Training Data Influence

Our starting point

- Hammoudeh and Lowd's 2023 Survey
- H&L for short in this slide deck
- Covers 'early' stats work (1980s), more 'recent' ML work (2017)
- Gives us a taxonomy of influence approaches and definitions

Basic definition

Imagine we have ourselves a nice, trained 'final model'. How much of its 'value' boils down to row 537 from our training set?

The grand question of influence

- How much did playing high school sports impact your ability to be successful in CS courses?
- How much did that extra serving of fries contribute to my stomach ache last night?
- How much did this particular set of lecture slides contribute to your learning outcomes in this course?

In general, these are all very hard questions. Quantitative social scientists, especially in economics and sociology, like to try to answer them when possible using experiments and clever analysis approaches.

But often in real life, we'll just never know how your CS career would have gone if you {did / did not} play high school {basketball / hockey / baseball}.

But models *should* be traceable

- In the case of models, however... why can't we figure out exactly how much row 537 contributed? It's all just math, right?

Why we want to do this

- a number of reasons given by H&L
- detect anomalies
- distribution shifts
- measurement error
- human labeling errors
- adversarial actors

Influence and a data economy?

- another big one: potential implications for a new post-AI 'data economy'

Can we calculate influences?

- Depends on the definition of influence we use and the particular case
- definitions can differ quite a bit
- sometimes 'provably hard' (in the CS sense)
- we can always estimate (but how good are the estimates?)

Terms

- Let's call our training data D
- Let's call our row of interest D_i (for now, more notation coming)

Different ways to relate the model to D

- influence analysis, aka data valuation, aka data attribution: which pieces of training data get credit (and how much) for a specific model output
- Leave-one-out influence = difference in a chosen evaluation metric with D_i vs. without D_i (e.g., loss on a test point)

Brute force influence

"We can easily get all influences for all rows in D. Just retrain n times!"

Well, if retraining was free...

Influence estimation

- Research area has sprung up around estimating training data influence
  - Made especially popular after the publication of: Pang Wei Koh and Percy Liang. “Understanding Black-box Predictions via Influence Functions”. In: Proceedings of the 34th International Conference on Machine Learning. ICML’17. Sydney, Australia: PMLR, 2017. url: https://arxiv.org/abs/1703.04730.]

About H&L's paper

- It's a survey of the many perspectives, definitions, and estimation approaches for training data influence
- The authors taxonomize the approaches

For the purposes of our course

- You do not need to understand everything in this paper for CMPT 419! But we will be able to get a lot out of it.

H&L's notation

$[r]$ is set of integers $1, ..., r$, i.e. the integers from 1 to r.

$A \overset{m}{\sim} B$ means that A is a set of cardinality m (it has m items) drawn uniformly at random from some other set B.

2^A is the *power set* of set A (set of all subsets, all combinations big and small). WP article is helpful here: https://en.wikipedia.org/wiki/Power_set.

$A \setminus B$ is set subtraction (remove stuff in set B from set A). See visual example here: https://www.mathwords.com/s/set_subtraction.htm.

More notation

We use the indicator function $1[a]$ for some condition a.

The indicator is 1 when a is true, and 0 otherwise.

More notation

$x \in \mathscr{X} \subseteq \mathbb{R}^d$

is a feature vector

$y \in \mathscr{Y}$ is our target

D is our training set

it's a set of n tuples z_i, each tuple is (x_i, y_i)

- subscript i means it's a training example
- subscript te means it's a test example, e.g. z_{te} is (x_{te}, y_{te})

Our model is f

It's a function that maps from $\mathscr{X}$ to $\mathscr{Y}$

Parameters are $\theta \in \mathbb{R}^p$

$p := |\theta|$, i.e. p is our number of parameters.

We have some loss function l.

We define the per-instance loss $L(z;\theta) := l(f(x;\theta), y)$.

The empirical risk over a dataset D is $L_D(\theta) := \frac{1}{n}\sum_{z_i \in D} L(z_i;\theta)$. For loss and risk, smaller is better.

Some assumptions (and more notation)

We also assume $p \gg d$ (many more parameters than columns in our data; common in modern deep nets)

Using first-order optimization (e.g., gradient descent) with T iterations

Start with initial params $\theta^{(0)}$ and we get new params $\theta^{(t)}$ at each time step t

Other hyperparams:

- learning rate $\eta^{(t)} > 0$
- weight decay λ

We won't worry too much about hyperparameters when we're dealing with data valuation -
- we kind of assume "all that stuff is sorted, let's worry about the data!"

Gradients

We get *training gradients* defined as

$$\nabla_\theta L(z_i; \theta^{(t)})$$

That is, the gradient (with respect to parameters θ) of the loss for instance z_i at time step t. Wikipedia's article on the gradient in vector calculus is pretty helpful as far as math articles go: https://en.wikipedia.org/wiki/Gradient

This will be important!

Hessian

empirical risk hessian is

$H_\theta^{(t)} := \frac{1}{n} \sum_{z_i \in D} \nabla_\theta^2 L(z_i; \theta^{(t)}).$

Data missing some subset

$D \setminus z_i$ is D without instance z_i

We can write $\theta^{(t)}_{D \setminus z_i}$ to mean the parameters when trained with that instance missing

You might begin to imagine why this will be useful!

Some vocab

- *proponents*, *excitatory examples*: training examples with positive influence, loss goes down when the example is added, which is "good".
- *opponents*, *inhibitory examples*: training examples with negative influence, loss goes up when the example is added, which is "bad".
- *pointwise influence*: effect of single instance on single metric (e.g. test loss)

Gradients Review

Quick review of training gradients

Imagine we're trying to predict a single output value y from a single input value x using a simple neural network. The network's prediction $\hat{y}$ is given by $\hat{y} = wx + b$

Where:

- w is the weight of the connection between the input and output neuron.
- b is the bias.
- x is the input.

Objective of our training

- want to adjust w and b to get $\hat{y}$ close to real y
- measure how we're doing with *loss*
- example choice: Mean Squared Error (MSE)
- $L = \frac{1}{2}(y - \hat{y})^2$

Training Gradients in this example

Training gradients indicate in which direction (and by how much) we should adjust our params (w and b) to minimize loss

To find these gradients, we use backpropagation, which involves calculating the derivative of the loss function with respect to each parameter

Note on terms

In our reading, we use
θ to refer to a vector of arbitrary number of params (cardinality is p).

So in this example we can assume

$\theta =< w, b >$ and $p = 2$ (there are just two params)

Gradient wrt w

$$\frac{\partial L}{\partial w} = -(y - \hat{y}) \cdot x$$

tells how a small change in w affects loss.

If $\frac{\partial L}{\partial w}$ is positive, increasing w will increase the loss, so we should decrease w to reduce the loss.

How did we get gradient wrt w?

- plug in $\hat{y}$ into L
- $L = \frac{1}{2}(y - (wx + b))^2$
- or, just apply chain rule
- $\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w}$

How did we get gradient wrt w?

- $\frac{\partial L}{\partial \hat{y}} = \frac{\partial}{\partial \hat{y}}(\frac{1}{2}(y - \hat{y})^2) = -(y - \hat{y}) = \hat{y} - y$
- $\frac{\partial \hat{y}}{\partial w} = \frac{\partial}{\partial w}(wx + b) = x$

Chain rule helps out

- Note that we can also just multiply everything out and compute partial derivatives without the chain rule
- Chain rule is just convenient for a composite function. Here $\hat{y}$ is a function of w, x, and b.
- Note that if we have an activation function, it's even more helpful...

Example with actual values, 1/n

Suppose

- Input value $x = 2$
- Actual output value $y = 5$
- Weight $w = 1$
- Bias $b = 1$

Want gradient of L wrt w still

Example with actual values, 2/n

($x = 2, y = 5, w = 1, b = 1$)

- $\hat{y} = wx + b = 1 * 2 + 1 = 3$
- $L = \frac{1}{2}(y - \hat{y})^2 = \frac{1}{2}(5 - 3)^2 = 2$
- $\frac{\partial L}{\partial w} = -(y - \hat{y})x = -(5 - 3) * 2 = -4$

Example with actual values, 3/n

- Small increase in w will decrease loss, so adjust w upwards
- We make our update based on learning rate, η

Gradient wrt b

$$\frac{\partial L}{\partial b} = -(y - \hat{y})$$

tells us how a small change in b affects the loss. Similarly, if $\frac{\partial L}{\partial b}$ is positive, increasing b will increase the loss, so we should decrease b to reduce the loss.

How did we get gradient wrt b

- $\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b}$
- $\frac{\partial L}{\partial \hat{y}} = -(y - \hat{y})$
- $\frac{\partial \hat{y}}{\partial b} = \frac{\partial}{\partial b}(wx + b) = 1$
- so, $\frac{\partial L}{\partial b} = -(y - \hat{y})$

Updating params

We update w and b using a learning rate η (a kind of step size, how far we adjust things based on the direction of our gradients):

$$w = w - \eta \frac{\partial L}{\partial w}$$

$$b = b - \eta \frac{\partial L}{\partial b}$$

Iteration

This process is repeated for many iterations (or epochs) over the training data, gradually reducing the loss and making the predictions $\hat{y}$ closer to the actual outputs y

Other resources

- We ignored activation function here
- Exercise for the reader! What if we add ReLU or a logistic activation function (hint: now we have $z = wx + b$ and $\hat{y} = a(z)$, so we have to do a 3-piece chain rule)
- Worth reviewing longer backprop materials^[see e.g. https://www.cs.toronto.edu/~rgrosse/courses/csc311_f20/slides/lec04.pdf, http://neuralnetworksanddeeplearning.com/chap2.html]

Relevance back to training data influence

- Gradient-based methods rely on the idea that training data only influences the final model via the gradients produced
- We should be able to keep track of these gradients to understand how a training instance affected a test instance

For a data valuator

- We can assume gradients are being calculated throughout training
- pytorch: `requires_grad=True`
- another example: https://fluxml.ai/Flux.jl/stable/models/basics/