Part 1: Theoretical Understanding

Differences between TensorFlow and PyTorch

TensorFlow and PyTorch are both popular frameworks for building machine learning models, but they differ in design and use cases. TensorFlow, developed by Google, uses a static computation graph approach where you define the entire model architecture upfront before running calculations. This allows for optimizations that make deployment efficient in production environments, especially on mobile or web platforms using tools like TensorFlow Lite. PyTorch, created by Facebook, employs dynamic computation graphs, meaning the graph is built on the fly during execution. This makes debugging simpler and feels more intuitive, as you can use standard Python loops and conditionals directly in your model code.
For deployment-heavy scenarios like serving models in apps or cloud services, TensorFlow's mature ecosystem (e.g., TensorFlow Serving) is advantageous. PyTorch is often preferred in research or rapid prototyping due to its flexibility and "Pythonic" style, which accelerates experimentation. PyTorch also integrates seamlessly with libraries like NumPy, making it easier for beginners. In summary, choose TensorFlow for scalable production systems and PyTorch for iterative development or academic projects.

Use Cases for Jupyter Notebooks

Jupyter Notebooks streamline AI development through interactive, document-like coding environments. A key use case is exploratory data analysis (EDA). Instead of writing standalone scripts, you can load datasets, visualise distributions with libraries like Matplotlib, and annotate findings in markdown—all in one place. For instance, you might plot histograms to detect skewed data, then add notes explaining how you'll normalise it. This iterative process helps uncover patterns or anomalies early.
Another use is model experimentation and sharing. You can train a small neural network, display its accuracy metrics in real-time, and visualize incorrect predictions alongside code. Colleagues can then replicate results by rerunning the notebook, seeing exactly how data preprocessing, training, and evaluation connect. This is invaluable for team collaborations or documenting experiments for future reference, turning notebooks into "living reports" that combine code, outputs, and explanations.

---

spaCy vs. Basic String Operations

Basic Python string operations (e.g., splitting text or regex) treat language as raw characters, lacking linguistic context. spaCy elevates NLP by providing pre-built statistical models that understand grammar, relationships, and meaning. For example, while string operations might split "New York" into two words, spaCy recognises it as a single entity (a city) via named entity

recognition. It also handles tasks like lemmatization (reducing "running" to "run") and part-of-speech tagging, which regex alone cannot achieve reliably.
Additionally, spaCy offers optimized pipelines for efficiency—processing thousands of documents rapidly—whereas manual string operations scale poorly. Features like dependency parsing reveal sentence structure (e.g., identifying subjects/objects), enabling advanced applications like chatbots that need grammatical context. Simply put, spaCy automates complex linguistic tasks that would require extensive, error-prone code with basic strings, making NLP development faster and more accurate.

Comparative Analysis

Target Applications

- Scikit-learn:
  Optimised for classical/traditional machine learning tasks using structured/tabular data. It supports algorithms like linear regression, decision trees, SVMs, clustering (k-means), and ensemble methods. Ideal for applications such as customer segmentation, price prediction, and data preprocessing 158.
  Example use case: Predicting loan defaults using a Random Forest classifier.
- TensorFlow:
  Built for deep learning and neural networks, especially with unstructured data (images, text, audio). Dominates in computer vision (CNNs), natural language processing (RNs, transformers), and large-scale AI deployments (e.g., mobile apps via TensorFlow Lite) 1815.
  Example use case: Real-time image recognition in a mobile app.

Ease of Use for Beginners

- Scikit-learn:
  - Low barrier to entry: Simple, consistent API (e.g., model.fit(), model.predict()).
  - Minimal setup: Models like logistic regression require ≤5 lines of code 135.
  - Integrated workflows: Handles preprocessing, training, and evaluation in one library 814.
- TensorFlow:
  - Steeper learning curve: Requires understanding neural network architecture (layers, optimisers).
  - Flexibility vs. complexity: High-level APIs (Keras) simplify basics, but custom models need low-level operations 815.
  - Debugging challenges: Dynamic graphs (PyTorch) are easier; TensorFlow’s static graphs complicate troubleshooting

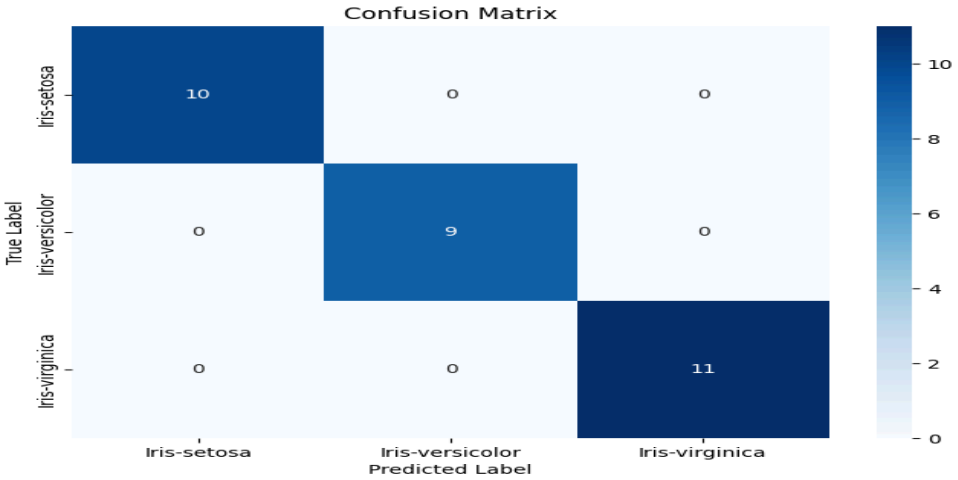
Community Support

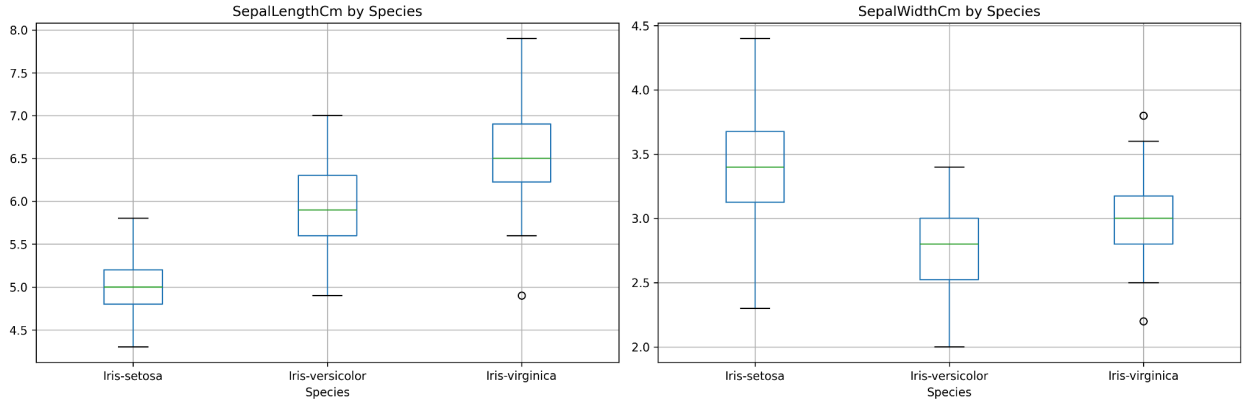
Aspect	Scikit-learn	TensorFlow
Documentation	Extensive, beginner-friendly tutorials	Comprehensive but technical
User Base	Strong in academia/research	Industry-dominated (Google, startups)
Resources	Many courses (e.g., Coursera, O'Reilly books)	Specialized forums (TF Hub, TF Forum)
Integration	Works with pandas/NumPy; limited deployment	Production tools (TF Serving, TF Lite)

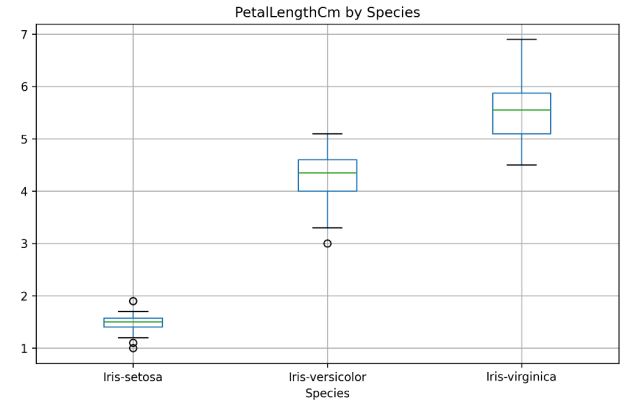
Screenshots of the three models.

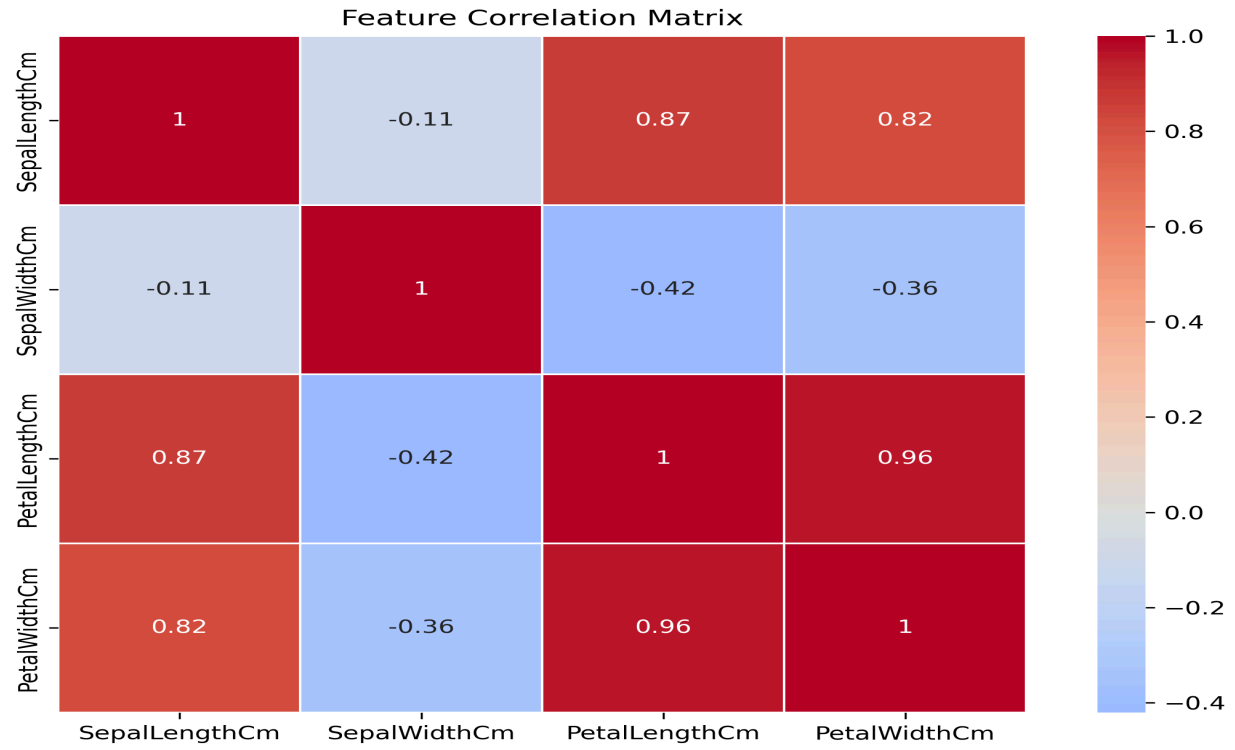
Classical ML with Scikit-learn (Iris Classification)

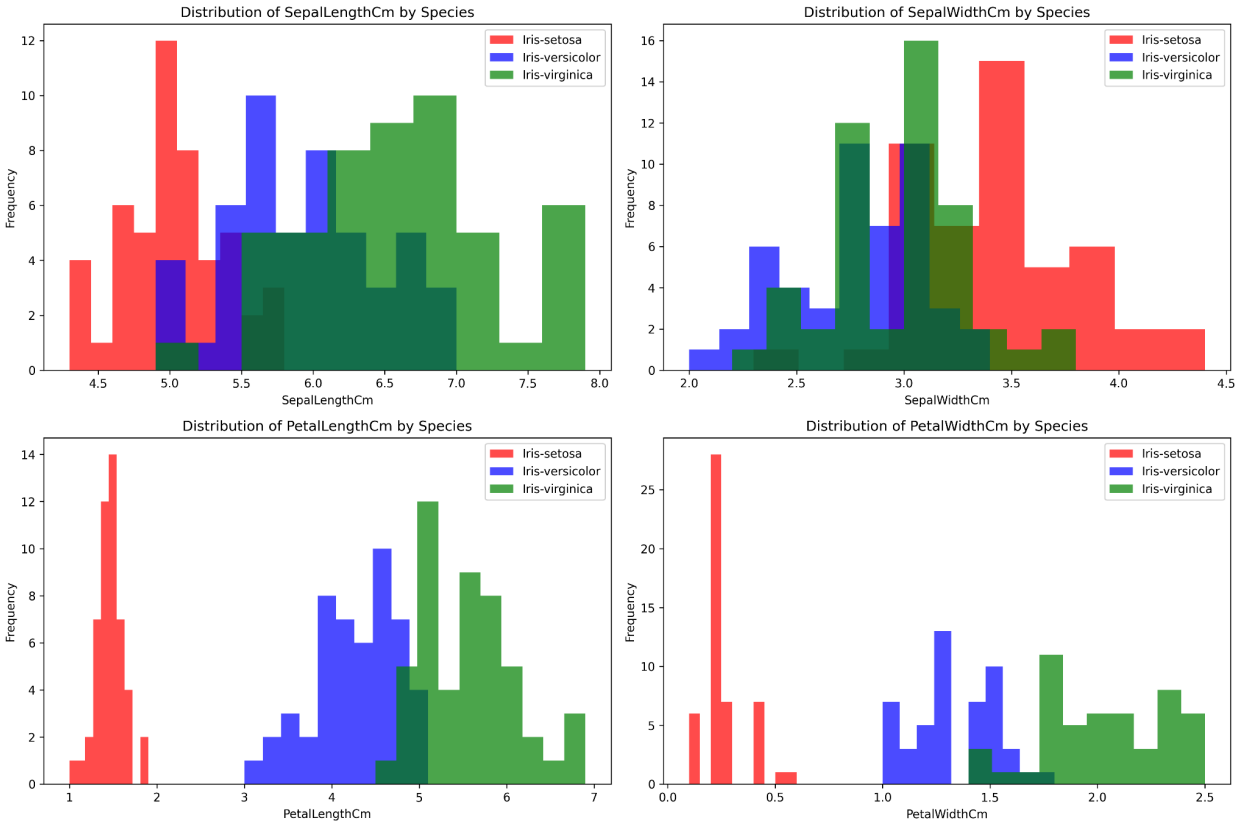
- Dataset: Iris Species (150 samples, 4 features, 3 classes)
- Model: Decision Tree Classifier
- Goal: Predict iris species with high accuracy

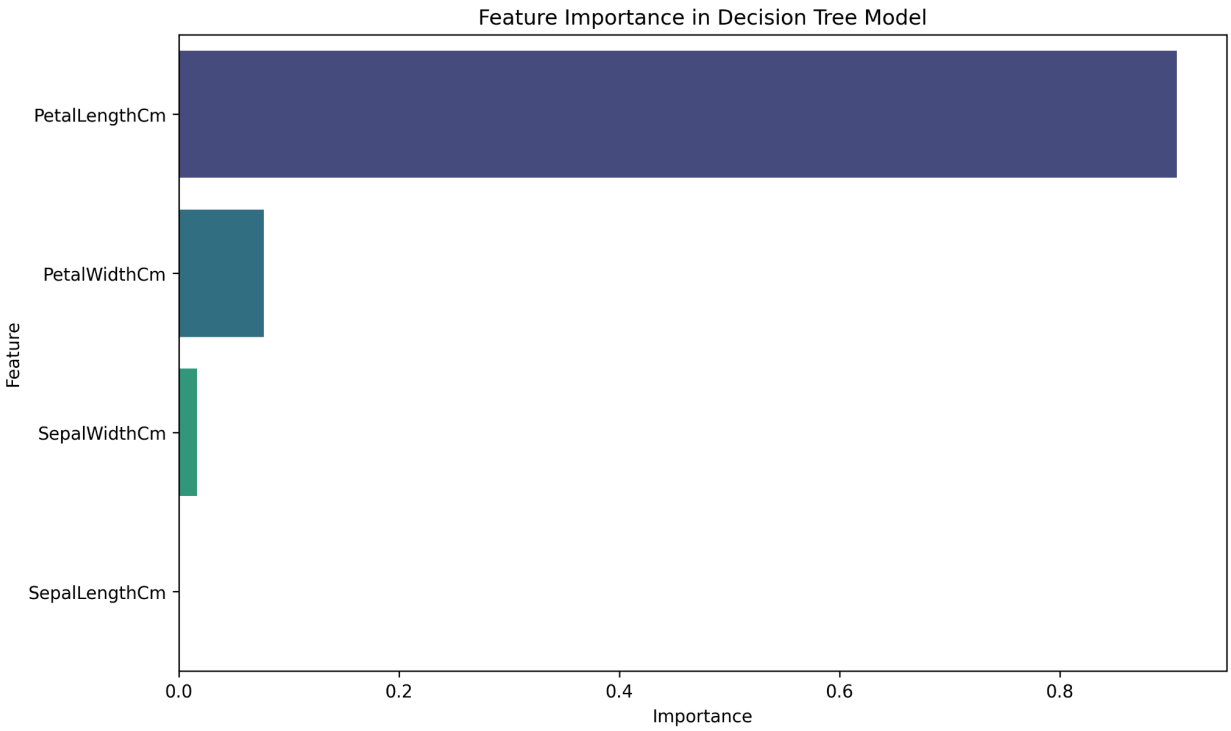






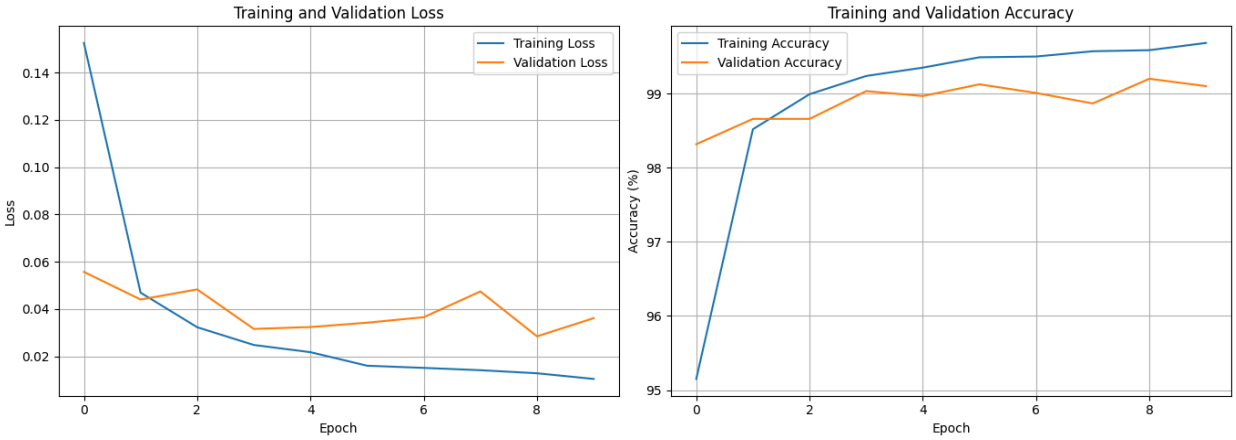

Feature Correlation Matrix
SepalLengthCm
SepalWidthCm
PetalLengthCm
PetalWidthCm
1
-0.11
0.87
0.82
-0.11
1
-0.42
-0.36
0.87
-0.42
1
0.96
0.82
-0.36
0.96
1
SepalLengthCm
SepalWidthCm
PetalLengthCm
PetalWidthCm
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4

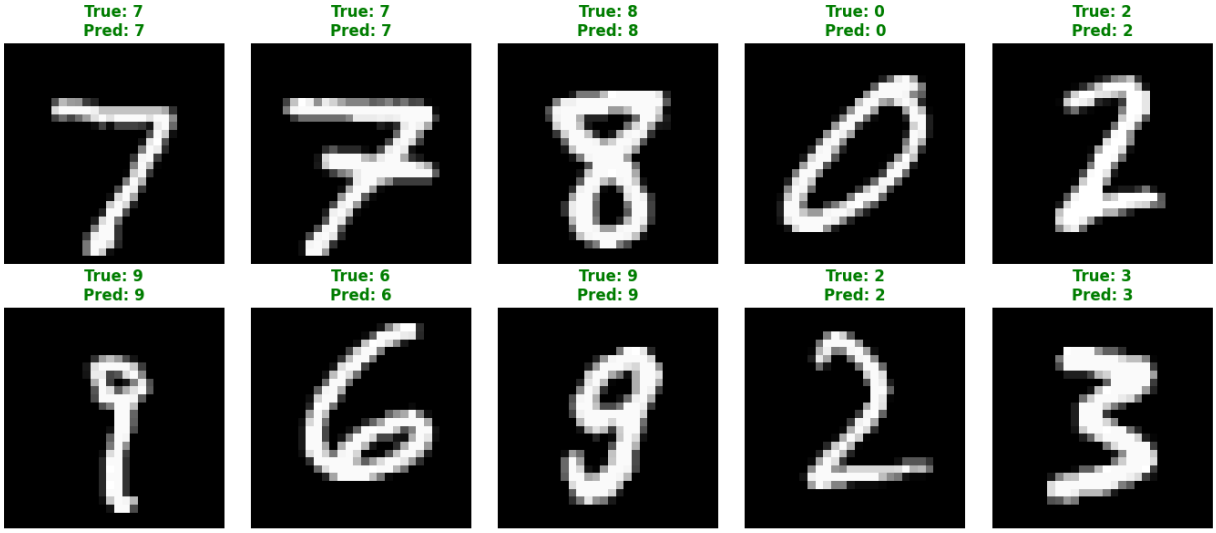

Distribution of SepalLengthCm by Species
Iris-setosa
Iris-versicolor
Iris-virginica
Frequency
SepalLengthCm
Distribution of SepalWidthCm by Species
Iris-setosa
Iris-versicolor
Iris-virginica
Frequency
SepalWidthCm
Distribution of PetalLengthCm by Species
Iris-setosa
Iris-versicolor
Iris-virginica
Frequency
PetalLengthCm
Distribution of PetalWidthCm by Species
Iris-setosa
Iris-versicolor
Iris-virginica
Frequency
PetalWidthCm

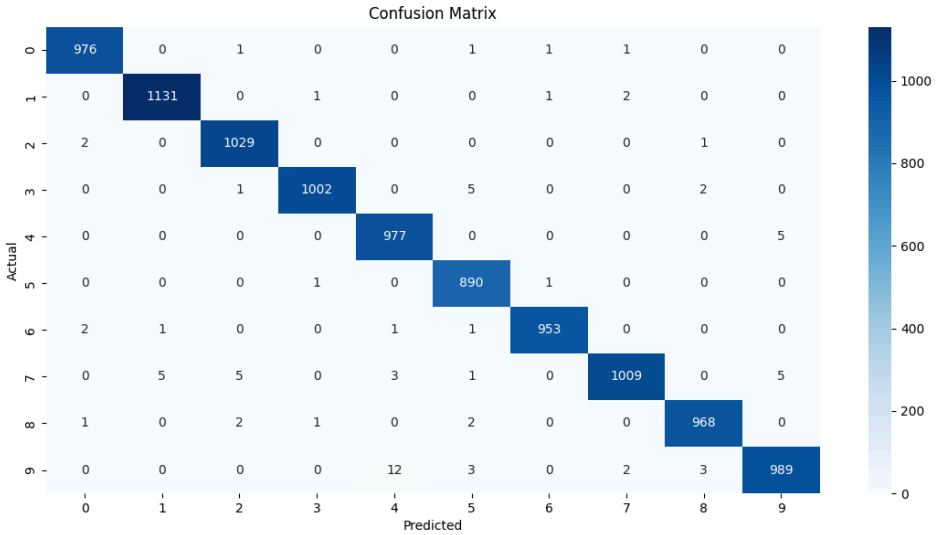

Feature Importance in Decision Tree Model
PetalLengthCm
PetalWidthCm
SepalWidthCm
SepalLengthCm
Feature
0.0
0.2
0.4
0.6
0.8
Importance

Task 2: Deep Learning with TensorFlow (MNIST Digit Classification)

- Dataset: MNIST Handwritten Digits (70,000 28x28 grayscale images)
- Model: Convolutional Neural Network (CNN)
- Goal: Achieve >95% test accuracy







Task 3: NLP with spaCy (Amazon Review Analysis)

- Dataset: Amazon Product Reviews
- Tasks:
- Named Entity Recognition (product names, brands)
- Rule-based sentiment analysis

Results

Sample Review:
"The new iPhone camera by Apple is revolutionary! Though battery life could be better."

Entities:
- iPhone (PRODUCT)
- Apple (ORG)

Sentiment: Positive (Positive words: 'revolutionary')

- Named Entity Recognition (product names, brands)
- Rule-based sentiment analysis

1. Ethical Considerations

Ethical Considerations in MNIST and Amazon Reviews Models

When building models like MNIST digit classifiers or Amazon review sentiment analysers, it's crucial to identify and mitigate biases that could lead to unfair outcomes. In MNIST, biases often emerge from handwriting variations—for instance, digits written with regional styles (like a '7' with a crossbar common in Europe) or faint strokes might be misclassified if the training data lacks diversity. Similarly, Amazon review models can exhibit biases if they overrepresent certain demographics (e.g., English-speaking users) or misinterpret cultural expressions (like sarcasm in reviews).

Tools like TensorFlow Fairness Indicators help address these issues by quantifying disparities across subgroups. For MNIST, you could slice performance metrics by digit styles to detect accuracy gaps—say, if '4's written with closed tops consistently underperform. The tool then visualizes these gaps in a dashboard, allowing you to adjust decision thresholds or add data augmentations (e.g., rotating digit images) to balance performance. For Amazon reviews, it could compare sentiment accuracy across product categories or demographic keywords, highlighting if reviews mentioning "budget" items are misclassified more often than "luxury" ones.

spaCy’s rule-based systems excel in preprocessing text to reduce lexical biases. In Amazon reviews, you could create rules to flag high-risk phrases—like cultural slang or demographic terms—and either anonymize them or sample additional reviews containing these terms to rebalance training data. For example, spaCy’s dependency parsing could identify sarcastic structures (e.g., "BEST product ever... said no one") that basic sentiment models might miss.

Combining both tools creates a robust workflow: spaCy cleans and structures input data to minimise hidden biases, while Fairness Indicators continuously audits model outputs. This is especially important for real-world deployment, ensuring your MNIST classifier works equally well for all handwriting styles or that your review system doesn’t disadvantage non-native speakers. Ultimately, ethical AI isn’t a one-time fix; it requires iterative monitoring with these tools to uphold fairness as models evolve.

2. Troubleshooting Challenge

Imagine you're training a neural network for image classification, but your TensorFlow script crashes with errors like dimension mismatches or incorrect loss functions. Let's walk through common issues and fixes. First, dimension mismatches often occur when your model's output shape doesn't match the label format. For example, if your model outputs a 10-unit softmax layer for MNIST digits (shape [batch_size, 10]), but your labels are integers ([batch_size]), TensorFlow throws an error because the loss function expects matching dimensions. To fix this, either use sparse categorical crossentropy (which handles integer labels) or one-hot encode your labels to match the model's output shape.
Another frequent error is choosing the wrong loss function. If you're doing binary classification (e.g., positive/negative Amazon reviews) but accidentally use categorical crossentropy (designed for 3+ classes), your model will fail. Switch to binary crossentropy for single-label binary tasks. Also, check activation functions: a sigmoid activation is needed for binary classification in the last layer, while softmax suits multi-class tasks.
Let’s fix a concrete bug. Suppose your script crashes with InvalidArgumentError: logits and labels must have the same shape. Inspect your model’s output shape with model.summary() and compare it to your label tensor’s shape. If labels are integers, add tf.keras.losses.SparseCategoricalCrossentropy() instead of standard categorical crossentropy. If you see NaN values in training, your learning rate might be too high, or you might have exploding gradients—add gradient clipping with clipvalue=1.0 in your optimiser.
Lastly, always validate data pipelines. Use tf.debugging.assert_shapes(...) to enforce tensor dimensions during preprocessing. For instance, if an image augmentation layer accidentally removes a channel dimension, catch it early. Remember: 90% of "model not learning" issues stem from data problems, not architecture flaws—double-check input normalisation and label encoding before tweaking layers.

Fixed Code Snippet Example

python
Copy
Download

```
# Buggy original
model.compile(optimizer='adam',
              loss='categorical_crossentropy')  # Mismatched loss for integer labels

# Fixed version
model.compile(optimizer=tf.keras.optimizers.Adam(clipvalue=1.0),
              loss=tf.keras.losses.SparseCategoricalCrossentropy())  # Handles integer labels
```