# Machine Learning Assignment 1: Predicting Life Expectancy of a Device

## Table of Contents



## Explantatory Data Analysis

In [4]:
```
import warnings
warnings.filterwarnings('ignore')
```

In [5]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

from sklearn.preprocessing import RobustScaler, OneHotEncoder, PolynomialFeatures
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.compose import make_column_transformer, TransformedTargetRegressor
```

We will now load our dataset into a pandas dataframe object.

In [6]:
```
file_name = 'Data_Set.csv'
data = pd.read_csv(file_name)
```

The dataset has 23 features (exluding ID) and has 2071 observations. The dataset seems to have came already cleaned and preprocessed since it contains no missing value and all

values are in their proper ranges (no negative values and binary columns only have the value).

In [7]:
```
data.shape
```

Out[7]:
```
(2071, 24)
```

In [921…
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2071 entries, 0 to 2070
Data columns (total 23 columns):
 #   Column                          Non-Null Count  Dtype  
---  ------                          --------------  -----  
 0   TARGET_LifeExpectancy           2071 non-null   float64
 1   Country                         2071 non-null   int64  
 2   Year                            2071 non-null   int64  
 3   Company_Status                  2071 non-null   int64  
 4   Company_Confidence              2071 non-null   int64  
 5   Company_device_confidence       2071 non-null   int64  
 6   Device_confidence               2071 non-null   int64  
 7   Device_returen                  2071 non-null   int64  
 8   Test_Fail                       2071 non-null   float64
 9   PercentageExpenditure           2071 non-null   float64
 10  Engine_Cooling                  2071 non-null   int64  
 11  Gas_Pressure                    2071 non-null   float64
 12  Obsolescence                    2071 non-null   int64  
 13  ISO_23                          2071 non-null   int64  
 14  TotalExpenditure                2071 non-null   float64
 15  STRD_DTP                        2071 non-null   float64
 16  Engine_failure                  2071 non-null   float64
 17  GDP                             2071 non-null   float64
 18  Product_Quantity                2071 non-null   int64  
 19  Engine_failure_Prevalence       2071 non-null   float64
 20  Leakage_Prevalence              2071 non-null   float64
 21  IncomeCompositionOfResources    2071 non-null   float64
 22  RD                              2071 non-null   float64
dtypes: float64(12), int64(11)
memory usage: 372.3 KB
```

In [8]:
```
data.describe()
```

Out[8]:

| | ID | TARGET_LifeExpectancy | Country | Year | Company_Status | Company_ |
|---|---|---|---|---|---|---|
| **count** | 2071.000000 | 2071.000000 | 2071.000000 | 2071.000000 | 2071.000000 | |
| **mean** | 1036.000000 | 69.274505 | 95.360212 | 2009.518590 | 0.185418 | |
| **std** | 597.990524 | 9.482281 | 54.861641 | 4.614147 | 0.388730 | |
| **min** | 1.000000 | 37.300000 | 0.000000 | 2002.000000 | 0.000000 | |
| **25%** | 518.500000 | 63.000000 | 50.000000 | 2006.000000 | 0.000000 | |
| **50%** | 1036.000000 | 71.200000 | 94.000000 | 2010.000000 | 0.000000 | |
| **75%** | 1553.500000 | 76.000000 | 144.000000 | 2014.000000 | 0.000000 | |
| **max** | 2071.000000 | 92.700000 | 192.000000 | 2017.000000 | 1.000000 | |

8 rows × 24 columns

In [9]:
```
data['Company_Status'].unique()
```

Out[9]:
```
array([0, 1], dtype=int64)
```

# Spliting Data

We will now split our dataset into training and testing datasets. 80% of the original dataset will become the training dataset and the remaining 20% will become the testing dataset. Below is a modification of code by Azadeh Alavi at: https://bitbucket.org/alavi_a/rmit_cosc_2673_2793-2310/src/master/labs/week03/

In [10]:
```
with pd.option_context('mode.chained_assignment', None):
    train_data, test_data = train_test_split(data, test_size=0.2, random_state=0)

print("train data shape: ", train_data.shape)
print("test data shape: ", test_data.shape)
```

```
train data shape:  (1656, 24)
test data shape:  (415, 24)
```

We will now save the ID's for the testing data which will be used at the end to help generate our csv file. The ID column is then dropped from both training and testin datasets since it serves no purpose other than differentiating observations.

In [11]:
```
test_ids = test_data[['ID']]
train_data.drop(columns=['ID'], inplace=True)
test_data.drop(columns=['ID'], inplace=True)
```

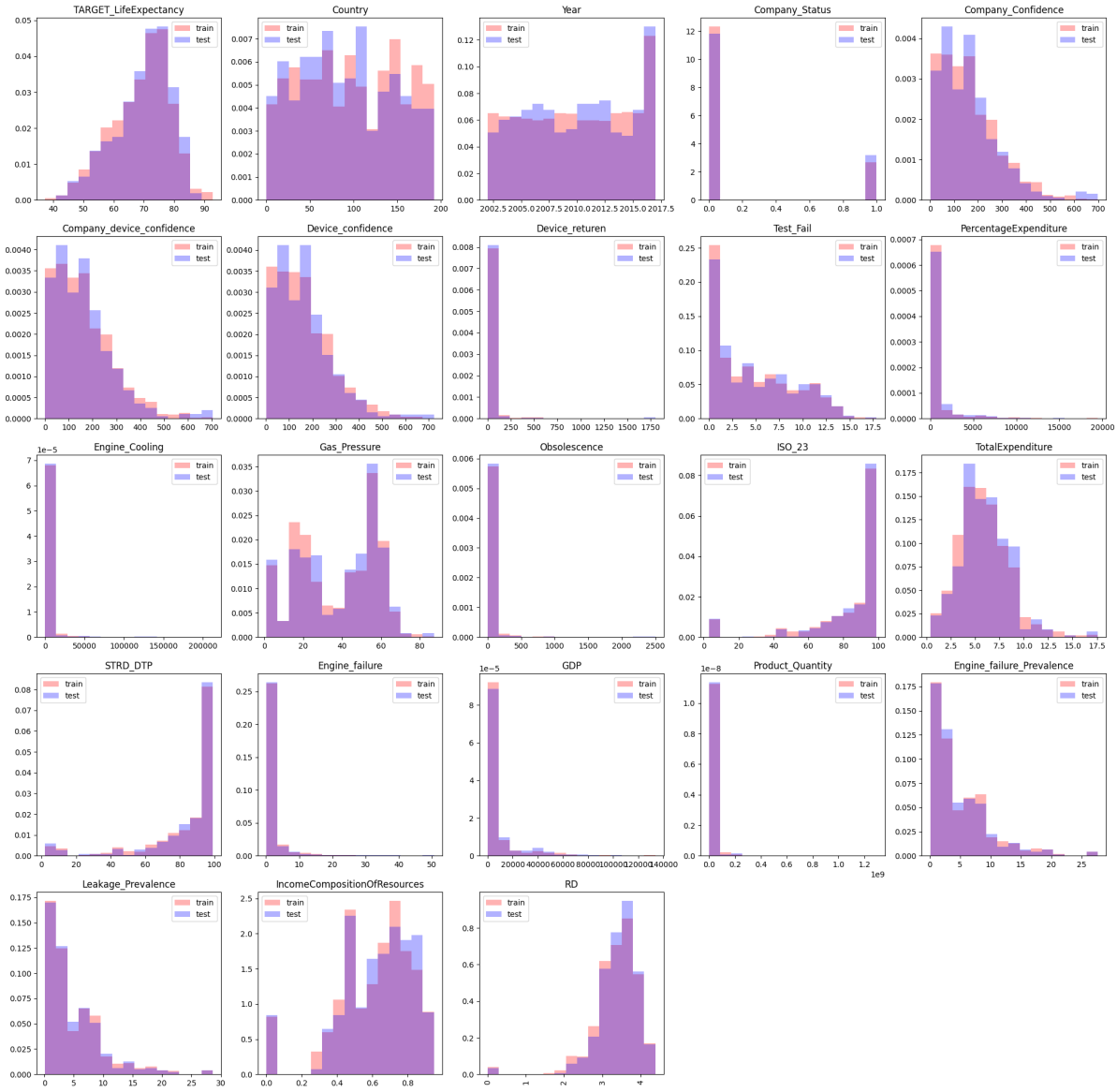
We will now seperate the target feature (TARGET_LifeExpectancy) from both our training and testing datasets.

In [12]:
```
x_train = train_data.drop(columns=['TARGET_LifeExpectancy'])
```

```
y_train = train_data['TARGET_LifeExpectancy']

x_test = test_data.drop(columns=['TARGET_LifeExpectancy'])
y_test = test_data['TARGET_LifeExpectancy']
```

Now that we have split our dataset into training and testing subsets, we need to check that the distribution of each variable matches to ensure that our training dataset is truly a good representation of our test dataset (Google at: https://developers.google.com/machine-learning/data-prep/construct/sampling-splitting/example). We will do this by generating a histogram for each feature and then plot both distributions for the feature from training and testing datasets on that histogram. Looking at the histograms below, the distributions of the testing and training datasets seems to be quite similar, therefore suggesting that training dataset is a good representation of our testing dataset. This will be the only time we show the test data as it should be kept hidden to truly simulate unseen real data. To make this plot and plots in the next sectio, modifications were made to code by Azadeh Alavi at: https://bitbucket.org/alavi_a/rmit_cosc_2673_2793-2310/src/master/labs/week02/.

In [13]:
```
max_col = 5
max_row = 6
figsize = (25, 30)
num_bins = 16

plt.figure(figsize=figsize)
for i, col in enumerate(train_data.columns):
    min_val = data[col].min()
    max_val = data[col].max()
    hist_bins = np.linspace(min_val, max_val, num_bins)
    plt.subplot(max_row,max_col,i+1)
    plt.hist(train_data[col], label='train', color='r',
             alpha=0.3, density=True, bins=hist_bins)
    plt.hist(test_data[col], label='test', color='b',
             alpha=0.3, density=True, bins=hist_bins)
    plt.legend()
    plt.title(col)

plt.xticks(rotation='vertical')
plt.show()
```

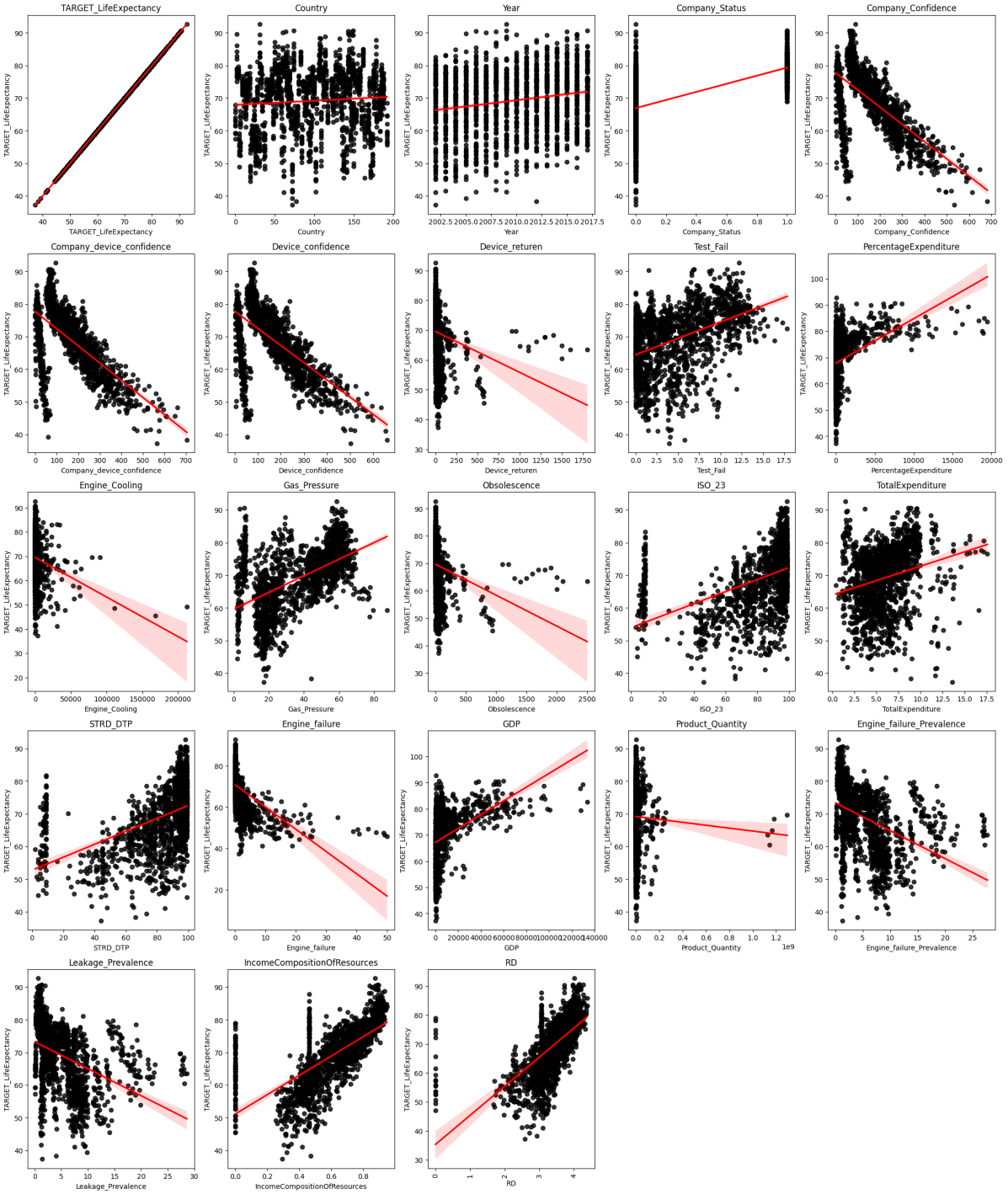



# Data Visualisations

We will now plot TARGET_LifeExpectancy over all the features, which is only for the training dataset (testing data should be kept hidden). From the graphs below we can see a few things. The first thing is a some plots look similar to each other. The Company_Confidence, Company_device_confidence and Device_confidence plots are similar. The same goes for the ISO_23 and STRD_DTP plots, alongside the Engine_failed_Prevalence and Leakage_Prevalnce plots. When the plots are similar, it suggests linear relationships between those features. This can be checked with a correlation matrix. The second thing is that Company_Confidence, Company_device_confidence, Device_confidence, IncomeCompositionOfResources and RD seem to have a strong linear relationship with TARGET_LifeExpectancy (if we disregard the outliers when the values are close to 0). This can also be checked with a correlation matrix.

In [14]:
```
max_col = 5
max_row = 6
figsize = (25, 30)
```

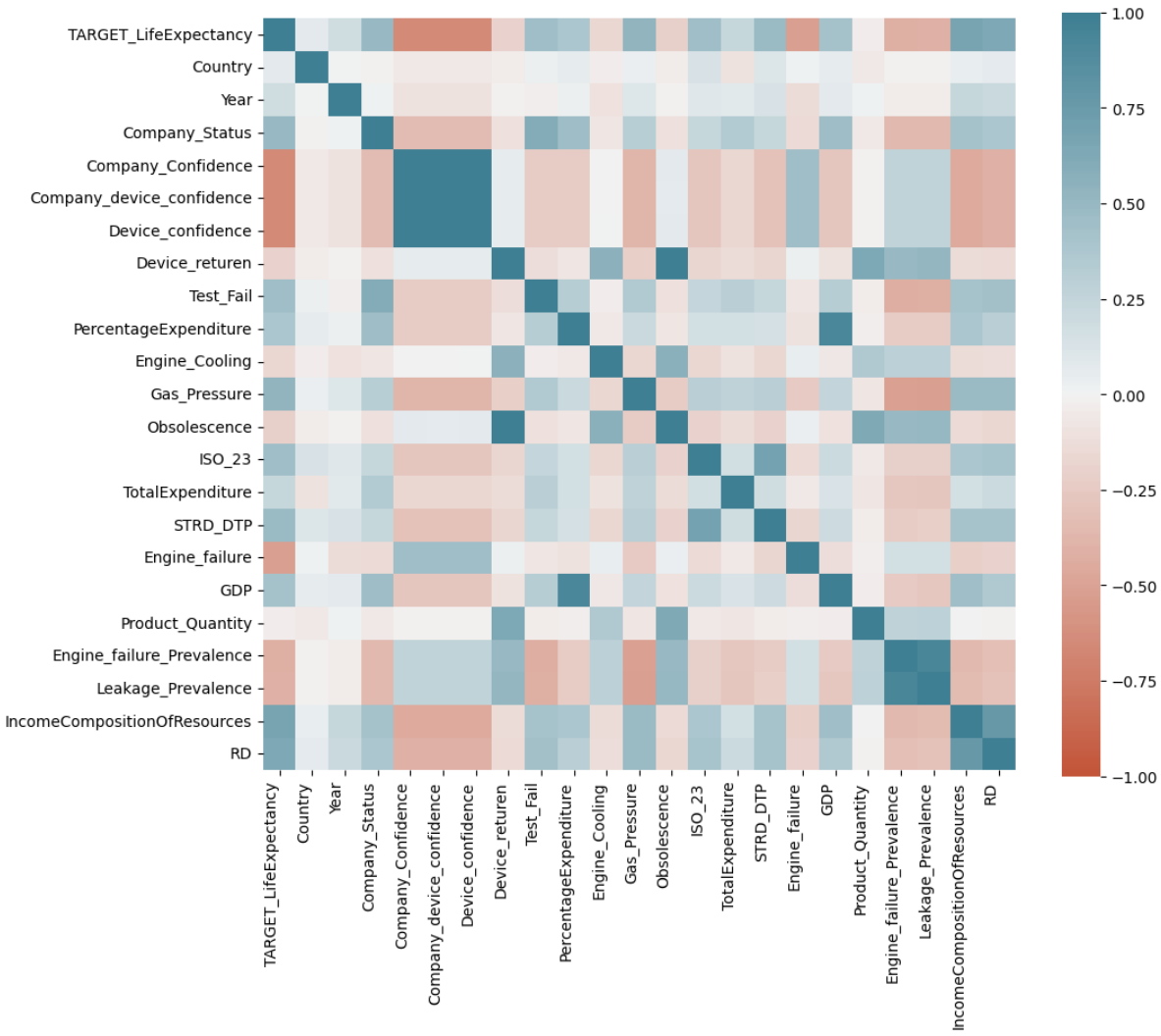
```
plt.figure(figsize=figsize)
for i, col in enumerate(train_data.columns):
    plt.subplot(max_col,max_col,i+1)
    sns.regplot(x=col, y='TARGET_LifeExpectancy', data=train_data,
                scatter_kws={"color": "black"}, line_kws={"color": "red"})
    plt.title(col)


plt.xticks(rotation='vertical')
plt.show()
```




We will now plot the correlation matrix. From the correlation matrix below, we can see that

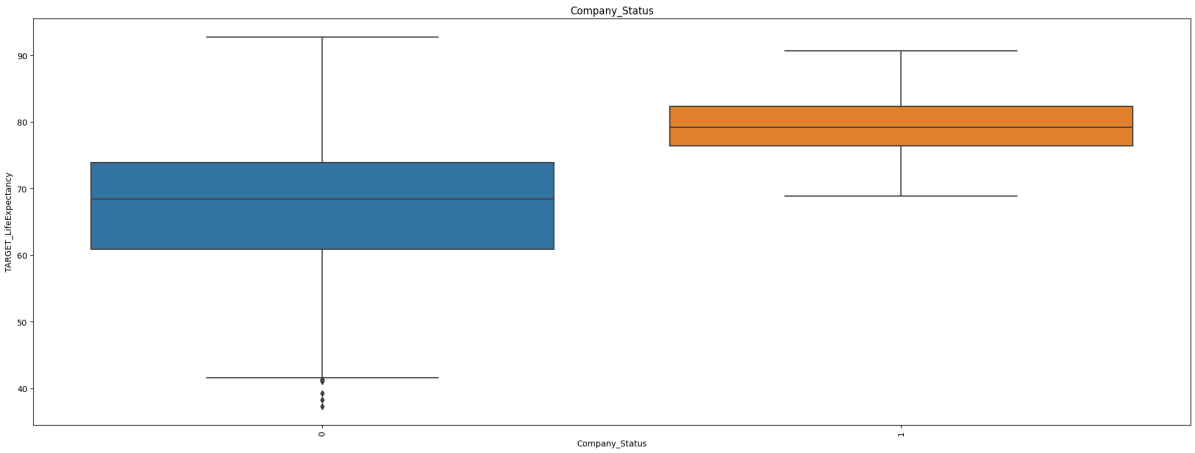
Company_Confidence, Company_device_confidence, Device_confidence are strongly positively correlated (shown with dark blue boxes). The same goes with Engine_failure_Prevalence and Leakage_Prevalence, alongside IncomeCompositionOfResources and RD (not as strong and shown with slightly lighter blue boxes). We can also see that Company_Confidence, Company_device_confidence and Device_confidence are strongly negatively correlated with TARGET_LifeExpectancy (shown with dark red boxes). IncomeCompositionOfResources and RD have a less than expected positive correlation (shown with slightly dark blue boxes), but this could be due to the influence of outliers when the values are close to 0, as seen in the scatter plots above.

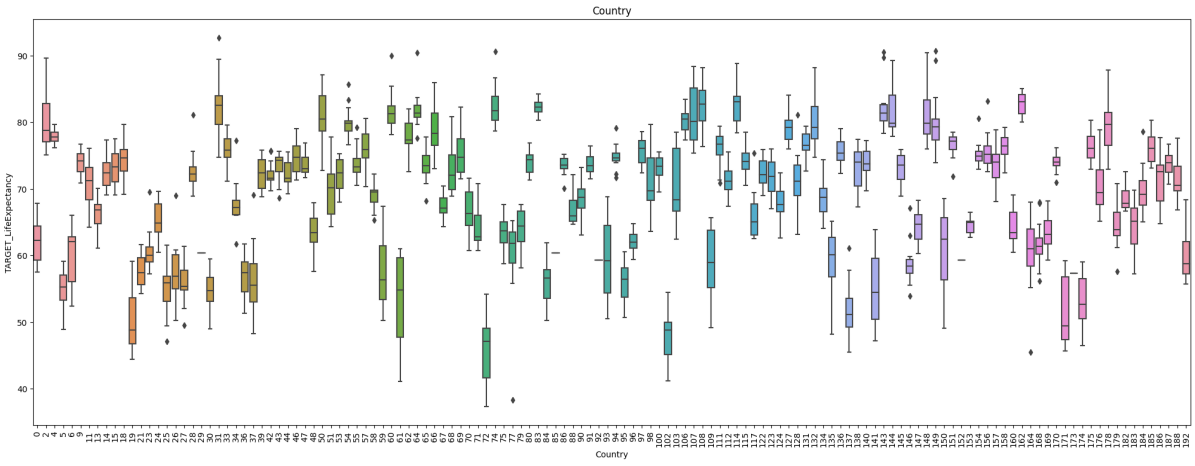
In [15]:
```
f, ax = plt.subplots(figsize=(11, 9))
corr = train_data.corr()
ax = sns.heatmap(
    corr,
    vmin=-1, vmax=1, center=0,
    cmap=sns.diverging_palette(20, 220, n=200),
    square=True
)
ax.set_xticklabels(
    ax.get_xticklabels(),
    rotation=90,
    horizontalalignment='right'
);
```

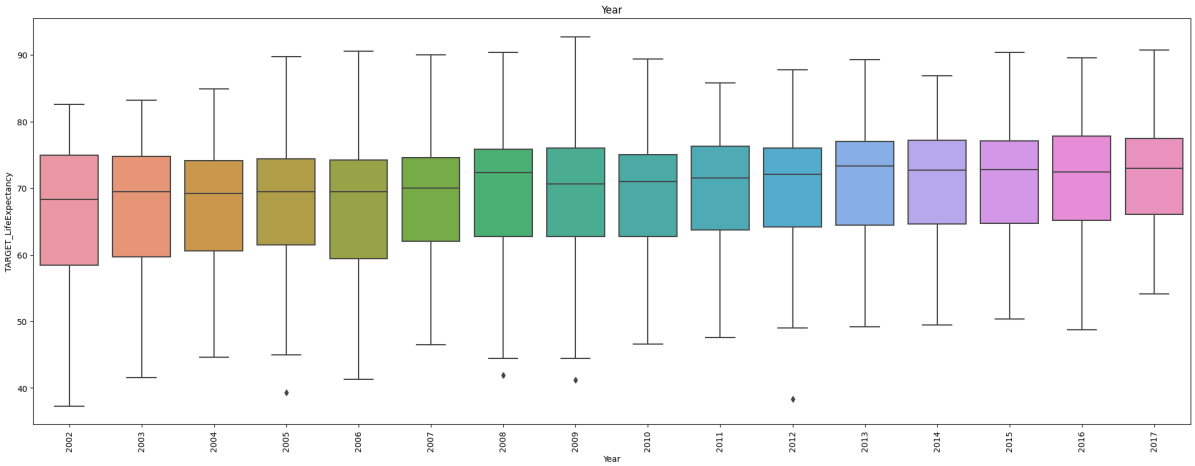



To visualise how categorical features affect the target feature, we will plot boxplots of the TARGET_LifeExpectancy with respect to the categorical features. From the boxplots below we can see the following relationships. If Company_Status is 1, then TARGET_LifeExpectancy is expected to be higher than if it were to be 0. Country 31 is expected to have the highest TARGET_LifeExpectancy than the other countries. Year doesn't seem to have an effect on the TARGET_LifeExpectancy as the median doesn't change throughout the years and all boxplots have similar interquartile ranges.

In [16]:
```
max_col = 1
max_row = 3
figsize = (25,30)

boxplot_cat_features = ['Company_Status', 'Country', 'Year']
plt.figure(figsize=figsize)
for i, col in enumerate(boxplot_cat_features):
    plt.subplot(max_row, max_col,i+1)
    sns.boxplot(x=col, y='TARGET_LifeExpectancy', data=train_data)
    plt.title(col)
    plt.xticks(rotation='vertical')
```



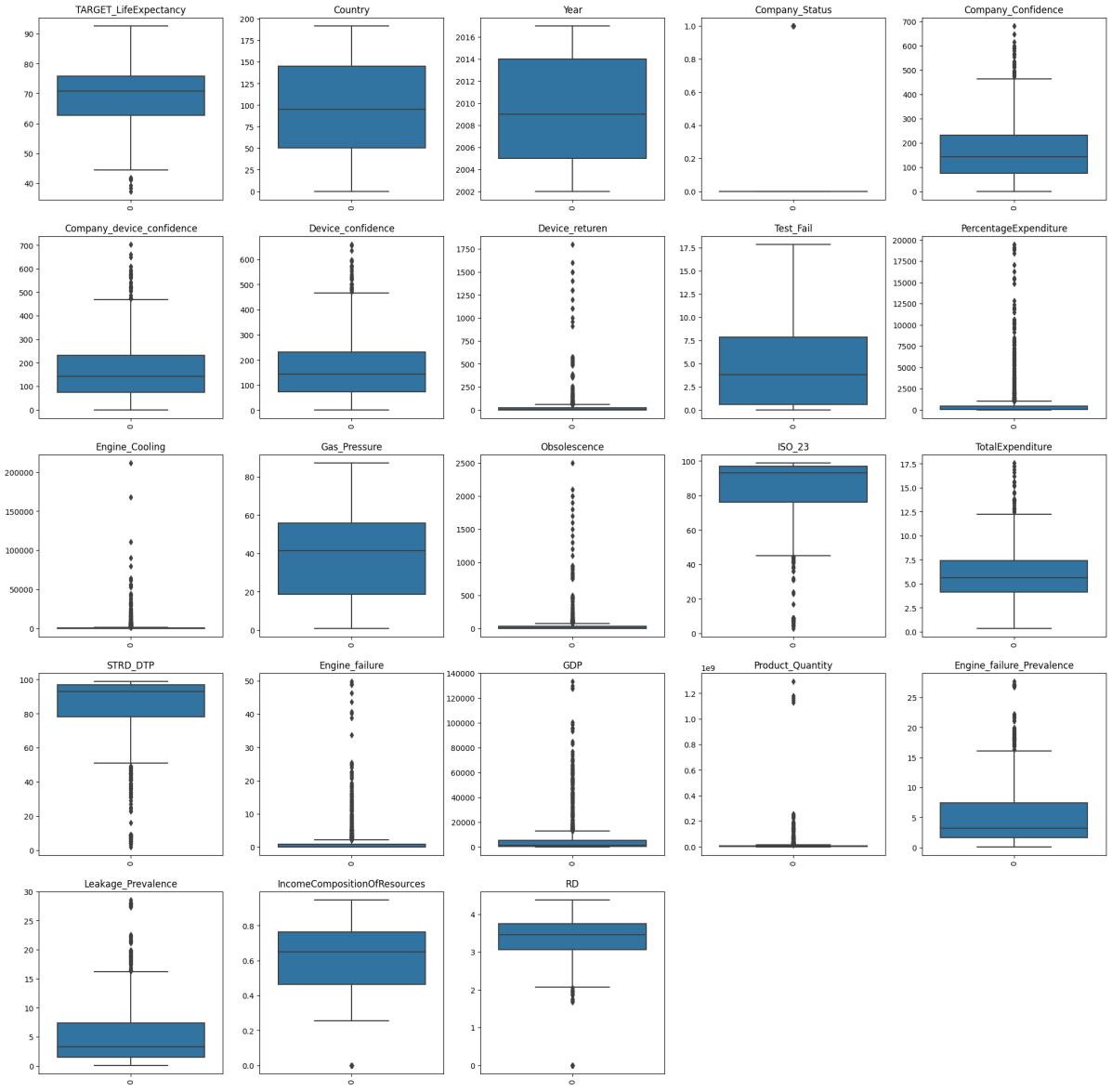




We will now plot the boxplots to visualize any outliers for our continuous features. In the box plots below, to be considered an outlier, the point must be either 1.5 times the interquartile range above the 3rd quarter or below the 1st quarter (Tukey's rule). That is if the point is below the upper fence or below the lower fence on the plot, it is an outlier. We can see that all continuous features except Gas_Pressure have at least one outlier.

In [17]:
```
max_col = 5
max_row = 6
figsize=(25,30)

plt.figure(figsize=figsize)
```

```
for i, col in enumerate(train_data.columns):
    plt.subplot(max_row, max_col,i+1)
    sns.boxplot(train_data[col])
    plt.title(col)
    plt.xticks(rotation='vertical')
```




# Scaling and Encoding

Before we begin finding the best model for our ultimate judgement, we need to figure out kind of problem this is: whether it is a classification or regression problem. The target variable we are trying to predict is the life expectancy of the device in months. This is a continuous value and thus we should consider this as a regression problem (classification is only used when our output is a categorical value).

Next, we need to figure out how we are going to scale our data and how to deal with categorical features. To scale our continuous data, we will be using robust scaling. The reason why this method was chosen over the other scaling methods is because it is "robust

to outliers" according to scikit-learn at: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html#sklearn.preproces Robust scaling is "robust to outliers" is because the values are subtracted by the median before being divided by the interquartile range. As shown in the box plots above, the dataset has many outliers, therefore making robust scaling perfect for our model or models. To deal with categorical data, we will be performing one hot encoding on only the Country feature. The reason why we aren't doing this on the other categorial features, Company_Status and Year is because Company_Status is already binary and adding the extra column will be redundant (Akshay Gupta at: https://www.kaggle.com/discussions/getting-started/114797). As for Year, it is ordinal and therefore should be treated as a continuous variable (which will be scaled) (Alexander Robixtzsch at: https://www.frontiersin.org/articles/10.3389/feduc.2020.589965/full).

Futhermore, one hot encoding should be done respect to the training set's categorical features. That is, the columns generated from one hot encoding a training set's categorical feature should also be apart of the transformed testing set. If the testing set has a categorical value that does not exist in the training set, then all its hot encoded columns (for that feature) for that observation should be 0. If not done this way and one hot encoding is done on the entire dataset (not seperately on training and testing), then the transformed training dataset will have knowledge of the testing dataset and therefore making the model unable to generalise for real unseen data (mickey at: https://stackoverflow.com/questions/55525195/do-i-have-to-do-one-hot-encoding-separately-for-train-and-test-dataset). This also applies the robust scaling. The testing data must be scaled with the training data's median and interquartile range (and not the combined dataset's median and interquartile range) so the training data set is not dependent on the testing data set.

The target variable might have to be scaled since "A target variable with a large spread of values, in turn, may result in large error gradient values causing weight values to change dramatically, making the learning process unstable" by Jason Brownie at https://machinelearningmastery.com/how-to-improve-neural-network-stability-and-modeling-performance-with-data-scaling/. The scaling should only be applied when training the model, so when we end up making predictions with the model, the predictions will not be scaled. Robust scaling was chosen to minimize the influence of outliers in the target column.

In [18]:
```
#year not included, its ordinal
categorical_features = ['Country', 'Company_Status']
#set difference to get numeric features
numeric_features = x_train.columns.difference(categorical_features)
#features to do one hot encoding on
hot_encoding_features = ['Country']
```

In [19]:
```
#transformer to do transformations on our data set columns
column_transformer = make_column_transformer(
    #robust scale numeric features
```

```
    #sklearn does scaling of x_test with respect to x_train for us
    #when we fit for x_train later
    (RobustScaler(), numeric_features),
    #one hot encoding, 'ignore' allows observations
    #with 0's for all one hot encoded columns
    #sklearn does one hot encoding of x_test with respect to x_train for us
    #when we fit for x_train later
    (OneHotEncoder(handle_unknown='ignore'), hot_encoding_features),
    #let Company_Status pass through
    remainder='passthrough'
)
```

# Models

## Normal Linear Regression

For our first model, we will first try normal linear regression to see what results the most basic approach will give us.

In [20]:
```
lin_reg = make_pipeline(
    #column transformer from above
    column_transformer,
    #scale the target variable before training
    TransformedTargetRegressor(regressor=LinearRegression(),
                               transformer=RobustScaler())
)
```

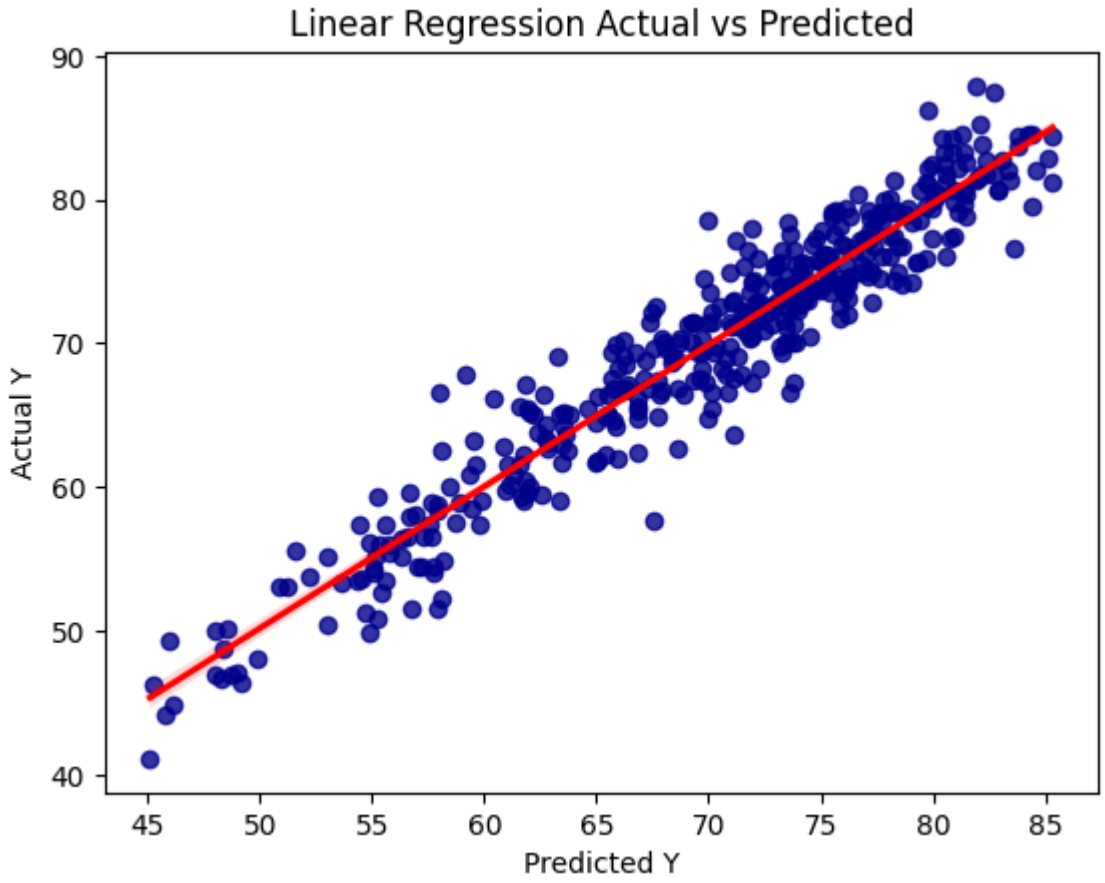
In [21]:
```
#train our linear regression model
lin_reg.fit(x_train, y_train);
```

In [22]:
```
#plot the actual Y over the predicted Y for the model
def showModelResults(model, x_test, y_test, title):
    y_test_hat = model.predict(x_test)
    sns.regplot(x=y_test_hat, y=y_test,
                scatter_kws={"color": "darkblue"}, line_kws={"color": "red"})
    plt.title(title)
    plt.xlabel('Predicted Y')
    plt.ylabel('Actual Y')
    plt.show()
```

From the plot below, we can see a strong correlation between the predicted output and the actual output for normal linear regression model. This is mathematically confirmed with the R-squared score of 0.9194 when the testing data is put through the model (R-squared score of 0.9209 for training data). "R-squared represents the proportion of variance of dependent variable that's explained by an independent variable or variables in a regression model" (Fernando, Smith, Perez at: https://www.investopedia.com/terms/r/r-squared.asp). Therefore 91.94% of the variation of the Life expectancy of the the device in this case is explained by

the indepedent variables. With this, we can see that with just a normal linear regression model, it can already give us accurate predictions.

In [23]:
```
showModelResults(lin_reg, x_test, y_test,
                 'Linear Regression Actual vs Predicted')
```




In [24]:
```
#.score defaults to R-squared for regression
print('Linear Regression Train R-squared: ',
      lin_reg.score(x_train, y_train))
print('Linear Regression Test R-squared: ',
      lin_reg.score(x_test, y_test))
```

```
Linear Regression Train R-squared:  0.9209279280724927
Linear Regression Test R-squared:  0.9194067389429678
```

## Normal Polynomial Regression

For our second model, we will try normal polynomial regression. However, before we do, we need to figure out when we should add our polynomial columns. Do we add them before scaling or after scaling? According to samcha at: https://samchaaa.medium.com/preprocessing-why-you-should-generate-polynomial-features-first-before-standardizing-892b4326a91, we should be scaling after we generate polynomial features. Multiplying interactions after they are scaled could give us a values with smaller than expected magnitudes or random negative values.

In [25]:
```
#this exists because I don't know how to make
#sklearn pipeline do scaling after generating polynomial features
#how do I reference the polynomial columns to do
#scaling for all of them instead of just scaling the numeric features?

#pass in the fitted polynomialFeatures, dataset,
#numeric featrure names and categorical feature names

#returns the entire dataset with the added
#polynomial columns alongside the othter columns
def create_poly_columns(poly_features, dataset, numeric_features, categorical_featu
    poly_numeric_features = poly_features.get_feature_names_out(numeric_features)

    dataset_numeric_poly = pd.DataFrame(
        poly_features.transform(dataset[numeric_features]),
        columns=poly_numeric_features)
    dataset_categorical = dataset[categorical_features]
    dataset_numeric_poly.reset_index(drop=True, inplace=True)
    dataset_categorical.reset_index(drop=True, inplace=True)
    dataset_poly = pd.concat([dataset_numeric_poly, dataset_categorical], axis=1)
    return dataset_poly
```

In [26]:
```
#PolynomialFeatures does all combinations of column multiplication
#and therefore will fry my latop if I do above degree 2
poly_features = PolynomialFeatures(degree=2).fit(x_train[numeric_features])
poly_numeric_features = poly_features.get_feature_names_out(numeric_features)

x_train_poly = create_poly_columns(
    poly_features, x_train, numeric_features, categorical_features)
x_test_poly = create_poly_columns(
    poly_features, x_test, numeric_features, categorical_features)
```

In [27]:
```
#this is nearly the same as the column transformer from above
poly_column_transformer = make_column_transformer(
    #now I can reference the added polynomial features to scale them
    (RobustScaler(), poly_numeric_features),
    (OneHotEncoder(handle_unknown='ignore'), hot_encoding_features),
    remainder='passthrough'
)
```

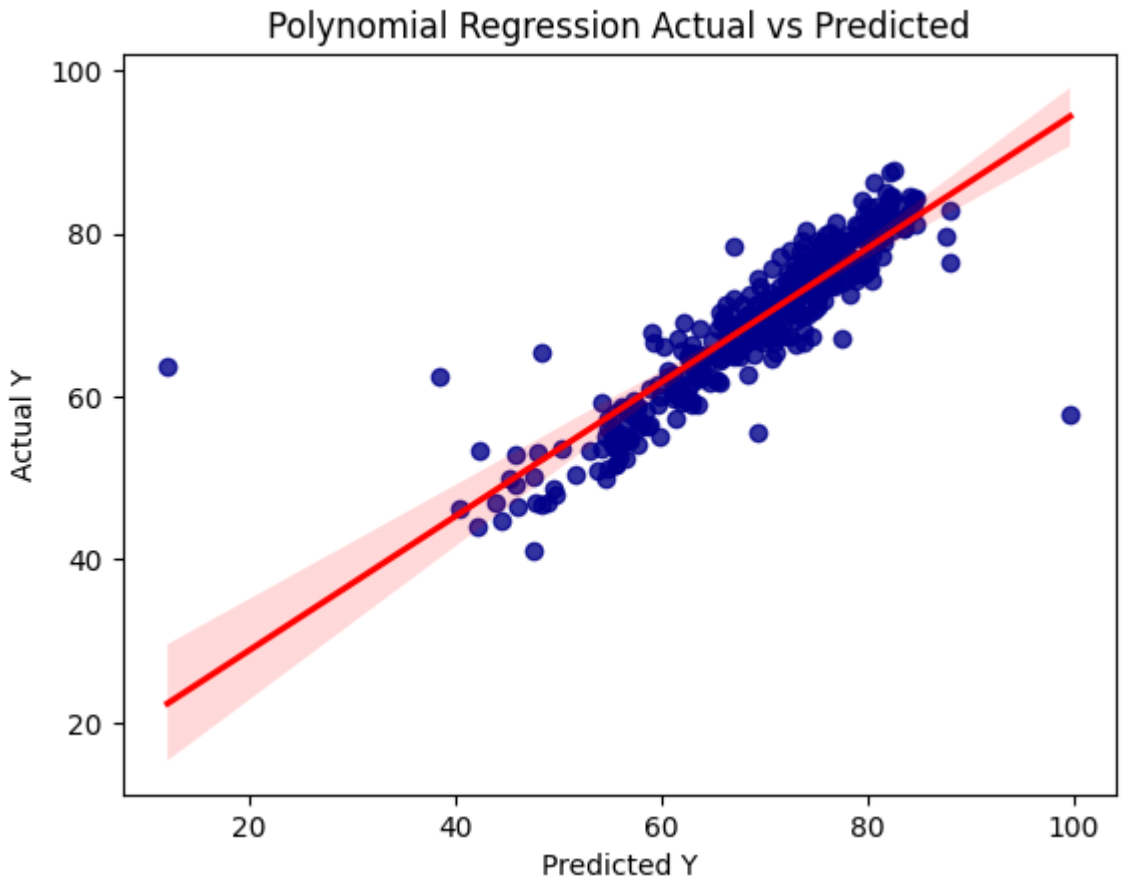
In [28]:
```
poly_reg = make_pipeline(
    poly_column_transformer,
    TransformedTargetRegressor(regressor=LinearRegression(),
                               transformer=RobustScaler())
)
```

In [29]:
```
poly_reg.fit(x_train_poly, y_train);
```

From the plot below we can see that there is some correlation between predicted and actual output for the Polynomial Regression model. We got a moderate R-squared score of 0.7463 when the testing data is put through the model and a high R-squared score of 0.9387 for the training data. This large difference in R-squared score values was expected. Adding polynomial features and optimising their coefficients on the training dataset is an example of

overfitting. The model will perform well for the training dataset since it will have been optimised for it (hence the high R-squared score of 0.9387 for training data), however when it is time to make predictions on the testing data the model will perform poorly since it has been too optimised on the training data and therefore cannot generalise (hence a moderate R-squared score of 0.7463). Overfitting is a bigger issue with Polynomial Regression than Linear Regression as the added polynomial features allows the model is be more flexible especially for higher degrees.

In [30]:
```
showModelResults(poly_reg, x_test_poly, y_test,
                 'Polynomial Regression Actual vs Predicted')
```




In [31]:
```
print('Polynomial Regression Train R-squared: ',
      poly_reg.score(x_train_poly, y_train))
print('Polynomial Regression Test R-squared: ',
      poly_reg.score(x_test_poly, y_test))
```

```
Polynomial Regression Train R-squared:  0.9386722158489341
Polynomial Regression Test R-squared:  0.7463387809087949
```

## Regularisation and Cross Validation

The above models could still be improved upon using regularisation. Regularisation includes a term in the loss function which increases the output of the loss function by a value (denoted as alpha by sklearn) either multiplied by the sum of the magnitude of cofficient

terms (Lasso) or sum of cofficient terms squared (Ridge). This way the model is punnished for having coefficients terms with high magnitudes. Ideally insignificant featues will have their coefficient term becomes 0 or close to it, which would essentially be same as removing the feature.

However, we need a way to find alpha (the tuneable hyperparameter for regularisation). To do this, we will use Gridsearch. Gridsearch simply loops through every alpha in a list of alphas and calculates a score (in our case the score is R^2) for each alpha. Then the alpha which adjusts the model's cofficients to give us the highest score is chosen.

However, choosing the alpha which gives us the highest score, could mean that we have overfitted on our testing dataset. If we keep manually changing the alpha until we get a good score on the testing dataset, then there is a good chance that the testing dataset has influenced the model and thus making the model unable to generalise. A solution to this problem would be optimising the score using a seperate dataset instead of the testing dataset. This dataset is usually referred as the validation set. we do not have to make another split on our original dataset to get our validation set, as this set can generated by k-fold cross validation.

k-fold cross validation works by spliting the dataset into k parts. Each part will have a turn to be the validation set and rest of the parts combined will be the training dataset. The model will be trained on the training set and the score will be calculated on the validation set. This means that k scores will be generated for each of the k validation sets. The mean of those k scores is then calculated which will be the final score for the current alpha value in Gridsearch. (scikit-learn cross validation at: https://scikit-learn.org/stable/modules/cross_validation.html).

## Linear Lasso Regularisation

For our third model, we will try Linear Lasso Regression to see whether regularisation really improves on normal Linear Regression which can already make accurate predictions.

In [32]:
```
#tuneable hyperparameter list
alphas = np.logspace(-8, 0, 100)

lin_lasso = make_pipeline(
    column_transformer,
    TransformedTargetRegressor(regressor=Lasso(), transformer=RobustScaler()),
)

#perform gridsearch
#cv: cross validation folds
#n_jobs: number of cores (-1 means all cores)
#return_train_score: whether to calculate the mean score on the testing set
lin_lasso_params = {'transformedtargetregressor__regressor__alpha': alphas}
grid_lin_lasso = GridSearchCV(lin_lasso, lin_lasso_params, cv=10, n_jobs=-1,
                              return_train_score=True)
```

In [33]:
```
grid_lin_lasso.fit(x_train, y_train);
```

Below we can see what what our alpha for Linear Lasso Regression is.

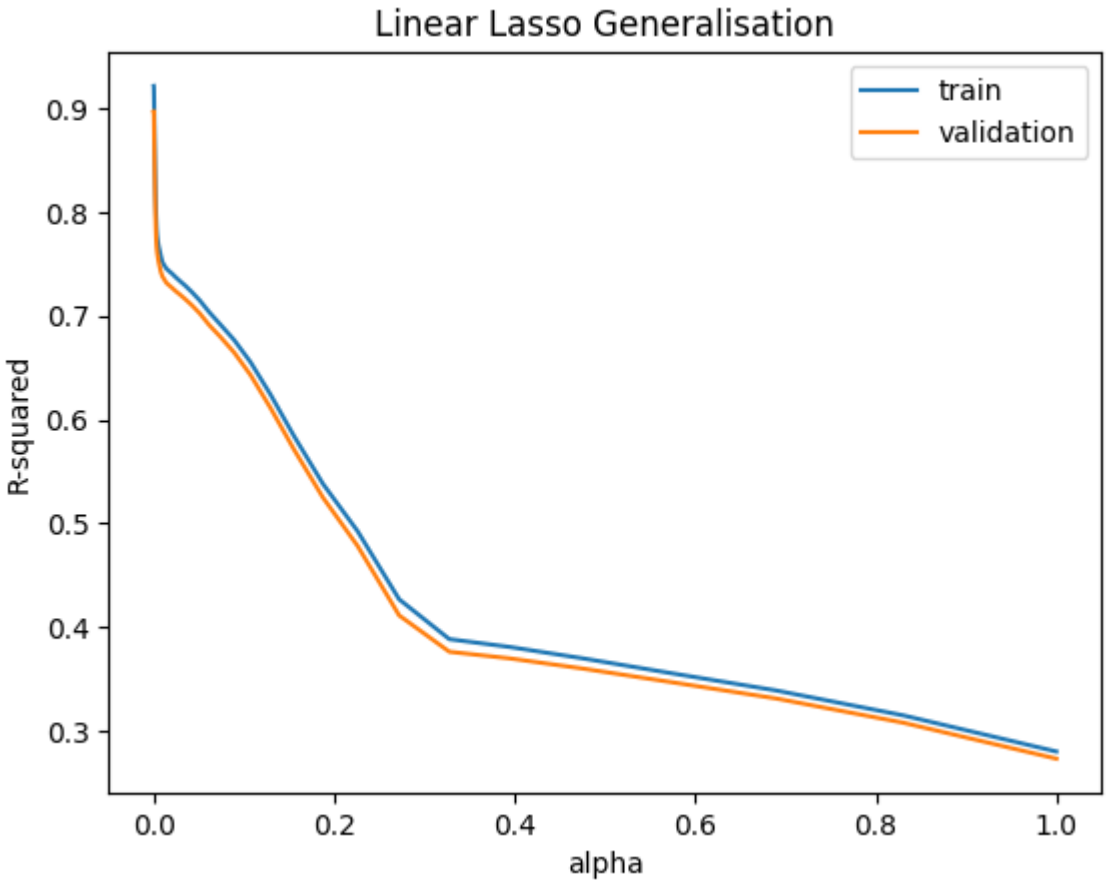
In [34]:
```
grid_lin_lasso.best_params_
```

Out[34]:
```
{'transformedtargetregressor__regressor__alpha': 2.0565123083486515e-05}
```

In [71]:
```
def showGeneralization(model, graph_name, metric_name):
    mean_train_score = model.cv_results_['mean_train_score']
    mean_val_score = model.cv_results_['mean_test_score']

    alpha_param_name = 'param_transformedtargetregressor__regressor__alpha'
    alphas = model.cv_results_[alpha_param_name].data

    plt.plot(alphas, mean_train_score, label='train')
    plt.plot(alphas, mean_val_score, label='validation')
    plt.title(graph_name)
    plt.ylabel(metric_name)
    plt.xlabel('alpha')
    plt.legend()
    plt.show()
```

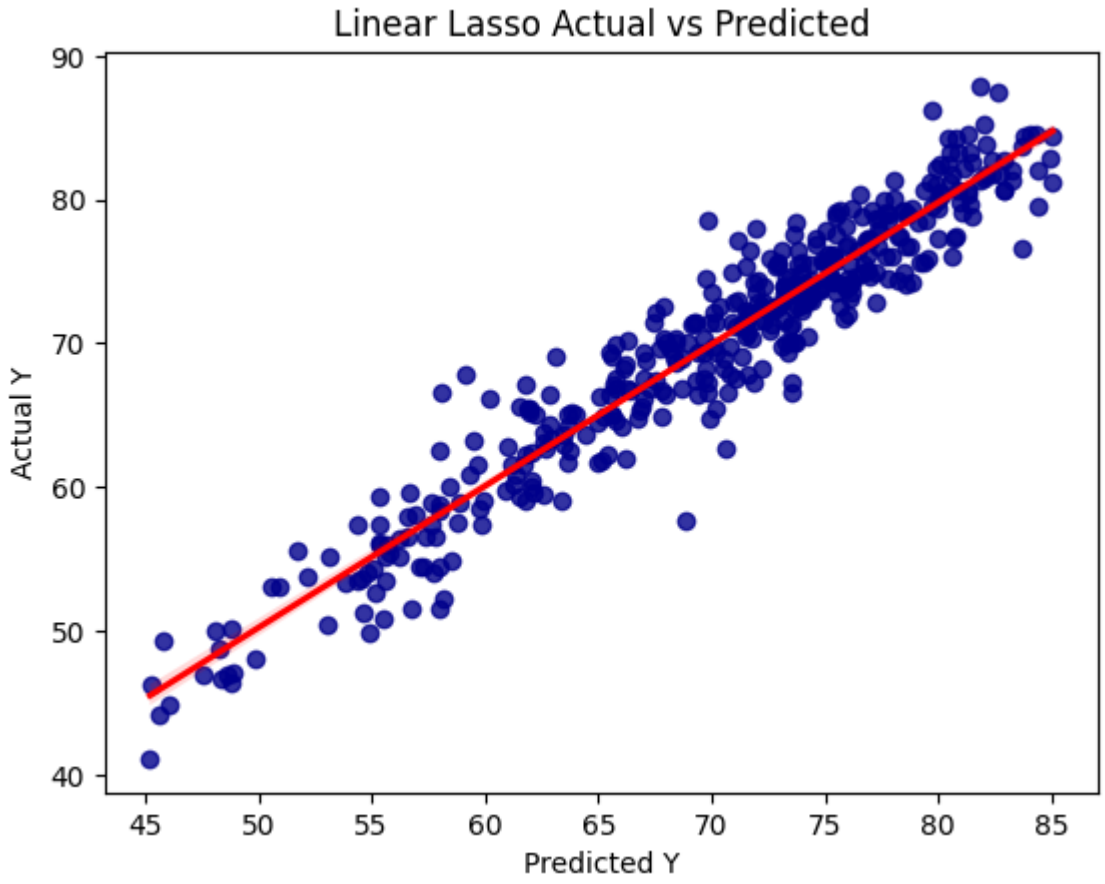
We will now graph how well Linear Lasso Regression was able to generalise on the validation set after training on the training data set. On the plot below, it looks like if alpha goes towards 0, then our model will generalise better (R-squared score on validation set is increasing). If this is the case, then we will converge towards normal Linear Regression. Therefore from the graph, it seems Lasso Regularisation might not improving the base Linear Regression model. This is proven with its R-squared score for testing below which is very similar to the one for normal Linear Regression.

In [36]:
```
showGeneralization(grid_lin_lasso,
                   'Linear Lasso Generalisation', 'R-squared')
```




From the plot below, we can see a strong correlation between the predicted output and the actual output for the Linear Lasso Regression model. For the training dataset we got an R-squared score of 0.9205 and for testing data we got 0.9196. This is not much of a difference with the results from the normal Linear Regression model (0.9209 and 0.9194 respectively).

In [37]:
```
showModelResults(grid_lin_lasso, x_test, y_test,
                 'Linear Lasso Actual vs Predicted')
```




In [38]:
```
print('Linear Lasso Regression Train R-squared:',
      grid_lin_lasso.score(x_train, y_train))
print('Linear Lasso Regression Test R-squared: ',
      grid_lin_lasso.score(x_test, y_test))
```

```
Linear Lasso Regression Train R-squared: 0.9205026124367696
Linear Lasso Regression Test R-squared:  0.9196252662146455
```

## Linear Ridge Regression

For our fourth model, we will try Linear Ridge Regression to check if there is difference between Lasso and Ridge for regularisation.

In [40]:
```
alphas = np.linspace(-8, 1, 100)

lin_ridge = make_pipeline(
    column_transformer,
    TransformedTargetRegressor(regressor=Ridge(),
                               transformer=RobustScaler())
)
lin_ridge_params = {'transformedtargetregressor__regressor__alpha': alphas}
grid_lin_ridge = GridSearchCV(lin_ridge, lin_ridge_params, cv=10, n_jobs=-1,
                              return_train_score=True)
```

In [41]:
```
grid_lin_ridge.fit(x_train, y_train);
```

As a side note, there is a noticeable difference of the performance of Ridge compared to Lasso. Ridge seems to be significantly faster than Lasso.

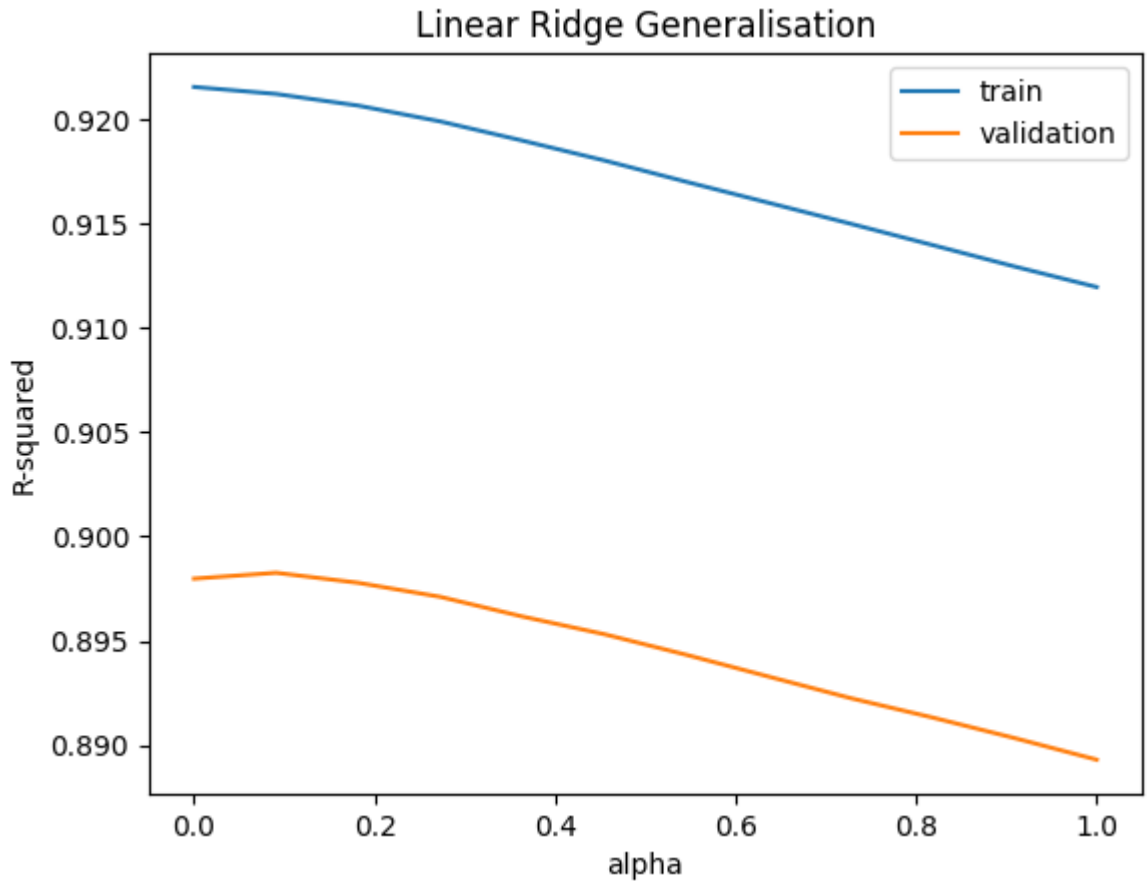
Below we can see what what our alpha for Linear Ridge Regression is.

In [42]:
```
grid_lin_ridge.best_params_
```

Out[42]:
```
{'transformedtargetregressor__regressor__alpha': 0.09090909090909172}
```

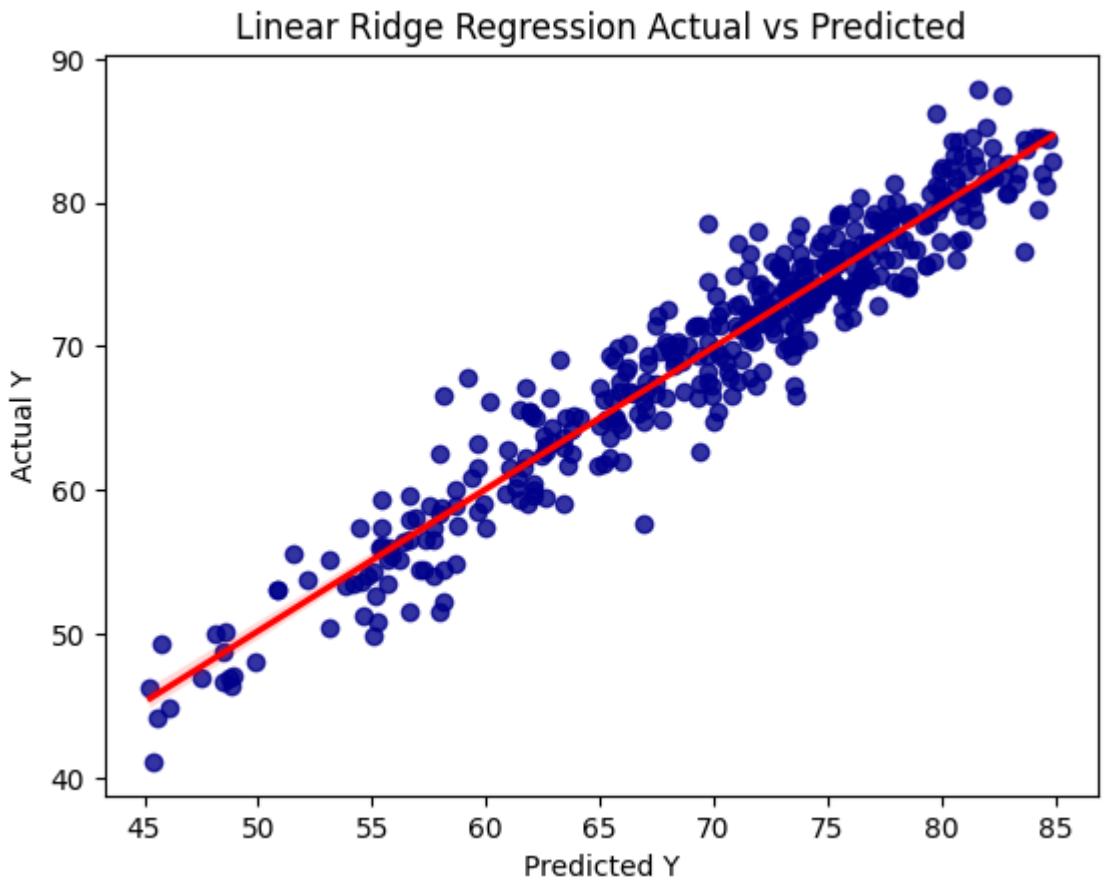
We will now graph how well Linear Ridge Regression was able to generalise on the validation set as the alpha changes. We can clearly see that there is a turning point where the validation score is at its maximum. Since the validation score is not at its maximum when the alpha is at zero or really really close to it, we can be sure that Ridge Regualarisation actually improves the base Linear Regression model.

In [807…
```
showGeneralization(grid_lin_ridge,
                   'Linear Ridge Generalisation', 'R-squared')
```




From the plot below, we can see a strong correlation between the predicted output and the actual output for Linear Ridge Regression model. For the training dataset we got an R-squared score of 0.9203 and for testing dataset we got 0.9213. This is slightly better than the results from the normal Linear Regression model (0.9209 and 0.9194 respectively) and Linear Lasso Regression model (0.9205 and 0.9196 respectively).

In [43]:
```
showModelResults(grid_lin_ridge, x_test, y_test,
                 'Linear Ridge Regression Actual vs Predicted')
```




In [44]:
```
print('Linear Ridge Regression Train R-squared: ',
      grid_lin_ridge.score(x_train, y_train))
print('Linear Ridge Regression Test R-squared: ',
      grid_lin_ridge.score(x_test, y_test))
```

```
Linear Ridge Regression Train R-squared:  0.9203599085313909
Linear Ridge Regression Test R-squared:  0.9213237113248863
```

## Polynomial Lasso Regression

For our fifth model, we will try a Polynomial Lasso Regression to see if adding polynomial terms for our regression with regularisation truly creates a better model.

In [45]:
```
alphas = np.logspace(-8, 0, 100)

poly_lasso = make_pipeline(
    #poly_column_transformer returns
    #degree of 2 so it doesnt take a gazillion years to complete
    poly_column_transformer,
    TransformedTargetRegressor(regressor=Lasso(), transformer=RobustScaler())
)
poly_lasso_params = {'transformedtargetregressor__regressor__alpha': alphas}
grid_poly_lasso = GridSearchCV(poly_lasso, poly_lasso_params, cv=10, n_jobs=-1,
                               return_train_score=True)
```

In [46]:
```
grid_poly_lasso.fit(x_train_poly, y_train);
```

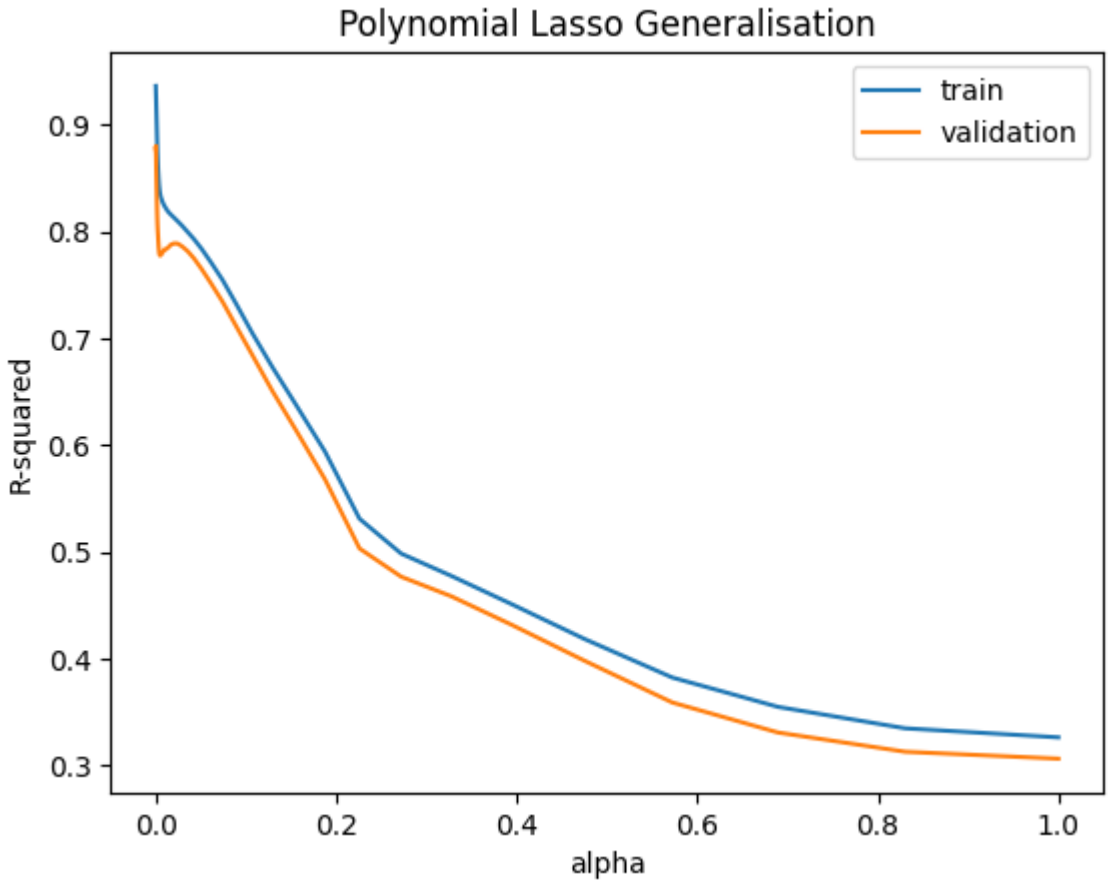
Below we can see what what our alpha for Polynomial Lasso Regression is.

In [47]:
```
grid_poly_lasso.best_params_
```

Out[47]:
```
{'transformedtargetregressor__regressor__alpha': 2.477076355991714e-05}
```

We will now graph how well Polynomial Lasso Regression was able to generalise on the validation set after training on the training data set. On the plot below, it looks like if alpha goes towards 0, then validation score will go towards its maximum. From the graph it looks Lasso Regularisation will make the model converge to a base Polynomial Regression model. However, this is not the case. If we look at the R-squared for the testing data set below, we got 0.9009 which is definitely not the same as the R-squared score of 0.7463 for the base Polynomial Regression. Therefore, there might be a turning point really close to 0 and that really small alpha was enough to for Regularisation to have a large enough influence on the base Polynomial Regression model.

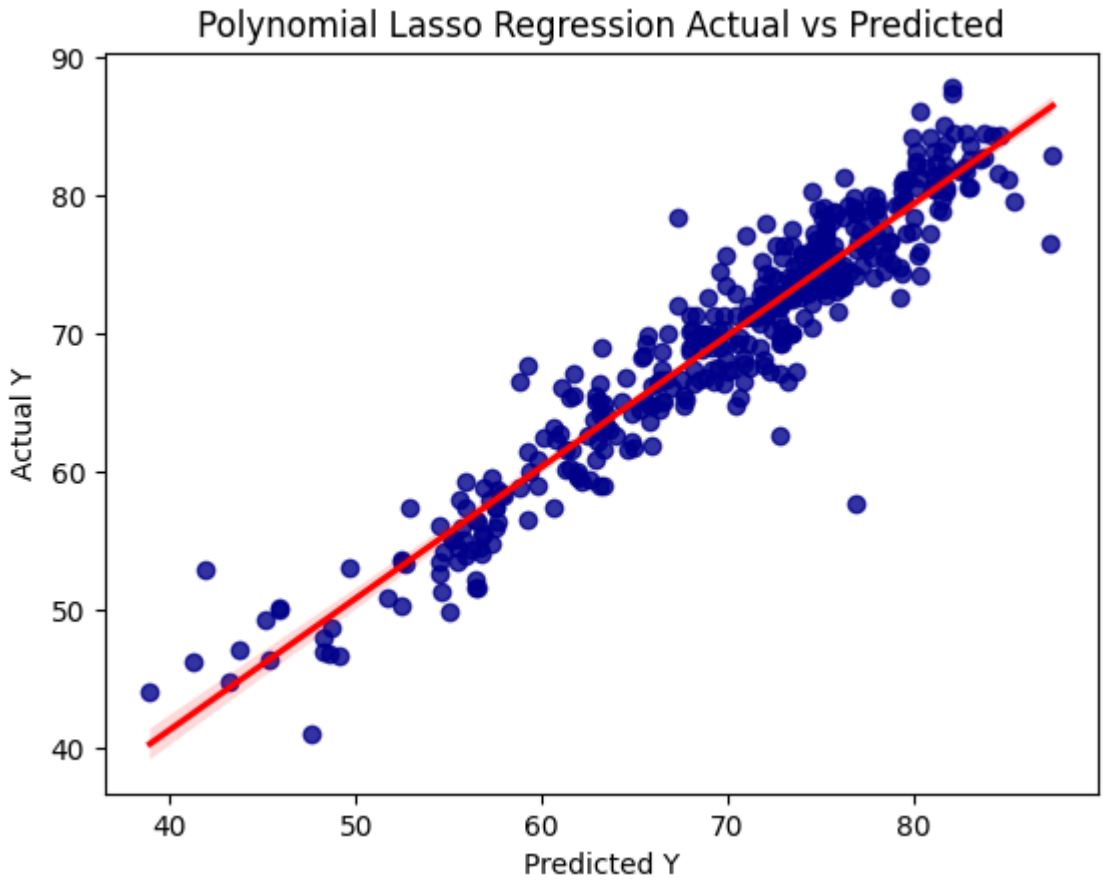
In [48]:
```
showGeneralization(grid_poly_lasso,
                   'Polynomial Lasso Generalisation', 'R-squared')
```




From the plot below, we can see a strong correlation between the predicted output and the actual output for Polynomial Lasso Regression model. For the training dataset we got an R-

squared score of 0.9340 and for testing dataset we got 0.9009. This was a big surprise as it was worse than all the Linear Regression models from above for the the testing dataset. We expected that if Linear Regression was truly the ideal model, the regularisation will converge the model to a Linear Regression model.

In [49]:
```
showModelResults(grid_poly_lasso, x_test_poly, y_test,
                 'Polynomial Lasso Regression Actual vs Predicted')
```




In [50]:
```
print('Polynomial Lasso Regression Train R-squared: ',
      grid_poly_lasso.score(x_train_poly, y_train))
print('Polynomial Ridge Regression Test R-squared: ',
      grid_poly_lasso.score(x_test_poly, y_test))
```

```
Polynomial Lasso Regression Train R-squared:  0.9339849346856447
Polynomial Ridge Regression Test R-squared:  0.9008880572293878
```

## Polynomial Ridge Regression

For our sixth and final model, we will try Polynomial Ridge Regression to check if it performs better than anyone of the models mentioned above.

In [51]:
```
alphas = np.logspace(-8, 1, 100)

poly_ridge = make_pipeline(
    poly_column_transformer,
```

```
    TransformedTargetRegressor(regressor=Ridge(), transformer=RobustScaler())
)
poly_ridge_params = {'transformedtargetregressor__regressor__alpha': alphas}
grid_poly_ridge = GridSearchCV(poly_ridge, poly_ridge_params, cv=10, n_jobs=-1,
                               return_train_score=True)
```

In [52]:
```
grid_poly_ridge.fit(x_train_poly, y_train);
```

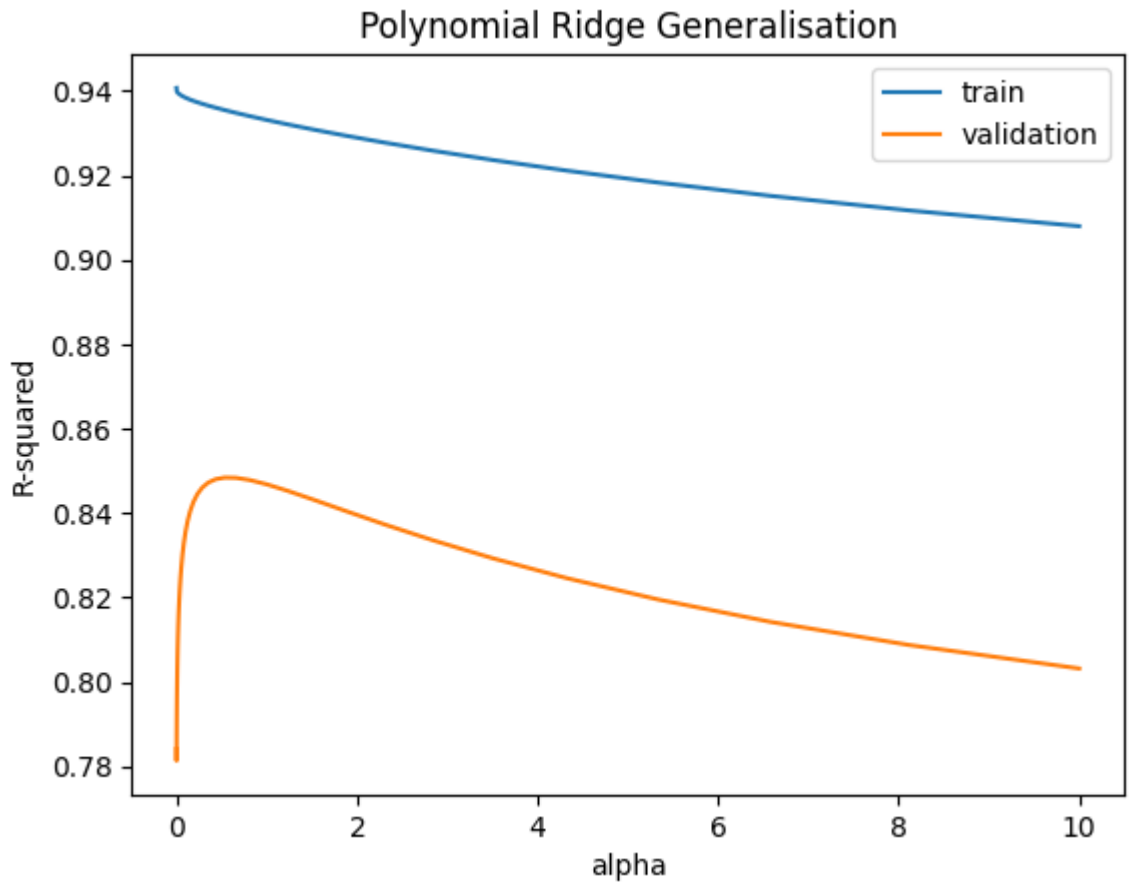
Below we can see what what our alpha for Polynomial Ridge Regression is.

In [53]:
```
grid_poly_ridge.best_params_
```

Out[53]:
```
{'transformedtargetregressor__regressor__alpha': 0.5336699231206312}
```

We will now graph how well Polynomial Ridge Regression was able to generalise on the validation set after training on the training data set. We can clearly see that there is a turning point where the validation score is at its maximum. Since the validation score is not at its maximum when the alpha is at zero or really really close to it, we can be sure that Ridge Regualarisation actually improves the base Polynomial Regression model.

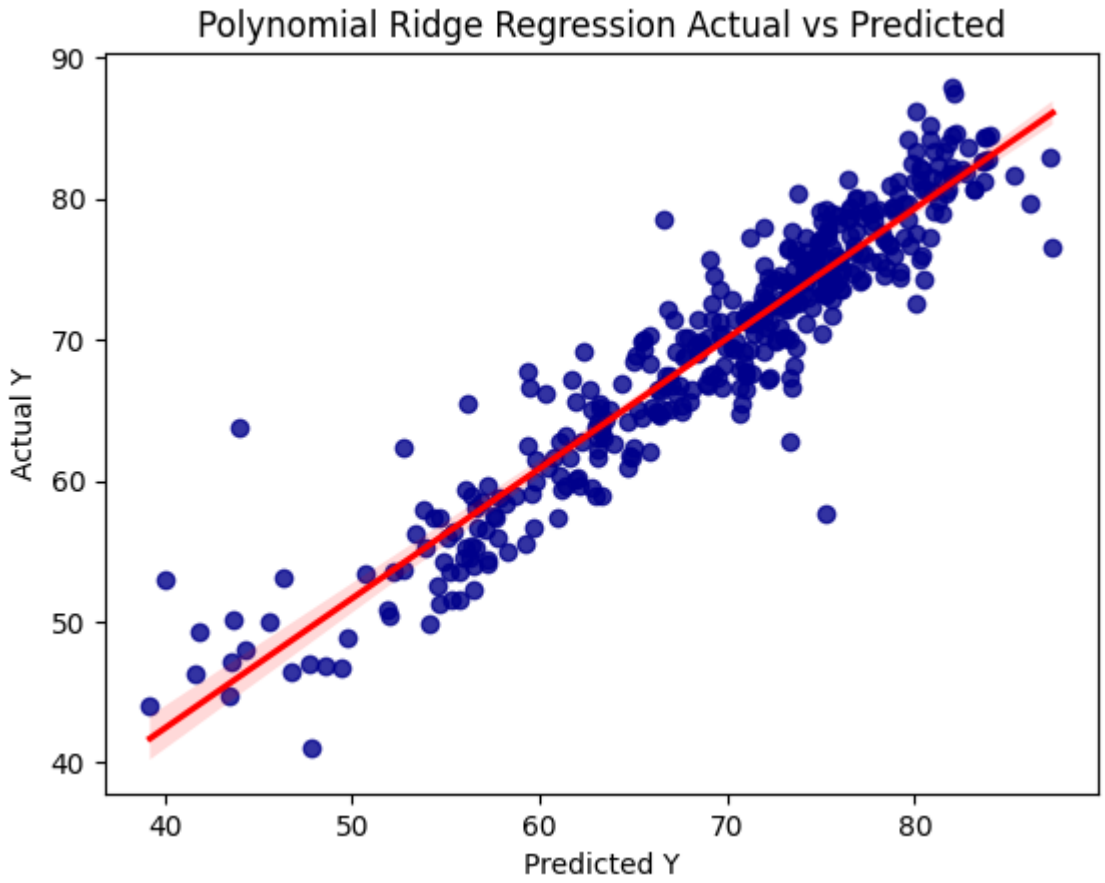
In [54]:
```
showGeneralization(grid_poly_ridge,
                   'Polynomial Ridge Generalisation', 'R-squared')
```




From the plot below, we can see a strong correlation between the predicted output and the actual output for Polynomial Lasso Regression model. For the training dataset we got an R-

squared score of 0.9334 and for testing dataset we got 0.8777. This is worse than all the mentioned models above except for normal Polynomial Regression. This came as a huge surprise as Linear Ridge Regression was slightly better than its alternative but Polynomial Ridge Regression was not.

In [55]:
```
showModelResults(grid_poly_ridge, x_test_poly, y_test,
                 'Polynomial Ridge Regression Actual vs Predicted')
```




In [56]:
```
print('Polynomial Ridge Regression Train R-squared: ',
      grid_poly_ridge.score(x_train_poly, y_train))
print('Polynomial Ridge Regression Test R-squared: ',
      grid_poly_ridge.score(x_test_poly, y_test))
```

```
Polynomial Ridge Regression Train R-squared:  0.9339386518435102
Polynomial Ridge Regression Test R-squared:  0.8772657701509915
```

# Ultimate Judgement

Linear Ridge Regression will be chosen as our Ultimate Judgement for a simple reason, it had the highest R-squared score of 0.9213 for testing (with an R-squared score of 0.9204 for training) compared to all the other models.

In [60]:
```
y_test_hat = grid_lin_ridge.predict(x_test)
```

In [63]:
```
predictions = test_ids.copy()
predictions['TARGET_LifeExpectancy'] = y_test_hat
```

In [66]:
```
out_file_name = 's3943224.csv'
predictions.to_csv(out_file_name, encoding='utf-8', index=False)
```

# References


- Alexander Robitzsch.(2020).Why Ordinal Variables Can (Almost) Always Be Treated as Continuous Variables.https://www.frontiersin.org/articles/10.3389/feduc.2020.589965/full
- Akshay Gupta.(2020).One Hot EnCoding.https://www.kaggle.com/discussions/getting-started/114797
- Azadeh Alavi.(2023).Week 3 Lab Exercises: Dataset splitting & Pre-Processing.https://bitbucket.org/alavi_a/rmit_cosc_2673_2793-2310/src/master/labs/week03/
- Azadeh Alavi.(2023).Week 3 Lab Exercises: Week 2 Lab Exercises: Reading data & Exploratory Data Analysis (EDA).https://bitbucket.org/alavi_a/rmit_cosc_2673_2793-2310/src/master/labs/week02/
- Fernando,Smith,Perez.(2021).R-Squared Formula, Regression, and Interpretations.https://www.investopedia.com/terms/r/r-squared.asp
- Google.(2022).Data Split Example.https://developers.google.com/machine-learning/data-prep/construct/sampling-splitting/example
- Jason Brownlee.(2020).How to use Data Scaling Improve Deep Learning Model Stability and Performance.https://machinelearningmastery.com/how-to-improve-neural-network-stability-and-modeling-performance-with-data-
- mickey.(2019).Do I have to do one-hot-encoding separately for train and test dataset?.https://stackoverflow.com/questions/55525195/do-i-have-to-do-one-hot-encoding-separately-for-train-and-test-dataset
- samcha.(2018).Preprocessing: why you should generate polynomial features first before standardizing.https://samchaaa.medium.com/preprocessing-why-you-should-generate-polynomial-features-first-before-standardizing-892b4326a91d
- Scikit-Learn.(2023).Cross-validation: evaluating estimator performance.https://scikit-learn.org/stable/modules/cross_validation.html
- Scikit-Learn.(2023).sklearn.preprocessing.RobustScaler.https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html#sklearn.prepr