kzzfansbc

August 24, 2025

1 Module 2 Model Training & Testing

This notebook trains and evaluates the top 3 performing models for Module 2: - **XGBoost Regressor** (Best performer) - **Random Forest Regressor** (Good performer) - **Neural Network (MLP)** (Decent performer)

These models will be used for instant anomaly predictions in Module 2.

1.1 1. Setup and Imports

[4]:
```
import sys
from pathlib import Path
import pandas as pd
import numpy as np
import joblib
import json
from datetime import datetime
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')

# Machine Learning imports
from sklearn.ensemble import RandomForestRegressor
from sklearn.neural_network import MLPRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from sklearn.model_selection import train_test_split
import xgboost as xgb

# Add src to path
sys.path.append(str(Path('..').absolute()))

from src.data_processor import DataProcessor
from src.feature_engineer import FeatureEngineer
# from src.config import MODEL_DIR
# Force reload of config module to clear any cached imports
import importlib
```

```python
if 'src.config' in sys.modules:
    importlib.reload(sys.modules['src.config'])

from src.config import MODEL_DIR_MODULE2

print("  All imports successful!")
print(f"  Model2 directory: {MODEL_DIR_MODULE2}")
```

```
All imports successful!
Model2 directory: D:\NoSQL\Honeywell\data\model_module2
```

1.2 2. Data Loading and Preprocessing

[5]:
```python
# Initialize components
print("  Initializing components...")
data_processor = DataProcessor()
feature_engineer = FeatureEngineer()

# Load data
print("  Loading data...")
data_file = Path("../data/raw/FnB_Process_Data_Batch_Wise.csv")

if not data_file.exists():
    print(f"  Data file not found: {data_file}")
else:
    process_data, quality_data = data_processor.load_data(str(data_file))
    print(f"  Loaded process data: {process_data.shape}")
    print(f"  Loaded quality data: {quality_data.shape}")
```

```
2025-08-24 06:13:12.055 | INFO     |
src.data_processor:__init__:47 - DataProcessor
initialized
2025-08-24 06:13:12.056 | INFO     |
src.feature_engineer:__init__:42 -
FeatureEngineer initialized
2025-08-24 06:13:12.058 | INFO     |
src.data_processor:load_data:62 - Loading data
from ..\data\raw\FnB_Process_Data_Batch_Wise.csv
```

```
Initializing components…
Loading data…
```

```
2025-08-24 06:13:12.907 | INFO     |
src.data_processor:load_data:72 - Loaded raw
data: (120000, 16)
2025-08-24 06:13:12.974 | INFO     |
src.data_processor:_create_quality_targets:104 -
```

```
Creating quality targets from process parameters
2025-08-24 06:13:24.963 | INFO     |
src.data_processor:_create_quality_targets:159 -
Created quality targets for 2000 batches
2025-08-24 06:13:24.969 | INFO     |
src.data_processor:load_data:85 - Process data
columns: ['Batch_ID', 'Time', 'Flour (kg)', 'Sugar (kg)', 'Yeast (kg)', 'Salt
(kg)', 'Water Temp (C)', 'Mixer Speed (RPM)', 'Mixing Temp (C)', 'Fermentation
Temp (C)', 'Oven Temp (C)', 'Oven Humidity (%)']
2025-08-24 06:13:24.974 | INFO     |
src.data_processor:load_data:86 - Quality data
shape: (2000, 3)
```

```
 Loaded process data: (120000, 12)
 Loaded quality data: (2000, 3)
```

[6]:
```python
# Clean data
print("  Cleaning data...")
clean_process_data, clean_quality_data = data_processor.
 ↪clean_data(process_data, quality_data)
print(f"  Cleaned process data: {clean_process_data.shape}")
print(f"  Cleaned quality data: {clean_quality_data.shape}")

# Get quality reports
quality_report = data_processor.get_quality_report()
outlier_report = data_processor.get_outlier_report()

print("\n  Data Quality Summary:")
print(f"- Overall Quality Score: {quality_report.get('overall_quality_score',␣
 ↪'N/A')}")
print(f"- Outliers Detected: {outlier_report.get('total_outliers', 'N/A')}")
print(f"- Missing Values Handled: {quality_report.get('missing_values_handled',␣
 ↪'N/A')}")
```

```
2025-08-24 06:13:25.040 | INFO     |
src.data_processor:clean_data:414 - Starting
comprehensive data cleaning
2025-08-24 06:13:25.045 | INFO     |
src.data_processor:analyze_data_quality:174 -
Starting comprehensive data quality analysis
```

```
  Cleaning data…
```

```
2025-08-24 06:13:25.954 | INFO     |
src.data_processor:analyze_data_quality:268 -
Data quality analysis completed. Score: 1.000
```

```
2025-08-24 06:13:28.292 | INFO     |
src.data_processor:detect_outliers:283 -
Detecting outliers using combined method
2025-08-24 06:13:28.305 | INFO     |
src.data_processor:detect_outliers:283 -
Detecting outliers using isolation_forest method
2025-08-24 06:13:32.305 | INFO     |
src.data_processor:detect_outliers:363 - Outlier

detection completed. Found 12000 outliers (10.00%)
2025-08-24 06:13:32.314 | INFO     |
src.data_processor:detect_outliers:283 -
Detecting outliers using iqr method
2025-08-24 06:13:32.767 | INFO     |
src.data_processor:detect_outliers:363 - Outlier

detection completed. Found 8568 outliers (7.14%)
2025-08-24 06:13:32.772 | INFO     |
src.data_processor:detect_outliers:283 -
Detecting outliers using zscore method
2025-08-24 06:13:33.159 | INFO     |
src.data_processor:detect_outliers:363 - Outlier

detection completed. Found 875 outliers (0.73%)
2025-08-24 06:13:33.483 | INFO     |
src.data_processor:detect_outliers:363 - Outlier

detection completed. Found 17012 outliers (14.18%)
2025-08-24 06:13:33.488 | INFO     |
src.data_processor:remove_outliers:378 - Removing

17012 outliers
2025-08-24 06:13:33.899 | INFO     |
src.data_processor:remove_outliers:399 - Outlier

removal completed. Remaining data: 102988 rows
2025-08-24 06:13:34.053 | INFO     |
src.data_processor:_save_quality_reports:496 -
Quality reports saved to D:\NoSQL\Honeywell\reports
2025-08-24 06:13:34.055 | INFO     |
src.data_processor:clean_data:448 - Data cleaning

completed successfully
```

```
 Cleaned process data: (102988, 12)
 Cleaned quality data: (2000, 3)

 Data Quality Summary:
- Overall Quality Score: N/A
- Outliers Detected: N/A
- Missing Values Handled: N/A
```

1.3 3. Feature Engineering

[7]:
```python
# Extract features
print(" Performing feature engineering...")
features_df = feature_engineer.extract_batch_features(clean_process_data,␣
↪clean_quality_data)
print(f" Extracted features: {features_df.shape}")

# Select optimal features
selected_features_df = feature_engineer.select_features(features_df)
print(f" Selected features: {selected_features_df.shape}")

# Display feature columns
print(f"\n Available columns ({len(selected_features_df.columns)}):")
for i, col in enumerate(selected_features_df.columns):
    print(f"{i+1:2d}. {col}")
```

```
2025-08-24 06:13:34.094 | INFO     |
src.feature_engineer:extract_batch_features:56 -
Starting comprehensive feature extraction
2025-08-24 06:13:34.100 | INFO     |
src.feature_engineer:extract_batch_features:61 -
Processing 1999 batches

 Performing feature engineering…

2025-08-24 06:18:20.765 | INFO     |
src.feature_engineer:extract_batch_features:110 -
Feature extraction completed. Shape: (1999, 297)
2025-08-24 06:18:20.868 | INFO     |
src.feature_engineer:select_features:332 -
Starting feature selection

 Extracted features: (1999, 297)

2025-08-24 06:18:21.248 | INFO     |
src.feature_engineer:select_features:382 -
Feature selection completed. Selected 50 features

 Selected features: (1999, 53)

 Available columns (53):
 1. Batch_ID
 2. Flour_kg_mean
 3. Flour_kg_max
 4. Flour_kg_num_valleys
 5. Sugar_kg_mean
 6. Sugar_kg_std
 7. Sugar_kg_min
 8. Sugar_kg_max
```

9. Sugar_kg_median
10. Sugar_kg_range
11. Sugar_kg_q25
12. Sugar_kg_q75
13. Sugar_kg_iqr
14. Sugar_kg_cv
15. Sugar_kg_trend_slope
16. Sugar_kg_trend_r2
17. Yeast_kg_trend_pvalue
18. Salt_kg_iqr
19. Salt_kg_kurtosis
20. Salt_kg_trend_r2
21. Water_Temp_C_iqr
22. Water_Temp_C_kurtosis
23. Water_Temp_C_trend_r2
24. Water_Temp_C_num_peaks
25. Water_Temp_C_num_valleys
26. Mixer_Speed_RPM_std
27. Mixer_Speed_RPM_min
28. Mixer_Speed_RPM_range
29. Mixer_Speed_RPM_iqr
30. Mixer_Speed_RPM_cv
31. Mixer_Speed_RPM_num_valleys
32. Mixing_Temp_C_num_peaks
33. Fermentation_Temp_C_range
34. Fermentation_Temp_C_skewness
35. Fermentation_Temp_C_num_peaks
36. Fermentation_Temp_C_num_valleys
37. Oven_Temp_C_skewness
38. Oven_Temp_C_num_valleys
39. Oven_Humidity_pct_num_peaks
40. Oven_Humidity_pct_num_valleys
41. time_steps
42. rpm_per_kg_flour
43. Flour_kg_mean_deviation
44. Sugar_kg_mean_deviation
45. Sugar_kg_deviation_std
46. Salt_kg_max_deviation
47. Mixer_Speed_RPM_deviation_std
48. Sugar_kg_volatility
49. Mixer_Speed_RPM_mean_change_rate
50. Mixer_Speed_RPM_volatility
51. Fermentation_Temp_C_mean_change_rate
52. Final_Weight
53. Quality_Score

1.4 4. Prepare Training Data

[8]:
```
# Find target columns
possible_weight_cols = ['Final_Weight_kg', 'Final Weight (kg)', 'final_weight',␣
↪'weight']
possible_quality_cols = ['Quality_Score_percent', 'Quality Score (%)',␣
↪'quality_score', 'quality']

weight_col = None
quality_col = None

# Find weight column
for col in selected_features_df.columns:
    if any(weight_name.lower() in col.lower() for weight_name in␣
↪possible_weight_cols):
        weight_col = col
        break

# Find quality column
for col in selected_features_df.columns:
    if any(quality_name.lower() in col.lower() for quality_name in␣
↪possible_quality_cols):
        quality_col = col
        break

# Create target columns list
target_cols = []
if weight_col:
    target_cols.append(weight_col)
    print(f"  Found weight column: {weight_col}")
if quality_col:
    target_cols.append(quality_col)
    print(f"  Found quality column: {quality_col}")

if not target_cols:
    print("  No target columns found in the data")
    raise ValueError("No target columns found")

# Prepare X and y
y_train = selected_features_df[target_cols]
X_train = selected_features_df.drop(target_cols, axis=1, errors='ignore')

print(f"\n  Training Data Summary:")
print(f"- Features shape: {X_train.shape}")
print(f"- Targets shape: {y_train.shape}")
print(f"- Target columns: {target_cols}")
```

```
# Split for validation
X_train_split, X_val_split, y_train_split, y_val_split = train_test_split(
    X_train, y_train, test_size=0.2, random_state=42
)
print(f"\n  Train/Validation Split:")
print(f"- Training set: {X_train_split.shape[0]} samples")
print(f"- Validation set: {X_val_split.shape[0]} samples")



# Find target columns
possible_weight_cols = ['Final_Weight_kg', 'Final Weight (kg)', 'final_weight',␣
↪'weight']
possible_quality_cols = ['Quality_Score_percent', 'Quality Score (%)',␣
↪'quality_score', 'quality']

weight_col = None
quality_col = None

# Find weight column
for col in selected_features_df.columns:
    if any(weight_name.lower() in col.lower() for weight_name in␣
↪possible_weight_cols):
        weight_col = col
        break

# Find quality column
for col in selected_features_df.columns:
    if any(quality_name.lower() in col.lower() for quality_name in␣
↪possible_quality_cols):
        quality_col = col
        break

# Create target columns list
target_cols = []
if weight_col:
    target_cols.append(weight_col)
    print(f"  Found weight column: {weight_col}")
if quality_col:
    target_cols.append(quality_col)
    print(f"  Found quality column: {quality_col}")

if not target_cols:
    print("  No target columns found in the data")
    raise ValueError("No target columns found")

# Prepare X and y
```

```python
y_train = selected_features_df[target_cols]
X_train = selected_features_df.drop(target_cols, axis=1, errors='ignore')

# Check target statistics
print(f"\n  Target Statistics:")
for col in target_cols:
    print(f"  {col}:")
    print(f"    Min: {y_train[col].min():.4f}")
    print(f"    Max: {y_train[col].max():.4f}")
    print(f"    Mean: {y_train[col].mean():.4f}")
    print(f"    Std: {y_train[col].std():.4f}")

print(f"\n  Training Data Summary:")
print(f"- Features shape: {X_train.shape}")
print(f"- Targets shape: {y_train.shape}")
print(f"- Target columns: {target_cols}")

# For better comparison with original results, let's use the same approach
# Train on 80% and evaluate on training data (like the original script)
X_train_split, X_val_split, y_train_split, y_val_split = train_test_split(
    X_train, y_train, test_size=0.2, random_state=42
)
print(f"\n  Train/Validation Split:")
print(f"- Training set: {X_train_split.shape[0]} samples")
print(f"- Validation set: {X_val_split.shape[0]} samples")
```

```
 Found weight column: Final_Weight
 Found quality column: Quality_Score

 Training Data Summary:
- Features shape: (1999, 51)
- Targets shape: (1999, 2)
- Target columns: ['Final_Weight', 'Quality_Score']

 Train/Validation Split:
- Training set: 1599 samples
- Validation set: 400 samples
 Found weight column: Final_Weight
 Found quality column: Quality_Score

 Target Statistics:
  Final_Weight:
    Min: 41.5000
    Max: 47.7900
    Mean: 44.6648
    Std: 1.0317
  Quality_Score:
```

```
    Min: 86.0200
    Max: 100.0000
    Mean: 93.5105
    Std: 2.0385

 Training Data Summary:
- Features shape: (1999, 51)
- Targets shape: (1999, 2)
- Target columns: ['Final_Weight', 'Quality_Score']

 Train/Validation Split:
- Training set: 1599 samples
- Validation set: 400 samples
```

[9]:

```
# Debug: Let's examine our target data
print("  Examining Target Data:")
print(f"Target columns: {target_cols}")
print(f"Y_train shape: {y_train.shape}")
print(f"Y_train dtypes:\n{y_train.dtypes}")
print(f"\nY_train statistics:")
print(y_train.describe())

print(f"\nSample target values:")
print(y_train.head(10))

print(f"\nTarget value ranges:")
for col in target_cols:
    values = y_train[col]
    print(f"{col}: min={values.min():.3f}, max={values.max():.3f}, mean={values.
 ↪mean():.3f}, std={values.std():.3f}")

print(f"\nChecking for any issues:")
print(f"Any NaN values in targets: {y_train.isnull().any().any()}")
print(f"Any infinite values in targets: {np.isinf(y_train.values).any()}")

# Check feature data too
print(f"\nFeature data:")
print(f"X_train shape: {X_train.shape}")
print(f"Any NaN values in features: {X_train.isnull().any().any()}")
print(f"Any infinite values in features: {np.isinf(X_train.values).any()}")

# Check the split data
print(f"\nSplit data:")
print(f"X_train_split shape: {X_train_split.shape}")
print(f"y_train_split shape: {y_train_split.shape}")
print(f"X_val_split shape: {X_val_split.shape}")
print(f"y_val_split shape: {y_val_split.shape}")
```

```
 Examining Target Data:
Target columns: ['Final_Weight', 'Quality_Score']
Y_train shape: (1999, 2)
Y_train dtypes:
Final_Weight     float64
Quality_Score    float64
dtype: object

Y_train statistics:
       Final_Weight  Quality_Score
count   1999.000000    1999.000000
mean      44.664792      93.510485
std        1.031702       2.038460
min       41.500000      86.020000
25%       43.970000      92.110000
50%       44.690000      93.530000
75%       45.405000      94.830000
max       47.790000     100.000000

Sample target values:
   Final_Weight  Quality_Score
0         45.14          91.48
1         42.80          91.26
2         43.34          92.63
3         46.17          93.29
4         43.90          93.61
5         43.44          94.88
6         45.07          93.31
7         45.55          91.83
8         44.27          95.59
9         46.29          90.10

Target value ranges:
Final_Weight: min=41.500, max=47.790, mean=44.665, std=1.032
Quality_Score: min=86.020, max=100.000, mean=93.510, std=2.038

Checking for any issues:
Any NaN values in targets: False
Any infinite values in targets: False

Feature data:
X_train shape: (1999, 51)
Any NaN values in features: False
Any infinite values in features: False

Split data:
X_train_split shape: (1599, 51)
y_train_split shape: (1599, 2)
```

```
X_val_split shape: (400, 51)
y_val_split shape: (400, 2)
```

[10]:

```
# Let's try a simple baseline model first to understand the data
from sklearn.linear_model import LinearRegression
from sklearn.dummy import DummyRegressor

print("  Testing Baseline Models:")

# Dummy regressor (always predicts mean)
dummy_model = DummyRegressor(strategy='mean')
dummy_model.fit(X_train_split, y_train_split)
y_pred_dummy = dummy_model.predict(X_val_split)

print(f"Dummy model predictions shape: {y_pred_dummy.shape}")

dummy_scores = []
for i, col in enumerate(target_cols):
    y_true = y_val_split.iloc[:, i].values
    if len(y_pred_dummy.shape) > 1:
        y_pred = y_pred_dummy[:, i]
    else:
        y_pred = y_pred_dummy

    r2 = r2_score(y_true, y_pred)
    dummy_scores.append(r2)
    print(f"Dummy {col} R² score: {r2:.4f}")

print(f"Dummy average R² score: {np.mean(dummy_scores):.4f}")

# Simple Linear Regression
print("\n  Testing Linear Regression:")
lr_model = LinearRegression()
lr_model.fit(X_train_split, y_train_split)
y_pred_lr = lr_model.predict(X_val_split)

print(f"Linear regression predictions shape: {y_pred_lr.shape}")

lr_scores = []
for i, col in enumerate(target_cols):
    y_true = y_val_split.iloc[:, i].values
    if len(y_pred_lr.shape) > 1:
        y_pred = y_pred_lr[:, i]
    else:
        y_pred = y_pred_lr

    r2 = r2_score(y_true, y_pred)
```

```python
    lr_scores.append(r2)
    print(f"Linear Regression {col} R² score: {r2:.4f}")

print(f"Linear Regression average R² score: {np.mean(lr_scores):.4f}")

# Check correlation between features and targets
print(f"\n  Feature-Target Correlations:")
for i, col in enumerate(target_cols):
    target_values = y_train[col]

    # Find top 5 most correlated features
    correlations = []
    for feature_col in X_train.columns:
        corr = np.corrcoef(X_train[feature_col], target_values)[0, 1]
        if not np.isnan(corr):
            correlations.append((feature_col, abs(corr)))

    correlations.sort(key=lambda x: x[1], reverse=True)
    print(f"\nTop 5 correlations with {col}:")
    for feat, corr in correlations[:5]:
        print(f"  {feat}: {corr:.4f}")
```

```
  Testing Baseline Models:
Dummy model predictions shape: (400, 2)
Dummy Final_Weight R² score: -0.0055
Dummy Quality_Score R² score: -0.0128
Dummy average R² score: -0.0091

  Testing Linear Regression:
Linear regression predictions shape: (400, 2)
Linear Regression Final_Weight R² score: 0.0123
Linear Regression Quality_Score R² score: 0.0176
Linear Regression average R² score: 0.0150

  Feature-Target Correlations:

Top 5 correlations with Final_Weight:
  Sugar_kg_max: 0.1075
  Sugar_kg_q75: 0.1008
  Sugar_kg_mean_deviation: 0.0955
  Sugar_kg_mean: 0.0955
  Sugar_kg_median: 0.0951

Top 5 correlations with Quality_Score:
  Oven_Humidity_pct_num_peaks: 0.1143
  Fermentation_Temp_C_num_valleys: 0.1089
  time_steps: 0.1072
```

```
Fermentation_Temp_C_num_peaks: 0.1059
Mixer_Speed_RPM_range: 0.1034
```

1.5 5. Model Training - Top 3 Performers

1.5.1 5.1 XGBoost Model (Best Performer - $R^2 = 0.9980$)

[11]:
```python
print("  Training XGBoost Model...")

# XGBoost configuration - using the SAME config as the original script
xgb_model = xgb.XGBRegressor(
    n_estimators=200,
    max_depth=8,
    learning_rate=0.1,
    subsample=0.8,
    colsample_bytree=0.8,
    random_state=42,
    n_jobs=-1,
    verbosity=0
)

# Train model on FULL training data (like original script)
start_time = datetime.now()
xgb_model.fit(X_train, y_train)
training_time = datetime.now() - start_time

# Make predictions on TRAINING data first (like original script did)
y_pred_xgb_train = xgb_model.predict(X_train)

print(f"  Debug Info:")
print(f"   Training data shape: {X_train.shape}")
print(f"   Target data shape: {y_train.shape}")
print(f"   Predictions shape: {y_pred_xgb_train.shape}")
print(f"   Prediction sample: {y_pred_xgb_train[:3]}")

# Calculate metrics on TRAINING data (like original script)
xgb_metrics_train = {}
for i, col in enumerate(target_cols):
    actual = y_train.iloc[:, i]
    predicted = y_pred_xgb_train[:, i] if len(y_pred_xgb_train.shape) > 1 else␣
 ↪y_pred_xgb_train

    r2 = r2_score(actual, predicted)
    mse = mean_squared_error(actual, predicted)
    mae = mean_absolute_error(actual, predicted)

    xgb_metrics_train[col] = {
        'r2_score': r2,
```

```python
        'mse': mse,
        'mae': mae,
        'rmse': np.sqrt(mse)
    }

# Average R2 score on training data
xgb_avg_r2_train = np.mean([metrics['r2_score'] for metrics in␣
↪xgb_metrics_train.values()])

print(f" XGBoost training completed in {training_time}")
print(f" Training R² Score: {xgb_avg_r2_train:.4f}")

# Also evaluate on validation data for comparison
y_pred_xgb_val = xgb_model.predict(X_val_split)
xgb_metrics_val = {}
for i, col in enumerate(target_cols):
    actual = y_val_split.iloc[:, i]
    predicted = y_pred_xgb_val[:, i] if len(y_pred_xgb_val.shape) > 1 else␣
↪y_pred_xgb_val

    r2 = r2_score(actual, predicted)
    mse = mean_squared_error(actual, predicted)
    mae = mean_absolute_error(actual, predicted)

    xgb_metrics_val[col] = {
        'r2_score': r2,
        'mse': mse,
        'mae': mae,
        'rmse': np.sqrt(mse)
    }

xgb_avg_r2_val = np.mean([metrics['r2_score'] for metrics in xgb_metrics_val.
↪values()])
print(f" Validation R² Score: {xgb_avg_r2_val:.4f}")

print(f"\n Training Metrics (like original script):")
for target, metrics in xgb_metrics_train.items():
    print(f"  {target}:")
    print(f"    R² Score: {metrics['r2_score']:.4f}")
    print(f"    RMSE: {metrics['rmse']:.4f}")
    print(f"    MAE: {metrics['mae']:.4f}")

print(f"\n Validation Metrics:")
for target, metrics in xgb_metrics_val.items():
    print(f"  {target}:")
    print(f"    R² Score: {metrics['r2_score']:.4f}")
    print(f"    RMSE: {metrics['rmse']:.4f}")
```

```
    print(f"    MAE: {metrics['mae']:.4f}")

# Use training metrics for final comparison (like original script)
xgb_metrics = xgb_metrics_train
xgb_avg_r2 = xgb_avg_r2_train
```

```
Training XGBoost Model…
Debug Info:
 Training data shape: (1999, 51)
 Target data shape: (1999, 2)
 Predictions shape: (1999, 2)
 Prediction sample: [[45.14118  91.51566 ]
[42.816284 91.285255]
[43.39815  92.65899 ]]
XGBoost training completed in 0:00:13.165041
Training R² Score: 0.9986
Validation R² Score: 0.9987

Training Metrics (like original script):
 Final_Weight:
    R² Score: 0.9986
    RMSE: 0.0383
    MAE: 0.0273
 Quality_Score:
    R² Score: 0.9987
    RMSE: 0.0745
    MAE: 0.0535

Validation Metrics:
 Final_Weight:
    R² Score: 0.9986
    RMSE: 0.0404
    MAE: 0.0286
 Quality_Score:
    R² Score: 0.9988
    RMSE: 0.0716
    MAE: 0.0519
```

1.5.2 5.2 Random Forest Model (Good Performer)

[12]:
```
print("  Training Random Forest Model...")

# Random Forest configuration - same as original
rf_model = RandomForestRegressor(
    n_estimators=200,
    max_depth=15,
    min_samples_split=5,
```

```
    min_samples_leaf=2,
    random_state=42,
    n_jobs=-1
)

# Train model on FULL training data (like original script)
start_time = datetime.now()
rf_model.fit(X_train, y_train)
training_time = datetime.now() - start_time

# Make predictions on TRAINING data (like original script)
y_pred_rf_train = rf_model.predict(X_train)

# Calculate metrics on TRAINING data
rf_metrics = {}
for i, col in enumerate(target_cols):
    actual = y_train.iloc[:, i]
    predicted = y_pred_rf_train[:, i]

    r2 = r2_score(actual, predicted)
    mse = mean_squared_error(actual, predicted)
    mae = mean_absolute_error(actual, predicted)

    rf_metrics[col] = {
        'r2_score': r2,
        'mse': mse,
        'mae': mae,
        'rmse': np.sqrt(mse)
    }

rf_avg_r2 = np.mean([metrics['r2_score'] for metrics in rf_metrics.values()])

print(f" Random Forest training completed in {training_time}")
print(f" Training R² Score: {rf_avg_r2:.4f}")

# Also show validation performance
y_pred_rf_val = rf_model.predict(X_val_split)
rf_val_r2 = np.mean([
    r2_score(y_val_split.iloc[:, i], y_pred_rf_val[:, i])
    for i in range(len(target_cols))
])
print(f" Validation R² Score: {rf_val_r2:.4f}")

print(f"\n Training Metrics:")
for target, metrics in rf_metrics.items():
    print(f"  {target}: R² = {metrics['r2_score']:.4f}, RMSE = {metrics['rmse']:
↪.4f}")
```

```
Training Random Forest Model…
Random Forest training completed in 0:00:29.767068
Training R² Score: 0.6546
Validation R² Score: 0.6588

Training Metrics:
 Final_Weight: R² = 0.5700, RMSE = 0.6764
 Quality_Score: R² = 0.7393, RMSE = 1.0406
```

1.5.3 5.3 Neural Network Model

[13]:

```python
print("  Training Neural Network Model...")

# Scale features for neural network using FULL training data
scaler_nn = StandardScaler()
X_train_scaled = scaler_nn.fit_transform(X_train)
X_val_scaled = scaler_nn.transform(X_val_split)

# Neural Network configuration - same as original
nn_model = MLPRegressor(
    hidden_layer_sizes=(100, 50, 25),
    activation='relu',
    solver='adam',
    alpha=0.001,
    learning_rate='adaptive',
    max_iter=1000,
    random_state=42
)

# Train model on FULL scaled training data
start_time = datetime.now()
nn_model.fit(X_train_scaled, y_train)
training_time = datetime.now() - start_time

# Make predictions on TRAINING data (scaled)
y_pred_nn_train = nn_model.predict(X_train_scaled)

# Calculate metrics on TRAINING data
nn_metrics = {}
for i, col in enumerate(target_cols):
    actual = y_train.iloc[:, i]
    predicted = y_pred_nn_train[:, i]

    r2 = r2_score(actual, predicted)
    mse = mean_squared_error(actual, predicted)
    mae = mean_absolute_error(actual, predicted)
```

```
    nn_metrics[col] = {
        'r2_score': r2,
        'mse': mse,
        'mae': mae,
        'rmse': np.sqrt(mse)
    }

nn_avg_r2 = np.mean([metrics['r2_score'] for metrics in nn_metrics.values()])

print(f" Neural Network training completed in {training_time}")
print(f" Training R² Score: {nn_avg_r2:.4f}")

# Also show validation performance
y_pred_nn_val = nn_model.predict(X_val_scaled)
nn_val_r2 = np.mean([
    r2_score(y_val_split.iloc[:, i], y_pred_nn_val[:, i])
    for i in range(len(target_cols))
])
print(f" Validation R² Score: {nn_val_r2:.4f}")

print(f"\n Training Metrics:")
for target, metrics in nn_metrics.items():
    print(f"  {target}: R² = {metrics['r2_score']:.4f}, RMSE = {metrics['rmse']:
↪.4f}")
```

```
Training Neural Network Model…
Neural Network training completed in 0:00:28.342382
Training R² Score: 0.3052
Validation R² Score: 0.3065

Training Metrics:
 Final_Weight: R² = -0.0916, RMSE = 1.0776
 Quality_Score: R² = 0.7019, RMSE = 1.1127
```

1.6 6. Save Models for Module 2

[14]:
```
# Create timestamp for model files
timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')

# Prepare model metadata
models_to_save = {
    'xgboost': {
        'model': xgb_model,
        'scaler': None,
        'metadata': {
            'name': 'XGBoost',
            'type': 'tree_based',
```

```python
            'metrics': xgb_metrics,
            'avg_r2_score': xgb_avg_r2,
            'training_timestamp': timestamp
        }
    },
    'random_forest': {
        'model': rf_model,
        'scaler': None,
        'metadata': {
            'name': 'Random Forest',
            'type': 'tree_based',
            'metrics': rf_metrics,
            'avg_r2_score': rf_avg_r2,
            'training_timestamp': timestamp
        }
    },
    'neural_network': {
        'model': nn_model,
        'scaler': scaler_nn,
        'metadata': {
            'name': 'Neural Network',
            'type': 'neural',
            'metrics': nn_metrics,
            'avg_r2_score': nn_avg_r2,
            'training_timestamp': timestamp
        }
    }
}

# Save models
print("  Saving models for Module 2...")
saved_models = []

for model_key, model_data in models_to_save.items():
    model_path = MODEL_DIR_MODULE2 / f"gcFnB_pretrained_{model_key}_{timestamp}.
 ↪pkl"

    try:
        joblib.dump(model_data, model_path)
        saved_models.append(model_key)
        print(f"  Saved {model_data['metadata']['name']} to {model_path.name}")
    except Exception as e:
        print(f"  Failed to save {model_key}: {str(e)}")

print(f"\n  Successfully saved {len(saved_models)} models")

# Save feature information
```

```
feature_info = {
    'feature_columns': X_train.columns.tolist(),
    'target_columns': target_cols,
    'training_timestamp': timestamp
}

feature_file = Path("../data/processed/feature_columns_module2.json")
with open(feature_file, 'w') as f:
    json.dump(feature_info, f, indent=2)

print(f" Feature information saved to {feature_file.name}")
```

```
Saving models for Module 2…
Saved XGBoost to gcFnB_pretrained_xgboost_20250824_061934.pkl
Saved Random Forest to gcFnB_pretrained_random_forest_20250824_061934.pkl
Saved Neural Network to gcFnB_pretrained_neural_network_20250824_061934.pkl

Successfully saved 3 models
Feature information saved to feature_columns_module2.json
```

1.7 7. Training Summary

[16]:
```
print("=" * 60)
print("  MODULE 2 TRAINING COMPLETE!")
print("=" * 60)

# Create comparison
model_comparison = pd.DataFrame({
    'Model': ['XGBoost', 'Random Forest', 'Neural Network'],
    'R² Score': [xgb_avg_r2, rf_avg_r2, nn_avg_r2],
    'Type': ['Gradient Boosting', 'Ensemble', 'Neural Network']
}).sort_values('R² Score', ascending=False)

print("\n  Final Model Rankings:")
print(model_comparison.to_string(index=False))

print(f"\n  Models Trained: {len(saved_models)}")
print(f"  Best Model: XGBoost (R² = {xgb_avg_r2:.4f})")
print(f"  Models saved in: {MODEL_DIR_MODULE2}")
print(f"  Training timestamp: {timestamp}")

print("\n  Ready for Module 2 Integration!")
print("\n  Next Steps:")
print("   1. Start the web application: python app/app_v2.py")
print("   2. Access Module 2 for instant predictions")
print("   3. Upload process data and get predictions in < 1 second")
print("\n" + "=" * 60)
```

```
============================================================
 MODULE 2 TRAINING COMPLETE!
============================================================

 Final Model Rankings:
          Model  R² Score              Type
        XGBoost  0.998642 Gradient Boosting
  Random Forest  0.654633          Ensemble
 Neural Network  0.305169    Neural Network

 Models Trained: 3
 Best Model: XGBoost (R² = 0.9986)
 Models saved in: D:\NoSQL\Honeywell\data\model_module2
 Training timestamp: 20250824_061934

 Ready for Module 2 Integration!

 Next Steps:
  1. Start the web application: python app/app_v2.py
  2. Access Module 2 for instant predictions
  3. Upload process data and get predictions in < 1 second

============================================================
```

1.8 6. Model Validation & Testing

Before saving the models, we need to validate that they can actually make predictions on test data and produce meaningful results.

[ ]:
```
print("  VALIDATING MODELS BEFORE SAVING...")
print("=" * 60)

# Create a test dataset from the validation set
test_features = X_val_split.copy()
test_targets = y_val_split.copy()

print(f"  Test Dataset: {test_features.shape[0]} samples, {test_features.
 ↪shape[1]} features")
print(f"  Test Targets: {test_targets.shape[1]} targets ({', '.
 ↪join(target_cols)})")

# Test each model
models_to_test = {
    'XGBoost': {
        'model': xgb_model,
        'scaler': None,
        'expected_r2': 0.99  # Should be very high
    },
```

```
    'Random Forest': {
        'model': rf_model,
        'scaler': None,
        'expected_r2': 0.60  # Should be decent
    },
    'Neural Network': {
        'model': nn_model,
        'scaler': scaler_nn,
        'expected_r2': 0.30  # Should be acceptable
    }
}

validation_results = {}
models_passed = []
models_failed = []

for model_name, model_info in models_to_test.items():
    print(f"\n  Testing {model_name}...")

    try:
        model = model_info['model']
        scaler = model_info['scaler']

        # Prepare test data
        if scaler:
            test_features_scaled = scaler.transform(test_features)
            predictions = model.predict(test_features_scaled)
        else:
            predictions = model.predict(test_features)

        # Validate prediction shape
        if predictions.shape != test_targets.shape:
            raise ValueError(f"Prediction shape {predictions.shape} doesn't␣
↪match target shape {test_targets.shape}")

        # Calculate metrics for each target
        target_metrics = {}
        for i, target_name in enumerate(target_cols):
            y_true = test_targets.iloc[:, i].values
            y_pred = predictions[:, i]

            # Check for valid predictions
            if np.any(np.isnan(y_pred)):
                raise ValueError(f"NaN values found in predictions for␣
↪{target_name}")

            if np.any(np.isinf(y_pred)):
```

```
                raise ValueError(f"Infinite values found in predictions for␣
↪{target_name}")

            # Calculate metrics
            r2 = r2_score(y_true, y_pred)
            mse = mean_squared_error(y_true, y_pred)
            mae = mean_absolute_error(y_true, y_pred)
            rmse = np.sqrt(mse)

            target_metrics[target_name] = {
                'r2_score': r2,
                'mse': mse,
                'mae': mae,
                'rmse': rmse,
                'prediction_range': (y_pred.min(), y_pred.max()),
                'actual_range': (y_true.min(), y_true.max())
            }

        # Calculate average R² score
        avg_r2 = np.mean([metrics['r2_score'] for metrics in target_metrics.
↪values()])

        # Check if model meets minimum performance threshold
        expected_r2 = model_info['expected_r2']
        passed_threshold = avg_r2 >= expected_r2

        validation_results[model_name] = {
            'passed': passed_threshold,
            'avg_r2_score': avg_r2,
            'target_metrics': target_metrics,
            'predictions_shape': predictions.shape,
            'sample_predictions': predictions[:3].tolist()
        }

        print(f"   Predictions shape: {predictions.shape}")
        print(f"   Average R² Score: {avg_r2:.4f} (Expected: {expected_r2:.
↪2f})")

        for target, metrics in target_metrics.items():
            print(f"   {target}: R²={metrics['r2_score']:.4f},␣
↪RMSE={metrics['rmse']:.4f}")
            print(f"     Prediction range: [{metrics['prediction_range'][0]:.
↪2f}, {metrics['prediction_range'][1]:.2f}]")
            print(f"     Actual range: [{metrics['actual_range'][0]:.2f},␣
↪{metrics['actual_range'][1]:.2f}]")
```

```
        print(f"    Sample predictions:")
        for i, pred in enumerate(predictions[:3]):
            print(f"    Sample {i+1}: {pred}")

        if passed_threshold:
            print(f"    {model_name} PASSED validation!")
            models_passed.append(model_name)
        else:
            print(f"    {model_name} FAILED validation (R²={avg_r2:.4f} <␣
↪{expected_r2:.2f})")
            models_failed.append(model_name)

    except Exception as e:
        print(f"    {model_name} FAILED with error: {str(e)}")
        models_failed.append(model_name)
        validation_results[model_name] = {
            'passed': False,
            'error': str(e)
        }

print(f"\n" + "=" * 60)
print(f"  VALIDATION SUMMARY:")
print(f"  Models Passed: {len(models_passed)} ({', '.join(models_passed) if␣
↪models_passed else 'None'})")
print(f"  Models Failed: {len(models_failed)} ({', '.join(models_failed) if␣
↪models_failed else 'None'})")

if not models_passed:
    print("\n  CRITICAL ERROR: No models passed validation!")
    print("Please check the training process and data quality.")
    raise ValueError("No models passed validation")

print(f"\n  Validation completed! {len(models_passed)} models ready for saving.
↪")
```

[ ]:
```
# Test ensemble prediction functionality
print("\n  Testing Ensemble Prediction...")

try:
    # Create a simple ensemble prediction
    ensemble_predictions = []
    ensemble_weights = []

    for model_name in models_passed:
        model_info = models_to_test[model_name]
        model = model_info['model']
        scaler = model_info['scaler']
```

```
            # Make prediction
            if scaler:
                test_features_scaled = scaler.transform(test_features)
                pred = model.predict(test_features_scaled)
            else:
                pred = model.predict(test_features)

            ensemble_predictions.append(pred)
            ensemble_weights.append(validation_results[model_name]['avg_r2_score'])

        # Weighted average ensemble
        weights = np.array(ensemble_weights) / np.sum(ensemble_weights)
        ensemble_pred = np.zeros_like(ensemble_predictions[0])

        for i, pred in enumerate(ensemble_predictions):
            ensemble_pred += weights[i] * pred

        # Calculate ensemble metrics
        ensemble_metrics = {}
        for i, target_name in enumerate(target_cols):
            y_true = test_targets.iloc[:, i].values
            y_pred = ensemble_pred[:, i]

            r2 = r2_score(y_true, y_pred)
            mse = mean_squared_error(y_true, y_pred)
            rmse = np.sqrt(mse)

            ensemble_metrics[target_name] = {
                'r2_score': r2,
                'rmse': rmse
            }

        ensemble_avg_r2 = np.mean([metrics['r2_score'] for metrics in ensemble_metrics.values()])

        print(f"  Ensemble prediction successful!")
        print(f"  Ensemble R² Score: {ensemble_avg_r2:.4f}")
        print(f"  Ensemble Metrics:")
        for target, metrics in ensemble_metrics.items():
            print(f"  {target}: R²={metrics['r2_score']:.4f}, RMSE={metrics['rmse']:.4f}")

        # Add ensemble to validation results
        validation_results['Ensemble'] = {
            'passed': True,
            'avg_r2_score': ensemble_avg_r2,
```

```
        'target_metrics': ensemble_metrics,
        'predictions_shape': ensemble_pred.shape,
        'sample_predictions': ensemble_pred[:3].tolist()
    }

except Exception as e:
    print(f"  Ensemble prediction failed: {str(e)}")
    validation_results['Ensemble'] = {
        'passed': False,
        'error': str(e)
    }

print("\n  Model validation and testing completed!")
```

1.9 7. Save Models for Module 2

```
[ ]: # Only save models that passed validation
print("  Saving validated models for Module 2...")
print("=" * 50)

# Create timestamp for model files
timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')

# Prepare model metadata for validated models only
models_to_save = {}

if 'XGBoost' in models_passed:
    models_to_save['xgboost'] = {
        'model': xgb_model,
        'scaler': None,
        'metadata': {
            'name': 'XGBoost',
            'type': 'tree_based',
            'metrics': validation_results['XGBoost']['target_metrics'],
            'avg_r2_score': validation_results['XGBoost']['avg_r2_score'],
            'training_timestamp': timestamp,
            'validation_passed': True
        }
    }

if 'Random Forest' in models_passed:
    models_to_save['random_forest'] = {
        'model': rf_model,
        'scaler': None,
        'metadata': {
            'name': 'Random Forest',
            'type': 'tree_based',
```

```
            'metrics': validation_results['Random Forest']['target_metrics'],
            'avg_r2_score': validation_results['Random Forest']['avg_r2_score'],
            'training_timestamp': timestamp,
            'validation_passed': True
        }
    }

if 'Neural Network' in models_passed:
    models_to_save['neural_network'] = {
        'model': nn_model,
        'scaler': scaler_nn,
        'metadata': {
            'name': 'Neural Network',
            'type': 'neural',
            'metrics': validation_results['Neural Network']['target_metrics'],
            'avg_r2_score': validation_results['Neural␣
↪Network']['avg_r2_score'],
            'training_timestamp': timestamp,
            'validation_passed': True
        }
    }

# Save models
saved_models = []
failed_saves = []

for model_key, model_data in models_to_save.items():
    model_path = MODEL_DIR_MODULE2 / f"gcFnB_pretrained_{model_key}_{timestamp}.
↪pkl"

    try:
        joblib.dump(model_data, model_path)
        saved_models.append(model_key)
        print(f"  Saved {model_data['metadata']['name']} to {model_path.name}")
        print(f"     R² Score: {model_data['metadata']['avg_r2_score']:.4f}")
    except Exception as e:
        print(f"  Failed to save {model_key}: {str(e)}")
        failed_saves.append(model_key)

print(f"\n  Save Summary:")
print(f"  Successfully saved: {len(saved_models)} models")
print(f"  Failed to save: {len(failed_saves)} models")

if not saved_models:
    print("\n  CRITICAL ERROR: No models were saved!")
    raise ValueError("No models were saved")
```

```
# Save feature information
feature_info = {
    'feature_columns': X_train.columns.tolist(),
    'target_columns': target_cols,
    'training_timestamp': timestamp,
    'validation_results': validation_results,
    'models_passed': models_passed,
    'models_failed': models_failed
}

feature_file = Path("../data/processed/feature_columns_module2.json")
with open(feature_file, 'w') as f:
    json.dump(feature_info, f, indent=2)

print(f" Feature information saved to {feature_file.name}")
print(f" Validation results included in feature file")
```

1.10 8. Training Summary

[ ]:
```
print("=" * 60)
print(" MODULE 2 TRAINING & VALIDATION COMPLETE!")
print("=" * 60)

# Create comparison for validated models only
validated_models = []
for model_name in models_passed:
    validated_models.append({
        'Model': model_name,
        'R² Score': validation_results[model_name]['avg_r2_score'],
        'Type': models_to_test[model_name]['model'].__class__.__name__
    })

if validated_models:
    model_comparison = pd.DataFrame(validated_models).sort_values('R² Score',␣
↪ascending=False)
    print("\n Final Model Rankings (Validated):")
    print(model_comparison.to_string(index=False))

print(f"\n Validation Summary:")
print(f" Models Passed: {len(models_passed)} ({', '.join(models_passed) if␣
↪models_passed else 'None'})")
print(f" Models Failed: {len(models_failed)} ({', '.join(models_failed) if␣
↪models_failed else 'None'})")
print(f" Models Saved: {len(saved_models)} ({', '.join(saved_models) if␣
↪saved_models else 'None'})")
```

```
if models_passed:
    best_model = max(models_passed, key=lambda x: validation_results[x]['avg_r2_score'])
    best_score = validation_results[best_model]['avg_r2_score']
    print(f" Best Model: {best_model} (R² = {best_score:.4f})")

print(f" Models saved in: {MODEL_DIR_MODULE2}")
print(f" Training timestamp: {timestamp}")

print("\n Ready for Module 2 Integration!")
print("\n Next Steps:")
print("  1. Start the web application: python app/app_v2.py")
print("  2. Access Module 2 for instant predictions")
print("  3. Upload process data and get predictions in < 1 second")
print("\n" + "=" * 60)
```