Chapter - 3
Linear Models

Aviral Janveja

1 Linear Classification

Let us consider a real world data-set of hand-written digits collected from postal-stamps. It is always better to test your models on real data, to get a better understanding of how your system would actually perform in the real world. Here is a sample from the data-set :

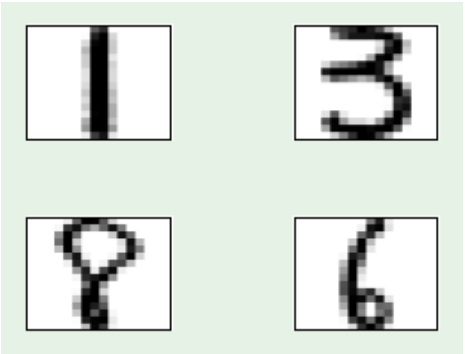

Figure 1: Real Data Example

As shown above, we have a bunch of hand-written digits, collected from postal stamps. We would like to design a model, that can learn to decipher these digits from the given images. People can sometimes write digits in weird ways, making it difficult to understand even for a human operator. Indeed, the error-rate for human operators is found to be around 2.5% and we would like to see if our machine learning model can at least match that or maybe even do better.

1.1 Input Representation

Let us begin by looking at the given input data more closely. We are given a set of grayscale images containing hand-written digits, as shown below :

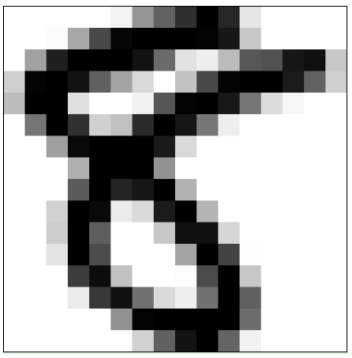

Figure 2: Hand-Written Digit Example

Now, each of these images is 16×16 pixels. Meaning, each digit is represented by 256 real-number attributes. The raw-input $\mathbf{x}$, would therefore look like :

$$\mathbf{x} = (x_1, x_2, x_3 ... x_{256})$$

That is a very long input to represent such a simple object. If we take this raw-input and try the perceptron directly on it, we get too many weights :

$$\mathbf{w} = (w_1, w_2, w_3 ... w_{256})$$

The idea of input representation is to simplify the algorithm's life. We know that it is not about the individual pixel values, when trying to recognize a digit. We can instead extract some relevant **features** from the raw-input and then give those to the learning model and let it figure out the pattern.

1.2 Feature Engineering

Features are basically useful information that can be extracted from the given raw-input. For example, density, symmetry, curve and so on. The digit 1 for instance, will score higher on the symmetry measure as compared to a 5, whereas 5 will score higher on the density score. Using such features instead of the raw-input, significantly simplifies our input representation :

$$\mathbf{x} = (x_1, x_2, x_3)$$

Admittedly, we are losing some information in this process of converting the raw-input to features. But chances are, most of it is irrelevant information anyway. From a generalization point of view as well, going from 256 to 3 parameters is a pretty good situation. Plotting a scatter diagram for digits 1 and 5 alongside just two features - symmetry and density. We get the following graph :

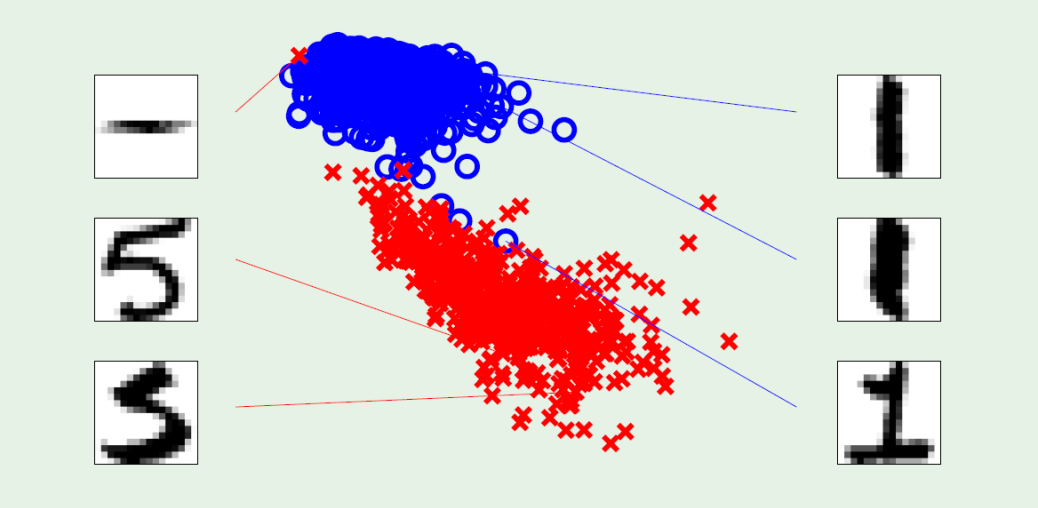

Figure 3: Illustration of Features

The blue points represent 1s and the red points represent 5s. The horizontal-axis represents density and the vertical-axis represents symmetry. Indeed the red 5s are tilted more towards the right on the horizontal-axis, corresponding to their higher density score. Meanwhile the blue 1s are higher on the vertical-axis owing to their higher symmetry score. Just by using these two features, we see that the above data is already almost classified correctly.

1.3 Pocket Perceptron

As seen in chapter 1, the perceptron model implements the following formula :

$$\text{sign}(\mathbf{w}^{\mathbf{T}}\mathbf{x})$$

We iterate over the misclassified points, trying to nudge the weight vector $\mathbf{w}$ such that all points are eventually correctly classified. However, the above digits data-set is not linearly separable, as visible in the graph above. There is a red point, deep in the blue region and other similar outliers that cannot be classified using a straight line.

This means that the perceptron learning algorithm will never stop and keep looping from one misclassified point to another. So, how do we solve this?

Well, we present a pretty straightforward solution here. We can set a limit on the number of iterations, let us say $i = 1000$. Where, one iteration represents one update to the weight-vector $\mathbf{w}$ upon encountering a misclassified point. During these iterations, we keep track of which hypothesis reports the lowest in-sample error E_{in} and report it as our final hypothesis g at the end. This is why, it is called the pocket perceptron. We pick the best candidate so far and put it in our "pocket".

2 Linear Regression

Let us continue with the loan example from chapter 1. We have the following data points :

$$(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), (\mathbf{x}_3, y_3)...(\mathbf{x}_N, y_N)$$

Here, $\mathbf{x}_n$ is a column matrix representing the application information of a customer like age, salary, outstanding debt and so on, same as the perceptron example.

$$\mathbf{x}_n = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ . \\ . \\ x_d \end{bmatrix}$$

However y_n in this case is different. Here, it represents the exact loan amount that is sanctioned for a customer and not just the binary decision, as was the case with perceptron.

2.1 Real Valued Output

Now, the linear regression formula is similar to the perceptron in the sense that, here as well, we assign weights to different customer attributes. However, instead of returning a binary decision $(+1, -1)$, we return the real number value corresponding to the sanctioned loan amount, that is given by :

$$\sum_{i=0}^{d} w_i x_i = w_0 x_0 + w_1 x_1 + .. + w_d x_d \; = \begin{bmatrix} w_0 & w_1 & . & . & w_d \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ . \\ . \\ x_d \end{bmatrix} = \mathbf{w^T x}$$

2.2 Computing Error

In the perceptron model, we minimized the in-sample error (E_{in}) by nudging the weight vector ($\mathbf{w}$) at each misclassified point as per the perceptron update rule. The error in that case was a binary error, as it was just about agreement or disagreement with the classification decision. However, in this case, as the output is real-valued, we use a different error-measure, called the **squared-error**. This happens to be the standard error-measure that is used with linear regression :

$$(\mathbf{w^T x}_n - y_n)^2$$

The above represents the error-value on a single data-point $(\mathbf{x}_n, y_n)$ given by our current weight vector = $\mathbf{w}$. In order to get the in-sample error (E_{in}) for $\mathbf{w}$, we simply take the average of individual error-values :

$$E_{in}(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} (\mathbf{w^T x}_n - y_n)^2$$

This gives us a snapshot of how this $\mathbf{w}$ is doing on the data-set. Next, we can re-write the above expression in matrix-notation as follows :

$$E_{in}(\mathbf{w}) = \frac{1}{N} ||\mathbf{Xw} - \mathbf{y}||^2$$

Here $\mathbf{X}$ is the combined matrix of all the customer input vectors $\mathbf{x}_n$ from 1 to N, as illustrated below :

$$\mathbf{X} = \begin{bmatrix} [...\mathbf{x_1^T}...] \\ [...\mathbf{x_2^T}...] \\ . \\ . \\ [...\mathbf{x_N^T}...] \end{bmatrix}$$

Whereas, $\mathbf{y}$ is the combined matrix of all the real-valued outputs :

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ y_N \end{bmatrix}$$

2.3 Minimizing Error

Next, in order to minimize the in-sample error $E_{in}(\mathbf{w})$ calculated above, we take its gradient (derivative) and equate it to zero.

$$\frac{d}{d\mathbf{w}} E_{in}(\mathbf{w}) = \frac{2}{N} \mathbf{X^T}(\mathbf{Xw} - \mathbf{y}) = \mathbf{0}$$

Solving the right-hand-side above for $\mathbf{w}$, we get :

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

This can be written as :

$$\mathbf{w} = \mathbf{X}^\dagger \mathbf{y}$$

Where $\mathbf{X}^\dagger = (\mathbf{X^T X})^{-1} \mathbf{X^T}$ is called the **pseudo-inverse** of $\mathbf{X}$ and pronounced as "x dagger".

And that's it, through the derivative equation, we get the required $\mathbf{w}$, corresponding to the minimum in-sample error. Note how all of this happens in a single step, unlike the point by point nudging witnessed in the perceptron model.

3 References

1. CalTech Machine Learning Course - CS156, Lecture 3.