The Kernel Trick, Gram Matrices, and Feature Extraction

CS6787 Lecture 4 — Fall 2017

Momentum for Principle Component Analysis

CS6787 Lecture 3.1 — Fall 2017

Principle Component Analysis

- Setting: find the dominant eigenvalue-eigenvector pair of a positive semidefinite symmetric matrix **A**.

$$u_1 = \arg\max_x \frac{x^T A x}{x^T x}$$

- Many ways to write this problem, e.g.

$\|B\|_F$ is *Frobenius norm*

$$\|B\|_F^2 = \sum_i \sum_j B_{i,j}^2$$

$$\sqrt{\lambda_1} u_1 = \arg\min_x \|xx^T - A\|_F^2$$

PCA: A Non-Convex Problem

- PCA is **not convex** in any of these formulations
- **Why?** Think about the solutions to the problem: **u** and **–u**
  - Two distinct solutions → can't be convex
- Can we still use momentum to run PCA more quickly?

Power Iteration

- Before we apply momentum, we need to choose what base algorithm we're using.

- Simplest algorithm: **power iteration**
  - Repeatedly multiply by the matrix A to get an answer

$$x_{t+1} = Ax_t$$

Why does Power Iteration Work?

- Let eigendecomposition of A be $A = \sum_{i=1}^{n} \lambda_i u_i u_i^T$
    - For $\lambda_1 > \lambda_2 \geq \lambda_1 \geq \cdots \geq \lambda_n$

- PI converges **in direction** because cosine-squared of angle to $\mathbf{u_1}$ is

$$\cos^2(\theta) = \frac{(u_1^T x_t)^2}{\|x_t\|^2} = \frac{(u_1^T A^t x_0)^2}{\|A^t x_0\|^2}$$

What about a more general algorithm?

- Use both current iterate, and **history of past iterations**

$$x_{t+1} = \alpha_t A x_t + \beta_{t,1} x_{t-1} + \beta_{t,2} x_{t-2} + \cdots + \beta_{t,t} x_0$$

  - for fixed parameters α and β

- **What class of functions can we express in this form?**

- Notice: x_t is always a degree-**t** polynomial in **A** times x_0
  - Can prove by induction that we can express **ANY polynomial**

Power Iteration and Polynomials

- Can also think of power iteration as a degree-**t** polynomial of **A**

$$x_t = A^t x_0$$

- Is there a better degree-**t** polynomial to use than $f_t(x) = x^t$?
  - If we use a different polynomial, then we get

$$x_t = f_t(A)x_0 = \sum_{i=1}^{n} f_t(\lambda_i) u_i u_i^T x_0$$

  - Ideal solution: choose polynomial with zeros at all non-dominant eigenvalues

Chebyshev Polynomials Again

- It turns out that Chebyshev polynomials solve this problem.

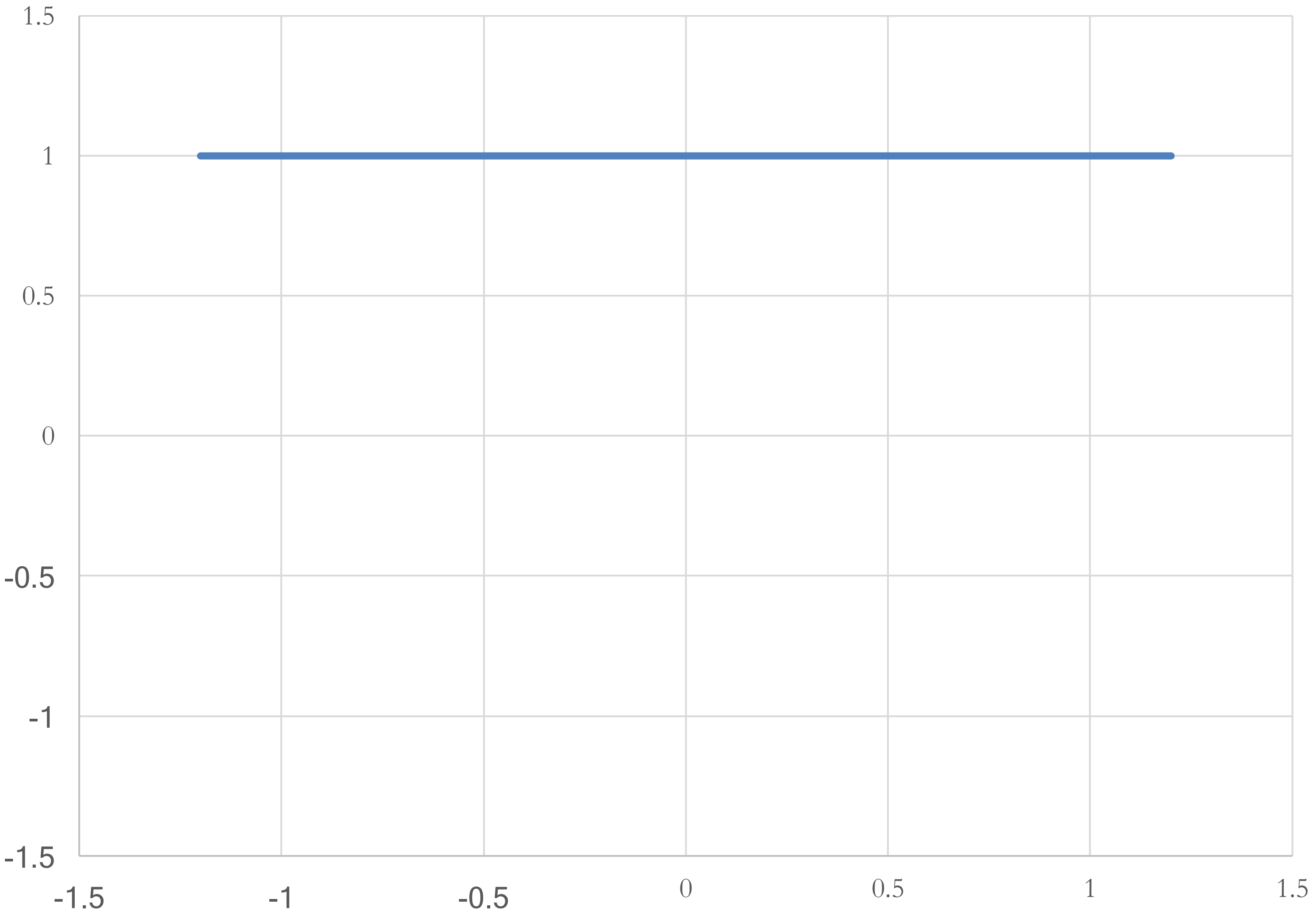
- Recall: $T_0(x) = 0$, $T_1(x) = x$ and

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$$

- Nice properties:

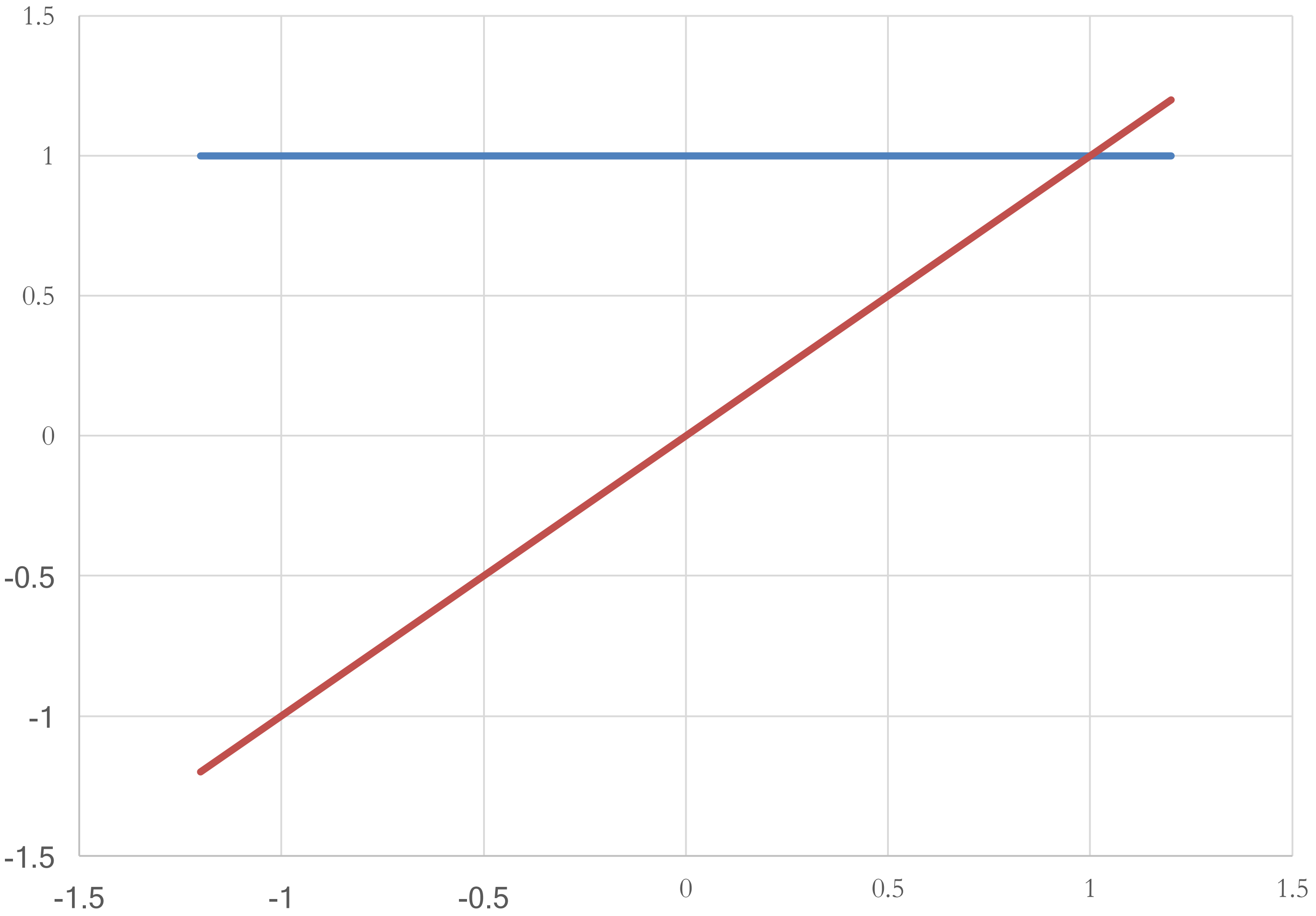
$$|x| \leq 1 \Rightarrow |T_n(x)| \leq 1$$

Chebyshev Polynomials



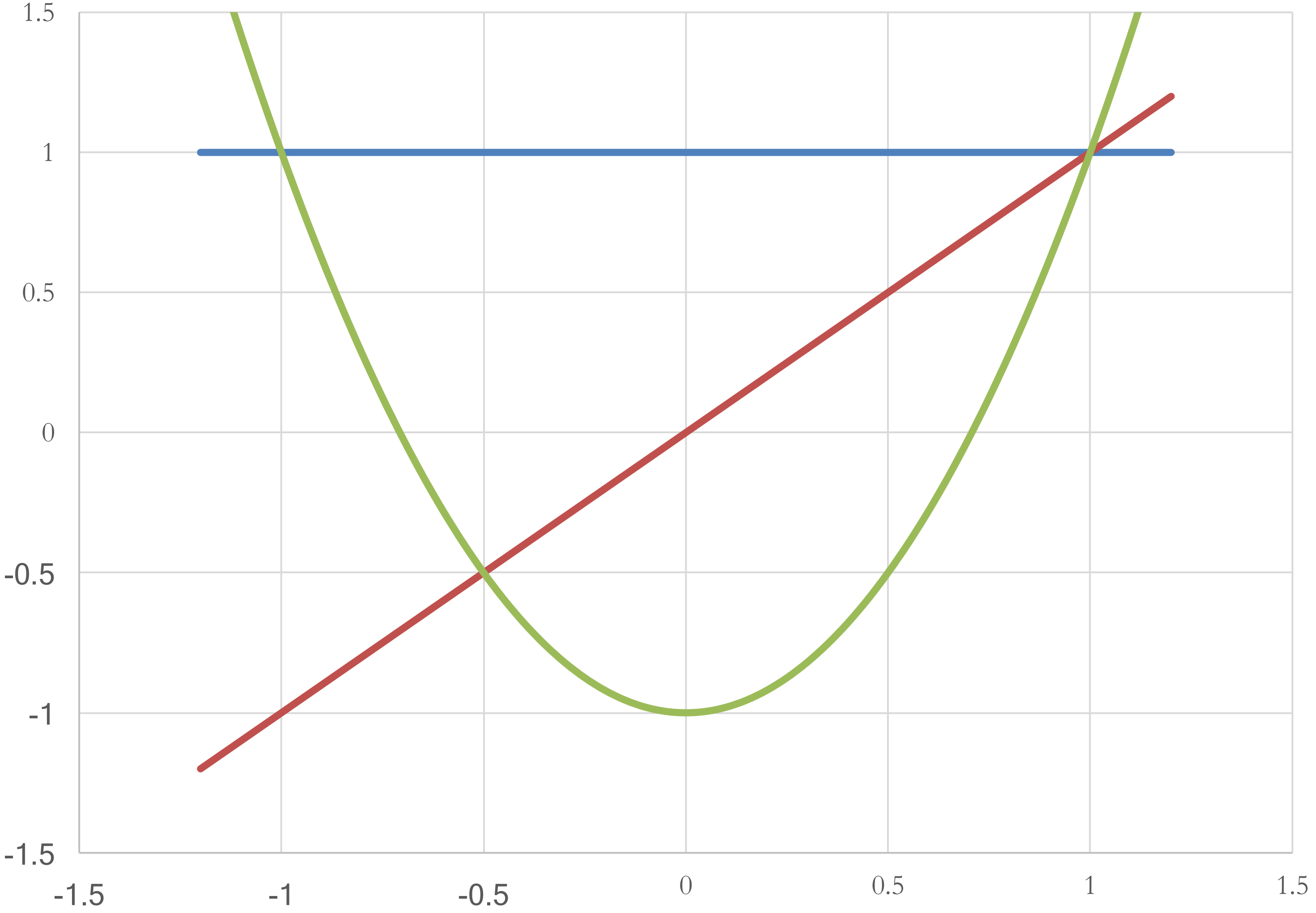
$$T_0(u) = 1$$

Chebyshev Polynomials



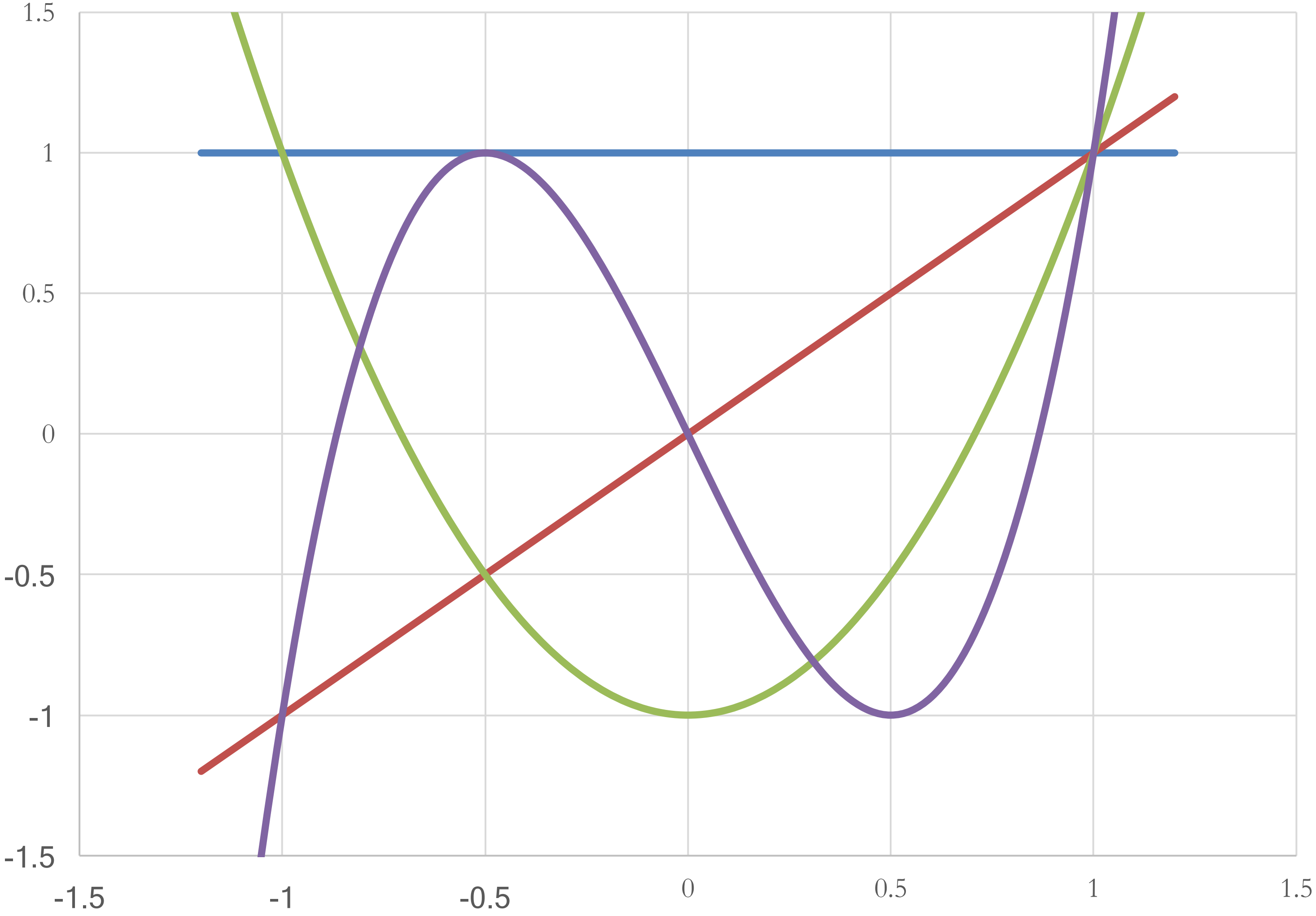
$$T_1(u) = u$$

Chebyshev Polynomials

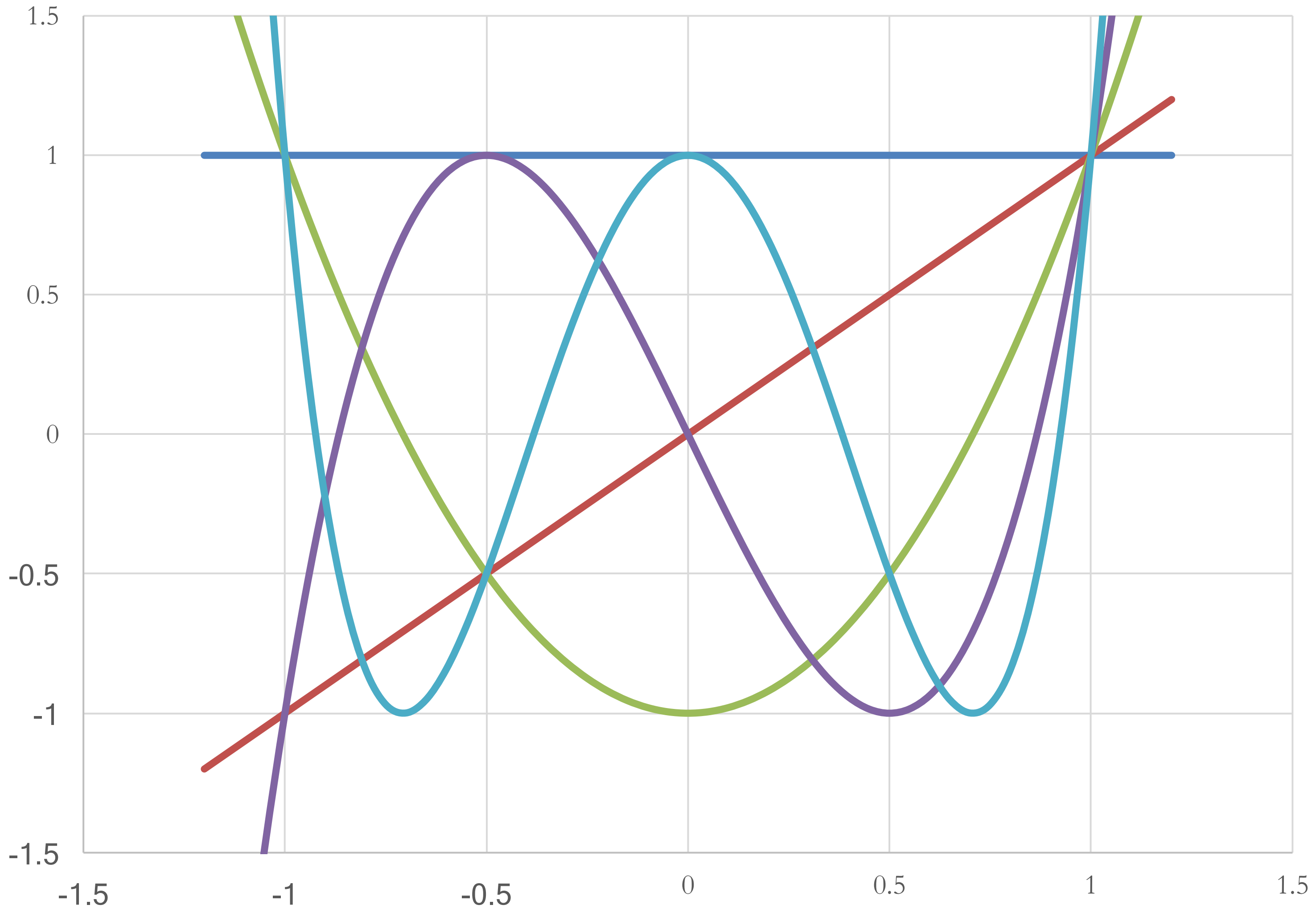


$$T_2(u) = 2u^2 - 1$$

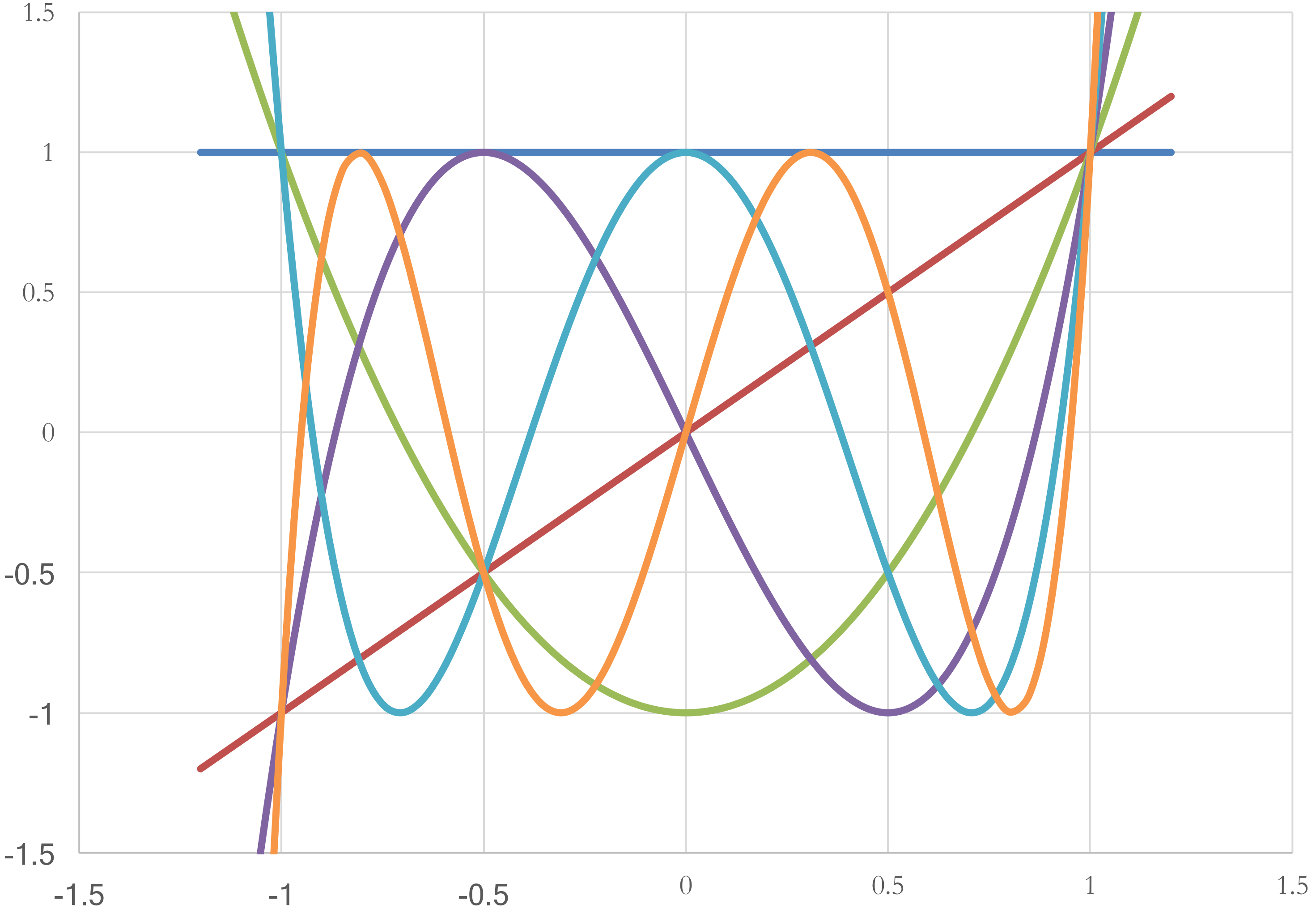
Chebyshev Polynomials



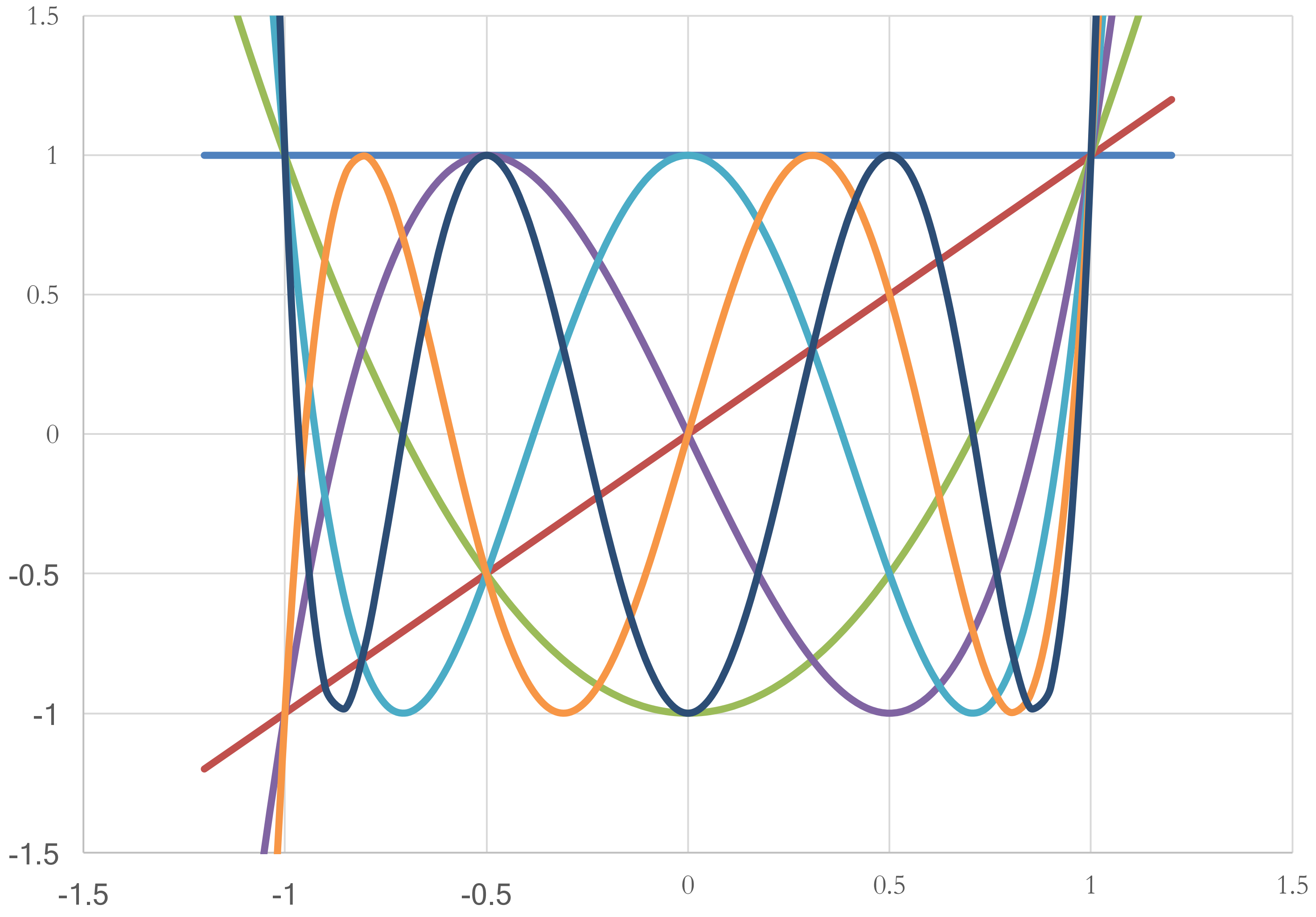
Chebyshev Polynomials



Chebyshev Polynomials



Chebyshev Polynomials



Chebyshev Polynomials Again

- It turns out that Chebyshev polynomials solve this problem.

- Recall: $T_0(x) = 0$, $T_1(x) = x$ and

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$$

- Nice properties:

$$|x| \leq 1 \Rightarrow |T_n(x)| \leq 1 \qquad T_n(1+\epsilon) \approx \Theta\left(\left(1+\sqrt{2\epsilon}\right)^n\right)$$

Using Chebyshev Polynomials

- So we can choose our polynomial f in terms of T
  - Want: $f_t(\lambda_1)$ to be as large as possible, subject to $|f_t(\lambda)| < 1$ for all $|\lambda| < \lambda_2$

- To make this work, set

$$f_n(x) = T_n\left(\frac{x}{\lambda_2}\right)$$

Convergence of Momentum PCA

$$\frac{x_t}{\|x_t\|} = \frac{\sum_{i=1}^{n} T_t\left(\frac{\lambda_i}{\lambda_2}\right) u_i u_i^T x_0}{\sqrt{\sum_{i=1}^{n} T_t^2\left(\frac{\lambda_i}{\lambda_2}\right) (u_i^T x_0)^2}}$$

- Cosine-squared of angle to dominant component:

$$\cos^2(\theta) = \frac{(u_1^T x_t)^2}{\|x_t\|^2} = \frac{T_t^2\left(\frac{\lambda_1}{\lambda_2}\right) (u_1^T x_0)^2}{\sum_{i=1}^{n} T_t^2\left(\frac{\lambda_i}{\lambda_2}\right) (u_i^T x_0)^2}$$

Convergence of Momentum PCA (continued)

$$\cos^2(\theta) = \frac{(u_1^T x_t)^2}{\|x_t\|^2} = \frac{T_t^2\left(\frac{\lambda_1}{\lambda_2}\right)(u_1^T x_0)^2}{\sum_{i=1}^n T_t^2\left(\frac{\lambda_i}{\lambda_2}\right)(u_i^T x_0)^2}$$

$$= 1 - \frac{\sum_{i=2}^n T_t^2\left(\frac{\lambda_i}{\lambda_2}\right)(u_i^T x_0)^2}{\sum_{i=1}^n T_t^2\left(\frac{\lambda_i}{\lambda_2}\right)(u_i^T x_0)^2}$$

$$\geq 1 - \frac{\sum_{i=2}^n (u_i^T x_0)^2}{T_t^2\left(\frac{\lambda_1}{\lambda_2}\right)(u_1^T x_0)^2} = 1 - \Omega\left(T_t^{-2}\left(\frac{\lambda_1}{\lambda_2}\right)\right)$$

Convergence of Momentum PCA (continued)

$$\cos^2(\theta) \geq 1 - \Omega\left(T_t^{-2}\left(\frac{\lambda_1}{\lambda_2}\right)\right) = 1 - \Omega\left(T_t^{-2}\left(1 + \frac{\lambda_1 - \lambda_2}{\lambda_2}\right)\right)$$
$$= 1 - \Omega\left(\left(1 + \sqrt{2\frac{\lambda_1 - \lambda_2}{\lambda_2}}\right)^{-2t}\right)$$

- Recall that standard power iteration had:

$$\cos^2(\theta) = 1 - \Omega\left(\left(\frac{\lambda_2}{\lambda_1}\right)^{2t}\right) = 1 - \Omega\left(\left(1 + \frac{\lambda_1 - \lambda_2}{\lambda_2}\right)^{-2t}\right)$$

  - So the momentum rate is asymptotically faster than power iteration

Questions?

The Kernel Trick, Gram Matrices, and Feature Extraction

CS6787 Lecture 4 — Fall 2017

Basic Linear Models

- For classification using model vector **w**

$$\text{output} = \text{sign}(w^T x)$$

- Optimization methods vary; here's logistic regression $(y_i \in \{-1, 1\})$

$$\text{minimize}_w \ \frac{1}{n} \sum_{i=1}^{n} \log\left(1 + \exp(-w^T x_i y_i)\right)$$

Benefits of Linear Models

- **Fast classification**: just one dot product
- **Fast training/learning**: just a few basic linear algebra operations
- **Drawback: limited expressivity**
  - Can only capture linear classification boundaries → bad for many problems
- How do we let linear models **represent a broader class of decision boundaries**, while **retaining the systems benefits**?

The Kernel Method

- Idea: in a linear model we can think about the **similarity** between two training examples **x** and **y** as being

$$x^T y$$

  - This is related to the rate at which a random classifier will separate **x** and **y**

- Kernel methods replace this dot-product similarity with an arbitrary **Kernel function** that computes the similarity between **x** and **y**

$$K(x, y) : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$$

Kernel Properties

- **What properties do kernels need to have to be useful for learning?**

- Key property: kernel must be **symmetric** $K(x, y) = K(y, x)$

- Key property: kernel must be **positive semi-definite**

$$\forall c_i \in \mathbb{R}, x_i \in \mathcal{X}, \sum_{i=1}^{n} \sum_{j=1}^{n} c_i c_j K(x_i, x_j) \geq 0$$

  - Can check that the dot product has this property

Facts about Positive Semidefinite Kernels

- Sum of two PSD kernels is a PSD kernel

$$K(x, y) = K_1(x, y) + K_2(x, y) \text{ is a PSD kernel}$$

- Product of two PSD kernels is a PSD kernel

$$K(x, y) = K_1(x, y)K_2(x, y) \text{ is a PSD kernel}$$

- Scaling by any function on both sides is a kernel

$$K(x, y) = f(x)K_1(x, y)f(y) \text{ is a PSD kernel}$$

Other Kernel Properties

- Useful property: kernels are often **non-negative**

$$K(x, y) \geq 0$$

- Useful property: kernels are often **scaled** such that

$$K(x, y) \leq 1, \ \text{and} \ K(x, y) = 1 \Leftrightarrow x = y$$

  - These properties capture the idea that the kernel is expressing the similarity between **x** and **y**

Common Kernels

- **Gaussian kernel/RBF kernel**: de-facto kernel in machine learning

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

- We can validate that this is a kernel
  - Symmetric? ✅
  - Positive semi-definite? ✅ **WHY?**
  - Non-negative? ✅
  - Scaled so that **K(x,x) = 1**? ✅

Common Kernels (continued)

- **Linear kernel**: just the inner product $K(x, y) = x^T y$

- **Polynomial kernel:** $K(x, y) = (1 + x^T y)^p$

- **Laplacian kernel:** $K(x, y) = \exp\left(-\beta \|x - y\|_1\right)$

- Last layer of a neural network:

  if last layer outputs $\phi(x)$, then kernel is $K(x, y) = \phi(x)^T \phi(y)$

Classifying with Kernels

- An equivalent way of writing a linear model on a training set is

$$\text{output}(x) = \text{sign}\left(\left(\sum_{i=1}^{n} w_i x_i \right)^T x \right)$$

- We can kernel-ize this by replacing the dot products with kernel evals

$$\text{output}(x) = \text{sign}\left(\sum_{i=1}^{n} w_i K(x_i, x) \right)$$

Learning with Kernels

- An equivalent way of writing linear-model logistic regression is

$$\text{minimize}_w \; \frac{1}{n} \sum_{i=1}^{n} \log \left(1 + \exp \left(- \left(\sum_{j=1}^{n} w_j x_j \right)^T x_i y_i \right) \right)$$

- We can kernel-ize this by replacing the dot products with kernel evals

$$\text{minimize}_w \; \frac{1}{n} \sum_{i=1}^{n} \log \left(1 + \exp \left(- \sum_{j=1}^{n} w_j y_i K(x_j, x_i) \right) \right)$$

The Computational Cost of Kernels

- Recall: benefit of learning with kernels is that **we can express a wider class of classification functions**
- Recall: another benefit is **linear classifier learning problems are "easy"** to solve because they are convex, and gradients easy to compute
- **Major cost of learning naively with Kernels**: have to evaluate **K(x, y)**
  - For SGD, need to do this effectively **n** times per update
  - Computationally intractable unless **K** is very simple

The Gram Matrix

- Address this computational problem by **pre-computing the kernel function** for all pairs of training examples in the dataset.

$$G_{i,j} = K(x_i, x_j)$$

- Transforms the learning problem into

$$\text{minimize}_w \ \frac{1}{n} \sum_{i=1}^{n} \log\left(1 + \exp\left(-y_i e_i^T G w\right)\right)$$

  - This is much easier than recomputing the kernel at each iteration

Problems with the Gram Matrix

- Suppose we have **n** examples in our training set.

- **How much memory** is required to store the Gram matrix **G**?

- **What is the cost** of taking the product $\mathbf{G}_i \mathbf{w}$ to compute a gradient?

- What happens if we have **one hundred million training examples**?

Feature Extraction

- Simple case: let's imagine that **X** is a finite set {1, 2, …, k}

- We can define our kernel as a matrix $M \in \mathbb{R}^{k \times k}$

$$M_{i,j} = K(i,j)$$

- Since M is positive semidefinite, it has a square root $U^T U = M$

$$\sum_{i=1}^{k} U_{k,i} U_{k,j} = M_{i,j} = K(i,j)$$

Feature Extraction (continued)

- So if we define a **feature mapping** $\phi(i) = Ue_i$ then

$$\phi(i)^T \phi(j) = \sum_{i=1}^{k} U_{k,i} U_{k,j} = M_{i,j} = K(i,j)$$

- The kernel is **equivalent to a dot product** in some space

- In fact, this is **true for all kernels**, not just finite ones

Classifying with feature maps

- Suppose that we can find a finite-dimensional feature map that satisfies

$$\phi(i)^T \phi(j) = K(i, j)$$

- Then we can simplify our classifier to

$$\begin{aligned} \text{output}(x) &= \text{sign}\left(\sum_{i=1}^{n} w_i K(x_i, x) \right) \\ &= \text{sign}\left(\sum_{i=1}^{n} w_i \phi(x_i)^T \phi(x) \right) = \text{sign}\left(u^T \phi(x) \right) \end{aligned}$$

Learning with feature maps

- Similarly we can simplify our learning objective to

$$\text{minimize}_u \ \frac{1}{n}\sum_{i=1}^{n} \log\left(1 + \exp\left(-u^T\phi(x_i)y_i\right)\right)$$

- Take-away: this is just **transforming the input data, then running a linear classifier in the transformed space**!

- Computationally: **super efficient**
  - As long as we can transform and store the input data in an efficient way

Problems with Feature Maps

- The dimension of the transformed data may be **much larger than the dimension of the original data**.

- Suppose that the feature map is $\phi : \mathbb{R}^d \to \mathbb{R}^D$ and there are **n** examples

- **How much memory is needed** to store the transformed features?

- **What is the cost** of taking the product $u^T \phi(x_i)$ to compute a gradient?

Feature Maps vs. Gram Matrices

- **Systems trade-offs exist here.**

- When number of examples gets very large, **feature maps are better**.

- When transformed feature vectors have high dimensionality, **Gram matrices are better**.

Another Problem with Feature Maps

- Recall: I said there was always a feature map for any kernel such that

$$\phi(i)^T \phi(j) = K(i, j)$$

- But this feature map is **not always finite-dimensional**
  - For example, the Gaussian/RBF kernel has an infinite-dimensional feature map
  - **Many kernels we care about in ML have this property**

- What do we do if ϕ has infinite dimensions?
  - **We can't just compute with it normally!**

Solution: Approximate Feature Maps

- Find a finite-dimensional feature map so that

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

- Typically, we want to find a family of feature maps ϕ_t such that

$$\phi_D : \mathbb{R}^d \to \mathbb{R}^D$$

$$\lim_{D \to \infty} \phi_D(x)^T \phi_D(y) = K(x, y)$$

Types of Approximate Feature Maps

- Deterministic feature maps
  - Choose a fixed-a-priori method of approximating the kernel
  - Generally not very popular because of the way they scale with dimensions

- Random feature maps
  - Choose a feature map at random (typically each feature is independent) such that

$$\mathbf{E}\left[\phi(x)^T \phi(y)\right] = K(x, y)$$

  - Then prove with high probability that over some region of interest

$$\left|\phi(x)^T \phi(y) - K(x, y)\right| \leq \epsilon$$

Types of Approximate Features (continued)

- Orthogonal randomized feature maps
  - Intuition behind this: if we have a feature map where for some i and j

$$e_i^T \phi(x) \approx e_j^T \phi(x)$$

    then we can’t actually learn much from having both features.
  - Strategy: choose the feature map at random, but subject to the constraint that the features be “orthogonal” in some way.

- Quasi-random feature maps
  - Generate features using a low-discrepancy sequence rather than true randomness

Adaptive Feature Maps

- Everything before this **didn’t take the data into account**

- Adaptive feature maps look at the actual training set and try to minimize the kernel approximation error using the training set as a guide
  - For example: we can do a random feature map, and then **fine-tune the randomness** to minimize the empirical error over the training set
  - Gaining in popularity

- Also, neural networks can be thought of as adaptive feature maps.

Systems Tradeoffs

- Lots of tradeoffs here
- Do we spend more work up-front constructing a more sophisticated approximation, to save work on learning algorithms?
- Would we rather scale with the data, or scale to more complicated problems?
- Another task for **metaparameter optimization**

Questions

- Upcoming things:
  - **Paper 2 review due tonight**
  - Paper 3 in class on Wednesday
  - Start thinking about the class project — it will come faster than you think!