Table of Contents

Submission by: ***Âu Dương Khang*** (21127621@student.hcmus.edu.vn)

```
student =   (name = "Âu Dương Khang", id = "21127621")
```

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.

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

```
MersenneTwister(0)
```

```
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.

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:

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

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,

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


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}:
 10.1475   6.88369  9.85505  8.20538  6.37392  …  9.90534  9.39467  14.3047   4.5988
  7.97587  4.37181  6.2714   4.69988  9.58528     5.184    6.07995   4.40377  8.39767
```

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

```
points2train =
2×500 Matrix{Float64}:
 -0.039622   2.09917  -4.5572  -5.43523  …  -1.45503   0.221639   3.81619  -4.32515
  6.12873   16.0328   10.0772  10.1525        9.43168  18.5044    12.0737   11.2626
```

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

```
points1test =
2×500 Matrix{Float64}:
 8.45704  12.2062   10.1817  4.4905   …  8.71818   1.39929   10.1025    6.06705
 7.78301   6.11425   1.5659  5.67126      6.96966  -0.825013   0.972006  5.28174
```

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

```
points2_test =
2×500 Matrix{Float64}:
 -3.61328  -7.89574  -2.70184  1.09431  -4.36052  …  -4.20653  -0.131558  2.79478
 10.2853   10.1288   10.0063   7.58113   8.98904      12.8835    6.43684   5.4429
```

```
points2_test = 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.

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
    # Initialize weights and bias
    θ = zeros(size(pos_data, 1) + 1)  # +1 for the bias term
    X = [pos_data neg_data]
    y = [ones(size(pos_data, 2)); -ones(size(neg_data, 2))]

    # Append a row of 1s for the bias term
    X = vcat(X, ones(1, size(X, 2)))

    # Training loop
    for epoch in 1:n_epochs
        for i in 1:size(X, 2)
            if sign(dot(θ, X[:, i])) != y[i]
                θ += η * y[i] * X[:, i]
            end
        end
    end
    # END YOUR CODE
    return θ
end
```

θ_{ml} = [0.645226, -0.830993, 1.65]

```
θ_ml = pla(points1_train, points2_train)
```

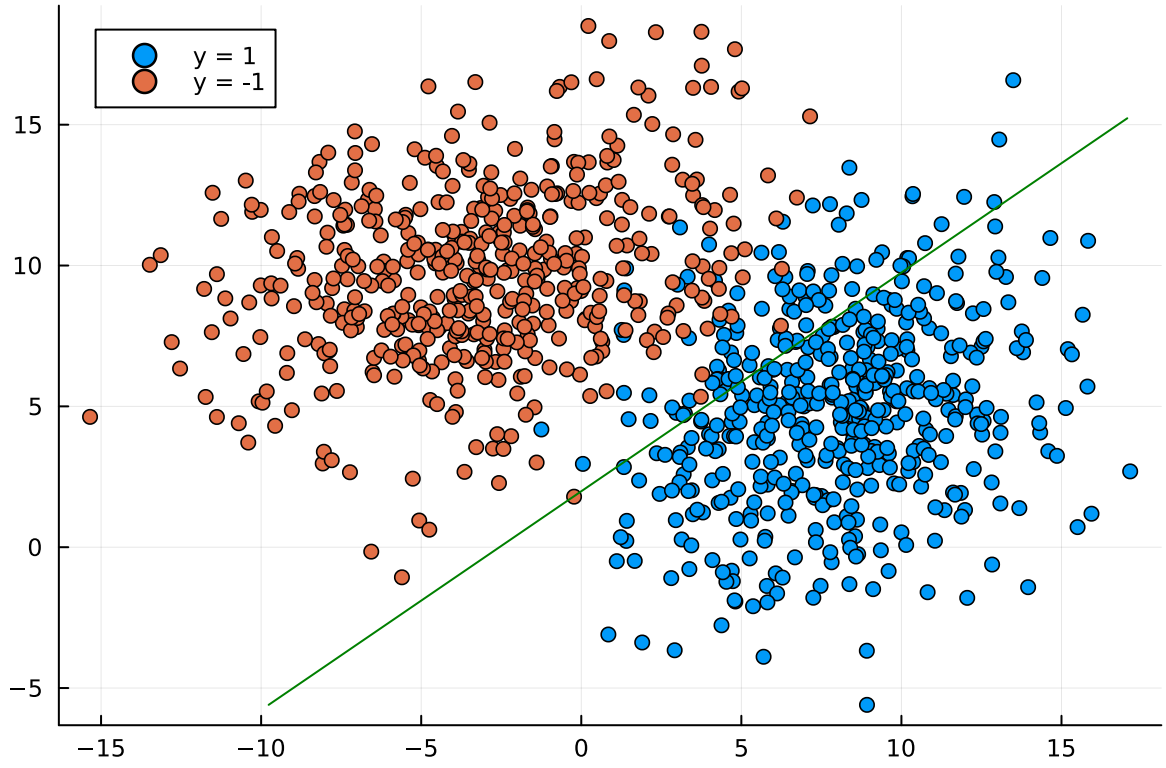
draw_pla

```
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)

```
"""
    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]
    θ_ml = θ[1:2]

    decision(x) = θ_ml' * x + b

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

    x_min = minimum(map((p) -> p[1][1], D))
    y_min = minimum(map((p) -> p[1][2], D))
    x_max = maximum(map((p) -> p[1][1], D))
    y_max = maximum(map((p) -> p[1][2], D))

    contour!(plt, x_min:0.1:x_max, y_min:0.1:y_max,
            (x, y) -> decision([x, y]),
            levels=[0], linestyles=:solid, label="Decision boundary",
            colorbar_entry=false, color=:green)
end
```




```
# Uncomment this line below when you finish your implementation
draw_pla(θ_ml, points1_train, points2_train)
```

tpfptnfn_cal

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

Fields

- y_test: Actual labels
- y_pred: Predicted labels

```
"""
    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)

```
"""
    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???
    tp, fp, tn, fn = tpfptnfn_cal(y_test, y_pred, 1)
    acc = (tp + tn) / (tp + fp + tn + fn)
    p = tp / (tp + fp)
    r = tp / (tp + fn)
    f1 =  2 * (p * r) / (p + r)
    # END YOUR CODE

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

(0.911, 0.997579, 0.824, 0.902519)

```
# Uncomment this line below when you finish your implementation
eval_pla(θ_ml, points1_test, points2_test)
```

```
acc: 0.911
precision: 0.997578692493946
recall: 0.824
f1_score: 0.9025191675794084
```

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)

START YOUR PROOF

Given:

- The θ^* is the true weight vector.
- The θ^k is the weight vector at iteration k.
- The η is the learning rate.
- The y_t is the true label of the misclassified instance $\mathbf{x}_t$.

The γ is a positive constant such that $y_t(\theta^*)^\top \mathbf{x}_t \geq \gamma > 0$ for all misclassified instances. We aim to show that the increase in $(\theta^*)^\top \theta^{(k)}$ after an update is at least linearly related to η.

Let's consider the inner product $(\theta^*)^\top \theta^{(k)}$ before an update as $P = (\theta^*)^\top \theta^{(k)}$.

After an update using a misclassified instance $\mathbf{x}_t$, the updated weight vector becomes

$$\theta^{(k+1)} = \theta^{(k)} + \eta y_t \mathbf{x}_t$$

The inner product after the update becomes

$$Q = (\theta^*)^\top \theta^{(k+1)} = (\theta^*)^\top (\theta^{(k)} + \eta y_t \mathbf{x}_t)$$

Using the distributive property of inner product, we get:

$$Q = (\theta^*)^\top \theta^{(k)} + \eta y_t (\theta^*)^\top \mathbf{x}_t$$

Given that $y_t(\theta^*)^\top \mathbf{x}_t \geq \gamma$, we can rewrite the equation as:

$$Q \geq P + \eta y_t \gamma$$

This shows that the increase $Q - P$ is at least linearly related to η, with a lower bound of $\eta y_t \gamma$.

Thus, $(\theta^*)^\top \theta^{(k)}$ increases at least linearly with each update in the Perceptron Learning Algorithm.

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

We aim to demonstrate that the squared norm of $\theta^{(k)}$, denoted as $||\theta^{(k)}||^2$, increases at most linearly with the number of updates k.

Given the update rule for the Perceptron Learning Algorithm:

$$\theta^{(k+1)} \leftarrow \theta^{(k)} + y_t \mathbf{x}_t \text{ if } y_t(\theta^{(k)})^\top \mathbf{x}_t < 0$$

We can express the squared norm at iteration $k+1$ as:

$$||\theta^{(k+1)}||^2 = ||\theta^{(k)} + y_t \mathbf{x}_t||^2$$

$$= ||\theta^{(k)}||^2 + 2(\theta^{(k)})^\top y_t \mathbf{x}_t + ||y_t \mathbf{x}_t||^2$$

$$\leq ||\theta^{(k)}||^2 + 2R \cdot ||\theta^{(k)}|| + R^2 \quad (\text{where } ||y_t \mathbf{x}_t|| \leq R)$$

$$= (||\theta^{(k)}|| + R)^2$$

This inequality shows that $||\theta^{(k+1)}||^2$ is bounded by a quadratic function of $||\theta^{(k)}||$, indicating that it increases at most linearly with the number of updates k. Therefore, $||\theta^{(k)}||^2$ increases at most linearly with k.

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

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

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

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

and

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

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

Given the equations of the hyperplanes:

- First one is: $\theta^\top \mathbf{x} + b = 1$ (for points on or above the boundary)
- The second one is: $\theta^\top \mathbf{x} + b = -1$ (for points on or below the boundary)

The distance between two parallel hyperplanes in the form $\theta^\top \mathbf{x} + b = c_1$ and $\theta^\top \mathbf{x} + b = c_2$ can be calculated as the absolute difference between their constant terms divided by the Euclidean norm of the normal vector θ.

Lets consider $c_1 = 1$ and $c_2 = -1$. Their difference:

$$Difference = |c_1 - c_2| = |1 - (-1)| = 2$$

The norm of the vector θ is given by $||\theta||$.

Therefore, the distance between the two hyperplanes is:

Distance = $\frac{Difference}{||\theta||} = \frac{2}{||\theta||}$

This demonstrates that the distance between the hyperplanes is indeed $\frac{2}{||\theta||}$

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

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

Finally, the optimization problem is

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

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.

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

```
"""
    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
     # Combine positive and negative data
    X = [pos_data neg_data]

    y = [ones(size(pos_data, 2)); -ones(size(neg_data, 2))]
    ## Create variables for the separating hyperplane w'*x = b.
    w = zeros(size(X, 1))  # Weight vector
    b = 0.0  # Bias term

    ## Loss function

    # Train using gradient descent

    ## For each epoch
    for epoch in 1:n_epochs
        # Initialize error flag
        error = false
    ### For each training instance ∈ D
        # Iterate over each training instance
        for i in 1:size(X, 2)
            xi = X[:, i]
            yi = y[i]

            ## Update weight
             if yi * (dot(w, xi) + b) <= 0
                w += η * yi * xi
                b += η * yi
                error = true
            end
```

```
        end

        # Check for convergence
        if !error
            break
        end
    end
    # END YOUR CODE
    ## Return hyperplane parameters
    return w, b
end
```

```
([0.860302, -1.10799], 2.2)
```

```
# Uncommment this line below when you finish your implementation
w, b = hardmargin_svm(points1_train, points2_train)
```

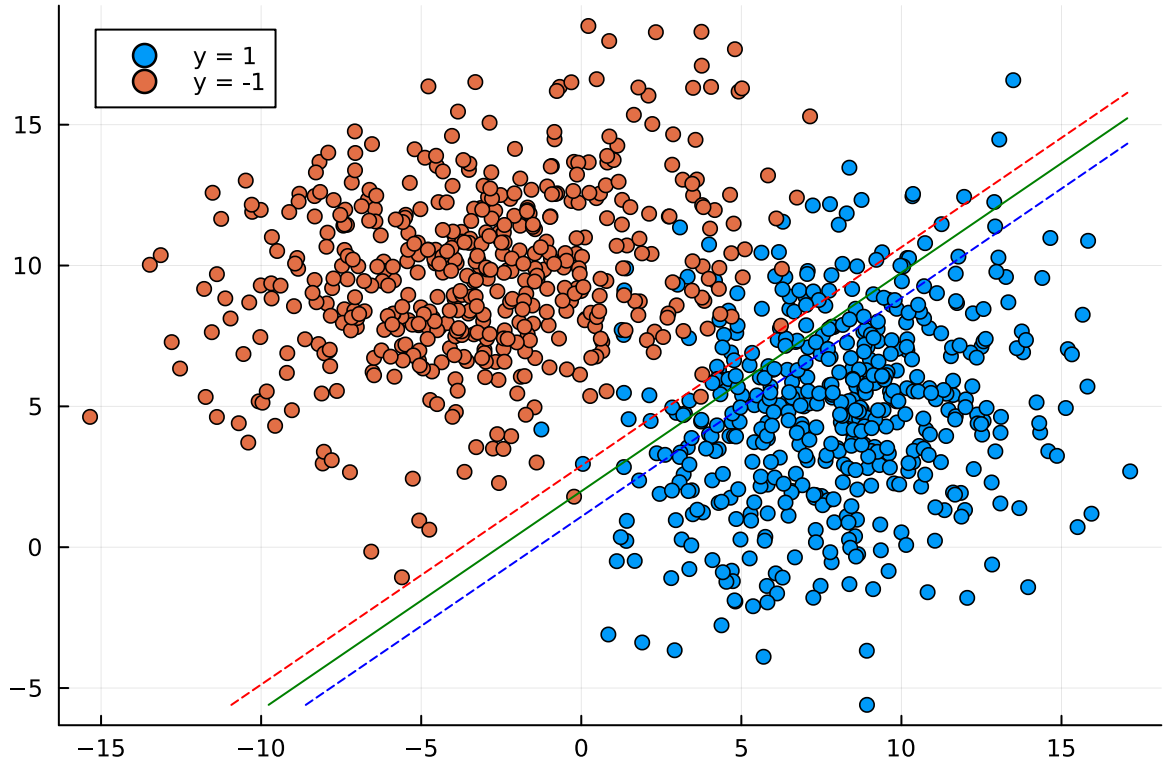
draw

```
Visualization function for SVM solving the primal formulation using gradient d
escent (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)

```
"""
    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
```




```
# 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)

```
"""
    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???
    tp_, fp_, tn_, fn_ = tpfptnfn_cal(y_test, y_pred, 1)
    acc = (tp_ + tn_) / (tp_ + fp_ + tn_ + fn_)
    p = tp_ / (tp_ + fp_)
    r = tp_ / (tp_ + fn_)
    f1 =  2 * (p * r) / (p + r)
    # END YOUR CODE

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

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

(0.911, 0.997579, 0.824, 0.902519)

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

```
acc: 0.911
precision: 0.997578692493946
recall: 0.824
f1_score: 0.9025191675794084
```

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.

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

The regularized optimization problem become as

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

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

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

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

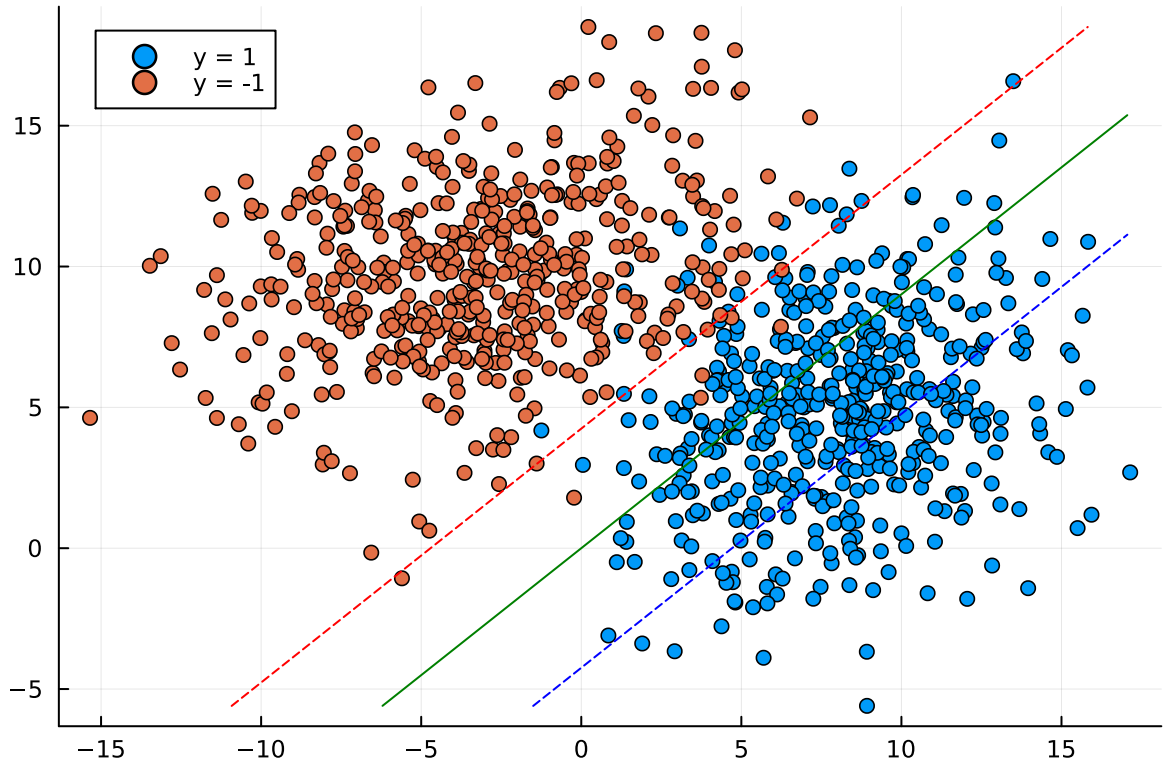
- pos_data::Matrix{Float64}: Input features for postive class (+1)
- neg_data::Matrix{Float64}: Input features for negative class (-1)
- C: relaxation variable control slack variables ς
- η::Float64=0.03: Learning rate. Default is 0.03
- n_epochs::Int64=10000: Maximum training epochs. Default is 10000

```
"""
    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
    # Combine positive and negative data
    X = [pos_data neg_data]

    # Create labels (+1 for positive, -1 for negative)
    y = [ones(size(pos_data, 2)); -ones(size(neg_data, 2))]

    ## Create variables for the separating hyperplane w'*x = b.
    w = zeros(size(X, 1))  # Weight vector
    b = 0.0  # Bias term

    ## Loss function

    # Train using gradient descent
    ## For each epoch
    for epoch in 1:n_epochs
    ### For each training instance ∈ D
    # Iterate over each training instance
        for i in 1:size(X, 2)
            xi = X[:, i]
            yi = y[i]

            # Calculate margin (yi * (w' * xi + b))
            margin = yi * (dot(w, xi) + b)

    #### Calculate slack variables ς
```

```
            slack = max(0, 1 - margin)

    ## Update weight
            w -= η * (C * w - yi * xi * (slack > 0))
            b -= η * (C * b - yi * (slack > 0))
        end
    end
    # END YOUR CODE
    ## Return hyperplane parameters
    return w, b
```

```
([0.212337, -0.235573], -6.62738e-6)
```

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




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

```
(0.851, 1.0, 0.702, 0.824912)
```

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

```
acc: 0.851
precision: 1.0
recall: 0.702
f1_score: 0.8249118683901292
```

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: $K(x_i, x_j) = \exp\left(-\frac{||x_i - x_j||^2}{2\sigma^2}\right)$
- Sigmoid: $K(x_i, x_j) = \tanh(\beta_0 x_i^\top x_j + \beta_1)^p$
- RBF kernel: $K(x_i, x_j) = \exp(-\gamma||x_i - x_j||^2)$

```
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
```

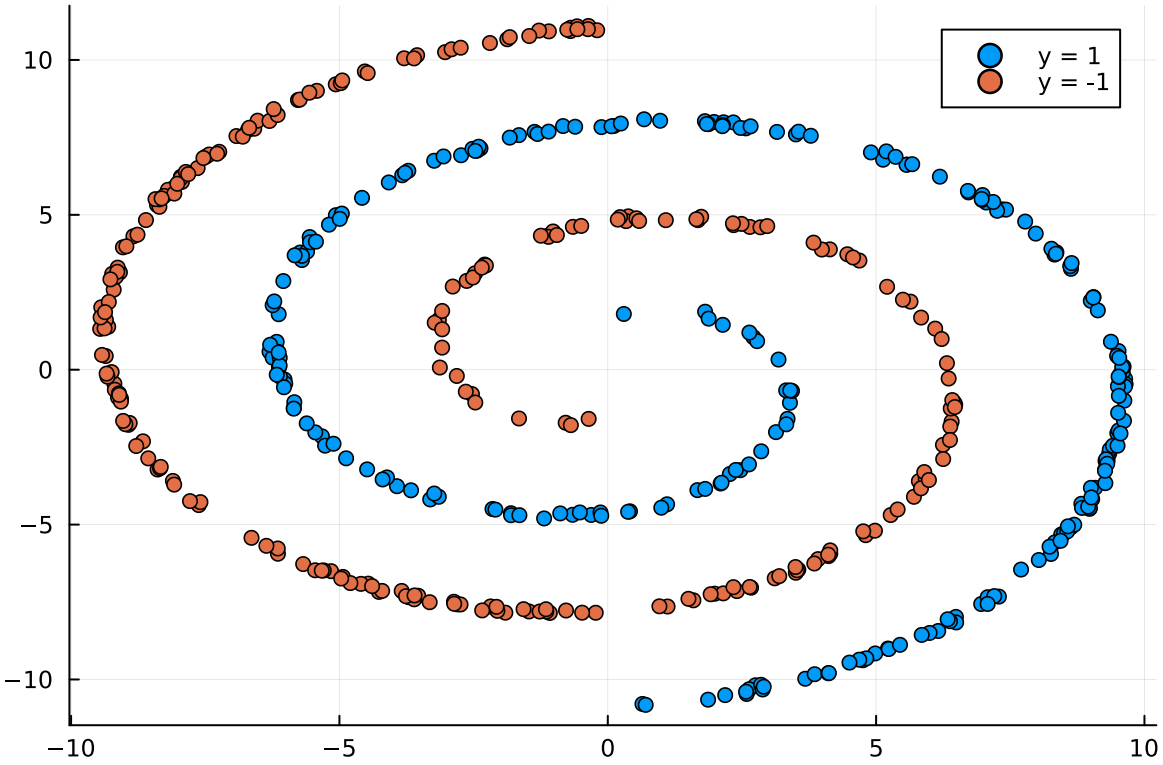
FIELDS

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

```
"""
    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
- n_samples: number of samples you want :)
- noise: noise rate for creating process you want :)
"""
function two_spirals(n_samples, noise::Float64=0.2)
  start_angle = π / 2
  total_angle = 3π

  N₁ = floor(Int, n_samples / 2)
  N₂ = n_samples - 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:                          , 2×250
   9.081     2.46954   2.27532   3.3497   …   9.00989  2.13588   9.55336  6.96648   -7.62
  -3.81222  -3.24412  -3.36782  -1.58574     -4.1235   7.86202  -2.05752  5.51086   -4.37
```

```
# 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

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

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

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

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

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

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

```
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)
```

```
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

    x_min = minimum(map((p) -> p[1][1], D))
    y_min = minimum(map((p) -> p[1][2], D))
    x_max = maximum(map((p) -> p[1][1], D))
    y_max = maximum(map((p) -> p[1][2], D))

    contour!(plt, x_min:0.1:x_max, y_min:0.1:y_max,
            (x, y) -> hyperplane([x, y]),
            levels=[-1],
            linestyles=:dash,
            colorbar_entry=false, color=:red, label = "Negative points")
    contour!(plt, x_min:0.1:x_max, y_min:0.1:y_max,
            (x, y) -> hyperplane([x, y]),
            levels=[0], linestyles=:solid, label="SVM prediction",
            colorbar_entry=false, color=:green)
    contour!(plt, x_min:0.1:x_max, y_min:0.1:y_max,
            (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.

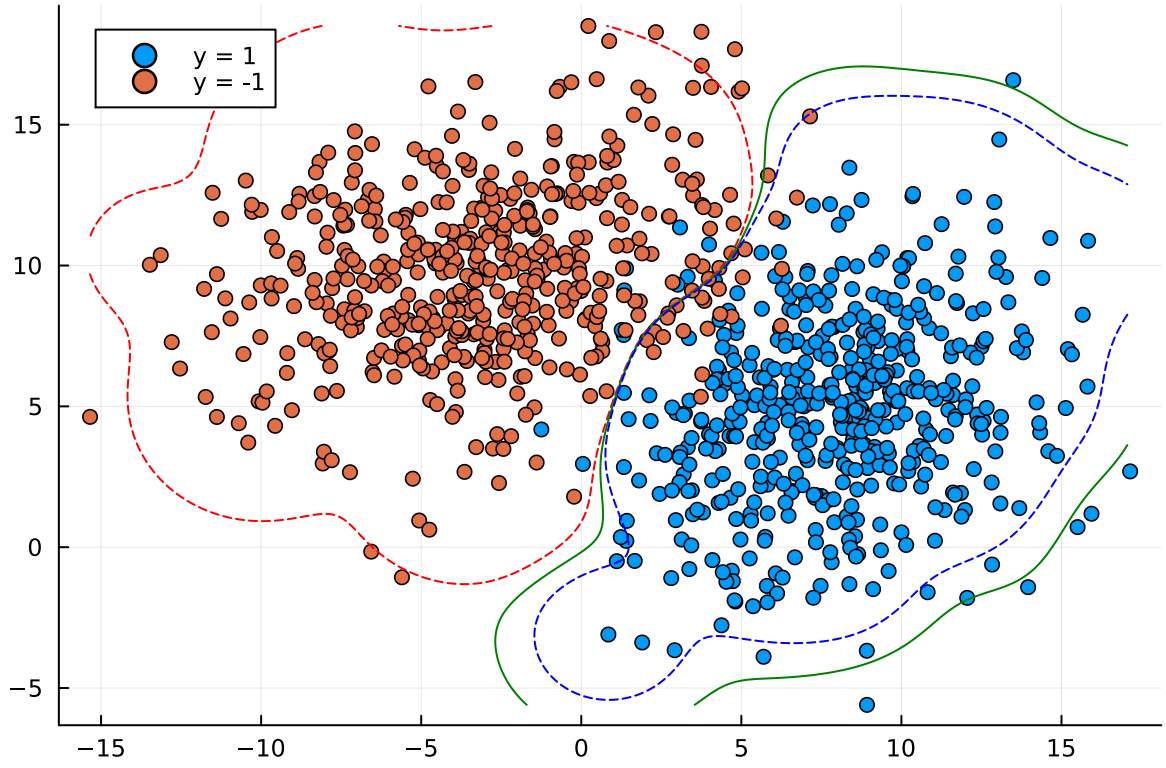
```
dualsvm_smo_hard (generic function with 4 methods)
```

```
function dualsvm_smo_hard(pos_data, neg_data, n_epochs=100, λ_tol=0.0001, err_tol=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
    D = [(x, 1) for x in eachcol(pos_data)]
    append!(D, [(x, -1) for x in eachcol(neg_data)])

    D = shuffle(D)
    # 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)
    C = Inf

    # Step 3: Training loop
    for epoch in 1:n_epochs
        num_changed_alphas = 0
        for i in 1:n
            E_i = b
            for j in 1:n
                E_i += λ[j] * Y[j] * dot(X[i], X[j])
            end
            E_i -= Y[i]

            if (Y[i] * E_i < -err_tol && λ[i] < C) || (Y[i] * E_i > err_tol && λ[i] >
                j = rand(1:n)  # Randomly select second alpha (j)
                while j == i
                    j = rand(1:n)
                end

                E_j = b
                for k in 1:n
                    E_j += λ[k] * Y[k] * dot(X[j], X[k])
                end
                E_j -= Y[j]

                old_λ_i, old_λ_j = copy(λ[i]), copy(λ[j])

                if Y[i] != Y[j]
                    L = max(0, λ[j] - λ[i])
                    H = min(C, C + λ[j] - λ[i])
                else
                    L = max(0, λ[i] + λ[j] - C)
```

```
                H = min(C, λ[i] + λ[j])
            end

            if L == H
                continue
            end

            eta = 2 * dot(X[i], X[j]) - dot(X[i], X[i]) - dot(X[j], X[j])
            if eta >= 0
                continue
            end

            λ[j] -= Y[j] * (E_i - E_j) / eta
            λ[j] = max(L, min(λ[j], H))

            if abs(λ[j] - old_λ_j) < err_tol
                continue
            end

            λ[i] += Y[i] * Y[j] * (old_λ_j - λ[j])
            b1 = b - E_i - Y[i] * (λ[i] - old_λ_i) * dot(X[i], X[i]) - Y[j] *
(λ[j] - old_λ_j) * dot(X[i], X[j])
            b2 = b - E_j - Y[i] * (λ[i] - old_λ_i) * dot(X[i], X[j]) - Y[j] *
(λ[j] - old_λ_j) * dot(X[j], X[j])
            b = (b1 + b2) / 2

            num_changed_alphas += 1
        end
    end

    if num_changed_alphas == 0
        break
    end
end
# END YOUR CODE
## Return hyperplane parameters
return λ, b
end
```

```
([0.0, 0.0, 0.0, 0.0, 0.0, 0.0227507, 0.0, 0.0, 0.0, 0.273806, 0.0, 0.00161508, 0.066941
```

```
# Uncomment this line below when you finish your implementation
λ_h, b_h = dualsvm_smo_hard(points1_train, points2_train)
```




```
# Uncomment this line below when you finish your implementation
draw_nl(λ_h, b_h, points1_train, points2_train)
```

Dual SVM - Soft-margin

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

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

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

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

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

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

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

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

$$\begin{gather*}
    \underset{\lambda}{\text{ max }}\sum_1^n\lambda_i -\frac{1}{2}\sum_i^n\sum_j^n\lambda_i\lambda_jy_iy_j\mathbf{x}_i\mathbf{x}_j\\
    \text{s.t.}\quad  0 \leq \lambda_i \leq C, \forall i = 1...n, \sum_{i=1}^n\lambda_iyi=0 \\
  \end{gather*}$$
"""
```

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.

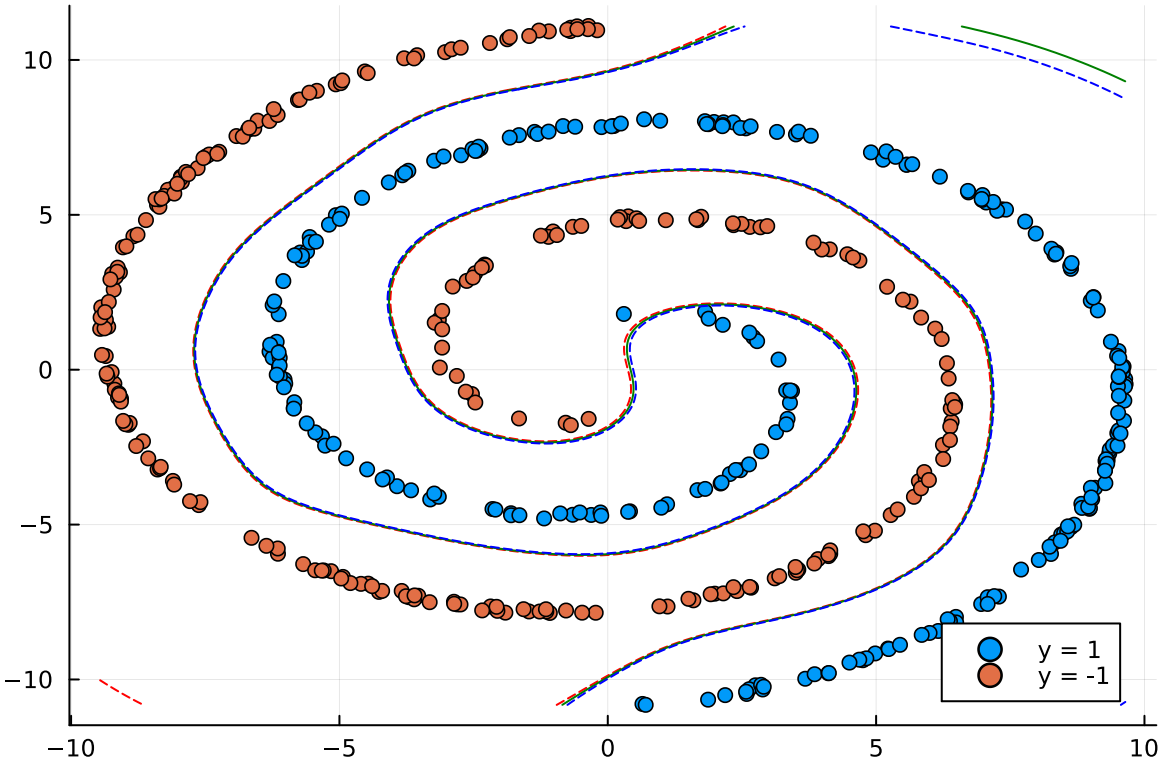
```
dualsvm_smo_soft (generic function with 5 methods)
```

```
function dualsvm_smo_soft(pos_data, neg_data, n_epochs=100, C=1000, λ_tol=0.0001, err_tol=0.0001)
    # START YOUR CODE

    # Step 1: Data preparation
    # First you construct and shuffle to obtain dataset D in a stochastically manner
    D = [(x, 1) for x in eachcol(pos_data)]
    append!(D, [(x, -1) for x in eachcol(neg_data)])

    D = shuffle(D)
    # 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
    for epoch in 1:n_epochs
        num_changed_alphas = 0
        for i in 1:n
            E_i = b
            for j in 1:n
                E_i += λ[j] * Y[j] * dot(X[i], X[j])
            end
            E_i -= Y[i]

            if (Y[i] * E_i < -err_tol && λ[i] < C) || (Y[i] * E_i > err_tol && λ[i] >
                j = rand(1:n)  # Randomly select second alpha (j)
                while j == i
                    j = rand(1:n)
                end

                E_j = b
                for k in 1:n
                    E_j += λ[k] * Y[k] * dot(X[j], X[k])
                end
                E_j -= Y[j]

                old_λ_i, old_λ_j = copy(λ[i]), copy(λ[j])

                if Y[i] != Y[j]
                    L = max(0, λ[j] - λ[i])
                    H = min(C, C + λ[j] - λ[i])
                else
                    L = max(0, λ[i] + λ[j] - C)
                    H = min(C, λ[i] + λ[j])
                end

                if L == H
```

```
                    continue
                end

                eta = 2 * dot(X[i], X[j]) - dot(X[i], X[i]) - dot(X[j], X[j])
                if eta >= 0
                    continue
                end

                λ[j] -= Y[j] * (E_i - E_j) / eta
                λ[j] = max(L, min(λ[j], H))

                if abs(λ[j] - old_λ_j) < err_tol
                    continue
                end

                λ[i] += Y[i] * Y[j] * (old_λ_j - λ[j])
                b1 = b - E_i - Y[i] * (λ[i] - old_λ_i) * dot(X[i], X[i]) - Y[j] *
(λ[j] - old_λ_j) * dot(X[i], X[j])
                b2 = b - E_j - Y[i] * (λ[i] - old_λ_i) * dot(X[i], X[j]) - Y[j] *
(λ[j] - old_λ_j) * dot(X[j], X[j])
                b = (b1 + b2) / 2

                num_changed_alphas += 1
            end
        end

        if num_changed_alphas == 0
            break
        end
    end
    # END YOUR CODE
    ## Return hyperplane parameters
    return λ, b
end
```

```
([0.499906, 2.34556, 0.0984003, 2.90495, 2.87907, 1.45962, 1.67945, 0.721028, 6.2426,
```

```
# Uncomment this line below when you finish your implementation
λ_s, b_s = dualsvm_smo_soft(sp_points1, sp_points2)
```




```
# 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)

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

Load MNIST dataset

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

```
10000
```

```
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
```

```
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
```

```
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))
```

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

Training SVMs

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

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

```
# STAR YOUR CODE
begin
    # Step 1: Preprocessing
    function preprocessing(train_x, test_x)
        # Normalize data
        train_x ./= 255.0
        test_x ./= 255.0

        # Reshape data for processing
        train_x = reshape(train_x, (28*28, :))
        test_x = reshape(test_x, (28*28, :))

        return train_x, test_x
    end

    # Step 2: Hinge Loss Calculation
    function hinge_loss(X, y, θ, b)
        # Hinge loss
        loss = 0.0
        for i in 1:size(X, 2)
            loss += max(0, 1 - y[i] * (dot(θ, X[:, i]) + b))
        end
        return loss / size(X, 2)
    end

    # Step 3: Gradient Descent
    function gradient_descent(X, y, θ, b, learning_rate)
        # Gradient calculation
        dθ = zeros(size(X, 1))
        db = 0.0
        for i in 1:size(X, 2)
            if y[i] * (dot(θ, X[:, i]) + b) < 1
                dθ -= y[i] * X[:, i]
                db -= y[i]
            end
        end

        # Update parameters
        θ -= learning_rate * (dθ / size(X, 2))
        b -= learning_rate * (db / size(X, 2))

        return θ, b
    end

    # Step 4: Training Loop
    function training_svms(train_x, train_y, epochs, learning_rate)
        θ = zeros(size(train_x, 1))  # Weight vector
        b = 0                        # Bias term
        losses = []

        for epoch in 1:epochs
            θ, b = gradient_descent(train_x, train_y, θ, b, learning_rate)
            loss = hinge_loss(train_x, train_y, θ, b)
```

```
            append!(losses, loss)
            println("Epoch: $epoch, Loss: $loss")
        end

        return 0, b, losses
    end
end
# END YOUR CODE
```

```
(784×60000 Matrix{Float64}:                                                , 7
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 ⋮                        ⋮              ⋱                 ⋮
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

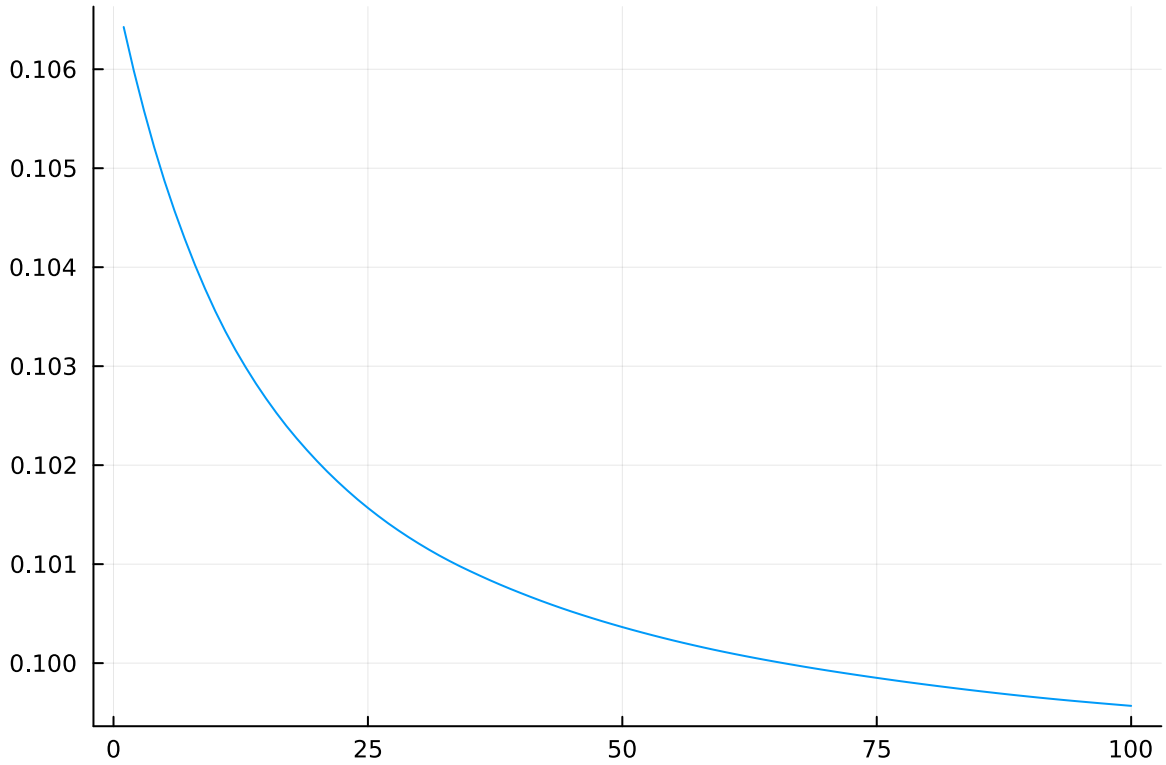
```
train_x_pre, test_x_pre = preprocessing(train_x, test_x)
```

```
([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,    more ,0.0], 0.0620397, [0.106427, 0.10598
```

```
begin
    learning_rate = 0.01
    epochs = 100
    0_train, b_train, losses = training_svms(train_x_pre, train_y, epochs,
    learning_rate)
end
```

```
Epoch: 1, Loss: 0.10642680728856593
Epoch: 2, Loss: 0.10598318134835877
Epoch: 3, Loss: 0.1055806013995576
Epoch: 4, Loss: 0.1052112844186988
Epoch: 5, Loss: 0.10487484581828599
Epoch: 6, Loss: 0.10456875914794629
Epoch: 7, Loss: 0.10428719974121524
Epoch: 8, Loss: 0.1040241928583897
Epoch: 9, Loss: 0.10378141132786278
Epoch: 10, Loss: 0.10355619323029042
Epoch: 11, Loss: 0.10335032456563671
Epoch: 12, Loss: 0.10316140769420129
Epoch: 13, Loss: 0.1029871890494009
Epoch: 14, Loss: 0.10282363331560172
Epoch: 15, Loss: 0.10267164097551325
Epoch: 16, Loss: 0.10252896512561684
Epoch: 17, Loss: 0.10239482816388358
Epoch: 18, Loss: 0.10227044106226138
Epoch: 19, Loss: 0.1021530301276678
Epoch: 20, Loss: 0.10204127993251737
Epoch: 21, Loss: 0.1019356388208509
Epoch: 22, Loss: 0.10183635102598283
Epoch: 23, Loss: 0.10174210468938368
Epoch: 24, Loss: 0.10165227527166068
Epoch: 25, Loss: 0.10156805629090504
Epoch: 26, Loss: 0.10148805139041638
Epoch: 27, Loss: 0.10141179793311264
Epoch: 28, Loss: 0.10134037167751127
Epoch: 29, Loss: 0.10127237173300827
Epoch: 30, Loss: 0.10120776984885506
Epoch: 31, Loss: 0.10114639377075695
Epoch: 32, Loss: 0.101088057133954
Epoch: 33, Loss: 0.10103293242977819
Epoch: 34, Loss: 0.10098060828159205
Epoch: 35, Loss: 0.10093136953468868
Epoch: 36, Loss: 0.10088365903992756
Epoch: 37, Loss: 0.1008371724499121
Epoch: 38, Loss: 0.10079270661218846
Epoch: 39, Loss: 0.10074992636456115
Epoch: 40, Loss: 0.10070883674766704
Epoch: 41, Loss: 0.10066866121906967
Epoch: 42, Loss: 0.10062988138592781
Epoch: 43, Loss: 0.10059252212080809
Epoch: 44, Loss: 0.10055634435099985
Epoch: 45, Loss: 0.10052151964386641
Epoch: 46, Loss: 0.10048807496744043
Epoch: 47, Loss: 0.10045572045309566
Epoch: 48, Loss: 0.10042406879061898
Epoch: 49, Loss: 0.10039376016946676
Epoch: 50, Loss: 0.10036419401380142
Epoch: 51, Loss: 0.10033546482567177
Epoch: 52, Loss: 0.10030787636142464
Epoch: 53, Loss: 0.10028082852878568
Epoch: 54, Loss: 0.10025460290857603
Epoch: 55, Loss: 0.10022918091952848
Epoch: 56, Loss: 0.10020466566295627
Epoch: 57, Loss: 0.10018091243837804
Epoch: 58, Loss: 0.10015761346639758
Epoch: 59, Loss: 0.10013518406067068
Epoch: 60, Loss: 0.1001136060538321
Epoch: 61, Loss: 0.10009274324846208
Epoch: 62, Loss: 0.10007246717678625
Epoch: 63, Loss: 0.10005271663873118
Epoch: 64, Loss: 0.10003350393217306
Epoch: 65, Loss: 0.10001493416272271
```

```
Epoch: 66, Loss: 0.09999677908210482
Epoch: 67, Loss: 0.09997904590028278
```




```
plot(1:epochs, losses, legend=false)
```

```
Epoch: 96, Loss: 0.09960321003225919
Epoch: 97, Loss: 0.09959428339620044
```

Evaluation

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

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

```
# START YOUR CODE
function accuracy(X, y, θ, b)
    correct = 0
    for i in 1:size(X, 2)
        if sign(dot(θ, X[:, i]) + b) == y[i]
            correct += 1
        end
    end
    return correct / size(X, 2)
end
# END YOUR CODE
```

```
# Evaluate on the training set
begin
    train_accuracy = accuracy(train_x, train_y, θ_train, b_train)
    println("Training Accuracy: $(train_accuracy * 100)%")

    # Evaluate on the test set
    test_accuracy = accuracy(test_x, test_y, θ_train, b_train)
    println("Test Accuracy: $(test_accuracy * 100)%")
end
```

```
Training Accuracy: 11.236666666666666%
Test Accuracy: 11.35%
```

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.