MLPR 2022: Week 1

Joshua Cesare Placidi

November 6, 2022

1 w1a: Course Introduction

1.1 Learning functions example: text classification

Machine learning is fitting a function to a set of data. As an example we can learn a function that could classify if a text extract contains spam or not. First we need to extract a *feature vector* $\mathbf{x}$, a vector of numbers indicating word counts, a linear function can then be constructed:

$$f(\mathbf{x}) = w_1 x_1 + w_2 x_2 + \cdots + w_D x_D = \sum_{d=1}^{D} w_d x_D = \mathbf{w}^\top \mathbf{x}$$

By manually setting each weight w_d to be positive for 'spammy' words and negative for 'normal' words we can create a spam text classifier. If $f(\mathbf{x}) > 0$ for a feature vector we can predict that this text contains spam.

2 w1b: Linear Regression

2.1 Affine Functions

An *affine function* (aka a linear function) of a vector $\mathbf{x}$ is a weighted sum of each value $x_i \in \mathbf{x}$ and an added constant, the bias term. For example, for $D = 3$ inputs $\mathbf{x} = [x_1\ x_2\ x_3]^\top$, a general (scalar) affine function is:

$$f(\mathbf{x}; \mathbf{w}, b) = w_1 x_1 + w_2 x_2 + w_3 x_3 + b = \mathbf{w}^\top \mathbf{x} + b$$

where $\mathbf{w}$ is a D-dimensional vector of *weights* (the coefficients of each of the x terms). b is the *bias weight* it gives the value of the function at $\mathbf{x} = \mathbf{0}$. $\mathbf{x}$ is the feature vector describing a setting that we want our model to consider.

2.2 Fitting to Data

Good values for the weights $\mathbf{w}$ and b can be found using training data, a set of N input-output pairs $\{\mathbf{x}^{(n)}, y^{(n)}\}_{n=1}^{N}$. The training data contains feature vectors

$\mathbf{x}^{(n)}$ and the desired output of our function $y^{(n)}$. To write fast code we stack training examples together:

$$y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(N)} \end{bmatrix}, \; X = \begin{bmatrix} x^{(1)\top} \\ x^{(2)\top} \\ \vdots \\ x^{(N)\top} \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \dots & x_D^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \dots & x_D^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(N)} & x_2^{(N)} & \dots & x_D^{(N)} \end{bmatrix} \tag{1}$$

We can simultaneously evaluate the function at every training input with one matrix-vector multiplication: $\mathbf{f} = X\mathbf{w} + b$, where the scalar b is added to each elements of the vector $X\mathbf{w}$. We can compute the total square error of the function predictions $\mathbf{f}$ compared to the labels $\mathbf{y}$ using $f_n = f(\mathbf{x}^{(\mathbf{n})}; \mathbf{w}, b) = \mathbf{w}^\top \mathbf{x}^{(n)} + b$ and calculating the sum of the squared difference between $\mathbf{f}$ and $\mathbf{y}$:

$$\sum_{n=1}^{N} [y^{(n)} - f(\mathbf{x}^{(n)}; \mathbf{w}, b)]^2 = (\mathbf{y} - \mathbf{f})^\top (\mathbf{y} - \mathbf{f})$$

The least-squares fitting problem is finding the parameters that minimise this error. If we assume $b = 0$ we can fit a *linear map*:

$$\mathbf{y} \approx f = X\mathbf{w}$$

the weights $\mathbf{w}$ can be found using the following Python and NumPy code: `w_fit = np.linalg.lstsq(X, yy, rcond=None)`, here `w_fit` has shape $D \times 1$ if X is $N \times D$ and $\mathbf{y}$ is $N \times 1$. This learned function will always pass through the origin because $b = 0$. This constraint can be removed by adding an extra column to the design matrix X that appends a 1 to every row:

$$\tilde{X} = \begin{bmatrix} x^{(1)\top} & 1 \\ x^{(2)\top} & 1 \\ \vdots & \vdots \\ x^{(N)\top} & 1 \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \dots & x_D^{(1)} & 1 \\ x_1^{(2)} & x_2^{(2)} & \dots & x_D^{(2)} & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_1^{(N)} & x_2^{(N)} & \dots & x_D^{(N)} & 1 \end{bmatrix}$$

Least squares can be used again to fit the weights $\tilde{\mathbf{w}}$. If the input was D-dimensional before, we will now fit $D+1$ weights, $\tilde{\mathbf{w}}$. The last weight $\tilde{w}_{D+1}$ will always be multiplied by 1 and so is actually the bias weight b, while the first D weights gives the regression weights for the original design matrix:

$$\tilde{X}\tilde{\mathbf{w}} = X\tilde{\mathbf{w}}_{1:D} + \tilde{w}_{D+1} = X\mathbf{w} + b$$

2.3 Design Matrix Transformation

The same linear-regression code can be used to fit non-linear functions by applying transformations to input features. We can represent the new transformed feature matrix as Φ with $\Phi_{n,:} = \boldsymbol{\phi}(\mathbf{x}^{(n)})^\top$. If the function $\boldsymbol{\phi}$ is non-linear then the function $f(\mathbf{x}) = \mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x})$ will be non-linear in $\mathbf{x}$. However, we can still use least-squares to fit the weights $\mathbf{w}$ because the function is still a linear map of $\boldsymbol{\phi}(\mathbf{x})$.

An introductory example is fitting a polynomial curve, this can be achieved by having each column in the new design matrix Φ be a monomial of the original feature:

$$\Phi = \begin{bmatrix} 1 & x^{(1)} & (x^{(1)})^2 & \dots & (x^{(1)})^{K-1} \\ 1 & x^{(2)} & (x^{(2)})^2 & \dots & (x^{(2)})^{K-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x^{(N)} & (x^{(N)})^2 & \dots & (x^{(N)})^{K-1} \end{bmatrix}$$

Using Φ as our design matrix we can then fit the model:

$$\boldsymbol{\phi}(\mathbf{x}) = \begin{bmatrix} 1 & x_1 & x_2 & x_3 & x_1x_2 & x_1x_3 & x_2x_3 & x_1^2 & \dots \end{bmatrix}^\top$$

this would fit a multivariate polynomial function of the original features. Given that a general polynomial includes cross terms like x_1x_2, x_1x_3, x_2x_3, the number of columns in Φ could be large. Any vector-valued function can be used to generate the columns of Φ:

$$\Phi_{n,:} = \boldsymbol{\phi}(\mathbf{x}^{(n)})^\top = \begin{bmatrix} \phi_1(\mathbf{x}^{(n)}) & \phi_2(\mathbf{x}^{(n)}) & \dots & \phi_K(\mathbf{x}^{(n)}) \end{bmatrix}^\top$$

each ϕ_k is called a *basis function*. The function we fit is a linear combination of the set of basis function transformations of $\mathbf{x}$:

$$f(\mathbf{x}) = \mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x}) = \sum_k w_k \phi_k(\mathbf{x})$$

The bias term can be added by setting ϕ_K to be constant.

2.4 Basis Function Variations

One choice for a basis function is a *Radial Basis Function* (RBF):

$$\text{RBF}(\mathbf{x}; \mathbf{c}, h) = \exp(-(\mathbf{x} - \mathbf{c})^\top (\mathbf{x} - \mathbf{c}) \frac{1}{h^2})$$

where $\mathbf{c}$ and h are parameters used to define the RBF. The function is proportional to a Gaussian probability density function, it is a bell shaped curve centred at $\mathbf{c}$ with *bandwidth h*. Another basis function is the *logistic-sigmoid* function:

$$\text{logistic-sigmoid}(\mathbf{x}; \mathbf{v}, b) = \sigma(\mathbf{v}^\top \mathbf{x} + b) = \frac{1}{1 + \exp(-\mathbf{v}^\top \mathbf{x} - b)}$$

this is an s-shaped curve which saturates at zero and one for extreme values of $\mathbf{x}$. The parameters $\mathbf{v}$ and b determine the steepness and position of the curve respectfully.

2.5 Summary

Using least-squares fitting code we can fit linear functions to training data. The same code can be further used to fit non-linear functions by applying transformation functions to the original training data. Using a set of non-linear transforms such as RBFs and logisitic-sigmoids a new feature matrix Φ can be generated which is non-linear in $\mathbf{x}$. Functions fitted using Φ no longer have the constraint of being linear in $\mathbf{x}$.