Linear Classifiers

CS178
Intro to Machine Learning & Data Mining
Prof. Alexander Ihler

Lecture 07
Regression vs. Classification

Regression

Features \( x \)
Real-valued target \( t \)

Predict continuous function \( \hat{y}(x) \)

Classification

Features \( x \)
Discrete class \( c \)
(usually 0/1 or +1/-1)

Predict discrete function \( \hat{y}(x) \)
Linear Classifiers

- Linear Classifiers
  - a linear classifier is a mapping which partitions feature space using a linear function (a straight line, or a hyperplane)
  - separates the two classes using a straight line in feature space
  - in 2 dimensions the decision boundary is a straight line

Linearly separable data

Linearly non-separable data"
Linear Classifiers: Parametric Form

- Let: feature 1 = “X1”, feature 2 = “X2”

- Linear classifier is a linear function of features X1 and X2, i.e.,
  - \( f(X1,X2) = a \times X1 + b \times X2 + c \)
  - Coefficients \([a, b, c]\) are the “weights” / “parameters” of the classifier
  - In general, \(d + 1\) coefficients (one for each feature, plus offset)

- Output of the classifier is a class, \([-1, 1]\) :
  - \( T(f) = -1 \) if \( f < 0 \), \( T(f) = +1 \) if \( f > 0 \)

- Decision boundary
  - Transition from one class decision to another at \( f(X1,X2) = 0 \)
  - Decision boundary is: \( a \times X1 + b \times X2 + c = 0 \) — Linear

- In higher dimensions, equation is a “hyperplane”
Perceptron Classifier (2 features)

Classifier

\[ f = w_1 X_1 + w_2 X_2 + w_0 \]

weighted sum of the inputs

Threshold Function

\[ T(f) \]

\[ \hat{C}(x) \]

\{-1, +1\}

output = class decision
Perceptrons

- Perceptron = a linear classifier
  - The w’s are the weights (denoted as a, b,c, earlier)
    - real-valued constants (can be positive or negative)
  - Define an additional constant input “1” (allows an intercept in decision boundary)

- A perceptron calculates 2 quantities:
  - 1. A weighted sum of the input features
  - 2. This sum is then thresholded by the T function

- A simple artificial model of human neurons
  - weights = “synapses”
  - threshold = “neuron firing”
Notation

• Inputs:
  – $x_0, x_1, x_2, \ldots, x_d$,
  – $x_1, x_2, \ldots, x_{d-1}, x_d$ are the values of the $d$ features
  – $x_0 = 1$ (a constant input)
  – $\mathbf{x} = (x_0, x_1, x_2, \ldots, x_d)$

• Weights:
  – $w_0, w_1, w_2, \ldots, w_d$,
  – we have $d+1$ weights
  – one for each feature + one for the constant
  – $\mathbf{w} = (w_0, w_1, w_2, \ldots, w_d)$
Perceptron Operation

- Equations of operation:

\[
\begin{align*}
o[x_1, x_2, \ldots, x_{d-1}, x_d] &= 1 \quad \text{(if } w_1 x_1 + \ldots + w_d x_d + w_0 > 0) \\
&= -1 \quad \text{(otherwise)}
\end{align*}
\]

Note that \( w = (w_0, \ldots, w_d) \), the “weight vector” (row vector, 1 x d+1)

and \( x = (x_0, \ldots, x_d) \), the “feature vector” (row vector, 1 x d+1)

\[
\Rightarrow w_0 x_0 + w_1 x_1 + \ldots + w_d x_d = \overrightarrow{w} \cdot \overrightarrow{x}
\]

and \( \overrightarrow{w} \cdot \overrightarrow{x} \) is the vector inner product (w*x’ or “sum(w.*x)” in MATLAB)
Perceptron Decision Boundary

- Equations of operation (in vector form):

\[ o(x_1, x_2, \ldots, x_d, x_{d+1}) = \begin{cases} 
1 & \text{if } \mathbf{w} \cdot \mathbf{x}' > 0 \\
-1 & \text{otherwise}
\end{cases} \]

The perceptron represents a hyperplane decision surface in d-dimensional space
  e.g., a line in 2d, a plane in 3d, etc

The equation of the hyperplane is
\[ \mathbf{w} \cdot \mathbf{x}' = 0 \]

This is the equation for points in x-space that are on the boundary
Example, Linear Decision Boundary

\[ w = (w_1, w_2, w_0) = (1, -1, 0) \]
Example, Linear Decision Boundary

\[
\mathbf{w} = (w_1, w_2, w_0) = (1, -1, 0)
\]

\[
\mathbf{w} \cdot \mathbf{x}^j = 0
\]

\[
\Rightarrow 1 \cdot x_1 - 1 \cdot x_2 + 0 \cdot 1 = 0
\]

\[
\Rightarrow x_1 - x_2 = 0
\]

\[
\Rightarrow x_1 = x_2
\]
Example, Linear Decision Boundary

\[ \mathbf{w} = (w_1, w_2, w_0) = (1, -1, 0) \]

\[ \mathbf{w} \cdot \mathbf{x}' = 0 \]

\[ \Rightarrow 1 \cdot x_1 - 1 \cdot x_2 + 0.1 = 0 \]

\[ \Rightarrow x_1 - x_2 = 0 \]

\[ \Rightarrow x_1 = x_2 \]

This is the equation for the decision boundary.
Example, Linear Decision Boundary

$\mathbf{w} \cdot \mathbf{x}' < 0$

$\Rightarrow x_1 - x_2 < 0$

$\Rightarrow x_1 < x_2$

(this is the equation for decision region -1)

$\mathbf{w} = (w_1, w_2, w_0)$

$= (1, -1, 0)$

$\mathbf{w} \cdot \mathbf{x}' = 0$
Representational Power of Perceptrons

- What mappings can a perceptron represent perfectly?
  - A perceptron is a linear classifier
  - thus it can represent any mapping that is linearly separable
  - some Boolean functions like AND (on left)
  - but not Boolean functions like XOR (on right)
Dimensionality & Extensions of Linear Methods

• Linear classifier is one of the “simplest” parametric forms
  – Can be easily extended to more complex decision boundaries

• Imagine a “quadratic” classifier
  – \( f(X,Y) = a*X^2 + b*Y^2 + c*X + d*Y + e \)
  – Decision boundary can be: parabola, ellipse, …

• Notice: \( f(X,Y) \) is quadratic in \( X, Y \)
  – but linear in \([X^2, Y^2, X, Y]\)

• Equivalence by “artificially” increasing the number of features:
  – Nonlinear method in original feature space
  – Linear method in higher dimensional feature space

• We saw this before in regression…
Representational Power of Perceptrons

- What mappings can a perceptron represent perfectly?
  - A perceptron is a linear classifier
  - thus it can represent any mapping that is linearly separable
  - some Boolean functions like AND (on left)
  - but not Boolean functions like XOR (on right)

What kinds of functions would we need to learn the data on the right?
Representational Power of Perceptrons

- What mappings can a perceptron represent perfectly?
  - A perceptron is a linear classifier
  - thus it can represent any mapping that is linearly separable
  - some Boolean functions like AND (on left)
  - but not Boolean functions like XOR (on right)

What kinds of functions would we need to learn the data on the right?
Effect of dimensionality

- Data are increasingly separable in high dimension – is this a good thing?

- “Good”
  - Separation is easier in higher dimensions (for fixed $N$)
  - Increase the number of features, and even a linear classifier will eventually be able to separate all the training examples!

- “Bad”
  - Remember training vs. test error? Remember overfitting?
  - Increasingly complex decision boundaries can eventually get all the training data right, but it doesn’t necessarily bode well for test data…

![Diagram showing predictive error vs. complexity](image-url)
Learning the Classifier Parameters

• Where do the parameters (weights) of the classifier come from?
  – If we know a lot about the problem, we could “design” them
  – Typically we don’t know ahead of time what the values should be

• Learning from Training Data:
  – training data = labeled feature vectors
  – i.e., a set of N feature vectors each with a class label
  – we can use the training data to try to find good parameters
  – “good” parameters are ones which provide low error
    • error is estimated on the training data
    • “true” error will be on future test data
  – Statement of the Learning Problem:
    • given a classifier, and some training data, find the values for the classifier’s parameters which maximize training accuracy
Learning the Weights from Data

An Example of a Training Data Set

<table>
<thead>
<tr>
<th>Example</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>....</th>
<th>$x_d$</th>
<th>true class label, $y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x(1)$</td>
<td>3.4</td>
<td>-1.2</td>
<td>....</td>
<td>7.1</td>
<td>1</td>
</tr>
<tr>
<td>$x(2)$</td>
<td>4.1</td>
<td>-3.1</td>
<td>....</td>
<td>4.6</td>
<td>-1</td>
</tr>
<tr>
<td>$x(3)$</td>
<td>5.7</td>
<td>-1.0</td>
<td>....</td>
<td>6.2</td>
<td>-1</td>
</tr>
<tr>
<td>$x(4)$</td>
<td>2.2</td>
<td>4.1</td>
<td>....</td>
<td>5.0</td>
<td>1</td>
</tr>
<tr>
<td>$x(n)$</td>
<td>1.2</td>
<td>4.3</td>
<td>....</td>
<td>6.1</td>
<td>1</td>
</tr>
</tbody>
</table>
Learning as a Search Problem

- The objective function:
  - the accuracy of the classifier (for a given set of weights $w$ and labeled data)

- Problem:
  - maximize this objective function

- Equivalent to an optimization or search problem
  - i.e., think of the vector $(w_1, w_2, w_0)$
  - this defines a 3-dimensional “parameter space”
  - we want to find the value of $(w_1, w_2, w_0)$ which maximizes the objective function
  - So we could use hill-climbing, systematic search, etc., to search this parameter space
    - many learning algorithms = hill-climbing with random restarts
Classification Accuracy

- Say we have N feature vectors (training data) for which we know the true class label of each feature vector.

- We can measure how accurate a classifier is by how many feature vectors it classifies correctly.

- Procedure:
  - accuracy = 0
  - For each of the N feature vectors:
    - calculate the output of the classifier for this vector
    - if the class label agrees with the true label
      - ++ accuracy
    - continue
  - Percentage Accuracy = (accuracy/N) * 100%
Training Data and Learning

• We have N examples
  – an example consists of a feature vector and a class label (target)
  – the ith feature vector is denoted $x(i)$

• Learning on the Training Data
  – Let $\hat{y}(i) = \hat{y}(x(i))$ be the output of a perceptron for the ith feature vector $x(i)$
  – Let $y(i)$ be the target value (true class)
  – goal is to find perceptron weights such that
    • $\hat{y}[i]$ is close to $y(i)$ for as many examples as possible
    • i.e., the output matches the desired target as often as possible

$$\text{TrainingAccuracy}(w) = \frac{1}{N} \sum \delta( \hat{y}(i) = y(i) )$$

where $\delta( \hat{y}(i) = y(i) ) = 1$ if $\hat{y}(i) = y(i)$, and 0 otherwise
Equivalent Parameter Settings?

- Which decision boundary is “better”?  
  - Both have zero training error (perfect training accuracy)  
  - But, one of them seems intuitively better…

- How can we quantify “better”, and learn the “best” parameter settings?
More on Classifier Quality

• Examples of measuring classifier quality

• Mini-max approaches
  – Maximize the min. distance to decision boundary
  – Example: Support Vector Machines (SVM)
  – Very popular method
More on Classifier Quality

- Average behavior approaches
  - This is what we’ll use for now
  - “Approximate” the threshold function

- Usually some smooth function of distance
  - Example: “sigmoid”, looks like an “S”

- Now, measure average similarity to true class labels
  - Example: Mean squared error
  
  \[
  E_{\sigma}(w) = \left(\frac{1}{N}\right) \sum_i \left( \sigma(f(x_i)) - t(i) \right)^2
  \]

- Far from the decision boundary: |f(.)| large, small error
- Nearby the boundary: |f(.)| near zero, larger error

- Notice: this is a “smooth” error function
  - Easier to train / optimize the choice of parameters…
Linear Classifier (2 features)

\[ f = w_1 X_1 + w_2 X_2 + w_0 \]

Threshold Function

Output \( \hat{c}(x) \)

\{0, 1\}

Decision boundary = “x such that \( T( w_1 x + w_0 ) \) transitions”
Training a linear classifier

• How should we measure error?
  – Natural measure = “fraction we get wrong”

• But, hard to train via gradient descent
  – Not continuous
  – As decision boundary moves, errors change abruptly

1D example:

\[
\sum \delta( y(j) \neq \hat{y}(j) ) = \frac{1}{2} \sum ( y(j) - \hat{y}(j) )^2
\]

- If we only guess +1/-1, this is one-half the MSE…

\[
T(f) = -1 \text{ if } f < 0 \\
T(f) = +1 \text{ if } f > 0
\]
Training a linear classifier

• “Online” gradient descent
  - Perform a gradient update one data point at a time
    - For each data point j, predict, calculate error, modify parameters; repeat

• Perceptron algorithm
  - For each data point j:
    \[ \hat{y}(j) = T( w \cdot x(j) ) \] : predict output for data point j
    \[ w \leftarrow w + \alpha ( y(j) - \hat{y}(j) ) x(j) \] : “gradient-like” step
  - Converges if data are linearly separable
Perceptron algorithm

- For each data point $j$:
  \[
  \hat{y}(j) = T( \mathbf{w} \cdot \mathbf{x}(j) ) \quad : \text{predict output for data point } j
  \]
  \[
  \mathbf{w} \leftarrow \mathbf{w} + \alpha ( y(j) - \hat{y}(j) ) \mathbf{x}(j) \quad : \text{“gradient-like” step}
  \]

$y(j)$ predicted incorrectly:
update weights
Perceptron algorithm

- Perceptron algorithm
  - For each data point $j$:
    
    $\hat{y}(j) = T\left(w \ast x(j)\right)$ : predict output for data point $j$
    
    $w \leftarrow w + \alpha \left(y(j) - \hat{y}(j)\right) x(j)$ : “gradient-like” step

$y(j)$ predicted correctly: no update
Perceptron algorithm

- For each data point $j$:
  \[ \hat{y}(j) = T( w \cdot x(j) ) \quad : \text{predict output for data point } j \]
  \[ w \leftarrow w + \alpha \left( y(j) - \hat{y}(j) \right) x(j) \quad : \text{“gradient-like” step} \]

- Converges if data are linearly separable
Training a linear classifier

- One solution is to use a “smooth” threshold
  - “Approximate” the threshold function
  - Usually some smooth function of distance
    - Example: “sigmoid”, looks like an “S”

- Now, measure e.g. MSE

\[
E_\sigma(w) = \frac{1}{N} \sum_i \left( \sigma(f(x_i)) - t(i) \right)^2
\]

- Far from the decision boundary: \(|f(\cdot)|\) large, small error
- Nearby the boundary: \(|f(\cdot)|\) near 1/2, larger error

- Notice: this is a “smooth” error function
  - Easier to train / optimize the choice of parameters…
Training a linear classifier

• How should we measure error?
  – Natural measure = “fraction we get wrong”
  – If we only guess 0/1, this is also the MSE…

• But, hard to train via gradient descent
  – Not continuous
  – As decision boundary moves, errors change abruptly

• Using linear regression & MSE
  – Easy to train by gradient descent
  – But, not optimizing the right error – e.g., outliers

1D example:

Classification error = MSE = 2/9

\[
\text{MSE} = (0^2 + 1^2 + .2^2 + .25^2 + .05^2 + \ldots)/9
\]
Training the Classifier

• Once we have a smooth measure of quality, we can find the “best” settings for the parameters of
  \( f(X_1, X_2) = aX_1 + bX_2 + c \)

• Example: 2D feature space \( \Leftrightarrow \) parameter space

\[ \text{MSE} = 1.9 \]
Training the Classifier

• Once we have a smooth measure of quality, we can find the “best” settings for the parameters of 
  \[ f(X_1, X_2) = aX_1 + bX_2 + c \]

• Example: 2D feature space \(\leftrightarrow\) parameter space

\[ \text{MSE} = 0.4 \]
Training the Classifier

- Once we have a smooth measure of quality, we can find the “best” settings for the parameters of $f(X_1,X_2) = a*X_1 + b*X_2 + c$
- Finding the minimum MSE in parameter space…

- $[a \ b \ c] = ?$
- $[\arctan(A/B), c] = [-\pi/4, \ 1]$
Training the Classifier

- Once we have a smooth measure of quality, we can find the “best” settings for the parameters of $f(X_1, X_2) = aX_1 + bX_2 + c$
- Finding the minimum MSE in parameter space...

- $[a \ b \ c] = ?$
- $[\arctan(A/B), c] = [-\pi/4, 1]$
Finding the Best MSE

- As in linear regression, this is now just optimization

- Methods:
  - Gradient descent
    - Improve MSE by small changes in parameters ("small" = learning rate)
  - Or, substitute your favorite optimization algorithm…
    - Coordinate descent
    - Stochastic search
    - Genetic algorithms
Gradient Equations

- MSE (note, depends on function $\sigma(.)$)

$$C(w = [a, b, c]) = \frac{1}{N} \sum_i \left( \sigma(ax_1^{(i)} + bx_2^{(i)} + c) - y^{(i)} \right)^2$$

- What’s the derivative with respect to one of the parameters?

$$\frac{\partial C}{\partial a} = \frac{1}{N} \sum_i 2 \left( \sigma(w \cdot x) - y^{(i)} \right) \partial \sigma(w \cdot x) x_1^{(i)}$$

  Error between class and prediction

  Sensitivity of prediction to changes in parameter “a”

- Similar for parameters $b, c$ [replace $x_1$ with $x_2$ or 1 (constant)]
Saturating Functions

- Many possible “saturating” functions

- “Logistic” sigmoid (scaled for range [0,1]) is

  \[ \sigma(x) = \frac{1}{1 + \exp(-x)} \]

- Derivative is

  \[ \partial \sigma(x) = \sigma(x) (1-\sigma(x)) \]

- Matlab Implementation:

  ```matlab
  function s = sig(x)
  % value of [0,1] sigmoid
  s = 1 ./ (1+exp(-x));
  
  function ds = dsig(x)
  % derivative of (scaled) sigmoid
  ds = sig(x) .* (1-sig(x));
  ```
Gradient Decent Algorithm (**BATCH**)

- **Algorithm outline**
  - Initialize the weights (e.g., randomly)
  - Loop “until convergence”
    - for each example calculate the output
    - calculate the difference between the output and the target
    - update each of the \(d+1\) weights using the gradient update rule
      \[
      w_j \leftarrow w_j - \eta \left( \frac{\partial E}{\partial w_j} \right)
      \]
  - Convergence condition:
    - when change in MSE is sufficiently small, stop iterating
  - Halt and return weights
Incremental Training Algorithm

• “Incremental Gradient Descent” – **online** version
• Often faster than batch gradient algorithm

Algorithm outline

– initialize the weights (e.g., randomly)

– loop through all N examples (this is 1 iteration)
  • for each example calculate the output
  • calculate the difference between the output and the target
  • update each of the d+1 weights using the **single example** gradient update rule
    – Like the full gradient, but only involves one training example

– after all N examples are gone through
  • check if the overall error (MSE) has decreased significantly since the previous iteration
  • if not, then perform another iteration through all N examples
  • if so, then halt and return weights
Gradient Descent Learning Rule

- Online (single-example) weight update rule:

\[
\begin{align*}
    w_j &\leftarrow w_j + \eta \left( t(i) - \sigma(f(i)) \right) \frac{\partial\sigma(f(i))}{\partial f(i)} x_j(i)
\end{align*}
\]

- \( t(i) \) is the target class of the \( i \)th training example
- \( f(i) \) is the weighted sum (respectively) for the \( i \)th example
- \( w_j \) is the \( j \)th input weight
- \( x_j(i) \) is the \( j \)th input feature value, for the \( i \)th example
- \( \eta \) is called the learning rate: a small positive number, \( 0 < \eta < 1 \)

- An example of how this works:
  - Say \( w_j \) and \( x_j(i) \) are both positive:
    - say \( t(i) > f(i) \) => we increase the value of the weight
    - say \( t(i) < f(i) \) => we decrease the value of the weight
    - \( \eta \) controls how quickly we increase or decrease the weight
Pseudocode for Logistic Regression

Initialize each weight (e.g., randomly)

iteration=0;
While (convergence_criterion not achieved)
  for i=1:N
    calculate the output of the network for example i
    for j = 1: d+1
      update weight j using the update rule
    end
  end
  calculate convergence_criterion
  ++ iteration
  (optional) plot current location of decision boundary
end
Summary

- Linear classifier $\Leftrightarrow$ perceptron

- Visualizing the decision boundary

- Measuring quality of a decision boundary
  - MSE criterion

- Learning the weights of a linear classifier from data
  - Reduces to an optimization problem
  - For MSE (and some others) we can do gradient descent
  - Batch gradient descent vs. Incremental gradient descent
  - Gradient equations & update rules