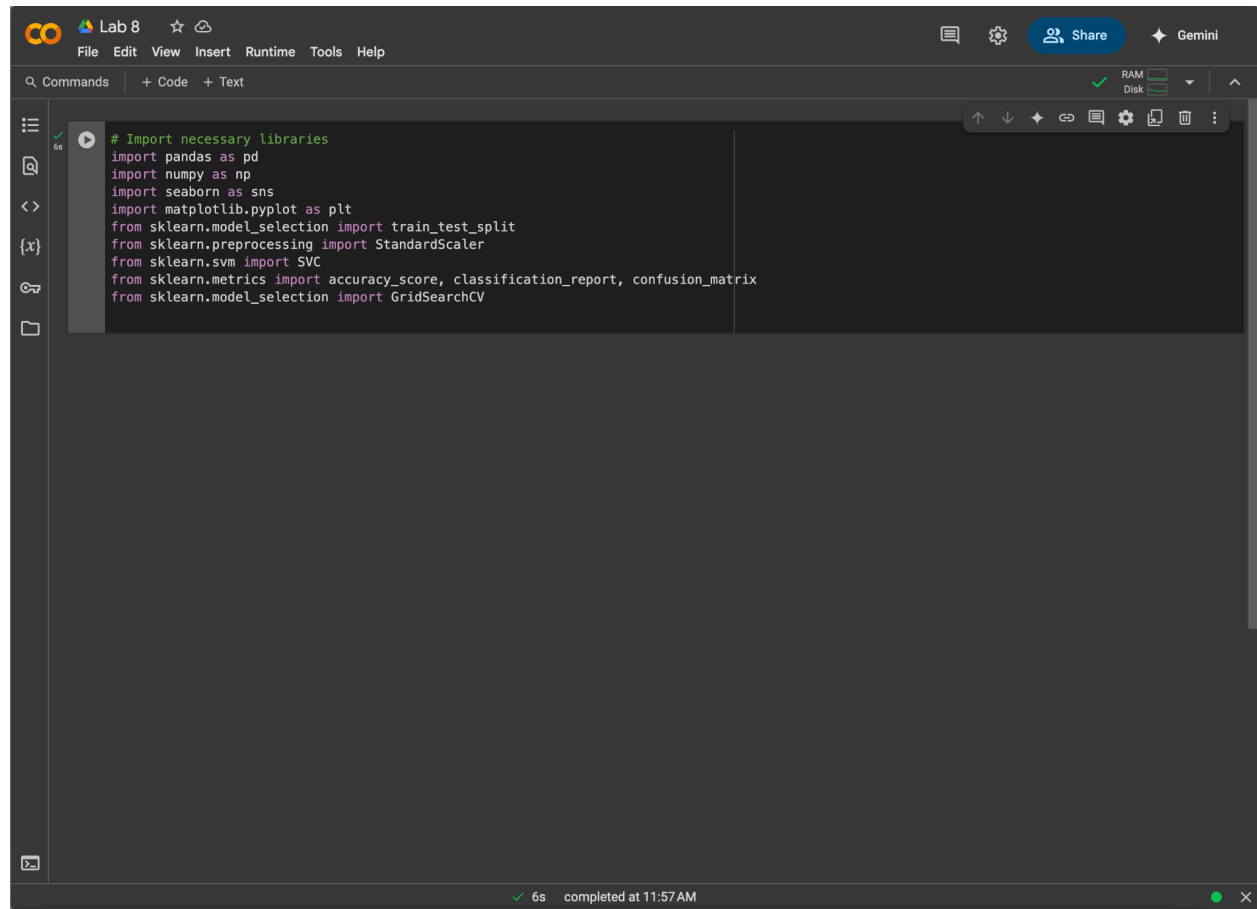


STEP 1: Environment Setup



The screenshot shows a JupyterLab environment with a dark theme. The top bar includes the 'Lab 8' title, a star icon, and a cloud icon. Below this is a menu bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. A search bar labeled 'Commands' is on the left, and a '+ Code + Text' button is next to it. On the right of the top bar are icons for a chat window, settings, a 'Share' button, and a 'Gemini' icon. Below the top bar, there are status indicators for 'RAM' and 'Disk' usage. The main area contains a code cell with a play button icon and a '6s' timer. The code in the cell is as follows:

```
# Import necessary libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.model_selection import GridSearchCV
```

At the bottom of the interface, a status bar shows a green checkmark, '6s', and 'completed at 11:57 AM'.

Figure 1: Importing necessary libraries

STEP 2: CritLoad and Explore Dataset

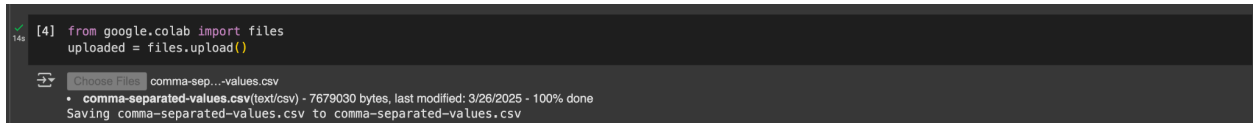


Figure 2: Importing comma-separated-values.csv from local machine

The screenshot shows a Google Colab interface. A code cell contains the following Python code:

```
import pandas as pd
df = pd.read_csv("comma-separated-values.csv")
df.head()
```

Below the code, the first five rows of the dataset are displayed in a table format:

	FileName	md5Hash	Machine	DebugSize	DebugRVA	MajorImageVersion	MajorOSVersion	ExportRVA	ExportSize
0	0124e21d-018c-4ce0-92a3-b9e205a76bc0.dll	79755c51e413ed3c6be4635d729a6e1	332	0	0	0	4	0	0
1	05c8318f98a5d301d80000009c316005.vertdll.dll	95e19f3657d34a432eada93221b0ea16	34404	84	121728	10	10	126576	4930
2	06054fba-5619-4a86-a861-ffb0464bef5d.dll	85c32641d77a54e19ba8ea4ab305c791	332	0	0	0	4	0	0
3	075822ac99a5d301660400009c316005.adhapi.dll	62e3b959d982ef534b66f819fe15f085	34404	84	19904	10	10	21312	252
4	090607dd9ba5d301ca0900009c316005.SensorsNative...	ae38c5f7d313ad0ff3bfb8826476767f	34404	84	97728	10	10	105792	1852

Figure 3: Loading file and displaying first couple of rows

I chose to upload the dataset file manually using the file upload option in Google Colab. After running the upload command, I selected the CSV file from my local machine and loaded it using pandas.

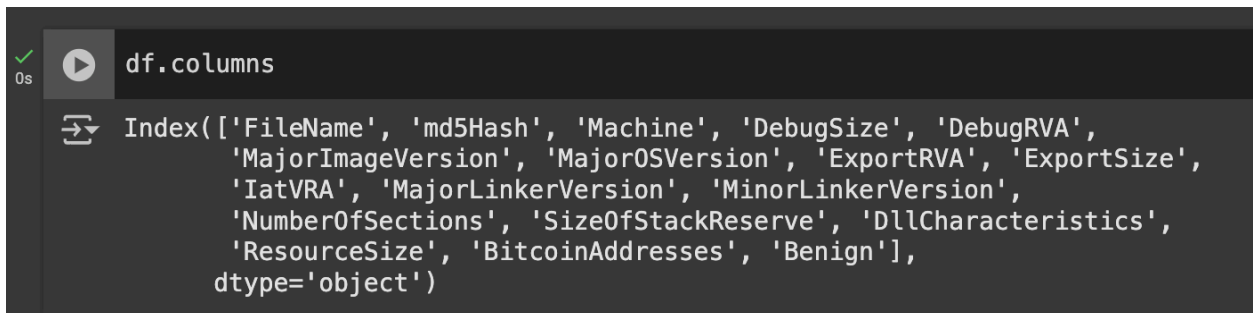


Figure 4: Using `df.columns` to look for columns that would fit malware/benign

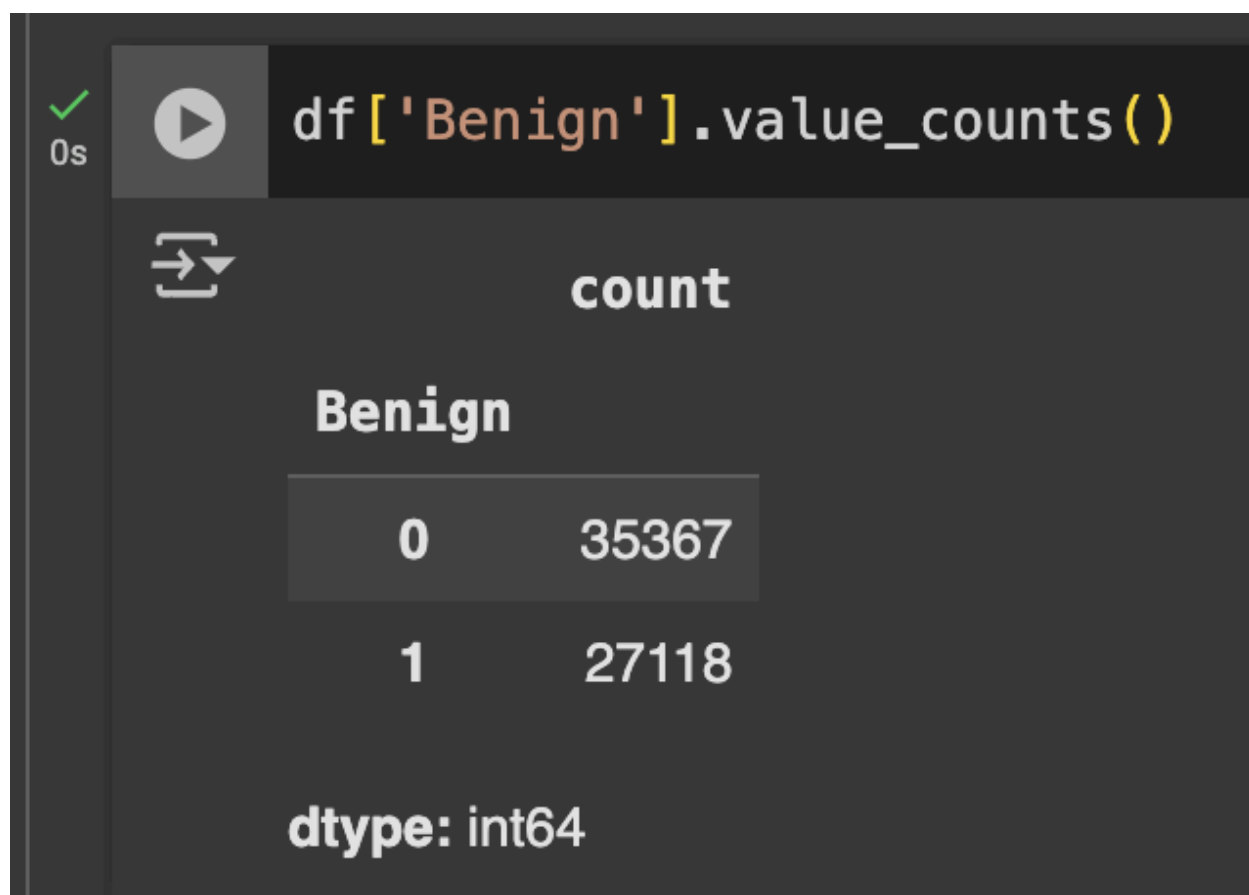


Figure 5: Identifying the target column using `df['Benign'].value_counts()`

The target column for classification is Benign. Present in that column is a binary representation of malware (represented by 0's) and benign files (represented by 1's).

```
[8] df.isnull().sum()

0
FileName      0
md5Hash       0
Machine       0
DebugSize     0
DebugRVA      0
MajorImageVersion 0
MajorOSVersion 0
ExportRVA     0
ExportSize    0
IatVRA        0
MajorLinkerVersion 0
MinorLinkerVersion 0
NumberOfSections 0
SizeOfStackReserve 0
DllCharacteristics 0
ResourceSize  0
BitcoinAddresses 0
Benign        0

dtype: int64
```

Figure 6: Checking to see if any null values are present

After running `df.isnull().sum()`, I found that none of the columns in the dataset have missing values. Therefore, no additional cleaning was necessary at this stage.

```
print(df['Benign'].dtype)
print(df['Benign'].value_counts())

int64
Benign
0      35367
1      27118
Name: count, dtype: int64
```

Figure 7: Checking target column values and data type

The target column Benign has a data type of int64, which is appropriate for binary classification. The class distribution shows 35,367 malware samples and 27,118 benign samples, which is slightly imbalanced but still usable.

STEP 3: Data Processing

```
df = df.drop(columns=['FileName', 'md5Hash'], errors='ignore')

y = df['Benign']
X = df.drop(columns=['Benign'])

X = X.apply(pd.to_numeric, errors='coerce')

X = X.fillna(X.median())

print("Preprocessing Step 3.1 complete.")
```

Preprocessing Step 3.1 complete.

Figure 8: Completing feature selection and cleanup

I dropped the columns `FileName` and `md5Hash` from the dataset as they were non numeric. I then separated the features (`X`) and target column (`Benign`) into `X` and `y`. All remaining features were confirmed to be numeric, and any missing values were filled with the median of each column.

```
from sklearn.model_selection import train_test_split

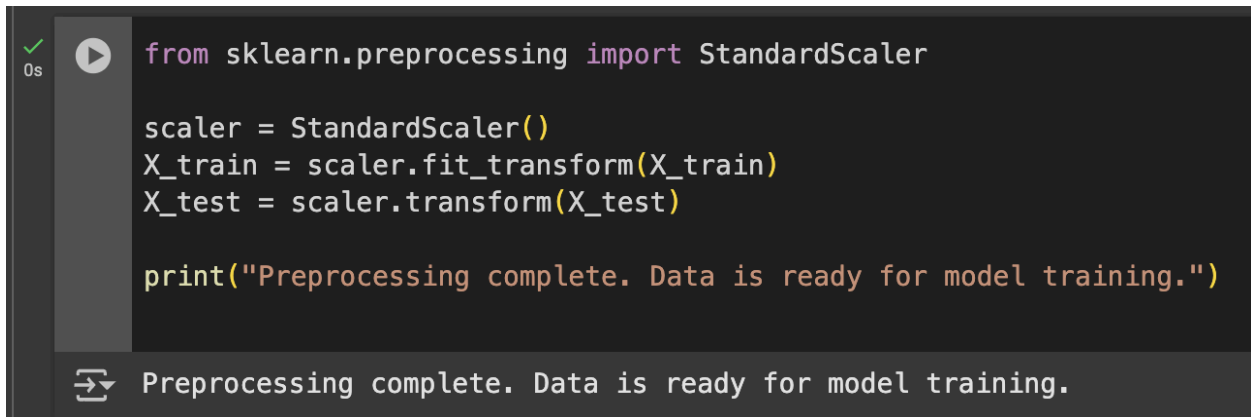
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

print("Shapes:")
print("X_train:", X_train.shape)
print("X_test:", X_test.shape)
print("y_train:", y_train.shape)
print("y_test:", y_test.shape)
```

Shapes:
X_train: (49988, 15)
X_test: (12497, 15)
y_train: (49988,)
y_test: (12497,)

Figure 9: Splitting the data

I split the data into 80% training and 20% testing using `train_test_split`. This allows the model to train on one portion of the data and evaluate on unseen data. I also set `random_state=42` to ensure consistent splits across runs. I used `train_test_split` to divide the dataset into training and testing sets with an 80/20 split. The results can be seen under 'Shapes:' in Figure 9. This ensures that the model is trained on the majority of the data and tested on a representative sample.



```
from sklearn.preprocessing import StandardScaler

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

print("Preprocessing complete. Data is ready for model training.")
```

Preprocessing complete. Data is ready for model training.

Figure 10: Standardizing the features

I used `StandardScaler()` to standardize the feature data. This scales all features to have a mean of 0 and a standard deviation of 1, which helps many machine learning models perform better by treating all features equally regardless of their original range. I used `fit_transform()` on the training set and `transform()` on the test set to avoid data leakage.

STEP 4: Train an SVM Classifier (Linear Kernel)

```
✓ 1m ▶ from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report

svm_linear = SVC(kernel='linear')
svm_linear.fit(X_train, y_train)

y_pred = svm_linear.predict(X_test)

print("SVM Linear Kernel Accuracy:", accuracy_score(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

➞ SVM Linear Kernel Accuracy: 0.8630871409138193

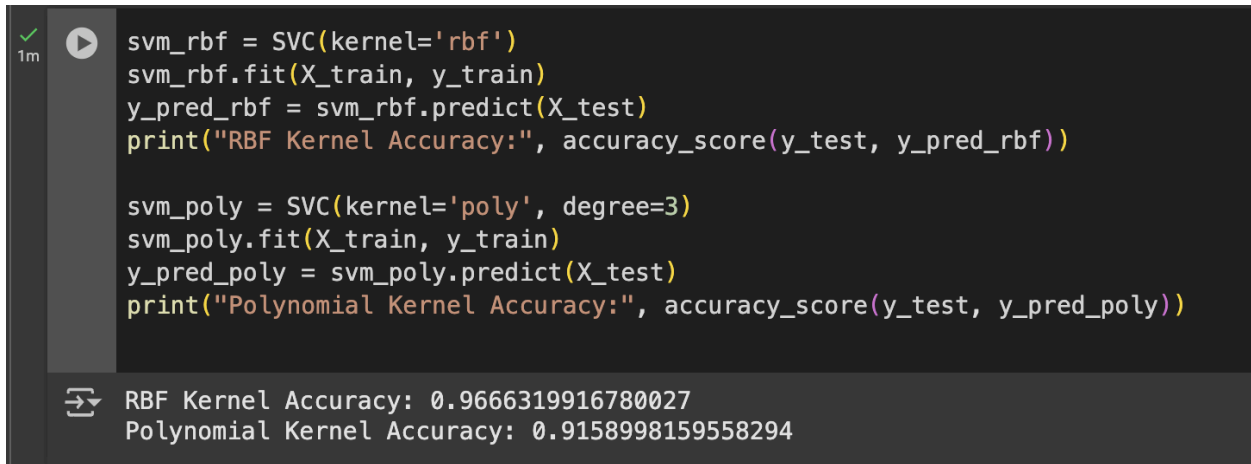
Classification Report:

	precision	recall	f1-score	support
0	0.82	0.97	0.89	7125
1	0.95	0.72	0.82	5372
accuracy			0.86	12497
macro avg	0.88	0.85	0.85	12497
weighted avg	0.88	0.86	0.86	12497

Figure 11: Training SVM Linear Kernel and Evaluate

I trained an SVM classifier using a linear kernel, which achieved an accuracy of approximately 86.31% on the test set (this percentage was found by multiplying the SVM Linear Kernel Accuracy (0.8630871409138193) by 100). The classification report shows the model is very good at identifying malware (high recall for class 0), but it occasionally misclassifies benign files as malware (lower recall for class 1). The overall weighted F1-score is 0.86, indicating balanced performance across both classes. When looking at false positives and false negatives, we can identify them as two things. A false positive means a benign file was incorrectly classified as malware. This is inconvenient but safer than the alternative. A false negative means malware was incorrectly classified as benign, which is a more severe issue since it can lead to undetected infections. The results show the model has few false negatives for malware (high recall for class 0), making it suitable for security applications where catching malware is the priority, even at the cost of some false positives.

STEP 5: Experiment with Different Kernels



A screenshot of a Jupyter Notebook cell. The cell contains two blocks of Python code. The first block trains an SVM with the RBF kernel, and the second block trains an SVM with the Polynomial kernel (degree=3). Both blocks use the same training and testing data (X_train, y_train, X_test, y_test). The output of the cell shows the accuracy scores for both models. The RBF kernel achieved an accuracy of approximately 0.9666, while the Polynomial kernel achieved an accuracy of approximately 0.9159.

```
svm_rbf = SVC(kernel='rbf')
svm_rbf.fit(X_train, y_train)
y_pred_rbf = svm_rbf.predict(X_test)
print("RBF Kernel Accuracy:", accuracy_score(y_test, y_pred_rbf))

svm_poly = SVC(kernel='poly', degree=3)
svm_poly.fit(X_train, y_train)
y_pred_poly = svm_poly.predict(X_test)
print("Polynomial Kernel Accuracy:", accuracy_score(y_test, y_pred_poly))
```

RBF Kernel Accuracy: 0.9666319916780027
Polynomial Kernel Accuracy: 0.9158998159558294

Figure 12: Train and Evaluate RBF & Polynomial Kernels

I compared three different SVM kernels on the same dataset: Linear Kernel with accuracy at 86.31%, RBF Kernel with accuracy at 96.66%, and Polynomial Kernel with accuracy at 91.59%. The RBF kernel performed the best, achieving the highest accuracy. This suggests the data has non-linear patterns that RBF can model better than a linear decision boundary. The Linear kernel performed the worst, likely due to its simplicity and inability to capture complex relationships in the feature space. The Polynomial kernel performed better than linear but not as well as RBF which is possibly due to overfitting.

STEP 6: Model Comparison (Trying Multiple Classifiers)

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, accuracy_score
rf_model = RandomForestClassifier()
knn_model = KNeighborsClassifier()
lr_model = LogisticRegression(max_iter=1000)
models = [rf_model, knn_model, lr_model]
for model in models:
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    print(f"\nModel: {model.__class__.__name__}")
    print(f"Accuracy: {accuracy_score(y_test, y_pred)}")
    print(classification_report(y_test, y_pred))
    print("-" * 50)
```

Model: RandomForestClassifier
Accuracy: 0.996959270224854

	precision	recall	f1-score	support
0	1.00	1.00	1.00	7125
1	1.00	1.00	1.00	5372
accuracy			1.00	12497
macro avg	1.00	1.00	1.00	12497
weighted avg	1.00	1.00	1.00	12497

Model: KNeighborsClassifier
Accuracy: 0.9917580219252621

	precision	recall	f1-score	support
0	0.99	0.99	0.99	7125
1	0.99	0.99	0.99	5372
accuracy			0.99	12497
macro avg	0.99	0.99	0.99	12497
weighted avg	0.99	0.99	0.99	12497

Model: LogisticRegression
Accuracy: 0.8726894454669121

	precision	recall	f1-score	support
0	0.85	0.94	0.89	7125
1	0.91	0.78	0.84	5372
accuracy			0.87	12497
macro avg	0.88	0.86	0.87	12497
weighted avg	0.88	0.87	0.87	12497

I also evaluated the performance of Random Forest with accuracy at 99.70%, KNeighbors with accuracy at 99.18%, and Logistic Regression with accuracy at 87.27%. Random Forest performed the best. It achieved perfect precision, recall, and F1 score on both classes. This is likely due to its ensemble nature, where many decision trees reduce

overfitting and model complex patterns well. KNeighbors also performed very well, but slightly below Random Forest. While KNeighbors can work great with well-structured data, it can become computationally expensive as the dataset grows due to distance calculations. Logistic Regression had the lowest accuracy. This is expected since it's a linear model, and our dataset seems to contain non linear patterns that are better handled by tree-based or kernel-based classifiers. Overall, Random Forest outperformed all other models, including the SVM with RBF kernel. This suggests it is the most effective model for this malware detection task.