Introduction to Computational Learning Theory **Spring 2021**

Lecture 12: Kernel Methods

Lecturer: Roi Livni *Scribe:*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

We recall the standard setting of Stochastic convex optimization, where we (in a nutshell) consider convex problem of the forms:

$$\text{minimize} \quad \mathbf{E}_{z\sim D} f(\mathbf{w}, z) \tag{12.1}$$

$$\text{s.t.} \quad \mathbf{w} \in K. \tag{12.2}$$

In the last two lectures we were concerned with two general issues.

- Generalization– What is the true expected loss of our candidate solution to eq. (12.1): Namely, what is the *generalization error* of $\mathbf{w}$, the output:

$$\text{Generalization error} = \frac{1}{m}\sum_{i=1}^{m} f(\mathbf{w}, z_i) - \mathop{\mathbb{E}}_{z\sim D}[f(\mathbf{w}, z)].$$

- Optimization – How to compute an approximate optimizer for the task in eq. (12.1):

$$\text{Optimization error} = \frac{1}{m}\sum_{i=1}^{m} f(\mathbf{w}, z_i) - \min_{\mathbf{w}\in K} \frac{1}{m}\sum_{i=1}^{m} f(\mathbf{w}, z_i).$$

As we discussed in last lecture, we can now bound our test error (the loss of the output) with these two terms. Roughly:

$$\text{test error} \leq \underset{\mathbf{w} \in K}{OPT} + O\left(\text{generalization error} + \text{optimization error}\right)$$

In this lecture, we will discuss *expressiveness*, or what types of model can we learn in this framework.

This is a completely non-exhaustive discussion and there is a very rich, active, and fruitful research direction to formulate many important and interesting learning problems in the framework of convex optimization. So we will focus on a particular, very popular, "trick" to enhance the expressiveness of convex optimization, which we already touched upon in the context of the perceptron.

12.1 (General) Linear Models

We start with an example that we discussed (extensively) throughout the class: General Linear Models. In a general linear model we consider some convex loss function ℓ and we wish to find a parameter $\mathbf{w} \in K$ that minimizes an objective of the form

$$\mathbb{E}[f(\mathbf{w}, z) = \underset{(x,y)}{\mathbb{E}}[\ell(\mathbf{w} \cdot x; y)].$$

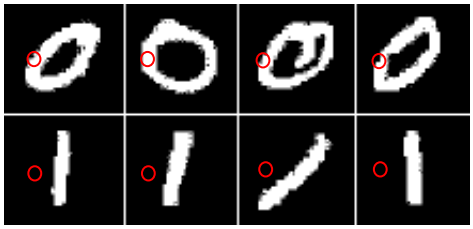
As we so far showed, if the function ℓ is "nice" (e.g. Lipschitz) the generalization error can be bounded using techniques such as Rademacher complexity. Also if ℓ has a computable derivative, the empirical variant of the above problem can be optimized using standard gradient descent methods (stochastic or not). A general linear model can be very useful, for example if we want to predict if a certain person is male or female: it makes sense to consider some averaging of features such as weight and height in order to obtain good prediction. You can also think of more advanced problems such as movie recommendation: we might want to average recommendations from different critiques in order to correctly decide how we want to score the movie: So in this setting we will represent each movie as a vector $(x_1, x_2, \ldots x_d)$ where each feature x_j represents a score given by reviewer j (say between $[-1, 1]$). Say we now obtain a list of examples $(\mathbf{x}_i, y_i)$ of new movies and new scores given by an independent person and we want to predict her scores on unseen movies: Then we might want to find some averaging schemes of existing reviews of the form

$$y = \mathbf{w} \cdot \mathbf{x} = \sum \mathbf{w}_i \cdot x_i.$$

However, it should be clear that these models are quite limited, as they can't express non-linear problems.

Question: How well does a linear predictor perform on MNIST dataset?

0.92



12.2 Warmup-Small Degree polynomials

Let us recall how we utilized kernel methods to when applying the Perceptron algorithm. We begin by considering the following slight strenghtening of linear models: Suppose we want to learn a model where, instead of just considering linear outputs, we want some polynomial. In other words we wish to minimize over p the problem:

$$\mathbb{E}[\ell(p(\mathbf{x}); y)],$$

where p is some multivariate polynomial of fixed dimension (for concretness 3). To exploit the framework of GLMs, we make the observation that a polynomial of degree 3 has the form

$$p(\mathbf{x}) = w_0 + \sum_{j_1} w_{j_1} \cdot x_{j_1} + \sum_{j_1 \le j_2} w_{j_1,j_2} \cdot (x_{j_1} \cdot x_{j_2}) + \sum_{j_1 \le j_2 \le j_3} w_{j_1,j_2,j_3} \left(x_{j_1} \cdot x_{j_2} \cdot x_{j_3}\right).$$

We can simplify the above equation if we assume that, say the first feature x_1 is always a constant (in other words, we add a feature x which equals 1 always). In this case we can write

$$p(\mathbf{x}) = \sum_{j_1 \le j_2 \le j_3} w_{j_1,j_2,j_3} \left(x_{j_1} \cdot x_{j_2} \cdot x_{j_3}\right).$$

where the first three terms of the last equation are suppressed by taking $j_1 = j_2 = j_3 = 1$ or $j_1 = j_2 = 1$ or $j_1 = 1$ (i.e. for example $w_{1,1,1}$ will equal w_0 in the former formulation etc..). So for simplicity let us consider the later formulation.

We can write it slightly different as follows: Let Φ_{p3} be an embedding of a vector $\mathbf{x} \in \mathbb{R}^d$ to $\mathbb{R}^{\binom{d}{3}}$ such that:

$$[\Phi_{p3}(\mathbf{x})]_{j_1,j_2,j_3} = x_{j_1} \cdot x_{j_2} \cdot x_{j_3}.$$

Then for each polynomial p we can associate a vector $\mathbf{w} \in \mathbb{R}^{\binom{d}{3}}$ such that

$$p(\mathbf{x}) = \mathbf{w} \cdot \Phi_{p3}(\mathbf{x}).$$

Now we are facing with a linear problem.

Let us summarize our idea in the following procedure in order to learn a degree-3 polynomial over our data

- Consider the data set $\mathbf{x} = (x_1, x_2, \cdots, x_d)$. We assume that $x_1 = 1$, if not we reparameterize our data set to obtain such a vector (i.e. we consider the mapping $\mathbf{x} \to (1, x_1, x_2, \cdots x_d)$).

- Represent your data in the form of a vector

$$\Phi_{p3}(x) = (1, x_1, \ldots, x_d, x_1^2, x_1, \ldots, x_2, x_1 \cdot x_3, \ldots, x_d^2, x_1^3, x_1^2 \cdot x_2, \ldots, x_1 \cdot x_2 \cdot x_3, \ldots)$$

- Learn a general linear model over the new representation in $\mathbb{R}^{\binom{d}{3}}$:. Namely, consider the optimization

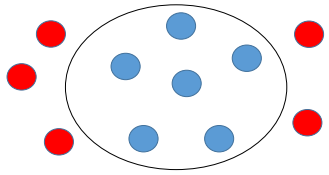


Figure 12.1: By embedding (x_1, x_2) into the feature space of monomials $(x_1, x_2, x_1^2, x_2^2, x_1 \cdot x_2)$, and learning a linear classifier, we can learn a target function of the form (for example): $p(\mathbf{x}) = x_1^2 + x_2^2 - 1$. This target function will assign negative sign to every point in the circle, and positive point to every point outside of the circle. The target function is linear in the ambient space but translates to a non-linear classifier in the original space.

problem

$$\begin{aligned} \text{minimize} \quad & \frac{1}{m}\sum_{i=1}^{m} \ell(\mathbf{w} \cdot \Phi_{p_3}(\mathbf{x}_i); y_i) \\ \text{s.t.} \quad & \mathbf{w} \in K. \end{aligned}$$

The constraint set K represents the types of polynomials we are learning. For example if $K = \{\mathbf{w} : \|\mathbf{w}\| \leq 1\}$ we will learn polynomials where the norm of the coefficient vector is bounded by 1.

Let us start by analysing generalization error as well as the optimization complexity of this approach

- **Sample Complexity** In terms of generalization note that we consider a GLM in $\mathbb{R}^{O(d^3)}$. As such, the tools we obtained using Rademacher readily applies. Note that they depend though on the constraint we put on the coefficient vector.

  Thus, for example, if we want to learn a polynomial with ℓ_2-bounded coefficient vector we can apply theorem 9.13 which states that if

  $$m = O\left(\max\left(\frac{\|\Phi_{p_3}(\mathbf{x})\|}{\epsilon^2}\right)\right).$$

  (here we neglect factors such as ρ-Lipschitz constant of ℓ and δ-confidence), we can achieve ϵ-generalization error and learn a coefficient vector with norm 1.

Suppose the original vector $\mathbf{x}$ had norm bounded by 1, then we can give a rough estimate to $\|\Phi_{p3}(\mathbf{x})\|$

$$\begin{aligned}\|\Phi_{p3}(\mathbf{x})\| &= \sqrt{\sum_{j_1 \le j_2 \le j_3} x_{j_1}^2 \cdot x_{j_2}^2 \cdot x_{j_3}^2} \\ &\le \sqrt{\sum_{j_1 j_2 j_3 \in \{1,\dots,d\}} x_{j_1}^2 \cdot x_{j_2}^2 \cdot x_{j_3}^2} \\ &= \sqrt{\left(\sum x_i^2\right)^3} \\ &= \|\mathbf{x}\|^3 \\ &\le 1\end{aligned}$$

Thus, to learn a polynomial with normalized coefficient vector we will need $O(1/\epsilon^2)$ examples.

- **Computational Complexity** In terms of optimization. As discussed, we an apply GD or SGD. SGD performs $O(G^2/\epsilon^2)$ iterations in order to achieve ϵ-accuracy if we want to learn norm 1 coefficient vectors (see theorem 10.2). The Lipschitness of the loss function is given by:

$$\|\nabla \ell(\mathbf{w} \cdot \Phi_{p_3}(\mathbf{x}); y)\| = \|\ell'(\mathbf{w} \cdot \Phi_{p_3}(\mathbf{x}); y)\Phi_{p_3}(\mathbf{x})\| = |\ell'(\mathbf{w} \cdot \Phi_{p_3}(\mathbf{x}); y)|\|\Phi_{p_3}(\mathbf{x})\|.$$

  So again we obtain dependence on $\|\Phi_{p_3}(\mathbf{x})\|$ which is normalized and the number of iterations we will have to perform is order $O(1/\epsilon^2)$ again. Note however that in order to calculate the derivative we need to calculate $\mathbf{w} \cdot \Phi_{p_3}(\mathbf{x})$ which requires $O(d^3)$ calculations.

To summarize, by embedding the data in higher-dimension feature space we altered the expressiveness of the models we are learning. On the other hand, more features mean more prohibitive computational cost, and it is clear that if we want to learn high-dimensional polynomial this approach becomes intractable very soon.

When discussing the perceptron, we discussed that for some embeddings Φ we can compute the scalar product in time linear in d instead of being dependent on the dimension of the ambient space. This can be described very neatly in the language of linear (or Hilbert) spaces:

12.2.1 Learning in Hilbert Spaces

Definition 12.1. *A Hilbert Space H is a linear space (i.e. a vector space equipped with addition and multiplication by scalar) equipped with a dot product which is a function $\langle \cdot, \cdot \rangle : H \times H \to \mathbb{R}$ s.t.:*

1. $\langle x; y_1 + y_2 \rangle = \langle x; y_1 \rangle + \langle x; y_2 \rangle$
2. $\langle c \cdot x; y \rangle = c \langle x; y \rangle$
3. $\langle x; y \rangle = \langle y; x \rangle$.

Every dot product in a Hilbert space defines a norm $\|\mathbf{x}\| = \sqrt{\langle \mathbf{x}; \mathbf{x} \rangle}$, which satisfies the Cauchy Schwartz inequality:

$$|\langle \mathbf{x}; \mathbf{y} \rangle| \leq \|\mathbf{x}\| \|\mathbf{y}\| \tag{12.3}$$

Example 12.1. *The simplest and most familiar example of a Hilbert-space is $\mathbb{R}^d$ with the scalar product $\langle \mathbf{x}; \mathbf{y} \rangle = \mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{d} x_i y_i$. So Hilbert-spaces can be thought of as a generalization of this setting.*

Definition 12.2. *A Hilbert space is called an RKHS if it is a space of continuous functions $f : \mathcal{X} \to \mathbb{R}$ from a domain $\mathcal{X}$ to $\mathbb{R}$ such that: There exists a continous mapping $\Phi : \mathcal{X} \to H$ for which every $f \in H$:*

$$\langle \Phi(\mathbf{x}); f \rangle = f(\mathbf{x})$$

The kernel function of the RKHS is defined as $k(\mathbf{x}_1, \mathbf{x}_2) = \langle \Phi(\mathbf{x}_1); \Phi(\mathbf{x}_2) \rangle$

Example 12.2. *Let us show that Φ_{p3} defines an RKHS. Indeed let us consider $H = \mathbb{R}^{\binom{d}{3}}$ equipped with the standard scalar product as a space of continous functions over $\mathcal{X} = \mathbb{R}^d$, operating through $w : \mathcal{X} \to \mathbb{R}$ as*

$$\mathbf{w}(\mathbf{x}) \to \mathbf{w} \cdot \Phi_{p3}(\mathbf{x}) = \sum_{j_1 \leq j_2 \leq j_3} w_{j_1, j_2, j_3} x_{j_1} \cdot x_{j_2} \cdot x_{j_3}.$$

The mapping Φ_{p3} indeed maps $\mathcal{X}$ to H and satisfies $f(\mathbf{x}) = \langle f; \Phi(\mathbf{x}) \rangle$.

Theorem 12.3 (Mercer Condition [2])**.** *A kernel function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ defines an RKHS if and only if*

for every finite sample $\{\mathbf{x}_i\}_{i=1}^m$ *the kernel matrix*

$$K_{i,j} = k(\mathbf{x}_i, \mathbf{x}_j)$$

is psd (positive semidefinite).

Recall that a symmetric matrix K is positive semidefinite if for every v we have

$$v^\top K v \geq 0.$$

An alternative (equivalent definition) is that the eigenvalues of K are all non-negative.

Example 12.3. *The following kernel functions define a kernel space*

1. $k(\mathbf{x}_1, \mathbf{x}_2) = (\mathbf{x}_1 \cdot \mathbf{x}_2)^d$, *Homogenous Polynomial Kernel*

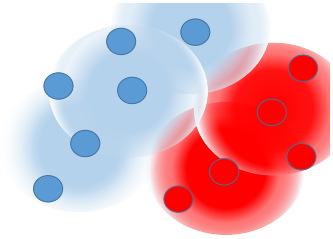
2. $k(\mathbf{x}_1, \mathbf{x}_2) = (1 + \mathbf{x}_1 \cdot \mathbf{x}_2)^d$, *Polynomial Kernel*

3. $k(\mathbf{x}_1, \mathbf{x}_2) = \exp^{-\|\mathbf{x}_1 - \mathbf{x}_2\|^2}$ *Gaussian Kernel*

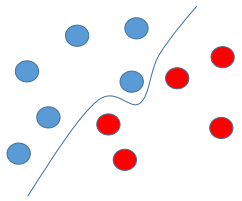
4. $k(\mathbf{x}_1, \mathbf{x}_2) = \exp^{-\langle \mathbf{x}_1; \mathbf{x}_2 \rangle}$ *Exponential Kernel*

There are many other kernels in the literature: What makes many of them helpful is the fact that we can efficiently compute the kernel: For example, both the embedding Φ_{p_3} and the polynomial kernels, can be considered as embedding in a polynomial feature space. However, while the kernel function of the first requires $O(d^3)$ computations, the kernel function of the later requires $O(d)$ computations.

As we will see this allows us to efficiently implement SGD in the Hilbert Space. In fact, it was observed that many algorithm, may be implemented given just an oracle for dot product: SGD is one example but also PCA [1] and other types of optimization algorithm. This observation has led to succefull kernelization of many other algorithms: not just classification. For a book about kernel learning we refer to [2].



(a) RBF Kernel: Using the RBF Kernel each point is mapped to a function $k(\mathbf{x}_i, \mathbf{x}) = e^{-\|\mathbf{x}_i - \mathbf{x}\|^2/\sigma}$. This mapping assign 1 to points close to $\mathbf{x}_i$ and zero for far away points. A linear function in the ambient space, translates into a linear combinations of such Radial basis functions.



(b) A polynomial kernel maps a point $\mathbf{x}_i$ to the function $p_{\mathbf{x}_i}(\mathbf{x}) = (\gamma + \mathbf{x}_i \cdot \mathbf{x})^d$ which is a polynomial, thus a linear separator in the RKHS translates into a linear combination of such polynomials, and allows learning an optimal polynomial for classification.

Figure 12.2: Different target functions expressible using kernel methods

The first question that comes to mind is the sample complexity of learning in an RKHS. The following result is proved in the same manner as theorem 9.13 (and in fact generelizes it to Hilbert spaces instead of ℓ_2 space):

Corollary 12.4 (Cor. of theorem 9.13)**.** *Let H be an RKHS such that $k(\mathbf{x}_i, \mathbf{x}_i) \leq B$. Let*

$$\mathcal{F} = \{\mathbf{w} : \mathcal{X} \to \langle \mathbf{w}; \Phi(\mathbf{x}) \rangle : \ \|\mathbf{w}\|_H \leq 1\}, \ \}$$

then for every convex Lipschitz loss function we have that

$$\sup_{\mathbf{w} \in \mathcal{F}} \mathcal{L}_D(\mathbf{w}) - \mathcal{L}_S(\mathbf{w}) \leq O(\frac{L \cdot B \log 1/\delta}{\sqrt{m}})$$

A crucial point is that the result is *dimension – independent*, i.e. the sample complexity of $\mathcal{F}$ is independent of the dimension of H. For the case of polynomial kernels, we get to learn in a feature space of monomials that is order of magnitude of $O(n^d)$, exponential in the degree. For the case of RBF kernel, H is even infinite dimensional – yet we can still obtain tractable sample complexity.

12.2.2 The computational cost of learning in an RKHS

At a first glance, learning in a high dimensional feature space might seem prohibitive, especially since we need to deal with high dimensional vector. However, a close examination of GD algorithm or SGD shows

why we can implement them (under some assumption) even in high dimensional feature space.

The crucial observation is that the bottleneck of the computation cost is to compute the scalar product: Going back to the embedding of $\mathbf{x}$ using $\Phi_{p3}(\mathbf{x})$: the number of iterations we perform is roughly $O(1/\epsilon^2)$ (negelecting dependence on the diameter and norm of $\mathbf{x}$'s) which is reasonable. However, each iteration, if implemented naively, requires time $O(d^3)$: the time it takes to compute a scalar product. But the point is that if we can implement SGD in a way that the time to compute each iteration will scale with the time it takes to compute the scalar product we can exploit different kernels that yield easy calculation of the scalar product:

The Kernel Trick: We now want to extend the idea of the perceptron to the SGD algorithm. Namely, we assume we have access to a kernel matrix K, where $K_{i,j} = \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j)$, and we want to implement the different steps of the SGD algorithm.

Similiar to the perceptro case, we would like to maintain a vector α_t at step t, such that $\mathbf{w}_t = \sum \alpha_t(i) \cdot \Phi(\mathbf{x}_i)$.

Then to compute the scalar product, we have

$$\langle \mathbf{w}_t; \Phi(\mathbf{x}_j) \rangle = \sum \alpha_t(i) \Phi(\mathbf{x}_i) \Phi(\mathbf{x}_j) = \sum \alpha_t(i) k(x_i, x_j) = (\alpha K)_i.$$

First we initialize at $\mathbf{w}_1 = 0$, then we can pick $\alpha = 0$.

At the next iteration, the algorithm sets

$$\begin{aligned} \mathbf{w}_{1/2} = \mathbf{w}_{1/2} &= 0 - \nabla \ell(\langle 0; \Phi(\mathbf{x}_1 \rangle, y_1) \\ &= 0 - \eta_1 \cdot \ell'(\langle 0; \Phi(\mathbf{x}_1) \rangle; y_1) \Phi(\mathbf{x}_1) \qquad\qquad \nabla \langle \mathbf{w}; \mathbf{x} \rangle = \mathbf{x}. \end{aligned}$$

The next step requires projection of $\mathbf{w}_{1/2}$

Note that if we know $k(\mathbf{x}_1, \mathbf{x}_1)$ (which is the underlying algorithmic assumption we make in the kernel trick)

we can easily compute $\|\mathbf{w}_{1/2}\|$:

$$\begin{aligned}\|\mathbf{w}_{1/2}\|^2 &= \left(\eta_1 \cdot \ell'(\langle 0; \Phi(\mathbf{x}_1)\rangle; y_1)\right)^2 \cdot \|\Phi(\mathbf{x}_1)\|^2 \\ &= \left(\eta_1 \cdot \ell'(\langle 0; \Phi(\mathbf{x}_1)\rangle; y_1)\right)^2 \cdot \Phi(\mathbf{x}_1) \cdot \Phi(\mathbf{x}_1) \\ &= \left(\eta_1 \cdot \ell'(\langle 0; \Phi(\mathbf{x}_1)\rangle; y_1)\right)^2 \cdot k(\mathbf{x}_1; \mathbf{x}_1).\end{aligned}$$

Then

$$\mathbf{w}_1 = \frac{\mathbf{w}_{1/2}}{\|\mathbf{w}_{1/2}\|} := \alpha_1(1) \cdot \Phi(\mathbf{x}_1).$$

Where we set by construction

$$\alpha_1(1) = -\frac{\eta_1 \cdot \ell'(0; y_1)}{k(\mathbf{x}_1; \mathbf{x}_1)},$$

Suppose we computed

$$\mathbf{w}_t = \sum \alpha_t(i)\Phi(\mathbf{x}_i).$$

Then

$$\begin{aligned}\mathbf{w}_{t+1/2} &= \mathbf{w}_t - \eta_t \nabla \ell(\langle \mathbf{w}_t; \Phi(\mathbf{x}_t)\rangle; y_t) \\ &= \mathbf{w}_t - \eta_t \ell'(\langle \mathbf{w}_t; \Phi(\mathbf{x}_t)\rangle; y_t)\Phi(\mathbf{x}_t) \qquad\qquad \nabla_{\mathbf{w}} \ell(\langle \mathbf{w}; \mathbf{u}\rangle; y_t) = \ell'(\langle \mathbf{w}; \mathbf{u}\rangle; y_t) \cdot \mathbf{u} \\ &:= \sum_{i=1}^{t-1} \alpha_t(i)\Phi(\mathbf{x}_i) + \alpha' \Phi(\mathbf{x}_t).\end{aligned}$$

where we set

$$\alpha' = -\eta_t \nabla \ell(\langle \mathbf{w}_t; \Phi(\mathbf{x}_t)\rangle; y_t).$$

Let us write $\alpha_{t+1/2}$ a vector such that $\alpha_{t+1/2}(i) = \alpha_t(i)$ for $i \leq t-1$, and $\alpha_{t+1/2}(t) = \alpha'$. Then, next we need to choose

$$\mathbf{w}_{t+1} = \frac{\mathbf{w}_{t+1/2}}{\|\mathbf{w}_{t+1/2}\|}.$$

For this we need to compute the norm of $\|\mathbf{w}_{t+1/2}\|$:

$$\begin{aligned}\|\mathbf{w}_{t+1/2}\|^2 &= \|\sum_{i=1}^{t} \alpha_{t+1/2}(i)\Phi(\mathbf{x}_i)\|^2 \\ &= \sum_{i,j\leq t} \alpha_{t+1/2}(i)\alpha_{t+1/2}j\Phi(\mathbf{x}_i)\cdot\Phi(\mathbf{x}_j) \\ &= \alpha_{t+1/2}^\top K \alpha_{t+1/2}\end{aligned}$$

Following these lines, we can provide the following *kernelized* version of the SGD algorithm:

Algorithm 1 Kernelized SGD for Learning

0: **Input:** Stochastic sample $\{\mathbf{x}_t, y_t\}_{t=1}^T$ drawn IID, an kernel function k defining an RKHS and a sequence of learning rates $\{\eta_t\}$

SET $\mathbf{w}_1 = 0$.

for $t = 1, 2 \dots T$ **do**

Let $\alpha' = \ell_t'(\langle \mathbf{w}_t; \Phi(\mathbf{x}_t)\rangle, y_t)$ $\quad \%\langle \mathbf{w}_t; \Phi(\mathbf{x}^{(t)})\rangle = (\alpha_t K)_t$

Update and Project:

$$\alpha_{t+1/2}(i) = \begin{cases} \alpha_t(i) & i \leq t-1 \\ \eta_t \alpha' & i = t \end{cases} \qquad \%\mathbf{w}_{t+1/2} = \sum \alpha_{t+1/2}(i)\Phi(\mathbf{x}_i)$$

$$\alpha_t = \frac{1}{\sqrt{\alpha_{t+1/2}^\top K \alpha_{t+1/2}}}\alpha_{t+1/2} \quad \%\mathbf{w}_{t+1} = \frac{\mathbf{w}_{t+1/2}}{\|\mathbf{w}_{t+1/2}\|}.$$

end for

return $\bar{\mathbf{w}}_T = \frac{1}{T}\sum_{t=1}^T \mathbf{w}_t$.

References

[1] Sebastian Mika, Bernhard Schölkopf, Alex J Smola, Klaus-Robert Müller, Matthias Scholz, and Gunnar Rätsch. Kernel pca and de-noising in feature spaces. In *Advances in neural information processing systems*, pages 536–542, 1999.

[2] Bernhard Scholkopf and Alexander J Smola. *Learning with kernels: support vector machines, regularization, optimization, and beyond*. MIT press, 2001.