Machine Learning and Pattern Recognition

Notes from lectures given by Dr Arno Onken in 2024

Thomas Davies

Last updated: September 25, 2024

Contents

Introduction

These are notes that were taken in 2024 for the MLPR course with lectures given by Dr Arno Onken. They were mostly taken live in lectures, in particular, all errors are almost surely mine. The provided lecture notes can be found here. These are not endorsed by the lecturer, and are mostly for my own reference. It should be noted that these notes will be much more maths focused than the ones provided in lectures, this is mainly for my own comfort (I come from a maths background). I may also later add a GitHub link with python (or maybe rust) implementations of the methods provided from scratch (where scratch is defined as starting from some base linear algebra library like numpy).

Lecture 1

This lecture was primarily describing why one would want to study machine learning, and gave some interesting examples such as face detection. I would recommend reading the provided notes, however I don't think there is any value in me reproducing this text.

Lecture 2

2.1 Least Squares

Very very often, machine learning problems can be reduced in some form to linear regression (see the Kernel Method, Gaussian Process Regression, and more exotically the Kernel Regime for neural networks). So it is important to understand the basics of what we are doing in linear regression. First we need a couple of basic definitions.

Definition 2.1. *An **affine function** $f : \mathbb{R}^k \to \mathbb{R}^m$ is one of the form*

$$f_{A,b}(x) = Ax + b$$

where $x \in \mathbb{R}^k$, $A \in \mathbb{R}^{m\times k}$, and $b \in \mathbb{R}^m$. Note that notationally m, k are often renamed and can change based on the context (although it is normally easy to infer the dimensions).

In the lectures, m was taken to be 1, and A was denoted as w^T, and none of the derivations of the ensuing results were given, however I think there is value in doing them. We now also assume that we have some **training data**.

Definition 2.2. ***Supervised training data*** *is a pair of "labelled" data*

$$\mathcal{T} = \{(x^{(n)}, y^{(n)}) \mid n = 1, 2, \ldots, N\}$$

In the unsupervised case (which we cover later in the course), we are missing $y^{(n)}$ but still hope to be able to recover our underlying function via building structure into our model.

We assume that it was sampled from a **linear model** i.e. there is some approximate relationship $f(x_n) \approx y_n$ for some affine function f (for a slightly better way to describe this see ordinary least squares). We would like a way to recover a good estimate of A and b from out training data, one way to do this is to minimise the loss in the 2-norm (squared error). We will denote our approximated (affine) function as $\tilde{f}$, and our approximated parameters as $\tilde{A}$, and $\tilde{b}$. The loss function will be denoted as $\mathcal{L}(g)$. Using the 2-norm (with no regularisation).

$$\mathcal{L}(\tilde{f}) = \sum_{n=1}^{N} \left\| y^{(n)} - \tilde{f}(x^{(n)}) \right\|_2^2 = \sum_{n=1}^{N} \sum_{i=1}^{k} (y_i^{(n)} - \tilde{f}_i(x^{(n)}))^2 \tag{1}$$

We seek (as we normally do in machine learning) to minimise this loss function over our parameters $\tilde{A}, \tilde{b}$. Note that as a sanity check, if we perfectly recovered f, that the loss would be 0 i.e. $\mathcal{L}(f) = 0$. Now, let's try find a closed form solution (under certain regularity conditions which we will state as they arise).

Definition 2.3. *A* ***design matrix*** $X \in \mathbb{R}^{N \times k}$*, and* ***output matrix*** $Y \in \mathbb{R}^{N \times m}$ *are stacked versions of our training data.*

$$X = \begin{bmatrix} - & x^{(1)T} & - \\ - & x^{(2)T} & - \\ & \vdots & \\ - & x^{(N)T} & - \end{bmatrix} \qquad Y = \begin{bmatrix} - & y^{(1)T} & - \\ - & y^{(2)T} & - \\ & \vdots & \\ - & y^{(N)T} & - \end{bmatrix}$$

We can re write the loss in equation 1 by subbing in our parameters, setting $\tilde{b} = 0$ (this will be expounded on later), and re-writing the norm in terms of the design matrix as follows, where $\tilde{a}_i$ is the ith row of $\tilde{A}$.

$$\mathcal{L}(\tilde{f}) = \sum_{i=1}^{m} (y_i^{(n)} - \tilde{a}_i X)^T (y_i^{(n)} - \tilde{a}_i X) \tag{2}$$

Now, differentiating with respect to $\tilde{A}$, and setting it to 0 we get the following closed form for $\tilde{A}$.

$$\tilde{A} = (X^T X)^{-1} X^T Y \tag{3}$$

Where we require $X^T X$ to be invertible for this to hold. This is clearly a global minima as when $\tilde{A} \to \infty$ we have $\mathcal{L}(\tilde{f}) \to \infty$ under the condition that $X^T X$ is non singular). If this is non obvious, note that we can minimise each row individually.

As for why we can wlog $b = 0$, we simply extend our matrix by one collum, and add one synthetic "observation" collum to Y with values all 1, one synthetic "input" collum to X with all 0s. So our design, output matrices become.

$$X = \begin{bmatrix} - & x^{(1)T} & - & 1 \\ - & x^{(2)T} & - & 1 \\ & \vdots & & \\ - & x^{(N)T} & - & 1 \end{bmatrix} \qquad Y = \begin{bmatrix} - & y^{(1)T} & - & 0 \\ - & y^{(2)T} & - & 0 \\ & \vdots & & \\ - & y^{(N)T} & - & 0 \end{bmatrix} \qquad \tilde{A} = \begin{bmatrix} - & \tilde{a}^{(1)T} & - & \tilde{b}_1 \\ - & \tilde{a}^{(2)T} & - & \tilde{b}_2 \\ & \vdots & & \\ - & \tilde{a}^{(m)T} & - & \tilde{b}_m \end{bmatrix} \tag{4}$$

The resulting loss function is the same, so all of our results above still hold.

Let's take some data, and find the least squares fit of it in python, we will use the numpy library, which has a lstsq function builtin.

```
import numpy as np
import matplotlib.pyplot as plt

x_grid = np.arange(-3, 3, 1)
m = 5
variance = 2
f_vals = m*x_grid + np.random.normal(0, variance, len(x_grid))
plt.plot(x_grid, m*x_grid, 'b-')
plt.plot(x_grid, f_vals, 'r.')
mpred = np.linalg.lstsq(np.array([x_grid]).T, f_vals, rcond=None)[0]
plt.plot(x_grid, mpred*x_grid, 'g-')
plt.show()
```

We have setup a 1 dimensional linear regression problem, with normal noise and our gradient being 5. Running the code, we predict that the gradient is 4.508, which is quite close to the true value. Plotting them in figure 1, we can see how even though the sampled points are quite far off the line, we can still get a good estimate when reconstructing the underlying function.

2.2 Linear Regression with Basis Functions

Now, an obvious question is what if we want to fit a model which isn't linear? Well we can actually still do linear regression, but we just transform our domain prior to regressing i.e. our features go from $(x_1, x_2, \ldots, x_m)$ to $(\phi_1(x), \phi_2(x), \ldots, \phi_D(x))$ where D is the number of basis functions, and

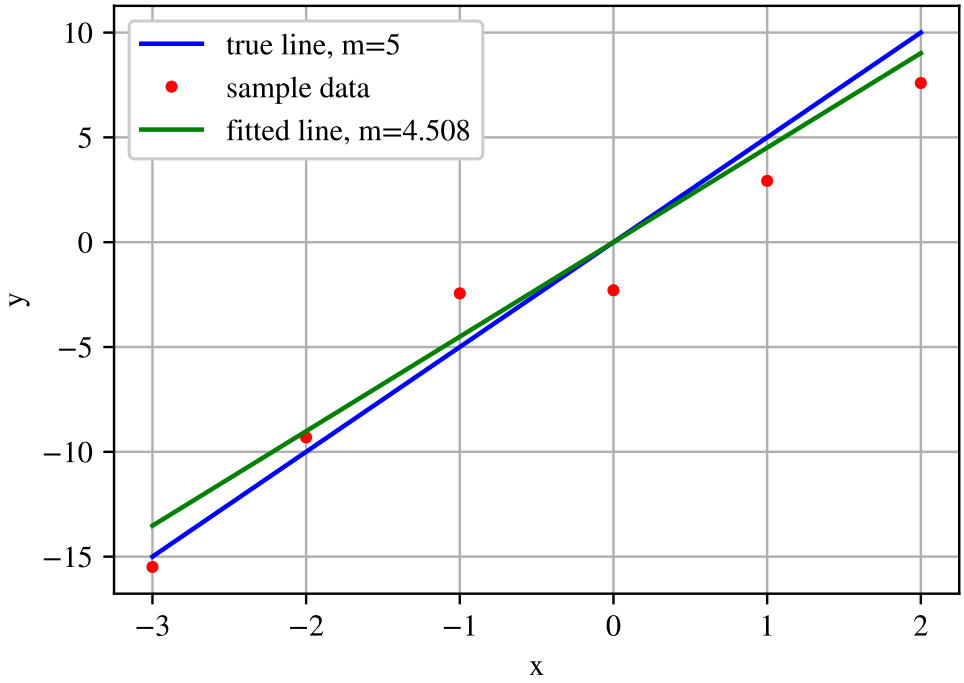

Figure 1: Least squares fit

$\phi_i : \mathbb{R}^m \to \mathbb{R}$, are some functions that we can specify. Again, if we look at the loss function above, our closed form still holds, we have just relabelled our input features to be some mapping of our vector space. This also extends to any linear regression technique like **lasso**, and **ridge** regression which we will cover later. We will now list some common basis functions.

2.2.1 Polynomials

An obvious first choice would be polynomials, they are dense in $C(K)$ where $K \subset \mathbb{R}^p$ is compact, so with infinite data, we could recover the function to arbitrary accuracy. However, this scales very badly in dimension as to fit a polynomial with D input features, a $k-1$ degree polynomial needs $\frac{(k+D-1)!}{k!(D-1)!}$ coefficients. For small dimensions however, these can be a good option. They also suffer from heavy overfitting, (which can be motivated by looking at taylor expansion remainders probably?). I will assume that you don't need me to plot some polynomials to know what they look like.

2.2.2 Radial Basis Functions

Definition 2.4. *A **Radial Basis Function** or **RBF** for short is a function $\phi_{c,h^2} : \mathbb{R}^p \to \mathbb{R}$ which has the form.*

$$\phi_{c,h^2} = \exp\left(-\frac{(x-c)^T(x-c)}{h^2}\right)$$

where $c \in \mathbb{R}^p$ and $h^2 \in \mathbb{R}^+$

A couple plots of RBFs with varied values of c, and h^2 can be found in figure 2. Note that c gives the center of the curve, and h^2 is the "sharpness", where smaller values give a sharper curve, and larger ones give a wider curve.

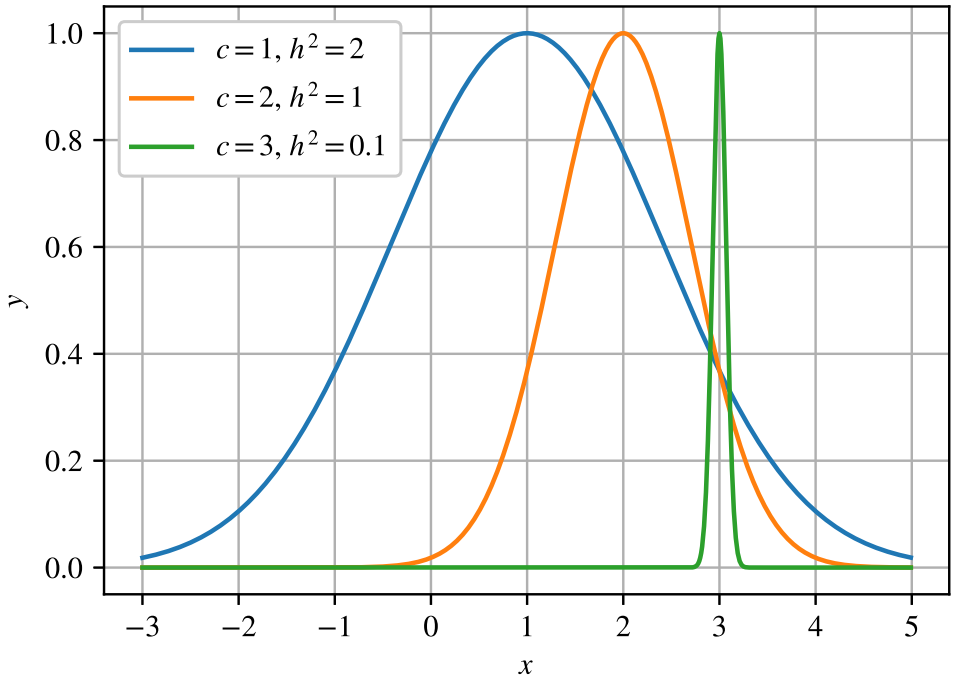

Figure 2: Plot of RBFs

Definition 2.5. *A **Sigmoidal Function** is a function $\psi_{v,b} : \mathbb{R}^p \to \mathbb{R}$ which has the form.*

$$\psi_{v,b} = \sigma(v^T x + b)$$

*Where $v \in \mathbb{R}^p$, and $b \in \mathbb{R}$, and with $\sigma : \mathbb{R} \to \mathbb{R}$ the **Logistic Sigmoid** function, which has the following form.*

$$\sigma(r) = \frac{1}{1 + \exp(-x)}$$

A couple of plots of sigmoidal functions with varied values of v and b can be found in figure 3. Note that v gives the "steepness" of the curve with larger values making it steeper, and b changes the "center" of the curve. These are especially useful in logistic regression.

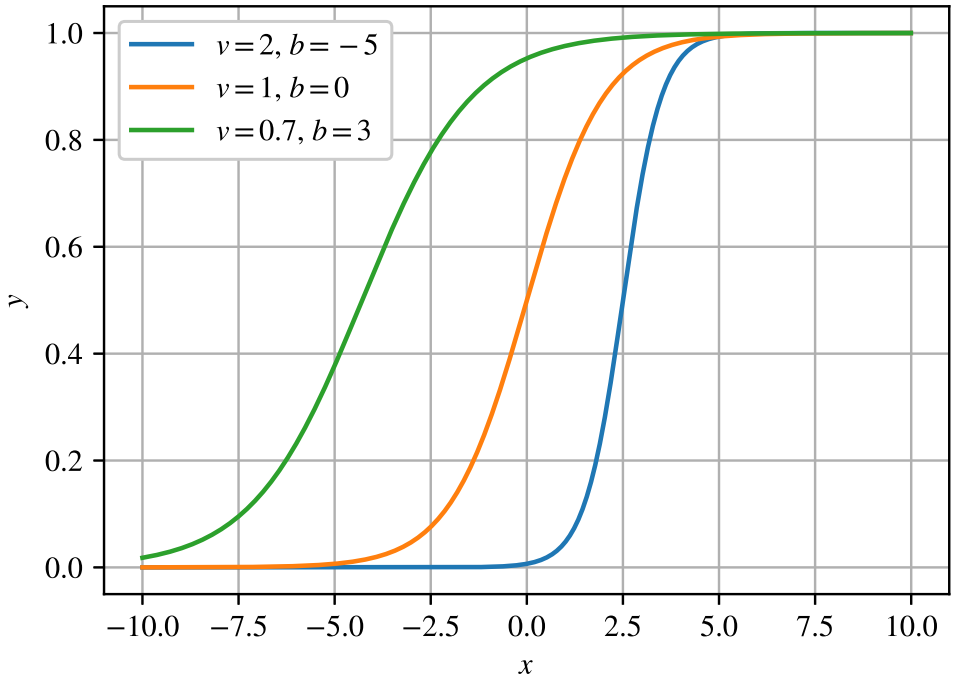

Figure 3: Plot of Sigmoidal functions

Lecture 3

3.1 Plotting and Understanding Basis Functions

Lets try fit some sample data with various different selection of basis functions. It's important to note that **none of the choices are wrong**, regression is an inherently ill posed problem. We'll

use 3 different sets of basis functions, a linear model, RBFs, and sigmoids. The code can be found below, and the plot can be found in figure 4.

```
import numpy as np
import matplotlib.pyplot as plt

# Set up and plot the dataset
yy = np.array([1.1, 2.3, 2.9]) # N,
X = np.array([[0.8], [1.9], [3.1]]) # N,1
# phi-functions to create various matrices of new features
# from an original matrix of 1D inputs.
def phi_linear(Xin):
    return np.hstack([np.ones((Xin.shape[0],1)), Xin])
def phi_quadratic(Xin):
    return np.hstack([np.ones((Xin.shape[0],1)), Xin, Xin**2])
def fw_rbf(xx, cc):
    """fixed-width RBF in 1d"""
    return np.exp(-(xx-cc)**2 / 2.0)
def phi_rbf(Xin):
    return np.hstack([fw_rbf(Xin, 1), fw_rbf(Xin, 2), fw_rbf(Xin, 3)])

def fit_and_plot(phi_fn, X, yy):
    # phi_fn takes N, inputs and returns N,K basis function values
    w_fit = np.linalg.lstsq(phi_fn(X), yy, rcond=None)[0] # K,
    X_grid = np.arange(0, 4, 0.01)[:,None] # N,1
    f_grid = np.dot(phi_fn(X_grid), w_fit)
    plt.plot(X_grid, f_grid)

fit_and_plot(phi_linear, X, yy)
fit_and_plot(phi_quadratic, X, yy)
fit_and_plot(phi_rbf, X, yy)
plt.plot(X, yy, 'o')
plt.legend(('linear fit', 'quadratic fit', 'rbf fit', 'data'), framealpha=1)
plt.xlabel('$x$')
plt.ylabel('$f$')
plt.show()
```

The reason none of these are wrong is because they are all (with the slight exception of the linear

model) consistent with our training data. We need to test the fits to ascertain which model is best.

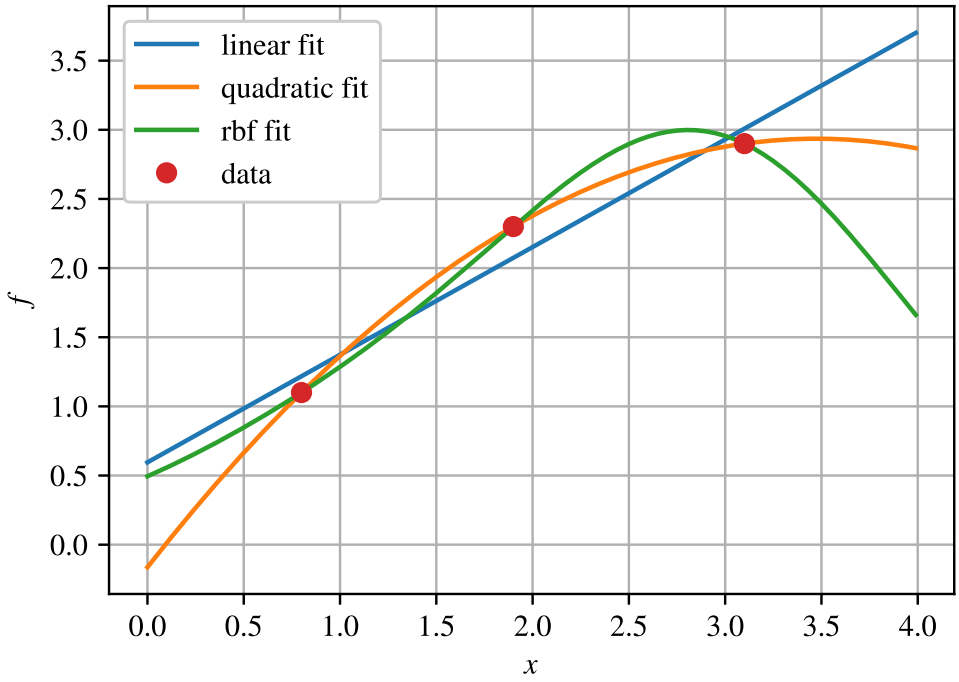

Figure 4: Fits with various different basis function

3.2 Overfitting, Underfitting, and Regularisation