Lecture 7b - Supervised Machine Learning II

Jesse Hoey
School of Computer Science
University of Waterloo

February 8, 2019

Readings: Poole & Mackworth (2nd ed.) Chapt. 7.3.2, 7.5-7.6
Linear Regression

**Linear regression** is where the output is a linear function of the input features.

\[
\hat{Y}(e) = w_0 + w_1 X_1(e) + \cdots + w_n X_n(e)
\]

\[
\hat{Y}(e) = \sum_{i=0}^{n} w_i X_i(e)
\]

where \( \mathbf{w} = \langle w_0, w_1, w_2, \ldots, w_n \rangle \). We invent a new feature \( X_0 \) which has value 1, to make it not a special case.
Linear Regression

**Linear regression** is where the output is a linear function of the input features.

\[
\hat{Y}(e) = w_0 + w_1 X_1(e) + \cdots + w_n X_n(e)
\]

\[
\hat{Y}(e) = \sum_{i=0}^{n} w_i X_i(e)
\]

where \( \overline{w} = \langle w_0, w_1, w_2, \ldots, w_n \rangle \). We invent a new feature \( X_0 \) which has value 1, to make it not a special case.

The sum of squares error on examples \( E \) for output \( Y \) is:

\[
Error(E, \overline{w}) = \sum_{e \in E} (Y(e) - \hat{Y}(e))^2
\]

\[
= \sum_{e \in E} (Y(e) - \sum_{i=0}^{n} w_i X_i(e))^2
\]

Goal: find weights that minimize \( Error(E, \overline{w}) \).
Finding weights that minimize $\text{Error}(E, \vec{w})$

Find the minimum analytically. Effective when it can be done (e.g., for linear regression). If

- $\vec{y} = [Y(e_1), Y(e_2), \ldots, Y(e_M)]$ is a vector of the output features for the $M$ examples
- $X$ is a matrix where the $j^{th}$ column is the values of the input features for the $j^{th}$ example
- $\vec{w} = [w_0, w_1, \ldots, w_n]$ is a vector of the weights

then,

\[
\vec{y}^T = \vec{w}X
\]

\[
\vec{y}^T X^T (XX^T)^{-1} = \vec{w}
\]
Finding weights that minimize $Error_E(\overline{w})$

Find the minimum iteratively.
Works for larger classes of problems (not just linear).
Gradient descent:

$$w_i \leftarrow w_i - \eta \frac{\partial Error(E, \overline{w})}{\partial w_i}$$

$\eta$ is the gradient descent step size, the learning rate.

If

$$Error(E, \overline{w}) = \sum_{e \in E} (Y(e) - \hat{Y}^\overline{w}(e))^2 = \sum_{e \in E} \left( Y(e) - \sum_{i=0}^{n} w_i X_i(e) \right)^2$$

then

$$w_i \leftarrow w_i + \eta \sum_{e \in E} \left( Y(e) - \sum_{i=0}^{n} w_i X_i(e) \right) X_i(e)$$

where we have set $\eta \rightarrow 2\eta$ (arbitrary scale)
Incremental Gradient Descent for Linear Regression

1: procedure LinearLearner($X, Y, E, \eta$)
2: Inputs $X$: set of input features, $X = \{X_1, \ldots, X_n\}$
3: $Y$: output feature
4: $E$: set of examples from which to learn
5: $\eta$: learning rate
6: initialize $w_0, \ldots, w_n$ randomly
7: repeat
8: for each example $e$ in $E$ do
9: $\delta \leftarrow Y(e) - \sum_{i=0}^{n} w_i X_i(e)$
10: for each $i \in [0, n]$ do
11: $w_i \leftarrow w_i + \eta \delta X_i(e)$
12: until some stopping criteria is true
13: return $w_0, \ldots, w_n$
Stochastic and Batched Gradient Descent

- Algorithm on the last slide is **incremental** gradient descent.
- If examples are chosen randomly at line 8 then it's **stochastic gradient descent**.
- **Batched gradient descent**:  
  - process a batch of size $n$ before updating the weights  
  - if $n$ is all the data, then it's **gradient descent**  
  - if $n = 1$, it's **incremental gradient descent**
- Incremental can be more efficient than batch, but convergence not guaranteed.
Assume we are doing binary classification, with classes \( \{0, 1\} \).

There is no point in making a prediction of less than 0 or greater than 1.

A **squashed linear function** is of the form:

\[
\hat{Y}^w(e) = f(w_0 + w_1X_1(e) + \cdots + w_nX_n(e))
\]

\[
= f\left(\sum_{i=0}^{n} w_iX_i(e)\right)
\]

where \( f \) is an **activation function**.

A simple activation function is the step function:

\[
f(x) = \begin{cases} 
1 & \text{if } x \geq 0 \\
0 & \text{if } x < 0
\end{cases}
\]
Gradient Descent for Linear Classifiers

If the activation is differentiable, we can use gradient descent to update the weights. The sum of squares error is:

$$\text{Error}(E, \overline{w}) = \sum_{e \in E} (Y(e) - f(\sum_{i=0}^{n} w_i \ast X_i(e)))^2$$

The partial derivative with respect to weight $w_i$ is:

$$\frac{\partial \text{Error}(E, \overline{w})}{\partial w_i} = -2 \ast \delta \ast f'(\sum_{i} w_i \ast X_i(e)) \ast X_i(e)$$

where $\delta = (Y(e) - f(\sum_{i=0}^{n} w_i X_i(e)))$.

Thus, each example $e$ updates each weight $w_i$ by

$$w_i \leftarrow w_i + \eta \ast \delta \ast f'(\sum_{i} w_i \ast X_i(e)) \ast X_i(e)$$
The sigmoid or logistic activation function is given by:

\[ f(x) = \frac{1}{1 + e^{-x}} \]
The sigmoid or logistic activation function

\[ f(x) = \frac{1}{1 + e^{-x}} \]

\[ f'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = f(x)(1 - f(x)) \]
\[ \hat{\text{Reads}}(e) = \text{sigmoid}(-8 + 7 \times \text{Short}(e) + 3 \times \text{New}(e) + 3 \times \text{Known}(e)) \]

Can be found in about 3000 iterations with a learning rate of \( \eta = 0.05 \)
A classification is **linearly separable** if there is a hyperplane where the classification is true on one side of the hyperplane and false on the other side.

The hyperplane is defined by where the predicted value, $f^w(X_1, \ldots, X_n) = f(w_0 + w_1X_1(e) + \cdots + w_nX_n(e))$ is 0.5. For the sigmoid function, the hyperplane is defined by $w_0 + w_1X_1(e) + \cdots + w_nX_n(e) = 0$.

Some data are not linearly separable.
More general: Discriminant

Linearly Separable

Y

X
More general: Discriminant

Multi-Linearly Separable
More general: Discriminant

?? Separable?
More general: Discriminant
Kernel Trick

Some arbitrary data:
Kernel Trick

Data is not linearly separable:
Kernel Trick

Add another dimension, data is now linearly separable:

\[ y = \text{rem}(x/2) \]
Kernel Trick

$$\phi(x_1, x_2) \rightarrow (x_1^2, \sqrt{2}x_1x_2, x_2^2)$$

$$\left(\frac{x_1}{a}\right)^2 + \left(\frac{x_2}{b}\right)^2 = 1 \rightarrow \frac{z_1}{a^2} + \frac{z_3}{b^2} = 1$$
Key idea:

- **Mercer’s Theorem**
- A dot product in the new space = function (kernel) in old space
- Means: never have to know what $\phi$ is!!
- Only have to compute distances with the Kernel.
Example:

\[ \phi(x_1, x_2) \rightarrow (x_1^2, \sqrt{2}x_1x_2, x_2^2) \]

dot product: \[ <x, w> = x_1 \ast w_1 + x_2 \ast w_2 \]

\[ K(x, w) = <\phi(x), \phi(w)> \]
\[ = x_1^2 w_1^2 + 2x_1x_2 w_1 w_2 + x_2^2 w_2^2 \]
\[ = (x_1 w_1 + x_2 w_2)^2 \]
\[ = (<x, w>)^2 \]

Circle data is linearly separable if distance (dot product) is computed using \( K(x, w) \)
Support Vector Machines

\[ o : c_i = -1 \]
\[ x : c_i = +1 \]

minimize \( \|w\|^2 \) subject to \( c_i(w \cdot x_i - b) > 1 \)

Quadratic Programming problem

Also: use Kernel trick
Neural Networks

- inspired by biological networks (brain)
- connect up many simple **units**
- simple neuron: threshold and fire
- can help gain understanding of how biological intelligence works
Neural Networks

- can learn the same things that a decision tree can
- imposes different learning bias (way of making new predictions)
- back-propagation learning: errors made are propagated backwards to change the weights
Each node $j$ has a set of weights $w_{j0}, w_{j1}, \ldots, w_{jN}$

Each node $j$ receives inputs $v_0, v_1, \ldots, v_N$

Number of weights = number of parents + 1 ($v_0 = 1$ constant bias term)

Output is the activation function output

$$o_j = f\left(\sum_i w_{ji} v_i\right)$$

Necessary!

A linear function of a linear function is a ... linear function
activation functions:

- step function = integrate-and-fire (biological)
  \[ f(z) = \begin{cases} 
  c & \text{if } z \geq 0 \\
  1 & \text{if } z < 0 
  \end{cases} \]

- sigmoid function \( f(z) = \frac{1}{1 + e^{-z}} \)

- rectified linear (ReLU): \( g(z) = \max\{0, z\} \)

output of entire network is the classification result
Deep Neural Networks

Figure 7.16: A deep neural network
back-propagation implements stochastic gradient descent

Recall:

$$w_i \leftarrow w_i - \eta \frac{\partial \text{Error}(E, \overline{w})}{\partial w_i}$$

$\eta$: learning rate.

Linear unit:

$$\frac{\partial (aw + b)}{\partial w} = a$$

Sigmoid unit (chain rule):

$$\frac{\partial f(g(w))}{\partial w} = f'(g(w)) \frac{\partial g(w)}{\partial w}$$
Learning weights

Using the **chain rule**, this can be extended throughout the network e.g. taking a derivative of the $L^{th}$ layer w.r.t a weight in the $R^{th}$ layer:

$$\frac{\partial \text{output}_L}{\partial w^R} = \frac{\partial f(\text{output}_{L-1})}{\partial w^R}$$

$$= f'(\text{output}_{L-1}) \frac{\partial \sum_i w_{ji}^{L-1} \text{input}_{L-1}}{\partial w^R}$$

$$= f'(\text{output}_{L-1}) \sum_i w_{ji} \frac{\partial f(\text{output}_{L-2})}{\partial w^R}$$

$$= f'(\text{output}_{L-1}) \sum_i w_{ji} f'(\text{output}_{L-2}) \ldots \frac{\partial \sum_k w_{lk}^R \text{input}_R}{\partial w^R}$$

$$= f'(\text{output}_{L-1}) \sum_i w_{ji} f'(\text{output}_{L-2}) \ldots \text{input}_R$$
Backpropagation implements stochastic gradient descent

- each layer $i = 1 \ldots L$ has:
  - $N_i$ input units with $\text{input}[j], j = 1 \ldots N_i$
  - $M_i$ output units with $\text{output}[j], j = 1 \ldots M_i$
- $Y[j]$ is the data output/labels ($\text{output}[L]$)
- $X[i]$ is the data input ($\text{input}[1]$)
- error on output layer unit $j$: $\text{error}[j] = (Y[j] - \text{output}[j])$
- for each other layer:
  1. weight update (linear layer) $w_{ji} \leftarrow w_{ji} + \eta \times \text{input}[i] \times \text{error}[j]$
  2. back-propagated error (linear layer) $\text{input\_error}[i] = \sum_j w_{ji} \text{error}[j]$
  3. back-propagated error (activation layer) $\text{input\_error}[i] = f'(\text{output}[i]) \times \text{error}[i]$
1: repeat
2:   for each example e in $E$ in random order do
3:     for each layer $i = 1 \ldots L$ do (forwards)
4:       $output_i = f(input_i)$
5:     for each layer $j = L \ldots 1$ do (backwards)
6:       compute back-propagated error
7:       update weights
8: until some stopping criteria is reached
Regularization

Regularized Neural nets: prevent overfitting, increased bias for reduced variance

- parameter norm penalties added to objective function
- dataset augmentation
- early stopping
- dropout
- parameter tying
  - Convolutional Neural nets: used for images
  - Recurrent Neural nets: used for sequences
Composite models

- **Random Forests**
  - Each decision tree in the forest is different
  - different features, splitting criteria, training sets
  - average or majority vote determines output

- **Ensemble Learning**: combination of base-level algorithms

- **Boosting**
  - sequence of learners
  - each learner is trained to fit the examples the previous learner did not fit well
  - learners progressively biased towards higher precision
  - early learners: lots of false positives, but reject all the clear negatives
  - later learners: problem is more difficult, but the set of examples is more focused around the challenging boundary
Next:

- Uncertainty (Poole & Mackworth (2nd ed.) Chapter 8)