Lecture 9b - Supervised Machine Learning II

Jesse Hoey
School of Computer Science
University of Waterloo
March 10, 2020

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 \( X_0 \) that has value 1, to make it not a special case.

Finding weights that minimize Error \( E, \mathbf{w} \)

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

- \( \hat{Y} = [Y(e_1), Y(e_2), \ldots, Y(e_M)] \) 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
- \( \mathbf{w} = [w_0, w_1, \ldots, w_n] \) is a vector of the weights

then,

\[
\hat{Y}^T X (X^T X)^{-1} = \mathbf{w}
\]

Finding weights that minimize \( \text{Error}_E(\mathbf{w}) \)

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

Gradient descent:

\[
w_i \leftarrow w_i - \eta \frac{\partial \text{Error}(E, \mathbf{w})}{\partial w_i}
\]

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

If

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

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 \( \eta \to 2\eta \) (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 \( w_0, \ldots, w_n \)
Algorithm on the last slide is **incremental** gradient descent.

If examples are chosen randomly at line 8 then its **stochastic gradient descent**.

**Batched gradient descent**:
- process a batch of size \( n \) before updating the weights
- if \( n \) is all the data, then its **gradient descent**
- if \( n = 1 \), its **incremental gradient descent**

Incremental can be more efficient than batch, but convergence not guaranteed.

---

**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:

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

The partial derivative with respect to weight \( w_i \) is:

\[
\frac{\partial Error(E, \overline{w})}{\partial w_i} = -2 \cdot \delta \cdot f'(\sum_{i} w_i \cdot X_i(e)) \cdot X_i(e)
\]

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

Thus, each example \( e \) updates each weight \( w_i \) by

\[
w_i \leftarrow w_i + \eta \cdot \delta \cdot f'(\sum_{i} w_i \cdot X_i(e)) \cdot X_i(e)
\]

---

**The sigmoid or logistic activation function**

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

\[
f'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = f(x)(1 - f(x))
\]

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:

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

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

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

A simple activation function is the step function:

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

---

**Discussion Board Example**

\[
\hat{Y}(\overline{w})(e) = \text{sigmoid}(\sum_{i} w_i \cdot X_i(e))
\]

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, \( f^w(X_1, \ldots, X_n) = f(w_0 + w_1X_1(e) + \cdots + w_nX_n(e)) = 0.5 \). 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.

---

### Kernel Trick

Some arbitrary data:

![Data](image1.png)

Data is not linearly separable:

![Data](image2.png)

Add another dimension, data is now linearly separable:

![Data](image3.png)

**Kernel Trick: another example**

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

\[
\left( \frac{x_1}{a} \right)^2 + \left( \frac{x_2}{b} \right)^2 = 1 \rightarrow \frac{z_1}{a^2} + \frac{z_3}{b^2} = 1
\]

### Mercer’s Theorem

**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:

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

Dot product: \(<x, w> = x_1 w_1 + x_2 w_2 \]

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

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 Basics**

- 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 ...

---

**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 \( f(z) = 1/(1 + e^{-z}) \)
- Rectified linear (ReLU): \( g(z) = \max\{0, z\} \)

Output of entire network is the classification result
**Deep Neural Networks**

**Backpropagation**

Backpropagation implements stochastic gradient descent. Recall:

\[
\text{w}_i \leftarrow \text{w}_i - \eta \frac{\partial \text{Error}(E, \hat{\text{w}})}{\partial \text{w}_i}
\]

\(\eta\): 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:

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

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

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

\[
= f'(\text{output}_{L-1}) \sum_i \text{w}_{ji} f'(\text{output}_{L-2}) \sum_k \text{w}_{ki} \text{input}_R
\]

\[
= f'(\text{output}_{L-1}) \sum_i \text{w}_{ji} f'(\text{output}_{L-2}) \text{input}_R
\]

**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

---

1: repeat
2: for each example \(e\) in \(E\) in random order do
3: for each layer \(i = 1 \ldots L\) do (forwards)
4: \(\text{output}_i = f(\text{input}_i)\)
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
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:
Unsupervised Learning with Uncertainty (Poole & Mackworth (2nd ed.) chapter 10.2, 10.3, 10.5)