Models

Alex Sverdlov
alex@theparticle.com

1 Introduction

Data Science is all about building models. Without models we don't have a *science*, we just have *data*. Actually, without models we're basically doing statistics. But what are these models we speak of? A model is a description of something—often much simpler description than the thing it is trying to describe.

This "*model is a description*" business implies that there's a language to describe things. It also implies an implementation: we must be able to check whether something satisfies the description.

One way to describe and verify things is to write them down. For example, a character string "2" describes a number 2. If we look at character string "3" we know it is not a number 2. Looking at another character string "2" we can verify that it is a number 2.

What string would we write for square root of 2? How about "square root of 2" or shorthand: $\sqrt{2}$. Notice how we changed the language: we cannot write down the digits for $\sqrt{2}$ because it is irrational, so we invent notation in which we can write it down. Similarly, we can also write down π as "ratio of diameter to circumference of a circle".

Unfortunately the trick of inventing notation doesn't get us very far. Most things are beyond our representation—similar to how we cannot write down irrational numbers. Even if we use English (or anything else) to describe them.

The models we *can* build must be: finite (in description size) and computable (running time is finite). Notice how that limits the things we can describe. Everything outside of that is forever beyond our capabilities to even talk about (well, except the way we're doing it now; e.g. we can write down 'infinity' as a word, or a symbol ∞).

Just look at any math book. All problems are finite. All problems are solvable in finite time. The building blocks are vertices, vectors, equations of lines, circles, polynomials, etc., functions such as sines, cosines, square roots, etc. There are no arbitrary curves or shapes—only those that can be described in finite number of characters on a page.

Computer science too has its own set of description based limitations: every function is a mapping from input to output. Yet there are infinitely more input-to-output mappings than there are programs. In other words, there are functions that cannot be described/computed by a program. So when we're programming, such functions represent things that we just can't write a program for.

To bring the discussion back to models: Our model description language will be made out of simple building blocks, such as vertices, vectors, linear equations, geometric objects (circles, rectangles, etc.), logic statements, etc. Models will combine these in several different ways, but always keeping the full model description finite and computable.

Another requirement of a model: it is desirable to have a model that we can learn or adjust from data. This is the reason why many models are trivially simple: it's easy to adjust them using data.

2 Location

One of the simpler models is to just record the midpoint of the data. Mean, or average, provides a typical (average) example of the data. We compute it via:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

We often use μ to refer to population mean, and $\bar{x}$ to sample mean. They are both computed in the same way.

The Law of Large Numbers tells us that sample mean approaches population mean. That means that the difference $\bar{x} - \mu$ approaches zero as we sample more values.

Mean is considered a measure of 'location' of the data: mean is often used as the center of the dataset.

Another measure of location is percentile; it is a value which bounds a certain percentage of observations. For example, for 95th percentile, 95% of values are below it and 5% are above. One straight forward way to calculate it is to sort the dataset, pick a value right below and right above the desired percentile, and then do a weighted average.

Median is just 50th percentile; it is the middle value (or weighted average of the two middle values). It is often used as a more stable version of the 'center of the dataset'.

Dispersion

While the average/median tells us *where* the data is (where the center is), it doesn't tell us how spread out it is. We get that from variance:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$$

or sample variance:

$$s^2 = \frac{1}{(n-1)}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

Notice the difference between σ^2 and s^2. The basic idea is that since we are not dealing with the whole population, we know sample variance has to be higher than population variance; the extra factor adjusts for that uncertainty.

The standard deviation is just a square root of variance:

$$\sigma = \sqrt{\sigma^2}$$

Replace σ with s to get sample standard deviation.

Just as with mean vs median, there is also interquartile range: is it the range of the middle 50% of the values; in other words, 75th percentile minus 25th percentile.

There's also coefficient of variation, which is a percentage measure:

$$cv = \frac{s}{\bar{x}} \times 100\%$$

Standard error is:

$$se = \frac{s}{\sqrt{n}}$$

A normal distribution with mean 0 and standard deviation of 1 is known as: Z distribution. It is used to normalize the dataset, and allow for direct comparisons of different magnitude data:

$$z = \frac{x - \bar{x}}{s}$$

Practically, we often do not want to calculate $\bar{x}$ before calculating variance. We can fix that with a bit of algebra:

$$s^2 = \frac{1}{(n-1)} \sum_{i=1}^{n} (x_i - \bar{x})^2 = \frac{\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2/n}{(n-1)}$$

The above formulation means that in order to calculate mean and standard deviation (or z-scores), we need to keep track of only: N, $\sum x$, and $\sum x^2$.

Another thing worth mentioning is *absolute deviation*: instead of summing over squares of differences, we can sum over absolute values of differences.

Covariance & Correlation

Variance can be extended to more than one variable via covariance:

$$cov(x, y) = (x - \bar{x})(y - \bar{y})$$

Covariance matrix is symmetric. Off-diagonal elements determine the amount of covariance between variables.

If we normalize covariance (divide it by standard deviations of x and y) we get correlation:

$$corr(x, y) = cov(x, y)/sd(x)sd(y)$$

To compute correlation in a single pass, we can write it as:

$$corr(x, y) = (E(x * y) - E(x) * E(y))/(\sqrt{E(x^2) - E(x)^2} * \sqrt{E(y^2) - E(y)^2})$$

where $E(x)$ is the expected value of x. Correlation matrix is symmetric, with every elements being in the -1 to 1 range.

Covariance is often used as a quick way to fit 2D lines to points. For example, a slope of a 2D line is:

$$slope(x, y) = cov(x, y)/var(y)$$

The y-intercept is:

$$intercept = avg(x) - slope(x, y) * avg(y)$$

Central Limit Theorem

The sample distribution of sample means will approach the normal distribution. Note that the theorem does not say which distributions we start out with—they will all approach the *normal distribution*.

Simple Novelty Detection

With a few assumptions (many of which are often ignored in practice), z-scores (e.g. standard deviation) can be used to detect unusual samples. For example, if a z-score is above 2, then that sample is "unusual", in a sense that (if our assumption of normal distribution pans out) it is unlike 95% of the other samples. If z-score is above 3, then it is about 99%. With such measurements, we can convince ourselves that the stock market crash of 2008 occurs once every few million years.

This kind of novelty detection can be applied online: maintain a window of count, sum, and sum of squares of (perhaps 20 minutes, or 20 days, etc. Then using these you can calculate average, and standard deviation (or z-scores), and then quickly apply the novelty check towards the previously unseen sample.

3 k-Nearest Neighbors

We can record the entire tranining set, and assume that there's continuity between those recordings. The k-NN method is often described as "model free", but it's anything but. The records that are saved define the space between records that we can sample. To classify a previously unseen instance, k-NN searches the database of saved instances to find k nearest ones (by some measure of distance). A majority vote by the k nearest instances determines output class.

The "model" here is the saved instances—and the assumption that locality matters (things that are close to a certain label probably also have the same label).

The k-NN is a model that records list of instances, with an assumption to interpolate between instances upon retrieval.

4 Linear Models

Another way to model things is to use linear equations. The basic idea is that things can be summarized using lines (or hyperplanes). Similarly, different classes may be split using lines or hyperplanes.

For both of these, we need to be able to solve linear equations. An example may be: find a line that passes through points $(2, 13)$ and $(3, 17)$. We need to solve for a and b in the below equations:

$$2a + b = 13$$

$$3a + b = 17$$

Such things are easier to write in matrix form:

$$\begin{bmatrix} 1 & 2 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} b \\ a \end{bmatrix} = \begin{bmatrix} 13 \\ 17 \end{bmatrix}$$

What we have is a classic equation: $\boldsymbol{X}\boldsymbol{w} = \boldsymbol{y}$, where we need to solve for $\boldsymbol{w}$. Rearranging things a bit, we end up with two solutions for $\boldsymbol{w}$[1]:

$$\boldsymbol{w} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

$$\boldsymbol{w} = \boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{X}^T)^{-1}\boldsymbol{y}$$

With these, we learn that $\boldsymbol{w}$ is:

$$\begin{bmatrix} 5 \\ 4 \end{bmatrix}$$

In other words, the line is: $4x + 5 = y$.

We can use this method to solve any such linear system! Often, the matrices $\boldsymbol{X}^T\boldsymbol{X}$ or $\boldsymbol{X}\boldsymbol{X}^T$ will not be invertible—so we modify the solutions to always create invertible matrices:

$$\boldsymbol{w} = (\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

$$\boldsymbol{w} = \boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{X}^T + \lambda\boldsymbol{I})^{-1}\boldsymbol{y}$$

Here, $\boldsymbol{I}$ is an appropriately sized identity matrix, with λ being a small constant, such as 0.001.

[1] The least squares loss function is derived from an assumption that the sample data set S is generated by a smooth function with Gaussian noise. The probability of the sample data S, is a product of probabilities of individual points, which is proportional to

$$P \propto \prod_{i=1}^{L} e^{-\frac{1}{2}\left(\frac{y_i - f_{w_1,\ldots,w_M}(\boldsymbol{x}_i)}{\sigma}\right)^2}$$

where σ is the sample data standard deviation, which we assume to be constant. Maximizing this probability is equivalent to minimizing the negative of its logarithm, which is equivalent to the sum of squares loss function.

Why are there two solutions, and which one would we use? That depends on the shape of our problem: Let us do a slightly more complicated example: find a line that passes through points $(2, 13)$, $(3, 17)$, $(5, 23)$, $(7, 29)$, $(11, 31)$, $(13, 37)$. We need to solve for a and b in the below equations:

$$2a + b = 13$$
$$3a + b = 17$$
$$5a + b = 23$$
$$7a + b = 29$$
$$11a + b = 31$$
$$13a + b = 37$$

Rewriting in matrix form:

$$\begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 1 & 5 \\ 1 & 7 \\ 1 & 11 \\ 1 & 13 \end{bmatrix} \begin{bmatrix} b \\ a \end{bmatrix} = \begin{bmatrix} 13 \\ 17 \\ 23 \\ 29 \\ 31 \\ 37 \end{bmatrix}$$

Plugging in solutions for $\boldsymbol{w}$, we get:

$$\begin{bmatrix} 11.4437 \\ 1.9836 \end{bmatrix}$$

In other words, the line is approximately: $1.9836x + 11.4437 = y$. Notice that this line does not fit any of the points perfectly, yet it approximately fits all of them! So why the two solutions? The $\boldsymbol{X}^T\boldsymbol{X}$, or primal solution, needs to invert a 2×2 matrix, while the $\boldsymbol{X}\boldsymbol{X}^T$, or dual solution, needs to invert a 6×6 matrix—in this case, doing the primal solution is much faster. For situations when we have few points in many dimensions, solving the dual solution is often faster. For example:

$$\begin{bmatrix} 1 & 2 & 3 & 5 & 7 \\ 1 & 3 & 5 & 7 & 11 \\ 1 & 5 & 7 & 11 & 13 \end{bmatrix} \begin{bmatrix} c \\ b \\ a \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}$$

Solving for $\boldsymbol{w}$ is much simpler via the dual method ($\boldsymbol{X}\boldsymbol{X}^T$) as it only requires inverting a 3×3 matrix, and not the 5×5 matrix required by the primal ($\boldsymbol{X}^T\boldsymbol{X}$) solution. The solution by either method is

$$\begin{bmatrix} 1.627295 \\ -0.075596 \\ -1.320899 \\ 1.476102 \\ -0.556916 \end{bmatrix}$$

Notice that the above isn't a line; but a hyperplane!

Non-Linear Embedding

Now for a bit of magic: this linear method can fit non-linear functions, via the process of embedding. For example, if we want to fit $y = Be^{Ax}$, we can take log of both sides to get $\ln(y) = Ax + \ln(B)$, which is linear. Now we simply use that form in $\boldsymbol{X}$ and $\boldsymbol{y}$ and what we're fitting will be the non-linear $y = Be^{Ax}$.

Similarly, to fit power function $y = B * x^a$ we can take log of both sides to get $\ln(y) = \ln(B) + a * \ln(x)$, which is now linear.

This embedding is very powerful. The idea is to embed non-linear data into some higher dimensional space that perhaps has linear structures, and then use a linear solver.

To fit polynomials we "embed" higher dimensions that are powers of x. For example, instead of

$$\begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 1 & 5 \\ 1 & 7 \\ 1 & 11 \\ 1 & 13 \end{bmatrix}$$

from example above, which would fit a line, you can fit a 3rd degree polynomial just by tweaking that matrix to be:

$$\begin{bmatrix} 1 & 2 & 2^2 & 2^3 \\ 1 & 3 & 3^2 & 3^3 \\ 1 & 5 & 5^2 & 5^3 \\ 1 & 7 & 7^2 & 7^3 \\ 1 & 11 & 11^2 & 11^3 \\ 1 & 13 & 13^2 & 13^3 \end{bmatrix}$$

The resulting solutions will have the form $y = D + Cx + Bx^2 + Ax^3$. This can be extended to any degree polynomial we care to fit.

Kernel Trick

The embedding procedure can be avoided by noticing $\boldsymbol{X}\boldsymbol{X}^T$ in the primal solution for w. The $\boldsymbol{X}\boldsymbol{X}^T$ can represent the *kernel* function, such as inner join of ith row with jth row, or such inner join squared, etc.

The kernel 'trick' is that treating $\boldsymbol{X}\boldsymbol{X}^T$ as a kernel often allows for very complicated non-linear embeddings—even into infinite dimensions—without us ever actually calculating the embedding itself.

Interpolation & Extrapolation

Once we get the 'line' (or polynomial, or hyperplane, etc.), what can we do with it? Well, we can fill in missing values—for sample, lets say we have values from 1 to 100, but we have

some gaps in the middle of the dataset. We can fill those in simply by plugging the values into the 'line' (or whatever we've fit). That is called interpolation.

The other thing we can do is project our queries outside the sample used to fit the line. For example, we've fit an exponential curve to earnings data for the last 2 years, and we would like to guess what the earnings will be next quarter. This is called extrapolation—and is often much less precise than interpolation.

Least Squares Discriminator

While the 'least squares' method described above is used primarily for interpolation and extrapolation, a similar technique can be used for classification. Given a training set:

$$S = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_L, y_L)\}$$

where $y_i \in \{-1, +1\}$ indicates the class. Our model is a hyperplane, with weights $\boldsymbol{w}$, and distance D, such that:

$$w_1 x_1 + \cdots + w_N x_N = D$$

With such a hyperplane, we get a notion of things being in 'front' of the plane and in the 'back' of the plane. If we plug $\boldsymbol{x}$ into the plane equation (represented by $\boldsymbol{w}$ and D), and get a positive value, then $\boldsymbol{x}$ is in front of the plane, etc.

To turn this problem into the 'least squares' problem described above consider the dual solution. We only used the inner products to find the interpolating line. Now we need to incorporate the y_i values into that Gram matrix. What we end up with is known as a Hessian matrix:

$$H_{ij} = y_i y_j \boldsymbol{x}_i \boldsymbol{x}_j$$

Notice that the H matrix is essentially the kernel multiplied by $y_i y_j$, e.g.

$$H_{ij} = y_i y_j \boldsymbol{K}(\boldsymbol{x}_i, \boldsymbol{x}_j)$$

This allows for non-linear classifiers.

We can obtain a KKT (Karush-Kuhn-Tucker) system:

$$\left[\begin{array}{c|c} 0 & \boldsymbol{y}^T \\ \hline \boldsymbol{y} & H \end{array}\right] \left[\begin{array}{c} -D \\ \hline \boldsymbol{\alpha} \end{array}\right] = \left[\begin{array}{c} 0 \\ \hline \mathbf{1} \end{array}\right]$$

where $\boldsymbol{y} = (y_1, \ldots, y_L)$, $\mathbf{1} = (1_1, \ldots, 1_L)$, H is the Hessian matrix, and $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_L)$ are Lagrange multipliers. We can then solve for $\boldsymbol{w}$ via:

$$\boldsymbol{w} = \boldsymbol{X}^T [\boldsymbol{\alpha} \times \boldsymbol{y}]$$

where $\boldsymbol{\alpha} \times \boldsymbol{y}$ is an element-wise multiplication.

Maximum Margin Classifier

In general, we want as much separation between classes as possible—we want the classifier to find the maximum buffer. More on this in class.

Support Vector Machines

SVMs are not really machines. They are maximum margin classifiers, with other nice features.

One of the major problems with the least squares method is that it becomes impractical with a relatively low number of samples. For N input samples, we would need an N by N Gram matrix, with most matrix multiplications taking $O(N^3)$ operations—consider a modest problem with 10000 samples to get an idea of how quickly this becomes impractical.

This is where *Support Vector Machines* come in. SVMs are binary classifiers, identical to least squares discriminator in every way, except they don't use all the input samples for training. The important points, as far as classification is concerned, are the ones on the boundaries. If we use just the boundary points, the classifier will be just as good as if we used all the points. The big question now is how to find the boundary points.

Most SVM algorithms have a notion of 'working set'; where, in every iteration, the algorithm picks a 'working set' of input points to use for training. The working set is generally relatively small. Some techniques pick points for the working set which have the maximum influence on the resulting classifier (essentially picking inputs with the corresponding largest Lagrange multipliers). Google for SMO algorithm.

SVMs are often used in conjunction with kernels.

5 Hyperplanes

Linear models capture a lot of the model building blocks: anything that can be formed by combining a few hyperplanes can be described by linear models.

For example, a decision tree partitions the concept space in axis aligned hyperplanes. A neural network is essentially many layers of hyperplanes. etc.

6 Mean Classifier

Another way to model things is by recording the center point, and some sort of threshold. Which leads to a mean classifier. The mean classifier is perhaps the simplest: we find the mean of all positive (label +1), and negative (label -1) samples. We can then compare new samples to the two means, and pick closest one.

7 Conclusion

So in the end, we're building models that are essentially points, histograms, hyperplanes, hyper-spheres, or table-lookups. We can combine these in all sorts of ways, but in the end, these are the building blocks out of which nearly all models we work with get built.