Lecture 16: Kernels and Feature Extraction

CS4787/5777 — Principles of Large-Scale Machine Learning Systems

Today we'll talk about a **powerful method** for scaling and improving the expressiveness of linear models: kernel methods.

Kernel Methods: The Essential Idea

Imagine we are training a (simple 2-class) linear model with $w \in \mathbb{R}^d$ and total loss

$$f(w) = \frac{1}{n} \sum_{i=1}^{n} \ell(x_i^T w; y_i).$$

SGD for this task has an update step at time t that samples $\tilde{i}_t$ uniformly at random and sets

$$w_{t+1} = w_t - \alpha_t \ell'(x_{\tilde{i}_t}^T w_t; y_{\tilde{i}_t}) x_{\tilde{i}_t}.$$

What is the computational cost of running T of these updates?

Answer: $O(Td)$ - **Sometimes, d is very large**, with $d \gg n \rightarrow$ very expensive to compute!

Goal: lower this cost.

Let's look at the SGD update again

$$w_{t+1} = w_t - \alpha_t \ell'(x_{\tilde{i}_t}^T w_t; y_{\tilde{i}_t}) x_{\tilde{i}_t}.$$

Lets keep track of another vector in $\mathbb{R}^n$

$$u_t = \begin{pmatrix} u_{t,1} \\ u_{t,2} \\ \vdots \\ u_{t,n} \end{pmatrix}$$

such that if we initialize $w_0 = 0$, w_t is always in the span of the x_i. That is, it can always be written as a linear combination of the x_i. That is, for some constants $u_{i,t} \in \mathbb{R}$, we can always write

$$w_t = \sum_{i=1}^{n} u_{i,t} x_i$$

where the **values $u_{i,t}$ depends on how often example i got selected and the corresponding ℓ' at those iterations.**

With this setup, the SGD update

$$w_{t+1} = w_t - \alpha_t \ell'(x_{\tilde{i}_t}^T w_t; y_{\tilde{i}_t}) x_{\tilde{i}_t}.$$

can be implicitly achieved by updating the vector u_t using

$$\begin{aligned} u_{\tilde{i}_t,t+1} &= u_{\tilde{i}_t,t} - \alpha_t \ell'(x_{\tilde{i}_t}^T w_t; y_{\tilde{i}_t}) \\ &= u_{\tilde{i}_t,t} - \alpha_t \ell' \left(x_{\tilde{i}_t}^T \left(\sum_{i=1}^n u_{i,t} x_i \right); y_{\tilde{i}_t} \right) \\ &= u_{\tilde{i}_t,t} - \alpha_t \ell' \left(\sum_{i=1}^n u_{i,t} x_{\tilde{i}_t}^T x_i; y_{\tilde{i}_t} \right) \end{aligned}$$

with $u_{i,t+1} = u_{i,t}$ for $i \neq \tilde{i}_t$, and

$$w_t = \sum_{i=1}^n u_{i,t} x_i$$

Our implicit update was:

$$u_{\tilde{i}_t,t+1} = u_{\tilde{i}_t,t} - \alpha_t \ell' \left(\sum_{i=1}^n u_{i,t} x_{\tilde{i}_t}^T x_i; y_{\tilde{i}_t} \right)$$

If we just run this naively, what is the computational cost of running T of these updates?

It would be $O(nTd)$.

Previously, if we just ran the SGD update for T iterations, it would cost $O(Td)$.

So this update doesn't seem to be faster...but is there something clever we can do to speed it up?

Let's look at the update again.

$$u_{\tilde{i}_t,t+1} = u_{\tilde{i}_t,t} - \alpha_t \ell' \left(\sum_{i=1}^n u_{i,t} x_{\tilde{i}_t}^T x_i; y_{\tilde{i}_t} \right)$$

Most of the work is happening in this red dot product between two examples.

But these these dot products are going to repeat many times! In fact there are only n^2 of them. So, we can pre-compute them and cache them.

Let G be the matrix defined by $G_{i,j} = x_i^T x_j$. This is usually called the **Gram matrix**. Then we can write our update as

$$u_{\tilde{i}_t,t+1} = u_{\tilde{i}_t,t} - \alpha_t \ell' \left(\sum_{i=1}^{n} G_{\tilde{i}_t,i} u_{i,t}; y_{\tilde{i}_t} \right).$$

$$u_{\tilde{i}_t,t+1} = u_{\tilde{i}_t,t} - \alpha_t \ell' \left(\sum_{i=1}^{n} G_{\tilde{i}_t,i} u_{i,t}; y_{\tilde{i}_t} \right).$$

The computational cost of computing **1** entry of the gram matrix is just $O(d)$: $G_{i,j} = x_i^T x_j$. And for all of G, $O(n^2 d)$.

If we just run this naively, what is the computational cost of pre-computing the Gram matrix and running T of these updates?

It is indeed $O(dn^2 + Tn)$

Takeaway: If $T \gg n^2$ and $d \gg n$, this approach can be faster.

When can we make this **even faster**?

- A: When we have a way of computing $x_i^T x_j$ in time less than proportional to d.

This commonly happens when we have some examples x_i that aren't necessarily stored as vectors in $\mathbb{R}^d$, but are stored in some format that we can map to vectors.

- i.e. we have not done feature extraction yet, but just have our original data examples (text, images, etc.) not mapped into a vector space yet
- Here, we want to run a linear model on vectors $\phi(x_i)$, where ϕ represents the **feature embedding** from the "raw" example space into the feature vector space $\mathbb{R}^d$
- In some cases, we can compute $\phi(x_i)^T \phi(x_j)$ directly as some function $K(x_i, x_j)$. This function is called a **kernel**!
- If it is faster for us to compute the kernel K than to compute the feature mappings ϕ and the dot products in $\mathbb{R}^d$, then we can get a speedup by computing the kernel function!

Learning with kernels

In this pre-feature-embedding setting, we effectively want to solve the learning problem

$$f(w) = \frac{1}{n} \sum_{i=1}^{n} \ell(w^T \phi(x_i); y_i).$$

We can do this with the Gram matrix by pre-computing G such that $G_{i,j} = K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ and using SGD update

$$u_{\tilde{i}_t,t+1} = u_{\tilde{i}_t,t} - \alpha_t \ell' \left(\sum_{i=1}^{n} G_{\tilde{i}_t,i} u_{i,t}; y_{\tilde{i}_t} \right)$$

This approach is sometimes called the **kernel trick**.

Pause

Designing learning tasks with kernels

Usually, rather than reasoning about (and designing and computing) feature embeddings ϕ, we reason directly about kernel functions K.

Which functions K are kernels? A kernel (also sometimes called a positive definite kernel) over a set $\mathcal{X}$ ($\mathcal{X}$ here represents the set that the "raw" examples live in) is a symmetric function $K(x, y) : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. By symmetric, we mean that $K(x, y) = K(y, x)$ for all x and y in $\mathcal{X}$.

To be a kernel, K must satisfy the condition that for any $x_1, x_2, \ldots, x_n \in \mathcal{X}$, and for any scalars $c_1, c_2, \ldots, c_n \in \mathbb{R}$

$$\sum_{i=1}^{n} \sum_{j=1}^{n} c_i c_j K(x_i, x_j) \geq 0.$$

This is equivalent to saying that if we define the matrix $\mathbf{K} \in \mathbb{R}^{n \times n}$ by $\mathbf{K}_{i,j} = K(x_i, x_j)$, then K will be a positive semidefinite matrix.

K is a positive semidefinite matrix:

- i.e. for any vector $c \in \mathbb{R}^n$, $c^T \mathbf{K} c \geq 0$
- This is also equivalent to saying that any eigenvalue of $\mathbf{K}$ must be nonnegative

(Occasionally you will see authors distinguish between positve definite kernels and positive semidefinite kernels, which correspond to positive definite and positive semidefinite matrices respectively in this condition. But for this class and for most practical machine learning we won't depend on this distinction.)

What do kernels represent?

Usually $K(x, y)$ represents how similar x and y are, where more similar objects will have larger values of $K(x, y)$ and less similar objects will have smaller values.

One very popular kernel is the **radial basis function** kernel or RBF kernel, sometimes also called the Gaussian kernel. The RBF kernel is over a Euclidean space $\mathcal{X} = \mathbb{R}^d$ and takes the form

$$K(x, y) = \exp\left(-\gamma \|x - y\|^2\right).$$

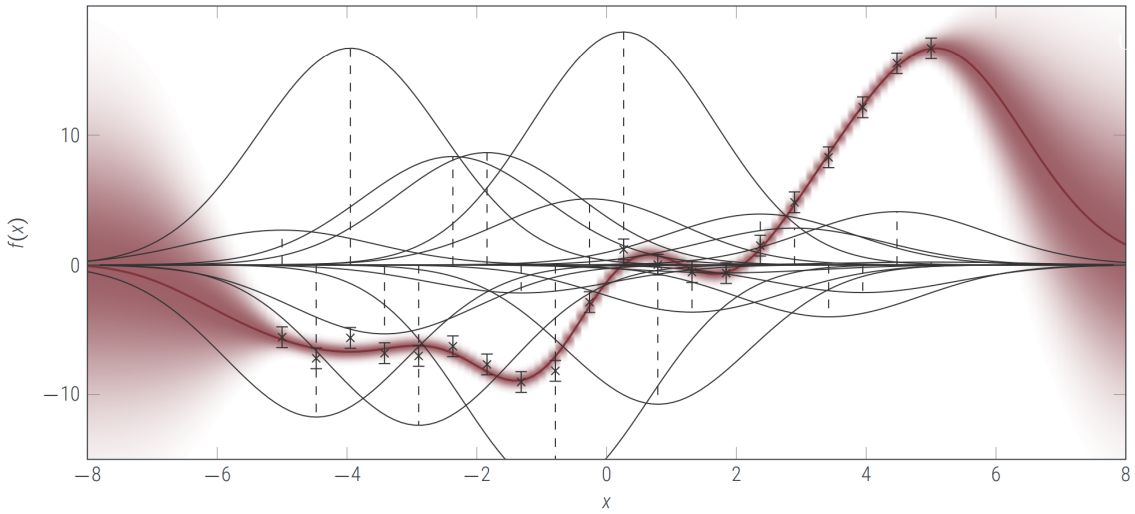
It's easy to see that $0 < K(x, y) \leq 1$ and $K(x, y) = 1$ if and only if $x = y$. These two properties informally mean that this kernel is expressing a similarity between x and y that ranges between 0 and 1, where 1 is the most similar (i.e. literally identical objects $x = y$).

Kernel regression under the RBF kernel

For some learned coefficients $w \in \mathbb{R}^n$, RBF kernel regression results in the following predictor on some arbitrary point x

$$f(x) = w^T K(x, X) = \sum_{i=1}^{n} w_i K(x, x_i) = \sum_{i=1}^{n} w_i \exp\left(-\gamma \|x - x_i\|^2\right)$$

Each prediction is a linear combination of a bunch of Gaussians.



(Figure credit: page 19 in Philipp Hennig's slides)

Question: What other kernels have you seen or used in your work? Do these kernels have the property that $0 < K(x, y) \leq 1$ with $K(x, y) = 1$ if and only if $x = y$?

Grand list of Kernels

- RBF
- Polynomial kernels: $(\gamma \cdot x^T y + r)^d$
- Sigmoid: $\tanh(\gamma \cdot x^T y + r)$
- Linear kernel: $x^T y$
- Kronecker delta:

$$K(x,y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

A kernel determines a feature map.

Important property of kernels: for every kernel over a set $\mathcal{X}$, *there exists* a real Hilbert space $\mathcal{H}$ *and* a feature map $\phi : \mathcal{X} \to \mathcal{H}$ such that for any $x, y \in \mathcal{X}$,

$$K(x,y) = \phi(x)^T \phi(y) = \langle \phi(x), \phi(y) \rangle.$$

(A Hilbert space is just a potentially infinite-dimensional generalization of Euclidean space that supports a dot product/inner product operation with the same properties as the Euclidean dot product.)

That is, every kernel is just a dot product in a (possibly infinite-dimensional) transformed space.

This fact also holds in reverse: **every feature map determines a kernel**.

Constructing kernels.

An important and very useful property of kernels is that we can construct kernels by combining other kernels.

Given any kernels K_1 and K_2 over a set $\mathcal{X}$ with corresponding features maps ϕ_1 and ϕ_2 mapping onto sets of dimension D_1 and D_2 respectively, any positive semi-definite matrix A, any scalar $c \geq 0$, and any function $f : \mathcal{X} \to \mathbb{R}$, the following are kernels.

Our Newly Constructed Kernel Function	Corresponding Feature Map	Dimension of ϕ
$K(x,y) = x^T A y$	$\phi(x) = \sqrt{A} \cdot x$	d (if $\mathcal{X} = \mathbb{R}^d$)
$K(x,y) = c \cdot K_1(x,y)$	$\phi(x) = \sqrt{c} \cdot \phi_1(x)$	D_1
$K(x,y) = K_1(x,y) + K_2(x,y)$	$\phi(x) = \begin{bmatrix} \phi_1(x) \\ \phi_2(x) \end{bmatrix}$	$D_1 + D_2$
$K(x,y) = K_1(x,y) \cdot K_2(x,y)$	$\phi(x) = \phi_1(x) \otimes \phi_2(x)$	$D_1 \cdot D_2$
$K(x,y) = f(x) \cdot K_1(x,y) \cdot f(y)$	$\phi(x) = \phi_1(x) \cdot f(x)$	D_1
$K(x,y) = \exp(K_1(x,y))$	(omitted for space)	∞

Constructing kernels.

These rules let us easily construct a wide range of highly interpretive kernels.

But, as a trade-off, the dimension of the feature map ϕ quickly becomes very large as we apply the rules. It can even become infinite! This can make

- computing directly with the mapped features costly or even impossible
- using the kernel trick particularly important when we construct kernels in this way!

Tradeoffs when Learning with Kernels

If we want to learn with a kernel where $K(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle$, then we have two broad options: to run the original SGD update step

$$w_{t+1} = w_t - \alpha_t \ell'(w^T \phi(x_{\tilde{i}_t}); y_{\tilde{i}_t}) \cdot \phi(x_{\tilde{i}_t}),$$

or the other option in terms of u

$$u_{\tilde{i}_t, t+1} - \alpha_t \ell' \left(\sum_{i=1}^{n} K(x_{\tilde{i}_t}, x_i) \cdot u_{i,t}; y_{\tilde{i}_t} \right).$$

For each option, we can choose whether or not to cache some of the computations. This gives us four simple options:

Option 1. Transform the features on the fly and compute SGD on w, computing $\phi(x_i)$ whenever it is necessary.

Tradeoffs when Learning with Kernels

If we want to learn with a kernel where $K(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle$, then we have two broad options: to run the original SGD update step

$$w_{t+1} = w_t - \alpha_t \ell'(w^T \phi(x_{\tilde{i}_t}); y_{\tilde{i}_t}) \cdot \phi(x_{\tilde{i}_t}),$$

or the other option in terms of u

$$u_{\tilde{i}_t, t+1} - \alpha_t \ell' \left(\sum_{i=1}^{n} K(x_{\tilde{i}_t}, x_i) \cdot u_{i,t}; y_{\tilde{i}_t} \right).$$

For each option, we can choose whether or not to cache some of the computations. This gives us four simple options:

Option 2. Pre-compute and cache the transformed features, forming vectors $z_i = \phi(x_i)$, and compute SGD on w.

Tradeoffs when Learning with Kernels

If we want to learn with a kernel where $K(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle$, then we have two broad options: to run the original SGD update step

$$w_{t+1} = w_t - \alpha_t \ell'(w^T \phi(x_{\tilde{i}_t}); y_{\tilde{i}_t}) \cdot \phi(x_{\tilde{i}_t}),$$

or the other option in terms of u

$$u_{\tilde{i}_t,t+1} - \alpha_t \ell' \left(\sum_{i=1}^{n} K(x_{\tilde{i}_t}, x_i) \cdot u_{i,t}; y_{\tilde{i}_t} \right).$$

For each option, we can choose whether or not to cache some of the computations. This gives us four simple options:

Option 3. Run SGD on u and compute the kernel values $K(x_j, x_i)$ on the fly as they are needed.

Tradeoffs when Learning with Kernels

If we want to learn with a kernel where $K(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle$, then we have two broad options: to run the original SGD update step

$$w_{t+1} = w_t - \alpha_t \ell'(w^T \phi(x_{\tilde{i}_t}); y_{\tilde{i}_t}) \cdot \phi(x_{\tilde{i}_t}),$$

or the other option in terms of u

$$u_{\tilde{i}_t,t+1} - \alpha_t \ell' \left(\sum_{i=1}^{n} K(x_{\tilde{i}_t}, x_i) \cdot u_{i,t}; y_{\tilde{i}_t} \right).$$

For each option, we can choose whether or not to cache some of the computations. This gives us four simple options:

Option 4. Pre-compute the kernel values for each pair of examples, forming the **Gram matrix** G where $G_{i,j} = K(x_i, x_j)$, store it in memory, and then use this to run SGD on u

Tradeoffs when Learning with Kernels

All of these methods will result in the same trained model at the end (assuming no numerical imprecision).

Suppose that

- the original examples are $x_i \in \mathbb{R}^d$
- the transformed features are $\phi(x_i) \in \mathbb{R}^D$
- there are n total training examples, and we run T iterations of SGD.
- the cost of computing a single feature map $\phi(x_i)$ is $\Theta(d \cdot D)$
- the cost of computing the kernel function $K(x_i, y_i)$ is $\Theta(d)$

What are the computational cost and memory required for these four methods, up to a big-Θ analysis? Remember to include the cost of precomputation!

Approach to learning with kernels	Computational cost	Memory use
1. Transform the features on the fly and compute SGD on w		+ training set

Approach to learning with kernels	Computational cost	Memory use
2. Pre-compute and cache the transformed features, and compute SGD on w		+ training set
3. Run SGD on u and compute the kernel values $K(x_j, x_i)$ on the fly		+ training set
4. Pre-compute the Gram matrix, then run SGD on u		+ training set

d -- original example size

D -- transformed example size

Approach to learning with kernels	Computational cost	Memory use
1. Transform the features on the fly and compute SGD on w	$O(dDT)$	$O(D)$ + training set
2. Pre-compute and cache the transformed features, and compute SGD on w	$O(ndD + DT)$	$O(nD)$ + training set
3. Run SGD on u and compute the kernel values $K(x_j, x_i)$ on the fly	$O(ndT)$	$O(n)$ + training set
4. Pre-compute the Gram matrix, then run SGD on u	$O(n^2d + nT)$	$O(n^2)$ + training set

A simple demo comparing two of these options

In [ ]:
```
using Plots
using Statistics
using LinearAlgebra
using Random
```

In [ ]:
```
function gen_y(x::Array{Float64,1})
    if (norm(x - [0.25, 0.75]) < 0.15)
        return -1.0
    elseif (norm(x - [0.75, 0.75]) < 0.15)
        return -1.0
    elseif (x[2] < 0.4) && (norm(x - [0.5,0.6]) < 0.5) && (norm(x - [0.5,0.55]
        return -1.0;
    else
        return 1.0;
    end
end

Random.seed!(123456);
n = 1024;
xs = rand(2,n);
ys = [gen_y(xs[:,i]) for i = 1:n];

xs_test = rand(2,n);
ys_test = [gen_y(xs_test[:,i]) for i = 1:n];
```

In [ ]:
```
colors = [yi == 1.0 ? "lightgreen" : "blue" for yi in ys]
scatter(xs[1,:], xs[2,:], markercolors=colors, legend = false, aspect_ratio=:e
```

In [ ]:
```
num_iters = 20 * n;
alpha = 0.1;
gamma = 100.0; # picked arbitrarily

# derivative of logistic loss
function ell_prime(wx::Float64, y::Float64)
    return -y ./ (1 .+ exp(wx*y));
end

function kernel_function(gamma::Float64, x::Array{Float64,1}, y::Array{Float64
    acc = 0.0
    for i = 1:length(x)
        acc += (x[i] - y[i])^2
    end
    return exp(-gamma * acc);
end

function logistic_regression_gram_matrix(gamma::Float64, alpha::Float64, xs::A
    n = size(xs, 2)
    gram_matrix = [kernel_function(gamma, xs[:,i], xs[:,j]) for i = 1:n, j = 1

    u = zeros(n);

    for t = 1:num_iters
        i = rand(1:n);
        u[i] -= alpha * ell_prime(dot(gram_matrix[:,i], u), ys[i])
    end

    return u;
end

function logistic_regression_kernelfxn(gamma::Float64, alpha::Float64, xs::Arra
    n = size(xs, 2)
    u = zeros(n);

    for t = 1:num_iters
        i = rand(1:n);
        u[i] -= alpha * ell_prime(sum(u[j] * exp(-gamma*sum((xs[k,i] - xs[k,j]
    end

    return u;
end
```

In [ ]:
```
Random.seed!(856235)
@time u_gram = logistic_regression_gram_matrix(gamma, alpha, xs, ys, num_iters
```

In [ ]:
```
Random.seed!(856235)
@time u_kernel = logistic_regression_kernelfxn(gamma, alpha, xs, ys, num_iters
```

In [ ]:
```
# did this get good test accuracy?

function kernel_classifier_prediction(u::Array{Float64,1}, xs_train::Array{Floa
    return sign(sum(u[j] * kernel_function(gamma, x_test, xs_train[:,j]) for j
end
```

```
function kernel_classifier_accuracy(u::Array{Float64,1}, xs_train::Array{Float
    return mean(kernel_classifier_prediction(u, xs_train, xs_test[:,i]) == ys_
end
```

In [ ]:
```
println("train accuracy: ", kernel_classifier_accuracy(u_gram, xs, xs, ys))
println(" test accuracy: ", kernel_classifier_accuracy(u_gram, xs, xs_test, ys_
```

Takeaway: This analysis shows there is a trade-off between the four different methods for kernel learning.

No one method is better than the others in all cases, and you should reason about how your learning task scales before deciding which method to use.

One problematic case: what if we want to use a kernel with an infinite-dimensional feature map?

- For example: the RBF kernel.
- Then $D = \infty$ in the analysis above, and methods (1) and (2) become impossible to run.
- On the other hand, methods (1) and (2) had the best dependence on n, and we would want to consider running one of these when the training set size becomes large.

How can we handle the case of large n and large/infinite D?

Approximate feature maps.

Also called "feature extraction" sometimes.

Idea: approximate a possibly infinite-dimensional feature map ϕ of a kernel with a finite-dimensional feature map $\psi : \mathcal{X} \to \mathbb{R}^D$ such that for all $x, y \in \mathcal{X}$

$$K(x, y) = \phi(x)^T \phi(y) \approx \psi(x)^T \psi(y).$$

Then we can use the approximate feature map ψ to learn with strategy (1) or (2) above.

Question: What are we trading off when we do this?

How do we construct these approximate feature maps?

There are many ways to do it. Here, I'm going to be talking about one way we can do this for the RBF kernel, a method called **Random Fourier features** (Ali Rahimi and Ben Recht, "Random Features for Large-Scale Kernel Machines." *Advances in neural information processing systems, 2008*. This paper won the test of time award at NeurIPS 2017.)

The strategy is straightforward (although the math can get a little tricky).

- First, we write the kernel $K(x, y)$ in terms of an expected value.

- Second, we use our **subsampling principle** to approximate this expected value with a finite sample to within the desired level of accuracy .

- Third, we use this finite sample to construct an approximate feature map.

If we sample ω from a d-dimensional multivariate Gaussian distribution with mean 0 and covariance $2\gamma I$, and independently sample b to be uniform on $[0, 2\pi]$, then we can show that for any $x, y \in \mathbb{R}^d$,

$$K(x,y) = \exp\left(-\gamma\|x-y\|^2\right) = \mathbf{E}_{\omega,b}\left[2 \cdot \cos(\omega^T x + b) \cdot \cos(\omega^T y + b)\right].$$

They did it with a proof...I'll show it in the Jupyter notebook!

In [ ]:
```
gamma = 0.1;

d = 10;
x = rand(d); y = rand(d); # get some random vectors

function supposed_estimator(gamma::Float64, x::Array{Float64,1}, y::Array{Floa
    d = length(x);
    w = randn(d) * sqrt(2*gamma)
    b = 2 * pi * rand();
    return 2 * cos(dot(w,x) + b) * cos(dot(w,y) + b);
end;
```

In [ ]:
```
num_samples = 10000000;

println("kernel function value: ", kernel_function(gamma, x, y));
println("      estimated value: ", mean(supposed_estimator(gamma, x, y) for i :
```

They're pretty close! So we can believe that this is indeed an unbiased estimator of the kernel function.

Now we can apply our subsampling principle to approximate this expected value with a finite sum. Pick some number D and let $\omega_1, b_1, \omega_2, b_2, \ldots, \omega_D, b_D$ be independent random samples of $\omega \sim \mathcal{N}(0, 2\gamma I)$ and $b \sim \text{Uniform}[0, 2\pi]$. Then

$$\begin{aligned} K(x,y) &\approx \frac{1}{D}\sum_{i=1}^{D} 2 \cdot \cos(\omega_i^T x + b_i) \cdot \cos(\omega_i^T y + b_i) \\ &= \sum_{i=1}^{D} \left(\sqrt{\frac{2}{D}} \cdot \cos(\omega_i^T x + b_i)\right) \cdot \left(\sqrt{\frac{2}{D}} \cdot \cos(\omega_i^T y + b_i)\right). \end{aligned}$$

So if we define the feature map $\psi(x)$ such that its ith element is

$$(\psi(x))_i = \sqrt{\frac{2}{D}} \cdot \cos(\omega_i^T x + b_i),$$

then

$$K(x,y) \approx \sum_{i=1}^{D} (\psi(x))_i (\psi(y))_i = \psi(x)^T \psi(y)$$

and we have our approximate feature map! Note that we can also write ψ in terms of matrix multiply as

$$\psi(x) = \sqrt{\frac{2}{D}} \cdot \cos(\Omega x + \mathbf{b}),$$

where Ω and $\mathbf{b}$ denote the matrix and vector

$$\Omega = \begin{bmatrix} \omega_1^T \\ \vdots \\ \omega_D^T \end{bmatrix} \qquad \text{and} \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_D \end{bmatrix}$$

and the $\cos$ operates elementwise.

What thing that we've already seen does this look like?

- It looks like a neural network with random weights and a cosine nonlinearity!

How close will this be to the exact RBF kernel? One way to get a sense of it is to use Hoeffding's inequality. Since the elements of this sum are all of the form $2\cos(\cdot)\cos(\cdot)$, they must have magnitude no greater than 2. As a result, $z_{\min} = -2$, $z_{\max} = 2$, and we get

$$\mathbf{P}\left(\left|K(x,y) - \psi(x)^T\psi(y)\right| \geq a\right) \leq 2\exp\left(-\frac{2Da^2}{(2-(-2))^2}\right) = 2\exp\left(-\frac{Da^2}{8}\right).$$

Takeaway: We can get arbitrarily good accuracy with arbitrarily high probability by increasing D. In their paper, Rahimi and Recht show how they can make a similar statement that holds **for all pairs x, y in a bounded region**, as opposed to just for a specific pair.

This is just one example of a broad class of techniques that can be used for feature extraction for kernels. This adds the following extra techniques to our list of ways to do kernel learning:

1. Pre-compute an approximate feature map (e.g. sample Ω and $\mathbf{b}$ for random Fourier features on RBF) and use it to compute approximate features on the fly to compute SGD in w-space.
2. Pre-compute an approximate feature map, then also pre-compute and cache the transformed approximate features, forming vectors $z_i = \psi(x_i)$. Then run SGD in w-space.

Often, using an approximate feature map like this is the most efficient way of training a model.

- But...not always!

When learning with a kernel, we need to **keep the properties of the problem in mind** to decide how to proceed with learning most efficiently.

In [ ]:

In [ ]: