Logistic Regression

Razvan C. Bunescu
School of Electrical Engineering and Computer Science
bunescu@ohio.edu
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:
  $$(x_1, t_1), (x_2, t_2), \ldots (x_n, t_n)$$
Supervised Learning

Training

Training Examples $\{(x_k, t_k)\}$ → Learning Algorithm → Model $h$

Testing

Test Examples $\{(x, t)\}$ → Model $h$ → Generalization Performance
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 $w$:
  – $h(x) \equiv h_w(x) \equiv h(w,x)$
  – what classes of functions are “nice”?
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.*
## Neuron Models


<table>
<thead>
<tr>
<th>Year</th>
<th>Model Name</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1907</td>
<td>Integrate and fire</td>
<td>[13]</td>
</tr>
<tr>
<td>1943</td>
<td>McCulloch and Pitts</td>
<td>[11]</td>
</tr>
<tr>
<td>1952</td>
<td>Hodgkin-Huxley</td>
<td>[12]</td>
</tr>
<tr>
<td>1958</td>
<td>Perceptron</td>
<td>[14]</td>
</tr>
<tr>
<td>1961</td>
<td>Fitzhugh-Nagumo</td>
<td>[15]</td>
</tr>
<tr>
<td>1965</td>
<td>Leaky integrate-and-fire</td>
<td>[16]</td>
</tr>
<tr>
<td>1981</td>
<td>Morris-Lecar</td>
<td>[17]</td>
</tr>
<tr>
<td>1986</td>
<td>Quadratic integrate-and-fire</td>
<td>[18]</td>
</tr>
<tr>
<td>1989</td>
<td>Hindmarsh-Rose</td>
<td>[19]</td>
</tr>
<tr>
<td>1999</td>
<td>Wilson Polynomial</td>
<td>[21]</td>
</tr>
<tr>
<td>2000</td>
<td>Integrate-and-fire or burst</td>
<td>[22]</td>
</tr>
<tr>
<td>2001</td>
<td>Resonate-and-fire</td>
<td>[23]</td>
</tr>
<tr>
<td>2003</td>
<td>Izhikevich</td>
<td>[24]</td>
</tr>
<tr>
<td>2003</td>
<td>Exponential integrate-and-fire</td>
<td>[25]</td>
</tr>
<tr>
<td>2004</td>
<td>Generalized integrate-and-fire</td>
<td>[26]</td>
</tr>
<tr>
<td>2005</td>
<td>Adaptive exponential integrate-and-fire</td>
<td>[27]</td>
</tr>
<tr>
<td>2009</td>
<td>Mihalas-Neibur</td>
<td>[28]</td>
</tr>
</tbody>
</table>
Fig. 2. (a) Illustration and (b) functional description of a leaky integrate-and-fire neuron. Weighted and delayed input signals are summed into the input current $I_{app}(t)$, which travel to the soma and perturb the internal state variable, the voltage $V$. Since $V$ is hysteric, the soma performs integration and then applies a threshold to make a spike or no-spike decision. After a spike is released, the voltage $V$ is reset to a value $V_{reset}$. The resulting spike is sent to other neurons in the network.
# Neuron Models


<table>
<thead>
<tr>
<th>Year</th>
<th>Model Name</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1907</td>
<td>Integrate and fire</td>
<td>[13]</td>
</tr>
<tr>
<td>1943</td>
<td>McCulloch and Pitts</td>
<td>[11]</td>
</tr>
<tr>
<td>1952</td>
<td>Hodgkin-Huxley</td>
<td>[12]</td>
</tr>
<tr>
<td>1958</td>
<td>Perceptron</td>
<td>[14]</td>
</tr>
<tr>
<td>1961</td>
<td>Fitzhugh-Nagumo</td>
<td>[15]</td>
</tr>
<tr>
<td>1965</td>
<td>Leaky integrate-and-fire</td>
<td>[16]</td>
</tr>
<tr>
<td>1981</td>
<td>Morris-Lecar</td>
<td>[17]</td>
</tr>
<tr>
<td>1986</td>
<td>Quadratic integrate-and-fire</td>
<td>[18]</td>
</tr>
<tr>
<td>1989</td>
<td>Hindmarsh-Rose</td>
<td>[19]</td>
</tr>
<tr>
<td>1999</td>
<td>Wilson Polynomial</td>
<td>[21]</td>
</tr>
<tr>
<td>2000</td>
<td>Integrate-and-fire or burst</td>
<td>[22]</td>
</tr>
<tr>
<td>2001</td>
<td>Resonate-and-fire</td>
<td>[23]</td>
</tr>
<tr>
<td>2003</td>
<td>Izhikevich</td>
<td>[24]</td>
</tr>
<tr>
<td>2003</td>
<td>Exponential integrate-and-fire</td>
<td>[25]</td>
</tr>
<tr>
<td>2004</td>
<td>Generalized integrate-and-fire</td>
<td>[26]</td>
</tr>
<tr>
<td>2005</td>
<td>Adaptive exponential integrate-and-fire</td>
<td>[27]</td>
</tr>
<tr>
<td>2009</td>
<td>Mihalas-Neibur</td>
<td>[28]</td>
</tr>
</tbody>
</table>
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

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

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

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

**Perceptron**

**Logistic Regression**

**Linear Regression**
Linear Regression

- Polynomial curve fitting is Linear Regression:
  \[ x = \phi(x) = [1, x, x^2, \ldots, x^M]^T \]
  \[ h(x) = w^T x \]
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 a monotonic activation / output function.
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**:

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

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

• How do we train a logistic regression model?
  – What **error/cost function** to minimize?
Logistic Regression Learning

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

• Least Squares error function?

$$E(w) = \frac{1}{2} \sum_{n=1}^{N} \left( h(x_n, w) - t_n \right)^2$$

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

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

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

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

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

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

• The **ML** solution is:

\[ \mathbf{w}_{ML} = \arg \max_{\mathbf{w}} p(\mathbf{t} \mid \mathbf{w}) = \arg \min_{\mathbf{w}} E(\mathbf{w}) \]

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

\[
\nabla E(\mathbf{w}) = \sum_{n=1}^{N} \left( h_n - t_n \right) \mathbf{x}_n^T
\]
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:**

\[
\mathbf{w}_{MAP} = \arg \max_{\mathbf{w}} p(\mathbf{w} \mid \mathbf{t}) = \arg \max_{\mathbf{w}} p(\mathbf{t} \mid \mathbf{w}) p(\mathbf{w}) \\
= \arg \min_{\mathbf{w}} -\ln p(\mathbf{t} \mid \mathbf{w}) p(\mathbf{w}) \\
= \arg \min_{\mathbf{w}} \ln p(\mathbf{t}) - \ln p(\mathbf{w}) \\
= \arg \min_{\mathbf{w}} E_D(\mathbf{w}) - \ln p(\mathbf{w}) \\
= \arg \min_{\mathbf{w}} E_D(\mathbf{w}) + \frac{\alpha}{2} \mathbf{w}^T \mathbf{w}
\]

\[
E_D(\mathbf{w}) = -\sum_{n=1}^{N} \{t_n \ln y_n + (1-t_n) \ln(1-y_n)\} \quad \text{data term}
\]

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

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

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

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

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

- Cannot solve analytically => solve numerically:
  - (stochastic) gradient descent [PRML 3.1.3], Newton Raphson iterative optimization [PRML 4.3.3], conjugate gradient, LBFGS.
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]:
  \[
  p(C_k \mid x) = \frac{\exp(w_k^T x)}{\sum_j \exp(w_j^T x)}
  \]
Softmax Regression ($K \geq 2$)

**Inference:**

$$C_* = \arg\max_{C_k} p(C_k \mid x)$$

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

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

$$= \arg\max_{C_k} w_k^T x$$

**Training using:**

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

$Z(x)$ a normalization constant
Softmax Regression

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

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

• The **Maximum Likelihood** solution is:

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

• The **gradient** is (**prove it**):

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

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:

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

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

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

\[
\nabla_{w_k} E(w) = - \frac{1}{N} \sum_{n=1}^{N} (\delta_k (t_n) - p(C_k | x_n)) x_n^T + \alpha w_k^T
\]
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 \([\text{cost}, \text{gradient}]:\)

\begin{align*}
\text{cost} & = -\frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} \delta_k(t_n) \ln p(C_k | \mathbf{x}_n) + \frac{\alpha}{2} \sum_{k=1}^{K} \mathbf{w}_k^T \mathbf{w}_k \\
\text{gradient}_k & = -\frac{1}{N} \sum_{n=1}^{N} (\delta_k(t_n) - p(C_k | \mathbf{x}_n)) \mathbf{x}_n^T + \alpha \mathbf{w}_k^T
\end{align*}

\[\Rightarrow\] need to compute, for \(k = 1, \ldots, K:\)

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

Overflow when \(\mathbf{w}_k^T \mathbf{x}_n\) are too large.
Implementation: Preventing Overflows

• Subtract from each product $w_k^T x_n$ the maximum product:

$$c_n = \max_{1 \leq k \leq K} w_k^T x_n$$

$$p(C_k \mid x_n) = \frac{\exp(w_k^T x_n - c_n)}{\sum_j \exp(w_j^T x_n - c_n)}$$
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 $G_{\text{num}}(\theta_i) = (J(\theta + v) - J(\theta - v)) / 2\varepsilon$
      – Aggregate all derivatives into numerical gradient $G_{\text{num}}(\theta)$.

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

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

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

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

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

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]
```
Implementation: Vectorization of LR

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

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

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

---

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

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

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

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

---

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

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

  \[
  \begin{align*}
  \text{cost} &= -\frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} \delta_k(t_n) \ln p(C_k \mid x_n) + \frac{\alpha}{2} \sum_{k=1}^{K} w_k^T w_k \\
  \text{gradient}_k &= -\frac{1}{N} \sum_{n=1}^{N} (\delta_k(t_n) - p(C_k \mid x_n)) x_n^T + \alpha w_k^T
  \end{align*}
  \]

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

from scipy.sparse import coo_matrix
groundTruth = coo_matrix((np.ones(N, dtype = np.uint8),
                        (labels, np.arange(N)))).toarray()
Vectorization of Softmax

- Compute \( cost = -\frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} \delta_k(t_n) \ln p(C_k | x_n) + \frac{\alpha}{2} \sum_{k=1}^{K} w_k^T w_k \)
  
  - Compute matrix of \( w_k^T x_n \).
  
  - Compute matrix of \( w_k^T x_n - c_n \).
  
  - Compute matrix of \( \exp(w_k^T x_n - c_n) \).
  
  - Compute matrix of \( \ln p(C_k | x_n) \).
  
  - Compute log-likelihood.
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 = $[\text{grad}_1 | \text{grad}_2 | ... | \text{grad}_K]$

- 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()
import scipy

- scipy.sparse.coo_matrix()
  
  groundTruth = coo_matrix((np.ones(numCases, dtype = np.uint8),
                           (labels, np.arange(numCases)))).toarray()

- scipy.optimize:
  
  - scipy.optimize.fmin_l_bfgs_b()
    
    theta, _, _ = fmin_l_bfgs_b(softmaxCost, theta,
                                args = (numClasses, inputSize, decay, images, labels),
                                maxiter = 100, disp = 1)
  
  - scipy.optimize.fmin_cg()
  
  - scipy.minimize

https://docs.scipy.org/doc/scipy-0.10.1/reference/tutorial/optimize.html
Multiclass Logistic Regression (K ≥ 2)

1) Train one weight vector per class [PRML Chapter 4.3.4]:

\[ p(C_k \mid x) = \frac{\exp(w_k^T \varphi(x))}{\sum_j \exp(w_j^T \varphi(x))} \]

2) More general approach:

\[ p(C_k \mid x) = \frac{\exp(w_k^T \varphi(x, C_k))}{\sum_j \exp(w_j^T \varphi(x, C_j))} \]

- Inference:

\[ C_* = \arg \max_{C_k} p(C_k \mid x) \]
Logistic Regression (K ≥ 2)

2) **Inference** in more general approach:

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

\[
= \arg \max_{C_k} \frac{\exp(w^T \varphi(x, C_k))}{\sum_j \exp(w^T \varphi(x, C_j))}
\]

\[
= \arg \max_{C_k} \exp(w^T \varphi(x, C_k))
\]

\[
= \arg \max_{C_k} w^T \varphi(x, C_k)
\]

- **Training** using:
  - Maximum Likelihood (ML)
  - Maximum A Posteriori (MAP) with a Gaussian prior on \( w \).
Logistic Regression (K ≥ 2) with ML

- The negative log-likelihood error function is:

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

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

- The gradient is (prove it):

\[
\nabla E_D(w) = \left[ \frac{\partial E_D(w)}{\partial w_0}, \frac{\partial E_D(w)}{\partial w_1}, \ldots, \frac{\partial E_D(w)}{\partial w_M} \right]
\]

\[
\frac{\partial E_D(w)}{\partial w_i} = -\sum_{n=1}^{N} \phi_i(x_n, t_n) + \sum_{n=1}^{N} \sum_{k=1}^{K} p(C_k | x_n) \phi_i(x_n, C_k)
\]

*convex in \(w\)*
Logistic Regression ($K \geq 2$) with ML

- Set $\nabla E_D(w) = 0 \implies$ **ML solution** satisfies:

$$\sum_{n=1}^{N} \varphi_i(x_n, t_n) = \sum_{n=1}^{N} \sum_{k=1}^{K} p(C_k | x_n) \varphi_i(x_n, C_k)$$

implies for every feature $\varphi_i$, the **observed value** on $D$ should be the same as the **expected value** on $D$!

- **Solve numerically:**
  - Stochastic gradient descent [chapter 3.1.3].
  - Newton Raphson iterative optimization (large Hessian!).
  - Limited memory Newton methods (e.g. L-BFGS).
The Maximum Entropy Principle

- **Principle of Insufficient Reason**
- **Principle of Indifference**
  - can be traced back to Pierre Laplace and Jacob Bernoulli.

  - “model all that is known and assume nothing about that which is unknown”.
  - “given a collection of facts, choose a model consistent with all the facts, but otherwise as uniform as possible”. 
Maximum Likelihood ⇔ Maximum Entropy

1) Maximize conditional likelihood:

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

\[ p(t \mid w) = \prod_{n=1}^{N} p_w(t_n \mid x_n) = \prod_{n=1}^{N} \frac{\exp(w^T \varphi(x_n, t_n))}{Z(x_n)} \]

2) Maximize conditional entropy:

\[ p_{ME} = \arg \max_p \sum_{n=1}^{N} \sum_{k=1}^{K} - p(C_k \mid x_n) \log p(C_k \mid x_n) \]

subject to:

\[ \sum_{n=1}^{N} \varphi(x_n, t_n) = \sum_{n=1}^{N} \sum_{k=1}^{K} p(C_k \mid x_n) \varphi(x_n, C_k) \]

\[ p_{ME}(t_n \mid x_n) = p_{w_{ML}}(t_n \mid x_n) = \frac{\exp(w_{ML}^T \varphi(x_n, t_n))}{Z(x_n)} \]