GRADIENT DESCENT

Logistic Regression

For Binary Classification: \( \mathcal{X} \rightarrow [0, 1] \)
\[
f(x; w) = \sigma(w^T \tilde{x}) = \frac{\exp(w^T \tilde{x})}{1 + \exp(w^T \tilde{x})}
\]

- Output is interpreted as probability \( Pr(y = 1) \)
- \( w^T \tilde{x} \) are the log-odds.

\[
f(x; \theta) = \frac{1}{1 + \exp(-w^T \tilde{x})} \quad 1 - f(x; \theta) = \frac{1}{1 + \exp(w^T \tilde{x})}
\]
\[
w^T \tilde{x} = \log \frac{f(x; \theta)}{1 - f(x; \theta)} = \log \frac{Pr(y = 1)}{1 - Pr(y = 1)} = \log \frac{Pr(y = 1)}{Pr(y = 0)}
\]

To classify, \( y = 1 \) if \( f(x; w) > 0.5 \) and 0 otherwise.
- \( y = 1 \) if \( w^T \tilde{x} > 0 \), and 0 if \( w^T \tilde{x} \leq 0 \).
- \( \tilde{x} \) is some augmented variable of \( x \).
  - "Linear Classifier" if \( \tilde{x} = [x^T; 1]^T \)
  - Could be polynomial \( \tilde{x} = [1, x, x^2, x^3] \)
  - Or other arbitrary non-linear functions of \( x \)
- \( \tilde{x} \) often called the "feature" vector.
- Some encoding of the input.
  - \( x \) could even be non-numeric, and you could define some numeric features.

Note: Classifier is linear in chosen encoding.
- \( w^T \tilde{x} \ll 0 \) defines a "separating hyperplane" between positive and negative part of the space of \( \tilde{x} \).
Logistic Regression

\[ f(x; w) = \sigma(w^T \tilde{x}) = \frac{\exp(w^T \tilde{x})}{1 + \exp(w^T \tilde{x})} \]

- Cross-entropy / Negative Likelihood Loss

\[ L(y, f(x; w)) = -y \log f(x; w) - (1 - y) \log(1 - f(x; w)) \]

\[ f(x; \theta) = \frac{1}{1 + \exp(-w^T \tilde{x})}, \quad 1 - f(x; \theta) = \frac{1}{1 + \exp(w^T \tilde{x})} \]

Putting it all together, given a training set of \( \{(x_t, y_t)\} \):

\[ w = \arg \min_w \frac{1}{T} \sum_{t=1}^{T} y_t \log [1 + \exp(-w^T \tilde{x}_t)] + (1 - y_t) \log [1 + \exp(w^T \tilde{x}_t)] \]
Logistic Regression

\[ w = \arg \min_w \frac{1}{T} \sum_{t=1}^T y_t \log [1 + \exp(-w^T x_t)] + (1 - y_t) \log [1 + \exp(w^T x_t)] \]

- You can show that this loss is a convex function of \( w \)
  (compute the Hessian matrix and show that it’s eigenvalues are non-negative)
- So it has a single global minimum.

But how do we find it?

**Gradient Descent**

Minimize a cost that is a function of parameters, and a sum of cost functions, each coming from a different training sample.

**Gradient Descent**

Begin with initial guess \( w_0 \)

At each iteration \( i \):
- \( w_{i+1} \leftarrow w_i - \gamma \nabla_w C(w_i) \)

\( \gamma \) is the step-size. When running optimization for training, often called the "learning rate".

More General Form

\[ w = \arg \min_w C(w) \quad C(w) = \frac{1}{T} \sum_t C_t(w) \]
If you select optimal step size by doing a "line search" for \( \gamma \), can prove that gradient-descent will converge.
If function is convex, converge to unique global minimum.
Second order variants that consider the Hessian matrix: Newton & Quasi-Newton Methods
  - Gauss-Newton, Levenberg-Marquardt, ...

But simple gradient descent suffices / our only choice when:
  - Function isn't convex.
  - Can't afford to do line search.
  - So many parameters that can't compute Hessian.

Also, no theoretical guarantees.
Theory still catching up. Meanwhile, we'll try to understand the "behavior" of the gradients.

---

Logistic Regression

What is \( \nabla_w C(w) \), the gradient of the loss from a single training example?

\[
\nabla_w C(w) = \begin{bmatrix}
\frac{\partial}{\partial w_1} C(w) \\
\frac{\partial}{\partial w_2} C(w) \\
\vdots
\end{bmatrix}
\]

If \( C(w) = \frac{1}{T} \sum_t C_t(w) \), then \( \nabla_w C(w) = \frac{1}{T} \nabla_w C(w) \)

Logistic Regression

\[
C_t(w) = y_t \log \left[ 1 + \exp(-w^T \hat{x}_t) \right] + (1 - y_t) \log \left[ 1 + \exp(w^T \hat{x}_t) \right]
\]

What is \( \nabla_w C_t(w) \), the gradient of the loss from a single training example?

---

Ok, what is the derivative of

\[
C_t(p) = y_t \log \left[ 1 + \exp(p) \right] + (1 - y_t) \log \left[ 1 + \exp(-p) \right]
\]

with respect to \( p \) (where \( p \) is a scalar).

\[
\frac{\partial}{\partial p} C_t(p) = y_t \frac{\exp(p)}{1 + \exp(p)} - y_t \frac{-\exp(-p)}{1 + \exp(-p)} + (1 - y_t) \frac{\exp(p)}{1 + \exp(p)}
\]

\[
= y_t \frac{\exp(p)}{1 + \exp(p)} - y_t \left[ \frac{1}{1 + \exp(-p)} + \frac{\exp(p)}{1 + \exp(p)} \right]
\]

\[
= y_t \frac{\exp(p)}{1 + \exp(p)} - y_t \left[ \frac{1}{1 + \exp(p)} + \frac{\exp(p)}{1 + \exp(p)} \right]
\]
GRADIENT DESCENT

\[ C_i(w) = y_i \log \left[ 1 + \exp(-w^T \hat{x}_i) \right] + (1 - y_i) \log \left[ 1 + \exp(w^T \hat{x}_i) \right] \]

\[ C_i(p) = y_i \log \left[ 1 + \exp(-p) \right] + (1 - y_i) \log \left[ 1 + \exp(p) \right] \]

\[ \frac{\partial}{\partial p} C_i(p) = \frac{\exp(p)}{1 + \exp(p)} - y_i \]

Observations

- \( \frac{\exp(p)}{1 + \exp(p)} \) is basically the output \( f(x_i; w) \), predicted probability that \( y_i = 1 \).
- Remember: this is the expression for gradient of \( p \), i.e. logit / log-odds.
- Gradient 0 if \( y_i = 0 \) and probability 0, \( y_i = 1 \) and probability 1.
  - Do nothing if predicting right answer with perfect confidence.
- If we say probability > 0, and \( y_i = 0 \). Gradient is positive.
- If we say probability < 1, and \( y_i = 1 \). Gradient is negative.

Remember we move in the opposite direction of gradient.

Putting it together:

\[ w = \arg \min_w \frac{1}{T} \sum_{i=1}^{T} \left[ y_i \log \left[ 1 + \exp(-w^T \hat{x}_i) \right] + (1 - y_i) \log \left[ 1 + \exp(w^T \hat{x}_i) \right] \right] \]

At each iteration \( i \),

- Based on current \( w \), compute \( f(x_i, w) = \hat{y}_i \)
- Compute derivative of the "output" as \( \hat{y}_i - y_i \)
- Multiply by \( x_i \) to get \( \nabla_w \)
- Change \( w \) by subtracting some \( \gamma \) times this gradient.
GRADIENT DESCENT

\[ w = \arg \min_w \frac{1}{T} \sum_{t=1}^{T} y_t \log [1 + \exp(-w^T \tilde{x}_t)] + (1 - y_t) \log [1 + \exp(w^T \tilde{x}_t)] \]

Putting it together:

- At each iteration \( i \),
  - Based on current \( w \), compute \( f(x_t, w) = \tilde{y}_t \) for every training sample
  - Compute derivative of the "output" as \( \nabla w_t \) for every training sample
  - Multiply by \( \gamma \) and average across all training samples to get \( \nabla_w \)
  - Change \( w \) by subtracting some \( \gamma \) times this gradient.

Expensive when we have a LOT of training data.

STOCHASTIC GRADIENT DESCENT

\[ w = \arg \min_w \frac{1}{T} \sum_{t} C(x_t, y_t; w) \]

\[ \nabla_w = \frac{1}{T} \sum_{t} \nabla_w C(x_t, y_t; w) \]

Remember, summation over training samples meant to approximate an expectation over \( P_{XY}(x, y) \).

\[ \frac{1}{T} \sum_{t} C(x_t, y_t; w) \rightarrow E_{P_{XY}(x,y)} C(x,y; w) \]

\[ \frac{1}{T} \sum_{t} \nabla_w C(x_t, y_t; w) \rightarrow E_{P_{XY}(x,y)} \nabla_w C(x,y; w) \]

In other words, we are approximating the "true" gradient with gradients over samples.

What if we used a smaller number of samples in each iteration, but different samples in different iterations?

STOCHASTIC GRADIENT DESCENT

- Single sample
  \[ w_{i+1} \leftarrow w_i - \gamma \nabla_w C(x_t, y_t; w_i) \]
  At each iteration, choose a random \( t \in \{1, 2, \ldots, T\} \).

- "Mini"-batched SGD (sometimes GD is called Batched GD)
  \[ w_{i+1} \leftarrow w_i - \gamma \nabla_w \frac{1}{B} \sum_{t \in \mathcal{B}} C(x_t, y_t; w_i) \]
  At each iteration, choose a random smaller batch \( \mathcal{B} \) of size \( B << T \).

With replacement? Without replacement?

STOCHASTIC GRADIENT DESCENT

- Shufflorder of training examples
- Choose a batch size
- Take consecutive groups of \( B \) samples as you loop through iterations
  - \( [1,B] \) in iteration 1
  - \( [B+1,2B] \) in iteration 2
  - \( \ldots \)
  - Once you reach the end of the training set (called one "epoch"),
    shuffle the order again.
STOCHASTIC GRADIENT DESCENT

**General Notes**

- The gradient over a mini-batch is an "approximation", or a "noisy" version of the gradient over the true training set.

\[
\frac{1}{B} \sum_{i \in B} \nabla_w C_i(x_t, y_t; w_i) = \nabla_w \bar{C}(x, y; w) + \epsilon
\]

- Typically, if you decrease the batch-size, you will want to decrease your step size (because you are "less sure" about the gradient).

STOCHASTIC GRADIENT DESCENT

**General Notes**

- Say your cost function is convex, and you care only about decreasing this cost (not worried about overfitting)

- Larger batch size will always give you "better" gradients.

- But diminishing returns after a batch size.

- Computational cost is number of examples per iteration \(\times\) number of iterations for convergence
  - Higher batch means more computation per iteration, but may mean fewer iterations required to converge.

- Best combination of step size and batch size is an empirical question.

- Another factor: parallelism.
  - Note that you can compute the gradient of all samples of your batch in parallel.
  - Ideally, you want to at least "saturate" all available parallel threads.

STOCHASTIC GRADIENT DESCENT

**Momentum**

Standard SGD:

\[
\begin{align*}
g_{i+1} &= \frac{1}{B} \sum_{i \in B} \nabla_w C_i(x_t, y_t; w_i) \\
{w}_{i+1} &\leftarrow {w}_i - \gamma g_{i+1}
\end{align*}
\]

With Momentum:

For \(\beta < 1\):

\[
\begin{align*}
g_{i+1} &= \frac{1}{B} \sum_{i \in B} \nabla_w C_i(x_t, y_t; w_i) + \beta g_i \\
{w}_{i+1} &\leftarrow {w}_i - \gamma g_{i+1}
\end{align*}
\]

- Keep adding the gradient from a previous batch, again and again across iterations, with decaying weight.

- Remember: \(g_i\) was computed with respect to a different position in \(w\) space.

- People often use \(\beta\) as high as 0.9 or 0.99.

- Will need to revisit "best" value of \(\gamma\) when you change \(\beta\).
LOGISTIC REGRESSION

\[ w = \arg \min_w \frac{1}{T} \sum_t C_t(w) \]

\[ C_t(w) = y_t \log[1 + \exp(-w^T \tilde{x}_t)] + (1 - y_t) \log[1 + \exp(w^T \tilde{x}_t)] \]

- Defined linear classifier on augmented vector \( \tilde{x} \)
- Used gradient descent to learn \( w \).
  - Looked at behavior of gradients.
  - Simplified computation with stochasticity.
- At test time, sign of \( w^T \tilde{x} \) gives us our label.

The problem is:

- The definition of augmented vector \( \tilde{x} \) is hand-crafted
- We have manually engineered our features.
- The only thing we're learning is a linear classifier on top.

Want to learn the features themselves!

Given that SGD works, what's stopping us from learning a \( g(x) = \tilde{x} \)?