Online Perceptron

**Online Perceptron**

**Input** Labeled examples \( \{(x_i, y_i)\}_{i=1}^n \) from \( \mathbb{R}^d \times \{-1, +1\} \).

1. initialize \( \hat{w}_1 := 0 \).
2. for \( t = 1, 2, \ldots, n \) do
3. if \( y_t \langle \hat{w}_t, x_t \rangle \leq 0 \) then
4. \( \hat{w}_{t+1} := \hat{w}_t + y_t x_t \).
5. else
6. \( \hat{w}_{t+1} := \hat{w}_t \)
7. end if
8. end for
9. return \( \hat{w}_{n+1} \).

**Theorem:** If \( R := \max_{t \in \{1, \ldots, n\}} \|x_t\|_2 \), and \( w_* \in \mathbb{R}^d \) satisfies

\[
y \langle w_*, x_t \rangle \geq 1 \quad \text{for all} \quad (x_t, y_t),
\]

then Online Perceptron makes at most \( \|w_*\|_2^2 \cdot R^2 \) mistakes (and updates).
Example run of Online Perceptron
Example run of Online Perceptron

Adding new features

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

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:

1 {“aardvark” appears}, 1 {“abacus” appears}, …, 1 {“zygote” appears}

▶ Bi-gram features:

1 {“bank deposit” appears}, 1 {“river bank” appears}, …

▶ Tri-gram features:

1 {“New York City” appears}, 1 {“wherefore art though” appears}, …

Example: new features from old features \( x \in \mathbb{R}^d \)

▶ Pairwise interactions:

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

▶ Etc.

All degree \( \leq 2 \) interaction features

Learning with the kitchen sink of features

▶ Let \( \phi: \mathbb{R}^d \rightarrow \mathbb{R}^D \) 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 \( f_w: \mathbb{R}^D \rightarrow \{\pm 1\} \) (i.e., learn a weight vector \( w \in \mathbb{R}^D \)) using data with expanded features \( \{(\phi(x_i), y_i)\}_{i=1}^n \).

▶ Caveat: can be computationally expensive to do this directly if \( D \) is large. Naively: takes \( \Omega(D) \) time to even make a prediction.
Kernel trick

- **Recall**: Online Perceptron weight vector (using expanded features) is
  \[ w = \sum_{(x,y) \in M} y\phi(x) \]
  where \( M \) is the subset of labeled examples where Online Perceptron made a mistake during training.

- **Prediction** using Online Perceptron weight vector on new point \( z \in \mathbb{R}^d \):
  \[ \langle w, \phi(z) \rangle = \sum_{(x,y) \in M} y \langle \phi(x), \phi(z) \rangle \]
  
  **Computational cost**: \( |M| \times \text{time to compute inner product } \langle \phi(x), \phi(z) \rangle \).
  Sometimes this can be faster than \( O(D) \).

Degree \( \leq 2 \) interaction features

- \( \phi : \mathbb{R}^d \rightarrow \mathbb{R}^{1+2d+\binom{d}{2}} \), where
  \[ \phi(x) = (1, \sqrt{2}x_1, \ldots, \sqrt{2}x_d, x_1^2, \ldots, x_d^2, \sqrt{2}x_1x_2, \ldots, \sqrt{2}x_1x_d, \ldots, \sqrt{2}x_{d-1}x_d) \]
  (Don’t mind the \( \sqrt{2} \)'s.)

- **Computing** \( \langle \phi(x), \phi(x') \rangle \) in \( O(d) \) time:
  \[ \langle \phi(x), \phi(x') \rangle = (1 + \langle x, x' \rangle)^2. \]

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

Products of all feature subsets

- \( \phi : \mathbb{R}^d \rightarrow \mathbb{R}^{2d} \), where
  \[ \phi(x) = \left( \prod_{i \in S} x_i : S \subseteq \{1, 2, \ldots, d\} \right) \]

- **Computing** \( \langle \phi(x), \phi(x') \rangle \) in \( O(d) \) time:
  \[ \langle \phi(x), \phi(x') \rangle = \prod_{i=1}^d (1 + x_i x'_i) . \]

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

Infinite dimensional feature expansion

- For any \( \sigma > 0 \), there is an infinite feature expansion \( \phi : \mathbb{R}^d \rightarrow \mathbb{R}^\infty \) (involving Hermite polynomials of all orders) such that
  \[ \langle \phi(x), \phi(x') \rangle = \exp \left( -\frac{\|x - x'\|^2}{2\sigma^2} \right) , \]
  which can be computed in \( O(d) \) time.

  (This is called the **Gaussian kernel** with bandwidth \( \sigma \).)
A kernel function $K : X \times X \rightarrow \mathbb{R}$ is a symmetric function with the following property:

For any $x_1, x_2, \ldots, x_n \in X$, the $n \times n$ matrix whose $(i, j)$-th entry is $K(x_i, x_j)$ is positive semidefinite.

For any kernel $K$, there is a feature mapping $\phi : X \rightarrow \mathbb{H}$ such that

$$\langle \phi(x), \phi(x') \rangle = K(x, x') ,$$

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

### String kernels

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

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

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

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

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

### Implicit representation of weight vector

- Implicit representation:

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

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

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

- 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).

<table>
<thead>
<tr>
<th># passes</th>
<th>0.1</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear kernel</td>
<td>0.045</td>
<td>0.039</td>
<td>0.038</td>
<td>0.038</td>
<td>0.038</td>
<td>0.037</td>
</tr>
<tr>
<td>Degree 2</td>
<td>0.024</td>
<td>0.012</td>
<td>0.010</td>
<td>0.009</td>
<td>0.009</td>
<td></td>
</tr>
<tr>
<td>Degree 4</td>
<td>0.020</td>
<td>0.009</td>
<td>0.008</td>
<td>0.007</td>
<td>0.007</td>
<td>0.006</td>
</tr>
</tbody>
</table>

More computational issues

▶ Implicit representation:

\[ w = \sum_{(x,y) \in M} y \phi(x) \]

▶ Number of mistakes \( |M| \) 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 \( z: \mathbb{R}^d \rightarrow \mathbb{R}^m \) such that

\[ (z(x), z(x')) \approx K(x, x') . \]

“Random projections / feature hashing”
“Random kitchen sinks”

Experimental results on e-mail data

▶ Spam data set (4601 e-mail messages, 39.4% are spam).
▶ \( Y = \{ \text{spam, not spam} \} \), \( X = \mathbb{R}^{57} \) (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)
1. Linear classifiers only as good as given feature representation.
2. Explicit feature expansion (sometimes okay!)
3. Kernel trick: sometimes never need \( \phi(x) \) directly, but only via \( \langle \phi(x), \phi(x') \rangle \), computed quickly as \( K(x, x') \).
5. High-level idea of using kernel approximations.