CSE 417T: Introduction to Machine Learning

Final Review

Henry Chai

12/4/18

Overfitting

- Overfitting is fitting the training data “more than is warranted”
- Fitting noise rather than signal

Estimating E_{out} instead of E_{in}

$$E_{out}(h) = E_{in}(h) + \underbrace{\text{overfit penalty}}$$

regularization estimates this quantity

Regularization

- Constrain hypothesis sets to prevent them from being able to fit noise
- Learning algorithms are optimization problems and regularization imposes constraints on that optimization

Regularization

$$\text{minimize } E_{aug}(\vec{w}, \lambda_C) = E_{in}(\vec{w}) + \frac{\lambda_C}{n}\Omega(\vec{w})$$

Ridge: $\Omega(\vec{w}) = \sum_{j=0}^{d} w_j^2$

Low Order: $\Omega(\vec{w}) = \sum_{j=0}^{d} j w_j^2$

Lasso: $\Omega(\vec{w}) = \sum_{j=0}^{d} |w_j|$

Estimating E_{out} instead of E_{in}

$$\underbrace{E_{out}(h)}_{\text{validation estimates this quantity}} = E_{in}(h) + \text{overfit penalty}$$

Test sets

- Estimate $E_{out}(g)$ using the error on some test dataset t$\mathcal{D}_{test}$, $E_{test}(g)$
- If $\mathcal{D}_{test}$ is not involved in the training process, then $P\{|E_{test}(g) - E_{out}(g)| > \epsilon\} \leq 2e^{-2\epsilon^2 k}$ $(k = |\mathcal{D}_{test}|)$

Picking k

- More test data leads to a tighter bound on $E_{out}(g^-)$ but fewer training data **generally** means the learned g^- is worse i.e. $E_{out}(g^-)$ tends to increases as $n - k$ decreases

- $E_{out}(g) \leq E_{out}(g^-) \leq E_{test}(g^-) + O\left(\frac{1}{\sqrt{k}}\right)$ (with high probability)

- Return g but bound $E_{out}(g)$ using $E_{test}(g^-) + O\left(\frac{1}{\sqrt{k}}\right)$

- Practical rule of thumb: $k = \frac{n}{5}$

Test sets

- Estimate $E_{out}(g)$ using the error on some test dataset $\mathcal{D}_{test}$, $E_{test}(g)$

- If $\mathcal{D}_{test}$ is **not involved in the training process**, then $P\{|E_{test}(g) - E_{out}(g)| > \epsilon\} \leq 2e^{-2\epsilon^2 k}$ $(k = |\mathcal{D}_{test}|)$

Validation set

- $\mathcal{D}_{train}$ is used to build a finite set of candidate hypotheses: $\mathcal{H}_{val} = \{g_1^-, g_2^-, \dots, g_m^-\}$
- $\mathcal{D}_{val}$ is used to select the hypothesis from $\mathcal{H}_{val}$: $g_{m^*}^-$
- $P\left\{\left|E_{val}(g_{m^*}^-) - E_{out}(g_{m^*}^-)\right| > \epsilon\right\} \leq 2(m)e^{-2\epsilon^2 k}$
- $E_{val}(g_{m^*}^-) - O\left(\frac{\ln(m)}{\sqrt{k}}\right) \leq E_{out}(g_{m^*}^-) \leq E_{val}(g_{m^*}^-) + O\left(\sqrt{\frac{\ln(m)}{k}}\right)$

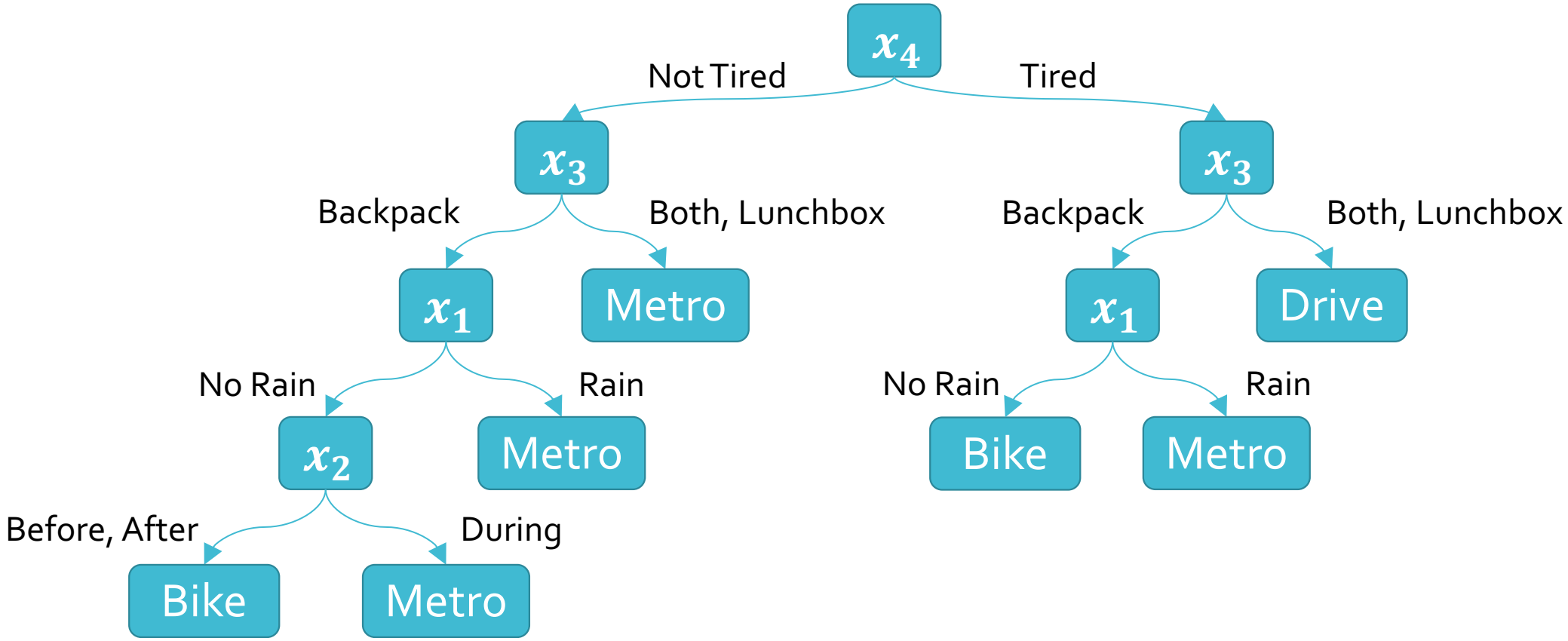
  with high probability

E_{in} vs. E_{val} vs. E_{test}

	Bias	Relationship to E_{out}
E_{in}	Incredibly biased	VC-bound
E_{val}	Slightly biased	Hoeffding's bound (multiple hypotheses)
E_{test}	Not biased	Hoeffding's bound (single hypothesis)

Three Learning Principles

- Occam's Razor
  - The simplest model that fits the data is also the most plausible
- Sampling Bias
  - If the data is sampled in a biased way, learning will produce a similarly biased outcome
- Data Snooping
  - If a data set has affected any step in the learning process, its ability to assess the outcome has been compromised



Decision Tree: Example

ID3 Learning Algorithm

- Initialize the tree as a single leaf that contains all labels
- While ∃ an impure leaf (not all labels are the same)
  - Pick an arbitrary impure leaf
  - Find the feature, x^*, with the largest information gain relative to the labels in that leaf
  - Create a child (or split) for each unique value of x^*
  - Assign each label in the original leaf to one of its children depending on its corresponding x^* value
    - The original leaf is no longer a leaf
    - All of its children are new leaves

Decision Tree / ID3 Pros

- Intuitive / explainable
- Can handle categorical and real-valued features
- Automatically performs feature selection
- The ID3 algorithm has a preference for shorter trees (simpler hypotheses)

Decision Tree / ID3 Cons

- The ID3 algorithm is greedy so no optimality guarantee
- Overfitting!
  - Heuristics ("regularization"):
    - Do not split leaves that are past a fixed depth δ or have fewer than c labels or where the maximal information gain is less than τ
  - Pruning ("validation"):
    - Evaluate each split using a validation set and remove the one that most improves the validation error

Bagging

- Short for **B**ootstrap **agg**regat**ing**
  - Combines the prediction of many "independent" hypotheses to reduce variance
- Bootstrapping:
  - A statistical method for estimating properties of a distribution, given (potentially a small number of) samples from that distribution
  - Relies on resampling the samples ***with replacement*** many, many times
- Aggregating:
  - Combining multiple hypotheses, $\{h_1, h_2, \dots, h_m\}$, to arrive at a single hypothesis

Split-Feature Randomization

- Predictions made by trees trained on similar datasets are highly correlated
- To decorrelate these predictions, randomly limit the features available at each iteration of the ID3 algorithm
- Every time the ID3 algorithm goes to split an impure leaf, randomly select $m < d$ features and only allow the algorithm to use one of those m features.
  - For classification, a common choice is $m = \sqrt{d}$
  - For regression, a common choice is $m = \frac{d}{3}$

Random Forests

- Input: $\mathcal{D} = \{(\overrightarrow{x_1}, y_1), (\overrightarrow{x_2}, y_2), \dots, (\overrightarrow{x_n}, y_n)\}$, B, m
- For $b = 1, 2, \dots, B$
  - Create a dataset, $\mathcal{D}_b$, by sampling n points from $\mathcal{D}$ with replacement
  - Learn a decision tree, t_b, using $\mathcal{D}_b$ and the ID3 algorithm ***with split-feature randomization***
- Output: $\bar{t}$, the aggregated hypothesis

Boosting

- Another ensemble method (like bagging) that combines the predictions of multiple hypotheses
- Aims to reduce the bias of a "weak" or highly biased hypothesis set (can also reduce variance)
- Intuition: iteratively reweight inputs, giving more weight to inputs that are difficult-to-predict correctly
- Fundamentally requires that we have access to weak learners that are better than random chance

AdaBoost

- Input: $\mathcal{D}$ ($\mathcal{Y} = \{-1, +1\}$), T
- Initialize input weights: $\omega_1^{(0)}, \ldots, \omega_n^{(0)} = \frac{1}{n}$
- For $t = 1, \ldots, T$
  1. Train a weak learner (hypothesis), h_t, by minimizing the weighted training error
  2. Compute the weighted training error of h_t:

$$\epsilon_t = \sum_{i=1}^{n} \omega_i^{(t-1)} [\![h_t(\vec{x_i}) \neq y_i]\!]$$

  3. Compute the "importance" of h_t:

$$\alpha_t = \frac{1}{2} \log\left(\frac{1 - \epsilon_t}{\epsilon_t}\right)$$

  4. Update the weights:

$$\omega_i^{(t)} = \frac{\omega_i^{(t-1)}}{Z_t} \times \begin{cases} e^{-\alpha_t} \text{ if } h_t(\vec{x_i}) = y_i \\ e^{\alpha_t} \text{ if } h_t(\vec{x_i}) \neq y_i \end{cases} =$$

- Output: an aggregated hypothesis

$$g_T(\vec{x}) = \text{sign}\left(H_T(\vec{x})\right)$$

$$= \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(\vec{x})\right)$$

Why AdaBoost?

1. If you only have access to weak learners …
2. … and want your final hypothesis to be a weighted combination of weak learners, …
3. … then Adaboost greedily minimizes the exponential loss:

$$e(h, f, \vec{x}) = e^{(-f(\vec{x})h(\vec{x}))}$$

1. Because of computational constraints
2. Because weak learners are not great on their own
3. Because the exponential loss upper bounds binary error

Nearest Neighbor Intuition

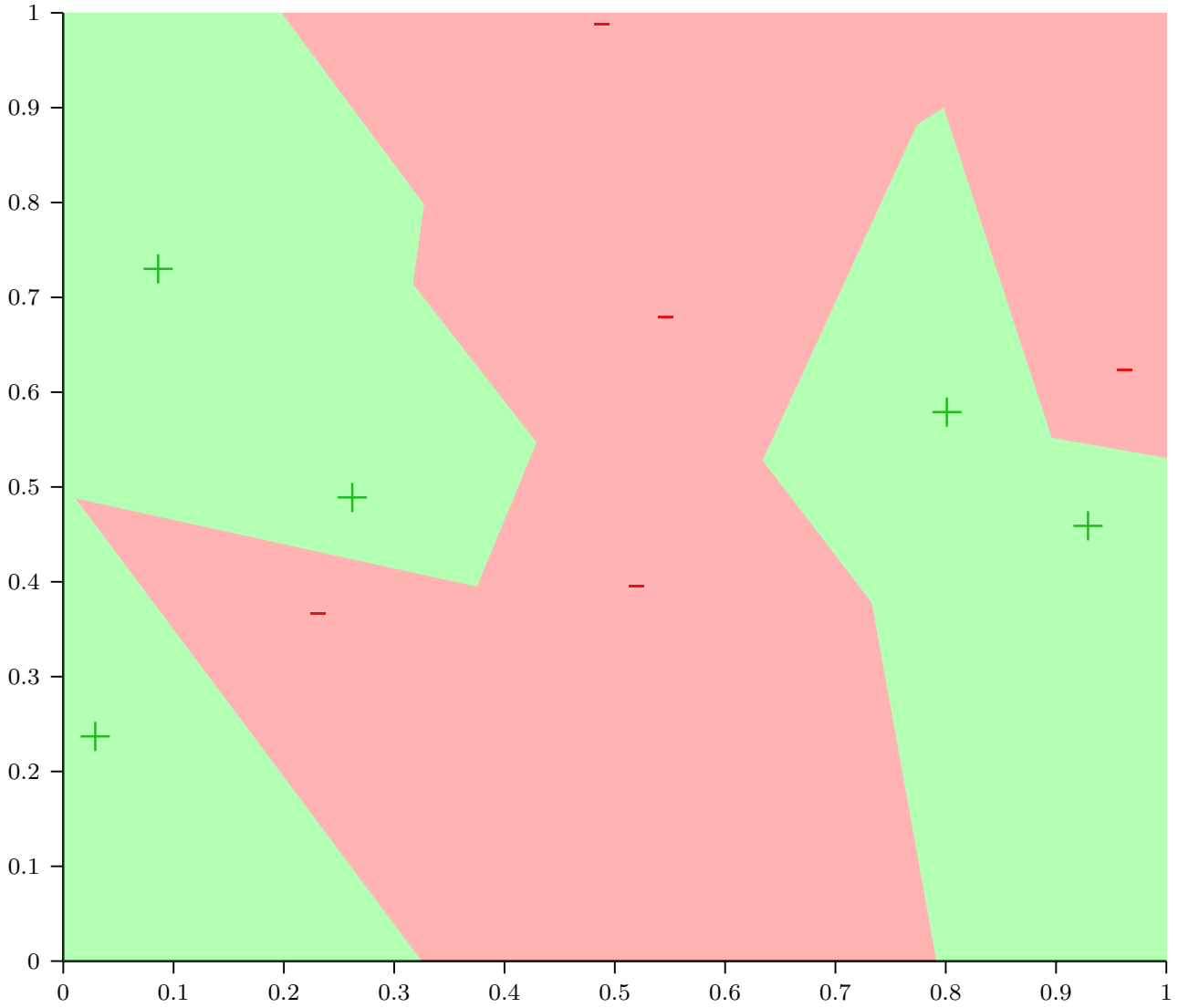
- Classify a point as the label of the "most similar" training point

- Use Euclidean distance as the similarity metric:

$$d(\vec{x}, \vec{x}') = \|\vec{x} - \vec{x}'\| = \sum_{j=1}^{D} \left(x_j - x_j'\right)^2$$

The Nearest Neighbor Hypothesis

$$g(\vec{x}) = y_{[1]}(\vec{x})$$



Generalization of Nearest Neighbor

- Claim: E_{out} for the nearest neighbor hypothesis is not much worse than the best possible E_{out}!
- Formally: with high probability, $E_{out}(g) \leq 2E_{out}(g^*)$ as $n \to \infty$
- Interpretation: half of the data's predictive power is in the nearest neighbor!

k-Nearest Neighbors (kNN)

- Classify a point as the most common label among the labels of the k nearest training points
- When $k = 1$, g is the nearest neighbor hypothesis
  - complicated decision boundaries; may overfit
- When $k = n$, g always predicts the most common label in the training dataset
  - no decision boundaries;may underfit
- k controls the complexity of the hypothesis set $\Longrightarrow$ k affects how well the learned hypothesis will generalize
- Practical rules of thumb:
  - $k = 3$
  - $k = \lfloor \sqrt{n} \rfloor$
  - Cross-validation

kNN Pros and Cons

- Pros:
  - Intuitive / explainable
  - No training / retraining
  - Provably near-optimal in terms of E_{out}
- Cons:
  - Computationally expensive
    - Always needs to store all data: $O(nD)$
    - Computing $g(\vec{x})$ requires computing $d(\vec{x}, \vec{x}') \; \forall \; \vec{x}' \in \mathcal{D}$ and finding the k closest points: $O(nD + n\log(k))$
  - Suffers from the "curse of dimensionality"

Curse of Dimensionality

- The fundamental assumption of kNN is that "similar" points or points close to one another should have the same label
- The closer two points are, the more confident we can be that they will have the same label
- As the number of dimensions the input has grows, the less likely it is that two random points will be close
- As the number of dimensions the input has grows, it takes more points to "cover" the input space

Curing the Curse of Dimensionality

- More data
- Fewer dimensions
- Blessing of non-uniformity: data from the real world is rarely uniformly distributed across the input space

Computational Cost of kNN

- No training required!
- Memory: $O(nD)$
- Computing $g(\vec{x})$: $O(nD + n\log(k))$

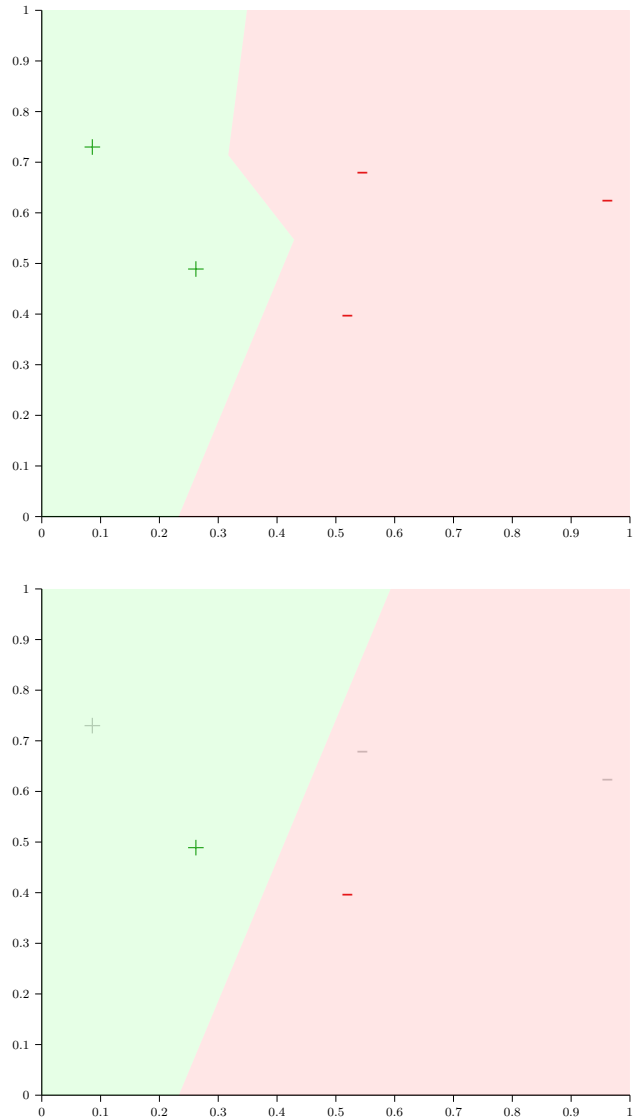
- Idea: preprocess inputs in order to speed up predictions
  - Reduce the number of inputs held in memory by eliminating redundancies
  - Organize inputs in data structures that make searching for nearest neighbors more efficient

Data Condensing

- Reduce the number of inputs while maintaining the same predictions on all inputs
- Let $g_{\mathcal{D}}$ be the kNN hypothesis when trained on $\mathcal{D}$
- $S \subseteq \mathcal{D}$ is training-set consistent if:

$$g_S(\overrightarrow{x_i}) = g_{\mathcal{D}}(\overrightarrow{x_i}) \; \forall \; \overrightarrow{x_i} \in \mathcal{D}$$

- Training-set consistent is a much weaker constraint than decision-boundary consistent



Organizing the Inputs

- Intuition: split the inputs into clusters, groups of points that are close to one another but far from other groups.
- If an input point is really close to one group of points and really far from the other groups …
- … then we can skip searching through the other groups and just look for nearest neighbors in the close group!

- We want cluster centers to be far apart and cluster radii to be small

Radial Basis Functions (RBF)

- kNN only considers some points and weights them equally
- RBFs consider all points but weight them unequally
  - Intuition: all points are useful but some points are more useful than others!
  - Bonus: no need to choose k

$$g(\vec{x}) = \text{sign}\left(\sum_{i=1}^{n}\left(\frac{e^{-\frac{\|\vec{x}-\vec{x_i}\|^2}{2r^2}}}{\sum_{i=1}^{n} e^{-\frac{\|\vec{x}-\vec{x_i}\|^2}{2r^2}}}\right) y_i\right)$$

Maximal Margin Linear Separators

- The margin of a separating hyperplane is the distance between the hyperplane and the nearest training point

- Questions:
  - How can we efficiently find a maximal-margin linear separator?
  - Why are linear separators with larger margins better?
  - What can we do if the data is not linearly separable?

Maximizing the Margin

$$\text{minimize} \quad \frac{1}{2}\vec{w}^T\vec{w}$$

$$\text{subject to} \quad y_i(\vec{w}^T\vec{x_i} + w_0) \geq 1 \; \forall(\vec{x_i}, y_i) \in \mathcal{D}$$

- This optimization problem to be solved (approximately) using quadratic programming (QP) in $O(D^3)$ time
- Let $\mathcal{H}_\rho$ = linear separators with minimum margin ρ. If the input space is a D-dimensional sphere of radius R, then:

$$d_{VC}(\mathcal{H}_\rho) \leq \min\left(D, \left\lceil \frac{R^2}{\rho^2} \right\rceil\right) + 1$$

Linearly Inseparable Data

- What can we do if the data is not linearly separable?
  - Accept some non-zero in-sample error
    - How much in-sample error should we tolerate?
  - Apply a non-linear transformation that shifts the data into a space where it is linearly separable
    - How can we pick a non-linear transformation?

Soft-Margin SVMs

$$\text{minimize} \quad \frac{1}{2}\vec{w}^T\vec{w} + C\sum_{i=1}^{n}\xi_i$$

$$\text{subject to} \quad y_i(\vec{w}^T\vec{x_i} + w_0) \geq 1 - \xi_i \;\forall(\vec{x_i}, y_i) \in \mathcal{D}$$

$$\xi_i \geq 0 \qquad \forall\, i \in \{1, \dots, n\}$$

- ξ_i is the "soft" error on the i^{th} training
  - If $\xi_i > 1$, then $y_i(\vec{w}^T\vec{x_i} + w_0) < 0 \Rightarrow (\vec{x_i}, y_i)$ is incorrectly classified
  - If $0 < \xi_i < 1$, then $y_i(\vec{w}^T\vec{x_i} + w_0) > 0 \Rightarrow (\vec{x_i}, y_i)$ is correctly classified but inside the margin
- $\sum_{i=1}^{n}\xi_i$ is the "soft" in-sample error

Nonlinear Dual SVMs

- Decide on a transformation $\Phi: \mathcal{X} \to \mathcal{Z}$

- Find a maximal-margin separating hyperplane in the transformed space, $[\vec{\tilde{w}}^*, \tilde{w}_0^*]$, by solving the QP:

$$\text{minimize} \quad \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\tilde{\alpha}_i\tilde{\alpha}_j y_i y_j \Phi(\vec{x_i})^T\Phi(\vec{x_j}) - \sum_{i=1}^{n}\tilde{\alpha}_i$$

$$\text{subject to} \sum_{i=1}^{n}\tilde{\alpha}_i y_i = 0$$

$$\tilde{\alpha}_i \geq 0 \; \forall \; i \in \{1, \ldots, n\}$$

- Return the corresponding predictor in the original space:

$$g(\vec{x}) = \text{sign}\left(\sum_{i:\alpha_i^* > 0}\tilde{\alpha}_i^* y_i \Phi(\vec{x_i})^T\Phi(\vec{x}) + \tilde{w}_0^*\right)$$

Perceptrons

	Low-Dimensional Input Space	High-Dimensional Input Space
E_{in}	High	Low
Generalization	Good	Bad

SVMs

	Low-Dimensional Input Space	High-Dimensional Input Space
E_{in}	High	Low
Generalization	Good	Okay

$$d_{VC}(\mathcal{H}) = D + 1 \text{ vs. } d_{VC}(\mathcal{H}_\rho) \leq \min\left(D, O\left(\frac{1}{\rho^2}\right)\right) + 1$$

Efficiency

- Depending on the transformation, Φ, and the dimensionality of the original input space. D, computing $\Phi(\vec{x})$ can be computationally expensive
  - Computing $\Phi_Q(\vec{x})$ requires $O(D^Q)$ time
- High-dimensional transformations can result in good hypotheses (as long as they don't overfit) but high-dimensional transformations are expensive
- Approach: instead of computing $\Phi(\vec{x})$, find a function K_Φ s.t. $K_\Phi(\vec{x}, \vec{x}') = \Phi(\vec{x})^T \Phi(\vec{x}') \; \forall \; \vec{x}, \vec{x}' \in \mathcal{X}$

Nonlinear Dual SVMs

- Decide on a (valid) kernel function K_Φ
- Find a maximal-margin separating hyperplane in the transformed space, $[\vec{\tilde{w}}^*, \tilde{w}_0^*]$, by solving the QP:

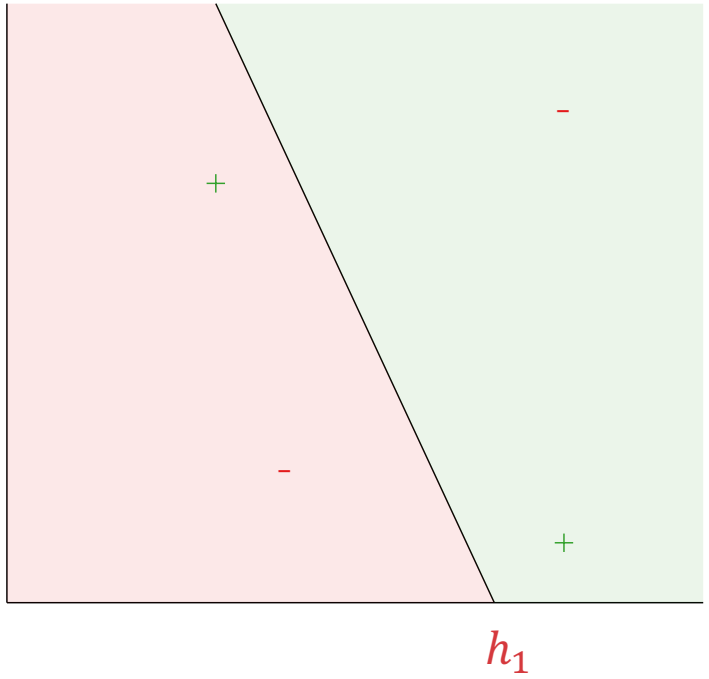
$$\text{minimize} \quad \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\tilde{\alpha}_i\tilde{\alpha}_j y_i y_j K_\Phi(\vec{x_i}, \vec{x_j}) - \sum_{i=1}^{n}\tilde{\alpha}_i$$

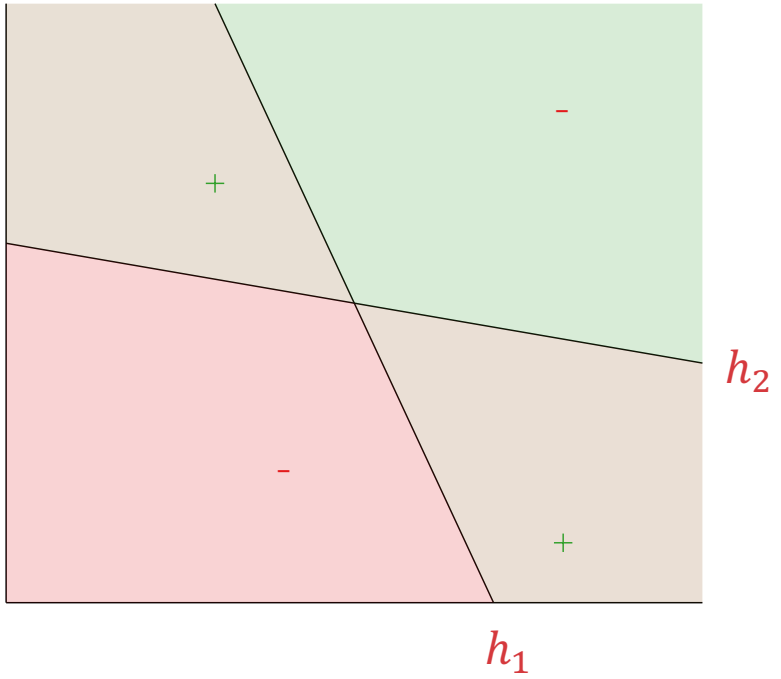
$$\text{subject to} \sum_{i=1}^{n}\tilde{\alpha}_i y_i = 0$$

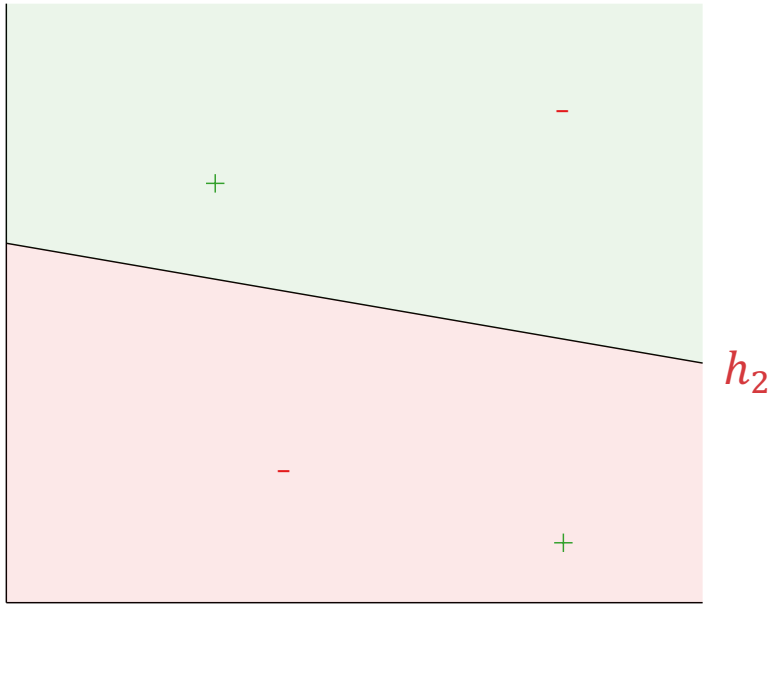
$$\tilde{\alpha}_i \geq 0 \; \forall \; i \in \{1, \ldots, n\}$$

- Return the corresponding predictor in the original space:

$$g(\vec{x}) = \text{sign}\left(\sum_{i:\alpha_i^* > 0}\tilde{\alpha}_i^* y_i K_\Phi(\vec{x_i}, \vec{x}) + \tilde{w}_0^*\right)$$



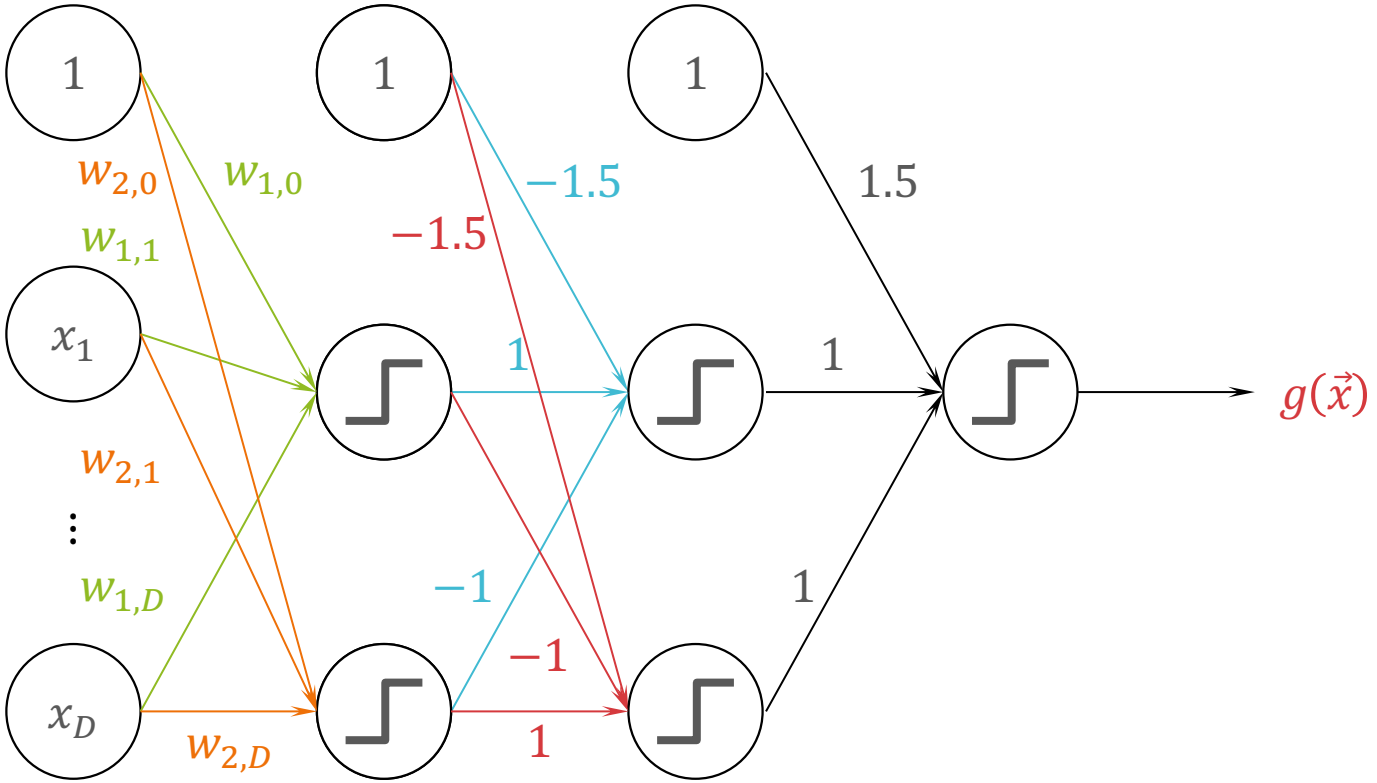




$$g(\vec{x}) = OR\left(AND\left(h_1(\vec{x}), \overline{h_2(\vec{x})}\right), AND\left(\overline{h_1(\vec{x})}, h_2(\vec{x})\right)\right)$$

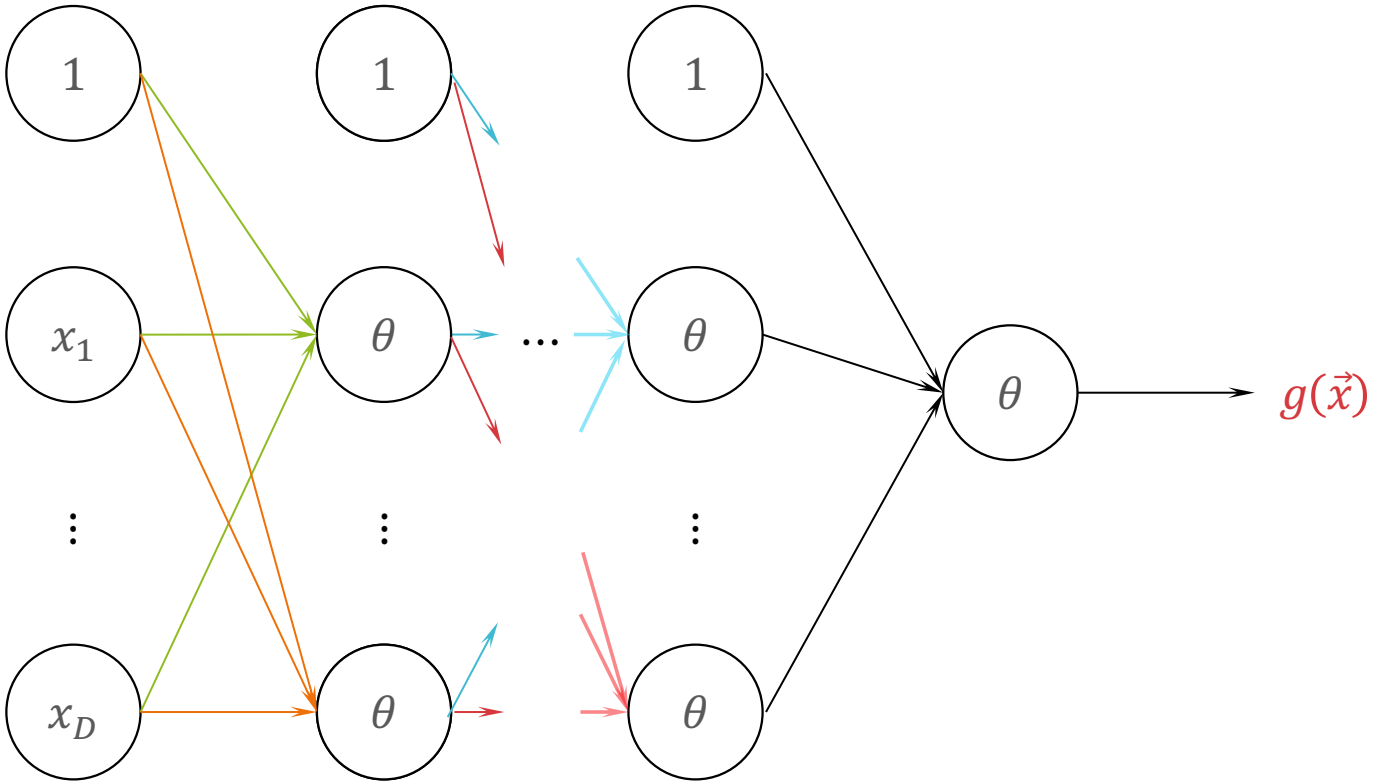
Building a Network

$$g(\vec{x}) = OR\left(AND\left(h_1(\vec{x}), \overline{h_2(\vec{x})}\right), AND\left(\overline{h_1(\vec{x})}, h_2(\vec{x})\right)\right)$$



Feed-Forward Neural Network (NN)

Replace the "hard" $\text{sign}(\cdot)$ function with a "soft", differentiable approximation, $\theta(\cdot)$



Architecture

- The architecture of a NN is the vector dimensionalities:

  $\vec{d} = \left[d^{(0)}, d^{(1)}, \ldots, d^{(L)}\right]$

  - $\left|\vec{d}\right| = L \rightarrow$ the NN has L layers, $L - 1$ hidden layers and 1 output layer
  - Layer l has dimension $d^{(l)} \rightarrow$ Layer l has $d^{(l)} + 1$ nodes, counting the bias node
  - Every architecture corresponds to a hypothesis set
  - A hypothesis is specified by setting all the weights

Weights, Signals and Outputs

- The weights between layer $l-1$ and layer l are a matrix: $W^{(l)} \in \mathbb{R}^{\left(d^{(l-1)}+1\right) \times d^{(l)}}$
  - $w_{ij}^{(l)}$ is the weight between node i in layer $l-1$ and node j in layer l
- Every node has an incoming signal, $s_j^{(l)}$, and an outgoing output, $x_j^{(l)}$:

$$\overrightarrow{x^{(l)}} = \left[\theta\left(\overrightarrow{s^{(l)}}\right)\right] \text{ and } \overrightarrow{s^{(l)}} = {W^{(l)}}^T \overrightarrow{x^{(l-1)}}$$

Forward Propagation

- Input: weights $W^{(1)}, \dots, W^{(L)}$ and a query point $\vec{x}$
- Initialize $\overrightarrow{x^{(0)}} = \begin{bmatrix} 1 \\ \vec{x} \end{bmatrix}$
- For $l = 1, \dots, L$
  - $\overrightarrow{s^{(l)}} = {W^{(l)}}^T \overrightarrow{x^{(l-1)}}$
  - $\overrightarrow{x^{(l)}} = \begin{bmatrix} 1 \\ \theta\left(\overrightarrow{s^{(l)}}\right) \end{bmatrix}$
- Output: $\overrightarrow{x^{(1)}}, \dots, \overrightarrow{x^{(L)}}$

Back-propagation

- Input: weights $W^{(1)}, \dots, W^{(L)}$ and a query point $\vec{x}$
- Run forward propagation to get $\overrightarrow{x^{(1)}}, \dots, \overrightarrow{x^{(L)}}$
- Initialize $\delta_1^{(L)} = 2\left(x_1^{(L)} - y_i\right)^2\left(1 - \left(x_1^{(L)}\right)^2\right)$
- For $l = L-1, \dots, 1$
  - Compute $\overrightarrow{\delta^{(l)}} = W^{(l+1)}\overrightarrow{\delta^{(l+1)}} \otimes \left(1 - \overrightarrow{x^{(l)}} \otimes \overrightarrow{x^{(l)}}\right)$
- Output: $\overrightarrow{\delta^{(1)}}, \dots, \overrightarrow{\delta^{(L)}}$

Computing Gradients

- Input: $W^{(1)}, \dots, W^{(L)}$ and $\mathcal{D} = \{(\overrightarrow{x_1}, y_1), \dots, (\overrightarrow{x_n}, y_n)\}$
- Initialize $E_{in} = 0$ and $G^{(l)} = 0 * W^{(l)}$ for $l = 1, \dots, L$
- For $i = 1, \dots, n$
  - Run forward propagation to get $\overrightarrow{x^{(1)}}, \dots, \overrightarrow{x^{(L)}}$
  - Run backpropagation to get $\overrightarrow{\delta^{(1)}}, \dots, \overrightarrow{\delta^{(L)}}$
  - Increment E_{in}: $E_{in} = E_{in} + \frac{1}{n}\left(\overrightarrow{x^{(L)}} - y_i\right)^2$
  - For $l = 1, \dots, L$
    - Compute $G_i^{(l)} = \overrightarrow{x^{(l-1)}}\left(\overrightarrow{\delta^{(l)}}\right)^T$
    - Increment $G^{(l)}$: $G^{(l)} = G^{(l)} + \frac{1}{n}G_i^{(l)}$
- Output: $G^{(1)}, \dots, G^{(L)}$, the gradients of E_{in} w.r.t $W^{(1)}, \dots, W^{(L)}$

Complexity

- Both forward and backpropagation contain matrix multiplications involving $W^{(1)}, \ldots, W^{(L)} \rightarrow$ both take time $O\left(\left|W^{(1)}\right| + \cdots + \left|W^{(L)}\right|\right)$ …
- Computing $G^{(1)}, \ldots, G^{(L)}$ requires running forward and backpropagation for each training point $(\vec{x}, y) \in \mathcal{D}$ …
- Each iteration of gradient descent for a neural network takes time $O\left(n\left(\left|W^{(1)}\right| + \cdots + \left|W^{(L)}\right|\right)\right)$
- Use stochastic gradient descent instead!
- Also use parallelization and GPUs / TPUs!

Stochastic Gradient Descent for Neural Networks

- Input: $\mathcal{D} = \{(\overrightarrow{x_1}, y_1), \dots, (\overrightarrow{x_n}, y_n)\}, \eta_0$
- Initialize all weights $W_{(0)}^{(1)}, \dots, W_{(0)}^{(L)}$ to small, random numbers and set $t = 0$
- While some termination condition is not satisfied
  - For $l = 1, \dots, L$
    - Randomly select a point $(\overrightarrow{x_1}, y_1) \in \mathcal{D}$
    - Compute $G^{(l)'} = \nabla_{W^{(l)}} e\left(h\left(\overrightarrow{x_i} \mid W_{(t)}^{(1)}, \dots, W_{(t)}^{(L)}\right), y_i\right)$
    - Update $W^{(l)}$: $W_{(t+1)}^{(l)} = W_{(t)}^{(l)} - \eta_0 G^{(l)'}$
  - Increment t: $t = t + 1$
- Output: $W_{(t)}^{(1)}, \dots, W_{(t)}^{(L)}$

Initialization and Termination

- Initialization:
  - Randomness is good for non-convex optimization
  - Initialize weights by sampling from $N(0, \sigma^2)$
- Termination:
  - For complicated surfaces, the gradient's magnitude is not a good metric for proximity to a minimum
  - A simple solution: combine multiple termination criteria e.g. stop if enough iterations have passed and the improvement in error is small