Statistical Methods for Machine Learning Experimental Project - Kernelized Linear Predictors

Lorenzo Lucarella

February 16, 2025

Contents

Declaration

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

1 Prelude

1.1 Usage

It is recommended to look into the ”proj” and ”results” notebooks. In both notebooks only the first cell needs to be evaluated in order to run the others. The ”searches.py” script is provided to run all searches from the command line.

2 Preprocessing and Testing

As prescribed by the project, all learning algorithms will be evaluated with a 5-fold cross validation, reporting the mean and maximum training loss. Rather than learning a predictor through a grid search, I'll run the grid search using the whole dataset as training data.

2.1 Data Assessment

The dataset, in $\mathcal{X} = \mathbb{R}^1 0 \times \mathcal{Y} = \{-1, +1\}$. All datapoints have the same features.

A quick visual inspection shows that the features x_3, x_6 and $x_1 0$ are linearly correlated across the whole dataset, although with some noise. Therefore, I will consider a preprocessing step in which either all features are kept or features x_6 and $x_1 0$ are removed.

2.2 Standardization

Feature standardization is a preprocessing step in which a map $S_{\mu,\sigma}(x) = \frac{x-\mu}{\sigma}$, with μ_i and σ_i being the estimates of $\mathbb{E}[X_i]$ and $\sqrt{Var[X_i]}$, is applied to the datapoints.

To prevent data leaks, the parameters will be learned from the training set: μ will be estimated using the sample mean, and σ will be estimated using the square root of the uncorrected sample variance.

2.3 Bias Term

A bias term is an artificial feature with a constant value added in preprocessing. Its main use in the context of linear predictors is to make it possible to express non-homogeneous separating hyperplanes, i.e. hyperplanes of the form $a + p(x)$; this extends to when such predictors are learned in the space induced by a polynomial feature map: adding a bias term before applying the feature map is an elegant way to introduce all lower-degree terms in the mapped space without explicitly accounting for them in the definition of the map.

I will add a bias term in preprocessing for all considered classifiers outside of the ones using the Gaussian kernel, as it can be trivially shown that it has no effect on the value of the kernel.

3 Feature Maps

In the context of classification with real-valued vectors, a feature map is a function $\phi : \mathbb{R}^d \to \mathbb{R}^{d'}$. Depending on the nature of the feature map, learning a simple separator in the mapped space can be equivalent to learning a more complex one in the original space.

Specifically, I will consider polynomial feature maps of degree n taking the form:

$$\phi_n^{poly}(x) = [\prod_{i \in 1 \ldots n} x_{c_i} : (c_1, \ldots c_n) \in C(n, 1 \ldots m)]$$

which allow a linear separator in the mapped space to act like a polynomial separator of degree n in the original space.

Such maps quickly become unfeasible to compute -let alone train classifiers with- given that $d' \sim d^{\Theta(n)}$. However, ϕ_2^{poly} can be used when d is low, such as in our case where $d = 11$.

4 Kernels

Again in the context of classification with real-valued features, a kernel is a function $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ such that there exists some feature map ϕ_K such that for all $(x, x') \in \mathcal{X}^2$ it holds that $K(x, x') = \langle \phi_K(x), \phi_K(x') \rangle$. The space induced by the feature map and therefore the kernel will be denoted as H_K, and the inner product as $\langle \cdot , \cdot \rangle_{H_K}$.

Due to the bilinearity of inner products, all elements of H_K can be represented as linear combinations of finitely many elements of $\mathcal{X}$, for instance as pairs $(\alpha, x) \in \mathbb{R}^n \times \mathcal{X}^n$ for some $n \in \mathbb{N}$. Then,

$$\langle (\alpha, x), (\alpha', x') \rangle_{H_K} = \sum_{i \in 1 \ldots n} \sum_{j \in 1 \ldots n'} \alpha_i \alpha'_j K(x_i, x'_j)$$

.

With this representation and given an upper bound t_K on the time complexity of a kernel evaluation, an inner product evaluation would have a time complexity of $\Theta(nn't_K)$, but if all kernel evaluations between pairs of elements of b and b' are known this drops to $\Theta(nn')$; this will be useful for kernelized Perceptron and Pegasos, where the $\Theta(m^2 t_K)$ time complexity of precomputing all kernel evaluations for pairs of elements in the training set turns out to be worthwhile.

4.1 Polynomial Kernel

I consider the biased polynomial kernel

$$K_n^{poly}(x, x') = 1 + \langle x, x' \rangle^n$$

with $n \in \{2, 3, 4\}$.

This kernel induces a polynomial feature map of degree n, identical to the one introduced earlier outside of the coefficients of the terms not being always 1 and the introduction of constant term; although as previously mentioned the bias term already introduces a constant term along with all the linear terms, I'll use this definition for the sake of simplicity.

4.2 Gaussian Kernel

I consider the Gaussian kernel

$$K_\sigma^{Gauss}(x, x') = exp(-\frac{1}{2\sigma^2}\|x - x'\|^2)$$

. This kernel induces a feature map to an infinitely-dimensional feature space which is much akin to an infinite-degree polynomial feature map.

4.2.1 Tuning σ

K^{Gauss} is readily interpretable as a distance function which outputs negligible values as soon as the Euclidean distance between the two arguments significantly exceeds σ, effectively becoming a "soft" neighborhood predicate. Therefore, a Gaussian kernelized linear classifier is similar in nature to kNN, and indeed with the correct choice of σ it can become consistent.

However given the necessarily small values of m and T, rather than trying to achieve consistency I will instead consider values for σ that are comparable to (a quick estimate of) the mean distance between elements of the training set, introducing a parameter ρ and defining

$$\sigma_\rho = \frac{\rho}{m} \sum_{i \in 1 \ldots m} \|x_i - x'_i\|$$

with x' being a random permutation of the datapoints in the training set.

5 Perceptron

The usual formulation of Perceptron learns a hyperplane w by updating it until it separates the entire training set. However, I consider a modified formulation which runs a fixed T updates, as the original variant runs the risk of never converging.

```
**Data:** $T \in \mathbb{N}, S \subseteq \mathbb{R}^{\lceil} \times \{-1, +1\}$
**Result:** $sgn(\langle w_t, \cdot \rangle)$
$w_0 \leftarrow 0$;
**for** $t \in 1...N$ **do**
    $x_t \leftarrow X_{t \bmod m}$;
    $y_t \leftarrow Y_{t \bmod m}$;
    $e_t \leftarrow \mathbb{I}\{y_t \langle w_{t-1}, x_t \rangle \leq 0\}$;
    $w_t \leftarrow w_{t-1} + e_t y_t x_t$;
**end**
```

Algorithm 1: Perceptron

Note that the actual implementation checks for convergence and terminates earlier if it can, but the result is the same.

Kernelized perceptron is identical up to the different representation of the separator.

6 Pegasos

Pegasos solves the SVM optimization problem by gradient descent using the hinge loss $l_h(y, \hat{y}) = max\{0, 1 - y\hat{y}\}$.

```
**Data:** $T \in \mathbb{N}, S \subseteq \mathbb{R}^{\lceil} \times \{-1, +1\}$
**Result:** $sgn(\langle \sum_{t \in 1 \ldots T} w_t, \cdot \rangle)$
$w_0 \leftarrow 0$;
**for** $t \in 1 \ldots T$ **do**
    $(x_t, y_t) \leftarrow$ a random element of $S$;
    $e_t \leftarrow \nabla l_h(\langle w_{t-1}, x_t \rangle, y_t) = \mathbb{I}\{y_t \langle w_{t-1}, x_t \rangle \leq 1\}$;
    $w_t \leftarrow \frac{t-1}{t} w_{t-1} + \frac{1}{\lambda t} \nabla l_t y_t x_t$;
**end**
```

Algorithm 2: Pegasos

I also consider a version that uses w_T as the underlying linear separator for the output predictor as opposed to the sum of all w, using the boolean parameter a to differentiate between the two. The expectation is that this version will yield a classifier which fits the training set better, which of course might introduce overfitting.

Note that in my code, the update for non-kernelized Pegasos is written like in Perceptron and the right term in the comparison is λt; this variant is equivalent to Pegasos proper for $\lambda \neq 0$, avoids $O(m)$ divisions in the training steps, is more readily understandable with respects to the kernelized variant, and makes it easier to introduce my approach to tuning λ.

Again, kernelized Pegasos is identical up to the representation of the separator.

Note that in the paper's pseudocode for Kernel Perceptron, the representation of the separator is slightly different to mine in that α is restrained to be in $\mathbb{N}^m$ rather than $\mathbb{Z}^m$, with the sign of the coefficients effectively being recovered through the labels; the author however wrongfully multiplies α_j with what my formulation expresses as y_t rather than with y_j in the terms of the summation which computes $\langle w_{t-1}, x_t \rangle_{H_K}$. In my code, I use the representation introduced earlier that lets α have negative coefficients.

6.1 Tuning λ

By rewriting Pegasos as seen before, it becomes apparent that there actually is a hard limit to λ which depends solely on the training data. Considering that for all $t \in \mathbb{N}_+$:

$$\begin{aligned}
&y_t \langle w_{t-1}, x_t \rangle \\
&\leq |\langle w_{t-1}, x_t \rangle| \\
&= |\sum_{i \in 1 \ldots t-1} e_i \langle x_i, x_t \rangle| \\
&\leq \sum_{i \in 1 \ldots t-1} |\langle x_i, x_t \rangle| \\
&\leq t(\max_{i \in 1 \ldots m} \|x_i\|^2)
\end{aligned}$$

we can conclude that for all

$$\lambda \geq \lambda_W = \max_{i \in 1 \ldots m} \|x_i\|^2$$

the algorithm will always behave in the same way by updating at every step, because the original term will always be less or equal to λt.

This introduces a natural upper limit to our range for λ, and the inequality

$$\mathbb{E}[F(\bar{w})] \leq F(w^\star) + 2\frac{X^2 log(T+1)}{\lambda T}$$

where $F[\cdot]$ is the SVM objective and X is the largest among the norms of the elements in the training set suggests that $\lambda = \Theta(\frac{log(T)}{T})$ will keep suboptimality stable in expectation.

I'll therefore introduce an auxiliary parameter T' and define $\lambda_{T'} = \lambda_{W^{0.98}} \frac{\log T'}{T'}$, where $\lambda_{W^{0.98}}$ is the 98th percentile of $\| \cdot \|^2$ for the datapoints in the training set rather than the maximum in order to account for outliers; all grid searches where this parameter is explored will consider $T' \in \{5m, 500m, 50000m\}$, under the assumption that this is a conservative way of picking λ.

Note that the given considerations also apply to kernelized Pegasos due to it being fully equivalent to Pegasos running in a feature mapped space.

7 Logistic Regression

The algorithm for logistic regression is very similar to Pegasos; the only difference is that the hinge loss is replaced with the logistic loss $l_{sgm}(y, \hat{y}) = 1 - e^{-y\hat{y}}$, which acts as a surrogate to it. Therefore, the algorithm is the exact same outside of the gradient being computed differently. Note that in this case I stick to the original algorithm.

8 Results

All tables refer to the best performer for each combination of the parameters in the leftmost column, with the remaining parameters listed in the central column and the losses on the training and test in the third column. The entries are sorted by loss on the test set.

9 Perceptron

9.1 No Feature Mapping (Pc)

The algorithm heavily underfits under any configuration and regardless of how many updates it's allowed to run (in fact, it fails to meaningfully improve after as little as 50000 updates); standardization gives marginally better results, but there's very little that can be done about the linear predictor faling to fit.

R	S	T	l_T	l_S
-	True	$5m$	0.3209	0.3179
x_6, x_{10}	True	$50000m$	0.3351	0.3340
-	False	$5m$	0.3816	0.3824
x_6, x_{10}	False	$5m$	0.3877	0.3892

9.2 Quadratic Feature Mapping ($Pc_{\phi_2^{poly}}$)

Given the increase in time complexity introduced by the feature mapping step, the maximum value for T is lowered to $5000m$. The feature map is applied after preprocessing both for consistency w.r.t. the kernelized variants and to ensure that the bias term is available in the input datapoints so that the predictor can also capture linear terms.

Standardization yields better results, but the algorithm is not quite as sensitive to it as Pe is -as long as it is given a large enough T-. Results are obviously better than with Perceptron, but the similarity between training and testing losses suggests that the algorithm still underfits.

R	S	T	l_T	l_S
-	True	$500m$	0.0715	0.0775
x_6, x_{10}	True	$5m$	0.0755	0.0785
-	False	$50000m$	0.0963	0.0925
x_6, x_{10}	False	$50000m$	0.1044	0.1079

9.3 Polynomial Kernelized ($Pc_{K^{poly}}$)

The complexity of an update step is now $\Theta(m)$, so the maximum value for T is again lowered to $50m$. Note that this already amply justifies caching the kernel evaluations: with caching the time complexity is $\Theta(m^2d + mT)$, while without it's $\Theta(mdT)$, with T being greater than m in all reasonable cases.

With $n = 2$ the behavior is in line with that of $Pc_{\phi_2^{poly}}$, both in terms of the loss over the test set and in how it compares to the loss over the training set. $n = 3$ performs best of all in terms of test loss, with the training loss still being comparable but clearly lower; on the other hand $n = 4$ overfits, being worse in comparison to $n = 3$ especially as it's given more epochs to properly fit to the training set; it still does perform better than $n = 2$, however.

With the limited number of updates allowed to the algorithm, standardization yields far better results. Perhaps with greater values of T we'd observe a similar behavior to $Pc_{\phi_2^{poly}}$, but this cannot be explored in this context. Interestingly, rather than merely providing competitive performance as seen with Pc and $Pc_{\phi_2^{poly}}$, feature removal turns out to be the outright better choice in this case.

n	S	R	T	l_T	l_S
3	True	x_6, x_{10}	$50m$	0.0207	0.0299
4	True	x_6, x_{10}	$50m$	0.0149	0.0420
2	True	x_6, x_{10}	$50m$	0.0706	0.0703
3	False	x_6, x_{10}	$5m$	0.3030	0.3045
4	False	x_6, x_{10}	$50m$	0.3442	0.3463
2	False	x_6, x_{10}	$5m$	0.3582	0.3619

9.4 Gaussian Kernelized ($Pc_{K^{Gauss}}$)

The same complexity considerations apply as with $Pc_{K^{poly}}$.

Unlike in the cases with polynomial feature maps, standardization doesn't have nearly as much of an effect in this case -and in fact it makes the algorithm perform worse as opposed to better, perhaps because in this case the magnitude of the features happens to be relevant. $\rho = 0.5$ and $\rho = 1$ are the best performers independently of other preprocessing, with values below underfitting and values above overfitting; this suggests that the choice for tuning σ is at least sensible, although the high sensitivity of the algorithm to this parameter suggests that the grid is too coarse.

In all cases where performance is acceptable to begin with, $T = 50m$ performs far better than $T = 5$.

ρ	S	R	T	l_T	l_S
0.5	False	-	$50m$	0.0061	0.0265
0.5	True	x_6, x_{10}	$50m$	0.0069	0.0335
1.0	False	-	$50m$	0.0302	0.0402
1.0	True	x_6, x_{10}	$50m$	0.0351	0.0430
0.25	False	-	$50m$	0.0000	0.0497
0.25	True	-	$50m$	0.0000	0.0563
4.0	False	-	$50m$	0.0842	0.0853
2.0	True	-	$50m$	0.0927	0.0955
2.0	False	-	$50m$	0.0955	0.0974
0.125	False	x_6, x_{10}	$50m$	0.0000	0.1022
0.125	True	x_6, x_{10}	$5m$	0.0000	0.1335
4.0	True	x_6, x_{10}	$50m$	0.1488	0.1521
8.0	False	x_6, x_{10}	$50m$	0.2728	0.2738
8.0	True	-	$5m$	0.3271	0.3293

10 Pegasos

The grids will be trimmed using information from the runs of the equivalent perceptron, and T will be kept the same as in Perceptron unless explicitly noted. Note that accumulative Pegasos effectively proved to be better in almost all cases, and unless otherwise noted the behavior of a given variant is usually similar to the equivalent Perceptron but with better training and testing errors.

10.1 No Feature Mapping (Pe)

Due to the larger size of the grid and the results generally being uninteresting, Pegasos will be run with $T = 5000m$ as the maximum as opposed to $T = 50000m$.

Lower values for T' appear to be better especially for smaller values of T, which means our approach for choosing λ was sensible in this case.

a	T	T'	R	S	l_T	l_S
True	$5000m$	$500m$	-	False	0.2679	0.2685
True	$500m$	$500m$	-	False	0.2689	0.2698
True	$500m$	$5m$	x_6, x_{10}	True	0.2693	0.2699
True	$5000m$	$5m$	-	True	0.2696	0.2700
False	$5000m$	$5m$	x_6, x_{10}	True	0.2694	0.2702
False	$500m$	$5m$	x_6, x_{10}	True	0.2693	0.2703
True	$50m$	$5m$	-	True	0.2703	0.2706
False	$5000m$	$500m$	x_6, x_{10}	True	0.2696	0.2707
True	$5m$	$5m$	x_6, x_{10}	True	0.2691	0.2708
False	$50m$	$5m$	-	True	0.2703	0.2710
True	$5000m$	$50000m$	x_6, x_{10}	True	0.2711	0.2711
False	$500m$	$500m$	-	False	0.2693	0.2717
True	$50m$	$500m$	-	False	0.2724	0.2729
True	$5m$	$500m$	-	True	0.2744	0.2742
True	$500m$	$50000m$	x_6, x_{10}	True	0.2747	0.2746
True	$5m$	$50000m$	-	True	0.2769	0.2775
True	$50m$	$50000m$	x_6, x_{10}	True	0.2774	0.2785
False	$5m$	$5m$	x_6, x_{10}	True	0.2865	0.2890
False	$5000m$	$50000m$	-	False	0.2955	0.2959
False	$500m$	$50000m$	-	False	0.2918	0.2964
False	$50m$	$500m$	x_6, x_{10}	True	0.2995	0.2986
False	$5m$	$500m$	-	False	0.3352	0.3362
False	$50m$	$50000m$	-	False	0.3577	0.3618
False	$5m$	$50000m$	x_6, x_{10}	True	0.3754	0.3774

10.2 Quadratic Feature Mapping ($Pe_{\phi_2^{poly}}$)

$T' = 5m$ leads to underfitting, while $T' = 500m$ and $T' = 50000m$ are ultimately comparable both in training and testing error regardless of whether if $T = 5m$ or $T = 50m$, so once again I consider my approach to tuning λ to have been sensible.

a	T	T'	R	S	l_T	l_S
True	$500m$	$50000m$	x_6, x_{10}	True	0.0420	0.0444
True	$5m$	$50000m$	x_6, x_{10}	True	0.0445	0.0462
True	$50m$	$50000m$	x_6, x_{10}	True	0.0430	0.0463
True	$50m$	$500m$	-	True	0.0433	0.0468
True	$5m$	$500m$	x_6, x_{10}	True	0.0442	0.0469
True	$500m$	$500m$	-	True	0.0439	0.0477
False	$500m$	$500m$	x_6, x_{10}	True	0.0445	0.0483
False	$50m$	$500m$	x_6, x_{10}	True	0.0468	0.0504
False	$500m$	$50000m$	x_6, x_{10}	True	0.0509	0.0588
False	$5m$	$500m$	x_6, x_{10}	True	0.0587	0.0616
False	$50m$	$50000m$	x_6, x_{10}	True	0.0724	0.0707
False	$5m$	$50000m$	x_6, x_{10}	True	0.0750	0.0751
False	$5m$	$5m$	x_6, x_{10}	True	0.0800	0.0832
True	$50m$	$5m$	x_6, x_{10}	True	0.0812	0.0841
True	$500m$	$5m$	x_6, x_{10}	True	0.0810	0.0850
False	$50m$	$5m$	x_6, x_{10}	True	0.0814	0.0850
True	$5m$	$5m$	x_6, x_{10}	True	0.0816	0.0850
False	$500m$	$5m$	x_6, x_{10}	True	0.0814	0.0853

10.3 Polynomial Kernelized ($Pe_{K^{poly}}$)

Given that $T' = 50000m$ is clearly better than the alternatives, with $T' = 5m$ underfitting, the approach for tuning λ likely was too conservative in this case.

n	T	T'	a	R	S	l_T	l_S
3	$50m$	$50000m$	True	x_6, x_{10}	True	0.0106	0.0224
4	$50m$	$50000m$	True	x_6, x_{10}	True	0.0099	0.0367
3	$50m$	$500m$	True	x_6, x_{10}	True	0.0307	0.0395
3	$5m$	$50000m$	True	x_6, x_{10}	True	0.0335	0.0427
2	$50m$	$50000m$	True	x_6, x_{10}	True	0.0428	0.0463
2	$50m$	$500m$	True	-	True	0.0427	0.0467
2	$5m$	$50000m$	True	x_6, x_{10}	True	0.0432	0.0470
3	$5m$	$500m$	True	x_6, x_{10}	True	0.0360	0.0473
2	$5m$	$500m$	True	x_6, x_{10}	True	0.0443	0.0474
4	$5m$	$50000m$	True	x_6, x_{10}	True	0.0268	0.0484
4	$50m$	$500m$	True	-	True	0.0308	0.0489
4	$5m$	$500m$	True	x_6, x_{10}	True	0.0342	0.0507
2	$50m$	$5m$	True	x_6, x_{10}	True	0.0764	0.0779
2	$5m$	$5m$	False	x_6, x_{10}	True	0.0750	0.0785
4	$5m$	$5m$	True	x_6, x_{10}	True	0.0897	0.0946
4	$50m$	$5m$	False	x_6, x_{10}	True	0.0886	0.0948
3	$5m$	$5m$	False	x_6, x_{10}	True	0.1050	0.1077
3	$50m$	$5m$	True	x_6, x_{10}	True	0.1040	0.1084

10.4 Gaussian Kernelized ($Pe_{K^{Gauss}}$)

$\rho = 0.5$ is again the best choice in terms of outright loss on the test set, $\rho = 1$ performs much better with regards to the best when comparing to $Pc_{K^{Gauss}}$; given that the training error is still quite close to the testing error, with higher values for T perhaps $\rho = 1$ could be the better choice outright. Interestingly, $T = 5m$ also performs well when $\rho = 0.5$. Though to a lesser extent than in $Pe_{K^{poly}}$ $T' = 5000m$ is the clear best choice, suggesting that the approach used to tune λ was again too conservative.

ρ	T	T'	a	R	S	l_T	l_S
0.5	$50m$	$50000m$	True	x_6, x_{10}	False	0.0016	0.0221
0.5	$50m$	$500m$	True	-	False	0.0026	0.0235
1.0	$50m$	$50000m$	True	x_6, x_{10}	True	0.0157	0.0266
0.5	$5m$	$50000m$	True	-	False	0.0134	0.0271
0.5	$5m$	$500m$	True	x_6, x_{10}	False	0.0135	0.0275
1.0	$50m$	$500m$	True	x_6, x_{10}	True	0.0217	0.0314
0.5	$50m$	$5m$	False	x_6, x_{10}	False	0.0331	0.0435
0.25	$50m$	$500m$	True	-	False	0.0000	0.0450
0.5	$5m$	$5m$	True	x_6, x_{10}	False	0.0350	0.0451
1.0	$5m$	$50000m$	True	x_6, x_{10}	True	0.0413	0.0454
1.0	$5m$	$500m$	True	x_6, x_{10}	False	0.0403	0.0461
0.25	$5m$	$50000m$	True	x_6, x_{10}	False	0.0036	0.0469
0.25	$50m$	$50000m$	True	x_6, x_{10}	False	0.0000	0.0470
0.25	$5m$	$500m$	True	x_6, x_{10}	False	0.0031	0.0474
0.25	$50m$	$5m$	False	x_6, x_{10}	False	0.0166	0.0477
0.25	$5m$	$5m$	True	x_6, x_{10}	False	0.0177	0.0489
1.0	$5m$	$5m$	False	x_6, x_{10}	False	0.0679	0.0715
1.0	$50m$	$5m$	True	x_6, x_{10}	False	0.0697	0.0726

11 Logistic Regression

In both cases, the algorithm performs very similarly to the equivalent Pegasos, with the unfortunate note of the accumulative variant not performing as well as Pegasos' does.

11.1 No Feature Mapping (Lr)

a	T'	R	S	T	l_T	l_S
True	$500m$	-	False	$50m$	0.2728	0.2737
True	$50000m$	-	False	$500m$	0.2723	0.2741
False	$500m$	-	False	$500m$	0.2742	0.2755
True	$5m$	-	True	$5m$	0.2766	0.2764
False	$5m$	-	True	$50m$	0.2782	0.2776
False	$50000m$	-	False	$500m$	0.3085	0.3124

11.2 Quadratic Feature Map ($Lr_{\phi_2^{poly}}$)

a	T'	R	S	T	l_T	l_S
True	$50000m$	x_6, x_{10}	True	$500m$	0.0464	0.0479
True	$500m$	x_6, x_{10}	True	$500m$	0.0483	0.0507
False	$500m$	-	True	$500m$	0.0478	0.0511
False	$50000m$	x_6, x_{10}	True	$500m$	0.0489	0.0523
False	$5m$	x_6, x_{10}	True	$5m$	0.0861	0.0888
True	$5m$	x_6, x_{10}	True	$500m$	0.0864	0.0895

12 Conclusions

The overall best performer among the trialled algorithms appears to be $Pe_{K_3^{poly}}$, although $Pe_{K^{Gauss}}$ may benefit with more iterations, and the choice for λ could likely be improved in both. Logistic gradient descent and Pegasos are both improvements over Perceptron in the given circumstances, especially when running in degree-2 feature spaces. Feature removal proved to be a good idea, consistently yielding comparable performances or even being slightly better outright.

References

[1] The course material from Nicolo Cesa-Bianchi's Statistical Methods For Machine Learning course. (`https://cesa-bianchi.di.unimi.it/MSA/index_23-24.html`)

[2] Shai Shalev-Shwartz, Yoram Singer, Nathan Srebro, Andrew Cotter Pegasos: Primal Estimated sub-GrAdient SOlver for SVM. (`https://home.ttic.edu/~nati/Publications/PegasosMPB.pdf`)