Machine Learning and Pattern Recognition - Week 1 - Linear Regression

Antonio León Villares

September 2022

Contents

Based on the online notes here.

1 Linear Regression

- **What is one of the main objectives of machine learning?**
  - we have a set of inputs $\{\underline{x}^{(i)}\}_{i=1}^{N}$ and corresponding outputs $\{y^{(i)}\}_{i=1}^{N}$
  - for example:
    - an input can be an image, and the output will be the probability of it containing a face
    - an input can be an email, and the output will be whether it is spam or not
  - machine learning focuses on **learning** a function f, which given an input, predicts the desired output
  - to do this, we need to answer 3 questions:
    1. How do we create a numerical (i.e vector) representation of the input?
    2. How do we create a numerical (i.e vector) representation of the output?
    3. How do we set the parameters of f?
- **What is linear regression?**
  - a method which uses an **affine function** as a model for prediction
  - given $\underline{x}$, an **affine function** is a weighted linear combination of the features of $\underline{x}$:

$$f(\underline{x};\underline{w},b) = \underline{w}\cdot\underline{x} + b = b + \sum_{i=1}^{D} w_i x_i$$

  - here:
    - $\underline{x} \in \mathbb{R}^D$ are the **input features**
    - $\underline{w} \in \mathbb{R}^D$ is a **weight vector**
    - $b \in \mathbb{R}$ is a **bias weight**
- **What is a design matrix?**
  - consider a training set of N observations:

$$\{\underline{x}^{(i)}\}_{i=1}^{N}$$

  - the inputs can be put into a **design matrix**, where each observation corresponds to a row:

$$X = \begin{pmatrix} \underline{x}^{(1)T} \\ \vdots \\ \underline{x}^{(N)T} \end{pmatrix} = \begin{pmatrix} x_1^{(1)} & x_2^{(1)} & \dots & x_D^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \dots & x_D^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(N)} & x_2^{(N)} & \dots & x_D^{(N)} \end{pmatrix}$$

2 Fitting Linear Regression

- **How can we use matrices to represent the outputs of a linear regression model?**
  - let X be our design matrix of data, and $\underline{w}, b$ the weights obtained from linear regression

- then our model predicts:

$$\underline{f} = X\underline{w} + b$$

where $\underline{f} \in \mathbb{R}^N$

- **What is the least squares fitting problem in linear regression?**
  - the problem of computing the parameters of the linear regression model
  - finding $\underline{w}, b$ such that:

$$\theta = \underset{\underline{w},b}{argmin} \sum_{i=1}^{N} \left(y^{(n)} - f(\underline{x}^{(n)}; \underline{w}, b) \right)^2$$

  - if we use:

$$\underline{f} = X\underline{w} + b$$

then more succintly:

$$\theta = \underset{\underline{w},b}{argmin} \left((\underline{y} - \underline{f})^T (\underline{y} - \underline{f}) \right)$$

  - other error functions can be used, such as $\sum_{i=1}^{N} |y^{(n)} - f(\underline{x}^{(n)}; \underline{w}, b)|$, but the sum of squared residuals is more convenient computationally

- **How can we solve the least squares fitting problem without a bias term?**
  - solving least squares without a bias is much simpler, since we have:

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

  - the answer is ... `numpy`:

```
import numpy as np

# we have 13 observations, each with 7 features
D,N = 7, 13

# create random design matrix
X = np.random.randn(N,D)

# generate random outputs
y = np.random.randn(D)

# compute weights by using least squares
w_pred = np.linalg.lstsq(X,y, rcond = None)[0]
```

  - if X were a square, invertible matrix, then:

$$\underline{y} = X\underline{w} \implies \underline{w} = X^{-1}\underline{y}$$

  - otherwise, we can use the **pseudo-inverse**, which is always guaranteed to exist:

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

- **How can we solve the least squares fitting problem with a bias term?**

– just modify the design matrix, by adding a column of 1s:

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

– the weight vector $\underline{w}$ will get modified, to include the bias term:

$$\underline{\tilde{w}} = \begin{pmatrix} w_1 \\ \vdots \\ w_D \\ b \end{pmatrix} \in \mathbb{R}^{D+1}$$

– then, we can treat the **bias** weight as a normal weight:

$$\underline{x} = \tilde{X}\underline{\tilde{w}} = X\underline{w} + b$$

– we can then fit least squares as above

```
X_bias = np.concatenate([X, np.ones((X.shape[0],1))], axis=1)
```

3 Basis Functions

- **What is the key issue of the linear regression model?**
  - it is **linear**, so it assumes that data is distributed **linearly**
  - however, this doesn't typically correspond with real world data
- **What is a basis function?**
  - a **vector-valued** function:

$$\underline{\phi} : \mathbb{R}^D \to \mathbb{R}^K$$

$$\underline{\phi(\underline{x})} = \begin{pmatrix} \phi_1(\underline{x}) \\ \vdots \\ \phi_K(\underline{x}) \end{pmatrix}$$

- **How can linear regression be fit to non-linear data?**
  - by applying ϕ to our input $\underline{x}$, we can create a new set of K features to represent an observation:

$$\underline{x} \mapsto \phi(\underline{x})$$

  - then, our linear regression model becomes a **linear combination** of **basis functions**:

$$f(\underline{x}) = \underline{w} \cdot \underline{\phi(\underline{x})} = \sum_{i=1}^{K} w_i \phi_i(\underline{x})$$

- if the basis functions are **non-linear**, then f will be **non-linear** in $\underline{x}$ (but still a linear function of $\phi(\underline{x})$)
- notice, we don't include the bias term, since we can define one of the $\phi_i(\underline{x}) = 1$
- out design matrix now becomes:

$$\tilde{X} = \begin{pmatrix} \underline{\phi(\underline{x}^{(1)})}^T \\ \underline{\phi(\underline{x}^{(2)})}^T \\ \vdots \\ \underline{\phi(\underline{x}^{(N)})}^T \end{pmatrix} = \begin{pmatrix} \phi_1(\underline{x}^{(1)}) & \phi_2(\underline{x}^{(1)}) & \dots & \phi_K(\underline{x}^{(1)}) \\ \phi_1(\underline{x}^{(2)}) & \phi_2(\underline{x}^{(2)}) & \dots & \phi_K(\underline{x}^{(2)}) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(\underline{x}^{(N)}) & \phi_2(\underline{x}^{(N)}) & \dots & \phi_K(\underline{x}^{(N)}) \end{pmatrix}$$

  which can be fitted as above

3.1 The Polynomial Basis Functions

- **What is a polynomial basis function?**
  - $\phi(\underline{x})$ combines the features in $\underline{x}$ to produce a polynomial
  - for example in $\mathbb{R}^2$ if:

$$\underline{x} = (x_1 \ x_2)^T$$

    a (possible) polynomial basis function could be:

$$\underline{\phi(\underline{x})} = \begin{pmatrix} x_1 \\ x_2^2 \\ x_1^3 + x_2^4 \\ x_1 x_2^2 \\ 1 \end{pmatrix}$$

  - the above example would then be a model for a polynomial of degree 4 (in terms of x_2)
- **Why are polynomial basis functions not used in practice?**
  - **feature space**: for high dimensional inputs, the number of polynomial features which can be extracted grows exponentially
  - **output size**: as $\underline{x}$ moves away from $\underline{0}$, $\phi(\underline{x})$ will produce very large outputs.

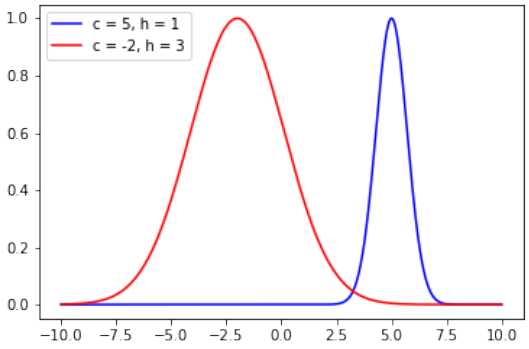
> *One case where I might consider polynomials is when I have **sparse binary features**. That is where only a few of the features are non-zero. If the features are binary, none of the polynomial terms take on extreme values.*
> *"Interaction terms", such as $x_1 x_2$, detect whether it's important for features to be on at the same time. These extra features are more sparse than the original features, and in careful implementations might not create much more work.*

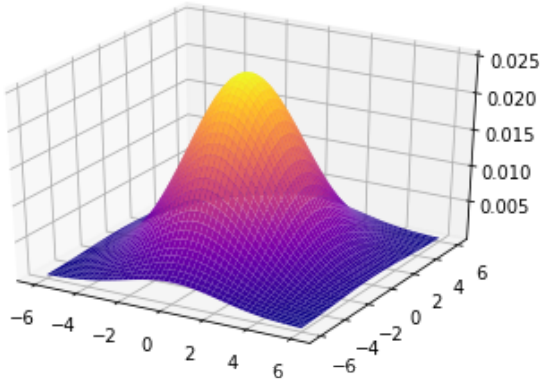
3.2 The Radial Basis Function

- **What is a radial basis function?**
  - a basis function with parameters $\underline{c}, h$:

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

  - this looks like a **Gaussian** curve, centered at $\underline{c}$
- **What do $\underline{c}$ and h control?**
  - $\underline{c}$:
    - the RBF is **radially** symmetric: the value only depends on the radial distance of $\underline{x}$ to $\underline{c}$
    - the closer that $\underline{x}$ is to $\underline{c}$, the larger the output of $\underline{\phi}$ (when $\underline{x} = \underline{c}$, $\underline{\phi} = 1$)
    - we can think of $\underline{c}$ as a "canonical example" for the RBF
  - h
    - h controls the "width" of the RBF
    - it determines how we "score" the closeness of $\underline{x}$ and $\underline{c}$
    - as $h \to 0$, $\underline{\phi} \to \underline{0}$
    - as $h \to \infty$, $\|\underline{\phi}\| \to \sqrt{K}$

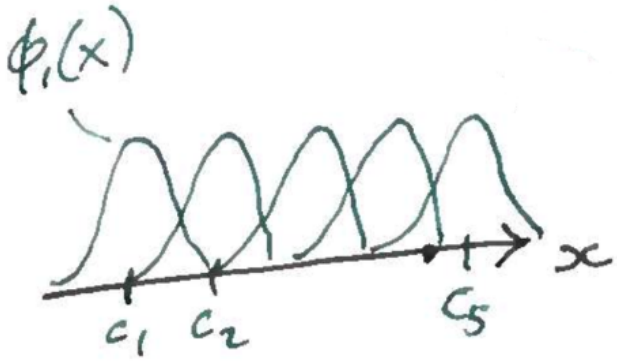




- **Why are RBFs useful?**

- given enough RBFs, it is possible to approximate many smooth functions easily
- makes it good for fitting non-linear data

- **How can we set the parameters of RBFs?**
  - for low-dimensional data, we can get away with evenly spacing them out:



  - in high-dimensions, this is unfeasible, and in practice, just select n random training points to use as centres

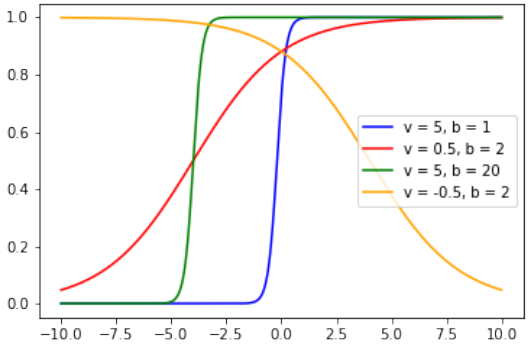
3.3 The Logistic-Sigmoid Basis Function

- **What is the logistic-sigmoid basis function?**
  - a basis function with parameters $\underline{v}, b$:

$$\sigma(\underline{v} \cdot \underline{x} + b) = \frac{1}{1 + exp(-\underline{v} \cdot \underline{x} - b)}$$

  - this takes the shape of a **sigmoid**, with extreme values leading to 0 or 1

- **What do $\underline{v}$ and b control?**
  - $\underline{v}$:
    - determines the "steepness" of the curve (greater steepness as $\|v\| \to \infty$)
    - if $\underline{v}$ is negated, flips the shape of the curve about a vertical plane
  - b:
    - shifts the curve horizontally



- **Why is the logistic-sigmoid useful?**
  - since it uses a dot product, it allows the function to focus on specific parts of the input
  - for example, if given an image, a logistic-sigmoid can have specific $\underline{v}, b$ to focus on a certain quadrant
  - it can then output a 1 if it detects a certain feature, and a 0 (or close) otherwise
  - RBFs would focus on the **whole** image, whilst the logistic-sigmoid can output a feature vector with values for specific features of the input

4 Overfitting and Regularisation

4.1 Residuals and Underfitting

- **What is a residual?**
  - the difference between the true label, and the predicted label of a model:

$$\underline{y}^{(n)} - f(\underline{x}^{(n)})$$

- **Why does a fitted function not always match the training data exactly?**
  1. **Noise/Randomness**: real-world data is inherently noisy, so it's impossible to **exactly** predict y from $\underline{x}$ (for instance, in baking, we can mix the same ingredients, and follow the same steps, but the same cake won't appear all the times)
  2. **Real Data Isn't a Function**: it is unlikely that real data is generated by a specific function, so a linear combination of basis functions won't be exact

- **What is underfitting?**
  - when the model doesn't gauge the nuances of the data (doesn't "understand" the underlying function)
  - it can't represent all of the structure evident in the training data
  - for example, if we have cubic data, but fit a model with quadratic basis functions, we will never fit the data accurately
  - can also arise if we don't use enough basis functions (i.e 3 RBFs to model a cubic won't be sufficient)

4.2 Overfitting

- **What is overfitting?**
  - when the model learns to model the noise inherent in the training data
  - this can mean that the residuals might be low, but the model is unreliable

- **Even if the residuals are small, can we trust a model?**
  - if we have a small number N of training points, it is possible to fit the data exactly (for instance by using N RBFs)
  - however, a small sample is **unreliable** - can't predict the future from 5 observations
  - if the sample has a lot of noise, then the model won't be useful

4.2.1 Note: An Approach to Modelling

*Start with a **simple** model with only a few parameters (this may underfit). We could then consider a series of more **complicated** models while we feel that fitting these models can still be justified by the amount of data we have.*
***However, limiting the number of parameters in a model isn't always easy or the right approach**. If our inputs have many features, even simple linear regression (without additional basis functions) has many parameters. An example is predicting some characteristic of an organism (a phenotype) from DNA data, which is often in the form of 10^5 features.*
*We could consider **removing features from high-dimensional inputs to make a smaller model**, but filtering is not always the correct approach either. If some features are noisy measurements of the same underlying property, it is better to average all of them rather than to select one of them. However, we may not know in advance which groups of features should be averaged, and which selected.*
*Another approach to modelling is to use **large** models (models with many free parameters), but to **discourage unreasonable fits** that match our noisy training data too closely.*

4.2.2 Overfitting Example: Many Features

We consider training data generated stochastically:

$$\mu^{(n)} \sim Uniform(0, 1)$$

$$x_d^{(n)} \sim \mathcal{N}(\mu^{(n)}, 0.01^2), \qquad d \in [1, D]$$

$$y_d^{(n)} \sim \mathcal{N}(\mu^{(n)}, 0.1^2), \qquad d \in [1, D]$$

That is, we generate means $\mu^{(n)}$ from a uniform distribution, and then generate our inputs and outputs, which are normally distributed, based on $\mu^{(n)}$.

- **Based on the problem set up, how could we predict $\mu^{(n)}$ from $x_d^{(n)}$s?**
  - the x_d are normally distributed, with low variance
  - we should expect that averaging these observations should give us a good estimate for μ
- **How can we then predict y_d from $x_d^{(n)}$?**
  - the y_d are also normally distributed, with the same mean as the inputs
  - hence, we can predict y_d by averaging the x_d
- **What weights should we expect for a model?**

- the weight $w_d = \frac{1}{D}$ would ensure that the linear combination of features returns the average μ

- **What will a linear regression model predict for the weights?**
  - if we run this through Python, we however get very **large** weights

```
import numpy as np

N,D = 15,10
mu = np.random.rand(N)
X = np.tile(mu[:,None], (1, D)) + 0.01*np.random.randn(N, D)
yy = 0.1*np.random.randn(N) + mu

w_pred = np.linalg.lstsq(X,yy,rcond = None)[0]
```

```
Weights: [3.87863335, 3.12722403, -4.56879925,
        11.38995375, 4.01304845, 6.47978537,
        -4.71122485, -14.51761502, -3.07998668,
        -0.91482638]
Average of weights: 0.10961927700012794
Expected average weight: 0.1
```

  - notice, the average of the weights **is** around $\frac{1}{D} = 0.1$
  - however, the weights are **meaningless**: they don't tell us anything about the importance of a given feature, they are just made to follow the training examples

- **Why do the weights become so large when overfitting?**
  - overfitting leads to learning noise
  - by using **large** weights, the model can more easily fit to the noise, by exploiting small differences in the input features
  - overall, this means that the model will generalise **badly**, especially if given input data generated with difference variance, but same mean (i.e $\mathcal{N}(\mu, 1^2)$)

4.2.3 Overfitting Example: Many Basis Functions

We consider training data generated stochastically:

$$x_d^{(n)} \sim Uniform[0,1], \qquad d \in [1, D]$$

$$y^{(n)} \sim \mathcal{N}(0,1)$$

Notice, the inputs have no relationship whatsoever to the outputs.

- **Based on the problem set up, how could we predict the output?**
  - inputs and outputs are unrelated
  - best bet is to define:

$$f(\underline{x}) = 0$$

    since the outputs are normally distributed about 1

- on average, we expect the average square error to be 1

- **What happens if we use many RBF basis functions?**
  - the model will overfit, and probably obtain a lower MSE than 1
  - in doing this, the model will probably learn extreme weights, to account for the unrelatedness

- **How would the model perform if we feed it new data?**
  - the extreme weights will make predictions for new data extreme and unreasonable

4.3 Regularisation

- **What is the aim of regularisation?**
  - create a more **robust** model, by enforcing that **weights** should be small
  - for example, L_2 **regularisation/ridge regression** penalises the **sum** of the **square** weights

- **How does regularisation affect the error function?**
  - we modify the error function:

$$\begin{aligned} E_\lambda(\underline{w}, \underline{y}, \Phi) &= \sum_{n=1}^{N} \left(y^{(n)} - f(\underline{x}^{(n)}; \underline{w}) \right)^2 + \lambda \sum_{k=1}^{K} w_k^2, \qquad \lambda \in \mathbb{R} \\ &= (\underline{y} - \Phi\underline{w})^T (\underline{y} - \Phi\underline{w}) + \lambda \underline{w}^T \underline{w} \end{aligned}$$

  - λ is a **regularisation parameter**:
    - as $\lambda \to 0$, we just prioritise fitting to the data
    - as $\lambda \to \infty$, we prioritise small weights, so $\|\underline{w}\| \to 0$
    - that is, the larger the λ, we reduce the priority of fit accuracy

- **How are the weights fit with regularisation?**
  - one could use **calculus** to figure out the optimal weights
  - however, we can reuse the least-squares fitting we have been using
  - we create K "dummy" observations (where K is the number of basis functions):

$$\tilde{\underline{y}} = \begin{pmatrix} \underline{y} \\ --- \\ \underline{0}_K \in \mathbb{R}^K \end{pmatrix} \qquad \tilde{\Phi} = \begin{pmatrix} \Phi \\ ------ \\ \sqrt{\lambda}\mathbb{I}_K \in \mathbb{R}^{K \times K} \end{pmatrix}$$

  - then:

$$\begin{aligned} E(\underline{w}; \tilde{\underline{y}}, \tilde{\Phi}) &= (\tilde{\underline{y}} - \tilde{\Phi}\underline{w})^T (\tilde{\underline{y}} - \tilde{\Phi}\underline{w}) \\ &= \begin{pmatrix} \underline{y} - \Phi\underline{w} \\ ----- \\ \underline{0}_K - \sqrt{\lambda}\mathbb{I}_K\underline{w} \end{pmatrix}^T \begin{pmatrix} \underline{y} - \Phi\underline{w} \\ ----- \\ \underline{0}_K - \sqrt{\lambda}\mathbb{I}_K\underline{w} \end{pmatrix} \\ &= \begin{pmatrix} \underline{y} - \Phi\underline{w} \\ ---- \\ -\sqrt{\lambda}\underline{w} \end{pmatrix}^T \begin{pmatrix} \underline{y} - \Phi\underline{w} \\ ---- \\ -\sqrt{\lambda}\underline{w} \end{pmatrix} \\ &= (\underline{y} - \Phi\underline{w})^T (\underline{y} - \Phi\underline{w}) + \lambda \underline{w}^T \underline{w} \end{aligned}$$

    as required

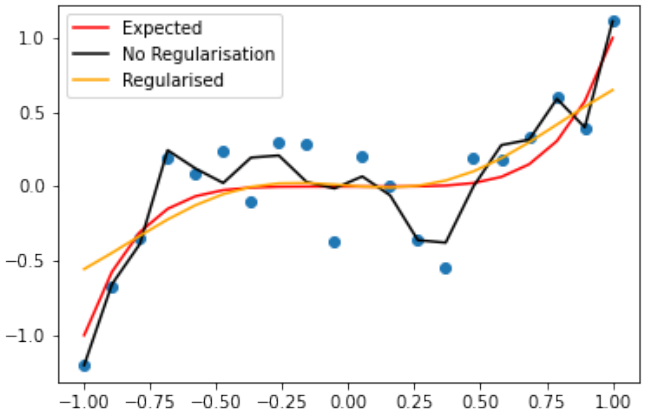


Figure 1: I generated data using a cubic $y = x^5, x \in [-1, 1]$, and added noise to it. Then, I fitted 20 RBFs, evenly distributed over the domain. This used $\lambda = 0.1$ for regularisation. Without regularisation, the model (black line) clearly follows the noise. On the other hand, the regularised model (orange line) is much closer to the underlying distribution of the data (red line).

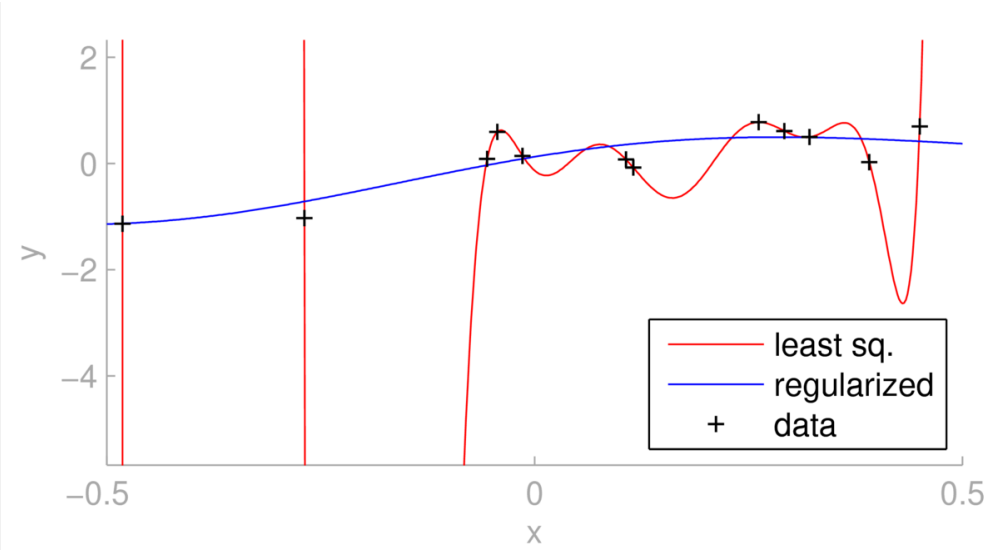


Figure 2: Extreme example of overfitting, and how regularisation hinders the effect of noise.

5 Class & Note Questions

5.1 Note Questions

- **An RBF basis function is centred at some position $\underline{c}$. We can create multiple columns of Φ by evaluating RBFs with different centres, from a set $\{\underline{c}^{(k)}\}$. For one-dimensional curve fitting, why does it not make sense to create a family of quadratic functions $\phi_k(x) = (x - c^{(k)})^2$, and include many features with different centres $c^{(k)}$?**
  - linear combinations of RBFs allow us to create a flexible family of functions

- however, if we use a quadratic basis, our model will be a linear combination of quadratics, which in itself will be a quadratic:

$$\sum_{k=1}^{K} w_k (x - c^{(k)})^2 = ax^2 + bx + c$$

- thus, at most we would just need $K = 3$ basis functions
- we have that:

$$a = \sum_k w_k \qquad b = -2 \sum_k w_k c^{(k)} \qquad c = \sum_k w_k (c^{(k)})^2$$

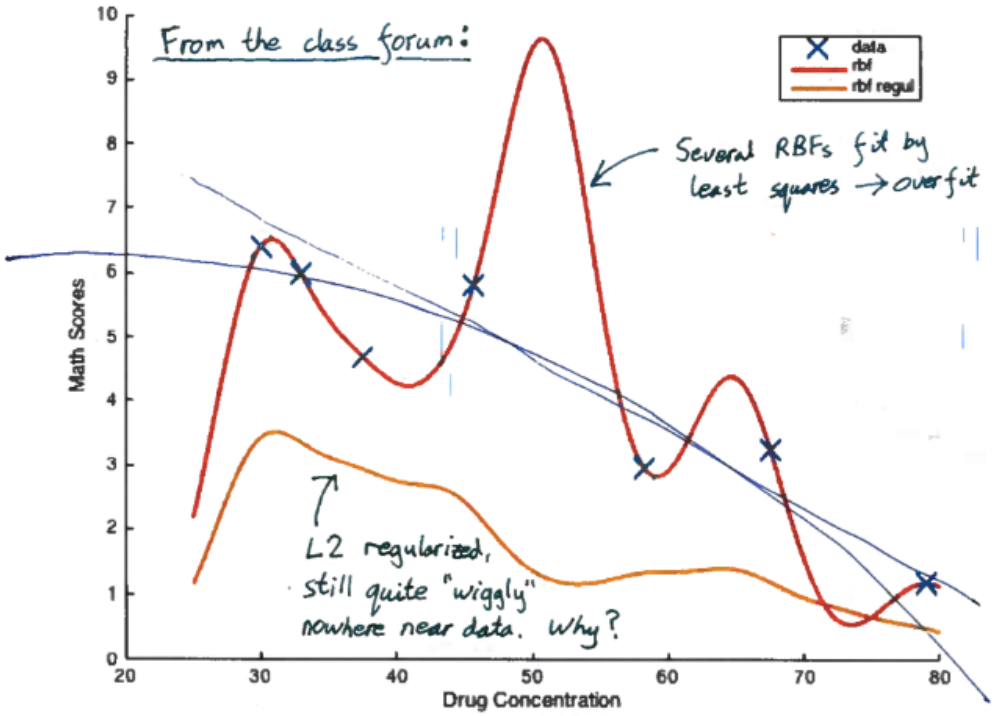
- **Why would it be a bad idea to pick the value of λ which minimises the cost function $E_\lambda(\underline{w}; \underline{y}, \Phi)$?**
  - by letting $\lambda \to -\infty$, the cost can be made arbitrarily low, without worrying about the weights or their magnitude
  - if $\lambda \geq 0$, if we set $\lambda = 0$, we attain the minimum cost, since the model will have the best possible fit (but this neglects the regularisation objective)

5.2 Class Questions

- **Would it be a good idea to fit a linear model to predict sunflower growth from water use?**
  - no; sunflower growth probably has a saturation point, after which adding more water negatively impacts plant growth
  - a linear model (without basis functions) would assume a positive correlation between the 2
  - also doesn't account for feature interactions: for example, increasing temperature might make the effect of water on plant growth decrease disproportionately
- **Why do we use a square in least squares fitting?**
  - emphasises **large** errors
  - it enforces that the sum of errors is always non-negative:
    - if we allowed for negative error:

$$\sum_{i=1}^{N} (y_i - \underline{w}^T \underline{x})$$

      we could just let $\|\underline{w}\| \to \infty$ to make $\underline{w}^T \underline{x}$ as large as possible, so that the error gets minimised and tends to $-\infty$; clearly, this would have no predictive power
  - it is a computationally convenient function
  - alternatively, we could use $\sum_{n=1}^{N} |y_n - f_n|$, which places less emphasis on errors, but is harder to fit
  - **We fit a model with RBFs, and then fit another one with regularisation. However, the regularised model is nowhere near the actual data. The regularisation parameter was changed: as $\lambda \to 0$, the model overfitted. As $\lambda \to \infty$, the model just flattened, and didn't come near the points. What may have caused this?**



* perhaps there were too many RBFs, very narrowly distributed?
* perhaps the RBFs were centered on the datapoints themselves, leading to quick overfitting?
* to fix, could think about adding a bias feature (without regularisation) to shift the curve up

6 Tutorial

1. **Alice fits a function $f(\underline{x}) = \underline{w}^T\underline{x}$ to a training set of N datapoints, The inputs $\underline{x}$ are D-dimensional column vectors. you can assume a unique setting of the weights $\underline{w}$ minimises the square error on the training set. Bob has heard that by transforming the inputs $\underline{x}$ with a vector valued function ϕ he can fit an alternative function $g(\underline{x}) = \underline{v}^T\phi(\underline{x})$, with the same fitting code. He decides to use a linear transformation $\phi(\underline{x}) = A\underline{x}$, where A is an invertible matrix.**

   (a) **Show that Bob's procedure will fit the same function as Alice's original procedure.**
   - Alice fits an optimal set of weights $\underline{w}$ by assumption
   - moreover, this function is a **linear combination** of $\underline{x}$
   - since Bob's function is a matrix, $\underline{v}^T A$ also produces linear combinations of $\underline{x}$
   - since Bob and Alice are using the same loss function to fit the weights, and Alice has optimal weights, Bob will find $\underline{v}$, such that:
   $$\underline{v}^T A = \underline{w}^T$$
   - hence, they fit the same function

   (b) **Could Bob's procedure be better than Alice's if the matrix is not invertible? (here, better is ambiguous on purpose)**
   - if A is not-invertible, then it won't have full-rank (i.e square with repeated rows/columns, or rectangular with more columns than rows)
   - transformations by A have the effect of mapping points to a lower dimensional subspace, so it is possible for 2 distinct points to get mapped to the same value:
   $$\underline{x} \neq \underline{x}' \implies A\underline{x} = A\underline{x}'$$

- hence, Bob's function now only generates a subset of the linear combinations which Alice's can generate; in particular, Bob's training error can't be better than Alice's
- **however**, training error is **meaningless** when deciding which model is better
- on the one hand, if Alice's fit is good and justified, Bob's function will underperform
- however, with high dimensionality $\underline{x}$, or small N, Alice's function is more likely to overfit, whilst Bob's might generalise better
- moreover, Bob's function has the effect of reducing the dimensionality of the input, reducing time and memory requirements for least squares fit
- if A is rectangular (but more rows than columns), then we increase the dimensionality of the data, so the linear combinations found by Bob will match Alice's

(c) **Alice becomes worried about overfitting, adds a regulariser $\lambda \underline{w}^T \underline{w}$ to the least-squares error function, and refits the model. Assuming A is invertible, can Bob choose a regulariser so that he will still always obtain the same function as Alice?**

- we have from above that with invertible A:

$$\underline{v}^T A = \underline{w}^T$$

- hence, for Bob to regularise he just needs to introduce:

$$\lambda \underline{w}^T \underline{w} = \lambda (\underline{v}^T A)(\underline{v}^T A)^T = \lambda \underline{v}^T A A^T \underline{v}$$

into his loss

(d) **Suppose we wish to find the vector $\underline{v}$ that minimises the function:**

$$(\underline{y} - \Phi \underline{v})^T (\underline{y} - \Phi \underline{v}) + \underline{v}^T M \underline{v}$$

i. **Show that:**

$$\underline{v}^T M \underline{v} = \underline{v}^T \left(\frac{1}{2} M + \frac{1}{2} M^T \right) \underline{v}$$

and hence that we can assume without loss of generality that M is symmetric.

$$\begin{aligned} \underline{v}^T \left(\frac{1}{2} M + \frac{1}{2} M^T \right) \underline{v} &= \frac{1}{2} \left(\underline{v}^T M \underline{v} + \underline{v}^T M^T \underline{v} \right) \\ &= \frac{1}{2} \left(\underline{v}^T M \underline{v} + (Mv)^T \underline{v} \right) \\ &= \frac{1}{2} \left(\underline{v}^T M \underline{v} + \underline{v}^T M \underline{v} \right) \\ &= \underline{v}^T M \underline{v} \end{aligned}$$

ii. **Why would we usually choose M to be positive semi-definite in a regulariser, meaning that $\underline{a}^T M \underline{a} \geq 0$ for all vectors $\underline{a}$?**

- we add a regularisation term to avoid fitting weights which are large
- if M is not positive semi-definite, then $\exists \underline{a}$ such that:

$$\underline{a}^T M \underline{a} < 0$$

- in particular, the weight αa, with $\alpha \to \infty$ means that the loss will tend to $-\infty$
- in other words, the minimum cost could just be assigned to some very large weight

iii. **Assume that we can find a factorisation $M = AA^T$. Can we minimise the function above using a standard routine that can minimise:**

$$(\underline{z} - X \underline{w})^T (\underline{z} - X \underline{w})$$

with respect to $\underline{w}$?

- since M decomposes, we could write:

$$\underline{v}^T M \underline{v} = (A^T \underline{v})^T (A^T \underline{v})$$

- we can then use the data augmentation trick:

$$\tilde{\underline{y}} = \begin{pmatrix} \underline{y} \\ --- \\ \underline{0}_K \in \mathbb{R}^K \end{pmatrix} \qquad \tilde{X} = \begin{pmatrix} X \\ ------ \\ \sqrt{\lambda} A^T \end{pmatrix}$$

and seek to minimise:

$$(\tilde{\underline{y}} - \tilde{X}\underline{v})^T (\tilde{\underline{y}} - \tilde{X}\underline{v})$$

2. **We consider applying linear regression to one-dimensional inputs:**

$$f(x) = \underline{w}^T \phi(x; h)$$

where $\phi(x; h)$ evaluates the input at 101 radial basis functions:

$$\phi_k(x) = \exp\left(-\frac{(x - c_k)^2}{h^2}\right)$$

where h is a user-specified bandwidth, and:

$$c_k = \frac{h(k - 51)}{\sqrt{2}}$$

We fit this model for $N = 70$ observations, where:

$$x \in [-1, 1] \qquad y \in [-1, 1]$$

by using regularised least squares:

$$C = (\underline{y} - \Phi\underline{w})^T (\underline{y} - \Phi\underline{w}) + 0.1\underline{w}^T \underline{w}$$

(a) **Explain why many of the weights will be close to zero when $h = 0.2$, and why even more weights will probably be close to zero when $h = 1$.**

- since $k \in [1, 101]$, the values of c_k are nearly always outside of the range $[-1, 1]$, so $\phi_k(x) \approx 0$ most of the time
- since we are regularising, weights can't be huge, and so, most of the $\phi_k(x)$ will have zero weights (they are so far away from the data that they don't affect prediction part of the loss, so the loss will seek to minimise the regularisation term)
- if we increase h to $h = 1$, c_k will become even larger, so even less of the centres c_k will be close to x, and so, even more of the $\phi_k(x)$ will have close-to-zero weights

(b) **It is suggested that we could choose h by fitting $\underline{w}$ for each h in a grid of values, and pick the h which led to a fit with the smallest cost C. Explain whether this suggestion is a good idea, or whether you would modify the procedure.**

- by the discussion above, we expect that small h will tend to fare better (they will have more activations with non-zero weights, so can more easily fit to the data)
- however, there is risk of overfitting and bad generalisation: we risk selecting very narrow RBFs, and we only have 70 training points (but 101 RBFs)
- to select h, we should use a validation set (since very little data, perhaps using k-fold cross validation)

- alternatively, if we have experience, we can set h by hand
- or try finding more data

(c) **Another data set with inputs $x \in [-1, 1]$ arrives, but now you notice that all of the observed outputs are larger:**

$$y \in [1000, 1010]$$

What problem would we encounter if we performed linear regression as above to this data? How could this problem be fixed?

- the $\phi_k(x)$ will always output values between 0 and 1
- however, the labels have values in the 1000s; this means that the model will try to learn massive weights
- but due to regularisation, we will fit a model which will tend to predict lower values
- removing regularisation could lead to overfitting
- to solve this:
  1. **Scaling**: we can apply:
  $$\frac{1005 - y}{5}$$
  to set the labels to the range $[-1, 1]$ again (we would then need to scale future predictions)
  2. **Add Bias**: we can add a bias term $\phi_{102}(x) = 1$ (but don't include it in regularisation, since this term will need to have a large weight to fit well to the data - likely it will take on a weight of around 1005)

3. **Sketch the contours of the sigmoid:**

$$\phi(\underline{x}) = \sigma(\underline{v}^T \underline{x} + b)$$

where:

$$\underline{v} = (1, 2) \qquad b = 5$$

- the contours of a sigmoid are straight lines
- for example, if $\underline{x}$ lies in the decision boundary ($\phi(\underline{x}) = 0.5$), this implies that:

$$x_1 + 2x_2 + 5 = 0 \implies x_1 = -2x_2 - 5$$

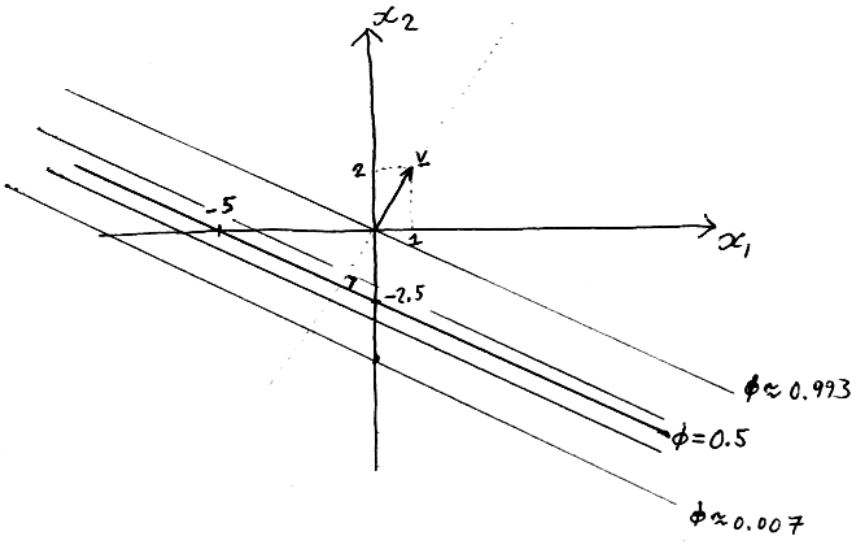
so:

$$\underline{x} = \begin{pmatrix} -2x_2 - 5 \\ x_2 \end{pmatrix}$$

This corresponds to lines going through $(-5, 0)$, with gradient $-\frac{1}{2}$ (not surprisingly, $\underline{x}$ is orthogonal to $\underline{v}$)

- other contours will be parallel to the decision boundary (since they must all be orthogonal to $\underline{v}$), and distributed symmetrically. For example, the contour passing through $(0, 0)$ will have:

$$\phi(\underline{x}) = \frac{1}{1 + \exp(-5)} \approx 0.993$$



- if we generalise this, for higher dimensions the contours will be hyperplanes, orthogonal to $\underline{v}$