DATA SCIENCE II: Machine Learning MTH 9899 Baruch College

Lecture 2: Machine Learning

Adrian Sisser

April 3, 2019

Outline

Linear Methods

- One hard problem with Regression, is to choose which variables are relevant to your model.
- You might have dozens of predictors, most of which might be irrelevant.
- This problem is much harder in the context of low R^2 and correlated predictors.

One technique is “Best Subset” regression, which finds the optimal subset of size k regressors for each value k. Computationally, this is very expensive.

Forward Stepwise Regression

- We look for the best variable from the remaining ones at each stage and add it into the regression,
- This again this gives us k models
- This is a greedy algorithm that might not be best.
- It will ALWAYS be worse than the corresponding best subset, but has lower variance.
- Caveat: Imagine you have 2 highly correlated x variables that predict y, but both are measured with significant noise. What will Forward Stepwise do? What would you prefer?

Backward Stepwise Regression

- We start off with all variabales and remove the ’worst’ one at each stage.
- Same Caveat as Forward Stepwise Regression

Model Selection

- AIC - Akaike Information Criterion. This can be shown to be asymptotically equivalent to N-Fold Cross-Validation [Stone, 1977].

$$\mathrm{AIC} = 2k - 2\ln(\mathcal{L})$$

- BIC - Bayesian Information Criterion

$$\mathrm{BIC} = \ln(n)k - 2\ln(\mathcal{L})$$

Lasso Regression

- Lasso is similar to Ridge Regression, except we use the L1 penalty:

$$\min_{\beta^L} \|Y - X\hat{\beta}^L\| + \lambda \|\hat{\beta}^L\|_1$$

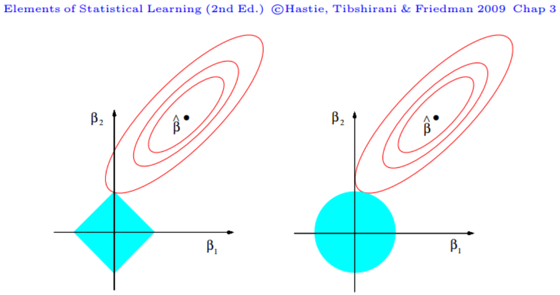
- Can we compute an analytical solution to this?

LARS

LARS - Least Angle Regression, is a technique with an intuitive geometric explanation that leads to a very efficient implementation of LASSO. For more details, the class text, Elements of Statistical Learning provides the best intuition of how it works.

Lasso vs Ridge Regression

- Lasso works well for Feature Selection
- Ridge works well for correlated features

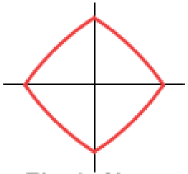

Elements of Statistical Learning (2nd Ed.) ©Hastie, Tibshirani & Friedman 2009 Chap 3
β_2
$\hat{\beta}$
β_1
β_2
$\hat{\beta}$
β_1

Elastic Net Regression

- Let's get the best of both worlds - we can use an L1 and L2 Penalty:

$$\min_{\beta^{EN}} \|Y - X\hat{\beta}^{EN}\| + \lambda_1\|\hat{\beta}^{EN}\|_1 + \lambda_2\|\hat{\beta}^{EN}\|_2$$

- How do we figure out λ_1?



Elastic Net

K-Means Introduction

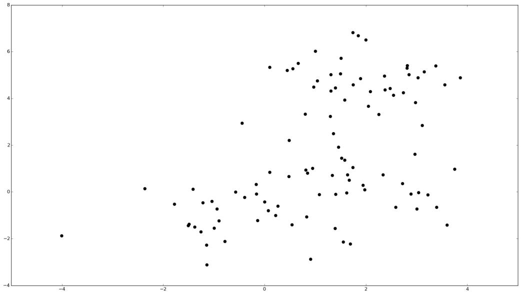
K-Means is a method of classifying points into 'groups' or 'clusters', based on their 'proximity'. For traditional K-Means, the proximity measure is Euclidean distance, ie $\| \cdot \|_2$. If we want to form K clusters, we minimize as follows:

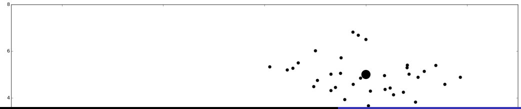
$$\arg\min_S \sum_{i=0}^{K} \sum_{x \in S_i} \|x - C_i\|$$

where C_i is the geometric center of all of the points belonging to S_i, the set of all points in cluster i. This is equivalent to assuming the points are normally distributed around each center.

K-Means Introduction

Here is sample of clusters in 2 dimensions.





K-Means Algorithm

So how do we do this? We need an algorithm to solve the minimization problem from earlier:

$$\arg\min_{S} \sum_{i=0}^{K} \sum_{x \in S_i} \|x - C_i\|$$

Solving this problem directly isn't tractable - in fact, it's NP-hard for almost all cases.

KMeans Algorithm

Algorithm 1 KMeans Algorithm (Lloyd's Algorithm)

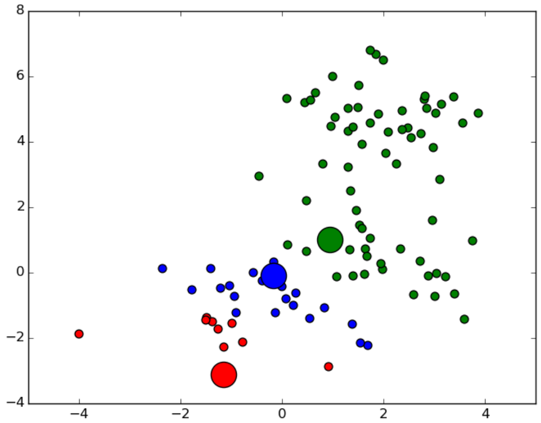
```
Require: $N > K > 1$
  $centroids \leftarrow$ select $K$ random entries from $points$
  repeat
    for $i < N$ do
      $assigned\_centers[i] \leftarrow$ find_nearest_center$(points[i])$
    end for
    for $i < K$ do
      $centroids[i] \leftarrow$ find_centroid$(i)$
    end for
  until $assigned\_centers$ does not change
```

Initialization

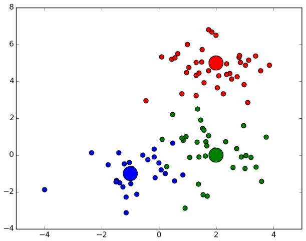
So how do we *randomly* assign the initial clusters? There are a few popular choices:

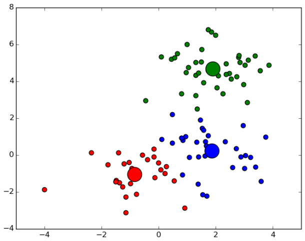
- Choose K random points from the initial list (Forgy Method).
- The Random partition method assigns each point a cluster at random, then calculates the centroids.

An example



An example





Our original data

Actual Centers		Calculated Centers
(2.0, 5.0)	$\rightarrow$	(1.9, 4.7)
(2.0, 0.0)	$\rightarrow$	(1.9, 0.2)
(-1.0, -1.0)	$\rightarrow$	(-0.8, -1.1)

Notes

- The algorithms discussed will only find a **LOCAL** minimum. To be sure we're getting a near-optimal solution, we should repeat this with different starting centroids.
- How do we know how many clusters, K, to look for? Adding more clusters will always improve the metrics.

GMeans

G-Means offers a way for us to intuit K:

Algorithm 2 GMeans Algorithm

```
  $K \leftarrow 0$
  **repeat**
    $K \leftarrow K + 1$
    $centers \leftarrow \text{KMeans}(points, K)$
  **until** $(points - centers) \sim \mathcal{N}$
```

The EM Algorithm

“Expectation-Maximization” is a generic algorithm for estimating MLE parameters. The derivation is complex, and we will go through it quickly here. An excellent reference is Andrew Ng’s ML Notes.

$$\begin{aligned}
X &= \{x_0, x_1, ..., x_{n-2}, x_{n-1}\} \\
Z &= \{z_0, z_1, ..., z_{n-2}, z_{n-1}\} \text{ \# These are are our latent variables} \\
\mathcal{L}(\theta|X,Z) &= \prod_{i=0}^{N} P(x_i;\theta) \\
\ell(\theta|X,Z) &= \sum_{i=0}^{N} \log P(x_i;\theta) \\
\ell(\theta|X,Z) &= \sum_{i=0}^{N} \log \sum_{j=0}^{K} P(x_i, z_i = j;\theta)
\end{aligned}$$

The EM Algorithm

Let's define Q_i as a probability distribution of z_i. Now we can say:

$$
\begin{aligned}
\ell(\theta|X,Z) &= \sum_{i=0}^{N} \log \sum_{j=0}^{K} P(x_i, z_i = j; \theta) \\
\ell(\theta|X,Z) &= \sum_{i=0}^{N} \log \sum_{j=0}^{K} Q_i(z_i = j) \frac{P(x_i, z_i = j; \theta)}{Q_i(z_i = j)} \\
\ell(\theta|X,Z) &= \sum_{i=0}^{N} \log \mathbb{E}_{z \sim Q_i} \frac{P(x_i, z_i = j; \theta)}{Q_i(z_i = j)} \\
\ell(\theta|X,Z) &\geq \sum_{i=0}^{N} \mathbb{E}_{z \sim Q_i} \log \frac{P(x_i, z_i = j; \theta)}{Q_i(z_i = j)} \\
\ell(\theta|X,Z) &\geq \sum_{i=0}^{N} \sum_{j=0}^{K} Q_i(z_i = j) \log \frac{P(x_i, z_i = j; \theta)}{Q_i(z_i = j)}
\end{aligned}
$$

The EM Algorithm

The next steps are tricky (again, refer to Andrew Ng's ML Notes for more details). We said that Q_i was a PDF for z_i, so let's choose a good one:

$$\begin{aligned} Q_i(z_i) &= \frac{P(x_i, z_i; \theta)}{\sum_j P(x_i, z_i = j; \theta)} \\ &= P(z_i | x_i; \theta) \end{aligned}$$

Now, we're ready to look at the algorithm itself.

The EM Algorithm

Algorithm 3 EM Algorithm

```
$\theta^0 =$ initial guess
$m \leftarrow 1$
**repeat**
    $Q_i^m = P(z_i|x_i;\theta^m)$
    $\theta^{m+1} = \arg\max_\theta \sum_{i=0}^N \sum_{j=0}^K Q_i^m(z_i = j) \log \frac{P(x_i,z_i=j;\theta^{m+1})}{Q_i^m(z_i=j)}$
    $m \leftarrow m + 1$
**until** convergence of $\ell$
```

Take careful note of θ in the MLE step. Proof of convergence can be found in the Ng reference mentioned above.

An EM Application: Soft KMeans

Let's look at this in the context of a 'soft' KMeans Algorithm with 2 clusters. This means that instead of assuming each point is in a given cluster, C, we'll assign a probability that it's in each cluster. Here's our setup:

$$
\begin{aligned}
X &= \{x_0, x_1, ..., x_n\} \\
Z &= \{z_0, z_1, ..., z_n\} \\
\theta &= \{\mu_0, \sigma_0^2, \mu_1, \sigma_1^2, \pi_0, \pi_1\}
\end{aligned}
$$

Here, π refers to the probability that a point belongs to C_i.

Soft KMeans: The “E” Step

$$
\begin{aligned}
Q_i(z_i = j) &= \frac{P(x_i, z_i = j; \theta)}{\sum_m P(x_i, z_i = m; \theta)} \\
&= \frac{\phi_j(x_i; \theta)}{\sum_m \phi_m(x_i; \theta)}
\end{aligned}
$$

$Q_i(z_i = j)$ is the probability that point i belongs to C_j. Since we don’t make a hard assignment to any cluster, this is why we call this a ’Soft K-Means’ algorithm.

Soft KMeans: The "M" Step

To make notation simpler, now that we've done the "E" step, we'll say $w_{i,j}$ is the probability that point i is in C_j. The "M" step is:

$$
\begin{aligned}
&\arg\max_{\theta} \quad \textstyle\sum_{i=0}^{N}\sum_{j=0}^{K} \quad w_{i,j}\log\frac{P(x_i, z_j;\theta)}{w_{i,j}} \\
&\arg\max_{\theta} \quad \textstyle\sum_{i=0}^{N}\sum_{j=0}^{K} \quad w_{i,j}\log\frac{P(x_i|z_j;\theta)P(z_j)}{w_{i,j}} \\
&\arg\max_{\theta} \quad \textstyle\sum_{i=0}^{N}\sum_{j=0}^{K} \quad w_{i,j}\log\frac{\phi_{j;\theta}(x_i)\pi_j}{w_{i,j}}
\end{aligned}
$$

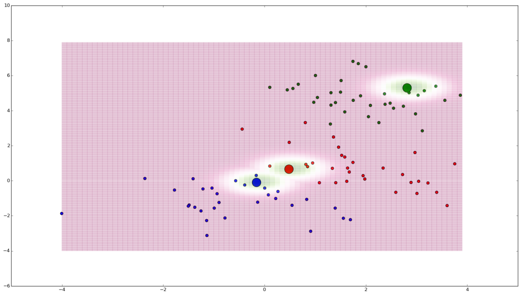
Soft KMeans: The “M” Step

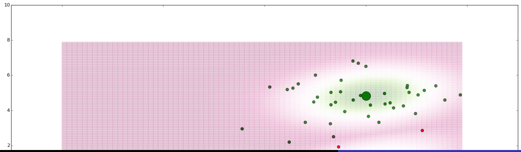
$$\arg\max_{\theta} \quad \textstyle\sum_{i=0}^{N} \sum_{j=0}^{K} \quad w_{i,j} \log \frac{\phi_{j;\theta}(x_i)\pi_j}{w_{i,j}}$$

If we take our function from before, and take some derivatives, we get very simple update rules:

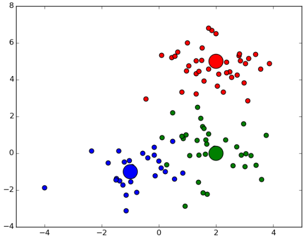
$$\begin{aligned}
\mu_j &= \frac{\sum_{i=0}^{N} w_{ij} x_i}{\sum_{i=0}^{N} w_{ij}} \\
\pi_j &= \frac{1}{N} \sum_i w_{ij} \\
\sigma_j^2 &= \frac{\sum_{i=0}^{N} w_{ij} \|x_i - \mu_j\|_2}{\sum_{i=0}^{N} w_{ij}}
\end{aligned}$$

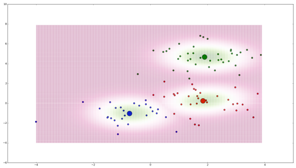
An example





An example





Our original data

Actual Centers		Calculated Centers
(2.0, 5.0)	$\rightarrow$	(1.9, 4.7)
(2.0, 0.0)	$\rightarrow$	(1.8, 0.2)
(-1.0, -1.0)	$\rightarrow$	(-0.7, -1.0)

Data Cleaning

- Last lecture, we discussed Winsorization and MAD filtering - don't forget them.
- Normalization, ie standardizing $\mu = 0$ and $\sigma = 1$ is essential to many ML techniques and will almost always produce a better result.
- When Normalizing stock returns, don't just use a cross-sectional volatility. Use a time-series volatility adjustment first, then, possibly apply a cross-sectional normalization.

Assume you have stock returns $r_{i,j}$ where i refers to a stock, and j refers to a time index.

$$r'_{i,j} = \frac{r_{i,j} - \mu_j}{\sigma_j}$$

vs

$$r'_{i,j} = \frac{r_{i,j} - \mu_{i,j-K\ldots j}}{\sigma_{i,j-K\ldots j}}$$

Clearly, the first method ignores the natural variance in stock volatilities. If you are also looking to do cross-sectional normalization, you should do it on the output of the second equation.

Sample Weights

Sample weights can help us overcome several obstacles in ML:

- Different points might have different amount of information. Do we believe a price point from a mega-cap company like AAPL as much as we do a $100mm company like ASTI?
- Weights can be use to control for heteroskedacity, similar to the normalization we described above. For example, Barra uses the $\frac{1}{\sigma^2}$ where σ is the residual volatility from a simple CAPM regression.
- We can weight more recent points more heavily in a long-term fitting. For example, Barra uses an exponentially decaying weight for calculating Σ. Note that decaying the weights over timeresults in a smaller effective sample size.

Effective Sample Size

When we have n weighted samples, and we have to think about our 'effective n'. If the weights are very uneven, we have many fewer effective points than we think. Let's take an example:

- We have 10 points in x with weights of $[1, 1, 1, 1, 1, 1, 1, 1, 1, 1000000]$
- We calculate $\mu = 1$ and $\sigma = 2$

What's the standard error of x? It's not $\frac{2}{\sqrt{10}}$!!

Effective Sample Size

We have to calculate the 'effective n':

$$n_{\text{eff}} = \frac{(\sum_{i=0}^{N} w_i)^2}{\sum_{i=0}^{N} (w_i^2)}$$

A quick look tells us this formula makes sense:

- When $w_i = 1$ for all i, n_{eff} = n
- When one point has infinite weight, n_{eff} approaches 1.

Sample Weights

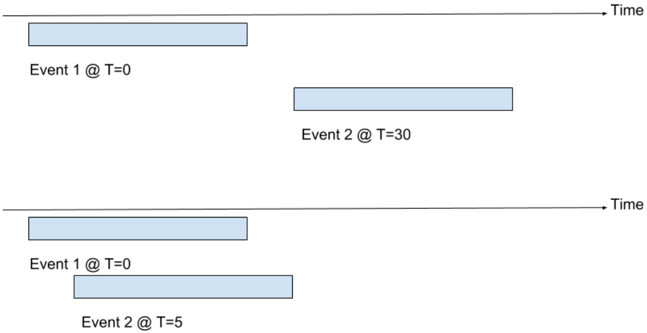
Another issue with sample weights is overlapping observations. Imagine you're modelling the returns of a stock for 21 days after it is upgraded by an analyst.

- For a stock with an upgrade 30 days apart, we have no issues.
- If the same stock has 2 upgrades 5 days apart, we now have overlapping returns in our y variable. If the first event happens at $T = 0$, the period from $T = 5$ to $T = 20$ is counted twice, even though the returns are largely the same.

This is a problem. You are effectively doubling the weight of this point. You're decreasing the effects of randomly sampling points through techniques like Bagging with replacement, etc that we will discuss later.

Lopez de Prado proposes weighting with a 'Uniqueness' measure in AFML - See Chapter 4 for more details.

Overlapping Samples



Why Training is Important

It can be show that a sufficiently large NN can learn any function. The hardest part is training the weights in the network. Why is it hard?

- The number of weights in a network is huge. Connections between an N and M neuron layer create a total of $\mathcal{O}(NM)$ connections.
- A training algorithm on a deep network based on gradients suffers from the *Vanishing/Exploding Gradient Problem*.

In summary, if we design a network that can learn anything, it's REALLY hard to make it learn what we want.

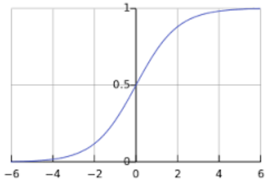
Basic Training

- When we talk about training a NN, we're talking about finding weights and biases for the neurons.
- We want to adjust the weights, such that we reduce the error.
- To do this, we'll follow the gradient of the error with respect to the weights.

Activation Functions

Before we can talk about training, we need to talk about Activation Functions. For now, let's talk about sigmoid - a very common and simple activation function. Later, we'll see why we need this.

$$\begin{aligned} \varphi(z) &= \frac{1}{1+e^{-z}} \\ \frac{d\varphi}{dz} &= \varphi(z)(1-\varphi(z)) \end{aligned}$$

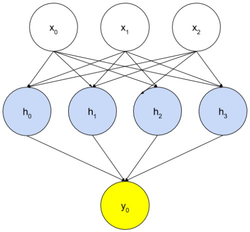


Backpropagation

Based on the last lecture, let's assume that a neuron can be modeled as:

$$n_j = \mathcal{A}(\sum_i w_{ij} n_i + b_j)$$

And a simple network topology:



Backpropagation

We're going to come up with a learning rule. The goal is to reduce the total error. For a simple regression problem:

$$E = \sum_i \frac{1}{2}(\hat{y}_i - y_i)^2$$

Now that we know the error, let's calculate the derivative with respect to $\hat{y}$:

$$\frac{\partial E}{\partial \hat{y}_i} = (\hat{y}_i - y_i)$$

Backpropagation

Now, we need to propagate the errors backwards, through the NN. First, let's setup some notations:

- x_i Row i of the feature matrix, X.
- $n_{l,i}$ is the value of the output of neuron i in layer l
- $z_{l,i}$ is the value of the input to the activation function for neuron i in layer l
- $\hat{y}_i$ The prediction for sample i, ie the output of the neuron
- $w_{l,ij}$ The weights from the (l) layer, neuron i, to the $l+1$ layer, neuron j.
- $b_{l,j}$ The bias of of neuron j in layer l

Backpropagation

Let's look at the error term for the hidden layer in the topology we talked about. Don't forget the activation function is the sigmoid function mentioned earlier.

$$\begin{aligned}
\frac{\partial E}{\partial w_{h,i0}} &= \frac{\partial E}{\partial n_{o,0}} \frac{\partial n_{o,0}}{\partial w_{h,i0}} \\
\frac{\partial E}{\partial w_{h,i0}} &= \frac{\partial E}{\partial n_{o,0}} \frac{\partial \mathcal{A}(z_{o,0})}{\partial w_{h,i0}} \\
\frac{\partial E}{\partial w_{h,i0}} &= \frac{\partial E}{\partial n_{o,0}} \frac{\partial \varphi(z_{o,0})}{\partial z_{o,0}} \frac{\partial z_{o,0}}{\partial w_{h,i0}} \\
\frac{\partial E}{\partial w_{h,i0}} &= (\hat{y}_i - y_i)\varphi(z_{o,0})(1 - \varphi(z_{o,0}))n_{h,i}
\end{aligned}$$

Backpropagation

Now we can turn this into a learning rule.

$$w'_{l,ij} = w_{l,ij} - \eta \frac{\partial E}{\partial w_{l,ij}}$$

η is called the learning rate. We'll discuss it in much more detail later, but we can use it to control the rate of convergence.

Backpropagation

How do we extend this to a network with more layers? We just keep going and apply the chain rule. Eventually, we'll end up with a very long chain, where we are taking the derivative of every activation function with respect to the previous layer.

https://google-developers.appspot.com/machine-learning/crash-course/backprop-scroll/