Lecture_10_sparkML_cloud_code

September 30, 2024

SparkML on the cloud

1 1. Running the NSLKDD Example on the Cloud

As usual, we start with the data ingestion and data engineering preprocessing.

[55]:
```
import pyspark
from pyspark.sql import SparkSession, SQLContext
from pyspark.ml import Pipeline,Transformer
from pyspark.ml.feature import␣
 ↪Imputer,StandardScaler,StringIndexer,OneHotEncoder, VectorAssembler

from pyspark.sql.functions import *
from pyspark.sql.types import *
import numpy as np

col_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","class","difficulty"]

nominal_cols = ['protocol_type','service','flag']
binary_cols = ['land', 'logged_in', 'root_shell', 'su_attempted',␣
 ↪'is_host_login',
'is_guest_login']
continuous_cols = ['duration' ,'src_bytes', 'dst_bytes', 'wrong_fragment'␣
 ↪,'urgent', 'hot',
'num_failed_logins', 'num_compromised', 'num_root' ,'num_file_creations',
'num_shells', 'num_access_files', 'num_outbound_cmds', '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']

class OutcomeCreater(Transformer): # this defines a transformer that creates␣
 ↪the outcome column

    def __init__(self):
        super().__init__()

    def _transform(self, dataset):
        label_to_binary = udf(lambda name: 0.0 if name == 'normal' else 1.0)
        output_df = dataset.withColumn('outcome',␣
 ↪label_to_binary(col('class'))).drop("class")
        output_df = output_df.withColumn('outcome', col('outcome').
 ↪cast(DoubleType()))
        output_df = output_df.drop('difficulty')
        return output_df

class FeatureTypeCaster(Transformer): # this transformer will cast the columns␣
 ↪as appropriate types
    def __init__(self):
        super().__init__()

    def _transform(self, dataset):
        output_df = dataset
        for col_name in binary_cols + continuous_cols:
            output_df = output_df.withColumn(col_name,col(col_name).
 ↪cast(DoubleType()))

        return output_df
class ColumnDropper(Transformer): # this transformer drops unnecessary columns
    def __init__(self, columns_to_drop = None):
        super().__init__()
        self.columns_to_drop=columns_to_drop
    def _transform(self, dataset):
        output_df = dataset
        for col_name in self.columns_to_drop:
            output_df = output_df.drop(col_name)
        return output_df

def get_preprocess_pipeline():
    # Stage where columns are casted as appropriate types
    stage_typecaster = FeatureTypeCaster()
```

```
  # Stage where nominal columns are transformed to index columns using␣
↪StringIndexer
  nominal_id_cols = [x+"_index" for x in nominal_cols]
  nominal_onehot_cols = [x+"_encoded" for x in nominal_cols]
  stage_nominal_indexer = StringIndexer(inputCols = nominal_cols, outputCols␣
↪= nominal_id_cols )

  # Stage where the index columns are further transformed using OneHotEncoder
  stage_nominal_onehot_encoder = OneHotEncoder(inputCols=nominal_id_cols,␣
↪outputCols=nominal_onehot_cols)

  # Stage where all relevant features are assembled into a vector (and␣
↪dropping a few)
  feature_cols = continuous_cols+binary_cols+nominal_onehot_cols
  corelated_cols_to_remove =␣
↪["dst_host_serror_rate","srv_serror_rate","dst_host_srv_serror_rate",
                   ␣
↪"srv_rerror_rate","dst_host_rerror_rate","dst_host_srv_rerror_rate"]
  for col_name in corelated_cols_to_remove:
      feature_cols.remove(col_name)
  stage_vector_assembler = VectorAssembler(inputCols=feature_cols,␣
↪outputCol="vectorized_features")

  # Stage where we scale the columns
  stage_scaler = StandardScaler(inputCol= 'vectorized_features', outputCol=␣
↪'features')


  # Stage for creating the outcome column representing whether there is␣
↪attack
  stage_outcome = OutcomeCreater()

  # Removing all unnecessary columbs, only keeping the 'features' and␣
↪'outcome' columns
  stage_column_dropper = ColumnDropper(columns_to_drop =␣
↪nominal_cols+nominal_id_cols+
      nominal_onehot_cols+ binary_cols + continuous_cols +␣
↪['vectorized_features'])
  # Connect the columns into a pipeline
  pipeline =␣
↪Pipeline(stages=[stage_typecaster,stage_nominal_indexer,stage_nominal_onehot_encoder,
      stage_vector_assembler,stage_scaler,stage_outcome,stage_column_dropper])
  return pipeline
```

[56]:

```
# Put the training and test data in the cluster. Uncomment this code if you␣
 ↪haven't run this before.

# !pip install wget

# !python -m wget https://www.andrew.cmu.edu/user/mfarag/14813/KDDTest+.txt
# !hadoop fs -put KDDTest+.txt /

# !python -m wget https://www.andrew.cmu.edu/user/mfarag/14813/KDDTrain+.txt
# !hadoop fs -put KDDTrain+.txt /
```

1.1 Set up spark to run in cluster mode

We are running the notebook on a DataProc cluster, which is designed to run spark on the cluster with multiple worker nodes.

To run spark on cluster, when creating the SparkSession, set the master as "yarn". (yarn is a type of cluster management tool that DataProc is using). In this cluster mode, the master node will serve as the "driver" that runs this notebook. However, each time we have a dataframe operation (e.g. the fit when training an ML model), spark will split the operation into stages, and each stages into tasks, and distribute the tasks to the worker nodes who will run the tasks in parallel.

In comparison, if the master is set as "local", then all the computation will happen locally on the master node (worker node will not be utilized).

[57]:
```
# If you want to run the spark in cluster in the dataproc cluster, set the␣
 ↪master as yarn
# If you want to run locally, set the master as local

spark = SparkSession.builder \
    .master("yarn") \
    .appName("SparkML-yarn") \
    .getOrCreate()

nslkdd_raw = spark.read.csv('/KDDTrain+.txt',header=False).toDF(*col_names)
nslkdd_test_raw = spark.read.csv('/KDDTest+.txt',header=False).toDF(*col_names)

preprocess_pipeline = get_preprocess_pipeline()
preprocess_pipeline_model = preprocess_pipeline.fit(nslkdd_raw)

nslkdd_df = preprocess_pipeline_model.transform(nslkdd_raw)
nslkdd_df_test = preprocess_pipeline_model.transform(nslkdd_test_raw)


nslkdd_df.cache()
nslkdd_df_test.cache()
```

```
24/09/28 21:02:42 INFO SparkEnv: Registering MapOutputTracker
24/09/28 21:02:42 INFO SparkEnv: Registering BlockManagerMaster
24/09/28 21:02:42 INFO SparkEnv: Registering BlockManagerMasterHeartbeat
24/09/28 21:02:42 INFO SparkEnv: Registering OutputCommitCoordinator
```

[57]:
```
DataFrame[features: vector, outcome: double]
```

In cluster node, each dataframe is stored in a distributed manner across worker nodes. We can check out how many partition a dataframe has and how many rows each partition is allocated.

[58]:
```
# Checking how many partitions the dataframe is split into
num_partitions = nslkdd_df.rdd.getNumPartitions()
print(f"Number of partitions: {num_partitions}")


# # Uncomment to check how many rows each partition has
# def show_partitions(index, iterator):
#     yield index, list(iterator)


# # Count how many rows each partition has
# partitions_data = nslkdd_df.rdd.mapPartitionsWithIndex(show_partitions).
 ↪collect()
# for partition, data in partitions_data:
#     print(f"Partition {partition}: contains {len(data)} rows")
```

```
Number of partitions: 2
```

[ ]:
```
from pyspark.ml.classification import RandomForestClassifier

rf = RandomForestClassifier(featuresCol = 'features', labelCol =␣
 ↪'outcome',numTrees=500)
rf_model = rf.fit(nslkdd_df)
```

```
24/09/28 21:04:06 WARN DAGScheduler: Broadcasting large task binary with size
1233.3 KiB
24/09/28 21:04:29 WARN DAGScheduler: Broadcasting large task binary with size
2.1 MiB
[Stage 23:============================>                              (1 + 1) / 2]
```

[ ]:
```
from pyspark.ml.evaluation import BinaryClassificationEvaluator

evaluator = BinaryClassificationEvaluator(rawPredictionCol='rawPrediction',
    labelCol='outcome', metricName='areaUnderROC')


rf_prediction_train = rf_model.transform(nslkdd_df)
rf_prediction_test = rf_model.transform(nslkdd_df_test)
```

```
rf_accuracy_train = (rf_prediction_train.filter(rf_prediction_train.outcome ==␣
 ↪rf_prediction_train.prediction)
    .count()/ float(rf_prediction_train.count()))
rf_accuracy_test = (rf_prediction_test.filter(rf_prediction_test.outcome ==␣
 ↪rf_prediction_test.prediction)
    .count() / float(rf_prediction_test.count()))

rf_auc = evaluator.evaluate(rf_prediction_test)

print(f"Train accuracy = {np.round(rf_accuracy_train*100,2)}%, test accuracy =␣
 ↪{np.round(rf_accuracy_test*100,2)}%, AUC = {np.round(rf_auc,2)}")
```

[ ]:
```
spark.stop()
```

2 2. Basics about Spark RDD

In Apache Spark, while DataFrame provide a higher-level abstraction for data, they are fundamentally built on top of RDDs (Resilient Distributed Datasets). Every DataFrame operation eventually gets translated into RDD transformations and actions.

[49]:
```
spark = SparkSession.builder \
    .master("yarn") \
    .appName("SparkML-RDD-basics") \
    .getOrCreate()
sc = spark.sparkContext

# create an RDD and store it distirbutedly.
rdd = sc.parallelize([1, 2, 3, 4])

# check out haw many partitions the RDD ahs
num_partitions = rdd.getNumPartitions()
print(f"Number of partitions: {num_partitions}")
```

```
24/09/28 20:58:42 INFO SparkEnv: Registering MapOutputTracker
24/09/28 20:58:42 INFO SparkEnv: Registering BlockManagerMaster
24/09/28 20:58:42 INFO SparkEnv: Registering BlockManagerMasterHeartbeat
24/09/28 20:58:42 INFO SparkEnv: Registering OutputCommitCoordinator

Number of partitions: 2
```

2.1 RDD transformations

When you perform operations on a DataFrame (like select, filter, groupBy, etc.) or conduct machine learning tasks (fit for a ML model), Spark builds a logical execution plan. This logical plan is then optimized into a physical execution plan, which is composed of RDD transformations and actions.

RDD transformations and actions are the most basic operations on RDDs and importantly, RDD transformations/actions can be implemented in a distributed manner across nodes.

There are dozens of types of transformations/actions, and the most basic ones are **map, reduce, and filter**.

2.2 Map

Map transformation applies a function to each element of RDD and returns a new RDD.

As each RDD is partitioned and stored on different nodes, the Map is implemented in parallel on different partitions.

[50]:
```
# suppose let's apply a square function "lambda x: x * x" to each element of␣
 ↪the RDD.
squared_rdd = rdd.map(lambda x: x * x)
```

[51]:
```
# Here collect means collect all the partitions into the master node
# (which is needed as we want to print the content of the RDD).
# Normally this shouldn't be done if the dataset is very large, in which case␣
 ↪collect() would crash the driver node.
print(squared_rdd.collect())
```

```
[Stage 0:>                                                          (0 + 2) / 2]
```

```
[1, 4, 9, 16]
```

2.3 Reduce

The reduce() action aggregates elements of an RDD using a binary function (a function that takes two arguments and output one argument). It is typically used to combine all elements into a single result (such as computing a sum, product, or another aggregated value).

Similar to Map, Reduce is implemented in parallel on different partitions.

[52]:
```
# let's calculate the product of all elements in the RDD
product_result = rdd.reduce(lambda a, b: a * b)
print(product_result)
```

```
24
```

2.4 Filter

The filter() transformation returns a new RDD that contains only the elements that satisfy a given condition. It is used to remove elements that don't meet the criteria.

[53]:
```
# Let's filter and keep all the EVEN numbers in the RDD
filtered_rdd = rdd.filter(lambda x: x % 2 == 0)
print(filtered_rdd.collect())
```

```
[2, 4]
```

[54]:
```
spark.stop()
```

[ ]:
```

```