December 5, 2017

CS 361: Probability & Statistics

Regression

Vector quantization

Classification and vectors

In this chapter and the last we gave some algorithms for doing machine learning on vectors of features

Perhaps the biggest part of machine learning in practice is figuring out *which* features to use

Suppose I was developing a classifier to decide which users of Facebook to target with an ad. The input to this classifier are the nodes of a graph, not vectors, so somehow I need to be able to look at a node in this graph and generate a vector of features

You could probably think of 10 features of a Facebook user that might work well for this task, and the point is that doing this kind of thinking often has a surprisingly high ROI for the ultimate accuracy of your classifier

Patterns

Images, videos, sounds, time series data, and accelerometer signals can also be the input of classification algorithms. We will be looking at a method for getting good features out of a variety of signals like these. An additional wrinkle with this kind of data is that each data point might be of a different size than the others

The key to this approach is that it exploits repetition. And most signals—like images or human speech sounds—can be thought of as a composition of a relatively small vocabulary of patterns

If you look at many pictures of beaches, for example, and you go and look at 10x10 pixel regions of the images, you will find that most beach scenes really just consist of some patches of waves, sky, and sand. And these patches can be surprisingly similar

Patterns

Another example is human speech. When we learn to read or learn a new language we spend a large amount of time “sounding things out” — we learn the phonemes (things like k- and th-) of the language and learn how to combine them in order to produce fluent human speech

This small vocabulary of phonemes is all we need to pronounce any given utterance

We can re-construct a given signal then, like a picture of a beach, as long as we know which patterns we should be using and how they should be combined to form the image

Patterns

We will develop an approach to look at a set of data and extract these patterns which we can then use to express any given data item as a feature vector—in particular a histogram of the counts of how many times each pattern appears

For the examples we will work on, this is all we will keep. We won't even store the order that the patterns appear in, just how many times each pattern appeared in a given item. This is sometimes enough to do classification on things like images and accelerometer data

Vector quantization

The way we will do this is we will take the signal we are trying to classify and break it up into fixed-size sub signals. We might take an image and break it up into a bunch of 10x10 pixel sub images, for example

These sub signals could be overlapping or not. In the homework it probably isn't necessary for them to be overlapping

We do this for every image in our dataset, getting a large collection of subsamples

Then we just cluster this large collection of subsamples

The centers of these resulting clusters become our patterns. We give each cluster an integer index [1, …, k]

Vector quantization

Now for any data item, in order to get a feature representation for it. We break it up into sub signals and for each sub signal we see which of the cluster centers it is closest to

A given data item then will become a histogram. Each of its sub signals contributes to the count of one of the *k* patterns in the vocabulary, and we represent the item by this histogram of counts of the number of patterns in the item

If we had used hierarchical k-means to do our clustering, the way we get the right cluster center for a sub signal is first look at which top level cluster center it is closest to, and then looking only at the sub-clusters within that cluster center, determine which cluster center the sub signal is closest to

Example

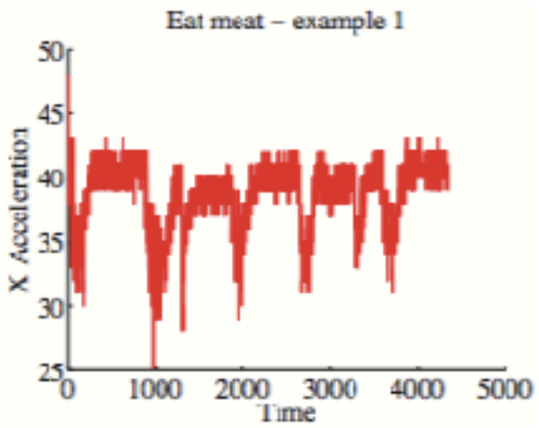
The textbook has a link to some accelerometer data, presumably from something like a FitBit

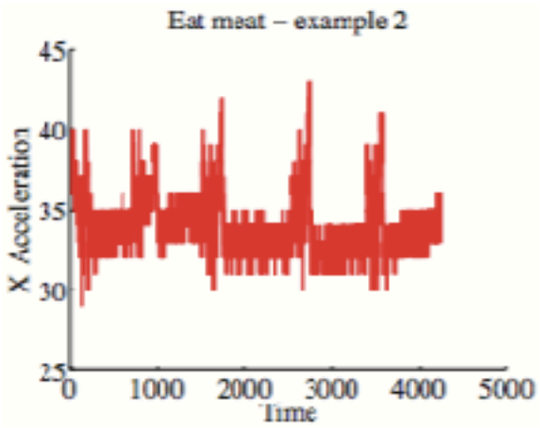
Each datafile corresponds to an instance of some activity like brushing your teeth and contains a long list of accelerations in the x, y, and z directions—32 measurements per second. The data files all have different lengths

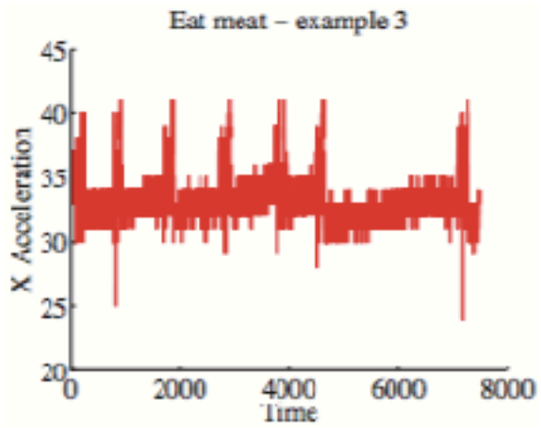
If we wanted to learn a classifier that could look at some accelerometer data of arbitrary length and output what activity is happening during the time period of interest, vector quantization is a good candidate for generating our features

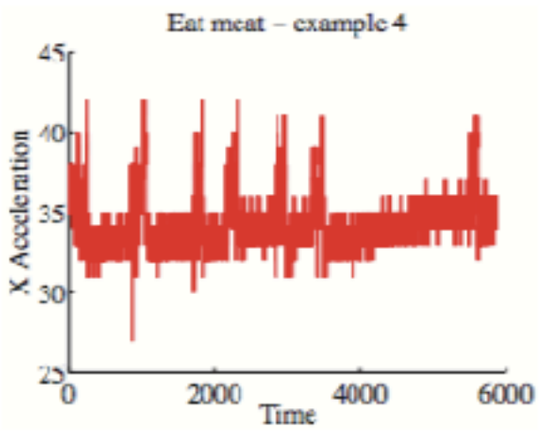
We wouldn't just input the raw data into a classifier because the data items aren't even of the same length

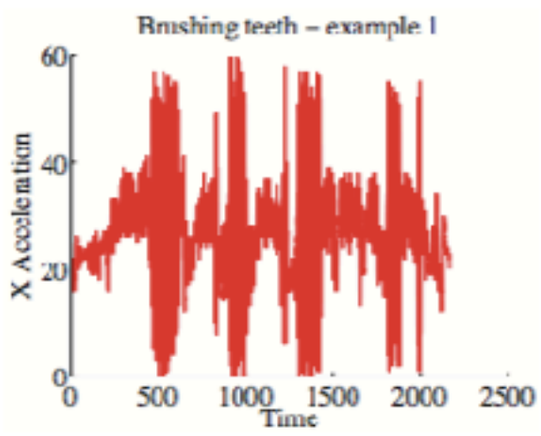
Example

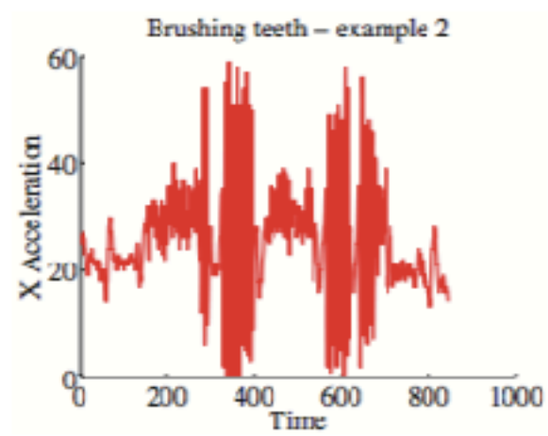

Eat meat – example 1
X Acceleration
Time

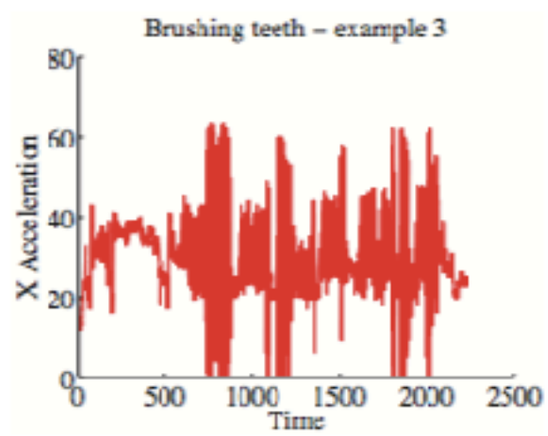

Eat meat – example 2
X Acceleration
Time

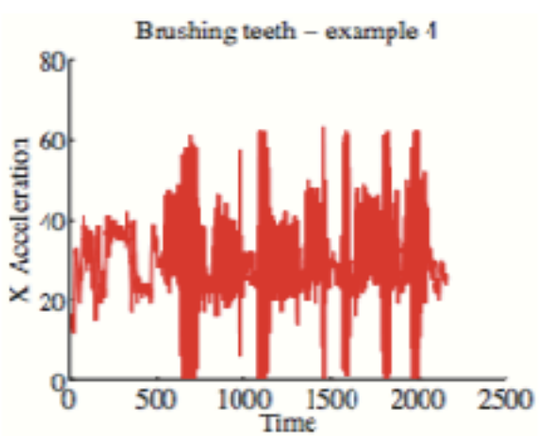

Eat meat – example 3
X Acceleration
Time


Eat meat – example 4
X Acceleration
Time


Brushing teeth – example 1
X Acceleration
Time


Brushing teeth – example 2
X Acceleration
Time


Brushing teeth – example 3
X Acceleration
Time

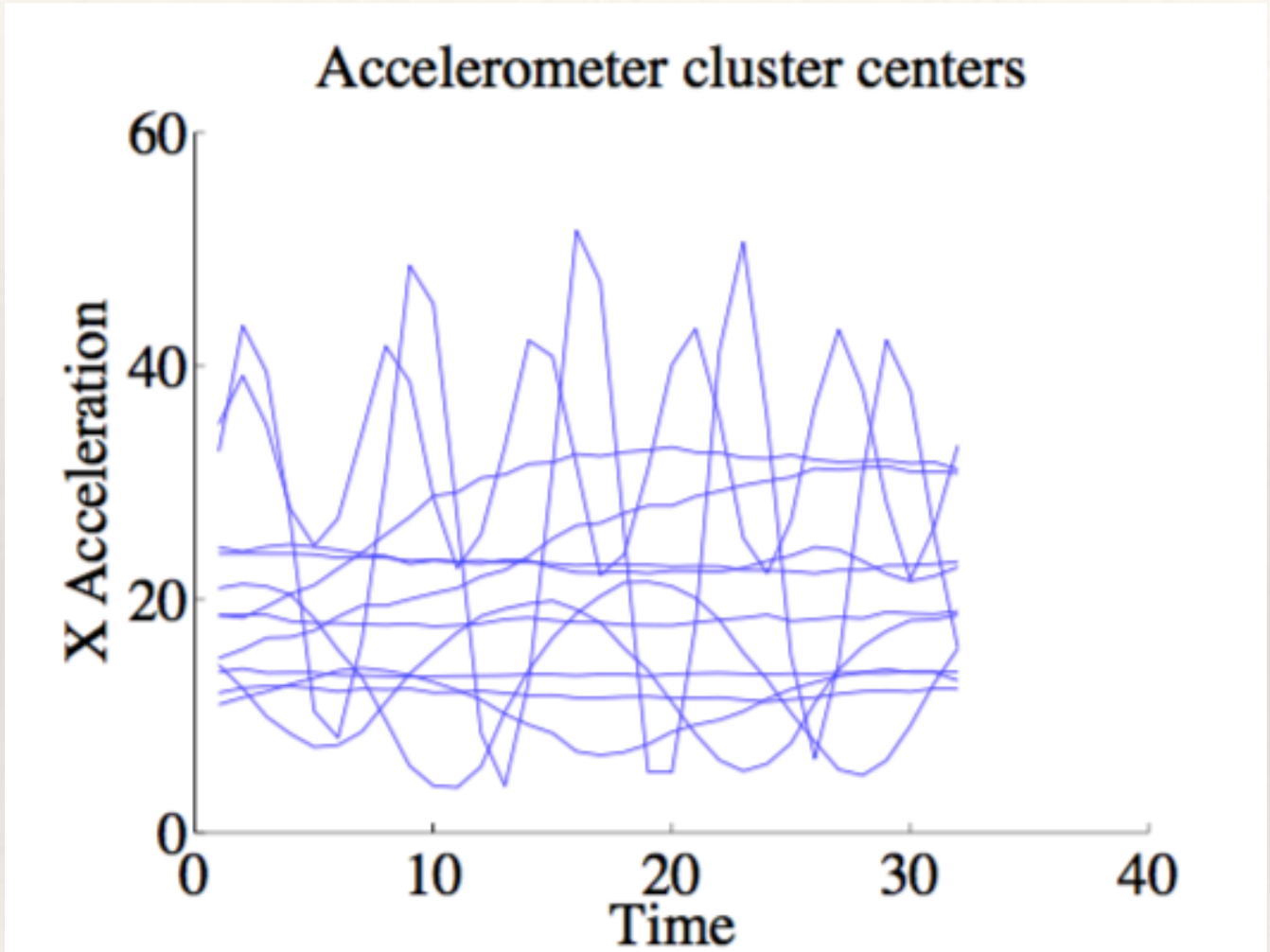

Brushing teeth – example 4
X Acceleration
Time

Example

Let's take each data file and chop it up into non-overlapping sub signals of 32 measurements each (i.e. 1 second intervals)

We can then cluster the set of all sub signals. We get a set of clusters. Each cluster center will then be a set of 32 measurements, which we can look at

Example

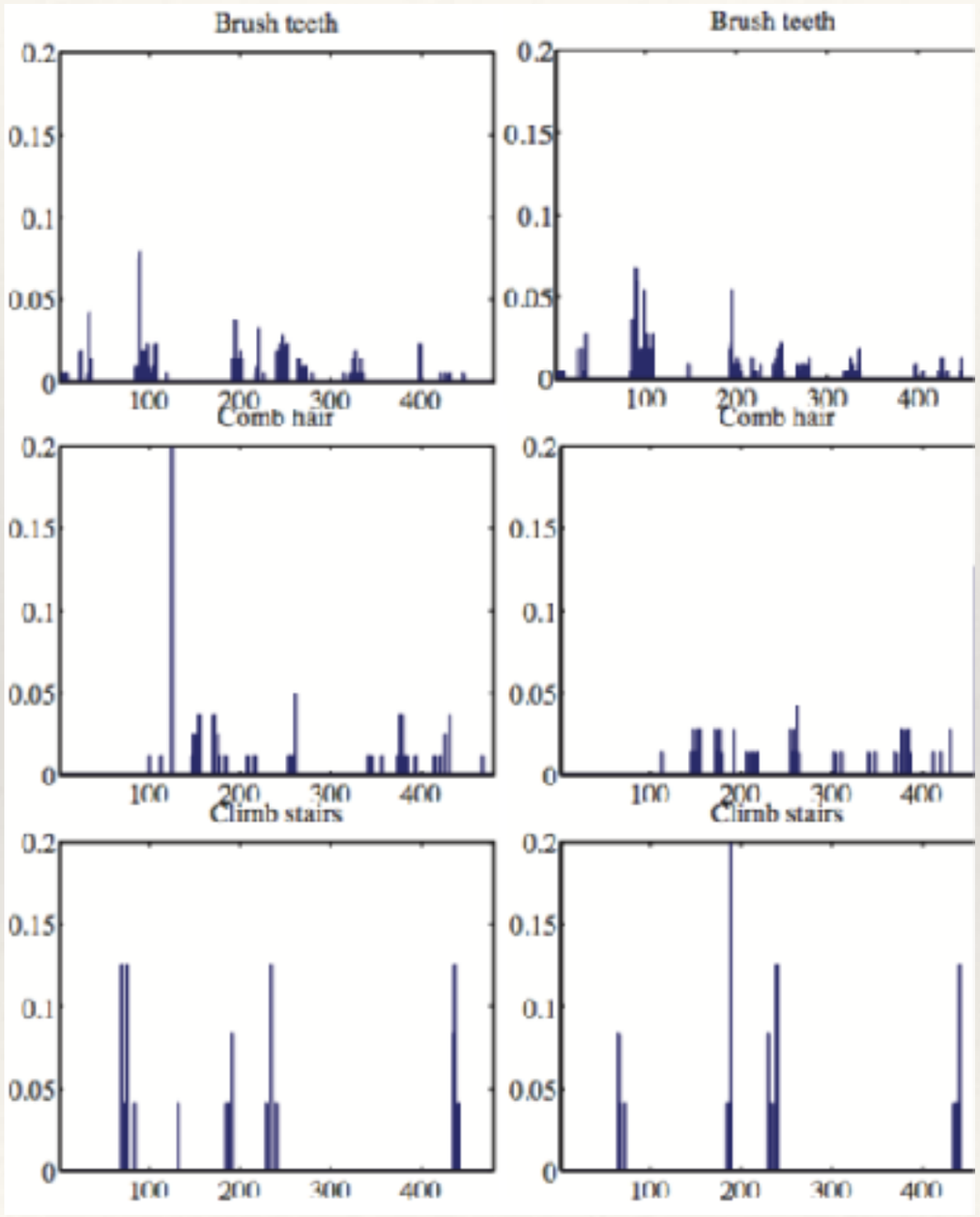


Some of the cluster centers

Example

Keeping these cluster centers around and giving each cluster an integer index, we can now take any given activity file and break it up into sub signals, align each sub signal to the closest cluster center, and compute a histogram for that activity

Example



Here we used hierarchical k-means, take a smaller sample of the data and clustering it into 40 clusters and then for each cluster, clustering the data in that cluster into 12 clusters, giving 480 total clusters/patterns

Our histograms have been normalized to give the relative frequency of each pattern rather than the raw count

Regression

Regression

Like with classification, regression proceeds from a set of labeled training data

We have N pairs of d-dimensional points plus their “label", except in this case the label is not a class label but a numerical value that the data item takes on at the corresponding point

For example, the data could be the square footage of a house and the “label” is the price that the house sold for recently

Our goal, then, would be to learn a mapping from d-dimensional points to these numerical values

Applications

Prediction is one application. We might have a whole bunch of historical labeled data about some quantity of interest and wish to make predictions about in the future on data items where the label won't be available at the time fo the prediction

Another application of regression is to visualize trends in data and to compare how well data items follow this trend

Example: trends

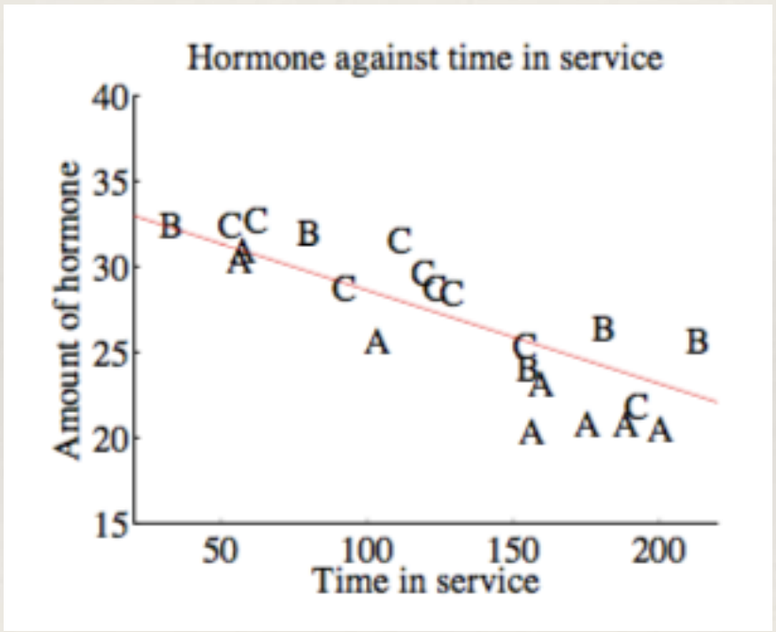
Suppose we have a dataset on a set of medical devices that remain in someone's body and release a hormone over time. The data are of the form (time_in_body, amount_of_hormone_in_device). Furthermore we have data from 3 different production lots of the device—A, B, and C—we are interested in doing quality control to see if there is any difference between the lots

We choose to model the amount of hormone remaining in the device as

$$a \text{ x (time in service)} + b$$

Example: trends

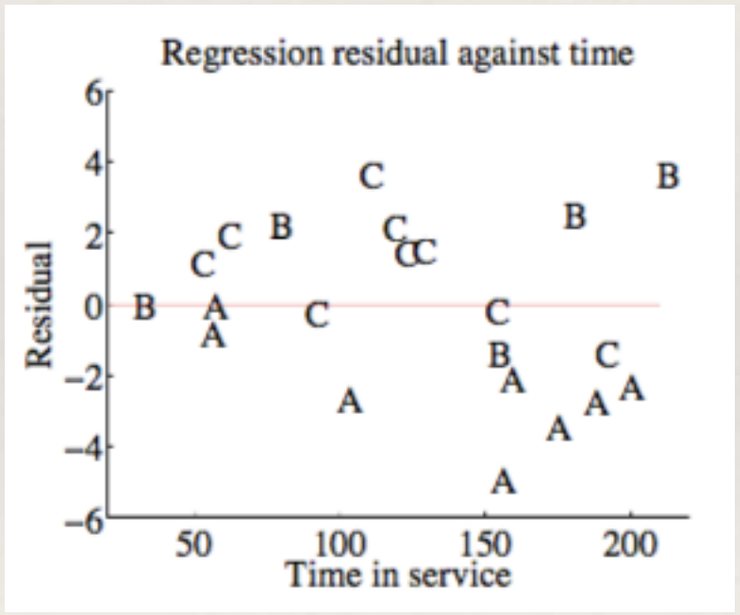
If we have some procedure for finding a good a and b we will have learned a line from the dataset which we can use to answer the question of differences between lots



It looks like the A's are consistently below the line in a way the other lots aren't

Example: trends

Since it can be hard to evaluate distances by eye, we can also make a plot by subtracting the amount of hormone that's predicted by the model from the amount that was measured. This difference is called **the residual**



And here the trend with the As being different is even more apparent

Regression

Assuming that we have *N* pairs (**x_i**, y_i), we think of the y_i as being the value of some function evaluated at **x_i**, with some random component added

We refer to the **x_i** as **explanatory variables** and the y_i as the **dependent variable**.

We want to use the *N* items we have—the training data—to build a model for the dependence between y and **x**. In order to predict values of y on data we have never seen before—test data.

Linear regression

A good simple model is to assume that the dependent variable is obtained by evaluating a linear function of the explanatory variables, then adding a zero-mean normal random variable

$$y = \mathbf{x}^T \beta + \xi$$

Where ξ is a zero-mean normal random variable of unknown variance

β is a random vector of weights which we must estimate or learn from the training data

Thus, in this model y should be thought of as a random variable and the prediction of a y from an $\mathbf{x}$ should be viewed probabilistically

Making predictions

When using this model to predict a value y for some new **x**, we cannot predict what value ξ will take, and so we will always predict that it takes the value of its mean, namely 0, or that for some new **x** our prediction is just

$$y = \mathbf{x}^T \beta$$

Looking at this prediction function, you might worry that our model can only produce lines going through the origin, but we can fix that since we have some control over how to model the explanatory variables, as the next example shows

Example

Suppose we are trying to model the dependent variable with just a single explanatory variable, i.e. our training data is a set of 1-dimensional x's and their corresponding y's

If, prior to training, we create for each data item a 2-dimensional vector given by $\mathbf{x} = \begin{bmatrix} u & 1 \end{bmatrix}^T$ where u was the original explanatory variable, then we learn the model, we will have learned a β_1, β_2 from the data and our prediction function will be

$$y = \beta_1 x + \beta_2$$

The line associated with this model doesn't necessarily go through the origin since the prediction for x = 0 is β_2

Training the model

So the question is how do we learn the weight vector beta? The answer is we will look at the problem probabilistically and come up with a good estimate

Let's look at our model

$$y = \mathbf{x}^T \beta + \xi$$

In training, the **x** and *y* are from the training data, Beta is what we are hoping to learn, and Xi is a zero-mean normal random variable of unknown variance

It makes sense to think of the value of *y* as being the outcome of a random process since there is a random variable on the right hand side, i.e. *y* is a random variable and we may be able to get a good value of Beta by considering the quantity $p(y \mid \mathbf{x})$

Training the model

$$y = \mathbf{x}^T\beta + \xi$$

For a given **x_i**, y_i, how can we express p(y_i | **x_i**)?

Well we have a zero-mean normal random variable to which we are adding a constant, so for training example *i* we can write p(y_i | **x_i**) as a normal random variable with mean $\mathbf{x_i}^T\beta$

Recall the form of the normal density $p(x) = \left(\frac{1}{\sqrt{2\pi}\sigma}\right)\exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right)$

Which means we have the following density

$$p(y_i|\mathbf{x_i},\beta) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(y_i-\mathbf{x_i}^T\beta)^2}{2\sigma^2}\right)$$

Training the model

$$p(y_i|\mathbf{x_i},\beta) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(y_i-\mathbf{x_i}^T\beta)^2}{2\sigma^2}\right)$$

We want to find a value for Beta that maximizes the probability of all of the data, i.e. a max-likelihood value of Beta

$$\mathcal{L}(\beta) = p(\text{data}|\beta) = \prod_i\left(\frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(y_i-\mathbf{x_i}^T\beta)^2}{2\sigma^2}\right)\right)$$

It will be easier to use log-likelihood as we have done in the past, which will give

$$\frac{1}{2\sigma^2}\sum_i -(y_i-\mathbf{x}_i^T\beta)^2 + \text{some term not depending on } \beta$$

Training the model

$$\frac{1}{2\sigma^2}\sum_i -(y_i - \mathbf{x}_i^T\beta)^2 + \text{some term not depending on } \beta$$

Dropping the constant, we want to maximize

maximize $\sum_i -(y_i - \mathbf{x_i}^T\beta)^2$

Notice that maximizing the above quantity is the same as minimizing the total squared error of our predictions, which had we proceeded to solve the problem from the point of view of minimizing some loss may have been a natural choice

minimize $\sum_i (y_i - \mathbf{x_i}^T\beta)^2$

Training the model

It is convenient to write things out with matrices and vectors for our ultimate solution, so write

$$y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}$$

and

$$X = \begin{pmatrix} \mathbf{x_1}^T \\ \mathbf{x_2}^T \\ \vdots \\ \mathbf{x_N}^T \end{pmatrix}$$

Then the function we wish to optimize becomes

$$(y - X\beta)^T (y - X\beta)$$

Training the model

$$(y - X\beta)^T(y - X\beta)$$

We would then expand this, take the derivative with respect to Beta and set that equal to 0. Pages of unenlightening matrix calculus would tell us that we are interested in finding a Beta that solves

$$X^T X\beta - X^T y = 0$$

Or

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

Training the model

So our algorithm for finding a good weight vector Beta for a given training data set is to covert the **x**'s of the training data to a matrix and the y's to a vector

$$y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix} \quad \text{and} \quad X = \begin{pmatrix} \mathbf{x_1}^T \\ \mathbf{x_2}^T \\ \vdots \\ \mathbf{x_N}^T \end{pmatrix}$$

And then solve $\beta = (X^T X)^{-1} X^T y$

Residuals

We don't expect that the line that we get by finding Beta will perfectly match our training data. Some of the predicted y's in our training data will be different than the actual y's if our data doesn't lie on a perfectly linear surface

The **residual** is the vector

$$\mathbf{e} = \mathbf{y} - X\beta$$

The mean squared error is the number we get by computing

$$m = \frac{\mathbf{e}^T\mathbf{e}}{N}$$

and it gives the average squared error of prediction on the training examples

R squared and explained variance

Another way to measure the quality of our predictions is to look at the amount of variance in the original dependent variable and compare it to the amount of variance in our predictions

The *y_i* over our whole training data can itself be regarded as a dataset, where computing the mean and variance as well-defined. Likewise our $\mathbf{x_i}^T \beta$ can be treated as a dataset

A useful quantity compares the variance of each

$$R^2 = \frac{\text{var}[\mathbf{x_i}^T \beta]}{\text{var}[y_i]}$$

R squared

$$R^2 = \frac{\text{var}[\mathbf{x_i}^T \beta]}{\text{var}[y_i]}$$

This quantity will achieve a maximum of 1 when our regression perfectly predicts the y_i and its minimum value would be 0 if our regression predicted a constant for every x_i

Good predictions result in a high value of R squared