The Kernel Trick, Gram Matrices, and Feature Extraction

CS6787 Lecture 4 — Fall 2019

Basic Linear Models

- For two-class classification using model vector **w**

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

- What is the compute cost of making a prediction in a **d**-dimensional linear model, given an example **x**?

- Answer: **d multiplies and d adds**
  - To do the dot product.

Optimizing Basic Linear Models

- For classification using model vector **w**

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

- Optimization methods for this task vary; here's logistic regression

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

$$(y_i \in \{-1, 1\})$$

SGD on Logistic Regression

- Gradient of a training example is

$$\nabla f_i(w) = \frac{-x_i y_i}{1 + \exp(w^T x_i y_i)}$$

- So SGD update step is

$$w_{t+1} = w_t + \alpha_t \frac{x_i y_i}{1 + \exp(w_t^T x_i y_i)}$$

What is the compute cost of an SGD update?

- For logistic regression on a **d**-dimensional model

$$w_{t+1} = w_t + \alpha_t \frac{x_i y_i}{1 + \exp(w_t^T x_i y_i)}$$

- Answer: **2d multiples and 2d adds + O(1) extra ops**
  - d multiplies and d adds to do the dot product
  - d multiplies and d adds to do the AXPY operation
  - O(1) additional ops for computing the exp, divide, etc.

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**?

Review: 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 $\mathbf{x}$ and $\mathbf{y}$

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:** $K(x,y) = \exp\left(-\beta \|x - y\|_1\right)$
- Hidden layer of a neural network:
  - if layer outputs $\phi(x)$, then kernel is $K(x,y) = \phi(x)^T \phi(y)$

Kernels as a feature mapping

- More generally, any function that can be written in the form

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

(where $\phi : \mathbb{R}^d \to \mathbb{R}^D$ is called a feature map) is a kernel.

- Even works for maps onto infinite dimensional **Hilbert space**
  - And in this case the converse is also true: any kernel has an associated (possibly infinite-dimensional) feature map.

Classifying with Kernels

- Recall the SGD update is

$$w_{t+1} = w_t + \alpha_t \frac{x_i y_i}{1 + \exp(w_t^T x_i y_i)}$$

  - **Resulting weight vectors will always be in the span of the examples.**

- So, our prediction will be:

$$w = \sum_{i=1}^{n} u_i x_i \Rightarrow h_w(x) = \text{sign}\left(w^T x\right) = \text{sign}\left(\sum_{i=1}^{n} u_i x_i^T x\right)$$

Classifying with Kernels

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

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

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

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

Learning with Kernels

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

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

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

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

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 logistic regression learning problem into

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

  - This is much easier than re-computing 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

- As we noted above, this is **true for all kernels**, not just finite ones
  - Just with a possibly infinite-dimensional feature map

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

$$h_u(x) = \operatorname{sign}\left(\sum_{i=1}^{n} u_i K(x_i, x) \right)$$

$$= \operatorname{sign}\left(\sum_{i=1}^{n} u_i \phi(x_i)^T \phi(x) \right) = \operatorname{sign}\left(w^T \phi(x) \right)$$

Learning with feature maps

- Similarly we can simplify our learning objective to

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

- 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

- **Interesting 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 including both features in the map.
  - Strategy: choose the feature map at random, but subject to the constraint that the features be statistically "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 **hyperparameter optimization**

Demo

Dimensionality reduction

Linear models are linear in the dimension

- But what if the dimension **d** is very large?
  - Example: if we have a high-dimensional kernel map

- It can be **difficult to run SGD** when the dimension is very high even if the cost is linear
  - This happens for other learning algorithms too

Idea: reduce the dimension

- If high dimension is the problem, can we just reduce **d**?

- This is the problem of **dimensionality reduction**.

- Dimensionality reduction benefits both statistics and systems
  - Statistical side: can **help with generalization** by identifying important subset of features
  - Systems side: lowers compute cost

Techniques for dimensionality reduction

- **Feature selection by hand**
  - Simple method
  - But costly in terms of human effort

- **Principal component analysis** (PCA)
  - Identify the **directions of highest variance** in the dataset
  - Then project onto those directions
  - Many variants: e.g. kernel PCA

More techniques for dimensionality reduction

- **Locality-sensitive hashing** (LSH)
  - Hash input items into buckets so close-by elements map into the same buckets with high probability
  - Many methods of doing this too
- **Johnson-Lindenstrauss transform** (random projection)
  - General method for reducing dimensionality of any dataset
  - Just **choose a random subspace** and project onto that subspace

Johnson-Lindenstrauss lemma

Given a desired error $\epsilon \in (0,1)$, a set of m points in $\mathbb{R}^d$, and a reduced dimension D that satisfies $D > \frac{8\log(m)}{\epsilon^2}$, there exists a linear map T such that

$$(1-\epsilon)\cdot\|x-y\|^2 \leq \|T(x)-T(y)\|^2 \leq (1+\epsilon)\cdot\|x-y\|^2$$

for all points x and y in the set.

In fact, a randomly chosen linear map T works with high probability!

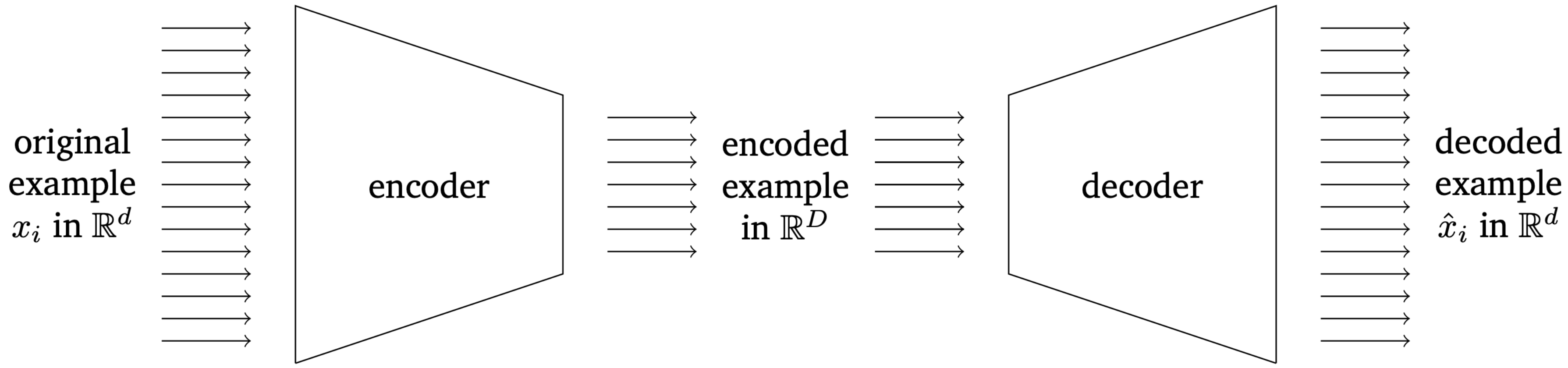
Consequences of J-L transform

- We only need $\mathbf{O(\log(m) / \varepsilon^2)}$ dimensions to map a dataset of size **m** with relative distance accuracy.
  - **No matter what the size of the input dataset was!**
- This is a very useful result for many applications
  - Provides a generic way of reducing the dimension with guarantees
- But there are more **specialized data-dependent ways of doing dimensionality reduction** that can work better.

Autoencoders

- Use **deep learning** to learn two models
  - The **encoder**, which maps an example to a dimension-reduced representation
  - The **decoder**, which maps it back

- Train to minimize the distance between encoded-and-decoded examples and the original example.



Questions

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