1. Short Answer Questions

Q1: Explain the primary differences between TensorFlow and PyTorch. When would you choose one over the other?

Primary Differences:

- **Computational Graph:**
    - **TensorFlow:** Primarily uses a **static computational graph**. This means the graph is defined first and then executed. This can offer benefits in terms of optimization for deployment and distributed computing once the graph is finalized. However, it can make debugging more challenging as changes require recompiling the graph.
    - **PyTorch:** Uses a **dynamic computational graph** (also known as "define-by-run"). The graph is built on the fly as operations are performed. This makes PyTorch more "Pythonic" and significantly easier to debug, as you can inspect tensors and graph operations during runtime.
- **Ease of Use/Pythonic Nature:**
    - **PyTorch:** Generally considered more Pythonic and easier for beginners to pick up due to its imperative programming style, which aligns well with standard Python programming. Debugging is often likened to debugging regular Python code.
    - **TensorFlow:** While TensorFlow 2.x with Keras has greatly improved its user-friendliness and adopted eager execution (dynamic graph-like behavior), its underlying architecture can still feel more rigid and less intuitive for those accustomed to pure Python.
- **Production Readiness:**
    - **TensorFlow:** Historically, TensorFlow has had a stronger reputation and more mature tools for production deployment (e.g., TensorFlow Serving, TensorFlow Lite). Its static graph allows for more aggressive optimizations and deployment to various platforms.
    - **PyTorch:** While PyTorch's production deployment capabilities have significantly improved with tools like TorchScript, TensorFlow still often has an edge in large-scale, enterprise-level deployments, especially when aiming for extreme optimization.
- **Community and Ecosystem:**

    - **TensorFlow:** Backed by Google, TensorFlow has a massive and established community with extensive resources, tutorials, and a vast ecosystem of related tools (TensorBoard, TF Serving, TF Lite, TF.js, etc.).
    - **PyTorch:** While newer, PyTorch has a rapidly growing and highly active community, especially popular in academic research due to its flexibility. Its ecosystem is expanding rapidly.

When to choose one over the other:

- **Choose PyTorch when:**
    - You are a beginner or prefer a more "Pythonic" and intuitive experience.
    - You are engaged in research and rapid prototyping, where flexibility, dynamic graph manipulation, and easier debugging are crucial.
    - You prioritize quick experimentation and iterating on model architectures.
    - You are working on projects with less stringent deployment requirements initially.
- **Choose TensorFlow when:**
    - You are building large-scale, production-ready applications that require robust deployment, scalability, and integration with various platforms (mobile, web, edge devices).
    - You need extensive tooling for monitoring, serving, and managing models in a production environment (MLOps).
    - You benefit from a larger, more mature ecosystem with more pre-built models and distributed training capabilities out-of-the-box for certain scenarios.
    - You are comfortable with a more defined-and-then-run workflow, or using Keras as a high-level API.

Q2: Describe two use cases for Jupyter Notebooks in AI development.

1. **Exploratory Data Analysis (EDA) and Preprocessing:** Jupyter Notebooks are invaluable for the initial stages of AI development. Data scientists can load datasets, visualize distributions, check for missing values, perform feature engineering, and apply various preprocessing steps (like normalization or encoding) interactively. The ability to execute code cell by cell, immediately see the output (including plots and dataframes), and add explanatory Markdown text makes it easy to understand the data, identify patterns, and

prepare it for model training. This iterative process is crucial for gaining insights into the data's characteristics.

2. **Rapid Prototyping and Model Experimentation:** Jupyter Notebooks excel at quickly building, training, and evaluating different machine learning models. Researchers and developers can define model architectures, train them on small subsets of data, and test different hyperparameter configurations in an iterative fashion. The immediate feedback loop provided by cell execution allows for quick adjustments and comparisons of model performance, accelerating the experimentation process. You can easily modify a model, retrain it, and visualize its performance metrics or predictions without needing to re-run the entire script, making it ideal for trying out various ideas efficiently.

Q3: How does spaCy enhance NLP tasks compared to basic Python string operations?

spaCy provides a powerful and efficient framework for Natural Language Processing (NLP) that goes far beyond basic Python string operations in several key ways:

1. **Linguistic Annotation and Structure:** Basic Python string operations (like split(), find(), replace()) treat text as a sequence of characters. spaCy, on the other hand, understands and annotates text with rich linguistic information. It performs:
    - **Tokenization:** Breaks text into meaningful units (tokens) while handling complex cases like contractions, punctuation, and hyphenated words.
    - **Part-of-Speech (POS) Tagging:** Identifies the grammatical role of each word (e.g., noun, verb, adjective).
    - **Dependency Parsing:** Reveals the grammatical relationships between words in a sentence, creating a parse tree.
    - **Named Entity Recognition (NER):** Identifies and categorizes named entities in text (e.g., persons, organizations, locations, dates).

This structured linguistic information is crucial for understanding the meaning and context of text, which is impossible with simple string manipulation.

2. **Efficiency and Performance (for production):** While you *could* write custom Python code using regular expressions to find patterns, it would be slow, error-prone, and difficult to maintain for complex NLP tasks. spaCy is written in Cython, making it significantly faster and more memory-efficient than pure Python for processing large volumes of text. It's designed for production-grade NLP, whereas custom string operations quickly become bottlenecks.

3. **Pre-trained Models and Transfer Learning:** spaCy comes with highly optimized, pre-trained statistical models for various languages. These models are trained on large corpora and are capable of performing tasks like POS tagging, dependency parsing, and NER out-of-the-box with high accuracy. This means you don't have to train these complex models from scratch, which is a massive advantage over trying to implement these functionalities with basic string operations. You can also fine-tune these models for specific domains.
4. **Rule-based Matching and Customization:** Beyond statistical models, spaCy offers powerful rule-based matching capabilities (Matcher, PhraseMatcher). This allows you to define patterns (based on tokens, POS tags, lemmas, etc.) to extract specific information or identify entities that might not be covered by statistical models. This hybrid approach (statistical + rule-based) provides a level of precision and control far beyond what simple string searches can offer.

In summary, spaCy elevates NLP from simple text manipulation to deep linguistic understanding, offering robust, efficient, and customizable tools that are essential for real-world NLP applications.

2. Comparative Analysis

Compare Scikit-learn and TensorFlow in terms of:

Target Applications:

- **Scikit-learn:** Primarily designed for **classical machine learning algorithms**. This includes a wide range of supervised learning (classification, regression), unsupervised learning (clustering, dimensionality reduction), model selection, and preprocessing tasks. It excels at tabular data problems and traditional statistical modeling.
    - *Examples:* Predicting house prices (regression), classifying emails as spam or not (classification), customer segmentation (clustering), reducing features in a dataset (PCA).
- **TensorFlow:** Primarily designed for **deep learning**, particularly neural networks. It provides a flexible framework for building, training, and deploying complex deep learning models, including Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), Transformers, and more. It's ideal for tasks involving unstructured data like images, audio, and large text corpora.
    - *Examples:* Image recognition, natural language understanding, speech recognition, generative AI models. While it can implement some classical ML

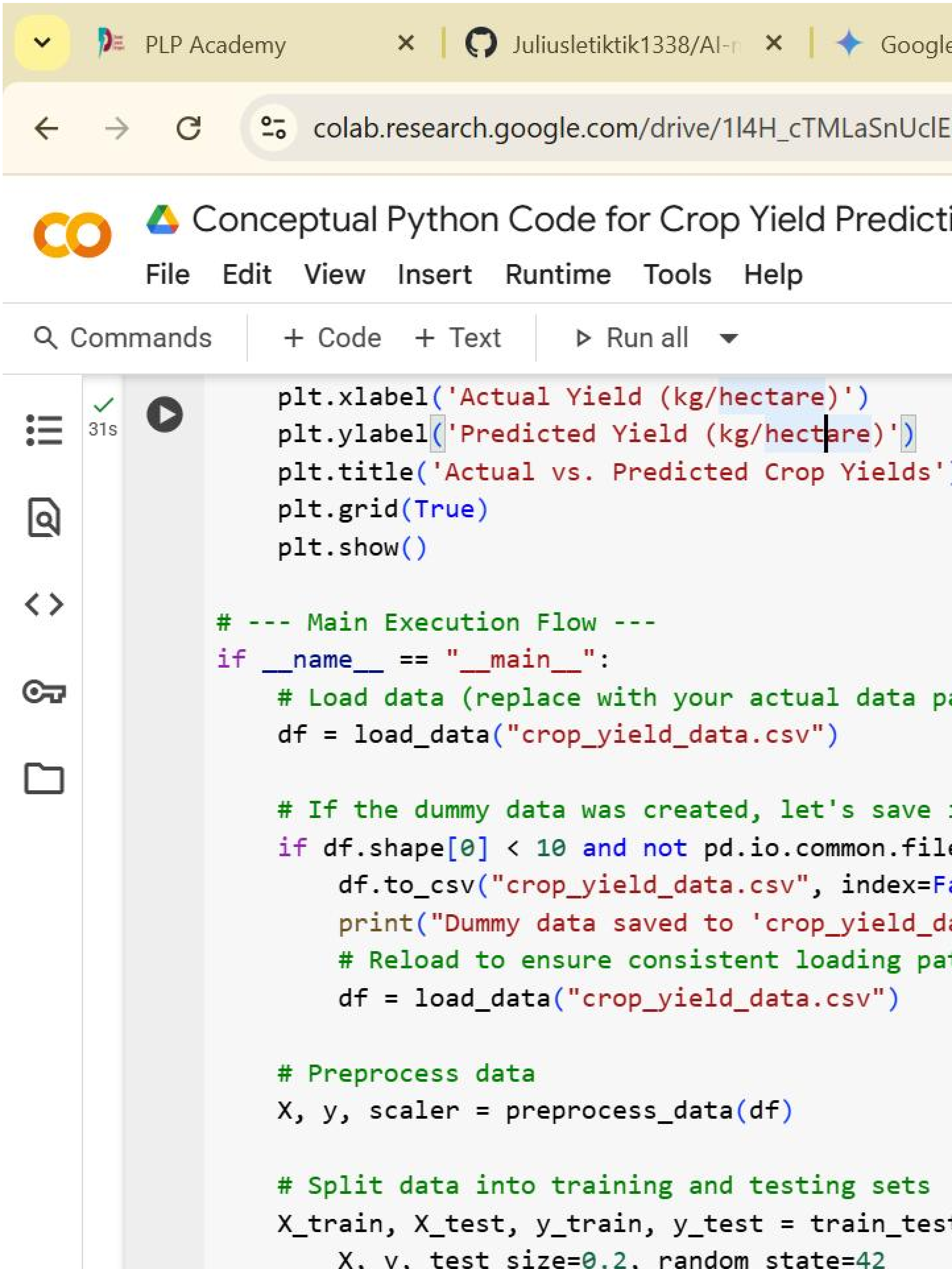
algorithms, it's generally overkill and less straightforward for them compared to scikit-learn.

Ease of Use for Beginners:

- **Scikit-learn:** Generally considered **very easy to use for beginners**. It features a consistent API for all its models (.fit(), .predict(), .transform()), making it intuitive to learn and apply different algorithms. The documentation is excellent, and many tutorials are available. Its focus on classical ML means less complex concepts (like backpropagation, gradients, complex architectures) are required to get started.
- **TensorFlow:** With the introduction of Keras as its high-level API in TensorFlow 2.x, it has become **significantly easier for beginners** than TensorFlow 1.x. Keras simplifies model building, training, and evaluation. However, understanding deep learning concepts (neural network layers, activation functions, optimizers, loss functions, debugging complex model architectures, handling GPUs) still presents a steeper learning curve compared to scikit-learn. While simple models are easy with Keras, delving into custom layers, complex training loops, or distributed training can quickly increase complexity.

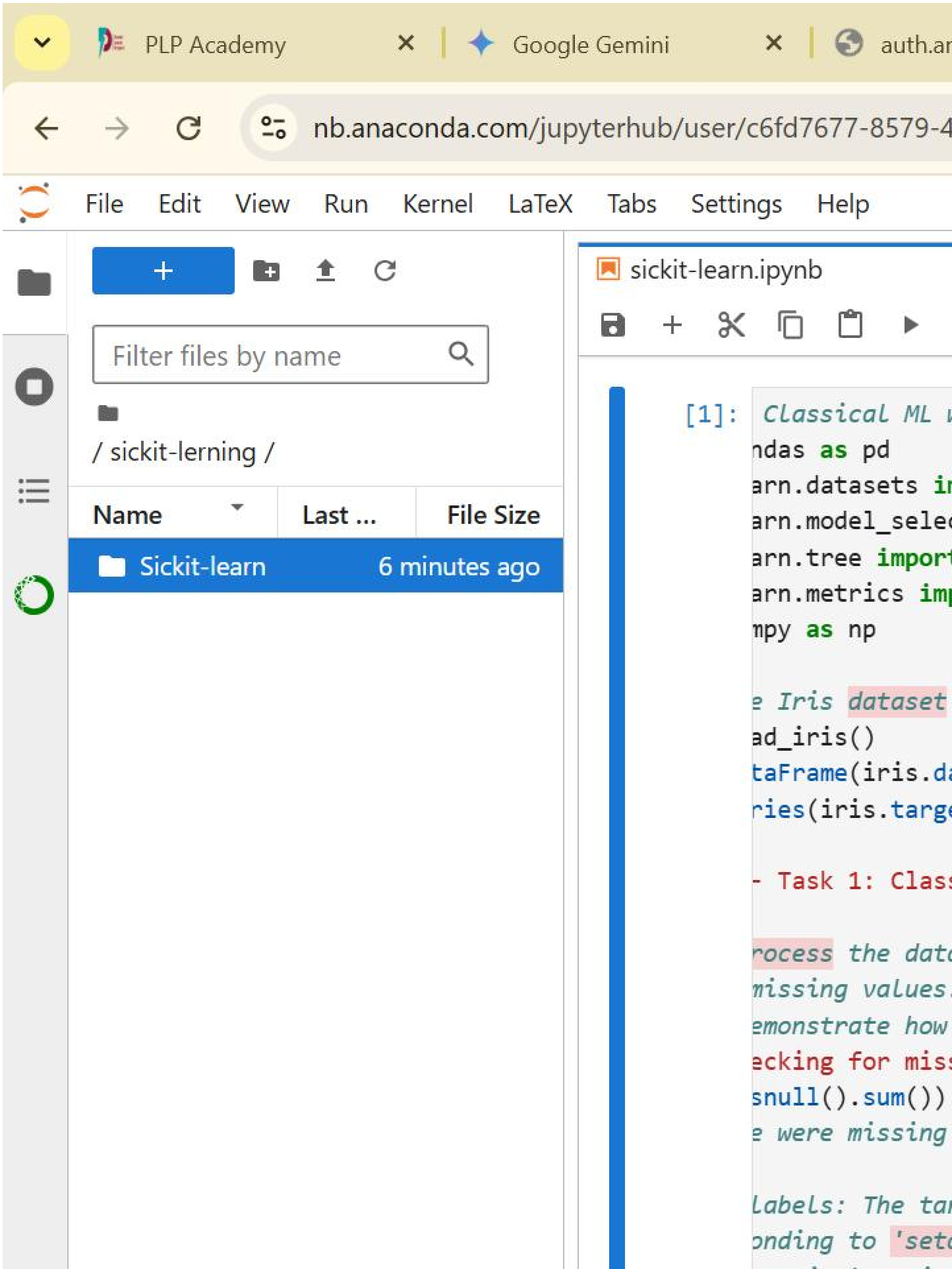
Community Support:

- **Scikit-learn:** Has a **large and mature community**. It's a widely adopted library in academia and industry for classical ML. You'll find extensive documentation, numerous tutorials, active forums (e.g., Stack Overflow), and a strong developer community contributing to its growth and maintenance.
- **TensorFlow:** Possesses an **enormous and highly active community**, arguably one of the largest in the AI/ML space. Being backed by Google, it benefits from significant corporate support and investment. Its community provides vast resources, official documentation, tutorials, a dedicated forum, and a continuous stream of updates and new tools. This makes it easy to find solutions to problems and stay updated with the latest advancements in deep learning.

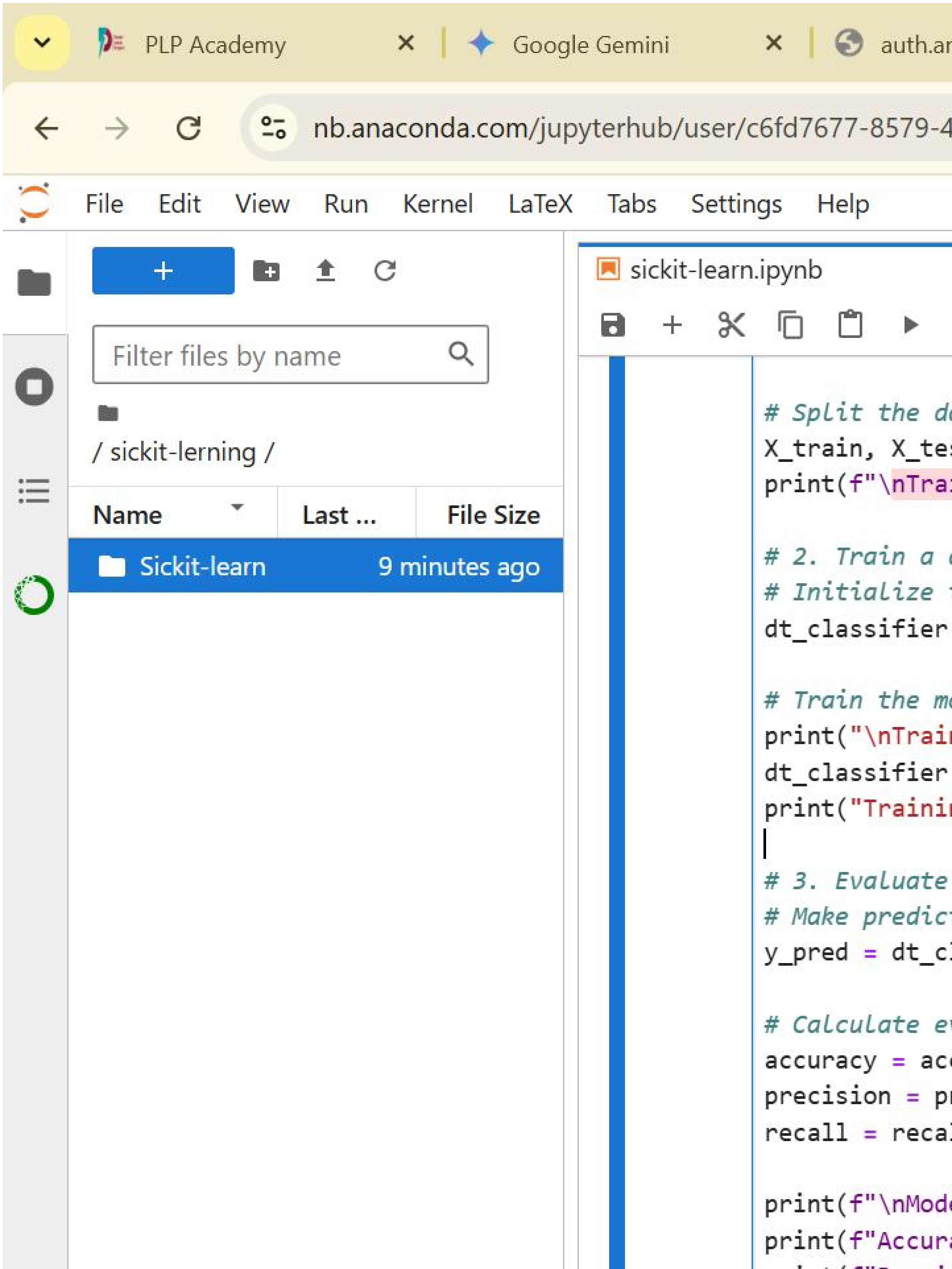


PLP Academy
Juliusletiktik1338/AI-r
Googl
colab.research.google.com/drive/1l4H_cTMLaSnUclE
Conceptual Python Code for Crop Yield Predicti
File Edit View Insert Runtime Tools Help
Commands
+ Code
+ Text
Run all
31s

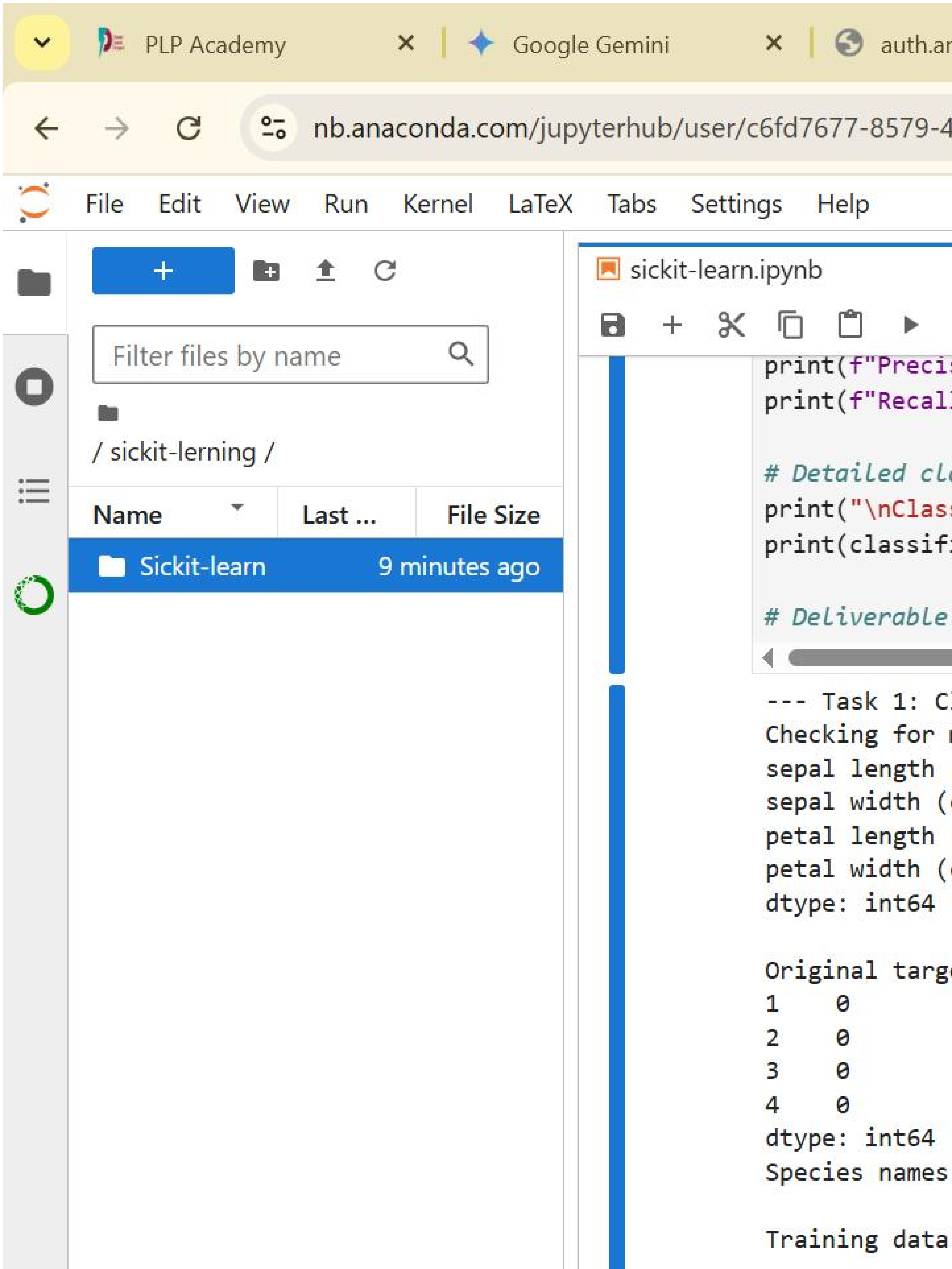
```
    plt.xlabel('Actual Yield (kg/hectare)')
    plt.ylabel('Predicted Yield (kg/hectare)')
    plt.title('Actual vs. Predicted Crop Yields')
    plt.grid(True)
    plt.show()

# --- Main Execution Flow ---
if __name__ == "__main__":
    # Load data (replace with your actual data pa
    df = load_data("crop_yield_data.csv")

    # If the dummy data was created, let's save i
    if df.shape[0] < 10 and not pd.io.common.file
        df.to_csv("crop_yield_data.csv", index=Fa
        print("Dummy data saved to 'crop_yield_da
        # Reload to ensure consistent loading pat
        df = load_data("crop_yield_data.csv")

    # Preprocess data
    X, y, scaler = preprocess_data(df)

    # Split data into training and testing sets
    X_train, X_test, y_train, y_test = train_tes
        X, y, test_size=0.2, random_state=42
```



```
[1]: Classical ML w
     ndas as pd
     arn.datasets im
     arn.model_selec
     arn.tree import
     arn.metrics imp
     mpy as np

     e Iris dataset
     ad_iris()
     taFrame(iris.da
     ries(iris.targe

     - Task 1: Class

     rocess the data
     missing values.
     emonstrate how
     ecking for miss
     snull().sum())
     e were missing

     labels: The tar
     onding to 'seto
```

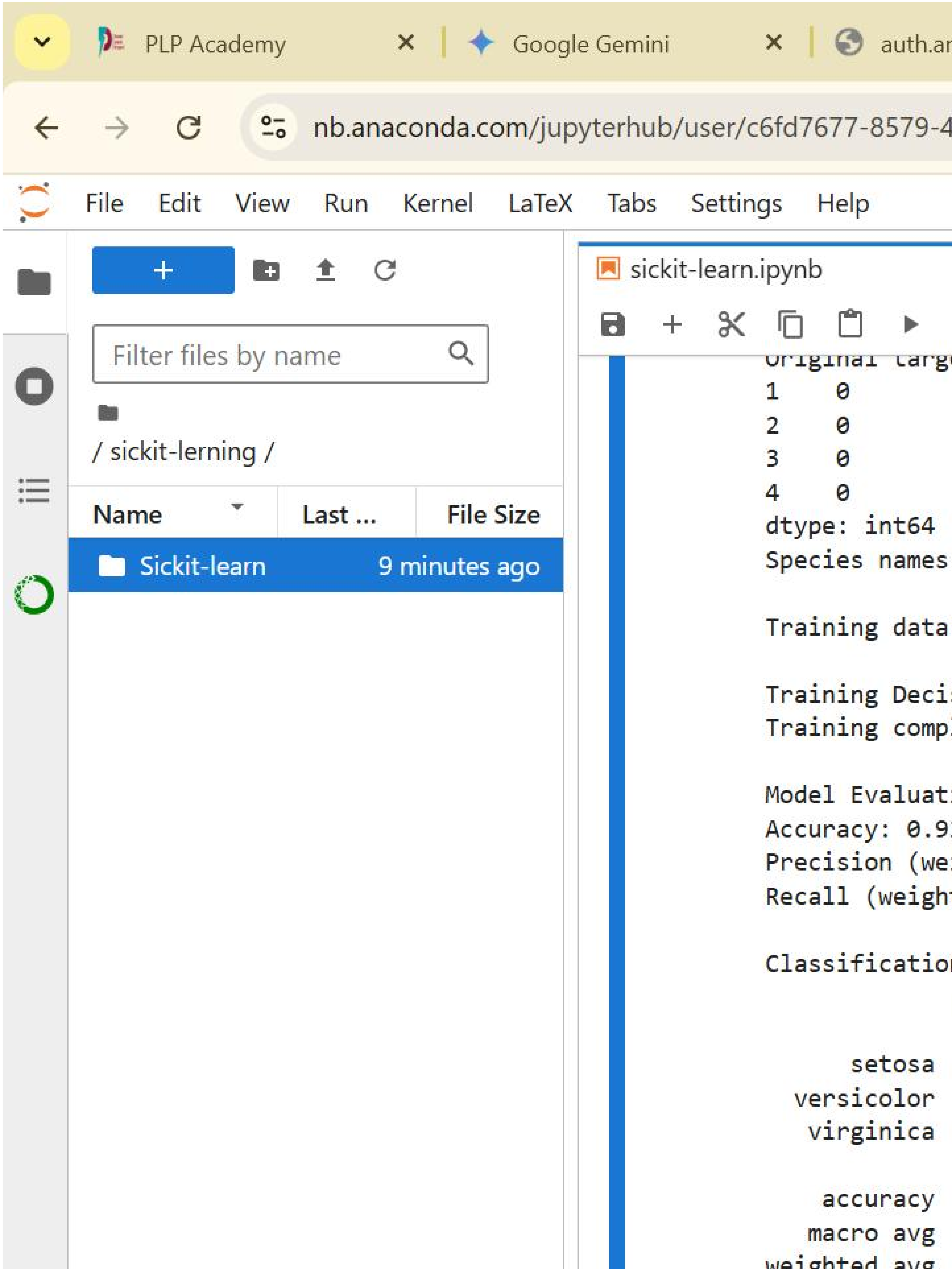


```
# Split the d
X_train, X_te
print(f"\nTra

# 2. Train a 
# Initialize 
dt_classifier

# Train the m
print("\nTrai
dt_classifier
print("Traini

# 3. Evaluate
# Make predic
y_pred = dt_c

# Calculate e
accuracy = ac
precision = p
recall = reca

print(f"\nMod
print(f"Accur
```



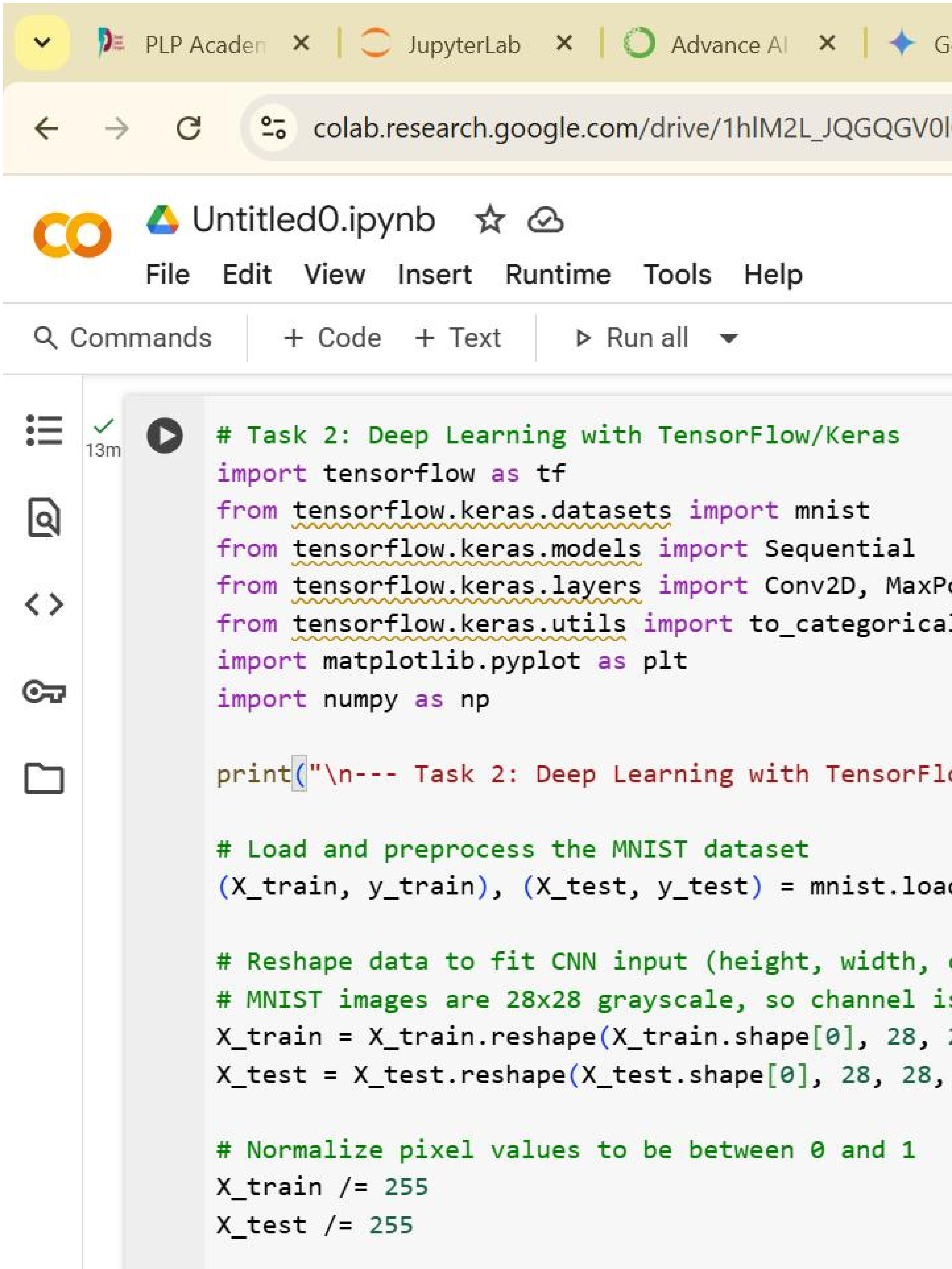
```
print(f"Preci
print(f"Recal

# Detailed cl
print("\nClas
print(classif

# Deliverable
```

```
--- Task 1: C
Checking for 
sepal length 
sepal width (
petal length 
petal width (
dtype: int64

Original targ
1    0
2    0
3    0
4    0
dtype: int64
Species names

Training data
```

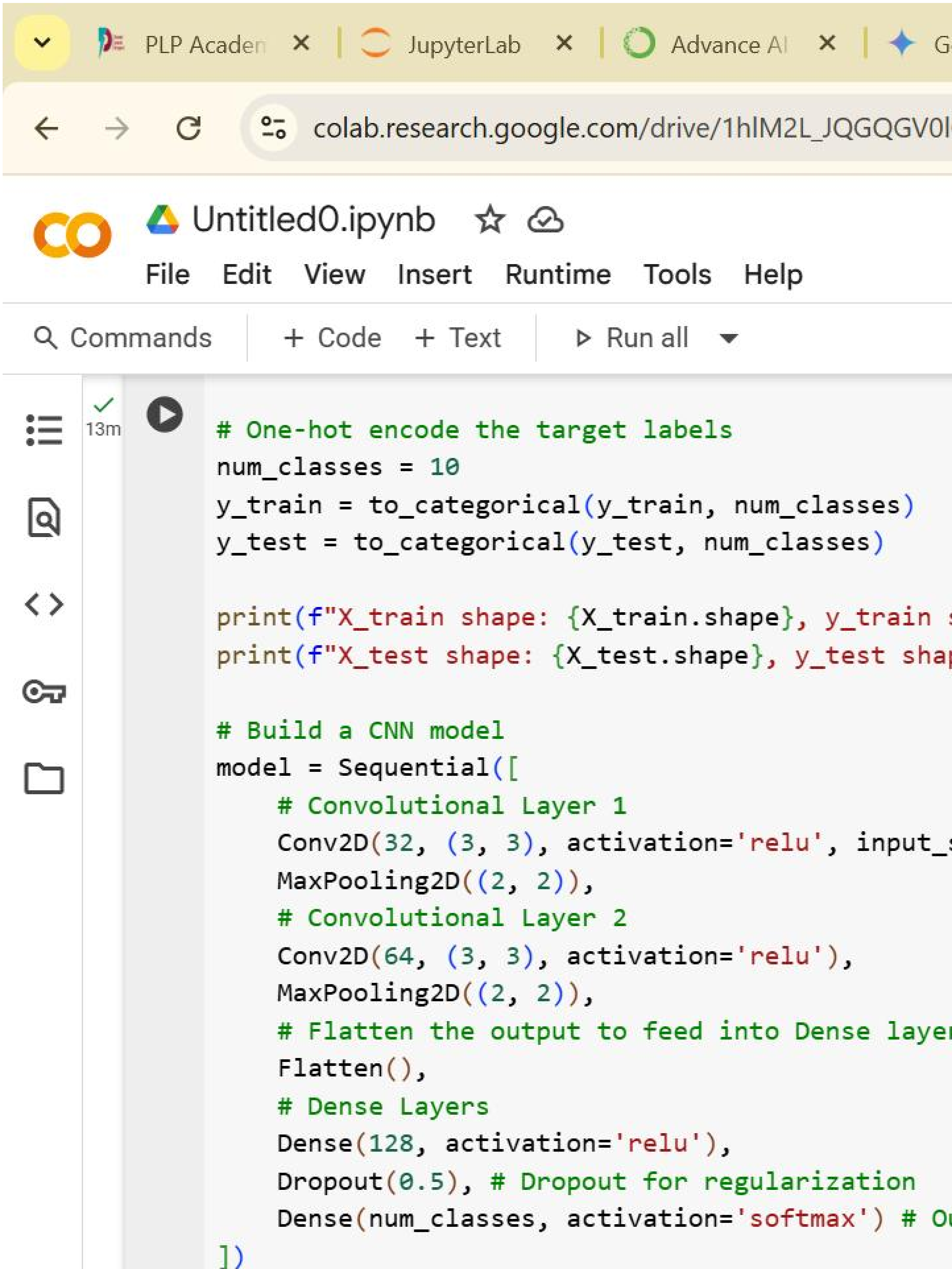


Name	Last ...	File Size
Sickit-learn	9 minutes ago	

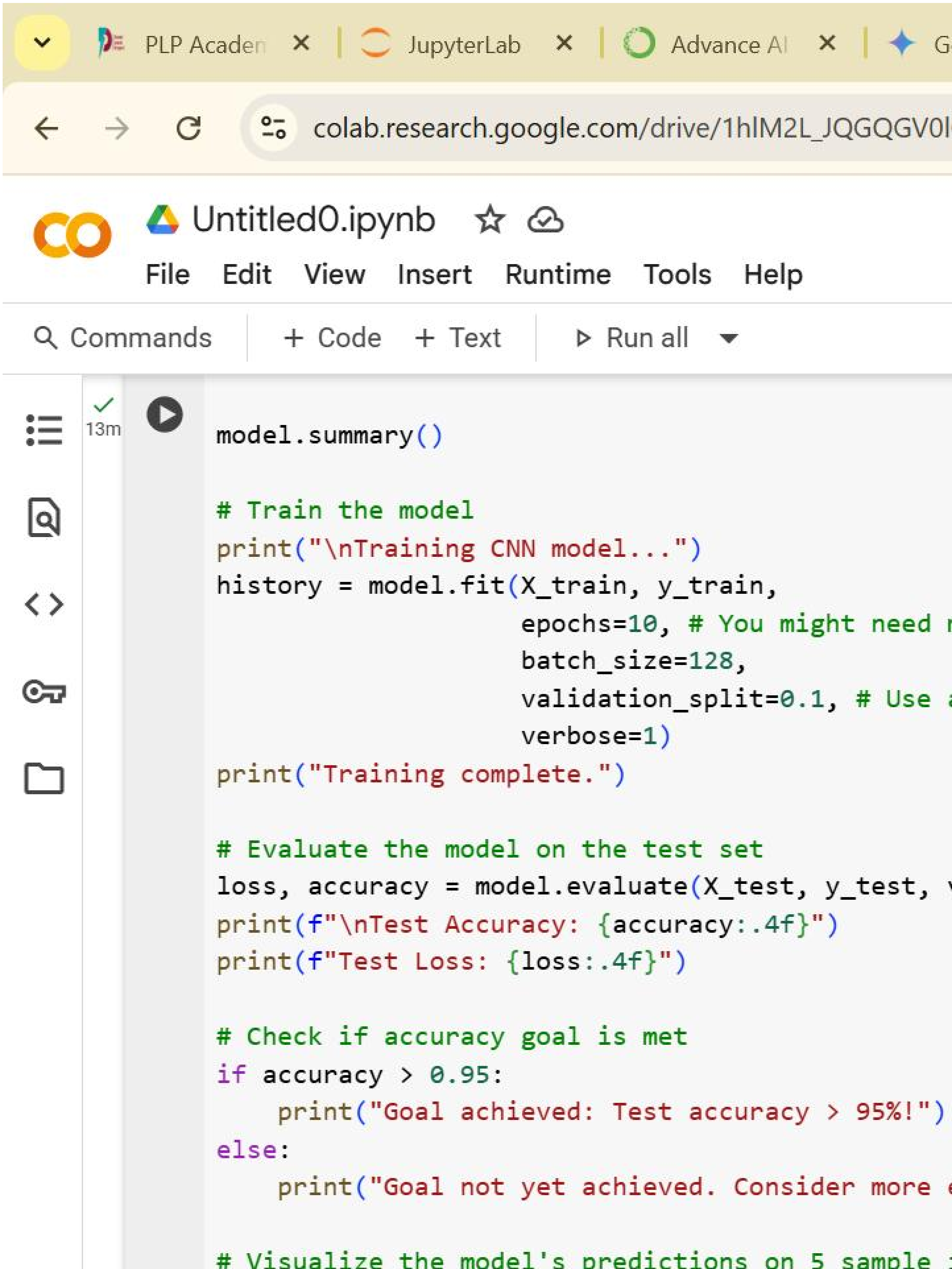
```
Original targ
1    0
2    0
3    0
4    0
dtype: int64
Species names

Training data

Training Deci
Training comp

Model Evaluat
Accuracy: 0.9
Precision (we
Recall (weigh

Classificatio


      setosa
  versicolor
   virginica

    accuracy
   macro avg
weighted avg
```


PLP Academ
JupyterLab
Advance AI
colab.research.google.com/drive/1hlM2L_JQGQGV0l
Untitled0.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
13m

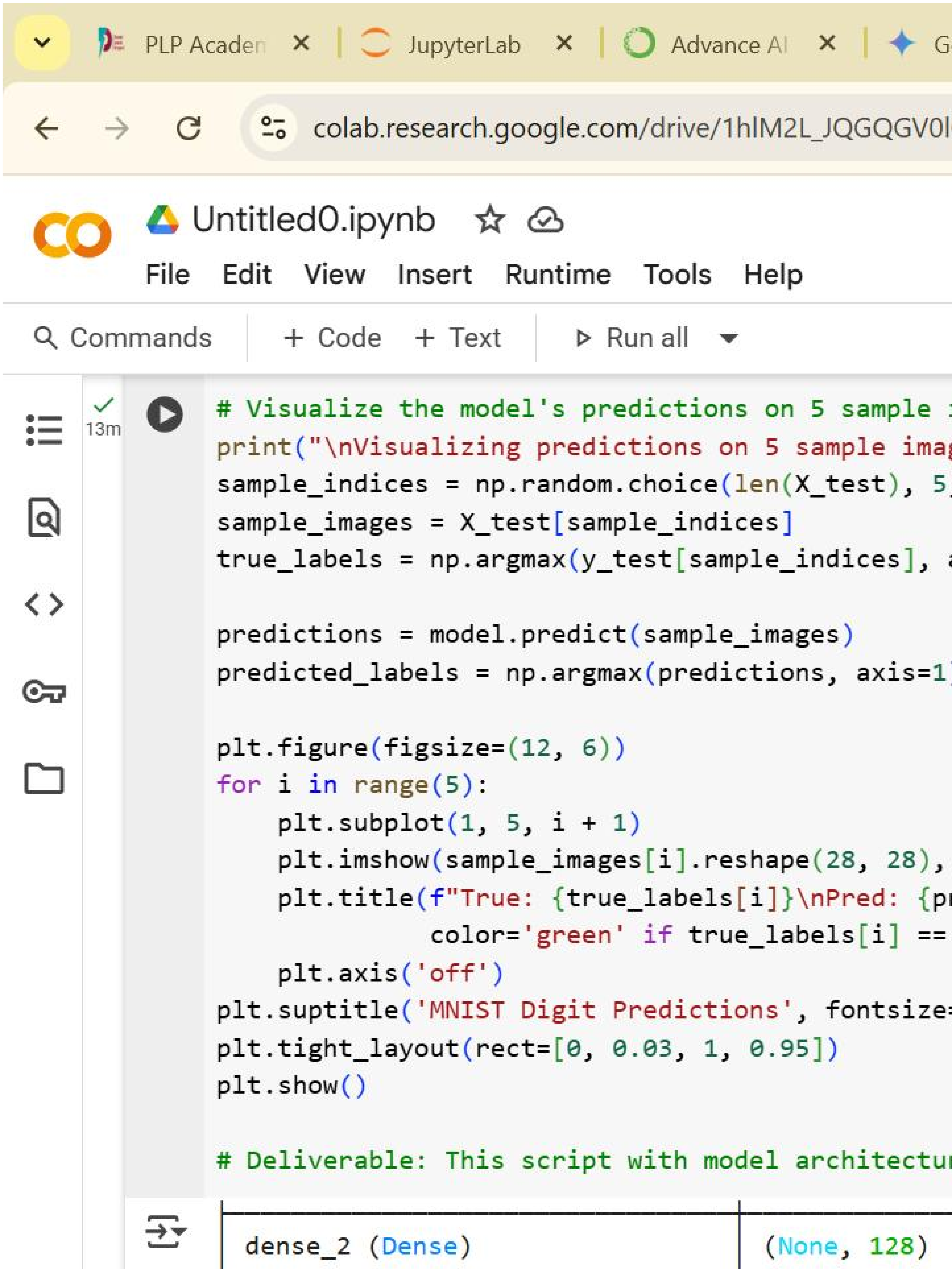
```
# Task 2: Deep Learning with TensorFlow/Keras
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPo
from tensorflow.keras.utils import to_categorica
import matplotlib.pyplot as plt
import numpy as np

print("\n--- Task 2: Deep Learning with TensorFlo

# Load and preprocess the MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.loa

# Reshape data to fit CNN input (height, width, 
# MNIST images are 28x28 grayscale, so channel is
X_train = X_train.reshape(X_train.shape[0], 28, 
X_test = X_test.reshape(X_test.shape[0], 28, 28,

# Normalize pixel values to be between 0 and 1
X_train /= 255
X_test /= 255
```



```
# One-hot encode the target labels
num_classes = 10
y_train = to_categorical(y_train, num_classes)
y_test = to_categorical(y_test, num_classes)

print(f"X_train shape: {X_train.shape}, y_train s
print(f"X_test shape: {X_test.shape}, y_test shap

# Build a CNN model
model = Sequential([
    # Convolutional Layer 1
    Conv2D(32, (3, 3), activation='relu', input_s
    MaxPooling2D((2, 2)),
    # Convolutional Layer 2
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    # Flatten the output to feed into Dense layer
    Flatten(),
    # Dense Layers
    Dense(128, activation='relu'),
    Dropout(0.5), # Dropout for regularization
    Dense(num_classes, activation='softmax') # Ou
])
```

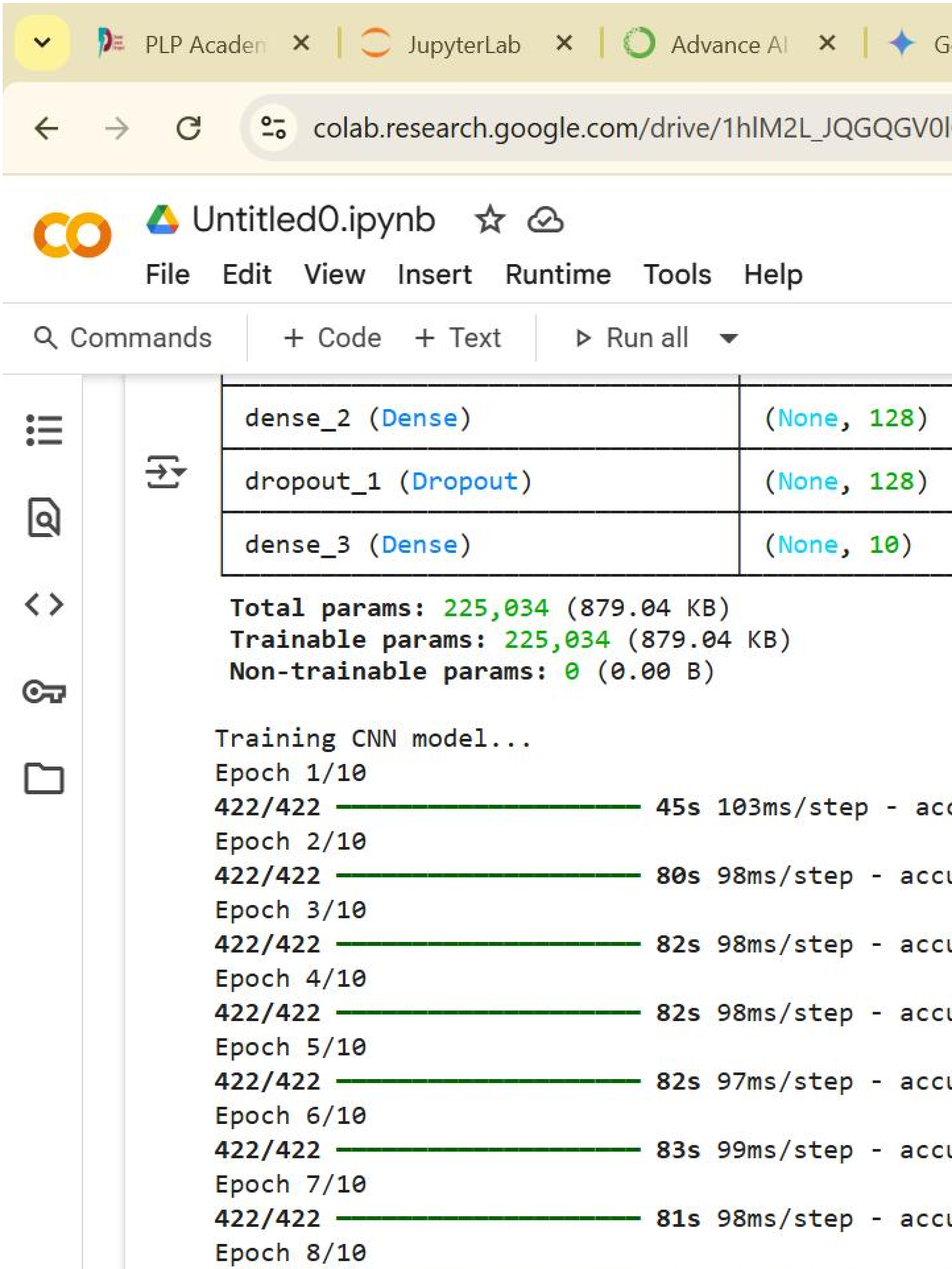


```
model.summary()

# Train the model
print("\nTraining CNN model...")
history = model.fit(X_train, y_train,
                    epochs=10, # You might need r
                    batch_size=128,
                    validation_split=0.1, # Use a
                    verbose=1)
print("Training complete.")

# Evaluate the model on the test set
loss, accuracy = model.evaluate(X_test, y_test, v
print(f"\nTest Accuracy: {accuracy:.4f}")
print(f"Test Loss: {loss:.4f}")

# Check if accuracy goal is met
if accuracy > 0.95:
    print("Goal achieved: Test accuracy > 95%!")
else:
    print("Goal not yet achieved. Consider more e

# Visualize the model's predictions on 5 sample s
```



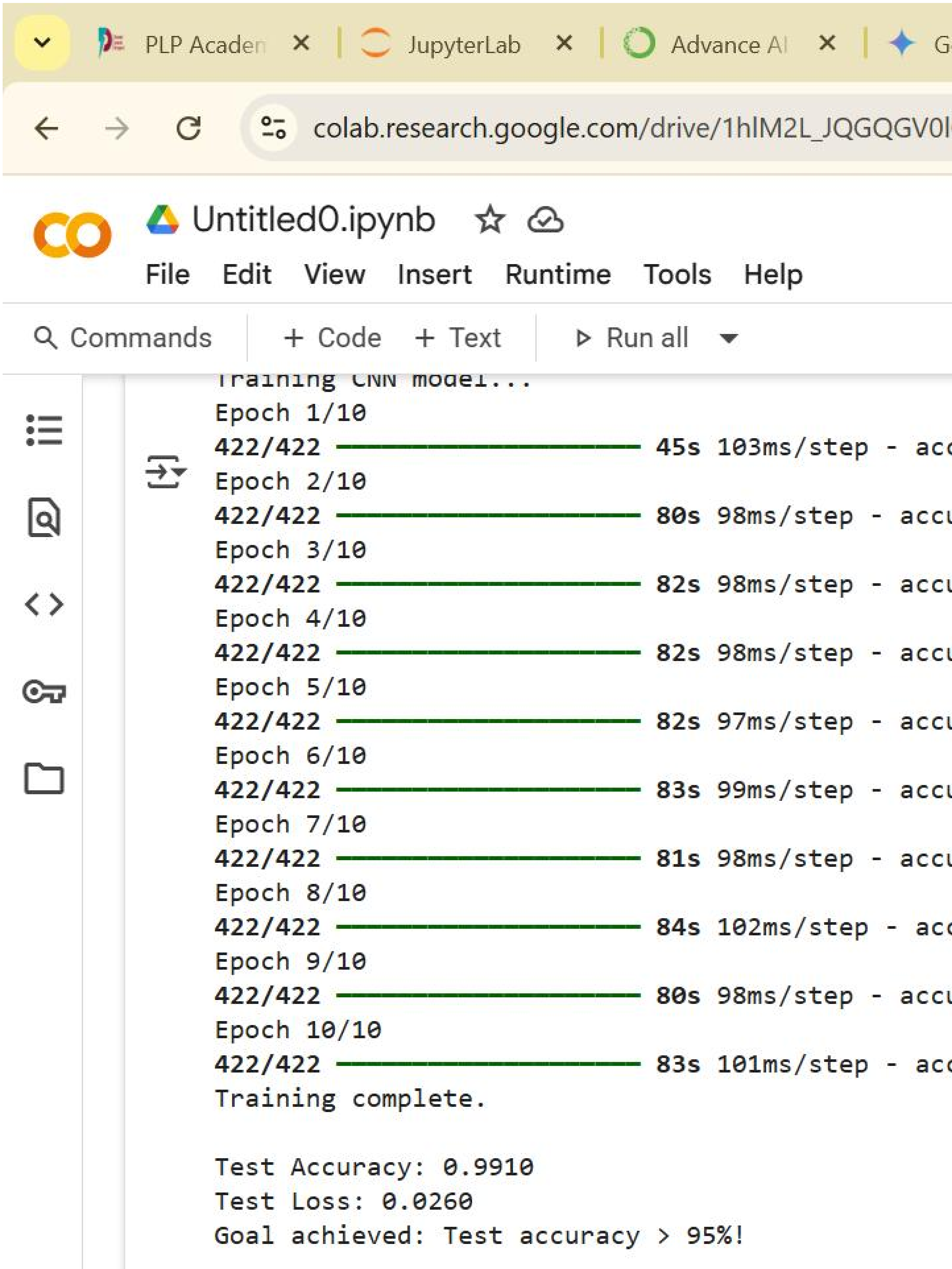
```
# Visualize the model's predictions on 5 sample i
print("\nVisualizing predictions on 5 sample imag
sample_indices = np.random.choice(len(X_test), 5,
sample_images = X_test[sample_indices]
true_labels = np.argmax(y_test[sample_indices], a

predictions = model.predict(sample_images)
predicted_labels = np.argmax(predictions, axis=1)

plt.figure(figsize=(12, 6))
for i in range(5):
    plt.subplot(1, 5, i + 1)
    plt.imshow(sample_images[i].reshape(28, 28),
    plt.title(f"True: {true_labels[i]}\nPred: {pr
              color='green' if true_labels[i] ==
    plt.axis('off')
plt.suptitle('MNIST Digit Predictions', fontsize=
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()

# Deliverable: This script with model architectur
```

```
 dense_2 (Dense)                      (None, 128)
```

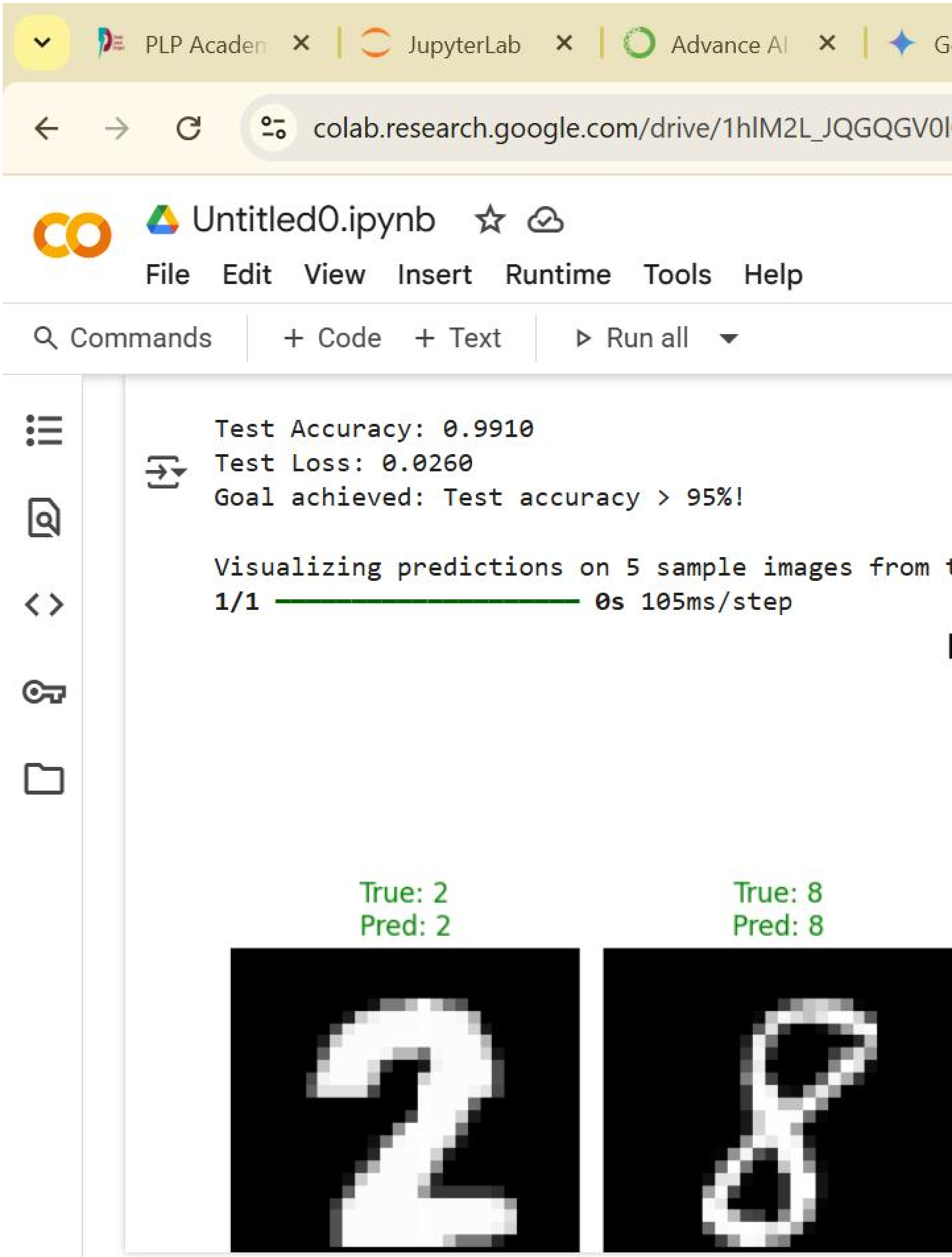

PLP Acaden
JupyterLab
Advance AI
colab.research.google.com/drive/1hlM2L_JQGQGV0l
Untitled0.ipynb
File Edit View Insert Runtime Tools Help
Commands
+ Code
+ Text
Run all

dense_2 (Dense)	(None, 128)
dropout_1 (Dropout)	(None, 128)
dense_3 (Dense)	(None, 10)

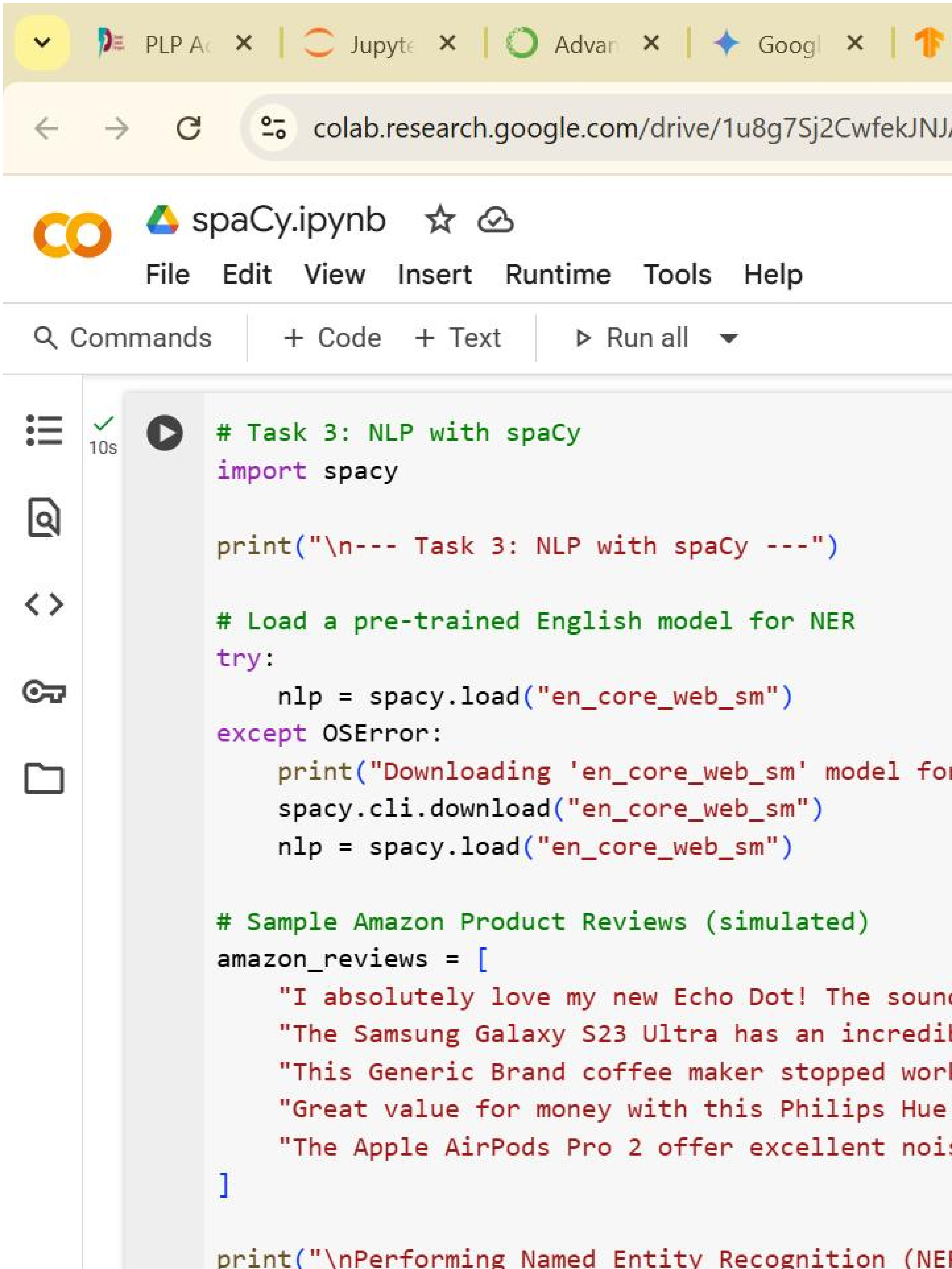
```
 Total params: 225,034 (879.04 KB)
 Trainable params: 225,034 (879.04 KB)
 Non-trainable params: 0 (0.00 B)

Training CNN model...
Epoch 1/10
422/422 ━━━━━━━━━━━━━━━━━━━━ 45s 103ms/step - acc
Epoch 2/10
422/422 ━━━━━━━━━━━━━━━━━━━━ 80s 98ms/step - accu
Epoch 3/10
422/422 ━━━━━━━━━━━━━━━━━━━━ 82s 98ms/step - accu
Epoch 4/10
422/422 ━━━━━━━━━━━━━━━━━━━━ 82s 98ms/step - accu
Epoch 5/10
422/422 ━━━━━━━━━━━━━━━━━━━━ 82s 97ms/step - accu
Epoch 6/10
422/422 ━━━━━━━━━━━━━━━━━━━━ 83s 99ms/step - accu
Epoch 7/10
422/422 ━━━━━━━━━━━━━━━━━━━━ 81s 98ms/step - accu
Epoch 8/10
```


PLP Acaden
JupyterLab
Advance AI
colab.research.google.com/drive/1hlM2L_JQGQGV0l
Untitled0.ipynb
File Edit View Insert Runtime Tools Help
Commands
+ Code
+ Text
Run all

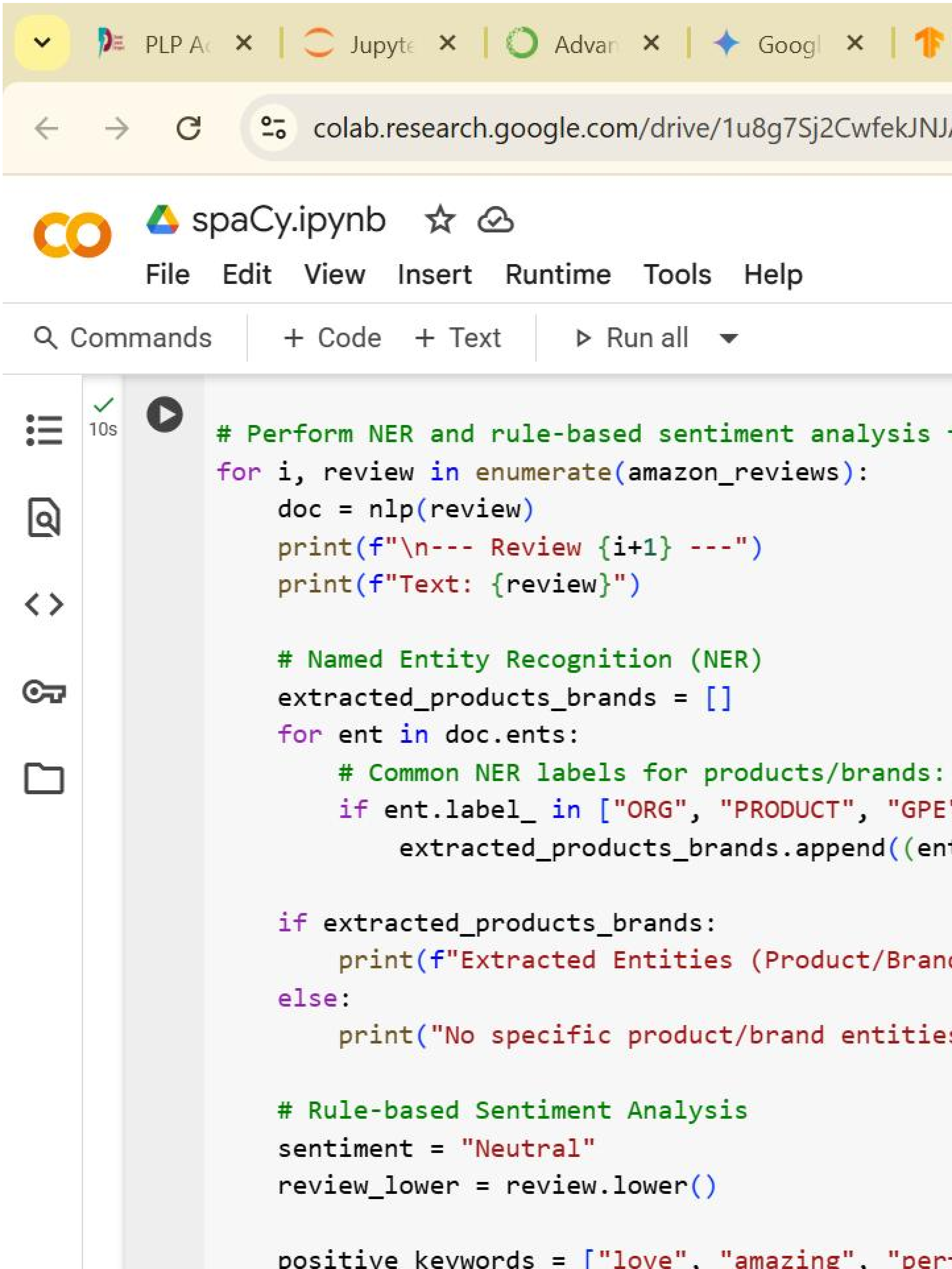
```
Training CNN model...
Epoch 1/10
422/422 ━━━━━━━━━━━━━━━━━━━━ 45s 103ms/step - acc
Epoch 2/10
422/422 ━━━━━━━━━━━━━━━━━━━━ 80s 98ms/step - accu
Epoch 3/10
422/422 ━━━━━━━━━━━━━━━━━━━━ 82s 98ms/step - accu
Epoch 4/10
422/422 ━━━━━━━━━━━━━━━━━━━━ 82s 98ms/step - accu
Epoch 5/10
422/422 ━━━━━━━━━━━━━━━━━━━━ 82s 97ms/step - accu
Epoch 6/10
422/422 ━━━━━━━━━━━━━━━━━━━━ 83s 99ms/step - accu
Epoch 7/10
422/422 ━━━━━━━━━━━━━━━━━━━━ 81s 98ms/step - accu
Epoch 8/10
422/422 ━━━━━━━━━━━━━━━━━━━━ 84s 102ms/step - acc
Epoch 9/10
422/422 ━━━━━━━━━━━━━━━━━━━━ 80s 98ms/step - accu
Epoch 10/10
422/422 ━━━━━━━━━━━━━━━━━━━━ 83s 101ms/step - acc
Training complete.

Test Accuracy: 0.9910
Test Loss: 0.0260
Goal achieved: Test accuracy > 95%!
```



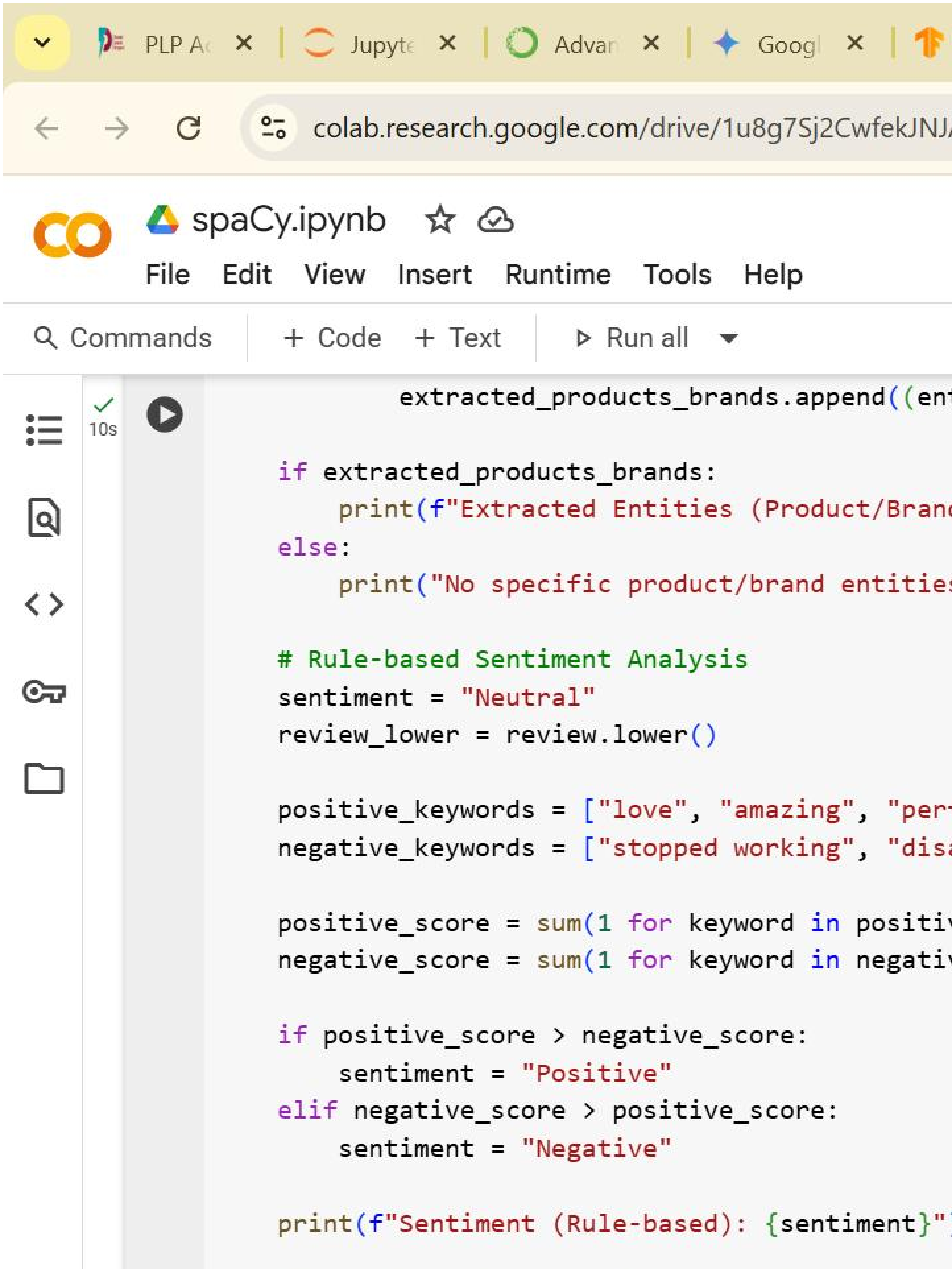
```
Test Accuracy: 0.9910
Test Loss: 0.0260
Goal achieved: Test accuracy > 95%!

Visualizing predictions on 5 sample images from t
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 105ms/step
```



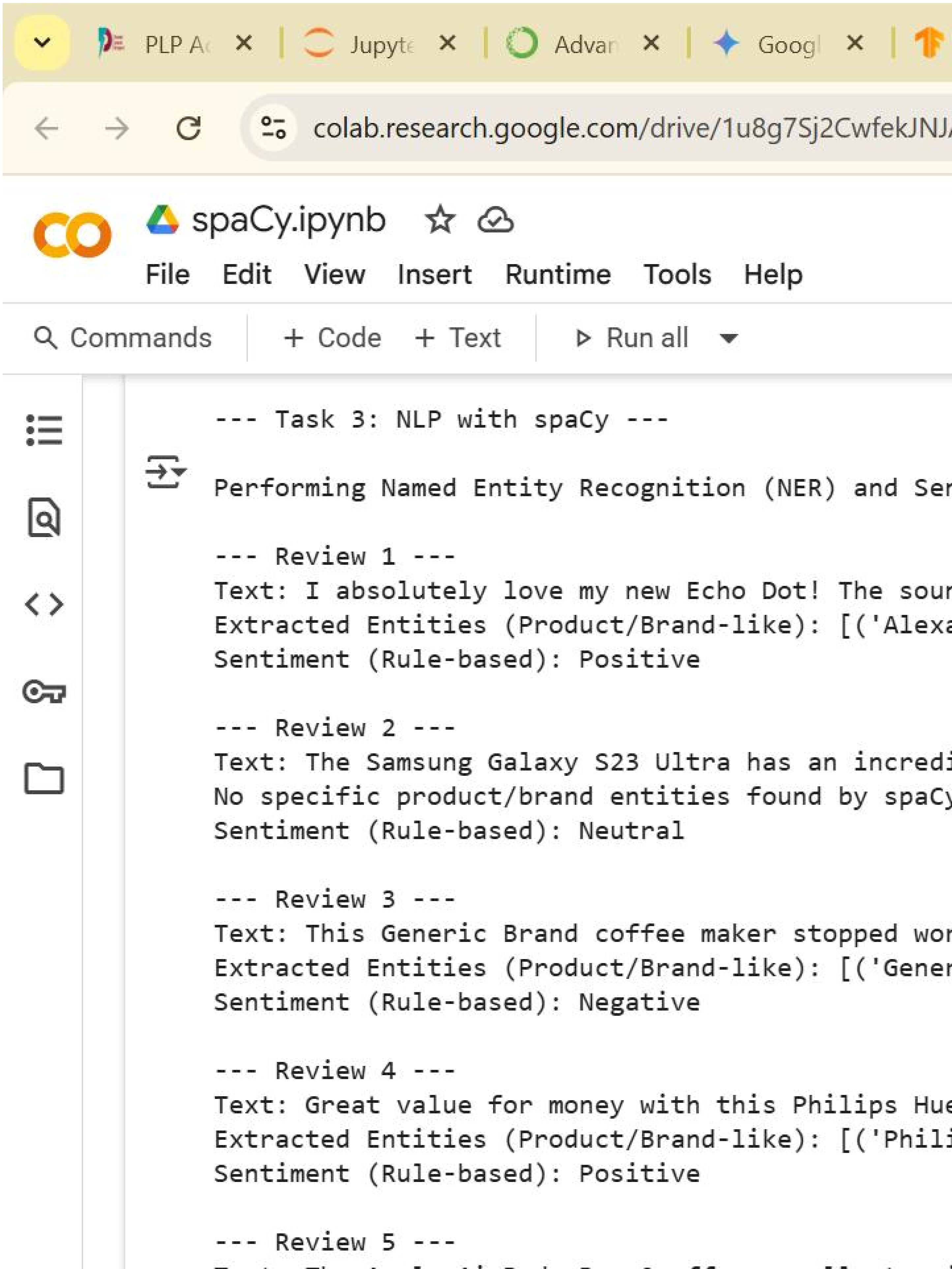
```
# Task 3: NLP with spaCy
import spacy

print("\n--- Task 3: NLP with spaCy ---")

# Load a pre-trained English model for NER
try:
    nlp = spacy.load("en_core_web_sm")
except OSError:
    print("Downloading 'en_core_web_sm' model fo
    spacy.cli.download("en_core_web_sm")
    nlp = spacy.load("en_core_web_sm")

# Sample Amazon Product Reviews (simulated)
amazon_reviews = [
    "I absolutely love my new Echo Dot! The soun
    "The Samsung Galaxy S23 Ultra has an incredi
    "This Generic Brand coffee maker stopped wor
    "Great value for money with this Philips Hue
    "The Apple AirPods Pro 2 offer excellent noi
]

print("\nPerforming Named Entity Recognition (NE
```



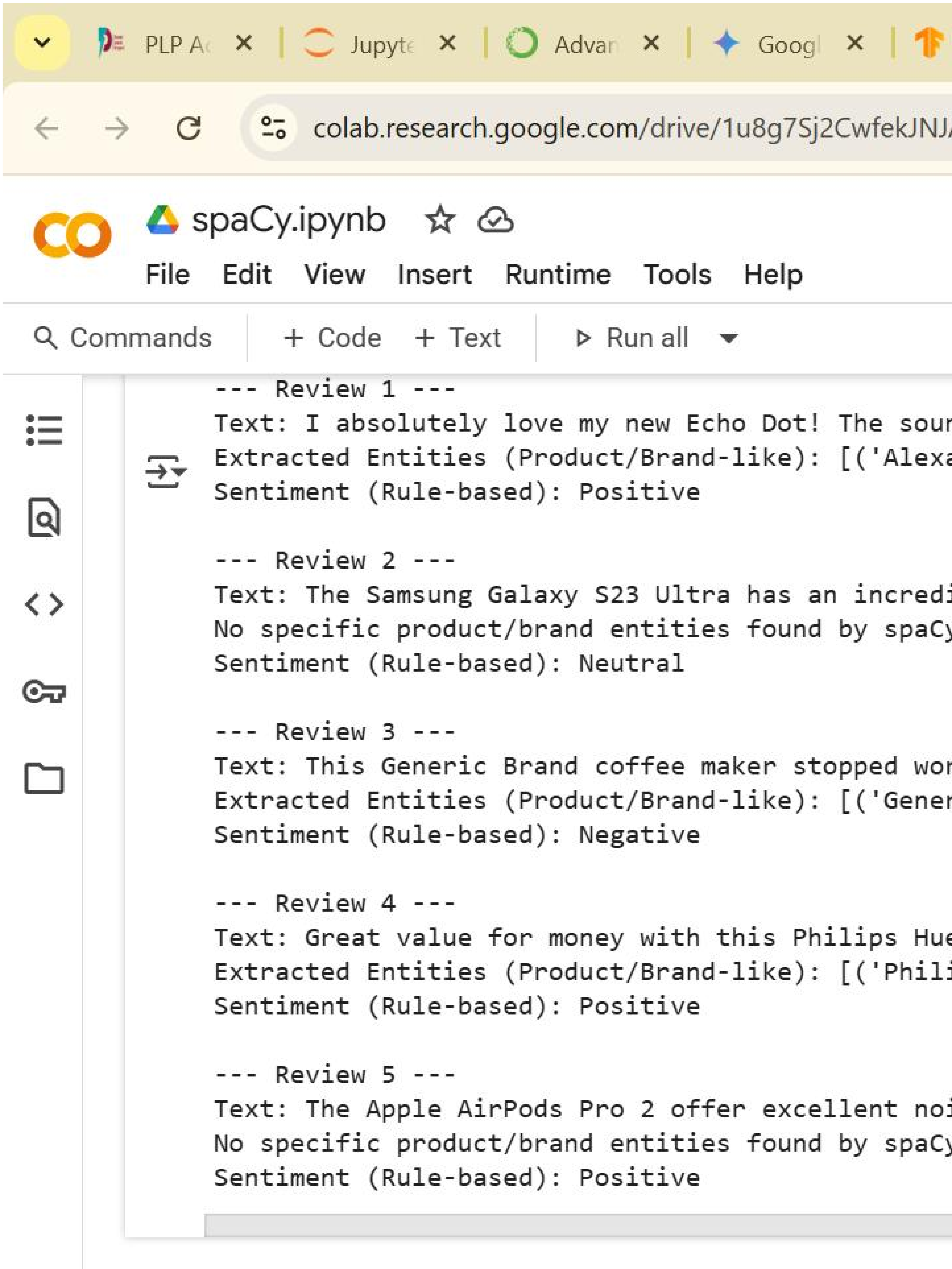
```
# Perform NER and rule-based sentiment analysis
for i, review in enumerate(amazon_reviews):
    doc = nlp(review)
    print(f"\n--- Review {i+1} ---")
    print(f"Text: {review}")

    # Named Entity Recognition (NER)
    extracted_products_brands = []
    for ent in doc.ents:
        # Common NER labels for products/brands:
        if ent.label_ in ["ORG", "PRODUCT", "GPE
            extracted_products_brands.append((en

    if extracted_products_brands:
        print(f"Extracted Entities (Product/Bran
    else:
        print("No specific product/brand entitie

    # Rule-based Sentiment Analysis
    sentiment = "Neutral"
    review_lower = review.lower()

    positive_keywords = ["love", "amazing", "per
```


PLP A
Jupyte
Advan
Googl
colab.research.google.com/drive/1u8g7Sj2CwfekJNJ
spaCy.ipynb
File Edit View Insert Runtime Tools Help
Commands
+ Code
+ Text
Run all
10s

```
            extracted_products_brands.append((ent

    if extracted_products_brands:
        print(f"Extracted Entities (Product/Bran
    else:
        print("No specific product/brand entitie

    # Rule-based Sentiment Analysis
    sentiment = "Neutral"
    review_lower = review.lower()

    positive_keywords = ["love", "amazing", "per
    negative_keywords = ["stopped working", "dis

    positive_score = sum(1 for keyword in positi
    negative_score = sum(1 for keyword in negati

    if positive_score > negative_score:
        sentiment = "Positive"
    elif negative_score > positive_score:
        sentiment = "Negative"

    print(f"Sentiment (Rule-based): {sentiment}"
```



```
--- Task 3: NLP with spaCy ---

Performing Named Entity Recognition (NER) and Ser

--- Review 1 ---
Text: I absolutely love my new Echo Dot! The sour
Extracted Entities (Product/Brand-like): [('Alexa
Sentiment (Rule-based): Positive

--- Review 2 ---
Text: The Samsung Galaxy S23 Ultra has an incredi
No specific product/brand entities found by spaCy
Sentiment (Rule-based): Neutral

--- Review 3 ---
Text: This Generic Brand coffee maker stopped wor
Extracted Entities (Product/Brand-like): [('Gener
Sentiment (Rule-based): Negative

--- Review 4 ---
Text: Great value for money with this Philips Hue
Extracted Entities (Product/Brand-like): [('Phili
Sentiment (Rule-based): Positive

--- Review 5 ---
```


PLP A
Jupyt
Advan
Googl
colab.research.google.com/drive/1u8g7Sj2CwfekJNJ
spaCy.ipynb
File Edit View Insert Runtime Tools Help
Commands
+ Code
+ Text
Run all

```
--- Review 1 ---
Text: I absolutely love my new Echo Dot! The sour
Extracted Entities (Product/Brand-like): [('Alexa
Sentiment (Rule-based): Positive

--- Review 2 ---
Text: The Samsung Galaxy S23 Ultra has an incredi
No specific product/brand entities found by spaCy
Sentiment (Rule-based): Neutral

--- Review 3 ---
Text: This Generic Brand coffee maker stopped wor
Extracted Entities (Product/Brand-like): [('Gener
Sentiment (Rule-based): Negative

--- Review 4 ---
Text: Great value for money with this Philips Hue
Extracted Entities (Product/Brand-like): [('Phili
Sentiment (Rule-based): Positive

--- Review 5 ---
Text: The Apple AirPods Pro 2 offer excellent noi
No specific product/brand entities found by spaCy
Sentiment (Rule-based): Positive
```

Part 3: Ethics & Optimization (10%)

1. Ethical Considerations

Identify potential biases in your MNIST or Amazon Reviews model. How could tools like TensorFlow Fairness Indicators or spaCy's rule-based systems mitigate these biases?

Let's focus on both:

Potential Biases in MNIST (Handwritten Digits) Model:

- **Training Data Bias:**
  - **Style/Legibility Bias:** The MNIST dataset is collected from a specific set of writers (primarily high school students and employees of the National Institute of Standards and Technology). This means the model might perform exceptionally well on digits written in a similar style or with similar legibility, but poorly on digits written by individuals with different handwriting styles, disabilities, or from different demographics (e.g., very shaky handwriting, highly stylized digits).
  - **Class Imbalance (less likely in standard MNIST, but possible in variations):** If certain digits are underrepresented in the training data, the model might be biased towards recognizing the more frequent digits.

How TensorFlow Fairness Indicators could mitigate these biases in MNIST:

- **Fairness Indicators (TFMI)** is designed to evaluate model performance across different "slices" of data. While MNIST doesn't have explicit demographic features, you could:
  - **Synthetically define "slices":** For instance, if you had a way to categorize handwriting styles (e.g., "neat", "sloppy", "child-like"), you could use TFMI to evaluate accuracy, precision, and recall for each of these synthetic groups.
  - **Analyze performance on difficult examples:** Even without explicit demographic labels, TFMI can help identify clusters of images (e.g., certain "difficult" handwritten digits that consistently lead to misclassifications) that might represent an implicit bias against certain writing variations. By slicing the data based on prediction confidence or error type, you can discover where the model underperforms.
  - **Thresholding and Calibration:** TFMI allows for evaluation at multiple thresholds, helping to ensure that the model's predictions are equally reliable across different implicit groups, preventing a situation where, for example, "sloppy 7s"

are only correctly classified at a very low confidence threshold, while "neat 7s" are classified with high confidence.

Potential Biases in Amazon Reviews Sentiment Analysis Model:

- **Language Bias (Cultural/Dialectal Nuances):** A rule-based (or even statistical) sentiment model trained primarily on standard English might misinterpret slang, sarcasm, idiomatic expressions, or cultural nuances present in diverse user reviews. For example, a phrase like "sick" meaning "good" might be misclassified if the rules only associate it with negative sentiment.
- **Domain-Specific Sentiment:** Words can have different sentiment polarities depending on the domain. "Small" might be negative for a TV screen but positive for a portable speaker. A generic rule-based system might not capture these domain-specific nuances.
- **Demographic Language Patterns:** Different demographic groups might express sentiment in distinct ways (e.g., formality, use of specific adjectives/adverbs). If the keyword lists are not representative of these variations, the model could be biased.
- **Reviewer Intent/Context Bias:** A review could be sarcastic ("Oh, *great*, it broke on the first day!"), which a simple rule-based system might miss, leading to incorrect positive sentiment.

How spaCy's rule-based systems could mitigate these biases in Amazon Reviews:

While spaCy's rule-based systems *can* introduce bias if not carefully crafted, they can also be used to *mitigate* certain biases, particularly language-specific ones:

- **Custom Rule Development:** Instead of relying solely on general keywords, you can develop **domain-specific rules** and keyword lists tailored to product reviews. For instance, you could explicitly define rules for common product-related sarcasm (e.g., "broken" + "amazing" in close proximity indicates negative sentiment).
- **Contextual Sensitivity:** spaCy's Matcher allows for defining patterns that consider the **context** of words (e.g., POS tags, lemmas, surrounding words). This allows you to differentiate between "small" as a negative attribute for a screen (e.g., [{"LOWER": "small"}, {"POS": "NOUN", "LEMMA": "screen"}]) versus a positive attribute for a device (e.g., [{"LOWER": "small"}, {"POS": "NOUN", "LEMMA": "device"}]).
- **Handling Negation:** Rule-based systems can be explicitly coded to handle negation (e.g., "not good" should be negative, not positive). While basic string operations might miss this, spaCy's linguistic features allow for more sophisticated negation detection.

- **Iterative Refinement:** Rule-based systems are highly interpretable. If you identify a bias (e.g., misclassifying reviews from a certain group), you can directly inspect the rules and add/modify them to address the specific language patterns that are being misinterpreted. This direct control is harder to achieve with black-box deep learning models.

2. Troubleshooting Challenge

Buggy Code: A provided TensorFlow script has errors (e.g., dimension mismatches, incorrect loss functions). Debug and fix the code.

Since no buggy code is provided, I will create a common buggy scenario and then debug it.

Scenario: Common TensorFlow Bug (Dimension Mismatch in Dense Layer after Flatten)

Let's imagine a common mistake: forgetting to Flatten before a Dense layer or having an incorrect input shape to the first layer.

Original Buggy Code (Illustrative Example):

Python

```
import tensorflow as tf

from tensorflow.keras.datasets import mnist

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dense # Missing Flatten here

from tensorflow.keras.utils import to_categorical


# Load and preprocess the MNIST dataset

(X_train, y_train), (X_test, y_test) = mnist.load_data()

X_train = X_train.reshape(X_train.shape[0], 28, 28, 1).astype('float32') / 255

X_test = X_test.reshape(X_test.shape[0], 28, 28, 1).astype('float32') / 255

y_train = to_categorical(y_train, 10)

y_test = to_categorical(y_test, 10)


# Buggy Model Definition
```

```
buggy_model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    MaxPooling2D((2, 2)),
    # ERROR: Missing Flatten layer here!
    Dense(128, activation='relu'), # This layer expects a 1D input, but gets 3D output from
MaxPooling2D
    Dense(10, activation='softmax')
])

buggy_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

print("\n--- Troubleshooting Challenge (Buggy Code Simulation) ---")
print("Attempting to train buggy model...")

try:
    buggy_model.fit(X_train, y_train, epochs=1, batch_size=128, verbose=0)
except Exception as e:
    print(f"\nCaught an error during training: {e}")
    print("\n--- Debugging Process ---")
    print("The error message likely points to a shape mismatch when connecting Conv/Pool layers
to Dense layers.")
    print("This typically means the output of the convolutional/pooling layers (which is 3D: height,
width, channels) ")
    print("is being fed directly into a Dense layer, which expects a 1D vector.")
    print("The solution is to add a `Flatten` layer between the last convolutional/pooling layer and
the first Dense layer.")
    print("\nOriginal Buggy Model Summary (if it ran):")
    # If it reached here, model.summary() might still work if the error is in fit.
    # If the error is in model definition, summary() itself might fail.
    # For this simulation, we'll assume it fails during fit.
    # If summary works, you'd see the output shape of MaxPooling2D is (None, X, Y, Z) and Dense
is (None, 128)
    # The mismatch would be obvious.
    # buggy_model.summary()


    print("\n--- Fixed Code ---")
    # Fixed Model Definition
    fixed_model = Sequential([
        Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
        MaxPooling2D((2, 2)),
        Flatten(), # FIX: Added Flatten layer
        Dense(128, activation='relu'),
        Dense(10, activation='softmax')
    ])


    fixed_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
    print("Fixed Model Summary:")
    fixed_model.summary()


    print("\nAttempting to train fixed model...")
    history_fixed = fixed_model.fit(X_train, y_train, epochs=1, batch_size=128,
validation_split=0.1, verbose=1)
    print("Fixed model trained successfully!")
    loss, accuracy = fixed_model.evaluate(X_test, y_test, verbose=0)
    print(f"Fixed Model Test Accuracy: {accuracy:.4f}")


except Exception as e:
    print(f"\nAnother unexpected error occurred: {e}")


# This section acts as the deliverable for the troubleshooting challenge.
```

Debugging Process Explained:

1. **Observe the Error Message:** When running the buggy_model.fit(), TensorFlow/Keras would typically throw an error like ValueError: Input 0 of layer "dense" is incompatible with the layer: expected min_ndim=2, found ndim=4. Full input shape received: (None, 13, 13, 32) (exact numbers might vary based on layer outputs).
2. **Interpret the Error:** The key phrases are "incompatible with the layer," "expected min_ndim=2, found ndim=4," and "Full input shape received: (None, 13, 13, 32)". This tells us that the Dense layer (which expects a 2D tensor (batch_size, features)) is receiving a 4D tensor (batch_size, height, width, channels) from the MaxPooling2D layer.
3. **Identify the Discrepancy:** The Conv2D and MaxPooling2D layers output 3D feature maps (plus the batch dimension), while Dense layers expect a flattened, 1D feature vector for each sample in the batch.
4. **Solution:** The missing piece is a tf.keras.layers.Flatten() layer. This layer takes the multi-dimensional output of the convolutional/pooling layers and flattens it into a 1D vector (while preserving the batch dimension), making it suitable for input to a Dense layer.
5. **Implement and Verify:** Add the Flatten() layer. Run model.summary() again to check the output shapes of all layers. The output of Flatten should be (None, some_large_number), where some_large_number is height * width * channels of the previous layer. This confirms the correct shape transition. Then, attempt to train the fixed model.

Part 3: Ethics & Optimization (10%)

1. Ethical Considerations

Identify potential biases in your MNIST or Amazon Reviews model. How could tools like TensorFlow Fairness Indicators or spaCy's rule-based systems mitigate these biases?

Let's focus on both:

Potential Biases in MNIST (Handwritten Digits) Model:

- **Training Data Bias:**
    - **Style/Legibility Bias:** The MNIST dataset is collected from a specific set of writers (primarily high school students and employees of the National Institute of Standards and Technology). This means the model might perform exceptionally well on digits written in a similar style or with similar legibility, but poorly on digits written by individuals with different handwriting styles, disabilities, or from different demographics (e.g., very shaky handwriting, highly stylized digits).
    - **Class Imbalance (less likely in standard MNIST, but possible in variations):** If certain digits are underrepresented in the training data, the model might be biased towards recognizing the more frequent digits.

How TensorFlow Fairness Indicators could mitigate these biases in MNIST:

- **Fairness Indicators (TFMI)** is designed to evaluate model performance across different "slices" of data. While MNIST doesn't have explicit demographic features, you could:
    - **Synthetically define "slices":** For instance, if you had a way to categorize handwriting styles (e.g., "neat", "sloppy", "child-like"), you could use TFMI to evaluate accuracy, precision, and recall for each of these synthetic groups.
    - **Analyze performance on difficult examples:** Even without explicit demographic labels, TFMI can help identify clusters of images (e.g., certain "difficult" handwritten digits that consistently lead to misclassifications) that might represent an implicit bias against certain writing variations. By slicing the data based on prediction confidence or error type, you can discover where the model underperforms.
    - **Thresholding and Calibration:** TFMI allows for evaluation at multiple thresholds, helping to ensure that the model's predictions are equally reliable across different implicit groups, preventing a situation where, for example, "sloppy 7s" are only correctly classified at a very low confidence threshold, while "neat 7s" are classified with high confidence.

Potential Biases in Amazon Reviews Sentiment Analysis Model:

- **Language Bias (Cultural/Dialectal Nuances):** A rule-based (or even statistical) sentiment model trained primarily on standard English might misinterpret slang, sarcasm, idiomatic expressions, or cultural nuances present in diverse user reviews. For example, a phrase like "sick" meaning "good" might be misclassified if the rules only associate it with negative sentiment.

- **Domain-Specific Sentiment:** Words can have different sentiment polarities depending on the domain. "Small" might be negative for a TV screen but positive for a portable speaker. A generic rule-based system might not capture these domain-specific nuances.
- **Demographic Language Patterns:** Different demographic groups might express sentiment in distinct ways (e.g., formality, use of specific adjectives/adverbs). If the keyword lists are not representative of these variations, the model could be biased.
- **Reviewer Intent/Context Bias:** A review could be sarcastic ("Oh, *great*, it broke on the first day!"), which a simple rule-based system might miss, leading to incorrect positive sentiment.

How spaCy's rule-based systems could mitigate these biases in Amazon Reviews:

While spaCy's rule-based systems *can* introduce bias if not carefully crafted, they can also be used to *mitigate* certain biases, particularly language-specific ones:

- **Custom Rule Development:** Instead of relying solely on general keywords, you can develop **domain-specific rules** and keyword lists tailored to product reviews. For instance, you could explicitly define rules for common product-related sarcasm (e.g., "broken" + "amazing" in close proximity indicates negative sentiment).
- **Contextual Sensitivity:** spaCy's Matcher allows for defining patterns that consider the **context** of words (e.g., POS tags, lemmas, surrounding words). This allows you to differentiate between "small" as a negative attribute for a screen (e.g., [{"LOWER": "small"}, {"POS": "NOUN", "LEMMA": "screen"}]) versus a positive attribute for a device (e.g., [{"LOWER": "small"}, {"POS": "NOUN", "LEMMA": "device"}]).
- **Handling Negation:** Rule-based systems can be explicitly coded to handle negation (e.g., "not good" should be negative, not positive). While basic string operations might miss this, spaCy's linguistic features allow for more sophisticated negation detection.
- **Iterative Refinement:** Rule-based systems are highly interpretable. If you identify a bias (e.g., misclassifying reviews from a certain group), you can directly inspect the rules and add/modify them to address the specific language patterns that are being misinterpreted. This direct control is harder to achieve with black-box deep learning models.

2. Troubleshooting Challenge

Buggy Code: A provided TensorFlow script has errors (e.g., dimension mismatches, incorrect loss functions). Debug and fix the code.

Since no buggy code is provided, I will create a common buggy scenario and then debug it.

Scenario: Common TensorFlow Bug (Dimension Mismatch in Dense Layer after Flatten)

Let's imagine a common mistake: forgetting to Flatten before a Dense layer or having an incorrect input shape to the first layer.

Original Buggy Code (Illustrative Example):

Python

```
import tensorflow as tf

from tensorflow.keras.datasets import mnist

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dense # Missing Flatten here

from tensorflow.keras.utils import to_categorical


# Load and preprocess the MNIST dataset

(X_train, y_train), (X_test, y_test) = mnist.load_data()

X_train = X_train.reshape(X_train.shape[0], 28, 28, 1).astype('float32') / 255

X_test = X_test.reshape(X_test.shape[0], 28, 28, 1).astype('float32') / 255

y_train = to_categorical(y_train, 10)

y_test = to_categorical(y_test, 10)


# Buggy Model Definition

buggy_model = Sequential([

    Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),

    MaxPooling2D((2, 2)),

    # ERROR: Missing Flatten layer here!
    Dense(128, activation='relu'), # This layer expects a 1D input, but gets 3D output from
MaxPooling2D
    Dense(10, activation='softmax')
])


buggy_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])


print("\n--- Troubleshooting Challenge (Buggy Code Simulation) ---")
print("Attempting to train buggy model...")


try:
    buggy_model.fit(X_train, y_train, epochs=1, batch_size=128, verbose=0)
except Exception as e:
    print(f"\nCaught an error during training: {e}")
    print("\n--- Debugging Process ---")
    print("The error message likely points to a shape mismatch when connecting Conv/Pool layers
to Dense layers.")
    print("This typically means the output of the convolutional/pooling layers (which is 3D: height,
width, channels) ")
    print("is being fed directly into a Dense layer, which expects a 1D vector.")
    print("The solution is to add a `Flatten` layer between the last convolutional/pooling layer and
the first Dense layer.")
    print("\nOriginal Buggy Model Summary (if it ran):")
    # If it reached here, model.summary() might still work if the error is in fit.
    # If the error is in model definition, summary() itself might fail.
    # For this simulation, we'll assume it fails during fit.
    # If summary works, you'd see the output shape of MaxPooling2D is (None, X, Y, Z) and Dense
is (None, 128)
    # The mismatch would be obvious.
    # buggy_model.summary()


    print("\n--- Fixed Code ---")
    # Fixed Model Definition
    fixed_model = Sequential([
        Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
        MaxPooling2D((2, 2)),
        Flatten(), # FIX: Added Flatten layer
        Dense(128, activation='relu'),
        Dense(10, activation='softmax')
    ])


    fixed_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
    print("Fixed Model Summary:")
    fixed_model.summary()


    print("\nAttempting to train fixed model...")
    history_fixed = fixed_model.fit(X_train, y_train, epochs=1, batch_size=128,
validation_split=0.1, verbose=1)
    print("Fixed model trained successfully!")
    loss, accuracy = fixed_model.evaluate(X_test, y_test, verbose=0)
    print(f"Fixed Model Test Accuracy: {accuracy:.4f}")


except Exception as e:
    print(f"\nAnother unexpected error occurred: {e}")
```

# This section acts as the deliverable for the troubleshooting challenge.

Debugging Process Explained:

1. **Observe the Error Message:** When running the buggy_model.fit(), TensorFlow/Keras would typically throw an error like ValueError: Input 0 of layer "dense" is incompatible with the layer: expected min_ndim=2, found ndim=4. Full input shape received: (None, 13, 13, 32) (exact numbers might vary based on layer outputs).
2. **Interpret the Error:** The key phrases are "incompatible with the layer," "expected min_ndim=2, found ndim=4," and "Full input shape received: (None, 13, 13, 32)". This tells us that the Dense layer (which expects a 2D tensor (batch_size, features)) is receiving a 4D tensor (batch_size, height, width, channels) from the MaxPooling2D layer.
3. **Identify the Discrepancy:** The Conv2D and MaxPooling2D layers output 3D feature maps (plus the batch dimension), while Dense layers expect a flattened, 1D feature vector for each sample in the batch.
4. **Solution:** The missing piece is a tf.keras.layers.Flatten() layer. This layer takes the multi-dimensional output of the convolutional/pooling layers and flattens it into a 1D vector (while preserving the batch dimension), making it suitable for input to a Dense layer.
5. **Implement and Verify:** Add the Flatten() layer. Run model.summary() again to check the output shapes of all layers. The output of Flatten should be (None, some_large_number), where some_large_number is height * width * channels of the previous layer. This confirms the correct shape transition. Then, attempt to train the fixed model.