Lecture 9. Kernel Trick and Decision Tree

Prof. Alan Yuille

Spring 2014

Outline

1 Kernel Trick

1.1 What is Kernel Trick?

The SVM classifier depends on $\vec{x}$ only by dot product. The prediction is $\hat{y}(\vec{x}) = \text{sign}(\vec{a} \cdot \vec{x} + b) = \text{sign}(\sum_\mu \alpha_\mu y_\mu \vec{x}_\mu \cdot \vec{x} + b)$. This depends on $\vec{x}$ only by: (i) the dot product $\vec{x} \cdot \vec{x}_\mu$, and (ii) the α_μ depend on solving the dual problem (maximizing the dual) which again depends only on the dot product of $\vec{x}_\mu \cdot \vec{x}_\nu$.

This motivates the Kernel Trick. We map the raw data $\vec{x}$ in data space to a feature vector in feature space $\vec{\phi}(\vec{x})$. Then we reformulate the problem in feature space, i.e.. seek a classifier of form:

$$\text{sign}(\vec{c} \cdot \vec{\phi}(\vec{x}) + b)$$

We replace $\vec{x}$ by $\vec{\phi}(\vec{x})$ everywhere in the primal & dual formulation. Then the classifier only depends on the dot product of the $\vec{\phi}(\vec{x})$'s. I.e. on the Kernel $K(\vec{x}, \vec{x}') = \vec{\phi}(\vec{x}) \cdot \vec{\phi}(\vec{x}')$

1.2 Why does kernel help?

First, using features $\phi(.)$ can make it possible to classify data by hyperplanes, which we could not classify in the original data space.

Example:

Logical X-OR, $\vec{x} = (x_1, x_2), x_j \in \{\pm 1\}, y \in \{\pm 1\}$.

The X-OR (exclusive or), see figure 1, requires a decision rule $\alpha(\vec{x})$ s.t.

$$\alpha(1,1) = \alpha(-1,-1) = 1, \quad \alpha(1,-1) = \alpha(-1,1) = -1$$

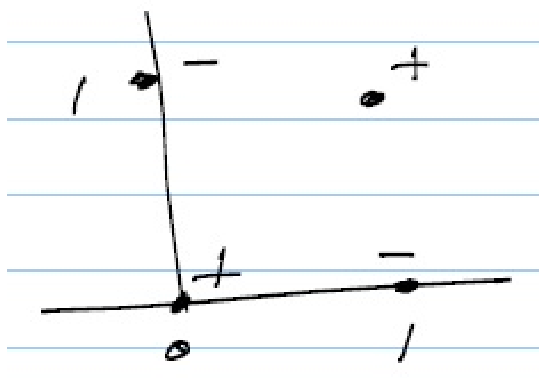

Figure 1: Data for the logical X-or problem. It is impossible to separate the positive and negative examples by a straight line (i.e. to classify them correctly by a linear classifier). But we can find features which will enable us to do this.

It is impossible to find a linear classifier to do this. But define feature $\phi(\vec{x}) = x_1x_2$. Now the classifier $\hat{y}(\vec{x}) = \text{sign}(\phi(\vec{x}))$ can separate the data.

This example is "cheating" because we guess the right feature x_1x_2. But in practice, we don't know beforehand what the right features are. More generally, we could set a higher-dimensional set of features

$$\vec{\phi}(x_1, x_2) = (x_1, x_2, x_1x_2, x_1^2, x_2^2, ...).$$

And we have

$$\vec{\lambda} \cdot \vec{\phi}(x_1, x_2) = \lambda_1 x_1 + \lambda_2 x_2 + \lambda_3 x_1 x_2 + ...$$

Then we let the training algorithm select the right features to separate the positive and negative examples.

The philosophy is that by using a higher-dimensional function $\vec{\phi}(\vec{x})$, it is more likely that we can find a hyperplane to separate the data. But it is also worth noting that we need more examples to train algorithms after using the kernel trick since we essentially increase the dimensionality of the data (most of the cases).

Second, we do not need to specify the features $\vec{\phi}(\vec{x})$ explicitly, because in some cases, feature space can be very high dimensional or even infinite dimensional. Instead, we only need to specify the kernel

$$K(\vec{x}, \vec{x}') = \vec{\phi}(\vec{x}) \cdot \vec{\phi}(\vec{x}').$$

Example: the dual problem of SVM reduces to maximizing

$$L_d(\{\alpha_\mu\}) = \sum_\mu \alpha_\mu - \frac{1}{2}\sum_{\mu,\nu} \alpha_\mu \alpha_\nu y_\mu y_\nu \vec{\phi}(\vec{x}_\mu) \cdot \vec{\phi}(\vec{x}_\nu)$$

$$= \sum_\mu \alpha_\mu - \frac{1}{2}\sum_{\mu,\nu} \alpha_\mu \alpha_\nu y_\mu y_\nu K(\vec{x}_\mu, \vec{x}_\nu)$$

The solution is

$$\hat{\vec{a}} = \sum_\mu \hat{\alpha}_\mu y_\mu \vec{\phi}(\vec{x}_\mu).$$

In prediction, we only need to compute

$$\hat{\vec{a}} \cdot \vec{\phi}(\vec{x}) + b = \sum_\mu \hat{\alpha}_\mu y_\mu \vec{\phi}(\vec{x}) \cdot \vec{\phi}(\vec{x}_\mu) + b = \sum_\mu \hat{\alpha}_\mu y_\mu K(\vec{x}, \vec{x}_\mu) + b.$$

1.3 What kernels to use?

There are many choices of kernels. The difficulty is knowing which one to use. As always, cross-validation is useful for checking whether a kernel can generalize.

There are two important types of kernels, see figure 2.

(1) Polynomial Kernel

$$K(\vec{x}, \vec{x}') = \{1 + \vec{x} \cdot \vec{x}'\}^d, d \geq 1$$

This kernel naturally generalizes the idea of hyperplanes.

(2) Radial Basis Function (RBF) kernel

$$K(\vec{x}, \vec{x}') = e^{-\frac{1}{\sigma^2}|\vec{x} - \vec{x}'|^2}$$

This kernel behaves like nearest neighbor. Only near data points affect classification.

Note that you can also specify a set of kernels, $K_1(\cdot,\cdot), ..., K_m(\cdot,\cdot)$ with parameter $\alpha = (\alpha_1, ..., \alpha_m)$. We consider a linear combination of kernels

$$K(\cdot,\cdot) = \sum_{j=1}^m \alpha_j K_j(\cdot,\cdot).$$

We use cross-validation to select the best parameter α.

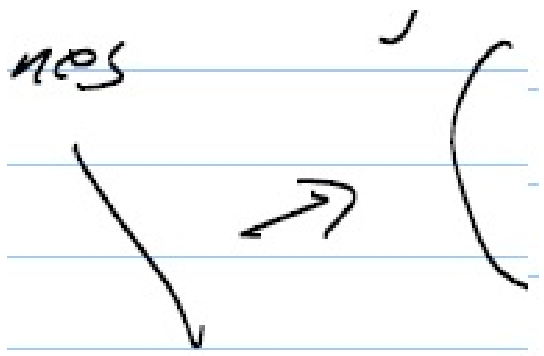

Figure 2: One type of kernel, e.g. $\{1 + \vec{x} \cdot \vec{x}'\}^d$, corresponds to using curved surfaces to separate the data. The other type of kernel, $\exp\{-|\vec{x} - \vec{x}'|^2\}$ is like nearest neighbour.

1.4 Geometric Interpretation

The kernel can be used to specify a distance measure. Recall Euclidean distance

$$|\vec{x}_1 - \vec{x}_2|^2 = \vec{x}_1 \cdot \vec{x}_1 - 2\vec{x}_1 \cdot \vec{x}_2 + \vec{x}_2 \cdot \vec{x}_2$$

Similarly,

$$|\vec{\phi}(\vec{x}_1) - \vec{\phi}(\vec{x}_2)|^2 = \vec{\phi}(\vec{x}_1) \cdot \vec{\phi}(\vec{x}_1) - 2\vec{\phi}(\vec{x}_1) \cdot \vec{\phi}(\vec{x}_2) + \vec{\phi}(\vec{x}_2) \cdot \vec{\phi}(\vec{x}_2) = K(\vec{x}_1, \vec{x}_1) - 2K(\vec{x}_1, \vec{x}_2) + K(\vec{x}_2, \vec{x}_2).$$

This is the Euclidean distance in the feature space.

Think of $\vec{x} \to \vec{\phi}(\vec{x})$ as a transformation on the geofmetry, which its the same as specifying a new distance in the original data space,

$$d(\vec{x}_1, \vec{x}_2) = K(\vec{x}_1, \vec{x}_1) - 2K(\vec{x}_1, \vec{x}_2) + K(\vec{x}_2, \vec{x}_2).$$

Note that the RBF kernel has interesting property

$$K(\vec{x}_1, \vec{x}_1) - 2K(\vec{x}_1, \vec{x}_2) + K(\vec{x}_2, \vec{x}_2) = 2(1 - \exp(-|\vec{x}_1 - \vec{x}_2|^2)).$$

As $|\vec{x}_1 - \vec{x}_2|$ increases, the Euclidean distance between data points in feature space tends to a maximum value of 2.

1.5 When do Kernels Correspond to Features?

Suppose we specify $K(\vec{x}, \vec{x}')$, is it equal to $\vec{\phi}(\vec{x}) \cdot \vec{\phi}(\vec{x}')$ for some features $\vec{\phi}(\vec{x})$?

Theoretical results can be obtained, e.g., Mercer's Theorem

There we do two things:

(1) give some partial understanding, by showing that this relates to the spectral theorem in linear algebra.

Consider $K(\vec{x}, \vec{y})$, where $\vec{x}, \vec{y}$ take values on a finite set $(z_1, ..., z_N)$. Then $K(z_i, z_j) = K_{ij}$. K is symmetric and positive semidefinite. By the spectral theorem, we can express it as

$$K_{ij} = \sum_{\mu=1}^{N} \lambda_\mu e_i^\mu e_j^\mu,$$

where λ_μ and e^μ are the eigenvalue and eigenvector of K, and $\lambda_\mu \geq 0$.

So

$$K_{ij} = \sum_{\mu=1}^{N} (\lambda_\mu^{1/2} e_i^\mu)(\lambda_\mu^{1/2} e_j^\mu) = \vec{\phi}(z_i)\vec{\phi}(z_j),$$

where $\vec{\phi}(z_i) = (\lambda_1^{1/2} e_i^1, ..., \lambda_N^{1/2} e_i^N)$. This answers the question when we restrict $\vec{x}$ and $\vec{y}$ to take values on discrete values.

For continuous values of $\vec{x}$ and $\vec{y}$, this requires functional analysis, finding eigenfunctions and eigenvalues

$$\int K(\vec{x}, \vec{y}) e(\vec{y}) d\vec{y} = \lambda e(\vec{x}).$$

Functional analysis extends linear algebra to functions, which is out of the scope of the class. But Mercer's themrem is like a generalization of the spectral theorem to functional analysis.

(2) explain why this result matters in practice.

Consider the spectral theorem again. Order the eigenvalues so that

$$\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_N.$$

We may get a very good approximation to K by using only a few terms in the spectral theorem, i.e., those with the biggest eigenvalue. We have

$$K_{ij} \approx \sum_{\mu=1}^{n} (\lambda_\mu^{1/2} e_i^\mu)(\lambda_\mu^{1/2} e_j^\mu)$$

where $n \leq N$. This approximation can be very useful for some kernel methods.

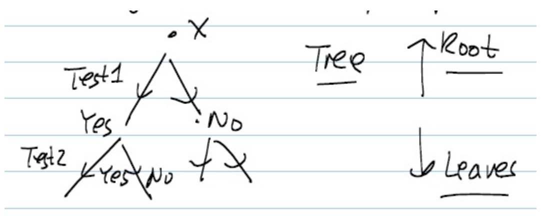

Figure 3:

2 Decision Trees

2.1 Notation

Set of classified Data $< (x_a, w_a) : a \in \Lambda >$
Set of Tests $< T^j : j \in \Phi >$
Each test has response "T" or "F" $T^j(x_a) \in < T, F >$
Tree nodes $< \mu >$
Root node μ_0 at top of tree
Each node either has two child nodes, or is a leaf node.
Each node $< \mu >$ has a test T_μ.
Its child node μ_1 is for data x_a s.t. $T_\mu(x_a) = T$
Its child node μ_2 is for data x_a s.t. $T_\mu(x_a) = F$

Define the data that gets to node μ recursively.
$\Lambda_{\mu_1} = \{x_a \in \Lambda_\mu \text{ s.t. } T_\mu(x_a) = T\}$
$\Lambda_{\mu_2} = \{x_a \in \Lambda_\mu \text{ s.t. } T_\mu(x_a) = F\}$

The root node contains all data $\Lambda_{\mu_0} = \Lambda$
Distribution of data at node μ
$P_\mu(w_j) = \frac{1}{|\Lambda_\mu|} \sum_{a \in \Lambda_\mu} \delta_{w_a, w_j}$
$\sum_{j=1}^{C} P_\mu(w_j) = 1$ $\quad$ C is number of classes.

Define an impurity measure (entropy) for node μ
$I(\mu) = -\sum_j P_\mu(w_j) \log P_\mu(w_j)$

Note: if a node is pure, then all data in it belongs to one class.

Intuition: Design a tree so that the leaf nodes are pure -yield good classification

2.2 Iterative Design

Initialize tree with the root node only (so it is a leaf node).

For all leaf nodes, calculate the maximal decrease in impurity by searching over all the tests.

Expand the leaf node with maximal decrease and add its child nodes to the tree.

Decrease in impurity at node μ due to test T^j

$\Lambda^j_{\mu,1} = \{x \in \Lambda_\mu : s.t.\ T^j_\mu(x) = T\}$

$\Lambda^j_{\mu,2} = \{x \in \Lambda_\mu : s.t.\ T^j_\mu(x) = F\}$

Decrease in entropy $\Delta^j I(\mu) = I(\mu) - I(\mu^j_1, \mu^j_2)$,

where $I(\mu^j_1, \mu^j_2) = \frac{|\Lambda^j_{\mu,1}|}{\Lambda_\mu} I(\mu^j_1) + \frac{|\Lambda^j_{\mu,2}|}{\Lambda_\mu} I(\mu^j_2)$

Hence, for all leaf node μ, calculate the best test $T^*_\mu = \max_j I(\mu^j_1, \mu^j_2)$.

2.3 Greedy Strategy

Start at root node μ_0

Expand root node with test that maximize the decrease in impurity - or maximize the gain in purity.

Repeat with leaf nodes until each node is pure.

Time Complexity: Learning algorithm is $O(|\Phi||\Lambda|\{\log|\Lambda|\}^2)$

$|\Phi|$ = no. of tests. Run time $O(\log|\Lambda|)$.
Notes: the design strategy is very greedy. There may be a shorter tree if you learn the tree by searching over a sequence of tests.

The number of children is arbitrary. You can extend the approach to having three, or more, children.

2.4 Others

There are alternative impurity measures.

The Gini index:
$I(\mu) = \sum_{i,j\{i \neq j\}} P_\mu(\omega_i) P_\mu(\omega_j) = 1 - \sum_j P_\mu^2(\omega_j)$

Note that expanding the tree until all nodes are pure risks overgeneralizing. It will give perfect performance on the training dataset, but will usually cause error on the test dataset. Better to stop splitting the data when the impurity reaches a positive threshold. Set a node to be a leaf if $J(\mu) \leq \beta$. Then at each leaf, classify data by majoring vote.

Cross Validation : learn the decision tree with different impurity thresholds β.