Lecture 16: Kernel Ridge Regression

- Previously, in our discussion on polynomial regression, we had shown how could map the input variables to a new feature space of polynomials

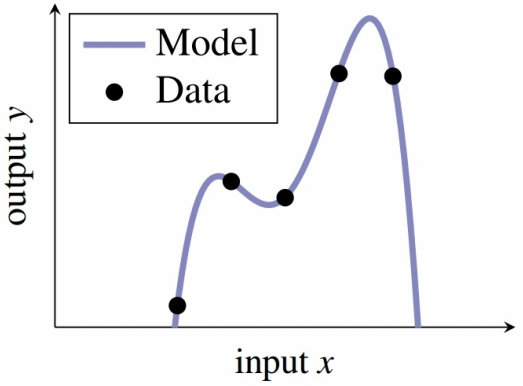
$$y = a_0 + a_1 x + \epsilon \xrightarrow{\text{Mapping}} y = b_0 + b_1 x + b_2 x^2 + \cdots + b_p x^p + \epsilon$$

Original input space — New input space

- While we created these non-linear transformations of the original input, we were still using linear regression, since the parameters $b_0, b_1, \cdots, b_p$ appear linearly with $\underline{\phi}(x) = \begin{bmatrix} 1 & x & x^2 & \cdots & x^p \end{bmatrix}^T$ as the new input

$$\boxed{y = \underline{\theta}^T \underline{\phi}(x) + \epsilon}$$

Linear regression with a 4th order polynomial



- For vector-valued input $\underline{x}$, the non-linear transformation could be expressed as

$$\underset{1\times 1}{y} = \underbrace{\underline{\phi}^T(\underline{x})}_{1\times d}\ \underset{d\times 1}{\underline{\theta}} + \underset{1\times 1}{\epsilon}$$

$$\underline{x} \in \mathbb{R}^p$$

$$\underline{\phi}(\underline{x}) \in \mathbb{R}^d$$

$$\underline{\theta} \in \mathbb{R}^d$$

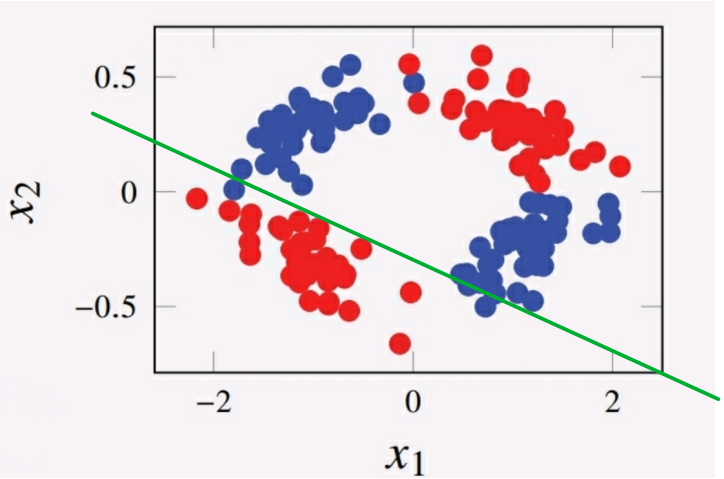
- Any choice of nonlinear transformation $\underline{\phi}(\underline{x})$ can be used!

- Writing the vectorized linear regression for training data $\{\underline{x}^{(i)}, y^{(i)}\}_{i=1}^N$

$$\underline{\underline{X}} = \underset{N\times p}{\begin{bmatrix} — \ \underline{x}^{(1)T} \ — \\ — \ \underline{x}^{(2)T} \ — \\ \vdots \\ — \ \underline{x}^{(p)T} \ — \end{bmatrix}}, \qquad \underline{\underline{\Phi}}(\underline{\underline{X}}) = \underset{N\times d}{\begin{bmatrix} — \ \underline{\phi}(\underline{x}^{(1)})^T \ — \\ — \ \underline{\phi}(\underline{x}^{(2)})^T \ — \\ \vdots \\ — \ \underline{\phi}(\underline{x}^{(N)})^T \ — \end{bmatrix}}, \qquad \underline{y} = \underset{N\times 1}{\begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(N)} \end{bmatrix}}$$

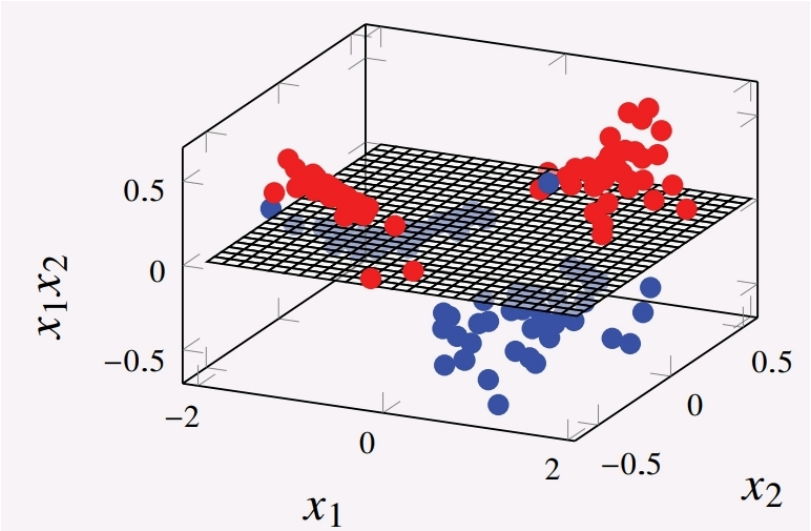
$$\underline{y} = \underline{\underline{\Phi}}(\underline{\underline{X}})\ \underline{\theta} + \underline{\epsilon}$$

Example of non-linear feature transformation for classification



A linear classifier would not work on the original input space

(there is no line that can separate the two classes)



With an introduction of an extra feature $x_1 x_2$ the problem becomes linearly separable

A carefully engineered transformation $\underline{\phi}(x)$ in linear regression or linear classification may perform very well for a specific ML problem

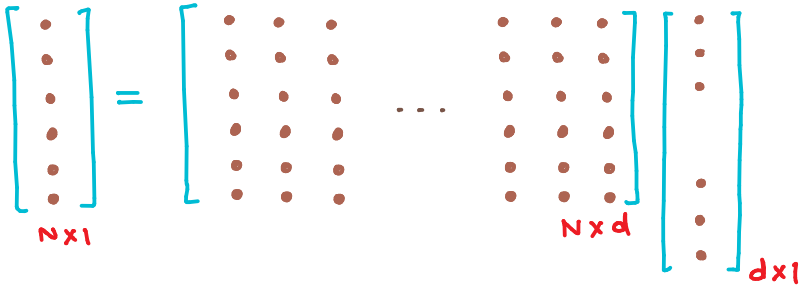
- We would like a $\underline{\phi}(\underline{x})$ that would work for most problems

- Thus, $\underline{\phi}(\underline{x})$ should contain a lot of transformations that could possibly be of interest to most problems

- Therefore, we should choose d, the dimension of $\underline{\phi}(\underline{x})$, really large
  - $d > N$ and eventually let $d \to \infty$

d: # of input features

N: # of training data-points



- However, increasing the flexibility of a model also means it can overfit the training data

- We will have to use some kind of regularization to prevent overfitting

- Let us use an L_2-regularization for now

- Reformulating the linear regression with transformed features $\underline{\phi}(\underline{x})$, we get the estimate of parameters as

$$\hat{\underline{\theta}} = \underset{\underline{\theta}}{\arg\min}\ J(\underline{\theta})$$

$$= \underset{\underline{\theta}}{\text{argmin}} \left[\frac{1}{N} \sum_{i=1}^{N} \left(y^{(i)} - \underline{\phi}(\underline{x}^{(i)})^T \underline{\theta} \right) + \lambda \| \underline{\theta} \|_2^2 \right]$$

- Linear regression with L_2-regularization has closed-form solution

$$\hat{\underline{\theta}} = \left(\underline{\underline{X}}^T \underline{\underline{X}} + N\lambda \underline{\underline{I}} \right)^{-1} \underline{\underline{X}}^T \underline{y} \quad \text{(recall !)}$$

$$\hat{\underline{\theta}} = \left(\underline{\underline{\Phi}}(\underline{\underline{X}})^T \underline{\underline{\Phi}}(\underline{\underline{X}}) + N\lambda \underline{\underline{I}} \right)^{-1} \underline{\underline{\Phi}}(\underline{\underline{X}})^T \underline{y}$$

- Linear regression with L_2-regularization has closed-form solution

$$\hat{\underline{\theta}} = \left(\underline{\underline{\Phi}}(\underline{x})^T \underline{\underline{\Phi}}(\underline{x}) + N\lambda \underline{\underline{I}} \right)^{-1} \underline{\underline{\Phi}}(\underline{x})^T \underline{y}$$

- The downside of choosing a very large number of features, d, is that we also have to learn 'd'-parameters and store them

- During prediction, we would use the d-dimensional parameter vector $\hat{\underline{\theta}}$

$$\hat{y}(\underline{x}^*) = \underset{1 \times d}{\underline{\phi}(x^*)^T} \underset{d \times 1}{\hat{\underline{\theta}}}$$

- But if $d \to \infty$, how to scale computations or meet storage demands ??

- Let's try to reformulate the prediction

$$\hat{y}(\underline{x}^*) = \underset{1\times d}{\underline{\phi}(\underline{x}^*)^T} \underset{d\times 1}{\hat{\underline{\theta}}} = \hat{\underline{\theta}}^T \underline{\phi}(\underline{x}^*)$$

$$= \left[\left(\underline{\underline{\Phi}}(\underline{\underline{x}})^T \underline{\underline{\Phi}}(\underline{\underline{x}}) + N\lambda \underline{\underline{I}} \right)^{-1} \underline{\underline{\Phi}}(\underline{\underline{x}})^T \underline{y} \right]^T \underline{\phi}(\underline{x}^*)$$

$$= \underbrace{\underbrace{\underline{y}^T}_{1\times n} \underbrace{\underline{\underline{\Phi}}(\underline{\underline{x}})}_{n\times d} \underbrace{\left(\underline{\underline{\Phi}}(\underline{\underline{x}})^T \underline{\underline{\Phi}}(\underline{\underline{x}}) + N\lambda \underline{\underline{I}} \right)^{-1}}_{d\times d} \underbrace{\underline{\phi}(\underline{x}^*)}_{d\times 1}}_{n\times 1}$$

This entire expression is independent of 'd' and if we could compute this n-dimensional vector directly, the it would be great!

- However, $\underbrace{\left(\underline{\underline{\Phi}}(\underline{\underline{x}})^T \underline{\underline{\Phi}}(\underline{\underline{x}}) + N\lambda \underline{\underline{I}} \right)^{-1}}_{d\times d}$ still requires inverting a d×d matrix!!

$$\hat{y}(\underline{x}^*) = \underline{y}^T \underline{\underline{\Phi}}(\underline{x}) \left(\underbrace{\underline{\underline{\Phi}}(\underline{x})^T \underline{\underline{\Phi}}(\underline{x})}_{d \times d} + N\lambda \underline{\underline{I}} \right)^{-1} \underline{\phi}(\underline{x}^*)$$

- To prevent inverting a $d \times d$ matrix, where d is very large, lets use a matrix identity $\underline{\underline{A}} \left(\underline{\underline{A}}^T \underline{\underline{A}} + \underline{\underline{I}} \right)^{-1} = \left(\underline{\underline{A}} \underline{\underline{A}}^T + \underline{\underline{I}} \right)^{-1} \underline{\underline{A}}$

$$\hat{y}(\underline{x}^*) = \underbrace{\underline{y}^T}_{1 \times N} \underbrace{\left(\underline{\underline{\Phi}}(\underline{x}) \, \underline{\underline{\Phi}}(\underline{x})^T + N\lambda \underline{\underline{I}} \right)^{-1}}_{N \times N} \underbrace{\underline{\underline{\Phi}}(\underline{x}) \, \underline{\phi}(\underline{x}^*)}_{N \times 1}$$

- We can now compute $\hat{y}(\underline{x}^*)$ without having to deal with any d-dimensional vectors or matrices, if we can compute $\underline{\underline{\Phi}}(\underline{x}) \, \underline{\underline{\Phi}}(\underline{x})^T$ & $\underline{\underline{\Phi}}(\underline{x}) \, \underline{\phi}(\underline{x}^*)$

$$\underset{N \times d}{\underline{\underline{\Phi}}(\underline{x})} = \begin{bmatrix} \text{——} \; \underline{\phi}(\underline{x}^{(1)})^T \; \text{——} \\ \text{——} \; \underline{\phi}(\underline{x}^{(2)})^T \; \text{——} \\ \vdots \\ \text{——} \; \underline{\phi}(\underline{x}^{(N)})^T \; \text{——} \end{bmatrix}$$

$$\underset{d \times N}{\underline{\underline{\Phi}}(\underline{x})^T} = \begin{bmatrix} | & & | \\ \underline{\phi}(\underline{x}^{(1)}) & \cdots & \underline{\phi}(\underline{x}^{(N)}) \\ | & & | \end{bmatrix}$$

$$\hat{y}(\underline{x}^*) = \underbrace{\underline{y}^T}_{1\times N}\underbrace{\left(\underline{\underline{\Phi}}(\underline{\underline{x}})\,\underline{\underline{\Phi}}(\underline{\underline{x}})^T + N\lambda\underline{\underline{I}}\right)^{-1}}_{N\times N}\underbrace{\underline{\underline{\Phi}}(\underline{\underline{x}})\,\underline{\phi}(\underline{x}^*)}_{N\times 1}$$

- Lets look at the two matrix multiplications

$$\underbrace{\underline{\underline{\Phi}}(\underline{\underline{x}})\,\underline{\underline{\Phi}}(\underline{\underline{x}})^T}_{\substack{N\times d \quad d\times N \\ N\times N}} = \begin{bmatrix} \underline{\phi}(\underline{x}^{(1)})^T\underline{\phi}(\underline{x}^{(1)}) & \underline{\phi}(\underline{x}^{(1)})^T\underline{\phi}(\underline{x}^{(2)}) & \cdots & \underline{\phi}(\underline{x}^{(1)})^T\underline{\phi}(\underline{x}^{(N)}) \\ \underline{\phi}(\underline{x}^{(2)})^T\underline{\phi}(\underline{x}^{(1)}) & \underline{\phi}(\underline{x}^{(2)})^T\underline{\phi}(\underline{x}^{(2)}) & \cdots & \underline{\phi}(\underline{x}^{(2)})^T\underline{\phi}(\underline{x}^{(N)}) \\ \vdots & \vdots & \ddots & \vdots \\ \underline{\phi}(\underline{x}^{(N)})^T\underline{\phi}(\underline{x}^{(1)}) & \underline{\phi}(\underline{x}^{(N)})^T\underline{\phi}(\underline{x}^{(2)}) & \cdots & \underline{\phi}(\underline{x}^{(N)})^T\underline{\phi}(\underline{x}^{(N)}) \end{bmatrix}$$

$$\underbrace{\underline{\underline{\Phi}}(\underline{\underline{x}})\,\underline{\phi}(\underline{x}^*)}_{\substack{N\times d \quad d\times 1 \\ N\times 1}} = \begin{bmatrix} \underline{\phi}(\underline{x}^{(1)})^T\underline{\phi}(\underline{x}^*) \\ \underline{\phi}(\underline{x}^{(2)})^T\underline{\phi}(\underline{x}^*) \\ \vdots \\ \underline{\phi}(\underline{x}^{(N)})^T\underline{\phi}(\underline{x}^*) \end{bmatrix}$$

- $\underline{\phi}(\underline{x})^T\underline{\phi}(\underline{x}')$ is an inner product between two d-dimensional vectors $\underline{\phi}(\underline{x})$ & $\underline{\phi}(\underline{x}')$
- $\underline{\phi}(\underline{x})$ enters the prediction $y(\underline{x}^*)$ only as these inner products

Lets take an example of polynomial transformation

$\underset{1\times1}{x}$ ← scalar $(p=1)$

$\phi(x)$ is say a third-order scaled polynomial of the form:

$$\underset{4\times1}{\begin{bmatrix} 1 \\ \sqrt{3}\,x \\ \sqrt{3}\,x^2 \\ x^3 \end{bmatrix}}$$

$$\underline{\phi}(x)^T \underline{\phi}(x') = \begin{bmatrix} 1 & \sqrt{3}x & \sqrt{3}x^2 & x^3 \end{bmatrix} \begin{bmatrix} 1 \\ \sqrt{3}\,x' \\ \sqrt{3}\,x'^2 \\ x'^3 \end{bmatrix}$$

$$= 1 + 3xx' + 3x^2x'^2 + x^3x'^3 = (1+xx')^3$$

In general, if $\underline{\phi}(x)$ is a suitably scaled polynomial of order 'd', then

$$\underline{\phi}(\underline{x})^T \phi(\underline{x}') = (1+xx')^d$$

inner product

- Usually to compute $\underline{\phi}(\underline{x})^T \underline{\phi}(\underline{x}')$
  - One has to first d-dimensional vectors $\underline{\phi}(\underline{x})$ and $\underline{\phi}(\underline{x}')$, and
  - then compute their inner product

- However, for the previous example, we found that we could have just evaluated the expression $(1 + x\,x')^d$ directly

- Important point: If we make the choice of $\underline{\phi}(\underline{x})$ s.t. the inner product $\underline{\phi}(\underline{x})^T \underline{\phi}(\underline{x}')$ can be computed without first calculating $\underline{\phi}(\underline{x})$, we can let $d \rightarrow$ very large

- Important point: If we make the choice of $\underline{\phi}(\underline{x})$ s.t. the inner product $\underline{\phi}(\underline{x})^T \underline{\phi}(\underline{x}')$ can be computed without first calculating $\underline{\phi}(\underline{x})$, we can let $d \rightarrow$ very large

- This might appear to be rather restrictive, since it seems that, for each case, we may have to derive a closed-form analytical form of $\underline{\phi}(\underline{x})^T \underline{\phi}(\underline{x})$, just like we got for polynomial transformation

e.g. $\underline{\phi}(x)\,\underline{\phi}(x') = \underbrace{(1 + xx')^d}$

Closed-form expression

- However, if you don't really care about $\underline{\phi}(\underline{x})$ explicitly sometimes, then the need of deriving $\underline{\phi}(\underline{x})^T \underline{\phi}(\underline{x}')$ can be bypassed by using the concept of kernels

Introducing the idea of kernels

- In simple terms, a kernel $K(\underline{x}, \underline{x}')$ (kappa) is any function that takes two arguments $\underline{x}$ and $\underline{x}'$ from the same space $\mathbb{R}^p$ and returns a scalar

- We will mostly limit ourselves to kernels that are real-valued and symmetric

  i.e. $K(\underline{x}, \underline{x}') = K(\underline{x}', \underline{x}) \in \mathbb{R}$ for all $\underline{x}$ and $\underline{x}'$

  For example, $K(x, x') = (1 + xx')^d$ is such a kernel

- In fact, the inner product of two non-linear input transformation is also an example of a kernel:

$$K(\underline{x}, \underline{x}') = \underline{\phi}(\underline{x})^T \underline{\phi}(\underline{x}')$$

- So instead of choosing $\underline{\phi}(\underline{x})$ and deriving its inner product $\underline{\phi}(\underline{x})^T \underline{\phi}(\underline{x}')$ sometimes one can choose a kernel $K(\underline{x}, \underline{x}')$ directly ← KERNEL TRICK

If $\underline{x}$ enters the model as $\underline{\phi}(\underline{x})^T \underline{\phi}(\underline{x}')$ only, we can choose a kernel $K(\underline{x}, \underline{x}')$ directly, instead of choosing $\underline{\phi}(\underline{x})$ KERNEL TRICK

- Mathematically, we can rewrite

$$\hat{y}(\underline{x}^*) = \underbrace{\underline{y}^T}_{1 \times N} \underbrace{\left(\underline{\underline{\Phi}}(\underline{x})\,\underline{\underline{\Phi}}(\underline{x})^T + N\lambda \underline{\underline{I}}\right)^{-1}}_{N \times N} \underbrace{\underline{\underline{\Phi}}(\underline{x})\,\underline{\phi}(\underline{x}^*)}_{N \times 1}$$

as

$$\hat{y}(\underline{x}^*) = \underline{y}^T \left(\underline{\underline{K}}(\underline{X}, \underline{X}) + N\lambda \underline{\underline{I}}\right)^{-1} \underline{\underline{K}}(\underline{X}, \underline{x}^*)$$

where

$$\underline{\underline{K}}(\underline{X}, \underline{X}) = \begin{bmatrix} K(\underline{x}^{(1)}, \underline{x}^{(1)}) & K(\underline{x}^{(1)}, \underline{x}^{(2)}) & \cdots & K(\underline{x}^{(1)}, \underline{x}^{(N)}) \\ K(\underline{x}^{(2)}, \underline{x}^{(1)}) & K(\underline{x}^{(2)}, \underline{x}^{(2)}) & \cdots & K(\underline{x}^{(2)}, \underline{x}^{(N)}) \\ \vdots & \vdots & \ddots & \vdots \\ K(\underline{x}^{(N)}, \underline{x}^{(1)}) & K(\underline{x}^{(N)}, \underline{x}^{(2)}) & \cdots & K(\underline{x}^{(N)}, \underline{x}^{(N)}) \end{bmatrix}, \quad \underline{\underline{K}}(\underline{X}, \underline{x}^*) = \begin{bmatrix} K(\underline{x}^{(1)}, \underline{x}^*) \\ K(\underline{x}^{(2)}, \underline{x}^*) \\ \vdots \\ K(\underline{x}^{(N)}, \underline{x}^*) \end{bmatrix}$$

- Recall, linear regression with L_2-regularization was called as ridge regression

- $$\hat{y}(\underline{x}^*) = \underline{y}^T \left(\underline{\underline{K}}(\underline{\underline{X}}, \underline{\underline{X}}) + N\lambda \underline{\underline{I}} \right)^{-1} \underline{\underline{K}}(\underline{\underline{X}}, \underline{x}^*)$$

  This equation describes linear regression with L_2-regularization using a kernel, hence is called kernel ridge regression

$\underline{\underline{K}}(\underline{\underline{X}}, \underline{\underline{X}})$ ⟵ Gram matrix (or Gramian matrix)

- The design choice is now to select a kernel $K(\underline{x}, \underline{x}')$ instead of $\underline{\phi}(\underline{x})$

- In practice, choosing $K(\underline{x}, \underline{x}')$ is much easier than choosing an appropriate $\underline{\phi}(\underline{x})$ especially when the number of transformed features (i.e. d) is very large

- From computation point of view, we can choose $K(\underline{x}, \underline{x}')$ arbitrarily, as long as we can compute

Gram matrix

$$\hat{y}(\underline{x}^*) = \underline{y}^T \left(\underline{\underline{K}}(\underline{\underline{X}}, \underline{\underline{X}}) + N\lambda \underline{\underline{I}} \right)^{-1} \underline{\underline{K}}(\underline{\underline{X}}, \underline{x}^*)$$

this must be invertible

- For the inverse $\left(\underline{\underline{K}}(\underline{\underline{X}}, \underline{\underline{X}}) + N\lambda \underline{\underline{I}} \right)^{-1}$ to exist, we will restrict ourselves to kernels for which the Gram matrix $\underline{\underline{K}}(\underline{\underline{X}}, \underline{\underline{X}})$ is always PSD

positive semi-definite

A matrix $\underline{\underline{M}}$ is said to be PSD if

- $\underline{v}^T \underline{\underline{M}} \underline{v} \geq 0$ for all $\underline{v}$
- equivalently, all eigenvalues of $\underline{\underline{M}} \geq 0$

- Kernels $K(\underline{x}, \underline{x}')$ that leads to a PSD $\underline{\underline{K}}(\underline{\underline{X}}, \underline{\underline{X}})$ are called PSD kernels

Examples of positive semi-definite kernels

- Squared exponential kernel (also known as radial basis function, RBF exponentiated quadratic, Gaussian kernel)

$$K(\underline{x}, \underline{x}') = \exp\left(-\frac{\|\underline{x} - \underline{x}'\|_2^2}{2\ell^2}\right)$$

where $\ell > 0$ is a hyperparameter to be chosen by the user (by cross-validation)

- Polynomial kernel

$$K(\underline{x}, \underline{x}') = (c + \underline{x}^T \underline{x}')^{d-1}$$

order of polynomial

- You will see more examples of symmetric PSD kernels later

$$\hat{y}(\underline{x}^*) = \underline{y}^T \underbrace{\left(\underline{\underline{K}}(\underline{\underline{X}}, \underline{\underline{X}}) + N\lambda \underline{\underline{I}} \right)^{-1}}_{N \times N \text{ matrix}} \underline{\underline{K}}(\underline{\underline{X}}, \underline{x}^*)$$

- Inversion of a high-dimensional matrix is a very heavy operation
- Do we need to invert the matrix $\left(\underline{\underline{K}}(\underline{\underline{X}}, \underline{\underline{X}}) + N\lambda \underline{\underline{I}} \right)$ every time we predict for a new test input $\underline{x}^*$?

  — Not necessary

- We can introduce an N-dimensional vector $\hat{\underline{\alpha}}$ ← (Dual parameter)

$$\hat{\underline{\alpha}} = \begin{bmatrix} \hat{\alpha}_1 \\ \hat{\alpha}_2 \\ \vdots \\ \hat{\alpha}_N \end{bmatrix} = \underline{y}^T \left(\underline{\underline{K}}(\underline{\underline{X}}, \underline{\underline{X}}) + N\lambda \underline{\underline{I}} \right)^{-1} \Rightarrow$$

Test prediction

$$\boxed{\hat{y}(\underline{x}^*) = \hat{\underline{\alpha}}^T \underline{\underline{K}}(\underline{\underline{X}}, \underline{x}^*)}$$

So now, we only need to compute and store $\hat{\underline{\alpha}}$ and $\underline{\underline{X}}$

Summary of Kernel Ridge Regression (KRR)

Training

Input: Training data $\mathcal{T} = \{\underline{x}^{(i)}, y^{(i)}\}_{i=1}^{N}$, a kernel K, regularization parameter λ

Output: Dual parameter $\hat{\underline{\alpha}}$

- Assemble $\underline{\underline{X}}$ and compute $\underline{\underline{K}}(\underline{\underline{X}}, \underline{\underline{X}})$
- Compute $\hat{\underline{\alpha}}$ as

$$\hat{\underline{\alpha}} = \underline{y}^{T}\left(\underline{\underline{K}}(\underline{\underline{X}}, \underline{\underline{X}}) + N\lambda\underline{\underline{I}}\right)^{-1}$$

Prediction with kernel ridge regression

Input: Learned dual parameter $\hat{\underline{\alpha}}$ and test input $\underline{x}^{*}$

Output: Prediction $\hat{y}(\underline{x}^{*}) = \hat{\underline{\alpha}}^{T}\underline{\underline{K}}(\underline{\underline{X}}, \underline{x}^{*})$

Primal vs Dual formulation

$$\hat{\underline{\theta}} = \left(\underline{\underline{\Phi}}(\underline{X})^T \underline{\underline{\Phi}}(\underline{X}) + N\lambda \underline{\underline{I}}\right)^{-1} \underline{\underline{\Phi}}(\underline{X})^T \underline{y}$$

$$\hat{y}(\underline{x}^*) = \underline{\phi}(\underline{x}^*)^T \hat{\underline{\theta}}$$

Primal formulation of linear regression

$$\hat{\underline{\alpha}} = \underline{y}^T \left(\underline{\underline{K}}(\underline{X}, \underline{X}) + N\lambda \underline{\underline{I}}\right)^{-1}$$

$$\hat{y}(\underline{x}^*) = \hat{\underline{\alpha}}^T \underline{\underline{K}}(\underline{X}, \underline{x}^*)$$

Dual formulation of linear regression

- $\hat{\underline{\theta}} \in \mathbb{R}^d$, $d \to \infty$
- $\hat{\underline{\alpha}} \in \mathbb{R}^N$, $N \to$ # of data pts (finite)

- By comparing the two formulation, we can find a relation between $\hat{\underline{\theta}}$ and $\hat{\underline{\alpha}}$

$$\hat{y}(\underline{x}^*) = \hat{\underline{\theta}}^T \underline{\phi}(\underline{x}^*) = \hat{\underline{\alpha}}^T \underbrace{\underline{\underline{\Phi}}(\underline{X}) \underline{\phi}(\underline{x}^*)}_{\underline{\underline{K}}(\underline{X}, \underline{x}^*)}$$

$$\Rightarrow \quad \underset{d\times 1}{\hat{\underline{\theta}}} = \underbrace{\underline{\underline{\Phi}}(\underline{X})^T}_{d\times N} \underset{N\times 1}{\hat{\underline{\alpha}}}$$

← this is a general result of Representer theorem

Simplied version of Representer's Theorem

Theorem: Let $\hat{y}(\underline{x}) = \underline{\theta}^T \underline{\phi}(\underline{x})$ with a fixed nonlinear transform $\underline{\phi}(\underline{x})$, with $\underline{\theta}$ learned from training data $\{\underline{x}^{(i)}, y^{(i)}\}_{i=1}^N$

(The dimensionality of $\underline{\theta}$ and $\underline{\phi}(\underline{x})$ need not be finite)

Furthermore, let $L(y, \hat{y})$ be any arbitrary loss function & $h : [0, \infty] \longrightarrow \mathbb{R}$ be a strictly monotonically increasing function

Then, the estimate $\hat{\underline{\theta}}$ which is the argmin of the cost function $J(\theta)$, i.e.

$$\hat{\underline{\theta}} = \underset{\underline{\theta}}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} L\Big(y^{(i)}, \underbrace{\underline{\theta}^T \underline{\phi}(\underline{x}^{(i)})}_{\hat{y}(\underline{x}^{(i)})}\Big) + h\left(\|\underline{\theta}\|_2^2\right)$$

can be written as

$$\hat{\underline{\theta}} = \underline{\underline{\Phi}}(\underline{\underline{X}})^T \underline{\alpha}, \text{ with some } N\text{-dimensional vector } \underline{\alpha}$$

- What does the representer theorem mean?

  - It suggests that if $\hat{y}(\underline{x}) = \underline{\theta}^T \underline{\phi}(\underline{x})$, and $\underline{\theta}$ is to be learned using any loss function and L_2-regularization, then $\hat{\underline{\theta}}$ can be learned also from its dual parameter $\hat{\underline{\alpha}}$, using: $\hat{\underline{\theta}} = \underline{\underline{\Phi}}(\underline{x})\,\hat{\underline{\alpha}}$

  - An important implication of the representer theorem is that L_2-regularization is crucial in order to obtain the dual formalism, and we could not have obtained KRR with say L_1-regularization

  - Representer theorem is very important for most kernel methods. It tell us that we can express some models in terms of dual parameters $\underline{\alpha}$ which are of finite length N, and a kernel $K(\underline{x}, \underline{x}')$, instead of the primal parameters $\underline{\theta}$ (maybe of infinite length d) and a $\underline{\phi}(\underline{x})$

Support Vector Regression

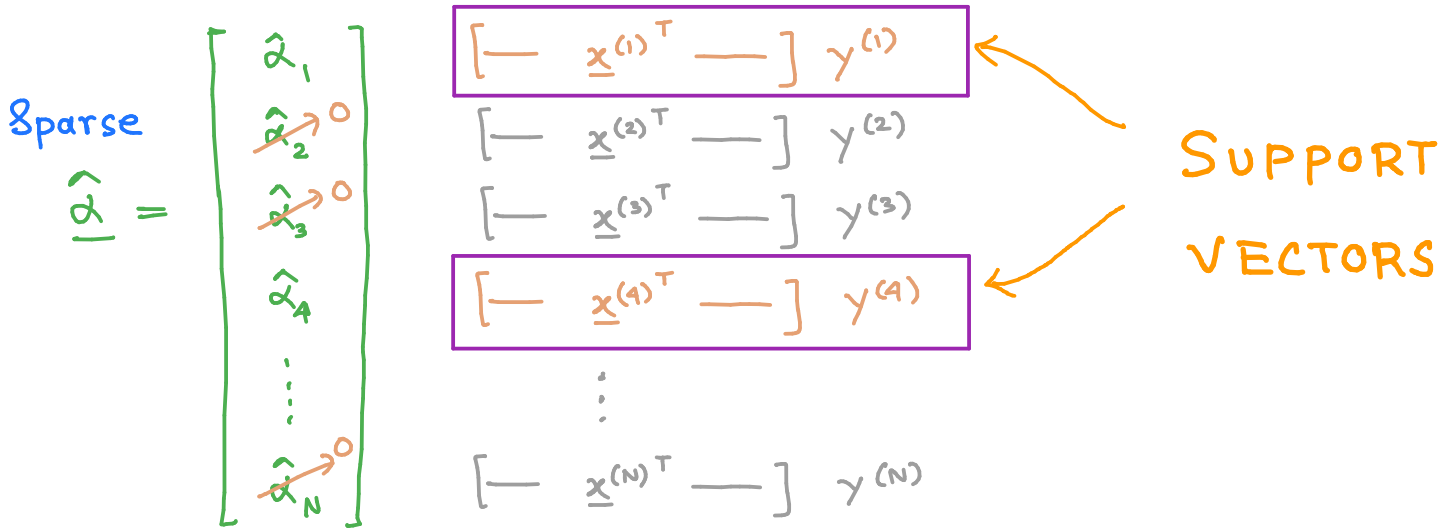
- Support vector regression, SVR, is another very useful kernel method for regression

- From a modelling perspective, it is very similar to kernel ridge regression, the only difference is in the use of a different loss function

- The new loss function is such that it makes the dual parameter $\hat{\underline{\alpha}}$ sparse, meaning several elements of $\hat{\underline{\alpha}}$ are exactly zero

- Recall that $\hat{\underline{\alpha}} \in \mathbb{R}^{N \times 1}$, so we can associate each element of $\hat{\underline{\alpha}}$ with one training data-point

- The training points corresponding to the non-zero elements of $\hat{\underline{\alpha}}$ are referred to as support vectors and the prediction $\hat{y}(\underline{x}^*)$ will only depend on these support vectors



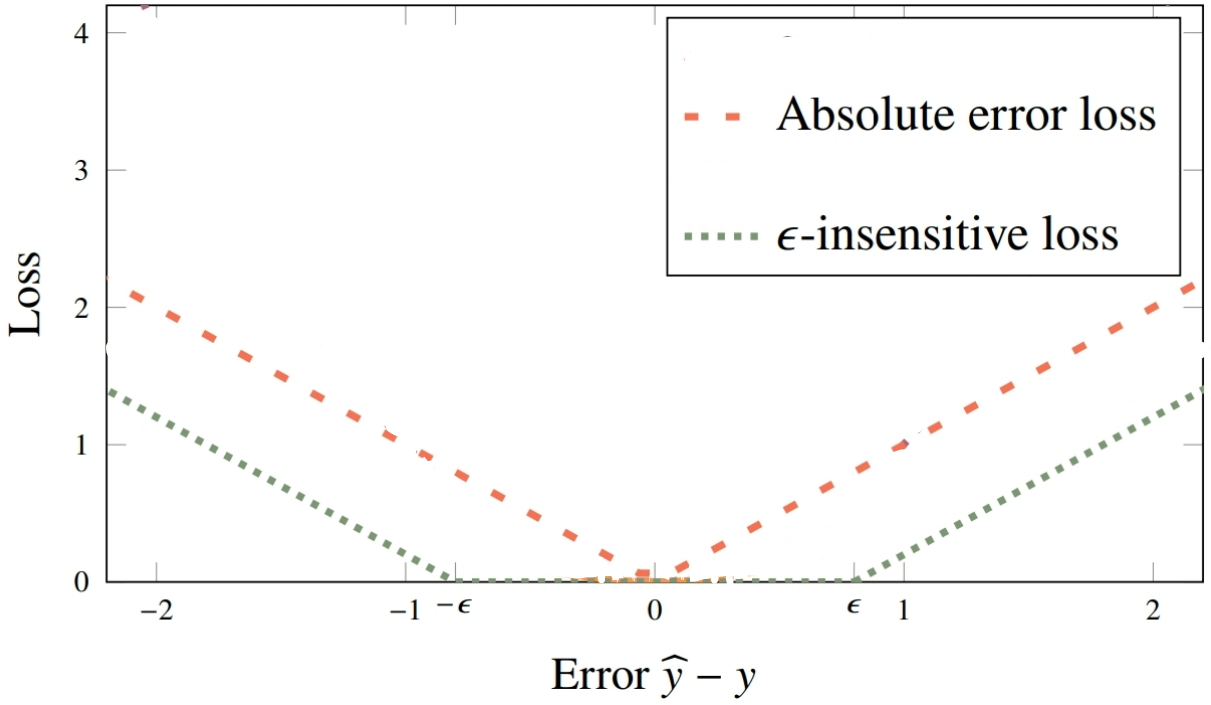
- SVR is an example of a so-called support vector machine (SVM) which is a family of methods with sparse dual parameter vectors

The loss function we will use for SVR is the ϵ-insensitive loss

$$L(y, \hat{y}) = \begin{cases} 0 & \text{if } |y - \hat{y}| < \epsilon, \\ |y - \hat{y}| - \epsilon & \text{otherwise} \end{cases}$$

$$L(y, \hat{y}) = \max\{0, |y - \hat{y}| - \epsilon\}$$

parameter ϵ is a design choice



- In primal formulation, SVR also makes use of the linear regression

$$\hat{y}(\underline{x}^*) = \underline{\theta}^T \underline{\phi}(\underline{x}^*)$$

but instead of the squared loss, we now have

$$\hat{\underline{\theta}} = \underset{\underline{\theta}}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} \max\{0, \left| y^{(i)} - \underline{\theta}^T \underline{\phi}(\underline{x}^{(i)}) \right| - \epsilon\} + \lambda \|\theta\|_2^2$$

there is no closed-form solution of $\underline{\theta}$

unlike in KRR

- Solution of $\hat{\underline{\theta}}$ has to be found using numerical optimization

- Similar to KRR, we use the kernel trick and move to the dual formulation with $\underline{\alpha}$ instead of $\underline{\theta}$

  — But there is no hope of closed-form solution of $\underline{\alpha}$

- In the dual formulation, we have

$$\hat{y}(\underline{x}^*) = \hat{\underline{\alpha}}^T \underline{\underline{K}}(\underline{X}, \underline{x}^*)$$

} ← same as KRR

where, $\hat{\underline{\alpha}}$ is obtained by solving a constrained optimization problem

$$\hat{\underline{\alpha}} = \underset{\underline{\alpha}}{\operatorname{argmin}} \; \frac{1}{2} \underline{\alpha}^T \underline{\underline{K}}(\underline{X}, \underline{X}) \underline{\alpha} - \underline{\alpha}^T \underline{y} + \epsilon \|\alpha\|_1$$

subject to $|\alpha_i| < \frac{1}{2N\lambda}$

acts as a regularization parameter

promotes sparsity

different from KRR where we had closed-form solution

Support vectors due to ϵ-insensitive loss

- This loss function is particularly interesting in the kernel context since the dual parameter vector $\underline{a}$ becomes sparse

- Since $\underline{a}$ has one entry per training data point, sparsity of $\underline{a}$ implies that the prediction $y(\underline{x}^*)$ will depend only on some of the training data points

  - These training points correspond to $a_i \neq 0$, and are called support vectors

Support vectors due to ϵ-insensitive loss

- The training points correspond to $\alpha_i \neq 0$ are support vectors
- It can be shown that support vectors are those data-points for which the loss function is non-zero :

$$\text{Support vectors} = \left\{ \{\underline{x}^{(i)}, y^{(i)}\} \text{ s.t. } \left|\hat{y}(\underline{x}^{(i)}) - y^{(i)}\right| \geq \epsilon \right\}$$

  - a larger ϵ will result in a fewer support vectors
  - ϵ acts as a regularization parameter in L_1-penalty in dual formulation
  - The number of support vectors is also affected by λ

- During prediction, only the support vectors contribute. So lesser support vectors means fewer computations

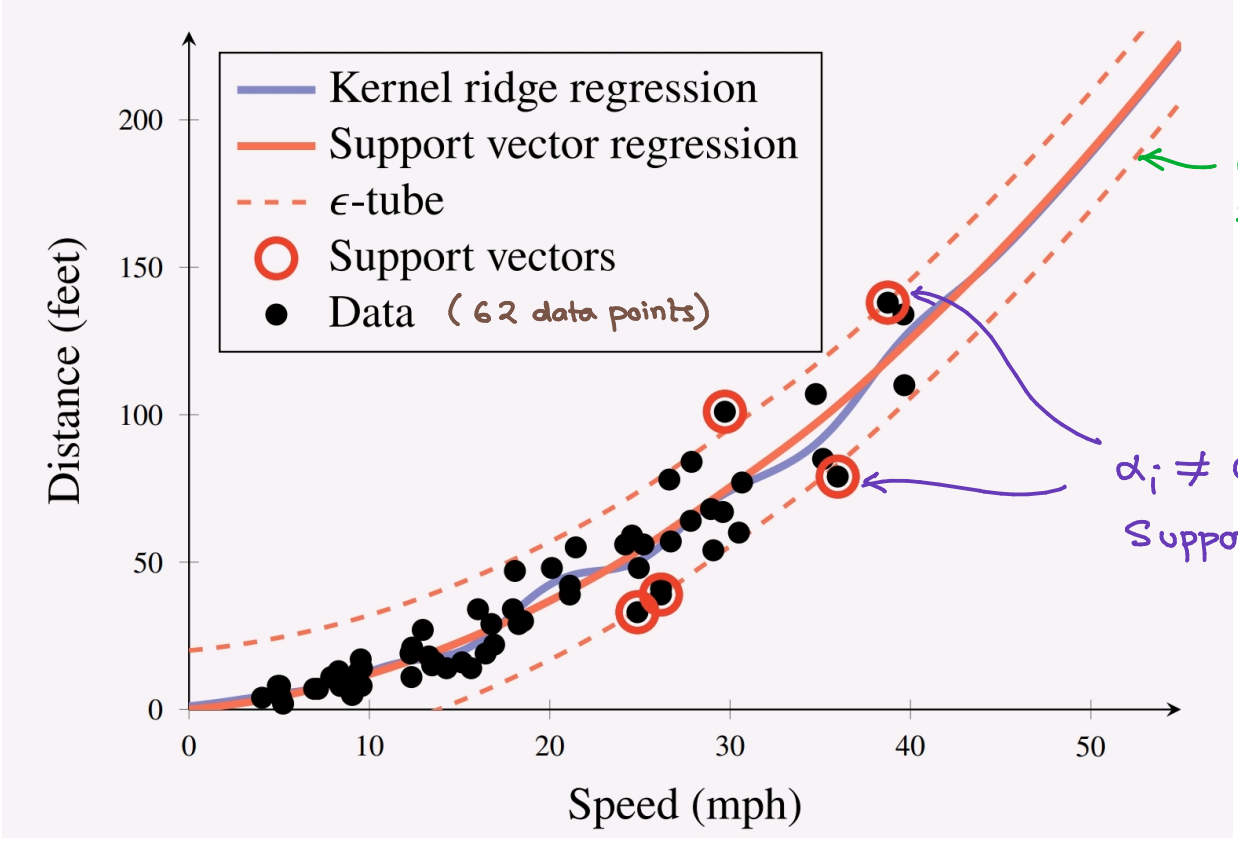
Example of regression with KRR and SVR

Car stopping distance problem : $x \rightarrow$ speed of car

$y \rightarrow$ time to stop after brakes applied

A combination of squared exponential & polynomial kernel is used

$\lambda = 0.01$, $\epsilon = 15$



all data points outside this ϵ-tube are support vectors

$\alpha_i \neq 0$

Support vectors

For prediction with SVR, it is sufficient to use only these 5 support vectors