Regularized Least Squares

Charlie Frogner

MIT

2011

1 Slides mostly stolen from Ryan Rifkin (Google).
In RLS, the Tikhonov minimization problem boils down to solving a linear system (and this is good).

We can compute the solution for each of a bunch of \( \lambda \)'s, by using the eigendecomposition of the kernel matrix.

We can compute the leave-one-out error over the whole training set about as cheaply as solving the minimization problem once.

The linear kernel allows us to do all of this when \( n \gg d \).
Training set: $S = \{(x_1, y_1), \ldots, (x_n, y_n)\}$.

Inputs: $X = \{x_1, \ldots, x_n\}$.

Labels: $Y = \{y_1, \ldots, y_n\}$. 

C. Frogner
Regularized Least Squares
RKHS $\mathcal{H}$ with a positive semidefinite kernel function $K$:

- linear: $K(x_i, x_j) = x_i^T x_j$
- polynomial: $K(x_i, x_j) = (x_i^T x_j + 1)^d$
- gaussian: $K(x_i, x_j) = \exp\left(-\frac{||x_i - x_j||^2}{\sigma^2}\right)$

Define the kernel matrix $K$ to satisfy $K_{ij} = K(x_i, x_j)$.

The kernel function with one argument fixed is $K_x = K(x, \cdot)$.

Given an arbitrary input $x_*$, $K_{x_*}$ is a vector whose $i$th entry is $K(x_i, x_*)$. (So the training set $X$ is assumed.)
Goal: Find the function \( f \in \mathcal{H} \) that minimizes the weighted sum of the square loss and the RKHS norm

\[
\arg\min_{f \in \mathcal{H}} \frac{1}{2} \sum_{i=1}^{n} (f(x_i) - y_i)^2 + \frac{\lambda}{2} \| f \|_{\mathcal{H}}^2.
\] (1)

This loss function makes sense for regression. We can also use it for binary classification, where it is less immediately intuitive but works great.

Also called “ridge regression.”
Applying the Representer

**Claim:** We can rewrite (1) as

$$\arg\min_{c \in \mathbb{R}^n} \frac{1}{2} \| Y - Kc \|_2^2 + \frac{\lambda}{2} || f ||_{\mathcal{H}}^2.$$  

**Proof:** The representer theorem guarantees that the solution to (1) can be written as

$$f(\cdot) = \sum_{j=1}^{n} c_j K_{x_j}(\cdot)$$

for some $c \in \mathbb{R}^n$.

So $Kc$ gives a vector whose $i$th element is $f(x_i)$:

$$f(x_i) = \sum_{j=1}^{n} c_j K_{x_j}(x_j) = \sum_{j=1}^{n} c_j K_{ij} = (K_{i,\cdot})c$$
Claim:

\[ \|f\|_{\mathcal{H}}^2 = c^T K c. \]

Proof:

\[ f(\cdot) = \sum_{j=1}^{n} c_j K x_j(\cdot), \]

so

\[ \|f\|_{\mathcal{H}}^2 = \langle f, f \rangle_{\mathcal{H}} \]

\[ = \left\langle \sum_{i=1}^{n} c_i K x_i, \sum_{j=1}^{n} c_j K x_j \right\rangle_{\mathcal{H}} \]

\[ = \sum_{i=1}^{n} \sum_{j=1}^{n} c_i c_j \langle K x_i, K x_j \rangle_{\mathcal{H}} \]

\[ = \sum_{i=1}^{n} \sum_{j=1}^{n} c_i c_j K(x_i, x_j) = c^T K c \]
Putting it all together, the RLS problem is:

$$\arg\min_{f \in \mathcal{H}} \frac{1}{2} \| Y - Kc \|_2^2 + \frac{\lambda}{2} c^T Kc$$

This is convex in $c$ (why?), so we can find its minimum by setting the gradient w.r.t $c$ to 0:

$$-K(Y - Kc) + \lambda Kc = 0$$

$$(K + \lambda I)c = Y$$

$$c = (K + \lambda I)^{-1} Y$$

We find $c$ by solving a system of linear equations.
The RLS Solution, Comments

- The solution exists and is unique (for $\lambda > 0$).
- Define $G(\lambda) = K + \lambda I$. (Often $\lambda$ is clear from context and we write $G$.)
- The prediction at a new test input $x_*$ is:

$$f(x_*) = \sum_{j=1}^{n} c_j K_{x_j}(x_*)$$

$$= K_{x_*} c$$

$$= K_{x_*} G^{-1} Y$$

- The use of $G^{-1}$ (or other inverses) is formal only. We do not recommend taking matrix inverses.
Situation: All hyperparameters fixed
We just need to solve a single linear system

$$(K + \lambda I)c = Y.$$ 

The matrix $K + \lambda I$ is symmetric positive definite, so the appropriate algorithm is Cholesky factorization.

In Matlab, the “slash” operator seems to be using Cholesky, so you can just write $c = (K+\lambda \cdot I) \backslash Y$, but to be safe, (or in octave), I suggest $R = \text{chol}(K+\lambda \cdot I); \ c = (R \backslash (R' \backslash Y ));$. 
Situation: We don’t know what $\lambda$ to use, all other hyperparameters fixed.

Is there a more efficient method than solving $c(\lambda) = (K + \lambda I)^{-1}Y$ anew for each $\lambda$?

Form the eigendecomposition $K = QQ^T$, where $\Lambda$ is diagonal with $\Lambda_{ii} \geq 0$ and $QQ^T = I$.

Then

$$G = K + \lambda I$$

$$= QQ^T + \lambda I$$

$$= Q(\Lambda + \lambda I)Q^T,$$

which implies that $G^{-1} = Q(\Lambda + \lambda I)^{-1}Q^T$. 

C. Frogner

Regularized Least Squares
Situation: We don’t know what \( \lambda \) to use, all other hyperparameters fixed.

Is there a more efficient method than solving \( c(\lambda) = (K + \lambda I)^{-1}Y \) anew for each \( \lambda \)?

Form the eigendecomposition \( K = Q \Lambda Q^T \), where \( \Lambda \) is diagonal with \( \Lambda_{ii} \geq 0 \) and \( QQ^T = I \).

Then

\[
G = K + \lambda I \\
= Q \Lambda Q^T + \lambda I \\
= Q(\Lambda + \lambda I)Q^T,
\]

which implies that \( G^{-1} = Q(\Lambda + \lambda I)^{-1}Q^T \).
$O(n^3)$ time to solve one (dense) linear system, or to compute the eigendecomposition (constant is maybe 4x worse). Given $Q$ and $\Lambda$, we can find $c(\lambda)$ in $O(n^2)$ time:

$$c(\lambda) = Q(\Lambda + \lambda I)^{-1}Q^T Y,$$

noting that $(\Lambda + \lambda I)$ is diagonal.

Finding $c(\lambda)$ for many $\lambda$’s is (essentially) free!
We showed how to find $c(\lambda)$ quickly as we vary $\lambda$.

But how do we decide if a given $\lambda$ is “good”?  

Simplest idea: Use the training set error.

Problem: This invariably overfits.  *Don’t do this!*  

Other methods are possible, but today we consider *validation*.  

Validation means checking our function’s behavior on points other than the training set.
We showed how to find $c(\lambda)$ quickly as we vary $\lambda$.

But how do we decide if a given $\lambda$ is “good”?

Simplest idea: Use the training set error.

Problem: This invariably overfits. **Don’t do this!**

Other methods are possible, but today we consider *validation*.

Validation means checking our function’s behavior on points other than the training set.
If we have a huge amount of data, we could hold back some percentage of our data (30% is typical), and use this development set to choose hyperparameters.

More common is \textit{k-fold cross-validation}, which means a couple of different things:

- Divide your data into \( k \) equal sets \( S_1, \ldots, S_k \). For each \( i \), train on the other \( k - 1 \) sets and test on the \( i \)th set.
- A total of \( k \) times, randomly split your data into a training and test set.

The limit of (the first kind of) k-fold validation is \textit{leave-one-out cross-validation}.
For each data point $x_i$, build a classifier using the remaining $n - 1$ data points, and measure the error at $x_i$.

Empirically, this seems to be the method of choice when $n$ is small.

Problem: We have to build $n$ different predictors, on data sets of size $n - 1$.

We will now proceed to show that for RLS, obtaining the LOO error is (essentially) free!
Define $S^i$ to be the data set with the $i$th point removed:

$$S^i = \{(x_1, y_1), \ldots, (x_{i-1}, y_{i-1}), \text{*poof*}, (x_{i+1}, y_{i+1}), \ldots, (x_n, y_n)\}$$

The $i$th leave-one-out value is $f_{S^i}(x_i)$.

The $i$th leave-one-out error is $y_i - f_{S^i}(x_i)$.

Define $L_V$ and $L_E$ to be the vectors of leave-one-out values and errors over the training set.

$\|L_E\|^2$ is considered a good empirical proxy for the error on future points, and we often want to choose parameters by minimizing this quantity.
Imagine that we already know $f_{Si}(x_i)$.

Define the vector $Y^i$ via

$$y_j^i = \begin{cases} 
    y_j & j \neq i \\
    f_{Si}(x_i) & j = i 
\end{cases}$$
Claim: Solving RLS using $Y^i$ gives us $f_{S_i}$, i.e.

$$f_{S_i} = \arg\min_{f \in \mathcal{H}} \frac{1}{2} \sum_{j=1}^{n} (y_j^i - f(x_j))^2 + \frac{\lambda}{2} \|f\|^2_{\mathcal{H}} = (*) .$$

Proof:

$$(1) = (y_i^i - f(x_i))^2 \geq 0 \ \forall f$$

and

$$(y_i^i - f_{S_i}(x_i))^2 = (f_{S_i}(x_i) - f_{S_i}(x_i))^2 = 0$$

$\Rightarrow$

$f_{S_i}$ minimizes (1)

$f_{S_i}$ also minimizes $\sum_{j \neq i} (y_j^i - f(x_j))^2 + \frac{\lambda}{2} \|f\|^2_{\mathcal{H}} = (2)$

$\Rightarrow$

$f_{S_i}$ minimizes (*) = (1) + (2)
Therefore,

\[ c^i = G^{-1}Y^i \]
\[ f_{Si}(x_i) = (KG^{-1}Y^i)_i \]

This is circular reasoning so far, because we need to know \( f_{Si}(x_i) \) to form \( Y^i \) in the first place.

However, assuming we have already solved RLS for the whole training set, and we have computed \( f_S(X) = KG^{-1}Y \), we can do something nice . . .
\begin{align*}
\mathcal{f}_{Si}(x_i) - \mathcal{f}_S(x_i) &= \sum_j (\mathbf{K}\mathbf{G}^{-1})_{ij}(y_j^i - y_j) \\
&= (\mathbf{K}\mathbf{G}^{-1})_{ii}(\mathcal{f}_{Si}(x_i) - y_i) \\
\mathcal{f}_{Si}(x_i) &= \frac{\mathcal{f}_S(x_i) - (\mathbf{K}\mathbf{G}^{-1})_{ii}y_i}{1 - (\mathbf{K}\mathbf{G}^{-1})_{ii}} \\
&= \frac{(\mathbf{K}\mathbf{G}^{-1}\mathbf{Y})_i - (\mathbf{K}\mathbf{G}^{-1})_{ii}y_i}{1 - (\mathbf{K}\mathbf{G}^{-1})_{ii}}.
\end{align*}
\( L_v = \frac{KG^{-1}Y - \text{diag}_m(KG^{-1})Y}{\text{diag}_v(I - KG^{-1})} \),

\( L_E = Y - L_v \)

\[
\begin{align*}
L_E &= Y + \frac{\text{diag}_m(KG^{-1})Y - KG^{-1}Y}{\text{diag}_v(I - KG^{-1})} \\
&= \frac{\text{diag}_m(I - KG^{-1})Y}{\text{diag}_v(I - KG^{-1})} + \frac{\text{diag}_m(KG^{-1})Y - KG^{-1}Y}{\text{diag}_v(I - KG^{-1})} \\
&= \frac{Y - KG^{-1}Y}{\text{diag}_v(I - KG^{-1})}.
\end{align*}
\]
We can simplify our expressions in a way that leads to better computational and numerical properties by noting

\[
KG^{-1} = Q\Lambda Q^T Q(\Lambda + \lambda I)^{-1} Q^T \\
= Q\Lambda(\Lambda + \lambda I)^{-1} Q^T \\
= Q(\Lambda + \lambda I - \lambda I)(\Lambda + \lambda I)^{-1} Q^T \\
= I - \lambda G^{-1}.
\]
Substituting into our expression for $L_E$ yields

$$L_E = \frac{Y - KG^{-1}Y}{\text{diag}_v(I - KG^{-1})}$$

$$= \frac{Y - (I - \lambda G^{-1})Y}{\text{diag}_v(I - (I - \lambda G^{-1}))}$$

$$= \frac{\lambda G^{-1}Y}{\text{diag}_v(\lambda G^{-1})}$$

$$= \frac{G^{-1}Y}{\text{diag}_v(G^{-1})}$$

$$= \frac{c}{\text{diag}_v(G^{-1})}.$$
The cost of computing $L_E$

- For RLS, we compute $L_E$ via

$$L_E = \frac{c}{\text{diag}_v(G^{-1})}.$$ 

- We already showed how to compute $c(\lambda)$ in $O(n^2)$ time (given $K = Q\Lambda Q^T$).

- We can also compute a single entry of $G(\lambda)^{-1}$ in $O(n)$ time:

$$G^{-1}_{ij} = (Q(\Lambda + \lambda I)^{-1}Q^T)_{ij}$$

$$= \sum_{k=1}^{n} \frac{Q_{ik}Q_{jk}}{\Lambda_{kk} + \lambda},$$

and therefore we can compute $\text{diag}(G^{-1})$, and compute $L_E$, in $O(n^2)$ time.
If we can (directly) solve one RLS problem on our data, we can find a good value of $\lambda$ using LOO optimization at essentially the same cost.

When can we solve one RLS problem? (I.e. what are the bottlenecks?)

We need to form $K$, which takes $O(n^2d)$ time and $O(n^2)$ memory. We need to perform a solve or an eigendecomposition of $K$, which takes $O(n^3)$ time.

Usually, we run out of memory before we run out of time.

The practical limit on today’s workstations is (more-or-less) 10,000 points (using Matlab).

How can we do more?
If we can (directly) solve one RLS problem on our data, we can find a good value of \( \lambda \) using LOO optimization at essentially the same cost.

When can we solve one RLS problem? (I.e. what are the bottlenecks?)

We need to form \( K \), which takes \( O(n^2d) \) time and \( O(n^2) \) memory. We need to perform a solve or an eigendecomposition of \( K \), which takes \( O(n^3) \) time.

Usually, we run out of memory before we run out of time.

The practical limit on today’s workstations is (more-or-less) 10,000 points (using Matlab).

How can we do more?
The linear kernel is $K(x_i, x_j) = x_i^T x_j$.

The linear kernel offers many advantages for computation.

Key idea: we get a decomposition of the kernel matrix for free: $K = XX^T$.

In the linear case, we will see that we have two different computation options.
With a linear kernel, the function we are learning is linear as well:

\[
\begin{align*}
f(x_*) &= K_{x_*,c} \\
&= x_\text{T}X^\text{T}c \\
&= x_\text{T}w,
\end{align*}
\]

where we define the hyperplane \(w\) to be \(X^\text{T}c\). We can classify new points in \(O(d)\) time, using \(w\), rather than having to compute a weighted sum of \(n\) kernel products (which will usually cost \(O(nd)\) time).
Assume $n$, the number of points, is bigger than $d$, the number of dimensions. (If not, the best bet is to ignore the special properties of the linear kernel.)

The economy-size SVD of $X$ can be written as $X = USV^T$, with $U \in \mathbb{R}^{n \times d}$, $S \in \mathbb{R}^{d \times d}$, $V \in \mathbb{R}^{d \times d}$, $U^T U = V^T V =VV^T = I_d$, and $S$ diagonal and positive semidefinite. (Note that $UU^T \neq I_n$).

We will express the LOO formula directly in terms of the SVD, rather than $K$. 

\[
\begin{align*}
K &= \mathbf{XX}^T = (USV^T)(VSU^T) = US^2U^T \\
K + \lambda I &= US^2U^T + \lambda I_n \\
&= \begin{bmatrix} U & U_\perp \end{bmatrix} \begin{bmatrix} S^2 + \lambda I_d & \lambda I_{n-d} \\ \lambda I_{n-d} & \lambda I_n \end{bmatrix} \begin{bmatrix} U^T \\ U_\perp^T \end{bmatrix} \\
&= U(S^2 + \lambda I_d)U^T + \lambda U_\perp U_\perp^T \\
&= U(S^2 + \lambda I_d)U^T + \lambda(I_n -UU^T) \\
&= US^2U^T + \lambda I_n
\end{align*}
\]
\[
(K + \lambda I)^{-1} = (US^2U^T + \lambda I_n)^{-1} \\
= \left( \begin{bmatrix} U & U_\perp \end{bmatrix} \begin{bmatrix} S^2 + \lambda I_d & \lambda I_{n-d} \\ \lambda I_{n-d} & \lambda I_{n-d} \end{bmatrix} \begin{bmatrix} U^T \\ U_\perp^T \end{bmatrix} \right)^{-1} \\
= \begin{bmatrix} U & U_\perp \end{bmatrix} \begin{bmatrix} S^2 + \lambda I_d \\ \lambda I_{n-d} \end{bmatrix}^{-1} \begin{bmatrix} U^T \\ U_\perp^T \end{bmatrix} \\
= U(S^2 + \lambda I)^{-1} U^T + \lambda^{-1} U_\perp U_\perp^T \\
= U(S^2 + \lambda I)^{-1} U^T + \lambda^{-1}(I - UU^T) \\
= U \left( (S^2 + \lambda I)^{-1} - \lambda^{-1} I \right) U^T + \lambda^{-1} I
\]
\[c = (K + \lambda I)^{-1}Y\]
\[= U \left[ (S^2 + \lambda I)^{-1} - \lambda^{-1} I \right] U^T Y + \lambda^{-1} Y\]

\[G^{-1}_{ij} = \sum_{k=1}^{d} U_{ik} U_{jk} \left[ (S_{kk} + \lambda)^{-1} - \lambda^{-1} \right] + [i = j] \lambda^{-1}\]

\[G^{-1}_{ii} = \sum_{k=1}^{d} U_{ik}^2 \left[ (S_{kk} + \lambda)^{-1} - \lambda^{-1} \right] + \lambda^{-1}\]

\[L_E = \frac{c}{\text{diag}_v(G^{-1})}\]
\[= \frac{U \left[ (S^2 + \lambda I)^{-1} - \lambda^{-1} I \right] U^T Y + \lambda^{-1} Y}{\text{diag}_v(U \left[ (S^2 + \lambda I)^{-1} - \lambda^{-1} I \right] U^T + \lambda^{-1} I)}\]
We need $O(nd)$ memory to store the data in the first place. The (economy-sized) SVD also requires $O(nd)$ memory, and $O(nd^2)$ time.

Once we have the SVD, we can compute the LOO error (for a given $\lambda$) in $O(nd)$ time.

Compared to the nonlinear case, we have replaced an $O(n)$ with an $O(d)$, in both time and memory. If $n \gg d$, this can represent a huge savings.
For the linear kernel,

\[
L = \arg\min_{c \in \mathbb{R}^n} \frac{1}{2} \|Y - Kc\|_2^2 + \frac{\lambda}{2} c^T K c
\]

\[
= \arg\min_{c \in \mathbb{R}^n} \frac{1}{2} \|Y - XX^T c\|_2^2 + \frac{\lambda}{2} c^T XX^T c
\]

\[
= \arg\min_{w \in \mathbb{R}^d} \frac{1}{2} \|Y - Xw\|_2^2 + \frac{\lambda}{2} \|w\|_2^2.
\]

Taking the derivative with respect to \(w\),

\[
\frac{\partial L}{\partial w} = X^T Xw - X^T Y + \lambda w,
\]

and setting to zero implies

\[
w = (X^T X + \lambda I)^{-1} X^T Y.
\]
If we are willing to give up LOO validation, we can skip the computation of $c$ and just get $w$ directly.

We can work with the *Gram matrix* $X^T X \in \mathbb{R}^{d \times d}$.

The algorithm is identical to solving a general RLS problem with kernel matrix $X^T X$ and labels $X^T y$.

Form the eigendecomposition of $X^T X$, in $O(d^3)$ time, form $w(\lambda)$ in $O(d^2)$ time.

Why would we give up LOO validation? Maybe $n$ is very large, so using a development set is good enough.
In RLS, the Tikhonov minimization problem boils down to solving a linear system:

$$\arg\min_{f \in \mathcal{H}} \frac{1}{2} \sum_{i=1}^{n} (y_i - f(x_i))^2 + \frac{\lambda}{2} \|f\|_{\mathcal{H}}^2 = K(\cdot)c$$

where \((K + \lambda I)c = Y\).

We can (more) cheaply compute \(c(\lambda)\) for a bunch of \(\lambda\)'s, by using the eigendecomposition of the kernel matrix: \(K = QQ^T\).

We can compute the leave-one-out error over the whole training set about as cheaply as solving for \(c\) once.

The linear kernel allows us to do all of this when \(n \gg d\).
“You should be asking how the answers will be used and what is really needed from the computation. Time and time again someone will ask for the inverse of a matrix when all that is needed is the solution of a linear system; for an interpolating polynomial when all that is needed is its values at some point; for the solution of an ODE at a sequence of points when all that is needed is the limiting, steady-state value. A common complaint is that least squares curve-fitting couldn’t possibly work on this data set and some more complicated method is needed; in almost all such cases, least squares curve-fitting will work just fine because it is so very robust.”

Leader, Numerical Analysis and Scientific Computation