Machine Learning 1, SS24
Homework 1
Linear and Logistic Regression. Gradient Descent.

Thomas Wedenig
thomas.wedenig@tugraz.at

Tutor:	Marharyta Papakina, marharyta.papakina@student.tugraz.at
Points to achieve:	25 pts
Bonus points:	4* pts
Deadline:	03.05.2024 23:59
Hand-in procedure:	Use the **cover sheet** that you can find in the TeachCenter. Submit **all python files and a report (PDF)** to the TeachCenter. Do not zip them. Do not upload the `data` and `plots` folders.
Submissions after the deadline:	Each missed day brings a (-5) points penalty.
Plagiarism:	If detected, 0 points for all parties involved. If this happens twice, we will grade the group with "Ungültig aufgrund von Täuschung"
Course info:	TeachCenter, https://tc.tugraz.at/main/course/view.php?id=1648

Contents

General remarks

Your submission will be graded based on:

- Correctness (Is your code doing what it should be doing? Is your derivation correct?)
- The depth of your interpretations (Usually, only a couple of lines are needed.)
- The quality of your plots (Is everything clearly readable/interpretable? Are axes labeled? …)
- Your submission should run with Python 3.9+.

Do not add any additional `import` statements anywhere in the code. **Do not modify the function signatures** of the skeleton functions (i.e., the function names and inputs).

For this assignment, we will use an implementation of Logistic Regression from scikit-learn. The documentation for this is available at the scikit-learn website.

For this class (and all scikit-learn model implementations), calling the `fit` method trains the model, calling the `predict` method with the training or testing data set gives the predictions for that data set (which you can use to calculate the training and testing errors), and calling the `score` method with the training or testing data set calculates the mean accuracy for that data set.

1 Linear Regression – Detection of memristor faults [10 points]

Information in the human brain is processed by *neurons* and stored in the form of *synaptic weights*. *Synaptic plasticity* — the ability to increase or decrease these weights — is the underlying mechanism responsible for knowledge-based learning. For the implementation of learning in neuro-inspired computing chips, nano-scale hardware devices, so-called *memristors*, have been proposed and fabricated. A memristor is a resistor with programmable (adjustable) resistance, hence a suitable candidate that mimics biological synapses. However, due to a number of non-idealities (e.g., fabrication, operational constraints, limited endurance), memristors, when programmed, often show deviations from the intended behavior. *Stuck memristors* do not change their resistance regardless of the magnitude of resistance change. Other memristors underestimate or overestimate the magnitude of resistance change, and we refer to such faulty behavior as *concordant faults*. Some memristors even produce resistance change in the opposite direction of the intended one, and we refer to such faults as *discordant faults*. In addition, when being programmed, the achieved resistance states are always noisy due to *programming noise*. An illustration of a memristor with an ideal behavior, a discordant fault, a stuck fault, and a concordant fault can be seen in Fig. 1A, B, C, D, respectively.

To detect memristor faults, one can use a linear regression model, interpret its parameters and finally, classify the faults. In this exercise, we will use two linear regression models, and decide which one works better.

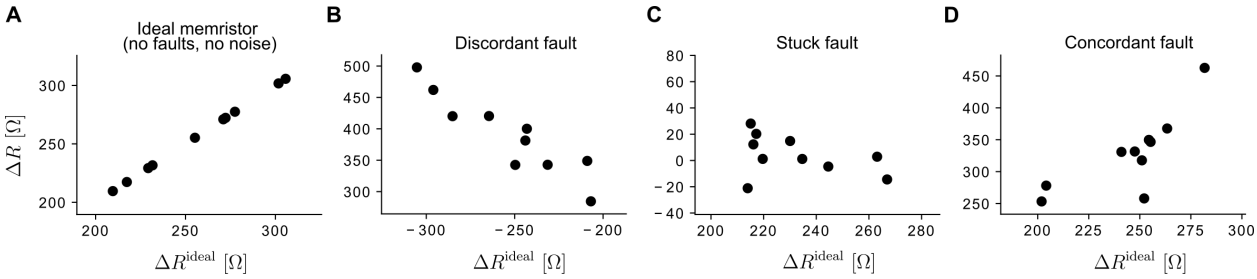

Figure 1: **Examples of memristor faults.** A memristor with (**A**) an ideal behavior (no faults, no noise), (**B**) a discordant fault, (**C**) a stuck fault, (**D**) a concordant fault. Expected resistance changes are given on the $x-$axis ($\Delta R^{\text{ideal}}[\Omega]$), the achieved ones on the $y-$axis ($\Delta R[\Omega]$).

Tasks:

1. **Model 1.** Use a zero-intercept linear regression model, that is, the hypothesis $h_\theta(\Delta R^{\text{ideal}}) = \theta \cdot \Delta R^{\text{ideal}}$, with only one parameter, θ. In this case, the error function to minimize reads as follows:

$$E(\theta) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(\Delta R_i^{\text{ideal}}) - \Delta R_i)^2,$$

where m is the number of data points (measurements) we use for regression, more precisely, $(\Delta R_i^{\text{ideal}}, \Delta R_i)$, $i \in 1, ..., m$ are the points used for fitting.

We wish to find a global minimizer θ^*, i.e.,

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\, E(\theta) \tag{1}$$

Derive a *closed-form analytical solution* for θ^*. (Find the derivative of the error function w.r.t. θ, set it to zero, and express θ^*. Include *all steps* of your derivation in the report.) **NOTE:** Your expression for θ^* should include sums – use the error function as given in this exercise sheet, without transformations.

Implement the equation for θ^* in the code (file `lin_reg_memristors.py`, function `fit_zero_intercept_lin_model`) by using the expressions with sums you have derived.

2. **Model 2.** Use a linear regression model with intercept, that is, the hypothesis $h_{\theta_0,\theta_1}(\Delta R^{\text{ideal}}) = \theta_0 + \theta_1 \Delta R^{\text{ideal}}$. The error function, in this case, is given by:

$$E(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} (h_{\theta_0,\theta_1}(\Delta R_i^{\text{ideal}}) - \Delta R_i)^2,$$

where m is the number of data points (measurements) we use for regression, and $(\Delta R_i^{\text{ideal}}, \Delta R_i), i \in 1, ..., m$ the points used for fitting. Again, we seek

$$(\theta_0^*, \theta_1^*) = \underset{\theta_0,\theta_1}{\operatorname{argmin}} E(\theta_0, \theta_1) \tag{2}$$

Derive a *closed-form analytical solution* for θ_0^* and θ_1^* (which will again include sums). (Find the *partial* derivatives of the error function w.r.t. θ_0 and θ_1, set both of them to zero, and express θ_0^* and θ_1^*. Include *all steps* of your derivation in the report.) **NOTE:** Your expression for θ^* should include sums – use the error function as given in this exercise sheet, without transformations.

Implement the equations for θ_0^* and θ_1^* in the code (file `lin_reg_memristors.py`, function `fit_lin_model_with_intercept`) by using the expressions with sums you have derived.

3. The data set with measurements for 8 memristors is already loaded (file `main.py`, function `task_1`). It contains a multidimensional numpy array with a shape of $(8, 10, 2)$, representing measurements for 8 memristors; for each memristor there are 10 measurements in the form $(\Delta R_i^{\text{ideal}}, \Delta R_i), i \in 1, ..., 10$, i.e., pairs representing an expected (ideal) resistance change, and an achieved resistance change.

   Implement a function call to fit Model 1 to the data (per memristor), and visualize the data and the fit for each memristor. The code that generates and saves the plot is already implemented.

   Implement a function call to fit Model 2 to the data (per memristor), and visualize the data and the fit for each memristor.

   Include all generated plots in the report.

4. Interpret the parameters of Model 1. Based on the parameter θ, how can we decide if there is a discordant, stuck, or concordant memristor fault? Interpret the parameters of Model 2, and state how you would use them to decide on the type of the memristor fault.

5. Which model would you prefer to use – Model 1 or Model 2? Explain your choice.

6. Finally, implement in the code (using if-statements and thresholds of your choice) how the parameters (of either Model 1 or Model 2, depending on your choice in the previous step) can be used to decide on the memristor fault type. In the report, describe how you did it (or include a screenshot of your code). Report (provide a list with) the memristor id and the fault type assigned to it according to your code output.

Bonus tasks [2.5* bonus points]: Reconsider the task of computing θ_0^*, θ_1^* in Model 2. Instead of computing the individual partial derivatives, we will reformulate the objective in matrix-vector notation. Let $\boldsymbol{\theta} = (\theta_0, \theta_1)^T$, $\mathbf{x} = (\Delta R_1^{\text{ideal}}, \ldots, \Delta R_m^{\text{ideal}})^T \in \mathbb{R}^m$ and $\mathbf{y} = (\Delta R_1, \ldots, \Delta R_m)^T \in \mathbb{R}^m$. We will define a matrix $\mathbf{X}$ such that we can re-write the minimization objective as

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} E(\boldsymbol{\theta}) \qquad \text{with} \quad E(\boldsymbol{\theta}) = \frac{1}{m} \|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2$$

Write down the definition of the design matrix $\mathbf{X}$ such that this objective is equivalent to Equation 2 and state the dimensions of $\mathbf{X}$. Using matrix calculus, derive an analytical expression for the solution $\boldsymbol{\theta}^*$ by computing the gradient of the error w.r.t. $\boldsymbol{\theta}$ and setting it to the zero vector, i.e., $\nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta}^*) = \mathbf{0}$. Your final expression for $\boldsymbol{\theta}^*$ should involve the Moore-Penrose pseudoinverse. Implement the computation for $\boldsymbol{\theta}^*$ in the function `bonus_fit_lin_model_with_intercept_using_pinv` by making use of numpy's `pinv` (read: pseudoinverse) function. You can test the functionality of your implementation by calling `bonus_fit_lin_model_with_intercept_using_pinv` instead of calling `fit_lin_model_with_intercept` in `main.py`. The resulting $\boldsymbol{\theta}$ should be equivalent. Clearly show all steps of your derivation in your report.

2 Logistic Regression [7 points]

For this task we will use 3 different data sets (`X-1-data.npy` should be used with `targets-dataset-1.npy`, and `targets-dataset-2.npy`, `X-2-data.npy` with `targets-dataset-3.npy`), and `sklearn` library.

`X-1-data.npy` contains two features – values for x_1, $x_2 \in \{0, 1, ..., 29\}$. `X-2-data.npy` contains two features – values for $x_1 \in [-2, 2]$, and $x_2 \in [-1.33, -0.14]$. The targets for all data sets are 0 or 1. The goal of this exercise is to train a classifier that predicts either Class 0 or Class 1, as shown in Fig.2A-C.

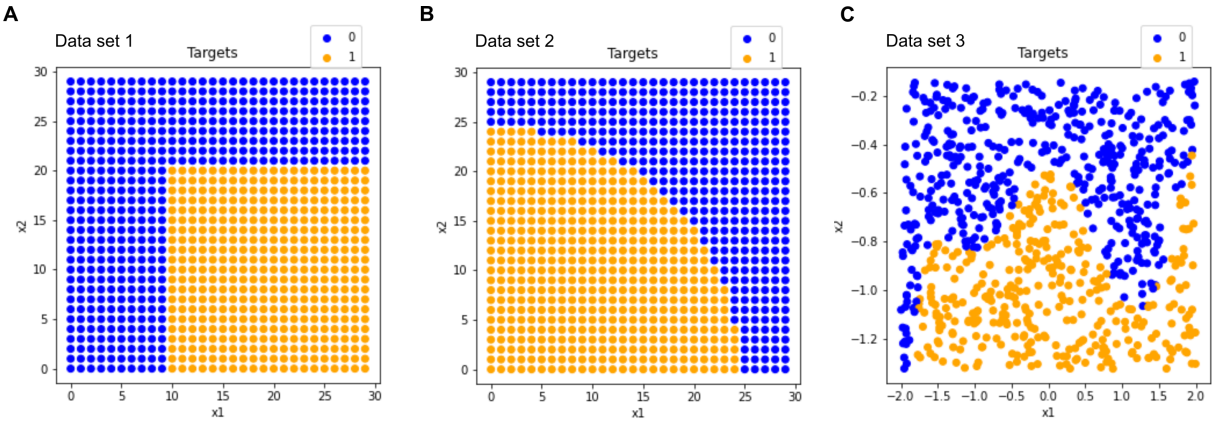

Figure 2: **Targets for: (A) Data set 1; (B) Data set 2; (C) Data set 3.**

Tasks:

1. For each of the three tasks, load the data, then create an appropriate design matrix X. Include any feature that you think is necessary. (There is no need to include a *constant* feature (e.g., a vector consisting of ones), because we will use `LogisticRegression` classifier from the `sklearn` library, for which, by default, the bias term (intercept) is added to the decision function internally.) In the report, for each task, state what design matrix you used, that is, name the features of your design matrices.

2. Split the data set, such that 20% of the data is used for testing. Use the `train_test_split` function (already imported) with the parameter `random_state=0`.

3. The code skeleton then creates a classifier with your parameters (`LogisticRegression` classifier). Fit the model to the data, then calculate accuracy on the train and test set. In addition, using `log_loss` from `sklearn.metrics`, calculate the *cross-entropy* loss on the train and test set. As the cross-entropy loss is a measure of dissimilarity between two distributions, you will first need to calculate the model's output probabilities for each data point in the train and test set.

   Try out different regularization norms ("penalties"). Check the documentation to see what options there are and report your final choice (the one that gives you the best accuracy on the test set). If you are not happy with the final results, and changing the penalty does not help, rethink your design matrices!

   Report the accuracy on the train and test set, the cross-entropy loss on the train and test set, and what penalty you used.

4. For each data set, include 2 plots in the report that you generated using the function `plot_logistic_regression` (one for the train set and one for the test set, respectively). These plots show the decision boundary of the trained classifier and the data points from the respective sets.

5. Report the parameters of your model (including the bias term). Hint: check the classifier's attributes

6. Assume we have trained a logistic regression classifier (binary classes) and are given a test dataset $\mathcal{D}$. Is the following statement correct? "If the classifier predicts the correct class for *all* elements in $\mathcal{D}$ (100% accuracy), then it follows that the cross-entropy loss (w.r.t. $\mathcal{D}$) is 0." Explain your reasoning.

3 Gradient descent [8 points]

Consider the following function $f : \mathbb{R}^2 \to \mathbb{R}$ (called the *Ackley* function):

$$f(x, y) = -20e^{-0.2\sqrt{0.5(x^2+y^2)}} - e^{0.5(cos(2\pi x)+cos(2\pi y))} + e + 20. \tag{3}$$

We want to minimize this function using the *Gradient Descent* algorithm, i.e., we wish to find

$$(x^*, y^*) = \underset{x,y}{\operatorname{argmin}} f(x, y).$$

The global minimum of this function is at the point $(0, 0)$ and you should be able to find it by running the Gradient Descent algorithm.

Tasks:

1. In the code, implement the gradient descent algorithm (function `gradient_descent`) with a *decaying learning rate*: In this setting, the learning rate η_t depends on the iteration count t. Specifically, set

   $$\eta_0 \leftarrow \texttt{learning\_rate}, \qquad \eta_t \leftarrow \texttt{lr\_decay} \cdot \eta_{t-1}$$

   where `learning_rate` is the initial learning rate and `lr_decay` $\in (0, 1]$.

2. In the code, implement the Ackley function (function `ackley`).

3. Using pen and paper, calculate the partial derivatives of f (with respect to x and y), i.e., $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$. Implement the expressions you have derived in the code (function `gradient_ackley`). Include all steps of your derivation in the report.

4. Choose initial points x_0, y_0 by sampling them from a standard normal distribution, i.e.,

   $$x_0 \sim \mathcal{N}(0, 1),\ y_0 \sim \mathcal{N}(0, 1)$$

   where $\mathcal{N}(0, 1)$ denotes a normal distribution with zero-mean ($\mu = 0$) and unit variance ($\sigma^2 = 1$). To make your code reproducable, we set numpy's random seed to a constant (which will result in the same samples x_0, y_0 in every run of your script).

5. Choose hyperparameters of the gradient descent algorithm (i.e., number of iterations, learning rate, and learning rate decay) and minimize the Ackley function. Report the parameters that you have used.

6. Generate a plot showing how the cost changes over iteration. Include it in the report.

7. Why is this function challenging to optimize?

8. Briefly discuss what happens when you set the parameter `lr_decay` to 1.0 (i.e., the learning rate stays the same in all iterations). Why is it very helpful to have a learning rate that decays when optimizing the Ackley function?

Bonus task [1.5* points]:

- Let $g : \mathbb{R} \to \mathbb{R}$ be a function whose derivative (w.r.t. its input) $\frac{dg}{dx}$ exists everywhere. To minimize g using the Gradient Descent algorithm, we must have access to the derivative of the function at an arbitrary point. Assume that we do not know the functional form of g, but we can only *evaluate* it at arbitrary points[1] (i.e., we can compute $g(x)$ for any $x \in \mathbb{R}$). How can we *approximate* $\frac{dg}{dx}\Big|_{x=x'}$ for an arbitrary point $x' \in \mathbb{R}$? Write down the definition of the derivative and explain how such a general approximation can be computed.

[1] We effectively treat g as a black box, which makes this method very general.