KMeans Clustering

- Import the necessary libraries.
- Create a Spark session, which is the entry point for all Spark applications. This allows users to perform operations over distributed datasets.

In [ ]:
```
import pyspark
import os
import sys
from pyspark import SparkContext
os.environ['PYSPARK_PYTHON'] = sys.executable
os.environ['PYSPARK_DRIVER_PYTHON'] = sys.executable
from pyspark.sql import SparkSession

spark = SparkSession.builder.config("spark.driver.memory", "16g").appName('chapter_
```

- The data is read using the read function of spark. This allows data to be read from various sources
- The option function allows users to set the read options:
  - `the schema is inferred automatically from the input data`
  - `there is no header in the input data`
- the csv function allows the csv file to be read and converted to a dataframe from the specified file path

The column names of the dataframe are specified. The toDF function is used to return a new Dataframe consisting of the specified column names

In [ ]:
```
data_without_header = spark.read.option("inferSchema", True).\
    option("header", False).\
    csv("data/kddcup.data_10_percent_corrected")
column_names = [ "duration", "protocol_type", "service", "flag",
"src_bytes", "dst_bytes", "land", "wrong_fragment", "urgent",
"hot", "num_failed_logins", "logged_in", "num_compromised",
"root_shell", "su_attempted", "num_root", "num_file_creations",
"num_shells", "num_access_files", "num_outbound_cmds",
"is_host_login", "is_guest_login", "count", "srv_count",
"serror_rate", "srv_serror_rate", "rerror_rate", "srv_rerror_rate",
"same_srv_rate", "diff_srv_rate", "srv_diff_host_rate",
"dst_host_count", "dst_host_srv_count",
"dst_host_same_srv_rate", "dst_host_diff_srv_rate",
"dst_host_same_src_port_rate", "dst_host_srv_diff_host_rate",
"dst_host_serror_rate", "dst_host_srv_serror_rate",
"dst_host_rerror_rate", "dst_host_srv_rerror_rate",
"label"]
data = data_without_header.toDF(*column_names)
```

The data is grouped by its labels, and the count of records for each distinct value of labels is aggregated and stored in descending order

In [ ]:
```
from pyspark.sql.functions import col

data.select("label").groupBy("label").count().\
    orderBy(col("count").desc()).show(25)
```

- Nonnumeric columns are dropped from the table
- A pipeline is created, consisting of:
  - An assembler which consolidates all the input feature columns into a single feature vector
    - A KMeans module, which is a clustering algorithm that groups data into clusters based on their similarity
- These stages are carried out in sequence on the input data
- The pipeline is fit on the training data to train the KMeans model
- This trained model is extarcted from the second stage of the trained pipeline
- The clusterCenters() function retreives the cluster centers learned by the model during training. These are the coordinates of the centroids of each cluster. These cluster centers are displayed. The number of centers indicates the number of distinct clusters that data was grouped into

In [ ]:
```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.clustering import KMeans, KMeansModel
from pyspark.ml import Pipeline

numeric_only = data.drop("protocol_type", "service", "flag").cache()

assembler = VectorAssembler().setInputCols(numeric_only.columns[:-1]).\
    setOutputCol("featureVector")
kmeans = KMeans().setPredictionCol("cluster").setFeaturesCol("featureVector")
pipeline = Pipeline().setStages([assembler, kmeans])

pipeline_model = pipeline.fit(numeric_only)
kmeans_model = pipeline_model.stages[1]

from pprint import pprint
pprint(kmeans_model.clusterCenters())
```

The transform function is applied on the data to classify it into the distinct clusters of the model.

The count of labels falling in each cluster is calculated (by counting records in distinct groups of cluster and label combinations), and stored in ascending order of clusters and descending order of counts within each cluster.

In [ ]:
```
with_cluster = pipeline_model.transform(numeric_only)
with_cluster.select("cluster", "label").groupBy("cluster", "label").count().\
    orderBy(col("cluster"), col("count").desc()).show(25)
```

To choose the best value for the number of clusters in the mdoel, a clustering score function is defined, which:

- drops nonnumeric columns from the input data
- creates a pipeline consisting of:
  - a vector assembler
  - a kmeans module with the parameter K (number of clusters) set by the parameter k passed to the function
- fits the pipeline modle on the data

- retrieves the trained kmeans model and extracts its training cast from the attribute trainingCost of the summary attribute of the model. The trainingCost indicates the mean of the sqaured distance of each point to its cluster center. A lower cost indicates better clustering of the data

This function is performed for different values of k and the training cost is displayed to compare the performance of the models

In [ ]:
```
from pyspark.sql import DataFrame
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]).set
    kmeans = KMeans().setSeed(randint(100,100000)).setK(k).setPredictionCol("cluste
    pipeline = Pipeline().setStages([assembler, kmeans])
    pipeline_model = pipeline.fit(input_numeric_only)
    kmeans_model = pipeline_model.stages[-1]
    training_cost = kmeans_model.summary.trainingCost
    return training_cost

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

A similar function is defined in this case to compare the performance of models with different number of clusters. However, in this case, for the Kmeans models defined, two additional attribiutes are set:

- maxIter: This is the maximum number of iterations the clustering is run for, to prevent it from stopping too early, before the optimal cluster centers are reached
- Tol: this is the minimum amount of cluster movement (change in centroid position) that is considered significant. A low value means that the algorithm will let the centroids move for longer

Once again, the function is performed for different values of k and the training cost is displayed to compare the performance of the models

In [ ]:
```
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).\
        setTol(1.0e-5).\
        setPredictionCol("cluster").setFeaturesCol("featureVector")
    pipeline = Pipeline().setStages([assembler, kmeans])
    pipeline_model = pipeline.fit(input_numeric_only)
    kmeans_model = pipeline_model.stages[-1]
    training_cost = kmeans_model.summary.trainingCost
    return training_cost

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

A similar function is once again defined, but in this case, the pipeline includes a StandardScaler transformer between the assembler and the kmeans estimator. The StandardScaler is used to normalize the input features so they all fall within a given range,

putting each feature on a more equal footing and do not skew the results. It does so by subtracting the mean of the feature's value from each value and dividing it by its standard deviation.

The function is performed for different values of k and the training cost is displayed to compare the performance of the models

In [ ]:
```
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)
    kmeans_model = pipeline_model.stages[-1]
    training_cost = kmeans_model.summary.trainingCost
    return training_cost

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

To deal with categorical variables, they need to be one hot encoded. For this a function is defined which:

- instantiates a string indexer for the input column. This builds an index for each unique value in the given column, to represent its category
- instantiates a one-hot-encoder for the indexed column. This converts the index of the record into a one-hot-encoded representation (to prevent any ranking of the different categories)
- creates and returns a pipeline consisting of the indexer and one-hot-encoder

In [ ]:
```
from pyspark.ml.feature import OneHotEncoder, StringIndexer
def one_hot_pipeline(input_col):
    indexer = StringIndexer().setInputCol(input_col).setOutputCol(input_col + "_inc
    encoder = OneHotEncoder().setInputCol(input_col + "_indexed").setOutputCol(inpu
    pipeline = Pipeline().setStages([indexer, encoder])
    return pipeline, input_col + "_vec"
```

A function to compare clustering performance is once again defined. In this case, the nonnumeric features are not dropped, but instead are indexed and converted into one-hot-encoded categorical variables. For this, one hot encoding pipelines are created fro each feature using the previously defined function and added to beginning stages of the overall pipeline. The one-hot-encoded columns are included in the features which are converted to a feature vector by the assembler.

The function is performed for different values of k and the training cost is displayed to compare the performance of the models

In [ ]:
```
def clustering_score_3(input_data, k):
    proto_type_pipeline, proto_type_vec_col = one_hot_pipeline("protocol_type")
    service_pipeline, service_vec_col = one_hot_pipeline("service")
    flag_pipeline, flag_vec_col = one_hot_pipeline("flag")
    assemble_cols = set(input_data.columns) - \
        {"label", "protocol_type", "service", "flag"} | \
        {proto_type_vec_col, service_vec_col, flag_vec_col}
    assembler = VectorAssembler().setInputCols(list(assemble_cols)).setOutputCol("f
    scaler = StandardScaler().setInputCol("featureVector").setOutputCol("scaledFeat
    kmeans = KMeans().setSeed(randint(100,100000)).setK(k).setMaxIter(40).setTol(1.
        setFeaturesCol("scaledFeatureVector")
    pipeline = Pipeline().setStages([proto_type_pipeline, service_pipeline, flag_pi
    pipeline_model = pipeline.fit(input_data)
    kmeans_model = pipeline_model.stages[-1]
    training_cost = kmeans_model.summary.trainingCost
    return training_cost
for k in list(range(60, 271, 30)):
print(k, clustering_score_3(data, k))
```

To further check the performance of the clustering, a measure called entropy is used.

Entropy is the measure of impurity. It is a function of the relative proportions of labels in a group and it produces a low value when the proportions are skewed towards few or one label, that is, have a large number of a single label and are relatively homogeneous (consisting of single type of labels). A good clustering algorithm should also create clusters that are homogeneous with a single label occurring frequently, and so should create clusters with low values of entropy.

A function is created that calculates the entropy for a given set as SUM(-p*log(p)), where p is the probability of each group in the set

In [ ]:
```
from math import log
def entropy(counts):
    values = [c for c in counts if (c > 0)]
    n = sum(values)
    p = [v/n for v in values]
    return sum([-1*(p_v) * log(p_v) for p_v in p])
```

- The transform function is used to process the data through the pipeline and produce the cluster labels for each reecord.
- These are grouped and aggregated to store the count of each label within each cluster
- A window is defined, which groups the data by the cluster. A window function is used to perform statistical operations on groups of data in the dataframe, in this case, the separate clusters
- The probability of each label within each cluster is calculated by finding the proportion of count of the given label to the total number of points in the cluster. The .over() function specifies this needs to be done separately for each window defined in the data
- The dataframe is grouped by cluster and aggregations are performed on each group to calculate the entropy using the probability of labels calculated previously, and the size of the cluster as the sum of the counts of each label in the cluster.
- Finally, the entropy of each cluster is weighted by multiplying it with the cluster size.
- The average weighted entropy is calculated by summing all of the weighted cluster entropies and dividing it by the total number of data points

In [ ]:
```
from pyspark.sql import functions as fun
from pyspark.sql import Window
cluster_label = pipeline_model.\
        transform(data).\
        select("cluster", "label")
df = cluster_label.\
        groupBy("cluster", "label").\
        count().orderBy("cluster")
w = Window.partitionBy("cluster")
p_col = df['count'] / fun.sum(df['count']).over(w)
with_p_col = df.withColumn("p_col", p_col)
result = with_p_col.groupBy("cluster").\
        agg((-fun.sum(col("p_col") * fun.log2(col("p_col"))))
        .alias("entropy"),fun.sum(col("count")).alias("cluster_size"))
result = result.withColumn('weightedClusterEntropy',fun.col('entropy') * fun.col('c
weighted_cluster_entropy_avg = result.\
        agg(fun.sum(
        col('weightedClusterEntropy'))).\
        collect()
weighted_cluster_entropy_avg[0][0]/data.count()
```

- A function is defined which creates a pipeline, including:
  - the pipelines to one-hot-encode all the nonnumeric columns
  - the vector assembler
  - the scaler
  - the KMeans model, with K set as the input parameter k.
- Finally, a function is defined on the value of k, for which the pipeline is created and fit on the data. The data is transformed to include cluster labels and the entropy of each cluster and weighted average cluster entropy is calculated and displayed (in a similar manner to the previous cell) to compare the performance of the clustering algorithm for each value of k

In [ ]:
```
def fit_pipeline_4(data, k):
    (proto_type_pipeline, proto_type_vec_col) = one_hot_pipeline("protocol_type")
    (service_pipeline, service_vec_col) = one_hot_pipeline("service")
    (flag_pipeline, flag_vec_col) = one_hot_pipeline("flag")
```

```
    assemble_cols = set(data.columns) - {"label", "protocol_type", "service", "flag
                    {proto_type_vec_col, service_vec_col, flag_vec_col}
    assembler = VectorAssembler(inputCols=list(assemble_cols), outputCol="featureVe
    scaler = StandardScaler(inputCol="featureVector", outputCol="scaledFeatureVecto
    kmeans = KMeans(seed=randint(100, 100000), k=k, predictionCol="cluster", /
                    featuresCol="scaledFeatureVector", maxIter=40, tol=1.0e-5)
    pipeline = Pipeline(stages=[proto_type_pipeline, service_pipeline, flag_pipelir
    return pipeline.fit(data)

def clustering_score_4(input_data, k):
    pipeline_model = fit_pipeline_4(input_data, k)
    cluster_label = pipeline_model.transform(input_data).select("cluster", "label")
    df = cluster_label.groupBy("cluster", "label").count().orderBy("cluster")
    w = Window.partitionBy("cluster")
    p_col = df['count'] / fun.sum(df['count']).over(w)
    with_p_col = df.withColumn("p_col", p_col)
    result = with_p_col.groupBy("cluster").agg(-fun.sum(col("p_col") * fun.log2(col
    fun.sum(col("count")).alias("cluster_size"))
    result = result.withColumn('weightedClusterEntropy', col('entropy') * col('clus
    weighted_cluster_entropy_avg = result.agg(fun.sum(col('weightedClusterEntropy')
    return weighted_cluster_entropy_avg[0][0] / input_data.count()
```

Using the weighted average cluster entropy scores, the best value of k is chosen as 180. A pipeline is created with a KMeans model of k=180.

The data is transformed to by processing through the pipeline to include cluster labels. This is aggregated to store the count of each label within each cluster (by grouping over unique combinations of clusters and labels) and displayed.

The results show highly homogeneous clusters, with most clusters containing only one type of label, or a high frequency of a single label.

In [ ]:
```
pipeline_model = fit_pipeline_4(data, 180)
count_by_cluster_label = pipeline_model.transform(data).\
select("cluster", "label").\
groupBy("cluster", "label").\
count().orderBy("cluster", "label")
count_by_cluster_label.show()
```