B1 Engineering Computation - Project B

Optimization for regression and classification models

November 4, 2023

1 Introduction

This project investigates how to apply different optimization methods for learning optimal parameters of a model that can predict a value of interest for a given input data point. To perform this, the model has to approximate the true function that maps an input data point to the true value of interest. There are many such examples in real-world applications. For instance, we may wish to approximate the function that maps characteristics of housing property (size, age, construction quality of a house) to a property's market price. This is an example of a *regression* task. Regression refers to the task of estimating the function that maps multi-variate input $\mathbf{x} = (x^{(1)}, ..., x^{(d)})$ to a real-valued output $y \in \mathbb{R}$. Another common task is *classification*, where we are interested to approximate a function that maps multi-variate input $\mathbf{x}$ to a categorical variable y, which takes one value for each input sample out of C possible categories (classes). Real world example of a classification task is, given blood measurements of a patient, to categorise them as healthy or unhealthy.

The input vector $\mathbf{x}$ that describes characteristics of a data point, a sample, is often called an input *feature-vector*. The variable y that we are interested to predict is often called the *target* variable.

The functions that map input characteristics to target variables can be very complex in real-world problems. Therefore, a variety of models have been developed for regression and classification tasks. These models have multiple parameters $\boldsymbol{\theta}$, where for every different combination of values for these parameters, the model represents a different function $f_{\boldsymbol{\theta}} : \mathbf{x} \rightarrow y$. For a model to estimate closely the "true" function f that maps $\mathbf{x}$ to y, we need to find appropriate value for a model's parameters $\boldsymbol{\theta}$. Models used in complex problems can have a very large number of parameters, in the order of millions and billions for modern deep neural networks. Optimal values of parameters for such models 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 samples, computational optimization methods find the optimal parameters $\theta^{\star}$ for which the model can predict y with minimum error when given $\mathbf{x}$. The model, with the optimal parameters $\boldsymbol{\theta}^{\star}$, can then be applied to new data $\mathbf{x}$, for which real y is not known, in order for the model to predict it. This framework of learning to approximate 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* models for regression and classification. Linear models are based on a linear transformation of the input $\mathbf{x}$, to estimate the target variable y, of the form $\mathbf{x}^T\mathbf{w} + b$, where $\boldsymbol{\theta} = (w^{(1)}, ..., w^{(d)}, b)$ the parameters of the linear model. We will see 3 different models: Linear regression, logistic regression for classification, and linear support vector machines for classification. We will study multiple optimization methods for each model, such as finding optimal parameters via closed form solution of the optimization problem, Gradient Descent (GD), Stochastic Gradient Descent (SGD), and Linear Programming (LP).

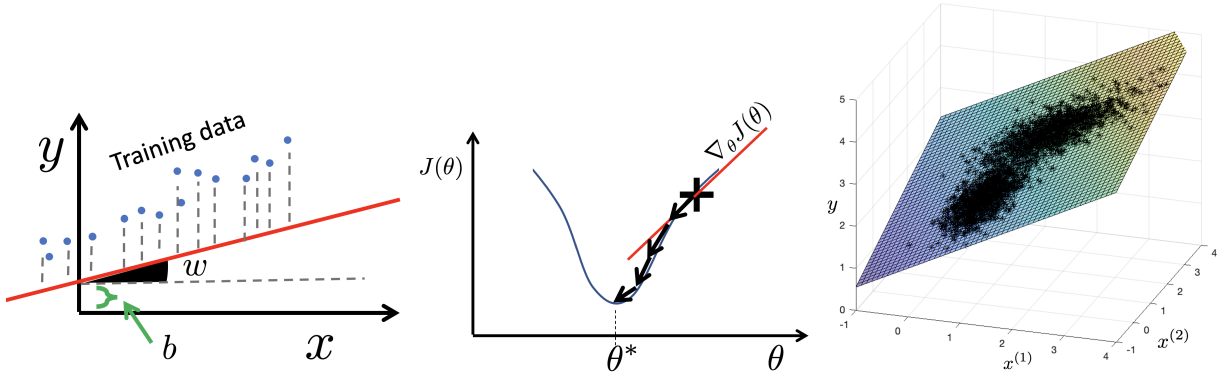
2 Dataset

You are given Matlab file *create_data.m* that implements the data-generating function *create_data*(*n_samples*). This function will be used throughout the project to generate data for training, validation, and testing. It takes as input the number of samples *n_samples*. For each sample, it creates a 2-dimensional feature vector $\mathbf{x} = (x^{(1)}, x^{(2)}) \in \mathbb{R}^2$ by sampling $\mathbf{x}$ from one of two different 2-dimensional Gaussian distributions. The Gaussians have a different, pre-defined mean and covariance matrix. Thus features $(x^{(1)}, x^{(2)})$ of samples from each Gaussian follow different distribution (Fig. 2a). Equal number of samples is taken from the two Gaussians. The two Gaussians represent two categories of samples, i.e. two classes. For each sample $\mathbf{x}$, the function also returns the associated "class" label, taking values $\{1, 2\}$, representing the Gaussian from where sample $\mathbf{x}$ came. This "class" will be the target variable that our classification models aim to predict, given $\mathbf{x}$, i.e. learn to separate samples generated by the two Gaussians. Finally, for each sample $\mathbf{x}$, the function *create_data* combines features $(x^{(1)}, x^{(2)})$ via a linear transformation and additive noise, producing a single value $\in \mathbb{R}$ per sample. This value will be the target y that our regression models will try to predict given $\mathbf{x}$. In other words, they will try to learn the linear function that produced y from $\mathbf{x}$. This is shown in Fig. 1c.

We recommend you to inspect the function *create_data*, to better understand the data-generating process that your models will try to learn.

Semantic meaning of the data: The data we use is synthetic, without real semantic meaning, to enable us to study properties of optimization and linear models. However, for an intuitive example of what such data could represent in the real world, one could imagine that the 2 classes represent 2 types of properties (apartments and houses), the features $(x^{(1)}, x^{(2)})$ represent the quality of neighborhood and the property's size (e.g. assuming that houses tend to be bigger), and the linear function of the two features produces the property's market price. Therefore the objective of the regression model is to predict a property's price given $\mathbf{x}$, and the objective of a classification model is to predict whether a sample $\mathbf{x}$ is a house or apartment.

Training, validation, test data: The below will become clearer when working on the following tasks, but here we provide an overview. We assume we have *n_samples* data samples (limited in real applications). 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 would 'generalize' on new, unseen data, e.g. if the model was deployed in the world to make predictions. To assess this, we commonly collect (here we create) another, separate *test* dataset and evaluate our models on it. Why? Because optimization may have found parameters optimal for training samples but they may be sub-optimal for new test data (over-fitting). Finally, some optimization methods have configurable *hyper-parameters*, such as learning rate of Gradient Descent (Sec. 4). These hyper-parameters are usually not optimized via an automatic method (unlike $\theta^\star$) but optimized by a human. How? The developer (you) tries to find hyper-parameter values that enable optimization to learn model parameters $\theta^\star$ that generalize well to new data. To do this, we form a separate *validation* set. Assuming we cannot collect more samples, we split the available data, e.g. 80% and 20% of the available *n_samples* samples as training and validation set respectively. We then train multiple models on train data using different hyper-parameter values, and assess their performance on the validation data. Then, we choose as optimal the model and the corresponding hyper-parameters that led to best performance on validation data. This model is finally evaluated on the test data, to measure how it would predict unseen data. Why don't we choose hyper-parameters by evaluating on the test data? Because we could over-fit the choice of hyper-parameters and model to the test data, and we would not know how the model would perform on completely new data.



(a) (b) (c)

Figure 1: In Linear regression, we attempt to learn a linear model that approximates the function $f : \mathbf{x} \rightarrow y$ that maps input features $\mathbf{x}$ to target variable of interest y. a) A linear model for a 1-dimensional x, with parameters $\boldsymbol{\theta} = (w, b)$. It's one weight w defines the slope and b its bias. Dash lines represent the error of the model's estimate for each training data point. b) 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 $\boldsymbol{\theta}^{\star}$. c) For 2-dimensional $\mathbf{x}$, the linear regression model finds a 2-dimensional plane instead of a line. In higher-dimensions, we model a hyper-plane.

3 Linear regression via analytical solution for Mean Squared Error (MSE)

3.1 Linear regression

We start with a general definition of Linear Regression and a first method for optimizing it, via finding the analytical solution for minimizing the mean squared error. 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*. $\mathbf{x}$ 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, this data is synthesized, with $d = 2$ the dimensionality of feature vectors $\mathbf{x}$ (Sec. 2), whereas in other applications they could have been collected from real-world measurements. We are interested to learn a model that can predict what value y takes when we are given specific values for $\mathbf{x}$. We assume that values of y are generated by a linear transformation of the d-features $\mathbf{x}$. Therefore, we will perform Linear Regression, using the linear model:

$$\bar{y} = \mathbf{x}^T \mathbf{w} + b = x^{(d)} w^{(d)} + ... + x^{(1)} w^{(1)} + b \tag{1}$$

In the above, we use $\bar{y} \in \mathbb{R}$ to represent the *estimated* (predicted) value of the target variable, as estimated by our model for a given feature vector $\mathbf{x}$ and model parameters $(\mathbf{w}, b)$, to separate it from the *real* value y of the target variable (that we have in our database for our n training samples). With $\mathbf{w} \in \mathbb{R}^d$ we represent a column vector that holds the *weights* of the linear model, and $b \in \mathbb{R}$ is called the *bias*. These can be jointly represented as the column vector $\boldsymbol{\theta} = (w^{(1)}, ..., w^{(d)}, b)$ that holds all the parameters of our linear model. If we now extend

$\mathbf{x}$ with the value 1, defining $\hat{\mathbf{x}} = (x^{(1)}, ..., x^{(d)}, 1)$, we can conveniently rewrite Eq. 1 and our linear model compactly as:

$$\bar{y} = \hat{\mathbf{x}}^T \boldsymbol{\theta} \tag{2}$$

The above form 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.

It is often convenient to use matrices to refer to the application of a model on a whole database, which 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, Eq. 1 and Eq. 2 are compactly expressed for all data points in our database as:

$$\bar{\mathbf{y}} = \mathbf{X}\mathbf{w} + b \tag{3a}$$
$$= \hat{\mathbf{X}}\boldsymbol{\theta} \tag{3b}$$

3.2 Optimizing Mean Squared Error via analytical solution

We now start considering how to learn *optimal* parameters $\boldsymbol{\theta}^\star$ of our model, such that it accurately estimates the true function $f : \mathbf{x} \rightarrow y$ that generates the value of y for a given $\mathbf{x}$. For this purpose, we need to define a *loss function* $\mathcal{L}(\boldsymbol{\theta}, \hat{\mathbf{x}}, y) \in \mathbb{R}$, which is a real-valued measure of the error between the prediction $\bar{y}$ made by a model with parameters $\boldsymbol{\theta}$ and the true target value y. This is visualised in Fig. 1a. We then define the *objective function* (often also called the *cost function*), $J(\boldsymbol{\theta}, \hat{\mathbf{X}}, \mathbf{y}) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(\boldsymbol{\theta}, \hat{\mathbf{x}}_i, y_i) \in \mathbb{R}$, which quantifies the amount of our model's error over all training samples in $\hat{\mathbf{X}}$. Using an optimization method, we can then search for the value $\boldsymbol{\theta}^\star$ of model parameters that minimize the objective function, thus minimizing the error. There are many loss functions and corresponding objective functions. One of the most commonly used losses is the squared-error, and respectively the *Mean Squared Error* (MSE) objective function. The loss is given by:

$$\mathcal{L}_{\text{mse}}(\boldsymbol{\theta}, \hat{\mathbf{x}}, y) = (\bar{y} - y)^2 = (\hat{\mathbf{x}}^T\boldsymbol{\theta} - y)^2 \in \mathbb{R} \Rightarrow \tag{4a}$$
$$\mathcal{L}_{\text{mse}}(\boldsymbol{\theta}, \hat{\mathbf{X}}, y) = (\bar{\mathbf{y}} - \mathbf{y})^2 = (\hat{\mathbf{X}}\boldsymbol{\theta} - \mathbf{y})^2 \in \mathbb{R}^n \tag{4b}$$

The latter Eq. 4b is a convenient form for computing the loss over multiple samples via matrix operations (e.g. in Matlab) and gives a column vector with the loss for each sample. Respectively, the MSE objective function is given by:

$$J_{\text{mse}}(\boldsymbol{\theta}, \hat{\mathbf{X}}, \mathbf{y}) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}_{\text{mse}}(\boldsymbol{\theta}, \hat{\mathbf{x}}_i, y_i) = \frac{1}{n}\sum_{i=1}^{n} (\hat{\mathbf{x}}_i^T\boldsymbol{\theta} - y_i)^2 \tag{5a}$$
$$= \frac{1}{n}(\hat{\mathbf{X}}\boldsymbol{\theta} - \mathbf{y})^T(\hat{\mathbf{X}}\boldsymbol{\theta} - \mathbf{y}) \in \mathbb{R} \tag{5b}$$

The optimal model parameters $\boldsymbol{\theta}^\star$ minimize the above objective function. Minimization of the objective function is the target of optimization. Here, we write this as:

$$\boldsymbol{\theta}^\star = \arg\min_{\boldsymbol{\theta}} J_{\text{mse}}(\boldsymbol{\theta}, \hat{\mathbf{X}}, \mathbf{y}) \tag{6a}$$

$$= \arg\min_{\boldsymbol{\theta}} \frac{1}{n}(\hat{\mathbf{X}}\boldsymbol{\theta} - \mathbf{y})^T(\hat{\mathbf{X}}\boldsymbol{\theta} - \mathbf{y}) \tag{6b}$$

The MSE objective function (Eq. 5) is convex and admits a single solution, the global minimum. As shown in Lecture 2 on Numerical Algorithms, we can find an analytical expression for this global minimum, by differentiating and then setting the gradient to 0. We then obtain a system of linear equations of the form:

$$\mathbf{A}\boldsymbol{\theta}^\star = \mathbf{B} \Rightarrow \boldsymbol{\theta}^\star = \mathbf{A}^{-1}\mathbf{B}, \tag{7}$$

In our case, these equations are given by:

$$\boldsymbol{\theta}^\star = (\hat{\mathbf{X}}^T\hat{\mathbf{X}})^{-1}\hat{\mathbf{X}}^T\mathbf{y} \tag{8}$$

The above gives the analytical solution to the optimization problem in Eq. 5. Note that the matrix $\mathbf{A}$ can be non-invertible. We need to take care of such situations. This can be done simply by adding a small constant (e.g. $\epsilon \approx 0.0001$) to the diagonal elements of $\mathbf{A}$.

3.3 Task 1: Linear regression via analytical solution to MSE

- Generate $n\_samples\_train = 1000$ training samples using the given function. Get X_train and y_train for seed of the random number generator (RNG) equal to 12345.
- Implement function $mse\_regression\_closed\_form$ that takes as input X_train and y_train and returns the optimal parameters $\boldsymbol{\theta}^\star$, by solving Eq. 8.
  Note: You can use Matlab's in-built function inv to compute matrix inverse.
- (R) Report what weights and bias are learned with 1000 training samples. Have a look at the function that generates the data. Are the found $\mathbf{w}, b$ the expected solution? Why? Explain based on the given implementation of the data generating function.
- Generate another 20000 samples using the same function, which will serve as *test data* (X_test, y_test), to measure model performance on data not seen during training.
- (R) Implement function $mean\_squared\_error$ that takes as input X, y and $\boldsymbol{\theta}$ and returns the Mean Squared Error (MSE) over the given data (Eq. 5b). What MSE does this function return for the above 20000 test samples?
- (R) Reduce the number of training samples to 10, while using the same number of test samples. What solution for the optimal model parameters do you find, for RNG seed 12345? Does the solution differ from the one found for 1000 training samples? If yes, explain why.
- (R) Run experiments for number of training samples $n\_samples\_train = [4, 10, 20, 100, 1000, 10000]$. For each value of $n\_samples\_train$ run 20 experiments, each with a different RNG seed (advice: use *for* loop). For each value of n_train, compute the mean and standard deviation of the MSE over the 20 seeds, separately for training samples and for test samples. In your report, present these values (e.g. in a table with 2 rows, 6 columns, each cell reporting mean $\pm$ std)). Explain how number of training samples affects MSE over training samples and MSE over test samples and why.

4 Linear Regression with Gradient Descent

Unlike minimization of MSE with linear regression models, the optimization of most models and objective functions does not admit an analytical solution. We here investigate optimization with Gradient Descent (GD), a method for optimizing any differentiable function.

The Gradient Descent algorithm is shown in Fig. 1b. Consider a model with parameters $\boldsymbol{\theta}$, such as our linear model (Eq. 3), and an objective function that we wish to minimize, such as MSE (Eq. 5). 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:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \lambda \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t, \hat{X}, \mathbf{y}) \tag{9a}$$

$$= \boldsymbol{\theta}_t - \lambda \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_t, \hat{\mathbf{x}}_i, y_i), \tag{9b}$$

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 the 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 $\boldsymbol{\theta}$. $\lambda \in \mathbb{R}$ is a pre-defined learning rate.

To fit the Linear regression model, we minimize the MSE loss and objective, i.e. $\mathcal{L}_{\text{mse}}(\boldsymbol{\theta}, \hat{\mathbf{x}}, y)$ and $J_{\text{mse}}(\boldsymbol{\theta}, \hat{\mathbf{X}}, \mathbf{y})$ from Eq. 4 and Eq. 5 respectively. To do this with GD we need to derive their gradients with respect to each of the parameters $\boldsymbol{\theta}$, to perform the updates of Eq. 9 on our training data until we find optimal parameters $\boldsymbol{\theta}^\star$. From Eq. 4 and Eq. 5 the gradients are:

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}_{MSE}(\boldsymbol{\theta}, \hat{\mathbf{x}}, y) = 2\hat{\mathbf{x}}(\hat{\mathbf{x}}^T \boldsymbol{\theta} - y) \in \mathbb{R}^{d+1} \tag{10a}$$

$$\nabla_{\boldsymbol{\theta}} J_{MSE}(\boldsymbol{\theta}, \hat{\mathbf{X}}, \mathbf{y}) = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} \mathcal{L}_{MSE}(\boldsymbol{\theta}_t, \hat{\mathbf{x}}_i, y_i) = \frac{2}{n} \hat{\mathbf{X}}^T (\hat{\mathbf{X}}\boldsymbol{\theta} - \mathbf{y}) \in \mathbb{R}^{d+1} \tag{10b}$$

In both of the above, the calculated gradient is vector $\in \mathbb{R}^{d+1}$. For each of the d+1 parameters, Eq. 10a calculates the gradient given one sample $\hat{\mathbf{x}}$, whereas Eq. 10b calculates the average gradient over all samples in a design matrix $\hat{\mathbf{X}} \in \mathbb{R}^{n \times d}$ (advice: easier to implement the latter).

4.1 Task 2: Linear Regression via Gradient Descent

- Generate X_train_val and y_train_val data with $n\_samples = 1000$ samples, calling the given function, similar to Task 1, using RNG $seed = 12345$. Write a new function $divide\_data$, which takes as input X_train_val and y_train_val, and divides these samples into 2 subsets: a subset X_train, y_train with 80% of the data for training, and a subset X_val, y_val with 20% of the samples for validation. The two subsets should be mutually exclusive (no sample in both). The samples for the validation split

should be sampled randomly (*not* first 80% - last 20% to avoid putting samples of only one class in one of the splits). You can use Matlab's *randperm* function to do this.

- Implement function $linear\_regression\_gd$ that takes as input X_train, y_train, a value for learning rate λ, and number of total training iterations n_iters. It should implement Eq. 9 and Eq. 10 to perform GD and return learned optimal parameters $\boldsymbol{\theta}^{\star}$.

- (R) Perform GD using the previously generated data and implemented function for $n\_iters = 1000$, $seed = 12345$, and find an appropriate value for learning rate λ. To find a good value for λ, repeat the same experiment with different values for λ. For example, starting from a small value $\lambda = 0.00001$ and increase by $\times 10$ in followup experiment, up to value 1. After each training experiment, use the function implemented in Task 1 to compute MSE achieved on the training and validation split by the optimized parameters. Based on this, choose the λ that results in the model that is most likely to achieve minimum MSE on unseen test data. In your report, show MSE results for all *lambda* values (e.g. in a table) and explain your choice of λ.

- (R) Generate 20000 test samples X_test, y_test, as in Task 1. Test on these samples the performance of the model parameters found with GD for the given value of n_iters and your chosen value of λ. Compare the MSE achieved on this test data and the validation data. Do they differ? Why?

- (R) Perform the same experiments as above, but for $n\_iters = 10000$. What is the minimum value of λ that gives results comparable as the ones for $n\_iters = 1000$ above?

- (R) Discuss in your report the relation between n_iters and λ of GD, and how they affect the quality of the solution. Which parameter between the two affects runtime? Which setting (n_iters short or large, coupled with appropriate λ) would you prefer in practice?

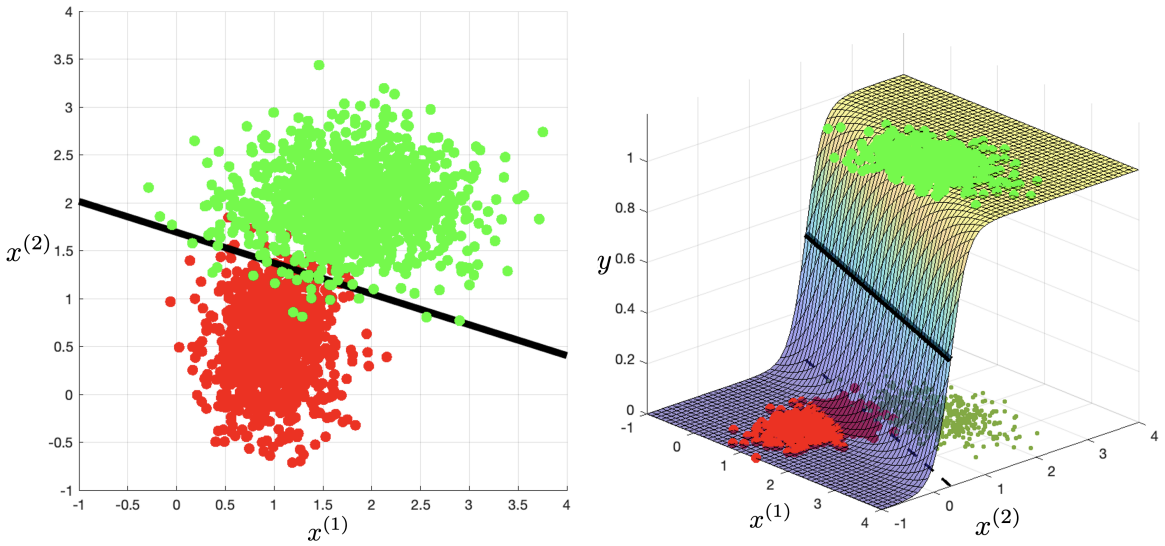
- (R) How does the best performing model found in this task compared with the optimal solution found in Task 1 when using the same number of training data? Is this what you expect in theory?

4.2 Classification with Logistic Regression and Gradient Descent

4.3 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. 2a, classes would be $c = \{1, 2\}$ corresponding to green and red samples respectively. When solving 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.

There are numerous models for classification. We here study Logistic Regression, which learns a linear decision boundary. To tackle classification, linear regression formulates it as follows: We design 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 training sample $\mathbf{x}_i$, the corresponding *true* value of the target variable y_i will take 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. 2b.



(a) (b)

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.

The model of Logistic Regression is given by:

$$\bar{y} = \sigma(\mathbf{x}^T\mathbf{w} + b) = \frac{1}{1 + e^{-\mathbf{x}^T\mathbf{w} - b}} \tag{11}$$

$$= \sigma(\hat{\mathbf{x}}^T\boldsymbol{\theta}) = \frac{1}{1 + e^{-\hat{\mathbf{x}}^T\boldsymbol{\theta}}}. \tag{12}$$

Here, $\sigma(\cdot)$ is the sigmoid function, applied on top of a linear transformation of the input $\mathbf{x}^T\mathbf{w} + b$, in order to "limit" the output value range of $\bar{y}$ between 0 and 1. It can then be interpreted as the probability that $\mathbf{x}$ is of class $c = 2$, whereas $1 - \bar{y}$ is probability that the input is of class $c = 1$. From this predicted probability $\bar{y}$, we get the predicted *class label* ($\{1, 2\}$ in our example) by simply assigning class label $c = 2$ if $\bar{y} > 0.5$ and $c = 1$ otherwise. The decision boundary of Linear Regression is a line, and its function can be found by setting $\bar{y} = 0.5$ in Eq. 11 and deriving the slope and intercept of a line as a function of $\mathbf{w}$ and $\mathbf{b}$. Vectors $\hat{\mathbf{x}}$ and $\boldsymbol{\theta}$ are defined similarly as in previous section for linear regression, to jointly treat weights $\mathbf{w}$ and bias b. It follows that for the whole design matrix $\hat{\mathbf{X}}$ we obtain predictions (useful for matrix-based implementation in Matlab):

$$\bar{\mathbf{y}} = \sigma(\hat{\mathbf{X}}\boldsymbol{\theta}) = \frac{1}{1 + e^{-\hat{\mathbf{X}}\boldsymbol{\theta}}} \tag{13}$$

4.4 Optimizing with Gradient Descent

We need to optimize parameters $\boldsymbol{\theta}$ such that predictions $\bar{y}_i$ match the true target y_i for each training sample $\mathbf{x}_i$. As in Task 1, we need to define a loss and objective function that quantify the error in the predictions over the training set. In Logistic Regression we use the following loss, often called the *Log-Loss*:

$$\mathcal{L}_C(\boldsymbol{\theta}, \hat{\mathbf{x}}, y) = -y \log(\bar{y}) - (1-y)\log(1-\bar{y}) \tag{14a}$$
$$= -y \log(\sigma(\hat{\mathbf{x}}^T\boldsymbol{\theta})) - (1-y)\log(1-\sigma(\hat{\mathbf{x}}^T\boldsymbol{\theta})) \in \mathbb{R} \tag{14b}$$

We then have the cost function for classification, similarly as defined for regression (Sec. 3.2), which is the *mean Log-Loss* over our training samples and quantifies the mean error in our predictions:

$$J_{\mathrm{C}}(\boldsymbol{\theta}, \hat{\mathbf{X}}, \mathbf{y}) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}_{\mathrm{C}}(\boldsymbol{\theta}, \hat{\mathbf{x}}_i, y_i) \tag{15}$$

We want to minimize this cost function, to learn optimal parameters $\boldsymbol{\theta}^\star$:

$$\boldsymbol{\theta}^\star = \arg\min_{\boldsymbol{\theta}} J_C(\boldsymbol{\theta}, \hat{\mathbf{X}}, \mathbf{y}) \tag{16}$$

Finding the minimum of this cost function and loss does not admit to an analytical solution, contrary to MSE. It is, however, differentiable, and we can therefore use Gradient Descent, with the same algorithm described in Sec. 4. We will iteratively update our parameters for a number of iterations until convergence and the minimum is found:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \lambda \nabla_{\boldsymbol{\theta}} J_C(\boldsymbol{\theta}_t, \hat{X}, \mathbf{y}) = \boldsymbol{\theta}_t - \lambda \frac{1}{n}\sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} \mathcal{L}_C(\boldsymbol{\theta}_t, \hat{\mathbf{x}}_i, y_i) \tag{17}$$

The difference is that now we 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:

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}_C(\boldsymbol{\theta}, \hat{\mathbf{x}}, y) = (\bar{y} - y)\hat{\mathbf{x}} \tag{18}$$

Finally, it is useful to implement metrics that will help monitor performance of the model. The mean Log-Loss (Eq. 26) could be used for this purpose, but it's not very intuitive. It is therefore common to use another metric for reporting model performance (both during training and testing), the *classification error ratio*:

$$e = \frac{\text{number of wrong predictions}}{\text{number of total predictions}} \tag{19}$$

4.5 Task 3: Logistic Regression using Gradient Descent

- Similarly to Task 2, generate X_train_val and y_train_val data with $n\_samples = 1000$ samples, using RNG $seed = 12345$, and use previously implemented function $divide\_data$ to divide them into 2 subsets, (X_train, y_train) with 80% and X_val, y_val with 20% of samples. Make sure to change the class-labels provided by the function to the appropriate values for y, 0 or 1 for the two classes respectively.
- (R) In your report, show how gradients of Log-Loss (Eq. 18) are derived from Eq. 14.

- Implement function *logistic_regression_gd*, which takes as input X_train, y_train, a value for learning rate λ, number of total training iterations n_iters, performs Gradient Descent and returns an estimate of the optimal parameters $\boldsymbol{\theta}^{\star}$.
- 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 as arguments $\mathbf{X}$, the true class probabilities $\mathbf{y}$, and model parameters $\boldsymbol{\theta}$, and calculates the cost over all training samples, i.e. the mean Log-Loss.
- Implement function $classif\_error$ that calculates Eq. 19, the ratio of errors to the total number of predictions. *Note: Ensure you are comparing true y to predicted $\bar{y}$ ($\in [0, 1]$), or true class labels to predicted class labels ($\in \{1, 2\}$).*
- (R) Perform GD using the data generated above, the implemented function for $n\_iters = 1000$, $seed = 12345$, and find appropriate value for learning rate λ. As in Task 2, repeat the experiment with different values for λ, starting from a small value $\lambda = 0.00001$ and increase by $\times 10$ in followup experiments. After each training, use the $classif\_error$ function to compute error on the training and validation split, to find a good value for λ. In your report, show the investigation (e.g. in a table) and optimal λ found.
- (R) Has optimization of your model converged after n_iters, and loss no longer decreases? Or would further adjustments to n_iters or λ be beneficial? To investigate this extend your function *logistic_regression_gd*. At every GD iteration, compute the average classification loss using $mean\_logloss$ and append it to a vector to keep track of it. Then, after the final iteration, using Matlab's *plot* function, plot the average loss for each iteration (y axis) against the number of iteration (x axis). If loss has not plateaued at end of training, adjust n_iters or λ appropriately. In your report include such plots and discuss any adaptations you made to improve convergence based them.
- (R) Evaluate on 20000 test samples the model that achieved the least validation error. What test error does it achieve? How does its test error compare to its validation error?
- (R) Investigate how well the model generalizes to new data after training with different amount of data. Use values of λ and n_iters that previously gave best results. Train the model using $n\_samples = [10, 20, 100, 1000, 10000]$ with 80% for training and 20% for validation. Then, evaluate each model on 20000 test samples. Repeat every setting for 20 random seeds. In your report, create a table with 3 rows (train/val/test) and a column per $n\_samples$ value. In each column report the mean $\pm$ standard deviation (over the seeds) of the classification error achieved on training, validation and test samples. Is any over-fitting observed and where? Also discuss how performance on validation set relates to performance on test set, the importance of their size, and what implications this may have for real world applications.

5 Logistic Regression with Stochastic Gradient Descent

5.1 Stochastic Gradient Descent

A computational deficiency of Gradient Descent is that the computation required for a single iteration, a parameter update, increases linearly with the amount of samples in $\mathbf{X}$. Real-

world databases can be very large and this can prohibit use of GD. A more suitable alternative for large databases is a variant called *Stochastic Gradient Descent* (SGD). The algorithm closely resembles the basic GD and can be used to optimize the same models and cost functions. As the name implies, the difference is that it contains stochasticity, due to an additional sampling step. In short, instead of processing all samples $\mathbf{X}$ in our training database to compute the cost $J_C(\boldsymbol{\theta}, \hat{\mathbf{X}}, \mathbf{y})$ and obtain gradients to perform the update, only a random subset of samples is used at every iteration. Specifically, at the beginning of every iteration of GD, we randomly sample n_B samples out of the n samples in our database $\{\hat{\mathbf{X}}, \mathbf{y}\}$, to form a *batch* consisting of $\{\hat{\mathbf{X}}_B, \mathbf{y}_B\}$ that holds only n_B samples. Appropriate value of n_B needs to be chosen by us. Commonly $n_B \ll n$, for example n_B is usually in the order of 1-1000 samples. The rest of the algorithm is the same as GD, except that now we calculate the loss, compute gradients and update parameters in the current iteration only based on the current batch:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \lambda \nabla_{\boldsymbol{\theta}} J_C(\boldsymbol{\theta}_t, \hat{X}_B, \mathbf{y}_B) \tag{20}$$

$$= \boldsymbol{\theta}_t - \lambda \frac{1}{n_B} \sum_{i=1}^{n_B} \nabla_{\boldsymbol{\theta}} \mathcal{L}_C(\boldsymbol{\theta}_t, \hat{\mathbf{x}}_{B,i}, y_{B,i}) \tag{21}$$

In the above, $\hat{\mathbf{x}}_{B,i}$ and $y_{B,i}$ are the i-th row of $\hat{\mathbf{X}}_B$ and $\mathbf{y}_B$ respectively, and are the feature vector and the real target value of the i-th sample in the batch (not the whole database).

Note: In Eq. 21 we average over the batch dividing by $\frac{1}{n_B}$, not $\frac{1}{n}$. Don't forget adapting this detail when extending to SGD from GD, else results can be suboptimal.

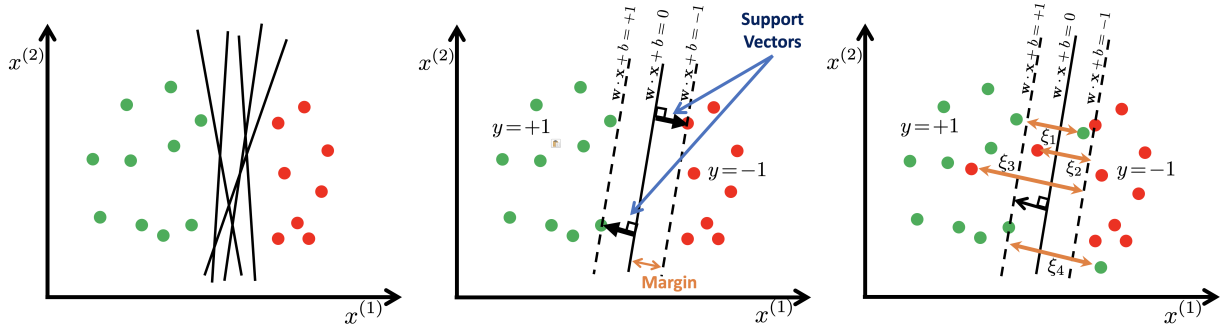
5.2 Task 4: Logistic Regression with Stochastic Gradient Descent

- As in Task 3, generate X_train_val and y_train_val data with $n\_samples = 1000$ samples and divide them into 80% (X_train, y_train) and 20% X_val, y_val samples. Also create 20000 test samples.

- Implement function *logistic_regression_sgd* that performs SGD on training batches. This can be done by extending the corresponding function from Task 3, to receive one extra input argument *batch_size*. Then, for every training iteration, a batch is *randomly* sampled from (X_train, y_train) and a model update is performed. You can use Matlab's *rand* or *randperm* functions to obtain random samples for your batch.

- (R) We now need to configure appropriate values for 3 hyper-parameters: n_iters, λ, and *batch_size*. To start, use the values for n_iters and λ that gave you best validation performance in Task 3. Then, run experiments with different values of $batch\_size = \{1, 10, 20, 50, 100\}$. For each value, run multiple experiments with at least 20 RNG seeds. In your report, show mean and std (over seeds) of error on the train and validation data for each *batch_size* value, and discuss the results. (*Note: Optimal settings may vary for different data. In this synthetic data you may only observe small differences.*)

- (R) Does your model converge well, regardless the stochasticity from the batch sampling? Set *batch_size* with best validation performance from previous experiments. As in Task 3, plot the mean (over batch) classification loss in each training iteration (y-axis) against the number of current iteration (x-axis). Has the loss plateaued at end of training? If not, adjust λ and n_iters. Present your investigation in the report.



(a) (b) (c)

Figure 3: (a) Many decision boundaries can achieve equally good performance on train data, but will generalize differently on unseen test data. (b) The SVM learns the boundary with the largest margin (distance) from samples of different classes. (c) If classes are not perfectly separable, some *slack* must be allowed for some errors. The formulation of the SVM with the use of Slack parameters ξ_i, one for each training sample $\mathbf{x}_i$, enables learning optimal decision boundary. That is, the boundary where the slack parameters have the lowest value in total. This formulation will be optimized via Linear Programming in this project.

- (R) Compare the plot of loss-over-iterations between SGD and GD for similar choices of n_iters and λ. Do you observe any "noise" in SGD convergence? If yes, what causes it?
- (R) Choose the model that performed best on your validation data. Evaluate its performance on the test data. In your report compare test and validation accuracies. How do they compare with performance achieved with GD in Task 3?
- (Optional) Discuss in your report how the number of training samples affects *runtime* (computations) and *memory* required by GD and SGD. How does the batch size affect SGD? What are the implications for large databases in real-world applications? You could obtain empirical insights by measuring how long the main calculations of (S)GD take for different number of training samples and batch size and make a plot. You can measure runtime by surrounding the relevant functions with the commands *tic* and $time\_taken = toc$. (Note: measuring exact memory required may be harder and subject to specific hardware and Matlab version. Simply discuss what you expect in theory.)

6 Support Vector Machine and Linear Programming

6.1 The model and its objective function

A Support Vector Machine (SVM) is another model for classification. Multiple variants of SVM have been developed and used for variety of classification problems. Here, we are exploring the linear variant for binary classification.

The Linear SVM is a linear model $\bar{y} = \mathbf{x}^T\mathbf{w} + b$, where $\bar{y} \in \mathbb{R}$ is the *classification score* predicted by the model for a sample $\mathbf{x}$. Predicted value $\bar{y} < 0$ indicates $\mathbf{x}$ belongs to the first class according to the model, whereas $\bar{y} > 0$ indicates $\mathbf{x}$ belongs to the second class. $\bar{y} = 0$ gives the function of the linear decision boundary. This is quite similar to Linear Regression, without the use of the sigmoid to turn $\bar{y}$ into a pseudo-probability $\in [0, 1]$. As in previous tasks, to simplify notation, we will use $\boldsymbol{\theta} = (w^{(1)}, ..., w^{(d)}, b)$ and $\hat{\mathbf{x}} = (x^{(1)}, ..., x^{(d)}, 1)$ to jointly treat

weights and bias. The SVM model then becomes simply:

$$\bar{y} = \hat{\mathbf{x}}^T \boldsymbol{\theta} \tag{22}$$

What distinguishes this SVM from other linear models is its learning objective. There can be multiple linear decision boundaries (for different $\boldsymbol{\theta}$) that achieve equally low error on the training data (Fig. 3a). They will not generalize equally well on unseen, test data, however. What is the optimal one and how do we learn it? The fundamental idea behind training the SVM is that the best decision boundary lies as far away as possible from training samples of different classes - it has the largest possible *margin* from the classes. The samples of the two classes that are closest to the decision boundary and define how large this margin can be, are the *support vectors*, hence the model's name. To learn the parameters $\boldsymbol{\theta}$ of the SVM, we define that the margin should be at least 1 unit wide. Then, we wish to learn parameters $\boldsymbol{\theta}$ such that for all samples belonging to the first class it is satisfied that $\hat{\mathbf{x}}^T\boldsymbol{\theta} < -1$, whereas for all samples belonging to the second class it is satisfied that $\hat{\mathbf{x}}^T\boldsymbol{\theta} > +1$ (Fig. 3b). For the SVM, the target variable y, which expresses the *true* class of a sample, is defined to take values -1 and $+1$. For each training sample $\hat{\mathbf{x}}_i$, we set the target variable y_i that expresses the *true* class of the training sample to take value $y_i = -1$ if sample $\mathbf{x}_i$ is from the first class, and $y_i = +1$ if it belongs to the second class. Then, we can concisely write that the parameters $\boldsymbol{\theta}$ of the SVM should satisfy the condition $(\hat{\mathbf{x}}_i^T\boldsymbol{\theta})y_i > 1$, for all $(\hat{\mathbf{x}}_i, y_i)$ in the training database. This is not entirely sufficient however...

It may not be possible to separate the training data perfectly with a linear decision boundary and therefore just the largest-margin desideratum is insufficient (Fig. 3c). To accommodate for this, to find optimal SVM parameters, we also make use of additional *slack parameters* $\boldsymbol{\xi} = (\xi_1, ..., \xi_n) \in \mathbb{R}^n$, where n is the number of training samples. Each slack parameter $\xi_i \in \mathbb{R}$ is associated with one training sample $(\hat{\mathbf{x}}_i, y_i)$, and the value of ξ_i defines how far on the "wrong" side of the decision boundary this sample lies (Fig. 3c). Essentially, each slack parameter quantifies the error of the model for the i-th sample. We can now formally define the optimization objective of the SVM as follows:

$$\boldsymbol{\theta}^\star, \boldsymbol{\xi}^\star = \arg\min_{\boldsymbol{\theta},\boldsymbol{\xi}} \sum_{i=1}^{n} \xi_i \tag{23a}$$

$$\text{subject to: } (\hat{\mathbf{x}}_i^T\boldsymbol{\theta})y_i \geq 1 - \xi_i, \forall i \tag{23b}$$

$$\xi_i \geq 0, \forall i \tag{23c}$$

This is a form of *constrained optimization*. It is interpreted as follows: We are looking to find the values for model parameters $\boldsymbol{\theta}$ and the values for the slack parameters $\boldsymbol{\xi}$, such that Eq. 23a is minimized, while enforcing the constrains of Eq. 23b and Eq. 23c. Therefore, the optimal model parameters $\boldsymbol{\theta}^\star$ are those for which the model $(\hat{\mathbf{x}}_i^T\boldsymbol{\theta})$ maps each sample $\hat{\mathbf{x}}_i$ further from the decision boundary by the margin 1 minus an allowed error ξ_i for each sample (Eq. 23b), while the sum of the errors is minimum (Eq. 23a), and errors cannot be negative (Eq. 23c). After optimization, only parameters $\boldsymbol{\theta}^\star$ are used, to predict the class of a new sample ($\bar{y} < 0$ or $\bar{y} > 0$) via Eq. 22, while slack parameters $\boldsymbol{\xi}^\star$ are discarded.

6.2 Optimization via Linear Programming

How do we optimize such a constrained optimization problem? We observe that the constraints of Eq. 23b and Eq. 23c form a system of linear equations, one for each sample i.

Such an optimization problem can be solved via Linear Programming. We will solve this using MatLab's function $linprog(\mathbf{f}, \mathbf{A}, \boldsymbol{\beta}, \boldsymbol{A_{eq}}, \boldsymbol{\beta_{eq}}, \mathbf{lb}, \mathbf{ub})$. This function finds the global minimum of an optimization problem of the form:

$$\boldsymbol{\psi}^\star = \arg\min_{\boldsymbol{\psi}} \mathbf{f}^T\boldsymbol{\psi} \text{ such that } \begin{cases} \mathbf{A}\cdot\boldsymbol{\psi} \leq \boldsymbol{\beta}\,, \\ \mathbf{A_{eq}}\cdot\boldsymbol{\psi} = \boldsymbol{\beta_{eq}}\,, \\ \mathbf{lb} \leq \boldsymbol{\psi} \leq \mathbf{ub} \end{cases} \tag{24}$$

In the above, $\boldsymbol{\psi}$, $\mathbf{f}$, $\boldsymbol{\beta}$, $\boldsymbol{\beta_{eq}}$, $\mathbf{lb}$, and $\mathbf{ub}$ are vectors, and $\mathbf{A}$ and $\boldsymbol{A_{eq}}$ are matrices. To optimize the SVM using *linprog*, you will need to reformulate the SVM optimization problem (Eq. 23) in the form of Eq. 24. *(Note: Some of the arguments may not be needed.)*

6.3 Task 5: Optimizing SVM via Linear Programming

- Generate 1000 training (X_train, y_train) and 20000 test samples (X_test, y_test) with seed = 12345. No validation set is needed here. Ensure that you change the class labels returned by the data-generating function to produce y_train and y_test that is compatible with SVM: i.e. $y{=}{-}1$ for class with label 1, and $y{=}{+}1$ for class with label 2.
- (R) Implement function $train\_SVM\_linear\_progr$, which takes as input arguments (X_train, y_train) and finds optimal parameters $(\mathbf{w}, b)$ of the SVM using linear programming, using the Matlab function *linprog*. In your report, present in 10-15 lines of Matlab or pseudocode how you implemented it.
- Implement a function $svm(X, w, b)$ (or similar) that takes as input samples X and parameters of the SVM, and predicts classification score $\bar{y}$.
- (R) Using the above function, apply the SVM on the test data, with the parameters optimized by linear programming. Compute and report the train and test error of the predictions. Compare with the test error achieved previously with logistic regression.
- (R) Derive the function of the line that is the decision boundary learned by the SVM in the 2-dimensional plane defined by axes $x^{(1)}, x^{(2)}$. This is the line that separates the two classes, i.e. where $\bar{y} = 0$. Report the function in the form $x^{(2)} = \alpha \cdot x^{(1)} + \beta$, where α the slope and β the intercept, both as functions of $w^{(1)}, w^{(2)}, b$. Compare it with the decision boundary learned by logistic regression in previous tasks (line for which $y = 0.5$ there). You could even plot it, to help you get clearer insights in what you have learned.

7 Optimizing SVM with Gradient Descent

7.1 The Hinge Loss for optimizing an SVM

We here consider the same Linear SVM model as described in Sec. 6.1 and Eq. 22, where the model predicts a classification score $\bar{y} = \hat{\mathbf{x}}^T\boldsymbol{\theta}$ for a sample. If $\bar{y} < 0$, then $\hat{\mathbf{x}}$ is predicted to belong to the first class, whereas if $\bar{y} < 0$, then $\hat{\mathbf{x}}$ is predicted to belong to the second class. We will now use an alternative method for learning the optimal parameters $\boldsymbol{\theta}^\star$. We do not use additional slack parameters $\boldsymbol{\xi}$. It has been shown that the solution of Eq. 23, the optimal parameters $\boldsymbol{\theta}^\star$ that define the maximum margin decision boundary with least amount

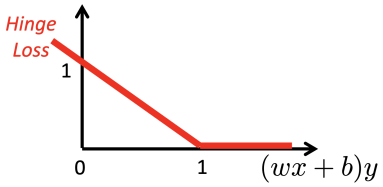


Figure 4: Hinge loss is a piece-wise linear function. It is not differentiable, but it is piece-wise differentiable. To use gradient descent, we will have to use sub-gradients, i.e. use the correct gradient for each input sample, depending on what value the loss has for the sample.

of error over the training set, can be equivalently found if we instead minimize another training objective, the average *Hinge Loss* $\mathcal{L}_{\text{H}}$ over our training data. The Hinge Loss is defined as:

$$\mathcal{L}_{\text{H}}(\boldsymbol{\theta}, \hat{\mathbf{x}}, y) = \max(0,\ 1 - y\bar{y}) = \max(0,\ 1 - y\ \hat{\mathbf{x}}^T\boldsymbol{\theta}) \ \in \mathbb{R} \tag{25}$$

As in Sec. 6.1, here y represents the *true* class of training sample $\hat{\mathbf{x}}$, and takes the value $y = -1$ if the true class of $\hat{\mathbf{x}}$ is the first class according to our training data, and $y = +1$ if $\hat{\mathbf{x}}$ belongs to the second class. We can then observe that the Hinge loss is 0 if the prediction $\hat{y}$ for a sample has the same sign as the true label y and $|\hat{\mathbf{x}}^T\boldsymbol{\theta}| > 1$, mapping $\hat{\mathbf{x}}$ beyond the desired margin 1. Else, it takes a positive value quantifying prediction error, playing equivalent role to ξ_i in Eq. 23b. Therefore, we can learn the optimal $\boldsymbol{\theta}^\star$ by minimizing the cost function J_H, which is the average Hinge Loss over our training data, and have:

$$J_{\text{H}}(\boldsymbol{\theta}, \hat{\mathbf{X}}, \mathbf{y}) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}_{\text{H}}(\boldsymbol{\theta}, \hat{\mathbf{x}}_i, y_i) \tag{26}$$

$$\boldsymbol{\theta}^\star = \arg\min_{\boldsymbol{\theta}} J_H(\boldsymbol{\theta}, \hat{\mathbf{X}}, \mathbf{y}) \tag{27}$$

We would like to optimize via Gradient Descent:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \lambda\frac{1}{n}\sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}}\mathcal{L}_H(\boldsymbol{\theta}, \hat{\mathbf{x}}_i, y_i) \tag{28}$$

The complication is that the Hinge loss is not fully differentiable. As Fig. 4 shows, its gradient when $\mathcal{L}_H = 1$ is undefined. The good news is that it is piece-wise differentiable, with two sub-gradients. To optimize it with Gradient Descent, we will therefore use the following form for its gradient:

$$\nabla_{\boldsymbol{\theta}}\mathcal{L}_{\text{H}}(\boldsymbol{\theta}, \hat{\mathbf{x}}_i, y_i) = \begin{cases} -y_i \cdot \hat{\mathbf{x}}_i & \text{if } \mathcal{L}_{\text{H}}(\boldsymbol{\theta}, \hat{\mathbf{x}}_i, y_i) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{29}$$

To optimize the SVM with GD, as requested in this task, you will need to implement Eq. 29, where for each training sample $\hat{\mathbf{x}}_i$, you will be assigning a different value of the gradient depending on whether the loss for the sample takes a value greater than 0 or not.

7.2 Task 6: Optimizing SVM via Gradient Descent and Hinge Loss

- As in previous tasks, generate X_train_val and y_train_val data with $n\_samples = 1000$ and divide samples into 80% (X_train, y_train) and 20% X_val, y_val samples. Also create 20000 test samples. Ensure that y takes values -1 and $+1$ for the 2 classes.

- (R) Implement function *train_SVM_hingeloss_gd* that performs (normal or stochastic) Gradient Descent to train the SVM by minimizing the Hinge loss. This could be closely similar to the functions implemented in Task 3 and 4, except the computation of gradients. In your report, present in a few lines of Matlab code (approx. 10 lines) how you implemented the gradient computation of Hinge loss.
- (R) Find appropriate values for the hyper-parameters of (S)GD such as λ, *n_iters* (and *batch_size* if SGD) following similar train/validation methodology as in previous tasks. In your report, show methodology used and values found.
- (R) Plot the cost J_H (mean loss) during training and how it evolves over iterations to observe how well optimization converges. Present it in your report.
- (R) Report training, validation and test error. Compare test error with that achieved previously by the SVM trained using linear-programming.
- (R) Report and compare the weights and bias learned by the SVM using (S)GD with those learned using linear programming. (*Note: converging to final value of parameters may take high n_iters, even if validation error does not change in the meantime.*)
- (R) Calculate and show in your report the decision boundary that the SVM learns and compare it with that learned using linear programming.
- (Optional) What is the maximum number of training samples that your machine's RAM memory allows you to use for linear programming optimization of SVM? Increase *n_samples* in Task 5 until no longer possible. If one has to train with a large database, will GD or SGD have similar issues?