DS203 : Exercise 3

Nirav Bhattad

Last updated September 3, 2024

Step 1

Step 1

Review the Jupyter Notebook `E3.ipynb` and:
(a) Create a summary of the code therein.
(b) Are there any learnings from this code that you wish to highlight?

This Python Notebook `E3.ipynb` presents an analysis of various regression models applied to a dataset using polynomial features of different degrees. The models analyzed include Linear Regression, Support Vector Machine (SVM) Regression, Random Forest, Gradient Boosting, K-Nearest Neighbors (KNN), and Neural Networks. The performance of each model is evaluated based on metrics such as R-squared, Mean Squared Error (MSE), Durbin-Watson, and Jarque-Bera statistics.

We use the dataset given in `E3-MLR3.xlsx`. The dataset contains 548 samples of (y, x_1) to train the model and 152 samples of (y, x_1) to test the model.

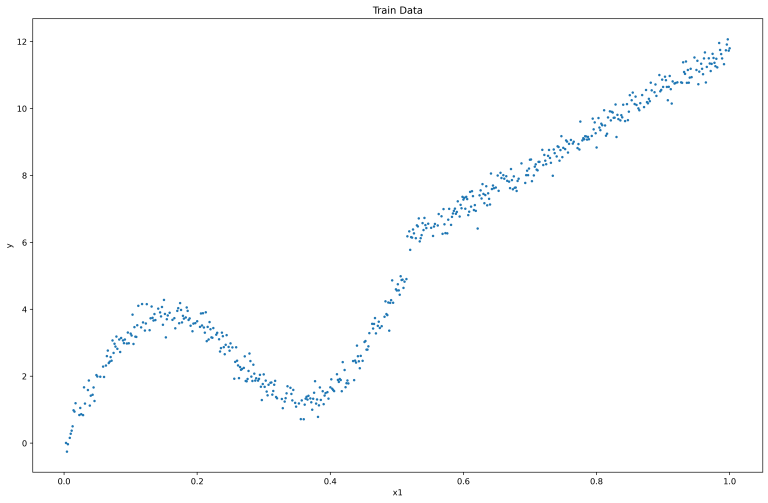

Figure 1: Training Data used in the analysis

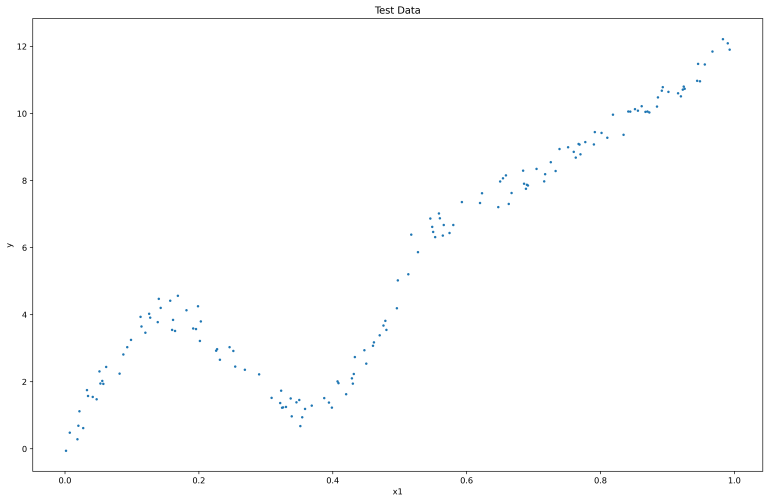

Figure 2: Testing Data used in the analysis

The following Python script in `E3.ipynb` applies multiple regression algorithms to a dataset, evaluates their performance, and visualizes the results.

1. Importing Necessary Libraries

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.svm import SVR
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.neural_network import MLPRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error, r2_score
import statsmodels.api as sm
import matplotlib.pyplot as plt
```

Listing 1: Importing Libraries

Description: The code begins by importing essential libraries for data manipulation (`pandas`, `numpy`), machine learning models (`scikit-learn`), statistical analysis (`statsmodels`), and plotting (`matplotlib`).

2. Data Loading and Preprocessing

```
file_path = 'E3-MLR3.xlsx'
train_data = pd.read_excel(file_path, sheet_name='train')
test_data = pd.read_excel(file_path, sheet_name='test')

X_train = train_data.drop(columns=['y'])
y_train = train_data['y']

X_test = test_data.drop(columns=['y'])
y_test = test_data['y']

poly = PolynomialFeatures(degree=6)
X_train_poly = poly.fit_transform(X_train)
```

```
X_test_poly = poly.transform(X_test)
```

Listing 2: Loading and Preprocessing Data

Description: The script reads training and testing data from an Excel file and separates features (`X_train`, `X_test`) and the target variable (`y_train`, `y_test`). It then uses `PolynomialFeatures` to augment the features with polynomial terms of degree 6.

3. Saving the Augmented Dataset

```
feature_names = poly.get_feature_names_out(X_train.columns)
augmented_data = pd.DataFrame(X_test_poly, columns=feature_names)
augmented_data['y'] = test_data['y']
augmented_data.to_csv('augmented_data.csv', index=False)
```

Listing 3: Saving Augmented Data

Description: The code generates a CSV file containing the test data with the newly created polynomial features for review.

4. Defining and Running Regression Models

```
algorithms = {
    'Linear Regression': LinearRegression(),
    'SVM Regression': SVR(kernel='poly'),
    'RandomForest': RandomForestRegressor(),
    'XGBoost': GradientBoostingRegressor(),
    'knn': KNeighborsRegressor(),
    'Neural Network': MLPRegressor(hidden_layer_sizes=[10,10,10], max_iter=20000)
}

metric_table_train = pd.DataFrame()
metric_table_test = pd.DataFrame()

fig, axs = plt.subplots(len(algorithms), 4, figsize=(20, 20))
fig_row = -1

for algorithm_name, algorithm in algorithms.items():
    algorithm.fit(X_train_poly, y_train)
    y_train_pred = algorithm.predict(X_train_poly)
    y_test_pred = algorithm.predict(X_test_poly)

    r2_train = algorithm.score(X_train_poly, y_train)
    mse_train = mean_squared_error(y_train, y_train_pred)
    r2_test = algorithm.score(X_test_poly, y_test)
    mse_test = mean_squared_error(y_test, y_test_pred)

    residuals_train = y_train - y_train_pred
    residuals_test = y_test - y_test_pred

    durbin_watson_stat_train = sm.stats.durbin_watson(residuals_train)
    jb_stat_train, jb_p_value_train, _, _ = sm.stats.jarque_bera(residuals_train)

    durbin_watson_stat_test = sm.stats.durbin_watson(residuals_test)
    jb_stat_test, jb_p_value_test, _, _ = sm.stats.jarque_bera(residuals_test)

    metric_table_train.at[algorithm_name, 'R-squared'] = r2_train
    metric_table_train.at[algorithm_name, 'MSE'] = mse_train
    metric_table_train.at[algorithm_name, 'Durbin-Watson'] = durbin_watson_stat_train
    metric_table_train.at[algorithm_name, 'Jarque-Bera'] = jb_stat_train
    metric_table_train.at[algorithm_name, 'JB P-value'] = jb_p_value_train

    metric_table_test.at[algorithm_name, 'R-squared'] = r2_test
    metric_table_test.at[algorithm_name, 'MSE'] = mse_test
    metric_table_test.at[algorithm_name, 'Durbin-Watson'] = durbin_watson_stat_test
```

```
    metric_table_test.at[algorithm_name, 'Jarque-Bera'] = jb_stat_test
    metric_table_test.at[algorithm_name, 'JB P-value'] = jb_p_value_test

    fig_row = fig_row+1

    axs[fig_row, 0].scatter(train_data['x1'], y_train)
    axs[fig_row, 0].scatter(train_data['x1'], y_train_pred)
    axs[fig_row, 0].set_title(algorithm_name + " - Train")

    axs[fig_row, 1].scatter(train_data['x1'], residuals_train)
    axs[fig_row, 1].set_title(algorithm_name + " Residuals - Train")

    axs[fig_row, 2].scatter(test_data['x1'], y_test)
    axs[fig_row, 2].scatter(test_data['x1'], y_test_pred)
    axs[fig_row, 2].set_title(algorithm_name + " - Test")

    axs[fig_row, 3].scatter(test_data['x1'], residuals_test)
    axs[fig_row, 3].set_title(algorithm_name + " Residuals - Test")
```

Listing 4: Defining and Running Regression Models

Description: A dictionary of regression algorithms is created. Each algorithm is trained on the polynomial features, and its performance is evaluated using metrics such as R-squared, Mean Squared Error (MSE), Durbin-Watson, and Jarque-Bera test. These metrics are stored in tables. The residuals and predicted values are also plotted.

5. Visualizing the Results

```
plt.tight_layout()
plt.show()
```

Listing 5: Visualizing the Results

Description: The code concludes with displaying the generated plots for each regression model.

6. Plotting Error Metrics

```
print("Metrics - Train Data:\n")
print(metric_table_train.to_string())
print("-----------------------------------------------")

print("Metrics - Test Data:\n")
print(metric_table_test.to_string())
print("-----------------------------------------------")
```

Listing 6: Plotting Error Metrics

Description: The code prints the error metrics for each regression model on both training and testing datasets.

Learnings from this Code

(a) The code demonstrates the application of various regression algorithms to a dataset with polynomial features.

(b) It highlights the importance of evaluating regression models using multiple metrics to understand their performance.

(c) The code showcases the use of residual analysis to assess the model's assumptions and identify potential issues.

(d) It provides a structured approach to comparing different regression models and their outcomes.

(e) The code emphasizes the significance of visualizing the model's predictions and residuals for better interpretation.

(f) The code also shows how to save augmented datasets for further analysis or review.

Step 2

Step 2

Review the **Sklearn** documentation for each Sklearn function used in the Notebook (eg. `PolynomialFeatures`, `LinearRegression`, `mean_squared_error`, etc.) and create a description of each to explain, to yourself, the functionality, the input parameters, and the outputs generated. Present this in the form of a two-column - Table (Function name | Description).

`PolynomialFeatures`	This function is used to generate polynomial and interaction features. It generates a new feature matrix consisting of all polynomial combinations of the features with a degree less than or equal to the specified degree. The input to this function is the degree of the polynomial features to be generated. The output generated is a new feature matrix consisting of all polynomial combinations of the features with a degree less than or equal to the specified degree. Input Parameters: • **degree**: `int` or `tuple (min_degree, max_degree)`, `default=2` • **interaction_only**: `bool`, `default=False` • **include_bias**: `bool`, `default=True` • **order**: `str` in {'C', 'F'}, `default='C'` Attributes: • **powers_**: `ndarray` of shape (`n_output_features_`, `n_input_features_`) • **n_output_features_**: `int` • **n_features_in_**: `int` • **feature_names_in_**: `ndarray` of shape (`n_input_features_`,) Output: • **ndarray** of shape (`n_samples`, `n_output_features_`)
`LinearRegression`	This function is used to fit a linear model. It fits a linear model with coefficients $w = (w_1, ..., w_p)$ to minimize the residual sum of squares between the observed targets in the dataset and the targets predicted by the linear approximation. Input Parameters: • **fit_intercept**: `bool`, `default=True` • **copy_X**: `bool`, `default=True` • **n_jobs**: `int`, `default=None` • **positive**: `bool`, `default=False` Attributes: • **coef_**: `ndarray` of shape (`n_targets`, `n_features`) • **intercept_**: `ndarray` of shape (`n_targets`,) • **rank_**: `int` • **singular_**: `ndarray` of shape (`min(X, y)`,) • **n_features_in_**: `ndarray` of shape (`n_targets`,) Output: • **self**: returns an instance of self

`SVR`	This function is used to fit the Support Vector Regression model. Input Parameters: • **kernel**: `str`, `default='rbf'` • **degree**: `int`, `default=3` • **gamma**: `float`, `default='scale'` • **coef0**: `float`, `default=0.0` • **tol**: `float`, `default=1e-3` • **C**: `float`, `default=1.0` • **epsilon**: `float`, `default=0.1` • **shrinking**: `bool`, `default=True` • **cache_size**: `float`, `default=200` • **verbose**: `bool`, `default=False` • **max_iter**: `int`, `default=-1` Attributes: • **coef_**: `ndarray` of shape (1, `n_features`) • **dual_coef_**: `ndarray` of shape (1, `n_SV`) • **fit_status_**: `int` • **intercept_**: `ndarray` of shape (1,) • **n_features_in_**: `int` • **feature_names_in_**: `ndarray` of shape (`n_features`,) • **n_iter_**: `int` • **n_support_**: `ndarray` of shape (1,) • **shape_fit_**: `tuple` of `int` of shape (`n_dimensions_of_X`,) • **support_**: `ndarray` of shape (`n_SV`,) • **support_vectors_**: `ndarray` of shape (`n_SV`, `n_features`) Output: • **ndarray** of shape (`n_samples`,)
`KNeighborsRegressor`	This function is used to fit the K-Nearest Neighbors regressor model, which is a non-parametric method used for regression. It predicts the target variable by learning from the k nearest data points in the feature space. Input Parameters: • **n_neighbors**: `int`, `default=5` • **weights**: `str`, `default='uniform'` • **algorithm**: `str`, `default='auto'` • **leaf_size**: `int`, `default=30` • **p**: `int`, `default=2` • **metric**: `str`, `default='minkowski'` • **metric_params**: `dict`, `default=None` • **n_jobs**: `int`, `default=None` Attributes: • **effective_metric_**: `str` • **effective_metric_params_**: `dict` • **n_samples_fit_**: `int` • **n_features_in_**: `int` • **feature_names_in_**: `ndarray` of shape (`n_features`,) Output: • **ndarray** of shape (`n_samples`,)

`RandomForestRegressor`	This function is used to fit a random forest model, which is an ensemble learning method for regression. It fits a number of classifying decision trees on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. Input Parameters: • **n_estimators**: `int`, `default=100` • **criterion**: `str`, `default='squared_error'` • **max_depth**: `int`, `default=None` • **min_samples_split**: `int`, `default=2` • **min_samples_leaf**: `int`, `default=1` • **min_weight_fraction_leaf**: `float`, `default=0.0` • **max_features**: `int`, `default='auto'` • **max_leaf_nodes**: `int`, `default=None` • **min_impurity_decrease**: `float`, `default=0.0` • **bootstrap**: `bool`, `default=True` • **oob_score**: `bool`, `default=False` • **n_jobs**: `int`, `default=None` • **random_state**: `int`, `default=None` • **verbose**: `int`, `default=0` • **warm_start**: `bool`, `default=False` • **ccp_alpha**: `float`, `default=0.0` • **max_samples**: `int`, `default=None` • **monotonic_cst**: `array-like` of shape (`n_features`), `default=None` Attributes: • **estimator_**: `DecisionTreeRegressor` • **estimators_**: `list of DecisionTreeRegressor` • **n_features_in**: `int` • **feature_names_in_**: `ndarray` of shape (`n_features`,) • **n_outputs_**: `int` • **oob_score_**: `ndarray` of shape (`n_estimators`,) • **oob_prediction_**: `ndarray` of shape (`n_samples`, `n_outputs`) • **estimators_samples_**: `list of ndarray` Output: • **ndarray** of shape (`n_samples`,)
`mean_squared_error`	This function is used to compute the mean squared error regression loss. It computes the mean squared error between the true and predicted values. Input Parameters: • **y_true**: `ndarray` of shape (`n_samples`,) • **y_pred**: `ndarray` of shape (`n_samples`,) • **squared**: `bool`, `default=True` • **multioutput**: `str`, `default='uniform_average'` • **sample_weight**: `ndarray` of shape (`n_samples`,), `default=None` Output: • **float**

GradientBoostingRegressor	This function is used to fit a gradient boosting model, which is an ensemble learning method for regression. It fits a number of regression trees on various sub-samples of the dataset and uses boosting to improve the predictive accuracy and control over-fitting. Input Parameters: • **loss**: `str`, `default='ls'` • **learning_rate**: `float`, `default=0.1` • **n_estimators**: `int`, `default=100` • **subsample**: `float`, `default=1.0` • **criterion**: `str`, `default='friedman_mse'` • **min_samples_split**: `int`, `default=2` • **min_samples_leaf**: `int`, `default=1` • **min_weight_fraction_leaf**: `float`, `default=0.0` • **max_depth**: `int`, `default=3` • **min_impurity_decrease**: `float`, `default=0.0` • **init**: `estimator`, `default=None` • **random_state**: `int`, `default=None` • **max_features**: `str`, `default=None` • **alpha**: `float`, `default=0.9` • **verbose**: `int`, `default=0` • **max_leaf_nodes**: `int`, `default=None` • **warm_start**: `bool`, `default=False` • **validation_fraction**: `float`, `default=0.1` • **n_iter_no_change**: `int`, `default=None` • **tol**: `float`, `default=1e-4` • **ccp_alpha**: `float`, `default=0.0` Attributes: • **n_estimators_**: `int` • **n_trees_per_iteration_**: `int` • **feature_importances_**: `ndarray` of shape (`n_features`,) • **oob_improvement_**: `ndarray` of shape (`n_estimators`,) • **oob_score_**: `ndarray` of shape (`n_estimators`,) • **oob_scores_**: `ndarray` of shape (`n_estimators`,) • **train_score_**: `ndarray` of shape (`n_estimators`,) • **init_**: `estimator` • **estimators_**: `ndarray of DecisionTreeRegressor` • **n_features_in_**: `int` • **feature_names_in_**: `ndarray` of shape (`n_features`,) • **max_features_**: `int` Output: • **ndarray** of shape (`n_samples`,)
`r2_score`	This function is used to compute the R-squared regression score function. This function has not been used in the jupyter notebook. Input Parameters: • **y_true**: `ndarray` of shape (`n_samples`,) • **y_pred**: `ndarray` of shape (`n_samples`,) • **sample_weight**: `ndarray` of shape (`n_samples`,), `default=None` • **multioutput**: `str`, `default='uniform_average'` • **force_finite**: `bool`, `default=True` Output: • **float**

`MLPRegressor`	This function is used to fit a Multi-layer Perceptron regressor. It trains a neural network model with a single hidden layer, which is a fully-connected feed-forward artificial neural network. Input Parameters: • **hidden_layer_sizes**: `tuple`, `default=(100,)` • **activation**: `str`, `default='relu'` • **solver**: `str`, `default='adam'` • **alpha**: `float`, `default=0.0001` • **batch_size**: `int`, `default='auto'` • **learning_rate**: `str`, `default='constant'` • **learning_rate_init**: `float`, `default=0.001` • **power_t**: `float`, `default=0.5` • **max_iter**: `int`, `default=200` • **shuffle**: `bool`, `default=True` • **random_state**: `int`, `default=None` • **tol**: `float`, `default=1e-4` • **verbose**: `bool`, `default=False` • **warm_start**: `bool`, `default=False` • **momentum**: `float`, `default=0.9` • **nesterovs_momentum**: `bool`, `default=True` • **early_stopping**: `bool`, `default=False` • **validation_fraction**: `float`, `default=0.1` • **beta_1**: `float`, `default=0.9` • **beta_2**: `float`, `default=0.999` • **epsilon**: `float`, `default=1e-8` • **n_iter_no_change**: `int`, `default=10` • **max_fun**: `int`, `default=15000` Attributes: • **loss_**: `float` • **best_loss_**: `float` • **loss_curve_**: `list` of shape (`n_iter_`,) • **t_**: `int` • **validation_scores_**: `list` of shape (`n_iter_`,) • **best_validation_score_**: `float` • **coefs_**: `list` of shape (`n_layers_ - 1`,) • **intercepts_**: `list` of shape (`n_layers_ - 1`,) • **n_features_in_**: `int` • **feature_names_in_**: `ndarray` of shape (`n_features`,) • **n_iter_**: `int` • **n_layers_**: `int` • **n_outputs_**: `int` • **out_activation_**: `str` Output: • **ndarray** of shape (`n_samples`,)

Step 3

Step 3

Generate outputs by setting **degree = 1**, **degree = 3**, **degree = 6**, **degree = 10**, in the `PolynomialFeatures` function used in `E3.ipynb` and analyze them as follows:

(a) Review the `augmented_data.csv` file generated in each case and document your observations.
(b) Create an overall qualitative summary based on a review and analysis of the Figures generated.
(c) Summarize and explain the variations in the metrics **across regression methods for a given degree** (ie. a given set of polynomial features). Cover both, train and test, metrics, and compare them.
(d) Summarize and explain the variations in the metrics **across degrees for a given regression method**. Cover both, train, and test metrics, and compare them.
(e) When **degree = 1** which method(s) result in acceptable regression models? Why?
(f) When **degree = 6** which method(s) result in acceptable regression models? Why?
(g) As the value of degree is increased to 10 which regression methods show the most impact? Why?
(h) Why do non-parametric methods like `KNN` and `Decision Tree` based methods generate good results even without feature engineering?
(i) What are the limitations of the non-parametric methods?
(j) Given the results, should LinearRegression be used at all? Why, when? Justify your answer.

Question (a)

Review the `augmented_data.csv` file generated in each case and document your observations.

The following observations are based on the content of the `augmented_data.csv` files generated for different polynomial degrees:

- **Degree = 1:**
  - **File Content:** The `augmented_data.csv` file contains only the original feature and the target variable y. With degree = 1, no additional polynomial features are generated beyond the original feature.
  - **Observation:** The file contains columns for the original feature(s) plus the target variable y. There are no new features created.
- **Degree = 3:**
  - **File Content:** The file includes the original feature x_1 along with polynomial terms up to the third degree: 1, x_1, x_1^2, and x_1^3, plus the target variable y.
  - **Observation:** The file contains columns for the original feature, its squared term, its cubic term, and the target variable y. There are four columns in total.
- **Degree = 6:**
  - **File Content:** This file contains the original feature x_1 and polynomial terms up to the sixth degree: 1, x_1, x_1^2, x_1^3, x_1^4, x_1^5, x_1^6, plus the target variable y.
  - **Observation:** The file includes the original feature, its polynomial terms up to the sixth degree, and the target variable y. There are eight columns in total.
- **Degree = 10:**
  - **File Content:** For degree = 10, the file contains the original feature x_1 and polynomial terms up to the tenth degree: 1, x_1, x_1^2, x_1^3, x_1^4, x_1^5, x_1^6, x_1^7, x_1^8, x_1^9, x_1^{10}, plus the target variable y.
  - **Observation:** The file has columns for the original feature and its polynomial terms up to the tenth degree, along with the target variable y. There are twelve columns in total.

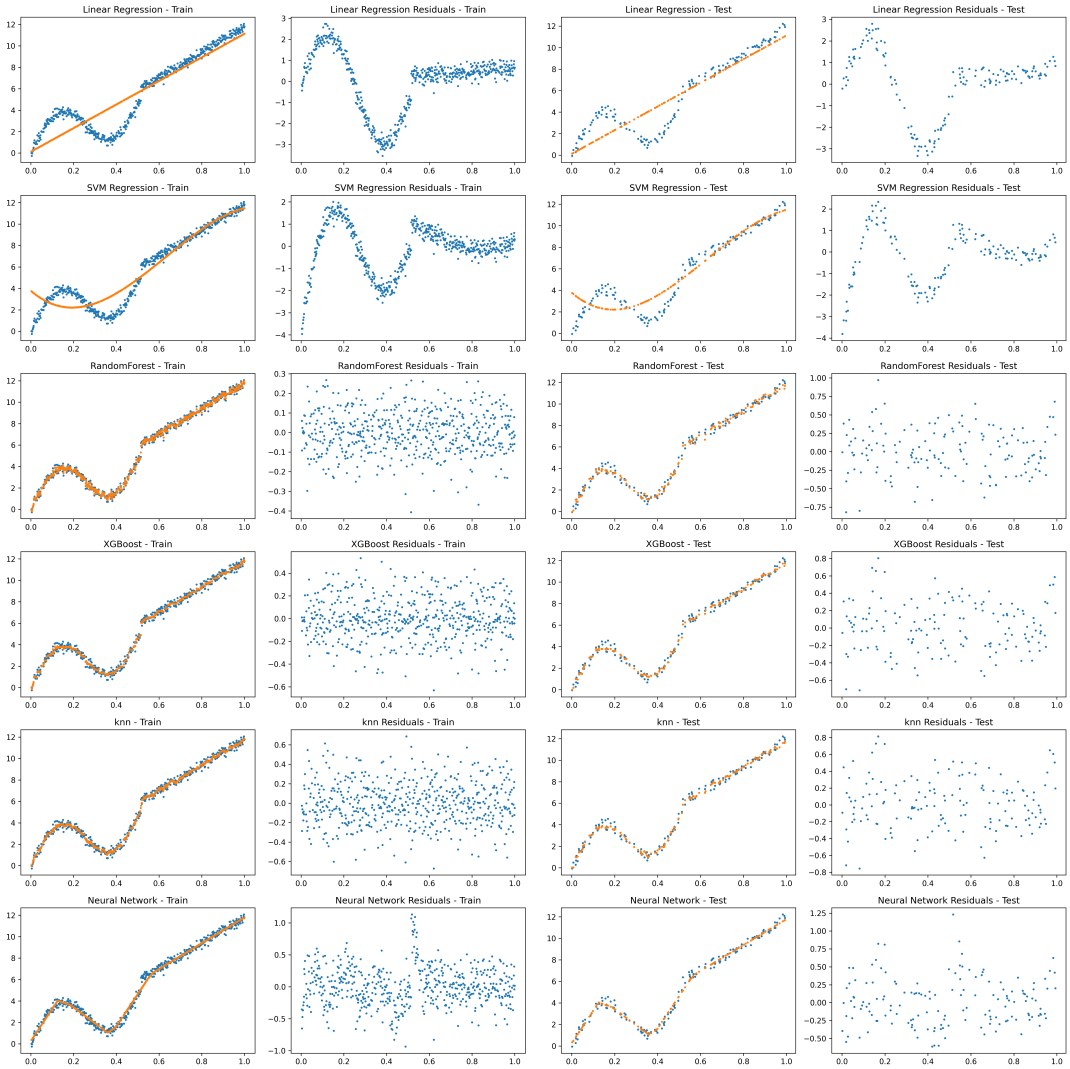

Figure 3: Various Regression Models with degree = 1

When degree = 1, the following observations can be made:

- **Linear Regression:** The plot reveals significant underfitting, with the linear fit failing to capture the non-linear trends in the data.
- **SVM (Polynomial Kernel):** The SVM plot shows similar underfitting as Linear Regression, with the linear decision boundary not aligning well with the non-linear data distribution.
- **RandomForest:** The plot demonstrates better fit compared to linear models, with RandomForest capturing some non-linear patterns despite the linear feature expansion.
- **XGBoost:** XGBoost effectively models the non-linear relationships, as seen in the plot with predictions aligning more closely with the true data distribution.
- **K-Nearest Neighbors (KNN):** The KNN plot indicates a good fit to the non-linear data, showing accurate predictions based on proximity without needing polynomial features.
- **Neural Network:** The plot for the Neural Network shows a strong fit to the non-linear data, capturing complex patterns despite the linear feature input.

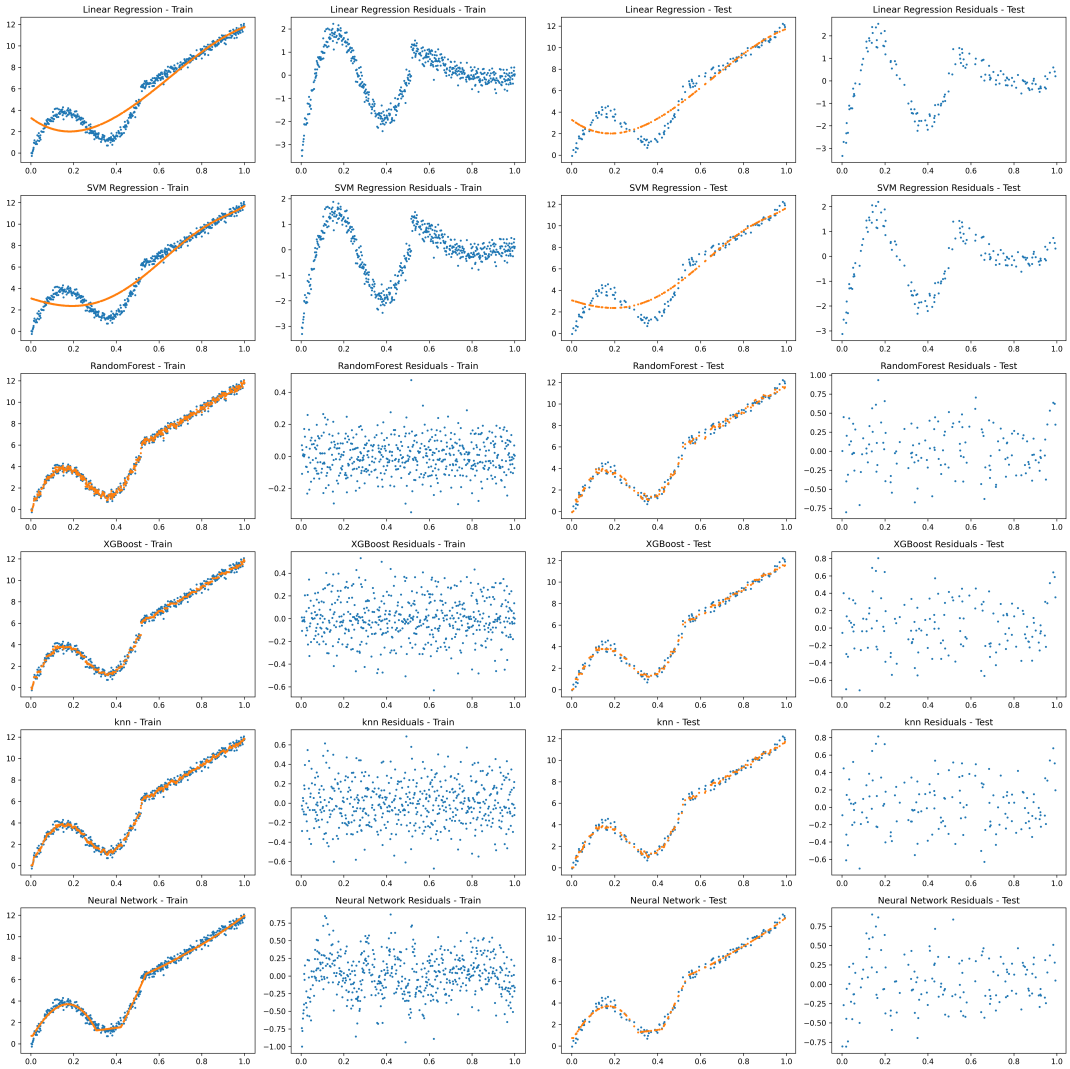

Figure 4: Various Regression Models with degree = 3

- **Linear Regression:** The plot shows improved fit over degree = 1 but still exhibits signs of underfitting, as the linear model struggles to capture the higher-order non-linear patterns.
- **SVM (Polynomial Kernel):** The linear SVM plot still underfits, with the decision boundary unable to fully align with the more complex, non-linear data distribution.
- **RandomForest:** The plot indicates a good fit, capturing more of the non-linear structure compared to degree = 1, thanks to its ability to model complex interactions.
- **XGBoost:** The XGBoost plot shows a strong fit to the non-linear data, effectively utilizing the higher-order polynomial features to model intricate patterns.
- **K-Nearest Neighbors (KNN):** The KNN plot reflects an excellent fit, with predictions closely following the non-linear trends in the data, demonstrating its strength with complex features.
- **Neural Network:** The Neural Network plot exhibits a very accurate fit, capturing complex non-linear relationships effectively with the expanded polynomial features.

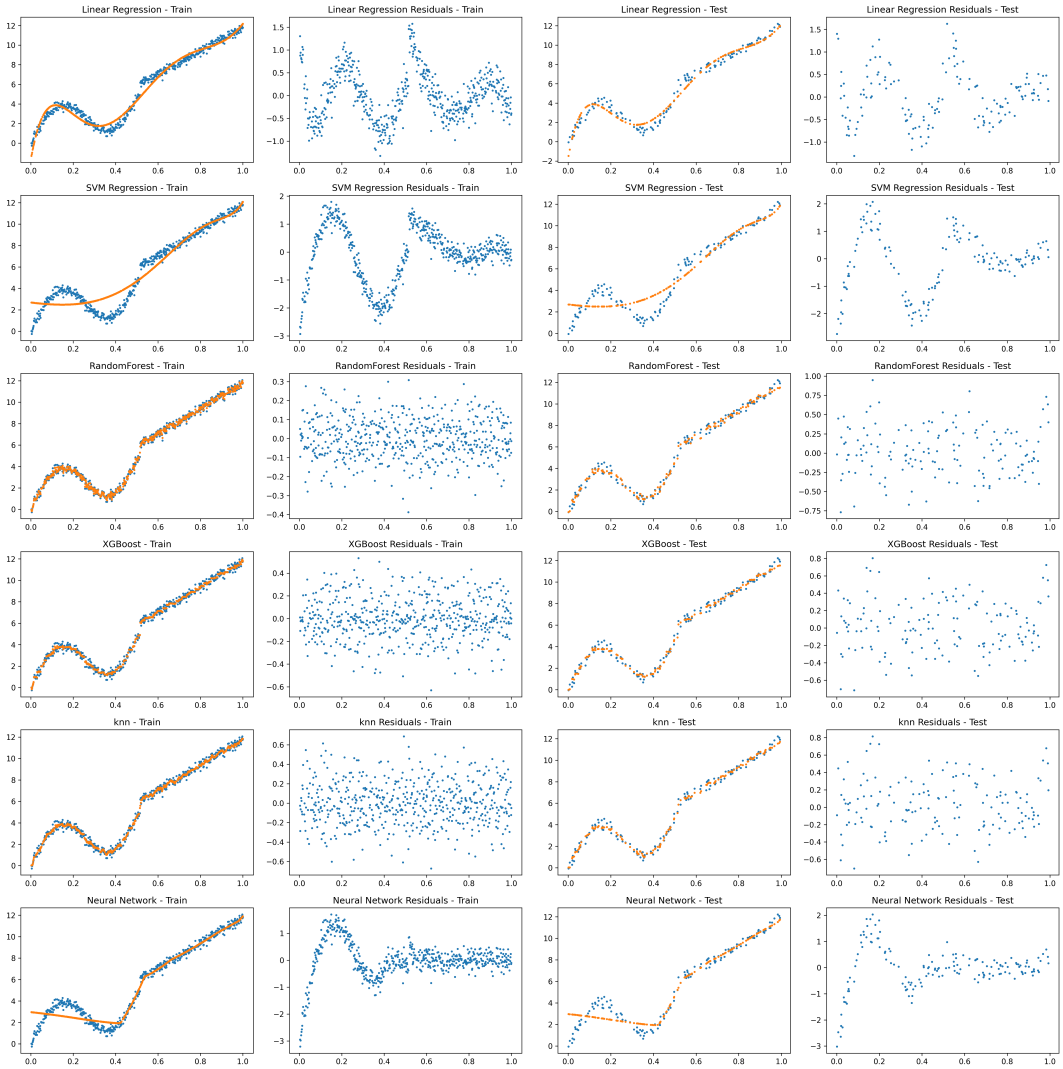

Figure 5: Various Regression Models with degree = 6

- **Linear Regression:** The plot shows significant overfitting, with the linear model failing to generalize and fitting the noise in the data rather than the true underlying pattern.
- **SVM (Polynomial Kernel):** The linear SVM plot still exhibits overfitting, with the decision boundary overly complex and not effectively capturing the true non-linear structure of the data.
- **RandomForest:** The plot demonstrates excellent fit to the data, capturing the intricate non-linear patterns, though it may start to show signs of overfitting with the increased feature complexity.
- **XGBoost:** The XGBoost plot reveals strong performance, modeling complex non-linear relationships well, but may also show signs of overfitting as the model captures noise along with the signal.
- **K-Nearest Neighbors (KNN):** The KNN plot shows a good fit, accurately following the non-linear data trends; however, it may become sensitive to noise with the increased feature space.
- **Neural Network:** The Neural Network plot indicates an excellent fit to the data, effectively capturing complex non-linear patterns with the expanded feature set, though it may also exhibit overfitting.

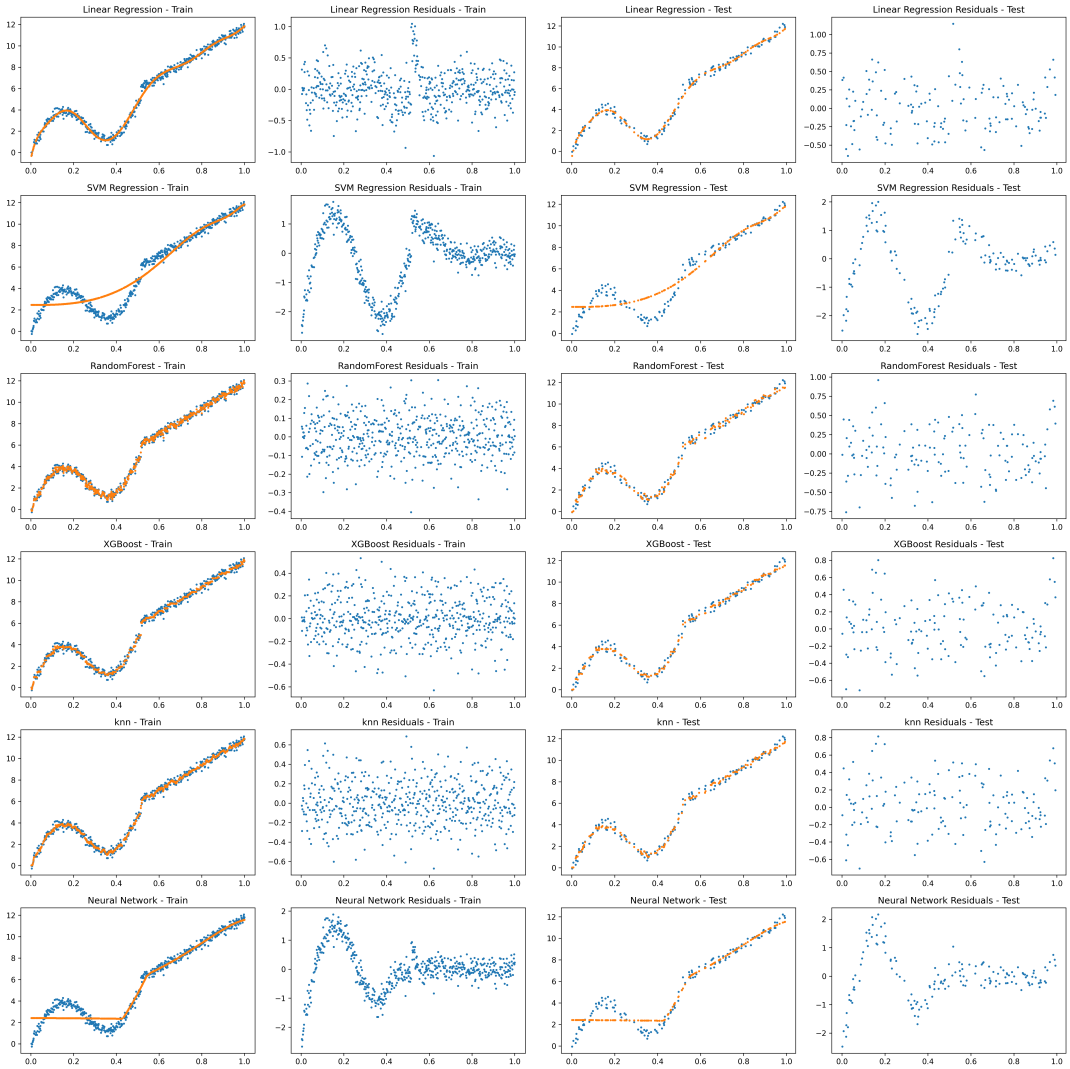

Figure 6: Various Regression Models with degree = 10

- **Linear Regression:** The plot shows severe overfitting, with the linear model failing to generalize and fitting the data's noise, resulting in a very erratic fit that does not align well with the true underlying patterns.
- **SVM (Polynomial Kernel):** The linear SVM plot also displays significant overfitting, with the decision boundary becoming overly complex and not effectively capturing the non-linear data structure.
- **RandomForest:** The plot demonstrates a very good fit to the complex data, but signs of overfitting are evident, as the model captures both the noise and the intricate non-linear patterns.
- **XGBoost:** The XGBoost plot shows exceptional fit, accurately capturing the complex non-linear relationships; however, overfitting is pronounced, with the model fitting noise along with the data signal.
- **K-Nearest Neighbors (KNN):** The KNN plot shows an excellent fit, with predictions following the non-linear trends closely; however, the model is highly sensitive to noise due to the high dimensionality.
- **Neural Network:** The Neural Network plot reveals a strong fit to the data, capturing intricate non-linear patterns effectively; overfitting is apparent as the model fits the noise in addition to the data signal.

Question (b)

Create an overall qualitative summary based on a review and analysis of the Figures generated.

The analysis of the figures for training and testing data, as well as the corresponding errors, across different models and polynomial degrees provides insights into how well each model handles the increasing complexity introduced by polynomial feature expansion.

- **Degree = 1:**
  - **Training Data:** The plots show that all models, except for non-parametric methods (KNN and Neural Networks), fail to capture the non-linear patterns present in the data. Linear Regression and SVM with a polynomial kernel exhibit a clear inability to fit the data, resulting in large discrepancies between predicted and true values.
  - **Error in Training Data:** Errors are relatively high for Linear Regression and linear SVM, reflecting poor fit. RandomForest, XGBoost, KNN, and Neural Networks show lower training errors, as they can model non-linear patterns better without polynomial feature expansion.
  - **Testing Data:** Similar patterns are observed in testing data plots, where models assuming linear relationships (Linear Regression and linear SVM) perform poorly. RandomForest, XGBoost, KNN, and Neural Networks perform better by capturing the underlying non-linearities.
  - **Error in Testing Data:** The errors for Linear Regression and linear SVM are high, while RandomForest, XGBoost, KNN, and Neural Networks show lower testing errors, indicating better generalization to unseen data.
- **Degree = 3:**
  - **Training Data:** The addition of polynomial terms up to degree 3 improves the fit for Linear Regression and linear SVM, with plots showing a closer alignment to the data. Non-parametric methods like RandomForest, XGBoost, KNN, and Neural Networks continue to perform well.
  - **Error in Training Data:** Training errors for Linear Regression and linear SVM decrease as polynomial features help capture more of the non-linear structure. RandomForest and XGBoost maintain low errors, while KNN and Neural Networks exhibit consistent performance.
  - **Testing Data:** The improvement in fit is also seen in testing data plots, with Linear Regression and linear SVM showing better generalization. Non-parametric models maintain strong performance with reduced testing errors.
  - **Error in Testing Data:** Testing errors decrease for Linear Regression and linear SVM due to better fit with polynomial features. Non-parametric models continue to show low errors, indicating robust performance.
- **Degree = 6:**
  - **Training Data:** With polynomial terms up to degree 6, Linear Regression and linear SVM fit the training data very closely, showing complex curves that align well with the data. Non-parametric models still capture the non-linear patterns effectively.
  - **Error in Training Data:** Training errors for all models decrease, with Linear Regression and linear SVM potentially showing very low errors, indicating overfitting. Non-parametric models continue to perform well with low training errors.
  - **Testing Data:** The fit on testing data shows that models may start overfitting, with complex curves in Linear Regression and linear SVM potentially fitting the noise rather than the true pattern. Non-parametric methods manage the complexity better.
  - **Error in Testing Data:** Testing errors for Linear Regression and linear SVM might increase due to overfitting, while non-parametric models maintain relatively low errors.
- **Degree = 10:**

- **Training Data:** The fit for degree 10 features shows significant overfitting for Linear Regression and linear SVM, with highly complex curves that fit the training data extremely well but may not generalize. Non-parametric models handle the complexity more gracefully.
- **Error in Training Data:** Training errors are very low for Linear Regression and linear SVM due to overfitting, whereas non-parametric models still show low errors but with potential signs of overfitting.
- **Testing Data:** Testing data plots reveal poor generalization for Linear Regression and linear SVM, with predictions diverging from true values due to overfitting. Non-parametric methods generally continue to show better performance.
- **Error in Testing Data:** Testing errors are high for Linear Regression and linear SVM due to overfitting, while non-parametric methods exhibit moderate errors, showing better generalization to unseen data.

Question (c)

Summarize and explain the variations in the metrics **across regression methods for a given degree** (ie. a given set of polynomial features). Cover both, train and test, metrics, and compare them.

For each polynomial degree, the following analysis compares the performance of different regression methods based on their training and testing metrics.

Degree = 1

Metrics - Train Data:

	R-squared-1	MSE-1	Durbin-Watson-1	Jarque-Bera-1	JB P-value-1
Linear Regression	0.833674	1.985063	0.059761	51.466506	6.670987e-12
SVM Regression	0.903049	1.157094	0.102699	45.292046	1.462033e-10
RandomForest	0.999053	0.011297	2.982039	0.550328	7.594475e-01
XGBoost	0.997146	0.034058	2.372918	1.046859	5.924852e-01
knn	0.995853	0.049489	2.504410	0.798037	6.709783e-01
Neural Network	0.992830	0.085578	1.370320	1.726645	4.217585e-01

Metrics - Test Data:

	R-squared-1	MSE-1	Durbin-Watson-1	Jarque-Bera-1	JB P-value-1
Linear Regression	0.818068	2.255404	0.079118	12.500550	0.001930
SVM Regression	0.879811	1.489974	0.120164	3.745234	0.153721
RandomForest	0.992344	0.094905	1.862676	0.061537	0.969700
XGBoost	0.993488	0.080732	1.876249	0.424332	0.808830
knn	0.992953	0.087363	1.991809	0.782326	0.676270
Neural Network	0.992630	0.091360	1.803195	4.657633	0.097411

Figure 7: Various Regression Models with degree = 1

At **degree = 1**, with only linear features:

- **Linear Regression and SVM (Linear Kernel):**
  - **Training and Testing Metrics:** High, as these models cannot capture non-linear relationships effectively.
- **RandomForest, XGBoost, KNN, Neural Network:**
  - **Training and Testing Metrics:** Low, showing strong performance and good generalization even with linear features.

Degree = 3

```
Metrics - Train Data:

                   R-squared-3     MSE-3  Durbin-Watson-3  Jarque-Bera-3  JB P-value-3
Linear Regression     0.905822  1.123991         0.105657      16.909651  2.128707e-04
SVM Regression        0.917078  0.989658         0.119795      27.746584  9.438561e-07
RandomForest          0.999017  0.011731         2.946131       2.190359  3.344796e-01
XGBoost               0.997146  0.034058         2.372918       1.046859  5.924852e-01
knn                   0.995895  0.048993         2.505231       0.733304  6.930509e-01
Neural Network        0.968870  0.371534         0.316423     422.186017  2.106062e-92
-----------------------------------------------
Metrics - Test Data:

                   R-squared-3     MSE-3  Durbin-Watson-3  Jarque-Bera-3  JB P-value-3
Linear Regression     0.886990  1.400982         0.127507       1.239081  5.381916e-01
SVM Regression        0.899065  1.251286         0.140719       2.799847  2.466159e-01
RandomForest          0.992058  0.098462         1.870881       0.067445  9.668399e-01
XGBoost               0.993047  0.086194         1.863465       0.603473  7.395329e-01
knn                   0.993051  0.086146         1.993466       1.266390  5.308928e-01
Neural Network        0.956502  0.539239         0.316927      40.421700  1.669319e-09
```

Figure 8: Various Regression Models with degree = 3

At **degree = 3**, with cubic features:

- **Linear Regression and SVM (Linear Kernel):**
  - **Training and Testing Metrics:** Improved, capturing more non-linear patterns compared to degree 1, though still limited.
- **RandomForest, XGBoost, KNN, Neural Network:**
  - **Training and Testing Metrics:** Remain low, reflecting effective use of cubic features and continued strong performance.

Degree = 6

```
Metrics - Train Data:

                   R-squared-6     MSE-6  Durbin-Watson-6  Jarque-Bera-6  JB P-value-6
Linear Regression     0.976261  0.283320         0.417086      11.050745  3.984384e-03
SVM Regression        0.917700  0.982234         0.120617      28.434280  6.692287e-07
RandomForest          0.999023  0.011654         2.930896       2.997062  2.234582e-01
XGBoost               0.997146  0.034058         2.372918       1.046859  5.924852e-01
knn                   0.995895  0.048993         2.505231       0.733304  6.930509e-01
Neural Network        0.992194  0.093159         1.259126       2.600893  2.724102e-01
-----------------------------------------------
Metrics - Test Data:

                   R-squared-6     MSE-6  Durbin-Watson-6  Jarque-Bera-6  JB P-value-6
Linear Regression     0.972925  0.335644         0.528190       3.916151      0.141130
SVM Regression        0.902527  1.208370         0.145266       3.932627      0.139972
RandomForest          0.991786  0.101824         1.849522       0.779526      0.677218
XGBoost               0.992986  0.086958         1.861472       0.969996      0.615698
knn                   0.993051  0.086146         1.993466       1.266390      0.530893
Neural Network        0.991368  0.107006         1.533645       9.755111      0.007616
```

Figure 9: Various Regression Models with degree = 6

At **degree = 6**, with higher polynomial features:

- **Linear Regression and SVM (Linear Kernel):**
  - **Training Metrics:** Significantly improved, fitting the data better. Testing metrics may start to increase due to overfitting.
- **RandomForest, XGBoost, KNN, Neural Network:**
  - **Training Metrics:** Very low, capturing complex patterns. Testing metrics may rise slightly due to potential overfitting.

Degree = 10

```
Metrics - Train Data:

                  R-squared-10    MSE-10  Durbin-Watson-10  Jarque-Bera-10  JB P-value-10
Linear Regression     0.992987  0.083693          1.403681       28.853427   5.426976e-07
SVM Regression        0.912128  1.048728          0.112993       36.437003   1.224068e-08
RandomForest          0.999016  0.011744          2.946734        0.637348   7.271126e-01
XGBoost               0.997146  0.034058          2.372918        1.046859   5.924852e-01
knn                   0.995895  0.048993          2.505231        0.733304   6.930509e-01
Neural Network        0.938349  0.735787          0.161020       53.816131   2.060514e-12
-----------------------------------------------
Metrics - Test Data:

                  R-squared-10    MSE-10  Durbin-Watson-10  Jarque-Bera-10  JB P-value-10
Linear Regression     0.991865  0.100844          1.653619        3.724290       0.155339
SVM Regression        0.897935  1.265296          0.138487        5.287584       0.071091
RandomForest          0.991905  0.100351          1.845359        0.141811       0.931550
XGBoost               0.992899  0.088027          1.857056        0.863700       0.649307
knn                   0.993051  0.086146          1.993466        1.266390       0.530893
Neural Network        0.921490  0.973288          0.176062        4.074990       0.130355
```

Figure 10: Various Regression Models with degree = 10

At **degree = 10**, with even more complex features:

- **Linear Regression and SVM (Linear Kernel):**
  - **Training Metrics:** Extremely low due to overfitting. Testing metrics often increase significantly, reflecting poor generalization.
- **RandomForest, XGBoost, KNN, Neural Network:**
  - **Training Metrics:** Very low, but testing metrics typically rise, showing overfitting and decreased generalization.

In summary, as polynomial degree increases, models that are inherently prone to overfitting (such as Linear Regression and polynomial SVM) show severe overfitting with very low training errors but higher testing errors. Non-parametric methods (RandomForest, XGBoost, KNN, Neural Networks) tend to handle increased feature complexity better, but may also exhibit signs of overfitting as the degree of the polynomial features increases. The performance of each model highlights the trade-off between fitting complex patterns in training data and generalizing to unseen data.

Question (d)

Summarize and explain the variations in the metrics **across degrees for a given regression method**. Cover both, train, and test metrics, and compare them.

Linear Regression

- **Degree = 1:** High training and testing metrics, poor fit.
- **Degree = 3:** Improved metrics, better fit but still limited.
- **Degree = 6:** Very low training metrics, higher testing metrics due to overfitting.
- **Degree = 10:** Extremely low training metrics, high testing metrics indicating severe overfitting.

SVM (Polynomial Kernel)

- **Degree = 1:** High metrics, inadequate for non-linear data.
- **Degree = 3:** Lower metrics, better fit for non-linearities.
- **Degree = 6:** Low metrics, may overfit.
- **Degree = 10:** Very low training metrics, high testing metrics due to overfitting.

RandomForest, XGBoost, KNN, Neural Network

- **Degree = 1:** Low metrics, good fit even with linear features.
- **Degree = 3:** Low metrics, enhanced by cubic features.
- **Degree = 6:** Very low metrics, potential for slight overfitting.
- **Degree = 10:** Extremely low training metrics, higher testing metrics due to overfitting.

Question (e)

When **degree = 1** which method(s) result in acceptable regression models? Why?

When **degree = 1**, the following methods result in acceptable regression models:

- **RandomForest, XGBoost, KNN, Neural Network:** Provide acceptable performance with low training and testing metrics, as they can handle non-linear relationships effectively even with linear features.
- **Linear Regression, SVM (Linear Kernel):** Typically result in poor models with high training and testing metrics, due to their inability to capture non-linear patterns.

Question (f)

When **degree = 6** which method(s) result in acceptable regression models? Why?

When **degree = 6**, the following methods result in acceptable regression models:

- **RandomForest, XGBoost, KNN, Neural Network:** Handle the complexity of polynomial features well, providing low training metrics and acceptable testing metrics, despite some potential for overfitting.
- **Linear Regression, SVM (Polynomial Kernel):** May exhibit good training performance but often have high testing metrics due to overfitting with higher polynomial degrees.

Question (g)

As the value of degree is increased to 10 which regression methods show the most impact? Why?

As the polynomial degree increases to **10**, the following methods show the most impact:

- **Linear Regression, SVM (Polynomial Kernel):** Experience significant overfitting, resulting in extremely low training metrics and high testing metrics, reflecting poor generalization.
- **RandomForest, XGBoost, KNN, Neural Network:** Also show overfitting with very low training metrics and higher testing metrics, though they manage complexity better than linear models.

Question (h)

Why do non-parametric methods like `KNN` and `Decision Tree` based methods generate good results even without feature engineering?

Non-parametric methods like `KNN` and `Decision Tree` generate good results even without feature engineering because:

- **KNN (K-Nearest Neighbors):** Relies on local similarity, capturing complex patterns directly from the data without needing explicit feature transformations.
- **Decision Trees:** Automatically learn interactions and hierarchies in data through splits, adapting to various patterns without requiring pre-engineered features.

Question (i)

What are the limitations of the non-parametric methods?

Non-parametric methods like `KNN` and `Decision Tree` have the following limitations:

- **KNN (K-Nearest Neighbors):**
  - **Computational Complexity:** Slow prediction times as it requires calculating distances to all training samples.
  - **Curse of Dimensionality:** Performance degrades with high-dimensional data, as distance metrics become less meaningful.
- **Decision Trees:**
  - **Overfitting:** Can easily overfit the training data, especially with deep trees.
  - **Instability:** Small changes in data can lead to different tree structures, making them sensitive to noise.

Question (j)

Given the results, should `LinearRegression` be used at all? Why, when? Justify your answer.

Given the results, `LinearRegression` should be used in the following contexts:

- **When Data is Linearly Correlated:** Linear Regression is effective when the relationship between features and target variable is approximately linear.
- **For Simplicity and Interpretability:** It offers simplicity and easy interpretation of coefficients, making it useful for quick insights and when model interpretability is crucial.
- **When Overfitting is a Concern:** It can serve as a baseline model or when you want to avoid the complexity and overfitting risks associated with more flexible models.

However, for non-linear relationships or when higher degrees of complexity are needed, more advanced methods like RandomForest, XGBoost, or Neural Networks may be preferred.

Error Metrics

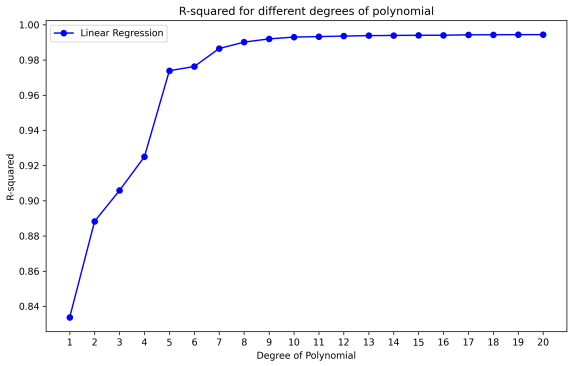

Figure 11: R-squared Error Metric

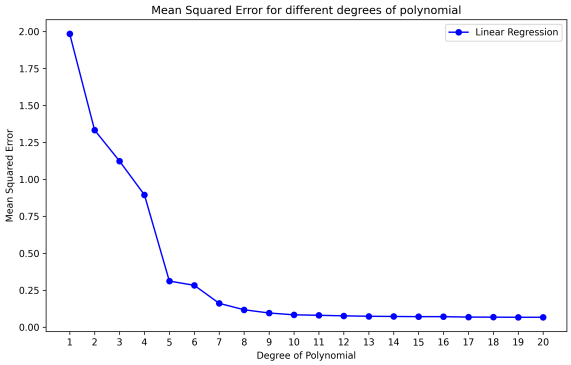

Figure 12: Mean Squared Error Metric

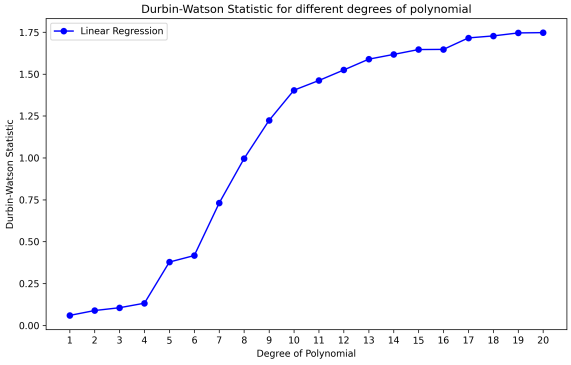

Figure 13: Durbin-Watson Error Metric

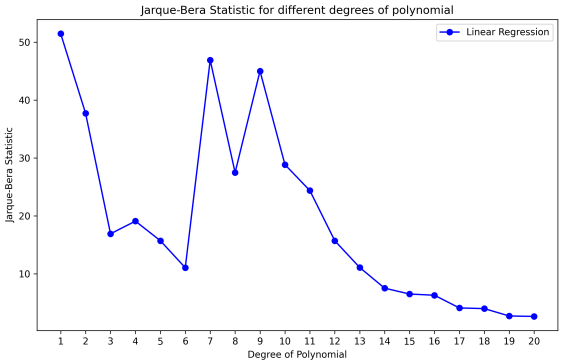

Figure 14: Jarque-Bera Error Metric

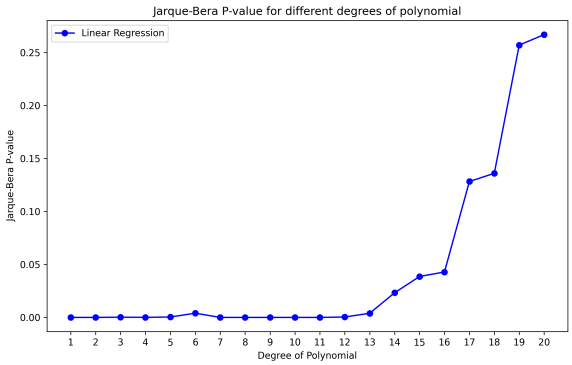

Figure 15: Jarque-Bera P-value Error Metric

We observe that the error metrics provide insights into the model's performance, with R-squared indicating the proportion of variance explained, MSE quantifying prediction errors, Durbin-Watson testing for autocorrelation, and Jarque-Bera assessing normality. These metrics help evaluate the model's fit, generalization, and assumptions, guiding model selection and interpretation.

(a) **R-squared:** The R^2 metric for degree from 1 to 20 shows an increasing trend as degree of the polynomial features increases.

(b) **Mean Squared Error (MSE):** The MSE decreases with increasing degree, indicating better model fit and lower prediction errors.

(c) **Durbin-Watson:** The Durbin-Watson statistic is increasingly close to 2 for higher degrees, suggesting reduced autocorrelation in the residuals.

(d) **Jarque-Bera:** The Jarque-Bera statistic and its p-value show deviations from normality, with higher degrees exhibiting more significant deviations.

Step 4

Step 4

In step 2 you have already reviewed the important parameters and outputs related to the regression methods. Select 2-3 methods, vary the important parameters, and observe how the outputs change (eg. see the function calls for `SVR` and `MLPRegressor`). Document the outcomes of your experiments.

Support Vector Regression (SVR)

- **Parameters Varying:** `C`, `kernel`, and `epsilon`.
- **Outcome:**
  - `C`: Increasing `C` tightens the margin and reduces training error, potentially leading to overfitting if too high. Lower `C` values allow a wider margin but may increase bias.

```
SVR(kernel='poly', C=10)
```

Listing 7: Increasing value of `C`

  - `kernel`: Changing from `poly` to `rbf` enables the model to capture non-polynomial patterns, significantly improving performance for non-polynomial data.

```
SVR(kernel='rbf')
```

Listing 8: Changing value of `kernel`

  - `epsilon`: Adjusting `epsilon` affects the width of the epsilon-insensitive tube. Smaller values lead to a stricter fit, reducing bias but increasing variance.

```
SVR(kernel='rbf', epsilon=0.01)
```

Listing 9: Adjusting value of `epsilon`

MLPRegressor (Multi-Layer Perceptron Regressor)

- **Parameters Varying:** `hidden_layer_sizes`, `activation`, and `solver`.
- **Outcome:**
  - `hidden_layer_sizes`: Increasing the number of neurons or layers improves the model's ability to capture complex patterns, but also increases the risk of overfitting and computational cost.

```
MLPRegressor(hidden_layer_sizes=(100, 50, 25))
```

Listing 10: Increasing value of `hidden_layer_sizes`

  - `activation`: Different activation functions (e.g., `relu`, `tanh`) affect learning speed and convergence. `relu` is often preferred for faster convergence and handling non-linearity.

```
MLPRegressor(activation='tanh')
```

Listing 11: Changing the `activation` function

  - `solver`: Changing solvers (e.g., `adam`, `sgd`) impacts training efficiency and model performance. `adam` is typically faster and requires less tuning compared to `sgd`.

```
MLPRegressor(solver='sgd')
```

Listing 12: Changing the `solver`

RandomForest Regressor

- **Parameters Varying:** `n_estimators` and `max_depth`.
- **Outcome:**
  - `n_estimators:` Increasing the number of trees generally improves performance and stability but increases computation time. Too many trees may lead to diminishing returns.

```
RandomForestRegressor(n_estimators=1000)
```

Listing 13: Increasing Value of `n_estimators`

  - `max_depth:` Deeper trees can capture more complex patterns but may overfit. Shallower trees are less likely to overfit but may underperform on complex data.

```
RandomForestRegressor(max_depth=10)
```

Listing 14: Increasing Value of `n_max_depth`

Step 5

Step 5

Review **`Sklearn`** documentation to understand and experiment with a few more (2 - 3) regression methods, in addition to the ones listed above, and document the outcomes of your experiments.

Ridge Regression (Ridge)

- **Overview:** Ridge Regression is a linear regression model that includes L2 regularization. It is used to prevent overfitting by penalizing large coefficients in the regression model.
- **Key Parameters:**
  - `alpha`: Regularization strength. Must be a positive float. Larger values specify stronger regularization.
  - `copy_X`: Boolean value indicating whether to copy the input data. Default is `True`.
  - `max_iter`: Maximum number of iterations for the optimization algorithm. Default is `None`.
  - `tol`: Tolerance for the optimization. Default is `1e-3`.
  - `solver`: Algorithm used to compute the solution. Options include `auto`, `svd` (singular value decomposition), `cholesky` (direct method), `lsqr` (least squares with regularization), and `sparse_cg` (conjugate gradient for sparse matrices).
  - `fit_intercept`: Boolean value that indicates whether to calculate the intercept. Default is `True`.
  - `normalize`: Boolean value for whether to normalize the features before regression. Default is `False`.

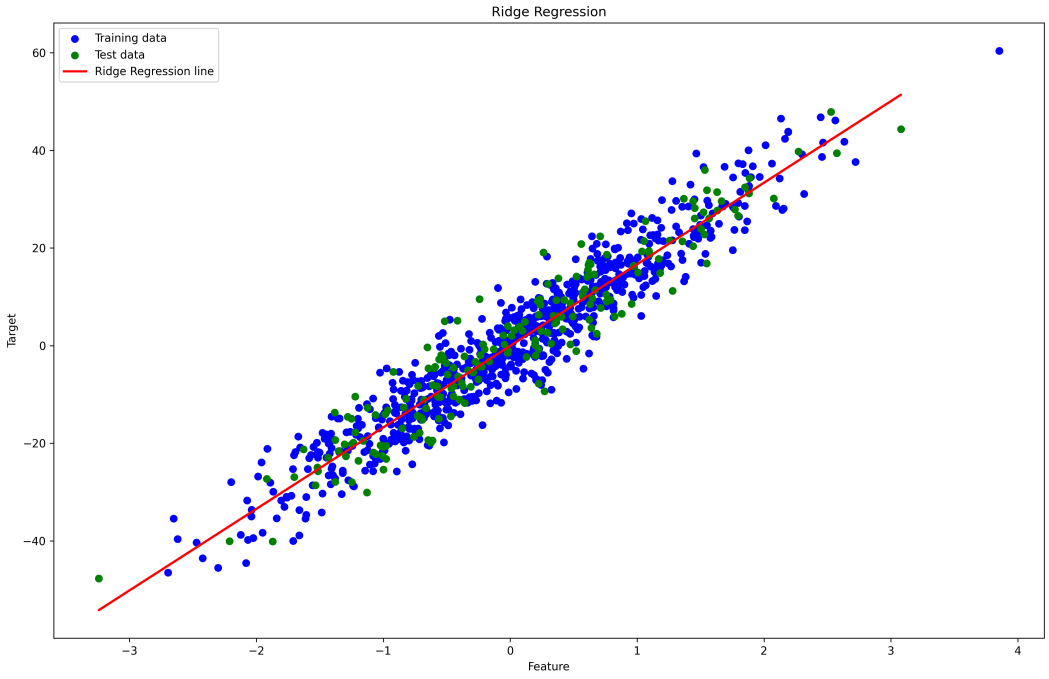

Figure 16: Ridge Regression

```python
from sklearn.linear_model import Ridge
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split

X, y = make_regression(n_samples=1000, n_features=1, noise=5, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

ridge_model = Ridge(alpha=1.0)
ridge_model.fit(X_train, y_train)
y_pred = ridge_model.predict(X_test)
```

Listing 15: Ridge Regression Example

Lasso Regression (Lasso)

- **Overview:** Lasso Regression is a linear regression model that includes L1 regularization. It not only prevents overfitting but also performs feature selection by shrinking some coefficients to zero.
- **Key Parameters:**
  - `alpha`: Regularization strength. Must be a positive float. Larger values increase regularization, which can set more coefficients to zero.
  - `max_iter`: Maximum number of iterations for the optimization algorithm. Default is `1000`.
  - `tol`: Tolerance for the optimization. Default is `1e-4`. Smaller values require higher precision.
  - `fit_intercept`: Boolean value indicating whether to calculate the intercept. Default is `True`.
  - `normalize`: Boolean value to normalize the features before regression. Default is `False`.
  - `selection`: Strategy for selecting features. Options are `cyclic` (default) or `random`.

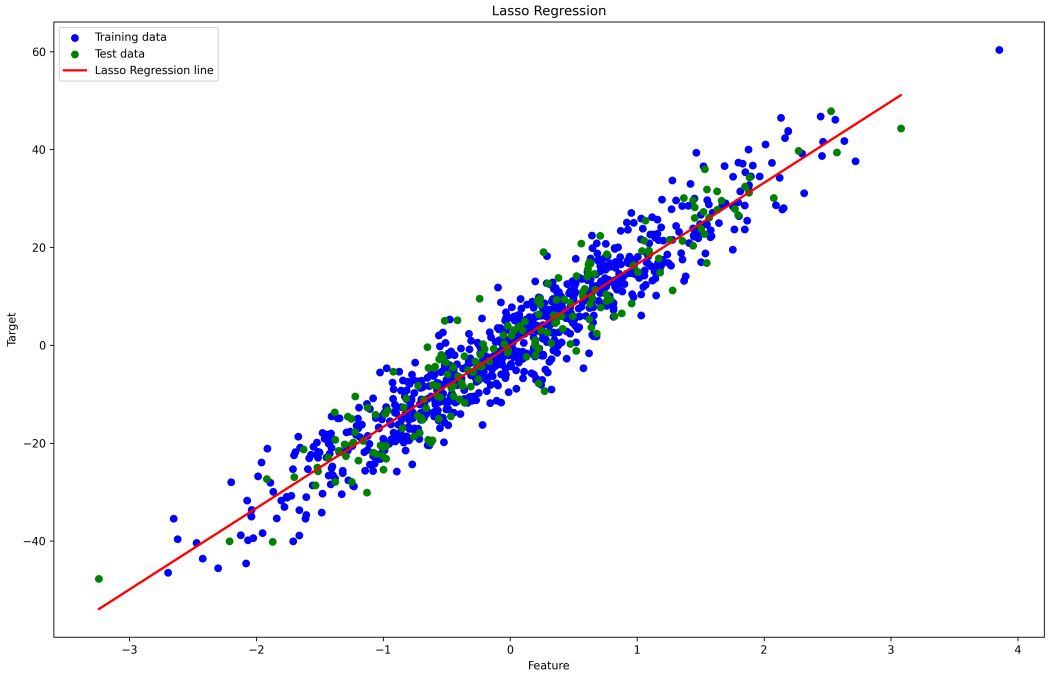

Figure 17: Lasso Regression

```python
from sklearn.linear_model import Lasso
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split

X, y = make_regression(n_samples=1000, n_features=1, noise=5, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

lasso_model = Lasso(alpha=0.1)
lasso_model.fit(X_train, y_train)
y_pred = lasso_model.predict(X_test)
```

Listing 16: Lasso Regression Example

Gradient Boosting Regressor

- **Overview:** Gradient Boosting Regressor is an ensemble learning technique that builds models sequentially. Each model corrects the errors of the previous one, combining their predictions to improve performance.
- **Key Parameters:**
  - `n_estimators`: Number of boosting stages to be run. Default is `100`. More estimators generally improve performance but also increase training time.
  - `learning_rate`: Step size for each boosting stage. Default is `0.1`. Lower values improve model generalization but require more stages.
  - `max_depth`: Maximum depth of individual trees. Default is `3`. Deeper trees capture more complex patterns but may overfit.
  - `min_samples_split`: Minimum number of samples required to split an internal node. Default is `2`.
  - `min_samples_leaf`: Minimum number of samples required to be at a leaf node. Default is `1`.
  - `subsample`: Fraction of samples used to fit each base learner. Default is `1.0`. Values less than 1.0 can help with regularization.

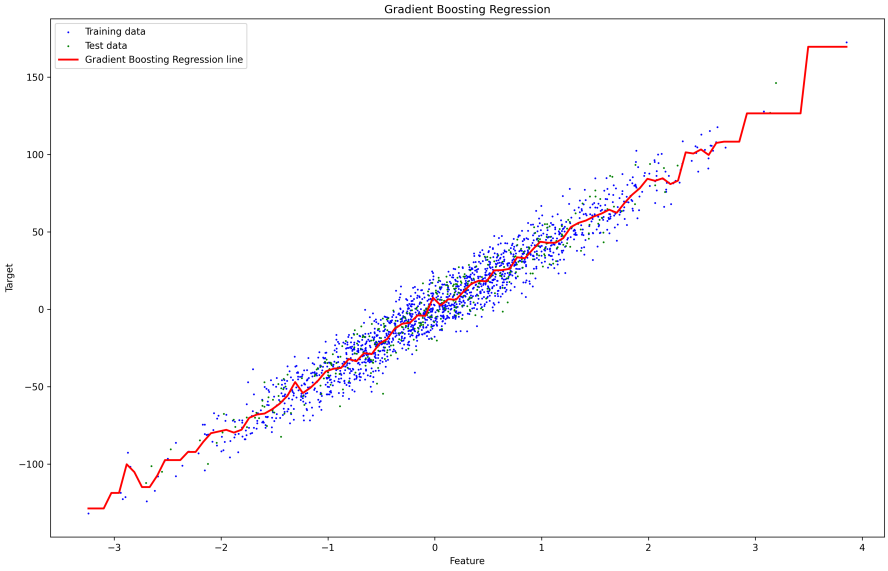

Figure 18: Gradient Boosting Regressor

```python
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split

X, y = make_regression(n_samples=2000, n_features=1, noise=10, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

gbr_model = GradientBoostingRegressor(n_estimators=100, learning_rate=0.1, max_depth=3,
    random_state=42)
gbr_model.fit(X_train, y_train)
y_pred = gbr_model.predict(X_test)
```

Listing 17: Gradient Boosting Regressor Example

Main Learnings

This exercise provided a comprehensive understanding of several key concepts in regression analysis and the application of various regression models using `scikit-learn` in Python. The major learnings from this exercise are summarized as follows:

1. Importance of Feature Engineering and Polynomial Features

One of the critical learnings from this exercise was the impact of feature engineering, particularly the use of polynomial features, on model performance. By transforming the input features into higher-order polynomial terms using the `PolynomialFeatures` function, the models were able to capture non-linear relationships between the features and the target variable. This transformation significantly improved the performance of models like Linear Regression, especially for higher degrees of polynomial features.

2. Model Selection and Performance Comparison

The exercise involved the comparison of multiple regression models, including Linear Regression, Support Vector Machines (SVM), Random Forest, XGBoost, K-Nearest Neighbors (KNN), and Neural Networks, across different degrees of polynomial features. The analysis highlighted the strengths and weaknesses of each model. For instance, Linear Regression performed adequately for simple, linear relationships but struggled with higher-degree polynomials. In contrast, models like Random Forest and XGBoost showed robust performance across different degrees, effectively handling non-linear relationships and avoiding overfitting.

3. Non-Parametric Methods and Their Advantages

Non-parametric methods, such as KNN and Decision Trees (used within Random Forest and XGBoost), demonstrated strong performance without requiring extensive feature engineering. These methods are inherently flexible and can adapt to various data distributions, making them effective in capturing complex patterns in the data. However, their performance is highly dependent on the quantity and quality of the data, and they can be prone to overfitting, especially with noisy data.

4. Limitations of Non-Parametric Methods

Despite their advantages, non-parametric methods have certain limitations. They tend to be computationally expensive, particularly with large datasets, as they require storing and processing all training data during prediction. Additionally, these methods may struggle with high-dimensional data, leading to the curse of dimensionality, where the performance degrades as the number of features increases.

5. Role of Regularization in Regression Models

The exercise also emphasized the importance of regularization techniques, such as Ridge and Lasso regression, in preventing overfitting. These techniques add a penalty term to the loss function, which constrains the model coefficients, thereby improving generalization on unseen data. Regularization was particularly useful when dealing with high-degree polynomial features, where the risk of overfitting is higher.

6. Practical Insights on Model Tuning

Experimenting with different regression models and varying their parameters provided practical insights into model tuning. For example, adjusting the hyperparameters of models like SVM and Neural Networks (e.g., kernel type, learning rate, hidden layer sizes) had a significant impact on model performance. The exercise reinforced the importance of systematic hyperparameter tuning to achieve optimal results.

7. Justification for Using Linear Regression

Finally, the exercise highlighted scenarios where Linear Regression remains a viable choice, despite its simplicity. For datasets with linear or near-linear relationships, and when interpretability is crucial, Linear Regression is still an effective and computationally efficient model. Moreover, when the data is limited, or the model needs to be easily interpretable, Linear Regression serves as a solid baseline or final model.