Table of Contents

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)
```

```
points2test =
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
```

```
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.

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

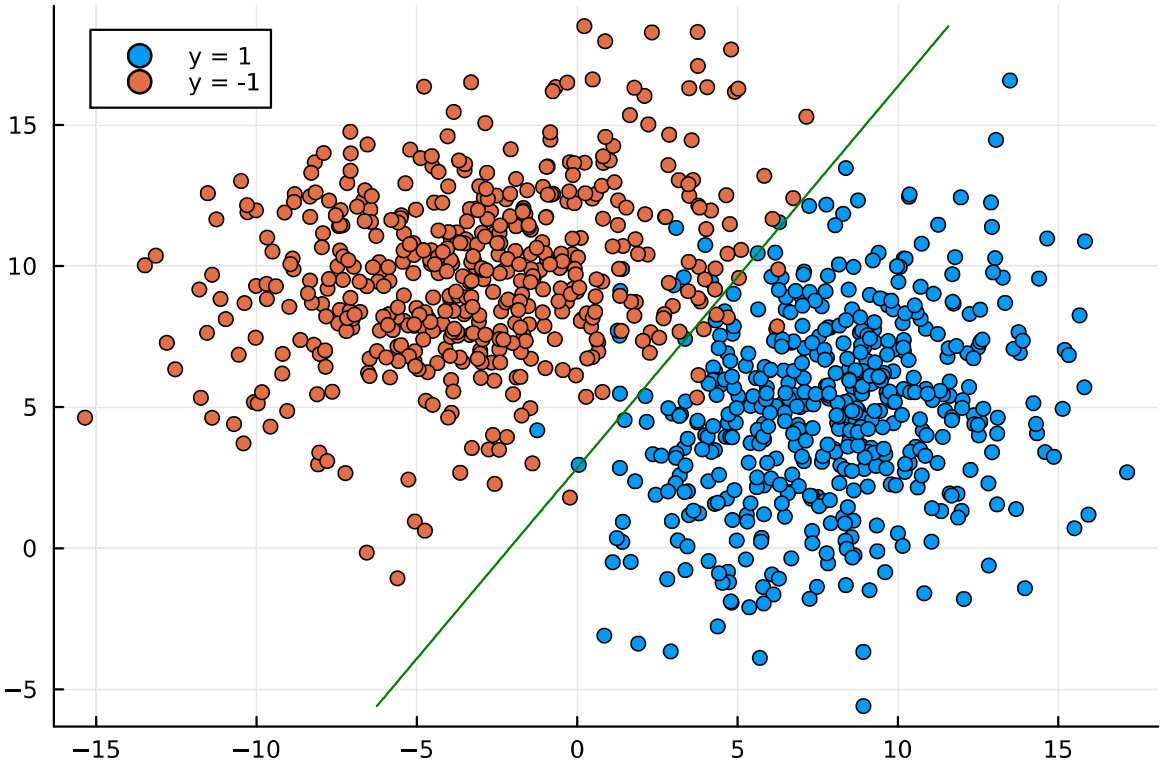
```
θml =  [0.9761, -0.721466, 2.0554]
```

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)




```
# 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 Negat
ives (TN), and False Negatives (FN)
```

Fields

- y_test: Actual labels
- y_pred: Predicted labels

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???
    acc = 0
    p = 0
    r = 0
    f1 = 0

    # Calculate true positives, false positives, false negatives, and true negatives
    true_positives, false_positives, true_negatives, false_negatives =
    tpfptnfn_cal(y_test,y_pred)

    # Catcutate precision, recatt, and F1-score
    acc += (true_positives + true_negatives) / (true_positives + false_positives +
    true_negatives + false_negatives)
    p += true_positives / (true_positives + false_positives)
    r += true_positives / (true_positives + false_negatives)

    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
```

```
(0.978, 0.983806, 0.972, 0.977867)
```

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

```
acc: 0.978
precision: 0.9838056680161945
recall: 0.972
f1_score: 0.977867203219316
```

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

Suppose we have $Q(k) = (\theta^*)^\top \theta^{(k)}$ and $Q(0) = \phi$ with $\phi \in R$
And we have : $\theta^{(k+1)} = \theta^{(k)} + y_t \mathbf{x}_t$ with $t = 1...n$

At k+1 : $Q(k+1) = (\theta^*)^\top \theta^{(k+1)} = (\theta^*)^\top (\theta^{(k)} + y_t \mathbf{x}_t) = (\theta^*)^\top \theta^{(k)} + y_t (\theta^*)^\top x_t \geq Q(k) + \gamma$

Conclusion : $Q(k)$ will increase by a maximum of γ for each update, or increase at least linearly of γ

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)

START YOUR PROOF

Suppose we have : $P(k) = ||\theta^{(k)}||^2 = (\theta^{(k)})^\top \theta^{(k)}$ và $P(0) = \rho$ with $\rho \in R$
And : $\theta^{(k+1)} = \theta^{(k)} + y_t \mathbf{x}_t$ với $t = 1...n$

At k+1 :
$P(k+1) = (\theta^{(k+1)})^\top \theta^{(k+1)} = (\theta^{(k)} + y_t \mathbf{x}_t)^\top (\theta^{(k)} + y_t \mathbf{x}_t) = ((\theta^k)^\top + y_t \mathbf{x}_t^\top)(\theta^{(k)} + y_t \mathbf{x}_t)$
$= (\theta^k)^\top (\theta^k) + y_t \mathbf{x}_t^\top \theta^{(k)} + y_t (\theta^k)^\top \mathbf{x}_t + y_t y_t \mathbf{x}_t^\top \mathbf{x}_t$
$= ||\theta^{(k)}||^2 + 2 y_t (\theta^k)^\top \mathbf{x}_t + ||x_t||^2 \leq P(k) + ||x_t||^2$

With $\mu = max(||x_t||^2) > 0$
We get: $P(k+1) \leq P(K) + \mu$

Conclusion : $P(k)$ increases at most linearly in the number of updates μ, or increase no more than linear rate of μ

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

We have $l_1 : \theta^\top \mathbf{x} + b = 1$ and $l_2 : \theta^\top \mathbf{x} + b = -1$ are 2 parallel hyperplanes in the same hyperspace (having $\theta^\top$ as the normal vector)
Given any $\mathbf{x_0}$ that belongs to hyperplane $l_1 : \theta^\top \mathbf{x} + b = 1$ we will have $\mathbf{x_0} = \frac{-b+1}{\theta^\top}$

So we have the distance between these 2 hyperplanes as below:

$$d(l_1, l_2) = d(x_0, l_2) = \frac{\theta^\top \mathbf{x_0} + b + 1}{||\theta^\top||} = \frac{\theta^\top \frac{-b+1}{\theta^\top} + b + 1}{||\theta^\top||} = \frac{2}{||\theta||}$$

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

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

Finally, the optimization problem is

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

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.

```
# Enter cell 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
    function check(w,b,x,y)
        return y*(w'*x+b)-1>=0
    end

    function update_w(w,b,x,y,state)
        return w - state*η*(w)
    end

    # Update b to ensure it's the same b
    function update_b(w,b,x,y,state)
        return b - state*η*b
    end

    function min_distance(w,b,x,y)
        return minimum([(y[j]*(w'*x[:,j]+b)-1) for j in  1:size(x)[2]])
    end

    pos_label = ones(size(pos_data)[2])
    neg_label = -ones(size(pos_data)[2])
    label = vcat(pos_label, neg_label)
    data = hcat(pos_data, neg_data)
    ## Create variables for the separating hyperplane w'*x = b.
    wb = pla(pos_data,neg_data)
    w = wb[1:2]
    b = wb[3]

    for i in 1:n_epochs
        all_good = true
        for j in randperm(size(data)[2])
            if !check(w,b,data[:,j],label[j])
                w = update_w(w,b,data[:,j],label[j],-1)
                b = update_b(w,b,data[:,j],label[j],-1)
```

```
                all_good =false
            end
        end
        if all_good
            b = b + (min_distance(w,b,neg_data,neg_label) -
            min_dictance(w,b,pos_data,pos_label))/2
            #Gradient descent
            w = update_w(w,b,data,label,1)
            b = update_b(w,b,data,label,1)
        end
        if norm(w) > 1e100 || norm(b) >1e100
            break
        end
    end
    # END YOUR CODE
    ## Return hyperplane parameters
    return w, b
end
```

```
([6.55678e99, -4.49138e99], 1.34355e100)
```

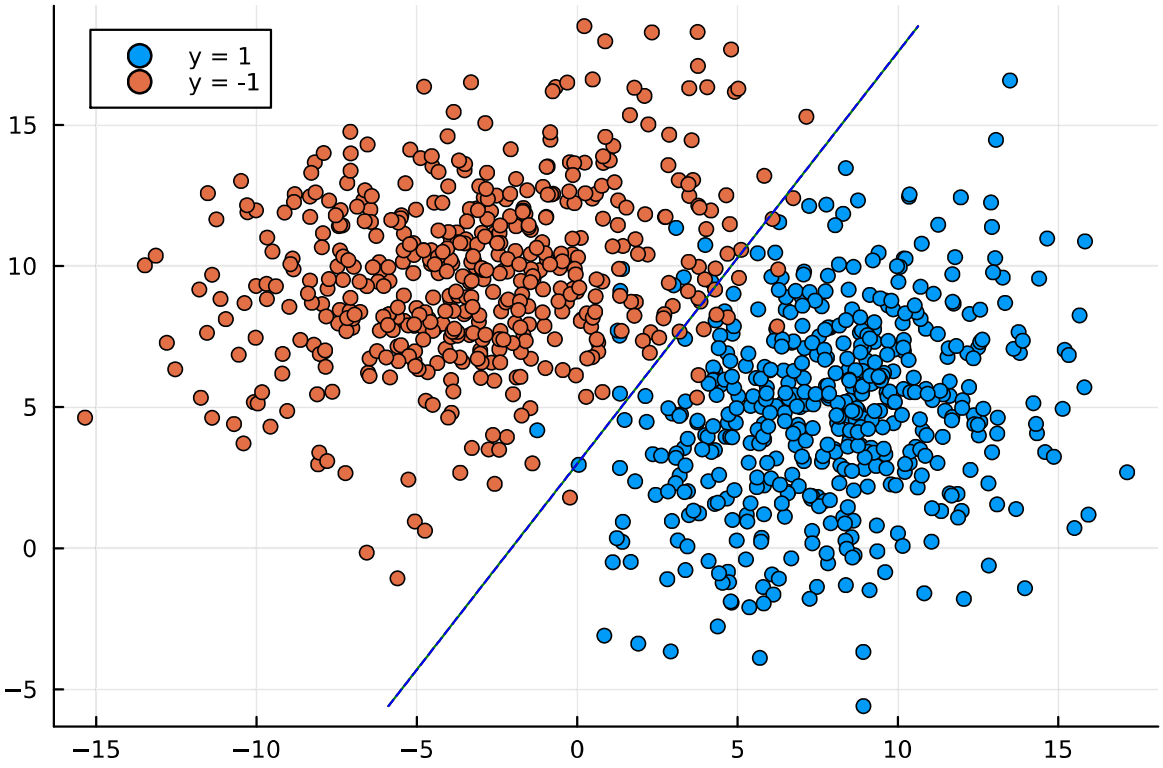
```
# Uncomment this line below when you finish your implementation
w, b = hardmargin_svm(points1train, points2train)
```

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)




```
# 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)

```
(0.981, 0.981964, 0.98, 0.980981)
```

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

```
acc: 0.981
precision: 0.9819639278557114
recall: 0.98
f1_score: 0.9809809809809811
```

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`

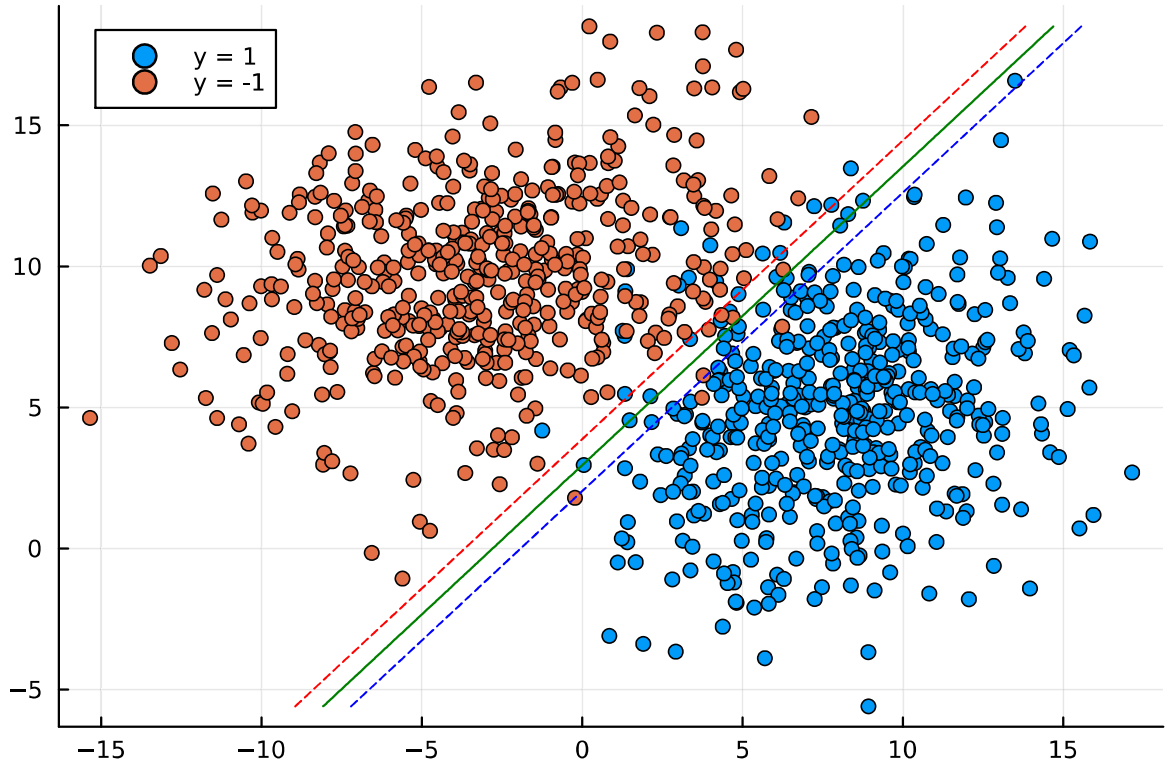
```
SVM solving the primal formulation using gradient descent (soft-margin)
```

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

```
([1.14802, -1.08417], 3.202)
```

```
# Uncomment this line below when you finish your implementation
sw, sb = softmargin_svm(points1_train, points2_train)
```




```
# Uncomment this line below when you finish your implementation
draw(sw, sb, points1_train, points2_train)
```

```
(0.967, 0.991579, 0.942, 0.966154)
```

```
# Uncomment this line below when you finish your implementation
eval_svm(sw, sb, points1_test, points2_test)
```

```
acc: 0.967
precision: 0.99157894736842
recall: 0.942
f1_score: 0.9661538461538461
```

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: $K(x_i, x_j) = \exp(-\gamma||x_i - x_j||^2)$

two_spirals

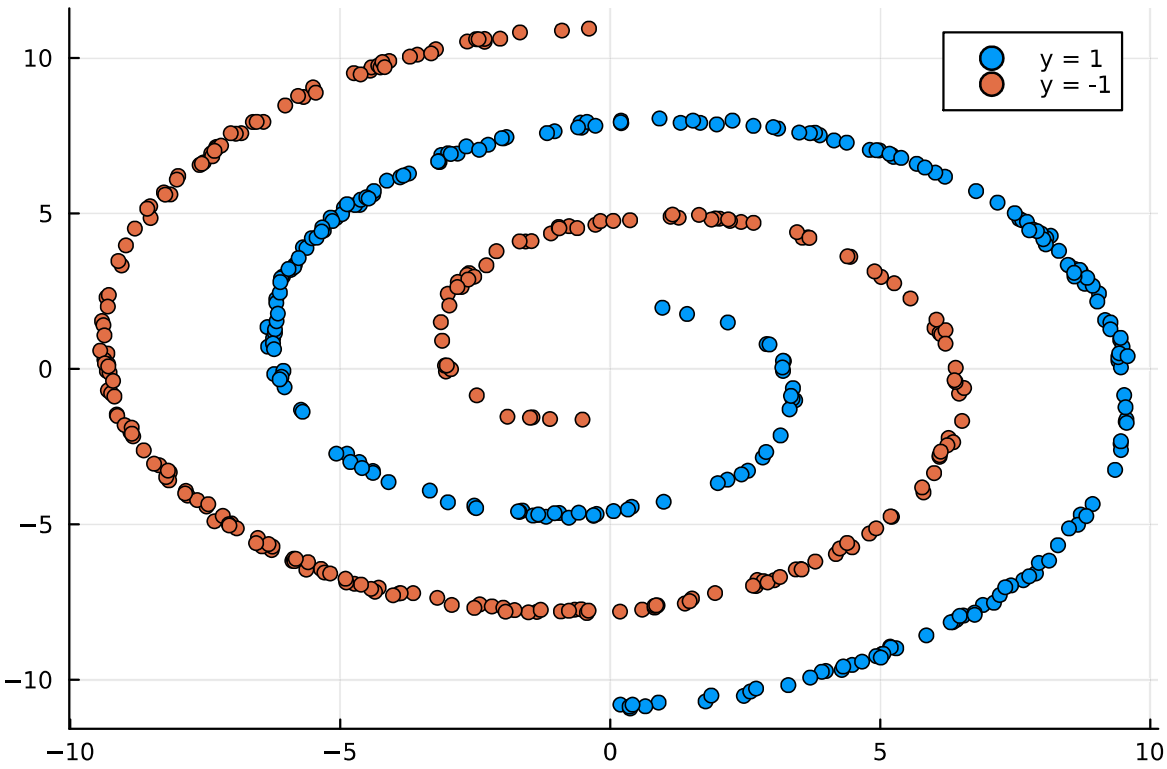
```
Function for creating two spirals dataset.

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

FIELDS

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

```
(2×250 adjoint(::Matrix{Float64}) with eltype Float64:                    , 2×2
  3.99336   3.38594   -6.06224  -0.942836  …   3.7047   5.38844  4.38062   7.3168   -7
 -9.71559  -0.614667   2.97254  -4.63471      -9.92429  6.78559  7.28102  -7.0207    7
```

```
# 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 $\boldsymbol{\lambda_i}$, the smallest possible problem involves two such multipliers.

The SMO algorithm proceeds as follows:

- Step 1: Find a Lagrange multiplier $\boldsymbol{\alpha_1}$ that violates the Karush–Kuhn–Tucker (KKT) conditions for the optimization problem.
- Step 2: Pick a second multiplier $\boldsymbol{\alpha_2}$ and optimize the pair $(\boldsymbol{\alpha_1}, \boldsymbol{\alpha_2})$
- 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

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

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

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

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

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

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)
```

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 5 methods)
```

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

    # Step 1: Data preparation

    pos_label = ones(size(pos_data)[2])
    neg_label = -ones(size(pos_data)[2])
    label = vcat(pos_label, neg_label)
    data = hcat(pos_data, neg_data)

    kernel = K

    λ = zeros(size(data)[2])
    b = 0
    n = length(λ)


    total_changed = 0
    do_all = 1
    epoch = 0


    # First you construct and shuffle to obtain dataset D in a stochastically manne
    function  compute_gram_matrix(X, kernel_func)
        gram_matrix = zeros(n, n)
        for i in 1:n
            for j in 1:n
                gram_matrix[i, j] = kernel_func(data[:,i], data[:,j])
            end
        end
        return gram_matrix
    end

    computed_gram_matrix = compute_gram_matrix(data, kernel)
    errors = zeros(2,n)

    function compute_error(x)
        error_x = (λ.*label)' * computed_gram_matrix[:,x] + b
        error_x = error_x - label[x]
        errors[1,x] = 1
        errors[2,x] = error_x
    end

    function update_alpha(i,j)
        if i==j
            return 0
        end
        alpha_i_old = λ[i]
        alpha_j_old = λ[j]
        if !(errors[1,i]==1)
            compute_error(i)
        end
        if !(errors[1,j]==1)
            compute_error(j)
        end
        error_i = errors[2,i]
```

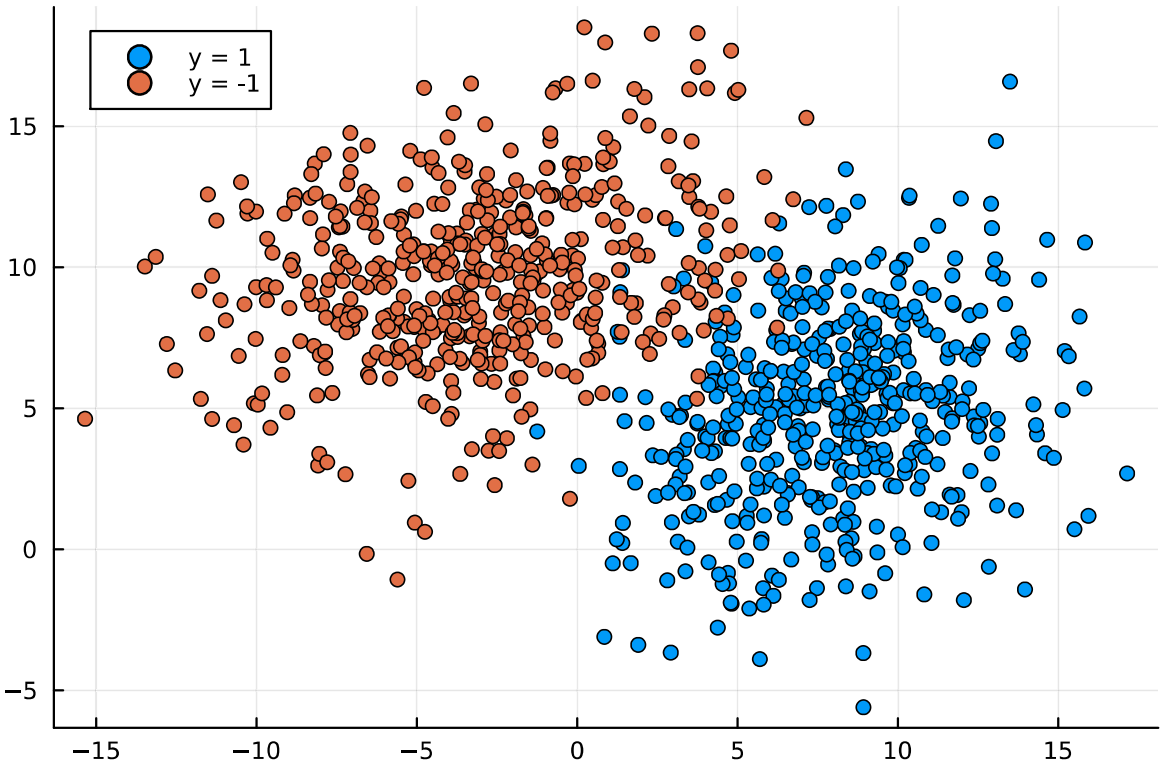
```
        error_j = errors[2,j]
        y_i = label[i]
        y_j = label[j]
        if y_i != y_j
            L = max(0, alpha_j_old - alpha_i_old)
        else
            L = max(0, alpha_i_old + alpha_j_old)
        end
        kii = computed_gram_matrix[i,i]
        kij = computed_gram_matrix[i,j]
        kjj = computed_gram_matrix[i,j]
        eta = 2*(kij - kii - kjj)
        if eta > 0
            alpha_j = alpha_j_old + y_j*(error_i-error_j)/eta
            if (alpha_j < L) alpha_j = L
            end
        else
            return 0
        end
        if (abs(alpha_j_old-alpha_j) < λtol*(alpha_j_old+alpha_j+λtol))
            return 0
        end
        alpha_i = alpha_i_old + y_i*y_j*(alpha_j_old-alpha_j)

        λ[j] = alpha_j
        λ[i] = alpha_i

        #f = computed_gram_matrix  * (λ.*label) .+ b
        #println(f)
        #errors = f - label

        b1 = b - error_i - y_i*(alpha_i - alpha_i_old)*kii - y_j*(alpha_j -
        alpha_j_old)*kij
        b2 = b - error_j - y_i*(alpha_i - alpha_i_old)*kij - y_j*(alpha_j -
        alpha_j_old)*kjj
        #println(b1,b2)
        if alpha_j > 0 && alpha_j < L
            b = b2
        elseif alpha_i > 0 && alpha_i < L
            b = b1
        else
            b = (b1+b2)/2
        end
        compute_error(i)
        compute_error(j)

        #sup_vec_M = [x>0 for x in λ]
        #sup_vec_S = [x>0 && x<C for x in λ]
        #b = sum(label[sup_vec_M] - sum(computed_gram_matrix[sup_vec_S,sup_vec_M]
        .* (λ[sup_vec_S].*label[sup_vec_S]), dims=1)')
        return 1
    end
    function process(j)
        y_j = label[j]
        alpha_j_old = λ[j]
        if !(errors[1,j]==1)
            compute_error(j)
        end
```

```
        error_j = errors[j]
        r_j = error_j*y_j
        if ((r_j < -err_tol) || (r_j > err_tol && alpha_j_old > 0))
            nonzero_alphas = (1:n)[[x>0 for x in λ]]
            if length(nonzero_alphas) > 0
                max_error = 0
                i=j
                 for x in nonzero_alphas
                     if errors[1,x]==1
                         compute_error(x)
                         if abs(errors[2,x] - error_j) > max_error
                             i=x
                         end
                     end
                 end
                #println("Heurictic here!",i,j)
                 if update_alpha(i,j)==1
                    return 1
                 end

            end
            for i in shuffle(nonzero_alphas)
                #println("nonzero_alphas here!",i,j)
                if update_alpha(i,j)==1
                    return 1
                end
            end
            for i in randperm(n)
                #println("random here!",i,j)
                if update_alpha(i,j)==1
                    return 1
                end
            end
        end
        return 0
    end

    # 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))

    while (total_changed > 0 || do_all ==1)
        epoch = epoch+1
        total_changed = 0
        if (do_all ==1)
            for j in randperm(n)
                total_changed = total_changed  + process(j)
            end
        else
            nonzero_alphas = (1:n)[[x>0 && x< C for x in λ]]
            for j in shuffle(nonzero_alphas)
                total_changed = total_changed  + process(j)
            end
        end
```

```
        if (do_all == 1)
            do_all = 0
        elseif (total_changed == 0)
            do_all = 1
        end
        #println(total_changed,' ',b)
        if epoch == n_epochs
            break
        end
    end
    # Step 3: Training loop
    # END YOUR CODE
    ## Return hyperplane parameters
    return λ, b
end
```

```
([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,    more ,0.0], 0)
```

```
# 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

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

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

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

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

    pos_label = ones(size(pos_data)[2])
    neg_label = -ones(size(pos_data)[2])
    label = vcat(pos_label, neg_label)
    data = hcat(pos_data, neg_data)

    kernel = K

    λ = zeros(size(data)[2])
    b = 0
    n = length(λ)

    total_changed = 0
    do_all = 1
    epoch = 0

    # First you construct and shuffle to obtain dataset D in a stochastically manne
    function  compute_gram_matrix(X, kernel_func)
        gram_matrix = zeros(n, n)
        for i in 1:n
            for j in 1:n
                gram_matrix[i, j] = kernel_func(data[:,i], data[:,j])
            end
        end
        return gram_matrix
    end

    computed_gram_matrix = compute_gram_matrix(data, kernel)
    errors = zeros(2,n)

    function compute_error(x)
        error_x = (λ.*label)' * computed_gram_matrix[:,x] + b
        error_x = error_x - label[x]
        errors[1,x] = 1
        errors[2,x] = error_x
    end
    function update_alpha(i,j)
        if i==j
            return 0
        end
        alpha_i_old = λ[i]
        alpha_j_old = λ[j]
        if !(errors[1,i]==1)
            compute_error(i)
        end
        if !(errors[1,j]==1)
            compute_error(j)
        end
        error_i = errors[2,i]
        error_j = errors[2,j]
        y_i = label[i]
        y_j = label[j]
```

```
        if y_i != y_j
            L = max(0, alpha_j_old - alpha_i_old)
            H = min(C, C + alpha_j_old - alpha_i_old)
        else
            L = max(0, alpha_i_old + alpha_j_old - C)
            H = min(C, alpha_i_old + alpha_j_old)
        end
        if L==H
            return 0
        end
        kii = computed_gram_matrix[i,i]
        kij = computed_gram_matrix[i,j]
        kjj = computed_gram_matrix[i,j]
        eta = kii+kjj-2*kij
        if eta > 0
            alpha_j = alpha_j_old + y_j*(error_i-error_j)/eta
            if (alpha_j < L) alpha_j = L
            elseif (alpha_j > H) alpha_j = H
            end
        else
            return 0
        end
        if (abs(alpha_j_old-alpha_j) < λtol*(alpha_j_old+alpha_j+λtol))
            return 0
        end
        alpha_i = alpha_i_old + y_i*y_j*(alpha_j_old-alpha_j)

        λ[j] = alpha_j
        λ[i] = alpha_i

        #f = computed_gram_matrix  * (λ.*label) .+ b
        #println(f)
        #errors = f - label

        b1 = b - error_i - y_i*(alpha_i - alpha_i_old)*kii - y_j*(alpha_j -
        alpha_j_old)*kij
        b2 = b - error_j - y_i*(alpha_i - alpha_i_old)*kij - y_j*(alpha_j -
        alpha_j_old)*kjj
        #println(b1,b2)
        if alpha_j > L && alpha_j < H
            b = b2
        elseif alpha_i > L && alpha_i < H
            b = b1
        else
            b = (b1+b2)/2
        end
        compute_error(i)
        compute_error(j)

        #sup_vec_M = [x>0 for x in λ]
        #sup_vec_S = [x>0 && x<C for x in λ]
        #b = sum(label[sup_vec_M] - sum(computed_gram_matrix[sup_vec_S,sup_vec_M]
        .* (λ[sup_vec_S].*label[sup_vec_S]), dims=1)')
        return 1
    end
    function process(j)
        y_j = label[j]
        alpha_j_old = λ[j]
```

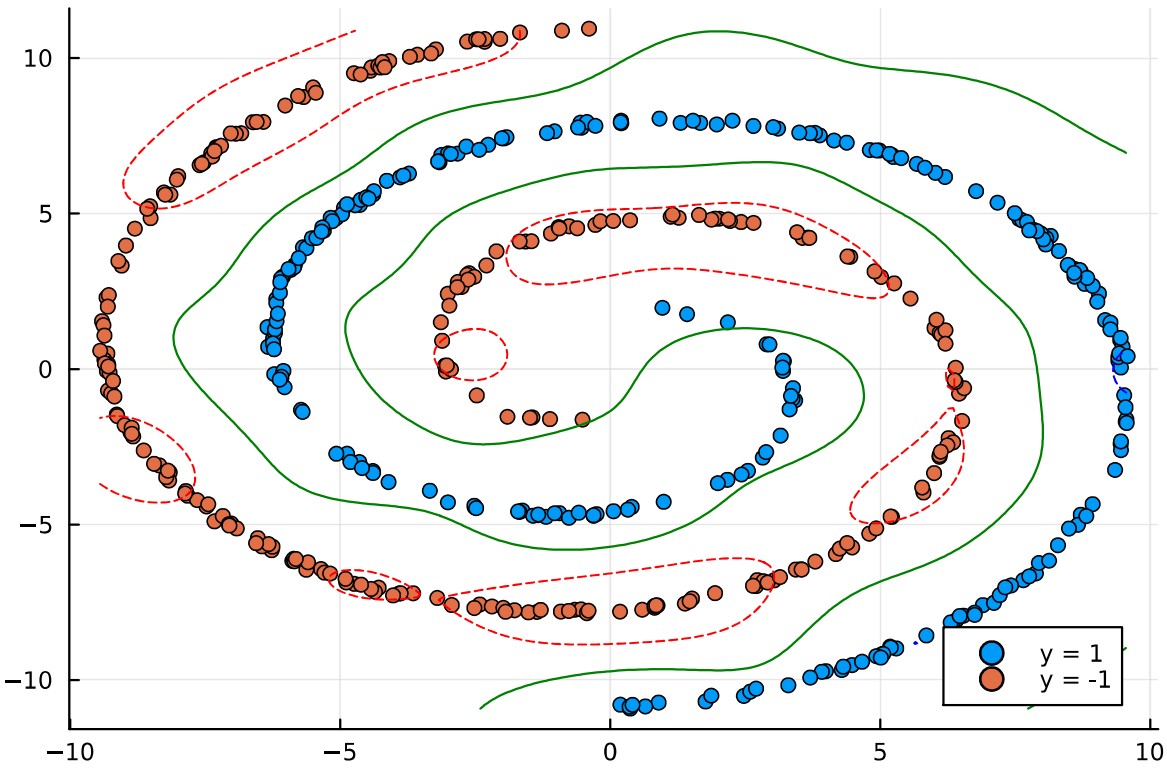
```
        if !(errors[1,j]==1)
            compute_error(j)
        end
        error_j = errors[j]
        r_j = error_j*y_j
        if ((r_j < -err_tol && alpha_j_old < C) || (r_j > err_tol && alpha_j_old > 0)
            nonzero_alphas = (1:n)[[x>0 && x< C for x in λ]]
            if length(nonzero_alphas) > 0
                max_error = 0
                i=j
                 for x in nonzero_alphas
                     if errors[1,x]==1
                         compute_error(x)
                         if abs(errors[2,x] - error_j) > max_error
                             i=x
                         end
                     end
                 end
                #println("Heurictic here!",i,j)
                 if update_alpha(i,j)==1
                    return 1
                 end

            end
            for i in shuffle(nonzero_alphas)
                #println("nonzero_alphas here!",i,j)
                if update_alpha(i,j)==1
                    return 1
                end
            end
            for i in randperm(n)
                #println("random here!",i,j)
                if update_alpha(i,j)==1
                    return 1
                end
            end
        end
        return 0
    end

    # 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))

    while (total_changed > 0 || do_all ==1)
        epoch = epoch+1
        total_changed = 0
        if (do_all ==1)
            for j in randperm(n)
                total_changed = total_changed  + process(j)
            end
        else
            nonzero_alphas = (1:n)[[x>0 && x< C for x in λ]]
            for j in shuffle(nonzero_alphas)
```

```
                total_changed = total_changed  + process(j)
            end
        end

        if (do_all == 1)
            do_all = 0
        elseif (total_changed == 0)
            do_all = 1
        end
        #println(total_changed,' ',b)
        if epoch == n_epochs
            break
        end
    end
    # Step 3: Training loop
    # END YOUR CODE
    ## Return hyperplane parameters
    return λ, b
end
```

```
Enter cell code...
```

```
([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.26385, 0.0,    more ,0.0], 0.328049)
```

```
# Uncomment this line below when you finish your implementation
λs, bs = 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
"""
```

```
# 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.