Week 2 – Learning Parameters of Logistic Regression

Goal: Learning parameters of logistic regression

Intuition behind maximum likelihood estimation

The quality metric is called the Maximum Likelihood estimation. The Likelihood function is the probability that the predicted class of the training data is correct. Larger likelihood values indicate a better set of coefficients. We are looking to maximize the likelihood estimate.

We want to find a set of coefficients, w, that make P(y = +1|x) go to 1.0 for training data with positive sentiment and make P(y = +1|x) go to 0 for training data with negative sentiment. Of course, we won't get perfect 1's or 0's for each row in the data.

Data Likelihood

Given this data;

Data Point	x[1]	x[2]	y
x1,y1	2	1	+1
x2,y2	0	2	-1
x3,y3	3	3	-1
x4,y4	4	1	+1

We want to find w that will cause us to match the output values y by maximizing the associated probability.

Data Point	x[1]	x[2]	y	Maximize
x1,y1	2	1	+1	P(y=+1|x1,w)
x2,y2	0	2	-1	P(y=-1|x2,w)
x3,y3	3	3	-1	P(y=-1|x3,w)
x4,y4	4	1	+1	P(y=+1|x4,w)

The final set of coefficients will based on combination of each row's probabilities. Probabilities are combined as a product (since they are independent events).

Explicitly for the data set in the table above;

$$l(\boldsymbol{w}) = P(y = +1|x[1] = 2, x[2] = 1, \boldsymbol{w}) \times P(y = +1|x[1] = 0, x[2] = 2, \boldsymbol{w}) \times P(y = +1|x[1] = 3, x[2] = 3, \boldsymbol{w}) \times P(y = +1|x[1] = 4, x[2] = 1, \boldsymbol{w})$$

Or more succinctly

$$l(\boldsymbol{w}) = P(y = +1|x_1, \boldsymbol{w}) \times P(y = +1|x_2, \boldsymbol{w}) \times P(y = +1|x_3, \boldsymbol{w}) \times P(y = +1|x_4, \boldsymbol{w})$$

Or more succinctly

$$l(\boldsymbol{w}) = P(y_1|x_1, \boldsymbol{w}) \times P(y_2|x_2, \boldsymbol{w}) \times P(y_3|x_3, \boldsymbol{w}) \times P(y_4|x_4, \boldsymbol{w})$$

Or most succinctly

$$l(\boldsymbol{w}) = \prod_{i=1}^{N} P(y_i|x_i, \boldsymbol{w})$$

Finding best linear classifier with gradient ascent
The goal is to pick w that maximizes $l(\boldsymbol{w})$ over all possible w.

$$\max_{(w_0, w_1, w_2)} \prod_{i=1}^{N} P(y_i|x_i, \boldsymbol{w})$$

Review of gradient ascent
Basically, we want to iterate on the gradient and move in the direction of the gradient until the gradient goes to within some tolerance of zero. This works because our goal is a concave curve, so the slope is zero at the maximum of the curve.

While not converged

$$\widehat{w}^{(t+1)} \leftarrow \widehat{w}^{(t)} + \eta \frac{dl}{dw}\Big|\widehat{w}^{(t)}$$

$\widehat{w}^{(t)}$ estimated coefficient in step t
$\widehat{w}^{(t+1)}$ estimated coefficient in step t+1
$\Big|\widehat{w}^{(t)}$ means "computed at $\widehat{w}^{(t)}$", so specifying that the derivative of the likelihood function with respect to w is calculated with $w^{(t)}$.
η is the step size, which scales the derivative before it is added to the coefficient to get the estimated coefficient for the next step.

The maximum is when $\frac{dl}{dw} = 0$, but in practice we stop when we get within some tolerance of zero;

$$\text{converged} = \frac{dl}{dw} < tolerance$$

In higher dimensional spaces we use the gradient, which is a vector of partial derivatives, one for each coefficient;

$$\nabla l(w) = \begin{bmatrix} \frac{\partial l}{\partial w_0} \\ \frac{\partial l}{\partial w_1} \\ \cdots \\ \frac{\partial l}{\partial w_D} \end{bmatrix}$$

So the gradient of the likelihood function is a D+1 dimensional vector where D is the number of features (not counting the constant feature). In this form, the gradient ascent algorithm is;

While not converged

$$\widehat{w}^{(t+1)} \leftarrow \widehat{w}^{(t)} + \eta \nabla l\left(\widehat{w}^{(t)}\right)$$

And we converge when all partial derivatives are within a tolerance of zero.

$$\text{converged} = \frac{\partial l}{\partial w_0} < tolerance\ AND\ \frac{\partial l}{\partial w_1} < tolerance\ AND\ \ldots AND\ \frac{\partial l}{\partial w_D} < tolerance$$

In fact, rather than the likelihood function, we use log-likelihood function. From https://en.wikipedia.org/wiki/Likelihood_function#Log-likelihood:

> For many applications, the natural logarithm of the likelihood function, called the **log-likelihood**, is more convenient to work with. Because the logarithm is a monotonically increasing function, the logarithm of a function achieves its maximum value at the same points as the function itself, and hence the log-likelihood can be used in place of the likelihood in maximum likelihoodestimation and related techniques. Finding the maximum of a function often involves taking the derivative of a function and solving for the parameter being maximized, and this is often easier when the function being maximized is a log-likelihood rather than the original likelihood function.
>
> For example, some likelihood functions are for the parameters that explain a collection of statistically independent observations. In such a situation, the likelihood function factors into a product of individual likelihood functions. The logarithm of this product is a sum of individual logarithms, and the derivative of a sum of terms is often easier to compute than the derivative of a product. In addition, several common distributions have likelihood functions that contain

products of factors involving exponentiation. The logarithm of such a function is a sum of products, again easier to differentiate than the original function.

The partial derivative of the log-likelihood with respect for feature j is given by;

$$\frac{\partial l(w)}{\partial w_j} = \sum_{i=1}^{N} h_j(x_i)(1[y_i = +1] - P(y = +1|x_i, w))$$

N	is the number of data points (rows of feature matrix H)
i	is the index of the data (the ith data input, ith row of feature matrix H)
$h_j(x_i)$	is the jth feature of the ith data input
$P(y = +1\|x_i, w)$	is the prediction that x_i is positive for the given set of coefficients w.
$1[y_i = +1]$	is the indicator function. This outputs 1 if the known output value y_i is labeled as +1 and 0 if y_i is labeled as -1.

$$indicator(y_i) = 1[y_i = +1] = \begin{cases} 1\ if\ y_i = 1 \\ 0\ if\ y_i = -1 \end{cases}$$

This part of the derivative is the difference between the truth (known value of y) and the prediction that this is a positive sample.

$$(1[y_i = +1] - P(y = +1|x_i, w))$$

This is then weighted by the feature value (which is the word count for the jth word in the bag of words if we are doing sentiment analysis). So if the count is zero, the does not ending up counting at all. If the word count is high, then any difference is heavily weighted.

So when the weighted difference for each feature prediction falls below the tolerance (when it gets close to zero), then the algorithm converges.

Example of computing derivative for logistic regression

Remembering the indicator function (see above) and the score function;

$$Score(\boldsymbol{x_i}) = \widehat{\boldsymbol{w}}^T h(\boldsymbol{x_i}) = w_0 + w_1\vec{\boldsymbol{x}}_i[1] + w_2\vec{\boldsymbol{x}}_i[2]$$

and that

$$\hat{P}(y = +1|\boldsymbol{x_i}, \widehat{\boldsymbol{w}}) = sigmoid\big(Score(\boldsymbol{x_i})\big) = \frac{1}{1 + e^{-Score(\boldsymbol{x_i})}}$$

This part of the derivative;

$$(1[y_i = +1] - P(y = +1|x_i, w))$$

is then

$$indicator(y_i) - sigmoid(score(x_i))$$

We apply each feature value (each word count) to this expression and sum these to get the gradient of the log-likelihood function for that input. To show this, we can build a table that calculates the contribution of each term (the w_0 term does not contribute because w_0 is zero, so we only look at w_1 and w_2 terms in the table).

w0	0
w1	1
w2	-2

Data Point	x[1]	x[2]	y	score(x)	P(y=+1|x,w)	indicator(y)	w1 contrib to gradient x[1]*(indicator(y)-score(x)		w2 contrib to gradient x[2]*(indicator(y)-score(x))	
x1,y1	2	1	1	0	0.50	1	2*(1-0.5)	1.00	1*(1-0.50)	0.50
x2,y2	0	2	-1	-4	0.02	0	0*(0-0.02)	0.00	2*(0-0.02)	-0.04
x3,y3	3	3	-1	-3	0.05	0	3*(0-0.05)	-0.14	3*(0-0.05)	-0.14
x4,y4	4	1	1	2	0.88	1	4*(1-0.88)	0.48	1*(1-0.88)	0.12

Interpreting derivative for logistic regression

The partial derivative of the log-likelihood with respect for feature j is given by;

$$\frac{\partial l(w)}{\partial w_j} = \sum_{i=1}^{N} h_j(x_i)(1[y_i = +1] - P(y = +1|x_i, w))$$

We can call the term is inside the summation the delta for the coefficient – it is how much the coefficient is changed in each iteration of the gradient ascent;

$$\Delta_i = h_j(x_i)(1[y_i = +1] - P(y = +1|x_i, w))$$

If we say for the sake of illustration that $h_j(x_i) = 1$, so that we can concentrate on the other term within the summation, we can build a table of how that term affects the gradient and so how the coefficient is changed;

The affect on coeefficient when truth and prediction match or don't match			
given h(xi) = 1		Prediction is positive P(y=+1|xi,w) ~= 1	Predicition is negative P(y=+1|xi,w) ~= 0
truth is positive	yi = +1	1*(1 - 1) = 0 don't change coefficient	1*(1 - 0) = 1 increase coefficient
truth is negative	yi = -1	1*(0 - 1) = -1 decrease coefficient	1*(0 - 0) = 0 don't change coefficient

We can see in the two cases where the prediction matches the truth that we will not make any change in the coefficient. That is good because our coefficients already predict the truth – we don't want them to change. But look at the case in the upper-right where the truth is positive, but our prediction is negative. In this case the delta is positive; it will cause the coefficient to increase, which is what we want. Also look at the case in the lower-left where the truth is negative, but we have predicted positive; in that case the negative delta would have us decrease the coefficient, which is again what we want.

Summary of gradient ascent for logistic regression
The inner loop that updates w is;

$$w_j^{(t+1)} \leftarrow w_j^{(t)} + \eta \sum_{i=1}^{N} h_j(x_i)(1[y_i = +1] - P(y = +1|x_i, w))$$

The entire algorithm is;

Gradient Ascent for Logistic Regression

At t=1, initialize $\widehat{w}^{(1)} = 0$, or some other smart choice.
while $\left\|\nabla l(\widehat{w}^{(t)})\right\| > \varepsilon$ do
  for j = 0..D
    partial[j]= $\frac{\partial l(w)}{\partial w_j} = \sum_{i=1}^{N} h_j(x_i)\left(1[y_i = +1] - P(y = +1|x_i, w^{(t)})\right)$
    $\widehat{w}^{(t+1)} = \widehat{w}^{(t)} + \eta \times partial[j]$
  t = t + 1

N	is the number of data points (rows of feature matrix H)
i	is the index of the data (the ith data input, ith row of feature matrix H)
$h_j(x_i)$	is the jth feature of the ith data input
$P(y = +1\|x_i, w^{(t)})$	is the prediction that x_i is positive for the given set of coefficients w at iteration t.
$1[y_i = +1]$	is the indicator function. This outputs 1 if the known output value y_i is labeled as +1 and 0 if y_i is labeled as -1.
η	is the step size, which scales the derivative before it is added to the coefficient to get the estimated coefficient for the next step.

More plainly;

partial[j]= $\frac{\partial l(w)}{\partial w_j} = \sum_{i=1}^{N} h_j(x_i)\left(indicator(y_i) - sigmoid(score\left(x_i, w^{(t)}\right))\right)$

and this term can be computed once for the ith input row and then applied to each feature in the row;

$$indicator(y_i) - sigmoid(score\left(x_i, w^{(t)}\right))$$

Choosing step size
It turns out that the algorithm is very sensitive to this, so it is something that has to be chosen carefully. If the step size is too small, then the gradient ascent algorithm will require a lot of iterations to converge. If the step size is large, the convergence will not be smooth – it will look early on a like a sawtooth oscillation, but eventually smooth out and converge.

Careful with step sizes that are too large

If the step size is too large, early oscillations become very large and curve will continue to oscillate and so may not converge. In effect, if the step size is too large, then we may jump past, back and forth around the goal but never converge on it. Another aspect of step sizes that are too large is that their log-likelihood does not maximize, so it stays below those step sizes that converge.

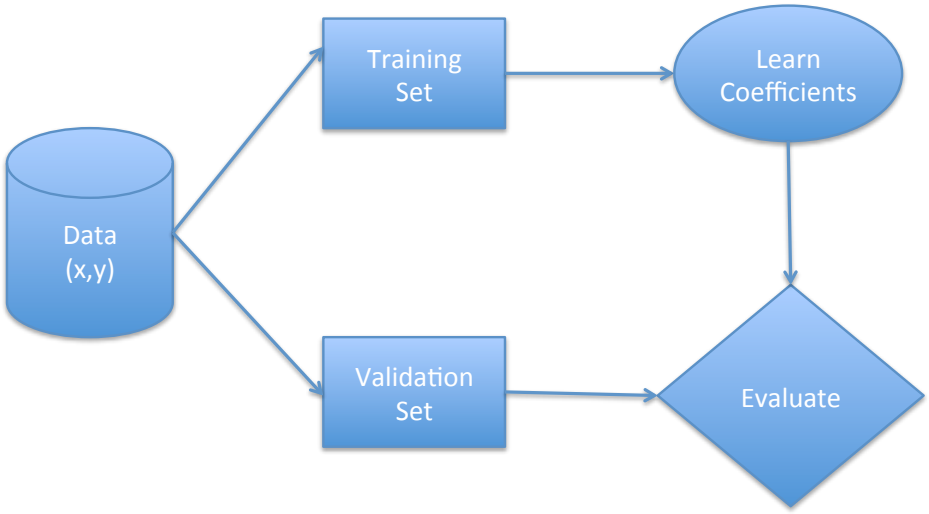
Rule of thumb for choosing step size

- Try several sizes that are exponentially spaced. For instance, try 10^{-4} and 10^{-6}.
- Goal
  - Find a step size that is too small (converges smoothly but slowly)
  - Find a step size that is too large (oscillates and does not converge)
  - Now look within this range for one that leads to the best training data likelihood.
- Advanced
  - Use the above procedure to find a starting step size $\eta^{(0)}$, but decrease the stepsize as the algorithm progresses.
  - For example, scale step size based on iteration; $\eta^{(t)} \leftarrow \frac{\eta^{(0)}}{t}$

Overfitting in classification

Evaluating a classifier

Vaidation



Classification error involves running the classifier on the data, then checking the results against the known values. Error is determined by fraction of errors

$$error = \frac{count(mistakes)}{total\ data\ points}$$

So the best possible error value is 0.0

The accuracy is the complement of the error; it is the number correct over the total data points. The best possible accuracy is 1.0

$$accuracy = 1 - error = \frac{count(correct)}{total\ data\ points}$$

Overfitting

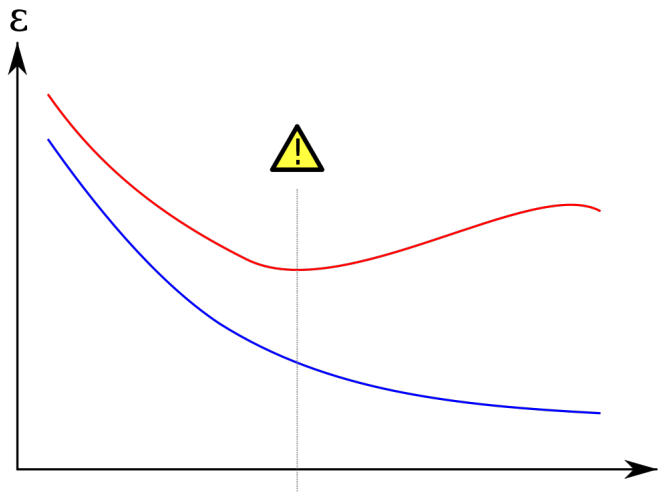
A model is overfit if there exists another model that is less well fit to the training data (has a higher training error) AND that has less generalization error. Another way to say this is that a model is overfit if the model is better fitted to the training data than another model but less well fit to the general data than that other model. This would be seen on the graph of model error vs complexity as a model where the slope of the training error is negative but the slope of the test error is

positive. In other words, we have gotten to a place on the graph where generalization error is increasing. For instance, look at the illustration above; Error versus Model Complexity, and look at the fourth tick on the horizontal axis. At that place the training error is going down, but the generalization error is going up. More formally,

A model w' is over-fitted if there exists another model w'' where

- Training error of w'' > training error of w'
  w'' has higher training error that w'
- True error of w'' < true error of w'
  w'' has lower true error than w'

Here is an illustration from this https://en.wikipedia.org/wiki/Overfitting of how training error (the bottom, blue line) can decrease, but generalization error (the top, red line) can increase. Those solutions to the right of the line, where the true error is increasing, are overfit; there exists a solution with higher training error but lower true error.



We will tend towards overfitting when we have small data sets or when we choose very complex models (lots of parameters and/or high order polynomials). As with Linear Regression, **overfit models are characterized by high coefficient values**. In classification, **overfit classification models have complex decision boundaries**, rather than smooth boundaries. The resulting model may fit the training data perfectly, but will do poorly on other data.

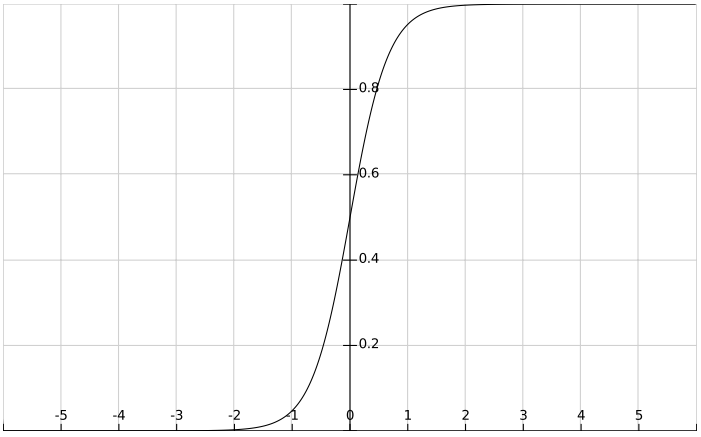
Over-fitting also affect our probability estimates. To get our estimated probability, we map the score function, which can range from $-\infty \; to \; +\infty$, to a probability which must be in the range 0 to 1.

$$Score(\boldsymbol{x_i}) = \widehat{\boldsymbol{w}}^T h(\boldsymbol{x_i}) = w_0 + w_1 \vec{\boldsymbol{x}}_{\boldsymbol{i}}[1] + w_2 \vec{\boldsymbol{x}}_{\boldsymbol{i}}[2]$$

and

$$\hat{P}(y = +1|\boldsymbol{x_i}, \hat{\boldsymbol{w}}) = sigmoid\big(Score(\boldsymbol{x_i})\big) = \frac{1}{1 + e^{-Score(\boldsymbol{x_i})}}$$

Overfitting results in large coefficients and these drive the score functions to be large positive or negative values, which in turn drives our prediction to be very close to either 0 or 1. In effect, the **large coefficients make the sigmoid curve very steep and narrow**, so we end up mapping to either very high probabilities or very low probabilities.



This results in very **narrow regions of uncertainty**. So, in overfitted classification models, our **predictions tend to be over-confident** in their positive or negative probabilities; there are no grey areas anymore. The model acts as if it can confidently predict every training data input's class (which it can), but this does not generalize.

L2 regularized logistic regression

Penalizing large coefficients to mitigate overfitting
We want to balance

- Measure of Fit - How well the coefficients fit the data
  Our Measure of Fit is the the data likelihood and we want this to be large (meaning a good fit). We generally use the log-likelihood because it makes the math easier.
- Magnitude of the coefficients
  Our measure of coefficient magnitude will penalize the Total Quality when coefficients get large.

Total Quality = Measure of Fit – Measure of Coefficient Magnitude

There are two measures of Coefficient Magnitude that we have used in the past and will use here;

- L2 Norm Squared = sum of the squared coefficients.
  Squaring the coefficients does not affect their relative sizes, but eliminates negative numbers.
  - $\|w\|_2^2 = {w_0}^2 + {w_1}^2 + {w_2}^2 + \cdots + {w_D}^2$
- L1 Norm – sum of the absolute values of the coefficients (also called the Sparse solution).
  - $\|w\|_1 = |w_0| + |w_1| + |w_2| + \cdots |w_D|$

Both of these measures will get positively larger as the coefficients get larger.

L2 regularized logistic regression

We will start with the L2 Norm Squared, so this is called L2 Regularized Logistic Regression. Our quality measure in that case is;

Total Quality = Measure of Fit – Measure of Coefficient Magnitude
$= l(w) - \lambda\|w\|_2^2$

λ is our tuning parameter and it has this effect;

If $\lambda = 0$, then the solution becomes the standard un-penalized solution, which can lead to an overfit model with low bias (it can fit whatever training data it is given), but very high variance (small changes in training data lead to a big change in the model).

if $\lambda = \infty$, then all coefficients will be penalized, leading the w = 0, all coefficients go to zero. This creates a constant model with low variance (we get the same model no matter what the data is), but high bias (it doesn't fit the data well).

We want $0 < \lambda < \infty$ that balances fit and coefficient magnitude; we want it to handle the bias-variance trade-off.

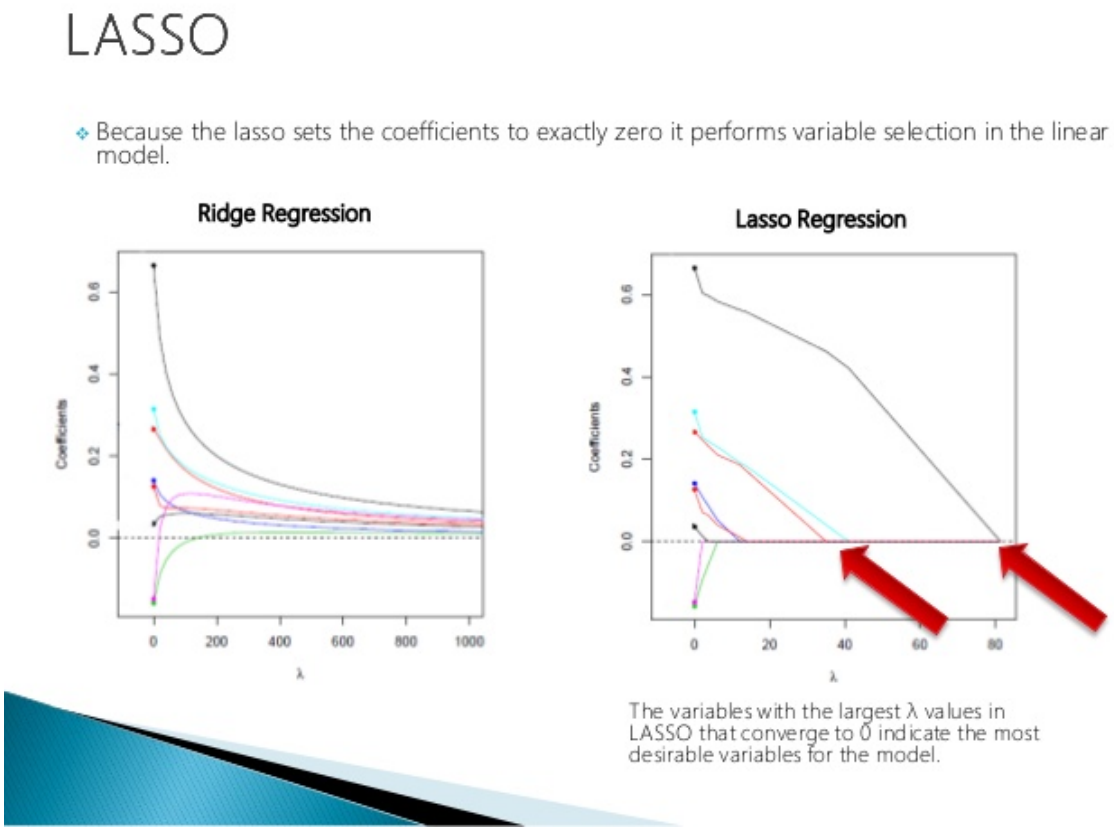
We choose λ using either either;

- A validation set, separate from the training and test sets, if we have enough data.
- Cross Validation if we don't have enough data for a validation set.
- Some or possibly some other validation method, but we don't rely on the training set and we don't use the validation error as a measure of generalization error.

Effect of L2 regularization in logistic regression

As λ get's larger, the coefficients get smaller. In the case of the L2 Norm Squared penalty, all the coefficients get smaller and approach zero as λ

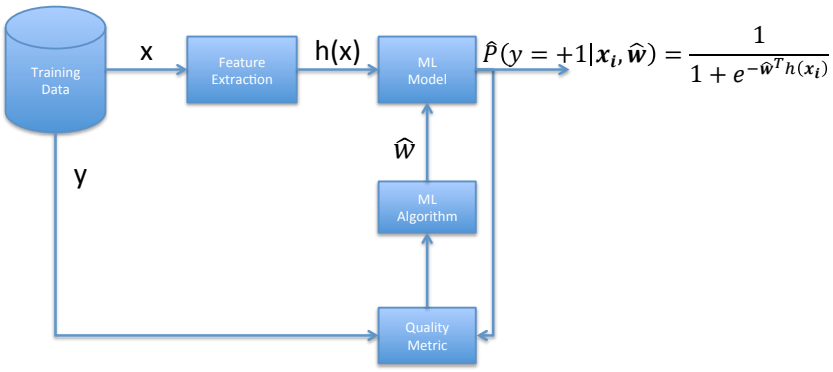
approaches infinity. This can be seen in the plot of the Coefficient Paths. Below is a slide taken from this presentation http://www.slideshare.net/DerekKane/data-science-part-xii-ridge-regression-lasso-and-elastic-nets by Derek Kane that compares the coefficient paths for L2norm squared cost (in this case Ridge Regression) and Lasso Regression (L1norm code). For now, focus on Ridge Regression and notice that all coefficients are reduced as the tuning parameter gets larger, but none go to zero.



This regularization also reduces 'over-confidence' by maintaining a healthy regions of uncertainty around the decision boundary.

Learning L2 regularized logistic regression with gradient ascent
The workflow is the same, but now our quality metric involves both the likelihood function and the L2 Norm Squared penalty.

Machine Learning Linear Classifier Workflow



We need to take the derivative of;
Total Quality $= l(w) - \lambda\|w\|_2^2$

so $\frac{\partial l(w)}{\partial w_j} - \lambda \frac{\partial \|w\|_2^2}{\partial w_j}$

We already know the derivative of the log-likelihood function

$$\frac{\partial l(w)}{\partial w_j} = \sum_{i=1}^{N} h_j(x_i)(1[y_i = +1] - P(y = +1|x_i, w))$$

The derivative of the L2 Norm Squared penalty is;

$$\frac{\partial \|w\|_2^2}{\partial w_j} = \frac{\partial [{w_0}^2 + {w_1}^2 + {w_2}^2 + \cdots {w_j}^2 + \cdots + {w_D}^2]}{\partial w_j} = 2w_j$$

So the total derivative is the what we had before minus $2\lambda w_j$

$$\frac{\partial l(w)}{\partial w_j} - 2\lambda w_j$$

The $-2\lambda w_j$ term has the effect of moving the result, and so the coefficient, towards zero;

	-2λWj	impact on Wj
Wj > 0	< 0	Wj decreases and moves closer to zero.
Wj < 0	> 0	Wj increases and moves closes to zero.

L2 Regularized Logistic Regression

At t=1, initialize $\widehat{w}^{(1)} = 0$, or some other smart choice.
while $\left\|\nabla l(\widehat{w}^{(t)})\right\| > \varepsilon$ do
  for j = 0..D
    partial[j]= $\frac{\partial l(w)}{\partial w_j} = \sum_{i=1}^{N} h_j(x_i)\left(1[y_i = +1] - P(y = +1|x_i, w^{(t)})\right)$
    $\widehat{w}^{(t+1)} = \widehat{w}^{(t)} + \eta \times (partial[j] - 2\lambda w_j)$
  t = t + 1

N	is the number of data points (rows of feature matrix H)
i	is the index of the data (the ith data input, ith row of feature matrix H)
$h_j(x_i)$	is the jth feature of the ith data input
$P(y = +1\|x_i, w^{(t)})$	is the prediction that x_i is positive for the given set of coefficients w at iteration t.
$1[y_i = +1]$	is the indicator function. This outputs 1 if the known output value y_i is labeled as +1 and 0 if y_i is labeled as -1.
$2\lambda w_j$	is the derivative of the L2norm Squared penalty
η	is the step size, which scales the derivative before it is added to the coefficient to get the estimated coefficient for the next step.

Note: the L2 penalty is not generally applied to the intercept (the constant feature). Having a large intercept is not indicative of overfitting, so there is no reason to penalize large intercepts.

Sparse logistic regression with L1 regularization

Sparsity it this case refers to w where many $w_j = 0$. If unimportant coefficients are made to be zero, then we get some benefits;

- Efficiency
  When we have a very large number of coefficients (models have been created with 100 billion parameters), then making a prediction becomes very computationally expensive. However, if we know which coefficients are zero, we can ignore that term in the prediction, since terms with zero coefficients will be zero.
- Interpretability
  Features with very small coefficients don't contribute much to the prediction. By driving them to a zero coefficient, we can more clearly interpret the results of a prediction by looking at only non-zero coefficients.

Total Quality = Measure of Fit – Measure of Coefficient Magnitude
= $l(w) - \lambda\|w\|_1$

λ is our tuning parameter and it has this effect;

If $\lambda = 0$, then the solution becomes the standard un-penalized solution were no coefficients are penalized to zero.

if $\lambda = \infty$, then all coefficients will be penalized, leading the w = 0, all coefficients go to zero.

We want $0 < \lambda < \infty$ that balances fit and L1 penalty; some coefficients will be penalized to zero, but others will not. This can be seen in the earlier chart that shows how increasing λ affects the coefficients (the coefficient path chart) – the Lasso regression shows these paths. As λ increases, coefficients move toward zero, some faster than others, then generally stay at zero.

Efficiency of Prediction - we only take into account those w that are not zero

$$\hat{y}_i = sign\left(\sum_{\hat{w}_j \neq 0} \hat{w}_j h_j(x_i)\right)$$

Total Quality = Measure of Fit – Measure of Coefficient Magnitude
= $l(w) - \lambda\|w\|_1$