M/L Commando Course 2018

Session 1a - Regression

Russell Moore

ALTA / Computer Laboratory

July 2018

Introduction

Last time we looked at supervised classification, probably *the* classic M/L task.

The other major classic tasks are *regression* and *clustering*.

In this session we'll look at regression, both using data we've seen (irises) and a new dataset (Boston house prices).

Regression - the problem

- In classification we looked at the features of a sample $\mathbf{x}$ and used these to predict the class of $\mathbf{x}$.
- Regression is similar, but instead of predicting the class, we use the known features of $\mathbf{x}$ to predict unknown features of $\mathbf{x}$.
- There are lots of cases of natural relationships that can be modelled this way: e.g. human height vs weight, car age vs price, petal length vs width, and so on.

Regression problems

- We have a set of samples $X = \{\vec{x_i}\}$ where i is an integer index
- Each $\vec{x_i}$ is a vector of m numerical features: $\vec{x_i} = [f_0 \ f_1 \ f_2 \ \cdots \ f_m]$
- The problem: we get a new sample of data, but it's incomplete $\vec{x_?} = [f_0 \ f_1 \ f_2 \ \cdots \ \mathbf{f}_? \ \cdots \ f_m]$
- Can we predict $f_?$ using the knowledge (patterns) given to us by X?
- This is known as a *regression* problem
- There are lots of cases of natural relationships that can be modelled this way: e.g. human height vs weight, car age vs price, petal length vs width, and so on.
- Note that regression outputs are continuous values.

The magic *hypothesis function*

- For consistency, for $\vec{x_i}$ call the 'target' feature y_i
- Imagine we have a magic function called $h(\vec{x_i}; params_X) \rightarrow y_i$
- The h stands for *hypothesis function*
- $params_X$ is some bunch of calibration settings that make it work, based on our known training data X.
- Can we build h?

One dimension: $A + Bx_i$

- Start by considering data pairs like (x, y)
- i.e. x, y both one-dimensional
- We've seen this: irises, petal width vs petal length
- if we assume that a line can fit our data we might hope to predict each y_i value using: $y_i^{pred} = A + Bx_i$ for some intercept A and slope B
- This is our hypothesis function: $h(x_i; params_X) \rightarrow y_i$
- $params_X = [A, B]$ in this case
- Generally we refer to $params_X$ as $\vec{w}$: the 'weights vector' for h.
- Then we can use vector components, to write $y_i^{pred} = w_0 + w_1 x_i = h(x_i;\ \vec{w})$
- Will be useful later when dealing with higher dimensions.

Squared Error, and Cost Function

- So far so good: given some $\vec{w}$ we can use $h(x_i; \vec{w})$ to get a prediction y_i^{pred}
- But is the prediction any good?
- Take the difference, square it: $sqerr_i = (y_i^{pred} - y_i)^2$
- Squared Error $sqerr_i$ is always positive and grows quickly for large outliers. Useful.
- Rewrite in terms of our existing toolkit; for the ith element: $sqerr_i(X, Y; \vec{w}) = (h(x_i; \vec{w}) - y_i)^2$
- Add all these sqerrors together, and call this our *cost function*, normally denoted Q:

$$Q(X, Y; \vec{w}) = \sum_i sqerr_i(X, Y; \vec{w})$$

- Smaller values of Q mean less error. So in seeking good predictions, we seek to *minimise* the cost function.

Turn the problem around

- Now we get clever - let's 'turn the problem around' so that we think of the weights $\vec{w}$ as being the variables.

$$Q(\vec{w};\ X, Y) = \sum_i sqerr_i(\vec{w};\ X, Y)$$

- The formulae are the same but rather than 'getting points off the line' defined by the params in $\vec{w}$, we instead seek to get the line from the points.
- 'All we have to do' is find $\vec{w}$ that yields the minimum value of Q! But how?
- If we can get the gradient of a function, we can use it to follow the function's slope (in our case downwards)...

Gradient Descent - set up

- Q is a **sum of functions** (squared errors) across our weights for a given set of data:

$$Q(\vec{w};\ X, Y) = \sum_i sqerr_i(\vec{w};\ X, Y)$$

- **Key fact: the derivative of a sum = the sum of its derivatives.**
- So we can indeed get the gradients of Q:

$$\nabla Q(...) = \sum_i \nabla sqerr_i(...)$$

- the ∇ symbol ("del") shows that we are taking the partial derivative along each of the function's axes.

Gradient Descent - calculus

- 'Break open' our 1D example, $\vec{w} \equiv [A, B]$

$$Q([A, B];\ X, Y) = \sum_i sqerr_i([A, B];\ X, Y)$$

- since for us $sqerr_i([A, B];\ X, Y) = (Ax_i + B - y_i)^2$ we can work out the gradients (use chain rule!):

$$\nabla_A sqerr_i([A, B];\ X, Y) = 2x_i(Ax_i + B - y_i)$$
$$\nabla_B sqerr_i([A, B];\ X, Y) = 2(Ax_i + B - y_i)$$

- ..and we use these, averaged over N samples and scaled by some fractional *learning rate* α, to (repeatedly until some limit) update A and B as follows:

$$A' := A - \frac{\alpha}{N} \sum_i \nabla_A sqerr_i([A, B],\ X, Y)$$

$$B' := B - \frac{\alpha}{N} \sum_i \nabla_B sqerr_i([A, B],\ X, Y)$$

Gradient Descent - generalising

- Remember we packaged the weights into $\vec{w}$ instead of clumsily listing A, B (and for higher dimensions: C, D ... Z ...)
- Then we can update all weights in one formula: it can just be written $\vec{w}' := \vec{w} - \alpha \sum_i \nabla err_i(\vec{w};\ X, Y)$
- Better yet, since our cost function Q already contains the summation step we can shorten this to:
  $\vec{w}' := \vec{w} - \alpha \nabla Q(\vec{w};\ X, Y)$
- Now we can work with any number of weights, and thus *multivariate* data with m-dimensions!
- Note that this approach assumes Q is a sum of differentiable terms, but that's not particularly strict. In fact many flavours of *objective function* exist.

Multivariate linear regression: housing

- How much is a house worth, based on features like age, size, number of rooms, distance to work and schools, etc?
- Useful in real-life!
- We can use Boston housing data from `sklearn.datasets`
- From 1978...

Boston housing data

13 numerical features (+ our 'y value', price), so we can use multivariate linear regression. We will:

- Scale the data
- Perform K-fold cross-validation
- Look at adding a penalty term to the cost function

Cross-validation

Rather than doing a simple train/test split, it is sometimes useful to do k-fold cross validation.

Split all our data into k subsets (also called *folds* or *splits*):

[s0, s1, s2, s3, s4]

Across k trials, pick out one subset as a test set. Combine the rest as training data:

- $Train_0 = \{s1 + s2 + s3 + s4\}$, $Test_0 = s0$
- $Train_1 = \{s2 + s3 + s4 + s0\}$, $Test_1 = s1$
- $Train_2 = \{s3 + s4 + s0 + s1\}$, $Test_2 = s2$
- ...etc...

Average the results from each trial to get an idea of how well the estimator predicts values within the dataset.

Regularisation (Penalty term)

- Sometimes called R for *regularisation*, the penalty term is an extra term we add to our cost function.

$$Q_{regularised}(\vec{w};\ X, Y) = Q(\vec{w};\ X, Y) + \lambda R(\vec{w})$$

- As before, we are trying to minimise the cost function. The R term gives us a subtle new power: we can use it as a sort of 'second opinion' on the quality of $\vec{w}$, beyond just reducing Q.
- This is useful because in general we want to reduce *overfitting*.
- $R_{L2} = \sum_{j=1}^{m} {w_j}^2$
- $R_{L1} = \sum_{j=1}^{m} |w_j|$
- (Note we start j from 1 not 0, because we don't normally regulate the 'intercept')
- As we crank λ higher, we enforce smaller values of w_j