10907 Pattern Recognition

Lecturers
Prof. Dr. Thomas Vetter ⟨thomas.vetter@unibas.ch⟩

Assistants
Dennis Madsen ⟨dennis.madsen@unibas.ch⟩
Dana Rahbani ⟨dana.rahbani@unibas.ch⟩
Moira Zuber ⟨moira.zuber@unibas.ch⟩
Genia Böing ⟨genia.boeing@unibas.ch⟩

Exercise 4 — Support Vector Machine

Introduction	04.11
Deadline	**12.11** Upload code to Courses.
	11.11+12.11 Group presentations, U1.001, see schedule online

In this series you will implement and compare linear and non-linear support vector machines. The classifiers will be applied to three problems: A linear separable toy problem, a non-linear separable toy problem and handwritten character recognition (OCR).

You can download the data needed for this exercise and a code skeleton from the following repository: `https://bitbucket.org/gravis-unibas/pattern-recognition-2019`.
A number of helper functions are provided - your task is to implement the *TODO* parts in the python (.py) files.

Remember to upload your code to courses in a ZIP file. Do NOT include the data folder. Only the python files you edited should be included + a file containing the group members names. Only 1 person from the team should upload!

Data:

Each `mat`-file contains the training and the test set for a problem, named `NAME_(train|test)`. The sets are encoded as $(d+1) \times N$-matrix, where N is the total number of data points and d the dimension of the feature vector. The first column contains the label $y \in \{-1, 1\}$. For the non-linear separable toy example, the data is available in the `.mat` files. The dimensionality of the data is structured the same as in the `mat`-files, except that the data and the labels are stored in separate files.
The provided skeleton code already loads the data for you and reshapes it into an appropriate format.

- `toy` ($d = 2$): A very small (x,y) toy data set. Use it for development and to study the behaviour of your linear classifiers. It can be easily visualized.
- `skin` ($d = 2$): A very small (x,y) toy data set. Use it for development and to study the behaviour of your kernel classifiers. It can be easily visualized.
- `zip13` ($d = 256$): Handwritten digits automatically scanned by the U.S. Postal Service. The set is reduced to binary classification scenarios: the digits 1 and 3 are provided. The digits are normalised to a grey scale 16×16 grid.
  To display the first digit image use

  ```
  img = np.reshape(x_array[:, 0], (16, 16))
  plt.imshow(img, cmap='gray')
  ```

- `zip38` ($d = 256$): As above, but with the digits 3 and 8.
- `ship_no_plane` ($d = 32 \times 32 \times 3 = 3072$): An extract from the popular CIFAR dataset.

If you are interested in the complete USPS data set, it can be obtained from
`http://statweb.stanford.edu/~tibs/ElemStatLearn/data.html`

Remarks:

- Do not use the test sets for training!
- Be aware of the computational demands. Some implementations may take a while to train with lots of data. During development use only a few data points until your implementation works, then train with more data. It might be impractical to use all data provided for your PC. In this case use as much of the training data as possible.

1 Linear SVM

Implement and test a Support Vector Machine classifier. The following TODO sections will first provide mathematic notation to help to solve the programming parts.

Todo 1 (Linear Support Vector Machine - CVXOPT interface) *Start with the quadratic programming problem given in the lecture script. To solve it, the function* `cvxopt.solvers.qp()` *will be used. Use the full documentation online to get more details:*
http://cvxopt.org/userguide/coneprog.html#quadratic-programming.

The primal problem of the SVM as seen in the lecture:

$$\underline{\omega} = \underset{\underline{\omega}}{\operatorname{argmin}} \frac{1}{2}\|\underline{\omega}\|^2 \text{ subject to } y_i(\underline{\omega}^T \underline{x}_i + \omega_0) \geq 1 \ , \forall i$$

And with the corresponding dual problem:

$$\underline{\lambda} = \underset{\underline{\lambda}}{\operatorname{argmax}} \left(\sum_i^N \lambda_i - \frac{1}{2} \sum_{i,j}^N \lambda_i \lambda_j y_i y_j \underline{x}_i^T \underline{x}_j \right) \text{ subject to } \sum_{i=1}^N \lambda_i y_i = 0 \ , \ \lambda_i \geq 0 \ , \forall i$$

In vector format, the dual problem looks as follows:

$$\underline{\lambda} = \underset{\underline{\lambda}}{\operatorname{argmin}} \left(\frac{1}{2} \underline{\lambda}^T \underline{\underline{H}} \underline{\lambda} - \underline{1}^T \underline{\lambda} \right) \text{ subject to } \sum_{i=1}^N \lambda_i y_i = 0 \ , \ \lambda_i \geq 0, \forall i$$

Where H is a $N \times N$ matrix ($N = \#ofSamples$) created from the given data x and their respective labels y as seen in the expanded dual problem: $y_i y_j \underline{x}_i^T \underline{x}_j$.
The problem needs to be translated into the interface provided by `CVXOPT` with the function `cvxopt.solvers.qp(P, q[, G, h[, A, b[, solver[, initvals]]]])`.

$$\underline{x} = \underset{\underline{x}}{\operatorname{argmin}} \left(\frac{1}{2} \underline{x}^T \underline{\underline{P}} \underline{x} + \underline{q}^T \underline{x} \right) \text{ subject to } \underline{\underline{A}} \underline{x} = \underline{b} \ , \ \underline{\underline{G}} \underline{x} \leq \underline{h}$$

Construct the respective matrices and vectors by using the CVXOPT matrix `cvx.matrix()`. The solution can be optioned through the x variable:

```
import cvxopt as cvx
cvx.solvers.options['show_progress'] = False
solution = cvx.solvers.qp(P, q, G, h, A, b)
lambdas = solution['x']
```

`Warning: The` x `matrix in CVXOPT should not be confused with the data matrix. The` x `matrix used in CVXOPT is instead the` λ `vector from the dual problem.`

Remember only to use the λ's that are larger than 0 (in practice use a value close to zero: 1e-5). Refer to the lecture slides for information about how to compute the $\underline{w}$ vector and the bias term w_0. The w_0 value can be computed by using the mean of all support vectors for stability.

Todo 2 (SVM training implementation) *Use the above information about the CVXOPT library to implement the* `train` *function in the* `SVM` *class found in the* `svm.py` *file.*

Todo 3 (SVM linear classification) *Implement the linear classification function* `classifyLinear()` *in the* `SVM` *class. The formula for the linear classifier (primal form):*

$$f(\underline{x}) = \underline{\omega}^T \underline{x} + \omega_0$$

Call the above classification function from the `printLinearClassificationError()`*. Compute and print the classification error.*

Todo 4 (Experiments) *Apply your classifier to the linear toy example. Skeleton code is given in the file* `ex3-SVM_1_LinearToy.py`*.*

Todo 5 (Soft-margin) *Extend the linear solution with a soft margin (controlled by the C parameter).*
Remember from the lecture slides that the contraint $0 \leq \lambda_i, \forall i$ *needs to be further constrained to*

$$0 \leq \lambda_i \leq C$$

Try running the `ex3-SVM_1_LinearToy.py` *with different* `C` *values and observe what happens. C values:* `1,10,100,1000,None`

- *What influence does the parameter* `C` *of the SVM have? Count the number of Support Vectors in the classifiers.*

Hint: To add the soft-margin, extend the G and the h matrices.

NOTE: the bias w_0 *should only be approximated from the support vectors, i.e. the data points on the margins. When using the slack-variables, data points not on the margin will also have* $\lambda > 0$*. For simplicity, you can however approximate the bias from all* $\lambda > 0$*.*

2 Kernel SVM

Todo 6 (Kernel functions and kernel matrix) *Extend the* `SVM` *class by implementing the local kernel functions.*
Implement the linear, the polynomial and the RBF kernel function

- *Linear kernel:* $k(\underline{x}, \underline{x'}) = \underline{x}^T \underline{x'}$
- *Polynomial kernel:* $k(\underline{x}, \underline{x'}) = (\underline{x}^T \underline{x'} + 1)^p$
- *RBF (Gaussian) kernel:* $k(\underline{x}, \underline{x'}) = \exp\left(-\frac{\|\underline{x}-\underline{x'}\|^2}{2\sigma^2}\right)$

The `__computeKernel__` *function also needs to be implemented. This computes the complete kernel matrix and takes in the kernel function to be used as an argument together with the kernel parameter.*

Todo 7 (Non-linear SVM) *Expand the* `SVM` *class with a* `classifyKernel()` *function and the* `printKernelClassificationError` *similar to the linear case.*
Formula for the non-linear classifier (dual form):

$$f(\underline{x}) = \omega_0 + \sum_{i=0}^{N_s} \lambda_i y_i K(\underline{x}_i, \underline{x})$$

Because of the non-linearity of the classification there are no w coefficients anymore. Think of how the decision surface is now defined and how a single data point is classified.

Todo 8 (Non-linear Experiments) *Apply your classifier to the non-linear separable toy example. Skeleton code is given in the file* `ex3-SVM_2_KernelToy.py`*. Try out different kernels and kernel parameters to see what gives the best solution boundary.*

Todo 9 (MNIST and CIFAR Experiments) *Train your non-linear SVM for the* `zip38` *and the* `CIFAR` *datasets. Skeleton code can be found in the* `ex3-SVM_3_MNIST.py` *file. Visualization of correct and incorrect classified images are done in the* `visualizeClassification()` *function. Compare the classification train and test error by using different kernels (linear, polynomial, RBF). Discuss the following questions:*

- *What is the influence of σ? Compare in particular the training and test error for different values $\sigma = 0.1, 1, 10$*
- *What is the influence of C when used together with the RBF kernel? Compare in particular the training and test error for different values.*

Todo 10 (Linear vs Non-linear speed comparison) *Use the linear separable dataset to perform a speed comparison of the linear svm and a kernel svm with a linear kernel. Implement this in the* `ex3-SVM_4_SpeedComparison.py` *file. Train the 2 SVM classifiers, and measure how much time the* `classifyLinear` *and* `classifyKernel` *calls take.*

- *Note down the average time for each classifier by running it 1000 times*
- *Which classifier is faster? How many times faster? And why?*