Examples and Case Studies for MAC Workshop of AIChE

Example 1

Prompt:

Using Prompt: Write a Python program to model a CSTR with first-order kinetics. What AI/ML does here: Traditionally, we'd derive and solve the mass balance ODEs analytically or numerically. AI/ML can instead learn a surrogate model of the CSTR from simulation or plant data mapping inputs (feed, temperature, residence time) to outputs (conversion, concentration) without explicitly solving the equations each time. This gives a fast, data-driven "digital twin" for optimisation, control, or fault detection. In this instance, What's the use of Digital Twin with respect to the model of CSTR where AI/ML can instead learn a surrogate model of the CSTR. [What's implied by a surrogate model, is it a replica model which is operational or an imaginary model?]

In this CSTR example, the surrogate model is not a physical replica and it is not merely an imaginary picture. It is a simplified mathematical or data-driven approximation of the real CSTR.

What is implied by a surrogate model?

A rigorous CSTR model may solve equations such as:

$$\frac{dC_A}{dt} = \frac{C_{A0} - C_A}{\tau} - k(T)C_A$$

For every new feed concentration, temperature, flow rate, or residence time, the differential equations must be solved again.

A surrogate model learns the relationship:

Inputs ⟶ CSTR outputs

For example:

$$[C_{A0}, T, \tau] \longrightarrow [C_A, X]$$

where:

- C_{A0} **= feed concentration**
- T **= reactor temperature**
- τ **= residence time**
- C_A **= outlet concentration**
- X **= conversion**

The surrogate may be an artificial neural network, random forest, Gaussian process, support-vector regression model, or another ML algorithm.

Once trained, it can predict CSTR behaviour without repeatedly solving the mass- and energy-balance equations.

Is the surrogate a replica?

It is best described as a behavioural approximation, not an exact replica.

It reproduces the input-output behaviour of the CSTR within the range of conditions on which it was trained. It may predict that:

- **Increasing temperature increases the reaction rate.**
- **Increasing residence time increases conversion.**
- **Increasing feed concentration changes outlet concentration.**
- **Cooling failure may cause an abnormal temperature rise.**

However, it does not necessarily contain the actual reaction mechanism, mass-balance equations, heat-transfer equations, or reactor geometry.

Therefore:

A surrogate model imitates the behaviour of the CSTR, whereas a first-principles model describes why the CSTR behaves that way.

Where does the digital twin enter?

The surrogate model can become one component of a digital twin.

A CSTR digital twin may contain:

CSTR data + engineering model + surrogate model + real-time updating

The surrogate provides fast predictions, while the wider digital twin connects those predictions to the physical reactor.

For example, a digital twin may receive real-time values of:

- Feed flow rate.
- Feed concentration.
- Reactor temperature.
- Cooling-water temperature.
- Agitator speed.
- Reactor level.
- Outlet concentration.

The surrogate then rapidly predicts:

- Conversion.
- Outlet concentration.
- Heat-generation rate.
- Future reactor temperature.
- Probability of off-specification product.
- Probability of cooling-system failure or runaway conditions.

Is it operational or imaginary?

That depends on how it is used.

1. Offline surrogate model

If the model is trained from simulated data and used only on a computer, it is an offline virtual model.

It may be used for:

- Education.
- Process design.

- Sensitivity analysis.
- What-if studies.
- Optimisation.

It is not yet a complete operational digital twin.

2. Online surrogate model

If the model receives real-time plant measurements, it becomes an online predictive model.

It can continuously estimate conversion, concentration, heat release, or other variables.

3. Operational digital twin

It becomes a practical digital twin when it is connected to the physical CSTR and continuously updated using real plant data.

The digital twin may then:

- Monitor the reactor.
- Compare actual and predicted performance.
- Detect abnormal deviations.
- Predict future operating conditions.
- Test corrective actions virtually.
- Recommend changes to temperature, flow rate, cooling, or residence time.
- Support operators and control systems.

Therefore, an operational digital twin is not imaginary. It is a software-based virtual representation that is linked to an operating physical reactor.

CSTR example

Suppose an ANN surrogate is trained using 20,000 CSTR simulations covering different:

- **Feed concentrations.**
- **Temperatures.**
- **Residence times.**
- **Cooling-water flow rates.**
- **Reaction-rate constants.**

The ANN learns:

$$[C_{A0}, T, \tau, F_c] \rightarrow [C_A, X, T_{\text{out}}]$$

When the real reactor is operating, the digital twin receives the current measurements and passes them to the ANN.

The ANN might predict:

$$X = 82\%$$

and also forecast that, if cooling-water flow continues to decrease:

$$T_{\text{reactor}} = 390 \text{ K in 10 minutes}$$

The digital twin could then test possible actions virtually:

- **Increase cooling-water flow.**
- **Reduce feed rate.**
- **Lower feed temperature.**

- Reduce reactant concentration.
- Initiate an emergency shutdown.

The surrogate supplies the rapid calculation, while the digital twin provides the real-time connection, monitoring, visualisation, prediction, and decision-support framework.

Important distinction

Term	Meaning
First-principles CSTR model	Uses mass balances, energy balances, kinetics, and heat-transfer equations
Surrogate model	Learns an approximate input-output relationship from simulation or plant data
Simulation model	Runs hypothetical CSTR operating scenarios, usually offline
Digital twin	A virtual representation linked to the physical CSTR and updated with operating data
AI-enabled digital twin	A digital twin using ML or ANN models for prediction, anomaly detection, optimisation, and decision support

The technically accurate conclusion is:

In the CSTR example, the AI/ML surrogate is a fast behavioural approximation of the reactor. When this surrogate is integrated with real-time reactor data, engineering models, monitoring systems, and operational feedback, it forms part of an AI-enabled digital twin. The surrogate is therefore neither a physical replica nor merely an imaginary model; it is a computational representation whose

operational value depends on its connection to the actual reactor.

```
"""
First-order isothermal CSTR plus an ANN surrogate digital twin.

Reaction: A -> B
Dynamic balance:
    dCA/dt = (CA0 - CA)/tau - k(T)*CA
Arrhenius kinetics:
    k(T) = A*exp[-Ea/(R*T)]
Steady state:
    CA = CA0/(1 + k*tau)
    X  = 1 - CA/CA0
"""

from pathlib import Path

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from scipy.integrate import solve_ivp
from sklearn.compose import TransformedTargetRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler


# ------------------------- Engineering constants -------------------------

R = 8.314                 # J/(mol K)
A = 1.0e6                 # 1/min
EA = 50_000.0             # J/mol
SEED = 42
N_SAMPLES = 3000

OUTDIR = Path.cwd() / "cstr_results"
OUTDIR.mkdir(exist_ok=True)


def rate_constant(temperature_k):
    """First-order Arrhenius rate constant, 1/min."""
    temperature_k = np.asarray(temperature_k, dtype=float)
```

```
    if np.any(temperature_k <= 0):
        raise ValueError("Temperature must be greater than 0 K.")
    return A * np.exp(-EA / (R * temperature_k))


def cstr_steady_state(feed_concentration, temperature_k, residence_time_min):
    """Ideal isothermal CSTR steady-state solution."""
    if np.any(np.asarray(feed_concentration) <= 0):
        raise ValueError("Feed concentration must be positive.")
    if np.any(np.asarray(residence_time_min) <= 0):
        raise ValueError("Residence time must be positive.")

    k = rate_constant(temperature_k)
    outlet_concentration = feed_concentration / (1.0 + k * residence_time_min)
    conversion = 1.0 - outlet_concentration / feed_concentration
    return outlet_concentration, conversion, k


def cstr_ode(t, y, feed_concentration, temperature_k, residence_time_min):
    """Dynamic component-A balance."""
    ca = y[0]
    k = float(rate_constant(temperature_k))
    dca_dt = (feed_concentration - ca) / residence_time_min - k * ca
    return [dca_dt]


def generate_dataset(n_samples=3000, seed=42):
    """Generate physically consistent synthetic operating data."""
    rng = np.random.default_rng(seed)

    ca0 = rng.uniform(0.50, 2.00, n_samples)     # mol/L
    temperature = rng.uniform(320.0, 420.0, n_samples)  # K
    tau = rng.uniform(0.50, 20.0, n_samples)    # min

    ca, conversion, k = cstr_steady_state(ca0, temperature, tau)

    return pd.DataFrame({
        "Feed concentration, mol/L": ca0,
        "Temperature, K": temperature,
        "Residence time, min": tau,
        "Rate constant, 1/min": k,
        "Outlet concentration, mol/L": ca,
        "Conversion, fraction": conversion,
        "Conversion, %": 100.0 * conversion,
    })


def build_ann():
    """Multi-output ANN with scaling of both inputs and outputs."""
```

```
    regressor = Pipeline([
        ("input_scaler", StandardScaler()),
        ("ann", MLPRegressor(
            hidden_layer_sizes=(48, 48),
            activation="tanh",
            solver="adam",
            learning_rate_init=0.001,
            alpha=1.0e-5,
            max_iter=2000,
            early_stopping=True,
            validation_fraction=0.15,
            n_iter_no_change=50,
            random_state=SEED,
        )),
    ])

    return TransformedTargetRegressor(
        regressor=regressor,
        transformer=StandardScaler(),
    )


def report_metrics(actual, predicted):
    """Calculate engineering prediction metrics for both outputs."""
    names = ["Outlet concentration, mol/L", "Conversion, fraction"]
    rows = []

    for i, name in enumerate(names):
        mae = mean_absolute_error(actual[:, i], predicted[:, i])
        rmse = np.sqrt(mean_squared_error(actual[:, i], predicted[:, i]))
        r2 = r2_score(actual[:, i], predicted[:, i])
        rows.append([name, r2, mae, rmse])

    return pd.DataFrame(
        rows,
        columns=["Output", "R-squared", "MAE", "RMSE"],
    )


def simulate_dynamic_case(ca0, temperature, tau):
    """Numerically solve the transient CSTR balance."""
    t_end = 5.0 * tau
    t_eval = np.linspace(0.0, t_end, 500)

    solution = solve_ivp(
        cstr_ode,
        (0.0, t_end),
        y0=[ca0],
        t_eval=t_eval,
```

```
        args=(ca0, temperature, tau),
        rtol=1.0e-8,
        atol=1.0e-10,
    )

    if not solution.success:
        raise RuntimeError(solution.message)

    ca = solution.y[0]
    conversion = 1.0 - ca / ca0
    return solution.t, ca, conversion


def main():
    print("=" * 72)
    print("FIRST-ORDER CSTR WITH ANN SURROGATE DIGITAL TWIN")
    print("=" * 72)
    print(
        "Inputs: feed concentration, temperature, residence time\n"
        "Outputs: outlet concentration and conversion\n"
    )

    # 1. Generate first-principles data
    data = generate_dataset(N_SAMPLES, SEED)
    print("Dataset statistics:")
    print(data.describe().T.round(5))
    print("\nFirst five observations:")
    print(data.head().round(5).to_string(index=False))

    features = [
        "Feed concentration, mol/L",
        "Temperature, K",
        "Residence time, min",
    ]
    targets = [
        "Outlet concentration, mol/L",
        "Conversion, fraction",
    ]

    x_train, x_test, y_train, y_test = train_test_split(
        data[features],
        data[targets],
        test_size=0.20,
        random_state=SEED,
    )

    # 2. Train ANN surrogate
    model = build_ann()
    model.fit(x_train, y_train)
```

```
predicted = model.predict(x_test)
actual = y_test.to_numpy()
metrics = report_metrics(actual, predicted)

print(f"\nTraining observations: {len(x_train)}")
print(f"Test observations:     {len(x_test)}")
print("\nANN test-set performance:")
print(metrics.round(6).to_string(index=False))

conversion_row = metrics.loc[
    metrics["Output"] == "Conversion, fraction"
].iloc[0]

print(
    "\nConversion error interpretation:\n"
    f"MAE  = {conversion_row['MAE']:.6f} fraction "
    f"= {100.0 * conversion_row['MAE']:.3f} percentage points\n"
    f"RMSE = {conversion_row['RMSE']:.6f} fraction "
    f"= {100.0 * conversion_row['RMSE']:.3f} percentage points"
)

fitted_ann = model.regressor_.named_steps["ann"]
print(
    "\nANN details:\n"
    "Architecture: 3 inputs -> 48 -> 48 -> 2 outputs\n"
    "Activation: tanh\n"
    "Optimizer: Adam\n"
    f"Training iterations: {fitted_ann.n_iter_}\n"
    f"Final training loss: {fitted_ann.loss_:.8f}"
)

# 3. Compare physics and ANN for one new operating point
ca0 = 1.50
temperature = 380.0
tau = 8.0

new_case = pd.DataFrame({
    "Feed concentration, mol/L": [ca0],
    "Temperature, K": [temperature],
    "Residence time, min": [tau],
})

ca_physics, x_physics, k = cstr_steady_state(ca0, temperature, tau)
ca_ann, x_ann = model.predict(new_case)[0]

print("\nNew operating case:")
print(f"Feed concentration = {ca0:.3f} mol/L")
```

```
print(f"Temperature        = {temperature:.2f} K")
print(f"Residence time     = {tau:.2f} min")
print(f"Rate constant      = {float(k):.6f} 1/min")

print("\nFirst-principles result:")
print(f"Outlet concentration = {float(ca_physics):.6f} mol/L")
print(f"Conversion           = {100.0 * float(x_physics):.3f}%")

print("\nANN prediction:")
print(f"Outlet concentration = {ca_ann:.6f} mol/L")
print(f"Conversion           = {100.0 * x_ann:.3f}%")

# 4. Dynamic ODE simulation
time, ca_dynamic, x_dynamic = simulate_dynamic_case(
    ca0, temperature, tau
)

fig1, axes = plt.subplots(2, 1, figsize=(9, 8), sharex=True)

axes[0].plot(time, ca_dynamic, label="ODE solution")
axes[0].axhline(
    float(ca_physics), linestyle="--", label="Steady state"
)
axes[0].set_ylabel("Concentration, mol/L")
axes[0].set_title("First-Order CSTR Dynamic Response")
axes[0].grid(True, alpha=0.3)
axes[0].legend()

axes[1].plot(time, 100.0 * x_dynamic, label="ODE solution")
axes[1].axhline(
    100.0 * float(x_physics), linestyle="--", label="Steady state"
)
axes[1].set_xlabel("Time, min")
axes[1].set_ylabel("Conversion, %")
axes[1].grid(True, alpha=0.3)
axes[1].legend()

fig1.tight_layout()
fig1.savefig(
    OUTDIR / "cstr_dynamic_response.png",
    dpi=300,
    bbox_inches="tight",
)

# 5. ANN parity plots
fig2, axes = plt.subplots(1, 2, figsize=(12, 5))
plot_labels = ["Outlet concentration, mol/L", "Conversion, fraction"]
```

```
    for i, axis in enumerate(axes):
        lower = min(actual[:, i].min(), predicted[:, i].min())
        upper = max(actual[:, i].max(), predicted[:, i].max())

        axis.scatter(actual[:, i], predicted[:, i], alpha=0.55)
        axis.plot(
            [lower, upper],
            [lower, upper],
            linestyle="--",
            label="Perfect prediction",
        )
        axis.set_xlabel(f"First-principles {plot_labels[i]}")
        axis.set_ylabel(f"ANN-predicted {plot_labels[i]}")
        axis.grid(True, alpha=0.3)
        axis.legend()

    fig2.suptitle("ANN Surrogate: Predicted Versus Actual")
    fig2.tight_layout()
    fig2.savefig(
        OUTDIR / "cstr_ann_parity.png",
        dpi=300,
        bbox_inches="tight",
    )

    data.to_csv(OUTDIR / "cstr_dataset.csv", index=False)
    metrics.to_csv(OUTDIR / "cstr_ann_metrics.csv", index=False)

    print(f"\nFiles saved in: {OUTDIR}")
    print(
        "\nAI/ML role:\n"
        "The first-principles equations generate physically consistent data. "
        "The ANN learns the nonlinear input-output mapping and becomes a fast
"
        "surrogate digital twin. After validation, it can support
optimization, "
        "control, sensitivity analysis, and fault detection. It does not
replace "
        "engineering limits, plant validation, or process-safety review."
    )

    plt.show(block=True)


if __name__ == "__main__":
    main()
```

[Running] python -u "c:\Users\Kayode Coker\OneDrive\Desktop\Python Codes\First-order isothermal CSTR plus an ANN surrogate digital twin.py"

==========

FIRST-ORDER CSTR WITH ANN SURROGATE DIGITAL TWIN

==========

Inputs: feed concentration, temperature, residence time

Outputs: outlet concentration and conversion

Dataset statistics:

	count	mean	...	75%	max
Feed concentration, mol/L	3000.0	1.24927	...	1.63412	1.99965
Temperature, K	3000.0	368.99765	...	393.42548	419.97149
Residence time, min	3000.0	10.27909	...	15.01366	19.99955
Rate constant, 1/min	3000.0	0.14933	...	0.22978	0.60387
Outlet concentration, mol/L	3000.0	0.71933	...	1.03474	1.96080
Conversion, fraction	3000.0	0.42324	...	0.67056	0.91957

Conversion,
% 3000.0 42.32356 ... 67.05617 91.95656

[7 rows x 8 columns]

First five observations:

Feed concentration, mol/L	Temperature, K	Residence time, min	Rate constant, 1/min	Outlet concentration, mol/L	Conversion, fraction	Conversion, %
1.66093	402.92211	9.38179	0.32945	0.40601	0.75555	75.55505
1.15832	379.40960	2.81773	0.13064	0.84666	0.26906	26.90643
1.78790	344.48547	11.59888	0.02619	1.37129	0.23302	23.30153
1.54605	394.56750	3.55036	0.24018	0.83448	0.46025	46.02529
0.64127	328.44809	1.68869	0.01117	0.62940	0.01851	1.85103

Training observations: 2400

Test observations: 600

ANN test-set performance:

Output	R-squared	MAE	RMSE
Outlet concentration, mol/L	0.999018	0.008490	0.01342
Conversion, fraction	0.999049	0.005867	0.00863

Conversion error interpretation:

MAE = 0.005867 fraction = 0.587 percentage points

RMSE = 0.008630 fraction = 0.863 percentage points

ANN details:

Architecture: 3 inputs -> 48 -> 48 -> 2 outputs

Activation: tanh

Optimizer: Adam

Training iterations: 197

Final training loss: 0.00055281

New operating case:

Feed concentration = 1.500 mol/L

Temperature = 380.00 K

Residence time = 8.00 min

Rate constant = 0.133898 1/min

First-principles result:

Outlet concentration = 0.724225 mol/L

Conversion = 51.718%

ANN prediction:

Outlet concentration = 0.707596 mol/L

Conversion = 52.692%

Files saved in: c:\Users\Kayode Coker\OneDrive\Desktop\Python Codes\cstr_results

AI/ML role:

The first-principles equations generate physically consistent data. The ANN learns the nonlinear input-output mapping and becomes a fast surrogate digital twin. After validation, it can support optimization, control, sensitivity analysis, and fault detection. It does not replace engineering limits, plant validation, or process-safety review.

Output	(R^2)	MAE	RMSE

Outlet concentration	0.9990	0.00849 mol/L	0.01342 mol/L
Conversion	0.9990	0.00587	0.00863

Generate a neural network to predict distillation tray efficiency" What AI/ML does here:

- Tray efficiency depends on many interacting variables: vapour/liquid rates, physical properties, tray design, fouling, operating regime.
- A neural network can learn the nonlinear relationship between these variables and actual measured efficiency from plant or rigorous simulation data.
- Once trained, it becomes a predictive model that we can use for design, bottleneck, and real-time performance monitoring.

```
"""
DISTILLATION TRAY EFFICIENCY PREDICTION USING AN ARTIFICIAL NEURAL NETWORK

Author: A. Kayode Coker / AKC Technology
Purpose
-------
This program demonstrates how an artificial neural network (ANN) can learn the
nonlinear relationship between distillation-tray operating conditions, fluid
properties, tray geometry, fouling condition, and tray efficiency.

The program:
1. Prints the program title and engineering purpose.
2. Prints every input and output variable, unit, and specified range.
3. Generates a physically inspired synthetic dataset, or reads an external
CSV.
4. Prints dataset statistics and sample observations.
5. Splits the data into training and test subsets.
6. Trains an ANN regression model.
7. Prints the ANN architecture, training information, and test results.
8. Prints representative actual and predicted tray efficiencies.
9. Predicts efficiency for a new operating condition.
10. Generates, saves, describes, and displays four plots:
    a. Tray-efficiency distribution
    b. Predicted versus actual efficiency
    c. Residual plot
```

```
    d. ANN training-loss history

Engineering caution
-------------------
The synthetic dataset and its equations are educational. They do not replace
validated plant data, vendor tray-rating methods, rigorous rate-based
simulation,
hydraulic calculations, mass-transfer analysis, or professional engineering
review.
"""

from __future__ import annotations

from pathlib import Path
import warnings

import joblib
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.exceptions import ConvergenceWarning
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor
from sklearn.preprocessing import StandardScaler


#
==============================================================================
# 1. USER SETTINGS
#
==============================================================================
RANDOM_SEED = 42
NUMBER_OF_SAMPLES = 5000
TEST_FRACTION = 0.20

# Change to True to read validated plant or rigorous-simulation data.
USE_EXTERNAL_DATA = False
EXTERNAL_DATA_FILE = Path("tray_efficiency_data.csv")

# Save all numerical results, models, and plots in this folder.
OUTPUT_DIRECTORY = Path("tray_efficiency_results")

# Display the plots when the program finishes.
# In Visual Studio, each plot should appear in a plot window.
SHOW_PLOTS = True

# Save high-resolution PNG copies of all plots.
SAVE_PLOTS = True
```

```
#
# ============================================================================
# 2. MODEL VARIABLES
#
# ============================================================================
FEATURE_COLUMNS = [
    "vapour_rate_kg_s",
    "liquid_rate_kg_s",
    "relative_volatility",
    "liquid_viscosity_mPa_s",
    "surface_tension_mN_m",
    "tray_spacing_m",
    "weir_height_mm",
    "hole_area_fraction",
    "fouling_index",
]

TARGET_COLUMN = "tray_efficiency_percent"

VARIABLE_INFORMATION = {
    "vapour_rate_kg_s": {
        "description": "Vapour mass flow rate",
        "unit": "kg/s",
        "minimum": 5.0,
        "maximum": 40.0,
    },
    "liquid_rate_kg_s": {
        "description": "Liquid mass flow rate",
        "unit": "kg/s",
        "minimum": 5.0,
        "maximum": 60.0,
    },
    "relative_volatility": {
        "description": "Key-component relative volatility",
        "unit": "dimensionless",
        "minimum": 1.10,
        "maximum": 4.00,
    },
    "liquid_viscosity_mPa_s": {
        "description": "Liquid dynamic viscosity",
        "unit": "mPa.s",
        "minimum": 0.20,
        "maximum": 3.00,
    },
    "surface_tension_mN_m": {
        "description": "Liquid surface tension",
        "unit": "mN/m",
```

```
        "minimum": 15.0,
        "maximum": 40.0,
    },
    "tray_spacing_m": {
        "description": "Vertical tray spacing",
        "unit": "m",
        "minimum": 0.45,
        "maximum": 0.90,
    },
    "weir_height_mm": {
        "description": "Outlet-weir height",
        "unit": "mm",
        "minimum": 30.0,
        "maximum": 80.0,
    },
    "hole_area_fraction": {
        "description": "Fractional tray-hole area",
        "unit": "fraction",
        "minimum": 0.06,
        "maximum": 0.16,
    },
    "fouling_index": {
        "description": "Tray fouling severity index",
        "unit": "0 = clean; 1 = severe",
        "minimum": 0.0,
        "maximum": 1.0,
    },
    "tray_efficiency_percent": {
        "description": "Measured or simulated tray efficiency",
        "unit": "%",
        "minimum": 15.0,
        "maximum": 95.0,
    },
}


#
================================================================================
# 3. REPORTING UTILITIES
#
================================================================================
def print_section(title: str) -> None:
    """Print a consistent console-report section heading."""

    print("\n" + "=" * 100)
    print(title)
    print("=" * 100)
```

```
def print_program_title() -> None:
    """Print the program title and purpose."""

    print("\n" + "#" * 100)
    print("DISTILLATION TRAY EFFICIENCY PREDICTION USING AN ARTIFICIAL NEURAL
NETWORK")
    print("#" * 100)
    print(
        "\nPURPOSE:\n"
        "The ANN learns a nonlinear mapping from vapour and liquid rates,
fluid "
        "properties, tray design, and fouling condition to tray efficiency.
Once "
        "trained, it acts as a rapid data-driven surrogate model for design "
        "screening, bottleneck analysis, optimisation, and performance
monitoring."
    )


def print_variable_information() -> None:
    """Print the input/output names, descriptions, units, and specified
ranges."""

    print_section("INPUT AND OUTPUT DATA")

    rows = []
    for position, column in enumerate(FEATURE_COLUMNS, start=1):
        item = VARIABLE_INFORMATION[column]
        rows.append(
            {
                "No.": position,
                "Type": "Input",
                "Program column": column,
                "Engineering variable": item["description"],
                "Unit": item["unit"],
                "Minimum": item["minimum"],
                "Maximum": item["maximum"],
            }
        )

    target_item = VARIABLE_INFORMATION[TARGET_COLUMN]
    rows.append(
        {
            "No.": len(rows) + 1,
            "Type": "Output",
            "Program column": TARGET_COLUMN,
            "Engineering variable": target_item["description"],
            "Unit": target_item["unit"],
            "Minimum": target_item["minimum"],
```

```
            "Maximum": target_item["maximum"],
        }
    )

    variable_table = pd.DataFrame(rows)
    print(
        variable_table.to_string(
            index=False,
            formatters={
                "Minimum": lambda value: f"{value:.4f}",
                "Maximum": lambda value: f"{value:.4f}",
            },
        )
    )

    print(
        "\nANN mapping:\n"
        "[vapour rate, liquid rate, relative volatility, viscosity, surface "
        "tension, tray spacing, weir height, hole area, fouling]\n"
        "                                        -> tray efficiency (%)"
    )


#
# ============================================================================
# 4. SYNTHETIC DATA GENERATION
#
# ============================================================================
def generate_synthetic_tray_data(
    number_of_samples: int,
    random_seed: int,
) -> pd.DataFrame:
    """
    Generate a physically inspired educational tray-efficiency dataset.

    The equations reproduce qualitative effects expected from:
    - a favourable intermediate vapour loading;
    - low-vapour weeping;
    - high-vapour entrainment or flooding;
    - excessive liquid/downcomer loading;
    - viscosity and surface-tension effects;
    - tray spacing, weir height, and open area;
    - relative volatility;
    - tray fouling;
    - nonlinear interactions and measurement noise.
    """

    rng = np.random.default_rng(random_seed)
```

```
    vapour_rate = rng.uniform(5.0, 40.0, number_of_samples)
    liquid_rate = rng.uniform(5.0, 60.0, number_of_samples)
    relative_volatility = rng.uniform(1.10, 4.00, number_of_samples)
    liquid_viscosity = rng.uniform(0.20, 3.00, number_of_samples)
    surface_tension = rng.uniform(15.0, 40.0, number_of_samples)
    tray_spacing = rng.uniform(0.45, 0.90, number_of_samples)
    weir_height = rng.uniform(30.0, 80.0, number_of_samples)
    hole_area_fraction = rng.uniform(0.06, 0.16, number_of_samples)

    # Most synthetic observations are clean or mildly fouled.
    fouling_index = rng.beta(1.5, 4.0, number_of_samples)

    # Normalised operating/design variables.
    vapour_n = (vapour_rate - 5.0) / 35.0
    liquid_n = (liquid_rate - 5.0) / 55.0
    spacing_n = (tray_spacing - 0.45) / 0.45

    # Qualitative mass-transfer effects.
    volatility_effect = 13.0 * np.tanh(0.95 * (relative_volatility - 1.35))
    viscosity_penalty = 8.0 * np.log1p(liquid_viscosity)
    surface_tension_effect = 6.0 * np.exp(-((surface_tension - 26.0) / 8.0) **
2)

    # Favourable operating/design regions.
    vapour_optimum_effect = 12.0 * np.exp(-((vapour_n - 0.56) / 0.24) ** 2)
    liquid_optimum_effect = 9.0 * np.exp(-((liquid_n - 0.50) / 0.30) ** 2)
    weir_effect = 6.0 * np.exp(-((weir_height - 55.0) / 16.0) ** 2)
    hole_area_effect = 7.0 * np.exp(-((hole_area_fraction - 0.105) / 0.030) **
2)
    spacing_effect = 4.0 * spacing_n

    # Fouling reduces active area and mass-transfer performance.
    fouling_penalty = 24.0 * fouling_index + 8.0 * fouling_index**2

    # Simplified educational hydraulic limits.
    vapour_capacity = (
        20.0
        + 105.0 * hole_area_fraction
        + 10.0 * spacing_n
        + 0.10 * surface_tension
    )
    flooding_ratio = vapour_rate / vapour_capacity
    flooding_penalty = 38.0 * np.maximum(flooding_ratio - 0.88, 0.0) ** 1.5

    minimum_stable_vapour = 8.0 + 32.0 * hole_area_fraction + 0.04 *
liquid_rate
    weeping_ratio = vapour_rate / minimum_stable_vapour
    weeping_penalty = 24.0 * np.maximum(0.78 - weeping_ratio, 0.0) ** 1.4
```

```
    liquid_capacity = 38.0 + 0.28 * weir_height + 8.0 * spacing_n
    downcomer_ratio = liquid_rate / liquid_capacity
    downcomer_penalty = 22.0 * np.maximum(downcomer_ratio - 0.90, 0.0) ** 1.3

    # Nonlinear interactions to be learned by the ANN.
    interaction_effect = (
        4.5 * np.tanh((relative_volatility - 1.6) * (1.0 - fouling_index))
        + 3.0 * np.sin(np.pi * vapour_n) * np.exp(-0.5 * liquid_viscosity)
        + 2.5 * spacing_n * (1.0 - fouling_index)
    )

    measurement_noise = rng.normal(0.0, 2.2, number_of_samples)

    tray_efficiency = (
        43.0
        + volatility_effect
        - viscosity_penalty
        + surface_tension_effect
        + vapour_optimum_effect
        + liquid_optimum_effect
        + weir_effect
        + hole_area_effect
        + spacing_effect
        - fouling_penalty
        - flooding_penalty
        - weeping_penalty
        - downcomer_penalty
        + interaction_effect
        + measurement_noise
    )

    tray_efficiency = np.clip(tray_efficiency, 15.0, 95.0)

    return pd.DataFrame(
        {
            "vapour_rate_kg_s": vapour_rate,
            "liquid_rate_kg_s": liquid_rate,
            "relative_volatility": relative_volatility,
            "liquid_viscosity_mPa_s": liquid_viscosity,
            "surface_tension_mN_m": surface_tension,
            "tray_spacing_m": tray_spacing,
            "weir_height_mm": weir_height,
            "hole_area_fraction": hole_area_fraction,
            "fouling_index": fouling_index,
            "tray_efficiency_percent": tray_efficiency,
        }
    )
```

```
def load_external_data(file_path: Path) -> pd.DataFrame:
    """Load and validate plant or rigorous-simulation data from a CSV file."""

    if not file_path.exists():
        raise FileNotFoundError(
            f"The external data file was not found: {file_path.resolve()}"
        )

    data = pd.read_csv(file_path)
    required_columns = FEATURE_COLUMNS + [TARGET_COLUMN]
    missing_columns = [column for column in required_columns if column not in
data]

    if missing_columns:
        raise ValueError(
            "The external dataset is missing these required columns: "
            + ", ".join(missing_columns)
        )

    data = data[required_columns].copy()
    data = data.replace([np.inf, -np.inf], np.nan).dropna()

    if len(data) < 100:
        warnings.warn(
            "The dataset contains fewer than 100 complete observations. "
            "Model validation may be unreliable.",
            stacklevel=2,
        )

    if not data[TARGET_COLUMN].between(0.0, 100.0).all():
        raise ValueError("Tray efficiency must lie between 0 and 100
percent.")

    return data


#
================================================================================
# 5. DATA REPORT
#
================================================================================
def print_dataset_report(data: pd.DataFrame, source_description: str) -> None:
    """Print the dataset source, size, sample data, and descriptive
statistics."""

    print_section("GENERATED OR IMPORTED DATASET")
    print(f"Data source:                  {source_description}")
```

```
    print(f"Number of complete records:  {len(data)}")
    print(f"Number of ANN inputs:         {len(FEATURE_COLUMNS)}")
    print(f"Number of ANN outputs:        1")

    print("\nFIRST TEN INPUT/OUTPUT OBSERVATIONS")
    print("-" * 100)
    print(
        data.head(10).to_string(
            index=False,
            float_format=lambda value: f"{value:10.4f}",
        )
    )

    statistics = data.describe().T[["min", "max", "mean", "std"]]
    statistics.columns = ["Minimum", "Maximum", "Mean", "Standard deviation"]

    print("\nMINIMUM, MAXIMUM, MEAN, AND STANDARD DEVIATION")
    print("-" * 100)
    print(
        statistics.to_string(
            float_format=lambda value: f"{value:12.4f}"
        )
    )


#
================================================================================
# 6. ANN TRAINING
#
================================================================================
def train_ann_model(data: pd.DataFrame):
    """Split and scale the dataset, train the ANN, and return all results."""

    X = data[FEATURE_COLUMNS].to_numpy(dtype=float)
    y = data[TARGET_COLUMN].to_numpy(dtype=float)

    X_train, X_test, y_train, y_test = train_test_split(
        X,
        y,
        test_size=TEST_FRACTION,
        random_state=RANDOM_SEED,
    )

    X_scaler = StandardScaler()
    y_scaler = StandardScaler()

    X_train_scaled = X_scaler.fit_transform(X_train)
    X_test_scaled = X_scaler.transform(X_test)
    y_train_scaled = y_scaler.fit_transform(y_train.reshape(-1, 1)).ravel()
```

```
    ann_model = MLPRegressor(
        hidden_layer_sizes=(64, 32, 16),
        activation="relu",
        solver="adam",
        alpha=1.0e-4,
        batch_size=64,
        learning_rate_init=1.0e-3,
        max_iter=2500,
        early_stopping=True,
        validation_fraction=0.15,
        n_iter_no_change=40,
        random_state=RANDOM_SEED,
        verbose=False,
    )

    with warnings.catch_warnings():
        warnings.filterwarnings("ignore", category=ConvergenceWarning)
        ann_model.fit(X_train_scaled, y_train_scaled)

    y_pred_scaled = ann_model.predict(X_test_scaled)
    y_pred = y_scaler.inverse_transform(y_pred_scaled.reshape(-1, 1)).ravel()
    y_pred = np.clip(y_pred, 0.0, 100.0)

    return {
        "model": ann_model,
        "X_scaler": X_scaler,
        "y_scaler": y_scaler,
        "X_train": X_train,
        "X_test": X_test,
        "y_train": y_train,
        "y_test": y_test,
        "y_pred": y_pred,
    }


def calculate_metrics(y_true: np.ndarray, y_pred: np.ndarray) -> dict[str,
float]:
    """Calculate standard regression metrics."""

    return {
        "R2": float(r2_score(y_true, y_pred)),
        "MAE_percentage_points": float(mean_absolute_error(y_true, y_pred)),
        "RMSE_percentage_points": float(
            np.sqrt(mean_squared_error(y_true, y_pred))
        ),
    }
```

```python
def print_model_and_results(
    training_results: dict,
    metrics: dict[str, float],
) -> pd.DataFrame:
    """Print the data split, ANN details, metrics, and sample predictions."""

    model = training_results["model"]
    X_train = training_results["X_train"]
    X_test = training_results["X_test"]
    y_train = training_results["y_train"]
    y_test = training_results["y_test"]
    y_pred = training_results["y_pred"]

    print_section("TRAINING AND TEST DATA")
    print(f"Total observations:     {len(X_train) + len(X_test)}")
    print(f"Training observations:  {len(X_train)}")
    print(f"Test observations:      {len(X_test)}")
    print(f"Training fraction:      {100.0 * len(X_train) / (len(X_train) +
len(X_test)):.1f}%")
    print(f"Test fraction:          {100.0 * len(X_test) / (len(X_train) +
len(X_test)):.1f}%")
    print(f"Mean training target:   {np.mean(y_train):.4f}%")
    print(f"Mean test target:       {np.mean(y_test):.4f}%")

    print_section("ARTIFICIAL NEURAL NETWORK MODEL")
    print("Architecture:              9 inputs -> 64 -> 32 -> 16 -> 1 output")
    print("Number of hidden layers:   3")
    print("Hidden-layer neurons:      64, 32, and 16")
    print("Activation function:       ReLU")
    print("Optimisation algorithm:    Adam")
    print("Input-feature scaling:     StandardScaler")
    print("Output-target scaling:     StandardScaler")
    print("Early stopping:            Enabled")
    print(f"Maximum iterations:         {model.max_iter}")
    print(f"Actual training iterations:{model.n_iter_:>9d}")
    print(f"Final scaled training loss:{model.loss_:>13.8f}")

    print_section("ANN TEST-SET RESULTS")
    print(f"Coefficient of determination, R² = {metrics['R2']:.5f}")
    print(
        "Mean absolute error, MAE          = "
        f"{metrics['MAE_percentage_points']:.4f} percentage points"
    )
    print(
        "Root mean squared error, RMSE     = "
        f"{metrics['RMSE_percentage_points']:.4f} percentage points"
    )
```

```
    print(
        "\nINTERPRETATION:\n"
        f"The ANN explains approximately {100.0 * metrics['R2']:.2f}% of the "
        "test-data variation. Its average absolute prediction error is "
        f"{metrics['MAE_percentage_points']:.3f} tray-efficiency percentage "
        "points, while larger errors are weighted more heavily by the RMSE of
"
        f"{metrics['RMSE_percentage_points']:.3f} percentage points."
    )

    prediction_table = pd.DataFrame(
        {
            "Actual efficiency (%)": y_test,
            "ANN prediction (%)": y_pred,
            "Residual, actual - predicted (percentage points)": y_test -
y_pred,
            "Absolute error (percentage points)": np.abs(y_test - y_pred),
        }
    )

    print("\nFIRST FIFTEEN TEST-SET PREDICTIONS")
    print("-" * 100)
    print(
        prediction_table.head(15).to_string(
            index=False,
            float_format=lambda value: f"{value:12.4f}",
        )
    )

    return prediction_table


#
================================================================================
# 7. NEW-CONDITION PREDICTION
#
================================================================================
def classify_efficiency(efficiency_percent: float) -> str:
    """Return an illustrative efficiency-performance classification."""

    if efficiency_percent >= 75.0:
        return "High predicted tray efficiency"
    if efficiency_percent >= 55.0:
        return "Moderate predicted tray efficiency"
    return "Low efficiency: investigate hydraulics, fouling, and data quality"


def predict_new_condition(
    model: MLPRegressor,
```

```
    X_scaler: StandardScaler,
    y_scaler: StandardScaler,
) -> tuple[pd.DataFrame, float]:
    """Predict tray efficiency for one illustrative operating condition."""

    new_condition = pd.DataFrame(
        [
            {
                "vapour_rate_kg_s": 24.0,
                "liquid_rate_kg_s": 32.0,
                "relative_volatility": 2.20,
                "liquid_viscosity_mPa_s": 0.85,
                "surface_tension_mN_m": 25.0,
                "tray_spacing_m": 0.60,
                "weir_height_mm": 50.0,
                "hole_area_fraction": 0.105,
                "fouling_index": 0.15,
            }
        ]
    )

    X_new_scaled = X_scaler.transform(
        new_condition[FEATURE_COLUMNS].to_numpy(dtype=float)
    )
    y_new_scaled = model.predict(X_new_scaled)
    prediction = y_scaler.inverse_transform(
        y_new_scaled.reshape(-1, 1)
    ).ravel()[0]
    prediction = float(np.clip(prediction, 0.0, 100.0))

    print_section("PREDICTION FOR A NEW OPERATING CONDITION")

    display_rows = []
    for column in FEATURE_COLUMNS:
        item = VARIABLE_INFORMATION[column]
        display_rows.append(
            {
                "Input variable": item["description"],
                "Value": float(new_condition.loc[0, column]),
                "Unit": item["unit"],
            }
        )

    print(
        pd.DataFrame(display_rows).to_string(
            index=False,
            formatters={"Value": lambda value: f"{value:.4f}"},
        )
```

```
    )

    print(f"\nANN-predicted tray efficiency = {prediction:.2f}%")
    print(f"Performance classification     =
{classify_efficiency(prediction)}")

    return new_condition, prediction


#
================================================================================
# 8. PLOTS
#
================================================================================
def save_figure(figure: plt.Figure, file_path: Path) -> None:
    """Save one figure when plot saving is enabled."""

    if SAVE_PLOTS:
        figure.savefig(file_path, dpi=300, bbox_inches="tight")


def create_plots(
    data: pd.DataFrame,
    training_results: dict,
    metrics: dict[str, float],
) -> list[tuple[str, Path, str]]:
    """Create four plots and return their titles, paths, and descriptions."""

    y_test = training_results["y_test"]
    y_pred = training_results["y_pred"]
    model = training_results["model"]
    residuals = y_test - y_pred

    plot_information: list[tuple[str, Path, str]] = []

    # Plot 1: Efficiency distribution
    file_path = OUTPUT_DIRECTORY / "01_tray_efficiency_distribution.png"
    fig1, ax1 = plt.subplots(figsize=(8, 6))
    ax1.hist(data[TARGET_COLUMN], bins=30, edgecolor="black", alpha=0.75)
    ax1.axvline(
        data[TARGET_COLUMN].mean(),
        linewidth=2,
        linestyle="--",
        label=f"Mean = {data[TARGET_COLUMN].mean():.2f}%",
    )
    ax1.set_xlabel("Tray efficiency (%)")
    ax1.set_ylabel("Number of observations")
    ax1.set_title("Distribution of Distillation Tray Efficiency")
    ax1.grid(True, alpha=0.30)
```

```
    ax1.legend()
    fig1.tight_layout()
    save_figure(fig1, file_path)
    plot_information.append(
        (
            "Plot 1: Tray-efficiency distribution",
            file_path,
            "Shows the range and frequency distribution of the generated or "
            "imported tray-efficiency data used to train and test the ANN.",
        )
    )

    # Plot 2: Parity plot
    file_path = OUTPUT_DIRECTORY / "02_predicted_versus_actual.png"
    lower = min(float(y_test.min()), float(y_pred.min()))
    upper = max(float(y_test.max()), float(y_pred.max()))

    fig2, ax2 = plt.subplots(figsize=(8, 6))
    ax2.scatter(y_test, y_pred, alpha=0.60)
    ax2.plot(
        [lower, upper],
        [lower, upper],
        linewidth=2,
        linestyle="--",
        label="Perfect prediction",
    )
    ax2.set_xlabel("Actual tray efficiency (%)")
    ax2.set_ylabel("ANN-predicted tray efficiency (%)")
    ax2.set_title(
        "Predicted versus Actual Tray Efficiency\n"
        f"R² = {metrics['R2']:.4f}; "
        f"MAE = {metrics['MAE_percentage_points']:.3f}; "
        f"RMSE = {metrics['RMSE_percentage_points']:.3f} percentage points"
    )
    ax2.grid(True, alpha=0.30)
    ax2.legend()
    fig2.tight_layout()
    save_figure(fig2, file_path)
    plot_information.append(
        (
            "Plot 2: Predicted versus actual efficiency",
            file_path,
            "Compares ANN predictions with actual test values. Points close to
"
            "the 45-degree line indicate accurate predictions.",
        )
    )
```

```
    # Plot 3: Residual plot
    file_path = OUTPUT_DIRECTORY / "03_residual_plot.png"
    fig3, ax3 = plt.subplots(figsize=(8, 6))
    ax3.scatter(y_pred, residuals, alpha=0.60)
    ax3.axhline(0.0, linewidth=2, linestyle="--")
    ax3.set_xlabel("ANN-predicted tray efficiency (%)")
    ax3.set_ylabel("Residual: actual - predicted (percentage points)")
    ax3.set_title("Residual Analysis for the Tray-Efficiency ANN")
    ax3.grid(True, alpha=0.30)
    fig3.tight_layout()
    save_figure(fig3, file_path)
    plot_information.append(
        (
            "Plot 3: Prediction residuals",
            file_path,
            "Shows prediction error against predicted efficiency. A random
band "
            "around zero suggests limited systematic bias.",
        )
    )

    # Plot 4: Training-loss history
    file_path = OUTPUT_DIRECTORY / "04_ann_training_loss.png"
    iterations = np.arange(1, len(model.loss_curve_) + 1)

    fig4, ax4 = plt.subplots(figsize=(8, 6))
    ax4.plot(iterations, model.loss_curve_, linewidth=2)
    ax4.set_xlabel("Training iteration")
    ax4.set_ylabel("Scaled ANN loss")
    ax4.set_title("ANN Training-Loss History")
    ax4.grid(True, alpha=0.30)
    fig4.tight_layout()
    save_figure(fig4, file_path)
    plot_information.append(
        (
            "Plot 4: ANN training-loss history",
            file_path,
            "Shows how the ANN loss decreases during training. Flattening of
the "
            "curve indicates that further iterations provide little
improvement.",
        )
    )

    return plot_information


def print_plot_report(
    plot_information: list[tuple[str, Path, str]],
```

```
) -> None:
    """Print each plot title, description, and saved file location."""

    print_section("PLOTS GENERATED BY THE PROGRAM")

    for title, file_path, description in plot_information:
        print(f"\n{title}")
        print("-" * len(title))
        print(description)
        if SAVE_PLOTS:
            print(f"Saved as: {file_path.resolve()}")

    if SHOW_PLOTS:
        print(
            "\nThe four plots will now be displayed. Close the plot windows to
"
            "allow the program to finish."
        )


#
================================================================================
# 9. FILE SAVING
#
================================================================================
def save_numerical_outputs(
    data: pd.DataFrame,
    training_results: dict,
    metrics: dict[str, float],
    prediction_table: pd.DataFrame,
    new_condition: pd.DataFrame,
    new_prediction: float,
) -> None:
    """Save the dataset, metrics, predictions, model, and scalers."""

    data.to_csv(
        OUTPUT_DIRECTORY / "tray_efficiency_dataset.csv",
        index=False,
    )

    pd.DataFrame([metrics]).to_csv(
        OUTPUT_DIRECTORY / "tray_efficiency_ann_metrics.csv",
        index=False,
    )

    prediction_table.to_csv(
        OUTPUT_DIRECTORY / "tray_efficiency_test_predictions.csv",
        index=False,
    )
```

```
    new_condition_output = new_condition.copy()
    new_condition_output["predicted_tray_efficiency_percent"] = new_prediction
    new_condition_output.to_csv(
        OUTPUT_DIRECTORY / "new_condition_prediction.csv",
        index=False,
    )

    joblib.dump(
        training_results["model"],
        OUTPUT_DIRECTORY / "tray_efficiency_ann_model.joblib",
    )
    joblib.dump(
        training_results["X_scaler"],
        OUTPUT_DIRECTORY / "tray_efficiency_input_scaler.joblib",
    )
    joblib.dump(
        training_results["y_scaler"],
        OUTPUT_DIRECTORY / "tray_efficiency_target_scaler.joblib",
    )


def print_saved_files() -> None:
    """Print all files created in the output directory."""

    print_section("FILES SAVED")
    for file_path in sorted(OUTPUT_DIRECTORY.iterdir()):
        print(file_path.resolve())


#
# ============================================================================
# 10. MAIN PROGRAM
#
# ============================================================================
def main() -> None:
    """Run the complete tray-efficiency ANN workflow."""

    np.set_printoptions(precision=4, suppress=True)
    pd.set_option("display.width", 180)
    pd.set_option("display.max_columns", None)

    OUTPUT_DIRECTORY.mkdir(parents=True, exist_ok=True)

    print_program_title()
    print_variable_information()

    if USE_EXTERNAL_DATA:
        data = load_external_data(EXTERNAL_DATA_FILE)
```

```
        source_description = f"External CSV file:
{EXTERNAL_DATA_FILE.resolve()}"
    else:
        data = generate_synthetic_tray_data(
            NUMBER_OF_SAMPLES,
            RANDOM_SEED,
        )
        source_description = (
            "Physically inspired synthetic educational dataset "
            f"({NUMBER_OF_SAMPLES} observations; random seed = {RANDOM_SEED})"
        )

    print_dataset_report(data, source_description)

    training_results = train_ann_model(data)
    metrics = calculate_metrics(
        training_results["y_test"],
        training_results["y_pred"],
    )

    prediction_table = print_model_and_results(training_results, metrics)

    new_condition, new_prediction = predict_new_condition(
        training_results["model"],
        training_results["X_scaler"],
        training_results["y_scaler"],
    )

    save_numerical_outputs(
        data,
        training_results,
        metrics,
        prediction_table,
        new_condition,
        new_prediction,
    )

    plot_information = create_plots(data, training_results, metrics)
    print_plot_report(plot_information)
    print_saved_files()

    print_section("ENGINEERING CONCLUSION")
    print(
        "The trained ANN is a fast nonlinear surrogate model of tray
efficiency. "
        "It can support design screening, bottleneck studies, optimisation,
soft "
```

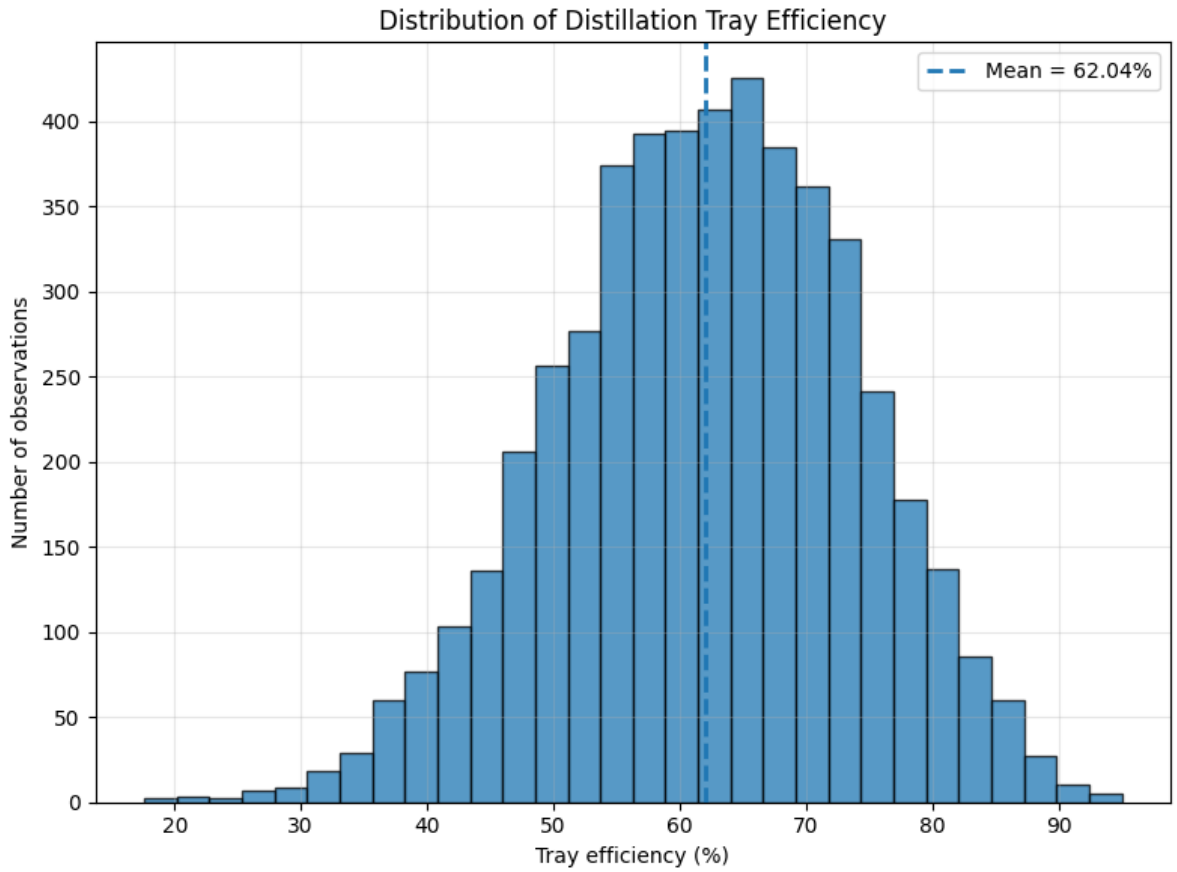
```
        "sensing, fouling surveillance, and real-time performance monitoring.
"
        "However, the model is reliable only within the range and quality of
its "
        "training data. Predictions outside those ranges are extrapolations
and "
        "must be checked using tray hydraulics, mass-transfer calculations, "
        "rigorous simulation, plant tests, and engineering judgement."
    )

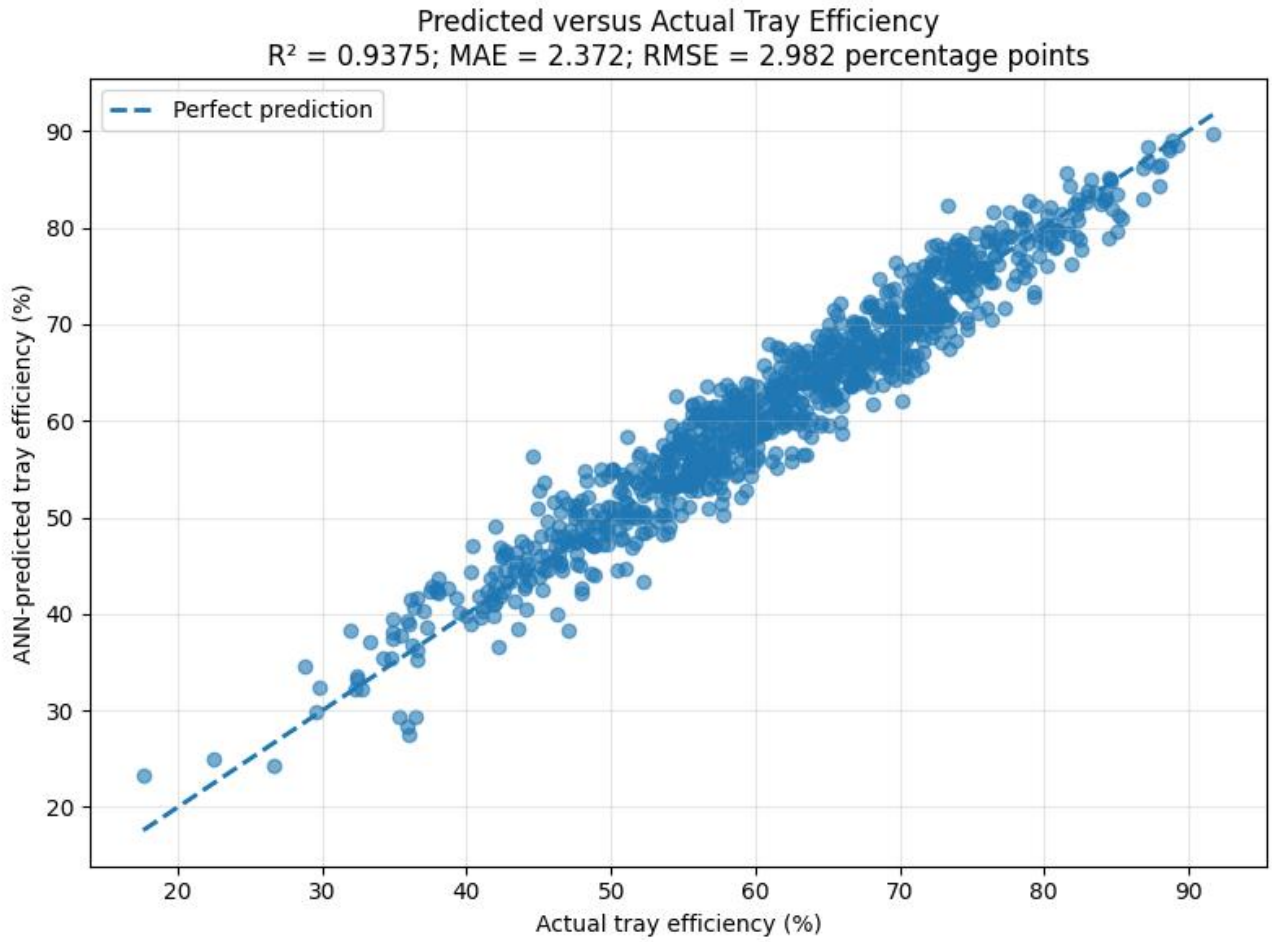
    if SHOW_PLOTS:
        plt.show()

    plt.close("all")

    print("\nPROGRAM COMPLETED SUCCESSFULLY.\n")


if __name__ == "__main__":
    try:
        main()
    except FileNotFoundError as error:
        print(f"\nFILE ERROR: {error}")
        raise SystemExit(1) from error
    except ValueError as error:
        print(f"\nDATA ERROR: {error}")
        raise SystemExit(1) from error
    except Exception as error:
        print(f"\nUNEXPECTED ERROR: {error}")
        raise
```


Distribution of Distillation Tray Efficiency
Mean = 62.04%
Number of observations
Tray efficiency (%)
0
50
100
150
200
250
300
350
400
20
30
40
50
60
70
80
90



CASE STUDY 1: JACKETED CSTR WITH ARRHENIUS k(T) AND COOLING FAILURE

The Python program simulates a jacketed Continuous Stirred Tank Reactor (CSTR) for a first-order exothermic reaction, A → B, using an Arrhenius temperature-dependent rate constant, k(T).

It compares two operating cases: normal cooling and cooling failure, where the heat-transfer coefficient UA is reduced to zero after 3 minutes. The program calculates the time variation of reactant concentration, product concentration, and reactor temperature, allowing the engineer to observe how loss of cooling affects conversion and reactor safety. The concentration plot shows that CA decreases as reactant A is consumed, while CB increases as product B is formed.

Under cooling failure, the temperature rise changes the reaction rate, causing a different concentration profile compared with normal cooling.

The temperature plot shows that the reactor remains more controlled under normal cooling, while the cooling failure case produces a sharper temperature rise after 3 minutes. This demonstrates the possible onset of an unsafe thermal condition.

AI/ML can strengthen this reactor analysis by learning patterns from reactor operating data, predicting abnormal temperature rise before it becomes dangerous, estimating T_failure, and supporting early-warning alarms.

In practical chemical and petroleum processing facilities, such models can be embedded in digital twins, advanced process control systems, and predictive safety tools to reduce the risk of thermal runaway, improve reactor reliability, and maintain safe, efficient operation.

```
"""
CASE STUDY 1: JACKETED CSTR WITH FIRST-ORDER KINETICS A -> B
Arrhenius k(T) and Cooling Failure Scenario
Author: Kayode Coker (A.K.C Technology Press)
"""

import numpy as np
import matplotlib.pyplot as plt

# ------------------------------------------------------------
# 1. TITLE
# ------------------------------------------------------------
print("\n============================================================")
print("CASE STUDY 1: JACKETED CSTR WITH ARRHENIUS k(T) AND COOLING FAILURE")
print("============================================================\n")
```

```
# ------------------------------------------------------------
# 2. INPUT DATA
# ------------------------------------------------------------
CA0 = 1.0         # mol/L
CB0 = 0.0         # mol/L
V = 10.0          # L
F = 1.0           # L/min
T0 = 300.0        # K (feed temperature)
Tj_normal = 290.0  # K (normal jacket temperature)
rho = 1000.0      # kg/m3
Cp = 4.18         # kJ/kg.K
UA_normal = 500.0  # kJ/min.K
dH = -50000.0     # J/mol (exothermic)

# Arrhenius parameters (chosen so k(T0) is significant)
k0 = 1.0e4        # 1/min
Ea = 20000.0      # J/mol
R = 8.314         # J/mol.K

tau = V / F       # residence time

def k_T(T):
    return k0 * np.exp(-Ea / (R * T))

k_T0 = k_T(T0)

print("INPUT DATA:")
print(f"Feed concentration CA0: {CA0} mol/L")
print(f"Reactor volume V: {V} L")
print(f"Flowrate F: {F} L/min")
print(f"Residence time tau: {tau} min")
print(f"Feed temperature T0: {T0} K")
print(f"Normal jacket temperature Tj: {Tj_normal} K")
print(f"Heat of reaction dH: {dH} J/mol")
print(f"Heat transfer UA (normal): {UA_normal} kJ/min.K")
print(f"Arrhenius parameters: k0 = {k0}, Ea = {Ea} J/mol")
print(f"Rate constant at T0: k(T0) = {k_T0:.3f} 1/min\n")

# ------------------------------------------------------------
# 3. KINETICS AND BALANCES
# ------------------------------------------------------------
def derivatives(CA, CB, T, Tj, UA):
    k = k_T(T)
    rA = -k * CA  # rate of A consumption (mol/L.min)

    dCA_dt = (F/V)*(CA0 - CA) + rA
    dCB_dt = (F/V)*(CB0 - CB) - rA
```

```
    # Energy balance (convert dH term to kJ)
    dT_dt = (F/V)*(T0 - T) \
            + (-dH * rA)/(rho * Cp * 1000.0) \
            + (UA/(rho * Cp * V))*(Tj - T)

    return dCA_dt, dCB_dt, dT_dt

# ------------------------------------------------------------------
# 4. SIMULATION FUNCTION (NORMAL AND FAILURE)
# ------------------------------------------------------------------
def simulate_cstr(Tj, UA, t_final=10.0, dt=0.01, t_fail=None):
    n_steps = int(t_final/dt) + 1
    time = np.linspace(0, t_final, n_steps)
    CA = np.zeros(n_steps)
    CB = np.zeros(n_steps)
    T = np.zeros(n_steps)

    CA[0] = CA0
    CB[0] = CB0
    T[0] = T0

    for i in range(n_steps-1):
        t = time[i]

        # Cooling failure: after t_fail, set UA = 0
        if (t_fail is not None) and (t >= t_fail):
            UA_eff = 0.0
        else:
            UA_eff = UA

        dCA_dt, dCB_dt, dT_dt = derivatives(CA[i], CB[i], T[i], Tj, UA_eff)

        CA[i+1] = CA[i] + dt * dCA_dt
        CB[i+1] = CB[i] + dt * dCB_dt
        T[i+1] = T[i] + dt * dT_dt

    return time, CA, CB, T

# ------------------------------------------------------------------
# 5. RUN SCENARIOS
# ------------------------------------------------------------------
print("Running normal cooling scenario...")
time_n, CA_n, CB_n, T_n = simulate_cstr(Tj_normal, UA_normal, t_final=10.0,
dt=0.01)

print("Running cooling failure scenario (UA -> 0 at t = 3 min)...")
t_fail = 3.0
```

```
time_f, CA_f, CB_f, T_f = simulate_cstr(Tj_normal, UA_normal, t_final=10.0,
dt=0.01, t_fail=t_fail)

# ------------------------------------------------------------
# 6. STEP-BY-STEP CALCULATIONS FOR FAILURE SCENARIO
# ------------------------------------------------------------
print("\nSTEP-BY-STEP CALCULATIONS (COOLING FAILURE SCENARIO):")
print("Time(min)\tCA(mol/L)\tCB(mol/L)\tT(K)")
for i in range(0, len(time_f), int(len(time_f)/10)):
    print(f"{time_f[i]:6.2f}\t{CA_f[i]:8.4f}\t{CB_f[i]:8.4f}\t{T_f[i]:8.2f}")

T_failure = T_f[-1]

print("\nFINAL RESULTS:")
print(f"Normal cooling: CA = {CA_n[-1]:.4f} mol/L, CB = {CB_n[-1]:.4f} mol/L,
T = {T_n[-1]:.2f} K")
print(f"Cooling failure: CA = {CA_f[-1]:.4f} mol/L, CB = {CB_f[-1]:.4f} mol/L,
T_failure = {T_failure:.2f} K\n")

# ------------------------------------------------------------
# 7. PLOTS: NORMAL VS FAILURE
# ------------------------------------------------------------
plt.figure(figsize=(12,5))

# Concentration profiles
plt.subplot(1,2,1)
plt.plot(time_n, CA_n, label="CA normal", color='blue')
plt.plot(time_n, CB_n, label="CB normal", color='green')
plt.plot(time_f, CA_f, "--", label="CA failure", color='blue')
plt.plot(time_f, CB_f, "--", label="CB failure", color='green')
plt.axvline(t_fail, color='gray', linestyle=':', label="Cooling failure at t=3
min")
plt.title("CSTR Concentration Profiles (A and B)")
plt.xlabel("Time (min)")
plt.ylabel("Concentration (mol/L)")
plt.grid(True)
plt.legend()

# Temperature profiles
plt.subplot(1,2,2)
plt.plot(time_n, T_n, label="T normal", color='red')
plt.plot(time_f, T_f, "--", label="T failure", color='orange')
plt.axvline(t_fail, color='gray', linestyle=':', label="Cooling failure at t=3
min")
plt.title("CSTR Temperature Profiles (Normal vs Failure)")
plt.xlabel("Time (min)")
plt.ylabel("Temperature (K)")
plt.grid(True)
```

```
plt.legend()

plt.tight_layout()
plt.show()

# ------------------------------------------------------------
# 8. DISCUSSION OF RESULTS
# ------------------------------------------------------------
print("""
DISCUSSION OF RESULTS:

With the revised Arrhenius parameters, the rate constant k(T) is large
enough at T0 to give significant conversion. As a result:

- The reactant concentration CA decreases asymptotically from CA0.
- The product concentration CB increases correspondingly, satisfying CB = CA0
- CA.
- The profiles are no longer flat; they show the expected CSTR behavior.

In the normal cooling case, the reactor temperature rises moderately and
stabilizes. In the cooling failure case (UA -> 0 at t = 3 min), the
temperature rises more sharply. The final temperature in this scenario
is referred to here as T_failure, indicating the higher, potentially
unsafe temperature reached after loss of cooling.
""")

# ------------------------------------------------------------
# 9. IMPLICATION OF AI/ML IN REACTOR PERFORMANCE
# ------------------------------------------------------------
print("""
IMPLICATION OF AI/ML IN REACTOR PERFORMANCE:

AI/ML models can:
- Learn how CA, CB, and T respond to changes in cooling (UA) and jacket
temperature.
- Predict T_failure under cooling loss scenarios and trigger alarms.
- Identify operating regions where conversion is high but T remains safe.
- Support advanced control strategies to avoid thermal runaway.

In chemical and petroleum processing industries, such models can be
embedded in digital twins and advanced control systems to improve
reactor safety, reliability, and energy efficiency.
""")
```

[Running] python -u "c:\Users\Kayode Coker\OneDrive\Desktop\Python Codes\Case Study 1. JACKETED CSTR WITH FIRST-ORDER KINETICS-akc1.py"

==========

CASE STUDY 1: JACKETED CSTR WITH ARRHENIUS k(T) AND COOLING FAILURE

==========

INPUT DATA:

Feed concentration CA0: 1.0 mol/L

Reactor volume V: 10.0 L

Flowrate F: 1.0 L/min

Residence time tau: 10.0 min

Feed temperature T0: 300.0 K

Normal jacket temperature Tj: 290.0 K

Heat of reaction dH: -50000.0 J/mol

Heat transfer UA (normal): 500.0 kJ/min.K

Arrhenius parameters: k0 = 10000.0, Ea = 20000.0 J/mol

Rate constant at T0: k(T0) = 3.293 1/min

Running normal cooling scenario...

Running cooling failure scenario (UA -> 0 at t = 3 min)...

STEP-BY-STEP CALCULATIONS (COOLING FAILURE SCENARIO):

Time(min)	CA(mol/L)	CB(mol/L)	T(K)
0.00	1.0000	0.0000	300.00
1.00	0.0605	0.9395	299.88
2.00	0.0306	0.9694	299.77
3.00	0.0297	0.9703	299.68
4.00	0.0297	0.9703	299.71
5.00	0.0297	0.9703	299.74
6.00	0.0297	0.9703	299.76
7.00	0.0296	0.9704	299.78
8.00	0.0296	0.9704	299.80
9.00	0.0296	0.9704	299.82
10.00	0.0296	0.9704	299.84

FINAL RESULTS:

Normal cooling: CA = 0.0300 mol/L, CB = 0.9700 mol/L, T = 299.27 K

Cooling failure: CA = 0.0296 mol/L, CB = 0.9704 mol/L, T_failure = 299.84 K

Python Code

```
"""
CASE STUDY 2:
Predicting Distillation Overhead Purity Using Machine Learning
```

```
Author: Kayode Coker
A.K.C. Technology Press

PURPOSE
-------
This program generates realistic synthetic distillation-column operating data,
trains an artificial neural network regression model, predicts overhead-
product
purity, evaluates the model, prints the results, saves the predicted-versus-
actual purity plot, displays the plot, and explains the engineering
implications.

NOTE
----
The dataset is synthetic and is intended for education and demonstration.
A plant application must use validated historian and laboratory data.
"""

from pathlib import Path
import os
import sys
import warnings
import webbrowser
from datetime import datetime

import matplotlib.pyplot as plt
import numpy as np
from sklearn.exceptions import ConvergenceWarning
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler


#
================================================================================
# SUPPORTING FUNCTIONS
#
================================================================================
def print_section(number: int, title: str) -> None:
    """Print a consistent heading for each section."""
    print("\n" + "=" * 78)
    print(f"{number}. {title}")
    print("=" * 78)


def describe_array(name: str, values: np.ndarray, unit: str) -> None:
```

```
    """Print summary statistics for an array."""
    print(
        f"{name:<30}"
        f"Minimum = {np.min(values):8.3f} {unit:<10} | "
        f"Maximum = {np.max(values):8.3f} {unit:<10} | "
        f"Mean = {np.mean(values):8.3f} {unit:<10} | "
        f"Std. dev. = {np.std(values):8.3f} {unit}"
    )


def get_output_directory() -> Path:
    """
    Return the folder containing the Python script.

    Saving beside the script avoids display problems caused by hidden,
    unavailable, or read-only subfolders.
    """
    if "__file__" in globals():
        return Path(__file__).resolve().parent

    return Path.cwd()


def open_saved_plot(plot_path: Path) -> bool:
    """Open the saved PNG with the operating system's default image viewer."""
    try:
        if sys.platform.startswith("win"):
            os.startfile(str(plot_path))  # type: ignore[attr-defined]
            return True

        return bool(webbrowser.open(plot_path.resolve().as_uri()))

    except Exception as exc:
        print(f"Automatic opening of the saved plot failed: {exc}")
        return False


def save_plot_with_fallback(
    figure: plt.Figure,
    output_directory: Path,
    preferred_filename: str,
) -> Path:
    """
    Save the plot using the preferred filename.

    If that file is locked by an open image viewer or cannot be overwritten,
    save a timestamped copy instead.
    """
    preferred_path = output_directory / preferred_filename
```

```
    try:
        figure.savefig(preferred_path, dpi=300, bbox_inches="tight")
        return preferred_path

    except PermissionError:
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        fallback_path = (
            output_directory
            / f"{preferred_path.stem}_{timestamp}{preferred_path.suffix}"
        )
        figure.savefig(fallback_path, dpi=300, bbox_inches="tight")

        print(
            "The previous plot file was open or locked, so a new "
            "timestamped plot file was created."
        )
        return fallback_path

    except OSError as exc:
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        fallback_directory = Path.cwd()
        fallback_path = (
            fallback_directory
            / f"{preferred_path.stem}_{timestamp}{preferred_path.suffix}"
        )

        try:
            figure.savefig(fallback_path, dpi=300, bbox_inches="tight")
            print(
                f"The preferred results folder was unavailable ({exc}). "
                "The plot was saved in the current working directory instead."
            )
            return fallback_path

        except OSError as second_exc:
            raise RuntimeError(
                "The plot could not be saved in either the preferred or "
                "fallback location."
            ) from second_exc


#
==============================================================================
# 1. PROGRAM TITLE
#
==============================================================================
print("\n" + "#" * 78)
print("CASE STUDY 2: DISTILLATION OVERHEAD PURITY PREDICTION")
```

```
print("ARTIFICIAL NEURAL NETWORK / MACHINE-LEARNING REGRESSION MODEL")
print("#" * 78)


#
# ============================================================================
# 2. PROGRAM DESCRIPTION
#
# ============================================================================
print_section(2, "PROGRAM DESCRIPTION")

print(
    """
The program investigates how five important distillation operating variables
influence the purity of a light overhead product:

1. Feed temperature
2. Reflux ratio
3. Light-component fraction in the feed
4. Column operating pressure
5. Overall tray efficiency

A multilayer perceptron artificial neural network is trained using 300
simulated operating observations. The model is then evaluated on independent
test data that were not used during training.
""".strip()
)


#
# ============================================================================
# 3. INPUT VARIABLES AND ENGINEERING RANGES
#
# ============================================================================
print_section(3, "INPUT VARIABLES AND ENGINEERING RANGES")

print("Feed temperature          : 120-180 degC")
print("Reflux ratio              : 1.20-3.50 mol reflux/mol distillate")
print("Light component in feed   : 0.10-0.35 mole fraction")
print("Column pressure           : 120-180 kPa absolute")
print("Overall tray efficiency   : 45-75 percent")
print("Predicted output          : Overhead light-component purity, mol
percent")


#
# ============================================================================
# 4. GENERATE SYNTHETIC DISTILLATION DATA
```

```
#
================================================================================
print_section(4, "SYNTHETIC DISTILLATION DATA GENERATION")

RANDOM_SEED = 42
NUMBER_OF_SAMPLES = 300
rng = np.random.default_rng(RANDOM_SEED)

feed_temp = rng.uniform(120.0, 180.0, NUMBER_OF_SAMPLES)
reflux_ratio = rng.uniform(1.2, 3.5, NUMBER_OF_SAMPLES)
feed_comp = rng.uniform(0.10, 0.35, NUMBER_OF_SAMPLES)
pressure = rng.uniform(120.0, 180.0, NUMBER_OF_SAMPLES)
tray_eff = rng.uniform(45.0, 75.0, NUMBER_OF_SAMPLES)

# Normalized variables used only to construct the synthetic output.
t_norm = (feed_temp - 120.0) / 60.0
r_norm = (reflux_ratio - 1.2) / 2.3
z_norm = (feed_comp - 0.10) / 0.25
p_norm = (pressure - 120.0) / 60.0
e_norm = (tray_eff - 45.0) / 30.0

measurement_noise = rng.normal(0.0, 0.22, NUMBER_OF_SAMPLES)

purity_pct = (
    89.0
    + 4.0 * r_norm
    + 2.4 * e_norm
    + 1.7 * z_norm
    - 1.3 * p_norm
    + 0.9 * t_norm
    + 0.9 * r_norm * e_norm
    - 0.5 * p_norm * z_norm
    + measurement_noise
)

purity_pct = np.clip(purity_pct, 80.0, 99.8)

X = np.column_stack(
    [feed_temp, reflux_ratio, feed_comp, pressure, tray_eff]
)
y = purity_pct

print(f"Random seed used           : {RANDOM_SEED}")
print(f"Number of observations     : {NUMBER_OF_SAMPLES}")

print("\nGenerated-data summary:")
describe_array("Feed temperature", feed_temp, "degC")
describe_array("Reflux ratio", reflux_ratio, "mol/mol")
```

```
describe_array("Light component in feed", feed_comp, "mole fraction")
describe_array("Column pressure", pressure, "kPa abs")
describe_array("Overall tray efficiency", tray_eff, "%")
describe_array("Overhead purity", purity_pct, "mol %")

print("\nFirst five generated observations:")
print(
    "Row | Feed T | Reflux | Feed light fraction | Pressure | Tray eff. |
Purity"
)
print("-" * 91)

for row in range(5):
    print(
        f"{row + 1:>3} | "
        f"{feed_temp[row]:6.2f} | "
        f"{reflux_ratio[row]:6.3f} | "
        f"{feed_comp[row]:19.4f} | "
        f"{pressure[row]:8.2f} | "
        f"{tray_eff[row]:9.2f} | "
        f"{purity_pct[row]:6.3f}%"
    )


#
================================================================================
# 5. TRAINING AND TEST DATA SPLIT
#
================================================================================
print_section(5, "TRAINING AND TEST DATA SPLIT")

X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.20,
    random_state=RANDOM_SEED,
)

print(f"Training observations      : {len(X_train)}")
print(f"Test observations          : {len(X_test)}")


#
================================================================================
# 6. DEFINE AND TRAIN THE ANN MODEL
#
================================================================================
print_section(6, "NEURAL-NETWORK MODEL DEFINITION AND TRAINING")
```

```
print("Model type                 : Multilayer Perceptron Regressor")
print("Hidden layers              : 20 neurons and 20 neurons")
print("Activation function        : ReLU")
print("Optimizer                  : Adam")
print("Maximum iterations         : 5000")
print("Feature preprocessing      : StandardScaler")

model = Pipeline(
    steps=[
        ("scaler", StandardScaler()),
        (
            "ann",
            MLPRegressor(
                hidden_layer_sizes=(20, 20),
                activation="relu",
                solver="adam",
                learning_rate_init=0.003,
                max_iter=5000,
                random_state=RANDOM_SEED,
                early_stopping=True,
                validation_fraction=0.15,
                n_iter_no_change=80,
                tol=1.0e-6,
            ),
        ),
    ]
)

with warnings.catch_warnings(record=True) as captured_warnings:
    warnings.simplefilter("always", ConvergenceWarning)
    model.fit(X_train, y_train)

ann = model.named_steps["ann"]

print("\nTraining completed.")
print(f"Iterations performed       : {ann.n_iter_}")
print(f"Final training loss        : {ann.loss_:.6f}")

if hasattr(ann, "best_validation_score_"):
    print(f"Best validation R-squared  : {ann.best_validation_score_:.6f}")

convergence_messages = [
    str(item.message)
    for item in captured_warnings
    if issubclass(item.category, ConvergenceWarning)
]

if convergence_messages:
```

```
    print("Convergence status         : Warning issued")
    for message in convergence_messages:
        print(f"  {message}")
else:
    print("Convergence status         : No convergence warning")


#
# ============================================================================
# 7. TEST-SET PREDICTIONS
#
# ============================================================================
print_section(7, "TEST-SET PURITY PREDICTIONS")

y_pred = model.predict(X_test)
residuals = y_pred - y_test
absolute_errors = np.abs(residuals)

print("First ten independent test predictions:")
print("Case | Actual purity | Predicted purity | Prediction error")
print("-" * 69)

for case_number, (actual, predicted, error) in enumerate(
    zip(y_test[:10], y_pred[:10], residuals[:10]),
    start=1,
):
    print(
        f"{case_number:>4} | "
        f"{actual:12.3f}% | "
        f"{predicted:15.3f}% | "
        f"{error:+10.3f} percentage points"
    )


#
# ============================================================================
# 8. MODEL PERFORMANCE RESULTS
#
# ============================================================================
print_section(8, "MODEL PERFORMANCE RESULTS")

mae = mean_absolute_error(y_test, y_pred)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
r2 = r2_score(y_test, y_pred)
bias = np.mean(residuals)
max_abs_error = np.max(absolute_errors)

within_025 = 100.0 * np.mean(absolute_errors <= 0.25)
within_050 = 100.0 * np.mean(absolute_errors <= 0.50)
```

```
within_100 = 100.0 * np.mean(absolute_errors <= 1.00)

print(f"Mean Absolute Error        : {mae:.4f} percentage points")
print(f"Root Mean Squared Error    : {rmse:.4f} percentage points")
print(f"R-squared score            : {r2:.4f}")
print(f"Mean prediction bias       : {bias:+.4f} percentage points")
print(f"Maximum absolute error     : {max_abs_error:.4f} percentage points")
print(f"Predictions within +/-0.25 : {within_025:.1f} percent")
print(f"Predictions within +/-0.50 : {within_050:.1f} percent")
print(f"Predictions within +/-1.00 : {within_100:.1f} percent")


#
================================================================================
# 9. CREATE AND SAVE THE PLOT
#
================================================================================
print_section(9, "PREDICTED-VERSUS-ACTUAL PURITY PLOT")

output_directory = get_output_directory()

plot_min = min(float(np.min(y_test)), float(np.min(y_pred))) - 0.5
plot_max = max(float(np.max(y_test)), float(np.max(y_pred))) + 0.5

fig, ax = plt.subplots(figsize=(9, 6))

ax.scatter(
    y_test,
    y_pred,
    alpha=0.78,
    edgecolors="black",
    linewidths=0.4,
    label="Independent test observations",
)

ax.plot(
    [plot_min, plot_max],
    [plot_min, plot_max],
    linestyle="--",
    linewidth=2.0,
    label="Ideal prediction: predicted = actual",
)

ax.set_xlabel("Actual overhead purity (mol %)")
ax.set_ylabel("Predicted overhead purity (mol %)")
ax.set_title("Predicted versus Actual Distillation Overhead Purity")
ax.set_xlim(plot_min, plot_max)
ax.set_ylim(plot_min, plot_max)
ax.grid(True, alpha=0.30)
```

```
ax.legend()

# Add the performance measures to the plot.
statistics_text = (
    f"$R^2$ = {r2:.4f}\n"
    f"MAE = {mae:.4f} percentage points\n"
    f"RMSE = {rmse:.4f} percentage points"
)

ax.text(
    0.04,
    0.96,
    statistics_text,
    transform=ax.transAxes,
    verticalalignment="top",
    bbox={"boxstyle": "round", "alpha": 0.85},
)

fig.tight_layout()

# Save before displaying so a permanent PNG always exists.
# If the previous PNG is open and locked, a timestamped file is used.
plot_path = save_plot_with_fallback(
    figure=fig,
    output_directory=output_directory,
    preferred_filename="Case_Study_2_Predicted_vs_Actual_Purity.png",
)

print(f"Plot saved successfully    : {plot_path}")
print(f"Matplotlib backend         : {plt.get_backend()}")


#
================================================================================
# 10. PLOT DESCRIPTION AND INTERPRETATION
#
================================================================================
print_section(10, "PLOT DESCRIPTION AND INTERPRETATION")

print(
    f"""
The horizontal axis contains the actual overhead-purity values and the
vertical
axis contains the ANN-predicted values for {len(y_test)} independent test
cases.

The dashed 45-degree line represents perfect agreement between predicted and
actual purity. Points close to this line indicate accurate predictions. Points
above the line represent overprediction, and points below it represent
```

```
underprediction.

For this run, R-squared is {r2:.4f}, MAE is {mae:.4f} percentage points, and
{within_050:.1f}% of predictions lie within +/-0.50 percentage point of the
actual purity. The mean prediction bias is {bias:+.4f} percentage points.
""".strip()
)


#
================================================================================
# 11. ENGINEERING IMPLICATIONS
#
================================================================================
print_section(11, "IMPLICATIONS FOR COMPONENT DISTILLATION")

print(
    f"""
1. Reflux-ratio management
   The model can estimate how changes in reflux ratio affect overhead purity
   before an operator changes the reflux set point.

2. Feed-composition compensation
   Feed-forward action may be taken when the light-component fraction changes,
   reducing the risk of off-specification overhead product.

3. Tray-efficiency monitoring
   Persistent differences between measured and predicted purity may indicate
   fouling, weeping, entrainment, damaged internals, or analyser problems.

4. Pressure effects
   The model represents the influence of pressure on vapour-liquid equilibrium
   and relative volatility.

5. Feed thermal condition
   Feed temperature influences feed quality and the internal vapour and liquid
   traffic within the column.

6. Soft-sensor application
   With an MAE of {mae:.4f} percentage points in this educational case, the
ANN
   demonstrates how overhead composition may be estimated between laboratory
   samples or analyser updates.

7. Energy-quality optimization
   The prediction model may be incorporated into an optimization system that
   satisfies product purity while reducing reflux, reboiler duty, and cooling
   demand.
```

```
8. Plant-implementation limitations
   A real application requires representative plant data, analyser and
   historian synchronization, validation, uncertainty analysis, periodic
   retraining, and formal management of change.
""".strip()
)


#
# ============================================================================
# 12. FINAL PROGRAM SUMMARY
#
# ============================================================================
print_section(12, "FINAL PROGRAM SUMMARY")

print("1. Generated 300 synthetic distillation observations.")
print("2. Split the data into training and independent test datasets.")
print("3. Scaled the inputs and trained an ANN regression model.")
print("4. Printed ten actual-versus-predicted purity results.")
print("5. Printed MAE, RMSE, R-squared, bias, and tolerance statistics.")
print("6. Created and saved the predicted-versus-actual purity plot.")
print("7. Printed the plot interpretation and engineering implications.")


#
# ============================================================================
# 13. DISPLAY THE PLOT
#
# ============================================================================
print_section(13, "DISPLAY OF THE PLOT")

# This is the corrected display logic. The original test searched for the word
# "agg", which incorrectly rejected interactive backends such as TkAgg and
# QtAgg. Here, only genuinely non-interactive backends are treated as
headless.
backend_key = str(plt.get_backend()).lower()
non_interactive_backends = {
    "agg",
    "pdf",
    "pgf",
    "ps",
    "svg",
    "template",
    "cairo",
}

if backend_key not in non_interactive_backends:
    print("Opening the Matplotlib plot window.")
    print("Close the plot window to complete the program.")
```

```
    try:
        plt.show(block=True)
    except TypeError:
        plt.show()
    except Exception as exc:
        print(f"Matplotlib could not display the plot: {exc}")
        print("Attempting to open the saved PNG instead.")

        if not open_saved_plot(plot_path):
            print(f"Open the plot manually from: {plot_path}")

else:
    print(
        f"The backend '{plt.get_backend()}' is non-interactive. "
        "Opening the saved PNG instead."
    )

    if not open_saved_plot(plot_path):
        print(f"Open the plot manually from: {plot_path}")

plt.close(fig)

print("\nPROGRAM EXECUTION COMPLETED SUCCESSFULLY.")
print(f"Saved plot: {plot_path}")
print("=" * 78)
```

Results

[Running] python -u "c:\Users\Kayode Coker\OneDrive\Desktop\Python Codes\Case Study 2. Predicting Distillation Overhead Purity Using Machine Learning.py"

##

CASE STUDY 2: DISTILLATION OVERHEAD PURITY PREDICTION

ARTIFICIAL NEURAL NETWORK / MACHINE-LEARNING REGRESSION MODEL

##

==========

2. PROGRAM DESCRIPTION

==========

The program investigates how five important distillation operating variables

influence the purity of a light overhead product:

1. Feed temperature

2. Reflux ratio

3. Light-component fraction in the feed

4. Column operating pressure

5. Overall tray efficiency

A multilayer perceptron artificial neural network is trained using 300

simulated operating observations. The model is then evaluated on independent

test data that were not used during training.

==========

3. INPUT VARIABLES AND ENGINEERING RANGES

==========

Feed temperature : 120-180 degC

Reflux ratio : 1.20-3.50 mol reflux/mol distillate

Light component in feed : 0.10-0.35 mole fraction

Column pressure : 120-180 kPa absolute

Overall tray efficiency : 45-75 percent

Predicted output : Overhead light-component purity, mol percent

==========

4. SYNTHETIC DISTILLATION DATA GENERATION

==========

Random seed used : 42

Number of observations : 300

Generated-data summary:

Feed temperature Minimum = 120.442 degC | Maximum = 179.543 degC | Mean = 149.199 degC | Std. dev. = 17.389 degC

Reflux ratio Minimum = 1.212 mol/mol | Maximum = 3.498 mol/mol | Mean = 2.362 mol/mol | Std. dev. = 0.653 mol/mol

Light component in feed Minimum = 0.100 mole fraction | Maximum = 0.350 mole fraction | Mean = 0.224 mole fraction | Std. dev. = 0.074 mole fraction

Column pressure Minimum = 120.031 kPa abs | Maximum = 179.965 kPa abs | Mean = 150.673 kPa abs | Std. dev. = 17.499 kPa abs

Overall tray efficiency Minimum = 45.032 % | Maximum = 74.900 % | Mean = 59.935 % | Std. dev. = 8.792 %

Overhead purity Minimum = 87.986 mol % | Maximum = 97.160 mol % | Mean = 92.900 mol % | Std. dev. = 1.617 mol %

First five generated observations:

Row | Feed T | Reflux | Feed light fraction | Pressure | Tray eff. | Purity

---

1 | 166.44 | 2.992 | 0.2194 | 148.10 | 61.99 | 94.872%

2 | 146.33 | 1.509 | 0.2904 | 135.57 | 61.53 | 92.021%

3 | 171.52 | 2.433 | 0.3258 | 122.71 | 69.86 | 95.574%

4 | 161.84 | 2.383 | 0.2802 | 148.89 | 66.32 | 94.246%

5 | 125.65 | 3.172 | 0.3408 | 177.56 | 45.80 | 92.745%

==========

5. TRAINING AND TEST DATA SPLIT

==========

Training observations : 240

Test observations : 60

==========

6. NEURAL-NETWORK MODEL DEFINITION AND TRAINING

==========

Model type : Multilayer Perceptron Regressor

Hidden layers : 20 neurons and 20 neurons

Activation function : ReLU

Optimizer : Adam

Maximum iterations : 5000

Feature preprocessing : StandardScaler

Training completed.

Iterations performed : 2193

Final training loss : 0.051634

Best validation R-squared : 0.880451

Convergence status : No convergence warning

==========

7. TEST-SET PURITY PREDICTIONS

==========

First ten independent test predictions:

Case | Actual purity | Predicted purity | Prediction error

---

1 | 92.095% | 91.432% | -0.663 percentage points

2 | 93.488% | 94.004% | +0.516 percentage points

3 | 92.841% | 92.812% | -0.029 percentage points

4 | 90.933% | 90.863% | -0.070 percentage points

5 | 92.411% | 92.239% | -0.172 percentage points

6 | 93.243% | 94.397% | +1.154 percentage points

7 | 92.588% | 92.195% | -0.393 percentage points

8 | 96.679% | 97.463% | +0.784 percentage points

9 | 92.103% | 91.950% | -0.153 percentage points

10 | 91.608% | 91.272% | -0.336 percentage points

==========

8. MODEL PERFORMANCE RESULTS

==========

Mean Absolute Error : 0.3181 percentage points

Root Mean Squared Error : 0.3936 percentage points

R-squared score : 0.9324

Mean prediction bias : +0.0180 percentage points

Maximum absolute error : 1.1538 percentage points

Predictions within +/-0.25 : 46.7 percent

Predictions within +/-0.50 : 76.7 percent

Predictions within +/-1.00 : 98.3 percent

==========

9. PREDICTED-VERSUS-ACTUAL PURITY PLOT

==========

Plot saved successfully : C:\Users\Kayode Coker\OneDrive\Desktop\Python Codes\Case_Study_2_Predicted_vs_Actual_Purity.png

Matplotlib backend : tkagg

==========

10. PLOT DESCRIPTION AND INTERPRETATION

==========

The horizontal axis contains the actual overhead-purity values and the vertical

axis contains the ANN-predicted values for 60 independent test cases.

The dashed 45-degree line represents perfect agreement between predicted and

actual purity. Points close to this line indicate accurate predictions. Points

above the line represent overprediction, and points below it represent

underprediction.

For this run, R-squared is 0.9324, MAE is 0.3181 percentage points, and

76.7% of predictions lie within +/-0.50 percentage point of the

actual purity. The mean prediction bias is +0.0180 percentage points.

==========

11. IMPLICATIONS FOR COMPONENT DISTILLATION

==========

1. Reflux-ratio management

The model can estimate how changes in reflux ratio affect overhead purity

before an operator changes the reflux set point.

2. Feed-composition compensation

Feed-forward action may be taken when the light-component fraction changes,

reducing the risk of off-specification overhead product.

3. Tray-efficiency monitoring

Persistent differences between measured and predicted purity may indicate

fouling, weeping, entrainment, damaged internals, or analyser problems.

4. Pressure effects

The model represents the influence of pressure on vapour-liquid equilibrium

and relative volatility.

5. Feed thermal condition

Feed temperature influences feed quality and the internal vapour and liquid

traffic within the column.

6. Soft-sensor application

With an MAE of 0.3181 percentage points in this educational case, the ANN

demonstrates how overhead composition may be estimated between laboratory

samples or analyser updates.

7. Energy-quality optimization

The prediction model may be incorporated into an optimization system that

satisfies product purity while reducing reflux, reboiler duty, and cooling

demand.

8. Plant-implementation limitations

A real application requires representative plant data, analyser and

historian synchronization, validation, uncertainty analysis, periodic

retraining, and formal management of change.

==========

12. FINAL PROGRAM SUMMARY

==========

1. Generated 300 synthetic distillation observations.

2. Split the data into training and independent test datasets.

3. Scaled the inputs and trained an ANN regression model.

4. Printed ten actual-versus-predicted purity results.

5. Printed MAE, RMSE, R-squared, bias, and tolerance statistics.

6. Created and saved the predicted-versus-actual purity plot.

7. Printed the plot interpretation and engineering implications.

==========

13. DISPLAY OF THE PLOT

==========

Opening the Matplotlib plot window.

Close the plot window to complete the program.

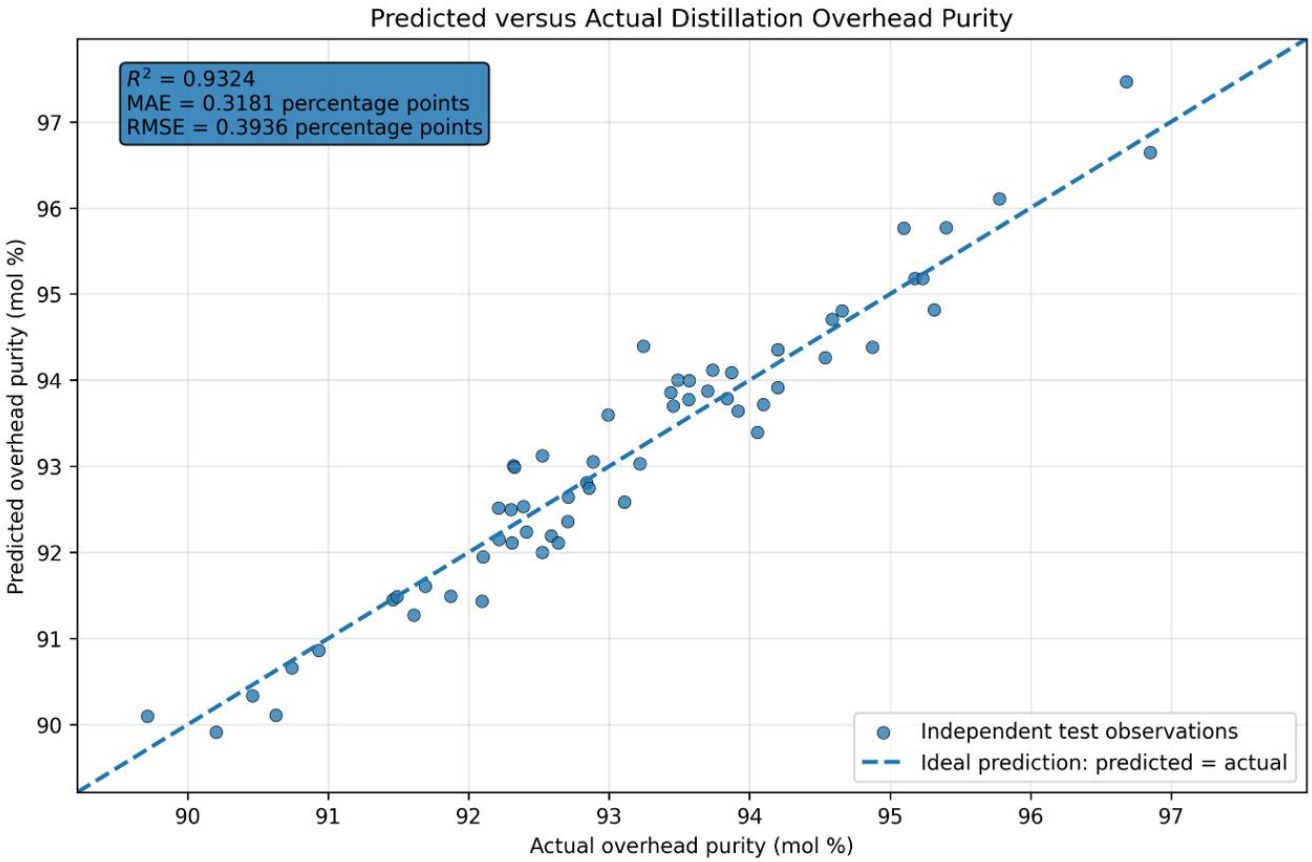
PROGRAM EXECUTION COMPLETED SUCCESSFULLY.

Saved plot: C:\Users\Kayode Coker\OneDrive\Desktop\Python Codes\Case_Study_2_Predicted_vs_Actual_Purity.png

==========

[Done] exited with code=0 in 8.404 seconds

A Plot


Predicted versus Actual Distillation Overhead Purity
R^2 = 0.9324
MAE = 0.3181 percentage points
RMSE = 0.3936 percentage points
Predicted overhead purity (mol %)
Actual overhead purity (mol %)
Independent test observations
Ideal prediction: predicted = actual
90
91
92
93
94
95
96
97