**Regression with Radial Basis Function**

Barışcan Tunalı

Big Data Masters Student

Istanbul Bahcesehir University,

bariscantunali@outlook.com

# ABSTRACT

In this paper, I will explain my study applying RBF on given regression data and analyze the results with different number of iterations and clusters. In this homework cluster also means hidden neuron.

The aim is to predict the Y value using hybrid learning RBF network.

# Categories and Subject Descriptors

[**Computing methodologies**]: Machine learning—*Machine learning algorithms; Ensemble methods; Perceptron; Simple Perceptron; Radial Basis Function*

**Keywords**

SLP, RBF, Machine Learning, Classification, Prediction, Regression

# INTRODUCTION

The problem is mainly regression problem. I tried to implement it by using hybrid learning RBF. Since problem needs to be implemented in RBF, First we need to cluster first based on X values.

The paper is organized as follows: In Section 2, we briefly describe the dataset and data preparation. Next, in Section 3, we explain our classification approach in detail. Experimental results and comparisons with other algorithms are provided in Section 4. Finally, Section 5 concludes the paper.

# DATASET AND FEATURES

Dataset constists of 100 values, 50 Train,50 Validation.

My attributes are : [X], [Y], [Month], [Day], [FFMC], [DMC], [DC], [temp], [RH],[wind],[rain],[area]

Attribute name: max value – min value

[X] : Spatial

[Y] : Spatial

In order to make my program more flexible I created a class and handled all data operations. On presentation layer you only create class. After that you only access necessary train and test sets and send them to appropriate classification algorithms. Since our dataset is very small, I created arrays hardcoded in data layer. If necessary, it can be changed to read from another source without any changes in the rest of program.

ExtractData.SetDataTrain();

ExtractData.SetDataValidation();

RBF rbf = new RBF(cluster\_count, iteration\_count);

Data preparation class structure as below :

public static void SetDataTrain()

public static void SetDataValidation()

public static List<Dots> GetDataValidation()

public static List<Dots> GetDataTrain()

# METHOD

As I mentioned above, I used RBF to predict spatial Y.

My RBF class has 5 functions :

private void AssignClusterCenters(numClusters)

Assigns random cluster centers based on given number of clusters parameter.

private void InitialCluster()

Assings each item to necessary cluster based on Manhattan Distance.

private void FindSpreads()

Finds each clusters’ spreads. Center-Max\_X / 2

private void SimplePerceptronTrain()

It first calls InitializeWeights() to randomly assign weights. Then iterates the simple perceptron using RBF Activation function using Math.Exp(-1 \* Math.Abs(X -Center) /(2 \* spread \* spread)) formula.

private void SimplePerceptronValidate()

It uses trained network and applies it to validation data. Computes all of the stats such as mean error, mean square error etc.

I didn’t use any c# library or dll to implement RBF itself but I used lambda expressions and Math library to avoid code complexity.

# CODES

Main Class for data. It keeps data, and data related stuff.

public class Dots

{

public decimal X { get; set; }

public decimal Y { get; set; }

public decimal outY { get; set; }

public decimal[] DistanceToCentersByX { get; set; }

public decimal[] DistanceToCentersByXY { get; set; }

public decimal[] BelongingCluster { get; set; }

}

Gets data and encapsulates it with Dots class. Puts the encapsulated data to Dots list.

public static class ExtractData

{

static decimal[,] DatasetTrainMatrix;

static decimal[,] DatasetValidationMatrix;

static List<Dots> DatasetTrain;

static List<Dots> DatasetValidation;

static decimal[] Distance = new decimal[10] { 100000, 100000, 100000, 100000, 100000, 100000, 100000, 100000, 100000, 100000 };

public static void SetDataTrain()

{

DatasetTrainMatrix = new decimal[,]{};

DatasetTrain = new List<Dots>();

for (int i = 0; i < 50; i++)

{

Dots x = new Dots()

{

X = DatasetTrainMatrix[i, 0],

Y = DatasetTrainMatrix[i, 1]

};

x.DistanceToCentersByX = new decimal[10];

x.DistanceToCentersByXY = new decimal[10];

x.BelongingCluster = new decimal[10];

Array.Copy(Distance, x.DistanceToCentersByX, 10);

Array.Copy(Distance, x.DistanceToCentersByXY, 10);

Array.Copy(Distance, x.BelongingCluster, 10);

DatasetTrain.Add(x);

}

}

public static void SetDataValidation()

{

DatasetValidationMatrix = new decimal[,]{ };

DatasetValidation = new List<Dots>();

for (int i = 0; i < 50; i++)

{

Dots x = new Dots()

{

X = DatasetValidationMatrix[i, 0],

Y = DatasetValidationMatrix[i, 1]

};

x.DistanceToCentersByX = new decimal[10];

x.DistanceToCentersByXY = new decimal[10];

x.BelongingCluster = new decimal[10];

Array.Copy(Distance, x.DistanceToCentersByX, 10);

Array.Copy(Distance, x.DistanceToCentersByXY, 10);

Array.Copy(Distance, x.BelongingCluster, 10);

DatasetValidation.Add(x);

}

}

public static List<Dots> GetDataValidation() { return DatasetTrain; }

public static List<Dots> GetDataTrain() { return DatasetValidation; }

}

Keeps static functions. I used it to store main common calculations.

public static class Utils

{

public static decimal SimpleDistance(decimal X, decimal Y)

{

return Math.Abs(X - Y);

}

public static decimal Distance(Dots tuple, decimal ClusterCenterC, decimal ClusterCenterY)

{

// Euclidean distance between two vectors

decimal sumSquaredDiffs = 0.0M;

sumSquaredDiffs += Convert.ToDecimal(Math.Pow((Convert.ToDouble(tuple.X - ClusterCenterC)), 2));

sumSquaredDiffs += Convert.ToDecimal(Math.Pow((Convert.ToDouble(tuple.Y - ClusterCenterY)), 2));

return Convert.ToDecimal(Math.Sqrt(Convert.ToDouble(sumSquaredDiffs)));

}

}

Main RBF Class. Main algorithm everything is being handled here.

public class RBF

{

public decimal[,] ClusterCenters;

public decimal[] ClusterSpreads;

public int numClusters;

public decimal trainingCoefficient = 0.1m;

public decimal Momentum = 0.99m;

private decimal Bias = 1;

public int iterationLimit = 100;

public decimal[] Weights;

public RBF(int ParamNumClusters, int numOfIterations)

{

iterationLimit = numOfIterations;

numClusters = ParamNumClusters;

AssignClusterCenters(numClusters);

InitialCluster();

FindSpreads();

SimplePerceptronTrain();

SimplePerceptronValidate();

List<Dots> asdasd = ExtractData.GetDataTrain();

}

private void SimplePerceptronTrain()

{

InitializeWeights();

decimal tempSum = 0;

for (int i = 0; i < iterationLimit; i++)

{

foreach (Dots member in ExtractData.GetDataTrain())

{

#region Compute Output

for (int j = 0; j < numClusters; j++)

{

decimal a = Activation(member.X, ClusterCenters[j, 0], ClusterSpreads[j]);

decimal w = Weights[j];

tempSum += a\*w ;

}

#endregion

#region UpdateWeights

decimal delta = 0;

for (int j = 0; j < numClusters ; j++)

{

delta = trainingCoefficient \* (member.Y - tempSum) \* Activation(member.X, ClusterCenters[j, 0], ClusterSpreads[j]);

Weights[j] += delta;

}

#endregion

tempSum = 0;

}

}

}

public decimal MeanError = 0;

public decimal SumOfSquareError = 0;

public decimal SumOfSquareErrorMean = 0;

public decimal SumOfError = 0;

public decimal ErrorStdDev = 0;

public decimal MeanValue = 0;

public decimal SumOfSquareValue = 0;

public decimal SumOfSquareValueMean = 0;

public decimal SumOfValue = 0;

public decimal ValueStdDev = 0;

public decimal MeanOutput = 0;

public decimal SumOfSquareOutput = 0;

public decimal SumOfSquareOutputMean = 0;

public decimal SumOfOutput = 0;

public decimal OutputStdDev = 0;

public int CorrectCount = 0;

public int FalseCount = 0;

private void SimplePerceptronValidate()

{

decimal LocalError = 0;

CorrectCount = 0;

FalseCount = 0;

foreach (Dots member in ExtractData.GetDataValidation())

{

#region Compute Output

decimal output = 0;

for (int j = 0; j < numClusters ; j++)

{

output += Activation(member.X, ClusterCenters[j, 0], ClusterSpreads[j]) \* Weights[j];

}

member.outY = output;

#endregion

#region Information Collection

LocalError = member.Y - output;

if (Math.Abs(Convert.ToDouble(output)) <= Math.Abs(Convert.ToDouble(member.Y)) \* 1.2 && Math.Abs(Convert.ToDouble(output)) >= Math.Abs(Convert.ToDouble(member.Y)) \* 0.8)

CorrectCount++;

else

FalseCount++;

SumOfError += LocalError;

SumOfSquareError += LocalError \* LocalError;

SumOfOutput += output;

SumOfSquareOutput += output \* output;

SumOfValue += member.Y;

SumOfSquareValue += member.Y \* member.Y;

#endregion

}

SumOfSquareErrorMean = SumOfSquareError / ExtractData.GetDataValidation().Count;

MeanError = SumOfError / ExtractData.GetDataValidation().Count;

ErrorStdDev = Convert.ToDecimal(Math.Sqrt(Convert.ToDouble((SumOfSquareError - (MeanError \* MeanError \* ExtractData.GetDataValidation().Count)) / ExtractData.GetDataValidation().Count)));

SumOfSquareValueMean = SumOfSquareValue / ExtractData.GetDataValidation().Count;

MeanValue = SumOfValue / ExtractData.GetDataValidation().Count;

ValueStdDev = Convert.ToDecimal(Math.Sqrt(Convert.ToDouble((SumOfSquareValue - (MeanValue \* MeanValue \* ExtractData.GetDataValidation().Count)) / ExtractData.GetDataValidation().Count)));

SumOfSquareOutputMean = SumOfSquareOutput / ExtractData.GetDataValidation().Count;

MeanOutput = SumOfOutput / ExtractData.GetDataValidation().Count;

OutputStdDev = Convert.ToDecimal(Math.Sqrt(Convert.ToDouble((SumOfSquareOutput - (MeanOutput \* MeanOutput \* ExtractData.GetDataValidation().Count)) / ExtractData.GetDataValidation().Count)));

}

private void InitializeWeights() // helper for ctor

{

Random rnd = new Random();

Weights = new decimal[numClusters];

for (int i = 0; i < numClusters; i++)

{

Weights[i] = (0.01m - 0.001m) \* Convert.ToDecimal(rnd.NextDouble()) + 0.001m;

}

}

public void AssignClusterCenters(int numClusters)

{

int MinX = -2999879;

int MaxX = 2918494;

int MinY = -19225610;

int MaxY = 30430512;

decimal dividor = 1000000m;

ClusterCenters = new decimal[numClusters,2];

Random rnd = new Random();

// initialize weights and biases to small random values

for (int i = 0; i < numClusters; i++)

{

ClusterCenters[i, 0] = (rnd.Next(MinX, MaxX)) / dividor; //X

ClusterCenters[i,1] = (rnd.Next(MinY, MaxY)) / dividor; //Y

}

}

private decimal Activation(decimal X, decimal Center, decimal spread)

{

return Convert.ToDecimal(Math.Exp(-1 \* Math.Abs(Convert.ToDouble(X -Center)) / Convert.ToDouble(2 \* spread \* spread)));

}

private void InitialCluster()

{

foreach (Dots item in ExtractData.GetDataTrain())

{

decimal Dist = 500000m;

for (int i = 0; i < numClusters; i++)

{

item.DistanceToCentersByX[i] = Utils.SimpleDistance(ClusterCenters[i, 0], item.X);

}

for (int i = 0; i < numClusters; i++)

{

if (item.DistanceToCentersByX[i] < Dist)

{

Dist = item.DistanceToCentersByX[i];

item.BelongingCluster[numClusters-1] = i+1;

}

}

}

ReCalculateCenters();

foreach (Dots item in ExtractData.GetDataValidation())

{

decimal Dist = 500000m;

for (int i = 0; i < numClusters; i++)

{

item.DistanceToCentersByX[i] = Utils.SimpleDistance(ClusterCenters[i, 0], item.X);

}

for (int i = 0; i < numClusters; i++)

{

if (item.DistanceToCentersByX[i] < Dist)

{

Dist = item.DistanceToCentersByX[i];

item.BelongingCluster[numClusters - 1] = i + 1;

}

}

}

}

private void FindSpreads()

{

ClusterSpreads = new decimal[numClusters];

for (int i = 0; i < numClusters; i++)

{

ClusterSpreads[i] = Utils.SimpleDistance(ClusterCenters[i, 0], ExtractData.GetDataTrain().Max(x=> x.X)) / 2;

}

}

private void ReCalculateCenters()

{

for (int i = 0; i < numClusters; i++)

{

decimal count = Convert.ToDecimal(ExtractData.GetDataTrain().Where(x => x.BelongingCluster[numClusters - 1] == i + 1).Count());

decimal summ = ExtractData.GetDataTrain().Where(x => x.BelongingCluster[numClusters - 1] == i + 1).Sum(y => y.X);

if (count != 0)

{

ClusterCenters[i, 0] = summ / count;

}

}

}

public override string ToString()

{

StringBuilder str = new StringBuilder();

str.AppendLine(String.Format("RBF with {0} Epoch, {1} Training Coefficient, {2} Clusters", iterationLimit, trainingCoefficient, numClusters));

str.AppendLine("");

str.AppendLine("MeanError : \t" + MeanError.ToString("#.##") + " ");

str.AppendLine("SumOfSquareErrorMean : \t" + SumOfSquareErrorMean.ToString("#.##") + " ");

str.AppendLine("SumOfSquareError : \t" + SumOfSquareError.ToString("#.##") + " ");

str.AppendLine("SumOfError : \t" + SumOfError.ToString("#.##") + " ");

str.AppendLine("ErrorStdDev : \t" + ErrorStdDev.ToString("#.##") + " ");

str.AppendLine("");

str.AppendLine("MeanValue : \t" + MeanValue.ToString("#.##") + " ");

str.AppendLine("SumOfSquareValue : \t" + SumOfSquareValue.ToString("#.##") + " ");

str.AppendLine("SumOfSquareValueMean : \t" + SumOfSquareValueMean.ToString("#.##") + " ");

str.AppendLine("SumOfValue : \t" + SumOfValue.ToString("#.##") + " ");

str.AppendLine("ValueStdDev : \t" + ValueStdDev.ToString("#.##") + " ");

str.AppendLine("");

str.AppendLine("MeanOutput : \t" + MeanOutput.ToString("#.##") + " ");

str.AppendLine("SumOfSquareOutput : \t" + SumOfSquareOutput.ToString("#.##") + " ");

str.AppendLine("SumOfSquareOutputMean : \t" + SumOfSquareOutputMean.ToString("#.##") + " ");

str.AppendLine("SumOfOutput : \t" + SumOfOutput.ToString("#.##") + " ");

str.AppendLine("OutputStdDev : \t" + OutputStdDev.ToString("#.##") + " ");

str.AppendLine("");

str.AppendLine("Weights : ");

for (int j = 0; j < numClusters; j++)

{

str.AppendLine(String.Format("Weight{0} : \t{1}", j + 1, Weights[j].ToString("#.##")));

}

str.AppendLine("");

str.AppendLine("");

return str.ToString();

}

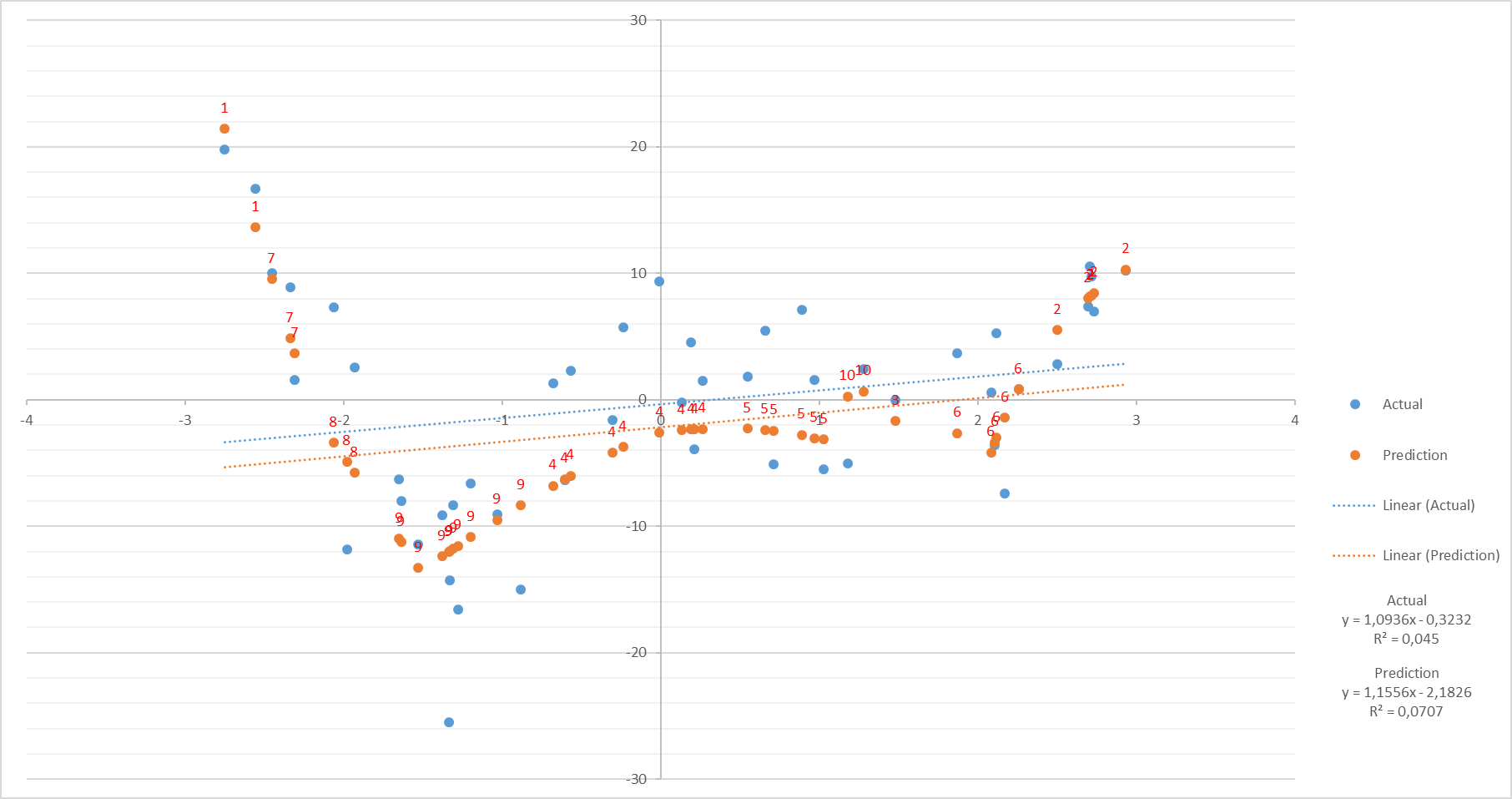
}

# EXPERIMENTS

I implemented 100 experiments, each experiment sends 100,1000,10000 iteration parameter, and with each different iteration there is a combination of 1 to 10 different cluster numbers.

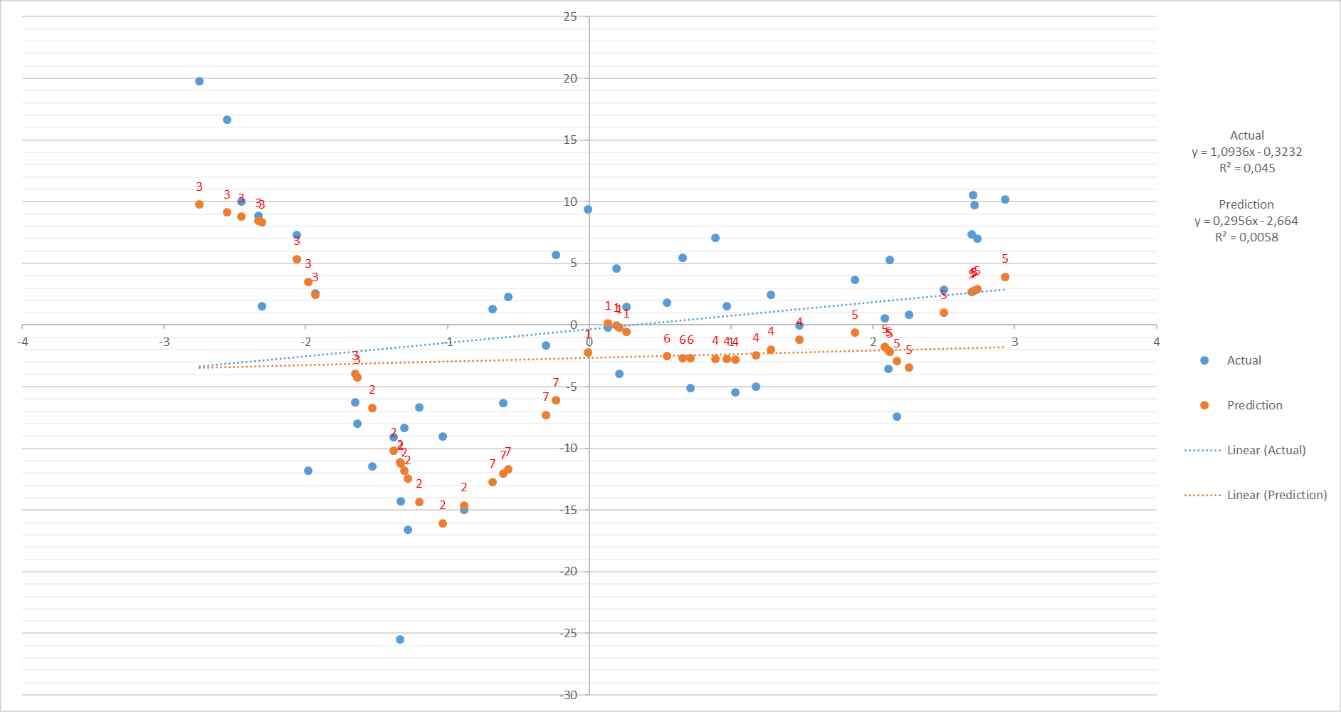
So we have 100\*3\*10 = 3000 result sets. I put some example result sets below. All of them is in the Results.zip file. Red numbers indicate belonging cluster. And there is a md5 hash key at the beginning of each run. This key connects, general results(errors, etc.) and result dataset.

RBF with, 10000 Epoch, 0.1 Training Coefficient, 10 Clusters.



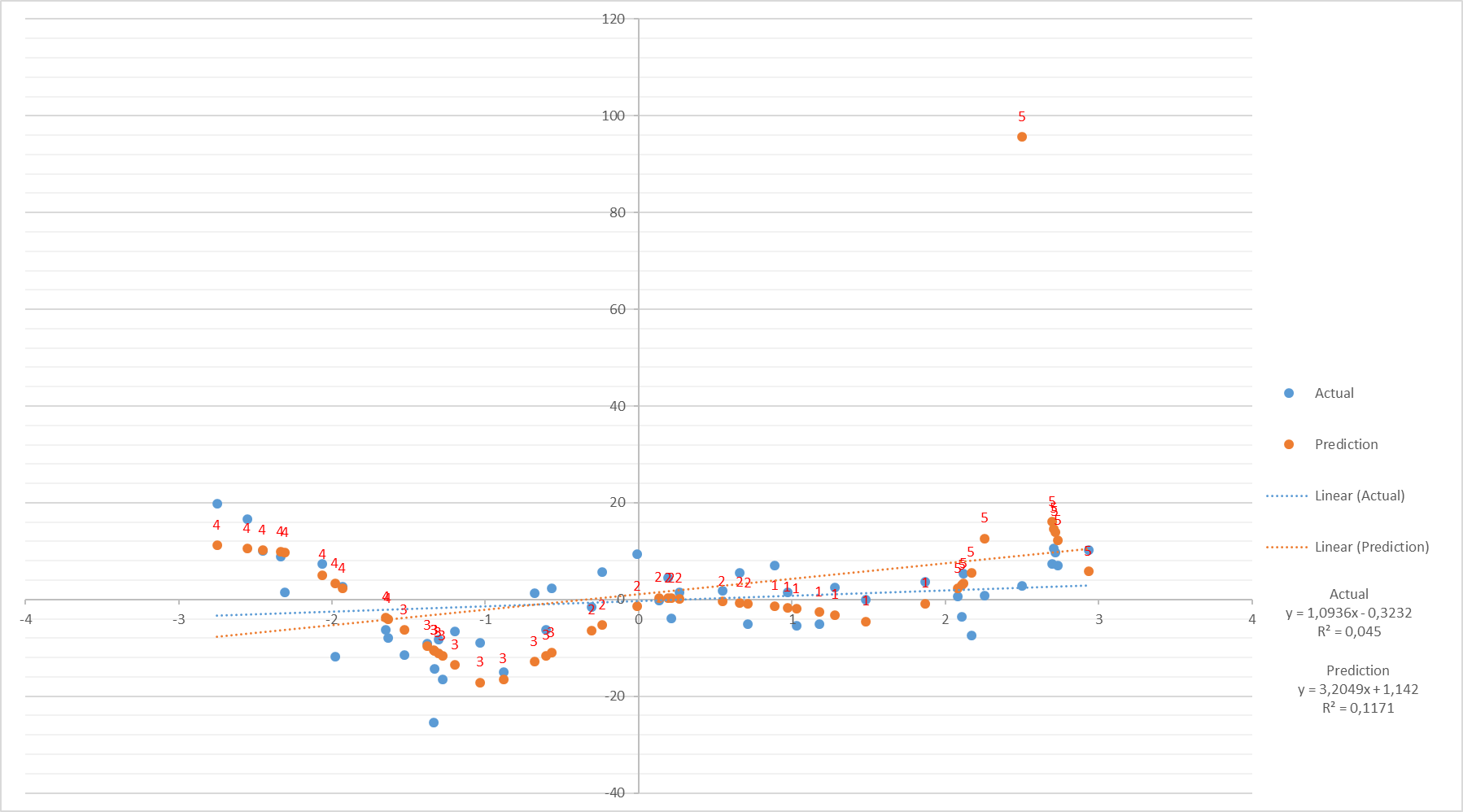
|  |  |  |
| --- | --- | --- |
| Mean Square Error | | 29,36 |
| Mean Error | | 1,86 |
| W1 | 279,58 | |
| W2 | 0,01 | |
| W3 | -9,99 | |
| W4 | 9,84 | |
| W5 | -24,73 | |
| W6 | -9,82 | |
| W7 | -49,35 | |
| W8 | -98,07 | |
| W9 | -136,42 | |
| W10 | 22,21 | |

RBF with, 10000 Epoch, 0.1 Training Coefficient, 7 Clusters.



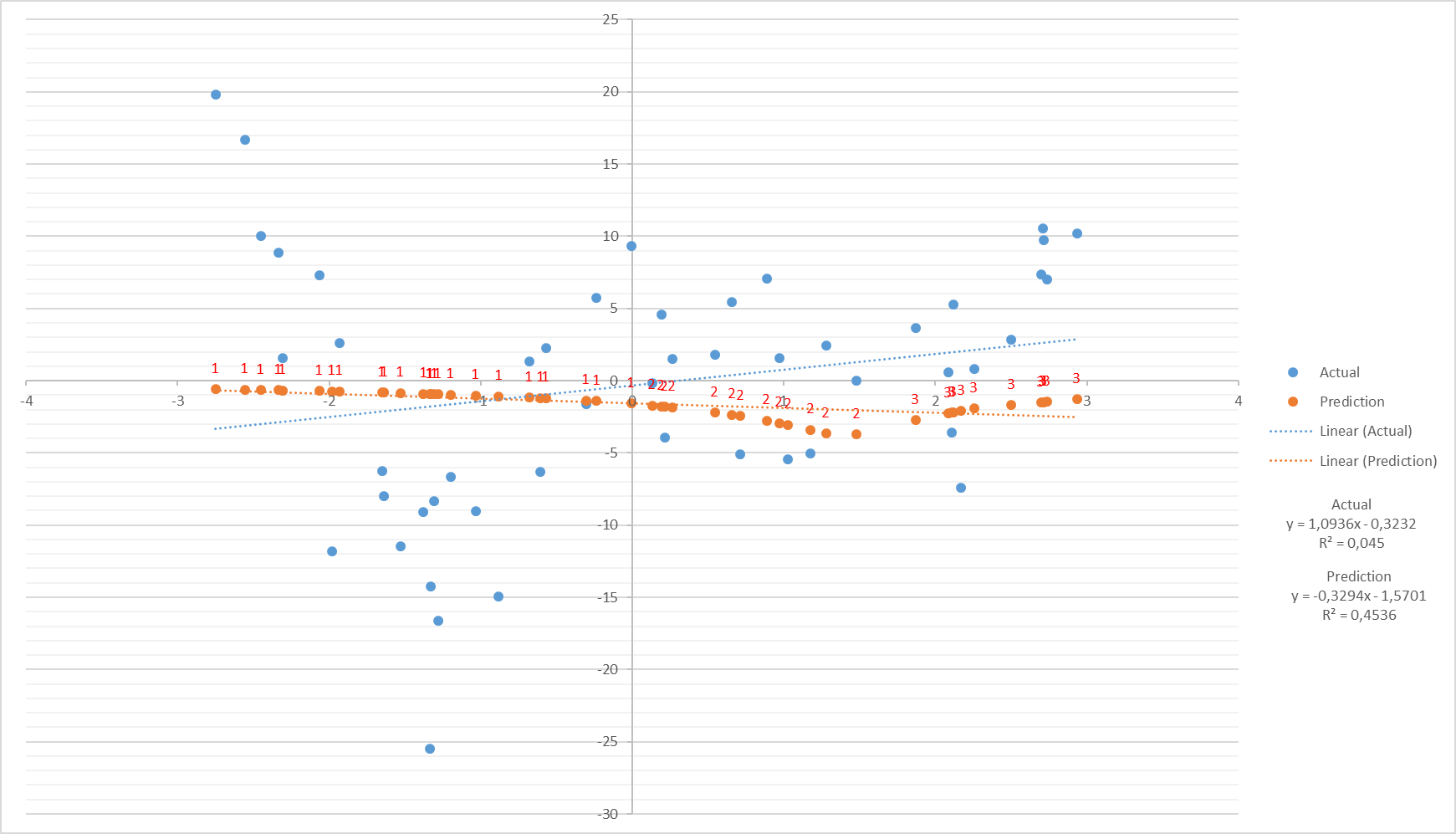
|  |  |
| --- | --- |
| Mean Square Error | 43,21 |
| Mean Error | 2,39 |
| W1 | 45,41 |
| W2 | -132,85 |
| W3 | 122,5 |
| W4 | -5,03 |
| W5 | -6,18 |
| W6 | -10,19 |
| W7 | -24,74 |

RBF with, 10000 Epoch, 0.1 Training Coefficient, 5 Clusters.



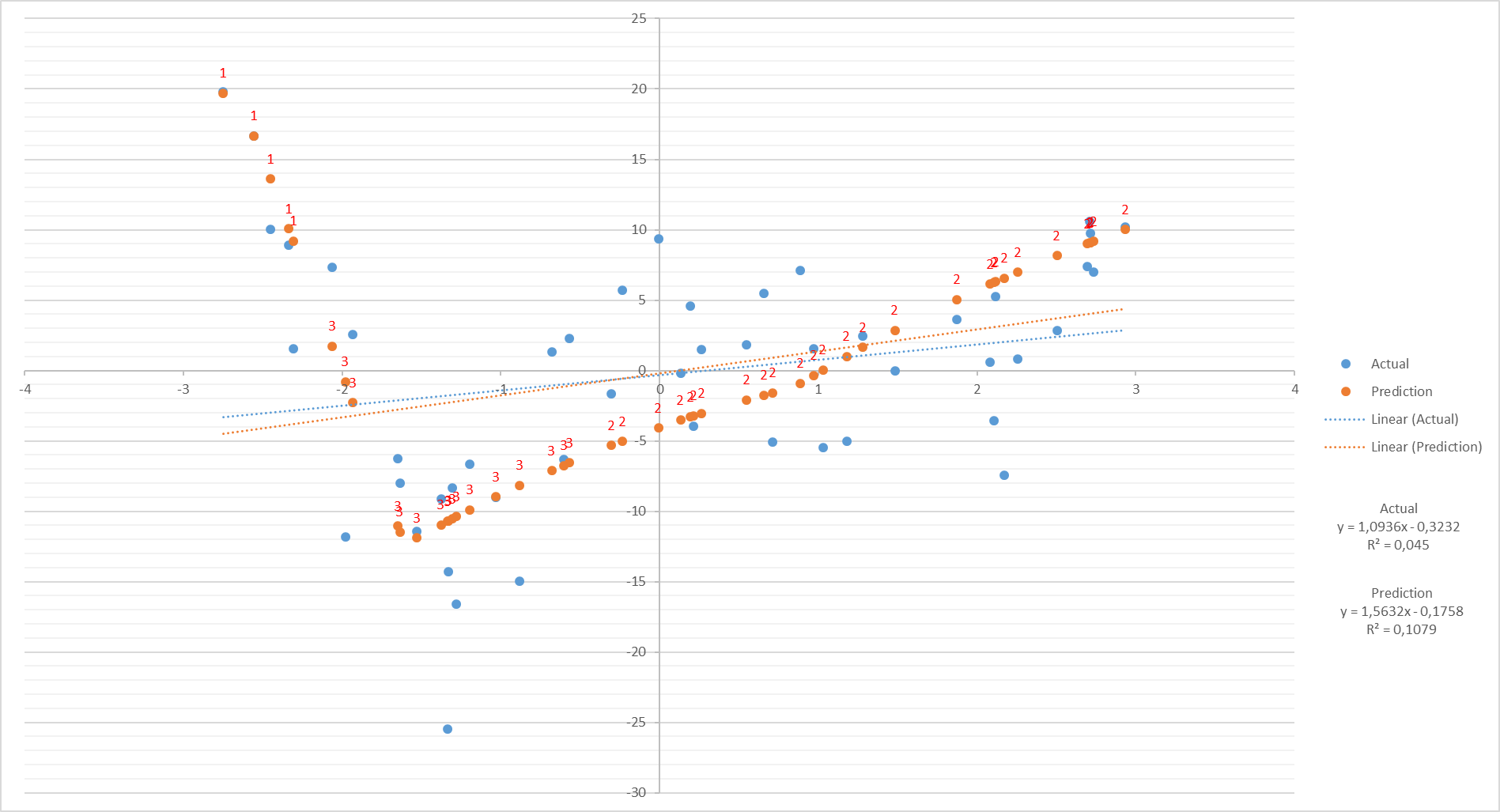
|  |  |
| --- | --- |
| Mean Square Error | 218,36 |
| Mean Error | -1,6 |
| W1 | -9,25 |
| W2 | 34,89 |
| W3 | -151,65 |
| W4 | 118,56 |
| W5 | 115,94 |

RBF with, 10000 Epoch, 0.1 Training Coefficient, 3 Clusters. Bad Prediction



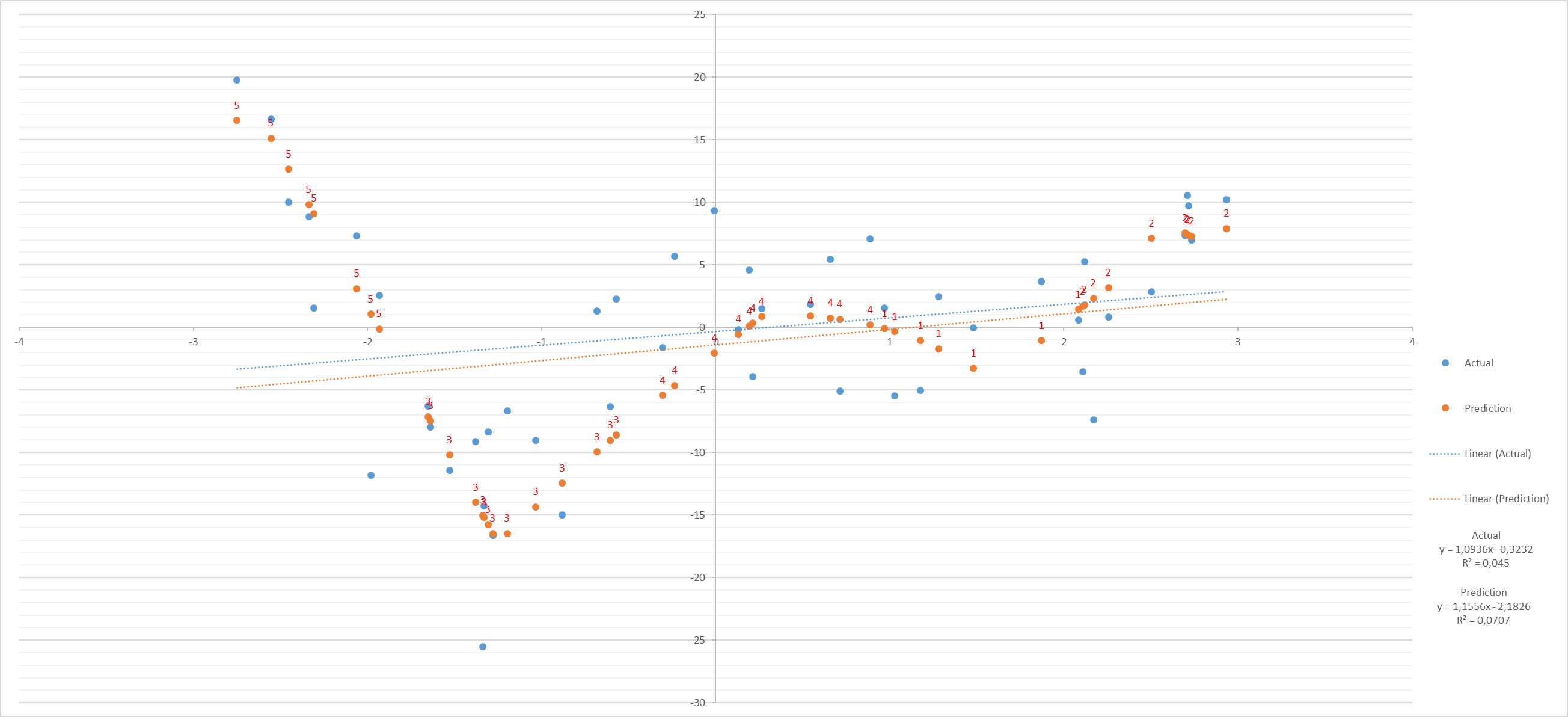
|  |  |
| --- | --- |
| Mean Square Error | 80,12 |
| Mean Error | 1,34 |
| W1 | -0,59 |
| W2 | -3,55 |
| W3 | 0,16 |
|  |  |

RBF with, 10000 Epoch, 0.1 Training Coefficient, 3 Clusters.



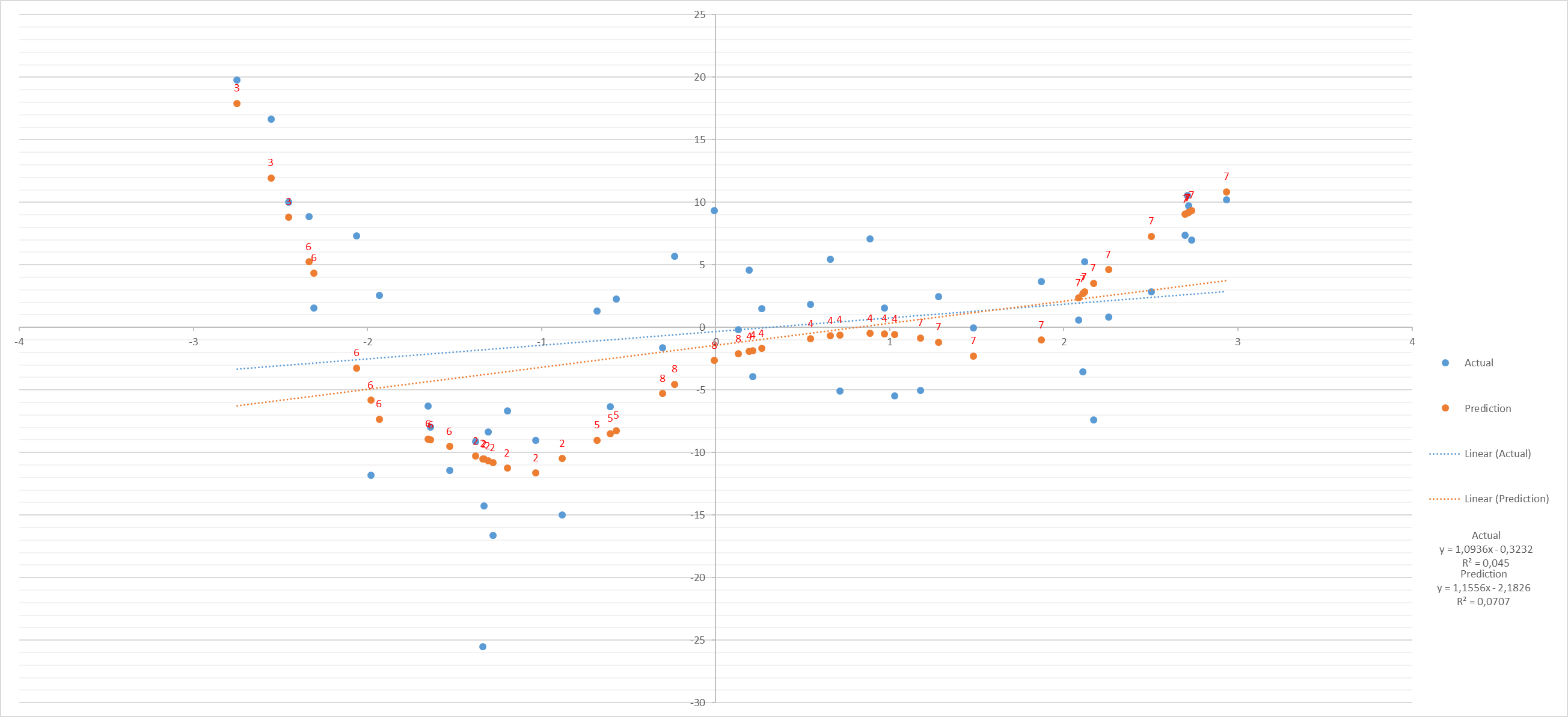
|  |  |
| --- | --- |
| Mean Square Error | 35,93 |
| Mean Error | -0,18 |
| W1 | 194,33 |
| W2 | -4,65 |
| W3 | -192,66 |

RBF with, 1000 Epoch, 0.1 Training Coefficient, 5 Clusters.



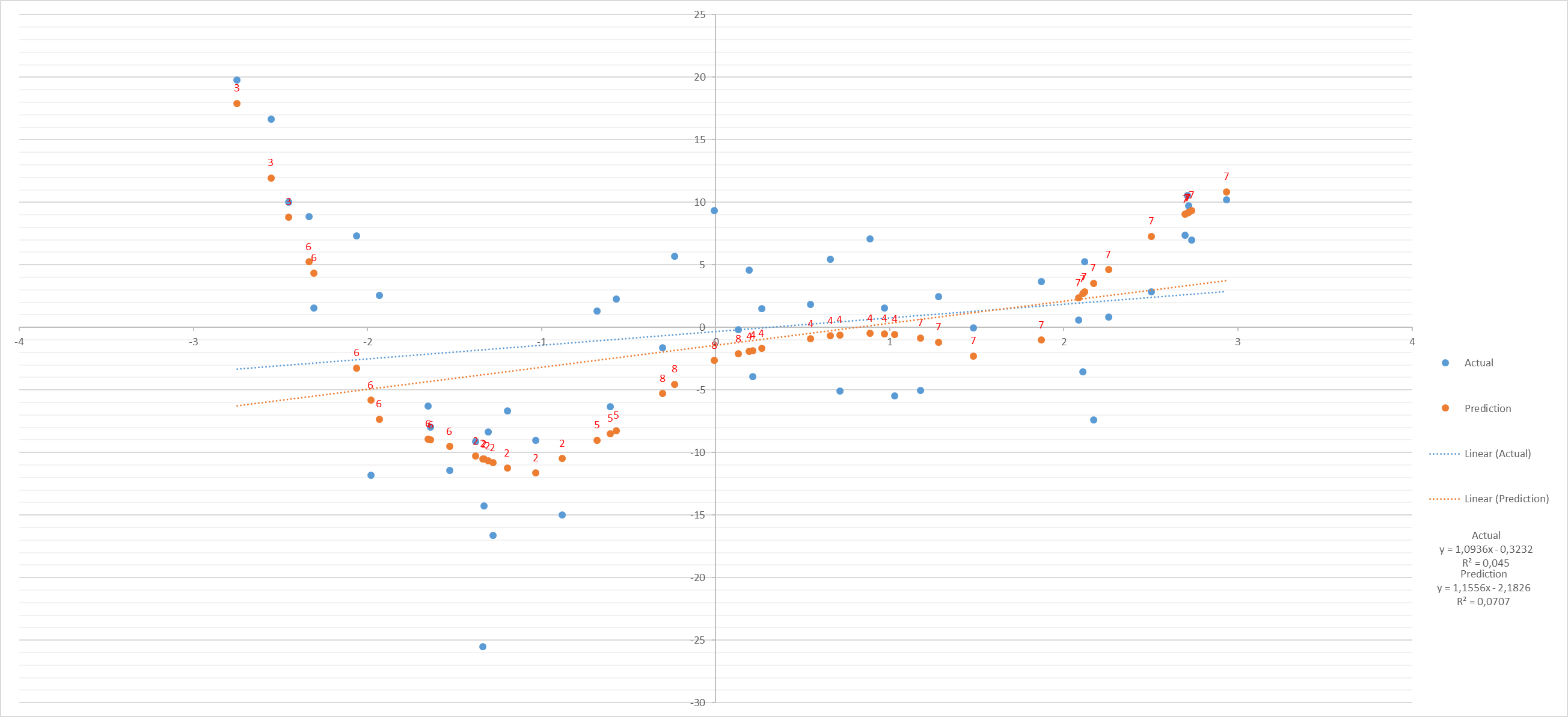
|  |  |
| --- | --- |
| Mean Square Error | 30,26 |
| Mean Error | 1,07 |
| W1 | -11,72 |
| W2 | 7,87 |
| W3 | -167,04 |
| W4 | 18,29 |
| W5 | 151,29 |

RBF with, 1000 Epoch, 0.1 Training Coefficient, 8 Clusters.



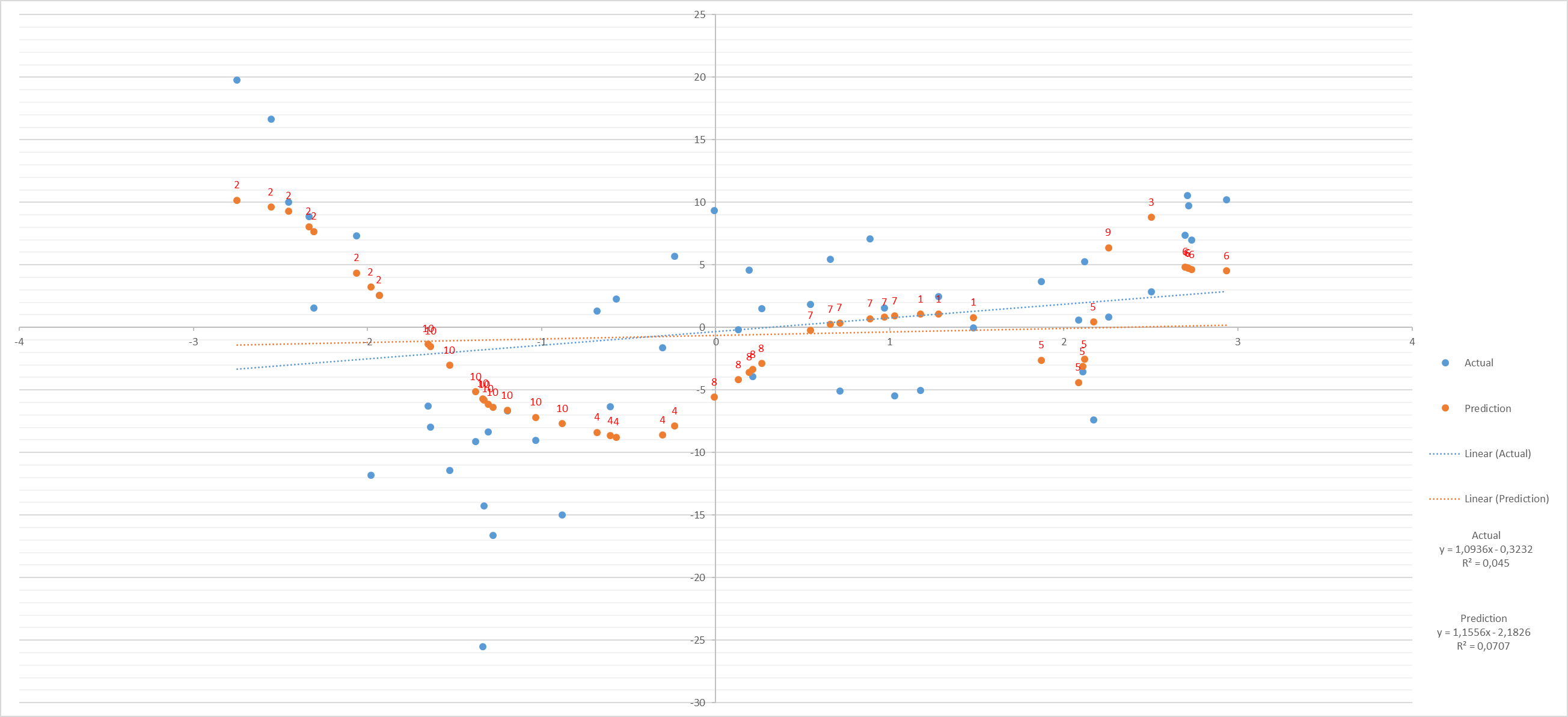
|  |  |
| --- | --- |
| Mean Square Error | 31,97 |
| Mean Error | 1,06 |
| W1 | 116,92 |
| W2 | -53,71 |
| W3 | 95,96 |
| W4 | -1 |
| W5 | -13,92 |
| W6 | -157,61 |
| W7 | -11,99 |
| W8 | 13,76 |

RBF with, 100 Epoch, 0.1 Training Coefficient, 2 Clusters.



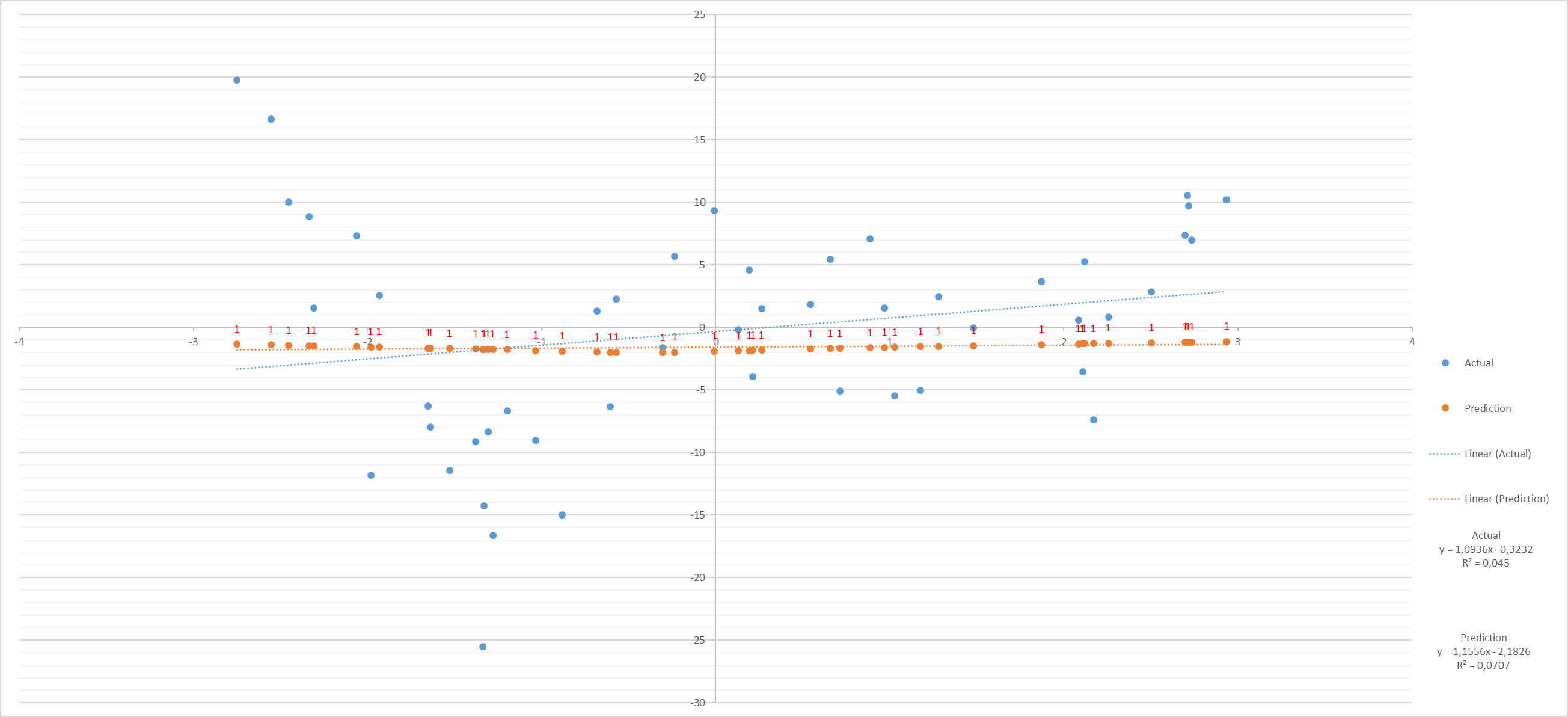
|  |  |
| --- | --- |
| Mean Square Error | 89,95 |
| Mean Error | 1,51 |
| W1 | 13,21 |
| W2 | -19,72 |

RBF with, 100 Epoch, 0.1 Training Coefficient, 10 Clusters.



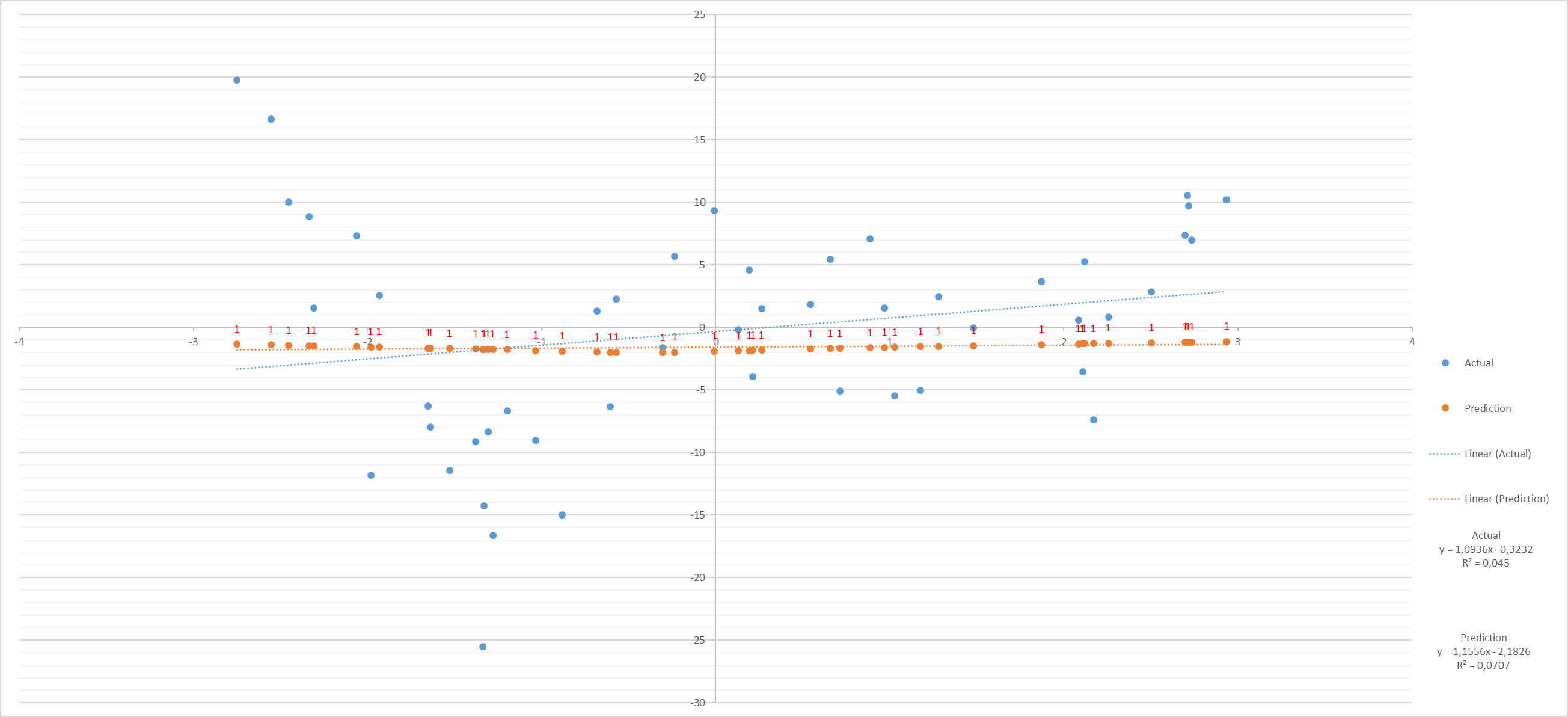
|  |  |
| --- | --- |
| Mean Square Error | 50,36 |
| Mean Error | 0,37 |
| W1 | 1,41 |
| W2 | 74,08 |
| W3 | 7,97 |
| W4 | -37,82 |
| W5 | -11,98 |
| W6 | 0,01 |
| W7 | 9,59 |
| W8 | 0,96 |
| W9 | 10,9 |
| W10 | -48,31 |
|  |  |

RBF with, 10000 Epoch, 0.1 Training Coefficient, 1 Clusters.



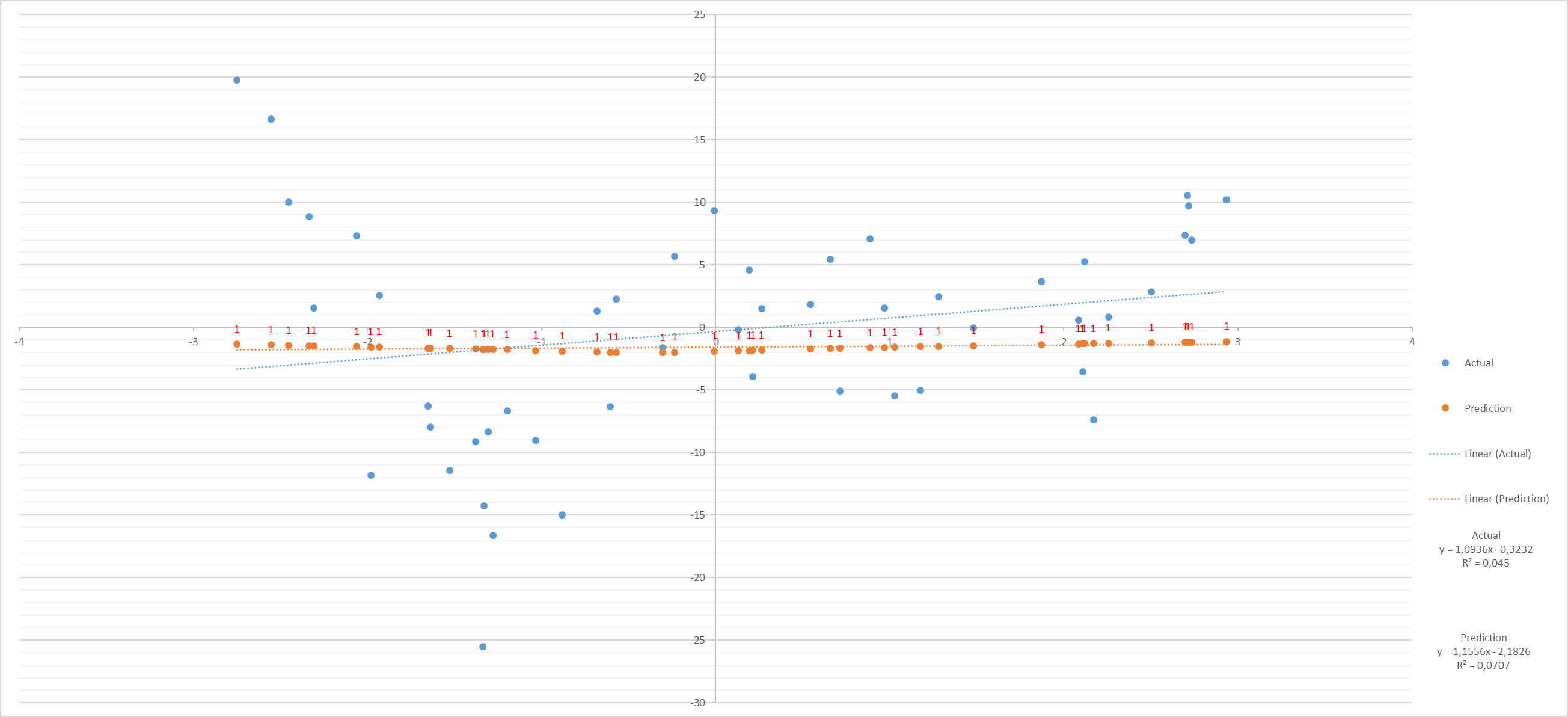
|  |  |
| --- | --- |
| Mean Square Error | 76,47 |
| Mean Error | 1,34 |
| W1 | -2,05 |

RBF with, 100 Epoch, 0.1 Training Coefficient, 1 Clusters.



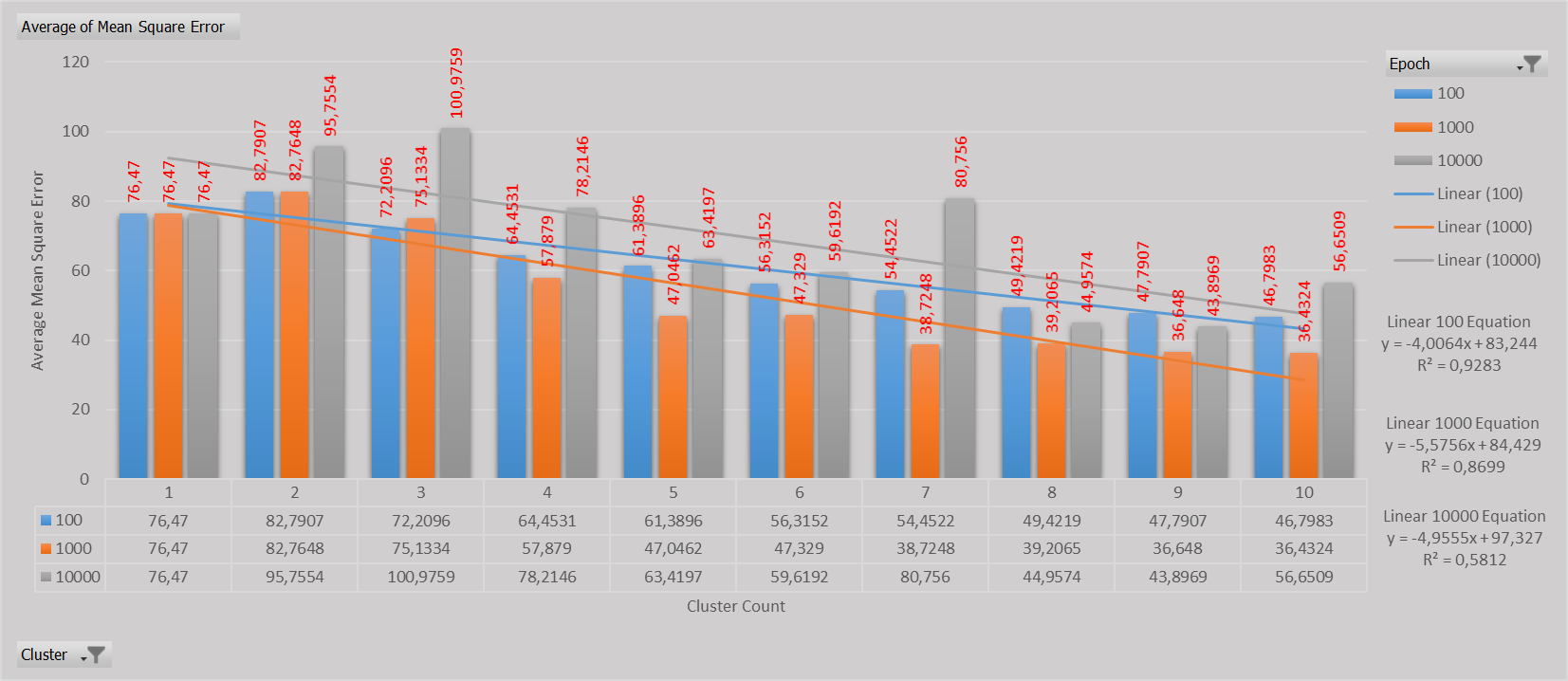
|  |  |
| --- | --- |
| Mean Square Error | 76,47 |
| Mean Error | 1,34 |
| W1 | -2,05 |

RBF with, 1000 Epoch, 0.1 Training Coefficient, 1 Clusters.



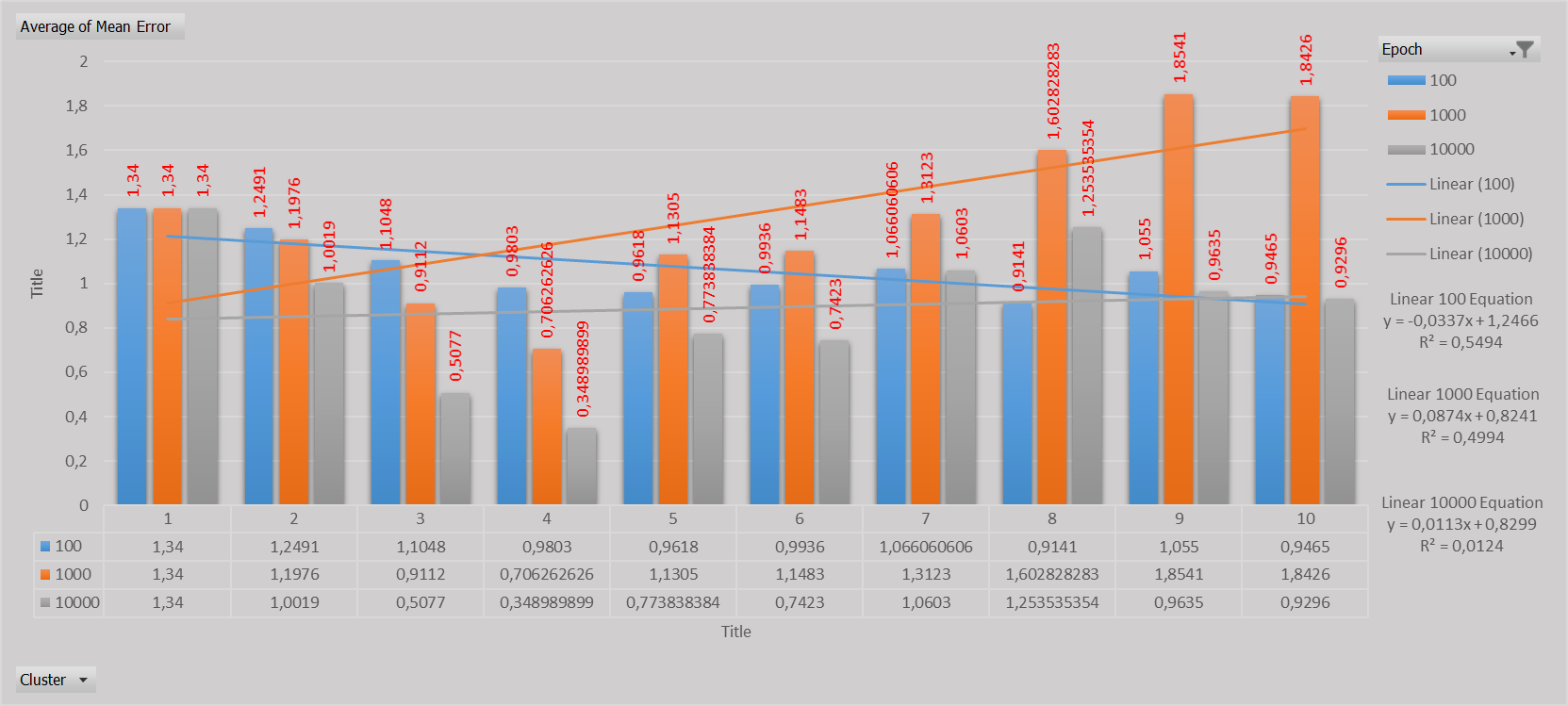
|  |  |
| --- | --- |
| Mean Square Error | 76,47 |
| Mean Error | 1,34 |
| W1 | -2,05 |

Average Mean Square Error 3000 runs (100 restart for each cluster and iteration. 100 x 3 x 10 = 3000)



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Average of Mean Square Error** | **Column Labels** |  |  |  |
| **Row Labels** | **100** | **1000** | **10000** | **Grand Total** |
| 1 | 76,47 | 76,47 | 76,47 | 76,47 |
| 2 | 82,7907 | 82,7648 | 95,7554 | 87,10363333 |
| 3 | 72,2096 | 75,1334 | 100,9759 | 82,77296667 |
| 4 | 64,4531 | 57,879 | 78,2146 | 66,8489 |
| 5 | 61,3896 | 47,0462 | 63,4197 | 57,28516667 |
| 6 | 56,3152 | 47,329 | 59,6192 | 54,42113333 |
| 7 | 54,4522 | 38,7248 | 80,756 | 57,97766667 |
| 8 | 49,4219 | 39,2065 | 44,9574 | 44,5286 |
| 9 | 47,7907 | 36,648 | 43,8969 | 42,77853333 |
| 10 | 46,7983 | 36,4324 | 56,6509 | 46,6272 |
| **Grand Total** | **61,20913** | **53,76341** | **70,0716** | **61,68138** |

Average Mean Error 3000 runs (100 restart for each cluster and iteration. 100 x 3 x 10 = 3000)



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Average of Mean Error** | **Column Labels** |  |  |  |
| **Row Labels** | **100** | **1000** | **10000** | **Grand Total** |
| 1 | 1,34 | 1,34 | 1,34 | 1,34 |
| 2 | 1,2491 | 1,1976 | 1,0019 | 1,149533333 |
| 3 | 1,1048 | 0,9112 | 0,5077 | 0,841233333 |
| 4 | 0,9803 | 0,706263 | 0,34899 | 0,679530201 |
| 5 | 0,9618 | 1,1305 | 0,773838 | 0,955986622 |
| 6 | 0,9936 | 1,1483 | 0,7423 | 0,9614 |
| 7 | 1,066060606 | 1,3123 | 1,0603 | 1,146488294 |
| 8 | 0,9141 | 1,602828 | 1,253535 | 1,255671141 |
| 9 | 1,055 | 1,8541 | 0,9635 | 1,290866667 |
| 10 | 0,9465 | 1,8426 | 0,9296 | 1,239566667 |
| **Grand Total** | **1,061121121** | **1,30487** | **0,892467** | **1,086209085** |

# CONCLUSIONS

Dataset was prety easy and easly converged even 100 iteration versions. But when it comes to single cluster. Result is always same independent from iteration. As seen in graphics, most of the linear regression lines is very similar between actual and result datasets.

As we can see in the error graphs, If we increase cluster count and iteration we mostly decrease mean square error, we can see the trend lines is decreasing seperately. On the other hand we see clearly 1000 iteration increases mean error, and more cluster increases mean error significantly in 1000 iterations. But at 100 iterations trend is tend to decrease very slightly if we continue to increase cluster numbers. At 10000 iterations’ show us mean error trend is almost 0 aim. We can presume from here if we overfit the data by running 10000 iterations, average mean error doesn’t change by cluster count.

All of the code and other files can be found in : <https://github.com/BrscnTnl/ANN_Homework_2>