ISCTE IUL
University Institute of Lisbon

Department of Information Science and Technology

Anomaly Detection in Network Traffic with K-means Clustering using PySpark

Algorithms for Big Data

MSc in Computer Engineering
Information Systems and Knowledge Management

Tiago Filipe Martinho Soares

January 2020

Contents

Introduction

This work is meant to show how one can use PySpark to solve machine learning problems, in this case, anomaly detection in network traffic with K-means clustering.
The problem and its respective solution are described in Sandy Ryza's "Advance Analytics with Spark: Patterns for Learning From Data at Scala" book, chapter 5.
However, in this book, the solution is written in the Scala programming language while the solution presented in this report is written in Python. One of the sections of the chapter, "Visualization", is not followed strictly, since the author uses RStudio for a specific use case of visualization. In this project, visualization of the data was made through the entire notebook, and also a separate visualization notebook using python libraries like Matplotlib and Seaborn.
As previously mentioned, the algorithm used for solving this problem is K-means clustering, an unsupervised learning technique and one of the most popular clustering algorithms. This algorithm tries to find *k* clusters in a dataset, a hyperparameter given by the data scientist trying to solve the problem.
Finding a good value for *k* is one of the main points of this project and it allows for grouping of connections based on their type. Any connection not close to one of these groups/clusters is a potential anomaly in the dataset.

Dataset retrieval

The datasets used for this project are listed in the "References" section.
The reference includes a list of features for the dataset (kddcup.names), the full data set (kddcup.data.gz) and a 10% subset of this dataset (kddcup.data_10_percent.gz) which was used on the first stages of the project to test the code and save some time.
Other resources like new test data are included in the URL previously provided.
The full data set has 708MB in size and contains about 4.9 million connections, meaning, 4.9 million lines/rows in CSV format, each with 38 features.

Here's a connection example:

0,tcp,http,SF,181,5450,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,8,8,0.00,0.00,0.00,0.00,1.00,0.00,0.00,9,9,1.00,0.00,0.11,0.00,0.00,0.00,0.00,0.00,normal.

This connection shows a TCP connection to an HTTP service, with 181 bytes sent and 5450 bytes received. Most of the remaining features indicate the presence or absence of a behaviour, with the values 1 or 0, respectively.
The last field is a label, which on this connection example has the value "normal". Since the goal is to find new and unknown attacks present in the data, this label will not be useful to determine if a connection is an anomaly.

Amazon AWS

It's possible to work with the dataset externally using Amazon AWS. By accessing the AWS Management Console, one can access services like S3 for scalable storage in the cloud and EMR for a Managed Hadoop Framework.

Uploading the dataset

Using the S3 service, it's possible to create a bucket to hold user files. After naming and creating the bucket the user should select that bucket and click the "Upload" button to upload the file. It's possible to set permissions, the storage class and even the encryption for the file. In this case, public access is blocked, the storage class is Standard and there is no encryption set. After the dataset is uploaded, the user should have a screen like the following one:

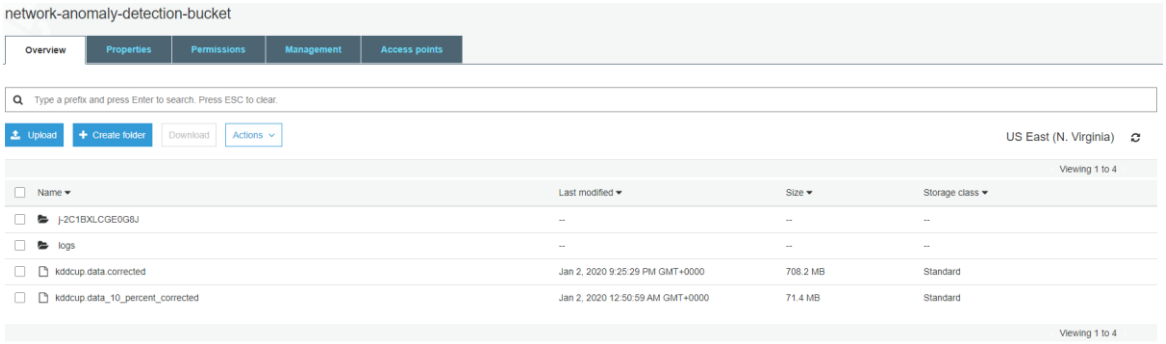


Figure 1 – S3 bucket containing user uploaded files

The user can then copy the file path by selecting the file and clicking the button "Copy Path".

Creating the cluster

Using the EMR service, it's possible to create a cluster and work with the dataset on a Jupyter Notebook. The user should click on the "Create cluster" button, name it and click on "Go to advanced options". The software configuration should include Hadoop, Hive, Spark and Livy services. In "General Options" it's possible to enable logging (and select a folder to save the logs), debugging and termination protection. Finally, for security, the user can select an EC2 key pair created with the EC2 service and set other permissions and security configuration.

In order to avoid session timeouts the following cluster configuration was used:

Classification ▲	Property	Value
livy-conf	livy.server.session.timeout-check	true
livy-conf	livy.server.session.timeout	5h
livy-conf	livy.server.yarn.app-lookup-timeout	180s

Figure 2 – Livy session JSON configuration

Creating the notebook and loading the dataset

After the cluster is created, the user can create a Jupyter Noteebok on the "Notebooks" tab and clicking the "Create notebook button". The user can set the name, choose the cluster previously created, define the location of the notebook on S3, etc. The user can then open the notebook by clicking the buttons "Open in Jupyter" or "Open in JupyterLab". By typing "spark" on the notebook, the user can validate if there's an active session. After that, the user can read the dataset by pointing to the dataset path copied before.

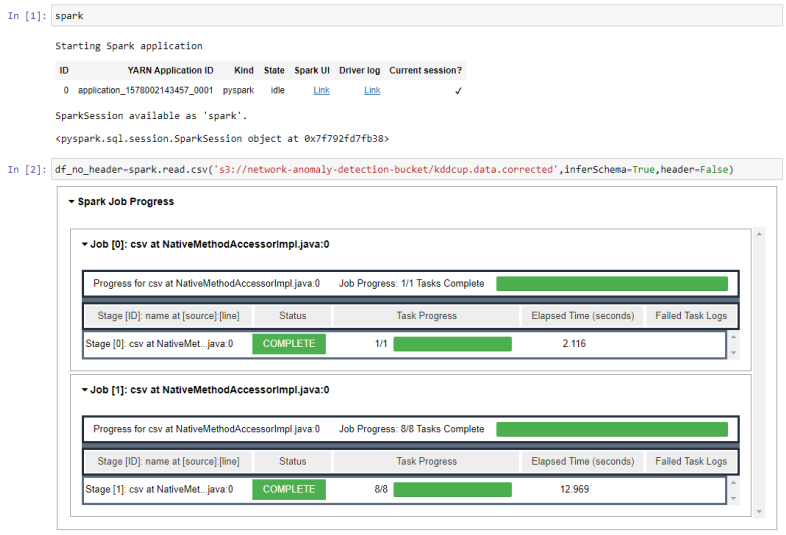


Figure 3 – Created notebook with dataset loaded

Analysis and processing

Loading the dataset locally

If the user is working locally, before reading the dataset it's necessary to put it into the Hadoop Distributed File System (HDFS. This can be accomplished by running the following command on the directory where the dataset is present:

hdfs dfs -put kddcup.data.corrected

After the dataset is in the HDFS it can be loaded along with the feature names to a DataFrame. The DataFrame will have the following structure:

```
Out[5]: DataFrame[duration: int, protocol_type: string, service: string, flag: string, src_bytes: int, dst_bytes:
        int, land: int, wrong_fragment: int, urgent: int, hot: int, num_failed_logins: int, logged_in: int, num_co
        mpromised: int, root_shell: int, su_attempted: int, num_root: int, num_file_creations: int, num_shells: in
        t, num_access_files: int, num_outbound_cmds: int, is_host_login: int, is_guest_login: int, count: int, srv
        _count: int, serror_rate: double, srv_serror_rate: double, rerror_rate: double, srv_rerror_rate: double, s
        ame_srv_rate: double, diff_srv_rate: double, srv_diff_host_rate: double, dst_host_count: int, dst_host_srv
        _count: int, dst_host_same_srv_rate: double, dst_host_diff_srv_rate: double, dst_host_same_src_port_rate:
        double, dst_host_srv_diff_host_rate: double, dst_host_serror_rate: double, dst_host_srv_serror_rate: doubl
        e, dst_host_rerror_rate: double, dst_host_srv_rerror_rate: double, label: string]
```

Figure 4 – DataFrame column names and types

Initial Clustering

To understand the dataset better, the connections may be grouped by their label value.

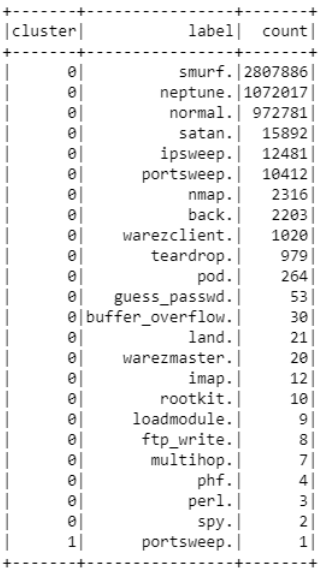
```
+----------------+-------+
|label           |count  |
+----------------+-------+
|smurf.          |2807886|
|neptune.        |1072017|
|normal.         |972781 |
|satan.          |15892  |
|ipsweep.        |12481  |
|portsweep.      |10413  |
|nmap.           |2316   |
|back.           |2203   |
|warezclient.    |1020   |
|teardrop.       |979    |
|pod.            |264    |
|guess_passwd.   |53     |
|buffer_overflow.|30     |
|land.           |21     |
|warezmaster.    |20     |
|imap.           |12     |
|rootkit.        |10     |
|loadmodule.     |9      |
|ftp_write.      |8      |
|multihop.       |7      |
|phf.            |4      |
|perl.           |3      |
|spy.            |2      |
+----------------+-------+
```

Figure 5 – Label values and number of occurrences

It's possible to see that most connections are labelled with the "smurf" and "neptune" values, which are DDoS and SYN flood attacks respectively.

Since K-means clustering requires numeric features, these either must be dropped or transformed into nonnumeric features. For now, features like “protocol_type” and “service” will be ignored.

Using a VectorAssembler to create a feature vector, a standard KMeans implementation to create the model and setting these as stages for a Pipeline it’s possible to run a simple clustering task.
With no processing, besides dropping nonnumeric features, we get the following results:



cluster	label	count
0	smurf.	2807886
0	neptune.	1072017
0	normal.	972781
0	satan.	15892
0	ipsweep.	12481
0	portsweep.	10412
0	nmap.	2316
0	back.	2203
0	warezclient.	1020
0	teardrop.	979
0	pod.	264
0	guess_passwd.	53
0	buffer_overflow.	30
0	land.	21
0	warezmaster.	20
0	imap.	12
0	rootkit.	10
0	loadmodule.	9
0	ftp_write.	8
0	multihop.	7
0	phf.	4
0	perl.	3
0	spy.	2
1	portsweep.	1

Figure 6 – Clusters formed

Only 2 clusters were formed (k=2), which is clearly not enough. Not only that, only one data point was assigned to cluster 1 while all others were assigned to cluster 0. This is an example of bad clustering.
Many values of k must be tried in order to have a good clustering. Clustering is considered good if each data point is “near” its closest centroid. The main idea is to minimize the cost, in this case, the sum of squared distances of points to their nearest centroid.

Note: The book proposes the usage of the computeCost method from pyspark.ml api to compute the training cost of the algorithm for each k. However, as pointed out in PySpark’s official documentation, this method is deprecated in version 2.4.0 and it will be removed in version 3.0.0. The cost is retrieved using the training cost calculated in the summary, as suggested in the official documentation. In fact, in the GitHub repository accompanying the book, the code is updated to reflect this API change.

Initially, the following values for k were tried: {20, 40, 60, 80, 100}. The following clustering function was used to compute the training cost for each value of k:

```
def clusteringScore0(df, k: int) -> float:
    assembler = VectorAssembler(inputCols=list(filter(lambda x: x != 'label', numericOnly.columns)), outpu
tCol='featureVector')
    kmeans = KMeans(predictionCol='cluster', featuresCol='featureVector').setSeed(randint(1, 10)).setK(k)
    pipeline = Pipeline(stages=[assembler, kmeans])
    kmeansModel = pipeline.fit(df).stages[-1]
    return kmeansModel.summary.trainingCost
```

Figure 7 – Base clustering function

And the results were the following:

K	Training cost
20	3888082742412247.0
40	614806655855173.5
60	54082862177276.9
80	54082861854051.6
100	81196455017251.8

Figure 8 – Clustering results using the clusteringScore0 function

The results should show that, as k increases, the cost decreases. This is the expected result since more clusters means more points closer to a centroid. However, in this case, the training cost increases from k=80 to k=100. This means that K-means is not able to find the optimal clustering for a given k. A solution for this can be increasing the number of iterations run (maxIter), preventing the algorithm from stopping too early and by decreasing the tolerance value (tol) that controls the minimum amount of centroid movement considered significant. The values of maxIter and tol were increased and decreased to 40 and 0.00001 from the default values of 20 and 0.0001, respectively. Increasing the number of iterations has obviously more computation costs.
The results were the following:

K	Training cost
20	1563494026082292.5
40	595053753658715.9
60	84528290382745.9
80	92512815090833.6
100	33838412826569.5

Figure 9 – Clustering results using the clusteringScore1 function

The score is now lower across the board except when k=80, which also should not happen. Like before, probably some suboptimal clustering happened when clustering with this value. In any case, the goal is to find a value of k, which by increasing it stops reducing the training cost substantially. This will provide the best clustering / computation cost ratio. This is easier to visualize with an elbow graph like the following (present in the notebook for all clustering score functions):

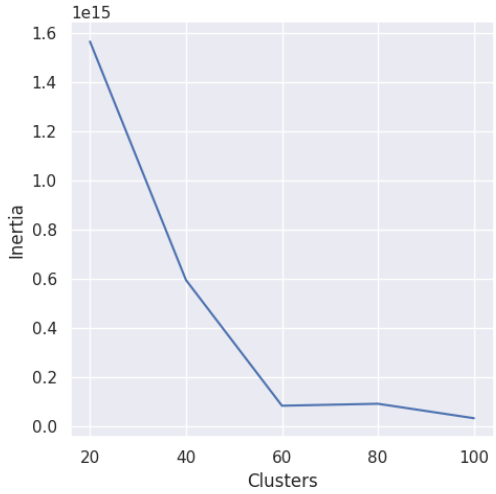


Figure 10 – Elbow graph for results of clusteringScore1 function

Using PCA (Principal Component Analysis), it's possible to have a better understanding of the data and the features that have a greater weight on calculations. Since visualization is not possible using a Spark DataFrame and it's not feasible to pull 4.9 million records to a non-distributed Pandas DataFrame only a sample of 1% (about 50.000 records) was used:

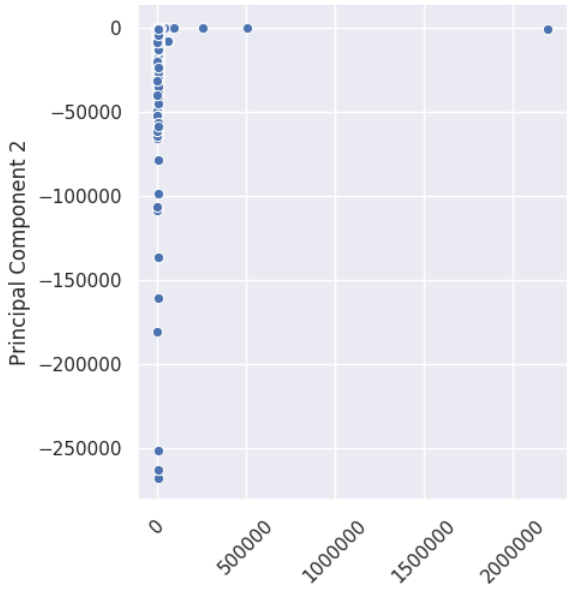


Figure 11 – Initial PCA plotting

This L-Shape makes sense since the features src_bytes and dst_bytes have values that are on a much larger scale than the other features, which have values 0 or 1. This also means that features must be normalized.

Feature Normalization

As with many machine learning algorithms, feature normalization can be very useful for clustering since it improves the numerical stability of the model and it changes the numeric distances between the nodes. Each feature value is converted to standard score using the formula:

$$normalized_i = \frac{feature_i - \mu_i}{\sigma_i}$$

Where μ is the mean of the feature's value and σ is the standard deviation.
Feature normalization can be applied programmatically by using the StandardScaler class.
After adding the StandardScaler to the clustering pipeline, the same test can be run with higher values of k. The results were the following:

K	Training cost
60	5957207.29
90	3608672.93
120	2721572.26
150	2114261.12
180	1618708.29
210	1383934.68
240	1263797.23
270	1076121.85

Figure 12 – Clustering results using the clusteringScore2 function

Now the training cost is much lower across the board and for every increase in k the training cost decreases. This is a step in the right direction but there's still no obvious value of k that can be chosen, although 180 might be a good candidate.

PCA was used again after a new clustering experiment:

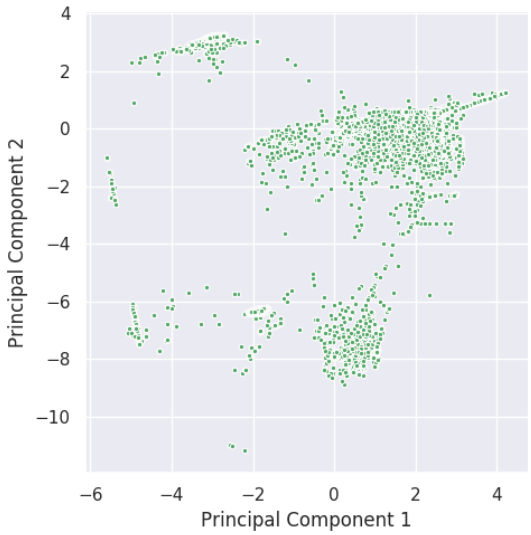


Figure 13 – PCA plotting after normalizing features

Normalization of features made a great difference since this plot reveals a richer structure and it's easier to distinguish some clusters of the others.

Categorical Variables

The nonnumeric features like "protocol_type" and "service" that were dropped before should be used since they provide valuable information. However, since K-Means clustering does not support nonnumeric features they must be transformed into numeric features. Using a StringIndexer and an OneHotEncoder it's possible to perform this transformation and encode each of the new numeric values into a vector. This small pipeline was incorporated in the main clustering function as follows:

```
def clusteringScore3(df, k: int) -> float:
    (protoTypeEncoder, protoTypeVecCol) = oneHotPipeline('protocol_type')
    (serviceEncoder, serviceVecCol) = oneHotPipeline('service')
    (flagEncoder, flagVecCol) = oneHotPipeline('flag')

    assembleCols = set(df.columns).difference({"label", "protocol_type", "service", "flag"}).union({protoT
ypeVecCol, serviceVecCol, flagVecCol})

    assembler = VectorAssembler(inputCols=list(assembleCols), outputCol='featureVector')
    scaler = StandardScaler(inputCol='featureVector', outputCol='scaledFeatureVector', withStd=True, withM
ean=False)
    kmeans = KMeans(predictionCol='cluster', featuresCol='scaledFeatureVector', maxIter=40, tol=0.00001).s
etSeed(randint(1, 10)).setK(k)
    pipeline = Pipeline(stages=[protoTypeEncoder, serviceEncoder, flagEncoder, assembler, scaler, kmeans])
    pipelineModel = pipeline.fit(df)
    kmeansModel = pipelineModel.stages[-1]
    return kmeansModel.summary.trainingCost
```

Figure 14 – clusteringScore3 function using oneHotPipeline function

This code is not referenced in the book, so an explanation is in order. The oneHotPipeline function returns a Tuple consisting of a Pipeline itself (an encoder) and a string (the new column name). The new set of columns are defined by removing the old columns and adding the new ones (except for the label since it's not needed). Then each of the returned encoders are added as stages to the main pipeline to be executed. The results for this function were the following:

K	Training cost
60	193038129.56
90	83494955.24
120	16603088.04
150	11176356.12
180	8635457.26
210	6378831.55
240	5522056.79
270	6422007.64

Figure 15 – Clustering results using the clusteringScore3 function

Now all input features are being used and there's more confidence in using a particular value of k for anomaly detection, like 150 or 180.

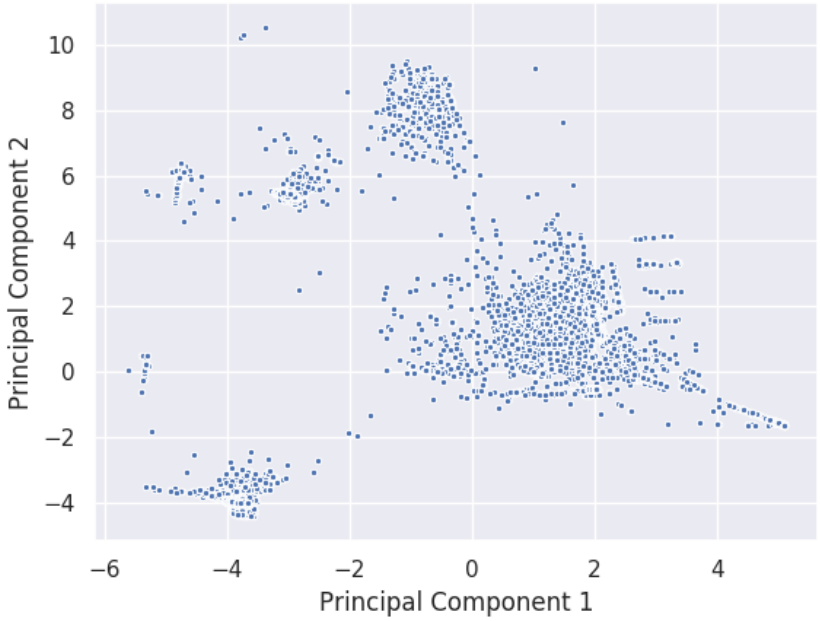


Figure 16 – PCA plotting with categorical features

Using all features, the plot structure is even richer, as it's now possible to see some sub-regions close to the main region on the right.

Entropy as Cluster Score

Points labelled with the same value, for example, “smurf” should end up in one or two clusters. A good clustering should not group together points with many different labels. This is the principle of homogeneity. A weighted average of entropy can be used as a cluster score because it will validate the quality of the produced clusters. Low entropy means more homogeneity in the clusters.

```
def clusteringScore4(df, k) -> float:
    pipelineModel = fitPipeline4(df, k)
    clusterLabel = pipelineModel.transform(df).select('cluster', 'label') # already as int and string
    clusterLabels = clusterLabel.rdd.groupByKey()
    weightedClusterEntropySum = 0
    clusterLabelsCol = clusterLabels.collect()
    for group in clusterLabelsCol:
        labels = list(group[1]) # we want the iterable (labels) of the tuple
        labelCounts = list(collections.Counter(labels).values())
        weightedClusterEntropySum += len(labels) * entropy(labelCounts)
    return weightedClusterEntropySum / df.count()
```

Figure 17 – clusteringScore4 function using entropy

The function fitPipeline4 simply returns a pipeline identical to the ones defined previously for the other clustering functions. All the label values are grouped by each cluster, where cluster is the key and the labels are the values for each cluster.
For each cluster, the entropy will be calculated as the product between the number of labels and the entropy of label counts. The labelCounts variable is a list containing the number of occurrences per label for a given cluster.
The entropy function is defined below:

```
def entropy(counts: iter) -> float:
    values = list(filter(lambda x: x > 0, counts))
    n = sum(map(lambda x: float(x), values))
    result = 0
    for v in values:
        p = v / n
        result += -p * math.log(p)
    return result
```

Figure 18 – Entropy function

The entropy function, in this case, takes the labelCounts as an argument and applies the formula:

$$S = -\sum_{i} P_i \log P_i$$

Where S is represented by the result variable and P_i is the probability derived by the fraction of v_i and n (the sum of label counts). The results for the clusteringScore4 function were the following:

K	Training cost
60	0.1034
90	0.0234
120	0.0188
150	0.0146
180	0.0197
210	0.0102
240	0.0103
270	0.0069

Figure 19 - Clustering results using the clusteringScore4 function

From these results it's possible to conclude that k=150 is a good value since its score is actually lower than when k=180. The value 210 is also a good value. The value 270 is also a possibility, albeit, computationally expensive.

Final clustering

Now, clustering can be done with a proper value of k, in this case, 210. Below are some of the clusters generated using this value.

```
+-------+----------+-------+
|cluster|     label|  count|
+-------+----------+-------+
|      0|   normal.|     51|
|      0|    smurf.|2807614|
|      1|  neptune.| 357450|
|      1|portsweep.|     62|
|      2|  neptune.|   1032|
|      2|portsweep.|      7|
|      2|    satan.|      3|
|      3|   normal.|     13|
|      3|portsweep.|     44|
|      4|  neptune.|   1038|
|      4|portsweep.|     13|
|      4|    satan.|      3|
|      5|  neptune.|    846|
|      5|   normal.|    449|
|      5|portsweep.|      7|
|      5|    satan.|      3|
|      6|   normal.|      7|
|      6|      pod.|      5|
|      7|  ipsweep.|     13|
|      7|  neptune.|   1042|
+-------+----------+-------+
only showing top 20 rows
```

Figure 20 – Sample of clusters and respective label count for k=210

Since a proper clustering algorithm is now implemented, it's now possible to detect anomalies in the data. A connection or data point is a potential anomaly if its distance to the nearest centroid exceeds a given threshold. For this anomaly detector, the threshold was defined as the 100th-farthest data point. The value of the distance, in this case, is ≈4623.41.
In addition, here is a potential anomaly example:

```
Row(duration=9, protocol_type='tcp', service='telnet', flag='SF', src_bytes=307, dst_bytes=2374, land=0, wrong_fragment=0, urge
nt=1, hot=0, num_failed_logins=0, logged_in=1, num_compromised=0, root_shell=1, su_attempted=0, num_root=1, num_file_creations=
3, num_shells=1, num_access_files=0, num_outbound_cmds=0, is_host_login=0, is_guest_login=0, count=1, srv_count=1, serror_rate=
0.0, srv_serror_rate=0.0, rerror_rate=0.0, srv_rerror_rate=0.0, same_srv_rate=1.0, diff_srv_rate=0.0, srv_diff_host_rate=0.0, d
st_host_count=69, dst_host_srv_count=4, dst_host_same_srv_rate=0.03, dst_host_diff_srv_rate=0.04, dst_host_same_src_port_rate=
0.01, dst_host_srv_diff_host_rate=0.75, dst_host_serror_rate=0.0, dst_host_srv_serror_rate=0.0, dst_host_rerror_rate=0.0, dst_h
ost_srv_rerror_rate=0.0, label='normal.', protocol_type_indexed=1.0, protocol_type_vec=SparseVector(2, {1: 1.0}), service_index
ed=11.0, service_vec=SparseVector(69, {11: 1.0}), flag_indexed=0.0, flag_vec=SparseVector(10, {0: 1.0}), featureVector=SparseVe
ctor(119, {0: 2374.0, 1: 4.0, 2: 1.0, 4: 0.75, 16: 1.0, 76: 0.03, 77: 1.0, 79: 1.0, 80: 9.0, 82: 1.0, 83: 1.0, 88: 1.0, 89: 3.
0, 90: 69.0, 93: 1.0, 94: 307.0, 98: 1.0, 99: 0.01, 101: 0.04, 108: 1.0, 118: 1.0}), scaledFeatureVector=SparseVector(119, {0:
0.0037, 1: 0.0378, 2: 2.8522, 4: 18.1775, 16: 33.857, 76: 0.073, 77: 138.5985, 79: 0.2539, 80: 0.0124, 82: 114.448, 83: 0.0047,
88: 2.5687, 89: 24.1574, 90: 1.0778, 93: 0.0041, 94: 0.0003, 98: 2.0583, 99: 0.0208, 101: 0.3685, 108: 2.3564, 118: 121.1072}),
cluster=77)
```

Figure 21 – Potential anomaly example

Some of the results found might not be obvious to a data scientist on why a given result is a potential anomaly. However, in this case, the connection has a value of 69 for the dst_host_count feature (69 connections to different hosts), even though it's labelled as a normal connection.

It was also possible to verify that, out of the 4.9 million connections present in the dataset, only 100 are potential anomalies, roughly 0.00002%.
Obviously, given a lower threshold value the number of potential anomalies might be greater.

Where to go from here

Silhouette Coefficient

The main approach for the project consisted of using different values of k, that is, different numbers of clusters in multiple experiences to figure out the cost associated with it. The lower the cost (the sum of squared distances of points to their nearest centre) for a value k, the better the value. Typically, a greater value of k produces lower cost. However, it's more computationally expensive so the goal was to pick a value of k where the cost started decreasing less, or not decreasing at all.
This was easier to notice with an elbow graph. However, this approach is not completely reliable.

As such, a more sophisticated approach used for picking a value for k is the silhouette coefficient. It contrasts the average distance to elements in the same cluster with the average distance to elements in other clusters. The silhouette value ranges from -1 to 1 where a value closer to 1 indicates that the objects are well clustered and a value closer to -1 might indicate poor clustering configuration or presence of outliers, meaning, excessive number of clusters or not enough clusters.

Using the ClusteringEvaluator class, which uses the Silhouette coefficient by default, and the previously defined pipeline (fitPipeline4) consisting of encoders, the assembler, the scaler and the KMeans algorithm, the following results were obtained:

K	Silhouette score
60	0.5575
90	0.7152
120	0.9438
150	0.9312
180	0.7690
210	0.8682
240	0.7459
270	0.7696

Figure 22 – K value and respective Silhouette score

In this case, the best value of k seems to be 120 since its respective silhouette score, ≈0.9438, is the closest to 1, indicating that data points are well clustered.
Using the 100^{th}-farthest point distance as a threshold as before the result is≈8688.4. The threshold value is greater, which makes sense, considering there are less clusters, which translates to greater distances from the centroids of each cluster to their farthest points.

Using the previously defined anomaly detector with the new threshold value it's possible to find potential anomalies. Like before, 100 potential anomalies were found. In practice, the Silhouette coefficient allowed to pick a more efficient value of k since it provides the same results with less computation.

Gaussian Mixture and Bisecting KMeans

Different models were also applied, namely, Gaussian Mixture and Bisecting KMeans. However, these models are very computationally expensive and only one value of k was used for the experiment, which is 120. The previously defined clustering evaluator was used to measure the performance of both the algorithms
For the Gaussian Mixture model, given it's an extremely slow algorithm, only 1% of the dataset was used, about 50.000 connections which is more than enough data for clustering. The computed silhouette score for this experiment, with k=120, is≈0.5025 which isn't horrible but isn't good either.
For the Bisecting KMeans model, the full dataset was used. Albeit very computationally expensive, it's still much faster than Gaussian Mixture Model. The computed silhouette score for this experiment, with k=120, is≈0.4842 which is even worse than the previous experience.
Reasons for these scores were not explored and it might be due to multiple reasons. The value of k might not have been a good one, the random seed picked for initialization was not appropriate, or there are simply too many dimensions for the algorithms to be effective (especially in Gaussian Mixture). At least the Gaussian Mixture model should have provided better clustering considering it does not assume clusters spherical like KMeans. Moreover, maybe it does in other conditions.

Conclusion

The essence of the problem described in this project, anomaly detection, is perfectly captured by the nature of clustering solutions like KMeans since these ultimately group data points into clusters. Any data point not inside the distance threshold of the cluster may be considered an anomaly.

Picking a good value of k is one of the most important tasks that a data scientist must do when implementing a clustering solution. A value of k that is too high, in the limit, will provide a cluster for each data point while a value of k that is too low will increase the average diameter of the clusters and potentially group unrelated data points in the same cluster. Using cluster evaluating methods like the elbow graph, entropy score and the silhouette coefficient allow to get a better sense of how many clusters should be used.

In problems like this, feature normalization also plays a big role since that step alone can completely change the output of a clustering pipeline, as seen previously. It decreases the cost (distance between data points) and prevents some features from drowning out other features.

With the aid of visualization tools and PCA it's also possible to see the distribution of the data, the most relevant features and how data points are clustered.

It is also possible to verify that it's relatively easy and relatively fast to cluster millions of data points using a vanilla KMeans algorithm implementation, at least when comparing with other algorithms like Gaussian Mixture or Bisecting KMeans.

The dataset, despite having millions of network connections didn't seem to have many anomalies, as only 100 (about 0.00002%) were considered potential anomalies given the provided implementation.

Amazon AWS provided a great experience thanks to its advantages when compared with a typical virtual machine. Better computational resources (more CPU cores, multiple instances, etc) as well as the ability to work remotely from any computer accelerated the whole process overall.

With all this being said, the dataset can still be processed in many ways. While it was almost exclusively processed using KMeans with Euclidean distance by default, another distance measure like Manhattan-distance, implemented by k-medians, could be used, even though it's not supported by Spark. Spark only supports Euclidian and Cosine distances, the latter one being used for textual problems.

Given more time and computational resources, deeper investigation could be made involving Gaussian Mixture model and other algorithms.

References

Datasets and related information:
http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

Spark and Python documentation:
https://spark.apache.org/docs/latest/api/python/index.html
https://spark.apache.org/docs/latest/sql-programming-guide.html
https://docs.python.org/3/

AWS documentation:
https://aws.amazon.com/pt/premiumsupport/knowledge-center/emr-session-not-found-http-request-error/

Visualization documentation:
https://seaborn.pydata.org/index.html
https://matplotlib.org/
https://towardsdatascience.com/cluster-analysis-create-visualize-and-interpret-customer-segments-474e55d00ebb

Theoretical basis:
https://en.wikipedia.org/wiki/Principal_component_analysis
https://en.wikipedia.org/wiki/Mixture_model#Gaussian_mixture_model
https://en.wikipedia.org/wiki/K-means_clustering
https://en.wikipedia.org/wiki/Silhouette_(clustering)
https://en.wikipedia.org/wiki/Entropy
https://towardsdatascience.com/gaussian-mixture-models-explained-6986aaf5a95