Logistic Regression

Razvan C. Bunescu

Department of Computer Science @ CCI

rbunescu@uncc.edu
Supervised Learning

Training Examples: \((x_k, t_k)\)

Learning Algorithm

Model \(h\)

Testing Examples: \((x, t)\)

Model \(h\)

Generalization Performance
Supervised Learning

- **Task** = learn an (unknown) function $t : X \rightarrow T$ that maps input instances $x \in X$ to output targets $t(x) \in T$:
  - **Classification**:
    - The output $t(x) \in T$ is one of a finite set of discrete categories.
  - **Regression**:
    - The output $t(x) \in T$ is continuous, or has a continuous component.

- Target function $t(x)$ is known (only) through (noisy) set of training examples:
  $\{(x_1, t_1), (x_2, t_2), \ldots, (x_n, t_n)\}$
Parametric Approaches to Supervised Learning

• **Task** = build a function $h(x)$ such that:
  – $h$ matches $t$ well on the training data:
    => $h$ is able to fit data that it has seen.
  – $h$ also matches $t$ well on test data:
    => $h$ is able to generalize to unseen data.

• **Task** = choose $h$ from a “nice” class of functions that depend on a vector of parameters $w$:
  – $h(x) ≡ h_w(x) ≡ h(w,x)$
  – what classes of functions are “nice”?
Three Parametric Approaches to Classification

1) **Discriminant Functions**: construct $f : X \rightarrow T$ that directly assigns a vector $x$ to a specific class $C_k$.
   - Inference and decision combined into a single learning problem.
   - *Linear Discriminant*: the decision surface is a hyperplane in $X$:
     - Perceptron
     - Support Vector Machines
     - Fisher ‘s Linear Discriminant
Three Parametric Approaches to Classification

2) **Probabilistic Discriminative Models**: directly model the posterior class probabilities $p(C_k \mid x)$.
   - Inference and decision are separate.
   - Less data needed to estimate $p(C_k \mid x)$ than $p(x \mid C_k)$.
   - Can accommodate many overlapping features.
     - Logistic Regression
     - Conditional Random Fields
Three Parametric Approaches to Classification

3) **Probabilistic Generative Models:**
   - Model class-conditional \( p(\mathbf{x} | C_k) \) as well as the priors \( p(C_k) \), then use Bayes’s theorem to find \( p(C_k | \mathbf{x}) \).
     - or model \( p(\mathbf{x}, C_k) \) directly, then marginalize to obtain the posterior probabilities \( p(C_k | \mathbf{x}) \).
   - Inference and decision are separate.
   - Can use \( p(\mathbf{x}) \) for *outlier* or *novelty detection*.
   - Need to model dependencies between features.
     - Naïve Bayes.
     - Hidden Markov Models.
Generative and Discriminative Classifiers

Suppose we're distinguishing cat from dog images

ImageNet

ImageNet
Generative Classifier:

- Build a model of what's in a cat image
  - Knows about whiskers, ears, eyes
  - Assigns a probability to any image:
    - how cat-y is this image?

Also build a model for dog images

Given a new image:

Run both models and see which one fits better.
Discriminative Classifier

Just try to distinguish dogs from cats

Oh look, dogs have collars!

*Let's ignore everything else.*
Finding the correct class \( c \) from a document \( d \) in

Generative vs Discriminative Classifiers

- **Naive Bayes**

\[
\hat{c} = \arg\max_{c \in C} \left( P(d|c) \cdot P(c) \right)
\]

- **Logistic Regression**

\[
\hat{c} = \arg\max_{c \in C} P(c|d)
\]
Neurons

**Soma** is the central part of the neuron:
- *where the input signals are combined.*

**Dendrites** are cellular extensions:
- *where majority of the input occurs.*

**Axon** is a fine, long projection:
- *carries nerve signals to other neurons.*

**Synapses** are molecular structures between axon terminals and other neurons:
- *where the communication takes place.*
McCulloch-Pitts Neuron Function

- Algebraic interpretation:
  - The output of the neuron is a **linear combination** of inputs from other neurons, **rescaled by** the synaptic **weights**.
    - weights $w_i$ correspond to the synaptic weights (activating or inhibiting).
    - summation corresponds to combination of signals in the soma.
  - It is often transformed through an **activation / output function**.
### Activation Functions

**unit step** \( f(z) = \begin{cases} 
0 & \text{if } z < 0 \\
1 & \text{if } z \geq 0 
\end{cases} \)

**Perceptron**

**logistic** \( f(z) = \frac{1}{1 + e^{-z}} \)

**Logistic Regression**

**identity** \( f(z) = z \)

**Linear Regression**

[Graph showing the activation functions: unit step, logistic, identity, and linear regression.]
Linear Regression

- Polynomial curve fitting is Linear Regression:
  \[ x = \varphi(x) = [1, x, x^2, ..., x^M]^T \]
  \[ h(x) = w^T x \]
• Assume classes $T = \{c_1, c_2\} = \{1, -1\}$.
• Training set is $(x_1, t_1), (x_2, t_2), \ldots, (x_n, t_n)$.

$x = [1, x_1, x_2, \ldots, x_k]^T$

$h(x) = sgn(w^T x) = sgn(w_0 + w_1 x_1 + \ldots + w_k x_k)$

$a linear discriminant function$
Linear Discriminant Functions

- Use a linear function of the input vector:
  \[ h(x) = w^T \varphi(x) + w_0 \]

- Decision:
  \[ x \in C_1 \text{ if } h(x) \geq 0, \text{ otherwise } x \in C_2. \]
  \[ \Rightarrow \text{decision boundary is hyperplane } h(x) = 0. \]

- Properties:
  - \( w \) is orthogonal to vectors lying within the decision surface.
  - \( w_0 \) controls the location of the decision hyperplane.
Geometric Interpretation
Logistic Regression

- Training set is \((x_1, t_1), (x_2, t_2), \ldots, (x_n, t_n)\).
  
  \[ x = [1, x_1, x_2, \ldots, x_k]^T \]
  
  \[ h(x) = \sigma(w^T x) \]

- Can be used for both classification and regression:
  - **Classification**: \(T = \{C_1, C_2\} = \{1, 0\}\).
  - **Regression**: \(T = [0, 1]\) (i.e. output needs to be normalized).
Logistic Regression for Binary Classification

• Model output can be interpreted as **posterior class probabilities**:

\[ p(C_1 \mid x) = \sigma(w^T x) = \frac{1}{1 + \exp(-w^T x)} \]

\[ p(C_2 \mid x) = 1 - \sigma(w^T x) = \frac{\exp(-w^T x)}{1 + \exp(-w^T x)} \]

• How do we train a logistic regression model?
  – What **error/cost function** to minimize?
Example: LR for Sentiment Classification
Logistic Regression Learning

- **Learning** = finding the “right” parameters $\mathbf{w}^T = [w_0, w_1, \ldots, w_k]$
  - Find $\mathbf{w}$ that minimizes an error function $E(\mathbf{w})$ which measures the misfit between $h(x_n, \mathbf{w})$ and $t_n$.
  - Expect that $h(\mathbf{x}, \mathbf{w})$ performing well on training examples $x_n \Rightarrow h(\mathbf{x}, \mathbf{w})$ will perform well on arbitrary test examples $\mathbf{x} \in X$.

- **Least Squares** error function?

\[
E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} \left( h(\mathbf{x}_n, \mathbf{w}) - t_n \right)^2
\]

  - Differentiable $\Rightarrow$ can use gradient descent $\checkmark$
  - Non-convex $\Rightarrow$ not guaranteed to find the global optimum $\times$
Maximum Likelihood

Training set is \( D = \{\langle x_n, t_n \rangle | t_n \in \{0,1\}, n \in 1 \ldots N\} \)

Let \( h_n = p(C_1 | x_n) \Leftrightarrow h_n = p(t_n = 1 | x_n) = \sigma(w^T x_n) \)

**Maximum Likelihood (ML) principle**: find parameters that maximize the likelihood of the labels.

- The **likelihood function** is
  \[
  p(t | w) = \prod_{n=1}^{N} h_n^{t_n} (1 - h_n)^{\left(1 - t_n\right)}
  \]

- The negative log-likelihood (cross entropy) **error function**:
  \[
  E(w) = -\ln p(t | x) = -\sum_{n=1}^{N} \left\{ t_n \ln h_n + (1 - t_n) \ln(1 - h_n) \right\} \times \frac{1}{N}
  \]
  
  *we also average*
Maximum Likelihood Learning for Logistic Regression

- The **ML** solution is:

\[ w_{ML} = \arg \max_w p(t \mid w) = \arg \min_w E(w) \]

- **ML** solution is given by \( \nabla E(w) = 0 \).
  - Cannot solve analytically => solve numerically with gradient based methods: (stochastic) gradient descent, conjugate gradient, L-BFGS, etc.
  - Gradient is (prove it):

\[
\nabla E(w) = \sum_{n=1}^{N} (h_n - t_n)x_n^T \times \frac{1}{N}
\]
Regularized Logistic Regression

- Use a Gaussian prior over the parameters:
  \[ \mathbf{w} = [w_0, w_1, \ldots, w_M]^T \]

  \[ p(\mathbf{w}) = N(\mathbf{0}, \alpha^{-1}\mathbf{I}) = \left( \frac{\alpha}{2\pi} \right)^{(M+1)/2} \exp\left\{ -\frac{\alpha}{2} \mathbf{w}^T \mathbf{w} \right\} \]

- Bayes’ Theorem:

  \[ p(\mathbf{w} | \mathbf{t}) = \frac{p(\mathbf{t} | \mathbf{w})p(\mathbf{w})}{p(\mathbf{t})} \propto p(\mathbf{t} | \mathbf{w})p(\mathbf{w}) \]

- MAP solution:

  \[ \mathbf{w}_{MAP} = \arg\max_{\mathbf{w}} p(\mathbf{w} | \mathbf{t}) \]
Regularized Logistic Regression

- **MAP solution:**

\[
\mathbf{w}_{MAP} = \arg \max_{\mathbf{w}} p(\mathbf{w} | \mathbf{t}) = \arg \max_{\mathbf{w}} p(\mathbf{t} | \mathbf{w}) p(\mathbf{w})
\]

\[
= \arg \min_{\mathbf{w}} - \ln p(\mathbf{t} | \mathbf{w}) p(\mathbf{w})
\]

\[
= \arg \min_{\mathbf{w}} \ln p(\mathbf{t} | \mathbf{w}) - \ln p(\mathbf{w})
\]

\[
= \arg \min_{\mathbf{w}} E_D(\mathbf{w}) - \ln p(\mathbf{w})
\]

\[
= \arg \min_{\mathbf{w}} E_D(\mathbf{w}) + \frac{\alpha}{2} \mathbf{w}^T \mathbf{w} = \arg \min_{\mathbf{w}} E_D(\mathbf{w}) + E_w(\mathbf{w})
\]

- **Data term**

\[
E_D(\mathbf{w}) = -\sum_{n=1}^{N} \left\{ t_n \ln y_n + (1 - t_n) \ln(1 - y_n) \right\} \times \frac{1}{N}
\]

- **Regularization term**

\[
E_w(\mathbf{w}) = \frac{\alpha}{2} \mathbf{w}^T \mathbf{w}
\]
Regularized Logistic Regression

- **MAP** solution:
  \[
  w_{MAP} = \arg \min_w E_D(w) + E_w(w)
  \]

- **ML** solution is given by \( \nabla E(w) = 0 \).

  \[
  \nabla E(w) = \nabla E_D(w) + \nabla E_w(w) = \frac{1}{N} \sum_{n=1}^{N} (h_n - t_n) x_n^T + \alpha w^T
  \]

  where \( h_n = \sigma(w^T x_n) \)

- Cannot solve analytically \( \Rightarrow \) solve numerically:
  - (stochastic) gradient descent [PRML 3.1.3], Newton Raphson iterative optimization [PRML 4.3.3], conjugate gradient, LBFGS.
Implementation: Vectorization of LR

- **Version 1:** Compute gradient component-wise.

\[
\nabla E(w) = \sum_{n=1}^{N} (h_n - t_n)x_n^T \times \frac{1}{N}
\]

- Assume example \( x_n \) is stored in column \( X[:,n] \) in data matrix \( X \).

```python
grad = np.zeros(K)
for n in range(N):
    h = sigmoid(w.dot(X[:,n]))
    temp = h - t[n]
    for k in range(K):
        grad[k] = grad[k] + temp * X[k,n] / N

def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```
Implementation: Vectorization of LR

- **Version 2**: Compute gradient, partially vectorized.

\[
\nabla E(w) = \sum_{n=1}^{N} (h_n - t_n) x_n^T \times \frac{1}{N}
\]

\[
\text{grad} = \text{np.zeros}(K)
\text{for n in range(N):
    grad} = \text{grad} + (\text{sigmoid}(w.T \cdot X[:,n])) - t[n]) \times X[:,n] / N
\]

```python
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```
Implementation: Vectorization of LR

- **Version 3**: Compute gradient, vectorized.

\[
\nabla E(w) = \sum_{n=1}^{N} (h_n - t_n) x_n^T \times \frac{1}{N}
\]

\[
\text{grad} = X.\text{dot}(\text{sigmoid}(w.\text{dot}(X)) - t) / N
\]

```python
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```
Vectorization of LR with Separate Bias

- Separate the bias $b$ from the weight vector $\mathbf{w}$.
- Compute gradient separately with respect to $\mathbf{w}$ and $b$:
  - Gradient with respect to $\mathbf{w}$ is:
    \[
    \nabla E(\mathbf{w}) = \sum_{n=1}^{N} (h_n - t_n) \mathbf{x}_n^T \times \frac{1}{N}
    \]
    \[
    h_n = \sigma(\mathbf{w}^T \mathbf{x}_n + b)
    \]
    \[
    \text{grad} = \mathbf{X} \cdot \text{sigmoid}(\mathbf{w} \cdot \mathbf{X} + b) - \mathbf{t} ) / N
    \]
  - Gradient with respect to bias $b$ is:
    \[
    \Delta b = -\frac{1}{N} \sum_{n=1}^{N} (h_n - t_n)
    \]

\[
\text{def sigmoid(x):}
    \text{return 1 / (1 + np.exp(-x))}
\]
Vectorization of LR with Regularization

• Only the gradient with respect to $w$ changes:
  – never use L2 regularization on bias.

\[
\nabla E(w) = \sum_{n=1}^{N} (h_n - t_n)x_n^T \times \frac{1}{N} + \alpha w
\]

\[
grad = X.\text{dot}(\text{sigmoid}(w.\text{dot}(X) + b) - t) / N + \alpha w
\]
Softmax Regression = Logistic Regression for Multiclass Classification

- Multiclass classification:
  \[ T = \{C_1, C_2, ..., C_K\} = \{1, 2, ..., K\}. \]

- Training set is \((x_1, t_1), (x_2, t_2), \ldots (x_n, t_n)\).
  \[ x = [1, x_1, x_2, ..., x_M] \]
  \[ t_1, t_2, \ldots t_n \in \{1, 2, ..., K\} \]

- One weight vector per class [PRML 4.3.4]:
  \[
p(C_k | x) = \frac{\exp(w_k^T x)}{\sum_j \exp(w_j^T x)}
  \]
  bias parameter inside each \(w_j\)

  \[
p(C_k | x_n) = \frac{\exp(w_k^T x_n + b_k)}{\sum_{j=1..K} \exp(w_j^T x_n + b_j)}
  \]
  separate bias parameter \(b_j\)
Softmax Regression (K ≥ 2)

**Inference:**

\[
C_\star = \arg \max_{C_k} p(C_k \mid x)
\]

\[
= \arg \max_{C_k} \exp(w_k^T x) \sum_j \exp(w_j^T x)
\]

\[
= \arg \max_{C_k} \exp(w_k^T x)
\]

\[
= \arg \max_{C_k} w_k^T x
\]

**Training using:**

- Maximum Likelihood (ML)
- Maximum A Posteriori (MAP) with a Gaussian prior on \(w\).
Softmax Regression

• The **negative log-likelihood** error function is:

\[
E_D(w) = -\frac{1}{N} \ln \prod_{n=1}^{N} p(t_n | x_n) = -\frac{1}{N} \sum_{n=1}^{N} \ln \frac{\exp(w^T t_n x_n)}{Z(x_n)}
\]

• The **Maximum Likelihood** solution is:

\[
w_{ML} = \arg \min_w E_D(w)
\]

• The **gradient** is (prove it):

\[
\nabla_{w_k} E_D(w) = -\frac{1}{N} \sum_{n=1}^{N} (\delta_k(t_n) - p(C_k | x_n)) x_n
\]

where \( \delta_i(x) = \begin{cases} 1 & x = t \\ 0 & x \neq t \end{cases} \) is the Kronecker delta function.
Regularized Softmax Regression

- The new **cost** function is:

\[
E(w) = E_D(w) + E_w(w)
\]

\[
= - \frac{1}{N} \sum_{n=1}^{N} \ln \frac{\exp(w^T_t x_n)}{Z(x_n)} + \frac{\alpha}{2} \|w\|^2
\]

- The new **gradient** is (prove it):

\[
\text{grad}_k = \nabla_w E(w) = -\frac{1}{N} \sum_{n=1}^{N} (\delta_k(t_n) - p(C_k | x_n)) x_n^T + \alpha w_k^T
\]
Softmax Regression

• ML solution is given by $\nabla E_D(w) = 0$.
  – Cannot solve analytically.
  – Solve numerically, by plugging $[\text{cost, gradient}] = [E(w), \nabla E(w)]$ values into general convex solvers:
    • L-BFGS
    • Newton methods
    • conjugate gradient
    • (stochastic / minibatch) gradient-based methods.
      – gradient descent (with / without momentum).
      – AdaGrad, AdaDelta
      – RMSProp
      – ADAM, …
Implementation

- Need to compute \([\text{cost, grad}]\):

\[
\text{cost} = -\frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} \delta_k(t_n) \ln p(C_k | x_n) + \frac{\alpha}{2} \sum_{k=1}^{K} w_k^T w_k
\]

\[
\text{grad}_k = -\frac{1}{N} \sum_{n=1}^{N} \left( \delta_k(t_n) - p(C_k | x_n) \right) x_n^T + \alpha w_k^T
\]

=> need to compute, for \(k = 1, \ldots, K\):

- **output** \(p(C_k | x_n) = \frac{\exp(w_k^T x_n)}{\sum_j \exp(w_j^T x_n)}\)

Overflow when \(w_k^T x_n\) are too large.
Implementation: Preventing Overflows

- Subtract from each product $\mathbf{w}_k^T \mathbf{x}_n$ the maximum product:

  $$c_n = \max_{1 \leq k \leq K} \mathbf{w}_k^T \mathbf{x}_n$$

  $$p(C_k | \mathbf{x}_n) = \frac{\exp(\mathbf{w}_k^T \mathbf{x}_n - c_n)}{\sum_j \exp(\mathbf{w}_j^T \mathbf{x}_n - c_n)}$$

- When using separate bias $b_k$, replace $\mathbf{w}_k^T \mathbf{x}_n$ everywhere with $\mathbf{w}_k^T \mathbf{x}_n + b_k$. 
Vectorization of Softmax with Separate Bias

- Separate the bias $b_k$ from the weight vector $w_k$.
- Compute gradient separately with respect to $w_k$ and $b_k$:
  
  - Gradient with respect to $w_k$ is:
    \[
    \text{grad}_k = -\frac{1}{N} \sum_{n=1}^{N} (\delta_k(t_n) - p(C_k | x_n)) x_n^T + \alpha w_k^T
    \]
    Gradient matrix is $[\text{grad}_1 | \text{grad}_2 | \ldots | \text{grad}_K]$.
  
  - Gradient with respect to $b_k$ is:
    \[
    \Delta b_k = -\frac{1}{N} \sum_{n=1}^{N} (\delta_k(t_n) - p(C_k | x_n))
    \]
    Gradient vector is $\Delta b = [\Delta b_1 | \Delta b_2 | \ldots | \Delta b_K]$

\[
p(C_k | x_n) = \frac{\exp(w_k^T x_n + b_k)}{\sum_{j=1..K} \exp(w_j^T x_n + b_j)}
\]

\[
\delta_k(t_n) = \begin{cases} 
1, & \text{if } t_n = k \\
0, & \text{if } t_n \neq k 
\end{cases}
\]
Vectorization of Softmax

• Need to compute \([cost, \, grad, \, \Delta b]\):

\[
\text{cost} = -\frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} \delta_k(t_n) \ln p(C_k | x_n) + \frac{\alpha}{2} \sum_{k=1}^{K} w_k^T w_k
\]

\[
\text{grad}_k = -\frac{1}{N} \sum_{n=1}^{N} (\delta_k(t_n) - p(C_k | x_n)) x_n^T + \alpha w_k^T
\]

\[p(C_k | x_n) = \frac{\exp(w_k^T x_n + b_k)}{\sum_{j=1..K} \exp(w_j^T x_n + b_j)}\]

=> compute ground truth matrix \(G\) such that \(G[k,n] = \delta_k(t_n)\)

\[
\delta_k(t_n) = \begin{cases} 
1, & \text{if } t_n = k \\
0, & \text{if } t_n \neq k 
\end{cases}
\]

```python
from scipy.sparse import coo_matrix

groundTruth = coo_matrix((np.ones(N, dtype = np.uint8),
                          (labels, np.arange(N)))).toarray()
```
Vectorization of Softmax

- Compute \( \text{cost} = -\frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} \delta_k(t_n) \ln p(C_k | x_n) + \frac{\alpha}{2} \sum_{k=1}^{K} w_k^T w_k \)
  - Compute matrix of \( w_k^T x_n + b_k \).
  - Compute matrix of \( w_k^T x_n + b_k - c_n \).
  - Compute matrix of \( \exp(w_k^T x_n + b_k - c_n) \).
  - Compute matrix of \( \ln p(C_k | x_n) \).
  - Compute log-likelihood cost using all the above.

\[
p(C_k | x_n) = \frac{\exp(w_k^T x_n + b_k)}{\sum_{j=1..K} \exp(w_j^T x_n + b_j)}
\]

\[
\delta_k(t_n) = \begin{cases} 
1, & \text{if } t_n = k \\
0, & \text{if } t_n \neq k
\end{cases}
\]

\[
c_n = \max_{1 \leq k \leq K} w_k^T x_n + b_k
\]
Vectorization of Softmax

- Compute \( \text{grad}_k = -\frac{1}{N} \sum_{n=1}^{N} (\delta_k(t_n) - p(C_k | x_n)) x_n^T + \alpha w_k^T \)

  - Gradient matrix = \([\text{grad}_1 | \text{grad}_2 | \ldots | \text{grad}_K]\)
    - Compute matrix of \( p(C_k | x_n) \).
    - Compute matrix of gradient of data term.
    - Compute matrix of gradient of regularization term.
Vectorization of Softmax

- Useful Numpy functions:
  - np.dot()
  - np.amax()
  - np.argmax()
  - np.exp()
  - np.sum()
  - np.log()
  - np.mean()
Implementation: Gradient Checking

- Want to minimize $J(\theta)$, where $\theta$ is a scalar.

- Mathematical definition of derivative:

$$\frac{d}{d\theta} J(\theta) = \lim_{\varepsilon \to 0} \frac{J(\theta + \varepsilon) - J(\theta - \varepsilon)}{2\varepsilon}$$

- Numerical approximation of derivative:

$$\frac{d}{d\theta} J(\theta) \approx \frac{J(\theta + \varepsilon) - J(\theta - \varepsilon)}{2\varepsilon} \quad \text{where } \varepsilon = 0.0001$$
Implementation: Gradient Checking

- If $\theta$ is a vector of parameters $\theta_i$,
  - Compute numerical derivative with respect to each $\theta_i$.
    - Create a vector $v$ that is $\varepsilon$ in position $i$ and 0 everywhere else:
      - *How do you do this without a for loop in NumPy?*
    - Compute $G_{num}(\theta_i) = (J(\theta + v) - J(\theta - v)) / 2\varepsilon$
    - Aggregate all derivatives into numerical gradient $G_{num}(\theta)$.

- Compare numerical gradient $G_{num}(\theta)$ with implementation of gradient $G_{imp}(\theta)$:
  \[
  \frac{\|G_{num}(\theta) - G_{imp}(\theta)\|}{\|G_{num}(\theta) + G_{imp}(\theta)\|} \leq 10^{-6}
  \]