COS 424 Lecture 2:
Connecting the dots with common sense and linear models

Scribe: Michael Gelbart

Lecture date: February 4, 2010

1 Introduction

We can begin without thinking about probabilities, but just by asking the simple question: how can we predict $y = f(x)$ given some data x? We are used to just drawing a curve through the data and then using the curve to predict new outcomes (see image on Slide 4), but what if you have high-dimensional data? We cannot easily visualize such a "curve". Can we learn something from the 1-d case and extend it to higher dimensions?

2 A simple linear model

Yes, we can try to do this with a simple polynomial model

$$f(x) = w_0 + w_1 x + w_2 x^2 + ... + w_n x^n$$

where we use the data to determine the coefficients w_j and the model depends linearly on these coefficients. We can generalize this to using more complicated basis functions than these polynomials. In general we might have

$$f(x) = w_0\phi_0(x) + w_1\phi_1(x) + w_2\phi_2(x) + ... + w_n\phi_n(x)$$

or, written more compactly:

$$f(x) = w^{\mathrm{T}}\Phi(x)$$

where w and Φ are now vectors as defined on Slide 6.

Given some model $\Phi(x)$, we still need a method to determine the vector of coefficients w. To do this, we compare the real values of the data with the output from the model for some inputs x. A common method is to find the least squares minimum: i.e., the vector w that minimizes the sum of squared distances between the training data points and the predicted points from the model. The objective then takes the form

$$C(w) = \sum_{i=1}^{n} (y_i - w^{\mathrm{T}}\Phi(x_i))^2$$

Because this quadratic objective function is convex, there is a unique minimum value of w. By taking the derivative with respect to w, we see that the optimum value of w satisfies

$$(X^{\mathrm{T}}X)w = X^{\mathrm{T}}Y$$

where X and Y are defined as follows:

$$X \equiv \begin{bmatrix} \phi_1(x_1) & \phi_2(x_1) & \cdots & \phi_n(x_1) \\ \phi_1(x_2) & \phi_2(x_2) & \cdots & \phi_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(x_n) & \phi_2(x_n) & \cdots & \phi_n(x_n) \end{bmatrix}, \quad Y \equiv \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

It seems that we can solve for w by computing $w = (X^{\mathrm{T}}X)^{-1}X^{\mathrm{T}}Y$, but computing the inverse is a *bad idea* because $X^{\mathrm{T}}X$ is not necessarily invertible. For example, if we pick two basis vectors to be same (say, $\phi_1 = \phi_8$) then we have a redundancy and the matrix is singular (because $(X^{\mathrm{T}}X)^{-1} = X^{-1}(X^{\mathrm{T}})^{-1}$). We want to be robust to these kinds of redundancies, since with this basis we should still be able to get a (non-unique) solution that does the job. Then we break the ties by picking the w with the smallest norm.

Slide 10 shows several numerical procedures for solving w without inverting $X^{\mathrm{T}}X$: diagonalization, and SVD and QR decompositions. Notation: on the slide, D^+ refers to the pseudo-inverse of the diagonal matrix D, meaning that all non-zero entries of D are inverted and all zero entries remain as zero. Also, Q, U, and V refer to orthogonal matrices and R refers to a triangular matrix. Methods like SVD are nice because you don't have to deal with the potentially messy matrix $X^{\mathrm{T}}X$. The final method introduces an artificial "tie-breaking" term ϵw^2 into the objective function. For sufficiently small ϵ, this term ensures that the w with the smallest norm is chosen in redundant cases like the one discussed above. In other words, this method assures that our matrix in question, now $X^TX + \epsilon I$ is no longer singular, so we can solve the system without further precaution, with any available method. The error introduced by this method is not a problem for $\epsilon \sim 10^{-5}$ in reasonable units.

3 Bigger not always better

Slides 11-17 show polynomial fits of different orders to the data. The cubic fit looks the most "reasonable" because the high order polynomials are overfitting the data. One problem with this is that due to being higher order they are extremely steep outside the domain of the training data points such that they may introduce extremely large error for test points outside this domain. E.g., for a degree 20 polynomial, $2^{20} \sim 10^6$ so even on the domain $-2 \leq x \leq 2$ we are adding and subtracting really big numbers (analogy to a "highly loaded spring"). Thus if we have a new point at $x = 5$, the high order fits would predict some enormous number for y (keeping in mind that $5^{20} \sim 10^{14}$), and this behavior goes against our intuition (and is indeed incorrect in this case). Thus there may be better bases to use that do not become so steep, such as the Chebyshev or Hermite polynomials.

4 Evaluating the models

We have discussed the pros and cons of some models, but we should come up with some metric of evaluating them quantitatively. One metric is the mean squared error (MSE) of the test model, computed by

$$\mathrm{MSE}_{\mathrm{training}} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{f}(x_i)^2$$

where $\hat{f}(x) = w^{\mathrm{T}}\Phi(x)$, the predicted value from the model. The true MSE just becomes an integral

$$\mathrm{MSE}_{\mathrm{true}} = \frac{1}{x_{max} - x_{min}}\int_{x_{min}}^{x_{max}} \sigma^2_{\mathrm{true}} + (f_{\mathrm{true}}(x) - \hat{f}(x))^2 dx$$

Here the noise is added in because we want to produce a number that is compatible with the error we would get if we computed the MSE of our model on a large testing set. For example, if our model

is exactly correct, we would still not get an MSE of zero on a test set because of the noise. Thus we include the extra term here as well.

From this metric it appears that higher order polynomials do increasingly well. However, since we know the true function in this case, we can compute the true MSE and we see that it increases dramatically with higher order. In this case the minimum of the true MSE occurs for degree 3 (see the graph on Slide 19).

However, the MSE is not necessarily the best choice here. For example, in this case we are weighting the domain uniformly when taking the integral. However, if the data points do not come uniformly then we would like to modify this by weighting the error metric with the distribution of x. For training, we are implicitly giving higher weight to those regions with more points and so a non-uniform distribution is ok. However, we might have the distributions for the training and test sets not equal, and then we have a problem. Example in class: handwriting recognition. We my collect data from some writers and then want it to work for others. Some benchmark data sets address this problem by mixing data from many writers in both the training and test sets.

Some more error metrics are described on Slide 20. The uniform metric L_∞ finds the biggest difference between the two curves over the whole domain. We can also use L_p norms other than L_2; i.e., integrate the function $(f - \hat{f})^p$ for some values of p other than 2. Larger p penalizes bigger deviations (so the notation L_∞ makes some sense now).

5 Piecewise linear models

If we don't like that polynomials get too steep, we can try piecewise linear models. Such a model is described by k knots separating $k+1$ lines. Each line is described by two parameters for a total of $2(k+1) = 2k+2$ parameters. However, we have the continuity constraint which gives us one equation at each knot, for a total of k constraints. Thus we are left with $k+2$ parameters for k knots. Note that the knots are not parameters here– they are chosen *a priori* and the parameters come from specifying the line segments.

Slides 22-23 show that like polynomials, things get worse when we give the model too much freedom (in this case, by making k too large). However, things aren't as bad as for the polynomials (see slide 24). This time the MSE is minimized for $k = 2$. Slide 26 shows a restatement of the basis in terms of ramps or triangles, but in the end they are all piecewise linear functions.

6 Splines

Of course there are lots of other bases we could choose, but one more common one is the piecewise polynomial or "spline". The quadratic spline basis is $\{1, x, x^2, \cdots, \max(0, x - r_k)^2\}$. See slide 27 for a drawing. For k knots we get $k+1$ parabolas, each with 3 parameters, for a total of $3k+3$. Then the knots introduce $2k$ constraints: k for continuity and k for the continuity of the derivative. Thus we have a total of $k+3$ dimensions. As always, overfitting is bad (see slide 28), but not nearly as bad as with the other bases: see slide 29, and in particular the scale of the y-axis of the graph. The reason the spline does well is that it is smooth (we imposed continuity of the derivative) and the true function is also smooth. Finally, note that the red curve begins to dip for large k: this is just noise, do not trust it.

7 Changing the data

Of course one way to do better is just add more examples. Slide 30 shows the improvement when using 300 examples instead of 30. Slide 31 shows the improvement when the noise σ is reduced from 0.5 to 0.1. But of course if we could just play with the noise, we wouldn't need to do anything in the first place... we could just set $\sigma = 0$ and be done.

In short, there is always a cost to improving the model, and the cost is either more data or better data. The best solution is to take the best data you can on the first try, and then take a model with complexity well matched to the data.

8 Estimating the quality of our work

In what we were just doing, we already knew the true function, and that is of course an unrealistic scenario when generating models and evaluating performance. Furthermore, we do not want to evaluate the performance of the model using the training data, because this just tests its ability to reproduce something it already learned from, not its ability to make valid predictions on new data (analogy with "learning by heart" - we can't analyze new problems this way). Thus we want to test our model on fresh data, but how do we get it? Answer: we can separate the data set at the beginning, e.g., 2/3 for training, 1/3 for testing. But there are other methods than this. For example, we can make measurements using a totally different phenomenon. Example in class: we have a protein soup, and we put it in a machine that does mass spectrometry. We want to find what's in the soup. Since we need delicate accuracy, mass spec is probably the most sensitive, but we don't have much test data. So we turn to other methods of checking what proteins are inside the soup: we can use fluorescent / radioactive markers, or look it up in the literature, or do PCR, etc.. So we can validate the model we built from the mass spec data with a different style of measurement. (But for the purposes of this courses, we will just be using data that we set aside).

And now, the all-important fact: test sets should only be used once! Example in class: results based on the UCI data base improve every year... is this because algorithms are really getting better, or because the test set is compromised? We can get fancier by also using a validation set. First, train a bunch of models on the training data. Then pick the best of these models using the validation set. Then re-train that model using the training and validation data together. Finally, test on the test set.

This all sounds great but it uses up a lot of examples, and these can be expensive. E.g., medical tests where one data point is one person or in some geology studies where drilling to get one data point can cost millions.

There are also some issues of the size of the validation set, if we are going to use one. Splitting $\frac{1}{3}$-$\frac{1}{3}$-$\frac{1}{3}$ is common and might be reasonable. But in general we want to be barely able to discern which model is best at the validation phase, so if we can tell easily then we have too much validation data, and if we cannot tell then we have too little. See slide 37 for a more complicated scheme called k-fold cross validation. In this scheme you take all data except one block T_i and train. Then test on T_i and do this for all i. Then take the average over the k cases and pick the best model. Finally, retrain using the whole training set. This is works pretty well but is computationally expensive. Thus it should be avoided for large data sets, but that is ok because we don't need such a complicated scheme when data is abundant anyway.

9 Beyond curve fitting

What if our data is not a bunch of numbers, but rather categorical variables or even complex data structures? This seems like a mess, but really all we need to do is choose $\Phi(x)$ and then everything else is the same. Of course, this doesn't mean that choosing Φ is easy.

Slide 40 shows the "adult" data set for predicting whether or not a person makes >\$50k/yr. Our input vectors x are just lists of the relevant attributes, such as $\{age = 30$, $sex =$ male, *native country* = Canada, ... $\}$. First, we choose our feature basis $\phi(x)$ such that each one corresponds to an individual attribute. For example, one of them is 1 if profession = taxi driver, and 0 otherwise; another is 1 if profession = baker, 0 otherwise, another is 1 if sex = male, 0 otherwise; etc. We bin the continuous variables into 5 bins each so they they also become discrete; after doing so, we get a total of 123 different possible properties (5 from each of the continuous variables, and the rest from the categorical variables). The whole basis then has 1+123=124 elements (the first one being $\phi_1(x) = 1$). Note that

these $\phi_i(x)$ are not vectors; they are simply numbers taking on the values 0 or 1. $\Phi(x)$ is a vector of length 124, and the product $w^{\mathrm{T}}\Phi$ is a real number. We then take the sign of this number to get y.

Once we have these $\phi_i(x)$ we can solve for w with the usual methods. When we solve, the magnitude of each entry in w will tell us how good a predictor each property is. For example, if the entry corresponding to profession = investment banker is very large in magnitude, then the fact that someone is an investment banker tells you a lot about whether or not their income is above \$50k/yr. The sign of the entries of w tell us the sign of the correlation, i.e., if the investment banker entry is positive it means that being an investment banker makes you more likely to make over \$50k/yr and if it is negative it means it makes you less likely.

We mentioned above that $w^{\mathrm{T}}\Phi(x)$ outputs a real number, and so to find y we must threshold the result by taking the sign of $w^{\mathrm{T}}\Phi(x)$. Since $\Phi(x)$ contains only ones and zeros, negative elements in w will result in the product $w^{\mathrm{T}}\Phi(x)$ being more negative, and positive values of w will make it more positive. This is why I was able to say that the sign of the coefficients w told us if the correlation was positive or negative. Of course, we could have picked a threshold other than zero, such as $+3$. In this case the resulting w would be the same, except that its first entry (the one corresponding to $\phi_1(x) = 1$) would just be increased by 3 such that the product $w^{\mathrm{T}}\Phi(x)$ would also be increased by 3 for all x. Thus the choice of threshold is arbitrary because it will be encoded in the first entry of w (and only in the first entry!). Given this freedom, the choice of zero as the threshold is most natural because it gives a meaningful interpretation to the sign of the elements of w.

And now, back to the performance of this model. After validating and retraining with this basis we get a misclassification rate of 15.47%. To improve on this, we try adding more features. Now we include one feature for each unique pair so that we get an additional $123(123-1)/2 = 7503$ features. These features are the product of the individual features, which is like the logical AND. This basis is a little redundant because we will get lots of $\phi_i(x)$ that are zero for all x (e.g., male AND female = 0). This is ok though, because our numerical methods can handle these degenerate cases as discussed in slide 10 (we discussed the case of two identical basis vectors, but having a basis vector equal to zero is the same as having two identical basis vectors since you can just subtract the two identical ones to get the zero vector).

When using this larger basis the fitting takes too long to run because X is an $\sim$8000-dimensional matrix so $X^{\mathrm{T}}X$ has $\sim$16 million entries. To help with this, we can use sparse matrix representation, but we still have to "invert" at the end (solve system) and this takes $\sim$20 mins.

On slide 43 we see a trick to force the coefficients of quadratic terms to be closer to 0, driving it back to a linear system. On slide 44 we see that $\epsilon = 100$ is best because validation set has lowest error.

10 An aside on speeding up the code (not in slides)

We want to speed up the code, but even though X is sparse, X^TX is not sparse so we can't use linear algebra tricks. We want to find

$$C(w) = \min \frac{1}{n} \sum_{i=1}^{n} \frac{1}{2}(y_i - w^T\phi(x_i))^2$$

One approach is gradient descent: start from some w_0 and follow the gradient towards the minimum. Thus we follow the recursion

$$w_{t+1} = w_t + \frac{\gamma}{n} \sum_{i=1}^{n} (y_i - w^T\phi(x_i))\phi(x_i)$$

where the second part is the derivative of the objective function. Even though $\phi(x)$ is sparse, this is still going to be costly because of the big sum (28,000 examples in the UCI case). So now we do something drastic: at each iteration we pick just one example at random

$$w_{t+1} = w_t + \gamma_t(y_t - w^T\phi(x_t))\phi(x_t)$$

This one term is a poor approximation of the average, but if we pick $\gamma_t = \gamma_0(1 + \epsilon t)^{-1}$ then things improve because γ is decreasing with time. If you do this for many epochs then you can plot the training and validation error, and then pick the minimum of the validation error. This is certainly not the least squares solution, but it is a solution to the fitting problem and it works really fast. In one case, this method sped up the analysis from 6 hours to a few seconds. Of course, this is only worth doing if you have a large data set.