Exploring Support Vectors Machines and Least Squares Support Vector Machines

Boris Shilov

June 20, 2019

1 Exercise II: Function estimation and time series prediction

In this exercise I will attempt to explore the least squares support vector machine formulation using the LS-SVM Matlab toolbox [4], particularly function estimation and time series prediction.

Least-squares support vector machine is an extension of the support vector machine approach in which the quadratic problem of the classic SVM formulation is replaced by the problem of solving a set of linear equations, which is more tractable, giving a large improvement in performance. A drawback presents itself in the fact that LS-SVM requires all of the training points to be used, and hence the solution is no longer sparse, unlike the classic approach where one need only preserve a few support vectors. LS-SVM uses a least squares cost function, as the name implies, but also uses equality constraints, unlike SVM [5]. The function to be minimised over the set of weights (or normals) w, the bias term b and the e error to find a solution in LS-SVM is:

$$J_p(\mathbf{w}, e) = \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\frac{1}{2}\sum_{i=1}^{N} e_i^2$$

With equality constraints $d_i = \mathbf{w}^T\varphi(\mathbf{x_i}) + b + e_i$, where φ is some function that maps $\mathbf{x_i}$ into a higher dimensional feature space.

We can vary e and also the C penalty parameter, increasing which places more weight on e variables.

1.1 Support vector machine for function estimation

Using the `uiregress` graphical command, we can construct toy datasets.

First, let us create a dataset where a linear classifier is optimal. One such example is a close cloud of points that are intended to have come from a straight line, which we construct to have 43 points. Increasing e above 0.25 leaves no support vectors, under that the number of support vectors goes up to the number of training points, when C is held fixed at infinity. Decreasing C leads the line to align with the principal direction of the cloud. Setting $C = 0.5$ and $e = 0.1$ leads to a good fit, shown in 1. Noticeably, the sparsity property comes into play here. By setting the e parameter above zero, we ignore errors smaller than this value. Hence, we only use a subset of the training points, achieving a measure of sparseness. This is similar to the ϵ-insensitive loss function in standard SVM [5].

A more challenging toy dataset is one that a linear kernel cannot approximate well. Let us construct one with 46 points that is wave-like. Here, a polynomial kerne of third degree, with $C = 3$ and $e = 0.06$, fits quite well with quite some sparsity.

The relation between least-squares SVM and ordinary least squares is straightforward. Both are types of regression analysis (or, at least, LS-SVM can be used as such), and use the least squares method for solving their respective systems of equations, where the sum of the squared differences between the actual datum and the prediction at the corresponding datum made by the model is minimised. In the case of OLS, linear models are used, whereas we use the SVM formulation for LS-SVM. This leads to, for example, a different definition of how the sum of squared differences (residuals) is calculated, with the following form in the OLS case for one residual:

$$e_i = y_i - x_i^T b$$

Contrast with the LS-SVM formulation of a residual:

$$e_i = y_i - (w^T\varphi(x_i) + b)$$

A yet larger difference between these is that OLS does not have hyperparameters, unlike LS-SVM. OLS derives all of its parameters from the training data, at least in the standard formulation with no shrinkage and such factors. LS-SVM on the other hand always requires some values of C and e (also known as ζ) to be provided.

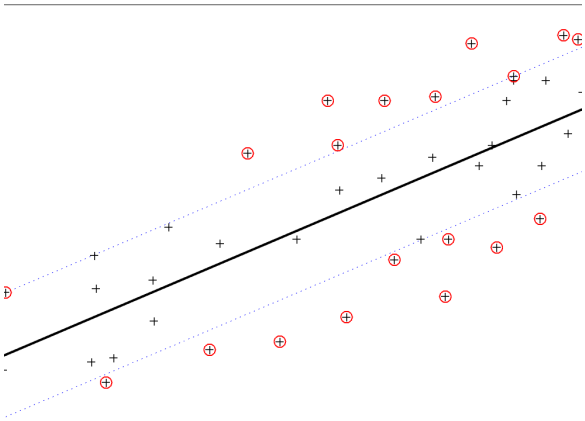

Figure 1: A linear least squares regression fitted to a data cloud of 43 points with parameters $C = 0.5$ and $e = 0.1$. Given these parameters, only 23 points are used as support vectors, labelled in red.

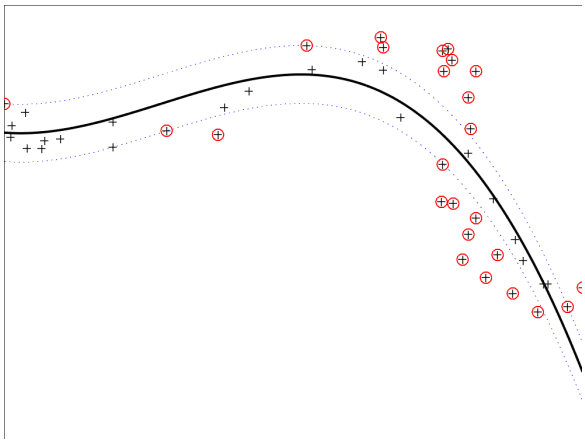

Figure 2: A third degree polynomial least squares regression fitted to a data wave of 46 points with parameters $C = 3$ and $e = 0.06$. Given these parameters, only 26 points are used as support vectors, labelled in red.

Thus, regularisation is built into the standard formulation of LS-SVM through these hyperparameters, it does not have to be extended to implement it like OLS. This makes it easier to avoid overfitting, but introduces the issue of hyperparameter choice.

1.2 Sinc function

We can construct an example dataset using a noisy sinc function. Sinc is thus the true data generating process, and the task of an LS-SVM classifier will be to approximate this function without overfitting to replicate the noise.

$$\mathbf{y} = sinc(\mathbf{x}) + 0.1 * \mathbf{z}$$

Where each $z_i \leftarrow N(\mu, \sigma)$.

We can attempt approximation using the RBF kernel, iterating over a small parameter space of the RBF parameters, that is, values of γ (confusingly, also referred to as σ) and C (strictly speaking not an actual kernel parameter since it comes from the LS-SVM formulation).

```
gams = [10, 10e3, 10e6]; sigmas = [0.01, 1, 100];
[sigmasMesh, gamsMesh] = meshgrid(sigmas, gams);
parameterSpace = num2cell([gamsMesh(:), sigmasMesh(:)], 2);
```

We can then evaluate model performance for each combination of these parameters.

```
perfList = cellfun(@(cell) crossvalidate({ Xtrain , Ytrain , 'f', ...
    cell(:, 1), cell(:, 2), 'RBF_kernel'}, 10, 'mse'), parameterSpace);
```

A plot of this resulting performance is provided in Fig. 3.

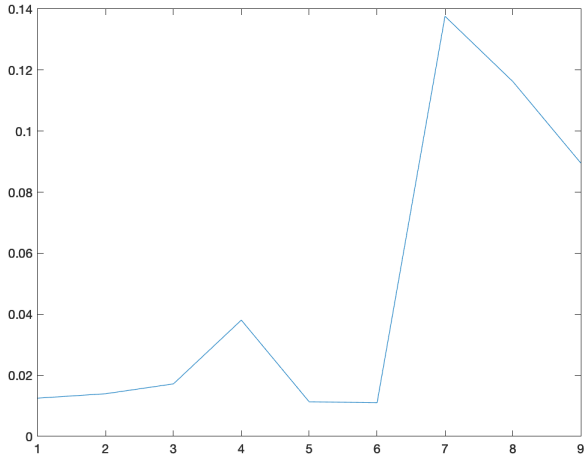

Figure 3: A plot of LS-SVM RBF kernel predictor performance over a range of parameters.

It seems that the fifth and sixth models are the best fitting on training set. Their parameters are $\gamma = 10000$, $c = 1$ and $\gamma = 10000000$, $c = 1$, respectively.

We can choose between them, as well as the first two models which are relatively well fitting, using the AUC of test set performance.

```
niceParams = cellfun(@(index) parameterSpace{index}, {1 2 5 6}, '
    UniformOutput', false);
niceModelSpecs = cellfun(@(cell) { Xtrain , Ytrain , 'f', ...
     cell(:, 1), cell(:, 2), 'RBF_kernel'}, niceParams, 'UniformOutput',
        false);
niceModels = cellfun(@(cell) trainlssvm(cell), niceModelSpecs, '
    UniformOutput', false);
niceSpecAndModel = cellfun(@(index) {niceModelSpecs{index}, ...
     niceModels{index}}, {1 2 3 4}, 'UniformOutput', false);
[Yest, Zt] = cellfun(@(cell) simlssvm(cell{1}, ...
     {cell{2}.alpha, cell{2}.b}, Xtest), niceSpecAndModel, 'UniformOutput',
        false);
test_mses = cellfun(@(cell) immse(cell, Ytest), Yest);
```

A plot of the test set performance shown in Fig. 4 reveals that the fifth model has the best performance.

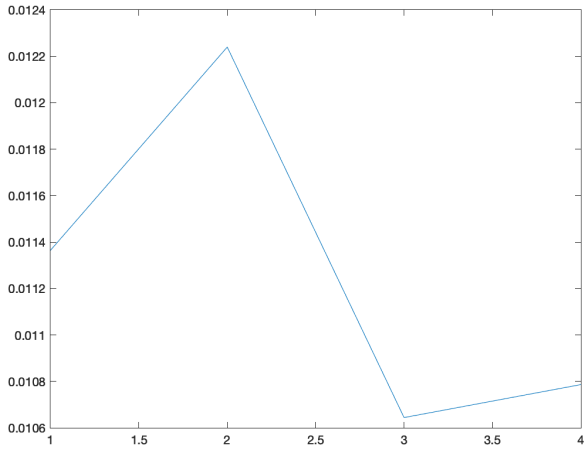

Figure 4: A plot of LS-SVM RBF kernel predictor performance over a range of parameters, on the test set.

It seems the fifth model fits the best on the test case. We may visualise the fifth model on the test data points., shown in Fig. 5. Here the hollow points are the test set that was not used to fit the data.

Since the RBF kernel function estimation SVM is quite flexible, there should exist a pair of parameters that are optimal in the sense that, given these parameters, the resulting SVM is the best approximation of the underlying data-generating function possible. However, as we typically do not know directly the data generating process, a broader optimality criterion for parameters is that the resulting SVM is the best performing on some

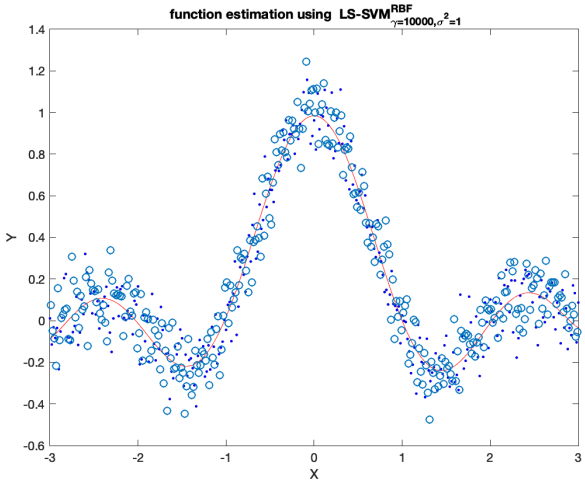

Figure 5: A plot of LS-SVM RBF kernel predictor number five out of all evaluated.

test set among the set of all possible SVMs. Some exceptions to this are possible - for example, there may be multiple near equal maxima or minima in the parameter space. This would correspond to having two or more models that fit the data equally well.

We can compare the resulting model to automatic parameter tuning.

```
[gamSimp, sigSimp, costSimp] = tunelssvm({Xtrain, Ytrain, 'f', ...
    [], [], 'RBF_kernel'}, 'simplex', 'crossvalidatelssvm', {10, 'mse'});
 [gamGrid, sigGrid, costGrid] = tunelssvm({Xtrain, Ytrain, 'f', ...
    [], [], 'RBF_kernel'}, 'gridsearch', 'crossvalidatelssvm', {10, 'mse'});
```

The simplex procedure results in $\gamma = 12727$ and $C = 0.7539$. The gridsearch procedure results in parameters $\gamma = 6212$ and $C = 0.6857$. The cost of both is approximately the same at 0.0108. Thus it appears we have found a better model while not using automatic parameter tuning.

1.3 Bayesian tuning

What is the probability that our data were generated by a particular model? How can we compare models and do regularisation using prior known information? These questions can be answered using Bayesian model comparison methodology. This process follows the principle of maximum parsimony - among all models/parameters/hyperparameters that fit well, choose the simplest [2].

The Bayesian inference process thus provides a full framework to select models, using two levels. First, model fitting is conducted on a single mode, and we infer the most likely parameters via the posterior probability of the parameter vector. In general:

$$P(\mathbf{w}|D, H_i) = \frac{P(D|\mathbf{w}, H_i)P(\mathbf{w}|H_i)}{P(D|H_i)}$$

Where $\mathbf{w}$ is the parameter vector, D is some data and H_i is some model (*hypothesis*). The denominator is the *model evidence*, and since at this first stage the task is to find the most plausible parameters, it is ignored as it does not include the parameter term. The left hand side term is the posterior probability of the particular parameters $\mathbf{w}$. Thus we can pick the best parameters for a given model and data.

Once we obtain some good models, we would like to infer which model is the most probable given the data. This is the task of the second level of inference, where we defined the posterior probability of each model:

$$P(H_i|D) \propto P(D|H_i)P(H_i)$$

The model evidence term $P(D|H_i)$ which we previously dismissed in the first level of inference becomes important. This equation allows us to rank models using their evidence. Further, models with more parameters and thus more complexity are mathematically penalised using the so called Ockham factor (Ockham's razor is another name for the principle of maximum parsimony) [2].

This two-step methodology is further extended for regression problems into a three-step methodology. The first step remains the same with parameter inference, but the second step is split further into a hyperparameter inference step and the model comparison step which also includes kernel parameter inference for the case of models using kernel functions such as LS-SVMs. Importantly, the hyperparameter evidence from the second step here is used in the third step.

We can do this:

```
sig = 0.4; gam = 10;
modelSpecBay = {Xtrain, Ytrain, 'f', gam, sig}
crit_Ls = arrayfun(@(level) bay_lssvm(modelSpecBay, level), [1 2 3])
bayOptims = arrayfun(@(level) bay_optimize(modelSpecBay, level), [1 2 3], '
    UniformOutput', false);
```

The above commands perform this three step process for us. This allows us to obtain a most probable model and error bars to reflect our uncertainty.

```
sigErrs = bay_errorbar({Xtrain, Ytrain, 'f', bayOptims{2}.gam, bayOptims{3}.
    kernel_pars}, 'figure')
```

The model is shown in Fig. 6.

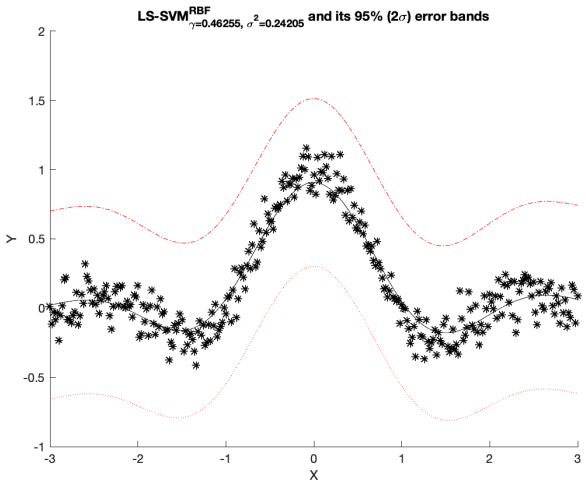

Figure 6: The most probable model derived using Bayesian inference, with computed error bars.

1.4 Automatic Relevance Determination

Bayesian inference can further be used to remove irrelevant features and obtain a sparser subset of explanatory features. This weighs each feature (dimension) of the input space and optimises the weights at the third level of Bayesian inference.

```
X = 6.* rand(100 , 3) - 3;
Y = sinc (X(: ,1)) + 0.1.* randn(100 ,1);
[selected, ranking] = bay_lssvmARD ({X, Y,'f', bayOptims{2}.gam, bayOptims
    {3}.kernel_pars});
```

Selected here are 1 and 2 and the ranking is $1, 2, 3$. This means that the first two dimensions of the input space are the most important, and the ranking of importance confirms this, with the third dimension being the least important.

This procedure has been shown to be equivalent to doing maximum a posteriori estimation, so an equivalent approach to this is possible using the `crossvalidate` function [6].

1.5 Robust regression

If the data are noisy, robust regression can be used to attempt to disregard the noise and/or outliers to fit closer to the underlying function. We construct a similar sinc dataset to demonstrate this, this time adding more noise.

```
dataGeneratingFun = @(X) sinc(X) + 0.1 * rand(size(X));
X = transpose(-6:0.2:6); Y = dataGeneratingFun(X);
outSet1 = [15 17 19]; outSet2 = [41 44 46];
Y(outSet1) = 0.7 * 0.3 + rand(size(outSet1)); Y(outSet2) = 1.5 * 0.2 + rand(
    size(outSet2));
```

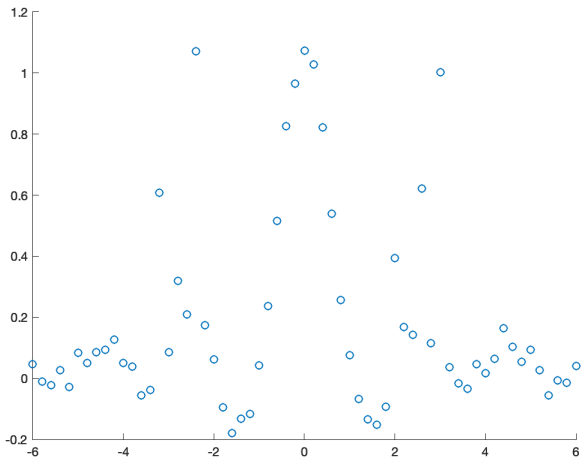

Figure 7: A noisy sinc dataset.

The resulting dataset is shown in Fig. 7.

What happens if we try naively fitting on these data? Fig. 8 demonstrates.

```
Model = initlssvm(X, Y, 'f', [], [],'RBF_kernel');
tunedNaiveModel = tunelssvm(Model, 'simplex', ...
    'crossvalidatelssvm', {10, 'mse'});
```

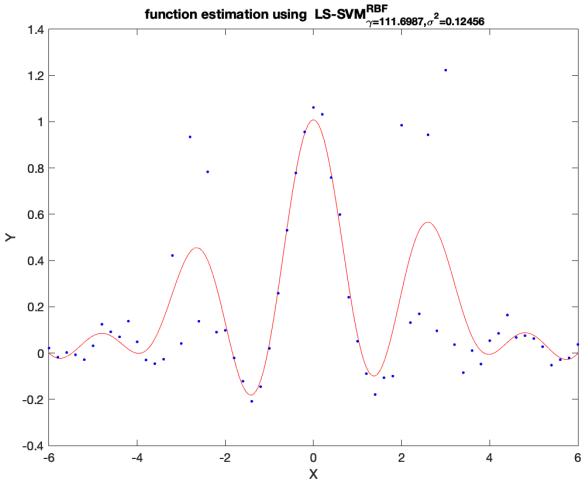

Figure 8: The result of fitting a non-robust LS-SVM model to the noisy sinc data.

Within the outlier regions, the estimation is very far from the true sinc function, being highly influenced by just three points in each region. This also has effects outside the outlier regions. By contrast, a robust model should minimize the influence of support vectors with big errors on the final function estimate. A robust Huber's weights function model on this data will generate warning about a lack of inverse but will nonetheless produce a fit. As can be seen in Fig. 9, this is very satisfactory and practically ignores the outliers.

Here the MSE function is used in the naive formulation - since the RMSE will square the residuals before summing them, this amplifies the influence of outliers. By contrast, MAE used in the robust regression penalizes large errors.

Other weight functions produce varying results. Hamper weights leads to a good approximation of the underlying function as seen in Fig. 10, slightly better than the Huber weight fit. Logistic weights produce approximately the same result as Huber, seen in Fig. 12. Myriad weights, on the other hand, appear to be no better than the naive fit, seen in Fig. **??**.

1.6 Time Series Prediction with Logmap Data

Time series prediction as done here is nonlinear autoregression exogenous (NARX) modelling. That means that we create a model that uses both the previous values of the dimension of interest and some other, exogenous, variables to predict the future.

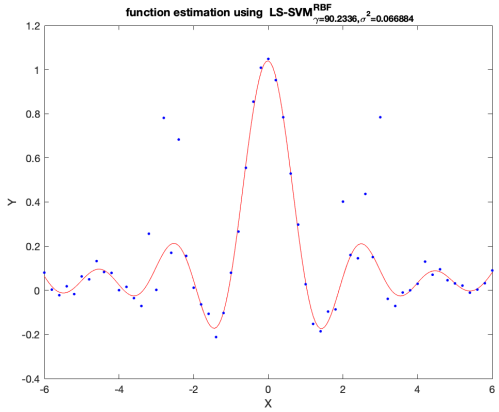

Figure 9: An LS-SVM regression with a Huber robust loss function on the noisy sinc data.

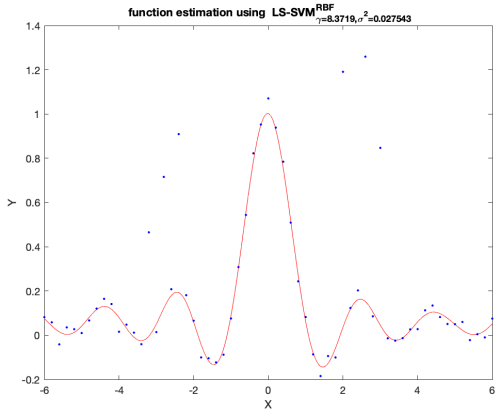

Figure 10: An LS-SVM regression with Hamper weights on the noisy sinc data.

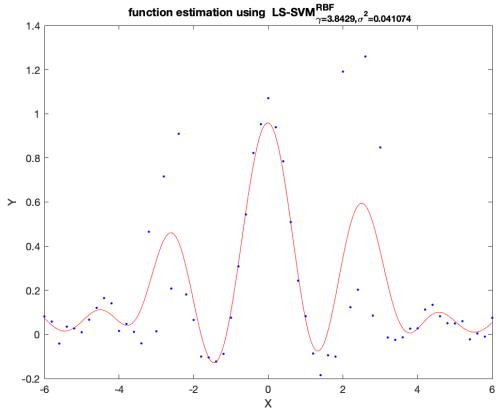

Figure 11: An LS-SVM regression with myriad weights on the noisy sinc data.

This time series dataset contains a number of points in time. The test set is defined to be the points at the last 50 timepoints. After loading the dataset, we can attempt naive prediction of these test points.

```
order = 10;
X = windowize(Z, 1:(order+1));
Y = X(:, end);
X = X(:, 1:order);
gam = 10; sig = 10;
timeSeriesModel1 = trainlssvm({X, Y, 'f', gam, sig});
Xs = Z(end - order + 1:end, 1);
nb = length(Ztest);
```

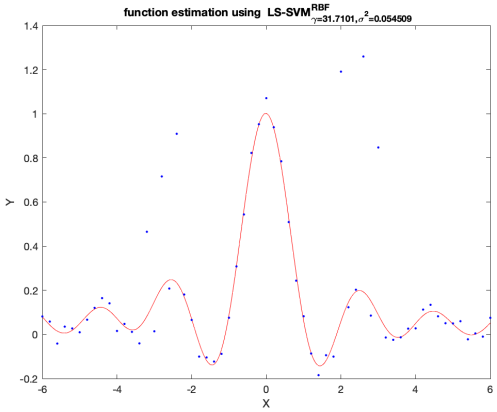

Figure 12: An LS-SVM regression with Logistic weights on the noisy sinc data.

```
prediction = predict({X, Y, 'f', gam, sig2}, Xs , nb);
```

This yields predictions on the test set as depicted in Fig. 13. Even discounting the fact that time series prediction is notoriously difficult, this is very poor.

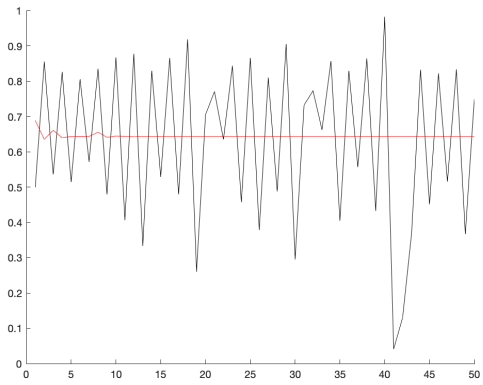

Figure 13: The result of applying a non-optimised time series prediction model to the logmap dataset, with the test set in black and the predictions in red.

Instead, we may use grid search to select a better model. First, use Bayesian inference to select good gam and sig values. Then, we do the grid search.

```
 gam = 2.4; sig = 24.2154; nb = length(Ztest);
 orderCell = {5 10 20 30 40 50 60 70 80 90 100};
windowizedData = cellfun(@(order) {order, windowize(Z, 1:(order+1))},
    orderCell, 'UniformOutput', false);
Yes = cellfun(@(tuple) tuple{2}(:, end), windowizedData, 'UniformOutput',
    false);
Xes = cellfun(@(tuple) tuple{2}(:, 1:tuple{1}), windowizedData, '
    UniformOutput', false);
Xses = cellfun(@(order) Z(end - order + 1:end, 1), orderCell, 'UniformOutput
    ', false);
YandXandXses = arrayfun(@(index) {Yes{index} Xes{index} Xses{index}}, 1:
    length(orderCell), 'UniformOutput', false);
predictions = cellfun(@(cell) predict({cell{2} , cell{1} , 'f', ...
     gam, sig}, cell{3} ,nb), YandXandXses, 'UniformOutput', false);
```

One of the better resulting models is shown in Fig. 14. Here we see that prediction is very good for some time points and remains relatively poor for others.

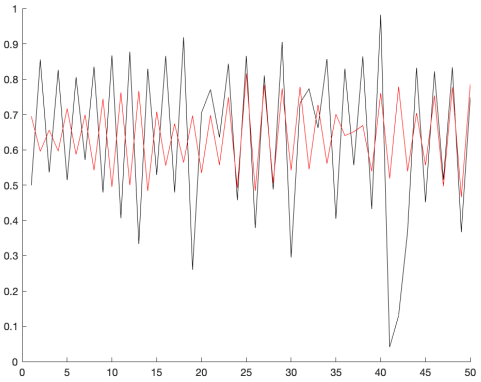

Figure 14: One of the models resulting from grid search, with the model parameters initially coming from Bayesian inference, with the test set in black and the predictions in red.

1.7 Time Series Prediction with Santa Fe Data

We may use the same strategy to optimise gam, sig and the order parameters on the Santa Fe dataset. Here a naive fit without optimising parameters results in a bad model. Bayesian inference finds the values $gam = 62.47$ and $sig = 9.25$ to be good, and we proceed to do grid search as previously, over the same magnitude of orders. At $order = 50$, we already have quite good approximation, as can be seen in Fig. 15.

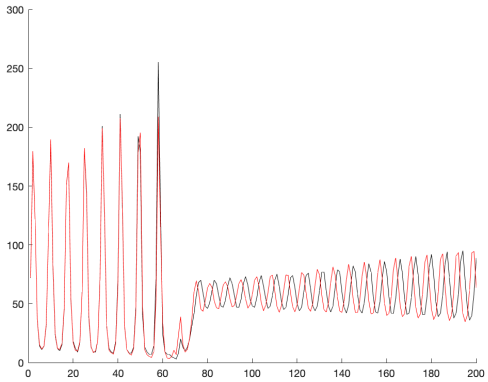

Figure 15: The result of first optimising parameters using Bayesian inference and then doing a grid search of model orders with the Santa Fe dataset, with the test set in black and the predictions in red.

2 Exercise III: Unsupervised Learning and Large Scale Problems

2.1 Kernel principal component analysis

Kernel PCA is a dimensionality reduction technique that relies on the fact that it is possible to linearly separate what seems like a non-linearly separable dataset by the addition of extra dimensions. This addition is accomplished using some function that maps our input data into this higher dimensional feature space in which such separation is possible. However, this function and the resulting kernel are not usually mathematically tractable to work with, so we cannot compute the principal components directly (nor indeed any part of the feature space), for example, but we can use the kernel trick to project existing data onto the components. The workaround is possible because what we can do is compute the inner product space of the feature space, which gives us a way of computing linear independence using the Gramian matrix, hence leading to the possibility of projection of the data onto the uncomputed principal components.

Denoising using KPCA works similarly to linear PCA. Because the first n principal components will explain a given amount of variance in their respective orthogonal directions, by applying PCA we can effectively denoise the data by dropping PCs that explain smaller amounts of variance than some threshold we determine, as determined by the eigenvalues corresponding to the PCs. However, a drawback of linear PCA is that by doing

so, interesting nonlinearity in the data may be lost. KPCA allows us to use nonlinear features to try and preserve that information, by first using the nonlinear mapping function and then doing linear PCA on the data, but avoiding explicitly computation of the feature space using Mercer theorem to ensure that the kernels we apply are able to have a Gramian matrix [3].

If we increase the number of principal components, to a certain point we are capturing more genuine variance, so we improve performance. However, we eventually begin incorporating components which are mostly noise. Hence, here performance starts to fall.

We see this readily with KPCA on a toy ying yang dataset. For the Lanczos approximation method, using up to 10 PCs seems to yield an OK denoising, yet at some point above 10 PCs, we capture the noise as well, as can be seen in Fig. 19.

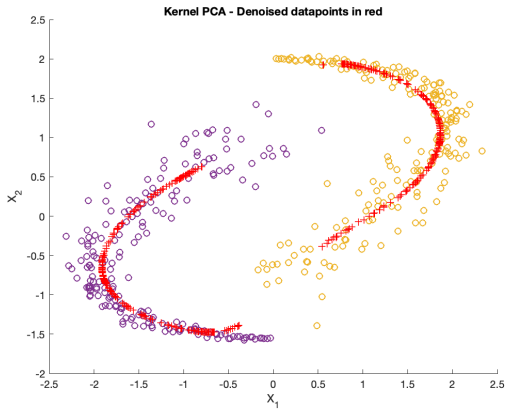

Figure 16: KPCA with RBF kernel, $sig2 = 0.4$ and Lanczos approximation, using the first 4 PCs.

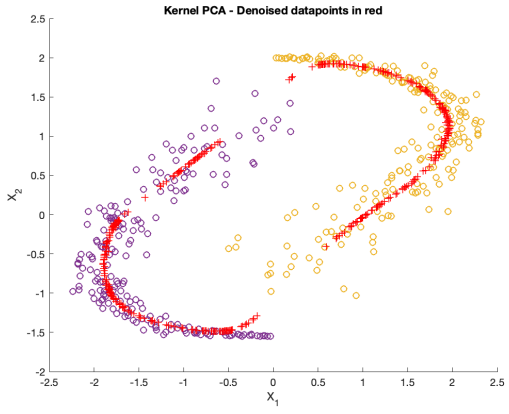

Figure 17: KPCA with RBF kernel, $sig2 = 0.4$ and Lanczos approximation, using the first 6 PCs.

In order to tune the parameters in this case we can use crossvalidation with the reconstruction error as the measure of the best model [3]. We can use a similar approach to choose the kernel and the number of principal components via crossvalidation, also using the reconstruction error, but of pre-images in original space [1].

2.2 Spectral clustering

Spectral clustering uses a similarity matrix computed using some similarity function (kernel function in our case) as a way to observe local structure. We form the Laplacian by substracting the symmetric matrix from the degree matrix (which is diagonal, with each element on the diagonal being the number of edges each vertex defined by the kernel function has). The eigenvectors of the Laplacian matrix are then compute (its spectrum, hence the name). The actual clustering is then done on these eigenvectors, usually using k-means, but other clustering algorithms can be used.

As the name implies, this is a clustering technique, not a classification technique. Clustering discovers groups within the data based on some metric. Clustering techniques allow us to explore our data. By contrast, in classification we usually already know the groups in our data (these can even be previously discovered via

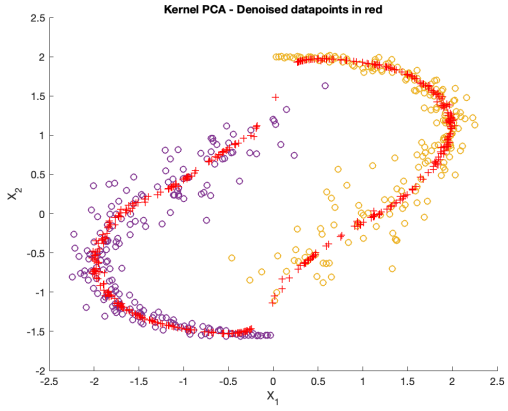

Figure 18: KPCA with RBF kernel, $sig2 = 0.4$ and Lanczos approximation, using the first 10 PCs.

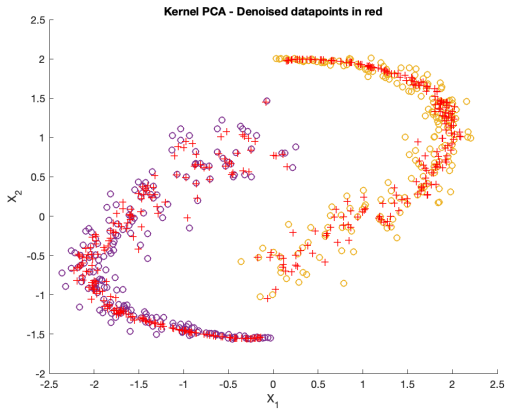

Figure 19: KPCA with RBF kernel, $sig2 = 0.4$ and Lanczos approximation, using the first 15 PCs.

clustering). The objective of classification is to create a classifier that will assign new examples to the groups we already know - that is, prediction.

Attempting spectral clustering with the example interleaved rings dataset, the influence of the similarity metric on clustering can clearly be seen by varying the parameters of our similarity function - that is, the RBF kernel. Starting at $sig2 = 0.05$ yields a grouping that is only somewhat correct, and is not acceptable if one is most interested in the region where the rings interleave, shown in Fig. 20. In Fig. 21 and 22 we see that $sig2$ in the region of 0.015 is a good choice, and neatly separates the rings correctly. Choosing a much lower value leads to the similarity function asserting that both rings are part of the same group, as can be seen in Fig. 23.

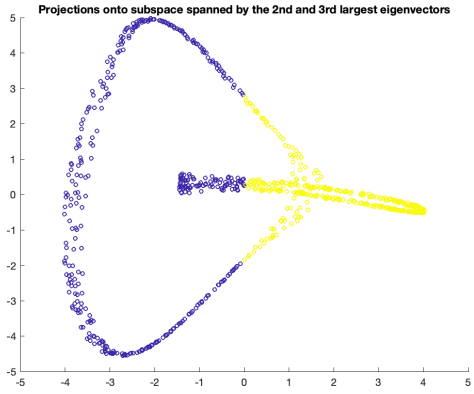

Figure 20: Spectral clustering using the RBF kernel with $sig2 = 0.05$.

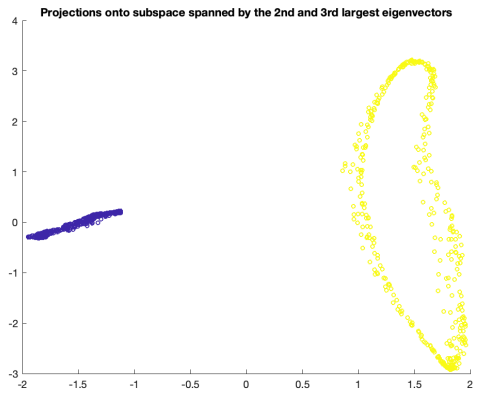

Figure 21: Spectral clustering using the RBF kernel with $sig2 = 0.02$.

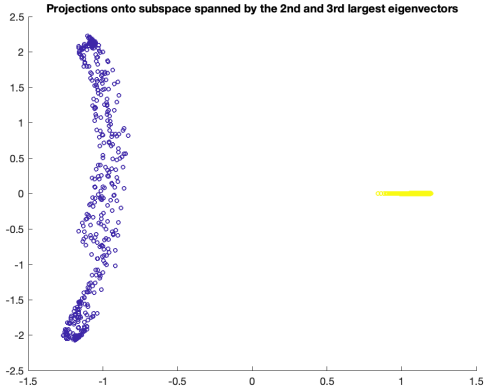

Figure 22: Spectral clustering using the RBF kernel with $sig2 = 0.01$.

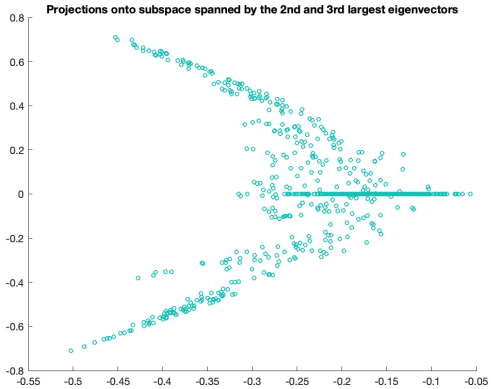

Figure 23: Spectral clustering using the RBF kernel with $sig2 = 0.001$.

2.3 Fixed size LS-SVM

Projecting data points onto a higher dimensional feature space required representing the data as a Gramian matrix, as described previously. This becomes problematic with large amounts of data because the Gramian matrix grows to be large, and further computations may involve very expensive matrix inversions. Previously we have mentioned using Lanczos bidiagonalisation, which is a type of low rank approximation. Nystrom is another such approximation method. These allow us to bypass using the full Gramian matrix by substituting a low rank approximation of it into our calculations - hence we use only a smaller block of the Gramian. In order to actually create this approximation we require a selection of training points to be pre-specified. This operates on the primal representation of an LS-SVM model. By using a block matrix we thus avoid the more expensive calculations on the full Gramian.

The choice between operating on the primal or dual representation in the case our mapping is known and not infinite dimensional depends mainly on the dimension of the set of weights in the primal and dual representation. If the dimension in primal is small but the dimension in dual is large it is computationally more convenient to use the primal formulation, and vice versa.

Modifying the RBF kernel parameter $sig2$ in FS-LS-SVM appears to result in different subsets of points being chosen. Namely, it appears that lower values result in points closer to the center of the data clouds being chosen, while higher values result in points more on the edge. This is seen by comparing Fig. 24 and 25.

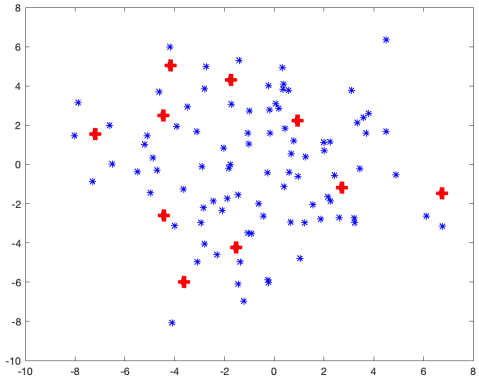

Figure 24: FS-LS-SVM using the RBF kernel with $sig2 = 0.1$.

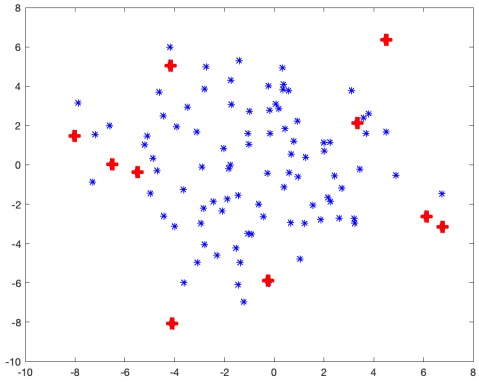

Figure 25: FS-LS-SVM using the RBF kernel with $sig2 = 100$.

We can attempt to apply L_0 regularisation to some initial FS-LS-SVM solution iteratively to try and obtain a sparser solution and hence reduce overfitting. Fig. 26 shows that the error estimate between just FS-LS-SVM and the regularised solution is about the same. Fig. 28 also shows that on this shuttle dataset the time performance for the computations is very similar. However, as expected, regularisation reduces the number of support vectors required to describe the same solution substantially, as can be seen in Fig. 27.

2.4 Digits denoising using kernel PCA

We can attempt to denoise a collection of handwritten Arabic numbers that have variable amounts of noise added onto them. We control the σ parameter using a multiplicative factor and the same for the noise parameter. Fig. 29 and 30 depict the denoising done using kernel PCA and linear PCA, respectively, with the parameters being 0.7 and 1, respectively. It appears that at this rather substantial level of noise kernel PCA is able to denoise much better than linear.

What influence does the σ parameter have on denoising? If we make the multiplicative factor equal 30, we get results shown in Fig. 31 for kernel PCA and Fig. 32 for linear PCA. The performance of both methods appears poor at this high σ value, highlighting the need for careful parameter picking.

For very small values of σ, we can use a multiplicative factor of 0.01 to see the effect. For kernel PCA this is depicted in Fig. 33 and for linear PCA in Fig. 34. Here, kernel PCA appears to perform well at denoising, but

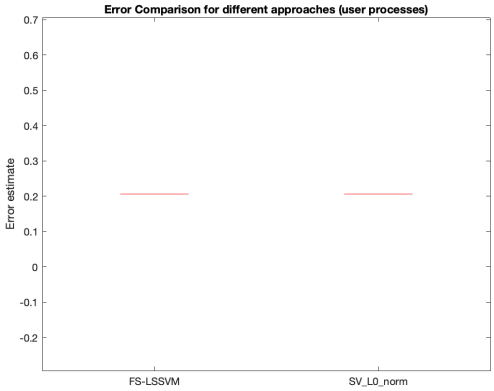

Figure 26: FS-LS-SVM on its own versus FS-LS-SVM followed by L_0 regularisation - error performance.

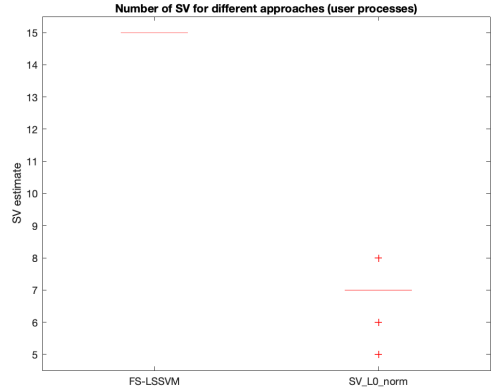

Figure 27: FS-LS-SVM on its own versus FS-LS-SVM followed by L_0 regularisation - number of support vectors.

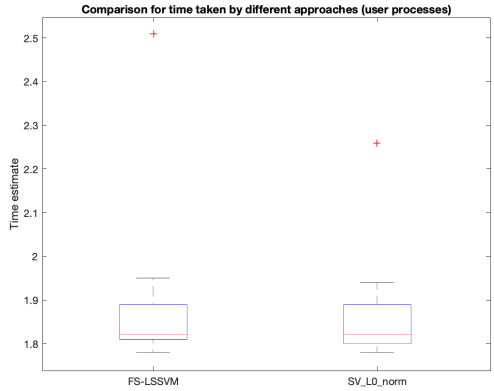

Figure 28: FS-LS-SVM on its own versus FS-LS-SVM followed by L_0 regularisation - time to compute.

in actuality the denoised numerals do not correspond to the original numerals. If we did not know the originals beforehand, this highlights the importance of having a test set.

References

[1] Alam. Hyperparameter Selection in Kernel Principal Component Analysis. *Journal of Computer Science 10*, 7 (July 2014), 1139–1150.

[2] MacKay, D. J. Bayesian Interpolation. *NEURAL COMPUTATION 4* (1991), 415–447.

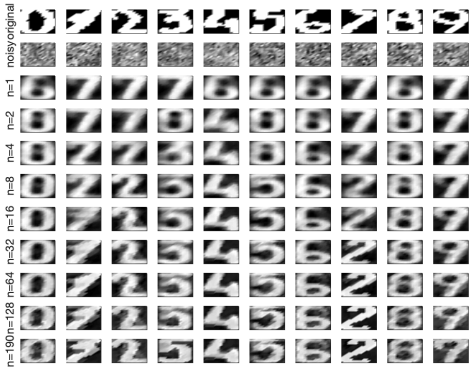

Figure 29: Denoising of handwritten Arabic numerals using a kernel PCA with σ factor 0.7.

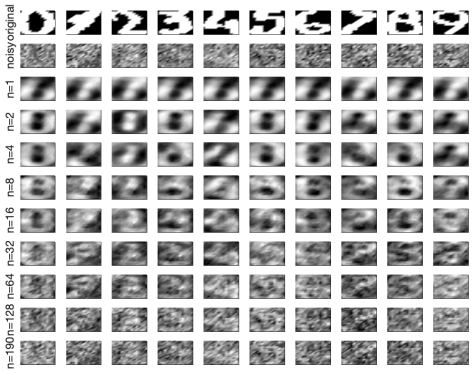

Figure 30: Denoising of handwritten Arabic numerals using a linear PCA with σ factor 0.7.

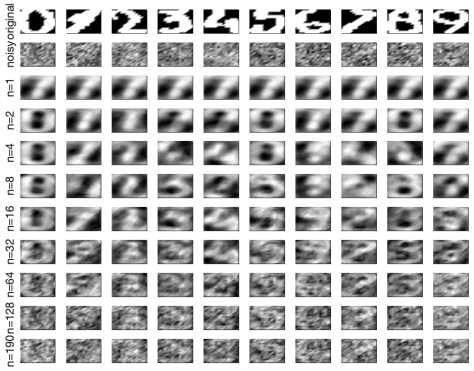

Figure 31: Denoising of handwritten Arabic numerals using a kernel PCA with σ factor 30.

[3] Mika, S., Scholkopf, B., Smola, A., Muller, K.-R., Scholz, M., and Riitsch, G. Kernel PCA and De-Noising in Feature Spaces. 7.

[4] Pelckmans, K., Suykens, J. A. K., Gestel, T. V., Brabanter, J. D., Lukas, L., Hamers, B., Moor, B. D., and Vandewalle, J. LS-SVMlab: A MATLAB/C toolbox for Least Squares Support Vector Machines. 8.

[5] Valyon, J., and Horváth, G. A Robust LS-SVM Regression. *Int Journal of Comp and Infor Engineering 1*, 7 (2007), 6.

[6] Wipf, D. P., and Nagarajan, S. S. A New View of Automatic Relevance Determination. 8.

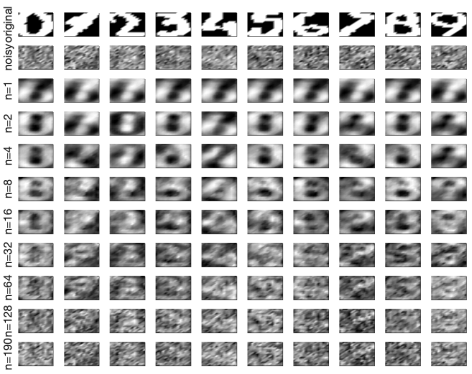

Figure 32: Denoising of handwritten Arabic numerals using a linear PCA with σ factor 30.

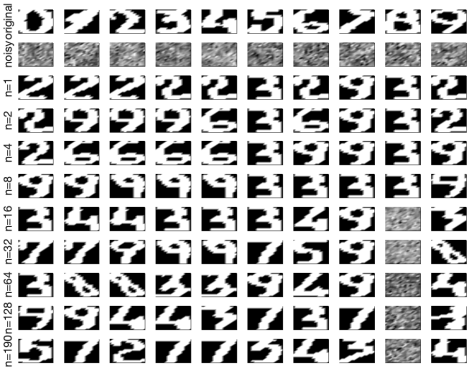

Figure 33: Denoising of handwritten Arabic numerals using a kernel PCA with σ factor 0.01.

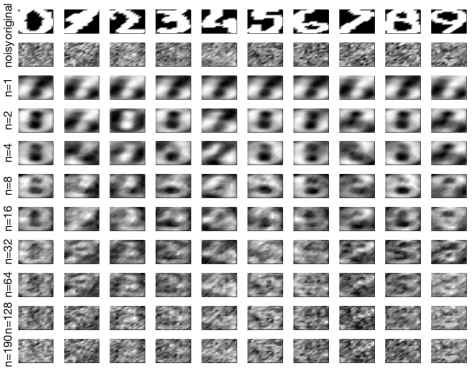

Figure 34: Denoising of handwritten Arabic numerals using a linear PCA with σ factor 0.01.