GPU-Accelerated Machine Learning with RAPIDS

CSCS-USI Summer School

Young-Jun Ko (NVIDIA)

29.07.2021

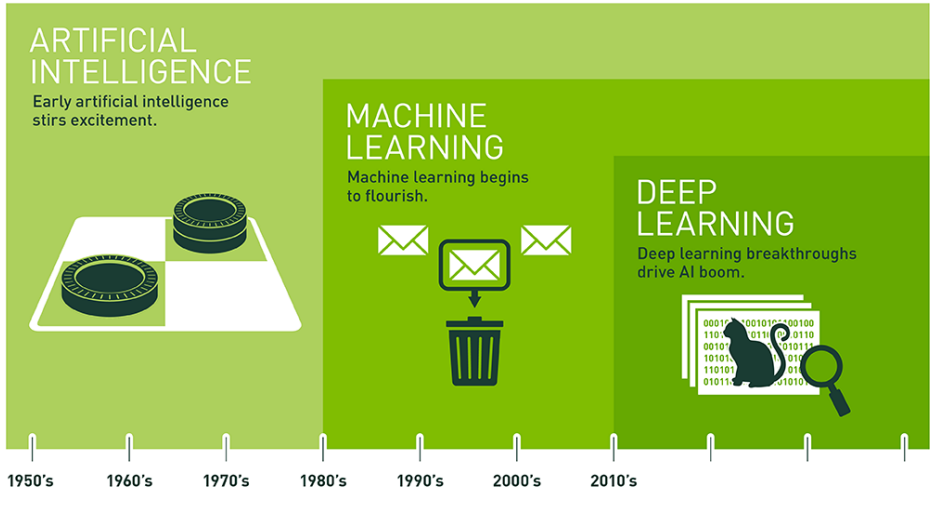
Course Overview and Organization

- Challenge of Data Science: we generate more and more data that needs to be processed and analyzed
- GPU accelerators, often associated with DL, can be helpful also in this broader context
- Requirements for adoption: Software
    - familiar, high-level APIs
    - open-source, community, eco-system
    - painless integration into existing workflows
- Part 1 (motivation): what is RAPIDS, and how does it fit into this context
- Part 2 (concepts): taking a look at some of the algorithms behind the high-level APIs
- Format: Presentation interleaved with small exercises (Notebooks in the Summerschool repository)
    - Presentation of a topic
    - Small exercise
    - Goal is not to finish everything, but to briefly play with the presented concepts

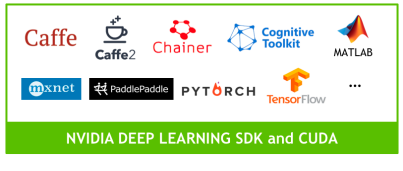
Outline

- The RAPIDS data science stack and the case for "Classical" ML
- Supervised learning
    - Recap of Fundamental Concepts
    - Generalized linear models (GLMs)
    - Gradient-boosted decision trees, XGBoost

ML and Libraries at NVIDIA



Since an early flush of optimism in the 1950s, smaller subsets of artificial intelligence – first machine learning, then deep learning, a subset of machine learning – have created ever larger disruptions.



Classical ML Libraries: Workhorses of Data Science

François Chollet @fchollet · Apr 3

We often talk about how following UX best practices for API design makes Keras more accessible and easier to use, and how this helps beginners.

But those who stand to benefit most from good UX *aren't* the beginners. It's actually the very best practitioners in the world.

1 7 50

François Chollet @fchollet · Apr 3

Because good UX reduces the overhead (development overhead & cognitive overhead) to setting up new experiments. It means you will be able to iterate faster. You will be able to try more ideas.

And ultimately, that's how you win competitions or get papers published.

2 11 78

François Chollet @fchollet · Apr 3

So I don't think it's mere personal preference if Kaggle champions are overwhelmingly using Keras.

Using Keras means you're more likely to win, and inversely, those who practice the sort of fast experimentation strategy that sets them up to win are more likely to prefer Keras.

8 8 74

Joshua Patterson @datametrician · Apr 3

Replying to @fchollet

This is the fundamental believe that drives @RAPIDSai. @nvidia #GPU infrastructure is fast, people need to iterate quickly, people want a known #python interface. Combine them and you're off to the races!

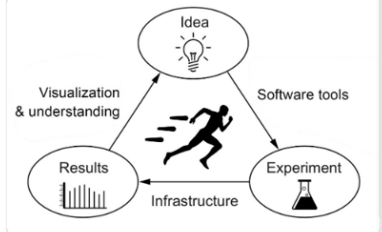
2 11

François Chollet @fchollet

Following

Winners are those who went through *more iterations* of the "loop of progress" -- going from an idea, to its implementation, to actionable results. So the winning teams are simply those able to run through this loop *faster*.

And this is were Keras gives you an edge.



12:31 PM - 3 Apr 2019

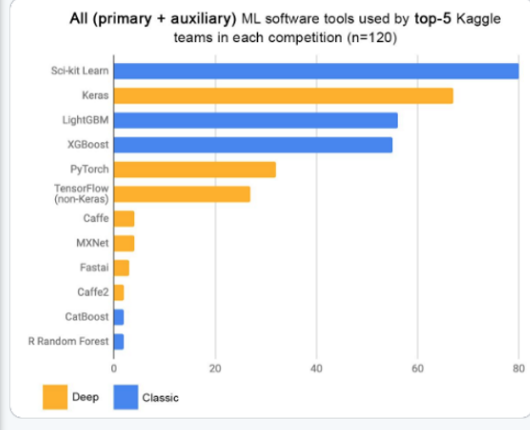
50 Retweets 158 Likes

5 50 158

François Chollet @fchollet · Apr 3

The second question asked about secondary frameworks -- usually teams win with an ensemble that involves many different ML frameworks. Here are *all* frameworks used.

Sklearn tops that ranking: everyone uses sklearn (although often as an auxiliary, for preprocessing or scoring).



4 44 129

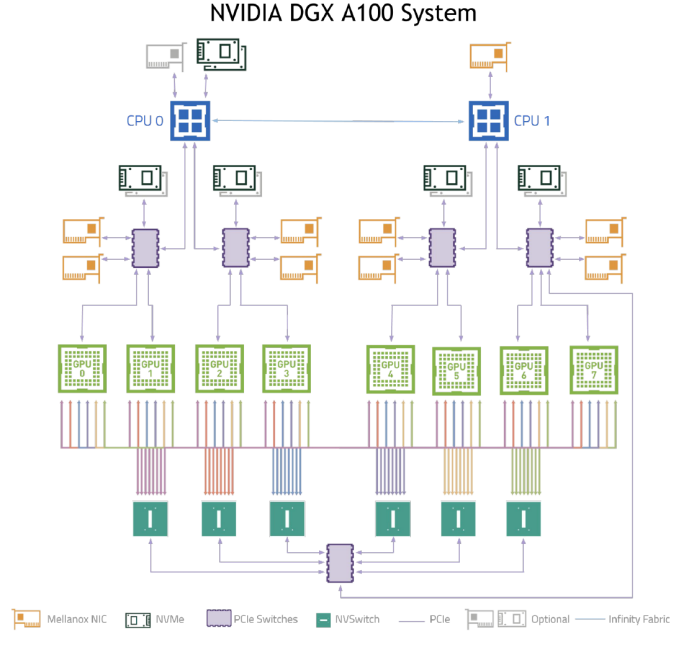
kaggle

Why GPUs for Data Science?

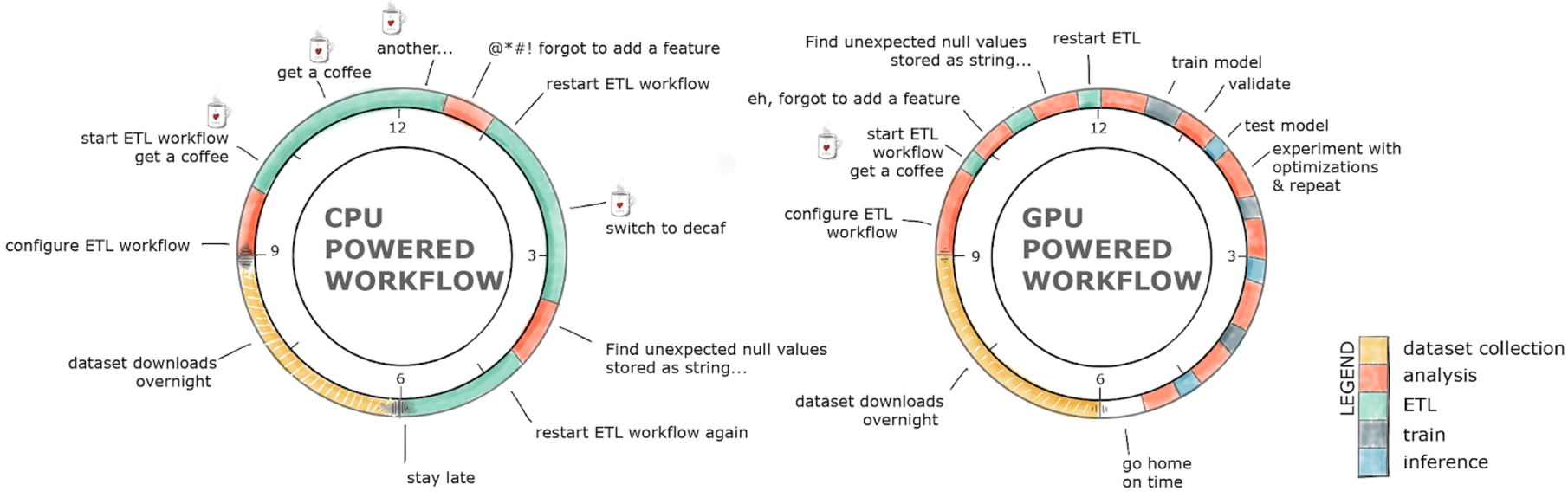
Numerous hardware advantages

- Thousands of cores with up to ~20 TeraFlops of general purpose compute performance
- Up to 1.5 TB/s of memory bandwidth
- Hardware interconnects for up to 600 GB/s bidirectional GPU <--> GPU bandwidth
- Can scale up to 16x GPUs in a single node

Almost never run out of compute relative to memory bandwidth!



RAPIDS Value Proposition



What is RAPIDS?

End-to-End GPU Accelerated Data Science

cuDF

- GPU-accelerated ETL functions
- Tracks pandas and other common PyData APIs
- Dask + UCX integration for scaling

cuML

- GPU-native ML library, including XGBoost, FIL, HPO, and more

cuGraph

- GPU graph analytics, including TSP, PageRank, and more

cuxfilter

- GPU-accelerated cross-filtering

pyViz integration

- Plotly Dash, Bokeh, Datashader, HoloViews, hvPlot

Domain-Specific Libraries

CLX + Morpheus
Cyber log processing + anomaly detection

cuStreamz
Streaming analytics

cuSignal
Signals processing

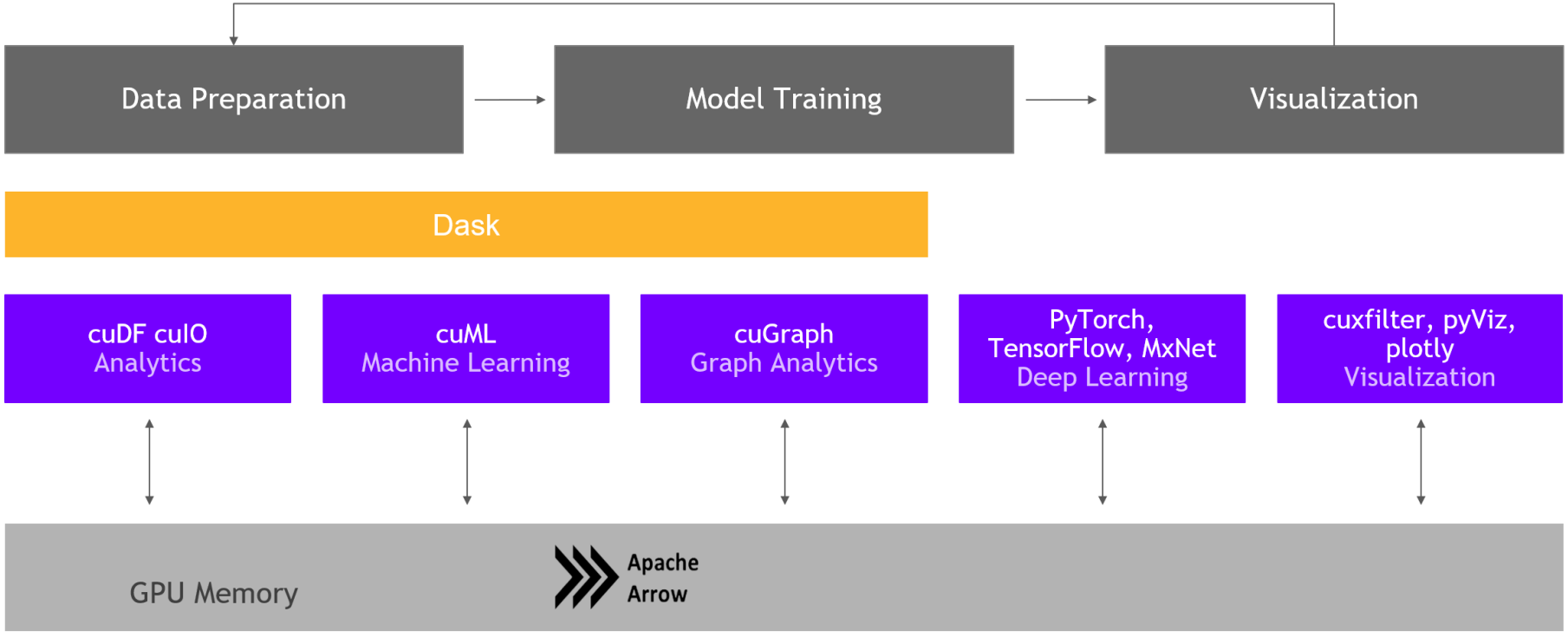
cuCIM
Computer vision & image processing primitives

...and more!

cuSpatial
Spatial analytics

node-RAPIDS
Bindings for node.js

RAPIDS Overview: End-to-end Datascience Platform



Move Faster with Familiar APIs

In many cases:

```
# ML on the CPU
from scikit import Model
model = Model()
model.fit(train_data)
preds = model.predict(test_data)
```

becomes:

```
# ML on the GPU with RAPIDS
from cuml import Model
model = Model()
model.fit(train_data)
preds = model.predict(test_data)
```

RAPIDS: Familiar APIs

Pandas/Scikit-learn	RAPDS cuDF/cuML

Classic ML Algorithms (cuML Library)

- Let us not forget about classic ML (i.e. not deep learning)

What you need: Classic Machine Learning - cheap, realiable, well-understood **What you want: Deep Learning - powerful, resource-hungry, cutting-edge**

Classical ML Libraries: Good to Know, Have in your Toolbox

Tabular Data: Deep Learning is Not All You Need

Ravid Shwartz-Ziv RAVID.ZIV@INTEL.COM
IT AI Group, Intel

Amitai Armon AMITAI.ARMON@INTEL.COM
IT AI Group, Intel

Abstract

A key element of AutoML systems is setting the types of models that will be used for each type of task. For classification and regression problems with tabular data, the use of tree ensemble models (like XGBoost) is usually recommended. However, several deep learning models for tabular data have recently been proposed, claiming to outperform XGBoost for some use-cases. In this paper, we explore whether these deep models should be a recommended option for tabular data, by rigorously comparing the new deep models to XGBoost on a variety of datasets. In addition to systematically comparing their accuracy, we consider the tuning and computation they require. Our study shows that XGBoost outperforms these deep models across the datasets, including datasets used in the papers that proposed the deep models. We also demonstrate that XGBoost requires much less tuning. On the positive side, we show that an ensemble of the deep models and XGBoost performs better on these datasets than XGBoost alone.

Summary

- Growing dataset sizes make data science challenging
    - Data exploration, cleanup, visualization, predictive modeling with "classic" methods
- Optimize for most valuable resource: scientists'/engineers', i.e. your time!
    - Get more iterations in: more experiments
    - Quick analysis with basic methods: well-understood, cheap, robust
- Leveraging hardware acceleration in this space requires easy-to-use tools
- Enter RAPIDS: open-source, actively maintained, integrated into the python eco-system
    - Goal: help you make better use of your time, reducing *both*, dev time *and* running time
    - How:
        - seamless, "drop-in" replacement using familiar *high-level* APIs
        - GPU-accelerated implementation
    - Both, ML and DL workflows can benefit
    - For more features, docs, blog, etc. head over to: rapids.ai ()

Outline

- The RAPIDS data science stack and the case for "Classical" ML
- Supervised learning
    - Recap of Fundamental Concepts
    - Generalized linear models (GLMs)
    - Gradient-boosted decision trees, XGBoost

Recap of ML Fundamentals

- Disclaimer: mild notational abuse and a lot of hand-waving ahead!
- Conceptual intuition > mathematical rigour

Checklist of Concepts

- Supervised learning
- Families of functions as models, learning algorithms, and loss functions
- Generalization, risk, and the bias-variance tradeoff

Supervised Learning

- The most common/successful paradigm
- The setup:
    - Input/feature space: $\mathcal{X}$, e.g. $\mathbb{R}^d$
    - Output/target space: $\mathcal{Y}$, e.g. $\mathbb{R}$ (regression), $\{\pm 1\}$ (classification)
- Hidden relationship: $P(X, Y)$
    - Probabilistic perspective useful for acknowledging noise
    - P is unknown (but we'll encounter it again in a moment)
    - But, we have access to N i.i.d. samples $(x_i, y_i) \sim P(X, Y)$
- Goal:
    - In practice: $P(Y \mid X)$
    - Find a "good" predictor, i.e. a deterministic $f : \mathcal{X} \mapsto \mathcal{Y}$

Example: Regression in 1D

- $\mathcal{X} = \mathbb{R}$
- $\mathcal{Y} = \mathbb{R}$
- find $f : \mathbb{R} \mapsto \mathbb{R}$, that can answer queries like "What is y at $x = 0.4$?"
- Learnt from previous "experience", i.e. pairs of (x_i, y_i)

Checklist of Concepts

- Supervised learning: $\mathcal{X}, \mathcal{Y}, P(X, Y), f$
- Families of functions as models, learning algorithms, and loss functions
- Generalization, risk, and the bias-variance tradeoff

Models, Learning Algorithms and Loss Functions

- We choose a family of functions $\mathcal{F}$ from which we choose the predictor: $f \in \mathcal{F}$
    - We will see families of linear and piece-wise constant functions
- Here, we assume that f is parameterized by a fixed-sized set of parameters θ, which we can use to index elements in $\mathcal{F}$
    - So called "parametric" models
- I.e. we can identify $f_\theta \in \mathcal{F}$ by their parameters θ
- A learning algorithm would then be a function that takes a sample and returns a set of parameters:

$$\mathcal{A}_{\mathcal{F}}(\{(x_i, y_i)\}) = \hat{\theta}$$

- It "fits the model to the data", which implies a way of quantifying "good" and "bad" (but there might be multiple notions involved!)
- Loss function:

$$l : \mathcal{Y} \times \mathcal{Y} \mapsto \mathbb{R}$$

    - Compares a true $y \in \mathcal{Y}$ to a predicted $\hat{y} = f(x)$
    - Encodes the cost we assign to errors
    - Examples:
        - squared loss: outliers cost a lot - avoid

$$l(y, \hat{y}) = (y - \hat{y})^2$$

        - absolute-error: outliers cost less - tolerate

$$l(y, \hat{y}) = |y - \hat{y}|$$

Example: The family of order-k polynomials

- Functions of this form: $f_\theta(x) = \sum_{i=0}^{k} w_i x^i$ ($k+1$ terms including bias)
- Parameters $\theta = \{w_0, \ldots, w_k\}$
- Family $\mathcal{F} = \{f_\theta\}$

- Learning algorithm and loss: e.g. least-squares regression when using squared error

$$\min_\theta \frac{1}{2}\|y - f_\theta(x)\|^2 = \min_\theta \frac{1}{2}\|y - \Phi(x)w\|^2 \qquad \mathcal{A}_\mathcal{F}(x, y) = (\Phi^T\Phi)^{-1}\Phi^T y$$

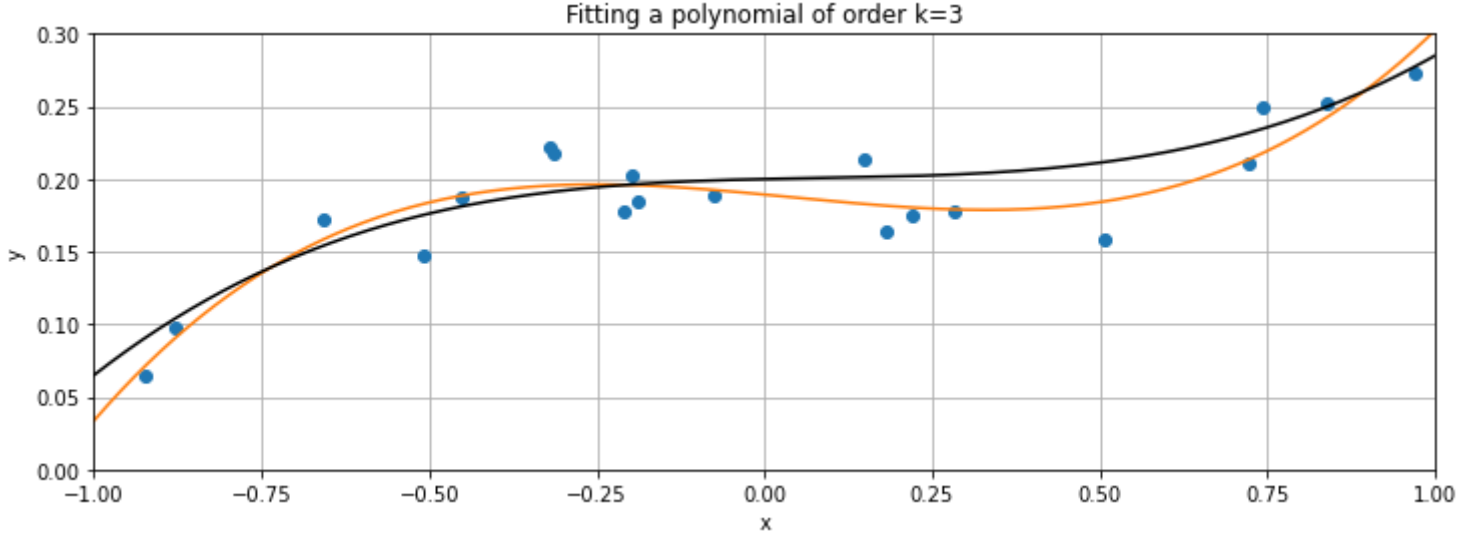
- Side note: linear in the parameters θ, but non-linear feature transform $\phi_i(x) = x^i$
    - Neural networks can be thought of as making the feature map trainable!

- In code:

```
def phi(x, order):
    Phi = np.concatenate([x ** k  for k in range(order+1) ], axis=1 )
    return Phi

def lsq(X, y):
    A = np.dot(X.T, X)
    b = np.dot(X.T, y)
    return np.linalg.solve(A,b)
```

Example of fitted Model



Exercise 1 (15 min)

Checklist of Concepts

- Supervised learning: $\mathcal{X}, \mathcal{Y}, P(X, Y), f$
- Families of functions as models, learning algorithms and loss functions: $f_\theta \in \mathcal{F}, \mathcal{A}_\mathcal{F}$, $l(y, \hat{y})$
- Generalization, risk, and the bias-variance tradeoff

The True Risk

- If we knew P (and could deal with it), for any predictor, we would want to compute the expected loss on the whole population to understand generalization

$$R(f) := \mathbb{E}_P[l(Y, f(X))]$$

- The risk is the ideal objective

$$f^* = \arg\min_f R(f) \qquad \text{or at least} \qquad f^*_{\mathcal{F}} := \arg\min_{f \in \mathcal{F}} R(f)$$

- Unfortunately, we can only *approximate* R statistically using our sample, i.e. calculate the *emprirical* risk

$$\hat{R}(f) := \frac{1}{N} \sum_{i=1}^{N} l(y_i, f(x_i))$$

- And in practice, *empirical risk minimization*:

$$f_N := \arg\min_{f \in \mathcal{F}} \hat{R}(f)$$

- We have:

$$R(f^*) < R(f^*_{\mathcal{F}}) < R(f_N)$$

Empirical Risk and Generalization

- More importantly, $R(f_N)$ and $\hat{R}(f_N)$ can be completely independent of each other
    - $\hat{R}$ is *not* a useful estimate of the generalization of f_N (could be 0!)
    - We "used up" the sample for fitting the model
- I.e. the empirical risk will not tell us anything about the generalization error (the error on the whole population)
- However, generalization is all we care about
- Need another dedicated test sample, to estimate $R(f_N)$!
    - Check if fitting the model captured something useful or mostly noise

Decomposing the Error

- Studying these quantities can give us some insights into what we can do about this
- We can compare our estimate f_N with the best possible predictor f^*, and consider the (positive) risk difference:

$$\mathbb{E}[R(f_N) - R(f^*)]$$

- The expectation is taken over the samples (f_N is a random quantity, if we don't condition on the training data)

$$\begin{aligned}
\mathbb{E}[R(f_N) - R(f^*)] &= \mathbb{E}[R(f_N) - R(f^*) + R(f^*_{\mathcal{F}}) - R(f^*_{\mathcal{F}})] \\
&= \mathbb{E}[R(f^*_{\mathcal{F}}) - R(f^*) + R(f_N) - R(f^*_{\mathcal{F}})] \\
&= (R(f^*_{\mathcal{F}}) - R(f^*)) + (\mathbb{E}[R(f_N)] - R(f^*_{\mathcal{F}})) \\
&= \text{Approximation Error} + \text{Estimation Error} \\
&\rightarrow \text{"Bias"} + \text{"Variance"}
\end{aligned}$$

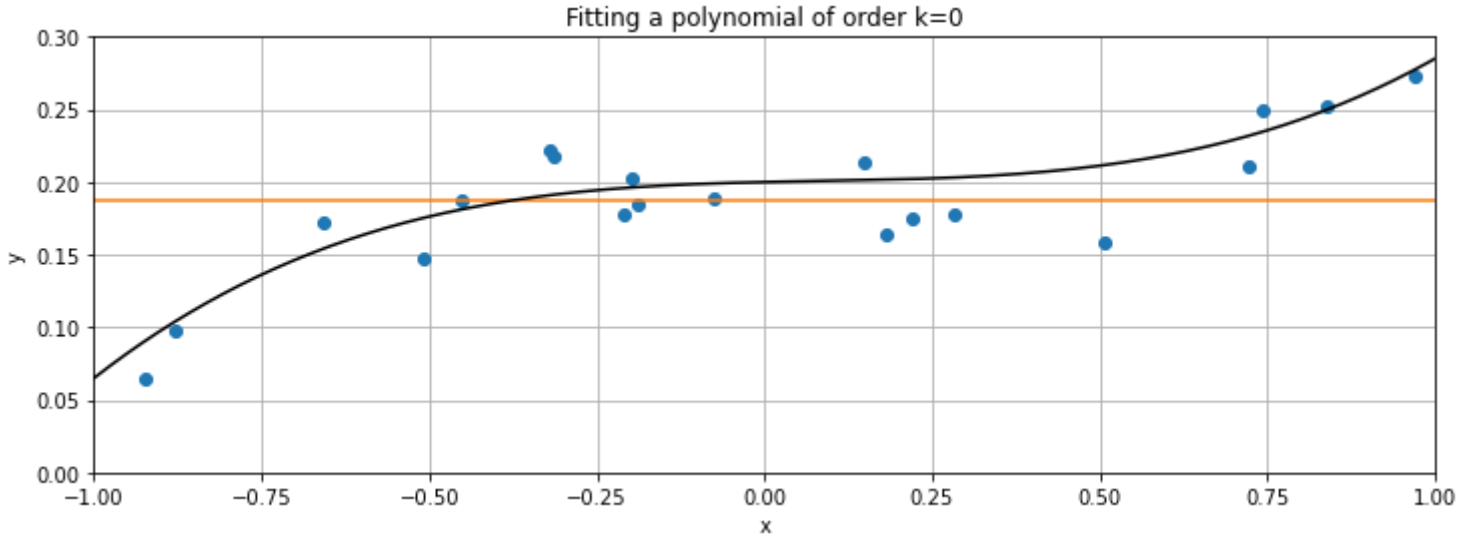
Decomposing the Error

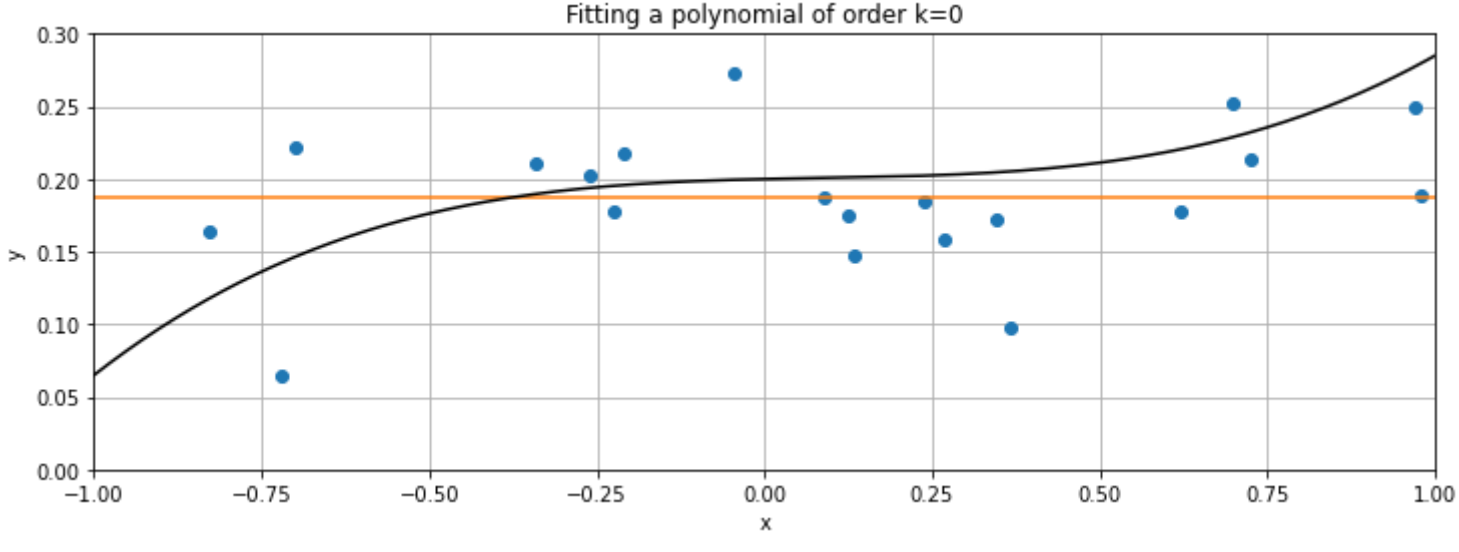
- Approximation error $R(f^*_{\mathcal{F}}) - R(f^*)$
    - How much do we loose because of our choice of $\mathcal{F}$?
    - The more different functions $\mathcal{F}$ offers, the smaller this term can be
- Estimation error $\mathbb{E}[R(f_N)] - R(f^*_{\mathcal{F}})$
    - How much do we loose by fitting on a limited, possibly very noisy sample?
    - The more samples we have, the closer the two terms will be
    - BUT: for a large function class, f_N fitted on different samples can look *dramatically* different
        - f_N has high variance
        - Most of them will be wrong, i.e. very different from $f^*_{\mathcal{F}}$
        - $\Rightarrow$ large estmation error
- For a limited amount of data (as in practice), we cannot "afford" a low approximation error because the estimation error will blow up
- Whereas for a large amount of data, we need a "sufficiently" large function class

Model Selection

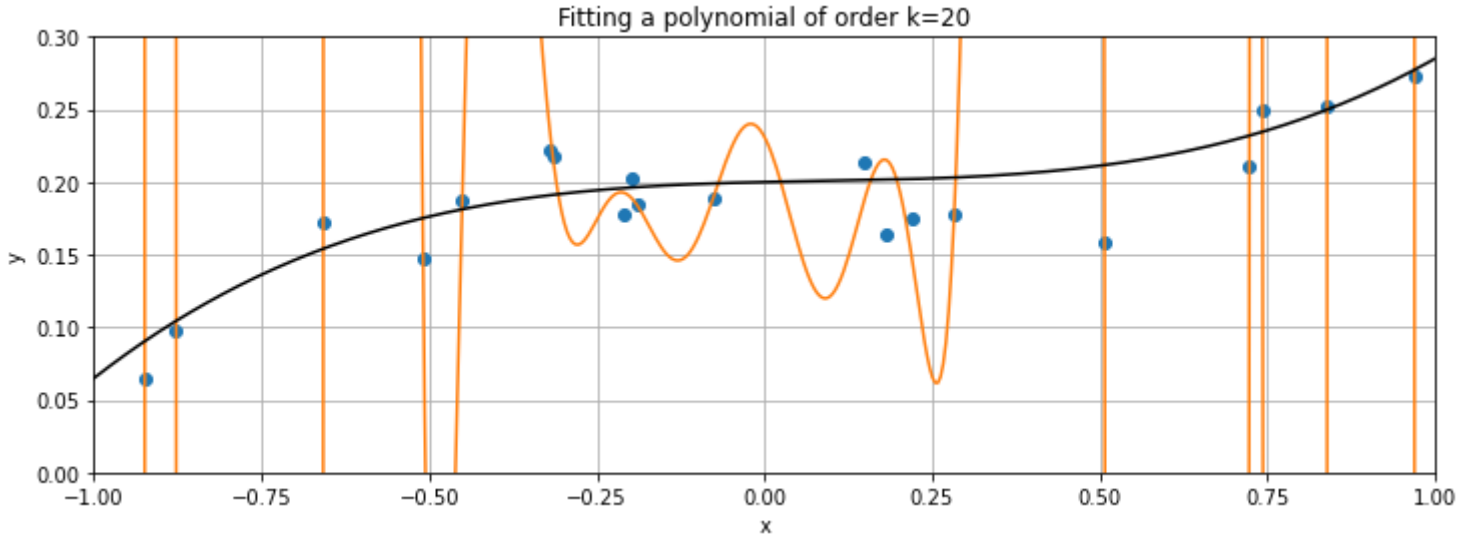
- The main object, we have control over: $\mathcal{F}$
    - allow less functions explicitely (e.g. restrict k)
    - add regularization that penalizes complexity
- Very "small" $\mathcal{F}$, i.e. low model complexity: high bias, low variance - we underfit
- Very "large" $\mathcal{F}$, i.e. high model complexity: low bias, high variance - we overfit
- Have different models in your toolkit
- Beware of optimistic risk estimate

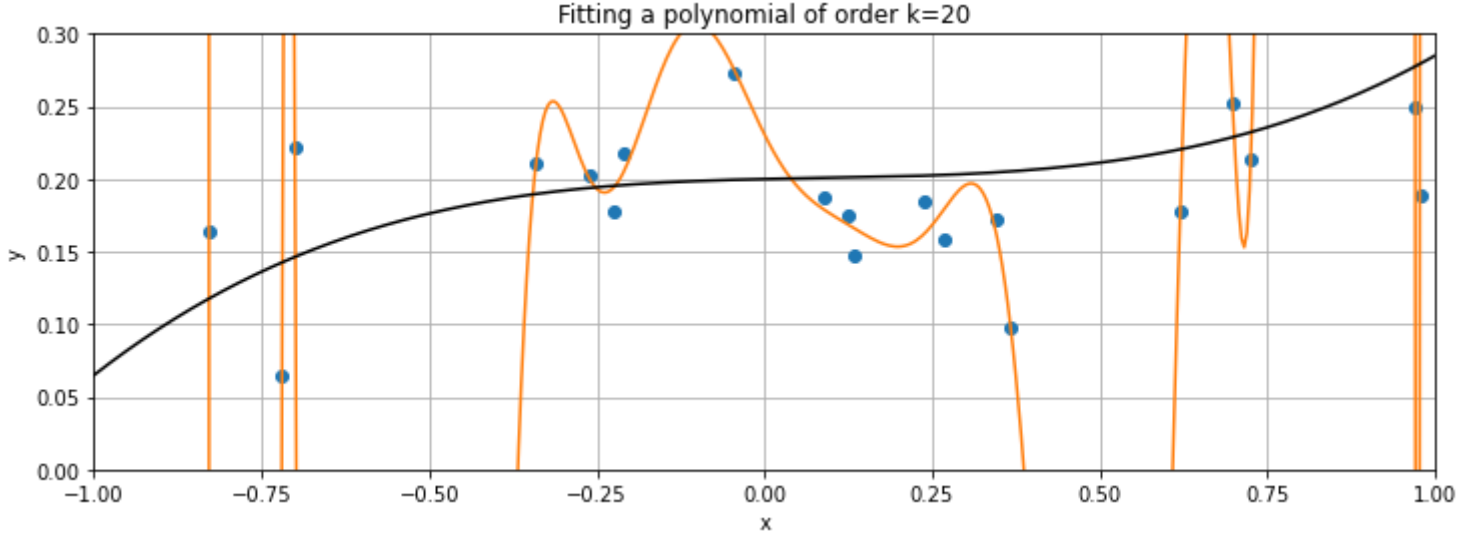
Extreme Underfitting: High Bias - Low Variance


Fitting a polynomial of order k=0
0.30
0.25
0.20
0.15
0.10
0.05
0.00
y
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
x

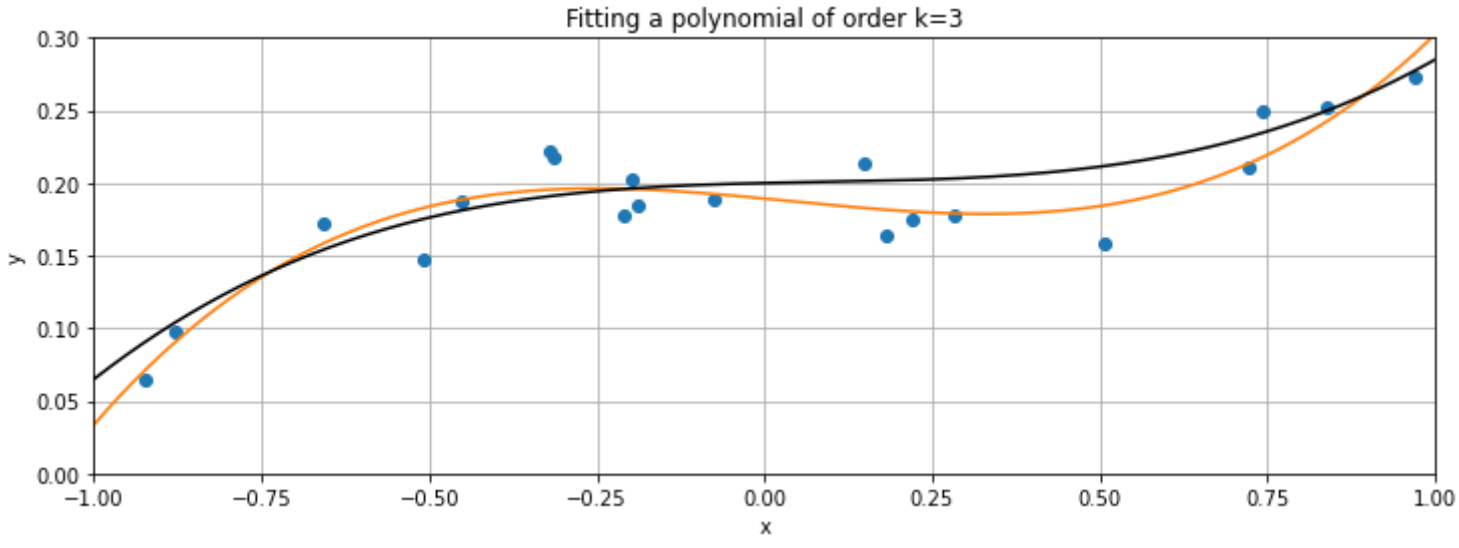

Fitting a polynomial of order k=0
0.30
0.25
0.20
0.15
0.10
0.05
0.00
y
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
x

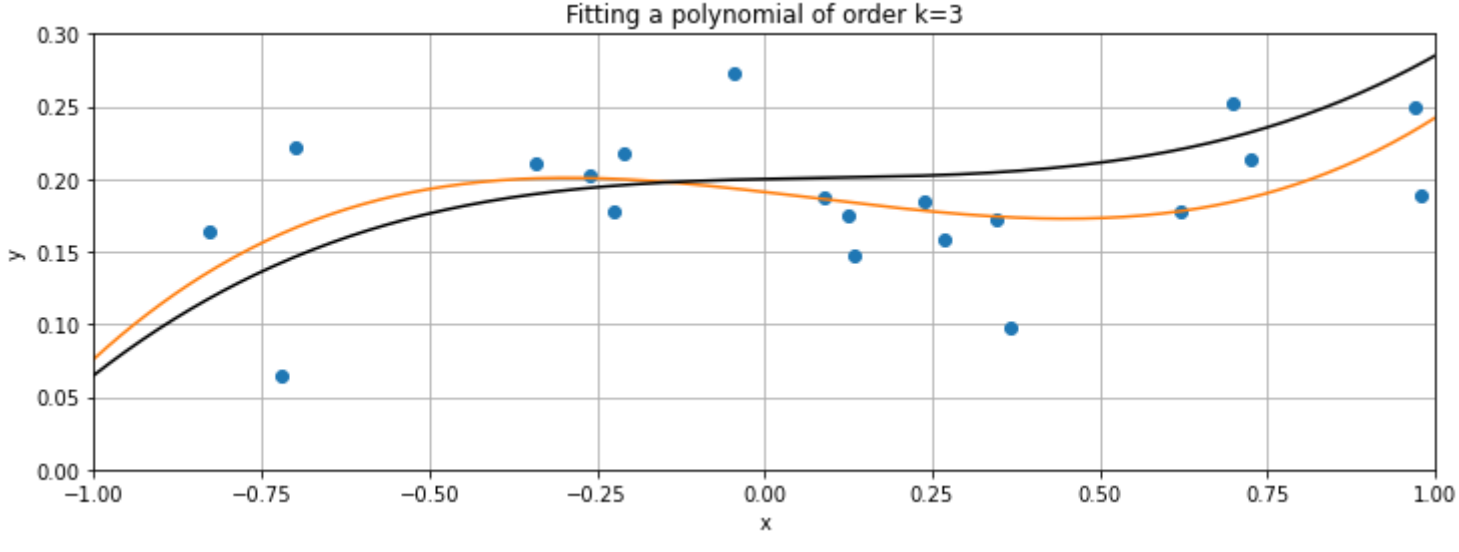
Extreme Overfitting: Low Bias but Arbitrarily High Error


Fitting a polynomial of order k=20
0.30
0.25
0.20
0.15
0.10
0.05
0.00
y
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
x


Fitting a polynomial of order k=20
0.30
0.25
0.20
0.15
0.10
0.05
0.00
y
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
x

Correct Order


Fitting a polynomial of order k=3
0.30
0.25
0.20
0.15
0.10
0.05
0.00
y
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
x


Fitting a polynomial of order k=3
0.30
0.25
0.20
0.15
0.10
0.05
0.00
y
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
x

ML: A Tale of Two Errors

- Error 1: Our choice of function class - Approximation Error
    - How expressive is my function class?
- Error 2: Learning from limited samples - Estimation Error
    - Do I have enough data to determine parameters?
- Relationship between these sources of error: reducing one increases the other

The End?

DEEP DOUBLE DESCENT: WHERE BIGGER MODELS AND MORE DATA HURT

Preetum Nakkiran[*] Harvard University
Gal Kaplun[†] Harvard University
Yamini Bansal[†] Harvard University
Tristan Yang Harvard University
Boaz Barak Harvard University
Ilya Sutskever OpenAI

ABSTRACT

We show that a variety of modern deep learning tasks exhibit a "double-descent" phenomenon where, as we increase model size, performance first gets *worse* and then gets better. Moreover, we show that double descent occurs not just as a function of model size, but also as a function of the number of training epochs. We unify the above phenomena by defining a new complexity measure we call the *effective model complexity* and conjecture a generalized double descent with respect to this measure. Furthermore, our notion of model complexity allows us to identify certain regimes where increasing (even quadrupling) the number of train samples actually *hurts* test performance.

Checklist of Concepts

- Supervised learning: $\mathcal{X}, \mathcal{Y}, P(X, Y), f$
- Families of functions as models, and learning algorithms: $f_\theta \in \mathcal{F}, \mathcal{A}_\mathcal{F}, l$
- Generalization, risk, and the bias-variance tradeoff: R, over-/under-fitting

Summary

- What Machine Learning is all about:
    - Choose an appropriate model (e.g. add regularization)
    - Come up with an efficient learning algorithm
    - Measuring Generalization is itself difficult: often estimates are optimistic (e.g. cross validation)
- Practical ML is by necessity very experimental (we don't have control over some key objects)
- Need tools to iterate quickly:
    - Wide selection of models: software matters
    - It's a computational discipline: performance matters

Exercise 2 (30 min)

Outline

- The RAPIDS data science stack and the case for "Classical" ML
- Supervised learning
    - Recap of Fundamental Concepts
    - Generalized linear models (GLMs)
    - Gradient-boosted decision trees, XGBoost

Logistic Regression - A Generalized Linear Model

- Linear models have been in the toolbox of statisticians for ages
- We've seen linear models for regression
- What about classification, i.e. $\mathcal{Y} = \{\pm 1\}$?
    - we could assume $\mathcal{Y}$ is just $\mathbb{R}$ and fit to the discrete labels. Better way?
- A probabilistic perspective: generative models and maximum likelihood estimation

GLM Perspective on Least-Squares

- Least squares typically derived from an white additive noise model:
$$y = f_\theta(x) + \epsilon, \epsilon \sim N(0, 1)$$
- Implies a Gaussian data likelihood
$$P(Y \mid X = x) = N(Y \mid f_\theta(x), 1)$$
- A Gaussian is appropriate since we are dealing with real values, i.e. regression
- In GLMs, we go the other way around:
    - Depending on the data *type* we would like to model, choose it's likelihood, e.g.:
        - Binary outcomes: Bernoulli
        - Counts: Poisson
    - Model the mean by transforming a linear function, or *linear predictor*
$$\mathbb{E}[Y \mid X] = g^{-1}(f_\theta(x))$$
    - Using the inverse of g from statistics
    - Think of it as mapping the real-valued output of f to the domain of the mean

GLM Perspective on Least-Squares

- For least squares, $\mu = f_\theta(x)$, i.e. $g(x) = x$:

$$P(Y \mid X = x) = N(Y \mid \mu(x), 1) \propto \exp(-\frac{1}{2}(Y - \mu(x))^2)$$

- The likelihood of the data is the probability of the dataset (conditioned on the parameters):

$$P(Y = y \mid X = x, \theta) = \prod_{i=1}^{N} P(Y = y_i \mid X = x_i, \theta)$$

- Maximizing the likelihood is equvalent to minimizing the negative log-likelihood:

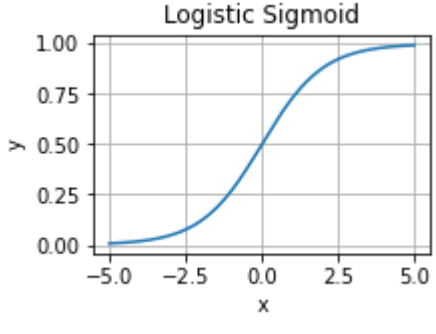
$$\begin{aligned} -\log P(Y = y \mid X = x, \theta) &= -\sum_{i=1}^{N} \log P(Y = y_i \mid X = x_i, \theta) \\ &= \sum_{i=1}^{N} \frac{1}{2}(y_i - \mu(x_i))^2 \\ &= \frac{1}{2}\|y - f_\theta(x)\|^2 \end{aligned}$$

Logistic Regression

- no-one prevents us to model other distributions/data types for y, just need to find the right distribution and parameterization in terms of the predictor
- Now consider a binary RV $Y \in \{0, 1\}$
- Bernoulli distribution: $P(Y = 1 \mid p) = p$ and $P(Y = 0 \mid p) = 1 - p$, can be written as:

$$P(Y = y \mid p) = p^y (1 - p)^{1-y}$$

- The parameter $p \in [0, 1]$ and $\mathbb{E}[Y] = p$
- In logistic regression, we model the mean of Y as $\mu(x) = p(x) = \sigma(f_\theta(x))$
- Logistc sigmoid $\sigma : \mathbb{R} \mapsto [0, 1]$ for a valid probability, with $\sigma(x) = 1/(1 + \exp(-x))$
- Interpretation:
    - Large positive values: we're sure about $Y = 1$
    - Small negative values: we're sure about $Y = 0$
    - Zero: uncertain about the outcome (close to the decision boundary)



Logistic Regresion: Likelihood

- Bernoulli distribution:

$$P(Y = y \mid p) = p^y (1-p)^{1-y} \text{ where } p(x) = \sigma(f_\theta(x))$$

- Neg.log-likelihood:

$$\begin{aligned} -\log P(Y = y \mid X = x, \theta) &= -\log \prod_{i=1}^{N} \sigma(f_\theta(x_i))^{y_i} (1 - \sigma(f_\theta(x_i)))^{1-y_i} \\ &= -\sum_{i=1}^{N} y_i \log \sigma(f_\theta(x_i)) + (1-y_i) \log(1 - \sigma(f_\theta(x_i))) \\ &= \sum_{i=1}^{N} l_\sigma(y_i, f_\theta(x_i)) \end{aligned}$$

- Simplifies a lot when expressed in $Y = \pm 1$
- Key point: l_σ differentiable, we have $\frac{\partial}{\partial f} l_\sigma(y, f)$ and even $\frac{\partial^2}{\partial f^2} l_\sigma(y, f)$

Logistic Regression: Learning Algorithms

- No closed form solution: Iterative, gradient based optimization needed
- For most regularizers a convex problem, well behaved, well studied (can be important in practice/production!)
- Vast literature (primal/primal-dual/dual, stochastic/batch, ...)
- Key ingredient: $\nabla_\theta \sum_{i=1}^{N} l_\sigma(y_i, f_\theta(x_i))$
- Last layer of a NN - Let's do (shallow) back-propagation!
- Forward pass: let $f_\theta(x) = x^T w$
    - MVM: $f = Xw$
    - loss vector: $l = l(y, f)$ (scalar operation)
    - loss value: $L = \sum_i l_i$ (reduction)
- Backward pass (chain rule): compute $\sum_{i=1}^{N} \nabla_\theta l_\sigma(y_i, f_\theta(x_i))$ as $\sum_i \frac{\partial}{\partial f_i} l_\sigma(y, f_i) \cdot \nabla_\theta x_i^T w$
    - loss derivative vector: $d_i = \frac{\partial}{\partial f_i} l_\sigma(y, f_i)$ (scalar operation)
    - MVM: $\nabla_\theta L(\theta) = X^T d$

Summary

- A probabilistic view allows us to generalize to other output spaces
- Differentiable objective: gradient based
- We can apply the same framework and the same learning algorithms, as long as we have differentiable log-likelihood terms
    - Counts: Poisson, Negative-Binomial
    - Multi-class: categorical distribution ("softmax")
    - etc.

Exercise 3 (30 min)

Decision Trees and XGBoost

- One of the most successful methods used on Kaggle
- Technically not part of RAPIDS
- XGBoost
    - has great out of the box performance
    - interpretable by examining the structure
    - has a fast, high-quality implementation (same people behind MxNet DL lib)
- Function class:
    - binary decision trees (think axis-aligned space partitioning)
    - each tree represents a piece-wise constant function
    - each node n either
        - is a leaf node: contains the function value
        - is a split node: has a condition on a single input dimension of the form $x_j < s_n$
    - We map an input (vector) to an output (scalar) by traversing the tree until we hit a leaf and report the value stored there
    - functions now look very different: non-linear

Decision Tree: 1D-Example

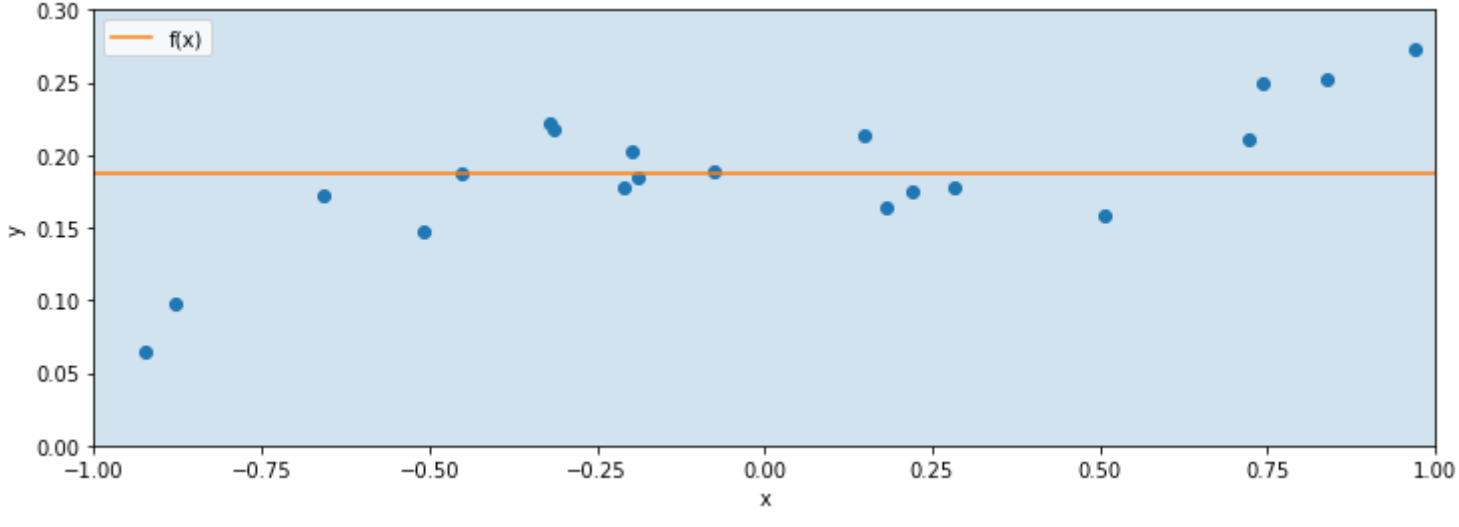
```
class Node:
    def __init__(self):
        self.split = None # split position
        self.left  = None # region left of the split
        self.right = None # region right of the split
        self.value = None # value of the node

def eval1d(node, x):              #traverse the tree to evaluate the function
    if node.split is None:        # no split: it's a leaf
        return n.value            #           report the function value
    if x < node.split:            # the point falls into the left region
        return eval1d(node.left, x)
    return eval1d(node.right, x) # the point falls into the right region
```

- Next:
  - First, some examples
  - How to build the tree

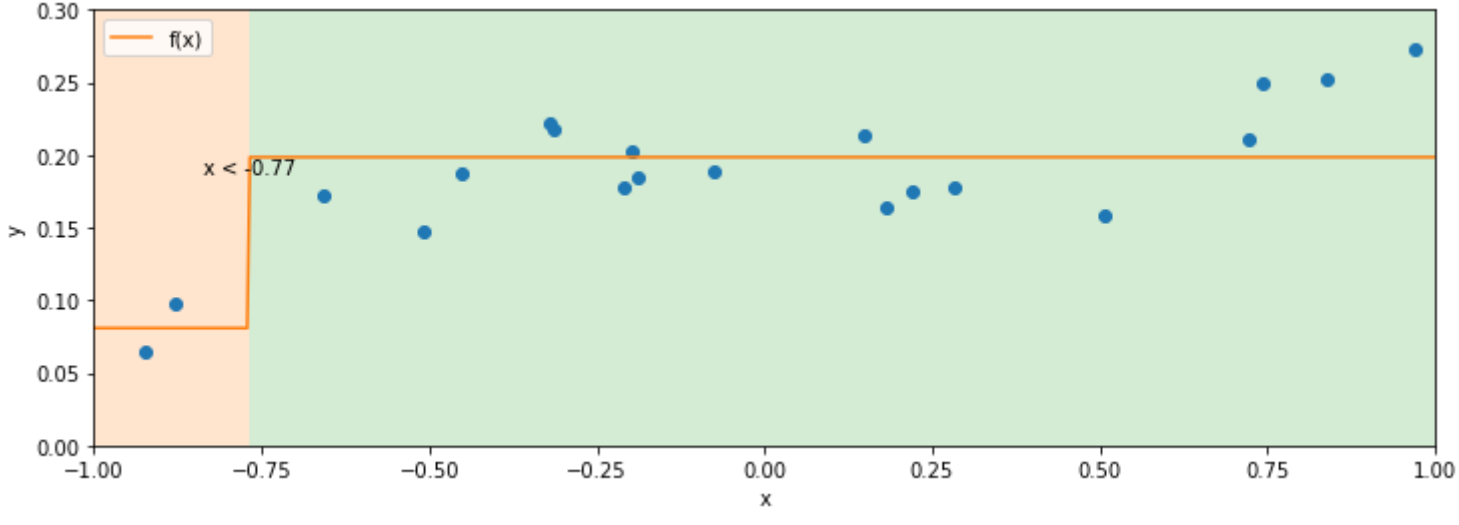
Depth 0

- No split
- A single leaf
- All points assigned
- Constant function value



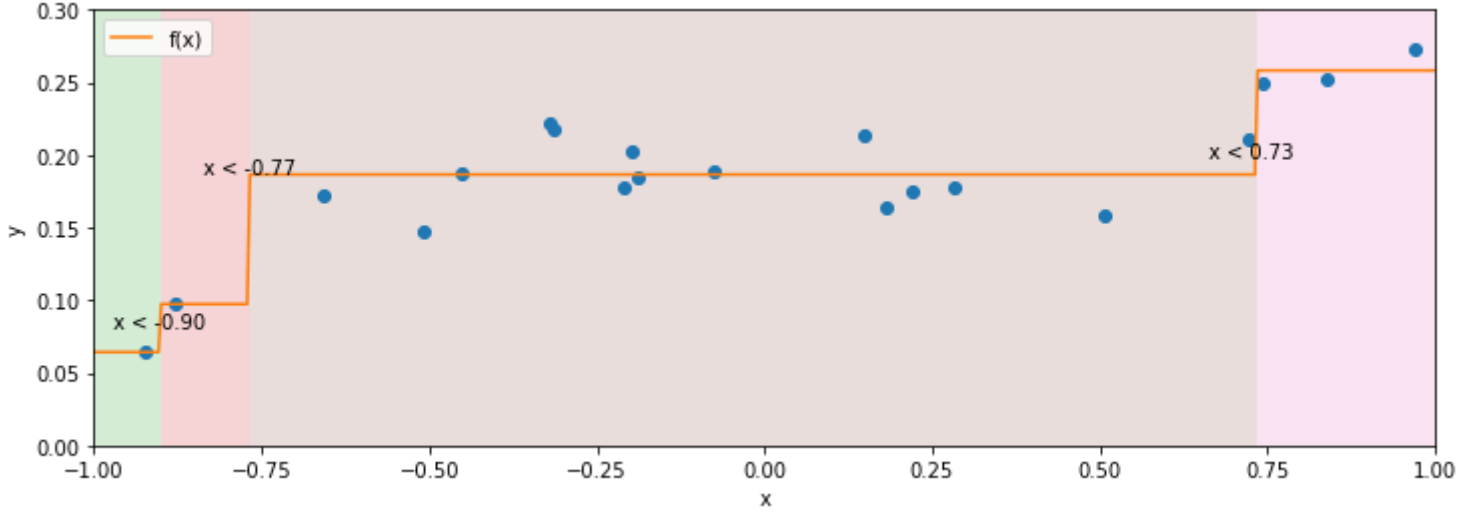
Depth 1

- One split
- Two leafs
- Two constant regions



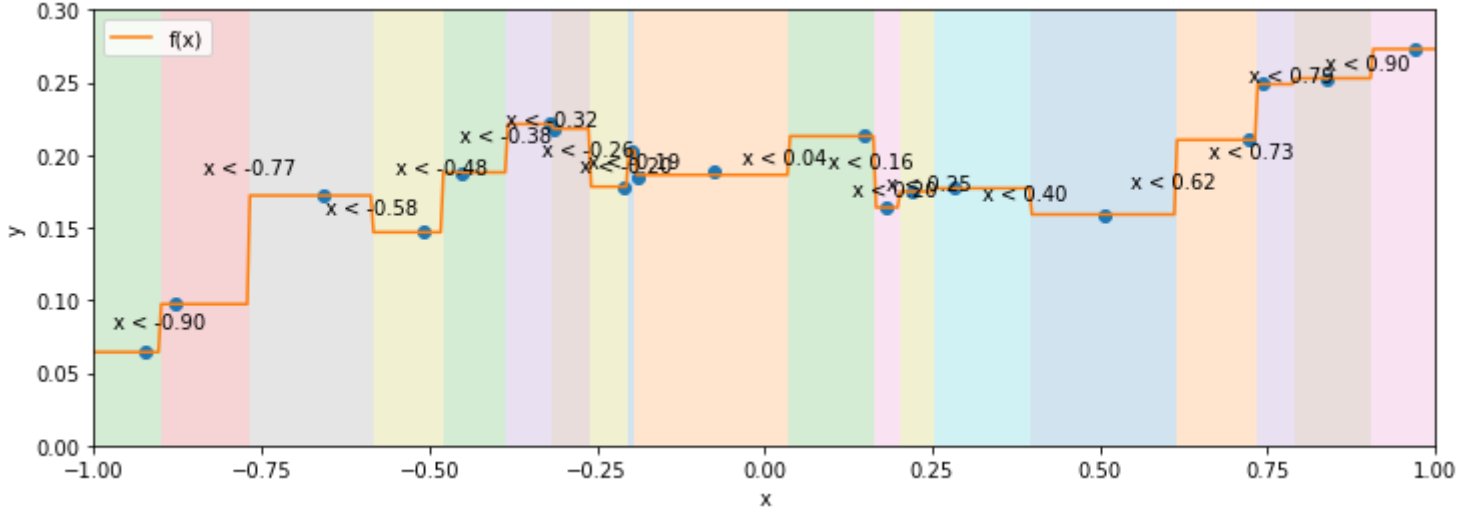
Depth 2

- Three splits
- Four leaves - four regions



Depth 8

- Etc.
- Unique mappings:
    - point to value (because function)
    - point to leaf (remember for later)
- So given a function represented by a tree $t(x)$, we can decompose it into two parts:
    - Map points to one of the L leaves: $q : \mathbb{R} \mapsto \{0, \ldots, L-1\}$
    - A vector $w \in \mathbb{R}^L$ of L weights: the constant values at the leaves
    - Then, $t(x) = w_{q(x)}$



Exercise 4

Challenges Learning Decision Trees

- Tree is discrete: no gradient-based training
    - optimization over discrete objects often result in combinatorial explosion
- A single tree is highly non-linear:
    - small changes in the data might dramatically affect the structure
- Techniques:
    - Discrete optimization: use greedy heuristic
    - High-variance of single tree: use an ensemble of many trees
- Simplest tree ensemble method: bagging/random forrest (also available in `cuml`)
    - Sub-sample the dataset and fit a different tree on each
    - Performs often worse than XGBoost

Learning Algorithm: XGBoost

- Goals:
  - high-level understand how the algorithm works
  - gain familiarity with notation and terminology to understand the API
- A few crucial details omitted, like regularization. But easy to add them, once the algorithm is clear
  - More details the excellent official intro: https://xgboost.readthedocs.io/en/latest/tutorials/model.html (https://xgboost.readthedocs.io/en/latest/tutorials/model.html)
- Notation:
  - $f_\theta(x)$: the function, represented by an ensemble of trees
  - $t_\theta(x)$: a single binary tree
    - L: the number of *leaves* in the tree
    - w: a L-vector, containing the *values* of the piece-wise constant regions
    - $q : \mathcal{X} \mapsto 1, \ldots, L$: mapping of input (e.g. training) points to leaves
    - $\mathcal{L}_n := \{i \mid q(x_i) = n\}$: set of training points assigned to leaf node n

Exercise 5

Learning Algorithm: XGBoost

- Boosting: stage-wise (greedy!) additive model of T trees defines the function f_θ as

$$f_\theta^{(T)}(x) = \sum_{k=1}^{T} t_\theta^{(k)}(x)$$

  - or recursively, which reflects, how we will learn the ensemble

$$f_\theta^{(T)}(x) = f_\theta^{(T-1)}(x) + t_\theta^{(T)}(x)$$

- The algorithm progresses sequentially, performing T rounds
- Implication: $t^{(k)}$ should not be trained on the original labels y but the *residual* $y - f^{(k)}$ (in regression)
  - the k-th tree tries to "fix" the errors, the model at the previous round made
  - But: *without* revisiting their parameters (greedy)
- Need two ingredients, that are related
  - Criterion to optimize to set the function values
  - Scoring structures to decide where to make a split

XGBoost: Loss Function

- Similar to GLMs, the trees in XGBoost are functions into the Reals, hence we can use the same objective functions/likelihood terms (but we will see again, that the details can be abstracted away as well)
    - regression: $l_{sq}(y, f_\theta(x)) = \frac{1}{2}(y - f_\theta(x))^2$
    - classification: $l_\sigma(y, f_\theta(x)) = \log \sigma(y f_\theta(x))$
- The optimization problem at each stage k is to add a tree such that the error is reduced, i.e.

$$E(t_\theta^{(k)}) = \sum_{i=1}^{N} l(y_i, f_\theta(x_i)^{(k-1)} + t_\theta^{(k)}(x_i))$$

XGBoost Loss: Quadratic Approximation

- Instead of considering E directly, the first approximation the XGBoost method introduces, is to approximate this error with it's *second order Taylor expansion*, i.e. with a quadratic function (which is of course exact for the squared loss)
- We treat the existing model as the point and the new tree as the perturbation, with g_i, h_i the first and second derivatives of the loss

$$E(t_\theta^{(k)}) = \sum_{i=1}^{N} l(y_i, f_\theta(x_i)^{(k-1)} + t_\theta^{(k)}(x_i))$$

$$\sum_{i=1}^{N} l(y_i, f_i + t_i) \approx \sum_{i=1}^{N} l(y_i, f_i) + g_i t_i + \frac{1}{2} h_i t_i^2 \doteq \sum_{i=1}^{N} g_i t_i + \frac{1}{2} h_i t_i^2$$

- Just like GLMs! Again, from the loss, we only need derivatives to drive the algorithm!
  - $g = \frac{\partial}{\partial f} l(y, f), g_i = g(y_i, f(x_i))$
  - $h = \frac{\partial^2}{\partial f^2} l(y, f), h_i = g(y_i, f(x_i))$
- Of course, for $l_{sq}(y, f)$ this is exact and $g = (f - y), h = 1$

XGBoost: Making w explicit

- Remember our view on trees:
  - A vector $w \in \mathbb{R}^L$ of L weights: the constant values at the leaves
  - Then, $t(x) = w_{q(x)}$
- Given a tree, i.e. w and q, we can find all training samples in a leaf node n
  - We can find the set $\mathcal{L}_n = \{i \mid q(x_i) = n\}$
  - Note: $t(x_i) = w_n$ for all x_i s.t. $i \in \mathcal{L}_n$
- Now for any given tree, with L the number of leaves , we can re-write $E(t) = \sum_{i=1}^{N} g_i t_i + \frac{1}{2} h_i t_i^2$ in terms of w and $\mathcal{L}$:

$$E(t) = \sum_{i=1}^{N} g_i t_i + \frac{1}{2} h_i t_i^2 = \sum_{n=1}^{L} (\sum_{i \in \mathcal{L}_n} g_i) w_n + \frac{1}{2} (\sum_{i \in \mathcal{L}_n} h_i) w_n^2$$

$$= \sum_{n=1}^{L} G_n w_n + \frac{1}{2} H_n w_n^2$$

- Nicely decomposes over the parameters of the tree

XGBoost: Summary

- Find the minimizer w^* of $E(t) = \sum_{n=1}^{L} G_n w_n + \frac{1}{2} H_n w_n^2$ analytically
  - Assumption: the loss is twice differentiable, (strongly) convex: $h > 0$

$$w_n^* = -\frac{G_n}{H_n} \qquad E(w^* \mid q) = -\frac{1}{2} \sum_{n=1}^{L} \frac{G_n^2}{H_n}$$

- We still assume that we are given some particular tree *structure*, i.e. the point to leaf mapping q
- Now: contribution of leaf n to the loss: $\propto \frac{G_n^2}{H_n} = \frac{\left(\sum_{i \in \mathcal{L}_n} g_i\right)^2}{\sum_{i \in \mathcal{L}_n} h_i}$ - is it worth breaking up the points in leaf n, i.e. $\mathcal{L}_n$ into two? What's the "gain"?
- Gain of new split into n_{left}, n_{right}: "contrib. $\mathcal{L}_{n_{left}}$ + contrib. $\mathcal{L}_{n_{right}}$ - contrib. $\mathcal{L}_n$"

$$\text{Gain} = \frac{G_{n_{left}}^2}{H_{n_{left}}} + \frac{G_{n_{right}}^2}{H_{n_{right}}} - \frac{G_n^2}{H_n}$$

XGBoost: Greedy Tree Building Algorithm

- Given a set of N training points:
    - for each feature:
        - sort the points by feature value
        - for each possible $N-1$ split positions find the split with the best gain
    - partition the points according to the split
    - recurse left of the split and right of the split

Tree Building Algorithm in 1D

- Does not take into account searching over features

```
def gain(g,h,it):
    Gl, Gr = sum(g[:it+1]), sum(g[it+1:])
    Hl, Hr = sum(h[:it+1]), sum(h[it+1:])
    return  Gl**2 / Hl # contrib. left
          + Gr**2 / Hr # contrib. right
          - (Gl+Gr)**2 / (Hl+Hr) # cost of removing current leaf

def build1d(x, g, h, d, max_depth):
    n = Node()
    n.value = -sum(g) / sum(h) # determine current function value E, i.e. the constant w
    # check stopping criterion:
    if d == max_depth or len(x) == 1:
        return n
    # evaluate splits and maximize gain
    max_score, split = max([(gain(g,h,it) , it) for it in range(len(x) - 1)])
    n.split = 0.5*(x[split] + x[split+1]) # split in the middle
    # divide points and recurse on the split
    n.left = build1d(x[:split+1], g[:split+1], h[:split+1], d+1, max_depth)
    n.right = build1d(x[split+1:], g[split+1:], h[split+1:], d+1, max_depth)
    return n
```

Toy XGBoost Algorithm for Regression

```
def xgboost1d(xs, ys, rounds, depth): #single input feature is already sorted
    forrest = []                      #we're growing a forrest
    yprev = np.zeros_like(ys)         #initial predictions are constant 0
    h = np.ones(len(ys))              #for squared loss, h is constant 1
    for r in range(rounds):
        g = yprev - ys                #update g, the residual
        root = build1d(xs, g, h, 0, depth) #tree
        yprev += np.array([eval1d(root, x_) for x_ in xs]) #update the prediction of the whole model
        forrest.append(root)          #add to the forrest
    return forrest

def pred1d(forrest, x):
    pred = np.zeros_like(x.flatten())
    for tree in forrest:              #evaluate contribution from each tree and sum up
        pred += np.array([eval1d(tree, x_) for x_ in x])
    return pred
```

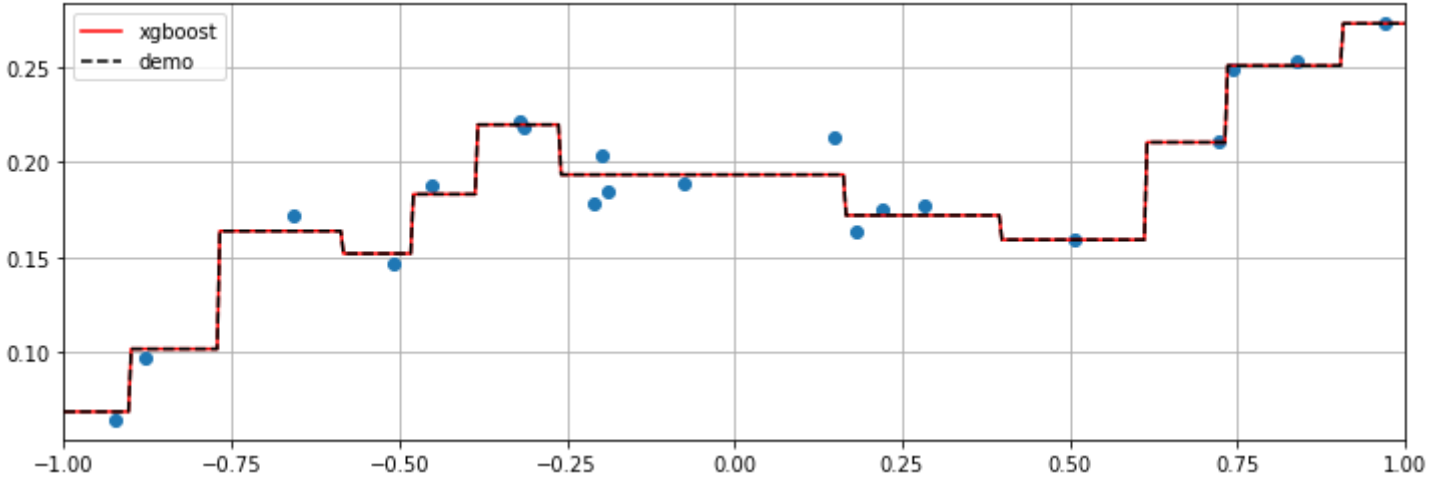
XGBoost API Example

```
import xgboost as xgb

dtrain = xgb.DMatrix(x, y)
dtest = xgb.DMatrix(xtest)
param = {'max_depth':3,                          #depth of the tree
         'eta':1,                                #shrinkage: fractional update
         'objective':'reg:squarederror',         #regression: loss function
         'reg_lambda':0}                         #l2 regularization parameter

num_round = 2                                    #number of rounds: trees in the forrest

bst = xgb.train(param, dtrain, num_round)
xgb_pred = bst.predict(dtest)
```



Exercise 6 (30 min)

Summary

- Discussed trees as a function class
- Saw the same techniques as in GLMs to model data types
- Worked through the XGBoost derivation
    - Greedy approximations
    - Taylor approximation
    - Loss-based splitting criterion (structure score)
- Looked at a toy implementation to clarify the concepts
- Important, but missing here: regularization
    - L2: penalize magnitude of weights
    - Prefer shallower trees: subtract penalty from gain of a split
    - Shrinking ("learning rate"): scale down contributions of individual trees added in each round

Outline

- The RAPIDS data science stack and the case for "Classical" ML
- Supervised learning
    - Recap of Fundamental Concepts
    - Generalized linear models (GLMs)
    - Gradient-boosted decision trees, XGBoost

Conclusion

- We discussed some of the concepts behind the high-level APIs of cuML
- Many more libraries, actively developed
- If performance is an issue in your workflow, perhaps something to consider