Lecture 7
Classification: logistic regression

Intro to NLP, CS585, Fall 2014
http://people.cs.umass.edu/~brenocon/inlp2014/
Brendan O’Connor (http://brenocon.com)
Today on classification

• Where do features come from?
• Where do weights come from?
• Regularization
• NEXT TIME (Exercise 4 due tomorrow night, class exercise on Thursday)
  • Multiclass outputs
  • Training, testing, evaluation
  • Where do labels come from? (Humans??!)
Linear models for classification

Train on \((x,y)\) pairs. Predict on new \(x\)'s.

Recap: binary case \((y=1 \text{ or } 0)\)

Feature vector \(x = (1, \text{count "happy"}, \text{count "hello"}, \ldots)\)

Weights/parameters \(\beta = (-1.1, 0.8, -0.1, \ldots)\)
Linear models for classification

Train on \((x,y)\) pairs. Predict on new \(x\)'s.

Recap: binary case \((y=1 \text{ or } 0)\)

Feature vector \(x = (1, \text{ count } \text{“happy”}, \text{ count } \text{“hello”}, ...)\)

Weights/parameters \(\beta = (-1.1, 0.8, -0.1, ... )\)

Dot product a.k.a. inner product \(\beta^T x = \sum_j \beta_j x_j = -1.1 + 0.8 (\text{#happy}) - 0.1 (\text{#hello}) + ...\) [it’s high when high \(\beta_j\)'s coincide with high \(x_j\)'s]

[Foundation of supervised machine learning!]
Linear models for classification

Train on \((x,y)\) pairs. Predict on new \(x\)'s.

Recap: binary case \((y=1\ \text{or}\ 0)\)

- Feature vector: \(x = (1, \text{count "happy"}, \text{count "hello"}, ...)\)
- Weights/parameters: \(\beta = (-1.1, \ 0.8, \ -0.1, ...)\)
- Dot product (a.k.a. inner product)
  \[\beta^T x = \sum_j \beta_j x_j = -1.1 + 0.8 (\text{#happy}) - 0.1 (\text{#hello}) + ...\]
  
  [it's high when high \(\beta_j\)'s coincide with high \(x_j\)'s]
  [this is why it's "linear"]

- Hard prediction ("linear classifier")
- Soft prediction ("linear logistic regression")
Linear models for classification

Train on $(x, y)$ pairs. Predict on new $x$’s.

Recap: binary case $(y = 1 \text{ or } 0)$

Feature vector $x = (1, \text{ count } \text{“happy”}, \text{ count } \text{“hello”}, \ldots)$

Weights/parameters $\beta = (-1.1, \ 0.8, \ -0.1, \ldots)$

Dot product a.k.a. inner product $\beta^T x = \sum_j \beta_j x_j = -1.1 + 0.8 \ (#\text{happy}) - 0.1 \ (#\text{hello}) + \ldots$

[It’s high when high beta_j’s coincide with high $x_j$’s]

Hard prediction (“linear classifier”) $\hat{y} = \begin{cases} 1 & \text{if } \beta^T x > 0 \\ 0 & \text{otherwise} \end{cases}$

Soft prediction (“linear logistic regression”) $\hat{y}$

[Foundation of supervised machine learning!]

Sunday, September 28, 14
Linear models for classification

Train on \((x, y)\) pairs. Predict on new \(x\)'s.

Recap: binary case \((y = 1 \text{ or } 0)\)

Feature vector
\[
x = (1, \text{ count "happy"}, \text{ count "hello"}, \ldots)
\]

Weights/parameters
\[
\beta = (-1.1, 0.8, -0.1, \ldots)
\]

Dot product
a.k.a. inner product
\[
\beta^T x = \sum_j \beta_j x_j = -1.1 + 0.8 (\text{#happy}) - 0.1 (\text{#hello}) + ... \\
\text{[this is why it's "linear"]}
\]

Hard prediction
("linear classifier")
\[
\hat{y} = \begin{cases} 
1 & \text{if } \beta^T x > 0 \\
0 & \text{otherwise}
\end{cases}
\]

Soft prediction
("linear logistic regression")
\[
p(y = 1|x, \beta) = g(\beta^T x) \\
g(z) = e^z /[1 + e^z] \\
\text{["logistic sigmoid function"]}
\]

[Foundation of supervised machine learning!]
Visualizing a classifier in feature space

Feature vector \( x = (1, \text{count "happy"}, \text{count "hello"}, \ldots) \)

Weights/parameters \( \beta = \)

50% prob where
\( \beta^T x = 0 \)
Predict \( y=1 \) when
\( \beta^T x > 0 \)
Predict \( y=0 \) when
\( \beta^T x \leq 0 \)
Visualizing a classifier in feature space

Feature vector: \( x = (1, \text{count "happy"}, \text{count "hello"}, \ldots) \)

Weights/parameters: \( \beta = (-1.0, 0.8, -0.1, \ldots) \)

50% prob where
\[ \beta^T x = 0 \]
Predict \( y = 1 \) when
\[ \beta^T x > 0 \]
Predict \( y = 0 \) when
\[ \beta^T x \leq 0 \]
Visualizing a classifier in feature space

Feature vector \( x = (1, \text{count "happy"}, \text{count "hello"}, ...) \)

Weights/parameters \( \beta = (-1.0, 0.8, -0.1, ...) \)

50% prob where \( \beta^T x = 0 \)

Predict \( y=1 \) when \( \beta^T x > 0 \)

Predict \( y=0 \) when \( \beta^T x \leq 0 \)
We have a model for probabilistic classification. Now what?

- Where do features come from?
- Where do weights come from?
- Regularization
- NEXT TIME:
  - Multiclass outputs
  - Training, testing, evaluation
  - Where do labels come from? (Humans??!)
• Input document $d$ (a string...)
• Engineer a feature function, $f(d)$, to generate feature vector $x$

$$f(d) = \begin{pmatrix}
\text{Count of “happy”}, \\
\text{(Count of “happy”) / (Length of doc)}, \\
\log(1 + \text{count of “happy”}), \\
\text{Count of “not happy”}, \\
\text{Count of words in my pre-specified word list, “positive words according to my favorite psychological theory”}, \\
\text{Count of “of the”}, \\
\text{Length of document}, \\
\text{...}
\end{pmatrix}$$

Typically these use feature templates:
Generate many features at once

- for each word $w$:
  - ${w}\_count$
  - ${w}\_log\_1\_plus\_count$
  - ${w}\_with\_NOT\_before\_it\_count$
  - ...

• Not just word counts. Anything that might be useful!
• Feature engineering: when you spend a lot of trying and testing new features. Very important for effective classifiers!! This is a place to put linguistics in.
Where do weights come from?

- Choose by hand
- Learn from labeled data
  - Analytic solution (Naive Bayes)
  - Gradient-based learning
Learning the weights

Maximize the training set's (log-)likelihood?

$$\beta^{\text{MLE}} = \arg\max_\beta \log p(y_1..y_n|x_1..x_n, \beta)$$

$$\log p(y_1..y_n|x_1..x_n, \beta) = \sum_i \log p(y_i|x_i, \beta) = \sum_i \log \begin{cases} p_i & \text{if } y_i = 1 \\ 1 - p_i & \text{if } y_i = 0 \end{cases}$$

where $$p_i \equiv p(y_i = 1|x, \beta)$$

- No analytic form, unlike our counting-based multinomials in NB, n-gram LM's, or Model 1.
- Use gradient ascent: iteratively climb the log-likelihood surface, through the derivatives for each weight.
- Luckily, the derivatives turn out to look nice...
Gradient ascent

Loop while not converged (or as long as you can):
For all features $j$, compute and add derivatives:

$$\beta_j^{(new)} = \beta_j^{(old)} + \eta \frac{\partial}{\partial \beta_j} \ell(\beta^{(old)})$$

- $\ell$: Training set log-likelihood
- $\eta$: Step size (a.k.a. learning rate)
- $\left( \frac{\partial \ell}{\partial \beta_1}, \ldots, \frac{\partial \ell}{\partial \beta_J} \right)$: Gradient vector (vector of per-element derivatives)

This is a generic optimization technique. Not specific to logistic regression! Finds the maximizer of any function where you can compute the gradient.
Gradient ascent in practice

Loop while not converged (or as long as you can):
For all features \( j \), compute and add derivatives:

\[
\beta_j^{(new)} = \beta_j^{(old)} + \eta \frac{\partial}{\partial \beta_j} \ell(\beta^{(old)})
\]

Better gradient methods dynamically choose good step sizes ("quasi-Newton methods")

The most commonly used is **L-BFGS**.

Use a library (exists for all programming languages, e.g. scipy).
Typically, the library function takes two callback functions as input:
- `objective(beta)`: evaluate the log-likelihood for beta
- `grad(beta)`: return a gradient vector at beta

Then it runs many iterations and stops once done.
Gradient of logistic regression

\[ \ell(\beta) = \log p(y_1 \ldots y_n|x_1 \ldots x_n, \beta) = \sum_i \log p(y_i|x_i, \beta) = \sum_i \ell_i(\beta) \]

where

\[ \ell_i(\beta) = \log \begin{cases} p(y_i = 1|x, \beta) & \text{if } y_i = 1 \\ p(y_i = 0|x, \beta) & \text{if } y_i = 0 \end{cases} \]
Gradient of logistic regression

\[ \ell(\beta) = \log p(y_1 \ldots y_n | x_1 \ldots x_n, \beta) = \sum_i \log p(y_i | x_i, \beta) = \sum_i \ell_i(\beta) \]

where \( \ell_i(\beta) = \log \begin{cases} p(y_i = 1 | x, \beta) & \text{if } y_i = 1 \\ p(y_i = 0 | x, \beta) & \text{if } y_i = 0 \end{cases} \)

\[ \frac{\partial}{\partial \beta_j} \ell(\beta) = \sum_i \frac{\partial}{\partial \beta_j} \ell_i(\beta) \]

\[ \frac{\partial}{\partial \beta_j} \ell_i(\beta) = \]
Gradient of logistic regression

\[ \ell(\beta) = \log p(y_1 \ldots y_n | x_1 \ldots x_n, \beta) = \sum_i \log p(y_i | x_i, \beta) = \sum_i \ell_i(\beta) \]

where

\[ \ell_i(\beta) = \log \begin{cases} p(y_i = 1 | x, \beta) & \text{if } y_i = 1 \\ p(y_i = 0 | x, \beta) & \text{if } y_i = 0 \end{cases} \]

\[ \frac{\partial}{\partial \beta_j} \ell(\beta) = \sum_i \frac{\partial}{\partial \beta_j} \ell_i(\beta) \]

\[ \frac{\partial}{\partial \beta_j} \ell_i(\beta) = [y_i - p(y_i | x, \beta)] x_j \]

Proabilistic error (zero if 100% confident in correct outcome)
Feature value (e.g. word count)
Gradient of logistic regression

\[ \ell(\beta) = \log p(y_1 \ldots y_n | x_1 \ldots x_n, \beta) = \sum_i \log p(y_i | x_i, \beta) = \sum_i \ell_i(\beta) \]

where \( \ell_i(\beta) = \log \left\{ \begin{array}{ll} p(y_i = 1 | x, \beta) & \text{if } y_i = 1 \\ p(y_i = 0 | x, \beta) & \text{if } y_i = 0 \end{array} \right\} \)

\[ \frac{\partial}{\partial \beta_j} \ell(\beta) = \sum_i \frac{\partial}{\partial \beta_j} \ell_i(\beta) \]

\[ \frac{\partial}{\partial \beta_j} \ell_i(\beta) = \left[ y_i - p(y_i | x, \beta) \right] x_j \]

E.g. \( y=1 \) (positive sentiment), and count("happy") is high, but you only predicted 10% chance of positive label: want to increase \( \beta_j \)!
Regularization

• Just like in language models, there’s a danger of overfitting the training data. (For LM’s, how did we combat this?)

• One method is count thresholding: throw out features that occur in < L documents (e.g. L=5). This is OK, and makes training faster, but not as good as....

• Regularized logistic regression: add a new term to penalize solutions with large weights. Controls the bias/variance tradeoff.

\[
\beta^{\text{MLE}} = \arg \max_{\beta} \left[ \log p(y_1 .. y_n | x_1 .. x_n, \beta) \right]
\]

\[
\beta^{\text{Regul}} = \arg \max_{\beta} \left[ \log p(y_1 .. y_n | x_1 .. x_n, \beta) - \lambda \sum_j (\beta_j)^2 \right]
\]

“Regularizer constant”: Strength of penalty

“Quadratic penalty” or “L2 regularizer”: Squared distance from origin
How to set the regularizer?

• Quadratic penalty in logistic regression ... Pseudocounts for count-based models ...
• Ideally: split data into
  • Training data
  • Development (“tuning”) data
  • Test data (don’t peek!)
• (Or cross-validation)
• Try different lambdas. For each train the model and predict on devset. Choose lambda that does best on dev set: e.g. maximizes accuracy or likelihood.
• Often we use a grid search like (2^-2, 2^-1 ... 2^4, 2^5) or (10^-1, 10^0 .. 10^3). Sometimes you only need to be within an order of magnitude to get reasonable performance.

Dev. set accuracy

Use this one

Lambda

Hopefully looks like this