732A54 Big Data Analytics

Lecture 11: Machine Learning with Spark

Jose M. Peña
IDA, Linköping University, Sweden

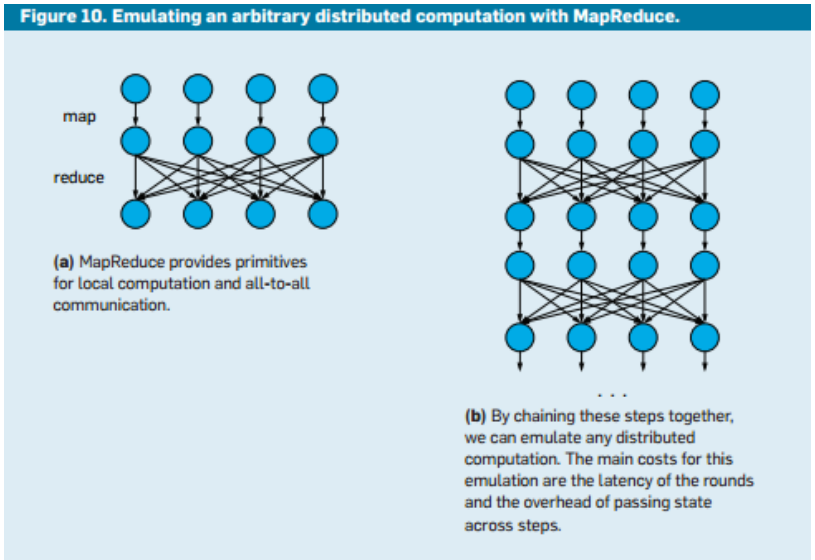
Contents

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):17, 2016.
  - MLlib manual available at `http://spark.apache.org/docs/latest/ml-guide.html`
- Additional sources
  - Zaharia, M. et al. Apache Spark: A Unified Engine for Big Data Processing. *Communications of the ACM*, 59(11):56-65, 2016.
  - Slides for 732A95 Introduction to Machine Learning.

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 major problem for iterative machine learning algorithms.
- Apache Spark is a framework to process large amounts of data by parallelizing computations across a cluster of nodes.
- It builds on MapReduce's ability to emulate any distributed computation but makes it more efficiently by emulating in-memory data sharing across MapReduce calls.
- It includes MLlib, a library for machine learning that uses linear algebra libraries on each node.

Spark Framework

- Data sharing is achieved via resilient distributed datasets (RDDs).
- RDD is a read-only, partitioned collection of records that can only be created through transformations applied to external storage or to other RDDs.

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.

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.

Spark Framework

- Data sharing is achieved via resilient distributed datasets (RDDs).
- RDD is a read-only, partitioned collection of records that can only be created through transformations applied to external storage or to other RDDs.

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.

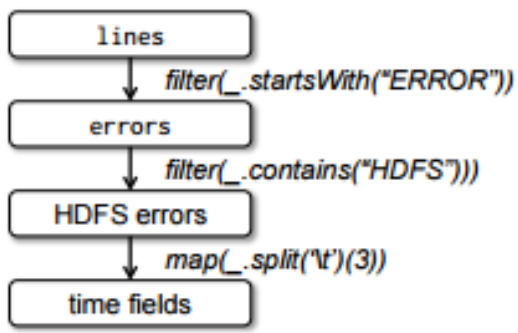
- The sequence of transformations that creates a RDD is called its lineage. It is used to rebuild it in case of failure.
- Users can indicate which RDDs to store in memory, e.g. because they will be reused.
- RDDs do not materialize (and are stored in memory) until an action is executed.

Spark Framework

- Example in Scala to find error lines in a log file:

```
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 3 indicates to store the error lines in memory.
  - However, this does not happen until line 4, when the RDDs materialize.
  - The rest of the RDDs are discarded after being used.
  - Line 5 does not access disk because the data are in memory.
  - If any partition of the in-memory data has gone lost, it can be rebuilt with the help of the lineage graph.



Spark Framework

- The lineage graph is also used by the master to schedule jobs similarly to MapReduce, with the exception that as many transformations as possible are pipelined and assigned to the same worker.

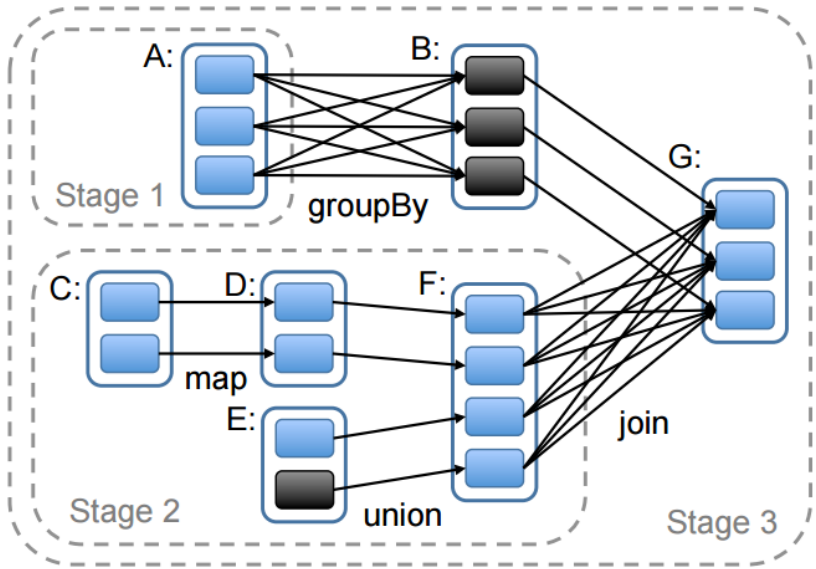


Figure 5: Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build build stages at wide dependencies and pipeline narrow transformations inside each stage. In this case, stage 1's output RDD is already in RAM, so we run stage 2 and then 3.

Machine Learning with Spark: Algorithms

- Consider regressing a binary random variable y on a D-dimensional continuous random variable $\boldsymbol{x}$.
- Classical formulation of logistic regression: $y(\boldsymbol{x}) = 1/(1 + \exp(\boldsymbol{w}^T \boldsymbol{x}))$ together with the cross-entropy loss function

$$L(\boldsymbol{w}) = -\sum_n \log p(y_n|\boldsymbol{w}) = -\sum_n [y_n \log y(\boldsymbol{x}_n) + (1 - y_n) \log(1 - y(\boldsymbol{x}_n))]$$

- Alternative formulation: Predict with the classical but fit $y(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x}$ using the logistic loss function

$$L(\boldsymbol{w}) = \sum_n \log(1 + \exp(-y_n y(\boldsymbol{x}_n)))$$

whose gradient is given by

$$-\sum_n y_n (1 - 1/(1 + \exp(-y_n \boldsymbol{w}^T \boldsymbol{x}_n)))\boldsymbol{x}_n$$

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

```
val points = spark.textFile(...)
                  .map(parsePoint).persist()
var w = // random initial vector
for (i <- 1 to ITERATIONS) {
  val gradient = points.map{ p =>
    p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
  }.reduce((a,b) => a+b)
  w -= gradient
}
```

Machine Learning with Spark: Algorithms

- Logistic regression in Python:

```
# Initialize w to a random value
w = 2 * np.random.ranf(size=D) - 1
print("Initial w: " + str(w))

# Compute logistic regression gradient for a matrix of data points
def gradient(matrix, w):
    Y = 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(-Y * X.dot(w))) - 1.0) * Y * X.T).sum(1)

def add(x, y):
    x += y
    return x

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))
```

Machine Learning with Spark: Algorithms

- *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))
```

Machine Learning with Spark: Algorithms

- Many machine learning methods are already implemented in MLlib, i.e. the user does not need to specify the map and reduce functions.
- 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)
```

Machine Learning with Spark: Algorithms

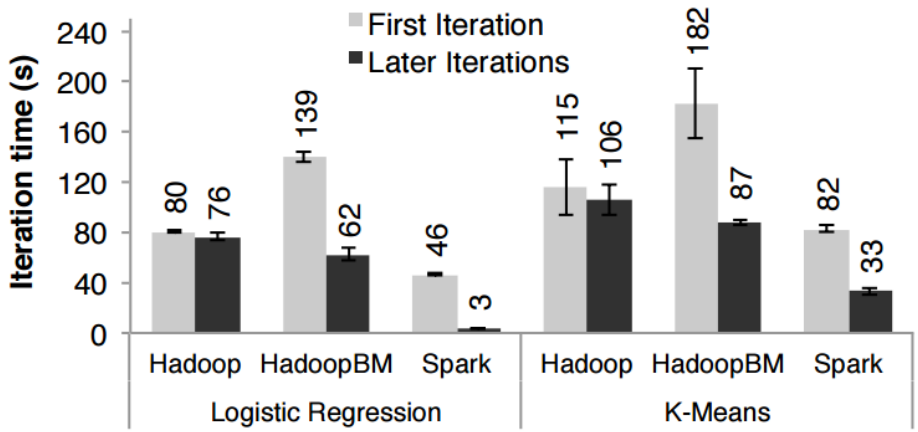
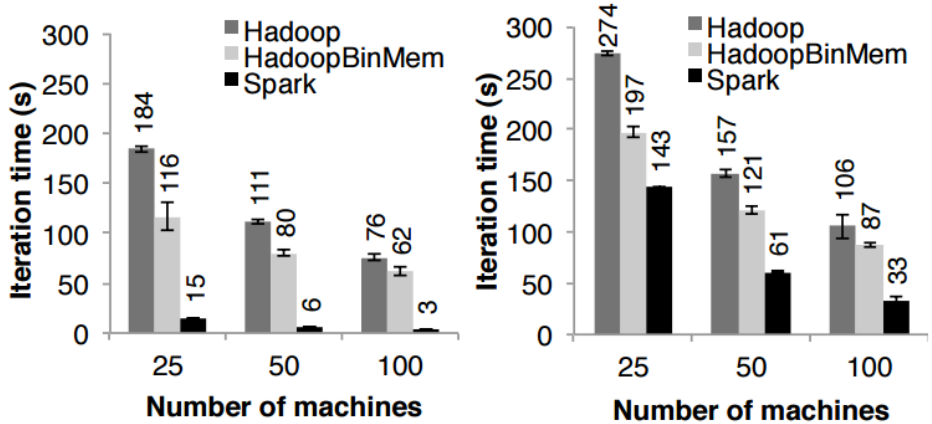


Figure 7: Duration of the first and later iterations in Hadoop, HadoopBinMem and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster.

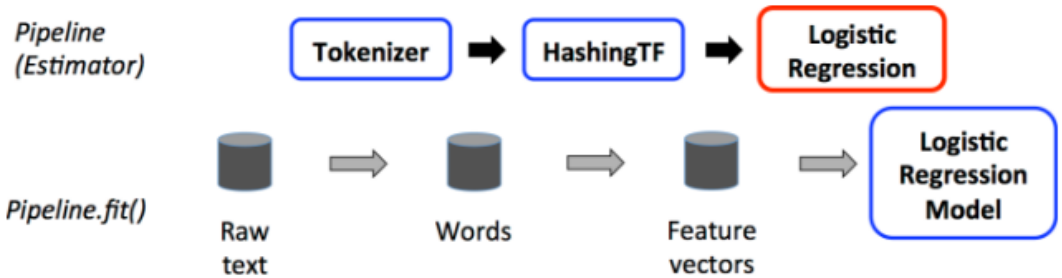


(a) Logistic Regression (b) K-Means

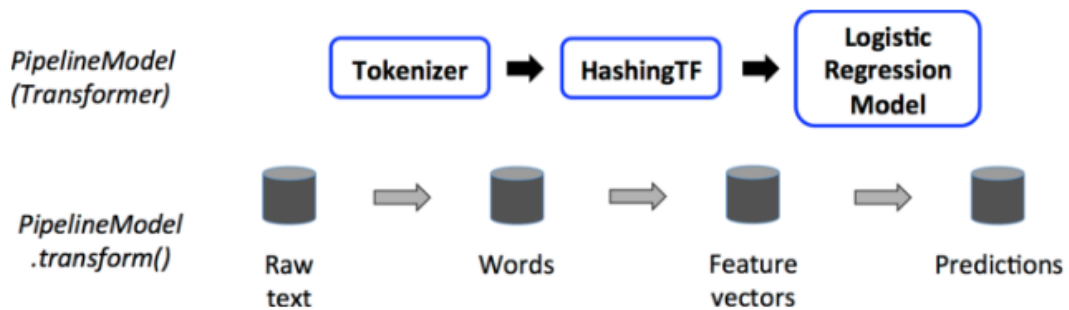
Figure 8: Running times for iterations after the first in Hadoop, HadoopBinMem, and Spark. The jobs all processed 100 GB.

Machine Learning with Spark: Pipelines

- A pipeline is a sequence of stages, where each stage is of one of two types:
  - Transformer: It transforms a dataset into another dataset, e.g. tokenizing a dataset into words is a transformer.
  - Estimator: It fits a model to a dataset. The model becomes a transformer, since it transforms a dataset into predictions.



- A pipeline typically contains estimators and, thus, the pipeline is an estimator itself.
- By fitting the pipeline, the estimators in it become transformers. Then, the pipeline becomes a transformer itself (called pipeline model), which is ready to be used.

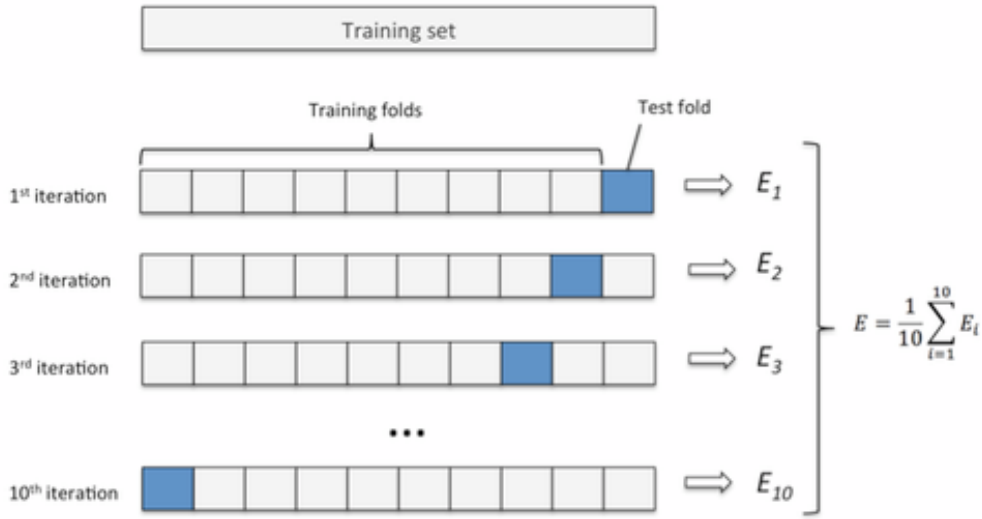


Machine Learning with Spark: Pipelines

```python
# Prepare training documents from a list of (id, text, label) tuples.
training = spark.createDataFrame([
    (0, "a b c d e spark", 1.0),
    (1, "b d", 0.0),
    (2, "spark f g h", 1.0),
    (3, "hadoop mapreduce", 0.0)], ["id", "text", "label"])

# Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.01)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

# Fit the pipeline to training documents.
model = pipeline.fit(training)

# Prepare test documents, which are unlabeled (id, text) tuples.
test = spark.createDataFrame([
    (4, "spark i j k"),
    (5, "l m n"),
    (6, "mapreduce spark"),
    (7, "apache hadoop")], ["id", "text"])

# Make predictions on test documents and print columns of interest.
prediction = model.transform(test)
selected = prediction.select("id", "text", "prediction")
for row in selected.collect():
    print(row)
```

Machine Learning with Spark: Cross-Validation

- Cross-validation is a technique to estimate the prediction error of a model.



- If the training set contains N points, note that cross-validation estimates the prediction error when the model is trained on $N - N/K$ points.
- Note that the model returned is trained on N points. So, cross-validation overestimates the prediction error of the model returned.
- This seems to suggest that a large K should be preferred. However, this typically implies a large variance of the error estimate, since there are only N/K test points.
- Typically, $K = 5, 10$ works well.

Machine Learning with Spark: Cross-Validation

```
# Prepare training documents, which are labeled.
training = spark.createDataFrame([
    (0, "a b c d e spark", 1.0),
    (1, "b d", 0.0),
    (2, "spark f g h", 1.0),
    (3, "hadoop mapreduce", 0.0),
    (4, "b spark who", 1.0),
    (5, "g d a y", 0.0),
    (6, "spark fly", 1.0),
    (7, "was mapreduce", 0.0),
    (8, "e spark program", 1.0),
    (9, "a e c l", 0.0),
    (10, "spark compile", 1.0),
    (11, "hadoop software", 0.0)
], ["id", "text", "label"])

# Configure an ML pipeline, which consists of tree stages: tokenizer, hashingTF, and lr.
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
lr = LogisticRegression(maxIter=10)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

# We now treat the Pipeline as an Estimator, wrapping it in a CrossValidator instance.
# This will allow us to jointly choose parameters for all Pipeline stages.
# A CrossValidator requires an Estimator, a set of Estimator ParamMaps, and an Evaluator.
# We use a ParamGridBuilder to construct a grid of parameters to search over.
# With 3 values for hashingTF.numFeatures and 2 values for lr.regParam,
# this grid will have 3 x 2 = 6 parameter settings for CrossValidator to choose from.
paramGrid = ParamGridBuilder() \
    .addGrid(hashingTF.numFeatures, [10, 100, 1000]) \
    .addGrid(lr.regParam, [0.1, 0.01]) \
    .build()
```

Machine Learning with Spark: Cross-Validation

```
crossval = CrossValidator(estimator=pipeline,
                          estimatorParamMaps=paramGrid,
                          evaluator=BinaryClassificationEvaluator(),
                          numFolds=2)  # use 3+ folds in practice

# Run cross-validation, and choose the best set of parameters.
cvModel = crossval.fit(training)

# Prepare test documents, which are unlabeled.
test = spark.createDataFrame([
    (4, "spark i j k"),
    (5, "l m n"),
    (6, "mapreduce spark"),
    (7, "apache hadoop")
], ["id", "text"])

# Make predictions on test documents. cvModel uses the best model found (lrModel).
prediction = cvModel.transform(test)
selected = prediction.select("id", "text", "probability", "prediction")
for row in selected.collect():
    print(row)
```

- Note that `CrossValidator` requires an estimator as input and, recall, that a pipeline is an estimator.
- Likewise, a pipeline can be used as estimator in another pipeline. This can be used to implement nested cross-validation.

Machine Learning with Spark: Lab

- 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 `temps.csv`. 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.

Machine Learning with Spark: Lab

- Consider regressing an unidimensional continuous random variable y on a D-dimensional continuous random variable $\boldsymbol{x}$.
- 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.

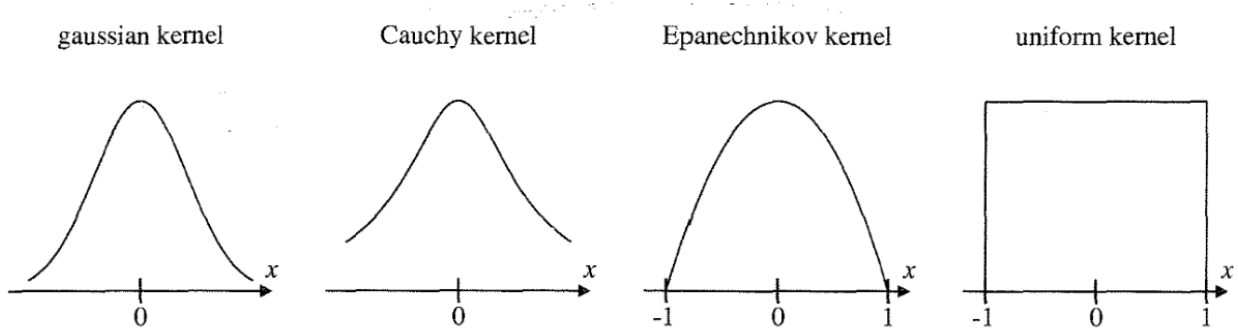
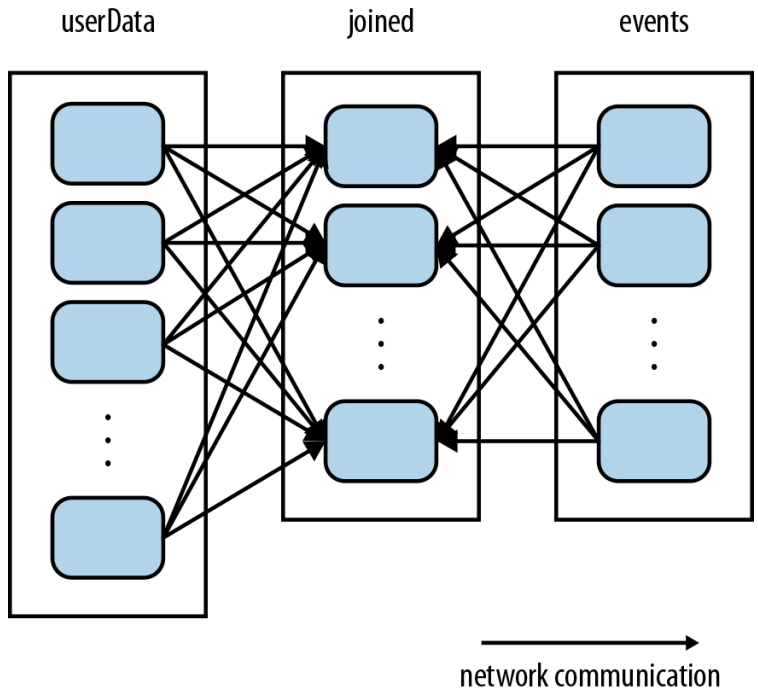


FIGURE 10.3. *Various kernels on $\mathcal{R}$.*

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

Machine Learning with Spark: Lab

- Bear in mind that a join operation may trigger a shuffle operation, which is time and memory consuming.



- 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.