Recap: Kernel machines are an approach to ML with nonlinear features.

- Pick a nonlinear feature map \( \phi : \mathbb{R}^d \rightarrow \mathbb{F} \)

  ex: \( \phi : [x_1 \ x_2] \rightarrow \{x_1 \ x_2 \ x_1^2 \ x_1x_2 \ x_2^2 \} \)

  \[ \phi : [x_1 \ x_2] \rightarrow \left[ \sqrt{x_1^2 + x_2^2} \ + \ \tan^{-1}\left(\frac{x_2}{x_1}\right) \right] \]

  \[ \phi : [x_1 \ x_2] \rightarrow \left[ \text{all the monomials in } x_1 \text{ and } x_2 \right] \]

- We showed if we want to learn a function of the form
  \[ f(x) = \langle w, \phi(x) \rangle \]
  this can be accomplished using just the kernel function
\[ k(x, y) = \langle \phi(x), \phi(y) \rangle \]

so learning can be done using \( K \) and the kernel matrix

\[ K_{ij} = k(x_i, x_j) \]

In particular, the optimal \( f \) for an ERM takes the form

\[ f(x) = \sum_{i=1}^{n} \alpha_i k(x, x_i) \]

so we just have to learn the \( \alpha \) weights
Last example: logistic regression with non-linear features

$$\omega^* = \arg\min_{\omega} \frac{1}{n} \sum_{i=1}^{n} \log (1 + e^{-y_i \langle \omega, \phi(x_i) \rangle}) + \frac{\lambda}{2} \lVert \omega \rVert^2$$

we know in fact $f(x_i) = \langle \omega^*, \phi(x_i) \rangle$

$$= \sum_{j=1}^{n} \alpha_i^* \kappa(x_i, x_j)$$

$$= \left( \kappa \alpha \right)_i \text{ i^{th} row of } \kappa \alpha$$
so we can predict using \( f(x) = \sum_{i=1}^{n} \alpha_i \ K(x, x_i) \)
if we find the optimal \( \alpha \):

\[
\alpha^* = \arg\min_{\alpha} \frac{1}{n} \sum_{i=1}^{n} \log(1 + e^{-y_i(k\alpha)_i}) + \frac{1}{2} \alpha^T K \alpha
\]

kernel logistic regression (recall \( K \in \mathbb{R}^{n \times n} \))
Question: how can we gain accuracy like kernel methods while having a faster runtime?

An answer:

- note the cost of kernel methods is that of
  1) multiplying by $K \propto O(n^2)$
    - for example in subgradient descent, or GrD
  2) inverting $K \propto O(n^3)$
    - for example in ridge regression

- approximate $K$ by a low-rank matrix

\[ \alpha = (K + n\lambda I)^{-1}y \]
$K \approx CC^T$ where $C \in \mathbb{R}^{n \times r}$

call $\hat{K} = CC^T$ our low-rank approximation.
and replace $K$ in all our algorithms with $\hat{K}$.

Now

1) cost of multiplying by $\hat{K}$ is $O(nr)$
   consider for example $n = 500$ and $r = 1000$

2) cost of inverting $\hat{K} + \lambda n I$ is $O(nr^2)$
   (use the Woodbury identity)
Two approaches to getting these low rank approximations:

1) Nyström approximations (originated in PDE literature)

2) Random feature maps (Recht & Rahimi, 2013)

**Nyström approximations**

Note that $K = K^T$ b/c $K(x_i, x_j) = K(x_j, x_i)$

Write

$$K = \begin{bmatrix}
W & C^T \\
C & \end{bmatrix}$$

where $C$ are $d$ sampled columns and $W$ is the principal submatrix where $C$ and $C^T$ overlap.
note that this is equivalent to selecting
2 ‘landmark points’ \( p_1, \ldots, p_L \) and
comparing all of our training points to these
landmark points.

\[ C_{i,j} = K(x_i, p_j) \]

and now we extend these comparisons to the
unobserved points (the non-landmark points) by
noting that

\[ K = C W C^T \]
where $W^+$ is the Moore-Penrose pseudoinverse of $W$. We call $W^+$ the coupling matrix.

Why is $K \approx CW^+C^T$?
- Assume $W$ is full rank, then $W^+ = W^{-1}$

and

$$CW^+C^T = \begin{bmatrix} W \\ \hline L \end{bmatrix} W^+ \begin{bmatrix} W & L^T \end{bmatrix}$$

$$= \begin{bmatrix} W & L^T \\ \hline L & LW^+L^T \end{bmatrix}$$
so \[ K - CW^+C^T = \begin{bmatrix} 0 & O \\ O & K - LW^+L^T \end{bmatrix} \]

if \[ K = \begin{bmatrix} W & L^T \\ L & K \end{bmatrix} \]

Schur complement of \( W \) in \( K \)

Fact: \( r_K(W) = r(K) \) then \( K - LW^+L^T = 0 \)

Take-away: if \( r_K(W) = r(K) \) then 
\[ K = CW^+C^T \]
Note we could pick any $C$ subset of columns of $R$. 

Still $R = CW^T + CT$.
Nystrom alg

Input: $k$ - kernel function
$X$ - training data
$l$ - # of landmark points

Output: $C$, $W$

Algorithm
1. randomly pick $l$ landmark points $\{p_1, \ldots, p_l\}$
2. construct $C$ by comparing all of the data points to my landmark points
   \[ C_{i,j} = k(x_i, p_j) \]
3. construct $W$ by comparing the landmark points to themselves
   \[ W_{i,j} = k(p_i, p_j) \]

return $C$, $W$
Random feature maps

Motivation:

\[ K = \Phi \Phi^T \quad \text{where} \quad \Phi = \begin{bmatrix} \phi(x_1)^T \\ \vdots \\ \phi(x_n)^T \end{bmatrix} \]

would be a low-rank approximation if \( D \ll n \)

But in general \( D \gg n \) (that's why we work with the kernel \( K \) instead of \( D \) directly)

What we want to do is find an approximate feature map \( \Phi : \mathbb{R}^d \to \mathbb{R}^E \) \( E \ll D \) and \( E \ll n \)
where $\Phi$ is chosen so:

- $E \subseteq n$, $D$
- $\langle \Phi(x), \Phi(y) \rangle \approx k(x, y)$

because then

$$K_{ij} = k(x_i, x_j) \approx \langle \Phi(x_i), \Phi(x_j) \rangle$$

so

$$K \approx \Phi \Phi^T \quad \text{where} \quad \Phi = \begin{bmatrix} \Phi(x_1)^T \\ \vdots \\ \Phi(x_n)^T \end{bmatrix} \in \mathbb{R}^{n \times E}$$

and $E \ll n$ means this is a low-rank approx, so is cheaper to use than $K$.
Q: how to find these approximate feature maps
A: randomness!

For a given kernel $K$, choose $\tilde{\phi}$ so that

$$E \langle \tilde{\phi}(x), \tilde{\phi}(y) \rangle = K(x, y)$$

Ex: if $K(x, y) = e^{-\frac{||x-y||^2}{2\sigma^2}}$

then $\tilde{\phi}(x) = \sum_\omega \cos(\omega^T x + b)$ where $\omega \sim N(0, \sigma^2 I)$ and $b \sim \text{Unif} [-\pi, \pi]$ satisfies

$$E_{\omega, b} \langle \tilde{\phi}(x), \tilde{\phi}(y) \rangle = K(x, y)$$
Another example: polynomial kernel

\[ k(x, y) = \langle x, y \rangle^r \]

Idea: \( \tilde{\phi}(x) = \epsilon^T x \) where \( \epsilon \in \mathbb{R}^d \) is a vector of uniformly random signs, iid \( \epsilon_i \in \{\pm 1\} \)

\[
E \langle \tilde{\phi}(x), \tilde{\phi}(y) \rangle = E \left[ (\epsilon^T x) (\epsilon^T y) \right]
= E \left( \sum_{i,j=1}^{d} x_i y_j \epsilon_i \epsilon_j \right)
= \frac{1}{d} \sum_{i,j=1}^{d} x_i y_j E [\epsilon_i \epsilon_j]
\]
Since the entries of $\varepsilon$ are independent,

\[ E[\varepsilon_i \varepsilon_j] = \begin{cases} 
E[\varepsilon_i] E[\varepsilon_j] = 0 & \text{if } i \neq j \\
E[\varepsilon_i^2] = 1 & \text{if } i = j
\end{cases} \]

\[
E \langle \tilde{\phi}(x), \tilde{\phi}(y) \rangle = \frac{1}{d} \sum_{i,j=1}^{d} x_i y_j E[\varepsilon_i \varepsilon_j] \\
= \frac{1}{d} \sum_{i=1}^{d} x_i y_i = \langle x, y \rangle
\]

Now I use this fact to approximate the polynomial feature map:
- Sample $\varepsilon_1, \varepsilon_1 \in \mathbb{R}^d, \ldots, \varepsilon_t \in \mathbb{R}^d$, i.i.d. random sign vectors
- $\tilde{\phi}(x) = (\varepsilon_1^T x) \ldots (\varepsilon_t^T x)$
with this choice:

\[
\mathbb{E} < \tilde{\phi}(x), \tilde{\phi}(y) > = \mathbb{E} \left[ \prod_{i=1}^{r} (e_i^T x)(e_i^T y) \right] \\
= \prod_{i=1}^{r} \mathbb{E} [e_i^T x y^T e_i] = \prod_{i=1}^{r} < x, y > = < x, y >^r
\]

Alg:
Sample \( w_1 \in \mathbb{R}^{D \times d} \) so all its entries are i.i.d. random signs.
Sample \( w_2, \ldots, w_r \) i.i.d. in the same manner.

\[\tilde{\phi}(x) = \frac{1}{\sqrt{D}} (w_1 x) \circ (w_2 x) \circ \ldots \circ (w_r x) \in \mathbb{R}^D\]
Note that with this choice,

\[ E \langle \tilde{\phi}(x), \tilde{\phi}(y) \rangle = E \sum_{i=1}^{D} \tilde{\phi}(x)_i \tilde{\phi}(y)_i \]

\[ = \sum_{i=1}^{D} E \left[ \tilde{\phi}(x)_i \tilde{\phi}(y)_i \right] \]

\[ = \frac{1}{D} \sum_{i=1}^{D} \frac{1}{D} E \left[ \left( w_{ij} \right)_i \times \left( w_{ji} \right)_i \right] \]

\[ = \frac{1}{D} \sum_{i=1}^{D} \frac{1}{D} \langle x, y \rangle^r \]

\[ = \langle x, y \rangle^r \]
A more general approach than kernel methods to learning nonlinear functions is that of neural networks.
Recall w/ kernel methods, we

1) pick a feature map $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^D$

2) results in a kernel $\langle \phi(x), \phi(y) \rangle$

3) know the resulting ERM is convex

and takes the form

$$f(x) = \sum_{i=1}^{n} \alpha_i \cdot K(x, x_i)$$

4) solve for $\alpha^*$ again is a convex prob

w/ any of the methods are talked about before and we know

$$f(x_t) \leq f(x^*) + \varepsilon$$ if we run long enough
Drawbacks:
1. We have to do a good job of selecting our kernel $K$.
2. We have to keep around our training data to predict.

Another approach is neural nets. This is more general and learns our features for use.
What is a neural network?
- Motivated by biological neural networks (collections of neurons)

\[ \text{output} = f(\text{input}) \]

- Neurons are connected together as a DAG (directed acyclic graph), no feedback loops or self-loops

[Diagram of neural networks with directed edges and nodes labeled as inputs and outputs]
Given a particular DAG (architecture of the neural network) we learn parameters for the neurons in the neural net to minimize our RERM objective

$$\omega = \arg \min_{\omega} \frac{1}{n} \sum_{i=1}^{n} l(f(x_i; \omega), y_i) + R(\omega)$$

The architecture determines how many parameters are in \( \omega \), and the way in which the function \( f \) depends on these parameters.
Architecture Choice I: Fully-connected neural networks
- The neurons are divided into an input layer, an output layer, and one or more hidden layers.
- All the neurons in layer \( l \) are connected to all the neurons in layer \( l+1 \).
Binary Classification (Logistic Regression)

Inputs are in $\mathbb{R}^3$

\[ o = f \left( \sum_{i=1}^{3} \omega x_i + b \right) = \sigma \left( \frac{3}{2} \sum_{i=1}^{3} \omega x_i + b \right) \]

\[ \sigma(a) = \frac{1}{1 + e^{-a}} \]
$x, y \in \mathbb{R}^3$, $\mathbb{R}^3$ inputs
$K \in \mathbb{R}^{n \times n}$ costs $\mathcal{O}(n^2)$ mult $\mathcal{O}(n^3)$ invert

$K \approx \Phi \Phi^T$ where $\Phi \in \mathbb{R}^{n \times E}$

randomized, low-dim $(E)$ costs $\mathcal{O}(nE)$ mult
approx feature map

$\Phi \Phi^T$ where $\Phi \in \mathbb{R}^{n \times D}$ where $D \geq n$
costs $\mathcal{O}(nD) = \mathcal{O}(n^2)$
deterministic, high-dim $(D)$
exact feature map
costs $\mathcal{O}(nD^2) = \mathcal{O}(n^3)$