Table of Contents

Homework 5: Support Vector Networks
- Instructions for homework and submission
  - Instructions for doing homework
  - Instructions for submission
  - Content of the assignment
  - Others
- Problem statement
- Recall: Perceptron & Geometry Margin (Maximum 2.5 points)
  - Linear classifiers through origin
  - Perceptron Learning Algorithms
  - Convergence Proof
  - Geometric margin & SVM Motivation
- Linear Support Vector Machine (Maximum 6 points)
  - Hard-margin
  - Soft-margin
- Computing the SVM classifier (To get beyond 8.5 points)
  - SMO algorithm
  - Dual SVM - Hard-margin
  - Dual SVM - Soft-margin
- Multi-classes classification problem with SVMs (To get beyond 10.0 points)
  - Load MNIST dataset
  - Training SVMs
  - Evaluation
- References

Submission by: ***Hoàng Anh Trà*** (21127453@student.hcmus.edu.vn)

```
student = ▸(name = "Hoàng Anh Trà", id = "21127453")
```

```
# edit the code below to set your name and student identity number (i.e. the number without @student.hcmus.edu.vn)

student = (name = "Hoàng Anh Trà", id = "21127453")

# you might need to wait until all other cells in this notebook have completed running.
# scroll around the page to see what's up
```

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
```

```
TaskLocalRNG()
```

```
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:

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

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

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

Perceptron Learning Algorithms

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

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

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

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

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

Geometry intuition of Perceptron

8

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

```
points1train =
2×500 Matrix{Float64}:
 5.39966  5.5192   8.74341  5.55515  …  11.0884   13.5534    6.75036  1.64483
 7.50427  4.48183  6.10562  7.01194      7.21851   8.58991  -0.67835  0.602675
```

```
points2train =
2×500 Matrix{Float64}:
 -9.69835  -4.23896  -11.2302   2.12898  …  -0.0658955  -7.75077  -2.36411  2.18109
 10.6977    9.67346    7.57078  3.95975     11.8345      5.42738  13.1457   9.10793
```

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

```
points1test =
2×500 Matrix{Float64}:
 7.82635  11.9104  6.94708  7.00186  13.8683   …  3.68567  5.49657  2.94853  11.9291
 4.35232   1.6469  7.91428  3.27454   8.53738      3.29324  4.23278  6.83867   2.3494
```

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

```
points2test =
2×500 Matrix{Float64}:
  2.00189  -8.2967  -1.70747  -7.20897  …  -4.58538   3.15526  -7.05993  0.189603
 11.5287   11.3127   9.01887   8.18879      5.39254  12.9144   11.1034   8.91144
```

```
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

```julia
"""
    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
    n = size(pos_data, 2)
    data = vcat(hcat(pos_data, neg_data), ones(n * 2)')'
    labels = vcat(ones(n), -ones(n))
    θ = zeros(size(data, 2))
    for epoch in 1:n_epochs
        # Random permutation of indices
        indices = randperm(size(data, 1))

        for i in indices
            # Check if the data point is misclassified
            if sign(θ' * data[i, :]) != labels[i]
                # Update weights
                θ += η * labels[i] * data[i, :]
            end
        end
    end

    return θ
    # END YOUR CODE
    # return θ
end
```

```
θml = ▸[0.914923, -0.982939, 2.79]
```

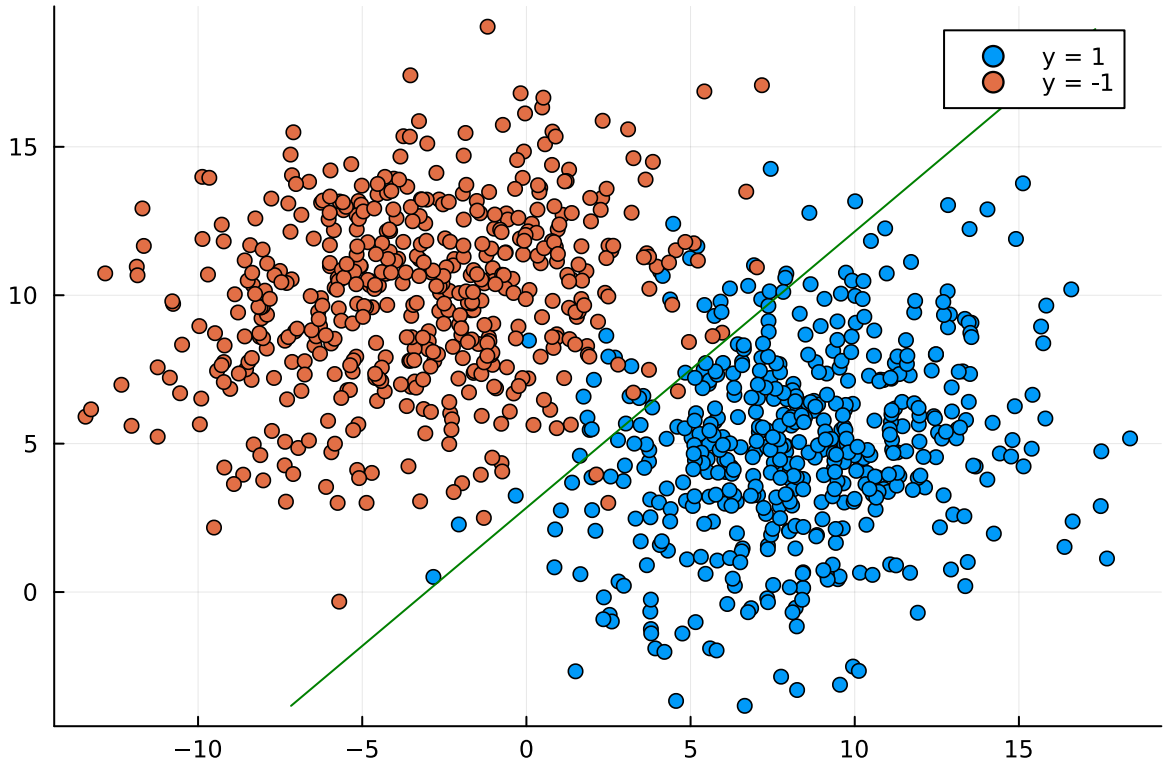
draw_pla

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

Fields

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

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

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

    b = θ[3]
    θml = θ[1:2]

    decision(x) = θml' * x + b

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

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

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




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

`tpfptnfn_cal`

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

Fields

- y_test: Actual labels
- y_pred: Predicted labels

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

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

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

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

eval_pla

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

Fields

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

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

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

    # START YOUR CODE
    # TODO: acc, p, r, f1???
    tp, fp, tn, fn = tpfptnfn_cal(y_test, y_pred)

    acc = (tp + tn) / length(y_test)
    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
```

▸ (0.941, 0.984615, 0.896, 0.93822)

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

```
acc: 0.941
precision: 0.9846153846153847
recall: 0.896
f1_score: 0.9382198952879582
```

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 $t = 1 \ldots n$.

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

Todo

Denote $\theta^{(k)}$ as the parameter vector after k updates, and θ^* as the optimal parameter vector. The PLA update rule is given by:

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

When an update is performed ($y_t(\theta^{(k+1)})^\top\mathbf{x}_t < 0$), the update rule is:

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

Now, let's calculate the inner product $(\theta^*)^\top\theta^{(k+1)}$:

$$\begin{aligned}(\theta^*)^\top\theta^{(k+1)} &= (\theta^*)^\top(\theta^{(k)} + y_t\mathbf{x}_t) \\ &= (\theta^*)^\top\theta^{(k)} + (\theta^*)^\top(y_t\mathbf{x}_t)\end{aligned}$$

Considering the update condition $y_t(\theta^{(k+1)})^\top\mathbf{x}_t < 0$ and $y_t(\theta^{(k)})^\top\mathbf{x}_t < 0$, we have $(\theta^*)^\top\mathbf{x}_t < 0$. Now, consider the term $(\theta^*)^\top(y_t\mathbf{x}_t)$:

$$(\theta^*)^\top(y_t\mathbf{x}_t) = y_t(\theta^*)^\top\mathbf{x}_t < 0$$

Therefore, we can rewrite the inner product as:

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

END YOUR PROOF

```
md"""
**START YOUR PROOF**

!!! todo

Denote $\theta^{(k)}$ as the parameter vector after $k$ updates, and $\theta^*$ as the optimal parameter vector. The PLA update rule is given by:

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

When an update is performed ($y_t(\theta^{(k+1)})^\top \mathbf{x}_t < 0$), the update rule is:

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

Now, let's calculate the inner product $(\theta^*)^\top \theta^{(k+1)}$:

$\begin{align*}

```

```
(\theta^*)^\top \theta^{(k+1)} &= (\theta^*)^\top (\theta^{(k)} + y_t \mathbf{x}_t)
\\
&= (\theta^*)^\top \theta^{(k)} + (\theta^*)^\top (y_t \mathbf{x}_t)
\end{align*}$

Considering the update condition $y_t(\theta^{(k+1)})^\top\mathbf{x}_t < 0$ and
$y_t(\theta^{(k)})^\top\mathbf{x}_t < 0$, we have $(\theta^*)^\top \mathbf{x}_t <
0$. Now, consider the term $(\theta^*)^\top (y_t \mathbf{x}_t)$:

$(\theta^*)^\top (y_t \mathbf{x}_t) = y_t (\theta^*)^\top \mathbf{x}_t < 0$

Therefore, we can rewrite the inner product as:

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

**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

Denote $\theta^{(k)}$ as the parameter vector after k updates in the Perceptron Learning Algorithm (PLA). The squared norm $||\theta^{(k)}||^2$ is given by:

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

Now, consider the PLA update rule:

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

When an update is performed ($y_t(\theta^{(k+1)})^\top\mathbf{x}_t < 0$), the update rule is:

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

Now, let's calculate the squared norm $||\theta^{(k+1)}||^2$:

$$\begin{aligned} ||\theta^{(k+1)}||^2 &= (\theta^{(k+1)})^\top \theta^{(k+1)} \\ &= (\theta^{(k)} + y_t\mathbf{x}_t)^\top(\theta^{(k)} + y_t\mathbf{x}_t) \\ &= (\theta^{(k)})^\top\theta^{(k)} + 2y_t(\theta^{(k)})^\top\mathbf{x}_t + y_t^2||\mathbf{x}_t||^2 \end{aligned}$$

Now, consider the update condition $y_t(\theta^{(k+1)})^\top\mathbf{x}_t < 0$ and $y_t(\theta^{(k)})^\top\mathbf{x}_t < 0$. We have $y_t(\theta^{(k)})^\top\mathbf{x}_t < 0$ since an update is performed. Therefore, $2y_t(\theta^{(k)})^\top\mathbf{x}_t < 0$.

Now, we can rewrite the squared norm $||\theta^{(k+1)}||^2$ as:

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

Therefore, we have shown that $||\theta^{(k+1)}||^2$ increases at most quadratically in the number of updates k.

END YOUR PROOF

```
md"""
**START YOUR PROOF**

Denote $\theta^{(k)}$ as the parameter vector after $k$ updates in the Perceptron Learning Algorithm (PLA). The squared norm $||\theta^{(k)}||^2$ is given by:

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

Now, consider the PLA update rule:

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

When an update is performed ($y_t(\theta^{(k+1)})^\top \mathbf{x}_t < 0$), the update rule is:
```

```

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

Now, let's calculate the squared norm $||\theta^{(k+1)}||^2$:

$\begin{align*}
||\theta^{(k+1)}||^2 &= (\theta^{(k+1)})^\top \theta^{(k+1)} \\
&= (\theta^{(k)} + y_t \mathbf{x}_t)^\top (\theta^{(k)} + y_t \mathbf{x}_t) \\
&= (\theta^{(k)})^\top \theta^{(k)} + 2y_t(\theta^{(k)})^\top \mathbf{x}_t + y_t^2 ||\mathbf{x}_t||^2
\end{align*}$

Now, consider the update condition $y_t(\theta^{(k+1)})^\top\mathbf{x}_t < 0$ and $y_t(\theta^{(k)})^\top\mathbf{x}_t < 0$. We have $y_t(\theta^{(k)})^\top\mathbf{x}_t < 0$ since an update is performed. Therefore, $2y_t(\theta^{(k)})^\top \mathbf{x}_t < 0$.

Now, we can rewrite the squared norm $||\theta^{(k+1)}||^2$ as:

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

Therefore, we have shown that $||\theta^{(k+1)}||^2$ increases at most quadratically in the number of updates $k$.

**END YOUR PROOF**
```

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

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

By combining the two we can show that the cosine of the angle between $\boldsymbol{\theta}^{(k)}$ and $\boldsymbol{\theta}^*$ 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

Let $\mathbf{x}_0$ be a point on the hyperplane defined by θ^*. The distance from a point $\mathbf{x}$ to the hyperplane is given by:

$$d(\mathbf{x}, \theta) = \frac{|\theta^\top \mathbf{x} + b|}{||\theta||}$$

Where θ is the normal vector to the hyperplane, and b is the bias term.

Now, consider two hyperplanes defined by weight vectors θ and θ^*. The distance between these two hyperplanes is the distance between a point on one hyperplane and the other hyperplane. Let $\mathbf{x}_0$ be a point on the hyperplane defined by θ^*. The distance between the two hyperplanes is then:

$$d = \frac{|\theta^\top \mathbf{x}_0 + b|}{||\theta||}$$

We aim to show that this distance is equal to $\frac{2}{||\theta||}$.

Now, consider the point $\mathbf{x}_1 = \mathbf{x}_0 + \frac{\theta}{||\theta||}$. This point lies on the hyperplane defined by θ. Calculate the distance between $\mathbf{x}_1$ and the hyperplane defined by θ^*:

$$d' = \frac{|{\theta^*}^\top \mathbf{x}_1 + b^*|}{||\theta^*||}$$

Since $\mathbf{x}_1$ lies on the hyperplane defined by θ, we have $\theta^\top \mathbf{x}_1 + b = 0$. Substitute this into the expression for d':

$$d' = \frac{|{\theta^*}^\top \mathbf{x}_1 + b^*|}{||\theta^*||} = \frac{|b^*|}{||\theta^*||}$$

Now, we can relate d and d':

$$d = \frac{|\theta^\top \mathbf{x}_0 + b|}{||\theta||} = \frac{|b|}{||\theta||}$$

To show that $d = \frac{2}{||\theta||}$, we need to show that $|b| = \frac{1}{2}$. This implies that $\mathbf{x}_0$ is a point on the hyperplane defined by $\frac{\theta}{2}$.

Therefore, the distance between the hyperplanes is indeed $\frac{2}{||\theta||}$.

END YOUR PROOF

```
md"""
**START YOUR PROOF**

Let $\mathbf{x}_0$ be a point on the hyperplane defined by $\mathbf{\theta}^*$. The distance from a point $\mathbf{x}$ to the hyperplane is given by:


```

```
$d(\mathbf{x}, \mathbf{\theta}) = \frac{|\mathbf{\theta}^\top \mathbf{x} + b|}
{||\mathbf{\theta}||}$

Where $\mathbf{\theta}$ is the normal vector to the hyperplane, and $b$ is the bias
term.

Now, consider two hyperplanes defined by weight vectors $\mathbf{\theta}$ and
$\mathbf{\theta}^*$. The distance between these two hyperplanes is the distance
between a point on one hyperplane and the other hyperplane. Let $\mathbf{x}_0$ be a
point on the hyperplane defined by $\mathbf{\theta}^*$. The distance between the
two hyperplanes is then:

$d = \frac{|\mathbf{\theta}^\top \mathbf{x}_0 + b|}{||\mathbf{\theta}||}$

We aim to show that this distance is equal to $\frac{2}{||\mathbf{\theta}||}$.

Now, consider the point $\mathbf{x}_1 = \mathbf{x}_0 + \frac{\mathbf{\theta}}
{||\mathbf{\theta}||}$. This point lies on the hyperplane defined by
$\mathbf{\theta}$. Calculate the distance between $\mathbf{x}_1$ and the hyperplane
defined by $\mathbf{\theta}^*$:

$d' = \frac{|\mathbf{\theta}^{*^\top} \mathbf{x}_1 + b^*|}{||\mathbf{\theta}^*||}$

Since $\mathbf{x}_1$ lies on the hyperplane defined by $\mathbf{\theta}$, we have
$\mathbf{\theta}^\top \mathbf{x}_1 + b = 0$. Substitute this into the expression
for $d'$:

$d' = \frac{|\mathbf{\theta}^{*^\top} \mathbf{x}_1 + b^*|}{||\mathbf{\theta}^*||} =
\frac{|b^*|}{||\mathbf{\theta}^*||}$

Now, we can relate $d$ and $d'$:

$d = \frac{|\mathbf{\theta}^\top \mathbf{x}_0 + b|}{||\mathbf{\theta}||} =
\frac{|b|}{||\mathbf{\theta}||}$

To show that $d = \frac{2}{||\mathbf{\theta}||}$, we need to show that $|b| =
\frac{1}{2}$. This implies that $\mathbf{x}_0$ is a point on the hyperplane defined
by $\frac{\mathbf{\theta}}{2}$.

Therefore, the distance between the hyperplanes is indeed $\frac{2}
{||\mathbf{\theta}||}$.


**END YOUR PROOF**
```

So we want to maximize the distance betweeen these two hyperplanes? Right? Equivalently, we minimize $||\boldsymbol{\theta}||$. 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 $\boldsymbol{\theta}$ and $\boldsymbol{b}$ 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

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

    # Combine positive and negative data, and add bias term
    n = size(pos_data, 2)
    data = vcat(hcat(pos_data, neg_data), ones(n * 2)')'
    labels = vcat(ones(n), -ones(n))

    # Initialize weights and bias
    w = θₘₗ

    # Gradient Descent
    for epoch in 1:n_epochs
        # Random permutation of indices
        indices = shuffle(1:length(labels))

        for i in indices
            x_i = data[i, :]
            y_i = labels[i]

            # Check if the current point violates the margin constraint
            if y_i * (dot(w, x_i)) < 1
                # Update weights using the gradient
                w += η * (y_i * x_i - w)

                # Add a regularization term
                w -= η * w / length(labels)
            end
        end
    end

    # Extract the weights and bias
    b = w[end]
    w = w[1:end-1]
```

```

    return w, b

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

```
([0.938794, -0.660465], 0.100564)
```

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

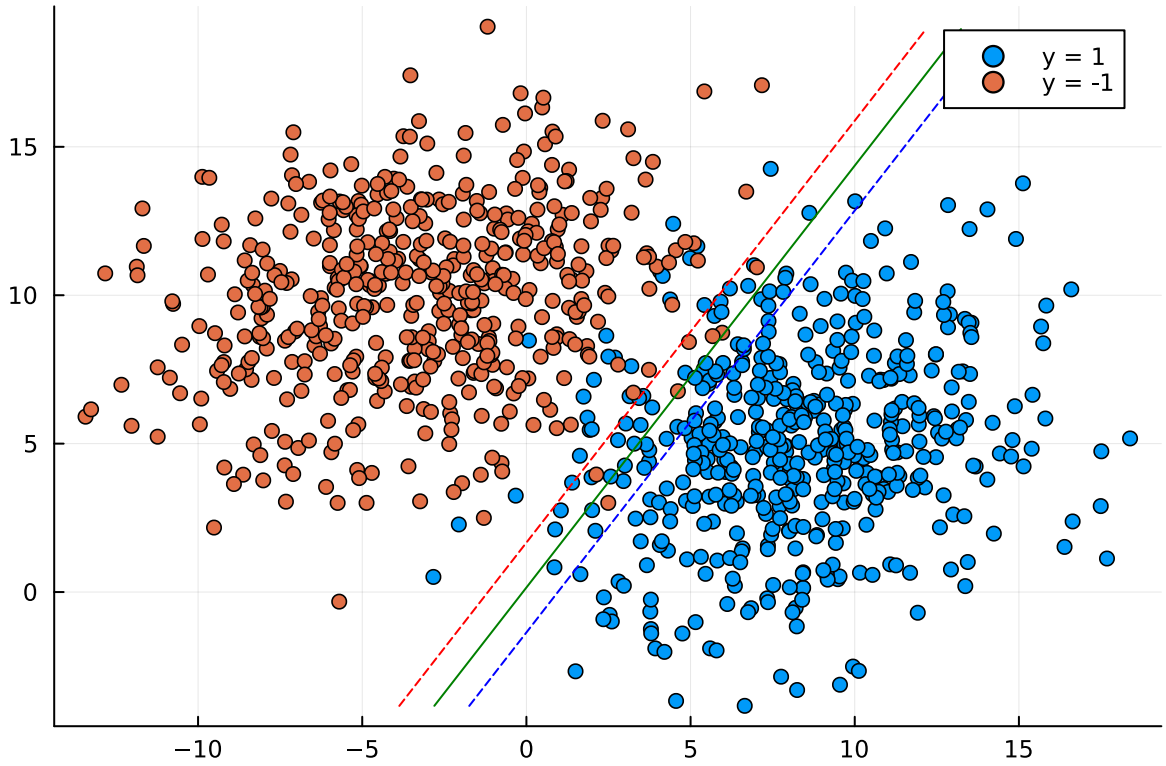
draw

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

Fields

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

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

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

    hyperplane(x)= w' * x + b

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

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

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




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

eval_svm

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

Fields

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

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

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

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

    # START YOUR CODE
    # TODO: acc, p, r, f1???
    tp, fp, tn, fn = tpfptnfn_cal(y_test, y_pred)

    acc = (tp + tn) / length(y_test)
    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
```

▶ (0.935, 0.995444, 0.874, 0.930777)

```julia
# Uncomment this line below when you finish your implementation
eval_svm(w, b, points1_test, points2_test)
```

```
acc: 0.935
precision: 0.9954441913439636
recall: 0.874
f1_score: 0.9307774227902025
```

Soft-margin

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

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

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

The regularized optimization problem become as

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

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

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

Todo

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

softmargin_svm

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

Fields

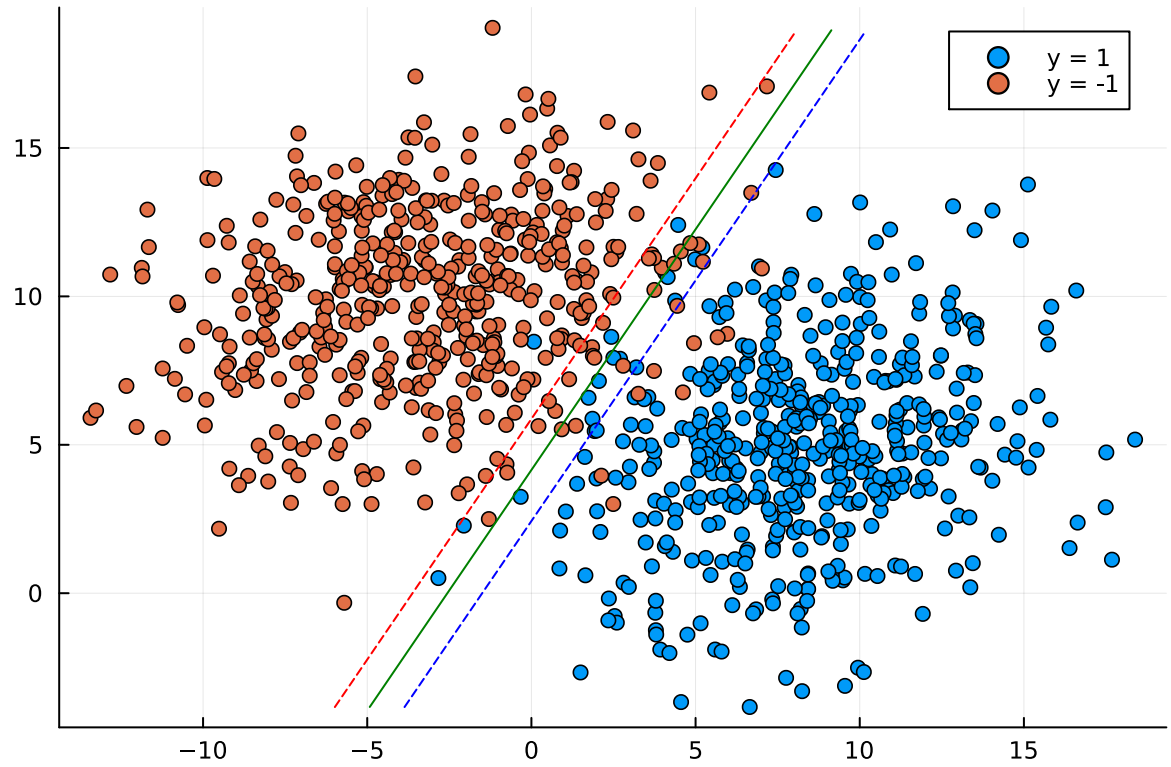
- pos_data::Matrix{Float64}: Input features for postive class (+1)
- neg_data::Matrix{Float64}: Input features for negative class (-1)
- C: relaxation variable control slack variables ς
- η::Float64=0.03: Learning rate. Default is 0.03
- n_epochs::Int64=10000: Maximum training epochs. Default is 10000

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

    # Combine positive and negative data, and add bias term
    n = size(pos_data, 2)
    data = vcat(hcat(pos_data, neg_data), ones(n * 2)')'
    labels = vcat(ones(n), -ones(n))

    # Initialize weights and bias
    w = θ_ml

    # Gradient Descent
    for epoch in 1:n_epochs
        # Random permutation of indices
        indices = shuffle(1:length(labels))

        for i in indices
            x_i = data[i, :]
            y_i = labels[i]

            # Calculate the hinge loss and slack variables
            loss = max(0, 1 - y_i * (dot(w, x_i)))

            # Calculate gradients
            grad = ifelse(loss > 0, -y_i * x_i, zeros(length(w)))

            # Update weights using the gradient and regularization
            w -= η * (grad + C * w / length(labels))
        end
    end

```

```
    # Extract the weights and bias
    b = w[end]
    w = w[1:end-1]

    return w, b
end
```

([0.942431, -0.581143], 2.41228)

```
# 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)
```

(0.966, 0.955078, 0.978, 0.966403)

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

```
acc: 0.966
precision: 0.955078125
recall: 0.978
f1_score: 0.966403162055336
```

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

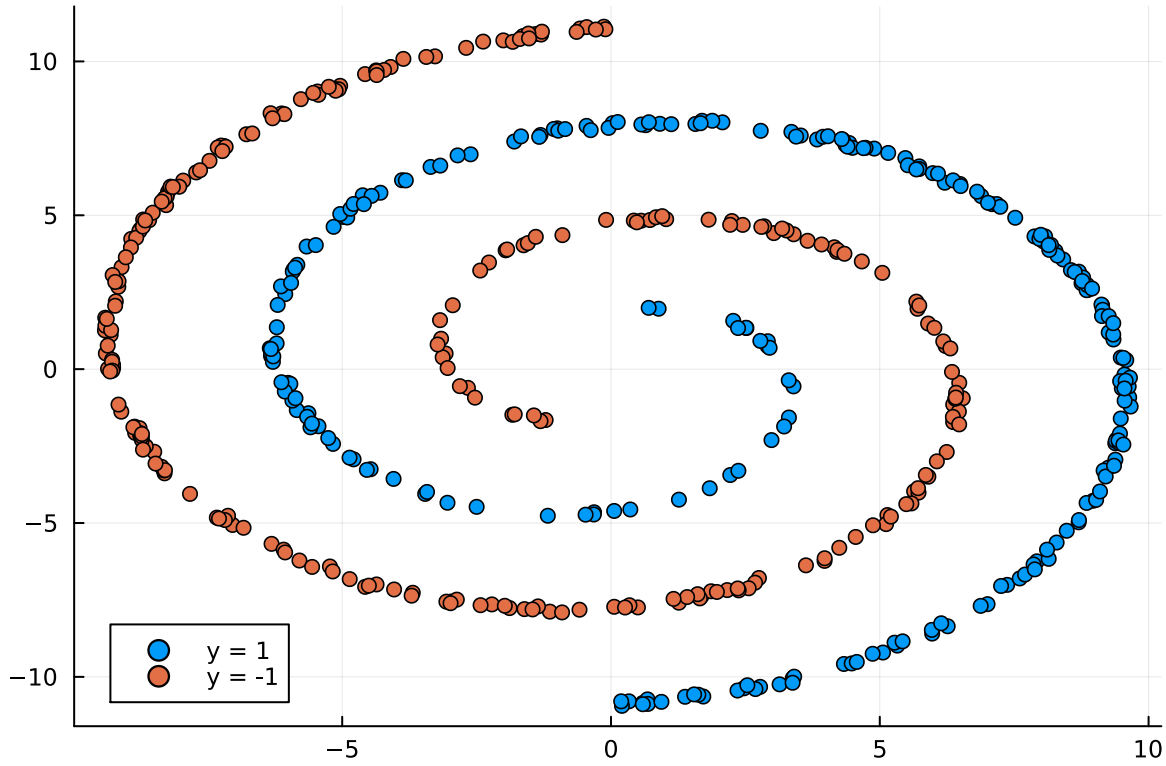
```
Function for creating two spirals dataset.

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
 -5.62797   4.33286  -6.32291   -4.9765      -3.04168   7.88372   9.53072  -6.20022
```

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




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

```
γ = 0.2
```

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

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

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

SMO algorithm

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

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

The SMO algorithm proceeds as follows:

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

Dual SVM - Hard-margin

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

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

where λ is Lagrange multiplier.

In terms of SVM optimization problem

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

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

Then the Lagrangian function is

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

Equivalently, Lagrangian primal problem is formulated as

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

Note

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

Danger

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

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

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

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

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

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

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

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

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

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

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

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

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

Todo

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

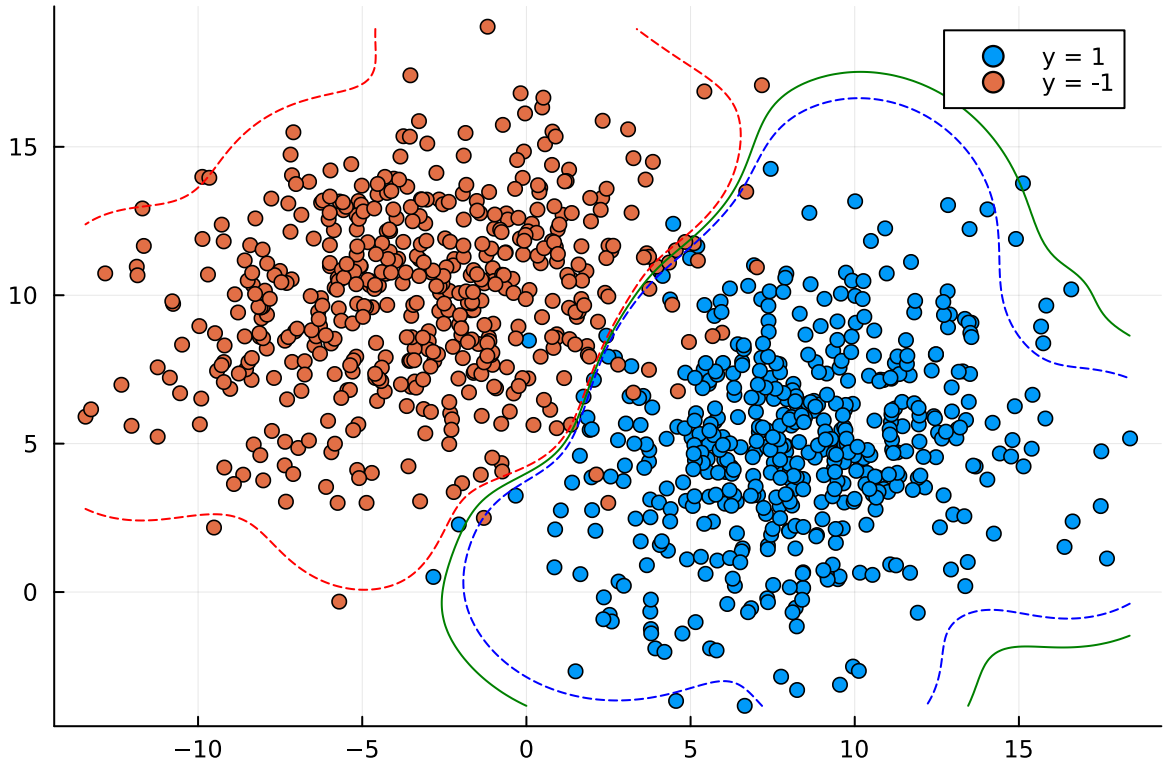
dualsvm_smo_hard (generic function with 4 methods)

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

    # START YOUR CODE
    # Step 1: Data preparation
    # First you construct and shuffle to obtain dataset D in a stochastically manner
    D = ([
      tuple.(eachcol(pos_data), 1)
      tuple.(eachcol(neg_data), -1)
    ])

    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: Main Loop (Sequential Minimal Optimization)
    for epoch in 1:n_epochs
        for i in 1:n
            Ei = sum(λ .* Y .* [K(X[j], X[i]) for j in 1:n]) + b - Y[i]

            # Check KKT conditions
            if (Y[i] * Ei < -errtol && λ[i] < λtol) || (Y[i] * Ei > errtol && λ[i] > 0)
                # Randomly select a second data point
                j = rand(1:n)
                while j == i
                    j = rand(1:n)
                end

                Ej = sum(λ .* Y .* [K(X[k], X[j]) for k in 1:n]) + b - Y[j]

                # Save the old values of α1 and α2
                old_λ1, old_λ2 = λ[i], λ[j]

                # Compute L and H, the bounds on new possible α values
                if Y[i] != Y[j]
                    L = max(0, λ[j] - λ[i])
                    H = min(n, n + λ[j] - λ[i])
                else
                    L = max(0, λ[i] + λ[j] - n)
                    H = min(n, λ[i] + λ[j])
                end

                if L == H
                    continue
                end

                # Compute η (the similarity between the data points)
```

```julia
                η = 2 * K(X[i], X[j]) - K(X[i], X[i]) - K(X[j], X[j])

                if η >= 0
                    continue
                end

                # Compute the new value for α2 (unclipped)
                λ[j] -= Y[j] * (Ei - Ej) / η

                # Clip α2 to be within the bounds [L, H]
                λ[j] = max(L, min(λ[j], H))

                # Check if α2 has changed significantly
                if abs(λ[j] - old_λ2) < 1e-5
                    continue
                end

                # Compute the new value for α1
                λ[i] += Y[i] * Y[j] * (old_λ2 - λ[j])

                # Compute the bias terms b1 and b2
                b1 = b - Ei - Y[i] * (λ[i] - old_λ1) * K(X[i], X[i]) - Y[j] * (λ[j] - old_λ2) * K(X[i], X[j])
                b2 = b - Ej - Y[i] * (λ[i] - old_λ1) * K(X[i], X[j]) - Y[j] * (λ[j] - old_λ2) * K(X[j], X[j])

                # Compute the new bias term b
                if 0 < λ[i] < n
                    b = b1
                elseif 0 < λ[j] < n
                    b = b2
                else
                    b = (b1 + b2) / 2
                end
            end
        end

    end


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

```
([0.0, 0.0, 0.0, 0.0, 0.0, 0.199722, 0.0, 0.0, 0.0, ⋯ more ,1.04083e-17], -0.309182)
```

```julia
# 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)
```

eval_dualsvm (generic function with 1 method)

```julia
function eval_dualsvm(pos_data, neg_data, λ, b)
    # Step 1: Data preparation
    D = ([
        tuple.(eachcol(pos_data), 1)
        tuple.(eachcol(neg_data), -1)
    ])

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

    # Step 2: Calculate predictions
    n = length(λ)
    predictions = zeros(length(Y))
    for i in 1:length(Y)
        predictions[i] = sign(
            sum(λ .* Y .* [K(X[j], X[i]) for j in 1:n]) + b
        )
    end

    # Step 3: Evaluate performance
    true_positives = sum((predictions .== 1) .& (Y .== 1))
    false_positives = sum((predictions .== 1) .& (Y .== -1))
    false_negatives = sum((predictions .== -1) .& (Y .== 1))

    accuracy = (true_positives + sum(predictions .== -1)) / length(Y)
    recall = true_positives / (true_positives + false_negatives)
    precision = true_positives / (true_positives + false_positives)
    f1_score = 2 * (precision * recall) / (precision + recall)

    print(" acc: $accuracy\n precision: $precision\n recall: $recall\n f1_score: $f1_score\n")
    return accuracy, recall, precision, f1_score
end

```

▶ (0.983, 0.986, 0.966667, 0.976238)

```julia
a, r, p, f = eval_dualsvm(points1train, points2train, λh, bh)
```

```
acc: 0.983
precision: 0.9666666666666667
recall: 0.986
f1_score: 0.9762376237623762
```

Dual SVM - Soft-margin

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

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

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

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

Todo

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

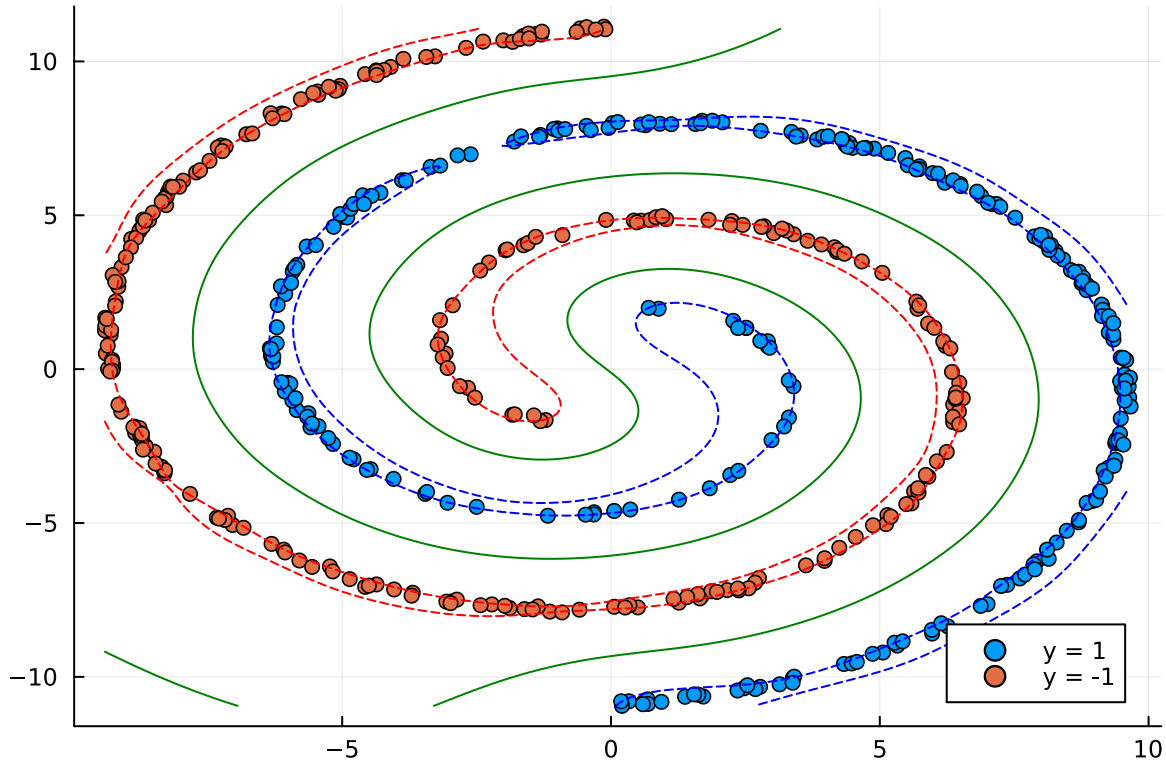
dualsvm_smo_soft (generic function with 5 methods)

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

    # Step 1: Data preparation
    # First you construct and shuffle to obtain dataset D in a stochastically manner
    D = ([
      tuple.(eachcol(pos_data), 1)
      tuple.(eachcol(neg_data), -1)
    ])
    # 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
        for i in 1:n
            Ei = sum(λ .* Y .* [K(X[j], X[i]) for j in 1:n]) + b - Y[i]

            # Check KKT conditions
            if (Y[i] * Ei < -errtol && λ[i] < C) || (Y[i] * Ei > errtol && λ[i] > 0)
                # Randomly select a second data point
                j = rand(1:n)
                while j == i
                    j = rand(1:n)
                end

                Ej = sum(λ .* Y .* [K(X[k], X[j]) for k in 1:n]) + b - Y[j]

                # Save the old values of α1 and α2
                old_λ1, old_λ2 = λ[i], λ[j]

                # Compute L and H, the bounds on new possible α values
                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

                # Compute η (the similarity between the data points)
                η = 2 * K(X[i], X[j]) - K(X[i], X[i]) - K(X[j], X[j])

                if η >= 0
                    continue
                end
```

```julia

                # Compute the new value for α2 (unclipped)
                λ[j] -= Y[j] * (Ei - Ej) / η

                # Clip α2 to be within the bounds [L, H]
                λ[j] = max(L, min(λ[j], H))

                # Check if α2 has changed significantly
                if abs(λ[j] - old_λ2) < 1e-5
                    continue
                end

                # Compute the new value for α1
                λ[i] += Y[i] * Y[j] * (old_λ2 - λ[j])

                # Compute the bias terms b1 and b2
                b1 = b - Ei - Y[i] * (λ[i] - old_λ1) * K(X[i], X[i]) - Y[j] * (λ[j] - old_λ2) * K(X[i], X[j])
                b2 = b - Ej - Y[i] * (λ[i] - old_λ1) * K(X[i], X[j]) - Y[j] * (λ[j] - old_λ2) * K(X[j], X[j])

                # Compute the new bias term b
                if 0 < λ[i] < C
                    b = b1
                elseif 0 < λ[j] < C
                    b = b2
                else
                    b = (b1 + b2) / 2
                end
            end
        end
    end

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

```
▶ ([0.0, 0.212901, 0.0695537, 0.0, 0.0, 0.0, 0.432387, 0.0, 0.302626, ⋯ more ,0.181228], 0
```

```julia
# 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)
```

```
(1.0, 1.0, 1.0, 1.0)
```

```
eval_dualsvm(sp_points1, sp_points2, λs, bs)
```

```
acc: 1.0
precision: 1.0
recall: 1.0
f1_score: 1.0
```

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

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
```

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

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

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

    train_x = train_x
end
```

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

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

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

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

    test_x = test_x
end
```

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

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

Training SVMs

```julia
using ScikitLearn
```

```
PyObject <class 'sklearn.linear_model._logistic.LogisticRegression'>
```

```
begin
    @sk_import metrics: accuracy_score
    @sk_import metrics: precision_score
    @sk_import metrics: recall_score
    @sk_import metrics: f1_score
    @sk_import linear_model: LogisticRegression
end
```

```
mkl not found, proceeding to installing non-mkl versions of sci-kit learn via C
onda
```

```
Running `conda install -y -c conda-forge 'scikit-learn>=1.2,<1.3'` in root envi
ronment
```

```
Collecting package metadata (current_repodata.json): ...working... done
Solving environment: ...working... done


==> WARNING: A newer version of conda exists. <==
  current version: 23.3.1
  latest version: 23.11.0

Please update conda by running

    $ conda update -n base -c conda-forge conda

Or to minimize the number of packages updated during conda update use

     conda install conda=23.11.0



# All requested packages already installed.
```

Comment

One-vs-Rest (OvR), also known as One-vs-All (OvA), is a strategy used in machine learning to extend binary classification algorithms to handle multiclass classification problems. In a multiclass classification problem, there are more than two classes, and the goal is to assign each instance to one of these multiple classes.

- In the case of the MNIST dataset, where you have multiple classes (digits 0 through 9), OvR involves training a separate binary classifier for each class. Each classifier is trained to distinguish between instances of its assigned class and all other classes combined. In essence, you create 10 binary classifiers (one for each digit) to solve the multiclass problem.
- The multi_class="ovr" setting in the logistic regression model indicates that you want to use the OvR strategy. This strategy simplifies the problem of training a multiclass classifier by reducing it to multiple binary classification subproblems.

`LogisticRegression(multi_class="ovr")` simplifies the implementation of a multiclass logistic regression model by using the OvR strategy, making it easier to use binary classification algorithms for multiclass problems.

```
[7, 2, 1, 0, 4, 1, 4, 9, 6, 9, 0, 6, 9, 0, 1, 5, 9, 7, 3, 4, ⋯ more ,7, 8, 9, 0, 1, 2, 3, 4
```

```julia
begin
    print(1)
    model = LogisticRegression(multi_class="ovr")
    print(2)
    # Reshape the training data
    train_x_reshaped = reshape(train_x, 28^2, NUMBER_TRAIN_SAMPLES)'
    # Train the model
    print(3)
    fit!(model, train_x_reshaped, train_y)

    # Reshape the test data
    test_x_reshaped = reshape(test_x, 28^2, NUMBER_TEST_SAMPLES)'
    # Make predictions on the test set
    predictions = predict(model, test_x_reshaped)
end
```

```
123/Users/mymac/.julia/conda/3/aarch64/lib/python3.10/site-packages/sklear
n/linear_model/_logistic.py:458: ConvergenceWarning: lbfgs failed to converge
(status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regress
ion
  n_iter_i = _check_optimize_result(
```

Evaluation

```
0.9178650729761928
```

```julia
begin
    # Evaluate the metrics
    accuracy = accuracy_score(test_y, predictions)
    precision = precision_score(test_y, predictions, average="weighted")
    recall = recall_score(test_y, predictions, average="weighted")
    f1 = f1_score(test_y, predictions, average="weighted")
end
```

```julia
begin
    println("Accuracy: ", accuracy)
    println("Precision: ", precision)
    println("Recall: ", recall)
    println("F1-score: ", f1)
end
```

```
Accuracy: 0.918
Precision: 0.9182007546277949
Recall: 0.918
F1-score: 0.9178650729761928
```

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.