Weighted Aggregate Logistic Regression (WALR)

An introduction

Problem Statement

Find a function `f(X)` that does a decent job predicting the likelihood of a conversion given an impression.

- `X` is a vector of “features”
- We are assuming that `X` is known to the entity showing the ads

Let’s start simple: Logistic Regression

$$\mathbf{f(X) = \sigma(\theta^T \bullet X)}$$

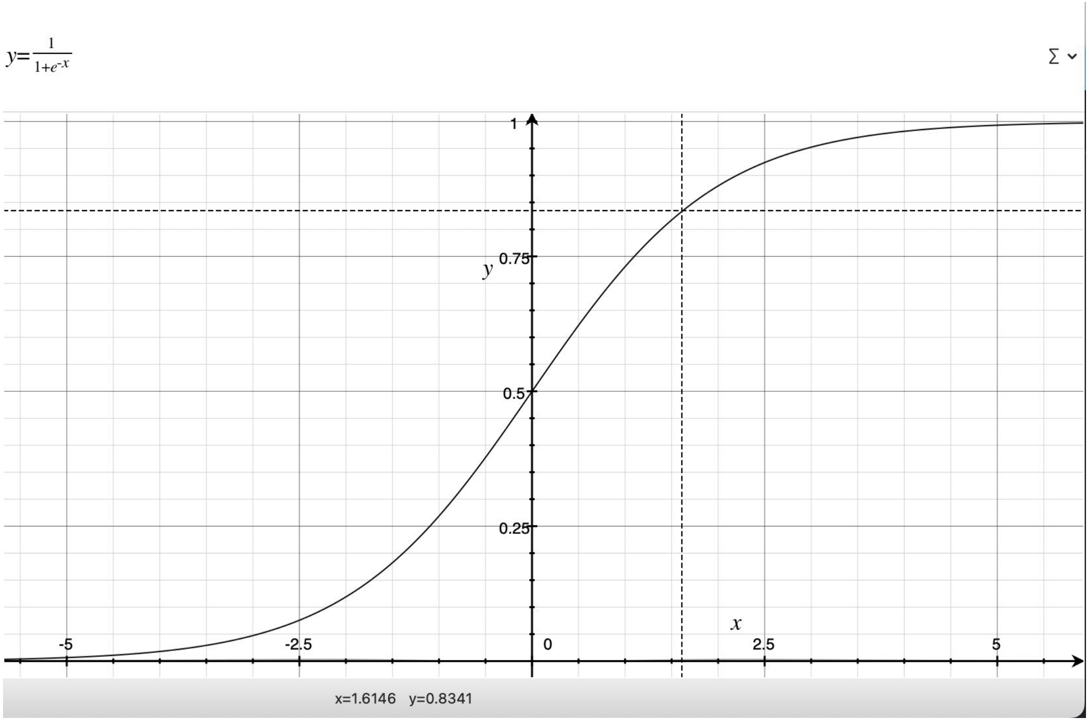
$$X = [x_1,\ x_2,\ \ldots,\ x_k]$$

$$\theta = [c_1,\ c_2,\ \ldots,\ c_k]$$

$$\theta^T \bullet X = c_1x_1 + c_2x_2 + \ldots + c_kx_k$$

$$\sigma(x) = 1/(1 + e^{-x})$$

Why the sigma function?



Converts a “score” into a probability in the range of (0, 1).

Large positive “score” =>
Probability close to 1.0

Large negative “score” =>
Probability close to 0.0

Simplifying assumption: features are binary

Let’s constrain all of the “features” to be binary inputs; either 1 or 0.

Example:

```
X = [0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0]
```

This will make it easier to analyze and interpret, and cheaper to perform in MPC.

How to interpret “Logistic Regression”

Once we have “trained” this simple model, we have a list of coefficients:

$$\theta = [c_1, c_2, \ldots, c_M]$$

Each coefficient tells us about the correlation of a “feature” with the likelihood of a conversion:

- **Large positive coefficients:** Positively correlated
- **Large negative coefficients:** Negatively correlated
- **Coefficients close to zero:** Not strongly correlated

Derivation of WALR

How do you *normally* train a Logistic Regression model?

The normal, non-private approach is "*gradient descent*" as follows:

1. Define a "Loss function"
2. Initialize θ to random values
3. Compute the "gradient" of the "Loss function"
4. Update θ by taking a small step in the direction of the gradient
5. As long as θ hasn't converged, go back to step 3.

Loss Function

Average loss across all examples:

$$L(\theta, \{X^{(i)}\}, \{y^{(i)}\}) = \frac{1}{N} \cdot \sum_{i=1}^{n} l_i(\theta, X^{(i)}, y^{(i)})$$

Let's use the "cross entropy loss" function:

$$l_i(\theta, X^{(i)}, y^{(i)}) = -[y^{(i)} \log(p_i) + (1 - y^{(i)}) \log(1 - p_i)]$$

Gradient of this loss function

Here is the gradient of the loss function (proof left as an exercise to the reader):

$$\nabla L(\theta) = \left(\frac{1}{N} \cdot \sum_{i=1}^{N} \sigma(\theta^T X^{(i)}) X^{(i)}\right) - \left(\frac{1}{N} \cdot \sum_{i=1}^{N} y^{(i)} X^{(i)}\right)$$

This component does not involve the labels, $y^{(i)}$, and as such can be computed in the clear. No need for MPC!

This component does not involve the model params, θ, meaning it only needs to be computed once! We just compute this once (in MPC) and we are done! We can replace it with a "noisy" version.

Noisy Gradient

This “noisy gradient” is probably close enough to be useful

$$\text{noisy-}\nabla L(\theta) = \left(\frac{1}{N} \cdot \sum_{i=1}^{N} \sigma(\theta^T X^{(i)}) X^{(i)}\right) - \text{noisy-dot-product},$$

This component is computed each step of gradient descent in the clear.

This is the component computed (once) in MPC

How does training work with WALR?

1. Initialize θ to random values
2. Compute the “noisy dot-product” in MPC
3. Compute the “gradient” of the “Loss function” in the clear using the “noisy dot-product”, the current value of θ and the features X.
4. Update θ by taking a small step in the direction of the gradient
5. As long as θ hasn’t converged, go back to step 3.

The tl;dr of WALR

We can privately train a Logistic Regression model (label privacy paradigm), in a very optimal way:

- The only thing we need to compute in MPC is just:
    - $\Sigma(X^{(i)} \cdot y^{(i)}) + \mathrm{random\_noise}$
- We can do the rest of the work outside of the MPC, in the clear, since it doesn't depend upon the labels $y^{(i)}$

This is pretty awesome. It means we only need to add DP noise one time, not every step. It's also a really simple function that's cheap to compute in MPC.

What are we computing in MPC exactly?

$\Sigma(X^{(i)} \cdot y^{(i)})$

Feature 1	**Feature 2**	**Feature 3**	**…**	**Feature k**	**Label**
1	0	1	…	1	**1**
1	1	0	…	1	**1**
0	1	1	…	0	**0**
0	0	0	…	1	**1**
1	1	1	…	1	**0**
1	1	0	…	0	**1**
3	**2**	**1**	**…**	**3**	**Totals**

How much random noise do we need to add?

- What's the maximum change in the L2 norm when *one label* changes?
    - Assuming there are `k` features
    - Features are all binary values
    - The maximum change would be adding a vector of all ones.
    - L2 norm changes by `sqrt(k)`
- Conclusion: add random gaussian noise to each coordinate
    - Variance: $2*k*\log(1.25/\delta)/\varepsilon^2$

Comparison with “noisy labels”

- Result is an aggregate sum.
  - Some PAT-CG members might be more comfortable with ***aggregate*** outputs
  - Assuming this is an aggregation across many users, the output is not linkable to any individual
- Central DP model (not local DP)
  - Intuitively seems like this should provide a better privacy ⇔ utility trade off (at scale)
- More “purpose limited”
  - It’s difficult to enumerate all possible downstream uses of “noisy labels”
  - This single, aggregate output is less flexible (pros and cons here…)