Optimization for Data Science
Lecture 11:
Practical Aspects of Stochastic (Sub-)Gradient, Advanced Stochastic Methods, Fundamentals of Learning

Kimon Fountoulakis

School of Computer Science
University of Waterloo

22/10/2019

Outline of this lecture

- Recap: ML loss functions, stochastic (sub-)gradient, iteration complexity
- Practical version of stochastic (sub-)gradient
- Advanced stochastic gradient methods
- Fundamentals of learning

Disclaimer

- Most of these slides have been adapted from slides of M. Schmidt, author of some of the methods that we will talk about today, and Ryan Tibshirani.
- Personally, I do not have a lot of practical experience with advanced stochastic methods since I have never done research on them.

Data

- We are given n data points (a_i, b_i) $i = 1, \ldots, n$.
- $a_i \in \mathbb{R}^d$ is a sample/vector of length d, i.e., for each sample we have d features.
- b_i are the given labels. It can be real or binary or more generally an integer.

Finite-sum optimization problems

- We consider the following optimization problem

$$\min_{x\in\mathbb{R}^n} \underbrace{\frac{1}{n}\sum_{i=1}^{n} f_i(x)}_{f(x)} = \mathbb{E}[f_i(x)] = f(x)$$

Stochastic Gradient Method

- Assuming that each f_i is differentiable then stochastic gradient is equivalent to:
- Pick randomly a sample i
- $x_{k+1} := x_k - \alpha_k \nabla f_i(x)$
- iteration cost is now independent of n, but how many iterations?

Iteration complexity for smooth functions

	Gradient Descent	**Accelerated Gradient**	**Stochastic Gradient**
Non-convex	$\mathcal{O}\left(\frac{L}{\epsilon}\right)$	-	$\mathcal{O}\left(e^{\frac{LB^2}{\epsilon}}\right)$
Convex	$\mathcal{O}\left(\frac{L}{\epsilon}\right)$	$\mathcal{O}\left(\sqrt{\frac{L}{\epsilon}}\right)$	$\mathcal{O}\left(e^{\frac{B^2}{\epsilon}}\right)$
Strongly convex	$\mathcal{O}\left(\frac{L}{\delta}\log\frac{1}{\epsilon}\right)$	$\mathcal{O}\left(\sqrt{\frac{L}{\delta}}\log\frac{1}{\epsilon}\right)$	$\mathcal{O}\left(\frac{GB^2}{\delta^2}\frac{1}{\epsilon}\right)$

Why would we consider stochastic methods?

- For two reasons
  - Extremely cheap iteration cost. Very good when we have millions (or more) data points and each data point is low-dimensional. This is good when we do not want to access all data points at each iteration.
  - Fast convergence to low accuracy. For most AI/ML problems a low accuracy solution to the optimization problem is good enough to obtain high accuracy w.r.t domain metrics of measuring performance, e.g., precision/recall.

Preferred step-sizes

- Practitioners prefer to use constant step-sizes for two reasons:

  - Fast convergence to low accuracy solutions. Remember the rate is $\mathcal{O}(1/t + \alpha)$ for non-convex and convex functions, and $\mathcal{O}((1-(\delta/L))^t + \alpha)$ for δ -strongly convex.

  - For most AI/ML problems a low accuracy solution to the optimization problem is good enough to obtain high accuracy w.r.t domain metrics of measuring performance, e.g., precision/recall.

Termination Criterion of Stochastic Gradient

- Usually we terminate an algorithm if the norm of the gradient is small.

- However, in this algorithm we only access one sample per iteration. This means that we cannot compute the norm of the whole gradient.

- If we do compute the norm of the whole gradient at each iteration, then this defeats the purpose of stochastic gradient.

Some options for terminating stochastic gradient

- Predefined maximum number of iterations
- Predefined upper bound for the running time
- Stop when the norm of the gradient of the chosen sample is small.
- Measure the norm of the whole gradient every n iterations, if it is small then terminate.
- Use your validation data and terminate the algorithm when precision/recall (or your preferred metric) are large enough.
- Measure validation error every n iterations. Stop if the validation error starts overfitting

Stochastic sub-gradient

- We consider the following optimization problem

$$\min_{x \in \mathbb{R}^n} \underbrace{\frac{1}{n}\sum_{i=1}^{n} f_i(x)}_{f(x)} = \mathbb{E}[f_i(x)] = f(x)$$

- where we assume that f_i are non-smooth and f is convex.

- The **stochastic sub-gradient method is**:
  - Pick randomly a sample i
  - $x_{k+1} := x_k - \alpha_k g_i(x)$, where $g_i(x) \in \partial f_i(x)$

Iteration complexity: non-smooth functions

	Gradient Descent	Accelerated Gradient	Stochastics Sub-Gradient	
Non-convex	$\mathcal{O}\left(\frac{D}{\epsilon^2}\right)$	-	$\mathcal{O}\left(\frac{1}{\epsilon^4}\right)$	**Just appeared 2018**
Convex	$\mathcal{O}\left(\frac{D}{\epsilon^2}\right)$	$\mathcal{O}\left(\frac{\sqrt{D}}{\epsilon}\right)$	$\mathcal{O}\left(e^{\frac{\sigma^2}{\epsilon}}\right)$	
Strongly convex	$\mathcal{O}\left(\frac{D}{\delta\epsilon}\log\frac{1}{\epsilon}\right)$	$\mathcal{O}\left(\sqrt{\frac{D}{\delta\epsilon}}\log\frac{1}{\epsilon}\right)$	$\mathcal{O}\left(\frac{G\sigma^2}{\delta^2}\frac{1}{\epsilon}\right)$	

Comments on iteration complexity

- For convex and strongly-convex functions, gradient descent and stochastic sub-gradient have similar complexity, but each iteration of stochastic sub-gradient is n times less expensive!!

- The result for non-convex functions just appeared in 2018 (Damek and Drusvyatskiy 2018). The result is much worse than the one of gradient descent, but practitioners still prefer stochastic sub-gradient.

Stochastic (Sub-)Gradient with Sparse Features

- Consider minimizing the hinge-loss objective (good for binary classification)
  minimize $\frac{1}{n}\sum_{i=1}^{n} \max\{0,1 - b_i(x^T a_i)\}$

- when d in $\alpha_i \in \mathbb{R}^d$ is huge but each a_i has at most z non-zero entries.

Stochastic (Sub-)Gradient with Sparse Features

- Stochastic sub-gradient for a randomly chosen sample requires computing the sub-gradient:
- $g_i(x) = \begin{cases} -b_i a_i & \text{if } 1 - b_i(x^T a_i) > 0 \\ 0 & \text{otherwise} \end{cases}$
- Then we perform $x_{k+1} = x_k - \alpha_k g_i(x_k)$
- Since each a_i has at most z non-zeros, then computing x_{k+1} should take $\mathcal{O}(z)$ FLOPS.

Sparse data

- For many datasets the feature vectors $a_i \in \mathbb{R}^d$ are **very sparse**.
- This means that out of d entries in $a_i \in \mathbb{R}^d$, at most z entries are non-zero.
- If $d \gg z$, then computing the (sub-)gradient for a chosen sample and computing x_{k+1} should cost, ideally, $\mathcal{O}(z)$ FLOPS instead of $\mathcal{O}(d)$.

Sparse data

- To get $\mathcal{O}(z)$ cost per iteration instead of $\mathcal{O}(d)$ we have to store any vectors (say g) that the algorithm uses using a sparse data structure.

- Consider a vector $g = [0\ 0\ 0\ 0.1\ 0.4\ 0\ \ \dots\ \ 3\ 0]$, which has z non-zero entries out of d, with $z \ll d$.

- We can store g using $\mathcal{O}(z)$ memory instead of $\mathcal{O}(d)$.

Sparse data

- We can store g using $\mathcal{O}(z)$ memory instead of $\mathcal{O}(d)$.
- Store only the non-zero entries and their indices.
- We need two vectors of length z
  - $g^{values} = [0.1 \ 0.4 \ 3]$
  - $g^{point} = [4 \ 5 \ d-1]$

Sparse data

- Using this compressed representation of vectors we can do standard operations in $\mathcal{O}(z)$ FLOPS.
- Scalar multiplication: αg in $\mathcal{O}(z)$ FLOPS by doing $g^{values} := \alpha g^{values}$
- Inner product: $w^T g$ in $\mathcal{O}(z)$ FLOPS by multiplying g^{values} by w only at positions g^{point}.

Lazy sparse updates for L2-regularized finite-sum problems

- Consider minimizing the hinge-loss objective

  minimize $\lambda\|x\|_2^2 + \frac{1}{n}\sum_{i=1}^{n} \max\{0, 1 - b_i(x^T a_i)\}$

- when d in $\alpha_i \in \mathbb{R}^d$ is huge but each a_i has at most z non-zero entries.

Lazy sparse updates for L2-regularized finite-sum problems

- Stochastic sub-gradient for a randomly chosen sample requires computing the sub-gradient:
- $g_i(x) = \begin{cases} -b_i a_i & \text{if } 1 - b_i(x^T a_i) > 0 \\ 0 & \text{otherwise} \end{cases}$
- Then we perform $x_{k+1} = x_k - \alpha_k g_i(x_k) - \lambda \alpha_k x_k$
- Because x_k might be dense then computing x_{k+1} might cost $\mathcal{O}(d)$ FLOPS instead of $\mathcal{O}(z)$ (number of non-zeros in $g_i(x_k)$).

Lazy sparse updates for L2-regularized finite-sum problems

- However, because the data samples are extremely sparse, i.e., $d \gg z$, then coordinates in x_k might not get updated often.

- For example, assume that coordinate j in x_k has not been updated for the last 20 iterations. Then

$$\begin{aligned}
[x_k]_j &= [x_{k-1}]_j - 0 - \lambda\alpha_k[x_{k-1}]_j \\
&= (1 - \lambda\alpha_k)[x_{k-1}]_j \\
&\vdots \\
&= (1 - \lambda\alpha_k)[x_{k-20}]_j
\end{aligned}$$

Lazy sparse updates for L2-regularized finite-sum problems

- This means that we can perform 20 iterations for coordinate j in $\mathcal{O}(1)$ FLOPS.

- This trick requires keeping track of the last time each coordinate in x_k was updated.

- It get be generalized to other regularizers as well, but we will work on this later in this course.

Hybrid Deterministic-Stochastic Gradient

- Instead of using only one data point per iteration we can you many, but less than n.

- Deterministic method uses all n gradients at each iteration
$$x_{k+1} = x_k - \alpha_k \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(x_k).$$

- All stochastic methods that we have seen so far use only one sample per iteration $x_{k+1} = x_k - \alpha_k \nabla f_i(x_k)$.

Hybrid Deterministic-Stochastic Gradient

- A common variant is to use **mini-batching**, a set of samples $\mathscr{B}_k$ (indices of chosen samples) and perform the following update $x_{k+1} = x_k - \alpha_k \frac{1}{|\mathscr{B}_k|} \sum_{i \in \mathscr{B}_k} \nabla f_i(x_k)$.

- Note that this approach can be easily parallelized. For example, if we have 16 cores, then we can choose 16 samples ($\mathscr{B}_k = 16$), where each gradient for each sample can be computed in parallel.

Convergence rate of Hybrid-Deterministic Stochastic Gradient

- Let's view mini-batching as a gradient method with error

$$\frac{1}{|\mathcal{B}_k|}\sum_{i\in\mathcal{B}_k}\nabla f_i(x_k) = \nabla f(x_k) + e_k$$

- where e_k is the error between mini-batching and full-gradient.

Convergence rate of Hybrid Deterministic-Stochastic Gradient

- Assuming that our problem has Lipschitz continuous gradient with constant L then using the FToC and constant step-sizes $\alpha_k = 1/L$ we get

$$f(x_{k+1}) \leq f(x_k) \quad \underbrace{-\frac{1}{L}\|\nabla f(x_k)\|_2^2}_{\text{negative, thus, good}} \quad + \quad \underbrace{\frac{1}{2L}\|e_k\|_2^2}_{\text{positive, thus, bad}}$$

Convergence rate of Hybrid Deterministic-Stochastic Gradient

- If $e_k = 0$, then for convex and non-convex functions the rate is $\mathcal{O}(1/k)$. But now we have $\mathcal{O}(1/k + \|e_k\|_2^2)$. To guarantee rate $\mathcal{O}(1/k)$ we need to have $\|e_k\|_2^2 = \mathcal{O}(1/k)$.

- If $e_k = 0$, then for δ-strongly convex functions the rate is $\mathcal{O}((1-\delta/L)^k)$. But now we have $\mathcal{O}((1-\delta/L)^k + \|e_k\|_2^2)$. To guarantee rate $\mathcal{O}((1-\delta/L)^k)$ we need to have $\|e_k\|_2^2 = \mathcal{O}((1-\delta/L)^k)$.

Effect of Batch Size on Error

- It is obvious that if we want to control the rate of the hybrid method, then we need to control the error $\|e_k\|_2^2$.

- The main way that we have to control the error $\|e_k\|_2^2$ is through the size of the batch-size $|\mathscr{B}_k|$.

Effect of Batch Size on Error

- First we need to observe that $e_k = \nabla f(x_k) - \frac{1}{|\mathscr{B}_k|} \sum_{i \in \mathscr{B}_k} \nabla f_i(x_k)$

- Taking expectation of the norm of the error we get

$$\mathbb{E}\left[\|e_k\|_2^2\right] = \mathbb{E}\left[\left\| \nabla f(x_k) - \frac{1}{|\mathscr{B}_k|} \sum_{i \in \mathscr{B}_k} \nabla f_i(x_k) \right\|_2^2\right]$$

- This is the definition of variance! If we can control variance (make it smaller), then we can obtain faster methods.

Effect of Batch Size on Error

- Let's assume that the sample size $|\mathscr{B}_k|$ at the beginning of the algorithm.
- If we sample with replacement then $\mathbb{E}\left[\|e_k\|_2^2\right] = \dfrac{\sigma^2}{|\mathscr{B}_k|}$, i.e., the expected norm of the error is equal to the scaled variance.
- This means that the smaller the larger size $|\mathscr{B}_k|$ is the smaller the error at each iteration.
- It also means that if at each iteration we double the sample size then there error gets halved.

Effect of Batch Size on Error

- Let's assume that the sample size $|\mathscr{B}_k|$ at the beginning of the algorithm.
- If we sample without replacement then $\mathbb{E}\left[\|e_k\|_2^2\right] = \dfrac{n - |\mathscr{B}_k|}{n} \dfrac{1}{|\mathscr{B}_k|} \sigma^2$, i.e., the expected norm of the error is equal to the scaled variance.
- This means that the error goes to zero as $|\mathscr{B}_k| \to n$.

Effect of Batch Size on Error

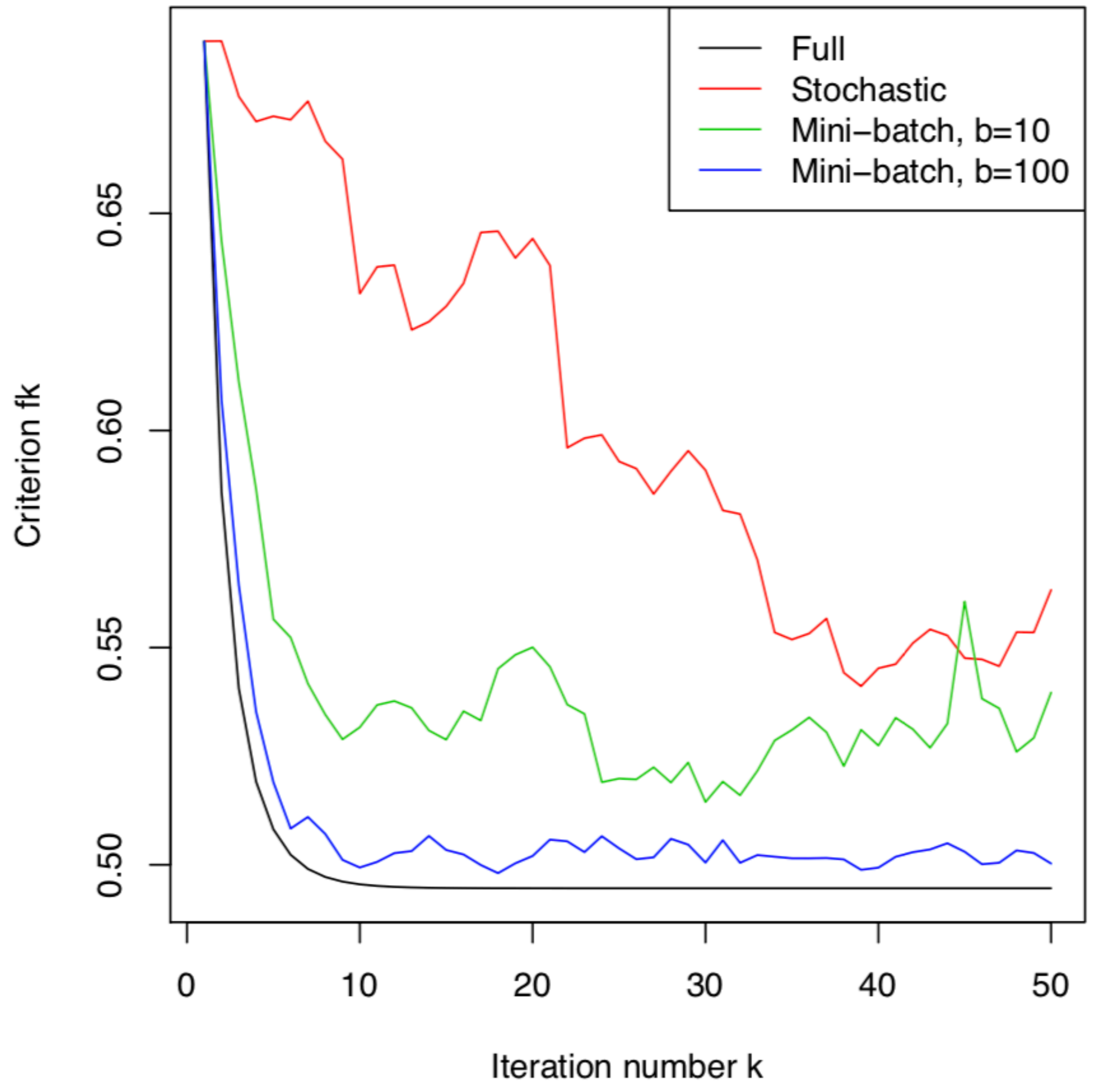
- To obtain sub-linear rate for convex functions we have to do $|\mathscr{B}_{k+1}| = |\mathscr{B}_k|$+const.
- To obtain linear rate for strongly convex functions we have to do $|\mathscr{B}_{k+1}| = |\mathscr{B}_k|/(1 - \delta/L)$.

Batching: growing-batch-size methods

- For fixed sample-size $|\mathscr{B}_k|$, the rate of stochastic gradient is sub-linear.
- For growing-batch-size methods achieve faster rate:
  - Initially iterations are inexpensive.
  - Later iterations converge to full-gradient

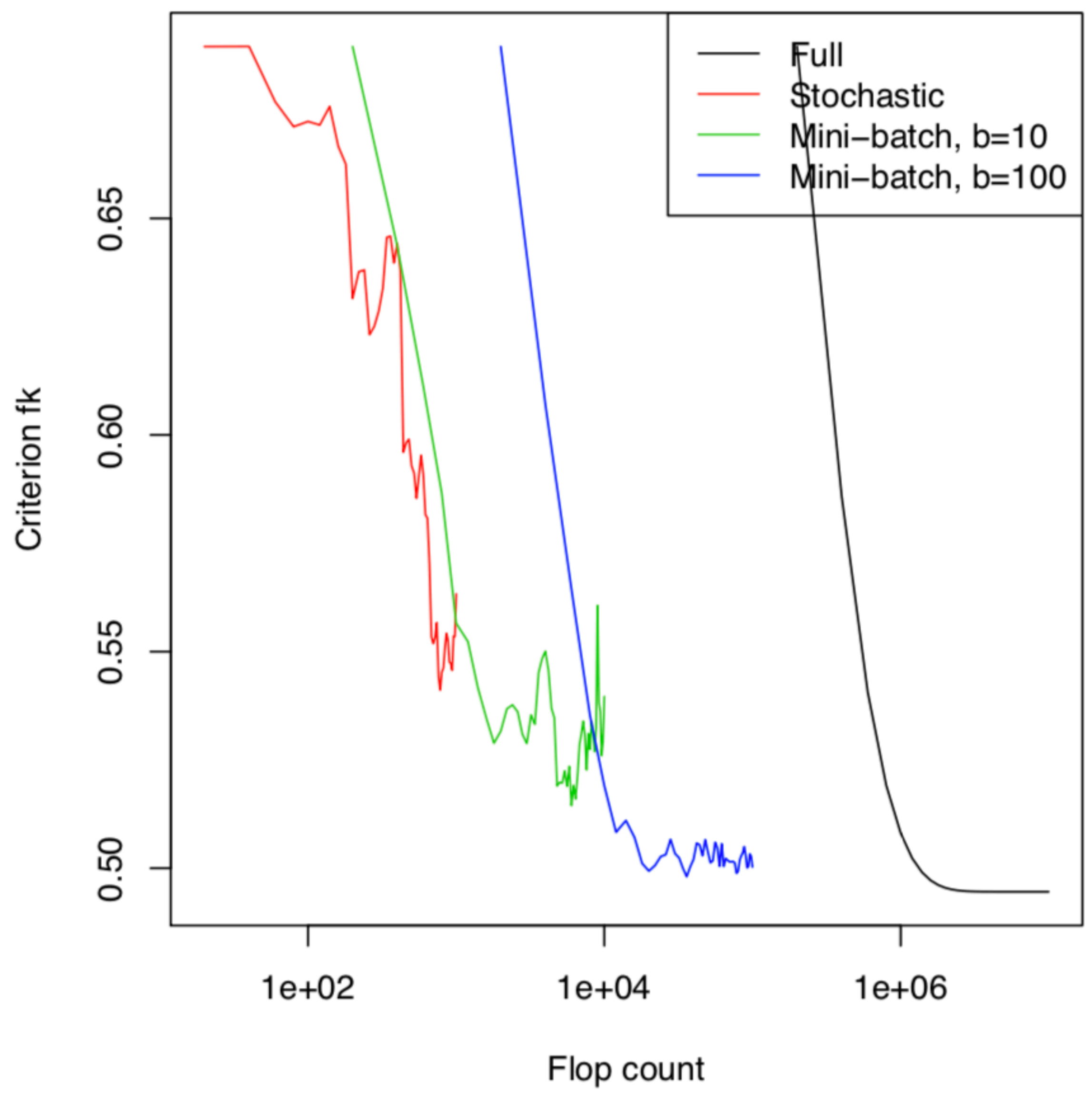
Practical performance

- $n = 10000$, $d = 20$, all methods use fixed step-sizes.



Practical performance

- $n = 10000$, $d = 20$, all methods use fixed step-sizes.



Variance

- In earlier slides we defined variance as $\mathbb{E}\left[\|e_k\|_2^2\right] = \mathbb{E}\left[\left\| \nabla f(x_k) - \nabla f_i(x_k) \right\|_2^2\right]$
- For stochastic (sub-)gradient methods we simply assumed that this term is bounded.
- We will now discuss methods that drag variance to zero and this results in faster algorithms (in worst-case).

Variance

- Fix the batch size.
- Loosely: because of non-decreasing variance, we have to decay the step size α_k to zero, which in turn means we can't take "large" steps, and hence the convergence rate is slow.
- Question: can we decrease variance to zero and maintain large step-sizes at the same time?

Stochastic Average Gradient

- For growing-batch-size methods the cost per iteration will depend eventually on all samples.
- Ideally, we would like a stochastic method that only requires computing one gradient for a chosen a sample and has as good convergence rate as full-gradient method.
- The first time that this was achieved was in 2012, by the algorithm: **stochastic average gradient (SAG)**.

Stochastic Average Gradient

- To motivate SAG let's view gradient descent as performing the following
$$x_{k+1} = x_k - \alpha_k \frac{1}{n} \sum_{i=1}^{n} v_i^k$$
- where at each iteration we set $v_i^k = \nabla f_i(x_k)$.
- For SAG at iteration k picks randomly a sample i_k and it sets $v_{i_k}^k = \nabla f_{i_k}(x_k)$.
- All other v_i^k are kept at their previous value.
- Unlike batching, we use gradient for every sample. However, some of the gradients might be out of date (computed at previous iterations).

Efficient Updates

- The cost of one SAG iteration is basically just as efficient as simple stochastic (sub-)gradient, as long as you perform the updates in a clever way:

$$x_{k+1} = x_k - \alpha_k \frac{1}{n} \left(\underbrace{g_{i_k}(x_k) - g_{i_k}(x_{k-1})}_{\text{replace previous with new}} + \underbrace{\sum_{i=1}^{n} g_i(x_{k-1})}_{\text{old part of the gradient}} \right)$$

Convergence Rate of Stochastic Average Gradient

- Assume that f is convex, f_i's are differentiable and each $\nabla f_i(x)$ is Lipschitz continuous with constant L (note that this is stronger than saying that the full-gradient is Lipschitz continuous).

- Denote $\bar{x}_k = \frac{1}{k}\sum_{l=0}^{k-1} x_l$

- Initialize the algorithm by $g_i(x_0) := \nabla f_i(x_0) - \nabla f(x_0)\ \forall i = 1,\ldots,n$

- We get $\mathbb{E}\left[f(\bar{x}_k)\right] - f^* \leq \frac{48\left(f(x_0) - f^*\right)}{k} + \frac{128L\|x_0 - x^*\|_2^2}{k}$

Convergence Rate of Stochastic Average Gradient

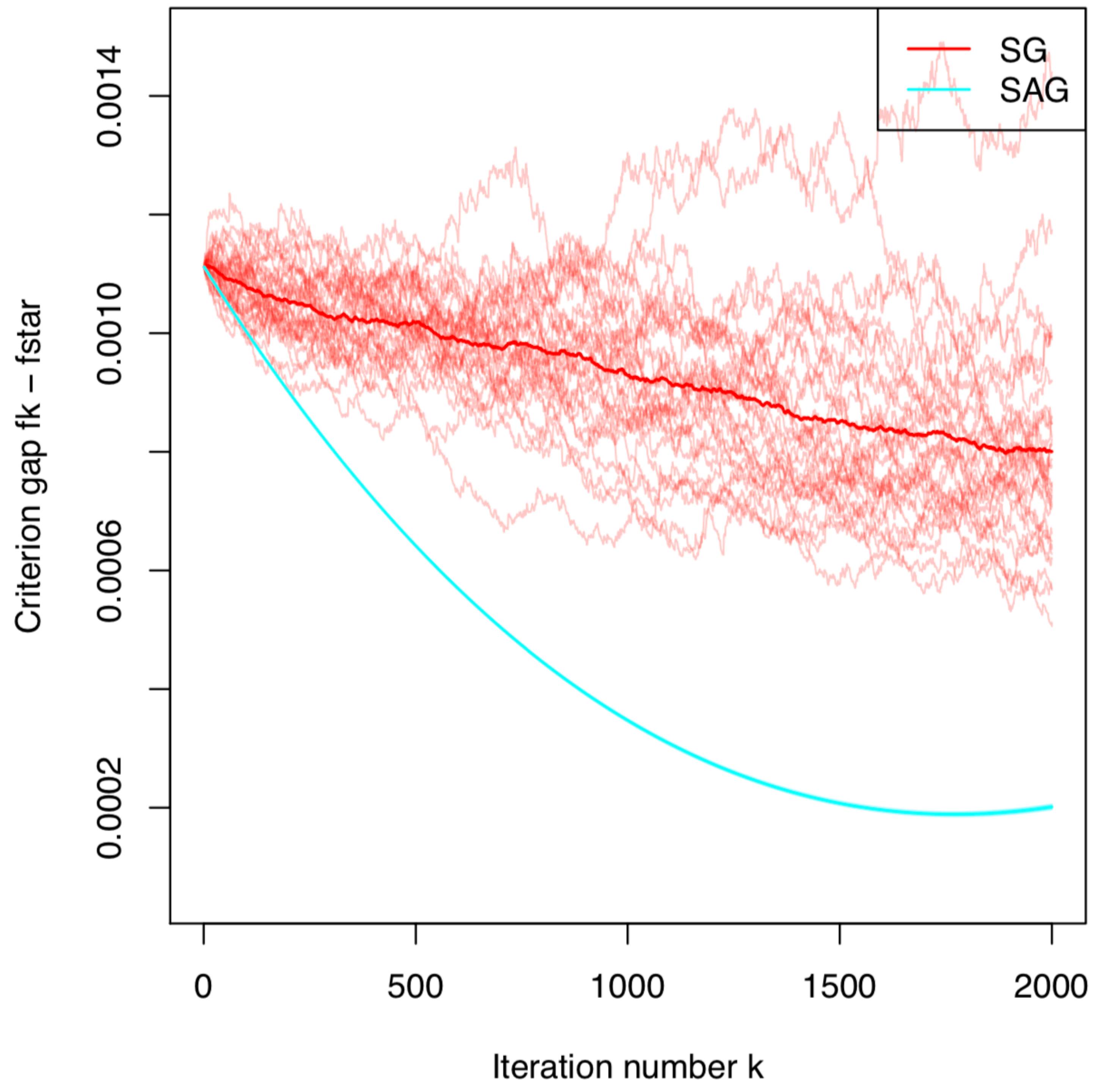
- Convergence rate is stated for $\bar{x}_k$, but a similar result can be shown for x_k^{best}; the best x_k that has been observed so far.
- The rate of SAG is $\mathcal{O}(1/k)$, which is an improvement over stochastic gradient $\mathcal{O}(1/k + \alpha)$ and similar to full-gradient $\mathcal{O}(1/k)$.
- But the constants are different!
- SAG: $\dfrac{48\left(f(x_0) - f^*\right) n}{k} + \dfrac{128L\|x_0 - x^*\|_2^2}{k}$
- Full-grad.: $\dfrac{L\|x_0 - x^*\|_2^2}{2k}$
- Stochastic. grad.: $\dfrac{L\|x_0 - x^*\|_2^2}{2k} + \dfrac{\sigma^2}{L}$
- SAG seems to need n times more iterations compared to full-grad. and stoc. grad. The authors of SAG suggest to find x_0 using the result of n steps of stochastic gradient.

Convergence Rate of Stochastic Average Gradient

- If each gradient $\nabla f_i(x)$ is Lipschitz continuous with constant L
- And the objective function f is δ-strongly convex
- And $\alpha_k = 1/(16L)$
- Then SAG satisfies

$$\mathbb{E}\left[f(x_k) - f^*\right] = \mathcal{O}\left(\left(1 - \min\left\{\frac{\delta}{16L}, \frac{1}{8n}\right\}\right)^k\right)$$

Practical performance



Multiple lines show different trials, 30 re-runs in total. The solid lines demonstrate average behaviour.

Practical performance

- SAG seems to perform well, but practitioners claim that it does not work out of the box. It requires a lot of parameter tuning, which we might not know a-priori.

- The previous plot shows behavior when x_0 is set as the output of n iterations of stochastic gradient.

- Results for stochastic gradient are reported after the n-th iteration, so we can have a fair comparison.

- Initialization of SAG is $g_i(x_0) := \nabla f_i(x_0) - \nabla f(x_0)\ \forall i = 1,\ldots,n$

- The results in the previous correspond to highly tuned step-sizes. In particular, the step-size was set as large as possible before SAG diverges.

Stochastic Variance Reduced Gradient

- Initialize with some $\tilde{x}_0$ and some step-size α
- For $k = 1, \ldots$
  - Set $\tilde{x} := \tilde{x}_{k-1}$
  - Compute $v := \nabla f(\tilde{x})$
  - Set $x_0 := \tilde{x}$
  - For $l = 1 \ldots n$
    - Pick sample i_l at random
    - $x_l = x_{l-1} - \alpha(\nabla f_{i_l}(x_{l-1}) - \nabla f_{i_l}(\tilde{x}) + v)$
  - Set $\tilde{x}_k := x_n$

Stochastic Variance Reduced Gradient

- Efficient updates can be performed by SVRG as well
- It can be shown to achieve variance reduction similar to SAG
- Convergence rates are similar to SAG.

AdaGrad

- $x_{k+1} = x_k - \alpha_k D^{-1} g_i(x_k)$
- where D is a diagonal matrix. Its j-th diagonal component is defined as $D_{jj} := \sqrt{\gamma + \sum_{l=0}^{k} \left(\nabla_j f_{i_l}(x_l) \right)^2}$.
- i_l is the index of the sampled data at the l-iteration of the algorithm.
- $\nabla_j f$ denotes the j-th partial derivative of some function f.

Heuristic Extensions of AdaGrad

- RMSprop: variant of AdaGrad where step-sizes do not go to zero (unpublished, first proposed by G. Hinton in his lecture notes)
- Adam: accelerated version of AdaGrad (https://arxiv.org/abs/1412.6980)

Fundamentals of Learning

Kimon Fountoulakis

School of Computer Science
University of Waterloo

22/10/2019

Supervised Learning

- Given data a_i and labels b_i for $i = 1, \ldots, n$
- our objective is to find the parameters x of a function $h(x; a_i)$
- such that $h(x; a_i) \approx b_i$, i.e., function h predicts label b_i.
- We find parameters x by minimizing the loss function $\frac{1}{n}\sum_{i=1}^{n} loss(f(a_i; x), b_i)$.
- Connection to previous notation: $loss(h(a_i; x), b_i) = f_i(x)$

Supervised Learning

- Examples of $h(x; a_i)$ are:
- Linear functions: $a_i^T x$ (we will mostly focus on linear functions at this stage)
- Non-linear functions: neural networks etc.

Supervised Learning

- Examples of loss functions are: $loss(h(x; a_i), b_i)$ are:
- **Squared error:** $\frac{1}{2}(a_i^T x - b_i)^2$
- **Absolute error:** $|a_i^T x - b_i|$, robust to outliers.
- **Hinge loss:** $\max\{0,1 - b_i a_i^T x\}$, better for binary labels b_i
- **Logistic loss:** $\log(1 + \exp(-b_i a_i^T x))$, better for binary labels b_i and also it's a smooth function.

Goal of Supervised Learning

- We are given **training data** where we know the labels:

A =

Egg	Milk	Fish	Wheat	Shellfish	Peanuts	…
0	0.7	0	0.3	0	0	
0.3	0.7	0	0.6	0	0.01	
0	0	0	0.8	0	0	
0.3	0.7	1.2	0	0.10	0.01	
0.3	0	1.2	0.3	0.10	0.01	

b =

Sick?
1
1
0
1
1

- But the goal is to do well on any possible testing data:

Egg	Milk	Fish	Wheat	Shellfish	Peanuts	…
0.5	0	1	0.6	2	1	
0	0.7	0	1	0	0	
3	1	0	0.5	0	0	

Sick?
?
?
?

Test Error

- Let $b^i_{model} := h(x^*; a_i)$ be the output label of the trained function h when data-point a_i is given as input.

- The **test error** is defined as the expected error $\mathbb{E}[\,|b^i_{model} - b^i_{true}|\,]$

- where expectation is taken over the distribution of the data-points a_i, i.e., we assume that the data-points a_i are random variables.

- Think $\mathbb{E}[\,|b^i_{model} - b^i_{true}|\,]$ are measuring the error using infinitely many data-points.

Test Set and Test Set Error

- We often approximate the **test error** by measuring the error on a **test set**.

- We approximate the **test error** by measuring the **test set error**: $\frac{1}{t}\sum_{i=1}^{t} |b^i_{model} - b^i_{test\ set}|$

- where we draw t data-points at random. We call the drawn data-points the **test set**.

Test Set Error vs Test Error

- Note that the **test error** is not equal to the **test set error**.
- The ideal goal would be low **test error** and not low **test set error**.
- The "golden rule" of machine learning:
  - You should not use the **test set** in any way that influences finding x^* (training function h).
  - Otherwise, the **test set error** is not an unbiased approximation to the **test error**.
  - If you do that, then you risk "overfitting" to the **test set**,

Typical Supervised Learning Steps

- Given data (A, b), where A is the data matrix, b is the vector of labels.
- A typical set of supervised learning steps are:
- Data splitting
  - Split the data into **training set** (A_{train}, b_{train}) and a **validation set** (A_{valid}, b_{valid})
  - Use the validation set to compute the **test set error** (the approximation to the test error).
- Parameter tuning
  - For each candidate value of the hyper-parameters
  - Find x^* using (A_{train}, b_{train})
  - Evaluate performance using (A_{valid}, b_{valid})
- Choose the model with the best performance on the validation set

Typical Supervised Learning Steps

- Note that because we use the validation to select the best hyper-parameters. This violates the golden rule of machine learning.
- This means that the model that we constructed might overfit the data.
- However, many practitioners do this and sometimes it seems to work.

Test Set Error $\approx$ Test Error?

- If E_{test} is the test error and E_{valid} is the error on the validation set (the test set error), then:
- $$E_{test} = \underbrace{E_{test} - E_{valid}}_{E_{approx}} + E_{valid}$$
- If E_{approx} is small, then $E_{valid} \approx E_{test}$
- However, we cannot measure E_{test} because we do not know the true distribution of the data.
- So how can we know that E_{approx} is small?

Bounding E_{approx}

- $\mathbb{P}(E_{approx} > \epsilon) \leq 2\exp(-2\epsilon^2 t)$
- The proof is non-trivial. It requires Hoeffding's inequality.
- This can be proved in the case that we have binary labels and iid data-points and loss is in $[0,1]$.
- The above inequality shows that as the number of data-points t increases, then the probability that E_{approx} is large goes to zero exponentially fast.
- This means that the bigger the validation set is, the better the approximation.

Bounding E_{approx}

- However, in the standard supervised learning pipeline that we mentioned earlier we select the best hyper-parameters using the validation set.

- This “breaks” the result in the previous slide.

- Let ζ be the number of hyper-parameter settings that we try. Then we get

$$\mathbb{P}(E_{approx} > \epsilon \text{ for any hyper-parameter}) \leq 2\zeta \exp(-2\epsilon^2 t)$$

Bounding E_{approx}

- The bound $\mathbb{P}(E_{approx} > \epsilon$ for any hyper-parameter$) \leq 2\zeta \exp(-2\epsilon^2 t)$

- means that it is ok to optimize over various parameter settings, as long as t is large.

Bounding E_{approx}

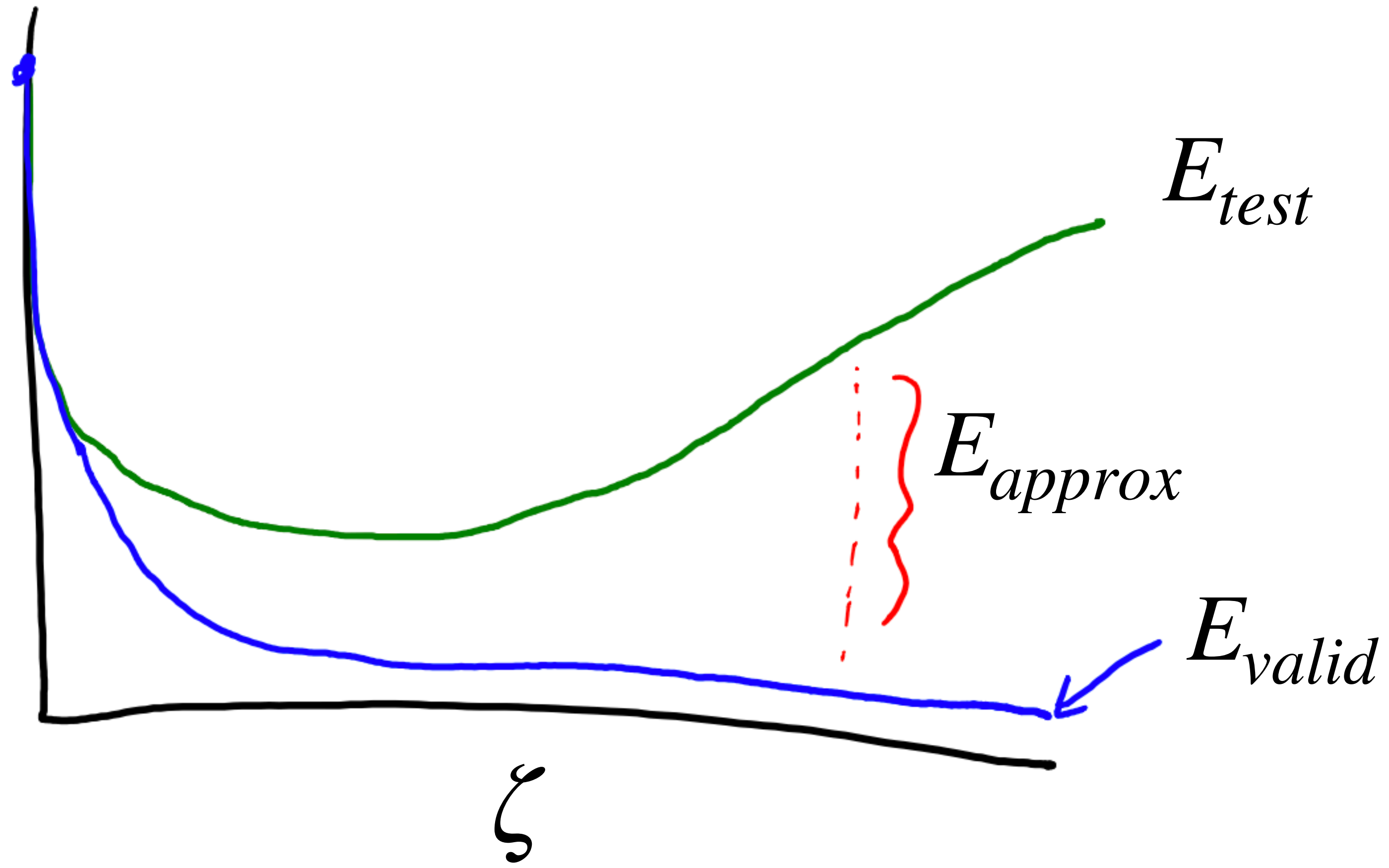
$$\mathbb{P}(E_{approx} > \epsilon \text{ for any hyper-parameter}) \leq 2\zeta \exp(-2\epsilon^2 t)$$

- Examples:
  - If $\zeta = 10$ and $t = 1000$, then probability that $E_{approx} > 0.05$ is less than 0.14.
  - If $\zeta = 10$ and $t = 10000$, then probability that $E_{approx} > 0.05$ is less than 10^{-20}.
  - If $\zeta = 10$ and $t = 1000$, then probability that $E_{approx} > 0.1$ is less than 2.7 (we cannot rule out that E_{approx} will not be small).
  - If $\zeta = 100$ and $t = 100000$, then probability that $E_{approx} > 0.01$ is less than 10^{-6}.

Bounding E_{approx}

- Fix the number of samples t. The the last probability result tells us that E_{valid} goes to zero as ζ increases, but E_{approx} increase.

- This implies that we overfit the validation data set.



Learning from data book

- Book

Generalization Error

- An alternative measure of performance is the **generalization error**.
  - Average error over a data set that has not been seen in the training set.
- **Test error** vs **generalization error** when labels are deterministic:

Test Error

$$E_{test} = \mathbb{E}[\,|\,b^i_{model} - b^i_{true}\,|\,]$$

Generalization Error

$$E_{gen} = \frac{1}{t} \sum_{(\alpha_i, b_i)\text{ not in training set}} |\,b^i_{model} - b^i_{true}\,|$$

Best and the Good Machine Learning Models

- Question 1: what is the “best” machine learning model?
  - The model that gets the lower generalization error
- Question 2: which models always do better than random guessing?
  - Models with lower generalization error than for random models

No Free Lunch Theorem

- No free lunch theorem:
  - There is no “best” model achieving the best generalization error for every problem.
  - If model A generalizes better to new data than model B on one dataset, there is another dataset where model B works better.

No Free Lunch Theorem

- Let's show the "no free lunch" theorem in a simple setting:
  - Let $\alpha_i \in \{0,1\}^d$ and $b_i \in \{0,1\}$ be binary
- We define a "learning problem" as a map from each of the 2^d feature combinations to 0 or 1: $\{0,1\}^d \to \{0,1\}$.

No Free Lunch Theorem

Feature 1	Feature 2	Feature 3
0	0	0
0	0	1
0	1	0
…	…	…

Map 1	Map 2	Map 3	…
0	1	0	…
0	0	1	…
0	0	0	…
…	…	…	…

- Let's pick one of these maps ("learning problems") and:
    - Generate a training set of n iid data-points.
    - Fit model A and model B.

No Free Lunch Theorem

- Define the "unseen" set as the $2^d - n$ data-points and their labels not seen in training.
    - Generalization error is the average error on these "unseen" data.
- Suppose that model A got 1% error and model B got 60% error.
- We want to show that model B beats model A on another learning problem.

No Free Lunch Theorem

- Among the set of “learning problems” find the one where:
  - The labels b_i agree on all training data-points.
  - The labels b_i disagree on all “unseen” data-points.
- On this other “learning problem”:
  - Model A gets 99% error and model B gets 40% error.

No Free Lunch Theorem

- Further, across all “learning problems” with this n data:
  - Average generalization error of every model is 50% on unseen examples.
  - With “k” classes, the average error is (k-1)/k% (random guessing)

No Free Lunch Theorem

- This is problematic since for general problems, no "machine learning" is better than simply predicting every label to be equal to zero.

Limits of the No Free Lunch Theorem

- The issue with the previous claim is that we allowed any possible set of labels to be generated.
- Fortunately, the world is structured: some “learning” problems are more likely than others.
- For example, it’s usually the case that “similar” a_i have similar b_i.
    - Datasets with properties like this are more likely.
    - Otherwise, you probably have no hope of learning.
- Models with the right “similarity” assumptions can beat naive approaches like predicting all labels to be equal to zero.
- With assumptions like this, you can consider consistency:
    - As n grows, model A converges to the optimal test error.

Learning from data book

- Book