

NEC - Exercise 1

Prediction with Back-Propagation and Linear Regression

Carlos García

January 2017

Contents

1	Main concepts	4
1.1	Back-propagation	4
1.2	Multiple Linear Regression	5
2	Back-Propagation (BP)	6
2.1	Considerations about my implementation	6
2.2	Trials performed	6
2.3	Final parameters	11
3	Multiple Linear Regression (MLR)	13
4	Comparison between results	19

Pràctica 1: Prediction with Back-Propagation and Linear Regression

Objective

Prediction of the power of the turbine of a hydro-electrical plant, using the following algorithms:

- Back-Propagation (BP), implemented by the student
- Multiple Linear Regression (MLR), using free software

Data

- File: turbine.txt
- Columns: 4 variables, 1 value to predict
 - Variables:
 - * Height above sea level
 - * Fall
 - * Net fall
 - * Flux
 - Prediction:
 - * Power of the turbine
- Patterns: 451 patterns
 - Training and Validation (and cross-validation): the first 401 patterns
 - Test: the remaining 50 patterns

1 Main concepts

1.1 Back-propagation

The Back-propagation is a common method of training in machine learning and artificial neural networks, it is usually used (like in this case) in conjunction with an optimization method such as gradient descent. This algorithm consists of two phases, one is a propagation and the other is a weight update, then the process is basically repeat this cycle as many times as necessary.

The discovery of the error back-propagation algorithm to train multilayer networks from examples has proved to be an excellent tool in classification, interpolation and prediction tasks. In fact, it has been proved that any sufficiently well-behaved function can be approximated by a neural network provided the number of units is large enough[?].

The motivation for developing the back-propagation algorithm was to find a way to train a multi-layered neural network such that it can learn the appropriate internal representations to allow it to learn any arbitrary mapping of input to output [2].

The main difficulties when using the back-propagation algorithm are those related to setup the network and try to find the best neurons distribution and settings values which result in a good behavior.

Back-propagation is a very popular neural network learning algorithm because it is conceptually simple, computationally efficient, and because it often works. However, getting it to work well, and sometimes to work at all, can seem more of an art than a science. Designing and training a network using back-prop requires making many seemingly arbitrary choices such as the number and types of nodes, layers, learning rates, training and test sets, and so forth. These choices can be critical, yet there is no foolproof recipe for deciding them because they are largely problem and data dependent [3].

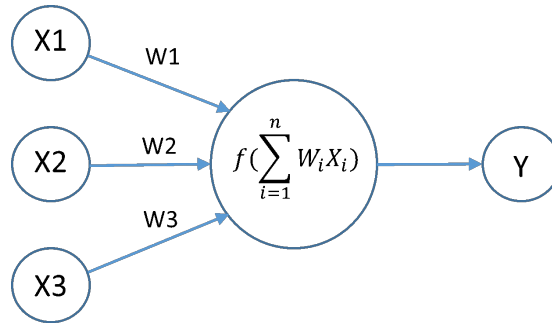


Figure 1: A simple neural network with three input units, one hidden layer with one unit and one output unit.

1.2 Multiple Linear Regression

In statistics, linear regression is an approach for modeling the relationship between a scalar dependent variable y and one or more explanatory variables (or independent variables) denoted X . The case of one explanatory variable is called simple linear regression. For more than one explanatory variable, the process is called multiple linear regression [4].

The very simplest case of a single scalar predictor variable x and a single scalar response variable y is known as simple linear regression. The extension to multiple and/or vector-valued predictor variables (denoted with a capital X) is known as multiple linear regression, also known as multivariable linear regression [5].

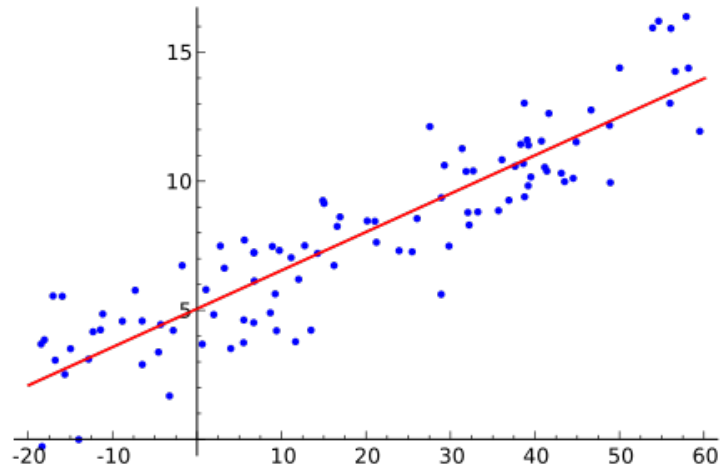


Figure 2: Regression analysis

2 Back-Propagation (BP)

2.1 Considerations about my implementation

There are several things to take in consideration at the moment of an implementation, in my case, the first important decision was the programming language and the structure of the code. As I'm trying to improve my Python skills, I tried to get advantage of this project to practice. When using Python, is a good idea using the Numpy library and matrices operations in order to perform faster computations.

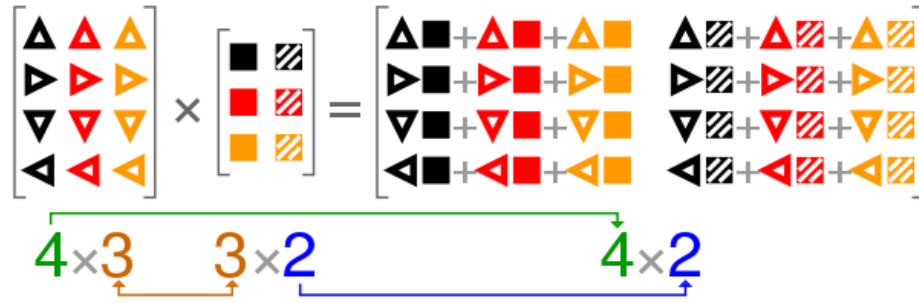


Figure 3: Multiplication of two matrices illustrated with a 43 and a 32 matrix with arbitrary symbols [6]

2.2 Trials performed

The first trials I did were specially focused on changing the momentum and learning rate (scalar) parameters, after a couple of tests it was evident that better values for momentum were near to 0.9 and for the learning rate (scalar) it was near to 0.1. But at this moment it was too early to be sure so I left this values and kept working changing another parameters (Figure 4).

```
12 NN.scalar = 0.1
13 NN.momentum = 0.9
```

Figure 4: Scalar and Momentum values in the Python code

Initially the network was configure for batch back-propagation, but then I wanted to check the improvements in the learning output when decreasing the size of the batch until 1. It was clear that for this problem, an on-line back-propagation worked much more efficiently, so I left it like this (Figure 5).

```

14 NN.batchSize = 1    # 0 means entire data
15
16

```

Figure 5: Batch size value in the Python code, one means on-line back-propagation.

After this I started to compare the behaviors of the network using different architectures, like including new hidden layers and increasing-decreasing the number of neurons in them (Figure 6). Personally, I think it was the most difficult setting because there are too much options and each of them can return very unexpected results, so I decided to create another Python script to go over different architectures and plot the learning behavior just to compare (Figure 7).

```

5  # Network architecture
6  NN.inputLayerSize = 4
7  NN.outputLayerSize = 1
8  NN.numberHiddenLayer = 2
9  NN.hiddenLayerSizes = [9,3]
10 #

```

Figure 6: Sample network architecture.

```

13 iniNN = copy.deepcopy(NN)
14 # -----
15 # Default settings
16 for i in range(1,21):
17     for j in range(1,21):
18         if False:
19             # -----
20             includeTraining = True
21             includeValidate = True
22             if includeTraining:
23                 costs, results, percsError = NN.trainNetwork(NN.epochs, './input/turbine-input-train')
24                 NN.plotTraining(costs, percsError, True, False) # showPlot, savePlot
25                 # NN.plotValidate(results)
26             if includeValidate:
27                 # Validate learning using new unknown set
28                 print "-----"
29                 print "Now validating"
30                 results = NN.validateLearning('./input/turbine-input-validate')
31                 NN.plotValidate(results, True, False) # showPlot, savePlot
32             # -----
33             # Changes before next loop
34             NN = copy.deepcopy(iniNN)
35

```

Figure 7: Python script for testing different network architectures.

In the Figure 8 we can see a summary of all network architecture tests I did when using the script. Note that the red square shows the structure of the layers

for that test, the first number before the colon symbol (:), tells the number of layers we have, and the following numbers are the numbers of neurons in every layer from input to output. Take into account that those values separated with a dash (-) denote the values tested for the iterative layer.

Name	Size	Type
best-ones	12 items	Folder
lay[3:4,1-20,1] sca[0.50] mom[0.5] bat[1] epo[50000]	40 items	Folder
lay[4:4,2,2-19,1] sca[0.50] mom[0.5] bat[1] epo[50000]	36 items	Folder
lay[4:4,3,2-19,1] sca[0.50] mom[0.5] bat[1] epo[50000]	36 items	Folder
lay[4:4,4,2-19,1] sca[0.50] mom[0.5] bat[1] epo[50000]	36 items	Folder
lay[4:4,5,2-19,1] sca[0.50] mom[0.5] bat[1] epo[50000]	36 items	Folder
lay[4:4,6,2-19,1] sca[0.50] mom[0.5] bat[1] epo[50000]	36 items	Folder
lay[4:4,7,2-20,1] sca[0.50] mom[0.5] bat[1] epo[50000]	38 items	Folder
lay[4:4,8,2-20,1] sca[0.50] mom[0.5] bat[1] epo[50000]	38 items	Folder
lay[4:4,9,2-20,1] sca[0.50] mom[0.5] bat[1] epo[50000]	38 items	Folder
lay[4:4,10,2-20,1] sca[0.50] mom[0.5] bat[1] epo[50000]	38 items	Folder
lay[4:4,11,2-20,1] sca[0.50] mom[0.5] bat[1] epo[50000]	38 items	Folder
lay[4:4,12,2-20,1] sca[0.50] mom[0.5] bat[1] epo[50000]	38 items	Folder
lay[4:4,13,2-20,1] sca[0.50] mom[0.5] bat[1] epo[50000]	38 items	Folder
lay[4:4,14,2-20,1] sca[0.50] mom[0.5] bat[1] epo[50000]	38 items	Folder
lay[4:4,15,2-20,1] sca[0.50] mom[0.5] bat[1] epo[50000]	38 items	Folder
lay[4:4,16,2-20,1] sca[0.50] mom[0.5] bat[1] epo[50000]	38 items	Folder
lay[4:4,17,2-20,1] sca[0.50] mom[0.5] bat[1] epo[50000]	38 items	Folder
terminal-logger	199,3 kB	Text

Figure 8: Plots folder created after iterate architectures.

Next we can check one of this folders inside, just to have an idea of the results, in this case we are showing the folder "lay[3:4,1-20,1]..." (Figure 9):

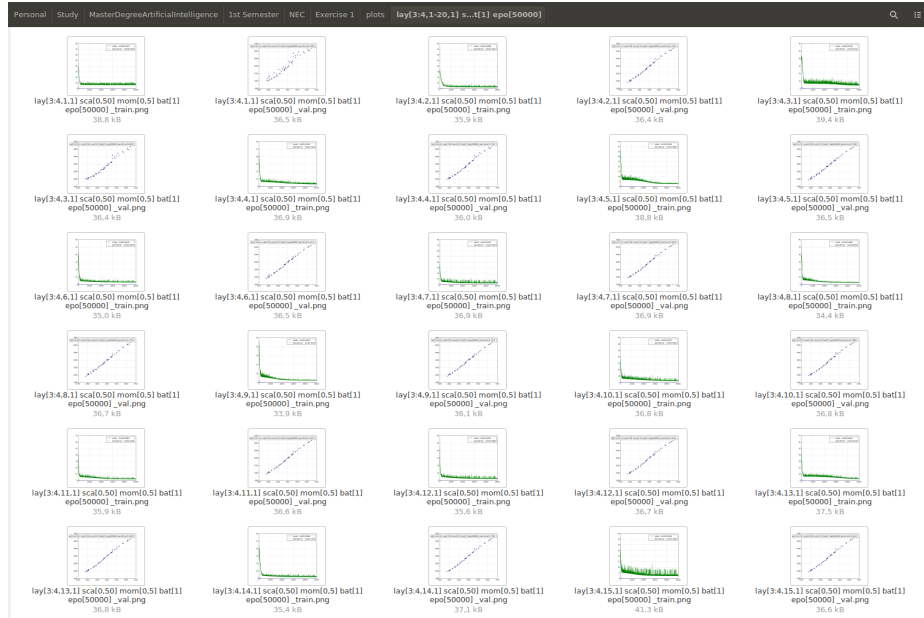


Figure 9: Sample plot folder results.

Finally, I did a manual selection for the best output results, and putted them into the folder "best-ones" (Figures 10 and 11).

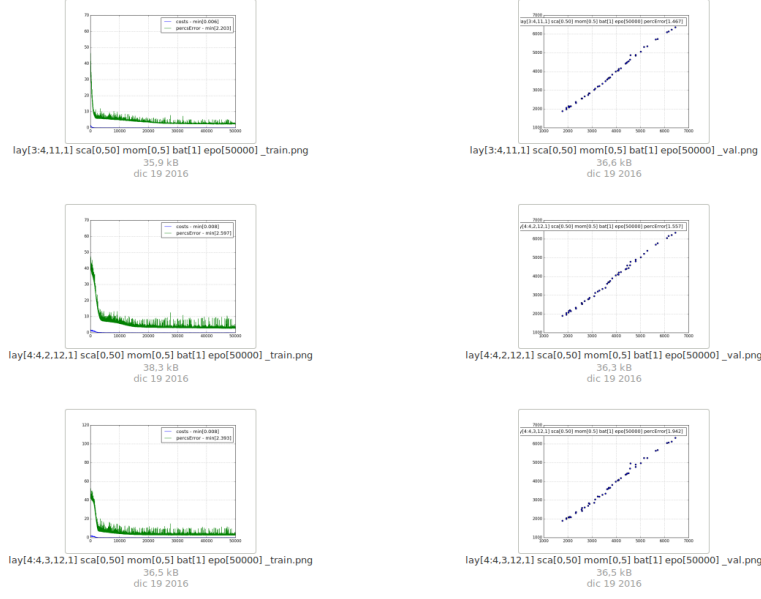


Figure 10: Best result plots (1/2)

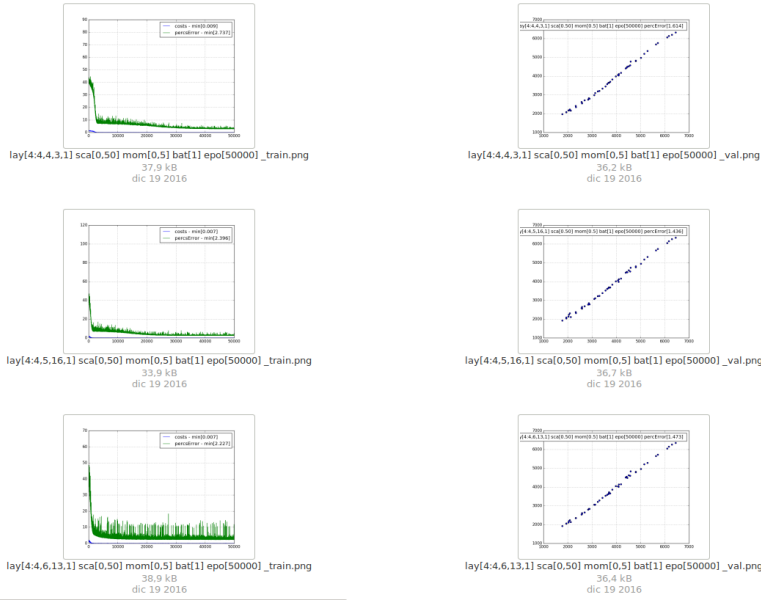


Figure 11: Best result plots (2/2)

2.3 Final parameters

After analyzing several results I decided to select the following values:

- Learning rate: 0.3
- Momentum: 0.9
- Batch size: 1 (on-line)
- Number of epochs: 1 million
- Cross validation while training, always comparing with last 50 rows of training set and learning with first 351 rows.
- Architecture: 4 layers
 - Layer 1: 4 neurons (input)
 - Layer 2: 9 neurons (hidden layer)
 - Layer 3: 3 neurons (hidden layer)
 - Layer 4: 1 neuron (output)

And my final results with this configuration were the following:

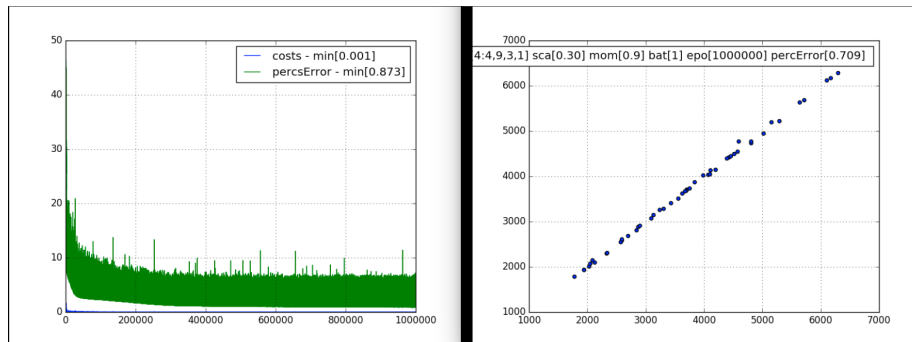



Figure 12: Learning progress and percentage error while validating.



The figure consists of three screenshots of a Notepad++ text editor, each displaying a list of numerical values representing final weights after a learning process. The windows are titled 'weight_0', 'weight_1', and 'weight_2'.

weight_0

```

1.938671474488361968e+00 -1.388051647982515791e+00 -6.045282771441463643e-01
-1.256913361066023693e+00 -3.391811135928365917e-01 -3.107337788221977259e+00
1.517562618441432587e+00 -8.862033964830591248e-01 1.799115143540156725e-01
7.058172245613437978e-02 7.523518300940088555e-01 -2.131660412015444095e+00
-3.808683802096148874e-01 -1.192531838542504774e+00 4.950958372453708733e-01
-1.347432834748305108e+00 -2.358446710039074357e-01 -1.685788762981181410e+00
3.749119251391638263e-01 -9.390310435296819747e-02 -1.166145334684336676e+00
1.032227460529536733e+00 -1.374241962953466345e-01 -1.173436940558151148e-01
7.064986690054985141e-01 -8.020658608965499647e-01 -7.956902278909029347e-01
-1.756058414374621846e+00 -1.088847978939401440e+00 1.127418305301672724e-01
-1.206549551740511372e+00 -2.771264197746772240e+00 1.384914463679180231e-02
-1.078243579082831971e+00 -2.281544982427848312e-01 3.049404375252402044e+00

```

weight_1

```

-2.002500820295084694e+00 -3.406927462287617780e-01 2.274998978038622610e+00
2.430764487436631316e-01 1.381735745596278431e+00 3.909156781267130731e-01
5.300129891156853956e-01 -1.445412783635982301e-01 -4.404439903036774795e-01
7.822367949201616266e-01 1.849364375021922635e+00 7.435092666178396703e-01
2.562595697169385289e+00 1.144344934851116102e+00 9.232744355447177753e-01
1.462652480030289581e+00 4.915770278746814270e+00 2.396046884137144062e-01
-1.280028074280330275e+00 4.657816620996844126e-01 5.136966894066879313e-02
1.704176470537702492e+00 2.144531932914636396e+00 2.116701061335722223e+00
-3.655891682633955142e+00 -3.628015601195885065e+00 -3.123381335772755230e-01

```

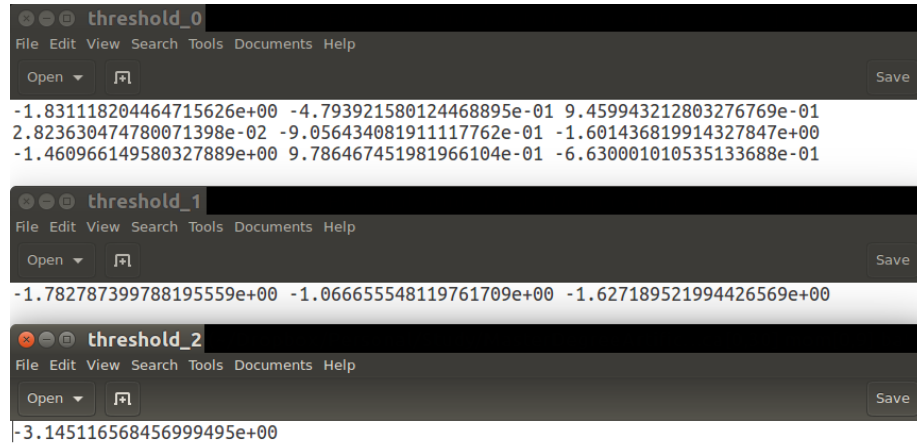
weight_2

```

-4.936179554728636454e+00
-5.128158585183546947e+00
2.118359942782483962e+00

```

Figure 13: Final weights after learning process.



The figure consists of three screenshots of a Notepad++ text editor, each displaying a list of numerical values representing final thresholds after a learning process. The windows are titled 'threshold_0', 'threshold_1', and 'threshold_2'.

threshold_0

```

-1.831118204464715626e+00 -4.793921580124468895e-01 9.459943212803276769e-01
2.823630474780071398e-02 -9.05643408191117762e-01 -1.601436819914327847e+00
-1.460966149580327889e+00 9.786467451981966104e-01 -6.630001010535133688e-01

```

threshold_1

```

-1.782787399788195559e+00 -1.066655548119761709e+00 -1.627189521994426569e+00

```

threshold_2

```

-3.145116568456999495e+00

```

Figure 14: Final thresholds after learning process.

3 Multiple Linear Regression (MLR)

For this project I did a small research about good libraries and softwares for machine learning and regression algorithms, according to several posts ([8], [9] and [10]), Weka is a very good option and it uses Java so I think is a good choice because of performance and multi platform abilities.

Weka is a collection of machine learning algorithms for data mining tasks. The algorithms can either be applied directly to a dataset or called from your own Java code. Weka contains tools for data pre-processing, classification, regression, clustering, association rules, and visualization. It is also well-suited for developing new machine learning schemes [7].

Weka supports several standard data mining tasks, more specifically, data preprocessing, clustering, classification, regression, visualization, and feature selection. All of Weka's techniques are predicated on the assumption that the data is available as one flat file or relation, where each data point is described by a fixed number of attributes (normally, numeric or nominal attributes, but some other attribute types are also supported) [10].



Figure 15: Weka logo, featuring weka, a bird endemic to New Zealand [7]

In order to use the software I first had to format our training and validating data like this:

	A	B	C	D	E
1	input1	input2	input3	input4	output
2	624	89.16	89.765	3.5	2512.85
3	628	93.16	93.765	3.5	2583.79
4	602	67.84	66.415	6.5	3748.77
5	599	64.84	63.415	6.5	3520.65
6	630	94.69	93.54	8	6673.84
7	620	84.89	84.665	6	4533.31
8	601	66.89	65.665	6	3402.31
9	622	88.21	87.935	3	2029.03
10	599	64.99	64.165	5	2702.84
11	611	75.89	75.665	6	4047.41
12	591	56.94	55.92	5.5	2506.19
13	591	56.99	56.165	5	2272.25
14	598	63.94	62.92	5.5	2930.92
15	618	83.04	83.365	4.5	3196.87
16	609	73.89	73.665	6	3934.63
17	608	72.79	72.14	7	4498.27
18	609	75.21	74.935	3	1753.19
19	614	79.04	79.365	4.5	3050.21
20	628	92.79	92.14	7	5785.19
21	608	72.94	72.92	5.5	3533.24
22	625	89.69	88.54	8	6318.53
23	603	69.1	68.55	4	2260.59
24	611	75.84	75.415	6.5	4386.78
25	620	84.69	83.54	8	5950.95
26	628	92.84	92.415	6.5	5370.96
27	617	82.16	82.765	3.5	2342.83
28	597	63.04	62.365	4.5	2315.89
29	626	90.89	90.665	6	4838.78
30	595	60.84	59.415	6.5	3229.58
31	627	92.1	92.55	4	3033.93
32	598	63.79	62.14	7	3664.19
33	625	91.21	90.935	3	2069.21
34	622	86.74	85.84	7.5	5775.22
35	630	94.84	94.415	6.5	5471.25
36	612	76.74	75.84	7.5	5062.02
37	615	79.74	78.84	7.5	5286.39
38	608	74.21	73.935	3	1729.06
39	623	87.74	86.84	7.5	5843.72
40	599	64.89	63.665	6	3263.81
41	596	61.79	60.14	7	3511.37
42	611	76.16	76.765	3.5	2176.83
43	624	89.1	89.55	4	2956.54
44	591	56.79	55.14	7	2915.86
45	625	89.74	88.84	7.5	5975.71
46	612	76.79	76.14	7	4768.88
47	593	58.79	57.14	7	3243.23
48	602	67.89	66.665	6	3474.88
49	601	67.1	66.55	4	2182.38

	A	B	C	D	E
1	input1	input2	input3	input4	output
2	620	84.99	85.165	5	3698.29
3	595	61.1	60.55	4	1935.13
4	629	93.94	93.92	5.5	4509.67
5	617	81.69	80.54	8	5708.99
6	591	57.04	56.365	4.5	2024.05
7	617	81.94	81.92	5.5	3981.74
8	623	87.69	86.54	8	6171.22
9	627	91.94	91.92	5.5	4427.36
10	609	74.16	74.765	3.5	2117.9
11	594	59.94	58.92	5.5	2692.87
12	603	68.94	67.92	5.5	3237.11
13	629	94.04	94.365	4.5	3558.33
14	592	57.99	57.165	5	2327.54
15	603	68.79	67.14	7	4063.35
16	614	78.94	78.92	5.5	3838.09
17	617	81.59	79.84	9	6098.9
18	625	90.04	90.365	4.5	3432.48
19	623	89.21	88.935	3	2042.36
20	620	85.1	85.55	4	2841.89
21	603	68.64	66.19	8.5	4572.62
22	626	90.94	90.92	5.5	4384.97
23	630	94.74	93.84	7.5	6297.76
24	603	69.04	68.365	4.5	2589.43
25	592	57.94	56.92	5.5	2569.39
26	602	67.74	65.84	7.5	4192.01
27	605	70.89	69.665	6	3692.94
28	626	92.21	91.935	3	2081.66
29	604	69.59	66.84	9	4591.21
30	629	94.16	94.765	3.5	2590.7
31	612	76.89	76.665	6	4102.62
32	602	67.99	67.165	5	2865.92
33	630	95.1	95.55	4	3090.96
34	599	64.69	62.54	8	4096.06
35	617	83.21	82.935	3	1938.47
36	605	70.64	68.19	8.5	4801.54
37	621	86.04	86.365	4.5	3299.93
38	601	66.84	65.415	6.5	3670.99
39	599	64.79	63.14	7	3743.56
40	597	62.89	61.665	6	3128.97
41	610	76.21	75.935	3	1776.96
42	608	72.59	70.84	9	5153.46
43	607	71.64	70.19	8.5	5016.03
44	604	69.89	68.665	6	3618.87
45	614	78.64	77.19	8.5	5632.13
46	612	76.69	75.54	8	5283.22
47	605	71.1	70.55	4	2337.68
48	621	85.59	83.84	9	6451.24
49	607	71.69	70.54	8	4806.83

Figure 16: Format train and validate data for Weka.

Then, we must first open Weka software and upload the training file into the main window:

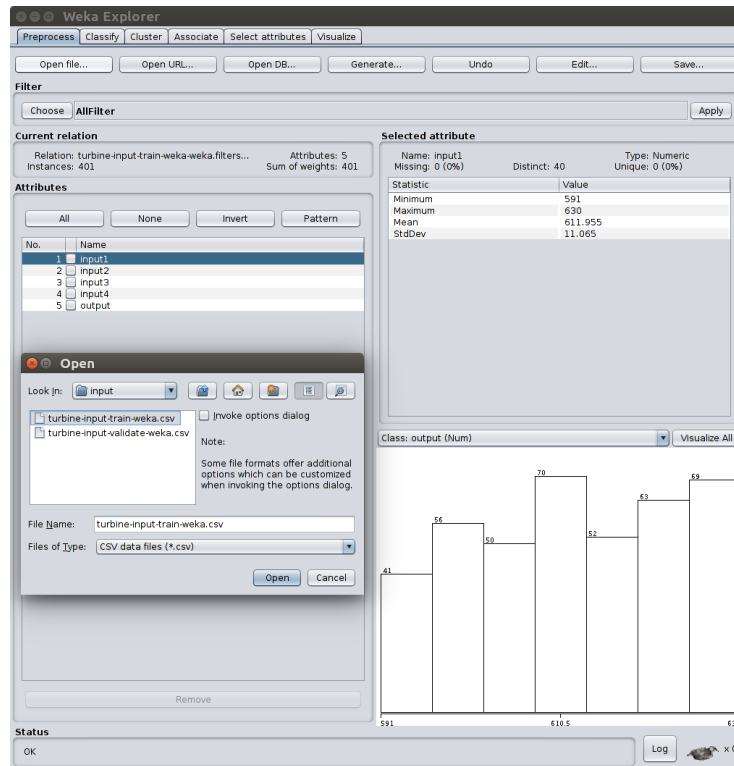


Figure 17: Upload training file to Weka.

It is important to note here that all attributes are being loaded correctly:

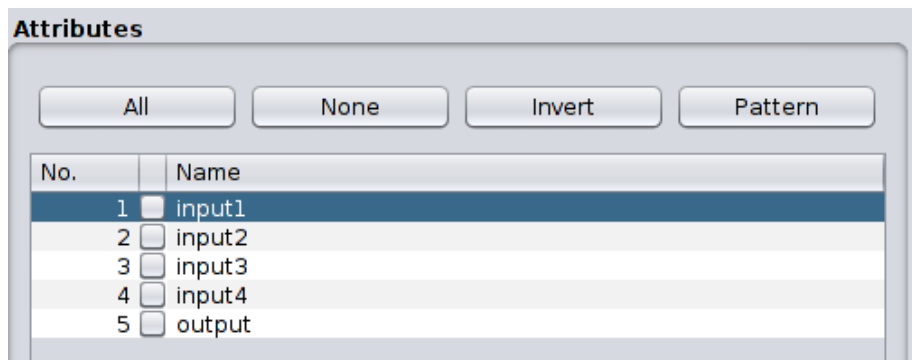


Figure 18: Weka attributes after training file load.

Later we go to "Classify" tab and there we press the button to select the type of classifier:

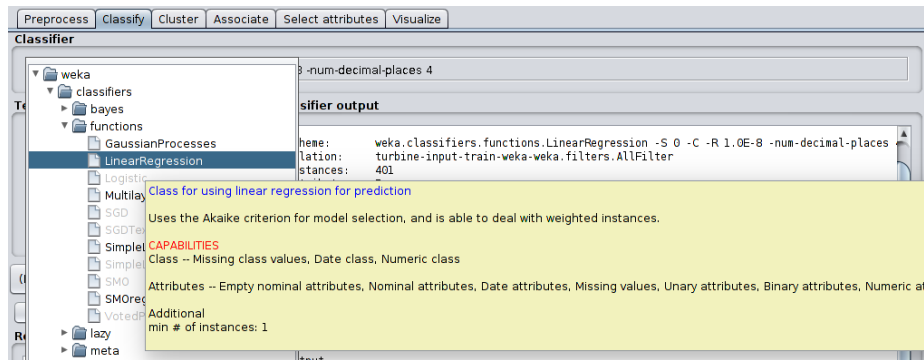


Figure 19: Selecting the right classifier on Weka.

As we want to include also the validating test, we must select "Supplied test set" into the section "Test options" and there include the corresponding file:

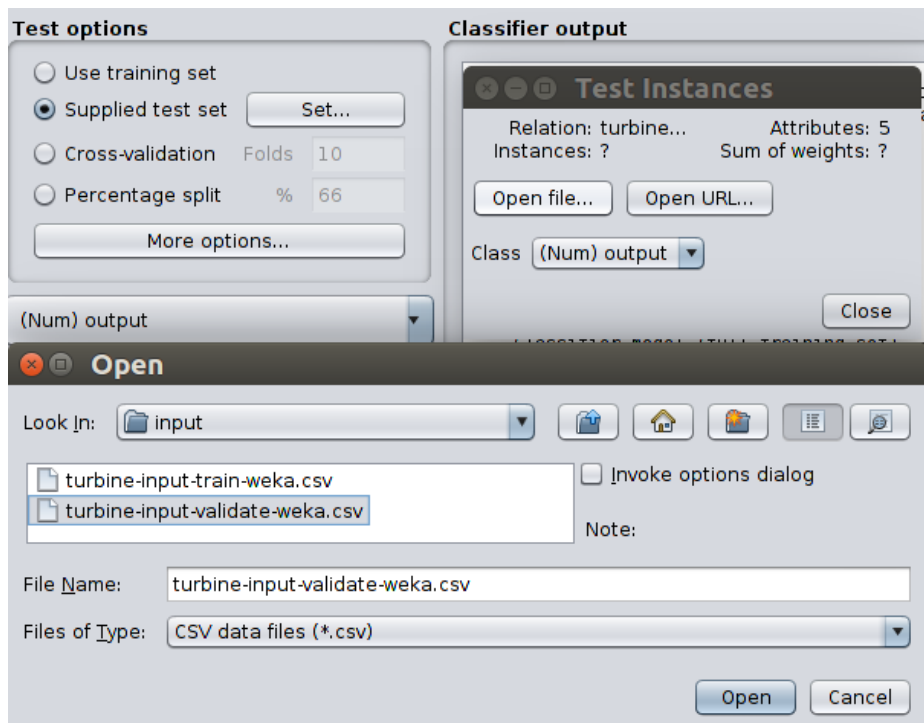


Figure 20: Upload validating file to Weka.

Finally we got the following output:

```

Classifier output

Scheme:      weka.classifiers.functions.LinearRegression -S 0 -C -R 1.0E-8 -num-decimal-places
Relation:    turbine-input-train-weka-weka.filters.AllFilter
Instances:   401
Attributes:  5
             input1
             input2
             input3
             input4
             output
Test mode:   evaluate on training data

=== Classifier model (full training set) ===

Linear Regression Model

output =
-796.4782 * input1 +
-95.1559 * input2 +
946.2515 * input3 +
1150.0931 * input4 +
419374.2937

Time taken to build model: 0 seconds

=== Evaluation on training set ===

Time taken to test model on training data: 0 seconds

=== Summary ===
Correlation coefficient      0.9867
Mean absolute error         174.4726
Root mean squared error     229.4631
Relative absolute error     14.6466 %
Root relative squared error 16.2252 %
Total Number of Instances   401

```

Figure 21: Weka output

We can use the output equation and LibreOffice-Calc software to compute all the predicted values:

H1														fx Σ =	
														=-(796.4782 * A1) - (95.1559 * B1) + (946.2515 * C1) + (1150.0931 * D1) + 419374.2937	
	A	B	C	D	E	F	H	I	J	K	L	M	N		
1	620	84.99	85.165	5	3698.29		3808.484259								
2	595	61.1	60.55	4	1935.13		1551.639935								
3	629	93.94	93.92	5.5	4509.67		4648.013584								
4	617	81.69	80.54	8	5708.99		5585.799439								
5	591	57.04	56.365	4.5	2024.05		1738.869712								
6	617	81.94	81.92	5.5	3981.74		3992.604784								
7	623	87.69	86.54	8	6171.22		5913.503839								
8	627	91.94	91.92	5.5	4427.36		4538.778784								
9	609	74.16	74.765	3.5	2117.9		2034.127603								
10	594	59.94	58.92	5.5	2692.87		2641.248684								
11	603	68.94	67.92	5.5	3237.11		3132.805284								
12	629	94.04	94.365	4.5	3558.33		3909.486812								
13	592	57.99	57.165	5	2327.54		2184.041156								
14	603	68.79	67.14	7	4063.35		4134.142149								
15	614	78.94	78.92	5.5	3838.09		3828.752584								
16	617	81.59	79.84	9	6098.9		6083.032079								
17	626	80.04	80.365	4.5	2422.40		2601.017911								

Figure 22: Computing predicted values.

Finally we can use the predicted values in the sample "turbine-results.xls" file to compute the percentage error and plot the result.

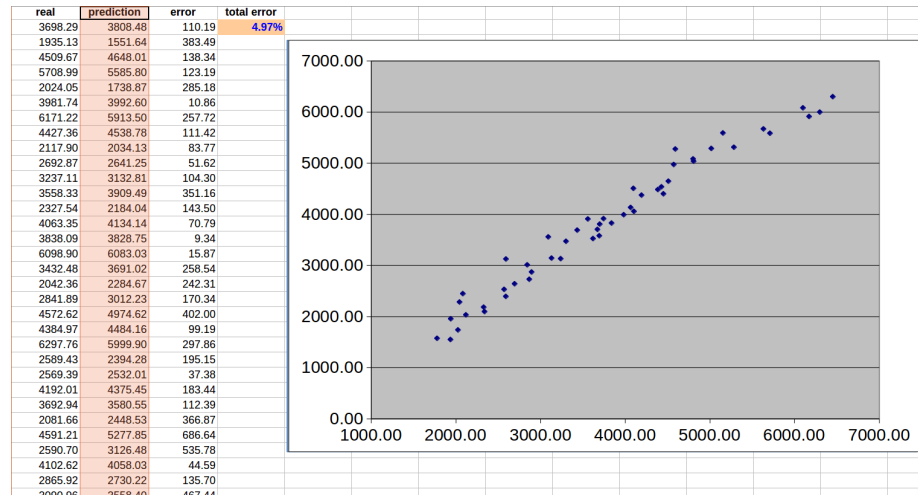


Figure 23: Percentage error from Weka output equation.

4 Comparison between results

After using this two machine learning methods, it is evident that Back-propagation gave us much more accurate predictions than those we got using Multiple Linear Regression (Figures 24 and 25), even though, we have to recognize that MLR execution is done much quicker than the Backprop.

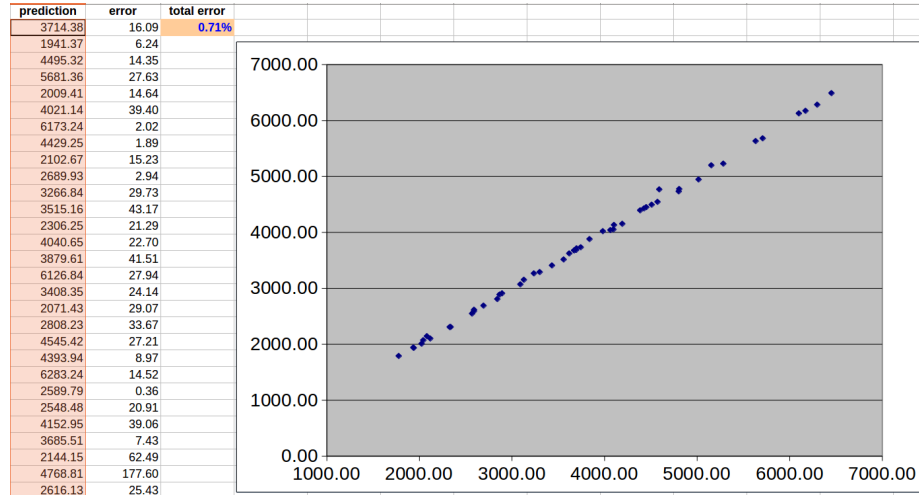


Figure 24: Backprop result using template xls file.

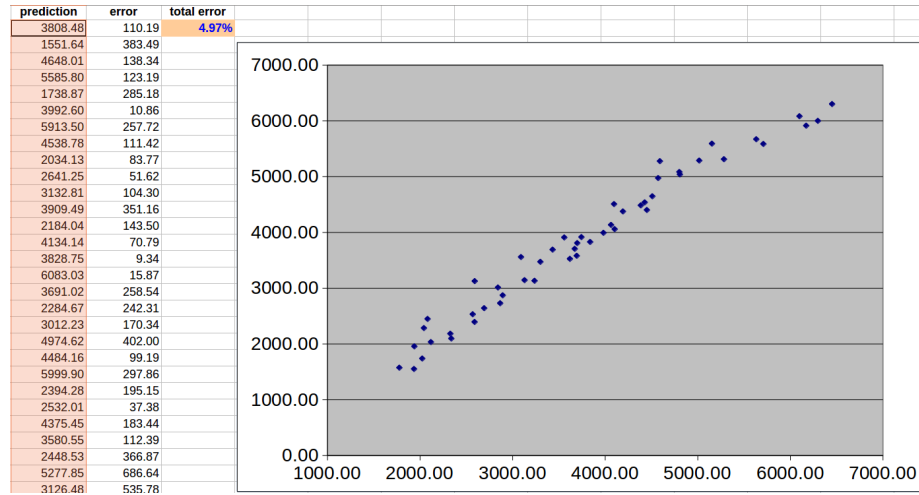


Figure 25: MLR result using template xls file.

References

- [1] Sergio Gómez Jiménez, "Multilayer neural networks: learning models and applications", Universitat de Barcelona, July 1994.
- [2] Rumelhart, David E.; Hinton, Geoffrey E.; Williams, Ronald J. (8 October 1986). "Learning representations by back-propagating errors". *Nature*. 323 (6088): 533–536.
- [3] Yann LeCun et al. "Efficient BackProp". Springer, 1998.
- [4] David A. Freedman. "Statistical Models: Theory and Practice". Cambridge University Press, 2009.
- [5] Wikipedia contributors. "Linear regression." Wikipedia, The Free Encyclopedia. Web. 28 Jan. 2017.
- [6] Wikipedia contributors. "Matrix multiplication." Wikipedia, The Free Encyclopedia. Web. 28 Jan. 2017.
- [7] Machine Learning Group at the University of Waikato, "Weka 3: Data Mining Software in Java",
- [8] Serdar Yegulalp, "11 open source tools for making the most of machine learning". InfoWorld. Dec 4, 2014
- [9] predictiveanalyticstoday contributors, "50 top free data mining software". 2014
- [10] Wikipedia contributors. "Weka (machine learning)." Wikipedia, The Free Encyclopedia. Web. 29 Jan. 2017.