Lecture 9b - Supervised Machine Learning II

Jesse Hoey
School of Computer Science
University of Waterloo

June 27, 2022

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

Linear Regression

Linear regression is a model in which the output is a linear function of the input features.

$$\hat{Y}^{\vec{w}}(e) = w_0 + w_1 X_1(e) + \cdots + w_n X_n(e)$$

$$\hat{Y}^{\vec{w}}(e) = \sum_{i=0}^{n} w_i X_i(e)$$

where $\vec{w} = \langle w_0, w_1, w_2 .... w_n \rangle$. We invent a new feature $X_0 \equiv 1$, to make it not a special case.

Linear Regression

Linear regression is a model in which the output is a linear function of the input features.

$$\hat{Y}^{\vec{w}}(e) = w_0 + w_1 X_1(e) + \cdots + w_n X_n(e)$$

$$\hat{Y}^{\vec{w}}(e) = \sum_{i=0}^{n} w_i X_i(e)$$

where $\vec{w} = \langle w_0, w_1, w_2 .... w_n \rangle$. We invent a new feature $X_0 \equiv 1$, to make it not a special case.
The sum of squares error on examples E for output Y is:

$$Error(E, \vec{w}) = \sum_{e \in E} (Y(e) - \hat{Y}^{\vec{w}}(e))^2$$

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

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

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

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,

$$\vec{y}^T = \vec{w} X$$

$$\vec{y}^T X^T (X X^T)^{-1} = \vec{w}$$

$(XX^T)^{-1}$ is the pseudo-inverse

Finding weights that minimize $Error_E(\vec{w})$

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

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

η is the gradient descent step size, the learning rate.
If

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

then update rule:

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

where we have set $\eta \rightarrow 2\eta$ (arbitrary scale)

Incremental Gradient Descent for Linear Regression

```
1:  procedure LinearLearner(X, Y, E, η)
2:      Inputs  X: set of input features, X = {X_1, ..., X_n}
3:              Y: output feature
4:              E: set of examples from which to learn
5:              η: learning rate
6:      initialize w_0, ..., w_n randomly
7:      repeat
8:          for each example e in E do
9:              δ ← Y(e) − Σ_{i=0}^{n} w_i X_i(e)
10:             for each i ∈ [0, n] do
11:                 w_i ← w_i + η δ X_i(e)
12:     until some stopping criteria is true
13:     return w_0, ..., 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 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

Linear Classifier

- 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:

$$\hat{Y}^{\vec{w}}(e) = f(w_0 + w_1 X_1(e) + \cdots + w_n X_n(e))$$
$$= f(\sum_{i=0}^{n} w_i X_i(e))$$

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

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

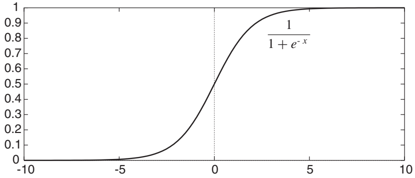
The partial derivative with respect to weight w_i is:

$$\frac{\partial Error(E, \vec{w})}{\partial w_i} = -2 * \delta * f'\left(\sum_i w_i * X_i(e) \right) * X_i(e)$$

where $\delta = (Y(e) - f(\sum_{i=0}^{n} w_i X_i(e)))$.
Thus, each example e updates each weight w_i by

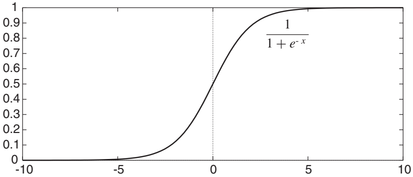
$$w_i \;\leftarrow\; w_i + \eta * \delta * f'\left(\sum_i w_i * X_i(e) \right) * X_i(e)$$

The sigmoid or logistic activation function



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

The sigmoid or logistic activation function

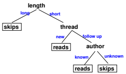


$$f(x) = \frac{1}{1 + e^{-x}}$$

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

so $f'(x)$ can be computed from $f(x)$

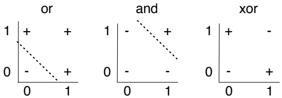
Discussion Board Example



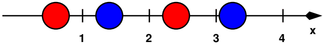
$$\widehat{Reads}(e) = sigmoid(-8 + 7 * Short(e) + 3 * New(e) + 3 * Known(e))$$

Using the 18 training examples from lecture 4, this can be found in about 3000 iterations with a learning rate of $\eta = 0.05$

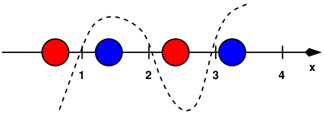
- A dataset is linearly separable if there is a hyperplane where the classification is true on one side of the hyperplane and false on the other side.
- The hyperplane is defined by where the predicted value, $f^{\vec{w}}(X_1, \ldots, X_n) = f(w_0 + w_1 X_1(e) + \cdots + w_n X_n(e))$ is 0.5. For the sigmoid function, the hyperplane is defined by $w_0 + w_1 X_1(e) + \cdots + w_n X_n(e) = 0$.
- Some data are not linearly separable



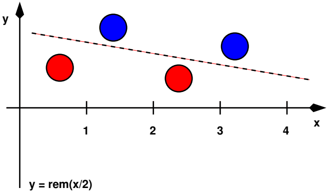
Some arbitrary data:



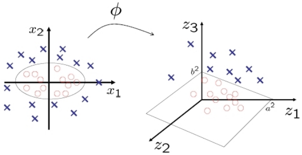
Data is not linearly separable:



Add another dimension, data is now linearly separable:



Kernel Trick: another example



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

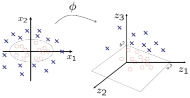
$$\left(\frac{x_1}{a}\right)^2 + \left(\frac{x_2}{b}\right)^2 = 1 \rightarrow \frac{z_1}{a^2} + \frac{z_3}{b^2} = 1$$

Mercer's Theorem

Key idea:

- Mercer's Theorem
- A dot product in the new "lifted" space = function (kernel) in old space
- Means: never have to know what ϕ is!!
- Only have to compute distances with the kernel.

Example



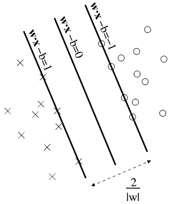
$\phi(x_1, x_2) \rightarrow (x_1^2, \sqrt{2}x_1x_2, x_2^2)$

dot product in old space: $< x, w >= x_1 * w_1 + x_2 * w_2$
dot product in new space:
kernel $K(x, w)$

$$\begin{aligned} K(x, w) &= < \phi(x), \phi(w) > \\ &= x_1^2 w_1^2 + 2x_1x_2w_1w_2 + x_2^2 w_2^2 \\ &= (x_1w_1 + x_2w_2)^2 \\ &= (< x, w >)^2 \end{aligned}$$

Circle data is linearly separable if distance (dot product) is computed using $K(x, w)$

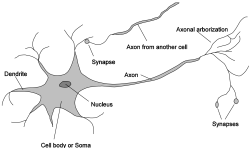
Support Vector Machines



find the classification boundary with the widest margin
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

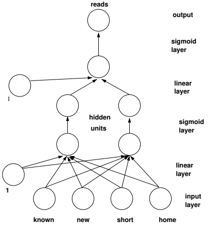
Neural Networks

- 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
- often the linear and sigmoid layers are treated as a single layer



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

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

necessarily non-linear because
A linear function of a linear function is a ...

linear function

Neural Networks Basics

- activation functions:
  - step function = integrate-and-fire (biological)
    $$f(z) = \begin{cases} c & \text{if} \quad z \geq 0 \\ 1 & \text{if} \quad z < 0 \end{cases}$$
  - 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

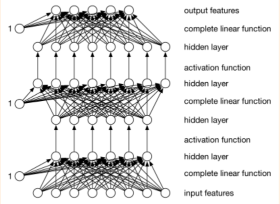


Figure 7.16: A deep neural network

Learning weights

back-propagation implements stochastic gradient descent
Recall:

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

η: 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:

$$\begin{aligned}
\frac{\partial output_L}{\partial w^R} &= \frac{\partial f(output_{L-1})}{\partial w^R} \\
&= f'(output_{L-1}) \frac{\partial \sum_i w_{ji}^{L-1} input_{L-1}}{\partial w^R} \\
&= f'(output_{L-1}) \sum_i w_{ji} \frac{\partial f(output_{L-2})}{\partial w^R} \\
&= f'(output_{L-1}) \sum_i w_{ji} f'(output_{L-2}) \ldots \frac{\partial \sum_k w_{lk}^R input_R}{\partial w^R} \\
&= f'(output_{L-1}) \sum_i w_{ji} f'(output_{L-2}) \ldots input_R
\end{aligned}$$

Backpropagation

back-propagation implements stochastic gradient descent

- each layer $i = 1 \ldots L$ has:
  - N_i input units with $input[j], j = 1 \ldots N_i$
  - M_i output units with $output[j], j = 1 \ldots M_i$
- $Y[j]$ is the data output/labels ($output[L]$)
- $X[i]$ is the data input ($input[1]$)
- error on output layer unit j: $error[j] = (Y[j] - output[j])$
- for each other layer:
  1. weight update (linear layer) $w_{ji} \leftarrow w_{ji} + \eta * input[i] * error[j]$
  2. back-propagated error (linear layer)
     $input\_error[i] = \sum_j w_{ji} error[j]$
  3. back-propagated error (activation layer)
     $input\_error[i] = f'(output[i]) * error[i]$

Backpropagation

repeat
  for each example e in E in random order **do**
    for each layer $i = 1 \ldots L$ **do** (forwards)
      $output_i = f(input_i)$
    for each layer $j = L \ldots 1$ **do** (backwards)
      compute back-propagated error
      update weights
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 focussed around the challenging boundary

Next:

- Unsupervised Learning with Uncertainty (Poole & Mackworth (2nd ed.)chapter 10.2,10.3,10.5)