732A54/TDDE31 Big Data Analytics

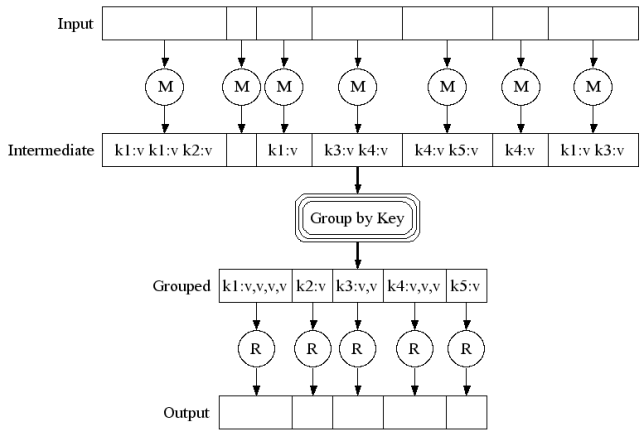
Lecture 9: Machine Learning with Spark

Mohammad Seidpisheh
IDA, Linköping University, Sweden

Contents

Spark Framework

- Recall from the previous lecture that MapReduce can emulate any distributed computation, since this can be divided into a sequence of MapReduce calls.



- However, the emulation may be **inefficient** since the message exchange relies on external storage, e.g. disk.
- This is a problem for iterative machine learning algorithms. **Even worse**: Each iteration (i.e., MapReduce call) loads the data anew from disk.
- Apache Spark is a framework to process large amounts of data by parallelizing computations across a cluster of nodes.

Spark Framework

- Spark builds on MapReduce's ability to emulate any distributed computation but it makes it more **efficient** by emulating in-memory data sharing across MapReduce calls.
- The main difference between Spark and MapReduce is that Spark processes data in memory, whereas MapReduce processes data on disk.
- Apache Spark has a similar programming model to MapReduce but extends it with a data-sharing abstraction called "resilient distributed datasets", or RDDs.
- Data sharing is achieved via RDDs, a distributed memory abstraction that allows programmers to perform in-memory computations, and keeping data in memory improves the performance of algorithms
- Splitting the MapeReduce construct into simpler operations
  - Transformations
  - Actions

Spark Framework

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

- Table 2 lists the main RDD transformations and actions available in Spark.
- Transformations are lazy operations that define a new RDD, while actions launch a computation to return a value to the program or write data to external storage.

Spark Framework

- Resilient Distributed Datasets (RDDs) are the primary data structure in Spark.
- RDD is a read-only, partitioned collection of data points that can only be **defined** through transformations applied to external storage or to other RDDs.
- The sequence of transformations that defines a RDD is called its lineage. It is used to rebuild it in case of failure, i.e. there is no data replication unlike in MapReduce.
- Actually, RDDs are **created each time an action is executed, unless the user persist them** in memory and/or disk.
- RDDs are reliable and memory-efficient when it comes to parallel processing.
- By storing and processing data in RDDs, Spark speeds up MapReduce processes.

Spark Framework

- A large website is experiencing errors and an operator wants to search terabytes of logs containing word "HDFS" to find time of errors and the cause.
- Using Spark, the operator can load just the error messages from the logs into RAM across a set of nodes and query them interactively.
- Operator would first write the following Scala code:

```
1.lines=spark.textFile("hdfs://...")
2.errors=lines.filter(_.startsWith("ERROR"))
3.errors.persist() //Store in memory
4.errors.count() //Materialize
5.errors.filter(_.contains("HDFS")).map(_.split('\t')(3)).collect()
```

- Note that:
  - Line 1 defines an RDD backed by an HDFS file (as a collection of lines of text)
  - Line 2 derives a filtered RDD from it, keeping the lines started with an error
  - Line 3 asks to store the error lines **in memory**. Note `persist()` = `persist(MEMORY_ONLY)` = `cache()` $\neq$ `persist(MEMORY_AND_DISK)` $\neq \ldots$
  - However, this does not happen until line 4, when the RDDs are computed.
  - The rest of the RDDs (e.g., `lines`) are discarded after being used. In fact, First RDD, entire lines, is never cached.
  - Line 5 does not access disk because the data are **in memory** and collect the time of errors containing "HDFS".

Machine Learning with Spark: Logistic Regression

- Many machine learning algorithms are iterative in nature because they run iterative optimization procedures, such as gradient descent, to optimize an objective function, e.g. logistic regression.
- These algorithms can be sped up substantially if their working set fits into RAM across a cluster.
- Furthermore, these algorithms often employ bulk operations like maps and sums, making them easy to express with RDDs.
- Logistic regression is a common classification algorithm that searches for a hyperplane that best separates two sets of data points e.g., spam and non-spam emails.

Machine Learning with Spark: Logistic Regression

- Consider a binary classification problem, i.e. $t \in \{-1, +1\}$. Then,

$$p(t = +1|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|t = +1)p(t = +1)}{p(\boldsymbol{x}|t = +1)p(t = +1) + p(\boldsymbol{x}|t = -1)p(t = -1)} = \sigma(s(\boldsymbol{x}))$$

- where $s(\boldsymbol{x}) = \log \frac{p(\boldsymbol{x}|t=+1)p(t=+1)}{p(\boldsymbol{x}|t=-1)p(t=-1)}$,
- and $\sigma(a) = \frac{1}{1+exp(-a)}$ is called logistic sigmoid function.
- We assume that $p(\boldsymbol{x}|t)$ is a member of the exponential family with equal scale parameter (e.g. Gaussian with equal covariance matrix, multinomial),
- This assumption implies that $s(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$.
- The model $y(\boldsymbol{x}) = p(t = +1|\boldsymbol{x}) = \sigma(\boldsymbol{w}^T\boldsymbol{x})$ is called logistic regression.

Machine Learning with Spark: Logistic Regression

- Note that

$$p(t=+1|\boldsymbol{x}) = \sigma(s(\boldsymbol{x})) = \frac{1}{1+\exp(-\boldsymbol{w}^T\boldsymbol{x})}$$

$$p(t=-1|\boldsymbol{x}) = 1-p(t=+1|\boldsymbol{x}) = 1-\sigma(s(\boldsymbol{x})) = 1-\frac{1}{1+\exp(-\boldsymbol{w}^T\boldsymbol{x})} = \frac{1}{1+\exp(\boldsymbol{w}^T\boldsymbol{x})}$$

therefore

$$p(t=-1|\boldsymbol{x}) = \frac{1}{1+\exp(\boldsymbol{w}^T\boldsymbol{x})}$$

and thus

$$p(t=t_n|\boldsymbol{x}_n) = \frac{1}{1+\exp(-t_n\boldsymbol{w}^T\boldsymbol{x}_n)}.$$

- We determine the parameters $\boldsymbol{w}$ by minimizing the negative log-likelihood:

$$L(\boldsymbol{w}) = -\sum_n \log p(t_n|\boldsymbol{x}_n) = \sum_n \log(1+\exp(-t_n\boldsymbol{w}^T\boldsymbol{x}_n))$$

whose gradient is $\sum_n t_n((1/(1+\exp(-t_n\boldsymbol{w}^T\boldsymbol{x}_n)))-1)\boldsymbol{x}_n$.

Machine Learning with Spark: Logistic Regression

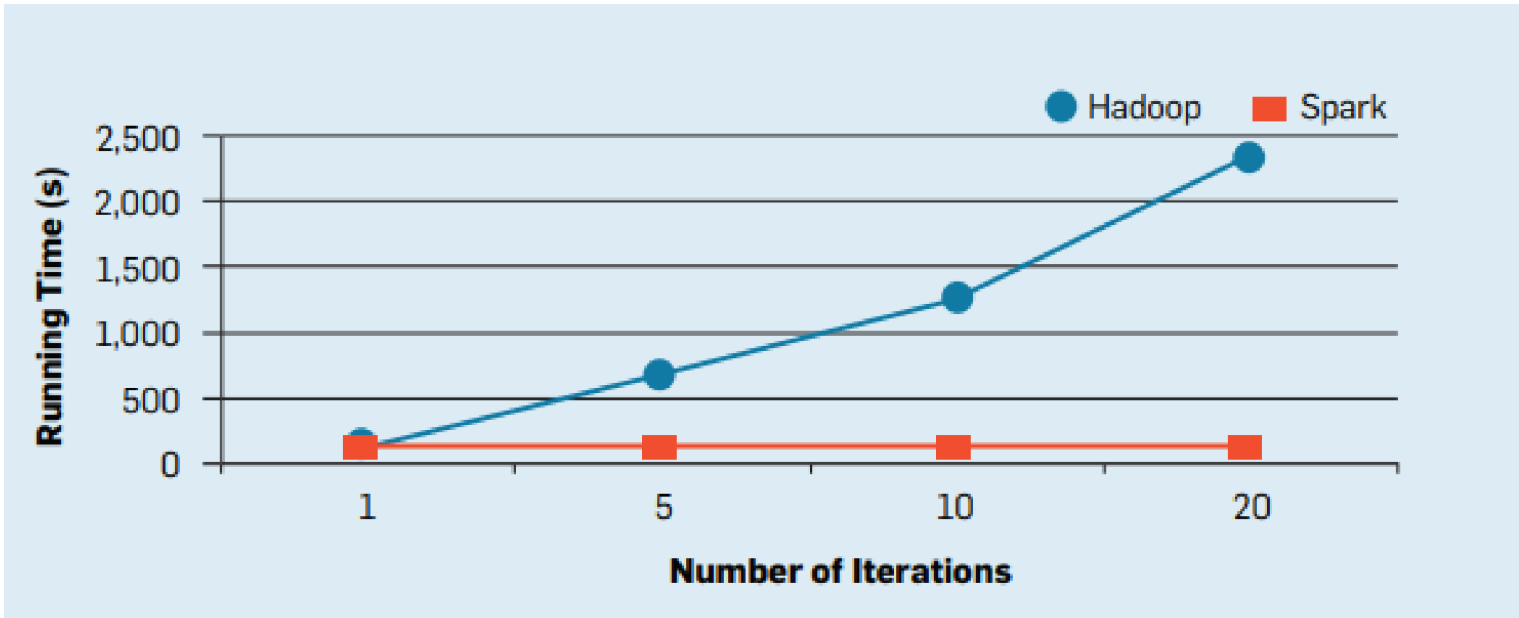
- Logistic regression in Scala (note the use of **persist**, map and reduce):

```
1   //   Load data into an RDD
2   val points = sc.textFile(...).map(readPoint).persist()
3   // Start with a random parameter vector
4   var w = DenseVector.random(D)
5   //For loop: On each iteration, update param with sum
6   for (i <- 1 to ITERATIONS) {
7   val gradient = points.map { p =>
8   t * (1/(1+exp(-t*(w.dot(x))))-1) * x
9    }.reduce((a, b) => a+b)
10   w -= gradient
11  }
12
```

- Defining a cached RDD called points as the result of a map transformation on a text file that parses each line of text into a data Point.
- Then repeatedly run map and reduce on data points to compute the gradient at each step by summing a function of the current w.
- It may be reused the RDD in an iterative algorithm, so cache it in memory with persist() to avoid re-evaluating it every time.

Machine Learning with Spark: Logistic Regression

- Spark runs faster than traditional MapReduce, since it make it easy to load data into RAM once and run multiple sums.



- Unlike Spark, the running time of MapReduce increased significantly, as the number of iterations increased.
- MapReduce takes more than 2000 seconds for 20 iterations because each iteration loads the data from disk, while Spark takes only 20 seconds for 100GB data.

Machine Learning with Spark: Logistic Regression

- Logistic regression in Python

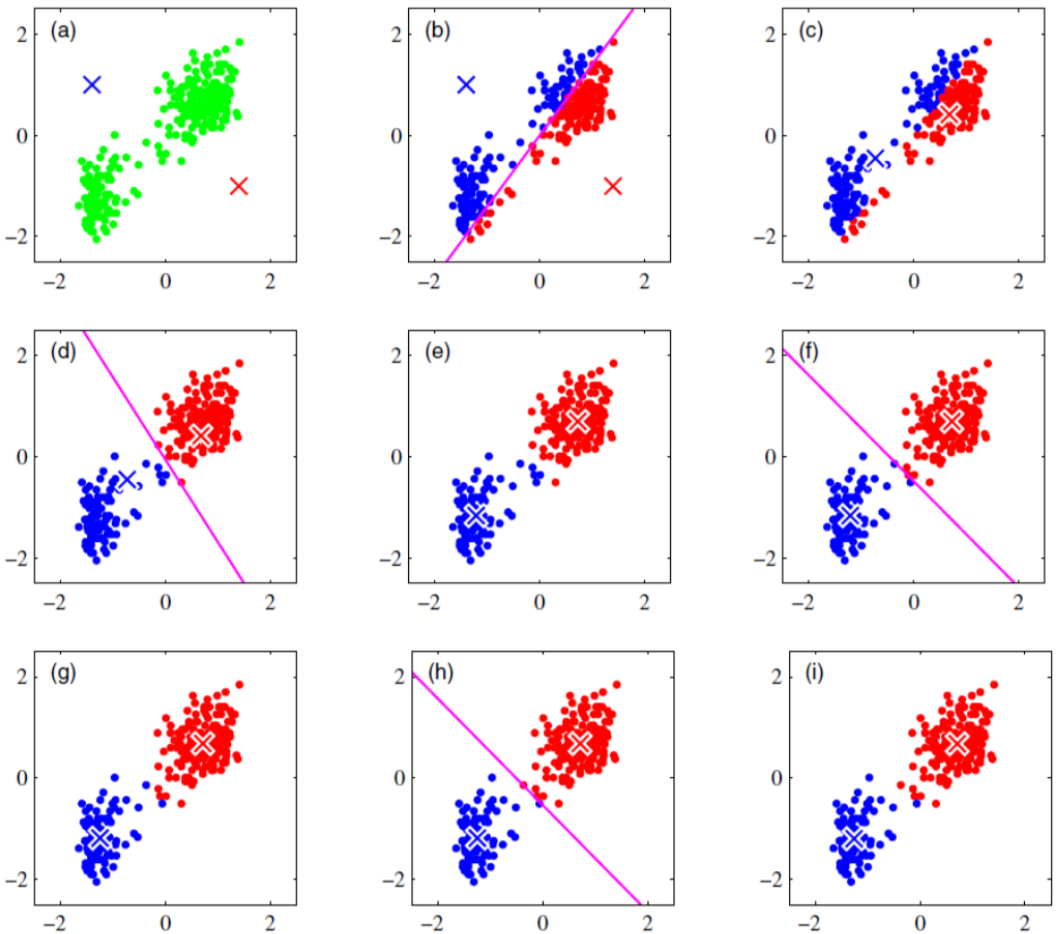
```
# Initialize w to a random value in the interval (-1, 1)
w = 2 * np.random.ranf(size=D) - 1 # D is Number of dimensions
print("Initial w: " + str(w))
# Compute logistic regression gradient for a matrix of data points
def gradient(matrix: np.ndarray, w: np.ndarray) -> np.ndarray:
        t = matrix[:, 0]    # point labels (first column of input file)
        X = matrix[:, 1:]   # point coordinates
# For each point (x, y), compute gradient function, then sum these up
        return ((1.0 / (1.0 + np.exp(-t * X.dot(w))) - 1.0) * t * X.T).sum(1)
def add(x: np.ndarray, y: np.ndarray) -> np.ndarray:
        x += y
        return x

    #  On each iteration of loop, update W with sum
for i in range(iterations):
        print("On iteration %i" % (i + 1))
        w -= points.map(lambda m: gradient(m, w)).reduce(add)
print("Final w: " + str(w))
```

- It uses batch gradient descent, a simple iterative algorithm that computes a gradient function over the data repeatedly as a parallel sum.
- A lambda expression is a special syntax for creating a function and passing it to another function all on one line of code. (All functions in Python can be passed as an argument to another function)

Machine Learning with Spark: K-Means

- Consider data clustering (a.k.a. unsupervised learning) via the K-means algorithm.

1	Set K points as centroids at random
2	Assign each point to a cluster with the closest centroid
3	recalculate the cluster centroids as the averages of the points assigned to each cluster
	Repeat steps 2 and 3 until the centroids do not change



Machine Learning with Spark: K-Means

- K-Means in Python

```
def closestPoint(p, centers):
    bestIndex = 0
    closest = float("+inf")
    for i in range(len(centers)):
        tempDist = np.sum((p - centers[i]) ** 2)
        if tempDist < closest:
            closest = tempDist
            bestIndex = i
    return bestIndex
kPoints = data.takeSample(False, K, 1)
tempDist = 1.0

while tempDist > convergeDist:
    closest = data.map(
        lambda p: (closestPoint(p, kPoints), (p, 1)))
    pointStats = closest.reduceByKey(
        lambda p1_c1, p2_c2: (p1_c1[0] + p2_c2[0], p1_c1[1] + p2_c2[1]))
    newPoints = pointStats.map(
        lambda st: (st[0], st[1][0] / st[1][1])).collect()

    tempDist = sum(np.sum((kPoints[iK] - p) ** 2) for (iK, p) in newPoints)

    for (iK, p) in newPoints:
        kPoints[iK] = p

print("Final centers: " + str(kPoints))
```

- The takeSample: generate the initial centroids with a random sampling from the RDD (step 1 in K-Means)
- The map transformation using closestPoint function: Assign the closest centroid to that data point. (step 2 in K-Means)
- The reduceByKey: sum over data points with associated same cluster. (step 2 in K-Means)
- The collect: recalculate cluster centroids. (step 3 in K-Means)

Machine Learning with Spark: MLlib

- Apache Spark includes MLlib, a library for machine learning that uses linear algebra libraries on each node
- Many machine learning methods are already implemented in MLlib, i.e. the user does not need to specify the transformations and actions.
- Logistic regression in Python:

```
lr = LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)
lrModel = lr.fit(training)
```

- SVMs in Python:

```
model = SVMWithSGD.train(parsedData, iterations=100)
```

- NNs in Python:

```
layers = [4, 5, 4, 3]
trainer = MultilayerPerceptronClassifier(maxIter=100, layers=layers,
blockSize=128, seed=1234)
model = trainer.fit(train)
```

- MMs in Python:

```
gmm = GaussianMixture().setK(2)
model = gmm.fit(dataset)
```

- K-Means in Python:

```
kmeans = KMeans().setK(2).setSeed(1)
model = kmeans.fit(dataset)
```

Lab with Spark

- Implement a kernel model to predict the hourly temperatures for a date and place in Sweden. To do so, you are provided with the files `stations.csv` and `temperature-readings.csv` (available at `https://www.ida.liu.se/ 732A54/lab/data.zip` or /software/sse/manual/spark/BDA_demo/input_data folder on Sigma). These files contain information about weather stations and temperature measurements for the stations at different days and times. The data have been kindly provided by the Swedish Meteorological and Hydrological Institute (SMHI) and processed by Zlatan Dragisic.
- You are asked to provide a temperature forecast for a date and place in Sweden. The forecast should consist of the predicted temperatures from 4 am to 24 pm in an interval of 2 hours. Use a kernel that is the sum of three Gaussian kernels:
  - The first to account for the distance from a station to the point of interest.
  - The second to account for the distance between the day a temperature measurement was made and the day of interest.
  - The third to account for the distance between the hour of the day a temperature measurement was made and the hour of interest.
- Repeat the exercise about multiplying instead of summing the three kernels above.

Lab with Spark

- The best regression function under the squared error loss function is $y^*(\boldsymbol{x}) = \mathbb{E}_Y[y|\boldsymbol{x}]$.
- Since $\boldsymbol{x}$ may not appear in the finite training set $\{(\boldsymbol{x}_n, y_n)\}$ available, then we output a weighted average over all the training points. That is

$$y(\boldsymbol{x}) = \frac{\sum_n k\left(\frac{\boldsymbol{x}-\boldsymbol{x}_n}{h}\right) y_n}{\sum_n k\left(\frac{\boldsymbol{x}-\boldsymbol{x}_n}{h}\right)}$$

where $k : \mathbb{R}^D \to \mathbb{R}$ is a kernel function, which is usually non-negative and monotone decreasing along rays starting from the origin. The parameter h is called smoothing factor or width.

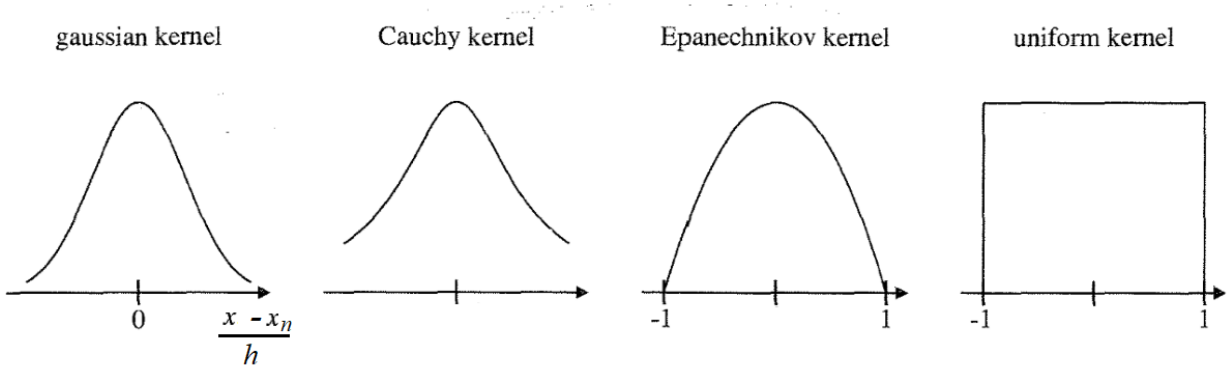


FIGURE 10.3. *Various kernels on $\mathcal{R}$.*

- Gaussian kernel: $k(u) = exp(-||u||^2)$ where $||\cdot||$ is the Euclidean norm.

Lab with Spark

- Repeat the exercise and implement at least two MLlib Library models to forecast temperature instead of the three kernels.
- Instead, broadcast one of the RDDs to join, if small. This sends a copy of the RDD to each node, and the join can be performed locally (or even skipped).

```
rdd = rdd.collectAsMap()
bc = sc.broadcast(rdd)
bc.value[i]
```

Summary

- Spark is a framework to process large datasets by parallelizing computations.
- It is particularly suitable for iterative distributed computations, since data can be store in memory.
- It includes MLlib, a machine learning library.

Literature

- Main sources
  - Zaharia, M. et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, 15-28, 2012.
  - Meng, X. et al. MLlib: Machine Learning in Apache Spark. *Journal of Machine Learning Research*, 17(34):1-7, 2016.
- Additional sources
  - Zaharia, M. et al. Apache Spark: A Unified Engine for Big Data Processing. *Communications of the ACM*, 59(11):56-65, 2016.
  - Spark programming guide available at `https://spark.apache.org/docs/latest/rdd-programming-guide.html`
  - MLlib manual available at `http://spark.apache.org/docs/latest/ml-guide.html`
  - Slides for 732A99/TDDE01 Machine Learning.