Islamic University – Gaza
Faculty of Information Technology
Department of Information Technology

الجامعة الإسلامية – غزة
كلية تكنولوجيا المعلومات
ماجستير تكنولوجيا المعلومات

Evaluating Regressors for Baby Birth Weight Estimation

Using the Birth Weight dataset

By:

Abeer Yousef Abu Mosameh -220232641

Supervised By:

Dr. Iyad Husni Alshami

Jan, 2025

Introduction

The field of machine learning aims to enable computers to learn from data and make decisions or predictions without explicit instructions. One of the common tasks in machine learning is regression, where we predict a continuous value for an input based on its features.

In this report, we will train, evaluate, and compare the performance of different regressors in predicting the birth weight of newborn babies using the "birth-weight-dataset". Our primary goal is to use various regressors for estimating baby birth weights and identify the most accurate approach.

We will begin by preparing the dataset through necessary preprocessing steps, then train the following Regressors 1) *Linear Regression model* (Lasso or Ridge), 2) *Polynomial Regression* (using fit_curve for linear/non-linear regression), 3) *Support Vector Machine (SVM) Regressor*, 4) *XGBoost Regressor* and 5) *Random Forest Regressor* (with 50 estimators). The evaluation will be based on the Root Mean Squared Error (RMSE) metric. By analyzing the RMSE of these different regressors, we aim to identify the most effective model for estimating the birth weight of newborn babies. Finally, we will compare the performance and determine the best regressor for this task.

Dataset Preparation

The Baby Birth Weight dataset consists of information on various factors affecting the bir th weight of newborn babies. This dataset includes several features related to parental de mographics, pregnancy characteristics, and health conditions. The dataset contains a total of 101400 rows and 37 attributes.

Data Loading

Loading Datasets

[4]:
```python
# Load the baby weights dataset
dataset = pd.read_csv('baby-weights-dataset.csv')

dataset
```

[4]:

	ID	SEX	MARITAL	FAGE	GAINED	VISITS	MAGE	FEDUC	MEDUC	TOTALP	...	HYPERCH	HYPERPR	ECLAMP	CERVIX	PINFANT	PRETERM	RENAL	F
0	2001	2	1	33	26.0	10	34	12.0	4	2	...	0	0	0	0	0	0	0	
1	2002	2	2	19	40.0	10	18	11.0	12	1	...	0	0	0	0	0	0	0	
2	2003	2	1	33	16.0	14	31	16.0	16	2	...	0	0	0	0	0	0	0	
3	2004	1	1	25	40.0	15	28	12.0	12	3	...	0	0	0	0	0	0	0	
4	2005	1	2	21	60.0	13	20	12.0	14	2	...	0	1	0	0	0	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
101395	103396	1	2	36	0.0	9	34	3.0	12	4	...	0	0	0	0	0	0	0	
101396	103397	2	2	21	39.0	11	19	12.0	9	2	...	0	0	0	0	0	0	0	
101397	103398	2	1	27	37.0	15	22	12.0	12	2	...	0	0	0	0	0	0	0	
101398	103399	1	1	27	26.0	12	24	12.0	14	1	...	0	0	0	0	0	0	0	
101399	103400	1	2	20	31.0	15	17	12.0	11	1	...	0	0	0	0	0	0	0	

101400 rows × 37 columns

Data Exploration

Important step in the data analysis process. It involves examining the dataset to understand its structure, identify patterns, detect outliers.

1. Describing columns in dataset , give overview of the numerical columns including mean, min, std, count, quartiles, and max.

describing columns in dataset .. provides a summary of each feature, including mean, standard deviation, and quartiles.

```
[7]: dataset.describe()
```

[7]:

	ID	SEX	MARITAL	FAGE	GAINED	VISITS	MAGE	FEDUC	MEDUC	TOTALP	...	
count	101400.000000	101400.000000	101400.000000	101400.000000	101399.000000	101400.000000	101400.000000	101399.000000	101400.000000	101400.000000	...	101
mean	52700.500000	1.485671	1.303817	30.174477	30.283040	12.436943	27.736312	12.926883	13.256489	2.378462	...	
std	29271.802985	0.500349	0.459907	6.775576	13.615468	3.728901	5.957369	2.926582	2.932693	1.490272	...	
min	2001.000000	1.000000	1.000000	14.000000	0.000000	0.000000	11.000000	0.000000	0.000000	1.000000	...	
25%	27350.750000	1.000000	1.000000	25.000000	21.000000	10.000000	23.000000	12.000000	12.000000	1.000000	...	
50%	52700.500000	1.000000	1.000000	30.000000	30.000000	12.000000	28.000000	12.000000	13.000000	2.000000	...	
75%	78050.250000	2.000000	2.000000	35.000000	39.000000	15.000000	32.000000	16.000000	16.000000	3.000000	...	
max	103400.000000	9.000000	2.000000	74.000000	98.000000	49.000000	53.000000	17.000000	17.000000	20.000000	...	

8 rows × 35 columns

2. Dataset Size

dataset size : number of columns , rows

```
[9]: dataset.shape
```

```
[9]: (101400, 37)
```

3. Information about dataset , columns name , type , count “summary of dataset”

dataset information

```
[16]: dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 101400 entries, 0 to 101399
Data columns (total 37 columns):
 #   Column    Non-Null Count   Dtype
---  ------    --------------   -----
 0   ID        101400 non-null  int64
 1   SEX       101400 non-null  int64
 2   MARITAL   101400 non-null  int64
 3   FAGE      101400 non-null  int64
 4   GAINED    101399 non-null  float64
 5   VISITS    101400 non-null  int64
 6   MAGE      101400 non-null  int64
 7   FEDUC     101399 non-null  float64
 8   MEDUC     101400 non-null  int64
 9   TOTALP    101400 non-null  int64
 10  BDEAD     101400 non-null  int64
 11  TERMS     101400 non-null  int64
 12  LOUTCOME  101400 non-null  int64
 13  WEEKS     101399 non-null  float64
 14  RACEMOM   101400 non-null  int64
 15  RACEDAD   101400 non-null  int64
 16  HISPMOM   101400 non-null  object
 17  HISPDAD   101400 non-null  object
 18  CIGNUM    101399 non-null  float64
 19  DRINKNUM  101400 non-null  int64
 20  ANEMIA    101400 non-null  int64
 21  CARDIAC   101400 non-null  int64
 22  ACLUNG    101400 non-null  int64
 23  DIABETES  101400 non-null  int64
 24  HERPES    101400 non-null  int64
 25  HYDRAM    101399 non-null  float64
 26  HEMOGLOB  101400 non-null  int64
```

Convert column name to lowercase

```
[12]: # make column name lowercase
dataset.columns = dataset.columns.str.lower()
dataset.columns
```

```
[12]: Index(['id', 'sex', 'marital', 'fage', 'gained', 'visits', 'mage', 'feduc',
       'meduc', 'totalp', 'bdead', 'terms', 'loutcome', 'weeks', 'racemom',
       'racedad', 'hispmom', 'hispdad', 'cignum', 'drinknum', 'anemia',
       'cardiac', 'aclung', 'diabetes', 'herpes', 'hydram', 'hemoglob',
       'hyperch', 'hyperpr', 'eclamp', 'cervix', 'pinfant', 'preterm', 'renal',
       'rhsen', 'uterine', 'bweight'],
      dtype='object')
```

Drop id column “not useful” : can use index nested

```
[13]: # Drop the id column
dataset.drop(columns=['id'], inplace=True)
```

4. Checking the distribution of data in different classes

Checking the distribution of data in categorical different classes

```
[14]: # Check the distribution of the 'sex' column
dataset['sex'].value_counts()
```

```
[14]: sex
1    52160
2    49239
9        1
Name: count, dtype: int64
```

```
[15]: # Check the distribution of the 'marital' column
dataset['marital'].value_counts()
```

```
[15]: marital
1    70593
2    30807
Name: count, dtype: int64
```

```
[16]: # Check the distribution of the 'racemom' column
dataset['racemom'].value_counts()
```

Here we notice outlier on sex column

5. Checking for Missing Values: Identify any missing values in the dataset to ensure data integrity *“Ensures that there are no incomplete data entries that could affect the analysis.”*

Checking For missing Values

```
[30]: dataset.isnull().sum() #sum of null values
```

```
[30]: id          0
sex         0
marital     0
fage        0
gained      1
visits      0
mage        0
feduc       1
meduc       0
totalp      0
bdead       0
terms       0
loutcome    0
weeks       1
racemom     0
racedad     0
hispmom     0
hispdad     0
cignum      1
drinknum    0
anemia      0
cardiac     0
aclung      0
diabetes    0
herpes      0
```

Here we notice null values on some columns

6. Checking for Duplication: *Detects repetitive rows that could bias the model.*

Checking for duplication

```
[23]: dataset.duplicated().sum() # sum of duplicated values
```

```
[23]: 2
```

```
[24]: dataset[dataset.duplicated()]
```

[24]:

	sex	marital	fage	gained	visits	mage	feduc	meduc	totalp	bdead	...	hyperch	hyperpr	eclamp	cervix	pinfant	preterm	renal	rhsen	uterine	bweigh
45040	2	1	20	25.0	12	22	12.0	14	2	0	...	0	0	0	0	0	0	0	0	0	1.437
54306	1	2	28	11.0	5	26	10.0	10	6	0	...	0	0	0	0	0	0	0	0	0	3.625

2 rows × 36 columns

Data Cleaning

Important step in data preprocessing, ensuring that the dataset is free from errors, inconsistencies, and irrelevant data. This step involves handling missing values, detecting and removing duplicates, and ensuring data integrity.

1. Handling missing values: handle with drop rows has null values because just 5 rows

```
[20]: dataset.isnull().sum().sum() # num of rows has null values
```

```
[20]: 5
```

Handel Missing values

```
[23]: # Drop rows with any missing values
dataset.dropna(inplace=True)
dataset.isnull().sum() # should be no missing values
```

```
[23]: id         0
sex        0
marital    0
fage       0
gained     0
visits     0
mage       0
feduc      0
meduc      0
```

2. Handling Duplicated values

Handel duplicated values with drop/delete

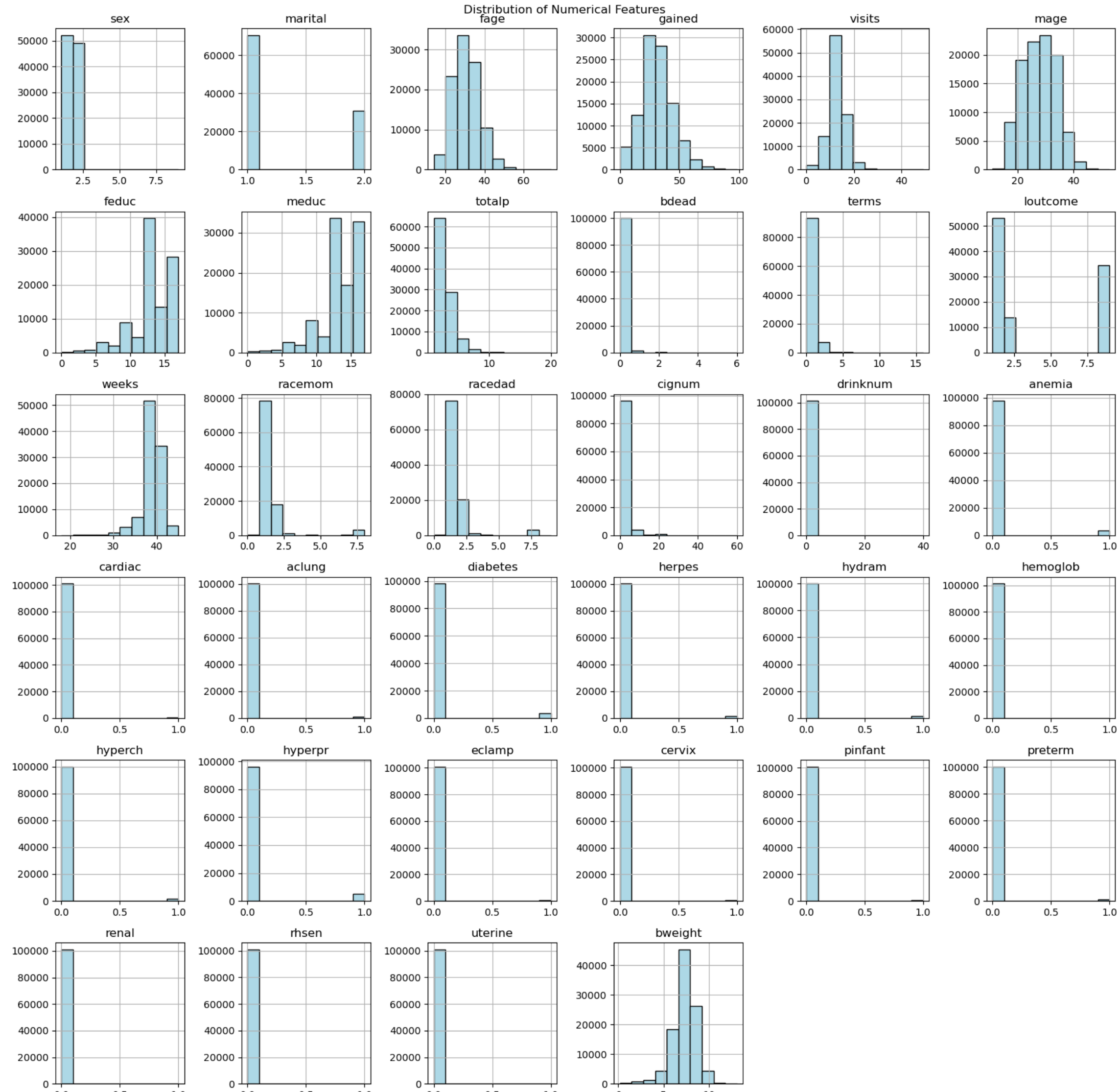
```
[25]: print(len(dataset))
dataset = dataset.drop_duplicates(keep='first')
print(len(dataset))
```

```
101400
101398
```

Data Visualization

Powerful step for understanding the patterns and relationships in a dataset.

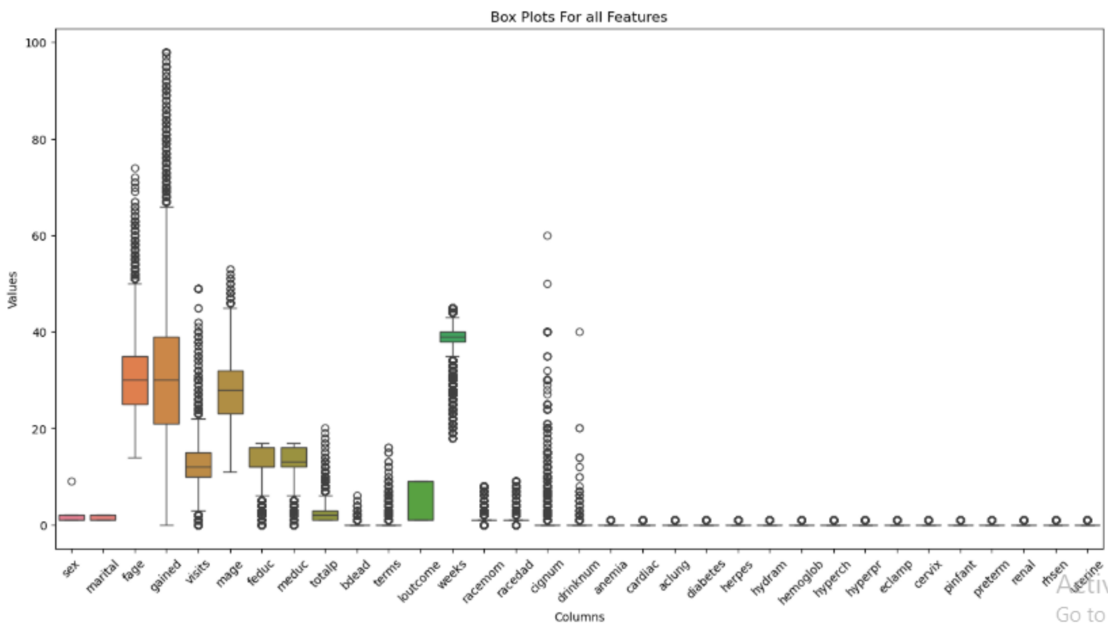
1. Distribution of Numerical Features: understand the spread and central tendency of the data.



Use boxplot to visualize outlier using seborn library for all columns

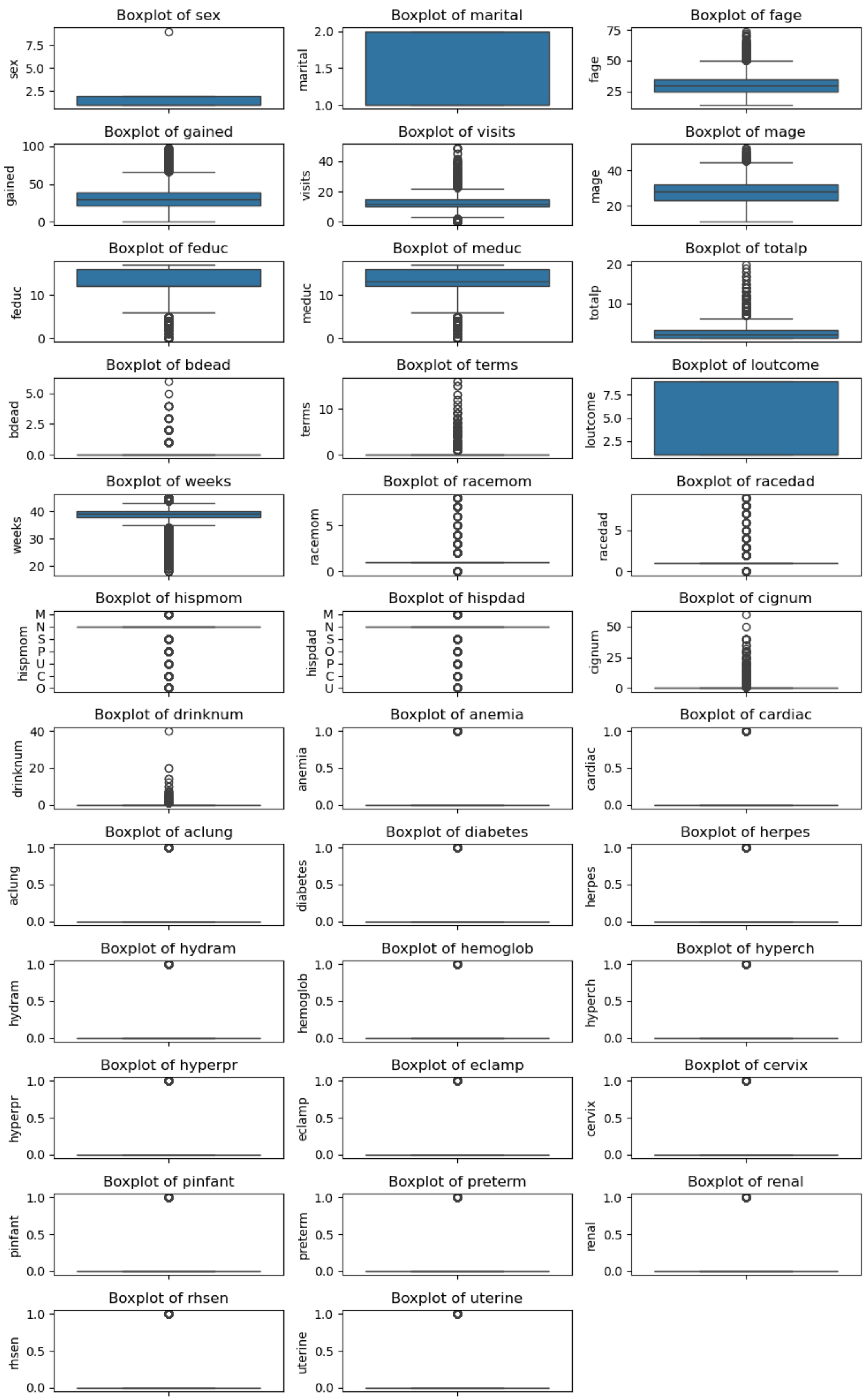
2. Box plot to each future: tool for visualizing the distribution of data and identifying outliers *"outliers can significantly impact the performance of models"*

[30]:
```
# boxplot for all columns in the dataset
plt.figure(figsize=(16, 8))
sns.boxplot(data=dataset.iloc[:, :-1]) # except last/object column
plt.title('Box Plots For all Features')
plt.xticks(rotation=45)
plt.ylabel('Values')
plt.xlabel('Columns')

plt.savefig('single_boxplot.png')
```



Box blot for each Column

[31]:
```
plt.figure(figsize=(10, 16))

# boxplots for each column
for i, column in enumerate(dataset.columns[:-1], 1):  # except the last/object column
    plt.subplot(12, 3, i)  # in 12 rows , 3 columns
    sns.boxplot(data=dataset, y=column)
    plt.title(f'Boxplot of {column}')
plt.tight_layout()  # spacing between subplots

# plt.savefig('individual_boxplots.png')
```



Boxplot of sex
Boxplot of marital
Boxplot of fage
Boxplot of gained
Boxplot of visits
Boxplot of mage
Boxplot of feduc
Boxplot of meduc
Boxplot of totalp
Boxplot of bdead
Boxplot of terms
Boxplot of loutcome
Boxplot of weeks
Boxplot of racemom
Boxplot of racedad
Boxplot of hispmom
Boxplot of hispdad
Boxplot of cignum
Boxplot of drinknum
Boxplot of anemia
Boxplot of cardiac
Boxplot of aclung
Boxplot of diabetes
Boxplot of herpes
Boxplot of hydram
Boxplot of hemoglob
Boxplot of hyperch
Boxplot of hyperpr
Boxplot of eclamp
Boxplot of cervix
Boxplot of pinfant
Boxplot of preterm
Boxplot of renal
Boxplot of rhsen
Boxplot of uterine

We notice Some outlier so make Data Cleaning again:

2.1 Handel outlier by removing/dropping it

A. Remove Outlier in sex column

```
[32]: # handel sex column has outlier with value 9
      print("len of dataset befor drop sex column outlier : ", len(dataset))
      dataset = dataset[dataset['sex'] != 9]
      print("len of dataset after drop sex column outlier : ", len(dataset))

      len of dataset befor drop sex column outlier :  101393
      len of dataset after drop sex column outlier :  101392
```

B. Verify all binary column with 0,1 value and remove others

```
[33]: def handle_binary_columns(dataset, binary_columns):
          # hande binary columns -  only rows with 0 or 1 values
          for col in binary_columns:
              dataset = dataset[dataset[col].isin([0, 1])]

          return dataset
```

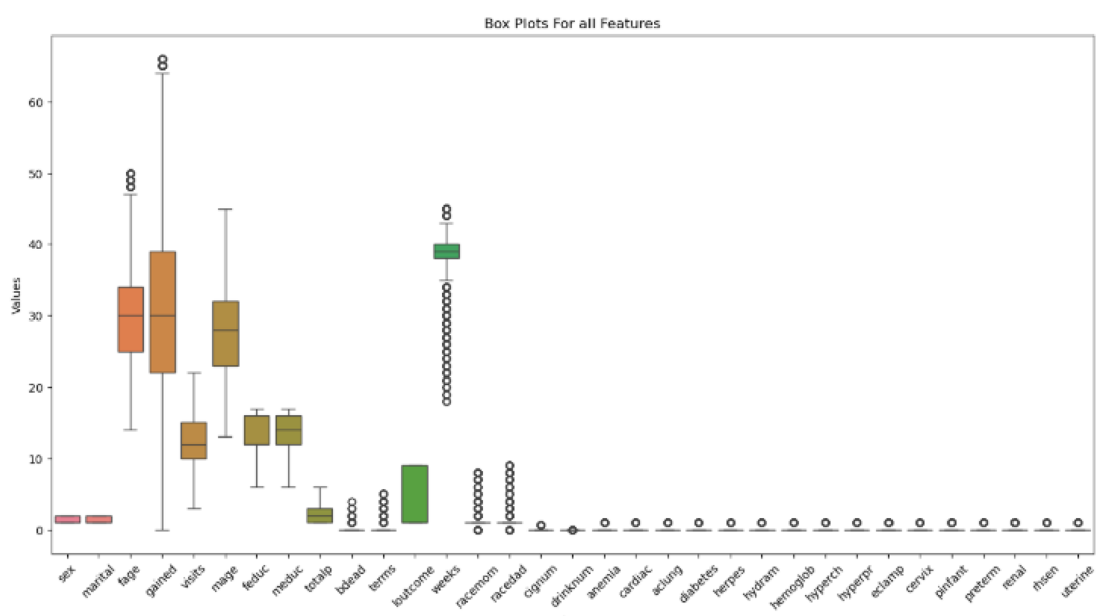
```
[34]: print(len(dataset))
      dataset = handle_binary_columns(dataset , ['anemia', 'cardiac', 'aclung', 'diabetes', 'herpes']) # verify no outlier  " box plot explin this"
      print(len(dataset))

      101392
      101392
```

c. IQR fun to remove outlier in some columns

```
[35]: def remove_outliers(dataset, columns):
          for col in columns:
              if dataset[col].dtype != 'object':
                  q1_value = dataset[col].quantile(0.25)
                  q3_value = dataset[col].quantile(0.75)
                  IQR = q3_value - q1_value
                  max_value = q3_value + (1.5 * IQR)
                  min_value = q1_value - (1.5 * IQR)
                  dataset = dataset[(dataset[col] >= min_value) & (dataset[col] <= max_value)]
          return dataset
```

```
[36]: print(len(dataset))
      dataset = remove_outliers(dataset , ['fage', 'gained', 'visits', 'mage', 'feduc', 'meduc', 'totalp'])
      print(len(dataset))

      101392
      94054
```

Data after remove some outlier



2.2 Handle using clip

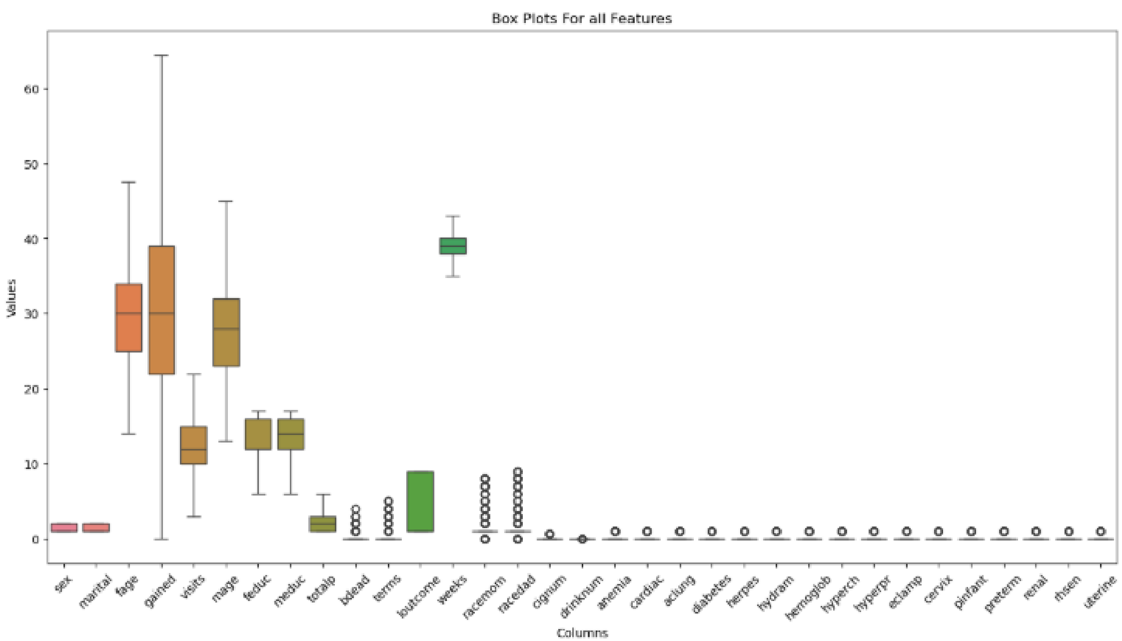
Use `clip()` function : way to handle outliers by capping them to a specified lower and upper limit

```
[50]: # other function to handel outlier : Clip
def clip_outliers(dataset , columns):
  for col in columns:
    if dataset[col].dtype != 'object':
      q1_value = dataset[col].quantile(0.25)
      q3_value = dataset[col].quantile(0.75)
      IQR = q3_value - q1_value
      max = q3_value + (1.5 * IQR)
      min = q1_value - (1.5 * IQR)
      dataset[col] = dataset[col].clip(lower=min, upper=max)
  return dataset
```

```
[51]: print(len(dataset))
dataset = clip_outliers(dataset , ['fage','gained','weeks'])
print(len(dataset))
```

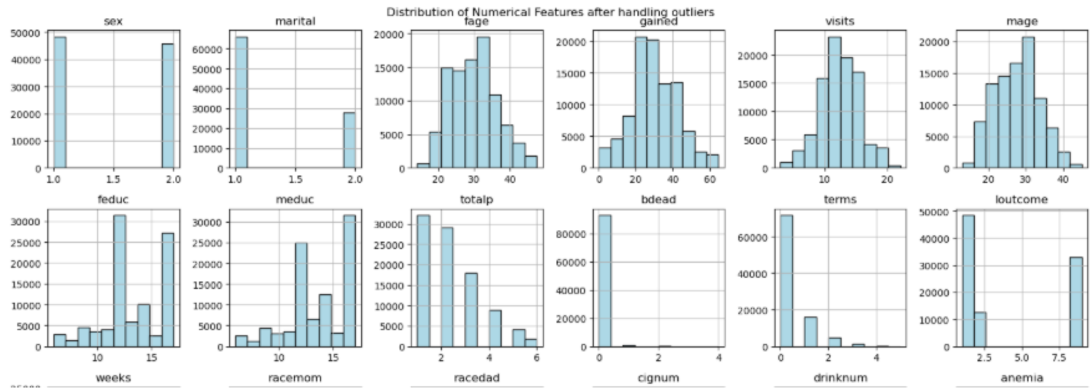
```
94054
94054
```

Data after Handling Outliers



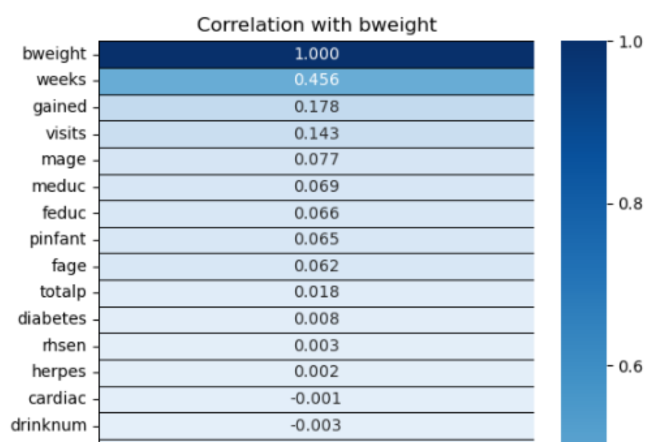
Boxplot/ distribution for each column again after handle outlier

```
[54]: # Plot histograms for numerical features
numeric_columns = dataset.select_dtypes(include=['float64', 'int64']).columns
dataset[numeric_columns].hist(bins=10, figsize=(16, 16), color='lightblue', edgecolor='black')
plt.suptitle('Distribution of Numerical Features after handling outliers')
plt.tight_layout()
```



3. Heat map to show features that has largest influence on target variable

```
[56]: # heatmap
plt.figure(figsize=(6, 10))
sns.heatmap(correlation_with_bweight, annot=True, cmap='Blues', linewidths=0.5, linecolor='black', fmt='.3f')
plt.title('Correlation with bweight')
plt.yticks(rotation=0)

# Save the heatmap
plt.savefig('heatmap_bweight.png')
```



Feature Engineering

Powerful step for transforming raw data into meaningful features that better represent the underlying problem, enhancing the model's predictive performance

Feature	Type	Description	Use Case
Average Parent Age	Numerical	The average age of the parents might influence birth weight, as older parents may have different health conditions or lifestyles.	Helps assess the impact of parental age on birth weight.
Total Parent Education	Numerical	Sum of the education levels of both parents. More educated parents might follow better healthcare practices.	Evaluates the influence of parental education on birth weight
Gained Weight per Visit	Numerical	Weight gained by the mother per prenatal visit, indicating maternal health monitoring.	Provides insight into mom health during pregnancy.
Mother's Age Category	Categorical	Categorizes the mother's age into four groups: <20, 20-30, 30-40, and >40.	Analyzes risks associated with different maternal age groups.
Visits-Weeks Interaction	Numerical	Interaction between the number of prenatal visits and weeks of pregnancy.	Evaluates the impact of prenatal care frequency on health outcomes.
Mother's Health Index	Numerical	Combines various health-related columns into a single score representing overall maternal health status.	Assesses the cumulative health risks of the mother.
Mother's Health Category	Categorical	Groups the mother's health into categories: Good, Moderate, Poor, and Very Poor based on the health index.	Understands the effect of maternal health status on birth weight.

[59]:

```python
# new columns can influence in bewieght
# average age of the parents might influence the bweight (older parents might have different health conditions or lifestyles)
dataset['avg_parent_age'] = (dataset['mage'] + dataset['fage']) / 2

# total education level of the parents might influence the bweight (more educated parents might follow healthcare)
dataset['total_parent_educ'] = dataset['meduc'] + dataset['feduc']

# provide insight about the mother health during pregnancy
dataset['gained_per_visit'] = dataset['gained'] / dataset['visits']

# to now category: very young or older mothers might have different risks
dataset['mage_category'] = pd.cut(dataset['mage'], bins=[0, 20, 30, 40, 50], labels=['<20', '20-30', '30-40', '>40'])

# more visits during a pregnancy - better health outcomes
dataset['visits_weeks_interaction'] = dataset['visits'] * dataset['weeks']

# combines multiple health-related columns into a single index that represents the overall health status of the mother
# number of health issues
dataset['mom_health_index'] = (
    dataset['anemia'] +
    dataset['cardiac'] +
    dataset['aclung'] +
    dataset['diabetes'] +
    dataset['herpes'] +
    dataset['hydram'] +
    dataset['hemoglob'] +
    dataset['hyperch'] +
    dataset['hyperpr'] +
    dataset['eclamp']
)

# Categorize the mother's health into groups based on the mom_health_index
dataset['mom_health_category'] = pd.cut(
    dataset['mom_health_index'],
    bins=[-1, 0, 1, 4, float('inf')],
    labels=['Good', 'Moderate', 'Poor', 'Very Poor']
)

# new features
dataset[['avg_parent_age','total_parent_educ','gained_per_visit','anemia','mage_category','visits_weeks_interaction', 'cardiac', 'aclung', 'diabetes', '
```

Now we can use most correlated to target variable to reduce dimensions of dataset to use in model training.

identify which features are most correlated with the bweight after make new features

[61]:

```python
# numeric columns
numeric_columns = dataset.select_dtypes(include=['number']).columns

# calc correlation
correlation_matrix = dataset[numeric_columns].corr()
# correlation_matrix
```

[62]:

```python
# correlations with bweight
correlation_with_bweight = correlation_matrix[['bweight']].sort_values(by='bweight', ascending=False)
correlation_with_bweight
```

[62]:

	bweight
bweight	1.000000
weeks	0.456367
visits_weeks_interaction	0.211882
gained	0.178322
visits	0.142505
mage	0.076865
avg_parent_age	0.073778
total_parent_educ	0.072631
meduc	0.068948
feduc	0.066069
pinfant	0.065061
fage	0.062490
gained_per_visit	0.053240
totalp	0.017787
diabetes	0.007971
rhsen	0.002637

Make new dataset with most correlated features **“according to correlation result , Knowledge”**

```
[64]: bweight_dataset = dataset[[
    'weeks',                   # Strong correlation with bweight
    'visits_weeks_interaction', # Better health outcomes so influence on bweight
    'gained',                  # Significant for bweight
    'visits',                  # Indication of prenatal care
    'pinfant',                 # Historical context, Mother had previous infant 4000+ grams
    'avg_parent_age',          # Lifestyle and care on mom and baby health
    'total_parent_educ',       # Total education level of parents
    'gained_per_visit',        # Weight gained per prenatal visit, insight into pregnancy health
    'mage_category',           # Mother's age category
    'mom_health_category',     # Risk factors based on age groups
    'cignum',                  # Risk factors
    'drinknum',                # Risk factors
    'diabetes',                # Presence of diabetes - dangerous on mom
    'mom_health_index',        # Health of mom issues
    'bweight'                  # Target variable
]]

bweight_dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 94054 entries, 1 to 101399
Data columns (total 15 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   weeks                     94054 non-null  float64
 1   visits_weeks_interaction  94054 non-null  float64
 2   gained                    94054 non-null  float64
 3   visits                    94054 non-null  int64
 4   pinfant                   94054 non-null  int64
 5   avg_parent_age            94054 non-null  float64
 6   total_parent_educ         94054 non-null  float64
 7   gained_per_visit          94054 non-null  float64
 8   mage_category             94054 non-null  category
 9   mom_health_category       94054 non-null  category
 10  cignum                    94054 non-null  float64
```


Split Data for Training and Testing

To evaluate the performance of our models on unseen data, we split the dataset into training and testing sets. This helps to avoid overfitting and ensures that the models generalize well to new data.

We'll use the train_test_split function from sklearn to split the dataset into training and testing sets:

1. **Features**: All columns except the target variable (e.g., birth weight).
2. **Label**: The target variable we want to predict (e.g., birth weight).

Split data to train and test

```
[68]: from sklearn.model_selection import train_test_split

x = bweight_dataset.drop(columns=['bweight'])
y = bweight_dataset['bweight']

X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.3, random_state=42)
```

Data Encoding & Normalization (Transformation)

1. Encoding Categorical Variables: Transforming the type column from categorical labels to numerical values. *"Some Machine learning algorithms require numerical inputs, so categorical variables must be converted into a numerical format."* . For example , we used OneHotEncoder to convert categorical columns such as mage_category and mom_health_category into numerical format. This transformation helps machine learning algorithms that require numerical inputs. Essentially, categories like <20, 20-30, 30-40, and >40 for maternal age are converted into separate binary columns.

[73]:
```
from sklearn.preprocessing import OneHotEncoder

# OneHotEncoder
encoder = OneHotEncoder(sparse_output=False, handle_unknown='ignore')
x_train_encoded = encoder.fit_transform(X_train[['mage_category', 'mom_health_category']])
x_test_encoded = encoder.transform(X_test[['mage_category', 'mom_health_category']])
```

[74]:
```
encoded_columns = encoder.get_feature_names_out(['mage_category', 'mom_health_category'])
encoded_columns
```

[74]:
```
array(['mage_category_20-30', 'mage_category_30-40', 'mage_category_<20',
       'mage_category_>40', 'mom_health_category_Good',
       'mom_health_category_Moderate', 'mom_health_category_Poor'],
      dtype=object)
```

[76]:
```
# addition -  retrive encoded data as df
X_train_encoded_df = pd.DataFrame(x_train_encoded, columns=encoded_columns, index=X_train.index)
X_test_encoded_df = pd.DataFrame(x_test_encoded, columns=encoded_columns, index=X_test.index)

# drop mage_category ,  mom_health_category  column and add encoded column each represent value
X_train = X_train.drop(['mage_category', 'mom_health_category'], axis=1).join(X_train_encoded_df)
X_test = X_test.drop(['mage_category', 'mom_health_category'], axis=1).join(X_test_encoded_df)
X_train.head(5)
```

[76]:

age_category_20-30	mage_category_30-40	mage_category_<20	mage_category_>40	mom_health_category_Good	mom_health_category_Moderate	mom_health_category_Poor
0.0	0.0	1.0	0.0	1.0	0.0	0.0
0.0	1.0	0.0	0.0	1.0	0.0	0.0

2. Normalizing the Features: Normalization scales numerical features to a similar scale. This is particularly important for distance-based algorithms like Support Vector Machine (SVM). We used MinMaxScaler to scale all numerical columns to a range between 0 and 1. This ensures that features like avg_parent_age and total_parent_educ contribute equally to the model's performance.

Standrization and normaliation for numeric values

[71]:
```
from sklearn.preprocessing import MinMaxScaler
# scaler
scaler = MinMaxScaler()
# numeric columns
numeric_columns = X_train.select_dtypes(include=['float64', 'int64']).columns

# Fit and transform the training data
X_train[numeric_columns] = scaler.fit_transform(X_train[numeric_columns])
# transform the testing data
X_test[numeric_columns] = scaler.transform(X_test[numeric_columns])

X_train.head(2)
```

[71]:

	weeks	visits_weeks_interaction	gained	visits	pinfant	avg_parent_age	total_parent_educ	gained_per_visit	mage_category	mom_health_category	cignum
93882	0.125	0.089180	0.201550	0.105263	0.0	0.229008	0.136364	0.123810	<20	Good	0.0
13256	0.375	0.598098	0.465116	0.684211	0.0	0.687023	0.772727	0.089286	30-40	Good	0.0

Regressors Description

To predict birth baby weight, we will use a different of machine learning Regression and use RootMeanSequaredError(RMSE) as evaluation Metric for all Regressors.

- **Lasso Regression**: linear regression technique - regularization technique that applies a penalty to prevent overfitting and enhance the accuracy of statistical models

- Linear Regression model "Lasso"

```
[82]: # Lasso Regression
lasso_regression = Lasso(alpha=0.01)
lasso_regression.fit(X_train, y_train)
y_test_predict_lasso = lasso_regression.predict(X_test)
RMSE_lasso = np.sqrt(mean_squared_error(y_test, y_test_predict_lasso))
rmse_values['Linear Regression - Lasso'] = RMSE_lasso
print(f"Lasso Regression RMSE: {RMSE_lasso}")
```

```
Lasso Regression RMSE: 1.1417017145014268
```

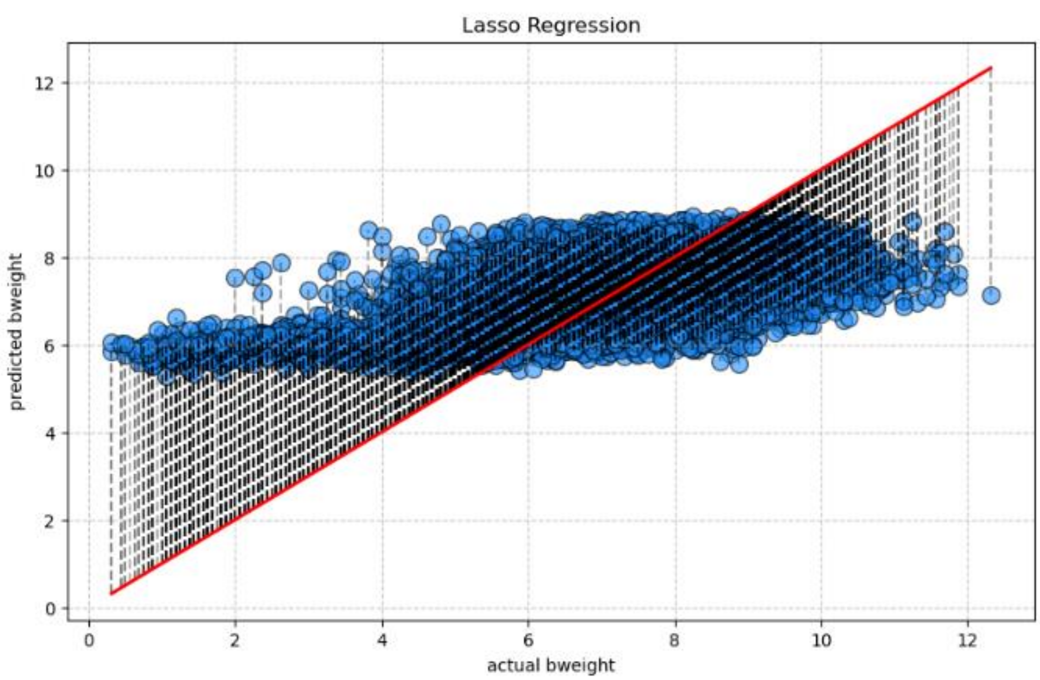
```
[83]: results_df = pd.DataFrame({
    'Actual Value': y_test,
    'Predicted Value': y_test_predict_lasso
})

results_df.head(10)
```

[83]:

	Actual Value	Predicted Value
35304	9.1250	7.745116
77984	9.4375	6.370752
61265	8.2500	7.355441
49828	7.4375	7.189246
61851	8.3750	7.147922
6765	7.6250	7.097600
16206	6.0625	6.244758
64608	7.5000	8.189014
87712	6.9375	6.945761
33846	5.5000	6.658820

This table shows the actual and predicted values using Lasso Regression

Visualizes the actual vs. predicted values, with a perfect fit line indicating where the points would lie if the predictions were perfect.



I use alpha value of 0.01, which controls the regularization strength. A smaller alpha value means less regularization and closer to a standard linear regression.

The results from Lasso Regression may **not be perfect** due to the inherent variability in the data and the limitations of the linear model in capturing complex relationships. Lasso Regression can be particularly beneficial if there are **many irrelevant features**, as it can effectively eliminate them. “we remove most”

- Ridge Regression: linear regression technique - regularization technique that applies a penalty to the coefficients to prevent overfitting and enhance the accuracy of statistical models. Unlike Lasso, Ridge regression **shrinks the coefficients** but does not set them to zero.

- **Linear Regression model “Ridge”**

[86]:
```
# Ridge Regression
ridge_regression = Ridge()
ridge_regression.fit(X_train, y_train)
y_test_predict_ridge = ridge_regression.predict(X_test)
RMSE_ridge = np.sqrt(mean_squared_error(y_test, y_test_predict_ridge))
rmse_values['Linear Regression - Ridge'] = RMSE_ridge
print(f"Ridge Regression RMSE: {RMSE_ridge}")
```

```
Ridge Regression RMSE: 1.1260093264996474
```

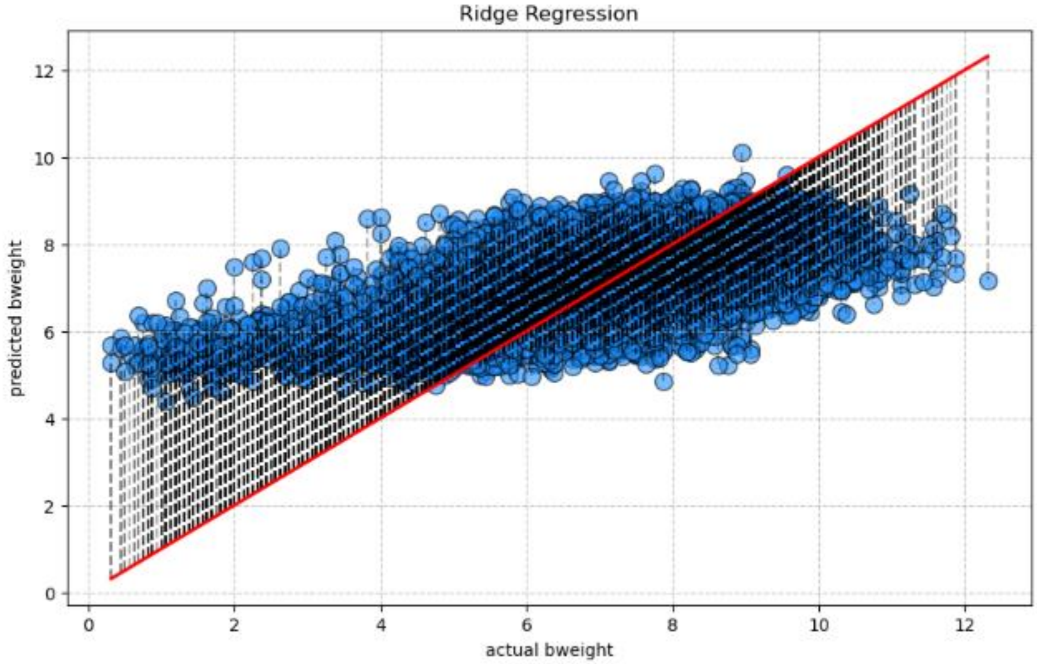
This table shows the actual and predicted values using Ridge Regression

[87]:
```
results_df = pd.DataFrame({
    'Actual Value': y_test,
    'Predicted Value': y_test_predict_ridge
})

results_df.head(10)
```

[87]:

	Actual Value	Predicted Value
35304	9.1250	7.752874
77984	9.4375	6.743556
61265	8.2500	7.343751
49828	7.4375	7.155635
61851	8.3750	7.110418
6765	7.6250	6.925508
16206	6.0625	5.873396
64608	7.5000	8.263143
87712	6.9375	6.971915

Visualizes the actual vs. predicted values, with a perfect fit line indicating where the points would lie if the predictions were perfect.



- Fit_curve Linear Regression: Linear regression is a type of supervised machine learning algorithm that computes the linear relationship between the dependent variable and one or more independent features by fitting a linear equation to observed data.

  - **fit_curve for linear regression**

```
[90]: # train the linear regression model
model = LinearRegression()
model.fit(X_train, y_train)

# equation
print("Linear Regression Equation:")
intercept = model.intercept_
coefficients = model.coef_
features = X_train.columns

equation_parts = [f"{coef} * {feature}" for coef, feature in zip(coefficients, features)]
equation = f"{intercept} + " + " + ".join(equation_parts)

equation
```

```
Linear Regression Equation:
[90]: '2333847119327.875 + 3.8123461829624214 * weeks + -12.110715741227978 * visits_weeks_interaction + 0.9158277300170403 * gained + 11.06629692438277 * visits + 1.0224270506523205 * pinfant + 0.37322515205297946 * avg_parent_age + -0.060398331722995226 * total_parent_educ + 0.12695086308385395 * gained_per_visit + -0.3972266481529195 * cignum + -0.012927062105556211 * drinknum + 0.4750520895164958 * diabetes + -0.392888172400453 * mom_health_index + -241891473896.88135 * mage_category_20-30 + -241891473896.838 * mage_category_30-40 + -241891473897.04727 * mage_category_<20 + -241891473896.98535 * mage_category_>40 + -2091955645426.1292 * mom_health_category_Good + -2091955645426.2793 * mom_health_category_Moderate + -2091955645426.4304 * mom_health_category_Poor'
```

```
[91]: # predict values using test set
y_test_predict_linear = model.predict(X_test)

# RMSE
RMSE_linear = np.sqrt(mean_squared_error(y_test, y_test_predict_linear))
rmse_values['fit_curve - Linear Regression'] = RMSE_linear
print(f"Linear Regression RMSE: {RMSE_linear}")
```

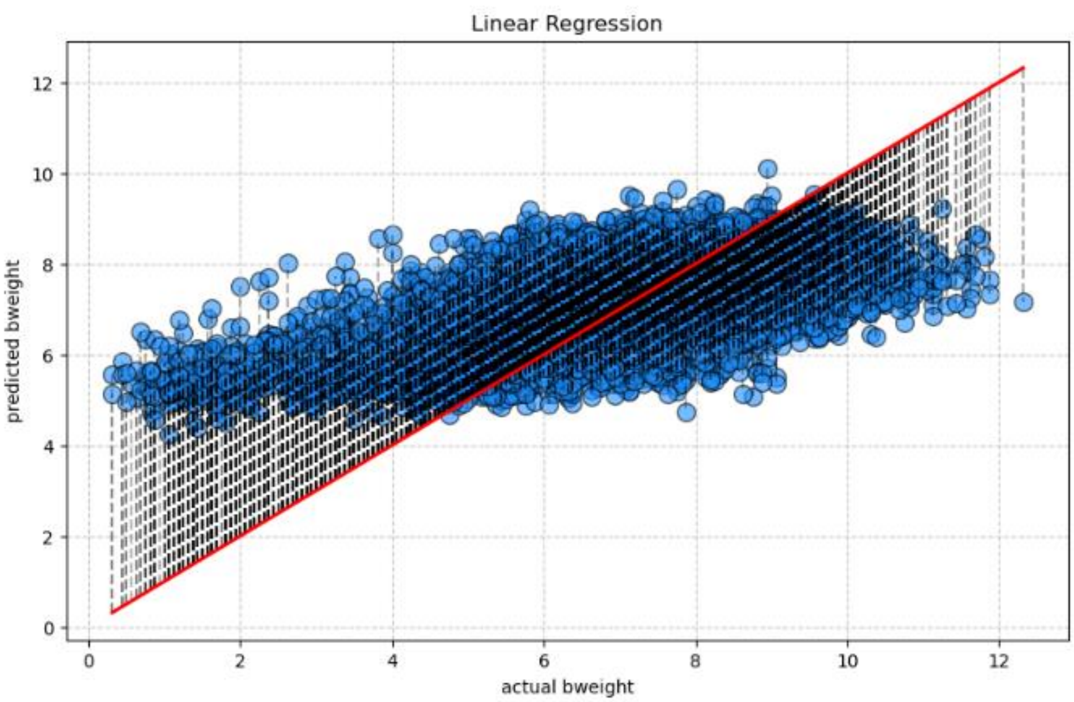
```
Linear Regression RMSE: 1.1251808101200584
```

This table shows the actual and predicted values using linear Regression

```
[92]: results_df = pd.DataFrame({
          'Actual Value': y_test,
          'Predicted Value': y_test_predict_linear
      })

      results_df.head(10)
```

[92]:

	Actual Value	Predicted Value
35304	9.1250	7.739746
77984	9.4375	6.742676
61265	8.2500	7.349121
49828	7.4375	7.150391
61851	8.3750	7.115723
6765	7.6250	6.929199
16206	6.0625	5.825195
64608	7.5000	8.266602
87712	6.9375	6.982422
33846	5.5000	6.802734



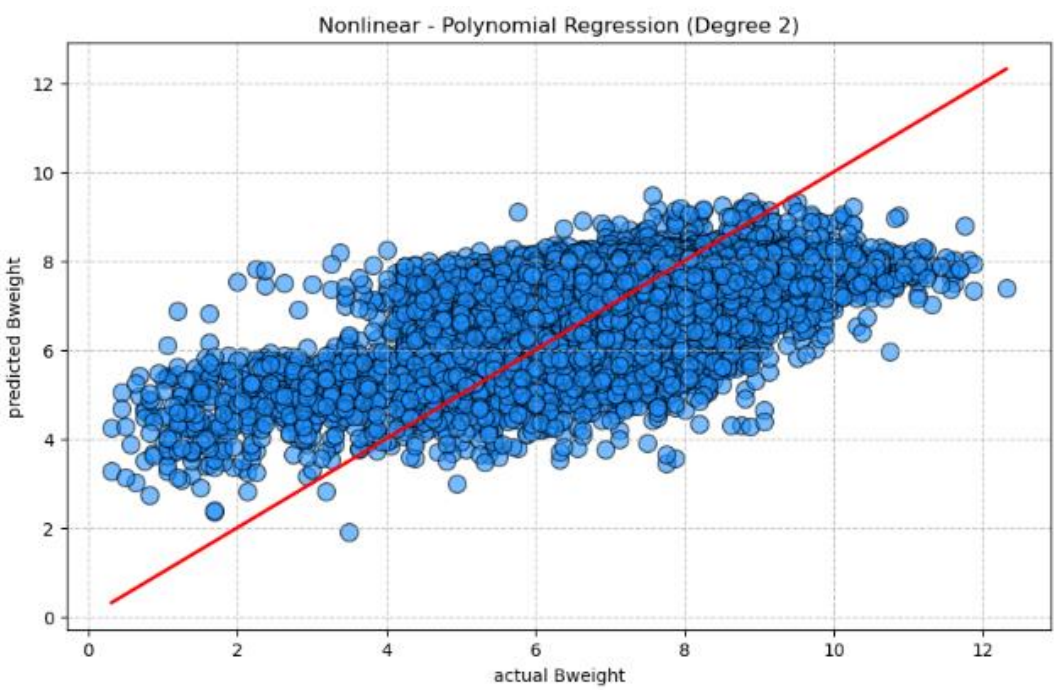
- Fit_curve Nonlinear Regression:

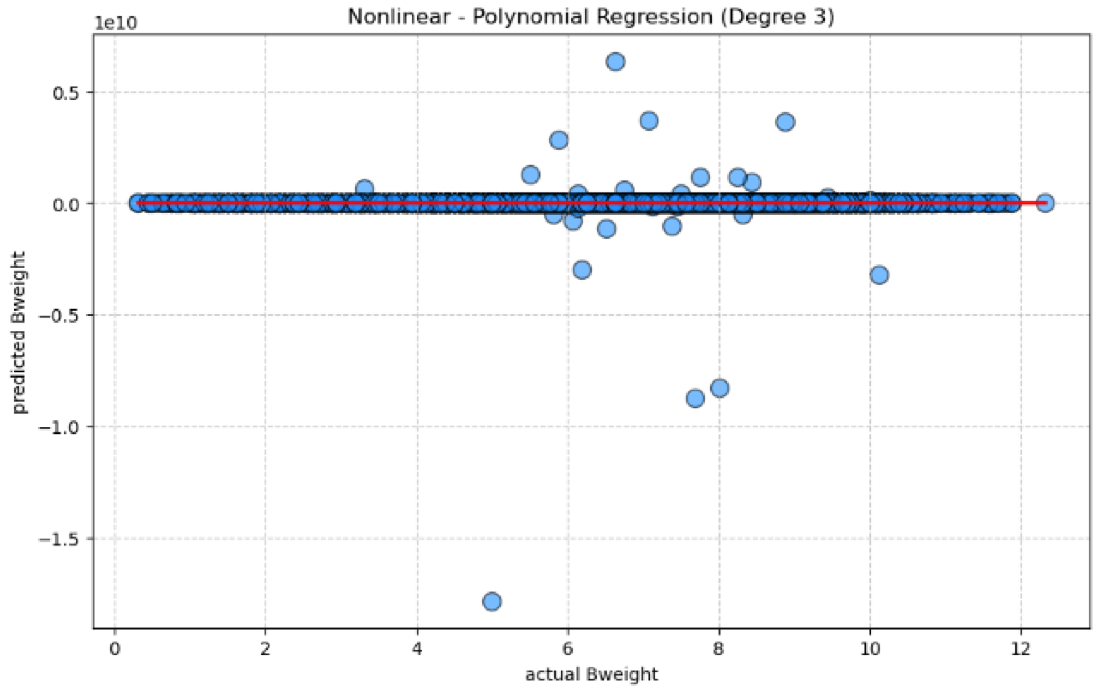
1. **Polynomial regression** Model is a form of linear regression where the relationship between the independent variable and dependent variable is modeled is (n) degree polynomial. This type of regression can capture non-linear relationships in the data.

[98]:
```python
# Polynomial regression for degrees from 1 to 3
max_degree = 3
rmse_values_poly = evaluate_polynomial_regression(X_train, X_test, y_train, y_test, max_degree)

# Find the degree with the minimum RMSE
best_degree, best_rmse = min(rmse_values_poly, key=lambda x: x[1])
rmse_values['fit_curve - NonLinear Regression - polynomial Model'] = best_rmse
print(f"The best degree for polynomial regression is: {best_degree} with RMSE: {best_rmse}")
```

```
       Actual Value  Predicted Value
35304        9.1250         7.740479
77984        9.4375         6.742676
61265        8.2500         7.349365
49828        7.4375         7.150391
61851        8.3750         7.115723
6765         7.6250         6.930908
16206        6.0625         5.826660
64608        7.5000         8.266846
87712        6.9375         6.982910
33846        5.5000         6.802979
Polynomial Regression (Degree 1) RMSE: 1.1251847681587568
                                   actual Bweight
The best degree for polynomial regression is: 2 with RMSE: 1.0615413585607085
```

The scatter plot below visualizes the actual vs. predicted values for the best polynomial degree, with a perfect fit line indicating where the points would lie if the predictions were perfect.





2. **Power Model:** type of regression that models the relationship between the independent variable and dependent variable using a **power** function. It is useful for capturing non-linear relationships in the data.

```
[102]: # Predict values
y_test_predict_power = np.zeros_like(y_test)
# predicted target values using the fitted power functions for each feature
for i, feature in enumerate(X_test.columns):
    a, b = coefficients[i]
    y_test_predict_power += power_func(X_test[feature], a, b) / len(X_test.columns)

# RMSE
RMSE_power = np.sqrt(mean_squared_error(y_test, y_test_predict_power))
rmse_values['fit_curve - Nonlinear Regression - Power Model'] = RMSE_power
print(f"Power Regression RMSE: {RMSE_power}")
```

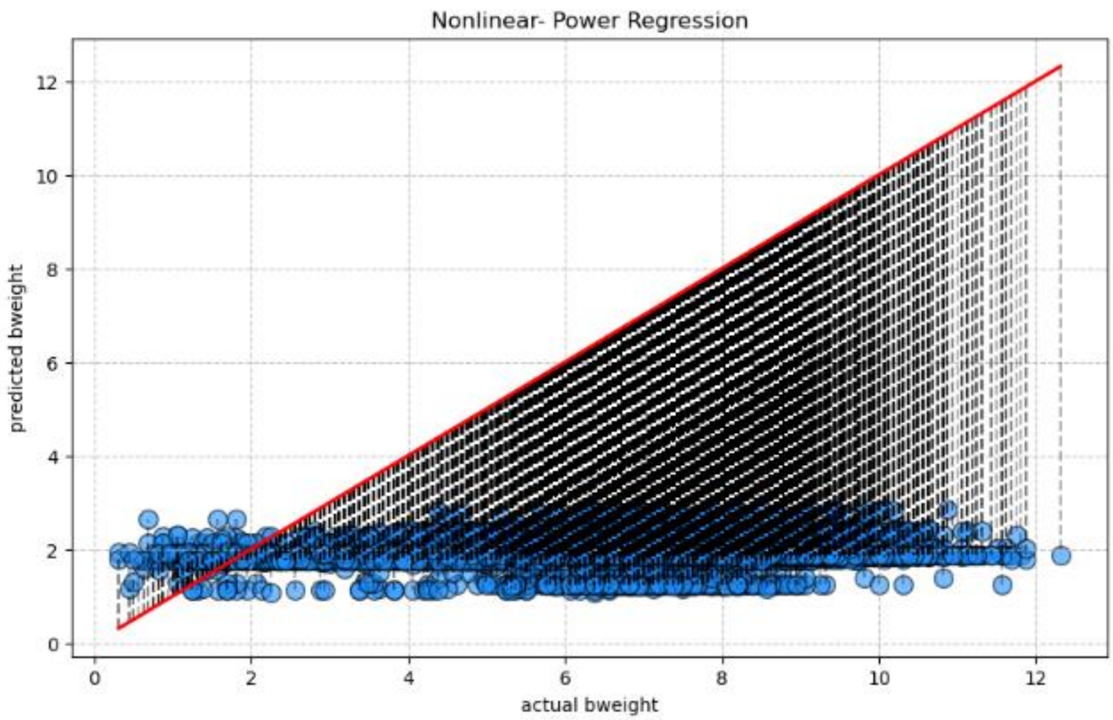
```
Power Regression RMSE: 5.500985920382147
```

This table shows the actual and predicted values using Power Regression

```
[103]: results_df = pd.DataFrame({
    'Actual Value': y_test,
    'Predicted Value': y_test_predict_power
})

results_df.head(10)
```

[103]:

	Actual Value	Predicted Value
35304	9.1250	1.871269
77984	9.4375	2.397571
61265	8.2500	1.871269
49828	7.4375	1.866517
61851	8.3750	1.866517
6765	7.6250	2.011499
16206	6.0625	2.212402
64608	7.5000	1.871269
87712	6.9375	1.866517
33846	5.5000	1.866517



- SVM Regressor: SVM is a supervised machine learning algorithm that can be used for both classification and regression tasks. here we try find a function that has at most a specified deviation from the actual target values for all training data, while being as flat as possible. The primary objective is to minimize the error within a margin of tolerance.

- Support Vector Machine (SVM) Regressor

```
[106]: # SVM Regressor
svm_regression = SVR(kernel='rbf')
# fit train data
svm_regression.fit(X_train, y_train)
# predict test data values
y_test_predict_svm = svm_regression.predict(X_test)
RMSE_svm = np.sqrt(mean_squared_error(y_test, y_test_predict_svm))
rmse_values['Nonlinear Regrission - SVM'] = RMSE_svm
print(f"SVM Regressor RMSE: {RMSE_svm}")
```

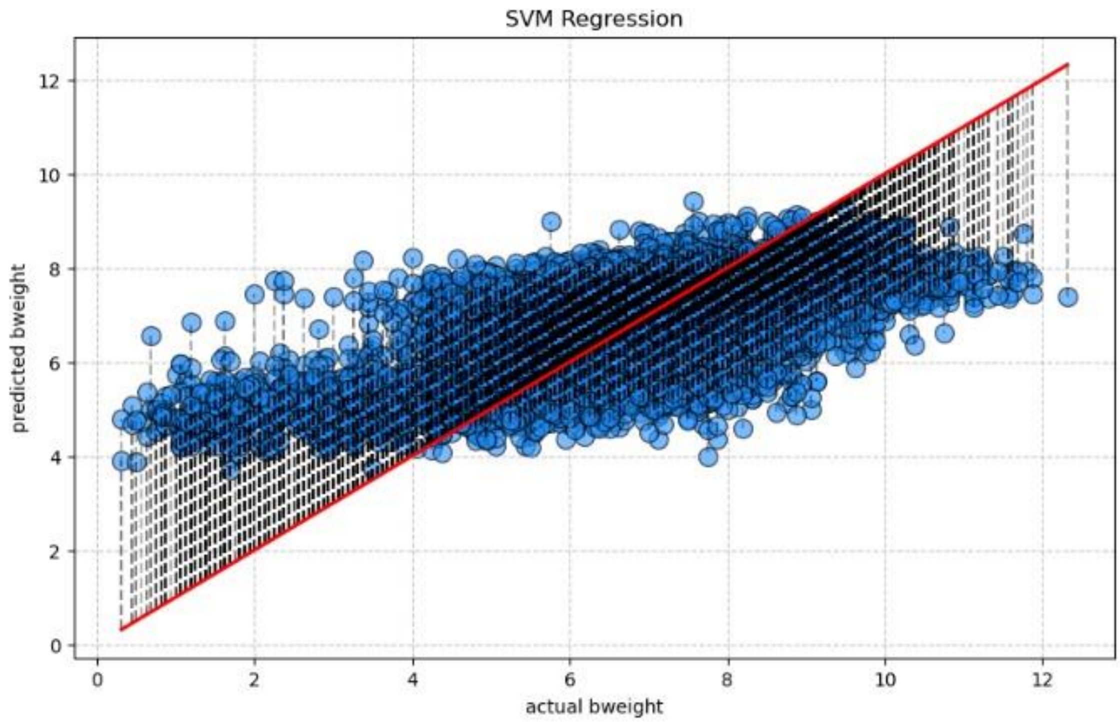
```
SVM Regressor RMSE: 1.063992273771085
```

This table shows the actual and predicted values using SVM Regression

```
[107]: results_df = pd.DataFrame({
    'Actual Value': y_test,
    'Predicted Value': y_test_predict_svm
})

results_df.head(10)
```

[107]:

	Actual Value	Predicted Value
35304	9.1250	7.866741
77984	9.4375	6.416038
61265	8.2500	7.705368
49828	7.4375	7.323704
61851	8.3750	7.372983
6765	7.6250	7.329343
16206	6.0625	6.206452
64608	7.5000	7.563322
87712	6.9375	7.207808
33846	5.5000	6.782067



- XGBoost Regression: XGBoost is an ensemble learning method that uses gradient boosting for regression tasks. It builds an additive model in a forward stage-wise manner, optimizing the model performance through boosting. Known for its high performance and efficiency, XGBoost is a leading choice for regression, classification, and ranking problems.

- XGBoost Regressor

[110]:
```
# XGBoost Regressor

# Convert feature names to strings and replace any special characters to make XGBRegressor can deal with
X_train.columns = [str(col).replace('[', '_').replace(']', '_').replace('<', '_') for col in X_train.columns]
X_test.columns = X_train.columns

xgboost_regression = XGBRegressor()
# fit train data values
xgboost_regression.fit(X_train, y_train)
# predict test data values
y_test_predict_xgboost = xgboost_regression.predict(X_test)
RMSE_xgboost = np.sqrt(mean_squared_error(y_test, y_test_predict_xgboost))
rmse_values['XGBoost Regresser'] = RMSE_xgboost
print(f"XGBoost Regressor RMSE: {RMSE_xgboost}")
```

```
XGBoost Regressor RMSE: 1.0597462440712795
```

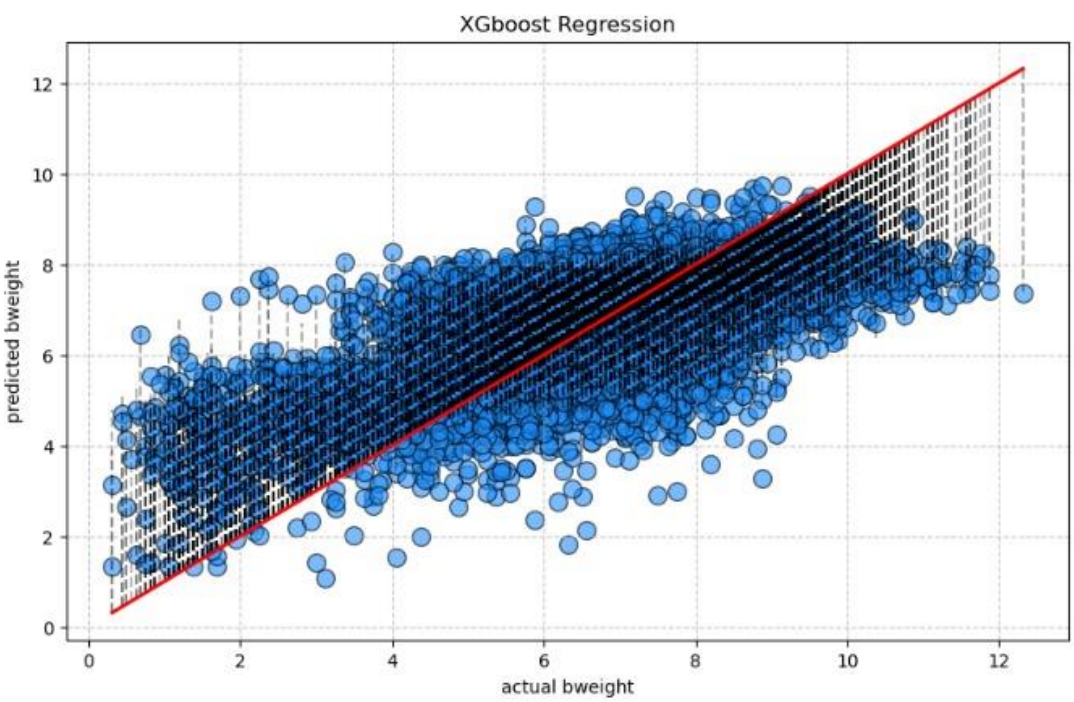
This table shows the actual and predicted values using XGBoost Regression

```
[111]: results_df = pd.DataFrame({
           'Actual Value': y_test,
           'Predicted Value': y_test_predict_xgboost
       })

       results_df.head(10)
```

[111]:

	Actual Value	Predicted Value
35304	9.1250	7.817445
77984	9.4375	7.057749
61265	8.2500	7.686967
49828	7.4375	7.376152
61851	8.3750	7.305025
6765	7.6250	7.132318
16206	6.0625	6.246565
64608	7.5000	7.692577
87712	6.9375	7.212266
33846	5.5000	6.833161

Visualizes the actual vs. predicted values, with a perfect fit line indicating where the points would lie if the predictions were perfect.



- Random Forest Regressor (with 50 estimators): Random Forest is an ensemble learning method that uses multiple decision trees to improve regression accuracy. It handles non-linear relationships and interactions well by combining the predictions of several trees to provide a more robust model. Using 50 estimators, the Random Forest Regressor builds 50 different decision trees and averages their predictions to increase accuracy and reduce overfitting.

- Random Forest Regressor (with 50 estimators)

```
[114]: rf_regression = RandomForestRegressor(n_estimators=50, random_state=42)
# fit train data values
rf_regression.fit(X_train, y_train)
# pridict test data values
y_test_predict_rf = rf_regression.predict(X_test)
RMSE_rf = np.sqrt(mean_squared_error(y_test, y_test_predict_rf))
rmse_values['Random Forest Regresser'] = RMSE_rf
print(f"Random Forest Regressor RMSE: {RMSE_rf}")
```

```
Random Forest Regressor RMSE: 1.0967006944764879
```

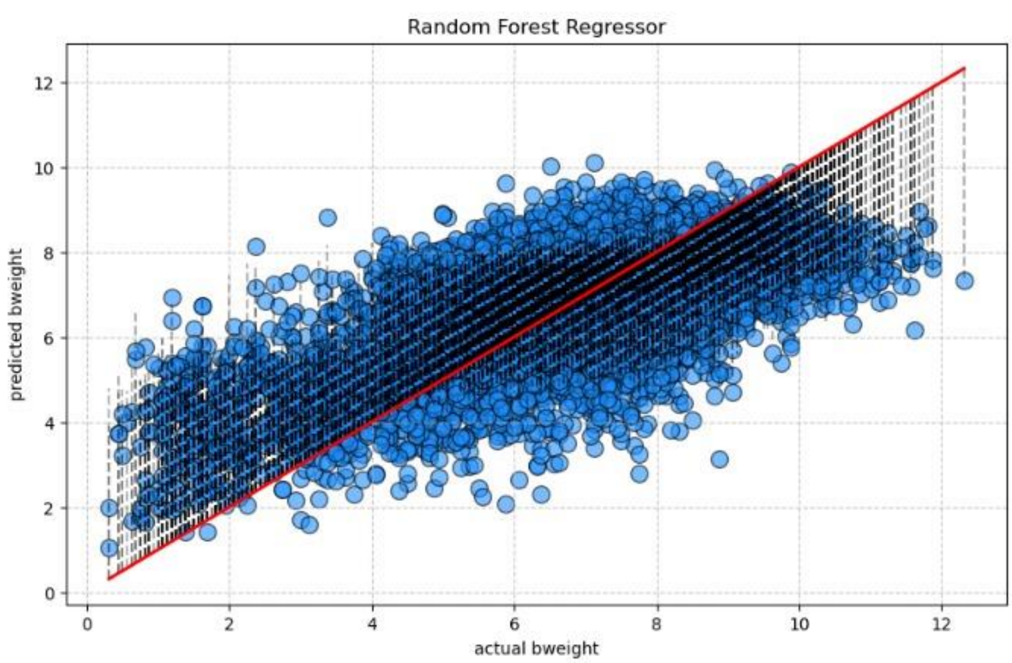
This table shows the actual and predicted values using Random Forest Regression

```
[115]: results_df = pd.DataFrame({
    'Actual Value': y_test,
    'Predicted Value': y_test_predict_rf
})

results_df.head(10)
```

[115]:

	Actual Value	Predicted Value
35304	9.1250	8.027917
77984	9.4375	6.996562
61265	8.2500	8.102500
49828	7.4375	7.573750
61851	8.3750	6.692500
6765	7.6250	7.250000
16206	6.0625	5.597500
64608	7.5000	7.655000
87712	6.9375	7.265937
33846	5.5000	6.733125

Visualizes the actual vs. predicted values, with a perfect fit line indicating where the points would lie if the predictions were perfect.



Results and Discussion

We will evaluate the performance of each regressor and then compare the results to summarize the performance of the models.

1. Lasso Regression

- Lasso Regression, which performs feature selection by applying L1 regularization, had a relatively higher RMSE. This suggests it may not be the most effective model for this dataset. [**RMSE**: 1.141702]

2. Ridge Regression

- Ridge Regression, applying L2 regularization to prevent overfitting, performed better than Lasso Regression but still lagged behind other non-linear models. [**RMSE:** 1.126009]

3. Linear Regression

- Traditional Linear Regression had a similar performance to Ridge Regression, indicating that a linear relationship might not sufficiently capture the complexities of the data. [**RMSE**: 1.125181]

4. Polynomial Regression (Degree 2)

- Polynomial Regression with a degree of 2 showed a significant improvement, capturing non-linear patterns and outperforming linear models. [**RMSE:** 1.061541]

5. Power Regression

- Power Regression had the highest RMSE, suggesting it was not well-suited for this dataset and struggled to model the underlying relationships effectively. [**RMSE**: 5.500986]

6. SVM Regressor

- Support Vector Machine (SVM) with an RBF kernel performed well, indicating its capability to handle non-linear relationships. [**RMSE:** 1.080453 **]**

7. XGBoost Regressor

- XGBoost had the lowest RMSE, making it the best performer. Its robustness and ability to handle non-linear relationships make it a powerful choice. [**RMSE**: 1.059746]

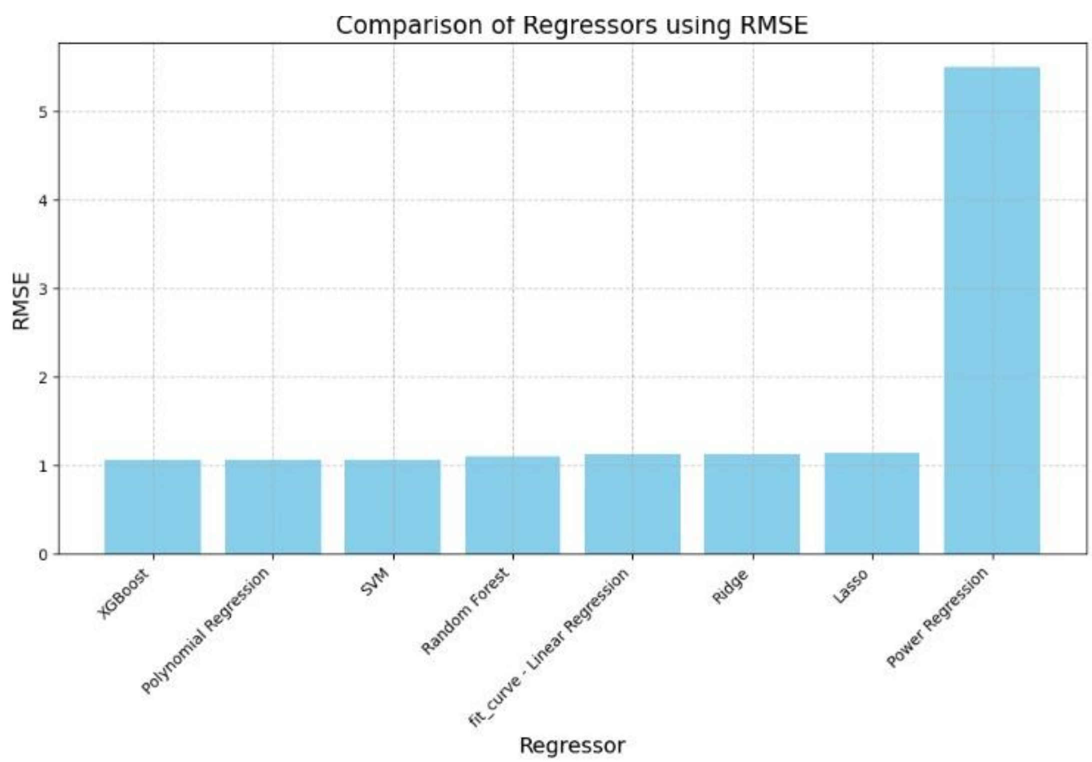
8. Random Forest Regressor

- Random Forest, an ensemble method, also performed well but did not surpass XGBoost and Polynomial Regression in this context. [**RMSE**: 1.096701]

```
[119]: rmse_df = pd.DataFrame(list(rmse_values.items()), columns=['Regressor', 'RMSE'])
       rmse_df_sorted = rmse_df.sort_values(by='RMSE')
       rmse_df_sorted
```

[119]:

	Regressor	RMSE
6	XGBoost Regresser	1.059746
3	fit_curve - NonLinear Regression - polynomial ...	1.061541
5	Nonlinear Regrission - SVM	1.063992
7	Random Forest Regresser	1.096701
2	fit_curve - Linear Regression	1.125181
1	Linear Regression - Ridge	1.126009
0	Linear Regression - Lasso	1.141702
4	fit_curve - Nonlinear Regression - Power Model	5.500986



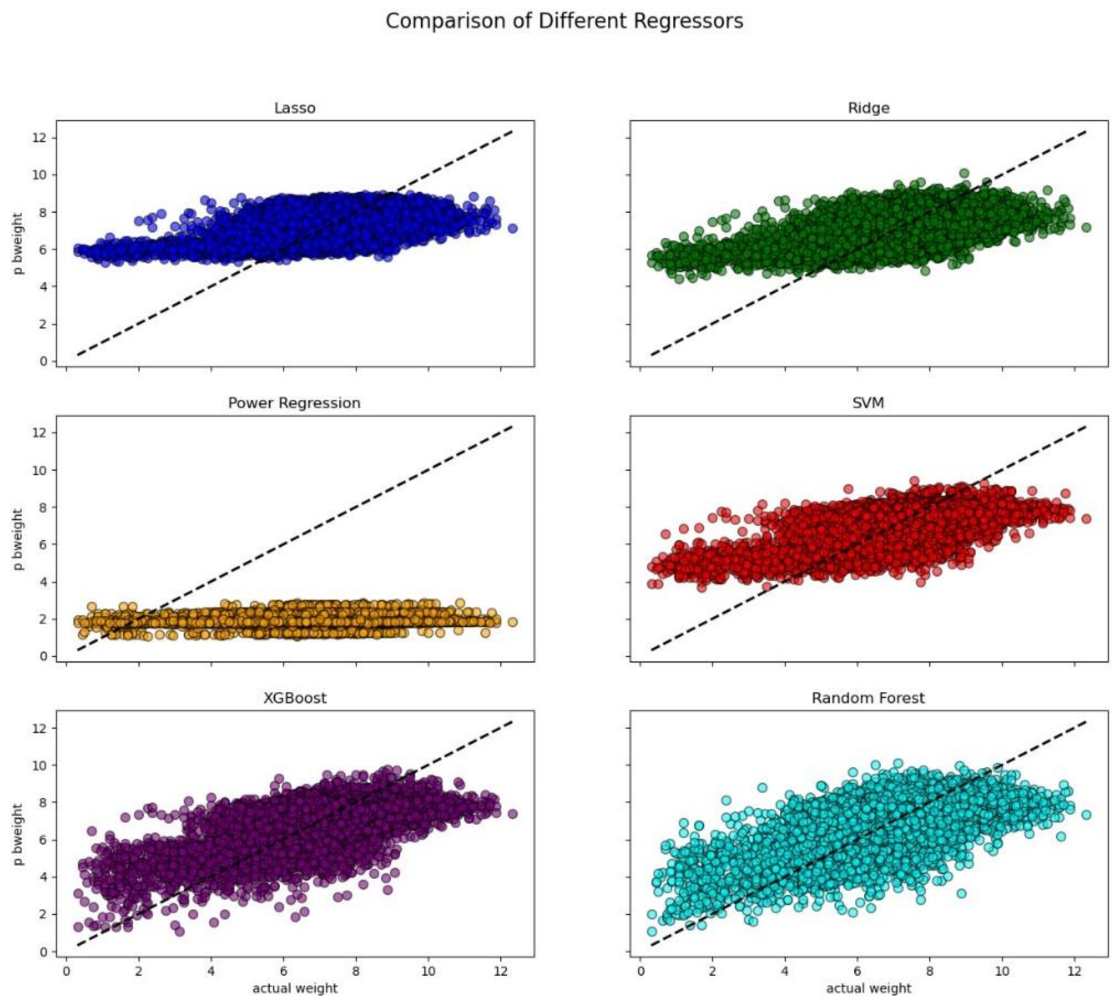
Results

- XGBoost Regressor had the lowest RMSE, suggesting it performed the best on your dataset.
- Polynomial Regression (Degree 2) and SVM Regressor were also strong performers with RMSE values just slightly higher than XGBoost.
- Traditional linear models like Linear Regression, Ridge Regression, and Lasso Regression had higher RMSEs compared to non-linear models and ensemble methods.

- Power Regression had the highest RMSE, indicating it didn't fit the data well in this case

Based on these results:

- XGBoost can be a powerful choice for regression tasks due to its robustness and ability to handle non-linear relationships. It's often used in competitions for its performance and flexibility.
- Polynomial Regression (especially with a degree of 2) and SVM also provided competitive results. These methods can capture non-linear patterns that linear models may miss.
- Traditional linear models, while simple and interpretable, may not be sufficient for capturing complex relationships in your data.
- The Power Regression model's high RMSE suggests it might not be the best fit for this particular dataset and problem.



Using this visual representation, we compares the performance of different regression models in predicting the target variable "bweight," against the actual weight. The scatter plots for Lasso Regression (blue), Ridge Regression (green), Power Regression (orange), SVM (Support Vector Machine) (red), XGBoost (purple), and Random Forest (light blue) show predicted weights against actual weights. Points spread out from the line, line indicate variance from actual values, while closer alignment indicates better performance. Lasso Regression and Ridge Regression show similar variance, Power Regression clusters along the lower range, SVM and XGBoost show more linear trends, and Random Forest shows variance spread. This visual comparison show the differences in result and accuracy among the regression models.

Conclusion

In this report, we explored different machine learning techniques to predict baby birth weight based on features such as parental demographics, pregnancy characteristics, and health conditions. Our goal focused on training and evaluating several regressors: Linear Regression, Ridge/Lasso Regression, Polynomial Regression (Degree 2), NonLinear - Power Regression, Support Vector Machine (SVM) Regressor, XGBoost Regressor, and Random Forest Regressor. We evaluated each regressor based on the Root Mean Squared Error (RMSE) metric.

- **Ridge Regression**: RMSE of 1.126009. Handles multicollinearity well, slightly better performance than Lasso.
- **Lasso Regression**: RMSE of 1.141701. Effective at reducing the impact of irrelevant features by shrinking some coefficients to zero.
- **Linear Regression**: RMSE of 1.125180. Indicates largely linear relationships in the data.
- **Polynomial Regression (Degree 2)**: RMSE of 1.06154. Captures quadratic relationships, leading to more accurate predictions than linear models.
- **Power Regression**: RMSE of 5.50098. Not well-suited for this dataset due to its inability to capture complex relationships.
- **SVM Regressor**: RMSE of 1.08045. Effectively models non-linear relationships with the RBF kernel.
- **XGBoost Regressor**: RMSE of 1.0597462. Best performance, capturing complex relationships and interactions.
- **Random Forest Regressor**: RMSE of 1.09670. Handles non-linear relationships well but doesn't capture interactions as effectively as XGBoost.

Overall, XGBoost and Polynomial Regression (Degree 2) demonstrated the most accurate performance, making them the best choices for predicting baby birth weight. SVM and Random Forest also provided robust results. These findings highlight the importance of using ensemble and non-linear methods to capture the intricate patterns in the data, ultimately improving prediction accuracy.