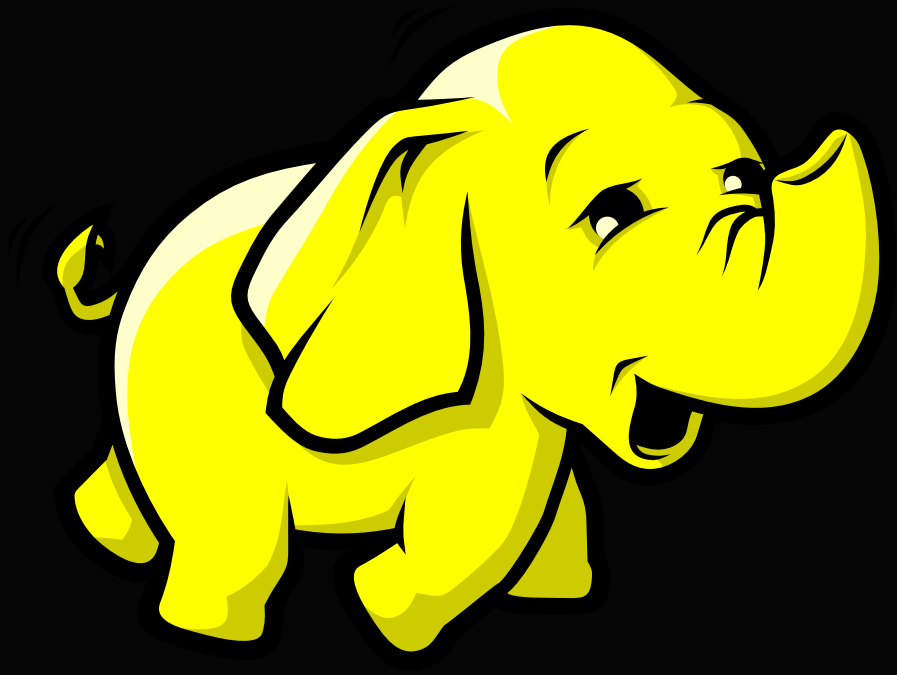
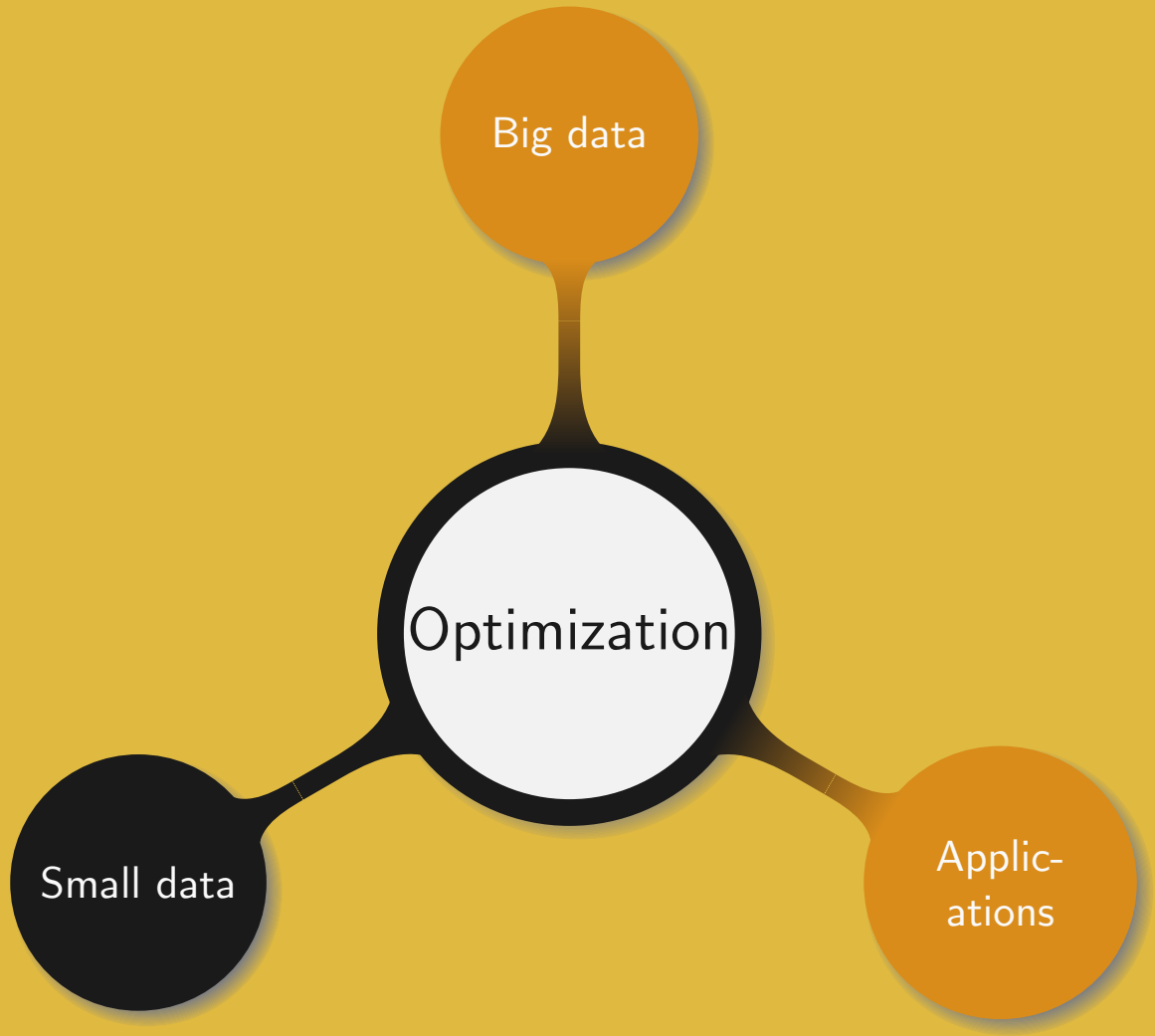


Methods and tools for big data

5. Optimization

Manuel – Summer 2022

Chapter organisation



Regression analysis

Basics on regression:

- Find the relationship between:
  - A *dependent variable*, i.e. the outcome or response variable
  - *Independent variables*, i.e. predictors or explanatory variables
- Many types of regression exist, most common ones are linear, logistics, and polynomial
- The goal is to minimize the error of the prediction

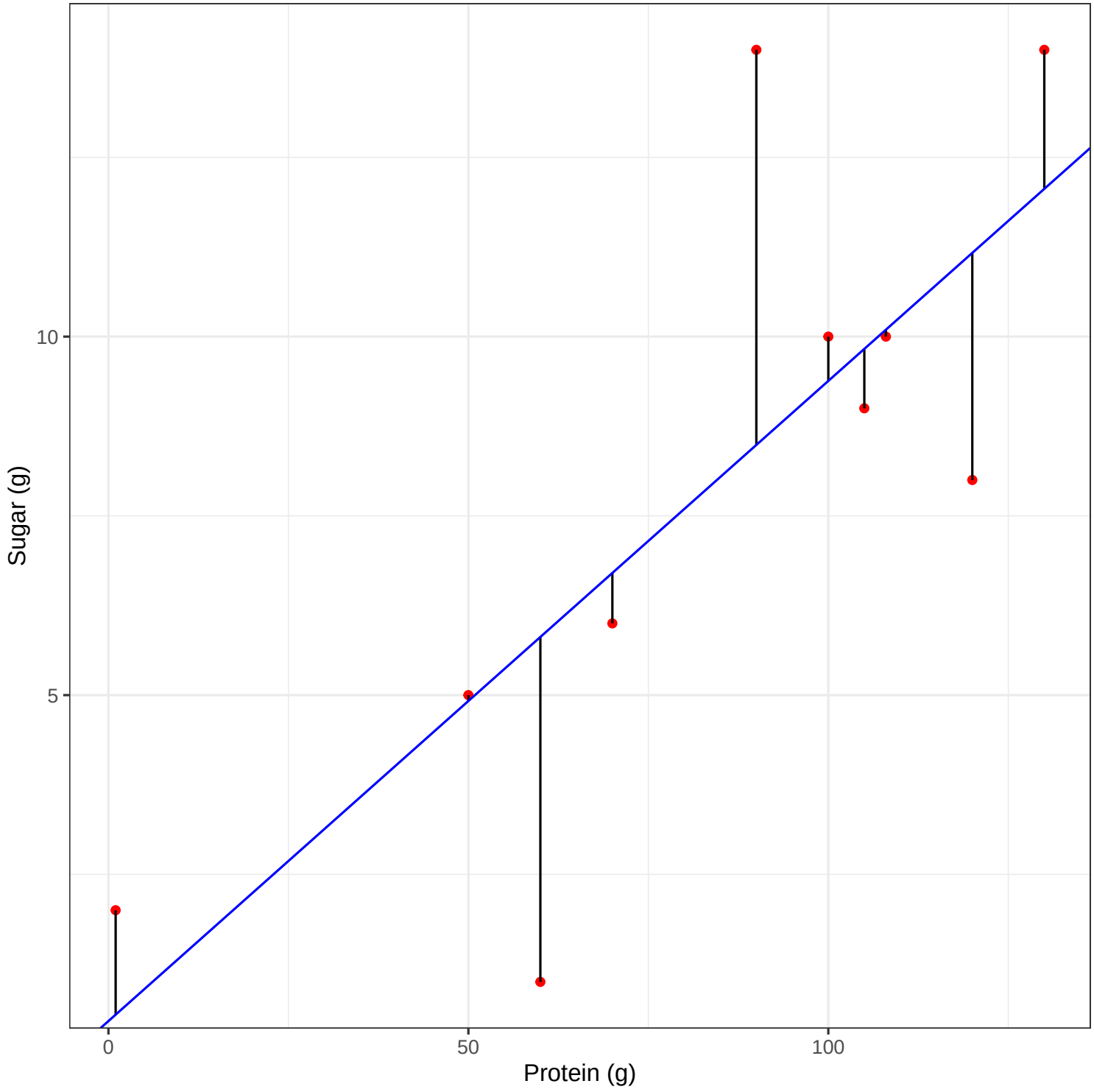
Regression analysis

Basics on regression:

- Find the relationship between:
  - A *dependent variable*, i.e. the outcome or response variable
  - *Independent variables*, i.e. predictors or explanatory variables
- Many types of regression exist, most common ones are linear, logistics, and polynomial
- The goal is to minimize the error of the prediction

Error evaluation:

- Error is often measured using the *the sum of squares*, i.e. summing up the square of the error at each point
- The minimum of the sum of squares is called *least squares*
- The smaller the error the better the model


Sugar (g)
10
5
0
50
100
Protein (g)

Optimization problems

Reminders on optimization:

- The goal is to maximize or minimize a function f depending in its input x
- The function f is called *objective function* or *criterion*
- During the minimization process f is often referred to as *cost*, *loss*, or *error function*

Optimization problems

Reminders on optimization:

- The goal is to maximize or minimize a function f depending in its input x
- The function f is called *objective function* or *criterion*
- During the minimization process f is often referred to as *cost*, *loss*, or *error function*

Remark on minimization with respect to least square:

- Whether positive or negative a large error is bad
- Squaring allows to penalize larger residuals more than smaller ones
- Error is often composed of *systematic* and *random* noises
- Minimizing the sum of squared errors is the same as minimizing the variance

Notes on optimization problems

Common examples of optimization problems:

- Chip design: ensure no tracks cross on a computer chip
- Timetable: given a list of students in each course, minimize the number of collisions
- Traveling salesman: given a list of cities, minimize the distance necessary to visit all of them

Notes on optimization problems

Common examples of optimization problems:

- Chip design: ensure no tracks cross on a computer chip
- Timetable: given a list of students in each course, minimize the number of collisions
- Traveling salesman: given a list of cities, minimize the distance necessary to visit all of them

No free lunch theorem:

- There is no best solution to all search problems
- Algorithms performing better on certain problems, do worse on others
- Work is often needed to find the most suitable algorithm

Gradient descent

Description is usually done in term of minimization

Basic setup for a function $f(X)$, where $X = (x_1, \cdots, x_n)$:

- Iteratively create a sequence of X_i
- At each step, determine the gradient in each direction
- Select a direction and keep going down
- Repeat until the gradient is 0 in all directions

Gradient descent

Description is usually done in term of minimization

Basic setup for a function $f(X)$, where $X = (x_1, \cdots, x_n)$:

- Iteratively create a sequence of X_i
- At each step, determine the gradient in each direction
- Select a direction and keep going down
- Repeat until the gradient is 0 in all directions

Remark. The function f should be convex, i.e. for any $x, y \in \mathbb{R}^d$, and $\alpha \in [0, 1]$,

$$f\left(\alpha x + (1-\alpha)y\right) \leq \alpha f(x) + (1-\alpha) f(y),$$

to ensure it has a global minimum, and the algorithm does not return a local one.

Gradient descent

Example. For $f(X) = 0.5x_1^2 + 0.2x_2^2 + 0.6x_3^2$, we obtain

$$\nabla f(x) = (x_1, 0.4x_2, 1.2x_3).$$

Using a step of length 1, and starting with $X_0 = (-2, 2, -2)$, the steepest downhill direction is $(-2, 0.8, -2.4)$, yielding $X_1 = (0, 1.2, 0.4)$.

After a few iterations we find $X_6 = (0, 0.0569, 0.0000256)$.

Gradient descent

Example. For $f(X) = 0.5x_1^2 + 0.2x_2^2 + 0.6x_3^2$, we obtain

$$\nabla f(x) = (x_1, 0.4x_2, 1.2x_3).$$

Using a step of length 1, and starting with $X_0 = (-2, 2, -2)$, the steepest downhill direction is $(-2, 0.8, -2.4)$, yielding $X_1 = (0, 1.2, 0.4)$.

After a few iterations we find $X_6 = (0, 0.0569, 0.0000256)$.

Remarks.

- Using a constant step could lead to either going very slowly or over-stepping the minimum
- Computation can be speeded up by using Taylor expansion. This involves inverting the Hessian matrix of f which has a cost of $\mathcal{O}(n^3)$. However this method allows to keep a constant step of 1, simplifying other parts of the computation

Comparison of gradient descent methods

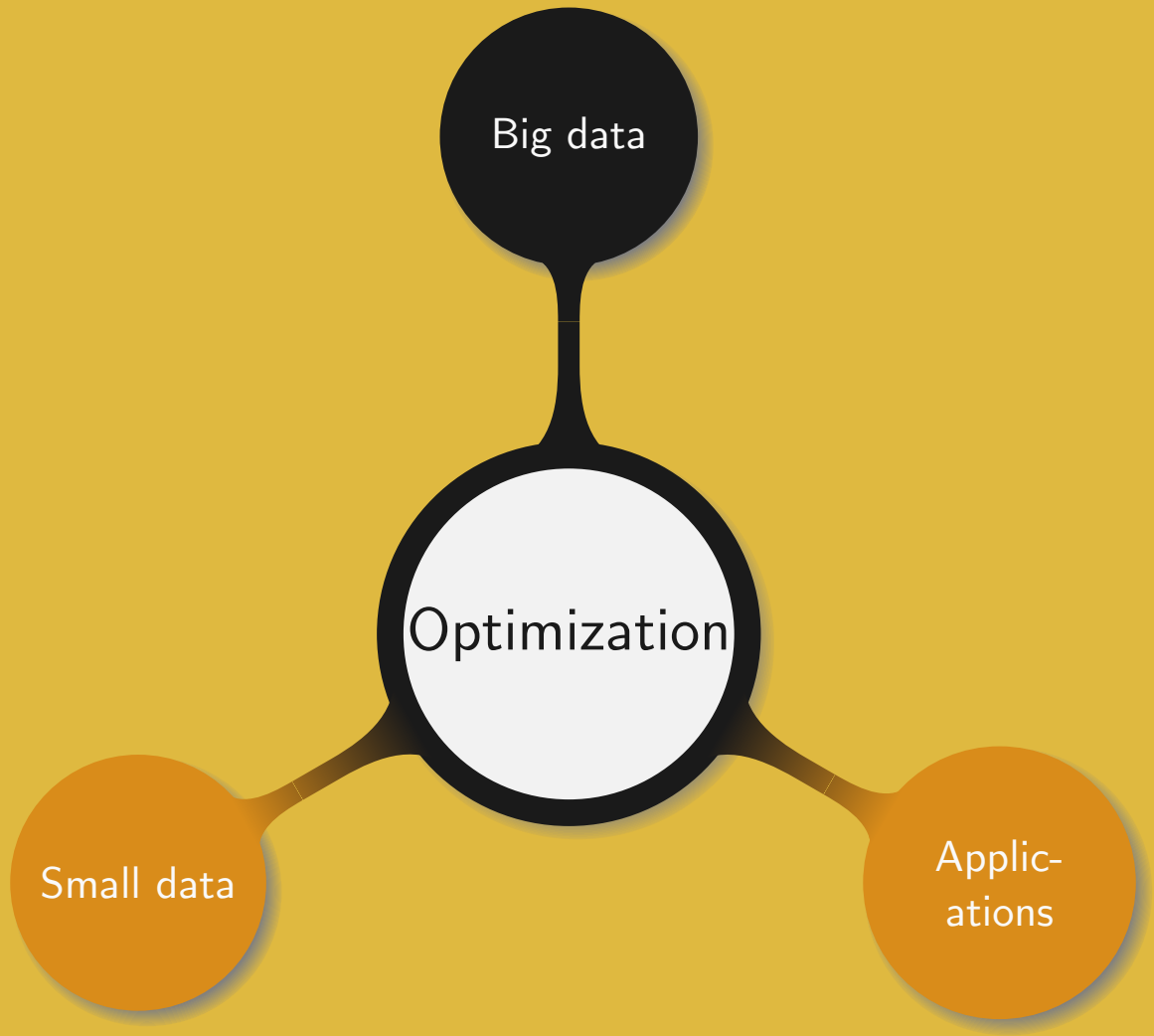
Gradient descent:

- Uses the whole dataset
- Deterministic method
- Slow but fast to converge
- Yields an optimal solution
- Slow to escape local minima

Stochastic gradient descent:

- Randomly selects a sample
- Stochastic method
- Fast but slow to converge
- Yields a good enough solution
- Faster to escape local minima

Chapter organisation



Directed Acyclic Graph

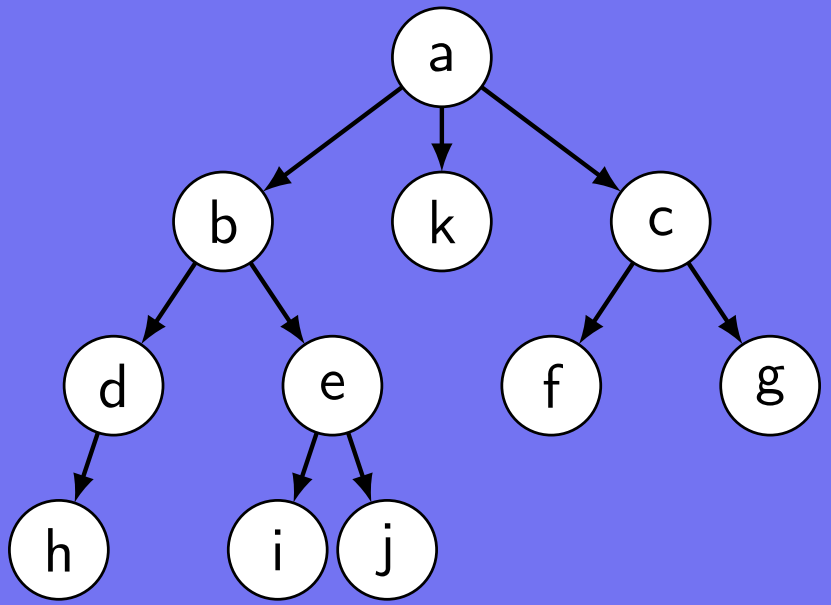
When working with PRAM (slide 1.9):

- Dependencies between instructions are represented using a DAG
- Each instruction is represented as a node
- An edge (u, v) represents the dependency of u upon v
- The root corresponds to the result of the computation

Directed Acyclic Graph

When working with PRAM (slide 1.9):

- Dependencies between instructions are represented using a DAG
- Each instruction is represented as a node
- An edge (u, v) represents the dependency of u upon v
- The root corresponds to the result of the computation



Work-depth model

Computation finishes when the last processor completes its job:

- The amount of time when using one CPU is referred to as T_1
- The amount of time when using p CPUs is referred to as T_p
- The amount of time when using infinitely many CPUs is referred to as T_∞

Work-depth model

Computation finishes when the last processor completes its job:

- The amount of time when using one CPU is referred to as T_1
- The amount of time when using p CPUs is referred to as T_p
- The amount of time when using infinitely many CPUs is referred to as T_∞

Remarks.

- The *depth of an algorithm* is defined with respect to the last CPU to complete its tasks
- The *work of an algorithm* corresponds to the amount of time necessary to complete all tasks multiplied by the number of CPUs

Does T_∞ tend to zero?

Bounds on T_p

Brent's theorem:

$$\frac{T_1}{p} \leq T_p \leq \frac{T_1}{p} + T_\infty$$

Bounds on T_p

Brent's theorem:

$$\frac{T_1}{p} \leq T_p \leq \frac{T_1}{p} + T_\infty$$

Meaning of the result:

- If work is evenly shared among all p CPUs then we get $\frac{T_1}{p}$
- T_∞ helps define how far we are from the ideal case

Remarks.

- Together T_1 and T_∞ provide information on how well an algorithm performs on p CPUs
- Increasing the number of CPUs will never impact performance

Bounds on T_p

Brent's theorem:

$$\frac{T_1}{p} \leq T_p \leq \frac{T_1}{p} + T_\infty$$

Meaning of the result:

- If work is evenly shared among all p CPUs then we get $\frac{T_1}{p}$
- T_∞ helps define how far we are from the ideal case

Remarks.

- Together T_1 and T_∞ provide information on how well an algorithm performs on p CPUs
- Increasing the number of CPUs will never impact performance

The work-depth model helps designing better parallel algorithms

Back to gradient descent

From a general point of view gradient descent is an optimization problem

$$\min_w(F(w)) = \sum_{i=1}^{m} F_i(w, x_i, y_i),$$

where $x_i \in \mathbb{R}^n$, $y_i \in \mathbb{R}$ is a "label", and w is the parameter we expect to optimize over.

In essence gradient descent starts with a random initial w and has the goal of iteratively improving on it, by computing $w_{k+1} = w_k - \alpha \nabla F(w_k)$, for some small α. In our case the objective function F corresponds to a loss function.

Remark. From a theoretical point of view, this works especially well when F is strongly convex, differentiable, and ∇F is L-Lipschitz continuous. In such a case taking $\alpha < \frac{1}{L}$ leads to an exponential convergence rate to a global minimum!

Gradient descend – Sum of squares

For the sum of squares loss function, we want to minimize

$$F(w) = \sum_{i=1}^{m} F_i(w, x_i, y_i) = \sum_{i=1}^{m} \|x_i^\top w - y_i\|_2^2,$$

with x_i, y_i, and w as above.

Notes on our setup:

- F is strongly convex and Lipschitz continuous
- m corresponds to the data parallelism
- n corresponds to the model parallelism

How well does gradient descent scale up?

Parallel summation

How to sum up n elements in parallel?

Algorithm. (*Basic summation*)

Input : a an array with n elements
Output: s the sum over all the elements of a

```
1 s ← 0 ;
2 for i ← 1, ··· , n do
3 |   s ← s + a[i] ;
4 end for
5 return s
```

Basic summation:

- Work: $\mathcal{O}(n)$
- Depth: $\mathcal{O}(n)$

How to achieve:

- Work: $\mathcal{O}(n)$
- Depth: $\mathcal{O}(\log n)$

Gradient descend – Work and depth

Complexity of computing

$$F(w) = \sum_{i=1}^{m} \|x_i^\top w - y_i\|_2^2$$

- Work: $\mathcal{O}(mn)$
- Depth: $\mathcal{O}(\log mn)$

Complexity of computing

$$\nabla F(w) = 2 \sum_{i=1}^{m} x_i^\top (x_i^\top w - y_i)$$

- Work: $\mathcal{O}(mn)$
- Depth: $\mathcal{O}(\log mn)$

Gradient descend – Work and depth

Complexity of computing

$$F(w) = \sum_{i=1}^{m} \|x_i^\top w - y_i\|_2^2$$

- Work: $\mathcal{O}(mn)$
- Depth: $\mathcal{O}(\log mn)$

Complexity of computing

$$\nabla F(w) = 2 \sum_{i=1}^{m} x_i^\top (x_i^\top w - y_i)$$

- Work: $\mathcal{O}(mn)$
- Depth: $\mathcal{O}(\log mn)$

Performing a complete gradient descend:

- Error $\frac{1}{\varepsilon}$ can be achieved after $\mathcal{O}\left(\log \frac{1}{\varepsilon}\right)$ iterations
- Total depth: $\mathcal{O}\left(\log \frac{1}{\varepsilon} \log mn\right)$

How suitable is gradient descent for parallelization and big data?

Stochastic gradient descend

Instead of computing a full gradient, apply it to a randomly selected point

Stochastic gradient descent (using notations from slide 5.14):

- Call s_k the index uniformly sampled at iteration k
- Compute the sequence

$$w_{k+1} = w_k - \alpha \nabla F_{s_k}(w_k)$$

Stochastic gradient descend

Instead of computing a full gradient, apply it to a randomly selected point

Stochastic gradient descent (using notations from slide 5.14):

- Call s_k the index uniformly sampled at iteration k
- Compute the sequence

$$w_{k+1} = w_k - \alpha \nabla F_{s_k}(w_k)$$

Notes on stochastic gradient descent:

- Error ε will be achieved after $\mathcal{O}\left(\frac{1}{\varepsilon}\right)$ iterations
- Work decreases linearly with the number of points considered
- The number of iterations does not increase linearly with the number of sample points

Stochastic vs. batch gradient descent

Gradient descent:

- Per iteration:
  - Work: $\mathcal{O}(mn)$
  - Depth: $\mathcal{O}(\log mn)$
- Total:
  - Work: $\mathcal{O}\left(mn \log \frac{1}{\varepsilon}\right)$
  - Depth: $\mathcal{O}\left(\log \frac{1}{\varepsilon} \log mn\right)$

Stochastic gradient descent:

- Per iteration:
  - Work: $\mathcal{O}(n)$
  - Depth: $\mathcal{O}(\log n)$
- Total:
  - Work: $\mathcal{O}\left(\frac{n}{\varepsilon}\right)$
  - Depth: $\mathcal{O}\left(\frac{\log n}{\varepsilon}\right)$

Stochastic vs. batch gradient descent

Gradient descent:

- Per iteration:
  - Work: $\mathcal{O}(mn)$
  - Depth: $\mathcal{O}(\log mn)$
- Total:
  - Work: $\mathcal{O}\left(mn \log \frac{1}{\varepsilon}\right)$
  - Depth: $\mathcal{O}\left(\log \frac{1}{\varepsilon} \log mn\right)$

Stochastic gradient descent:

- Per iteration:
  - Work: $\mathcal{O}(n)$
  - Depth: $\mathcal{O}(\log n)$
- Total:
  - Work: $\mathcal{O}\left(\frac{n}{\varepsilon}\right)$
  - Depth: $\mathcal{O}\left(\frac{\log n}{\varepsilon}\right)$

Which is best, gradient descent or stochastic gradient descent?

Parallelizing Stochastic gradient descent

Setup in PRAM:

- Save w in a shared piece of the memory
- All CPUs have access to w and the whole dataset

Parallelizing Stochastic gradient descent

Setup in PRAM:

- Save w in a shared piece of the memory
- All CPUs have access to w and the whole dataset

Things which could go wrong:

- A model is read, transformed, and written in the memory but in the meantime the model has been updated by another CPU
- An updated model is overwritten by an older one

Parallelizing Stochastic gradient descent

Setup in PRAM:

- Save w in a shared piece of the memory
- All CPUs have access to w and the whole dataset

Things which could go wrong:

- A model is read, transformed, and written in the memory but in the meantime the model has been updated by another CPU
- An updated model is overwritten by an older one

How bad is this situation?

Parallelizing Stochastic gradient descent

Adding locks:

- Solves the race condition problem
- Only one CPU can access w at a time
- All the benefits from the parallelism get lost

Are locks really needed?

Parallelizing Stochastic gradient descent

Adding locks:

- Solves the race condition problem
- Only one CPU can access w at a time
- All the benefits from the parallelism get lost

Are locks really needed?

In stochastic gradient descent:

- Both $x_i = (x_i^{(1)}, \cdots, x_i^{(n)})$ and $w = (w^{(1)}, \cdots, w^{(n)})$ are n-dimensional vectors
- If x_i is sparse, then only a few $w^{(k)}$ will be updated

In a big data setup the probability of collision will likely be low

Going Hogwild!

Hogwild! strategy for p processors:

- Proceed in parallel over all available CPUs
- Until the expected error condition is met:
  - Select a random index j from $\{1, \cdots, m\}$
  - Concurrently compute $F_j(w)$ and $\nabla F_j(w)$ for the w currently in the shared memory
  - For all k such that $x_i^{(k)} \neq 0$, update $w^{(k)}$ with $w^{(k)} - \alpha \left[\nabla F_j(w)\right]^{(k)}$

Going Hogwild!

Hogwild! strategy for p processors:

- Proceed in parallel over all available CPUs
- Until the expected error condition is met:
  - Select a random index j from $\{1, \cdots, m\}$
  - Concurrently compute $F_j(w)$ and $\nabla F_j(w)$ for the w currently in the shared memory
  - For all k such that $x_i^{(k)} \neq 0$, update $w^{(k)}$ with $w^{(k)} - \alpha \left[\nabla F_j(w)\right]^{(k)}$

Remarks.

- No locks are used, leading to speed up nearly linear in term of CPUs
- The locking overhead should be avoided when processing big data
- Hogwild! requires the cost function to be sparse

Stochastic vs. batch gradient descent

Looking back at gradient descent in PRAM:

- Stochastic has lower depth but is hard to parallelize
- Batch is easier to parallelize but slower
- Hogwild! renders stochastic almost *embarrassingly parallel*

Stochastic vs. batch gradient descent

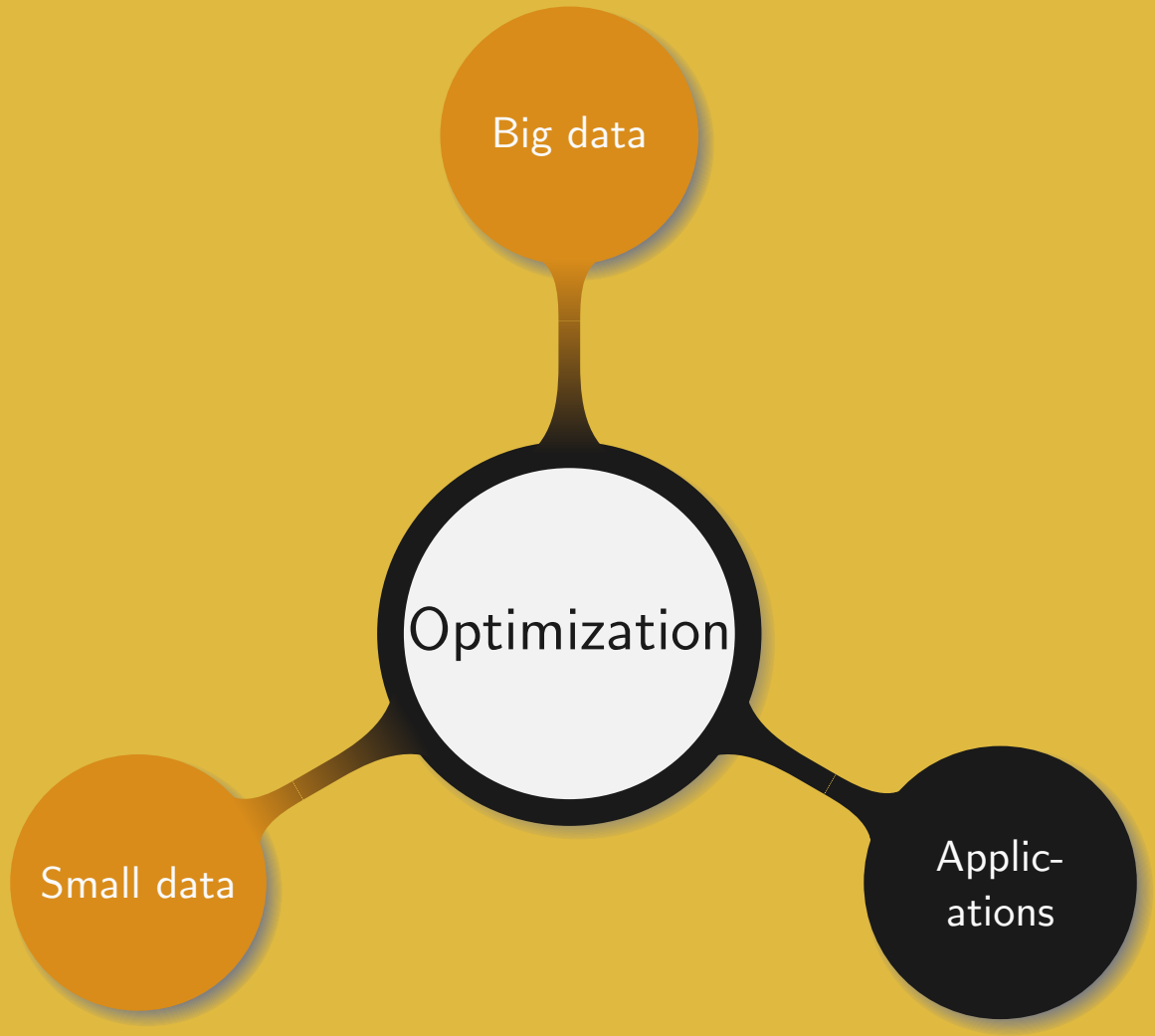
Looking back at gradient descent in PRAM:

- Stochastic has lower depth but is hard to parallelize
- Batch is easier to parallelize but slower
- Hogwild! renders stochastic almost *embarrassingly parallel*

Stochastic and batch on a distributed system:

- Communication based on the number of iterations:
  - Batch: $\mathcal{O}\left(\log \frac{1}{\varepsilon}\right)$
  - Stochastic: $\mathcal{O}\left(\frac{1}{\varepsilon}\right)$
- Stochastic is often preferred on a single computer with many GPUs
- Batch is more common on distributed systems

Chapter organisation



Implementing gradient descent

Questions to consider first:

- What tool to use?
- Is batch or stochastic gradient descent most appropriate?
- How large is the data, i.e. can n and m fit in memory?

Implementing gradient descent

Questions to consider first:

- What tool to use?
- Is batch or stochastic gradient descent most appropriate?
- How large is the data, i.e. can n and m fit in memory?

In our setup we expect to:

- Use Spark
- Check how both gradient descents strategies behave
- Work with n small enough to fit in memory but no restriction on m

How to store the data on the cluster?

Batch gradient descent in Spark

High-level idea for a Spark implementation:

1. Organise the data by row and store it in an RDD
2. Use a `map` transformation to generate a closure for each point
3. Use the `cache` action to ensure Spark keeps the RDD in memory
4. For each point p, apply a `map` to transform p into $\nabla F_p(w)$
5. Apply a `reduce` to sum up all the $\nabla F_p(w)$
6. Update w and repeat from step 4 until the expected error is reached

Batch gradient descent in Spark

Optimizing our approach:

- How many times is w sent?
- Is w modified by the mappers?
- With respect to bandwidth and memory usage, how large is w?
- How should w be shared among all the machines?

Batch gradient descent in Spark

Optimizing our approach:

- How many times is w sent?
- Is w modified by the mappers?
- With respect to bandwidth and memory usage, how large is w?
- How should w be shared among all the machines?

Basic analysis:

- Where is the bottleneck in our approach?
- How good or bad is the communication cost?

Stochastic gradient descent in Spark

Reminders on stochastic gradient descent:

- Total depth is much larger than for batch gradient descent
- Apply Hogwild! to speed up the process

Stochastic gradient descent in Spark

Reminders on stochastic gradient descent:

- Total depth is much larger than for batch gradient descent
- Apply Hogwild! to speed up the process

Hogwild! on Spark:

- Broadcast needs to be completed to start the mappers
- All mappers and reducers must be done before broadcasting again
- Spark achieves fault-tolerance through synchronisation barriers

Stochastic gradient descent in Spark

Reminders on stochastic gradient descent:

- Total depth is much larger than for batch gradient descent
- Apply Hogwild! to speed up the process

Hogwild! on Spark:

- Broadcast needs to be completed to start the mappers
- All mappers and reducers must be done before broadcasting again
- Spark achieves fault-tolerance through synchronisation barriers

Minimizing an objective function in Spark:

- Try random updates and accept any one lowering the objective
- Use mini-batches:
  - At each iteration select "many" samples, instead of one
  - Apply batch gradient descent to them

Gradient descent and PCA

Common strategy for gradient descent when data is too large:

- Extract the principal components of the big dataset
- Apply gradient descent on the resulting approximation

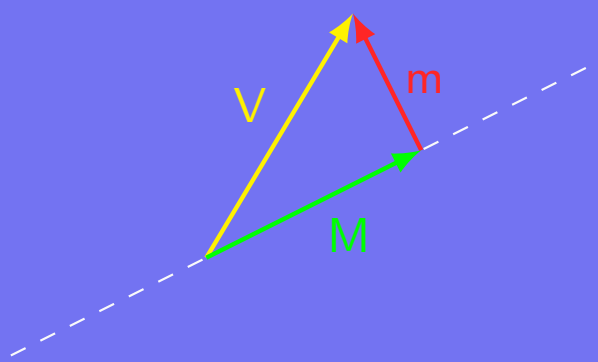
Gradient descent and PCA

Common strategy for gradient descent when data is too large:

- Extract the principal components of the big dataset
- Apply gradient descent on the resulting approximation

Relating gradient descent to PCA:

- Find the axes which maximize the variance
- Find the direction for which the expectation of $XX^\top$ is maximized
- Find the direction which minimizes the residuals



For mean centered data:

- Pythagorean theorem: $V^2 = M^2 + m^2$
- PCA maximizes M
- Gradient descent minimizes m

Linear programming

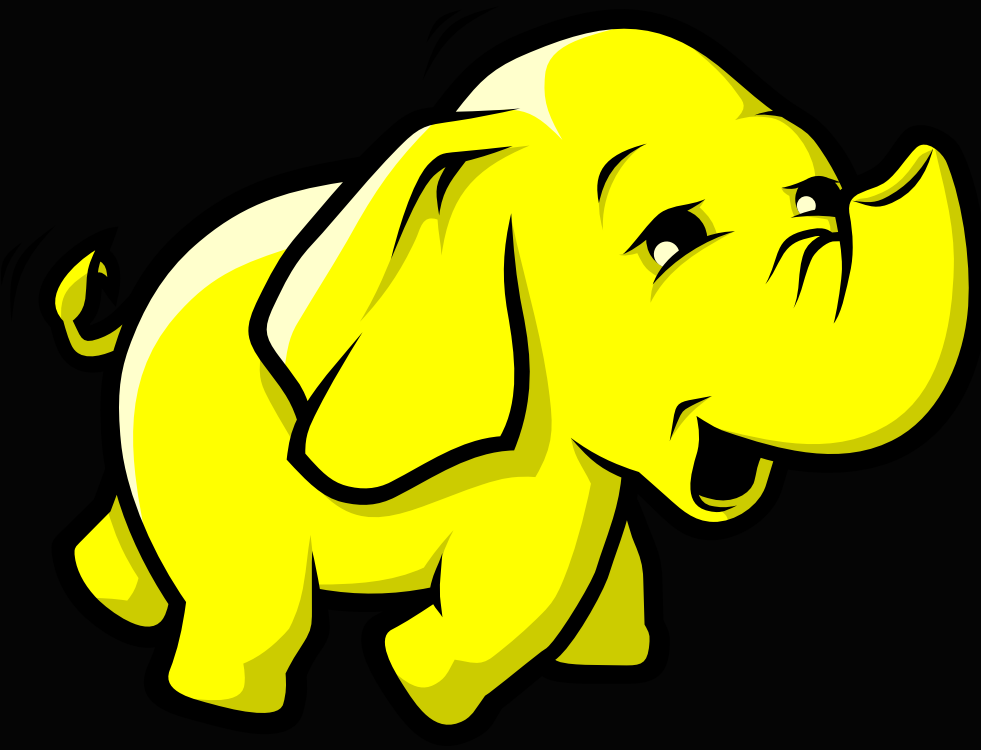
Given a mathematical model where an objective is represented as a linear function and some constraints are expressed as equalities or inequalities, maximize the objective while respecting the constraints.

A gradient descent reading of linear programming:

- Any maximization problem can be rephrased into a minimization one
- The simplex method solves linear programming problems:
  - A basic solution is used to start with
  - A "local search" finds a *pivot* to improve on the basic solution
  - Different choices of pivot lead to different convergence rates
  - Any pivot leads to an optimal solution
- Finding the best pivot in the simplex is equivalent to finding the best direction to improve on the objective

Key points

- What are batch and stochastic gradient descents?
- How to batch and stochastic gradient descent compare?
- What are PRAM, work, and depth?
- Explain how Hogwild! works
- Discuss gradient descent implementation on Spark



Thank you!