ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Pattern Recognition and Machine Learning
Miniproject

Lecturer: Matthias Seeger
Assistants: Abdallah Elguindy, Mathias Humbert, Young Jun Ko, Xinchao Wang

Miniproject

October 24, 2013

1 Introduction

The aim of this miniproject is to provide you with experience on implementing different machine learning methods and applying them to a time-honoured machine learning problem: hand-written digits pattern recognition. You will implement both error back-propagation gradient descent optimization for a multi-layer perceptron and a support vector machine algorithm, then apply your solution to the MNIST dataset, an excerpt of which is shown below.

The main aim of this exercise for you is to gain hands-on experience with some of the most widely used pattern classification methods, to understand how they work by implementing them in a clean, elegant way, and above all, to provide a thorough and statistically sound state-of-the-art evaluation of this methodology on real data. Make sure to keep your code as clean and simple as possible and focus on understanding and elegance rather than fine-tuning the bits. We are interested only in you understanding how the algorithms work, that your implementation works and is well tested, and in the resulting evaluation. Excessive optimization is not rewarded.

2 Project Overview

In this section, we provide a general overview, before going into details in subsequent sections. In terms of programming language for this project, we encourage you to use `MATLAB` or `Python/Scipy`, which are the major development tools used by machine learners. You may also use C++, Java or other languages, but in that case we may be less able to help with specific questions.
The most important rule: **You must not use any kind of foreign code** for this project, neither any piece of code from another group. In fact, you will have to demonstrate convincingly that you implemented all relevant algorithms by yourself. Copying even small pieces of code from elsewhere is considered **plagiarism**: a serious offense. We will automatically match your submitted code against code submitted by other groups, and against code out there in the web, so **do not copy**. The following exceptions apply (please check with us when in doubt):

- In case you do not use `MATLAB`, you are allowed to use libraries for basic linear algebra (vectors, matrices, matrix multiplication, ...). In fact, you are **discouraged** from implementing this for yourself.
- You are allowed to use foreign code in order to load or store data, or to plot results (we provide I/O code for `MATLAB` and `Python`).

The assignment consists of the following steps (this is more of a checklist, details are provided in subsequent sections):

1. **Register your group (max. 2 persons) on Moodle.**
   - There, please specify the programming language which you are planning to use (this influences your assignment to a TA).
   - Each group is assigned a TA who is responsible for answering your questions and monitoring your progress.
   - We will also assign training and test sets to each group.
2. **Download the MNIST database.**
   - You will work on two randomly drawn subsets (provided on the moodle) corresponding to binary subproblems of the full 10-class dataset, i.e. you are given patterns corresponding to two different digits (e.g. 9 and 4) encoded as $+1$ and -1 (exception is the bonus competition, see below).
   - For the multi-layer perceptron exercise, you will have to split the training data into a training and a validation set (at random, separately for each binary problem). Use the former for training, the latter to validate the method, for example for early stopping.
   - For the SVM exercise, you will use cross-validation, so no hold-out validation set is required.
3. **Implement multi-layer perceptron learning.**
   - Implement gradient-descent optimization (with momentum term) based on the error backpropagation rule as discussed in the lecture.
   - Train and evaluate it separately on the **two** different binary subtasks assigned to your group, and report results.

4. **Implement the SMO algorithm for SVM training.**
   - Implement the sequential minimal optimization (SMO) algorithm for training a support vector machine (SVM) with Gaussian kernel
   - You will receive a separate note with details about SMO.
   - Select the two free model parameters by cross-validation, then train and evaluate the resulting SVM on the **single** binary subtask assigned to your group, and report results.

5. **Write and submit a short report (max. 7 pages).**
   - Compare the algorithms. Report your findings not only in quantitative but also in qualitative terms.
   - **Hand in your source code as well. Include comments, clearly marking the core of your implementation in your code, that is, the most important lines of code**.
   - Be prepared for in-depth questions about the workings of your code.

3 The MNIST Data Set – Hand-written Digits Recognition

3.1 Downloading the MNIST Dataset

To get you started quickly, we provide two `.mat` files containing the training and test sets for the two binary classification problems. The format is native to MATLAB and you can load it directly. For Python users we provide a small code snipped `load_mnist.py` that illustrates how to load the files.
The first problem is to distinguish between digits 3 and 5 and the second to distinguish between 4 and 9. You will apply the MLP to both problems and the SVM only to 4 vs. 9. For the bonus competition, where you have to solve a 10-class classification problem, the full MNIST data set will be required for training[1]. The database is available at http://yann.lecun.com/exdb/mnist/.
Once you downloaded the database and set up the code to load it, verify that the input works for you. Load the database, plot a histogram of the labels and plot some digits (e.g. use `imagesc` and `reshape` in `MATLAB` to do that – or likewise `imshow` and `reshape` in Python).
Remember that the data comes in vectorized form, i.e. each image of a digit has a resolution of 28×28 pixels and is stored as a vector of length 784. The files that we provide contain pattern matrices of size $n \times 784$, i.e. each row corresponds to a pattern. **In the first contact meeting with your assigned TA, you should demonstrate that you can load the data and know how to get started. You should also demonstrate how you splitted the training part into a training set and a validation set.** Please communicate with your assigned TA early to get help. Do not wait until the end of the term!

3.2 Split the Data Set

The `.mat` files contain training and test sets. The test set is to be used only to determine the final performance of your implementation. All other operations related to training,

[1] To load the full dataset from the original format we provide example scripts on moodle (MATLAB/Python) that you might find helpful.

such as tuning parameters, has to be performed on the samples of the training set only. For that we ask you to use different validation strategies.
For the multi-layer perceptron (MLP) exercise, you have to **split** the training part into a **training set** (2/3 of cases) and a **validation set** (1/3 of cases), do this at random (in `MATLAB`, use `randperm`). Now, you have three datasets:

- A training dataset
- A validation dataset, for selecting between different parameters and for early stopping (MLP exercise only)
- A test dataset, which you use to report final results **only**

Do this split once, then keep it that way. Do not resample training and validation set anew each time.
For the support vector machine (SVM) exercise, you do not need a split into training and validation dataset, as you will use cross-validation. As you learn during the course, validation is an important part of machine learning, a measure to guard against *overfitting*.
Do not touch the test dataset **at any time** during training, model selection and parameter tweaking, certainly not for early stopping. This set has a single purpose only: to evaluate the performance (a single number, the test set error) of the very final trained classifier you commit to at the very end. You must not go back and tweak things after this final evaluation. Notice that there are no bonus points whatsoever for the best performance in absolute terms: you get points for a clean and statistically sound evaluation.
In the first contact meeting with your assigned TA, you should demonstate how you split the training part of the data into a training and a validation set.

3.3 Preprocess the Data

Preprocessing is in general an important part of any machine learning application. However, in the exercise here, a simple normalization suffices (if you are done with everything else, you are encouraged to try other ideas as well). You should normalize all input patterns to have coefficients in $[0, 1]$. To do this, compute the minimum $\alpha_{\min}$ and maximum $\alpha_{\max}$ over all coefficients of all *training patterns*. Then, replace each input pattern $\boldsymbol{x} \in \mathbb{R}^{784}$ by

$$\frac{1}{\alpha_{\max} - \alpha_{\min}} (\boldsymbol{x} - \alpha_{\min} \mathbf{1}) .$$

Apply this normalization to all patterns of the dataset, training and test. We recommend that you store the preprocessed data and work on it exclusively henceforth.
Note that $\alpha_{\min}$, $\alpha_{\max}$ are computed on the training part of the data only. This is because you *never* touch the test set for anything, not even for preprocessing.

4 Learning Multi-Layer Perceptrons (Start Now!!)

Your first task is to implement and evaluate gradient descent training for a multi-layer perceptron. You will use the MLP to solve the two different binary classification problems we provide, i.e. distinguish between digits 3 and 5 as well as digits 4 and 9.
As the miniproject takes substantial time to complete, not only the coding, but also the evaluation, it is important that you start now – do not wait until the end of the term. We first describe the setup, then give comments about

implementation and debugging, and finally explain how the evaluation should be done, and what you should provide in your final report (Section 4.4).

4.1 Setup

Make sure to study chapter 3 ("The multi-layer perceptron") in the course notes, we will use the same notation here. We recommend you use a two-layer MLP, with one hidden and one output layer. Use the *gating* transfer function

$$g(a_1, a_2) = a_1\sigma(a_2), \quad \sigma(x) = \frac{1}{1+e^{-x}}$$

for the *hidden* units. This setup deviates slightly from what is done in the course, in that each neuron (or transfer function) receives inputs from *two* different linear combinations of the inputs, with different weight vectors. The binary classifier is

$$f(\boldsymbol{x}) = \operatorname{sgn}(a^{(2)}(\boldsymbol{x})), \quad a^{(2)}(\boldsymbol{x}) = \sum_{q=1}^{h_1} w_q^{(2)} g\left(a_{2q-1}^{(1)}, a_{2q}^{(1)}\right) + b^{(2)},$$

$$a_k^{(1)} = \sum_{j=1}^{d} w_{kj}^{(1)} x_j + b_k^{(1)}, \quad k = 1, \ldots, 2h_1.$$

Notice that the activations $a_{2q-1}(\cdot)$ and $a_{2q}(\cdot)$ feeding into the q-th hidden neuron have different associated weight vectors and biases.
Suppose the binary training dataset is $\mathcal{D} = \{(\boldsymbol{x}_i, t_i) \mid i = 1, \ldots, n\}$, where $t_i \in \{-1, +1\}$. Your code should minimize the *logistic error function*, which evaluated over the whole training set is

$$E_{\log}(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n} \log\left(1 + e^{-t_i a^{(2)}(\boldsymbol{x}_i)}\right).$$

This error function is discussed in Section 8.2 ("Logistic Regression") of the course notes. Here, $\boldsymbol{w}$ is a placeholder for **all** the weights and biases of the network. Notice that this is **not** the "usual" squared error function, and also that the output layer activation $a^{(2)}(\cdot)$ is not pushed through another transfer function: its value goes directly into the logistic error function.
As this setup deviates from what is detailed on in the course notes, you will have to derive the backpropagation equations. **Please do this early, and then take the opportunity to show your solution to your assigned TA, who will help you spotting mistakes. Do not start coding before you verify the correctness of the equations!**

4.2 Implementation

Before you start hacking away, make sure to understand the structure of what you want to do. Your goal is to minimize the logistic error function $E_{\log}(\boldsymbol{w})$ w.r.t. all free parameters of the network, collectively called $\boldsymbol{w}$. To this end, you choose an *optimization algorithm*: gradient descent minimization with or without a momentum term. It is recommended that you use *stochastic gradient descent*, updating on single patterns or small batches. The optimizer is driven by the gradient $\nabla_{\boldsymbol{w}} E_{\log}(\boldsymbol{w})$, which you compute using the error backpropagation rule. The main computational backbone of your code will be this latter part.

Here some hints on the implementation. As discussed in detail in the course notes, MLPs consists of layers interleaving *linear* (inner product with weight vectors, adding biases) and *componentwise nonlinear* operations (evaluating transfer functions). For the linear parts, **it is very inefficient, error-prone and not elegant** to use `for` loops. You avoid `for` loops by using matrices and vectors, then coding the linear parts by way of matrix-vector or matrix multiplications. Use built-in language facilities:

- Matrix-vector or matrix-matrix multiplication $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$, $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{y} \in \mathbb{R}^m$, $\boldsymbol{A} \in \mathbb{R}^{m \times n}$: `y = a*x` in `MATLAB` .
- Pointwise multiplication: $\boldsymbol{z} = \boldsymbol{x} \circ \boldsymbol{y}$, meaning that $z_i = x_i y_i$: `z = x.*y` in `MATLAB` .
- Inner (or scalar) product: $\alpha = \boldsymbol{x}^T \boldsymbol{y} = \sum_i x_i y_i$: `alpha = x'*y` in `MATLAB` .
- Outer product: $\boldsymbol{A} = \boldsymbol{x}\boldsymbol{y}^T$ (if $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{y} \in \mathbb{R}^m$, then $\boldsymbol{A} \in \mathbb{R}^{n \times m}$): `a = x*y'` in `MATLAB` .

The golden rule: matrix-vector multiplication is better than a loop over vector-vector multiplication (inner product), and matrix-matrix multiplication is better than a loop over matrix-vector multiplication. For example, while it is admissable to run a `for` loop over a batch of training patterns to compute the error function and gradient, computing forward and backward pass for each point separately, it is more elegant and faster to process all cases in the batch together, propagating matrices through the network instead of vectors.
Here some general hints on getting your code to work (specific debugging hints are given below):

- Start with an artificial XOR problem (Section 3.1 in the course notes) with $\boldsymbol{x} \in \mathbb{R}^2$ and just four datapoints, and say $h_1 = 4$ hidden units.
  Use stochastic gradient descent, as described in Section 4.3. Test your gradient code (see below).
- Numerical errors: MLP training is fairly uncritical to implement, but still beware of numerical errors, in particular for the nonlinear componentwise steps in forward and backward passes. For example, terms of the form A/B, where either $A, B \approx 0$, or A, B very large, are critical: try to rewrite them into a new form C/D, where either C or D are order of unity (not too large, not too close to 0). A simple example is
$$\frac{e^{-x}}{1+e^{-x}} \quad \Rightarrow \quad \frac{1}{1+e^{x}}.$$
  The term on the left may be critical for x large negative. The term on the right is the same in exact arithmetic, but it is numerically uncritical.
  Another example: Computing $\log(1+e^x)$ directly does not work for large positive x, but notice that
$$\log(1+e^x) = x + \log(1+e^{-x}).$$
  Use the function `log1p`, available both in `MATLAB` and `C++`, to evaluate $\log(1+e^x)$ for $x < 0$. In short, to evaluate $[\log(1+e^{x_i})]$ for a vector $[x_i]$, you could make a case distinction between $x_i < 0$ and $x_i \geq 0$. You do not need a `for` loop for that in `MATLAB` : the expression `find(x>=0)` gives you the positions in $\boldsymbol{x}$ where $x_i \geq 0$, so you can compute the two parts separately. If in doubt about such points, please contact your assigned TA and show him/her your code.

- Choose small learning rates (such as 0.01). Experiment with different values and with a momentum term.
- Since the objective that we optimize is highly non-convex, a good initialization is crucial for the algorithm to converge to a good solution. Initialize all weights to small values drawn at random from a Gaussian $N(0, \sigma^2)$ where a good heuristic for the variance σ^2 is the inverse of the number of inputs to the correspinding hidden node, e.g. for a weight on the first layer it should be $\sigma^2 = \frac{1}{d}$ where d is the number of input dimensions(given its variance, how do you sample from this Gaussian?). Play around with these settings.
- Run your code several times from different initial settings. Do you always obtain similar results?

Debugging

Methods based on numerical optimization, in particular non-convex minimization as for MLPs, can be difficult to debug. As an important lesson to be taken away, you should implement the following measures (please do show that you have done so to your TA, and ask questions if something is not clear).
The most important lesson is to **test the gradients** which your backpropagation code produces. Gradient testing is discussed in Section 3.4.1 ("Gradient descent optimization in practice") in the course notes. Once you have established the correctness of your gradient code, a second property to test is whether your gradient descent optimization decreases the criterion value in every step. If this does not happen, try to reduce the learning rate and play with the momentum term. If problems persist, there is usually a bug.
One general comment. Do not debug your code on the full training set, choose smaller tasks to speed up bug-finding:

- Test the gradient on single data points (a few different ones), do not average over all the training data. You will use stochastic gradient descent anyway.
- Choose a small number h_1 of hidden units, first $h_1 = 1$, but then also $h_1 > 1$.
- MNIST digits are vectors in $\mathbb{R}^{784}$. To save time, it makes sense to create a debugging dataset of a few patterns by downsampling the bitmaps. In `MATLAB` , the relevant command is `imresize`. Choose a resolution of 12×12 or even 8×8.

4.3 Model Evaluation

Now is a good time to read the beginning of Section 10.1 in the course notes, on evaluating models by way of an independent validation set. If you have not already done so, split your training dataset into a training part (2/3) and an evaluation part (1/3). You do your gradient descent training on the training part only, the validation set is reserved for:

- Early stopping (Section 7.2.1 of the course notes): Evaluate the classification error on the validation set after each run over all training points (this is called an epoch). Plot both the logistic error function on the training set and the validation error. Stop training once the latter starts increasing.

- Model selection: Compare different network architectures (number of layers: we recommend you stick with 1, at most 2, hidden layers; number of hidden units) by their error on the validation set.

Here some general advice on how to run your method, given you established that the implementation is correct. Notice that most people working with MLPs use a host of "tricks of the trade" which they learn by experience and playing around, so this is an integral part of this exercise.

- It is recommended[2] that you use *stochastic (online) gradient descent* (Section 2.4.2 in the course notes). Each epoch (run over all training points) consists of two phases. First, iterate over the training points in random ordering and update $\boldsymbol{w}$ based on the stochastic gradient of $E_{\log}(\boldsymbol{w})$ evaluated on single points. Second, evaluate the *full* logistic error function over *all* training points (as well as the validation error for early stopping), this is what you plot.
  Play around with the learning rate and the momentum term in order to improve convergence speed. Theory predicts that in order to get stochastic gradient descent to converge, the learning rate has to be decreased eventually, but you should stick with a fixed rate initially for faster decrease.

- To get a feeling of how learning rate and momentum term affect convergence, use a small training subset (say, 50 patterns) and compare plots. However, be aware that you will have to optimize these parameters for the final training set size eventually.

In machine learning, evaluation of a method (or a set of methods, for a comparative study) is a two-stage process. First, you play around and compare many different options, in order to find a configuration which works best. You train on the training set and use the validation set in order to compare choices. If you have a lot of data, it is best to use *different* validation sets to make different choices, so to reduce the risk of overfitting. Of course, **you never touch the test set at all**. In the second stage, you fix a setting which you are happy with and do a final training run with early stopping, then report test set performance of the final classifier. Here are some ideas for your first stage exploration (requirements for the second stage are given in Section 4.4):

- Compare curves for different numbers of hidden units (say, between 10 and 100), inspect both the training and the validation error. You should witness overfitting at some point (in order to force overfitting, you can also decrease the training set size).

- Compare different choices of learning rate and momentum term parameter. Do they only affect convergence speed or also the absolute error values at the end? What happens with too large a learning rate? With too little momentum?

- Run the algorithm several times from different initializations (all drawn at random, see above). Do you always get a similar final performance?

- Look at the misclassified patterns, in particular those for which the value $t_i a^{(2)}(\boldsymbol{x}_i)$ is largest negative, versus some for which $t_i a^{(2)}(\boldsymbol{x}_i)$ is close to zero, but negative. Any interpretation?

[2] It is admissable to use gradient descent on the criterion evaluated on the whole training set as well, but you will find that stochastic gradient descent runs faster.

4.4 Requirements for the Report

Writing a good final report is part of the exercise, and you have some freedom how you do this. It is highly recommended **to show (parts of) your report to your TA early on and get feedback**. Here is what you should include:

(I) Introduction Write a short introduction (3-5 sentences only) stating the topic and aim of the project. Note that it is best to write a joint introduction to the MLP and SVM part.

(II) Methods Describe what you did and how you did it. This section should suffice for someone (who knows about MLPs and has the course notes) to reproduce your results.

1. How did you treat the data. Describe splitting, preprocessing, etc.
2. The final setup of the MLP you chose to use. Describe the network structure, transfer function, etc. Justify your design and parameter choices. To do so, show comparative plots that illustrate the effects of learning rate, number of hidden units, momentum, etc. on (a) overfitting and (b) convergence speed, and comment on them qualitatively.
3. Detail whether or not you used the same parameters for the two different binary sub-tasks. Motivate this with the plots you show.
4. Show at least one typical example of overfitting (say, by using a large number of hidden units and keep on training once the validation error increases).

(III) Results & Discussion To evaluate the performace of the MLP you should

1. Plot curves of (a) the training set logistic error, (b) the validation set logistic error, (c) the validation set (zero/one) error as a function of the epoch number

$$\frac{1}{m}\sum_{j=1}^{m} \mathrm{I}_{\{t_j a^{(2)}(\boldsymbol{x}_j)\leq 0\}}.$$

2. For both binary decision tasks provided to you: Annotate each plot separately by the error on the test set of the final classifier and give the standard deviation (but do *not* plot a test error curve – remember that the test set is never used during training). Briefly discuss these findings.
3. Finally, for one of the subtasks of your choice: plot examples of patterns which are misclassified, one for large negative $t_i a^{(2)}(\boldsymbol{x}_i)$, another for negative $t_i a^{(2)}(\boldsymbol{x}_i)$ close to zero. Comments?

5 Support Vector Machines

You will implement the sequential minimal optimization (SMO) algorithm for the binary (soft margin) support vector machine (SVM). Additional documentation about this algorithm comes with this document. Also, there will be part of a tutorial session dedicated to this topic (check `moodle`).
You will use the SVM to solve one of the binary classification problems we provide, i.e. distinguish between digits 4 and 9.
General hints about implementation and debugging apply here as well. The following points are different from the MLP task:

- You only apply SVM to the binary subtask 4 vs. 9 (not both of them).
- You do not use a validation set. Instead, you select free parameters by 10-fold cross-validation (Section 10.2.1 of the course notes).

You may use a Gaussian or a polynomial kernel (Section 9.2.3 of the course notes), we recommend the Gaussian. In general, SVM code is somewhat more difficult to write than MLP code, but it is easier to debug: you simply check whether the criterion always goes down, and whether the KKT conditions are eventually fulfilled. Recommendations about convergence criteria are given in the additional documentation. Start once more on a simple toy dataset like XOR, then move to debugging subsets of your task. Downsampling MNIST patterns speeds up the debugging, since kernel evaluations are the most expensive part of SMO.

5.1 Requirements for the Report

Once more, you have some freedom, but here are essential points:

(I) Introduction The introduction is already included in the MLP part, therefore you do not need to write anything here.

(II) Methods Similar to the MLP part, describe precisely what you did and how you did it. This section should suffice for someone to reproduce your results.

1. Describe what you did with the data, that is splitting, preprocessing, etc.
2. Describe the final setup of the SVM. Include here all implementation details. Remember that there are two free parameters: C and τ for the Gaussian kernel. You select the best combination by 10-fold cross-validation (Section 10.2.1 of the course notes). To this end, pick a range of sensible values (say, 10) for each parameter (common practice is to use a range $\{a2^k \mid k = 0, \ldots, 9\}$). Finally, evaluate 10-fold CV scores for all combinations of the matrix.

(III) Results & Discussion In this part, you do the final evaluation of your SVM classifier.

1. For your final grid choice, **include the matrix of 10-fold CV scores in the report, highlighting the winning configuration** (either numbers or a `MATLAB imagesc` plot). The winning configuration is the one with minimal CV score, but if several best entries are close together, pick the one with the smallest C and τ. Use this winning configuration for your final evaluation.
2. Run your algorithm on the full training set, with the parameters found by cross-validation. Plot the SVM criterion as function of SMO iterations (only every 20 SMO steps). Additionally, plot the value of the convergence criterion, based on KKT condition violations (see additional note). For the latter plot, the vertical axis should have a logarithmic scale.
3. For the final classifier after SMO convergence, report the training and test set (zero/one) error.
4. On the common binary subtask, compare the performance of your final MLP classifier versus the final SVM classifier. Produce a **bar plot** comparing the training and the test set (zero/one) errors and make some comments on the results.

6 Report

The report must be written in English and must not exceed 7 pages (use 10pt font). Any pages beyond the limit will not be taken into account. Please note, that you are free to consult text-books, talk to friends, surf the web (but **do not copy code**). You should work in teams of two.

7 Submission and Fraud Detection

There are two submission deadlines for the miniproject. Deadlines are strictly enforced, no exceptions are made. Every member of each team has to attend a *fraud detection* session, where he/she has to discuss details of the submitted code and report with a TA. This is a interview of approximately five minutes, where you should bring your computer and demonstrate that you are the original author of your code and report. **Each student has to attend the session on his/her own. Failure to attend the fraud detection session will result in zero points for the miniproject**.
Teams who submit at the first deadline are free to choose one of the two fraud detection sessions. Clearly indicate your preferred session together with your submission. Teams who submit at the second deadline have to attend the fraud detection session at the second date.

1. Submission deadline HIER!!!; fraud detection session on HIER!!! or on HIER!!!.
2. Submission deadline HIER!!!; fraud detection session on HIER!!!.

Each submission must be clearly marked with:

- Name and Sciper number of each group member
- In case of HIER!!! deadline: Which fraud detection session will each group member attend?

Note that for organisatorial reasons, **both group members have to submit the report on moodle**. Please submit exactly the same documents and files.