Module-5

COMPILED BY,

DR. PRAKASH KALINGRAO AITHAL

Anomaly Detection using K-Means Clustering

These techniques do not learn to predict a target value, because none is available. They can, however, learn structure in data and find groupings of similar inputs, or learn what types of input are likely to occur and what types are not.

Anomaly Detection using K-Means Clustering

Anomaly detection is often used to find fraud, detect network attacks, or discover problems in servers or other sensor-equipped machinery. In these cases, it's important to be able to find new types of anomalies that have never been seen before—new forms of fraud, intrusions, and failure modes for servers. Unsupervised learning techniques are useful in these cases because they can learn what input data normally looks like and therefore detect when new data is unlike past data. Such new data is not necessarily attacks or fraud; it is simply unusual and therefore worth further investigation.

K-Means Clustering

The inherent problem of anomaly detection is, as its name implies, that of finding unusual things. If we already knew what “anomalous” meant for a dataset, we could easily detect anomalies in the data with supervised learning. An algorithm would receive inputs labeled “normal” and “anomaly” and learn to distinguish the two. However, the nature of anomalies is that they are unknown unknowns. Put another way, an anomaly that has been observed and understood is no longer an anomaly.

K-Means Clustering

Clustering is the best-known type of unsupervised learning. Clustering algorithms try to find natural groupings in data. Data points that are like one another but unlike others are likely to represent a meaningful grouping, so clustering algorithms try to put such data into the same cluster.

K-Means Clustering

K-means clustering may be the most widely used clustering algorithm. It attempts to detect *k* clusters in a dataset, where *k* is given by the data scientist. *k* is a hyperparameter of the model, and the right value will depend on the dataset.

K-Means Clustering

K-means requires a notion of distance between data points. It is common to use simple Euclidean distance to measure distance between data points with K-means, and as it happens, this is one of two distance functions supported by Spark MLlib as of this writing, the other one being Cosine. The Euclidean distance is defined for data points whose features are all numeric. "Like" points are those whose intervening distance is small.

K-Means Clustering

To K-means, a cluster is simply a point: the center of all the points that make up the cluster. These are, in fact, just feature vectors containing all numeric features and can be called vectors. However, it may be more intuitive to think of them as points here, because they are treated as points in a Euclidean space.

K-Means Clustering

This center is called the cluster *centroid* and is the arithmetic mean of the points— hence the name K-*means*. To start, the algorithm picks some data points as the initial cluster centroids. Then each data point is assigned to the nearest centroid. Then for each cluster, a new cluster centroid is computed as the mean of the data points just assigned to that cluster. This process is repeated.

Identifying Anomalous Network Traffic

Cyberattacks are increasingly visible in the news. Some attacks attempt to flood a computer with network traffic to crowd out legitimate traffic. But in other cases, attacks attempt to exploit flaws in networking software to gain unauthorized access to a computer. While it's quite obvious when a computer is being bombarded with traffic, detecting an exploit can be like searching for a needle in an incredibly large haystack of network requests.

Identifying Anomalous Network Traffic

Some exploit behaviors follow known patterns. For example, accessing every port on a machine in rapid succession is not something any normal software program should ever need to do. However, it is a typical first step for an attacker looking for services running on the computer that may be exploitable.

Identifying Anomalous Network Traffic

If you were to count the number of distinct ports accessed by a remote host in a short time, you would have a feature that probably predicts a port-scanning attack quite well. A handful is probably normal; hundreds indicate an attack. The same goes for detecting other types of attacks from other features of network connections—number of bytes sent and received, TCP errors, and so forth.

Identifying Anomalous Network Traffic

But what about those unknown unknowns? The biggest threat may be the one that has never yet been detected and classified. Part of detecting potential network intrusions is detecting anomalies. These are connections that aren't known to be attacks but do not resemble connections that have been observed in the past.

Identifying Anomalous Network Traffic

Here, unsupervised learning techniques like K-means can be used to detect anomalous network connections. K-means can cluster connections based on statistics about each of them. The resulting clusters themselves aren't interesting per se, but they collectively define types of connections that are like past connections. Anything not close to a cluster could be anomalous. Clusters are interesting insofar as they define regions of normal connections; everything else is unusual and potentially anomalous.

Choosing K

A clustering could be considered good if each data point were near its closest centroid, where “near” is defined by the Euclidean distance. This is a simple, common way to evaluate the quality of a clustering, by the mean of these distances over all points, or sometimes, the mean of the distances squared. In fact, KMeansModel offers a ClusteringEvaluator method that computes the sum of squared distances and can easily be used to compute the mean squared distance.

Choosing K

```
from pyspark.sql import DataFrame
from pyspark.ml.evaluation import ClusteringEvaluator

from random import randint

def clustering_score(input_data, k):
    input_numeric_only = input_data.drop("protocol_type", "service", "flag")
    assembler = VectorAssembler().setInputCols(input_numeric_only.columns[:-1]).\
                                  setOutputCol("featureVector")
    kmeans = KMeans().setSeed(randint(100,100000)).setK(k).\
                      setPredictionCol("cluster").\
                      setFeaturesCol("featureVector")
    pipeline = Pipeline().setStages([assembler, kmeans])
    pipeline_model = pipeline.fit(input_numeric_only)

    evaluator = ClusteringEvaluator(predictionCol='cluster',
                                    featuresCol="featureVector")
    predictions = pipeline_model.transform(numeric_only)
    score = evaluator.evaluate(predictions)
    return score

for k in list(range(20,100, 20)):
    print(clustering_score(numeric_only, k)) ❶
```

Choosing K

However, this much is obvious. As more clusters are added, it should always be possible to put data points closer to the nearest centroid. In fact, if *k* is chosen to equal the number of data points, the average distance will be 0 because every point will be its own cluster of one!

Choosing K

Worse, in the preceding results, the distance for *k*=80 is higher than for *k*=60. This shouldn't happen because a higher *k* always permits at least as good a clustering as a lower *k*. The problem is that K-means is not necessarily able to find the optimal clustering for a given *k*. Its iterative process can converge from a random starting point to a local minimum, which may be good but is not optimal.

Choosing K

The random starting set of clusters chosen for *k*=80 perhaps led to a particularly suboptimal clustering, or it may have stopped early before it reached its local optimum. We can improve it by running the iteration longer. The algorithm has a threshold via setTol that controls the minimum amount of cluster centroid movement considered significant; lower values mean the K-means algorithm will let the centroids continue to move longer. Increasing the maximum number of iterations with setMaxIter also prevents it from potentially stopping too early at the cost of possibly more computation.

Choosing K

```
def clustering_score_1(input_data, k):
    input_numeric_only = input_data.drop("protocol_type", "service", "flag")
    assembler = VectorAssembler().\
                setInputCols(input_numeric_only.columns[:-1]).\
                setOutputCol("featureVector")
    kmeans = KMeans().setSeed(randint(100,100000)).setK(k).setMaxIter(40).\ ❶
      setTol(1.0e-5).\ ❷
      setPredictionCol("cluster").setFeaturesCol("featureVector")
    pipeline = Pipeline().setStages([assembler, kmeans])
    pipeline_model = pipeline.fit(input_numeric_only)
    #
    evaluator = ClusteringEvaluator(predictionCol='cluster',
                                    featuresCol="featureVector")
    predictions = pipeline_model.transform(numeric_only)
    score = evaluator.evaluate(predictions)
    #
    return score


for k in list(range(20,101, 20)):
    print(k, clustering_score_1(numeric_only, k))
```

Choosing K

1→Increase from default 20.

2→Decrease from default 1.0e-4.

Feature Normalization

We can normalize each feature by converting it to a standard score. This means subtracting the mean of the feature's values from each value and dividing by the standard deviation, as shown in the standard score equation:

Feature Normalization

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

Feature Normalization

In fact, subtracting means has no effect on the clustering because the subtraction effectively shifts all the data points by the same amount in the same direction. This does not affect interpoint Euclidean distances. MLlib provides StandardScaler, a component that can perform this kind of standardization and be easily added to the clustering pipeline. We can run the same test with normalized data on a higher range of *k*:

Feature Normalization

```
from pyspark.ml.feature import StandardScaler

def clustering_score_2(input_data, k):
    input_numeric_only = input_data.drop("protocol_type", "service", "flag")
    assembler = VectorAssembler().\
                setInputCols(input_numeric_only.columns[:-1]).\
                setOutputCol("featureVector")
    scaler = StandardScaler().setInputCol("featureVector").\
                              setOutputCol("scaledFeatureVector").\
                              setWithStd(True).setWithMean(False)
    kmeans = KMeans().setSeed(randint(100,100000)).\
                      setK(k).setMaxIter(40).\
                      setTol(1.0e-5).setPredictionCol("cluster").\
                      setFeaturesCol("scaledFeatureVector")
    pipeline = Pipeline().setStages([assembler, scaler, kmeans])
    pipeline_model = pipeline.fit(input_numeric_only)
    #
    evaluator = ClusteringEvaluator(predictionCol='cluster',
                                    featuresCol="scaledFeatureVector")
    predictions = pipeline_model.transform(numeric_only)
    score = evaluator.evaluate(predictions)
    #
    return score

for k in list(range(60, 271, 30)):
    print(k, clustering_score_2(numeric_only, k))
```

Feature Normalization

This has helped put dimensions on more equal footing, and the absolute distances between points (and thus the cost) is much smaller in absolute terms. However, the above output doesn't yet provide an obvious value of *k* beyond which increasing it does little to improve the cost.

One hot encoder

```
from pyspark.ml.feature import OneHotEncoder, StringIndexer

def one_hot_pipeline(input_col):
    indexer = StringIndexer().setInputCol(input_col).\
                              setOutputCol(input_col + "-_indexed")
    encoder = OneHotEncoder().setInputCol(input_col + "indexed").\
                              setOutputCol(input_col + "_vec")
    pipeline = Pipeline().setStages([indexer, encoder])
    return pipeline, input_col + "_vec" ❶
```

❶ Return pipeline and name of output vector column

Anomaly detector

```
import numpy as np

from pyspark.spark.ml.linalg import Vector, Vectors
from pyspark.sql.functions import udf

k_means_model = pipeline_model.stages[-1]
centroids = k_means_model.clusterCenters

clustered = pipeline_model.transform(data)

def dist_func(cluster, vec):
    return float(np.linalg.norm(centroids[cluster] - vec))
dist = udf(dist_func)

threshold = clustered.select("cluster", "scaledFeatureVector").\
    withColumn("dist_value",
        dist(col("cluster"), col("scaledFeatureVector"))).\
    orderBy(col("dist_value").desc()).take(100)
```

References

Sandya Ryza, Uri Laserson, Sean Owen and Josh Wills, Advanced Analytics with Spark (2e), O'Reilly Media Inc, 2017