Machine Learning and Data Science: Logistic and Linear Regression Regularization

Written on July 31, 2017 by Dr. Donald Kinghorn

Share

Logistic (and Linear) Regression Regularization

In the last post I went through some of the reasoning behind Logistic Regression and derived most of the equations needed for implementation (and I will look at "Regularization" in order to address an important problem that is common with implementations, namely over-fitting. Well look at this for logistic regression and linear regression. After getting the equations for regularization worked out well look at an example in Python showing how this can be used for a badly over-fitted linear regression model.

The post that laid the foundation for this post was "Logistic Regression the Theory".

Avoiding Over-Fitting

In part 6 of the posts on Linear Regression I talked about over-fitting in that context. The same problems happen in Logistic Regression and can be quite bad in cases with small datasets and many features.

> I think of methods like Regularization as being in the class of "Keep my optimization from blowing up".

The more complicated your model, the more likely you are to have numerical instabilities develop during optimization. The reason is, that it's often the case that making things "blow up" looks like the best path to take for an optimization program. It can find a path that causes the cost to "fall of a cliff" then it may just do that because it "reckons" the cost function even if the result is garbage for your model.

Some things you can do to help with the over-fitting problem are:

- **Model Interpretation and Feature Elimination** Reducing the number of feature variables and help with over-fitting and model interpretation. If you have only a few "good" features that might be better to use just the features that provide the most information. Interpretation is a big part of Data Science. Looking at the "importance" of features and performing some ad-hoc analysis for feature elimination. It's usually done by analyzing a model with and without a given feature to see if the feature makes a statistically significant contribution to the accuracy of the model. This can be very useful for deciding if a "question" is relevant to the study you are looking at. For example, a study that is looking at lifestyle "indicators" on the likelihood of developing cancer may find that smoking is a significant "feature" but watching TV for 4 hours a day is not significant. That is a case where you have clear reasoning support to your features and this is valuable for interpreting the models "meaning". However, if the features are just random pieces in an image then it might be obvious here to interpret any individual pixel. It is also, common to have many features that are statistically poor predictors but taken collectively give a very good model with high predictive value.
- **Constrained Optimization** If there is a physical interpretation of what the parameters in a mean then there may be a logical mathematical constraint that could be applied. That is the realm of algorithms referred to as "Constrained Optimization". That means to add extra terms to the Model and/or Cost Function to constrain the parameters you are optimizing within some bounds. That is a common practice and is often referred in the nature of the problem you are working on.
- **Regularization - Optimization Improvement with "model type parameters"** It's really most common in machine learning to add a term to the cost function that is a function of the model parameters. This is done so that the parameters may be extremely constrained or impossible. Your main concern is to stabilize the optimization so you get a model that gives a good fit to the training data, generalizes well on tested data and has good predictive value. This is where Regularization comes in. With regularization an adjustable optimization parameter constant in the form of an additional cost function term can be used to algorithmically control the stability of the optimization.

Regularized Cost Function

Regularization acts by penalizing large values of the optimization parameters. It gives a more balanced result that is usually much better behaved, provides a good fit to the dataset and provides good predictive value.

The most common way to "regularize" the cost function is to add a "scaled square of the 2-norm". There are other norms that can used for this and there is some theoretical justification for all of the choices. However, in my opinion it's really about doing "anything" to stabilize the optimization! The squared 2-norm is easy to work with and has the advantage that it is a "smooth" function so the gradient will be well behaved.

$$R(\mathbf{x}) = \|\mathbf{x}\|_2^2 = \|\mathbf{x}\|_2^2$$

It is most common to exclude the first element of $\mathbf{x}$, the 0 order term, from the parameter vector $\mathbf{x}$. The first element of $\mathbf{x}$ can be thought of as multiplying a feature variable of the form $x^0 = 1$. That term is the "intercept" or what is often referred to as the "bias term". That element is excluded from the regularization term R.

In more detail R looks like:

$$\begin{aligned} R(\mathbf{x}[1:n]) &= \lambda(\mathbf{x}_{[n]}\|_2^2) \\ &= \lambda\mathbf{x}[1:n]^T\mathbf{x}[1:n] \\ &= \lambda\mathbf{x}^T\mathbf{I}\mathbf{x} \end{aligned}$$

$$\text{(where } \mathbf{x}[1:n] = \mathbf{x} \text{ with the 0th* element excluded)}$$

$$\text{(where } \mathbf{I} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & \ddots & \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix} \text{, identity matrix with the [1,1]* element} = 0)$$

*Note: referring to the first term of a vector as the 0th element is annoying to the personally but many C based languages like Python use this convention. In the first post I did on Linear Regression I warned that it was important to be flexible with notation and to understand what things really mean! You could say the 1.1 element of $\mathbf{I}$ is the 0.0 element if that's what you like.

Equations for Logistic Regression with Regularization in Matrix Form

In my last post on Logistic Regression I derived the equations for logistic regression in matrix form including an alternative derivation for J and the the gradient of J dotted by the regularization term, R. Is a simple modification. With the form of R we are using the modification to the gradient is easy to derive using the differential of R.

$$\begin{aligned} dR(\mathbf{x}) &= d\left[\frac{\lambda}{2m}\mathbf{x}^T\mathbf{I}\mathbf{x}\right] = \frac{\lambda}{m}\mathbf{x}^T d\mathbf{x} \\ \nabla R &= \frac{\lambda}{m}\mathbf{I}\mathbf{x} \end{aligned}$$

$\mathbf{I}\mathbf{x}$ is formal matrix notation, it is actually trivial. It is just the vector $\mathbf{x}$ with the first element set to 0.

Equations for logistic regression with regularization.

$$\begin{aligned} m &= \text{the number of elements in the training set} \\ n &= \text{the number of elements in the parameter vector } \mathbf{x} \\ \lambda &= \text{the adjustable regularization weight} \\ g(z) &= \frac{1}{1+e^{-z}} \\ h_{\mathbf{x}}(X) &= g(X\mathbf{x}) \\ J(\mathbf{x}) &= -\frac{1}{m}\left[y^T\log(h_{\mathbf{x}}(X)) + (1-y)^T\log(1-h_{\mathbf{x}}(X))\right] + \frac{\lambda}{2m}\mathbf{x}^T\mathbf{I}\mathbf{x} \\ \nabla J(\mathbf{x}) &= \frac{1}{m}X^T(h-y) + \frac{\lambda}{m}\mathbf{I}\mathbf{x} \\ &= \frac{1}{m}X^T(g(X\mathbf{x}) - y) + \frac{\lambda}{m}\mathbf{I}\mathbf{x} \end{aligned}$$

In my next post on logistic regression I'll do an implementation of these equations and some examples. To finish this post I'd like to show how regularization can be used in Linear Regression.

Regularization for Linear Regression

In part 6 of the posts on Linear Regression I derived the matrix "solution" for multivariate Linear Regression. In part 6 of the Linear Regression posts I took an example of over-fitting. In this section we'll add regularization to those equations and use that over-fit example to see how regularization works.

Equations for linear regression with regularization.

$$\begin{aligned} h_{\mathbf{x}}(X) &= X\mathbf{x} \\ J(\mathbf{x}) &= \frac{1}{2m}(X\mathbf{x} - y)^T(X\mathbf{x} - y) + \frac{\lambda}{2m}\mathbf{x}^T\mathbf{I}\mathbf{x} \\ \nabla_{\mathbf{x}}J(\mathbf{x}) &= \frac{1}{m}(X^TX\mathbf{x} - X^Ty) + \frac{\lambda}{m}\mathbf{I}\mathbf{x} \\ \mathbf{x} &= (X^TX + \lambda\mathbf{I}_1)^{-1}X^Ty \end{aligned}$$

The linear regression expressions are similar to logistic regression but in this case the solution for $\mathbf{x}$ can easily be found explicitly. To solve for $\mathbf{x}$ set the gradient, $\nabla_{\mathbf{x}}J(\mathbf{x})$ to 0.

$$\begin{aligned} \nabla_{\mathbf{x}}J(\mathbf{x}) &= 0 \text{ (gradient)} \\ X^TX\mathbf{x} + \lambda\mathbf{I}\mathbf{x} &= X^Ty \\ (X^TX + \lambda\mathbf{I}\mathbf{x} &= X^Ty \\ \mathbf{x} &= (X^TX + \lambda\mathbf{I}_1)^{-1}X^Ty \end{aligned}$$

Example of regularization and over-fitting

This example is from part of the Linear Regression posts. It is a linear regression fit to 8 points of training-set data selected from a known function, $x\sin(x)$ [2]. There is 8 points of data for a test-set that is taken from other places along the curve. The training and test sets also include a small amount of random "noise". The cost functions, J and RF values are calculated to evaluate the goodness of fit which is then checked against the training-set and the test-set.

The model is over-fitting because it includes as many "features" as there are data points. We'll apply regularization to see if this overfit model can be smoothed out.

Note: When using the linear regression normal equation solution to determine a regularization cost only be used if the number of parameters in $\mathbf{x}$ is less than or equal to the number of data points in the training set. This is because in this case the size of $\mathbf{I}_1$ can not be larger than X^TX. If gradient optimization routine is used along with the ∇J then this restriction could be avoided.

First we need to load some Python modules, define a few functions and setup the data.

In [2]:
```
%matplotlib inline
import numpy as np, pandas as pd, matplotlib as mpl
import matplotlib.pyplot as plt, time
import seaborn as sns, # higher level plotting tools
```

In [3]:
```
def f(x,w):
    # w = np.array([w0,w1,w2,...])
    return w[0]

def f_opt(x,y):
    # x = np.array([x0,x1,x2,...])
    # y = np.array([y0,y1,y2,...])
    return np.dot( np.linalg.pinv(X), y)

def J(x,w,y):
    # w = cost function
    m = len(x)
    s = 1/(2*m) * np.dot( [ f(x,w)[i] - y[i], f(x,w)[i] - y[i] ]
    return s

def RS(x,w,lr):
    # R squared
    y_mean = y.mean()
    s1 = 1 - ( np.dot( [ f(x,w)[i] - y[i], [ f(x,w)[i] - y[i] ] / np.dot( [ y[i] - y_mean ],[ y[i] - y_mean
    s = s1
    return s1
```

In [4]:
```
def f(x):
    # the data
    y = x*np.sin(x/2)
    return y

np.random.seed(0)
noise = np.random.rand(8)/2  # some random "normal" noise

x = np.linspace(0,10,8)  # for training-set
y = f(x) + noise[:8]     # for training-set

xtest = np.linspace(1.5,9.5)    # for test set
ytest = f(xtest) + noise[0:16:2]  # for test set
```

Here's the model a solution of the regression and a plot of the test fit.

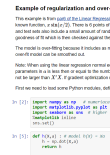
$$h(\mathbf{x}) = w_0 + w_1x + w_2x^2 + w_3x^3 + w_4x^4$$

In [96]:
```
X = np.column_stack( [np.ones(len(x,x,x)), x, x**2, x**3, x**4, x**5 ]) # construct the augmente
# matrix X
xtest = np.column_stack( [np.ones(len(xtest,xtest,xtest)), xtest, xtest**2, xtest**3, xtest**4, xtest**5 ])
# model for error analysis
w = X_opt(X,y)  # solve the regression

fig,ax = plt.subplots()
ax.plot(x,y, 'o', color='g', label='training data')  # plot the test data
ax.plot(xtest,ytest, 'o', color='r', label='testset data')

xt = np.linspace(1.5,10,100)
ax.plot(xt, f(xt), '-', color='r', label='source function')
ax.plot(xt, f(xt,w), '-', color='b', label='fit to training-set data: '+str(w[0]*x**0 + w[1]*x**1+
        label='fit: '+ 0.2f + 0.2f*x + 0.2f*x**2 + 0.2f*x**3 + 0.2f*x**4 + w[0],w[1],w[2],w[3],w[4],
        ax.legend()
```

OUT[96]:



In [96]:
```
print('a =', w)  # print the values of w
print('training-set: ',J(x,w,y),'   R2 =',RS(x,w,y))

print('test-set:     ',J(xtest,w,ytest),'   R2 =',RS(xtest,w,ytest))
```

```
a = [ -0.20429180-02  -1.09653584e-00  0.81128534e+00  -1.08923995e+00
  1.08485329e-01  -4.03724164e-02]
training-set:  J = 3.8543313180E-12   R2 = 0.7354731183572
test-set:      J = 15.3963808526      R2 = -6.7354731183572
```

The model is a perfect fit to the training data but evaluating it against the test-set shows how bad it really is. RF for the model is 1 because of the perfect fit to the training data but for the test data RF is very far from +1.

Now lets solve this same problem and include a regularization term

In [99]:
```
# x_opt_reg(x,y,l_reg) # a linear regression with regularization w = (X'X + l_1)^-1 X'y
    l_1 = np.eye(X.shape[1])
    l_1[0,0] = 0.0
    XX = np.dot(X.T,X) + l_reg * l_1
    x_opt_reg = np.dot( np.linalg.pinv(XX),X.T), y)
    return x_opt_reg

# J_reg(x,y,w,l_reg) # regularized cost function
    m = len(x)
    l_reg = 1/(2*m) * np.dot( [ f(x,w)[i] - y[i], [ f(x,w)[i] - y[i] ] ] * np.dot( w[1:len(l_1)],l_1)
    return l_reg
```

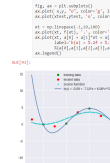
In [100]:
```
X = np.column_stack( [np.ones(len(x,x,x)), x, x**2, x**3, x**4, x**5 ]) # construct the augmente
# matrix X
xtest = np.column_stack( [np.ones(len(xtest,xtest,xtest)), xtest, xtest**2, xtest**3, xtest**4, xtest**5 ])
# model for error analysis

l_reg = .2
w = X_opt_reg(X,y,l_reg)  # solve the regression

fig,ax = plt.subplots()
ax.plot(x,y, 'o', color='g', label='training data')  # plot the test data
ax.plot(xtest,ytest, 'o', color='r', label='testset data')

xt = np.linspace(1.5,10,100)
ax.plot(xt, f(xt), '-', color='r', label='source function')
ax.plot(xt, f(xt,w), '-', color='b', label='source function')
        label='fit: '+ 0.2f + 0.2f*x + 0.2f*x**2 + 0.2f*x**3 + 0.2f*x**4 + w[0],w[1],w[2],w[3],w[4],
        Xtest[:,1],f(xtest,w),Xtest[:,1],f(xtest,w)[:] # plot the model function
ax.legend()
```

OUT[100]:



In [96]:
```
print('a =', w)  # print the values of w
print('training-set: ',J(x,w,y,l_reg),'   R2 =',RS(x,w,y))

print('test-set:     ',J(xtest,w,ytest,l_reg),'   R2 =',RS(xtest,w,ytest))
```

```
a = [ 0.00292107  0.17253276  0.12518276  0.12518276  0.00491186  0.00292107]
training-set:  J = 0.2328809216879   R2 = 0.982276891
test-set:      J = 0.9150653570751   R2 = 0.9150653570
```

Much better! The regularization has smoothed the model and it now gives a good fit and generalizes well to the test-set.

In the next post I'll get back to Logistic Regression with an Implementation, examples and a discussion of evaluating goodness of fit.

Happy computing! --dbk

Tags: Machine Learning, Data Science, Python, Jupyter notebook, Programming

0 Comments Puget Systems Disqus' Privacy Policy Login

Recommend Tweet Share Sort by Oldest

Start the discussion...

LOG IN WITH OR SIGN UP WITH DISQUS

Name

Be the first to comment.

Subscribe Add Disqus to your site Do Not Sell My Data

DISQUS

Related HPC Posts

- Beginning with Machine Learning and AI
- Machine Learning and Data Science: Multivariate (Multivariate) Logistic Regression
- Machine Learning and Data Science: Logistic Regression Examples 1
- Machine Learning and Data Science: Logistic Regression Implementation
- Machine Learning and Data Science: Logistic Regression Theory
- Machine Learning and Data Science: Linear Regression Part 6