1 SVM Tutorial

1.1 Setup and Introduction

The tutorial is *Old School*. It consists of a set of ROOT scripts and compiled programs. You need some basic knowledge in Unix and ROOT. We first use `git` to download the example files, the `libsvm` and `SVM-HINT` libararies[1].

Login on one of the DESY servers with your `username` and `password`.

```
> git clone https://stash.desy.de/scm/~kruecker/svm_tutorial.git
> git clone https://github.com/cjlin1/libsvm
> git clone https://github.com/ml-hint/svm-hint
```

Now you should have 3 directories:
`svm-hint svm_tutorial libsvm`

There is a tutorial example in svm-hint. PLEASE, do not run it! It is meant to run on a multi-core architecture and will create a 1000% CPU load. It will kill our servers if all participants run it at the same time.

libsvm (`https://www.csie.ntu.edu.tw/~cjlin/libsvm/`) is an efficient and fast SVM library. It is written in pure C and comes with a few command line tools. The library is used in our SVM-HINT interface but for this tutorial we start with the unmodified version from `github`. We first build all necessary tools for this and the following sections. ROOT and a reasonable modern C++ compiler must be available for SVM-HINT.

```
> module load root/5.34
> module load gcc/47
> cd svm-hint
> make
> make asimov
> cd ../libsvm
> make
```

[1] The code we used for our paper http://arxiv.org/abs/1601.02809 on "Performance and optimization of support vector machines in high-energy physics classification problems" is available at `https://github.com/ml-hint/svm-hint`.

Next we copy the command line tools into the tutorial folder:

```
> cp svm-predict svm-scale svm-train tools/grid.py ../svm_tutorial
> cd ..
> cp svm-hint/asimov svm_tutorial
```

We have now a copy of the tools in the `svm_tutorial` folder where the tutorial continues.

1.2 LIBSVM – Spiral example

We start with the spiral example from the lecture. The script `makeSpiral.C` creates a simple example file with spiral data (8000 points). Each call creates a new random sequence. We create a training and a test sample.

```
> cd svm_tutorial
# root must have been loaded: module load root6/6.02
> root -q -b makeSpiral.C
> mv spiral.txt spiral_train.in
> root -q -b makeSpiral.C
> mv spiral.txt spiral_test.in
```

We have now 2 statistically independent datasets. The files are pure text files and you can inspect the content with your favourite editor. The class label is the first column, each feature starts with `n:real` $n = 1 \ldots n$.

The command line tool to train the SVM is `svm-train`:

```
> ./svm-train -g 50 -c 10 spiral_train.in spiral_train.model
...
 optimization finished, #iter = 11894
 nu = 0.156441
 obj = -8733.721244, rho = 0.560677
 nSV = 1539, nBSV = 1000
 Total nSV = 1539
```

(Although we do not really need these details, `#iter` is the number of iterations the algorithm needed to find the minimum, `obj` is the value of the objective function at the minimum and `rho` is called b^* in the lecture.)

The exact numbers here and in all other examples depend on your sample! On the

command line '`-g 10`' specifies the γ value for the RBF Kernel, '`-c 10`' specifies the penalty parameter for overlapping distributions. The parameters and support vectors describing the trained SVM are saved in `spiral_train.model`. The model file is also a pure text file that can be inspected with an text editor.

The number of support vectors `nSV` and bounded SV `nBSV` is large with this first parameter choice. Support vectors are called bounded if they reach the upper limit $\alpha = C$. The parameter C allows to switch between a soft margin to a hard margin SVM with $C \to \infty$. In the spiral example the data is in a certain sense not really overlapping. With a proper mapping into a higher dimensional space it should be possible to choose a large value of C with a small number of bounded SV. Before we investigate this further, we try the prediction step. We used the model file we have created before and predict the class labels for the second, independent test dataset.

```
> ./svm-predict spiral_test.in spiral_train.model spiral_test.pred
 Accuracy = 97.05% (7764/8000) (classification)
```

Here, **Accuracy** is the percentage of correctly predicted class labels ($+1$ and -1) divided by the total number of data points. We are already doing quite well but based on a complex model with about 1500 support vectors.

Cross-validation is a different way to obtain a performance measure. Instead of running on an independent test data sample the trainings data itself is used.

```
> ./svm-train -g 50 -c 10 -v 5 spiral_train.in
 <some output>
 Cross Validation Accuracy = 96.525%
```

The flag '`-v 5`' selects 5-fold cross-validation. The input sample is divided in 5 sub-samples and the SVM is trained on $4/5$ of the data leaving $1/5$ as test sample. This is repeated 5 times taking always a different sub-sample as test sample.

As next step, try different values of $\gamma = 5, 50, 100$.

'`-g 5`' gives quite a bad performance which is worth to be visualised as a negative example:

```
> ./svm-train -g 5 -c 10 spiral_train.in spiral_train.model
> ./svm-predict spiral_test.in spiral_train.model spiral_test.pred
> root -l inspectSpiral.C
```

The upper two plots show the trainings data, the lower two the correctly classified (left) and wrongly classified (right) points.

Scaling of the input data is important to improve the performance. The `svm-scale` command normalizes the trainings data by the max-min-range for each feature. If we scale the trainings data we must apply the same transformation to the test data before the prediction step.

Train the SVM on the with ($\gamma = 100, C = 10$) and note accuracy and nSV,nBSV. Then do the same with the scaled datasets:

```
> ./svm-scale -s spiral_train.scale spiral_train.in > spiral_train_scaled.in
> ./svm-train -g 100 -c 10 spiral_train_scaled.in spiral_train_scaled.model
 ..*...*
 optimization finished, #iter = 5582
 nu = 0.070783
 obj = -3983.328302, rho = 0.266557
 nSV = 704, nBSV = 459
 Total nSV = 704
```

We scale the test data with the factors we have saved before in `spiral_train.scale`.

```
> ./svm-scale -r spiral_train.scale spiral_test.in > spiral_test_scaled.in
> ./svm-predict spiral_test_scaled.in spiral_train_scaled.model spiral_test_scaled.pred
 Accuracy = 98.4375% (7875/8000) (classification)
```

We note that after scaling not only the accuracy improves but also the number of support vectors becomes smaller. This shows that it becomes easier for the algorithm to find a suitable model. A last test on the spiral model;

```
./svm-train -g 15 -c 1000000  spiral_train_scaled.in spiral_train_scaled.model
 optimization finished, #iter = 1011840
 nu = 0.000429
 obj = -1717416.539103, rho = 65.153782
 nSV = 102, nBSV = 0
 Total nSV = 102
> ./svm-predict spiral_test_scaled.in spiral_train_scaled.model spiral_test_scaled.pred
 Accuracy = 99.075% (7926/8000) (classification)
> root -l inspectSpiral.C # change the name of the 2 input files first!!
```

We get a small number of support vectors, none (or only a small number) of bounded support vectors and a high accuracy! In general one cannot expect the number of support vectors to be small. It depends on the complexity of the boundary.

As a last test on this dataset
`./svm-predict spiral_train_scaled.in spiral_train_scaled.model spiral_train_scaled.pred`
Often this will give a 100% accuracy. Do you know why?

1.3 Invariant Mass Example

Automatized Parameter Tuning In the first example we have studied the influence of the (C, γ)-values. The performance of an Support Vector Machine strongly depends on these parameters. The simplest way to find a good parameter set is a simple grid search. Within a certain range all possible parameter pairs are evaluated. LIBSVM provides a simple Python script for such a grid search `grid.py`.

First we run the invariant mass toy MC and create 2 datasets with 2000 events. Then the data sets are scaled and the grid search is started.

```
> root -q -b makeToy.C
> mv toy.txt toy_train.in
> mv toy.root toy_tune.root
> root -q -b makeToy.C
> mv toy.txt toy_test.in
> mv toy.root toy_eval.root
> ./svm-scale -s toy_train.scale toy_train.in >toy_train_scaled.in
> ./svm-scale -r toy_train.scale toy_test.in >toy_test_scaled.in
> ./grid.py -log2g 1,-3,-1 -log2c 5,15,3 -svmtrain ./svm-train toy_train_scaled.in
```

It will run a few minutes.

The logarithmically spaced search grid is defined by `-log2g 1,-3,-1` and `-log2c 5,15,3`. The tool performs a 5-fold cross-validation on the trainings data `toy_train_scaled.in` and optimises the accuracy. The SVM executable is defined by `-svmtrain ./svm-train`. The script creates a 2-dim surface plot that shows the accuracy as a function of C and γ which will be saved as `toy_train_scaled.in.png`. For the tutorial the search range is a bit pre-optimised to get a quick (slightly sub-optimal) results.

Inspect the contour plot (`display toy_train_scaled.in.png`) is the final result an optimal parameter pair?

The script can easily run with multiple workers (line 15: `nr_local_worker`) but, **please, do not** use these mode with more then 2 workers as we do not want to overload our servers during the tutorial.

In the next step we train an SVM with the found best pair (16384, 0.5) and run the prediction step on the test sample.

```
> ./svm-train -c 16384 -g 0.5 toy_train_scaled.in toy_train_scaled.model
 <some output>
 nSV = 1454, nBSV = 1415
 Total nSV = 1454
> ./svm-predict toy_test_scaled.in toy_train_scaled.model toy_test_scaled.pred
 Accuracy = 84.7% (3388/4000) (classification)
```

The results can be visualised:

```
> root -l inspectToyA.C
```

1st plot: signal in trainings sample. Note the clear correlation in the three variables $E1, E2$ and ϕ. **2nd plot:** background in trainings sample. $E1, E2$ and ϕ are independently scattered. **3rd plot:** distribution classified by the SVM as signal. Red true signal, blue false signal. The SVM selects the correct phase space area and the pattern from plot 1 is reproduced. **4th plot:** False signal alone. Note that the ML algorithm picks those points from the independently distributed background that looks like signal. A ML algorithm cannot do magic if accidentally the background looks like the signal it is selected and the background is forced to look like a false signal.

```
> root -l inspectToyB.C
```

1st plot: Signal invariant mass distribution in the trainings sample. **2nd plot:** Background invariant mass distribution in the trainings sample (the background is downscaled by a factor of 10). **3rd plot:** Selection efficiency as function of the invariant mass i.e. (predicted signal)/all. Without knowing about the invariant mass the SVM nicely cuts out a band around the mass peak. **4th plot:** As background classified part of the invariant mass distribution for signal and background. The band around the invariant mass is cut out as it looks as signal.

2 Significance based tuning

In the previous example we saw that the SVM is able to identify the correlation produced by the invariant mass of the mother particle. A cut on the probability should be able to improve the significance. We use the `svm-hint` library.

First we build the test program `svm_hint_example.cpp` for the significance based tuning.

```
> make
```

Have a look at the code. It reads the files we have created before for the `grid.py` example. The program uses our `svm-hint` library to find the optimal triple (C, γ, p_{cut}) by an adaptive grid search. For the significance calculation the sum of event weights is used as number of events. A systematic error on the background can be included and it is set to 10% in this example. The code is doing the scaling automatically.

```
> ./svm_hint_example
  <lots of output>
  Optimized C 0.1000000000000000 optimized gamma 9.7288097312706245 \
  optimized discriminator cut 0.5250 with the significance of 4.3939 in the test sa
  <more output>
  Significance obtained from the given cut: 4.253506218689  signal yield:\\
   575.000000000000  background yield: 1130.000000000000
   <...>
```

The first significance value refers to the trainings sample, the last to the evaluation sample.