Notebook

November 21, 2024

1 Regression Implementation

[356]:
```
# Import necessary libraries
try:
    import re
    import os
    import glob
    import torch
    import joblib
    import pandas as pd
    import numpy as np

    import seaborn as sns
    from scipy import stats
    from itertools import groupby
    import matplotlib.pyplot as plt
    import matplotlib.patches as mpatches
    from scipy.signal import savgol_filter

    from sklearn.pipeline import Pipeline
    from sklearn.impute import SimpleImputer
    from sklearn.base import BaseEstimator, RegressorMixin
    from sklearn.inspection import PartialDependenceDisplay
    from sklearn.preprocessing import (
        PolynomialFeatures,
        StandardScaler,
        MinMaxScaler,
        OrdinalEncoder,
        OneHotEncoder,
        LabelEncoder
    )
    from sklearn.compose import ColumnTransformer
    from sklearn.utils import class_weight
    from sklearn.linear_model import (
        LinearRegression, LogisticRegression        # Baseline models
    )
    from sklearn.ensemble import RandomForestRegressor
```

```
    from sklearn.cluster import KMeans
    from sklearn.model_selection import train_test_split
    from sklearn.metrics import (
        mean_absolute_error,
        mean_squared_error,
        root_mean_squared_error,
        classification_report,
        confusion_matrix,
        accuracy_score,
        silhouette_score,
        precision_score,
        recall_score,
        f1_score,
        r2_score
    )
    from xgboost import XGBRegressor
    from kneed import KneeLocator
    from imblearn.over_sampling import SMOTE

    # TensorFlow and Keras for the Neural Network
    import keras
    import tensorflow as tf
    from tensorflow.keras.models import Sequential                          #␣
 ↪type: ignore
    from tensorflow.keras import layers, models, regularizers               #␣
 ↪type: ignore
    from tensorflow.keras.layers import (                                   #␣
 ↪type: ignore
        Dense, Dropout, BatchNormalization, Input
    )
    from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau   #␣
 ↪type: ignore
    from tensorflow.keras.optimizers import Adam                            #␣
 ↪type: ignore

    # Set random seeds for reproducibility
    random_state = 42
    np.random.seed(random_state)
    tf.random.set_seed(random_state)

except Exception as e:
    print(f"Error importing libraries: {e}")
```

[357]:
```
# Set the model saving path
# TODO: Add Google Colab support
destination = '../Models/'                          # Destination folder for the␣
 ↪model
```

```
os.makedirs(destination, exist_ok=True)           # Create the folder if it␣
 ↪doesn't exist
print(f"Model will be saved to: {destination}")  # Output the model destination␣
 ↪folder
```

```
Model will be saved to: ../Models/
```

[358]:
```
# Define color palette for the plots
green = '#2ECC71'  # Green
blue = '#3498DB'   # Blue
red = '#E74C3C'    # Red
```

Function defintion Custom-made functions in this notebook offer several benefits that enhance the efficiency, readability, and maintainability of the code. By encapsulating specific tasks into functions, we achieve modularity, which allows for easier debugging and testing. Functions such as `regression_evaluation`, `load_dataset`, and `one_hot_encoding` streamline repetitive tasks, ensuring consistency and reducing the likelihood of errors. Additionally, these functions improve code reusability, enabling us to apply the same logic across different parts of the notebook without redundancy. This approach not only saves time but also makes the notebook more organized and easier to understand for collaborators and future reference.

[359]:
```
def regression_evaluation(model, X, y, message="Model", scale=False):
    if scale:
        scaler = StandardScaler()
        X = pd.DataFrame(
            scaler.fit_transform(X),
            columns=X.columns,
            index=X.index
        )

    # Get predictions - handle both sklearn and keras models
    if isinstance(model, tf.keras.Model):
        predictions = model.predict(X, verbose=0).flatten()  # Add flatten()␣
 ↪here
    else:
        predictions = model.predict(X)

    # Convert predictions to 1D array if needed
    if isinstance(predictions, np.ndarray) and predictions.ndim > 1:
        predictions = predictions.flatten()

    if isinstance(model, tf.keras.Model):
        predictions = model.predict(X, verbose=0)
        print(f"Raw predictions shape        : {predictions.shape}")
        predictions = predictions.flatten()
        print(f"Flattened predictions shape  : {predictions.shape}")
        print(f"Target shape                 : {y.shape}\n")
```

```
    # Calculate residuals
    residuals = y - predictions

    # Calculate metrics
    rmse = root_mean_squared_error(y, predictions)
    r2 = r2_score(y, predictions)
    mae = mean_absolute_error(y, predictions)


    title = f"--- {message} Evaluation Metrics ---"
    # Print evaluation metrics
    print(f"{title}")
    print(f"-"*len(title))
    print(f"    RMSE: {rmse:.2f}")
    print(f"    R²  : {r2:.2f}")
    print(f"    MAE : {mae:.2f}")
    print(f"-"*len(title))

    # Set style
    plt.style.use('default')

    # Create figure with three subplots
    fig = plt.figure(figsize=(20, 6))
    gs = fig.add_gridspec(1, 3)
    ax1 = fig.add_subplot(gs[0, 0])
    ax2 = fig.add_subplot(gs[0, 1])
    ax3 = fig.add_subplot(gs[0, 2])

    fig.suptitle(f'Regression Model Evaluation - {message}', fontsize=14, y=1.
↪05)

    # Color palette
    actual_color = '#2ECC71'  # Green
    pred_color = '#3498DB'    # Blue
    trend_color = '#E74C3C'   # Red

    # Plot 1: Actual vs Predicted
    line_x = np.linspace(y.min(), y.max(), 100)

    ax1.scatter(y, predictions, alpha=0.5, color=pred_color,␣
↪label='Predictions', s=100)
    ax1.plot(line_x, line_x, '--', color='gray', alpha=0.8, label='Perfect␣
↪Prediction', linewidth=2)

    z = np.polyfit(y, predictions, 1)
    p = np.poly1d(z)
```

```
    ax1.plot(line_x, p(line_x), '-', color=trend_color, alpha=0.8, label='Trend␣
↪Line')

    ax1.set_xlabel('Actual Values', fontsize=12)
    ax1.set_ylabel('Predicted Values', fontsize=12)
    ax1.set_title('Actual vs Predicted Values', fontsize=13, pad=15)
    ax1.legend(loc='upper left')
    ax1.grid(True, alpha=0.3)

    # Plot 2: Residuals
    ax2.scatter(predictions, residuals, alpha=0.5, color=pred_color, s=100)
    ax2.axhline(y=0, color='gray', linestyle='--', alpha=0.8)

    z = np.polyfit(predictions, residuals, 1)
    p = np.poly1d(z)
    ax2.plot(predictions, p(predictions), color=trend_color, alpha=0.8,␣
↪label='Trend Line')

    ax2.set_xlabel('Predicted Values', fontsize=12)
    ax2.set_ylabel('Residuals', fontsize=12)
    ax2.set_title('Residuals Plot', fontsize=13, pad=15)
    ax2.grid(True, alpha=0.3)
    ax2.legend()

    std_dev = np.std(residuals)
    ax2.axhline(y=std_dev, color=trend_color, linestyle=':', alpha=0.5,␣
↪label='+1 Std Dev')
    ax2.axhline(y=-std_dev, color=trend_color, linestyle=':', alpha=0.5,␣
↪label='-1 Std Dev')

    # Plot 3: Model Fit Visualisation
    # Sort by target values for smooth curve
    sort_idx = np.argsort(y)
    y_sorted = y.iloc[sort_idx]
    pred_sorted = predictions[sort_idx]

    # Create index for x-axis
    x_range = np.arange(len(y))

    # Plot actual and predicted values
    ax3.scatter(x_range, y_sorted, alpha=0.5, color=actual_color, label='Actual␣
↪Values', s=100)
    ax3.scatter(x_range, pred_sorted, alpha=0.5, color=pred_color,␣
↪label='Predictions', s=100)

    # Add smoothed curve through predictions
```

```
    window = len(y) // 20  # Adjust window size as needed
    if window % 2 == 0:
        window += 1
    smoothed = savgol_filter(pred_sorted, window, 3)
    ax3.plot(x_range, smoothed, '-', color=trend_color, alpha=0.8, label='Model␣
 ↪Fit', linewidth=2)

    ax3.set_xlabel('Ordered Samples', fontsize=12)
    ax3.set_ylabel('Values', fontsize=12)
    ax3.set_title('Model Fit Visualisation', fontsize=13, pad=15)
    ax3.legend(loc='upper left')
    ax3.grid(True, alpha=0.3)

    # Adjust layout and display
    plt.tight_layout()
    plt.show()

    del X, y, predictions, rmse, r2, mae # Clear memory
```

Example usage: `regression_evaluation(model, X_test_reg, y_test_reg, "Your message here")`

Key features of this evaluation function:

1. Supports optional feature scaling
2. Calculates key metrics (RMSE, R^2 and MAE)
3. Creates three different visualisations for model performance analysis
4. Uses consistent color coding (green for actual, blue for predictions, red for trend lines)
5. Includes error bands (± 1 standard deviation) in residuals plot
6. Uses **Savitzky-Golay** filtering for smoothing the model fit visualisation

This is a comprehensive evaluation function that provides both numerical metrics and visual analysis tools to understand model performance, residual patterns, and fit quality.

[360]:
```
# Utility function to load dataset
# TODO: Add Google Colab support
def load_dataset(data_path='../Datasets/*.csv'):       # Dynamically load␣
 ↪dataset from the Datasets directory
    file_list = glob.glob(data_path)
    if len(file_list) == 1:
        df = pd.read_csv(file_list[0])
        print(f"Loaded dataset: {file_list[0]}")
    else:
        raise FileNotFoundError("No CSV file found or multiple CSV files found␣
 ↪in the Datasets directory.")
    return df
```

[361]:
```
# Function to rename columns ending with '^{1}' suffix from PolynomialFeatures
def rename_columns(df):
```

```
    new_columns = {col: col.replace('^{1}', '') for col in df.columns if col.
↪endswith('^{1}')}
    df = df.rename(columns=new_columns)
    return df
```

[362]:
```
# Load the dataset
df = load_dataset()
```

```
Loaded dataset: ../Datasets/Dataset.csv
```

[363]:
```
df.head()
```

[363]:

	Lifespan	partType	microstructure	coolingRate	quenchTime	forgeTime	\
0	1469.17	Nozzle	equiGrain	13	3.84	6.47	
1	1793.64	Block	singleGrain	19	2.62	3.48	
2	700.60	Blade	equiGrain	28	0.76	1.34	
3	1082.10	Nozzle	colGrain	9	2.01	2.19	
4	1838.83	Blade	colGrain	16	4.13	3.87	

	HeatTreatTime	Nickel%	Iron%	Cobalt%	Chromium%	smallDefects	\
0	46.87	65.73	16.52	16.82	0.93	10	
1	44.70	54.22	35.38	6.14	4.26	19	
2	9.54	51.83	35.95	8.81	3.41	35	
3	20.29	57.03	23.33	16.86	2.78	0	
4	16.13	59.62	27.37	11.45	1.56	10	

	largeDefects	sliverDefects	seedLocation	castType
0	0	0	Bottom	Die
1	0	0	Bottom	Investment
2	3	0	Bottom	Investment
3	1	0	Top	Continuous
4	0	0	Top	Die

[364]:
```
df.describe()
```

[364]:

	Lifespan	coolingRate	quenchTime	forgeTime	HeatTreatTime	\
count	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000	
mean	1298.556320	17.639000	2.764230	5.464600	30.194510	
std	340.071434	7.491783	1.316979	2.604513	16.889415	
min	417.990000	5.000000	0.500000	1.030000	1.030000	
25%	1047.257500	11.000000	1.640000	3.170000	16.185000	
50%	1266.040000	18.000000	2.755000	5.475000	29.365000	
75%	1563.050000	24.000000	3.970000	7.740000	44.955000	
max	2134.530000	30.000000	4.990000	10.000000	59.910000	

	Nickel%	Iron%	Cobalt%	Chromium%	smallDefects	\
count	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000	
mean	60.243080	24.553580	12.434690	2.768650	17.311000	

```
std       5.790475     7.371737     4.333197     1.326496     12.268365
min      50.020000     6.660000     5.020000     0.510000      0.000000
25%      55.287500    19.387500     8.597500     1.590000      7.000000
50%      60.615000    24.690000    12.585000     2.865000     18.000000
75%      65.220000    29.882500    16.080000     3.922500     26.000000
max      69.950000    43.650000    19.990000     4.990000     61.000000

       largeDefects  sliverDefects
count   1000.000000    1000.000000
mean       0.550000       0.292000
std        1.163982       1.199239
min        0.000000       0.000000
25%        0.000000       0.000000
50%        0.000000       0.000000
75%        0.000000       0.000000
max        4.000000       8.000000
```

[365]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 16 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Lifespan        1000 non-null   float64
 1   partType        1000 non-null   object
 2   microstructure  1000 non-null   object
 3   coolingRate     1000 non-null   int64
 4   quenchTime      1000 non-null   float64
 5   forgeTime       1000 non-null   float64
 6   HeatTreatTime   1000 non-null   float64
 7   Nickel%         1000 non-null   float64
 8   Iron%           1000 non-null   float64
 9   Cobalt%         1000 non-null   float64
 10  Chromium%       1000 non-null   float64
 11  smallDefects    1000 non-null   int64
 12  largeDefects    1000 non-null   int64
 13  sliverDefects   1000 non-null   int64
 14  seedLocation    1000 non-null   object
 15  castType        1000 non-null   object
dtypes: float64(8), int64(4), object(4)
memory usage: 125.1+ KB
```

[366]:
```
df.shape
```

[366]: (1000, 16)

[367]:
```
# Check for missing values
df.isnull().sum()
```

[367]:
```
Lifespan          0
partType          0
microstructure    0
coolingRate       0
quenchTime        0
forgeTime         0
HeatTreatTime     0
Nickel%           0
Iron%             0
Cobalt%           0
Chromium%         0
smallDefects      0
largeDefects      0
sliverDefects     0
seedLocation      0
castType          0
dtype: int64
```

[368]:
```
# Using nunique()
num_parts = df['partType'].nunique()
print(f"Number of unique parts types: {num_parts}")

# Or using value_counts() to see the distribution
parts_distribution = df['partType'].value_counts()
print("\nDistribution of parts types:")
print(parts_distribution)
```

```
Number of unique parts types: 4

Distribution of parts types:
partType
Valve     265
Block     253
Nozzle    245
Blade     237
Name: count, dtype: int64
```

[369]:
```
unique_lifespan_count = df['Lifespan'].nunique()
print(f"Number of unique values in 'Lifespan' column: {unique_lifespan_count}")
```

```
Number of unique values in 'Lifespan' column: 998
```

[370]:
```
# Find the unique values that occur more than once in the 'Lifespan' column
duplicate_values = df['Lifespan'].value_counts()
duplicate_values = duplicate_values[duplicate_values > 1]
```

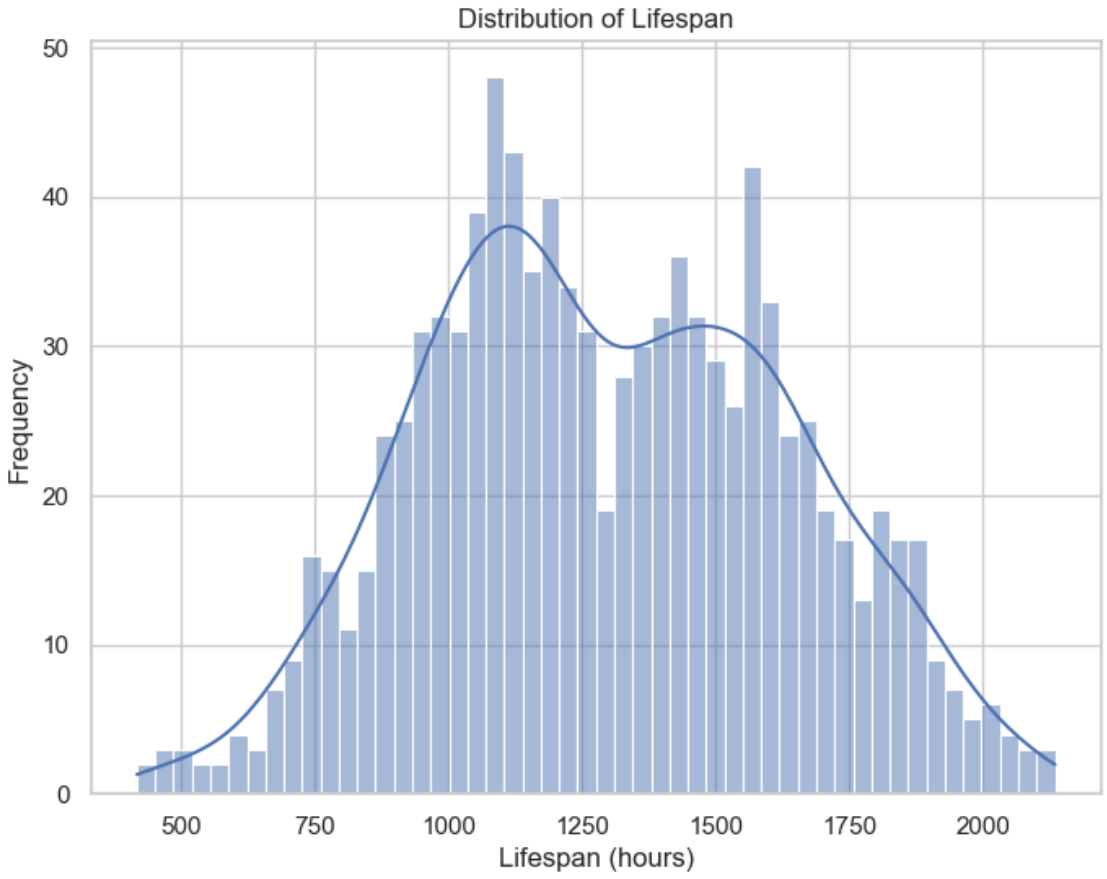
```
# Output the unique values and their counts
print(duplicate_values)
```

```
Lifespan
1262.14    2
932.69     2
Name: count, dtype: int64
```

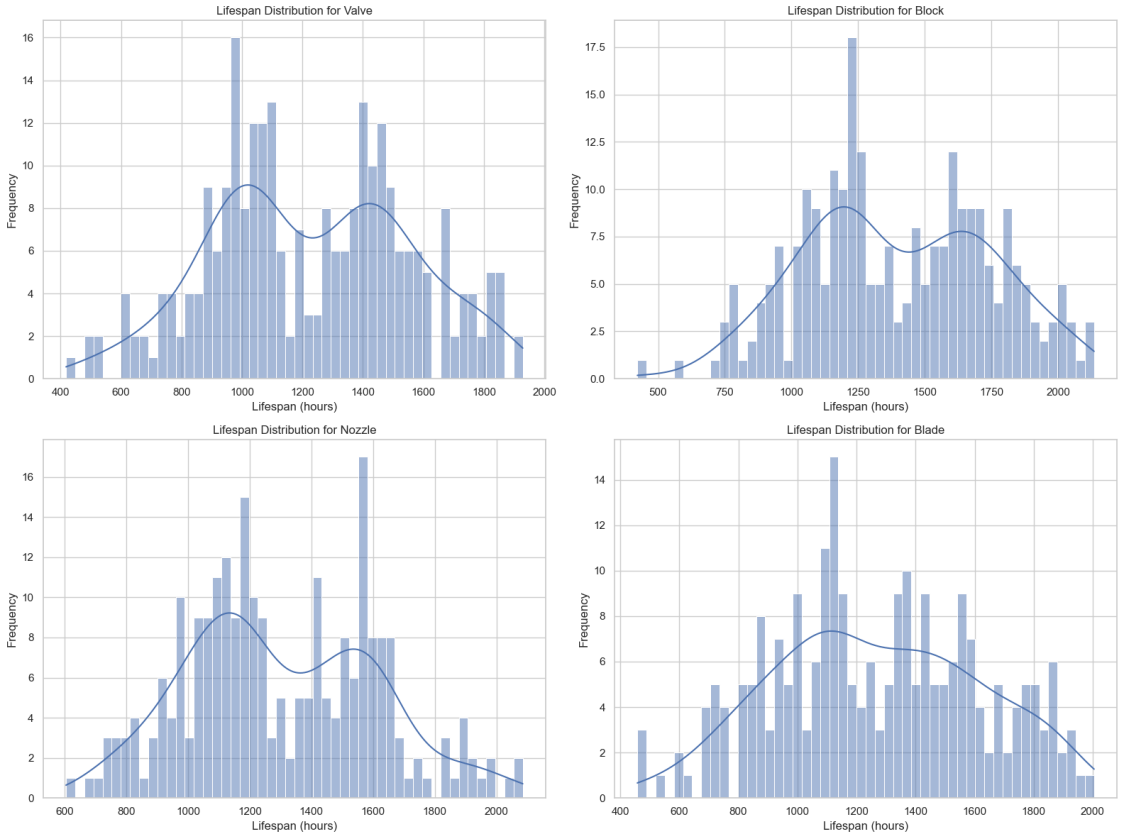
We have found that there are 4 disticnt metal parts in this dataset.

1.0.1 Distribution of Lifespan

[371]:
```
bins = 50

# Plotting the distribution of 'Lifespan'
plt.figure(figsize=(8, 6))
sns.histplot(df['Lifespan'], bins=bins, kde=True)
plt.title('Distribution of Lifespan')
plt.xlabel('Lifespan (hours)')
plt.ylabel('Frequency')
plt.show()
```

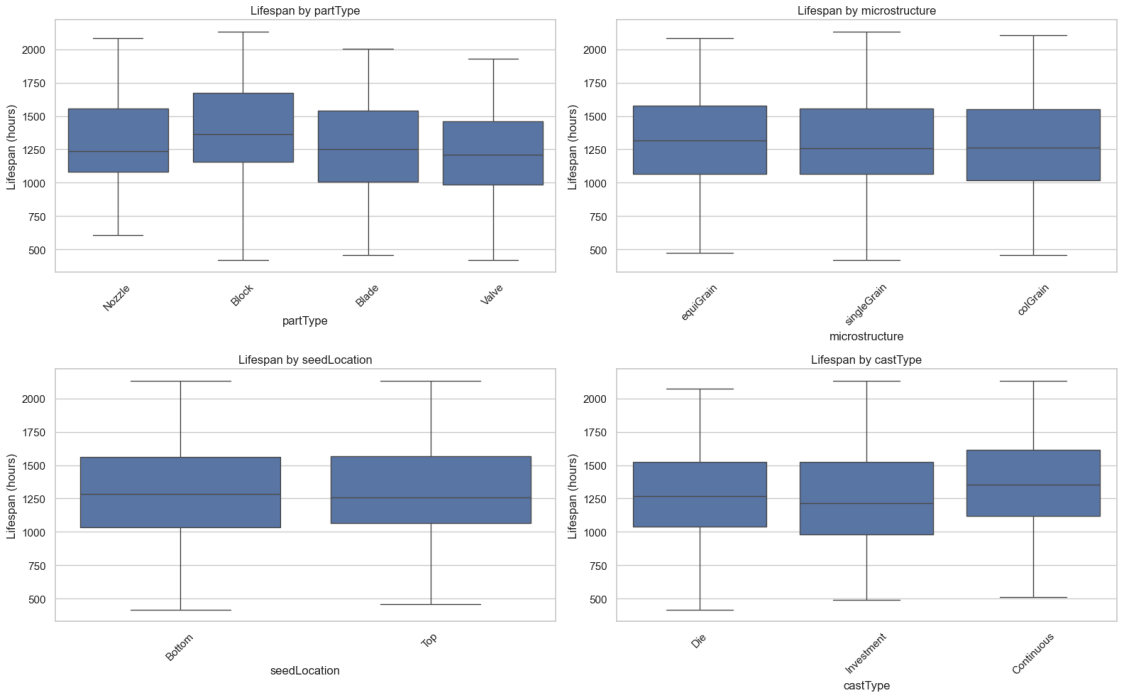


[372]:
```
# List of metal parts
part_types = ['Valve', 'Block', 'Nozzle', 'Blade']

# Set up the subplots: 2 rows and 2 columns (one for each metal part)
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(16, 12))
axes = axes.flatten()  # Flatten the axes array for easier iteration

# Loop through each metal part and create a histogram for its lifespan␣
 ↪distribution
for i, part in enumerate(part_types):
    subset = df[df['partType'] == part]
    sns.histplot(subset['Lifespan'], bins=bins, kde=True, ax=axes[i])
    axes[i].set_title(f'Lifespan Distribution for {part}')
    axes[i].set_xlabel('Lifespan (hours)')
    axes[i].set_ylabel('Frequency')

# Adjust layout for better visualisation
plt.tight_layout()
plt.show()
```

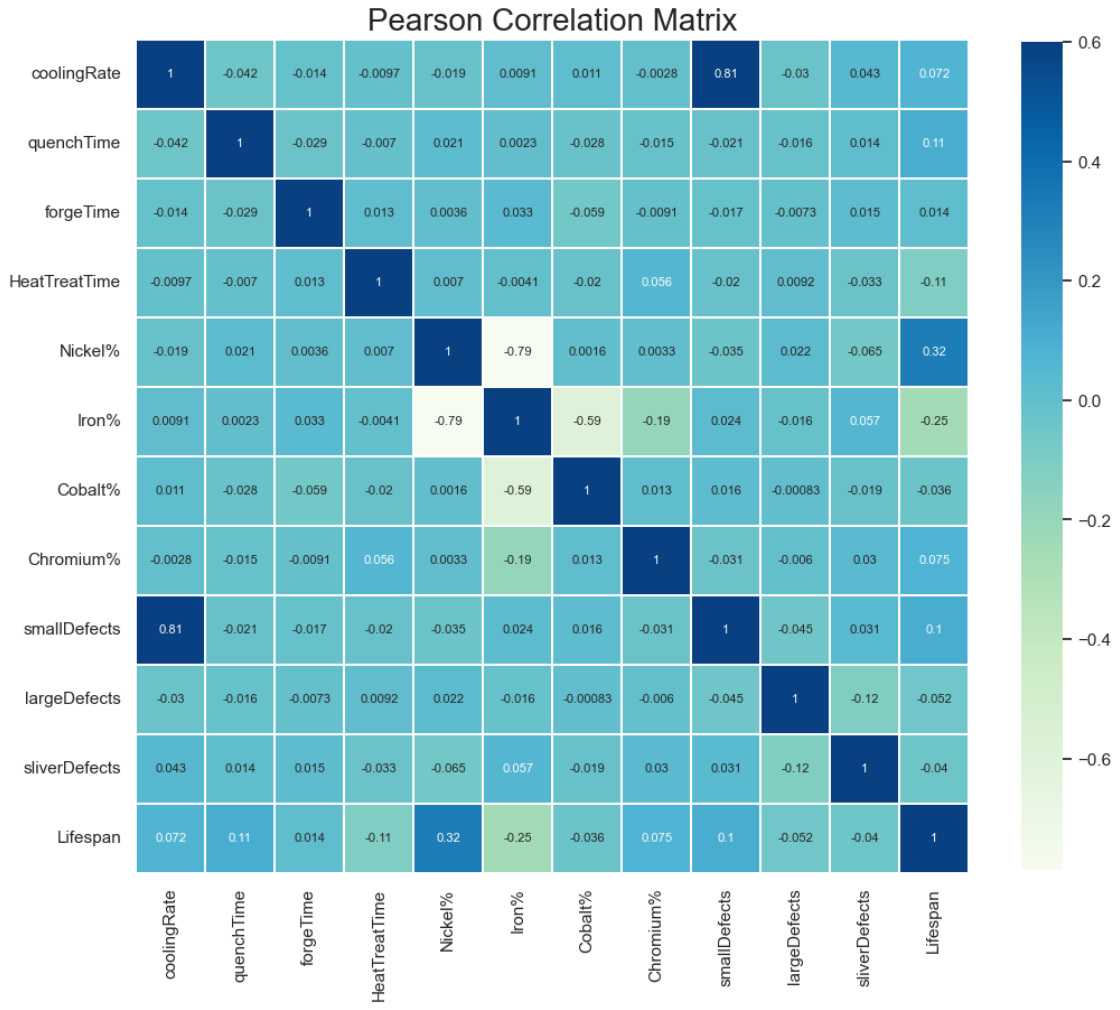


[373]:
```
# List of numerical columns excluding 'Lifespan'
numerical_features = [col for col in df.columns if df[col].dtype in ['int64',␣
 ↪'float64'] and col != 'Lifespan']
categorical_features = df.select_dtypes(include=['object']).columns.tolist()
```

[374]:
```
# Determine the number of rows and columns for the subplots
num_cols = len(categorical_features)
nrows = num_cols // 2 + (num_cols % 2 > 0)  # Set up for 2 columns per row for␣
 ↪better readability

# Create the subplots
fig, axes = plt.subplots(nrows=nrows, ncols=2, figsize=(16, nrows * 5))
axes = axes.flatten()  # Flatten the axes array to easily iterate over

# Plot each boxplot in a different subplot
for i, col in enumerate(categorical_features):
    sns.boxplot(ax=axes[i], x=col, y='Lifespan', data=df)
    axes[i].set_title(f'Lifespan by {col}')
    axes[i].set_xlabel(col)
    axes[i].set_ylabel('Lifespan (hours)')
    axes[i].tick_params(axis='x', rotation=45)

# Remove any empty subplots if the number of features is odd
for j in range(len(categorical_features), len(axes)):
    fig.delaxes(axes[j])

# Adjust layout for better visualisation
plt.tight_layout()
plt.show()
```

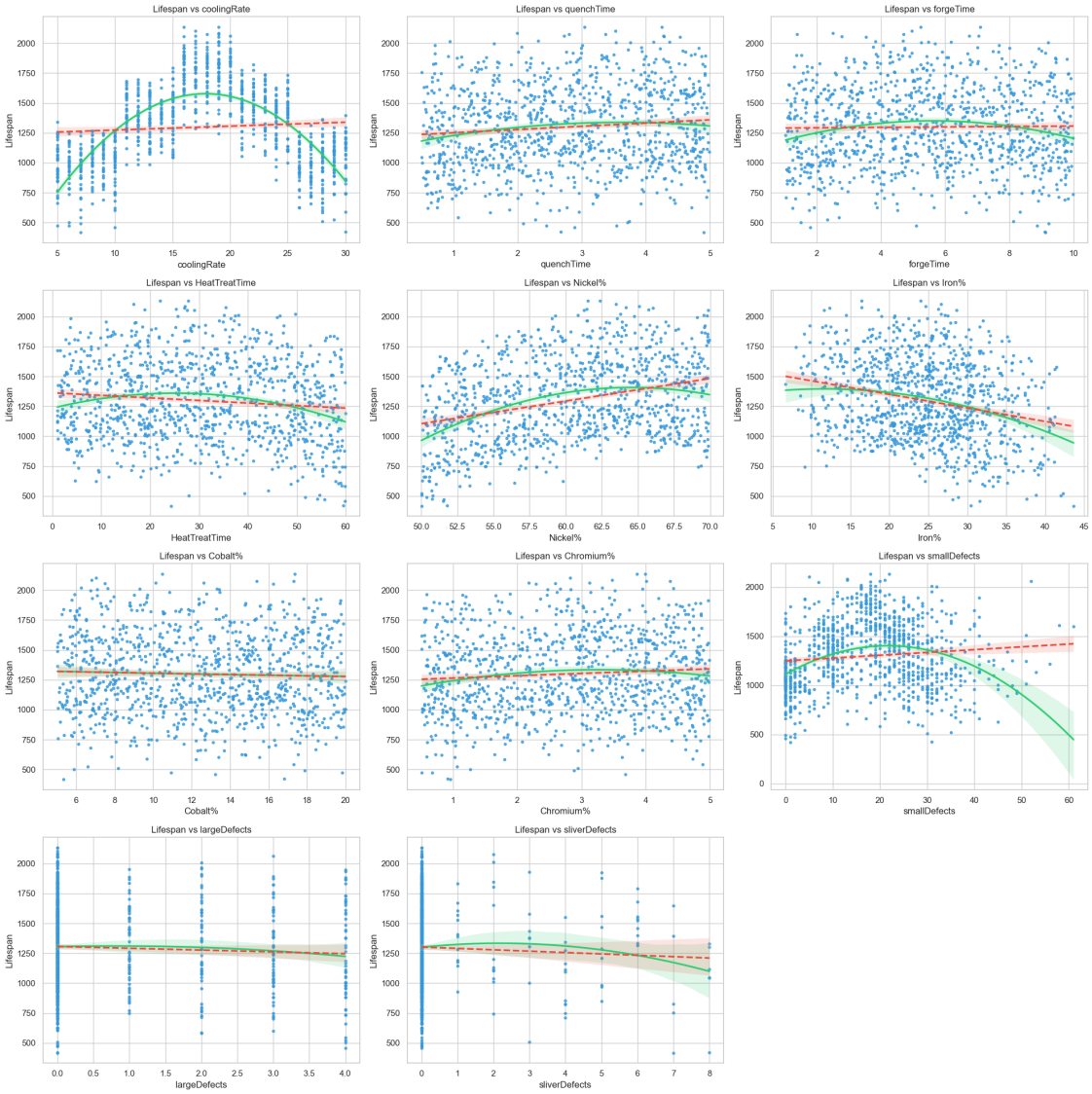


[375]:
```
# Include 'Lifespan' in the correlation matrix
numerical_cols_with_target = numerical_features + ['Lifespan']
corr_matrix = df[numerical_cols_with_target].corr()

sns.set_theme(style="whitegrid", font_scale=1)

plt.figure(figsize=(12,12))
plt.title('Pearson Correlation Matrix',fontsize=20)
sns.heatmap(corr_matrix,linewidths=0.25,vmax=0.
  ↪6,square=True,cmap="GnBu",linecolor='w',
            annot=True, annot_kws={"size":8}, cbar_kws={"shrink": .8})
plt.show()
```



[376]:
```
# Determine the number of rows and columns for the subplots
num_cols = len(numerical_features)
nrows = num_cols // 3 + (num_cols % 3 > 0)  # Set up for 3 columns per row

# Create the subplots
fig, axes = plt.subplots(nrows=nrows, ncols=3, figsize=(20, nrows * 5))
axes = axes.flatten()  # Flatten to easily iterate over

# Plot each scatterplot with a polynomial regression trend line in a different␣
↪subplot
for i, col in enumerate(numerical_features):
    sns.regplot(
        ax=axes[i],
        data=df,
        x=col,
        y='Lifespan',
```

```
        scatter_kws={'s': 10, 'color': blue},  # Set scatter plot color to blue
        line_kws={'color': green},
        order=2  # Using polynomial order 2 to create a curved trend line
    )
    axes[i].set_title(f'Lifespan vs {col}')
    axes[i].set_xlabel(col)
    axes[i].set_ylabel('Lifespan (hours)')

    # Linear regression line
    sns.regplot(ax=axes[i], data=df, x=col, y='Lifespan', scatter=False,␣
 ↪line_kws={'color': red, 'linestyle': '--'})

# Remove any empty subplots if the number of features is not perfectly␣
 ↪divisible by 3
for j in range(len(numerical_features), len(axes)):
    fig.delaxes(axes[j])

# Adjust layout for better visualisation
plt.tight_layout()
plt.show()
```



There is a parabolic (U-shaped) trend visible, which confirms your earlier findings. This suggests that there is an optimal cooling rate where the lifespan reaches a peak, beyond which the lifespan decreases. A quadratic relationship fits well here.

[377]:
```
# Define features to transform
features_to_transform = {
    'coolingRate': 7,
    'forgeTime': 2,
    'HeatTreatTime': 2,
    'smallDefects': 3,
    'sliverDefects': 2
}
```

```python
# Keep a copy of original features
df_transformed = df.copy()

# Transform features using PolynomialFeatures
for feature, degree in features_to_transform.items():
    # Create polynomial features
    poly = PolynomialFeatures(degree=degree, include_bias=False)
    poly_features = poly.fit_transform(df[[feature]])

    # Get column names for the new polynomial features
    poly_feature_names = [f"{feature}^{{{i}}}" for i in range(1, degree + 1)]
    poly_df = pd.DataFrame(poly_features, columns=poly_feature_names, index=df.
 ↪index)

    # Drop the original feature and add polynomial features to the dataset
    df_transformed = df_transformed.drop(columns=[feature]).join(poly_df)

# Adding linear features that were not transformed back to the transformed␣
 ↪DataFrame
# Only add features that were not involved in polynomial transformation
linear_features = ['quenchTime', 'Nickel%', 'Iron%', 'Cobalt%', 'Chromium%',␣
 ↪'largeDefects']
df_transformed = df_transformed[linear_features + [col for col in␣
 ↪df_transformed.columns if col not in linear_features]]

# The df_transformed now contains both the polynomial and linear features␣
 ↪without redundancy
display(df_transformed.head())
display(df.head())
```

	quenchTime	Nickel%	Iron%	Cobalt%	Chromium%	largeDefects	Lifespan	\
0	3.84	65.73	16.52	16.82	0.93	0	1469.17	
1	2.62	54.22	35.38	6.14	4.26	0	1793.64	
2	0.76	51.83	35.95	8.81	3.41	3	700.60	
3	2.01	57.03	23.33	16.86	2.78	1	1082.10	
4	4.13	59.62	27.37	11.45	1.56	0	1838.83	

	partType	microstructure	seedLocation	…	coolingRate^{7}	forgeTime^{1}	\
0	Nozzle	equiGrain	Bottom	…	6.274852e+07	6.47	
1	Block	singleGrain	Bottom	…	8.938717e+08	3.48	
2	Blade	equiGrain	Bottom	…	1.349293e+10	1.34	
3	Nozzle	colGrain	Top	…	4.782969e+06	2.19	
4	Blade	colGrain	Top	…	2.684355e+08	3.87	

	forgeTime^{2}	HeatTreatTime^{1}	HeatTreatTime^{2}	smallDefects^{1}	\
0	41.8609	46.87	2196.7969	10.0	
1	12.1104	44.70	1998.0900	19.0	

```
2         1.7956              9.54           91.0116              35.0
3         4.7961             20.29          411.6841               0.0
4        14.9769             16.13          260.1769              10.0

   smallDefects^{2}  smallDefects^{3}  sliverDefects^{1}  sliverDefects^{2}
0             100.0            1000.0                0.0                0.0
1             361.0            6859.0                0.0                0.0
2            1225.0           42875.0                0.0                0.0
3               0.0               0.0                0.0                0.0
4             100.0            1000.0                0.0                0.0

[5 rows x 27 columns]

   Lifespan partType microstructure  coolingRate  quenchTime  forgeTime  \
0   1469.17   Nozzle      equiGrain           13        3.84       6.47
1   1793.64    Block    singleGrain           19        2.62       3.48
2    700.60    Blade      equiGrain           28        0.76       1.34
3   1082.10   Nozzle       colGrain            9        2.01       2.19
4   1838.83    Blade       colGrain           16        4.13       3.87

   HeatTreatTime  Nickel%  Iron%  Cobalt%  Chromium%  smallDefects  \
0          46.87    65.73  16.52    16.82       0.93            10
1          44.70    54.22  35.38     6.14       4.26            19
2           9.54    51.83  35.95     8.81       3.41            35
3          20.29    57.03  23.33    16.86       2.78             0
4          16.13    59.62  27.37    11.45       1.56            10

   largeDefects  sliverDefects seedLocation    castType
0             0              0       Bottom         Die
1             0              0       Bottom  Investment
2             3              0       Bottom  Investment
3             1              0          Top  Continuous
4             0              0          Top         Die
```

[378]:
```python
def print_transformed_columns_info(df_transformed):
    for col in df_transformed.columns:
        print(f"- {col}")
    print(f"Number of columns: {len(df_transformed.columns)}")

# Call the function to print the transformed columns info
print_transformed_columns_info(df_transformed)
```

```
- quenchTime
- Nickel%
- Iron%
- Cobalt%
- Chromium%
- largeDefects
- Lifespan
```

```
- partType
- microstructure
- seedLocation
- castType
- coolingRate^{1}
- coolingRate^{2}
- coolingRate^{3}
- coolingRate^{4}
- coolingRate^{5}
- coolingRate^{6}
- coolingRate^{7}
- forgeTime^{1}
- forgeTime^{2}
- HeatTreatTime^{1}
- HeatTreatTime^{2}
- smallDefects^{1}
- smallDefects^{2}
- smallDefects^{3}
- sliverDefects^{1}
- sliverDefects^{2}
Number of columns: 27
```

[379]:
```
df1_test = df_transformed[['coolingRate^{1}', 'forgeTime^{1}',␣
 ↪'HeatTreatTime^{1}', 'smallDefects^{1}', 'sliverDefects^{1}']]
df2_test = df_transformed[['coolingRate^{6}', 'forgeTime^{2}',␣
 ↪'HeatTreatTime^{2}', 'smallDefects^{3}', 'sliverDefects^{2}']]

display(df1_test.head())
display(df2_test.head())
```

	coolingRate^{1}	forgeTime^{1}	HeatTreatTime^{1}	smallDefects^{1}	\
0	13.0	6.47	46.87	10.0	
1	19.0	3.48	44.70	19.0	
2	28.0	1.34	9.54	35.0	
3	9.0	2.19	20.29	0.0	
4	16.0	3.87	16.13	10.0	

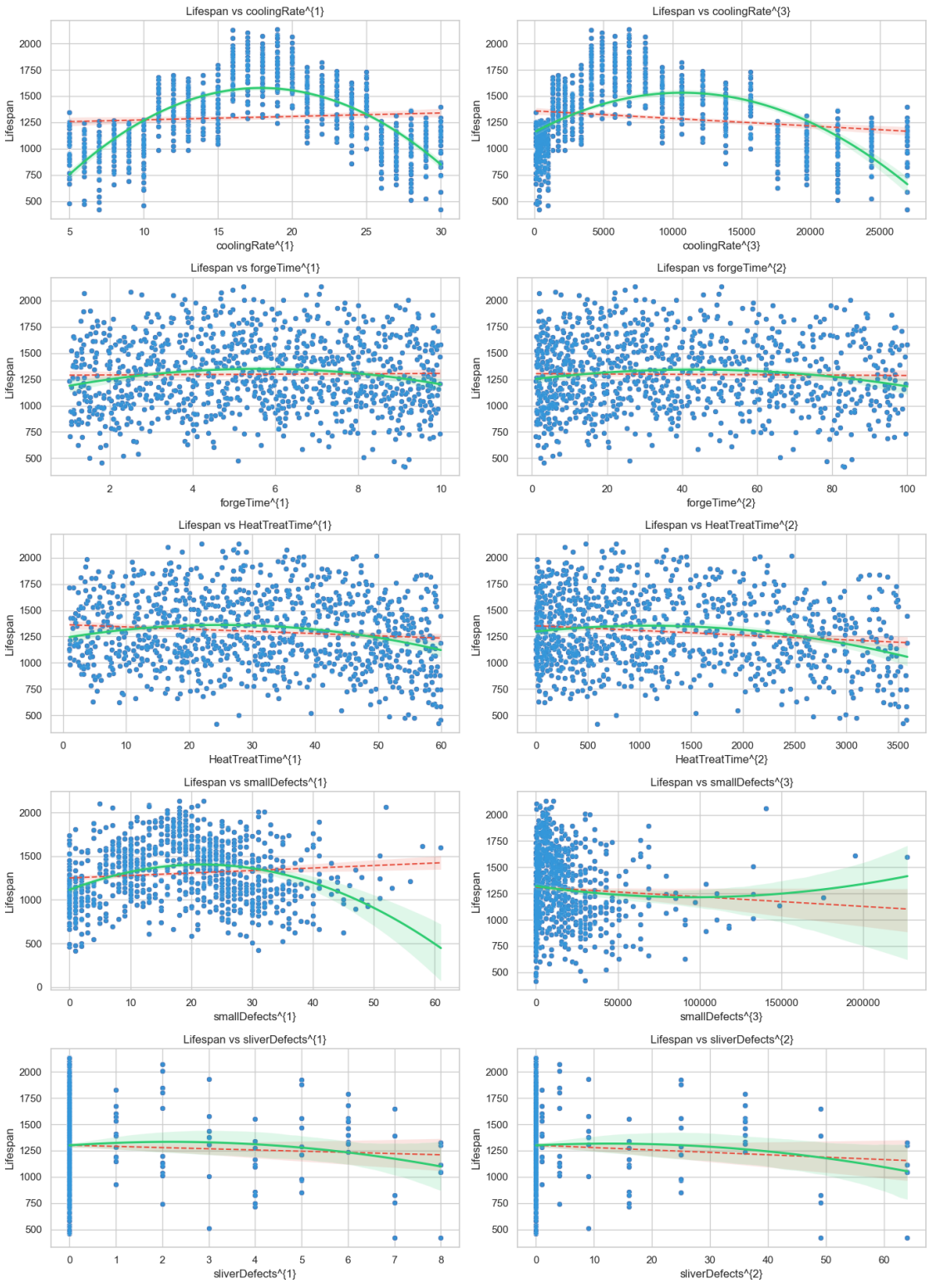
	sliverDefects^{1}
0	0.0
1	0.0
2	0.0
3	0.0
4	0.0

	coolingRate^{6}	forgeTime^{2}	HeatTreatTime^{2}	smallDefects^{3}	\
0	4826809.0	41.8609	2196.7969	1000.0	
1	47045881.0	12.1104	1998.0900	6859.0	
2	481890304.0	1.7956	91.0116	42875.0	

```
3         531441.0          4.7961           411.6841              0.0
4       16777216.0         14.9769           260.1769           1000.0

   sliverDefects^{2}
0                0.0
1                0.0
2                0.0
3                0.0
4                0.0
```

[380]:
```
# Scatter plots for the newly added polynomial features vs. Lifespan.

# List of newly created polynomial features
polynomial_features = [
    'coolingRate^{1}', 'coolingRate^{3}',
    'forgeTime^{1}', 'forgeTime^{2}',
    'HeatTreatTime^{1}', 'HeatTreatTime^{2}',
    'smallDefects^{1}', 'smallDefects^{3}',
    'sliverDefects^{1}', 'sliverDefects^{2}'
]

# Number of rows and columns for the plot grid
n_cols = 2
n_rows = len(polynomial_features) // n_cols + (len(polynomial_features) %␣
 ↪n_cols > 0)

# Create subplots
fig, axes = plt.subplots(n_rows, n_cols, figsize=(14, n_rows * 4))

# Plot each polynomial feature against 'Lifespan'
for i, feature in enumerate(polynomial_features):
    row, col = divmod(i, n_cols)
    sns.scatterplot(ax=axes[row, col], x=df_transformed[feature],␣
 ↪y=df_transformed['Lifespan'])
    sns.regplot(ax=axes[row, col], x=df_transformed[feature],␣
 ↪y=df_transformed['Lifespan'], scatter=False, color=red,␣
 ↪line_kws={"linewidth": 1.5, "linestyle":'--'})
    axes[row, col].set_title(f"Lifespan vs {feature}")
    sns.regplot(
        ax=axes[row, col],
        data=df_transformed,
        x=df_transformed[feature],
        y=df_transformed['Lifespan'],
        scatter_kws={'s': 10, 'color': blue},  # Set scatter plot color to blue
        line_kws={'color': green},
        order=2  # Using polynomial order 2 to create a curved trend line
    )
```

```
# Set the title for the entire figure
fig.suptitle("Polynomial Features Comparison", fontsize=16)
plt.tight_layout(rect=[0, 0, 1, 0.98])  # Adjust the rect parameter to make␣
 ↪space for the title
plt.show()
```


Polynomial Features Comparison
Lifespan vs coolingRate^{1}
Lifespan vs coolingRate^{3}
Lifespan vs forgeTime^{1}
Lifespan vs forgeTime^{2}
Lifespan vs HeatTreatTime^{1}
Lifespan vs HeatTreatTime^{2}
Lifespan vs smallDefects^{1}
Lifespan vs smallDefects^{3}
Lifespan vs sliverDefects^{1}
Lifespan vs sliverDefects^{2}
Lifespan
coolingRate^{1}
coolingRate^{3}
forgeTime^{1}
forgeTime^{2}
HeatTreatTime^{1}
HeatTreatTime^{2}
smallDefects^{1}
smallDefects^{3}
sliverDefects^{1}
sliverDefects^{2}

[439]:
```
# Define features to transform
features_to_transform = {    # Update if necessary
    'coolingRate': 2,        # Square the cooling rate
    # 'forgeTime': 2,        # Square the forge time
    # 'HeatTreatTime': 2,    # Square the heat treatment time
    'smallDefects': 2,       # Square the small defects
    # 'sliverDefects': 2     # Square the sliver defects
}

# Reset the dataframe
del df_transformed

# Keep a copy of original features
df_transformed = df.copy()

# Transform features using PolynomialFeatures
for feature, degree in features_to_transform.items():
    # Create polynomial features
    poly = PolynomialFeatures(degree=degree, include_bias=False)
    poly_features = poly.fit_transform(df[[feature]])

    # Get column names for the new polynomial features
    poly_feature_names = [f"{feature}^{{{i}}}" for i in range(1, degree + 1)]
    poly_df = pd.DataFrame(poly_features, columns=poly_feature_names, index=df.
 ↪index)

    # Drop the original feature and add polynomial features to the dataset
    df_transformed = df_transformed.drop(columns=[feature]).join(poly_df)
    new_columns = {col: col.replace('^{1}', '') for col in df_transformed.
 ↪columns if col.endswith('^{1}')}
    df_transformed = df_transformed.rename(columns=new_columns)

# Adding linear features that were not transformed back to the transformed␣
 ↪DataFrame
# Only add features that were not involved in polynomial transformation
linear_features = ['quenchTime', 'Nickel%', 'Iron%', 'Cobalt%', 'Chromium%',␣
 ↪'largeDefects']
df_transformed = df_transformed[linear_features + [col for col in␣
 ↪df_transformed.columns if col not in linear_features]]

# The df_transformed now contains both the polynomial and linear features␣
 ↪without redundancy
display(df_transformed.head())
display(df.head())
```

```
   quenchTime  Nickel%  Iron%  Cobalt%  Chromium%  largeDefects  Lifespan  \
0        3.84    65.73  16.52    16.82       0.93             0   1469.17
```

```
1        2.62    54.22  35.38     6.14       4.26             0   1793.64
2        0.76    51.83  35.95     8.81       3.41             3    700.60
3        2.01    57.03  23.33    16.86       2.78             1   1082.10
4        4.13    59.62  27.37    11.45       1.56             0   1838.83

  partType microstructure  forgeTime  HeatTreatTime  sliverDefects  \
0   Nozzle      equiGrain       6.47          46.87              0
1    Block    singleGrain       3.48          44.70              0
2    Blade      equiGrain       1.34           9.54              0
3   Nozzle       colGrain       2.19          20.29              0
4    Blade       colGrain       3.87          16.13              0

  seedLocation    castType  coolingRate  coolingRate^{2}  smallDefects  \
0       Bottom         Die         13.0            169.0          10.0
1       Bottom  Investment         19.0            361.0          19.0
2       Bottom  Investment         28.0            784.0          35.0
3          Top  Continuous          9.0             81.0           0.0
4          Top         Die         16.0            256.0          10.0

   smallDefects^{2}
0             100.0
1             361.0
2            1225.0
3               0.0
4             100.0

   Lifespan partType microstructure  coolingRate  quenchTime  forgeTime  \
0   1469.17   Nozzle      equiGrain           13        3.84       6.47
1   1793.64    Block    singleGrain           19        2.62       3.48
2    700.60    Blade      equiGrain           28        0.76       1.34
3   1082.10   Nozzle       colGrain            9        2.01       2.19
4   1838.83    Blade       colGrain           16        4.13       3.87

   HeatTreatTime  Nickel%  Iron%  Cobalt%  Chromium%  smallDefects  \
0          46.87    65.73  16.52    16.82       0.93            10
1          44.70    54.22  35.38     6.14       4.26            19
2           9.54    51.83  35.95     8.81       3.41            35
3          20.29    57.03  23.33    16.86       2.78             0
4          16.13    59.62  27.37    11.45       1.56            10

   largeDefects  sliverDefects seedLocation    castType
0             0              0       Bottom         Die
1             0              0       Bottom  Investment
2             3              0       Bottom  Investment
3             1              0          Top  Continuous
4             0              0          Top         Die
```

[382]:
```
print(f"Transformed Shape : {df_transformed.shape}")
print(f"Original Shape    : {df.shape}")
```

```
Transformed Shape : (1000, 18)
Original Shape    : (1000, 16)
```

[435]:
```
# Apply the function to the dataframe
reg_df = rename_columns(df_transformed)

def extract_features(df, target='Lifespan'):
    if target not in df.columns:
        target = 'Lifetime'

    categorical_features = df.select_dtypes(include=['object']).columns.tolist()
    totalnum_features = [col for col in df.columns if df[col].dtype in␣
 ↪['int64', 'float64'] and col != target]
    polynomial_features = [col for col in df.columns if re.
 ↪search(r'\^\{\d+\}$', col)]
    numerical_features = [col for col in totalnum_features if col not in␣
 ↪polynomial_features]

    print(f"Target feature: {target}\n")

    feature_types = [
        ("Categorical features", categorical_features),
        ("Numerical features", numerical_features),
        ("Polynomial features", polynomial_features)
    ]

    for feature_type, features in feature_types:
        print(f"{feature_type}: {len(features)}")
        for col in features:
            print(f"- {col}")
        print()   # Add a new line for better readability
    print("Total numerical features:", len(totalnum_features))
    display(df.head())

    return categorical_features, totalnum_features, polynomial_features,␣
 ↪numerical_features

categorical_features, totalnum_features, polynomial_features,␣
 ↪numerical_features = extract_features(reg_df, target='Lifespan')
```

```
Target feature: Lifespan

Categorical features: 4
- partType
- microstructure
```

```
- seedLocation
- castType

Numerical features: 11
- quenchTime
- Nickel%
- Iron%
- Cobalt%
- Chromium%
- largeDefects
- forgeTime
- HeatTreatTime
- sliverDefects
- coolingRate
- smallDefects

Polynomial features: 2
- coolingRate^{2}
- smallDefects^{2}

Total numerical features: 13
```

	quenchTime	Nickel%	Iron%	Cobalt%	Chromium%	largeDefects	Lifespan	\
0	3.84	65.73	16.52	16.82	0.93	0	1469.17	
1	2.62	54.22	35.38	6.14	4.26	0	1793.64	
2	0.76	51.83	35.95	8.81	3.41	3	700.60	
3	2.01	57.03	23.33	16.86	2.78	1	1082.10	
4	4.13	59.62	27.37	11.45	1.56	0	1838.83	

	partType	microstructure	forgeTime	HeatTreatTime	sliverDefects	\
0	Nozzle	equiGrain	6.47	46.87	0	
1	Block	singleGrain	3.48	44.70	0	
2	Blade	equiGrain	1.34	9.54	0	
3	Nozzle	colGrain	2.19	20.29	0	
4	Blade	colGrain	3.87	16.13	0	

	seedLocation	castType	coolingRate	coolingRate^{2}	smallDefects	\
0	Bottom	Die	13.0	169.0	10.0	
1	Bottom	Investment	19.0	361.0	19.0	
2	Bottom	Investment	28.0	784.0	35.0	
3	Top	Continuous	9.0	81.0	0.0	
4	Top	Die	16.0	256.0	10.0	

	smallDefects^{2}
0	100.0
1	361.0
2	1225.0
3	0.0

```
4                100.0
```

To find the best encoding for the this dataset, I have developed a Python script that iterates through all the possible encodings and calculates the accuracy of each encoding. The script then returns the encoding that has the highest accuracy. This Encoding.py script is available in the Solutions folder of this repository. This specific script is using a **Random Forest Regressor** model to identify whether the data work best with ordinal or non-ordinal encoding or hybrid approach.

[437]:
```
encoder_ohe = OneHotEncoder(handle_unknown='ignore', sparse_output=False,␣
 ↪dtype=int, drop=None)
encoder_le = LabelEncoder()
encoder_ord = OrdinalEncoder()
imputer_num = SimpleImputer(strategy='median')
imputer_cat = SimpleImputer(strategy='constant', fill_value='missing')
scaler_std = StandardScaler()
scaler_mnm = MinMaxScaler()
scaler_poly = PolynomialFeatures(degree=degree, include_bias=False)
```

[440]:
```
# Define features to transform along with their degrees
features_to_transform = {
    'coolingRate': 2,    # Square the 'coolingRate' feature
    'smallDefects': 2,   # Square the 'smallDefects' feature
    # Add or remove features and degrees as needed
}
```

[438]:
```
def create_preprocessor(encoding_method='onehot', include_polynomial=False,␣
 ↪poly_degree=2):
    # Numerical Transformer
    numeric_transformer = Pipeline(steps=[
        ('imputer', imputer_num),       # Impute missing numerical values
        ('scaler', scaler_std)          # Scale numerical features
    ])

    # Include Polynomial Features if specified
    if include_polynomial:
        numeric_transformer.steps.append(
            ('poly', PolynomialFeatures(degree=poly_degree, include_bias=False))
        )

    # Handling Encoding Methods
    if encoding_method == 'onehot':
        # One-Hot Encoding for all categorical features
        categorical_transformer = Pipeline(steps=[
            ('imputer', imputer_cat),    # Impute missing categorical values
            ('onehot', encoder_ohe)      # One-Hot Encode categorical features
        ])

        preprocessor = ColumnTransformer(
```

```
            transformers=[
                ('num', numeric_transformer, totalnum_features),
                ('cat', categorical_transformer, categorical_features)
            ],
            verbose_feature_names_out=False
        )

    elif encoding_method == 'ordinal':
        # Ordinal Encoding for all categorical features
        categorical_transformer = Pipeline(steps=[
            ('imputer', imputer_cat),
            ('ordinal', encoder_ord)
        ])

        preprocessor = ColumnTransformer(
            transformers=[
                ('num', numeric_transformer, totalnum_features),
                ('cat', categorical_transformer, categorical_features)
            ],
            verbose_feature_names_out=False
        )

    elif encoding_method == 'hybrid':
        # Hybrid Encoding: Ordinal for key features, One-Hot for others
        key_features = ['partType']  # Features to ordinally encode
        other_categorical_features = [
            col for col in categorical_features if col not in key_features
        ]

        # Transformer for 'partType' using Ordinal Encoding
        part_type_transformer = Pipeline(steps=[
            ('imputer', imputer_cat),
            ('ordinal', encoder_ord)
        ])

        # Transformer for other categorical features using One-Hot Encoding
        other_categorical_transformer = Pipeline(steps=[
            ('imputer', imputer_cat),
            ('onehot', encoder_ohe)
        ])

        preprocessor = ColumnTransformer(
            transformers=[
                ('num', numeric_transformer, totalnum_features),
                ('part_type', part_type_transformer, key_features),
                ('cat', other_categorical_transformer,␣
↪other_categorical_features)
```

```
            ],
            verbose_feature_names_out=False
        )

    else:
        raise ValueError("Unsupported encoding method. Choose 'onehot',␣
 ↪'ordinal', or 'hybrid'.")

    return preprocessor
```

[ ]:

[ ]:
```
# OneHotEncoder
# Preprocessing for numeric features
numeric_transformer = Pipeline(steps=[
    ('imputer', imputer_num),         # Impute missing values if any present in␣
 ↪the dataset
    ('scaler', scaler_std)
])

# Preprocessing for categorical features
categorical_transformer_onehot = Pipeline(steps=[
    ('imputer', imputer_cat),         # Impute missing values if any present in␣
 ↪the dataset
    ('onehot', encoder_ohe)
])

# Combine preprocessing steps
preprocessor_onehot = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, totalnum_features),
        ('cat', categorical_transformer_onehot, categorical_features)
    ],
    verbose_feature_names_out=False
)

# OrdinalEncoder
# Preprocessing for categorical features using OrdinalEncoder
categorical_transformer_label = Pipeline(steps=[
    ('imputer', imputer_cat),         # Impute missing values if any present in␣
 ↪the dataset
    ('ordinal_encoder', encoder_ord)
])

# Combine preprocessing steps
preprocessor_ordinal = ColumnTransformer(
    transformers=[
```

```
        ('num', numeric_transformer, totalnum_features),
        ('cat', categorical_transformer_label, categorical_features)
    ],
    verbose_feature_names_out=False
)

# Hybrid Approach
# Identify 'partType' and other categorical features
key_features = ['partType']
other_categorical_features = [col for col in categorical_features if col not in␣
 ↪key_features]

# Preprocessing pipeline for 'partType' using Label Encoding
part_type_transformer = Pipeline(steps=[
    ('imputer', imputer_cat),
    ('ordinal_encoder', encoder_ord)
])

# Preprocessing pipeline for other categorical features using One-Hot Encoding
other_categorical_transformer = Pipeline(steps=[
    ('imputer', imputer_cat),
    ('onehot', encoder_ohe)
])

# Combine preprocessing steps
preprocessor_hybrid = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, totalnum_features),
        ('non_poly', numeric_transformer, numerical_features),
        ('part_type', part_type_transformer, key_features),
        ('cat', other_categorical_transformer, other_categorical_features)
    ],
    verbose_feature_names_out=False
)
```

[386]:
```
# Define the script file with relative path
script_file = 'Encoding.py'

def execute_script(script_file):
    if os.path.exists(script_file):
        # Execute the script using exec() or by importing it
        with open(script_file) as script_file:
            exec(script_file.read())
    else:
        print(f"Script {script_file} not found. Moving onto the next block.")

# Call the function
```

```
# execute_script(script_file)  # Uncomment to execute the script
```

From the above, we can see that the Random Forest Regressor model using `One-Hot` encoding has shown the best performance with an R^2 score of 0.94, RMSE of 81.78 and MAE of 64.77. Therefore we can conclude that the `One-Hot` encoding is the best encoding method for this dataset.

[387]:
```
# Bin 'Lifespan' into quartiles
reg_df['Lifespan_bin'] = pd.qcut(reg_df['Lifespan'], q=4, duplicates='drop')

# First, split into training and temp (validation + test)
reg_train_df, temp_df = train_test_split(
    reg_df,
    test_size=0.3,
    stratify=reg_df['Lifespan_bin'],
    random_state=random_state
)

# Then, split temp_df equally into validation and test sets
reg_val_df, reg_test_df = train_test_split(
    temp_df,
    test_size=0.5,
    stratify=temp_df['Lifespan_bin'],
    random_state=random_state
)

# Remove the temporary DataFrame
del temp_df

# Optionally, drop the temporary columns
reg_train_df = reg_train_df.drop(columns=['Lifespan_bin'])
reg_val_df = reg_val_df.drop(columns=['Lifespan_bin'])
reg_test_df = reg_test_df.drop(columns=['Lifespan_bin'])

train_shape, val_shape, test_shape = reg_train_df.shape, reg_val_df.shape,␣
 ↪reg_test_df.shape
print(f"Training set shape   : {train_shape}")
print(f"Validation set shape : {val_shape}")
print(f"Test set shape       : {test_shape}")
```

```
Training set shape   : (700, 18)
Validation set shape : (150, 18)
Test set shape       : (150, 18)
```

Here, we are splitting the data into common train, validation and test sets with split ratio of 0.7, 0.15 and 0.15 respectively. We are also using stratification to ensure that the distribution of the target variable is similar in all the sets especially in the train set. Furthermore, we have two additional features which extracted using the polynomial features of `coolingRate` and `smallDefects` features.

[388]:
```
reg_train_df.isnull().sum()
```

[388]:
```
quenchTime          0
Nickel%             0
Iron%               0
Cobalt%             0
Chromium%           0
largeDefects        0
Lifespan            0
partType            0
microstructure      0
forgeTime           0
HeatTreatTime       0
sliverDefects       0
seedLocation        0
castType            0
coolingRate         0
coolingRate^{2}     0
smallDefects        0
smallDefects^{2}    0
dtype: int64
```

[389]:
```
# Features (X) and target (y)
X_train_reg = reg_train_df.drop(columns=['Lifespan'])   # Features excluding␣
 ↪the target variable
y_train_reg = reg_train_df['Lifespan']                  # Target variable

# Features (X) and target (y)
X_test_reg = reg_test_df.drop(columns=['Lifespan'])     # Features excluding␣
 ↪the target variable
y_test_reg = reg_test_df['Lifespan']                    # Target variable

# Features (X) and target (y)
X_val_reg = reg_val_df.drop(columns=['Lifespan'])       # Features excluding␣
 ↪the target variable
y_val_reg = reg_val_df['Lifespan']                      # Target variable

# Saving the dataset without polynomial features
nonpoly_columns = [col for col in X_train_reg.columns if col not in␣
 ↪polynomial_features]

X_train_reg_nonpoly = X_train_reg[nonpoly_columns]
X_test_reg_nonpoly = X_test_reg[nonpoly_columns]
X_val_reg_nonpoly = X_val_reg[nonpoly_columns]
```

[398]:
```
# Initialise the Linear Regression model
LinRegModel = LinearRegression(n_jobs=-1)
```

Here the parameter `n_jobs` is set to `-1`, which means that all available cores will be used for parallel processing.

[399]:
```
def create_pipeline(model, encoding_type):
    if encoding_type == 'onehot':
        preprocessor = preprocessor_onehot
    elif encoding_type == 'label':
        preprocessor = preprocessor_ordinal
    elif encoding_type == 'hybrid':
        preprocessor = preprocessor_hybrid
    else:
        raise ValueError(f"Unknown encoding type: {encoding_type}")

    pipeline = Pipeline(steps=[
        ('preprocessor', preprocessor),
        ('model', model)
    ])
    return pipeline
```

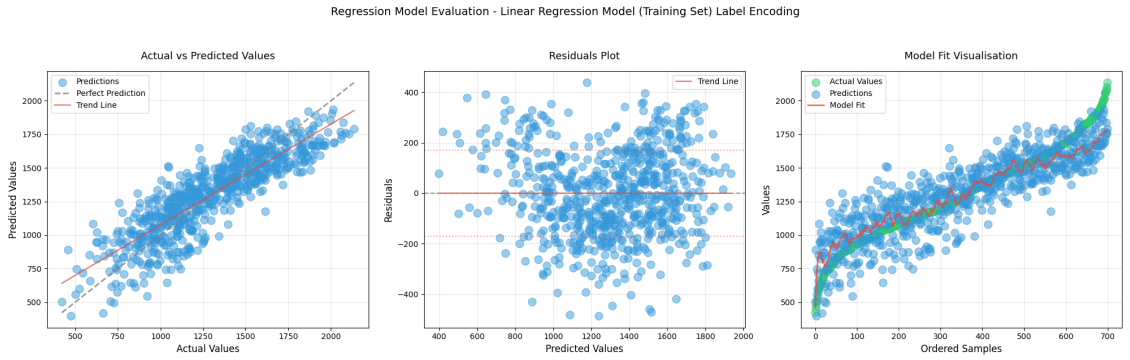
[422]:
```
# Create a pipeline for the Linear Regression model
LinRegPipeline = create_pipeline(
    LinRegModel,    # Main model to be pipelined
    'onehot'        # # TODO: Change to 'label', 'hybrid' or 'onehot' for performance comparison
)

# Fit the pipelined model
LinRegPipeline.fit(
    X_train_reg,
    y_train_reg
)
```

[422]:
```
Pipeline(steps=[('preprocessor',
                 ColumnTransformer(transformers=[('num',
                                                  Pipeline(steps=[('imputer',
SimpleImputer(strategy='median')),
                                                                  ('scaler',
StandardScaler())]),
                                                  ['quenchTime', 'Nickel%',
                                                   'Iron%', 'Cobalt%',
                                                   'Chromium%', 'largeDefects',
                                                   'forgeTime', 'HeatTreatTime',
                                                   'sliverDefects',
                                                   'coolingRate',
                                                   'coolingRate^{2}',
                                                   'smallDefects',
                                                   'smallDefects^{2}']),
```

```
                                                  ('cat',
                                                   Pipeline(steps=[('imputer',
SimpleImputer(fill_value='missing',
 strategy='constant')),
                                                                   ('onehot',
OneHotEncoder(dtype=<class 'int'>,
 handle_unknown='ignore',
 sparse_output=False))]),
                                                   ['partType', 'microstructure',
                                                    'seedLocation',
                                                    'castType'])],
                                      verbose_feature_names_out=False)),
                    ('model', LinearRegression(n_jobs=-1))])
```

[423]:
```
# Evaluate the Linear Regression model using custom function
regression_evaluation(
    LinRegPipeline,
    X_train_reg,
    y_train_reg,
    "Linear Regression Model (Training Set) Label Encoding"
)
```

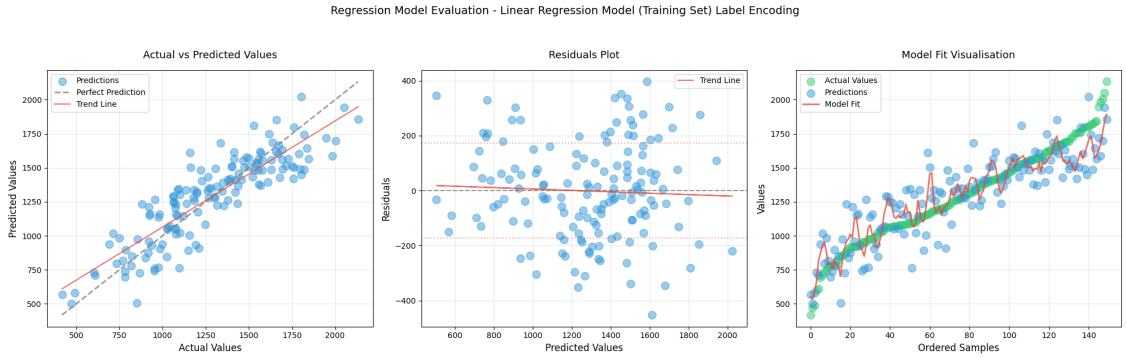
```
--- Linear Regression Model (Training Set) Label Encoding Evaluation Metrics ---
--------------------------------------------------------------------------------
    RMSE: 170.54
    R²  : 0.75
    MAE : 139.05
--------------------------------------------------------------------------------
```



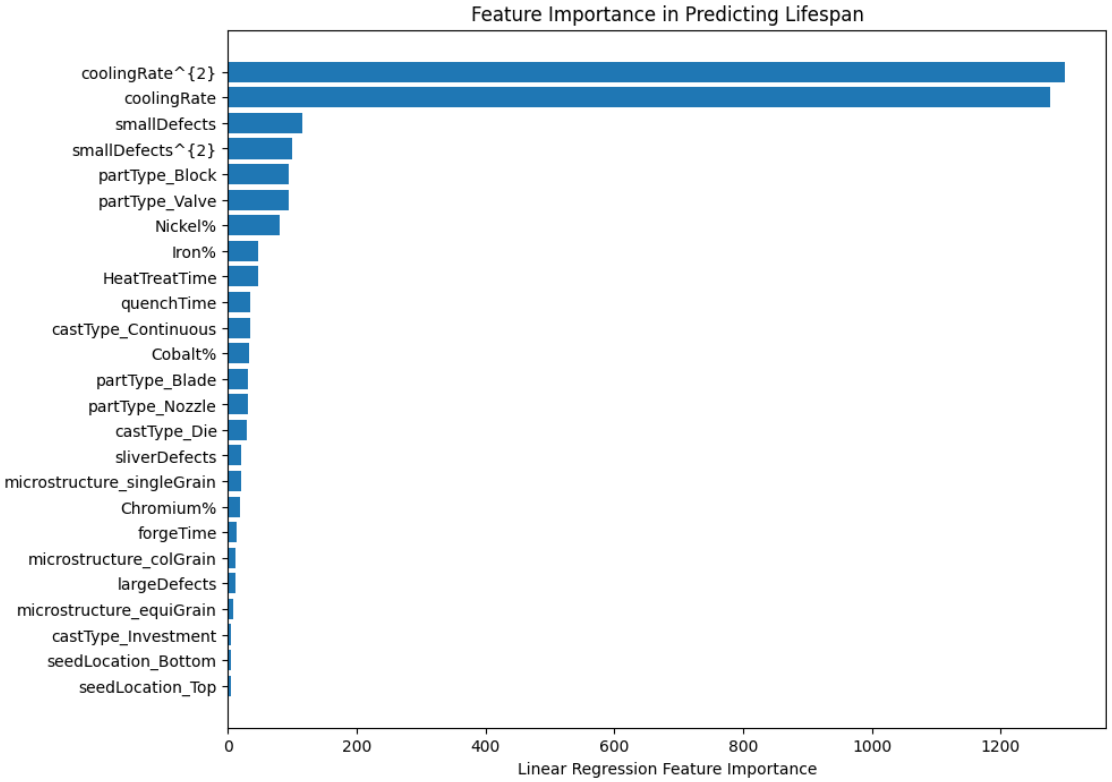
[424]:
```
# Evaluate the Linear Regression model using custom function
regression_evaluation(
    LinRegPipeline,
    X_test_reg,
    y_test_reg,
```

```
    "Linear Regression Model (Training Set) Label Encoding"
)
```

```
--- Linear Regression Model (Training Set) Label Encoding Evaluation Metrics ---
--------------------------------------------------------------------------------
    RMSE: 173.29
    R²  : 0.76
    MAE : 139.51
--------------------------------------------------------------------------------
```



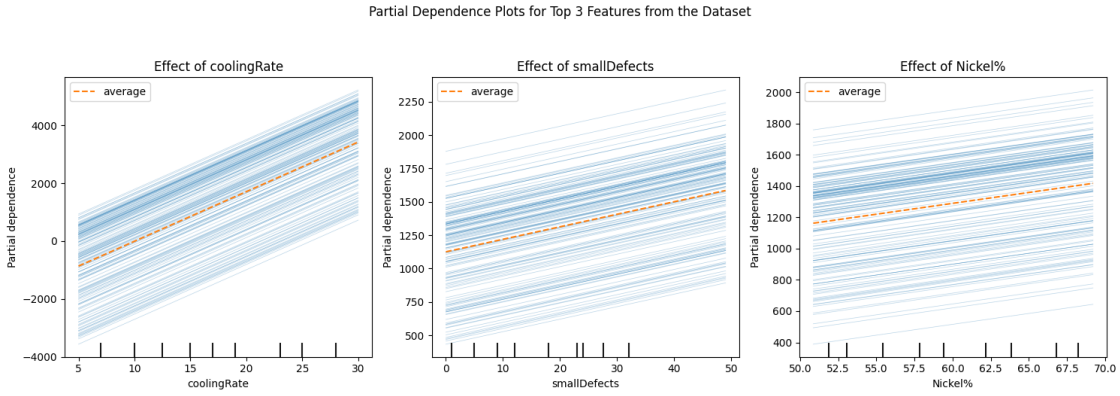
[425]:
```
# Feature importance analysis
pipeline_model = LinRegPipeline
model = pipeline_model.named_steps['model']
feature_importances = np.abs(model.coef_)
feature_names = pipeline_model.named_steps['preprocessor'].
 ↪get_feature_names_out()

# Sort features by importance
sorted_idx = np.argsort(feature_importances)
plt.figure(figsize=(10, 8))
plt.barh(feature_names[sorted_idx], feature_importances[sorted_idx])
plt.xlabel("Linear Regression Feature Importance")
plt.title(f"Feature Importance in Predicting Lifespan")
plt.show()
```



[301]:
```python
def plot_partial_dependence(
        model, X, feature_names, feature_importances,
        n_features=3, grid_resolution=100, figsize=(15, 5),␣
↪processed_data=False):
    # Create feature-importance pairs
    feature_importance_pairs = list(zip(feature_names, feature_importances))

    # Filter out polynomial and 'partType' features
    filtered_features = [
        (feature, importance)
        for feature, importance in feature_importance_pairs
        if not (
            feature.startswith('partType_') or
            re.search(r'\^\{\d+\}$', feature)
        )
    ]

    # Sort by importance and get top n features
    filtered_features.sort(key=lambda x: x[1], reverse=True)
    top_features = [feature for feature, _ in filtered_features[:n_features]]

    # Create figure with subplots
```

```
    fig, axes = plt.subplots(1, n_features, figsize=figsize)
    if n_features == 1:
        axes = [axes]

    # Generate partial dependence plots for each feature
    for idx, feature in enumerate(top_features):
        display = PartialDependenceDisplay.from_estimator(
            model,
            X,
            features=[feature],
            grid_resolution=grid_resolution,
            kind='both',
            ax=axes[idx]
        )
        axes[idx].set_title(f'Effect of {feature}')
        axes[idx].grid(True, linestyle='--', alpha=0.7)

    fig.suptitle(f'Partial Dependence Plots for Top {n_features} Features from␣
 ↪the Dataset', y=1.05)
    plt.tight_layout()
    plt.show()
```

[426]:
```
# For Linear Regression
plot_partial_dependence(
    LinRegPipeline,
    X_test_reg,
    feature_names,
    feature_importances
)
```



[115]:
```
# Initialising Random Forest Regressor
RFModel = RandomForestRegressor(n_estimators=100, random_state=42, n_jobs=-1)
```

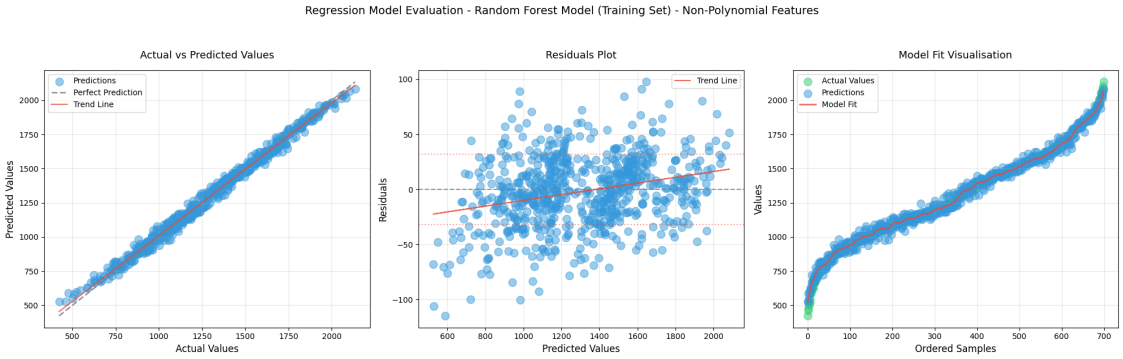
Here the parameters such as `n_estimators` and `random_state` are set to default values. The `n_estimators` parameter specifies the number of trees in the forest. The `random_state` parameter is used to set the random seed for reproducibility.

[116]:
```
RFModel.fit(X_train_reg_nonpoly, y_train_reg)
```

[116]:
```
RandomForestRegressor(n_jobs=-1, random_state=42)
```

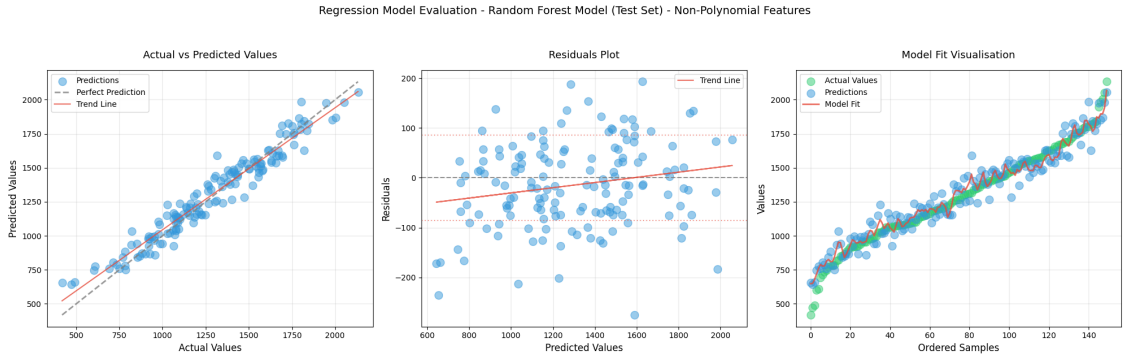
[117]:
```
regression_evaluation(RFModel, X_train_reg_nonpoly, y_train_reg, "Random Forest␣
↪Model (Training Set) - Non-Polynomial Features")
```

```
--- Random Forest Model (Training Set) - Non-Polynomial Features Evaluation
Metrics ---
--------------------------------------------------------------------------------
-------
    RMSE: 32.01
    R²   : 0.99
    MAE : 25.05
--------------------------------------------------------------------------------
-------
```



[118]:
```
regression_evaluation(RFModel, X_test_reg_nonpoly, y_test_reg, "Random Forest␣
↪Model (Test Set) - Non-Polynomial Features")
```

```
--- Random Forest Model (Test Set) - Non-Polynomial Features Evaluation Metrics
---
--------------------------------------------------------------------------------
---
    RMSE: 86.84
    R²   : 0.94
    MAE : 69.96
--------------------------------------------------------------------------------
---
```

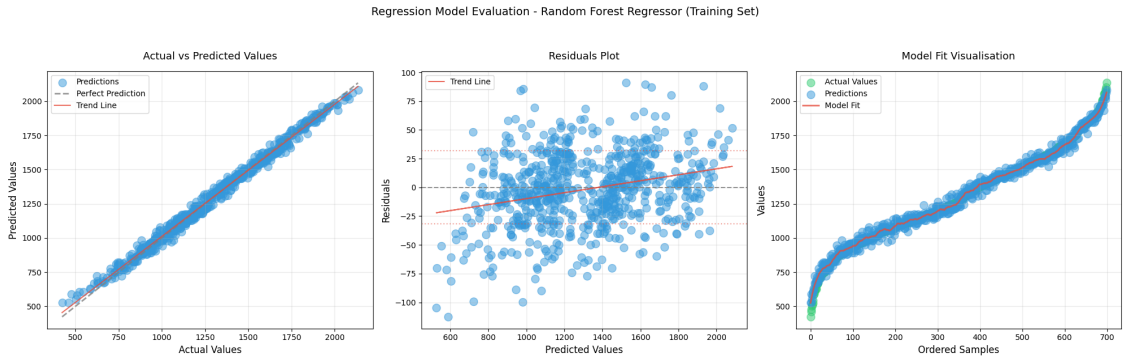


[119]:
```
# Fit the model with polynomial features
RFModel.fit(X_train_reg, y_train_reg)
```

[119]:
```
RandomForestRegressor(n_jobs=-1, random_state=42)
```

[120]:
```
# Evaluate the model on the training set with polynomial features of degree 2
regression_evaluation(RFModel, X_train_reg, y_train_reg, "Random Forest␣
↪Regressor (Training Set)")
```

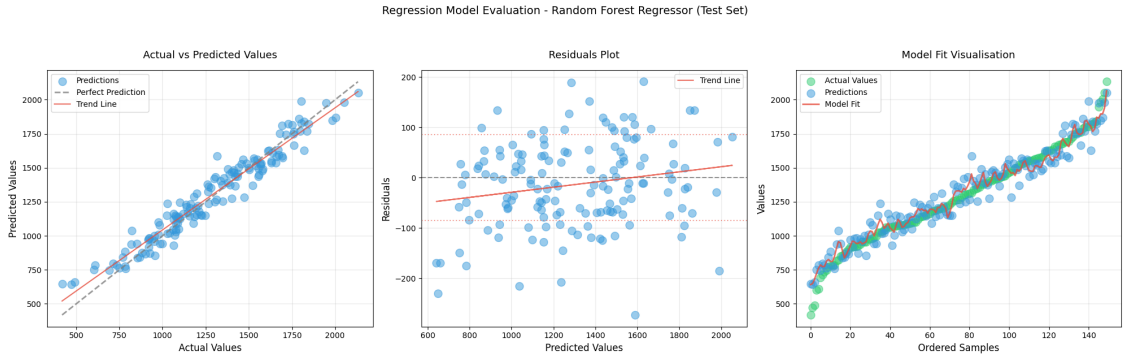
```
--- Random Forest Regressor (Training Set) Evaluation Metrics ---
-----------------------------------------------------------------
    RMSE: 31.85
    R²  : 0.99
    MAE : 24.74
-----------------------------------------------------------------
```



[151]:
```
# Evaluate the model on the test set with polynomial features
regression_evaluation(RFModel, X_test_reg, y_test_reg, "Random Forest Regressor␣
↪(Test Set)")
```

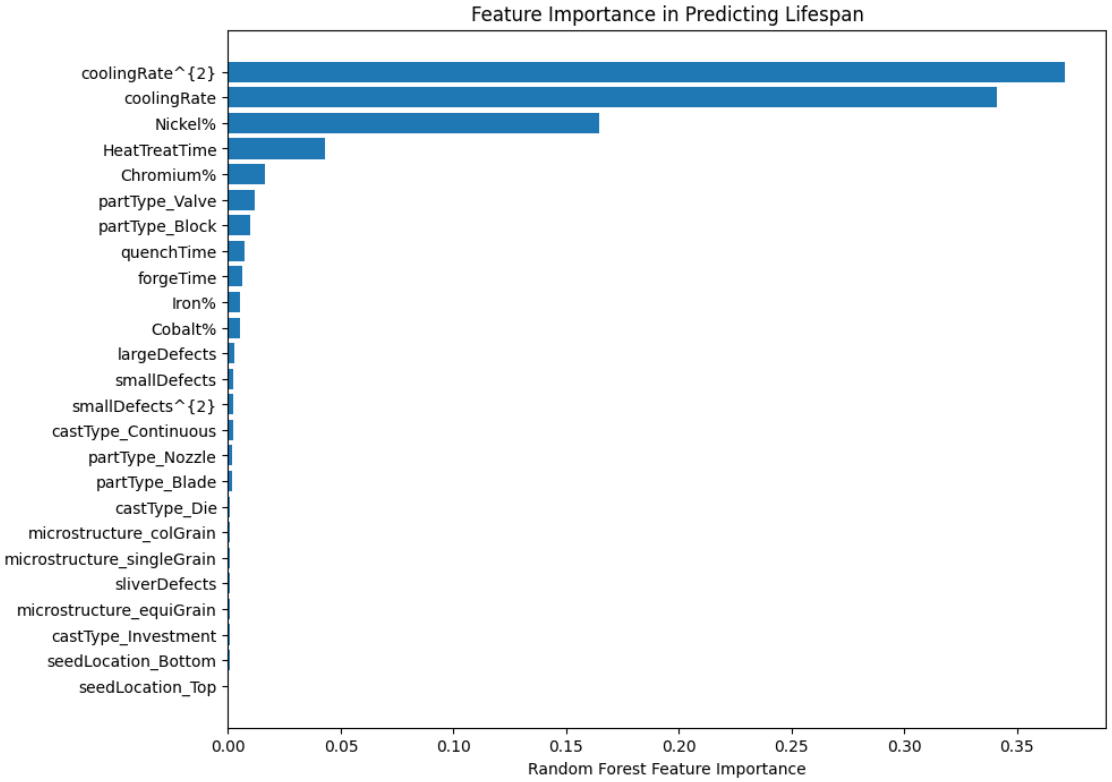
```
--- Random Forest Regressor (Test Set) Evaluation Metrics ---
------------------------------------------------------------
```

```
RMSE: 86.87
R²  : 0.94
MAE : 69.72
------------------------------------------------------------
```

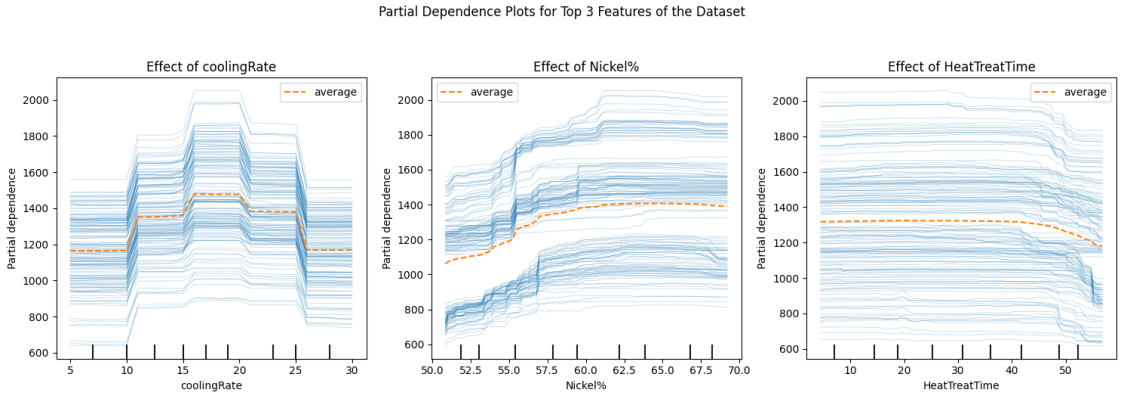


Here we can see that the accuracy drops because the model was overfitted on training set. Even though the model was overfitted on training set, it is still able to predict the test set with a good accuracy.

[152]:
```
# Feature importance analysis
feature_importances = RFModel.feature_importances_
feature_names = X_train_reg.columns

# Sort features by importance
sorted_idx = np.argsort(feature_importances)
plt.figure(figsize=(10, 8))
plt.barh(feature_names[sorted_idx], feature_importances[sorted_idx])
plt.xlabel("Random Forest Feature Importance")
plt.title(f"Feature Importance in Predicting Lifespan")
plt.show()
```



[156]:
```
# For Random Forest Regression
plot_partial_dependence(
    RFModel,
    X_test_reg,
    feature_names,
    feature_importances
)
```

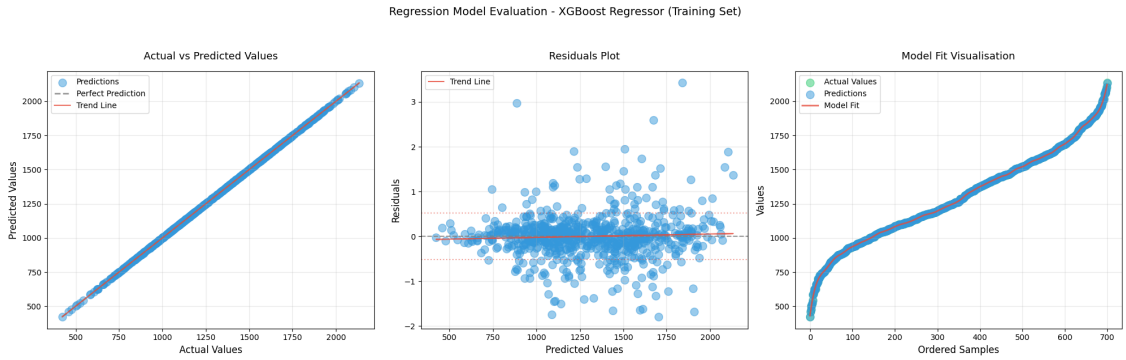


[124]:
```
XGBModel = XGBRegressor(n_estimators=100, n_jobs=-1, random_state=42)

# Fit the model
XGBModel.fit(X_train_reg, y_train_reg)
```

[124]:
```
XGBRegressor(base_score=None, booster=None, callbacks=None,
             colsample_bylevel=None, colsample_bynode=None,
             colsample_bytree=None, device=None, early_stopping_rounds=None,
             enable_categorical=False, eval_metric=None, feature_types=None,
             gamma=None, grow_policy=None, importance_type=None,
             interaction_constraints=None, learning_rate=None, max_bin=None,
             max_cat_threshold=None, max_cat_to_onehot=None,
             max_delta_step=None, max_depth=None, max_leaves=None,
             min_child_weight=None, missing=nan, monotone_constraints=None,
             multi_strategy=None, n_estimators=100, n_jobs=-1,
             num_parallel_tree=None, random_state=42, …)
```

[125]:
```
regression_evaluation(XGBModel, X_train_reg, y_train_reg, "XGBoost Regressor␣
↪(Training Set)")
```

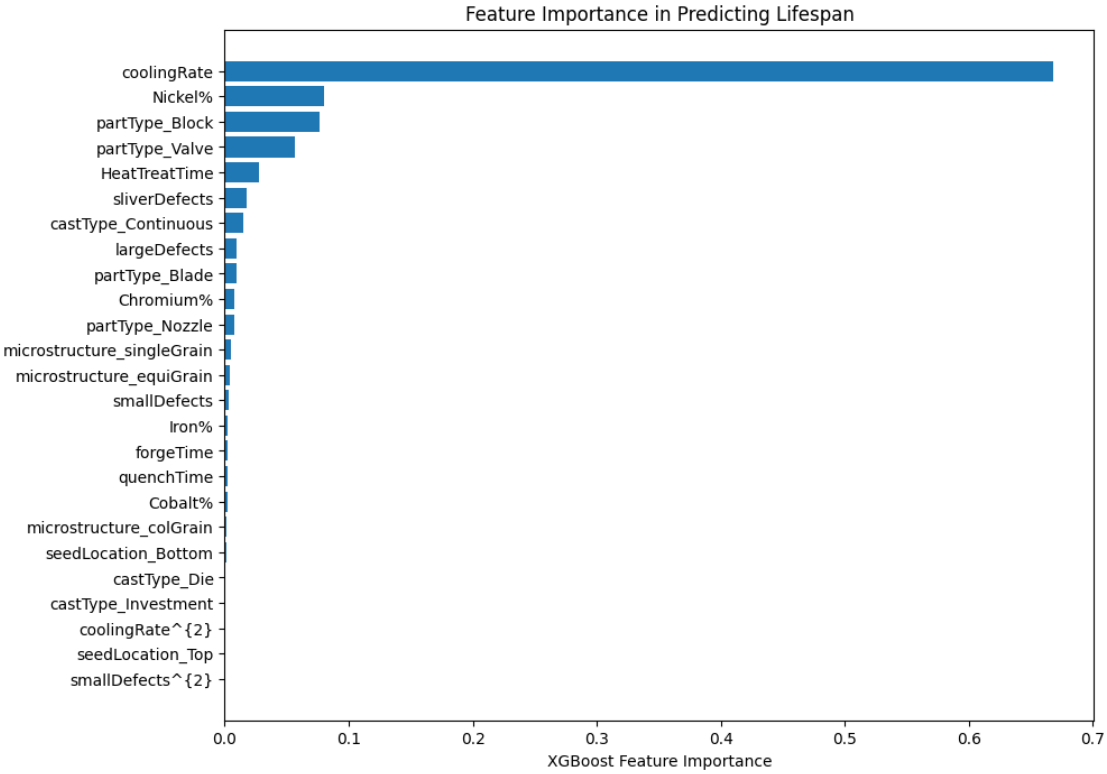
```
--- XGBoost Regressor (Training Set) Evaluation Metrics ---
------------------------------------------------------------
    RMSE: 0.52
    R²   : 1.00
    MAE : 0.33
------------------------------------------------------------
```



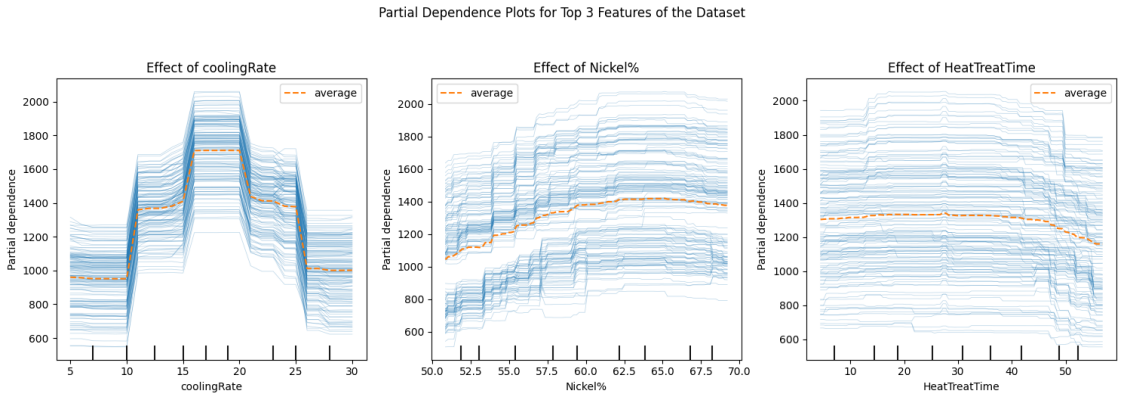
[159]:
```
regression_evaluation(XGBModel, X_test_reg, y_test_reg, "XGBoost Regressor␣
↪(Test Set)")
```

```
--- XGBoost Regressor (Test Set) Evaluation Metrics ---
------------------------------------------------------
    RMSE: 65.68
    R²   : 0.97
    MAE : 48.92
```

[160]:
```
# Feature importance analysis
feature_importances = XGBModel.feature_importances_
feature_names = X_train_reg.columns

# Sort features by importance
sorted_idx = np.argsort(feature_importances)
plt.figure(figsize=(10, 8))
plt.barh(feature_names[sorted_idx], feature_importances[sorted_idx])
plt.xlabel("XGBoost Feature Importance")
plt.title(f"Feature Importance in Predicting Lifespan")
plt.show()
```



[161]:
```
plot_partial_dependence(
    XGBModel,
    X_test_reg,
    feature_names,
    feature_importances
)
```



[163]:
```
# Switch between the two datasets with and without polynomial features
NN_X_train = X_train_reg
NN_X_test = X_test_reg
NN_X_val = X_val_reg
```

[164]:
```
numerical_cols = [col for col in NN_X_train.columns if NN_X_train[col].dtype in␣
↪['int64', 'float64'] and col != 'Lifespan']
categorical_cols = NN_X_train.select_dtypes(include=['object']).columns.tolist()
ohe = OneHotEncoder(handle_unknown='ignore', sparse_output=False, dtype=int,␣
↪drop=None)
scaler = StandardScaler()

preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numerical_cols)
    ]
)
```

[165]:
```
early_stopping = EarlyStopping(
    monitor="val_loss",
    patience=20,
    restore_best_weights=True
)
reduce_lr = ReduceLROnPlateau(
    monitor="val_loss",
    factor=0.2,
    patience=10,
    min_lr=1e-5
)
```

[166]:
```
# Preprocess training data
X_train_processed = preprocessor.fit_transform(NN_X_train)
X_val_processed = preprocessor.transform(NN_X_val)
X_test_processed = preprocessor.transform(NN_X_test)
```

[167]:
```
input_dim = NN_X_train.shape[1]

NNModel = tf.keras.Sequential([
    Input(shape=(input_dim,)),
    Dense(64, activation="relu"),
    BatchNormalization(),
    Dropout(0.3),
    Dense(32, activation="relu"),
    BatchNormalization(),
    Dropout(0.2),
    Dense(16, activation="relu"),
    BatchNormalization(),
```

```
    Dropout(0.1),
    Dense(1)
])

optimizer = Adam(learning_rate=0.0010)
NNModel.compile(optimizer="adam", loss="mse")

NNModel.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
dense_8 (Dense)	(None, 64)	1,664
batch_normalization_6 (BatchNormalization)	(None, 64)	256
dropout_6 (Dropout)	(None, 64)	0
dense_9 (Dense)	(None, 32)	2,080
batch_normalization_7 (BatchNormalization)	(None, 32)	128
dropout_7 (Dropout)	(None, 32)	0
dense_10 (Dense)	(None, 16)	528
batch_normalization_8 (BatchNormalization)	(None, 16)	64
dropout_8 (Dropout)	(None, 16)	0
dense_11 (Dense)	(None, 1)	17

Total params: 4,737 (18.50 KB)

Trainable params: 4,513 (17.63 KB)

Non-trainable params: 224 (896.00 B)

[134]:
```
# Train model with processed data
NNModel.fit(
    x=X_train_processed,
    y=y_train_reg,
    validation_data=(X_val_processed, y_val_reg.values),
    epochs=450,
    batch_size=38,
    callbacks=[
        EarlyStopping(monitor="val_loss", patience=20,␣
 ↪restore_best_weights=True),
        ReduceLROnPlateau(monitor="val_loss", factor=0.2, patience=10,␣
 ↪min_lr=1e-5)
    ],
    verbose=1
)
```

```
Epoch 1/450
19/19             1s 8ms/step - loss:
1761787.5000 - val_loss: 1785622.5000 - learning_rate: 0.0010
Epoch 2/450
19/19             0s 2ms/step - loss:
1761518.1250 - val_loss: 1785750.0000 - learning_rate: 0.0010
Epoch 3/450
19/19             0s 2ms/step - loss:
1761093.8750 - val_loss: 1785713.5000 - learning_rate: 0.0010
Epoch 4/450
19/19             0s 2ms/step - loss:
1760690.6250 - val_loss: 1785488.8750 - learning_rate: 0.0010
Epoch 5/450
19/19             0s 2ms/step - loss:
1760099.3750 - val_loss: 1785158.1250 - learning_rate: 0.0010
Epoch 6/450
19/19             0s 2ms/step - loss:
1759456.0000 - val_loss: 1784671.2500 - learning_rate: 0.0010
Epoch 7/450
19/19             0s 2ms/step - loss:
1758949.6250 - val_loss: 1784078.8750 - learning_rate: 0.0010
Epoch 8/450
19/19             0s 2ms/step - loss:
1758148.5000 - val_loss: 1783285.5000 - learning_rate: 0.0010
Epoch 9/450
19/19             0s 2ms/step - loss:
1757344.8750 - val_loss: 1782327.2500 - learning_rate: 0.0010
Epoch 10/450
19/19             0s 2ms/step - loss:
1756354.5000 - val_loss: 1781268.7500 - learning_rate: 0.0010
Epoch 11/450
19/19             0s 2ms/step - loss:
```

```
1755239.5000 - val_loss: 1780139.1250 - learning_rate: 0.0010
Epoch 12/450
19/19            0s 2ms/step - loss:
1754068.5000 - val_loss: 1778995.8750 - learning_rate: 0.0010
Epoch 13/450
19/19            0s 2ms/step - loss:
1752750.8750 - val_loss: 1777566.1250 - learning_rate: 0.0010
Epoch 14/450
19/19            0s 2ms/step - loss:
1751140.1250 - val_loss: 1775955.5000 - learning_rate: 0.0010
Epoch 15/450
19/19            0s 2ms/step - loss:
1749436.5000 - val_loss: 1774492.7500 - learning_rate: 0.0010
Epoch 16/450
19/19            0s 2ms/step - loss:
1747830.6250 - val_loss: 1773029.5000 - learning_rate: 0.0010
Epoch 17/450
19/19            0s 2ms/step - loss:
1746227.1250 - val_loss: 1771425.1250 - learning_rate: 0.0010
Epoch 18/450
19/19            0s 2ms/step - loss:
1743985.7500 - val_loss: 1769015.0000 - learning_rate: 0.0010
Epoch 19/450
19/19            0s 2ms/step - loss:
1741729.8750 - val_loss: 1766050.5000 - learning_rate: 0.0010
Epoch 20/450
19/19            0s 2ms/step - loss:
1739686.1250 - val_loss: 1763522.3750 - learning_rate: 0.0010
Epoch 21/450
19/19            0s 1ms/step - loss:
1737966.0000 - val_loss: 1761128.8750 - learning_rate: 0.0010
Epoch 22/450
19/19            0s 2ms/step - loss:
1734987.6250 - val_loss: 1759257.2500 - learning_rate: 0.0010
Epoch 23/450
19/19            0s 2ms/step - loss:
1732813.5000 - val_loss: 1757262.7500 - learning_rate: 0.0010
Epoch 24/450
19/19            0s 2ms/step - loss:
1730019.7500 - val_loss: 1754371.2500 - learning_rate: 0.0010
Epoch 25/450
19/19            0s 1ms/step - loss:
1727867.3750 - val_loss: 1751255.2500 - learning_rate: 0.0010
Epoch 26/450
19/19            0s 1ms/step - loss:
1725376.7500 - val_loss: 1748556.5000 - learning_rate: 0.0010
Epoch 27/450
19/19            0s 2ms/step - loss:
```

```
1722530.5000 - val_loss: 1747223.2500 - learning_rate: 0.0010
Epoch 28/450
19/19            0s 2ms/step - loss:
1719598.3750 - val_loss: 1744201.7500 - learning_rate: 0.0010
Epoch 29/450
19/19            0s 2ms/step - loss:
1716141.2500 - val_loss: 1740910.8750 - learning_rate: 0.0010
Epoch 30/450
19/19            0s 2ms/step - loss:
1713408.0000 - val_loss: 1738170.6250 - learning_rate: 0.0010
Epoch 31/450
19/19            0s 2ms/step - loss:
1711204.5000 - val_loss: 1735799.0000 - learning_rate: 0.0010
Epoch 32/450
19/19            0s 2ms/step - loss:
1708340.3750 - val_loss: 1732870.5000 - learning_rate: 0.0010
Epoch 33/450
19/19            0s 2ms/step - loss:
1705149.8750 - val_loss: 1729792.2500 - learning_rate: 0.0010
Epoch 34/450
19/19            0s 2ms/step - loss:
1701423.5000 - val_loss: 1726950.1250 - learning_rate: 0.0010
Epoch 35/450
19/19            0s 3ms/step - loss:
1697693.3750 - val_loss: 1722785.8750 - learning_rate: 0.0010
Epoch 36/450
19/19            0s 2ms/step - loss:
1695012.7500 - val_loss: 1719589.5000 - learning_rate: 0.0010
Epoch 37/450
19/19            0s 2ms/step - loss:
1692004.0000 - val_loss: 1716781.3750 - learning_rate: 0.0010
Epoch 38/450
19/19            0s 2ms/step - loss:
1688190.8750 - val_loss: 1714067.1250 - learning_rate: 0.0010
Epoch 39/450
19/19            0s 2ms/step - loss:
1683458.7500 - val_loss: 1710523.7500 - learning_rate: 0.0010
Epoch 40/450
19/19            0s 2ms/step - loss:
1680639.5000 - val_loss: 1707978.0000 - learning_rate: 0.0010
Epoch 41/450
19/19            0s 1ms/step - loss:
1677253.5000 - val_loss: 1703575.6250 - learning_rate: 0.0010
Epoch 42/450
19/19            0s 1ms/step - loss:
1673508.2500 - val_loss: 1700712.8750 - learning_rate: 0.0010
Epoch 43/450
19/19            0s 1ms/step - loss:
```

```
1670570.0000 - val_loss: 1696836.7500 - learning_rate: 0.0010
Epoch 44/450
19/19            0s 2ms/step - loss:
1664751.6250 - val_loss: 1695913.8750 - learning_rate: 0.0010
Epoch 45/450
19/19            0s 2ms/step - loss:
1661569.1250 - val_loss: 1693334.1250 - learning_rate: 0.0010
Epoch 46/450
19/19            0s 2ms/step - loss:
1658009.3750 - val_loss: 1687951.5000 - learning_rate: 0.0010
Epoch 47/450
19/19            0s 2ms/step - loss:
1654533.8750 - val_loss: 1682194.8750 - learning_rate: 0.0010
Epoch 48/450
19/19            0s 2ms/step - loss:
1649015.8750 - val_loss: 1679195.6250 - learning_rate: 0.0010
Epoch 49/450
19/19            0s 2ms/step - loss:
1646341.0000 - val_loss: 1676766.6250 - learning_rate: 0.0010
Epoch 50/450
19/19            0s 2ms/step - loss:
1642083.3750 - val_loss: 1673303.2500 - learning_rate: 0.0010
Epoch 51/450
19/19            0s 2ms/step - loss:
1638137.1250 - val_loss: 1668364.1250 - learning_rate: 0.0010
Epoch 52/450
19/19            0s 2ms/step - loss:
1634097.7500 - val_loss: 1662967.6250 - learning_rate: 0.0010
Epoch 53/450
19/19            0s 2ms/step - loss:
1629910.1250 - val_loss: 1657455.6250 - learning_rate: 0.0010
Epoch 54/450
19/19            0s 2ms/step - loss:
1624275.5000 - val_loss: 1654750.0000 - learning_rate: 0.0010
Epoch 55/450
19/19            0s 2ms/step - loss:
1620037.5000 - val_loss: 1650298.3750 - learning_rate: 0.0010
Epoch 56/450
19/19            0s 3ms/step - loss:
1615975.1250 - val_loss: 1644529.2500 - learning_rate: 0.0010
Epoch 57/450
19/19            0s 1ms/step - loss:
1608427.7500 - val_loss: 1640172.7500 - learning_rate: 0.0010
Epoch 58/450
19/19            0s 1ms/step - loss:
1604991.7500 - val_loss: 1633757.0000 - learning_rate: 0.0010
Epoch 59/450
19/19            0s 2ms/step - loss:
```

```
1602645.0000 - val_loss: 1630337.6250 - learning_rate: 0.0010
Epoch 60/450
19/19            0s 1ms/step - loss:
1595271.2500 - val_loss: 1630499.7500 - learning_rate: 0.0010
Epoch 61/450
19/19            0s 1ms/step - loss:
1590923.5000 - val_loss: 1630394.2500 - learning_rate: 0.0010
Epoch 62/450
19/19            0s 1ms/step - loss:
1590169.6250 - val_loss: 1625963.5000 - learning_rate: 0.0010
Epoch 63/450
19/19            0s 1ms/step - loss:
1582018.5000 - val_loss: 1619140.8750 - learning_rate: 0.0010
Epoch 64/450
19/19            0s 1ms/step - loss:
1577286.2500 - val_loss: 1616656.6250 - learning_rate: 0.0010
Epoch 65/450
19/19            0s 1ms/step - loss:
1573055.1250 - val_loss: 1611522.1250 - learning_rate: 0.0010
Epoch 66/450
19/19            0s 1ms/step - loss:
1567390.2500 - val_loss: 1605034.5000 - learning_rate: 0.0010
Epoch 67/450
19/19            0s 1ms/step - loss:
1562805.7500 - val_loss: 1598105.8750 - learning_rate: 0.0010
Epoch 68/450
19/19            0s 1ms/step - loss:
1557541.1250 - val_loss: 1590019.0000 - learning_rate: 0.0010
Epoch 69/450
19/19            0s 1ms/step - loss:
1553262.3750 - val_loss: 1583596.5000 - learning_rate: 0.0010
Epoch 70/450
19/19            0s 1ms/step - loss:
1549658.2500 - val_loss: 1577863.3750 - learning_rate: 0.0010
Epoch 71/450
19/19            0s 1ms/step - loss:
1541279.7500 - val_loss: 1576735.3750 - learning_rate: 0.0010
Epoch 72/450
19/19            0s 3ms/step - loss:
1538323.3750 - val_loss: 1569701.1250 - learning_rate: 0.0010
Epoch 73/450
19/19            0s 1ms/step - loss:
1532029.0000 - val_loss: 1563871.2500 - learning_rate: 0.0010
Epoch 74/450
19/19            0s 1ms/step - loss:
1526241.2500 - val_loss: 1563901.7500 - learning_rate: 0.0010
Epoch 75/450
19/19            0s 1ms/step - loss:
```

```
1522305.2500 - val_loss: 1567302.6250 - learning_rate: 0.0010
Epoch 76/450
19/19            0s 1ms/step - loss:
1518132.2500 - val_loss: 1563076.0000 - learning_rate: 0.0010
Epoch 77/450
19/19            0s 1ms/step - loss:
1509969.7500 - val_loss: 1551575.2500 - learning_rate: 0.0010
Epoch 78/450
19/19            0s 1ms/step - loss:
1504094.8750 - val_loss: 1543720.6250 - learning_rate: 0.0010
Epoch 79/450
19/19            0s 1ms/step - loss:
1502366.5000 - val_loss: 1534534.3750 - learning_rate: 0.0010
Epoch 80/450
19/19            0s 1ms/step - loss:
1493760.5000 - val_loss: 1528392.7500 - learning_rate: 0.0010
Epoch 81/450
19/19            0s 1ms/step - loss:
1490551.0000 - val_loss: 1523269.7500 - learning_rate: 0.0010
Epoch 82/450
19/19            0s 2ms/step - loss:
1486469.1250 - val_loss: 1516294.2500 - learning_rate: 0.0010
Epoch 83/450
19/19            0s 2ms/step - loss:
1477142.5000 - val_loss: 1516380.1250 - learning_rate: 0.0010
Epoch 84/450
19/19            0s 1ms/step - loss:
1473717.0000 - val_loss: 1507173.1250 - learning_rate: 0.0010
Epoch 85/450
19/19            0s 1ms/step - loss:
1467389.7500 - val_loss: 1504623.1250 - learning_rate: 0.0010
Epoch 86/450
19/19            0s 3ms/step - loss:
1460279.6250 - val_loss: 1501000.3750 - learning_rate: 0.0010
Epoch 87/450
19/19            0s 1ms/step - loss:
1457362.7500 - val_loss: 1501534.3750 - learning_rate: 0.0010
Epoch 88/450
19/19            0s 1ms/step - loss:
1452083.0000 - val_loss: 1495259.7500 - learning_rate: 0.0010
Epoch 89/450
19/19            0s 1ms/step - loss:
1444259.6250 - val_loss: 1485031.8750 - learning_rate: 0.0010
Epoch 90/450
19/19            0s 1ms/step - loss:
1439051.5000 - val_loss: 1480770.3750 - learning_rate: 0.0010
Epoch 91/450
19/19            0s 2ms/step - loss:
```

```
1433245.5000 - val_loss: 1474318.8750 - learning_rate: 0.0010
Epoch 92/450
19/19            0s 1ms/step - loss:
1425988.6250 - val_loss: 1468876.6250 - learning_rate: 0.0010
Epoch 93/450
19/19            0s 1ms/step - loss:
1419724.2500 - val_loss: 1457876.6250 - learning_rate: 0.0010
Epoch 94/450
19/19            0s 1ms/step - loss:
1416221.0000 - val_loss: 1448333.3750 - learning_rate: 0.0010
Epoch 95/450
19/19            0s 1ms/step - loss:
1407463.1250 - val_loss: 1443315.2500 - learning_rate: 0.0010
Epoch 96/450
19/19            0s 1ms/step - loss:
1400844.8750 - val_loss: 1444378.5000 - learning_rate: 0.0010
Epoch 97/450
19/19            0s 1ms/step - loss:
1396335.5000 - val_loss: 1442175.1250 - learning_rate: 0.0010
Epoch 98/450
19/19            0s 1ms/step - loss:
1393218.7500 - val_loss: 1436290.1250 - learning_rate: 0.0010
Epoch 99/450
19/19            0s 1ms/step - loss:
1384140.1250 - val_loss: 1430938.2500 - learning_rate: 0.0010
Epoch 100/450
19/19            0s 3ms/step - loss:
1380645.7500 - val_loss: 1423049.7500 - learning_rate: 0.0010
Epoch 101/450
19/19            0s 1ms/step - loss:
1372441.8750 - val_loss: 1421738.0000 - learning_rate: 0.0010
Epoch 102/450
19/19            0s 1ms/step - loss:
1371820.2500 - val_loss: 1421315.1250 - learning_rate: 0.0010
Epoch 103/450
19/19            0s 1ms/step - loss:
1364459.8750 - val_loss: 1416111.7500 - learning_rate: 0.0010
Epoch 104/450
19/19            0s 1ms/step - loss:
1355739.8750 - val_loss: 1408458.5000 - learning_rate: 0.0010
Epoch 105/450
19/19            0s 1ms/step - loss:
1353114.6250 - val_loss: 1399598.8750 - learning_rate: 0.0010
Epoch 106/450
19/19            0s 1ms/step - loss:
1347147.5000 - val_loss: 1388628.8750 - learning_rate: 0.0010
Epoch 107/450
19/19            0s 1ms/step - loss:
```

```
1337883.6250 - val_loss: 1379249.2500 - learning_rate: 0.0010
Epoch 108/450
19/19            0s 1ms/step - loss:
1334337.8750 - val_loss: 1369974.6250 - learning_rate: 0.0010
Epoch 109/450
19/19            0s 1ms/step - loss:
1331137.5000 - val_loss: 1356367.1250 - learning_rate: 0.0010
Epoch 110/450
19/19            0s 1ms/step - loss:
1321298.1250 - val_loss: 1353882.3750 - learning_rate: 0.0010
Epoch 111/450
19/19            0s 1ms/step - loss:
1317506.2500 - val_loss: 1353878.0000 - learning_rate: 0.0010
Epoch 112/450
19/19            0s 1ms/step - loss:
1305261.8750 - val_loss: 1346753.8750 - learning_rate: 0.0010
Epoch 113/450
19/19            0s 1ms/step - loss:
1304204.0000 - val_loss: 1343691.0000 - learning_rate: 0.0010
Epoch 114/450
19/19            0s 2ms/step - loss:
1299080.2500 - val_loss: 1335551.1250 - learning_rate: 0.0010
Epoch 115/450
19/19            0s 1ms/step - loss:
1297982.8750 - val_loss: 1328653.3750 - learning_rate: 0.0010
Epoch 116/450
19/19            0s 1ms/step - loss:
1281049.6250 - val_loss: 1328483.3750 - learning_rate: 0.0010
Epoch 117/450
19/19            0s 1ms/step - loss:
1279730.2500 - val_loss: 1328641.5000 - learning_rate: 0.0010
Epoch 118/450
19/19            0s 1ms/step - loss:
1269048.7500 - val_loss: 1320186.0000 - learning_rate: 0.0010
Epoch 119/450
19/19            0s 1ms/step - loss:
1270206.5000 - val_loss: 1316307.3750 - learning_rate: 0.0010
Epoch 120/450
19/19            0s 2ms/step - loss:
1258720.6250 - val_loss: 1315635.8750 - learning_rate: 0.0010
Epoch 121/450
19/19            0s 2ms/step - loss:
1258608.3750 - val_loss: 1313751.6250 - learning_rate: 0.0010
Epoch 122/450
19/19            0s 1ms/step - loss:
1247219.5000 - val_loss: 1300637.0000 - learning_rate: 0.0010
Epoch 123/450
19/19            0s 2ms/step - loss:
```

```
1245404.3750 - val_loss: 1293827.7500 - learning_rate: 0.0010
Epoch 124/450
19/19            0s 3ms/step - loss:
1235174.2500 - val_loss: 1282937.0000 - learning_rate: 0.0010
Epoch 125/450
19/19            0s 1ms/step - loss:
1230647.8750 - val_loss: 1282610.1250 - learning_rate: 0.0010
Epoch 126/450
19/19            0s 1ms/step - loss:
1225035.8750 - val_loss: 1276306.5000 - learning_rate: 0.0010
Epoch 127/450
19/19            0s 2ms/step - loss:
1225582.8750 - val_loss: 1270829.2500 - learning_rate: 0.0010
Epoch 128/450
19/19            0s 1ms/step - loss:
1208939.3750 - val_loss: 1267667.0000 - learning_rate: 0.0010
Epoch 129/450
19/19            0s 1ms/step - loss:
1206118.2500 - val_loss: 1253657.7500 - learning_rate: 0.0010
Epoch 130/450
19/19            0s 1ms/step - loss:
1202326.2500 - val_loss: 1248533.2500 - learning_rate: 0.0010
Epoch 131/450
19/19            0s 1ms/step - loss:
1196842.0000 - val_loss: 1242010.8750 - learning_rate: 0.0010
Epoch 132/450
19/19            0s 1ms/step - loss:
1189070.3750 - val_loss: 1238428.3750 - learning_rate: 0.0010
Epoch 133/450
19/19            0s 1ms/step - loss:
1178583.0000 - val_loss: 1228835.2500 - learning_rate: 0.0010
Epoch 134/450
19/19            0s 1ms/step - loss:
1172185.1250 - val_loss: 1214948.0000 - learning_rate: 0.0010
Epoch 135/450
19/19            0s 1ms/step - loss:
1169014.8750 - val_loss: 1216984.3750 - learning_rate: 0.0010
Epoch 136/450
19/19            0s 3ms/step - loss:
1166786.5000 - val_loss: 1210395.7500 - learning_rate: 0.0010
Epoch 137/450
19/19            0s 1ms/step - loss:
1156177.3750 - val_loss: 1206530.5000 - learning_rate: 0.0010
Epoch 138/450
19/19            0s 1ms/step - loss:
1154481.0000 - val_loss: 1199347.6250 - learning_rate: 0.0010
Epoch 139/450
19/19            0s 1ms/step - loss:
```

```
1144464.3750 - val_loss: 1191673.6250 - learning_rate: 0.0010
Epoch 140/450
19/19            0s 1ms/step - loss:
1140748.5000 - val_loss: 1191146.3750 - learning_rate: 0.0010
Epoch 141/450
19/19            0s 1ms/step - loss:
1130640.6250 - val_loss: 1188979.1250 - learning_rate: 0.0010
Epoch 142/450
19/19            0s 1ms/step - loss:
1118815.7500 - val_loss: 1171284.2500 - learning_rate: 0.0010
Epoch 143/450
19/19            0s 1ms/step - loss:
1117588.3750 - val_loss: 1153465.3750 - learning_rate: 0.0010
Epoch 144/450
19/19            0s 1ms/step - loss:
1121275.5000 - val_loss: 1150191.3750 - learning_rate: 0.0010
Epoch 145/450
19/19            0s 1ms/step - loss:
1111758.2500 - val_loss: 1143666.0000 - learning_rate: 0.0010
Epoch 146/450
19/19            0s 1ms/step - loss:
1099569.6250 - val_loss: 1139071.2500 - learning_rate: 0.0010
Epoch 147/450
19/19            0s 1ms/step - loss:
1098521.6250 - val_loss: 1138500.5000 - learning_rate: 0.0010
Epoch 148/450
19/19            0s 1ms/step - loss:
1088895.7500 - val_loss: 1131448.8750 - learning_rate: 0.0010
Epoch 149/450
19/19            0s 1ms/step - loss:
1086563.6250 - val_loss: 1125557.5000 - learning_rate: 0.0010
Epoch 150/450
19/19            0s 3ms/step - loss:
1075436.2500 - val_loss: 1123774.8750 - learning_rate: 0.0010
Epoch 151/450
19/19            0s 1ms/step - loss:
1081564.1250 - val_loss: 1118916.1250 - learning_rate: 0.0010
Epoch 152/450
19/19            0s 1ms/step - loss:
1063839.8750 - val_loss: 1104640.6250 - learning_rate: 0.0010
Epoch 153/450
19/19            0s 1ms/step - loss:
1055524.8750 - val_loss: 1103652.2500 - learning_rate: 0.0010
Epoch 154/450
19/19            0s 1ms/step - loss:
1050856.1250 - val_loss: 1104745.0000 - learning_rate: 0.0010
Epoch 155/450
19/19            0s 1ms/step - loss:
```

```
1053720.7500 - val_loss: 1094584.6250 - learning_rate: 0.0010
Epoch 156/450
19/19            0s 1ms/step - loss:
1048569.3125 - val_loss: 1091819.5000 - learning_rate: 0.0010
Epoch 157/450
19/19            0s 1ms/step - loss:
1036872.7500 - val_loss: 1073211.0000 - learning_rate: 0.0010
Epoch 158/450
19/19            0s 1ms/step - loss:
1038082.7500 - val_loss: 1059152.6250 - learning_rate: 0.0010
Epoch 159/450
19/19            0s 1ms/step - loss:
1021461.4375 - val_loss: 1056726.0000 - learning_rate: 0.0010
Epoch 160/450
19/19            0s 1ms/step - loss:
1016362.3750 - val_loss: 1057598.1250 - learning_rate: 0.0010
Epoch 161/450
19/19            0s 1ms/step - loss:
1010940.1250 - val_loss: 1052092.5000 - learning_rate: 0.0010
Epoch 162/450
19/19            0s 1ms/step - loss:
1002141.7500 - val_loss: 1045619.1875 - learning_rate: 0.0010
Epoch 163/450
19/19            0s 2ms/step - loss:
1000699.8750 - val_loss: 1037380.5625 - learning_rate: 0.0010
Epoch 164/450
19/19            0s 1ms/step - loss:
998422.3750 - val_loss: 1032030.6250 - learning_rate: 0.0010
Epoch 165/450
19/19            0s 1ms/step - loss:
988078.8750 - val_loss: 1027220.2500 - learning_rate: 0.0010
Epoch 166/450
19/19            0s 1ms/step - loss:
981973.6875 - val_loss: 1028609.6875 - learning_rate: 0.0010
Epoch 167/450
19/19            0s 1ms/step - loss:
973993.5000 - val_loss: 1017934.3750 - learning_rate: 0.0010
Epoch 168/450
19/19            0s 1ms/step - loss:
971823.6250 - val_loss: 1018562.3750 - learning_rate: 0.0010
Epoch 169/450
19/19            0s 1ms/step - loss:
969888.0000 - val_loss: 1014748.3750 - learning_rate: 0.0010
Epoch 170/450
19/19            0s 1ms/step - loss:
955491.0625 - val_loss: 1006996.6875 - learning_rate: 0.0010
Epoch 171/450
19/19            0s 1ms/step - loss:
```

```
952666.1250 - val_loss: 992090.5625 - learning_rate: 0.0010
Epoch 172/450
19/19            0s 1ms/step - loss:
938962.7500 - val_loss: 979207.6875 - learning_rate: 0.0010
Epoch 173/450
19/19            0s 1ms/step - loss:
936849.4375 - val_loss: 964808.9375 - learning_rate: 0.0010
Epoch 174/450
19/19            0s 1ms/step - loss:
933908.1250 - val_loss: 955345.3750 - learning_rate: 0.0010
Epoch 175/450
19/19            0s 2ms/step - loss:
924508.8750 - val_loss: 954938.2500 - learning_rate: 0.0010
Epoch 176/450
19/19            0s 1ms/step - loss:
921109.0000 - val_loss: 953085.0000 - learning_rate: 0.0010
Epoch 177/450
19/19            0s 1ms/step - loss:
914896.0625 - val_loss: 959715.0000 - learning_rate: 0.0010
Epoch 178/450
19/19            0s 1ms/step - loss:
911556.7500 - val_loss: 965005.1250 - learning_rate: 0.0010
Epoch 179/450
19/19            0s 1ms/step - loss:
898507.6250 - val_loss: 960886.8125 - learning_rate: 0.0010
Epoch 180/450
19/19            0s 1ms/step - loss:
902505.9375 - val_loss: 961118.0625 - learning_rate: 0.0010
Epoch 181/450
19/19            0s 1ms/step - loss:
895656.0000 - val_loss: 948573.8750 - learning_rate: 0.0010
Epoch 182/450
19/19            0s 1ms/step - loss:
884494.8750 - val_loss: 938460.1875 - learning_rate: 0.0010
Epoch 183/450
19/19            0s 2ms/step - loss:
880049.3750 - val_loss: 934106.8750 - learning_rate: 0.0010
Epoch 184/450
19/19            0s 2ms/step - loss:
872382.0000 - val_loss: 926404.6875 - learning_rate: 0.0010
Epoch 185/450
19/19            0s 2ms/step - loss:
863973.6875 - val_loss: 922329.1875 - learning_rate: 0.0010
Epoch 186/450
19/19            0s 1ms/step - loss:
868126.6875 - val_loss: 912230.0000 - learning_rate: 0.0010
Epoch 187/450
19/19            0s 1ms/step - loss:
```

```
854408.0000 - val_loss: 901364.5625 - learning_rate: 0.0010
Epoch 188/450
19/19            0s 1ms/step - loss:
852184.3750 - val_loss: 892716.9375 - learning_rate: 0.0010
Epoch 189/450
19/19            0s 1ms/step - loss:
844948.3125 - val_loss: 886771.9375 - learning_rate: 0.0010
Epoch 190/450
19/19            0s 2ms/step - loss:
836671.3750 - val_loss: 876266.2500 - learning_rate: 0.0010
Epoch 191/450
19/19            0s 2ms/step - loss:
833730.2500 - val_loss: 862567.5000 - learning_rate: 0.0010
Epoch 192/450
19/19            0s 2ms/step - loss:
825430.1250 - val_loss: 863184.7500 - learning_rate: 0.0010
Epoch 193/450
19/19            0s 2ms/step - loss:
822805.1250 - val_loss: 861274.7500 - learning_rate: 0.0010
Epoch 194/450
19/19            0s 1ms/step - loss:
816059.4375 - val_loss: 861048.0000 - learning_rate: 0.0010
Epoch 195/450
19/19            0s 1ms/step - loss:
812081.8750 - val_loss: 856169.9375 - learning_rate: 0.0010
Epoch 196/450
19/19            0s 3ms/step - loss:
807794.5625 - val_loss: 844130.7500 - learning_rate: 0.0010
Epoch 197/450
19/19            0s 1ms/step - loss:
794156.7500 - val_loss: 848226.0000 - learning_rate: 0.0010
Epoch 198/450
19/19            0s 1ms/step - loss:
791262.2500 - val_loss: 842331.0000 - learning_rate: 0.0010
Epoch 199/450
19/19            0s 1ms/step - loss:
785394.2500 - val_loss: 826772.7500 - learning_rate: 0.0010
Epoch 200/450
19/19            0s 2ms/step - loss:
780483.6250 - val_loss: 818712.1875 - learning_rate: 0.0010
Epoch 201/450
19/19            0s 2ms/step - loss:
773991.9375 - val_loss: 817266.3750 - learning_rate: 0.0010
Epoch 202/450
19/19            0s 1ms/step - loss:
771513.1250 - val_loss: 823951.5625 - learning_rate: 0.0010
Epoch 203/450
19/19            0s 1ms/step - loss:
```

```
764231.3125 - val_loss: 819221.8125 - learning_rate: 0.0010
Epoch 204/450
19/19            0s 2ms/step - loss:
745910.6250 - val_loss: 805531.6875 - learning_rate: 0.0010
Epoch 205/450
19/19            0s 2ms/step - loss:
752541.1250 - val_loss: 804086.1875 - learning_rate: 0.0010
Epoch 206/450
19/19            0s 1ms/step - loss:
740953.7500 - val_loss: 800552.1250 - learning_rate: 0.0010
Epoch 207/450
19/19            0s 2ms/step - loss:
753295.1250 - val_loss: 799259.1250 - learning_rate: 0.0010
Epoch 208/450
19/19            0s 1ms/step - loss:
731810.3750 - val_loss: 792865.3750 - learning_rate: 0.0010
Epoch 209/450
19/19            0s 1ms/step - loss:
727747.4375 - val_loss: 783420.0625 - learning_rate: 0.0010
Epoch 210/450
19/19            0s 1ms/step - loss:
724710.1250 - val_loss: 780730.6250 - learning_rate: 0.0010
Epoch 211/450
19/19            0s 1ms/step - loss:
716236.3125 - val_loss: 769792.3125 - learning_rate: 0.0010
Epoch 212/450
19/19            0s 1ms/step - loss:
711330.0000 - val_loss: 765359.1875 - learning_rate: 0.0010
Epoch 213/450
19/19            0s 1ms/step - loss:
708807.0625 - val_loss: 766361.8125 - learning_rate: 0.0010
Epoch 214/450
19/19            0s 1ms/step - loss:
697083.4375 - val_loss: 761208.6875 - learning_rate: 0.0010
Epoch 215/450
19/19            0s 1ms/step - loss:
693321.5000 - val_loss: 749031.6875 - learning_rate: 0.0010
Epoch 216/450
19/19            0s 2ms/step - loss:
694932.5625 - val_loss: 745481.5000 - learning_rate: 0.0010
Epoch 217/450
19/19            0s 2ms/step - loss:
685417.1250 - val_loss: 744248.6250 - learning_rate: 0.0010
Epoch 218/450
19/19            0s 3ms/step - loss:
677787.9375 - val_loss: 731547.3125 - learning_rate: 0.0010
Epoch 219/450
19/19            0s 2ms/step - loss:
```

```
668770.6250 - val_loss: 721130.8750 - learning_rate: 0.0010
Epoch 220/450
19/19            0s 1ms/step - loss:
667479.5625 - val_loss: 720358.7500 - learning_rate: 0.0010
Epoch 221/450
19/19            0s 1ms/step - loss:
663840.8125 - val_loss: 715960.7500 - learning_rate: 0.0010
Epoch 222/450
19/19            0s 1ms/step - loss:
659406.5000 - val_loss: 710091.1875 - learning_rate: 0.0010
Epoch 223/450
19/19            0s 2ms/step - loss:
643984.3750 - val_loss: 696023.8750 - learning_rate: 0.0010
Epoch 224/450
19/19            0s 1ms/step - loss:
644615.8750 - val_loss: 692464.5000 - learning_rate: 0.0010
Epoch 225/450
19/19            0s 2ms/step - loss:
635932.8750 - val_loss: 684795.9375 - learning_rate: 0.0010
Epoch 226/450
19/19            0s 2ms/step - loss:
631369.3750 - val_loss: 678457.5625 - learning_rate: 0.0010
Epoch 227/450
19/19            0s 2ms/step - loss:
622245.3125 - val_loss: 677584.7500 - learning_rate: 0.0010
Epoch 228/450
19/19            0s 2ms/step - loss:
616589.8125 - val_loss: 682088.8750 - learning_rate: 0.0010
Epoch 229/450
19/19            0s 2ms/step - loss:
616610.3750 - val_loss: 677363.4375 - learning_rate: 0.0010
Epoch 230/450
19/19            0s 2ms/step - loss:
609429.9375 - val_loss: 670177.6875 - learning_rate: 0.0010
Epoch 231/450
19/19            0s 2ms/step - loss:
596289.1250 - val_loss: 657777.9375 - learning_rate: 0.0010
Epoch 232/450
19/19            0s 2ms/step - loss:
597236.6875 - val_loss: 652344.8750 - learning_rate: 0.0010
Epoch 233/450
19/19            0s 2ms/step - loss:
596411.4375 - val_loss: 649239.0625 - learning_rate: 0.0010
Epoch 234/450
19/19            0s 1ms/step - loss:
592643.5000 - val_loss: 650262.7500 - learning_rate: 0.0010
Epoch 235/450
19/19            0s 2ms/step - loss:
```

```
579556.6875 - val_loss: 634528.0000 - learning_rate: 0.0010
Epoch 236/450
19/19            0s 1ms/step - loss:
583122.8125 - val_loss: 632020.5625 - learning_rate: 0.0010
Epoch 237/450
19/19            0s 3ms/step - loss:
568040.1250 - val_loss: 622994.1250 - learning_rate: 0.0010
Epoch 238/450
19/19            0s 2ms/step - loss:
568328.2500 - val_loss: 613090.6875 - learning_rate: 0.0010
Epoch 239/450
19/19            0s 2ms/step - loss:
560588.3750 - val_loss: 603663.6875 - learning_rate: 0.0010
Epoch 240/450
19/19            0s 2ms/step - loss:
553064.0625 - val_loss: 594470.0000 - learning_rate: 0.0010
Epoch 241/450
19/19            0s 2ms/step - loss:
548490.8750 - val_loss: 585095.9375 - learning_rate: 0.0010
Epoch 242/450
19/19            0s 1ms/step - loss:
548069.1250 - val_loss: 579981.3750 - learning_rate: 0.0010
Epoch 243/450
19/19            0s 1ms/step - loss:
544601.2500 - val_loss: 570442.0625 - learning_rate: 0.0010
Epoch 244/450
19/19            0s 1ms/step - loss:
534586.6250 - val_loss: 562059.4375 - learning_rate: 0.0010
Epoch 245/450
19/19            0s 1ms/step - loss:
530248.8750 - val_loss: 562774.0000 - learning_rate: 0.0010
Epoch 246/450
19/19            0s 1ms/step - loss:
524404.2500 - val_loss: 558211.6875 - learning_rate: 0.0010
Epoch 247/450
19/19            0s 1ms/step - loss:
526672.3750 - val_loss: 557107.2500 - learning_rate: 0.0010
Epoch 248/450
19/19            0s 3ms/step - loss:
515379.5625 - val_loss: 548182.2500 - learning_rate: 0.0010
Epoch 249/450
19/19            0s 1ms/step - loss:
508976.7500 - val_loss: 540339.9375 - learning_rate: 0.0010
Epoch 250/450
19/19            0s 1ms/step - loss:
497502.8438 - val_loss: 533754.8750 - learning_rate: 0.0010
Epoch 251/450
19/19            0s 1ms/step - loss:
```

```
493977.4062 - val_loss: 530566.7500 - learning_rate: 0.0010
Epoch 252/450
19/19            0s 1ms/step - loss:
502051.7500 - val_loss: 534171.3750 - learning_rate: 0.0010
Epoch 253/450
19/19            0s 1ms/step - loss:
489217.5312 - val_loss: 532795.3125 - learning_rate: 0.0010
Epoch 254/450
19/19            0s 1ms/step - loss:
482748.4062 - val_loss: 527609.5625 - learning_rate: 0.0010
Epoch 255/450
19/19            0s 1ms/step - loss:
482493.6875 - val_loss: 524360.6250 - learning_rate: 0.0010
Epoch 256/450
19/19            0s 1ms/step - loss:
477923.3125 - val_loss: 520791.4688 - learning_rate: 0.0010
Epoch 257/450
19/19            0s 3ms/step - loss:
469729.2188 - val_loss: 515571.6250 - learning_rate: 0.0010
Epoch 258/450
19/19            0s 1ms/step - loss:
464518.3125 - val_loss: 513010.7188 - learning_rate: 0.0010
Epoch 259/450
19/19            0s 1ms/step - loss:
467470.7188 - val_loss: 503180.3750 - learning_rate: 0.0010
Epoch 260/450
19/19            0s 1ms/step - loss:
456269.9062 - val_loss: 492896.5312 - learning_rate: 0.0010
Epoch 261/450
19/19            0s 2ms/step - loss:
449111.5000 - val_loss: 484473.3750 - learning_rate: 0.0010
Epoch 262/450
19/19            0s 2ms/step - loss:
456507.2500 - val_loss: 486275.3125 - learning_rate: 0.0010
Epoch 263/450
19/19            0s 2ms/step - loss:
438380.5312 - val_loss: 477875.7812 - learning_rate: 0.0010
Epoch 264/450
19/19            0s 2ms/step - loss:
434928.0938 - val_loss: 470107.3750 - learning_rate: 0.0010
Epoch 265/450
19/19            0s 2ms/step - loss:
433595.7812 - val_loss: 468838.5000 - learning_rate: 0.0010
Epoch 266/450
19/19            0s 3ms/step - loss:
430818.8750 - val_loss: 468806.8750 - learning_rate: 0.0010
Epoch 267/450
19/19            0s 2ms/step - loss:
```

```
422253.5000 - val_loss: 464486.1875 - learning_rate: 0.0010
Epoch 268/450
19/19            0s 2ms/step - loss:
421841.7812 - val_loss: 453529.0000 - learning_rate: 0.0010
Epoch 269/450
19/19            0s 2ms/step - loss:
413422.6250 - val_loss: 442871.7188 - learning_rate: 0.0010
Epoch 270/450
19/19            0s 2ms/step - loss:
411357.6250 - val_loss: 434102.7812 - learning_rate: 0.0010
Epoch 271/450
19/19            0s 2ms/step - loss:
400030.4688 - val_loss: 432151.1562 - learning_rate: 0.0010
Epoch 272/450
19/19            0s 2ms/step - loss:
399470.3438 - val_loss: 429338.7188 - learning_rate: 0.0010
Epoch 273/450
19/19            0s 1ms/step - loss:
392782.5312 - val_loss: 429332.5000 - learning_rate: 0.0010
Epoch 274/450
19/19            0s 1ms/step - loss:
382724.1250 - val_loss: 426740.2812 - learning_rate: 0.0010
Epoch 275/450
19/19            0s 2ms/step - loss:
375622.2188 - val_loss: 420681.0312 - learning_rate: 0.0010
Epoch 276/450
19/19            0s 2ms/step - loss:
370037.5000 - val_loss: 419977.7500 - learning_rate: 0.0010
Epoch 277/450
19/19            0s 3ms/step - loss:
375927.3750 - val_loss: 416177.2500 - learning_rate: 0.0010
Epoch 278/450
19/19            0s 2ms/step - loss:
372296.0000 - val_loss: 409081.7188 - learning_rate: 0.0010
Epoch 279/450
19/19            0s 2ms/step - loss:
360648.7188 - val_loss: 399634.6562 - learning_rate: 0.0010
Epoch 280/450
19/19            0s 2ms/step - loss:
364088.4688 - val_loss: 395018.6562 - learning_rate: 0.0010
Epoch 281/450
19/19            0s 2ms/step - loss:
353640.7812 - val_loss: 389102.0938 - learning_rate: 0.0010
Epoch 282/450
19/19            0s 1ms/step - loss:
353955.7188 - val_loss: 392325.9062 - learning_rate: 0.0010
Epoch 283/450
19/19            0s 2ms/step - loss:
```

```
350582.0625 - val_loss: 382530.4688 - learning_rate: 0.0010
Epoch 284/450
19/19            0s 2ms/step - loss:
346648.6250 - val_loss: 376878.2500 - learning_rate: 0.0010
Epoch 285/450
19/19            0s 2ms/step - loss:
335872.6875 - val_loss: 372985.9688 - learning_rate: 0.0010
Epoch 286/450
19/19            0s 1ms/step - loss:
340786.3438 - val_loss: 371250.8125 - learning_rate: 0.0010
Epoch 287/450
19/19            0s 3ms/step - loss:
329281.5625 - val_loss: 371680.7188 - learning_rate: 0.0010
Epoch 288/450
19/19            0s 2ms/step - loss:
327814.5000 - val_loss: 365528.6250 - learning_rate: 0.0010
Epoch 289/450
19/19            0s 2ms/step - loss:
337912.3125 - val_loss: 360849.3750 - learning_rate: 0.0010
Epoch 290/450
19/19            0s 2ms/step - loss:
319597.9062 - val_loss: 356617.2812 - learning_rate: 0.0010
Epoch 291/450
19/19            0s 2ms/step - loss:
320208.3125 - val_loss: 347528.1562 - learning_rate: 0.0010
Epoch 292/450
19/19            0s 2ms/step - loss:
307492.8125 - val_loss: 339214.2812 - learning_rate: 0.0010
Epoch 293/450
19/19            0s 2ms/step - loss:
303753.9688 - val_loss: 340367.4375 - learning_rate: 0.0010
Epoch 294/450
19/19            0s 2ms/step - loss:
300377.0625 - val_loss: 335559.8438 - learning_rate: 0.0010
Epoch 295/450
19/19            0s 2ms/step - loss:
291298.9375 - val_loss: 329170.5312 - learning_rate: 0.0010
Epoch 296/450
19/19            0s 2ms/step - loss:
291752.0938 - val_loss: 325777.0938 - learning_rate: 0.0010
Epoch 297/450
19/19            0s 2ms/step - loss:
289818.9688 - val_loss: 323325.7188 - learning_rate: 0.0010
Epoch 298/450
19/19            0s 1ms/step - loss:
288862.1562 - val_loss: 320803.3750 - learning_rate: 0.0010
Epoch 299/450
19/19            0s 1ms/step - loss:
```

```
282841.8750 - val_loss: 316740.0938 - learning_rate: 0.0010
Epoch 300/450
19/19            0s 1ms/step - loss:
284480.1875 - val_loss: 311353.1875 - learning_rate: 0.0010
Epoch 301/450
19/19            0s 1ms/step - loss:
278348.0625 - val_loss: 307871.4688 - learning_rate: 0.0010
Epoch 302/450
19/19            0s 1ms/step - loss:
270046.1875 - val_loss: 301306.3125 - learning_rate: 0.0010
Epoch 303/450
19/19            0s 1ms/step - loss:
268512.1250 - val_loss: 298537.4375 - learning_rate: 0.0010
Epoch 304/450
19/19            0s 1ms/step - loss:
261102.3750 - val_loss: 292921.9062 - learning_rate: 0.0010
Epoch 305/450
19/19            0s 1ms/step - loss:
256570.9688 - val_loss: 291744.8125 - learning_rate: 0.0010
Epoch 306/450
19/19            0s 1ms/step - loss:
254621.0312 - val_loss: 294497.3750 - learning_rate: 0.0010
Epoch 307/450
19/19            0s 1ms/step - loss:
253210.9844 - val_loss: 287749.7500 - learning_rate: 0.0010
Epoch 308/450
19/19            0s 2ms/step - loss:
255979.2031 - val_loss: 280931.7188 - learning_rate: 0.0010
Epoch 309/450
19/19            0s 1ms/step - loss:
251738.1875 - val_loss: 278839.0938 - learning_rate: 0.0010
Epoch 310/450
19/19            0s 1ms/step - loss:
247440.9219 - val_loss: 278135.7188 - learning_rate: 0.0010
Epoch 311/450
19/19            0s 1ms/step - loss:
239802.7344 - val_loss: 271949.9062 - learning_rate: 0.0010
Epoch 312/450
19/19            0s 1ms/step - loss:
239712.9062 - val_loss: 268546.3125 - learning_rate: 0.0010
Epoch 313/450
19/19            0s 1ms/step - loss:
236299.0312 - val_loss: 264084.0625 - learning_rate: 0.0010
Epoch 314/450
19/19            0s 1ms/step - loss:
231394.1562 - val_loss: 263531.0000 - learning_rate: 0.0010
Epoch 315/450
19/19            0s 3ms/step - loss:
```

```
222304.8594 - val_loss: 261412.1406 - learning_rate: 0.0010
Epoch 316/450
19/19            0s 1ms/step - loss:
224761.7031 - val_loss: 256450.8750 - learning_rate: 0.0010
Epoch 317/450
19/19            0s 1ms/step - loss:
217694.5156 - val_loss: 250544.1875 - learning_rate: 0.0010
Epoch 318/450
19/19            0s 2ms/step - loss:
217504.7188 - val_loss: 248090.5938 - learning_rate: 0.0010
Epoch 319/450
19/19            0s 1ms/step - loss:
210955.5000 - val_loss: 246083.6250 - learning_rate: 0.0010
Epoch 320/450
19/19            0s 2ms/step - loss:
212344.8281 - val_loss: 241446.3438 - learning_rate: 0.0010
Epoch 321/450
19/19            0s 2ms/step - loss:
201651.9531 - val_loss: 236052.3750 - learning_rate: 0.0010
Epoch 322/450
19/19            0s 2ms/step - loss:
205452.5469 - val_loss: 229310.0781 - learning_rate: 0.0010
Epoch 323/450
19/19            0s 2ms/step - loss:
199387.0156 - val_loss: 226299.6875 - learning_rate: 0.0010
Epoch 324/450
19/19            0s 3ms/step - loss:
193773.0938 - val_loss: 225197.2812 - learning_rate: 0.0010
Epoch 325/450
19/19            0s 1ms/step - loss:
196898.2188 - val_loss: 225556.0000 - learning_rate: 0.0010
Epoch 326/450
19/19            0s 2ms/step - loss:
193651.4844 - val_loss: 222198.4375 - learning_rate: 0.0010
Epoch 327/450
19/19            0s 1ms/step - loss:
188330.1406 - val_loss: 214155.0000 - learning_rate: 0.0010
Epoch 328/450
19/19            0s 2ms/step - loss:
182957.4062 - val_loss: 206186.6250 - learning_rate: 0.0010
Epoch 329/450
19/19            0s 1ms/step - loss:
180757.2812 - val_loss: 205479.7812 - learning_rate: 0.0010
Epoch 330/450
19/19            0s 1ms/step - loss:
185247.7500 - val_loss: 202506.9375 - learning_rate: 0.0010
Epoch 331/450
19/19            0s 1ms/step - loss:
```

```
177912.9375 - val_loss: 201429.2344 - learning_rate: 0.0010
Epoch 332/450
19/19            0s 1ms/step - loss:
180450.4375 - val_loss: 194461.4062 - learning_rate: 0.0010
Epoch 333/450
19/19            0s 2ms/step - loss:
168738.7500 - val_loss: 191483.9688 - learning_rate: 0.0010
Epoch 334/450
19/19            0s 1ms/step - loss:
167411.7500 - val_loss: 193374.1094 - learning_rate: 0.0010
Epoch 335/450
19/19            0s 3ms/step - loss:
170025.6094 - val_loss: 196232.1875 - learning_rate: 0.0010
Epoch 336/450
19/19            0s 1ms/step - loss:
164857.7656 - val_loss: 189397.0156 - learning_rate: 0.0010
Epoch 337/450
19/19            0s 1ms/step - loss:
157546.9688 - val_loss: 182992.0938 - learning_rate: 0.0010
Epoch 338/450
19/19            0s 1ms/step - loss:
161020.9688 - val_loss: 182009.7656 - learning_rate: 0.0010
Epoch 339/450
19/19            0s 1ms/step - loss:
151244.9531 - val_loss: 183185.0781 - learning_rate: 0.0010
Epoch 340/450
19/19            0s 1ms/step - loss:
157562.0000 - val_loss: 180262.7656 - learning_rate: 0.0010
Epoch 341/450
19/19            0s 1ms/step - loss:
152069.2188 - val_loss: 176164.1562 - learning_rate: 0.0010
Epoch 342/450
19/19            0s 2ms/step - loss:
149392.4844 - val_loss: 170195.6406 - learning_rate: 0.0010
Epoch 343/450
19/19            0s 1ms/step - loss:
143520.8906 - val_loss: 167482.3125 - learning_rate: 0.0010
Epoch 344/450
19/19            0s 1ms/step - loss:
148385.6875 - val_loss: 165548.4688 - learning_rate: 0.0010
Epoch 345/450
19/19            0s 1ms/step - loss:
141305.2344 - val_loss: 162682.7188 - learning_rate: 0.0010
Epoch 346/450
19/19            0s 1ms/step - loss:
139428.9062 - val_loss: 160752.5938 - learning_rate: 0.0010
Epoch 347/450
19/19            0s 1ms/step - loss:
```

```
137673.2344 - val_loss: 156598.7656 - learning_rate: 0.0010
Epoch 348/450
19/19            0s 1ms/step - loss:
127447.5781 - val_loss: 151842.6562 - learning_rate: 0.0010
Epoch 349/450
19/19            0s 1ms/step - loss:
130595.7812 - val_loss: 148853.3594 - learning_rate: 0.0010
Epoch 350/450
19/19            0s 1ms/step - loss:
128420.2500 - val_loss: 145681.8906 - learning_rate: 0.0010
Epoch 351/450
19/19            0s 1ms/step - loss:
127346.4297 - val_loss: 147391.7656 - learning_rate: 0.0010
Epoch 352/450
19/19            0s 3ms/step - loss:
120733.9688 - val_loss: 147393.0938 - learning_rate: 0.0010
Epoch 353/450
19/19            0s 2ms/step - loss:
118020.8906 - val_loss: 144386.1094 - learning_rate: 0.0010
Epoch 354/450
19/19            0s 2ms/step - loss:
116001.0859 - val_loss: 140628.5000 - learning_rate: 0.0010
Epoch 355/450
19/19            0s 2ms/step - loss:
113086.0547 - val_loss: 135641.0156 - learning_rate: 0.0010
Epoch 356/450
19/19            0s 2ms/step - loss:
114822.6953 - val_loss: 129691.9297 - learning_rate: 0.0010
Epoch 357/450
19/19            0s 2ms/step - loss:
107294.2812 - val_loss: 124280.2109 - learning_rate: 0.0010
Epoch 358/450
19/19            0s 2ms/step - loss:
112247.8203 - val_loss: 123542.8516 - learning_rate: 0.0010
Epoch 359/450
19/19            0s 3ms/step - loss:
109440.2422 - val_loss: 123221.4922 - learning_rate: 0.0010
Epoch 360/450
19/19            0s 2ms/step - loss:
109803.8359 - val_loss: 122067.0391 - learning_rate: 0.0010
Epoch 361/450
19/19            0s 2ms/step - loss:
104230.9844 - val_loss: 119205.2031 - learning_rate: 0.0010
Epoch 362/450
19/19            0s 2ms/step - loss:
109244.9062 - val_loss: 118967.8281 - learning_rate: 0.0010
Epoch 363/450
19/19            0s 1ms/step - loss:
```

```
105322.7734 - val_loss: 118268.9609 - learning_rate: 0.0010
Epoch 364/450
19/19            0s 2ms/step - loss:
101020.3281 - val_loss: 116279.3203 - learning_rate: 0.0010
Epoch 365/450
19/19            0s 2ms/step - loss:
96946.1172 - val_loss: 114809.5469 - learning_rate: 0.0010
Epoch 366/450
19/19            0s 1ms/step - loss:
100811.1484 - val_loss: 113432.4688 - learning_rate: 0.0010
Epoch 367/450
19/19            0s 1ms/step - loss:
88656.4062 - val_loss: 111534.1328 - learning_rate: 0.0010
Epoch 368/450
19/19            0s 3ms/step - loss:
90774.9922 - val_loss: 108296.6953 - learning_rate: 0.0010
Epoch 369/450
19/19            0s 1ms/step - loss:
94499.3516 - val_loss: 104515.6953 - learning_rate: 0.0010
Epoch 370/450
19/19            0s 1ms/step - loss:
90725.2500 - val_loss: 101019.1797 - learning_rate: 0.0010
Epoch 371/450
19/19            0s 1ms/step - loss:
86615.1562 - val_loss: 97781.1172 - learning_rate: 0.0010
Epoch 372/450
19/19            0s 1ms/step - loss:
88720.6016 - val_loss: 96050.2500 - learning_rate: 0.0010
Epoch 373/450
19/19            0s 1ms/step - loss:
83727.5859 - val_loss: 93012.1406 - learning_rate: 0.0010
Epoch 374/450
19/19            0s 1ms/step - loss:
80436.6562 - val_loss: 93525.4453 - learning_rate: 0.0010
Epoch 375/450
19/19            0s 2ms/step - loss:
74956.2891 - val_loss: 94565.8828 - learning_rate: 0.0010
Epoch 376/450
19/19            0s 1ms/step - loss:
78247.9922 - val_loss: 94917.4922 - learning_rate: 0.0010
Epoch 377/450
19/19            0s 1ms/step - loss:
78839.2656 - val_loss: 96560.9297 - learning_rate: 0.0010
Epoch 378/450
19/19            0s 1ms/step - loss:
77011.3594 - val_loss: 93389.0078 - learning_rate: 0.0010
Epoch 379/450
19/19            0s 1ms/step - loss:
```

```
73873.9531 - val_loss: 87214.1484 - learning_rate: 0.0010
Epoch 380/450
19/19            0s 1ms/step - loss:
77495.5625 - val_loss: 85319.0547 - learning_rate: 0.0010
Epoch 381/450
19/19            0s 1ms/step - loss:
70558.9141 - val_loss: 84167.6016 - learning_rate: 0.0010
Epoch 382/450
19/19            0s 3ms/step - loss:
69114.1172 - val_loss: 81251.0000 - learning_rate: 0.0010
Epoch 383/450
19/19            0s 1ms/step - loss:
72969.8750 - val_loss: 79236.6641 - learning_rate: 0.0010
Epoch 384/450
19/19            0s 1ms/step - loss:
64313.1641 - val_loss: 80400.6016 - learning_rate: 0.0010
Epoch 385/450
19/19            0s 2ms/step - loss:
66116.3047 - val_loss: 78379.3750 - learning_rate: 0.0010
Epoch 386/450
19/19            0s 2ms/step - loss:
63035.5312 - val_loss: 77653.3828 - learning_rate: 0.0010
Epoch 387/450
19/19            0s 2ms/step - loss:
63714.7031 - val_loss: 76144.4453 - learning_rate: 0.0010
Epoch 388/450
19/19            0s 2ms/step - loss:
59340.1016 - val_loss: 73309.4688 - learning_rate: 0.0010
Epoch 389/450
19/19            0s 2ms/step - loss:
59537.4609 - val_loss: 74502.0156 - learning_rate: 0.0010
Epoch 390/450
19/19            0s 2ms/step - loss:
60021.6445 - val_loss: 73019.6562 - learning_rate: 0.0010
Epoch 391/450
19/19            0s 3ms/step - loss:
62804.0703 - val_loss: 70730.4844 - learning_rate: 0.0010
Epoch 392/450
19/19            0s 2ms/step - loss:
61317.9023 - val_loss: 70863.7188 - learning_rate: 0.0010
Epoch 393/450
19/19            0s 2ms/step - loss:
55811.9102 - val_loss: 67463.0703 - learning_rate: 0.0010
Epoch 394/450
19/19            0s 3ms/step - loss:
55890.7266 - val_loss: 65815.2109 - learning_rate: 0.0010
Epoch 395/450
19/19            0s 2ms/step - loss:
```

```
57825.5703 - val_loss: 64554.9805 - learning_rate: 0.0010
Epoch 396/450
19/19            0s 2ms/step - loss:
56216.4844 - val_loss: 62578.5469 - learning_rate: 0.0010
Epoch 397/450
19/19            0s 2ms/step - loss:
54341.5039 - val_loss: 59358.9609 - learning_rate: 0.0010
Epoch 398/450
19/19            0s 1ms/step - loss:
52488.1562 - val_loss: 60250.7383 - learning_rate: 0.0010
Epoch 399/450
19/19            0s 1ms/step - loss:
52414.2422 - val_loss: 58703.3398 - learning_rate: 0.0010
Epoch 400/450
19/19            0s 1ms/step - loss:
53485.1562 - val_loss: 56578.4609 - learning_rate: 0.0010
Epoch 401/450
19/19            0s 2ms/step - loss:
51540.8945 - val_loss: 55143.0742 - learning_rate: 0.0010
Epoch 402/450
19/19            0s 1ms/step - loss:
46445.2344 - val_loss: 54569.4258 - learning_rate: 0.0010
Epoch 403/450
19/19            0s 1ms/step - loss:
48632.0703 - val_loss: 54184.1836 - learning_rate: 0.0010
Epoch 404/450
19/19            0s 2ms/step - loss:
44713.0273 - val_loss: 51514.8828 - learning_rate: 0.0010
Epoch 405/450
19/19            0s 2ms/step - loss:
44308.1836 - val_loss: 49812.6094 - learning_rate: 0.0010
Epoch 406/450
19/19            0s 2ms/step - loss:
47124.6484 - val_loss: 49549.8750 - learning_rate: 0.0010
Epoch 407/450
19/19            0s 2ms/step - loss:
43758.6680 - val_loss: 48956.3750 - learning_rate: 0.0010
Epoch 408/450
19/19            0s 2ms/step - loss:
46608.5352 - val_loss: 48642.5859 - learning_rate: 0.0010
Epoch 409/450
19/19            0s 2ms/step - loss:
44099.9414 - val_loss: 48444.4883 - learning_rate: 0.0010
Epoch 410/450
19/19            0s 2ms/step - loss:
42520.7422 - val_loss: 48575.6055 - learning_rate: 0.0010
Epoch 411/450
19/19            0s 1ms/step - loss:
```

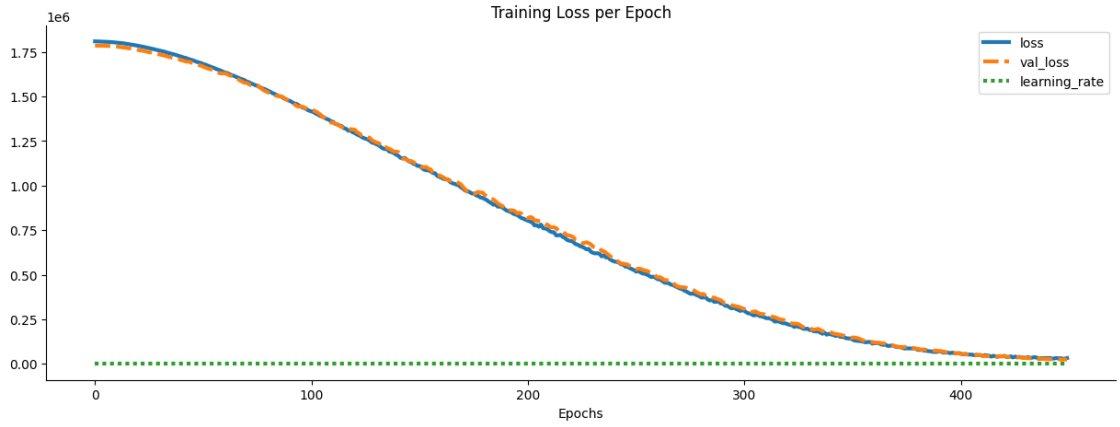
```
40480.8320 - val_loss: 44872.7305 - learning_rate: 0.0010
Epoch 412/450
19/19            0s 2ms/step - loss:
42925.7930 - val_loss: 43495.2266 - learning_rate: 0.0010
Epoch 413/450
19/19            0s 1ms/step - loss:
42205.0195 - val_loss: 43873.3750 - learning_rate: 0.0010
Epoch 414/450
19/19            0s 1ms/step - loss:
41111.3359 - val_loss: 43375.1914 - learning_rate: 0.0010
Epoch 415/450
19/19            0s 1ms/step - loss:
35141.0352 - val_loss: 44782.3711 - learning_rate: 0.0010
Epoch 416/450
19/19            0s 2ms/step - loss:
42097.9062 - val_loss: 46169.8125 - learning_rate: 0.0010
Epoch 417/450
19/19            0s 1ms/step - loss:
35982.5586 - val_loss: 42378.4883 - learning_rate: 0.0010
Epoch 418/450
19/19            0s 1ms/step - loss:
36279.5586 - val_loss: 41360.2344 - learning_rate: 0.0010
Epoch 419/450
19/19            0s 2ms/step - loss:
38978.8008 - val_loss: 41915.8516 - learning_rate: 0.0010
Epoch 420/450
19/19            0s 1ms/step - loss:
36470.2188 - val_loss: 42950.7148 - learning_rate: 0.0010
Epoch 421/450
19/19            0s 3ms/step - loss:
36348.3672 - val_loss: 43026.3008 - learning_rate: 0.0010
Epoch 422/450
19/19            0s 1ms/step - loss:
35488.2891 - val_loss: 39589.0391 - learning_rate: 0.0010
Epoch 423/450
19/19            0s 1ms/step - loss:
36132.2578 - val_loss: 37636.3906 - learning_rate: 0.0010
Epoch 424/450
19/19            0s 1ms/step - loss:
37782.0859 - val_loss: 37210.9297 - learning_rate: 0.0010
Epoch 425/450
19/19            0s 1ms/step - loss:
30909.2363 - val_loss: 35698.2539 - learning_rate: 0.0010
Epoch 426/450
19/19            0s 3ms/step - loss:
39340.4922 - val_loss: 35327.9336 - learning_rate: 0.0010
Epoch 427/450
19/19            0s 1ms/step - loss:
```

```
33593.7969 - val_loss: 35469.9648 - learning_rate: 0.0010
Epoch 428/450
19/19            0s 1ms/step - loss:
34391.1523 - val_loss: 36591.4648 - learning_rate: 0.0010
Epoch 429/450
19/19            0s 1ms/step - loss:
32903.1758 - val_loss: 35424.3789 - learning_rate: 0.0010
Epoch 430/450
19/19            0s 1ms/step - loss:
31462.4277 - val_loss: 33141.6875 - learning_rate: 0.0010
Epoch 431/450
19/19            0s 1ms/step - loss:
31617.8145 - val_loss: 32112.4727 - learning_rate: 0.0010
Epoch 432/450
19/19            0s 1ms/step - loss:
30286.0488 - val_loss: 31719.2207 - learning_rate: 0.0010
Epoch 433/450
19/19            0s 2ms/step - loss:
30091.7695 - val_loss: 30921.2207 - learning_rate: 0.0010
Epoch 434/450
19/19            0s 1ms/step - loss:
30652.6152 - val_loss: 29433.8203 - learning_rate: 0.0010
Epoch 435/450
19/19            0s 1ms/step - loss:
35580.3008 - val_loss: 30289.5605 - learning_rate: 0.0010
Epoch 436/450
19/19            0s 1ms/step - loss:
30506.9238 - val_loss: 28553.6992 - learning_rate: 0.0010
Epoch 437/450
19/19            0s 1ms/step - loss:
24741.4180 - val_loss: 27899.4980 - learning_rate: 0.0010
Epoch 438/450
19/19            0s 1ms/step - loss:
28129.7461 - val_loss: 27913.2168 - learning_rate: 0.0010
Epoch 439/450
19/19            0s 1ms/step - loss:
27636.4102 - val_loss: 27498.1582 - learning_rate: 0.0010
Epoch 440/450
19/19            0s 2ms/step - loss:
30850.2422 - val_loss: 27262.8594 - learning_rate: 0.0010
Epoch 441/450
19/19            0s 1ms/step - loss:
28399.5586 - val_loss: 26931.9707 - learning_rate: 0.0010
Epoch 442/450
19/19            0s 1ms/step - loss:
30987.3477 - val_loss: 25921.2773 - learning_rate: 0.0010
Epoch 443/450
19/19            0s 1ms/step - loss:
```

```
26159.1875 - val_loss: 26429.6816 - learning_rate: 0.0010
Epoch 444/450
19/19             0s 1ms/step - loss:
26100.8965 - val_loss: 26718.6035 - learning_rate: 0.0010
Epoch 445/450
19/19             0s 1ms/step - loss:
30358.0820 - val_loss: 25873.8633 - learning_rate: 0.0010
Epoch 446/450
19/19             0s 2ms/step - loss:
25888.7734 - val_loss: 24774.5605 - learning_rate: 0.0010
Epoch 447/450
19/19             0s 1ms/step - loss:
27374.4785 - val_loss: 24095.1660 - learning_rate: 0.0010
Epoch 448/450
19/19             0s 1ms/step - loss:
25804.2930 - val_loss: 24524.4668 - learning_rate: 0.0010
Epoch 449/450
19/19             0s 1ms/step - loss:
26449.8496 - val_loss: 24396.0879 - learning_rate: 0.0010
Epoch 450/450
19/19             0s 1ms/step - loss:
29587.4805 - val_loss: 22355.7500 - learning_rate: 0.0010
```

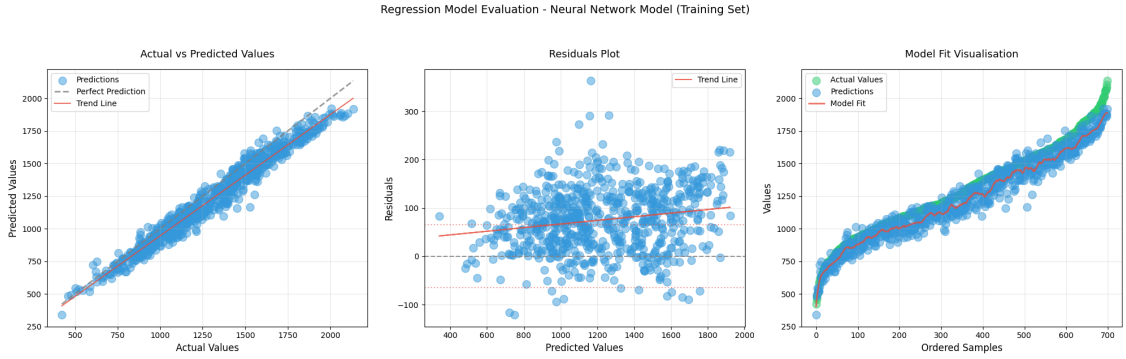
[134]:
```
<keras.src.callbacks.history.History at 0x34c3b14b0>
```

[135]:
```
losses = pd.DataFrame(NNModel.history.history)

plt.figure(figsize=(15,5))
sns.lineplot(data=losses,lw=3)
plt.xlabel('Epochs')
plt.ylabel('')
plt.title('Training Loss per Epoch')
sns.despine()
plt.show()
```



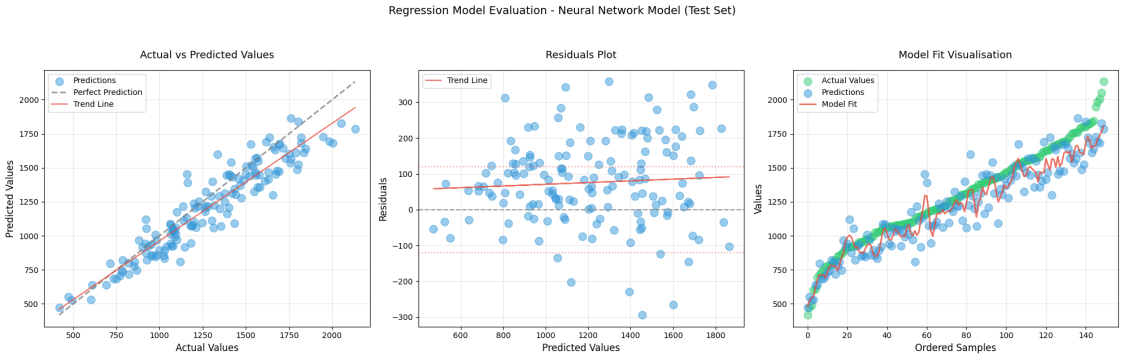
[136]:
```
regression_evaluation(NNModel, X_train_processed, y_train_reg, "Neural Network␣
↪Model (Training Set)")
```

```
Raw predictions shape        : (700, 1)
Flattened predictions shape  : (700,)
Target shape                 : (700,)

--- Neural Network Model (Training Set) Evaluation Metrics ---
------------------------------------------------------------
    RMSE: 99.62
    R²   : 0.92
    MAE : 82.59
------------------------------------------------------------
```



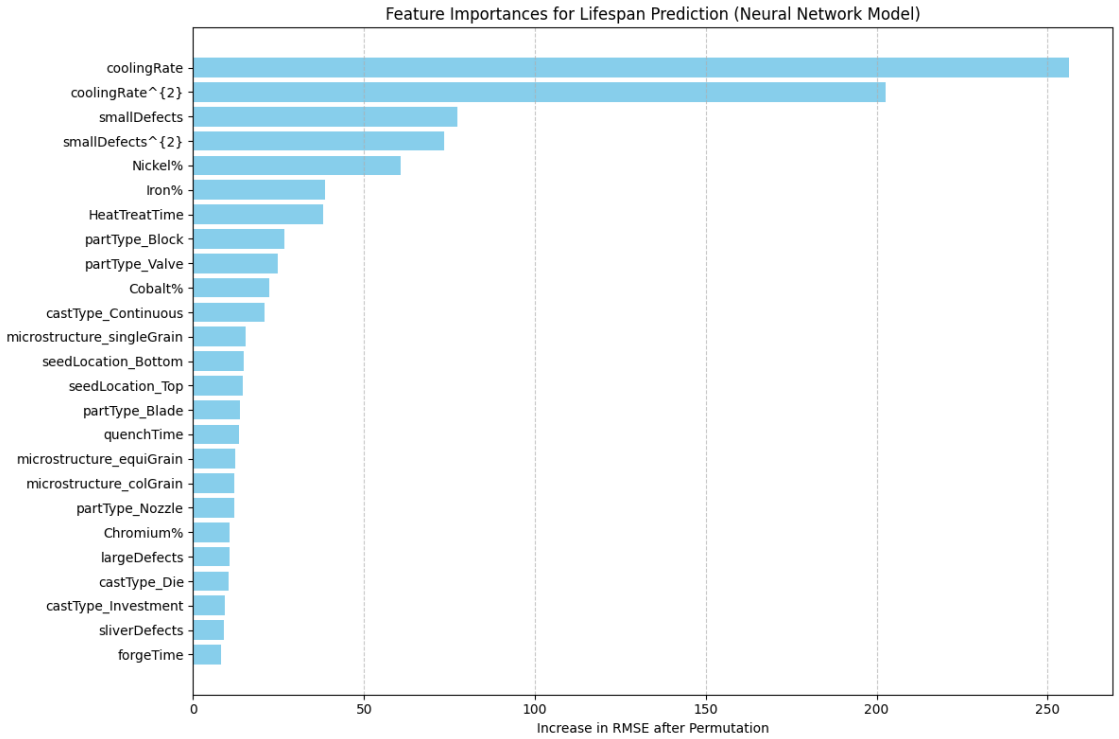
[137]:
```
regression_evaluation(NNModel, X_test_processed, y_test_reg, "Neural Network␣
↪Model (Test Set)")
```

```
Raw predictions shape        : (150, 1)
Flattened predictions shape  : (150,)
Target shape                 : (150,)

--- Neural Network Model (Test Set) Evaluation Metrics ---
----------------------------------------------------------
    RMSE: 142.17
    R²   : 0.84
    MAE : 112.83
----------------------------------------------------------
```



[138]:
```
# Get feature names from your numerical columns
feature_names = numerical_cols

# Baseline performance on the training set
y_pred = NNModel.predict(X_train_processed, verbose=0)
baseline_rmse = root_mean_squared_error(y_train_reg, y_pred)

# Initialize list to store importance scores
feature_importances = []

# Number of times to permute each feature
num_repeats = 5

# Loop over all features to compute importance
for i in range(X_train_processed.shape[1]):
    importance_scores = []
    for _ in range(num_repeats):
        # Copy the original data to avoid modifying it
        X_permuted = X_train_processed.copy()
        # Shuffle the values of the current feature
        np.random.shuffle(X_permuted[:, i])
        # Predict with the permuted data
        y_pred_permuted = NNModel.predict(X_permuted, verbose=0)
        # Calculate new RMSE
        permuted_rmse = root_mean_squared_error(y_train_reg, y_pred_permuted)
        # Importance is the increase in RMSE
        importance = permuted_rmse - baseline_rmse
        importance_scores.append(importance)
    # Average importance over the repeats
    feature_importances.append(np.mean(importance_scores))

# Create a DataFrame to hold feature importances
importance_df = pd.DataFrame({
    'Feature': feature_names,
```

```
    'Importance': feature_importances
})

# Sort features by importance in descending order
importance_df.sort_values(by='Importance', ascending=False, inplace=True)

# Plot horizontal bar chart
plt.figure(figsize=(12, 8))
plt.barh(importance_df['Feature'], importance_df['Importance'], color='skyblue')
plt.xlabel('Increase in RMSE after Permutation')
plt.title('Feature Importances for Lifespan Prediction (Neural Network Model)')
plt.gca().invert_yaxis()  # Keep this to maintain top-to-bottom ordering
plt.grid(True, axis='x', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```

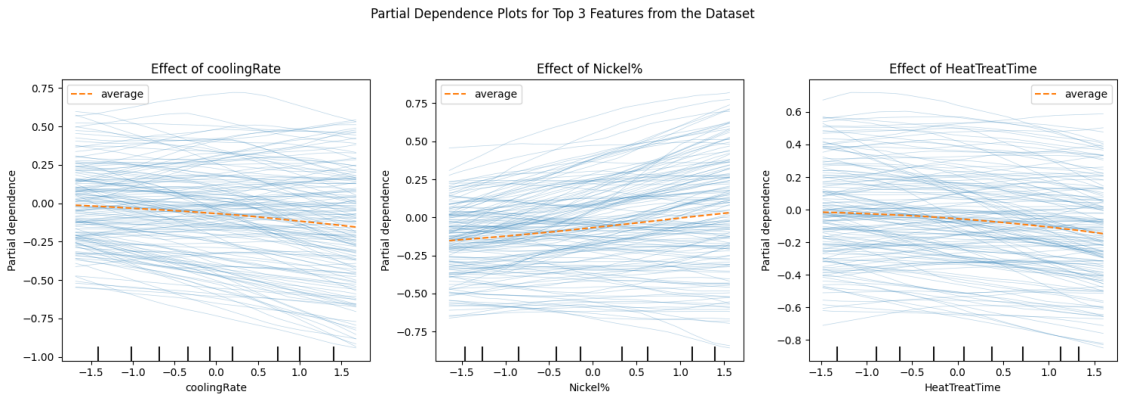


[172]:
```
# 1. Modify the wrapper class
class KerasRegressorWrapper(BaseEstimator, RegressorMixin):
    def __init__(self, model):
        self.model = model

    def fit(self, X, y):
        # Set a fitted attribute
```

```
        self.n_features_in_ = X.shape[1]
        return self

    def predict(self, X):
        return self.model.predict(X, verbose=0).flatten()

# 2. Prepare the processed data with feature names
X_test_processed_df = pd.DataFrame(X_test_processed, columns=numerical_cols)

# 3. Wrap the model
NNModel_Wrapped = KerasRegressorWrapper(NNModel)

# 4. Fit the wrapper (even though the model is already trained)
NNModel_Wrapped.fit(X_test_processed_df, y_train_reg)
```

[172]:
```
KerasRegressorWrapper(model=<Sequential name=sequential_2, built=True>)
```

[173]:
```
plot_partial_dependence(
    NNModel_Wrapped,
    X_test_processed_df,
    feature_names,
    feature_importances,
    processed_data=True
)
```



2 Classification Implementation

[175]:
```
# Utility function for creating Lifetime labels based on lifespan distribution␣
↪using K-Means Clustering
def create_labels(df, threshold_value=1500):
    # TODO: Add silhouette score
```

```
    df['Lifetime'] = np.where(df['Lifespan'] < threshold_value, 'Unacceptable',␣
↪None)
    acceptable_df = df[df['Lifespan'] >= threshold_value].copy()

    # K-Means Clustering to create labels beyond 'Unacceptable'
    if len(acceptable_df) > 1:
        X_acceptable = acceptable_df[['Lifespan']]
        max_k = min(10, len(acceptable_df))
        k_values = range(1, max_k)

        # Finding the optimal number of clusters
        inertia = [KMeans(n_clusters=k, random_state=random_state).
↪fit(X_acceptable).inertia_ for k in k_values]
        kneedle = KneeLocator(k_values, inertia, curve='convex',␣
↪direction='decreasing')
        elbow_k = kneedle.elbow if kneedle.elbow is not None else 2

        kmeans = KMeans(n_clusters=elbow_k, random_state=random_state)
        acceptable_df['cluster'] = kmeans.fit_predict(X_acceptable)

        # Create labels based on clustering
        cluster_means = acceptable_df.groupby('cluster')['Lifespan'].mean()
        cluster_order = cluster_means.sort_values().index.tolist()
        desired_labels = ['Fair', 'Good', 'Excellent']

        # Assign appropriate labels
        labels = {cluster_order[i]: desired_labels[i] if i <␣
↪len(desired_labels) else f'Cluster-{i}' for i in range(elbow_k)}
        acceptable_df['Lifetime'] = acceptable_df['cluster'].map(labels)

        # Update main dataframe
        df.loc[acceptable_df.index, 'Lifetime'] = acceptable_df['Lifetime']
        df.drop(columns=['cluster'], errors='ignore', inplace=True)

        # Cell 6: Plot the Elbow Method
        plt.figure(figsize=(10, 6))
        # Plot line segments with different colors
        plt.plot(k_values[:elbow_k], inertia[:elbow_k], 'bo-',␣
↪label="Decreasing Phase")
        plt.plot(k_values[elbow_k - 1:], inertia[elbow_k - 1:], 'go-',␣
↪label="Slow Decrease Phase")

        # Vertical line at elbow
        plt.axvline(x=elbow_k, linestyle='--', color='r', label=f'Optimal $k$ =␣
↪{elbow_k}')
```

```
        # Highlight the elbow point with a red marker and annotation
        plt.plot(elbow_k, inertia[elbow_k - 1], 'ro')  # red point at elbow
        plt.annotate(f"Optimal $k$ = {elbow_k}", xy=(elbow_k, inertia[elbow_k -␣
 ↪1]),
                     xytext=(elbow_k + 1, inertia[elbow_k - 1] + 0.2e7),
                     arrowprops=dict(facecolor='black', arrowstyle="->"))

        # Annotate each segment with inertia differences
        for i in range(1, len(k_values)):
            plt.annotate(f"{inertia[i-1] - inertia[i]:.2e}",
                         (k_values[i] - 0.5, (inertia[i-1] + inertia[i]) / 2),
                         fontsize=8, color='gray')

        # Set plot labels and title
        plt.xlabel(f'Number of Clusters ($k$)')
        plt.ylabel('Inertia')
        plt.title(f'Elbow Method for Optimal $k$')
        plt.grid(color='grey', linestyle='--', linewidth=0.5)
        plt.legend()
        plt.show()

    else:
        print("Not enough data to perform clustering on acceptable parts.")
    return df
```

[194]:
```
# Creating a new dataframe with the transformed ploynomial features for␣
 ↪classification
clf_df = rename_columns(df_transformed)

# Extracting the numerical, categorical and polynomial features from the␣
 ↪dataframe
numerical_features, categorical_features, polynomial_features =␣
 ↪extract_features(clf_df, target='Lifespan')
```

```
Categorical features: 4
- partType
- microstructure
- seedLocation
- castType

Numerical features: 13
- quenchTime
- Nickel%
- Iron%
- Cobalt%
- Chromium%
- largeDefects
```

```
- forgeTime
- HeatTreatTime
- sliverDefects
- coolingRate
- coolingRate^{2}
- smallDefects
- smallDefects^{2}

Polynomial features: 2
- coolingRate^{2}
- smallDefects^{2}
```

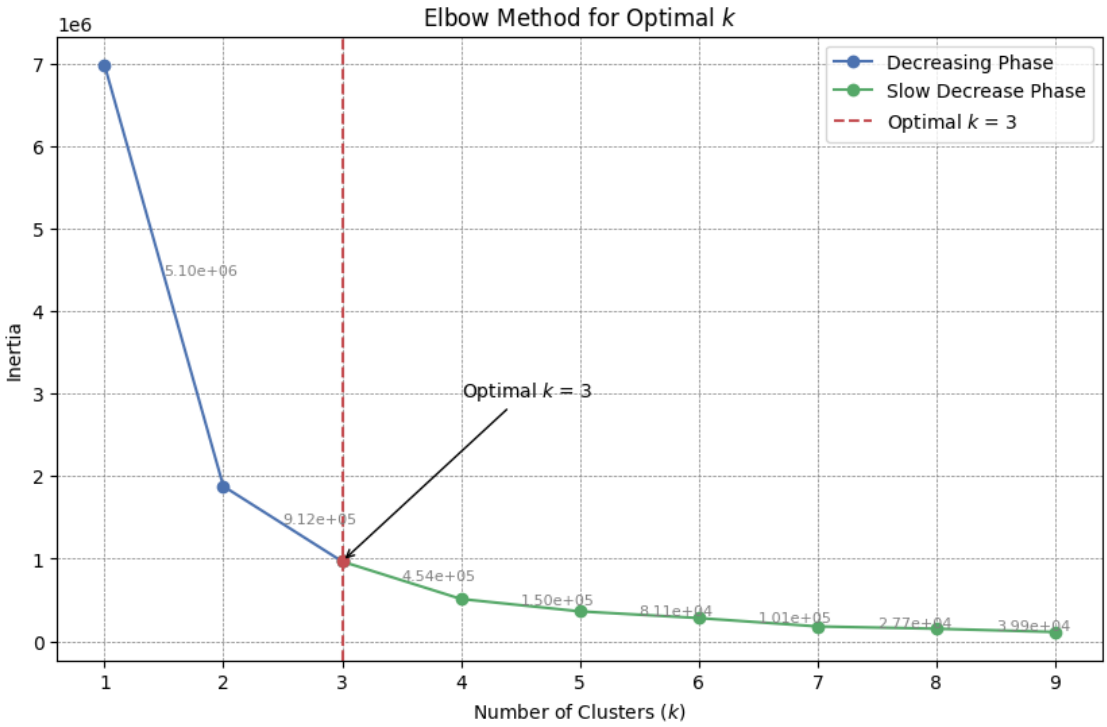
	quenchTime	Nickel%	Iron%	Cobalt%	Chromium%	largeDefects	Lifespan	\
0	3.84	65.73	16.52	16.82	0.93	0	1469.17	
1	2.62	54.22	35.38	6.14	4.26	0	1793.64	
2	0.76	51.83	35.95	8.81	3.41	3	700.60	
3	2.01	57.03	23.33	16.86	2.78	1	1082.10	
4	4.13	59.62	27.37	11.45	1.56	0	1838.83	

	partType	microstructure	forgeTime	HeatTreatTime	sliverDefects	\
0	Nozzle	equiGrain	6.47	46.87	0	
1	Block	singleGrain	3.48	44.70	0	
2	Blade	equiGrain	1.34	9.54	0	
3	Nozzle	colGrain	2.19	20.29	0	
4	Blade	colGrain	3.87	16.13	0	

	seedLocation	castType	coolingRate	coolingRate^{2}	smallDefects	\
0	Bottom	Die	13.0	169.0	10.0	
1	Bottom	Investment	19.0	361.0	19.0	
2	Bottom	Investment	28.0	784.0	35.0	
3	Top	Continuous	9.0	81.0	0.0	
4	Top	Die	16.0	256.0	10.0	

	smallDefects^{2}
0	100.0
1	361.0
2	1225.0
3	0.0
4	100.0

[177]:
```
# Feature Engineering - Create Lifetime Labels
clf_df = create_labels(clf_df)
```



[178]:
```
# View the loaded data
clf_df.head()
```

[178]:

	quenchTime	Nickel%	Iron%	Cobalt%	Chromium%	largeDefects	Lifespan	\
0	3.84	65.73	16.52	16.82	0.93	0	1469.17	
1	2.62	54.22	35.38	6.14	4.26	0	1793.64	
2	0.76	51.83	35.95	8.81	3.41	3	700.60	
3	2.01	57.03	23.33	16.86	2.78	1	1082.10	
4	4.13	59.62	27.37	11.45	1.56	0	1838.83	

	partType	microstructure	forgeTime	HeatTreatTime	sliverDefects	\
0	Nozzle	equiGrain	6.47	46.87	0	
1	Block	singleGrain	3.48	44.70	0	
2	Blade	equiGrain	1.34	9.54	0	
3	Nozzle	colGrain	2.19	20.29	0	
4	Blade	colGrain	3.87	16.13	0	

	seedLocation	castType	coolingRate	coolingRate^{2}	smallDefects	\
0	Bottom	Die	13.0	169.0	10.0	
1	Bottom	Investment	19.0	361.0	19.0	
2	Bottom	Investment	28.0	784.0	35.0	
3	Top	Continuous	9.0	81.0	0.0	
4	Top	Die	16.0	256.0	10.0	

```
   smallDefects^{2}      Lifetime
0             100.0  Unacceptable
1             361.0          Good
2            1225.0  Unacceptable
3               0.0  Unacceptable
4             100.0          Good
```

[179]:
```
# Display the cluster ranges
# Group the data by 'Lifetime' and aggregate to find the min and max Lifespan␣
 ↪for each group
cluster_ranges = clf_df.groupby('Lifetime')['Lifespan'].agg(['min', 'max']).
 ↪sort_values(by='min').reset_index()
# Display the sorted DataFrame
display(cluster_ranges)
```

```
      Lifetime      min      max
0  Unacceptable   417.99  1499.31
1          Fair  1501.76  1661.54
2          Good  1666.64  1850.75
3     Excellent  1854.50  2134.53
```

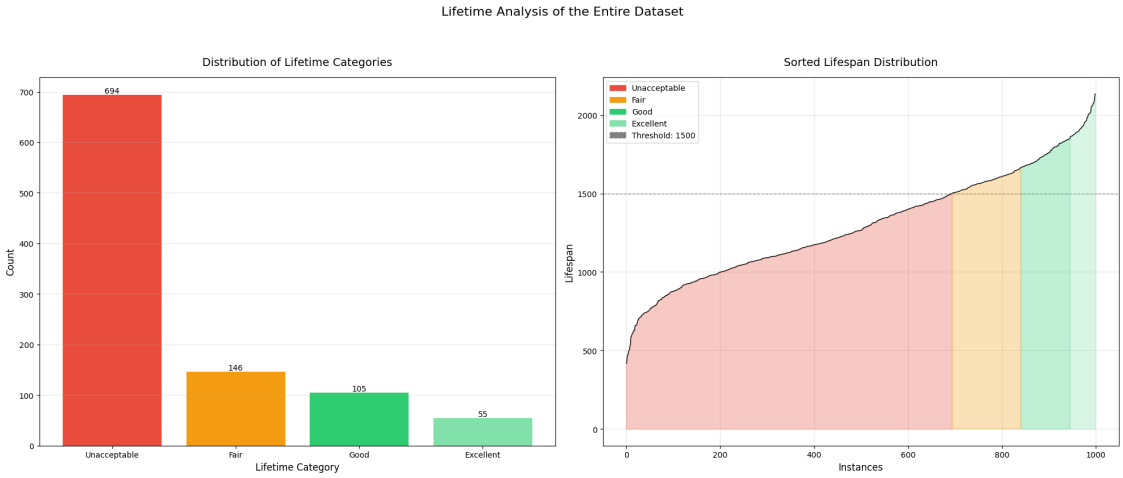
[180]:
```
unique_lifespan_count = clf_df['Lifespan'].nunique()
print(f"Number of unique values in 'Lifespan' column: {unique_lifespan_count}")

# Find the unique values that occur more than once in the 'Lifespan' column
print("\nDuplicate values:")
duplicate_values = clf_df['Lifespan'].value_counts()
duplicate_values = duplicate_values[duplicate_values > 1]

# Output the unique values and their counts
print(duplicate_values)
```

```
Number of unique values in 'Lifespan' column: 998

Duplicate values:
Lifespan
1262.14    2
932.69     2
Name: count, dtype: int64
```

[181]:
```
def lifetime_analysis(df, message='Entire Dataset'):
    # Set style
    plt.style.use('default')

    # Create figure with two subplots
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 8))
    fig.suptitle(f'Lifetime Analysis of the {message}', fontsize=16, y=1.05)
```

```
    # Color palette
    colors = {
        'Unacceptable': '#E74C3C',  # Red
        'Fair': '#F39C12',          # Orange
        'Good': '#2ECC71',          # Green
        'Excellent': '#82E0AA'      # Light Green
    }

    # Plot 1: Distribution
    counts = df['Lifetime'].value_counts().reindex(cluster_ranges['Lifetime'].
↪unique())
    bars = ax1.bar(range(len(counts)), counts, color=[colors[cat] for cat in␣
↪counts.index])

    # Customize first plot
    ax1.set_xticks(range(len(counts)))
    ax1.set_xticklabels(counts.index, rotation=0)
    ax1.set_xlabel('Lifetime Category', fontsize=12)
    ax1.set_ylabel('Count', fontsize=12)
    ax1.set_title('Distribution of Lifetime Categories', fontsize=14, pad=15)
    ax1.grid(True, axis='y', alpha=0.3)

    # Add count labels on bars
    for bar in bars:
        height = bar.get_height()
        ax1.text(bar.get_x() + bar.get_width()/2., height,
                f'{int(height)}',
                ha='center', va='bottom')

    # Plot 2: Sorted Lifespan
    df_sorted = df.sort_values('Lifespan').reset_index(drop=True)
    x = df_sorted.index
    y = df_sorted['Lifespan']
    categories = df_sorted['Lifetime']

    # Plot the main line
    ax2.plot(x, y, color='black', linewidth=1, zorder=3)

    # Fill areas by category
    for key, group in groupby(zip(x, y, categories), lambda t: t[2]):
        group = list(group)
        x_vals = [item[0] for item in group]
        y_vals = [item[1] for item in group]
        ax2.fill_between(x_vals, y_vals, color=colors[key], alpha=0.3, zorder=2)

    # Add threshold line
```

```
    ax2.axhline(y=1500, color='gray', linestyle='--', linewidth=1, zorder=1,
                label='Threshold: 1500')

    # Customize second plot
    ax2.set_xlabel('Instances', fontsize=12)
    ax2.set_ylabel('Lifespan', fontsize=12)
    ax2.set_title('Sorted Lifespan Distribution', fontsize=14, pad=15)
    ax2.grid(True, alpha=0.3)

    # Create legend
    handles = [mpatches.Patch(color=colors[cat], label=cat) for cat in colors.
 ↪keys()]
    handles.append(mpatches.Patch(color='gray', linestyle='--',␣
 ↪label='Threshold: 1500'))
    ax2.legend(handles=handles, loc='upper left')

    # Adjust layout
    plt.tight_layout()
    plt.show()

# Call the function
lifetime_analysis(clf_df)
```

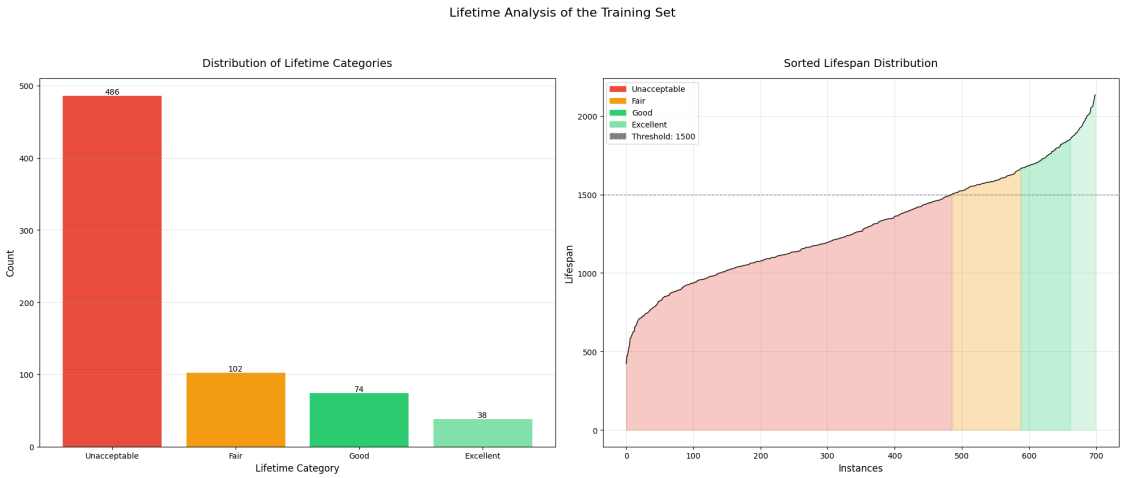


[182]:
```
# Bin 'Lifespan' into quartiles
clf_df['Lifespan_bin'] = pd.qcut(clf_df['Lifespan'], q=4, duplicates='drop')

# Create a combined stratification label
clf_df['Stratify_Label'] = clf_df['Lifetime'] + '_' + clf_df['Lifespan_bin'].
 ↪astype(str)
```

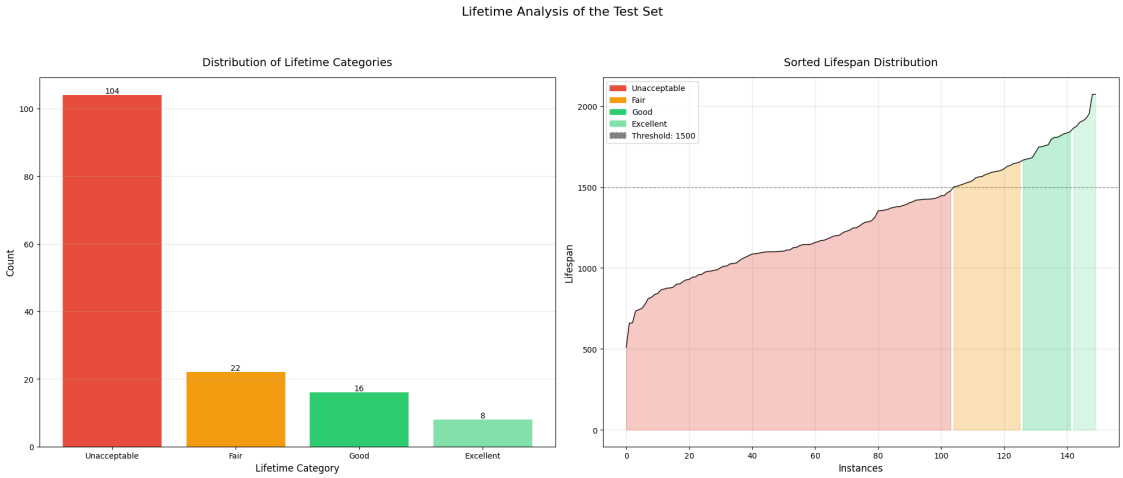
```
# First, split into training and temp (validation + test)
clf_train_df, temp_df = train_test_split(
    clf_df,
    test_size=0.3,
    stratify=clf_df['Stratify_Label'],
    random_state=random_state
)

# Then, split temp_df equally into validation and test sets
clf_val_df, clf_test_df = train_test_split(
    temp_df,
    test_size=0.5,
    stratify=temp_df['Stratify_Label'],
    random_state=random_state
)

# Remove the temporary DataFrame
del temp_df

# Optionally, drop the temporary columns
clf_train_df = clf_train_df.drop(columns=['Lifespan_bin', 'Stratify_Label'])
clf_val_df = clf_val_df.drop(columns=['Lifespan_bin', 'Stratify_Label'])
clf_test_df = clf_test_df.drop(columns=['Lifespan_bin', 'Stratify_Label'])

train_shape, val_shape, test_shape = clf_train_df.shape, clf_val_df.shape,␣
 ↪clf_test_df.shape
print(f"Training set shape   : {train_shape}")
print(f"Validation set shape : {val_shape}")
print(f"Test set shape       : {test_shape}")
```

```
Training set shape   : (700, 19)
Validation set shape : (150, 19)
Test set shape       : (150, 19)
```

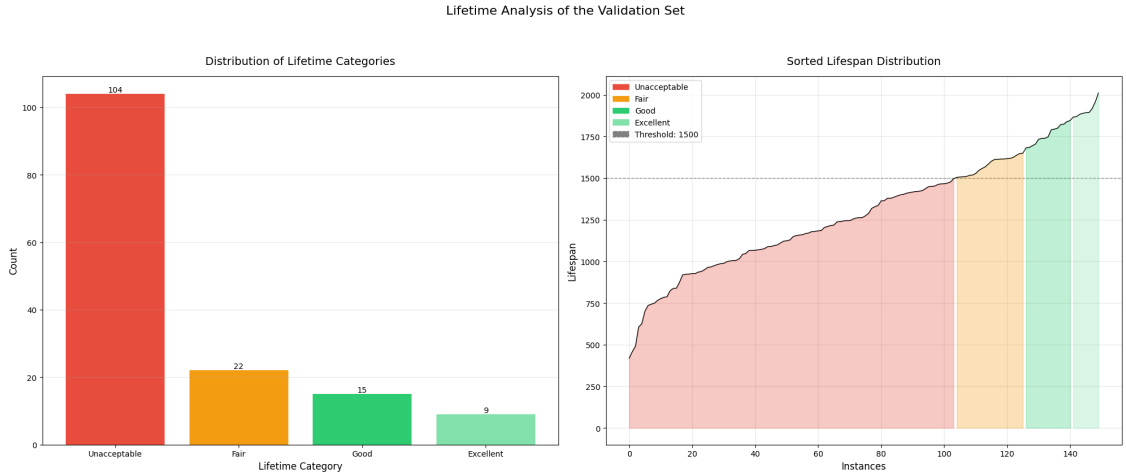
[183]:
```
lifetime_analysis(clf_train_df, message='Training Set')
```



[184]:
```
lifetime_analysis(clf_test_df, message='Test Set')
```



[185]:
```
lifetime_analysis(clf_val_df, message='Validation Set')
```



[186]:
```
clf_train_df = clf_train_df.drop(columns=['Lifespan'])
clf_val_df = clf_val_df.drop(columns=['Lifespan'])
clf_test_df = clf_test_df.drop(columns=['Lifespan'])

# Removing the 'Lifespan' column from the split sets.
train_shape, val_shape, test_shape = clf_train_df.shape, clf_val_df.shape,␣
 ↪clf_test_df.shape
print("After removing the 'Lifespan' column:")
print(f"Training set shape   : {train_shape}")
print(f"Validation set shape : {val_shape}")
print(f"Test set shape       : {test_shape}")
```

```
After removing the 'Lifespan' column:
Training set shape   : (700, 18)
Validation set shape : (150, 18)
Test set shape       : (150, 18)
```

[187]:
```
# Apply one-hot encoding to the training set and train the model
onehot_train_df = one_hot_encoding(clf_train_df, categorical_features)

target = 'Lifetime'

# Features (X) and target (y)
X_train_clf = onehot_train_df.drop(columns=[target])    # Features excluding the␣
 ↪target variable
y_train_clf = onehot_train_df[target]                   # Target variable

# Apply one-hot encoding to the test set and test the model
onehot_test_df = one_hot_encoding(clf_test_df, categorical_features)

# Features (X) and target (y)
```

```
X_test_clf = onehot_test_df.drop(columns=[target])      # Features excluding the␣
 ↪target variable
y_test_clf = onehot_test_df[target]                     # Target variable

# Apply one-hot encoding to the validation set and validate the model
onehot_val_df = one_hot_encoding(clf_val_df, categorical_features)

# Features (X) and target (y)
X_val_clf = onehot_val_df.drop(columns=[target])        # Features excluding the␣
 ↪target variable
y_val_clf = onehot_val_df[target]                       # Target variable

# Saving the dataset without polynomial features
nonpoly_columns = [col for col in X_train_clf.columns if col not in␣
 ↪polynomial_features]

X_train_clf_nonpoly = X_train_clf[nonpoly_columns]
X_test_clf_nonpoly = X_test_clf[nonpoly_columns]
X_val_clf_nonpoly = X_val_clf[nonpoly_columns]
```

[188]:
```
# Define the model
LogRegModel = LogisticRegression(max_iter=1000, random_state=random_state)
```

[189]:
```
X_train_scaled = scaler.fit_transform(X_train_clf)
X_test_scaled = scaler.transform(X_test_clf)

LogRegModel.fit(X_train_scaled, y_train_clf)
```

[189]:
```
LogisticRegression(max_iter=1000, random_state=42)
```

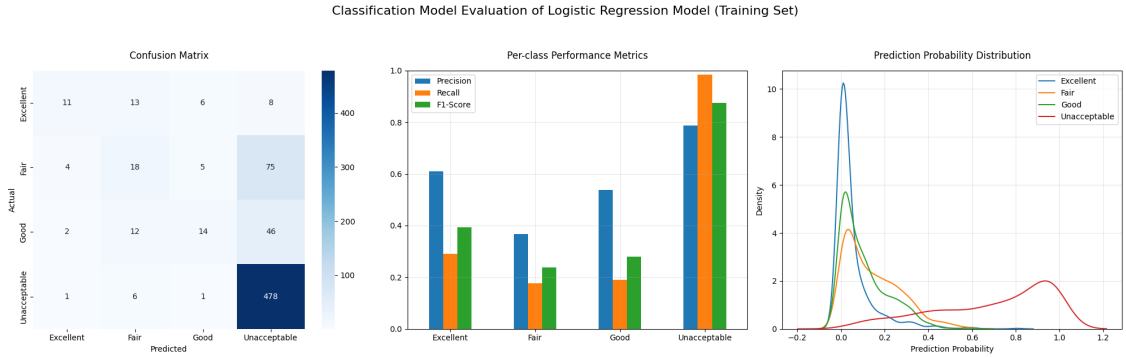
[190]:
```
def classification_evaluation(model, X, y, message="Model", scale=False):
    if scale:
        scaler = StandardScaler()
        X = pd.DataFrame(
            scaler.fit_transform(X),
            columns=X.columns,
            index=X.index
        )

    # Get predictions and probabilities
    if isinstance(model, tf.keras.Model):
        y_pred_probs = model.predict(X, verbose=0)
        y_pred = np.argmax(y_pred_probs, axis=1)
    else:
        y_pred = model.predict(X)
        y_pred_probs = model.predict_proba(X)
```

```
    # Calculate metrics
    accuracy = accuracy_score(y, y_pred)
    f1_weighted = f1_score(y, y_pred, average='weighted')

    # Calculate recall specifically for "Unacceptable" class
    unacceptable_recall = recall_score(y, y_pred,
                                       labels=['Unacceptable'],
                                       average=None)[0]

    # Calculate per-class precision
    precisions = precision_score(y, y_pred,
                                 labels=np.unique(y),
                                 average=None)

    # Print metrics
    title = f"--- {message} Evaluation Metrics ---"
    print(f"{title}")
    print(f"-"*len(title))
    print(f"    Accuracy             : {accuracy:.2f}")
    print(f"    Weighted F1-Score    : {f1_weighted:.2f}")
    print(f"    Unacceptable Recall  : {unacceptable_recall:.2f}")
    print("\n    Per-class Precision:")
    for cls, prec in zip(np.unique(y), precisions):
        print(f"        {cls}: {prec:.2f}")
    print(f"-"*len(title))

    # Plotting
    plt.style.use('default')
    fig = plt.figure(figsize=(20, 6))
    fig.suptitle(f'Classification Model Evaluation of {message}', fontsize=16,␣
↪y=1.05)
    gs = fig.add_gridspec(1, 3)

    # Plot 1: Confusion Matrix
    ax1 = fig.add_subplot(gs[0, 0])
    cm = confusion_matrix(y, y_pred)
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=np.unique(y),
                yticklabels=np.unique(y),
                ax=ax1)
    ax1.set_title('Confusion Matrix', pad=15)
    ax1.set_xlabel('Predicted')
    ax1.set_ylabel('Actual')

    # Plot 2: Per-class Performance
    ax2 = fig.add_subplot(gs[0, 1])
    class_metrics = pd.DataFrame({
```

```
        'Precision': precision_score(y, y_pred, average=None),
        'Recall': recall_score(y, y_pred, average=None),
        'F1-Score': f1_score(y, y_pred, average=None)
    }, index=np.unique(y))

    class_metrics.plot(kind='bar', ax=ax2)
    ax2.set_title('Per-class Performance Metrics', pad=15)
    ax2.set_ylim(0, 1)
    ax2.grid(True, alpha=0.3)
    plt.setp(ax2.xaxis.get_majorticklabels(), rotation=0)

    # Plot 3: Prediction Probabilities Distribution
    ax3 = fig.add_subplot(gs[0, 2])

    # Get probabilities for each class
    for i, class_name in enumerate(np.unique(y)):
        class_probs = y_pred_probs[:, i]
        sns.kdeplot(data=class_probs, label=class_name, ax=ax3)

    ax3.set_title('Prediction Probability Distribution', pad=15)
    ax3.set_xlabel('Prediction Probability')
    ax3.set_ylabel('Density')
    ax3.grid(True, alpha=0.3)
    ax3.legend()

    plt.tight_layout()
    plt.show()

    # Memory cleanup
    del X, y, y_pred, y_pred_probs
```

[191]:
```
classification_evaluation(LogRegModel, X_train_scaled, y_train_clf, "Logistic␣
↪Regression Model (Training Set)")
```

```
--- Logistic Regression Model (Training Set) Evaluation Metrics ---
----------------------------------------------------------------------
    Accuracy             : 0.74
    Weighted F1-Score    : 0.69
    Unacceptable Recall  : 0.98

    Per-class Precision:
        Excellent: 0.61
        Fair: 0.37
        Good: 0.54
        Unacceptable: 0.79
----------------------------------------------------------------------
```



[ ]:
```
# Preprocessing for numeric features
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

# Preprocessing for categorical features
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
    ('onehot', ohe)
])

# Combine preprocessing steps
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numerical_features),
        ('cat', categorical_transformer, categorical_features)
    ])

# Separate features and target
X_train = clf_train_df.drop(columns=['Lifetime'])
y_train = clf_train_df['Lifetime']

# Fit and transform the training data
X_train_processed = preprocessor.fit_transform(X_train)

# Encode the target labels using LabelEncoder
y_train_encoded = le.fit_transform(y_train)

# Compute class weights to handle class imbalance
class_weights = class_weight.compute_class_weight(
    class_weight='balanced',
    classes=np.unique(y_train_encoded),
    y=y_train_encoded
```

```
)
class_weights_dict = dict(enumerate(class_weights))

# Build the Neural Network model (Warning Fixed)
model = models.Sequential([
    layers.Input(shape=(X_train_processed.shape[1],)),
    layers.Dense(128, activation='relu'),
    layers.BatchNormalization(),
    layers.Dropout(0.5),
    layers.Dense(64, activation='relu'),
    layers.BatchNormalization(),
    layers.Dropout(0.5),
    layers.Dense(len(np.unique(y_train_encoded)), activation='softmax')
])

# Compile the model with appropriate loss function and optimizer
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

# Set up early stopping to prevent overfitting
early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=5,
    restore_best_weights=True
)

# Train the model
history = model.fit(
    X_train_processed,
    y_train_encoded,
    epochs=100,
    batch_size=32,
    class_weight=class_weights_dict,
    validation_split=0.2,
    callbacks=[early_stopping],
    verbose=1
)

# Evaluate the model on the training data
train_loss, train_accuracy = model.evaluate(X_train_processed, y_train_encoded,␣
 ↪verbose=0)
print(f"Neural Network training accuracy: {train_accuracy:.2f}")
```