CODING TRICKS

MACHINE LEARNING AND CODING TRICKS

Leave a Comment

Updated on October 15, 2016

Generalized linear regression with

Python and scikit-learn library

One of the most used tools in machine learning, statistics and applied mathematics in general is the regression tool. I say the regression, but there are lots of regression models and the one I will try to cover here is the well known generalized linear regression. The idea underneath this complex word is quite simple: the observation we try to fit is supposed to be a linear combination of input explanatory variables. Mathematically, if the observation vector is called $\widehat{\mathbf{y}}$ this can be formulated by

$$\hat{\mathbf{y}} = \sum_k w_k \mathbf{x_k}$$

where the $latex\mathbf{x_k}$ stands for the explanatory vector *k*, and the w_k are the weight of each explanatory variable. Furthermore, as we usually need for an intercept, by convention we set $x_0 = 1$ so that the weight w_0 represents the intercept.

Linear Regression

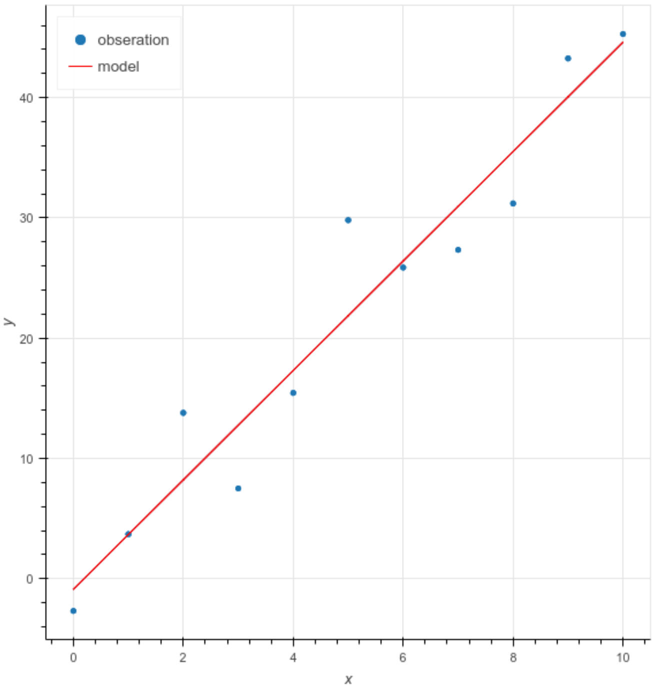
As a first example, let's begin by a simple linear regression which consists of minimizing the sum of error squares. The error is defined as the difference between the expected true value and the predicted value obtained by our model. Mathematically, it expresses

$$\underset{w}{min}||X\mathbf{w}-\mathbf{y}||_2^2$$

To illustrate this simple example, let's use the awesome library scikit-learn and especially the package `sklearn.linear_model`

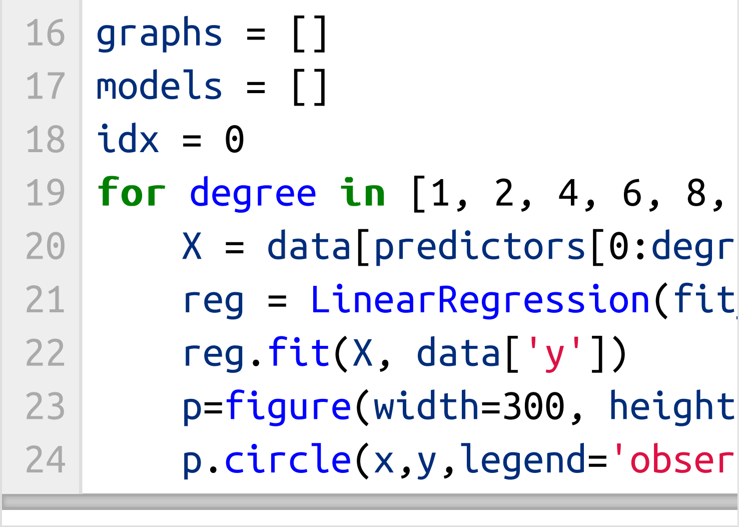
```
# General
import numpy as np
import pandas as pd
# Charting
from bokeh.plotting import fi
from bokeh.models import Colu
from bokeh.io import output_r
# Regression
from sklearn.linear_model impo


# For this example, I use outp
output_notebook()

```

```
# Define data
x_1 = np.linspace(0,10,11)
x_0 = np.ones(len(x_1))
y = 4*x+1+np.random.normal(0,
data = pd.DataFrame(np.column_

# Define the model by explicit
reg = LinearRegression(fit_int

# Fit the data
```

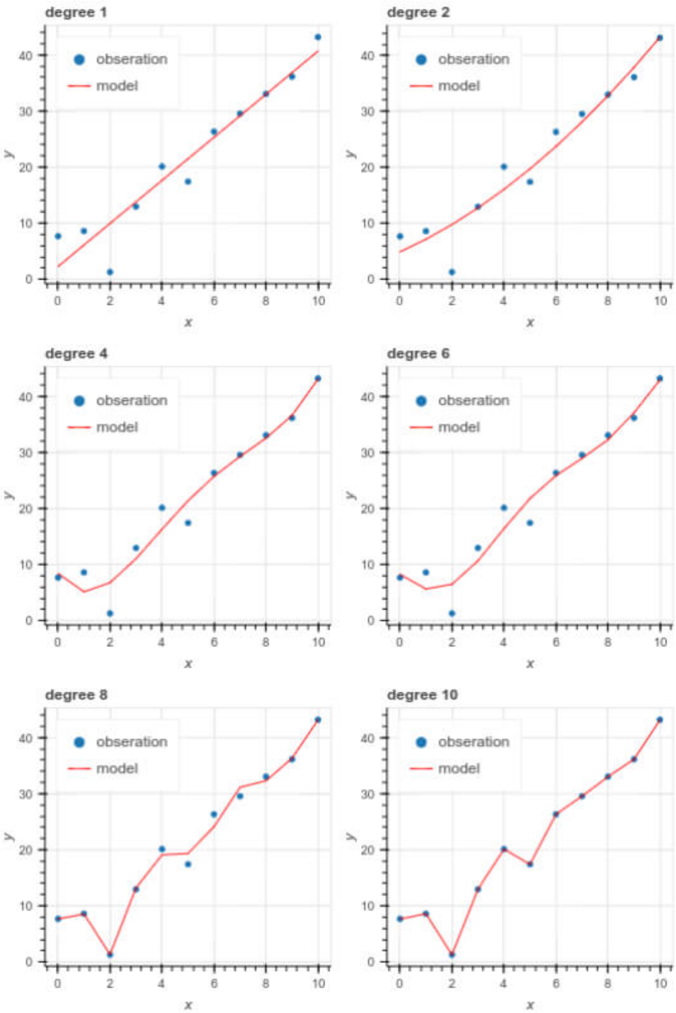


The model we use here is quite simple, it is just a line. The model seems quite good with fitted coefficients of $\latex w_0=-0.87998$ and $\latex w_1=4.54914$, but the error is not null (mean squared error = 15.57 in my example). In general in machine learning, a way to reduce the residual error is to change the model by a slightly more complex one. In our case, we simply fitted a polynom of degree 1. What if we increase the polynom degree ? For example, let's say we increase the degree up to 12:

```
np.random.seed(10)
x_1=np.linspace(0,10,11)
x_0=np.ones(len(x_1))
y=4*x+1+np.random.normal(0, 5
data = pd.DataFrame(np.column_

predictors=['x_0','x_1']
predictors.extend(['x_{}'.form
for i in range(2,13):
    data['x_{}'.format(i)] = 

col = ['rss','intercept'] + ['
ind = ['alpha_%.2g'%alpha for 
coef_matrix_lasso = pd.DataFra

```



```
graphs = []
models = []
idx = 0
for degree in [1, 2, 4, 6, 8,
    X = data[predictors[0:degr
    reg = LinearRegression(fit
    reg.fit(X, data['y'])
    p=figure(width=300, height
    p.circle(x,y,legend='obser
```



As we can see, the more we increase the polynomial degree of the model, the more we reduce the residual error. However, and it is particularly evident in this example, the reductions of the error is not necessarily a sign of a better model. Indeed, imagine we use a high degree polynom as our model, the error tends to be null (and it is actually here as we have a polynom degree equal to the number of observations). But if we add an extra observation, our model surely experiences a high residual error.

```
new_x = 15
data_x = np.array([new_x**i f
new_y = 4*new_x+1+np.random.n
idx = 0
for degree in [1, 2, 4, 6, 8,
    y_pred = models[idx].predi
    rss = (y_pred-new_y)**2
    idx += 1
    print "The residual error
```

and the result is

```
The residual error for X=15 is
The residual error for X=15 is
The residual error for X=15 is
The residual error for X=15 is
The residual error for X=15 is
The residual error for X=15 is
```

As we can see in this example, adding an observation at x=15 leads to increasing error with the polynom degree. This behavior is known as overfitting, i.e. the model fit very well the data but tends to poorly perform on new data. We say that is suffers of great variance.

To overcome this problem, we need to choose the right polynom degree. We could for example split our dataset into two parts, a train set and a test set. Then, the best model would be the one with the least residual error on the test set.

However, there a clever method to limit the overfitting phenomenon: regularization.

Regularization

Regularization consists of adding an penalty to a model, with the goal of preventing overfitting. It comes from the constatation that when the degree of the polynom increases (to take our first example), the weights of each monom also increases. Therefore, to overcome overfitting, we penalize monoms with high weight. The minimization function now becomes

$$\underset{w}{min} ||X\mathbf{w} - \mathbf{y}||_2^2 + \alpha||w||^2$$

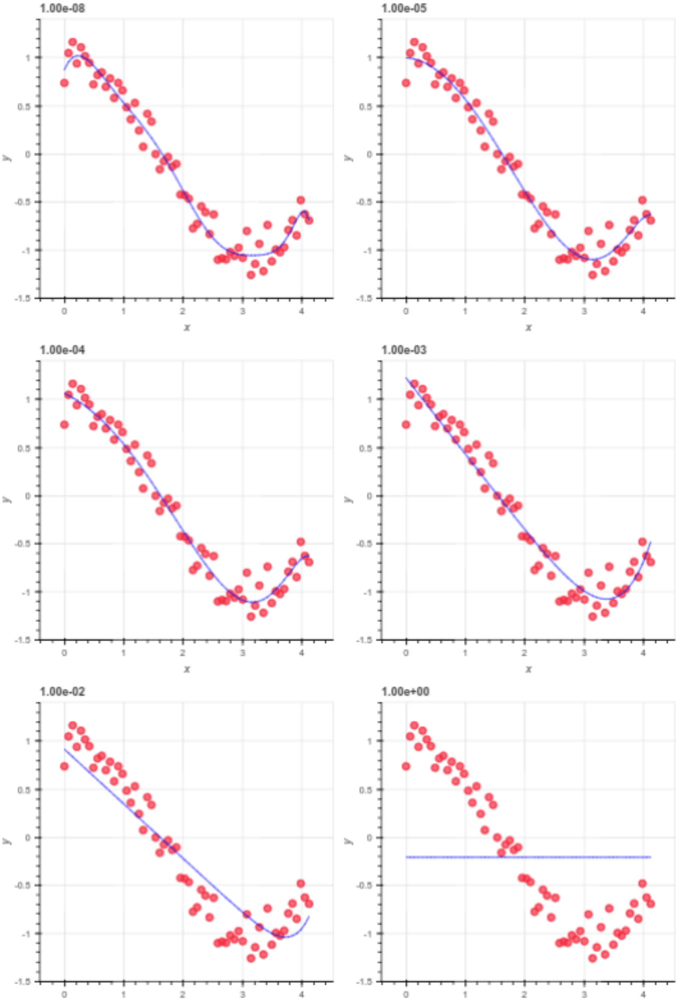
where $||\cdot||$ is typically L1 or L2 norm, and α is an hyper-parameter that can be tunable to adjust the penality sensibility (0 means no penalty, i.e. unregularized model). The two widely used regularization methods are L1 and L2 regularization, also called lasso and ridge regression.

Lasso

To cite content of the scikit-learn library

Lasso is useful in some contexts due to its tendency to prefer solutions with fewer parameter values, effectively reducing the number of variables upon which the given solution is dependent. For this reason, the Lasso and its variants are fundamental to the field of compressed sensing. Under certain conditions, it can recover the exact set of non-zero weights.

```python
# General
import numpy as np
import pandas as pd
# Charting
from bokeh.charts import show
from bokeh.models.glyphs impo
from bokeh.plotting import fi
from bokeh.models import Colu
from bokeh.layouts import gri
from bokeh.io import output_n
# Regression
from sklearn.linear_model imp
output_notebook()


def generate_plot(data, title=
    source = ColumnDataSource(
    p = Scatter(data, x='x', y
    line = Line_glyph(x='x', y
    p.add_glyph(source, line)
    return p


def lasso_regression(data, pre
```


1.00e-08

1.00e-05

1.00e-04

1.00e-03

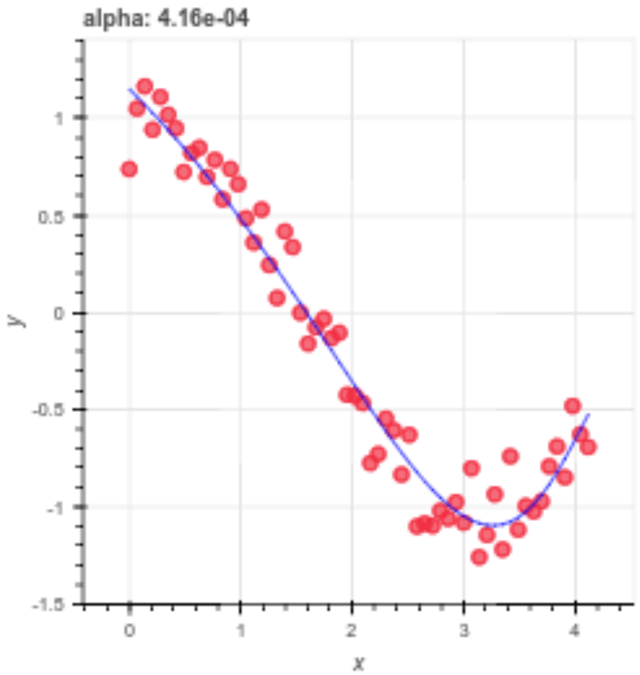
1.00e-02

1.00e+00

We clearly see the effect of regularization. When we increase the penalty, we strongly limit the weights of each coefficient, up to only keep the intercept. But how to choose the right parameter ? Well, here again, we need to look at the residual error computed on a set which is not the training set. Such a method is known as validation. The principle is to split the data set into three parts, say 80% for the training set, 10% for validation and 10% for test. The model is trained on the training set, then the validation set is used to choose the hyper-parameter. Finally, the test set is used to estimate the true error of the model. However, on small data sets, this approach is not efficient enough as it limits the amount of data available for training. For such small data sets, we can apply the method of cross-validation. For that, we split the data set into two parts, one for the training and the other for the test. The training is then performed on all the training set but some k samples. So let's imagine that the training set is composed of N samples. We perform N/k regressions on the N-k samples and we compute the validation error on the k remaining samples. After all those regressions, the validation error is the mean error of all the validation errors.

In the scikit-learn library, there is a class that implement this approach and finds the optimal hyper-parameter: LassoCV. We then re-use the preceeding code sample omitting the `alpha` parameter to force the use of the `LassoCV` model :

```
graph, model = lasso_regressio
graph.title.text = 'alpha: {:.
show(graph);
```



According to the LassoCV model, the optimal hyper-parameter $\alpha = 4.16e^{-4}$.

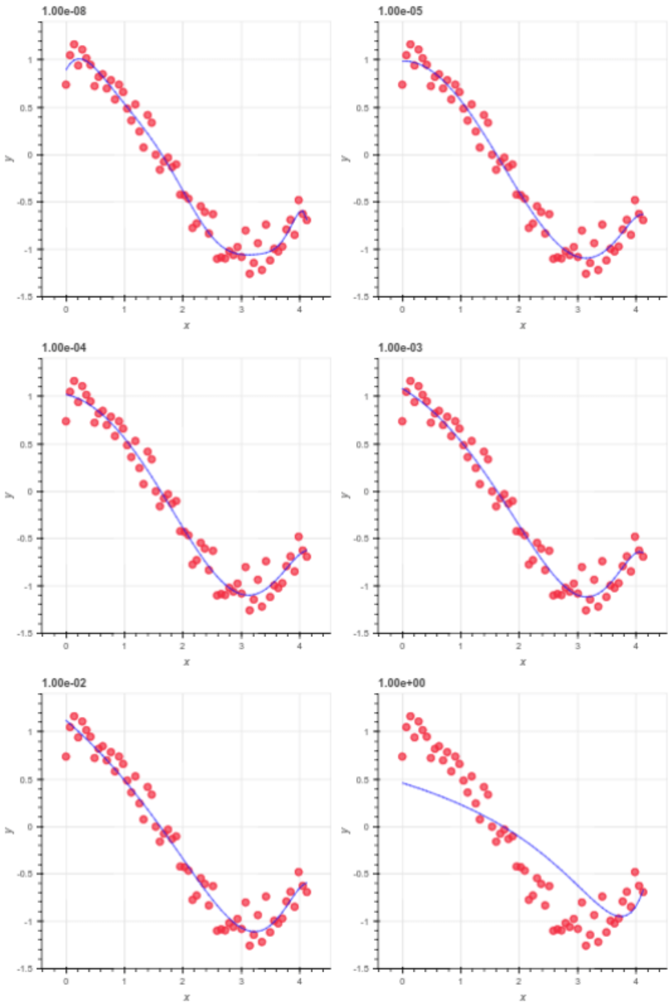
Ridge Regression

The ridge regression is quite similar to lasso, and differs only by the order of the norm used in the regularization term. In lasso, we used a norm of order 1 (L1) and in the ridge regression we use a norm of order 2 (L2). The behavior of this regularization technique is that all resulting weights of the models are still non null, but eventually with very small value so that their influence on the predicted value is quite low. On the opposite, lasso imposes sparsity of the model by eventually setting weights to null which make the model interpretation easier. The main advantage of the ridge regression is that it is indifferent to multiplicative factor, and tends to equals weights of highly-correlated variables whereas lasso will choose or the other.

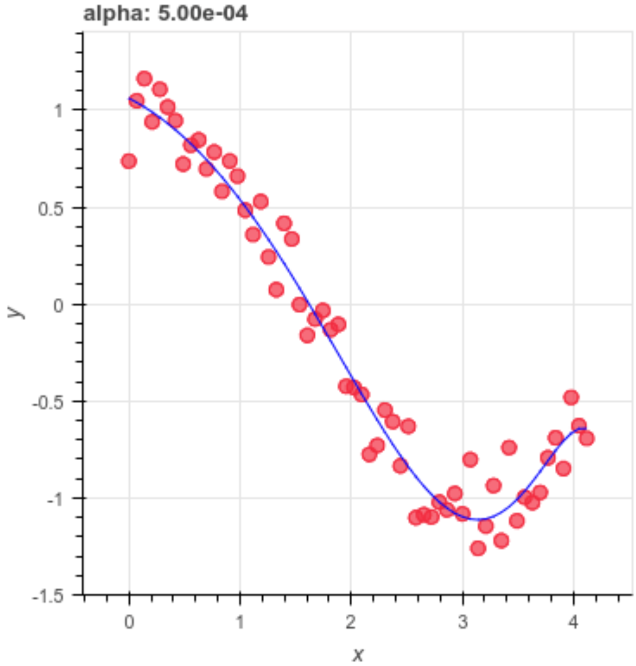
The implementation of our example is really similar to the preceeding with lasso. All we have to do is replacing `Lasso` and `LassoCV` by `Ridge` and `RidgeCV` !

```
# General
import numpy as np
import pandas as pd
# Charting
from bokeh.charts import show
from bokeh.models.glyphs impo
from bokeh.plotting import fi
from bokeh.models import Colu
from bokeh.layouts import gri
from bokeh.io import output_n
# Regression
from sklearn.linear_model imp
output_notebook()


def generate_plot(data, title=
    source = ColumnDataSource(
    p = Scatter(data, x='x', y
    line = Line_glyph(x='x', y
    p.add_glyph(source, line)
    return p

```

```

def ridge_regression(data, pre
```



And with the use of cross-validation:



As we can see, the resulting model is a little different as with the lasso regularization. And if we now look at the weight coefficients:

Degree	1	2	3
L1	-0.569	-0.101	0
L2	-0.292e-01	-0.225e-01	-0 02

6	7	8	9	10	11	12	13
5.813e-04	0	0	0	0	0	0	0
3.389e-04	5.571e-05	6.083e-06	-6.79 08	-2.72e-07	-9.600e-08	-2.353e-08	-4.4 09

Conclusion

In this article, we saw two regularization techniques and the importance to ALWAYS use regularization when we try to fit a model. We also saw that noth techniques, although quite similar, give very different results. Indeed, whereas the ridge technique includes all the explanatory variables, lasso results in a sparse model which is often easier to understand. However, lasso performs less well in the case of highly correlated variables as it tends to produce high sparsity in the result. And that's exactly what we saw in our example when we tried to fit a cosinus with a polynom. Each variable are highly correlated so that the resulting model has a lot of zero values as weight coefficients. Along with Ridge and Lasso, Elastic Net is another useful techniques which combines both L1 and L2 regularization. It allows for learning a sparse model while it also keep ridge properties.

End note

We briefly evocate the subject, but as always in machine learning, we need several datasets to train the model, validate it and test it. We discussed that a little in this post and we quickly saw how to deal with small datasets with cross-validation. This could be the subject of a next post.

Like this:

Loading...

Category: Algorithm, Machine Learning, Python   Tags: Lasso, Machine Learning, Python, Regression, Regularization, Ridge

← Theano demo with Docker help

Your first CNN made easy with Docker and Tensorflow →

Leave a Reply

SEARCH

Search …

GO

RECENT POSTS

- Your first CNN made easy with Docker and Tensorflow
- Generalized linear regression with Python and scikit-learn library
- Theano demo with Docker help
- Slack Python client to send messages
- Scrapping bank website to access account operations

RECENT COMMENTS

ARCHIVES

November 2017

October 2016

September 2016

March 2016

October 2015

May 2015

April 2015

December 2014

November 2014

CATEGORIES

.NET

Algorithm

C#

Database

Deep learning

- Docker
- Google Service
- JSON
- Machine Learning
- Matlab
- MongoDB
- Nodejs
- Python
- remote desktop
- SCrapping
- SQLite
- Tips
- WPF

META

- Log in

Entries RSS

Comments RSS

WordPress.org

SHARE

SEARCH

Search …

GO