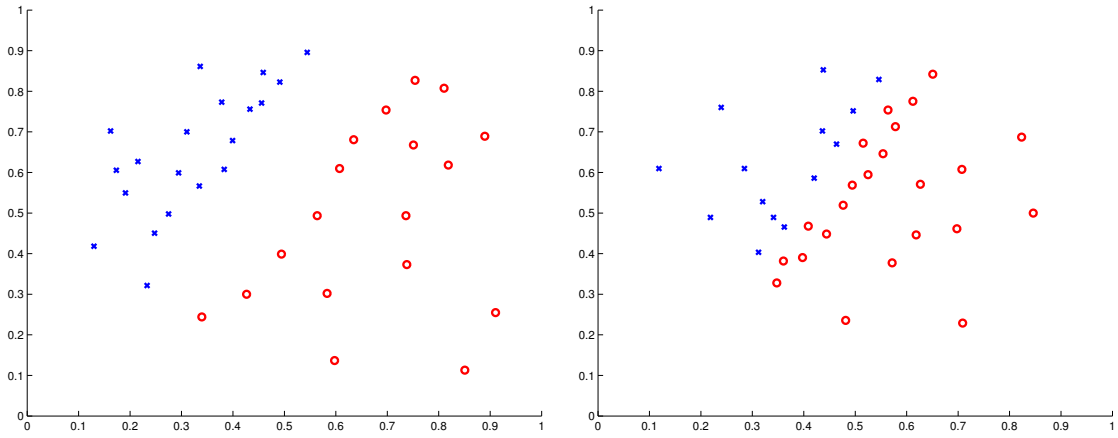
George Lydakis <lydakis@vision.rwth-aachen.de>
Kadir Yilmaz <yilmaz@vision.rwth-aachen.de>
Julia Berger <berger@vision.rwth-aachen.de>

Exercise 2: Least Square Linear Classifiers, Linear Regression

due **before** 2025-11-24

Important information regarding the exercises:

- The exercises are not mandatory. Still, we strongly encourage you to solve them! All submissions will be corrected. If you submit your solution, please read on:
- Use the Moodle system to submit your solution. You will also find your corrections there.
- Due to the large number of participants, we require you to submit your solution to Moodle **in groups of 3 to 4 students**. You can use the **Discussion Forum** on Moodle to organize groups.
- If applicable submit your code solution as a zip/tar.gz file named `mn1_mn2_mn3.{zip/tar.gz}` with your **matriculation numbers** (`mn`).
- Please do **not** include the data files in your submission!
- Please upload your pen & paper problems as PDF. Alternatively, you can also take pictures (.png or .jpeg) of your hand written solutions. Please make sure your handwriting is legible, the pictures are not blurred and taken under appropriate lighting conditions. All non-readable submissions will be discarded immediately.



(a) Training data (b) Test data

Figure 1: Datasets for linear classifier

Question 1: Least Square Linear Classifier ..... $(\Sigma = 5)$

Train a least-squares linear classifier on the 2D training data (c.f. Figure 1a) and test it on the training and test set (c.f. Figure 1b). The data for this exercise are stored in the files `lc_train_data.dat`, `lc_train_label.dat`, `lc_test_data.dat` and `lc_test_label.dat`. To this end write two functions:

(a) The function **(2 pts)**

```python
def leastSquares(data, label):
    # Sum of squared error shoud be minimized
    #
    # INPUT:
    # data        : Training inputs  (num_samples x dim)
    # label       : Training targets (num_samples x 1)
    #
    # OUTPUT:
    # weights     : weights   (dim x 1)
    # bias        : bias term (scalar)
    return weight, bias
```

that trains a least-squares classifier based on a data matrix `data` and its class label vector `label`. It provides as output the linear classifier weight vector `weight` and bias `bias`.

(b) The function **(2 pts)**

```python
def linclass(weight, bias, data):
    # Linear Classifier
    #
    # INPUT:
    # weight      : weights                (dim x 1)
    # bias        : bias term              (scalar)
    # data        : Input to be classified (num_samples x dim)
    #
    # OUTPUT:
    # class_pred  : Predicted class (+-1) values  (num_samples x
        1)
    return class_pred
```

that classifies a data matrix `data` based on a trained linear classifier `weight, bias`.

(c) Run the script `apply`. This function loads the train and the test datasets. It first trains the linear classifier on the training data and then applies it on both the training and the test datasets. Analyze the classification plots for both the datasets. Are the sets optimally classified? Explain! **(1 pt)**

Question 2: Linear Regression ..... $(\Sigma = 8)$

In this exercise, we will investigate the use of linear regression as a method for fitting mathematical models to real-world data. More specifically, we are interested in finding a model that describes the yield of a particular crop (typically measured in tons per hectare) as a function of the rainfall over a growing season (typically measured in centimeters or millimeters). Up to some point, more rainfall translates to improved crop yields, but after some level overwatering can actually decrease yield by causing issues such as root rot. In this exercise, our dataset consists of 300 training data samples, each of which corresponds to a measurement of crop yield and associated rainfall level. To evaluate the quality of the models, the dataset includes 50 validation data samples.

Note: The resulting figures will feature negative values for both the independent and dependent variable. This is due to the fact that we *normalize* our data such that the training dataset has zero mean and unit standard deviation. This is standard practice in machine learning and improves the numerical stability of the algorithms that are used. If we wanted to use our models for an actual prediction of crop yield, we would have to keep this in mind by a) properly normalizing our waterfall level with the training data statistics b)

"de-normalizing" the model's output to obtain a number that makes sense when interpreted as crop yield.

(a) The simplest model we'd like to use is standard linear regression: assuming $x \in \mathbb{R}$ is the level of rainfall and $y \in \mathbb{R}$ the crop yield, we want to find $w, b \in \mathbb{R}$ such that $y \approx wx + b$. To this end, fill in the function `fit_linear_model` of `modeling.py` by implementing the closed-form solution to linear least-squares regression discussed in the lecture: **(2 pts)**

$$\mathbf{w} = (\mathbf{X}^\intercal \mathbf{X})^{-1} \mathbf{X}^\intercal \mathbf{y},$$

where the i-th row of $\mathbf{X} \in \mathbb{R}^{N \times D}$ equals the features for the i-th D-dimensional training sample (from a total of N samples), with an additional prepended coordinate equal to 1 ("bias trick") and the i-th row of $\mathbf{y} \in \mathbb{R}^N$ equals the expected output for the i-th sample. The accompanying code comments further clarify what this function expects as input and what it should yield as output.

Note: If you already want to test your code at this point, see the last part of this exercise and set `max_degree = 0`, which will not fit any polynomial models. In addition, have `fit_linear_model_with_ridge_regression` simply call `fit_linear_model` and return its result.

(b) We are also interested in fitting more complicated models that can express non-linear relationships. As you have seen in the lecture, this is still possible with linear regression, as long as one uses an appropriate set of *basis functions*. Here we will use the polynomial basis functions up to degree M, which, for our single-dimensional input x, result in: **(2 pts)**

$$\phi(x) = (1, x, x^2, \ldots, x^M)$$

Fill in the function `compute_polynomial_basis_funcs` that computes this quantity, excluding the constant first coordinate which is already handled by `fit_linear_model` . Once again, take a look at the accompanying code comments for an exact specification of the input and output.

Observe, in `main.py`, how we still use the *same function* as in (a) to perform the actual regression: the only thing that changes is the *space* in which linear regression is performed, which is now M-dimensional instead of one-dimensional.

(c) You have seen in the lecture that regularization of models becomes important as their complexity increases (here the complexity of our model is determined by the maximum polynomial degree being used). Regularization serves to penalize models that are "too complex" and are more likely to result in overfitting. The particular type of regularization being used here is l-2 regularization. The combination of linear least-squares regression with l-2 regularization is also known as *ridge regression*. **(1 pt)**

Recall from the lectures that the corresponding closed-form solution is very similar to the one we previously implemented:

$$\mathbf{w} = (\mathbf{\Phi}^\intercal \mathbf{\Phi} + \lambda \mathbf{I})^{-1} \mathbf{\Phi}^\intercal \mathbf{y},$$

where we have used $\mathbf{\Phi}$ to refer to the result of applying the polynomial basis functions to our input $\mathbf{X}$ (in the case of simple linear regression, $\mathbf{\Phi} = \mathbf{X}$). λ is the tunable hyperparameter that controls the "strength" of regularization, i.e., how heavily should model complexity be penalized.

Fill in the function `fit_linear_model_with_ridge_regression` of `modeling.py` to implement this (again, take a look at the comments in the code).

(d) If everything has been implemented correctly, you can now run `main.py` from your command line and experiment with various settings. There are three tunable parameters: **(3 pts)**

- `ridge_lambda` is the λ hyperparameter being used in ridge regression.
- `num_training_samples` is the number of training samples to use during fitting. The maximum value for this is 300, and decreasing it corresponds to using smaller training datasets.
- `max_degree` is the maximum degree that will be used for the polynomial basis functions. Namely, for `max_degree` $= M$ the code will perform linear regression for a polynomial basis with degree $1, 2, \ldots M$ (degree 1 corresponds to simple linear regression) and will yield M different models.

The script displays a `matplotlib` figure that shows the training samples that were used, the validation samples on which we evaluate the models, and the graph of each model that was fitted to the data. The legend also shows the mean square error that the model achieves on both its training and validation sets. Recall that overfitting means high validation but low training error, and underfitting means low training and low validation error. Reflect on the following questions:

- What happens as we keep the number of training samples being used constant and we increase the maximum degree of the polynomial being fitted?
- How does increasing λ for ridge regression affect the results? Intuitively, what do you expect will happen as $\lambda \to \infty$? Do the resulting plots validate your intuition as you increase `ridge_lamdba`?
- Experiment with varying the number of training samples being used. What do you observe with respect to the relationship between the "optimal" model complexity and the number of training samples?