



Machine Learning and Data Mining

Lecture 5.1: Soft-Margin Support Vector Machines and Kernels



- Mix of multiple-choice / true-false / and written response.
- Concept quizzes are good examples of some of the basic concept questions
- Written response will sometimes include computation. Mostly deal with the operation of algorithms we've covered:
 - kNN, logistic regression, naïve bayes, SVM

Points	Section
_____ / 2	Bonus Questions On This Page
_____ / 40	Multiple Choice Concept Questions
_____ / 17	Support Vector Machines
_____ / 17	kNN and LOO Cross-Validation
_____ / 17	Fitting Binary Naïve Bayes Models
_____ / 9	Logistic Regression and Regularization
_____ / 100	Total



Consider computing the logistic:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Two cases to consider:

- **z is large and positive**
 - $z \rightarrow \infty$. $e^{-z} \rightarrow 0$. $\sigma(z) \rightarrow 1$
- **z is large and negative**
 - $z \rightarrow -\infty$. $e^{-z} \rightarrow \infty$. $\sigma(z) \rightarrow 0$

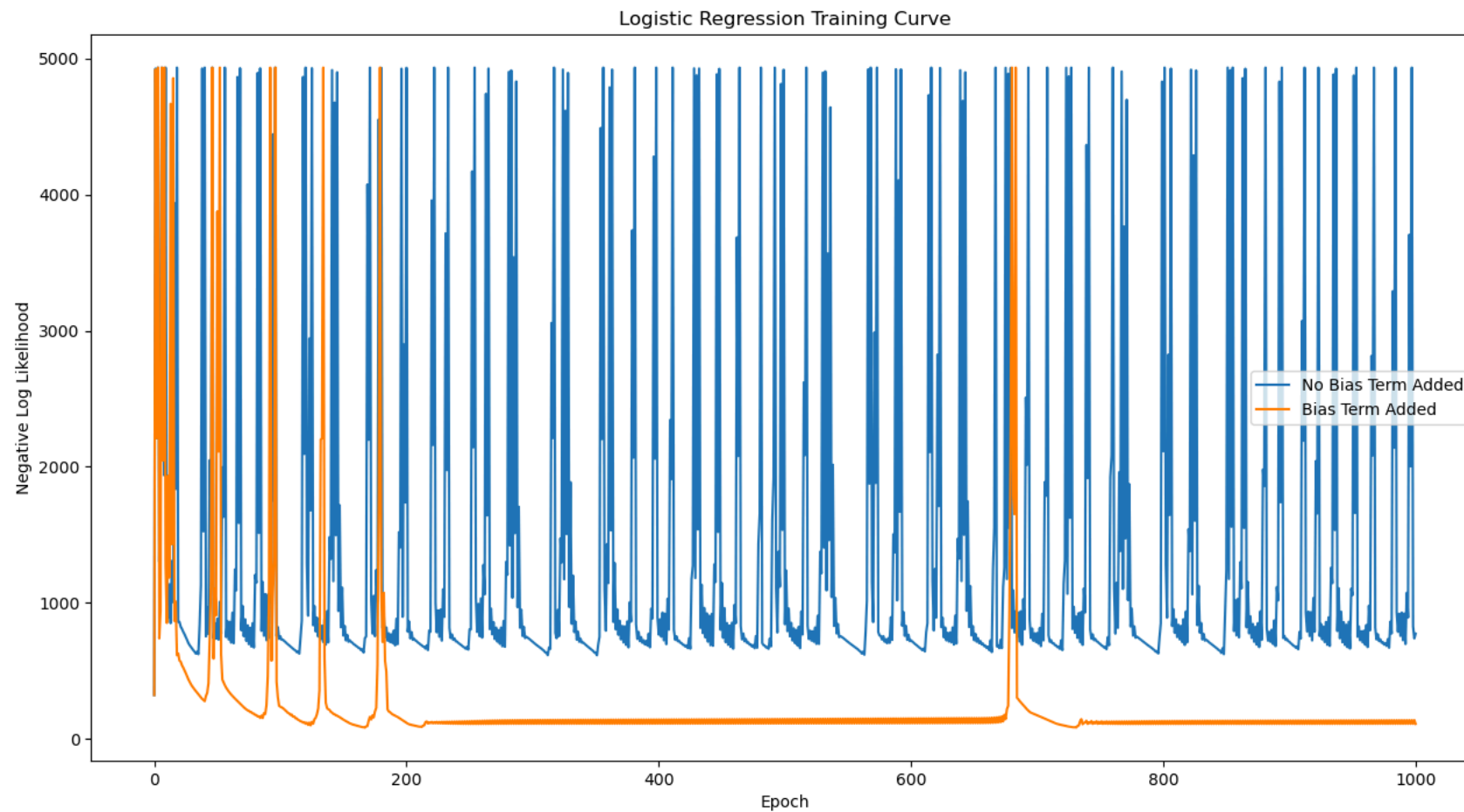


This time, consider computing the log-logistic:

$$\log \sigma(z) = \log \frac{1}{1 + e^{-z}}$$

Two cases to consider:

- **z is large and positive**
 - $z \rightarrow \infty$. $e^{-z} \rightarrow 0$. $\sigma(z) \rightarrow 1$. $\log \sigma(z) \rightarrow 0$
- **z is large and negative**
 - $z \rightarrow -\infty$. $e^{-z} \rightarrow \infty$. $\sigma(z) \rightarrow 0$. $\log \sigma(z) \rightarrow \infty$





This time, consider computing the log-logistic:

$$\begin{aligned}\log \sigma(\mathbf{z}) &= \log \frac{1}{1 + e^{-\mathbf{z}}} = \log 1 - \log(1 + e^{-\mathbf{z}}) \\ &= -\log(1 + e^{-\mathbf{z}}) = -\log(e^0 + e^{-\mathbf{z}}) \\ &= -\log(e^{-\mathbf{z}}(e^{\mathbf{z}} + e^0)) \\ &= -\log(e^{\mathbf{z}} + e^0) - \log(e^{-\mathbf{z}}) \\ &= -\log(e^{\mathbf{z}} + 1) + \mathbf{z}\end{aligned}$$



This time, consider computing the log-logistic:

$$\log \sigma(\mathbf{z}) = \log \frac{1}{1 + e^{-\mathbf{z}}} = \log 1 - \log(1 + e^{-\mathbf{z}})$$

$$\text{(A)} \quad = -\log(1 + e^{-\mathbf{z}}) = -\log(e^0 + e^{-\mathbf{z}})$$

$$= -\log(e^{-\mathbf{z}}(e^{\mathbf{z}} + e^0))$$

$$= -\log(e^{\mathbf{z}} + e^0) - \log(e^{-\mathbf{z}})$$

$$\text{(B)} \quad = -\log(e^{\mathbf{z}} + 1) + \mathbf{z}$$



This time, consider computing the log-logistic:

$$\log \sigma(\mathbf{z}) = \log \frac{1}{1 + e^{-\mathbf{z}}} = -\log(1 + e^{-\mathbf{z}}) \quad (\text{A})$$

$$= -\log(e^{\mathbf{z}} + 1) + \mathbf{z} \quad (\text{B})$$

Two cases to consider:

- **z is large and positive (use A)**
 - $z \rightarrow \infty$. $e^{-z} \rightarrow 0$. $\log(1 + 0) \rightarrow \log 1$. $\log \sigma(z) \rightarrow 0$
- **z is large and negative (use B)**
 - $z \rightarrow -\infty$. $e^z \rightarrow 0$. $\log(0 + 1) \rightarrow \log 1$. $\log \sigma(z) \rightarrow z$



This time, consider computing the log-logistic:

$$\log \sigma(\mathbf{z}) = \log \frac{1}{1 + e^{-\mathbf{z}}} = -\log(1 + e^{-\mathbf{z}}) \quad (\text{A})$$

$$= -\log(e^{\mathbf{z}} + 1) + \mathbf{z} \quad (\text{B})$$

Let $c = \max(0, -\mathbf{z})$: $\log \sigma(\mathbf{z}) = -\log(e^{0-c} + e^{-\mathbf{z}-c}) - c$

$$\log \sigma(\mathbf{z}) = -\log(e^{0-0} + e^{-\mathbf{z}-0}) - 0 = -\log(1 + e^{-\mathbf{z}}) \quad (\text{A when } \mathbf{z} > 0)$$

$$\log \sigma(\mathbf{z}) = -\log(e^{0+\mathbf{z}} + e^{-\mathbf{z}+\mathbf{z}}) + \mathbf{z} = -\log(e^{\mathbf{z}} + 1) + \mathbf{z} \quad (\text{B when } \mathbf{z} < 0)$$



Numpy (and other packages) know this is a common problem:

numpy.logaddexp

```
numpy.logaddexp(x1, x2, /, out=None, *, where=True, casting='same_kind',  
order='K', dtype=None, subok=True[, signature, extobj]) = <ufunc 'logaddexp'> ¶
```

Logarithm of the sum of exponentiations of the inputs.

Calculates $\log(\exp(x1) + \exp(x2))$. This function is useful in statistics where the calculated probabilities of events may be so small as to exceed the range of normal floating point numbers. In such cases the logarithm of the calculated probability is stored. This function allows adding probabilities stored in such a fashion.

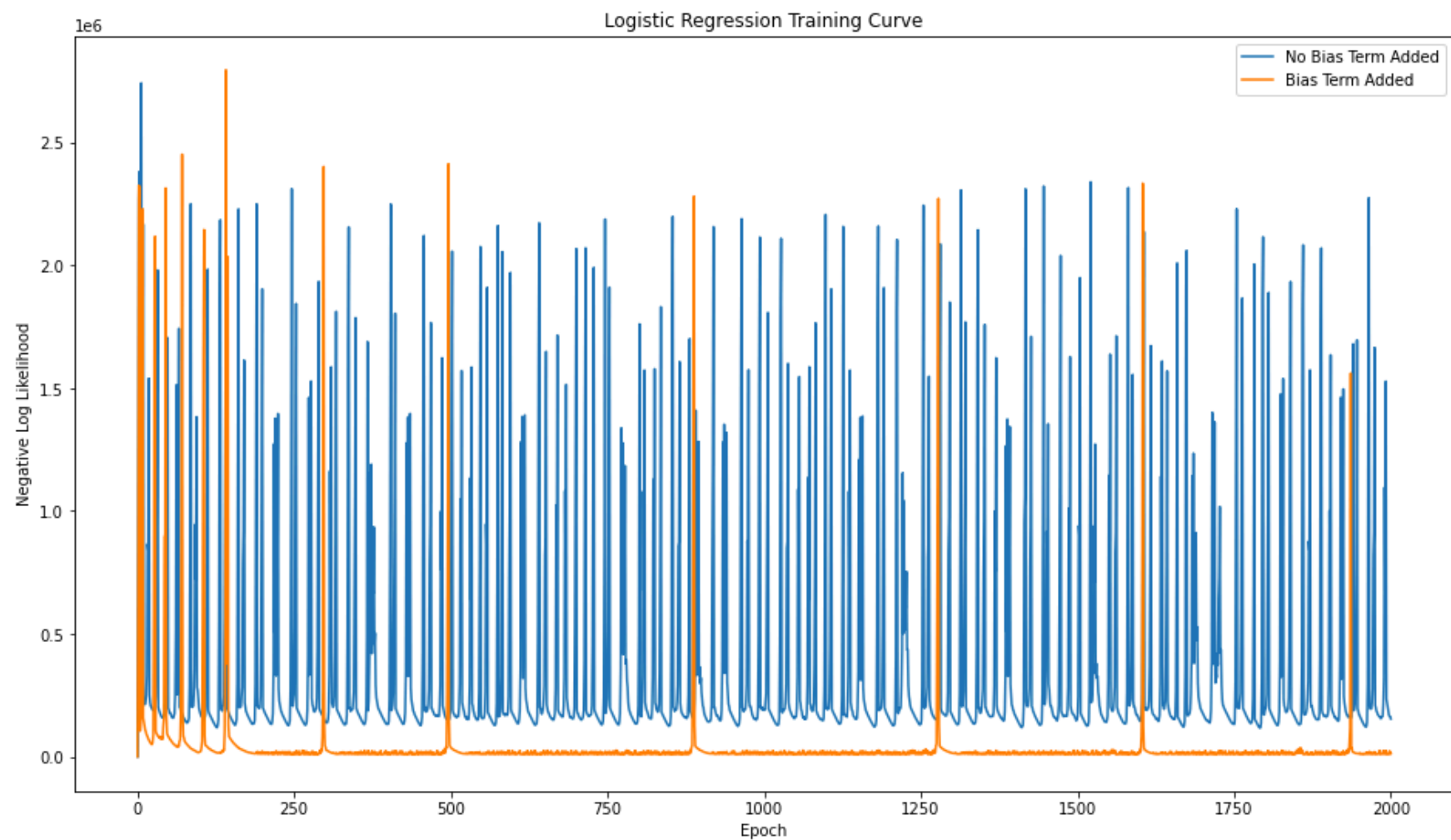
Works the same way under the hood.

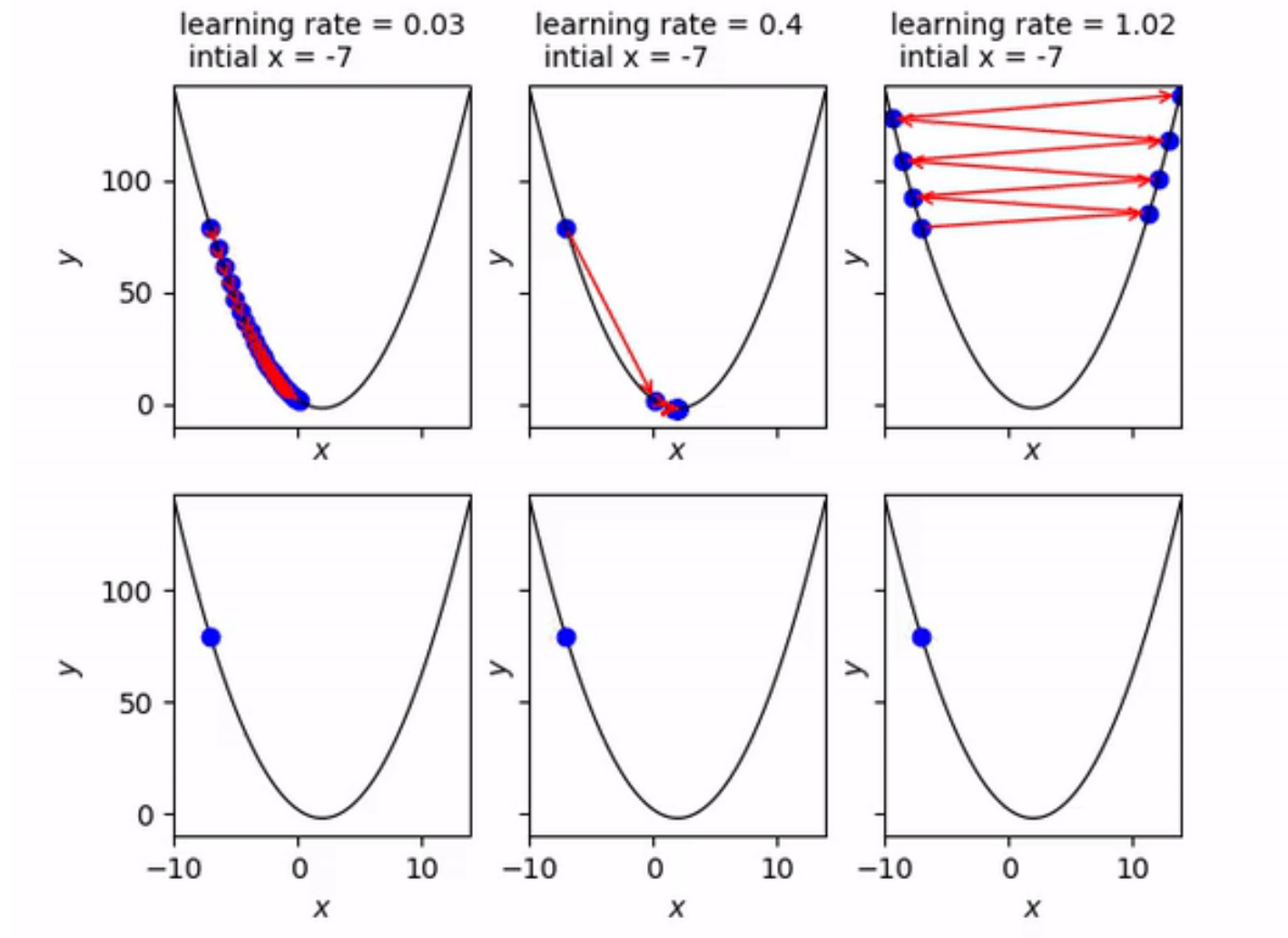


Let's look at this in a colab:

<https://colab.research.google.com/drive/17egtw1j2MkmWKIU52Y62dIW0pqYBIAhR?usp=sharing>

Step size = 1

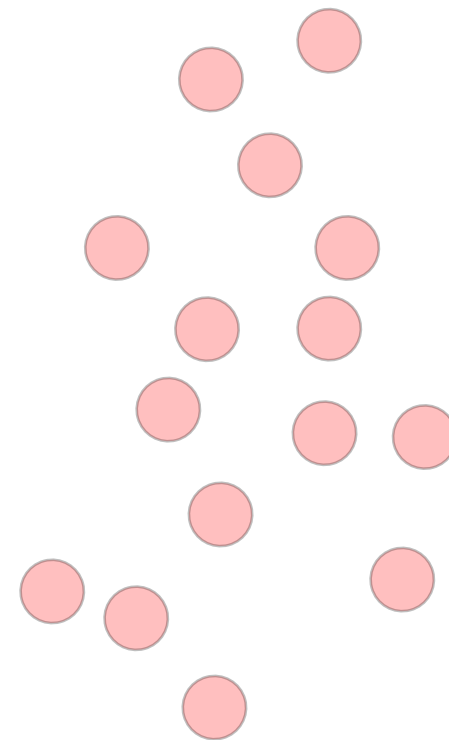
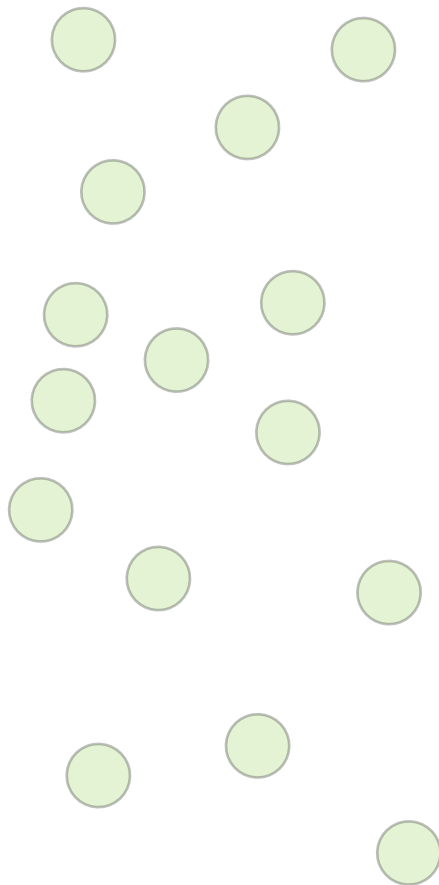


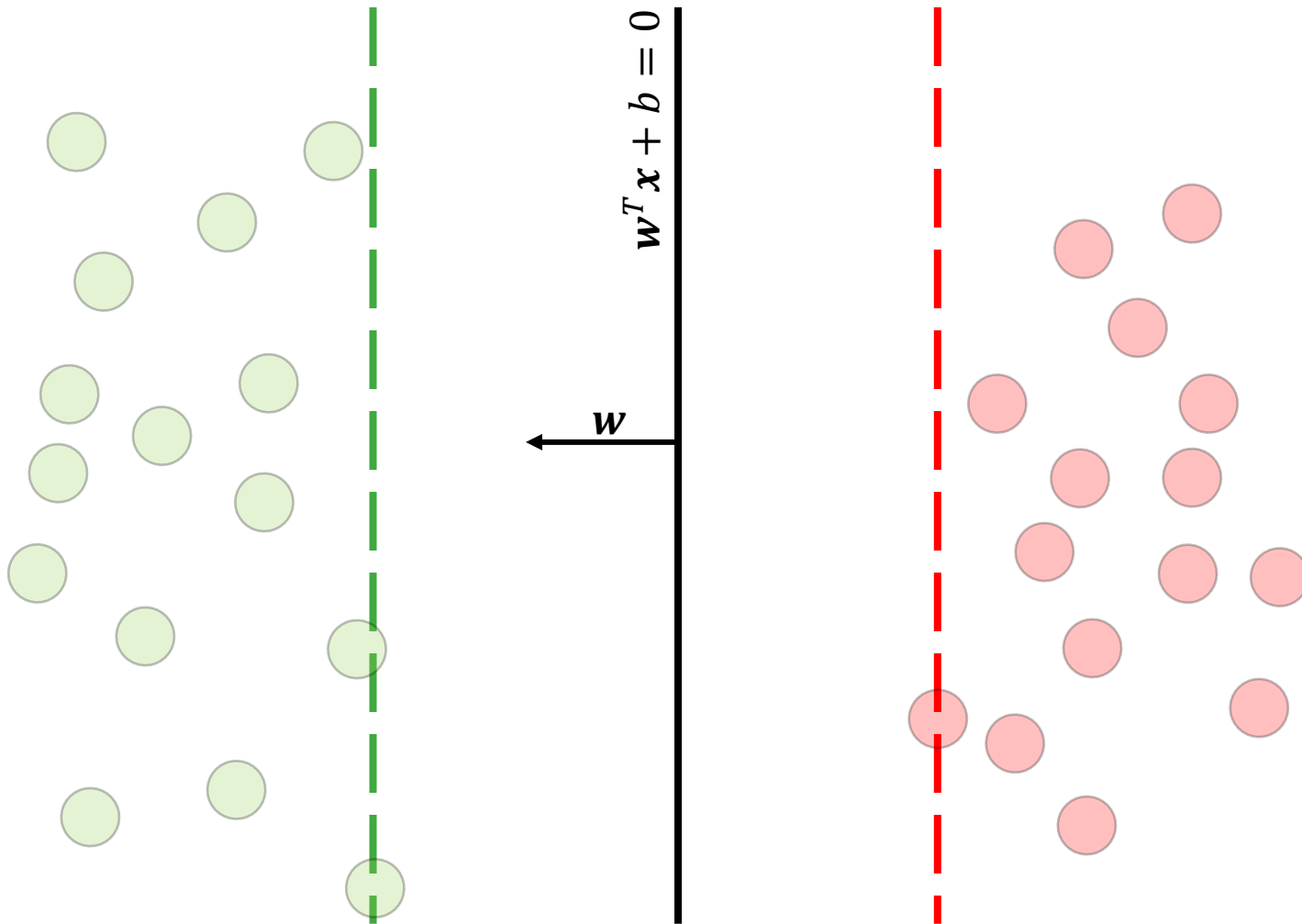




RECAP

From Last Lecture





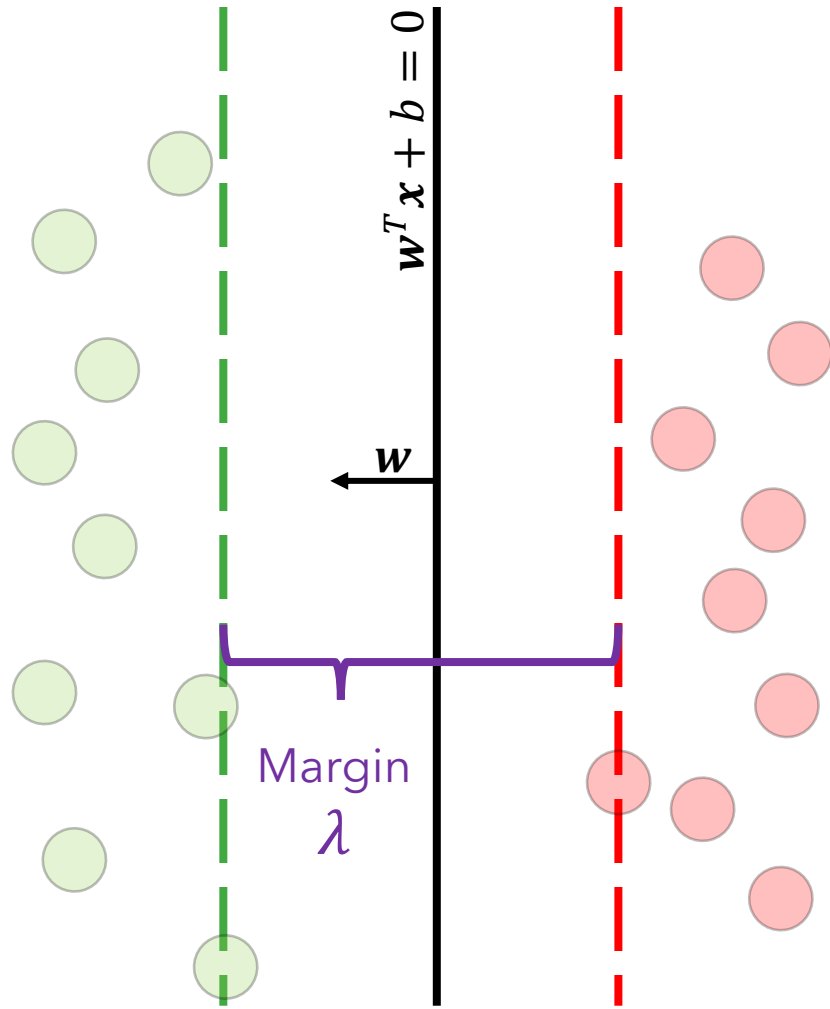
Algorithm Name: Support Vector Machines (SVM)

Type: Supervised Linear Classifier

Key Idea: Find the linear decision boundary that maximizes the margin between positive and negative examples.



What is a support vector? We'll need to run through a lot of somewhat unintuitive math to really understand the structure of this problem. But the result will give us tremendous insight into this problem ... and it'll build character.



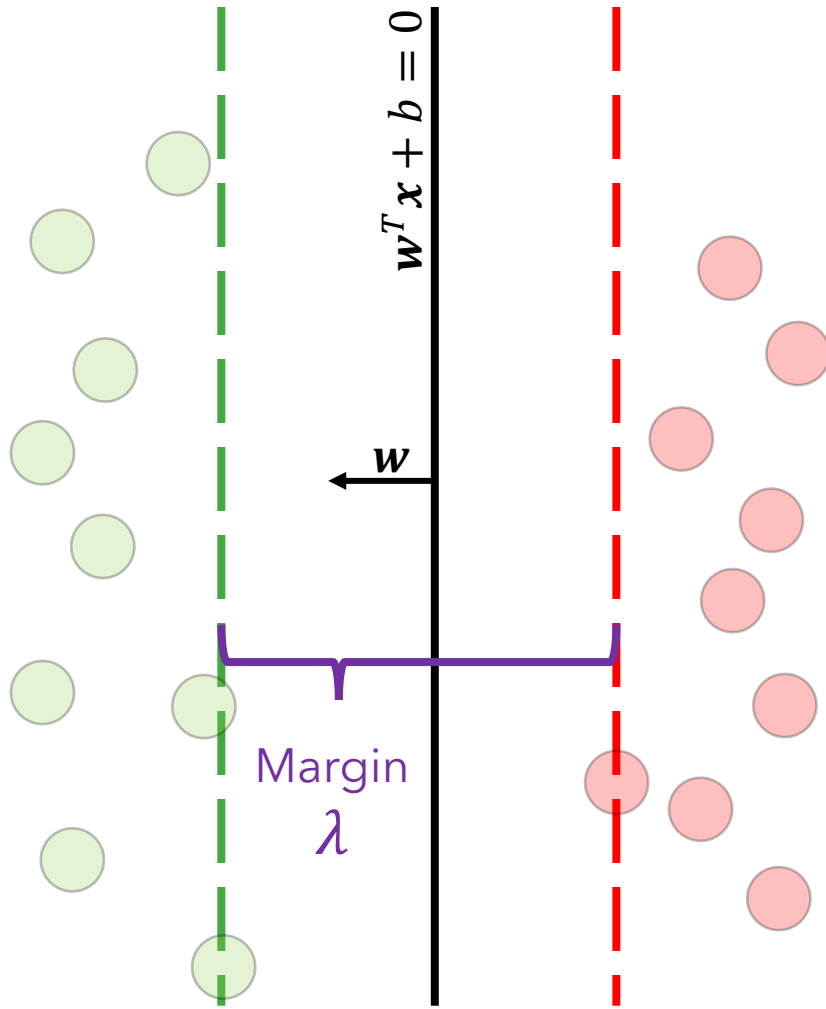
Intuition for Hard-Margin SVM:

We want to find a classifier that separates the classes while maximizing the margin between positive and negative examples.

Assumes linear separability.

$$\max_{w,b} \text{margin}$$

s. t. all examples are correct



Our goal for today:

We want to find a classifier that separates the classes while maximizing the margin between positive and negative examples. *For now, we'll assume linear separability and fix that later.*

Our plan of attack:

1. Set up some convenient notation
2. Derive an expression for our intuition
3. Set up a constrained optimization problem
4. Solve it to arrive at a Support Vector Machine
(some new math tools to analyze this- Lagrange multipliers)



Step 1) Establish Some Convenient Notation

Problem Setup and Decision Rule

Consider a binary classification setting where we represent the classes as $y \in \{-1, 1\}$

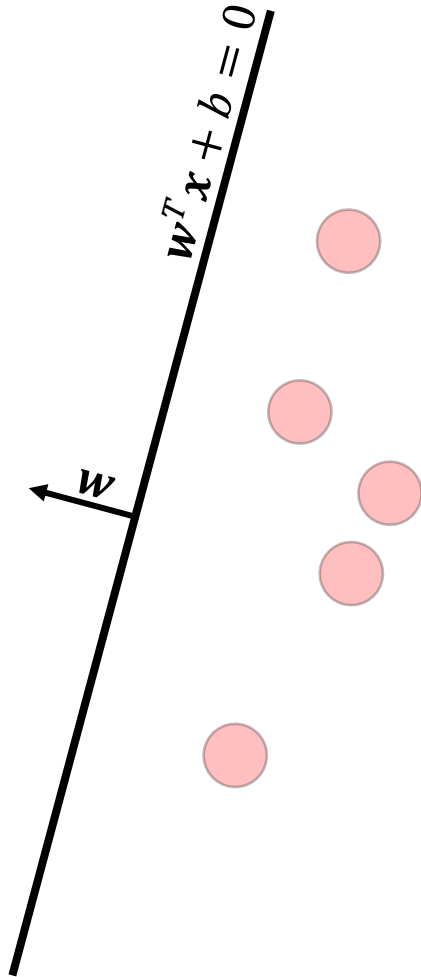
Let our linear decision rule be:

$$\mathbf{w}^T \mathbf{x} + b > 0 \Rightarrow \text{predict } 1$$

$$\mathbf{w}^T \mathbf{x} + b < 0 \Rightarrow \text{predict } -1$$

Means a correct classification has:

$$y_i(\mathbf{w}^T \mathbf{x}_i + b) > 0$$





How do we actually solve this? And by “solve” I mean optimal find \mathbf{w} and b

$$\min_{\mathbf{w}, b} \frac{1}{2} \mathbf{w}^T \mathbf{w}$$

$$s. t. \quad y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 \quad \forall i$$

This is a quadratic function with linear constraints. Many (non-trivial) algorithms to solve these sorts of things – called Quadratic Program Solvers (QP-Solvers).



What is a Quadratic Program Solver? Given equations of the form:

$$\begin{aligned} \min_{\mathbf{z}} \quad & \frac{1}{2} \mathbf{z}^T \mathbf{P} \mathbf{z} + \mathbf{q}^T \mathbf{z} && \text{Quadratic equation in matrix notation} \\ \text{s.t.} \quad & \mathbf{G} \mathbf{z} \leq \mathbf{h} && \text{Linear inequality constraints} \\ & \mathbf{A} \mathbf{z} = \mathbf{b} && \text{Linear equality constraints} \\ & \mathbf{b}_{\text{lower}} \leq \mathbf{z} \leq \mathbf{b}_{\text{upper}} && \text{Limits on the values of } \mathbf{z} \end{aligned}$$

Runs algorithms from optimization to provide the vector \mathbf{z}^* that minimizes the function without violating the constraints. (Or returns infeasible if no such solution exists)

- One example from Python (<https://pypi.org/project/qpsolvers/>)



Hard-Margin Support Vector Machine Objective

How can we put this into a quadratic form?

$$\min_{\mathbf{w}, b} \frac{1}{2} \mathbf{w}^T \mathbf{w}$$

$$s.t. \quad y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 \quad \forall i$$



$$\min_{\mathbf{z}} \frac{1}{2} \mathbf{z}^T P \mathbf{z} + \mathbf{q}^T \mathbf{z}$$

$$s.t. \quad G \mathbf{z} \leq \mathbf{h}$$

$$A \mathbf{z} = \mathbf{b}$$

$$\mathbf{b}_{lower} \leq \mathbf{z} \leq \mathbf{b}_{upper}$$

$$\mathbf{z} = \begin{bmatrix} \mathbf{w} \\ b \end{bmatrix}_{(d+1) \times 1}$$

$$P = \begin{bmatrix} I_d & 0 \\ 0 & 0 \end{bmatrix}_{(d+1) \times (d+1)}$$

$$\mathbf{q} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}_{d+1}$$

$$\mathbf{h} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}_{n+1}$$

$$G = \begin{bmatrix} y_1 \mathbf{x}_1^T, y_1 \\ y_2 \mathbf{x}_2^T, y_2 \\ y_3 \mathbf{x}_3^T, y_3 \\ \vdots \\ y_n \mathbf{x}_n^T, y_n \end{bmatrix}_{n \times (d+1)}$$



How can we put this into a quadratic form?

$$\min_{\mathbf{w}, b} \frac{1}{2} \mathbf{w}^T \mathbf{w}$$

$$s. t. \quad y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 \quad \forall i$$



$$\min_{\mathbf{w}, b} \frac{1}{2} [\mathbf{w} \ b] \begin{bmatrix} I_d & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{w} \\ b \end{bmatrix} + \vec{0}^T \begin{bmatrix} \mathbf{w} \\ b \end{bmatrix}$$

$$s. t. \quad \begin{bmatrix} y_1 \mathbf{x}_1^T, y_1 \\ y_2 \mathbf{x}_2^T, y_2 \\ y_3 \mathbf{x}_3^T, y_3 \\ \vdots \\ y_n \mathbf{x}_n^T, y_n \end{bmatrix} \begin{bmatrix} \mathbf{w} \\ b \end{bmatrix} \geq \vec{\mathbf{1}}$$

Can pass this to a QP solver and let some optimization people deal with it.

For our purposes however, there is some deeper bit of understanding to be gained from pushing this further with another method - Lagrange multipliers.

<https://colab.research.google.com/drive/1LTpGaj0ALwlrzq84yho6mpXTi9l4pwGN?usp=sharing>



Hard-Margin SVM Primal and Dual Formulations

Hard-Margin
SVM Primal:

$$\min_{\mathbf{w}, b} \frac{1}{2} \mathbf{w}^T \mathbf{w} \quad s. t. \quad y_i (\mathbf{w}^T \mathbf{x}_i + b) \geq 1 \quad \forall i$$

Find best weights and bias directly

Hard-Margin
SVM Dual:

$$\max_{\alpha_i \geq 0} \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_i^n \sum_j^n \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \quad s. t. \quad \sum_{i=1}^n \alpha_i y_i = 0$$

Find these optimization variables alpha, then derive best weights and bias



Recap of what we've been doing so far

Motivation: We wanted to find a classifier that maximizes the margin between classes.

Derivation: We derived the following constrained optimization problem that does that:

Hard-Margin
SVM Primal:

$$\min_{\mathbf{w}, b} \frac{1}{2} \mathbf{w}^T \mathbf{w} \quad s.t. \quad y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 \quad \forall i$$

We said that is a Quadratic Programming problem and totally solvable. Then I ran us through a lot of math to arrive at the dual formalization below which solves the same problem (KKT's hold).

Hard-Margin
SVM Dual:

$$\max_{\alpha_i \geq 0} \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_i^n \sum_j^n \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \quad s.t. \quad \sum_{i=1}^n \alpha_i y_i = 0$$

Which is also a Quadratic Programming problem (in α 's this time rather than \mathbf{w}) and is totally solvable to global optimality..... **so why did I do this to us all?**



Examining the Dual Let Us Observe A Few Neat Things

Hard-Margin
SVM Dual:

$$\max_{\alpha_i \geq 0} \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_i^n \sum_j^n \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \quad s.t. \sum_{i=1}^n \alpha_i y_i = 0$$

Observation 1: The optimal weight vector is a linear combination of only a few of our training examples.

$$\mathbf{w}^* = \sum_{i=1}^n \alpha_i^* y_i \mathbf{x}_i$$

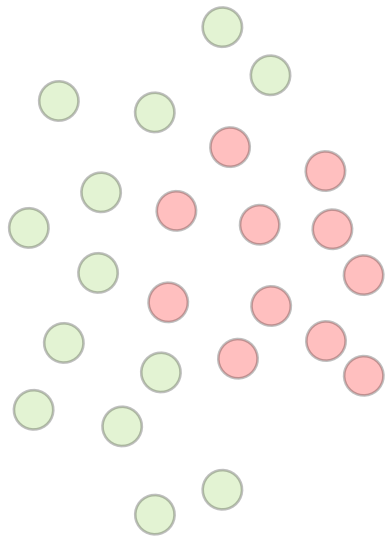
Observation 2: Both the objective and classifying new examples are just based on dot-products between input vectors.

$$\mathbf{w}^* = \sum_{i=1}^n \alpha_i^* y_i \mathbf{x}_i \quad \Rightarrow \quad \mathbf{w}^T \mathbf{x} + b = b + \sum_{i=1}^n \alpha_i^* y_i \mathbf{x}_i^T \mathbf{x}$$



Teaser for Next Time: What about the non-linearly separable case?

Teaser 1: We can modify our objective function to try to maximize margin while making as few mistakes as possible. Will need to introduce some notion of the tradeoff between the two.



Maximize margin



Minimize violations
of the margin



$$\min_{\mathbf{w}, b} \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_i \xi_i$$

$$\text{s.t. } y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 - \xi_i \quad \forall i$$
$$\xi_i \geq 0$$



Allow some
violations

We'll talk about how this works out next time.

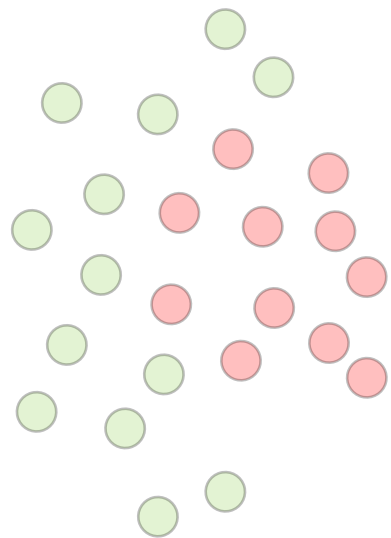


Teaser for Next Time: What about the non-linearly separable case?

Teaser 2: Apply the non-linear basis function idea again and just replace each point with some $\Phi(x)$ that is non-linear. Then fit a linear classifier in that higher-dimensional space.

Train and find $\mathbf{w}^* = \sum_{i=1}^n \alpha_i^* y_i \Phi(\mathbf{x}_i)$

Predict based on $\mathbf{w}^T \Phi(\mathbf{x}) + b \longrightarrow b + \sum_{i=1}^n \alpha_i^* y_i \Phi(\mathbf{x}_i)^T \Phi(\mathbf{x})$



This is expensive and we have to deal with those higher dimensions. What if we just defined a kernel that compute non-linear similarity without actually representing these high dimensional vectors.

$$b + \sum_{i=1}^n \alpha_i^* y_i \mathcal{K}(\mathbf{x}_i, \mathbf{x})$$

$$\max_{\alpha_i \geq 0} \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \mathcal{K}(\mathbf{x}_i, \mathbf{x}_j)$$

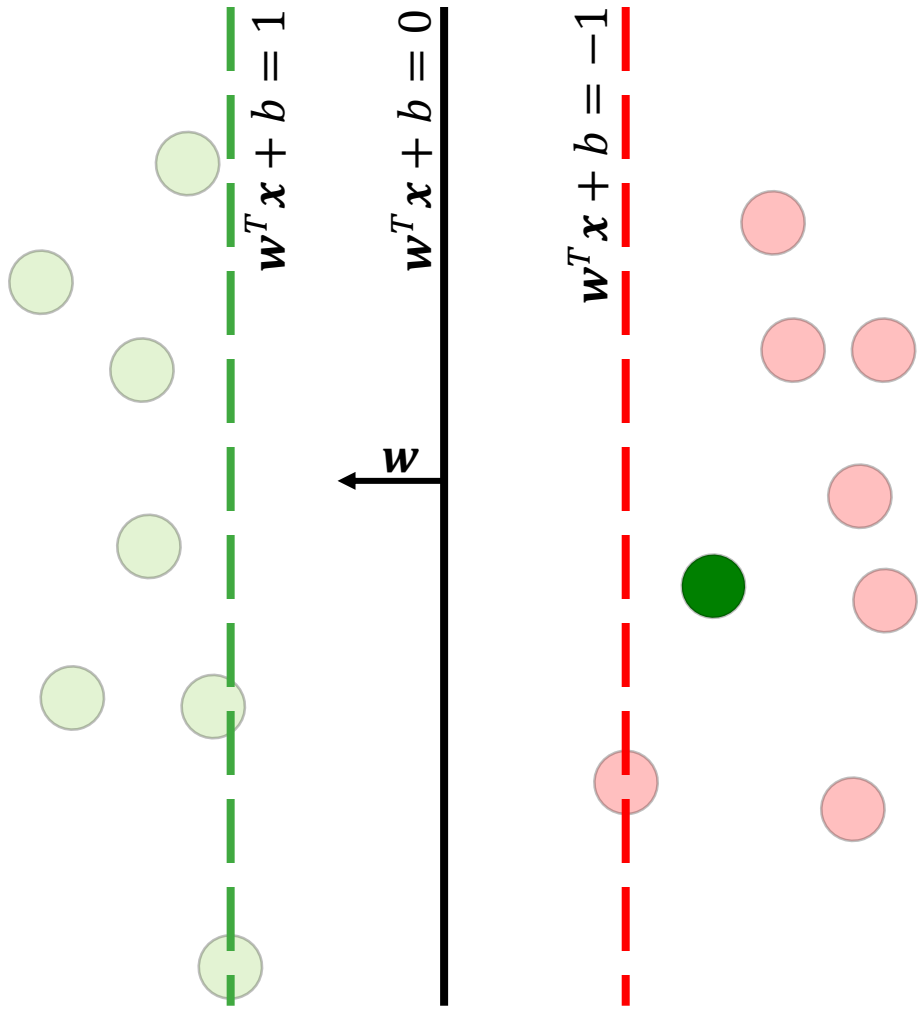
SVMs operate on dot products between examples. Can “pretend” we are in a higher dimension – call this the **kernel trick**. More next time.

Today's Learning Objectives

Be able to answer:

- How can we handle non-linearly separable data with a soft-margin Support Vector Machine?
 - What is the interpretation of a support vector now?
 - How can we implement one of these things with a QP solver?
- What is the kernel trick and why is it useful?
- What is the definition of a kernel?
- What is a kernel SVM?
- What else can we kernelize?
 - How do you kernelize perceptron?
 - How do you kernelize linear regression?





Hard-Margin SVMs assume the data is linearly separable – the optimization below has no solution for the case shown on the left:

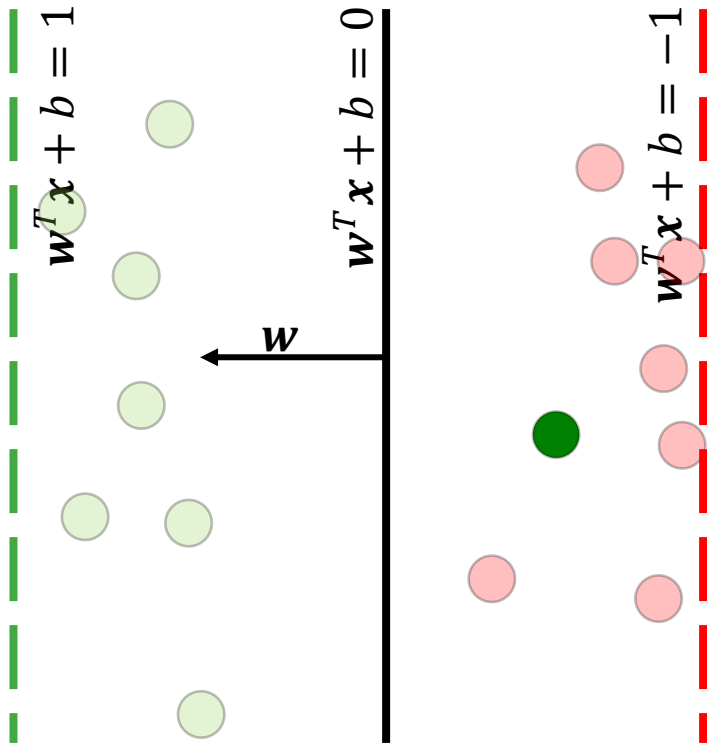
Hard-Margin
SVM Primal:

$$\min_{w,b} \frac{1}{2} w^T w$$
$$s.t. \quad y_i(w^T x_i + b) \geq 1 \quad \forall i$$

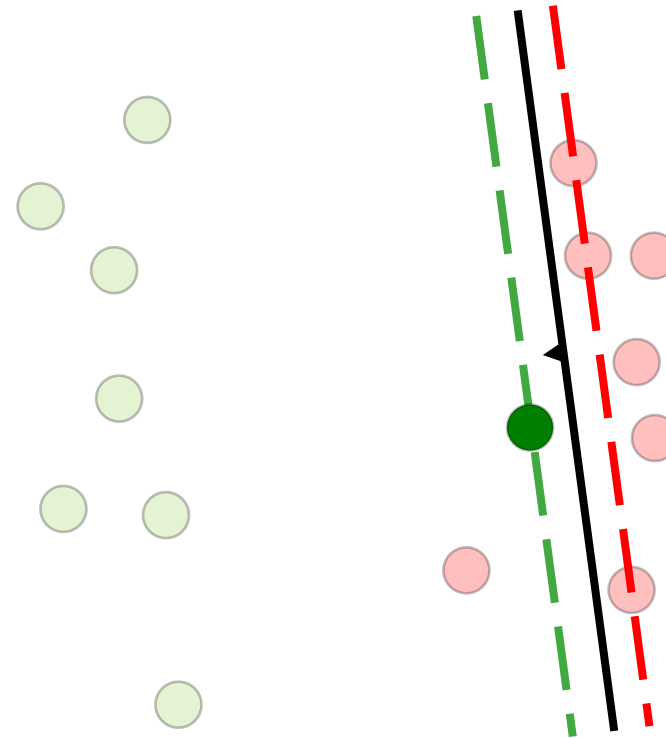
Our goal for today: Adjust this optimization to enable some misclassifications. Will derive the Soft-Margin SVM.

Idea: Maximize the margin while minimizing errors. Will balance these two opposing goals.

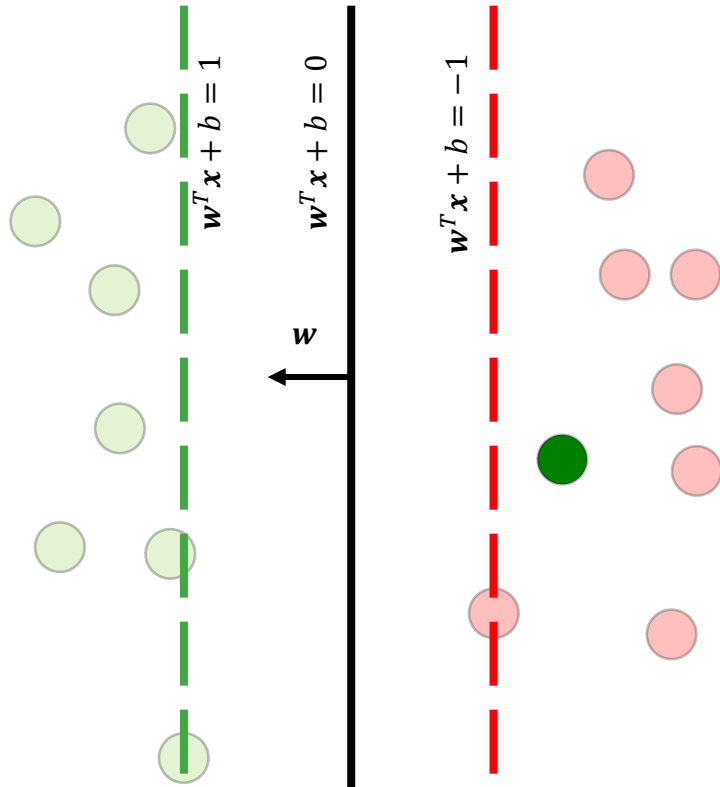
Very large margin.
Many errors.



Very small margin.
Few errors.



We already know how to maximize the margin. How can we express minimizing errors?



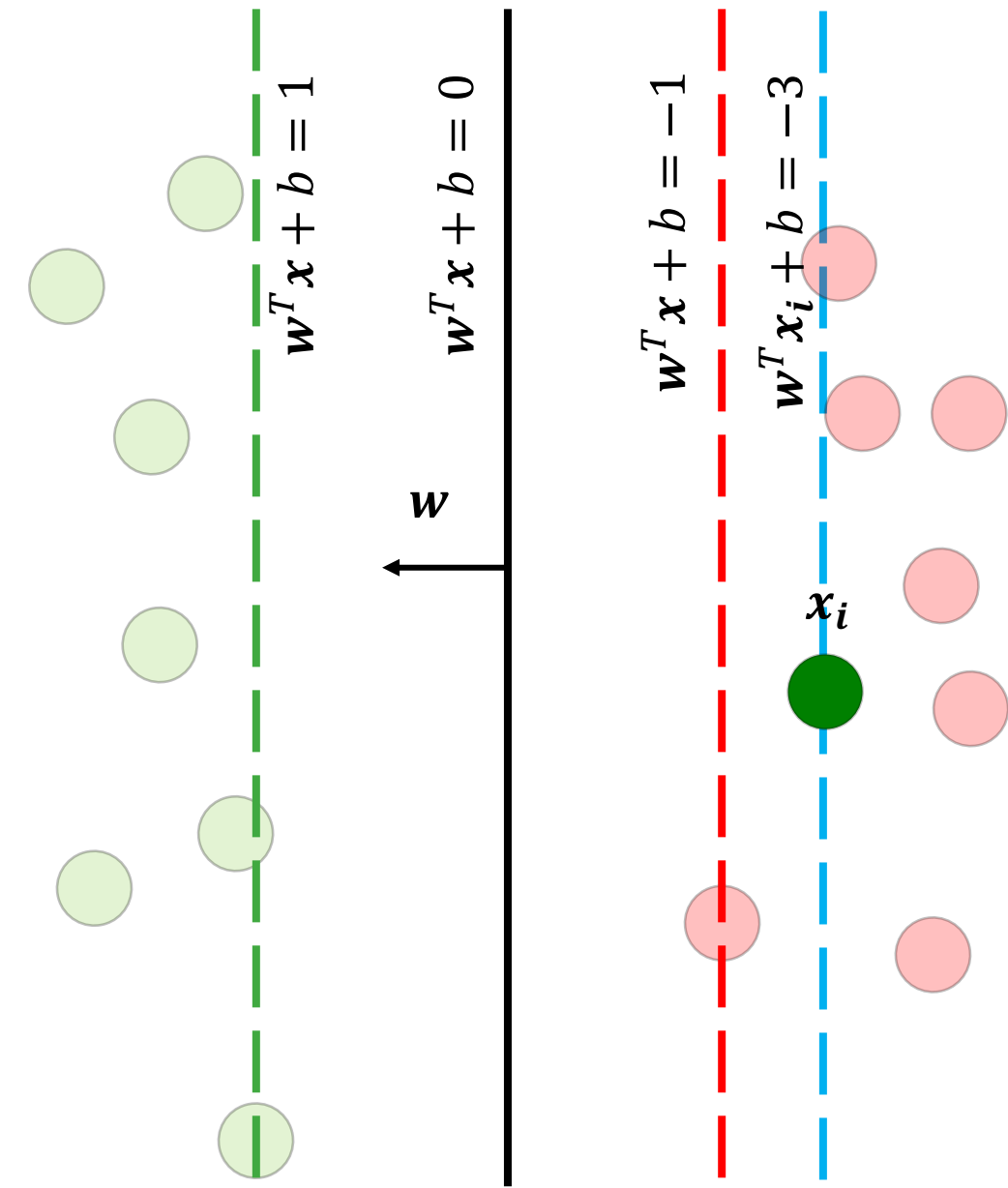
Last time, we had constraints that kept things out of the margin

$$y_i(w^T x_i + b) \geq 1 \quad \forall i$$

Let's allow each example to violate this by up to some value:

$$y_i(w^T x_i + b) \geq 1 - \xi_i \quad \forall i$$

So now each example i has some error tolerance $\xi_i \geq 0$. We refer to these as **slack variables**.



Let's get a feeling for the slack variables.

$$w^T = [0, -2], \quad x_i^T = [1, 2], \\ b = 1 \quad y_i = 1$$

$$w^T x_i + b = -3$$

$$y_i(w^T x_i + b) = -3$$

If we want the constraint below to hold:


$$y_i(w^T x_i + b) \geq 1 - \xi_i \quad \forall i$$


Then the slack variable ξ_i needs to be ≥ 4

We can combine this “soft error” constraint with our previous optimization. This results in the **soft-margin SVM** primal objective.


Soft-Margin SVM Primal:

$$\min_{\mathbf{w}, b} \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_i \xi_i$$

Maximize margin 

Minimize violations of the margin 

$$s. t. \quad y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 - \xi_i \quad \forall i$$
$$\xi_i \geq 0$$

 Allow some violations

Call the hyperparameter **C** the **slack penalty**. Let's do some questions to see if this formulation passes the sniff test.



What happens as if the **slack penalty** C goes to zero?

Maximize margin



Minimize violations
of the margin



$$\min_{\mathbf{w}, b} \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_i \xi_i$$

$$s. t. \quad y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 - \xi_i \quad \forall i$$

$$\xi_i \geq 0$$



Allow some
violations

A

The soft-margin SVM reverts to the hard-margin SVM formulation.

B

The margin gets infinitely big.

C

Errors aren't penalized.

D

Errors are very heavily penalized.



What happens as if the **slack penalty** C goes to infinity?

Maximize margin



$$\min_{\mathbf{w}, b} \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_i \xi_i$$

Minimize violations
of the margin



$$s. t. \quad y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 - \xi_i \quad \forall i$$

$$\xi_i \geq 0$$

Allow some
violations



A

The soft-margin SVM reverts to the hard-margin SVM formulation.

B

The margin gets infinitely big.

C

Errors aren't penalized.

D

Errors are very heavily penalized.



How do we pick the value of the **slack penalty** C ?

A Just go with your first guess.

B Whatever value of C works best on the test data

C Cross-validation or evaluation on held-out validation data

D Tune it until performance on training data gets good enough.



Soft-Margin Support Vector Machine - Dual Formulation

Just like the hard-margin case, we can derive the Lagrangian dual formulation. I'll spare you the bloody details this time and just write it below:

Soft-Margin
SVM Dual:

$$\max_{\alpha_i} \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_i^n \sum_j^n \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j$$

$$\begin{aligned} s. t. \quad & \sum_{i=1}^n \alpha_i y_i = 0 \\ & 0 \leq \alpha_i \leq C \quad \forall i \end{aligned}$$

Has a very, very similar form to the hard-margin dual except the addition of an upper bound on α 's. Can still be solved by passing it to a Quadratic Programming solver.



Just like the hard-margin case, this derivation leads us to the optimal weights being:

$$\mathbf{w}^* = \sum_{i=1}^n \alpha_i^* y_i \mathbf{x}_i$$

And from the KKT conditions we know only a sparse set of training examples have non-zero α :

$$\alpha_i^* (y_i (w^{*T} x_i + b) - 1 + \xi_i) = 0, \quad \forall i \quad \leftarrow \text{Complementary slackness}$$

Only training points within (or on the edge of) the margin are support vectors and effect the solution.

Hard-Margin SVM

Hard-Margin
SVM Primal:

$$\min_{\mathbf{w}, b} \frac{1}{2} \mathbf{w}^T \mathbf{w}$$
$$s. t. \quad y_i (\mathbf{w}^T \mathbf{x}_i + b) \geq 1 \quad \forall i$$

Hard-Margin
SVM Dual:

$$\max_{\alpha_i} \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j$$
$$s. t. \quad \sum_{i=1}^n \alpha_i y_i = 0$$
$$0 \leq \alpha_i \quad \forall i$$

Given optimal alphas - weight vector is:

$$\mathbf{w}^* = \sum_{i=1}^n \alpha_i^* y_i \mathbf{x}_i$$

Soft-Margin SVM

Soft-Margin
SVM Primal:

$$\min_{\mathbf{w}, b} \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_i \xi_i$$
$$s. t. \quad y_i (\mathbf{w}^T \mathbf{x}_i + b) \geq 1 - \xi_i \quad \forall i$$
$$\xi_i \geq 0$$

Soft-Margin
SVM Dual:

$$\max_{\alpha_i} \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j$$
$$s. t. \quad \sum_{i=1}^n \alpha_i y_i = 0$$
$$0 \leq \alpha_i \leq C \quad \forall i$$

Given optimal alphas - weight vector is:

$$\mathbf{w}^* = \sum_{i=1}^n \alpha_i^* y_i \mathbf{x}_i$$

A Note About Solving this Equation

To plug this into a quadratic program solver, we need to get it into a canonical form

$$\text{Let } \boldsymbol{\alpha} = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix}$$

$$\max_{\alpha_i} \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_i^n \sum_j^n \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j$$

$$\begin{aligned} \text{s.t. } & \sum_{i=1}^n \alpha_i y_i = 0 \\ & 0 \leq \alpha_i \leq C \quad \forall i \end{aligned}$$



$$\begin{aligned} & \underset{\boldsymbol{\alpha}}{\text{minimize}} && \frac{1}{2} \boldsymbol{\alpha}^T P \boldsymbol{\alpha} + \mathbf{q}^T \boldsymbol{\alpha} \\ & \text{subject to} && A \boldsymbol{\alpha} = \mathbf{h} \\ & && \mathbf{b}_l \leq \boldsymbol{\alpha} \leq \mathbf{b}_u \quad \forall i \end{aligned}$$

where P, q, A, h, b_l , and b_u define a quadratic program with linear equality constraints.

A Note About Solving this Equation

To plug this into a quadratic program solver, we need to get it into a canonical form Let $\boldsymbol{\alpha} = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix}$

$$\max_{\alpha_i} \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_i^n \sum_j^n \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j$$

$$\begin{aligned} \text{s.t. } & \sum_{i=1}^n \alpha_i y_i = 0 \\ & 0 \leq \alpha_i \leq C \quad \forall i \end{aligned}$$



$$\underset{\boldsymbol{\alpha}}{\text{minimize}} \quad \frac{1}{2} \boldsymbol{\alpha}^T \begin{bmatrix} y_1 y_1 \mathbf{x}_1^T \mathbf{x}_1 & \cdots & y_1 y_n \mathbf{x}_1^T \mathbf{x}_n \\ \vdots & \ddots & \vdots \\ y_n y_1 \mathbf{x}_n^T \mathbf{x}_1 & \cdots & y_n y_n \mathbf{x}_n^T \mathbf{x}_n \end{bmatrix} \boldsymbol{\alpha} - \vec{\mathbf{1}}^T \boldsymbol{\alpha}$$

$$\begin{aligned} \text{subject to } & \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}^T \boldsymbol{\alpha} = 0 \\ & \vec{\mathbf{0}} \leq \boldsymbol{\alpha} \leq \vec{\mathbf{1}} C \quad \forall i \end{aligned}$$

A Note About Solving this Equation

I've implemented this and passed it to a QP solver to make my own SVM implementation in the Colab notebook below. Let's walk through this a bit.

$$\underset{\alpha}{\text{minimize}} \quad \frac{1}{2} \alpha^T \begin{bmatrix} y_1 y_1 x_1^T x_1 & \cdots & y_1 y_n x_1^T x_n \\ \vdots & \ddots & \vdots \\ y_n y_1 x_n^T x_1 & \cdots & y_n y_n x_n^T x_n \end{bmatrix} \alpha - \vec{\mathbf{1}}^T \alpha$$

$$\text{subject to} \quad \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}^T \alpha = 0$$

$$\vec{\mathbf{0}} \leq \alpha \leq \vec{\mathbf{1}} C \quad \forall i$$

https://colab.research.google.com/drive/1MznKB-UxlvIkYtG_vBon1Qg_jKx-sb3o?usp=sharing

Soft-Margin Support Vector Machines (SVMs) find the decision boundary that maximizes the margin between the positive and negative points while minimizing errors.

Can handle non-linearly separable datasets by defining a penalty for margin violations / errors.

The tradeoff between margin width and minimizing errors is controlled by a new hyperparameter C .

The soft-margin case has a very similar dual formulation as the hard-margin case and many of the same results hold. Both soft- and hard-margin SVM duals can be solved with QP solvers.

Today's Learning Objectives

Be able to answer:

- ~~How can we handle non-linearly separable data with a soft-margin Support Vector Machine?~~
 - ~~What is the interpretation of a support vector now?~~
 - ~~How can we implement one of these things with a QP solver?~~
-
- What is the kernel trick and why is it useful?
 - What is the definition of a kernel?
 - What is a kernel SVM?
-
- What else can we kernelize?
 - How do you kernelize perceptron?
 - How do you kernelize linear regression?





Who remembers learning non-linear functions with *ordinary least-squares linear regression* by first representing the input data via *non-linear basis* functions?

A I do.

B I sort of do.

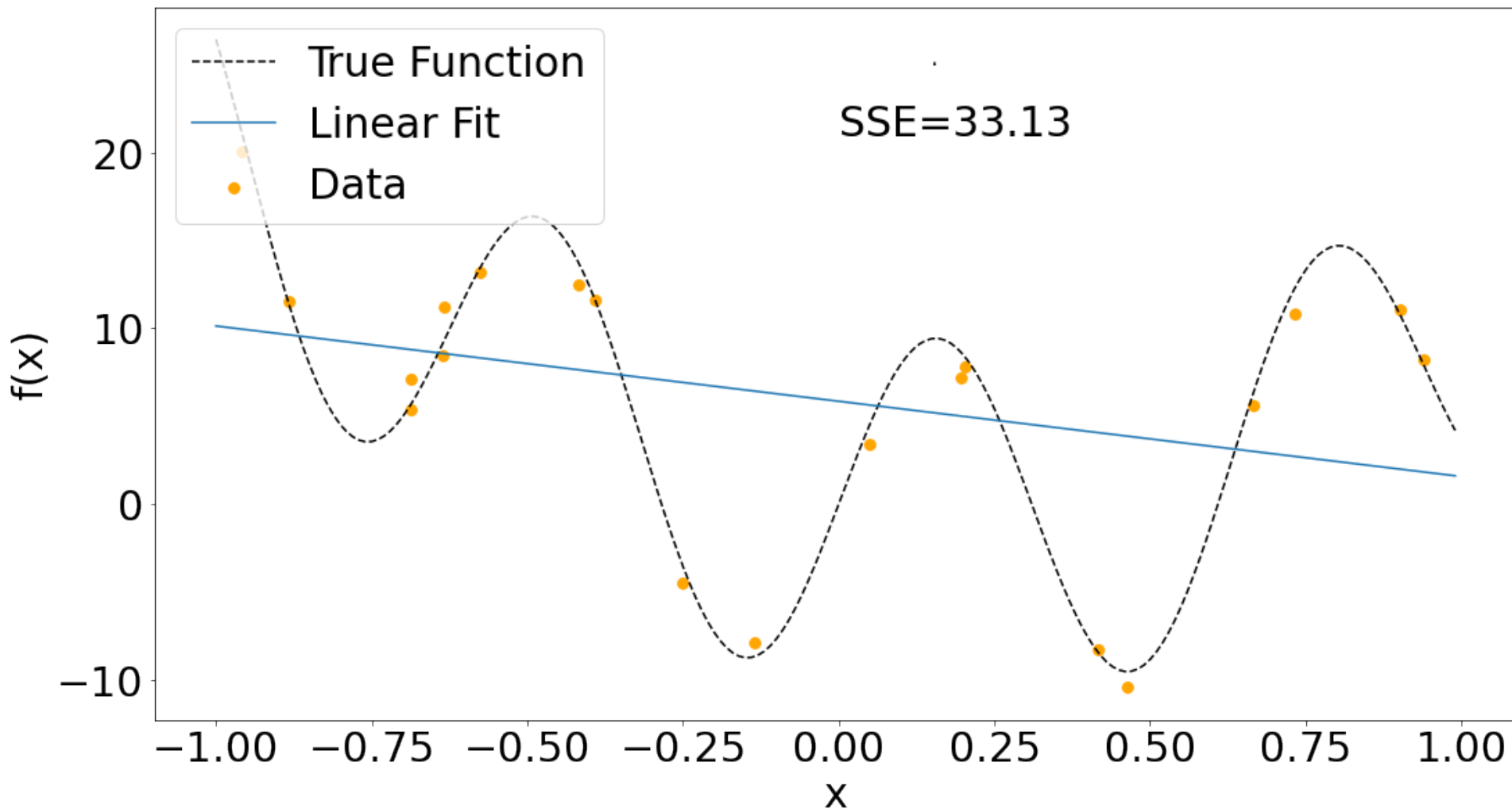
C I don't think I do.

D I definitely don't.



Back when we were looking at linear regression, we said:

Fitting lines seems cool but many, many functions of interest aren't lines.

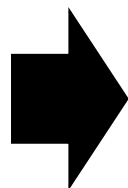




A Quick Reminder About Something We Covered Before

Idea: Solve a linear regression problem in a feature space that is non-linear in the original input! For example, could add a x^2 term.

X		Y
	x	Y
1	-0.25	-4.47
1	0.9	11.08
1	0.46	-10.44
1	0.2	7.19
1	-0.69	7.08
1	-0.69	5.38
1	-0.88	11.55
1	0.73	10.85
1	0.2	7.85
1	0.42	-8.31



X'			Y
	x	x^2	Y
1	-0.25	0.06	-4.47
1	0.9	0.81	11.08
1	0.46	0.21	-10.44
1	0.2	0.04	7.19
1	-0.69	0.48	7.08
1	-0.69	0.48	5.38
1	-0.88	0.77	11.55
1	0.73	0.53	10.85
1	0.2	0.04	7.85
1	0.42	0.18	-8.31

Solve for w such that:

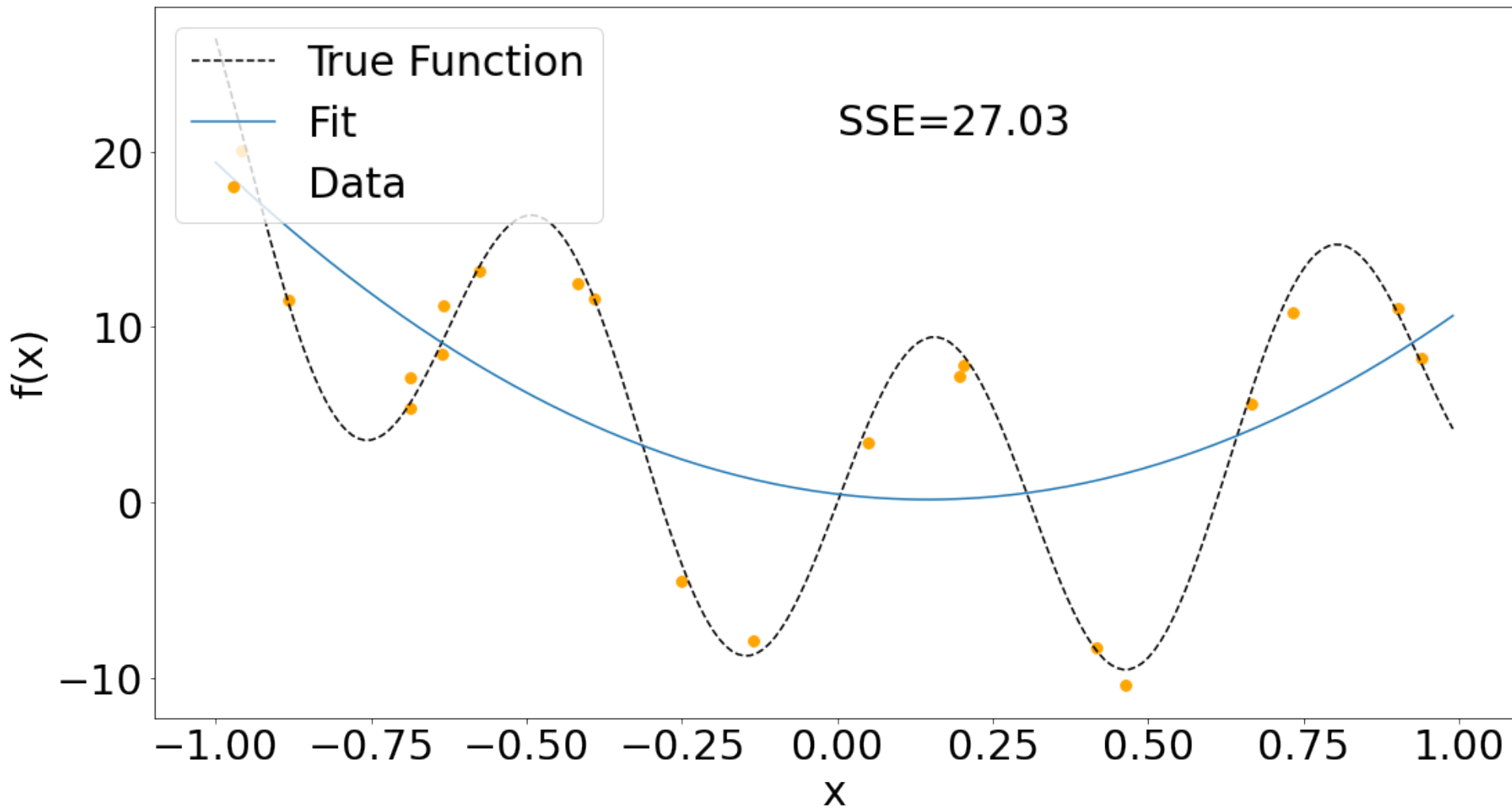
$$y_i = w_0 1 + w_1 x_i + w_2 x_i^2$$

Linear function of non-linear transformations of x

$$y_i = [1, x_i, x_i^2] \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}$$



A Quick Reminder About Something We Covered Before

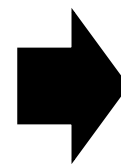




More general example: m-order polynomial regression in the 1d case.

$$\Phi(x) = \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^m \end{bmatrix}^T$$

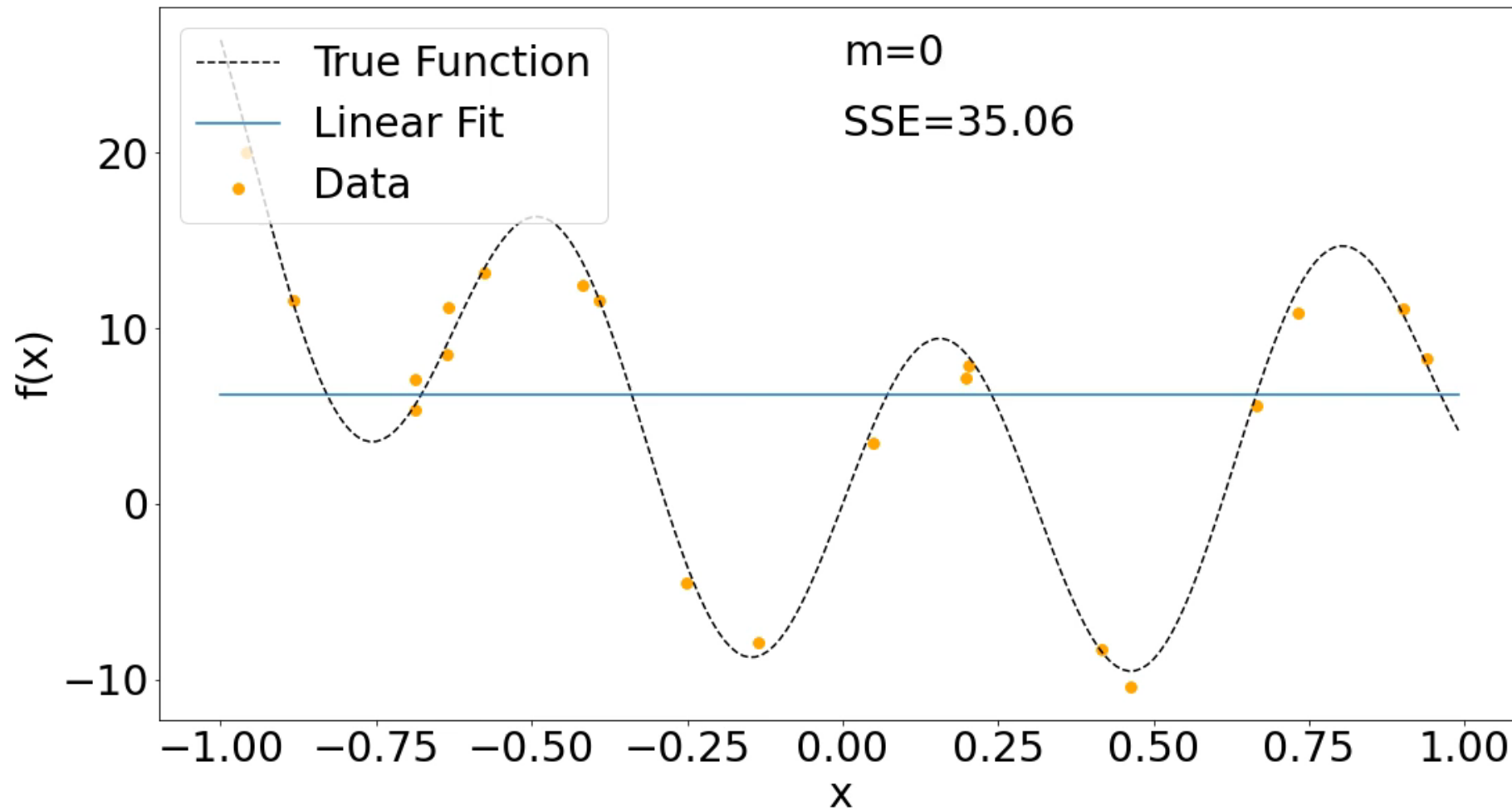
$$\mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}_{n \times 1}$$



$$\mathbf{X}' = \begin{bmatrix} \Phi(x_1) \\ \Phi(x_2) \\ \vdots \\ \Phi(x_n) \end{bmatrix}_{n \times m}$$



A Quick Reminder About Something We Covered Before



Number of data points $n=20$



What just happened?

We only had one **real** feature \mathbf{x} , but we transformed it into a vector of non-linear “derived” features $(\mathbf{1}, \mathbf{x}, \mathbf{x}^2, \mathbf{x}^3, \dots)$ or “basis functions”. Then we solved the linear regression problem in this non-linear feature space.

To predict y for a new point x , just do $\phi(x)w$!

What about when the input features are already multidimensional?



Let's generalize this further and consider basis functions.

Each data point has d features. Let $\Phi: \mathbb{R}^d \rightarrow \mathbb{R}^{d'}$ be our basis function. Can make any arbitrary choice we want here based on how the data looks.

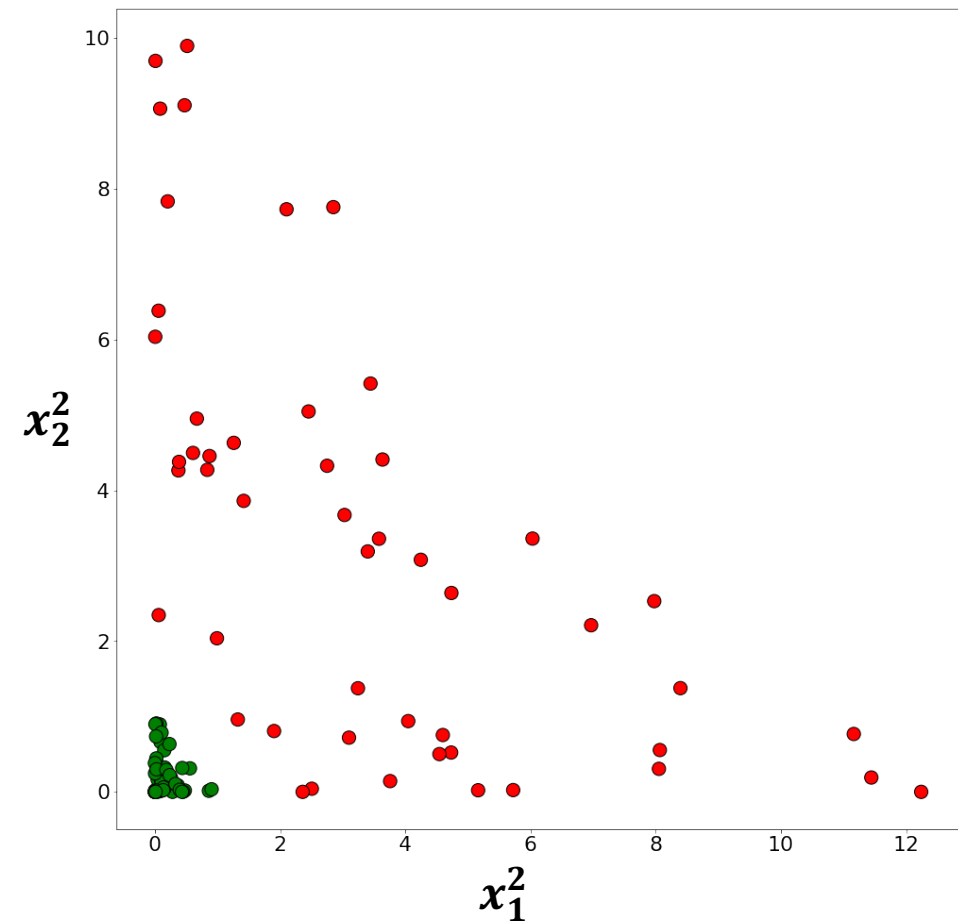
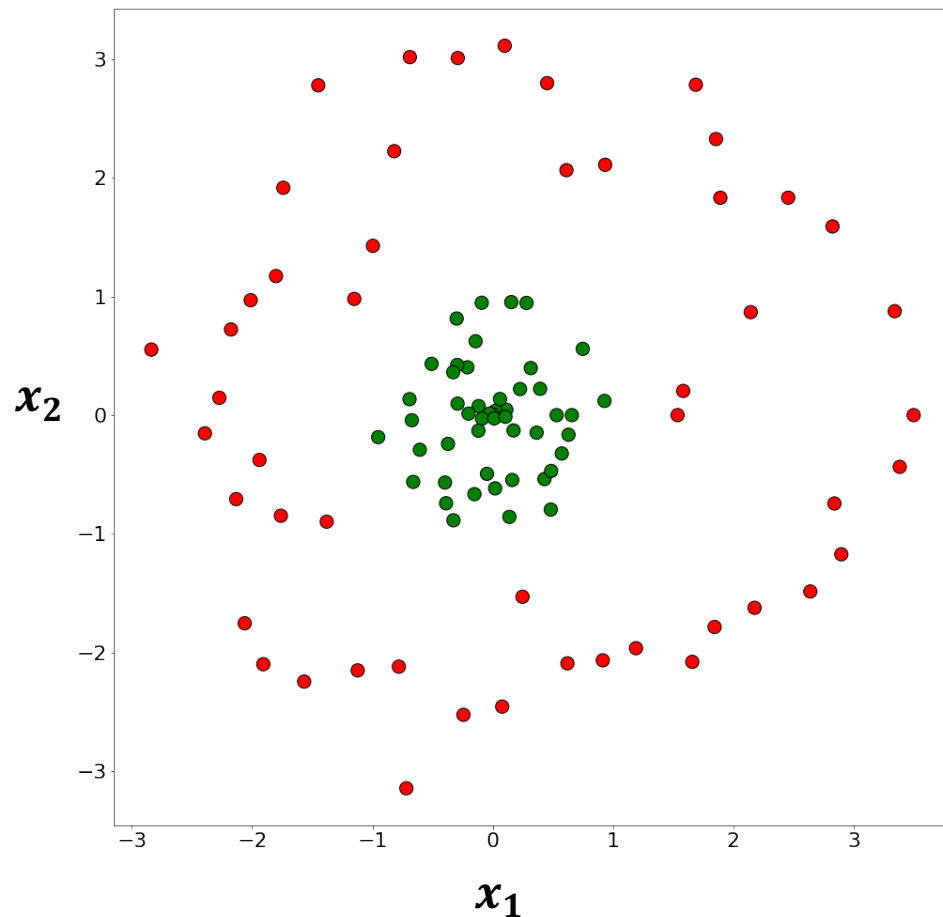
$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}_{1 \times d}^T \quad \Phi_A(\mathbf{x}) = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1 x_2 \\ \sin(x_1) \\ \cos(x_2) \end{bmatrix}^T \quad \Phi_B(\mathbf{x}) = \begin{bmatrix} 1 \\ x_1 x_2 \\ x_1 x_3 \\ x_1 x_4 \\ \vdots \\ x_n x_n \end{bmatrix}^T \quad \Phi_C(\mathbf{x}) = \begin{bmatrix} 1 \\ \prod x_i \\ \sqrt{x_1} \end{bmatrix}^T$$

No matter what we choose, the procedure is the same. Replace each \mathbf{x} (row) in our data matrix with $\Phi(\mathbf{x})$, then solve the linear regression problem.



Can we apply this to learn non-linear decision boundaries with linear classifiers?

That is, can we transform our input features non-linearly and then learn a linear boundary in that transformed space - could be very useful for examples below.



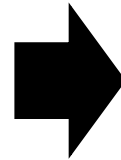


What would this look like in all the classifiers we've derived so far?

1. Define some transform $\Phi(x)$ that maps the vector x to a new vector
 - For example $\Phi([x_1, x_2]) = [x_1^2, x_2^2]$ or with real values $\Phi([2, -3]) = [4, 9]$
2. Replace \mathbf{x}_i with $\Phi(\mathbf{x}_i)$ in all our equations / derivations.
 - This doesn't change any of the derivations.

Soft-Margin
SVM Dual:

$$\begin{aligned} \max_{\alpha_i} \quad & \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \\ \text{s.t.} \quad & \sum_{i=1}^n \alpha_i y_i = 0 \\ & 0 \leq \alpha_i \leq C \quad \forall i \end{aligned}$$



$$\begin{aligned} \max_{\alpha_i} \quad & \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \Phi(\mathbf{x}_i)^T \Phi(\mathbf{x}_j) \\ \text{s.t.} \quad & \sum_{i=1}^n \alpha_i y_i = 0 \\ & 0 \leq \alpha_i \leq C \quad \forall i \end{aligned}$$



Applying Basis functions to the SVM dual formulation and solution:

Soft-Margin
SVM Dual:

$$\begin{aligned} \max_{\alpha_i} \quad & \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \Phi(\mathbf{x}_i)^T \Phi(\mathbf{x}_j) \\ \text{s. t.} \quad & \sum_{i=1}^n \alpha_i y_i = 0 \\ & 0 \leq \alpha_i \leq C \quad \forall i \end{aligned}$$


Soft-Margin
SVM Prediction:

$$\mathbf{w}^* = \sum_{i=1}^n \alpha_i^* y_i \Phi(\mathbf{x}_i) \quad \Rightarrow \quad \mathbf{w}^T \Phi(\mathbf{x}) + b = b + \sum_{i=1}^n \alpha_i^* y_i \Phi(\mathbf{x}_i)^T \Phi(\mathbf{x})$$



Applying Basis functions to the SVM dual formulation and solution:

Soft-Margin
SVM Dual:

$$\begin{aligned} \max_{\alpha_i} \quad & \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \Phi(\mathbf{x}_i)^T \Phi(\mathbf{x}_j) \\ \text{s. t.} \quad & \sum_{i=1}^n \alpha_i y_i = 0 \\ & 0 \leq \alpha_i \leq C \quad \forall i \end{aligned}$$


Seems like all that is going on
here is computing similarity in
some transformed feature space



Soft-Margin
SVM Prediction:

$$\mathbf{w}^* = \sum_{i=1}^n \alpha_i^* y_i \Phi(\mathbf{x}_i) \quad \Rightarrow \quad \mathbf{w}^T \Phi(\mathbf{x}) + b = b + \sum_{i=1}^n \alpha_i^* y_i \Phi(\mathbf{x}_i)^T \Phi(\mathbf{x})$$

Example: Quadratic Feature Space

$$\Phi(\mathbf{x}) = \begin{bmatrix} 1 \\ \sqrt{2}x_1 \\ \sqrt{2}x_2 \\ \vdots \\ \sqrt{2}x_d \\ x_1^2 \\ x_2^2 \\ \vdots \\ x_d^2 \\ \sqrt{2}x_1x_2 \\ \sqrt{2}x_1x_3 \\ \vdots \\ \sqrt{2}x_1x_d \\ \sqrt{2}x_2x_3 \\ \vdots \\ \sqrt{2}x_2x_d \\ \vdots \\ \sqrt{2}x_{d-1}x_d \end{bmatrix}$$

Constant Term

Linear Terms

Pure Quadratic Terms

Quadratic Cross-Terms

- **Quadratic Feature Space:** Assume d input dimensions $\mathbf{x} = [x_1, \dots, x_d]^T$. Can define a quadratic feature space transform as on the left.
- **Why use a quadratic feature space?** If you believe multiplicative interactions of your features are important.
- **How many terms are there in this representation?**
 - $1 + d + d + \frac{d(d-1)}{2} \approx O(d^2)$
- **What about higher order - i.e. cubic?**
 - For cubic space: $\approx O(d^3)$ terms.
 - Number of dimensions after the transform increased very rapidly with dimension.



$$\Phi(\mathbf{x}) = \begin{bmatrix} 1 \\ \sqrt{2}x_1 \\ \sqrt{2}x_2 \\ \vdots \\ \sqrt{2}x_d \\ x_1^2 \\ x_2^2 \\ \vdots \\ x_d^2 \\ \sqrt{2}x_1x_2 \\ \sqrt{2}x_1x_3 \\ \vdots \\ \sqrt{2}x_1x_d \\ \sqrt{2}x_2x_3 \\ \vdots \\ \sqrt{2}x_2x_d \\ \vdots \\ \sqrt{2}x_{d-1}x_d \end{bmatrix}$$

} Constant Term
} Linear Terms
} Pure Quadratic Terms
} Quadratic Cross-Terms

Pro: Transforming our data to higher dimensions / richer representation enables non-linear decision boundaries.

Con: The amount of computation to process these transforms and then operate on the resulting high-dimensional representations scales up quickly and can become a problem!

- For example in HW1, if we had applied a quadratic transform, our data would have had over 7000 feature dimensions.

What if there was a way to compute the similarity in this high dimensional space *without actually transforming the points*?



Dot Product in Quadratic Feature Space

$$\Phi(\mathbf{a}) \cdot \Phi(\mathbf{b}) = \begin{bmatrix} 1 \\ \sqrt{2}a_1 \\ \sqrt{2}a_2 \\ \vdots \\ \sqrt{2}a_d \\ a_1^2 \\ a_2^2 \\ \vdots \\ a_d^2 \\ \sqrt{2}a_1a_2 \\ \sqrt{2}a_1a_3 \\ \vdots \\ \sqrt{2}a_1a_d \\ \sqrt{2}a_2a_3 \\ \vdots \\ \sqrt{2}a_2a_d \\ \vdots \\ \sqrt{2}a_{d-1}a_d \end{bmatrix} \cdot \begin{bmatrix} 1 \\ \sqrt{2}b_1 \\ \sqrt{2}b_2 \\ \vdots \\ \sqrt{2}b_d \\ b_1^2 \\ b_2^2 \\ \vdots \\ b_d^2 \\ \sqrt{2}b_1b_2 \\ \sqrt{2}b_1b_3 \\ \vdots \\ \sqrt{2}b_1b_d \\ \sqrt{2}b_2b_3 \\ \vdots \\ \sqrt{2}b_2b_d \\ \vdots \\ \sqrt{2}b_{d-1}b_d \end{bmatrix}$$

$\left. \begin{matrix} 1 \\ \sqrt{2}b_1 \\ \sqrt{2}b_2 \\ \vdots \\ \sqrt{2}b_d \end{matrix} \right\} + \sum_{i=1}^d 2a_i b_i$
 $\left. \begin{matrix} b_1^2 \\ b_2^2 \\ \vdots \\ b_d^2 \end{matrix} \right\} + \sum_{i=1}^d a_i^2 b_i^2$
 $+ \sum_{i=1}^d \sum_{j=i+1}^d 2a_i a_j b_i b_j$

$$\Phi(\mathbf{a}) \cdot \Phi(\mathbf{b}) = 1 + 2 \sum_{i=1}^d a_i b_i + \sum_{i=1}^d a_i^2 b_i^2 + \sum_{i=1}^d \sum_{j=i+1}^d 2a_i a_j b_i b_j$$

Consider the following function: $(\mathbf{a} \cdot \mathbf{b} + 1)^2$

$$(\mathbf{a} \cdot \mathbf{b} + 1)^2 = 1 + 2\mathbf{a} \cdot \mathbf{b} + (\mathbf{a} \cdot \mathbf{b})^2$$

$$= 1 + 2 \left(\sum_{i=1}^d a_i b_i \right) + \left(\sum_{i=1}^d a_i b_i \right)^2$$

$$= 1 + 2 \sum_{i=1}^d a_i b_i + \sum_{i=1}^d \sum_{j=1}^d 2a_i a_j b_i b_j$$

$$= 1 + 2 \sum_{i=1}^d a_i b_i + \sum_{i=1}^d a_i^2 b_i^2 + \sum_{i=1}^d \sum_{j=i+1}^d 2a_i a_j b_i b_j$$

$$(\mathbf{a} \cdot \mathbf{b} + 1)^2 = \Phi(\mathbf{a}) \cdot \Phi(\mathbf{b})$$



Which is more efficient to compute for the quadratic feature representation $\Phi(\cdot)$ for $a, b \in \mathbb{R}^d$?

$$\Phi(\mathbf{a}) \cdot \Phi(\mathbf{b})$$

Dot product in transformed space

$$(\mathbf{a} \cdot \mathbf{b} + 1)^2$$

The function we just talked about

A They are equally efficient - both taking $O(d^2)$ operations.

B The dot product in the transformed space is cheaper - taking $O(d)$ operations.

C The function we just talked about is more efficient - taking $O(d^2)$ operations.

D The function we just talked about is more efficient - taking $O(d)$ operations.



Dot Product in Quadratic Feature Space as a Kernel Function

$$\Phi(\mathbf{a}) \cdot \Phi(\mathbf{b}) = \begin{bmatrix} 1 \\ \sqrt{2}a_1 \\ \sqrt{2}a_2 \\ \vdots \\ \sqrt{2}a_d \\ a_1^2 \\ a_2^2 \\ \vdots \\ a_d^2 \\ \sqrt{2}a_1a_2 \\ \sqrt{2}a_1a_3 \\ \vdots \\ \sqrt{2}a_1a_d \\ \sqrt{2}a_2a_3 \\ \vdots \\ \sqrt{2}a_2a_d \\ \vdots \\ \sqrt{2}a_{d-1}a_d \end{bmatrix} \cdot \begin{bmatrix} 1 \\ \sqrt{2}b_1 \\ \sqrt{2}b_2 \\ \vdots \\ \sqrt{2}b_d \\ b_1^2 \\ b_2^2 \\ \vdots \\ b_d^2 \\ \sqrt{2}b_1b_2 \\ \sqrt{2}b_1b_3 \\ \vdots \\ \sqrt{2}b_1b_d \\ \sqrt{2}b_2b_3 \\ \vdots \\ \sqrt{2}b_2b_d \\ \vdots \\ \sqrt{2}b_{d-1}b_d \end{bmatrix} = \underbrace{1 + \sum_{i=1}^d 2a_i b_i}_{\text{Linear terms}} + \underbrace{\sum_{i=1}^d a_i^2 b_i^2}_{\text{Quadratic terms}} + \underbrace{\sum_{i=1}^d \sum_{j=i+1}^d 2a_i a_j b_i b_j}_{\text{Cross terms}}$$

$O(d^3)$ computations required for the explicit transform-then-dot-product approach.

Consider the following function: $(\mathbf{a} \cdot \mathbf{b} + 1)^2$

$$(\mathbf{a} \cdot \mathbf{b} + 1)^2 = 1 + 2\mathbf{a} \cdot \mathbf{b} + (\mathbf{a} \cdot \mathbf{b})^2$$

$$= 1 + 2 \left(\sum_{i=1}^d a_i b_i \right) + \left(\sum_{i=1}^d a_i b_i \right)^2$$

$$= 1 + 2 \sum_{i=1}^d a_i b_i + \sum_{i=1}^d \sum_{j=1}^d 2a_i a_j b_i b_j$$

$$= 1 + 2 \sum_{i=1}^d a_i b_i + \sum_{i=1}^d a_i^2 b_i^2 + \sum_{i=1}^d \sum_{j=i+1}^d 2a_i a_j b_i b_j$$

$$(\mathbf{a} \cdot \mathbf{b} + 1)^2 = \Phi(\mathbf{a}) \cdot \Phi(\mathbf{b})$$

$O(d)$ computations required for applying this *kernel* function

$$\kappa(\mathbf{a}, \mathbf{b}) = (\mathbf{a} \cdot \mathbf{b} + 1)^2$$

Definition: A function $\kappa(\mathbf{a}, \mathbf{b})$ is called a kernel function if $\kappa(\mathbf{a}, \mathbf{b}) = \Phi(\mathbf{a}) \cdot \Phi(\mathbf{b})$ for some mapping function Φ .

- Can be intuitively thought of as computing some similarity between \mathbf{a} and \mathbf{b} .
- Not all mappings you might propose have a corresponding kernel function and not every kernel you might propose has a corresponding mapping.

However, for many popular mapping functions we might be interested in, we can find a corresponding kernel function that efficiently computes similarity.

Why we care? We can compute similarity in the high-dimensional mapping space without actually transforming the data into that space – gaining significant computational speedup. Call doing this the kernel trick.



What functions are kernels?

Consider a finite set of n input vectors $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n$, we define the kernel (gram) matrix composed of all pair-wise kernel values:

$$K = \begin{bmatrix} \kappa(\mathbf{x}_1, \mathbf{x}_1) & \kappa(\mathbf{x}_1, \mathbf{x}_2) & \dots & \kappa(\mathbf{x}_1, \mathbf{x}_n) \\ \kappa(\mathbf{x}_2, \mathbf{x}_1) & \kappa(\mathbf{x}_2, \mathbf{x}_2) & \dots & \kappa(\mathbf{x}_2, \mathbf{x}_n) \\ \vdots & \vdots & \ddots & \vdots \\ \kappa(\mathbf{x}_n, \mathbf{x}_1) & \kappa(\mathbf{x}_n, \mathbf{x}_2) & \dots & \kappa(\mathbf{x}_n, \mathbf{x}_n) \end{bmatrix}$$

Kernel matrices are square and symmetric by definition.

Mercer Theorem: A function κ is a kernel function if and only if for any finite sample $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n$, its corresponding kernel matrix is positive semidefinite (aka has non-negative eigenvalues).

Closure Properties of Kernels

If κ_1 and κ_2 are kernel functions then the following are all also kernel functions:

- $\kappa(a, b) = \kappa_1(a, b) + \kappa_2(a, b)$
 - $\Phi = \Phi_1 + \Phi_2$
- $\kappa(a, b) = c\kappa_1(a, b)$, for $c > 0$
 - $\Phi = \sqrt{a}\Phi_1$
- $\kappa(a, b) = \kappa_1(a, b)\kappa_2(a, b)$
 - $\Phi_{ij} = \Phi_{1i}\Phi_{2j} \leftarrow$ full cross product

In practice, we directly use the kernel functions without explicitly stating the transformation Φ . Will show this next slide for SVMs.

Some simple kernels have very complicated corresponding transforms. For instance, the **radial basis function kernel** (below) corresponds to a dot product in infinite dimensional space.


$$\kappa(a, b) = e^{-\frac{||a-b||^2}{2\sigma^2}}$$

Proof is based on the identity of $e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$. See full proof [here](#).



For SVM, if we applied the mapping directly we would do the following.

Soft-Margin
SVM Dual:

$$\begin{aligned} \max_{\alpha_i} \quad & \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \Phi(\mathbf{x}_i)^T \Phi(\mathbf{x}_j) \\ \text{s. t.} \quad & \sum_{i=1}^n \alpha_i y_i = 0 \\ & 0 \leq \alpha_i \leq C \quad \forall i \end{aligned}$$


Seems like all that is going on
here is computing similarity in
some transformed feature space



Soft-Margin
SVM Prediction:

$$\mathbf{w}^* = \sum_{i=1}^n \alpha_i^* y_i \Phi(\mathbf{x}_i) \quad \Rightarrow \quad \mathbf{w}^T \Phi(\mathbf{x}) + b = b + \sum_{i=1}^n \alpha_i^* y_i \Phi(\mathbf{x}_i)^T \Phi(\mathbf{x})$$



To apply to SVMs, we just replace the feature projected dot product with a kernel!

Soft-Margin
SVM Dual:

$$\begin{aligned} \max_{\alpha_i} \quad & \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \kappa(\mathbf{x}_i, \mathbf{x}_j) \\ \text{s. t.} \quad & \sum_{i=1}^n \alpha_i y_i = 0 \\ & 0 \leq \alpha_i \leq C \quad \forall i \end{aligned}$$



Just use the similarity from the
kernel to do our learning /
prediction.



Soft-Margin
SVM Prediction:

$$\mathbf{w}^* = \sum_{i=1}^n \alpha_i^* y_i \Phi(\mathbf{x}_i) \quad \Rightarrow \quad \mathbf{w}^T \Phi(\mathbf{x}) + b = b + \sum_{i=1}^n \alpha_i^* y_i \kappa(\mathbf{x}_i, \mathbf{x}_j)$$

A Note About Solving this Equation

I've implemented this and passed it to a QP solver to make my own SVM implementation in the Colab notebook below. Let's walk through this a bit.

$$\begin{aligned} \underset{\alpha}{\text{minimize}} \quad & \frac{1}{2} \alpha^T \begin{bmatrix} y_1 y_1 \kappa(x_1, x_1) & \dots & y_1 y_n \kappa(x_1, x_n) \\ \vdots & \ddots & \vdots \\ y_n y_1 \kappa(x_n, x_1) & \dots & y_n y_n \kappa(x_n, x_n) \end{bmatrix} \alpha + \vec{\mathbf{1}}^T \alpha \\ \text{subject to} \quad & \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}^T \alpha = 0 \\ & \vec{\mathbf{0}} \leq \alpha \leq \vec{\mathbf{1}} C \quad \forall i \end{aligned}$$

https://colab.research.google.com/drive/1MznKB-UxlvIkYtG_vBon1Qg_jKx-sb3o?usp=sharing

Today's Learning Objectives

Be able to answer:

- ~~How can we handle non-linearly separable data with a soft-margin Support Vector Machine?~~
 - ~~What is the interpretation of a support vector now?~~
 - ~~How can we implement one of these things with a QP solver?~~
-
- ~~What is the kernel trick and why is it useful?~~
 - ~~What is the definition of a kernel?~~
 - ~~What is a kernel SVM?~~
-
- What else can we kernelize?
 - How do you kernelize perceptron?
 - How do you kernelize linear regression?



The kernel trick was the “deep learning” of the 1990s. Everything that could be kernelized was kernelized.

- Kernel Linear Regression
- Kernel Logistic Regression
- Kernel PCA
- ...

Across these cases, the commonality is expressing the objective via the dual formulation and replacing feature dot-products with kernel computations.

Lots of work on specialized kernels for different data types as well – e.g. graph kernels for when the input is a graph.



Recall from many lectures ago...

The Perceptron is a _____ that makes predictions as _____ and is trained by _____

A Linear regressor; $y = w^T x + b$; applying a closed form solution for the weights

B Non-Linear classifier; $y = \operatorname{argmax} P(x|y)P(y)$; fitting distribution parameter with MLE

C Linear classifier; $y = \sigma(w^T x + b)$; gradient descent

D Linear classifier; $y = \operatorname{sign}(w^T x + b)$; an iterative updating algorithm



Recall the Perceptron

Perceptron predicts $y_i = \text{sign}(w^T x_i + b)$
Trained with a simple iterative algorithm

Perceptron Learning Algorithm

$w = \text{zeros}(d)$

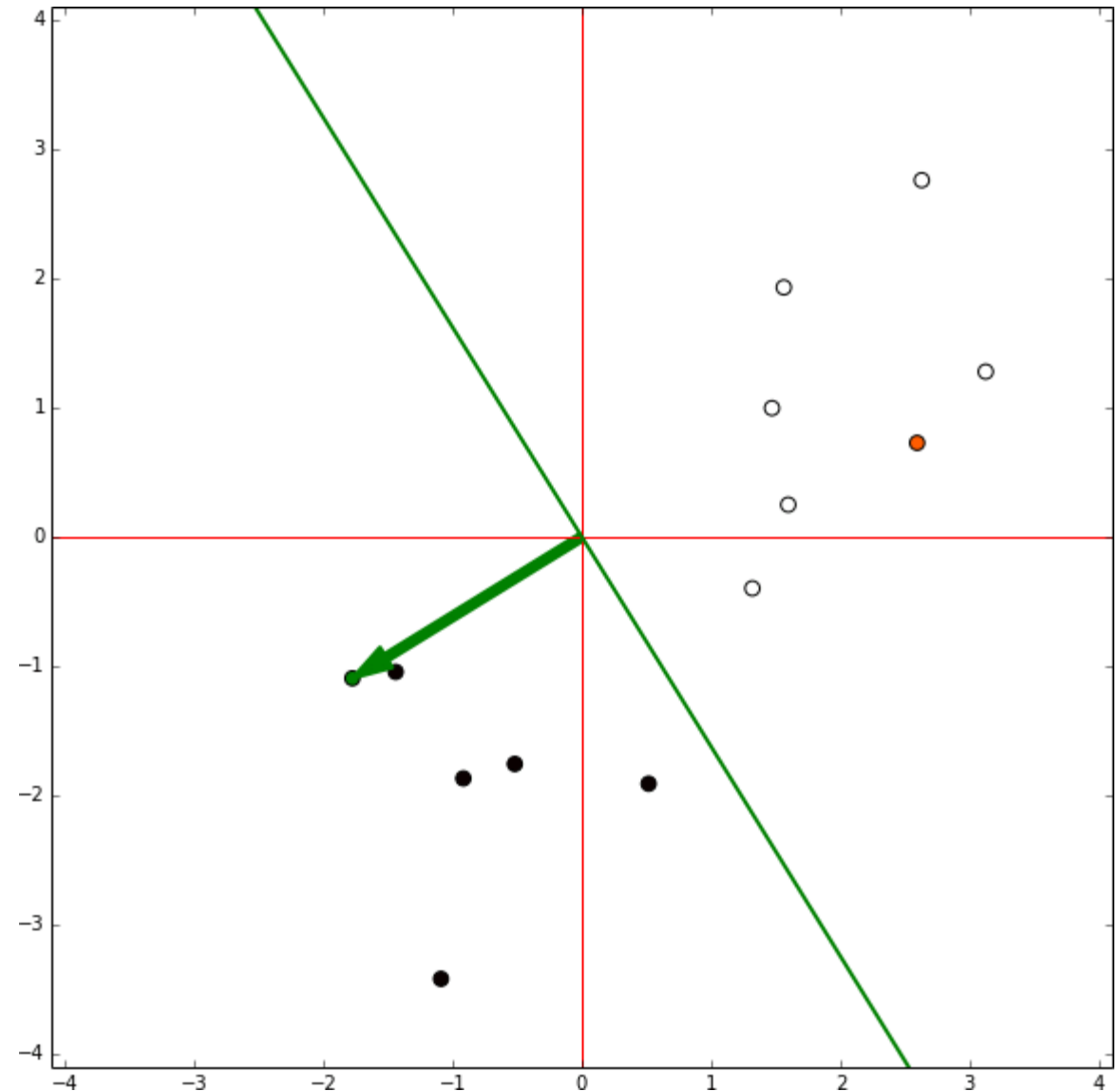
Repeat:

for each example $x_i, y_i \in D$:

if $y_i w_{t-1}^T x_i < 0$: // if misclassification

$$w_t = w_{t-1} + \alpha y_i x_i$$

Until no errors or max iterations





Can you see how to write the weights w_t (i.e. the weights after the t 'th error) as a function of our training data vectors?

$w = \text{zeros}(d)$

Repeat:

for each example $x_i, y_i \in D$:

if $y_i w_{t-1}^T x_i < 0$: // if misclassification

$$w_t = w_{t-1} + \alpha y_i x_i$$

Until no errors or max iterations



If we stare at this hard for a bit, we observe that w looks like a sum of our training instances...

Perceptron Learning Algorithm

$w = \text{zeros}(d)$

Repeat:

for each example $x_i, y_i \in D$:

if $y_i w_{t-1}^T x_i < 0$:

$$w_t = w_{t-1} + \alpha y_i x_i$$

Until no errors or max iterations

After the t th misclassification, the weight vector is:

$$w_t = \alpha \sum_{i \in M} y_i x_i$$

where M is a list of indices of misclassifications.



Let's modify this to keep track of how many times each example is errored on:

$w = \text{zeros}(d)$

$c = \text{zeros}(n)$

Repeat:

for each example $\mathbf{x}_i, y_i \in D$:

if $y_i \mathbf{w}_{t-1}^T \mathbf{x}_i < 0$:

$$\mathbf{w}_t = \mathbf{w}_{t-1} + \alpha y_i \mathbf{x}_i$$

$$c_i = c_i + 1$$

Until no errors or max iterations

$$\mathbf{w}_t = \alpha \sum_{i=1}^n c_i y_i \mathbf{x}_i$$



Let's plug in our definition of w_t and drop the update

$w = \text{zeros}(d)$

$c = \text{zeros}(n)$

Repeat:

for each example $\mathbf{x}_i, y_i \in D$:

if $y_i \alpha \sum_{j=1}^n c_j y_j \mathbf{x}_j^T \mathbf{x}_i < 0$:

$c_i = c_i + 1$

Until no errors or max iterations

$$\mathbf{w}_t = \alpha \sum_{i=1}^n c_i y_i \mathbf{x}_i$$



This only depends on dot products as well... kernelize it! Both training and prediction can be expressed as dot products between input vectors!

$w = \text{zeros}(d)$

$c = \text{zeros}(n)$

Repeat:

for each example $\mathbf{x}_i, y_i \in D$:

if $y_i \alpha \sum_{j=1}^n c_j y_j \kappa(\mathbf{x}_j, \mathbf{x}_i) < 0$:

$c_i = c_i + 1$

Until no errors or max iterations

$$\mathbf{w}_t = \alpha \sum_{i=1}^n c_i y_i \mathbf{x}_i$$

$$y = \text{sign}(b + \mathbf{w}^T \mathbf{x})$$



$$y = \text{sign}(b + \alpha \sum_{j=1}^n c_j y_j \kappa(\mathbf{x}_j, \mathbf{x}))$$



Can also kernelize linear regression (with L2 regularization):

Prediction:

Original model: $y = \mathbf{w}^T \mathbf{x}$

Kernel version: $y = \sum_{i=1}^n \alpha_i \kappa(\mathbf{x}_i, \mathbf{x})$

Objective:

Original SSE: $\|\mathbf{y} - X\mathbf{w}\| + \lambda\|\mathbf{w}\|$

Kernel version: $\|\mathbf{y} - \boldsymbol{\alpha}K\| + \lambda\boldsymbol{\alpha}^T K \boldsymbol{\alpha}$

K is the gram matrix

Solution:

Original version: $(X^T X + \lambda I)^{-1} X^T \mathbf{y}$

Kernel version: $(K + \lambda I)^{-1} \mathbf{y}$



Kernels are a general way to compute dot products between high-dimensional representations of data *without* ever actually having to represent that high-dimensional vector.

Not all similarity function are kernels, but many useful ones are. Common kernels are linear, polynomial, sigmoid, and radial basis functions. Many specialized kernels have been developed to measure similarity across specific data like graphs.

Kernels can be directly applied to SVMs and with a bit of work many other models.



Next Time: Review for the midterm!