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

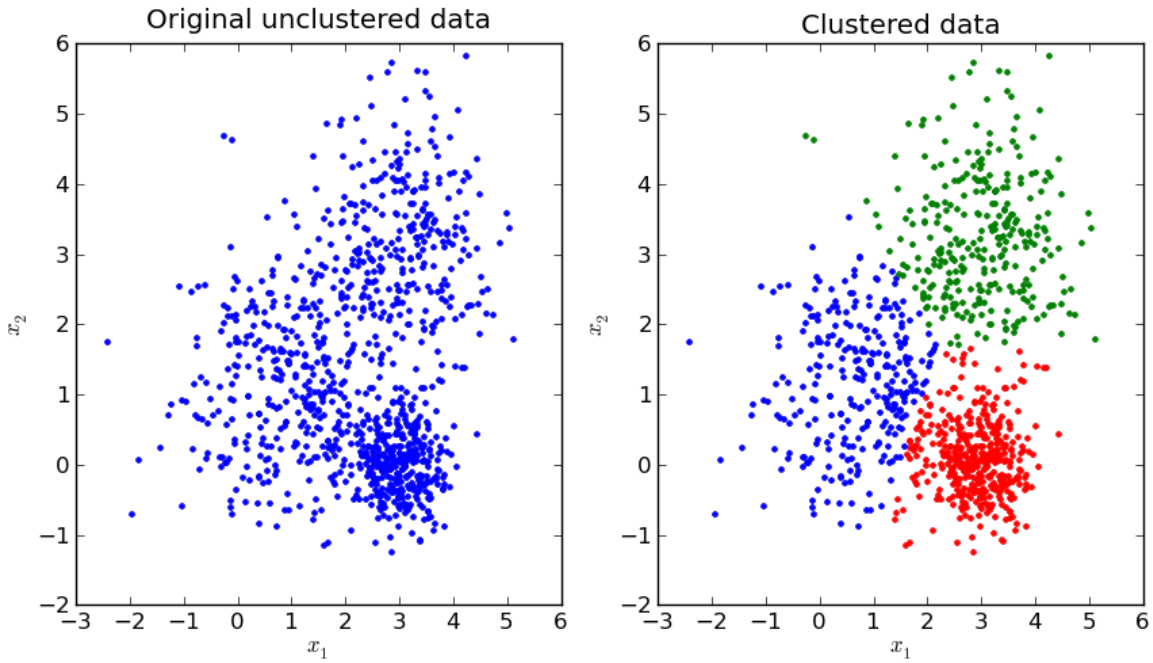
Clustering Objective

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)

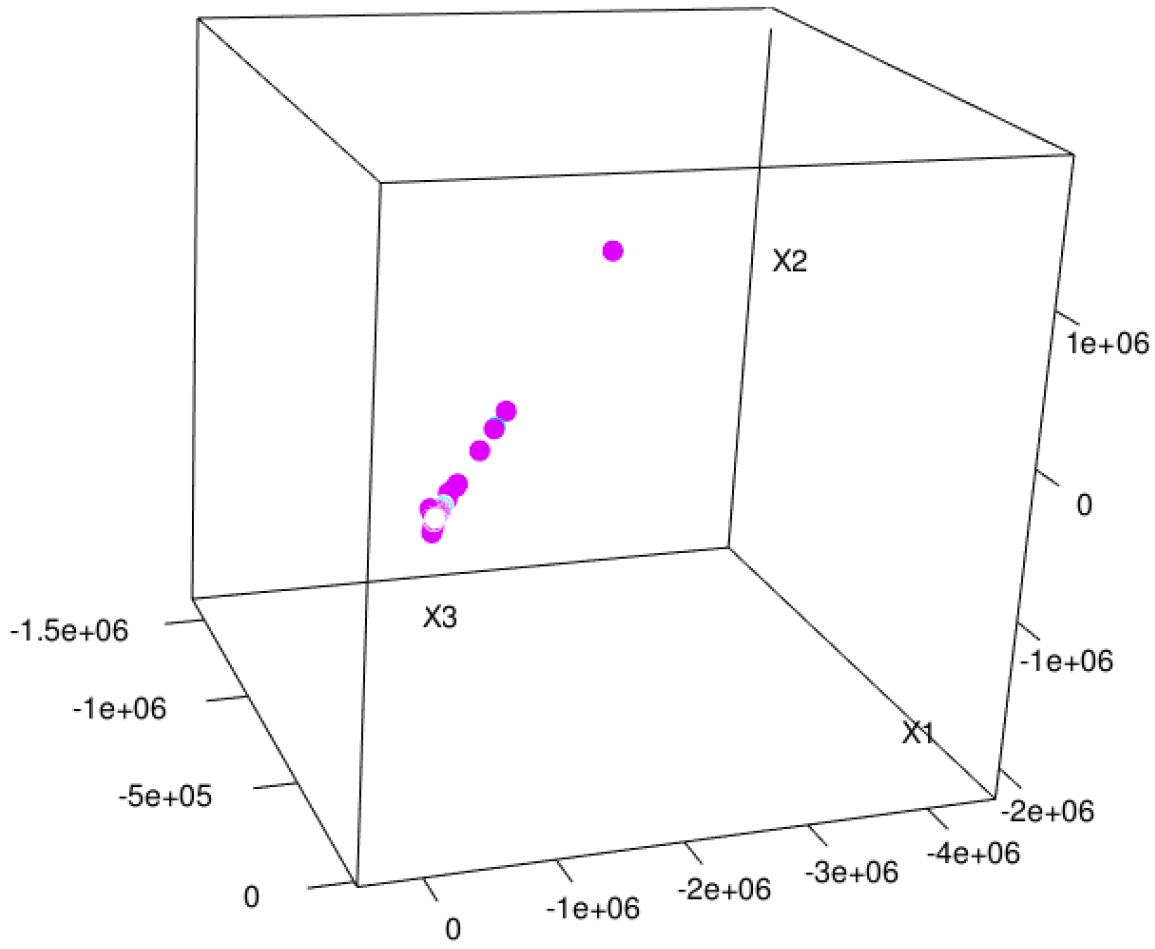
- By switching to an R shell, we can create a **3d-plot** (using a random projection of the 38 feature dimensions) of our 1%-sample of the data:

```
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

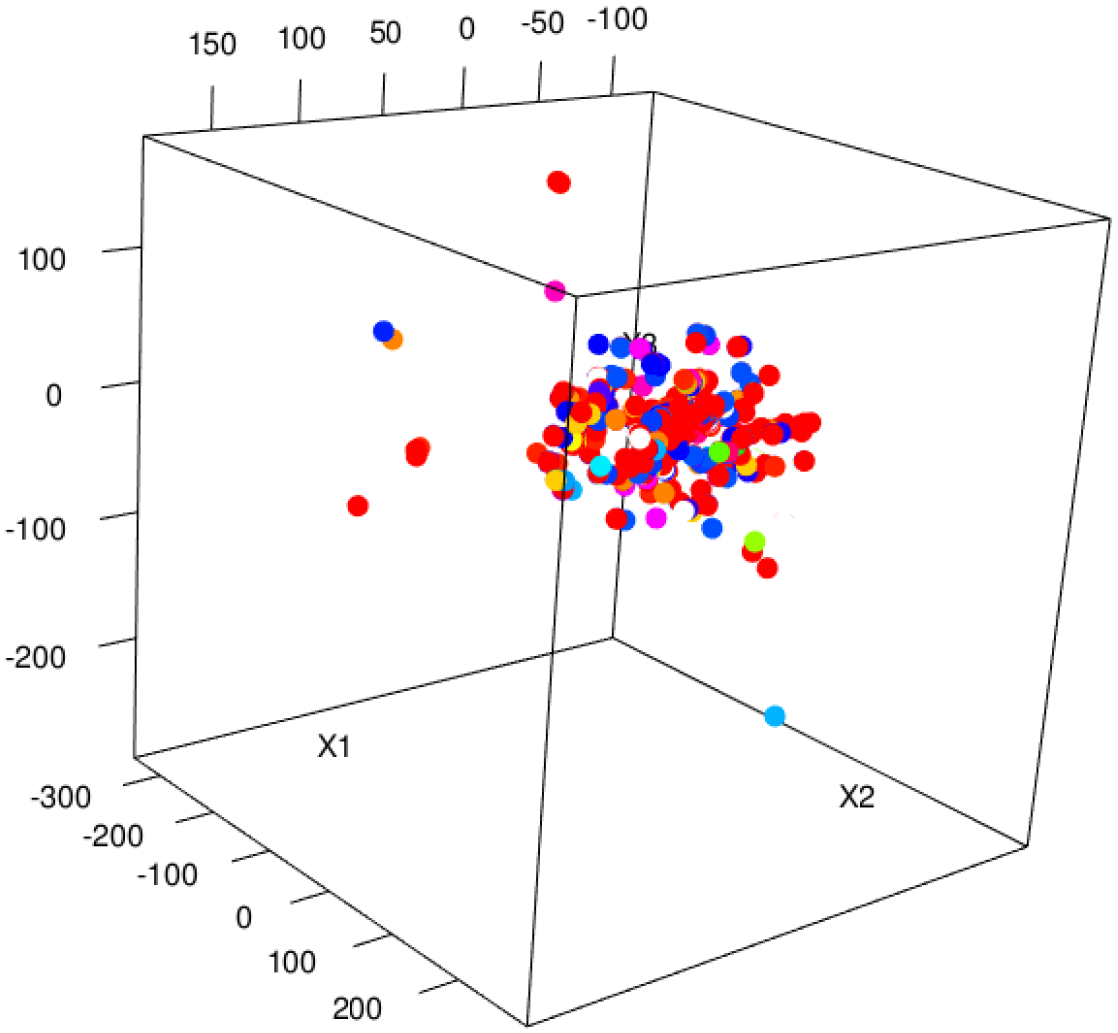
- We can now run the same test with normalized data, also on a higher range of k:

```
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)

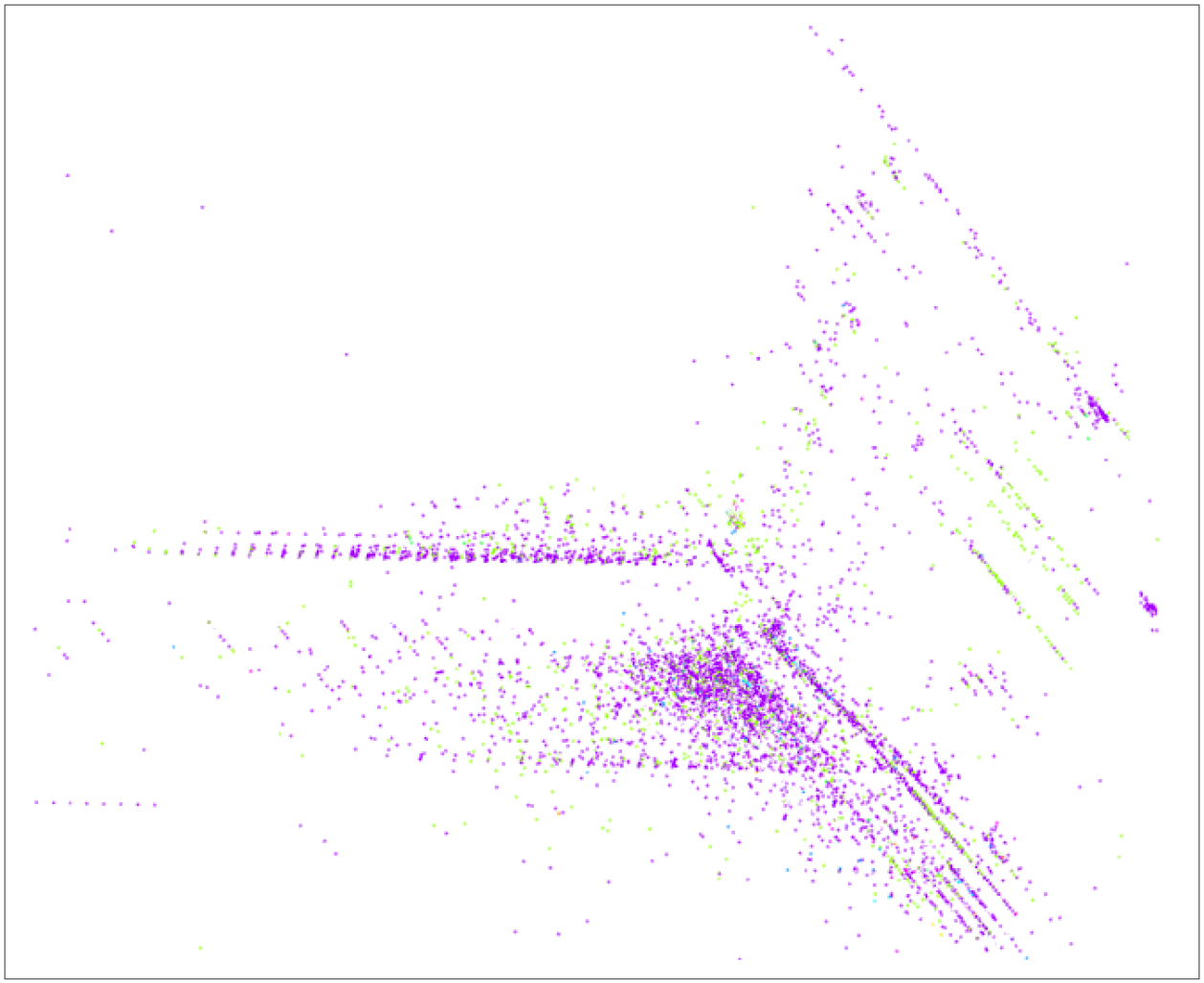
- The normalized data reveals a much richer structure than before:

$k = 100$



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…