B1 Engineering Computation - Project

Optimization for classification models

Summary Description

Michaelmas Term, 2025

1 Summary Description

This project investigates how to use optimization to learn parameters of a model so that it can differentiate between data that belong to different classes (categories). We will focus on the *binary classification* task, where data belong in *one out of two* possible *classes*. There are multiple examples of classification tasks in the real world, such as categorizing a patient as healthy or unhealthy (2 classes) based on measurements from their blood tests.

Each data point, also called a *sample* from the data, is represented by a multi-variate input vector $\mathbf{x} = (x^{(1)}, ..., x^{(d)})$. This vector describes characteristics of the specific sample and is often called a *feature-vector* (blood measurements in previous example). Each sample is also associated with a variable y that we want to predict, which is called the *target* variable. For each sample $\mathbf{x}$, the corresponding y can take one of two possible values, defining to which class the sample $\mathbf{x}$ belongs. To perform *binary classification*, we need the model to approximate the true function that maps $\mathbf{x}$ to the categorical variable y according to which the data was generated (e.g. the true relation between blood measurements and disease in our example). This is quite similar to the *Regression* task that you learned in B1 lectures, where a model is optimized to predict a continuous value y, instead of a categorical variable y.

The functions that map input characteristics to target variables can be very complex. Therefore variety of classification models have been developed. These models have multiple parameters $\boldsymbol{\theta}$. The model represents a different function $f_{\boldsymbol{\theta}} : \mathbf{x} \rightarrow y$ for different values of $\boldsymbol{\theta}$. For a model to estimate well the "true" function f that maps $\mathbf{x}$ to y, we need to find appropriate values for parameters $\boldsymbol{\theta}$. Models used in complex problems can have a large number of parameters (billions for advanced models). Therefore optimal parameter values cannot be found via "manual search" or intuition. This is where optimization is an invaluable tool. Given some *training data* for a task of interest, such as multiple pairs of corresponding $\mathbf{x}$ and y values, computational optimization methods find the optimal parameters $\boldsymbol{\theta}^\star$ for which the model can predict y with minimum error when given $\mathbf{x}$. The model with optimal parameters $\boldsymbol{\theta}^\star$ can then be applied to new data $\mathbf{x}$, for which real y is assumed unknown, in order for the model to predict y. This framework of approximating a function with a model, where the model parameters are learned via computational optimization using training data, is Machine Learning.

In this project, we will take a small step in this exciting field and investigate Linear and Non-Linear classification models. To classify each input $\mathbf{x}$ into one of two categories, Linear models learn a linear *decision boundary* – a line separating samples from the two categories. This is based on applying a linear transformation of the form $\mathbf{x}^T\mathbf{w} + b$ to the input, where $\boldsymbol{\theta} = (w^{(1)}, ..., w^{(d)}, b)$ are model parameters. We will study the method of Logistic Regression for learning a linear classifier. Most types of data, however, cannot be accurately categorized via a linear decision boundary. Therefore we will also study how to learn a *non-linear* decision boundary, by extending Logistic Regression with higher-degree polynomials of input features.

The project will use **synthetic data**. You are given a file that contains a pre-implemented function that generates data for training and evaluating models that you will implement. The function generates 2-dimensional data, with each sample being a feature vector $\mathbf{x} = (x^{(1)}, x^{(2)}) \in \mathbb{R}^2$. The generated data come from two possible classes. Data from *each class* are generated from a different multi-modal distribution, shown in Fig. 1. For each sample $\mathbf{x}$, the function also returns the associated "class" label y, taking values 1 or 2. Using this data we will train models such that if they are given a new $\mathbf{x}$ but not the corresponding class label y, they will be able to predict it.

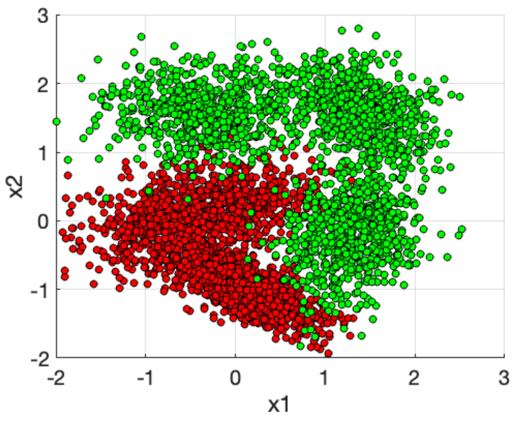


Figure 1: Synthetic data used for the project. You will train models that can separate samples from the two classes.

You are given **the task** of implementing linear and non-linear models, where the latter process higher-degree polynomials of the feature vectors $\mathbf{x}$. These are similar to models you saw in B1 lectures for linear and non-linear regression, but adapted for the classification task. You will also implement Gradient Descent (GD) to optimize these models. You can find details about these models and optimization in **Section 2** of the accompanying ***Detailed Description*** of the project.

Your **task includes** analysing aspects of the models and their training. For this, you are asked to: implement training/validation splits and perform experiments for configuring hyper-parameters of the optimization and the model, plot and analyse training loss and convergence, discuss considerations about algorithm runtime, learn how to handle noise in the data and how it affects training of a model, and analyse overfitting. **Exact specification** of the tasks you are expected to complete can be found in **Section 3.2** of the accompanying ***Detailed Description*** of the project.

Note that although this synthetic data points have no semantic meaning, similar models and analysis are applicable to variety of binary classification tasks in real world, such as classifying whether a patient is healthy or not based on their clinical tests. The synthetic data are designed to enable light-weight experiments that **can run on your personal laptops or desktops** without need for high-end hardware or GPU computation.

To assist you in implementing these models and perform the analysis, you are given a second file, that contains a skeleton code, wherein you can implement the models, gradient descent and other functions needed for evaluation and analysis. The project can be completed **either in Matlab or Python**, whatever you prefer. To facilitate this, we provide you with **two versions of the files** for the data generating function and the code skeleton, one per programming language.

For further information please see the **Detailed Description** of the project.

B1 Engineering Computation - Project

Optimization for classification models

Detailed Description

Michaelmas Term, 2025

1 Dataset

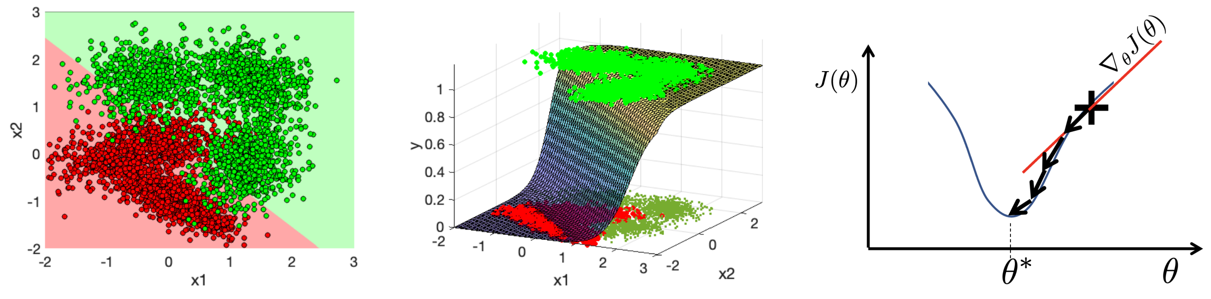
You are given a Matlab (*create_data.m*) and a Python (*create_data.py*) file that implement the data-generating function *create_data(n_samples)*. They are equivalent, so choose the coding language you wish. This function will be used throughout the project to generate data for training and validation. It takes as input the number of samples *n_samples* the user wishes to create. It then returns samples from two possible classes, half samples from each.

In this project we will use 2-dimensional (2D) data, thus each input sample is a feature vector $\mathbf{x} = (x^{(1)}, x^{(2)}) \in \mathbb{R}^2$. We assume that data from *each class* follow a *multi-modal distribution*: Distribution of Class-1 is made by a combination of two 2D Gaussians, each with different mean and covariance, while distribution of Class-2 is made by three 2D Gaussians. This is shown in Fig. 1a. Therefore, to sample a data-point, the function creates a 2-dimensional feature vector $\mathbf{x}$ by first selecting one of the two classes, and then samples $\mathbf{x}$ from one of the 2D Gaussians that make up the distribution of the specific class. For each sample $\mathbf{x}$, the function also returns the associated "class" label, taking values 1 or 2, representing from which class the sample $\mathbf{x}$ came. This class label is what we hope to predict with our classification models, given $\mathbf{x}$, i.e. learn to separate samples generated by the two multi-modal distributions. We recommend you to inspect the function *create_data* to better understand the data generation.

Semantic meaning of the data: We here use synthetic data, without real semantic meaning, to enable us to study properties of optimization and classification models. However, for an intuitive example of what such data could represent, one could imagine that the 2 classes represent 2 types of patients in a hospital (healthy and unhealthy) and the features $(x^{(1)}, x^{(2)})$ represent 2 measurements from the patient's blood tests. Thus the objective of a classification model is to predict whether a patient is healthy or unhealthy based on blood test $\mathbf{x}$.

In a real-world setting, you would not have available the actual function that generates data (here *create_data*). Instead, you would collect data. In the example of patients, you could obtain historic hospital records which document blood tests $\mathbf{x}$ and the "true" class of the corresponding patient, i.e. if they were found healthy or unhealthy. You would then train a model on this historic data and deploy it in a clinic, so that for new patients, it would make predictions about their health to inform doctors and accelerate their diagnosis. To study and develop optimization and machine learning models, however, we often study simpler problems, such as herein, where we define the data-creation process ourselves, with properties that we pre-define (e.g. the Gaussians that create our data). We then try to develop models that can learn from this data and solve this synthetic problem, to study the algorithm properties.

Training and validation data: The below will become clearer when working on the following tasks, but here we provide an overview. We assume we have specific number of data samples. We can use them as *training* data for optimization, to learn model parameters $\theta^\star$ that minimize an error on training data. Then, we wish to assess how well these model parameters



(a) (b) (c)

Figure 1: (a) Samples from the two multi-modal distributions that generate data for each class (Red is Class-1, green is Class-2). Distribution of Class-1 is made of 2 Gaussians, while Class-2 consists of 3 Gaussians, each with different mean and co-variance. Each sample is a 2-dimensional feature vector $\mathbf{x} = (x^{(1)}, x^{(2)})$. Decision boundary of a linear model is also shown. (b) Target labels y for binary Logistic Regression take values $\in \{0, 1\}$, corresponding to the 2 classes, denoting true class of sample $\mathbf{x}$. Logistic Regression fits a sigmoid function to the data, predicting $\bar{y} = \sigma(\mathbf{x}^T\mathbf{w} + b) \in [0, 1]$. $\bar{y}$ can be interpreted as probability predicted by the model that $\mathbf{x}$ is of class 2, and $1 - \bar{y}$ the probability of class 1. (c) Demonstration of Gradient Descent. At each iteration, the gradient of the loss is computed, and a step is taken in the opposite direction, until the process converges to local minimum where we find optimal parameters $\boldsymbol{\theta}^\star$.

would 'generalize' on new, unseen data, e.g. if the model was deployed in the world to predict the category of a sample $\mathbf{x}$, when its class (category) is assumed unknown. To assess this, we commonly collect (here we create) a separate *validation* dataset, for which we also know the correct class of the samples. We do not optimize parameters using this validation data – we only use them for evaluating performance. We use the model that was already optimized on the training data to make predictions on the validation data, and measure the error between model predictions and the correct class. This also allows us to compare what hyper-parameters for the optimisation process lead to best training. Why don't we simply assess performance on the training data? Because optimization may have found parameters optimal for training samples but they may be sub-optimal for new validation data (over-fitting).

2 Classification models and their optimisation

We start with a general definition of important terms. We follow with defining Logistic Regression, a method for linear classification. We then describe how Gradient Descent (GD) can be used to optimize model parameters. Finally, we describe how the method can be extended with polynomials of input features to model non-linear decision boundaries for classification.

2.1 Overview and mathematical definitions

We define input variable $\mathbf{x} = (x^{(1)}, ..., x^{(d)}) \in \mathbb{R}^d$, which we will often refer to as the input *feature vector*. Data of this project are 2-dimensional, $d = 2$, but we keep notation general for now. $\mathbf{x}$ is a *column* vector that represents d real-valued features $x^{(i)}$ (characteristics) about an input data point. We also define $y \in \mathbb{R}$, which represents the "true" value of a *target* variable that we are interested to estimate. We have available a *training database* $D_{tr} = \{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)\}$ with n samples, where for the i-th sample we have a pair of feature vector $\mathbf{x}_i$ and the corresponding value of the target variable y_i. In this project, feature vectors $\mathbf{x}$ have dimensionality $d = 2$ (Sec. 1). Our goal is to learn a model that can predict what value y takes when we are given specific values for $\mathbf{x}$.

2.2 Binary Classification with Logistic Regression

We here explore a *binary classification task*. In this setting, each data-point $\mathbf{x}$ is associated with one of two possible *classes*. For example in Fig. 1a, classes would be $c = \{1, 2\}$ corresponding to red and green samples respectively. To solve a classification task, we wish to make a model that can predict for each sample $\mathbf{x}$ which class it belongs to. In other words, we want to model a *decision boundary* in space defined by $\mathbf{x}$, such that the boundary separates samples from each class.

Logistic Regression is one of the possible methods for learning a linear decision boundary for classification. We define a model parameterized by $\boldsymbol{\theta}$, which predicts the *probability* $\bar{y} \in [0, 1]$ that a sample belongs to class $c = 2$ and, indirectly, $1 - \bar{y}$ the probability it belongs to class $c = 1$. For a data sample $\mathbf{x}_i$, the corresponding *true* value of the target variable y_i will have value 1 if *true* class of $\mathbf{x}_i$ is $c = 2$, and $y_i = 0$ if true class of $\mathbf{x}_i$ is $c = 1$. This is shown in Fig. 1b. The classification model of Logistic Regression is then given by:

$$\bar{y} = \sigma(\mathbf{x}^T\mathbf{w} + b) = \frac{1}{1 + e^{-\mathbf{x}^T\mathbf{w} - b}} \tag{1a}$$

$$= \sigma(\hat{\mathbf{x}}^T\boldsymbol{\theta}) = \frac{1}{1 + e^{-\hat{\mathbf{x}}^T\boldsymbol{\theta}}} \tag{1b}$$

The model has parameters $(\mathbf{w}, b)$, where $\mathbf{w} \in \mathbb{R}^d$ represents a column vector that holds the *weights* of the model, and $b \in \mathbb{R}$ is called the *bias*. We therefore have $\mathbf{x}^T\mathbf{w} + b = x^{(d)}w^{(d)} + ... + x^{(1)}w^{(1)} + b$. These can be jointly represented as the column vector $\boldsymbol{\theta} = (w^{(1)}, ..., w^{(d)}, b)$ that holds all parameters of our model (weights and bias). If we now extend $\mathbf{x}$ with the value 1, defining $\hat{\mathbf{x}} = (x^{(1)}, ..., x^{(d)}, 1)$, we have $\hat{\mathbf{x}}^T\boldsymbol{\theta} = \mathbf{x}^T\mathbf{w} + b$. Representing all parameters as θ is often more convenient and we will use it throughout this project, while we will refer back to weights and biases when it is meaningful to separate them.

In Eq. 1, we first apply a linear transformation $\mathbf{x}^T\mathbf{w} + b$ to the input, which is similar to linear regression you have been taught. We further apply the *sigmoid function* $\sigma(\cdot)$, to "limit" the output value range of $\bar{y}$ between 0 and 1. The value of $\bar{y}$ can then be interpreted as predicted probability of a class. For the input $\mathbf{x}$, we then **obtain the predicted *class label*** ($\{1, 2\}$ in binary classification) by simply assigning class label $c = 2$ if $\bar{y} > 0.5$ and $c = 1$ otherwise. The decision boundary of Logistic Regression is a line, and its function can be found by setting $\bar{y} = 0.5$ in Eq. 1a and deriving the slope and intercept of a line as a function of $\mathbf{w}$ and $\mathbf{b}$. Such a decision boundary that attempts to separate the two classes can be seen in Fig. 1a.

It is often convenient to use matrices to refer to the application of a model on a whole database that includes n samples. For this, we define the *"Design Matrix"* $\mathbf{X} \in \mathbb{R}^{n \times d}$, which has one row for each of the n samples, and one column for each of the d features. Respectively, we define its extended version $\hat{\mathbf{X}} \in \mathbb{R}^{n \times (d+1)}$, which has an additional column filled with values 1, and its i-th row is $\hat{\mathbf{x}}$, to enable joint treatment of $\mathbf{w}$ and b with $\boldsymbol{\theta}$. Respectively, we define the column vector $\mathbf{y} \in \mathbb{R}^n$, where the i-th element is the target value y_i that corresponds to the i-th sample $\mathbf{x}_i$ and the i-th row of the design matrix. With this matrix notation, the predicted classification probabilities for all n samples are calculated as:

$$\bar{\mathbf{y}} = \sigma(\hat{\mathbf{X}}\boldsymbol{\theta}) = \frac{1}{1 + e^{-\hat{\mathbf{X}}\boldsymbol{\theta}}} \tag{2}$$

where $\bar{\mathbf{y}}$ is a vector with n class probabilities, one per sample. The above is particularly useful for matrix-based implementation in Matlab or Python.

2.3 Loss, Cost and Objective functions of Optimization

We need to optimize parameters $\boldsymbol{\theta}$ such that model prediction $\bar{y}_i$ matches the true target y_i for each sample $\mathbf{x}_i$ as closely as possible. For this we use the training data, for which the true labels are known, and an optimization method that will find model parameters that lead to minimum prediction errors. First, we need to define a **loss function** that quantifies the error in the prediction for a specific training sample. For Logistic Regression we use the following loss, often called the *Log-Loss*:

$$\begin{aligned}\mathcal{L}_C(\boldsymbol{\theta}, \hat{\mathbf{x}}, y) &= -y\log(\bar{y}) - (1-y)\log(1-\bar{y}) && \text{(3a)}\\ &= -y\log(\sigma(\hat{\mathbf{x}}^T\boldsymbol{\theta})) - (1-y)\log(1-\sigma(\hat{\mathbf{x}}^T\boldsymbol{\theta})) \in \mathbb{R} && \text{(3b)}\end{aligned}$$

We then define the **cost function**, which is the *mean loss* over our training samples, quantifying the average error in our predictions. We will use the *Mean Log-Loss* as cost function:

$$J_{\mathrm{C}}(\boldsymbol{\theta}, \hat{\mathbf{X}}, \mathbf{y}) = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}_{\mathrm{C}}(\boldsymbol{\theta}, \hat{\mathbf{x}}_i, y_i) \tag{4}$$

This cost J_C will serve as the *objective function* of our optimization method, which will seek optimal model parameters $\boldsymbol{\theta}^\star$ that minimize it. We write this as:

$$\boldsymbol{\theta}^\star = \arg\min_{\boldsymbol{\theta}} J_C(\boldsymbol{\theta}, \hat{\mathbf{X}}, \mathbf{y}) \tag{5}$$

2.4 Optimization with Gradient Descent (GD) and Evaluation

Finding the minimum of this cost function J_C does not admit to an analytical solution. It is differentiable though and thus we can use Gradient Descent that you have seen in class.

The Gradient Descent algorithm is demonstrated in Fig. 1c. Consider a model with parameters $\boldsymbol{\theta}$ and an objective function J that we wish to minimize. Gradient descent *initializes* parameters with some arbitrary value $\boldsymbol{\theta}_0$, for example all parameters equal to 0. It then computes the value of the cost function $J(\boldsymbol{\theta_0}, \hat{\mathbf{X}}, \mathbf{y})$ for the current parameters, $\boldsymbol{\theta}_0$. Next, it computes the gradients of the cost function with respect to the model parameters, $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta_0}, \hat{\mathbf{X}}, \mathbf{y})$. Finally, the algorithm updates each model parameter by a small value, *opposite* to the gradient's direction, in order to move the model towards the value $\boldsymbol{\theta}^\star$ that *minimizes* the loss. The size of the change is determined by a *hyper-parameter* λ called the *learning rate*. This process is repeated for a pre-defined number of iterations n_{iters}, chosen large enough for the process to *converge* to a local minimum. If the objective function is convex with respect to $\boldsymbol{\theta}$, the minimum is found for globally optimal parameters $\boldsymbol{\theta}^\star$.

The update of model parameters at iteration t of Gradient Descent is given by:

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \lambda\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t, \hat{X}, \mathbf{y}) && \text{(6a)}\\ &= \boldsymbol{\theta}_t - \lambda\frac{1}{n}\sum_{i=1}^{n}\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}_t, \hat{\mathbf{x}}_i, y_i), && \text{(6b)}\end{aligned}$$

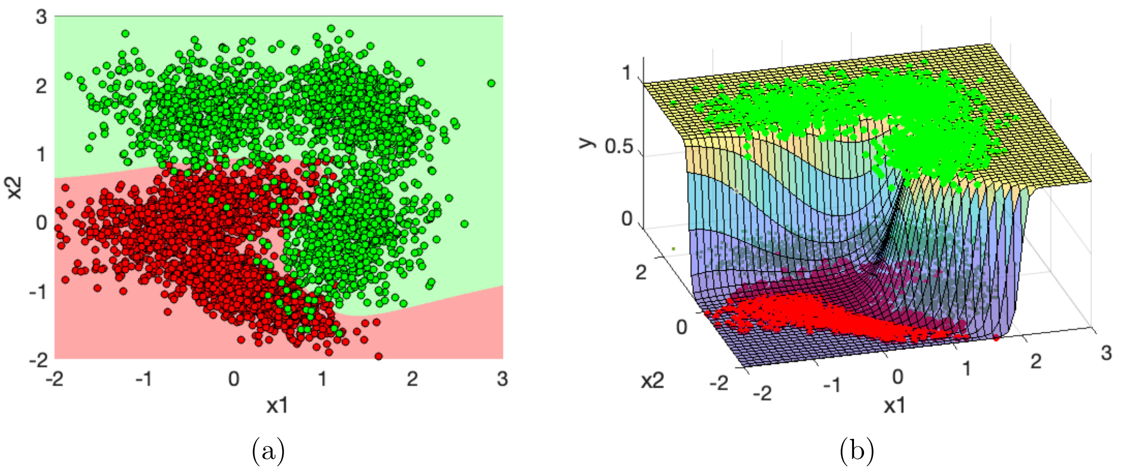


Figure 2: Logistic regression for binary classification task (2 classes), where datapoints have 2 feature dimensions ($\mathbf{x} = (x^{(1)}, x^{(2)})$). (a) We wish to learn a model that predicts the class of a sample $\mathbf{x}$, or equally, learn a *decision boundary* (black line) that separates the classes. Linear models such as logistic regression learn a linear decision boundary. (b) Target labels y for binary logistic regression take values $\in \{0, 1\}$, corresponding to the 2 classes, and value of y denotes the true class of sample $\mathbf{x}$. Logistic regression fits a sigmoid function to the data, predicting $\bar{y} = \sigma(\mathbf{x}^T\mathbf{w} + b) \in [0, 1]$. $\bar{y}$ can be interpreted as predicted probability by the model that input $\mathbf{x}$ is of class 2, and $1 - \bar{y}$ the probability it is of class 1.

where $\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}_t, \hat{\mathbf{x}}_i, y_i) \in \mathbb{R}^{d+1}$ is a column vector with one element for each of the d weights and 1 more for bias b. The i-th element of the vector is the partial derivative of the loss for sample $(\hat{\mathbf{x}}_i, y_i)$ with respect to the i-th parameter in $\boldsymbol{\theta}$. Similar is $\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta}_t, \hat{X}, \mathbf{y}) \in \mathbb{R}^{d+1}$, holding partial derivatives with respect to the cost function (average loss). Finally, $\lambda \in \mathbb{R}$ is the *learning rate*, an important hyper-parameter, for which the method developer (you) has to set an appropriate value so that the algorithm can converge adequately.

In our classification task we minimize the (mean) Log-Loss (Eq. 3) and therefore $\mathcal{L} = \mathcal{L}_C$ and $J = J_C$. We therefore need to compute the gradients of the Log-Loss $\nabla_{\boldsymbol{\theta}}\mathcal{L}_C(\boldsymbol{\theta}, \hat{\mathbf{x}}, y)$. Its gradients are given by:

$$\nabla_{\boldsymbol{\theta}}\mathcal{L}_C(\boldsymbol{\theta}, \hat{\mathbf{x}}, y) = (\bar{y} - y)\hat{\mathbf{x}} \tag{7}$$

Therefore we have all pieces we need to implement the model and the optimization via gradient descent. Finally, it is useful to implement **metrics** that will help **evaluate performance** of the optimized (trained) model. The mean Log-Loss (Eq. 4) could be used for this purpose, but it's not very intuitive. It is therefore common to use another metric for reporting model performance, the *classification error ratio* (as percentage %):

$$e = 100 \times \frac{\text{number of wrong predictions}}{\text{number of total predictions}} \quad (\%) \tag{8}$$

2.5 Learning a Non-Linear Classifier

In most real-world classification tasks, data cannot be accurately separated in different categories using simply a linear decision boundary, such as the one that can be learned by the linear model of Logistic Regression presented above. This is the case for the synthetic data we use in this task, as can be seen in Fig. 1a. To better classify such data, we would like to model non-linear decision boundaries. Can you extend Logistic Regression for this purpose?

For the *Regression task*, you have been taught during lectures of B1 how Linear Regression can be extended to non-linear Polynomial Regression, by deriving higher-degree features from those of the input, and learning optimal parameters of a higher-degree polynomial. We will use the same methodology for learning a non-linear classifier.

We will extend our Logistic Regression model (Eq. 1) to include non-linear relations between the original input features in $\mathbf{x}$ by incorporating a polynomial of $\mathbf{x}$. For simplicity, we will examine the case relevant to this project, where the original input vector $\mathbf{x}$ has 2 features, $\mathbf{x} = (x_1, x_2)$. In this case, for the standard (linear) Logistic Regression we have $\mathbf{x}^T\mathbf{w}+b = x_1w_1+x_2w_2+b$. We extend it to model non-linear decision boundaries by introducing an i-th degree polynomial, and Eq. 1 becomes:

$$\begin{aligned}
\bar{y} &= \sigma(\mathbf{x}_p^T\mathbf{w}_p + b) \\
&= \sigma(x_1^i w_{i,1} + x_1^{i-1}x_2 w_{i,2} + ... + x_1 x_2^{i-1} w_{i,i} + x_2^i w_{i,i+1} + \\
&\quad\; ... \\
&\quad\; x_1^4 w_{4,1} + x_1^3 x_2 w_{4,2} + x_1^2 x_2^2 w_{4,3} + x_1 x_2^3 w_{4,4} + x_2^4 w_{4,5} + \\
&\quad\; x_1^3 w_{3,1} + x_1^2 x_2 w_{3,2} + x_1 x_2^2 w_{3,3} + x_2^3 w_{3,4} + \\
&\quad\; x_1^2 w_{2,1} + x_1 x_2 w_{2,2} + x_2^2 w_{2,3} + \\
&\quad\; x_1 w_{1,1} + x_2 w_{1,2} + b) \\
&= \sigma(\hat{\mathbf{x}}_p^T \boldsymbol{\theta}_p) \qquad (9)
\end{aligned}$$

Here, $\hat{\mathbf{x}}_p = (\mathbf{x}_p, 1) = (x_1^i, x_1^{i-1}x_2, ..., x_1x_2^{i-1}, x_2^i, ..., x_1, x_2, 1)$ is a column vector that includes the original input features x_1, x_2, along with all terms up to the i-th degree, and a term 1 for the bias (cf. $\hat{\mathbf{x}}$ in Sec. 2.2). Row vector $\boldsymbol{\theta}_p = (w_{i,1}, w_{i,2}, ..., w_{i,i}, w_{i,i+1}, ..., w_{1,1}, w_{1,2}, b)$ contains all coefficients and the bias of the polynomial. These parameters of θ_p have to be learned with optimization.

This model can be implemented in various ways. A simple implementation is to create a function that creates $\hat{\mathbf{x}}_p$ for any degree of polynomial, given the original $\mathbf{x} = (x_1, x_2)$ input features of each sample. This can be implemented concisely with a short for-loop if one observes that the terms of $\hat{\mathbf{x}}_p$ in the i-th row of Eq. 9 can be obtained by multiplying the terms of the (i-1)-degree row with x_1, and then concatenate the term x_2^i. Vector $\boldsymbol{\theta}_p$ can then be created with as many parameters as the length of $\hat{\mathbf{x}}_p$. The values of $\boldsymbol{\theta}_p$ can be initialized with random values, for example all set equal to 0. Then, this model and $\boldsymbol{\theta}_p$ can be trained with Gradient Descent, with exactly the same process as the linear Logistic Regression model described previously. Because Eq. 9 can model non-linear relations between the original features, it can learn non-linear decision boundaries for classification of our data, as shown in Fig. 2.

3 Task

3.1 Implementing the models and optimizer

In this project you will implement the above described models, you will run experiments, and you will study the performance of these models, describing the analysis in your report. The first steps are to implement the linear Logistic Regression model, the Gradient Descent (GD) optimization function, the functions for evaluating the model performance, and functions for extending the model to its non-linear variant via polynomials of the input features. Start by looking at the given files. You can base your implementation either on the Matlab of Python version of these files (your choice). We recommend the following steps for the implementation:

- Call the *create_data* function twice, once for creating training and once for validation data. Generate 400 samples *X_train* and class labels *class_labels_train* for training. Generate 4000 samples *X_val* and class labels *class_labels_val* for validation. In all the below, optimization is to performed on the training data, whereas the validation data are only to be used for evaluating performance of the trained model. The given code includes a plotting function to help you visualise the sampled data.

- Prepare the data for logistic regression. First of all, the class labels have values 1 and 2 for the two classes respectively. Change the class-labels provided by the function to the appropriate values for the target variable y, 0 and 1 respectively (Sec. 2.2). Moreover, extend $\mathbf{X}$ with a term 1 corresponding to the bias, to obtain $\hat{\mathbf{X}}$ (Eq. 2).

- Implement function *grad_descent*, which takes as input $\mathbf{X}$ and $\mathbf{y}$ of the training data, a value for learning rate λ, and number of total training iterations *n_iters*. The function should initialize parameters of the model with a random value (we recommend all initialised from 0), then performs Gradient Descent, and returns the learned optimal parameters $\boldsymbol{\theta}$.

- Implement function *log_regr*, which takes as input data $\mathbf{X}$ and model parameters $\boldsymbol{\theta}$, and outputs predicted $\bar{y}$.

- Implement function *mean_logloss* that receives $\mathbf{X}$, the true class probabilities $\mathbf{y}$, and model parameters $\boldsymbol{\theta}$, and calculates the cost (Mean Log-Loss) over all training samples.

- Implement function *classif_error* that takes as input predictions $\hat{\mathbf{y}}$ and true labels $\mathbf{y}$, and calculates Eq. 8, the percentage of errors. The function should be able to calculate the error either for predictions made on the training data, or the validation data, as you will need both. *Note: Ensure you are comparing true y to predicted $\bar{y}$* ($\in [0, 1]$), *or true class labels to predicted class labels* ($\in \{1, 2\}$).

- Implement function *create_features_for_poly* that takes as input the original 2-d data returned from *create_data* and a variable *degree*. The function should calculate and return a design matrix $\mathbf{X}_p$ that has the higher-degree features for a polynomial with the specified *degree*. These can then be used with the rest of the above functions to train the non-linear model of Eq. 9. See Sec. 2.5 for hints on how to implement it. (*Note: This is perhaps the trickiest part of the project to implement. You can perform some of the analysis described below with the linear model, debug until you are satisfied with the results, and then come back to implement and analyse this non-linear model.*)

Implementing the above should enable you to run experiments to analyse performance of these models for your report, as per the below section. Code is not directly assessed and is not included in the report. Rather, we are interested in your experiments and analysis. Different implementations of these models are possible and allowed, so feel free to implement them as most convenient to you, as long as they correctly implement the models described in Sec. 2.

3.2 Your report: Analysis of the methods

Having implemented the above, you are now ready to run experiments to investigate and discuss performance of these methods. In your report, do not include code. Instead, focus on presenting the results of the below experiments and making related discussions.

1. You now need to perform optimizaton of your models. For this, you first need to configure hyper-parameters of your optimization process, specifically the learning rate λ and number of iterations n_iters of Gradient Descent (GD), by finding appropriate values. As is common practice, you need to perform multiple training and validation experiments, for variety of combinations of values for n_iters and λ. Train a model with GD and the 400 training samples, evaluate performance (via $classif\_error$ function) on 4000 validation samples, record it, and repeat for different hyper-parameters. In the end, choose the values that led to best performing model. This can be implemented easily via two external for-loops. A recommended set of values to explore is $n\_iters = \{100, 500, 1000, 10000\}$ and $\lambda = \{0.01, 0.1, 0.5, 1.0\}$. In your report, show the investigation in a table or plot and explain how you chose the best value of λ and n_iters.

   Important to note is that hyper-parameters that are appropriate for non-linear models are likely to also be good enough for linear model, but not vice-versa. To simplify this analysis, we recommend that you perform and present this investigation with a non-linear model of degree 2, 3, or 4 (just choose any). Then, find best hyper-parameters with this one model, and adopt these values for all following experiments.

2. Configuring hyper-parameters can be helped by plotting the value of the Loss at each GD iteration (y axis) against the number of the iteration (x axis). It can help if the model has converged if loss no longer decreases significantly. This can be implemented inside $grad\_descent$. If loss has not stabilised at end of training, adjust n_iters or λ appropriately. You can include in your report such plot for the model of your choice as evidence of appropriate configuration. *Note: Whether the loss has stabilised can be subjective based on visual inspection of the plot, or other means. Just explain your reasoning in the report.*

3. Briefly discuss in your report the relation between n_iters and λ of GD, how they affect quality of the solution and runtime? Which setting (n_iters small or large, coupled with appropriate λ) would you prefer in a real-world application of optimization?

4. Using the above best hyper-parameters, train any of your models. After training, calculate the average prediction error over your training samples, and over the validation samples. Repeat the training and validation experiment. If you have not explicitly set a constant *random number generator (RNG) seed* in your code (if you have, unset it), then you will observe different performance for each repetition of the experiment. This is because slightly different data are sampled from $create\_data$ each time the code is run. To judge performance of a model, we would like to 'average away' the 'chance' from sampling different data. For this, implement an external for-loop that repeats your whole training-validation experiment 10-20 times (less if your PC is slow), records for each repetition the average error over data samples, and finally reports the average error over these 10-20 repetitions. This average (of 10-20 error percentages) is more representative of your model's performance. Implement this so that it reports both the average error over training samples, and the average error over validation samples, as you will need both for the following items. *Note: You don't need to report anything for this item in your report.*

5. Compare performance of models of different degrees of polynomials. Using the best hyper-parameters found above, train models of degree 1 (linear), 2, 3, 4 and 5 using the 400 training samples. After training is complete, calculate their error on the 400 training samples. Also calculate their error on 4000 validation samples. Repeat each experiment 10-20 times as per above and report the average training error and validation error over these repetitions. Present in your report the results in a table (e.g. 1 column per model, 2 rows for training/validation error). Which model (degree) leads to best predictions? Are any models more prone to over-fitting the training data? *Note: Some differences can be small, 0.5% or less, and results can be 'noisy'. Simply discuss what you observe.*

6. Investigate how important is the amount of training samples for obtaining models with good predictive performance. How well can the best model from the previous item perform if it is trained with less data? Use *degree* of polynomials, λ and n_iters that gave best results previously. Re-train this model using number training samples equal to $n\_samples = [50, 100, 200, 400]$. Repeat each experiment 10-20 times. Calculate average error over training samples and average error over 4000 validation samples per setting). Present results in your report in a table with 2 rows (train/val) and a column per $n\_samples$ value. Is any over-fitting observed?

7. Which model performed the best out of all your experiments? Is this what you expected and why? Include in your report the optimal parameter values learned for this model. Can you plot the decision boundary it has learned, in a plot similar to Fig. 2a?

8. Have you had any numerical issues to resolve during the implementation of these methods? For example any numerical instabilities, etc? If so, include a brief description in your report and how you dealt with them.

Note-1: If you have not managed to implement some of the methods, for example the non-linear variant of the model, focus your analysis on the rest of your results. If you have space, you can extend your analysis with other models you may be familiar with, or analyse other aspects of these methods.

Note-2: As the report is short, 4-pages, make use of the space to concisely present your above experiments and analysis. Please do not copy material provided within this project description as this will be a waste of space. Also, there is no need to provide code within the report. Correctness and understanding is assessed from the results and your discussion.