Features and kernels

Online Perceptron

Online Perceptron

input Labeled examples $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^{n}$ from $\mathbb{R}^d \times \{-1,+1\}$.

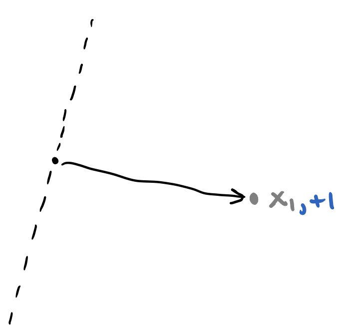
initialize $\hat{\boldsymbol{w}}_1 := \boldsymbol{0}$.
for $t = 1, 2, \dots, n$ **do**
    if $y_t \langle \hat{\boldsymbol{w}}_t, \boldsymbol{x}_t \rangle \leq 0$ **then**
        $\hat{\boldsymbol{w}}_{t+1} := \hat{\boldsymbol{w}}_t + y_t \boldsymbol{x}_t$.
    else
        $\hat{\boldsymbol{w}}_{t+1} := \hat{\boldsymbol{w}}_t$
    end if
end for
return $\hat{\boldsymbol{w}}_{n+1}$.

Theorem: If $R := \max_{t \in \{1,\dots,n\}} \|\boldsymbol{x}_t\|_2$, and $\boldsymbol{w}_\star \in \mathbb{R}^d$ satisfies

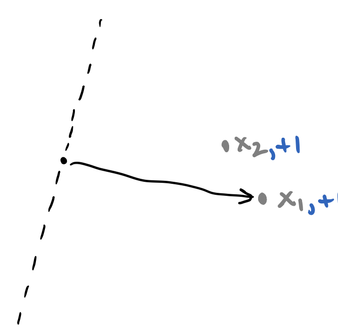
$$y \langle \boldsymbol{w}_\star, \boldsymbol{x}_t \rangle \; \geq \; 1 \quad \text{for all } (\boldsymbol{x}_t, y_t),$$

then Online Perceptron makes at most $\|\boldsymbol{w}_\star\|_2^2 \cdot R^2$ mistakes (and updates).

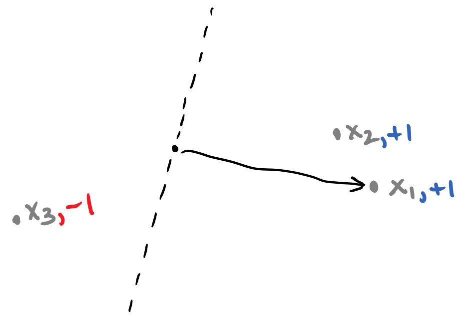
Example run of Online Perceptron



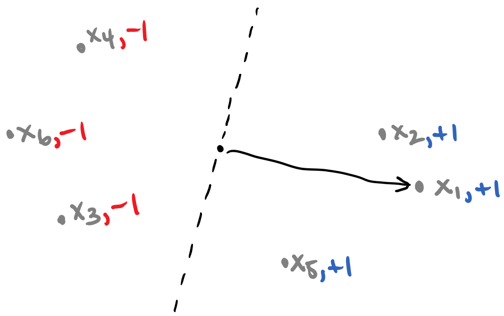
Example run of Online Perceptron



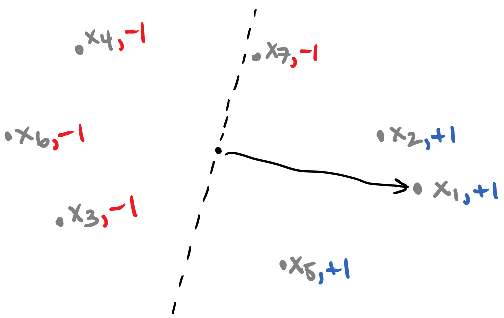
Example run of Online Perceptron



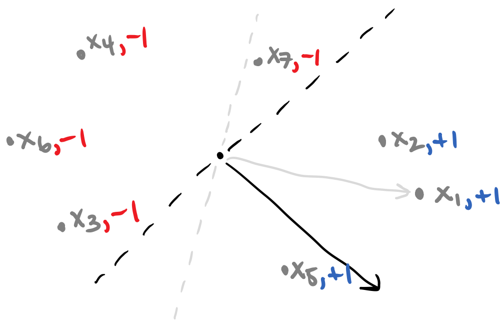
Example run of Online Perceptron



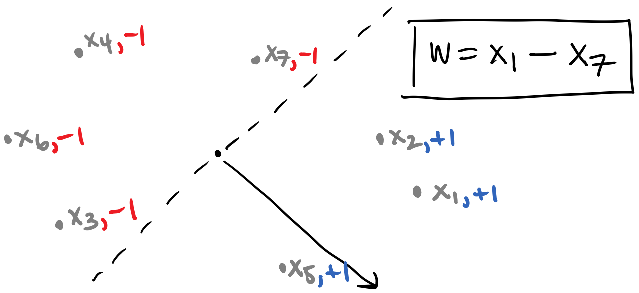
Example run of Online Perceptron



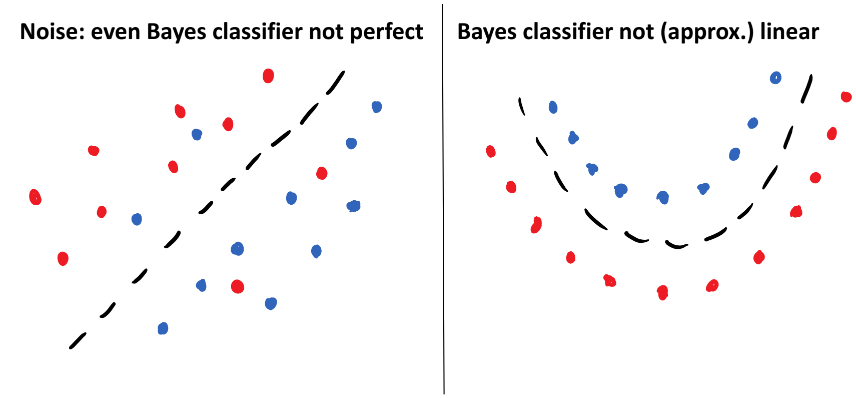
Example run of Online Perceptron



Example run of Online Perceptron

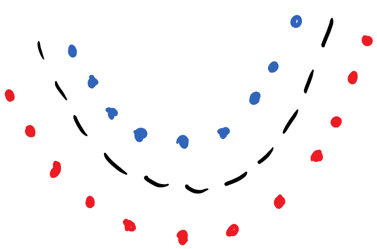


What to do if data is not linearly separable



Adding new features

Original feature vector: $\boldsymbol{x} = (x_1, x_2)$.
New feature vector: $\boldsymbol{\phi}(\boldsymbol{x}) = (1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, \sqrt{2}x_1x_2, x_2^2)$.



Decision boundary is non-linear in $\boldsymbol{x}$, but is linear in $\boldsymbol{\phi}(\boldsymbol{x})$.

Getting the most out of linear classifiers

Often, with a "good" set of features, a linear classifier can well-approximate the Bayes classifier.

Two approaches:

1. Think very hard and carefully about which features to use.
2. Use all features that come to mind.

The kitchen sink of features

Example: document classification

- **Word features**:

$$\mathbb{1}\{\text{“aardvark” appears}\}, \mathbb{1}\{\text{“abacus” appears}\}, \dots, \mathbb{1}\{\text{“zygote” appears}\}$$

- **Bi-gram features**:

$$\mathbb{1}\{\text{“bank deposit” appears}\}, \mathbb{1}\{\text{“river bank” appears}\}, \dots$$

- **Tri-gram features**:

$$\mathbb{1}\{\text{“New York City” appears}\}, \mathbb{1}\{\text{“wherefore art though” appears}\}, \dots$$

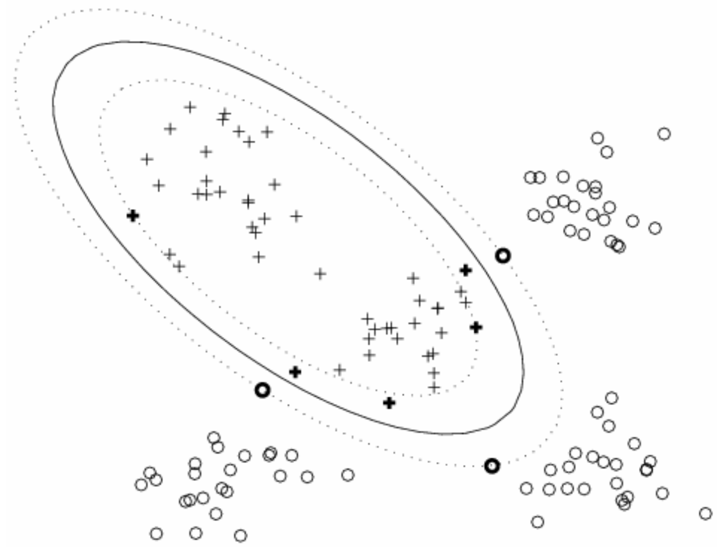
Example: new features from old features $\boldsymbol{x} \in \mathbb{R}^d$

- **Pairwise interactions**:

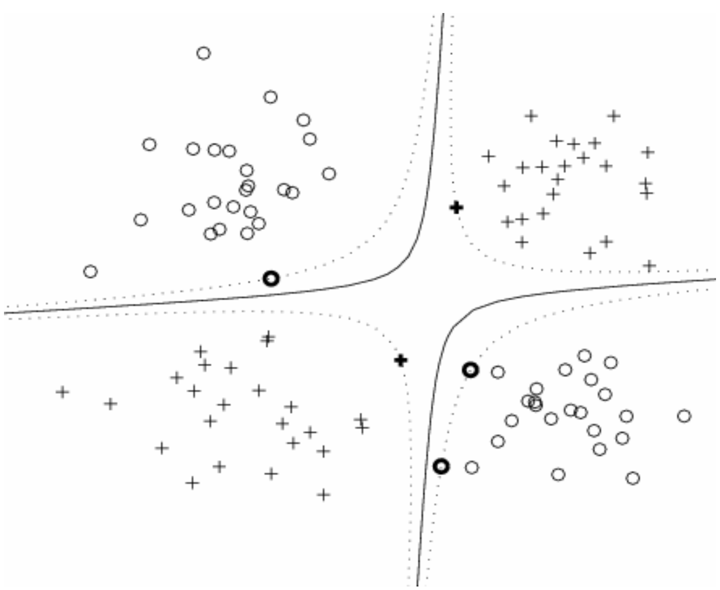
$$(x_1 x_2,\ x_1 x_3,\ \dots,\ x_1 x_d,\ x_2 x_3,\ \dots,\ x_{d-1} x_d) \ \in\ \mathbb{R}^{\binom{d}{2}}$$

- Etc.

All degree ≤ 2 interaction features



All degree ≤ 2 interaction features



Learning with the kitchen sink of features

- Let $\boldsymbol{\phi} \colon \mathbb{R}^d \to \mathbb{R}^D$ be the mapping to the expanded feature space (e.g., all original features and all deg. 2 interactions, so $D = \Omega(d^2)$).
- Learn linear classifier $f_{\boldsymbol{w}} \colon \mathbb{R}^D \to \{\pm 1\}$ (i.e., learn a weight vector $\boldsymbol{w} \in \mathbb{R}^D$) using data with expanded features $\{(\boldsymbol{\phi}(\boldsymbol{x}_i), y_i)\}_{i=1}^n$.
- **Caveat**: can be computationally expensive to do this directly if D is large. Naïvely: takes $\Omega(D)$ time to even make a prediction.

Kernel trick

- **Recall**: Online Perceptron weight vector (using expanded features) is

$$\boldsymbol{w} \;=\; \sum_{(\boldsymbol{x},y)\in\mathcal{M}} y\boldsymbol{\phi}(\boldsymbol{x})$$

where $\mathcal{M}$ is the subset of labeled examples where Online Perceptron made a mistake during training.

- **Prediction** using Online Perceptron weight vector on new point $\boldsymbol{z} \in \mathbb{R}^d$:

$$\langle \boldsymbol{w}, \boldsymbol{\phi}(\boldsymbol{z})\rangle \;=\; \sum_{(\boldsymbol{x},y)\in\mathcal{M}} y\langle \boldsymbol{\phi}(\boldsymbol{x}), \boldsymbol{\phi}(\boldsymbol{z})\rangle\,.$$

Computational cost: $|\mathcal{M}|\times$ time to compute inner product $\langle \boldsymbol{\phi}(\boldsymbol{x}), \boldsymbol{\phi}(\boldsymbol{z})\rangle$.
Sometimes this can be faster than $O(D)$.

Degree ≤ 2 interaction features

- $\boldsymbol{\phi}\colon \mathbb{R}^d \to \mathbb{R}^{1+2d+\binom{d}{2}}$, where

$$\boldsymbol{\phi}(\boldsymbol{x}) \;=\; \Big(1,\, \sqrt{2}x_1,\, \dots,\, \sqrt{2}x_d,\, x_1^2,\, \dots,\, x_d^2,\, \sqrt{2}x_1x_2,\, \dots,\, \sqrt{2}x_1x_d,\, \dots,\, \sqrt{2}x_{d-1}x_d\Big)$$

(Don't mind the $\sqrt{2}$'s.)

- **Computing** $\langle \boldsymbol{\phi}(\boldsymbol{x}), \boldsymbol{\phi}(\boldsymbol{x}')\rangle$ **in** $O(d)$ **time**:

$$\langle \boldsymbol{\phi}(\boldsymbol{x}), \boldsymbol{\phi}(\boldsymbol{x}')\rangle \;=\; (1 + \langle \boldsymbol{x}, \boldsymbol{x}'\rangle)^2\,.$$

- Much better than $\Omega(d^2)$.

Products of all feature subsets

- $\boldsymbol{\phi}\colon \mathbb{R}^d \to \mathbb{R}^{2^d}$, where

$$\boldsymbol{\phi}(\boldsymbol{x}) \;=\; \left(\prod_{i\in S} x_i \;:\; S \subseteq \{1,2,\dots,d\}\right)$$

- **Computing** $\langle \boldsymbol{\phi}(\boldsymbol{x}), \boldsymbol{\phi}(\boldsymbol{x}')\rangle$ **in** $O(d)$ **time**:

$$\langle \boldsymbol{\phi}(\boldsymbol{x}), \boldsymbol{\phi}(\boldsymbol{x}')\rangle \;=\; \prod_{i=1}^{d}(1 + x_i x_i')\,.$$

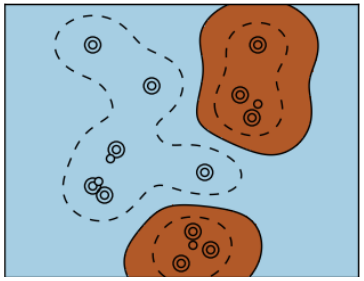
- Much better than $\Omega(2^d)$.

Infinite dimensional feature expansion

For any $\sigma > 0$, there is an infinite feature expansion $\boldsymbol{\phi}\colon \mathbb{R}^d \to \mathbb{R}^\infty$ (involving Hermite polynomials of all orders) such that

$$\langle \boldsymbol{\phi}(\boldsymbol{x}), \boldsymbol{\phi}(\boldsymbol{x}')\rangle \;=\; \exp\left(-\frac{\|\boldsymbol{x}-\boldsymbol{x}'\|_2^2}{2\sigma^2}\right),$$

which can be computed in $O(d)$ time.



(This is called the **Gaussian kernel** with bandwidth σ.)

Kernels

A **kernel function** $K\colon \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a symmetric function with the following property:

For any $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n \in \mathcal{X}$, the $n \times n$ matrix whose (i,j)-th entry is $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is positive semidefinite.

For any kernel K, there is a feature mapping $\boldsymbol{\phi}\colon \mathcal{X} \to \mathbb{H}$ such that

$$\langle \boldsymbol{\phi}(\boldsymbol{x}), \boldsymbol{\phi}(\boldsymbol{x}') \rangle \;=\; K(\boldsymbol{x}, \boldsymbol{x}'),$$

($\mathbb{H}$ is a Hilbert space—i.e., a special kind of inner product space—called the *Reproducing Kernel Hilbert Space corresponding to* K.)

String kernels

- $\boldsymbol{\phi}$: Strings $\to \mathbb{N}^{\text{Strings}}$, where

$$\boldsymbol{\phi}(\boldsymbol{x}) \;=\; (\text{number of times } \boldsymbol{s} \text{ appears in } \boldsymbol{x} \;:\; \boldsymbol{s} \in \text{Strings})$$

$$K(\boldsymbol{x}, \boldsymbol{x}') \;=\; \langle \boldsymbol{\phi}(\boldsymbol{x}), \boldsymbol{\phi}(\boldsymbol{x}') \rangle \;=\; \text{measure of similarity between strings}\,.$$

- **Computing** $K(\boldsymbol{x}, \boldsymbol{x}')$:
  For each substring $\boldsymbol{s}$ of $\boldsymbol{x}$, count how many times $\boldsymbol{s}$ appears in $\boldsymbol{x}'$ and add to total.

  Dynamic programming in time $O(\text{length}(\boldsymbol{x}) \times \text{length}(\boldsymbol{x}'))$.

Implicit representation of weight vector

- **Implicit representation**:

$$\boldsymbol{w} \;=\; \sum_{(\boldsymbol{x},y) \in \mathcal{M}} y \boldsymbol{\phi}(\boldsymbol{x})$$

- Never explicitly form weight vector $\boldsymbol{w}$.
- Instead, store all labeled examples in $\mathcal{M}$.
- Whenever need to compute $\langle \boldsymbol{w}, \boldsymbol{\phi}(\boldsymbol{z}) \rangle$ for new point $\boldsymbol{z}$, iterate over examples in $\mathcal{M}$ to compute

$$\langle \boldsymbol{w}, \boldsymbol{\phi}(\boldsymbol{z}) \rangle \;=\; \sum_{(\boldsymbol{x},y) \in \mathcal{M}} y \langle \boldsymbol{\phi}(\boldsymbol{x}), \boldsymbol{\phi}(\boldsymbol{z}) \rangle \;=\; \sum_{(\boldsymbol{x},y) \in \mathcal{M}} y K(\boldsymbol{x}, \boldsymbol{z})\,.$$

The kernel approach

- Focus on designing good kernels (rather than feature maps), which means designing good **similarity functions**.
- Lots of ways to construct kernels.
  (E.g., combine existing kernels.)
- Lots of algorithms can be / have been “kernelized” (like Perceptron).

Experimental results on OCR

- OCR digits data, binary classification problem: distinguish "9" from other digits.
- # training examples: 60000 (about 6000 are from class "9").
- Test error rates using Kernelized Averaged Perceptron (similar to Voted Perceptron).

# passes	0.1	1	2	3	4	10
Linear kernel	0.045	0.039	0.038	0.038	0.038	0.037
Degree 2	0.024	0.012	0.010	0.010	0.009	0.009
Degree 4	0.020	0.009	0.008	0.007	0.007	0.006

More computational issues

- **Implicit representation**:

$$\boldsymbol{w} = \sum_{(\boldsymbol{x},y)\in\mathcal{M}} y\boldsymbol{\phi}(\boldsymbol{x})$$

- Number of mistakes $|\mathcal{M}|$ could be $\Omega(n)$.
- Computing predictions as expensive as brute-force NN search.
- Training algorithms quite slow (e.g., $\Omega(n^2)$).

Kernel approximations

Many ways to try to speed-up kernel methods using approximations.

- Limit number of examples used to represent weight vector.
  "Nystrom approximation"
  "Budgeted Perceptron"
- Use *explicit* feature maps $\boldsymbol{z}\colon \mathbb{R}^d \to \mathbb{R}^m$ such that

$$\langle \boldsymbol{z}(\boldsymbol{x}), \boldsymbol{z}(\boldsymbol{x}')\rangle \approx K(\boldsymbol{x},\boldsymbol{x}')\,.$$

  "Random projections / feature hashing"
  "Random kitchen sinks"

Experimental results on e-mail data

- Spam data set (4601 e-mail messages, 39.4% are spam).
- $\mathcal{Y} = \{$spam, not spam$\}$, $\mathcal{X} = \mathbb{R}^{57}$ (message features)
- # training examples: 3065, # test examples: 1536
- Test error rates
  - **Decision tree learning**: 9.3%
  - **Averaged Perceptron** (128 passes): 8.27%
  - **Random Kitchen Sink Averaged Perceptron** (64 passes): 6.12% (approximates Gaussian kernel)

Key takeaways

1. Linear classifiers only as good as given feature representation.
2. Explicit feature expansion (sometimes okay!)
3. Kernel trick: sometimes never need $\boldsymbol{\phi}(\boldsymbol{x})$ directly, but only via $\langle \boldsymbol{\phi}(\boldsymbol{x}), \boldsymbol{\phi}(\boldsymbol{x}') \rangle$, computed quickly as $K(\boldsymbol{x}, \boldsymbol{x}')$.
4. Kernel approach: "similarity engineering" rather than "feature engineering"
5. High-level idea of using kernel approximations.