Table of Contents

Submission by: ***Lê Nhựt Nam*** (20xx2023@student.hcmus.edu.vn)

```
student = (name = "Lê Nhựt Nam", id = "20xx2023")
```

Homework 5: Support Vector Networks

`CSC14005`, Introduction to Machine Learning

This notebook was built for FIT@HCMUS student to learn about Support Vector Machines/or Support Vector Networks in the course CSC14005 - Introduction to Machine Learning.

Instructions for homework and submission

It's important to keep in mind that the teaching assistants will use a grading support application, so you must strictly adhere to the guidelines outlined in the instructions. If you are unsure, please ask the teaching assistants or the lab instructors as soon as you can. **Do not follow your personal preferences at stochastically**

Instructions for doing homework

- You will work directly on this notebook; the word **TODO** indicates the parts you need to do.
- You can discuss the ideas as well as refer to the documents, but *the code and work must be yours*.

Instructions for submission

- Before submitting, save this file as `<ID>.jl`. For example, if your ID is 123456, then your file will be `123456.jl`. Submit that file on Moodle.

Danger

Note that you will get 0 point for the wrong submit.

Content of the assignment

- Recall: Perceptron & Geometriy Margin
- Linear support vector machine (Hard-margin, soft-margin)
- Popular non-linear kernels
- Computing SVM: Primal, Dual
- Multi-class SVM

Others

Other advice for you includes:

- Starting early and not waiting until the last minute
- Proceed with caution and gentleness.

"Living 'Slow' just means doing everything at the right speed – quickly, slowly, or at whatever pace delivers the best results." Carl Honoré.

- Avoid sources of interference, such as social networks, games, etc.

```julia
using Plots, Distributions, LinearAlgebra, Random
```

```
MersenneTwister(0)
```

```julia
Random.seed!(0)
```

Problem statement

Let $\mathcal{D} = \{(x_i, y_i) | x_i \in \mathbb{R}^d, y_i \in \{-1, 1\}\}_{i=1}^n$ be a dataset which is a set of pairs where $x_i \in \mathbb{R}^d$ is *data point* in some d-dimension vector space, and $y_i \in \{-1, 1\}$ is a *label* of the corespondent x_i data point classifying it to one of the two classes.

The model is trained on $\mathcal{D}$ after which it is present with x_{i+1}, and is asked to predict the label of this previously unseen data point.

The prediction function is donated by $f(x) : \mathbb{R}^d \to \{-1, 1\}$

Recall: Perceptron & Geometry Margin (Maximum 2.5 points)

In fact, it is always possible to come up with such a "perfect" binary function if training samples are distinct. However, it is unclear whether such rules are applicable to data that does not exist in the training set. We don't need "learn-by-heart" learners; we need "intelligent" learners. More especially, such trivial rules do not suffice because our task is not to correctly classify the training set. Our task is to find a rule that works well for all new samples we would encounter in the access control setting; the training set is merely a helpful source of information to find such a function. We would like to find a classifier that "generalizes" well.

The key to finding a generalized classifier is to constrain the set of possible binary functions we can entertain. In other words, we would like to find a class of classifier functions such that if a function in this class works well on the training set, it is also likely to work well on the unseen images. This problem is considered a key problem named "model selection" in machine learning.

```
md"""
## Recall: Perceptron & Geometry Margin (Maximum 2.5 points)

In fact, it is always possible to come up with such a "perfect" binary function if training samples are distinct. However, it is unclear whether such rules are applicable to data that does not exist in the training set. We don't need "learn-by-heart" learners; we need "intelligent" learners. More especially, such trivial rules do not suffice because our task is not to correctly classify the training set. Our task is to find a rule that works well for all new samples we would encounter in the access control setting; the training set is merely a helpful source of information to find such a function. We would like to find a classifier that "generalizes" well.

The key to finding a generalized classifier is to constrain the set of possible binary functions we can entertain. In other words, we would like to find a class of classifier functions such that if a function in this class works well on the training set, it is also likely to work well on the unseen images. This problem is considered a key problem named "model selection" in machine learning.
"""
```

Linear classifiers through origin

For simplicity, we will just fix the function class for now. We will only consider a type of *linear classifiers*. For more formally, we consider the function of the form:

$$f(\mathbf{x}, \theta) = \text{sign}(\theta_1\mathbf{x}_1 + \theta_2\mathbf{x}_2 + \cdots + \theta_d\mathbf{x}_d) = \text{sign}(\theta^\top \mathbf{x})$$

where $\theta = [\theta_1, \theta_2, \ldots, \theta_d]^\top$ is a column vector of real valued parameters.

Different settings of the parameters give different functions in this class, i.e., functions whose value or output in $\{-1, 1\}$ could be different for some input $\mathbf{x}$.

Perceptron Learning Algorithms

After chosen a class of functions, we still have to find a specific function in this class that works well on the training set. This task often refers to estimation problem in machine learning. We would like to find θ that minimize the *training error*, i.e we would like to find a linear classifier that make fewest mistake in the training set.

$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{t=1}^{n}(1 - \delta(y_t, f(\mathbf{x};\theta))) = \frac{1}{n}\sum_{t=1}^{n}\text{Loss}(y_t, f(\mathbf{x};\theta))$$

where $\delta(y, y') = 1$ if $y = y'$ and 0 if otherwise.

Perceptron update rule: Let k donates the number of parameter updates we have performed and $\theta^{(k)}$ is the parameter vector after k updates. Initially $k = 0$, and $\theta^{(k)} = 0$. We the loop through all the training instances $(\mathbf{x}_t, y)t)$, and updates the parameters only in response to mistakes,

$$\begin{cases}\theta^{(k+1)} \leftarrow \theta^{(k)} + y_t\mathbf{x}_t \text{ if } y_t(\theta^{(k+1)})^\top\mathbf{x}_t < 0\\ \text{The parameters unchanged}\end{cases}$$

Geometry intuition of Perceptron

8

```
begin
    n = 1000 # sample size
    d = 2; # dimensionality of data
    μ = 5 # mean
    Σ = 8 # variance
end
```

```
points1ₜᵣₐᵢₙ =
2×500 Matrix{Float64}:
 9.63346  5.39397   6.67285  12.4209   …   1.0458   8.02516  12.5985   4.6004
 8.10227  3.11485  -2.07827   4.62534     -3.02241  9.0917    8.29807  2.27181
```

```
points1ₜᵣₐᵢₙ = rand(MvNormal([Σ, μ], 5 .* [2 (μ - d)/Σ; (μ - d)/Σ d]), n ÷ 2)
```

```
points2ₜᵣₐᵢₙ =
2×500 Matrix{Float64}:
 0.0865891  -0.172354  -7.39168  -1.825    …   0.597771  -4.65396  -2.17438   4.42893
 9.37609    10.483      6.2005   11.6646      12.0652     6.94485   4.4954   10.4031
```

```
points2ₜᵣₐᵢₙ = rand(MvNormal([-μ+d, Σ+d], 5 .* [3 (μ - d)/μ; (μ - d)/μ d]), n ÷ 2)
```

```
points1ₜₑₛₜ =
2×500 Matrix{Float64}:
 10.4694   7.27176  10.6064   10.3368   …  10.399    6.32523  3.67006  4.26559
 -1.53045  3.94667   4.20854   4.10918      3.6226  10.3159   3.40714  2.8007
```

```
points1ₜₑₛₜ = rand(MvNormal([Σ, μ], 5 .* [2 (μ - d)/Σ; (μ - d)/Σ d]), n ÷ 2)
```

```
points2ₜₑₛₜ =
2×500 Matrix{Float64}:
 -5.15394  -9.12328  -4.35624  -2.15082  …  1.09356  -7.3331   -6.76925  1.35783
  9.48395   4.7303    6.69915   7.73966     6.90306   7.76199   7.93746  6.75461
```

```
points2ₜₑₛₜ = rand(MvNormal([-μ+d, Σ+d], 5 .* [3 (μ - d)/μ; (μ - d)/μ d]), n ÷ 2)
```

Todo

Your task here is implement the PLA (1 point). You can modify your own code in the area bounded by START YOUR CODE and END YOUR CODE.

```
md"""
!!! todo
    Your task here is implement the PLA (1 point). You can modify your own code in the area bounded by START YOUR CODE and END YOUR CODE.
"""
```

`pla`

```
Perceptron learning algorithm (PLA) implement function.
```

Fields

- pos_data::Matrix{Float64}: Input features for postive class (+1)
- neg_data::Matrix{Float64}: Input features for negative class (-1)
- n_epochs::Int64=10000: Maximum training epochs. Default is 10000
- η::Float64=0.03: Learning rate. Default is 0.03

```
"""
    Perceptron learning algorithm (PLA) implement function.

### Fields
- pos_data::Matrix{Float64}: Input features for postive class (+1)
- neg_data::Matrix{Float64}: Input features for negative class (-1)
- n_epochs::Int64=10000: Maximum training epochs. Default is 10000
- η::Float64=0.03: Learning rate. Default is 0.03
"""
function pla(pos_data::Matrix{Float64}, neg_data::Matrix{Float64},
    n_epochs::Int64=10000, η::Float64=0.03)
    # START YOUR CODE

    # END YOUR CODE
    # return θ
end
```

θₘₗ =

```
θₘₗ = pla(points1ₜᵣₐᵢₙ, points2ₜᵣₐᵢₙ)
```

`draw_pla`

```
Decision boundary visualization function for PLA
```

Fields

- θ: PLA paramters
- pos_data::Matrix{Float64}: Input features for postive class (+1)
- neg_data::Matrix{Float64}: Input features for negative class (-1)

```julia
"""
    Decision boundary visualization function for PLA

### Fields
- θ: PLA paramters
- pos_data::Matrix{Float64}: Input features for postive class (+1)
- neg_data::Matrix{Float64}: Input features for negative class (-1)
"""
function draw_pla(θ, pos_data::Matrix{Float64}, neg_data::Matrix{Float64})
    plt = scatter(pos_data[1, :], pos_data[2, :], label="y = 1")
    scatter!(plt, neg_data[1, :], neg_data[2, :], label="y = -1")

    b = θ[3]
    θₘₗ = θ[1:2]

    decision(x) = θₘₗ' * x + b

    D = ([
      tuple.(eachcol(pos_data), 1)
      tuple.(eachcol(neg_data), -1)
    ])

    xₘᵢₙ = minimum(map((p) -> p[1][1], D))
    yₘᵢₙ = minimum(map((p) -> p[1][2], D))
    xₘₐₓ = maximum(map((p) -> p[1][1], D))
    yₘₐₓ = maximum(map((p) -> p[1][2], D))

    contour!(plt, xₘᵢₙ:0.1:xₘₐₓ, yₘᵢₙ:0.1:yₘₐₓ,
            (x, y) -> decision([x, y]),
            levels=[0], linestyles=:solid, label="Decision boundary",
            colorbar_entry=false, color=:green)
end
```

```julia
# Uncomment this line below when you finish your implementation
# draw_pla(θₘₗ, points1ₜᵣₐᵢₙ, points2ₜᵣₐᵢₙ)
```

`tpfptnfn_cal`

```
Calculating values for True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN)
```

Fields

- y_test: Actual labels
- y_pred: Predicted labels

```julia
"""
    Calculating values for True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN)

### Fields
- y_test: Actual labels
- y_pred: Predicted labels
"""
function tpfptnfn_cal(y_test, y_pred, positive_class=1)
    true_positives = 0
    false_positives = 0
    true_negatives = 0
    false_negatives = 0

    # Calculate true positives, false positives, false negatives, and true negatives
    for (true_label, predicted_label) in zip(y_test, y_pred)
        if true_label == positive_class && predicted_label == positive_class
            true_positives += 1
        elseif true_label != positive_class && predicted_label == positive_class
            false_positives += 1
        elseif true_label == positive_class && predicted_label != positive_class
            false_negatives += 1
        elseif true_label != positive_class && predicted_label != positive_class
            true_negatives += 1
        end
    end

    return true_positives, false_positives, true_negatives, false_negatives
end
```

`eval_pla`

```
Evaluation function for PLA to calculate accuracy
```

Fields

- θ: PLA paramters
- pos_data::Matrix{Float64}: Input features for postive class (+1)
- neg_data::Matrix{Float64}: Input features for negative class (-1)

```julia
"""
    Evaluation function for PLA to calculate accuracy

### Fields
- θ: PLA paramters
- pos_data::Matrix{Float64}: Input features for postive class (+1)
- neg_data::Matrix{Float64}: Input features for negative class (-1)
"""
function eval_pla(θ, pos_data, neg_data)
    n = size(pos_data, 2)
    X = vcat(hcat(pos_data, neg_data), ones(n * 2)')

    y_test = vcat(ones(n), -ones(n))'
    y_pred = [sign(x) for x ∈ θ' * X]

    # START YOUR CODE
    # TODO: acc, p, r, f1???

    # END YOUR CODE

    print(" acc: $acc\n precision: $p\n recall: $r\n f1_score: $f1\n")
    return acc, p, r, f1
end
```

```julia
# Uncomment this line below when you finish your implementation
# eval_pla(θₘₗ, points1ₜₑₛₜ, points2ₜₑₛₜ)
```

Convergence Proof

Assume that all the training instances have bounded Euclidean norms), i.e $||\mathbf{x}|| \leq R$. Assume that exists a linear classifier in class of functions with finite parameter values that correctly classifies all the training instances. For precisely, we assume that there is some $\gamma > 0$ such that $y_t(\theta^*)^\top \mathbf{x}_t \geq \gamma$ for all $t = 1 .. . n$.

The convergence proof is based on combining two results:

- **Result 1**: we will show that the inner product $(\theta^*)^\top \theta^{(k)}$ increases at least linearly with each update.

Todo

Your task here is show the proof of result 1. (0.25 point)

```julia
md"""
!!! todo
Your task here is show the proof of result 1. (0.25 point)
"""
```

START YOUR PROOF

<content>

END YOUR PROOF

- **Result 2**: The squared norm $||\theta^{(k)}||^2$ increases at most linearly in the number of updates k.

Todo

Your task here is show the proof of result 2. (0.25 point)

```
md"""
!!! todo
Your task here is show the proof of result 2. (0.25 point)
"""
```

START YOUR PROOF

<content>

END YOUR PROOF

We can now combine parts 1) and 2) to bound the cosine of the angle between $\theta^{(k)}$ and θ^*. Since cosine is bounded by one, thus

$$1 \geq \frac{k\gamma}{\sqrt{kR^2}\left\|\theta^{(*)}\right\|} \leftrightarrow k \leq \frac{R^2\left\|\theta^{(*)}\right\|^2}{\gamma^2}$$

By combining the two we can show that the cosine of the angle between $\theta^{(k)}$ and θ^* has to increase by a finite increment due to each update. Since cosine is bounded by one, it follows that we can only make a finite number of updates.

Geometric margin & SVM Motivation

There is a question? Does $\frac{\|\theta^{(*)}\|^2}{\gamma^2}$ relate to how difficult the classification problem is? Its inverse, i.e., $\frac{\gamma^2}{\|\theta^{(*)}\|^2}$ is the smallest distance in the vector space from any samples to the decision boundary specified by $\theta^{(*)}$. In other words, it serves as a measure of how well the two classes of data are separated (by a linear boundary). We call this is gemetric margin, donated by γ_{geom}. As a result, the bound on the number of perceptron updates can be written more succinctly in terms of the geometric margin γ_{geom} (You know that man, Vapnik–Chervonenkis Dimension)

$$k \leq \left(\frac{R}{\gamma_{geom}}\right)^2$$

. We note some interesting thing about the result:

- Does not depend (directly) on the dimension of the data, nor
- number of training instances

Can't we find such a large margin classifier directly? YES, in this homework, you will do it with Support Vector Machine :)

Linear Support Vector Machine (Maximum 6 points)

From the problem statement section, we are given

$$\{(x_i, y_i) | x_i \in \mathbb{R}^{d}, y_i \in \{-1, 1\}\}_{i=1}^{n}$$

And based on previous section, we want to find the "maximum-geometric margin" that divides the space into two parts so that the distance between the hyperplane and the nearest point from either class is maximized. Any hyperplane can be written as the set of data points $\mathbf{x}$ satisfying

$$\boldsymbol{\theta}^\top\mathbf{x} + b = 0$$

```
md"""
## Linear Support Vector Machine (Maximum 6 points)

From the problem statement section, we are given

$\{(x_i, y_i) | x_i \in \mathbb{R}^{d}, y_i \in \{-1, 1\}\}_{i=1}^{n}$

And based on previous section, we want to find the "maximum-geometric margin" that divides the space into two parts so that the distance between the hyperplane and the nearest point from either class is maximized. Any hyperplane can be written as the set of data points $\mathbf{x}$ satisfying

$\mathbf{\theta}^\top\mathbf{x} + b = 0$
"""
```

Hard-margin

The goal of SVM is to choose two parallel hyperplanes that separate the two classes of data in order to maximize the distance between them. The region defined by these two hyperplanes is known as the "margin," and the maximum-margin hyperplane is the one located halfway between them. And these hyperplane can be decribed as

$$\theta^\top \mathbf{x} + b = 1 (\text{anything on or above this boundary is of one class, with label 1})$$

and

$$\theta^\top \mathbf{x} + b = -1 (\text{anything on or below this boundary is of the other class, with label -1})$$

Geometrically, the distance between these two hyperplanes is $\frac{2}{||\theta||}$

Todo

Your task here is show that the distance between these two hyperplanes is $\frac{2}{||\theta||}$ (1 point). You can modify your own code in the area bounded by START YOUR PROOF and END YOUR PROOF.

START YOUR PROOF

<content>

END YOUR PROOF

So we want to maximize the distance betweeen these two hyperplanes? Right? Equivalently, we minimize $||\theta||$. We also have to prevent data points from falling into the margin, we add the following constraint: for each i either

$$\theta^\top \mathbf{x}_i + b \geq 1 \text{ if } y_i = 1$$

and

$$\theta^\top \mathbf{x} + b \leq -1 \text{ if } y_i = -1$$

And, we can rewrite this as

$$y_i(\theta^\top \mathbf{x}_i + b) \geq 1, \forall i \in \{1 \ldots n\}$$

Finally, the optimization problem is

$$\min_{\theta, b} \frac{1}{2} ||\theta||^2$$
$$\text{s.t.} \quad y_i(\theta^\top \mathbf{x}_i + b) - 1 \geq 0, \forall i = 1 \ldots n$$

The parameters θ and b that solve this problem determine the classifier

$$\mathbf{x} \to \text{sign}(\theta^\top \mathbf{x}_i + b)$$

Todo

Your task here is implement the hard-margin SVM solving the primal formulation using gradient descent (3 points). You can modify your own code in the area bounded by START YOUR CODE and END YOUR CODE.

```
md"""
!!! todo
 Your task here is implement the hard-margin SVM solving the primal formulation using gradient descent (3 points). You can modify your own code in the area bounded by START YOUR CODE and END YOUR CODE.
"""
```

`hardmargin_svm`

```
SVM solving the primal formulation using gradient descent (hard-margin)
```

Fields

- pos_data::Matrix{Float64}: Input features for postive class (+1)
- neg_data::Matrix{Float64}: Input features for negative class (-1)
- η::Float64=0.03: Learning rate. Default is 0.03
- n_epochs::Int64=10000: Maximum training epochs. Default is 10000

```julia
"""
	SVM solving the primal formulation using gradient descent (hard-margin)
### Fields
- pos_data::Matrix{Float64}: Input features for postive class (+1)
- neg_data::Matrix{Float64}: Input features for negative class (-1)
- η::Float64=0.03: Learning rate. Default is 0.03
- n_epochs::Int64=10000: Maximum training epochs. Default is 10000
"""
function hardmargin_svm(pos_data, neg_data, η=0.04, n_epochs=10000)
	# START YOUR CODE

	## Create variables for the separating hyperplane w'*x = b.

	## Loss function

	# Train using gradient descent
	## For each epoch
	### For each training instance ∈ D

	## Update weight

	# END YOUR CODE
	## Return hyperplane parameters
	#return w, b
end
```

```julia
# Uncomment this line below when you finish your implementation
# w, b = hardmargin_svm(points1ₜᵣₐᵢₙ, points2ₜᵣₐᵢₙ)
```

`draw`

```
Visualization function for SVM solving the primal formulation using gradient descent (hard-margin)
```

Fields

- w & b: SVM parameters
- pos_data::Matrix{Float64}: Input features for postive class (+1)
- neg_data::Matrix{Float64}: Input features for negative class (-1)

```julia
"""
    Visualization function for SVM solving the primal formulation using gradient descent (hard-margin)

### Fields
- w & b: SVM parameters
- pos_data::Matrix{Float64}: Input features for postive class (+1)
- neg_data::Matrix{Float64}: Input features for negative class (-1)
"""
function draw(w, b, pos_data, neg_data)
    plt = scatter(pos_data[1, :], pos_data[2, :], label="y = 1")
    scatter!(plt, neg_data[1, :], neg_data[2, :], label="y = -1")

    hyperplane(x)= w' * x + b

    D = ([
      tuple.(eachcol(pos_data), 1)
      tuple.(eachcol(neg_data), -1)
    ])

    xₘᵢₙ = minimum(map((p) -> p[1][1], D))
    yₘᵢₙ = minimum(map((p) -> p[1][2], D))
    xₘₐₓ = maximum(map((p) -> p[1][1], D))
    yₘₐₓ = maximum(map((p) -> p[1][2], D))

    contour!(plt, xₘᵢₙ:0.1:xₘₐₓ, yₘᵢₙ:0.1:yₘₐₓ,
            (x, y) -> hyperplane([x, y]),
            levels=[-1],
            linestyles=:dash,
            colorbar_entry=false, color=:red, label = "Negative points")
    contour!(plt, xₘᵢₙ:0.1:xₘₐₓ, yₘᵢₙ:0.1:yₘₐₓ,
            (x, y) -> hyperplane([x, y]),
            levels=[0], linestyles=:solid, label="SVM prediction", colorbar_entry=false, color=:green)
    contour!(plt, xₘᵢₙ:0.1:xₘₐₓ, yₘᵢₙ:0.1:yₘₐₓ,
            (x, y) -> hyperplane([x, y]), levels=[1], linestyles=:dash, colorbar_entry=false, color=:blue, label = "Positive points")
end
```

```julia
# Uncomment this line below when you finish your implementation
# draw(w, b, points1ₜᵣₐᵢₙ, points2ₜᵣₐᵢₙ)
```

`eval_svm`

```
Evaluation function for hard-margin & soft-margin SVM to calculate accuracy
```

Fields

- θ: PLA paramters
- pos_data::Matrix{Float64}: Input features for postive class (+1)
- neg_data::Matrix{Float64}: Input features for negative class (-1)

```julia
"""
    Evaluation function for hard-margin & soft-margin SVM to calculate accuracy

### Fields
- θ: PLA paramters
- pos_data::Matrix{Float64}: Input features for postive class (+1)
- neg_data::Matrix{Float64}: Input features for negative class (-1)
"""
function eval_svm(w, b, pos_data, neg_data)
    n = size(pos_data, 2)
    X = hcat(pos_data, neg_data)

    # Actual labels, and predicted labels
    y_test = vcat(ones(n), -ones(n))'
    y_pred = [sign(x) for x ∈ w' * X .+ b]

    # START YOUR CODE
    # TODO: acc, p, r, f1???

    # END YOUR CODE

    print(" acc: $acc\n precision: $p\n recall: $r\n f1_score: $f1\n")

    return acc, p, r, f1
end
```

```julia
# Uncomment this line below when you finish your implementation
# eval_svm(w, b, points1ₜₑₛₜ, points2ₜₑₛₜ)
```

Soft-margin

The limitation of Hard Margin SVM is that it only works for data that can be separated linearly. In reality, however, this would not be the case. In practice, the data will almost certainly contain noise and may not be linearly separable. In this section, we will talk about soft-margin SVM (an relaxation of the optimization problem).

Basically, the trick here is very simple, we add slack variables ς_i to the constraint of the optimization problem.

$$y_i(\theta^\top \mathbf{x}_i + b) \geq 1 - \varsigma_i, \forall i = 1 \ldots n$$

The regularized optimization problem become as

$$\begin{aligned} &\min_{\theta, b, \varsigma} \frac{1}{2}\|\theta\|^2 + \sum_{i=1}^{n} \varsigma_i \\ \text{s.t.} \quad & y_i(\theta^\top \mathbf{x}_i + b) \geq 1 - \varsigma_i, \forall i = 1 \ldots n \end{aligned}$$

Furthermore, we ad a regularization parameter C to determine how important ς should be. And, we got it :)

$$\begin{aligned} &\min_{\theta, b, \varsigma} \frac{1}{2}\|\theta\|^2 + C\sum_{i=1}^{n} \varsigma_i \\ \text{s.t.} \quad & y_i(\theta^\top \mathbf{x}_i + b) \geq 1 - \varsigma_i, \varsigma_i \geq 0, \forall i = 1 \ldots n \end{aligned}$$

Todo

Your task here is implement the soft-margin SVM solving the primal formulation using gradient descent (3 points). You can modify your own code in the area bounded by START YOUR CODE and END YOUR CODE.

`softmargin_svm`

```
SVM solving the primal formulation using gradient descent (soft-margin)
```

Fields

- pos_data::Matrix{Float64}: Input features for postive class (+1)
- neg_data::Matrix{Float64}: Input features for negative class (-1)
- C: relaxation variable control slack variables ς
- η::Float64=0.03: Learning rate. Default is 0.03
- n_epochs::Int64=10000: Maximum training epochs. Default is 10000

```julia
"""
    SVM solving the primal formulation using gradient descent (soft-margin)
### Fields
- pos_data::Matrix{Float64}: Input features for postive class (+1)
- neg_data::Matrix{Float64}: Input features for negative class (-1)
- C: relaxation variable control slack variables ς
- η::Float64=0.03: Learning rate. Default is 0.03
- n_epochs::Int64=10000: Maximum training epochs. Default is 10000
"""
function softmargin_svm(pos_data, neg_data, n_epochs=10000, C=0.12, η=0.01)
    # START YOUR CODE

    ## Create variables for the separating hyperplane w'*x = b.

    ## Loss function

    # Train using gradient descent
    ## For each epoch
    ### For each training instance ∈ D

    #### Calculate slack variables ς

    ## Update weight

    # END YOUR CODE
    ## Return hyperplane parameters
    # return w, b
end
```

```julia
# Uncomment this line below when you finish your implementation
# sw, sb = softmargin_svm(points1ₜᵣₐᵢₙ, points2ₜᵣₐᵢₙ)
```

```julia
# Uncomment this line below when you finish your implementation
# draw(sw, sb, points1ₜᵣₐᵢₙ, points2ₜᵣₐᵢₙ)
```

```julia
# Uncomment this line below when you finish your implementation
# eval_svm(sw, sb, points1ₜₑₛₜ, points2ₜₑₛₜ)
```

Computing the SVM classifier (To get beyond 8.5 points)

We should know about some popular kernel types we could use to classify the data such as linear kernel, polynomial kernel, Gaussian, sigmoid and RBF (radial basis function) kernel.

- Linear Kernel: $K(x_i, x_j) = x_i^\top x_j$
- Polynomial kernel: $K(x_i, x_j) = (1 + x_i^\top x_j)^p$
- Gaussian: $K(x_i, x_j) = \text{exp}\left(-\frac{||x_i - x_j||^2}{2\sigma^2}\right)$
- Sigmoid: $K(x_i, x_j) = \text{tanh}(\beta_0x_i^\top x_j + \beta_1)^p$
- RBF kernel: $K(x_i, x_j) = \text{exp}(-\gamma||x_i - x_j||^2)$

```
md"""
## Computing the SVM classifier (To get beyond 8.5 points)

We should know about some popular kernel types we could use to classify the data such as linear kernel, polynomial kernel, Gaussian, sigmoid and RBF (radial basis function) kernel.
- Linear Kernel: $K(x_i, x_j) = x_i^\top x_j$
- Polynomial kernel: $K(x_i, x_j) = (1 + x_i^\top x_j)^p$
- Gaussian: $K(x_i, x_j) = \text{exp}\left(-\frac{||x_i - x_j||^2}{2\sigma^2}\right)$
- Sigmoid: $K(x_i, x_j) = \text{tanh}(\beta_0x_i^\top x_j + \beta_1)^p$
- RBF kernel: $K(x_i, x_j) = \text{exp}(-\gamma||x_i - x_j||^2)$
"""
```

`two_spirals`

```
Function for creating two spirals dataset.

You can check the MATLAB implement here: 6 functions for generating artificial
datasets, https://www.mathworks.com/matlabcentral/fileexchange/41459-6-functio
ns-for-generating-artificial-datasets
```

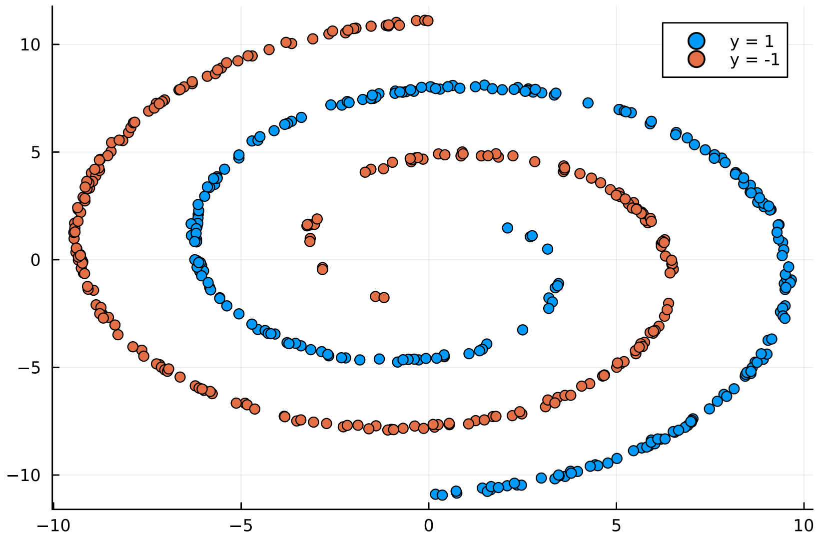
FIELDS

- $n_{samples}$: number of samples you want :)
- noise: noise rate for creating process you want :)

```julia
"""
    Function for creating two spirals dataset.

    You can check the MATLAB implement here: 6 functions for generating artificial
    datasets, https://www.mathworks.com/matlabcentral/fileexchange/41459-6-functions-
    for-generating-artificial-datasets
### FIELDS
- nₛₐₘₚₗₑₛ: number of samples you want :)
- noise: noise rate for creating process you want :)
"""
function two_spirals(nₛₐₘₚₗₑₛ, noise::Float64=0.2)
  start_angle = π / 2
  total_angle = 3π

  N₁ = floor(Int, nₛₐₘₚₗₑₛ / 2)
  N₂ = nₛₐₘₚₗₑₛ - N₁

  n = start_angle .+ sqrt.(rand(N₁, 1)) .* total_angle
  d₁ = [-cos.(n) .* n + rand(N₁, 1) .* noise sin.(n) .* n + rand(N₁, 1) .* noise]

  n = start_angle .+ sqrt.(rand(N₂, 1)) .* total_angle
  d₂ = [cos.(n) .* n + rand(N₂, 1) * noise -sin.(n) .* n + rand(N₂, 1) .* noise]

  return d₁', d₂'
end
```

```
(2×250 adjoint(::Matrix{Float64}) with eltype Float64:                        , 2×25
 -0.400044  9.12106  -2.66661  -5.44424  …   6.29711   6.9828    6.65789   8.85921   -6.
```

```julia
# create two spirals which are not linearly seperable
sp_points1, sp_points2 = two_spirals(500)
```




```
scatter!(scatter(sp_points1[1, :], sp_points1[2, :], label="y = 1"), sp_points2[1,
:], sp_points2[2, :], label="y = -1")
```

γ = 0.2

```
# Kernel function: in this lab, we use RBF kernel function, you want to do more
experiment, please try again at home
γ = 1 / 5
```

K (generic function with 1 method)

```
K(x, y) = exp(-γ * (x - y)' * (x - y))
```

SMO algorithm

For more detail, you should read: Platt, J. (1998). Sequential minimal optimization: A fast algorithm for training support vector machines.

Wikipedia just quite good for describes this algorithm: MO is an iterative algorithm for solving the optimization problem. MO breaks this problem into a series of smallest possible sub-problems, which are then solved analytically. Because of the linear equality constraint involving the Lagrange multipliers λ_i, the smallest possible problem involves two such multipliers.

The SMO algorithm proceeds as follows:

- Step 1: Find a Lagrange multiplier α_1 that violates the Karush–Kuhn–Tucker (KKT) conditions for the optimization problem.
- Step 2: Pick a second multiplier α_2 and optimize the pair (α_1, α_2)
- Step 3: Repeat steps 1 and 2 until convergence.

Dual SVM - Hard-margin

If you want to find minimum of a function f under the equality constraint g, we can use Largrangian function

$$f(x) - \lambda g(x) = 0$$

where λ is Lagrange multiplier.

In terms of SVM optimization problem

$$\begin{aligned} & \min_{\theta, b} \frac{1}{2} \|\theta\|^2 \\ \text{s.t.} \quad & y_i(\theta^\top \mathbf{x}_i + b) - 1 \geq 0, \forall i = 1 \ldots n \end{aligned}$$

The equality constraint is $g(\theta, b) = y_i(\theta^\top \mathbf{x}_i + b) - 1, \forall i = 1 \ldots n$

Then the Lagrangian function is

$$\mathcal{L}(\theta, b, \lambda) = \frac{1}{2} \|\theta\|^2 + \sum_1^n \lambda_i \left(y_i(\theta^\top \mathbf{x}_i + b) - 1 \right)$$

Equivalently, Lagrangian primal problem is formulated as

$$\begin{aligned} & \min_{\theta, b} \ \max \mathcal{L}(\theta, b, \lambda) \\ \text{s.t.} \quad & \lambda_i \geq 0, \forall i = 1 \ldots n \end{aligned}$$

Note

We need to MINIMIZE the MAXIMIZATION of $\mathcal{L}(\theta, b, \lambda)$? What we are doing???

Danger

More precisely, λ here should be KKT (Karush-Kuhn-Tucker) multipliers

$$\lambda[-y_i \left(\theta^\top \mathbf{x}_i + b \right) + 1] = 0, \forall i = 1 \ldots n$$

```
md"""
### Dual SVM - Hard-margin

If you want to find minimum of a function $f$ under the equality constraint $g$, we can use Largrangian function

$$f(x)-\lambda g(x)=0$$
where $\lambda$ is Lagrange multiplier.

In terms of SVM optimization problem

$$\begin{gather*}
    \underset{\theta, b}{\text{ min }}\frac{1}{2}\left\| \theta\right\|^2 \\
    \text{s.t.}\quad y_i(\mathbf{\theta}^\top \mathbf{x}_i+b) -1 \geq 0, \forall i = 1...n \\
  \end{gather*}$$

The equality constraint is $$g(\theta, b) = y_i(\mathbf{\theta}^\top \mathbf{x}_i+b) -1,\forall i = 1...n$$

Then the Lagrangian function is

$$\mathcal{L}(\theta, b, \lambda) = \frac{1}{2}\left\| \theta\right\|^2 + \sum_1^n \lambda_i\left(y_i(\mathbf{\theta}^\top \mathbf{x}_i+b)-1)\right)$$

Equivalently, Lagrangian primal problem is formulated as

$$\begin{gather*}
    \underset{\theta, b}{\text{ min }} {\text{ max }} \mathcal{L}(\theta, b, \lambda)\\
    \text{s.t.}\quad \lambda_i \geq 0, \forall i = 1...n \\
  \end{gather*}$$

!!! note
    We need to MINIMIZE the MAXIMIZATION of $\mathcal{L}(\theta, b, \lambda)$? What we are doing???

!!! danger
    More precisely, $\lambda$ here should be KKT (Karush-Kuhn-Tucker) multipliers

    $$\lambda [-y_i\left(\theta^\top\mathbf{x}_i + b\right) + 1] = 0, \forall i = 1...n$$
"""
```

With the Lagrangian function

$$\min_{\theta,b} \max \mathcal{L}(\theta, b, \lambda) = \frac{1}{2}\|\theta\|^2 + \sum_{i=1}^{n} \lambda_i \left(y_i(\theta^\top \mathbf{x}_i + b - 1)\right)$$
$$\text{s.t.} \quad \lambda_i \geq 0, \forall i = 1 \ldots n$$

Setting derivatives to 0 yield:

$$\nabla_\theta \mathcal{L}(\theta, b, \lambda) = \theta - \sum_{i=1}^{n} \lambda_i y_i \mathbf{x}_i = 0 \Leftrightarrow \theta^* = \sum_{i=1}^{n} \lambda_i y_i \mathbf{x}_i$$
$$\nabla_b \mathcal{L}(\theta, b, \lambda) = -\sum_{i=1}^{n} \lambda_i y_i = 0$$

We substitute them into the Lagrangian function, and get

$$W(\lambda, b) = \sum_{i=1}^{n} \lambda_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \lambda_i \lambda_j y_i y_j \mathbf{x}_i \mathbf{x}_j$$

So, dual problem is stated as

$$\max_{\lambda} \sum_{1}^{n} \lambda_i - \frac{1}{2} \sum_{i}^{n} \sum_{j}^{n} \lambda_i \lambda_j y_i y_j \mathbf{x}_i \mathbf{x}_j$$
$$\text{s.t.} \quad \lambda_i \geq 0, \forall i = 1 \ldots n, \sum_{i=1}^{n} \lambda_i yi = 0$$

To solve this one has to use quadratic optimization or **sequential minimal optimization**

```
draw_nl (generic function with 1 method)
```

```julia
function draw_nl(λ, b, pos_data, neg_data)
    plt = scatter(pos_data[1, :], pos_data[2, :], label="y = 1")
    scatter!(plt, neg_data[1, :], neg_data[2, :], label="y = -1")

    D = ([
      tuple.(eachcol(pos_data), 1)
      tuple.(eachcol(neg_data), -1)
    ])

    X = [x for (x, y) in D]
    Y = [y for (x, y) in D]

    k(x, y) = exp(-1 / 5 * (x - y)' * (x - y))

    hyperplane(x)= (λ .* Y) ⋅ k.(X, Ref(x)) + b

    xₘᵢₙ = minimum(map((p) -> p[1][1], D))
    yₘᵢₙ = minimum(map((p) -> p[1][2], D))
    xₘₐₓ = maximum(map((p) -> p[1][1], D))
    yₘₐₓ = maximum(map((p) -> p[1][2], D))

    contour!(plt, xₘᵢₙ:0.1:xₘₐₓ, yₘᵢₙ:0.1:yₘₐₓ,
            (x, y) -> hyperplane([x, y]),
            levels=[-1],
            linestyles=:dash,
            colorbar_entry=false, color=:red, label = "Negative points")
    contour!(plt, xₘᵢₙ:0.1:xₘₐₓ, yₘᵢₙ:0.1:yₘₐₓ,
            (x, y) -> hyperplane([x, y]),
            levels=[0], linestyles=:solid, label="SVM prediction",
            colorbar_entry=false, color=:green)
    contour!(plt, xₘᵢₙ:0.1:xₘₐₓ, yₘᵢₙ:0.1:yₘₐₓ,
            (x, y) -> hyperplane([x, y]), levels=[1], linestyles=:dash,
            colorbar_entry=false, color=:blue, label = "Positive points")
end
```

Todo

Your task here is implement the hard-margin SVM solving the dual formulation using sequential minimal optimization (2 points). You can modify your own code in the area bounded by START YOUR CODE and END YOUR CODE.

dualsvm_smo_hard (generic function with 4 methods)

```
function dualsvm_smo_hard(pos_data, neg_data, n_epochs=100, λₜₒₗ=0.0001,
errₜₒₗ=0.0001)
    # You do not need implement kernel, please use the K(.) kernel function in
    previous cell code.

    # START YOUR CODE
    # Step 1: Data preparation
    # First you construct and shuffle to obtain dataset D in a stochastically manner

    # For more easily access to data point
    X = [x for (x, y) ∈ D]
    Y = [y for (x, y) ∈ D]

    # Step 2: Initialization
    # Larangian multipliers, and bias
    λ = zeros(length(D))
    b = 0
    n = length(λ)

    # Step 3: Training loop


    # END YOUR CODE
    ## Return hyperplane parameters
    # return λ, b
end
```

```
# Uncomment this line below when you finish your implementation
# λₕ, bₕ = dualsvm_smo_hard(points1ₜᵣₐᵢₙ, points2ₜᵣₐᵢₙ)
```

```
# Uncomment this line below when you finish your implementation
# draw_nl(λₕ, bₕ, points1ₜᵣₐᵢₙ, points2ₜᵣₐᵢₙ)
```

Dual SVM - Soft-margin

As we know that, the regularized optimization problem in the case of soft-margin as

$$\min_{\theta,b,\varsigma} \frac{1}{2}\|\theta\|^2 + C\sum_{i=1}^{n}\varsigma_i$$
$$\text{s.t.} \quad y_i(\theta^\top \mathbf{x}_i + b) \geq 1 - \varsigma_i, \varsigma_i \geq 0, \forall i = 1\ldots n$$

We use Larangian multipliers, and transform to a dual problem as

$$\max_{\lambda} \sum_{1}^{n}\lambda_i - \frac{1}{2}\sum_{i}^{n}\sum_{j}^{n}\lambda_i\lambda_j y_i y_j \mathbf{x}_i\mathbf{x}_j$$
$$\text{s.t.} \quad 0 \leq \lambda_i \leq C, \forall i = 1\ldots n, \sum_{i=1}^{n}\lambda_i yi = 0$$

Todo

Your task here is implement the soft-margin SVM solving the dual formulation using sequential minimal optimization (2 points). You can modify your own code in the area bounded by START YOUR CODE and END YOUR CODE.

```
dualsvm_smo_soft (generic function with 5 methods)
```

```julia
function dualsvm_smo_soft(pos_data, neg_data, n_epochs=100, C=1000, λₜₒₗ=0.0001, errₜₒₗ=0.0001)
    # START YOUR CODE

    # Step 1: Data preparation
    # First you construct and shuffle to obtain dataset D in a stochastically manner

    # For more easily access to data point
    X = [x for (x, y) ∈ D]
    Y = [y for (x, y) ∈ D]

    # Step 2: Initialization
    # Larangian multipliers, and bias
    λ = zeros(length(D))
    b = 0
    n = length(λ)

    # Step 3: Training loop

    # END YOUR CODE
    ## Return hyperplane parameters
    # return λ, b
end
```

```julia
# Uncomment this line below when you finish your implementation
# λₛ, bₛ = dualsvm_smo_soft(sp_points1, sp_points2)
```

```julia
# Uncomment this line below when you finish your implementation
# draw_nl(λₛ, bₛ, sp_points1, sp_points2)
```

Multi-classes classification problem with SVMs (To get beyond 10.0 points)

```julia
md"""
## Multi-classes classification problem with SVMs (To get beyond 10.0 points)
"""
```

Load MNIST dataset

```julia
md"""
### Load MNIST dataset
"""
```

```
10000
```

```julia
begin
    data_dir = joinpath(dirname(@__FILE__), "data")
    train_x_dir = joinpath(data_dir, "train/images/train-images.idx3-ubyte")
    train_y_dir = joinpath(data_dir, "train/labels/train-labels.idx1-ubyte")

    test_x_dir = joinpath(data_dir, "test/images/t10k-images.idx3-ubyte")
    test_y_dir = joinpath(data_dir, "test/labels/t10k-labels.idx1-ubyte")

    NUMBER_TRAIN_SAMPLES = 60000
    NUMBER_TEST_SAMPLES = 10000
end
```

```
784×60000 Matrix{Float64}:
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 ⋮                        ⋮              ⋱                 ⋮
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

```julia
begin
    train_x = Array{Float64}(undef, 28^2, NUMBER_TRAIN_SAMPLES)
    train_y = Array{Int64}(undef, NUMBER_TRAIN_SAMPLES)

    io_images = open(train_x_dir)
    io_labels = open(train_y_dir)

    for i ∈ 1:NUMBER_TRAIN_SAMPLES
        seek(io_images, (i-1)*28^2 + 16) # offset 16 to skip header
        seek(io_labels, (i-1)*1 + 8) # offset 8 to skip header
        train_x[:,i] = convert(Array{Float64}, read(io_images, 28^2))
        train_y[i] = convert(Int, read(io_labels, UInt8))
    end
    close(io_images)
    close(io_labels)

    train_x = train_x
end
```

```
784×10000 Matrix{Float64}:
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 ⋮                        ⋮              ⋱                 ⋮
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

```julia
begin
    test_x = Array{Float64}(undef, 28^2, NUMBER_TEST_SAMPLES)
    test_y = Array{Int64}(undef, NUMBER_TEST_SAMPLES)

    io_images_test = open(test_x_dir)
    io_labels_test = open(test_y_dir)

    for i ∈ 1:NUMBER_TEST_SAMPLES
        seek(io_images_test, (i-1)*28^2 + 16) # offset 16 to skip header
        seek(io_labels_test, (i-1)*1 + 8) # offset 8 to skip header
        test_x[:,i] = convert(Array{Float64}, read(io_images_test, 28^2))
        test_y[i] = convert(Int, read(io_labels_test, UInt8))
    end
    close(io_images)
    close(io_labels)

    test_x = test_x
end
```

```
▸ ((784, 60000), (60000), (784, 10000), (10000))
```

```julia
size(train_x), size(train_y), size(test_x), size(test_y)
```

Training SVMs

```
md"""
### Training SVMs
"""
```

```
# START YOUR CODE

# END YOUR CODE
```

Evaluation

```
md"""
### Evaluation
"""
```

```
# START YOUR CODE

# END YOUR CODE
```

This is the end of Lab 05. However, there still a lot of things that you can learn about SVM. There are many open tasks to do in your sparse time such as how to deal with multi-class, or Bayesian SVM. :) Hope all you will enjoy SVM. Good luck!

References

[1] Boyd, S. P., & Vandenberghe, L. (2004). Convex optimization. Cambridge university press.

[2] Griva, I., Nash, S. G., & Sofer, A. (2008). Linear and Nonlinear Optimization 2nd Edition. Society for Industrial and Applied Mathematics.

[3] Schölkopf, B., & Smola, A. J. (2002). Learning with kernels: support vector machines, regularization, optimization, and beyond. MIT press.

[4] Lab 3, Logistic Regresion, Introduction to Machine Learning course, Department of Computer Science, Faculty of Information Technology, Ho Chi Minh University of Science, Vietnam National University.