AI in Civil Engineering

CASE STUDY

Open Access

A step-by-step tutorial on machine learning for engineers unfamiliar with programming

M. Z. Naser[1,2*]

Abstract

Machine learning (ML) has garnered significant attention within the engineering domain. However, engineers without formal ML education or programming expertise may encounter difficulties when attempting to integrate ML into their work processes. This study aims to address this challenge by offering a tutorial that guides readers through the construction of ML models using Python. We introduce three simple datasets and illustrate how to preprocess the data for regression, classification, and clustering tasks. Subsequently, we navigate readers through the model development process utilizing well-established libraries such as NumPy, pandas, scikit-learn, and matplotlib. Each step, including data preparation, model training, validation, and result visualization, is covered with detailed explanations. Furthermore, we explore explainability techniques to help engineers understand the underlying behavior of their models. By the end of this tutorial, readers will have hands-on experience with three fundamental ML tasks and understand how to evaluate and explain the developed models to make engineering projects efficient and transparent.

Keywords Machine learning, Engineering, Tutorial, Python

1 Introduction

Machine learning (ML) has evolved into an effective technology that helps engineers harness the vast amounts of data generated in their domains (Thai, 2022). By seamlessly integrating extensive sets of highly nonlinear data, ML has shown promise in overcoming the limitations associated with traditional methodologies such as physical tests and numerical simulations. Such limitations often entail the necessity for idealizations, high computational expenses, and the intricate task of capturing intricate system behaviors (please refer to the following sources for additional and in-depth discussion: Karniadakis et al., 2021; Molnar et al., 2020). As such, the interest in ML continues to grow in order to address various intricate phenomena across different scales (Naser, 2019; Nguyen-Sy et al., 2020; Zarringol & Thai, 2022).

In the domain of materials engineering, for instance, ML models are increasingly being leveraged to forecast material properties like strength, durability, etc., based on compositional data and processing parameters (al-Bashiti & Naser, 2022; Ben Chaabene et al., 2020; Xie et al., 2021). Such predictive capabilities expedite the material design and selection processes. Similarly, ML approaches have also been used in infrastructure health monitoring to analyze data from sensors embedded in structures like bridges, tunnels, and buildings to detect early signs of deterioration or potential failure (Malekloo et al., 2022; Yuan et al., 2020). The implementation of predictive maintenance strategies not only enhances safety but also optimizes resource allocation for repairs. Furthermore, the field of robotics and control systems is incorporating ML techniques to bolster adaptive behaviors and decision-making within complex environments (Tapeh & Naser, 2022; You et al., 2023).

*Correspondence:
M. Z. Naser
mznaser@clemson.edu
[1] School of Civil and Environmental Engineering & Earth Sciences, Clemson University, Clemson, SC, USA
[2] Artificial Intelligence Research Institute for Science and Engineering (AIRISE), Clemson University, Clemson, SC, USA

However, alongside the significant learning curve involved for newcomers, the expanding array of ML methodologies often presents a challenge to engineers who have limited or no exposure to ML concepts. While engineers typically excel at solving well-defined physical or mathematical problems by applying analytical methods and established best practices, the adoption of ML approaches requires a different mindset, one that focuses on pattern recognition and empirical modeling. Instead of relying on first-principle equations or domain-specific assumptions alone, ML techniques leverage historical or observed data to "learn" relationships between input variables and outputs. This becomes especially advantageous in situations where articulating an explicit analytical model proves arduous or implementing it computationally expensive (Naser, 2023).

The multitude of libraries, frameworks, and algorithms presents a formidable challenge for engineers new to the field: discerning the practical steps required to construct ML models (Pine, 2019). On one hand, Python has quickly become an effective language for ML due to its extensive ecosystem of open-source packages, rendering Python an optimal choice for both prototyping and deploying models with efficiency. In addition, Python's readability and thriving community make it particularly appealing for individuals transitioning from engineering disciplines. With the aid of pre-built libraries and packages such as NumPy, pandas, and scikit-learn, the manipulation of datasets, the construction of models, and their deployment for production or research are rendered relatively uncomplicated.

This short paper presents a distinctive end-to-end tutorial that bridges the gap between theoretical ML concepts and practical engineering applications. Unlike existing resources that often focus on isolated techniques, we provide an integrated approach spanning from data setup through model explainability and visualization. In addition, we provide a complete tutorial, which, to the author's knowledge, might not exist in the capacity presented on engineering problems at the moment. We focus on three fundamental ML tasks that cover a broad range of engineering applications: regression, classification, and clustering. Regression problems are universal in predictive modeling for phenomena with a numerical output. Classification comes into play when engineers want to sort items into distinct categories, such as identifying failure modes. Clustering methods are employed when engineers seek to uncover natural groupings without predefined categories, thereby revealing hidden patterns and segments. It is worth noting that other theoretical foundations on ML tasks, as well as algorithms, also exist, and we advise interested readers to examine the following notable sources to gain a better understanding of such methods as, for brevity and given the beginner's nature of this short paper, such methods are not covered herein (Bishop, 2007; Goodfellow et al., 2016; Hastie et al., 2001).

Beyond conventional tutorials, our emphasis lies on the essential aspects of model validation and explainability, a significant deficiency in many existing resources which often neglect these fundamental engineering prerequisites. Additionally, we confront the distinct challenges of interpretability within engineering frameworks, where stakeholders necessitate evidence of a model's dependability and operational characteristics prior to integration into critical workflows. This tutorial sets itself apart by integrating explainability techniques like SHAP to illustrate how to decipher model results systematically and transparently (Lundberg & Lee, 2017). By the conclusion of this tutorial, it is our aspiration that readers will feel adept at employing machine learning techniques in their own engineering datasets. This paper offers a technically proficient yet approachable roadmap that can be utilized as a groundwork for the broader incorporation of machine learning in engineering practices.

2 Tutorial and examples

Below is a structured Python-based tutorial demonstrating how to set up a ML workflow for regression, classification, and clustering. The tutorial covers synthetic data generation, utilization of pre-existing datasets, model training, validation, plotting, and model explainability. All scripts are developed using Python.

2.1 Setup and imports

The initiation of any ML project involves setting up a functional environment by importing essential libraries. In Python, certain libraries have become staples for ML analysis. For example, NumPy and pandas are critical for data-related tasks such as loading, cleaning, and transforming data. Scikit-learn is another vital library

that provides a comprehensive range of ML algorithms, alongside functionalities for dataset splitting, feature engineering, and performance evaluation. Matplotlib is a fundamental library for visualizing data and results. In recent years, there has been an emergence of model interpretability libraries such as SHAP. Thus, the first portion of a typical ML script starts by importing the related libraries.

```
#Imports
import numpy as np                          # Package for numerical computing
import pandas as pd                         # Data analysis and manipulation tool.
import matplotlib.pyplot as plt             # Package for plotting
import shap                                 # For explainability
```

2.2 Data

Prior to the training of any ML model, it is imperative for an engineer to procure a dataset that encapsulates the nuances of the problem under consideration. For those new to the field, employing synthetic datasets is frequently an optimal approach to grasp the foundational principles without becoming ensnared in the intricacies of real-world data challenges, such as data incompleteness. Synthetic datasets can be meticulously crafted to elucidate specific learning objectives, including regression, classification, and clustering tasks.

In this section, we generate three distinct datasets (see Fig. 1). The first dataset models a linear relationship suitable for regression to demonstrate how algorithms handle continuous outputs. The second dataset is designed to produce a binary outcome, suitable for classification tasks, and the third dataset comprises a collection of random points, ideal for clustering purposes. To elaborate, the regression dataset is based on a single feature, X_reg, with 200 samples and values randomly distributed between 0 and 10. The target variable, y_reg, is generated using the following formula:

$$y = 3.5x + noise \tag{1}$$

The classification dataset comprises two primary features: X_class, which encompasses 200 individual data samples, each exhibiting values within the range of 0 to 1; and the target variable, denoted as y_class, which assumes binary values (specifically 0 or 1) predicated upon the condition wherein the aggregate sum of the aforementioned two feature values surpasses 1.

$$x_1 + x_2 > 1 \text{are labeled as 1, otherwise 0.} \tag{2}$$

The clustering dataset is just a random 2D spread of points with two features (200 samples) with values between 0 and 10.

It is advised to verify that the user-defined dataset encompasses a practical range of parameters and meets the criteria outlined by Van Smeden et al. (2018) and Riley et al. (2019) (a minimum of 10 and 23 observations per feature, respectively), and Frank and Todeschini (1994) (a minimum ratio of 3 and 5 between the number of observations and features).

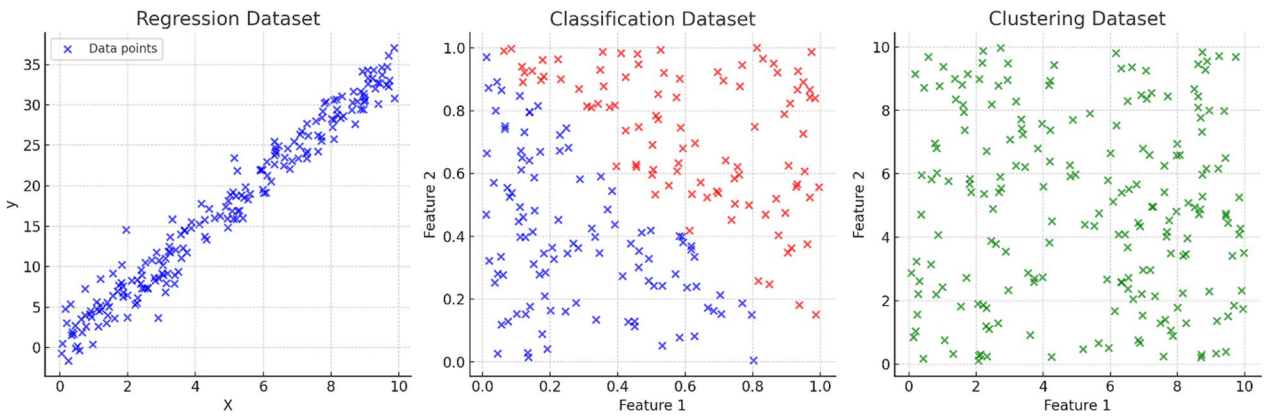

Fig. 1 Plots of datasets used in regression, classification, and clustering

```
# Import necessary libraries
import numpy as np                                  # For numerical computations and random data generation
import matplotlib.pyplot as plt                     # For creating plots and visualizations

# Set a random seed for reproducibility (ensures the same random numbers are generated each time)
np.random.seed(42)

# ---- Dataset generation ----
# Regression dataset: Generate 200 random data points for X between 0 and 10
X_reg = np.random.rand(200, 1) * 10
# Generate corresponding y values with a linear relationship (y = 3.5 * X + noise)
y_reg = 3.5 * X_reg.squeeze() + np.random.randn(200) * 2

# Classification dataset: Generate 200 random points in 2D space (features)
X_class = np.random.rand(200, 2)
# Create binary labels (0 or 1) based on whether the sum of features > 1
y_class = (X_class[:, 0] + X_class[:, 1] > 1).astype(int)

# Clustering dataset: Generate 200 random points in 2D space scaled by 10
X_cluster = np.random.rand(200, 2) * 10

# ---- Plotting the datasets ----
# Create a figure for the plots with a specified size (15 inches wide, 5 inches tall)
plt.figure(figsize=(15, 5))

# Plot the regression dataset
plt.subplot(1, 3, 1)                                # Define the first subplot (1 row, 3 columns, first plot)
plt.scatter(X_reg, y_reg, alpha=0.7, color='blue', label='Data points')  # Scatter plot of X and y
plt.title('Regression Dataset')                     # Title for the regression plot
plt.xlabel('X')                                     # Label for the x-axis
plt.ylabel('y')                                     # Label for the y-axis
plt.legend()                                        # Display the legend

# Plot the classification dataset
plt.subplot(1, 3, 2)                                # Define the second subplot (1 row, 3 columns, second plot)
# Scatter plot with points colored based on their class (0 or 1), using a blue-red colormap
plt.scatter(X_class[:, 0], X_class[:, 1], c=y_class, cmap='bwr', alpha=0.7)
plt.title('Classification Dataset')                 # Title for the classification plot
plt.xlabel('Feature 1')                             # Label for the x-axis
plt.ylabel('Feature 2')                             # Label for the y-axis

# Plot the clustering dataset
plt.subplot(1, 3, 3)                                # Define the third subplot (1 row, 3 columns, third plot)
# Scatter plot of the clustering data points in green
plt.scatter(X_cluster[:, 0], X_cluster[:, 1], alpha=0.7, color='green')
plt.title('Clustering Dataset')                     # Title for the clustering plot
plt.xlabel('Feature 1')                             # Label for the x-axis
plt.ylabel('Feature 2')                             # Label for the y-axis

# Adjust the layout of the plots to avoid overlapping elements
plt.tight_layout()

# Display the plots on the screen
plt.show()
```

In addition, engineers may possess their own datasets, and as such, it is possible to augment the aforementioned script to facilitate the incorporation of such datasets (whether in xslx or csv format):

```
#Imports
import pandas as pd
#Script to enable user-based datasets
# Load datasets from Excel or CSV files in the same folder
regression_file = 'regression_dataset.xlsx'                    # Replace with the actual file name
classification_file = 'classification_dataset.xlsx'            # Replace with the actual file name
clustering_file = 'clustering_dataset.csv'                     # Replace with the actual file name
# Function to dynamically split features and target
def load_data(file_path):
    data = pd.read_excel(file_path) if file_path.endswith('.xlsx') else pd.read_csv(file_path)
    X = data.iloc[:, :-1].values                               # All columns except the last are input
    y = data.iloc[:, -1].values                                # Last column is the target
    return X, y
# Regression Dataset
X_reg, y_reg = load_data(regression_file)
# Classification Dataset
X_class, y_class = load_data(classification_file)
# Clustering Dataset
X_cluster, _ = load_data(clustering_file)                      # No target for clustering
```

2.3 Tutorial

For engineers new to ML, navigating this complex field can be challenging. The following versatile Python script is a versatile tool that provides a comprehensive technical exploration, specifically designed to carry out three fundamental machine learning tasks: regression, classification, and clustering. Through a detailed analysis of each component (referred to as blocks) of the script, this tutorial aims to empower engineers with the necessary knowledge to effectively use and customize the script for a wide range of engineering applications.

The Python script is engineered to handle multiple ML tasks, such as regression, classification, and clustering, with ease. Furthermore, the script offers flexibility, allowing users to seamlessly switch between tasks, employ synthetic or user-provided datasets, and comprehensively visualize the results. Additionally, the script encompasses mechanisms for model evaluation and interpretability, ensuring that engineers are able to make sense of the model's outputs. Its modular, block-like design positions it as a user-friendly starting point for engineers seeking to integrate machine learning into their workflows, even with limited prior experience in the field.

2.3.1 Importing essential libraries

At the outset, the script imports several critical libraries essential for data manipulation, numerical computations, visualization, and ML operations:

```
# Import necessary libraries for numerical operations, data visualization, and ML tasks
import numpy as np                                             # NumPy for numerical computations
import matplotlib.pyplot as plt                                # Matplotlib for plotting graphs
import pandas as pd                                            # Pandas for data manipulation and analysis
import os                                                      # OS module for interacting with the operating system
import shap                                                    # SHAP for model explainability
from sklearn.model_selection import train_test_split, cross_val_predict   # Functions for splitting data and cross-validation
from sklearn.metrics import (
    mean_absolute_error, r2_score,                             # Metrics for regression tasks
    accuracy_score, f1_score,                                  # Metrics for classification tasks
    silhouette_score, davies_bouldin_score                     # Metrics for clustering tasks
)
from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier   # ML models for regression and classification
from sklearn.cluster import KMeans                             # ML model for clustering
from sklearn.decomposition import PCA                          # Principal Component Analysis for dimen. reduction
from sklearn.metrics import ConfusionMatrixDisplay             # Function to display confusion matrices
from matplotlib.gridspec import GridSpec                       # Grid specification for complex layouts in Matplotlib
```

Table 1 Criteria for ML tasks

Criterion	Regression	Classification	Clustering
Type of Problem	Predicting a continuous numerical value	Assigning instances to predefined categories	Grouping similar instances without predefined labels
Supervised/unsupervised	Supervised	Supervised	Unsupervised
Target variable	Continuous (e.g., temperature)	Categorical (e.g., failure vs. not no failure)	None (no target variable)
Data requirements	Labeled data with continuous targets	Labeled data with categorical targets	No labels required
Goal	Estimate the relationship between inputs to predict the target	Classify input data into distinct categories based on training data	Discover inherent groupings or structures within the data
Common algorithms	- Random Forest Regressor - Support Vector Regression - XGBoost	- Logistic Regression - Random Forest Classifier - Neural Networks	- K-Means Clustering - Hierarchical Clustering - DBSCAN - Gaussian Mixture Models
Evaluation metrics	- R^2 (Coefficient of Determination) - MAE (Mean Absolute Error) - MSE (Mean Squared Error)	- Accuracy - Precision - Recall - F1-Score - ROC-AUC	- Silhouette Score - Davies-Bouldin Index
Data structure	Input features can be numerical or categorical, but target is numerical	Both input features and target are categorical or a mix of numerical and categorical	Input features are typically numerical but can include categorical through encoding
Scalability	Scales well with large datasets, especially with ensemble methods like Random Forest	Scales well with large datasets, though computational complexity can increase with model complexity	Scalability varies; algorithms like K-Means are more scalable compared to hierarchical clustering
Handling of outliers	Sensitive to outliers; outliers can significantly affect model performance	Can be sensitive depending on the algorithm (e.g., Logistic Regression vs. Tree-based methods)	Varies; algorithms like DBSCAN are more robust to outliers compared to K-Means
Feature relationships	Assumes a relationship between features and the target variable, which can be linear or non-linear	Assumes that input features contribute to distinguishing between classes, potentially with complex relationships	Does not assume any predefined relationship; focuses on similarity between data points
Dimensionality considerations	Can handle high-dimensional data but may require regularization to prevent overfitting	Can handle high-dimensional data; feature selection or dimensionality reduction may improve performance	Often benefits from dimensionality reduction to improve clustering quality and computational efficiency
Example scenario	An engineer wants to predict the compressive strength of a material based on its composition and processing parameters	An engineer aims to classify defects in a manufacturing process as either "critical" or "non-critical" based on sensor data	An engineer seeks to group maintenance records to identify common failure patterns without predefined categories

It is worth noting that the following software packages are included:

- *NumPy (numpy)*: NumPy facilitates efficient numerical computations and handling of large multi-dimensional arrays. It supports matrices and arrays, high-level mathematical functions, vectorized operations for efficient element-wise computations, and broadcasting rules for operations on arrays of different shapes. NumPy also includes universal functions (ufuncs) for fast, memory-efficient computation, as well as tools for linear algebra, Fourier transforms, and random number generation.
- *Matplotlib (matplotlib.pyplot)*: Matplotlib enables the creation of static, interactive, and animated visualizations. It offers an intuitive interface with extensive customization options, allowing users to manipulate figures, axes, and plot elements to create various plots. The library supports comprehensive styling, annotations, and color mapping, providing detailed control over aesthetics and layout.
- *Pandas (pandas)*: Pandas provides data structures and functions for data manipulation and analysis. It introduces robust data structures like Series and DataFrame designed for handling heterogeneous and structured data. Pandas facilitates efficient data ingestion, cleaning, transformation, and aggregation, with features like intuitive indexing and label-based slicing for simplifying merging, joining, and reshaping datasets. Its optimized performance, often leveraging NumPy, enables rapid analysis of large datasets.
- *OS (os)*: The OS module allows interaction with the operating system, including file handling. It provides a portable way to use operating system-dependent functionalities for tasks like interacting with the file system, managing directories, and performing file operations such as reading, writing, and deleting. The module also facilitates environment variable access, process management, and platform-independent path manipulations to ensure uniform script functionality across operating systems, allowing for flexible applications managing system resources efficiently.
- *SHAP (shap)*: SHAP offers tools for interpreting and explaining machine learning model predictions. It interprets ML models by computing feature contributions using game theory, quantifying the impact of each feature on predictions by assigning Shapley values. This approach ensures a fair distribution of contribution across features and supports local explanations for individual predictions and global interpretations for understanding overall model behavior. SHAP's model-agnostic nature and integration with various algorithms enable transparent debugging and validation of ML outputs.
- *Scikit-learn (sklearn)*: Scikit-learn is a comprehensive machine learning library providing tools for data splitting, model training, evaluation, and more. It includes implementations of classification, regression, clustering, and dimensionality reduction techniques, along with features for pipeline construction, cross-validation, and performance metrics computation to ensure robust model selection and validation. The library prioritizes ease-of-use and interoperability with other Python libraries like NumPy and pandas, making it accessible for rapid prototyping.
- *Matplotlib GridSpec (matplotlib.gridspec)*: Matplotlib GridSpec assists in creating complex subplot layouts by subdividing a figure into a grid. This feature enables users to allocate subplots across defined rows and columns with customizable spacing and alignment. The modular approach allows for non-uniform, overlapping, or nested grid arrangements, facilitating sophisticated visualizations with meticulous layout configurations.

Understanding these libraries is fundamental as they form the backbone of the script's functionality.

2.3.2 Setting a random seed

```
np.random.seed(42)                                    # Seed for randomness
```

Setting a random seed ensures that the results are reproducible across multiple runs. This is crucial for debugging and verifying model performance consistently.

2.3.3 Selecting the ML task

The variable TASK determines the specific machine learning task that will be executed by the script. By adjusting its value, engineers have the flexibility to transition between regression, classification, and clustering analyses without the need to modify the foundational code. Table 1 presents an overview of the various task types commonly associated with engineering problems.

```
# Choose the ML task you want to perform: 'regression', 'classification', or 'clustering'
TASK = 'regression'                                   # You can change this to 'classification' or 'clustering' as needed
```

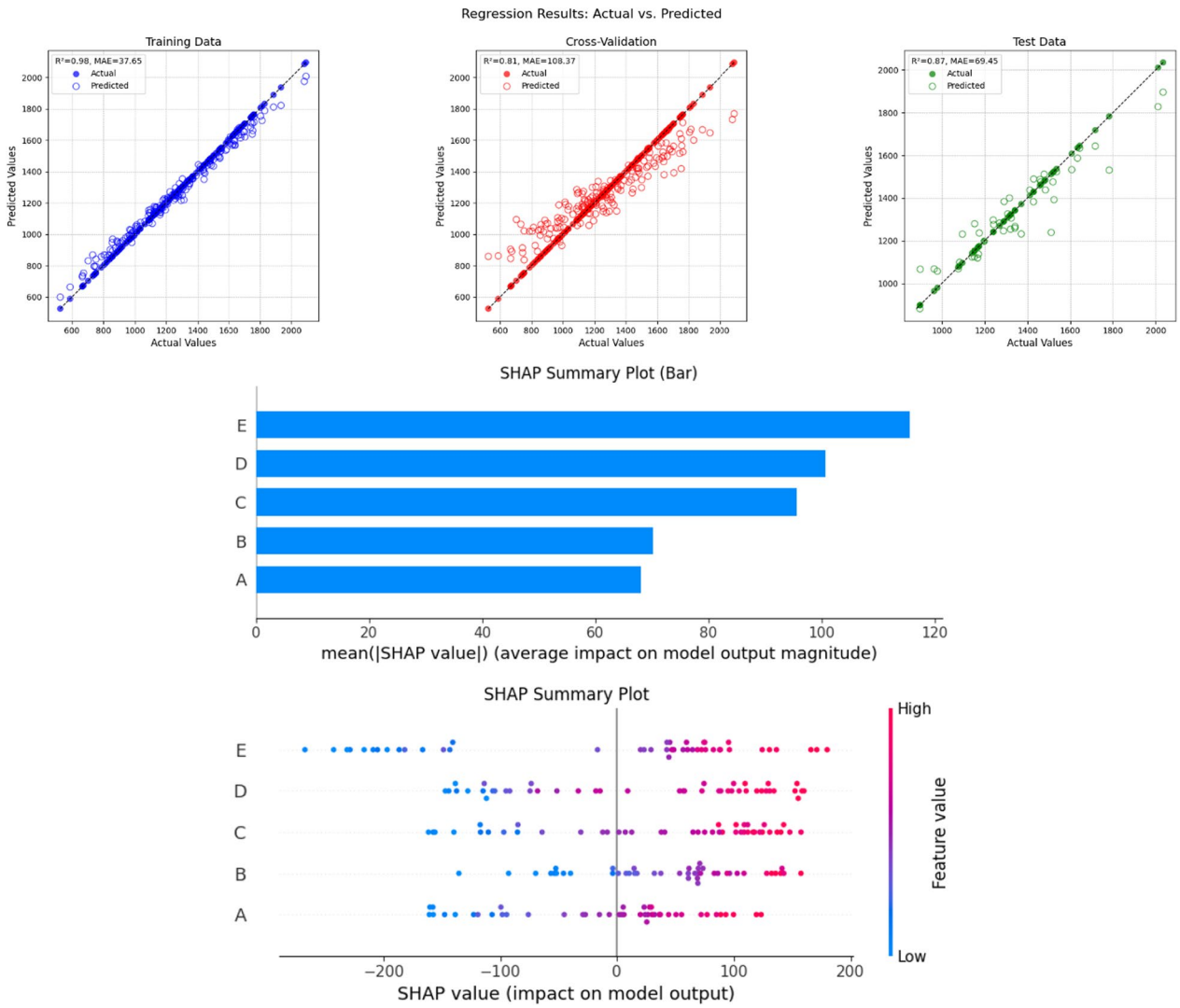

Fig. 2 Results from the regression example

2.3.4 Data source toggle

This toggle allows users to select between utilizing synthetic (generated) data or their own datasets. Synthetic data can be advantageous for educational purposes and testing, while user-generated data is indispensable for practical applications in real-world scenarios.

```
# Decide whether to use your own dataset or use synthetic (generated) data
USE_USER_DATA = False                                    # Set to True if you want to load your own data files
```

2.3.5 Defining synthetic datasets

The script defines synthetic datasets tailored to each ML task (as seen in Sec. 2.2). These datasets serve to facilitate the evaluation and comprehension of the manuscript's algorithms without dependence on external data repositories.

```
# Synthetic data for Regression
X_reg_synthetic = np.random.rand(200, 1) * 10                # 200 samples, 1 feature scaled by 10
y_reg_synthetic = 3.5 * X_reg_synthetic.squeeze() + np.random.randn(200) * 2    # Linear relationship with noise

# Synthetic data for Classification
X_class_synthetic = np.random.rand(200, 2)                   # 200 samples, 2 features
y_class_synthetic = (X_class_synthetic[:, 0] + X_class_synthetic[:, 1] > 1).astype(int)    # Binary target based on feature sum

# Synthetic data for Clustering
X_cluster_synthetic = np.random.rand(200, 2) * 10            # 200 samples, 2 features scaled by 10
```

2.3.6 Loading user-provided data

This function is tasked with the importation of data from Excel (.xlsx) or CSV (.csv) file formats. It undertakes the processing of the data in accordance with the task selected by the user. Specifically, for tasks such as regression and classification, the dataset is anticipated to contain a minimum of two columns: one for features and the other for the target variable. In these cases, the final column is designated as the target variable. Conversely, the clustering task option incorporates all columns as features, with no column designated as a target variable.

```python
def load_data(file_path, task):
    """
    Load data from an Excel or CSV file.
    Parameters:
        file_path (str): Path to the data file.
        task (str): The task type ('regression', 'classification', 'clustering').
    Returns:
        X (np.ndarray): Features.
        y (np.ndarray or None): Target. None for clustering.
    """
    # Check if the file exists in the specified path
    if not os.path.exists(file_path):
        raise FileNotFoundError(f"File '{file_path}' does not exist in the current directory.")

    # Load data based on file extension
    if file_path.endswith('.xlsx'):
        data = pd.read_excel(file_path)                                    # Read Excel file
    elif file_path.endswith('.csv'):
        data = pd.read_csv(file_path)                                      # Read CSV file
    else:
        raise ValueError("Unsupported file format. Please provide an Excel (.xlsx) or CSV (.csv) file.")

    # Process data based on the selected task
    if task in ['regression', 'classification']:
        if data.shape[1] < 2:
            raise ValueError("Data must contain at least one feature column and one target column.")
        X = data.iloc[:, :-1].values                                       # All columns except the last are input features
        y = data.iloc[:, -1].values                                        # Last column is the target variable
    elif task == 'clustering':
        X = data.values                                                    # All columns are features for clustering
        y = None                                                           # No target variable for clustering
    else:
        raise ValueError("Unsupported task. Choose from 'regression', 'classification', or 'clustering'.")

    return X, y                                                            # Return features and target
```

2.3.7 Utilizing user data or synthetic data

Based on the USE_USER_DATA flag, the script either loads user-provided data or utilizes predefined synthetic datasets. For supervised tasks (regression and classification), the data is split into training and testing sets using an 80–20 split via the train_test_split function. In contrast, clustering tasks, which are inherently unsupervised, utilize the complete dataset, as these methods do not necessitate the presence of target labels for their execution.

```
if USE_USER_DATA:
    # Define file paths based on the selected task
    if TASK == 'regression':
        data_file = 'regression_dataset.xlsx'                    # Replace with your actual regression data file
    elif TASK == 'classification':
        data_file = 'classification_dataset.xlsx'                # Replace with your actual classification data file
    elif TASK == 'clustering':
        data_file = 'clustering_dataset.csv'                     # Replace with your actual clustering data file
    else:
        raise ValueError("Unsupported task. Choose from 'regression', 'classification', or 'clustering'.")

    try:
        X, y = load_data(data_file, TASK)                        # Load your own data
        print(f"Loaded {TASK} data from '{data_file}'.")         # Confirm successful loading
    except Exception as e:
        print(f"Error loading {TASK} data: {e}")                 # Print error if loading fails
        exit(1)                                                  # Exit the program if data loading fails
else:
    # Use synthetic data based on the selected task
    if TASK == 'regression':
        X = X_reg_synthetic
        y = y_reg_synthetic
        print("Using synthetic regression data.")                # To inform the user
    elif TASK == 'classification':
        X = X_class_synthetic
        y = y_class_synthetic
        print("Using synthetic classification data.")            # To inform the user
    elif TASK == 'clustering':
        X = X_cluster_synthetic
        y = None
        print("Using synthetic clustering data.")                # To inform the user
    else:
        raise ValueError("Unsupported task. Choose from 'regression', 'classification', or 'clustering'.")
```

2.3.8 Split data into training and test sets (supervised tasks)

Subsequently, the dataset is split into training and test sets in a 80%/20%. These values can be updated by the user. A common split ratio can vary between 70%/30% to 80%/20%.

```
if TASK in ['regression', 'classification']:
    # For regression and classification, split data into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42                     # 80% training, 20% testing
    )
else:
    # For clustering, use all data for training (no target variable)
    X_train = X
    X_test = X
    y_train = None
    y_test = None
```

2.3.9 Selecting and initializing the model

Depending on the selected TASK, the script initializes the appropriate ML model. For example,

- *Regression*: Utilizes RandomForestRegressor, an ensemble learning technique that constructs a multitude of decision trees and amalgamates their predictions to enhance accuracy and robustness.
- *Classification*: Employs RandomForestClassifier, similar to its regressor counterpart but tailored for categorical target variables.
- *Clustering*: Implements KMeans, a partitioning method that divides data into k distinct clusters based on the similarity of their features.

```
if not COMPARE_ALL:
    # Single-model workflow
    if TASK in ['regression', 'classification']:
        y_cv_pred = cross_val_predict(model, X_train, y_train, cv=5)
        model.fit(X_train, y_train)
        y_train_pred = model.predict(X_train)
        y_test_pred = model.predict(X_test)
    else:  # clustering
        model.fit(X_train)
        # For KMeans/Birch/GaussianMixture .predict(...) works
        cluster_labels = model.predict(X_test)

else:
    # Multi-model workflow
    all_models = get_all_models(TASK)
    for name, this_model in all_models.items():
        print(f"Training model: {name} ...")
        if TASK in ['regression', 'classification']:
            y_cv_pred = cross_val_predict(this_model, X_train, y_train, cv=5)
            this_model.fit(X_train, y_train)
            y_train_pred = this_model.predict(X_train)
            y_test_pred = this_model.predict(X_test)

            if TASK == 'regression':
                data_metrics = {
                    "Model": name,
                    "Train R2": r2_score(y_train, y_train_pred),
                    "Train MAE": mean_absolute_error(y_train, y_train_pred),
                    "CV R2": r2_score(y_train, y_cv_pred),
                    "CV MAE": mean_absolute_error(y_train, y_cv_pred),
                    "Test R2": r2_score(y_test, y_test_pred),
                    "Test MAE": mean_absolute_error(y_test, y_test_pred),
                    "_preds_": {
                        "train_pred": y_train_pred,
                        "test_pred": y_test_pred,
                        "cv_pred": y_cv_pred
                    }
                }
            else:
                data_metrics = {
                    "Model": name,
                    "Train Accuracy": accuracy_score(y_train, this_model.predict(X_train)),
                    "Train F1": f1_score(y_train, this_model.predict(X_train), average='weighted'),
                    "CV Accuracy": accuracy_score(y_train, y_cv_pred),
                    "CV F1": f1_score(y_train, y_cv_pred, average='weighted'),
                    "Test Accuracy": accuracy_score(y_test, this_model.predict(X_test)),
                    "Test F1": f1_score(y_test, this_model.predict(X_test), average='weighted'),
                    "_preds_": {
                        "train_pred": y_train_pred,
                        "test_pred": y_test_pred,
                        "cv_pred": y_cv_pred
                    }
                }
            multi_model_metrics.append(data_metrics)

        else:
            # For kmeans, birch, gaussian_mixture
            this_model.fit(X_train)
            cluster_labels = this_model.predict(X_test)
            data_metrics = {
                "Model": name,
                "Silhouette": silhouette_score(X_test, cluster_labels),
                "Davies-Bouldin": davies_bouldin_score(X_test, cluster_labels),
                "_labels_": cluster_labels
            }
            multi_model_metrics.append(data_metrics)
```

2.3.10 Training and testing the model

For supervised tasks (regression and classification), the script performs cross-validation using cross_val_predict with 5-folds to obtain reliable performance estimates. The model is then fitted on the entire training dataset, and p predictions are generated for both the training and testing datasets. For clustering, the KMeans model is fitted to the entire dataset, with cluster labels being assigned based on the model's output. The script calculates performance metrics specific to each ML task:

- Regression:
  - R^2 *Score*: This metric quantifies the proportion of the variance in the dependent variable that can be predicted from the independent variables. For example, a score of 0.85 signifies that 85% of the variance in the target variable is explained by the model. The best possible value for this metric is 1.0, indicative of perfect prediction.
  - *Mean Absolute Error (MAE)*: This represents the average magnitude of errors between predicted and actual values. For example, an MAE of 2.3 indicates that the predictions deviate from the actual values by an average of 2.3 units. The best possible value for this metric is 0.0, which indicates no prediction error.
- Classification:
  - *Accuracy*: This metric measures the proportion of instances that are correctly classified. For instance, an accuracy of 0.92 means the model correctly classifies 92% of all cases. The best possible value for the accuracy metric is 1.0 or 100%, indicating perfect classification.
  - *F1-Score*: This is the harmonic mean of precision and recall, providing a balance between the two. For example, an F1-score of 0.88 indicates strong overall performance on both precision and recall (with the best possible being 1.0).
- Clustering:
  - *Silhouette Score*: This metric assesses how similar an object is to its own cluster compared to other clusters. Here, a score of 0.65 indicates good cluster separation and cohesion, and a value of 1.0 indicates perfect cluster definition.
  - *Davies-Bouldin Index*: This evaluates the average similarity ratio of each cluster to its most similar counterpart, with lower values indicating better clustering. An index of 0.8 suggests well-defined and separated clusters, with a best possible value of zero indicating perfect cluster separation.

2.3.11 Visualizing results

```
if TASK == 'regression':
    # Calculate regression metrics
    metrics = {
        "Metric": ["R²", "MAE"],                                    # R-squared and Mean Absolute Error
        "Training": [
            r2_score(y_train, y_train_pred),                        # R² on training data
            mean_absolute_error(y_train, y_train_pred)              # MAE on training data
        ],
        "Cross-Validation": [
            r2_score(y_train, y_cv_pred),                           # R² from cross-validation
            mean_absolute_error(y_train, y_cv_pred)                 # MAE from cross-validation
        ],
        "Test": [
            r2_score(y_test, y_test_pred),                          # R² on test data
            mean_absolute_error(y_test, y_test_pred)                # MAE on test data
        ],
    }
    metrics_df = pd.DataFrame(metrics)                              # Create a DataFrame for easy viewing
elif TASK == 'classification':
    # Calculate classification metrics
    metrics = {
        "Metric": ["Accuracy", "F1-Score"],                         # Accuracy and F1-Score
        "Training": [
            accuracy_score(y_train, model.predict(X_train)),         # Accuracy on training data
            f1_score(y_train, model.predict(X_train), average='weighted')   # F1-Score on training data
        ],
        "Cross-Validation": [
            accuracy_score(y_train, y_cv_pred),                     # Accuracy from cross-validation
            f1_score(y_train, y_cv_pred, average='weighted')        # F1-Score from cross-validation
        ],
        "Test": [
            accuracy_score(y_test, model.predict(X_test)),          # Accuracy on test data
            f1_score(y_test, model.predict(X_test), average='weighted')     # F1-Score on test data
        ],
    }
    metrics_df = pd.DataFrame(metrics)                              # Create a DataFrame for easy viewing
elif TASK == 'clustering':
    # Calculate clustering metrics
    metrics = {
        "Metric": ["Silhouette Score", "Davies-Bouldin Index"],     # Silhouette and Davies-Bouldin scores
        "Score": [
            silhouette_score(X_test, cluster_labels),               # Silhouette Score
            davies_bouldin_score(X_test, cluster_labels)            # Davies-Bouldin Index
        ],
    }
    metrics_df = pd.DataFrame(metrics)                              # Create a DataFrame for easy viewing
else:
    raise ValueError("Unsupported task. Choose from 'regression', 'classification', or 'clustering'.")
```

The results become accessible upon successful training and validation of the model. The script auto-generates scatter plots to visualize regression tasks, illustrating the comparison between actual and predicted values across training, cross-validation, and testing datasets. An ideal diagonal prediction line ($y = x$) is also included to provide a reference for perfect predictions. Two regression metrics (R^2 and MAE) are annotated in the legends for immediate insight into model performance. Then, for classification tasks, the script generates confusion matrices for the training, cross-validation, and testing datasets utilizing ConfusionMatrixDisplay. These matrices provide a detailed breakdown of true versus predicted classifications and highlight areas where the model performs well or requires improvement. Accuracy and F1-Score metrics are annotated directly on the plots to facilitate quick assessment. Finally, for clustering tasks, the script employs Principal Component Analysis (PCA) to reduce t the dimensionality of the data to two components that can be plotted in a 2D space.

```
# Style the metrics table for better readability
styled_metrics = metrics_df.style.set_properties(**{'text-align': 'center'})\
                          .set_table_styles([
                              {'selector': 'th', 'props': [('text-align', 'center')]},  # Center align headers
                              {'selector': 'td', 'props': [('text-align', 'center')]}   # Center align cells
                          ])\
                          .set_caption("Model Performance Metrics")                     # Add a caption to the table

# Display the Styled Table
# Note: If running outside a Jupyter environment, the table might not display as intended
try:
    from IPython.display import display                                                 # Import display function for Jupyter
    display(styled_metrics)                                                             # Display the styled table
except ImportError:
    print(metrics_df)                                                                   # Print the DataFrame as a fallback

# Visualization based on the selected task
if TASK in ['regression', 'classification']:
    # Function to plot results for supervised tasks (regression and classification)
    def plot_results_supervised(ax, y_actual, y_pred, title, color, metric1, metric2):
        """
        Plot Actual vs Predicted values or Confusion Matrix.

        Parameters:
            ax (matplotlib.axes.Axes): The axes to plot on.
            y_actual (np.ndarray): Actual target values.
            y_pred (np.ndarray): Predicted target values.
            title (str): Title of the subplot.
            color (str): Color for the plots.
            metric1 (float): First metric (e.g., R² or Accuracy).
            metric2 (float): Second metric (e.g., MAE or F1-Score).
        """
        if TASK == 'regression':
            # Scatter plot for Actual vs Predicted values
            ax.scatter(y_actual, y_actual, color=color, label='Actual', marker='o', s=60, alpha=0.7, edgecolors='none')
            ax.scatter(y_actual, y_pred, facecolors='none', edgecolors=color, label='Predicted', marker='o', s=60, alpha=0.7)
            # Add a dashed line representing the ideal prediction
            min_val = min(np.min(y_actual), np.min(y_pred))
            max_val = max(np.max(y_actual), np.max(y_pred))
            ax.plot([min_val, max_val], [min_val, max_val], 'k--', linewidth=1)
            # Set titles and labels
            ax.set_title(title, fontsize=14)
            ax.set_xlabel('Actual Values', fontsize=12)
            ax.set_ylabel('Predicted Values', fontsize=12)
            ax.set_aspect('equal', adjustable='box')                                    # Ensure equal scaling
            # Add a legend with metrics
            legend = ax.legend(title=f'R²={metric1:.2f}, MAE={metric2:.2f}')
        elif TASK == 'classification':
            # Display a confusion matrix
            ConfusionMatrixDisplay.from_predictions(y_actual, y_pred, ax=ax, cmap='Blues', colorbar=False)
            ax.set_title(title, fontsize=14)
            # Add metrics as a text box
            ax.text(0.95, 0.05, f'Accuracy={metric1:.2f}\nF1-Score={metric2:.2f}',
                    verticalalignment='bottom', horizontalalignment='right',
                    transform=ax.transAxes,
                    color='black', fontsize=12,
                    bbox=dict(facecolor='white', alpha=0.5, boxstyle='round,pad=0.5'))

        # Add grid lines for better readability
        ax.grid(True, linestyle='--', linewidth=0.5)

    if TASK == 'regression':
        # Create a figure with multiple subplots for regression results
        fig = plt.figure(constrained_layout=True, figsize=(20, 6))                      # Define figure size
        gs = GridSpec(1, 3, figure=fig, wspace=0.3)                                     # Define a grid layout with 1 row and 3 columns

        # Define individual subplots
        ax1 = fig.add_subplot(gs[0, 0])                                                 # Training data subplot
        ax2 = fig.add_subplot(gs[0, 1])                                                 # Cross-validation subplot
        ax3 = fig.add_subplot(gs[0, 2])                                                 # Test data subplot

        # Extract metrics for each dataset split
        train_r2 = metrics_df.loc[metrics_df['Metric'] == 'R²', 'Training'].values[0]
        train_mae = metrics_df.loc[metrics_df['Metric'] == 'MAE', 'Training'].values[0]

        cv_r2 = metrics_df.loc[metrics_df['Metric'] == 'R²', 'Cross-Validation'].values[0]
        cv_mae = metrics_df.loc[metrics_df['Metric'] == 'MAE', 'Cross-Validation'].values[0]

        test_r2 = metrics_df.loc[metrics_df['Metric'] == 'R²', 'Test'].values[0]
        test_mae = metrics_df.loc[metrics_df['Metric'] == 'MAE', 'Test'].values[0]

        # Plot the results on each subplot
        plot_results_supervised(ax1, y_train, y_train_pred, 'Training Data', color='blue', metric1=train_r2, metric2=train_mae)
        plot_results_supervised(ax2, y_train, y_cv_pred, 'Cross-Validation', color='red', metric1=cv_r2, metric2=cv_mae)
        plot_results_supervised(ax3, y_test, y_test_pred, 'Test Data', color='green', metric1=test_r2, metric2=test_mae)

        # Add an overall title to the figure
        fig.suptitle('Regression Results: Actual vs. Predicted', fontsize=16)

        # Display the plots
        plt.show()

    elif TASK == 'classification':
        # Create a figure with multiple subplots for classification results
        fig = plt.figure(constrained_layout=True, figsize=(20, 6))                      # Define figure size
        gs = GridSpec(1, 3, figure=fig, wspace=0.3)                                     # Define a grid layout with 1 row and 3 columns

        # Define individual subplots
        ax1 = fig.add_subplot(gs[0, 0])                                                 # Training data subplot
        ax2 = fig.add_subplot(gs[0, 1])                                                 # Cross-validation subplot
        ax3 = fig.add_subplot(gs[0, 2])                                                 # Test data subplot

        # Extract metrics for each dataset split
        train_acc = metrics_df.loc[metrics_df['Metric'] == 'Accuracy', 'Training'].values[0]
        train_f1 = metrics_df.loc[metrics_df['Metric'] == 'F1-Score', 'Training'].values[0]

        cv_acc = metrics_df.loc[metrics_df['Metric'] == 'Accuracy', 'Cross-Validation'].values[0]
        cv_f1 = metrics_df.loc[metrics_df['Metric'] == 'F1-Score', 'Cross-Validation'].values[0]

        test_acc = metrics_df.loc[metrics_df['Metric'] == 'Accuracy', 'Test'].values[0]
        test_f1 = metrics_df.loc[metrics_df['Metric'] == 'F1-Score', 'Test'].values[0]

        # Plot the results on each subplot
        plot_results_supervised(ax1, y_train, model.predict(X_train), 'Training Data', color='blue', metric1=train_acc, metric2=train_f1)
        plot_results_supervised(ax2, y_train, y_cv_pred, 'Cross-Validation', color='red', metric1=cv_acc, metric2=cv_f1)
        plot_results_supervised(ax3, y_test, y_test_pred, 'Test Data', color='green', metric1=test_acc, metric2=test_f1)

        # Add an overall title to the figure
        fig.suptitle('Classification Results: Confusion Matrices', fontsize=16)

        # Display the plots
        plt.show()

elif TASK == 'clustering':
    # For clustering, visualize the clusters using PCA for dimensionality reduction
    pca = PCA(n_components=2)                                                           # Initialize PCA to reduce data to 2 dimensions
    X_pca = pca.fit_transform(X_test)                                                   # Fit PCA on the data and transform it

    # Create a figure for clustering results
    fig = plt.figure(constrained_layout=True, figsize=(10, 6))                          # Define figure size
    gs = GridSpec(1, 1, figure=fig)                                                     # Define a grid layout with 1 row and 1 column

    ax = fig.add_subplot(gs[0, 0])                                                      # Define the subplot
    # Scatter plot of the PCA-reduced data colored by cluster labels
    scatter = ax.scatter(X_pca[:, 0], X_pca[:, 1], c=cluster_labels, cmap='viridis', alpha=0.7, edgecolors='k', s=60)
    ax.set_title('Clustering Results (PCA-Reduced Data)', fontsize=16)                  # Set title
    ax.set_xlabel('PCA Component 1', fontsize=12)                                       # X-axis label
    ax.set_ylabel('PCA Component 2', fontsize=12)                                       # Y-axis label
    legend1 = ax.legend(*scatter.legend_elements(), title="Clusters")                   # Create a legend for clusters
    ax.add_artist(legend1)                                                              # Add the legend to the plot
    ax.grid(True, linestyle='--', linewidth=0.5)                                        # Add grid lines

    plt.show()                                                                          # Display the clustering plot
```

2.3.12 Model explainability with SHAP

The results from the model can also be explained via SHAP (SHapley Additive exPlanations) (Lundberg & Lee, 2017), which offers valuable insights into the significance of different features and the decision-making process of the model. SHAP values are calculated in the context of supervised tasks to elucidate the specific contribution of each feature to the model's predictions. These values quantify the impact of each feature on individual predictions, enabling a granular understanding of model behavior. The script generates two primary SHAP visualizations (1) Summary bar to display the mean absolute SHAP value for each feature, indicating overall feature importance, and (2) Summary dot to illustrate the distribution of SHAP values to demonstrate how feature values influence the predictions.

As an example, results in Fig. 2 show a comparable performance during training, cross validation, and tests data for the regression model. This implies the development of a balanced model. In practical model development scenarios, it is advisable to incorporate additional independent performance metrics (3–7) (Xie et al., 2021). The same figure highlights the significance of each feature in facilitating accurate model predictions. For example, the summary bar plot shows that Feature E holds the highest importance as the model heavily relies on this particular feature for accurate target prediction. The subsequent plot illustrates that each dot on the graph corresponds to an individual data point, where the horizontal axis represents the SHAP values quantifying the contribution of the corresponding feature to the prediction. Positive SHAP values contribute to an increase in the prediction, while negative values lead to a decrease.

The SHAP summary plot utilizes a color gradient ranging from blue (indicating low values) to red (indicating high values) to encode feature values, thereby elucidating the relationship between features and predictions. The positioning of these points along the SHAP value axis delineates both the magnitude and the direction of the influence exerted by each feature. Upon scrutinizing the clustering patterns, a consistent rightward displacement of red points signifies a positive correlation, implying that an increase in feature values is associated with higher model predictions. Conversely, a leftward shift suggests an inverse relationship. For example, feature E demonstrates a pronounced bimodal distribution of impact, with lower values (blue) congregating towards negative SHAP values and higher values (red) towards positive ones. This bimodality suggests a non-linear relationship that the model has successfully captured, and such non-monotonic behavior may point to threshold effects or interaction terms that merit further exploration.

Additionally, the vertical dispersion of points for each feature offers insights into prediction uncertainty and potential feature interactions; a wider spread often signifies contextual dependencies, where the same feature value may exert varying impacts contingent upon other variables.

From the standpoint of model enhancement, these visualizations can guide the features of engineering efforts. For instance, features that exhibit distinct separation between red and blue clusters but display substantial vertical dispersion may benefit from categorization or transformation techniques to effectively capture their nonlinear effects. On the other hand, features demonstrating notable overlap between red and blue data points could highlight opportunities for introducing interaction terms or developing more sophisticated encoding schemes. Additionally, analyzing the density of points along the SHAP value axis can offer insights for feature selection, with sparse or evenly distributed data points potentially indicating lower predictive capability. It is important to note that alternative explainability tools, such as LIME, can also be utilized for this purpose (Ribeiro et al., 2016).

2.3.13 Reporting results

In an engineering context, the thorough reporting and documentation of ML studies are essential to ensure reproducibility, facilitate peer review, and promote the efficient transfer of knowledge. To accomplish these objectives, engineers can adhere to the following guidelines:

- *Clearly define the study's scope and objectives*: Articulate the specific engineering problem, define performance metrics for success evaluation, and outline any constraints imposed by domain requirements or resource limitation.
- *Provide detailed information on data collection/preprocessing*: Explain the dataset's origin, sampling strategy, and rationale behind data source selection. Describe cleaning processes, feature extraction methods, normalization techniques, and data augmentation steps for replicability and potential challenges.
- *Specify model architecture and hyperparameters*: Justify the choice of algorithm and elucidate the tuning of hyperparameters (e.g., number of layers, learning rate) to enhance reproducibility.
- *Document training procedures and computational resources*: Specify the hardware configuration (GPU/CPU type, etc.) and software environment (Python version, library dependencies, operating system, etc.). Mention training epochs, batch size, and convergence criteria employed.
- *Present performance metrics with transparency*: Select domain-appropriate metrics, incorporate confidence intervals or statistical significance tests to showcase result robustness, and demonstrate alignment with real-world engineering needs.
- *Compare baseline and benchmark models*: Provide context by referencing existing solutions or classical engineering models to help readers assess performance gains and trade-offs in accuracy, speed, or resource consumption.
- *Include error analysis and failure cases*: Highlight instances of model underperformance, elucidate potential causes (e.g., data imbalance, insufficient training samples, domain-specific anomalies), and propose corrective actions like additional data collection or model architecture refinements.
- *Evaluate model robustness and generalizability*: Illustrate the model's performance across diverse data variations, environmental conditions, or operational scenarios using cross-validation or out-of-sample testing. Discuss prediction reliability in realistic engineering applications.
- *Provide version control and traceability*: Maintain a repository (e.g., Git) that captures code changes, model versions, and relevant documentation for each iteration, ensuring that future team members can seamlessly trace model evolution.
- *Ensure compliance with ethical, regulatory, and safety standards*: Document how sensitive data is protected, how decisions align with regulatory guidelines (e.g., for medical devices or autonomous vehicles), and what safety assurances are embedded in the ML solution.
- *Outline limitations and future work*: Provide a candid assessment of model deficiencies and identify areas necessitating further investigation or engineering resources to encourage continuous refinement.
- *Summarize findings in a concise, accessible format*: Despite the technical depth, provide a succinct executive summary that can be shared with stakeholders who require high-level insights rather than detailed methodological explanations.

3 Conclusions

In this short paper, we presented a hands-on tutorial for engineers with limited ML backgrounds by demonstrating the development of regression, classification, and clustering models in Python. We highlighted best practices for dataset generation, preprocessing, training, and validation, elucidated through concrete code examples. Moreover, we introduced explainability techniques using SHAP to focus on the importance of model transparency.

While this tutorial provides a comprehensive introduction to key ML tasks, it does have its limitations. The script developed herein does not extend into advanced topics such as deep learning, neural networks, or reinforcement learning—domains within ML that are rapidly advancing. These more sophisticated techniques, despite their enhanced capabilities, necessitate a profound grasp of both theoretical underpinnings and technical infrastructure. Consequently, their inclusion would surpass the tutorial's intended scope, which is tailored for engineers with minimal programming expertise. Future endeavors could expand upon this foundation to delve deeper into these advanced topics. Similarly, while this tutorial only scratches the surface of what is possible in ML, it lays the groundwork for more advanced explorations, as well such as (graph/generative neural networks, multi-objective optimization, etc.), data types (text-based, images/audio data, etc.) and problems (i.e., semi-supervised learning, association rules, symbolic regression, etc.) to be tackled in the future. We encourage engineers to tailor these examples to their specific domains, refining and expanding upon the techniques presented herein.

Supplementary Information

The online version contains supplementary material available at https://doi.org/10.1007/s43503-025-00053-x.

Supplementary material 1.

Author contribution

MZ Naser performed the analysis, experimental testing and writing of the manuscript (initial draft, review, editing and final draft).

Data availability

Data is available on request from the author.

Declarations

Competing interests

The author declares no conflict of interest.

Received: 8 January 2025 Revised: 23 February 2025 Accepted: 1 March 2025
Published online: 21 April 2025

References

al-Bashiti, M. K., & Naser, M. Z. (2022). Verifying domain knowledge and theories on Fire-induced spalling of concrete through eXplainable artificial intelligence. *Construction and Building Materials, 348*, 128648.

Ben Chaabene, W., Flah, M., & Nehdi, M. L. (2020). Machine learning prediction of mechanical properties of concrete: Critical review. *Construction and Building Materials*. https://doi.org/10.1016/j.conbuildmat.2020.119889

Bishop, C. (2007). Pattern recognition and machine learning. *Technometrics*. https://doi.org/10.1198/tech.2007.s518

Frank I, Todeschini R. (1994). The data analysis handbook. Retrieved June 21, 2019, from https://books.google.com/books?hl=en&lr=&id=SXEpB0H6L3YC&oi=fnd&pg=PP1&ots=zfmIRO_XO5&sig=dSX6KJdkuav5zRNxaUdcftGSn2k

Goodfellow I, Benigo Y, Courville A. (2016). Deep Learning (Adaptive Computation and Machine Learning series): Ian Goodfellow, Yoshua Bengio, Aaron Courville: 9780262035613: Amazon.com: Books, MIT Press.

Hastie T, Tibshirani R, Friedman JH, MyiLibrary. (2001). The elements of statistical learning data mining, inference, and prediction : with 200 full-color illustrations, Springer Series in Statistics.

Karniadakis, G. E., Kevrekidis, I. G., Lu, L., Perdikaris, P., Wang, S., & Yang, L. (2021). Physics-informed machine learning. *Nature Reviews Physics*. https://doi.org/10.1038/s42254-021-00314-5

Lundberg SM. Lee SI. (2017). A unified approach to interpreting model predictions, in: Adv. Neural Inf. Process. Syst.

Malekloo, A., Ozer, E., AlHamaydeh, M., & Girolami, M. (2022). Machine learning and structural health monitoring overview with emerging technology and high-dimensional data source highlights. *Structural Health Monitoring*. https://doi.org/10.1177/14759217211036880

Molnar, C., Casalicchio, G., & Bischl, B. (2020). Interpretable machine learning—A brief history, state-of-the-art and challenges, communications in computer and information. *Science, 1323*, 417–431. https://doi.org/10.1007/978-3-030-65965-3_28

Naser, M. Z. (2019). Can past failures help identify vulnerable bridges to extreme events? A biomimetical machine learning approach. *Engineering with Computers*. https://doi.org/10.1007/s00366-019-00874-2

Naser, M. Z. (2023). *Machine learning for civil and environmental engineers: a practical approach to data-driven analysis, explainability, and causality*. Wiley.

Nguyen-Sy, T., Wakim, J., To, Q. D., Vu, M. N., Nguyen, T. D., & Nguyen, T. T. (2020). Predicting the compressive strength of concrete from its compositions and age using the extreme gradient boosting method. *Construction and Building Materials, 260*, 119757. https://doi.org/10.1016/j.conbuildmat.2020.119757

Pine DJ. (2019). Introduction to Python for Science and Engineering. https://doi.org/10.1201/9780429506413

Ribeiro MT, Singh S, Guestrin C. (2016). Why should i trust you? Explaining the predictions of any classifier, in: Proc ACM SIGKDD Int. Conf. Knowl. Discov. Data Min. https://doi.org/10.1145/2939672.2939778.

Riley, R. D., Snell, K. I. E., Ensor, J., Burke, D. L., Harrell, F. E., Moons, K. G. M., & Collins, G. S. (2019). Minimum sample size for developing a multivariable prediction model: PART II—binary and time-to-event outcomes. *Statistics in Medicine*. https://doi.org/10.1002/sim.7992

Tapeh, A., & Naser, M. Z. (2022). Artificial Intelligence, machine learning, and deep learning in structural engineering: a scientometrics review of trends and best practices. *Archives of Computational Methods in Engineering*. https://doi.org/10.1007/s11831-022-09793-w

Thai, H.-T.T. (2022). Machine learning for structural engineering: A state-of-the-art review. *Elsevier*. https://doi.org/10.1016/j.istruc.2022.02.003

van Smeden, M., Moons, K. G., de Groot, J. A., Collins, G. S., Altman, D. G., Eijkemans, M. J., & Reitsma, J. B. (2018). Sample size for binary logistic prediction models: beyond events per variable criteria. *Statistical Methods in Medical Research, 28*, 2455–2474. https://doi.org/10.1177/0962280218784726

Xie, Q., Suvarna, M., Li, J., Zhu, X., Cai, J., & Wang, X. (2021). Online prediction of mechanical properties of hot rolled steel plate using machine learning. *Materials and Design*. https://doi.org/10.1016/j.matdes.2020.109201

You, K., Zhou, C., & Ding, L. (2023). Deep learning technology for construction machinery and robotics. *Automation in Construction*. https://doi.org/10.1016/j.autcon.2023.104852

Yuan F-G, Zargar SA, Chen Q, Wang S. (2020). Machine learning for structural health monitoring: challenges and opportunities. https://doi.org/10.1117/12.2561610.

Zarringol, M., & Thai, H. T. (2022). Prediction of the load-shortening curve of CFST columns using ANN-based models. *Journal of Building Engineering*. https://doi.org/10.1016/j.jobe.2022.104279

Publisher's Note