Introduction to Kernel Methods

Ron Parr
CPS 271

Outline

• Motivation for kernel methods
• The dual view
• What makes a kernel?
• The Gaussian Process view
• Review of why we care
• Classification
Parametric Methods

- Most methods discussed so far have been *parametric*
- Learning mechanism/representation is characterized by a fixed set of parameters
  - Regression coefficients
  - Neural network weights
  - Linear discriminant means and covariances
- Connection between data and output is complicated and circuitous:
  - First we pick an algorithm, then our features,
  - Algorithm tunes parameters to data
  - Data are discarded
- Final result is a function from data to predictions
- Is there a more direct route?

Non-Parametric Methods

- Traditionally the simplest methods
- Not discussed heavily in the book
- K-Nearest neighbor
  - Classify according to nearest (in feature space) neighbor
  - For \( k>1 \), vote
- Locally weighted regression
  - Use query dependent regression weights (rerun regression for every query)
  - Weights decay with distance from query point
- Issues
  - Defining closeness (both scale and dimensions)
  - Efficiency
  - Tendency to overfit w/o careful tweaking
But how different are these, really?

• Both are functions from data to predictions
• Parametric methods
  – Seem more constrained (less expressive)
  – Store less information

• Is there a unifying view?

---

Dual View of Regression

• Recall (regularized) regression minimizes:

\[ J(w) = 0.5 \lambda w^T w + 0.5 \sum_{i=1}^{N} (w^T \phi(x^{(i)}) - t^{(i)})^2 \]

• Setting \( \delta J/\delta w = 0 \):

\[ w = \frac{-1}{\lambda} \sum_{i=1}^{N} (w^T \phi(x^{(i)}) - t^{(i)}) \phi(x^{(i)}) \]
Dual View Continued

\[ w = \frac{-1}{\lambda} \sum_{i=1}^{N} (w^T \phi(x^{(i)}) - t^{(i)}) \phi(x^{(i)}) \]

- Some notation:

\[
\begin{bmatrix}
\phi(x^{(1)}) \\
\phi(x^{(2)}) \\
\vdots \\
\phi(x^{(N)})
\end{bmatrix}
\quad a =
\begin{bmatrix}
\frac{-1}{\lambda}(w^T \phi(x^{(1)}) - t^{(1)}) \\
\frac{-1}{\lambda}(w^T \phi(x^{(2)}) - t^{(2)}) \\
\vdots \\
\frac{-1}{\lambda}(w^T \phi(x^{(N)}) - t^{(N)})
\end{bmatrix}
\]

- Simplifying: \( w = \Phi^T a \)

Substituting

\[
a = \begin{bmatrix}
\frac{-1}{\lambda}(w^T \phi(x^{(1)}) - t^{(1)}) \\
\frac{-1}{\lambda}(w^T \phi(x^{(2)}) - t^{(2)}) \\
\vdots \\
\frac{-1}{\lambda}(w^T \phi(x^{(N)}) - t^{(N)})
\end{bmatrix}
\]

\[
\Phi a + \frac{1}{\lambda} t = (\Phi^T + \lambda I)^{-1} t
\]

\[
\begin{bmatrix}
\frac{-1}{\lambda}(a^T \Phi \phi(x^{(1)}) - t^{(1)}) \\
\frac{-1}{\lambda}(a^T \Phi \phi(x^{(2)}) - t^{(2)}) \\
\vdots \\
\frac{-1}{\lambda}(a^T \Phi \phi(x^{(N)}) - t^{(N)})
\end{bmatrix}
\]

\[
\begin{bmatrix}
\frac{-1}{\lambda}(a^T \Phi \phi(x^{(1)}) - t^{(1)}) \\
\frac{-1}{\lambda}(a^T \Phi \phi(x^{(2)}) - t^{(2)}) \\
\vdots \\
\frac{-1}{\lambda}(a^T \Phi \phi(x^{(N)}) - t^{(N)})
\end{bmatrix}
\]
The Gram Matrix

- Define $K = \Phi \Phi^T$

$$a = (K + \lambda I)^{-1} t$$

- We call $K$ a kernel

Prediction

- Recall that to predict a new value we use:

$$y(x) = w^T \phi(x)$$

- Substituting:

$$w = \Phi^T a \quad a = (K + \lambda I)^{-1} t$$

$$y(x) = w^T \phi(x) = a^T \Phi \phi(x) = k(x)^T (K + \lambda I)^{-1} t$$

Extension to the Gram matrix
Why this matters?

- We have expressed regression entirely in terms of the gram matrix, $K$, which we call the kernel
- But $K$ is defined entirely of dot products between vectors in our training set
- Note that $K$ is $(nxn)$ not $(kxk)$!

- Let’s think about what $K$ really means...

What does dot product mean?

- Elements of $K$ are dot products between vectors, $x^Ty$
- If $x=y$, $x^Ty$ is squared magnitude
- If $x$ has norm 1, then $x^Ty$ is projection of $y$ onto $x$
- In general $x^Ty/(||x|| ||y||)$ is the cosine between $x$ and $y$
- If $x$ and $y$ have the same magnitude, then $x^Ty$ is maximized when $x=y$
What happens if we redefine dot product?

- If we change the meaning of \( x^T y \), then we change what it means for two vectors to be similar
- The big idea: Turn feature concept on its head
- Instead of investing effort in finding features, invest effort in finding kernels

- Algorithmically equivalent to redefining the K matrix

Kernel example

- The default (linear) kernel says \( k(x,y) = x^T y \)
- What if \( k(x,y) = (x^T y)^2 \) instead?
- Suppose \( x=(x_1, x_2) \), \( y=(y_1, y_2) \)

\[
k(x, y) = (x^T y)^2 = (x_1 y_1 + x_2 y_2)^2
= (x_1^2 y_1^2 + 2x_1 y_1 x_2 y_2 + x_2^2 y_2^2)
= \begin{pmatrix} x_1^2 & \sqrt{2} x_1 x_2 \\ \sqrt{2} x_1 x_2 & x_2^2 \end{pmatrix}
\begin{pmatrix} y_1^2 & \sqrt{2} y_1 y_2 \\ \sqrt{2} y_1 y_2 & y_2^2 \end{pmatrix}
= \phi(x)^T \phi(y)
\]

where,

\[
\phi(z) = \begin{pmatrix} z_1^2 \\ \sqrt{2} z_1 z_2 \\ z_2^2 \end{pmatrix}^T
\]
What just happened?

- We just changed the feature space from linear in $x$, to quadratic in $x$
- But we don’t actually need to construct these features; we just need to redefine $K$

- Question: Does every choice of $K$ make sense? (Does every $K$ correspond to sensible basis?)

What makes a valid kernel?

- In general, we want our $K$ matrix to be symmetric, positive semidefinite (not proved here)
- A sufficient (but not necessary) condition is for $K$ to behave like a distance metric
  - Nonnegative
  - $K(x,x)=0$
  - Symmetric
  - Obeys triangle inequality
- Fancy kernels can be constructed by combining simple ones
Valid Kernel Combinations

\[ k(x, x') = ck_1(x, x') \]
\[ k(x, x') = f(x)k_1(x, x')f(x') \]
\[ k(x, x') = q(k_2(x, x')) \]
\[ k(x, x') = e^{k_i(x, x')} \]
\[ k(x, x') = k_1(x, x') + k_2(x, x') \]
\[ k(x, x') = k_1(f(x), f(x')) \]
\[ k(x, x') = x^T A x' \]
\[ k(x, x') = k_a(x_a, x_a') + k_b(x_b, x_b') \]
\[ k(x, x') = k_a(x_a, x_a')k_b(x_b, x_b') \]

Assumptions:
- \( c > 0 \)
- \( f(.) \) is any function
- \( q(.) \) is polynomial w/\( \geq 0 \) coefficients
- \( k_i \) is any valid kernel function
- \( x = (x_a, x_b) \)
- \( A \) is symmetric and PSD

Other Interesting Kernels

- Polynomial kernel:
  \[ k(x, x') = (x^T x' + c)^M \]

- Gaussian Kernel:
  \[ k(x, x') = e^{-\frac{||x-x'||^2}{2\sigma^2}} \]
More Kernels

• Kernels over sets:

\[ k(A, A') = 2^{|A \cap A'|} \]

• Kernels from probabilities

\[ k(x, x') = p(x)p(x') \]

Gaussian Process View

• Recall that regularized regression is equivalent to the ML solution to Bayesian regression with a mean 0 Gaussian prior on the weights

• In the dual view, we interpret our training data as inducing a joint distribution over the y targets
### Gaussian Process Formulation

\[ y = \Phi w \]

- \( w \) is a vector of Gaussian RVs, so \( y \) is a vector of Gaussian RVs
  
  \[
  E(y) = \Phi E(w) = 0
  \]

  \[
  \text{cov}(y) = E(yy^T) = \Phi E(ww^T)\Phi^T = \frac{1}{\alpha} \Phi \Phi^T = K
  \]

- Notes:
  - This assumes prior variance on \( w = 1/\alpha \)
  - We can hide this constant in \( K \)

### Noise model for targets

- Recall that we assume our targets values \( t \), are the result of an underlying \( y \), corrupted with Gaussian noise

\[
\begin{align*}
p(y) &= N(y \mid 0, K) \\
p(t \mid y) &= N(t \mid y, \beta^{-1}) \\
p(t \mid y) &= N(t \mid y, \beta^{-1}I) \\
p(t) &= \int p(t \mid y) \rho(y) dy = N(t \mid 0, C) \\
C &= K + \beta^{-1}I
\end{align*}
\]
Extending to Prediction

• For prediction, we want

\[ p(t_{N+1}) = N(t_{N+1} \mid 0, C_{N+1}) \]

• We don’t need \( y \) to define \( C_{n+1} \)

\[ C_{N+1} = \begin{pmatrix} C_N & k \\ k^T & c \end{pmatrix} \]

• Where \( C_N \) is our previous \( C \), \( k \) is the kernel between old (training) and new \( (x_{N+1}) \) points, and \( c = k(x_{N+1}, x_{N+1}) + \beta^{-1} \)

Now we turn the magic crank

• All we need to do now is to condition on previously observed target values to get

\[ P(t_{N+1} \mid t) \]

• Using the conditioning results from chapter 2:

\[ \mu(x_{N+1}) = k^T C_N^{-1} t \]

\[ \sigma^2(x_{N+1}) = c - k^T C_N^{-1} k \]
Compare with Regularized Regression

- Regularized regression with kernels:
  \[ y(x) = w^T \phi(x) = a^T \Phi \phi(x) = k(x)^T (K + \lambda I)^{-1} t \]

- Gaussian process regression:
  \[
  \mu(x_{N+1}) = k^T C^{-1} t \\
  \sigma^2(x_{N+1}) = c - k^T C^{-1} k
  \]

- Recall:
  \[ C = K + \beta^{-1} I \]

Advantages of the GP View

- Why bother with GP if regularized regression does the same thing?

- GP also gives us a variance in prediction
- GP gives us a distribution over targets
- GP is more general, can incorporate other types of priors, including priors over priors
GP Regression Example

Variance increases with distance from training points (good) and is a function of $\beta$, but how do you pick $\beta$?

Reality Check

- Cost of ordinary regression:
  - Feature space of dimension $k$, $N$ training points
  - Storage of solution $O(k)$
  - Computation:
    - Cubic in $k$
    - linear in $N$

- Cost of kernel version:
  - Feature space of dimension $k$, $N$ training points
  - Storage of solution $O(N)$
  - Computation
    - Cubic in $N$,
    - No explicit dependence on $k$
Why do we like Kernels?

• Let us experiment with feature spaces without paying the cost of constructing the features

• But what about overfitting? (Isn’t k>N dangerous?)

• Yes! This is why we need regularization!
• This issue becomes particularly interesting in the context of support vector machines

Kernels for Classification

• Idea 1:
  – Use logistic regression, replacing $x^Tw$ with some kind of kernel regression
  – Problem:
    • No clean training algorithm
    • Must use approximations

• Idea 2: Support Vector Machines
  (next chapter/lecture)
Some Concluding Thoughts

• Kernels and Gaussian processes have become very popular in recent years
• Why?
  – Coolness factor
  – Ability to work in weird spaces implicitly
• Is this a good thing?
  – Not a substitute for standard linear + good features
  – Some advantages over other non-parametric methods
    • Potential for elegant treatment of regularization
    • GP provides probability distribution
  – Kernels useful for classification and other techniques beyond regression