Some Notes on Kernel CRFs

Xiaojin Zhu

January 26, 2005

1 Import Vector Selection for Kernel CRFs

In the basic kernel CRF model, each clique c is associated with $|y|^{|c|}$ parameters $\alpha_j^c(\mathbf{y}_c)$. Even if we only consider vertex cliques, there would be hundreds of thousands of parameters for a typical protein dataset. This seriously affects the training efficiency.

To solve the problem, we adopt the notion of "import vector machines" [ZH01]. That is, we use a subset of the training examples instead of all of them. The subset is constructed by greedily selecting training examples one at a time to minimize the loss function:

$$\arg\min_k \mathcal{O}(f_{A\cup\{k\}}, \lambda) - \mathcal{O}(f_A, \lambda) \tag{1}$$

where

$$f_A(\mathbf{x}, \mathbf{y}) = \sum_{j\in A} \alpha_j(\mathbf{y}) K(\mathbf{x}_j, \mathbf{x}) \tag{2}$$

and A is the current *active import vector set.*

(1) is hard to compute: we need to update all the parameters for $f_{A\cup\{k\}}$. Even if we keep old parameters in f_A fixed, we still need to use expensive forward-backward algorithm to train the new parameters $\alpha_k(\mathbf{y})$ and compute the loss. Following [McC03], we make a set of speed up approximations.

Approximation 1: Mean field approximation. With the old f_A we have an old distribution $P(\mathbf{y}|\mathbf{x}) = 1/Z \exp(\sum_c f_A^c(\mathbf{x}, \mathbf{y}))$ over a label sequence y. We approximate $P(\mathbf{y}|\mathbf{x})$ by the mean field

$$P_o(\mathbf{y}|\mathbf{x}) = \prod_i P_o(y_i|x_i) \tag{3}$$

i.e. the mean field approximation is the independent product of marginal distributions at each position i. It can be computed with the Forward-Backward algorithm on $P(\mathbf{y}|\mathbf{x})$.

Approximation 2: Consider only the vertex kernel. In conjuction with the mean field approximation, we only consider the vertex kernel $K(x_i, x_j)$ and ignore edge or other higher order kernels. The loss function becomes

$$\mathcal{O}(f_A, \lambda) = -\sum_{i\in T} \log P_o(y_i|x_i) + \frac{\lambda}{2} \sum_{i,j\in A} \sum_y \alpha_i(y)\alpha_j(y) K(x_i, x_j) \tag{4}$$

where $T = \{1, \ldots, M\}$ is the set of training positions on which to evaluate the loss function. Once we add a candidate import vector x_k to the active set, the new model is

$$P_n(y_i|x_i) = \frac{P_o(y_i|x_i) \exp(\alpha_k(y_i) K(x_i, x_k))}{\sum_y P_o(y|x_i) \exp(\alpha_k(y) K(x_i, x_k))} \tag{5}$$

The new loss function is

$$\mathcal{O}(f_{A\cup\{k\}}, \lambda) = -\sum_{i\in T} \log P_n(y_i|x_i) + \frac{\lambda}{2} \sum_{i,j\in A\cup\{k\}} \sum_y \alpha_i(y)\alpha_j(y) K(x_i, x_j) \tag{6}$$

And (1) can be written as

$$\begin{aligned}\mathcal{O}(f_{A\cup\{k\}},\lambda)-\mathcal{O}(f_A,\lambda) &= -\sum_{i\in T}\alpha_k(y_i)K(x_i,x_k) && (7)\\ &+\sum_{i\in T}\log\sum_y P_o(y|x_i)\exp(\alpha_k(y)K(x_i,x_k)) && (8)\\ &+\lambda\sum_{j\in A}\sum_y \alpha_j(y)\alpha_k(y)K(x_j,x_k)+\frac{\lambda}{2}\sum_y \alpha_k^2(y)K(x_k,x_k) && (9)\end{aligned}$$

This change of loss is a convex function of the $|y|$ parameters $\alpha_k(y)$. We can find the best parameters with Newton's method. The first order derivatives are

$$\begin{aligned}\frac{\partial\mathcal{O}(f_{A\cup\{k\}},\lambda)-\mathcal{O}(f_A,\lambda)}{\partial\alpha_k(y)} &= -\sum_{i\in T}K(x_i,x_k)\delta(y_i,y) && (10)\\ &+\sum_{i\in T}P_n(y|x_i)K(x_i,x_k) && (11)\\ &+\lambda\sum_{j\in A\cup\{k\}}\alpha_j(y)K(x_j,x_k) && (12)\end{aligned}$$

And the second order derivatives are

$$\begin{aligned}\frac{\partial^2\mathcal{O}(f_{A\cup\{k\}},\lambda)-\mathcal{O}(f_A,\lambda)}{\partial\alpha_k(y)\partial\alpha_k(y')} &= \sum_{i\in T}\left[P_n(y|x_i)K^2(x_i,x_k)\delta(y,y')-P_n(y|x_i)K^2(x_i,x_k)P_n(y'|x_i)\right] && (13)\\ &+\lambda K(x_k,x_k)\delta(y,y') && (14)\end{aligned}$$

Approximation 1 and 2 allow us to estimate the change in loss function independently for each position in T. This avoids the need of dynamic programming. Although the time complexity to evaluate each candidate x_k is still linear in $|T|$, we save by a (potentially large) constant factor. Further more, they allow a more dramatic approximation as shown next.

Approximation 3: Sparse evaluation of likelihood. A typical protein database has around 500 sequences, with hundreds of amino acid residuals per sequence. Therefore M, the total number of training positions, can easily be around 100,000. Normally $T=\{1,\ldots,M\}$, i.e. we need to sum over all training positions to evaluate the log-likelihood. However we can speed up by reducing T. There are several possibilities:

1. Focus on errors: $T=\{i|y_i\neq\arg\max_y P_o(y|x_i)\}$
2. Focus on low confidence: $T=\{i|P_o(y_i|x_i)<p_0\}$
3. Skip positions: $T=\{ai|ai\leq M; a,i\in N\}$
4. Random sample: $T=\{i|i\sim uniform(1,M)\}$
5. Error/confidence guided sample: errors / low confidence positions have higher probability to be sampled.

We need to scale the log likelihood term to maintain the balance between it and the regularziation term:

$$\mathcal{O}(f_A,\lambda)=-\frac{M}{|T|}\sum_{i\in T}\log P_o(y_i|x_i)+\frac{\lambda}{2}\sum_{i,j\in A}\sum_y\alpha_i(y)\alpha_j(y)K(x_i,x_j) \tag{15}$$

and scale the derivatives accordingly.

Other approximations: We may want to add more than one candidate import vector to A at a time. But we need to eliminate redundant vectors, possibly by the kernel distance. We may not want to fully train $f_{A\cup\{k\}}$ once we selected k.

2 Kernel CRF training with Active Set

For the special case of a linear CRF, and with the active set A, the loss function has the form

$$\begin{aligned}
\mathcal{O}(A,\lambda) &= -\sum_{s\in\text{seq}}(\sum_{i=1}^{|s|}\sum_{j\in A}\alpha_j(y_i)K(x_i,x_j)+\sum_{i=1}^{|s|-1}\beta(y_i,y_{i+1})) && (16)\\
&+\sum_{s\in\text{seq}}\log\sum_{\vec{\mathbf{z}}}\exp(\sum_{i=1}^{|s|}\sum_{j\in A}\alpha_j(z_i)K(x_i,x_j)+\sum_{i=1}^{|s|-1}\beta(z_i,z_{i+1})) && (17)\\
&+\frac{\lambda}{2}\left(\sum_y\sum_{i\in A}\sum_{j\in A}\alpha_i(y)\alpha_j(y)K(x_i,x_j)+\sum_y\sum_z\beta^2(y,z)\right) && (18)
\end{aligned}$$

The gradient for the parameter $\alpha_j(y), j\in A$ are

$$\begin{aligned}
\frac{\partial\mathcal{O}(A,\lambda)}{\partial\alpha_j(y)} &= -\sum_{s\in\text{seq}}\sum_{i=1}^{|s|}\delta(y_i,y)K(x_i,x_j) && (19)\\
&+\sum_{s\in\text{seq}}\frac{\sum_{\vec{\mathbf{z}}}\exp(\sum_{i=1}^{|s|}\sum_{j\in A}\alpha_j(z_i)K(x_i,x_j)+\sum_{i=1}^{|s|-1}\beta(z_i,z_{i+1}))(\sum_{i=1}^{|s|}\delta(z_i,y)K(x_i,x_j))}{\sum_{\vec{\mathbf{z}}}\exp(\sum_{i=1}^{|s|}\sum_{j\in A}\alpha_j(z_i)K(x_i,x_j)+\sum_{i=1}^{|s|-1}\beta(z_i,z_{i+1}))} && (20)\\
&+\lambda\sum_{i\in A}\alpha_i(y)K(x_i,x_j) && (21)\\
&= -\sum_{s\in\text{seq}}\sum_{i=1}^{|s|}\delta(y_i,y)K(x_i,x_j) && (22)\\
&+\sum_{s\in\text{seq}}\sum_{\vec{\mathbf{z}}}P(\vec{\mathbf{z}})(\sum_{i=1}^{|s|}\delta(z_i,y)K(x_i,x_j)) && (23)\\
&+\lambda\sum_{i\in A}\alpha_i(y)K(x_i,x_j) && (24)\\
&= -\sum_{s\in\text{seq}}\sum_{i=1}^{|s|}\delta(y_i,y)K(x_i,x_j) && (25)\\
&+\sum_{s\in\text{seq}}\sum_{i=1}^{|s|}\sum_{z_1}\cdots\sum_{z_{|s|}}P(z_1\cdots z_{|s|})\delta(z_i,y)K(x_i,x_j) && (26)\\
&+\lambda\sum_{i\in A}\alpha_i(y)K(x_i,x_j) && (27)\\
&= -\sum_{s\in\text{seq}}\sum_{i=1}^{|s|}\delta(y_i,y)K(x_i,x_j)+\sum_{s\in\text{seq}}\sum_{i=1}^{|s|}P(y)K(x_i,x_j)+\lambda\sum_{i\in A}\alpha_i(y)K(x_i,x_j) && (28)
\end{aligned}$$

Similarly,

$$\frac{\partial \mathcal{O}(A,\lambda)}{\partial \beta(y_1,y_2)} = -\sum_{s\in\text{seq}}\sum_{i=1}^{|s|-1}\delta(y_i,y_1)\delta(y_{i+1},y_2) \tag{29}$$

$$+\sum_{s\in\text{seq}}\sum_{\vec{\mathbf{z}}}P(\vec{\mathbf{z}})\sum_{i=1}^{|s|-1}\delta(z_i,y_1)\delta(z_{i+1},y_2) \tag{30}$$

$$+\lambda\beta(y_1,y_2) \tag{31}$$

$$= -\sum_{s\in\text{seq}}\sum_{i=1}^{|s|-1}\delta(y_i,y_1)\delta(y_{i+1},y_2)+\sum_{s\in\text{seq}}\sum_{i=1}^{|s|-1}P(y_1,y_2)+\lambda\beta(y_1,y_2) \tag{32}$$

3 Multiple Kernels

We can extend the framework by having multiple kernels on the vertex cliques,

$$K_{cq}(\mathbf{x},\mathbf{y}_c;\mathbf{x}',\mathbf{y}'_c) = K_{cq}(\mathbf{x},\mathbf{x}')\delta(\mathbf{y}_c,\mathbf{y}'_c) \tag{33}$$

where the clique c is a single vertex, and q is an index over different kernels (for example an RBF kernel and a semi-supervised kernel). Note the kernels $K_{cq}(\mathbf{x},\mathbf{x}')$ are not restricted to a single vertex as $K_q(x_c,x'_c)$: they can apply to sub-sequences of $\mathbf{x},\mathbf{x}'$ indexed by c.

3.1 Greedy Active Set

In terms of training position selection, now we need to greedily select both the active training positions and their kernels. The active set A are composed of (c,q) pairs to specify the position and kernel. The function under the active set is

$$f(x_i,y_i) = \sum_{(j,q)\in A}\alpha_{jq}(y_i)K_q(x_i,x_j) \tag{34}$$

The mean field loss function is

$$\mathcal{O}_{MF}(f_A,\Lambda) = -\frac{M}{|T|}\sum_{i\in T}\log P_o(y_i|x_i)+\sum_q\frac{\lambda_q}{2}\sum_{(i,q),(j,q)\in A}\sum_y\alpha_{iq}(y)\alpha_{jq}(y)K_q(x_i,x_j) \tag{35}$$

Now a candidate is a position/kernel pair (k,r). The new model after adding the candidate is

$$P_n(y_i|x_i) = \frac{P_o(y_i|x_i)\exp(\alpha_{kr}(y_i)K_r(x_i,x_k))}{\sum_y P_o(y|x_i)\exp(\alpha_{kr}(y)K_r(x_i,x_k))} \tag{36}$$

The new loss function is

$$\mathcal{O}_{MF}(f_{A\cup\{(k,r)\}},\Lambda) = -\frac{M}{|T|}\sum_{i\in T}\log P_n(y_i|x_i)+\sum_q\frac{\lambda_q}{2}\sum_{(i,q),(j,q)\in A\cup\{(k,r)\}}\sum_y\alpha_{iq}(y)\alpha_{jq}(y)K_q(x_i,x_j) \tag{37}$$

And the difference in loss can be written as

$$\mathcal{O}_{MF}(f_{A\cup\{(k,r)\}},\Lambda)-\mathcal{O}_{MF}(f_A,\Lambda) = -\frac{M}{|T|}\sum_{i\in T}\alpha_{kr}(y_i)K_r(x_i,x_k) \tag{38}$$

$$+\frac{M}{|T|}\sum_{i\in T}\log\sum_y P_o(y|x_i)\exp(\alpha_{kr}(y)K_r(x_i,x_k)) \tag{39}$$

$$+\lambda_r\sum_{(j,r)\in A}\sum_y\alpha_{jr}(y)\alpha_{kr}(y)K_r(x_j,x_k)+\frac{\lambda_r}{2}\sum_y\alpha_{kr}^2(y)K_r(x_k,x_k) \tag{40}$$

This is a convex function of the $|y|$ parameters $\alpha_{kr}(y)$. We can find the best parameters with Newton's method. The first order derivatives are

$$\begin{aligned}
\frac{\partial \mathcal{O}_{MF}(f_{A\cup\{(k,r)\}},\Lambda) - \mathcal{O}_{MF}(f_A,\Lambda)}{\partial \alpha_{kr}(y)} &= -\frac{M}{|T|}\sum_{i\in T} K_r(x_i,x_k)\delta(y_i,y) && (41)\\
&+\frac{M}{|T|}\sum_{i\in T} P_n(y|x_i)K_r(x_i,x_k) && (42)\\
&+\lambda_r \sum_{(j,r)\in A\cup\{(k,r)\}} \alpha_{jr}(y)K_r(x_j,x_k) && (43)
\end{aligned}$$

And the second order derivatives are

$$\begin{aligned}
\frac{\partial^2 \mathcal{O}_{MF}(f_{A\cup\{(k,r)\}},\Lambda) - \mathcal{O}_{MF}(f_A,\Lambda)}{\partial \alpha_{kr}(y)\partial\alpha_{kr}(y')} &= \frac{M}{|T|}\sum_{i\in T}\left[P_n(y|x_i)K_r^2(x_i,x_k)\delta(y,y') - P_n(y|x_i)K_r^2(x_i,x_k)P_n(y'|x_i)\right] && (44)\\
&+\lambda_r K_r(x_k,x_k)\delta(y,y') && (45)
\end{aligned}$$

We select the candidate $(k,r)*$ with the minimum mean field loss.

3.2 KCRF training with Active Set and Multiple Kernels

Once we add a selected (k,r) pair to the active set, we need to retrain all the active set parameters for KCRF. In the special case of linear CRFs, the loss function is

$$\begin{aligned}
\mathcal{O}(A,\Lambda) &= -\sum_{s\in\text{seq}}\left(\sum_{i=1}^{|s|}\sum_{(j,q)\in A}\alpha_{jq}(y_i)K_q(x_i,x_j) + \sum_{i=1}^{|s|-1}\beta(y_i,y_{i+1})\right) && (46)\\
&+\sum_{s\in\text{seq}}\log\sum_{\vec{\mathbf{z}}}\exp\left(\sum_{i=1}^{|s|}\sum_{(j,q)\in A}\alpha_{jq}(z_i)K_q(x_i,x_j) + \sum_{i=1}^{|s|-1}\beta(z_i,z_{i+1})\right) && (47)\\
&+\sum_q\frac{\lambda_q}{2}\sum_y\sum_{(i,q)\in A}\sum_{(j,q)\in A}\alpha_{iq}(y)\alpha_{jq}(y)K_q(x_i,x_j) + \frac{\lambda_\beta}{2}\sum_y\sum_z\beta^2(y,z) && (48)
\end{aligned}$$

The gradient for $\alpha_{jq}(y), (j,q)\in A$ is

$$\frac{\partial\mathcal{O}(A,\Lambda)}{\partial\alpha_{jq}(y)} = -\sum_{s\in\text{seq}}\sum_{i=1}^{|s|}\delta(y_i,y)K_q(x_i,x_j) + \sum_{s\in\text{seq}}\sum_{i=1}^{|s|}P(y|i)K_q(x_i,x_j) + \lambda_q\sum_{(i,q)\in A}\alpha_{iq}(y)K_q(x_i,x_j) \qquad (49)$$

Similarly,

$$\frac{\partial\mathcal{O}(A,\Lambda)}{\partial\beta(y_1,y_2)} = -\sum_{s\in\text{seq}}\sum_{i=1}^{|s|-1}\delta(y_i,y_1)\delta(y_{i+1},y_2) + \sum_{s\in\text{seq}}\sum_{i=1}^{|s|-1}P(y_1,y_2|i,i+1) + \lambda_\beta\beta(y_1,y_2) \qquad (50)$$

3.3 Greedy Selection with Multiple Kernels

It is found that the estimated loss function $\mathcal{O}_{MF}(f_{A\cup\{(k,r)\}},\Lambda)$ may be incomparable among different kernels. For example some kernels tend to consistently overestimate $\mathcal{O}_{MF}$. And candidates with these kernels are not selected even though their actural loss $\mathcal{O}$ are small. Other kernels may underestimate $\mathcal{O}_{MF}$ and whose candidates are always selected.

The ultimate solution is to compute the actural loss $\mathcal{O}(f_{A\cup\{(k,r)\}},\Lambda)$ for every candidate. However this requires several forward-backward passes for each candidate, which is computationally expensive. We observe that although comparison of $\mathcal{O}_{MF}$ is difficult among multiple kernels, it seems to be meaninful *within* the same kernels. Therefore we use the simple method: Compute $\mathcal{O}_{MF}(f_{A\cup\{(k,r)\}}, \Lambda)$ for all candidates (k, r). Then select the "winner" (k^*, r) separately for each kernel r. Finally compute the actual loss (also allow old parameters to vary) $\mathcal{O}(f_{A\cup\{(k^*,r)\}}, \Lambda)$ for the winners and pick the best one.

4 Experiments

4.1 Greedy selection with a single kernel

We first compute the greedy training points for the galaxy dataset. The dataset contains 100 sequences, each with 20 positions. We use the first 50 sequences as training and the rest as test. Figure 1 compares our greedy algorithm with the baseline of using all 1000 training positions. Here we use the semi-supervised kernel. With merely 2 selected training positions, the loss function is already reduced to 143.0, and the test set error rate is down to 6.6%. As comparison, with the full set of 1000 training positions, the loss is 121.7 and test error is 6.2%. Figure 2 shows the positions of the first few selected training points. Interestingly the first two selected positions are on the end of the spirals.

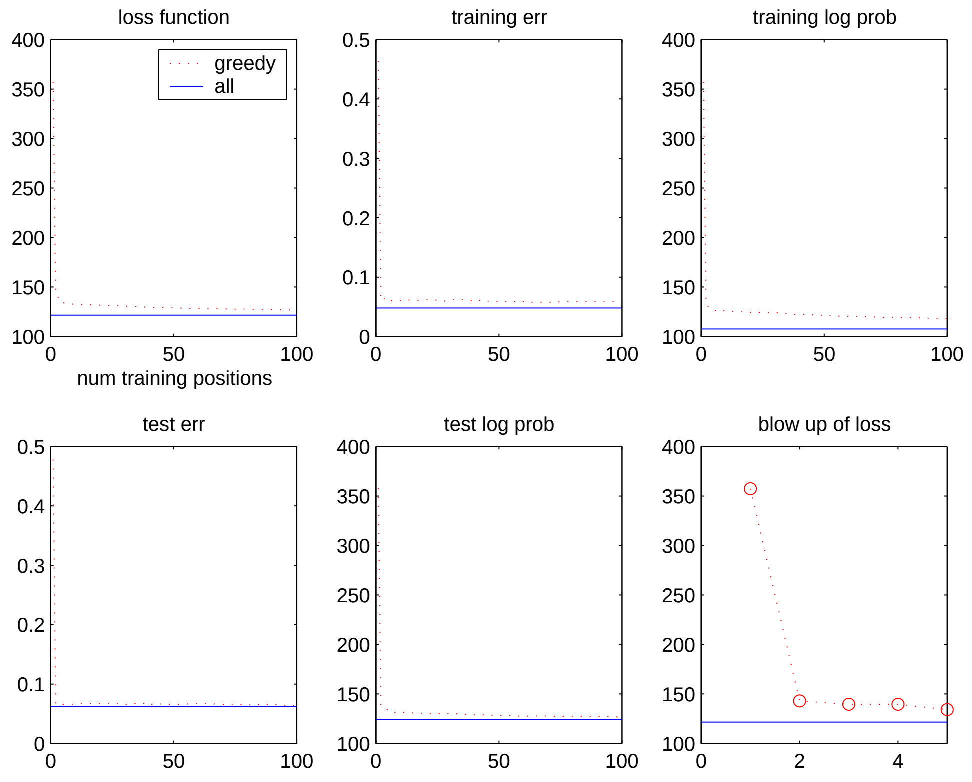

Figure 1: The galaxy data: greedy training position selection vs. using all training positions

Next we run greedy selection on a 'starfish' dataset, which is similar to the galaxy dataset except that each class now has multiple arms. The data looks like a 12-arms starfish, with 6 arms positive and 6 negative. Again there are 100 sequences with length 20, and the first half is used for training. We use the same method to construct a semi-supervised kernel as in the galaxy dataset. Will the greedy algorithm picks one point from the tip of each arm first? The answer is almost. The top 13 selected positions cover 11 arms. With these 13 positions, the risk=236.1, and test error = 11.5%. If we select more, say 100 positions, the risk=219.6,

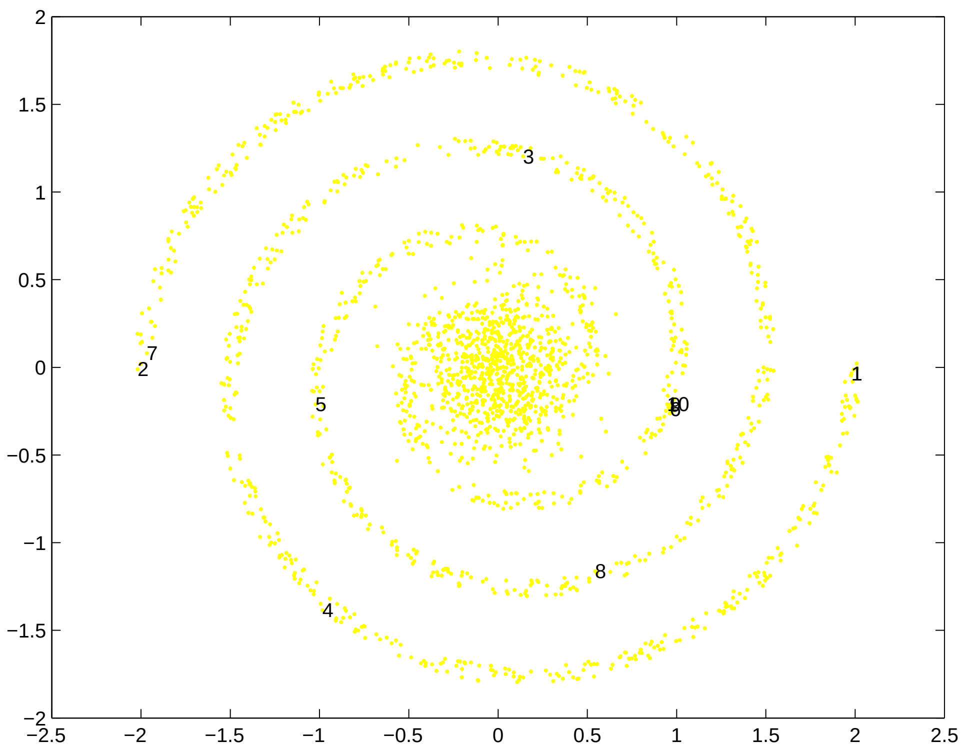

Figure 2: The galaxy data: the first 10 training points selected by the greedy algorithm

test error = 10.3%. The baseline of using all 1000 positions has risk=203.77, test error = 9.2%. Therefore greedy selection again recovers most of the benefit. The risk is plotted in Figure 3, and the first 13 selected training points are shown in Figure 4.

The baseline is trained by fminunc() with maxiter=1000, while in greedy selection, each new position is 'shallow' trained with only maxiter=30. If we run the final training of greedy selection after selecting 100 points with maxiter=1000 also, the risk=217.7, test error = 10.0%. There is some improvement but not much – it seems shallow training is not a big problem.

We then run greedy selection on a 'mini protein sequence' dataset with 5 training and 10 test protein sequences. The training sequences contain a total of 879 positions. The features are derived from PHD histograms of 13*21, and we use a second order polynomial kernel. Figure 5 shows the reduction of risk as we greedily select training positions. Table 1 lists the risk and test set error rate with different number of greedily selected training positions.

num greedy positions	risk	test error
150	7.3	40.5%
300	1.3	39.6%
879(full)	0.6	39.7%

Table 1: The mini protein data

There are several ways one can further speed up training. We compare the following particular methods on the mini protein dataset. These experiments are merely indications of speed up trends.

1. $C-1$ parameters: We can fix $\alpha_j(C) = 0, \forall j \in A$, observing that $\alpha_j(1), \cdots, \alpha_j(C-1)$ are sufficient to

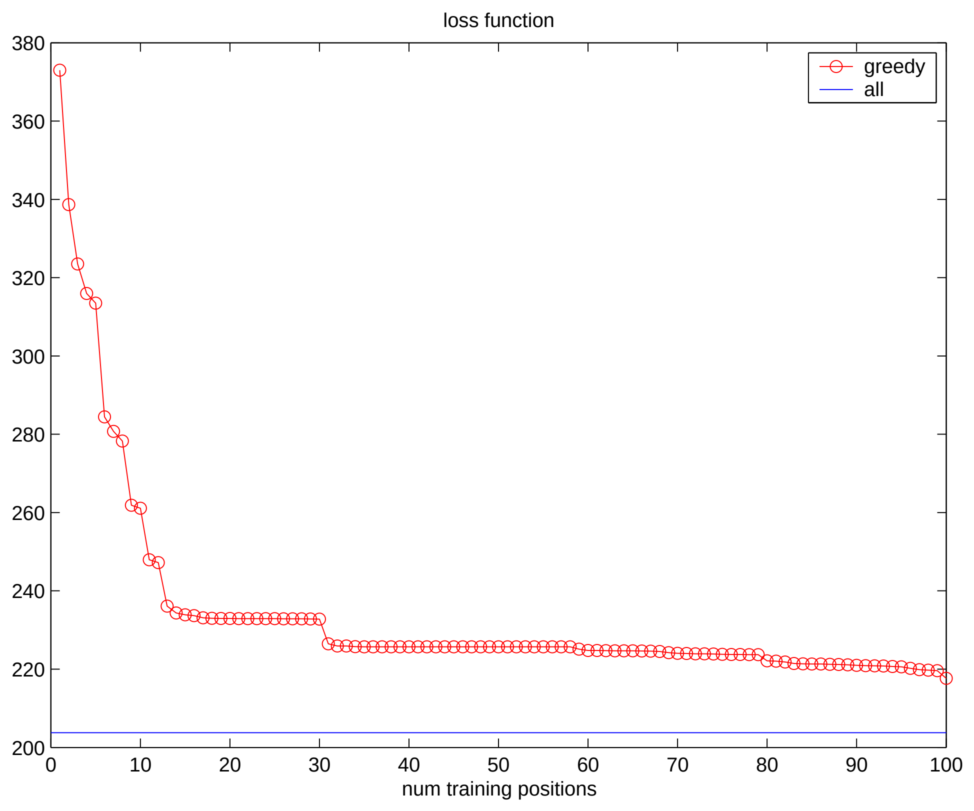

Figure 3: The starfish data: greedy training position selection vs. using all training positions

determine the probabilities, C is the number of classes. This simplification is not equivalent to using all C free parameters because of regularization. In practice we found the risk and test set accuracy to be inferior, as shown in Table 2 and Figure 6. The gain in speed is less than 4%. The experiment does not support going from C to $C - 1$ parameters per position.

2. T random: Instead of using all training positions for evaluation, we randomly sample some evaluation positions to form the evaluation set T. The sampling is repeated every time we add a new training position to the active set A.

3. MaxIter: After we add a new training position to A, we re-train the complete model with matlab fminunc(). Normally the MaxIter (maximum number of iterations) option for fminunc() is set to 30, which empirically fully trains the model. By setting it to a smaller number, we do not fully train the model.

4. Select multiple positions: at each iteration, select the top n positions and add them to the active set. Redundancy is not considered. The experiment shows that MaxIter=30 is not sufficient to train the multiple addition, and a larger setting needs to be used.

4.2 Greedy selection with multiple kernels

Let's revisit the galaxy dataset. The dataset is plotted in Figure 7 with the true labels. The sequence data is generated from a 2 state HMM with high probability (0.9) of staying in the same state. There

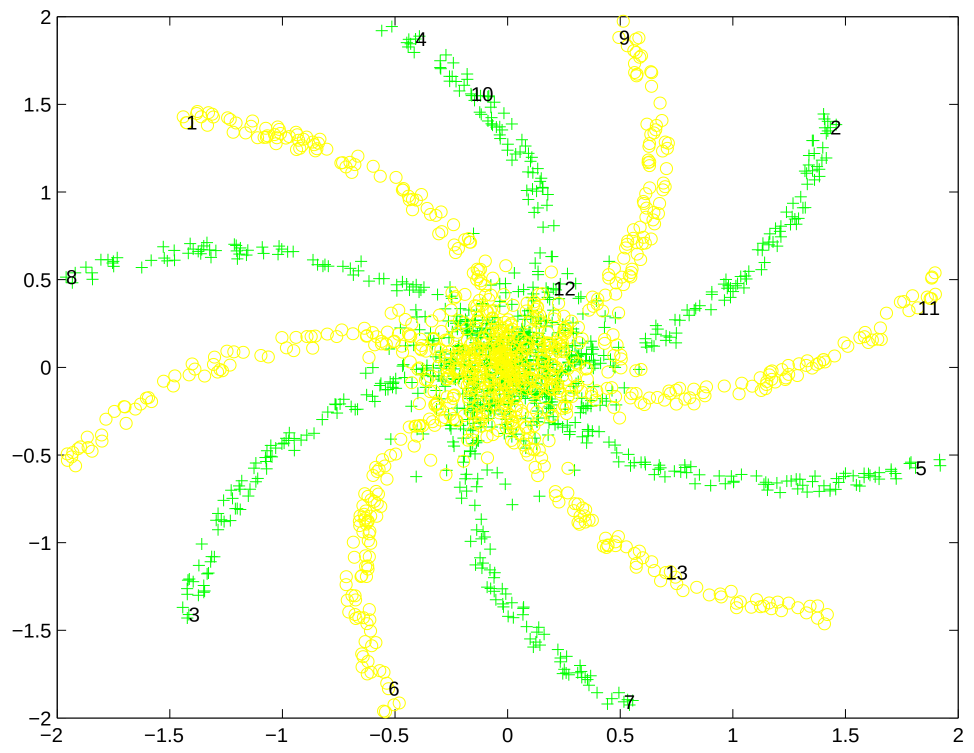

Figure 4: The starfish data: the first 13 training points selected by the greedy algorithm

are 100 sequences of length 20, and of which 50 sequences are the training set. We focus on two kernels: a (semi-supervised) graph kernel and a Gaussian RBF kernel. The graph kernel is constructed from the 10NN unweighted graph on the dataset, and has the form $K = 10(\Delta + (1/300)^2 I)^{-1}$. The RBF kernel is $K(i,j) = \exp(-||x_i - x_j||^2/0.35^2)$. Both the smoothing factor $(1/300)^2$ and the bandwidth term 0.35^2 are selected by maximizing the training sequence likelihood.

Before going into multiple kernels, let's look at the inner work of greedy training position selection with a single kernel. At each iteration we use the mean field estimation of change in loss function (40) to rank the candidate positions. It is illustrative to look at this quantity for the Galaxy dataset under different kernels. Figure 8 shows the estiamted change in loss of all candidates for the first four iterations, with the graph kernel alone. We found that

- The most informative candidate for the first position is at the tip of an arm (the first plot). We suspect that is because the tip has the least correlation with the core (where the labels are mixed) or the other arm. Also note the two tips have different estimated loss, which may be due to the particular data/label distribution.

- Once we add the tip (the second plot), the other tip becomes the most informative position.

- With the two tips in the active set (the third plot), the arms are almost determined and provide little further information. Thus the amrs are flat and close to zero. The informative points are in the core, but their usefulness is quite limited, as indicated by the scale of z-axis.

Similarly Figure 9 shows the same experiment, with the RBF kernel alone. This kernel fits less with the dataset. Also note its estimated change in loss is on a different scale than the graph kernel, which means we cannot directly compare them when using multiple kernels.

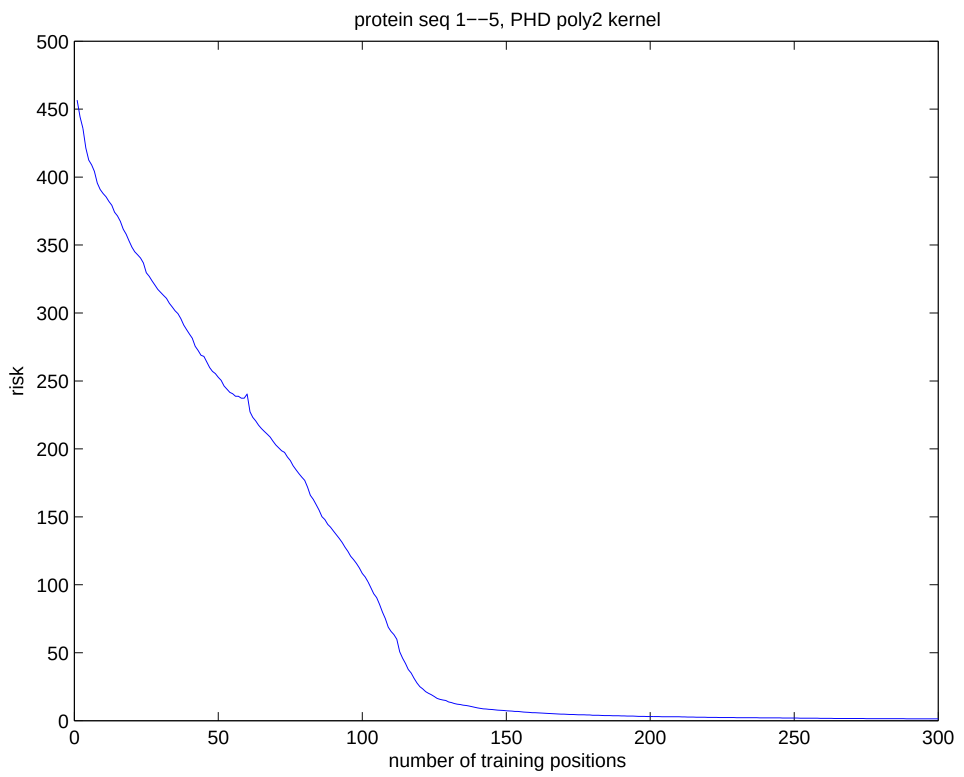

Figure 5: The mini protein data: risk of greedy training position selection. The full position baseline is risk=0.6

Now we move on to multiple kernels. Figure 10 shows the actual loss function with different kernel settings on the Galaxy dataset. In each setting we greedily selection 10 training position/kernel pairs.

- If we only use the graph kernel, the loss stablizes after two position/kernel pairs. This is consistent with Figure 8.
- If we only use the RBF kernel, the loss reduces more slowly. This is also consistent with Figure 9. RBF kernel is 'worse' than the graph kernel in general. However after the first selection, RBF has a better loss (339.63) than the graph kernel (341.41).
- If we use both kernels (by subjecting 'local winners' to the full training), surprisingly the active set selection is exactly the same as the RBF's. This at first looks puzzling, since we expect the extended freedom in candidates will lead to an active set no worse than that of the graph kernel's. We believe the greedy selection is to blame: the first RBF selection is indeed better than the graph selection, but this is a local optimum. It fails to discover that although picking the first graph selection is inferior initially, the second graph selection will dramatically reduce loss.
- To further confirm the greediness of the algorithm, we manually select the graph kernel's first position at the beginning. As state above, this selection alone is sub-optimal. Then we let the greedy selection algorithm continue. It successfully finds a better active set (note this is not guaranteed in general either).

We conclude that local optimum is a significant problem with multiple kernels. Using multiple kernels may not generate an active set that is better than using the individual kernels alone. We may want to add all

method	CPU time (seconds)	risk	test error
baseline	5807	7.3	40.5%
$C-1$ parameters	5577	15.4	47.3%
T 200 random	5390	41.3	42.6%
T 50 random	5310	75.7	42.4%
MaxIter=10	3186	161.2	46.6%
MaxIter=5	2514	187.2	40.5%
Select 10	620	970	43.6%
Select 10, MaxIter=90	1414	24.8	43.3%
A Random	452 ± 72	176.0 ± 19.3	$44.6 \pm 2.1\%$

Table 2: Comparing different speed up tricks on the mini protein data with 150 greedy training positions. The 'baseline' has C free parameters per position, uses all 879 training positions for evaluation, with MaxIter=30, and selects one training position per iteration. The 'A Random' line randomly selects 150 training positions and train it with MaxIter=1000; the numbers are the mean and standard error over 10 trials.

'local winners' to the active set, instead of picking the best local winner. Empirically this works very well, at the price of a larger active set.

References

[McC03] Andrew McCallum. Efficiently inducing features of conditional random fields. In *Nineteenth Conference on Uncertainty in Artificial Intelligence (UAI03)*, 2003.

[ZH01] Ji Zhu and Trevor Hastie. Kernel logistic regression and the import vector machine. In *NIPS 2001*, 2001.

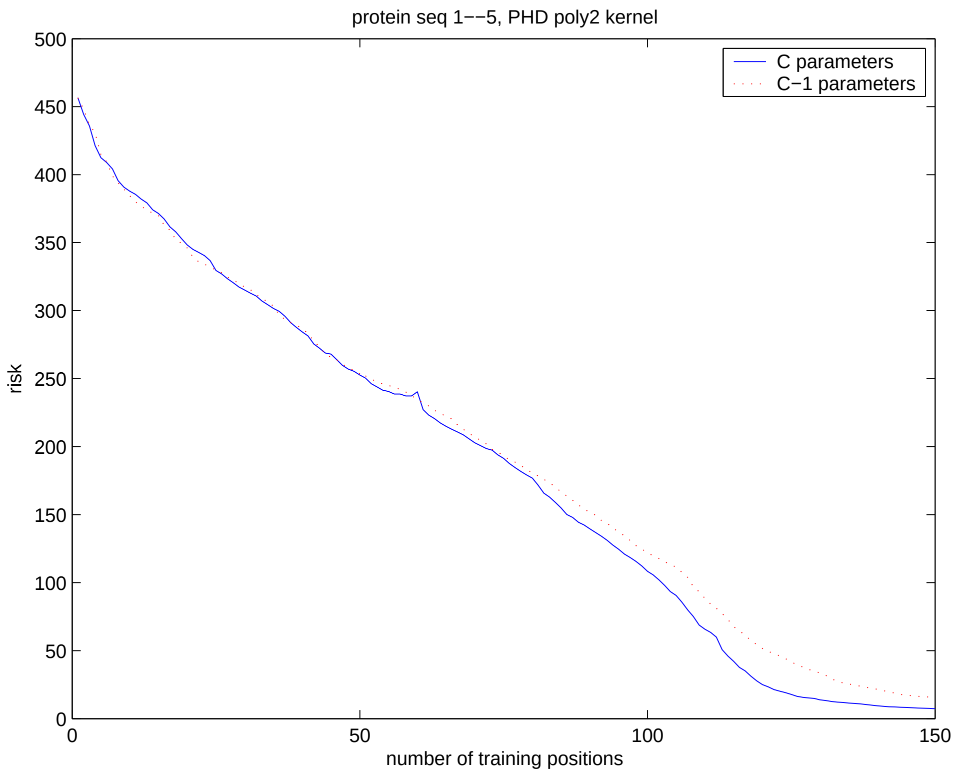

Figure 6: Risk on the mini protein data: C vs. $C-1$ free parameters per position

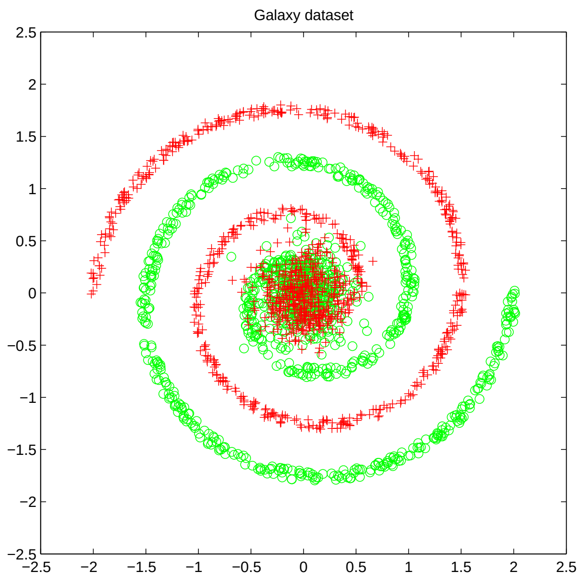

Figure 7: The galaxy dataset.

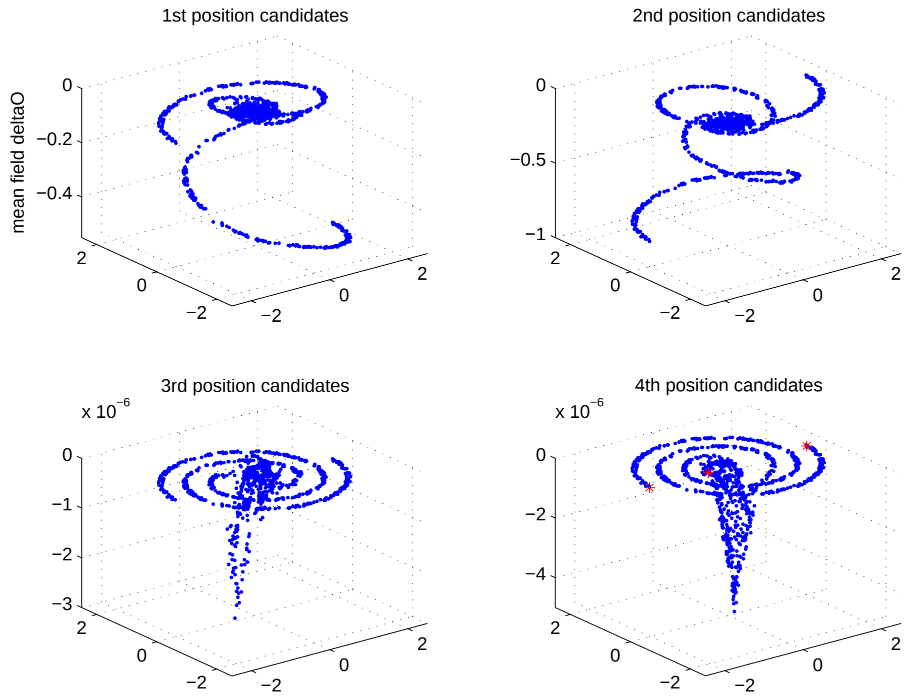

Figure 8: Mean field estimate of change in loss function with the *graph kernel* for the first four iterations on the galaxy dataset.

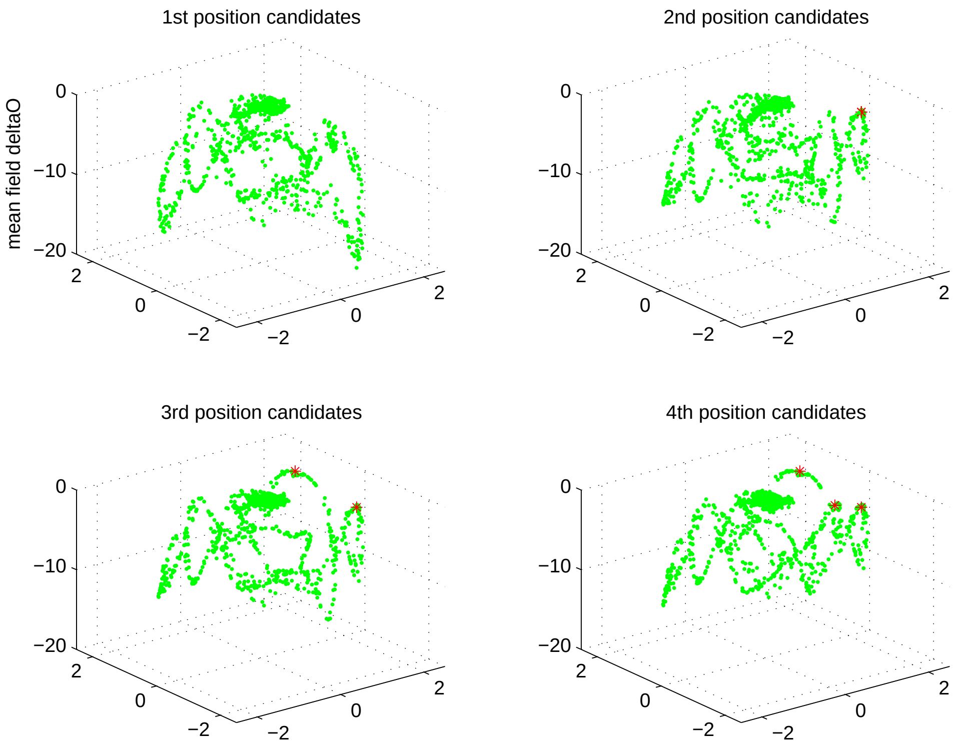

Figure 9: Mean field estimate of change in loss function with the *RBF kernel* for the first four iterations on the galaxy dataset.

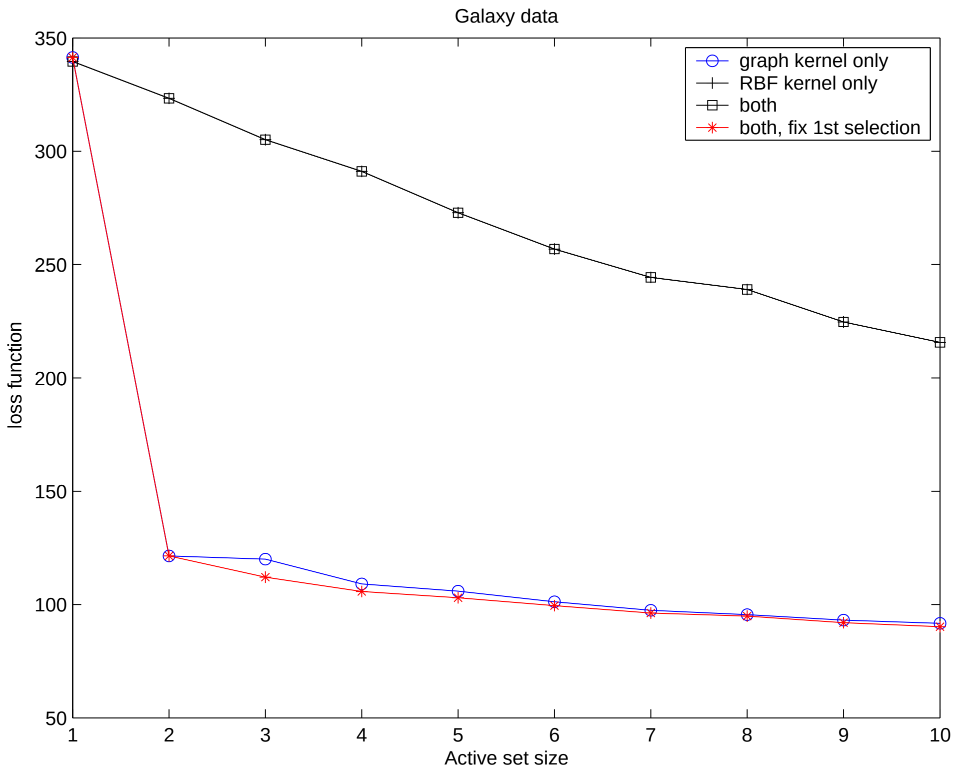

Figure 10: Actual loss with different kernels on the Galaxy dataset, with greedy active set selection