# clustering

April 7, 2024

```
import pyspark
import os
import sys
```

Imports the `pyspark` library and the `os` and `sys` modules.

```
os.environ['PYSPARK_PYTHON'] = sys.executable
os.environ['PYSPARK_DRIVER_PYTHON'] = sys.executable
```

Sets the `PYSPARK_PYTHON` and `PYSPARK_DRIVER_PYTHON` environment variables to the current Python executable path, ensuring that PySpark uses the same Python environment.

```
from pyspark.sql import SparkSession
```

Import SparkSession from the `pyspark.sql` module.

[ ]:
```
import pyspark
import os
import sys

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_5').getOrCre
```

Creates a new spark session with 16GB memory and the name `chapter_5`.

[ ]:
```
spark = SparkSession.builder.config("spark.driver.memory", "16g").
↪appName('chapter_5').getOrCreate()
```

The `column_names` list contains the names of the columns in the CSV file. These names are assigned to the columns of the DataFrame using the `toDF()` method.

The resulting DataFrame, `data`, contains the data from the CSV file with the specified column names. This allows for easier access and manipulation of the data using the assigned column names instead of the default column names (`_c0`, `_c1`, etc.).

[ ]:
```
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)
```

`from pyspark.sql.functions import col`

Imports the `col` function from the `pyspark.sql.functions` module.

`data.select("label")` Selects the "label" column from the `data` DataFrame.
`.groupBy("label")` Groups the selected data by the "label" column.
`.count()` Counts the number of rows for each group.
`.orderBy(col("count").desc())` Orders the grouped and counted data in descending order based on the "count" column.

[ ]:
```
from pyspark.sql.functions import col
data.select("label").groupBy("label").count().orderBy(col("count").desc()).
  ↪show(25)
```

First, it creates a new DataFrame `numeric_only` by dropping the non-numeric columns from the original `data` DataFrame and caching it for better performance.

Next, it sets up a `VectorAssembler` to combine the input columns (all columns except the last one) into a single feature vector column named "featureVector".

It then initializes a `KMeans` object, specifying the output column for the predicted cluster ("cluster") and the input column for the feature vector ("featureVector").

The `VectorAssembler` and `KMeans` are combined into a `Pipeline`, which is then fit on the `numeric_only` DataFrame to create a `pipeline_model`.

The trained `KMeans` model is extracted from the `pipeline_model` and its cluster centers are printed using `pprint()`.

[ ]:
```
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())

with_cluster = pipeline_model.transform(numeric_only)
with_cluster.select("cluster", "label").groupBy("cluster", "label").count().
  ↪orderBy(col("cluster"), col("count").desc()).show(25)
```

```
from pyspark.sql import DataFrame
from random import randint
```

The `clustering_score` function takes an `input_data` DataFrame and a number `k` as input. It drops the columns "protocol_type", "service", and "flag" from the input DataFrame to create a

new DataFrame `input_numeric_only` with only numeric columns.

```
input_numeric_only = input_data.drop("protocol_type", "service", "flag")
```

It then creates a `VectorAssembler` to combine the numeric columns (excluding the last column) into a single feature vector column named "featureVector".

```
assembler = VectorAssembler().setInputCols(input_numeric_only.columns[:-1]).setOutputCol("featu
```

A `KMeans` model is initialized with a random seed, the specified `k` value, and the "featureVector" column as input. The predicted cluster for each data point will be stored in the "cluster" column.

```
kmeans = KMeans().setSeed(randint(100,100000)).setK(k).setPredictionCol("cluster").setFeatures(
```

The `VectorAssembler` and `KMeans` are combined into a `Pipeline`, which is then fitted on the `input_numeric_only` DataFrame to create a `pipeline_model`.

```
pipeline = Pipeline().setStages([assembler, kmeans])
pipeline_model = pipeline.fit(input_numeric_only)
```

The trained `KMeans` model is extracted from the `pipeline_model`, and its `trainingCost` (sum of squared distances of points to their nearest center) is returned as the clustering score.

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

Finally, the script iterates over `k` values from 20 to 100 (exclusive) in steps of 20 and prints the clustering score for each `k` using the `numeric_only` DataFrame.

"'python

[ ]:
```python
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]).
 ↪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)
    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))
```

```
input_numeric_only = input_data.drop("protocol_type", "service", "flag")
```

Drops the columns "protocol_type", "service", and "flag" from the `input_data` DataFrame and assigns the result to `input_numeric_only`.

`assembler = VectorAssembler().setInputCols(input_numeric_only.columns[:-1]).setOutputCol("featu`

Creates a `VectorAssembler` object that combines all columns of `input_numeric_only` except the last one into a single vector column named "featureVector".

`kmeans = KMeans().setSeed(randint(100,100000)).setK(k).setMaxIter(40).setTol(1.0e-5).setPredict`

Initializes a `KMeans` object with a random seed, sets the number of clusters to `k`, maximum iterations to 40, convergence tolerance to 1.0e-5, prediction column name to "cluster", and features column name to "featureVector".

`pipeline = Pipeline().setStages([assembler, kmeans])`

Creates a `Pipeline` object that consists of the `VectorAssembler` and `KMeans` stages.

`pipeline_model = pipeline.fit(input_numeric_only)`

Fits the pipeline to the `input_numeric_only` DataFrame and assigns the resulting model to `pipeline_model`.

`kmeans_model = pipeline_model.stages[-1]`

Extracts the trained `KMeans` model from the last stage of the pipeline.

`training_cost = kmeans_model.summary.trainingCost`

Retrieves the training cost from the `KMeans` model summary and assigns it to `training_cost`.

[ ]:
```
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))
```

First, it removes the non-numeric columns "protocol_type", "service", and "flag" from the input data using `drop()`.

Then, it creates a pipeline of three stages:
1. `VectorAssembler` combines the remaining columns into a single feature vector column named "featureVector".
2. `StandardScaler` standardizes the feature vectors by scaling them to unit variance, outputting the result as "scaledFeatureVector".
3. `KMeans` performs the clustering with the specified number of clusters `k`, maximum iterations, and convergence tolerance, using the scaled feature vectors. The cluster assignments are stored in the "cluster" column.

The pipeline is fit to the numeric-only data, and the resulting `KMeansModel` is extracted from the pipeline.

The training cost (sum of squared distances between points and their nearest cluster center) is obtained from the model summary and returned.

[ ]:
```
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))
```

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

Imports the `OneHotEncoder` and `StringIndexer` classes from the `pyspark.ml.feature` module.

`def one_hot_pipeline(input_col):`

Defines a function `one_hot_pipeline` that takes an `input_col` parameter representing the input column name.

`indexer = StringIndexer().setInputCol(input_col).setOutputCol(input_col + "_indexed")`

Creates a `StringIndexer` object that converts the string values in `input_col` to numeric indices. The output column is named `input_col + "_indexed"`.

`encoder = OneHotEncoder().setInputCol(input_col + "_indexed"). setOutputCol(input_col + "_vec"`

Creates a `OneHotEncoder` object that performs one-hot encoding on the indexed column. The output column is named `input_col + "_vec"`.

`pipeline = Pipeline().setStages([indexer, encoder])`

Creates a `Pipeline` object and sets the stages to the `indexer` and `encoder` objects.

`return pipeline, input_col + "_vec"`

[ ]:
```
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"
```

The pipeline consists of the following stages:
- `proto_type_pipeline`, `service_pipeline`, and `flag_pipeline`: One-hot encode the `protocol_type`, `service`, and `flag` columns respectively using the `one_hot_pipeline` function.
- `assembler`: Assembles the feature columns (excluding `label`, `protocol_type`, `service`, and `flag`) and the one-hot encoded columns into a single feature vector column named `featureVector`.
- `scaler`: Scales the `featureVector` using `StandardScaler` without centering (mean=0) and outputs the scaled vector as `scaledFeatureVector`.
- `kmeans`: Initializes the k-means model with a random seed, the specified number of clusters `k`, maximum iterations, tolerance, prediction column name, and the input features column.

The pipeline is then fit on the `input_data` to obtain the `pipeline_model`. The k-means model is extracted from the pipeline stages, and its training cost is returned.

[ ]:
```
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("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([proto_type_pipeline,␣
 ↪service_pipeline,flag_pipeline, assembler, scaler, kmeans])
    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))
```

`from math import log`

Imports the `log` function from the built-in `math` module.

```
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])
```

Defines a function named `entropy` that takes a list `counts` as input.
Creates a new list `values` containing only the positive elements from `counts`.
Calculates the sum of all elements in `values` and assigns it to the variable `n`.
Creates a new list `p` containing the normalized values of `values` by dividing each element by `n`.

[ ]:
```
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])
```

1. `cluster_label = pipeline_model.transform(data).select("cluster", "label")`: Applies a pipeline model to the `data` DataFrame, selects the `"cluster"` and `"label"` columns, and assigns the result to `cluster_label`.
2. `df = cluster_label.groupBy("cluster", "label").count().orderBy("cluster")`: Groups the `cluster_label` DataFrame by `"cluster"` and `"label"`, counts the occurrences of each combination, orders the result by `"cluster"`, and assigns it to `df`.
3. `w = Window.partitionBy("cluster")`: Creates a window partitioned by the `"cluster"` column.
4. `p_col = df['count'] / fun.sum(df['count']).over(w)`: Calculates the proportion of each label within each cluster by dividing the count of each label by the total count of labels in the cluster using the window function.
5. `with_p_col = df.withColumn("p_col", p_col)`: Adds the calculated proportion column `"p_col"` to the `df` DataFrame.
6. 
```
   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"))
```
   Groups the `with_p_col` DataFrame by `"cluster"`, calculates the entropy and cluster size for each cluster using aggregate functions, and assigns the result to `result`.
7. `result = result.withColumn('weightedClusterEntropy',fun.col('entropy') * fun.col('cluster_size'))`: Adds a new column `'weightedClusterEntropy'` to the `result` DataFrame by multiplying the entropy by the cluster size.
8. “‘python
   weighted_cluster_entropy_avg = result.agg(fun.sum(
   col(‘weightedClusterEntropy’))).collect()

[ ]:
```
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('cluster_size'))
weighted_cluster_entropy_avg = result.agg(fun.sum(
col('weightedClusterEntropy'))).collect()
weighted_cluster_entropy_avg[0][0]/data.count()
```

`fit_pipeline_4` takes `data` and `k` as parameters and performs the following steps:
1. Creates three pipelines (`proto_type_pipeline`, `service_pipeline`, `flag_pipeline`) using the `one_hot_pipeline` function to one-hot encode the categorical columns "protocol_type", "service", and "flag".
2. Assembles the feature columns (excluding "label", "protocol_type", "service", "flag" and including the one-hot encoded columns) into a single vector column "featureVector" using `VectorAssembler`.
3. Scales the "featureVector" using `StandardScaler` to create "scaledFeatureVector".
4. Applies `KMeans` clustering with random seed, specified `k`, and other parameters on the "scaledFeatureVector".
5. Combines the stages into a `Pipeline` and fits the pipeline on the input `data`.

`clustering_score_4` takes `input_data` and `k` as parameters and calculates the weighted cluster entropy average:
1. Fits the pipeline using `fit_pipeline_4` on the `input_data` with the specified `k`.
2. Transforms the `input_data` using the fitted pipeline model and selects the "cluster" and "label" columns.
3. Groups the data by "cluster" and "label", counts the occurrences, and orders by "cluster".
4. Calculates the proportion of each label within each cluster using a window function and adds it as a new column "p_col".
5. Groups the data by "cluster" and calculates the entropy and cluster size for each cluster.
6. Computes the weighted cluster entropy by multiplying the entropy by the cluster size.
7. Calculates the weighted cluster entropy average by summing the weighted cluster entropies and dividing by the total number of data points.

[ ]:
```
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="featureVector")
    scaler = StandardScaler(inputCol="featureVector",␣
↪outputCol="scaledFeatureVector", withStd=True, withMean=False)
    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_pipeline, assembler, scaler, kmeans])
    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("p_col"))).alias("entropy"),
    fun.sum(col("count")). alias("cluster_size"))
    result = result.withColumn('weightedClusterEntropy', col('entropy') *␣
↪col('cluster_size'))
    weighted_cluster_entropy_avg = result.agg(fun.␣
↪sum(col('weightedClusterEntropy'))).collect()
    return weighted_cluster_entropy_avg[0][0] / input_data.count()
```

Next, the code transforms the `data` using the `pipeline_model` and selects the `"cluster"` and `"label"` columns. It then groups the transformed data by `"cluster"` and `"label"` and counts the number of occurrences for each combination. The result is ordered by `"cluster"` and `"label"`.

[ ]:
```
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(
```