Prof. Dr. Michael Griebel
Prof. Dr. Jochen Garcke
Dr. Bastian Bohn
Jannik Schürg

1

LINEAR LEAST SQUARES AND k-NEAREST NEIGHBORS

Send your solutions to this chapter's tasks until October 30th.

DATA ANALYSIS BASICS

In most branches of science, economy and industry the amount of available data has become immense during the recent years. Most of these data do not contain any valuable information at all. However, the differentiation between useful data compared to meaningless "data waste" is seldom straightforward. The phenomenon of the availability of enormous amounts of data and the consequential tasks and problems arising from this are commonly summarized by the term *Big Data*. To meet the different challenges of Big Data, such as describing the useful information in a more compact format (*dimensionality reduction*) or making predictions on unseen data (*machine learning*), many ideas and approaches have emerged. For a thorough introduction into machine learning we refer the interested reader to [3].

In this practical lab, we aim at exploring and implementing several well-known machine learning and data analysis algorithms. Furthermore, we will apply them to real-world data sets to get an intuition on the specific needs in different applications. To this end, you will need to be able to use certain `python` tools/libraries.

Task 1.0. (a) *Make yourself familiar with programming in* Python[1] *and its libraries* NumPy[2] *and* MatPlotLib.[3] *Furthermore, you will need* Jupyter Notebooks[4] *to run the template codes.*

(b) *Have a look at the tutorial notebook on the practical lab website and familiarize yourself with the concept of vectorization, i.e. using operations on whole arrays instead of using loops and operating on single array elements.*

(c) *Create a jupyter-notebook in which you create an array z consisting of* 10000 *random numbers drawn from* $\{0, 1, 2\}$. *Implement two versions of a function which counts the number of appearances of the subsequence* $(2, 0, 1)$ *in z. The first version should work with a loop that*

[1] https://www.python.org/
[2] http://www.numpy.org/
[3] https://matplotlib.org/
[4] https://jupyter.org/

accesses the array z elementwise and makes elementwise comparisons. The second version should be a vectorized one (Hint: The numpy function `logical_and` *might help you), which operates on (almost) the whole array z. Compare the runtime of the two versions.*

CLASSIFICATION AND REGRESSION

One of the most common tasks in machine learning is *supervised learning*. Here a function[5] f is learned from input–output samples. The goal is that not only the sample points—usually called *training data*—are (approximately) fitted by f, but also new data points—usually called *test data* or *evaluation data*—which stem from the same distribution as the training data.

Some specific examples are:

- Identifying handwritten letters or digits.
- Estimate risk of disease from patient data.
- Identify email messages that are spam and those that are not.
- Detect critical failures in industrial facilities.

Let us state a supervised learning problem in a mathematical way: Let Ω, Γ be arbitrary sets. To this end, we assume we are given input data $\mathcal{D} := \{(\mathbf{x}_i, \mathbf{y}_i) \in \Omega \times \Gamma \mid i = 1, \ldots, n\}$ drawn i.i.d. according to some probability measure μ on $\Omega \times \Gamma$. Our goal is to find $f : \Omega \to \Gamma$ such that

$$f(\mathbf{x}_i) \approx \mathbf{y}_i \text{ for all } i = 1, \ldots, n. \tag{1.1}$$

Furthermore, we want $f(\tilde{\mathbf{x}}_i) \approx \tilde{\mathbf{y}}_i$ for $i = 1, \ldots, \tilde{n}$ on a test data set, which is also drawn i.i.d. according to μ. This is called a *regression* problem. We will make the notion of $f(\mathbf{x}_i) \approx \mathbf{y}_i$ precise soon.

In the special case of Γ being finite, in particular if given in categorical form, and if we substitute "$\approx$" by "$=$" above, we call this a *classification* problem.

LINEAR LEAST SQUARES (LLS)

Now, let $\Omega = \mathbb{R}^d$ and $\Gamma \subset \mathbb{R}$. Instead of searching for any function f which fits the input data, we will make a restriction on the *model*, i.e. we assume that f has a certain structure. Let us first consider the most simple, a linear structure of f:

$$f(\mathbf{t}) = \alpha_0 + \sum_{i=1}^{d} \alpha_i \cdot t_i = \underbrace{\left(1\ t_1\ t_2\ \ldots t_d\right)}_{=: \hat{\mathbf{t}}^\top} \cdot \boldsymbol{\alpha}. \tag{LIN}$$

[5] More generally, one could also consider learning a measure modelling the connection between the input–output pairs.

Here, the $d+1$ free parameters, which determine the function f are $\alpha_0, \dots, \alpha_d \in \mathbb{R}$. Our task is now to determine $\boldsymbol{\alpha}$ using the input data $\mathcal{D}$. To this end, we have to reformulate the problem of finding an f which fulfills (1.1) into a mathematical (optimization) problem. To achieve this, we use a so-called *loss function* $\mathcal{L} : (\Gamma \times \Gamma)^n \to [0, \infty]$. One of the most common loss functions is the *quadratic* or *least squares* loss:

$$\mathcal{L}\left((a_1, b_1), (a_2, b_2), \dots, (a_n, b_n)\right) = \frac{1}{n} \sum_{i=1}^{n} (a_i - b_i)^2 .$$

The so-called *linear least squares* algorithm then solves

$$\begin{aligned} &\min_{\text{affine linear } f} \mathcal{L}\left((f(\mathbf{x}_1), \mathbf{y}_1), (f(\mathbf{x}_2), \mathbf{y}_2), \dots, (f(\mathbf{x}_n), \mathbf{y}_n)\right) \\ = &\min_{\text{affine linear } f} \frac{1}{n} \sum_{i=1}^{n} (f(\mathbf{x}_i) - \mathbf{y}_i)^2 \\ = &\min_{\boldsymbol{\alpha} \in \mathbb{R}^{d+1}} \frac{1}{n} \sum_{i=1}^{n} (\boldsymbol{\alpha}^\top \cdot \hat{\mathbf{x}}_i - \mathbf{y}_i)^2. \end{aligned} \tag{1.2}$$

If we let $\mathbf{X} := (\hat{\mathbf{x}}_1\ \hat{\mathbf{x}}_2\ \dots\ \hat{\mathbf{x}}_n)^\top \in \mathbb{R}^{n \times (d+1)}$ be the (modified) *input data matrix* and $\mathbf{y} := (\mathbf{y}_1\ \mathbf{y}_2\ \dots\ \mathbf{y}_n)^\top \in \mathbb{R}^n$ be the vector of outputs, we can rewrite the algorithm as

$$\min_{\boldsymbol{\alpha} \in \mathbb{R}^{d+1}} \frac{1}{n} (\mathbf{X}\boldsymbol{\alpha} - \mathbf{y})^\top (\mathbf{X}\boldsymbol{\alpha} - \mathbf{y}) = \frac{1}{n} \min_{\boldsymbol{\alpha} \in \mathbb{R}^{d+1}} \|\mathbf{X}\boldsymbol{\alpha} - \mathbf{y}\|^2.$$

Since this is a quadratic optimization problem, the optimal coefficients are given by

$$\boldsymbol{\alpha} = \left(\mathbf{X}^\top \mathbf{X}\right)^{-1} \mathbf{X}^\top \mathbf{y}. \tag{LLS}$$

The numerically inclined reader should prefer using a QR factorization.

Let us try this least squares algorithm on some artificial data, which we need to create first. We consider an easy classification example with $\Omega = \mathbb{R}^2$ and $\Gamma = \{0, 1\}$.

Task 1.1. *Create $n = 200$ data points in the following way:*

(a) *Draw ten random i.i.d. samples from the two-variate normal distribution $\mathcal{N}\left((\tfrac{3}{2}\ 0)^\top, \mathbf{I}\right)$ and store them in a `numpy` array* a. *Draw another ten samples according to $\mathcal{N}\left((0\ \tfrac{3}{2})^\top, \mathbf{I}\right)$ and store them in another `numpy` array* b. *Use `MatPlotLib` to make a scatter plot (i.e. plot the points in a 2D coordinate system) of the elements in* a *and the elements in* b *using different colors for the two arrays.*

(b) *Pick 100 equidistributed indices $i_1, \dots, i_{100}$ from $\{1, 2, \dots, 10\}$ and set the j-th data point $\mathbf{x}_j$ to*

$$\mathbf{x}_j := \underbrace{\mathsf{a}[i_j]}_{i_j\text{-th element of } \mathsf{a}} + \varepsilon_j \textit{ for all } j = 1, \dots, 100 \textit{ with } \varepsilon_j \sim \mathcal{N}\left((0\ 0)^\top, \frac{1}{4}\mathbf{I}\right).$$

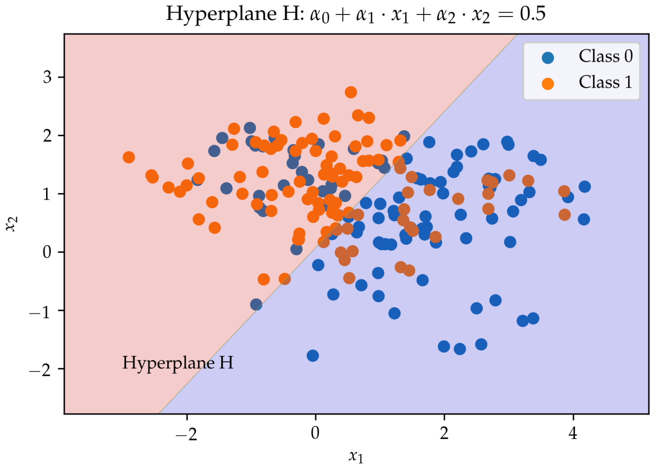
Proceed analogously for $j = 101, \dots, 200$ by substituting a *by* b. *Make a scatter plot for the data points $\mathbf{x}_j$ with $j = 1, \dots, 200$ with different colors for the first 100 points and the second 100 points.*

(c) The first $j = 1, \dots, 100$ data points get the label $\mathbf{y}_j = 0$, the next $j = 101, \dots, 200$ ones get $\mathbf{y}_j = 1$.

Task 1.2. *Implement a linear least squares algorithm, i.e. solve* (LLS)*. Hint: You can use* `numpy.linalg.solve` *to solve a system of linear equations. Apply it to the data from task 1.1. Plot the scattered input data as in step (b) of task 1.1 together with the separating hyperplane, i.e. the contour line given by*

$$\alpha_0 + \alpha_1 x_1 + \alpha_2 x_2 = \frac{1}{2},$$

where x_1 and x_2 denote the coordinates in $\mathbb{R}^2$ (not to be confused with the data $\mathbf{x}_i$). The result should look (approximately) like this:



Quantifying the misclassification error

The separating hyperplane from task 1.2 can be used to divide or *classify* the data into two parts. Let us quantify how good our classifier really is.

Task 1.3. *Build the so-called* confusion matrix *for the data and the hyperplane from task 1.2, i.e. a matrix C, with entries*

$$C_{ij} = \# \{ \text{Points classified as } i, \text{ where the real label is } j \} .$$

In our case this is a 2×2 matrix with $i, j \in \{0, 1\}$ since $|\Gamma| = 2$. Calculate the accuracy $\frac{\operatorname{trace}(C)}{n}$.

As we mentioned above, machine learning engineers are not really interested in building algorithms which only perform well on the training data, but rather in having methods which generalize well to (unseen) test data, which—in the best case—follow the same law/distribution as the training data.

Task 1.4. *Create 10 000 test points for each of the two classes in the same way as you created the training data in step (b) of task 1.1. Evaluate the*

LLS classifier built in task 1.2 on the test data and compute the confusion matrix and the accuracy of the test data. Compare your results to the ones from task 1.3.

Our first "real" data set and Pandas

Next, we will try our LLS classifier on real-world data, namely the *Iris* dataset [1, 2]. The dataset consists of 150 points, which describe three different types of Iris plants. We have three classes

$$\{\text{Iris-setosa}, \ \text{Iris-versicolor}, \ \text{Iris-virginica}\}.$$

The four *features*, i.e. the coordinates in $\Omega = \mathbb{R}^4$ refer to certain length and width measurements of the plants.

We will classify one of the three plant classes against both of the remaining classes by using our LLS algorithm. To this end, we first have to read in the data set and cast the class names to $\Gamma = \{0, 1\}$. Instead of reading in the data by hand, we employ the very useful PANDAS library[6] in python:

```
import pandas as pd
url = 'https://archive.ics.uci.edu/ml/machine-learning-
    databases/iris/iris.data'
irisDataFrame = pd.read_csv(url, header=None)
```

In PANDAS, the data is stored in an instance of DataFrame on which many useful operations can be run.

Task 1.5. *Make yourself familiar with the basics of PANDAS.*

(a) Read in the Iris data set and use the data labels $\mathbf{y}_i = 0$ for the Iris-setosa instances and $\mathbf{y}_i = 1$ for the Iris-versicolor and Iris-virginica classes:

a.1. Run the LLS algorithm by using only the first two dimensions of Ω in the input data, i.e. we only look at the first two features. Plot the scattered data and the separating hyperplane as in task 1.2.

a.2. Now run the LLS algorithm by using all four features/dimensions of the input data. Compute the confusion matrix and the accuracy.

(b) Finally, run the same two steps as in (a), but now try to classify Iris-versicolor instances (label $\mathbf{y}_i = 0$) against both Iris-setosa and Iris-virginica (label $\mathbf{y}_i = 1$). What do you observe?

Another approach to solving the optimization problem

Although (LLS) is the exact solution for the linear least squares approach above, we could pursue a different, iterative approach to solving (1.2). To this end, we consider a *gradient descent* method approach.

[6] https://pandas.pydata.org/

(a) Iris Setosa.[a] (b) Iris Versicolor.[b] (c) Iris Virginica.[c]

Figure 1.1: Pictures of three Iris plants.

[a] Photo by Emma Forsberg from Anchorage, USA.
[b] Photo by Cliff from Arlington, Virginia, USA.
[c] Photo by Christer T. Johansson.

Let $J(\boldsymbol{\alpha}) := \frac{1}{n}\sum_{i=1}^{n}\left(\boldsymbol{\alpha}^{\top} \cdot \hat{\mathbf{x}}_i - \mathbf{y}_i\right)^2$ be the goal functional, which should be minimized. A gradient descent approach corresponds to the following pseduo-algorithm:

```
Initialize alpha randomly
step = 0
while (not converged) and (step < maxSteps):
    alpha = alpha - nu * grad(J(alpha))
    step = step + 1
```

The stepwidth $\nu > 0$ and the maximum number of iterations maxSteps have to be chosen before running this method. A possible convergence criterion is reached if $\nabla J(\alpha)$ is almost zero.

Task 1.6. *(**Master students only**) Implement the gradient descent method and run an LLS algorithm with a gradient descent optimizer for the data from task 1.5 (a.1.). Choose $\nu \in \{1, 10^{-1}, 10^{-2}, \ldots\}$ as the largest value such that convergence is achieved. Create a plot of the value of J vs. the actual iteration number. What do you observe?*

Although it is not very meaningful to use a gradient descent optimizer for our LLS algorithm, we will encounter more elaborate models for the function f later on for which this approach will be more appropriate.

Data normalization

An underestimated pre-processing step in data analysis is *data normalization* or *data scaling*. The way in which a data set is scaled can have significant impact on the outcome and/or the runtime of a machine learning algorithm.

Task 1.7. *(**Master students only**) Normalize the data from task 1.5 (a.1.). To this end, calculate the mean μ_j and the standard deviation σ_j for each feature*

j (i.e. each coordinate direction j of the data set) and set the j-th component of the i-th data point to

$$[\mathbf{x}_i]_j := \frac{[\mathbf{x}_i]_j - \mu_j}{\sigma_j}.$$

Now run the gradient descent LLS algorithm on the normalized data. Again, choose ν as the largest value such that convergence is achieved. Compare the first 100 iteration steps by plotting the value of J vs. the iteration number for both the normalized and the unnormalized case. What do you observe?

k-NEAREST NEIGHBORS

Next, we consider a different regression/classification technique: The so-called *k-nearest neighbor* algorithm. Here, a data point $\mathbf{x}$ from a test set is assigned a specific mean of training data values

$$\mathrm{nearNeigh}_k(\mathbf{x}) = \frac{1}{k} \sum_{\{i \mid \mathbf{x}_i \in N_k(\mathbf{x})\}} \mathbf{y}_i,$$

where $N_k(\mathbf{x})$ are the closest k training points to $\mathbf{x}$. For a two class problem ($\Gamma = \{0, 1\}$), the data point $\mathbf{x}$ would be assigned to class 0 if $\mathrm{nearNeigh}_k(\mathbf{x}) < 0.5$ and to class 1 otherwise. If $\mathrm{nearNeigh}_k(\mathbf{x}) = 0.5$ you can choose which class to assign to $\mathbf{x}$.

Task 1.8. *Implement the k-nearest neighbor algorithm. There are many possible ways to calculate the pairwise distances between data sets. For instance, you can use the* `scipy.spatial.distance` *library. Run the algorithm with $k = 1$, $k = 15$ and $k = 30$ and calculate the confusion matrix and the accuracy for the data set from task 1.1 (using it as training and test data set). Make a scatter plot of the data and plot the separation level set, i.e. the contour of $\mathrm{nearNeigh}_k(\mathbf{x}) = \frac{1}{2}$.*

Task 1.9. *Let us test how the algorithm performs for every possible k.*

(a) Run the k-nearest neighbors algorithm for the data from task 1.1 for all $k = 1, \ldots, 200$ and store the accuracy for each k.

(b) Do the same thing as in step (a) but now use the data created in task 1.4 as test data.

(c) Plot the accuracies from steps (a) and (b) vs. the value of k. What do you observe?

WHAT WE DID NOT COVER...

STOCHASTICS Since we can usually assume that the input data is drawn according to an (unknown) probability distribution, we can formulate the problem of finding an optimal classificator/regressor as a stochastic problem. In this context, especially the so-called *Bayesian* *methods* are commonly used. Here, Bayes' theorem is applied to obtain a solution to the corresponding stochastic optimization problem, see e.g. [3].

LOGISTIC REGRESSION Another famous linear model to obtain optimal classifiers is the *logistic regression* model, where the distribution of the underlying random variables is modelled, see [3]. The approach also involves a different loss function than least squares.

REGULARIZATION Instead of simply minimizing a loss function as in the case of linear least squares, we could add a *regularization* term to the minimization problem. This can be interpreted as a trade-off between minimizing the loss on the training data and obtaining a simple or sparse model, see [3]. Examples for such regularization terms are ℓ_p norms of the coefficients (*Lasso, Tikhonov*) or more complex norms involving derivatives of the minimizer.

REFERENCES

[1] Dua Dheeru and Efi Karra Taniskidou. *UCI Machine Learning Repository*. 2017. URL: http://archive.ics.uci.edu/ml.

[2] R. A. Fisher. "The use of multiple measurements in taxonomic problems." In: *Annals of Eugenics* 7.2 (1936), pp. 179–188. ISSN: 2050-1439. DOI: 10.1111/j.1469-1809.1936.tb02137.x.

[3] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. New York, NY, USA: Springer New York Inc., 2009. URL: https://web.stanford.edu/~hastie/ElemStatLearn/download.html.