CSE517A Machine Learning **Spring 2025**

Lecture 7: Kernel Methods

Instructor: Marion Neumann

Reading: LFD 3.4, 8.3 (Kernel Trick), FCML 5.3.2 (SVM & Kernel Methods), GPML[1] 4.2.4 (Making New Kernels)

Application

Consider modeling and predicting the concentration of CO2 in the atmosphere. Our data are atmospheric carbon dioxide measurements taken in the Mauna Loa weather station.

Take some time to answer the following warm-up questions: (1) Is this a regression or classification problem? (2) What are the features and how would you represent them? (3) What is the prediction task? (4) How well do you think a linear model will perform?

Terminology: *kernels* ≠ *kernels*

Note, that the notation of "*kernel*" is heavily overloaded in machine learning and we have to be careful about its meaning in various contexts. In this lecture, we will define the kind of kernels (so-called *Mercer kernels*) that are used in kernel methods. Those kernels are **covariance functions** that have to be *symmetric* and *positive-semi definite* (*psd*).

In other use cases like **kernel density estimation** the function needs to be positive and integrate to 1. Note that this is a stronger requirement that being *psd*.

For kernel regression and radial basis functions networks (cf. RBF networks lecture unit) the main property of the used kernel functions is that they are **positive functions of the distance** $\|\mathbf{x}-\mathbf{x}'\|$ that output lower values/weights for larger distances. We used the following notation of this kind of RBF or *kernel* function: $k(\mathbf{x}-\mathbf{x}') = g(z)$ to distinguish them from the notation we will use for (Mercer) kernels $k(\mathbf{x},\mathbf{x}')$.

Note, that some functions like the Gaussian kernel (also called RBF kernel) fulfill all of the above conditions, they are symmetric, psd (as we will see below), they are RBFs, and (if we use the normalization constant) they integrate to 1. But in general this is not true.

1 Introduction

Basic idea: We can make linear ML models non-linear by applying basis function (feature) transformations on the input space. For an input vector $\mathbf{x} \in \mathbb{R}^d$, do transformation $\mathbf{x} \to \phi(\mathbf{x})$ where $\phi(\mathbf{x}) \in \mathbb{R}^D$. Usually, $D >> d$ because you add dimensions that capture non-linear interactions among features.

[1] http://www.gaussianprocess.org/gpml/chapters/RW4.pdf

For example:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix} \quad \text{Define} \quad \phi(\mathbf{x}) = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \\ \vdots \\ x_1 x_2 \\ x_1 x_3 \\ \vdots \\ x_{d-1} x_d \\ \vdots \\ x_1 x_2 \ldots x_d \end{bmatrix}$$

Quiz: What is the dimensionality D of $\phi(\mathbf{x})$?

Solution: $D = 2^d$. For each element you choose if x_i is used (binary decision).

So, $\phi(\mathbf{x})$ is is very expressive but the dimensionality is extremely high.

2 Kernel Trick

Kernel trick aims to avoid computing explicit feature transformations $\phi(\mathbf{x})$ by changing the *entire* **ML algorithm** (training and prediction procedure) to use only *inner products* of feature transformations $\langle \phi(\mathbf{x}), \phi(\mathbf{x}') \rangle$ which are computed directly using a *kernel* (rather than the feature transformations themselves).

2.1 Example: Linear Model

Consider a linear model where we use gradient descent with a squared loss function to learn the parameters. So, we have $h(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$ and the squared loss function:

$$l(\mathbf{w}) = \sum_{i=1}^{n} (\mathbf{w}^\top \mathbf{x} - y_i)^2 \tag{1}$$

Training:
The gradient w.r.t. $\mathbf{w}$ is

$$g(\mathbf{w}) = \frac{\partial l}{\partial \mathbf{w}} = \sum_{i=1}^{n} 2(\mathbf{w}^\top \mathbf{x}_i - y_i)\mathbf{x}_i \tag{2}$$

Since the loss is convex, the final solution is independent of the initialization, and we can initialize $\mathbf{w}_0$ to be whatever we want. For convenience, we choose $\mathbf{w}_0 = \mathbf{0}$.

Recall gradient descent update rule:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - s\, g(\mathbf{w}_t) \tag{3}$$

where s is the step size. We can show that for every t, $\mathbf{w}_t$ is a linear combination of $\mathbf{x}_i$

$$\mathbf{w}_t = \sum_{i=1}^{n} \alpha_i^{(t)} \mathbf{x}_i \tag{4}$$

Using Eq. (2) and (3), we get

$$\begin{aligned} \mathbf{w}_{t+1} &= \mathbf{w}_t - s\, g(\mathbf{w}_t) \\ &= \sum_{i=1}^{n} \alpha_i^{(t)} \mathbf{x}_i - s \sum_{i=1}^{n} 2(\mathbf{w}_t^\top \mathbf{x}_i - y_i)\mathbf{x}_i \\ &= \sum_{i=1}^{n} \alpha_i^{(t+1)} \mathbf{x}_i \end{aligned} \tag{5}$$

where $\alpha_i^{(t+1)} = \alpha_i^{(t)} - 2s(\mathbf{w}_t^\top \mathbf{x}_i - y_i)$. So, instead of updating $\mathbf{w}_t$, we can update $\boldsymbol{\alpha}^{(t)}$.

Now, $\mathbf{w}$ can be written as a linear combination of the training set and we can also express the inner-product of $\mathbf{w}$ with any input $\mathbf{x}_i$ purely in terms of inner-products between training inputs:

$$\mathbf{w}_t^\top \mathbf{x}_i = \sum_{j=1}^{n} \alpha_j^{(t)} \mathbf{x}_j^\top \mathbf{x}_i \tag{6}$$

Consequently, the entire loss function/gradient descent algorithm is now in terms of $\mathbf{x}_j^\top \mathbf{x}_i$ which can now be replaced by $\langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle$ ($\rightarrow$ *kernel trick*).

Prediction: Now, for a new test input $\mathbf{x}^*$ we can use Eq. (6) as well:

$$y^* = \sum_{i=1}^{n} \alpha_i^{(conv)} \mathbf{x}_i^\top \mathbf{x}^* \text{ or with the } \textit{kernel trick}\text{: } y^* = \sum_{i=1}^{n} \alpha_i^{(conv)} \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}^*) \rangle \tag{7}$$

where $\boldsymbol{\alpha}^{(conv)}$ is the final output of our reformulated gradient descent algorithm at convergence. Note that we have turned our original learning problem that had d parameters ($d \in \mathbb{R}^d$) into one that now has n parameters ($\boldsymbol{\alpha} \in \mathbb{R}^n$). That's essentially the price we pay for getting a non-linear model.

2.2 From Feature Transformations to Kernels

Instead of computing $\phi(\mathbf{x}_i)$ for all $\mathbf{x}_i$, $i = 1, \ldots, n$, we **pre-compute** $\langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle$ for all $\mathbf{x}_i, \mathbf{x}_j$. Then we store the values in an $n \times n$ matrix K where $K_{ij} = \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle$. K is called the *kernel matrix* or *Gram matrix*.

So, what did we gain? In the previous example, computing $\phi(\mathbf{x}_i)$ is $\mathcal{O}(2^d) \times n = \mathcal{O}(2^d n)$. On the other hand, pre-computing the inner products can be more done efficiently as can be seen from:

$$\begin{aligned} \langle \phi(\mathbf{x}), \phi(\mathbf{x}') \rangle &= \phi(\mathbf{x})^\top \phi(\mathbf{x}') \\ &= 1 * 1 + x_1 x_1' + x_2 x_2' + \cdots + x_1 x_2 x_1' x_2' + x_1 x_3 x_1' x_3' + \cdots + x_1 x_2 \cdots x_d x_1' x_2' \cdots x_d' \\ &= \prod_{k=1}^{d} (1 + x_k x_k'). \end{aligned} \tag{8}$$

This takes $\mathcal{O}(d)$ time and we can compute the entire kernel matrix K in $\mathcal{O}(dn^2)$ time. ☺

Now, to get predictions at test time we also use Eq. (7) and plug in the new test input $\mathbf{x}$ and compute the kernel values between $\mathbf{x}$ and all the training points $\mathbf{x}_i$.

Prediction:

$$h(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i \mathrm{k}(\mathbf{x}, \mathbf{x}_i), \tag{9}$$

where $\mathrm{k}(\cdot, \cdot)$ is the *kernel function*.

Definition 2.1. ***Kernel:***

A kernel is the ***inner product*** *of a fixed (non-linear) feature space mapping* $\phi(\mathbf{x})$

$$k(\mathbf{x}, \mathbf{x}') = \langle \phi(\mathbf{x}), \phi(\mathbf{x}') \rangle.$$

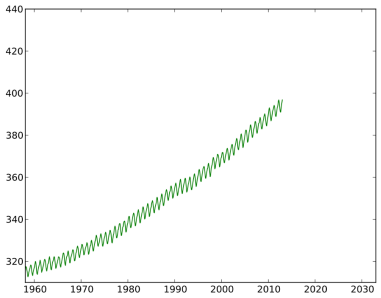
Hence, a kernel is a function $k(\cdot, \cdot) : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$.

Note that the model derived in the above example and in fact all kernel methods are *non-parametric models* as we need to keep training data to be able to compute the kernel values between new test inputs $\mathbf{x}$ and the training inputs $\mathbf{x}_i \; \forall i$ in Eq. (9).

Application

Back to our application modeling atmospheric carbon dioxide measurements.

The observed data is *non-linear* and also shows *periodic* behavior. It would be **extremely hard** (if not impossible) to come up with an explicit feature transformation to model this data. Instead we will construct a **kernel** to model this behavior. ☺



Can we construct kernels without explicit feature space transformations? The answer is YES!

On the one hand, kernels are defined as the inner product of (*explicit*) feature transformations. But on the other hand, kernels also define *implicit* feature transformations.

3 Kernels

Question: Can any function $k(\cdot,\cdot)$ be a kernel?
Answer: No.

3.1 Another Definition

Definition 3.1. ***Valid Kernel***

$k(\cdot,\cdot)$ is a valid or well-defined kernel, if the function $k(\mathbf{x},\mathbf{x}')$ is both

- *symmetric: $k(\mathbf{x},\mathbf{x}') = k(\mathbf{x}',\mathbf{x})$ for all $\mathbf{x},\mathbf{x}'$*
- *positive semi-definite: $k(\cdot,\cdot)$ is PSD if for all finite subsets $\{\mathbf{x}_1,...,\mathbf{x}_m\}, \mathbf{x}_i \in \mathcal{X}$, K is a PSD matrix.*

Linear Algebra Recap: PSD Matrices

- A matrix $A \in \mathbb{R}^{m\times m}$ is *positive-semi definite* (PSD) iff $\forall \mathbf{q} \in \mathbb{R}^m$ the following holds: $\mathbf{q}^\top A\mathbf{q} \geq 0$.
- A *symmetric* matrix $A \in \mathbb{R}^{m\times m}$ is *positive-semi definite* (PSD) iff all eigenvalues λ of A are non-negative (where eigenvalues satisfy $\lambda\mathbf{v} = A\mathbf{v}$ for $\mathbf{v} \neq 0$).
- A *symmetric* matrix A is *positive-semi definite* (PSD) if all its upper-left sub-matrices have non-negative determinants.

For *symmetric* matrices (e.g. kernel matrices) all of the above are equivalent.

Exercise 3.1.

(a) For positive definite matrices A, B show that $C = A .* B$ is positive definite, where $.*$ means element-wise multiplication.

(b) For a positive semi-definite matrix K show whether K^2 is strictly positive definite, positive semi-definite, or neither.

3.2 Common Kernels

Some of the most popular kernels are

- Linear kernel: $k(\mathbf{x},\mathbf{x}') = \mathbf{x}^\top\mathbf{x}'$

- Polynomial kernel: $\mathrm{k}(\mathbf{x}, \mathbf{x}') = (1 + \mathbf{x}^\top \mathbf{x}')^{\tilde{d}}$, where $\tilde{d}$ is the degree of the polynomial
- RBF/Gaussian kernel: $\mathrm{k}(\mathbf{x}, \mathbf{x}') = e^{-\frac{1}{2\ell^2}\|\mathbf{x}-\mathbf{x}'\|^2}$

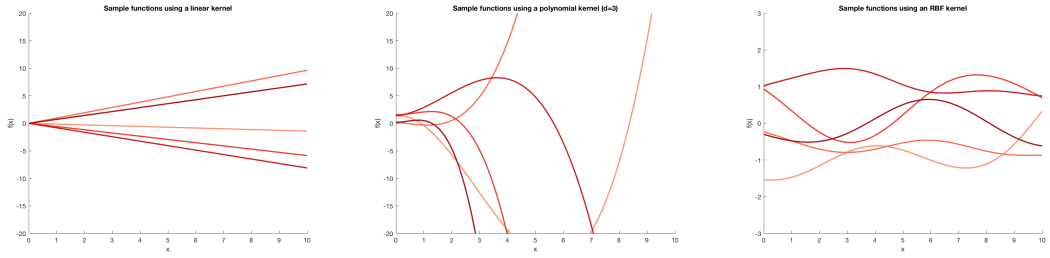


Figure 1: Illustration of kernels (left to right: linear, polynomial, RBF) modeling one dimensional sample functions $f(x)$ dependent on the "*similarity*" of x and x'. For more kernels and their illustrations, see https://www.cs.toronto.edu/~duvenaud/cookbook/.

Note, that some kernels such as the RBF kernel have an **infinite-dimensional feature space transformation** (cf. *homework problem*). Hence, it is impossible to compute the feature space transformation explicitly.

Exercise 3.2.

(a) Assume we have two training points $\mathbf{x}_1, \mathbf{x}_2$ and a test-point $\mathbf{z}$. In addition, you know that $\mathbf{z}$ is very close to $\mathbf{x}_1$, but very far away from $\mathbf{x}_2$. What statements are true about the **RBF kernel** entries (multiple are possible):

T/F $k(\mathbf{x}_1, \mathbf{z}) \geq 0$ and $k(\mathbf{x}_2, \mathbf{z}) \geq 0$

T/F $k(\mathbf{x}_1, \mathbf{z})$ is close to 0, whereas $k(\mathbf{x}_2, \mathbf{z})$ is very large.

T/F $k(\mathbf{x}_1, \mathbf{z})$ is close to 1, whereas $k(\mathbf{x}_2, \mathbf{z})$ is very close to 0.

T/F $k(\mathbf{x}_1, \mathbf{z})k(\mathbf{x}_2, \mathbf{z})$ is close to 0.

(b) Explain the effects on a classifier if the constant ℓ is moved from a very small value to a very large value.

(c) What property of the kernel can you control with ℓ?

(d) How do we pick a good value for ℓ?

(e) What is a drawback of the RBF kernel when you think about ℓ and multi-dimensional inputs?

3.3 Kernel Functions: Some Intuition

Let $y = f(\mathbf{x}) + \epsilon$. Then f does not vary a lot if $\mathbf{x}, \mathbf{x}'$ are close enough. This can be modeled by the co-variance of the $y's$.

Let's represent the co-variance of the outputs in terms of the co-variance of the inputs.

$$cov(y, y') = \mathrm{k}(\mathbf{x}, \mathbf{x}') + \underbrace{\sigma_n^2 \delta_{\mathbf{x},\mathbf{x}'}}_{\text{noise term}} \tag{10}$$

As y are function values of $f(\mathbf{x})$, the *kernel function* can be viewed as an extension of the covariance matrix Σ of random vectors to the covaraince of (random) functions.

3.4 Kernel Construction

Kernels built by recursively combining one or more of the following rules are also valid (or well-defined) kernels:

(1) $\mathrm{k}(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\top \mathbf{x}'$

(2) $\mathrm{k}(\mathbf{x}, \mathbf{x}') = c\mathrm{k}_1(\mathbf{x}, \mathbf{x}')$ where $c \geq 0$ and k_1 is a valid kernel

(3) $\mathrm{k}(\mathbf{x}, \mathbf{x}') = \mathrm{k}_1(\mathbf{x}, \mathbf{x}') + \mathrm{k}_2(\mathbf{x}, \mathbf{x}')$ where $\mathrm{k}_1, \mathrm{k}_2$ are valid kernels

(4) $\mathrm{k}(\mathbf{x}, \mathbf{x}') = g(\mathrm{k}_1(\mathbf{x}, \mathbf{x}'))$ where g is a polynomial with positive coefficients

(5) $\mathrm{k}(\mathbf{x}, \mathbf{x}') = \mathrm{k}_1(\mathbf{x}, \mathbf{x}') \times \mathrm{k}_2(\mathbf{x}, \mathbf{x}')$

(6) $\mathrm{k}(\mathbf{x}, \mathbf{x}') = f(\mathbf{x})\mathrm{k}_1(\mathbf{x}, \mathbf{x}')f(\mathbf{x}')$ where $f : \mathbb{R}^d \to \mathbb{R}$

(7) $\mathrm{k}(\mathbf{x}, \mathbf{x}') = e^{\mathrm{k}_1(\mathbf{x}, \mathbf{x}')}$

(8) $\mathrm{k}(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\top A \mathbf{x}'$ where A is PSD

Exercise 3.3. Proof all **kernel construction rules**.

HINT: to prove the rules there are a couple of different things to try. Note that not everything works for every rule.

- use the definition of a kernel (that is, construct a feature space, where $k(.,.)$ corresponds to the inner product)
- use the spectral decomposition $k_1(\mathbf{x}, \mathbf{x}') = \sum_{i=1}^{\infty} \lambda_i \phi_i(\mathbf{x})\phi_i(\mathbf{x}')$, where λ_i are the eigenvalues in the transformed feature space (cf. *Mercer's Theorem*)
- show that the *Gram matrix* corresponding to any arbitrary subset of elements in the input space is positive-semi definite by
  - showing that the quadratic form is non-negative for arbitrary vectors
  - showing that all eigenvalues are non-negative
- use already proven construction rules

Theorem: RBF is a valid kernel.

Proof:

$$\begin{aligned} \mathrm{k}(\mathbf{x}, \mathbf{x}') &= e^{\frac{-\|\mathbf{x}-\mathbf{x}'\|^2}{2\ell^2}} \\ &= \underbrace{\underbrace{e^{\frac{-\mathbf{x}^\top \mathbf{x}}{2\ell^2}}}_{f(\mathbf{x})} \quad \underbrace{e^{\frac{\mathbf{x}^\top \mathbf{x}'}{\ell^2}}}_{\text{use (1),(2),(7)}} \quad \underbrace{e^{\frac{-\mathbf{x}'^\top \mathbf{x}'}{2\ell^2}}}_{f(\mathbf{x}')}}_{\text{use (6)}} \end{aligned} \tag{11}$$

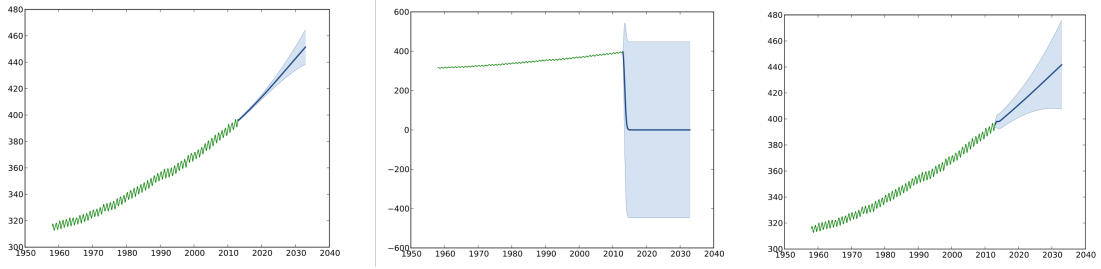
Exercise 3.4. Which of the following functions are valid kernels? Give a proof or argument that clearly confirms your decision.

(a) $\mathrm{k}(\mathbf{x}, \mathbf{x}') = \prod_{\alpha=1}^{d} h(\frac{x_\alpha - c}{a})h(\frac{x'_\alpha - c}{a})$, where $h : \mathbb{R} \to \mathbb{R}$ and $x_\alpha = [\mathbf{x}]_\alpha$

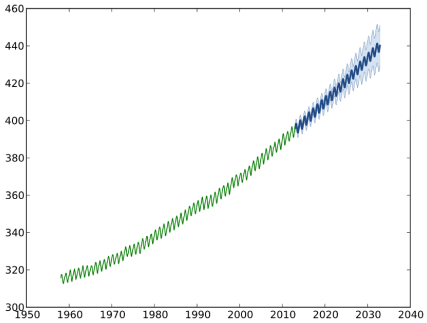
(b) $\mathrm{k}(\mathbf{x}, \mathbf{x}') = -\frac{\langle \mathbf{x}, \mathbf{x}' \rangle}{\|\mathbf{x}\|_2 \|\mathbf{x}'\|_2}$

Application

Let's look at kernel construction for our **atmospheric carbon dioxide model**. We will use the sum of several kernels. Let's start with two kernels, an RBF-kernel modeling the the long term smooth rising trend (*left*) and another RBF-kernel modeling the noise (*middle*) and let's add them up (*right*):[2]



Now, the predictions go in the right direction and the predictive uncertainty (light blue area) shows a nice behavior of getting larger for predictions that are further in the future. However, we are still missing the periodic component. Let's add a version of the periodic kernel $\mathrm{k}(x, x') = \sigma^2 \exp\left(-\frac{2\sin^2(\phi|x-x'|/p)}{\ell^2}\right)$:



See http://www.gaussianprocess.org/gpml/chapters/RW5.pdf for the full derivation and exact kernel definitions.

Summary

- A **kernel** is the inner product of a fixed non-linear feature mapping $\phi(\mathbf{x})$.
- The **kernel trick** refers to the fact that we do not have to explicitly compute $\phi(\mathbf{x})$ for both training and prediction. We only ever compute $\langle \phi(\mathbf{x}), \phi(\mathbf{x}') \rangle = \mathrm{k}(\mathbf{x}, \mathbf{x}')$.
- We can construct a kernel directly → it needs to be a **positive semi-definite** and **symmetric** function.
- The kernel implicitly defines the feature transformation of the input space. But we don't ever need it (so we don't worry about it).
- The feature mapping can be **infinite dimensional**.

[2] http://www.ra.cs.uni-tuebingen.de/lehre/ss12/advanced_ml/lecture3.pdf

Exercise 3.5. Practice Retrieving!
For this summary exercise, it is intended that your answers are based on **your own** (current) understanding of the concepts (and not on the definitions you read and copy from these notes or from elsewhere). Don't hesitate to **say it out loud** to your seat neighbor, your pet or stuffed animal, or to yourself before **writing it down**. After writing it down, check your answers with your (lecture)notes and the provided reading. Correct any mistakes you made. Research studies show that this practice of retrieval and phrasing out loud will help you retain the knowledge!

(a) Using *your own words*, recap each of the above summary points in 2-3 sentences by retrieving the knowledge from the top of your head.

(b) Why does a kernel that we construct directly need to be *symmetric* and *positive semi-definite*?

And always remember: It's not bad to get it wrong. *Getting it wrong is part of learning!* Use your notes or other resources to get the correct answer or come to our office hours to get help!

4 Kernel Machines

There are two steps to kernelize an ML method:

(1) Rewrite learning and prediction algorithm entirely in terms of inner-products.

(2) Define a kernel function and substitute $\mathrm{k}(\mathbf{x}_i, \mathbf{x}_j)$ for $\mathbf{x}_i^\top \mathbf{x}_j$

4.1 Example: Kernel Ridge Regression (Regularization Network)

In the following, we will derive kernel ridge regression which extends linear ridge regression to be a "true" kernel method. Note that this model is the regularized version of a kernel regression model. And also note that *this* kernel regression model (which extends ordinary least squares regression) is different from *kernel regression* (aka the Nadaraya-Watson model) derived in Lecture 6 (RBF Networks).

Let's start with the linear ridge regression model: $y = \mathbf{w}^\top \mathbf{x} + \epsilon$ (Primal)

From SRM, we know that the closed form solution is $\mathbf{w} = (XX^\top + \lambda I)^{-1} X\mathbf{y}$. Now our goal is to cast this solution in terms of inner products in order to get to the *dual representation*:

$$y = \mathbf{x}^\top X\boldsymbol{\alpha} \text{ (Dual)}$$

where $\mathbf{w} = X\boldsymbol{\alpha}$, $\boldsymbol{\alpha} = (X^\top X + \lambda I)^{-1}\mathbf{y}$ with $\boldsymbol{\alpha} \in \mathbb{R}^n$, $X^\top X \in \mathbb{R}^{n \times n}$, $\mathbf{y} \in \mathbb{R}^n$, $\mathbf{w} \in \mathbb{R}^d$. Note that this requires us to represent $\mathbf{w}$ as a linear combination of training data points: $\mathbf{w} = \sum_{i=1}^n \alpha_i \mathbf{x}_i = X\boldsymbol{\alpha}$ (as we did before in Eq. (4)). This representation is in fact guaranteed by the **Representer Theorem**.[3] Now, we can write:

$$\begin{aligned}
&\mathbf{w} = (XX^\top + \lambda I)^{-1} X\mathbf{y} = X\boldsymbol{\alpha} \\
&\iff (XX^\top + \lambda I) X\boldsymbol{\alpha} = X\mathbf{y} \\
&\iff (XX^\top X + \lambda X)\boldsymbol{\alpha} = X\mathbf{y} \\
&\iff X(X^\top X + \lambda I)\boldsymbol{\alpha} = X\mathbf{y} \\
&\iff \underbrace{X^\top X}_{=K}\Big(\underbrace{X^\top X}_{=K} + \lambda I\Big)\boldsymbol{\alpha} = \underbrace{X^\top X}_{=K}\mathbf{y} \\
&\iff (K + \lambda I)\boldsymbol{\alpha} = \mathbf{y} \\
&\iff \boldsymbol{\alpha} = (K + \lambda I)^{-1}\mathbf{y}
\end{aligned}$$

[3] **Representer Theorem**: *the minimizer of a regularized empirical loss function defined over a reproducing kernel Hilbert space can be represented as a finite linear combination of kernel values evaluated on the training data* (Schölkopf et. al 2001).

Kernel ridge regression model: $y = \mathbf{w}^\top \phi(\mathbf{x}) + \epsilon$ (Primal[4]) or $y = K_{\mathbf{x}}\boldsymbol{\alpha}$ with $\boldsymbol{\alpha} = (K + \lambda I)^{-1}\mathbf{y}$ (Dual)

Exercise 4.1. Linear ridge regression

(a) Derive the (closed-form) solution $\mathbf{w}$ to linear ridge regression (again) in matrix notation.

(b) Verify that all the dimensions for all matrix and vector multiplications line up correctly!

Prediction:

$$h(\mathbf{x}) = K_{\mathbf{x}}\boldsymbol{\alpha} = \sum_{i=1}^{n} \alpha_i \mathrm{k}(\mathbf{x}_i, \mathbf{x}) \tag{12}$$

where $K_{\mathbf{x}} = \mathrm{k}(\mathbf{x}, X)$ which is a row in the *test-train* portion of kernel matrix that corresponds to $\mathbf{x}$ (if pre-computed) and $\boldsymbol{\alpha} = (K + \lambda I)^{-1}\mathbf{y}$ with $K = \mathrm{k}(X, X)$ being the *train-train* portion of the kernel matrix (pre-compute that as part of your training phase).

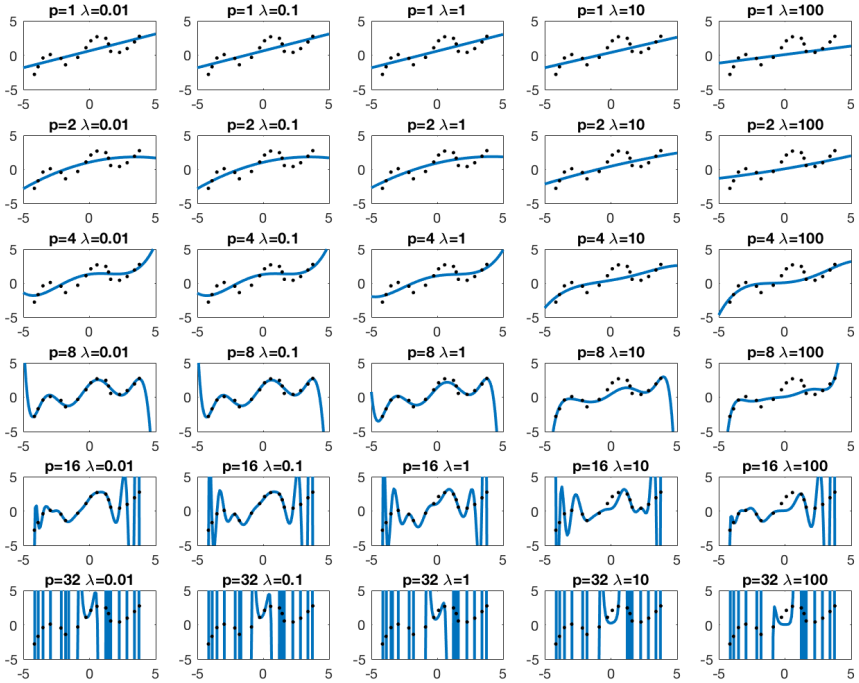


Figure 2: Illustration of kernel ridge regression with a polynomial kernel (where p is the degree of the polynomial)

[4]Note that we can only ever write and compute this if the feature transformation is **finite dimensional**.

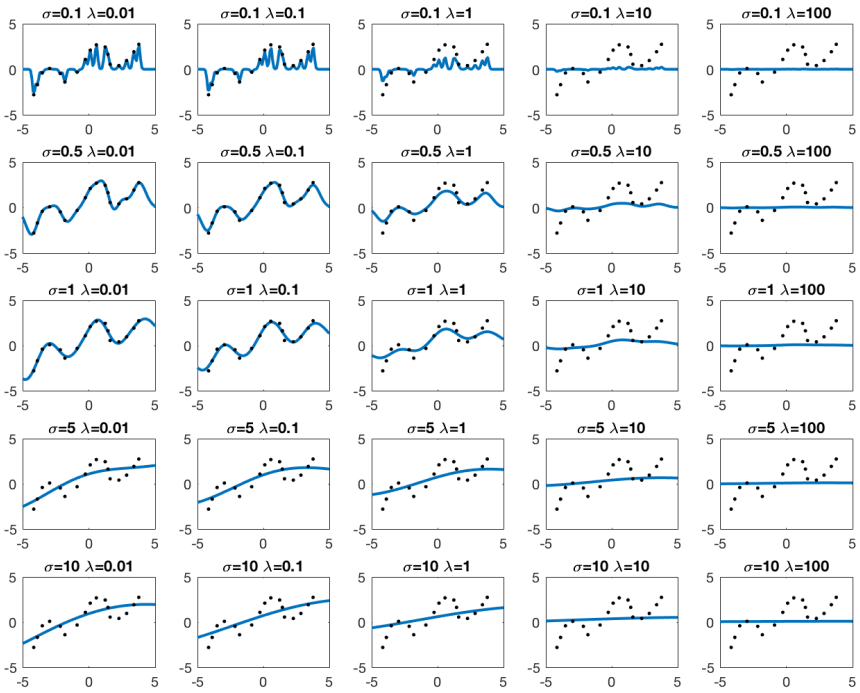


Figure 3: Illustration of kernel ridge regression with a Gaussian/RBF kernel (where σ is the kernel width)

4.2 Example: Kernel SVM

The original (primal) SVM is a quadratic programming problem:

$$\begin{aligned} & \min \frac{1}{2}\mathbf{w}^\top \mathbf{w} + c\sum_{i=1}^{n}\xi_i \\ \text{s.t.} \quad & y_i(\mathbf{w}^\top \mathbf{x}_i + b) \geq 1 - \xi_i \quad \forall i \\ & \xi_i \geq 0 \quad \forall i \end{aligned} \tag{13}$$

It has a dual form:

$$\begin{aligned} & \min_{\mathbf{a}} \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} a_i\, a_j\, y_i\, y_j \underbrace{\mathbf{x}_i^\top \mathbf{x}_j}_{\mathrm{k}(\mathbf{x}_i,\mathbf{x}_j)} - \sum_{i=1}^{n} a_i \\ \text{s.t.} \quad & 0 \leq a_i \leq c \quad \sum_{i=1}^{n} a_i\, y_i = 0 \quad \forall i \end{aligned} \tag{14}$$

Almost all $a_i = 0$, only the support vectors have $a_i \neq 0$. Hence, SVM is also called a *sparse* kernel machine.

Use the **dual** form of the model for prediction:

$$f(\mathbf{x}) = \sum_{i=1}^{n} a_i\, y_i\, \mathrm{k}(\mathbf{x}_i, \mathbf{x}) + b \tag{15}$$

$$h(\mathbf{x}) = sign(f(\mathbf{x})) \tag{16}$$

4.3 Other Kernel Machines

- relevance vector machines
- Gaussian processes (Bayesian kernel machine) → next lecture unit
- kernel k-means, kernel PCA