Advanced Machine Learning
Practical 3: Classification
(SVM, RVM & AdaBoost)

Professor: Aude Billard
Assistants: Mikhail Koptev and Mert Kayaalp
E-mails: aude.billard@epfl.ch,
mikhail.koptev@epfl.ch, mert.kayaalp@epfl.ch

Spring Semester 2021

1 Introduction

During this week's practical we will focus on understanding and comparing the performance of the different non-linear classification methods seen in class. We will begin by studying the influence of hyper-parameters on *kernel* methods, such as SVM and its variants (C-SVM and ν-SVM, RVM) on 2D Datasets. Then, we will try an instance of a *Boosting* method, specifically AdaBoost, and compare its performance to the *kernel methods*.

2 ML_toolbox

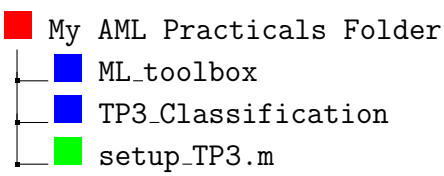
`ML_toolbox` contains a set of methods and examples for easily learning and testing machine learning methods on your data in MATLAB. It is available in the following link:

`https://github.com/epfl-lasa/ML_toolbox`

From the course Moodle webpage (or the website), the student should download and extract the .zip file named `TP3-Classification.zip` which contains the following files:

Code	**Datasets**	setup_TP3.m
Code/TP3_svm.m	Datasets/rvm-ripley-dataset.mat	
Code/TP3_rvm.m	Datasets/very-nonlinear-data.mat	
Code/TP3_adaboost.m		

Before proceeding make sure that `./ML_toolbox` is at the same directory level as the TP directory `./TP3-Classification` and `setup_TP3.m` as follows:



Now, to add these directories to your search path, type the following in the MATLAB command window:

```
>> setup_TP3.m
```

NOTE: To test that all is working properly with ML_Toolbox you can try out some examples of the toolbox; look in the **examples** sub-directory.

3 Non-linear Classification Techniques

In classification problems, we typically have a dataset $D = \{(\mathbf{x}^1, y_1), \ldots, (\mathbf{x}^M, y_M)\}$ where $\mathbf{x}^i \in \mathbb{R}^N$ is the i-th N-dimensional data point (or feature vector) from M samples and $y_i \in \{-1, 1\}$ is the categorical outcome (or class label) corresponding to each data point. The goal is then to learn a mapping function $y = f(\mathbf{x}) : \mathbb{R}^N \rightarrow \mathbb{Z}$, such that, given a new sample (or query point) $\mathbf{x}' \in \mathbb{R}^N$ we can predict its label; i.e. $y' = f(\mathbf{x}') \in \{-1, 1\}$ for the binary classification case. If a dataset is linearly separable, then $f(\mathbf{x})$ can be a simple linear function, this, however, is rarely the case for real-world datasets. Hence, one must apply non-linear classification techniques in order to find the non-linear decision boundaries in the dataset.

In this practical, we will cover two types of techniques: (i) *kernel methods* and (ii) *Boosting*. For the former, we will study Support Vector Machines **(SVM)** and Relevance Vector Machines **(RVM)**. These methods encode the decision boundary as a hyper-plane in feature space $\phi(\mathbf{x}) : \mathbb{R}^F$. They follow the same logic as previously seen kernel methods:

1. Lift data to high-dimensional feature space $\phi(\mathbf{x}) : \mathbb{R}^F$, where it might be separable.
2. Apply linear operations in that space.

To learn $f(\mathbf{x})$ with SVM/RVM, we must find the optimal hyper-parameters[1] which will optimize the objective function (and consequently the parameters of the class decision function). Choosing the best hyper-parameters for your dataset is not a trivial task, we will analyze their effect on the classifier and how to choose an admissible range of parameters. For the *Boosting* technique, we will focus on the well-known Adaboost algorithm. This method, constructs a *strong* classifier as a linear combination of *weak* classifiers. The *weak* classifiers (WC) are chosen iteratively among a large set of randomly created WC and combined by updating weights of data-points not well classified by the previous combination of WC. We will compare the performance of SVM vs Adaboost (with decision stumps as the WC) on multiple datasets.

A standard way of finding optimal hyper-parameters is by doing a **grid search** with **cross-validation** over a range of parameters, i.e. systematically evaluating each possible combination of hyper-parameters within a given range. We will do this for different datasets and discuss the difference in performance, model complexity and sensitivity to hyper-parameters.

[1] C: penalty for C-SVM, ν: bounds for ν-SVM and kernel type hyper-parameters for all methods.

Classification Metrics

Many metrics have been proposed to evaluate the performance of a classifier. Most of which, come from a combination of values from the **confusion matrix** (or error matrix)[2]. In this matrix, the rows represents the real classes of the data and the columns the estimated classes. The diagonal represents the well classified examples while the rest indicates confusions. In the case of a binary classifier (i.e. $y \in \{-1, 1\}$), the following quantities have to be computed:

- **True positives (TP)**: number of test samples with a positive estimated label for which the actual label is also positive (good classification)
- **True negatives (TN)**: number of test samples with a negative estimated label for which the actual label is also negative (good classification)
- **False positives (FP)**: number of test samples with a positive estimated label for which the actual label is negative (classification errors)
- **False negatives (FN)**: number of test samples with a negative estimated label for which the actual label is positive (classification errors)

Table 1: Confusion matrix

		Estimated labels	
		Positive	**Negative**
Real labels	**Positive**	True positives (TP)	False negatives (FN)
	Negative	False positives (FP)	True negatives (TN)

In this practical, we will use the following metrics to evaluate our classification algorithms:

1. Accuracy: The classification accuracy represents the percentage of correctly classified data points, as follows:

$$ACC = \frac{TP + TN}{P + N} \tag{1}$$

2. F-measure: The $\mathcal{F}$-measure is a well-known classification metric which represents the harmonic mean between Precision ($P = \frac{TP}{TP+FP}$) and Recall ($R = \frac{TP}{TP+FN}$)

$$\mathcal{F} = \frac{2PR}{P + R} \tag{2}$$

It conveys the balance between *exactness* (i.e. precision) and *completeness* (i.e. recall) of the learned classifier.

[2]See this wikipedia entry for more metrics: `https://en.wikipedia.org/wiki/Confusion_matrix`

Grid Search with Cross-Validation

To tune the optimal hyper-parameters in methods such as SVM and RVM, we can do an exhaustive search over a grid of the parameter space. This is typically done by learning the decision function for each combination of hyper-parameters and computing a metric of performance. However, testing the learned decision function on the data used to train it is a big mistake, due to over-fitting. This is where **Cross-Validation (CV)** comes in:

"Cross-validation refers to the practice of confirming an experimental finding by repeating the experiment using an independent assay technique" (Wikipedia).

In machine learning, CV consists in *holding out* part of the data to *validate* the performance of the model on un-seen samples. The two main CV approaches are k-fold and Leave One Out (LOO). In this practical we will use the former. In k-fold CV the training set is split into k smaller sets and the following procedure is repeated k times:

1. Train a model using $k-1$ folds as training data.
2. Validate the model by computing a classification metric on the remaining data.

For grid search, the previous steps are repeated k times for each combination of hyper-parameters in the parameter grid. The overall classification performance can then be analyzed through the statistics (e.g. mean, std. deviation, etc) of the metrics computed from the k validation sets.

4 Support Vector Machines (SVM)

SVM is one of the most powerful non-linear classification methods to date, as it is able to find a separating hyper-plane for non-seperable data on a high-dimensional feature space using the kernel trick. Yet, in order for it to perform as expected, we need to find the 'best' hyper-parameters given the dataset at hand. In this section we will compare its variants (C-SVM, ν-SVM) and get an underlying intuition of the effects of its hyper-parameters and how to choose them.

4.1 C-SVM

Given a data set $D = \{(\mathbf{x}^1, y_1), \ldots, (\mathbf{x}^M, y_M)\}$ of M samples, for $\mathbf{x}^i \in \mathbb{R}^N$, in which the class label $y_i \in \{-1, +1\}$ is either positive or negative, C-SVM seeks to minimize the following optimization problem:

$$\begin{aligned} &\min_{\mathbf{w}\in\mathcal{H}, \xi\in\mathbb{R}^M} \left(\frac{1}{2}||\mathbf{w}||^2 + \frac{C}{M}\sum_{i=1}^{M}\xi_i \right) \\ &\text{s.t.} \quad y_i \left(\langle \mathbf{w}, \Phi(\mathbf{x}^i) \rangle + b \right) \geq 1 - \xi_i, \quad \forall i = 1, \ldots M \end{aligned} \tag{3}$$

where $\xi_i \geq 0$ are the slack variables, $b \in \mathbb{R}$ is the bias, $C \in \mathbb{R}$ is a penalty factor used to trade-off between maximizing the margin and minimizing classification errors; $\Phi : \mathbb{R}^N \to \mathbb{R}^F$ is mapping function of the input data into some higher-dimensional Hilbert space $\mathcal{H}$ and $\mathbf{w} \in \mathcal{H}$ is orthogonal to the separating hyper-plane in that space.

As seen in class, $\mathbf{w} \in \mathcal{H}$ can be expressed as a linear combination of the training (feature) vectors:

$$\mathbf{w} = \sum_{i=1}^{M} \alpha_i y_i \Phi(\mathbf{x}^i) \tag{4}$$

where $\alpha_i > 0$ for support vectors. The constraint in (3) can then be represented as: $y_i \left(k(\mathbf{x}, \mathbf{x}_i) + b\right) \geq 1 - \xi_i$, where $k(.)$ is a positive definite kernel, representing the dot product $k(\mathbf{x}, \mathbf{x}_i) = \langle \Phi(\mathbf{x}), \Phi(\mathbf{x}_i) \rangle$ of the data-points in $\mathcal{H}$. The SVM decision function $y(\mathbf{x}) \to \{-1, +1\}$ then takes the following form:

$$y(\mathbf{x}) = \text{sgn}\left(\sum_{i=1}^{M} \alpha_i y_i \, k(\mathbf{x}, \mathbf{x}^i) + b\right), \tag{5}$$

whose parameters are estimated by maximizing the *Lagrangian dual* of (3) wrt. α,

$$\begin{aligned} &\underset{\alpha_i \geq 0}{\text{maximize}} \sum_{i=1}^{M} \alpha_i - \frac{1}{2} \sum_{i=1}^{M} \sum_{j=1}^{M} \alpha_i \alpha_j y_i y_j k(\mathbf{x}^i, \mathbf{x}^j) \\ &\text{s.t.} \quad 0 \geq \alpha_i \geq C \quad \forall i, \quad \sum_{i=1}^{M} \alpha_i y_i = 0. \end{aligned} \tag{6}$$

where $k(\mathbf{x}, \mathbf{x}^i)$ is the kernel function, which will be the Radial Basis function for this tutorial. The parameters for this decision function are learned by solving the Lagrangian dual of the optimization problem (3) in feature space. As this was seen in class, it will not be covered here. There are extensions such to be able to handle multi-class problems, but in this tutorial we will be focusing on the binary case.

Kernels: SVM has the flexibility to handle many types of kernels, we have three options implemented in **ML_toolbox**:

- Homogeneous Polynomial: $k(\mathbf{x}, \mathbf{x}^i) = (\langle \mathbf{x}, \mathbf{x} \rangle)^p$,
  where $p > 0$ and corresponds to the polynomial degree.
- Inhomogeneous Polynomial: $k(\mathbf{x}, \mathbf{x}^i) = (\langle \mathbf{x}, \mathbf{x} + d \rangle)^p$,
  where $p > 0$ and corresponds to the polynomial degree and $d \geq 0$, generally $d = 1$.
- Radial Basis Function (Gaussian): $k(\mathbf{x}, \mathbf{x}^i) = \exp\left\{-\frac{1}{2\sigma^2}||\mathbf{x} - \mathbf{x}^i||^2\right\}$,
  where σ is the width or scale of the Gaussian kernel centered at $\mathbf{x}^i$

Hyper-parameters: Depending on the kernel, one has several open hyper-parameters to choose. For example, when using the RBF kernel, we would need to find an optimal value for both C and σ. Intuitively, C is a parameter that trades-off the misclassification of training examples against the *simplicity* of the decision function. The lower the C, the smoother the decision boundary (with risk of some misclassifications). Conversely, the higher the C, the more support vectors are selected far from the margin yielding a more fitted decision boundary to the data-points. Intuitively, σ is the radius of influence of the selected support vectors, if σ is very low, it will be able to encapsulate only those points in the feature space which are very close, on the other hand if σ is very big, the support vector will influence points further away from them.

`Illustrative example:` To see the effects of the hyper-parameters on C-SVM with RBF Kernel, open the accompanying MATLAB script: **TP3_svm.m** and run the first code sub-block `1a` to load the circles dataset shown in Figure 1(a). In code sub-block `2(a)` you can set the desired hyper-parameter values as follows:

```
%% 2a) EXAMPLE SVM OPTIONS FOR C-SVM with RBF Kernel%
clear options
options.svm_type    = 0;    % 0: C-SVM, 1: nu-SVM
options.kernel_type = 0;    % 0: RBF, 1: Polynomial (0 is the default)
options.C           = 1;    % Cost (C \in [1,1000]) in C-SVM
options.sigma       = 1;    % radial basis function: ...
```

The next code sub-block `2(b)` gives an example of setting the hyper-parameters for a polynomial kernel. After choosing these parameters you can run the train-svm code presented in the following sub-block:

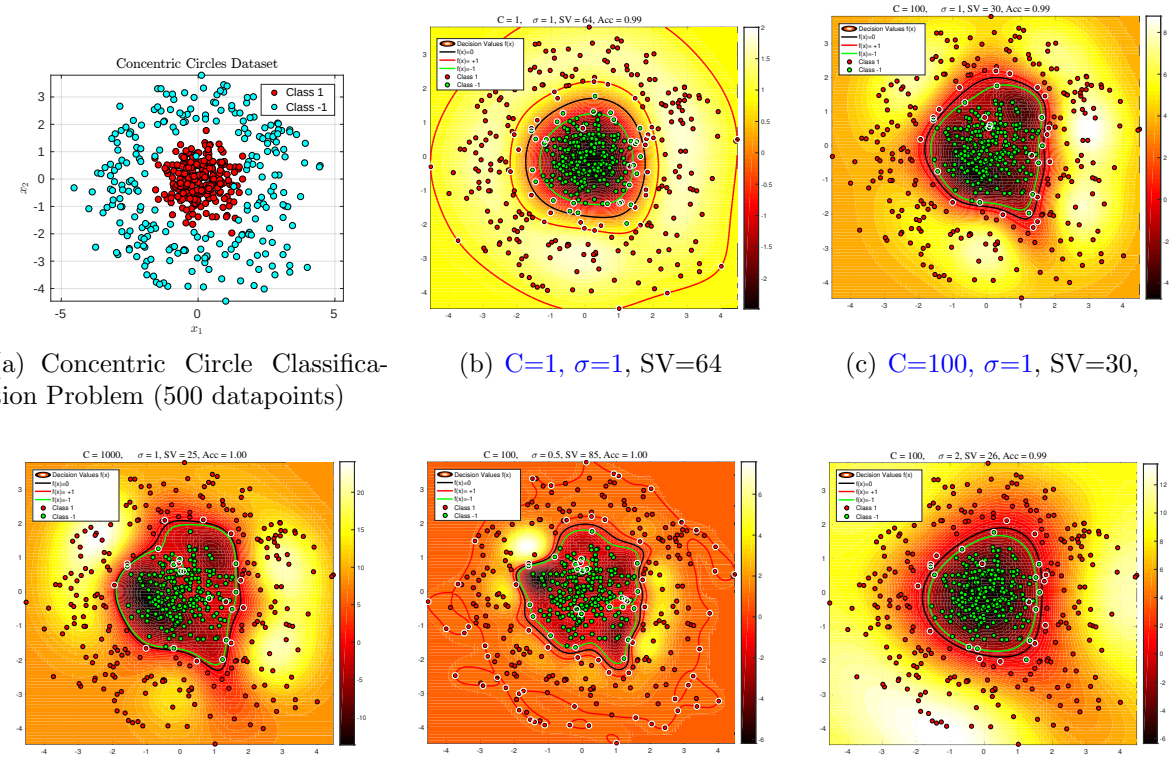
```
%% Train SVM Classifier
[predicted_labels, model] = svm_classifier(X, labels, options, []);

% Plot SVM decision boundary
ml_plot_svm_boundary( X, labels, model, options, 'draw');
```

To visualize a 3D surface mesh for the decision function change '`draw`' to '`surf`'. This will generate the plot in Figure 1(b). Try this yourself and generate the results in Figure 1(c)1(d)1(e)1(f). You will see that for many combinations of C and σ a satisfactory classification is achieved (i.e. $ACC = [0.99, 1]$).

So, how do we find the optimal parameter choices? Manually seeking the optimal combination of these values is quite a tedious task. For this reason, we use grid search on the open parameters. We must choose admissible ranges for these, in the case of RBF it would be for C and σ. Additionally, in order to get statistically relevant results one has to cross-validate with different test/train ratio. A standard way of doing this is applying k-fold Cross Validation for each of the parameter combinations and keeping some statistics such as mean and std. deviation of the accuracy for the k runs of that specific combination of parameters, a typical number of fold is $k = 10$;

Grid Search with 10-fold Cross Validation For the circle dataset we can run C-SVM (RBF kernel) with 10-fold CV over the following range of parameters $C\_\text{range} = [1 : 500]$ and $\sigma\_\text{range} = [0.25 : 3]$, in the same MATLAB script you can specify the steps to partition the range within these limits in code sub-block `2(c)`. This should generate the plots in Figure 2.

```
%% 2c)  Set options for SVM Grid Search and Execute
clear options
options.svm_type    = 0;          % SVM Type (0:C-SVM, 1:nu-SVM)
options.limits_C    = [1, 500];   % Limits of penalty C
options.limits_w    = [0.25, 3]; % Limits of kernel width \sigma
options.steps       = 10;         % Step of parameter grid
options.K           = 10;         % K-fold CV parameter
options.log_grid    = 1;          % log-sampled grid, set 0 for linear
...
```



(a) Concentric Circle Classification Problem (500 datapoints)

(b) C=1, σ=1, SV=64

(c) C=100, σ=1, SV=30,

(d) C=1000, σ=1, SV=25

(e) C=100, σ=0.5, SV=85

(f) C=100, σ=2, SV=26

Figure 1: Decision Boundaries with different hyper-parameter values for the circle dataset. *Support Vector: data-points with white edges.*

The heatmaps in Figure 2 represent the mean *ACC* and *F*-measure on both training and testing sets. To choose the optimal parameters, we normally ignore the metric on the training test and focus on the performance of the hyper-parameters in the testing set. However, when a classifier is showing very poor performance on the testing set, it is useful to analyze the performance on the training set. If the performance on the training set is poor, perhaps the parameter ranges are off, or you are not using the appropriate kernel or your dataset is simply not separable with this method. Notice the option `options.log_grid = 1;` by setting it to 1, this will generate a logged-grid over the parameter ranges you selected. Using a logged-grid is common in machine learning and might be able to explore better the parameter range. By setting `options.log_grid = 0;` this will generate a linear grid within the range of parameters. Try both to see if there's a difference in the optimal parameter regions.

Given the grid search results, how do we choose the optimal parameters? One way of choosing the optimal hyper-parameter combination is to blindly select the iteration which yields the maximum Test *ACC* or *F*-measure. This is already implemented for you in code sub-block `2(c)`, the optimal values (according to the mean test *ACC*) are stored in `C_opt` and `sigma_opt`. For this specific run we achieved a Max. Test *ACC* of 97% with $C = 31.58, \sigma = 3$. We can run code sub-block `2(d)` to learn an *C*-SVM with these optimal parameters and plot the decision boundary, as in Figure 3(a).

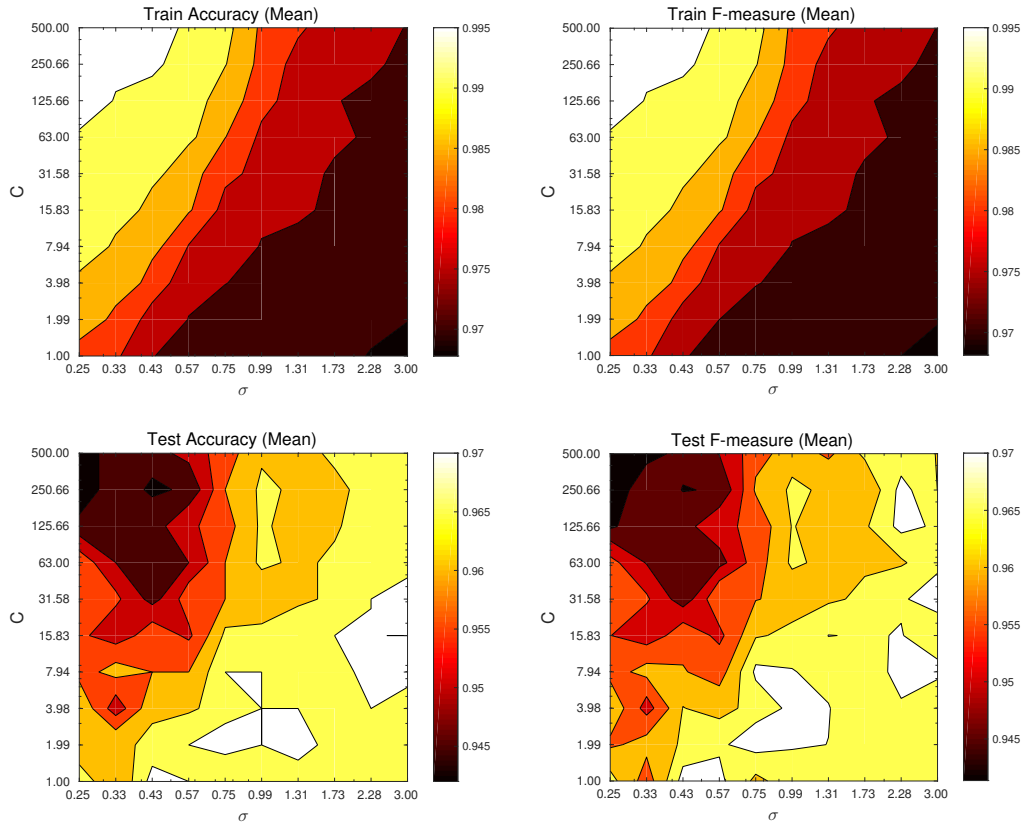
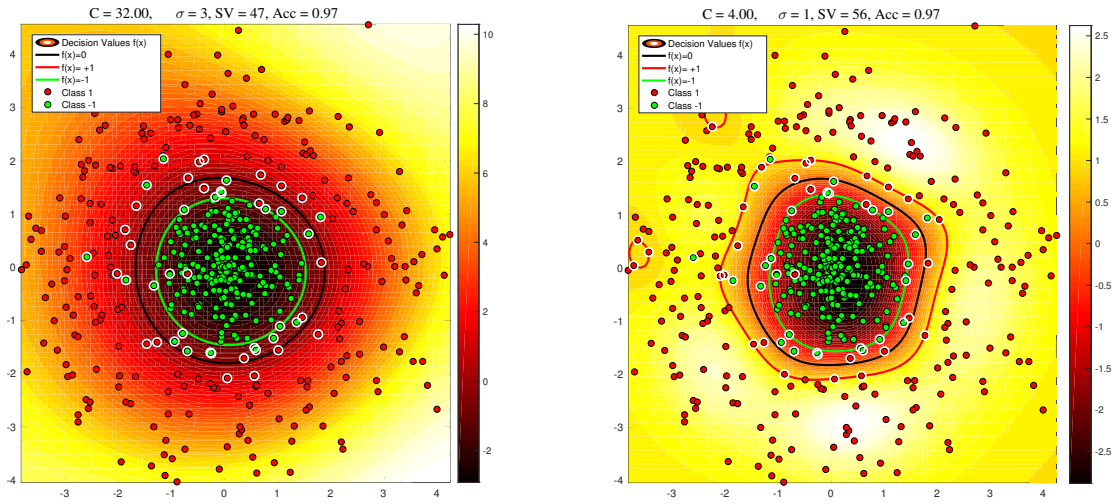


Figure 2: Heatmaps from Grid search on C and σ with 10-fold Cross Validation. (top row) Mean Train Accuracy and $\mathcal{F}-measure$ and (right) Mean Test Accuracy and $\mathcal{F}-measure$

As can be seen, the classifier does recover the circular shape of the real boundary from the dataset. However, this is not the only optimal value. If we take a look back at the heatmap for Test Mean ACC (Figure 2). The max ACC was chosen from the white area on the right-bottom region of the heatmap. We can see that there is another big region with really high accuracy at the bottom-center of the heatmap where both C and σ values are smaller. Intuitively, if σ is smaller, we should have more support vectors, so, let's choose the mid-point of this area which gives us $C = 4, \sigma = 1$, this yields the decision boundary seen in Figure 3(b), it yields the same accuracy, but as we can see, it is a bit more over-fitted to the overlapping points in the boundary. Luckily, in this run we obtained the 'best' model from taking the maximum of the grid search results. Sometimes, this is not the case, and one should analyze the heatmaps.

`NOTE:` It's fine if you can't reproduce exactly the same heatmaps. In fact, due to the random selection of points for the k-folds one will tend to get slightly different results.

Why should we seek for the least number of support vectors? The number of support vectors needed to recover the decision boundary for our classifier is in fact the measure of model complexity for SVMs.

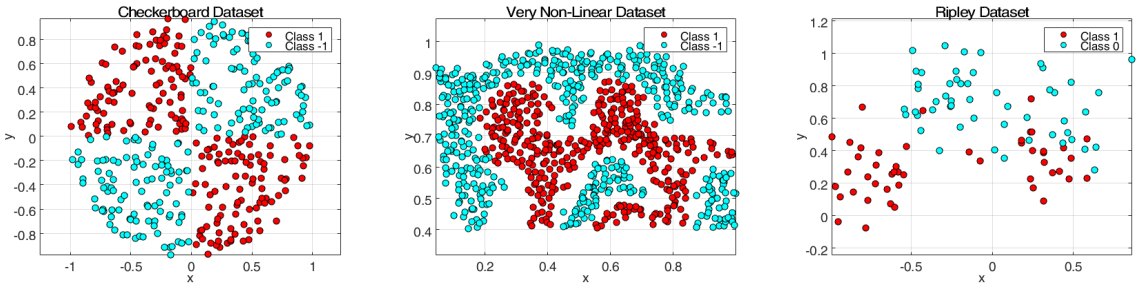


(a) C=32, σ=3, SV=47,ACC=0.97 (b) C=4, σ=1, SV=56,ACC=0.97

Figure 3: C-SVM decision boundary with optimal values from Grid Search with CV.

TASK 1: Find optimal C-SVM parameters for different datasets

- Follow the same evaluation for C-SVM on the different datasets (Figure 4) available in the MATLAB script **TP3_svm.m** by running code sub-blocks `1(b)-(d)`.
- Try out different kernels and evaluate their performance or feasibility depending on the dataset.
- Find the admissible range of parameters for each dataset and for each method.
- Do grid search with CV on the selected parameter ranges.



(a) Checkerboard Dataset
Samples $M = 400$

(b) Very non-linear dataset
Samples $M = 863$

(c) Ripley Dataset
Samples $M = 100$

Figure 4: Datasets for non-linear classification.

4.2 ν-SVM

As can be seen, for a fixed value of σ, the C parameter plays a very important role in the number of support vectors obtained from solving the C-SVM optimization problem (3). To have more control over this interaction, the ν-SVM formulation was proposed. In ν-SVM, rather than setting a fixed value as a penalty (C) for mis-classifications, we use a new variable ρ which controls for the lower bound on $||\mathbf{w}||$ and set the hyper-parameter $\nu \in (0, 1)$ which modulates the effect of ρ on the new objective function:

$$\begin{aligned} &\min_{\mathbf{w}\in\mathcal{H},\xi\in\mathbb{R}^M,\rho\in\mathbb{R}} \left(\frac{1}{2}||\mathbf{w}||^2 - \nu\rho + \frac{1}{M}\sum_{i=1}^{M}\xi_i\right) \\ &\text{s.t.} \quad y_i(\langle \mathbf{w}^T, \mathbf{x}^i\rangle + b) \geq \rho - \xi_i \\ &\xi_i \geq 0, \rho \geq 0, \quad \forall i = 1, \dots M \end{aligned} \tag{7}$$

ν represents an *upper bound* on the fraction of margin error (i.e. number of data-points misclassified in the margin) and a *lower bound* for the ratio of support vector wrt. to total number of data-points $\nu \leq \frac{\#SV}{M}$. Adding these new variables only affects the objective function (Equation 7), the decision function stays the same as for C-SVM (Eq. 5).

`Illustrative example:` To see the effects of the ν on ν-SVM with RBF Kernel, open the accompanying MATLAB script: **TP3_svm.m** and run the first code sub-block `1a` to load the circles dataset shown in Figure 1(a). In code sub-block `3(a)` you can set the desired hyper-parameter values as follows:

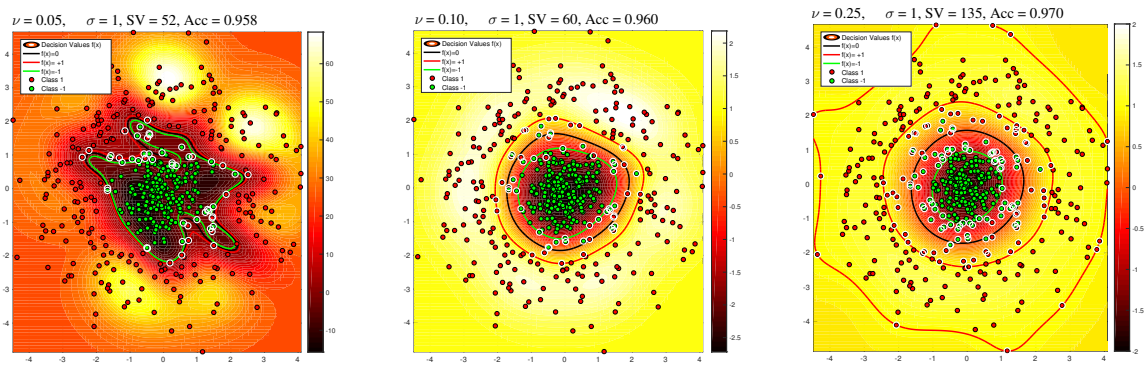
```
%% 3a) EXAMPLE SVM OPTIONS FOR nu-SVM with RBF Kernel%
clear options
options.svm_type    = 1;    % 0: C-SVM, 1: nu-SVM
options.nu          = 0.05; % nu \in (0,1) ...
options.sigma       = 1;    % radial basis function: ...
```

The next code sub-block `3(b)` gives an example of setting the hyper-parameters for a polynomial kernel. After choosing these parameters you can run the train-svm code presented in the following sub-block:

```
%% Train SVM Classifier
[predicted_labels, model] = svm_classifier(X, labels, options, []);

% Plot SVM decision boundary
ml_plot_svm_boundary( X, labels, model, options, 'draw');
```

This will generate the plot in Figure 5(a). Try this yourself and generate the results in Figure 5(b) and 5(c) for the different values of ν with fixed $\sigma = 1$. As can be seen, the larger the ν, the more support vectors. To find the optimal combination of ν and the kernel hyper-parameter, in this case σ, we must apply grid search on a range of hyper-parameters.

As can be seen in Figure 6, there are certain combinations which yield very poor performance. For this reason, all the values above 0.9 are shaded as white. By selecting the combination of $\sigma = 2.014$ and $\nu = 0.32$, we can achieve a maximum mean $ACC = 0.966$. The decision boundaries of these optimal parameters are presented in Figure 7(a). Because of the high value of ν, the optimization algorithm was able to capture an almost



(a) ν=0.05, σ=1, SV=52 (b) ν=0.1, σ=1, SV=60 (c) ν=0.25, σ=1, SV=135

Figure 5: Decision Boundaries with different hyper-parameter values for ν-SVM on the circle dataset. *Support Vector: data-points with white edges.*

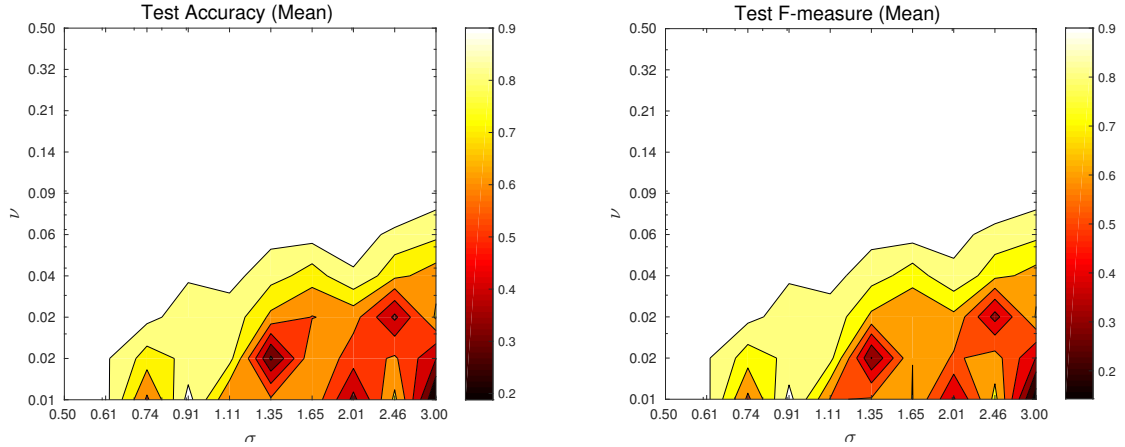
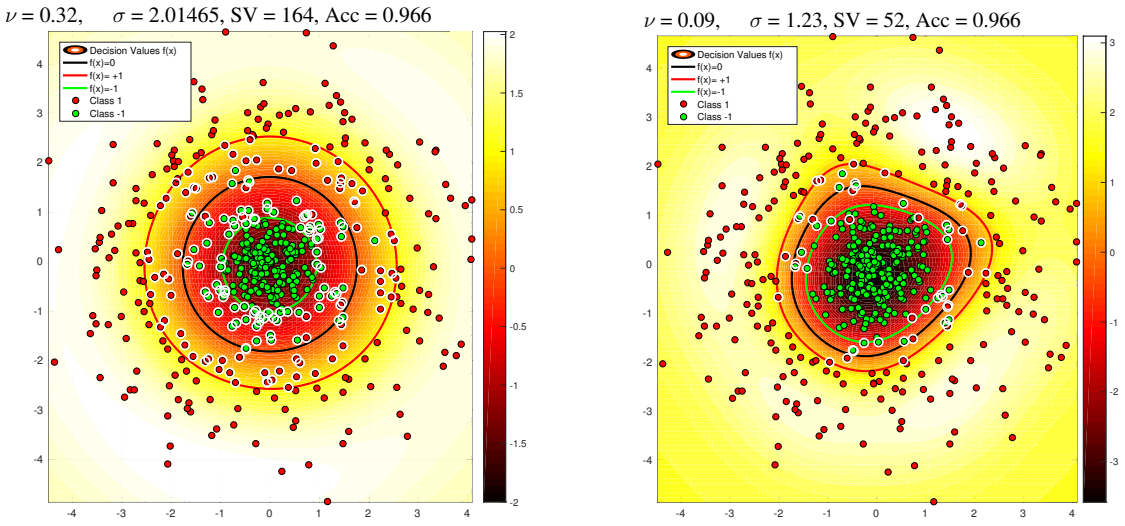


Figure 6: Heatmaps from Grid search on ν and σ with 10-fold Cross Validation. (left) Mean Test Accuracy and (right)Mean Test $\mathcal{F}-measure$

ideal margin, however, the number of support vectors is quite high. If we look deep into the heatmap, we can see that we can achieve the same accuracy with a lower value of $\nu = 0.09$ and $\sigma = 1.23$, which yields the decision boundary in Figure 7(b). The class decision boundary is not a perfect circle anymore, however, it generates a tighter margin with much fewer support vectors.

TASK 2: Find optimal ν-SVM parameters for different datasets

- Follow the same evaluation for ν-SVM on the different datasets (Figure 4) available in the MATLAB script **TP3_svm.m** by running code sub-blocks `1(b)-(d)`.
- Try out different kernels and evaluate their performance or feasibility depending on the dataset.
- Find the admissible range of parameters for each dataset and for each method.
- Do grid search with CV on the selected parameter ranges.



(a) ν=0.32, σ=2.014, SV=164,ACC=0.97 (b) ν=0.09, σ=1.23, SV=52,ACC=0.97

Figure 7: ν-SVM decision boundary with optimal values from Grid Search with CV.

4.3 Relevance Vector Machines (RVM)

Another non-linear classification algorithm, that can be seen as a variant of SVM with the goal of optimizing for the least number of support vectors as possible, is the Relevance Vector Machine (RVM). The RVM applies the Bayesian 'Automatic Relevance Determination' (ARD) methodology to linear kernel models, which have a very similar formulation to the SVM, hence, it is considered as a *sparse SVM*. When predictive models are linear in parameters, ARD can be used to infer a flexible, non-linear, predictive model which is both accurate and uses a very small number of relevant basis functions (in our case support vectors) from a large initial set. The predictive linear model for RVM has the following form:

$$y(\mathbf{x}) = \sum_{i=1}^{M} \alpha_i \, k(\mathbf{x}, \mathbf{x}^i) = \alpha^T \Psi(\mathbf{x}) \tag{8}$$

where $\Psi(\mathbf{x}) = [\Psi_0(\mathbf{x}), \ldots, \Psi_M(\mathbf{x}),]^T$ is a linear combination of basis functions $\Psi(\mathbf{x}) = k(\mathbf{x}, \mathbf{x}^i)$, α is a sparse vector of weights and $y(\mathbf{x})$ is the binary classification decision function $y(\mathbf{x}) \rightarrow \{1, 0\}$. The problem in RVM is now to find a sparse solution for α, where α_i is non-zero when the datapoint is a relevant 'support' vector and zero otherwise. Finding such a sparse solution is not trivial. ARD is a method tailor-made to discover the relevance of parameters for a predictive model by imposing a prior on the parameter whose relevance needs to be determined. When attempting to calculate α we take a Bayesian approach and assume that all output labels y^i are i.i.d samples from a true model $y(\mathbf{x^i})$ (Equation 8) subject to some Gaussian noise with zero-mean:

$$\begin{aligned} y^i(\mathbf{x}) &= y(\mathbf{x}^i) + \epsilon \\ &= \alpha^T \Psi(\mathbf{x}^i) + \epsilon \end{aligned} \tag{9}$$

where $\epsilon \propto \mathcal{N}(0, \sigma_\epsilon^2)$. We can then estimate the probability of a label y^i given x with a Gaussian distribution with mean on $y(\mathbf{x^i})$ and variance σ_ϵ^2, this is equivalent to:

$$\begin{aligned} p(y^i|\mathbf{x}) &= \mathcal{N}(y^i|y(\mathbf{x^i}), \sigma_\epsilon^2) \\ &= (2\pi\sigma_\epsilon^2)^{-\frac{1}{2}} \exp\left\{-\frac{1}{2\sigma_\epsilon^2}(y^i - y(\mathbf{x}^i)^2\right\} \\ &= (2\pi\sigma_\epsilon^2)^{-\frac{1}{2}} \exp\left\{-\frac{1}{2\sigma_\epsilon^2}(y^i - \alpha^T\Psi(\mathbf{x}^i))^2\right\} \end{aligned} \tag{10}$$

Thanks to the independence assumption we can then estimate the likelihood of the complete dataset with the following form:

$$p(\mathbf{y}|\mathbf{x}, \sigma_\epsilon^2) = (2\pi\sigma_\epsilon^2)^{-\frac{M}{2}} \exp\left\{-\frac{1}{2\sigma_\epsilon^2}||\mathbf{y} - \alpha^T\Psi(\mathbf{x})||^2\right\} \tag{11}$$

We could then approximate α and σ_ϵ^2 through MLE (Maximum Likelihood Estimation), however, this would not yield a sparse vector α, instead it would give us an over-fitting model, setting each point as a relevant basis function. To promote sparsity on the solution, we define an explicit prior probability distribution over α, namely a zero-mean Gaussian:

$$p(\alpha|\mathbf{a}) = \prod_{i=0}^{M} \mathcal{N}(\alpha^i|0, a_i^{-1}) \tag{12}$$

where $\mathbf{a} = [a^0, \ldots, a^M]$ is a vector of parameters a^i, each corresponding to an independent weight α^i indicating it's strength (or *relevance*). For the binary classification problem, rather than predicting a discriminative membership of a class $y \to \{1, 0\}$, in RVM we predict the posterior probability $P(y|\mathbf{x})$ of one class $y \in \{1, 0\}$ given input $\mathbf{x}$ by generalizing the linear model from Equation 8 with the logistic sigmoid function $\sigma(y)$ of the form:

$$\sigma(y) = \frac{1}{1 + \exp\{-y\}} \tag{13}$$

The Bernoulli distribution is used for $P(y|\mathbf{x})$ and the likelihood then becomes:

$$P(\mathbf{y}|\alpha) = \prod_{i=1}^{M} \sigma\{y(\mathbf{x}^i)\}^{y^i}[1 - \sigma\{y(\mathbf{x}^i)\}]^{1-y^i} \tag{14}$$

Now, following Bayesian inference, to learn the unknown parameters α, a we start with the decomposed posterior[3] over unknowns given the data:

$$p(\alpha, \mathbf{a}, |\mathbf{y}) = p(\alpha|\mathbf{y}, \mathbf{a})p(\mathbf{a}|\mathbf{y}) \tag{15}$$

where unfortunately the posterior distribution over the weights $p(\alpha|\mathbf{y}, \mathbf{a})$ and the hyperparameter posterior $p(\mathbf{a}|\mathbf{y})$ are not analytically solvable. It can however be approximated

[3]Refer to the Tipping's paper on RVM to understand how this and subsequent terms were derived. Tipping, M. E. (2001). Sparse Bayesian learning and the relevance vector machine. Journal of Machine Learning Research 1, 211–244.

by using Laplace's method[4]. Estimating for α then becomes a problem of maximizing the weight posterior distribution $p(\alpha|\mathbf{y}, \mathbf{a})$, and since $p(\alpha|\mathbf{y}, \mathbf{a}) \propto P(\mathbf{y}|\alpha)p(\alpha|\mathbf{a})$ (which are Equations 14 and 12 corresponding to the likelihood and prior of the weights α) this is equivalent to maximizing the following equation over α:

$$\log\{P(\mathbf{y}|\alpha)p(\alpha|\mathbf{a})\} = \sum_{i=1}^{M}[y^i \log \sigma\{y(\mathbf{x}^i)\} + (1 - y^i)\log(1 - \sigma\{y(\mathbf{x}^i)\})] - \frac{1}{2}\alpha^T \mathbf{A}\alpha \quad (16)$$

where $\mathbf{A} = diag(a^0, \ldots, a^M)$. This is optimized in an iterative procedure, a is updated in each step following some derivations from Tipping. Once we have learned our parameters, Equation 14 is then used to estimate probabilities for each class, in the binary case when $p(y|\mathbf{x}) = 0.5$ this corresponds to the decision boundary, values $p(y|\mathbf{x}) > 0.5$ correspond to the positive class and consequently $p(y|\mathbf{x}) < 0.5$ to the negative class. Contrary to SVM, the basis functions for the linear model $\Psi_i(\mathbf{x})$ are not restricted to positive-definite kernels. Nevertheless, in this tutorial we will consider only the RBF kernel for the RVM.

`Illustrative Example:` To see the effects of σ on RVM with RBF kernel, open the accompanying MATLAB script: **TP3_rvm.m** and run the first code sub-block `1(a)` to load the circles dataset used in the previous examples. In code sub-block `2(a)` you can set the desired hyper-parameter values as follows:

```
%% 4a) Try RVM on Data
%Set RVM OPTIONS%
clear rvm_options
rvm_options.useBias = true;
rvm_options.kernel_ = 'gauss';
rvm_options.width   = 0.25;
rvm_options.maxIts  = 100;

% Train RVM Classifier
[predict_labels, model] = rvm_classifier(X, labels , rvm_options, []);
```

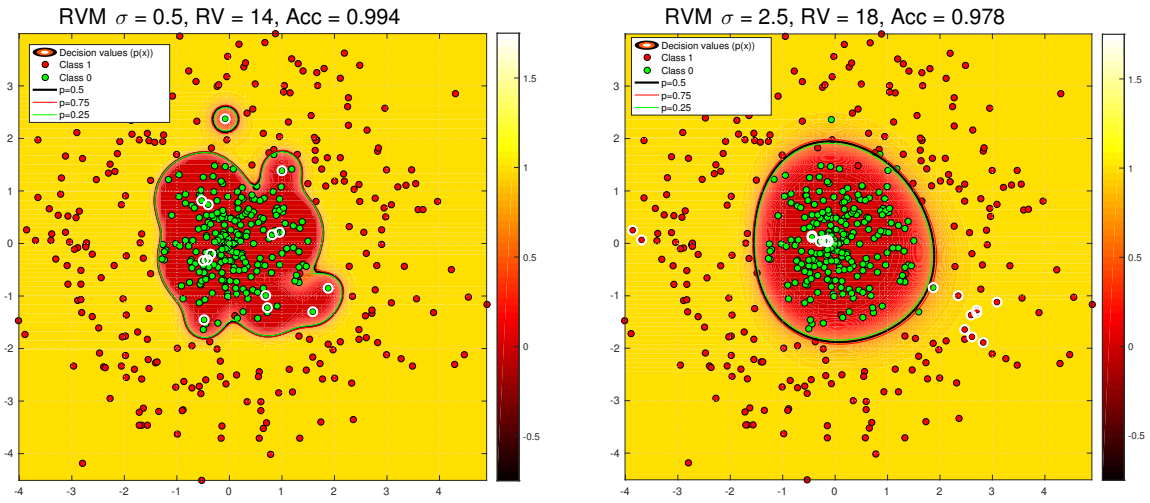
The variable `maxIts` sets the maximum number of iterations for the iterative optimization algorithm used to estimate the parameters of the RVM model. In Figure 8(a) and 8(b) we can see the decision boundaries generated by learning RVMs with different values of σ. As in the previous methods, we can use grid search with cross-validation to find the optimal parameter. By running code sub-block `2(b)` we can do this grid search for RVM with RBF kernel by setting the range of sigma as follows:

```
%% 4b) Do Grid Search on RVM
clear options
% Set Options for RVM Grid Search
options.limits_w   = [0.5, 5]; % Limits of kernel width \sigma
options.steps      = 10;       % Step of parameter grid
options.K          = 5;        % K-fold CV parameter
...
```

Note that in this case, we are using $k = 5$ as the optimization for RVM takes longer for each iteration, by increasing $k = 10$ you may get similar results, perhaps with less

[4]Refer to Tipping paper for Laplace's Approximation.

variance on the performance. For this dataset, grid search with cross-validation results in the plots in Figure 9, where we can visualize both mean and std. deviation for the k folds. From these curves we can infer that the optimal σ is 2.32, which gives a very close result to that of Figure 8(b).



(a) σ=0.5, RV=14, ACC=0.994 (b) σ=2.5, RV=18, ACC=0.978

Figure 8: RVM with different values of σ for the RBF Kernel.

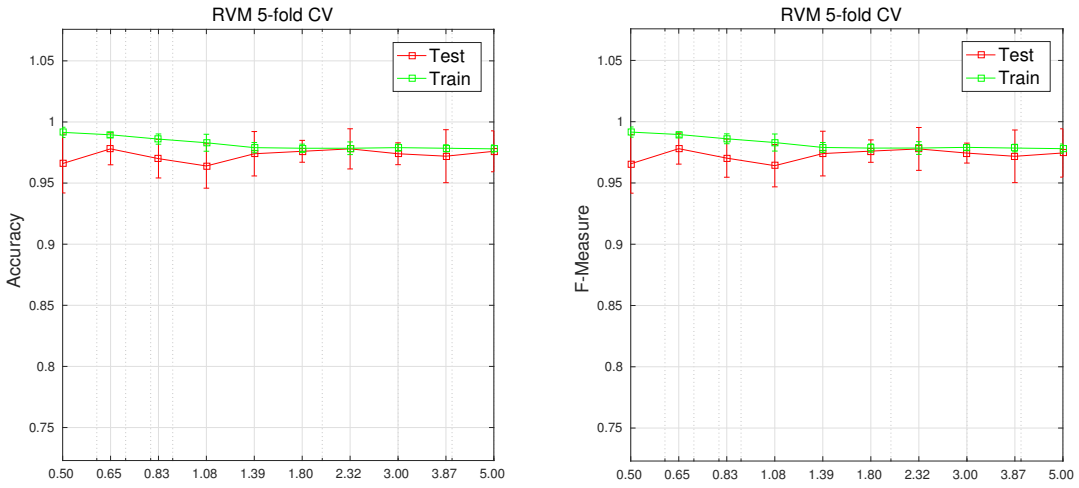


Figure 9: Plots from Grid search on σ with 5-fold Cross Validation on *Circles* dataset. (left) Mean/Std.dev. Test and Train Accuracy and (right) Mean/Std.dev. Test and Train $\mathcal{F} - measure$

TASK 3: Find optimal RVM parameters for different datasets

- Follow the same evaluation as for C-SVM and ν-SVM on the different datasets (Figure 4) available in the MATLAB script **TP3_rvm.m** by running code sub-blocks `1(b)-(d)`.
- Only the RBF Kernel is available in ML_toolbox, find the admissible range of σ.
- Do grid search with CV on the selected parameter ranges.

`HINT`: The estimation of the RVM parameters tends to give numerical errors when the Kernel matrix becomes ill-conditioned. This may happen for the RBF kernel when σ is set too high or too low. If this is the case, the following message will show up in the MATLAB command window and nothing will be estimated:

```
** (PosteriorMode) warning: ill-conditioned Hessian (1.87281e-16)
** Giving up!
** This error was probably caused by an ill-conditioned kernel matrix.
** Try adjusting (reducing) the length scale (width) parameter of ...
   the kernel
```

If you don't take this into consideration and continue with doing grid search on a ill-fitted range, the performance metrics (i.e. ACC and F-measure) for these σ values will give zero values.

4.4 AdaBoost

The AdaBoost algorithm iteratively builds a *strong* classifier, $C(\mathbf{x}) \rightarrow \{-1, +1\}$, through a weighted linear combination of simple, also known as *weak* classifiers $\phi(\mathbf{x}) \rightarrow \{-1, +1\}$. The original formulation of AdaBoost considers binary classification problems. There are extensions such as to be able to handle multi-class problems, but in this tutorial we will be focusing on the binary case. Given a data set $(\mathbf{x}^1, y_1), \ldots, (\mathbf{x}^M, y_M)$ of M samples in which the class label $y_i \in \{-1, +1\}$ is either positive or negative and a set of T *weak* classifiers $\phi_1(\mathbf{x}), \ldots, \phi_T(\mathbf{x})$, we seek to learn the strong classifier $C(\mathbf{x})$, Equation 17,

$$C(\mathbf{x}) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t \, \phi_t(\mathbf{x})\right), \;\; \alpha_i \in R \tag{17}$$

In theory you can use any classifier for $\phi_t(\mathbf{x}_i)$, but usually it should be very simple and cheap to compute. In this practical we will consider **decision stumps** as our weak classifier, Equation 18:

$$\phi_t(\mathbf{x}; \theta) = \begin{cases} +1 & \text{if } \mathbf{x}_{\theta_1} > \theta_2 \\ -1 & \text{otherwise} \end{cases} \tag{18}$$

A decision stump is a function which separates the classes along a dimension. It has two parameters $\theta = \{\theta_1, \theta_2\}$. The first parameter indicates in which dimension the decision boundary will be placed and the second parameter decides where along this dimension does the split occur. The Adaboost algorithm begins at iteration $t = 1$ with an equally distributed set of weights for all training samples; $D_{t=1}(i) = 1/M$ for $i = 1, \ldots, M$. Then, the following steps are iteratively performed for $t = 1, ..., T$ *weak* classifiers [1]:

1. The t-th *weak* classifier $\phi_t(\mathbf{x}; \theta_t, D_t)$ tries to find the best threshold in one of the data dimensions to separate the data into two classes -1 and 1, which gives a classification error of:
$$\epsilon_t = \sum_{i=1}^{M} \delta_{[\phi(\mathbf{x}^i; \theta_t, D_t) \neq y_i]} \tag{19}$$

2. The influence of this *weak* classifier on the total result, α_t, is computed through the following weighting function:
$$\alpha_t = \frac{1}{2} \ln\left(\frac{1 - \epsilon_t}{\epsilon_t}\right) \tag{20}$$

3. Training sample weights D_{t+1} are update such that wrongly classified samples will have more weight,
$$D_{t+1}(i) = \frac{D_t \exp(-\alpha_t y_i \phi(\mathbf{x}^i; \theta_t, D_t))}{\sum_{i=1}^{M} D_t \exp(-\alpha_t y_i \phi(\mathbf{x}^i; \theta_t, D_t))} \tag{21}$$

`Illustrative Example:` To understand the mechanism of building a *strong* non-linear classifier from a set of *weak* linear classifiers, we will work on the circles dataset used in the previous examples. Open the accompanying MATLAB script: **TP3_adaboost.m** and run the first code sub-block `1(a)` to load the circles dataset. In code sub-block `2(a)` you can set the desired number of *weak* classifiers T as follows:

```
%% 2a) Ada-boost
N_weak = 1;
[classestimate,model,D]= adaboost_classifier(X,labels,N_weak,[]);
f                      = @(X)adaboost_classifier(X,[],[],model);
```

This is the sole hyper-parameter one must tune for Adaboost. By setting it to different values; i.e. $T = [1, 3, 5, 10, 30]$ we can see that different decision boundaries are achieved. In Figure 10(a)-10(d) the result of AdaBoost when only one weak classifier is used; we clearly see the decision stump.

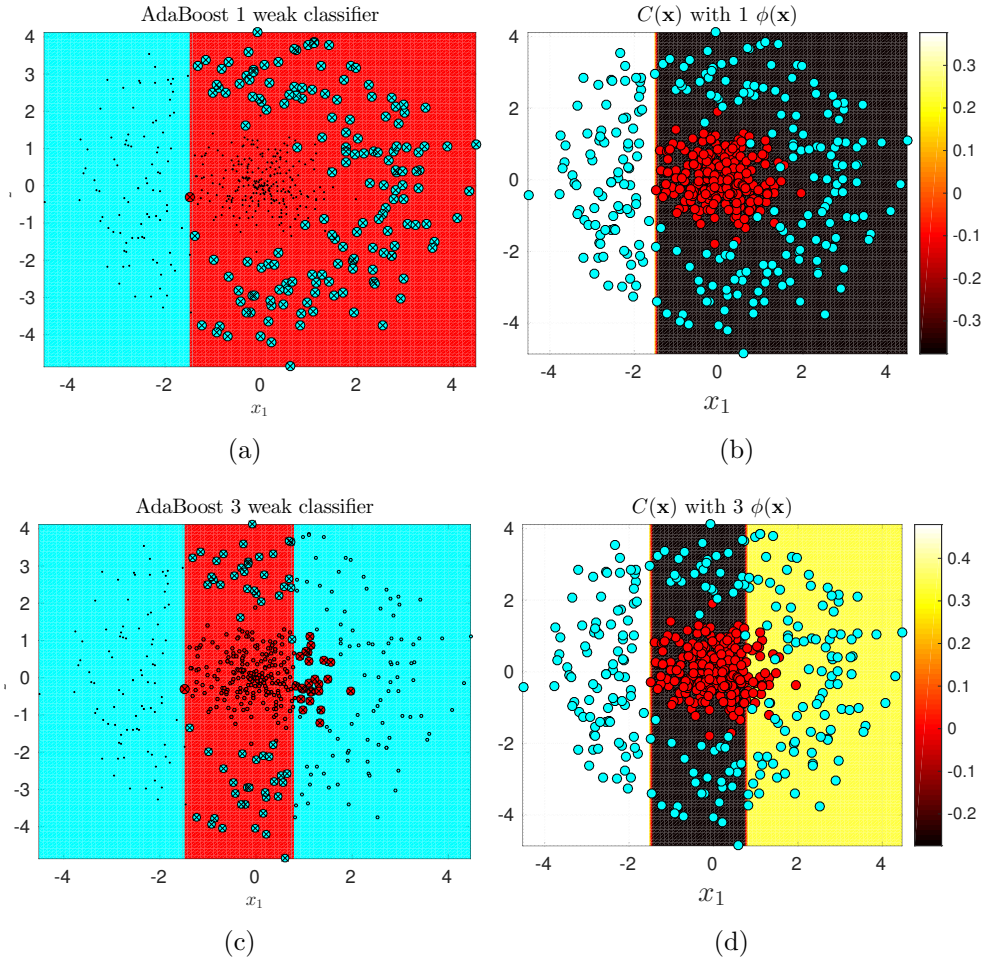


Figure 10: AdaBoost applied to the circle data set. Width of circles on the *Left column* are proportional to the classification error and the crosses depict the points which have been misclassified. In the *Right column*, the value is of the strong classifier function $C(\mathbf{x})$ before the sign operator is applied.

The figures on the right illustrate the non-signed output of Equation 17 and the figure on the left is the result after taking the sign. Points which are misclassified have a cross on them, but in addition you will see that the width of the data points have changed (*Left column*). The **width of a data point** is proportional to the weight given by the AdaBoost algorithm. When $T = 3$, the number of weak classifiers goes from one to three. One can see how the second and third decision stumps were trained on the **new weighted**

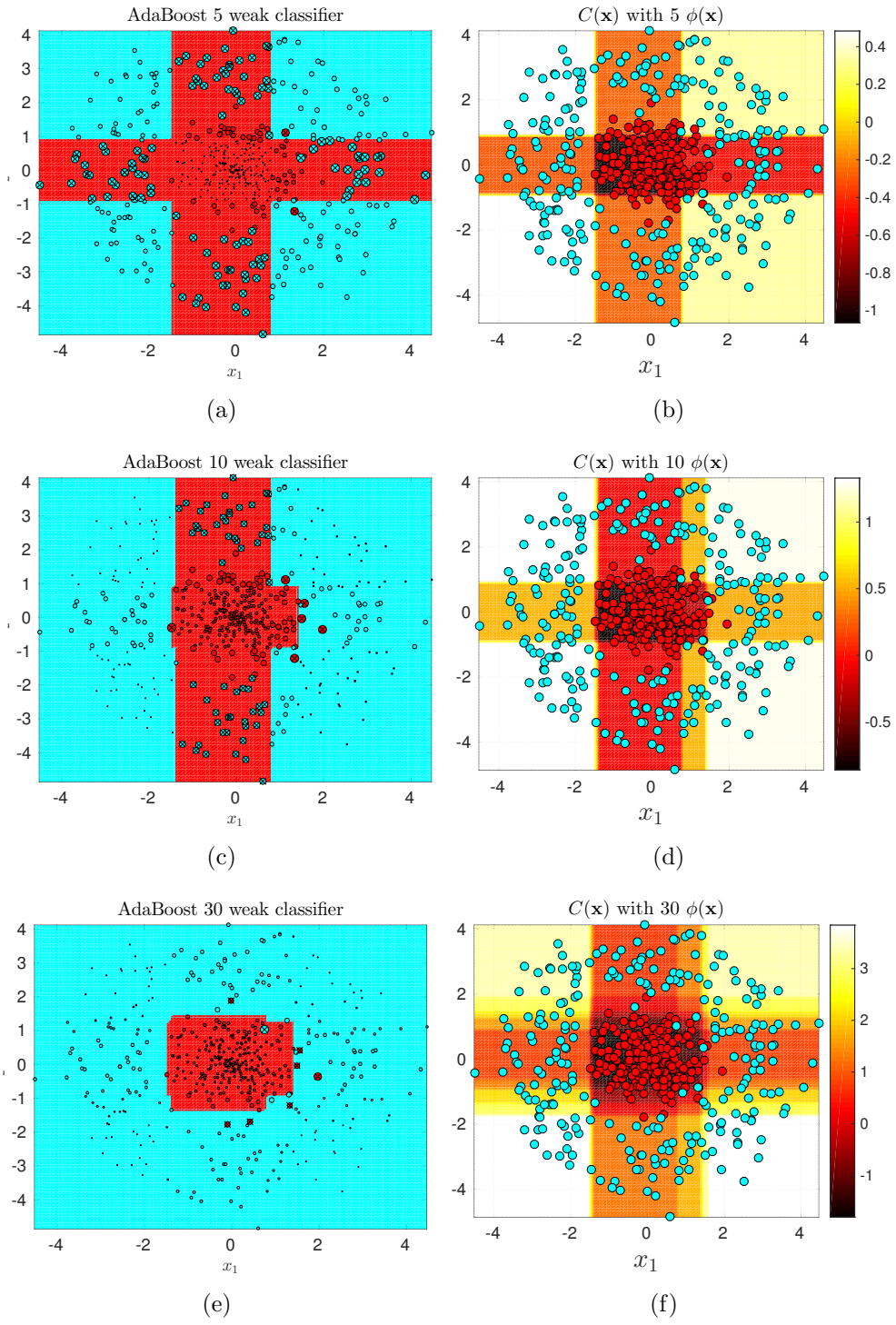


Figure 11: AdaBoost applied to the circle data set. Width of circles on the *Left column* are proportional to the classification error and the crosses depict the points which have been misclassified. In the *Right column*, the value is of the strong classifier function $C(\mathbf{x})$ before the sign operator is applied.

dataset from the previous iteration. By increasing the number of weak classifiers, we can see that a nearly perfect classification is achieved, see Figure 11(a)-11(f).

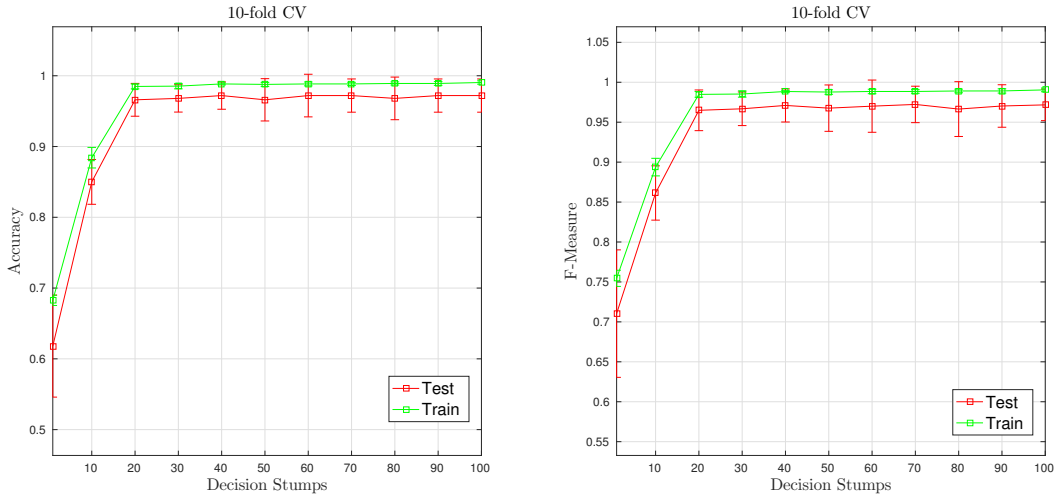


Figure 12: AdaBoost, 10-fold CV over an increasing number of weak classifiers; for the circle data set. The **accuracy** for both the train (green) and test (red) data sets are shown.

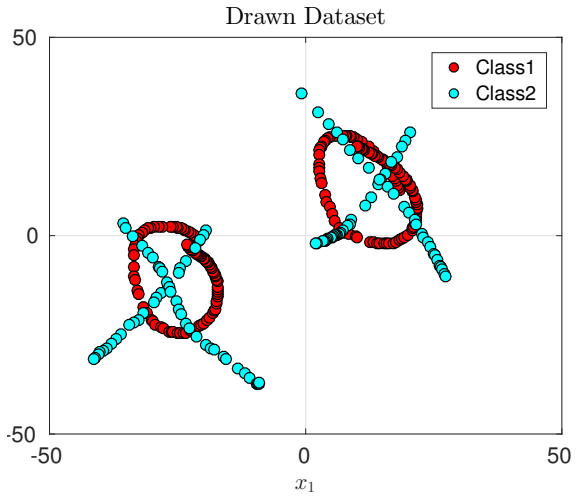
Of course, finding the optimal number of *weak* classifiers like this is a tedious task. Hence, we can apply the good ol' grid search with cross-validation to find the optimal T. In order to do so, we can run code sub-block `2(b)` which should generate the plots in Figure 12. You can see that the test error after 20 decision stumps has settled at around a 96.6% accuracy.

TASK 4: Find the optimal number of weak classifiers for each dataset and compare performance with other methods

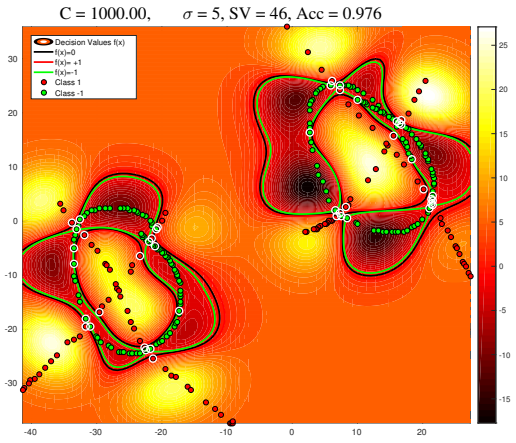
- Follow the same evaluation procedure as for SVM on the different datasets available (Figure 4) in the MATLAB script **TP3_adaboost.m**.
- Compare the performance of Adaboost to the previously seen SVM variants and RVM, in terms of: accuracy/F-measure, training efficiency and model complexity. Which algorithm is best suited for each dataset?
- How do all these methods compare when we have datasets with overlapping, noisy, or unbalanced classes? Try generating your own datasets with the drawing GUI in ML_toolbox. In each corresponding MATLAB script: **TP3_svm.m**, **TP3_rvm.m** and **TP3_adaboost.m**, the following code-sublock `1(e)` is provided so that you can draw your own datasets and convert them to a binary classification task:

```
%% 1e) Draw Data with ML_toolbox GUI
clear all; close all; clc;
ml_draw_data
[N,M] = size(X);
X      = X';
labels  = ml_2binary(labels)';
```

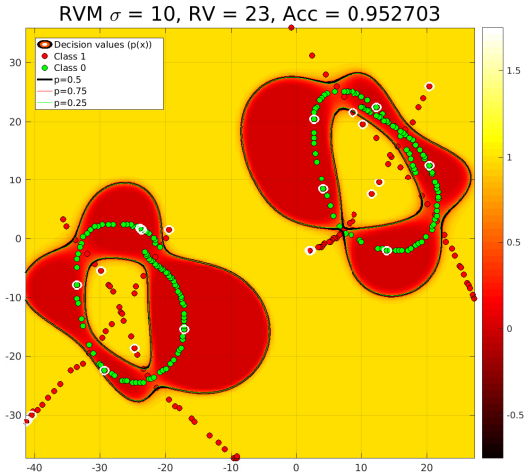
Then we can apply each method on the dataset as shown in Figure 13.



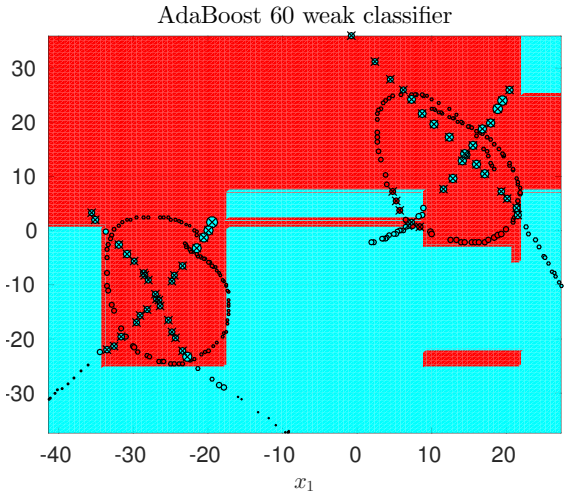
(a) Drawn Dataset



(b) C-SVM Decision Boundary



(c) RVM Decision Boundary



(d) Adaboost Decision Boundary

Figure 13: Dataset with overlapping classes and the decision boundaries obtained by the different algorithms.

References

[1] Robert E. Schapire. Explaining adaboost.