Optimization (Wrap-up), and Hyperplane based Classifiers (Perceptron and Support Vector Machines)

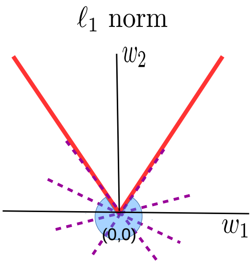
Piyush Rai

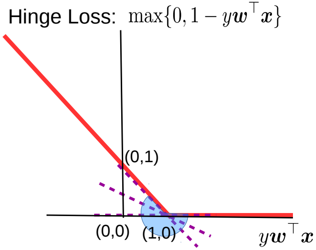
Introduction to Machine Learning (CS771A)

August 30, 2018

Recap: Subgradient Descent

- Use subgradient at non-differentiable points, use gradient elsewhere





- Left: each entry of (sub)gradient vector for $||\boldsymbol{w}||_1$, Right: (sub)gradient vector for hinge loss

$$t_d = \begin{cases} -1, & \text{for } w_d < 0 \\ [-1, +1] & \text{for } w_d = 0 \\ +1 & \text{for } w_d > 0 \end{cases} \qquad \boldsymbol{t} = \begin{cases} 0, & \text{for } y_n \boldsymbol{w}^\top \boldsymbol{x}_n > 1 \\ -y_n \boldsymbol{x}_n & \text{for } y_n \boldsymbol{w}^\top \boldsymbol{x}_n < 1 \\ k y_n \boldsymbol{x}_n & \text{for } y_n \boldsymbol{w}^\top \boldsymbol{x}_n = 1 \quad (\text{where } k \in [-1, 0]) \end{cases}$$

Recap: Constrained Optimization via Lagrangian

$$\boxed{\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} f(\boldsymbol{w}), \quad \text{s.t.} \quad g(\boldsymbol{w}) \leq 0}$$

$$c(\boldsymbol{w}) = \max_{\alpha \geq 0} \ \alpha g(\boldsymbol{w}) = \begin{cases} \infty, & \text{if } g(\boldsymbol{w}) > 0 \quad \text{(constraint violated)} \\ 0 & \text{if } g(\boldsymbol{w}) \leq 0 \quad \text{(constraint satisfied)} \end{cases}$$

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} f(\boldsymbol{w}) + c(\boldsymbol{w})$$

Same as $f(w)$ when constraint satisfied

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \left\{ f(\boldsymbol{w}) + \max_{\alpha \geq 0} \ \alpha g(\boldsymbol{w}) \right\}$$

Lagrangian: $\boxed{\mathcal{L}(\boldsymbol{w}, \alpha) = f(\boldsymbol{w}) + \alpha g(\boldsymbol{w})}$

Recap: Constrained Optimization via Lagrangian

- We minimize the Lagrangian $\mathcal{L}(\boldsymbol{w}, \alpha)$ w.r.t. $\boldsymbol{w}$ and maximize w.r.t. α

$$\boxed{\mathcal{L}(\boldsymbol{w}, \alpha) = f(\boldsymbol{w}) + \alpha g(\boldsymbol{w})}$$

- For certain problems, the order of maximization and minimization does not matter
- Approach 1: Can first maximize w.r.t. α and then minimize w.r.t. $\boldsymbol{w}$

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \left\{ \max_{\alpha} \mathcal{L}(\boldsymbol{w}, \alpha) \right\}$$

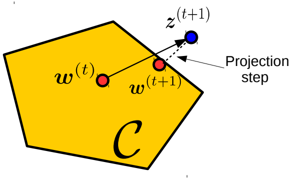
- Approach 2: Can first minimize w.r.t. $\boldsymbol{w}$ and then maximize w.r.t. α

$$\hat{\alpha} = \arg\max_{\alpha} \left\{ \min_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}, \alpha) \right\}$$

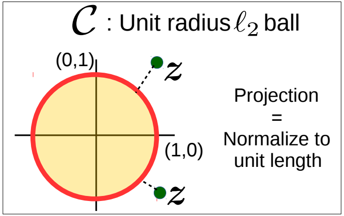
- Approach 2 is known as optimizing via the dual (popular in SVM solvers; will see today!)
- KKT condition: At the optimal solution $\hat{\alpha} g(\hat{\boldsymbol{w}}) = 0$
- Multiple constraints (inequality/equality) can also be handled likewise

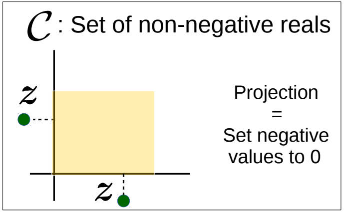
Recap: Projected Gradient Descent

- Same as GD + extra projection step we step out of the constraint set



- In some cases, the projection step is very easy





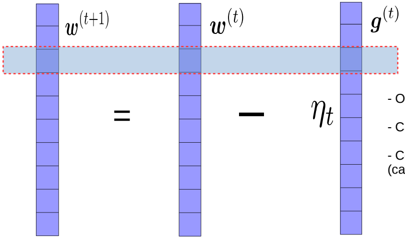
Co-ordinate Descent (CD)

- Standard GD update for $\boldsymbol{w} \in \mathbb{R}^D$ at each step

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta_t \boldsymbol{g}^{(t)}$$

- CD: Each step updates one component (co-ordinate) at a time, keeping all others fixed

$$w_d^{(t+1)} = w_d^{(t)} - \eta_t g_d^{(t)}$$



- Order: Random/cyclic

- Can also update in "blocks"

- Cost of update independent of *D* (caching previous computations helps)

Alternating Optimization

- Consider an optimization problems with several variables, say 2 variables $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$

$$\{\hat{\boldsymbol{w}}_1, \hat{\boldsymbol{w}}_2\} = \arg\min_{\boldsymbol{w}_1, \boldsymbol{w}_2} \mathcal{L}(\boldsymbol{w}_1, \boldsymbol{w}_2)$$

- Often, this "joint" optimization is hard/impossible. We can consider an alternating scheme

ALT-OPT

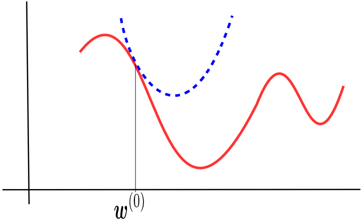
1. Initialize one of the variables, e.g., $\boldsymbol{w}_2 = \boldsymbol{w}_2^{(0)}, t = 0$
2. Solve $\boldsymbol{w}_1^{(t+1)} = \arg\min_{\boldsymbol{w}_1} \mathcal{L}(\boldsymbol{w}_1, \boldsymbol{w}_2^{(t)})$ $\quad$ // $\boldsymbol{w}_2$ "fixed" at its most recent value $\boldsymbol{w}_2^{(t)}$
3. Solve $\boldsymbol{w}_2^{(t+1)} = \arg\min_{\boldsymbol{w}_2} \mathcal{L}(\boldsymbol{w}_1^{(t+1)}, \boldsymbol{w}_2)$ $\quad$ // $\boldsymbol{w}_1$ "fixed" at its most recent value $\boldsymbol{w}_1^{(t+1)}$
4. $t = t + 1$. Go to step 2 if not converged yet.

- Usually converges to a local optima of $\mathcal{L}(\boldsymbol{w}_1, \boldsymbol{w}_2)$. Also connections to EM (will see later)
- VERY VERY useful!!! Also extends to more than 2 variables. CD is somewhat like ALT-OPT.

Newton's Method

- Newton's method uses second-order information (second derivative a.k.a. Hessian)
- At each point $\boldsymbol{w}^{(t)}$, minimize the quadratic (second-order) approximation of the function

$$\boldsymbol{w}^{(t+1)} = \arg\min_{\boldsymbol{w}} \left\{ f(\boldsymbol{w}^{(t)}) + \nabla f(\boldsymbol{w}^{(t)})^\top (\boldsymbol{w} - \boldsymbol{w}^{(t)}) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^{(t)})^\top \nabla^2 f(\boldsymbol{w}^{(t)})(\boldsymbol{w} - \boldsymbol{w}^{(t)}) \right\}$$

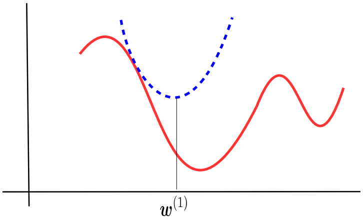


- Exercise: Verify that $\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - (\nabla^2 f(\boldsymbol{w}^{(t)}))^{-1} \nabla f(\boldsymbol{w}^{(t)})$. Also no learning rate needed!
- Converges much faster than GD. But also expensive due to Hessian computation/inversion.
- Many ways to approximate the Hessian (e.g., using previous gradients); also look at L-BFGS etc.

Newton's Method

- Newton's method uses second-order information (second derivative a.k.a. Hessian)
- At each point $\boldsymbol{w}^{(t)}$, minimize the quadratic (second-order) approximation of the function

$$\boldsymbol{w}^{(t+1)} = \arg\min_{\boldsymbol{w}} \left\{ f(\boldsymbol{w}^{(t)}) + \nabla f(\boldsymbol{w}^{(t)})^\top (\boldsymbol{w} - \boldsymbol{w}^{(t)}) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^{(t)})^\top \nabla^2 f(\boldsymbol{w}^{(t)})(\boldsymbol{w} - \boldsymbol{w}^{(t)}) \right\}$$

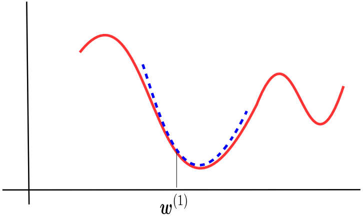


- Exercise: Verify that $\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - (\nabla^2 f(\boldsymbol{w}^{(t)}))^{-1} \nabla f(\boldsymbol{w}^{(t)})$. Also no learning rate needed!
- Converges much faster than GD. But also expensive due to Hessian computation/inversion.
- Many ways to approximate the Hessian (e.g., using previous gradients); also look at L-BFGS etc.

Newton's Method

- Newton's method uses second-order information (second derivative a.k.a. Hessian)
- At each point $\boldsymbol{w}^{(t)}$, minimize the quadratic (second-order) approximation of the function

$$\boldsymbol{w}^{(t+1)} = \arg\min_{\boldsymbol{w}} \left\{ f(\boldsymbol{w}^{(t)}) + \nabla f(\boldsymbol{w}^{(t)})^\top (\boldsymbol{w} - \boldsymbol{w}^{(t)}) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^{(t)})^\top \nabla^2 f(\boldsymbol{w}^{(t)})(\boldsymbol{w} - \boldsymbol{w}^{(t)}) \right\}$$

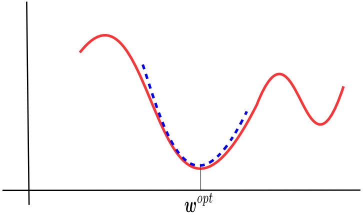


- Exercise: Verify that $\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - (\nabla^2 f(\boldsymbol{w}^{(t)}))^{-1} \nabla f(\boldsymbol{w}^{(t)})$. Also no learning rate needed!
- Converges much faster than GD. But also expensive due to Hessian computation/inversion.
- Many ways to approximate the Hessian (e.g., using previous gradients); also look at L-BFGS etc.

Newton's Method

- Newton's method uses second-order information (second derivative a.k.a. Hessian)
- At each point $\boldsymbol{w}^{(t)}$, minimize the quadratic (second-order) approximation of the function

$$\boldsymbol{w}^{(t+1)} = \arg\min_{\boldsymbol{w}} \left\{ f(\boldsymbol{w}^{(t)}) + \nabla f(\boldsymbol{w}^{(t)})^\top (\boldsymbol{w} - \boldsymbol{w}^{(t)}) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^{(t)})^\top \nabla^2 f(\boldsymbol{w}^{(t)})(\boldsymbol{w} - \boldsymbol{w}^{(t)}) \right\}$$

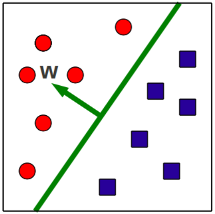


- Exercise: Verify that $\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - (\nabla^2 f(\boldsymbol{w}^{(t)}))^{-1} \nabla f(\boldsymbol{w}^{(t)})$. Also no learning rate needed!
- Converges much faster than GD. But also expensive due to Hessian computation/inversion.
- Many ways to approximate the Hessian (e.g., using previous gradients); also look at L-BFGS etc.

Summary

- Gradient methods are simple to understand and implement
- More sophisticated optimization methods often use gradient methods
  - Backpropagation algorithm used in deep neural nets is GD + chain rule of differentiation
- Use subgradient methods if function not differentiable
- Constrained optimization require methods such as Lagrangian or projected gradient
- Second order methods such as Newton's method are much faster but computationally expensive
- But computing all this gradient related stuff looks scary to me. Any help?
  - Don't worry. Automatic Differentiation (AD) methods available now
  - AD only requires specifying the loss function (useful for complex models like deep neural nets)
  - Many packages such as Tensorflow, PyTorch, etc. provide AD support
  - But having a good understanding of optimization is still helpful

Hyperplane based Classification



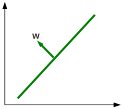
All linear models for classification are basically about learning hyperplanes!

Already saw logistic regression (probabilistic linear classifier).

Will look at some more today - Perceptron, SVM
(also how some of the optimization methods we saw can be applied in these cases)

Hyperplanes

- Separates a D-dimensional space into two **half-spaces** (positive and negative)

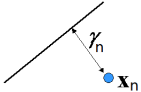


- Defined by normal vector $\boldsymbol{w} \in \mathbb{R}^D$ (pointing towards positive half-space)
- Equation of the hyperplane: $\boldsymbol{w}^\top \boldsymbol{x} = 0$
- Assumption: The hyperplane passes through origin. If not, add a bias term $b \in \mathbb{R}$

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

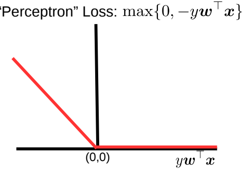
- $b > 0$ means moving it parallely in the direction of $\boldsymbol{w}$ ($b < 0$ means moving in opposite direction)
- Distance of a point $\boldsymbol{x}_n$ from a hyperplane (can be +ve/-ve)

$$\gamma_n = \frac{\boldsymbol{w}^T \boldsymbol{x}_n + b}{||\boldsymbol{w}||}$$



A Mistake-Driven Method for Learning Hyperplanes

- Let's ignore the bias term b for now. So the hyperplane is simply $\boldsymbol{w}^\top \boldsymbol{x} = 0$
- Consider SGD to learn a hyperplane based model with loss: $\mathcal{L}(\boldsymbol{w}) = \sum_{n=1}^{N} \max\{0, -y_n \boldsymbol{w}^\top \boldsymbol{x}_n\}$



- Loss not differentiable at $y_n \boldsymbol{w}^\top \boldsymbol{x}_n = 0$, so we will use subgradients there. The (sub)gradient will be

$$\boldsymbol{g}_n = \begin{cases} 0, & \text{for } y_n \boldsymbol{w}^\top \boldsymbol{x}_n > 0 \\ -y_n \boldsymbol{x}_n & \text{for } y_n \boldsymbol{w}^\top \boldsymbol{x}_n < 0 \\ k y_n \boldsymbol{x}_n & \text{for } y_n \boldsymbol{w}^\top \boldsymbol{x}_n = 0 \quad (\text{where } k \in [-1, 0]) \end{cases}$$

- If we use $k = 0$ then $\boldsymbol{g}_n = 0$ for $y_n \boldsymbol{w}^\top \boldsymbol{x}_n \geq 0$, and $\boldsymbol{g}_n = -y_n \boldsymbol{x}_n$ if $y_n \boldsymbol{w}^\top \boldsymbol{x}_n < 0$
- Thus $\boldsymbol{g}_n$ nonzero only when $y_n \boldsymbol{w}^\top \boldsymbol{x}_n < 0$ (mistake). SGD will update $\boldsymbol{w}$ only in these cases!

Mistake-Driven Learning of Hyperplanes

- The complete SGD algorithm for a model with this loss function will be

Stochastic SubGD

1. Initialize $\boldsymbol{w} = \boldsymbol{w}^{(0)}, t = 0$, set $\eta_t = 1, \forall t$
2. Pick some $(\boldsymbol{x}_n, y_n)$ randomly.
3. If current $\boldsymbol{w}$ makes a mistake on $(\boldsymbol{x}_n, y_n)$, i.e., $y_n \boldsymbol{w}^{(t)^\top} \boldsymbol{x}_n < 0$

$$
\begin{aligned}
\boldsymbol{w}^{(t+1)} &= \boldsymbol{w}^{(t)} + y_n \boldsymbol{x}_n \\
t &= t + 1
\end{aligned}
$$

4. If not converged, go to step 2.

- This is the Perceptron algorithm. An example of an online learning algorithm
- Note: Assuming $\boldsymbol{w}^{(0)} = 0$, easy to see the final $\boldsymbol{w}$ has the form $\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n y_n \boldsymbol{x}_n$
  - .. where α_n is total number of mistakes made by the algorithm on example $(\boldsymbol{x}_n, y_n)$
  - As we'll see, many other models will also lead to $\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n y_n \boldsymbol{x}_n$ (for some suitable α_n's)

Perceptron: Corrective Updates and Convergence

- Suppose true $y_n = +1$ (positive example) and the model mispredicts, i.e., ${\boldsymbol{w}^{(t)}}^\top \boldsymbol{x}_n < 0$
- After the update $\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} + y_n \boldsymbol{x}_n = \boldsymbol{w}^{(t)} + \boldsymbol{x}_n$

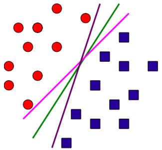
$${\boldsymbol{w}^{(t+1)}}^\top \boldsymbol{x}_n = {\boldsymbol{w}^{(t)}}^\top \boldsymbol{x}_n + \boldsymbol{x}_n^\top \boldsymbol{x}_n$$

.. which is less negative than ${\boldsymbol{w}^{(t)}}^\top \boldsymbol{x}_n$ (so the model has improved)

- Exercise: Verify that the model also improves after updating on a mistake on negative example
- Note: If training data is linearly separable, Perceptron converges in finite iterations
  - Proof: Block & Novikoff theorem (will provide the proof in a separate note)
  - What this means: It will eventually classify every training example correctly
  - Speed of convergence depends on the margin of separation (and on nothing else, such as N, D)
  - Note: In practice, we might want to stop sooner (to avoid overfitting)

Perceptron and (Lack of) Margins

- Perceptron learns a hyperplane (of many possible) that separates the classes



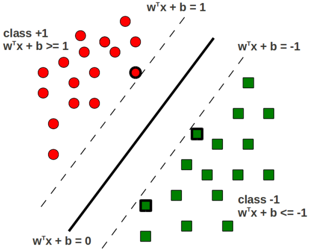
- The one learned will depend on the initial $\boldsymbol{w}$
- Standard Perceptron doesn't guarantee any "margin" around the hyperplane
- Note: Possible to "artificially" introduce a margin in the Perceptron
  - Simply change the Perceptron mistake condition to

$$y_n \boldsymbol{w}^T \boldsymbol{x}_n < \gamma$$

    where $\gamma > 0$ is a pre-specified margin. For standard Perceptron, $\gamma = 0$
  - **Support Vector Machine (SVM)** does this directly by learning the maximum margin hyperplane

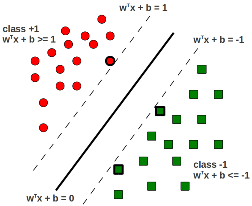
Support Vector Machine (SVM)

- SVM is a hyperplane based (linear) classifier that ensures a large margin around the hyperplane
- Note: We will assume the hyperplane to be of the form $\boldsymbol{w}^\top \boldsymbol{x} + b = 0$ (will keep the bias term b)



- Note: SVMs can also learn nonlinear decision boundaries using **kernel methods** (will see later)
- Reason behind the name "Support Vector Machine"?
  - SVM optimization discovers the most important examples (called "support vectors") in training data
  - These examples act as "balancing" the margin boundaries (hence called "support")

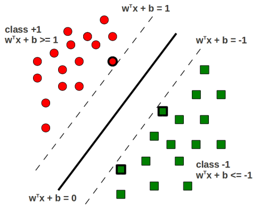
Learning a Maximum Margin Hyperplane



- *Suppose* we want a hyperplane $\boldsymbol{w}^\top \boldsymbol{x} + b = 0$ such that
  - $\boldsymbol{w}^T \boldsymbol{x}_n + b \geq 1$ for $y_n = +1$
  - $\boldsymbol{w}^T \boldsymbol{x}_n + b \leq -1$ for $y_n = -1$
  - Equivalently, $y_n(\boldsymbol{w}^T \boldsymbol{x}_n + b) \geq 1 \quad \forall n$
  - Define the margin on each side: $\gamma = \min_{1 \leq n \leq N} \frac{|\boldsymbol{w}^T \boldsymbol{x}_n + b|}{||\boldsymbol{w}||} = \frac{1}{||\boldsymbol{w}||}$
  - Total margin $= 2\gamma = \frac{2}{||\boldsymbol{w}||}$
- Want the hyperplane $(\boldsymbol{w}, b)$ that gives the largest possible margin
- **Note:** Can replace $y_n(\boldsymbol{w}^T \boldsymbol{x}_n + b) \geq 1$ by $y_n(\boldsymbol{w}^T \boldsymbol{x}_n + b) \geq m$ for some $m > 0$. It won't change the solution for $\boldsymbol{w}$, will just scale it by a constant, without changing the direction of $\boldsymbol{w}$ (exercise)

Hard-Margin SVM

- Hard-Margin: Every training example has to fulfil the margin condition $y_n(\boldsymbol{w}^T \boldsymbol{x}_n + b) \geq 1$



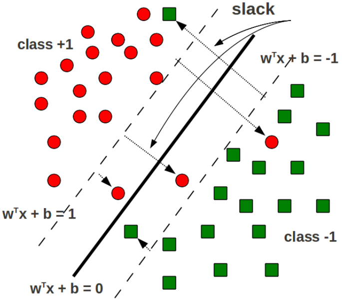
- Also want to maximize the margin $\gamma \propto \frac{1}{||\boldsymbol{w}||}$. Equivalent to minimizing $||\boldsymbol{w}||^2$ or $\frac{||\boldsymbol{w}||^2}{2}$
- The objective for hard-margin SVM

$$\boxed{\begin{aligned} &\min_{\boldsymbol{w},b} \quad f(\boldsymbol{w}, b) = \frac{||\boldsymbol{w}||^2}{2} \\ &\text{subject to} \quad y_n(\boldsymbol{w}^T \boldsymbol{x}_n + b) \geq 1, \qquad n = 1, \ldots, N \end{aligned}}$$

- Constrained optimization with N inequality constraints (note: function and constraints are convex)

Soft-Margin SVM (More Commonly Used)

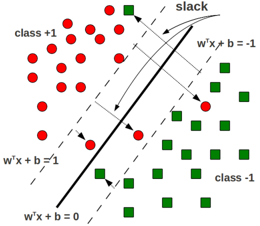
- Allow some training examples to fall **within** the margin region, or be even misclassified (i.e., fall on the wrong side). Preferable if training data is noisy



- Each training example $(\boldsymbol{x}_n, y_n)$ given a "slack" $\xi_n \geq 0$ (distance by which it "violates" the margin). If $\xi_n > 1$ then $\boldsymbol{x}_n$ is totally on the wrong side
  - Basically, we want a soft-margin condition: $y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b) \geq 1 - \xi_n, \quad \xi_n \geq 0$

Soft-Margin SVM (More Commonly Used)

- Goal: Maximize the margin, while also minimizing the sum of slacks (don't want too many training examples violating the margin condition)



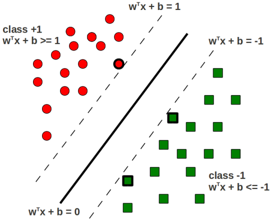
- The primal objective for soft-margin SVM can thus be written as

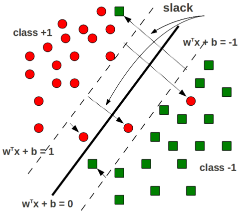
$$\min_{\boldsymbol{w},b,\boldsymbol{\xi}} \quad f(\boldsymbol{w},b,\boldsymbol{\xi}) = \frac{||\boldsymbol{w}||^2}{2} + C\sum_{n=1}^{N}\xi_n$$

$$\text{subject to} \quad y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b) \geq 1 - \xi_n, \quad \xi_n \geq 0 \qquad n = 1,\ldots,N$$

- Constrained optimization with $2N$ inequality constraints
- Parameter C controls the trade-off between large margin vs small training error

Summary: Hard-Margin SVM vs Soft-Margin SVM





- Objective for the hard-margin SVM (unknowns are $\boldsymbol{w}$ and b)

$$\boxed{\begin{aligned} &\min_{\boldsymbol{w},b} \quad f(\boldsymbol{w},b) = \frac{||\boldsymbol{w}||^2}{2} \\ &\text{subject to} \quad y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b) \geq 1, \qquad n = 1,\ldots,N \end{aligned}}$$

- Objective for the soft-margin SVM (unknowns are $\boldsymbol{w}$, b, and $\{\xi_n\}_{n=1}^N$)

$$\boxed{\begin{aligned} &\min_{\boldsymbol{w},b,\boldsymbol{\xi}} \quad f(\boldsymbol{w},b,\boldsymbol{\xi}) = \frac{||\boldsymbol{w}||^2}{2} + C\sum_{n=1}^{N}\xi_n \\ &\text{subject to} \quad y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b) \geq 1 - \xi_n, \quad \xi_n \geq 0 \qquad n = 1,\ldots,N \end{aligned}}$$

- In either case, we have to solve a constrained, convex optimization problem

Solving Hard-Margin SVM

Solving Hard-Margin SVM

- The hard-margin SVM optimization problem is:

$$\boxed{\begin{aligned} &\min_{\boldsymbol{w},b} \quad f(\boldsymbol{w},b) = \frac{||\boldsymbol{w}||^2}{2} \\ &\text{subject to} \quad 1 - y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b) \leq 0, \qquad n = 1,\ldots,N \end{aligned}}$$

- A constrained optimization problem. Can solve using Lagrange's method
- Introduce Lagrange Multipliers α_n $(n = \{1,\ldots,N\})$, one for each constraint, and solve

$$\boxed{\min_{\boldsymbol{w},b} \max_{\boldsymbol{\alpha}\geq 0} \mathcal{L}(\boldsymbol{w},b,\boldsymbol{\alpha}) = \frac{||\boldsymbol{w}||^2}{2} + \sum_{n=1}^{N} \alpha_n\{1 - y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b)\}}$$

- Note: $\boldsymbol{\alpha} = [\alpha_1,\ldots,\alpha_N]$ is the vector of Lagrange multipliers
- Note: It is easier (and helpful; we will soon see why) to solve the dual problem: min and then max

Solving Hard-Margin SVM

- The dual problem (min then max) is

$$\boxed{\max_{\boldsymbol{\alpha} \geq 0} \min_{\boldsymbol{w}, b} \mathcal{L}(\boldsymbol{w}, b, \boldsymbol{\alpha}) = \frac{\boldsymbol{w}^\top \boldsymbol{w}}{2} + \sum_{n=1}^{N} \alpha_n \{1 - y_n(\boldsymbol{w}^T \boldsymbol{x}_n + b)\}}$$

- Take (partial) derivatives of $\mathcal{L}$ w.r.t. $\boldsymbol{w}$, b and set them to zero

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = 0 \Rightarrow \boxed{\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n y_n \boldsymbol{x}_n} \qquad \frac{\partial \mathcal{L}}{\partial b} = 0 \Rightarrow \sum_{n=1}^{N} \alpha_n y_n = 0$$

- Important: Note the form of the solution $\boldsymbol{w}$ - it is simply a weighted sum of all the training inputs $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$ (and α_n is like the "importance" of $\boldsymbol{x}_n$)
- Substituting $\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n y_n \boldsymbol{x}_n$ in Lagrangian, we get the dual problem as (verify)

$$\boxed{\max_{\boldsymbol{\alpha} \geq 0} \mathcal{L}_D(\boldsymbol{\alpha}) = \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{m,n=1}^{N} \alpha_m \alpha_n y_m y_n (\boldsymbol{x}_m^T \boldsymbol{x}_n)}$$

Solving Hard-Margin SVM

- Can write the objective more compactly in vector/matrix form as

$$\boxed{\max_{\alpha \geq 0} \quad \mathcal{L}_D(\boldsymbol{\alpha}) = \boldsymbol{\alpha}^\top \mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}^\top \mathbf{G}\boldsymbol{\alpha}}$$

  where $\mathbf{G}$ is an $N \times N$ matrix with $G_{mn} = y_m y_n \boldsymbol{x}_m^\top \boldsymbol{x}_n$, and $\mathbf{1}$ is a vector of 1s

- **Good news:** This is maximizing a concave function (or minimizing a convex function - verify that the Hessian is $\mathbf{G}$, which is p.s.d.). Note that our original SVM objective was also convex

- **Important:** Inputs $\boldsymbol{x}$'s only appear as inner products (helps to "kernelize"; more when we see kernel methods)

- Can solve[†] the above objective function for $\boldsymbol{\alpha}$ using various methods, e.g.,
  - Treating the objective as a Quadratic Program (QP) and running some off-the-shelf QP solver such as `quadprog` (MATLAB), CVXOPT, CPLEX, etc.
  - Using (projected) gradient methods (projection needed because the α's are constrained). Gradient methods will usually be much faster than QP methods.

[†] If interested in more details of the solver, see: "Support Vector Machine Solvers" by Bottou and Lin

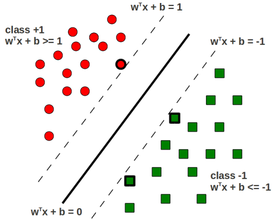
Hard-Margin SVM: The Solution

- Once we have the α_n's, $\boldsymbol{w}$ and b can be computed as:

$$\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n y_n \boldsymbol{x}_n \qquad \text{(we already saw this)}$$

$$b = -\frac{1}{2}\left(\min_{n:y_n=+1} \boldsymbol{w}^T \boldsymbol{x}_n + \max_{n:y_n=-1} \boldsymbol{w}^T \boldsymbol{x}_n\right) \qquad \text{(exercise)}$$

- **A nice property:** Most α_n's in the solution will be zero (sparse solution)



- Reason: Karush-Kuhn-Tucker (KKT) conditions
- For the optimal α_n's

$$\alpha_n\{1 - y_n(\boldsymbol{w}^T \boldsymbol{x}_n + b)\} = 0$$

- α_n is non-zero only if $\boldsymbol{x}_n$ lies on one of the two margin boundaries, i.e., for which $y_n(\boldsymbol{w}^T \boldsymbol{x}_n + b) = 1$
- These examples are called support vectors
- Recall the support vectors "support" the margin boundaries

Solving Soft-Margin SVM

Solving Soft-Margin SVM

- Recall the soft-margin SVM optimization problem:

$$\begin{aligned} \min_{\boldsymbol{w}, b, \boldsymbol{\xi}} \quad & f(\boldsymbol{w}, b, \boldsymbol{\xi}) = \frac{||\boldsymbol{w}||^2}{2} + C \sum_{n=1}^{N} \xi_n \\ \text{subject to} \quad & 1 \leq y_n(\boldsymbol{w}^T \boldsymbol{x}_n + b) + \xi_n, \quad -\xi_n \leq 0 \qquad n = 1, \ldots, N \end{aligned}$$

- Note: $\boldsymbol{\xi} = [\xi_1, \ldots, \xi_N]$ is the vector of slack variables
- Introduce Lagrange Multipliers α_n, β_n $(n = \{1, \ldots, N\})$, for constraints, and solve the Lagrangian:

$$\min_{\boldsymbol{w}, b, \boldsymbol{\xi}} \max_{\boldsymbol{\alpha} \geq 0, \boldsymbol{\beta} \geq 0} \mathcal{L}(\boldsymbol{w}, b, \xi, \alpha, \beta) = \frac{||\boldsymbol{w}||^2}{2} + C \sum_{n=1}^{N} \xi_n + \sum_{n=1}^{N} \alpha_n \{1 - y_n(\boldsymbol{w}^T \boldsymbol{x}_n + b) - \xi_n\} - \sum_{n=1}^{N} \beta_n \xi_n$$

- Note: The terms in red above were not present in the hard-margin SVM
- Two sets of dual variables $\boldsymbol{\alpha} = [\alpha_1, \ldots, \alpha_N]$ and $\boldsymbol{\beta} = [\beta_1, \ldots, \beta_N]$. We'll eliminate the primal variables $\boldsymbol{w}, b, \boldsymbol{\xi}$ to get dual problem containing the dual variables (just like in the hard margin case)

Solving Soft-Margin SVM

- The Lagrangian problem to solve

$$\boxed{\min_{\boldsymbol{w},b,\boldsymbol{\xi}} \max_{\boldsymbol{\alpha}\geq 0,\boldsymbol{\beta}\geq 0} \mathcal{L}(\boldsymbol{w},b,\xi,\alpha,\beta) = \frac{\boldsymbol{w}^\top \boldsymbol{w}}{2} + +C\sum_{n=1}^{N}\xi_n + \sum_{n=1}^{N}\alpha_n\{1-y_n(\boldsymbol{w}^T\boldsymbol{x}_n+b)-\xi_n\} - \sum_{n=1}^{N}\beta_n\xi_n}$$

- Take (partial) derivatives of $\mathcal{L}$ w.r.t. $\boldsymbol{w}$, b, ξ_n and set them to zero

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = 0 \Rightarrow \boxed{\boldsymbol{w} = \sum_{n=1}^{N}\alpha_n y_n \boldsymbol{x}_n}, \quad \frac{\partial \mathcal{L}}{\partial b} = 0 \Rightarrow \sum_{n=1}^{N}\alpha_n y_n = 0, \quad \frac{\partial \mathcal{L}}{\partial \xi_n} = 0 \Rightarrow C - \alpha_n - \beta_n = 0$$

- Note: Solution of $\boldsymbol{w}$ again has the same form as in the hard-margin case (weighted sum of all inputs with α_n being the importance of input $\boldsymbol{x}_n$)
- Note: Using $C - \alpha_n - \beta_n = 0$ and $\beta_n \geq 0 \Rightarrow \alpha_n \leq C$ (recall that, for the hard-margin case, $\alpha \geq 0$)
- Substituting these in the Lagrangian $\mathcal{L}$ gives the Dual problem

$$\boxed{\max_{\boldsymbol{\alpha}\leq C,\boldsymbol{\beta}\geq 0} \mathcal{L}_D(\boldsymbol{\alpha},\boldsymbol{\beta}) = \sum_{n=1}^{N}\alpha_n - \frac{1}{2}\sum_{m,n=1}^{N}\alpha_m\alpha_n y_m y_n(\boldsymbol{x}_m^T\boldsymbol{x}_n) \quad \text{s.t.} \quad \sum_{n=1}^{N}\alpha_n y_n = 0}$$

Solving Soft-Margin SVM

- Interestingly, the dual variables $\boldsymbol{\beta}$ don't appear in the objective!
- Just like the hard-margin case, we can write the dual more compactly as

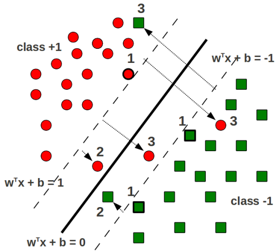
$$\boxed{\max_{\boldsymbol{\alpha} \leq C} \quad \mathcal{L}_D(\boldsymbol{\alpha}) = \boldsymbol{\alpha}^\top \mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}^\top \mathbf{G}\boldsymbol{\alpha}}$$

  where $\mathbf{G}$ is an $N \times N$ matrix with $G_{mn} = y_m y_n \boldsymbol{x}_m^\top \boldsymbol{x}_n$, and $\mathbf{1}$ is a vector of 1s
- Like hard-margin case, solving the dual requires concave maximization (or convex minimization)
- Can be solved[†] the same way as hard-margin SVM (except that $\boldsymbol{\alpha} \leq C$)
  - Can solve for $\boldsymbol{\alpha}$ using QP solvers or (projected) gradient methods
- Given $\boldsymbol{\alpha}$, the solution for $\boldsymbol{w}, b$ has the same form as hard-margin case
- **Note:** $\boldsymbol{\alpha}$ is again sparse. Nonzero α_n's correspond to the support vectors

[†] If interested in more details of the solver, see: "Support Vector Machine Solvers" by Bottou and Lin

Support Vectors in Soft-Margin SVM

- The hard-margin SVM solution had only one type of support vectors
  - .. ones that lie on the margin boundaries $\boldsymbol{w}^T\boldsymbol{x} + b = -1$ and $\boldsymbol{w}^T\boldsymbol{x} + b = +1$
- The soft-margin SVM solution has three types of support vectors



1. Lying on the margin boundaries $\boldsymbol{w}^T\boldsymbol{x} + b = -1$ and $\boldsymbol{w}^T\boldsymbol{x} + b = +1$ ($\xi_n = 0$)
2. Lying within the margin region ($0 < \xi_n < 1$) but still on the correct side
3. Lying on the wrong side of the hyperplane ($\xi_n \geq 1$)

SVMs via Dual Formulation: Some Comments

- Recall the final dual objectives for hard-margin and soft-margin SVM

$$\boxed{\text{Hard-Margin SVM:} \quad \max_{\boldsymbol{\alpha} \geq 0} \ \mathcal{L}_D(\boldsymbol{\alpha}) = \boldsymbol{\alpha}^\top \mathbf{1} - \frac{1}{2} \boldsymbol{\alpha}^\top \mathbf{G} \boldsymbol{\alpha}}$$

$$\boxed{\text{Soft-Margin SVM:} \quad \max_{\boldsymbol{\alpha} \leq C} \ \mathcal{L}_D(\boldsymbol{\alpha}) = \boldsymbol{\alpha}^\top \mathbf{1} - \frac{1}{2} \boldsymbol{\alpha}^\top \mathbf{G} \boldsymbol{\alpha}}$$

- The dual formulation is nice due to two primary reasons:
  - Allows conveniently handling the margin based constraint (via Lagrangians)
  - Important: Allows learning nonlinear separators by replacing inner products (e.g., $G_{mn} = y_m y_n \boldsymbol{x}_m^\top \boldsymbol{x}_n$) by kernelized similarities (kernelized SVMs)
- However, the dual formulation can be expensive if N is large. Have to solve for N variables $\boldsymbol{\alpha} = [\alpha_1, \ldots, \alpha_N]$, and also need to store an $N \times N$ matrix $\mathbf{G}$
- A lot of work[†] on speeding up SVM in these settings (e.g., can use co-ord. descent for $\boldsymbol{\alpha}$)

[†] See: "Support Vector Machine Solvers" by Bottou and Lin

SVM: Some Notes

- A hugely (perhaps the most!) popular classification algorithm
- Reasonably mature, highly optimized SVM softwares freely available (perhaps the reason why it is more popular than various other competing algorithms)
  - Some popular ones: libSVM, LIBLINEAR, sklearn also provides SVM
- Lots of work on scaling up SVMs† (both large N and large D)
- Extensions beyond binary classification (e.g., multiclass, structured outputs)
- Can even be used for regression problems (Support Vector Regression)
- Nonlinear extensions possible via kernels

† See: "Support Vector Machine Solvers" by Bottou and Lin