Big Data and Cloud Computing, 23/24 (Part 2)

Inês Dutra and Eduardo Marques
DCC-FCUP
ines@dcc.fc.up.pt (room: 1.31)

April 10th, 2024

Data Mining and Machine Learning: recap

- Learning?
  - "An agent **learns** if it improves its performance in future tasks after making observations about the past or current world." (Tom Mitchel)

Machine Learning: very brief overview

- Learning?
  - Given observations O,
    described by features $f_1, f_2, \ldots, f_n$,
    the task of a machine learning algorithm is:
    - to find patterns based on features $f_1, f_2, \ldots, f_n$ (all or some of them), that distinguish among different groups of observations OR
    - to find a function that will **predict** new observations

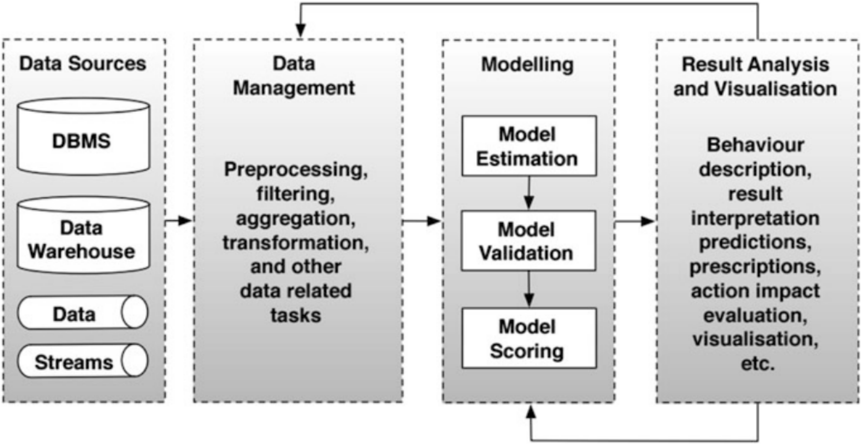
Machine Learning: very brief overview

- Learning?
  - Can be **supervised**:
    - Given features $f_1, f_2, \ldots, f_n$,
      and a special feature, the **target** variable (ground truth),
      find a model that can **predict** the target variable for **new** observations that are described by features $f_1, f_2, \ldots, f_n$
    - The supervised learning task can be **classification** or **regression**
  - Can be **unsupervised**: find subgroups of patterns, no target variable is known or provided
    - clustering
    - association rules
  - Other learning methods: reinforcement learning, matrix factorization for recommender systems
  - *background/prior knowledge*: description of observations, necessary to improve the learning

Data Mining and Machine Learning: recap

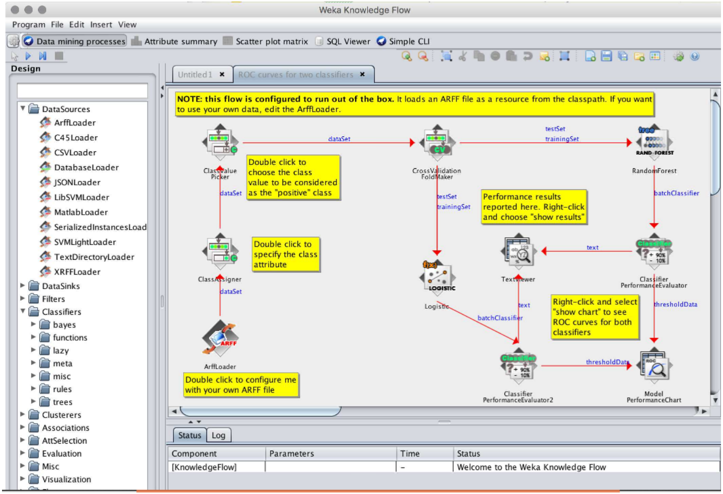
- Workflow (Dataflow - Knowledgeflow):
  - Data preprocessing
    - transformation: normalization, standardization, averaging, median, denoising, filtering
    - preparation: depends on the task, algorithm, package or library being used
  - Machine learning task, algorithm
  - Validation: cross-validation, bootstrapping
- Workflow tools: WEKA KnowledgeFlow, RapidMiner, Orange3, Taverna, Condor DAGMan, Pegasus, Google Dataflow, Google Composer (Apache Airflow)

Data Mining and Machine Learning: workflow



Assunção, M.D., Calheiros, R.N., Bianchi, S., Netto, M.A.S., Buyya, R.: Big data computing and clouds: trends and future directions. J. Parallel Distrib. Comput. 79–80, 3–15 (2015)

Example of workflow in WEKA



java -jar weka.jar ⇒ KnowledgeFlow

Example of workflow with Orange3

(installation needed, go to https://orange.biolab.si/)

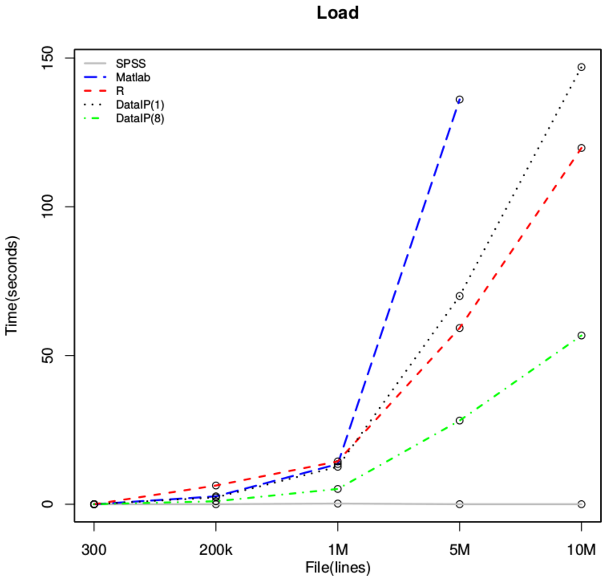
Limitations

- Most systems and tools for data analysis are not scalable
- I/O, memory, computing power

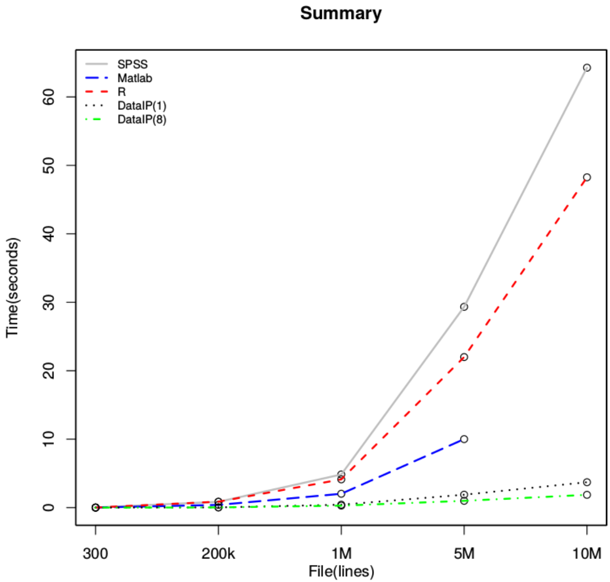
Scalability

- Computational resources: memory, CPU, I/O, storage
- I/O:
  - Experiment 1: SPSS, MatLab, R and DataIP (in-house implementation)
    - dataset of patients, originally 200K entries, 6 numeric variables without nulls
    - varying sizes: 300, 200k, 1M, 5M, 10M
  - Experiment 2: job that needs to fetch data files from a remote site

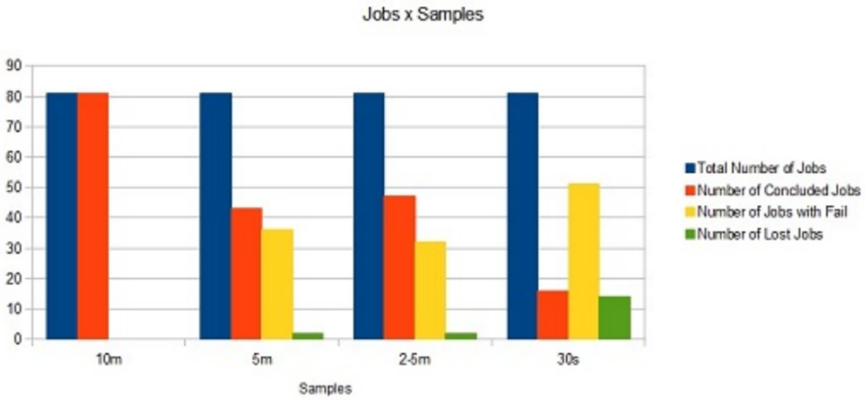
Scalability: Experiment 1, I/O



Scalability: Experiment 1, simple computing: summary



Scalability: Experiment 2, file transfer



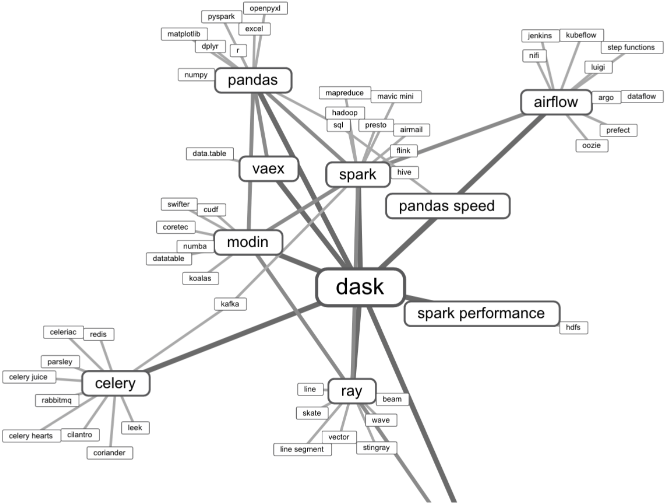
Scalability

- Alternatives
  - break file in multiple smaller files that can be read in parallel: useful if the reading can be done in parallel
  - undersampling: need to be careful about data distribution
  - use of specific hardware and software: distributed disks, distributed file systems, distributed databases, in-memory databases, parallel and distributed software
  - work with compressed files: zip, parquet, CSR, CSR5 (for sparse matrices) etc

Scalability

- Python libraries we will be studying:
  - Dask
  - Modin
  - Vaex
  - Koalas
  - Ray
  - ...

Alternative libraries for big data



source: `https://www.datarevenue.com/en-blog/pandas-vs-dask-vs-vaex-vs-modin-vs-rapids-vs-ray`

Differences in libraries

Function	Pandas/Modin/Koala	Vaex	Dask DataFrame	Turicreate	H2O	PySpark	Datatable
Read file	pd.read_csv() pd.read_parquet()	vaex.read_csv() vaex.open()	dd.read_csv() dd.read_parquet()	tc.read_csv() tc.SFrame()	h20. upload_file() h2o.import_file()	sqlContext.read.csv() sqlContext.read.parquet()	dt.fread() dt.open()
Count	len(df)	len(df)	len(df)	len(df)	len(df)	df.count()	df.shape[0]
Mean	df.x.mean()	df.x.mean()	df.x.mean() .compute()	df['x'].mean() tc.Sketch(df['x']).mean()	df['x'].mean()	df.select(f.mean('x')) .collect()	df[:, dt.mean(dt.f.x)]
Standard deviation	df.x.std()	df.x.std()	df.x.std() .compute()	df['x'].std() tc.Sketch(df['x']).std()	df['x'].sd()	df.select(f.stddev('x')) .collect()	df[:, dt.sd(dt.f.x)]
Sum columns	df['x']+df['y'] df.x + df.y	df['x']+df['y'] df.x + df.y	df['x']+df['y'] df.x + df.y	df['x']+df['y']	df['x']+df['y']	df['x']+df['y']	df[:, f.x + f.y]
Sum columns mean	(df.x + df.y).mean()	(df.x + df.y).mean()	(df.x + df.y).mean() .compute()	(df['x'] + df['y']) .mean()	(df['x'] + df['y']) .mean()	df.select(f.mean(df['x'] + df['y'])) .collect()	df[:, dt.mean (f.x + f.y)]
Value counts	df.x.value_counts()	df.x.value_counts()	df.x.value_counts() .compute()	df['x'].value_counts()	df['x'].table()	df.select('x').distinct() .collect()	df[:,dt.count(f.x),'x']
Group-by	df.groupby(by='z') .agg({ 'x': ['mean', 'std'], 'y': ['mean', 'std']})	df.groupby(by='z') .agg({ 'x': ['mean', 'std'], 'y': ['mean', 'std']})	df.groupby(by='z') .agg({ 'x': ['mean', 'std'], 'y': ['mean', 'std']}) .compute()	df.groupby('z', operations={ 'c1':tc.aggregate.MEAN('x'), 'c2':tc.aggregate.STD('x'), 'c3':tc.aggregate.MEAN('y'), 'c4':tc.aggregate.STD('y')})	df.group_by('z') .mean(col = ['x', 'y']) .sd(col = ['x', 'y']) .get_frame()	df.groupby('z') .agg(f.mean('x'), f.stddev('x'), f.mean('y'), f.stddev('y'))	aggs = { 'c1': dt.mean(f.x), 'c2': dt.sd(f.x), 'c3': dt.mean(f.y), 'c4': dt.sd(f.y),} df[:, aggs, dt.by(f.z)]
Join	df.join(other, on = 'key') pd.merge(df, other)	df.join(other, on = 'key')	dd.merge(df, other)	df.join(other, on = 'key')	df.merge(other)	df.join(other, on = 'key')	other.key = 'key' df[:,:,dt.join(other)]

source: `https://towardsdatascience.com/`

`beyond-pandas-spark-dask-vaex-and-other-big-data-technologies-battling-head-to-head-a453a1f8cc13`

Scalability

Let's start with dataflows...

Apache Beam

- Provides a place to run Apache Beam jobs on the GCP
- Offers the ability to create jobs based on **templates**
- No need to address common aspects of running jobs on a cluster:
  - load balancing
  - scaling number of workers for a job
- These tasks are done automatically for both **batch** and **streaming**

Apache BEAM – Batch + strEAM

- Evolution of Google Dataflow that separates the dataflow logic from the programming issues (language, runners etc)
- Unified model for both batch and stream data processing
- Programs can be executed in different processing frameworks (via runners) using a set of different IOs

Apache BEAM – Batch + strEAM

Why to use BEAM instead of only Hadoop, Spark, Flink, GCP Dataflow etc?

→ The Apache Beam framework provides an abstraction between your application logic and the big data ecosystem

Apache BEAM – Batch + strEAM

In the BEAM ecosystem:

- **DataSource**: can be batches, micro-batches or streaming data
- **SDK**: Java or Python
- **Runner**: Apache Spark, Apache Flink, Google Cloud Dataflow, among others and DirectRunner (runs locally)
- To build a BEAM logic: Pipeline, PCollection, PTransform, ParDO and DoFn

Apache BEAM – Batch + strEAM: Components

- **Pipeline**: encapsulates the workflow of your entire data processing tasks from start to finish. Includes:
  - reading input data
  - transforming that data (almost all data transformations are supported including database operations)
  - writing output data
- All Beam driver programs must create a Pipeline
- When you create the Pipeline, you must also specify the execution options that tell the Pipeline where and how to run
- Beam can run independently of the Google Cloud Platform

Apache BEAM – Batch + strEAM: Components

- **PCollection**: distributed data set that your Beam pipeline operates on
- data may come from a fixed source like a file, or from a continuously updating source via a subscription or other mechanism

Apache BEAM – Batch + strEAM: Components

- **PTransform**: represents a data processing operation, or a step, in your pipeline
- Every PTransform takes one or more PCollection objects as input, performs a processing function that you provide on the elements of that PCollection, and produces zero or more output PCollection objects.

Apache BEAM – Batch + strEAM: Components

- **ParDo**: for generic parallel processing
- similar to the “Map” phase of a Map/Shuffle/Reduce-style algorithm
- a ParDo transform considers each element in the input PCollection, performs some processing function (your user code) on that element, and emits zero, one, or multiple elements to an output PCollection.

Apache BEAM – Batch + strEAM: Components

- **DoFn**: applies your logic in each element in the input PCollection and lets you populate the elements of an output PCollection
- to be included in your pipeline, it's wrapped in a ParDo PTransform.