datahack

# SVM models: classification, regression and density estimation

## 1. Preliminary instructions

This document provides you with explanations and instructions on how to carry out the practical assignment. Throughout the text you will find the following icons with special instructions:

**Question icons** are used for questions you must answer or tasks you are asked to perform. **Your mark will depend on how you tackle all these.**

**Exclamation icons** give you suggestions or hints to help you answer the compulsory questions.


**Pro icons** suggest additional tasks that are not strictly part of the assignment, but can give you a broader view of the topic. **Completing these successfully will give you extra points.**

For this practical assignment, it is assumed that you have already installed in your computer the following:

- *Python 3.5*
- The following Python packages and their dependencies:
  - *scikit-learn*
  - *numpy*
  - *matplotlib*
  - *tkinter*

It is strongly advised, instead of installing all the above separately, to **install the *Anaconda* distribution of *Python 3.5***, which already includes all these packages. Please note that **using other versions of Python (e.g. *Python 2.7*) may require manual changes in the code**.

Make sure as well **you have all the additional material required for this assignment (datasets, source code...) available in your PC** before proceeding.

Once you have installed all the above and located the source code provided, try to run the script named `svm_gui.py`. Depending on your personal preferences, you can execute the script directly in a terminal or through a Python IDE such as *Spyder* (included in Anaconda) or *PyDev*.

A window with a blank canvas and a set of controls will pop up, similar to the following:

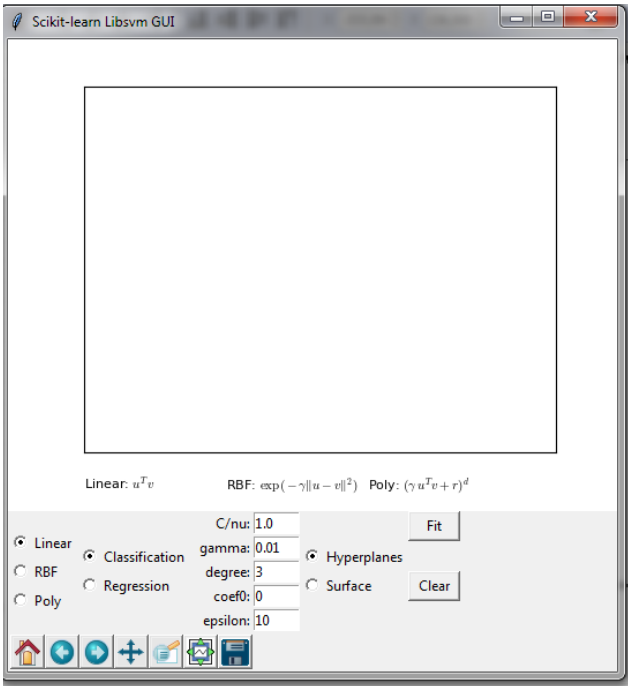


Figure 1: Window expected to appear after executing `svm_gui.py`.

If you are using *Anaconda*, **it has been observed in some computers that the execution crashes on Windows**. If that is the case, try to do the following:

1. Uninstall *matplotlib* with the command `conda uninstall matplotlib`.
2. Download the corresponding binary package (whl file) from `http://www.lfd.uci.edu/~gohlke/pythonlibs/`, which will depend on your computer (32/64 bits, Python version...).
3. Install the package with the command `pip install <name of whl file>`.
4. Repeat the previous 3 steps with the package *pillow*.
5. Run again the script `svm_gui.py`.

# 2. Classification

## 2.1 Linear SVMs

We will start by studying linear SVM models. The window in Figure 1 allows you to draw 2–dimensional datasets at will, and test different SVM models to check their performance. You can add positive class points to the canvas by right–clicking on the desired position, and negative class points by left–clicking.

Create a linearly separable dataset in this way, with a positive class and a negative class clusters similar to what is shown in Figure 2. Now click the the `Fit` button on the bottom control bar. A linear SVM model will be trained and

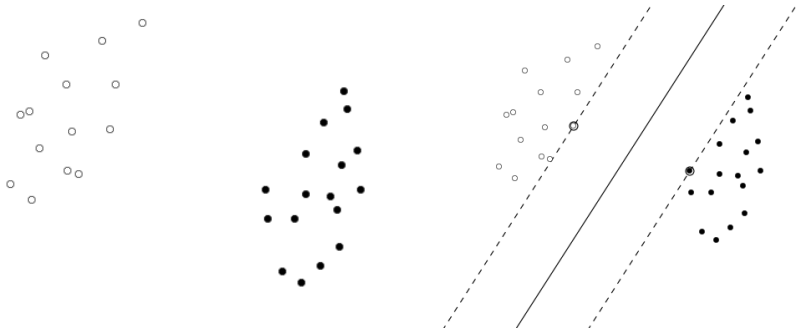

Figure 2: Plot of a linearly separable dataset and a linear SVM model over that dataset.

represented over the dataset. The solid line is the decision hyperplane of the SVM, while the dashed lines represent the margins (support hyperplanes). Any data point with a surrounding circle is a support vector of the trained model.

Compare your model with the one shown in Figure 2. How many SVs does your model have? Why are those very points the SVs? What is their relationship with the margins?

Recall from the theory that the KKT conditions tell which points are chosen as SVs and which are not.

Try adding new points to each cluster in different positions (keeping the classes separable). When do the support vectors change? Why is that?

Recall that the model is totally determined by the SVs. The rest of points do not have any influence.

Now add a new point on top of the decision hyperplane. A new decision hyperplane will appear between this new point and the opposite class, and the margins will shrink as shown in the left part of Figure 3. The SVM model is trying to achieve perfect classification accuracy on the data, sacrificing margin in order to do so.

However, this behavior might be undesirable if the new point is actually a noisy data point. In such settings we would be interested in preserving a large margin as shown in the right part of Figure 3, even if we misclassify such point. Such preferences can be enforced into the SVM by tuning its $C$ parameter.

Change the value of the $C$ parameter in the control bar and see how the SVM model changes (you will need to press the `Fit` button after each change in the value of $C$). Which value produces a margin similar to the one you obtained before adding the noisy point to the dataset? Are there any differences in the SVs? Why/why not?

## 2.2 Non–linear SVMs

The dataset we just built is linearly separable, so that perfect classification can be achieved with a linear SVM. Nevertheless, this will rarely be the case in real-life scenarios. We learnt that SVMs are able to separate non–linear datasets as well after the inclusion of a kernel function.

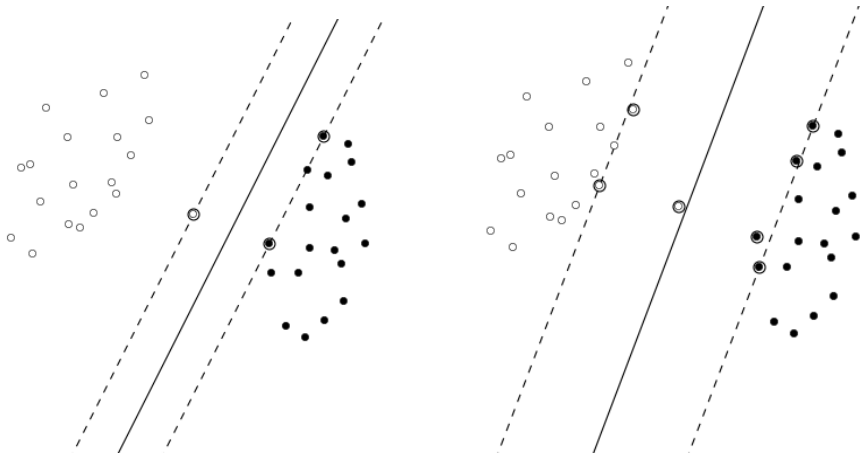

Figure 3: Different SVM models obtained by changing $C$ for a linearly separable dataset with a noisy point.

To test this, we will build a new dataset on which linear models can only achieve poor results. Click on the `Clear` button to remove all data points from the canvas, and then draw a dataset similar to the one shown in Figure 4.

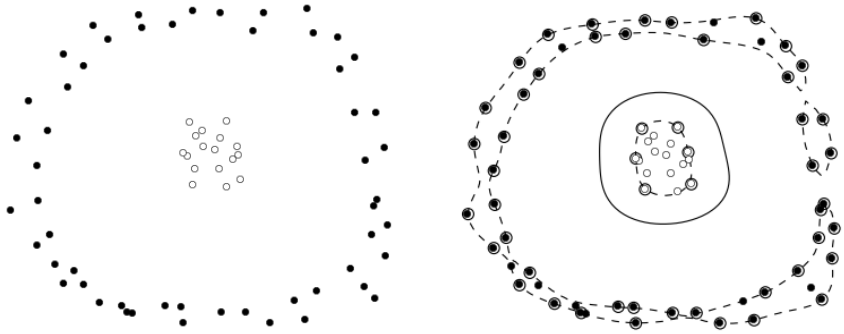

Figure 4: Plot of a non–linear dataset and a Gaussian kernel (RBF) SVM model trained with it.

Train a linear SVM over the dataset. Do you think the results are good? Now select the `RBF` kernel in the left–hand side of the control bar. The model will change to a non–linear Gaussian (RBF) kernel SVM. Does the separating frontier look more appropriate now?

You can read the accuracy of the models being constructed as a text output in the terminal from which the `svm_gui.py` script was launched. You can use this to compare the suitability of the linear and non–linear SVM models for this dataset.

Can you make the RBF SVM reach 100% accuracy? What parameter values are the best you can find? The polynomial kernel is also available in the application. Try using it instead of the RBF kernel and compare the results.

$C$ penalizes the errors made by the model. If the accuracy is low, it means that there are too many errors.

## 2.3 Tuning SVM parameters

As the previous example illustrated, better accuracy results can be obtained by fine–tuning the penalty parameter $C$ and the kernel parameters. This tuning, however, has to be done carefully to avoid underfitting or overfitting effects. To better appreciate the impact of the parameters in the model, add some more class points to the external ring, as well as a few points from the opposite class, as has been made in Figure 5.

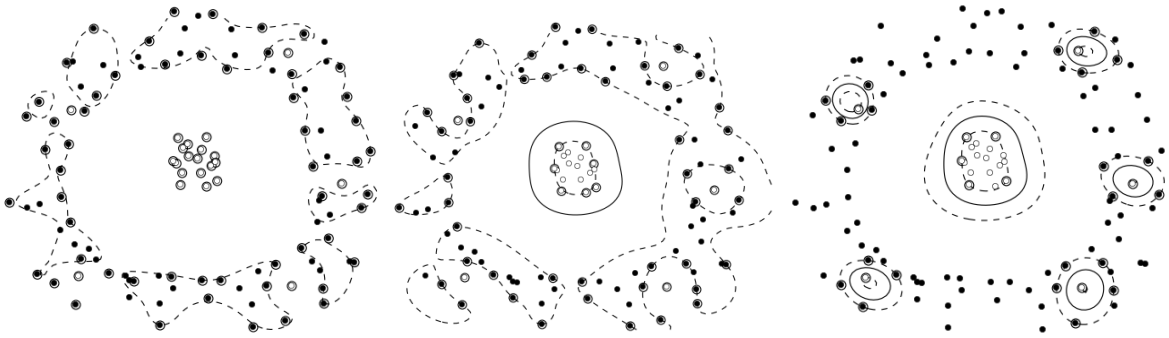

Figure 5: Plots of 3 RBF SVM models over a non–linear dataset with noise, for varying values of the $C$ parameter. The left plot presents a case of underfitting, while the right plot show overfitting. A model with a correct value of $C$ is shown in the center plot.

Train a model with a small $C$ value (0.01 or smaller), and observe what happens with the decision frontier. Do the same for a large $C$ value (100 or larger). Which of these cases corresponds to overfitting and which to underfitting? Why is this happening?

Moving now to the kernel parameters, since we are using the Gaussian (RBF) kernel, there is a single parameter $\gamma$ to tune, which corresponds to the Gaussian width, and which you can also modify through the control bar.

Proceed similarly to the previous tests on the $C$ parameter, but this time for the kernel parameter $\gamma$ observing the results for small ($\gamma \simeq 0.0001$) or large ($\gamma \simeq 0.05$) values. When does overfitting/underfitting appear? Why is this so?

Recall from the theory what the influence of $C$ and $\gamma$ is.

PRO

The polynomial kernel does not use the $\gamma$ parameter, but the `degree` and `coef0` parameters. Try to tune these values and to understand what kind of effect they are having in the model. Explain the results obtained.

## 2.4 Real–world example

Up to now, the classification problems seen were artificial datasets for illustration purposes. We can proceed now to address a real–world problem. The dataset *thyroid* presents information from a number of patients, some of them suffering from abnormal thyroid function (hypothyroidism/hyperthyroidism), while the rest are healthy. The problem, given the numerical results from 5 clinical tests, consists of building a model able to perform diagnosis automatically.

To address this problem we will use the non–interactive script `svm_train.py`. When running it the data for the *thyroid* dataset will be loaded, and a linear SVM model will be trained using a subset of it (training set), testing then the accuracy of the built model on a separate subset (test set) not used for training.

Doing this so is common practice. Issues such as overfitting can be hard to detect by just looking at the accuracy obtained by the model in the data used for training. That is why the accuracy of the model being tuned is measured over an independent set of data, generally named validation or test set. With this, it is safe to adjust the model parameters by trying to maximize the accuracy of the model over the validation set.

Open the script `svm_train.py` with a Python editor and examine the source code. Notice that the parameter values are defined in the `CONFIG` section, using the same notation as the graphical interface of the previous exercises. Run the script and note down the accuracy level obtained.

Explore how accuracy changes with linear and non–linear SVMs, as well as with different kernels and regularizations (you will need to re–run the script after every change). Which are the best values you can find? What accuracy do they produce?

Let us concentrate on the RBF SVM. When tuning it, a common strategy is to test different $C$ and $\gamma$ values following a logarithmic scale. For example, for $C$ we can try with values such as $C = 1, 10, 20, 50, 100, \ldots$. Same goes for $\gamma$, so that a grid of possible values is explored. Hence the name of *grid search* given to this procedure.

The *thyroid* dataset is small and we can do this in a short time. Note however that for large datasets the time grid search takes can be way too long. Although there are some heuristics and more advanced techniques, efficient parameter tuning is still an open problem.

Edit `svm_train.py`, trying with a grid of values for $C$ and $\gamma$. Base the limits and resolution of your grid on your previous findings. Do you manage to improve accuracy? If so, with what values?


Create in `svm_train.py` a new function named `tune` that does grid search automatically. Given a grid of $C$ and $\gamma$ values, the function should train an SVM with each possible pair of values, measure the accuracy on the validation set for each choice, and return the parameter values that perform best. Test that the function works fine with the same grid you tried manually.

# 3. Support Vector Regression

## 3.1 Linear SVR

We will now illustrate the use of Support Vector Machines for regression, resorting again to the `svm_gui.py` script. Run the script and draw a dataset roughly following a straight line, such as the one shown in the left part of Figure 6.

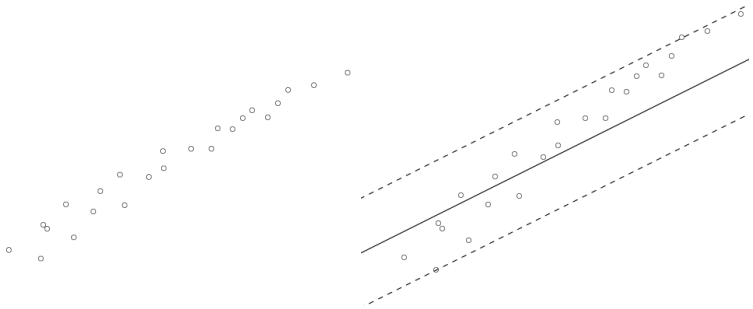

Figure 6: Plot of a linear regression problem (left–hand side) and linear SVR model over it (right–hand side).

Now select `Regression` in the control bar and click the `Fit` button. This will train a linear Support Vector Regression model, showing the resultant regression line as a solid line in the canvas, and the $\epsilon$–tube marked through two dashed lines. If done correctly, you should obtain a model similar to the one shown in the right part of Figure 6.

The SVR model has an additional parameter on top of the usual $C$ parameter: $\epsilon$. This parameter controls the width of the regression tube, and also sets the amount of acceptable regression error in our model.

Try modifying $\epsilon$ to some other values and fitting the model again. How does the angle of the regression line and the margins change? Can you explain this phenomenon?

The theory says that the wider the $\epsilon$–tube is, the less non–zero errors are made by the model.

## 3.2 Non–linear SVR

Let us now address a non–linear regression problem with SVR. To do so, `Clear` the canvas and draw a sinusoid–like dataset, such as the one shown in Figure 7. Now select the RBF kernel and `Fit` the model. The model obtained will probably perform poorly, so some tuning is necessary. A non–linear SVR model has three parameters to tune: the $C$ penalty parameter, the kernel parameter $\gamma$ and the tube width $\epsilon$. The initial $\epsilon$ value should be good enough for this problem, but you might need to increase $C$ and $\gamma$ to obtain a good fit.

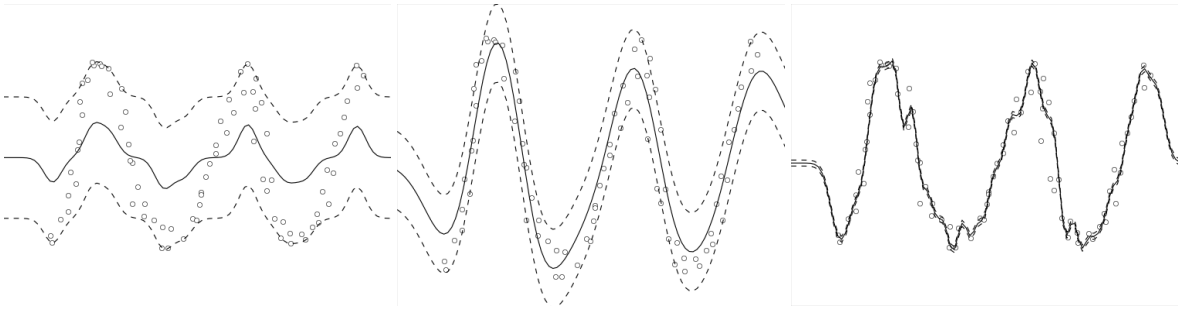

Figure 7: Different SVR models trained over a non–linear dataset, for varying values of $\epsilon$. The left–hand side image shows an underfitting case, while the right–hand side image presents an overfitting case. The image at the middle shows the results of a correct tuning of $\epsilon$.

Adjust the $C$ and $\gamma$ values of the SVR until you obtain a model similar to the one shown in the middle of Figure 7. Note down these values.

The influence of $\epsilon$ on the model is more significant in non–linear problems, as non–linear models are more prone to overfitting due to their higher capacity, i.e. their ability to represent very complex functions.

Modify the value of $\epsilon$ until you can obtain models similar to the underfitting and overfitting cases shown in Figure 7. For which values of $\epsilon$ does each problem arise? Why is so?

You may observe similar behaviors for quite different choices of $C$, $\gamma$ and $\epsilon$, as the effect of each parameter can compensate the others. If in doubt, fix two of the values and modify the remaining one, so that you understand better what its individual effect is.

Create a new version or modification of the script `svm_train.py` so that it trains an SVR model. Then build a model for the real–world dataset `wind`, which contains historic data on wind speed and direction at the wind power plant of Sotavento [a]. The objective is to predict the power production using such wind info.

[a] `http://www.sotaventogalicia.com`

You can find hints on how to do this in `http://scikit-learn.org/stable/modules/svm.html#regression`. Basically you should replace the class `svm.SVC` with `svm.SVR`.

Explore manually some values for the SVR parameters. Note down the best ones you get, as well as the performance of your SVR model.

The performance of SVR is measured in terms of the $R^2$ score, instead of accuracy percentage. The best possible value is a score of 1. A model that always predicts the same $y$ scores 0, while an even worse model can score negatively.

Create in your SVR training script a new function name `tune` that performs automatic tuning of all the $C$, $\gamma$ and $\epsilon$, similarly to the previously proposed exercise for SVM.

## 4. One–class SVM

To conclude this assignment, we will see the One–class SVM. Once again, we will resort to the `svm_gui.py` script to do so. Draw a dataset following a ball–like shape, similar to the one shown in Figure 8, whose points belong to a single class. Linear One–class SVM models are seldom useful, since usually it is not possible to enclose the support of a distribution with just a linear boundary. Therefore, one–class models are nearly always be used in their non–linear form.

One–class SVMs are generally only used in problems where the number of available training patterns from one of the classes is extremely small. In such situations, training a standard classification SVM can lead to poor models. Training a one–class SVM just on the patterns of the majority class can sometimes produce better results.

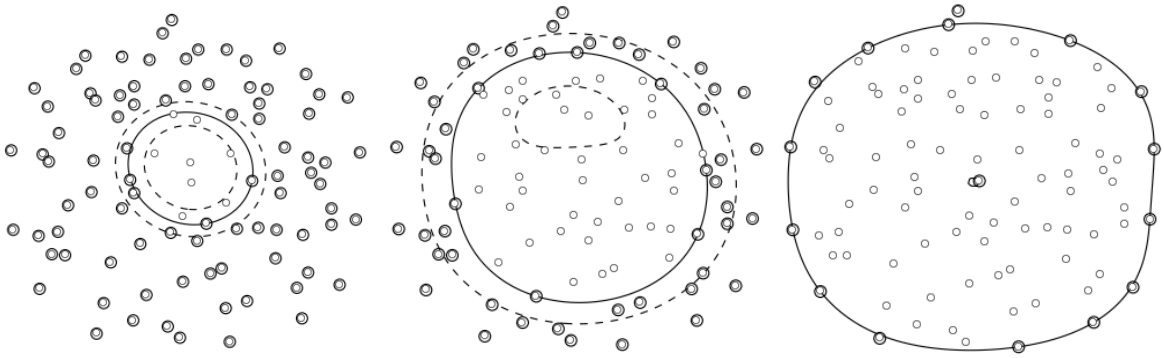

Figure 8: Different One–class SVM models trained over a non–linear dataset, for different values of $\nu$ that produce different supports.

Because of this, let us choose directly the RBF kernel. Now, One–class SVMs do not have a $C$ parameter, but a $\nu$ parameter instead. For now, just set $\nu = 0.5$, and also $\gamma = 0.001$, and `Fit` the model. You should obtain a model similar to the one shown in the center plot of Figure 8. The solid line represents the decision boundary of the One–class SVM: any data point outside this enclosing is regarded as an outlier or anomaly by the model. The larger the area enclosed inside this line –the so–called *support* of the distribution–, the less sensitive the model will be to outliers.

The effect of the $\nu$ becomes clear when building the One–class SVM with two extreme options of this parameter. Try modifying the value $\nu$ and observe the changes in the model. Note that $\nu \in (0, 1]$. How does the support change? For which values do you obtain models similar to those shown in Figure 8?

Notice as well the difference in the SVs chosen by the model for different values of $\nu$. What is the relationship between them? Why does this happen?

Revise the theory for the influence of the $\nu$ parameter.

Add some points to the dataset, far away from the original data. These points are anomalies that do not follow the usual data distribution. Find a value for $\nu$ so that the model captures most of the points in the central distribution but none of the anomalies. Which value provides the best solution?

Once you find this value of $\nu$, test different values of $\gamma$. How do the models change? Why?

Create a new version or modification of the script `svm_train.py` so that it trains a One–class SVM model. Then build a model for the real–world dataset `sonar`, which contains patterns of bouncing sonar signals from metal cylinders (mines). The objective is to be able to tell the difference between genuine mines and oddities in the environment, such as cylinder–shaped rocks.

You can find hints on how to do this in `http://scikit-learn.org/stable/modules/svm.html#density-estimation-novelty-detection`. The class involved now is `svm.OneClassSVM`.

Explore manually some values for the One–class SVM parameters. Note down the best ones you get, as well as the performance of your model.

The `sonar` problem is hard to solve, so expect low accuracy rates.

Adapt the function `tune` you wrote for SVMs that performs automatic tuning of $C$ and $\gamma$ to automatically tune a One–class SVM for the `sonar` dataset.

If you consider that $C$ acts as $\nu$, as the graphical interface does, the code may not need any particular adaptation.