Data Generation

We'll generate a synthetic dataset that simulates the complex process

```
pip install seaborn
```

```
Collecting seaborn
  Using cached seaborn-0.13.2-py3-none-any.whl.metadata (5.4 kB)
Requirement already satisfied: numpy!=1.24.0,>=1.20 in c:\users\yoga\
appdata\local\programs\python\python39\lib\site-packages (from
seaborn) (1.26.4)
Requirement already satisfied: pandas>=1.2 in c:\users\yoga\appdata\
local\programs\python\python39\lib\site-packages (from seaborn)
(2.2.3)
Requirement already satisfied: matplotlib!=3.6.1,>=3.4 in c:\users\
yoga\appdata\local\programs\python\python39\lib\site-packages (from
seaborn) (3.8.4)
Requirement already satisfied: contourpy>=1.0.1 in c:\users\yoga\
appdata\local\programs\python\python39\lib\site-packages (from
matplotlib!=3.6.1,>=3.4->seaborn) (1.2.1)
Requirement already satisfied: cycler>=0.10 in c:\users\yoga\appdata\
local\programs\python\python39\lib\site-packages (from matplotlib!
=3.6.1,>=3.4->seaborn) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in c:\users\yoga\
appdata\local\programs\python\python39\lib\site-packages (from
matplotlib!=3.6.1,>=3.4->seaborn) (4.51.0)
Requirement already satisfied: kiwisolver>=1.3.1 in c:\users\yoga\
appdata\local\programs\python\python39\lib\site-packages (from
matplotlib!=3.6.1,>=3.4->seaborn) (1.4.5)
Requirement already satisfied: packaging>=20.0 in c:\users\yoga\
appdata\roaming\python\python39\site-packages (from matplotlib!
=3.6.1,>=3.4->seaborn) (23.2)
Requirement already satisfied: pillow>=8 in c:\users\yoga\appdata\
local\programs\python\python39\lib\site-packages (from matplotlib!
=3.6.1,>=3.4->seaborn) (10.3.0)
Requirement already satisfied: pyparsing>=2.3.1 in c:\users\yoga\
appdata\local\programs\python\python39\lib\site-packages (from
matplotlib!=3.6.1,>=3.4->seaborn) (3.1.2)
Requirement already satisfied: python-dateutil>=2.7 in c:\users\yoga\
appdata\roaming\python\python39\site-packages (from matplotlib!
=3.6.1,>=3.4->seaborn) (2.8.2)
Requirement already satisfied: importlib-resources>=3.2.0 in c:\users\
yoga\appdata\local\programs\python\python39\lib\site-packages (from
matplotlib!=3.6.1,>=3.4->seaborn) (6.4.0)
Requirement already satisfied: pytz>=2020.1 in c:\users\yoga\appdata\
local\programs\python\python39\lib\site-packages (from pandas>=1.2-
>seaborn) (2024.2)
Requirement already satisfied: tzdata>=2022.7 in c:\users\yoga\
appdata\local\programs\python\python39\lib\site-packages (from
```

```
pandas>=1.2->seaborn) (2024.2)
Requirement already satisfied: zipp>=3.1.0 in c:\users\yoga\appdata\
roaming\python\python39\site-packages (from importlib-
resources>=3.2.0->matplotlib!=3.6.1,>=3.4->seaborn) (3.17.0)
Requirement already satisfied: six>=1.5 in c:\users\yoga\appdata\
roaming\python\python39\site-packages (from python-dateutil>=2.7-
>matplotlib!=3.6.1,>=3.4->seaborn) (1.16.0)
Downloading seaborn-0.13.2-py3-none-any.whl (294 kB)
Installing collected packages: seaborn
Successfully installed seaborn-0.13.2
Note: you may need to restart the kernel to use updated packages.


[notice] A new release of pip is available: 24.2 -> 24.3.1
[notice] To update, run: python.exe -m pip install --upgrade pip
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

Set Random Seed To ensure reproducibility.

```
np.random.seed(42)
```

Generate Input Variables Let's assume we have 10 input variables.

```
n_samples = 1000
n_features = 10

# Generate random input features between 0 and 10
X = np.random.uniform(0, 10, size=(n_samples, n_features))
```

Assign Coefficients Randomly assign coefficients β for each term.

```
beta_linear = np.random.uniform(-5, 5, size=n_features)

# Coefficients for interaction terms (n_features choose 2)
n_interactions = int(n_features * (n_features - 1) / 2)
beta_interaction = np.random.uniform(-1, 1, size=n_interactions)

# Linear combination of inputs
linear_terms = np.dot(X, beta_linear)

# Initialize interaction terms array
interaction_terms = np.zeros(n_samples)

# Index to keep track of interaction coefficients
idx = 0
```

```python
# Loop over all unique pairs of features
for i in range(n_features):
    for j in range(i + 1, n_features):
        # Add interaction term
        interaction_terms += beta_interaction[idx] * X[:, i] * X[:, j]
        idx += 1

epsilon = np.random.normal(0, 1, n_samples)

# Total output
y = beta_linear[0] + linear_terms + interaction_terms + epsilon

# Feature names
columns = [f'x{i+1}' for i in range(n_features)]

# Combine features and target into a DataFrame
data = pd.DataFrame(X, columns=columns)
data['y'] = y

print(data.head(5))
```

```
         x1        x2        x3        x4        x5        x6
x7  \
0  3.745401  9.507143  7.319939  5.986585  1.560186  1.559945
0.580836
1  0.205845  9.699099  8.324426  2.123391  1.818250  1.834045
3.042422
2  6.118529  1.394939  2.921446  3.663618  4.560700  7.851760
1.996738
3  6.075449  1.705241  0.650516  9.488855  9.656320  8.083973
3.046138
4  1.220382  4.951769  0.343885  9.093204  2.587800  6.625223
3.117111

         x8        x9       x10           y
0  8.661761  6.011150  7.080726  178.654324
1  5.247564  4.319450  2.912291   85.918166
2  5.142344  5.924146  0.464504   76.613757
3  0.976721  6.842330  4.401525  -94.338920
4  5.200680  5.467103  1.848545  -38.892191
```

1. Data Preprocessing Handling Missing Values Our synthetic data has no missing values, but in real scenarios, you'd handle them here.

Feature Scaling Neural networks converge faster with scaled features.

```python
from sklearn.preprocessing import StandardScaler

# Initialize scaler
scaler = StandardScaler()
```

```python
# Fit and transform features
X_scaled = scaler.fit_transform(data.drop('y', axis=1))

# Update DataFrame
data_scaled = pd.DataFrame(X_scaled, columns=columns)
data_scaled['y'] = data['y']
```

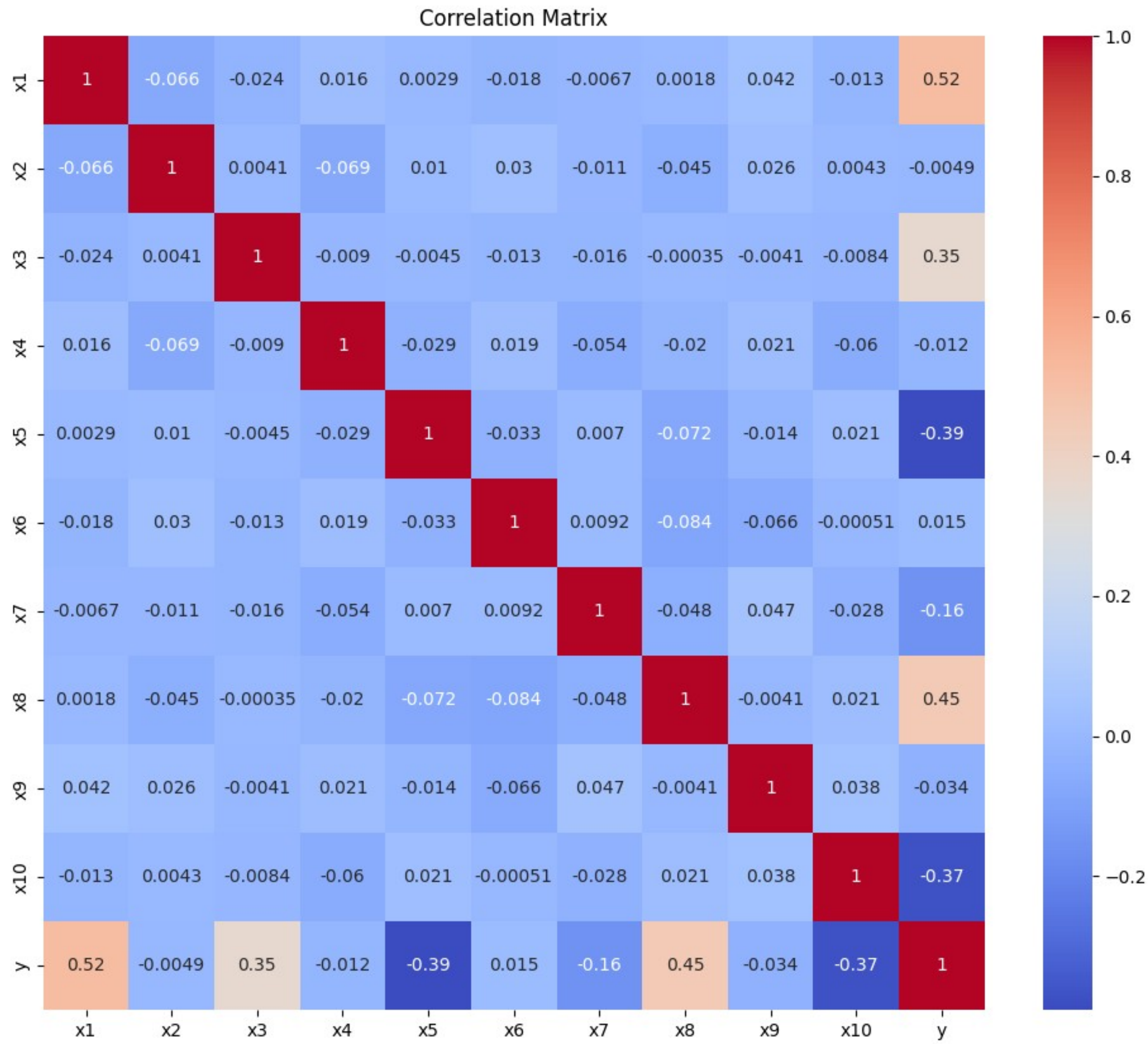
Check Scaled Data

```python
print(data_scaled.head())
```

```
         x1        x2        x3        x4        x5        x6
x7  \
0 -0.363066  1.547478  0.821234  0.361551 -1.177298 -1.196034 -
1.532007
1 -1.603520  1.612962  1.167403 -1.011189 -1.087152 -1.101643 -
0.670775
2  0.468607 -1.219938 -0.694585 -0.463887 -0.129166  0.970644 -
1.036627
3  0.453509 -1.114080 -1.477198  1.606041  1.650825  1.050610 -
0.669475
4 -1.247970 -0.006553 -1.582870  1.465451 -0.818335  0.548268 -
0.644644

         x8        x9       x10           y
0  1.302662  0.337660  0.761895  178.654324
1  0.121575 -0.252714 -0.690874   85.918166
2  0.085176  0.307297 -1.543968   76.613757
3 -1.355854  0.627727 -0.171851  -94.338920
4  0.105356  0.147797 -1.061607  -38.892191
```

1. Feature Selection Correlation Analysis

```python
# Compute correlation matrix
corr_matrix = data_scaled.corr()

# Plot heatmap
plt.figure(figsize=(12, 10))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm')
plt.title('Correlation Matrix')
plt.show()
```



Explanation:

Purpose: Identify highly correlated features. Action: We can decide to remove features with high multicollinearity. Select Features For simplicity, we'll proceed with all features, but in practice, you might select a subset.

1. Feature Engineering Creating Interaction Features Using DOE principles to include interaction terms.

```
from sklearn.preprocessing import PolynomialFeatures

# Create polynomial features (degree=2 includes interactions)
poly = PolynomialFeatures(degree=2, interaction_only=True,
include_bias=False)

# Fit and transform features
X_poly = poly.fit_transform(data_scaled.drop('y', axis=1))
```

```
# Get feature names
feature_names = poly.get_feature_names_out(columns)
```

Explanation PolynomialFeatures: degree=2: Includes all combinations of features up to degree 2. interaction_only=True: Excludes squared terms, focusing on interactions.

Update DataFrame with Interaction Features

```
# Create DataFrame with interaction features
data_poly = pd.DataFrame(X_poly, columns=feature_names)
data_poly['y'] = data_scaled['y']
```

1. Neural Network Model Implementation Libraries

Split Data into Training and Testing Sets

```
from sklearn.model_selection import train_test_split

# Features and target variable
X = data_poly.drop('y', axis=1).values
y = data_poly['y'].values

# Split the data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
```

Define the Neural Network Architecture

```
# Input dimension after adding interaction terms
input_dim = X_train.shape[1]

# Initialize the model
model = Sequential()

# Input layer
model.add(Dense(128, input_dim=input_dim, activation='relu'))

# Hidden layers
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.3))  # Dropout for regularization
model.add(Dense(128, activation='relu'))

# Output layer
model.add(Dense(1, activation='linear'))
```

```
WARNING:tensorflow:From c:\Users\Yoga\AppData\Local\Programs\Python\Python39\lib\site-packages\keras\src\backend.py:873: The name tf.get_default_graph is deprecated. Please use tf.compat.v1.get_default_graph instead.
```

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 dense (Dense)               (None, 128)               7168      
                                                                 
 dense_1 (Dense)             (None, 256)               33024     
                                                                 
 dropout (Dropout)           (None, 256)               0         
                                                                 
 dense_2 (Dense)             (None, 128)               32896     
                                                                 
 dense_3 (Dense)             (None, 1)                 129       
                                                                 
=================================================================
Total params: 73217 (286.00 KB)
Trainable params: 73217 (286.00 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
model.compile(
    optimizer='adam',
    loss='mean_squared_error',
    metrics=['mean_absolute_error']
)
```

```
WARNING:tensorflow:From c:\Users\Yoga\AppData\Local\Programs\Python\Python39\lib\site-packages\keras\src\optimizers\__init__.py:309: The name tf.train.Optimizer is deprecated. Please use tf.compat.v1.train.Optimizer instead.
```

Explanation:

Optimizer: Adam optimizer adapts learning rates during training. Loss Function: Mean Squared Error for regression tasks. Metrics: Mean Absolute Error to evaluate model performance.

1. Model Training Set Up Early Stopping

```
from tensorflow.keras.callbacks import EarlyStopping

early_stop = EarlyStopping(
    monitor='val_loss',
```

```
    patience=10,
    restore_best_weights=True
)
```

Train the Model

```
history = model.fit(
    X_train, y_train,
    validation_split=0.2,
    epochs=100,
    batch_size=32,
    callbacks=[early_stop],
    verbose=1
)
```

```
Epoch 1/100
WARNING:tensorflow:From c:\Users\Yoga\AppData\Local\Programs\Python\
Python39\lib\site-packages\keras\src\utils\tf_utils.py:492: The name
tf.ragged.RaggedTensorValue is deprecated. Please use
tf.compat.v1.ragged.RaggedTensorValue instead.

WARNING:tensorflow:From c:\Users\Yoga\AppData\Local\Programs\Python\
Python39\lib\site-packages\keras\src\engine\base_layer_utils.py:384:
The name tf.executing_eagerly_outside_functions is deprecated. Please
use tf.compat.v1.executing_eagerly_outside_functions instead.

20/20 [==============================] - 1s 9ms/step - loss: 8648.7988
- mean_absolute_error: 73.9305 - val_loss: 6188.2852 -
val_mean_absolute_error: 59.7385
Epoch 2/100
20/20 [==============================] - 0s 3ms/step - loss: 8036.2563
- mean_absolute_error: 71.0626 - val_loss: 5332.3345 -
val_mean_absolute_error: 55.5473
Epoch 3/100
20/20 [==============================] - 0s 3ms/step - loss: 4886.9434
- mean_absolute_error: 53.6557 - val_loss: 2312.6646 -
val_mean_absolute_error: 36.2888
Epoch 4/100
20/20 [==============================] - 0s 3ms/step - loss: 1039.7753
- mean_absolute_error: 24.2208 - val_loss: 453.5901 -
val_mean_absolute_error: 16.8014
Epoch 5/100
20/20 [==============================] - 0s 3ms/step - loss: 292.7521
- mean_absolute_error: 13.1802 - val_loss: 213.3192 -
val_mean_absolute_error: 11.3393
Epoch 6/100
20/20 [==============================] - 0s 3ms/step - loss: 181.5477
- mean_absolute_error: 10.2701 - val_loss: 151.1905 -
val_mean_absolute_error: 9.8167
```

```
Epoch 7/100
20/20 [==============================] - 0s 3ms/step - loss: 167.8574
- mean_absolute_error: 10.1185 - val_loss: 151.0729 -
val_mean_absolute_error: 9.5091
Epoch 8/100
20/20 [==============================] - 0s 3ms/step - loss: 136.5610
- mean_absolute_error: 9.0322 - val_loss: 117.6982 -
val_mean_absolute_error: 8.3595
Epoch 9/100
20/20 [==============================] - 0s 3ms/step - loss: 133.2912
- mean_absolute_error: 8.8254 - val_loss: 111.0519 -
val_mean_absolute_error: 7.9624
Epoch 10/100
20/20 [==============================] - 0s 3ms/step - loss: 114.8808
- mean_absolute_error: 8.2795 - val_loss: 115.4543 -
val_mean_absolute_error: 8.3367
Epoch 11/100
20/20 [==============================] - 0s 3ms/step - loss: 123.8664
- mean_absolute_error: 8.6454 - val_loss: 102.8765 -
val_mean_absolute_error: 7.8222
Epoch 12/100
20/20 [==============================] - 0s 3ms/step - loss: 106.8634
- mean_absolute_error: 7.9053 - val_loss: 94.6192 -
val_mean_absolute_error: 7.4617
Epoch 13/100
20/20 [==============================] - 0s 3ms/step - loss: 100.8406
- mean_absolute_error: 7.5977 - val_loss: 98.9632 -
val_mean_absolute_error: 7.5522
Epoch 14/100
20/20 [==============================] - 0s 3ms/step - loss: 108.3899
- mean_absolute_error: 7.7403 - val_loss: 102.0158 -
val_mean_absolute_error: 7.5881
Epoch 15/100
20/20 [==============================] - 0s 3ms/step - loss: 106.1804
- mean_absolute_error: 7.8147 - val_loss: 101.9140 -
val_mean_absolute_error: 7.7441
Epoch 16/100
20/20 [==============================] - 0s 3ms/step - loss: 102.3587
- mean_absolute_error: 7.4590 - val_loss: 79.9470 -
val_mean_absolute_error: 6.7715
Epoch 17/100
20/20 [==============================] - 0s 3ms/step - loss: 80.9630 -
mean_absolute_error: 6.8022 - val_loss: 81.3400 -
val_mean_absolute_error: 6.8737
Epoch 18/100
20/20 [==============================] - 0s 3ms/step - loss: 84.8275 -
mean_absolute_error: 7.0133 - val_loss: 85.9735 -
val_mean_absolute_error: 7.0195
Epoch 19/100
```

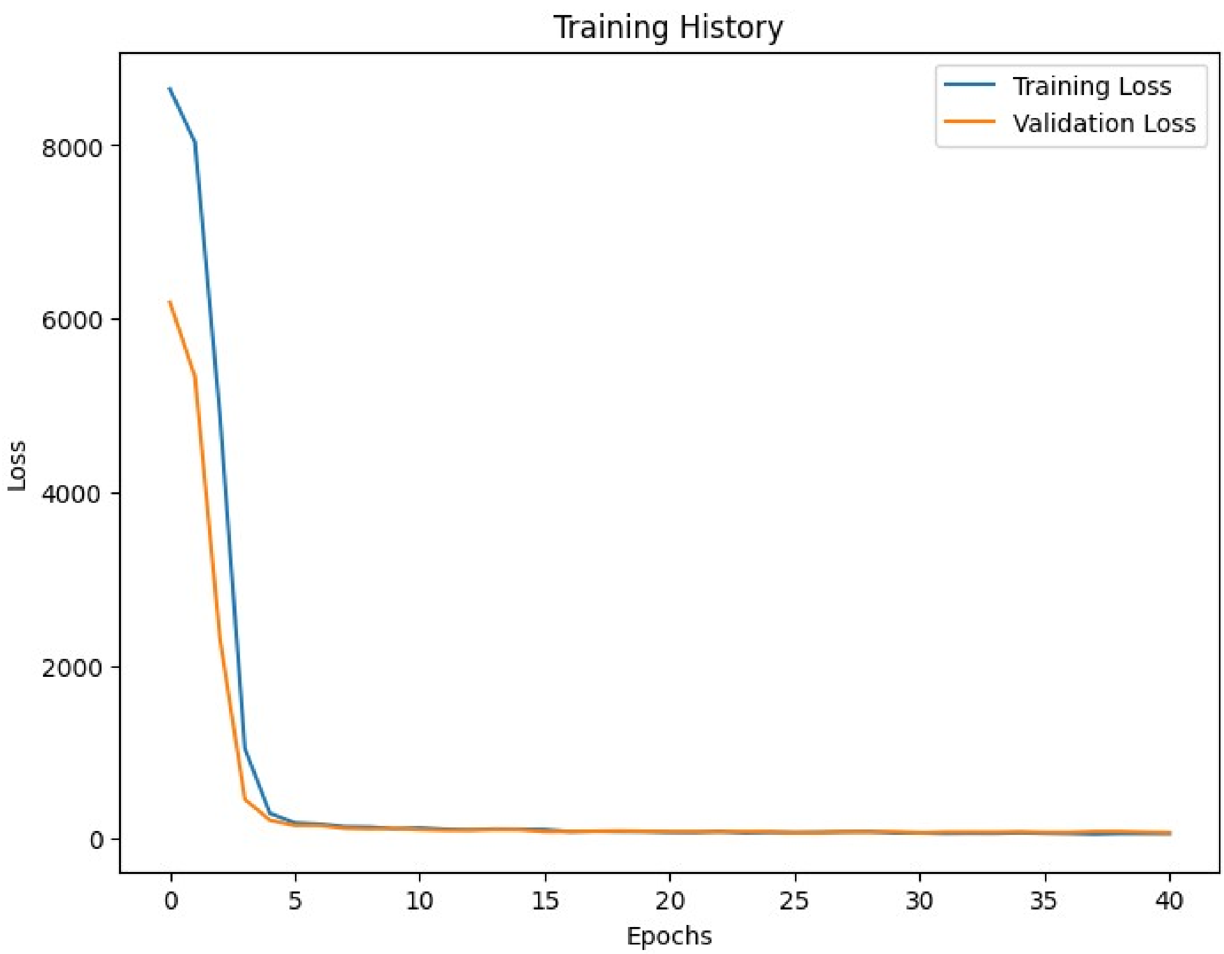
```
20/20 [==============================] - 0s 3ms/step - loss: 81.6997 -
mean_absolute_error: 6.6208 - val_loss: 89.1710 -
val_mean_absolute_error: 7.2989
Epoch 20/100
20/20 [==============================] - 0s 3ms/step - loss: 80.7513 -
mean_absolute_error: 6.7058 - val_loss: 84.8360 -
val_mean_absolute_error: 7.0509
Epoch 21/100
20/20 [==============================] - 0s 3ms/step - loss: 74.2774 -
mean_absolute_error: 6.5526 - val_loss: 79.8263 -
val_mean_absolute_error: 6.7771
Epoch 22/100
20/20 [==============================] - 0s 3ms/step - loss: 72.4148 -
mean_absolute_error: 6.4641 - val_loss: 79.8499 -
val_mean_absolute_error: 6.8056
Epoch 23/100
20/20 [==============================] - 0s 3ms/step - loss: 82.6289 -
mean_absolute_error: 6.5561 - val_loss: 77.0856 -
val_mean_absolute_error: 6.5755
Epoch 24/100
20/20 [==============================] - 0s 3ms/step - loss: 67.6780 -
mean_absolute_error: 6.1833 - val_loss: 80.7937 -
val_mean_absolute_error: 6.7611
Epoch 25/100
20/20 [==============================] - 0s 2ms/step - loss: 73.6766 -
mean_absolute_error: 6.2056 - val_loss: 80.0893 -
val_mean_absolute_error: 6.8974
Epoch 26/100
20/20 [==============================] - 0s 3ms/step - loss: 71.0776 -
mean_absolute_error: 6.3182 - val_loss: 66.9290 -
val_mean_absolute_error: 6.4197
Epoch 27/100
20/20 [==============================] - 0s 3ms/step - loss: 73.5757 -
mean_absolute_error: 6.4198 - val_loss: 69.1794 -
val_mean_absolute_error: 6.3932
Epoch 28/100
20/20 [==============================] - 0s 3ms/step - loss: 79.4177 -
mean_absolute_error: 6.4063 - val_loss: 77.1633 -
val_mean_absolute_error: 6.6143
Epoch 29/100
20/20 [==============================] - 0s 3ms/step - loss: 81.6471 -
mean_absolute_error: 6.6154 - val_loss: 76.0644 -
val_mean_absolute_error: 6.7377
Epoch 30/100
20/20 [==============================] - 0s 2ms/step - loss: 66.4865 -
mean_absolute_error: 6.1363 - val_loss: 78.9291 -
val_mean_absolute_error: 6.9183
Epoch 31/100
20/20 [==============================] - 0s 3ms/step - loss: 66.5892 -
```

```
mean_absolute_error: 6.1937 - val_loss: 66.3639 -
val_mean_absolute_error: 6.2720
Epoch 32/100
20/20 [==============================] - 0s 2ms/step - loss: 60.3523 -
mean_absolute_error: 5.7764 - val_loss: 74.7099 -
val_mean_absolute_error: 6.6413
Epoch 33/100
20/20 [==============================] - 0s 2ms/step - loss: 63.0094 -
mean_absolute_error: 6.0050 - val_loss: 75.5275 -
val_mean_absolute_error: 6.6400
Epoch 34/100
20/20 [==============================] - 0s 2ms/step - loss: 61.7266 -
mean_absolute_error: 5.7045 - val_loss: 73.9673 -
val_mean_absolute_error: 6.5526
Epoch 35/100
20/20 [==============================] - 0s 2ms/step - loss: 68.6619 -
mean_absolute_error: 6.1549 - val_loss: 77.3238 -
val_mean_absolute_error: 6.5474
Epoch 36/100
20/20 [==============================] - 0s 2ms/step - loss: 62.0706 -
mean_absolute_error: 5.8968 - val_loss: 71.5218 -
val_mean_absolute_error: 6.5054
Epoch 37/100
20/20 [==============================] - 0s 2ms/step - loss: 57.7333 -
mean_absolute_error: 5.6403 - val_loss: 70.3059 -
val_mean_absolute_error: 6.5438
Epoch 38/100
20/20 [==============================] - 0s 3ms/step - loss: 50.6825 -
mean_absolute_error: 5.3124 - val_loss: 82.3516 -
val_mean_absolute_error: 7.0180
Epoch 39/100
20/20 [==============================] - 0s 3ms/step - loss: 57.7500 -
mean_absolute_error: 5.6879 - val_loss: 81.5721 -
val_mean_absolute_error: 6.9372
Epoch 40/100
20/20 [==============================] - 0s 3ms/step - loss: 57.5124 -
mean_absolute_error: 5.6813 - val_loss: 74.0030 -
val_mean_absolute_error: 6.8042
Epoch 41/100
20/20 [==============================] - 0s 3ms/step - loss: 56.5801 -
mean_absolute_error: 5.6828 - val_loss: 68.1339 -
val_mean_absolute_error: 6.5257
```

Plot Training History

```
# Plot loss over epochs
plt.figure(figsize=(8, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
```

```
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training History')
plt.show()
```



Explanation:

Early Stopping: Prevents overfitting by stopping training when validation loss doesn't improve. Batch Size: Set to 32 for efficient computation. Validation Split: 20% of training data used for validation

1. Model Evaluation Evaluate on Test Data

```
test_loss, test_mae = model.evaluate(X_test, y_test, verbose=0)
print(f'Test MSE: {test_loss:.4f}')
print(f'Test MAE: {test_mae:.4f}')

Test MSE: 73.7218
Test MAE: 6.5903
```

Predict on Test Data

```
y_pred = model.predict(X_test)
```

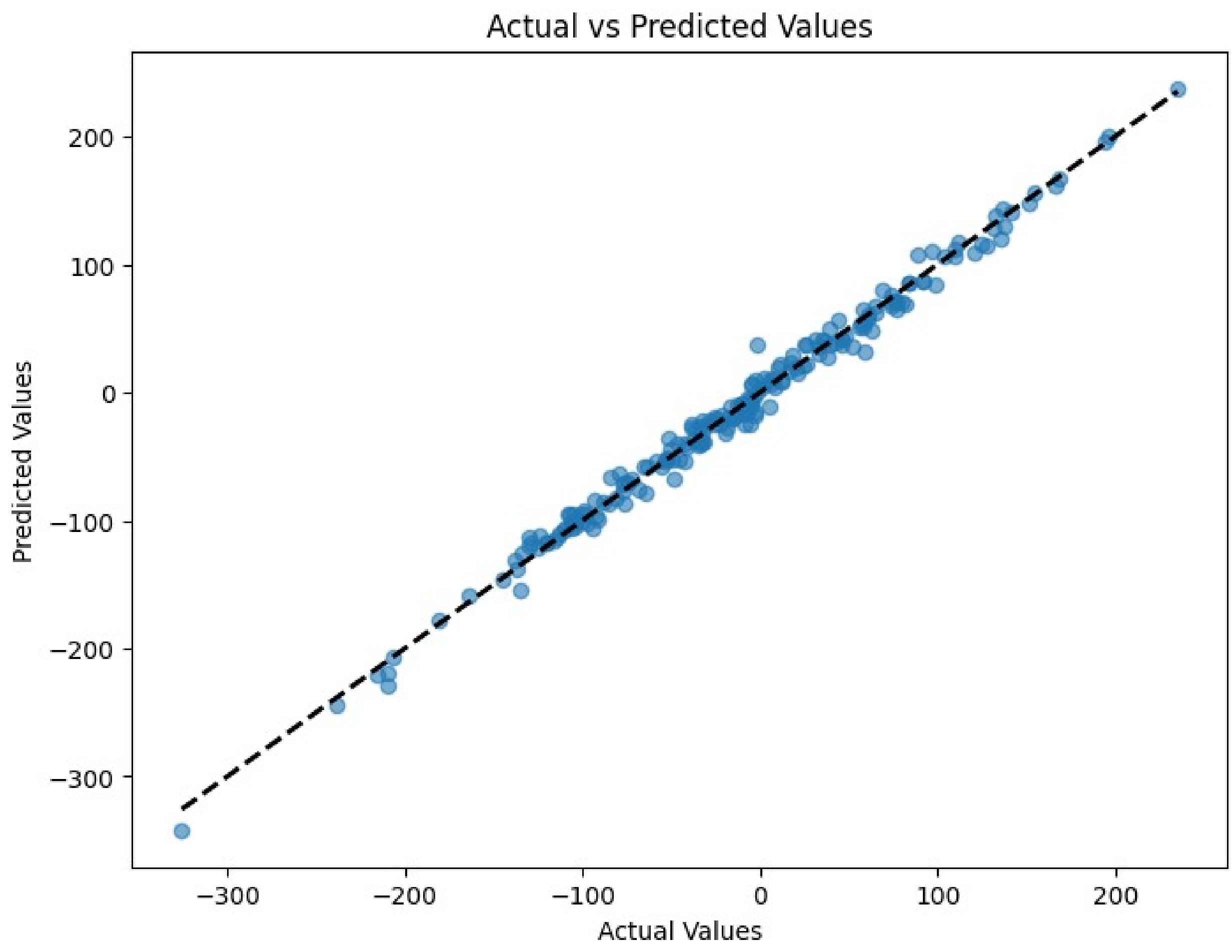
```
7/7 [==============================] - 0s 1ms/step
```

```
from sklearn.metrics import r2_score, mean_squared_error

mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f'Mean Squared Error: {mse:.4f}')
print(f'R^2 Score: {r2:.4f}')
```

```
Mean Squared Error: 73.7218
R^2 Score: 0.9906
```

Plot Predicted vs Actual Values

```
plt.figure(figsize=(8, 6))
plt.scatter(y_test, y_pred, alpha=0.6)
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Actual vs Predicted Values')
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()],
'k--', lw=2)
plt.show()
```



Explanation:

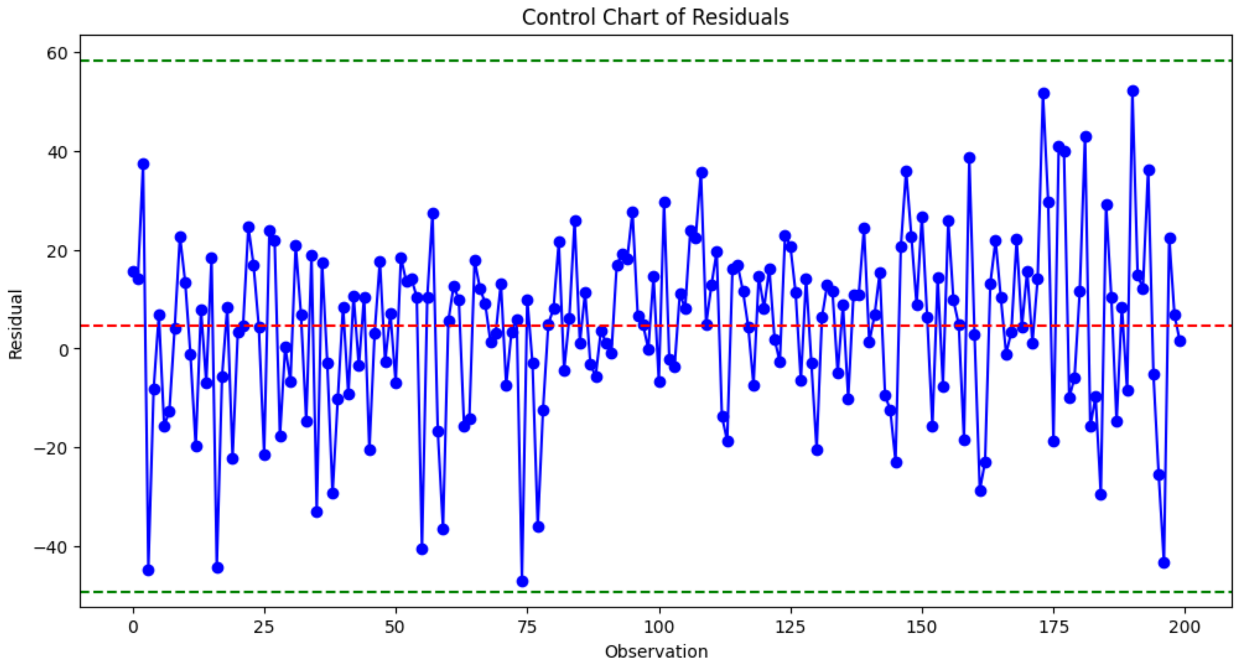
R^2 Score: Indicates how well the model explains the variability of the target variable. Mean Squared Error: Average squared difference between predicted and actual values.

1. Optimization and Conclusion Hyperparameter Tuning To improve the model, consider tuning hyperparameters:

Number of Layers and Neurons: Experiment with different architectures. Learning Rate: Adjust optimizer learning rate. Batch Size and Epochs: Modify based on convergence behavior. Activation Functions: Try different activation functions like tanh or leaky ReLU. Cross-Validation Implement k-fold cross-validation to ensure model robustness.

```
from sklearn.model_selection import KFold

# Define KFold
kfold = KFold(n_splits=5, shuffle=True, random_state=42)

# Placeholder for results
mse_scores = []
```

```
for train_index, val_index in kfold.split(X):
    # Split data
    X_train_fold, X_val_fold = X[train_index], X[val_index]
    y_train_fold, y_val_fold = y[train_index], y[val_index]

    # Build model (you might want to define a function for this)
    model_fold = Sequential()
    model_fold.add(Dense(64, input_dim=n_features, activation='relu'))
    model_fold.add(Dense(128, activation='relu'))
    model_fold.add(Dropout(0.2))
    model_fold.add(Dense(64, activation='relu'))
    model_fold.add(Dense(1, activation='linear'))
    model_fold.compile(optimizer='adam', loss='mean_squared_error')

    # Train model
    model_fold.fit(X_train_fold, y_train_fold, epochs=50,
batch_size=32, verbose=0)

    # Evaluate model
    y_val_pred = model_fold.predict(X_val_fold)
    mse_fold = mean_squared_error(y_val_fold, y_val_pred)
    mse_scores.append(mse_fold)

print(f'Cross-Validated MSE: {np.mean(mse_scores):.4f} ±
{np.std(mse_scores):.4f}')

7/7 [==============================] - 0s 1ms/step
7/7 [==============================] - 0s 1ms/step
7/7 [==============================] - 0s 1ms/step
7/7 [==============================] - 0s 1ms/step
7/7 [==============================] - 0s 1000us/step
Cross-Validated MSE: 487.9427 ± 56.5823

# Calculate residuals
residuals = y_test - y_pred.flatten()

# Plot control chart
plt.figure(figsize=(12, 6))
plt.plot(residuals, marker='o', linestyle='-', color='b')
plt.axhline(y=np.mean(residuals), color='r', linestyle='--')
plt.axhline(y=np.mean(residuals) + 3*np.std(residuals), color='g',
linestyle='--')
plt.axhline(y=np.mean(residuals) - 3*np.std(residuals), color='g',
linestyle='--')
plt.title('Control Chart of Residuals')
plt.xlabel('Observation')
plt.ylabel('Residual')
plt.show()
```



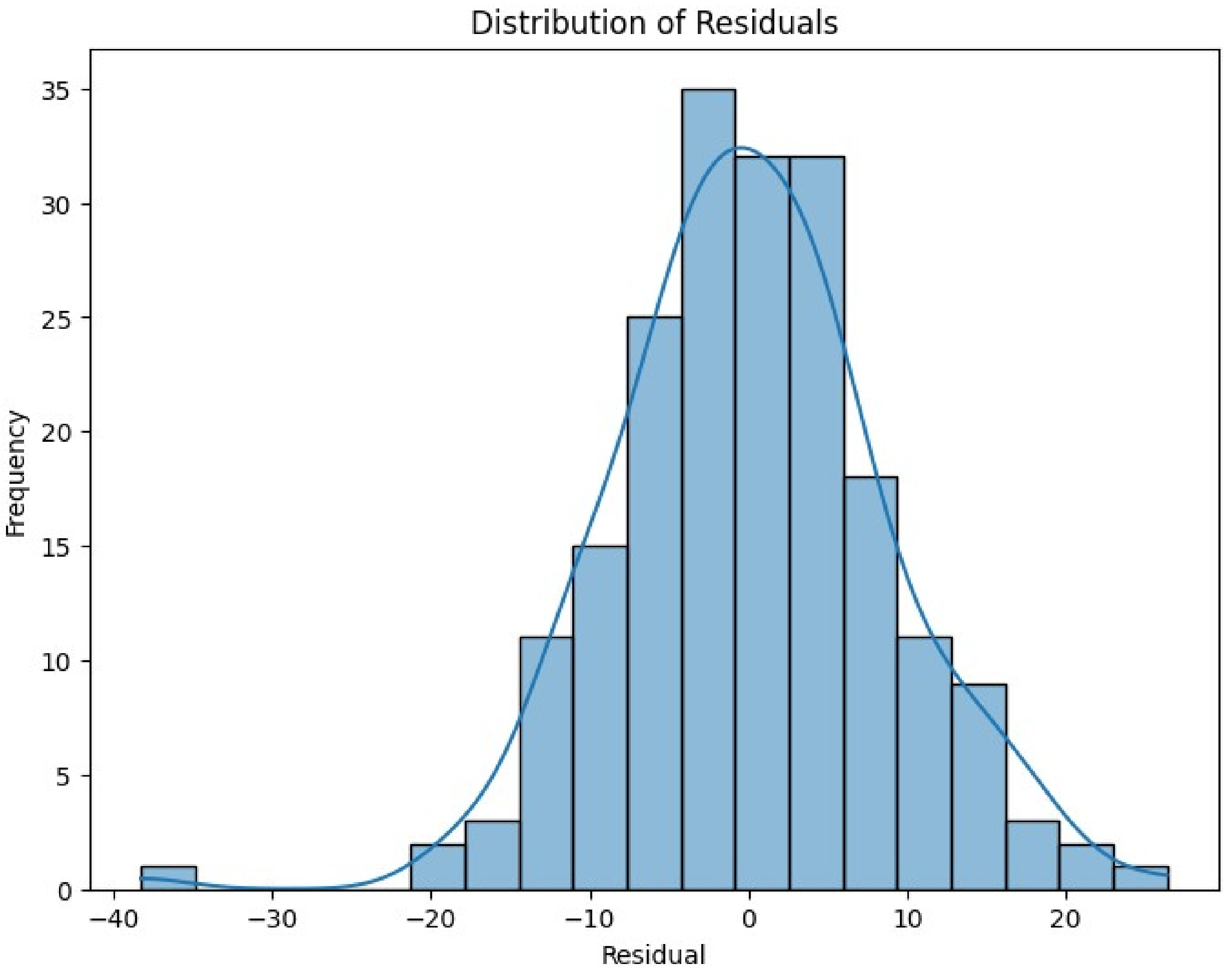
Explanation:

Control Limits: Set at ±3 standard deviations from the mean residual. Purpose: Detect any out-of-control signals indicating model issues. Conclusion Model Performance: Our neural network captures the complex relationships in the data. Optimization: Further tuning and feature engineering can enhance performance. SPC Application: Helps in monitoring and maintaining model reliability over time.

Next Steps Feature Engineering: Incorporate more interaction terms and nonlinear transformations. Advanced Techniques: Use models like XGBoost or Random Forests for comparison. Deployment: Implement the model in a production environment with continuous monitoring.

1. Applying Statistical Process Control Residual Analysis

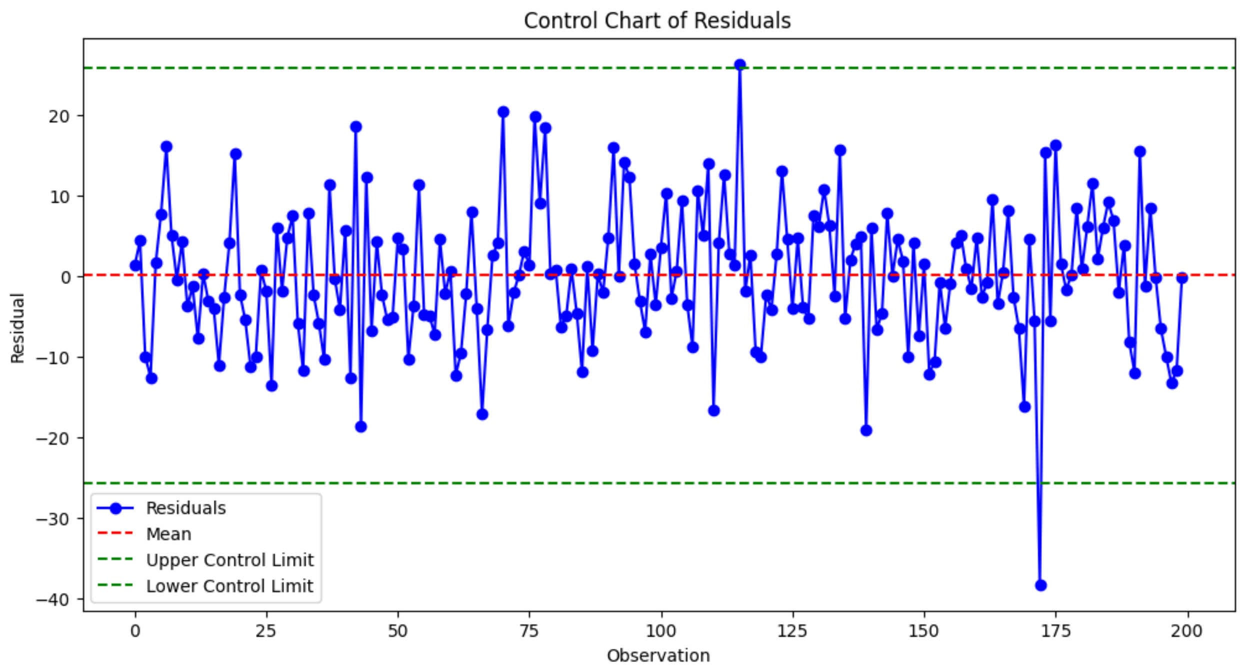
```
# Calculate residuals
residuals = y_test - y_pred.flatten()

# Plot histogram of residuals
plt.figure(figsize=(8, 6))
sns.histplot(residuals, kde=True)
plt.title('Distribution of Residuals')
plt.xlabel('Residual')
plt.ylabel('Frequency')
plt.show()
```



Control Chart of Residuals

```
# Control chart parameters
mean_residual = np.mean(residuals)
std_residual = np.std(residuals)
upper_control_limit = mean_residual + 3 * std_residual
lower_control_limit = mean_residual - 3 * std_residual

# Plot control chart
plt.figure(figsize=(12, 6))
plt.plot(residuals, marker='o', linestyle='-', color='b',
label='Residuals')
plt.axhline(y=mean_residual, color='r', linestyle='--', label='Mean')
plt.axhline(y=upper_control_limit, color='g', linestyle='--',
label='Upper Control Limit')
plt.axhline(y=lower_control_limit, color='g', linestyle='--',
label='Lower Control Limit')
plt.title('Control Chart of Residuals')
plt.xlabel('Observation')
plt.ylabel('Residual')
```

```
plt.legend()
plt.show()
```



Explanation Purpose: Identify any out-of-control points indicating model issues. Control Limits: Set at ±3 standard deviations from the mean residual.

Process Capability Analysis

```
# Process capability indices
process_std = np.std(y_test)
spec_limits = [np.min(y_test), np.max(y_test)]  # Assuming
specifications are the min and max of y_test
process_mean = np.mean(y_test)

# Calculate Cp and Cpk
Cp = (spec_limits[1] - spec_limits[0]) / (6 * process_std)
Cpk = min((spec_limits[1] - process_mean), (process_mean -
spec_limits[0])) / (3 * process_std)

print(f'Process Capability Cp: {Cp:.4f}')
print(f'Process Capability Cpk: {Cpk:.4f}')

Process Capability Cp: 1.0575
Process Capability Cpk: 0.9323
```

Explanation Cp and Cpk: Indices to assess the process capability in meeting specifications. Interpretation: Cp > 1: Process has the potential to meet specifications. Cpk > 1: Process is centered between the specification limits.

1. Quality Management Considerations Plan-Do-Check-Act (PDCA) Cycle Plan: Defined the problem, objectives, and planned data generation and modeling approach. Do: Executed data generation, preprocessing, and model training. Check: Evaluated model performance and applied SPC tools. Act: Identified areas for improvement, such as model tuning or data quality enhancements. Continuous Improvement Feedback Loop: Use evaluation results to refine the model. Employee Training: Ensure team members understand SPC and DOE principles applied in modeling. Documentation: Maintain detailed records of modeling procedures and findings. Compliance with ISO Standards ISO 9001 Principles: Customer Focus: Model aims to improve process quality, benefiting customers. Process Approach: Systematic modeling process aligns with process approach principle. Improvement: Continuous model evaluation and refinement.

2. Conclusion We successfully developed a neural network model to optimize a complex process by incorporating SPC and DOE principles. The model captures linear relationships and interactions between variables, providing valuable insights into the process dynamics.

Key Takeaways Data Generation: Simulated a realistic process with interactions and variability. Modeling Approach: Employed neural networks capable of handling high-dimensional data. Quality Tools: Applied SPC methods to monitor and control model performance. Quality Management: Aligned the project with quality management principles for continuous improvement.