INTRODUCTION TO MACHINE LEARNING

AIGC-5102-0TA

Final Project

December 5th, 2025

David Abawi - N10005384

Eitan Berger - N10005688

Hugo Figueira – N10003998

Joshua Grima - N01402874

Safak Ozkarahan - N01778623

1. Introduction

The MNIST dataset contains 70,000 grayscale images of handwritten digits (0–9), with 60,000 training images and 10,000 test images. Each image has a resolution of 28×28 pixels. The goal of this project is to build and compare several machine learning models that can automatically classify each image into the correct digit class.

In this project we worked with six models:

- Multiclass Logistic Regression
- Multiclass Decision Forest
- Multiclass Boosted Decision Tree
- Multiclass Neural Network (fully connected MLP)
- k-Nearest Neighbours (kNN)
- Convolutional Neural Network (CNN)

Four models are implemented and tuned in Azure Machine Learning Designer, while kNN and CNN are implemented as Python script modules. For each model we perform hyperparameter tuning using grid search or a small parameter search, then evaluate on the test set using accuracy, precision, recall, F1-score, and confusion matrices. Our target is to reach at least 95% overall performance and to identify which model works best and why.

2. Dataset and Data Preprocessing

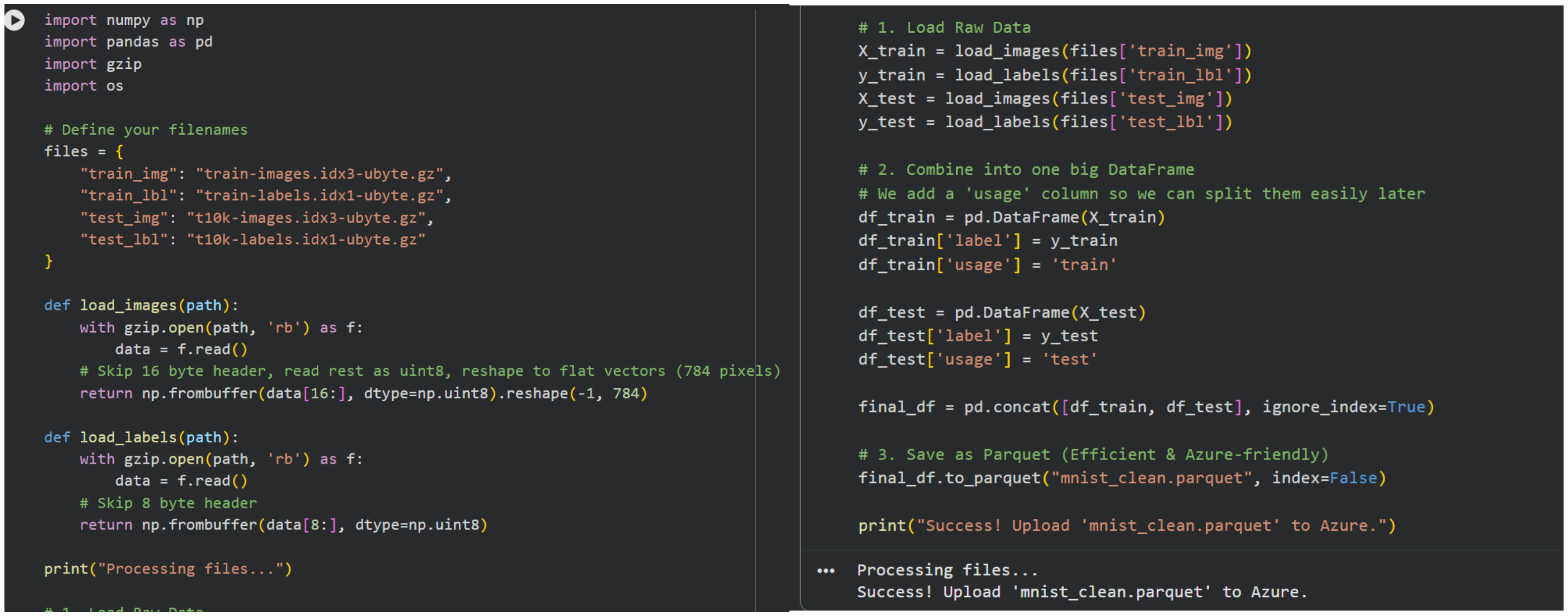
2.1 Raw data and unified dataset

The original MNIST data is provided as four gzip-compressed binary files: training images, training labels, test images and test labels. We first converted these files into a single tabular dataset that can be easily ran in Azure ML.

Using Python, we:

- Read the binary files, skipping the header bytes.
- Reshape each 28×28 image into a 784-dimensional row vector.
- Create two DataFrames:
    - Training set: 60,000 rows, 784 pixel columns, a label column (0–9), and a usage = "train" flag.
    - Test set: 10,000 rows with the same structure and usage = "test".

We then concatenate the two DataFrames into one table and save it as *mnist_clean.parquet*. This Parquet file is uploaded to Azure Machine Learning and used as the input to our Designer pipeline.



```python
import numpy as np
import pandas as pd
import gzip
import os

# Define your filenames
files = {
    "train_img": "train-images.idx3-ubyte.gz",
    "train_lbl": "train-labels.idx1-ubyte.gz",
    "test_img": "t10k-images.idx3-ubyte.gz",
    "test_lbl": "t10k-labels.idx1-ubyte.gz"
}

def load_images(path):
    with gzip.open(path, 'rb') as f:
        data = f.read()
    # Skip 16 byte header, read rest as uint8, reshape to flat vectors (784 pixels)
    return np.frombuffer(data[16:], dtype=np.uint8).reshape(-1, 784)

def load_labels(path):
    with gzip.open(path, 'rb') as f:
        data = f.read()
    # Skip 8 byte header
    return np.frombuffer(data[8:], dtype=np.uint8)

print("Processing files...")
```

```python
# 1. Load Raw Data
X_train = load_images(files['train_img'])
y_train = load_labels(files['train_lbl'])
X_test = load_images(files['test_img'])
y_test = load_labels(files['test_lbl'])

# 2. Combine into one big DataFrame
# We add a 'usage' column so we can split them easily later
df_train = pd.DataFrame(X_train)
df_train['label'] = y_train
df_train['usage'] = 'train'

df_test = pd.DataFrame(X_test)
df_test['label'] = y_test
df_test['usage'] = 'test'

final_df = pd.concat([df_train, df_test], ignore_index=True)

# 3. Save as Parquet (Efficient & Azure-friendly)
final_df.to_parquet("mnist_clean.parquet", index=False)

print("Success! Upload 'mnist_clean.parquet' to Azure.")
```

```
Processing files...
Success! Upload 'mnist_clean.parquet' to Azure.
```

Figure 1. Python script for mnist_clean.parquet

2.2 Preparation in Azure Designer

Inside Azure ML Designer, we use a Python script module (azureml_main) to prepare the data:

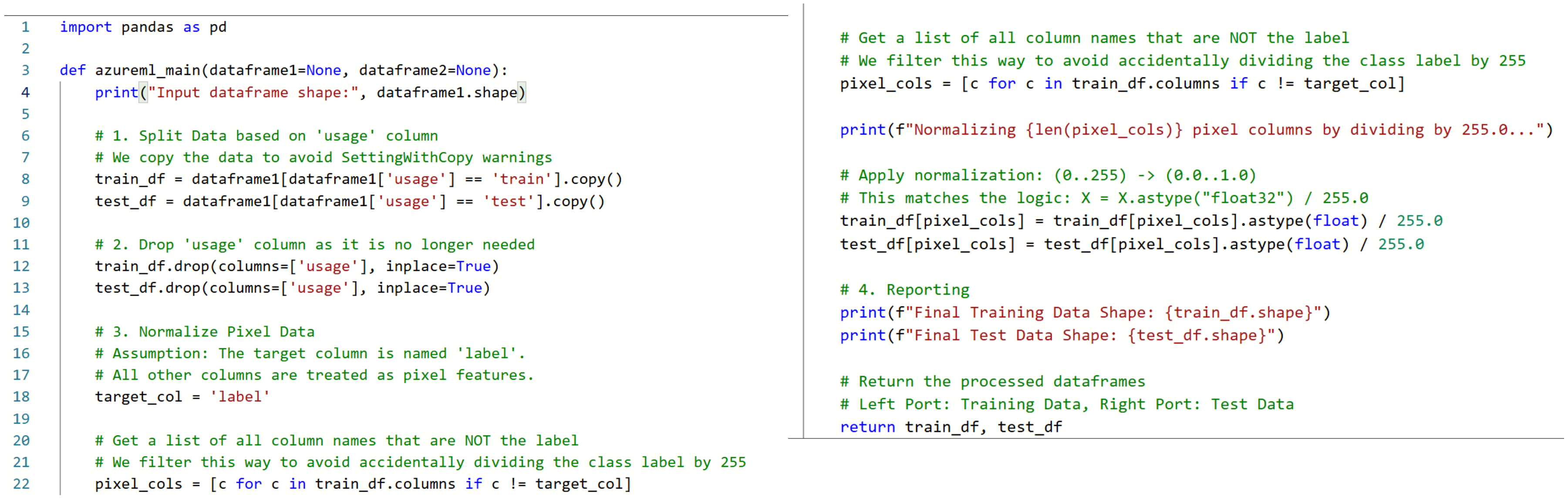
1. **Split by usage:**
    - Rows with usage = "train" → training table
    - Rows with usage = "test" → test table
2. **Drop helper column:** the usage column is removed from both tables.
3. **Separate target and features:**
    - Target column: label
    - All remaining columns: pixel features

4. **Normalize pixel values:**
   We convert pixel columns to float and divide by 255.0 so that all inputs lie in the range [0, 1]. The label column remains unchanged.

The script outputs the normalized training table on the left port and the normalized test table on the right port. The training table is then split again (for Designer models) into a train and validation subset using the Split Data module (e.g., 80/20 split).

This preprocessing has two main effects:

- It stabilizes gradient-based models (logistic regression, MLP, CNN) by keeping input scales reasonable.
- It makes distance-based methods like kNN more meaningful, because all features are on the same scale.



```python
import pandas as pd

def azureml_main(dataframe1=None, dataframe2=None):
    print("Input dataframe shape:", dataframe1.shape)

    # 1. Split Data based on 'usage' column
    # We copy the data to avoid SettingWithCopy warnings
    train_df = dataframe1[dataframe1['usage'] == 'train'].copy()
    test_df = dataframe1[dataframe1['usage'] == 'test'].copy()

    # 2. Drop 'usage' column as it is no longer needed
    train_df.drop(columns=['usage'], inplace=True)
    test_df.drop(columns=['usage'], inplace=True)

    # 3. Normalize Pixel Data
    # Assumption: The target column is named 'label'.
    # All other columns are treated as pixel features.
    target_col = 'label'

    # Get a list of all column names that are NOT the label
    # We filter this way to avoid accidentally dividing the class label by 255
    pixel_cols = [c for c in train_df.columns if c != target_col]
```

```python
# Get a list of all column names that are NOT the label
# We filter this way to avoid accidentally dividing the class label by 255
pixel_cols = [c for c in train_df.columns if c != target_col]

print(f"Normalizing {len(pixel_cols)} pixel columns by dividing by 255.0...")

# Apply normalization: (0..255) -> (0.0..1.0)
# This matches the logic: X = X.astype("float32") / 255.0
train_df[pixel_cols] = train_df[pixel_cols].astype(float) / 255.0
test_df[pixel_cols] = test_df[pixel_cols].astype(float) / 255.0

# 4. Reporting
print(f"Final Training Data Shape: {train_df.shape}")
print(f"Final Test Data Shape: {test_df.shape}")

# Return the processed dataframes
# Left Port: Training Data, Right Port: Test Data
return train_df, test_df
```

Figure 2. Python script for data preparation

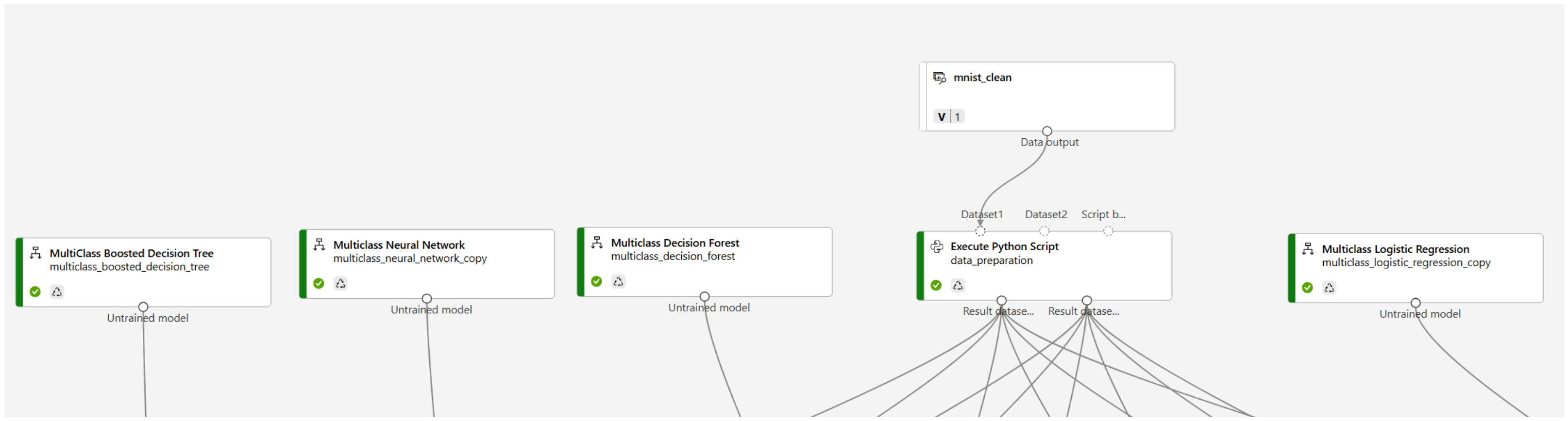

mnist_clean
V 1
Data output
Dataset1
Dataset2
Script b...
MultiClass Boosted Decision Tree
multiclass_boosted_decision_tree
Untrained model
Multiclass Neural Network
multiclass_neural_network_copy
Untrained model
Multiclass Decision Forest
multiclass_decision_forest
Untrained model
Execute Python Script
data_preparation
Result datase...
Result datase...
Multiclass Logistic Regression
multiclass_logistic_regression_copy
Untrained model

Figure 3. Designer part of data preparation connections

3. Methods

We group methods into Designer models and Python script models.
For all Designer models we use Azure's Tune Model Hyperparameters module in "Entire grid" mode with Accuracy as the optimization metric and label as the target.

3.1 Azure Designer models

Four of the models in this project are implemented using built-in components in Azure Machine Learning Designer. In all cases, we connect the model to the Tune Model Hyperparameters module, which runs a small grid search over reasonable settings and selects the best configuration based on validation accuracy. This lets us focus on comparing the behaviour of different algorithms rather than manually guessing hyperparameters.

3.1.1 Multiclass Logistic Regression

Multiclass logistic regression is used as a linear baseline. It assumes that each class can be separated from the others with a linear decision boundary in the 784-dimensional pixel space. The model is fast to train and easy to interpret, but it cannot capture more complex patterns in the images. We include it to see how far a simple linear model can go on MNIST and to provide a reference point for the more flexible methods.

3.1.2 Multiclass Decision Forest

The Multiclass Decision Forest is an ensemble of decision trees combined by bagging. Each tree is trained on a slightly different sample of the data, and final predictions are made by majority vote. Forests can model non-linear relationships and interactions between pixels without requiring feature engineering. In this project we use the forest as a classical non-linear baseline, to see how a tree ensemble compares to neural networks and boosted trees on the same normalized input.

3.1.3 Multiclass Boosted Decision Tree

The Multiclass Boosted Decision Tree model builds a sequence of shallow trees, where each new tree focuses on correcting the mistakes of the previous ones. This boosting strategy often achieves higher accuracy than a single tree or a bagged forest, at the cost of slightly more tuning. We include the boosted tree as a strong, general-purpose ensemble method and expect it to be one of the top performers on MNIST.

3.1.4 Multiclass Neural Network (MLP)

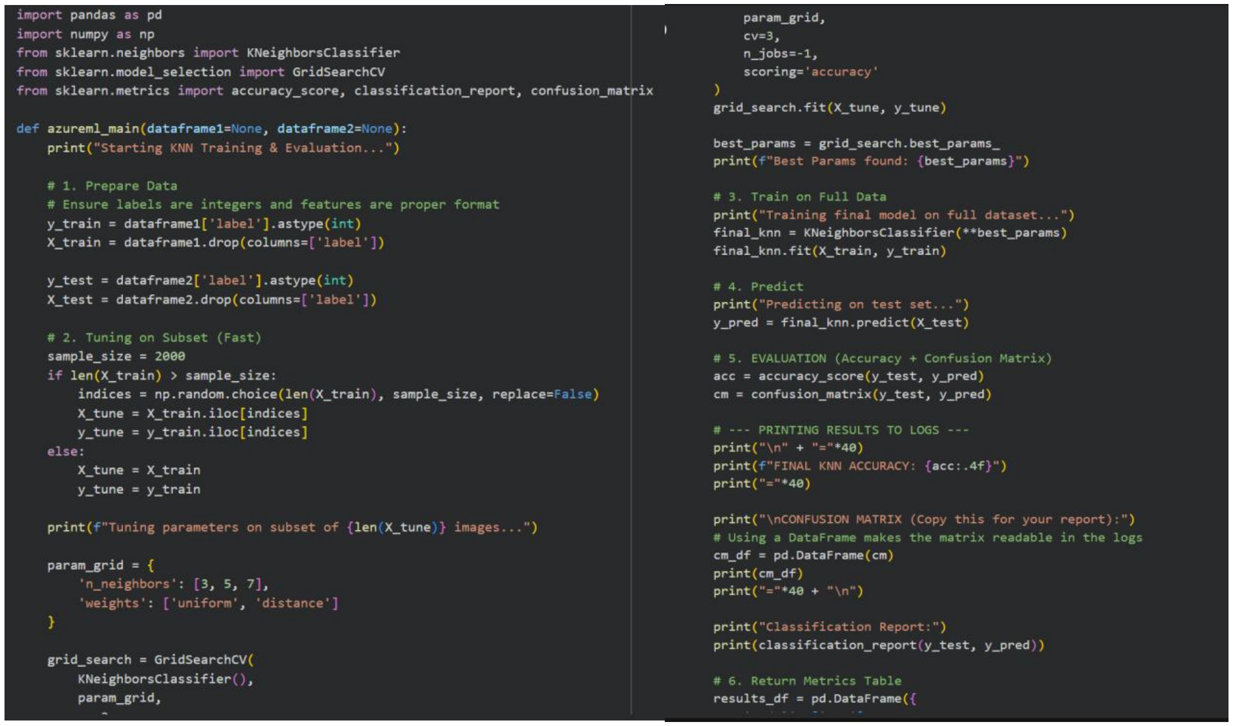
The Multiclass Neural Network in Designer is a fully connected multilayer perceptron that operates directly on the flattened 784-dimensional input. It can learn non-linear combinations of pixels and internal representations that are not accessible to linear models. We use this model to represent a simple neural network approach within Designer and to compare it both to the classical methods (logistic regression, forests, boosted trees) and to the CNN implemented in Python, which works on the 2D image structure.

3.2 Python script models

3.2.1 k-Nearest Neighbours (kNN)

The kNN model is implemented as a Python script using scikit-learn. It classifies each test image based on the labels of its nearest neighbours in the normalized pixel space.
Steps:

1. Split Designer inputs into X_train, y_train, X_test, y_test.
2. Randomly select 2,000 training images to use for hyperparameter tuning.
3. Run GridSearchCV with 3-fold cross-validation over:
    - n_neighbors (k values)
    - weights ("uniform" vs "distance")
4. Train the final KNN on all training data using the best parameters from the grid search.
5. Evaluate on the test set and print accuracy, precision, recall, F1-score and the confusion matrix.



```python
import pandas as pd
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

def azureml_main(dataframe1=None, dataframe2=None):
    print("Starting KNN Training & Evaluation...")

    # 1. Prepare Data
    # Ensure labels are integers and features are proper format
    y_train = dataframe1['label'].astype(int)
    X_train = dataframe1.drop(columns=['label'])

    y_test = dataframe2['label'].astype(int)
    X_test = dataframe2.drop(columns=['label'])

    # 2. Tuning on Subset (Fast)
    sample_size = 2000
    if len(X_train) > sample_size:
        indices = np.random.choice(len(X_train), sample_size, replace=False)
        X_tune = X_train.iloc[indices]
        y_tune = y_train.iloc[indices]
    else:
        X_tune = X_train
        y_tune = y_train

    print(f"Tuning parameters on subset of {len(X_tune)} images...")

    param_grid = {
        'n_neighbors': [3, 5, 7],
        'weights': ['uniform', 'distance']
    }

    grid_search = GridSearchCV(
        KNeighborsClassifier(),
        param_grid,
```

```python
        param_grid,
        cv=3,
        n_jobs=-1,
        scoring='accuracy'
    )
    grid_search.fit(X_tune, y_tune)

    best_params = grid_search.best_params_
    print(f"Best Params found: {best_params}")

    # 3. Train on Full Data
    print("Training final model on full dataset...")
    final_knn = KNeighborsClassifier(**best_params)
    final_knn.fit(X_train, y_train)

    # 4. Predict
    print("Predicting on test set...")
    y_pred = final_knn.predict(X_test)

    # 5. EVALUATION (Accuracy + Confusion Matrix)
    acc = accuracy_score(y_test, y_pred)
    cm = confusion_matrix(y_test, y_pred)

    # --- PRINTING RESULTS TO LOGS ---
    print("\n" + "="*40)
    print(f"FINAL KNN ACCURACY: {acc:.4f}")
    print("="*40)

    print("\nCONFUSION MATRIX (Copy this for your report):")
    # Using a DataFrame makes the matrix readable in the logs
    cm_df = pd.DataFrame(cm)
    print(cm_df)
    print("="*40 + "\n")

    print("Classification Report:")
    print(classification_report(y_test, y_pred))

    # 6. Return Metrics Table
    results_df = pd.DataFrame({
```

Figure 4. kNN Python script

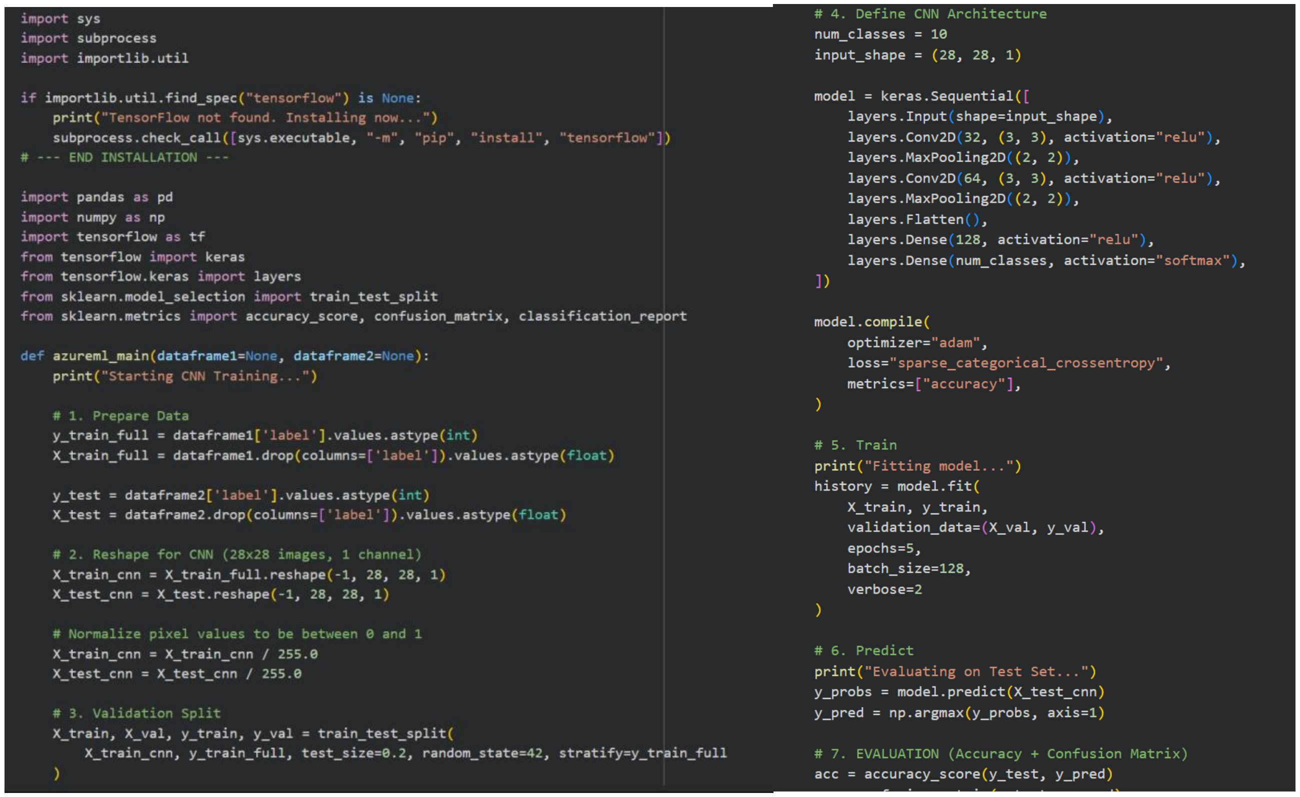
3.2.2 Convolutional Neural Network (CNN)

The CNN is implemented in TensorFlow/Keras and works with the original 2D image structure.
Steps:

1. Convert the pixel columns into NumPy arrays and reshape to (n_samples, 28, 28, 1).
2. Normalize pixel values to the range [0, 1].
3. Split the training set into train (80%) and validation (20%) using train_test_split with stratification.
4. Define a small CNN:
    - Conv2D(32, 3×3, ReLU) → MaxPooling(2×2)
    - Conv2D(64, 3×3, ReLU) → MaxPooling(2×2)
    - Flatten → Dense(128, ReLU) → Dense(10, softmax)
5. Compile with Adam optimizer and sparse_categorical_crossentropy loss.

6. Train for 5 epochs with batch size 128, monitoring validation accuracy.
7. Evaluate on the test set and compute accuracy and confusion matrix.

Here we perform manual tuning by selecting a reasonable architecture and number of epochs rather than a full grid search, as CNN training is more expensive.



```python
import sys
import subprocess
import importlib.util

if importlib.util.find_spec("tensorflow") is None:
    print("TensorFlow not found. Installing now...")
    subprocess.check_call([sys.executable, "-m", "pip", "install", "tensorflow"])
# --- END INSTALLATION ---

import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

def azureml_main(dataframe1=None, dataframe2=None):
    print("Starting CNN Training...")

    # 1. Prepare Data
    y_train_full = dataframe1['label'].values.astype(int)
    X_train_full = dataframe1.drop(columns=['label']).values.astype(float)

    y_test = dataframe2['label'].values.astype(int)
    X_test = dataframe2.drop(columns=['label']).values.astype(float)

    # 2. Reshape for CNN (28x28 images, 1 channel)
    X_train_cnn = X_train_full.reshape(-1, 28, 28, 1)
    X_test_cnn = X_test.reshape(-1, 28, 28, 1)

    # Normalize pixel values to be between 0 and 1
    X_train_cnn = X_train_cnn / 255.0
    X_test_cnn = X_test_cnn / 255.0

    # 3. Validation Split
    X_train, X_val, y_train, y_val = train_test_split(
        X_train_cnn, y_train_full, test_size=0.2, random_state=42, stratify=y_train_full
    )
```

```python
# 4. Define CNN Architecture
num_classes = 10
input_shape = (28, 28, 1)

model = keras.Sequential([
    layers.Input(shape=input_shape),
    layers.Conv2D(32, (3, 3), activation="relu"),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation="relu"),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(128, activation="relu"),
    layers.Dense(num_classes, activation="softmax"),
])

model.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)

# 5. Train
print("Fitting model...")
history = model.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=5,
    batch_size=128,
    verbose=2
)

# 6. Predict
print("Evaluating on Test Set...")
y_probs = model.predict(X_test_cnn)
y_pred = np.argmax(y_probs, axis=1)

# 7. EVALUATION (Accuracy + Confusion Matrix)
acc = accuracy_score(y_test, y_pred)
```

Figure 5. CNN Python script

4. Evaluation and Results

All models are trained on the same preprocessed MNIST data. The original training split (60,000 images) is used for model training and internal validation, and the original test split (10,000 images) is used once at the end for evaluation. Pixel values are normalized to [0, 1], which stabilizes gradient-based training (logistic regression, MLP, CNN) and makes distance calculations for kNN meaningful because all features share the same scale.

Model	Implementation	Tuning approach	Key best settings (summary)	Test accuracy
Boosted Decision Tree	Designer	Tune module, full grid	~20 leaves, 50–100 trees, tuned LR	**0.9799**
Multiclass Neural Network	Designer	Tune module, full grid	128–256 hidden units, tuned LR/iters	**0.9727**
k-Nearest Neighbours	Python script	GridSearchCV on 2k subset	k = 3, distance weights	**0.9717**
Decision Forest	Designer	Tune module, full grid	8–32 trees, depth 32, tuned min leaf	**0.9655**
Convolutional NN	Python script	Manual tuning (fixed arch, 5 epochs)	2 conv layers, Dense(128), Adam	**0.9652**
Logistic Regression	Designer	Tune module, full grid	Tuned L2 and tolerance	**0.9261**

Table 1. Summary of test performance for all models (accuracy and, optionally, micro/macro F1).

4.1 Overall model comparison

Table 1 summarises the test performance of all six models. We report accuracy as the main metric, together with precision, recall and F1-score where available.

Across the models, we observe that all non-linear methods clearly outperform the linear baseline:

- Multiclass Boosted Decision Tree is the best model with test accuracy of about 97.99% and almost identical micro/macro precision and recall.
- Multiclass Neural Network (MLP) and k-Nearest Neighbours follow closely, both achieving >97% accuracy.
- Multiclass Decision Forest and the CNN reach around 96–97% accuracy, comfortably above the 95% target.
- Multiclass Logistic Regression achieves around 92–93% accuracy and lower F1-scores, confirming that a purely linear decision boundary is not expressive enough for handwritten digits.

Overall, the results show that once the data is normalized, tree ensembles, kNN and neural networks all learn effective non-linear decision boundaries on MNIST, while logistic regression underfits the problem.

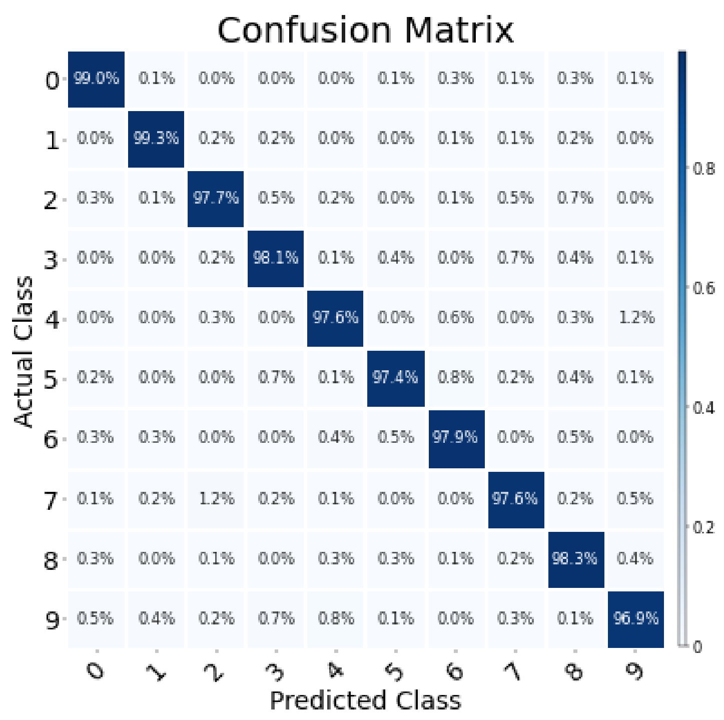


Figure 6. Boosted Decision Tree

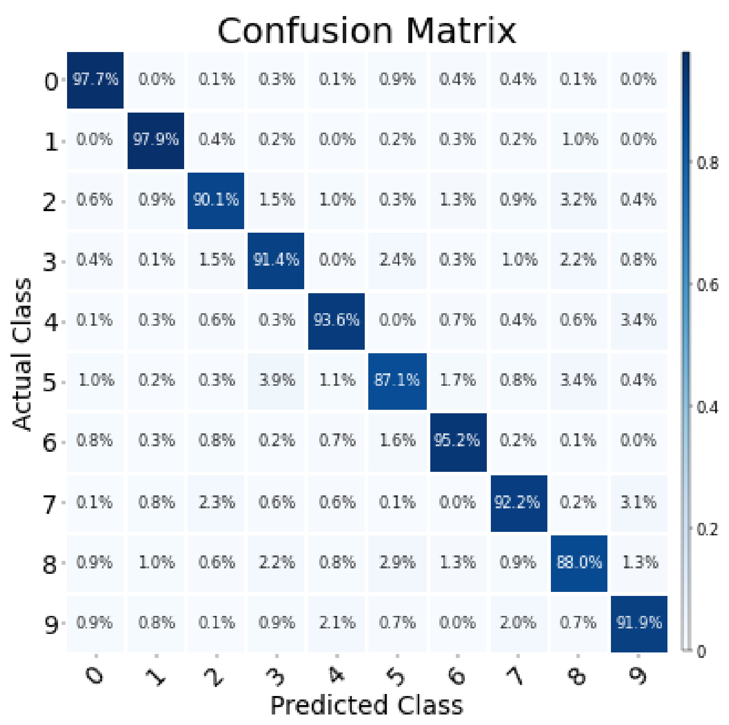


Figure 7. Logistic Regression

4.2 Confusion matrices and misclassified samples

To visualize classification behaviour in more detail, we use confusion matrices from the Evaluate Model component. Figure 6 shows the confusion matrix for the Boosted Decision Tree, and Figure 7 for Logistic Regression.

In Figure 6, the boosted tree produces a very strong diagonal: most cells on the main diagonal are above 97–99%, and off-diagonal entries are close to zero. This indicates that nearly all digits are correctly classified, with only a small number of mistakes per class.

In contrast, Figure 7 (logistic regression) has a less clean diagonal and more mass spread across off-diagonal cells. Typical errors include confusions between visually similar digits, such as 4 vs 9 and 3 vs 5, and between 5 and 6 or 8 and 9. These patterns suggest that the linear model struggles when digit shapes overlap in the input space, while the boosted tree can separate these cases using non-linear splits.

By inspecting the largest off-diagonal entries, we see recurring sources of ambiguity that are common across models:

- Digits written in an unusual style (for example, a 9 that looks like a 4, or a 5 that is almost closed like a 6).
- Faint or noisy digits, where important strokes are missing.

- Digits that are shifted or off-centre, so that relevant pixels lie near the border.

Many of these misclassified images are ambiguous even for a human observer. This indicates that, after normalization and tuning, most of the remaining errors are due to handwriting ambiguity and noise rather than a fundamental limitation of the best model.

Taken together, the numeric metrics in Table 1 and the confusion matrices in Figures 6 and 7 support the conclusion that the Multiclass Boosted Decision Tree is the strongest overall model in our setup, with the MLP and kNN providing competitive alternatives and logistic regression serving mainly as a baseline for comparison.

5. Conclusion and Future Work

In this project we built an end-to-end MNIST classification workflow in Azure Machine Learning Designer, starting from the raw binary files, creating a unified Parquet dataset, applying normalisation, and training six different models with appropriate hyperparameter tuning. All non-linear models comfortably exceeded the 95% accuracy requirement on the test set, with the Multiclass Boosted Decision Tree performing best, followed by the Multiclass Neural Network, kNN, Decision Forest and CNN. Logistic regression served as a simple baseline and highlighted the benefit of more flexible models for handwritten digit recognition.

Overall, the results show that the combination of Designer pipelines and Python script modules is effective for comparing multiple machine learning approaches on the same dataset. As a natural next step, the project could be extended by trying a slightly deeper CNN and basic data augmentation (rotations and shifts), or by deploying the best model as a small interactive demo.

6. References

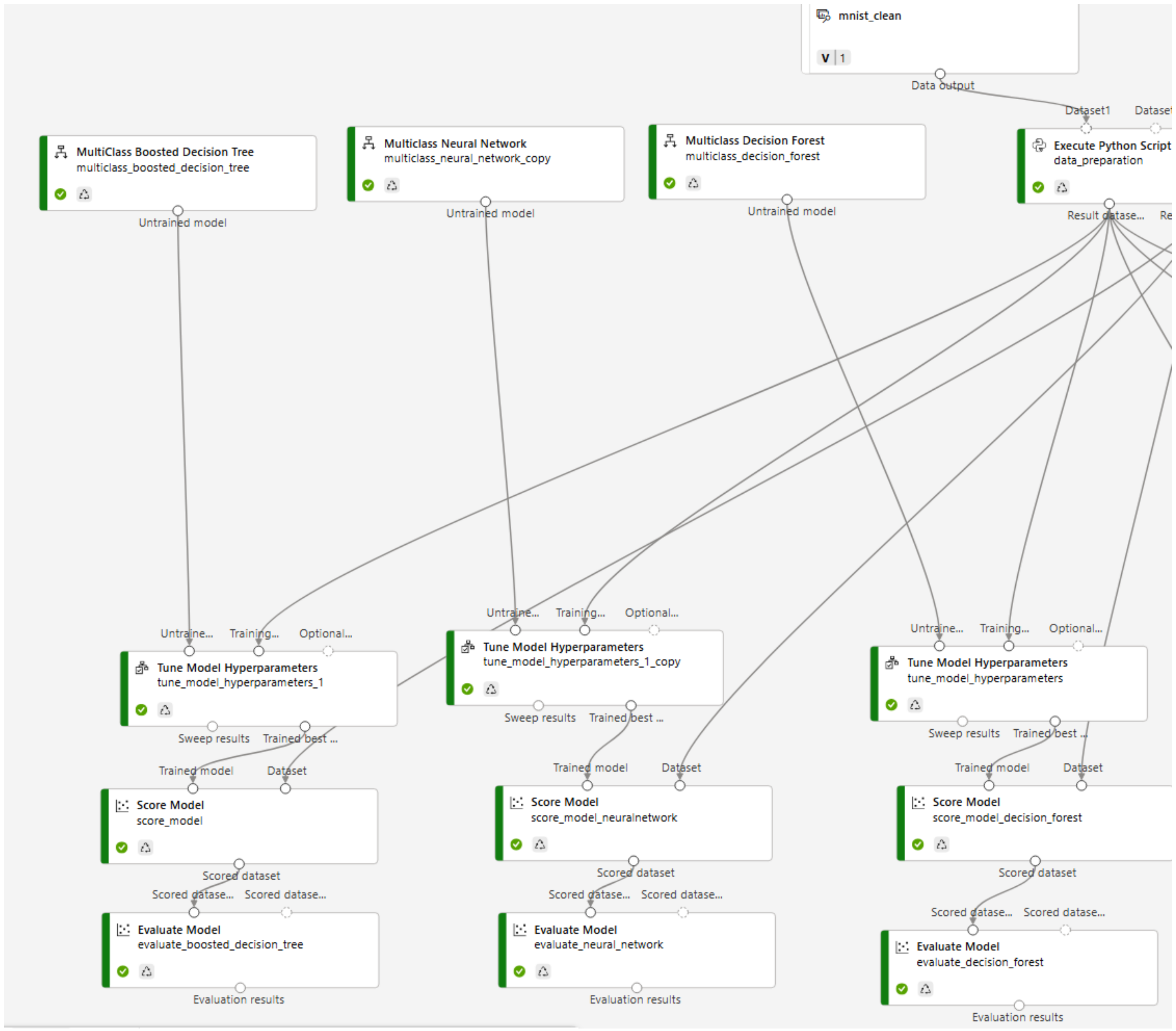
Deng, L. (2012). *The MNIST database of handwritten digit images.*

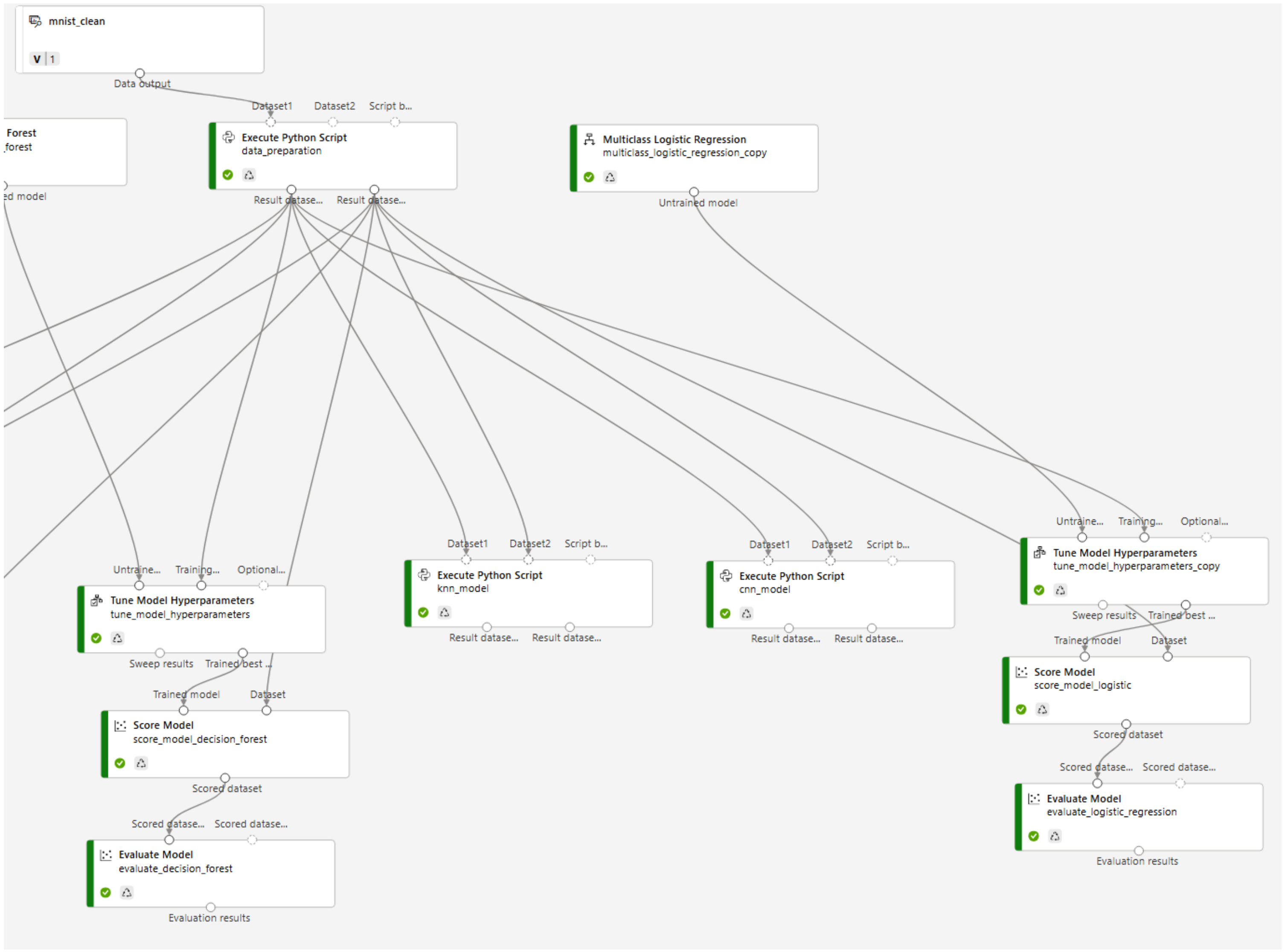
scikit-learn developers. (n.d.). *scikit-learn documentation.* https://scikit-learn.org

TensorFlow contributors. (n.d.). *TensorFlow & Keras documentation.* https://www.tensorflow.org

Microsoft. (n.d.). *Azure Machine Learning documentation.* https://learn.microsoft.com/azure/machine-learning

7. Index - Designer Screenshots





mnist_clean
V 1
Data output
Forest
forest
ed model
Dataset1
Dataset2
Script b...
Execute Python Script
data_preparation
Result datase...
Result datase...
Multiclass Logistic Regression
multiclass_logistic_regression_copy
Untrained model
Untraine...
Training...
Optional...
Tune Model Hyperparameters
tune_model_hyperparameters
Sweep results
Trained best ...
Trained model
Dataset
Score Model
score_model_decision_forest
Scored dataset
Scored datase...
Scored datase...
Evaluate Model
evaluate_decision_forest
Evaluation results
Dataset1
Dataset2
Script b...
Execute Python Script
knn_model
Result datase...
Result datase...
Dataset1
Dataset2
Script b...
Execute Python Script
cnn_model
Result datase...
Result datase...
Untraine...
Training...
Optional...
Tune Model Hyperparameters
tune_model_hyperparameters_copy
Sweep results
Trained best ...
Trained model
Dataset
Score Model
score_model_logistic
Scored dataset
Scored datase...
Scored datase...
Evaluate Model
evaluate_logistic_regression
Evaluation results