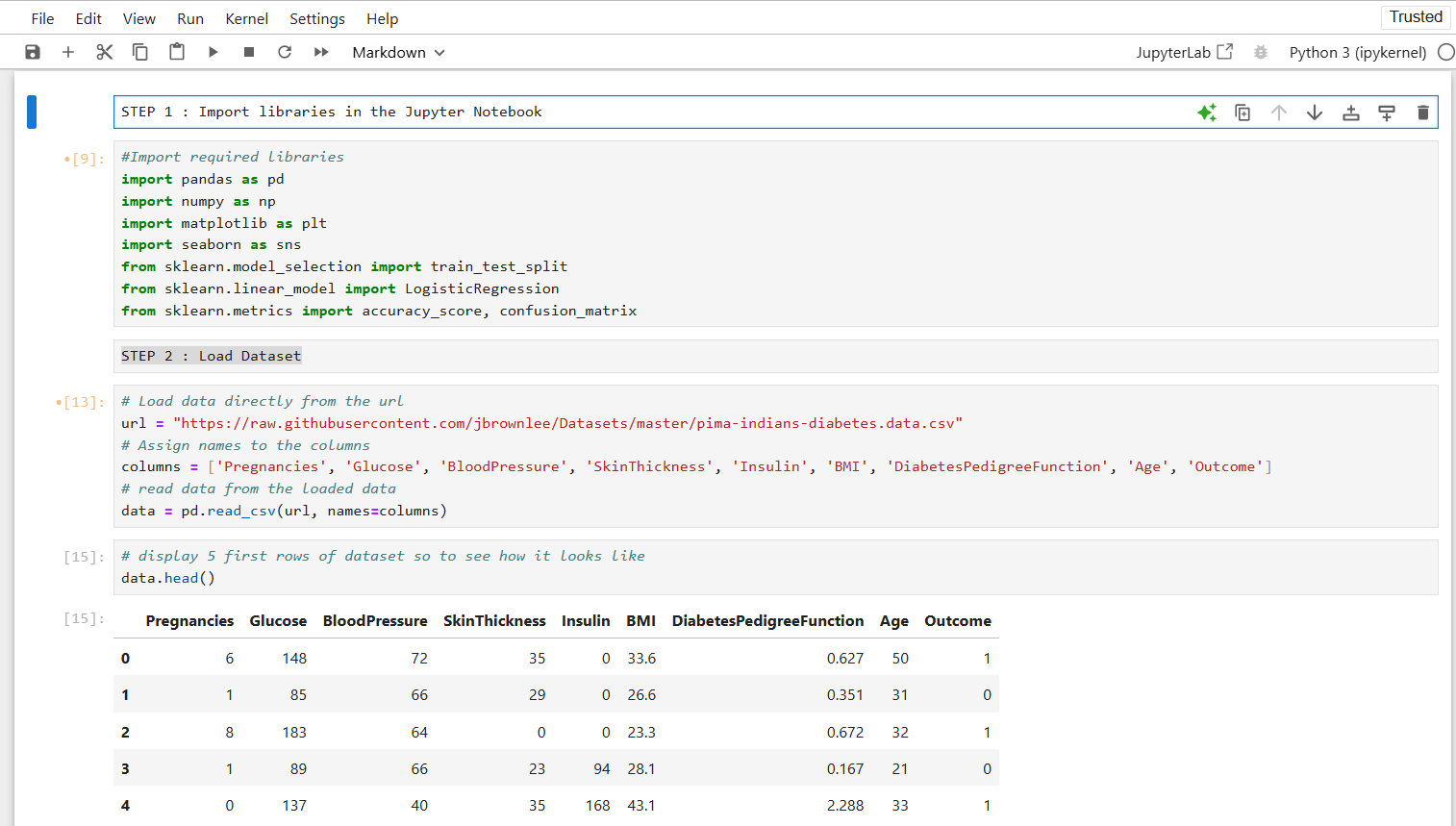
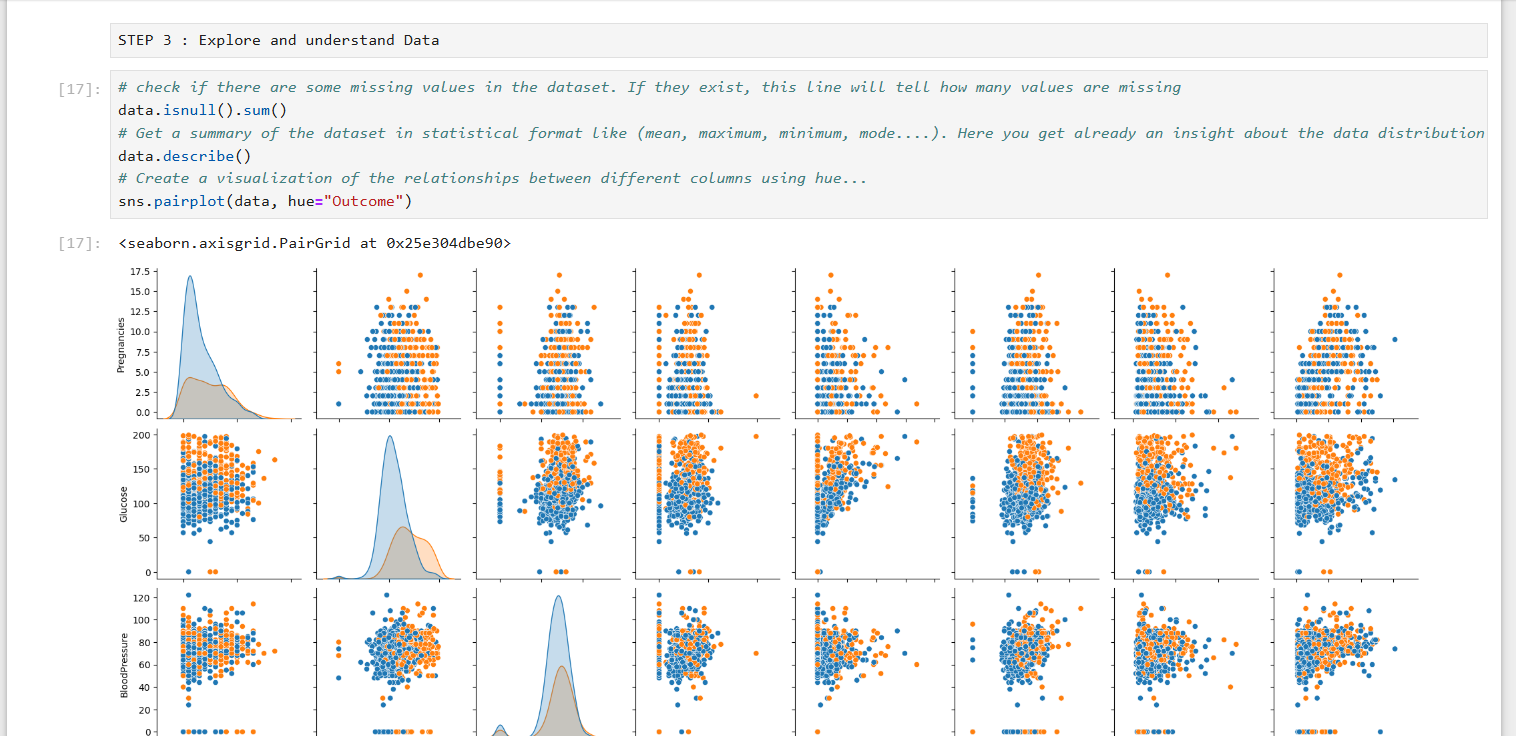
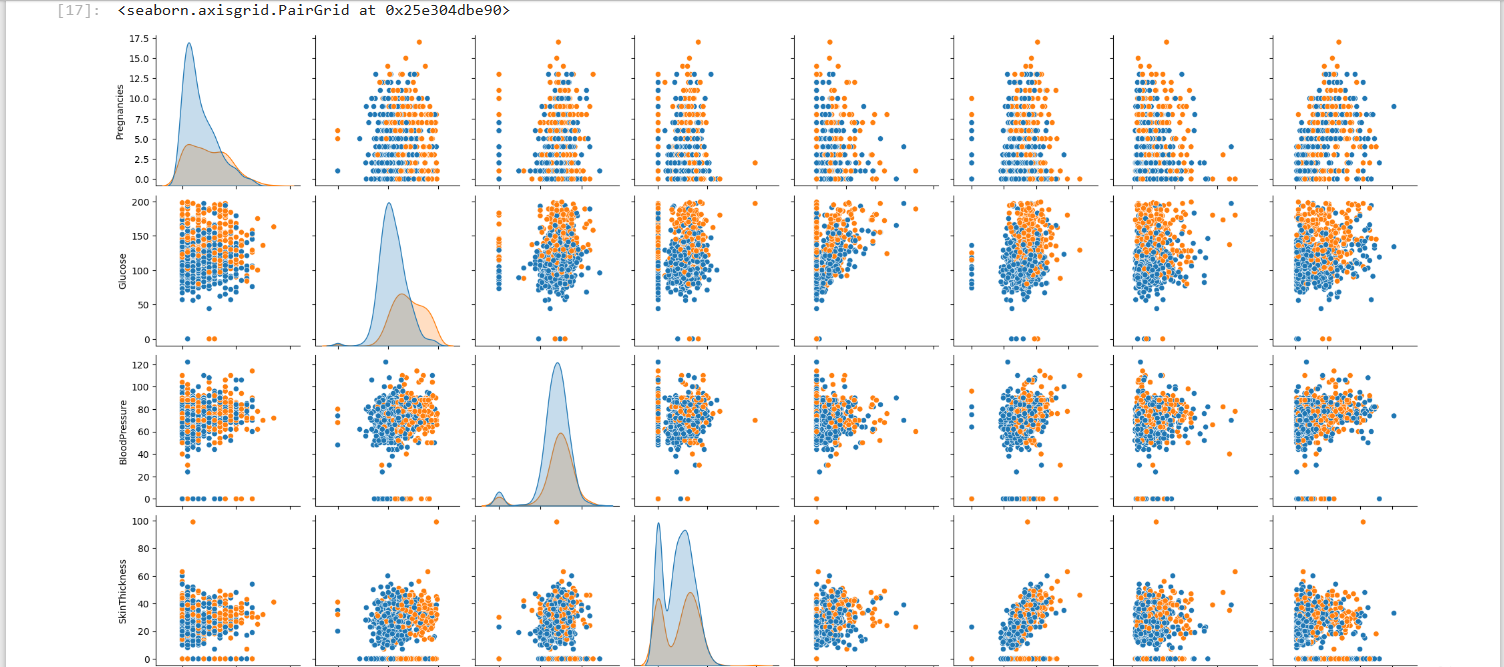
PERSONAL LABS

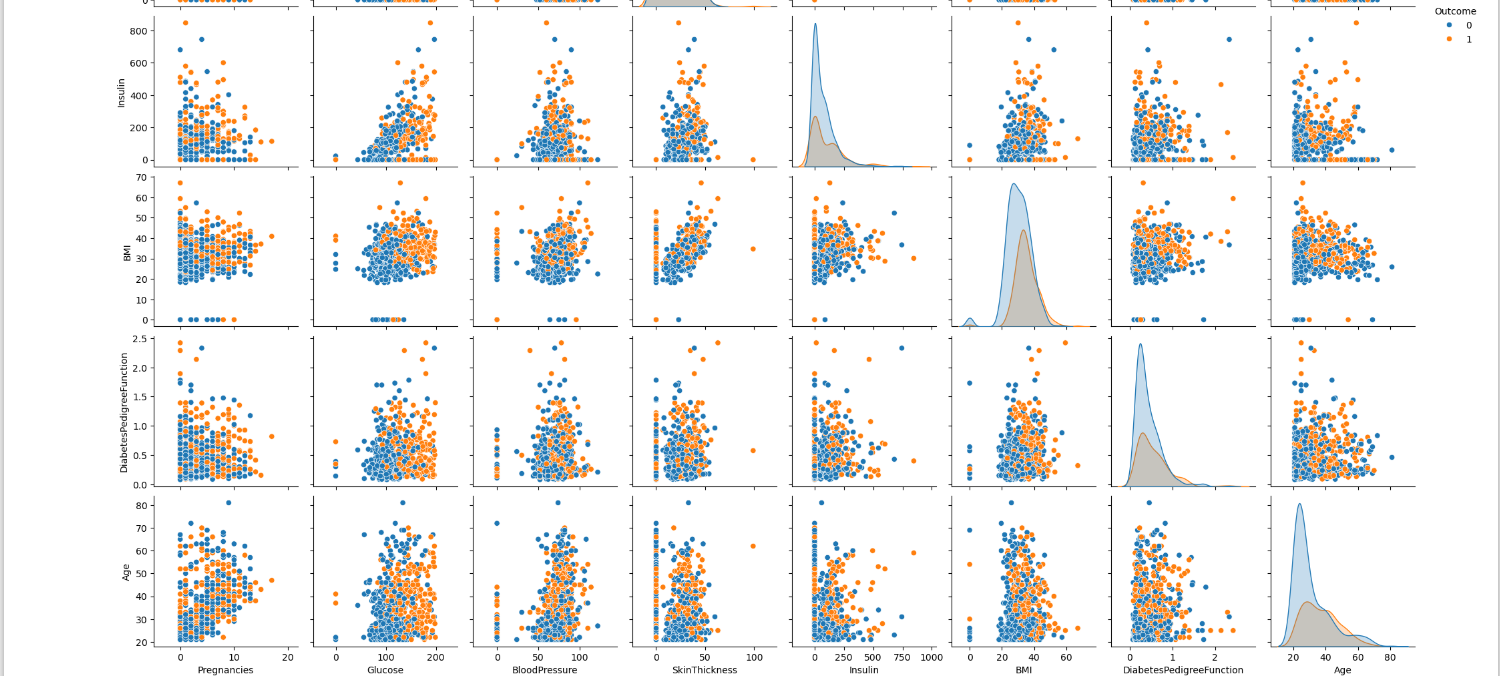
1. Simple ML project model for prediction

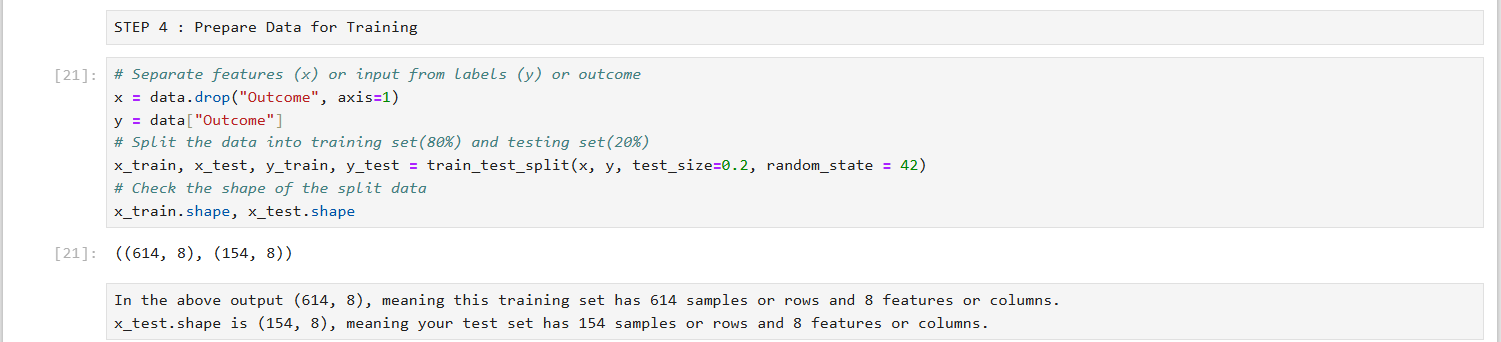
Build a basic ML model that predicts whether a person has diabetes or not based on some health data. For this project, use the **Pima Indians Diabetes Dataset** (from Kaggle or available in sklearn datasets).

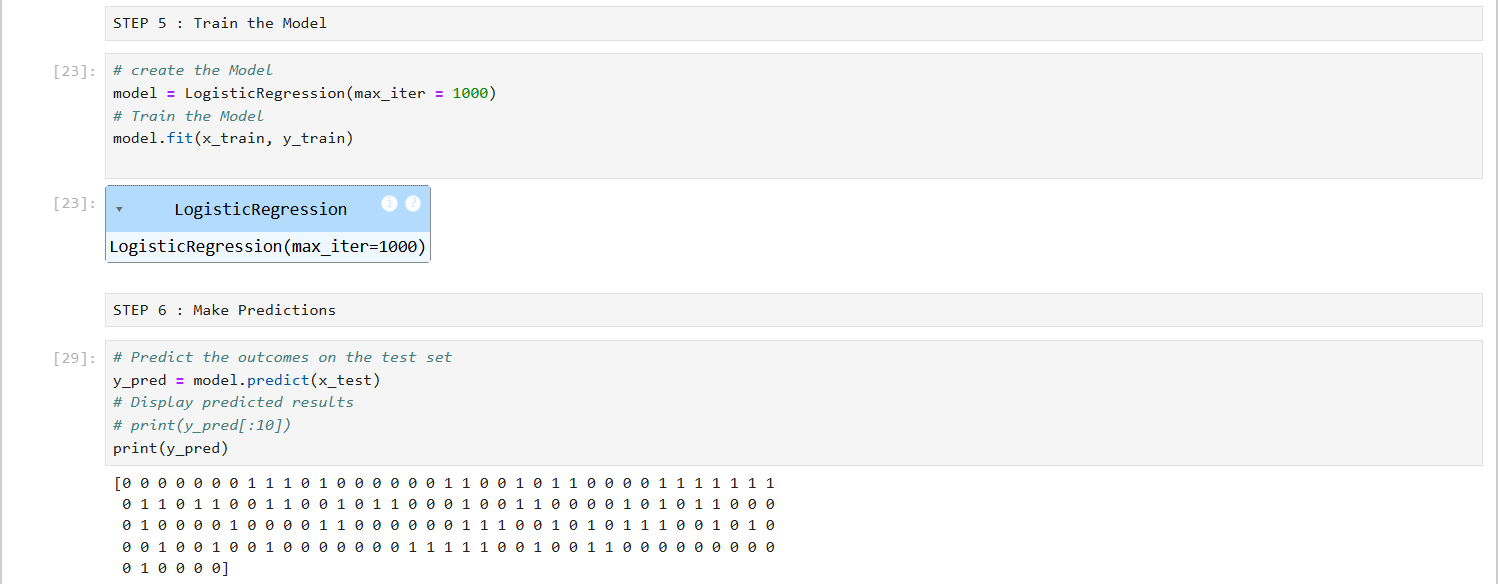




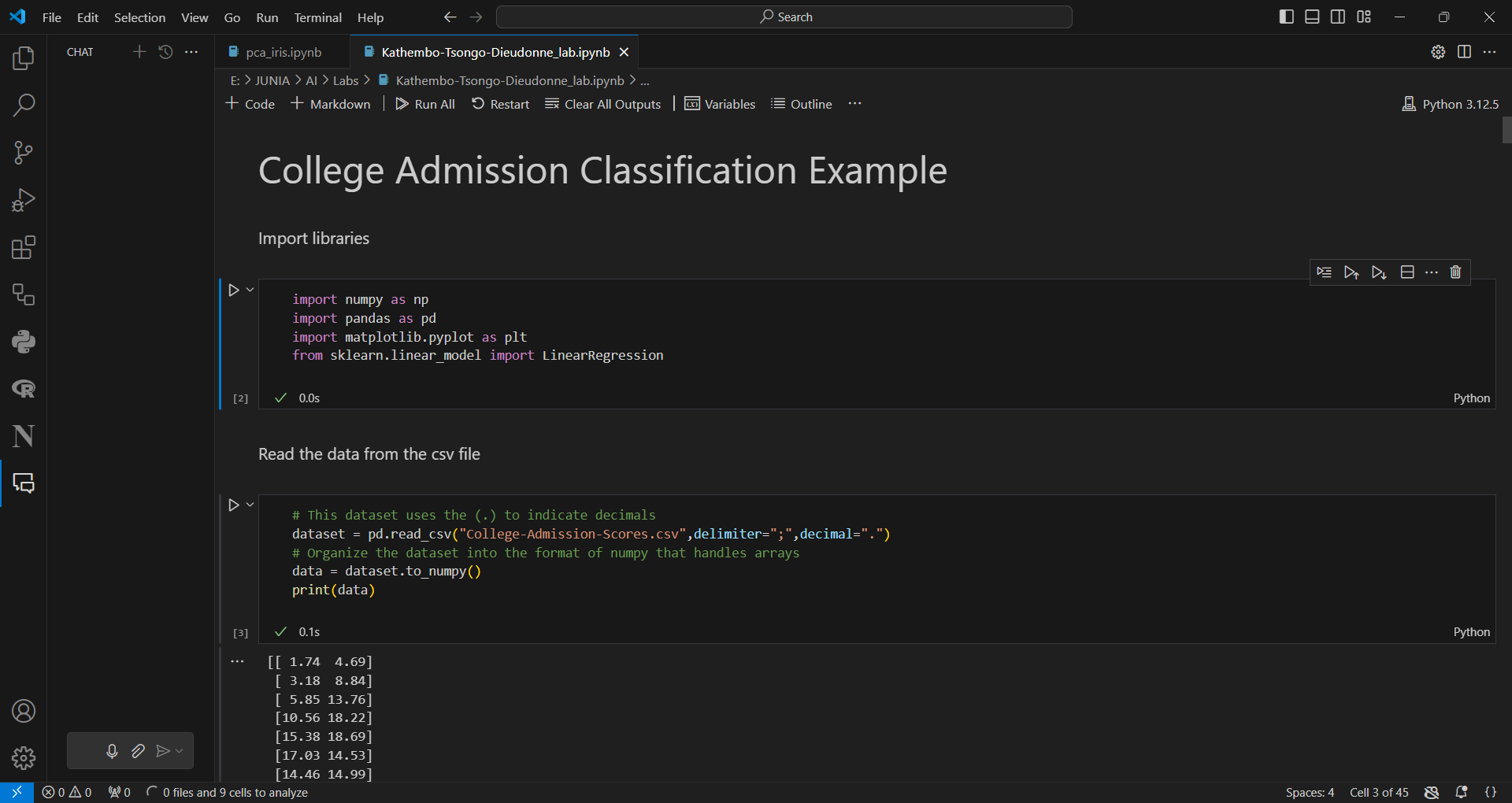


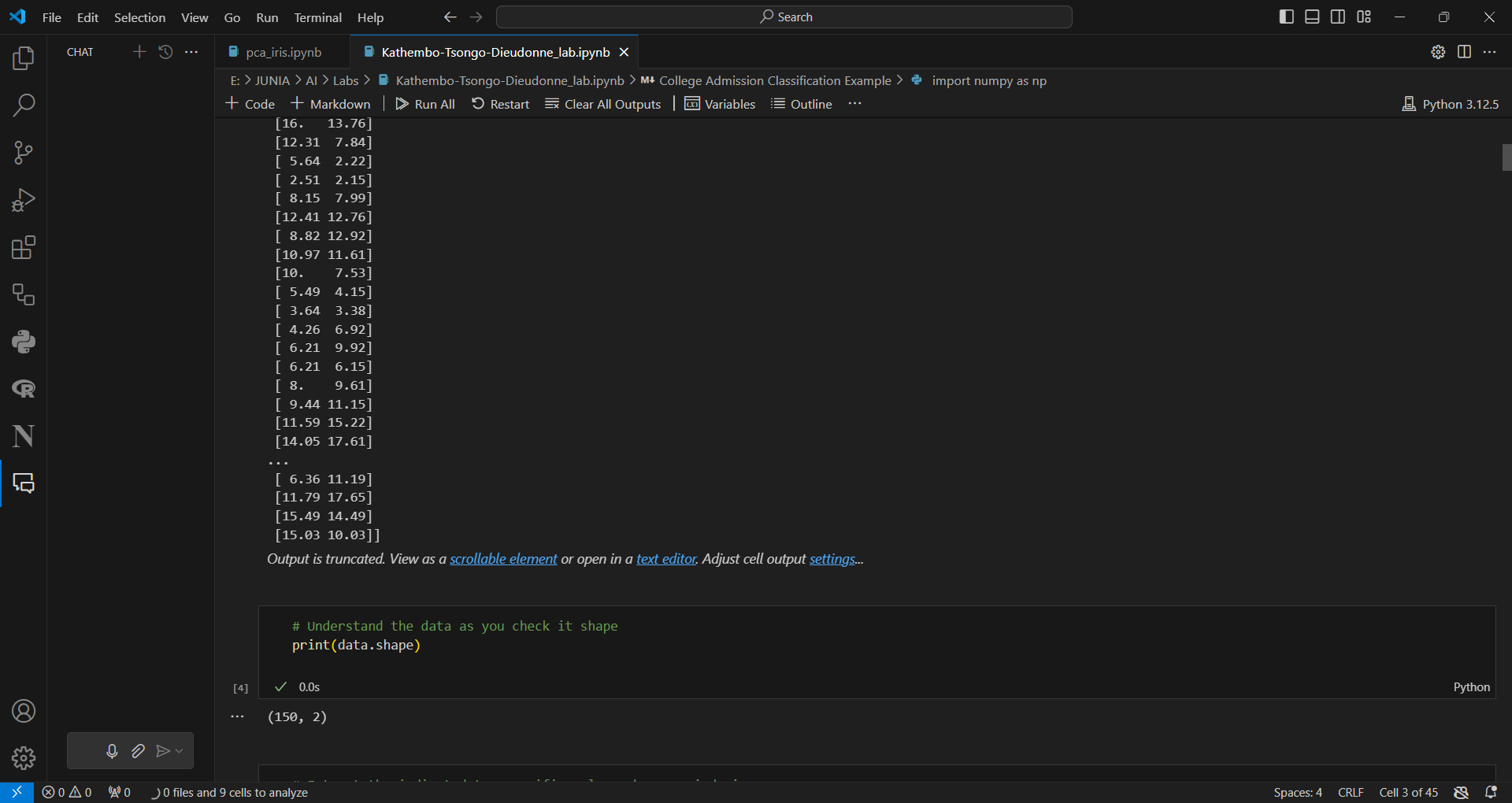


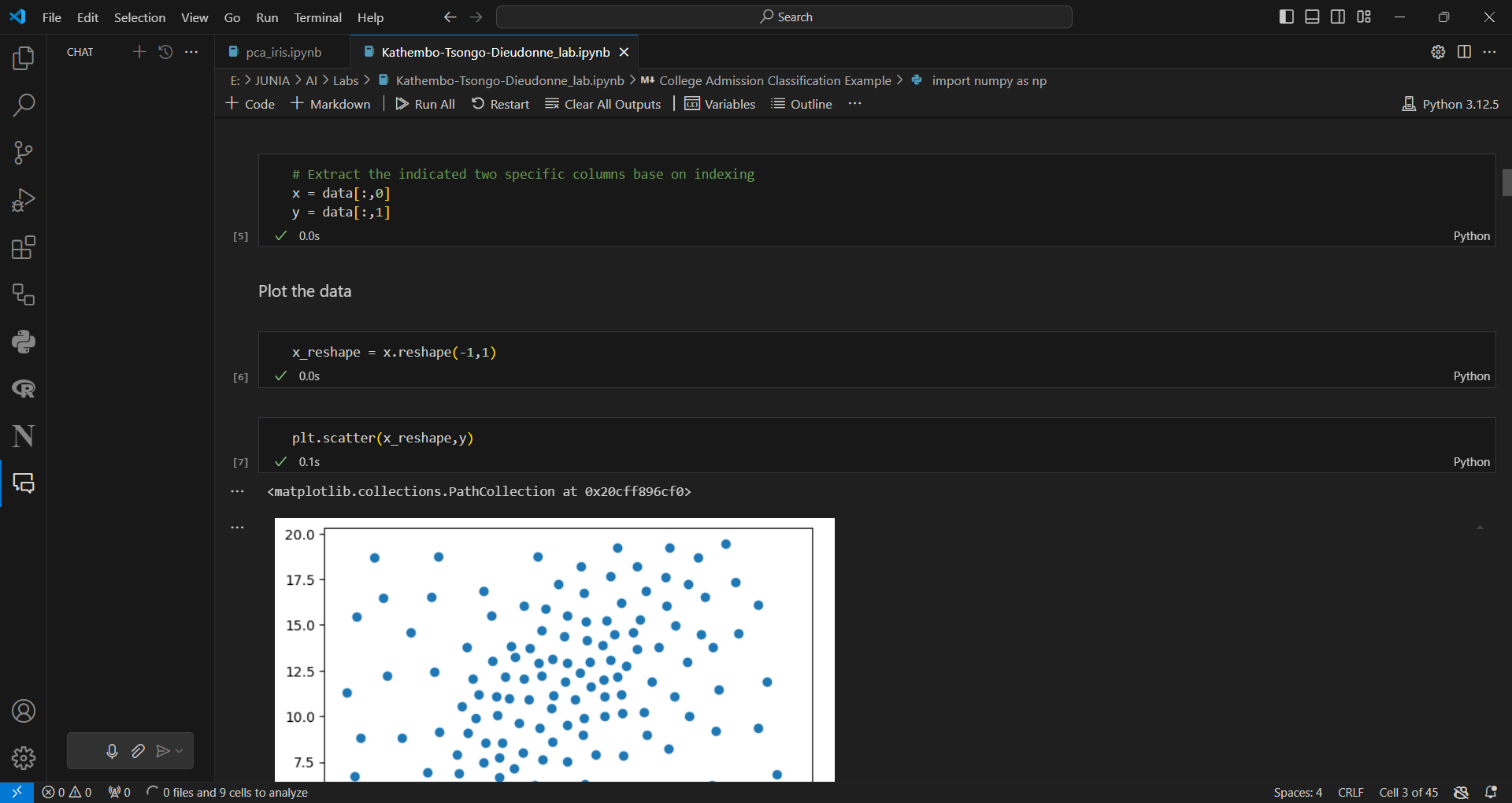


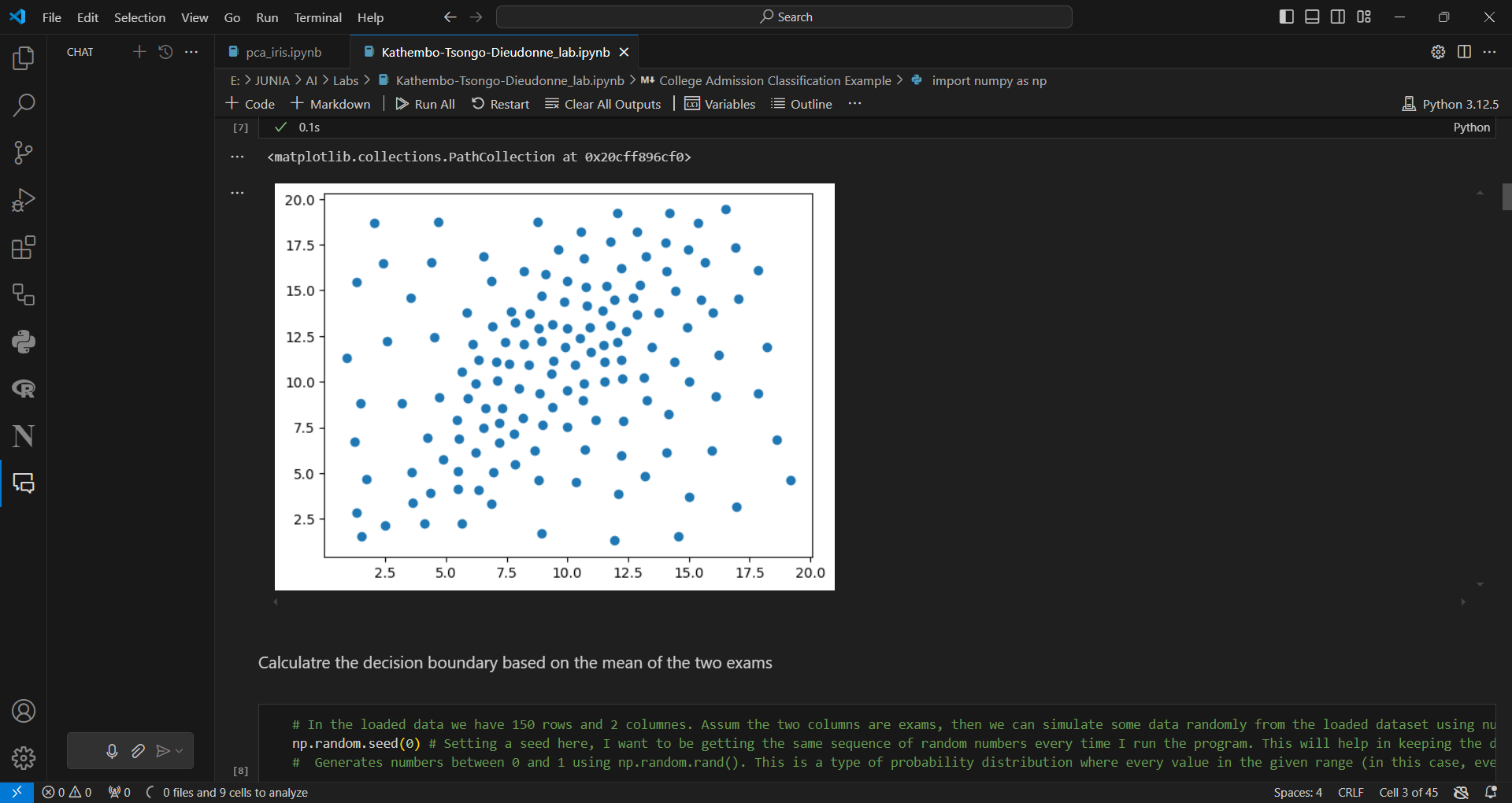


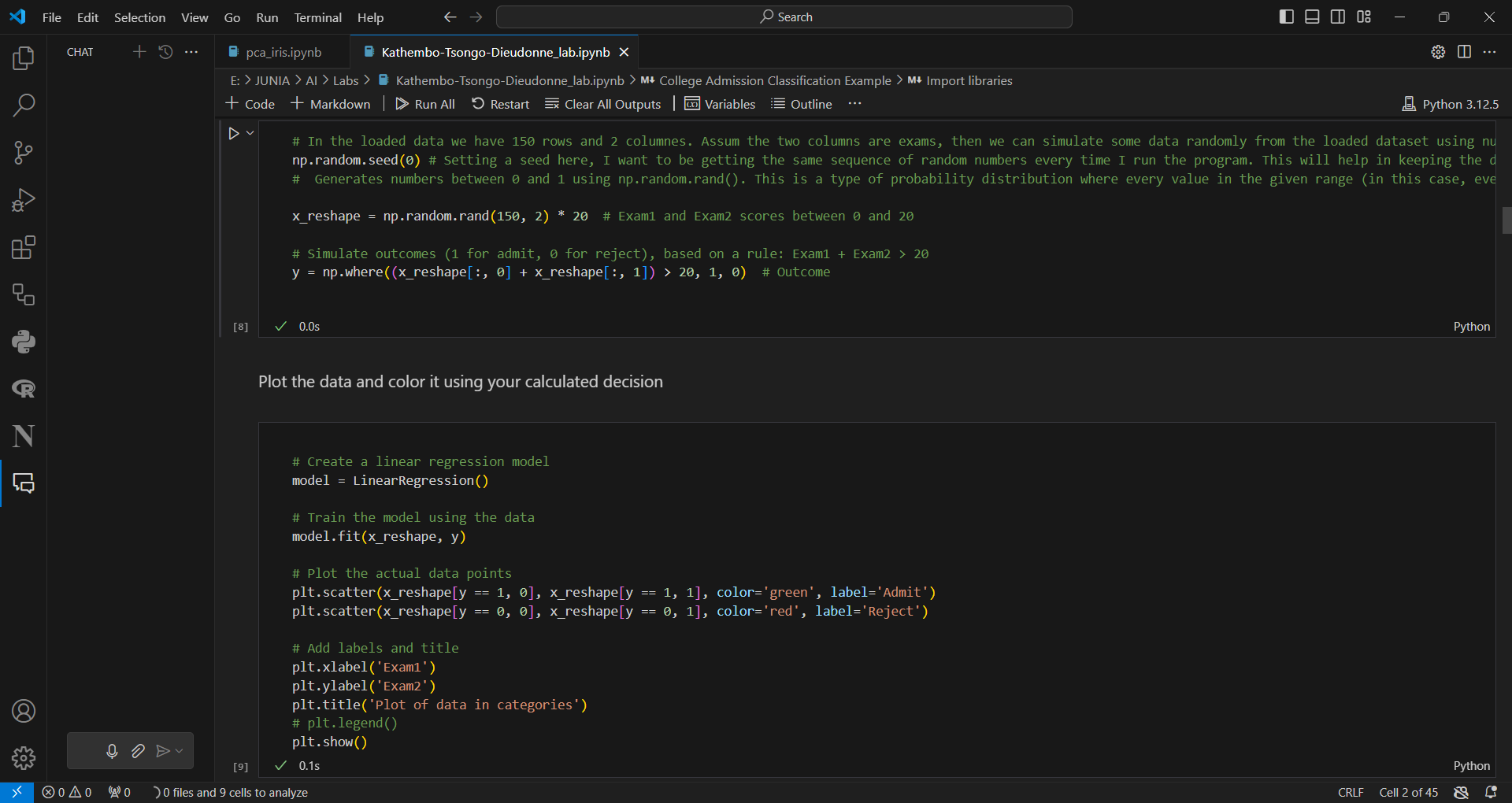
LAB 2



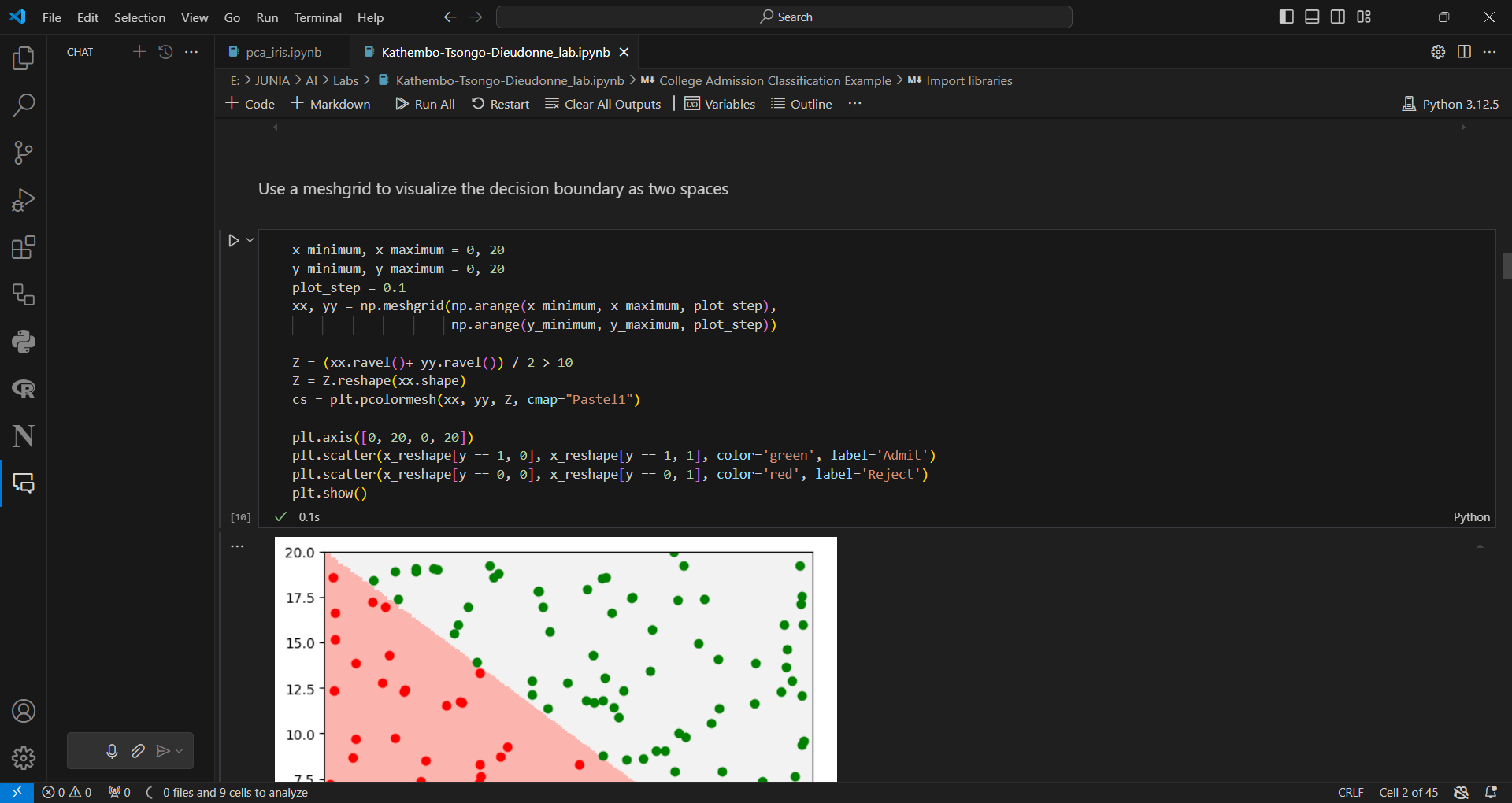


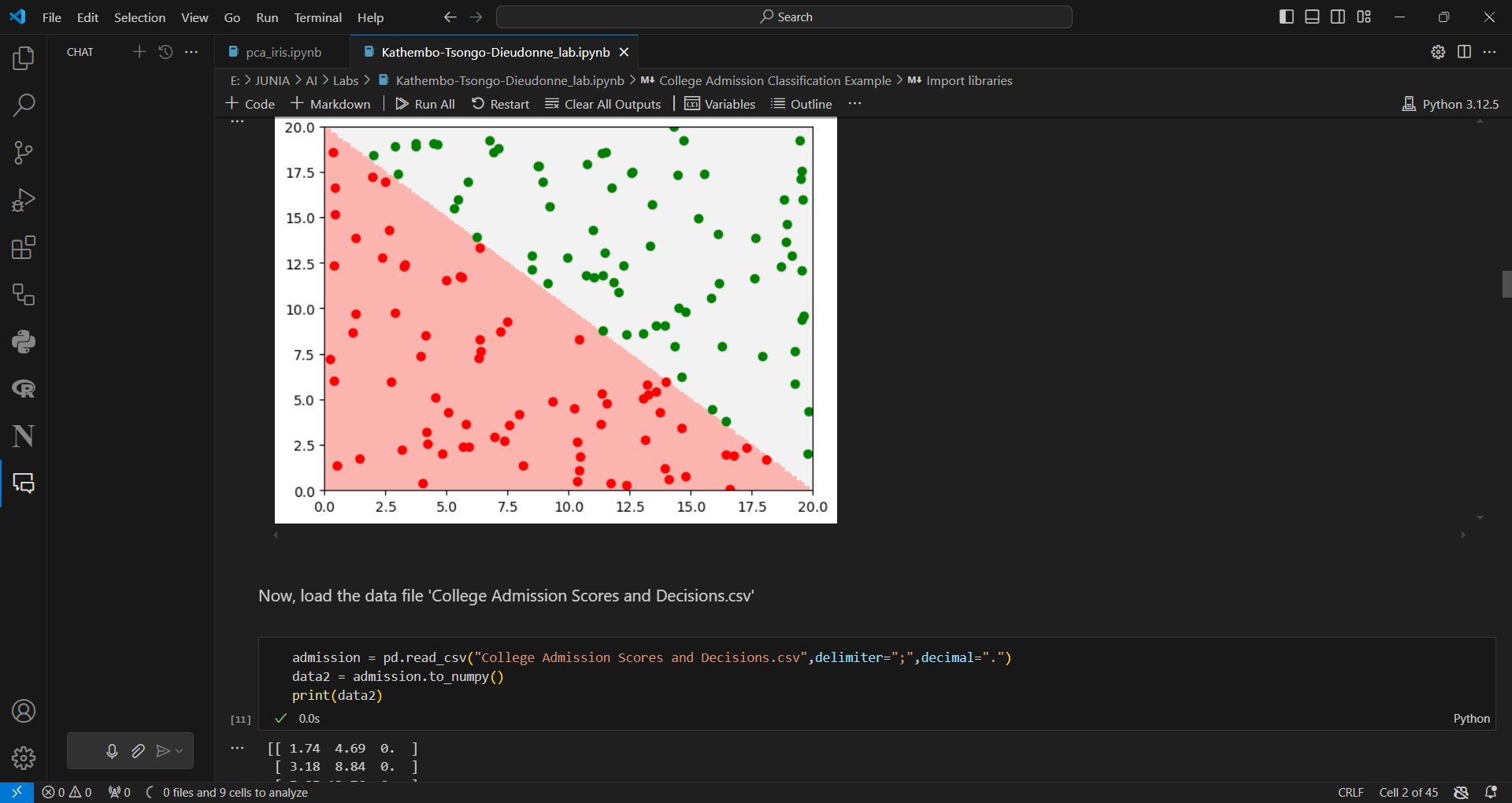


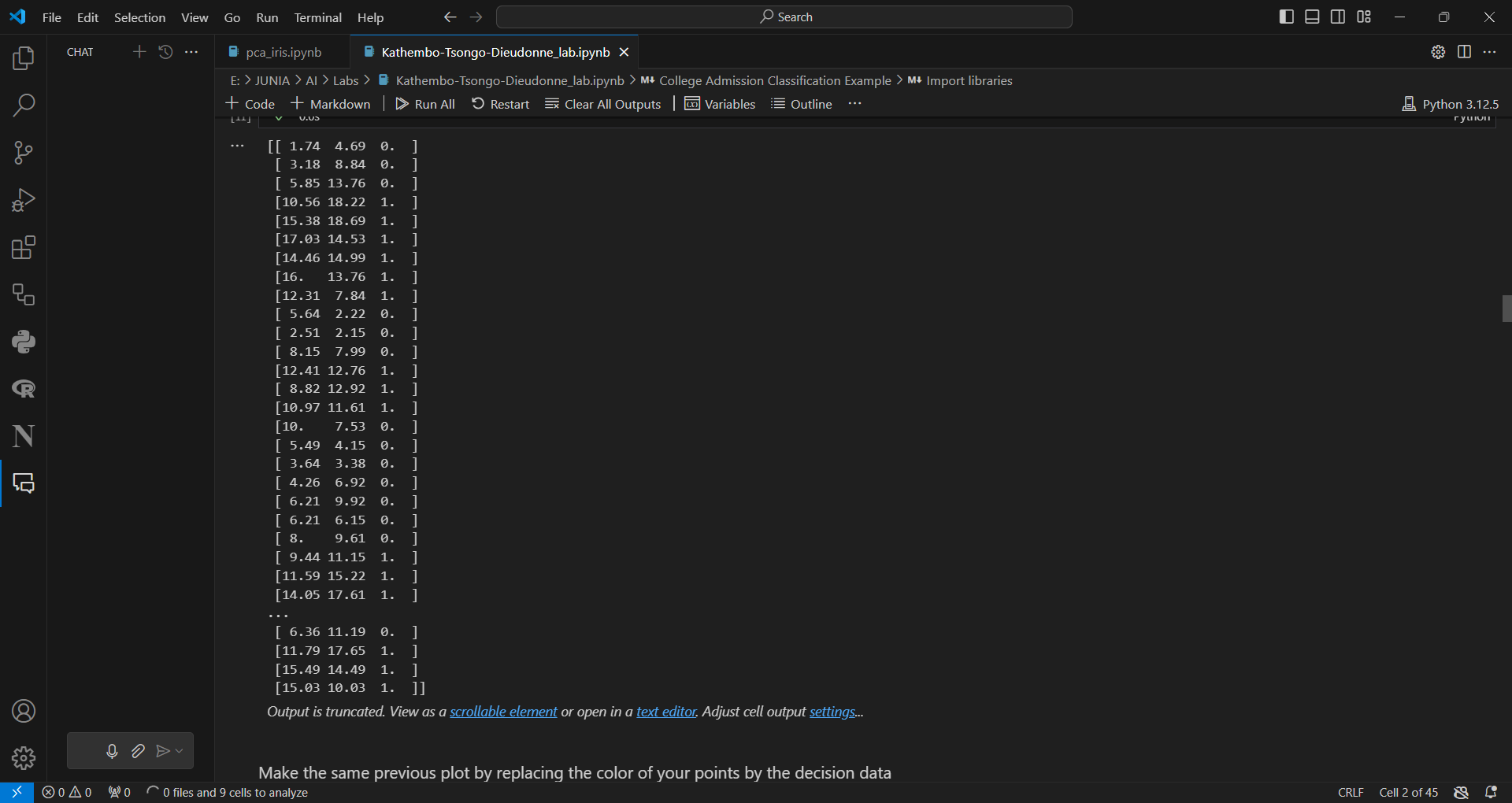




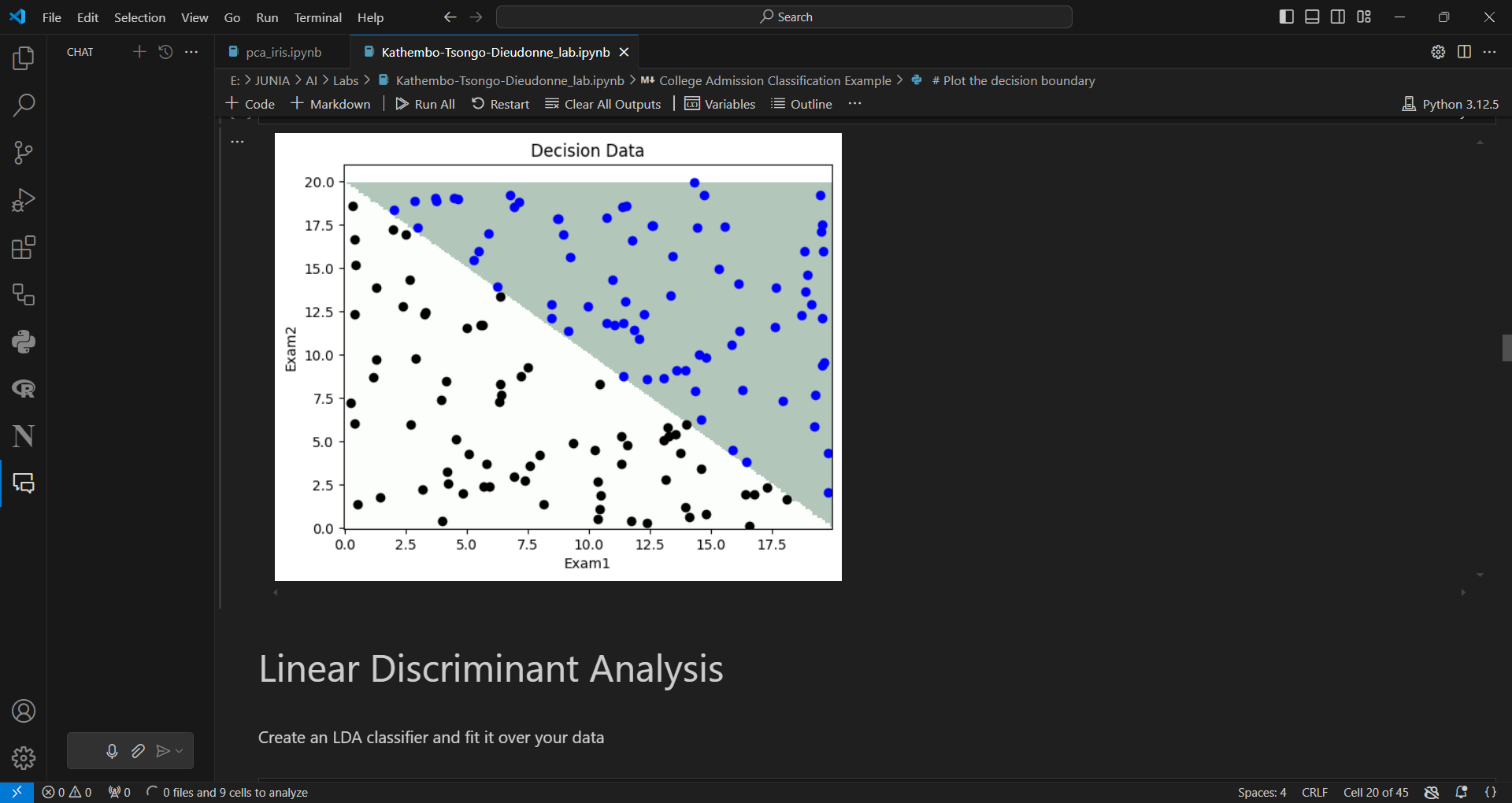


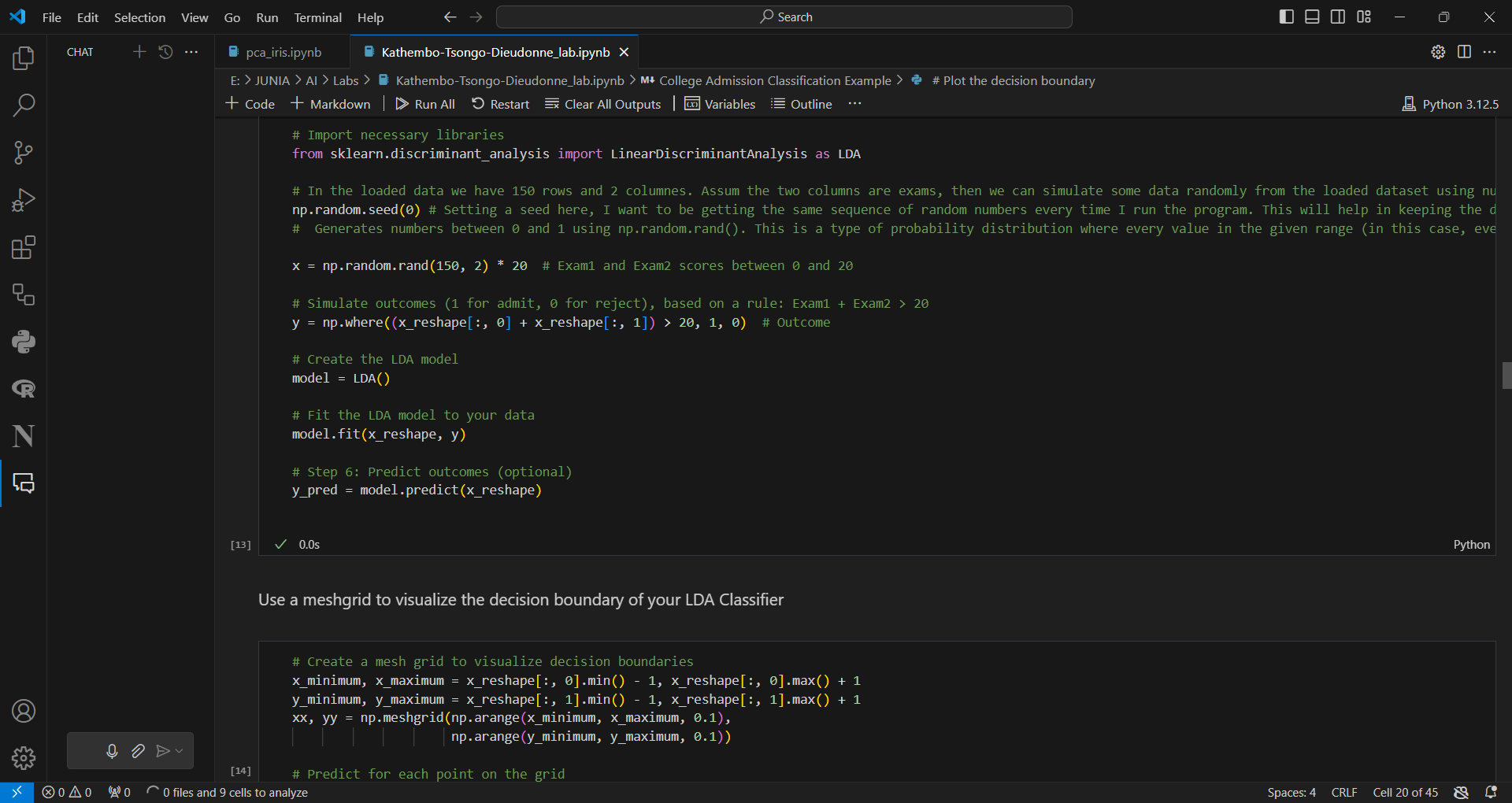


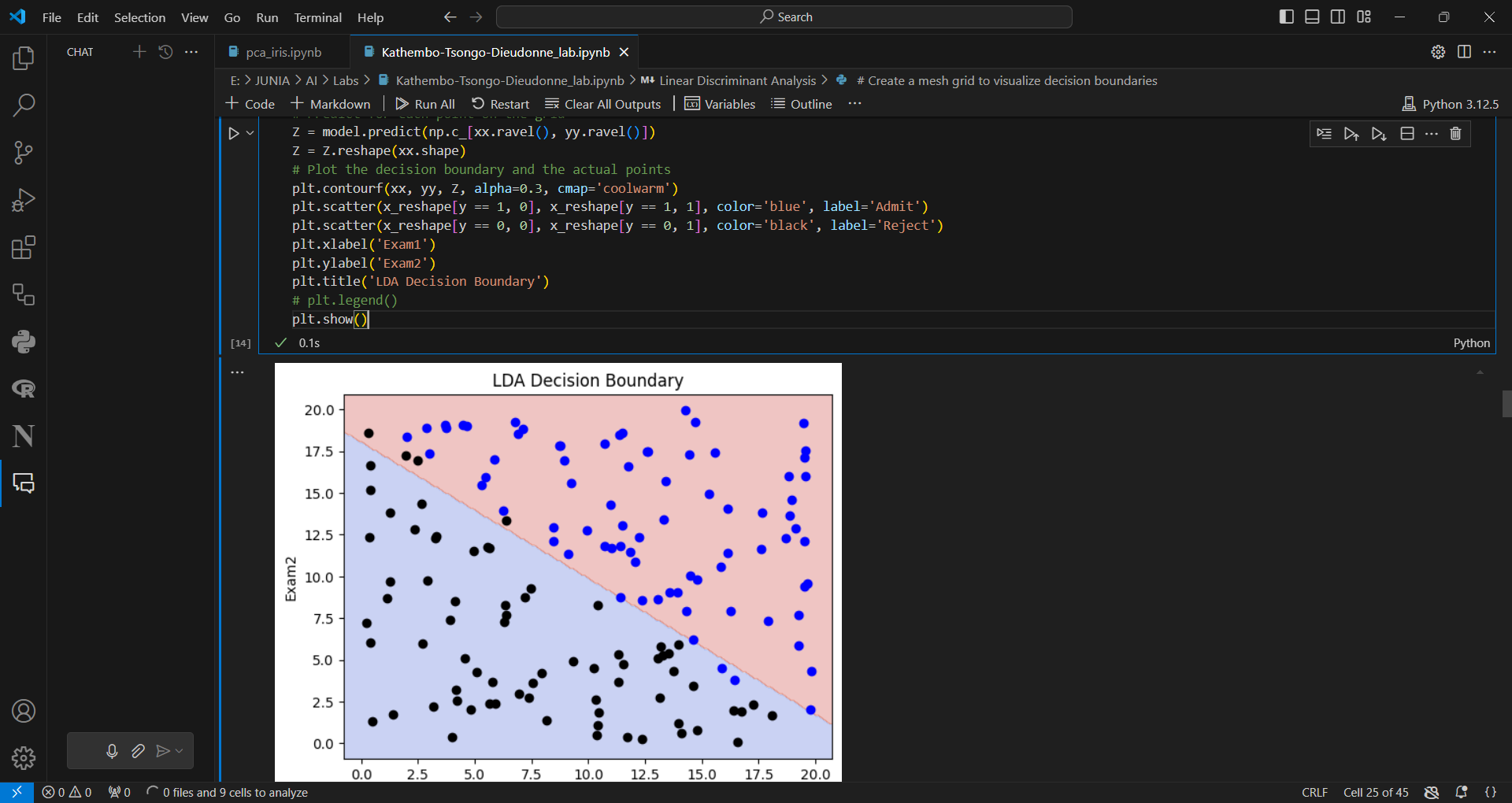


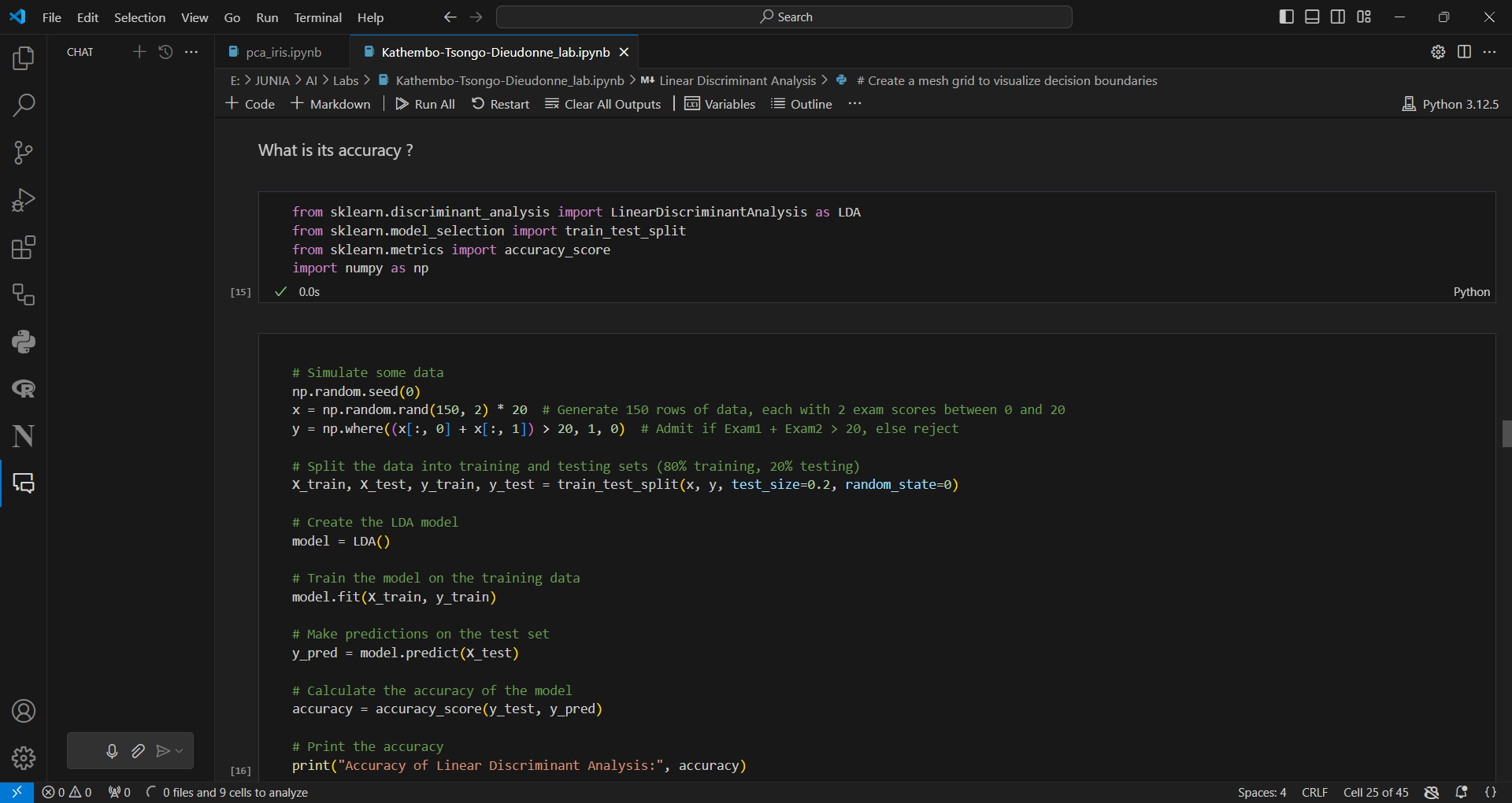


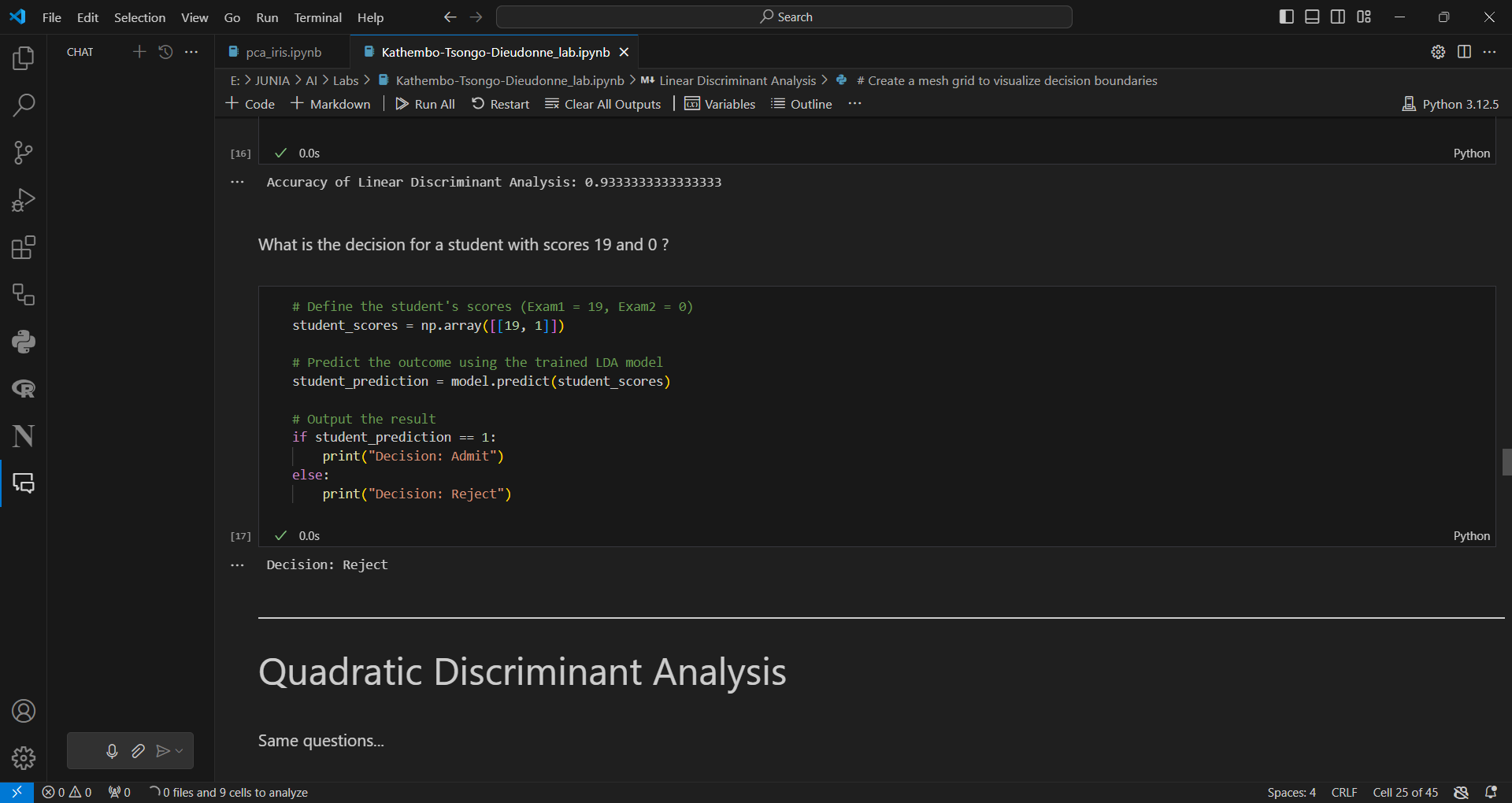


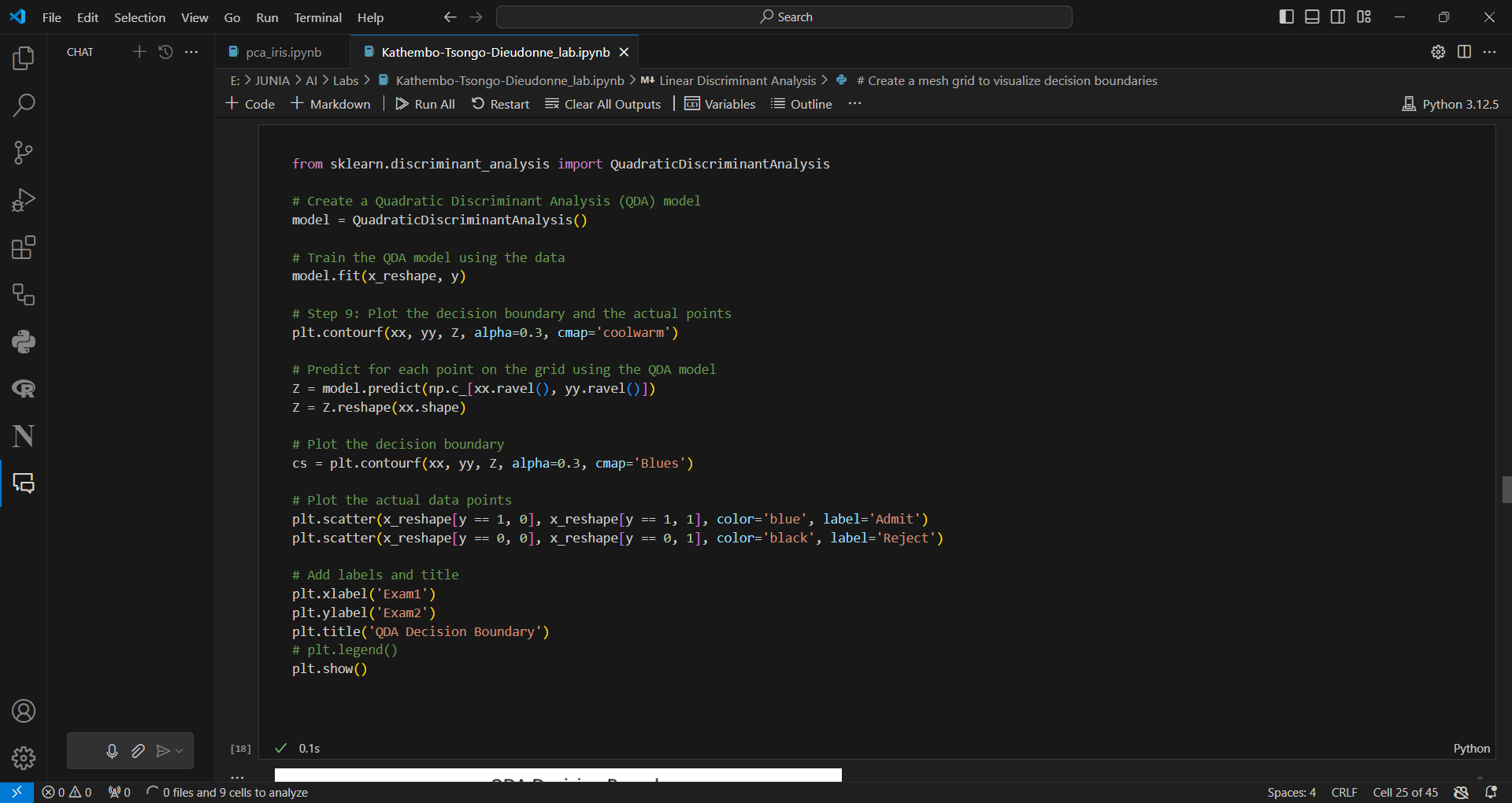


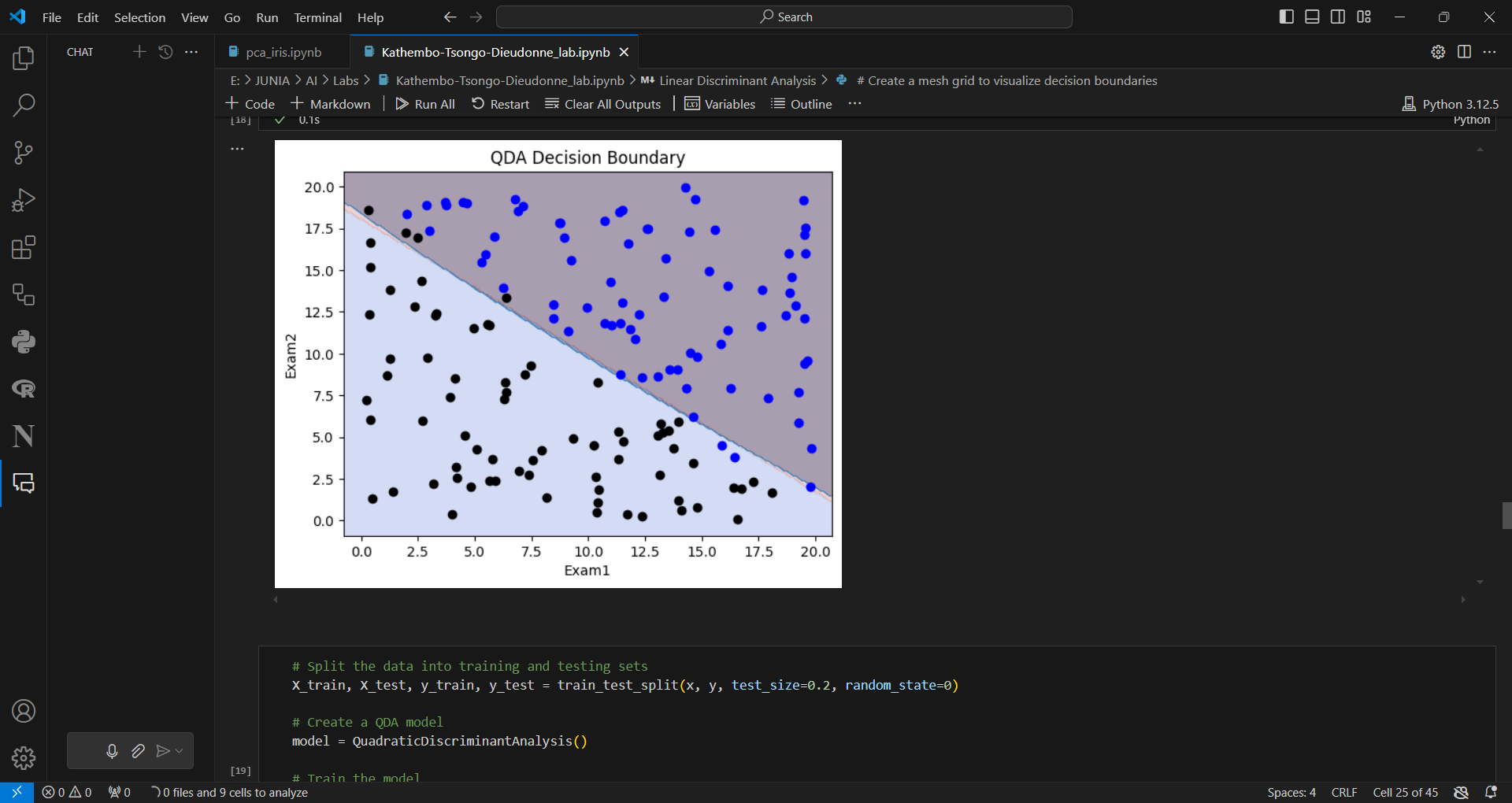


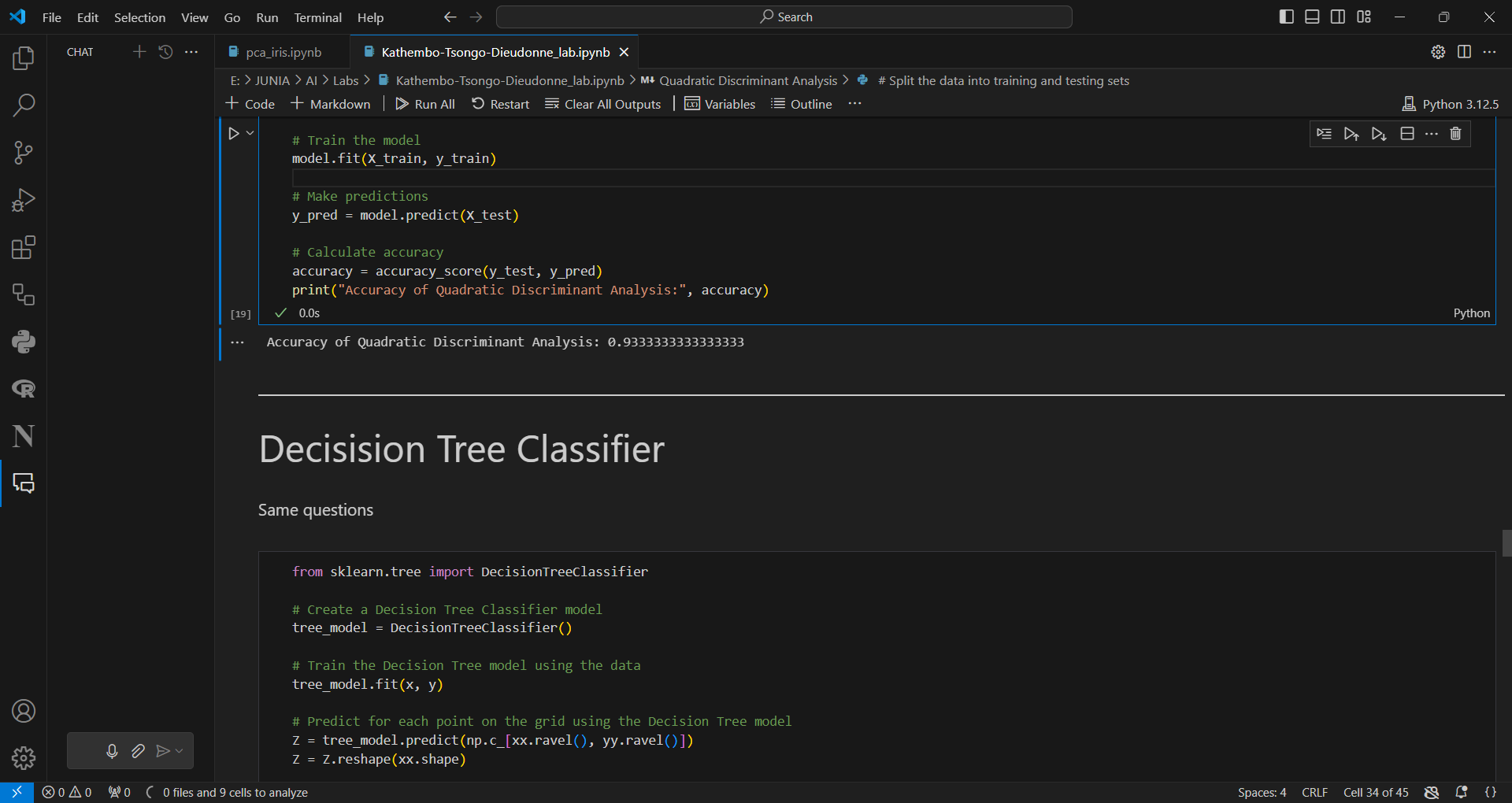


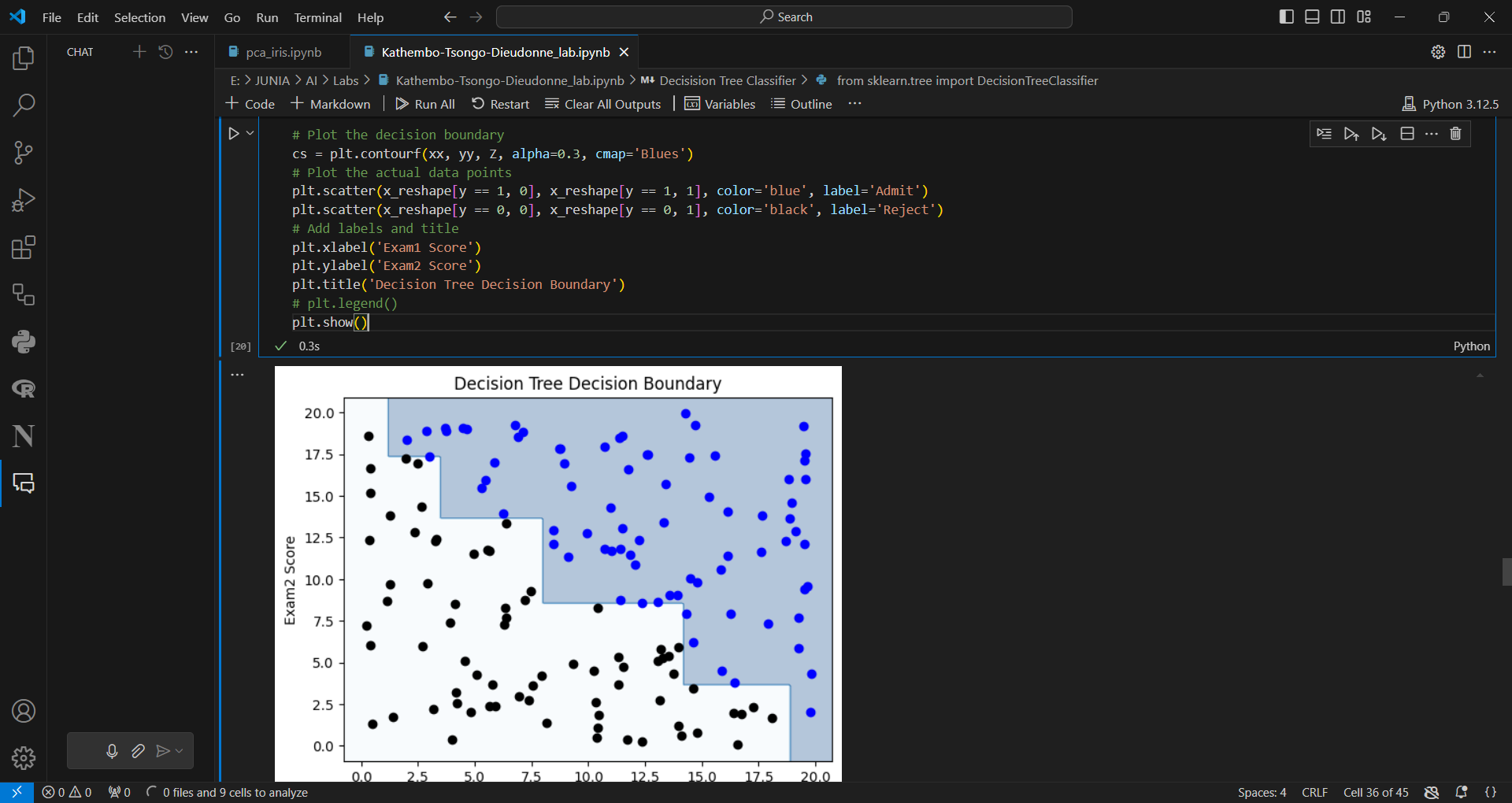


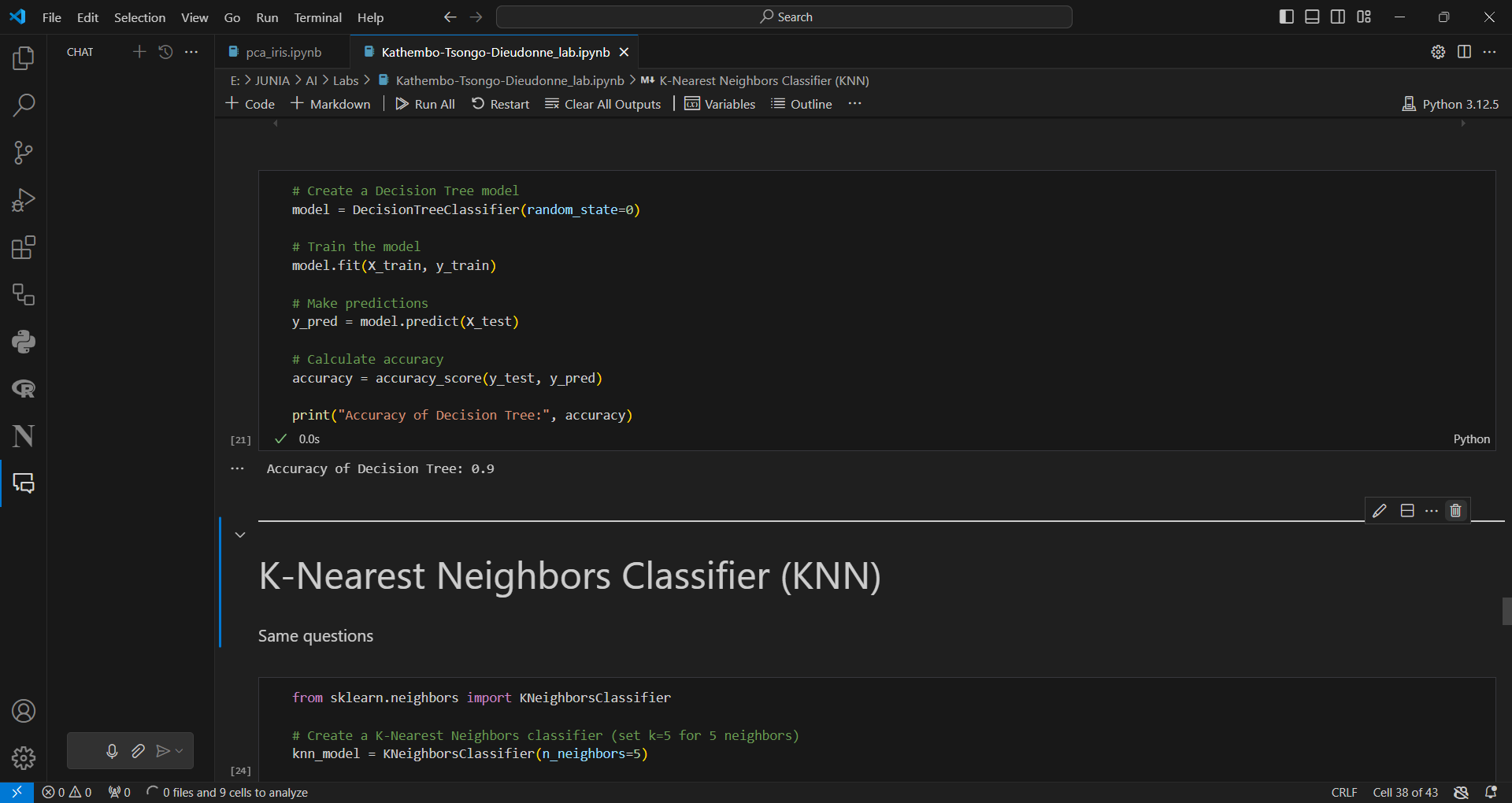


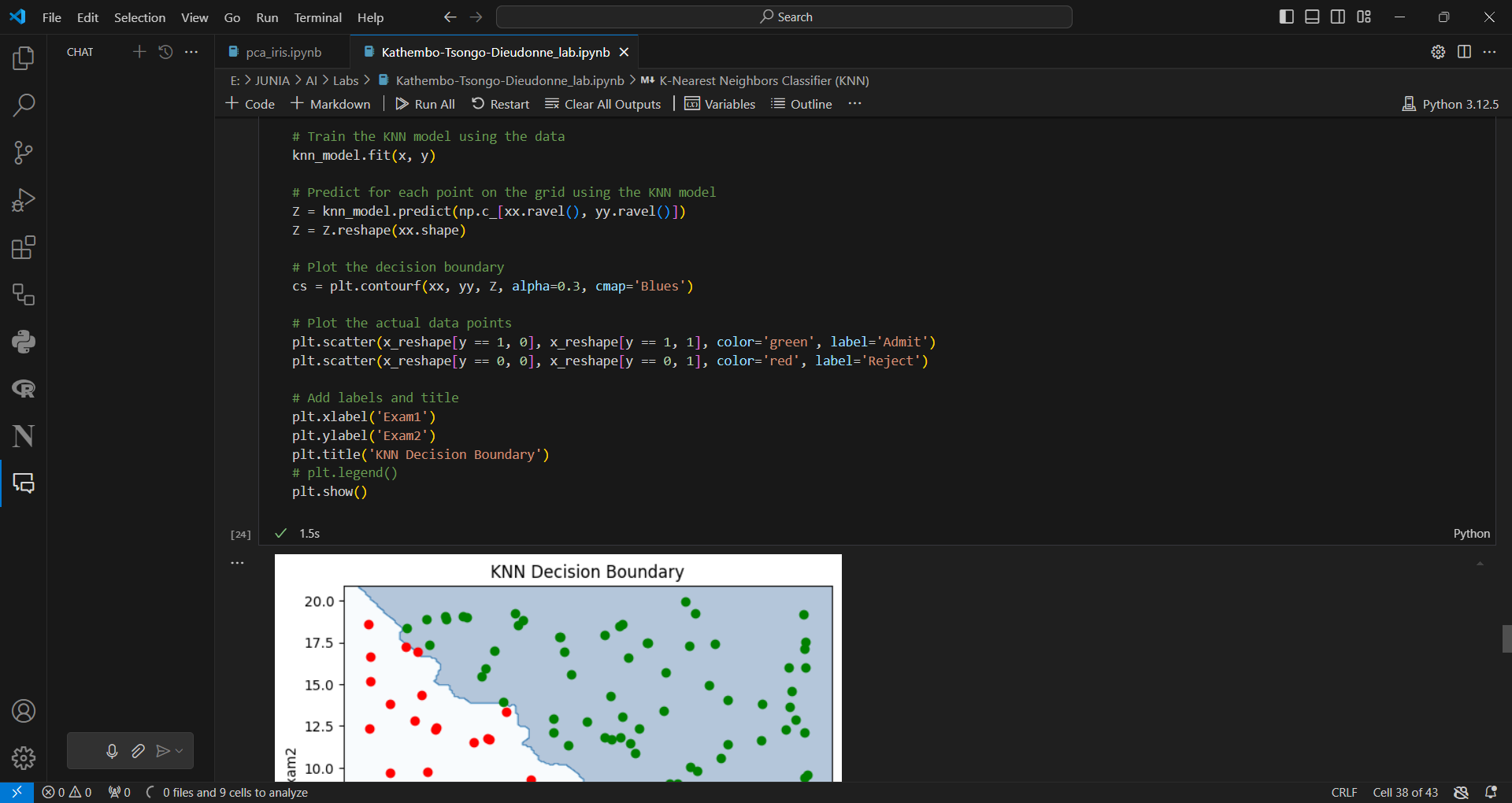


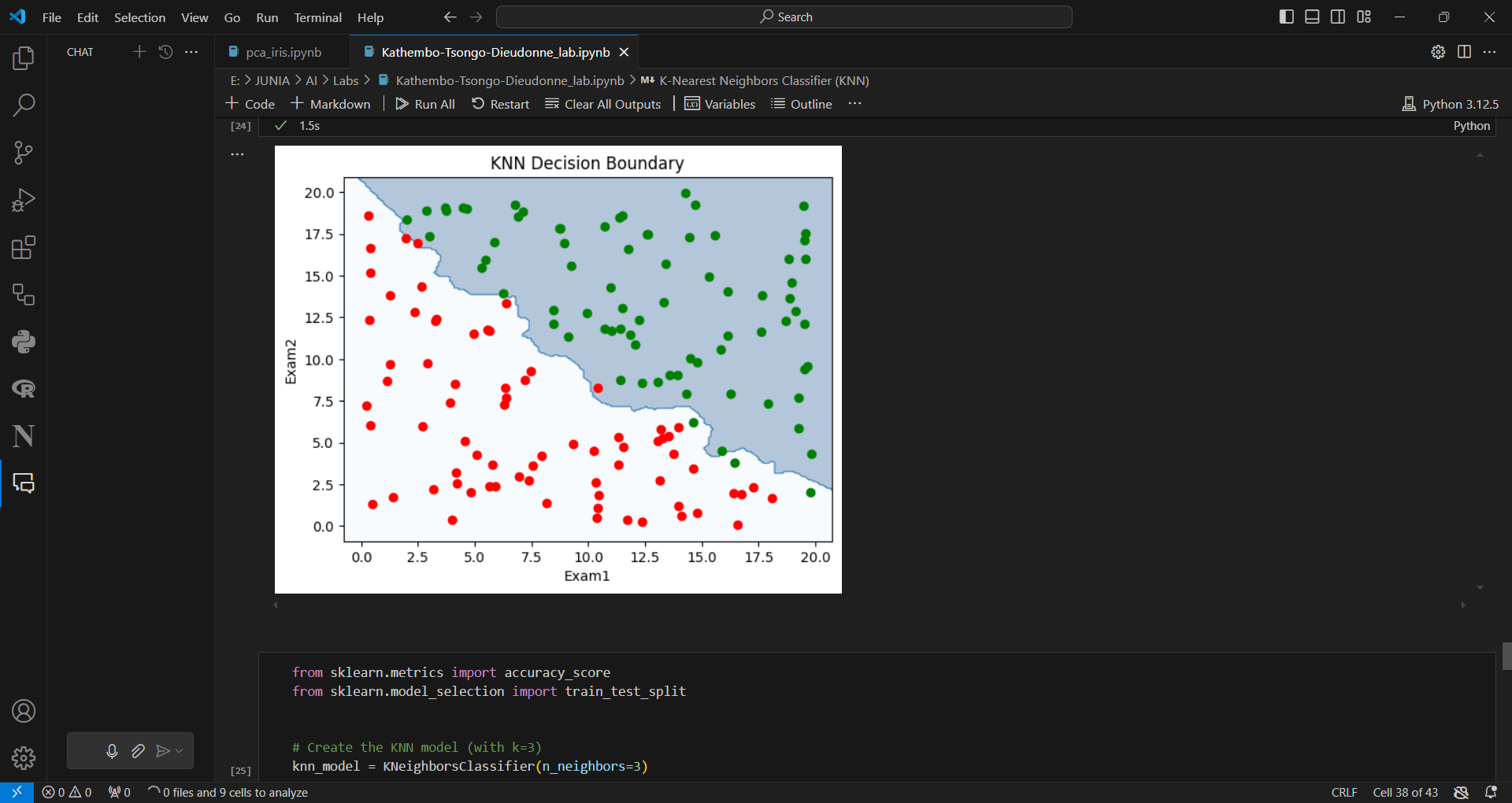


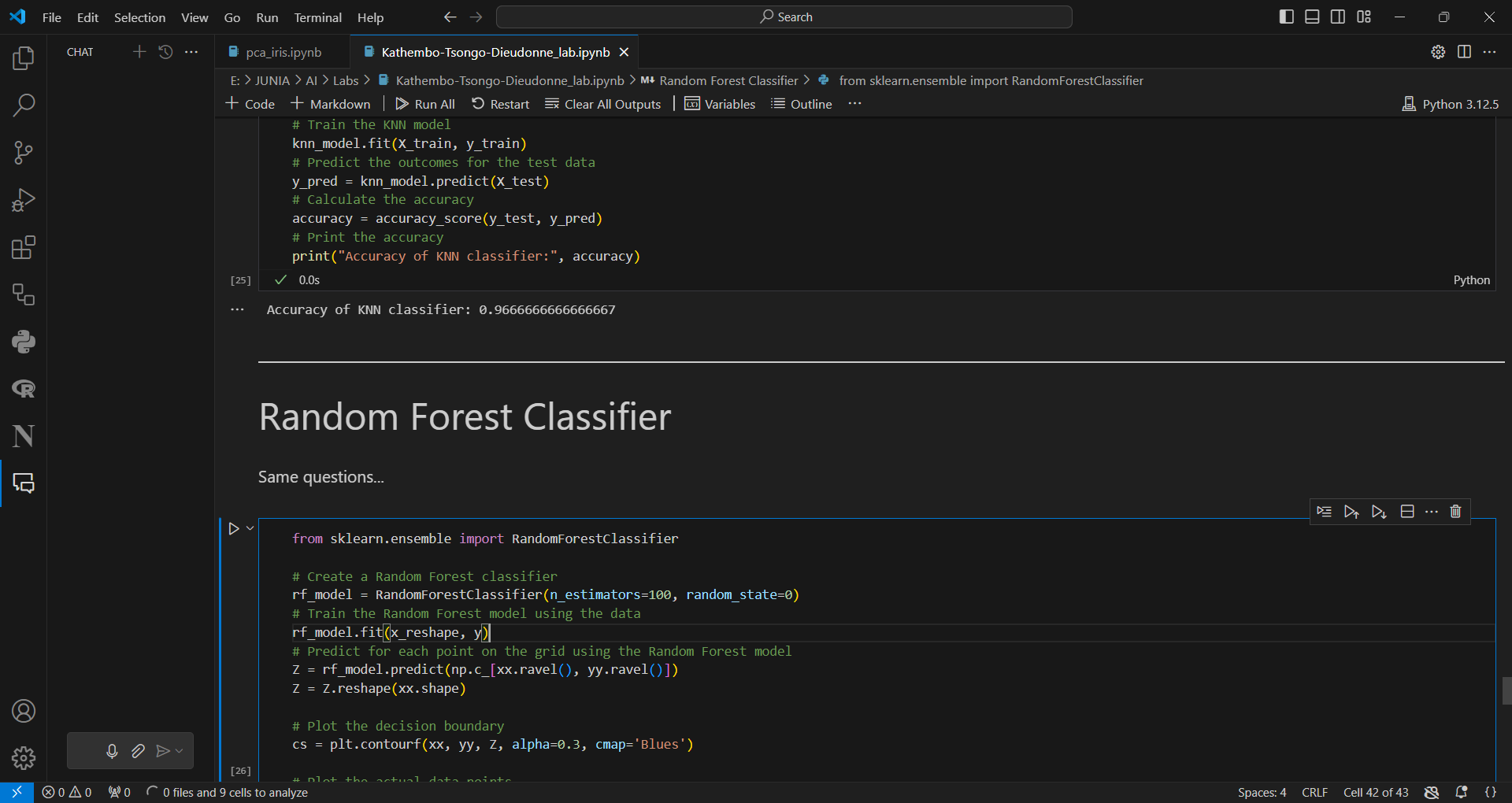


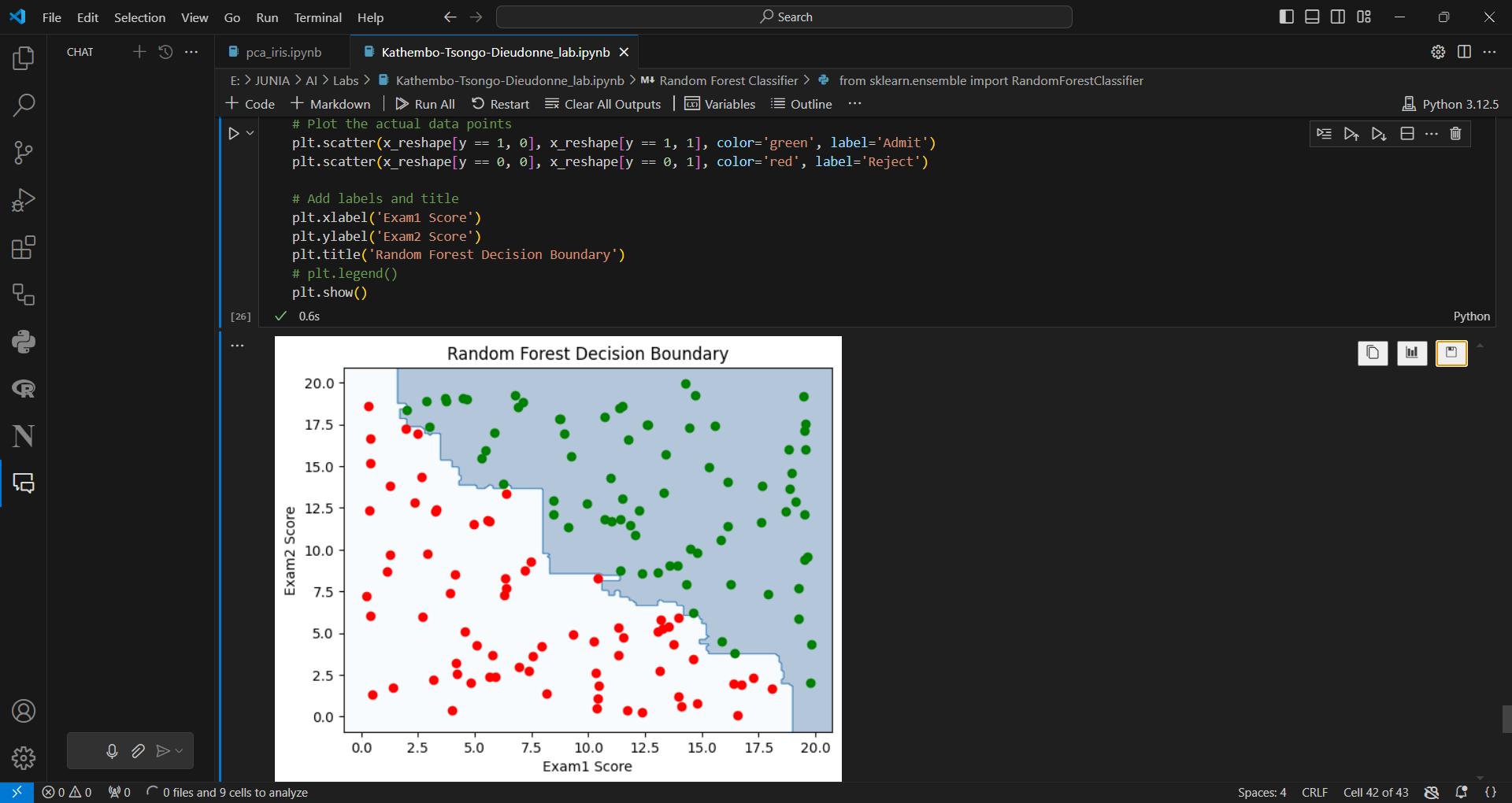


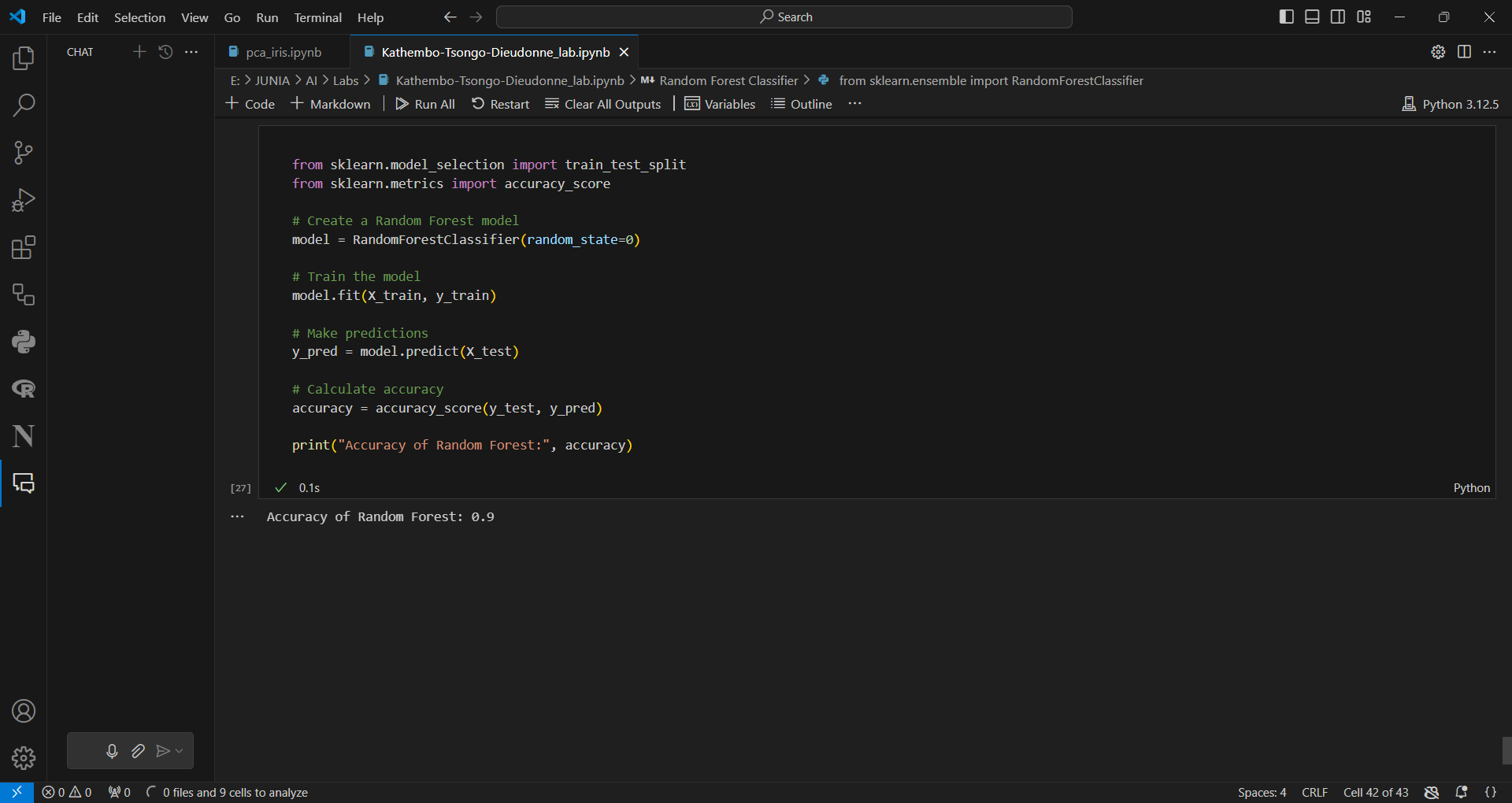












**SET 3 OF LABs: CLUSTERING WITH IRIS DATASET**

**Step 1: Import required libraries in your environment**

Important libraries for clustering using the **Iris dataset are : scikit-learn, matplotlib, seaborn, pandas and numpy**. Here’s a breakdown of what each part does:

**1. from sklearn.datasets import load\_iris**

* **Purpose**: This imports the Iris dataset from scikit-learn, which is a popular dataset in machine learning for classification and clustering tasks.
* **Iris Dataset**: This dataset contains 150 samples of iris flowers, with 4 features: *sepal length, sepal width, petal length, and petal width*. The goal is often to classify the flowers into three species (*Setosa, Versicolor, and Virginica*).

**2. from sklearn.decomposition import PCA**

* **Purpose**: This imports **Principal Component Analysis (PCA)** from scikit-learn.
* **PCA**: A dimensionality reduction technique used to reduce the number of features in your dataset while retaining most of the variation. In clustering, PCA helps visualize high-dimensional data in 2D or 3D plots.

**3. import matplotlib.pyplot as plt**

* **Purpose**: This imports matplotlib, a Python library for data visualization.
* **plt**: It is used to create different types of plots such as scatter plots, line plots, and bar graphs. In this context, you'll use it to visualize the Iris data and clusters after applying PCA.

**4. import seaborn as sns**

* **Purpose**: This imports seaborn, a Python library built on top of matplotlib for statistical data visualization.
* **sns**: It provides a high-level interface for drawing attractive and informative statistical graphics. For example, it helps in visualizing clustered data more clearly with functions like sns.scatterplot.

**5. import pandas as pd**

* **Purpose**: This imports pandas, which is the go-to library for data manipulation and analysis in Python.
* **pd**: It allows you to work with data in tabular form (like in a spreadsheet), with data structures called **DataFrames**. You'll likely use this to load, manipulate, and structure the Iris dataset for easier processing.

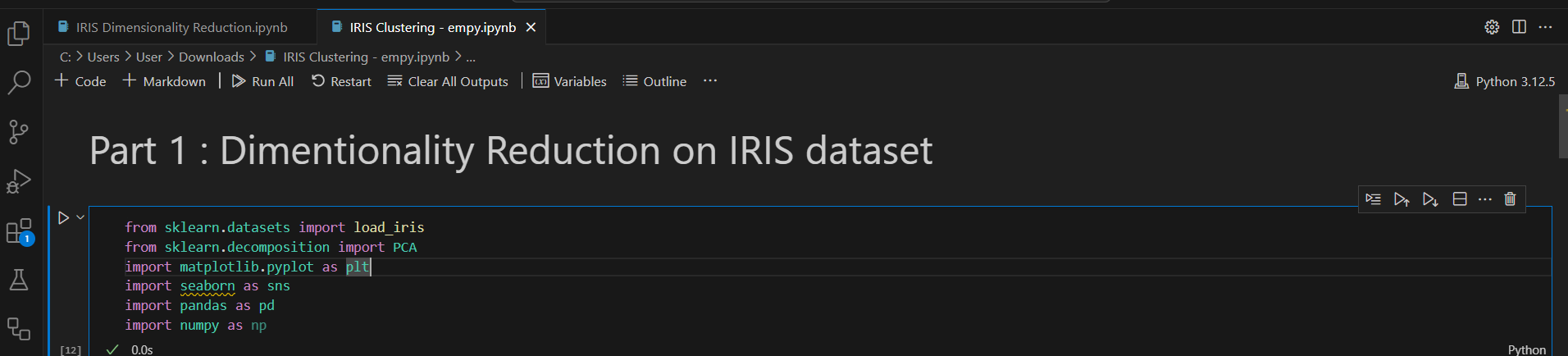
**6. import numpy as np**

* **Purpose**: This imports numpy, a core library for numerical computing in Python.
* **np**: It is used to work with arrays, perform mathematical operations, and for efficient handling of large datasets. In clustering, you might use it to handle the feature arrays from the Iris dataset.

**Overall Workflow:**

1. **Load the Iris dataset**.
2. **Apply PCA** to reduce dimensionality and visualize relationships between data points.
3. Use **matplotlib and seaborn** to create plots for data visualization.
4. **Manipulate data** using pandas and **perform mathematical operations** with numpy.

This combination is common when working with clustering algorithms to explore patterns in datasets like Iris.



**Step 2. Load the Iris dataset and prepare it for analysis**

iris = load\_iris()

* **Purpose**: This line loads the Iris dataset into the variable iris.
* **Output**: The load\_iris() function returns a dictionary-like object containing the data and metadata about the Iris dataset, including:
  + iris.data: The feature values (the measurements of the flowers).
  + iris.target: The labels (species of the iris flowers).
  + iris.feature\_names: The names of the features.

X = iris.data # The inputs

* **Purpose**: This line assigns the feature data (the input variables) from the Iris dataset to the variable X.
* **Output**: X is now a NumPy array containing the measurements for sepal length, sepal width, petal length, and petal width.

y = iris.target # The wanted output

* **Purpose**: This line assigns the target labels (the output variables) to the variable y.
* **Output**: y is now a NumPy array with values corresponding to the species of the iris flowers (0 for setosa, 1 for versicolor, and 2 for virginica).

df = pd.DataFrame(X, columns=iris.feature\_names)

* **Purpose**: This line creates a pandas DataFrame df using the feature data X and assigns the column names from iris.feature\_names.
* **Output**: The DataFrame df now contains the feature data organized in a tabular format with columns named after the features (e.g., *sepal length (cm)*, *sepal width (cm)*, *petal length (cm)*, and *petal width (cm)*).

df['Label'] = y

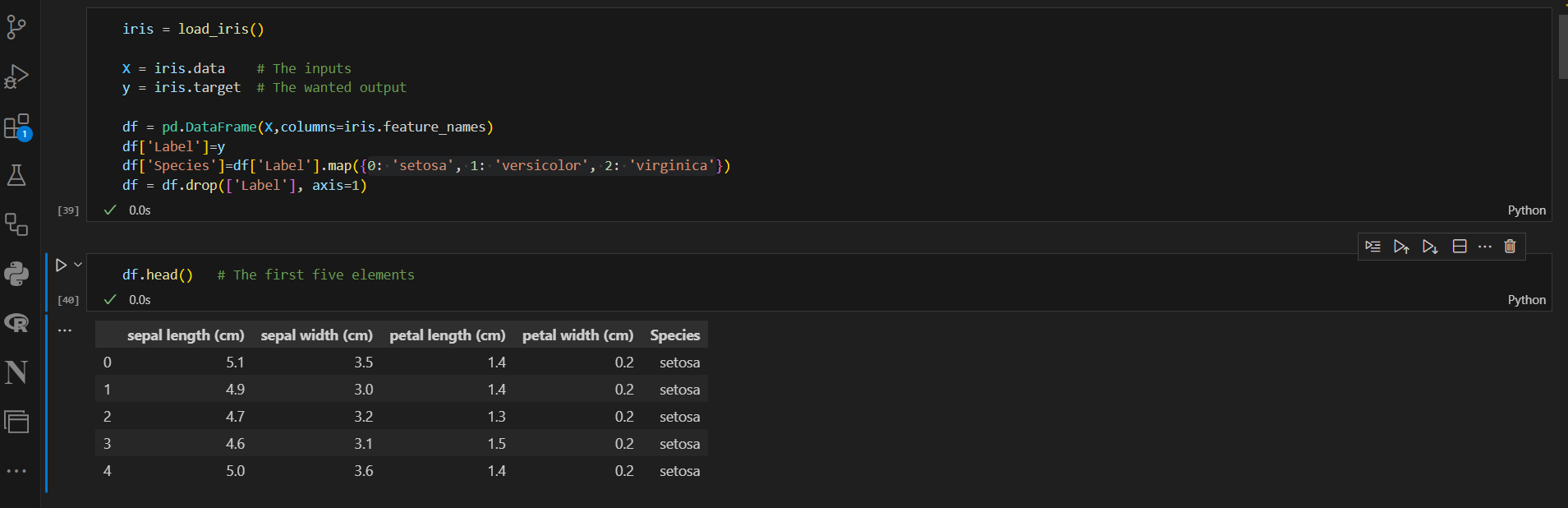
* **Purpose**: This line adds a new column called Label to the DataFrame df, which contains the target labels from the y array.
* **Output**: Now, df has a new column Label that indicates the species classification of each iris flower based on the target variable.

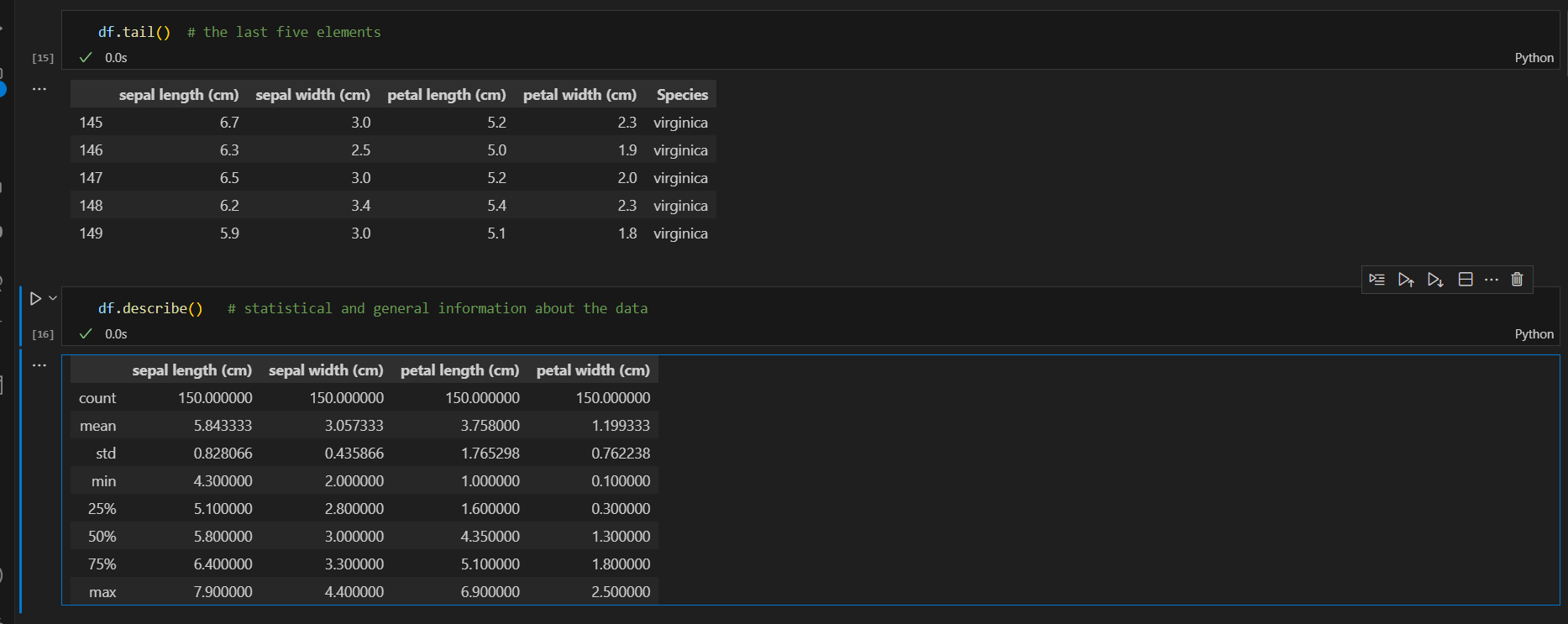
df['Species'] = df['Label'].map({0: 'setosa', 1: 'versicolor', 2: 'virginica'})

* **Purpose**: This line creates a new column Species in the DataFrame that maps the numerical labels (0, 1, 2) to the corresponding species names (setosa, versicolor, virginica).
* **Output**: The Species column now contains the readable species names for each iris flower, making it easier to interpret the data.

df = df.drop(['Label'], axis=1)

* **Purpose**: This line drops the Label column from the DataFrame df as it is no longer needed. This is done to clean up the DataFrame and focus on the features and species names.
* **Output**: The DataFrame df now contains only the feature columns and the Species column.





### Step 3 Explanation: Visualizing the Iris Dataset with Pairplot

In this step, you're using Seaborn's pairplot function to create a grid of scatter plots that visualize the relationships between the different features of the Iris dataset. Here's a detailed breakdown of the line of code:

python

Copy code

sns.pairplot(df, hue='Species')

#### Breakdown of the Code

1. **Function Call**: sns.pairplot(...)
   * This function creates a matrix of scatter plots for each pair of features in the DataFrame.
2. **Input Data**: df
   * The DataFrame df contains the features and species labels you created in the previous step. It has columns for sepal length, sepal width, petal length, petal width, and species names.
3. **Hue Parameter**: hue='Species'
   * The hue parameter specifies that the points in the scatter plots should be colored according to the Species column. This means that each species (setosa, versicolor, virginica) will be represented by a different color in the plots.

#### Result of the Pairplot

When you run this code, the following happens:

* **Matrix of Scatter Plots**: You will see a grid of scatter plots, where each plot represents the relationship between two features of the dataset. For example, you may have scatter plots for:
  + Sepal length vs. Sepal width
  + Sepal length vs. Petal length
  + Petal length vs. Petal width
  + etc.
* **Diagonal Histograms**: The diagonal of the grid shows the distribution of each feature using histograms or kernel density plots.
* **Color Coding**: Each point in the scatter plots will be colored based on the species. This visual representation helps you see how well-separated the species are based on their feature measurements.

#### Example Visualization

Here's an example of what the resulting pairplot might look like:

* **Axes**: Each subplot will have axes labeled with the names of the features being compared.
* **Points**: Points will be plotted in different colors representing each species. For example:
  + Setosa points may be blue,
  + Versicolor points may be orange,
  + Virginica points may be green.

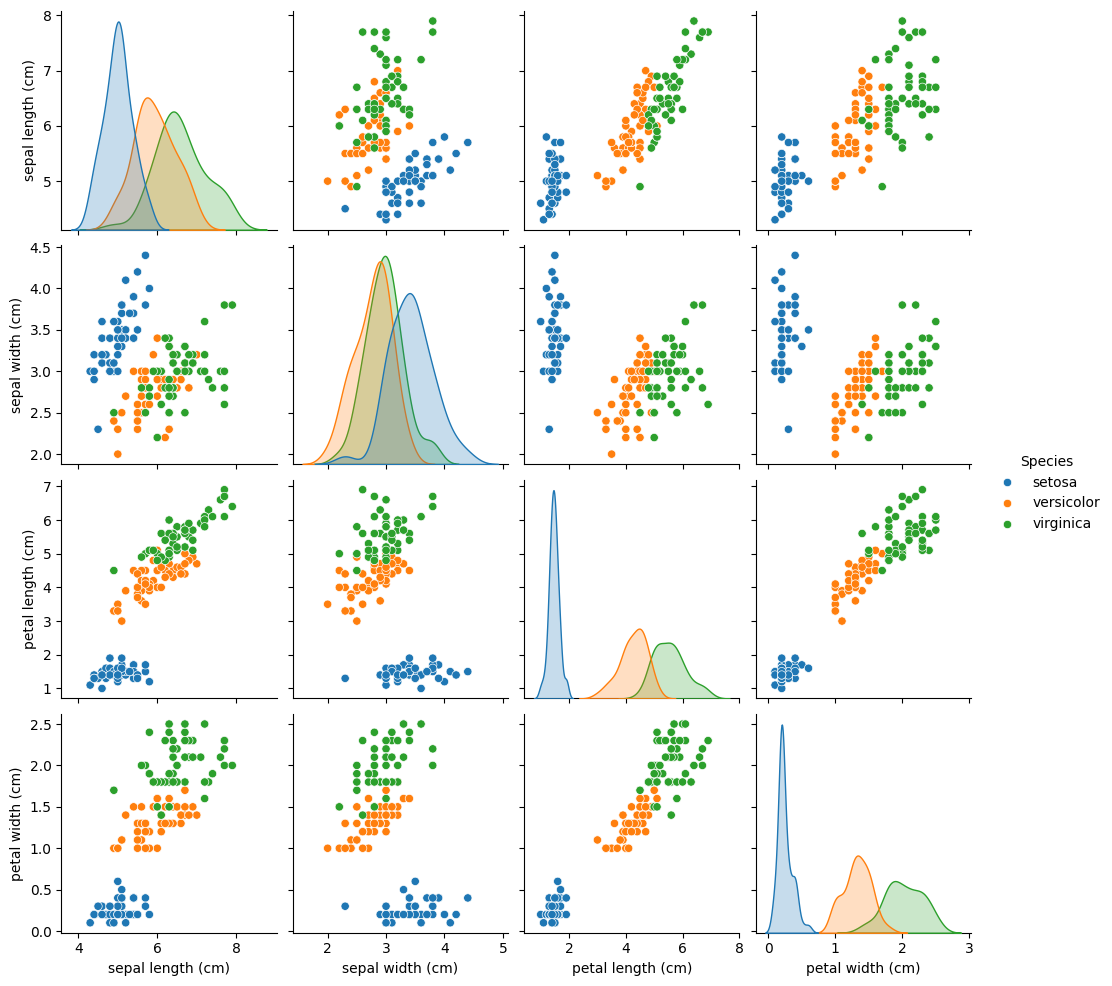
#### Interpretation

The pairplot helps in understanding the relationships between different features and how they relate to each species. For instance, you may observe that:

* Setosa is well-separated from the other two species, particularly in the petal dimensions.
* There may be some overlap between versicolor and virginica, indicating that these two species may be harder to distinguish based solely on their features.

### Final Note

Using sns.pairplot is a great way to visually explore the dataset and identify patterns, correlations, and potential clusters among the different iris species before applying further analysis or clustering algorithms.



### Step 4. Applying PCA and Merging Results

In this step, you're applying Principal Component Analysis (PCA) to the Iris dataset to reduce its dimensionality and then merging the PCA-transformed features back into the original DataFrame. Here's a detailed breakdown of the code:

1. **Importing PCA**:

pca = PCA()

* + Here, you create an instance of the PCA class from the sklearn.decomposition module. This initializes the PCA model, which will be used to perform dimensionality reduction.

1. **Transforming the Features**:

X\_pca = pca.fit\_transform(X)

* + fit\_transform(X) does two things:
    - **Fit**: It computes the principal components and the amount of variance each principal component captures from the dataset.
    - **Transform**: It projects the original features (in this case, the Iris features) onto the principal component space. This effectively reduces the dimensionality of your dataset while preserving as much variance as possible.
  + The output X\_pca is an array where each row corresponds to a sample (iris flower) and each column corresponds to a principal component (PC).

1. **Creating a DataFrame for PCA Results**:

pca\_df = pd.DataFrame(X\_pca, columns=['PC1', 'PC2', 'PC3', 'PC4'])

* + Here, you create a new DataFrame pca\_df that contains the PCA-transformed features. You assign column names like 'PC1', 'PC2', 'PC3', and 'PC4' to represent the first four principal components.

1. **Merging PCA Results with Original DataFrame**:

df = pd.merge(df, pca\_df, right\_index=True, left\_index=True)

* + This line merges the PCA DataFrame (pca\_df) back into the original DataFrame (df).
  + The right\_index=True and left\_index=True arguments ensure that the merge is done based on the index of both DataFrames. This effectively combines the PCA features with the original features and species labels into one DataFrame.

### Why Use PCA?

* **Dimensionality Reduction**: The Iris dataset has four features (sepal length, sepal width, petal length, petal width). By applying PCA, you can reduce these four dimensions down to two (or even one) while retaining most of the information in the dataset.
* **Visualizing Relationships**: By focusing on the principal components, you can visualize the data in a way that emphasizes the most significant relationships, making it easier to identify clusters or groupings.

### Result of This Step

At this point in your analysis, df now contains the original features of the Iris dataset along with the PCA-transformed features. This allows you to visualize the data in a lower-dimensional space, which is particularly useful for clustering and understanding the structure of the dataset.

### Step 5. Explained Variance Ratio of Principal Components

In this step, you're evaluating how much variance each principal component (PC) captures from the original dataset. This is crucial for understanding the effectiveness of PCA in reducing dimensions while retaining important information.

**User this code:**

for i in range(4):

print('PC{}: {}'.format(i+1, pca.explained\_variance\_ratio\_[i]))

#### Breakdown of the Code

1. **Loop Through Principal Components**:

 **for i in range(4):**

* This is a **for loop** that will iterate over the numbers 0, 1, 2, and 3 (a total of 4 iterations).
* The **range(4)** generates a sequence of integers from 0 to 3, which corresponds to the first four principal components (PC1, PC2, PC3, PC4).

 **print('PC{}: {}'.format(i+1, pca.explained\_variance\_ratio\_[i]))**

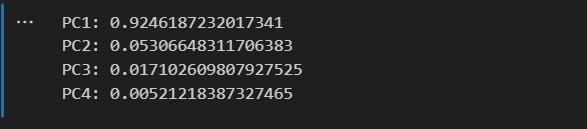
* **'PC{}: {}'** is a format string. The curly braces {} are placeholders that will be filled in with values by the **format()** function.
* **format(i+1, pca.explained\_variance\_ratio\_[i])**:
  + **i+1**: This calculates the principal component number (PC1, PC2, PC3, etc.). Since the loop starts at i = 0, adding 1 gives the human-readable form (i.e., PC1 instead of PC0).
  + **pca.explained\_variance\_ratio\_[i]**: This retrieves the proportion of variance explained by the i-th principal component.
    - **pca.explained\_variance\_ratio\_** is an array that contains the variance ratio for each principal component. It tells us how much variance each principal component explains relative to the total variance in the dataset.

### Purpose of Explained Variance Ratio

* **Understanding Variance**: The explained variance ratio tells you how much information (variance) each principal component captures from the original features. For example:
  + If PC1 explains 70% of the variance, it means that 70% of the total variability in the dataset can be attributed to the first principal component.
* **Selecting Components**: This information is essential for deciding how many principal components to keep. Typically, you want to retain components that capture a significant amount of the variance (e.g., 95% or more).
* **Model Performance**: Understanding which components explain the most variance can help in improving the performance of machine learning models, as retaining significant components can enhance predictive capabilities.

### Expected Output

When you run this code, you should see output similar to:



Each line indicates the percentage of variance explained by each principal component:

* **PC1**: This component captures a significant amount of the variance (e.g., around 92% in the example), indicating that it is very informative.
* **PC2**: This component captures a much smaller percentage of the variance (e.g., around 5%), suggesting it is less informative compared to PC1.
* **PC3** and **PC4**: These components capture very little variance, indicating that they add minimal additional information.

### Next Steps

After evaluating the explained variance ratios, you can decide how many principal components to use for further analysis or visualization. If a few components explain most of the variance, you can effectively reduce the dimensionality of your dataset while preserving essential information. This would then lead you to visualize the data using the retained principal components, typically the first two, in a scatter plot or similar visualization.

### Step 6 : Visualizing the Iris Dataset with PCA

In this step, you are using Seaborn's stripplot to visualize the relationship between the first principal component (PC1) and the species of the Iris flowers. This visualization helps in understanding how the different species are distributed along the first principal component, which captures a significant amount of variance from the dataset.

**Use this code:**

sns.stripplot(x="PC1", y="Species", data=df, jitter=True)

plt.title('Iris Data Visualized in One Dimension');

#### Breakdown of the Code

1. **Seaborn Strip Plot**:

sns.stripplot(x="PC1", y="Species", data=df, jitter=True)

* + sns.stripplot: This function creates a scatter plot where one of the axes is categorical (in this case, the species of the iris). It shows the distribution of data points along the specified axis.
  + x="PC1": This specifies that the x-axis will represent the first principal component, which contains the transformed data from PCA.
  + y="Species": This specifies that the y-axis will categorize the data points based on the species of Iris flowers (setosa, versicolor, virginica).
  + data=df: This specifies that the data for the plot comes from the df DataFrame, which contains both the PCA results and the species labels.
  + jitter=True: This option adds some random noise (jitter) to the points on the y-axis, which helps to spread out overlapping points and makes it easier to see the distribution of the different species.

1. **Setting the Title**:

plt.title('Iris Data Visualized in One Dimension');

* + This line adds a title to the plot, indicating what the visualization represents.

### Purpose of the Visualization

* **Understanding Distribution**: The strip plot allows you to see how the first principal component (PC1) relates to the different species of Iris flowers. It shows whether the species are distinctly separated or if there is overlap.
* **Visualizing Clustering**: If the points for different species are well-separated, it indicates that the PCA transformation has effectively captured the differences among the species. Conversely, if there is a lot of overlap, it might suggest that PC1 does not fully distinguish between the species.
* **Dimensionality Reduction Impact**: By visualizing the data in terms of the principal components, you can assess how well PCA has reduced the dimensionality of the dataset while maintaining the important distinguishing features.

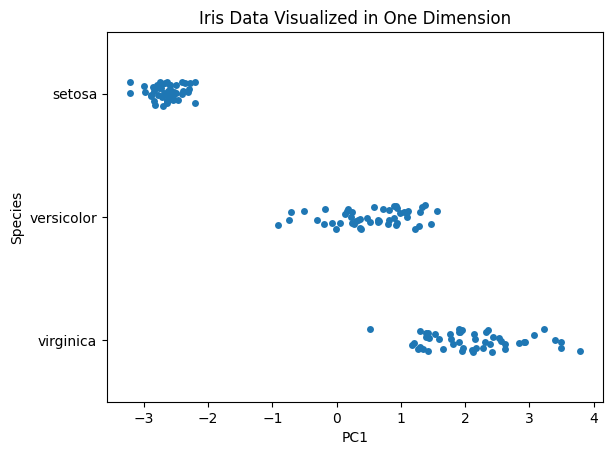
### Expected Output

When you run this code, you should see a plot that looks something where:

* The x-axis will show the values of PC1, and the y-axis will display the three species of Iris flowers.
* The points corresponding to each species will be plotted along the x-axis, with some jitter to spread out overlapping points.

### Interpretation

* If the three species are distinctly spread along PC1, it suggests that this principal component captures key differences between them.
* You might observe clusters of points that belong to the same species, indicating that PCA has effectively reduced the dimensionality while preserving the essential characteristics that differentiate the species.



### Next Steps

Following this visualization, you could explore further visualizations using the second principal component (PC2) or a combination of both PC1 and PC2 to gain deeper insights into the data distribution and relationships among the species. You could also consider applying clustering algorithms to see if they align with the actual species labels.

### Let’s try Visualizing PCA Results using the 2D

The code here will use Seaborn's lmplot to create a scatter plot visualizing the first two principal components (PC1 and PC2) of the Iris dataset, colored by species.

sns.lmplot(x='PC1', y='PC2', data=df, hue='Species', fit\_reg=False)

plt.title('Iris Data Visualized in Two Dimensions');

#### Breakdown of the Code

1. **Seaborn's lmplot Function**:

sns.lmplot(x='PC1', y='PC2', data=df, hue='Species', fit\_reg=False)

* + **sns.lmplot**: This function creates a scatter plot (with the option to fit a regression line, but here it is set to False) that allows for grouping and coloring of points based on a categorical variable. It is typically used for linear regression plots, but here it’s simply used for scatter visualization.
  + **x='PC1'**: This sets the x-axis of the plot to the values of the first principal component, which captures a significant amount of the variance in the dataset.
  + **y='PC2'**: This sets the y-axis of the plot to the values of the second principal component. Together, PC1 and PC2 provide a two-dimensional representation of the original four-dimensional dataset (the four features of the Iris flowers).
  + **data=df**: This specifies that the source of the data is the df DataFrame, which includes the PCA-transformed values along with the species labels.
  + **hue='Species'**: This parameter colors the points in the scatter plot based on the species of the Iris flowers. Each species will be represented in a different color, making it easy to visualize how the species are distributed in the space defined by PC1 and PC2.

**Explanation of fit\_reg:**

* **fit\_reg=False**: This parameter is used to control whether a regression line should be drawn on the plot. The default value is True, which would fit and display a regression line. By setting it to False, you are telling Seaborn **not** to fit a regression line to the data.
* This is important because in this case, you are visualizing **principal components** (PC1 and PC2) of the **Iris dataset**, and you are interested in the scatterplot of the data points for different species, not in the linear relationship between these components. A regression line wouldn't be meaningful in this context since PCA doesn't imply any regression-based relationship.

### Purpose of the Visualization

* **Two-Dimensional Visualization**: The main purpose of this plot is to visualize the relationships between the Iris species in a two-dimensional space formed by the first two principal components (PC1 and PC2). This can help reveal clusters or patterns that may exist among the species.
* **Understanding Clusters**: By visualizing the data in two dimensions, it becomes easier to identify how well-separated the species are. If the points of different species form distinct clusters, it indicates that PCA has successfully captured important differences between them.
* **Exploring Data Relationships**: The scatter plot allows for a quick visual assessment of the distribution of species in relation to the principal components, helping to identify areas of overlap or separation among the species.

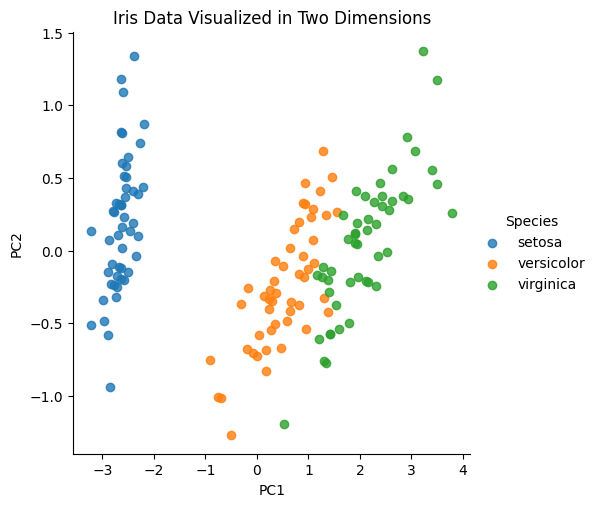
### Expected Output

When you run this code, you should see a scatter plot with:

* **X-axis**: Representing values of the first principal component (PC1).
* **Y-axis**: Representing values of the second principal component (PC2).
* **Colored Points**: Different species of Iris (Setosa, Versicolor, and Virginica) will be represented in different colors.

### Example of Expected Visualization

You might see a plot resembling the one below, where each color corresponds to a different species:

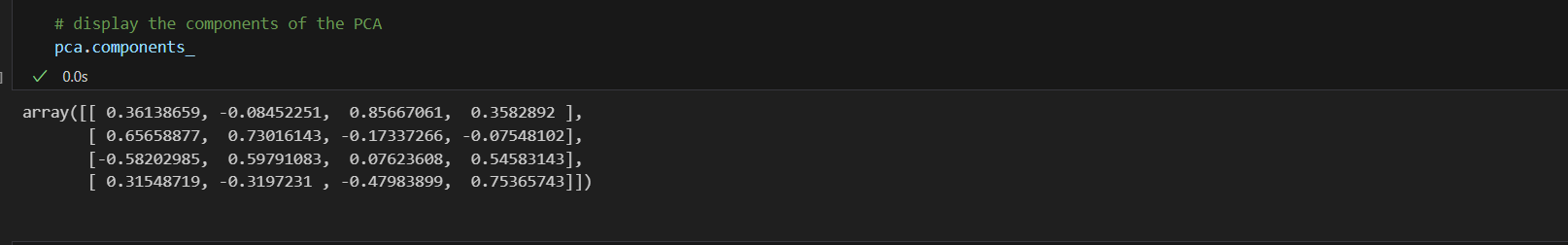


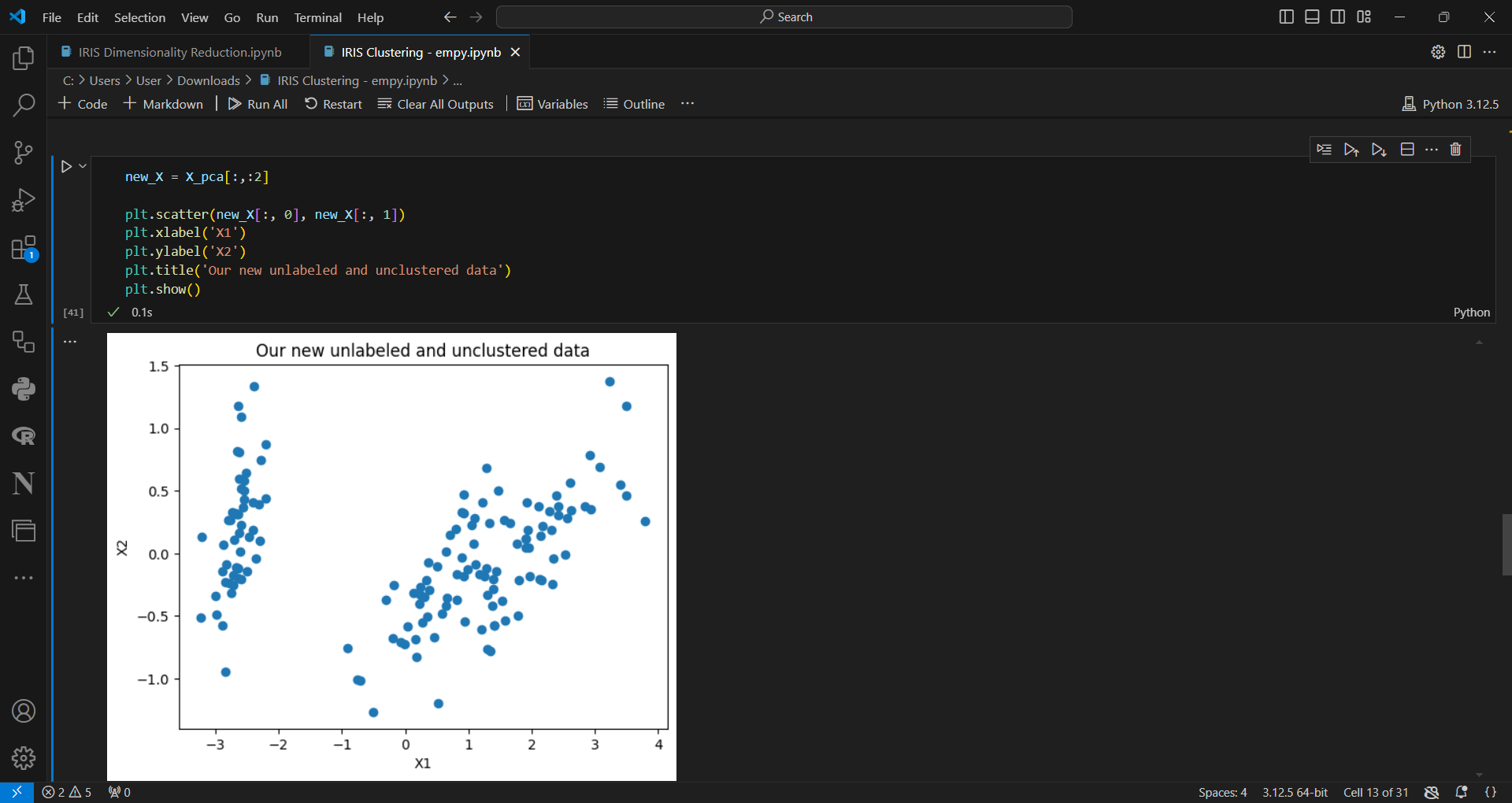
(This is just a placeholder; your plot will reflect your specific dataset.)

* **Clusters**: Ideally, you'll observe clusters corresponding to each species. For example, Setosa may be distinctly separate from the other two species, while Versicolor and Virginica may show some overlap.

### Conclusion

This visualization is an essential part of the exploratory data analysis (EDA) process, helping to understand the underlying structure of the dataset after applying PCA. It allows you to visually assess the effectiveness of PCA in reducing dimensionality while preserving the relationships between data points. If clear clusters are present, it suggests that the principal components have captured meaningful information about the species differences.







**K-Means Clustering on the Reduced Dataset:**

python

Copy code

from sklearn.cluster import KMeans

# Step 1: Create a KMeans model with 3 clusters (since we know there are 3 iris species)

kmeans = KMeans(n\_clusters=3, random\_state=42)

# Step 2: Fit the model to the 2D data from PCA

kmeans.fit(new\_X)

# Step 3: Predict the cluster labels for each point

clusters = kmeans.predict(new\_X)

# Step 4: Add the cluster labels to the DataFrame for visualization

df['Cluster'] = clusters

# Step 5: Visualize the clustering result using a scatter plot

plt.scatter(new\_X[:, 0], new\_X[:, 1], c=clusters, cmap='viridis')

plt.xlabel('PC1')

plt.ylabel('PC2')

plt.title('K-Means Clustering of PCA-Reduced Iris Data')

plt.show()

**Step-by-step Explanation:**

**1. Importing the K-Means algorithm:**

python

Copy code

from sklearn.cluster import KMeans

* **KMeans** is imported from Scikit-learn's cluster module. K-Means is an unsupervised learning algorithm used to group data into clusters.

**2. Initializing the K-Means model:**

python

Copy code

kmeans = KMeans(n\_clusters=3, random\_state=42)

* **n\_clusters=3**: This specifies that we want to group the data into 3 clusters (because the Iris dataset has 3 species, though K-Means does not need to know this beforehand).
* **random\_state=42**: Setting a random seed ensures reproducibility of the clustering results.

**3. Fitting the K-Means model to the PCA-reduced data:**

python

Copy code

kmeans.fit(new\_X)

* The **fit()** method applies the K-Means algorithm to the data (new\_X), which contains the 2D data from the PCA.
* The algorithm works by:
  + Randomly initializing 3 centroids (since n\_clusters=3).
  + Assigning each point in new\_X to the nearest centroid.
  + Updating the centroids based on the average position of the points assigned to each cluster.
  + Repeating the assignment and update steps until the centroids stabilize (convergence).

**4. Predicting the cluster labels:**

python

Copy code

clusters = kmeans.predict(new\_X)

* **predict()** assigns each data point to one of the 3 clusters.
* clusters is an array of labels, where each entry represents the cluster to which the corresponding point in new\_X belongs (i.e., cluster 0, 1, or 2).

**5. Adding the cluster labels to the DataFrame:**

python

Copy code

df['Cluster'] = clusters

* A new column **Cluster** is added to the df DataFrame, where each row corresponds to the cluster label (0, 1, or 2) assigned by the K-Means algorithm.
* This allows us to visualize the clusters alongside the actual species labels (Species).

**6. Visualizing the clustering result:**

python

Copy code

plt.scatter(new\_X[:, 0], new\_X[:, 1], c=clusters, cmap='viridis')

plt.xlabel('PC1')

plt.ylabel('PC2')

plt.title('K-Means Clustering of PCA-Reduced Iris Data')

plt.show()

* **plt.scatter()** creates a scatter plot of the PCA-reduced data (new\_X) in two dimensions (PC1 vs PC2).
* The points are colored using **c=clusters**, where clusters represents the predicted cluster labels (0, 1, or 2).
* **cmap='viridis'**: A color map is applied to the plot so that each cluster is represented by a distinct color.
* The plot will show how K-Means grouped the data points into clusters, helping us visualize how well the algorithm was able to separate the iris species.

**Final Outcome:**

* After applying K-Means, you'll see a scatter plot with points colored by their cluster assignments. This lets you see how well the PCA-reduced data points are grouped into 3 clusters based on the 2D data.
* K-Means doesn't know the actual species labels, but you can visually compare the clustering result to the true species using the species labels (df['Species']).

**Important Notes:**

* Since K-Means is unsupervised, it doesn’t use the species labels to group the data; it only tries to group the points based on their positions in the 2D PCA-reduced space.
* The clustering might not perfectly align with the actual species because PCA reduces the data to two dimensions, which might lose some information compared to the original 4D feature space.

If you want to apply **K-Means clustering** with k=3 (which is the number of clusters corresponding to the three species in the Iris dataset: setosa, versicolor, and virginica), you don't need to change much since your current K-Means code already uses n\_clusters=3.

However, if you were referring to changing from a different value of k (like k=2 or another value) to k=3, you would only need to ensure that n\_clusters=3 in your K-Means model initialization.

Here’s the relevant line where k=3 is specified:

python

Copy code

# Initialize K-Means with 3 clusters

kmeans = KMeans(n\_clusters=3, random\_state=42)

**Step-by-Step Explanation:**

* n\_clusters=3 means you are asking K-Means to find 3 clusters in the data, which corresponds to the 3 species in the Iris dataset.

If you wanted to change the number of clusters (say to k=4), you would simply change the value of n\_clusters:

python

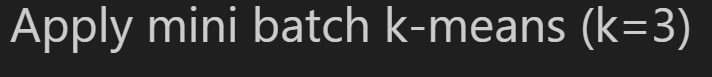
Copy code

# Initialize K-Means with 4 clusters instead of 3

kmeans = KMeans(n\_clusters=4, random\_state=42)

**Summary:**

* To apply K-Means with **k=3** (which you are already doing), ensure that n\_clusters=3 is set when initializing the model.
* If you want a different number of clusters, just adjust n\_clusters to the desired value.

To apply **Mini Batch K-Means** clustering on the Iris dataset with k=3, follow the steps below. Mini Batch K-Means is a more efficient version of K-Means that uses small random batches of the data to reduce computational cost.

Here's the code and explanation for using **Mini Batch K-Means** with k=3:

**Step-by-Step Implementation**

python

Copy code

from sklearn.datasets import load\_iris

from sklearn.decomposition import PCA

from sklearn.cluster import MiniBatchKMeans

import matplotlib.pyplot as plt

import seaborn as sns

import pandas as pd

# Step 1: Load the Iris dataset

iris = load\_iris()

X = iris.data # Features (input data)

y = iris.target # Actual species labels

# Step 2: Perform PCA (optional for visualization purposes)

pca = PCA(n\_components=2)

X\_pca = pca.fit\_transform(X)

# Step 3: Apply Mini Batch K-Means with k=3

minibatch\_kmeans = MiniBatchKMeans(n\_clusters=3, random\_state=42, batch\_size=10)

minibatch\_kmeans.fit(X\_pca) # Fit Mini Batch K-Means to the PCA-reduced data

# Step 4: Get the cluster labels predicted by Mini Batch K-Means

labels = minibatch\_kmeans.labels\_

# Step 5: Visualize the clustering result

plt.scatter(X\_pca[:, 0], X\_pca[:, 1], c=labels, cmap='viridis')

plt.title('Mini Batch K-Means Clustering (k=3) on Iris Dataset (PCA-reduced)')

plt.xlabel('PC1')

plt.ylabel('PC2')

plt.show()

**Explanation of Code:**

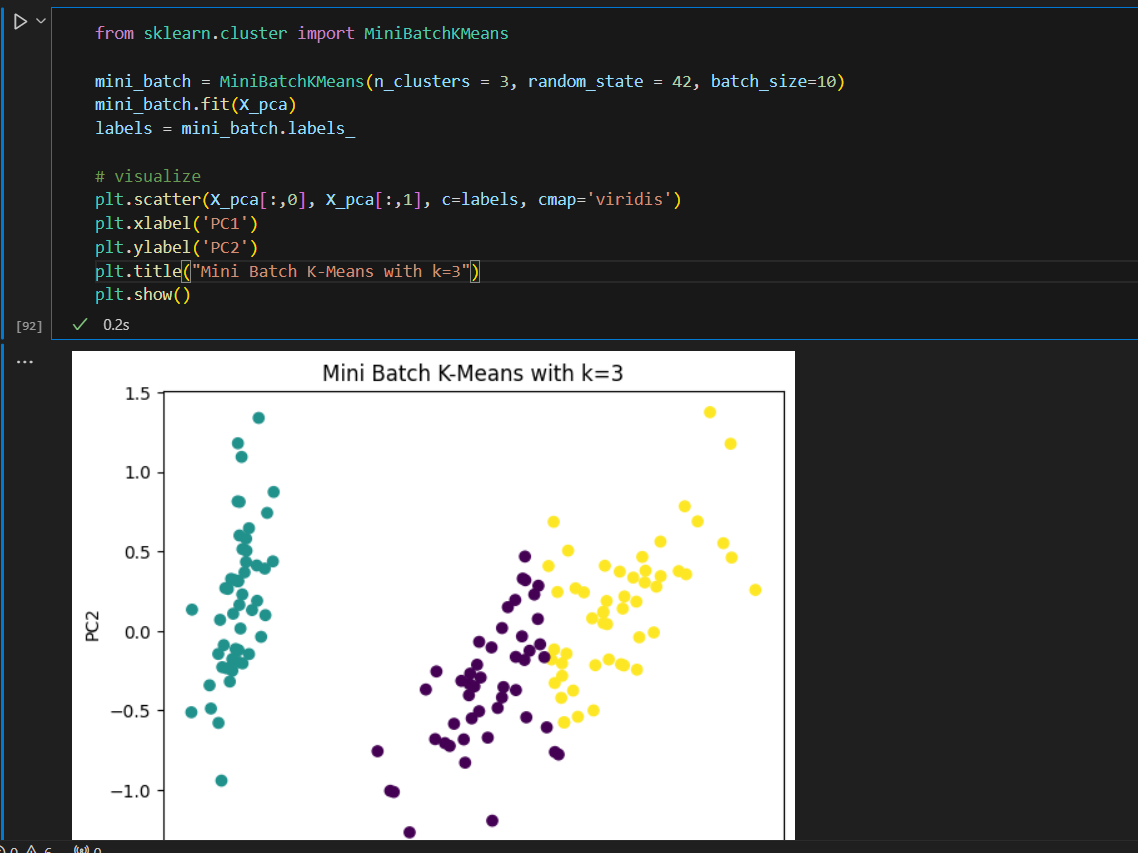
1. **Loading the Data**: We load the Iris dataset using load\_iris(). This gives us both the feature matrix X and the true species labels y.
2. **PCA**: We use PCA to reduce the dimensions of the dataset to 2 principal components for easier visualization.
3. **Mini Batch K-Means with k=3**:
   * We initialize **Mini Batch K-Means** with n\_clusters=3, which means the algorithm will try to divide the data into 3 clusters.
   * batch\_size=10 specifies that the algorithm will use small random batches of 10 data points to update the centroids at each iteration, reducing computational cost.
4. **Fitting the Model**: We fit the Mini Batch K-Means model to the PCA-reduced data (X\_pca).
5. **Get Cluster Labels**: The model predicts the clusters and returns labels for each sample in the dataset.
6. **Visualization**: We plot the PCA-reduced data with points colored according to their predicted clusters. Each color represents one of the three clusters.

**Benefits of Mini Batch K-Means:**

* **Efficiency**: It is faster and more scalable than standard K-Means, especially for large datasets, as it updates the centroids in batches.
* **Accuracy**: It often provides results similar to standard K-Means, but with less computation, especially when the dataset is large.

By using k=3, the Mini Batch K-Means algorithm will cluster the data into three groups, corresponding to the three species of the Iris dataset (Setosa, Versicolor, and Virginica).

Output





To apply **Affinity Propagation Clustering** to the Iris dataset, we will use the AffinityPropagation algorithm from the sklearn.cluster module. This method works by finding exemplars (representative data points) through a process of sending messages between data points based on their similarity.

Here’s how to apply **Affinity Propagation Clustering** step by step:

**Step-by-Step Code:**

python

Copy code

from sklearn.datasets import load\_iris

from sklearn.decomposition import PCA

from sklearn.cluster import AffinityPropagation

import matplotlib.pyplot as plt

import pandas as pd

import seaborn as sns

# Step 1: Load the Iris dataset

iris = load\_iris()

X = iris.data # Features (input data)

y = iris.target # Actual species labels

# Step 2: Perform PCA to reduce the dimensions (optional, for visualization)

pca = PCA(n\_components=2) # Reduce to 2 components for easy plotting

X\_pca = pca.fit\_transform(X)

# Step 3: Apply Affinity Propagation Clustering

affinity\_propagation = AffinityPropagation(random\_state=42)

affinity\_propagation.fit(X\_pca) # Fit the model to the PCA-reduced data

# Step 4: Get the cluster labels predicted by Affinity Propagation

labels = affinity\_propagation.labels\_

# Step 5: Visualize the clustering result

plt.scatter(X\_pca[:, 0], X\_pca[:, 1], c=labels, cmap='viridis')

plt.title('Affinity Propagation Clustering on Iris Dataset (PCA-reduced)')

plt.xlabel('PC1')

plt.ylabel('PC2')

plt.show()

**Explanation of the Code:**

1. **Loading the Data**: The Iris dataset is loaded using load\_iris() as usual, giving us the feature matrix X and the actual species labels y.
2. **PCA**: We apply **Principal Component Analysis (PCA)** to reduce the data to 2 principal components. This step is not strictly necessary for clustering but helps visualize the clustering result on a 2D plane.
3. **Affinity Propagation Clustering**:
   * We initialize the **AffinityPropagation** algorithm without specifying any particular parameters (you can tweak preferences and damping if needed).
   * random\_state=42 ensures that the results are reproducible.
   * The algorithm is fitted to the PCA-reduced data (X\_pca).
4. **Get Cluster Labels**: The clustering algorithm assigns each point in the dataset to one of the clusters. The labels (cluster assignments) are stored in labels.
5. **Visualization**: We plot the PCA-reduced data, coloring the points based on the cluster they belong to. Each color represents a different cluster.

**How Affinity Propagation Works:**

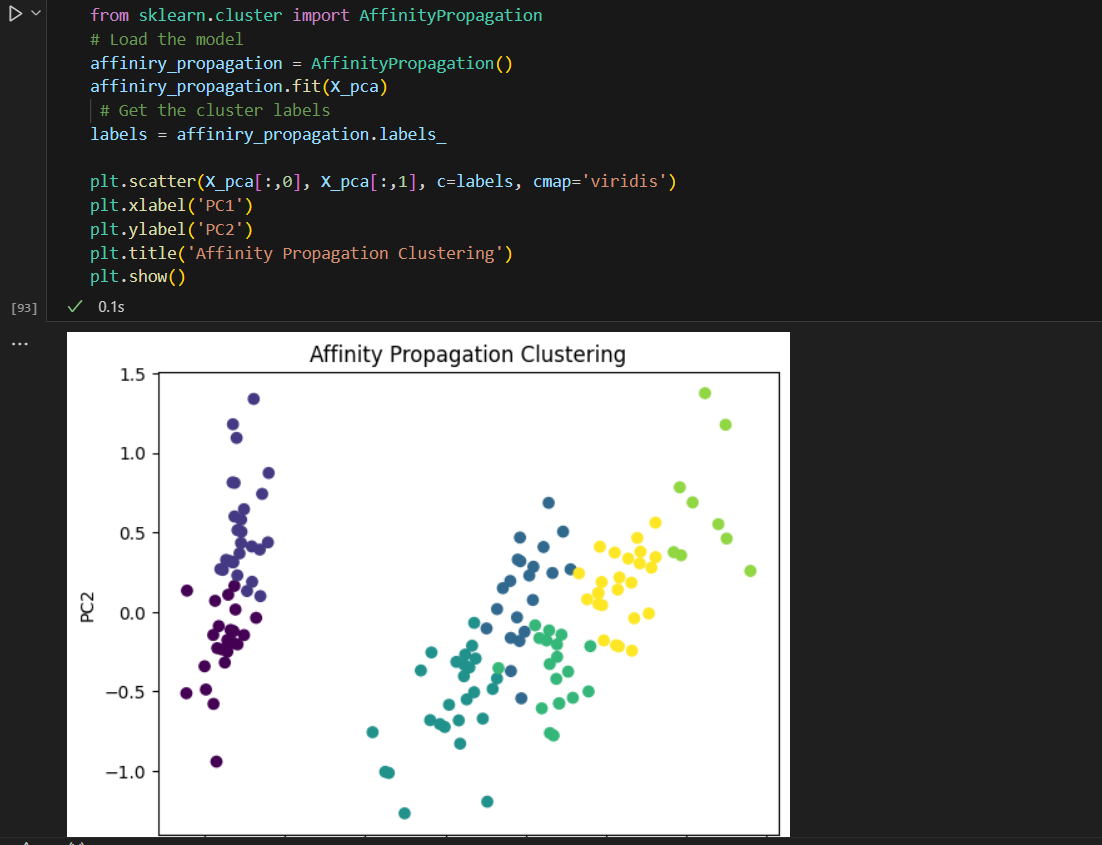
* **Similarity Matrix**: Affinity Propagation doesn't require you to specify the number of clusters (k) beforehand. Instead, it works by building a similarity matrix based on the pairwise similarity between data points (e.g., negative squared Euclidean distance).
* **Message Passing**: Data points exchange two types of messages:
  + **Responsibility**: How well-suited a point is to serve as an exemplar (cluster center) for another point.
  + **Availability**: How appropriate it is for a point to choose another point as its exemplar.
* **Convergence**: After several iterations of message passing, the algorithm identifies a set of exemplars and forms clusters around these exemplars.

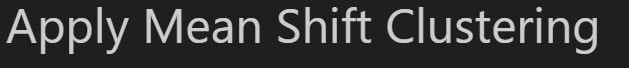
**When to Use Affinity Propagation:**

* **No Predefined Number of Clusters**: Unlike K-Means, you don’t need to specify the number of clusters (k), which is useful when the number of clusters is unknown.
* **Automatic Detection of Clusters**: The algorithm automatically determines the number of clusters based on the data.

However, this method can be more computationally intensive than other clustering methods like K-Means, especially for large datasets.

Output





Here’s how you can apply **Mean Shift Clustering** to the Iris dataset, step by step:

**Step-by-Step Code:**

python

Copy code

from sklearn.datasets import load\_iris

from sklearn.decomposition import PCA

from sklearn.cluster import MeanShift

import matplotlib.pyplot as plt

import pandas as pd

import seaborn as sns

# Step 1: Load the Iris dataset

iris = load\_iris()

X = iris.data # Features (input data)

y = iris.target # Actual species labels

# Step 2: Perform PCA to reduce the dimensions (optional, for visualization)

pca = PCA(n\_components=2) # Reduce to 2 components for easy plotting

X\_pca = pca.fit\_transform(X)

# Step 3: Apply Mean Shift Clustering

mean\_shift = MeanShift()

mean\_shift.fit(X\_pca) # Fit the model to the PCA-reduced data

# Step 4: Get the cluster labels predicted by Mean Shift

labels = mean\_shift.labels\_

# Step 5: Visualize the clustering result

plt.scatter(X\_pca[:, 0], X\_pca[:, 1], c=labels, cmap='viridis')

plt.title('Mean Shift Clustering on Iris Dataset (PCA-reduced)')

plt.xlabel('PC1')

plt.ylabel('PC2')

plt.show()

# Step 6: Optional - Display the cluster centers

cluster\_centers = mean\_shift.cluster\_centers\_

plt.scatter(X\_pca[:, 0], X\_pca[:, 1], c=labels, cmap='viridis')

plt.scatter(cluster\_centers[:, 0], cluster\_centers[:, 1], s=300, c='red', marker='X')

plt.title('Mean Shift Clustering with Cluster Centers')

plt.xlabel('PC1')

plt.ylabel('PC2')

plt.show()

**Explanation of the Code:**

1. **Loading the Data**: As usual, we load the Iris dataset using load\_iris() to get the feature matrix X and the species labels y.
2. **PCA**: We apply **Principal Component Analysis (PCA)** to reduce the data to 2 principal components. This step helps us visualize the data in 2D space, which is more intuitive than the original 4D space.
3. **Mean Shift Clustering**:
   * We initialize the **MeanShift** clustering algorithm with default parameters.
   * The algorithm is fitted to the PCA-reduced data (X\_pca). Unlike K-Means, **Mean Shift** doesn’t require you to specify the number of clusters upfront; it automatically determines the optimal number of clusters based on the data.
4. **Get Cluster Labels**: The algorithm assigns each data point to a cluster. The cluster labels are stored in labels, and we use these labels to color the data points based on which cluster they belong to.
5. **Visualization**: We use plt.scatter() to create a scatter plot of the PCA-reduced data, where each point is colored based on its assigned cluster. This allows us to visualize how Mean Shift has clustered the Iris dataset.
6. **Cluster Centers (Optional)**: In the second plot, we display the cluster centers using red X markers, showing where the algorithm considers the center of each cluster to be.

**How Mean Shift Clustering Works:**

* **Kernel Density Estimation**: Mean Shift is a non-parametric clustering algorithm that finds clusters by estimating the density of data points in the feature space. It uses a sliding window (kernel) to move toward regions with higher data point density.
* **Bandwidth**: The bandwidth of the kernel (window size) determines how far the algorithm shifts the data points toward regions of higher density. A larger bandwidth results in fewer clusters, while a smaller bandwidth results in more clusters.
* **Automatic Cluster Detection**: Mean Shift doesn’t require you to specify the number of clusters. Instead, it groups data points based on their proximity to high-density areas.

**Advantages of Mean Shift:**

* **No need to specify number of clusters**: Unlike K-Means, where you have to specify k, Mean Shift determines the number of clusters automatically.
* **Non-parametric**: It doesn't make assumptions about the number or shape of the clusters, making it more flexible than K-Means or other algorithms that assume spherical clusters.

**Disadvantages:**

* **Computationally expensive**: Mean Shift can be slower for large datasets compared to other algorithms like K-Means, especially when the bandwidth is small.

**Output**



Here’s how you can apply **Spectral Clustering** to the Iris dataset step by step:

**Step-by-Step Code:**

python

Copy code

from sklearn.datasets import load\_iris

from sklearn.decomposition import PCA

from sklearn.cluster import SpectralClustering

import matplotlib.pyplot as plt

import pandas as pd

import seaborn as sns

# Step 1: Load the Iris dataset

iris = load\_iris()

X = iris.data # Features (input data)

y = iris.target # Actual species labels

# Step 2: Perform PCA to reduce the dimensions (optional, for visualization)

pca = PCA(n\_components=2) # Reduce to 2 components for easy plotting

X\_pca = pca.fit\_transform(X)

# Step 3: Apply Spectral Clustering

spectral\_clustering = SpectralClustering(n\_clusters=3, affinity='nearest\_neighbors', random\_state=42)

labels = spectral\_clustering.fit\_predict(X\_pca) # Fit and predict the clusters

# Step 4: Visualize the clustering result

plt.scatter(X\_pca[:, 0], X\_pca[:, 1], c=labels, cmap='viridis')

plt.title('Spectral Clustering on Iris Dataset (PCA-reduced)')

plt.xlabel('PC1')

plt.ylabel('PC2')

plt.show()

**Explanation of the Code:**

1. **Loading the Data**: We use the load\_iris() function from sklearn.datasets to load the Iris dataset, as in previous steps. X holds the feature data, and y holds the actual species labels.
2. **PCA**: We apply **Principal Component Analysis (PCA)** to reduce the Iris data from 4 dimensions to 2 dimensions. This step helps with visualizing the clustering output in 2D space.
3. **Spectral Clustering**:
   * We initialize the **SpectralClustering** class from sklearn.cluster.
   * We set n\_clusters=3 because we know the Iris dataset has 3 species/classes.
   * The affinity='nearest\_neighbors' parameter tells the algorithm to use a nearest-neighbor approach to compute similarities between points.
   * The fit\_predict() method is applied to the PCA-reduced data X\_pca. It computes the clusters and assigns labels to each data point, storing the cluster assignments in labels.
4. **Visualization**:
   * The scatter plot is created using plt.scatter(), where each point represents an Iris sample, and its color is based on the cluster assigned by the Spectral Clustering algorithm.
   * The c=labels argument ensures that the colors of the points correspond to their cluster assignments.
   * The plot is labeled with axes PC1 and PC2, representing the two principal components after PCA reduction.

**How Spectral Clustering Works:**

* **Graph-Based Approach**: Spectral Clustering works by constructing a similarity graph from the data. It then partitions this graph into clusters by solving an eigenvalue problem.
* **Affinity Matrix**: The affinity matrix represents the similarities between data points. In this example, the nearest-neighbors method is used to compute the affinity matrix, meaning each data point is connected to its closest neighbors.
* **Eigenvectors**: The algorithm computes the eigenvectors of the affinity matrix and uses these to cluster the data in a transformed space.
* **Number of Clusters**: You need to specify the number of clusters (n\_clusters=3) for Spectral Clustering. However, the method itself is not limited to spherical clusters like K-Means and can handle more complex cluster shapes.

**Advantages:**

* **Can capture non-linear cluster structures**: Spectral Clustering is more flexible than K-Means in detecting arbitrarily shaped clusters.
* **Graph-based**: It’s based on the properties of the similarity graph, which can capture complex relationships between data points.

**Disadvantages:**

* **Requires the number of clusters upfront**: Similar to K-Means, you still need to specify the number of clusters beforehand.
* **Computationally intensive**: It requires computing the eigenvalues of the similarity matrix, which can be expensive for large datasets.

This code applies Spectral Clustering to the PCA-reduced Iris dataset and visualizes the clusters. If you want to change the number of clusters, simply modify n\_clusters=3 to another value!



Here's how you can apply **Agglomerative Clustering** on the PCA-transformed Iris dataset:

**Step-by-Step Code for Agglomerative Clustering**

python

Copy code

# Import the necessary library

from sklearn.cluster import AgglomerativeClustering

# Load and train the model

agglomerative\_clustering = AgglomerativeClustering(n\_clusters=3)

labels = agglomerative\_clustering.fit\_predict(X\_pca)

# Plot the results

plt.scatter(X\_pca[:, 0], X\_pca[:, 1], c=labels, cmap='viridis')

plt.xlabel('PC1')

plt.ylabel('PC2')

plt.title('Agglomerative Clustering')

plt.show()

**Code Explanation:**

1. **Import the model**:
   * AgglomerativeClustering is imported from sklearn.cluster. This is a hierarchical clustering algorithm that merges clusters iteratively based on distance.
2. **Define and train the model**:
   * We initialize the AgglomerativeClustering model with n\_clusters=3, meaning we're looking to cluster the data into 3 groups (you can adjust the number of clusters as needed).
   * The fit\_predict method trains the model on the data X\_pca and returns the cluster labels for each data point.
3. **Plot the results**:
   * We use a scatter plot to visualize the clusters. X\_pca[:, 0] and X\_pca[:, 1] are the two principal components from PCA, and the points are colored according to the cluster labels returned by the model.

This will show the clustered data points as per Agglomerative Clustering using the first two principal components.