Support Vector Machines - Exercise session 1: Classification

Prepared by Ömer Yılmaz (r0779149)
omer.yilmaz@student.kuleuven.be

1. Exercises

1.1. A simple example: two Gaussians

- The centers of the Gaussians are around [1, 1] and [-1, -1]. We can think of a line passing through these centers. A normal to this line passing through the middle point of the centers would be an ideal decision boundary. Because, their spread is also equal with 1 std. Their covariance matrix is the same. This line can be characterized with passing through [1, -1] and [-1, 1] points and be seen in Fig. 1.

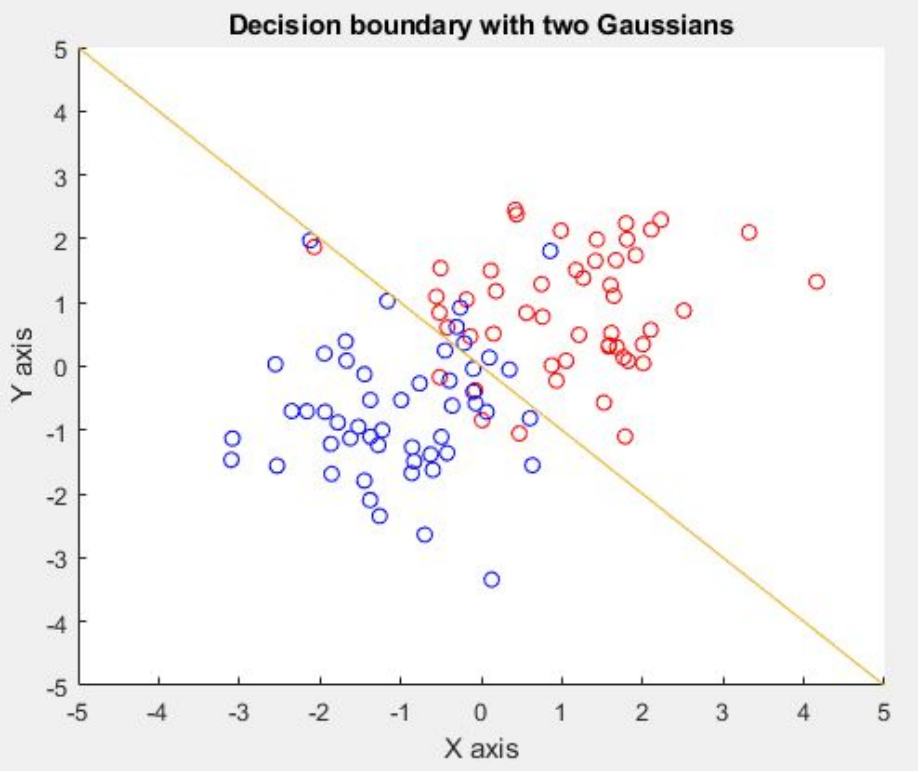


Fig. 1: The line classifying 2 Gaussian datasets

- This line may not be optimum in the sense that it doesn't classify some points correctly and some other boundaries, perhaps even linear ones, may supply this feature. However, when we run this random datasets creation experiment more and more, the best linear decision boundary converges to this normal line with most accuracy. As long as both datasets have the same covariance matrix and each diagonal unit is scaled equally, this line is the best linear fit. Nonlinear methods like RBP or multilayer perceptron can be used for other optimum definitions. Splitting methods may change the optimum momentarily though in the long run of this experiment they also converge to the normal distribution.

1.2. Support vector machine classifier

- Support vectors are the points of the training set of the SVM model. They form the actual weight of the main support vector inequality (for LS-SVM equation) as below.

$$y[w^T x + b] \geq 1, \quad \text{for LS-SVM: } y[w^T x + b] = 1$$

When it is nonlinear we replace x with φ(x). We actually don't calculate w and φ(x). Instead we solve a dual problem and the solution of the optimization problem gives us the weight vector as a combination of support vectors scaled by support values as shown below.

$$w = \sum_{k=1}^{N} \alpha_k y_k x_k$$

- When I add more points both on the right and wrong side, the hyperplane attempts to classify the new points too. However, it doesn't always succeed. When a point is placed such a way that the groups are no longer linearly separable, the linear kernel makes an error. When the C value is small, both kernels behave more freely with their decision boundaries and a more general decision boundary can be observed. However, when the C value is getting very big, they try to make as accurate estimations as possible creating somehow unintuitive decision boundaries. This also means high variance and low bias. For example, the RBF kernel creates different zones as can be seen in Fig. 2 rather than creating a big cluster.

My important observation is that when I add new points to the right side of the decision boundary of linear kernel, they don't become support vectors. However, when I add to the wrong side failing the inequality (in between or even further wrong), they become support vectors.

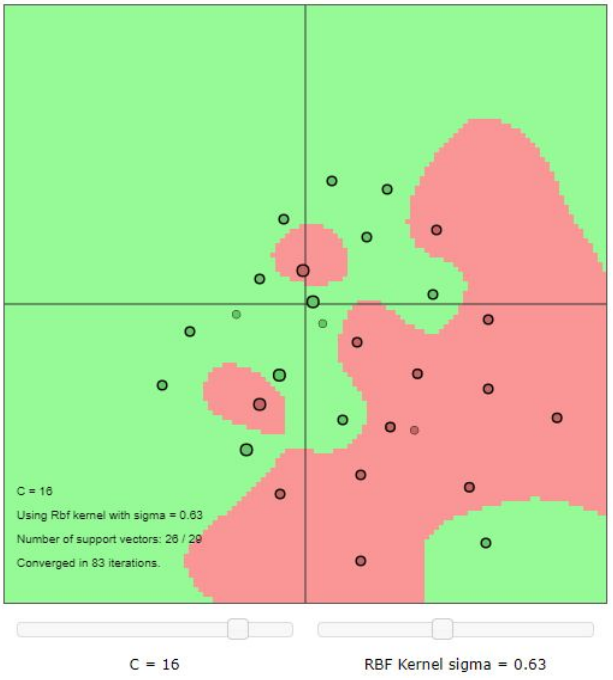


Fig. 2: An RBF kernel with comparatively high C value is trying to predict as accurate as possible

When the RBF kernel σ is getting bigger, the decision boundary tends to become linear. When it is small, because of the decreased variance of the Gaussians, the boundaries get nonlinear as in Fig. 3. This also means that there is high variance and the model is highly dependant on the support vectors.

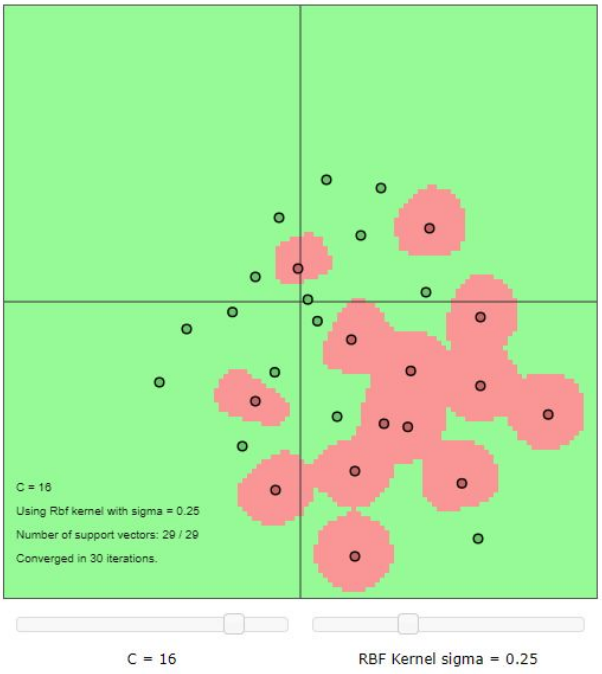


Fig. 3: When σ gets smaller, the RBF kernel creates sharper zones

RBF kernel performs better overall. Because linear kernel can only classify linearly separable datasets. When the dataset in this particular shape, we can go with the linear kernel. Though we can still classify correctly with RBF, linear kernel requires less support vectors as shown in Fig.4.

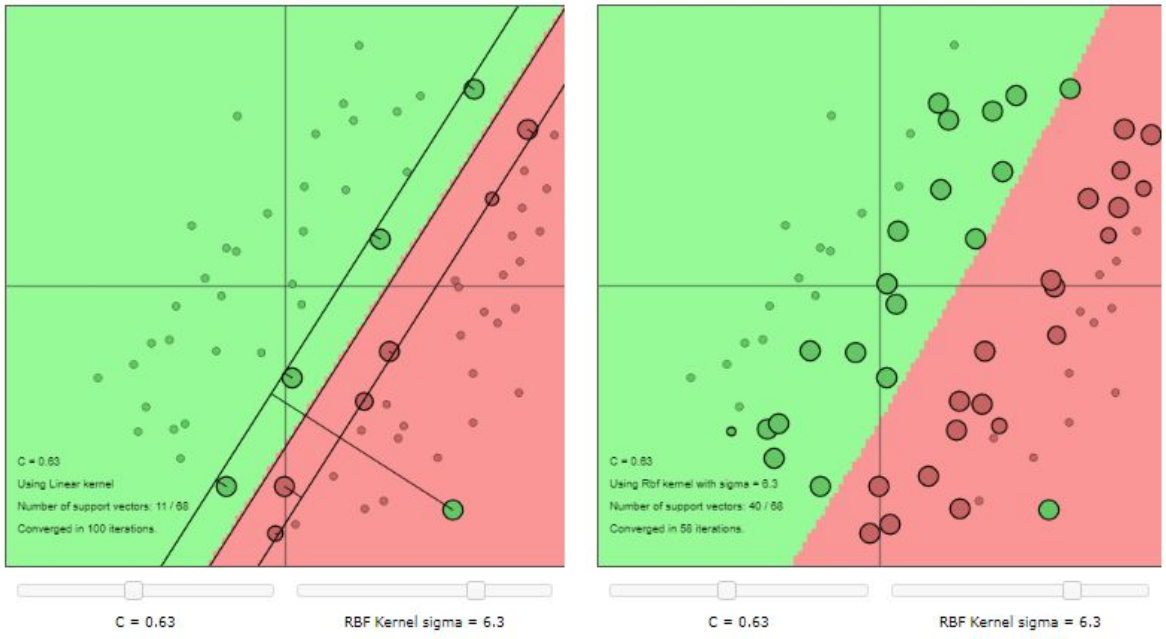


Fig. 4: Linear kernel compared to RBF kernel for a linearly separable dataset

1.3. Least-squares support vector machine classifier

1.3.1. Influence of hyperparameters and kernel parameters

- Degree 1 kernel couldn't classify all the data correctly. Because, the dataset was not linearly separable and the 1 degree polynomial is linear. Degree 2 kernel classified most of the dataset quite well although some points are not classified correctly. Degree 3 showed similar accuracy with the class 1 zone getting a bit more overfit around the points creating a cluster. The degree 4 kernel was quite nonlinear with no observable increase in the accuracy. When the degree of the kernels increased, the nonlinearity and the variance increased.
- 0.1 to 10 can be acceptable though 0.1 is a better choice because it is creating a cluster around the training points and it is not very strict which is suitable for new possible test value variants. And with a good choice of γ, more points can be classified correctly. 0.01 on the other hand is not general enough for test values.

With σ^2 being 0.1, the bigger γ had more penalty for wrong predictions on the cost function. When I chose 10 for γ, the model have performed well. 100 gave better accuracy; however, it relies on the training dataset more (higher variance).

- 10 and 100 γ values (σ^2=0.1) can be compared like in Fig 5.

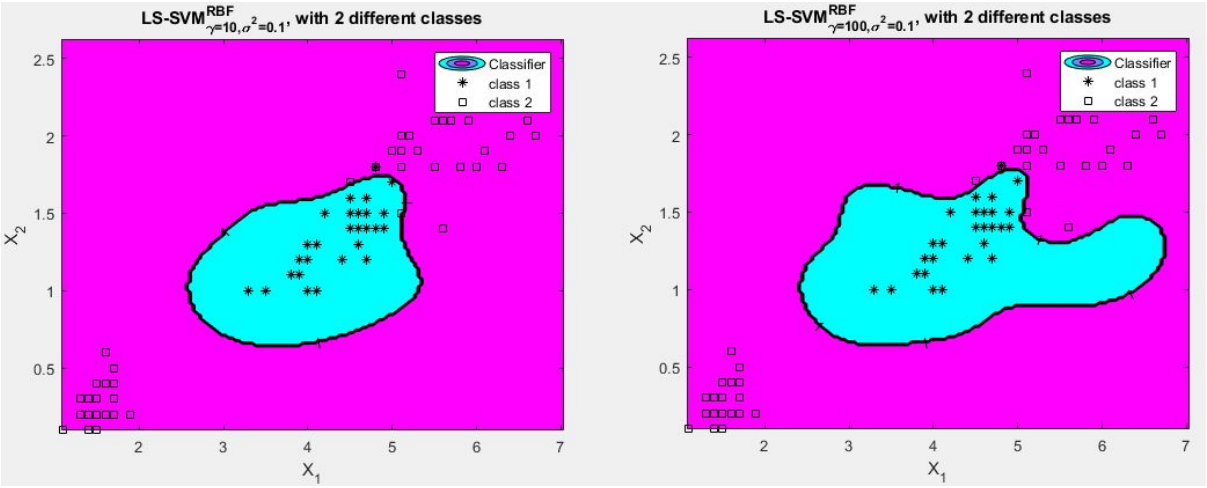


Fig. 5: a) Comparison of γ values 10 (σ^2=0.1) **b)** and 100 (σ^2=0.1)

1.3.2. Tuning parameters using validation

- Misclassification ratio for 80% training set - 20% validation set random split is given in Tab. 1. Best fit on the test set is visualized in Fig 6 a).

Error (inaccuracy)	γ = 1	γ = 100	γ = 10^4	γ = 10^6
σ^2 = 0.01	0	0.05	0.10	0
σ^2 = 0.1	0	0.10	0.05	0.15
σ^2 = 1	0	0	0.05	0
σ^2 = 10	0.05	0.10	0.05	0

Tab. 1: Misclassification errors for different γ and σ^2 values using (0.8) *rsplitvalidate()*

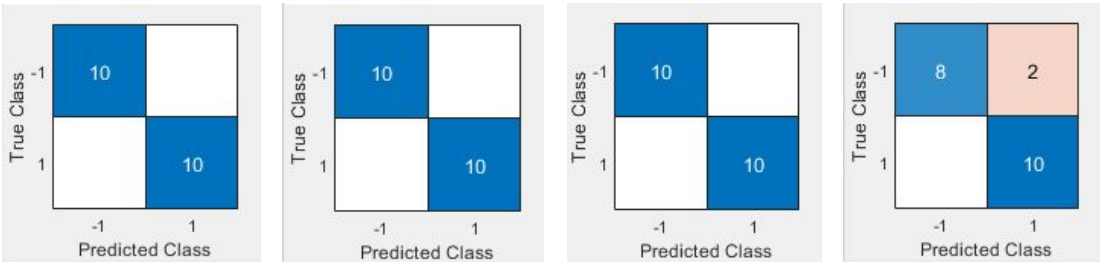


Fig. 6: Confusion chart of **a)** random split with γ = 1, σ^2 = 1 **b)** crossvalidation with γ = 10^6, σ^2 = 10 **c)** leave-one-out with γ = 10^6, σ^2 = 1 **d)** an unsuccessful attempt with random split

Misclassification ratio for 10-fold crossvalidation is given in Tab. 2. Best fit on the test set is visualized in Fig 6 b).

Error (inaccuracy)	**γ = 1**	**γ = 100**	**γ = 10^4**	**γ = 10^6**
σ^2 = 0.01	0.07	0.05	0.05	0.05
σ^2 = 0.1	0.04	0.05	0.06	0.10
σ^2 = 1	0.05	0.04	0.04	0.04
σ^2 = 10	0.05	0.05	0.05	0.03

Tab. 2: Misclassification ratio for different γ and σ^2 values using 10-fold *crossvalidate()*

Misclassification ratio for leave-one-out validation is given in Tab. 3. Best fit on the test set is visualized in Fig 6 c).

Error (inaccuracy)	**γ = 1**	**γ = 100**	**γ = 10^4**	**γ = 10^6**
σ^2 = 0.01	0.05	0.05	0.05	0.05
σ^2 = 0.1	0.04	0.04	0.05	0.12
σ^2 = 1	0.05	0.04	0.04	0.04
σ^2 = 10	0.05	0.04	0.05	0.04

Tab. 3: Misclassification ratio for different γ and σ^2 values using *leaveoneout()*

- Crossvalidation gives more generalized models than random splits. Because random splits do not always represent the data distribution well. Actually leave-one-out gives better results than crossvalidation. It is validating according to the one left point each time it iterates and runs through every point. Therefore, it is the maximum-fold crossvalidation an expected to have less variance towards small subsets of the dataset. However, in practice it takes long time to run for big datasets. So, one should try leave-one-out for smaller datasets. 100-fold for bigger datasets or 10-fold for even bigger ones should be preferred.

1.3.3. Automatic parameter tuning

- We can use the *tunelssvm()* method of the toolbox. It has two different implementations. One implementation is based on the brute force grid search method. The other one is Nelder-Mead (simplex) algorithm which uses a local search iteration method and returns when no better parameter set is discovered. There is a comparison of both algorithms in Tab. 4.

		1st iteration	2nd iteration	3rd iteration	4th iteration
Simplex Algorithm	**σ^2**	0.43	0.91	1.10	0.04
	γ	0.08	151.80	0.05	4.24
	Cost:	0.04	0.04	0.03	0.04
	Time:	0.42	0.40	0.52	0.41
Grid Search Algorithm	**σ^2**	0.19	0.34	0.02	1.12
	γ	0.04	0.04	1.40	0.05
	Cost:	0.04	0.03	0.03	0.03
	Time:	0.63	0.66	0.66	0.68

Tab. 4: Automatic parameter tuning comparison with simplex method and grid search method

Both algorithms show quite similar accuracy. However, the simplex algorithm is considerably faster.

1.3.4. Using ROC curves

- ROC (Receiver Operating Characteristic) curves are a useful classification measures because they have some advantages over accuracy rates. for example we can have a test set of size 20 and only 2 of them belong to a particular class. When our model says none of the belong to the class we would have 90% “success”. But, it may basically be giving the same negative output whatever the input is. Also, for medical cases we have to think broadly. We can wrongly classify a person to be a patient but can never miss a patient. This is related with specificity concept. We also want to increase sensitivity. The concepts are defined as below:

$$Sensitivity = \frac{True\ Positives}{True\ Positives + False\ Negatives},$$

$$Specificity = \frac{True\ Negatives}{True\ Negatives + False\ Positives}$$

One would like increase both but there is a trade-off between them. In ROC curves there is a curve which indicates the sensitivity and also 1-specificity. The are under the curve indicates how successful our model is. We also want to test the model on the test set because on the

train set ROC curve will be a bit more stable telling us less. Because there is an accepted truth about the distribution and we just show a trade-off. However, on test set we can see the actual performance with some dramatic changes and decide where to stop on the curve.

- The ROC curves on the test set of *iris.mat* showed success. It has only 20 elements and it is comparatively easier to have 100% accuracy on this small dataset. You can see the best ROC curve accuracy and a bad model in Fig.7.

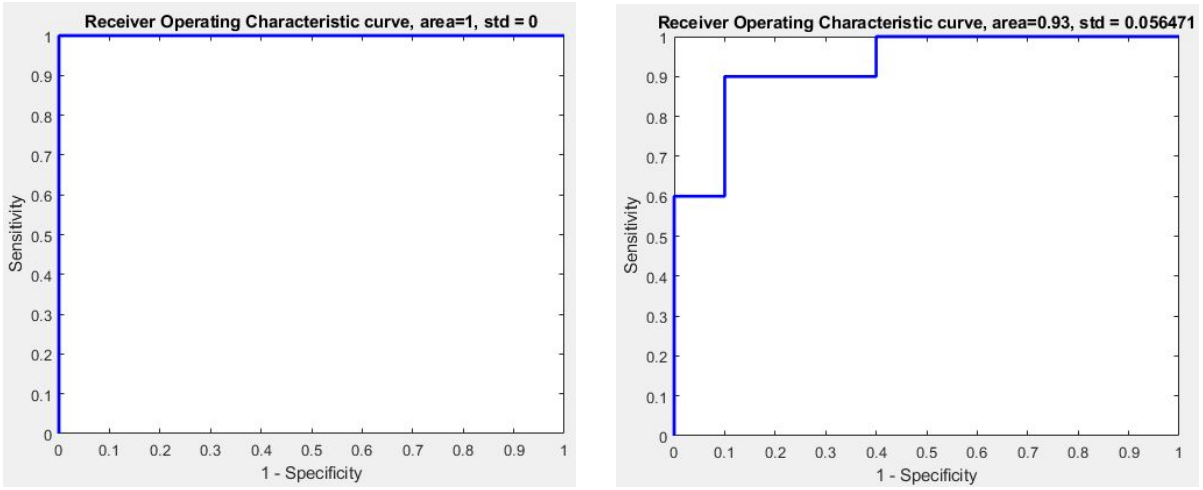


Fig. 7: ROC curves of **a)** γ = 10^6, σ^2 = 0.1 and **b)** γ = 1000, σ^2 = 0.1

1.3.5. Bayesian framework

- The colors scales from blue to pink. Pink color indicates the areas in which the probability of being a member of positive class is high (according to the saturation of pink).

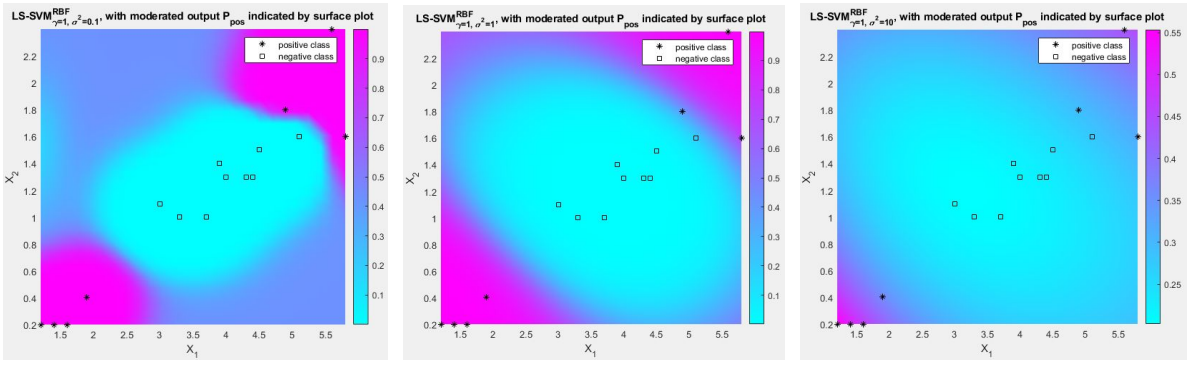
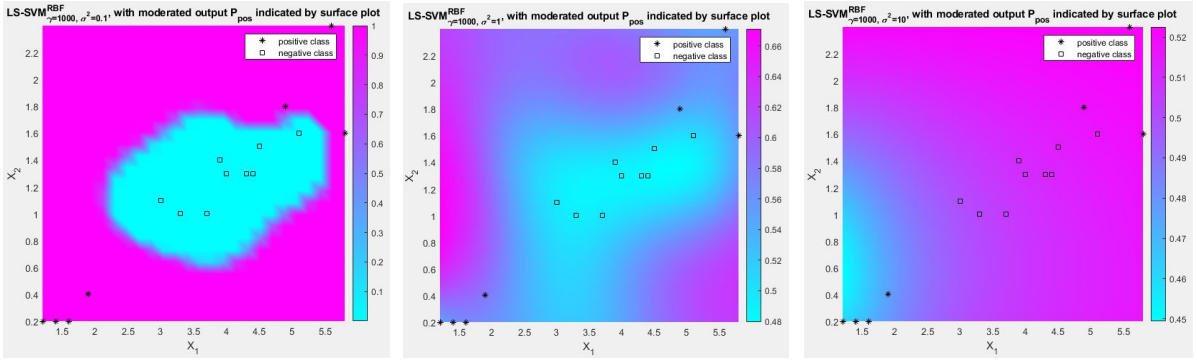
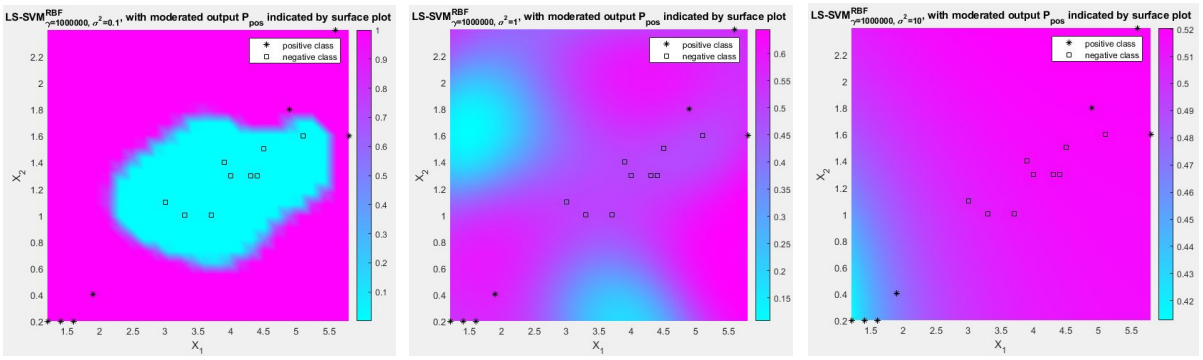


Fig. 8: ROC curves of **a)** γ = 1, σ^2 = 0.1 **b)** γ = 1, σ^2 = 1 **c)** γ = 1, σ^2 = 10



d) γ = 1000, σ^2 = 0.1 **e)** γ = 1000, σ^2 = 1 **f)** γ = 1000, σ^2 = 10



g) γ = 10^6, σ^2 = 0.1 **h)** γ = 10^6, σ^2 = 1 **i)** γ = 10^6, σ^2 = 10

- In Fig. 8, models with different γ and σ^2 values are plotted. Here, we can observe the smoothing effect of higher σ^2. γ increases the nonlinearity.

2. Homework

2.1. The ripley dataset

- This dataset looked like a reasonable candidate for a linear kernel from Fig. 9.

It has 250 training and 1000 test values (half of them are positive for both sets). Because, it looks like it can be divided into two regions from the image of the distribution. Polynomial kernel with more than 3 degrees could also be a good candidate to handle the polynomial twist through the center of the image. However, I needed to analyse thoroughly before proceeding. Therefore, I executed a *leaveoneout()* command for the three kernel types. I didn't use robust methods because there aren't many outliers in the figure. The costs are given in Tab.5. I favored RBF kernel.

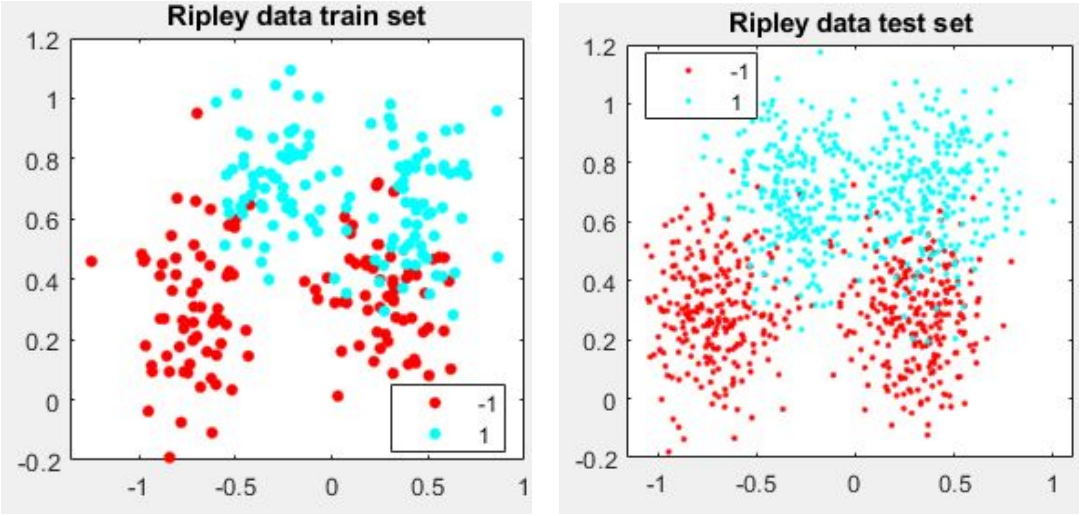


Fig. 9: Ripley dataset **a)** training set **b)** test set

	Linear kernel	Polynomial kernel	RBF kernel
Mean Error	0.140	0.128	0.112

Tab. 5: *leaveonout()* error comparison for three kernel types

- I showed test set results of all types in Fig. 10 and Tab. 6. ROC of the models are given. ROC's look quite similar. The highest area (0.95661) is given with RBF kernel.

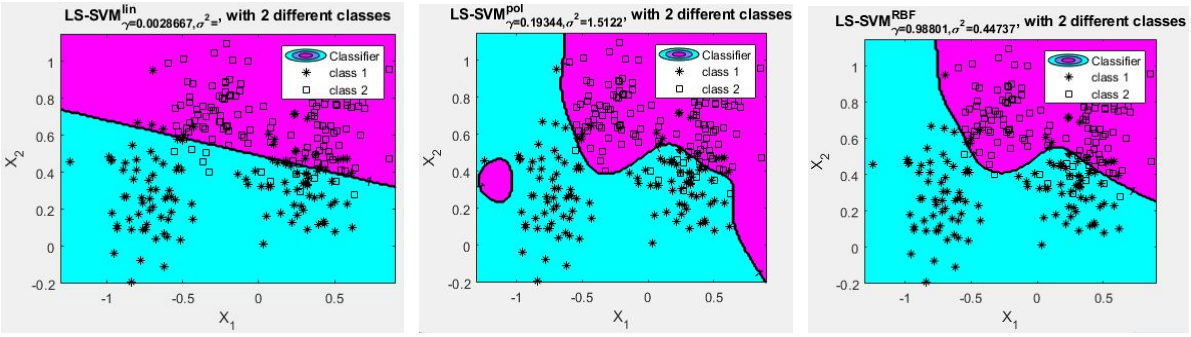


Fig. 10: Simulations of **a)** linear kernel (γ = 0.0029) **b)** polynomial kernel (γ = 0.19, t = 1.51, degree = 7) **c)** RBF kernel (γ = 0.99, σ^2 = 0.45)

	Linear kernel	**Polynomial kernel**	**RBF kernel**
Accuracy	0.894	0.903	0.908

Tab. 6: Accuracy of test set predictions for three kernel types

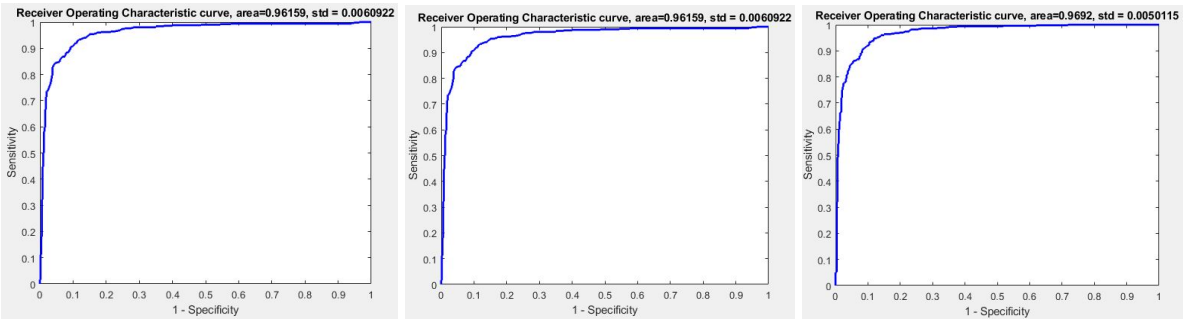


Fig. 11: ROC of **a)** linear kernel **b)** polynomial kernel **c)** RBF kernel

- As we suspected, polynomial kernel with 7 degree also performed very well. Overall, 91% classification accuracy is not a great result though. Looking at the graph, we can see that positive and negative datasets are somewhat mixed and separating them very successfully (98% accuracy or more) doesn't seem as an achievable target with the kernel types of the toolbox. We may consider implementing another kernel or better find a dataset with more features.

2.2. The Wisconsin breast cancer dataset

- The dataset have 400 training (250 benign and 150 malignant) and 169 test set (107 benign and 62 malignant) values with dimensionality 30. The largest 4 PCA coefficients of the the train set is [4.1154, 0.0579, 0.0055, 0.0005]. I visualized the sets using t-distributed stochastic neighbor embedding as in Fig. 12. I could observe that the data are separated enough to have a reasonably successful model. Here, not missing any cancer is crucial.
- Tuned models had accuracies of 0.9527, 0.9349, 0.9704 for linear, polynomial and RBF kernels. ROC's are given in Fig. 13. I easily eliminated the polynomial kernel as it has much less area (0.996, 0.954, 0.992 respectively). Between, linear and RBF kernels, I went

for the highest sensitivity when the specificity is perfect; because, we can't miss any cancer possibility. That's why linear kernel should be chosen. It also has a slightly higher area.

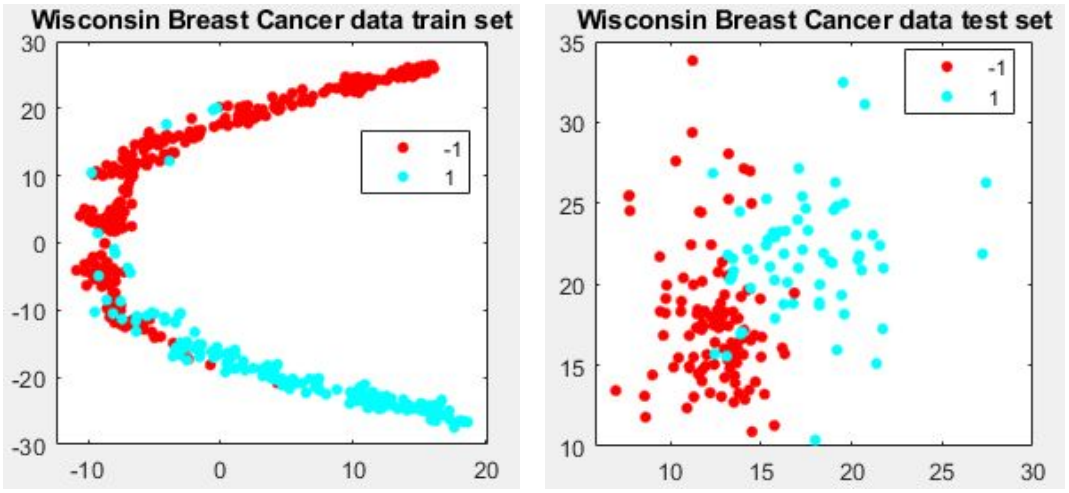


Fig. 12: Wisconsin breast cancer dataset **a)** training set **b)** test set

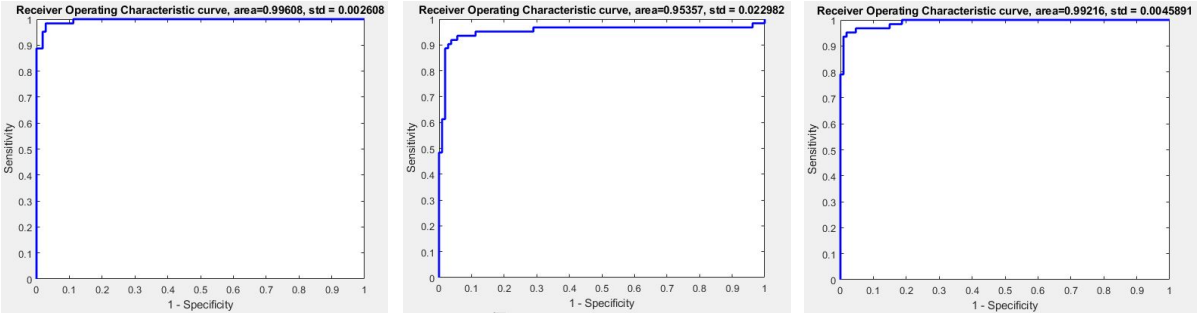


Fig. 13: ROC of **a)** linear kernel **b)** polynomial kernel **c)** RBF kernel

- For this model, I went for the linear kernel and the performance is satisfactory.

2.3. The diabetes dataset

- The dataset has 300 points of which 95 are positive and 205 are negative. Test set has 168 points (62, 106). Points have 8 dimensions. Because this is a medical application I would choose to have higher specificity though it may not be as critical as the cancer data.
- I tuned the models with three kernels. ROC's are given below. For this model, It is hard to decide. For the best specificity, one should go with the linear kernel. It also has the most area together with the RBF kernel (0.845).
- They all performed poorly. More features or a complex kernel are needed.

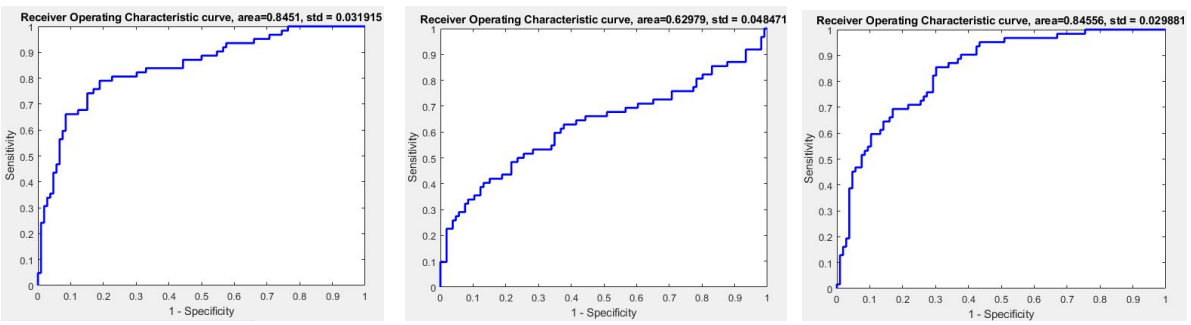


Fig. 14 ROC of **a)** linear kernel **b)** polynomial kernel **c)** RBF kernel

Support Vector Machines - Exercise session 2: Function Estimation and Time Series Prediction

Prepared by Ömer Yılmaz (r0779149)
omer.yilmaz@student.kuleuven.be

1. Exercises

1.1. Support vector machine for function estimation

- e is the bounds of the regression in which the estimation is counted as correct. When e gets bigger, the bounds are more relaxed and more estimations are considered correct. When it gets smaller, more data points are considered wrong because the channel is thinner. When e is small, the optimization problem become harder to solve and more non-zero alpha values are needed to solve it which means more support vectors are used. This means decreased sparsity. In contrast to LS-SVMs ($\alpha = \gamma e_k$), we can use smaller number of support vectors than the dataset size N because of the inequality in the optimization problem. When the e value is bigger then we can use less support vectors meaning sparsity.

Bound is the importance we want to give to how accurate the regressions should be. When it is small, the model doesn't fit sufficiently well to train data and regression isn't accurate enough. Because the importance is given to minimize the weight parameters in the optimization problem instead. When it gets bigger, model fits and when it is too big, the model is overfit showing high variance and low bias with decreased generalization capabilities (likely to fail more on the test data compared to optimum bound value case).

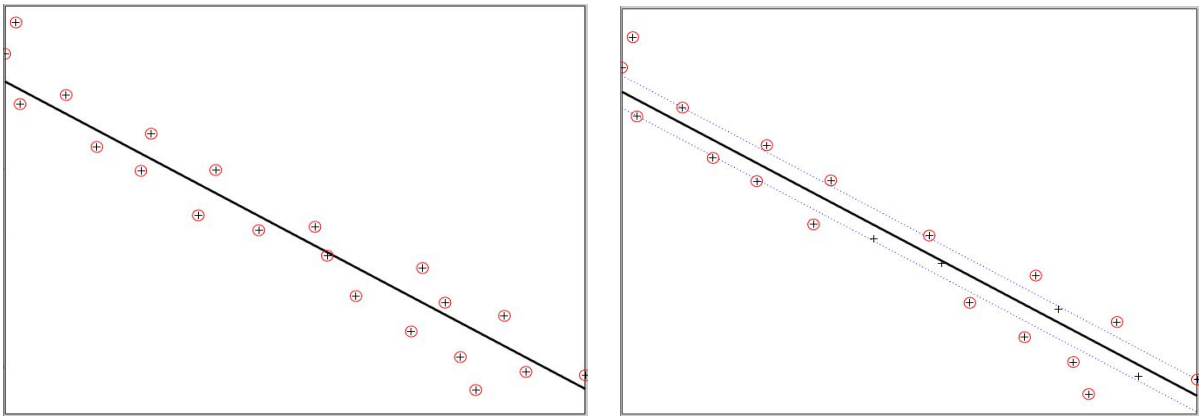
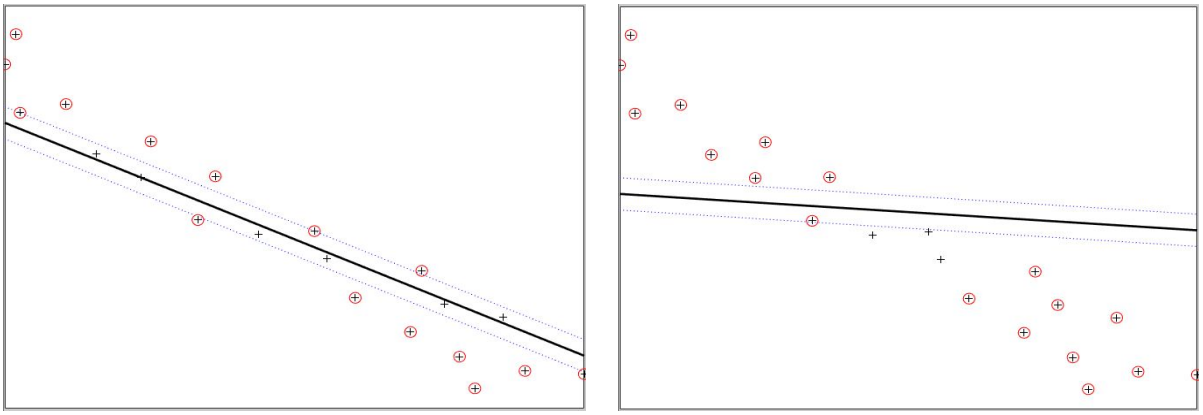
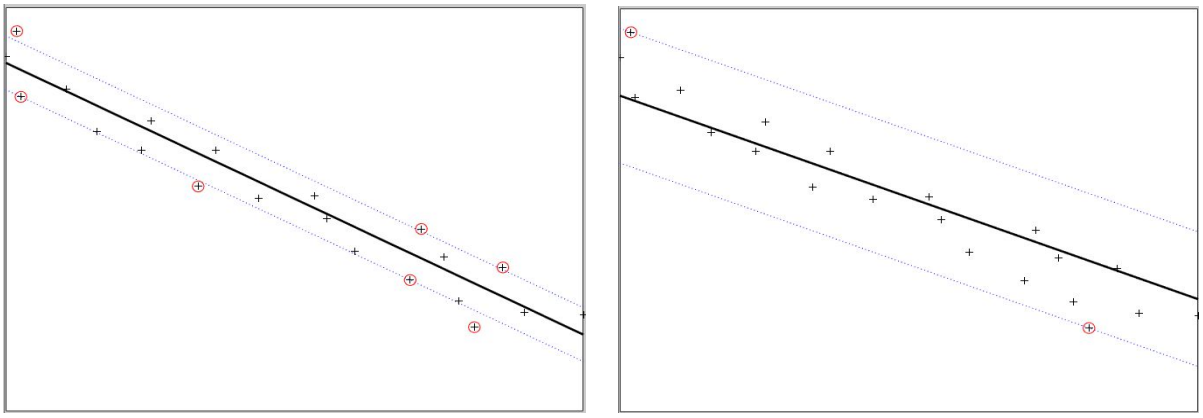


Fig. 1: a) bound = inf, e = 0 **b)** bound = 10, e = 0.1



c) bound = 0.1 , e = 0.1 **d)** bound = 0.01, e = 0.1



e) bound = 10 , e = 0.2 **f)** bound = 10, e = 0.5

In Fig. 1 b) c) and d) we see the effect of decreased bound for e = 0.1. In Fig. 1 b) e) f) we see the effect of increased e for bound = 10.

- For more challenging datasets, some other kernels may be more appropriate (all use bound = 100 , e = 0.01). For example, below in Fig. 2 I have a dataset which has a cubic polynomial shape. Here in Fig. 2 e), we can observe that linear kernel does a very poor job at regression. Because the linear kernel is supposed to regress line shaped datasets only. This compels us to use more complex kernels. Here, not surprisingly a cubic kernel (Fig. 2 f)) and also a Gaussian kernel (Fig. 2 a)) gave the best results in terms of sparsity and accuracy.

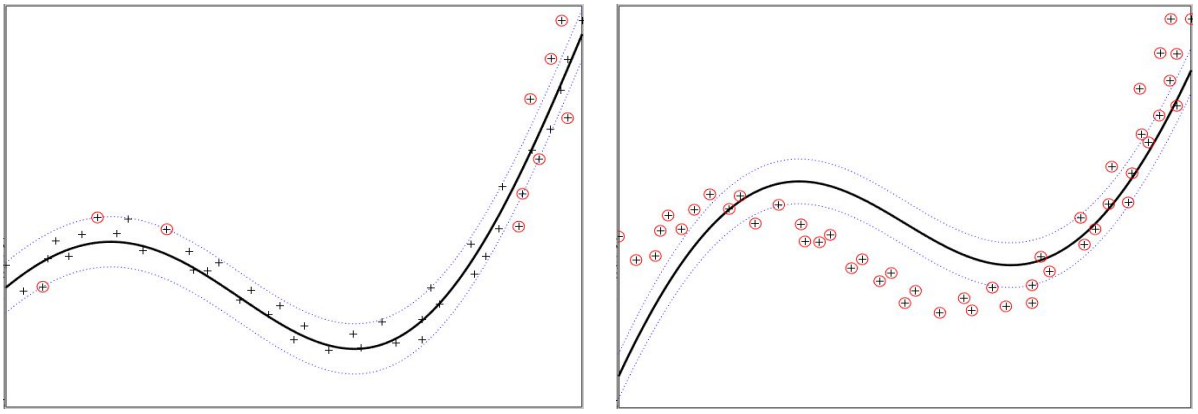
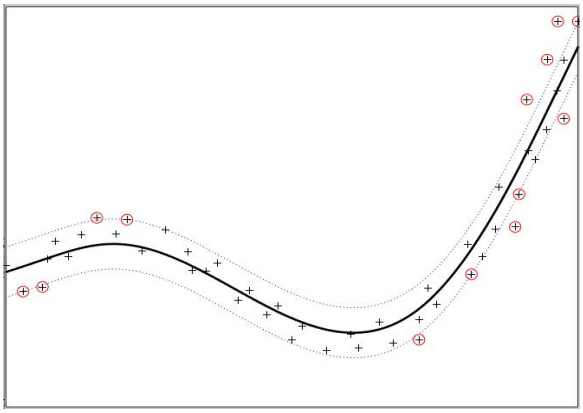
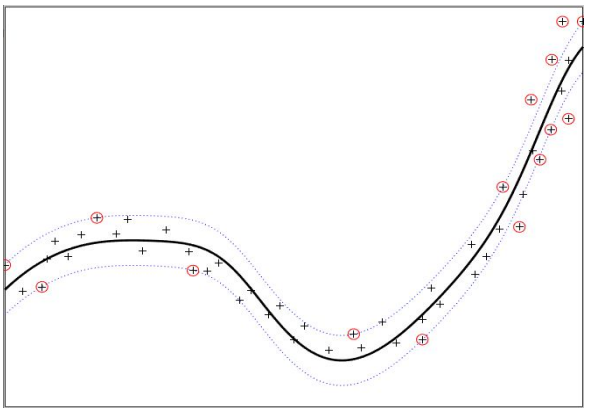
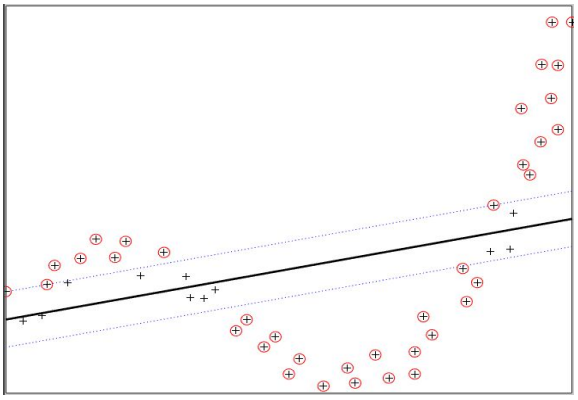
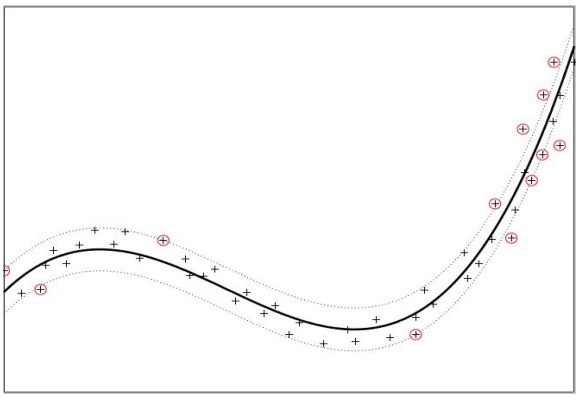


Fig. 2: a) Gaussian kernel **b)** multilayer perceptron

c) linear spline **d)** linear bspline

e) linear kernel **f)** polynomial degree 3 kernel

- The minimization problem has a least squares term and a quadratic term (weight parameters). So, this is not a least squares fit. However, when the bound goes to infinity, the problem converges to a least squares fit.

1.2. A simple example: the sinc function

1.2.1. Regression of the sinc function

- γ value is a regularization parameter. When it is too small, the optimization problem will not focus on correct predictions. This indicates high bias. When γ is too big; however, the optimization problem focus on giving more accurate estimations which results in overfitting to the train values and high variance.

In general, σ^2 determines the spread of a Gaussian function. Here, small σ^2 means that in order for the support vector to be more dominant, the data point to be estimated should be close to the data point of the support vector. This indicates high variance and overfitting. When σ^2 is too big, the effect of the support vector vary too smoothly and an underfit is observed.

Different cases for γ and σ^2 values can be observed in Fig 3. From left to right, we see increasing values of σ^2 {0.01, 1, 100}. From top to bottom, we see increasing values of γ {10, 1000, 10^6}. Clearly the best fit is $\sigma^2 = 1$, $\gamma = 10^6$ (Fig. 2 g)). We can observe the same result from the root mean squared errors as shown in Tab. 1.

Error (rmse)	$\sigma^2 = 0.01$	$\sigma^2 = 1$	$\sigma^2 = 100$
$\gamma = 1$	0.1051	0.1750	0.3641
$\gamma = 1000$	0.1094	0.1122	0.3381
$\gamma = 10^6$	0.1116	0.1030	0.3291

Tab. 1: Root mean squared errors for different γ and σ^2 values

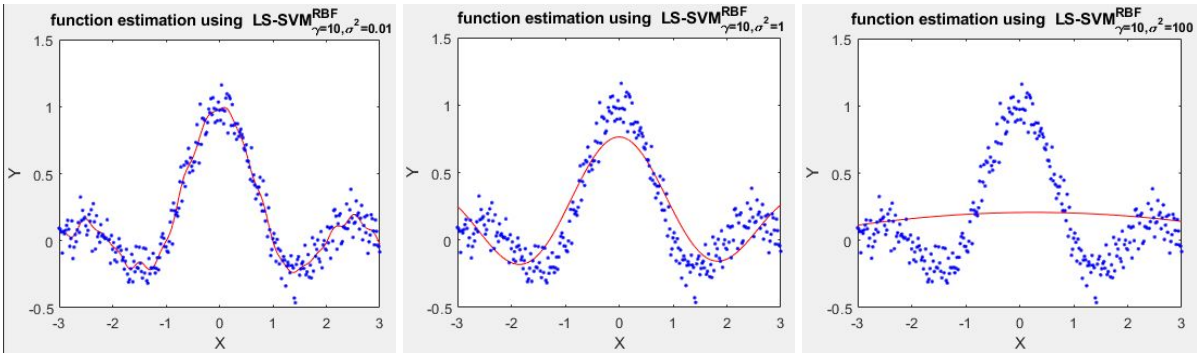
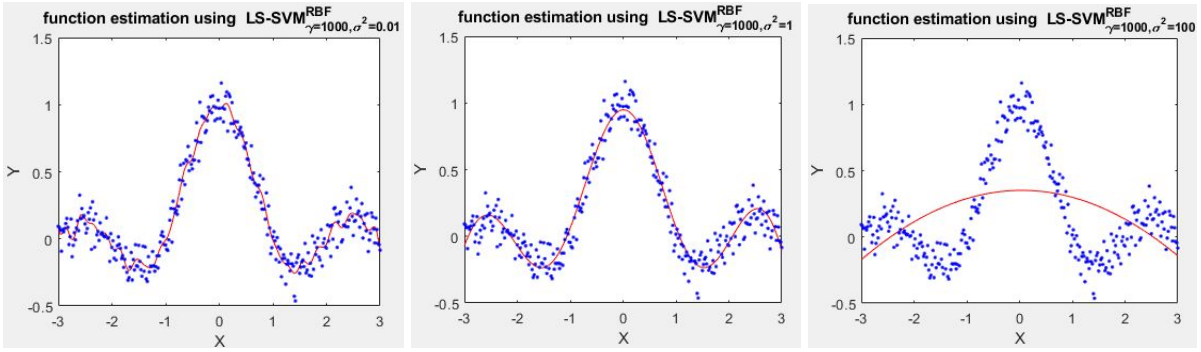
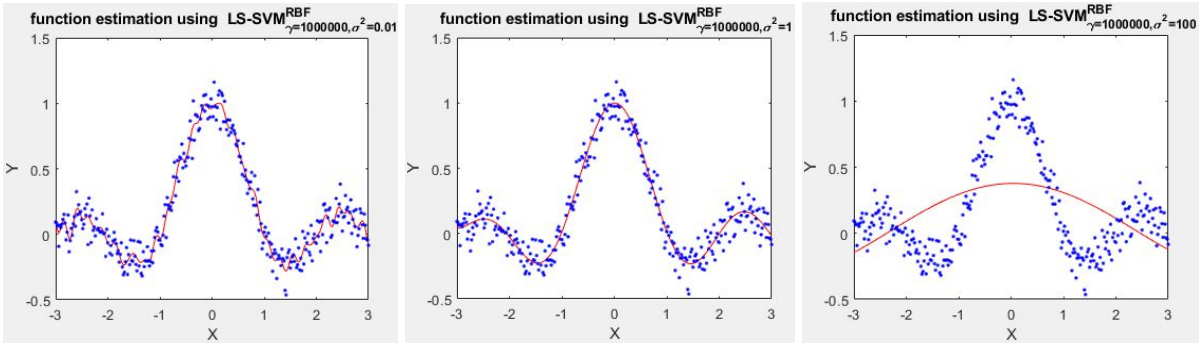


Fig. 3: a) γ = 10, σ^2 = 0.01 **b)** γ = 10, σ^2 = 1 **c)** γ = 10, σ^2 = 100



c) γ = 1000, σ^2 = 0.01 **d)** γ = 1000, σ^2 = 1 **e)** γ = 1000, σ^2 = 100



f) γ = 10^6, σ^2 = 0.01 **g)** γ = 10^6, σ^2 = 1 **h)** γ = 10^6, σ^2 = 100

- In the case above, σ^2 = 1, γ = 10^6 was the best choice. However, in general with options not limited to 9 different cases, one may find different nice fits. There are hyperparameters one decides and because of that in which sense the optimization is meant is not clear. If it is decreasing the cost alone, one may speculate there is indeed one minimum mean cost regardless if we can actually find it or not.
- I ran 10 times each method and took average of them. First of all, 10-fold

gridsearch is considerably slower, 1.00 s compared to 0.65 s on my machine. However, mean costs were very similar 0.0103 (gridsearch) compared to 0.0104 (simplex) also with similar variance. mean σ^2 values were also similar though the variance for σ^2 is greater for simplex method. γ values showed extremely big variance difference (1.16×10^6 for simplex, 74 for gridsearch) with mean values of 371.6 for simplex and 24.0 for gridsearch.
My conclusion is that, the costs are very similar for both methods. One should go with the faster simplex method because of computational power considerations.

1.2.2. Application of the Bayesian framework

- We can use three level Bayesian inference to tune the parameters of the model. In the first level, we tune α and b; in the second and in the third the hyperparameter σ. We try to maximize the likelihood of outputting correct values searching for the right set of parameters. *bay_optimize()* function of the toolbox handles these and *bay_lssvm()* actually computes the posterior costs. To do this, we first provide a prior and infer later. After applying all three levels of Bayesian inference, the regression output of the model and the 5% error band is shown in Fig 4.

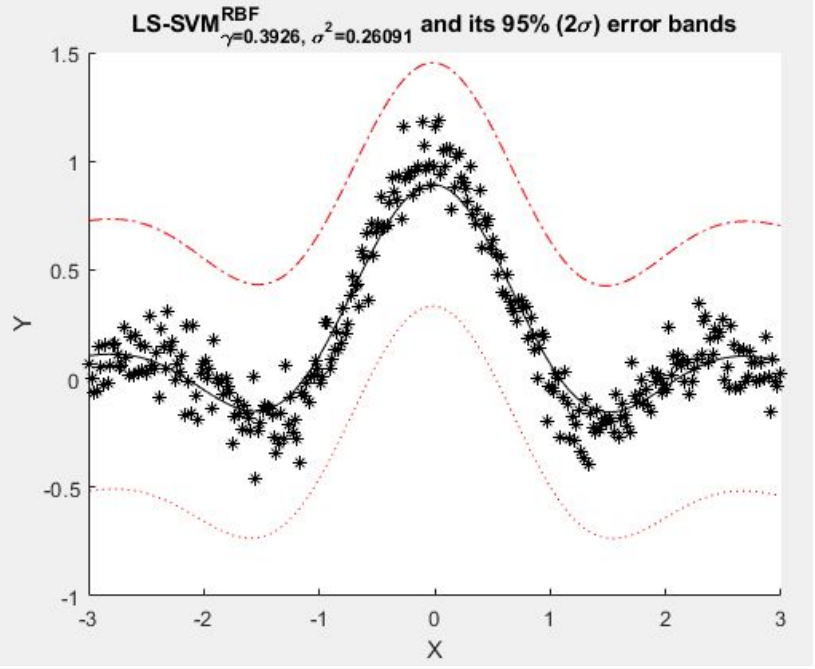


Fig. 4: LS-SVM output on the test set with 2 σ error bands

Is this really the optimal? I searched for different values after the first level of inference. The is indeed the optimal one (Fig. 5 a)). However, after updating γ I searched for different σ values and the one predicted is a local minimum as in Fig. 5 b).

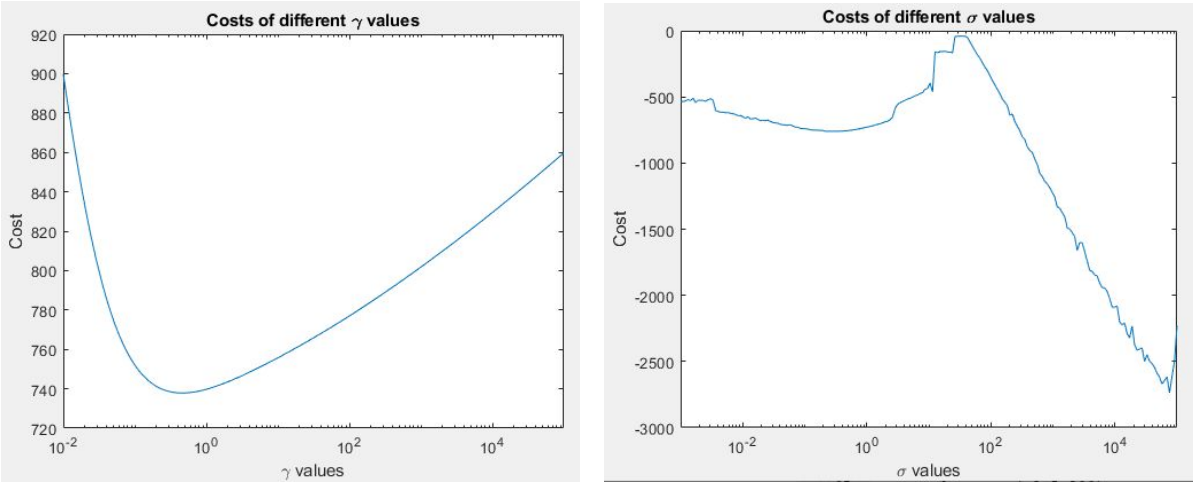


Fig. 5: a) Costs of different γ values, $\sigma^2 = 0.4$ **b)** costs of different σ^2 values, $\gamma = 0.3926$ (found)

1.3. Automatic Relevance Determination

- In this case, we can actually see from the output equation that only the first dimension is relevant for the output. We can observe this fact using automatic relevance determination based on Bayesian inference. In the third (hyperparameters) level of the Bayesian inference, we can remove the irrelevant units from the diagonal matrix S of the kernel equation for RBF shown below (we can apply ARD only with RBF kernel).

$$K(x, z) = e^{-(x-z)^T S^{-1}(x-z)}$$

For relevancy, of course, greater numbers of $1/s_i^2$ is required. Before doing this we need to standardize the data and chose an equal value of s as a prior. Using the *bay_optimize()* function of the toolbox, I was able to show only the first dimension is relevant for our dataset. Though the function sometimes show another dimension to be relevant as well (first dimension is always ranked first).

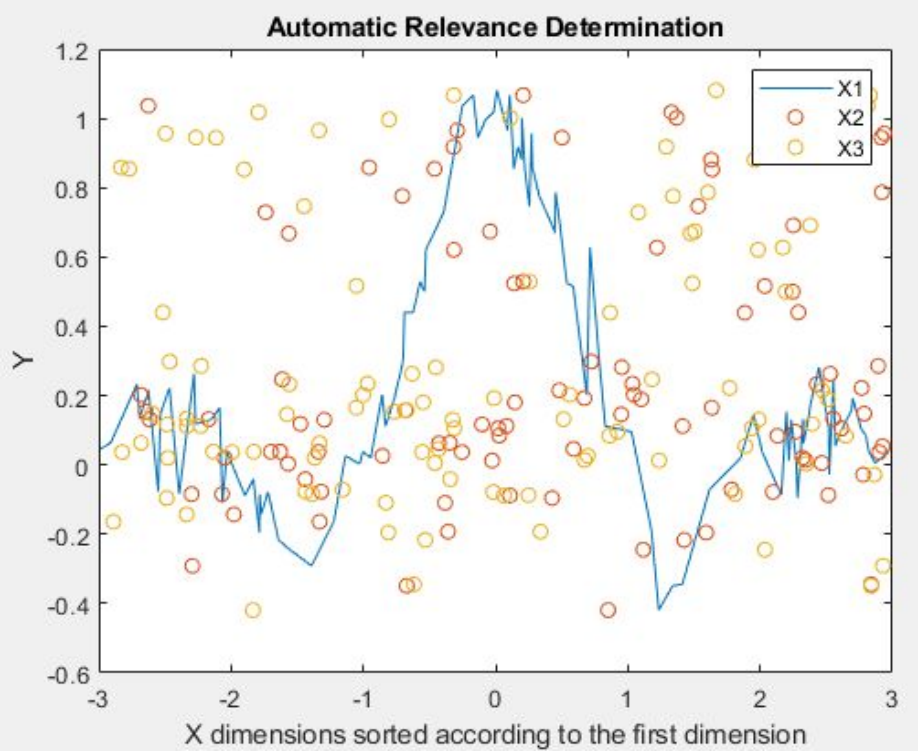


Fig. 6: Relevant dimension and irrelevant dimensions of X

- Instead of ARD mentioned above, we can actually remove the dimensions and observe the costs on validation set for each case. To carry out this we can manually split train and validation sets and observe errors or we can run a cross validation on the training set. Then we choose the lowest cost case which might be a removed set resulting in relevance determination similar to the above method. Irrelevant dimensions are easily checked in Tab. 2. Their addition increased the error.

Dimensions	1	2	3	{1,2}	{2,3}	{1,3}	{1,2,3}
Cost	0.0274	0.1524	0.1515	0.0559	0.1655	0.0557	0.1036

Tab. 2: Errors (mse) with different dimensions of the input dataset

1.4. Robust regression

- The graph of the non-robust version is diverged towards the outliers as shown in Fig. 7. According to the error, this divergence increased or decreased.

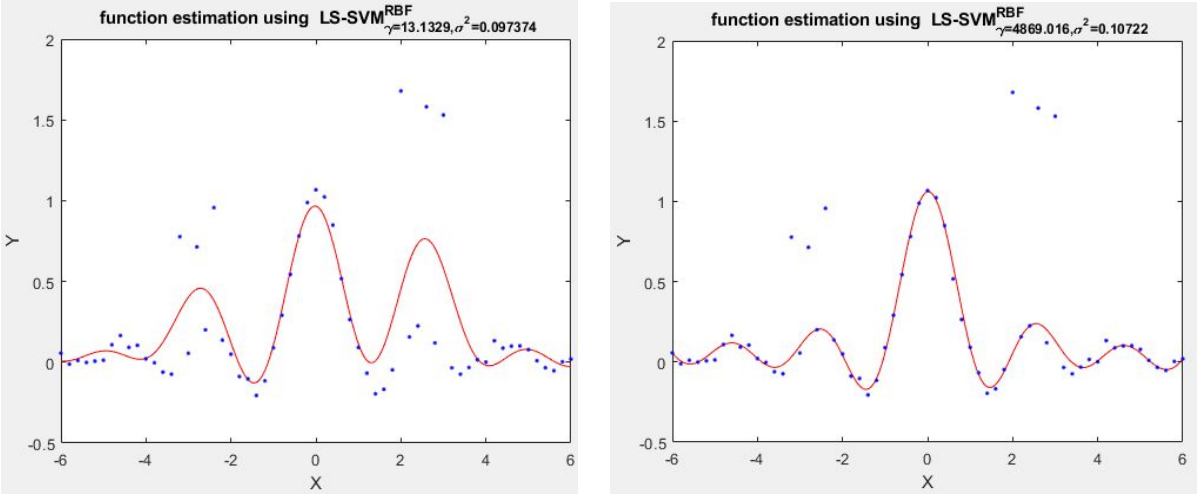
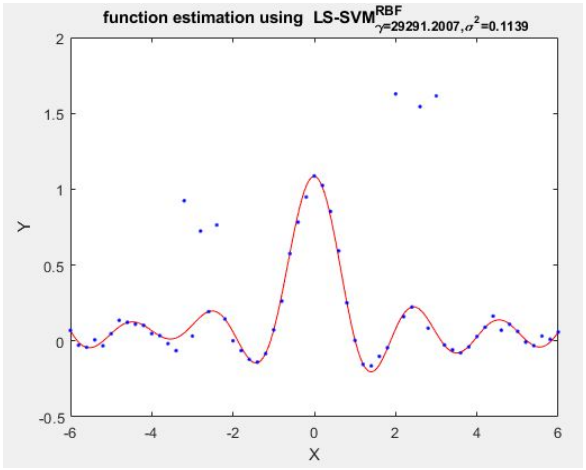


Fig. 7: a) Non-robust LSSVM performance compared to **b)** the robust version

- Mean squared error is susceptible to outlier points; because, the error is squared and the total loss is greatly influenced by a single point. That's why we use mean absolute error to decrease the effect of outliers on the total error.
- In order to address these dramatical effect of few outlier points, we can apply weighting to the losses. Three methods offered in the toolbox can be observed in Fig. 8. Clearly the results are much better compared to the non-robust model.



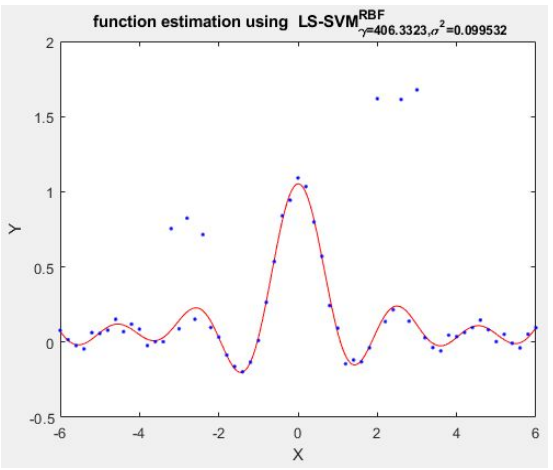
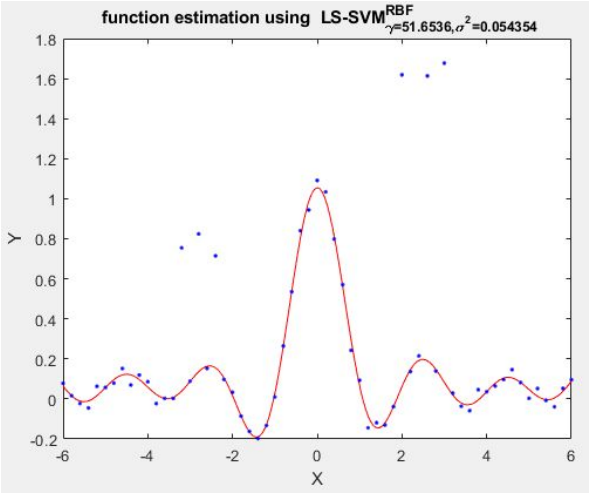


Fig. 8: a) Whampel weighting function **b)** Wlogistic weighting function



c) Wmyriad weighting function

2. Homework

2.1. Introduction: time series prediction

- In order to shape the data into the correct regression format with given lag, we use *windowize()* function of the toolbox.

2.2. Logmap dataset

- As I already learned, for hyperparameter tuning one can use *tunelssvm()* which utilizes *crossvalidate()*, *leaveoneout()* or *rsplitvalidate()* functions to find the loss of training set with a validation method. *tunelssvm()* provides us the hyperparameters like γ and σ^2 for RBF or degree for the polynomial kernel. After tuning hyperparameters, we can observe the performance of the SVM model on a test set. For this, we train the model with the hyperparameters found and get α and bias values. We have, now, the complete model and can predict the test set values using *predict()* function. We can then use a proper loss function comparing the prediction results and test results. For classification, binary cross entropy is a good choice. For regression, we can use mean squared error. If there are some outliers, these can be emphasized with a squared error loss. In order to address this, we can use robust functions given in the exercises with mean absolute error loss and some statistical methods which suppresses the effect of outlier points.

I kept the same procedure here, only now I have an *order* parameter to tune. To find the optimum *order* value, I used a for loop as given below. By the way, I kept this procedure for 5 times (with the inner for loop) for a given hyperparameter set. I could have used a mean of those 5 iterations. However, finding the absolute minimum error make more sense to find the best performing set. I recorded the optimum parameters that give the least error inside the if clause. Here, I should mention that I used the test set as to find the loss which is not completely fair in a real world application. I just tried to show the capabilities of the time series prediction on this homework. Also, the given code introduces elements from the training set already. Instead of training, predicting and calculating the test set loss; using the cross validation loss for prediction would be much easier codewise anyways. Though that wouldn't result in predictions as successful as the one here which is quite understandable.

- The model performed quite well (Fig. 8 b)) compared to the non optimal version given (Fig. 8 a)).

```
orderOpt = 2;
gamOpt = 10;
sig2Opt = 10;
errMin = inf;

for ord = 2:50
    X = windowize(Z, 1:( ord + 1));
    Y = X(:, end);
    X = X(:, 1: ord );
    Xs = Z(end - ord +1: end , 1);
    nb = 50;

    for i = 1:5;
        [gam, sig2, cost] = tunelssvm({X, Y, 'f', [], [], 'RBF_kernel'},...
            'simplex', 'crossvalidatelssvm', {10, 'mse';});
        [alpha, b] = trainlssvm({X, Y, 'f', gam, sig2});
        prediction = predict({X, Y, 'f', gam, sig2}, Xs, nb);
        err=sqrt(mean((Ztest-prediction).^2))
        if err < errMin
            orderOpt = ord;
            gamOpt = gam
            sig2Opt = sig2
            errMin = err
        end
    end
end
```

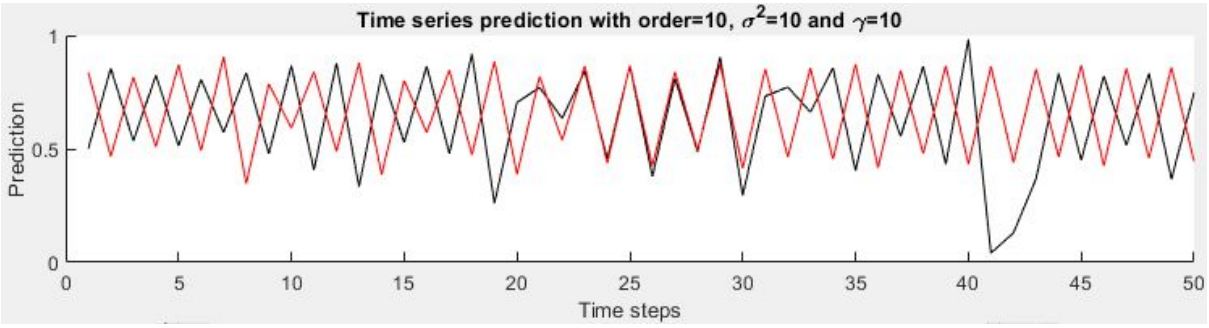
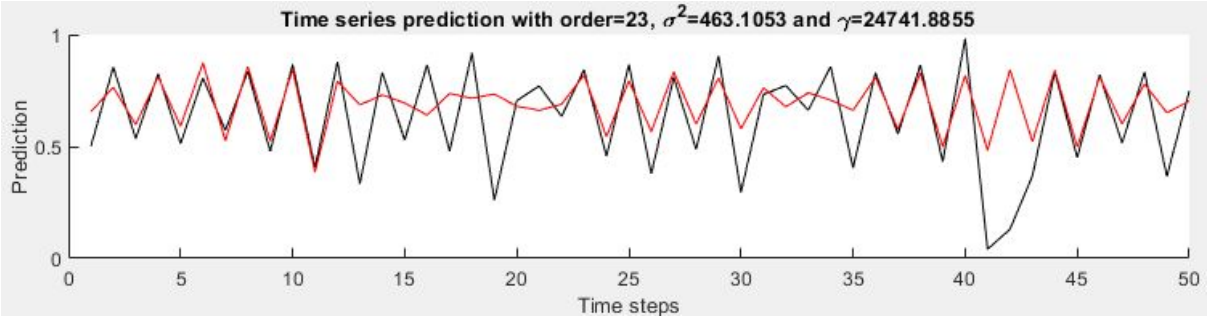


Fig. 9: a) Time series prediction with order = 10, σ^2 = 10 and γ = 10



b) Time series prediction with order = 23, σ^2 = 364.01 and γ = 17357.05

The second model has found not only the best σ^2 and γ values, but also the optimum number of sequence length to look into the past. Although the second model is far from perfect, it still found the correct frequency and sometimes the correct amplitude as well.

2.3. Santa Fe dataset

- 50 is not a bad choice because the test set now has 200 points to look into. We can can run 150 predictions using only the test set values. This can give us a good idea about the error and performance of the hyperparameter. I actually made an exhaustive search from 2 to 100 with a similar code to the one given on the logmap dataset homework above. One thing to note that the code given in the homework integrates the last part of the training set as a history value for prediction. These values are used in the training part though not exactly as the earlier parts of the training sequence. Therefore, when we increase the number of lag values, the error tends to decrease unfairly. That's why i didn't try more than half the size of test dataset. In the above homework, the size of the logmap test data was already very small. Therefore, I made an exception for that (predicted 50 which had lots of values coming from the training set.)
- It would make sense as I already discussed in the logmap dataset homework. The validation set would give us a fair comparison. When we tune according to the test data, we directly take into account the test set's tendencies, break points etc… In a real world application, we wouldn't know about the critical points of future. Therefore, a validation set would give us a blind and fair idea about the performance on the test set. In practice, we can separate the last part of the training set as validation set (with a ratio); then, continue on the test set.
- I again used cross validation in *tunelsssvm()* function and run 5 iterations. Predictions can be seen in Fig. 10 below. The optimum numbers found are *order*: 29, σ^2: 9.64 and γ = 16.09.

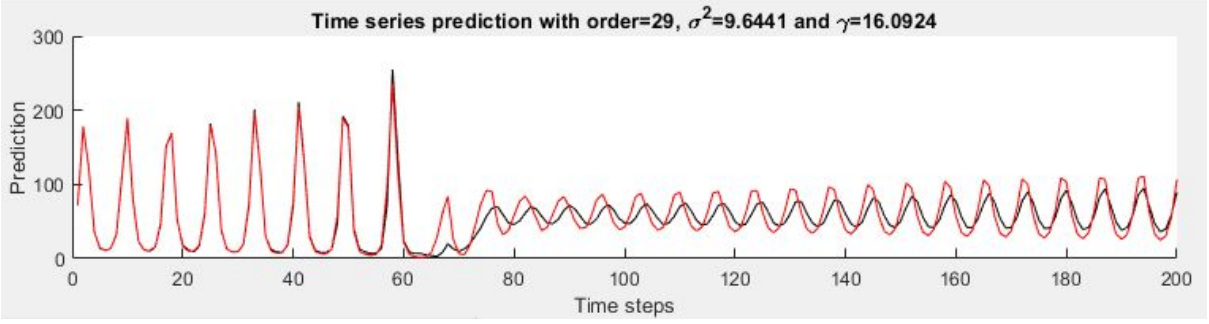


Fig. 10: Time series prediction with order = 29, σ^2 = 9.64 and γ = 16.09

Support Vector Machines - Exercise Session 3: Unsupervised Learning and Large Scale Problems

Prepared by Ömer Yılmaz (r0779149)
omer.yilmaz@student.kuleuven.be

1. Exercises

1.1. Kernel principal component analysis

- Denoising can be done examining the main components of the kernel matrix and reconstructing like an autoencoder. We can do this on the linear kernel, the usual principal component analysis. In that case we do not use all the dimensions of the original points. Otherwise no noise will be cancelled obviously. However, reducing the dimension and reconstructing the dataset help.
- Executing a similar autoencoder method on a nonlinear kernel result in a better performance. In KPCA method we can actually increase the number of components (hidden layer dimension of the autoencoder) as much as the number of data points, N. So, we can reconstruct the image using much more eigenvectors if we have a dataset size more than the number of dimensions which is usually the case (otherwise, as a side note, we still use the smaller number for PCA). Though we shouldn't use a very high number of components (especially N similar to above case) as in Fig. 1 a). Because, we may include the effects of noise instead of showing only the general patterns. On the other hand, we cannot include few components as in Fig. 1 c). That would not reflect the main patterns. Therefore, we need to balance, Fig. 1 b). In addition, nonlinear kernel can actually reveal the nonlinear patterns of the dataset whereas linear kernel can only show linear patterns as can be observed in Fig. 1 d). This is a fundamental advantage for nonlinear kernels.

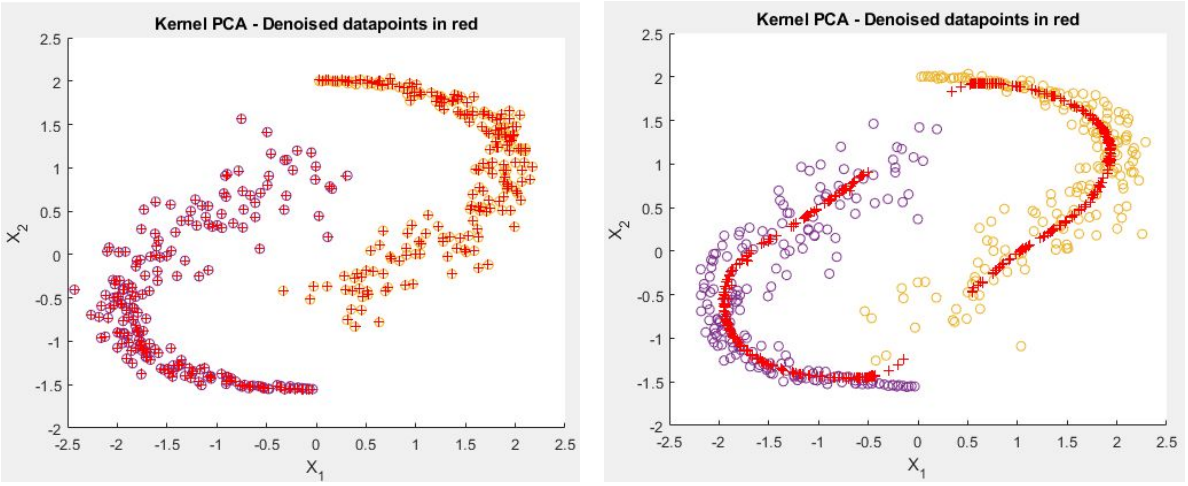
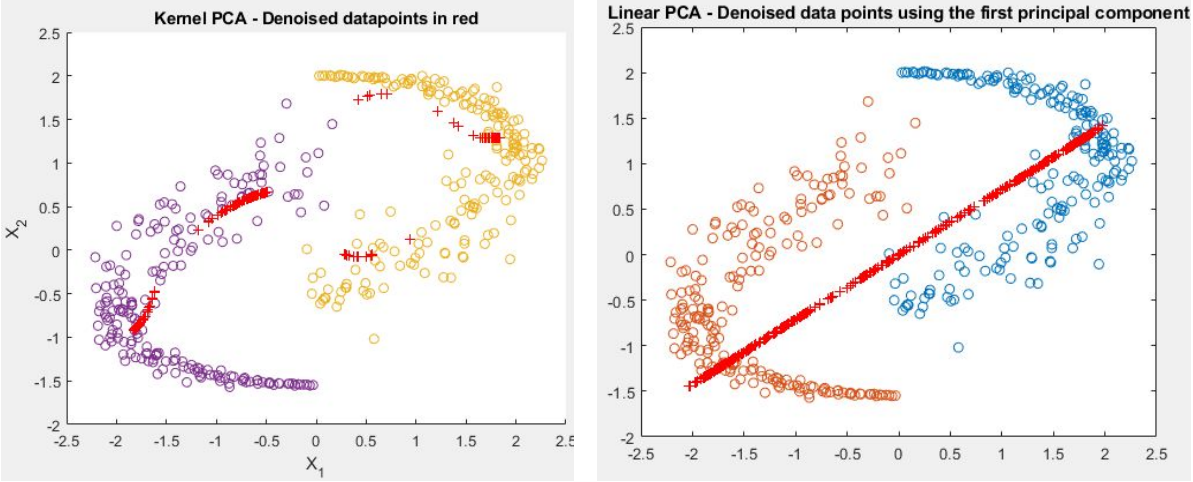


Fig. 1: Nonlinear kernel reconstruction with **a)** 40 components **b)** 6 components



c) 2 components **d)** Linear kernel reconstruction with 1 component

- If we have the noise free version of the dataset we can of course check for the mean squared error between the noiseless version and the denoised version but if we have the noiseless version we wouldn't do this anyways. However, if we don't have a noiseless dataset but we can somehow represent the main patterns mathematically, we can create a noise-free dataset and perform the error check on this. For example, an expert given the noisy data above, could guess a ying yang shape and use the mathematical create the original dataset. How to match the number of points and the right point to check for in the noiseless dataset? For that we can use the closest point available and generate much more points to compensate for the discreteness of the dataset or perhaps use a more complex mathematical approach to find the nearest point of the continuous curve.

1.2. Spectral clustering

- Spectral clustering is a method that tries to group similar points in clusters. Points in the same cluster are alike whereas points on different cluster are dissimilar. To perform a spectral clustering on a dataset of size N we need 3 steps.
    - ❖ We create a similarity (affinity) matrix among points (for example RBF kernel),
    - ❖ We find the Laplacian matrix (first finding the degree matrix similar to a graph but based on the similarity matrix) and compute the first k eigenvectors of it,
    - ❖ Find clusters by an optimization problem (like performing k-means clustering on these eigenvectors which creates k clusters),

Spectral clustering is similar to KPCA upto the point eigenvectors are found.Then, we use weights and cluster the vectors.

- Spectral clustering is used to cluster points in an unsupervised way and assign a label to the cluster while classification focuses on finding the right label with a supervised method. Spectral clustering focuses on features of the data points while classification don't have such a goal though it can utilize features extracted. Spectral clustering works in an automatic way though we still need to decide on hyperparameters (like the k number of clusters above). Classification on the other hand requires a train and a test set possibly supported with a validation set as well for early and fair performance check all given with a label.
- The effect of different σ^2 values on the spectral clustering can be observed in Fig. 2

When σ^2 is too small, all the elements of the centered matrix will approach to 0 except the diagonals showing no similarities. For two vectors to be similar, it would require them to be extremely close. Otherwise, they wouldn't fall into the first standard deviation of the Gaussian function.

While σ^2 is getting bigger, centered matrix starts to create diagonal blocks (shown as whitening effect below) which actually corresponds to the spectral clusters indicating high within-block similarity.

If σ^2 is too big, the spread of the Gaussian increases which results in considering every other point close and classifying them as similar. This is equal to the blocks of the centered matrix to be dissolving and creating one big cluster of points. That wouldn't find the individual clusters that we are after.

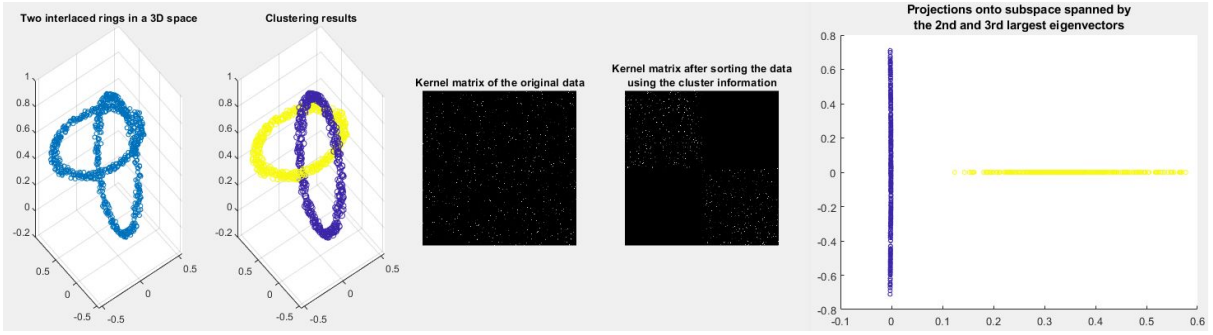
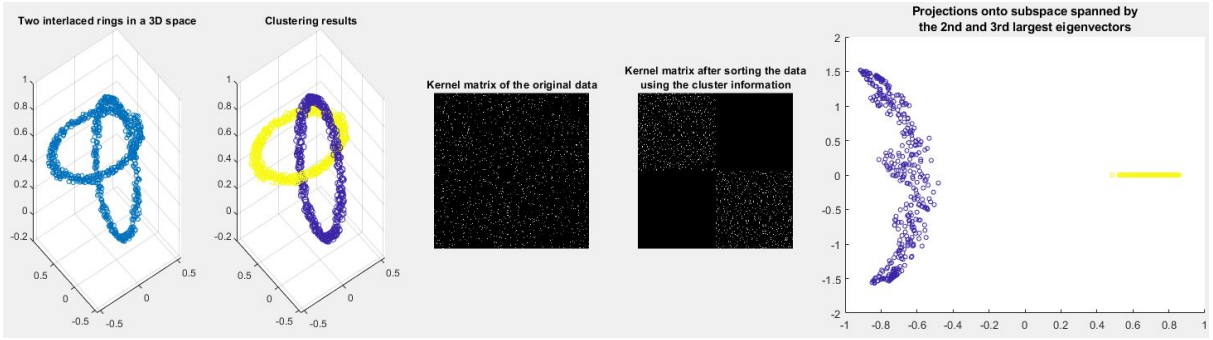
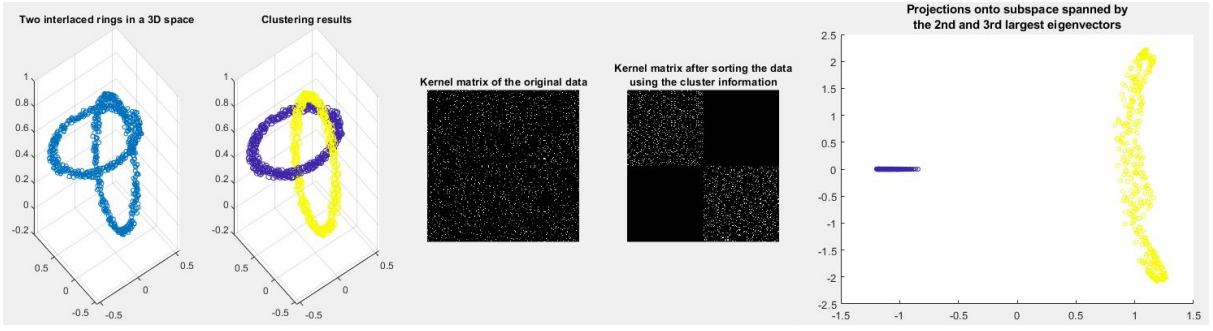


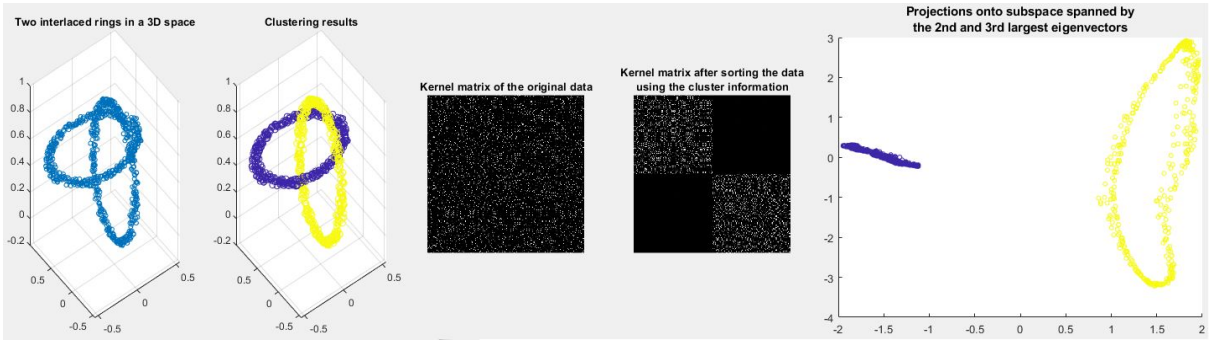
Fig. 2: Spectral clustering with **a)** $\sigma^2 = 0.001$



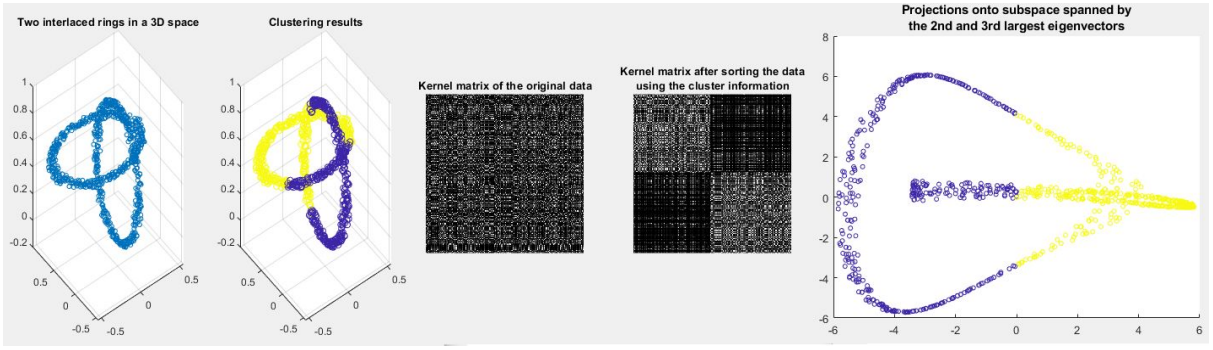
b) $\sigma^2 = 0.005$



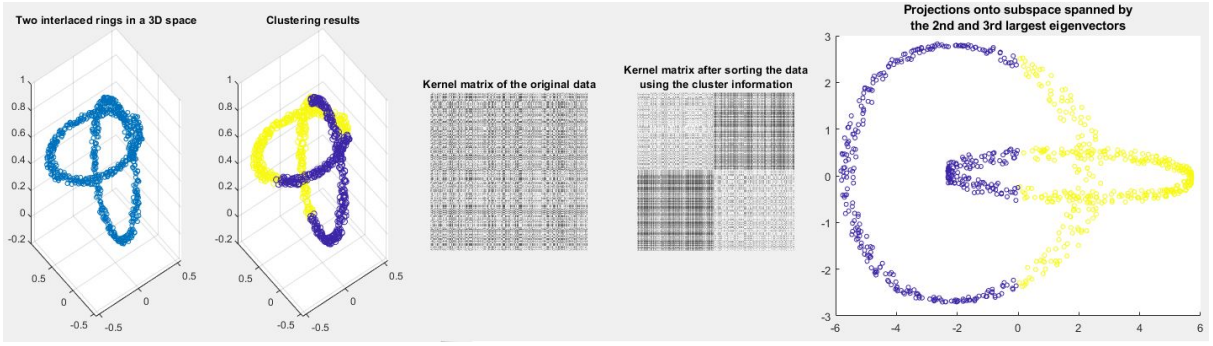
c) $\sigma^2 = 0.01$



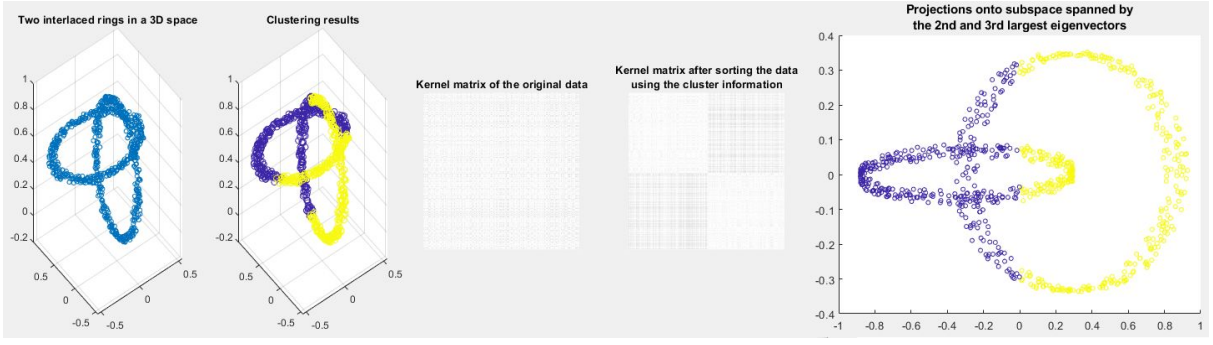
d) $\sigma^2 = 0.02$



e) $\sigma^2 = 0.1$



f) $\sigma^2 = 1$



g) $\sigma^2 = 10$

1.3. Fixed-size LS-SVM

- If the number of data points is much more than the dimension, we typically use primal. For dual, we are not restricted by the dimensionality of the individual data points. It could be even infinite with the use of a right kernel function. That ‘s why when the dimension is larger, we should choose dual. We can deal with large datasets with the fixed-size LS-SVM as well. Nyström method is used to approximate the NxN dataset with MxM subset (M < N) with minimal information loss. This is useful in terms of time and memory

considerations. We use a subset of the whole dataset. This solves computational and memory problems. It basically tries each point one-by-one replacing another and checks for the increased entropy in a subset. The entropy differences derive from the distribution differences between the whole dataset and the sampled subset. And, we try to find the optimum subset which represents the original dataset best. For the entropy check, the quadratic Renyi entropy can be used.

- Effects of different σ^2 values can be observed at Fig. 3. Again, very small σ^2 values results in all zero-like elements (except diagonals) in the kernel matrix because they require almost perfect similarities. Because this is harder to provide, it seems as if the points are really dissimilar and entropy is assumed high very easily. This is misleading. On the other hand, very big σ^2 values are the causes for all elements going to one. Therefore, they cannot differentiate if they are similar to another point or not. Because, the distance doesn't affect the output of the internal Gaussian function enough. Both low and high values don't represent the distributions well. We can observe from the Fig. 3 that small σ^2 values gather around the center and large ones result in spreaded sample points. Best σ^2 then, should show maximum entropy that is not misleading which means increasing σ^2 upto the point the entropy starts to decrease. The σ^2 values for both the datasets shouldn't differ dramatically.

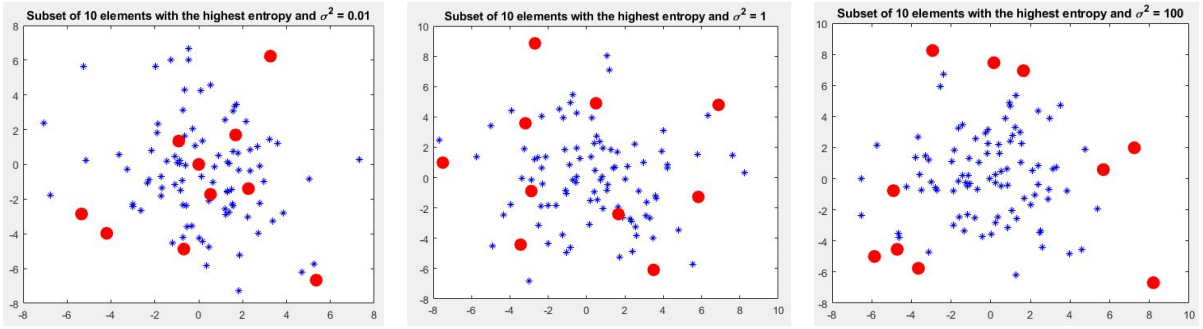


Fig. 3: Fixed size LS-SVM results with **a)** σ^2 = 0.01 **b)** σ^2 = 1 **c)** σ^2 = 100

- Both methods addresses the sparseness problem of the LS-SVM's. We can prune the LS-SVM after having the parameters and fine-tune it. L0-approximation on the other hand utilizes the nonzero elements of the vectors. Its implementation to reduce LS-SVM's has O(N3) complexity per iteration, however. Apart from LS-SVM computation costs typically a low number of iterations (15 - 20) is required[1].

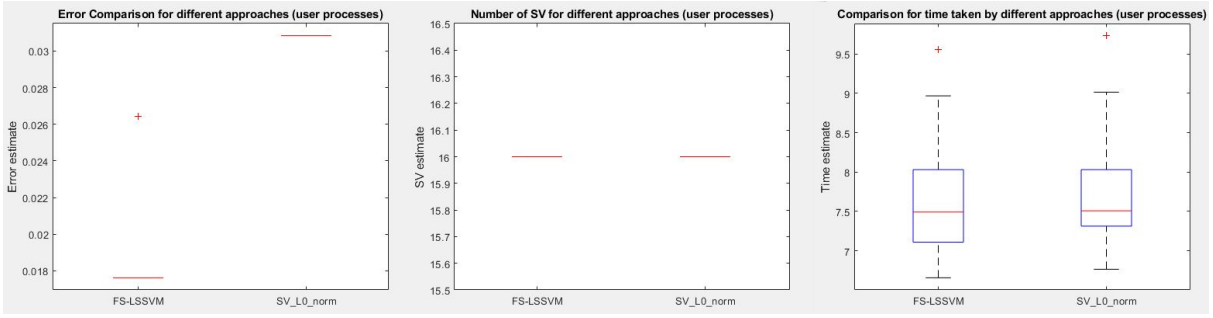


Fig. 4: Comparison of FS-LSSVM and SV L0 norm **a)** error comparison **b)** number of SV **c)** time taken

[1] Lopez, J., De Brabanter, K., Dorronsoro, J. R., & Suykens, J. A. K. (2011, April). Sparse LSSVMs with L0-norm minimization. In *Proceedings of the European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN 2011)* (pp. 189-194).

Error estimate is clearly lower for FS-LSSVM. Numbers of support vectors are the same and the times taken are also very similar (slightly in favor of FS-LSSVM). Therefore, I would suggest using FS-LSSVM. Also, the paper mentioned above suggests several sequential minimal optimization algorithms[2].

2. Homework

2.1. Kernel principal component analysis

- Differences between PCA and KPCA denoising (*noisefactor* = 1.0) can be compared looking at Fig. 5. Clearly, KPCA performs much better as I have analysed on exercises section. Because, KPCA can memorize nonlinear features whereas PCA is only able to extract linear features. Also, I had discussed about the KPCA's high number of eigenvectors which can possibly learn better. We observe that increasing the number of components resulted in better reconstruction. Though from the Fig 5. a), it looks PCA can never reconstruct successfully.

One important thing is that KPCA isn't able to reconstruct 7. The reason is the high amount of noise. Noisy 7's are confused with 2's as they are very similar visually.

- When a large σ^2 value (large *sigmafactor*) is used, it would result in all the images to look similar and enhance the effect of noise because of the reasons we discussed. A small σ^2 on the other hand, wouldn't classify even very similar looking images of the same class. Therefore, we need to choose the σ^2 so that it should differentiate the noises while not being too restrictive. Reconstruction with different σ^2 values can be compared for both methods in Fig. 6.

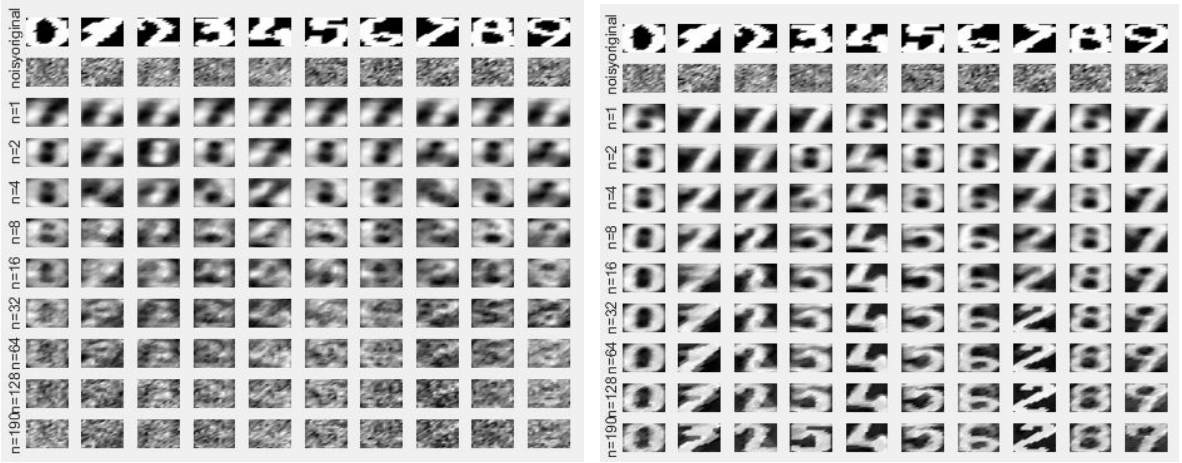


Fig. 5: Comparison of denoising methods with a noise scale 1 **a)** PCA **b)** KPCA

- σ^2 shouldn't be chosen very low or very high due to before-mentioned reasons. The former is too hard for the points to be similar, because it compels the data points to be extremely close. The latter is not able to detect dissimilarities well. As we have analysed earlier, very small σ^2 unfairly evaluates the points as dissimilar with near-zero kernel matrix elements for different points. And the high σ^2, because of its extreme big spread, thinks the points are quite close (the ratio of their distance to the spread of the Gaussian is very small).

[2] López, J., & Suykens, J. A. (2011). First and second order SMO algorithms for LS-SVM classifiers. *Neural Processing Letters*, *33*(1), 31-44.

In addition, the rule of thumb says σ^2 should be equal to the mean of the variances of each dimension multiplied with the dimension and scaled with a constant chosen which lets us tune the kernel similarity.

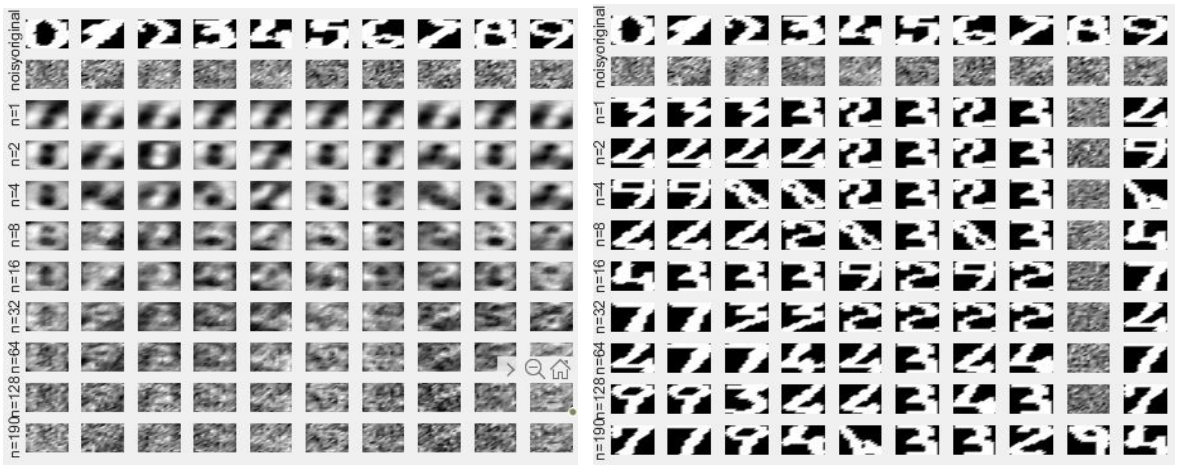
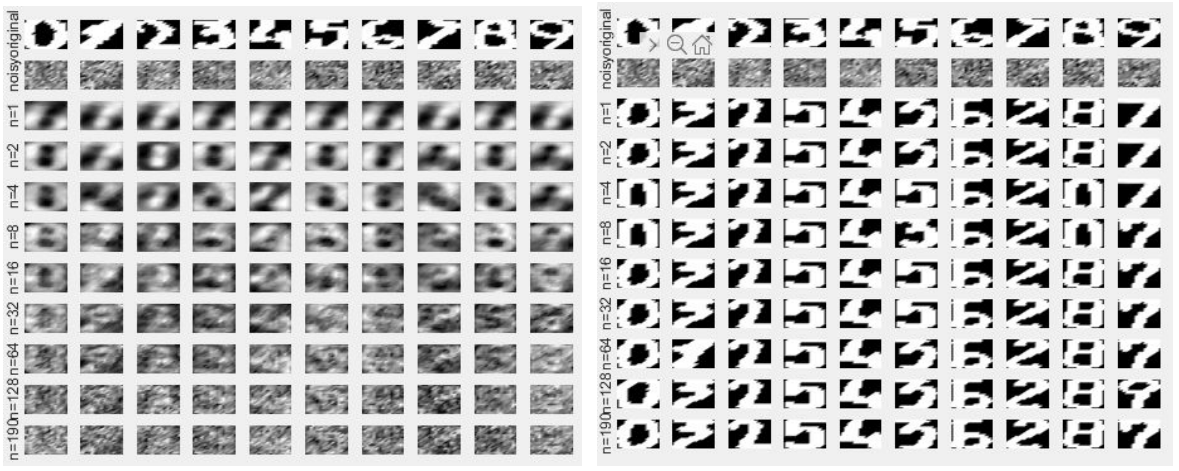
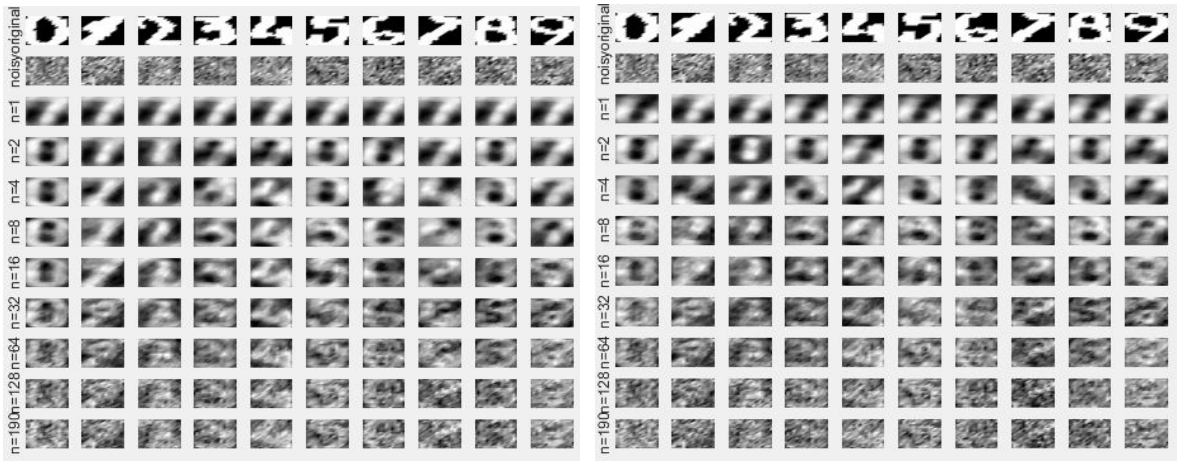


Fig. 6: Comparison of denoising methods **a)** PCA (*sigmafactor* = 0.01) **b)** KPCA (*sigmafactor* = 0.01)



c) PCA (*sigmafactor* = 0.1) **d)** KPCA (*sigmafactor* = 0.1)



e) PCA (*sigmafactor* = 10) **f)** KPCA (*sigmafactor* = 10)

- Optimum value of σ^2 can also be found using the RMSME. It is the RMSE operation

on the mean error of all the pixel estimations for an image reconstruction. We can run for loops to finetune the parameters. We simply search for the least *mean(mean((Xtest1 - Xdt).^2))* and *mean(mean((Xtest1 - Xdt_lin).^2))* values from the given assignment code. Results are given in Tab. 1. We confirmed that losses are compatible with the figures. Also, increasing noise inreases RMSME.

	{noise=0.1, σ^2=1}	**{noise=0.3, σ^2=1}**	**{noise=1, σ^2=0.1}**	**{noise=1, σ^2=1}**	**{noise=1, σ^2=10}**	**{noise=1, σ^2=100}**
RMSME	0.0897	0.0935	0.2091	0.1553	0.8242	1.8573

Tab. 1: RMSME errors of different parameter sets for KPCA

2.2. Fixed-size LS-SVM

2.2.1. Shuttle (statlog) dataset

- Shuttle dataset is a multi-class classification dataset. The dataset includes 58000 data points. Points have dimensionality of 10 of which the first 9 are attributes. The first attribute is the time. Attributes are {Rad Flow, Fpv Close, Fpv Open, High, Bypass, Bpv Close, Bpv Open}. They indicate the physical state of the shuttle. The class label is given in the last column of the dataset. There are 7 distinct classes. The majority of the data (~80%) belongs to the class 1. Therefore, 99 - 99.9% classification success should be targeted. I used 700 points of the whole dataset. In Fig. 6, we can see the dataset with the most variant dimensions (6th and 9th).

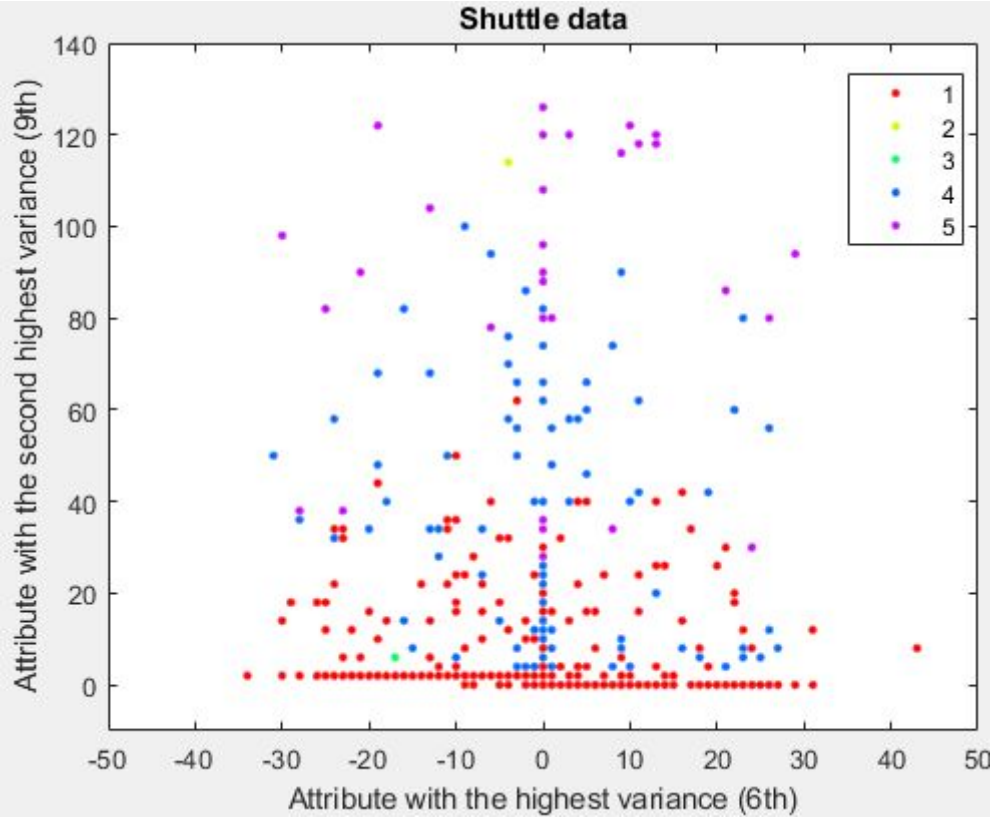


Fig. 6: Distribution of the shuttle dataset according to most variant 2 attributes

- I used the Coupled Simulated Annealing and investigated different number of components with different kernel types.

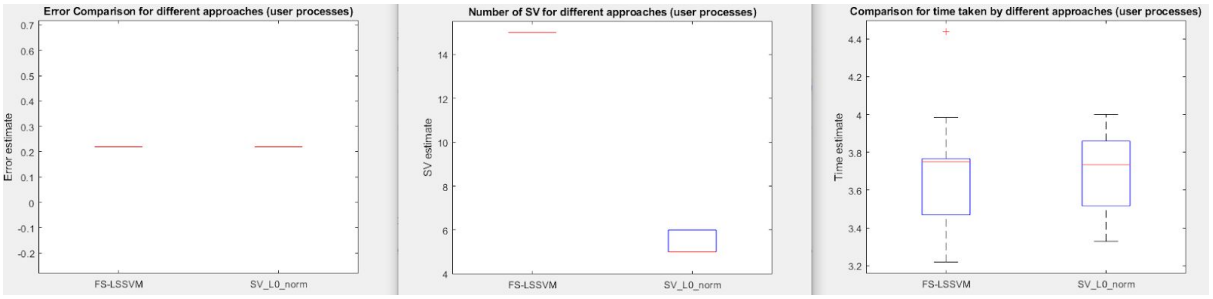
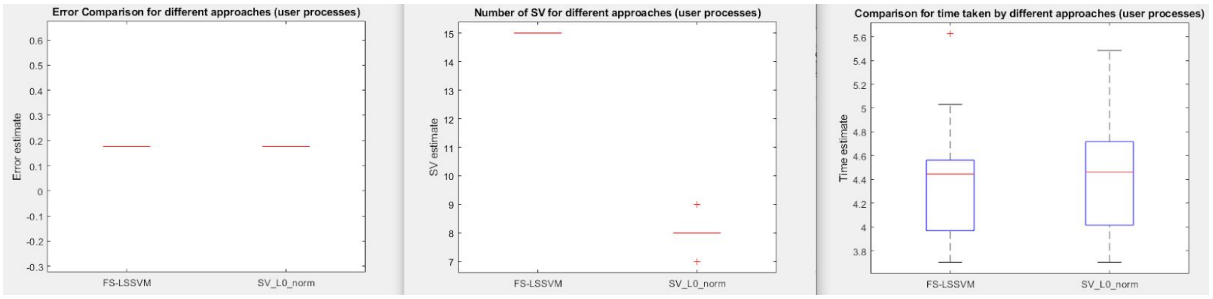
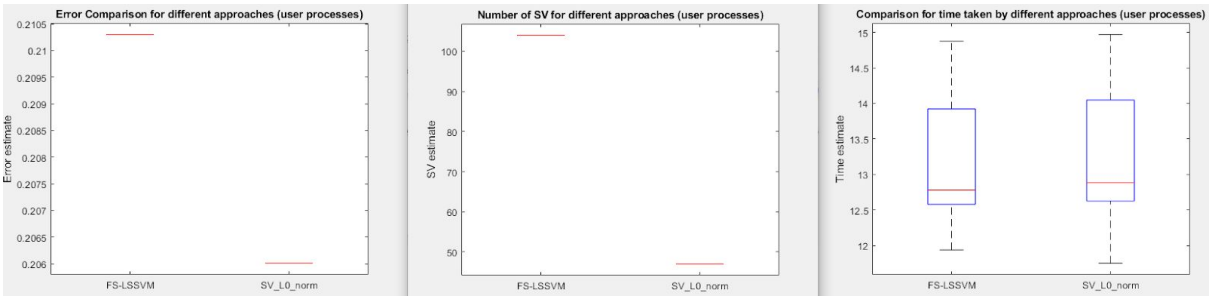


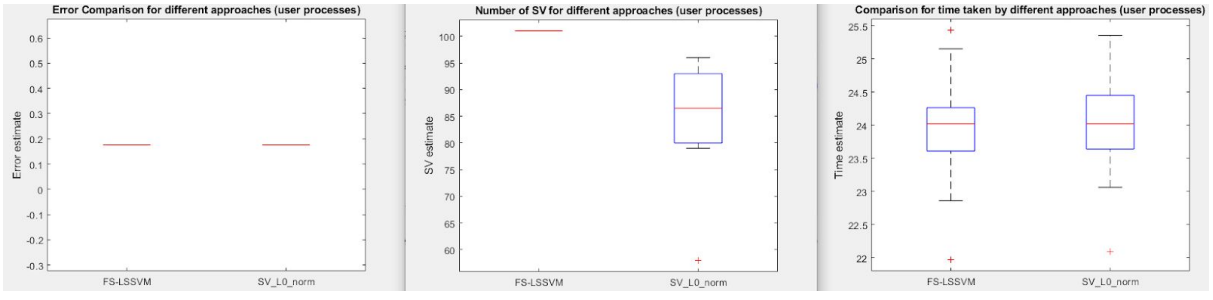
Fig. 7: a) k=4 with linear kernel



b) k=6 with linear kernel



c) k=4 with polynomial kernel



d) k=4 with RBF kernel

I suggest using k=6 with linear kernel simply because it has much better time consumption and still achieves the best error (least) with a very good (small) number of SV's.

2.2.2. California dataset

- California Housing dataset contains 20460 data. It is used to price houses in

California. That's why it is a regression problem. There are 8 attributes {median house value, median income, housing median age, total rooms, total bedrooms, population, households, latitude, longitude} in this order column-wise. The last column includes prices of the houses.

- I again investigated using CSA as in Fig. 8. This time I tried polynomial and RBF kernels with k=6 and k=8.

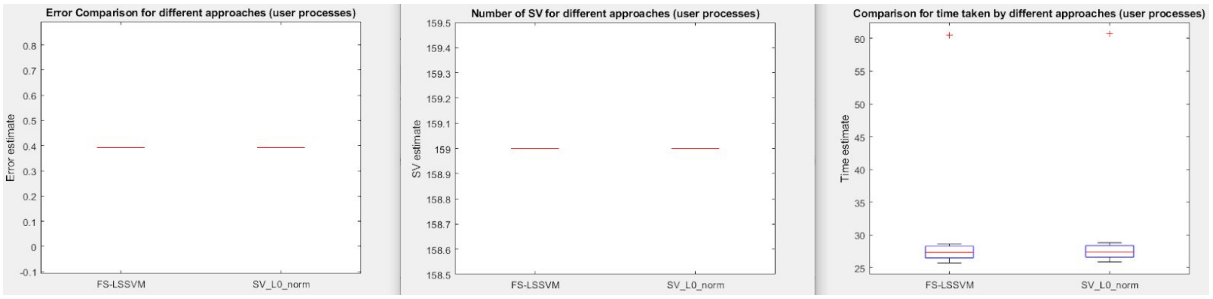
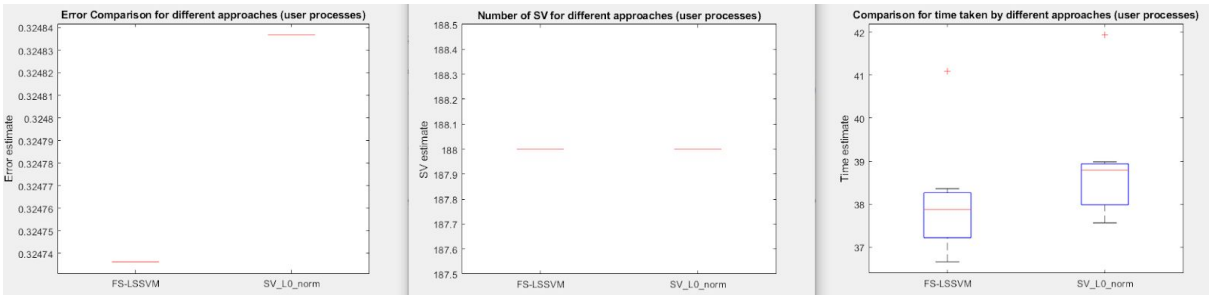
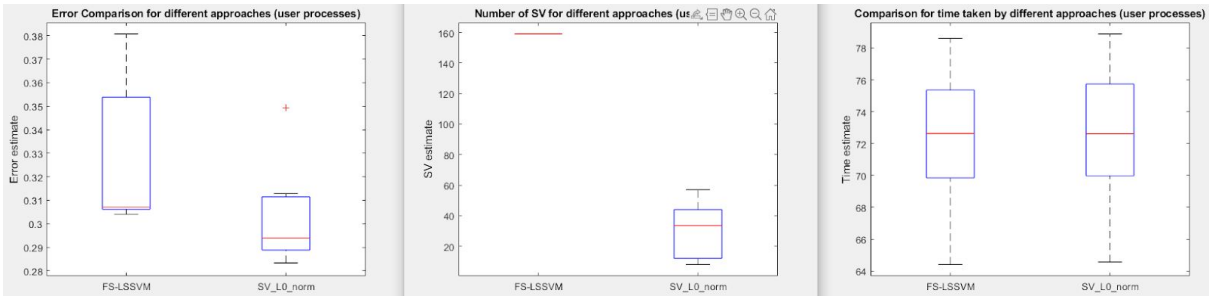


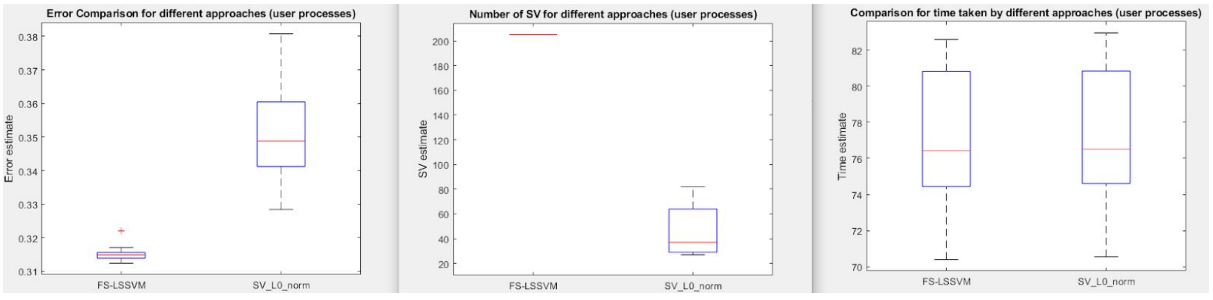
Fig. 8: a) k=6 with polynomial kernel



b) k=8 with polynomial kernel



c) k=6 with RBF kernel



d) k=8 with RBF kernel

With k=6 and polynomial kernel, the error is simply too much. With k=6 and RBF kernel, L0 norm has unacceptable error and LS-SVM has too many SV's.

With k=8 and polynomial kernel, FS-LSSVM error is good and time estimate is great. There are too many SV's though. With k=6 and RBF kernel, L0 norm achieves a very small number of SV's with higher time consumption. One should make a trade-off between them.