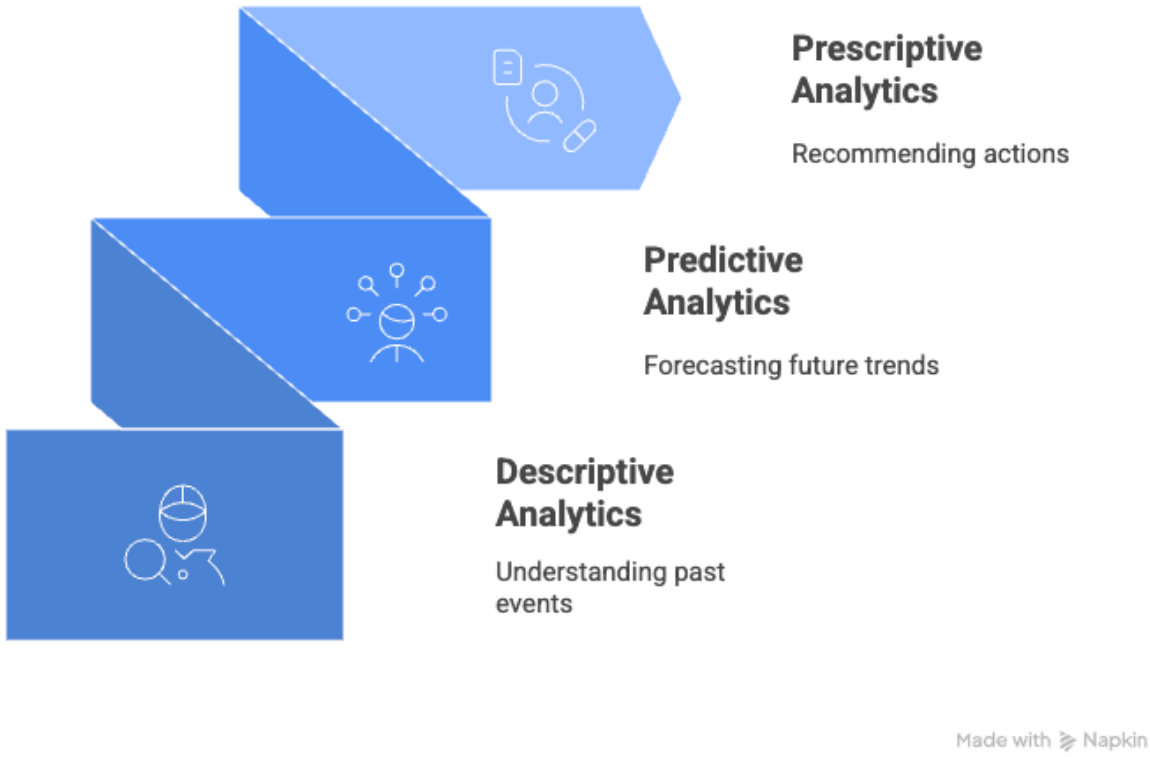
Topic 1: Introduction to Analytics

1.1 Overview

Data analytics is the practice of using data to generate insights and support decision-making. In mushroom cultivation, analytics helps farmers adjust conditions like temperature, CO_2, and humidity to improve yield, reduce contamination, and optimize resource usage.

Analytics progresses through three levels:

1. **Descriptive Analytics** – *What happened?*
2. **Predictive Analytics** – *What is likely to happen?*
3. **Prescriptive Analytics** – *What should we do about it?*



This topic introduces these levels, the analytics cycle, and practical exercises to prepare you for mushroom-focused applications later in the course.

1.2 The Data Analytics Cycle

Analytics is best seen as a cycle, not a one-time process.

Steps in the cycle:

- **Data Collection** → Gather measurements (e.g., timestamp, temperature, humidity).
- **Data Preparation** → Clean, normalise, and transform into usable form.
- **Exploration & Visualisation** → Identify patterns and anomalies.
- **Modeling** → Use machine learning to predict outcomes.
- **Prescription** → Recommend actions based on model insights.
- **Evaluation & Feedback** → Measure impact and refine.

💡 *Example in Mushroom Analytics:*

- Sensors collect **temperature** and **CO_2** data.
- A model predicts the likelihood of a yield drop.
- The system prescribes, "*Increase humidity by 5%.*"

1.3 Types of Analytics

1. Descriptive Analytics – *What happened?*

- Summarises past data.
- Techniques: averages, counts, plots.
- Example: *"Average humidity last week was 70%."*

2. Predictive Analytics – *What will happen?*

- Uses machine learning to forecast outcomes.
- Example: *"If temperature rises above 28°C, CO_2 levels will likely exceed safe thresholds."*

3. Prescriptive Analytics – *What should we do?*

- Suggests optimal actions.
- Example: *"If humidity < 60%, turn on misting system for 15 minutes."*

1.4 Analytics in Agriculture and Mushroom Cultivation

Mushroom cultivation is highly sensitive to environmental conditions. The key variables are:

- **Timestamp** → tracks when conditions were recorded.
- **Temperature** → affects fungal metabolism.
- **CO_2 Levels** → indicator of air quality and ventilation.
- **Humidity** → critical for fruiting body development.
- **Voltage** → tracks equipment operation and efficiency.
- **Image Data** → enables disease/quality detection.

These datasets provide a **rich testbed** for learning how to apply analytics in practice.

1.5 Tools for Analytics

We will use Python's data ecosystem:

- **Pandas** – Tabular data handling.
- **Numpy** – Numerical computations.
- **Matplotlib & Seaborn** – Visualization.
- **Scikit-learn** – Machine learning.
- **Streamlit/Dash** – Dashboard creation.
- **TensorFlow/Keras** – Image analytics.
- **PuLP / Scipy.optimize** – Optimization for prescriptions.

1.6 Hands-On Lab: First Steps in Data Analytics

Exercise 1: Load and Explore Data

```
import pandas as pd

# Load sample dataset (Iris for Day 1)
from sklearn.datasets import load_iris
iris = load_iris(as_frame=True)
df = iris.frame

# View first rows
print(df.head())

# Quick summary
print(df.describe())
```

Explanation

- `load_iris(as_frame=True)` loads the **Iris dataset** into a Pandas DataFrame.
- `df.head()` shows the first 5 rows (quick inspection).
- `df.describe()` computes averages, min, max, and standard deviations.

💡 In mushroom analytics, this will later be replaced with **temperature, humidity, CO_2, and voltage readings**.

Exercise 2: Visualize Patterns

```
import seaborn as sns
import matplotlib.pyplot as plt

sns.scatterplot(x="sepal length (cm)",
                y="sepal width (cm)",
                hue="target",
                data=df)

plt.title("Iris Dataset Visualization")
plt.show()
```

Explanation

- `sns.scatterplot(...)` creates a scatter plot.
  - `x` = sepal length, `y` = sepal width.
  - `hue="target"` colours points by species.
- `plt.show()` displays the chart.

💡 Later, we can replace **sepal length** with **temperature** and **sepal width** with **CO_2 levels** to visualize mushroom growth conditions.

Exercise 3: Build a Simple Prediction

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Split data into features (X) and labels (y)
X = df.drop(columns="target")
y = df["target"]

# Train-test split (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Build and train model
model = RandomForestClassifier()
model.fit(X_train, y_train)

# Predict on test data
y_pred = model.predict(X_test)

# Evaluate accuracy
print("Accuracy:", accuracy_score(y_test, y_pred))
```

Explanation

- **Step 1:** `train_test_split(...)` splits data into training (80%) and testing (20%).
- **Step 2:** `RandomForestClassifier()` builds a model using multiple decision trees.
- **Step 3:** `model.fit(...)` trains the model.
- **Step 4:** `model.predict(...)` makes predictions on unseen test data.
- **Step 5:** `accuracy_score(...)` checks how accurate the model is.

💡 With mushroom data, this same workflow predicts whether a mushroom is **edible or poisonous or** whether **current conditions will result in good growth**.

1.7 Summary

- Analytics can be **descriptive, predictive, or prescriptive**.
- The **data analytics cycle** is iterative, feeding back into itself.
- Python's ecosystem (Pandas, Seaborn, and Scikit-learn) provides powerful tools.
- Through small exercises, we practised loading, visualising, and modeling data.
- These same techniques will later be applied to **mushroom datasets**.

1.8 Self-Assessment Questions

1. Explain the difference between **predictive** and **prescriptive** analytics.
2. Why is visualisation important before model building?
3. What is the role of the `train_test_split` function in machine learning?
4. How would you use image data in mushroom cultivation analytics?

<details>
<summary>Q1. Explain the difference between predictive and prescriptive analytics.</summary>

Answer:

- **Predictive analytics** uses historical data and machine learning models to **forecast future outcomes**. It answers the question: *"What is likely to happen?"*
  - Example: Predicting mushroom yield based on temperature and humidity data.
- **Prescriptive analytics** goes a step further by recommending **actions to optimise outcomes**. It answers, "*What should we do?*"
  - Example: If humidity is predicted to drop below 60%, prescribe turning on the misting system for 15 minutes to prevent yield loss.

👉 Predictive = forecast, Prescriptive = action.
</details>

Q2. Why is visualisation important before model building?

Answer:
Visualization is essential because it:

- Helps **understand data distribution** (e.g., normal vs skewed).
- Reveals **patterns, clusters, or relationships** between features (e.g., high CO_2 correlating with lower yield).
- Identifies **outliers or anomalies** that may distort the model.
- Provides **intuitive insights** that guide feature engineering and model selection.

👉 Example: A scatter plot of **temperature vs CO_2** can show whether these two variables cluster into distinct growth conditions before any predictive model is applied

Q3. What is the role of the `train_test_split` function in machine learning?

Answer:
The `train_test_split` function divides the dataset into:

- **Training set** → used to build (fit) the model.
- **Test set** → used to evaluate the model's performance on unseen data.

This prevents **overfitting**, where a model performs well on training data but poorly on new data. By holding out a portion of the dataset for testing, we ensure the model can **generalize** to real-world scenarios.

👉 In mushroom analytics, we might use 80% of sensor readings to train a model and 20% to test whether it can predict new conditions accurately.

Q4. How would you use image data in mushroom cultivation analytics?

Answer:
Image data adds a **visual dimension** to sensor-based analytics. It can be used for:

- **Disease detection** → Identify fungal infections or contamination on mushroom caps.
- **Growth stage monitoring** → Classify mushrooms as *spawning, pinning, or fruiting*.
- **Quality grading** → Detect deformities, discoloration, or size irregularities.

This is typically done using **Convolutional Neural Networks (CNNs)** or other computer vision techniques. When combined with sensor data (temperature, humidity, CO_2), image data enables **richer, more accurate prescriptions**.

👉 Example: If CO_2 levels are high **and** images show signs of stunted growth, the system could prescribe better ventilation and alert the farmer.

1.9 Further Reading

- Provost, F., & Fawcett, T. (2013). *Data Science for Business*. O'Reilly.
- UCI Machine Learning Repository: Mushroom Dataset
- Kaggle: Mushroom Classification

Topic 2: Python Refresher

2.1 Overview

Python is the most widely used language in data analytics because of its simplicity, flexibility, and powerful ecosystem of libraries. Before we dive into predictive and prescriptive analytics, it's important to revisit the **core Python tools** that will form the backbone of our work:

- **Pandas** → Handling tabular data (like Excel).
- **Numpy** → Fast numerical operations.
- **Matplotlib & Seaborn** → Data visualisation.

This refresher assumes participants already have some experience with Python but may need to reinforce **practical data analysis skills**.

For assistance with Python setup and revisiting fundamental concepts, please refer to the lessons titled "Setting up Python" and "Python Basics".

Setting Up Python

Python Basics

2.2 Learning Objectives

By the end of this topic, participants will be able to:

- Load and manipulate datasets using **Pandas**.
- Perform numerical calculations with **Numpy**.
- Create effective **visualisations** using Matplotlib and Seaborn.
- Understand how these tools work together in a typical data workflow.

2.3 Working with Pandas

Loading Data

```
import pandas as pd

# Load a CSV file
df = pd.read_csv("titanic.csv")

# Show first 5 rows
print(df.head())
```

🔎 Explanation

- `pd.read_csv("titanic.csv")` → loads a dataset from a CSV file into a **DataFrame**, the core data structure in Pandas.
- `df.head()` → shows the first 5 rows for inspection.

👉 Later, this will be applied to **mushroom datasets** (e.g., temperature, humidity, CO_2 readings).

Selecting Data

```
# Select a single column
ages = df["Age"]

# Select multiple columns
subset = df[["Age", "Fare"]]

# Filter rows
adults = df[df["Age"] > 18]
```

🔎 Explanation

- `df["Age"]` → selects a single column.
- `df[["Age", "Fare"]]` → selects multiple columns.
- `df[df["Age"] > 18]` → filters rows where Age > 18.

👉 In mushroom analytics, this could filter for rows where **humidity > 70%**.

Grouping and Summarizing

```
# Average fare by passenger class
avg_fare = df.groupby("Pclass")["Fare"].mean()
print(avg_fare)
```

🔎 Explanation

- `groupby("Pclass")` → groups rows by passenger class.
- `["Fare"].mean()` → calculates the mean fare for each group.

👉 In mushrooms, this could compute **average yield per temperature range**.

2.4 Working with Numpy

```
import numpy as np

# Create an array
arr = np.array([1, 2, 3, 4, 5])

# Basic statistics
print("Mean:", np.mean(arr))
print("Standard Deviation:", np.std(arr))
```

🔎 Explanation

- `np.array([...])` → creates a Numpy array (fast and efficient for math operations).
- `np.mean()` and `np.std()` → calculate average and spread of values.

👉 In mushroom cultivation, this could calculate the **mean CO_2 level** or **variation in humidity**.

2.5 Visualization with Matplotlib & Seaborn

Matplotlib Example

```
import matplotlib.pyplot as plt

# Histogram of passenger ages
plt.hist(df["Age"].dropna(), bins=20, color="skyblue")
plt.title("Age Distribution")
plt.xlabel("Age")
plt.ylabel("Count")
plt.show()
```

🔎 Explanation

- `plt.hist(...)` → creates a histogram.
- `dropna()` removes missing values.
- `bins=20` → number of bars in the histogram.

👉 Later, we can plot **temperature distribution** from mushroom sensor data.

Seaborn Example

```
import seaborn as sns

# Boxplot of fare by passenger class
sns.boxplot(x="Pclass", y="Fare", data=df)
plt.title("Fare by Passenger Class")
plt.show()
```

🔎 **Explanation**

- `sns.boxplot(...)` → shows distribution of fares across passenger classes.
- Boxplots highlight **medians, quartiles, and outliers**.

👉 In mushroom analytics, this could show **humidity levels grouped by growth stage**.

2.6 Integrated Example: Quick EDA

```
# Summary statistics
print(df.describe())

# Correlation heatmap
sns.heatmap(df.corr(), annot=True, cmap="coolwarm")
plt.title("Correlation Heatmap")
plt.show()
```

🔎 **Explanation**

- `df.describe()` → statistical summary (mean, min, max, std).
- `df.corr()` → computes correlation between numeric columns.
- `sns.heatmap(...)` → visualises correlation as a colour-coded matrix.

👉 With mushroom data, this would reveal **how temperature, humidity, and CO_2 are correlated**.

2.7 Summary

In this topic, we reviewed:

- **Pandas** for loading, selecting, filtering, and summarising data.
- **Numpy** for fast numerical operations.
- **Matplotlib & Seaborn** for creating informative plots.
- How these tools combine to perform quick **exploratory data analysis (EDA)**.

2.8 Self-Assessment Questions

1. What is the difference between a Pandas DataFrame and a Numpy array?
2. How would you filter rows in a DataFrame where humidity is greater than 80%?
3. Why are boxplots useful in visualising mushroom growth conditions?
4. What does it `df.corr()` reveal about a dataset?

Q1. What is the difference between a Pandas DataFrame and a Numpy array?

Answer:

- A **Pandas DataFrame** is a **tabular structure** with labeled rows and columns. It is ideal for working with datasets similar to Excel spreadsheets, where data can have mixed types (e.g., numbers, text, dates).
- A **Numpy array** is a **homogeneous numerical structure** optimized for mathematical operations. It is much faster for numerical computations but lacks labels for rows and columns.

👉 Example:

- Use **Numpy arrays** for vectorized calculations (e.g., mean CO_2 levels).
- Use **Pandas DataFrames** for real-world datasets (e.g., mushroom growth logs with timestamp, temperature, humidity).

Q2. How would you filter rows in a DataFrame where humidity is greater than 80%?

Answer:

```
# Assume df has a 'Humidity' column
high_humidity = df[df["Humidity"] > 80]
print(high_humidity.head())
```

- `df["Humidity"] > 80` → creates a Boolean condition (True/False for each row).
- `df[...]` → selects rows where the condition is True.

👉 In mushroom cultivation, this could isolate data points from **overly humid conditions**, which may increase the risk of contamination.

Q3. Why are boxplots useful in visualising mushroom growth conditions?

Answer:

- Boxplots show **distribution and spread** of a variable, including:
  - Median (central value).
  - Interquartile range (spread of the middle 50%).
  - Outliers (unusual data points).
- This makes them excellent for detecting:
  - **Abnormal sensor readings** (e.g., sudden CO_2 spikes).
  - **Variability** in growth conditions across different batches.

👉 Example: A boxplot of **humidity per growing chamber** would reveal which chambers are unstable or prone to extreme values.

Q4. What does it `df.corr()` reveal about a dataset?

Answer:

- `df.corr()` calculates the **correlation coefficients** (ranging from -1 to +1) between all numerical columns in a dataset.
- A **positive correlation** (+1) means two variables increase together.
- A **negative correlation** (-1) means one variable increases while the other decreases.
- A value close to **0** means no linear relationship.

👉 Example:

- If mushroom data shows a **positive correlation** between **humidity and yield**, it means higher humidity generally improves growth.
- If there is a **negative correlation** between **CO_2 and yield**, it indicates high CO_2 reduces production.

Setting Up Python

Setting Up Python

Before diving into programming, it is important to have Python installed on your system.

1. **Download Python**
   - Visit the official Python website: https://www.python.org/downloads/ ↗
   - Choose the latest stable release (Python 3.x).
   - During installation, ensure that the option **"Add Python to PATH"** is checked.
2. **Verify Installation**
   - Open a terminal (Command Prompt or PowerShell on Windows, Terminal on macOS/Linux).
   - Type:

     ```
     python --version
     ```

   - You should see something like `Python 3.11.6` (version may vary).
3. **Package Management with pip**
   - Python uses `pip` to install and manage packages.
   - Example:

     ```
     pip install pandas
     ```

   - This installs the `pandas` library, which is essential for data analysis.

Setting Up JetBrains DataSpell

DataSpell is a specialised IDE from JetBrains designed for data scientists. It combines code editing, Jupyter Notebook support, and data visualisation features in one environment.

Steps to Set Up:

1. Download DataSpell: https://www.jetbrains.com/dataspell/ ↗
2. Install and activate using a JetBrains account (free academic licences are available for students and educators).
3. Configure Python interpreter:
    - Go to **File → Settings → Project Interpreter**.
    - Add your Python installation or create a virtual environment.
4. Install recommended plugins (optional):
    - Jupyter Support
    - Markdown Navigator
    - CSV Plugin for quick dataset previews

Why Use DataSpell?

- Integrated environment for Python coding and notebooks.
- Strong debugging and visualisation tools.
- Easy management of virtual environments and dependencies.

Setting Up Jupyter Notebook

Jupyter Notebook is one of the most popular tools for data analytics, combining code, visualisations, and explanations in one interactive document.

Steps to Install Jupyter Notebook:

1. Install using pip:

```
pip install notebook
```

2. Start Jupyter Notebook:

```
jupyter notebook
```

3. This will open a local web interface at `http://localhost:8888/`.
4. Create a new notebook and select **the Python 3 kernel**.

Key Features:

- Code and narrative text (Markdown) in one place.
- Interactive visualisations (with Matplotlib, Seaborn, Plotly, etc.).
- Widely used in research and industry.

Python Basics

Python Basics Refresher

Variables and Data Types

```
x = 10        # integer
y = 3.14      # float
name = "Mushroom"  # string
is_edible = True   # boolean
```

Lists and Dictionaries

```
# List
mushrooms = ["Shiitake", "Oyster", "Button"]

# Dictionary
cultivation = {
    "Shiitake": "Wood logs",
    "Oyster": "Straw",
    "Button": "Compost"
}
```

Control Structures

```
for mushroom in mushrooms:
    print(mushroom)

if is_edible:
    print("Safe to eat")
else:
    print("Not safe to eat")
```

Functions

```
def yield_estimate(area, productivity):
    return area * productivity

print(yield_estimate(10, 5))  # Output: 50
```

Libraries for Analytics

- **pandas** → Data manipulation
- **numpy** → Numerical operations
- **matplotlib / seaborn** → Visualization
- **scikit-learn** → Machine learning

Hands-On Activity

1. Open **DataSpell** or **Jupyter Notebook**.
2. Import a sample dataset (CSV of mushroom cultivation).

   ```
   import pandas as pd
   data = pd.read_csv("mushroom_data.csv")
   print(data.head())
   ```

3. Write a function to calculate average yield per species.
4. Visualise the frequency of different mushroom species using `matplotlib`.

Python Pandas

Introduction to Pandas

Pandas is one of the most important libraries in Python for data analysis. It provides high-level data structures and methods designed to make working with structured data both intuitive and efficient. At its core, Pandas introduces two fundamental data structures: **Series** and **DataFrame**. These structures allow us to handle data in tabular form, similar to spreadsheets or SQL tables, but with the full flexibility of Python programming.

In the context of **analytics for mushroom cultivation**, Pandas plays a central role because it allows us to clean sensor data (temperature, humidity, CO_2, voltage), manage time-series readings, and even join structured sensor outputs with image metadata. Without Pandas, handling raw CSV files, JSON logs, or real-time database feeds would be tedious and error-prone.

Pandas Data Structures

Series

- A one-dimensional labelled array capable of holding any data type.
- Think of it as a column in an Excel sheet or a single column of a SQL table.
- Example: Mushroom sensor readings (temperature over time).

```
import pandas as pd

temperature = pd.Series([22.5, 23.1, 22.8, 23.5],
                        index=['2025-09-17 08:00', '2025-09-17 09:00',
                               '2025-09-17 10:00', '2025-09-17 11:00'])
print(temperature)
```

DataFrame

- A two-dimensional labelled data structure with rows and columns.
- Similar to a full spreadsheet or a database table.
- Ideal for managing sensor data collected from mushroom farms.

```
data = {
    'Temperature': [22.5, 23.1, 22.8, 23.5],
    'Humidity': [85, 87, 86, 88],
    'CO2': [450, 470, 455, 480],
    'Voltage': [3.3, 3.3, 3.2, 3.3]
}

df = pd.DataFrame(data,
                  index=['2025-09-17 08:00', '2025-09-17 09:00',
                         '2025-09-17 10:00', '2025-09-17 11:00'])
print(df)
```

Key Operations in Pandas

1. **Importing Data**
   - Pandas can load data from CSV, Excel, JSON, SQL databases, and even web APIs.

```
df = pd.read_csv('mushroom_sensors.csv')
```

2. **Exploring Data**
   - Checking the shape, head, tail, and basic statistics.

```
print(df.head())     # First 5 rows
print(df.tail())     # Last 5 rows
print(df.describe()) # Summary statistics
```

3. **Indexing and Selection**
   - Selecting specific rows/columns using `loc` and `iloc`.

```
print(df['Temperature'])
print(df.loc['2025-09-17 09:00'])
print(df.iloc[2])  # third row
```

4. **Filtering Data**
   - Extracting values that meet certain conditions.

```
high_temp = df[df['Temperature'] > 23]
```

5. **Handling Missing Values**
   - Filling or dropping missing data.

```
df.fillna(0, inplace=True)
df.dropna(inplace=True)
```

6. **Aggregation and Grouping**
   - Useful when summarising data over time.

```
avg_daily = df.groupby(df.index).mean()
```

Why Pandas Matters in Mushroom Analytics

- **Sensor Fusion**: Combine data streams from temperature, humidity, and CO_2 sensors.
- **Data Cleaning**: Handle missing or corrupted sensor readings.
- **Time-Series Analysis**: Index sensor data by timestamps for predictive modeling.
- **Integration**: Join environmental data with growth outcomes (e.g., mushroom weight/size).
- **Visualisation Support**: Pandas integrates seamlessly with libraries like Matplotlib and Seaborn for quick plotting.

Practical Example

Imagine a mushroom farm with daily sensor readings. Pandas allows easy analysis:

```
# Calculate average temperature and humidity
print("Average Temperature:", df['Temperature'].mean())
print("Average Humidity:", df['Humidity'].mean())

# Find times when CO2 exceeds 460
alert_times = df[df['CO2'] > 460]
print(alert_times)
```

This could directly inform farm operators when to adjust ventilation or humidity to optimize mushroom growth.

Sample CSV Data

Loading Realistic Data Examples with Pandas

Sample CSV Data

CSV files are very common on Kaggle. For mushroom cultivation, a CSV might look like this:

`mushroom_sensors.csv`

```
Timestamp,Temperature,Humidity,CO2,Voltage
2025-09-17 08:00,22.5,85,450,3.3
2025-09-17 09:00,23.1,87,470,3.3
2025-09-17 10:00,22.8,86,455,3.2
2025-09-17 11:00,23.5,88,480,3.3
```

Load in Pandas:

```
import pandas as pd

df_csv = pd.read_csv("mushroom_sensors.csv")
print(df_csv.head())
```

Sample SQL Table

On Kaggle, datasets are often stored in relational formats. For mushrooms, an SQL table might look like:

SQL Table: **`sensor_readings`**

```
CREATE TABLE sensor_readings (
    id INTEGER PRIMARY KEY,
    timestamp TEXT,
    temperature REAL,
    humidity REAL,
    co2 INTEGER,
    voltage REAL
);

INSERT INTO sensor_readings VALUES
(1, '2025-09-17 08:00', 22.5, 85, 450, 3.3),
(2, '2025-09-17 09:00', 23.1, 87, 470, 3.3),
(3, '2025-09-17 10:00', 22.8, 86, 455, 3.2),
(4, '2025-09-17 11:00', 23.5, 88, 480, 3.3);
```

Load in Pandas:

```
import sqlite3

# Connect to SQLite
conn = sqlite3.connect("mushroom_data.db")

# Load SQL data into Pandas
df_sql = pd.read_sql_query("SELECT * FROM sensor_readings", conn)

print(df_sql.head())
conn.close()
```

Sample JSON Data

Kaggle also provides JSON logs or API-style datasets. For mushroom analytics, JSON might represent nested IoT sensor data.

`mushroom_sensors.json`

```
[
  {
    "timestamp": "2025-09-17 08:00",
    "sensors": {
      "temperature": 22.5,
      "humidity": 85,
      "co2": 450,
      "voltage": 3.3
    }
  },
  {
    "timestamp": "2025-09-17 09:00",
    "sensors": {
      "temperature": 23.1,
      "humidity": 87,
      "co2": 470,
      "voltage": 3.3
    }
  }
]
```

Load in Pandas:

```
df_json = pd.read_json("mushroom_sensors.json")

# Normalize nested fields
df_json_flat = pd.json_normalize(df_json["sensors"])
df_json_flat["timestamp"] = df_json["timestamp"]

print(df_json_flat.head())
```

EDA on Mushroom Dataset

1. Data Exploration & Preprocessing

- **Check missing values**: Handle NaN or abnormal readings (e.g., negative CO_2).
- **Statistical summary**: Mean, median, standard deviation, min, max.
- **Correlation analysis**: Pearson correlation among temperature, humidity, CO_2, voltage.

👉 Useful for identifying dependencies, e.g., whether CO_2 increases with temperature.

2. Visualization with Plotly

Here are some interactive plots to build:

- **Time Series Trends**

```
import plotly.express as px

fig = px.line(mushroom_df, x="timestamp", y="temperature_C",
title="Temperature Over Time")
fig.show()
```

  - Repeat for `humidity_percent`, `co2_ppm`, `voltage_V`.

- **Multi-variable Comparison**

```
fig = px.line(mushroom_df, x="timestamp", y=
["temperature_C","humidity_percent","co2_ppm"])
fig.show()
```

- **Scatter Plot (Relationship Analysis)**

```
fig = px.scatter(mushroom_df, x="temperature_C",
y="humidity_percent", color="co2_ppm",
                 title="Temperature vs Humidity (colored by CO2)")
fig.show()
```

- **Distribution**

```
fig = px.histogram(mushroom_df, x="temperature_C", nbins=20,
title="Temperature Distribution")
fig.show()
```

- **Correlation Heatmap**

```
import plotly.figure_factory as ff
corr = mushroom_df.drop(columns=["timestamp","image_file"]).corr()
fig = ff.create_annotated_heatmap(
    z=corr.values,
    x=list(corr.columns),
    y=list(corr.index),
    annotation_text=corr.round(2).values,
    colorscale="Viridis"
)
fig.show()
```

---

3. Analytics

- **Rolling Averages & Trends**
  - 6-hour or 12-hour rolling averages for smoothing.
- **Outlier Detection**
  - Z-score or IQR method to detect anomalies in CO_2 or humidity.
- **Feature Engineering**
  - Create new features such as "Temperature-Humidity Index (THI)" relevant for mushroom growth.

---

4. Forecasting

Since we have time-series data, we can try predictive modeling.

(a) Classical Models

- **ARIMA / SARIMA** → good for univariate forecasting (e.g., temperature only).
- **Prophet (Facebook)** → flexible, handles seasonality & trend.

Example with Prophet:

```
from prophet import Prophet

df_temp = mushroom_df[["timestamp","temperature_C"]].rename(columns=
{"timestamp":"ds","temperature_C":"y"})
model = Prophet()
model.fit(df_temp)

future = model.make_future_dataframe(periods=24, freq="H")
forecast = model.predict(future)

fig = model.plot(forecast)
```

(b) Deep Learning

- **LSTM (Long Short-Term Memory)** networks → effective for multivariate forecasting (temperature, CO_2, humidity, voltage together).
- Dummy pipeline:
  - Normalize data (MinMaxScaler).
  - Create sequences (lookback window).
  - Train/test split.
  - Build LSTM in Keras/TensorFlow.

---

5. Insights & Prescriptive Analytics

- **Threshold alerts**: e.g., if CO_2 > 1800 ppm → trigger alert.
- **Optimal ranges**:
  - Temperature: ~24–27°C
  - Humidity: ~85–90%
  - CO_2: ~800–1200 ppm
- Provide recommendations when readings deviate from optimal ranges.

Installing Prophet Library - Forecasting

Method 1: Using Anaconda Navigator (GUI)

1. **Open Anaconda Navigator**
   Launch Anaconda Navigator from your Start menu (Windows) or Applications folder (Mac).
2. **Select Environment**
   - Go to the **Environments** tab.
   - Choose the environment where you want Prophet installed (e.g., `base (root)` or your custom environment).
   - Click the **triangle/arrow** beside the environment → select **Open Terminal** (or "Open with Jupyter Notebook" if you want to run inline).
3. **Run the Install Command**
   In the terminal, type:

   ```
   conda install -c conda-forge prophet
   ```

   This will install Prophet along with its dependencies (like `cmdstanpy`).
4. **Verify Installation**
   After installation, open a Python console inside the same environment and run:

   ```
   from prophet import Prophet
   print("Prophet installed successfully")
   ```

Python Numpy

Introduction to NumPy

NumPy (Numerical Python) is the foundation of scientific computing in Python. It provides powerful data structures for efficient numerical operations, especially on large arrays and matrices. Unlike regular Python lists, NumPy arrays are stored in contiguous memory blocks and allow vectorised operations, which makes them much faster for numerical computations.

For **mushroom cultivation analytics**, NumPy is useful when dealing with sensor readings at scale. Imagine thousands of hourly measurements of temperature, CO_2, humidity, and voltage across multiple farms — NumPy allows us to compute averages, apply mathematical transformations, and prepare this data for machine learning models quickly.

NumPy Arrays

Creating Arrays

NumPy introduces the `ndarray` multi-dimensional array object.

```
import numpy as np

# One-dimensional array (temperature readings)
temp = np.array([22.5, 23.1, 22.8, 23.5])

# Two-dimensional array (temperature, humidity, CO₂, voltage)
data = np.array([
    [22.5, 85, 450, 3.3],
    [23.1, 87, 470, 3.3],
    [22.8, 86, 455, 3.2],
    [23.5, 88, 480, 3.3]
])
```

Here, each row represents a timestamp and each column a sensor type. This is similar to Pandas DataFrame, but with more emphasis on speed and mathematical operations.

Array Attributes

NumPy arrays come with properties that describe their structure:

```
print(data.shape)   # (4, 4) → 4 rows, 4 columns
print(data.ndim)    # 2 → two-dimensional
print(data.dtype)   # float64
```

Key Operations in NumPy

1. **Indexing and Slicing**
   - Accessing elements is similar to Python lists but supports multidimensional slicing.

```
print(temp[0])      # First temperature
print(data[2, 1])   # Row 3, Column 2 → Humidity at time 3
print(data[:, 0])   # All temperatures
```

2. **Vectorized Operations**
   - Instead of looping, you can apply operations to entire arrays.

```
temp_celsius = np.array([22.5, 23.1, 22.8, 23.5])
temp_fahrenheit = temp_celsius * 9/5 + 32
print(temp_fahrenheit)
```

   This avoids slow Python loops and makes analytics efficient.

3. **Statistical Functions**
   - NumPy provides fast aggregation functions.

```
print("Mean Temperature:", np.mean(data[:, 0]))
print("Max CO2:", np.max(data[:, 2]))
print("Min Humidity:", np.min(data[:, 1]))
```

4. **Reshaping Arrays**
   - Reshape data for machine learning models.

```
reshaped = data.reshape(2, 8)  # 2 rows, 8 columns
```

5. **Random Number Generation**
   - Useful for simulations or creating dummy sensor datasets.

```
random_temp = np.random.normal(loc=23, scale=1, size=10)
print(random_temp)
```

2.9 Why NumPy Matters in Mushroom Analytics

- **Efficient Sensor Computation**: Calculate rolling averages of CO_2 or humidity in real time.
- **Data Transformation**: Convert voltage signals into normalised values before feeding to models.
- **Time-Series Modelling:** Prepare large arrays of sensor readings for deep learning frameworks (TensorFlow, PyTorch).
- **Simulation**: Generate artificial data for training predictive models when real data is limited.

For example, simulating a week's worth of sensor data with random fluctuations:

```
hours = 24 * 7
simulated_temp = np.random.normal(loc=23, scale=1, size=hours)
simulated_humidity = np.random.normal(loc=85, scale=3, size=hours)
```

This allows researchers to test models before real-world deployment.

2.10 Practical Example: Combining Pandas and NumPy

Often, Pandas and NumPy work together:

```
import pandas as pd

# Create DataFrame
df = pd.DataFrame(data, columns=['Temperature', 'Humidity', 'CO2',
'Voltage'])

# Use NumPy inside Pandas
df['Temp_Normalized'] = (df['Temperature'] -
np.mean(df['Temperature'])) / np.std(df['Temperature'])

print(df)
```

Visualisation: Matplotlib

Introduction to Matplotlib

Matplotlib is the most widely used visualisation library in Python. It provides a flexible framework for creating static, animated, and interactive plots. In data analytics, visualisation is not just about making charts — it is about **communicating insights clearly and effectively**.

For mushroom cultivation analytics, visualising sensor readings such as **temperature, humidity, CO_2 levels, and voltage** can help researchers and farmers quickly identify patterns, detect anomalies, and make informed decisions. A time-series line chart of temperature, for instance, may reveal overheating issues in the cultivation chamber.

Basic Plotting

Line Plot

The simplest and most common visualisation for time-series data.

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

# Example time series (hours vs. temperature)
hours = np.arange(0, 10, 1)
temperature = [22.5, 22.7, 23.0, 23.1, 23.5, 23.3, 23.7, 23.6, 23.8,
24.0]

plt.plot(hours, temperature, marker='o')
plt.title("Mushroom Chamber Temperature Over Time")
plt.xlabel("Hour")
plt.ylabel("Temperature (°C)")
plt.grid(True)
plt.show()
```

This plot helps us observe how the temperature changes across 10 hours.

Bar Plot

Useful for comparing categorical data, such as **average conditions in different mushroom chambers**.

```
chambers = ['Chamber A', 'Chamber B', 'Chamber C']
avg_humidity = [85, 88, 82]

plt.bar(chambers, avg_humidity, color=['blue', 'green', 'orange'])
plt.title("Average Humidity by Mushroom Chamber")
plt.xlabel("Chamber")
plt.ylabel("Humidity (%)")
plt.show()
```

Histogram

Great for understanding the distribution of sensor readings.

```
# Simulated CO2 readings
co2_levels = np.random.normal(loc=450, scale=15, size=100)

plt.hist(co2_levels, bins=10, color='purple', edgecolor='black')
plt.title("Distribution of CO₂ Levels")
plt.xlabel("CO₂ (ppm)")
plt.ylabel("Frequency")
plt.show()
```

This reveals whether CO_2 levels stay close to a normal range or fluctuate dangerously.

2.13 Customizing Plots

Matplotlib offers extensive customisation:

- **Colors and Styles**

```
plt.plot(hours, temperature, color='red', linestyle='--',
marker='x')
```

- **Multiple Lines**

```
humidity = [85, 86, 87, 88, 87, 86, 85, 84, 85, 86]

plt.plot(hours, temperature, label="Temperature (°C)")
plt.plot(hours, humidity, label="Humidity (%)")

plt.title("Temperature vs Humidity Over Time")
plt.xlabel("Hour")
plt.ylabel("Values")
plt.legend()
plt.show()
```

- **Subplots**

```
fig, axs = plt.subplots(2, 1, figsize=(6, 8))

axs[0].plot(hours, temperature, color='red')
axs[0].set_title("Temperature Over Time")

axs[1].plot(hours, humidity, color='blue')
axs[1].set_title("Humidity Over Time")

plt.tight_layout()
plt.show()
```

This way, multiple metrics can be visualised in one figure for easy comparison.

Why Matplotlib Matters in Mushroom Analytics

- **Trend Detection**: Farmers can see temperature spikes or CO_2 dips that may affect mushroom growth.
- **Comparisons**: Different chambers or cultivation methods can be visually compared.
- **Data Quality Checks**: Outliers in humidity or voltage readings are easier to spot visually.
- **Decision Support**: Clear visual reports can guide environmental adjustments (ventilation, misting, lighting).

Practical Example: Multi-Sensor Visualization

```
import pandas as pd

# Example DataFrame of sensor readings
data = {
    "Hour": np.arange(1, 6),
    "Temperature": [22.5, 23.0, 22.8, 23.3, 23.5],
    "Humidity": [85, 86, 87, 88, 86],
    "CO2": [450, 460, 455, 470, 465]
}
df = pd.DataFrame(data)

# Plot multiple sensors
plt.figure(figsize=(8, 5))
plt.plot(df["Hour"], df["Temperature"], marker='o', label="Temperature
(°C)")
plt.plot(df["Hour"], df["Humidity"], marker='s', label="Humidity (%)")
plt.plot(df["Hour"], df["CO2"], marker='^', label="CO₂ (ppm)")

plt.title("Mushroom Cultivation Sensor Readings")
plt.xlabel("Hour")
plt.ylabel("Values")
plt.legend()
plt.grid(True)
plt.show()
```

Visualisation: Seaborn

Introduction to Seaborn

Seaborn is a Python visualisation library built on top of Matplotlib. While Matplotlib provides the basic plotting framework, Seaborn adds a **higher-level interface** with a focus on **statistics, patterns, and aesthetics**.

Where Matplotlib requires manual customisation, Seaborn automatically handles themes, colours, and layouts. This makes it especially powerful for **exploratory data analysis (EDA)** quickly spotting correlations, distributions, and anomalies in datasets.

For **mushroom cultivation analytics**, Seaborn can help us answer questions like:

- How does **humidity correlate with mushroom yield**?
- Is there a relationship between **CO_2 levels and voltage fluctuations**?
- What does the **distribution of temperature readings** look like across different chambers?

Getting Started with Seaborn

Seaborn integrates seamlessly with Pandas DataFrames.

```
import seaborn as sns
import pandas as pd

# Example DataFrame
data = {
    "Temperature": [22.5, 23.0, 22.8, 23.3, 23.5, 24.0, 23.7],
    "Humidity": [85, 86, 87, 88, 86, 89, 90],
    "CO2": [450, 460, 455, 470, 465, 480, 475],
    "Chamber": ["A", "A", "B", "B", "C", "C", "C"]
}
df = pd.DataFrame(data)
```

Seaborn Plot Types

Distribution Plots

Understand the spread of a variable.

```
sns.histplot(df["Temperature"], bins=5, kde=True, color="red")
plt.title("Temperature Distribution in Mushroom Chambers")
plt.show()
```

The KDE (Kernel Density Estimate) curve shows the smoothed distribution — useful to detect if sensor readings cluster around certain values.

Scatter Plots

Examine relationships between two variables.

```
sns.scatterplot(x="Temperature", y="Humidity", hue="Chamber", data=df)
plt.title("Temperature vs Humidity Across Chambers")
plt.show()
```

This helps see if chambers differ in climate control efficiency.

Correlation Heatmap

Visualise correlations between multiple features.

```
corr = df.corr(numeric_only=True)
sns.heatmap(corr, annot=True, cmap="coolwarm")
plt.title("Correlation Between Mushroom Sensor Readings")
plt.show()
```

For example, a strong positive correlation between **temperature and CO_2** may indicate poor ventilation.

Boxplots

Detect variability and outliers.

```
sns.boxplot(x="Chamber", y="CO2", data=df)
plt.title("CO₂ Levels by Mushroom Chamber")
plt.show()
```

This highlights if one chamber consistently records abnormal CO_2 levels.

Customization and Themes

Seaborn makes it easy to set styles and palettes globally:

```
sns.set_style("whitegrid")
sns.set_palette("Set2")

sns.lineplot(x="Temperature", y="CO2", hue="Chamber", data=df)
plt.title("Temperature vs CO₂ by Chamber")
plt.show()
```

Themes such as `"darkgrid"`, `"white"`, and `"ticks"` improve readability without extra code.

Why Seaborn Matters in Mushroom Analytics

- **Statistical Insights**: Quickly highlight correlations and anomalies.
- **EDA Power**: Before predictive modeling, Seaborn helps in spotting trends in sensor data.
- **Comparison Across Groups**: Easily compare chambers, batches, or growth stages.
- **Cleaner Visualisation:** Better defaults than Matplotlib for reports and presentations.

Practical Example: Multi-Sensor Visualization

```
# Pairplot for quick relationship overview
sns.pairplot(df, hue="Chamber", diag_kind="kde")
plt.suptitle("Pairwise Relationships in Mushroom Sensor Data", y=1.02)
plt.show()
```

Streamlit Web App

Prerequisites

- Python 3.9–3.12 installed (check with `python --version`).
- A code editor (VS Code, PyCharm, or your favourite).
- Basic familiarity with Python (variables, functions, lists/dicts).

Set Up Your Environment

Why? Separate project dependencies to avoid conflicts with other Python projects.

macOS / Linux

```
# 1. Create a project folder
mkdir streamlit-beginner && cd streamlit-beginner

# 2. Create & activate a virtual environment
python -m venv .venvsource .venv/bin/activate

# 3. Install Streamlit and basics
pip install streamlit pandas numpy altair
```

Windows (PowerShell)

```
# 1. Create a project folder
mkdir streamlit-beginner; cd streamlit-beginner

# 2. Create & activate a virtual environment
python -m venv .venv.\.venv\Scripts\Activate.ps1

# 3. Install Streamlit and basics
pip install streamlit pandas numpy altair
```

> ✅ Verify: `streamlit hello` opens a demo app in your browser.

Your First App (Hello, Streamlit!)

Create `app.py` with the minimal starter:

```
import streamlit as st
st.set_page_config(page_title="Hello Streamlit", page_icon="👋",
layout="centered")

st.title("👋 Hello, Streamlit!")
st.write("This is your first Streamlit app. Change the text and hot-
reload will update.")

name = st.text_input("What’s your name?", value="Friend")

if st.button("Greet"):st.success(f"Nice to meet you, {name}!")
```

Run it:

```
streamlit run app.py
```

What just happened?

- `st.title`, `st.write`, `st.button` are **widgets** and **elements** that render in the browser.
- Every interaction re-runs the script from top to bottom (Streamlit’s reactive model).

Layout Basics (Columns, Tabs, Expanders)

Add to `app.py` to learn layout primitives:

```
import streamlit as st

st.header("Layout Playground")

col1, col2 = st.columns([1, 2])
with col1:
st.subheader("Left")
st.write("Use columns for side-by-side content.")
with col2:
st.subheader("Right")
st.info("Column widths can be proportional.")


with st.expander("See tips"):
st.markdown("- Use `st.tabs([..])` for grouped views\n- Use
`st.container()` to group widgets")


tab1, tab2 = st.tabs(["Overview", "Details"])
with tab1:
st.write("High-level summary here.")
with tab2:
st.write("Deep dive here.")


st.header("Layout Playground")


col1, col2 = st.columns([1, 2])
with col1:
st.subheader("Left")
st.write("Use columns for side-by-side content.")
with col2:
st.subheader("Right")
st.info("Column widths can be proportional.")


with st.expander("See tips"):
st.markdown("- Use `st.tabs([..])` for grouped views\n- Use
`st.container()` to group widgets")

tab1, tab2 = st.tabs(["Overview", "Details"])
with tab1:
st.write("High-level summary here.")
with tab2:
st.write("Deep dive here.")
```

Widgets 101 (Input → Output)

```
st.header("Widgets 101")
age = st.slider("Age", min_value=1, max_value=100, value=21)
options = st.multiselect("Favorite fungi", ["Shiitake", "Oyster",
"Enoki", "Button"], ["Oyster"])
date = st.date_input("Harvest date")


st.write({"age": age, "options": options, "date": str(date)})
```

Tip: Most Streamlit widgets return a Python value. Use those values to compute and display results.

Display Data (Tables & Charts)

```
import pandas as pd
import numpy as npyth


st.header("Data & Charts")


df = pd.DataFrame({
"hour": np.arange(1, 11),
"temperature": [22.5, 22.7, 23.0, 23.1, 23.5, 23.3, 23.7, 23.6, 23.8,
24.0],
"humidity": [85, 86, 87, 88, 87, 86, 85, 84, 85, 86],
})


st.dataframe(df, use_container_width=True)


st.line_chart(df.set_index("hour")["temperature"])
st.bar_chart(df.set_index("hour")["humidity"])
```

File Uploads (CSV) + Quick EDA

Let users upload a CSV and plot it.

```
st.header("Upload a CSV")
file = st.file_uploader("Choose a CSV", type=["csv"])


if file is not None:
    user_df = pd.read_csv(file)
    st.subheader("Preview")
    st.dataframe(user_df.head(100), use_container_width=True)


    # Try to pick numeric columns automatically
    num_cols = user_df.select_dtypes(include=
[np.number]).columns.tolist()
    if num_cols:
        y = st.selectbox("Choose a numeric column to plot", num_cols)
    st.line_chart(user_df[y])
    else:
        st.warning("No numeric columns detected.")
else:
    st.info("Upload a CSV to begin (with headers).")
```

Multipage Apps

Create a `pages/` folder and add `1_Overview.py`, `2_Insights.py`.

app.py

```
import streamlit as st

st.set_page_config(page_title="Mushroom App", page_icon="🍄")
st.title("Main Page")
st.write("Navigate using the sidebar → Pages.")
```

pages/1_Overview.py

```
import streamlit as st

st.title("Overview")
st.write("High-level summary page.")
```

pages/2_Insights.py

```
import streamlit as st
st.title("Insights")
st.write("Charts, KPIs, and analysis.")
```

Run again and `streamlit run app.py` see the new sidebar navigation.

Beginner Streamlit App - Sample Mushroom Data

Beginner Streamlit App

```
import streamlit as st
import pandas as pd
import plotly.express as px

# --------------------------------
# Title
# --------------------------------
st.title("🍄 Beginner Streamlit App - Mushroom Data")

# --------------------------------
# Create dummy dataset
# --------------------------------
data = {
    "timestamp": pd.date_range("2025-01-01", periods=10, freq="D"),
    "temperature_C": [22, 23, 21, 24, 22, 23, 25, 26, 24, 23],
    "humidity_%": [70, 72, 68, 75, 73, 71, 74, 76, 72, 70],
    "CO2_ppm": [400, 420, 410, 430, 415, 425, 435, 440, 430, 420],
}
df = pd.DataFrame(data)

st.write("### Mushroom Dataset (Sample)")
st.dataframe(df)

# --------------------------------
# Select columns for visualization
# --------------------------------
x_axis = st.selectbox("Select X-axis:", df.columns, index=0)
y_axis = st.selectbox("Select Y-axis:", df.columns, index=1)

# --------------------------------
# Plot visualization
# --------------------------------
st.write("### Visualization")
fig = px.line(df, x=x_axis, y=y_axis, title=f"{y_axis} over {x_axis}")
st.plotly_chart(fig)
```

How to Run

1. Save this as `app.py`.
2. Run in terminal:

```
streamlit run app.py
```

3. The browser will open at `http://localhost:8501` with your interactive app.

Streamlit - Plotly

Instead of `fig.show()`, you should use `st.plotly_chart(fig)` to render Plotly figures.
Here's your revised code:

```
import streamlit as st
import plotly.express as px

# Example: assuming mushroom_df is already loaded
fig = px.line(mushroom_df, x="timestamp", y="temperature_C",
              title="Temperature Over Time")

# Display in Streamlit
st.plotly_chart(fig, use_container_width=True)
```

🔹 `use_container_width=True` makes the chart expand to fit the Streamlit page width.

Streamlit - Forecasting with Prophet

✅ Option 1: Keep Matplotlib (minimal change)

```
import streamlit as st
from prophet import Prophet

# Prepare data
df_temp = mushroom_df[["timestamp","temperature_C"]].rename(
    columns={"timestamp":"ds","temperature_C":"y"}
)

# Fit model
model = Prophet()
model.fit(df_temp)

# Make forecast
future = model.make_future_dataframe(periods=24, freq="H")
forecast = model.predict(future)

# Plot using Matplotlib
fig1 = model.plot(forecast)

# Show in Streamlit
st.pyplot(fig1)
```

✅ Option 2: Convert to Plotly (nicer in Streamlit)

```
import streamlit as st
from prophet import Prophet
import plotly.graph_objects as go

# Prepare data
df_temp = mushroom_df[["timestamp","temperature_C"]].rename(
    columns={"timestamp":"ds","temperature_C":"y"}
)

# Fit model
model = Prophet()
model.fit(df_temp)

# Make forecast
future = model.make_future_dataframe(periods=24, freq="H")
forecast = model.predict(future)

# Build Plotly figure
fig2 = go.Figure()
fig2.add_trace(go.Scatter(x=df_temp["ds"], y=df_temp["y"],
                          mode="lines", name="Actual"))
fig2.add_trace(go.Scatter(x=forecast["ds"], y=forecast["yhat"],
                          mode="lines", name="Forecast"))
fig2.add_trace(go.Scatter(x=forecast["ds"], y=forecast["yhat_upper"],
                          mode="lines", name="Upper Bound",
line=dict(dash="dash")))
fig2.add_trace(go.Scatter(x=forecast["ds"], y=forecast["yhat_lower"],
                          mode="lines", name="Lower Bound",
line=dict(dash="dash")))

fig2.update_layout(title="Temperature Forecast (Prophet)",
                   xaxis_title="Time", yaxis_title="Temperature (°C)")

# Show in Streamlit
st.plotly_chart(fig2, use_container_width=True)
```

Streamlit - Mushroom IoT Sensor Data Visualization and Forecasting

```
import streamlit as st
import plotly.express as px
import pandas as pd
from prophet import Prophet

st.title("Mushroom IoT Sensor Data Visualization and Forecasting")
st.sidebar.info("Visualize and forecast mushroom IoT sensor data using
Plotly and Prophet.")

df = pd.read_csv("mushroom_dataset.csv")

# Example: assuming mushroom_df is already loaded
fig = px.line(df, x="timestamp", y="temperature_C",
              title="Temperature Over Time")

# Display in Streamlit
st.plotly_chart(fig, use_container_width=True)

fig = px.scatter(df, x="temperature_C", y="humidity_percent",
color="co2_ppm",
                 title="Temperature vs Humidity (colored by CO2)")
# Display in Streamlit
st.plotly_chart(fig, use_container_width=True)

# Prepare data
df_temp = df[["timestamp","temperature_C"]].rename(
    columns={"timestamp":"ds","temperature_C":"y"}
)

# Fit model
model = Prophet()
model.fit(df_temp)

# Make forecast
future = model.make_future_dataframe(periods=24, freq="H")
forecast = model.predict(future)

# Plot using Matplotlib
fig1 = model.plot(forecast)

# Show in Streamlit
st.pyplot(fig1)
```

Streamlit - Date Picker

```
import streamlit as st
from prophet import Prophet
import plotly.graph_objects as go
import pandas as pd

# Assume mushroom_df already loaded
# Ensure timestamp is datetime
mushroom_df = pd.read_csv("mushroom_dataset.csv")
mushroom_df["timestamp"] = pd.to_datetime(mushroom_df["timestamp"])

st.subheader("🔮 Temperature Forecast with Prophet")

# Sidebar calendar picker
col1, col2 = st.columns(2)
with col1:
    start_date = st.date_input("Start date",
mushroom_df["timestamp"].min().date())
with col2:
    end_date = st.date_input("End date",
mushroom_df["timestamp"].max().date())

# Filter dataset by selected range
mask = (mushroom_df["timestamp"].dt.date >= start_date) &
(mushroom_df["timestamp"].dt.date <= end_date)
df_filtered = mushroom_df.loc[mask, ["timestamp",
"temperature_C"]].rename(
    columns={"timestamp": "ds", "temperature_C": "y"}
)

if df_filtered.empty:
    st.warning("No data available in the selected range.")
else:
    # Fit model
    model = Prophet(daily_seasonality=True, weekly_seasonality=True)
    model.fit(df_filtered)

    # Forecast horizon: extend 24h beyond selected end date
    last_date = pd.to_datetime(end_date)
    horizon_hours = 24
    future = model.make_future_dataframe(periods=horizon_hours,
freq="H")
    forecast = model.predict(future)

    # Build Plotly figure
    fig = go.Figure()
    fig.add_trace(go.Scatter(x=df_filtered["ds"], y=df_filtered["y"],
                             mode="lines", name="Actual"))
    fig.add_trace(go.Scatter(x=forecast["ds"], y=forecast["yhat"],
```

```
                              mode="lines", name="Forecast"))
    fig.add_trace(go.Scatter(x=forecast["ds"],
y=forecast["yhat_upper"],
                              mode="lines", name="Upper Bound",
line=dict(dash="dash")))
    fig.add_trace(go.Scatter(x=forecast["ds"],
y=forecast["yhat_lower"],
                              mode="lines", name="Lower Bound",
line=dict(dash="dash")))

    fig.update_layout(title="Temperature Forecast (Prophet)",
                      xaxis_title="Time", yaxis_title="Temperature
(°C)")

    # Show in Streamlit
    st.plotly_chart(fig, use_container_width=True)
```

Streamlit Sidebar Example

Beginner Streamlit Sidebar Example

```
import streamlit as st
import pandas as pd
import plotly.express as px

# -------------------------------
# Title
# -------------------------------
st.title("🍄 Streamlit Sidebar Example")

# -------------------------------
# Sidebar
# -------------------------------
st.sidebar.title("Settings")
st.sidebar.write("Use the controls below to customize the chart:")

# Dummy dataset
data = {
    "Day": list(range(1, 11)),
    "Temperature_C": [22, 23, 21, 24, 22, 23, 25, 26, 24, 23],
    "Humidity_%": [70, 72, 68, 75, 73, 71, 74, 76, 72, 70],
}
df = pd.DataFrame(data)

# Sidebar selectboxes
x_axis = st.sidebar.selectbox("Select X-axis:", df.columns, index=0)
y_axis = st.sidebar.selectbox("Select Y-axis:", df.columns, index=1)

# Sidebar radio button for chart type
chart_type = st.sidebar.radio("Select Chart Type:", ["Line", "Bar",
"Scatter"])

# -------------------------------
# Show dataset
# -------------------------------
st.write("### Sample Mushroom Data")
st.dataframe(df)

# -------------------------------
# Visualization
# -------------------------------
st.write("### Visualization")

if chart_type == "Line":
    fig = px.line(df, x=x_axis, y=y_axis, title=f"{y_axis} over
{x_axis}")
elif chart_type == "Bar":
    fig = px.bar(df, x=x_axis, y=y_axis, title=f"{y_axis} by {x_axis}")
else:
```

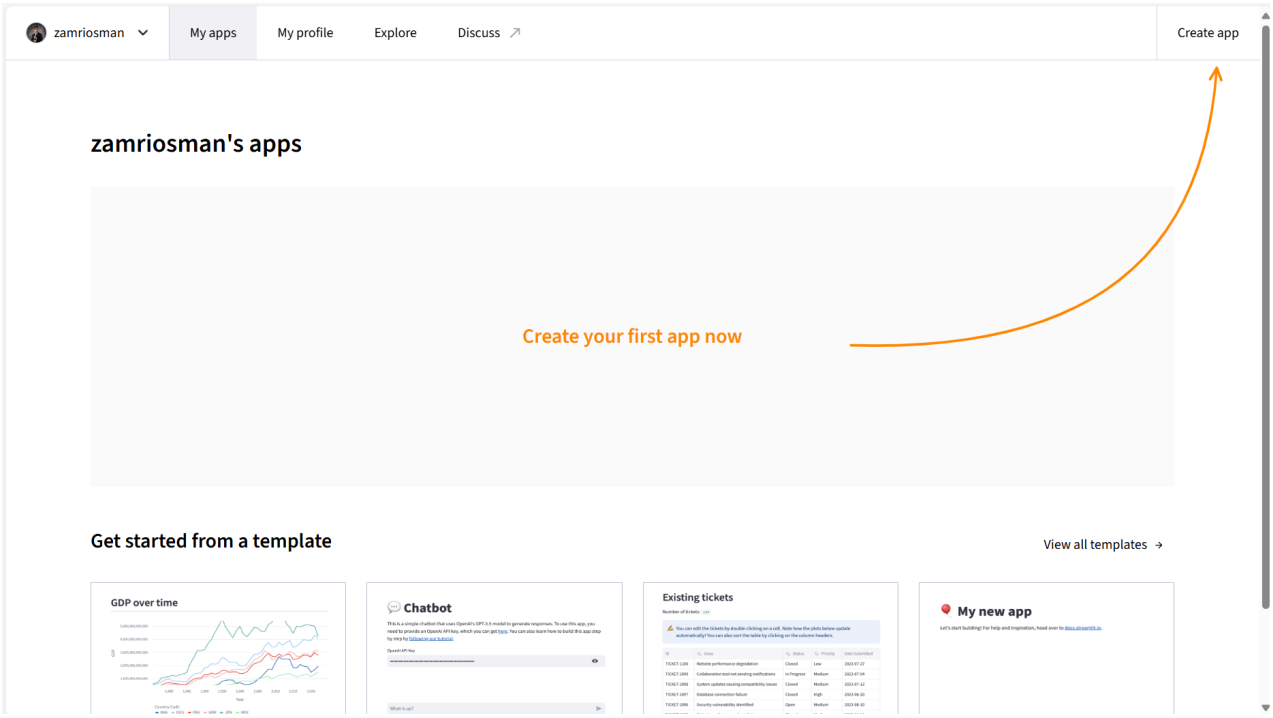
```
    fig = px.scatter(df, x=x_axis, y=y_axis, title=f"{y_axis} vs
{x_axis}")

st.plotly_chart(fig)
```

What This App Does

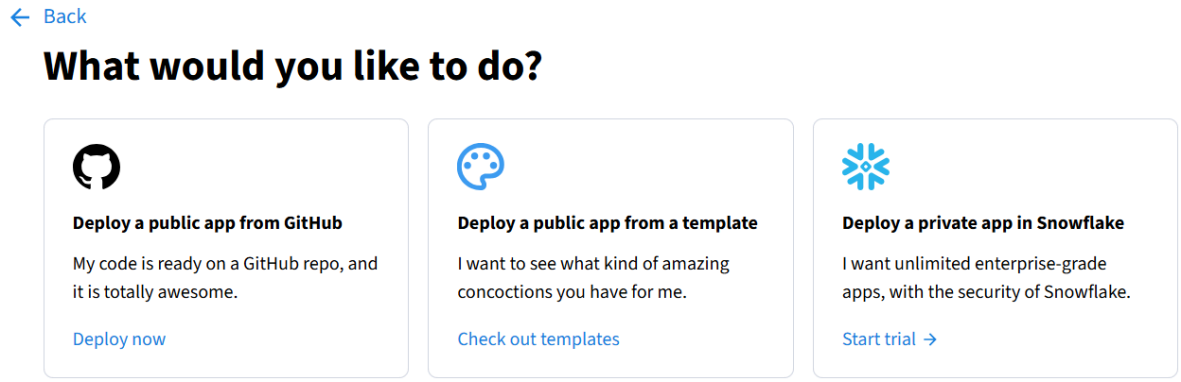
- ✅ Creates a **sidebar** with `st.sidebar`.
- ✅ Lets you pick **X-axis and Y-axis** from the sidebar.
- ✅ Lets you switch chart type (Line, Bar, Scatter).
- ✅ Displays a **dummy mushroom dataset** and a chart.

Deploy Streamlit Web App

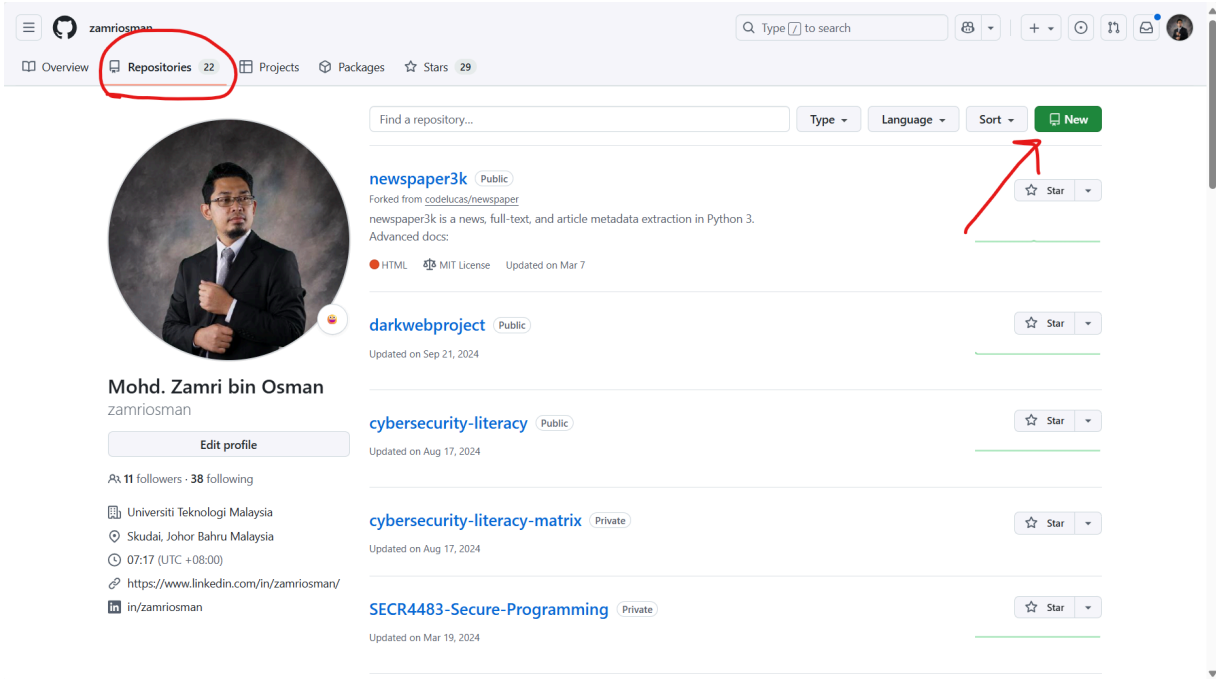
1. Signup using Google or Github Account. Recommended using Github



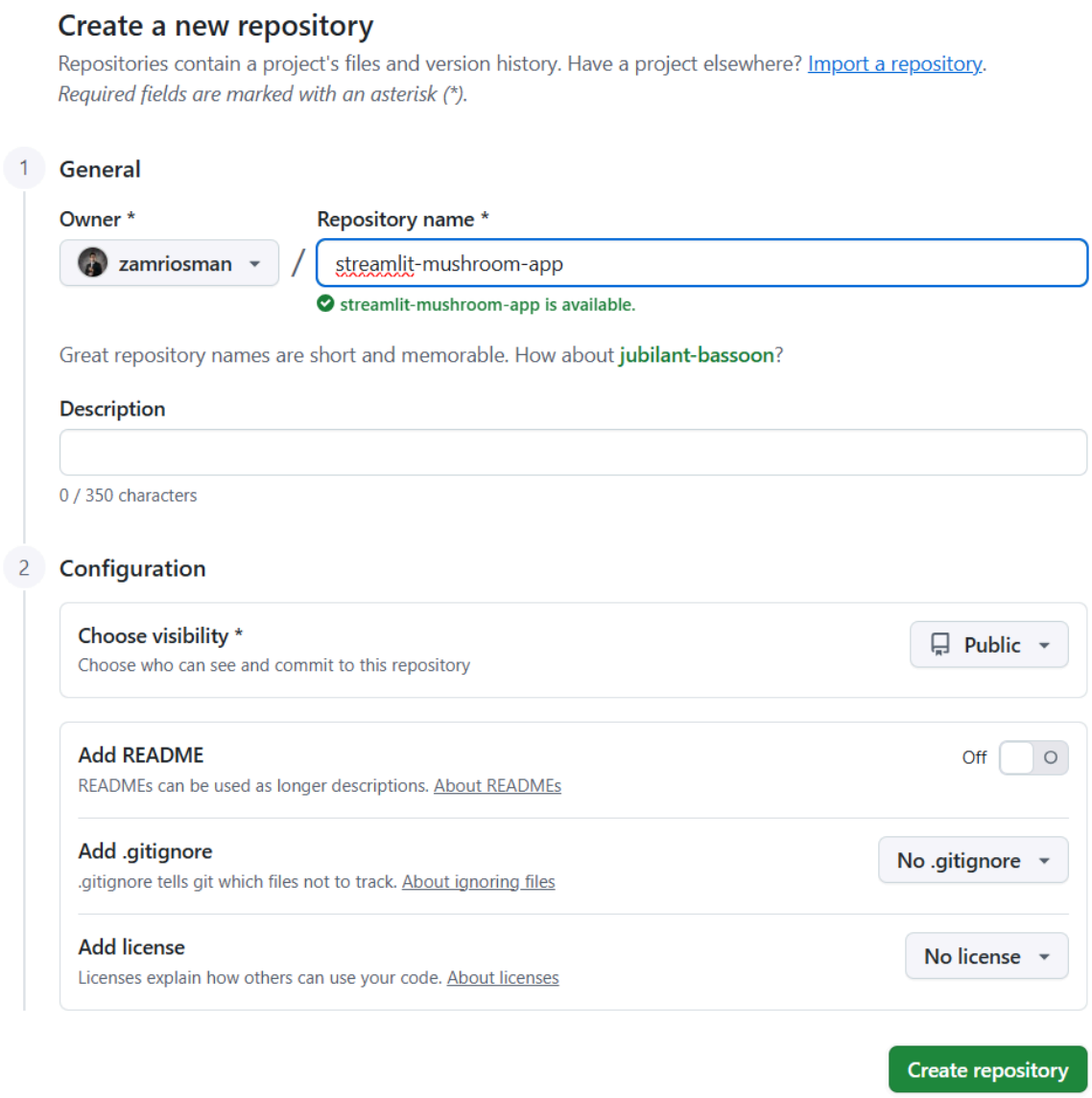
2. Click on "Create app"
3. Choose the first option (Github) - Deply a public app from Github



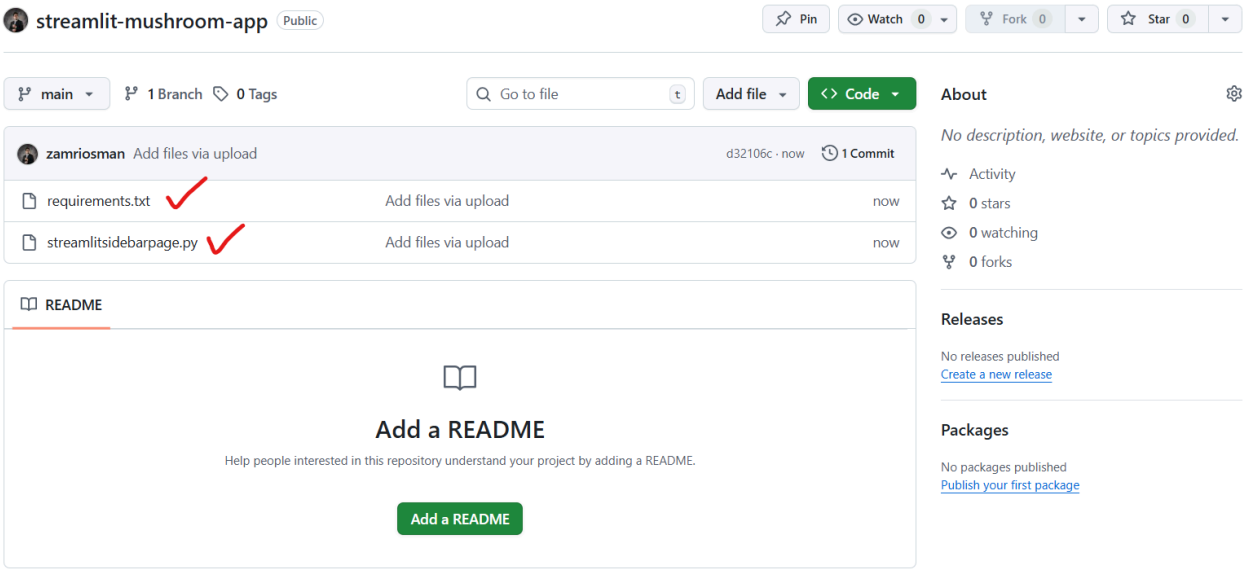
4. Create Github Repo



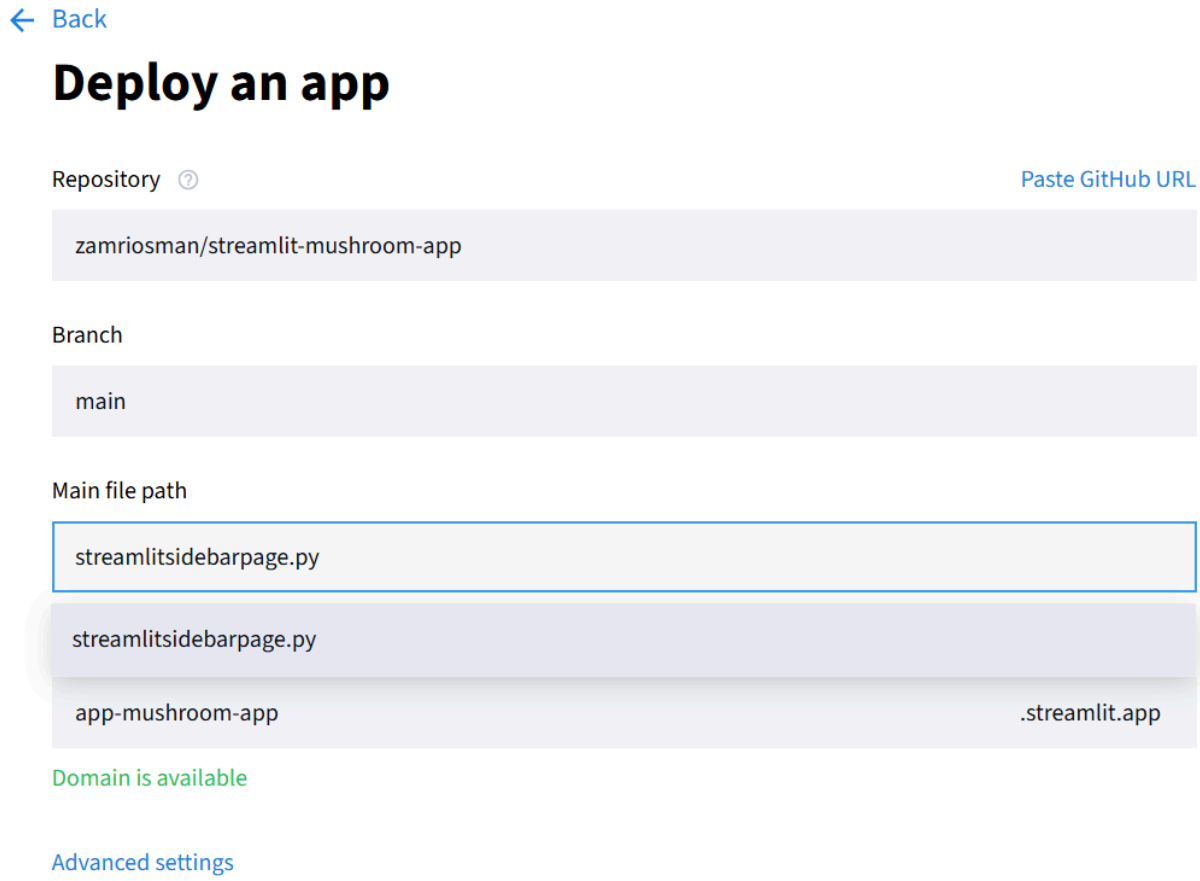
5. Fill up new Github Repo



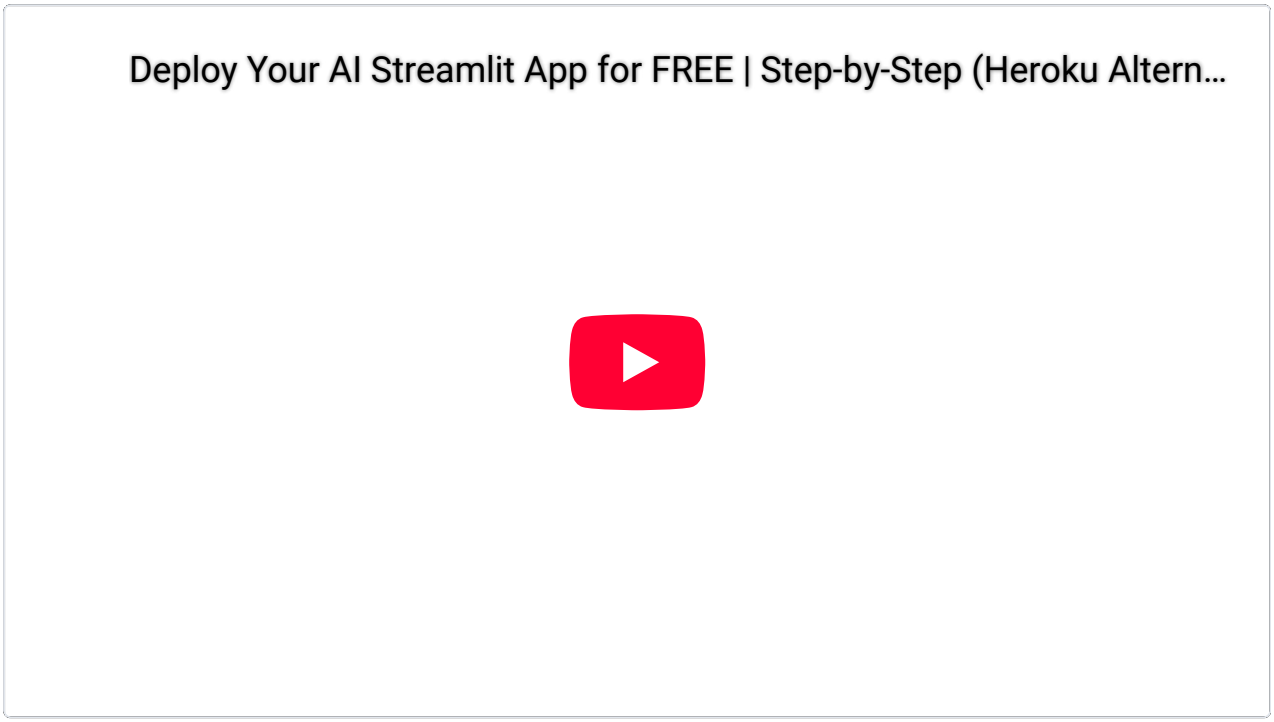
4. Upload project's files: app.py and requirements.txt



5. Get back to Streamlit Cloud. Choose the Repo that has been created in your Github



Video - Deploy App



Topic 3: Predictive Modeling

3.1 Overview

Predictive modeling is at the heart of data analytics. It uses **historical data** and **machine learning algorithms** to forecast outcomes. In the context of mushroom analytics, predictive modeling helps answer questions such as:

- *"Will the mushroom grow successfully under the current temperature and humidity?"*
- *"Is this mushroom edible or poisonous?"*
- *"What will the yield be in the next cycle?"*

This topic introduces the concepts, workflows, and practical implementation of predictive modeling.

3.2 Learning Objectives

By the end of this topic, participants will be able to:

- Understand the basics of **supervised learning** (classification vs regression).
- Split data into **training and testing sets** for model evaluation.
- Build predictive models using **Scikit-learn**.
- Evaluate models using metrics such as **accuracy, precision, recall, RMSE, and ROC AUC**.
- Apply predictive modeling to both **general datasets** (Titanic, Boston Housing) and later to **mushroom data**.

3.3 Types of Predictive Modeling

1. Classification Models

- Used when the output is a **category**.
- Examples:
  - Edible vs poisonous mushroom.
  - Survival vs non-survival in Titanic dataset.
- Common algorithms:
  - Logistic Regression
  - Decision Trees
  - Random Forests
  - Support Vector Machines (SVM)

🔗 References:

- Scikit-learn Classification Overview → https://scikit-learn.org/stable/supervised_learning.html
- UCI Mushroom Dataset → https://archive.ics.uci.edu/ml/datasets/Mushroom

---

2. Regression Models

- Used when the output is a **numeric value**.
- Examples:
  - Predicting mushroom yield (in kg) given temperature and humidity.
  - Predicting house prices in the Boston Housing dataset.
- Common algorithms:
  - Linear Regression
  - Ridge/Lasso Regression
  - Random Forest Regression

🔗 References:

- Regression Analysis Guide (Scikit-learn) → https://scikit-learn.org/stable/modules/linear_model.html
- Boston Housing Dataset → https://www.kaggle.com/c/boston-housing

---

3.4 Workflow for Predictive Modeling

The general steps are:

1. **Data Preparation** → Clean, encode, and scale features.
2. **Train-Test Split** → Divide into training and testing sets.
3. **Model Selection** → Choose appropriate algorithm.
4. **Training** → Fit the model to training data.
5. **Prediction** → Use the model on test data.
6. **Evaluation** → Measure performance with metrics.

3.5 Hands-On Lab: Classification (Titanic Dataset)

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report

# Load Titanic dataset
df = pd.read_csv("titanic.csv")

# Select features and target
X = df[["Pclass", "Age", "Fare"]].fillna(0)
y = df["Survived"]

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Train model
model = RandomForestClassifier()
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)

# Evaluation
print("Accuracy:", accuracy_score(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

🔎 **Explanation**

- **Features (X)**: Pclass, Age, Fare.
- **Target (y)**: Survival (0 = no, 1 = yes).
- **RandomForestClassifier**: Builds multiple decision trees and averages results.
- `classification_report` → shows accuracy, precision, recall, and F1-score.

🔗 Titanic Dataset → https://www.kaggle.com/c/titanic

3.6 Hands-On Lab: Regression (Boston Housing Dataset)

```
from sklearn.datasets import load_boston
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import numpy as np

# Load dataset
boston = load_boston()
X = boston.data
y = boston.target

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Train model
model = LinearRegression()
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)

# Evaluation
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
print("Root Mean Squared Error:", rmse)
```

🔎 **Explanation**

- Linear Regression → predicts numeric outcomes.
- RMSE (Root Mean Squared Error) → measures prediction error size.
- Smaller RMSE = better performance.

🔗 Boston Housing Dataset Guide → https://www.kaggle.com/c/boston-housing

---

3.7 Model Evaluation Metrics

- **Accuracy** → Percentage of correct predictions (useful for balanced classification).
- **Precision** → Out of predicted positives, how many were correct?
- **Recall** → Out of actual positives, how many were identified?
- **F1-Score** → Harmonic mean of precision & recall (balances false positives/negatives).
- **RMSE** → Standard error measure for regression tasks.
- **ROC AUC** → Measures ability of classification model to separate classes.

🔗 Guide to Model Evaluation Metrics → https://scikit-learn.org/stable/modules/model_evaluation.html

---

3.8 Application to Mushroom Data

Once learners are comfortable with the Titanic and Boston datasets, the same workflow is applied to **mushroom datasets**:

- **Classification**: Predict whether a mushroom is **edible or poisonous** (UCI Mushroom dataset).
- **Regression**: Predict **yield or growth** based on **temperature, humidity, and CO_2** (Kaggle Mushroom Growth dataset).

🔗 Mushroom Growth Dataset → https://www.kaggle.com/datasets/maysee/mushroom-growth

---

3.9 Summary

In this topic, you learnt:

- The difference between **classification** and **regression** models.
- The standard **predictive modeling workflow**.
- How to build and evaluate models in Python.
- How general datasets (Titanic, Boston) prepare you for domain datasets (Mushroom).

3.10 Self-Assessment Questions

1. What is the main difference between classification and regression?
2. Why do we split data into training and testing sets?
3. Which evaluation metric is better when detecting poisonous mushrooms: accuracy or recall? Why?
4. How could regression models help predict mushroom yield?

<details>
<summary>Q1. What is the main difference between classification and regression?</summary>

Answer:

- **Classification** predicts **categories (discrete values)** such as *edible vs poisonous* or *survived vs not survived*.
- **Regression** predicts **continuous values (numeric outputs)** such as mushroom yield in kilograms or house prices in dollars.

👉 Example:

- Classification → “Is this mushroom safe to eat?”
- Regression → “What will the mushroom yield be given current condition

</details>

Q2. Why do we split data into training and testing sets?

Answer:
Splitting ensures that the model is **trained on one subset (training set)** and **evaluated on unseen data (test set)**.

- This prevents **overfitting**, where the model memorises training data but fails on new data.
- It allows us to measure how well the model will perform in **real-world scenarios**.

👉 Example: Train on 80% of mushroom sensor data and test on 20% to confirm the model generalises to new conditions.

Q3. Which evaluation metric is better when detecting poisonous mushrooms: accuracy or recall? Why?

Answer:

- **Recall** is more important than accuracy in this case.
- Recall measures how many of the **actual poisonous mushrooms** were correctly identified.
- High recall ensures that **fewer poisonous mushrooms are misclassified as edible**, which is critical for safety.

👉 Example: Even if a model has 95% accuracy, missing 5% of poisonous mushrooms could be deadly. A model with high recall ensures safety.

Q4. How could regression models help predict mushroom yield?

Answer:
Regression models can estimate **future yields** based on environmental factors like temperature, humidity, CO_2, and voltage.

- Input (features): sensor data (humidity, temperature, CO_2).
- Output (target): predicted mushroom yield (numeric value, e.g., kilograms).
- This allows farmers to plan resources, adjust conditions, and maximize harvests.

👉 Example: A regression model could predict that if **humidity is maintained at 75% and CO_2 below 800 ppm**, yield will increase by 10%.

Topic 4: From Prediction to Prescription

4.1 Overview

While predictive analytics answers *"What is likely to happen?"*, prescriptive analytics goes further to answer:

👉 *"What should we do about it?"*

Prescriptive analytics combines **predictions, decision rules, and optimisation techniques** to recommend the best course of action.

In mushroom cultivation, prediction might tell us that **humidity is likely to drop below 60%**, while prescription would suggest:

- *"Turn on the misting system for 15 minutes."*
- *"Increase ventilation to reduce CO_2 buildup."*

This topic introduces the principles, techniques, and hands-on practice of prescriptive analytics.

4.2 Learning Objectives

By the end of this topic, participants will be able to:

- Explain the difference between **predictive** and **prescriptive analytics**.
- Understand the role of **decision rules** in guiding actions.
- Apply **optimisation techniques** using Python libraries such as **Scipy.optimize** and **PuLP**.
- Design a simple prescriptive model for agricultural decision-making.

4.3 Decision Rules

Decision rules are **if–then statements** that translate predictions into actions.

Example in Mushroom Cultivation:

- **Rule 1:** If humidity < 60%, then activate misting.
- **Rule 2:** If CO_2 > 1200 ppm, then increase ventilation.
- **Rule 3:** If temperature > 28°C, then reduce voltage on heating equipment.

These rules can be manually defined or derived from optimisation models.

🔗 Reference:

- Business Rules in Analytics → IBM Decision Management

4.4 Optimization in Prescriptive Analytics

Optimisation is the process of finding the **best possible decision** given a set of constraints and objectives.

Common techniques:

- **Linear Programming (LP)** → optimising a linear objective subject to constraints.
- **Integer Programming (IP)** → similar to LP but with discrete decision variables.
- **Multi-objective optimisation** → balancing multiple goals (e.g., maximise yield while minimising energy cost).

🔗 References:

- PuLP Linear Programming Library → https://coin-or.github.io/pulp/
- Scipy Optimize Documentation → https://docs.scipy.org/doc/scipy/reference/optimize.html

4.5 Hands-On Lab: Simple Resource Allocation

Problem:

A mushroom farm needs to decide how many hours per day to run **misting** and **ventilation** systems.

- Constraint: Total electricity available = 10 units.
- Misting consumes 2 units/hour.
- Ventilation consumes 1 unit/hour.
- Each hour of misting increases yield by 5 units.
- Each hour of ventilation increases yield by 3 units.

Goal → Maximise mushroom yield.

Python Code with PuLP

```
import pulp

# Define problem
model = pulp.LpProblem("Mushroom_Farm_Optimization", pulp.LpMaximize)

# Decision variables
misting = pulp.LpVariable("Misting_Hours", lowBound=0, cat="Integer")
ventilation = pulp.LpVariable("Ventilation_Hours", lowBound=0,
cat="Integer")

# Objective function: Maximize yield
model += 5 * misting + 3 * ventilation

# Constraint: Electricity budget
model += 2 * misting + 1 * ventilation <= 10

# Solve
model.solve()

# Results
print("Misting Hours:", misting.value())
print("Ventilation Hours:", ventilation.value())
print("Max Yield:", pulp.value(model.objective))
```

🔎 Explanation

- `LpProblem(..., LpMaximize)` → defines a maximisation problem.
- `misting` and `ventilation` → decision variables.
- `model += ...` → adds objective and constraints.
- `model.solve()` → runs the optimisation solver.

📌 Result: The solver finds the optimal number of misting and ventilation hours to maximise yield given limited electricity.

👉 Later, this can be extended to include **real mushroom sensor data**.

4.6 Prescriptive Dashboard Integration

In practice, prescriptive analytics should feed into an interactive dashboard. Example features:

- Display current sensor readings (temperature, CO_2, humidity).
- Show predicted yield (from Topic 3 models).
- Recommend optimal actions (from optimisation models).

🔗 Streamlit for dashboards → https://streamlit.io/

4.7 Summary

- **Predictive analytics** forecasts outcomes.
- **Prescriptive analytics** recommends **actions** to achieve goals.
- **Decision rules** provide straightforward mappings from prediction → action.
- **Optimisation techniques** (LP, IP) ensure the best use of limited resources.
- Mushroom cultivation benefits from prescriptive models to balance yield, quality, and resource costs.

4.8 Self-Assessment Questions

1. What is the main difference between predictive and prescriptive analytics?
2. Give an example of a decision rule in mushroom cultivation.
3. Why is optimisation important in prescriptive analytics?
4. In the resource allocation example, what was the objective, and what was the constraint?

Q1. What is the main difference between predictive and prescriptive analytics?

Answer:

- **Predictive analytics** → forecasts what is likely to happen in the future based on historical data and models.
- **Prescriptive analytics** → recommends what actions should be taken to achieve the best outcome, often using optimisation and decision rules.

👉 Example: Predictive = *"CO_2 levels will rise above safe limits tomorrow."*
Prescriptive = *"Open ventilation for 2 hours tomorrow to reduce CO_2."*

Q2. Give an example of a decision rule in mushroom cultivation.

Answer:

- **Rule:** If **humidity < 60%**, then turn on the **misting system for 15 minutes**.
- **Why:** Low humidity can prevent mushrooms from fruiting, so maintaining proper levels ensures yield quality.

👉 Other rules:

- If CO_2 > 1200 ppm → Increase ventilation.
- If temperature > 28°C → Reduce heating system voltage.

Q3. Why is optimisation important in prescriptive analytics?

Answer:
Optimisation ensures that limited resources (time, electricity, water, space) are used in the **most effective way** to maximise outcomes.

- Without optimisation → actions may be inefficient or conflicting.
- With optimisation → prescriptive models find the **best balance** between constraints and goals.

👉 Example: Decide the **optimal hours** to run misting vs ventilation when electricity is limited so that yield is maximised without overspending energy.

Q4. In the resource allocation example, what was the objective, and what was the constraint?

Answer:

- **Objective:** Maximise mushroom yield → `5 * misting + 3 * ventilation`.
- **Constraint:** Electricity usage cannot exceed 10 units → `2 * misting + 1 * ventilation ≤ 10`.

👉 This ensures the farm maximises yield **while staying within energy limits**.

4.9 References

- Provost, F., & Fawcett, T. (2013). *Data Science for Business*. O'Reilly.
- Winston, W. L. (2020). *Operations Research: Applications and Algorithms*. Cengage.
- Scikit-learn: Model Evaluation → https://scikit-learn.org/stable/modules/model_evaluation.html
- PuLP Documentation → https://coin-or.github.io/pulp/
- Scipy Optimize Documentation → https://docs.scipy.org/doc/scipy/reference/optimize.html

PuLP

What you'll learn

- How to translate a real problem into a **Linear Program (LP)**.
- How to use **PuLP** (Python) to define decision variables, an objective, and constraints.
- How to read a simple **Kaggle-style CSV** and use it to build the model.
- How to debug infeasibility and add practical constraints (max servings, "pick at most K items", integer choices).

Setup

```
pip install pulp pandas
```

PuLP ships with the open-source CBC solver by default. No extra solver config is needed for this tutorial.

Open-source CBC

> Open-source CBC refers to software projects and tools related to the Cipher Block Chaining (CBC) mode of operation in cryptography that are made available with open-source licenses. These projects allow developers and researchers to access, modify, and distribute the source code freely. This transparency promotes collaboration and innovation in the development of cryptographic methods, ensuring robust security practices are maintained.

A simple Kaggle-style dataset (CSV)

Create a file named **diet_simple.csv** with this content (or use a similar Kaggle diet dataset; the column names are the key part):

```
food,cost,calories,protein_g,fat_g,sodium_mg
oatmeal,0.50,150,5,3,2
chicken_breast,2.20,165,31,3.6,74
eggs,0.40,78,6,5,62
milk,0.30,103,8,2.4,107
apples,0.35,95,0.5,0.3,2
rice,0.20,206,4.3,0.4,1
```

> Assumptions (typical of many Kaggle diet datasets): units are per serving; costs are in arbitrary currency; nutrients per serving are approximate.

4.4 Problem statement (in everyday terms)

> **Goal:** Find the cheapest combination of foods (number of servings of each) that meets daily nutrition needs.

We'll **minimise cost** subject to:

- Calories ≥ 2000
- Protein ≥ 50 g
- Fat between 20 g and 70 g
- Sodium ≤ 2400 mg

Decision variables:

$x_i \geq 0$ = number of servings of food i.

4.5 Baseline model in PuLP (continuous variables)

```
import pandas as pd
from pulp import LpProblem, LpMinimize, LpVariable, lpSum, LpStatus,
value

# 1) Load data
df = pd.read_csv("diet_simple.csv")

# 2) Build the LP model
model = LpProblem("Diet_MinCost", LpMinimize)

# 3) Decision variables: servings of each food (continuous, >= 0)
foods = df["food"].tolist()
x = {f: LpVariable(f"x_{f}", lowBound=0) for f in foods}

# 4) Objective: minimize total cost
costs = dict(zip(df.food, df.cost))
model += lpSum(costs[f] * x[f] for f in foods), "Total_Cost"

# 5) Constraints (nutrition requirements)
calories = dict(zip(df.food, df.calories))
protein  = dict(zip(df.food, df.protein_g))
fat      = dict(zip(df.food, df.fat_g))
sodium   = dict(zip(df.food, df.sodium_mg))

# Calories ≥ 2000
model += lpSum(calories[f] * x[f] for f in foods) >= 2000,
"Calorie_min"

# Protein ≥ 50 g
model += lpSum(protein[f]  * x[f] for f in foods) >= 50, "Protein_min"

# Fat between 20 and 70 g
model += lpSum(fat[f]      * x[f] for f in foods) >= 20, "Fat_min"
model += lpSum(fat[f]      * x[f] for f in foods) <= 70, "Fat_max"

# Sodium ≤ 2400 mg
model += lpSum(sodium[f]   * x[f] for f in foods) <= 2400, "Sodium_max"

# 6) Solve
model.solve()  # CBC by default

print("Status:", LpStatus[model.status])
print("Min cost:", value(model.objective))

# 7) Show results
sol = []
for f in foods:
    servings = x[f].value()
```

```
    if servings and servings > 1e-6:
        sol.append((f, servings))
sol_df = pd.DataFrame(sol, columns=["food", "servings"])
print(sol_df)

# 8) Check nutrient totals vs. bounds
tot_cal = sum(calories[f] * (x[f].value() or 0) for f in foods)
tot_pro = sum(protein[f]  * (x[f].value() or 0) for f in foods)
tot_fat = sum(fat[f]      * (x[f].value() or 0) for f in foods)
tot_sod = sum(sodium[f]   * (x[f].value() or 0) for f in foods)

print({"calories": tot_cal, "protein_g": tot_pro, "fat_g": tot_fat,
"sodium_mg": tot_sod})
```

What to expect:

- **Status** should be `Optimal`.
- You'll see the minimum cost and a few foods with non-zero servings.
- The nutrient totals should satisfy the bounds.

4.6 Add realism: per-item upper bounds (palatability / availability)

Sometimes you don't want the model to choose 30 servings of one cheap item. Add **upper bounds**, e.g., at most 10 servings of any single food:

```
MAX_SERVINGS = 10
for f in foods:
    x[f].upBound = MAX_SERVINGS
```

Re-solve and examine how the mix changes.

4.7 “Pick at most K different foods” (binary selection)

Add a binary variable $y_f \in \{0, 1\}$ that indicates whether food f is selected. Link x_f to y_f with a “big-M” constraint $x_f \leq M \cdot y_f$. Then cap the number of chosen foods.

```
from pulp import LpBinary

# Rebuild from scratch (cleanest for beginners):
model2 = LpProblem("Diet_MinCost_WithBinaries", LpMinimize)
x = {f: LpVariable(f"x_{f}", lowBound=0) for f in foods}
y = {f: LpVariable(f"y_{f}", lowBound=0, upBound=1, cat=LpBinary) for f
in foods}

# Objective
model2 += lpSum(costs[f] * x[f] for f in foods)

# Same nutrition constraints
model2 += lpSum(calories[f] * x[f] for f in foods) >= 2000
model2 += lpSum(protein[f]  * x[f] for f in foods) >= 50
model2 += lpSum(fat[f]      * x[f] for f in foods) >= 20
model2 += lpSum(fat[f]      * x[f] for f in foods) <= 70
model2 += lpSum(sodium[f]   * x[f] for f in foods) <= 2400

# Linking and selection constraints
M = 10.0  # big-M per item (max servings if selected)
for f in foods:
    model2 += x[f] <= M * y[f]

# Choose at most 3 different foods
model2 += lpSum(y[f] for f in foods) <= 3

model2.solve()
print("Status:", LpStatus[model2.status])
print("Min cost:", value(model2.objective))
print([(f, x[f].value(), y[f].value()) for f in foods if (x[f].value()
or 0) > 1e-6 or (y[f].value() or 0) > 0.5])
```

4.8 Common pitfalls & how to debug

- **Infeasible model** (Status = Infeasible):
  Your bounds may be too tight. Try relaxing one constraint at a time (e.g., raise sodium limit, widen fat range) to find the culprit.
- **Unbounded model**:
  You might be missing a key constraint (e.g., calories ≥ 2000). Add reasonable bounds.
- **Weirdly large servings**:
  Add upper bounds per item, or add binary selection to limit variety.

4.9 Packaging results for reporting

Turn the solution into a tidy table with totals and gaps vs. constraints:

```
import math

def safe(v):
    return 0.0 if v is None or math.isnan(v) else v

res = []
for f in foods:
    s = safe(x[f].value())
    if s > 1e-6:
        res.append({
            "food": f, "servings": round(s, 3),
            "cost": round(costs[f]*s, 2),
            "calories": round(calories[f]*s, 1),
            "protein_g": round(protein[f]*s, 1),
            "fat_g": round(fat[f]*s, 1),
            "sodium_mg": round(sodium[f]*s, 1),
        })
out = pd.DataFrame(res).sort_values("cost", ascending=False)
print(out)

totals = {
    "total_cost": round(sum(r["cost"] for r in res), 2),
    "calories": round(sum(r["calories"] for r in res), 1),
    "protein_g": round(sum(r["protein_g"] for r in res), 1),
    "fat_g": round(sum(r["fat_g"] for r in res), 1),
    "sodium_mg": round(sum(r["sodium_mg"] for r in res), 1),
}
print("Totals:", totals)
```

4.10 Exercises (do these to build intuition)

1. **Tighten protein:** Increase minimum protein to 80 g. How does the cost change? Which foods get added?
2. **Sodium cap:** Lower sodium max to 1500 mg. Does the model stay feasible? If not, which constraint is binding?
3. **Integer servings:** Force **integer servings**: make `x[f]` integer with `cat="Integer"` in `LpVariable`. What's the new optimal cost?
4. **At most 2 foods:**Change the selection limit to 2 and see if it stays feasible.
5. **Swap dataset:** Replace `diet_simple.csv` with a small Kaggle diet dataset that has similar columns; update column names if needed.

4.11 Connect to mushroom prescriptive analytics

Once you're comfortable:

- **Resource mix:** maximise total **yield (kg)** subject to substrate, labour hours, shelves, and humidity control time limits.
- **Cycle scheduling:** Decide batch sizes across weeks to meet an order target while minimising overtime or energy cost.
  Both are direct analogues of what you just built: change the objective and swap nutrients for farm resources.

Mushroom Analytics Application

1. Introduction

This module applies your analytics toolkit to a real agritech scenario: **mushroom cultivation**. You'll build an end-to-end system that ingests **sensor streams**—**timestamp, temperature, CO_2, humidity, voltage**—and **image data**, then produces **predictions** (growth, risk, yield) and **prescriptions** (actions to optimize conditions). The outcome is a working prototype with a live dashboard and decision logic suitable for farm operations.

---

2. Learning Objectives

By the end, you will be able to:

- Design a **data pipeline** for multi-modal inputs (time-series + images).
- Perform **EDA** on sensor data and detect anomalies.
- Train **classification** (e.g., edible/poisonous, risk flags) and **regression** models (e.g., yield/quality scores).
- Implement **prescriptive analytics** (rule-based + optimization) to recommend actions (misting, ventilation, lighting/voltage).
- Build an **interactive dashboard** that connects predictions to actionable recommendations.
- Evaluate your system with **clear metrics** and communicate results effectively.

---

3. Target Data & Use Cases

Data modalities

- **Timestamp**: aligns multi-sensor signals; enables resampling/lag features.
- **Temperature (°C)**: growth rate, contamination risk.
- **CO_2 (ppm)**: ventilation indicator; fruiting quality.
- **Humidity (%)**: pinning/fruiting success, cap integrity.
- **Voltage (V)**: proxy for equipment state/energy cost.
- **Image**: quality grading (healthy/diseased, growth stage).

Typical use cases

- **Yield prediction** (regression).
- **Risk detection** (e.g., contamination, suboptimal microclimate).
- **Anomaly detection** (sensor drift, stuck actuators).
- **Prescriptions** (how long to mist/ventilate; when to adjust voltage).

---

4. System Architecture (Concept)

Ingest → Prepare → Learn → Prescribe → Visualize → Operate

1. **Ingest**: CSV/JSON logs, IoT broker (simulated), image folders.
2. **Prepare**: cleaning, resampling by timestamp, feature engineering (rolling means, lags, interaction terms), label encoding for categorical sets.
3. **Learn**: ML models (Random Forest, Logistic/Linear Regression), optional CNN for images (transfer learning).
4. **Prescribe**: rules + optimization (PuLP/`scipy.optimize`) under constraints (energy budget, thresholds).
5. **Visualize**: Streamlit/Dash dashboard—trends, predictions, recommendations, alerts.
6. **Operate**: human-in-the-loop acceptance; log outcomes for continuous improvement.

---

5. Tools & Tech Stack

- **Python**: Pandas, NumPy, Scikit-learn
- **Visualization**: Matplotlib, Seaborn
- **Dashboards**: Streamlit or Dash
- **Optimization**: PuLP, `scipy.optimize`
- **Images**: Pillow; TensorFlow/Keras (transfer learning with MobileNet/ResNet)
- **Quality & Ops**: `scikit-learn` pipelines, train/test splits, metrics dashboards

---

6. Datasets (suggested)

- **UCI Mushroom** (categorical, edible vs poisonous) — classification baseline
- **Kaggle Mushroom Growth / cultivation** (time-series: temp, humidity, CO_2, voltage) — regression + anomaly detection
- **Kaggle Mushroom Images** — image classification for quality/disease

(Swap in client's real sensor streams when available.)

---

7. Session Flow (Application Phase)

1. **Data Onboarding & EDA**
    - Load sensor data; align by **timestamp**; visualize trends & correlations.
    - Outlier/shift detection (rolling z-scores, IQR, STL).
2. **Feature Engineering**
    - Rolling windows (mean/max), **lag features**, temp–humidity interactions, CO_2 rate of change, voltage duty cycles.
3. **Modeling**
    - **Regression**: predict yield/quality index.
    - **Classification**: risk flags (OK / Attention / Critical).
    - **Metrics**: RMSE/MAE (regression), recall/F1/ROC AUC (classification).
4. **Prescriptive Layer**
    - Decision rules (thresholds).
    - LP/IP optimization: e.g., maximize yield subject to electricity ≤ budget, humidity ∈ [lo, hi], CO_2 ≤ cap.
5. **Dashboard Build**
    - Live charts (temp, humidity, CO_2, voltage), model outputs, **recommended actions with rationales**, what-if controls.
6. **Validation & Demo**
    - Backtest on historical slices; stress test with edge cases; present findings.

---

8. Hands-On Labs (capstone oriented)

- **Lab A**: Time-series ETL & EDA (resampling, lags, rolling windows).
- **Lab B**: Yield predictor (Random Forest Regressor) + feature importance.
- **Lab C**: Risk classifier (Logistic/Forest) prioritizing **recall** for safety.
- **Lab D**: Optimization mini-project (misting vs ventilation vs energy).
- **Lab E**: Streamlit dashboard with actions & alerts.
- **(Optional) Lab F**: Image quality classifier via transfer learning; integrate score into prescriptions.

---

9. Prescriptive Patterns (starter rules)

- If **humidity < 60%** → *mist N minutes* (optimize N).
- If **CO_2 > 1200 ppm** → *increase ventilation duty cycle*.
- If **temperature > 28 °C** and **CO_2 rising** → *prioritize ventilation over heating*.
- If **image risk score ≥ τ** → *raise contamination alert* and recommend **cooldown + airflow** adjustment.

(Convert rules to constraints/variables to co-optimize with energy limits.)

---

10. Evaluation & Success Criteria

- **Predictive**:
  - Regression RMSE/MAE meets target; residuals show no major bias.
  - Classification recall ≥ threshold for **critical risk**; stable ROC AUC.
- **Prescriptive**:
  - Feasible action plans under energy constraints; reduced time in unsafe ranges.
- **System**:
  - Dashboard responsive; clear recommendations with explanations; reproducible pipeline.

---

11. Deliverables

- Clean, versioned **notebooks/scripts** (ETL, modeling, prescription).
- **Streamlit/Dash app** (packaged with instructions).
- **Model report** (data schema, features, metrics, error analysis).
- **Prescriptive spec** (rules, optimization model, constraints).
- **Demo deck** with screenshots and scenario walkthroughs.

Topic 5: Mushroom Dataset Exploration

5.1 Overview

After mastering analytics with general datasets, we now move to **mushroom cultivation data**. Mushroom farming is highly sensitive to environmental conditions such as **temperature, humidity, and CO_2 levels**, and these datasets provide an excellent case study for **predictive and prescriptive analytics**.

We will explore three key types of mushroom datasets:

1. **Categorical attributes (UCI Mushroom Dataset)** → edible vs poisonous classification.
2. **Sensor-style data (Kaggle Mushroom Growth Dataset)** → time-series of environmental conditions (temperature, humidity, CO_2, voltage).
3. **Image datasets (Kaggle Mushroom Images)** → for quality control and disease detection.

5.2 Learning Objectives

By the end of this topic, participants will be able to:

- Load and explore mushroom datasets from UCI and Kaggle.
- Perform data cleaning and preprocessing (encoding, handling missing values).
- Conduct Exploratory Data Analysis (EDA) on sensor-style mushroom data.
- Understand how categorical, numerical, and image features complement one another.

5.3 UCI Mushroom Dataset (Categorical Data)

- Contains **8,124 mushrooms** from 23 species.
- Each mushroom is described by **22 categorical features**, e.g.:
  - `cap-shape`, `cap-surface`, `cap-color`, `odor`, `gill-size`, etc.
- Target label: **edible (e)** or **poisonous (p)**.

Example: Load UCI Mushroom Dataset

```
import pandas as pd

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

# View first rows
print(df.head())

# Check class distribution
print(df["class"].value_counts())
```

🔎 **Explanation**

- `mushrooms.csv` contains categorical features.
- `df["class"].value_counts()` shows how many mushrooms are edible vs poisonous.

👉 This dataset is a **classic classification problem**.

🔗 UCI Mushroom Dataset → https://archive.ics.uci.edu/ml/datasets/Mushroom

---

5.4 Kaggle Mushroom Growth Dataset (Sensor Data)

- Provides **time-series data** with environmental readings:
  - **Timestamp**
  - **Temperature (°C)**
  - **Humidity (%)**
  - **CO_2 level (ppm)**
  - **Voltage (V)**
- Useful for **predicting growth/yield** and for prescriptive recommendations.

Example: Load and Explore

```
# Load mushroom growth dataset
growth_df = pd.read_csv("mushroom_growth.csv")

# Inspect structure
print(growth_df.info())

# Summary statistics
print(growth_df.describe())

# Time-series plot
import matplotlib.pyplot as plt
plt.plot(growth_df["timestamp"], growth_df["humidity"])
plt.title("Humidity over Time")
plt.xlabel("Time")
plt.ylabel("Humidity (%)")
plt.show()
```

🔎 Explanation

- `growth_df.info()` → structure of dataset.
- `growth_df.describe()` → mean, min, max for sensor values.
- Time-series plot shows how humidity changes during cultivation.

👉 This simulates **real sensor data pipelines** for mushroom farms.

🔗 Kaggle Mushroom Growth Dataset → https://www.kaggle.com/datasets/maysee/mushroom-growth

---

5.5 Mushroom Image Dataset

- Contains thousands of **mushroom images** across different species and conditions.
- Useful for **computer vision tasks**:
  - Detecting diseases.
  - Identifying species.
  - Grading quality.

Example: Load and Display an Image

```
from PIL import Image
import matplotlib.pyplot as plt

# Load an image
img = Image.open("mushroom.jpg")

# Show image
plt.imshow(img)
plt.axis("off")
plt.title("Mushroom Sample Image")
plt.show()
```

🔎 **Explanation**

- Used `PIL.Image` to load and display an image.
- This will later be connected to **deep learning models** (CNNs).

🔗 Kaggle Mushroom Image Dataset →
https://www.kaggle.com/datasets/kmader/mushroom-classification ↗

5.6 Exploratory Data Analysis (EDA) for Mushroom Data

Typical EDA tasks include:

- **Distribution analysis**: histograms of temperature, humidity, and CO_2.
- **Correlation analysis**: find relationships between environmental factors and growth.
- **Outlier detection**: abnormal sensor readings.
- **Categorical frequency**: which cap colours are most common among poisonous mushrooms?

Example: Correlation Heatmap

```
import seaborn as sns

corr = growth_df.corr()
sns.heatmap(corr, annot=True, cmap="coolwarm")
plt.title("Correlation between Sensor Variables")
plt.show()
```

🔎 Explanation

- `growth_df.corr()` → correlation between numerical features.
- `sns.heatmap()` → visualises correlations with colour coding.

👉 Example: High correlation between **CO_2** and **temperature** may indicate ventilation issues.

5.7 Summary

- Mushroom datasets provide both **categorical (edibility)** and **numerical (sensor)** features.
- Data preprocessing is critical: encoding categorical values and handling missing sensor readings.
- EDA provides insight into how environmental conditions affect mushroom growth.
- Image datasets add another dimension by allowing **visual inspection and classification**.

5.8 Self-Assessment Questions

1. What is the target variable in the UCI Mushroom Dataset?
2. Which features are typically available in mushroom sensor datasets?
3. Why is correlation analysis important when exploring sensor data?
4. How could image datasets enhance mushroom cultivation analytics beyond sensor data?

<details>
<summary>Q1. What is the target variable in the UCI Mushroom Dataset?</summary>

Answer:

- The target variable is **class**, which indicates whether a mushroom is **edible (e)** or **poisonous (p)**.
- This makes the dataset a **classification problem**, where the goal is to predict mushroom edibility based on attributes such as odour, cap colour, and gill size.

👉 In practice, this is useful for **food safety** and as a benchmark for classification models.
</details>

Q2. Which features are typically available in mushroom sensor datasets?

Answer:
Mushroom sensor datasets often include:

- **Timestamp** → when the reading was recorded.
- **Temperature (°C)** → affects fungal metabolism and growth rate.
- **Humidity (%)** → crucial for pinning and fruiting.
- **CO_2 (ppm)** → indicates ventilation and air exchange quality.
- **Voltage (V)** → power consumption of equipment (e.g., fans, misting systems).

👉 These features make sensor datasets suitable for **time-series analysis** and **predictive modeling** of yield and risk.

Q3. Why is correlation analysis important when exploring sensor data?

Answer:
Correlation analysis helps to:

- **Identify relationships** between environmental factors (e.g., temperature and CO_2).
- **Detect redundancy** (e.g., if two sensors provide overlapping information).
- **Guide feature selection** by showing which variables strongly influence outcomes (e.g., yield, contamination risk).

👉 Example: A strong **negative correlation** between CO_2 and yield may suggest that high CO_2 reduces production.

▽ Q4. How could image datasets enhance mushroom cultivation analytics beyond sensor data?

Answer:
Image datasets add a **visual layer of analysis** that sensors cannot capture, including:

- **Disease detection** → spotting fungal infections or mold growth.
- **Growth stage classification** → identifying whether mushrooms are in pinning, fruiting, or harvesting stage.
- **Quality grading** → detecting cap deformities, discoloration, or size irregularities.

👉 Combining image analysis with sensor data allows a **multi-modal analytics system**, providing more accurate and actionable insights for mushroom farming.

5.9 References

- UCI Mushroom Dataset → https://archive.ics.uci.edu/ml/datasets/Mushroom
- Kaggle Mushroom Growth Dataset → https://www.kaggle.com/datasets/maysee/mushroom-growth
- Kaggle Mushroom Images → https://www.kaggle.com/datasets/kmader/mushroom-classification
- Géron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly.

Topic 6: Building the Analytics Pipeline

6.1 Overview

An **analytics pipeline** is the structured process of transforming raw data into insights and actions. It connects the stages of:

1. **Extract** → Gather data from sources (CSV, JSON, sensors, images).
2. **Transform** → Clean, preprocess, and enrich data.
3. **Load** → Store data for further analysis or dashboards.
4. **Model** → Apply machine learning models for predictions.
5. **Prescribe** → Use optimization and decision rules to recommend actions.

In mushroom analytics, the pipeline allows us to take sensor logs (temperature, humidity, CO_2, voltage) and images, process them, and produce **yield predictions** and **optimal cultivation strategies**.

6.2 Learning Objectives

By the end of this topic, participants will be able to:

- Design an **end-to-end pipeline** for mushroom data.
- Implement data ingestion, preprocessing, and feature engineering.
- Handle **time-series sensor data**, **categorical mushroom attributes**, and **image features**.
- Prepare clean datasets for predictive and prescriptive modeling.

6.3 Pipeline Stages in Mushroom Analytics

1. Data Extraction (E)

- Load CSV/JSON sensor logs.
- Connect to IoT devices (simulated in training).
- Import image files for vision tasks.

```
import pandas as pd

# Load sensor data
sensor_df = pd.read_csv("mushroom_growth.csv")

# Load categorical data
mushroom_df = pd.read_csv("mushrooms.csv")

print(sensor_df.head())
print(mushroom_df.head())
```

---

2. Data Transformation (T)

Includes **cleaning, handling missing values, encoding, feature engineering**.

Handling Missing Values

```
# Fill missing humidity values with median
sensor_df["humidity"] =
sensor_df["humidity"].fillna(sensor_df["humidity"].median())
```

Encoding Categorical Features

```
# Encode mushroom class (e = edible, p = poisonous)
mushroom_df["class"] = mushroom_df["class"].map({"e": 0, "p": 1})
```

Feature Engineering

```
# Create rolling averages for humidity (window = 3 hours)
sensor_df["humidity_avg"] =
sensor_df["humidity"].rolling(window=3).mean()
```

👉 Rolling features capture **short-term environmental trends** that impact growth.

---

3. Data Loading (L)

- Store cleaned datasets in **CSV/Parquet** format for efficient access.
- Optional: Insert into SQL/NoSQL databases for dashboard integration.

```
sensor_df.to_csv("cleaned_sensor.csv", index=False)
mushroom_df.to_csv("cleaned_mushrooms.csv", index=False)
```

---

4. Modeling

This step links to **Topic 3 (Predictive Modeling)**, but now uses the **cleaned and engineered features**.

- Classification → edible vs poisonous (categorical).
- Regression → yield prediction (sensor-driven).
- Image-based CNN → disease/quality detection.

---

5. Prescription

This step links to **Topic 4 (From Prediction to Prescription)**. Once predictions are ready, they are fed into:

- **Decision rules** → If humidity < threshold, mist.
- **Optimisation models** → Maximize yield under energy and ventilation constraints.

6.4 Hands-On Lab: Integrated Pipeline

```
# Step 1: Load
df = pd.read_csv("mushroom_growth.csv")

# Step 2: Transform
df["humidity"] = df["humidity"].fillna(df["humidity"].median())
df["humidity_avg"] = df["humidity"].rolling(window=3).mean()

# Step 3: Split features and target
X = df[["temperature", "humidity_avg", "co2", "voltage"]]
y = df["yield"]

# Step 4: Train-test split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Step 5: Model
from sklearn.ensemble import RandomForestRegressor
model = RandomForestRegressor()
model.fit(X_train, y_train)

# Step 6: Predict
y_pred = model.predict(X_test)

# Step 7: Evaluate
from sklearn.metrics import mean_squared_error
import numpy as np
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
print("RMSE:", rmse)
```

🔎 Explanation

- Pipeline integrates extraction, transformation, and modeling.
- Uses rolling averages for humidity to reduce noise.
- Random Forest predicts **yield** from environmental features.
- RMSE measures prediction error.

6.5 Pipeline Design Considerations

- **Scalability** → Handle continuous IoT data streams.
- **Data Quality** → Fill missing values, detect sensor errors.
- **Feature Engineering** → Use domain knowledge (e.g., CO_2 rise → poor ventilation).
- **Multi-Modal Fusion** → Combine sensor, categorical, and image features.
- **Automation** → Pipelines should run automatically at defined intervals.

🔗 Reference:

- Scikit-learn Pipelines → https://scikit-learn.org/stable/modules/compose.html

---

6.6 Summary

- An **analytics pipeline** structures the journey from raw data → insights → actions.
- Mushroom analytics requires **multi-modal handling** (sensor, categorical, image).
- ETL ensures clean, structured datasets for modeling.
- Feature engineering (e.g., rolling averages, lags) improves predictions.
- Pipelines prepare the ground for prescriptive dashboards.

6.7 Self-Assessment Questions

1. What are the three main stages of an ETL pipeline?
2. How can rolling averages help in analysing mushroom sensor data?
3. Why is encoding necessary when working with the UCI Mushroom Dataset?
4. What metric would you use to evaluate a regression model predicting mushroom yield, and why?

Q1. What are the three main stages of an ETL pipeline?

Answer:

The three main stages of ETL are:

1. **Extract** → Collect data from sources (e.g., CSV logs, IoT sensors, image datasets).
2. **Transform** → Clean, preprocess, and enrich data (handle missing values, encode categorical features, and create rolling averages).
3. **Load** → Store the prepared data into a database, file, or system for analysis and modeling.

👉 Example: Extract mushroom sensor data → clean humidity and CO_2 values → load into a cleaned CSV for modeling.

Q2. How can rolling averages help in analysing mushroom sensor data?

Answer:

- Rolling averages smooth short-term fluctuations and highlight long-term trends.
- They reduce the effect of **noise** in raw sensor readings.
- They make it easier to detect **shifts or anomalies** in environmental conditions.

👉 Example: A 3-hour rolling average of **humidity** provides a clearer trend of farm conditions, avoiding false alarms caused by momentary sensor spikes.

⌄ Q3. Why is encoding necessary when working with the UCI Mushroom Dataset?

Answer:

- The UCI Mushroom Dataset has **categorical variables** (e.g., odor = “almond,” “foul,” “fishy”).
- Machine learning models require **numeric inputs**, so encoding is needed.
- Techniques include:
  - **Label Encoding** → Assigns numbers to categories (e.g., almond = 0, foul = 1).
  - **One-Hot Encoding** → Creates binary columns for each category.

👉 Without encoding, models cannot interpret the mushroom attributes correctly

⌄ Q4. What metric would you use to evaluate a regression model predicting mushroom yield, and why?

Answer:

- Common metrics:
  - **RMSE (Root Mean Squared Error)** → measures average error size, penalises large errors.
  - **MAE (Mean Absolute Error)** → measures average absolute difference, more robust to outliers.
- For mushroom yield prediction, **RMSE** is preferred because:
  - It gives more weight to large errors (important when predicting yield, as big mistakes could cause resource misallocation).

👉 Example: An RMSE of 2.5 kg means the model’s predictions are off by ~2.5 kg on average, which helps farmers plan more accurately.

6.8 References

- Scikit-learn Pipelines → https://scikit-learn.org/stable/modules/compose.html
- Pandas Rolling Windows → https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.rolling.html
- UCI Mushroom Dataset → https://archive.ics.uci.edu/ml/datasets/Mushroom
- Kaggle Mushroom Growth Dataset → https://www.kaggle.com/datasets/maysee/mushroom-growth

Topic 7: Dashboard Development

7.1 Overview

A dashboard transforms raw analytics and model outputs into **actionable, user-friendly insights**. For mushroom cultivation, dashboards allow farmers or researchers to:

- **Monitor real-time conditions** (temperature, humidity, CO_2, voltage).
- **Visualize predictions** (yield forecasts, contamination risk).
- **Receive prescriptive recommendations** (ventilation hours, misting actions).
- **Display image analysis results** (disease detection, growth stage).

This topic teaches how to design and build an **interactive dashboard** using Python tools such as **Streamlit** and **Dash**.

7.2 Learning Objectives

By the end of this topic, participants will be able to:

- Understand the purpose and design principles of dashboards.
- Build interactive dashboards with **Streamlit/Dash**.
- Integrate sensor data, model predictions, and prescriptive recommendations.
- Add interactive features such as filters, inputs, and what-if scenarios.

7.3 Dashboard Design Principles

A good dashboard should be:

- **Clear** → Simple charts with labels, not overloaded with information.
- **Actionable** → Present insights that lead to decisions (not just raw numbers).
- **Interactive** → Allow filtering, input, and exploration.
- **Real-time (if possible)** → Reflect sensor updates as they come in.

👉 Example layout for Mushroom Analytics Dashboard:

1. **Header** → Project title, last data refresh time.
2. **KPIs** → Current humidity, CO_2, temperature, voltage.
3. **Trends** → Line charts for time-series data.
4. **Predictions** → Forecast yield, risk classification.
5. **Prescriptions** → Suggested actions (e.g., misting/ventilation).
6. **Image Panel** → Display results from mushroom image classification.

7.4 Tools for Dashboard Development

- **Streamlit** → Easiest for beginners; Python scripts → instant dashboard.
- **Dash (Plotly)** → More flexible; great for multi-page apps.
- **Panel/Bokeh** → Advanced visualization options.

🔗 References:

- Streamlit → https://streamlit.io/
- Dash → https://dash.plotly.com/

7.5 Hands-On Lab: Streamlit Dashboard

Basic Setup

```
import streamlit as st
import pandas as pd
import matplotlib.pyplot as plt

# Load sensor data
df = pd.read_csv("cleaned_sensor.csv")

# Title
st.title("🍄 Mushroom Analytics Dashboard")

# Show data table
st.subheader("Raw Data Preview")
st.dataframe(df.head())

# Plot humidity trend
st.subheader("Humidity Over Time")
st.line_chart(df["humidity"])
```

🔎 Explanation

- `st.title()` and `st.subheader()` → create headings.
- `st.dataframe()` → shows interactive tables.
- `st.line_chart()` → quickly plots time-series data.

👉 This creates a **first dashboard** showing mushroom sensor data.

Adding Predictions

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split

# Features and target
X = df[["temperature", "humidity", "co2", "voltage"]]
y = df["yield"]

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Train model
model = RandomForestRegressor()
model.fit(X_train, y_train)

# Predictions
df["predicted_yield"] = model.predict(X)

# Add to dashboard
st.subheader("Predicted Yield vs Actual")
st.line_chart(df[["yield", "predicted_yield"]])
```

🔎 **Explanation**

- Train a **Random Forest model** to predict yield.
- Add predictions as a new column.
- Display actual vs predicted yield trends.

Adding Prescriptions

```
# Simple decision rule
def prescribe_action(row):
    if row["humidity"] < 60:
        return "Increase misting"
    elif row["co2"] > 1200:
        return "Increase ventilation"
    else:
        return "Conditions OK"

df["recommendation"] = df.apply(prescribe_action, axis=1)

# Display in dashboard
st.subheader("Prescriptive Recommendations")
st.write(df[["timestamp", "humidity", "co2",
"recommendation"]].head(10))
```

🔎 **Explanation**

- Adds **decision rules** as prescriptive output.
- Shows recommended actions alongside sensor data.

Image Display (Optional Extension)

```
from PIL import Image

# Load example image
img = Image.open("mushroom_sample.jpg")
st.subheader("Mushroom Image Analysis")
st.image(img, caption="Detected Mushroom Sample",
use_column_width=True)
```

🔎 **Explanation**

- Displays mushroom images alongside sensor analytics.
- Can later integrate CNN predictions for disease/quality.

7.6 Advanced Features

- **What-if Analysis** → sliders to simulate different humidity or CO_2 levels.
- **Alerts** → red indicators if thresholds are breached.
- **Multi-page apps** → separate sections for EDA, prediction, prescription, and images.

7.7 Summary

- Dashboards present insights in **real-time and actionable form**.
- Streamlit/Dash make it easy to connect analytics code with visualisation.
- Mushroom dashboards integrate:
  - Sensor data (time series).
  - Predictions (yield, risks).
  - Prescriptions (decision rules, optimisation).
  - Image analytics (disease/quality).

7.8 Self-Assessment Questions

1. Why are dashboards important in analytics applications?
2. What are two differences between Streamlit and Dash?
3. How could you visualise both **actual yield** and **predicted yield** in a dashboard?
4. Give an example of a prescriptive rule that could be integrated into a mushroom dashboard.

Q1. Why are dashboards important in analytics applications?

Answer:

- **Rule:** If **humidity < 60%**, recommend: *"Turn on misting for 15 minutes."*
- **Rule:** If **CO_2 > 1200 ppm**, recommend: *"Increase ventilation by 20%."*
- **Rule:** If **voltage > threshold** and CO_2 is still rising, recommend: *"Check ventilation equipment efficiency."*

👉 These rules transform raw data into **clear actions** that improve cultivation conditions.

Q2. What are two differences between Streamlit and Dash?

Answer:

1. **Ease of Use:**
   - **Streamlit** → Very beginner-friendly; turn Python scripts into dashboards with minimal code.
   - **Dash** → More flexible but requires more setup (HTML/CSS-like callbacks).
2. **Customization:**
   - **Streamlit** → Great for rapid prototyping but with fewer customization options.
   - **Dash** → Offers advanced customization and supports complex, multi-page applications.

👉 Both can be used for mushroom analytics dashboards, but Streamlit is ideal for quick prototypes while Dash suits production-grade apps.

Q3. How could you visualise both actual yield and predicted yield in a dashboard?

Answer:

- Use a **line chart** or **bar chart** with two series:
  - One for **actual yield**.
  - One for **predicted yield**.
- Overlaying the two helps compare model performance against real outcomes.

👉 Example in Streamlit:

```
st.line_chart(df[["yield", "predicted_yield"]])
```

This lets farmers see whether the model is **accurately forecasting mushroom yield**.

Q4. Give an example of a prescriptive rule that could be integrated into a mushroom dashboard.

Answer:

- **Rule:** If **humidity < 60%**, recommend: *"Turn on misting for 15 minutes."*
- **Rule:** If **CO_2 > 1200 ppm**, recommend: *"Increase ventilation by 20%."*
- **Rule:** If **voltage > threshold** and CO_2 is still rising, recommend: *"Check ventilation equipment efficiency."*

👉 These rules transform raw data into **clear actions** that improve cultivation conditions.

7.9 References

- Streamlit Documentation → https://docs.streamlit.io/
- Dash Documentation → https://dash.plotly.com/
- Géron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly.

Topic 8: Project Development Sessions

8.1 Overview

Project development sessions are the **capstone stage** of this training. Participants form teams and work collaboratively to design and implement a **Mushroom Analytics System**. The system integrates:

- **Sensor data analytics** (temperature, humidity, CO_2, voltage, timestamp).
- **Predictive modeling** (yield forecasts, risk detection).
- **Prescriptive analytics** (decision rules, optimisation for actions).
- **Dashboard visualisation** (Streamlit/Dash interface).
- (Optional) **Image analysis** (quality grading, disease detection).

This phase emphasizes **collaboration, problem-solving, and creativity**, turning concepts into working prototypes.

This phase emphasizes **collaboration, problem-solving, and creativity**, turning concepts into working prototypes.

8.2 Learning Objectives

By the end of this topic, participants will be able to:

- Work in teams to design a **real-world analytics pipeline**.
- Integrate multiple data sources (sensor logs, categorical data, images).
- Develop predictive and prescriptive components into a **unified solution**.
- Build a **user-facing dashboard** that presents insights and recommendations.
- Present their project findings effectively through a live demo.

8.3 Project Scope

Each team should build a **Mushroom Analytics Application** that:

1. **Ingests and preprocesses data** → sensor logs + categorical attributes.
2. **Runs predictive models** → yield prediction, risk classification.
3. **Implements prescriptive logic** → recommendations based on decision rules or optimization.
4. **Displays results** → dashboard with sensor monitoring, model outputs, and recommended actions.
5. **(Optional Extension)** → Incorporates image-based analytics for disease/quality detection.

8.4 Suggested Workflow

Phase 1 – Project Kick-off

- Define **problem scope** (e.g., "Optimize mushroom yield under limited electricity").
- Assign team roles (data engineer, model builder, dashboard developer, presenter).
- Draft **system architecture diagram**.

Phase 2 – Data Preparation

- Load mushroom datasets (UCI categorical + Kaggle growth + optional images).
- Clean and preprocess (handle missing values, encode categorical variables).
- Perform feature engineering (rolling averages, lags, correlations).

Phase 3 – Modeling

- Choose appropriate models:
  - **Classification** → edible vs poisonous, risk categories.
  - **Regression** → yield prediction from sensor data.
- Train and evaluate using metrics: Accuracy, Recall, RMSE.

Phase 4 – Prescription

- Define **decision rules** (if humidity < threshold, then mist).
- Implement **optimization** with PuLP or `scipy.optimize`.
- Integrate prescriptions into pipeline output.

Phase 5 – Dashboard Development

- Build a **Streamlit or Dash app** with:
  - Current sensor readings.
  - Trend charts (temperature, humidity, CO_2).
  - Model predictions (yield, risk).
  - Prescriptive recommendations (actions).
  - Optional image classification panel.

Phase 6 – Presentation Preparation

- Prepare **slides** explaining:
  - Dataset overview.
  - Analytics pipeline (diagram).
  - Modeling results (metrics).
  - Dashboard demo.
  - Prescriptions & recommendations.
- Rehearse a **10-minute live demo**.

8.5 Example Project Ideas

1. **Smart Ventilation Controller**
   - Predict CO_2 build-up.
   - Prescribe ventilation cycles under energy budget.
2. **Humidity Optimization for Yield**
   - Predict yield from humidity and temperature trends.
   - Prescribe misting durations to maximize growth.
3. **Mushroom Quality Grader**
   - Use images to classify mushrooms as healthy vs diseased.
   - Combine with sensor data to issue contamination alerts.

8.6 Summary

- Project sessions consolidate skills from earlier topics.
- Teams build **end-to-end mushroom analytics systems**.
- Deliverables include: ETL pipeline, models, prescriptions, dashboard, and demo.
- Emphasis is on **collaboration, problem-solving, and real-world application**.

Additional

🍄 Grafana for Mushroom Cultivation — End-to-End Plan

Plotly Dash

🍄 Grafana for Mushroom Cultivation — End-to-End Plan

🍄 Grafana for Mushroom Cultivation — End-to-End Plan

1) Reference architecture

- **Data Ingest**:
  - From CSV (historical) and/or MQTT (live sensors) → **Telegraf**
- **TSDB** (choose one):
  - **InfluxDB v2** (Flux): easiest with Telegraf, strong downsampling
  - TimescaleDB (SQL/Postgres): if you want SQL + joins
  - Prometheus: great for pull-based metrics; less ideal for CSV history
- **Dashboards & Alerts**: **Grafana**
- **(Optional)** Logs: **Loki** for controller/app logs

> Recommended starter: **InfluxDB v2 + Telegraf + Grafana**.

---

2) Data model (Influx line protocol)

Treat each reading as a point:

Measurement: `mushroom_env`
Tags (low cardinality dimensions): `farm_id`, `room_id`, `strain`, `sensor_id`
Fields (metrics): `temperature_C`, `humidity_percent`, `co2_ppm`, `voltage_V`

Example line protocol:

```
mushroom_env,farm_id=F01,room_id=R2,strain=Oyster,sensor_id=S001
temperature_C=24.8,humidity_percent=88.5,co2_ppm=1050,voltage_V=4.12
1735815600000000000
```

(Last value is Unix ns timestamp.)

Cardinality tips: keep tags few & stable (farm/room/strain); avoid high-cardinality tags (like timestamps or unique image filenames).

---

3) InfluxDB buckets & retention

- **hot bucket**: `mushroom_hot` (retention: 30d)
- **cold bucket**: `mushroom_cold` (retention: 365d)
- **Downsampling task** (InfluxDB task):

```
option task = {name: "downsample_5m", every: 5m}
from(bucket: "mushroom_hot")
  |> range(start: -1h)
  |> filter(fn: (r) => r._measurement == "mushroom_env")
  |> aggregateWindow(every: 5m, fn: mean, createEmpty: false)
  |> to(bucket: "mushroom_cold", org: "your_org")
```

---

4) Telegraf ingestion

a) From CSV (bootstrapping historical data)

`telegraf.conf` (snippet):

```
[[inputs.file]]
  files = ["/data/mushroom_dummy.csv"]
  data_format = "csv"
  csv_header_row_count = 1
  csv_timestamp_column = "timestamp"
  csv_timestamp_format = "2006-01-02T15:04:05Z07:00"
  csv_tag_columns = ["farm_id","room_id","strain","sensor_id"]
  csv_skip_rows = 0
  name_override = "mushroom_env"

  # Map CSV columns to fields via processors
[[processors.converter]]
  [processors.converter.fields]
    float = ["temperature_C","humidity_percent","co2_ppm","voltage_V"]

[[outputs.influxdb_v2]]
  urls = ["http://influxdb:8086"]
  token = "$INFLUX_TOKEN"
  organization = "your_org"
  bucket = "mushroom_hot"
```

> If your CSV doesn't have tag columns yet, skip `csv_tag_columns` or add defaults using `processors.enum`.

b) From MQTT (live sensors)

```
[[inputs.mqtt_consumer]]
  servers = ["tcp://mqtt-broker:1883"]
  topics = ["mushroom/+/+/env"] # e.g., mushroom/F01/R2/env
  data_format = "json"
  json_time_key = "timestamp"
  json_time_format = "unix_ms"
  json_name_key = "measurement"
  tag_keys = ["farm_id","room_id","strain","sensor_id"]

[[outputs.influxdb_v2]]
  urls = ["http://influxdb:8086"]
  token = "$INFLUX_TOKEN"
  organization = "your_org"
  bucket = "mushroom_hot"
```

---

5) Docker Compose (InfluxDB + Telegraf + Grafana)

Save as `docker-compose.yml`:

```yaml
services:
  influxdb:
    image: influxdb:2
    ports: ["8086:8086"]
    environment:
      - DOCKER_INFLUXDB_INIT_MODE=setup
      - DOCKER_INFLUXDB_INIT_USERNAME=admin
      - DOCKER_INFLUXDB_INIT_PASSWORD=adminpass
      - DOCKER_INFLUXDB_INIT_ORG=your_org
      - DOCKER_INFLUXDB_INIT_BUCKET=mushroom_hot
      - DOCKER_INFLUXDB_INIT_RETENTION=30d
      - DOCKER_INFLUXDB_INIT_ADMIN_TOKEN=supersecrettoken
    volumes:
      - influx-data:/var/lib/influxdb2

  telegraf:
    image: telegraf:1.30
    depends_on: [influxdb]
    environment:
      - INFLUX_TOKEN=supersecrettoken
    volumes:
      - ./telegraf.conf:/etc/telegraf/telegraf.conf:ro
      - ./data:/data

  grafana:
    image: grafana/grafana:10.4.2
    ports: ["3000:3000"]
    environment:
      - GF_SECURITY_ADMIN_USER=admin
      - GF_SECURITY_ADMIN_PASSWORD=adminpass
      - GF_SERVER_DOMAIN=localhost
    volumes:
      - grafana-data:/var/lib/grafana

volumes:
  influx-data:
  grafana-data:
```

6) Grafana data source (InfluxDB v2)

In Grafana → **Connections → Data sources → Add data source → InfluxDB**:

- Query language: **Flux**
- URL: `http://influxdb:8086`
- Org: `your_org`
- Token: `supersecrettoken`
- Default bucket: `mushroom_hot` (you can switch per dashboard/variable)

---

7) Dashboard design (panels you'll want)

A. Environment Overview

- **Time series**: Temp, Humidity, CO_2 (overlay) with **threshold bands**:
  - Temp safe zone 24–27°C (shaded)
  - Humidity safe 85–90%
  - CO_2 optimal 800–1200 ppm
- **Stats**: Last value + 6h/24h deltas
- **Gauge**: Current CO_2 and humidity vs target

Flux example (smoothed temp):

```
from(bucket: v.bucket)
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn:(r)=> r._measurement == "mushroom_env" and r._field ==
"temperature_C")
  |> aggregateWindow(every: 10m, fn: mean)
  |> yield(name: "mean_10m")
```

B. Quality & Distribution

- **Histogram** panels: Temperature, Humidity
- **Box plot**: CO_2 by room (requires Grafana box plot panel)
- **Heatmap**: Hour-of-day vs humidity (visual humidity rhythm)
  - Create a Flux column `hour = date.hour(t: r._time)` then aggregate

Flux for hourly heatmap:

```
from(bucket: v.bucket)
  |> range(start: v.timeRangeStart)
  |> filter(fn:(r)=> r._measurement=="mushroom_env" and
r._field=="humidity_percent")
  |> map(fn:(r)=> ({ r with hour: int(v: date.hour(t: r._time)) }))
  |> group(columns:["hour"])
  |> mean()
```

C. Correlations & Relationships

- **Scatter plot**: Temp vs Humidity, color by CO_2
  Use Grafana's **Scatter plot** panel (plugin if not built-in).
- **Table**: Pearson correlations (precompute via Flux or write a small service to push a correlation series).

Flux quick z-score filter (outlier marking):

```
data = from(bucket: v.bucket)
  |> range(start: v.timeRangeStart)
  |> filter(fn:(r)=> r._measurement=="mushroom_env" and
r._field=="co2_ppm")
stats = data |> mean(column:"_value") |> set(key:"stat", value:"mean")
```

*(For full z-score, easier to compute upstream; in Grafana use **Transformations → Add field from calculation** or precompute in Influx tasks.)*

D. Forecasts

- Preferred: train **Prophet/LSTM** externally → write predicted series back into Influx:
  - Write to `mushroom_forecast` measurement with fields `temperature_pred`, `co2_pred`, etc.
  - Plot **Actual vs Forecast** with 95% bands (store `*_lower`, `*_upper` as separate fields).

---

8) Variables & templating

Create dashboard variables for exploration:

- `$farm` (query tags)
- `$room`
- `$strain`
- `$sensor`
  Use them in panel queries and in titles. Add a **Time range** picker.

Flux to list rooms:

```
import "influxdata/influxdb/schema"
schema.tagValues(bucket: v.bucket, tag: "room_id", predicate: (r) =>
r._measurement == "mushroom_env", start: -30d)
```

---

9) Unified alerting (Grafana v9+)

Example alert rule: "CO_2 > 1800 ppm for 10 minutes in any room"

- Query A (Flux):

```
from(bucket: "mushroom_hot")
  |> range(start: -15m)
  |> filter(fn:(r)=> r._measurement=="mushroom_env" and
r._field=="co2_ppm")
  |> aggregateWindow(every: 1m, fn: mean)
  |> filter(fn:(r)=> r._value > 1800)
```

- Condition: **WHEN A is above 0 for 10m**
- Labels: `severity=high`, `room={{room_id}}`
- Contact points: Email/Slack/Telegram
- **Silences**: Set quiet hours for harvesting windows (timezone: **Asia/Kuala_Lumpur**)

Another rule: “Humidity outside 85–90% for 15 minutes”
Use two expressions or a **math** expression to check `(value < 85) OR (value > 90)`.

---

10) Annotations & events

- Create **annotation streams** for:
  - **Harvest** events
  - **Fan/vent changes**
  - **Misting** cycles
- You can write these as Influx points to `mushroom_events` and display as annotations on time series panels.

---

11) Images in Grafana

- Install **Image panel** plugin (or use **Text** panel with Markdown).
- Host your images (e.g., `/images/<timestamp>.jpg` or object storage).
- Link an image URL template using variables, e.g.,
  `https://yourhost/images/${__from:date:YYYY-MM-DD_HH-mm}.jpg`
  Or display a small gallery alongside stats (manual mapping if needed).

---

12) Security & multi-tenant tips

- Use **Grafana folders** + **roles** per farm/customer.
- Influx: issue **per-dashboard tokens** with restricted bucket scopes.
- Avoid tag explosion; watch series cardinality (keep tags compact).

---

13) Ops & maintenance

- **Backups**: snapshot Influx DB dir; export Grafana dashboards as JSON.
- **Provisioning**: store Grafana data sources & dashboards as code in `/etc/grafana/provisioning`.
- **Downsampling**: use tasks to keep hot data small; query cold for long ranges.
- **Health**: add a dashboard for Telegraf agent health and MQTT lag.

---

14) Bringing forecasts into Grafana (simple pattern)

1. Train Prophet/LSTM in a Python job (cron).
2. Write predictions back to Influx as:

```
mushroom_forecast,farm_id=F01,room_id=R2
temperature_pred=25.6,temperature_lower=24.3,temperature_upper=26.9
1735822800000000000
```

3. In Grafana: line panel with actual temp vs pred + shaded CI band (two additional series).

---

15) Ready-to-import dashboard (skeleton JSON)

Create a new dashboard and paste this minimal **panel JSON** into a panel's "Edit JSON" to fast-start a Temp chart:

```
{
  "type": "timeseries",
  "title": "Temperature (°C)",
  "fieldConfig": { "defaults": { "unit": "celsius" } },
  "targets": [
    {
      "datasource": { "type": "influxdb", "uid": "__DS__InfluxDB" },
      "query": "from(bucket: v.bucket)\n  |> range(start:
v.timeRangeStart, stop: v.timeRangeStop)\n  |> filter(fn:(r)=>
r._measurement==\"mushroom_env\" and r._field==\"temperature_C\")\n  |>
aggregateWindow(every: 10m, fn: mean, createEmpty: false)"
    }
  ]
}
```