Data contains value and knowledge
Data Mining

- But to extract the knowledge data needs to be
  - Stored (systems)
  - Managed (databases)
  - And ANALYZED ← this class

Data Mining ≈ Big Data ≈ Predictive Analytics ≈ Data Science ≈ Machine Learning
Data mining = extraction of actionable information from (usually) very large datasets, is the subject of extreme hype, fear, and interest

- It’s not all about machine learning
- But most of it is

- Emphasis in CS246 on algorithms that scale
  - Parallelization often essential
Data Mining Methods

- **Descriptive methods**
  - Find human-interpretable patterns that describe the data
    - **Example**: Clustering

- **Predictive methods**
  - Use some variables to predict unknown or future values of other variables
    - **Example**: Recommender systems
This combines best of machine learning, statistics, artificial intelligence, databases but more stress on:

- **Scalability** (big data)
- **Algorithms**
- **Computing architectures**
- Automation for handling large data
What will we learn?

- **We will learn to mine different types of data:**
  - Data is high dimensional
  - Data is a graph
  - Data is infinite/never-ending
  - Data is labeled

- **We will learn to use different models of computation:**
  - MapReduce
  - Streams and online algorithms
  - Single machine in-memory
What will we learn?

- We will learn to **solve real-world problems:**
  - Recommender systems
  - Market Basket Analysis
  - Spam detection
  - Duplicate document detection

- We will learn **various “tools”:**
  - Linear algebra (SVD, Rec. Sys., Communities)
  - Optimization (stochastic gradient descent)
  - Dynamic programming (frequent itemsets)
  - Hashing (LSH, Bloom filters)
How the Class Fits Together

**High dim. data**
- Locality sensitive hashing
- Clustering
- Dimensionality reduction

**Graph data**
- PageRank, SimRank
- Graph Neural Networks
- Spam Detection

**Infinite data**
- Filtering data streams
- Web advertising
- Queries on streams

**Machine learning**
- Learning Embeddings
- Decision Trees
- Experimentation

**Apps**
- Recommender systems
- Association Rules
- Duplicate document detection

How do you want that data?
Course Logistics
Course Staff

Instructor

Jure Leskovec

Co-Instructor

Mina Ghashami

Course Coordinator

Course Assistants

Hongyu Ren
Head TA

Nikhil Cheerla

Tracey Chen

Drew Kaul

Jacky Lin

Yige Liu

Mihir Patel

Xuan Su
Lectures: Tue/Thu 1:30-3:00pm PST
Live in-person (in NVIDIA classroom), recording available on Canvas

- ~70 min lecture:
  - If you have a clarification question, post it in Ed, TAs will answer

- ~20 min Q&A:
  - Ask questions, Jure will answer and discuss
Logistics: Communication

- **Ed:**
  - Use Ed for all questions and public communication
    - Search the feed before asking a duplicate question
    - Please tag your posts and please no one-liners
  
- **For e-mailing course staff always use:**
  - cs246-win2122-staff@lists.stanford.edu
  
- **We will post course announcements to Ed (hence check it regularly!)**

Auditors are welcome!

(please send request to Lata Nair <lnairp24@stanford.edu> to add you to Canvas)
High-frequency feedback:

- Weekly survey about class morale
- Randomly select students to give us feedback
  - Content
  - Course setup
  - Anything the teaching team should know/improve
  - Anything that is confusing to you
  - ...

Resources

- **Course website:** [http://cs246.stanford.edu](http://cs246.stanford.edu)
  - Lecture slides (at least 30min before the lecture)
  - Homework, solutions, readings posted on Ed/Canvas

- **Class textbook:** *Mining of Massive Datasets* by A. Rajaraman, J. Ullman, and J. Leskovec
  - Sold by Cambridge Uni. Press but available for free at [http://mmds.org](http://mmds.org)

- **MOOC:** [www.youtube.com /channel/UC_Oao2FYkLAUlUVkBfze4jg/videos](http://www.youtube.com /channel/UC_Oao2FYkLAUlUVkBfze4jg/videos)
Office hours:

- See course website [http://cs246.stanford.edu](http://cs246.stanford.edu) for TA office hours
  - *We start Office Hours this Friday!*

- Office hours will be held on Zoom and use [QueueStatus](#)
  - Links will be posted on Ed and Canvas
  - We will hold special group office hours, homework review office hours as well as one-on-one office hours
Recitation Sessions

- Videos and materials on Canvas
- **Spark tutorial:**
  - Video
  - Follows Colab 0
- **Review of basic probability and proof techniques:**
  - Video and handout
- **Review of linear algebra:**
  - Video and handout
4 longer homeworks: 40%

- Four major assignments, involving programming, proofs, algorithm development.
- Assignments take lots of time (+20h). Start early!!

How to submit?

- Homework write-up:
  - Submit via Gradescope
  - Enroll to CS246 on Canvas, and you will be automatically added to the course Gradescope

- Homework code:
  - If the homework requires a code submission, you will find a separate assignment for it on Gradescope, e.g., HW1 (Code)
  - Forgetting to submit code will result in point deduction.
Homework schedule:

<table>
<thead>
<tr>
<th>Date (23:59 PT)</th>
<th>Out</th>
<th>In</th>
</tr>
</thead>
<tbody>
<tr>
<td>01/06, Thu</td>
<td>HW1</td>
<td></td>
</tr>
<tr>
<td>01/20, Thu</td>
<td>HW2</td>
<td>HW1</td>
</tr>
<tr>
<td>02/03, Thu</td>
<td>HW3</td>
<td>HW2</td>
</tr>
<tr>
<td>02/17, Thu</td>
<td>HW4</td>
<td>HW3</td>
</tr>
<tr>
<td>03/03, Thu</td>
<td>HW4</td>
<td></td>
</tr>
</tbody>
</table>

Two late periods for HWs for the quarter:

- Late period expires on the following Monday 23:59 PST
- Can use max 1 late period per HW
Work for the Course: Colabs

- **Short weekly Colab notebooks: 30%**
  - Colab notebooks are posted every **Thursday**
    - 10 in total, from 0 to 9, each worth 3%
  - Due one week later on **Thursday 23:59 PST. No late days!**
    - First 2 Colabs will be posted on Thu, including detailed submission instructions to Gradescope
    - Colab 0 (Spark Tutorial) is solved step-by-step in the [Spark Recitation video](http://cs246.stanford.edu).
  - Colabs require around **1hr of work.**
    - And a few lines of code.
  - “Colab” is a **free cloud service from Google**, hosting Jupyter notebooks with free access to GPU and TPU
Final exam: 30%
- Exact format will be announced later.
- Most likely we will do a take-home 3h exam which you will be able to take at any time during a 24h time window.

Extra credit: Proportional to your contribution (up to 2%)
- Course attendance, asking questions, discussion
- For participating in Ed discussions
  - Especially valuable are answers to questions posed by other students
- Reporting bugs in course materials
Prerequisites

- **Programming**: Python or Java
- **Basic Algorithms**: CS161 is surely sufficient
- **Probability**: e.g., CS109 or Stats116
  - There will be a review session and a review doc is linked from the class home page
- **Linear algebra**:
  - Another review doc + review session is available
- **Multivariable calculus**
- **Database systems** (SQL, relational algebra):
  - CS145 is sufficient but not necessary
Each of the topics listed is important for a part of the course:
- If you are missing an item of background, you could consider just-in-time learning of the needed material.

The exception is programming:
- To do well in this course, you really need to be comfortable with writing code in Python or Java.
We’ll follow the standard CS Dept. approach: You can get help, but you **MUST** acknowledge the help on the work you hand in.

Failure to acknowledge your sources is a violation of the Honor Code.

We use MOSS to check the originality of your code.
Honor Code – (2)

- You can talk to others about the algorithm(s) to be used to solve a homework problem;
  - As long as you then mention their name(s) on the work you submit.
- You should not use code of others or be looking at code of others when you write your own:
  - (don’t search/post code on Github, and similar)
  - You can talk to people but have to write your own solution/code
  - If you fail to mention your sources, MOSS will catch it, which will result in an HC violation.
Final Thoughts

- **CS246 is fast paced!**
  - Requires programming maturity
  - Strong math skills
    - SCPD students tend to be rusty on math/theory
- **Course time commitment:**
  - Homeworks take +20h
  - Colab notebooks take about 1h
- Form study groups

- It’s going to be **fun and hard** work. 😊
Distributed Computing for Data Mining
Large-scale computing for data mining problems on commodity hardware

Challenges:

- How do you distribute computation?
- How can we make it easy to write distributed programs?

Machines fail:
- One server may stay up 3 years (1,000 days)
- If you have 1,000 servers, expect to lose 1/day
- With 1M machines 1,000 machines fail every day!
An Idea and a Solution

**Issue:**
Copying data over a network takes time

**Idea:**
- Bring computation to data
- Store files multiple times for reliability

**Spark/Hadoop** address these problems
- **Storage Infrastructure – File system**
  - Google: GFS. Hadoop: HDFS
- **Programming model**
  - MapReduce
  - Spark

Problem:
- If nodes fail, how to store data persistently?

Answer:
- **Distributed File System**
  - Provides global file namespace

Typical usage pattern:
- Huge files (100s of GB to TB)
- Data is rarely updated in place
- Reads and appends are common
Distributed File System

- **Chunk servers**
  - File is split into contiguous chunks
  - Typically each chunk is 16-64MB
  - Each chunk replicated (usually 2x or 3x)
  - Try to keep replicas in different racks

- **Master node**
  - a.k.a. Name Node in Hadoop’s HDFS
  - Stores metadata about where files are stored
  - Master nodes are typically more robust to hardware failure and run critical cluster services.

- **Client library for file access**
  - Talks to master to find chunk servers
  - Connects directly to chunk servers to access data
- **Reliable distributed file system**
- Data kept in “chunks” spread across machines
- Each chunk **replicated** on different machines

- **Seamless recovery from disk or machine failure**

![Diagram of distributed file system with chunk servers and data chunks]

**Notation:** $C_2$... chunk no. 2 of file C

**Bring computation directly to the data!**

**Chunk servers also serve as compute servers**
MapReduce: Early Distributed Computing Programming Model
MapReduce is a style of programming designed for:

1. Easy parallel programming
2. Invisible management of hardware and software failures
3. Easy management of very-large-scale data

It has several implementations, including Hadoop, Spark (used in this class), Flink, and the original Google implementation just called “MapReduce”
3 steps of MapReduce

- **Map:**
  - Apply a user-written *Map function* to each input element
    - *Mapper* applies the Map function to a single element
    - Many mappers grouped in a *Map task* (the unit of parallelism)
  - The output of the Map function is a set of 0, 1, or more key-value pairs.

- **Group by key:** Sort and shuffle
  - System sorts all the key-value pairs by key, and outputs key-(list of values) pairs

- **Reduce:**
  - User-written *Reduce function* is applied to each key-(list of values)

Outline stays the same, Map and Reduce change to fit the problem
MapReduce Pattern

key-value pairs

Input

Mappers

Reducers

Output

Example MapReduce task:

- We have a huge text document
- Count the number of times each distinct word appears in the file

Many applications of this:

- Analyze web server logs to find popular URLs
- Statistical machine translation:
  - Need to count number of times every 5-word sequence occurs in a large corpus of documents
The crew of the space shuttle Endeavor recently returned to Earth as ambassadors, harbingers of a new era of space exploration. Scientists at NASA are saying that the recent assembly of the Dextre bot is the first step in a long-term space-based man/machine partnership. "The work we're doing now -- the robotics we're doing -- is what we're going to need.....................
map(key, value):
# key: document name; value: text of the document
    for each word w in value:
        emit(w, 1)

reduce(key, values):
# key: a word; value: an iterator over counts
    result = 0
    for each count v in values:
        result += v
    emit(key, result)
Map-Reduce: A diagram

**MAP:**
Read input and produces a set of key-value pairs

**Intermediate**

<table>
<thead>
<tr>
<th>k1:v</th>
<th>k1:v</th>
<th>k2:v</th>
<th>k1:v</th>
<th>k3:v</th>
<th>k4:v</th>
<th>k4:v</th>
<th>k5:v</th>
<th>k4:v</th>
<th>k1:v</th>
<th>k3:v</th>
</tr>
</thead>
</table>

**Group by key:**
Collect all pairs with same key (Hash merge, Shuffle, Sort, Partition)

**Grouped**

<table>
<thead>
<tr>
<th>k1:v,v,v,v</th>
<th>k2:v</th>
<th>k3:v,v</th>
<th>k4:v,v,v</th>
<th>k5:v</th>
</tr>
</thead>
</table>

**Reduce:**
Collect all values belonging to the key and output

| R | R | R | R | R |

**Output**

Map-Reduce: In Parallel

All phases are distributed with many tasks doing the work
MapReduce: Environment

MapReduce environment takes care of:

- **Partitioning** the input data
- **Scheduling** the program’s execution across a set of machines
- Performing the **group by key** step
  - In practice this is the bottleneck
- Handling machine **failures**
- Managing required inter-machine **communication**
Map worker failure

- Map tasks completed or in-progress at worker are reset to idle and rescheduled
- Reduce workers are notified when map task is rescheduled on another worker

Reduce worker failure

- Only in-progress tasks are reset to idle and the reduce task is restarted
Spark: Extends MapReduce
MapReduce:
- Incurs substantial overheads due to data replication, disk I/O, and serialization
Two major limitations of MapReduce:

- Difficulty of programming directly in MapReduce
  - Many problems aren’t easily described as map-reduce
- Performance bottlenecks, or batch not fitting the use cases
  - Persistence to disk typically slower than in-memory work

In short, MapReduce doesn’t compose well for large applications

- Many times, one needs to chain multiple map-reduce steps.
MapReduce uses two “ranks” of tasks:
One for **Map** the second for **Reduce**
- Data flows from the first rank to the second

**Data-Flow Systems** generalize this in two ways:
1. Allow any number of tasks/ranks
2. Allow functions other than Map and Reduce
- As long as data flow is in one direction only, we can have the blocking property and allow recovery of tasks rather than whole jobs
Data Analytics Software Stack

- **Spark**
  - **Streaming**
    - Stream processing
  - **GraphX**
    - Graph computation
  - **MLlib**
    - User-friendly machine learning
  - **SparkSQL**
    - SQL API

- **Spark**
  - Fast memory-optimized execution engine (Python/Java/Scala APIs)

- **Tachyon**
  - Distributed Memory-Centric Storage System

- **Hadoop Distributed File System (HDFS)**

- **Mesos**
  - Cluster resource manager, multi-tenancy

- **Hive**
- **Storm**
- **MPI**
- **Hadoop MR**
Spark: Most Popular Data-Flow System

- Expressive computing system, not limited to the map-reduce model

- Additions to MapReduce model:
  - Fast data sharing
    - Avoids saving intermediate results to disk
    - Caches data for repetitive queries (e.g. for machine learning)
  - General execution graphs (DAGs)
  - Richer functions than just map and reduce

- Compatible with Hadoop
Spark: Overview

- Open source software (Apache Foundation)
- Supports Java, Scala and Python

- **Key construct/idea:** Resilient Distributed Dataset (RDD)

- **Higher-level APIs:** DataFrames & DataSets
  - Introduced in more recent versions of Spark
  - Different APIs for aggregate data, which allowed to introduce SQL support
Key concept **Resilient Distributed Dataset** (RDD)

- Partitioned collection of records
  - Generalizes (key-value) pairs
- Spread across the cluster, Read-only
- Caching dataset in memory
  - Different storage levels available
  - Fallback to disk possible
- RDDs can be created from Hadoop, or by transforming other RDDs (you can stack RDDs)
- RDDs are best suited for applications that apply the same operation to all elements of a dataset
Transformations build RDDs through deterministic operations on other RDDs:
- Transformations include *map, filter, join, union, intersection, distinct*
- **Lazy evaluation:** Nothing computed until an action requires it

**Actions** to return value or export data
- Actions include *count, collect, reduce, save*
- Actions can be applied to RDDs; actions force calculations and return values
Task Scheduler: General DAGs

- Supports general task graphs
- Pipelines functions where possible
- Cache-aware data reuse & locality
- Partitioning-aware to avoid shuffles

- Stage 1
  - A: join
  - B: groupBy

- Stage 2
  - C: map
  - D: filter

- Stage 3
  - E: join
  - F: = cached partition

= RDD

DataFrame & Dataset

- **DataFrame:**
  - Unlike an RDD, data organized into named columns, e.g. a **table in a relational database**.
  - Imposes a structure onto a distributed collection of data, allowing higher-level abstraction.

- **Dataset:**
  - Extension of DataFrame API which provides **type-safe, object-oriented programming interface** (compile-time error detection).

Both built on Spark SQL engine. Both can be converted back to an RDD.
Useful Libraries for Spark

- Spark SQL
- Spark Streaming – stream processing of live datastreams
- MLlib – scalable machine learning
- GraphX – graph manipulation
  - Extends Spark RDD with Graph abstraction: a directed multigraph with properties attached to each vertex and edge
Spark vs. Hadoop MapReduce

- **Performance:** *Spark normally faster* but with caveats
  - Spark can process data in-memory; Hadoop MapReduce persists back to the disk after a map or reduce action
  - Spark generally outperforms MapReduce, but it *often needs lots of memory to perform well*; if there are other resource-demanding services or can’t fit in memory, Spark degrades
  - MapReduce easily runs alongside other services with minor performance differences, & works well with the 1-pass jobs it was designed for

- **Ease of use:** *Spark is easier to program* (higher-level APIs)

- **Data processing:** *Spark more general*
Problems Suited for MapReduce
Example: Host size

- Suppose we have a large web corpus
- Look at the metadata file
  - Lines of the form: (URL, size, date, ...)
- For each host, find the total number of bytes
  - That is, the sum of the page sizes for all URLs from that particular host

- Other examples:
  - Link analysis and graph processing
  - Machine Learning algorithms
Statistical machine translation:
- Need to count number of times every 5-word sequence occurs in a large corpus of documents

Very easy with MapReduce:
- Map:
  - Extract (5-word sequence, count) from document
- Reduce:
  - Combine the counts
Compute the natural join $R(A,B) \bowtie S(B,C)$
- $R$ and $S$ are each stored in files
- Tuples are pairs $(a,b)$ or $(b,c)$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td>$b_1$</td>
</tr>
<tr>
<td>$a_2$</td>
<td>$b_1$</td>
</tr>
<tr>
<td>$a_3$</td>
<td>$b_2$</td>
</tr>
<tr>
<td>$a_4$</td>
<td>$b_3$</td>
</tr>
</tbody>
</table>

$R$  \hspace{2cm} \bowtie \hspace{2cm} S

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_2$</td>
<td>$c_1$</td>
</tr>
<tr>
<td>$b_2$</td>
<td>$c_2$</td>
</tr>
<tr>
<td>$b_3$</td>
<td>$c_3$</td>
</tr>
</tbody>
</table>

$S$

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_3$</td>
<td>$c_1$</td>
</tr>
<tr>
<td>$a_3$</td>
<td>$c_2$</td>
</tr>
<tr>
<td>$a_4$</td>
<td>$c_3$</td>
</tr>
</tbody>
</table>
Map-Reduce Join

- Use a hash function \( h \) from B-values to 1...\( k \)
- A Map process turns:
  - Each input tuple \( R(a,b) \) into key-value pair \( (b,(a,R)) \)
  - Each input tuple \( S(b,c) \) into \( (b,(c,S)) \)

- Map processes send each key-value pair with key \( b \) to Reduce process \( h(b) \)
  - Hadoop does this automatically; just tell it what \( k \) is.
- Each Reduce process matches all the pairs \( (b,(a,R)) \) with all \( (b,(c,S)) \) and outputs \( (a,b,c) \).
Problems NOT suitable for MapReduce

- **MapReduce is great for:**
  - Problems that require sequential data access
  - Large batch jobs (not interactive, real-time)

- **MapReduce is inefficient for problems where random (or irregular) access to data required:**
  - Graphs
  - Interdependent data
    - Machine learning
    - Comparisons of many pairs of items
In MapReduce we quantify the cost of an algorithm using

1. **Communication cost** = total I/O of all processes
2. **Elapsed communication cost** = max of I/O along any path
3. **(Elapsed) computation cost** analogous, but count only running time of processes

Note that here the big-O notation is not the most useful (adding more machines is always an option)
Example: Cost Measures

- For a map-reduce algorithm:
  - Communication cost = input file size + 2 × (sum of the sizes of all files passed from Map processes to Reduce processes) + the sum of the output sizes of the Reduce processes.
  - Elapsed communication cost is the sum of the largest input + output for any map process, plus the same for any reduce process.

What Cost Measures Mean

- Either the I/O (communication) or processing (computation) cost dominates
  - Ignore one or the other

- Total cost tells what you pay in rent from your friendly neighborhood cloud

- Elapsed cost is wall-clock time using parallelism
Cost of Map-Reduce Join

- **Total communication cost**
  \[ = O(|R| + |S| + |R \bowtie S|) \]
- **Elapsed communication cost**
  \[ = O(s) \]

- We’re going to pick \( k \) and the number of Map processes so that the I/O limit \( s \) is respected.
- We put a limit \( s \) on the amount of input or output that any one process can have. \( s \) could be:
  - What fits in main memory
  - What fits on local disk

- With proper indexes, computation cost is linear in the input + output size.
  - So, computation cost is like communication cost