# Large-Scale Machine Learning: k-NN, Perceptron

Mining of Massive Datasets

Jure Leskovec, Anand Rajaraman, Jeff Ullman
Stanford University

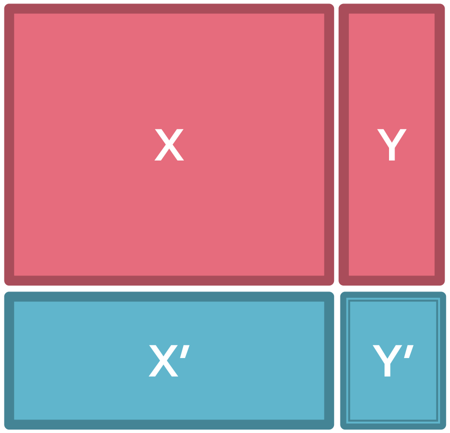
http://www.mmds.org

# New Topic: Machine Learning!

| High dim. data | Graph data | Infinite data | Machine learning | Apps |
|---|---|---|---|---|
| Locality sensitive hashing | **PageRank,** SimRank | Filtering data streams | SVM | Recommender systems |
| Clustering | Community Detection | Web advertising | Decision Trees | Association Rules |
| Dimensionality reduction | Spam Detection | Queries on streams | Perceptron, kNN | Duplicate document detection |

# Supervised Learning

- **Would like to do prediction:**
  **estimate** a function **f(x)** so that ***y = f(x)***
- **Where *y* can be:**
  - **Real number**: Regression
  - **Categorical**: Classification
  - Complex object:
    - Ranking of items, Parse tree, etc.
- **Data is labeled:**
  - Have many pairs **{(x, y)}**
    - **x** … vector of binary, categorical, real valued features
    - **y** … class ({+1, -1}, or a real number)




Training and test set

**Estimate *y = f(x)* on *X,Y*.**
**Hope that the same *f(x)***
***also works on unseen X', Y'***

# Large Scale Machine Learning

- **We will talk about the following methods:**
  - **k**-Nearest Neighbor (Instance based learning)
  - Perceptron and Winnow algorithms
  - Support Vector Machines
  - Decision trees

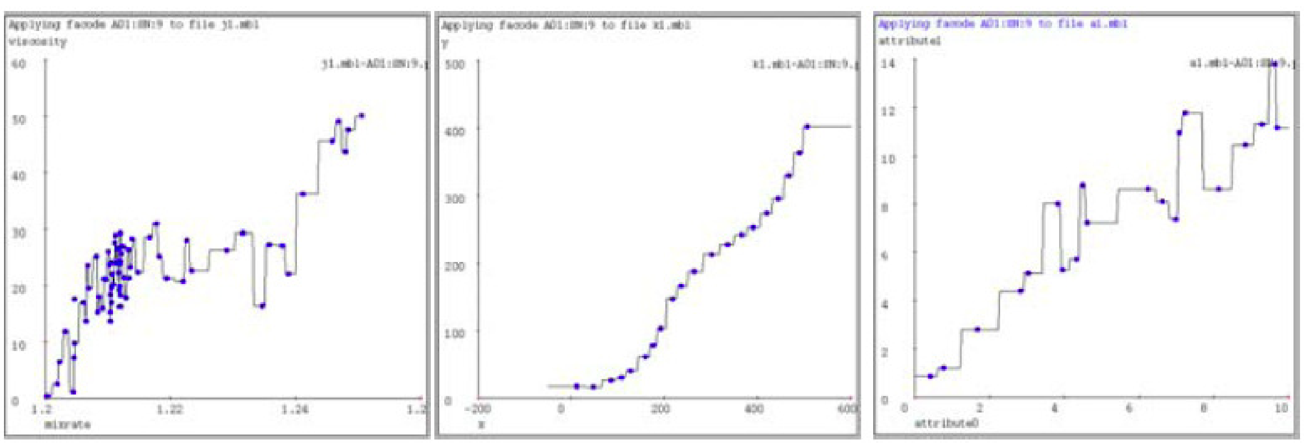
- **Main question:**
  **How to efficiently train**
  (build a model/find model parameters)**?**

# Instance Based Learning

- **Instance based learning**
- **Example: Nearest neighbor**
  - Keep the whole training dataset: **{(x, y)}**
  - A query example (vector) ***q*** comes
  - Find closest example(s) $\mathbf{x}^*$
  - Predict $\mathbf{y}^*$
- **Works both for regression and classification**
  - **Collaborative filtering** is an example of k-NN classifier
    - Find ***k*** most similar people to user **x** that have rated movie **y**
    - Predict rating $\mathbf{y}_x$ of **x** as an average of $\mathbf{y}_k$

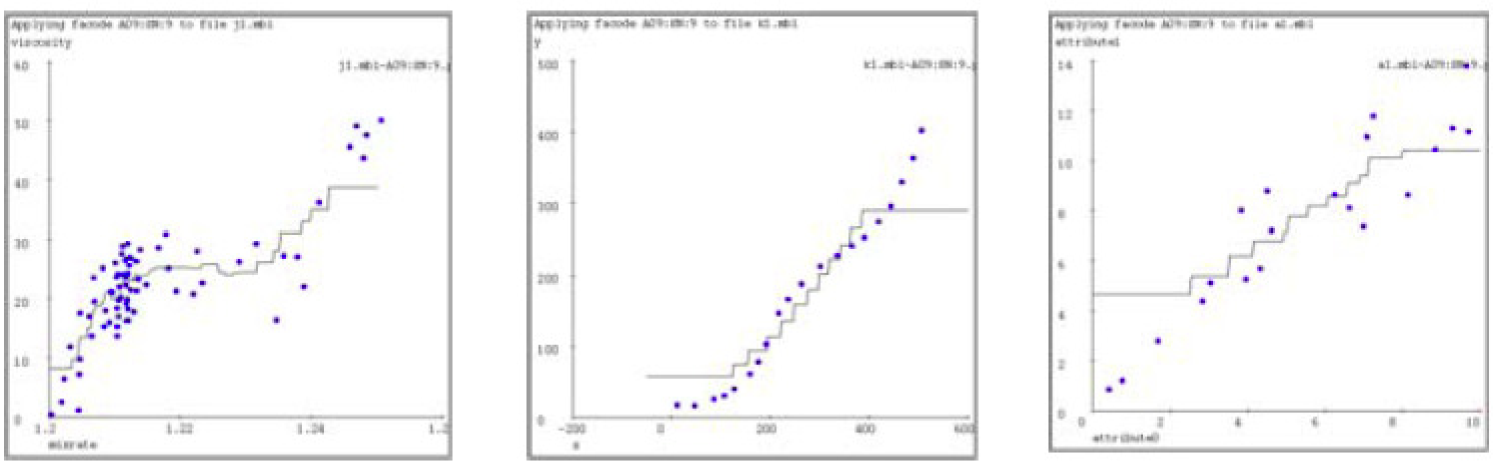
# 1-Nearest Neighbor

- **To make Nearest Neighbor work we need 4 things:**
  - **Distance metric:**
    - Euclidean
  - **How many neighbors to look at?**
    - One
  - **Weighting function (optional):**
    - Unused
  - **How to fit with the local points?**
    - Just predict the same output as the nearest neighbor



# *k*-Nearest Neighbor

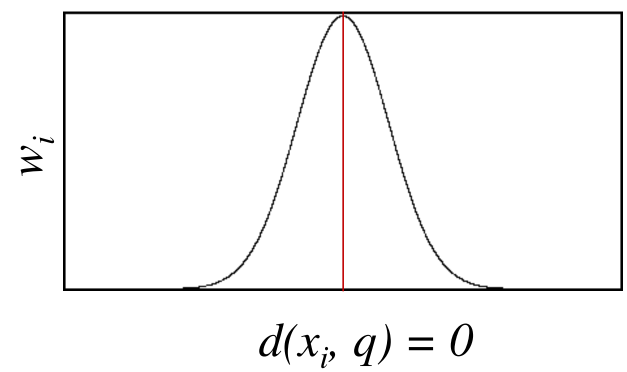
- **Distance metric:**
  - Euclidean
- **How many neighbors to look at?**
  - ***k***
- **Weighting function (optional):**
  - Unused
- **How to fit with the local points?**
  - Just predict the average output among *k* nearest neighbors

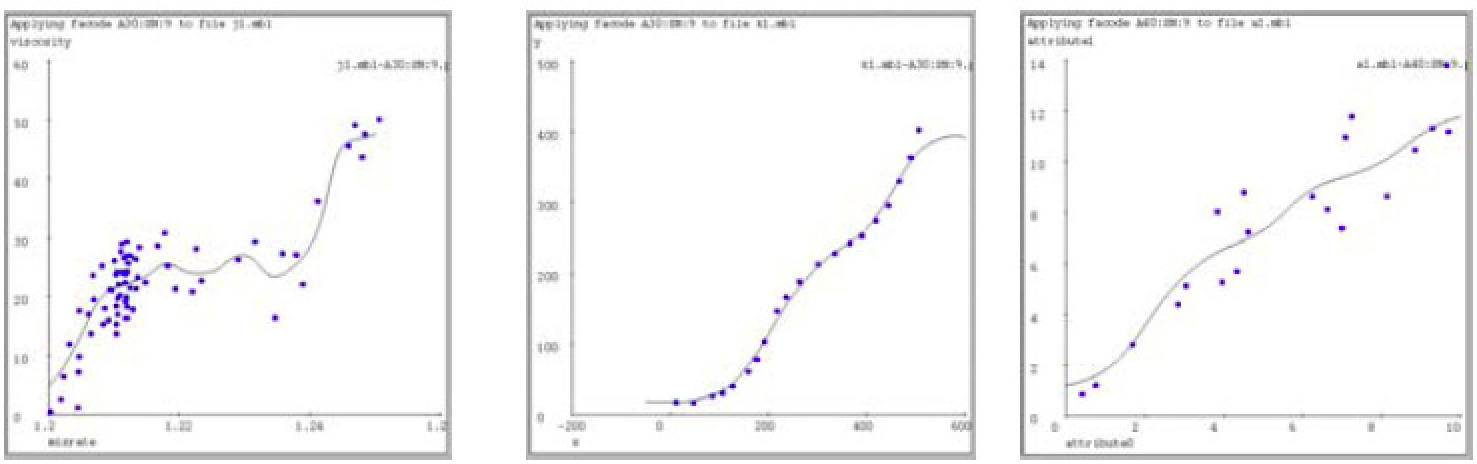


**k=9**

# Kernel Regression

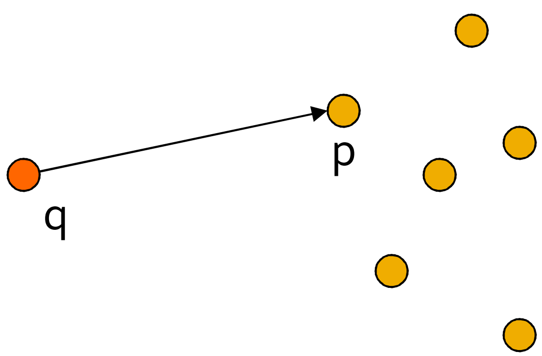
- **Distance metric:**
  - Euclidean
- **How many neighbors to look at?**
  - **All of them (!)**
- **Weighting function:**
  - $w_i = \exp(-\frac{d(x_i, q)^2}{K_w})$
    - Nearby points to query q are weighted more strongly. $K_w$…kernel width.
- **How to fit with the local points?**
  - **Predict weighted average:** $\frac{\sum_i w_i y_i}{\sum_i w_i}$



# How to find nearest neighbors?

- **Given:** a set ***P*** of ***n*** points in $R^d$
- **Goal: Given a query point *q***
  - **NN:** Find the *nearest neighbor* ***p*** of ***q*** in ***P***
  - **Range search:** Find one/all points in ***P*** within distance ***r*** from ***q***



# Algorithms for NN

- **Main memory:**
  - **Linear scan**
  - **Tree based:**
    - Quadtree
    - kd-tree
  - **Hashing:**
    - Locality-Sensitive Hashing
- **Secondary storage:**
  - R-trees

(1958)
F. Rosenblatt

The perceptron: a probabilistic model
for information storage and organization in the brain
*Psychological Review* 65:386–408

# Perceptron

# Linear models: Perceptron

- **Example:** **Spam filtering**

| | viagra | learning | the | dating | nigeria | $spam?$ |
|---|---|---|---|---|---|---|
| $\vec{x}_1 = ($ | 1 | 0 | 1 | 0 | 0 $)$ | $y_1 = 1$ |
| $\vec{x}_2 = ($ | 0 | 1 | 1 | 0 | 0 $)$ | $y_2 = -1$ |
| $\vec{x}_3 = ($ | 0 | 0 | 0 | 0 | 1 $)$ | $y_3 = 1$ |

- **Instance space x ∈ X** (|**X**|= **n** data points)
  - **Binary or real-valued feature vector *x* of word occurrences**
  - ***d*** features (words + other things, **d**~100,000)
- **Class y ∈ Y**
  - ***y***: Spam (+1), Ham (-1)

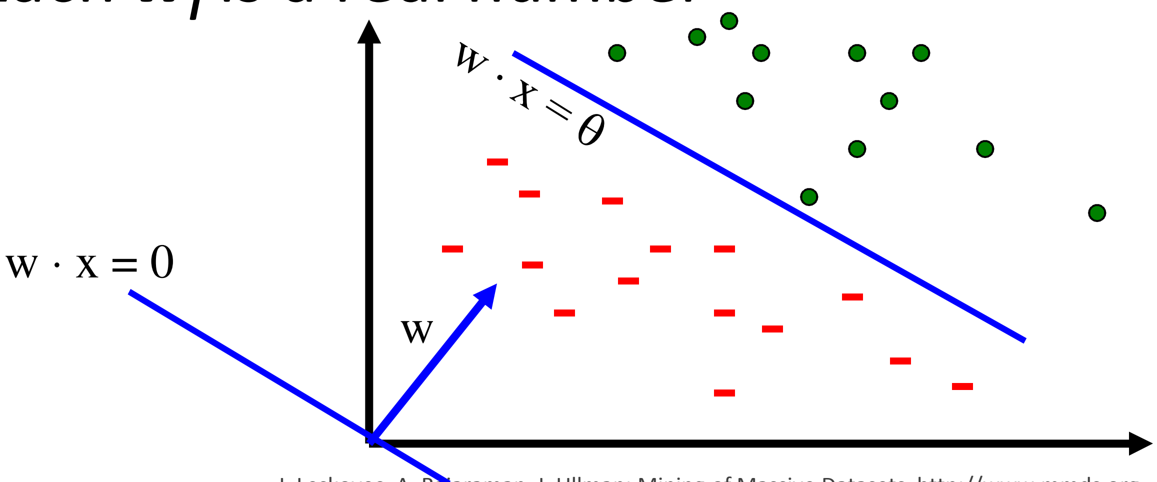
# Linear models for classification

- **Binary classification:**

$$f(\mathbf{x}) = \begin{cases} +1 & \text{if} \quad \mathbf{w_1 x_1 + w_2 x_2 + \ldots w_d x_d} \geq \theta \\ -1 & \text{otherwise} \end{cases}$$

Decision boundary is **linear**

- **Input:** Vectors $\boldsymbol{x^{(j)}}$ and labels $\boldsymbol{y^{(j)}}$
  - Vectors $\boldsymbol{x^{(j)}}$ are real valued where $\|x\|_2 = 1$
- **Goal:** Find vector $\boldsymbol{w = (w_1, w_2, \ldots, w_d)}$
  - Each $\boldsymbol{w_i}$ is a real number



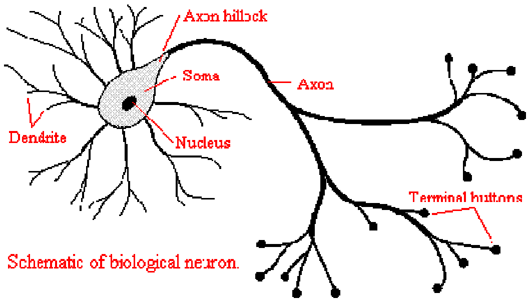

**Note:**

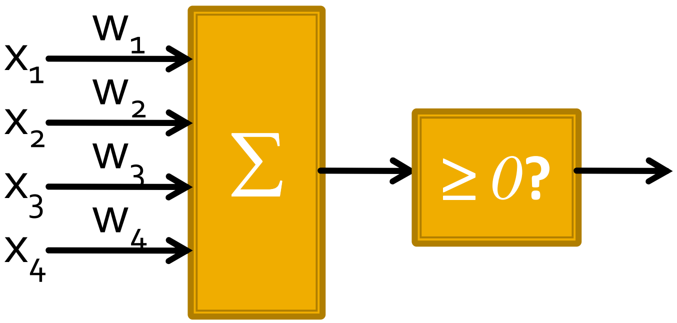
$$\mathbf{x} \Leftrightarrow \langle \mathbf{x}, 1 \rangle \quad \forall \mathbf{x}$$

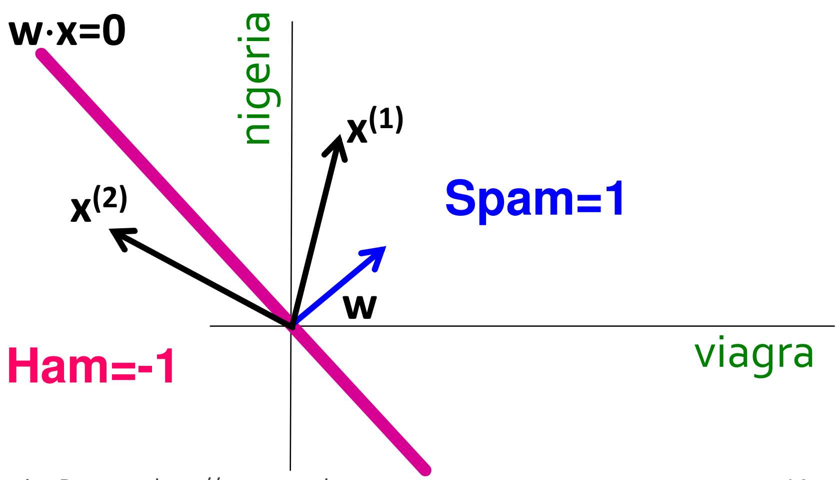
$$\mathbf{w} \Leftrightarrow \langle \mathbf{w}, -\theta \rangle$$

# Perceptron [Rosenblatt '58]

- **(very) Loose motivation: Neuron**
- Inputs are feature values
- Each feature has a weight $\boldsymbol{w_i}$
- **Activation is the sum:**
  - $f(x) = \Sigma_i \, w_i \, x_i = w \cdot x$
- If the ***f(x)*** is:
  - **Positive:** Predict **+1**
  - **Negative:** Predict **-1**



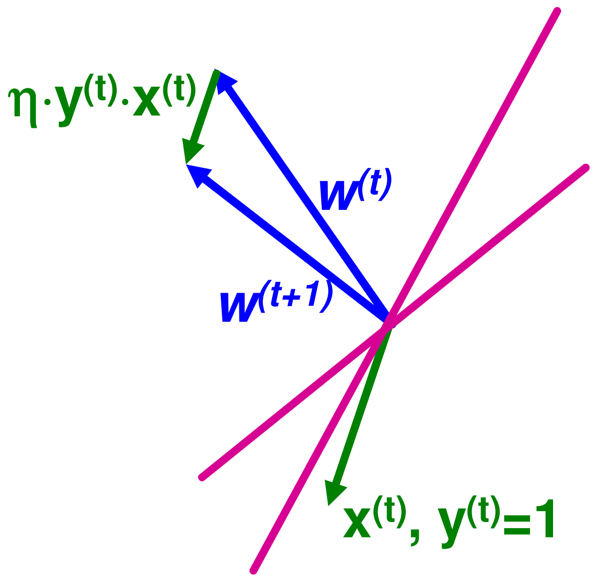
# Perceptron: Estimating *w*

Note that the Perceptron is a conservative algorithm: it ignores samples that it classifies correctly.

- **Perceptron:** $y' = sign(w \cdot x)$
- **How to find parameters $w$?**
  - Start with $w_0 = 0$
  - Pick training examples $x^{(t)}$ **one by one (from disk)**
  - Predict class of $x^{(t)}$ using current weights
    - $y' = sign(w^{(t)} \cdot x^{(t)})$
  - **If $y'$ is correct (i.e., $y_t = y'$)**
    - No change: $w^{(t+1)} = w^{(t)}$
  - **If $y'$ is wrong:** adjust $w^{(t)}$

$$w^{(t+1)} = w^{(t)} + \eta \cdot y^{(t)} \cdot x^{(t)}$$

- $\eta$ is the learning rate parameter
- $x^{(t)}$ is the t-th training example
- $y^{(t)}$ is true t-th class label ({+1, -1})



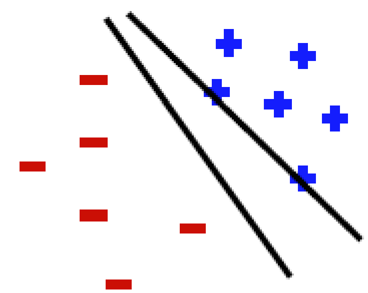
# Perceptron Convergence

- **Perceptron Convergence Theorem:**
  - If there exist a set of weights that are consistent (i.e., the data is linearly separable) the Perceptron learning algorithm will converge
- **How long would it take to converge?**
- **Perceptron Cycling Theorem:**
  - If the training data is not linearly separable the Perceptron learning algorithm will eventually repeat the same set of weights and therefore enter an infinite loop
- **How to provide robustness, more expressivity?**

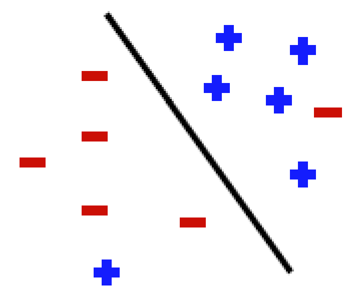
# Properties of Perceptron

- **Separability:** Some parameters get training set perfectly
- **Convergence:** If training set is separable, perceptron will converge
- **(Training) Mistake bound:**
  Number of mistakes $< \frac{1}{\gamma^2}$
  - **where** $\boldsymbol{\gamma = \min_{t,u} |x^{(t)} u|}$
    and $\|u\|_2 = 1$
    - Note we assume **x** Euclidean length **1**, then **γ** is the minimum distance of any example to plane **u**

Separable



Non-Separable



# Updating the Learning Rate

- **Perceptron will oscillate and won't converge**
- **When to stop learning?**
- **(1)** Slowly decrease the learning rate $\eta$
  - A classic way is to: $\eta = \mathbf{c_1/(t + c_2)}$
    - But, we also need to determine constants $\mathbf{c_1}$ and $\mathbf{c_2}$
- **(2)** Stop when the training error stops chaining
- **(3)** Have a small test dataset and stop when the test set error stops decreasing
- **(4)** Stop when we reached some maximum number of passes over the data

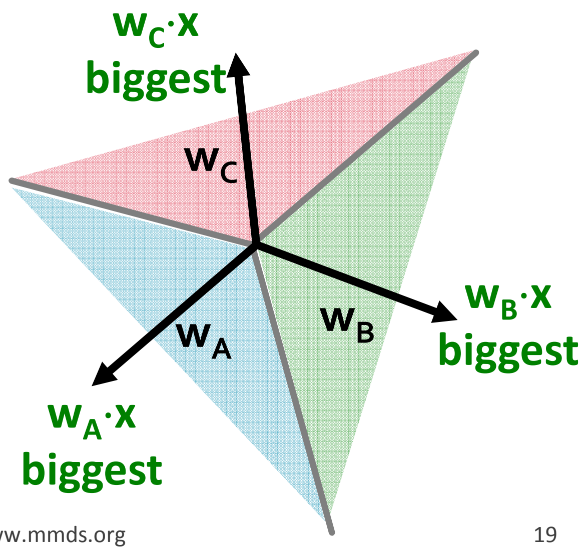
# Multiclass Perceptron

- **What if more than 2 classes?**
- Weight vector $\boldsymbol{w_c}$ for each class **c**
  - **Train one class vs. the rest:**
    - Example: 3-way classification **y = {A, B, C}**
    - Train 3 classifiers: $\mathbf{w_A}$: A vs. B,C; $\mathbf{w_B}$: B vs. A,C; $\mathbf{w_C}$: C vs. A,B
- **Calculate activation for each class**

$$f(x,c) = \Sigma_i \; w_{c,i} \, x_i \; = \; w_c \cdot x$$

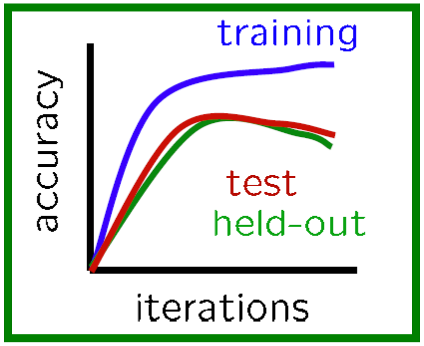
- **Highest activation wins**

$$c = \arg\max_c f(x,c)$$

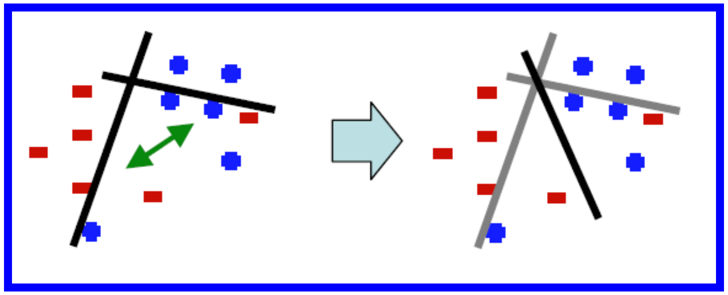


# Issues with Perceptrons

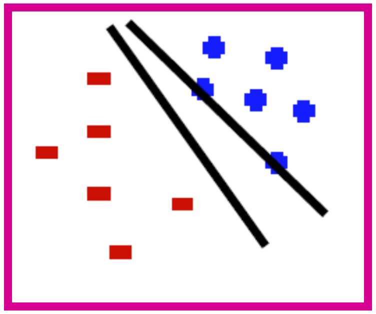
- **Overfitting:**




- **Regularization:** If the data is not separable weights dance around



- **Mediocre generalization:**
  - Finds a “barely” separating solution



# Improvement: Winnow Algorithm

- **Winnow :** Predict **f(x) = +1** iff $\boldsymbol{w} \cdot \boldsymbol{x} \geq \boldsymbol{\theta}$
  - Similar to perceptron, just different updates
  - **Assume *x* is a real-valued feature vector,** $\|\boldsymbol{x}\|_2 = \mathbf{1}$

    - Initialize: $\boldsymbol{\theta} = \frac{d}{2},\ \boldsymbol{w} = \left[\frac{1}{d}, \dots, \frac{1}{d}\right]$
    - **For every training example** $\boldsymbol{x}^{(t)}$
      - **Compute** $\boldsymbol{y'} = \boldsymbol{f}(\boldsymbol{x}^{(t)})$
      - If no mistake ($\boldsymbol{y}^{(t)} = \boldsymbol{y'}$): do nothing
      - If mistake then: $\boldsymbol{w_i} \leftarrow \boldsymbol{w_i} \frac{\mathbf{exp}\left(\boldsymbol{\eta y}^{(t)} \boldsymbol{x}_i^{(t)}\right)}{\boldsymbol{Z}^{(t)}}$

  - **w** … weights **(can never get negative!)**
  - $\boldsymbol{Z}^{(t)} = \sum_i \boldsymbol{w_i} \mathbf{exp}\left(\boldsymbol{\eta y}^{(t)} \boldsymbol{x}_i^{(t)}\right)$ is the normalizing const.

# Improvement: Winnow Algorithm

- **About the update:** $\boldsymbol{w_i \leftarrow w_i \frac{\exp\left(\eta y^{(t)} x_i^{(t)}\right)}{Z^{(t)}}}$
  - If **x** is false negative, increase $\mathbf{w}_i$ **(promote)**
  - If **x** is false positive, decrease $\mathbf{w}_i$ **(demote)**

- **In other words:** Consider $\boldsymbol{x_i^{(t)}} \in \{-1, +1\}$

- Then $\boldsymbol{w_i^{(t+1)}} \propto \boldsymbol{w_i^{(t)}} \cdot \begin{cases} \boldsymbol{e^{\eta}} & if\ x_i^{(t)} = y^{(t)} \\ \boldsymbol{e^{-\eta}} & else \end{cases}$
  - **Notice: This is a weighted majority algorithm of "experts" $x_i$ agreeing with y**

# Extensions: Winnow

- **Problem:** **All $w_i$ can only be >0**
- **Solution:**
  - For every feature $\mathbf{x_i}$, introduce a new feature $x_i' = -x_i$
  - Learn Winnow over ***2d*** features
- **Example:**
  - Consider: $\boldsymbol{x} = [1, .7, -.4], \boldsymbol{w} = [.5, .2, -.3]$
  - Then new $\boldsymbol{x}$ and $\boldsymbol{w}$ are $\boldsymbol{x} = [1, .7, -.4, -1, -.7, .4], \mathbf{w} = [.5, .2, 0, 0, 0, .3]$
  - Note this results in the same dot values as if we used original $\boldsymbol{x}$ and $\mathbf{w}$
- New algorithm is called **Balanced Winnow**

# Extensions: Balanced Winnow

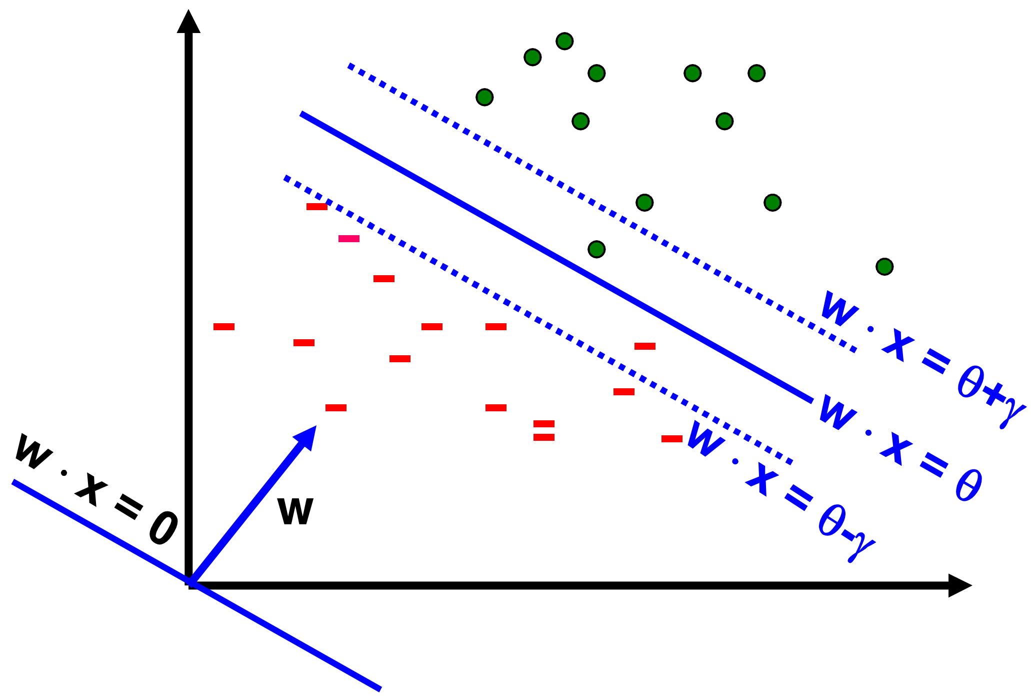
- **In practice we implement Balanced Winnow:**
  - **2** weight vectors $\mathbf{w}^+$, $\mathbf{w}^-$; effective weight is the difference

- **Classification rule:**
  - $f(x) = +1$ if $(\mathbf{w}^+ - \mathbf{w}^-) \cdot \mathbf{x} \geq \theta$
- **Update rule:**
  - **If mistake:**
    - $\mathbf{w}_i^+ \leftarrow \mathbf{w}_i^+ \dfrac{\exp\left(\eta y^{(t)} x_i^{(t)}\right)}{Z^{+(t)}}$
    - $\mathbf{w}_i^- \leftarrow \mathbf{w}_i^- \dfrac{\exp\left(-\eta y^{(t)} x_i^{(t)}\right)}{Z^{-(t)}}$

$$Z^{-(t)} = \sum_i w_i \exp\left(-\eta y^{(t)} x_i^{(t)}\right)$$

# Extensions: Thick Separator

- **Thick Separator** (aka **Perceptron with Margin**) (Applies both to Perceptron and Winnow)
  - Set margin parameter $\gamma$
  - **Update** if **y=+1** but $\mathbf{w} \cdot \mathbf{x} < \theta + \gamma$
  - **or** if **y=-1** but $\mathbf{w} \cdot \mathbf{x} > \theta - \gamma$




**Note:** $\gamma$ is a functional margin. Its effect could disappear as ***w*** grows. Nevertheless, this has been shown to be a very effective algorithmic addition.

# Summary of Algorithms

- **Setting:**
  - **Examples:** $x \in \{0, 1\}$, **weights** $w \in R^d$
  - **Prediction:** $f(x) = +1$ iff $w \cdot x \geq \theta$ else $-1$
- **Perceptron:** Additive weight update

$$\boxed{w \leftarrow w + \eta\, y\, x}$$

  - **If $y=+1$ but $w \cdot x \leq \theta$ then $w_i \leftarrow w_i + 1$ (if $x_i=1$)** **(promote)**
  - **If $y=-1$ but $w \cdot x > \theta$ then $w_i \leftarrow w_i - 1$ (if $x_i=1$)** **(demote)**
- **Winnow:** Multiplicative weight update

$$\boxed{w \leftarrow w \exp\{\eta\, y\, x\}}$$

  - **If $y=+1$ but $w \cdot x \leq \theta$ then $w_i \leftarrow 2 \cdot w_i$ (if $x_i=1$)** **(promote)**
  - **If $y=-1$ but $w \cdot x > \theta$ then $w_i \leftarrow w_i / 2$ (if $x_i=1$)** **(demote)**

# Perceptron vs. Winnow

- **How to compare learning algorithms?**
- **Considerations:**
  - Number of features **d** is **very large**
  - **The instance space is sparse**
    - Only few features per training example are non-zero
  - **The model is sparse**
    - Decisions depend on a small subset of features
    - In the “true” model on a few $\mathbf{w}_i$ are non-zero
  - Want to learn from a number of examples that is small relative to the dimensionality **d**

# Perceptron vs. Winnow

## Perceptron

- **Online**: Can adjust to changing target, over time
- **Advantages**
  - Simple
  - Guaranteed to learn a linearly separable problem
  - **Advantage with few relevant features per training example**
- **Limitations**
  - Only linear separations
  - Only converges for linearly separable data
  - Not really "efficient with many features"

## Winnow

- **Online**: Can adjust to changing target, over time
- **Advantages**
  - Simple
  - Guaranteed to learn a linearly separable problem
  - **Suitable for problems with many irrelevant attributes**
- **Limitations**
  - Only linear separations
  - Only converges for linearly separable data
  - Not really "efficient with many features"

# Online Learning

- **New setting: Online Learning**
  - Allows for modeling problems where we have a continuous stream of data
  - We want an algorithm to learn from it and slowly adapt to the changes in data
- **Idea: Do slow updates to the model**
  - Both our methods Perceptron and Winnow make updates if they misclassify an example
  - **So:** First train the classifier on training data. Then for every example from the stream, if we misclassify, update the model (using small learning rate)

# Example: Shipping Service

- **Protocol:**
  - User comes and tell us origin and destination
  - We offer to ship the package for some money (\$10 - \$50)
  - Based on the price we offer, sometimes the user uses our service (**y = 1**), sometimes they don't (**y = -1**)
- **Task:** Build an algorithm to optimize what price we offer to the users
- **Features *x* capture:**
  - Information about user
  - Origin and destination
- **Problem: Will user accept the price?**

# Example: Shipping Service

- **Model whether user will accept our price:**
  **y = f(x; w)**
  - **Accept: y =1**, **Not accept: y=-1**
  - Build this model with say Perceptron or Winnow
- **The website that runs continuously**
- **Online learning algorithm would do something like**
  - User comes
  - She is represented as an **(x,y)** pair where
    - **x:** Feature vector including price we offer, origin, destination
    - **y:** If they chose to use our service or not
  - The algorithm updates **w** using just the **(x,y)** pair
  - Basically, we update the **w** parameters every time we get some new data

# Example: Shipping Service

- We discard this idea of a data "set"
- Instead we have a continuous stream of data
- **Further comments:**
  - For a major website where you have a massive stream of data then this kind of algorithm is pretty reasonable
  - Don't need to deal with all the training data
  - If you had a small number of users you could save their data and then run a normal algorithm on the full dataset
    - Doing multiple passes over the data

# Online Algorithms

- An online algorithm can adapt to changing user preferences
- For example, over time users may become more price sensitive
- **The algorithm adapts and learns this**
- So the system is dynamic