Chapter 2: Decision Trees and Random Forest (pp3-36)

p4: Data science workflow
p5: Regression, classification, decision trees, random forests
p6: Decision tree example
p11: Decision tree
pp.12-14: Precison, recall, accuracy
p15: Issues with trees
p16: Finding good splitting condition: gini purity and entropy
p17: Algorithm to learn decision tree
pp18, 19: Example: splitting conditions
p20: Random forests
p21: Decision trees for regression
pp23-36: Decision trees in Spark's MLlib

Entropy as impurity measure: In the context of decision trees, entropy is used as a criterion to decide the best split at each node. Entropy here measures the impurity of a dataset before and after a split. If a dataset is completely pure (i.e., all data points belong to the same class), the entropy is 0. If the data is evenly distributed among different classes, the entropy is high.

Information gain: The decision tree algorithm aims to reduce entropy or impurity at each split. It calculates the information gain, which is the difference between the entropy of the parent node and the weighted sum of the entropies of the child nodes. The split that maximizes information gain is chosen as the best split.

Chapter 3: Recommender Systems via Matrix Factorization (pp. 37-69)

p38: Recommender system definition
p42: Matrix factorization
pp43-44: Alternating least squares (ALS)
p45: Distributed ALS: method 1
p46: Distributed ALS: method 2
p47: Distributed ALS: runtimes
pp48-68: Recommender Systems in Spark's MLlib
pp49-52: Preparing dataset
p53: Broadcasting closure variables
p54: Training data
p55: Training recommender system
pp58-61: Evaluating model: ROC, AUC, TRP (sensitiviy), FPR (specificity)
pp62-63: Cross-validation
p64: Baseline selection
pp65-66: Hyperparameters tuning
p67: Creating batch recommentations

Chapter 4: Text processing with latent semantic analysis (pp69-108)

p70: How to analyze text data (+TF-IDF)
p73: Latent semantic (LS) analysis vs LS indexing
p74: SVD
pp75-77: SVD + cosine similarity
pp78-80: SVD reduced decomposition
p81: Processing queries
pp82-83: Finding SVD of a word-document matrix
pp84-: Text analysis and SVD in Spark's MLlib
p87: NLP pipeline
p88: Filtering
p89: From text to word lemmas
p91: Computing TF weights
pp93-95: Computing DF weights
p96: Broadcasting term dictionary
p97: Merging TF & IDF weights into sparse vectors
p98 : Computing SVD
p99: Finding top terms for latent concepts
p100: Finding top documents for latent concepts
p103: Finding most similar documents to a given query document
p104: Finding top similar terms to a given query term
p105: Finding top similar documents to a given query term
pp106-107: Multi-term queries
p108: Summary

Chapter 5: K-Means Clustering & Anomaly Detection (pp109-141)

p110: Clustering
p114: Clustering objective
p115: Naive clustering algorithm
p116: Basic clustering algorithm
p117: K-means discussion
pp118-: K-means in Spark's MLlib
pp126-129: Finding good value for k
pp130-132: Visualization in R
pp133-136: Normalization
p137: Translating categorical attributed into numerical
pp138-140: Measuring entropy
p141: Final anomaly detection (+ anomaly definition)

In information theory, entropy is a measure of uncertainty or randomness. In the context of clustering, entropy measures the uncertainty or randomness of the labels within each cluster. If all data points in

a cluster have the same label, the entropy is low, indicating high purity or homogeneity. Conversely, if the labels within a cluster are mixed or diverse, the entropy is high, indicating low purity or homogeneity.

Chapter 6: Medical Network Analysis in GraphX (pp143-180)

p144: GraphX
pp147-150: Parsing XML data
p151: Absolute frequencies
p152: Co-Occurrences of topics + maximum amount of unique edges in undirected graphs
pp153-154: Hashing for vertices
p155: Vertices
p156: Edges
p157: Network analysis algorithms: - connected components - degree distribution - cliques/triangles & clustering coefficients - diameter & average path length

pp158-161: Connected components
pp162-163: Degree distribution
pp164-169: Digression: filtering out noisy edges (Chi-squared test)
pp170-171: Cliques, triangle counts & clustering coefficients
pp172-177: Average path length
pp178-179: PageRank algorithm
p180: Summary

Chapter 7: Geospatial, Temporal & Streaming Data Analysis (pp181-210)

p182: Temporal & spatial data
p183: Spheres & polygons
p184: GeoJSON
p186: Date & time APIs
pp187-189: Geometric APIs
pp190-191: GeoJSON API
p192: Custom TaxiTrip data structure
p193: *parse* function
pp194-198: Safe parsing & filtering: *Either[left, right]*
pp199-200: Loading boroughs geometry
p201: Combining datasets
p203: "Sessionization" of taxi trips
pp204-205: Secondary sorting
p207: *Sliding* operator
p210: Summary

Chapter 8: Financial Risk Analysis via Monte Carlo Simulations (pp212-242)

p213: Value-at-Risk estimation
p215: Stock markets & returns of investments
p216: Linear regression model
p217: Value-at-Risk & financial risk
p218: Portfolio, stocks, factors
p219: Time series data
p220: Monte Carlo simulations for market conditons & VaR
p221: Multivaraite normal
p222: Datasets for stocks and factors
pp223-224: Parsing time series files
p225: Trimming time series
p226: Auto-completing time series
p227: Sliding window & bi-weekly returns
pp228-229: Plotting factor returns
p230: Fitting the paramters: computing the factor means and co-variances
p231: "Featurizing" factor returns
p232: Training linear-regression model
p233: Parallel sampling
p234: Spark *Dataset* API
p235: Running the sampler
p236: Predicting the average stock return
p237: Calculating the VaR
p240: Conditional VaR
p241: Summary

Strongly typed data structures are those where types are explicitly declared and enforced. Type checking: In strongly typed languages, the compiler performs rigorous type checking at compile time. This prevents operations that are not type-safe, such as trying to add a string to an integer.

Weakly typed data structures allow more flexibility and implicit type conversion. Variables can change types, and operations can be performed on mismatched types through implicit coercion. Implicit Conversions: Weakly typed languages often perform implicit type conversions, which can lead to unexpected behavior and bugs that are harder to trace.

Big Data Analytics

Chapter 2: Predictive Data Analysis with Decision Trees & Random Forests

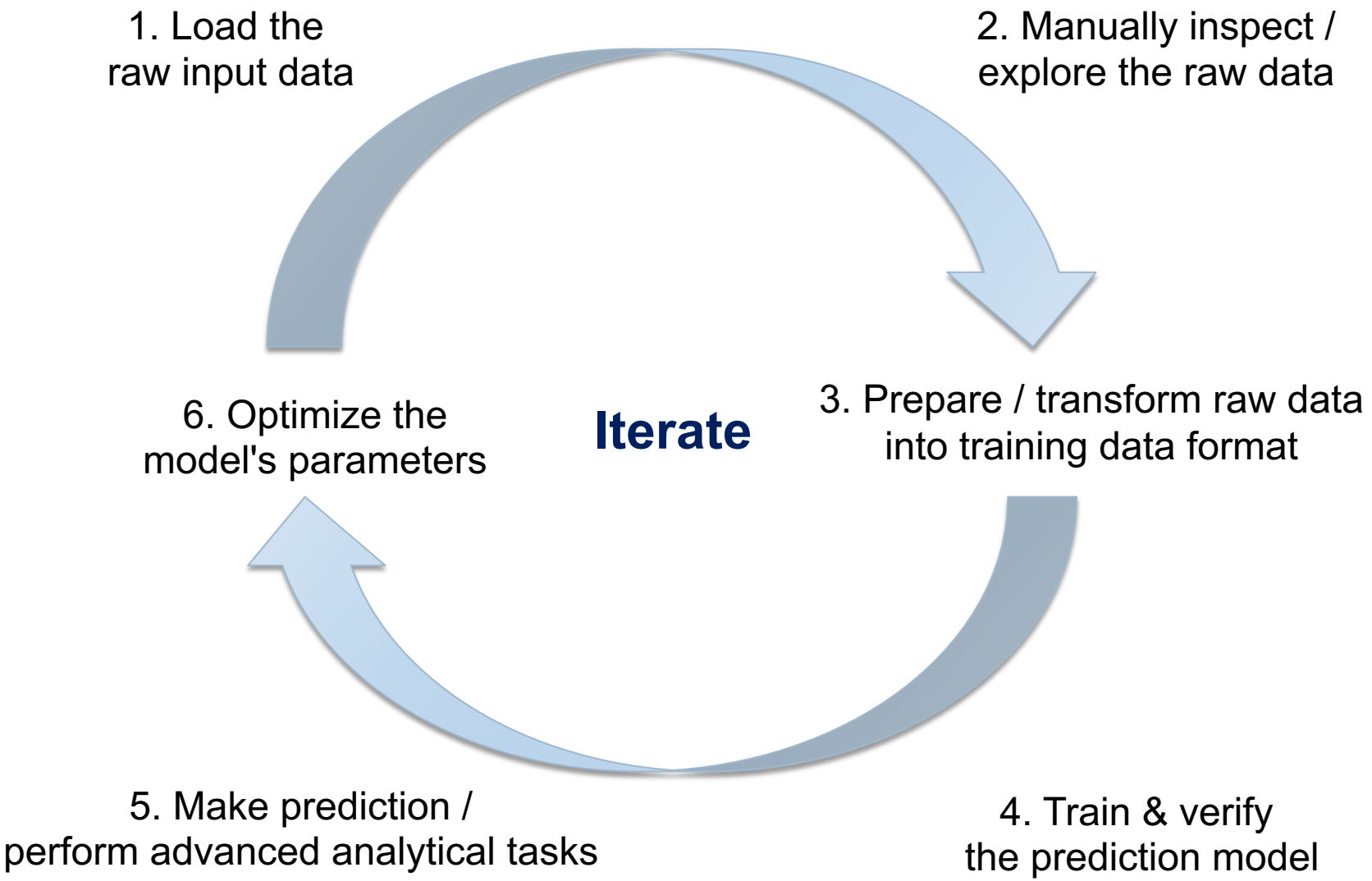
Following: [3] "**Advanced Analytics with Spark**", Chapter 4

Prof. Dr. Martin Theobald

Faculty of Science, Technology & Communication

University of Luxembourg

Typical Data Science Workflow



Fast-Forward from Regression to Classification

Regression denotes the broad task of analyzing the dependencies among a **number of input variables** (either categorical or numerical), typically in order to predict the value (then also called the "label") of a **single output variable**.
(see the `LinearRegression` example from Chapter 1.2)

More specifically, the task of predicting the **label of a new object** based on the **labels of a set of training objects** belongs to the class of **supervised learning techniques**.

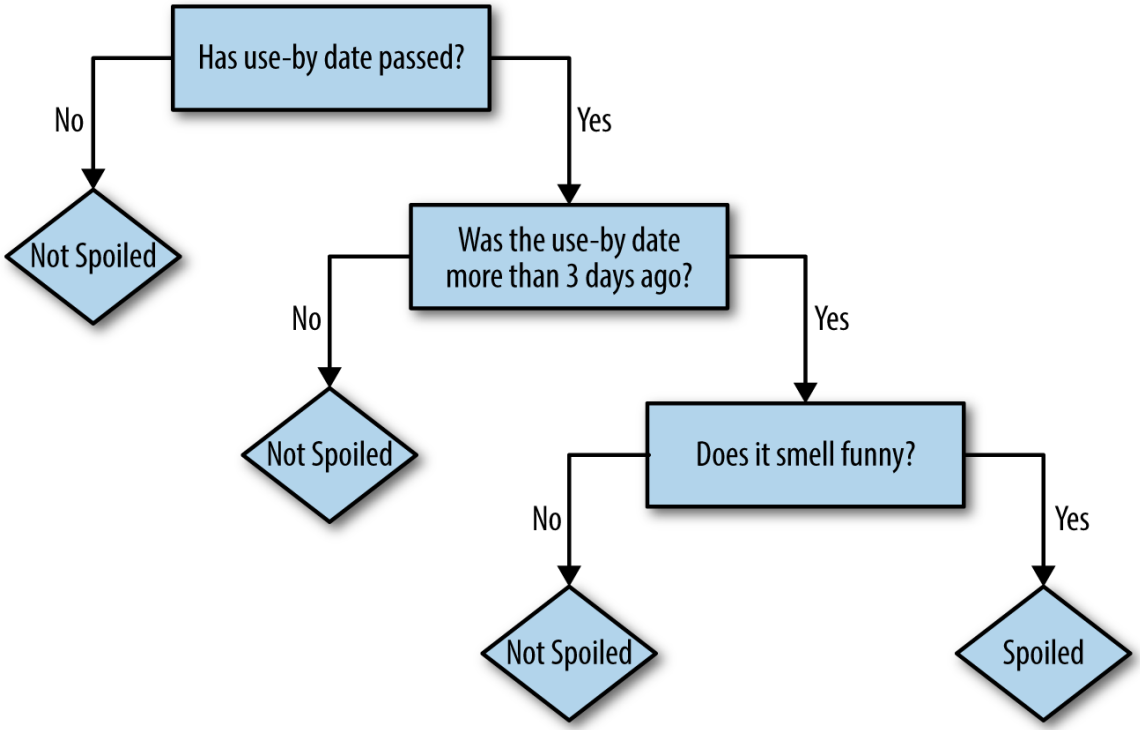
Classification is a supervised learning technique that usually **predicts a categorical label**, such as the type or class that a new object may belong to.

Decision trees are a simple (both intuitive and effective) form of classification which can handle both numerical and categorical data, typically by imposing a

- a **binary decision** at each inner node of the tree, thus predicting
- a **single label** for each object at a leaf node of the tree.

Random forests are sets of (more or less "randomly" generated) decision trees. The prediction then is usually made by a **majority vote** among the individual trees.

First Decision Tree Example



- Here, the products are supposed to be classified into the two categories "**Spoiled**" and "**Not Spoiled**" based on their "use-by date" and whether they "smell funny".
- However, this may not be the best-possible decision tree for this kind of prediction:
  - "Was the use-by date more than 3 days ago?" condition completely subsumes the "Has the use-by date passed?" condition.
  - But what if something smells funny even before the use-by date has passed?

The CoverType Data Set

The **CoverType data set** records the various types of forests that cover parcels of land in Colorado (as of 1998).

https://archive.ics.uci.edu/ml/machine-learning-databases/covtype/

The main data file, covtype.data.gz (11 MB), consists of 518,012 parcels with 54 attributes (both numerical and categorical) that describe each parcel, including its *elevation*, *slope*, *distance to water*, *shade*, *soil type*, etc., along with the *type of forest* that covers the parcel.

The second file, covtype.info, contains a description of the attributes and their measurements in covtype.data.gz.

- The first 10 columns are **quantitative measurements** (of various units).
- The following 4 columns are (binary) **qualitative measurements** that describe the location of the wilderness area.
- The following 40 columns are (binary) **qualitative measurements** that describe the soil type.

CoverType is a very well-studied data set in many predictive-analysis and machine-learning applications.

- Kaggle includes a separate competition based on the CoverType data set.

Outline of Chapter 2

2.1 Decision Trees & Random Forests

- Labeled Feature Vectors
- Finding a Good Splitting Condition: Gini Purity & Entropy
- C4.5 Algorithm for Learning a Binary Decision Tree
- Random Forests: Prediction & Regression

2.2 Building Decision Trees in Spark's MLlib

- Preparing the Training Data
- Optimizing the Parameters of the Model
- Transforming Categorical Attributes
- Prediction & Regression with Random Forests

2.1 Decision Trees & Random Forests

Labeled Feature Vectors

- Consider the following table of **labeled training data** about pets for kids:

ID	Pet Name	Weight (kg)	Number of Legs	Color	Suitable for Kids?
1	Fido	20.5	4	Brown	**Yes**
2	Mr. Slither	3.1	0	Green	**No**
3	Nemo	0.2	0	Tan	**Yes**
4	Dumbo	1,390.8	4	Grey	**No**
5	Kitty	12.1	4	Grey	**Yes**
6	Jim	150.9	2	Tan	**No**
7	Millie	0.1	100	Brown	**No**
8	McPigeon	1.0	2	Grey	**No**
9	Spot	10.0	4	Brown	**Yes**

- Every row encodes a **vector of attribute values** that describe the pet.
- "**Suitable for Kids?**" is a distinguished attribute (i.e., the **label**) whose value we would like to predict also for new pets we have not seen yet.

Decision Tree

A **decision tree** is a tree-structured classifier whose inner nodes (including the root) encode **decision rules** and whose leafs represent the **labels** we wish to assign to the **conjunction of decisions** made under each root-to-leaf path in the tree.

A **binary decision tree** is a decision tree whose decision rules are all binary.

Both the decision rules and the labels at the leafs of the tree are not necessarily disjoint. Thus, sometimes decision trees are compacted into a **directed-acyclic graph** (DAG), then called a **decision diagram**.

Goals in learning a decision-tree classifier:

- A decision tree should **separate the training data well** according to the provided labels at each of its leaf nodes.
- Decision trees should **not be too deep** nor contain too many **redundant** subtrees (and hence avoid "overfitting" to the training data).
- Decision trees should usually be **balanced**.

Precision, Recall & Accuracy (Binary Classification)

For **binary classification** tasks, i.e., when all items to be classified carry only one out of **two possible labels** (such as Yes/No), we distinguish among the following **four classes of predictions** made by the classifier:

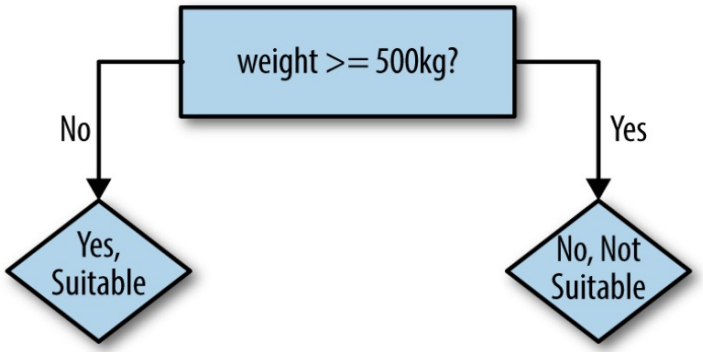
		Actual (Given) Label	
		Yes	No
Predicted Label	Yes	True Positive (TP)	False Positive (FP)
	No	False Negative (FN)	True Negative (TN)

Explanation:

- **True**: correct prediction by the system
- **False**: wrong prediction by the system
- **Positive**: an item predicted by the system
- **Negative**: an item missed by the system

From the above definitions of TP, TN, FP & FN, we can define the three principal quality measures **precision**, **recall** & **accuracy** (see next slides).

Small Decision Tree Example



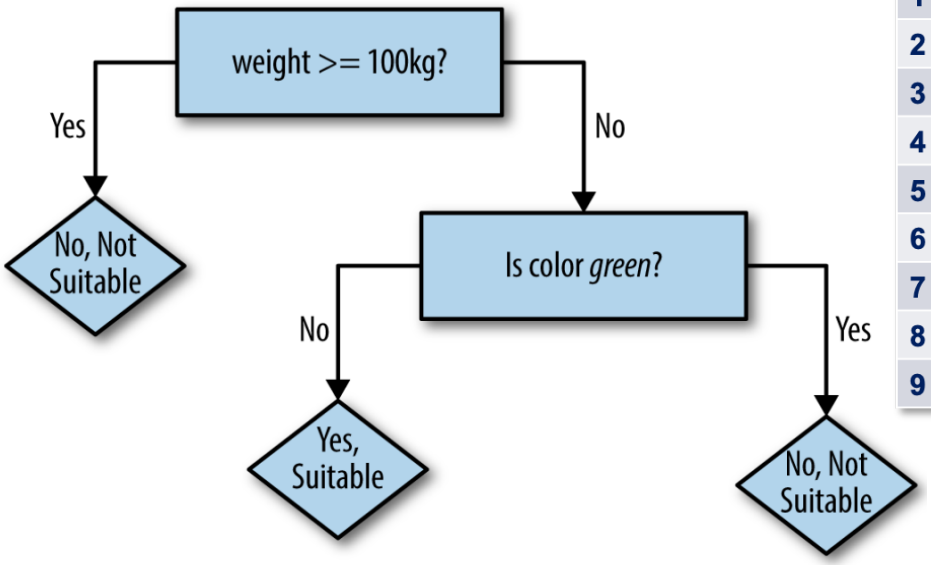
ID	Pet Name	Weight (kg)	Number of Legs	Color	Suitable for Kids?
1	Fido	20.5	4	Brown	Yes
2	Mr. Slither	3.1	0	Green	No
3	Nemo	0.2	0	Tan	Yes
4	Dumbo	1,390.8	4	Grey	No
5	Kitty	12.1	4	Grey	Yes
6	Jim	150.9	2	Tan	No
7	Millie	0.1	100	Brown	No
8	McPigeon	1.0	2	Grey	No
9	Spot	10.0	4	Brown	Yes

- The above decision tree classifies the data of Slide 8 with respect to being in the class "**Yes, Suitable**" into the following four categories:
  - **True Positives** (TP): 1, 3, 5, 9
  - **True Negatives** (TN): 4
  - **False Positives** (FP): 2, 6, 7, 8
  - **False Negatives** (FN): none
- Thus, we obtain a **precision** of $prec = \frac{|TP|}{|TP|+|FP|} = \frac{1}{2}$,

  a **recall** of $rec = \frac{|TP|}{|TP|+|FN|} = 1$,

  and an **accuracy** of $acc = \frac{|TP|+|TN|}{|TP|+|TN|+|FP|+|FN|} = \frac{5}{9}$.

Larger Decision Tree Example



ID	Pet Name	Weight (kg)	Number of Legs	Color	Suitable for Kids?
1	Fido	20.5	4	Brown	Yes
2	Mr. Slither	3.1	0	Green	No
3	Nemo	0.2	0	Tan	Yes
4	Dumbo	1,390.8	4	Grey	No
5	Kitty	12.1	4	Grey	Yes
6	Jim	150.9	2	Tan	No
7	Millie	0.1	100	Brown	No
8	McPigeon	1.0	2	Grey	No
9	Spot	10.0	4	Brown	Yes

- This decision tree classifies the data of Slide 8 with respect to being "**Suitable for Kids?**" into the following categories:
  - **True Positives** (TP): 1, 3, 5, 9
  - **True Negatives** (TN): 2, 4, 6
  - **False Positives** (FP): 7, 8
  - **False Negatives** (FN): none
- This time, we obtain a **precision** of $prec = \frac{|TP|}{|TP|+|FP|} = \frac{2}{3}$, a **recall** of $rec = \frac{|TP|}{|TP|+|FN|} = 1$, and an **accuracy** of $acc = \frac{|TP|+|TN|}{|TP|+|TN|+|FP|+|FN|} = \frac{7}{9}$.

Basic Issues

- For a categorical attribute with n possible values, we might have to consider $\mathbf{2^n - 1}$ **possible binary decision rules** for learning the best decision tree, i.e., the one that best separates the training data according to the given labels.
- For m categorical attributes, each with n possible values, we might even obtain $\mathbf{2^{m+n} - 1}$ **possible binary decision rules**.
- For numerical attributes, there might be **uncountably many** such rules.
- Even if we managed to find the best decision tree, it might end up to be completely **overfitted to the training data** but does not well classify new objects with an unknown label.

Finding a Good Splitting Condition

- Suppose that we split a **set of labeled training vectors** D vectors into **two subsets** D_{left} and D_{right}.
- Ideally, each of the two subsets would contain just one distinct label, i.e., they would be completely "**pure**".
- We can thus measure the "**gain in purity**" ΔI based on the (im-)purity of the sets before and after the splits:

$$\Delta I = I(D) - I\left(D_{left}\right) - I\left(D_{right}\right)$$

- Since the three sets are not of equal size, we should **normalize** their impact on this gain in purity:

$$I(D) = \frac{|D|}{N} I_{G|E}(D)$$

Where N is the total number of labeled feature vectors in the entire training set.

- Usual choices for **measuring purity** which applied in decision-tree learning are:

Gini purity: $I_G(D) = 1 - \sum_{i=1}^{n} f_i^2$

Entropy: $I_E(D) = -\sum_{i=1}^{n} f_i \log f_i$

Where n is the number of distinct labels and f_i is the relative frequency of label i in D.

Learning a Decision Tree

Greedy algorithm for learning a **binary decision tree** (aka. "**C4.5**"):

1. Let D denote the set of **labeled training vectors** (having n distinct labels)
2. For each **categorical attribute** A of vectors in D
   - For each distinct value v of A
     - Compute the gain ΔI in Gini purity or entropy by splitting D into D_{left} and D_{right} using $A = v$ as decision rule
3. For each **numerical attribute** A of vectors in D
   - Discretize the range of A into m uni-width bins
   - For each center v of a uni-width bin for A
     - Compute the gain ΔI in Gini purity or entropy by splitting D into D_{left} and D_{right} using $A < v$ as decision rule
4. Let $(A, v)^{best}$ denote the splitting condition that maximizes the gain ΔI
5. Repeat from step 1 for both D_{left} and D_{right} based on $(A, v)^{best}$ as splitting condition until $maxdepth$ or a gain of $\Delta I \leq 0$ is reached
6. Add the **most common label** in D **as leaf**

Example: Splitting Conditions

Let $I(D) = \frac{9}{9}\left(1 - \left(\left(\frac{4}{9}\right)^2 + \left(\frac{5}{9}\right)^2\right)\right) \approx 0.494$

- Consider a split on $weight < 500$:

$$I(D_{left}) = \frac{8}{9}\left(1 - \left(\left(\frac{4}{8}\right)^2 + \left(\frac{4}{8}\right)^2\right)\right) \approx 0.444$$

$$I(D_{right}) = \frac{1}{9}\left(1 - \left(\frac{1}{1}\right)^2\right) = 0$$

That is: $\Delta I \approx 0.05$

- Consider a split on $weight < 100$:

$$I(D_{left}) = \frac{7}{9}\left(1 - \left(\left(\frac{3}{7}\right)^2 + \left(\frac{4}{7}\right)^2\right)\right) \approx 0.381$$

$$I(D_{right}) = \frac{2}{9}\left(1 - \left(\frac{2}{2}\right)^2\right) = 0$$

That is: $\Delta I \approx 0.113$

ID	Weight (kg)	Suitable for Kids?
1	20.5	Yes
2	3.1	No
3	0.2	Yes
4	1,390.8	No
5	12.1	Yes
6	150.9	No
7	0.1	No
8	1.0	No
9	10.0	Yes

Example: Splitting Conditions (ct'd)

Let $I(D') = \frac{7}{9}\left(1 - \left(\left(\frac{3}{7}\right)^2 + \left(\frac{4}{7}\right)^2\right)\right) \approx 0.381$

- Consider a split on $color = Green$:

$$I(D'_{left}) = \frac{1}{9}\left(1 - \left(\frac{1}{1}\right)^2\right) = 0$$

$$I(D'_{right}) = \frac{6}{9}\left(1 - \left(\left(\frac{4}{6}\right)^2 + \left(\frac{2}{6}\right)^2\right)\right) \approx 0.296$$

That is: $\Delta I \approx 0.085$

- Consider a split on $color = Brown$:

$$I(D'_{left}) = \frac{3}{9}\left(1 - \left(\left(\frac{2}{3}\right)^2 + \left(\frac{1}{3}\right)^2\right)\right) \approx 0.148$$

$$I(D'_{right}) = \frac{4}{9}\left(1 - \left(\left(\frac{2}{4}\right)^2 + \left(\frac{2}{4}\right)^2\right)\right) = 0.222$$

That is: $\Delta I \approx 0.011$

ID	Color	Suitable for Kids?
1	Brown	Yes
2	Green	No
3	Tan	Yes
5	Grey	Yes
7	Brown	No
8	Grey	No
9	Brown	Yes

Random Forests

- When initialized on the complete set of labeled feature vectors as input, the C4.5 algorithm of Slide 15 behaves in a **completely deterministic** manner.
- Thus, although it is a greedy algorithm, i.e., it may not find the truly best decision tree that minimizes classification mistakes, it will always return the same tree with the same decision rules.

- There are two basic ways how to **randomize C4.5**:
  1. Consider a **random** subset A' of **attributes** in D (with replacement) for finding the splitting conditions.
  2. Consider a **random** subset of **labeled feature vectors** D (with replacement) for learning the decision tree.

- **Label prediction** with a random forest then simply works by choosing the
  - **arithmetic mean** of the individual tree predictions for a numerical label, or by
  - **majority vote** among the individual tree predictions for a categorical label.

- Randomization helps us to **avoid overfitting**!

- Multiple decision trees can be learned **in parallel**!

Decision Trees for Regression

- Decision Trees and Random Forests can be used to **approximate an arbitrary regression function** that then predicts a **numerical label** instead of a categorical one.
- Only **numerical attributes** are allowed in this case as input. However, a same attribute may **occur repeatedly** at different levels or in different subtrees of each of the learned decision trees.
- The main difference to classification is the choice of the splitting condition for each numerical attribute, which usually is based on **minimizing the sample variance**:

$$\mathrm{Var}(D) = \frac{1}{|D|}\sum_{i=1}^{|D|}(y_i - \mu)^2$$

Where y_i is the label of an instance in D and $\mu = \frac{1}{|D|}\sum_{i=1}^{|D|} y_i$ is the **sample mean** of all instances in D.

- **Prediction** in a random forest then again calculates the average label (i.e., the arithmetic mean) among the individual tree predictions.

2.2 Decision Trees in Spark's MLlib

Prepare the Training Data (I)

- Download and copy the covtype.data file into a local directory and/or your home directory on the IRIS cluster.
- The file covtype.info provides a concise description of the data, so we may refrain from collecting more statistics about the data at this point.
- Instead, directly load the data file into an RDD and transform the data into an array of `LabeledPoint` objects (much like in the `LinearRegression` examples shown earlier).

```
import org.apache.spark.mllib.linalg._
import org.apache.spark.mllib.regression._

val rawData = sc.textFile("./covtype.data")
val data = rawData.map { line =>
  val values = line.split(',').map(_.toDouble)
  val featureVector = Vectors.dense(values.init)
  val label = values.last - 1
  LabeledPoint(label, featureVector) }
```

Prepare the Training Data (II)

- `values.init` returns all but the last values of the array.
- `values.last` returns the last value of the array. We subtract 1 from this value, since the MLlib decision-tree implementation needs numerical labels that start at 0.
- Note that the data set contains 4 binary attributes for the wilderness area and 40 binary attributes for the soil type, of which only 1 each is active at a time (this is also called a "**1-hot encoding**").
- This encoding may or may not be a good choice for learning decision trees, as it effectively turns these numerical attributes into categorical ones. We will revisit this issue later…

Prepare the Training Data (III)

- We will work on three splits of sizes 80%/10%/10% of the raw data, called **training set**, **validation set**, and actual **testing set**, in the following.

```
val Array(trainData, valData, testData) =
  data.randomSplit(Array(0.8, 0.1, 0.1))
trainData.cache()
valData.cache()
testData.cache()
```

- We will train the model on one fixed subset, called `trainData`, and optimize the basic parameters of the model on the second set, called `valData`.
- The third set, called `testData`, will be used for the final evaluation.
- All the three RDD's are cached for later re-use.

Build a First Model

- Now we are already good to build our first decision tree in Spark:

```
import org.apache.spark.mllib.tree._
import org.apache.spark.mllib.tree.model._

val model = DecisionTree.trainClassifier(
   trainData, 7, Map[Int,Int](), "gini", 4, 100)
```

- Here, we must already specify the following parameters:
  - `7` is the number of prediction labels to expect
  - `Map[Int,Int]()` holds data about the number of distinct categories per feature (will be determined automatically if left empty)
  - `gini` is the desired Gini purity measure
  - `4` is the maximum depth of the tree
  - `100` is the maximum bin count for discretizing numerical features

Evaluate the Model

- The `evaluation` package of Spark's MLlib contains implementations of various **evaluation metrics**.
- We thus connect our trained model with Spark's evaluation metrics package for multi-class labels, called `MulticlassMetrics`.

```
import org.apache.spark.rdd._
import org.apache.spark.mllib.evaluation._

def getMetrics(model: DecisionTreeModel, data:
     RDD[LabeledPoint]):
  MulticlassMetrics = {
    val predictionsAndLabels = data.map(example =>
      (model.predict(example.features), example.label)
    )
    new MulticlassMetrics(predictionsAndLabels)
  }

val metrics = getMetrics(model, valData)
```

Show Precision & Recall

- Get a glance at the overall accuracy across all the 7 labels:

```
metrics.accuracy
```

- Our basic model thus already achieves an **accuracy of about 70%**.
- To get a detailed view on precision and recall for each of the 7 labels, we can do the following:

```
(0 until 7).map(
  label => (metrics.precision(label), metrics.recall(label))
).foreach(println)
```

- Note that precision, recall and accuracy are actually defined for binary classification tasks only.
- Accuracy (at least as defined in Spark's `MulticlassMetrics` package) simply counts for how many elements in `valData` the predicted label indeed matches the actual (i.e., given) label and then normalizes this count by the number of elements in `valData`.

Compare to a Reasonable Baseline

- Consider a classifier that randomly—but proportionally—picks a label according to the relative frequency of labels in the training and validation sets.

```
def classProbabilities(data: RDD[LabeledPoint]): Array[Double] = {
  val countsByCategory = data.map(_.label).countByValue()
  val counts = countsByCategory.toArray.sortBy(_._1).map(_._2)
  counts.map(_.toDouble / counts.sum)
}
val trainPriorProbabilities = classProbabilities(trainData)
val valPriorProbabilities = classProbabilities(valData)
trainPriorProbabilities.zip(valPriorProbabilities).map {
  case (trainProb, valProb) => trainProb * valProb
}.sum
```

- Random guessing achieves an **accuracy of only 37%** according to the above calculation, so our basic model already performs much better than random guessing!

Optimizing the Hyper-Parameters of the Model

```
val evaluations =
  for (impurity <- Array("gini", "entropy");
    depth <- Array(10, 20, 30);
    bins <- Array(50, 100, 300))
  yield {
    val model = DecisionTree.trainClassifier(
      trainData, 7, Map[Int,Int](), impurity, depth, bins)
    val predictionsAndLabels = valData.map(example =>
      (model.predict(example.features), example.label)
    )
    val accuracy =
      new MulticlassMetrics(predictionsAndLabels).accuracy
    ((impurity, depth, bins), accuracy) }

  evaluations.sortBy(_._2).reverse.foreach(println)
```

- The result is a **best accuracy value of about 91.2%** across all labels.

Transform Binary Categorical Attributes into Multivalued Ones

- Using 4 and 40 binary attributes to encode a two categorical attributes may be wasteful after all and consume a lot of time for training.
- So let's transform these into two multivalued categorical attributes as follows:

```
val data = rawData.map { line =>
  val values = line.split(',').map(_.toDouble)
  val wilderness = values.slice(10, 14).indexOf(1.0).toDouble
  val soil = values.slice(14, 54).indexOf(1.0).toDouble
  val featureVector =
    Vectors.dense(values.slice(0, 10) :+ wilderness :+ soil)
  val label = values.last - 1
  LabeledPoint(label, featureVector)
}

val Array(trainData, valData, testData) =
  data.randomSplit(Array(0.8, 0.1, 0.1))
```

Revalidate the Model

- Once more optimize the parameters of the model. Also explicitly provide the number of categorical values for the two new attributes:

```
val evaluations =
  for (impurity <- Array("gini", "entropy");
      depth <- Array(10, 20, 30);
      bins <- Array(50, 100, 300))
  yield {
    val model = DecisionTree.trainClassifier(
       trainData, 7, Map(10 -> 4, 11 -> 40),
       impurity, depth, bins)
    val trainAccuracy = getMetrics(model, trainData).accuracy
    val valAccuracy = getMetrics(model, valData).accuracy
    ((impurity, depth, bins), (trainAccuracy, valAccuracy))
  }
```

- This steps even yields a **final accuracy of about 94.5%** when evaluated over the `valData` split, which is a gain of 3% over the binary encoding!

Train a Random Forest

- After having optimized our data encoding and the model parameters, we finally merge the `trainData` and `valData` splits to train a **Random Forest** based on 90% of the available covtype data set.

- Based on the afore defined parameters, we may thus train an entire forest of 10 binary decision trees as follows:

```
val forest = RandomForest.trainClassifier(
  trainData.union(valData), 7, Map(10 -> 4, 11 -> 40), 10,
  "auto", "entropy", 30, 300)
```

- The `RandomForest` implementation in Spark's MLlib involves two new parameters:
  - `10` is the number of randomly generated decision trees
  - `"auto"` denotes an automatic choice of training subsets and attributes

Make a Prediction

- Finally, we can also "manually" classify a feature vector:

```
val input = "2709,125,28,67,23,3224,253,207,61,6094,0,29"
val vector = Vectors.dense(input.split(',').map(_.toDouble))

forest.predict(vector)
```

Using Random Forests for Regression

```
import org.apache.spark.mllib.tree.RandomForest
import org.apache.spark.mllib.util.MLUtils

// Load and parse the data file.
val data = MLUtils.loadLibSVMFile(sc, "./sample_libsvm_data.txt")
val splits = data.randomSplit(Array(0.7, 0.3))
val (trainingData, testData) = (splits(0), splits(1))

// Train a RandomForest model. Empty categoricalFeaturesInfo indicates all features are continuous.
val numClasses = 2
val categoricalFeaturesInfo = Map[Int, Int]()
val featureSubsetStrategy = "auto" // Let the algorithm choose the attributes
val impurity = "variance"; val maxDepth = 4; val maxBins = 32; val numTrees = 3

val model = RandomForest.trainRegressor(trainingData, categoricalFeaturesInfo,
  numTrees, featureSubsetStrategy, impurity, maxDepth, maxBins)

// Evaluate model on test instances and compute test error
val labelsAndPredictions = testData.map { point =>
  val prediction = model.predict(point.features)
  (point.label, prediction) }
val testMSE = labelsAndPredictions.map{ case(v, p) => math.pow((v - p), 2)}.mean()
```

See: https://spark.apache.org/docs/2.2.0/mllib-ensembles.html

Extensions

- **More Tuning**
  - Systematically explore a larger parameter space.
  - Transform dependent features into a more expressive form (e.g., avoid "1-hot encodings").
  - Perform more thorough cross-validations (instead of using just a single 80/10/10 split).

- **Compare to other classification techniques** (e.g., in Spark's MLlib)
  - Linear Regression / Logistic Regression
  - Naïve Bayes
  - Support Vector Machines (SVMs)

Big Data Analytics

Chapter 3: Recommender Systems via Matrix Factorization

Following: [3] "**Advanced Analytics with Spark**", Chapter 3

Prof. Dr. Martin Theobald

Faculty of Science, Technology & Communication

University of Luxembourg

What is a Recommender System?

Recommender systems are at the intersection of data mining and predictive analysis, in which the goal is to predict a rating that a user would give to a (previously unrated) item.

- **Collaborative filtering** aims to find recommendations either based on the users' past behavior, or on the similarity of the user to other users who have previously rated items.
- **Content-based filtering** aims to find recommendations based on the similarity of an item to other items that the same user has rated.
- These two aspects are often combined…

Most "famous" example of a recommender system: http://www.netflixprize.com, which awarded 1 Mio USD to the winning team in 2009.

- The most accurate algorithm in 2007 used an ensemble method of 107 different algorithmic approaches, which were blended into a single prediction.
- The Netflix data set consisted of 100 Mio movie ratings of the following structure:

```
<user_id, movie_id, date_of_rating, rating>
```

The AudioScrobbler Data Set

AudioScrobbler was one of the first recommender platforms developed for online music portals such as last.fm (similar to Spotify, Amazon Music, iTunes, etc.)

- Obviously, the goal is to recommend new artists to users who have an existing profile of artists they already played.
- Available at: https://datahub.io/dataset/audioscrobbler (and on Moodle!)
- The archive contains 3 files of approximately 135 MB in compressed format.

user_artist_data.txt

- "Implicit" collection of 24 Mio individual user ratings (141K users, 1.6M artists):
- About 426 MB in uncompressed space-separated format:

```
<user_id <SPACE> artist_id <SPACE> number_of_times_played>
```

artist_data.txt

- Mappings of about 1.8 Mio artist ids to their actual names in TSV format:

```
<artist_id <TAB> artist_name>
```

artist_alias.txt

- Pairwise artist aliases, i.e., capturing different name variations of a same artist:

```
<artist_id <SPACE> artist_alias_id>
```

Outline of Chapter 3

3.1 Matrix Factorization

- Approximate Matrix Factorization
- Alternating Least Squares Algorithm
- Performance Comparison

3.2 Recommender Systems in Spark's MLlib

- Loading and Exploring the Data Set
- Broadcasting Closure Variables
- Preparing the Training Data
- Evaluation of Recommendation Quality
- Hyper-Parameter Tuning

3.1 Matrix Factorization

Matrix Factorization: Goal & Examples

Goal: decompose a non-negative $n \times m$ matrix $\boldsymbol{R}$ into two smaller matrices $\boldsymbol{X}^T$ and $\boldsymbol{Y}$ of sizes $n \times k$ and $k \times m$, respectively, such that $\boldsymbol{X}^T \times \boldsymbol{Y}$ "*best resembles*" $\boldsymbol{R}$, i.e.:

$$\boldsymbol{R} \approx \boldsymbol{X}^T \times \boldsymbol{Y}$$

- $\boldsymbol{R}$ is typically *sparse* but of *high rank*.
- $\boldsymbol{X}$ and $\boldsymbol{Y}$ are typically *dense* and also of *high rank*.
- Parameter k controls the amount of "**latent features**". These can be thought of as the different types of music tastes represented in the AudioScrobbler data set.

Examples: (using $k = 2$)

Low-rank matrix $\boldsymbol{R}$:

- $$\boldsymbol{R} = \begin{bmatrix} 1 & 2 & 3 & 5 \\ 2 & 4 & 8 & 12 \\ 3 & 6 & 7 & 13 \end{bmatrix} = \underbrace{\begin{bmatrix} 0.76 & 1.05 \\ 2.55 & 2.03 \\ 1.28 & 3.23 \end{bmatrix}}_{\boldsymbol{X}^T} \times \underbrace{\begin{bmatrix} 0.06 & 0.24 & 2.03 & 2.15 \\ 0.90 & 1.72 & 1.37 & 3.17 \end{bmatrix}}_{\boldsymbol{Y}} = \begin{bmatrix} 1.00 & 2.00 & 2.99 & 4.99 \\ 2.00 & 4.12 & 7.96 & 11.94 \\ 2.99 & 5.87 & 7.01 & 13.01 \end{bmatrix}$$

High-rank matrix $\boldsymbol{R}$:

- $$\boldsymbol{R} = \begin{bmatrix} 5 & 8 & 0 & 0 \\ 0 & 0 & 9 & 8 \\ 6 & 0 & 0 & 0 \end{bmatrix} \approx \underbrace{\begin{bmatrix} 1.62 & 1.92 \\ 3.06 & 1.47 \\ 2.12 & 2.07 \end{bmatrix}}_{\boldsymbol{X}^T} \times \underbrace{\begin{bmatrix} 1.52 & 2.01 & 2.46 & 2.06 \\ 1.33 & 2.46 & 0.96 & 1.11 \end{bmatrix}}_{\boldsymbol{Y}} = \begin{bmatrix} 5.01 & 7.98 & 6.47 & 5.81 \\ 6.75 & 10.96 & 8.99 & 7.99 \\ 5.98 & 9.17 & 7.26 & 6.65 \end{bmatrix}$$

See: https://www.youtube.com/watch?v=o8PiWO8C3zs for solving the low-rank example.

Alternating Least Squares (ALS): Objective Function

Solving a set of linear equations even for the low-rank case is quite cumbersome.

- If k is less than the rank of matrix $\boldsymbol{R}$, then there is no exact solution at all, i.e., the resulting factorization is going to be "lossy".
- The resulting approximation of $\boldsymbol{R}$ however has the desired effect that $r_{ui} \approx x_u^T y_i$ can provide also *new recommendations* for user u with respect to artist i.

$$X = \begin{bmatrix} | & & | \\ x_1 & \cdots & x_n \\ | & & | \end{bmatrix}, Y = \begin{bmatrix} | & & | \\ y_1 & \cdots & y_m \\ | & & | \end{bmatrix}$$

Idea:

- *Fix* the **two hyper-parameters** k (e.g., 10 to 100 limiting X, Y) and λ (e.g., 0 to 1)
- *Minimize*, with some degree of freedom, the following **objective function**:

$$\min_{X,Y} \sum_{r_{ui}\ observed} (r_{ui} - x_u^\intercal y_i)^2 + \lambda \left(\sum_u \|x_u\|^2 + \sum_i \|y_i\|^2 \right) \quad (1)$$

Approximation of $\boldsymbol{R} \approx \boldsymbol{X}^T \times \boldsymbol{Y}$

"Regularization" with parameter λ

Iterative ALS Algorithm (Single Machine)

Algorithm 1 ALS for Matrix Completion

Initialize X, Y

repeat

for $u = 1 \dots n$ **do**

$$x_u = \Big(\sum_{r_{ui} \in r_{u*}} y_i y_i^\intercal + \lambda I_k \Big)^{-1} \sum_{r_{ui} \in r_{u*}} r_{ui} y_i \tag{2}$$

end for

for $i = 1 \dots m$ **do**

$$y_i = \Big(\sum_{r_{ui} \in r_{*i}} x_u x_u^\intercal + \lambda I_k \Big)^{-1} \sum_{r_{ui} \in r_{*i}} r_{ui} x_u \tag{3}$$

end for

until convergence

(repeated either until convergence or for a fixed number of iterations)

- Update costs of each x_u: $O(n_u k^2 + k^3)$ where user u has listened to n_u artists
- Update costs of each y_i: $O(n_i k^2 + k^3)$ where artist i has been listened to by n_i users

Distributed ALS: Method 1

By using **fully distributed** encodings of the three matrices as RDDs:

$$\boldsymbol{R}: RDD\big((u, i, r_{ui}), \ldots\big)$$
$$\boldsymbol{X}: RDD(x_1, \ldots, x_n)$$
$$\boldsymbol{Y}: RDD(y_1, \ldots, y_m)$$

To compute Eq. (2) (see previous slide)

- Join $\boldsymbol{R}$ with $\boldsymbol{Y}$ on i (the items/artists to be recommended)
- Map each y_i into $y_i y_i^T$ and change the key to u (the users)
- ReduceByKey on u to compute $\sum_i y_i y_i^T$ and invert the resulting matrix
- ReduceByKey on u to compute $\sum_i r_{ui} y_i$
- Join & Map to multiply the latter two RDDs

Perform a similar series of transformations also to compute Eq. (3)

Repeat until convergence.

Distributed ALS: Method 2

By using one RDD for $\boldsymbol{R}$ and by **broadcasting** the local matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$:

$$\boldsymbol{R}\text{: } RDD\big((u, i, r_{ui}), \dots\big)$$

Partition $\boldsymbol{R}$ into

- $\boldsymbol{R}_{1,1}, \dots, \boldsymbol{R}_{1,p}$ by same users u and
- $\boldsymbol{R}_{2,1}, \dots, \boldsymbol{R}_{2,p}$ by same items/artists i

such that each partition p is assigned to one worker node.

Repeat

- Broadcast $\boldsymbol{X}$ and $\boldsymbol{Y}$ among all worker nodes
- MapPartitions on $\boldsymbol{R}_{1,1}, \dots, \boldsymbol{R}_{1,p}$ and $\boldsymbol{Y}$ to compute Eq. (2)
- MapPartitions on $\boldsymbol{R}_{2,1}, \dots, \boldsymbol{R}_{2,p}$ and $\boldsymbol{X}$ to compute Eq. (3)

until convergence.

(Caution: broadcasting $\boldsymbol{X}, \boldsymbol{Y}$ is tricky in Spark because broadcast variables are immutable!)

Distributed ALS Algorithm: Runtime Comparison

System	Wall-clock /me (seconds)
MATLAB	15,443
Mahout	4,206
GraphLab	291
MLlib	481

- Dataset: scaled version of Netflix data (9X in size!)
- Cluster: 9 machines
- MLlib is an order of magnitude faster than Mahout (based on Hadoop)
- MLlib is within factor of 2 of GraphLab

3.2 Recommender Systems in Spark's MLlib

Load the Data Set

- Copy artist_alias.txt, artist_data.txt, and user_artist_data.txt into either your local directory or your home directory on IRIS. We will assume you are using your local home directory for the following steps.
- To run ALS from your Scala implementation, specify --driver-memory 4g to make sure Spark reserves enough main memory to store your repeatedly read RDD's from cache.

  spark-shell --driver-memory 4g

- Start by reading the three data files into one flat RDD each:

```
val rawArtistAlias = sc.textFile("./artist_alias.txt")
val rawArtistData = sc.textFile("./artist_data.txt")
val rawUserArtistData = sc.textFile("./user_artist_data.txt")
```

- The latter file user_artist_data.txt is by far the largest file with 426 MB.

Explore the Data Set: Get Some Basic Statistics

- Spark's ALS implementation uses 32-bit integers to represent row and column indices. That is, no entries in user_artist_data.txt may be larger than `Integer.MAX_VALUE`, which is $2^{32-1}-1 = 2147483647$ for a signed integer.
- First, explore the range of values in user_artist_data.txt:

```
rawUserArtistData.map(_.split(' ')(0).toDouble).stats()
rawUserArtistData.map(_.split(' ')(1).toDouble).stats()
```

- Here, `split()` splits each line by the provided truncation character(s).
- `stats()` provides a concise summary about the distribution of the values within the provided list (including the maximum and minimum value).

Prepare the Data Set: Parsing & Error Handling

- Prepare a PairRDD, called `artistByID`, that holds the mappings from artist ids (Int) to their actual names (String):

```
val artistByID = rawArtistData.map { line =>
  val (id, name) = line.span(_ != '\t') (id.toInt, name.trim) }
```

- Here, `span()` attempts to split each line by its first tab. It then parses the first portion as the numeric artist id and retains the rest as the name.

- However, a small number of the lines are corrupt, such that either no id or no name can be parsed. So here is an improved version:

```
val artistByID = rawArtistData.flatMap {
  line => val (id, name) = line.span(_ != '\t')
  if (name.isEmpty) { None } else {
    try {
      Some((id.toInt, name.trim)) }
    catch { case e: NumberFormatException => None }
} }
```

Prepare the Data Set: Resolving Aliases

- The artist_alias.txt file contains two ids per line, again separated by a tab. This file is relatively small, only containing about 200,000 entries.
- Both ids are indeed used in the artist_data.txt and user_artist_data.txt file, but they both represent the same real-world artist.
- Hence, it will be useful to collect these aliases into a local map, called `artistAlias`, thus mapping "bad" artist ids to "good" ones.
- Again, some lines are missing the first id, and thus are skipped:

```
val artistAlias = rawArtistAlias.flatMap { line =>
  val tokens = line.split('\t')
  if (tokens(0).isEmpty) { None } else {
    Some((tokens(0).toInt, tokens(1).toInt))
  } }.collectAsMap()
```

- Also check out some of these actual aliases:

```
artistByID.lookup(6803336).head
artistByID.lookup(1000010).head
```

Broadcasting Closure Variables

- Recall that the **closure of a function** contains all the data structures within the current scope of the runtime environment that are referenced by the function's body.
- That is, running a distributed computation with a map function that references additional data structures requires Spark to **serialize the entire function's closure into a binary format** and then **repeatedly ship** this closure **for every task** that is processed by the function (e.g., one line of artist_user_data.txt).
- This is exactly the use-case for a **broadcast variable** (see Chapter 1.2):

```
val bArtistAlias = sc.broadcast(artistAlias)
```

- The broadcast variable `bArtistAlias` is **shipped only once for every executor** of a map function (e.g., based on the 6 to 7 file splits we expect to obtain for artist_user_data.txt).

Prepare & Cache the Training Data

- Our final preparation of the training data consists of

1. mapping all artist ids to their canonical ids (the second column from artist_alias.txt), and
2. transforming `rawUserArtistData` into a `Rating` data structure, which is the default data structure used by Spark to hold user-artist-rating triples.

```
import org.apache.spark.mllib.recommendation._

val trainData = rawUserArtistData.map {
  line => val Array(userID, artistID, count) =
    line.split(' ').map(_.toInt)
  val finalArtistID = bArtistAlias.value.getOrElse(artistID,
    artistID)
  Rating(userID, finalArtistID, count)
}.cache()
```

- The training data is used many times by the ALS algorithm, and hence it should be cached!

Train a First Recommender Model

- Trigger the ALS computation in Spark using the `trainData` RDD:

```
val model = ALS.trainImplicit(trainData, 10, 5, 0.01, 1.0)
```

- Note:
  - Running ALS may take a couple of minutes when using the entire AudioScrobbler training set on the IRIS HPC cluster!
  - Thus, always use a smaller training set (e.g., use `.sample(false, 0.01)`) to first debug your algorithms!

- Inspect the resulting model:

```
model.userFeatures.mapValues(_.mkString(", ")).first()
```

(which returns nothing else but the first row of the $\boldsymbol{X}$ matrix in the factorization)

See https://spark.apache.org/docs/latest/mllib-collaborative-filtering.html for a more comprehensive API documentation of the `org.apache.spark.mllib.recommendation` package.

Spot-Checking Users & Recommendations (I)

- Verify the viability of your model by comparing a user's actually played artists with the recommendations we would obtain from the model:

```
val rawArtistsForUser = rawUserArtistData.map(_.split(' ')).
  filter { case Array(user,_,_) => user.toInt == 2093760 }
val existingArtists =
  rawArtistsForUser.map {case Array(_,artist,_) => artist.toInt}
    .collect().toSet
artistByID.filter {case (id, name) =>
  existingArtists.contains(id)}.values.collect().foreach(println)
```

- And compare these to the top-5 recommended artists for this user:

```
val recommendations = model.recommendProducts(2093760, 5)
recommendations.foreach(println)
```

Note: the new ratings exclude artists that the user actually played already!

Spot-Checking Users & Recommendations (II)

- Finally, inspect the top-5 recommendations by looking up the artists' names in the `artistByID` map.

```
val recommendedArtistIDs = recommendations.map(_.product).toSet
artistByID.filter { case (id, name) =>
   recommendedArtistIDs.contains(id)
}.values.collect().foreach(println)
```

After this step, you may want to check the progress and current memory consumption of your running Spark jobs: http://localhost:4040

- Enable port forwarding from an IRIS node to your localhost:
  - ssh -p 8022 <student-id>@access-iris.uni.lu
  - srun -p batch --time=00:30:0 -N 2 -c 12 --pty bash –i (remember the node id!)
  - ./launch-spark-shell.sh
- Then open a second terminal and replace XXX with the id of your IRIS node:
  - ssh -p 8022 -L 4040:iris-XXX.iris-cluster.uni.lux:4040 <student-id>@access-iris.uni.lu

Evaluating Recommendation Quality: ROC & AUC

Receiver Operating Characteristic (ROC)

- Suppose we process a ranked list of results from top to end.
- At each rank *r*, we measure the **true positive rate** (the number of correctly returned items at rank *r*) and the **false positive rate** (the number of erroneously returned items at rank *r*).
- The function obtained by plotting these points over all ranks is called the **receiver operator characteristic**.

Area Under the Curve (AUC)

- The **area under the ROC curve** is a typical measure for the goodness of a recommender system. It is high (i.e., approaching 1) if a system returns more correct items than erroneous items. It is 0.5 if results are random (correct and erroneous items are returned interchangeably at the ranks).
- Since ROC is not a smooth (i.e., differentiable) function, we usually approximate it by **averaging over various ROC points**.

Receiver-Operator Statistic (ROC)

The so-called "**Receiver-Operator Characteristics**" (ROC) plots the true-positive rate (TPR) against the false-positive rate (FPR) of a system.

TPR (i.e., the relative size of TP at a given rank) is also referred to as $sensitivity$, which is identical to $recall$, and describes how good a system is at returning the relevant items.

FPR (i.e., the relative size of FP at a given rank) is also referred to as $1 - specificity$, where $specificity$ describes how good a system is at avoiding the non-relevant items.

- Formally, we have:

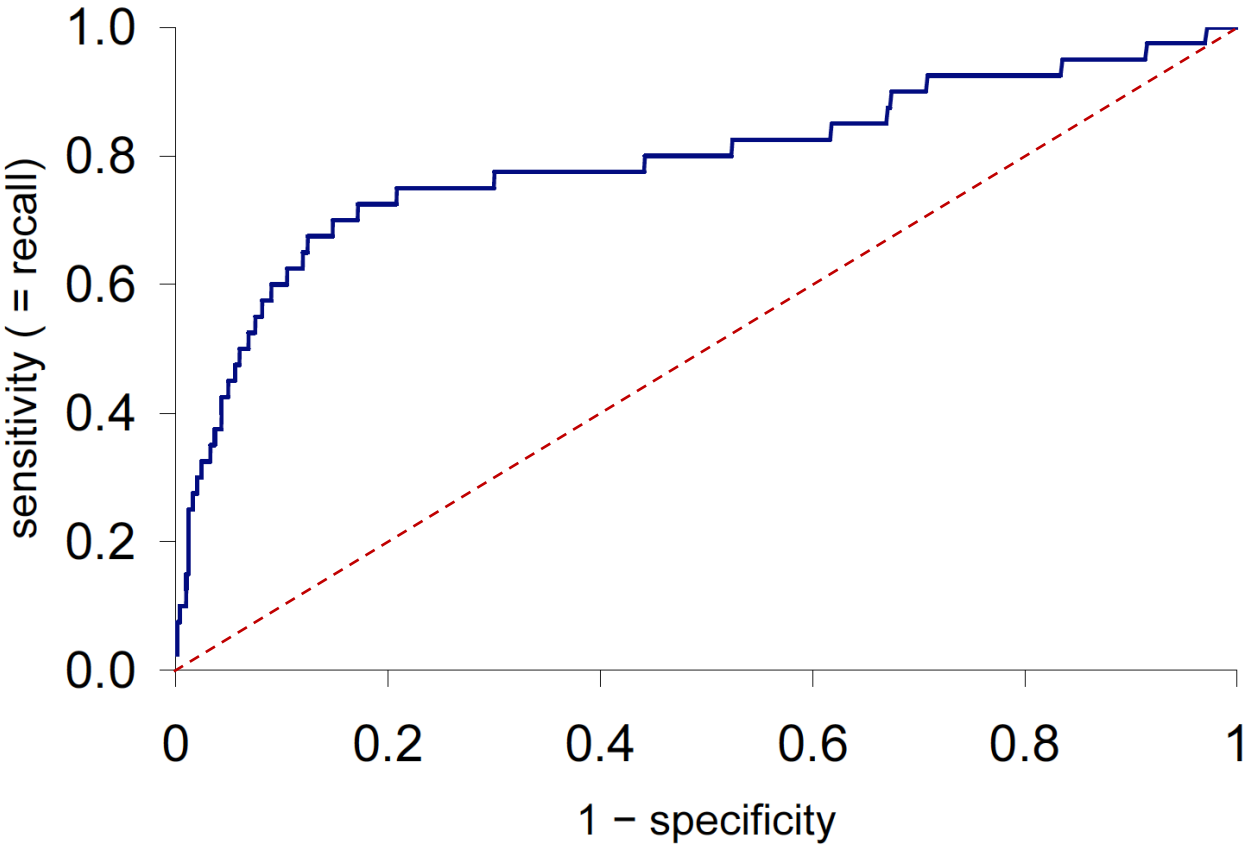
$$\mathrm{TPR} = sensitivity = \frac{|TP|}{|TP|+|FN|}$$

$$\mathrm{FPR} = 1 - specificity = 1 - \frac{|TN|}{|FP|+|TN|} = \frac{|FP|}{|FP|+|TN|}$$

The **Area-Under-the-Curve** (AUC) measure then is defined as the average TPR value over a number of predefined FPR points.

See also: https://en.wikipedia.org/wiki/Sensitivity_and_specificity

Illustration of ROC



- An example of a typical ROC curve is shown by the upper blue plot.
- Note that a "random" system, which interchangeably retrieves a relevant and then a non-relevant result, results in a straight line from (0,0) to (1,1).

ROC & AUC Example Calculation

Create new RDD: `predictionsAndLabels`

Rank	Predicted Relevance (ranked)	Actual Relevance (binary)	TPR = $\frac{\lvert TP \rvert}{\lvert TP \rvert + \lvert FN \rvert}$	FPR = $\frac{\lvert FP \rvert}{\lvert FP \rvert + \lvert TN \rvert}$
1	0.43	1	1/(1+6) = 0.14	0/(0+3) = 0.00
2	0.34	1	2/(2+5) = 0.28	0/(0+3) = 0.00
3	0.21	0	2/(2+5) = 0.28	1/(1+2) = 0.33
4	0.17	0	2/(2+5) = 0.28	2/(2+1) = 0.67
5	0.12	1	3/(3+4) = 0.42	2/(2+1) = 0.67
6	0.11	1	4/(4+3) = 0.57	2/(2+1) = 0.67
7	0.08	1	5/(5+2) = 0.71	2/(2+1) = 0.67
8	0.06	1	6/(6+1) = 0.86	2/(2+1) = 0.67
9	0.00	0	6/(6+1) = 0.86	3/(3+0) = 1.00
10	-0.01	1	7/(7+0) = 1.00	3/(3+0) = 1.00

AUC = 0.54

Fortunately, the ROC and AUC measures are readily built into Spark's `BinaryClassificationMetrics` package:
https://spark.apache.org/docs/latest/mllib-evaluation-metrics.html

Evaluating Recommendation Quality: Cross-Validation

Cross-Validation

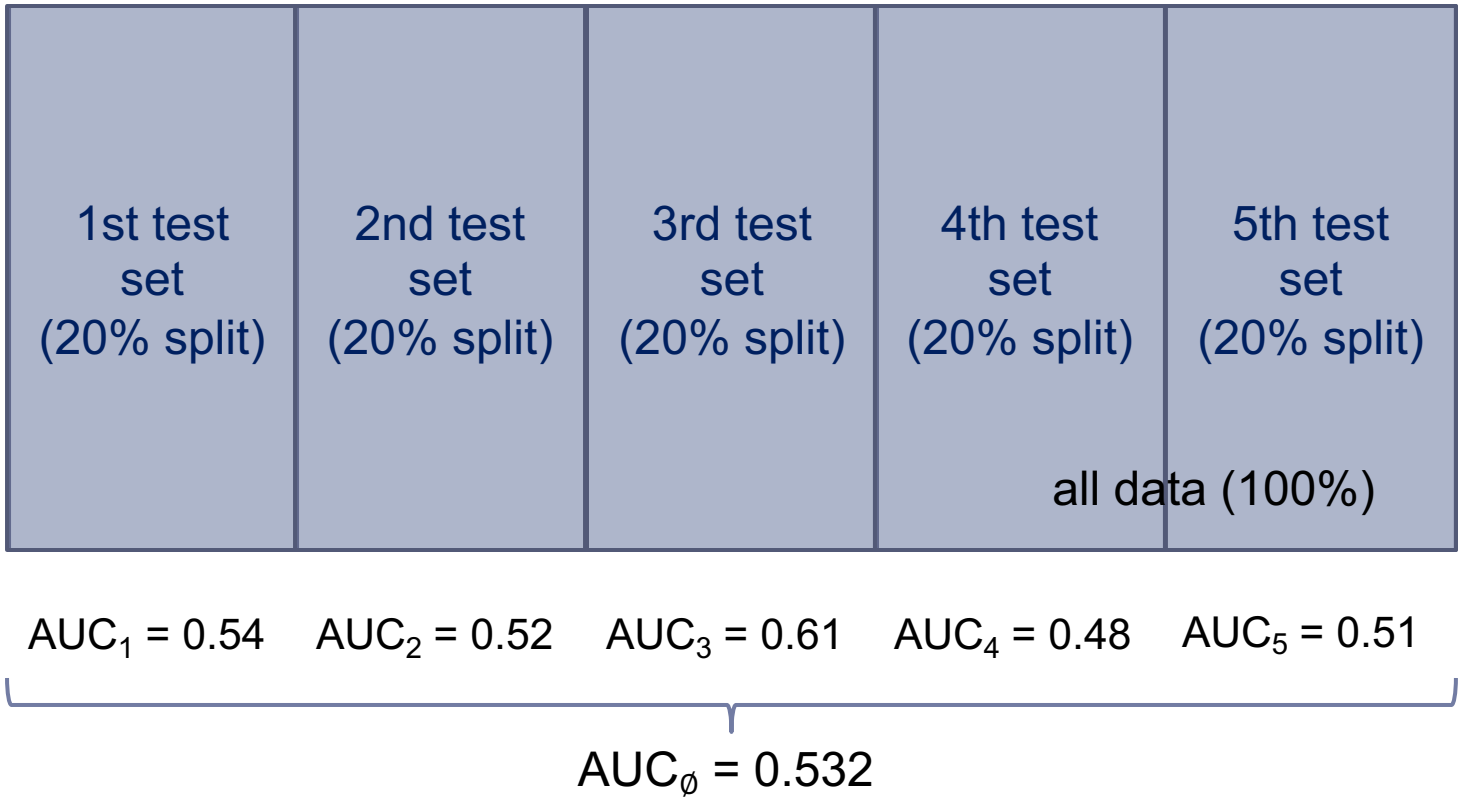
- Withhold a (typically small) fraction (e.g., 90%) of the overall data set for training, use the remaining part (e.g., 10%) for testing the accuracy of the learned model.

K-fold Cross-Validation

- Iterate the cross-validation step *k* times by splitting the overall data set into *k* disjoint partitions.
- Use *k*-1 partitions for training and the remaining partition for testing at each iteration, then subsequently shift the training and testing partitions.
- Finally measure the accuracy or AUC at each split and use the averaged result over the *k* steps to select the best hyper-parameter setting.

K-Fold Cross-Validation Illustration

5-fold cross-validation: 5x split each using 80% for training data and 20% for testing



Fortunately, also cross-validations are readily built into Spark's `CrossValidator` package:

https://spark.apache.org/docs/latest/ml-tuning.html

Baseline Selection

- Similarly to our earlier example about Decision Trees, we may also want to compare our new recommender model to a **reasonable baseline**.
- So how good is a system that would simply recommend the same "*most frequently listened to*" artists to every user?

```
val artistsTotalCount = trainData.map(r => (r.product,
  r.rating)).reduceByKey(_ + _).collect().sortBy(-_._2)
def predictMostPopular(user: Int, numArtists: Int) = {
  val topArtists = artistsTotalCount.take(numArtists)
  topArtists.map{case (artist, rating) => Rating(user,
     artist, rating)}
}
val auc = ... // see RunRecommender-shell.sh on Moodle!
```

- Note: `predictMostPopular(...)` thus creates artificial `Rating` objects based on the total number of times all users listened to the artists!
- Here, `predictMostListened` even yields an AUC value of about 0.930!

Hyper-Parameters of Spark's `MatrixFactorizationModel`

So far, the so-called **hyper-parameters** used to build the `MatrixFactorization-Model` were simply given. They are not learned by the algorithm and must be chosen by the caller. The following arguments are used by the function:

```
ALS.trainImplicit(trainData, rank, iterations, lambda, alpha)
```

- `rank = 10`
  The number of latent factors in the model, or equivalently, the number of columns *k* in the user-feature and product-feature matrices. In nontrivial cases, this is also their rank.
- `iterations = 5`
  The number of iterations that the factorization runs. More iterations take more time but may produce a better factorization.
- `lambda = 0.01`
  A standard regularization parameter. Higher values resist overfitting, but values that are too high hurt the factorization's accuracy.
- `alpha = 1.0`
  Controls the relative weight of observed versus unobserved user-product interactions in the factorization (internal Spark parameter).

Hyper-Parameter Tuning

- Using the idea of cross-validation, we now have a powerful approach to systematically investigate the **hyper-parameter space**:

```
val evaluations = for(rank <- Array(10, 50);
  lambda <- Array(1.0, 0.0001);
  alpha <- Array(1.0, 40.0))
yield {
 val model = ALS.trainImplicit(trainData, rank, 10, lambda, alpha)
 val auc = ... // see RunRecommender-shell.sh on Moodle!
 ((rank, lambda, alpha), auc)
}
evaluations.sortBy(_._2).reverse.foreach(println)
```

- Here, we set `rank` to `{10, 50}`, `lambda` to `{1.0, 0.0001}` and `alpha` to `{1, 40}`, hence we obtain 8 combinations of parameters.
- The resulting combination of `rank=10`, `lambda=1.0` and `alpha=40.0` yields an average AUC value of 0.976 over k = 10 cross-validations.

Creating Batch Recommendations

- Spark's ALS implementation only allows one recommendation at a time.
- However, we can still slightly improve the performance for making recommendations to multiple users by **batch-processing** them from within a preselected list of users:

```
val someUsers = trainData.map(x => x.user).distinct().take(10)
val someRecommendations = someUsers.map(userID =>
  model.recommendProducts(userID, 5))
someRecommendations.map(recs => recs.head.user + " -> " +
  recs.map(_.product).mkString(", ")).foreach(println)
```

Extensions

- **Fix Data Errors**
  - Improve exception handling for more cases of errors: can some lines still be saved?
- **Remove Obvious Outliers**
  - Some users may have played suspiciously many artists; while some artists are played by suspiciously many users.
  - Often, removing the most frequent and/or infrequent patterns from the data set may help to improve the prediction accuracy.
- **More Tuning**
  - Systematically explore a larger parameter space.
  - Use ensemble techniques, e.g., both collaborative and content-based filtering.

→ Compare also to the "Data Science Workflow" slide from Chapter 2!

Big Data Analytics

Chapter 4: Text Processing with Latent Semantic Analysis

Following: [3] "**Advanced Analytics with Spark**", Chapter 6

Prof. Dr. Martin Theobald

Faculty of Science, Technology & Communication

University of Luxembourg

How to Analyze Text Data?

Unlike the previous data sets, text data is often considered as "**unstructured data**" – although this is far from actually being true:

- In fact, natural language follows a very fine-grained structure, with a well-defined grammar and usually a clear intention of an author in describing a possibly complex matter. However, this structure still is quite difficult to process for a machine…

Thus, in the following we focus on a very **basic semantic analysis** of words occurring in documents based on **large-scale co-occurrence statistics**.

We will represent these statistics in the form of a **large** (and usually very **sparse**) **matrix**, where the rows denote words and the columns denote documents.

The cell entries of the matrix are usually based on **TF-IDF weights**:

- The **term frequency** (TF) of a word is the number of times the word (or its lemma) occurs within a document.
- The **document frequency** (DF) of a word is the number of documents that contain the word (or its lemma).
- This coincides with a so-called **bag-of-words representation** for documents in many Information Retrieval and Data Mining applications.

The Wikipedia Data Set

- We have extracted 500 MB of **English Wikipedia articles** from a recent snapshot and converted the resulting 41,784 articles into a plain-text (UTF-8) format.
- The dump is split into 1,060 individual files, each of about 500 KB size.
- Each such split is of the following format:

```
<doc id="38298" url="https://en.wikipedia.org/wiki?curid=38298"
        title="George Boole">
         George Boole was an English mathematician, educator, philosopher
        and logician. ...
<doc>
```

4.1 Latent Semantic Analysis

Latent Semantic Analysis vs. Latent Semantic Indexing

Latent Semantic Analysis (LSA) and **-Indexing** (LSI) are mostly synonymous.

- The general technique of applying a Singular Value Decomposition (SVD) to a **word-document matrix** $\boldsymbol{A}$ is called Latent Semantic Analysis (LSA).
- By applying SVD to $\boldsymbol{A}$, similar words are mapped to their **implicit** (i.e., "latent") **meanings**, which are represented as a k-dimensional vector for each word.
- Conversely, documents are also represented by k-dimensional vectors, which each represent the strength of the **latent concepts** the documents contains.
- The usage of LSA for **indexing documents** in the context of **Information Retrieval** and/or **Data Mining** is usually called Latent Semantic Indexing.

Basic LSI Idea:

- When given the following documents:

  $\boldsymbol{d}_1 = "Jaguar\ animal\ cat"$

  $\boldsymbol{d}_2 = "Jaguar\ car\ brand"$

  $\boldsymbol{d}_3 = "Jaguar\ luxury\ vehicle"$

- The query $\boldsymbol{q} = "car\ brand"$ should return $\boldsymbol{d}_2$, $\boldsymbol{d}_3$, $\boldsymbol{d}_1$ (in that particular order).

Singular Value Decomposition

Generally, an **SVD of a word-document matrix $\boldsymbol{A}$** is of the following form:

$$\boldsymbol{A}_{m\times n} = \boldsymbol{U}_{m\times m}\, \boldsymbol{S}_{m\times n}\, \boldsymbol{V}^T{}_{n\times n}$$

- $\boldsymbol{U}$ is an orthogonal matrix (i.e., $\boldsymbol{U}\,\boldsymbol{U}^T = \boldsymbol{I}$) whose columns are orthonormal eigenvectors of $\boldsymbol{A}\,\boldsymbol{A}^T$.
- $\boldsymbol{S}$ is a diagonal matrix containing the square roots of the eigenvalues of $\boldsymbol{U}$ and $\boldsymbol{V}$, respectively, *in descending order*.
- $\boldsymbol{V}^T$ is the transpose of an orthogonal matrix (i.e., $\boldsymbol{V}\,\boldsymbol{V}^T = \boldsymbol{I}$) whose rows are orthonormal eigenvectors of $\boldsymbol{A}^T\boldsymbol{A}$.

Suppose that $\boldsymbol{A}$ is an $m \times n$ matrix, then a **full SVD decomposition** yields an $m \times m$ matrix $\boldsymbol{U}$, an $m \times n$ matrix $\boldsymbol{S}$ and an $n \times n$ matrix $\boldsymbol{V}^T$.

However, by **reducing $\boldsymbol{S}$** to its k largest singular values, we may achieve the desired **dimensionality reduction** also for $\boldsymbol{U}$ and $\boldsymbol{V}^T$:

$$\boldsymbol{A}_{m\times n} \approx \boldsymbol{U}_{m\times k}\, \boldsymbol{S}_{k\times k}\, \boldsymbol{V}^T{}_{k\times n}$$

The goal here usually is not to reconstruct $\boldsymbol{A}$, but to find word-to-word or document-to-document similarities:

- **Word-to-word similarities**: compare rows $\boldsymbol{U}_{m\times k}\, \boldsymbol{S}_{k\times k}$
- **Document-to-document similarities**: compare rows $[\boldsymbol{S}_{k\times k}\, \boldsymbol{V}^T{}_{k\times n}]^T = \boldsymbol{V}_{n\times k}\, \boldsymbol{S}_{k\times k}$

SVD Example: Full Decomposition (I)

$$\boldsymbol{A} = \begin{bmatrix} 2 & 0 & 8 & 6 & 0 \\ 1 & 6 & 0 & 1 & 7 \\ 5 & 0 & 7 & 4 & 0 \\ 7 & 0 & 8 & 5 & 0 \\ 0 & 10 & 0 & 0 & 7 \end{bmatrix} = \begin{bmatrix} -0.54 & 0.07 & 0.82 & -0.11 & 0.12 \\ -0.10 & -0.59 & -0.11 & -0.79 & -0.06 \\ -0.53 & 0.06 & -0.21 & 0.12 & -0.81 \\ -0.65 & 0.07 & -0.51 & 0.06 & 0.56 \\ -0.06 & -0.80 & 0.09 & 0.59 & 0.04 \end{bmatrix}$$

$$\times \begin{bmatrix} 17.92 & 0 & 0 & 0 & 0 \\ 0 & 15.17 & 0 & 0 & 0 \\ 0 & 0 & 3.56 & 0 & 0 \\ 0 & 0 & 0 & 1.98 & 0 \\ 0 & 0 & 0 & 0 & 0.35 \end{bmatrix}$$

$$\times \begin{bmatrix} -0.46 & 0.02 & -0.87 & 0.00 & 0.17 \\ -0.07 & -0.76 & 0.06 & 0.60 & 0.23 \\ -0.74 & 0.10 & 0.28 & 0.22 & -0.56 \\ -0.48 & 0.03 & 0.40 & -0.33 & 0.70 \\ -0.07 & -0.64 & -0.04 & -0.69 & -0.32 \end{bmatrix}$$

- This full matrix decomposition is indeed **lossless**, i.e., the product $\boldsymbol{U}_{m\times m}\,\boldsymbol{S}_{m\times n}\,\boldsymbol{V}^T{}_{n\times n}$ returns again $\boldsymbol{A}_{m\times n}$.

SVD Example: Full Decomposition (II)

- Computing **word-to-word similarities** based on

$$\boldsymbol{W}_{m\times n} = \boldsymbol{U}_{m\times m}\ \boldsymbol{S}_{m\times n} = \begin{bmatrix} -9.72 & 0.99 & 2.93 & -0.21 & 0.04 \\ -1.82 & -9.00 & -0.40 & -1.56 & -0.02 \\ -9.41 & 0.90 & -0.76 & 0.23 & -0.28 \\ -11.56 & 1.07 & -1.81 & 0.12 & 0.20 \\ -1.16 & -12.09 & 0.32 & 1.18 & 0.02 \end{bmatrix}$$

- and using, e.g., **Cosine similarity**

$$\mathrm{CosSim}\left(\boldsymbol{w}_i, \boldsymbol{w}_j\right) = \frac{\boldsymbol{w}_i \cdot \boldsymbol{w}_j}{\left\|\boldsymbol{w}_i\right\|_2 \left\|\boldsymbol{w}_j\right\|_2}$$

- we get:

$$\begin{aligned} \mathrm{CosSim}(\boldsymbol{w}_1, \boldsymbol{w}_1) &= 1.00 \\ \mathrm{CosSim}(\boldsymbol{w}_1, \boldsymbol{w}_2) &= 0.08 \\ \mathrm{CosSim}(\boldsymbol{w}_1, \boldsymbol{w}_3) &= 0.93 \\ \mathrm{CosSim}(\boldsymbol{w}_1, \boldsymbol{w}_4) &= 0.90 \\ \mathrm{CosSim}(\boldsymbol{w}_1, \boldsymbol{w}_5) &= 0.00 \end{aligned}$$

- These values coincide with the pairwise row distances in $\boldsymbol{A}$.

SVD Example: Full Decomposition (III)

- Computing **document-to-document similarities** based on

$$\boldsymbol{D}_{n\times m} = [\boldsymbol{S}_{m\times n}\, \boldsymbol{V}^T{}_{n\times n}]^T = \begin{bmatrix} -8.33 & -0.33 & -3.10 & -0.00 & 0.06 \\ -1.26 & -11.53 & 0.22 & 1.19 & 0.08 \\ -13.17 & 1.50 & 1.01 & 0.44 & -0.20 \\ -8.68 & 0.39 & 1.42 & -0.66 & 0.25 \\ -1.16 & -9.73 & -0.16 & -1.37 & -0.11 \end{bmatrix}$$

- and again using **Cosine similarity**

$$\mathrm{CosSim}(\boldsymbol{d}_i, \boldsymbol{d}_j) = \frac{\boldsymbol{d}_i \cdot \boldsymbol{d}_j}{\left\|\boldsymbol{d}_i\right\|_2 \left\|\boldsymbol{d}_j\right\|_2}$$

- we get:

$$\begin{aligned} \mathrm{CosSim}(\boldsymbol{d}_1, \boldsymbol{d}_1) &= 1.00 \\ \mathrm{CosSim}(\boldsymbol{d}_1, \boldsymbol{d}_2) &= 0.06 \\ \mathrm{CosSim}(\boldsymbol{d}_1, \boldsymbol{d}_3) &= 0.90 \\ \mathrm{CosSim}(\boldsymbol{d}_1, \boldsymbol{d}_4) &= 0.87 \\ \mathrm{CosSim}(\boldsymbol{d}_1, \boldsymbol{d}_5) &= 0.08 \end{aligned}$$

- These values in turn coincide with the pairwise column distances in $\boldsymbol{A}$.

SVD Example: Reduced Decomposition (I)

$$k = 3$$

$$\boldsymbol{A} = \begin{bmatrix} 2 & 0 & 8 & 6 & 0 \\ 1 & 6 & 0 & 1 & 7 \\ 5 & 0 & 7 & 4 & 0 \\ 7 & 0 & 8 & 5 & 0 \\ 0 & 10 & 0 & 0 & 7 \end{bmatrix} \approx \left[\begin{array}{ccc|cc} -0.54 & 0.07 & 0.82 & -0.11 & 0.12 \\ -0.10 & -0.59 & -0.11 & -0.79 & -0.06 \\ -0.53 & 0.06 & -0.21 & 0.12 & -0.81 \\ -0.65 & 0.07 & -0.51 & 0.06 & 0.56 \\ -0.06 & -0.80 & 0.09 & 0.59 & 0.04 \end{array}\right]$$

$$\times \left[\begin{array}{ccc|cc} 17.92 & 0 & 0 & 0 & 0 \\ 0 & 15.17 & 0 & 0 & 0 \\ 0 & 0 & 3.56 & 0 & 0 \\ \hline 0 & 0 & 0 & 1.98 & 0 \\ 0 & 0 & 0 & 0 & 0.35 \end{array}\right]$$

$$\times \begin{bmatrix} -0.46 & 0.02 & -0.87 & 0.00 & 0.17 \\ -0.07 & -0.76 & 0.06 & 0.60 & 0.23 \\ -0.74 & 0.10 & 0.28 & 0.22 & -0.56 \\ -0.48 & 0.03 & 0.40 & -0.33 & 0.70 \\ -0.07 & -0.64 & -0.04 & -0.69 & -0.32 \end{bmatrix}$$

- This reduced matrix decomposition now becomes "**lossy**", i.e., the product $\boldsymbol{U}_{m\times k}\,\boldsymbol{S}_{k\times k}\,\boldsymbol{V}^T{}_{k\times n}$ returns only an approximation of $\boldsymbol{A}_{m\times n}$.

SVD Example: Reduced Decomposition (II)

- Computing **word-to-word similarities** based on

$$k = 3$$

$$\boldsymbol{W}_{m\times k} = \boldsymbol{U}_{m\times k}\ \boldsymbol{S}_{k\times k} = \begin{bmatrix} -9.72 & 0.99 & 2.93 \\ -1.82 & -9.00 & -0.40 \\ -9.41 & 0.90 & -0.76 \\ -11.56 & 1.07 & -1.81 \\ -1.16 & -12.09 & 0.32 \end{bmatrix}$$

- and using, e.g., **Cosine similarity**

$$\mathrm{CosSim}(w_i, w_j) = \frac{\boldsymbol{w}_i \cdot \boldsymbol{w}_j}{\|\boldsymbol{w}_i\|_2 \|\boldsymbol{w}_j\|_2}$$

- we get:

$$\mathrm{CosSim}(\boldsymbol{w}_1, \boldsymbol{w}_1) = 1.00$$
$$\mathrm{CosSim}(\boldsymbol{w}_1, \boldsymbol{w}_2) = 0.08$$
$$\mathrm{CosSim}(\boldsymbol{w}_1, \boldsymbol{w}_3) = 0.93$$
$$\mathrm{CosSim}(\boldsymbol{w}_1, \boldsymbol{w}_4) = 0.90$$
$$\mathrm{CosSim}(\boldsymbol{w}_1, \boldsymbol{w}_5) = 0.00$$

- These values now approximate the pairwise row distances in $\boldsymbol{A}$.

SVD Example: Reduced Decomposition (III)

- Computing **document-to-document similarities** based on

$$k = 3$$

$$\boldsymbol{D}_{n\times k} = [\boldsymbol{S}_{k\times k}\, \boldsymbol{V}^T{}_{k\times n}]^T = \begin{bmatrix} -8.33 & 1.06 & -2.62 \\ 0.39 & -11.53 & 0.35 \\ -15.56 & 0.96 & 1.01 \\ -0.01 & 9.12 & 0.80 \\ 3.07 & 3.46 & -2.01 \end{bmatrix}$$

- and again using **Cosine similarity**

$$\mathrm{CosSim}(\boldsymbol{d}_i, \boldsymbol{d}_j) = \frac{\boldsymbol{d}_i \cdot \boldsymbol{d}_j}{\|\boldsymbol{d}_i\|_2 \|\boldsymbol{d}_j\|_2}$$

- we get:

$$\mathrm{CosSim}(\boldsymbol{d}_1, \boldsymbol{d}_1) = 1.00$$
$$\mathrm{CosSim}(\boldsymbol{d}_1, \boldsymbol{d}_2) = 0.08$$
$$\mathrm{CosSim}(\boldsymbol{d}_1, \boldsymbol{d}_3) = 0.92$$
$$\mathrm{CosSim}(\boldsymbol{d}_1, \boldsymbol{d}_4) = -0.14$$
$$\mathrm{CosSim}(\boldsymbol{d}_1, \boldsymbol{d}_5) = -0.54$$

- These values now approximate the pairwise column distances in $\boldsymbol{A}$.

Processing Queries

- A **new query vector** $\boldsymbol{q}_{1\times m}$ can be processed against the reduced decomposition by transforming it in a similar manner, namely by computing

$$\boldsymbol{q'}_{1\times k} = \boldsymbol{q}_{1\times m} \times \boldsymbol{U}_{m\times k} \times \boldsymbol{S}^{-1}_{k\times k}$$

- and then by computing the pairwise Cosine similarities of $\boldsymbol{q}_{1\times k}$ with the row vectors in $\boldsymbol{D}_{n\times k}$ which each represents a reduced document vector.

Example:

- For the query vector $\boldsymbol{q}_{1\times m} = [1,3,0,2,0]$, we obtain the **transformed and reduced query vector** $\boldsymbol{q'}_{1\times k} = [-0.12, -0.10, -0.15]$ with the following Cosine similarities to the row vectors in $\boldsymbol{D}_{n\times k} = [\boldsymbol{S}_{k\times k}\, \boldsymbol{V^T}_{k\times n}]^T$ (see previous slide):

$$\begin{aligned}
\text{CosSim}(\boldsymbol{q'}, \boldsymbol{d}_1) &= 0.78 \\
\text{CosSim}(\boldsymbol{q'}, \boldsymbol{d}_2) &= 0.44 \\
\text{CosSim}(\boldsymbol{q'}, \boldsymbol{d}_3) &= 0.47 \\
\text{CosSim}(\boldsymbol{q'}, \boldsymbol{d}_4) &= -0.53 \\
\text{CosSim}(\boldsymbol{q'}, \boldsymbol{d}_5) &= -0.39
\end{aligned}$$

$k = 3$

Finding an SVD of a Word-Document Matrix (I)

That is, just like the matrix factorization shown in Chapter 3, we can compute the SVD of a matrix $\boldsymbol{A}$ by **solving a set of linear equations**:

1. Compute the k largest (non-zero) eigenvalues of either $\boldsymbol{A}\,\boldsymbol{A}^T$ or $\boldsymbol{A}^T\boldsymbol{A}$.
2. To find $\boldsymbol{U}$:
   - Use these eigenvalues to compute k eigenvectors of $\boldsymbol{A}\,\boldsymbol{A}^T$.
   - Turn each eigenvector into a column vector of a new matrix $\boldsymbol{U}'$.
   - Convert $\boldsymbol{U}'$ into the orthogonal matrix $\boldsymbol{U}$ (e.g., using the Gram–Schmidt procedure)
3. To find $\boldsymbol{V}^T$:
   - Use these eigenvalues to compute k eigenvectors of $\boldsymbol{A}^T\boldsymbol{A}$.
   - Turn each eigenvector into a row vector of a new matrix $\boldsymbol{V}^{T\prime}$.
   - Convert $\boldsymbol{V}^{T\prime}$ into the orthogonal matrix $\boldsymbol{V}^T$ (e.g., using the Gram–Schmidt procedure)
4. To find $\boldsymbol{S}$:
   - Populate a new $k \times k$ diagonal matrix with the square roots of the non-zero eigenvalues computed in step 1.

Finding an SVD of a Word-Document Matrix (II)

- Spark imports a number of low-level Fortran libraries for matrix operations (ARPACK) that provide different numerical optimizations also for **approximate eigenvalue decompositions**, including SVD.
- See also here for a **tutorial on LSA** via SVD and eigenvalue decomposition.
- See the previous chapter on possible ways to implement **distributed matrix multiplications** (Methods 1 & 2) in Spark.

4.2 Text Analysis & Singular-Value-Decomposition in Spark's MLlib

Loading the Data Set (I)

- First of all, we need to define a special loading function for our new Wikipedia file format.
- Recall that a Wikipedia article is delimited by a pair of `<doc …>` and `</doc>` tags, such that a simple line-based input format would not guarantee that consecutive lines in the input documents are also stored as consecutive lines generated by the `sc.textFile(…)` RDD.
- On the other hand, we do not want to assume that all Wikipedia files fit into the main-memory of our driver node, hence we should keep working with partitioned RDDs from the beginning.
- Fortunately, there is an alternative function that loads whole text files into an RDD:

```
sc.wholeTextFiles("./pathToTextFiles")
```

Loading the Data Set (II)

- We can then transform the RDD with the input files (each containing many Wikipedia articles) into a new RDD containing **one entry per Wikipedia article** by a single `flatMap` transformation:

```
val textFiles = sc.wholeTextFiles("./path-to-wiki-articles")
val plainText = textFiles.flatMap{ case (uri, text) =>
   parse(text.split("\n")) } // contains one entry for each
                                    Wikipedia article
```

- Note that the `parse` function (see Moodle) is executed on all partitions of input files **in parallel**.

- This way, we do **not need to assume that all articles fit into the main memory of our driver node** at any time. Every processing step is performed only via RDD transformations!

Create a Default NLP Pipeline

- Our basic **natural-language-processing** (NLP) **pipeline** consists of:
  - Sentence splitting
  - Tokenization
  - Part-Of-Speech (POS) tagging
  - Lemmatization

- The **Stanford CoreNLP tools** provide a very comprehensive API for these (and other) NLP tasks:

```
import edu.stanford.nlp.pipeline._
import edu.stanford.nlp.ling.CoreAnnotations._

def createNLPPipeline(): StanfordCoreNLP = {
  val props = new Properties()
  props.put("annotators", "tokenize, ssplit, pos, lemma")
  new StanfordCoreNLP(props)
}
```

Filtering Out Non-Letter Tokens & Stopwords

- For a more compact dictionary of words, we may still want to filter out tokens that are **not proper sequences of letters** and very frequent words, the so-called **stopwords**.

```
def isOnlyLetters(str: String): Boolean = {
  str.forall(c => Character.isLetter(c)) }
```

- Stopwords are extremely frequent words of a language, such as **prepositions**, **articles**, **auxiliary verbs**, etc., that do not add much information to our semantic indexing approach.

```
import scala.io.Source._
val bStopWords = sc.broadcast(
  fromFile("stopwords.txt").getLines().toSet)
```

- A respective file with English stopwords is available on Moodle.

From Text to Word Lemmas

- An input text of type `String` should be tokenized and the words should be reduced to their lemmatized form:

```
def plainTextToLemmas(text: String,
    pipeline: StanfordCoreNLP): Seq[String] = {
  val doc = new Annotation(text)
  pipeline.annotate(doc)
  val lemmas = new ArrayBuffer[String]()
  val sentences = doc.get(classOf[SentencesAnnotation])
  for (sentence <- sentences;
      token <- sentence.get(classOf[TokensAnnotation])) {
    val lemma = token.get(classOf[LemmaAnnotation])
    if (lemma.length > 2 && !bStopWords.value.contains(lemma)
        && isOnlyLetters(lemma)) {
      lemmas += lemma.toLowerCase } }
  lemmas }
```

Initialize the NLP Pipelines

- Use a `mapPartitions` transformation on the `plainText` RDD so that we only **initialize** the NLP pipeline object **once per partition** instead of once per document:

```
val lemmatized: RDD[Seq[String]] =
    plainText.mapPartitions(it => {
  val pipeline = createNLPPipeline()
  it.map { case(title, contents) =>
    plainTextToLemmas(contents, pipeline)
  }
})
```

Computing TF Weights

- The term frequency of a term in a document is a local property of that document. So we can compute it via a **local aggregation** over the documents.

```
import scala.collection.mutable.HashMap

val docTermFreqs = lemmatized.map(terms => {
  val termFreqs = terms.foldLeft(new HashMap[String, Int]()) {
    (map, term) => {
      map += term -> (map.getOrElse(term, 0) + 1)
    map }
  }
  termFreqs
})
```

- For frequent access, the **term-frequency map** should be **cached**.

```
docTermFreqs.cache()
```

Remove Infrequent Terms

- To further reduce the term space, we may run a **standard word count** to filter out infrequent terms (in this case with a document frequency of less than 12):

```
val docFreqs = docTermFreqs.flatMap(_.keySet).map((_, 1)).
  reduceByKey(_ + _, 12)
```

- Keep only the **top 50,000 terms** from within all documents:

```
val ordering = Ordering.by[(String, Int), Int](_._2)
val topDocFreqs = docFreqs.top(50000)(ordering)
```

Computing DF Weights (I)

The document frequency of a term across all documents is a global property of the collection. Since we are operating over the partitioned data structure `docTermFreqs`, we need to aggregate it in two steps:

1. We count the **document frequencies locally for each partition**.

```
val zero = new HashMap[String, Int]()

def merge(dfs: HashMap[String, Int],
          tfs: (String, HashMap[String, Int])
    : HashMap[String, Int] = {
  tfs._2.keySet.foreach { term =>
    dfs += term -> (dfs.getOrElse(term, 0) + 1) }
  dfs
}
```

Computing DF Weights (II)

The document frequency of a term across all documents is a global property of the collection. Since we are operating over the partitioned data structure `docTermFreqs`, we need to aggregate it in two steps:

2. We combine the **document frequencies globally across all partitions**.

```
def comb(dfs1: HashMap[String, Int], dfs2: HashMap[String, Int])
    : HashMap[String, Int] = {
  for ((term, count) <- dfs2) {
    dfs1 += term -> (dfs1.getOrElse(term, 0) + count) }
  dfs1
}
```

Computing DF Weights (III)

The document frequency of a term across all documents is a global property of the collection. Since we are operating over the partitioned data structure `docTermFreqs`, we need to aggregate it in two steps:

```
docTermFreqs.aggregate(zero)(merge, comb)
```

And finally invert the DF values into their **IDF counterparts**:

```
val idfs = topDocFreqs.map {
  case (term, count) =>
    (term, math.log(numDocs.toDouble / count))
}.toMap
```

And **broadcast** this map:

```
bIdfs = sc.broadcast(idfs)
```

Broadcast the Term Dictionary

- Do not forget to **broadcast the static term-id dictionary**:

```
val termIds = idfs.keys.zipWithIndex.toMap
val bTermIds = sc.broadcast(termIds)
```

Finally Merge the TF and IDF Weights into Sparse Vectors

- The last transformation **combines the TF and IDF weights** for all terms in a document into a **sparse vector** representation:

```
import scala.collection.JavaConversions._
import org.apache.spark.mllib.linalg.Vectors

val vecs = docTermFreqs.map(termFreqs => {
  val docTotalTerms = termFreqs.values().sum
  val termScores = termFreqs.filter {
    case (term, freq) => bTermIds.value.containsKey(term)
  }.map{
    case (term, freq) => (bTermIds.value(term),
      bIdfs.value(term) * termFreqs(term) / docTotalTerms)
  }.toSeq
  Vectors.sparse(bTermIds.value.size, termScores)
})
```

Compute the Singular Value Decomposition

- The `RowMatrix` class is used to represent entries of a **document-term matrix** in a sparse way. It is directly used as input for the SVD computation:

```
import org.apache.spark.mllib.linalg.distributed.RowMatrix

vecs.persist()
val mat = new RowMatrix(vecs)
val k = 1000 // number of latent concepts
                    in the reduced matrix
val svd = mat.computeSVD(k, computeU=true)
```

- Caution: Spark MLlib *unfortunately* swaps the common convention used to encode the term-document matrix $\boldsymbol{A}$:
  - `svd.V` encodes the word-to-concept mappings
  - `svd.U` encodes the document-to-concept mappings
  - `svd.S` contains the singular values of $\boldsymbol{A}$ (in descending order)

Finding the Top Terms for Latent Concepts

We can now use V to inspect the **latent concepts** represented in our Wikipedia collection.

- First, we may want to **rank terms** by their similarity to the top concepts:

```
def topTermsInTopConcepts(
    svd: SingularValueDecomposition[RowMatrix, Matrix],
    numConcepts: Int, numTerms: Int): Seq[Seq[(String, Double)]] = {
  val v = svd.V
  val topTerms = new ArrayBuffer[Seq[(String, Double)]]()
  for (i <- 0 until numConcepts) {
    val offs = i * v.numRows
    val termWeights = v.toArray.slice(offs, offs + v.numRows).zipWithIndex
    val sorted = termWeights.sortBy(-_._1)
    topTerms += sorted.take(numTerms).map {
      case (score, id) =>
        (bTermIds.value.find(_._2 == id).getOrElse(("", -1))._1, score) } }
  topTerms }
```

Finding the Top Documents for the Latent Concepts

We can also use $\boldsymbol{U}$ to inspect the **latent concepts** represented in our Wikipedia collection.

- Next, we may want to **rank documents** by these top concepts:

```
def topDocsInTopConcepts(
    svd: SingularValueDecomposition[RowMatrix, Matrix],
    numConcepts: Int, numDocs: Int): Seq[Seq[(Long, Double)]] = {
    val u = svd.U
    val topDocs = new ArrayBuffer[Seq[(Long, Double)]]()
    for (i <- 0 until numConcepts) {
      val docWeights = u.rows.map(_.toArray(i)).zipWithUniqueId()
      topDocs += docWeights.top(numDocs).map {
        case (score, id) => (id, score) } }
    topDocs }
```

Print the Results

- Now we may finally **print the results of all our efforts** so far by using the two afore defined functions:

```
val topConceptTerms = topTermsInTopConcepts(svd, 4, 10)
val topConceptDocs = topDocsInTopConcepts(svd, 4, 10)
for ((terms, docs) <- topConceptTerms.zip(topConceptDocs)) {
  println("Concept terms: " + terms.map(_._1).mkString(", "))
  println("Concept docs: " + docs.map(_._1).mkString(", "))
  println()
}
```

- Note that MLlib stores $\boldsymbol{V}$ **locally at the driver node**, while $\boldsymbol{U}$ is stored in a **distributed manner across all executor nodes** (being sharded on its rows).

Querying the Latent Semantic Index

With the reduced decomposition of $\boldsymbol{A}$ into $\boldsymbol{U} \times \boldsymbol{S} \times \boldsymbol{V}^T$ at hand, we can now perform the following tasks in the "latent" concept space instead of using the actual terms and documents:

- **Document-to-document similarities**:
  - Compute row similarities of $\boldsymbol{U} \times \boldsymbol{S}$
- **Term-to-term similarities**:
  - Compute row similarities of $\boldsymbol{V} \times \boldsymbol{S}$
- **Single-term-to-document similarities:**
  - Multiply $\boldsymbol{U} \times \boldsymbol{S}$ with the row vector of term i in $\boldsymbol{V}$
- **Multi-term-to-document similarities:**
  - Transform $\boldsymbol{q}' = \boldsymbol{q}_{1\times m} \times \boldsymbol{V}_{m\times k}$
  - Multiply $\boldsymbol{U} \times \boldsymbol{S}$ with the transformed query vector $\boldsymbol{q}'^T$

Document-Document Relevance

- Based on the reduced representation, we can find the **most similar documents** to a given query document:

```
import org.apache.spark.mllib.linalg.Matrices

def topDocsForDoc(normalizedUS: RowMatrix, docId: Long)
    : Seq[(Double, Long)] = {
  val docRowArr = row(normalizedUS, docId)
  val docRowVec = Matrices.dense(docRowArr.length, 1, docRowArr)
  val docScores = normalizedUS.multiply(docRowVec)
  val allDocWeights = docScores.rows.map(_.toArray(0)).
    zipWithUniqueId()
  allDocWeights.filter(!_._1.isNaN).top(10)
}
val US = multiplyByDiagonalMatrix(svd.U, svd.s)
val normalizedUS = rowsNormalized(US)
topDocsForDoc(normalizedUS, idDocs(doc), docIds)
```

Term-Term Relevance

- Similarly, we can also find the **top similar terms** to a given query term:

```
import breeze.linalg.{SparseVector => BSparseVector}
import breeze.linalg.{DenseVector => BDenseVector}
import breeze.linalg.{DenseMatrix => BDenseMatrix}

def topTermsForTerm(
    normalizedVS: BDenseMatrix[Double],
    termId: Int): Seq[(Double, Int)] = {
  val rowVec = new BDenseVector[Double](
    row(normalizedVS, termId).toArray)
  val termScores = (normalizedVS * rowVec).toArray.zipWithIndex
  termScores.sortBy(-_._1).take(10)
}

val VS = multiplyByDiagonalMatrix(svd.V, svd.s)
val normalizedVS = rowsNormalized(VS)
topTermsForTerm(normalizedVS, idTerms(term), termIds)
```

Term-Document Relevance

- As in an actual **Information Retrieval** setting, we can find the **most similar documents** for a given query term:

```
def topDocsForTerm(US: RowMatrix, V: Matrix, termId: Int)
    : Seq[(Double, Long)] = {
  val termRowArr = row(V, termId).toArray
  val termRowVec = Matrices.dense(termRowArr.length, 1, termRowArr)
  val docScores = US.multiply(termRowVec)
  val allDocWeights = docScores.rows.map(_.toArray(0)).
    zipWithUniqueId()
  allDocWeights.top(10)
}

topDocsForTerm(normalizedUS, svd.V, idTerms(term))
```

Multi-Term Queries (I)

- First, we transform a query vector into a compatible `SparseVector` representation using IDF values as term weights:

```
def termsToQueryVector(
    terms: Seq[String],
    idTerms: Map[String, Int],
    idfs: Map[String, Double]): BSparseVector[Double] = {
  val indices = terms.map(idTerms(_)).toArray
  val values = terms.map(idfs(_)).toArray
  new BSparseVector[Double](indices, values, idTerms.size)
}
```

Multi-Term Queries (II)

- Finally, we multiply $\boldsymbol{U} \times \boldsymbol{S}$ with the transformed query vector $\boldsymbol{q'}^T$:

```
def topDocsForTermQuery(
    US: RowMatrix,
    V: Matrix,
    query: BSparseVector[Double]): Seq[(Double, Long)] = {
  val breezeV = new BDenseMatrix[Double](V.numRows, V.numCols, V.toArray)
  val termRowArr = (breezeV.t * query).toArray
  val termRowVec = Matrices.dense(termRowArr.length, 1, termRowArr)
  val docScores = US.multiply(termRowVec)
  val allDocWeights = docScores.rows.map(_.toArray(0)).
    zipWithUniqueId()
  allDocWeights.top(10) }

val queryVec = termsToQueryVector(terms, idTerms, idfs)
topDocsForTermQuery(US, svd.V, queryVec)
```

Summary

- **LSA** and related techniques, such as PCA, pLSA, LDA, are very broadly applied techniques for **analyzing text data**.
  - Dimensionality reduction
  - Content filtering
  - Clustering
  - Visualization
  - Indexing & retrieval

- **LSA** also has a variety of applications also **outside text analysis**:
  - Face detection ("eigenfaces") to detect common patterns in human appearance
  - Detection of climate patterns, etc.

Big Data Analytics

Chapter 5: *k*-Means Clustering & Anomaly Detection

Following: [3] "**Advanced Analytics with Spark**", Chapter 5

Prof. Dr. Martin Theobald

Faculty of Science, Technology & Communication

University of Luxembourg

Another Unsupervised Learning Technique: Clustering

Detecting anomalies within an unknown data set is tricky.

Clustering is an unsupervised learning technique that groups together objects that – in some sense – have similar properties.

Object groups whose properties are **distinguished from a majority** of the overall objects may thus indicate an "**anomaly**".

Unsupervised techniques can usually only help to detect anomalies; a **manual inspection** of the clusters is needed to decide if/what kind of an anomaly may have been detected this way.

***k*-Means** is by far the most common type of clustering:

1. Consider a **metric distance measure** such as **Euclidian distance** or **squared Euclidian distance**.
2. Find k **centroids** located at the "dense" regions of the data points.
3. Assign each object to its closest centroid such that the overall **sum of square distances** (SSD) between the objects and their centroids **is minimized**.

The KDD Cup 1999 Data Set

The **KDD Cup** is a (nearly) annual data-mining challenge that has been organized in conjunction with the **ACM KDD** (for "Knowledge Discovery from Data") **Conference** since 1997.

In 1999, the challenge was to **automatically detect anomalous patterns of network traffic** that would indicate a potential network intrusion.

Similarly to our earlier examples about matrix factorization and LSA, this is an **unsupervised setting**, where no labeled training data that would explicitly indicate such an intrusion (or even explicitly give us a hint on what an actual intrusion could be) is available.

Download the data set from here:
http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

A description of the original KDD task (incl. a description of the features) is available here: http://kdd.ics.uci.edu/databases/kddcup99/task.html

Outline of Chapter 5

5.1 *k*-Means Clustering

- Naïve Clustering Approach
- Iterative *k*-Means Algorithm
- Discussion & Limitations

5.2 *k*-Means Clustering & Anomaly Detection in Spark

- Loading and Exploring the Data Set
- Visualization in R
- Finding the Best Value for k
- Final Anomaly Detection

5.1 *k*-Means Clustering

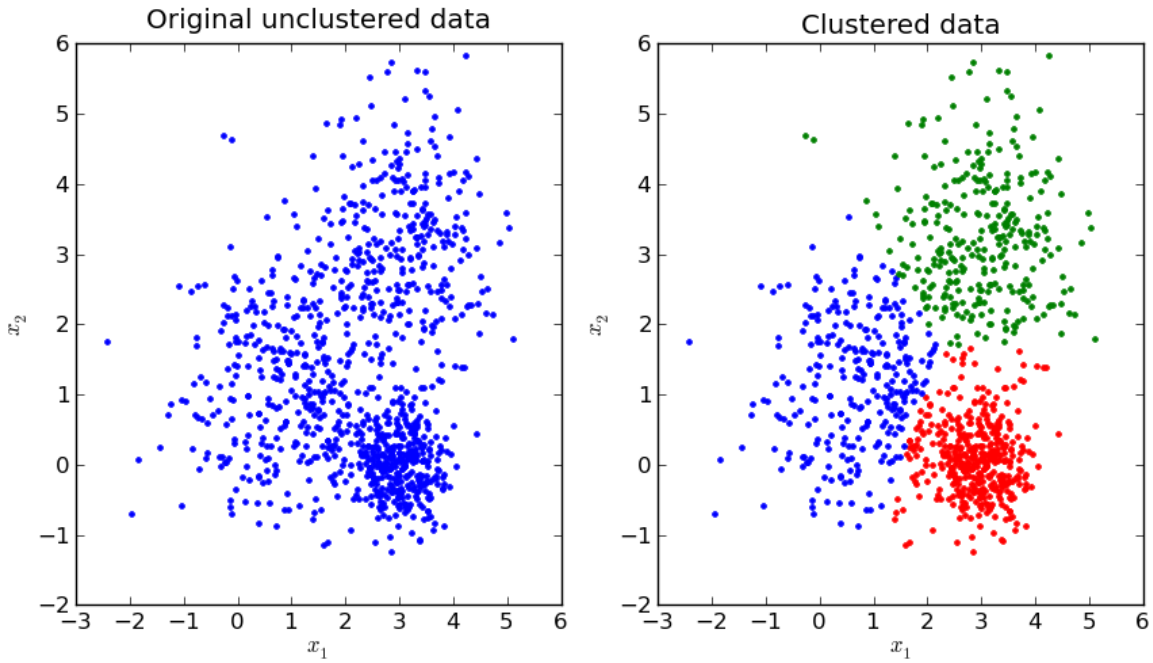
Clustering Objective

General Clustering Objective:

For a given set of k clusters $\boldsymbol{C} = \{C_1, \ldots, C_k\}$, we aim to assign each data object $o_j \in \boldsymbol{O}$ to exactly one of the C_l's, such that:

- the **intra-cluster distance** among objects within each $C_l \in \boldsymbol{C}$ is small,
- and the **inter-cluster distance** among objects between $C_l, C_p \in \boldsymbol{C}$, for $l \neq p$, is large.

Example:



Source: http://pypr.sourceforge.net/kmeans.html

A Naïve Clustering Approach (Exact Solution)

1. Generate **all possible clusterings** $\boldsymbol{C}$ of size k (i.e., assignments of n objects to k clusters) one-by-one.
   1. Compute the **centroids** μ_l of each obtained cluster $C_l \in \boldsymbol{C}$:
      $\mu_l = \frac{1}{|C_l|} \sum_{o_j \in C_l} o_j$
   2. Compute the **sum of squared distances** (SSD) among the objects in each cluster and their centroids, summed over all clusters:
      $\sum_{C_l \in \boldsymbol{C}} \sum_{o_j \in C_l} (o_j - \mu_l)^2$
2. Select the clustering C_l that yields the **overall minimum** sum of squared distances.

Unfortunately, this naïve approach is already *infeasible* in practice:

- There are k^n possible assignments of objects to clusters (incl. empty clusters).
- Even when excluding empty clusters, the number of possible assignments still is growing by the **Stirling number of the second kind**:

$$S(n,k) = \left\{ {n \atop k} \right\} = \frac{1}{k!} \sum_{j=0..k} (-1)^{k-j} \binom{k}{j} j^n$$

Basic *k*-Means Clustering (Approximate Solution)

The ***k*-Means algorithm** is by far the most commonly applied clustering technique:

1. Initially "guess" k points in the feature space; these will serve as the **initial centroids** of our clusters.
2. Assign each object to its closest centroid.
3. Repeat:
   1. Update the centroids as the "mean" of the objects assigned to each cluster:
      $\mu_l = \frac{1}{|C_l|} \sum_{o_j \in C_l} o_j$
   2. Re-assign each object to its **closest centroid** using the square distance among the object and each centroid.
   3. Compute the **sum of square distances** (SSD) between each data point and its nearest centroid:
      $\sum_{C_l \in \boldsymbol{C}} \sum_{o_j \in C_l} (o_j - \mu_l)^2$
4. Until a **local minimum** in the SSD measure is reached.

Fortunately, this approach is indeed *feasible*:

- It usually terminates after (at most) a few tens to hundreds of iterations.
- The runtime is $O(k\ n)$ per iteration, with a constant (i.e., fixed) number of iterations.

k-Means Discussion

- Finding the **optimal assignment** of n objects to k clusters that truly minimizes the sum of square distances (see the "naïve" approach) is known to be **NP-complete**.
- *k*-Means is a fairly simple heuristic that works sufficiently well in practice. It however is an iterative approach that may return a **local optimum** only.

Possible improvement:

- Randomly restart the *k*-Means initialization r times and choose the overall assignment with the lowest sum of square distances.

Note:

- For an unspecified number of clusters k, the sum of squared distances is trivially minimized by setting $k = n$, i.e., by assigning each object to a separate cluster. Various heuristics exist to find a good value for k.
- The second clustering objective we identified earlier (high inter-cluster distance) is not even considered by *k*-Means.

k-Means in Spark's MLlib

- Spark's MLlib implements a variant of the original *k*-Means algorithm with both an improved initialization phase and a parallel implementation, coined "***k*-Means++**" and "***k*-Means||**", the latter of which is described here in detail.
- The key idea is to initialize the algorithm by choosing only the first of the centroids uniformly at random among all data objects.
- The remaining $k-1$ **centroids** are then **iteratively initialized**, such that all data points have a **uniform sum of square distances** (SSD) to their closest centroids.
- After this initialization, all **centroids** are **iteratively updated** by the default *k*-Means algorithm until a local minimum in the SSD measure is reached.
- Both the initialization and the iterative clustering can be implemented in a **MapReduce-like setting**: Map assigns the n data objects to their closest centroids; Reduce then re-computes the k centroids for the next iteration

5.2 *k*-Means Clustering & Anomaly Detection in Spark

Load the Data Set

- The network traffic data from the 1999 KDD cup is again stored in a **line-based CSV format** that is available as a single file.
- After downloading and extracting the files, we may just load its contents into a first RDD by using the common `SparkContext.textFile` function:

```
val rawData = sc.textFile("./kddcup.corrected.csv")
```

Explore the Data Set (I)

- A glance at some line reveals the **internal structure** of the CSV file:

```
rawData.first

rawData.take(10)

rawData.takeSample(false, 10)
```

- The last column is indeed a **label** that indicates the type of network attack (as it manually detected) that was provided as part of the KDD cup. We can do a simple frequency analysis of these labels in the following line of Scala instructions:

```
rawData.map(_.split(',').last).countByValue().toSeq.
  sortBy(_._2).reverse.foreach(println)
```

Explore the Data Set (II)

For our first approach to cluster the data, we will just ignore the three non-numerical attributes in the first columns.

```
0,tcp,http,SF,215,45076,0,0,0,0,...,0.00,0.00,normal
```

At this point, we can identify **two basic issues** related to this format:

1. Note that many columns encode **binary** (yes/no) **features**. These however do **not** correspond to the **one-hot encoding** we have seen earlier. Here, multiple of these binary features may be active for a given line of the file. Hence, we cannot just combine these binary numerical features into a single feature as before for the CovType dataset.

2. Also note that the features are **not normalized**. Hence a binary feature that is active (with value of 1) will have a square distance to a non-active feature (with value of 0) of 1. Other features (such as *#bytes*) may easily create much larger square distances that will then dominate the other features in the Euclidian distance measure.

Extract the Labels and Feature Vectors

- We once more extract labels and dense feature vectors from the file to create the input to our clustering algorithm.
- The `vector` component of the resulting RDD can be conveniently accessed via `values` and is cached.

```
import org.apache.spark.mllib.linalg._

val labelsAndData = rawData.map { line =>
  val buffer = line.split(',').toBuffer
  buffer.remove(1, 3)
  val label = buffer.remove(buffer.length-1)
  val vector = Vectors.dense(buffer.map(_.toDouble).toArray)
  (label, vector)
}

val data = labelsAndData.values.cache()
```

First Take on *k*-Means Clustering

- Using *k*-Means in Spark's MLlib simply works by importing the `KMeans` class from the `clustering` package and calling the `run` function.

```
import org.apache.spark.mllib.clustering._

val kmeans = new KMeans()
kmeans.setK(2)
kmeans.setEpsilon(1.0e-4)
val model = kmeans.run(data)

model.clusterCenters.foreach(println)
```

- Here, `k` is the number of clusters, and `epsilon` is the convergence condition below which we stop the iterations.
- The given value of k is 2, so two vectors will be printed which represent the cluster centers (i.e., the centroids determined by the *k*-Means algorithm).

Analyzing the Clusters

- We can now **compare the assigned clusters** with the **23 labels of network anomalies** that were initially provided in the data set.
- We can do so by predicting the cluster that each input vector is assigned to (using the closest centroid) and then by counting how many labels of the input vectors were assigned to each of the two cluster ids:

```
val clusterLabelCount = labelsAndData.map {
  case (label, vector) =>
    val cluster = model.predict(vector)
 (cluster, label)
}.countByValue

clusterLabelCount.toSeq.sorted.foreach {
  case ((cluster, label), count) =>
    println(f"$cluster%1s$label%18s$count%8s") }
```

Finding a Good Value for *k*

- Two clusters seem to be clearly insufficient for distinguishing network anomalies from normal activity.
- To find the best value for the number of clusters k, we might try to adjust k such that the **average distance of all data points to their closest centroid is minimized**.
- We can try to do so in the following two steps…

(1) Define the Euclidian Distance Function in Scala

- The following Scala code explicitly implements the **Euclidian distance** measure:

```
def distance(a: Vector, b: Vector) =
  math.sqrt(a.toArray.zip(b.toArray).
    map(p => p._1 - p._2).map(d => d * d).sum)

def distToCentroid(vector: Vector, model: KMeansModel) = {
  val cluster = model.predict(vector)
  val centroid = model.clusterCenters(cluster)
  distance(centroid, vector)
}
```

(2) Define a "Scoring" Function for a Clustering Model

- We compute the `clusteringScore` for a given value of k and calculate the respective average distance of the vectors to their closest centroids:

```
import org.apache.spark.rdd._

def clusteringScore(data: RDD[Vector], k: Int) = {
  val kmeans = new KMeans()
  kmeans.setK(k)
  val model = kmeans.run(data)
  data.map(vector => distToCentroid(vector, model)).mean() }
```

- And we additionally can run this function for a number of values of k *in parallel* by invoking multiple tasks in the Spark shell to perform this computation:

```
(10 to 60 by 10).par.map(k => (k, clusteringScore(data, k))).
  foreach(println)
```

Caution!

There is a **major flaw** in our methodology at this point:

- The more clusters (i.e., centroids) we have, the lower the distance of a data point to its nearest centroid will be.
- In the extreme case, we might create **as many centroids as data points** and obtain an average distance of 0 between all data points and the centroids!
- Even worse, since *k*-Means is only approximate, we might end up choosing a value of k that just happens to represent a **local minimum** among the resulting distances in these clustering iterations.
- We thus need to find a better stopping condition for finding k.
- We will be coming back to this issue later…

Visualization in R (I)

- We can visualize what happened with our clustering approach so far by dumping a sample of your predictions to a text file.
- We resort to fixing k to 100 for this visualization.

```
val kmeans = new KMeans()
kmeans.setK(100)
val model = kmeans.run(data)

val sample = data.map(vector =>
  model.predict(vector) + "," + vector.toArray.mkString(",")
).sample(false, 0.01)

sample.saveAsTextFile("./kmeans-sample")
```

Visualization in R (II)

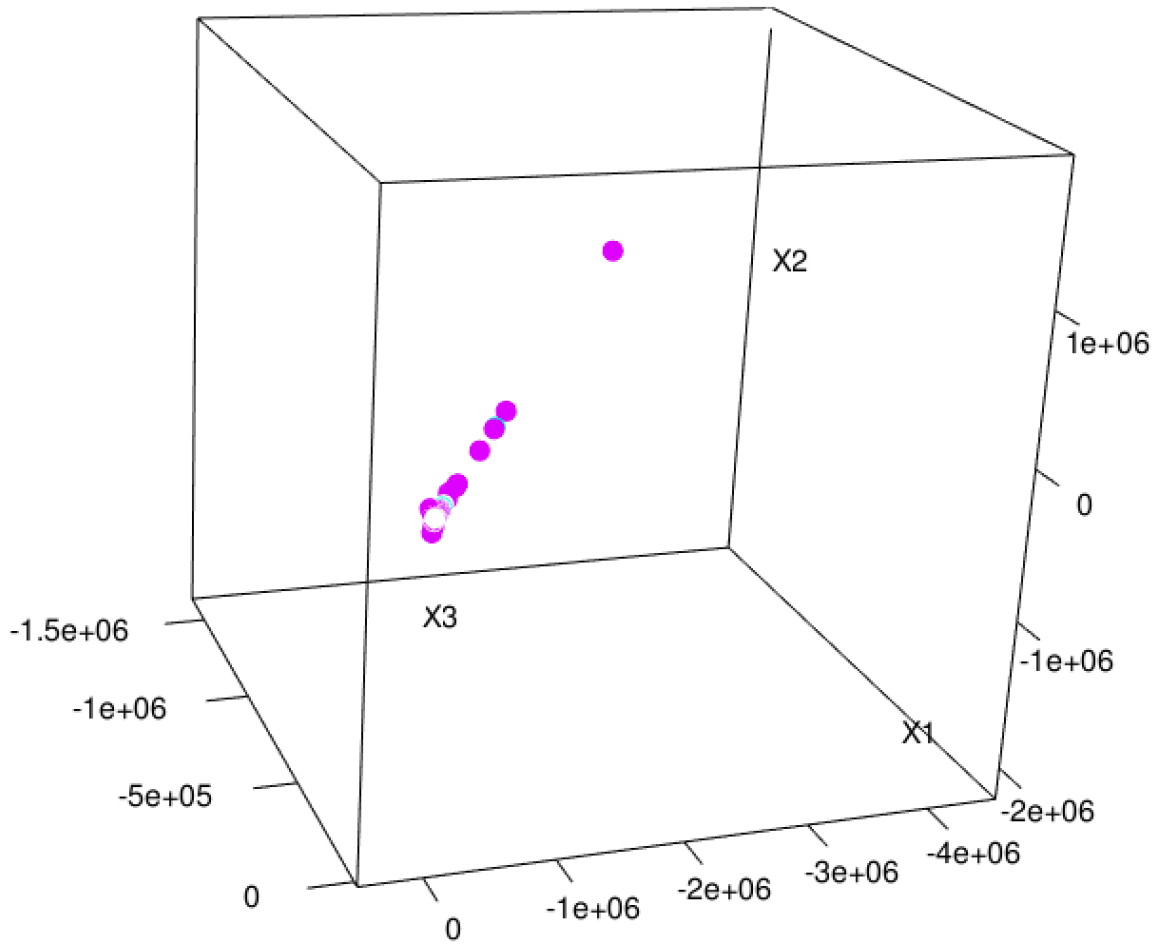
- By switching to an R shell, we can create a **3d-plot** (using a random projection of the 38 feature dimensions) of our 1%-sample of the data:

```
install.packages("rgl") # Use for first time only
library(rgl)
clusters_data <- read.csv(pipe("cat ./kmeans-sample/*"))
clusters <- clusters_data[1]
data <- data.matrix(clusters_data[-c(1)])
rm(clusters_data)
random_projection <- matrix(data = rnorm(3*ncol(data)), ncol = 3)
random_projection_norm <- random_projection / sqrt(rowSums(random_projection*random_projection))
projected_data <- data.frame(data %*% random_projection_norm)
num_clusters <- nrow(unique(clusters))
palette <- rainbow(num_clusters)
colors = sapply(clusters, function(c) palette[c])
plot3d(projected_data, col = colors, size = 10)
```

Visualization in R (III)

- Unfortunately, the clusters for the original data points seem to be rather indistinguishable:

$k = 100$



Normalizing the Attributes

- One of the issues we identified already was that attribute values are not normalized; hence *#bytes* likely dominates all binary features.
- A common statistical approach is to normalize each attribute i by **subtracting the mean of the features' values** from each individual feature value j under i, and by **dividing this value by the standard deviation** of the feature, as shown in the following standard equation:

$$normlized_{i,j} = \frac{value_{i,j} - \mu_i}{\sigma_i}$$

- Notice that the subtraction of the mean will not have any actual effect on the Euclidian distance measure; it is just used here for completeness.

→ The respective Scala code is available in `RunKMeans-shell.scala` on Moodle.

Analyzing the Normalized Data

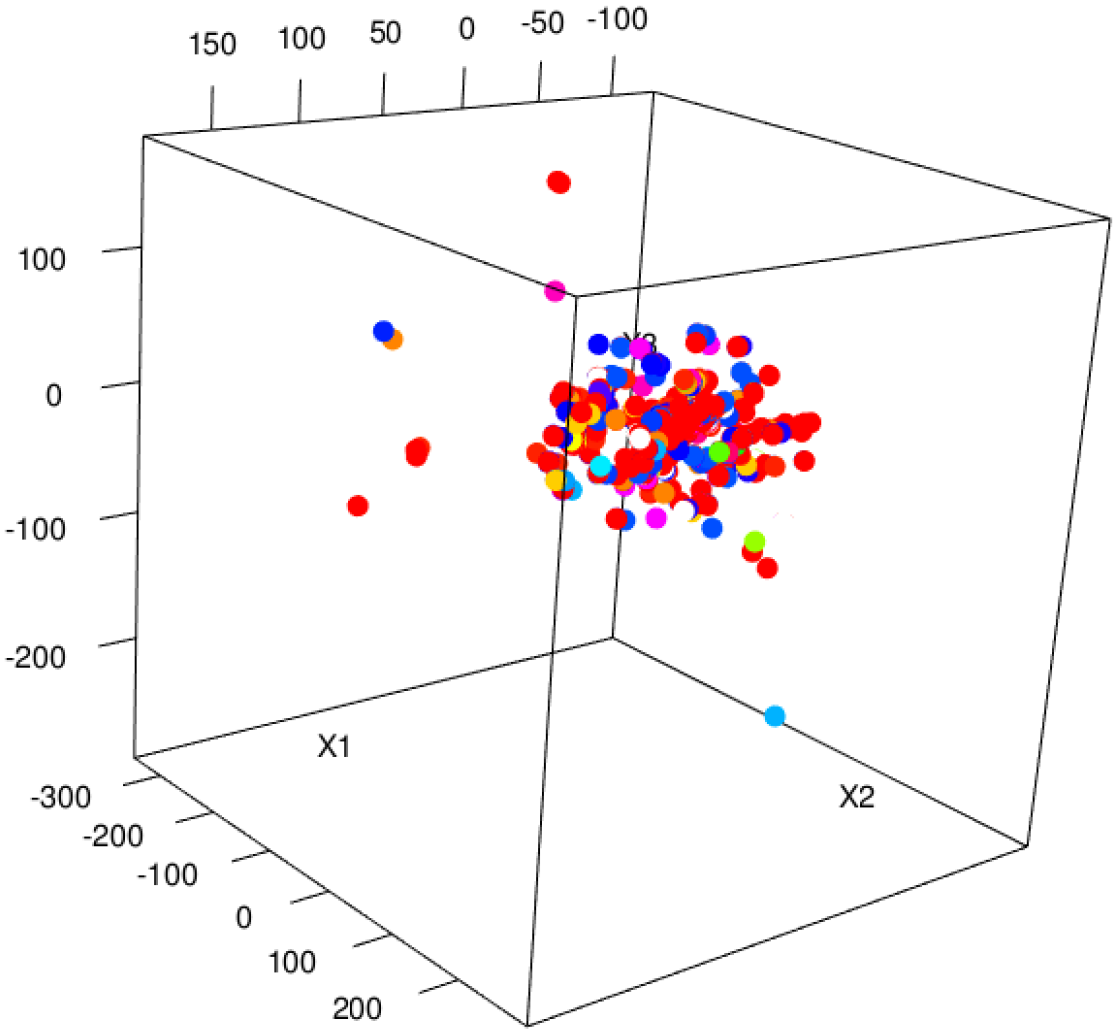
- We can now run the same test with normalized data, also on a higher range of k:

```
val normalizedData = data.map(normalize).cache()
(60 to 120 by 10).par.map(k =>
  (k, clusteringScore(normalizedData, k))).
     toList.foreach(println)
```

Visualizing the Normalized Data (I)

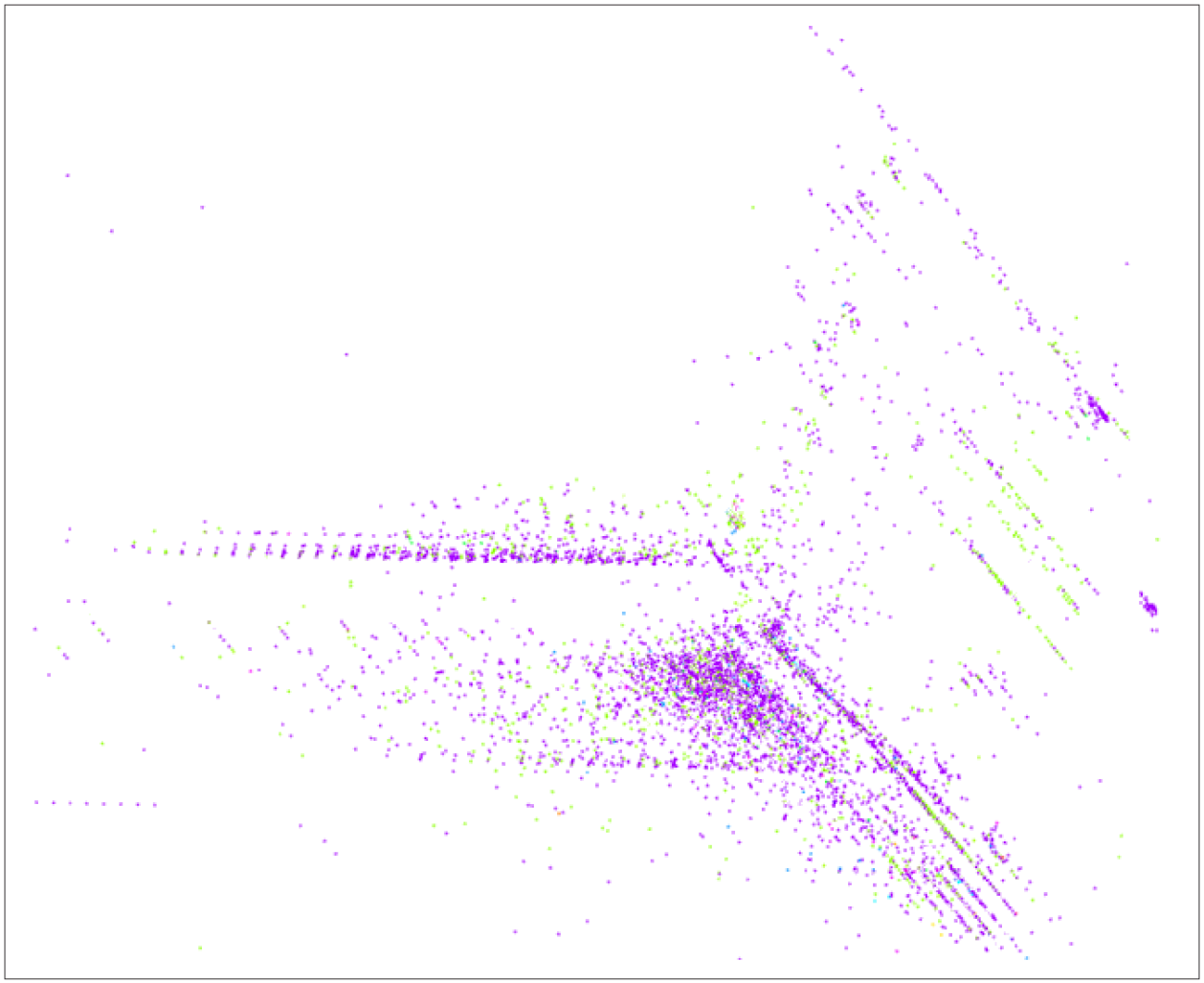
- The normalized data reveals a much richer structure than before:

$k = 100$



Visualizing the Normalized Data (II)

$k = 100$



Zoomed-in 3d projection of the normalized data

Translating Categorical Attributes into Numerical Ones

- The second issue we identified was that **categorical features** cannot directly be included into the Euclidian distance measure.
- However, the categorical features can be translated into several binary indicator features using a **one-hot encoding**, which can then be viewed as numeric dimensions. For example, the second column contains the protocol type with the following possible values: `tcp`, `udp`, or `icmp`.
- This feature could be thought of as three individual features, perhaps `is_tcp`, `is_udp`, and `is_icmp`. The single feature value `tcp` might thus become `1,0,0`; `udp` might be `0,1,0`; and so on.
- The resulting one-hot encoding (especially when normalized) provides meaningful distance values when fed into the Euclidian distance function.

→ Also here, the respective Scala code is available in `RunKMeans-shell.scala` on Moodle.

Measuring the Entropy Among the Labels in Each Cluster (I)

- Finally, we may consider the **entropy** (i.e., the "impurity") among the given labels in the clusters as a measure for how accurately the clustering algorithm was able to detect commonalities among the vectors that share the same label.
- As before, entropy is defined as: $\sum_{Labels\ x \in C_l} f(x) \log \frac{1}{f(x)}$
- This can be implemented in Scala as follows (using the non-zero counts of labels per cluster as input):

```
def entropy(counts: Iterable[Int]) = {
  val values = counts.filter(_ > 0)
  val n: Double = values.sum
  values.map { v =>
    val p = v / n
    - p * math.log(p)}.sum }
}
```

Measuring the Entropy Among the Labels in Each Cluster (II)

- The entropy measure is included into our earlier clustering score function, which then replaces the simple distance measure from before:

```
def clusteringScoreByEntropy(
    normalizedLabelsAndData: RDD[(String,Vector)],
    k: Int) = {
 ...
 val model = kmeans.run(normalizedLabelsAndData.values)
 val labelsAndClusters =
   normalizedLabelsAndData.mapValues(model.predict)
 val clustersAndLabels = labelsAndClusters.map(_.swap)
 val labelsInCluster = clustersAndLabels.groupByKey().values
 val labelCounts = labelsInCluster.map(
   _.groupBy(l => l).map(_._2.size))
 val n = normalizedLabelsAndData.count()
 labelCounts.map(m => m.sum * entropy(m)).sum / n }
```

Measuring the Entropy Among the Labels in Each Cluster (III)

- Run the entire pipeline by parsing both the labels and the vectors from the raw data RDD and by properly encoding and normalizing all attributes:

```
val parseFunction = onehot(rawData)
val labelsAndData = rawData.map(parseFunction)
val normalizedLabelsAndData =
 labelsAndData.mapValues(normalize(labelsAndData.values)).cache()
(120 to 160 by 10).par.map(k =>
   (k, clusteringScoreByEntropy(normalizedLabelsAndData,
     k))).toList.foreach(println)
```

- Notice that this methodology strictly speaking is a "cheat" because we strongly exploit the fact that intrusion labels are provided as part of the data from the KDD cup to compute the entropy measure.
- In reality, one might want to manually inspect the "purity" of the clusters with respect to possible anomalies to choose a value of k that maximizes this purity within each cluster.

Final Anomaly Detection

- An **anomaly** is a new data point whose distance to its closest centroid still is suspiciously large.
- We first build an `anomaly` detection function by creating a *k*-Means model from the normalized and one-hot encoded data. This model is built using the best value of k at 150 that we found by our previous steps.
- This anomaly function is used to detect objects whose distance to the centroid is **larger than the distance of the 100th object** in each cluster.

```
val parseFunction = onehot(rawData)
val originalAndData = rawData.map(line => (line,
  parseFunction(line)._2))
val data = originalAndData.values
val normalizeFunction = normalize(data)
val anomalyFunction = anomaly(data, normalizeFunction)
val anomalies = originalAndData.filter {
  case (original, vector) => anomalyFunction(vector) }.keys
anomalies.take(10).foreach(println)
```

Summary

After an initial *k*-Means model (represented as k centroids) is computed, new data objects can also be analyzed in a **streaming manner**, which is useful, e.g., for the online monitoring of network attacks based on the previously computed clustering model, etc.

Spark's MLlib therefore includes a libraries called `StreamingKMeans` and `StreamingKMeansModel` which can incrementally update and analyze a *k*-Means model from streaming data.

Other **distance measures** and/or **transformations** (via so-called "kernel" functions) into a higher-dimensional space may reveal **more subtle similarities** among data objects.

More principled **projections** into a lower-dimensional space (e.g., using SVD) may reveal **correlations among attributes** and are better suitable for visualization of the clustered data.

Finally, many more clustering techniques other than *k*-Means are available in the literature and still wait for their adaptation to a distributed stetting…

Big Data Analytics

Chapter 6: Medical Network Analysis in GraphX

Following: [3] "**Advanced Analytics with Spark**", Chapter 7

Prof. Dr. Martin Theobald

Faculty of Science, Technology & Communication

University of Luxembourg

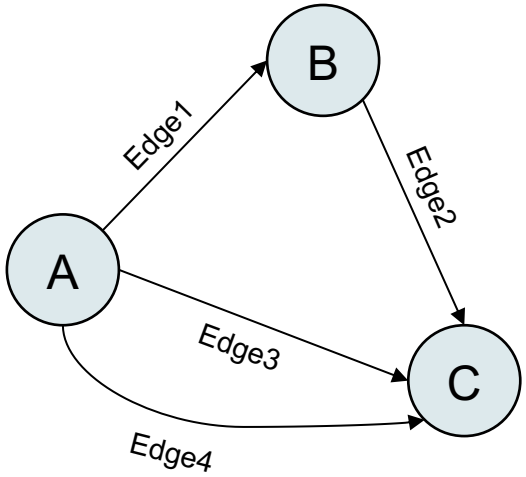
Spark's GraphX

- Just like MLlib, **GraphX** is part of the core Spark distribution.
- It provides a number of **parallel-processing algorithms** for **distributed directed labeled multi-graphs**, which are again stored as RDDs.
- **Core components** of the GraphX API:

```
val vertices: VertexRDD[<vertex-type>]
val edges: EdgeRDD[<edge-type>]

class Graph[<vertex-type>, <edge-type>] {
  val vertices: VertexRDD[<vertex-type>]
  val edges: EdgeRDD[<edge-type>] }

class EdgeTriplet[<vertex-type>, <edge-type>] {
  val srcAttr: <vertex-type>
  val dstAttr: <vertex-type>
  val attr: <edge-type> }
```



As before, instances of `VertexRDD` and `EdgeRDD` are automatically partitioned among all nodes in the Spark cluster.

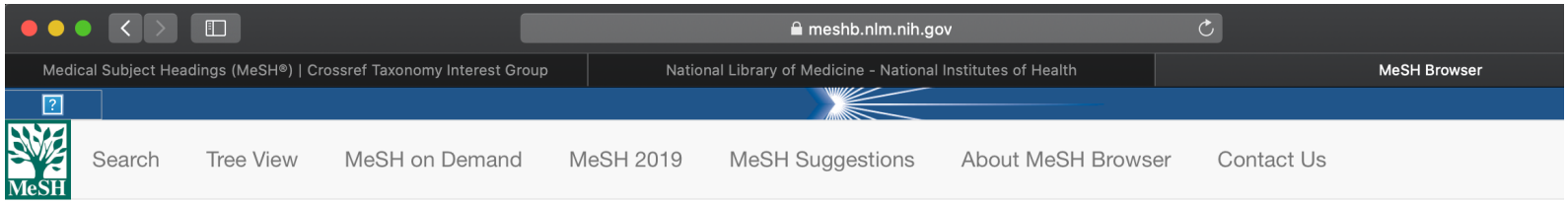
The MedLine Citation Index

- The Medline (Medical Literature Analysis and Retrieval System) is a huge database of medical research papers and clinical trials that have been published in the life sciences and in medicine.
- The main database contains more than 20 million medical publications.
- The provided dump from nlm.nih.gov which is available via FTP

  wget ftp://ftp.nlm.nih.gov/nlmdata/sample/medline/*.gz

- contains metadata and abstracts of 240,000 Medline articles in XML format.
- Each paper entry is wrapped into a <MedlineCitation> element.
- The <MeshHeading> elements contain keywords capturing the major topics by which the papers were annotated according to the MeSH taxonomy and can viewed via the MeSH browser.

MeSH Taxonomy Structure



Anatomy [A]
Organisms [B]
Diseases [C]
Chemicals and Drugs [D]
Analytical, Diagnostic and Therapeutic Techniques, and Equipment [E]
Psychiatry and Psychology [F]
Phenomena and Processes [G]
Disciplines and Occupations [H]
Anthropology, Education, Sociology, and Social Phenomena [I]
Technology, Industry, and Agriculture [J]
Humanities [K]
Information Science [L]
Named Groups [M]
Health Care [N]
Publication Characteristics [V]
Geographicals [Z]

Our Medline subset:

- 8 XML files
- 240,000 article abstracts
- 280,464 topic occurrences
- 14,548 unique topics
- 213,745 topic pairs

Load the XML Files into an RDD (I)

- The `cloudera.datascience` package contains (amongst others) rich APIs for parsing XML data.

```
import com.cloudera.datascience.common.XmlInputFormat

import scala.xml._
import org.apache.spark.SparkContext
import org.apache.spark.rdd._
import org.apache.hadoop.io.{Text, LongWritable}
import org.apache.hadoop.conf.Configuration
```

Load the XML Files into an RDD (II)

- The `newAPIHadoopFile` method of the `SparkContext` class supports custom delimiters for extracting parts of the contents of a text file into an RDD. We employ this to extract the Medline abstracts – one at a time – into an RDD of type `Array[String]`.

```
def loadMedline(sc: SparkContext, path: String) = {
  @transient val conf = new Configuration()
  conf.set(XmlInputFormat.START_TAG_KEY, "<MedlineCitation ")
  conf.set(XmlInputFormat.END_TAG_KEY, "</MedlineCitation>")
  val in = sc.newAPIHadoopFile(path, classOf[XmlInputFormat],
    classOf[LongWritable], classOf[Text], conf)
  in.map(line => line._2.toString) }

val medline_raw = loadMedline(sc, "./path-to-medline-files")
```

Basic XML Support in Scala

- As of version 1.2, Scala considers **XML as a first-class data type** (just like `Int`, `String`, etc.). Thus, the following code is syntactically valid:

```
import scala.xml._
val data = "My Medline Citation Node."
val citation = <MedlineCitation>data</MedlineCitation>
```

- After loading the Medline data into an XML element of type `scala.xmlElem`:

```
val raw_xml = medline_raw.take(1)(0)
val elem = XML.loadString(raw_xml)
```

the following **path operations** can be performed directly on that element:

```
elem.label
elem.attributes
elem \ "MeshHeadingList"
(elem \\ "DescriptorName").map(_.text)
```

Parse the MeSH Codes from the XML Format

- A "**major topic**" is denoted by a special markup tag in the Medline XML format. We extract all of these fields for each citation record in our RDD and cache the result:

```
def majorTopics(elem: Elem): Seq[String] = {
  val dn = elem \\ "DescriptorName"
  val mt = dn.filter(n => (n \ "@MajorTopicYN").text == "Y")
  mt.map(n => n.text) }
majorTopics(elem)
```

- And let's apply this function to the entire set of XML files via an RDD transformation:

```
val mxml: RDD[Elem] = medline_raw.map(XML.loadString)
val mesh_topics: RDD[Seq[String]] = mxml.map(majorTopics).cache()
mesh_topics.take(1)(0)
```

Analyze the MeSH Major Topics and Their Co-Occurrences

- A quick count over the topics reveals their **absolute frequencies**:

```
mesh_topics.count()
val topics: RDD[String] = mesh_topics.flatMap(mesh => mesh)
val topicCounts = topics.countByValue()
topicCounts.size

val tcSeq = topicCounts.toSeq
tcSeq.sortBy(_._2).reverse.take(10).foreach(println)
```

- First analyze the topic distribution by a simple frequency count:

```
val valueDist = topicCounts.groupBy(_._2).mapValues(_.size)
valueDist.toSeq.sorted.take(20).foreach(println)
```

- Show also some of the "long tail" topics:

```
tcSeq.sortBy(_._2).take(100).foreach(println)
```

Analyze the Co-Occurrences of Topics

- Next, we consider **pairs of topics that co-occur in a same Medline abstract**. The function `combinations(2)` achieves this effect very conveniently for us:

```
val topicPairs = mesh_topics.flatMap(t =>
     t.sorted.combinations(2))
val cooccurs = topicPairs.map(p => (p, 1)).reduceByKey(_+_)
```

- The results are up to $n\ (n-1)/2$ topic pairs. This is going to be a frequently accessed RDD, so we should cache it:

```
cooccurs.cache()
cooccurs.count()
```

Notes:

- Although the GraphX APIs support *directed* graphs, all of the following analyses consider the underlying mesh-topic graph as *undirected*.
- For an undirected graph with n vertices, the maximum amount of unique edges (excluding self-loop edges) is given by the **binomial coefficient** $n\ (n-1)/2$.

Option 1: Choose a Good Hashing Function for Vertices

- The VertexRDD and EdgeRDD objects we defined in the GraphX API require vertices to be identified by a unique `Long` value rather than by a `String`.

- Thus, for building the co-occurrence graph among major topics in the Medline articles, we need to first **turn** the **topic names** into **unique topic ids**:

```
import com.google.common.hash.Hashing

def getTopicId(str: String) = {
  Hashing.md5().hashString(str).asLong() }
```

`Hashing.md5` is an implementation of the MD5 message digest algorithm specification.

Option 2: Use a Counter for Unique Vertex Ids

- If we do not trust our hashing function, we may also **explicitly map** the topic string **to a unique id by using a counter**.

```
val topicIds =
  topics.zipWithUniqueId().collectAsMap()
val bTopicIds = sc.broadcast(topicIds).value

def getTopicId(str: String) = {
  bTopicIds(str) }
```

- Note that the mapping from the string to a unique integer is backed by a `HashMap` in Scala/Java. Thus, if there is a collision within this `HashMap`, the map interface still guarantees that the correct value is returned for each key.

- Hence, a `HashMap` actually stores a list of key/value pairs for each hash-code rather than just the hash-code/value pairs.

GraphX: Create the Graph Vertices as an RDD

- The unique MeSH topics from the Medline articles will form the vertices of our **co-occurrence network** among the major topics we found within a same Medline abstract.

- We start by importing the GraphX libraries:

```
import org.apache.spark.graphx._
```

- However, before we continue, we may wish to first verify that the assigned topic ids are indeed unique:

```
val vertices = topics.map(topic => (getTopicId(topic), topic))
val uniqueHashes = vertices.map(_._1).countByValue()
val uniqueTopics = vertices.map(_._2).countByValue()
uniqueHashes.size == uniqueTopics.size
```

GraphX: Create the Edges as another RDD

- Thus, edges are built from the unique ids and are based on the pairwise co-occurrences of MeSH topics in the Medline articles:

```
val edges = cooccurs.map(p => {
  val (topics, cnt) = p
  val ids = topics.map(getTopicId).sorted
  Edge(ids(0), ids(1), cnt) })
```

- We first initialize the `Graph` object in GraphX as a container for our vertices and edges. Also this data structure should be cached:

```
val topicGraph = Graph(vertices, edges)
topicGraph.cache()
```

- And we quickly analyze the graph's main properties:

```
vertices.count()
topicGraph.vertices.count()
topicGraph.edges.count()
```

More Advanced Network Analysis Algorithms

1. **Connected Components**

   A (strongly) connected-components analysis of a (directed) graph computes all subgraphs in which every pair of nodes is connected to each other. For both an undirected and a directed graph, these can be computed in linear time in the number of vertices and edges in the graph by using, for example, a simple a breath-first search over all edges and by storing the set of visited vertices for each intermediate vertex.

2. **Degree Distribution**

   The degree distribution is a coarse measure for how densely connected the graph is. Its average may range between 0 (no edges) and $\#vertices$ (fully connected).

3. **Cliques/Triangles & Clustering Coefficients**

   The clustering coefficient of vertices in a graph is a more detailed measure for how densely connected the graph is (based on the number of triangles each node participates in). Its average may range between 0 (no edges) and 1 (fully connected).

4. **Diameter & Average Path Length**

   Diameter and average path length require an iterative processing starting from all vertices either via breath- or depth-first search. The maximum number of iterations from any source to any (formerly unseen) target corresponds to the diameter of the graph.

1. Connected Components Analysis (I)

- A **connected component** of a graph is a subgraph in which every pair of nodes is reachable to each other.
- Fortunately, connected components can be computed very conveniently via a built-in API function of GraphX:

```
val connectedComponentGraph: Graph[VertexId, Int] =
  topicGraph.connectedComponents()
```

- We can also conveniently count in how many connected components each vertex participates:

```
def sortedConnectedComponents(
    connectedComponents: Graph[VertexId, _])
      : Seq[(VertexId, Long)] = {
      val componentCounts =
        connectedComponents.vertices.map(_._2).countByValue
      componentCounts.toSeq.sortBy(_._2).reverse }
```

1. Connected Components Analysis (II)

- We can also count the size of each connected component in the graph:

```
val componentCounts = sortedConnectedComponents(
  connectedComponentGraph)
componentCounts.size
componentCounts.take(10).foreach(println)
```

- The final result is obtained by an (inner) join between the two vertex sets:

```
val nameCID = topicGraph.vertices.
  innerJoin(connectedComponentGraph.vertices) {
    (topicId, name, componentId) => (name, componentId) }
```

Exploring the Connected Components

- Let's have a look at the second-largest component in the graph:

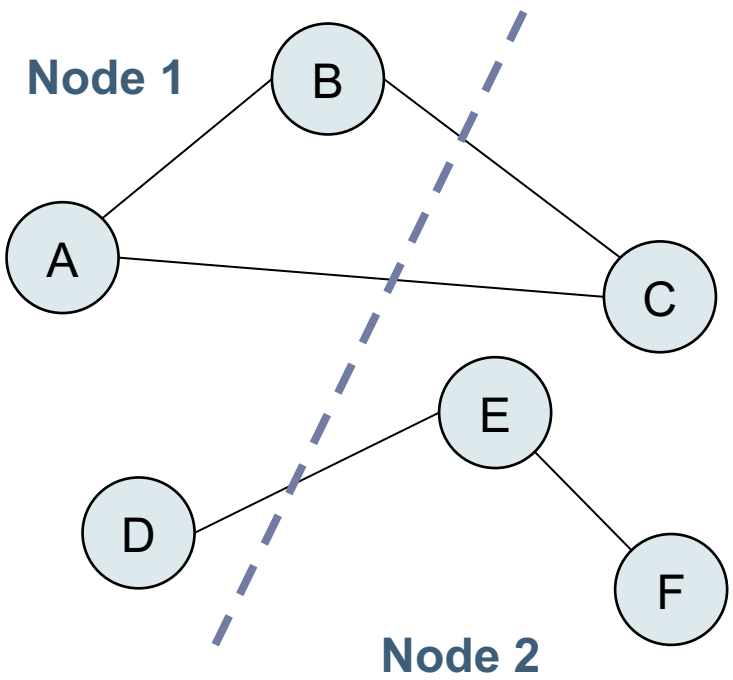
```
val c1 = nameCID.filter(x => x._2._2 == topComponentCounts(1)._2)
c1.collect().foreach(x => println(x._2._1))
```

- Compare these to the topic names containing the substring "HIV":

```
val hiv = topics.filter(_.contains("HIV")).countByValue()
hiv.foreach(println)
```

Iterative & Distributed Connected-Components Analysis

1. Initialize each vertex state with a unique number (e.g. a counter or id);
2. Do:
3. For all vertices in parallel do:
4. Send own state to all neighbors;
5. For all vertices in parallel do:
6. Receive all neighbors' states;
7. Set new state to be the *minimum* among own and all incoming states;
8. While "any node state changed";
9. Connected components are sets of vertices with same state;



Initialization

Vertex	State
A	1
B	2
C	3
D	4
E	5
F	6

Iteration 1

Vertex	State
A	1
B	1
C	1
D	4
E	4
F	5

Iteration 2

Vertex	State
A	1
B	1
C	1
D	4
E	4
F	4

Note:
In a distributed setting, this form of iterative computation requires state exchange (and hence communication) along the cut edges.

2. Analyzing the Degree Distribution (I)

- GraphX provides the `degrees` method which returns a `VertexRDD` of integers that captures the degree of each vertex:

```
val degrees: VertexRDD[Int] = topicGraph.degrees.cache()
degrees.map(_._2).stats()
```

- Compare the number of these RDDs with the number of the singleton topics, the ones that have no incoming or outgoing edges.

```
val singletonTopicGroups = mesh_topics.filter(x => x.size == 1)
singletonTopicGroups.count()

val singletonTopics = singletonTopicGroups.flatMap(mesh =>
  mesh).distinct()
singletonTopics.count()

val flatTopics = topicPairs.flatMap(p => p)
singletonTopics.subtract(flatTopics).count()
```

2. Analyzing the Degree Distribution (II)

- Let's take a closer look at the high-degree vertices by
  - joining the `degrees` VertexRDD to the vertices in the topic graph and
  - combining the topic name and the degree of the vertex into a tuple.

```
def topNamesAndDegrees(degrees: VertexRDD[Int],
    topicGraph: Graph[String, Int]): Array[(String, Int)] = {
  val namesAndDegrees = degrees.innerJoin(topicGraph.vertices) {
    (topicId, degree, name) => (name, degree) }

  val ord = Ordering.by[(String, Int), Int](_._2)
  namesAndDegrees.map(_._2).top(10)(ord) }
```

- And finally pretty-print the resulting tuples:

```
topNamesAndDegrees(degrees, topicGraph).foreach(println)
```

Digression: Filtering Out Noisy Edges

- In our current co-occurrence graph, the edges are weighted based on the **count** of **how often a pair of major topics** appears within a same abstract.
- The problem with this simple weighting scheme is that it does not **distinguish** topic pairs that occur together because they have a **meaningful semantic relationship** from topic pairs that occur together because they happen to **both occur frequently** for any type of abstract.
- We may thus apply the X^2 ("**Chi-Square**") **test of independence** to distinguish frequently co-occurring from just randomly (i.e., "**independently**") co-occurring topic pairs.
- For co-occurrences among two topics A and B into the four categories YY, YN, NY, NN, we can determine the coefficient of the X^2 test statistic as follows:

$$X^2 = N \frac{(|YY \times NN - YN \times NY| - N/2)^2}{(YA \times NA \times YB \times NB)}$$

where the values are obtained from a respective **contingency table** (see rhs.) among A and B.

	Yes B	No B	A Total
Yes A	YY	YN	YA
No A	NY	NN	NA
B Total	YB	NB	N

Digression: The X^2 Test Statistic

- The value of the test statistic is computed as follows:

```
def chiSq(YY: Int, YB: Int, YA: Int, N: Long): Double = {
  val NB = N - YB
  val NA = N - YA
  val YN = YA - YY
  val NY = YB - YY
  val NN = N - NY - YN - YY
  val inner = (YY * NN - YN * NY) - N / 2.0
  N * math.pow(inner, 2) / (YA * NA * YB * NB) }
```

- These new weights are assigned to all edges in the graph:

```
val N = mesh_topics.count()
val chiSquaredGraph = topicCountGraph.mapTriplets(triplet => {
  chiSq(triplet.attr, triplet.srcAttr, triplet.dstAttr, N) })

chiSquaredGraph.edges.map(x => x.attr).stats()
```

Digression: The X^2 Test

- For a contingency table with r rows and c columns, the basic parameter of the X^2 distribution, called the "**degrees of freedom**", is computed as $(r-1)(c-1)$. For our 2-by-2 contingency table, this is just 1.

- The ***p*-value** of the X^2 test is defined as one minus the area under the X^2 distribution for the computed value of the test statistic at 1 degree(s) of freedom (see next slide).

- Formally, the *p*-value is defined as the probably that the co-occurrence frequencies of topics A and B are "**at least as extreme**" as the ones we observe in our graph, given that we assume that A and B indeed occur independently of each other.

- If this *p*-value is very small, say less than 1%, we may reject the **null hypothesis** stating that A and B just occur independently and thus keep the respective edge in our graph because the pair may actually be a meaningful one.

Digression: The X^2 Distribution

1.1.2.10. Obere 100α-prozentige Werte χ^2_α der χ^2-Verteilung (s. 5.2.3.)

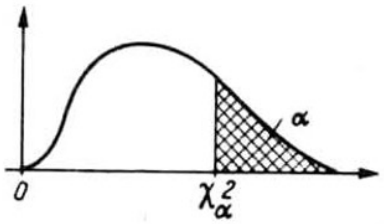


Abb. 1.4

Anzahl der Freiheitsgrade m	Wahrscheinlichkeit $p = \alpha$															
	0,99	0,98	0,95	0,90	0,80	0,70	0,50	0,30	0,20	0,10	0,05	0,02	0,01	0,005	0,002	0,001
1	0,000 16	0,0006	0,0039	0,016	0,064	0,148	0,455	1,07	1,64	2,7	3,8	5,4	6,6	7,9	9,5	10,83
2	0,020	0,040	0,103	0,211	0,446	0,713	1,386	2,41	3,22	4,6	6,0	7,8	9,2	10,6	12,4	13,8
3	0,115	0,185	0,352	0,584	1,005	1,424	2,366	3,67	4,64	6,3	7,8	9,8	11,3	12,8	14,8	16,3
4	0,30	0,43	0,71	1,06	1,65	2,19	3,36	4,9	6,0	7,8	9,5	11,7	13,3	14,9	16,9	18,5
5	0,55	0,75	1,14	1,61	2,34	3,00	4,35	6,1	7,3	9,2	11,1	13,4	15,1	16,8	18,9	20,5
6	0,87	1,13	1,63	2,20	3,07	3,83	5,35	7,2	8,6	10,6	12,6	15,0	16,8	18,5	20,7	22,5
7	1,24	1,56	2,17	2,83	3,82	4,67	6,35	8,4	9,8	12,0	14,1	16,6	18,5	20,3	22,6	24,3
8	1,65	2,03	2,73	3,49	4,59	5,53	7,34	9,5	11,0	13,4	15,5	18,2	20,1	22,0	24,3	26,1
9	2,09	2,53	3,32	4,17	5,38	6,39	8,34	10,7	12,2	14,7	16,9	19,7	21,7	23,6	26,1	27,9
10	2,56	3,06	3,94	4,86	6,18	7,27	9,34	11,8	13,4	16,0	18,3	21,2	23,2	25,2	27,7	29,6
11	3,1	3,6	4,6	5,6	7,0	8,1	10,3	12,9	14,6	17,3	19,7	22,6	24,7	26,8	29,4	31,3
12	3,6	4,2	5,2	6,3	7,8	9,0	11,3	14,0	15,8	18,5	21,0	24,1	26,2	28,3	30,9	32,9
13	4,1	4,8	5,9	7,0	8,6	9,9	12,3	15,1	17,0	19,8	22,4	25,5	27,7	29,8	32,5	34,5
14	4,7	5,4	6,6	7,8	9,5	10,8	13,3	16,2	18,2	21,1	23,7	26,9	29,1	31,3	34,0	36,1
15	5,2	6,0	7,3	8,5	10,3	11,7	14,3	17,3	19,3	22,3	25,0	28,3	30,6	32,8	35,6	37,7
16	5,8	6,6	8,0	9,3	11,2	12,6	15,3	18,4	20,5	23,5	26,3	29,6	32,0	34,3	37,1	39,3
17	6,4	7,3	8,7	10,1	12,0	13,5	16,3	19,5	21,6	24,8	27,6	31,0	33,4	35,7	38,6	40,8
18	7,0	7,9	9,4	10,9	12,9	14,4	17,3	20,6	22,8	26,0	28,9	32,3	34,8	37,2	40,1	42,3
19	7,6	8,6	10,1	11,7	13,7	15,4	18,3	21,7	23,9	27,2	30,1	33,7	36,2	38,6	41,6	43,8
20	8,3	9,2	10,9	12,4	14,6	16,3	19,3	22,8	25,0	28,4	31,4	35,0	37,6	40,0	43,0	45,3
21	8,9	9,9	11,6	13,2	15,4	17,2	20,3	23,9	26,2	29,6	32,7	36,3	38,9	41,4	44,5	46,8
22	9,5	10,6	12,3	14,0	16,3	18,1	21,3	24,9	27,3	30,8	33,9	37,7	40,3	42,8	45,9	48,3
23	10,2	11,3	13,1	14,8	17,2	19,0	22,3	26,0	28,4	32,0	35,2	39,0	41,6	44,2	47,3	49,7
24	10,9	12,0	13,8	15,7	18,1	19,9	23,3	27,1	29,6	33,2	36,4	40,3	43,0	45,6	48,7	51,2
25	11,5	12,7	14,6	16,5	18,9	20,9	24,3	28,2	30,7	34,4	37,7	41,6	44,3	46,9	50,1	52,6
26	12,2	13,4	15,4	17,3	19,8	21,8	25,3	29,2	31,8	35,6	38,9	42,9	45,6	48,3	51,6	54,1
27	12,9	14,1	16,2	18,1	20,7	22,7	26,3	30,3	32,9	36,7	40,1	44,1	47,0	49,6	52,9	55,5
28	13,6	14,8	16,9	18,9	21,6	23,6	27,3	31,4	34,0	37,9	41,3	45,4	48,3	51,0	54,4	56,9

Digression: Generate the Filtered Graph

- We filter out all edges whose ***p*-values are below 1%** (this confirms to a value of the test statistic of about 6.6 at 1 degree of freedom):

```
val interesting = chiSquaredGraph.subgraph(
  triplet => triplet.attr > 6.6)
interesting.edges.count
```

Digression: Further Analyze the Filtered Graph

1. **Connected components** of the filtered graph:

```
val interestingComponentCounts = sortedConnectedComponents(
  interesting.connectedComponents())
interestingComponentCounts.size
interestingComponentCounts.take(10).foreach(println)
```

2. **Degree distribution** of the filtered graph:

```
val interestingDegrees = interesting.degrees.cache()
interestingDegrees.map(_._2).stats()
topNamesAndDegrees(interestingDegrees,
  topicGraph).foreach(println)
```

3. Cliques & Clustering Coefficients (I)

- A **clique** is a fully connected subgraph in which every pair of nodes is directly connected via an edge with each other.

- A **triangle count** thus counts all cliques that consist of exactly three vertices:

```
val triangleCountGraph = topicGraph.triangleCount()
triangleCountGraph.vertices.map(x => x._2).stats()
```

3. Cliques & Clustering Coefficients (II)

- If have computed the triangle count of the graph, then the **local clustering coefficient** C for a vertex that has k neighbors (for $k > 1$) and takes part in t triangles is defined as follows:

$$C = \frac{2t}{k(k-1)}$$

- We can then compute the **average local clustering coefficient** over all vertices in the graph in Spark as follows:

```
val maxTriangleGraph = topicGraph.degrees.mapValues(
  d => d * (d - 1) / 2.0)
val clusterCoefficientGraph = triangleCountGraph.vertices.
  innerJoin(maxTriangleGraph) {
    (vertexId, triCount, maxTris) => {
      if (maxTris == 0) 0 else triCount / maxTris } }
clusterCoefficientGraph.map(_._2).sum() /
     topicGraph.vertices.count()
```

4. Average Path Length in GraphX (I)

Pregel was the first MapReduce-based graph engine that introduced the concept of **node-centric computing**. The average path length can be implemented as follows:

1. Each vertex v stores a local map of reachable vertices and their distances to v.
2. Messages sent from v to its successors contain the map of v (including v itself) with an added distance of 1.
3. Incoming messages at each vertex are merged into a new local map using the minimum distance of v to any node in the incoming maps.

```
def mergeMaps(m1: Map[VertexId, Int], m2: Map[VertexId, Int])
     : Map[VertexId, Int] = {
  def minThatExists(k: VertexId): Int = {
     math.min(
       m1.getOrElse(k, Int.MaxValue),
       m2.getOrElse(k, Int.MaxValue))
  }
  (m1.keySet ++ m2.keySet).map {
     k => (k, minThatExists(k)) }.toMap }
```

4. Average Path Length in GraphX (II)

- `Update` is part of the Pregel API (now simulated in GraphX) and in this case just calls the `mergeMaps` function to merge the local map with each incoming map:

```
def update(id: VertexId, state: Map[VertexId, Int],
    msg: Map[VertexId, Int]) = {
  mergeMaps(state, msg) }
```

4. Average Path Length in GraphX (III)

- The `checkIncrement` function increments the distances in the local map of each vertex v by 1, merges the local map with the one from an incoming message using `mergeMaps`, and sends the results of the `mergeMaps` function to the successors of v.

```
def checkIncrement(a: Map[VertexId, Int], b: Map[VertexId, Int],
    bid: VertexId) = {
  val aplus = a.map { case (v, d) => v -> (d + 1) }
  if (b != mergeMaps(aplus, b)) {
    Iterator((bid, aplus))
  } else {
    Iterator.empty
  }
}
```

4. Average Path Length in GraphX (IV)

- `iterate` is another Pregel function (now simulated in GraphX), that merges the iterators returned by `checkIncrement` in both forward and backward direction, hence we perform both a **forward and backward breadth-first search** (BFS) over the graph.

```
def iterate(e: EdgeTriplet[Map[VertexId, Int], _]) = {
  checkIncrement(e.srcAttr, e.dstAttr, e.dstId) ++
    checkIncrement(e.dstAttr, e.srcAttr, e.srcId)
}
```

4. Average Path Length in GraphX (V)

- With the afore defined functions at hand, we could now compute **the path length between all pairs of nodes** (if connected) in the entire graph, which will consume $O(\#vertices^2)$ memory in the Spark cluster.
- For a large graph, this may quickly grow very large, such that we resort to using a **vertex-induced subgraph** with consisting of 2% of randomly sampled vertices of our topic graph to approximate the average path length:

```
val sampleVertices = topicGraph.vertices.map(v =>
     v._1).sample(false, 0.02).collect().toSet

val mapGraph = interesting.mapVertices((id, _) => {
  if (sampleVertices.contains(id)) {
    Map(id -> 0)
  } else {
    Map[VertexId, Int]()
  } })
```

4. Average Path Length in GraphX (VI)

- Start the Pregel-style form of **iterative breadth-first search**:

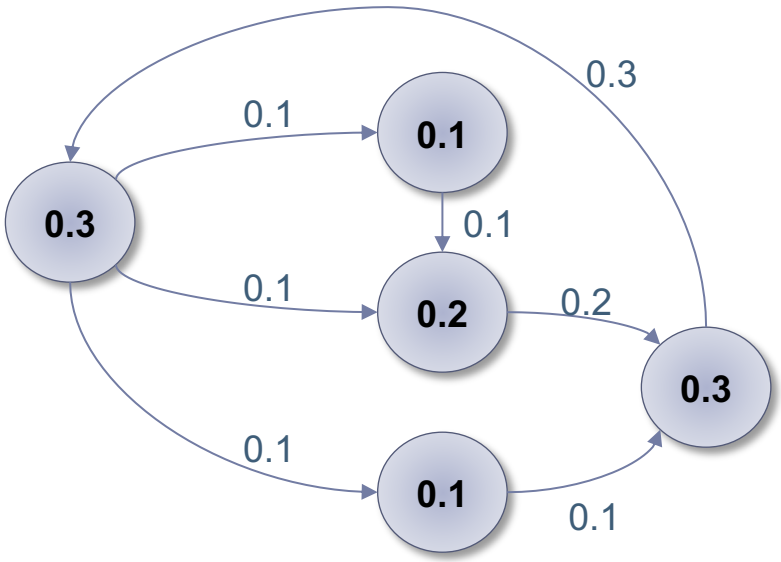
```
val start = Map[VertexId, Int]()
val res = mapGraph.pregel(start)(update, iterate, mergeMaps)
```

- And finally **extract the path lengths** (as distinct pairs of vertices with their distances) from all the local maps stored at the vertices:

```
val paths = res.vertices.flatMap { case (id, m) =>
  m.map { case (k, v) =>
    if (id < k) { (id, k, v) } else { (k, id, v) }
  }
}.distinct()
paths.cache()
paths.map(_._3).filter(_ > 0).stats()
val hist = paths.map(_._3).countByValue()
hist.toSeq.sorted.foreach(println)
```

Side Note: PageRank in GraphX (I)

- The **PageRank algorithm** (Brin, Page: 1998) was the initial break-through for Google's ranking algorithm.
- It considers the hyperlinks in a web crawl as edges of a directed graph and measures the importance of each vertex in this graph, assuming an **edge** from u to v **represents an endorsement** of v's importance by u.
- For example, if a web page v is linked to by many other important pages $u_1, \ldots, u_n$, then v itself becomes more important.



(not showing random jumps)

- Mathematically, this recursive dependency can be expressed as follows

$$PageRank(v) = \frac{\varepsilon}{N} + \sum_{u_i \in inlinks(v)} \frac{PageRank(u_i)}{|outlinks(u_i)|}$$

where ε is a "**random-jump probability**" (usually 0.05 – 0.15) and N is the number of all vertices in the graph.

Side Note: PageRank in GraphX (II)

- Fortunately, PageRank is readily implemented in GraphX:

```
val ranks = graph.pageRank(0.0001, 0.15).vertices
```

The first parameter is a threshold for the convergence (i.e., the sum of the absolute differences in PageRank values of all vertices between two iterations), the second one is the random-jump probability ε.

☞ See Moodle for a direct implementation of **PageRank in Spark using regular RDDs**!

- Once more, sort and pretty-print the results:

```
val namesAndRanks = ranks.innerJoin(topicGraph.vertices) {
  (topicId, rank, name) => (name, rank) }
val ord = Ordering.by[(String, Double), Double](_._2)
namesAndRanks.map(_._2).top(20)(ord).foreach(println)
```

Summary

- GraphX is a great extension of the Spark core libraries that is optimized for **processing distributed graph algorithms** in parallel.
- A graph is **automatically sharded** on both vertices and edges (both stored internally as RDDs).
- The term "**node-centric computing**" was first coined in the context of Pregel (and implemented in MapReduce):
  - Each vertex has a *local state* only.
  - Each vertex communicates only with its *immediate neighbors* (both successors and predecessors are allowed for communication).
  - Messages among neighboring vertices are *iteratively exchanged* and *merged* among the neighboring vertices.
- Graph-based data representations are **ubiquitous** in everyday life (social nets, citations & references, web graphs, etc.)
- See http://snap.stanford.edu for a wide collection of **real-world** (both small and large-scale) **graphs**!

Big Data Analytics

Chapter 7: Geospatial, Temporal & Streaming Data Analysis

Following: [3] "**Advanced Analytics with Spark**", Chapter 8

Prof. Dr. Martin Theobald

Faculty of Science, Technology & Communication

University of Luxembourg

What is Temporal & Spatial Data?

Spatial data is a form of multidimensional data in which some – typically a few – attributes denote annotations over a continuous (spatial) domain. Most common such annotations are:

- **Temporal**: 1d annotations (*time points* where a series of consecutive measurements from the same source are then called a "**session**" or a "**time-series**")
- **Geographic**: mostly 2d (*latitude*/*longitude* pairs as in the **GIS** and **GeoJSON** standards), perhaps 3d annotations (plus *elevation* as in the **GPS** standard)

The most common mathematical representation for multidimensional data points is the **Euclidian space** with **Euclidian distance** as distance metric:

$$\mathrm{d}(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Spheres & Polygons

However, Euclidian distance does not work well for the **actual surface distance** of two points **on a spherical shape** (e.g., the shortest path between north and south pole would go through the center of the earth).

- Thus, for the surface distance of two pairs of longitude/latitude points, the **Haversine formula** is more appropriate:

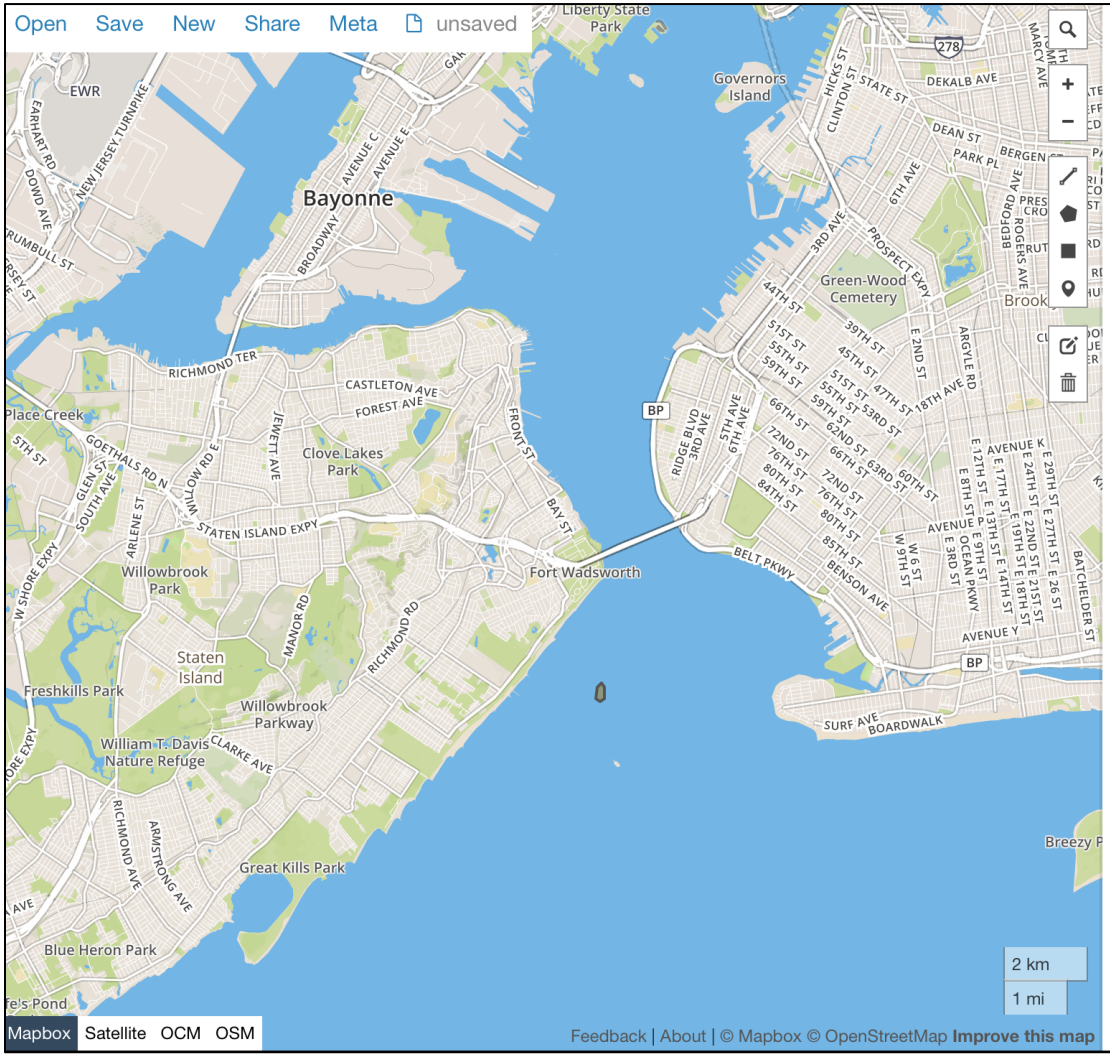
$$\mathrm{d}\left((lat_1, lon_1), (lat_2, lon_2)\right) = 2R \arcsin\left(\sqrt{\sin^2\left(\frac{lat_2 - lat_1}{2}\right) + \cos(lat_1)\cos(lat_2)\sin^2\left(\frac{lon_2 - lon_1}{2}\right)}\right)$$

($\rightarrow$ use $R = 6{,}371$ km for this planet!)

Finally, **polygons** can approximate arbitrary 2d and 3d surfaces (which may then additionally also be projected onto a sphere) by a discrete set of polygon boundaries.

- **Convex polygon**: all internal angles among adjacent edges are $\leq 180°$.
- **Concave polygon**: at least one internal angle among a pair of adjacent edges is > 180°.

GeoJSON Example & Online Demo



http://geojson.io/#map=12/40.6097/-74.0657

The NYC-Taxi-Trips & NYC-Boroughs Data Sets

The **NYC Taxi Trips** data set is a collection of taxi trips and fares in downtown New York City from January 2013 in CSV format:

http://www.andresmh.com/nyctaxitrips/

- The uncompressed file contains 2.5 GB of data for 14.8 million individual taxi rides. Each line contains a driver's license id, start- and end-time, a pick-up GPS location and a drop-off GPS location.

The second data set contains polygons of **NYC city areas** ("**boroughs**") such as Manhattan, Brooklyn, Queens, etc. in GeoJSON format:

https://github.com/haghard/streams-recipes/blob/master/master/nyc-borough-boundaries-polygon.geojson

A basic **geospatial & temporal analytical query pattern** (which we will try to solve in the following slides) may look as follows:

- **How long does it take on average for a taxi driver to find a new customer with respect to the borough and the hour-of-day?**

See here for a very cool visualization of the NYC taxi trips!

Date & Time APIs

- The Java "built-in" `DateTime` and `SimpleDateFormat` libraries are used for **parsing date/time strings**:

```
import java.text.SimpleDateFormat
val format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss")
val date = format.parse("2014-10-12 10:30:44")
val datetime = new DateTime(date)
```

- The `joda` and `nscala` libraries are used to capture **durations**:

```
import com.github.nscala_time.time.Imports._
val dt1 = new DateTime(2014, 9, 4, 9, 0)
val dt2 = new DateTime(2014, 10, 31, 15, 0)
val d = new Duration(dt1, dt2)
d.getMillis
d.getStandardHours
d.getStandardDays
```

Geometric APIs

- The `esri` library provides a special `Geometry` object as a basic data structure for **2d geospatial objects** that are encoded as polygons of longitude/latitude pairs.

```
import com.esri.core.geometry.Geometry
import com.esri.core.geometry.GeometryEngine
import com.esri.core.geometry.SpatialReference
```

Extended Geometry API (I)

- To spare syntax in our further programs, we define a new helper class, called `RichGeometry`, which serves as a wrapper for the `Geometry` class.

- The standard we will use for **spherical distance computations** among a pair of GPS coordinates is WKID 4326.

```
class RichGeometry(val geometry: Geometry,
    val spatialReference: SpatialReference =
  SpatialReference.create(4326)) {

    def area2D() = geometry.calculateArea2D()

    def contains(other: Geometry): Boolean = {
      GeometryEngine.contains(geometry, other, spatialReference) }

    def distance(other: Geometry): Double = {
      GeometryEngine.distance(geometry, other, spatialReference) }
}
```

Extended Geometry API (II)

- Next, we declare a companion object for `RichGeometry` that provides support for implicitly converting instances of the `Geometry` class into the `RichGeometry` class:

```
object RichGeometry {
  implicit def wrapRichGeo(g: Geometry) = {
    new RichGeometry(g) } }
```

- And make sure to import the implicit function definition into the Scala environment:

```
import RichGeometry._
```

GeoJSON API

- The `spray` package provides a commonly used API for the **GeoJSON standard**:

```
import spray.json.JsValue

case class Feature(
  val id: Option[JsValue],
  val properties: Map[String, JsValue],
  val geometry: RichGeometry) {
    def apply(property: String) = properties(property)
    def get(property: String) = properties.get(property) }
```

- Polygons are encoded in GeoJson as so-called **feature collections**:

```
case class FeatureCollection(features: Array[Feature])
    extends IndexedSeq[Feature] {
  def apply(index: Int) = features(index)
  def length = features.length }
```

Reading and Writing a Collection of GeoJSON Objects

```
implicit object FeatureJsonFormat extends
    RootJsonFormat[Feature] {
  def write(f: Feature) = {
    val buf = scala.collection.mutable.ArrayBuffer(
      "type" -> JsString("Feature"),
      "properties" -> JsObject(f.properties),
      "geometry" -> f.geometry.toJson)
    f.id.foreach(v => { buf += "id" -> v})
    JsObject(buf.toMap) } }

  def read(value: JsValue) = {
    val jso = value.asJsObject
    val id = jso.fields.get("id")
    val properties = jso.fields("properties").asJsObject.fields
    val geometry = jso.fields("geometry").convertTo[RichGeometry]
    Feature(id, properties, geometry) }
```

Geospatial Data Structure for the Taxi-Trip Data Set

- Based on the `esri` geometric and `nscala` date/time packages, we now define a **custom data structure** for our taxi trips:

```
import com.esri.core.geometry.Point
import com.github.nscala_time.time.Imports._

case class TaxiTrip(
  pickupTime: DateTime,
  dropoffTime: DateTime,
  pickupLoc: Point,
  dropoffLoc: Point)
```

- The parser for the specific date/time format used in the taxi data:

```
val formatter = new SimpleDateFormat(
  "yyyy-MM-dd HH:mm:ss")
```

- And finally a GPS coordinate is parsed from two input strings:

```
def point(longitude: String, latitude: String): Point = {
  new Point(longitude.toDouble, latitude.toDouble) }
```

Parse the Taxi-Trip Data Set

- The `parse` function extracts all the information we need capture taxi drivers and their trip data from each line of the CSV file:

```
def parse(line: String): (String, TaxiTrip) = {
  val fields = line.split(',')
  val license = fields(1)
  val pickupTime = new DateTime(formatter.parse(fields(5)))
  val dropoffTime = new DateTime(formatter.parse(fields(6)))
  val pickupLoc = point(fields(10), fields(11))
  val dropoffLoc = point(fields(12), fields(13))
  val trip = TaxTrip(pickupTime, dropoffTime, pickupLoc,
    dropoffLoc)
  (license, trip) }
```

Interactively Analyze Invalid Records

- `Either[left, right]` provides an interesting mechanism in Scala to **split** the output of a function **into two disjoint sets**. These are initialized from the `Left` and `Right` constructors.

- We can employ this as a wrapper for our actual parse function to debug interactively inspect and possibly debug mistakes in the CSV data.

```
def safe[S, T](f: S => T): S => Either[T, (S, Exception)] = {
  new Function[S, Either[T, (S, Exception)]] with Serializable {
    def apply(s: S): Either[T, (S, Exception)] = {
      try {
         Left(f(s))
      } catch {
         case e: Exception => Right((s, e))
      }
    } } }
```

Wrapper for Safe Parsing

- Now we may finally read the taxi data from the CSV file into an RDD by using the regular line-by-line input format:

```
val taxiRaw = sc.textFile("./path-to-taxi-trip-data")
val taxiHead = taxiRaw.take(10)
taxiHead.foreach(println)
```

- This first RDD is transformed into a second RDD using the safe parsing function:

```
val safeParse = safe(parse)
val taxiParsed = taxiRaw.map(safeParse)
taxiParsed.cache()
```

- And inspect the good vs. the bad (i.e., erroneous) tuples:

```
taxiParsed.map(_.isLeft).countByValue().foreach(println)
```

Handling Bad Records

- We need two steps to **filter the out the bad tuples** from the RDD with all parsed tuples:

```
val taxiBad = taxiParsed.filter(_.isRight).map(_.right.get)
```

```
val taxiBad = taxiParsed.collect({
  case t if t.isRight => t.right.get
})
```

- And inspect the bad tuples (these are actually not too many):

```
taxiBad.collect().foreach(println)
```

Handling Good Records

- The **good tuples are kept and also cached** for further processing:

```
val taxiGood = taxiParsed.collect({
  case t if t.isLeft => t.left.get
})
taxiGood.cache()
```

- The further inspect the good tuples for **unusual trip durations**:

```
import org.joda.time.Duration

def getHours(trip: TaxiTrip): Long = {
  val d = new Duration(
    trip.pickupTime,
    trip.dropoffTime)
    d.getStandardHours }

taxiGood.values.map(getHours).countByValue().
  toList.sorted.foreach(println)
```

Filtering Out Meaningless Records

- Suspicious taxi trips are those that have a **negative trip time** or that take **more than 3 hours** (at least for NYC circumstances). The remaining ones are kept in a third RDD:

```
val taxiClean = taxiGood.filter {
  case (lic, trip) => {
    val hrs = hours(trip)
    0 <= hrs && hrs < 3
  }
}
```

Geospatial Data Analysis: Load the NYC Boroughs Geometry

- GeoJSON objects are first loaded from a file …

```
val geojson = scala.io.Source.
  fromFile("./nyc-borough-boundaries-polygon.geojson").mkString
```

- ... and can then very conveniently be parsed directly via the `spray` API:

```
import com.cloudera.science.geojson._
import GeoJsonProtocol._
import spray.json._

val features = geojson.parseJson.convertTo[FeatureCollection]
```

- Let's test our new geometry function to look up the borough of a given query point:

```
val p = new Point(-73.994499, 40.75066)
val borough = features.find(f => f.geometry.contains(p))
```

Mapping from Borough Codes to Geometry Objects

- Next, we will store the polygons captured by the feature collections and map them to their borough names (such as Queens, Brooklyn, etc.).

```
val areaSortedFeatures = features.sortBy(f => {
  val borough = f("boroughCode").convertTo[Int]
  (borough, -f.geometry.area2D())
})
```

- The polgyons are broadcast and accessed via a new `borough` function:

```
val bFeatures = sc.broadcast(areaSortedFeatures)

def borough(trip: TaxiTrip): Option[String] = {
    val feature: Option[Feature] = bFeatures.value.find(f => {
    f.geometry.contains(trip.dropoffLoc) })
  feature.map(f => {
    f("borough").convertTo[String]
  })}
```

Analyze the Filtered Records

- We can now for the first time **combine the information from both data sets** by analyzing the frequencies of the individual boroughs as the drop-off locations for the tax trips:

```
taxiClean.values.map(borough).countByValue().foreach(println)
```

- Since there still seem to be many empty ones, we'll do more filtering and check for the most frequent drop-off boroughs:

```
taxiClean.values.filter(t => borough(t).isEmpty).
  take(10).foreach(println)
```

More Filtering

- If the GPS recording of a trip in the taxi data set did not work properly, the coordinates seem to have been set to a default value of (0.0/0.0).
- We can filter out these entries as follows:

```
def hasZero(trip: Trip): Boolean = {
  val zero = new Point(0.0, 0.0)
  (zero.equals(trip.pickupLoc) || zero.equals(trip.dropoffLoc))
}
```

- This will result in our final RDD which we will cache for further processing:

```
val taxiDone = taxiClean.filter {
  case (lic, trip) => !hasZero(trip)
}.cache()

taxiDone.values.map(borough).countByValue().foreach(println)
```

"Sessionization" of Taxi Trips

- A **session** of consecutive tax trips is a series of rides offered by the same driver.

- We can obtain all of these sessions in a **single pass** over all taxi trips by:
  1. using the drivers' license ids as **primary sorting condition**;
  2. using the trips starting times as **secondary sorting condition**;
  3. **splitting** sessions that take more than a certain time span.

Sessionization via Secondary Sorting (I)

- Recall that the first attribute in the `taxiDone` RDD is the driver's license. This attribute will also serve as the **primary sorting condition** for our taxi trips.
- A **secondary sorting condition** can be defined by the pickup times of each driver and taxi trip, which are selected by an additional function:

```
def secondaryKey(trip: TaxiTrip) = trip.pickupTime.getMillis
```

- An optional split function can be applied to split groups of drivers/consecutive trips with a duration of more than 4 hours (because then a driver is obliged to take a break):

```
def split(t1: TaxiTrip, t2: TaxiTrip): Boolean = {
  val p1 = t1.pickupTime
  val p2 = t2.pickupTime
  val d = new Duration(p1, p2)
  d.getStandardHours >= 4 }
```

Sessionization via Secondary Sorting (II)

- The final sessions are then obtained by a single `groupByKeyAndSortValues` call over the `taxiDone` RDD.
- The resulting RDD is also cached. Here, `30` denotes the desired number of partitions in this RDD.

```
val sessions = groupByKeyAndSortValues(
  taxiDone, secondaryKey, split, 30)

sessions.cache()
```

Analyzing Sessions by the City Boroughs

- The next function **analyzes all pairs of taxi trips**, in which the first trip ended in a particular borough and the second trip ended in any (other) borough.
- The function returns the drop-off borough of the first trip together with the duration between the drop-off time of the first trip and the pick-up time of the second trip.
- The function will be called for pairs of trips that are done by the same taxi driver/license id:

```
def boroughDuration(t1: TaxiTrip, t2: TaxiTrip) = {
  val b = borough(t1)
  val d = new Duration(t1.dropoffTime, t2.pickupTime)
  (b, d)
}
```

The "Sliding" Operator

- Using the `sliding` operator we may obtain an iterator over the sorted trips we defined to entail our sessions.
- The `filter` function ensures that we only analyze sessions that consist of exactly two trips.
- The result is of type `RDD[(Option[String], Duration)]` which we cache.

```
val boroughDurations: RDD[(Option[String], Duration)] =
  sessions.values.flatMap(trips => {
    val iter: Iterator[Seq[TaxiTrip]] = trips.sliding(2)
    val viter = iter.filter(_.size == 2)
    viter.map(p => boroughDuration(p(0), p(1)))
}).cache()
```

Analyze the Session Durations

- Another inspection of these durations (in hours) reveals that very few remaining **trip pairs seem to be invalid** (e.g., have a negative session duration). We may decide to either keep or ignore these in our final analysis.

```
boroughDurations.values.map(_.getStandardHours).
  countByValue().toList.sorted.foreach(println)
```

Finally Analyze the Idle-Time Durations by City Borough

- Recall our initial geospatial and temporal analytical query pattern we defined in the beginning. Here is the solution:

```
import org.apache.spark.util.StatCounter

boroughDurations.filter {
  case (b, d) => d.getMillis >= 0
}.mapValues(d => {
  val s = new StatCounter()
  s.merge(d.getStandardSeconds)
}).
  reduceByKey((a, b) => a.merge(b)).collect().foreach(println)
```

Summary

- The `nscala` and `Joda` packages provide **extended APIs** for managing temporal data (various date/time formats, durations, etc.).
- The `spray` and `esri` packages provide **very rich APIs** for geospatial data (GeoJson, Points, FeatureCollections, etc.)
- Noise and meaningless annotations in the source data usually require a good understanding of the application setting by a human analyst to first **clean the raw data sets**.
- Both geospatial and temporal annotations may **reveal interesting patterns and dependencies** among individual data objects that would otherwise remain concealed behind a simple CSV or TSV format.
- If **linked with other data sources** (such as twitter streams, RSS feeds, or RDF data), this allows us to perform some very exciting data-analytics tasks…


Data
Analytics
Big
Hadoop
Spark
Prediction
Distribution
Matrix
Scalability
Predictive
Sparse
Scala
Trees
GraphX
Rank
GFS
Purity
Systems
JSON
XML
MLlib
Sampling
Variable
Random
GeoJSON
Recommender
Analytics
Science
Ham
Confusion
Clustering
Reduce
Parallelism
Language
RDD
Streaming
Multivariate
AudioScrobbler
Twitter
MeSH
Recall
Broadcast
Precision
HDFS
Visualization
Decision
Validation
Triangles
Statistical
Components
Value
Normal
Classification
Outlier
Combiner
Action
Gradient
ALS
CoverType
Partitioner
GPS
Cosine
Java
Eigenvalue
Cross
Vector
Entropy
Semantic
HITS
Efficiency
Map
Linear
Networks
Testsal
WordCount
Decomposition
Trips
Connected
API
Natural
Forests
Transformation
Cliques
Latent
Processing
Gini
Singular
Medline
Taxi
Medical
Weather
Detection
Strongly
Risk
Financial
Eigenvector
Factorization
PageRank
Anomaly
Probability
Accumulator
Confidence
Dense
DStream
K-Means
Spam
Regression
Graphs
DataFrame
Dataset
Dataflow
Boosted
SQL
Open
NoSQL
GIS

Big Data Analytics

Chapter 8: Financial Risk Analysis via Monte Carlo Simulations

Following: [3] "**Advanced Analytics with Spark**", Chapter 9

Prof. Dr. Martin Theobald

Faculty of Science, Technology & Communication

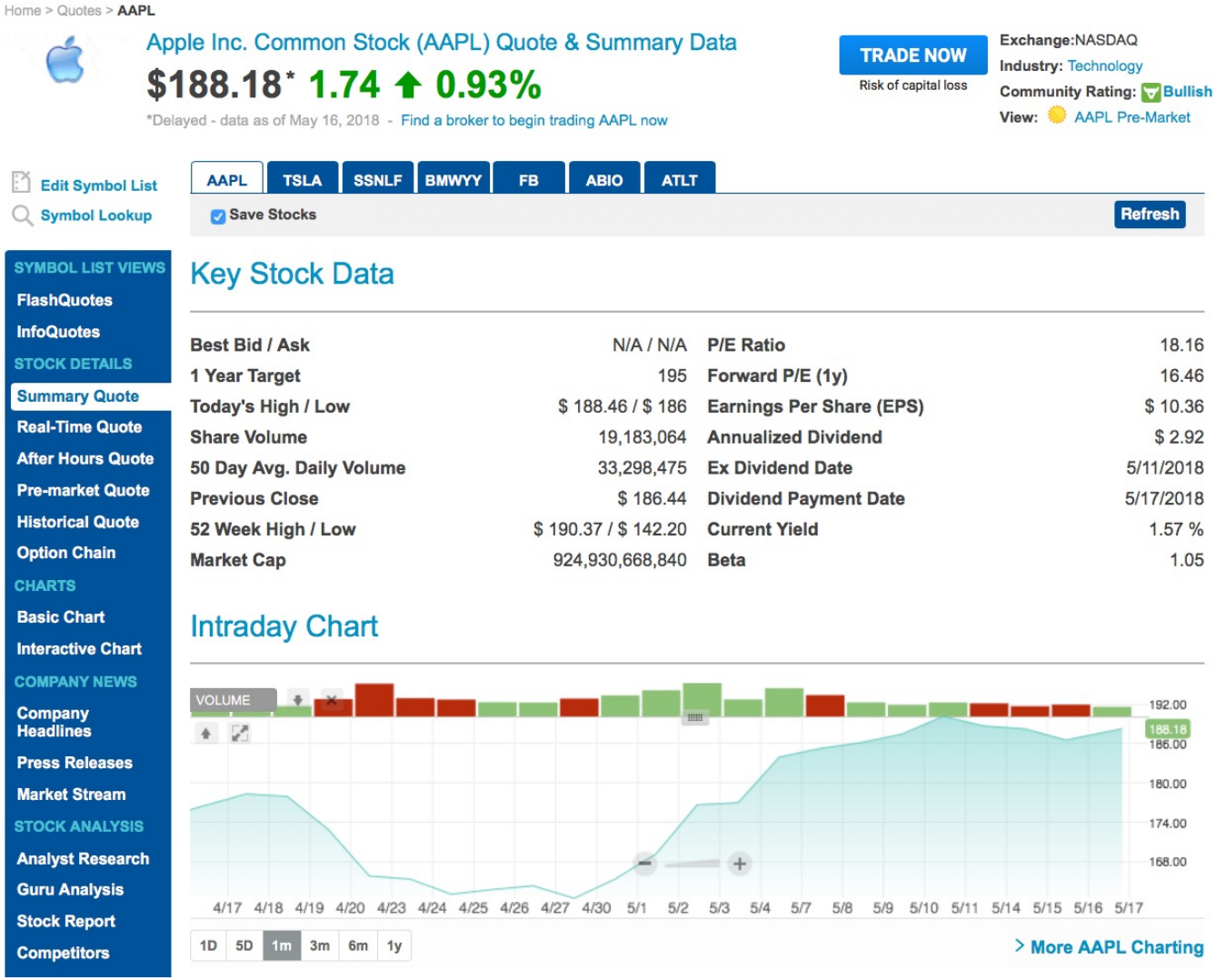
University of Luxembourg

7 Steps to Estimate the "Value-at-Risk" from Market Data

The **Value-at-Risk** is defined as the *5% quantile of worst returns* of a given portfolio (of stocks or other kinds of shares) over a *fixed period of time*.

1. Fix a **time period** (e.g. "2 weeks") over which to measure stock returns and market factors
2. Define **stock returns** r_i of given portfolio over a sliding window of market data periods
3. Similarly define **factor returns** f_j of given market factors over a sliding window of market data periods
4. Train a **linear-regression model** to predict returns r_i with respect to market factors f_j from past market data
5. Refine the linear-regression model by computing **pairwise correlations** between market factors f_j from past market data via a multivariate normal distribution
6. **Sample** returns r_i under all combinations of correlated market factors f_j
7. **VaR** then is the 5% quantile of the resulting distribution of returns r_i

Example: NASDAQ Apple (AAPL) Stocks



Source: https://www.nasdaq.com/symbol/aapl

Stock Markets & Returns of Investments

Consider a typical **stock-market setting**:

- Every **stock price** may in- or decrease over a **given period of time** (e.g., 1 week or 1 month) in reaction to a given set of **market conditions**.
- The **return** r_i of a stock i denotes the relative difference between its closing and opening price under the observed time period:

$$r_i = \frac{closingPrice_i - openingPrice_i}{closingPrice_i}$$

In addition to the stocks, we thus are also interested in a set of market conditions that analysts may have identified as **good indicators** or **influential factors** for the development of the stock prices. Also these will be measured in terms of returns.

Examples of market conditions/factors:

- S&P 500 drops by 5%
- US GDP increases by 0.2%
- Gold price increases by 1%

In the following, we consider only **large stock indexes** as such market conditions…

A Linear Regression Model for Stocks and Factors

Next, we may aim to estimate the **impact of each market condition** (then called "**factor**") on stock i from the developments of the stock under a set of previously observed market conditions:

$$r_i = c_i + \sum_{j=1}^{m} w_{i,j} \, f_j$$

The above formula denotes a simple form of **linear regression**:

- r_i are the **observed return values** (i.e., the "labels") obtained from the previous development of stock i.
- f_j are the so-called **features** (e.g., the observed factor return, squared factor return, etc.) derived from the previous development of the factors.
- c_i and $w_{i,j}$ are the **parameters** of the model which we wish to learn.

Note:

- To be able to compare the return of the stock r_i and the impact of each feature f_j on it, we need to ensure that the observations of both the stocks and the factors are perfectly temporally aligned.

How Can We Estimate "Financial Risk" ?

Large customers (such as banks) usually have an entire **portfolio** (consisting of multiple stocks and other investments) that they manage.

- If one stock goes up another one may go down, such that the portfolio will change its return value according to the sum or average of the returns' changes.

The **Value-at-Risk** (VaR) is a statistic that is often applied in Finance. It reflects *how much aggregated return* a given portfolio of stocks likely loses over a given period of time by considering all possible market conditions.

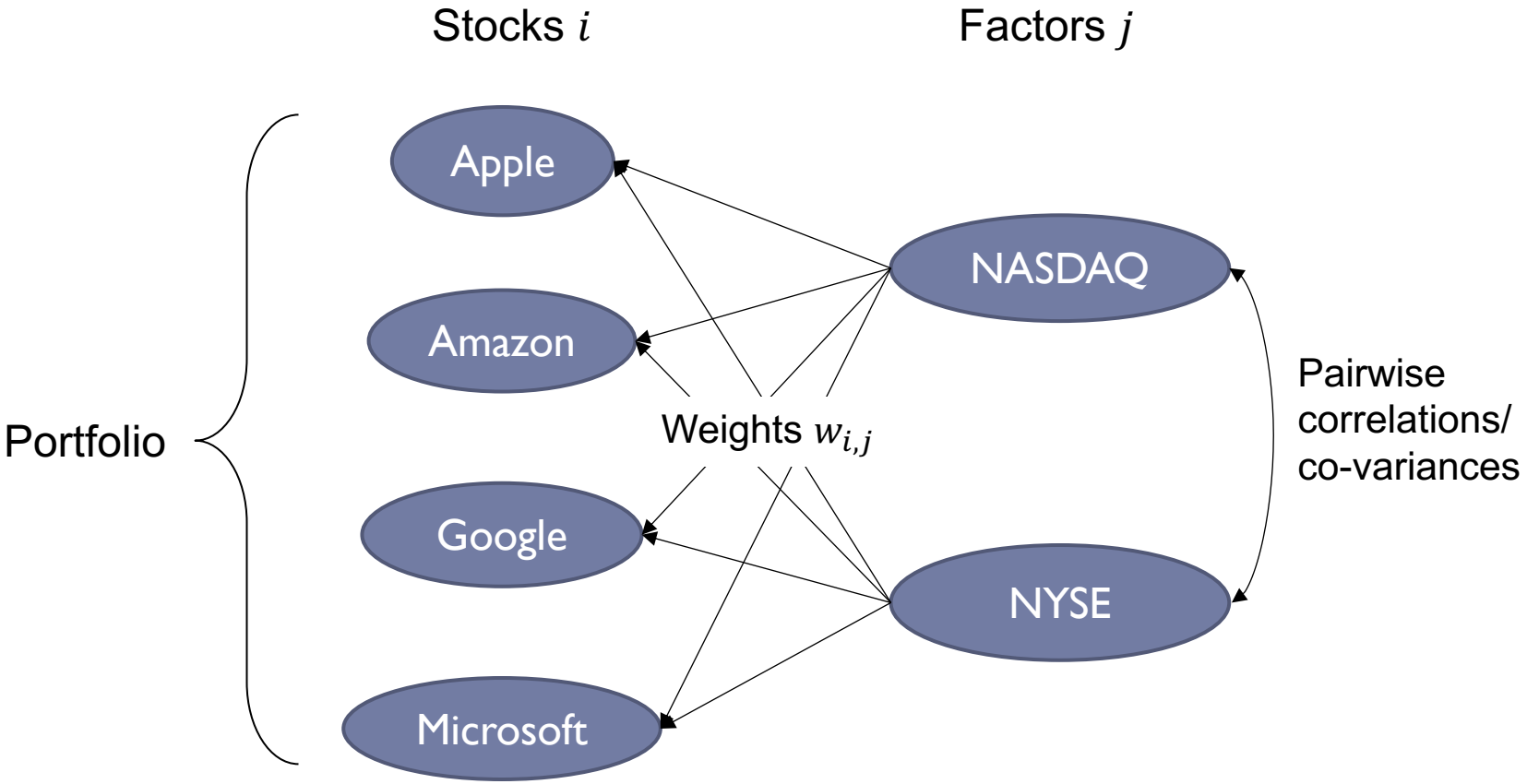
Example:

- A VaR of 1 million dollars at a 5%-quantile of the worst estimated returns over a 2-week period indicates that the entire portfolio has a 5% chance of losing more than 1 million dollars within 2 weeks.

Notes:

- Stocks typically do not react independently to changes in the market conditions, since a change in each market condition may affect multiple stocks.
- Even the market conditions may not develop independently of each other (see all the major crashes in the past).
- The number of combinations of possible market conditions is typically too large to be searched exhaustively to calculate the VaR directly.

Stocks vs. Factors



- A portfolio is a collection of stocks we may wish to invest in.
- Various market factors have an impact on each stock individually.
- Market factors are generally not independent (captured as pairwise correlations).

Time-Series Data for Stocks & Factors

Both stocks and factors are represented as **time-series data** (consisting of at least a *time-stamp* and a *measureable unit*, e.g., the market value of the stock or an entire index) at that time-stamp.

```
Date,       Open,  High,  Low,   Close, Volume
31-Dec-13, 79.17, 80.18, 79.14, 80.15, 55819372
30-Dec-13, 79.64, 80.01, 78.90, 79.22, 63407722
27-Dec-13, 80.55, 80.63, 79.93, 80.01, 56471317
26-Dec-13, 81.16, 81.36, 80.48, 80.56, 51002035
24-Dec-13, 81.41, 81.70, 80.86, 81.10, 41888735
23-Dec-13, 81.14, 81.53, 80.39, 81.44, 125326831
20-Dec-13, 77.91, 78.80, 77.83, 78.43, 109103435
....
```

$$r_i = \frac{80.01 - 78.43}{80.01} = 0.02$$

(for a 1-week period)

(The above CSV format is the one that was used by Google Finance to export portfolios until May 2018; we will extract only the Date and Close fields.)

Monte Carlo Simulations

The previous regression formula serves two purposes:

1. From the historical observations of stocks and factors, we can calculate the returns r_i and the features f_j of the portfolio we are interested in. We can use these to **train the parameters** c_i and $w_{i,j}$ of the linear-regression model.

2. To calculate the actual VaR, we feed the same regression formula (now with the learned parameters) with a large number of samples that **simulate all possible market conditions** to predict the estimated returns under these conditions.
   - From these samples, we obtain a distribution of the aggregated (e.g., using *sum* or *avg*) returns r_i of all stocks in our portfolio. The 5%-quantile of this distribution then is the VaR.
   - This repeated form of sampling is known as a **Monte Carlo Simulation**.

The Multivariate Normal Distribution

In principle, we could simply generate the samples of all market conditions that go into our predictions independently of each other (e.g., "*S&P 500 drops by 5% and Dow Jones increases by 10%*" could be one such sample).

But this would violate the assumptions we made earlier...

To consider correlations among market conditions, we assume that the factors derived from these market conditions follow a **multivariate normal distribution,** and we **sample the factors** according to this distribution.

- That is, we assume the factors $X = (x_1, \ldots, x_k)$ to follow $X \sim N(\mu, \Sigma)$ with **means vector** μ, **pairwise co-variance matrix** Σ and the following **probability density function**:

$$f_{\mathbf{X}}(x_1, \ldots, x_k) = \frac{\exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)}{\sqrt{(2\pi)^k|\boldsymbol{\Sigma}|}}$$

- Both μ and Σ can be estimated from our observations of the previous market conditions as well.

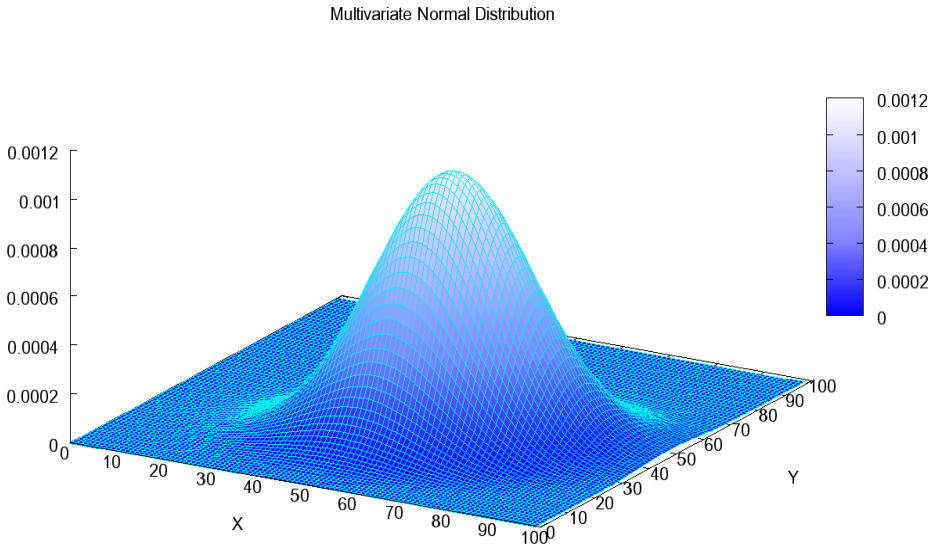


Image source: Wikipedia

Datasets for Stocks and Factors

As mentioned earlier, both stocks and factors are represented as time-series data in the Google Finance format.

- The stock files in Moodle capture the stock prices of **Amazon** (AMZN.csv), **Apple** (AAPL.csv), **Google** (GOOGL.csv) and **Microsoft** (MSFT.csv) between 1-Jan-2000 and 31-Dec-2013.
- The factor files contain the **iShares 20+ Year Treasury Bond ETF** (NASDAQ-TLT), the **iShares U.S. Credit Bond ETF** (NYSEARCA-CRED.csv) and the **SPDR Gold Shares** (NYSEARCA-GLD.csv) from a similar time range.
- Both the stock and factors files together merely consist of about **20,000 lines of CSV data**…

Parsing the Time-Series Files (I)

- We start by defining the usual parsing and cleaning functions in Scala. Note that, since the input files are not too large, we may just start with a regular **file-based API** in Scala rather than by creating an RDD.
- We will treat both stock and factor files exactly analogously in the following steps.

```
import java.io.File
import java.time.LocalDate
import java.time.format.DateTimeFormatter

// reads a stock history in the Google Finance time-series format
def readGoogleHistory(file: File): Array[(LocalDate, Double)] = {
  val formatter = DateTimeFormatter.ofPattern("d-MMM-yy")
  val lines = scala.io.Source.fromFile(file).getLines().toSeq
  lines.tail.map { line =>
    val cols = line.split(',')
    val date = LocalDate.parse(cols(0), formatter)
    val value = cols(4).toDouble
    (date, value)
  }.reverse.toArray
}
```

Parsing the Time-Series Files (II)

- We read the files from the stocks directory one-by-one, thereby catching possible exceptions:

```
val stocksDir = new File("./stocks/")
val files = stocksDir.listFiles()
val allStocks = files.iterator.flatMap { file =>
  try {
    Some(readGoogleHistory(file))
  } catch {
    case e: Exception => None
  }
}
```

- And we finally filter out only time-series of stocks that capture more than 5 years of trading days:

```
val rawStocks = allStocks.filter(_.size >= 260 * 5 + 10)
```

Trimming the Time-Series Data

- Next, the time-series data of each input file is trimmed to also match a given range of dates:

```
def trimToRegion(history: Array[(LocalDate, Double)],
    start: LocalDate, end: LocalDate): Array[(LocalDate, Double)] = {
  var trimmed = history.dropWhile(_._1.isBefore(start)).
    takeWhile(x => x._1.isBefore(end) || x._1.isEqual(end))
  if (trimmed.head._1 != start) {
    trimmed = Array((start, trimmed.head._2)) ++ trimmed
  }
  if (trimmed.last._1 != end) {
    trimmed = trimmed ++ Array((end, trimmed.last._2)) }
  trimmed }
```

- And here is the actual call of the function using a `map` transformation:

```
val start = LocalDate.of(2009, 10, 23)
val end = LocalDate.of(2014, 10, 23)
val trimmedStocks = rawStocks.map(trimToRegion(_, start, end))
```

Auto-Completing the Time-Series Data

- To compare the returns of stocks with their factors, both types of time-series have to be perfectly aligned. We achieve this by imputing missing values into the time-series data from the input files:

```
def fillInHistory(history: Array[(LocalDate, Double)],
    start: LocalDate, end: LocalDate): Array[(LocalDate, Double)] = {
  var cur = history
  val filled = new ArrayBuffer[(LocalDate, Double)]()
  var curDate = start
  while (curDate.isBefore(end)) {
    if (cur.tail.nonEmpty && cur.tail.head._1 == curDate)
      cur = cur.tail
    filled += ((curDate, cur.head._2))
    curDate = curDate.plusDays(1)
    if (curDate.getDayOfWeek.getValue > 5) // skipping weekends!
      curDate = curDate.plusDays(2) }
  filled.toArray }
```

- And create the final stocks time-series:

```
val stocks = trimmedStocks.map(fillInHistory(_, start, end))
```

Using Sliding Windows to Compute Bi-Weekly Returns

- We will fix our analysis to **bi-weekly returns** for all of the following steps.
- To do so, we implement a **sliding window** that computes (overlapping) bi-weekly returns over each of the stocks' and factors' time-series:

```
def twoWeekReturns(history: Array[(LocalDate, Double)]):
      Array[Double] = {
  var i = 0
  history.sliding(10).map { window =>
    val next = window.last._2
    val prev = window.head._2
    i += 1
    (next - prev) / prev
  }.toArray
}

val stockReturns = stocks.map(twoWeekReturns).toArray.toSeq
val factorReturns = factors.map(twoWeekReturns).toArray.toSeq
```

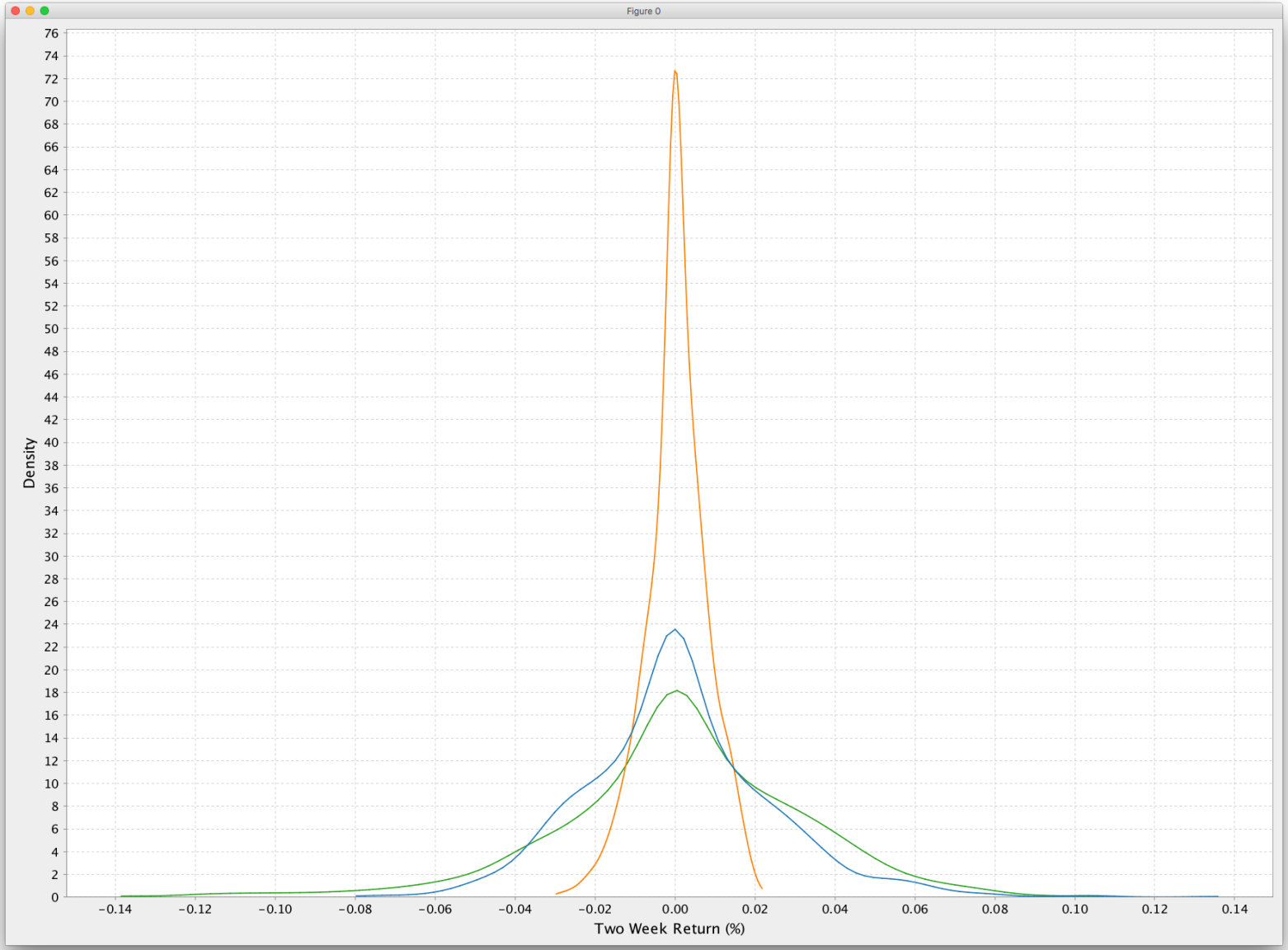
- Note that a "history" here refers to the observed time-series data of a stock or factor file. The `sliding` operator creates an overlapping sliding window over these. Compare: `List(1,2,3,4,5).sliding(3).foreach(println)`

Plotting the Distributions of Factor Returns

- The `breeze` and `jfree` libraries offer a variety of options to draw plots in Scala.
- Note that the `KernelDensity` class of Spark is used here to smooth the curve given by the raw samples of factor returns.

```
def plotDistributions(sampleSets: Seq[Array[Double]]): Figure = {
  val f = Figure()
  val p = f.subplot(0)
  …
  for (samples <- sampleSets) {
    val min = samples.min
    val max = samples.max
    val stddev = new StatCounter(samples).stdev
    val bandwidth = 1.06 * stddev * math.pow(samples.size, -.2)
    val domain = Range.Double(min, max, (max - min) / 100).toList.toArray
    val kd = new KernelDensity().
      setSample(sc.parallelize(samples)).setBandwidth(bandwidth)
    val densities = kd.estimate(domain)
    p += plot(domain, densities) }
  f }
plotDistributions(factorReturns)
```

Distributions of Factor Returns



Computing the Factor Means and Co-Variances

We now pursue in three steps to **fit the parameters** of our multivariate-normal-distribution from the factor returns:

1. We compute the means of the distributions directly from the `factorReturns`.
2. We compute a `factorMatrix` object that captures the three arrays with the factor returns in (transposed) matrix form.
3. From the `factorMatrix`, we compute also the pairwise co-variances of the factor returns.

```
import org.apache.commons.math3.stat.correlation.Covariance

def factorMatrix(histories: Seq[Array[Double]]): Array[Array[Double]] = {
   val mat = new Array[Array[Double]](histories.head.length)
   for (i <- histories.head.indices) {
     mat(i) = histories.map(_(i)).toArray
   }
   mat }

val factorMeans = factorReturns.map(factor => factor.sum / factor.size).toArray
val factorMat = factorMatrix(factorReturns)
val factorCov = new Covariance(factorMat).getCovarianceMatrix().getData()
```

"Featurizing" the Factor Returns

- To turn the factor returns we calculated from the time-series data into the actual features f_j, which we will feed into our linear-regression model, we include also the *squares* and the *square roots* of the factor returns into the final features.

```
// include additional feature functions such as square, square root, etc.
def featurize(factorReturns: Array[Double]): Array[Double] = {
  val squaredReturns = factorReturns.map(x => math.signum(x) * x * x)
  val squareRootedReturns = factorReturns.map(x => math.signum(x) *
     math.sqrt(math.abs(x)))
  squaredReturns ++ squareRootedReturns ++ factorReturns
}
```

- And we invoke the learning step for the linear-regression model over both the observed stock and feature returns:

```
// learn the parameters of the linear-regression model
val factorFeatures = factorMat.map(featurize)
val factorWeights = computeFactorWeights(stockReturns, factorFeatures)
```

Training the Linear-Regression Model

We pursue in a similar manner to also **fit the parameters** of our linear-regression model to later predict the stock returns from the simulated factor returns:

1. We compute the means of the distributions directly from the `factorReturns`.
2. We compute a `factorMatrix` object that captures the three arrays with the factor returns in (transposed) matrix form.
3. From the `factorMatrix`, we compute also the pairwise co-variances of the factor returns.

```
import org.apache.commons.math3.stat.regression.OLSMultipleLinearRegression
def linearModel(instrument: Array[Double], factorMatrix: Array[Array[Double]]):
     OLSMultipleLinearRegression = {
  val regression = new OLSMultipleLinearRegression()
  regression.newSampleData(instrument, factorMatrix)
  regression
}
def computeFactorWeights(
  stocksReturns:  Seq[Array[Double]],
  factorFeatures: Array[Array[Double]]): Array[Array[Double]] = {
  stocksReturns.map(linearModel(_,
     factorFeatures)).map(_.estimateRegressionParameters()).toArray }
```

Parallel Sampling

- To facilitate a form of **parallel sampling in Spark**, we distribute both the stock and factor returns by turning them into an RDD (specifically: a `Dataset`).
- Parallel sampling requires the repeated generation of random numbers by some form of random-number generator (here: a `MersenneTwister`).
- To avoid a repeated sampling of the same values, we initialize a new random-number generator for each partition of the RDD with a different seed value.

```
val numTrials = 1000000
val parallelism = 100
val baseSeed = 1001L

import org.apache.spark.sql.Dataset
import org.apache.spark.sql.SQLContext

val sqlContext = new SQLContext(sc)
import sqlContext.implicits._

val seeds = (baseSeed until baseSeed + parallelism)
val seedsDS = seeds.toDS().repartition(parallelism)
```

Spark's Dataset API

- Spark 1.6 introduced the `Dataset` API as an extension of the `DataFrame` API.
- Unlike a `DataFrame`, a `Dataset[T]` is a strongly typed data structure and thereby allows for better compile-time type safety than a `DataFrame` (which, internally, is actually an alias for a `Dataset[Row]`, where the type of `Row` however need not explicitly be specified by the programmer).
- Both are backed up internally by an `RDD` and thus fully support all `RDD` transformations and actions.
- By importing `spark.implicits._` one can cast dynamically among an `RDD`, `Dataset` and `DataFrame` using `toDS()` and `toDF()`, respectively.
- By accessing `Dataset.agg`, one has access to a number of SQL-style aggregation functions.

  Example:

```
samples.agg(functions.min($"value"), functions.max($"value"))
```

- By accessing `Dataset.stat`, one has access to a number of statistical functions.

  Example:

```
samples.approxQuantile("value", Array(0.05), 0.0))
```

Running the Sampler

- The `trialReturns` function computes a number of samples (called "trials") from the assumed multivariate normal distribution which is initialized with the same parameters for all partitions (but each with a different random-number generator).

```
def trialReturns(
     seed:       Long,
     numTrials:  Int,
     instruments:Seq[Array[Double]],
     factorMeans:Array[Double],
     factorCov:  Array[Array[Double]]): Seq[Double] = {
    val rand = new MersenneTwister(seed)
    val mvn = new MultivariateNormalDistribution(rand, factorMeans, factorCov)
    val trialReturns = new Array[Double](numTrials)
    for (i <- 0 until numTrials) {
      val trialFactorReturns = multivariateNormal.sample()
      val trialFeatures = featurize(trialFactorReturns)
      trialReturns(i) = trialReturn(trialFeatures, instruments) }
    trialReturns }

val trialsDS = seedDS.flatMap(trialReturns(_, numTrials / parallelism, factorWeights,
                                           factorMeans, factorCov))
```

Aggregating the Trial Returns

- We are now ready to **predict the average stock return** of each stock in our portfolio for a sampled set of factor returns: $r_{avg} = \frac{1}{n}\sum_{i=1}^{n}(c_i + \sum_{j=1}^{m} w_{i,j}\, f_j)$
- An "instrument" here denotes the array of weights $w_{i,j}$ learned for each stock i.
- A "trial" is the array of features f_j computed from the sampled factor returns; c_i is set to the first weight $w_{i,0}$ of each instrument.

```
def instrumentTrialReturn(instrument: Array[Double], trial: Array[Double]):
     Double = {
    var instrumentTrialReturn = instrument(0)
    var i = 0
    while (i < trial.length) {
      instrumentTrialReturn += trial(i) * instrument(i + 1)
      i += 1 }
  instrumentTrialReturn }

def trialReturn(trial: Array[Double], instruments: Seq[Array[Double]]): Double = {
  var totalReturn = 0.0
  for (instrument <- instruments) {
    totalReturn += instrumentTrialReturn(instrument, trial) }
  totalReturn / instruments.size }
```

Calculating the VaR

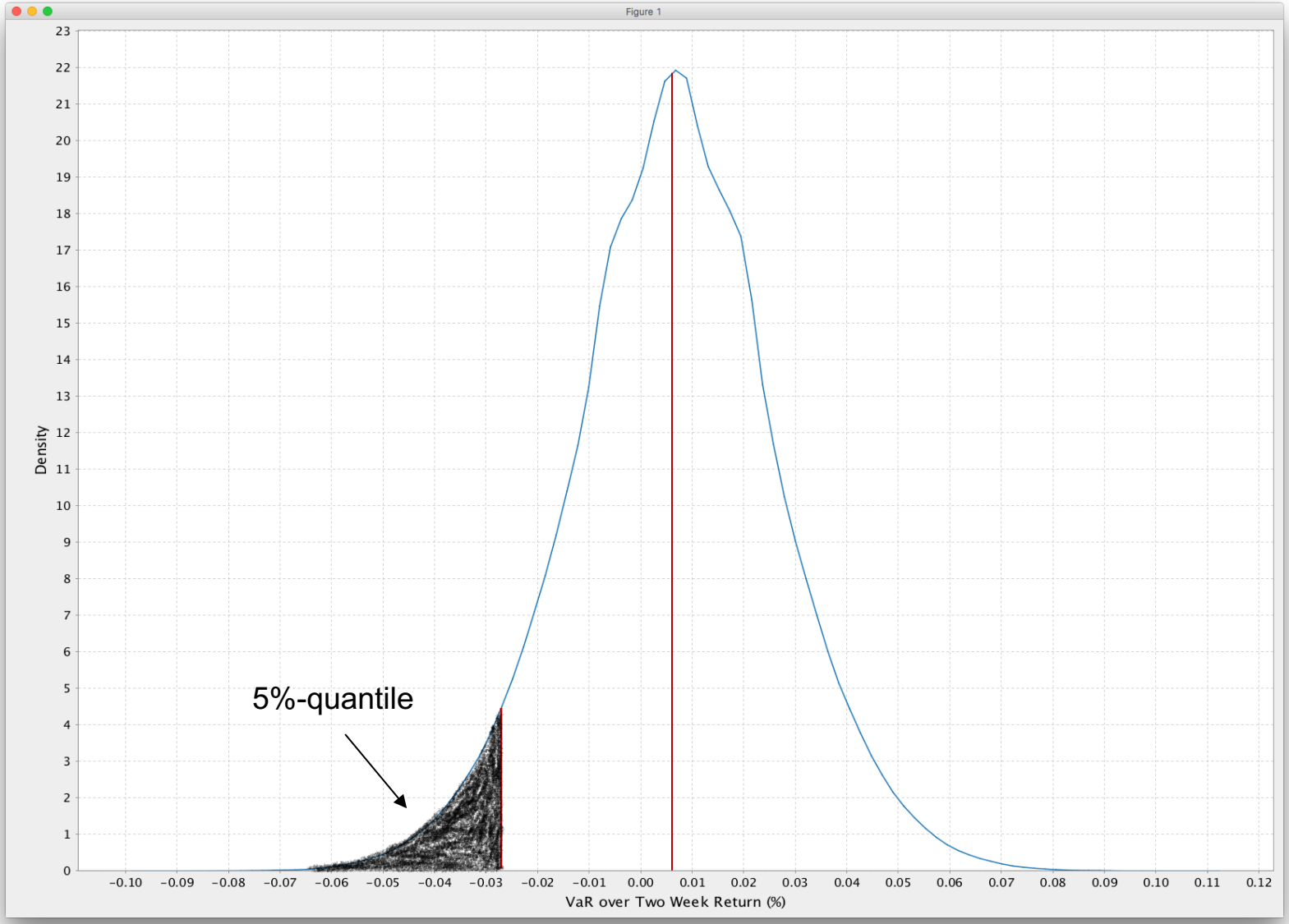
- The VaR now confirms to the **5%-quantile of the distribution of the average stock returns** that we estimated by the trials from the previous step.
- This can also very conveniently be obtained from the `Dataset` API as follows:

```
def fivePercentVaR(trialsDS: Dataset[Double]): Double = {
  val quantiles = trialsDS.stat.approxQuantile("value", Array(0.05), 0.0)
  quantiles.head
}

val valueAtRisk = fivePercentVaR(trialsDS)
```

- About 200 lines of Spark/Scala code and ca. 100,000,000 Monte Carlo samples later…

Plotting the VaR for our Portfolio (AMZN/AAPL/GOOGL/MSFT)



The 5%-quantile is at about -0.026.

Conditional Value-at-Risk

- Similarly, the **Conditional Value-at-Risk** (CVaR) is defined as the *average* over the 5%-quantile of the worst returns:

```
def fivePercentCVaR(trialsDS: Dataset[Double]): Double = {
  val topLosses = trialsDS.orderBy("value").limit(math.max(trialsDS.count().
     toInt / 20, 1))
  topLosses.agg("value" -> "avg").first()(0).asInstanceOf[Double]
}

val conditionalValueAtRisk = fivePercentCVaR(trialsDS)
```

Summary

- VaR and CVaR are only two of the commonly applied statistics to assess financial risk.
- VaR has also been criticized in the following ways:
  - Rare events, such as major crashes, still cannot be predicted by VaR alone.
  - VaR may give overly high confidence in high-risk investments and thus may lead to excessive risk-taking by investors.
- Both of the measures however demonstrate an interesting **combination of machine-learning techniques** (learning factors and weights from historical data) **and more traditional statistical approaches** (computing co-variances and sampling from an assumed distribution) to simulate future events.
- **Distributed sampling** is "naturally" facilitated here by Spark's `RDD` and `Dataset` APIs.


Data
Analytics
Big
Hadoop
Spark
Prediction
Distribution
Matrix
Scalability
Predictive
Sparse
Scala
Trees
GraphX
Rank
GFS
Purity
Systems
JSON
XML
MLlib
Sampling
Variable
Random
GeoJSON
Recommender
Analytics
Science
Ham
Confusion
Clustering
Reduce
Parallelism
Language
RDD
Streaming
Multivariate
AudioScrobbler
Twitter
MeSH
Recall
Broadcast
Precision
HDFS
Visualization
Decision
Validation
Triangles
Statistical
Components
Value
Normal
Classification
Outlier
Combiner
Action
Gradient
ALS
CoverType
Partitioner
GPS
Cosine
Java
Eigenvalue
Cross
Vector
Entropy
Semantic
HITS
Efficiency
Map
Linear
Networks
Testsal
WordCount
Decomposition
Trips
Connected
API
Natural
Forests
Transformation
Cliques
Latent
Processing
Gini
Singular
Medline
Taxi
Medical
Weather
Detection
Strongly
Risk
Financial
Eigenvector
Factorization
PageRank
Anomaly
Probability
Accumulator
Confidence
Dense
DStream
K-Means
Spam
Regression
Graphs
DataFrame
Dataset
Dataflow
Boosted
SQL
Open
NoSQL
GIS