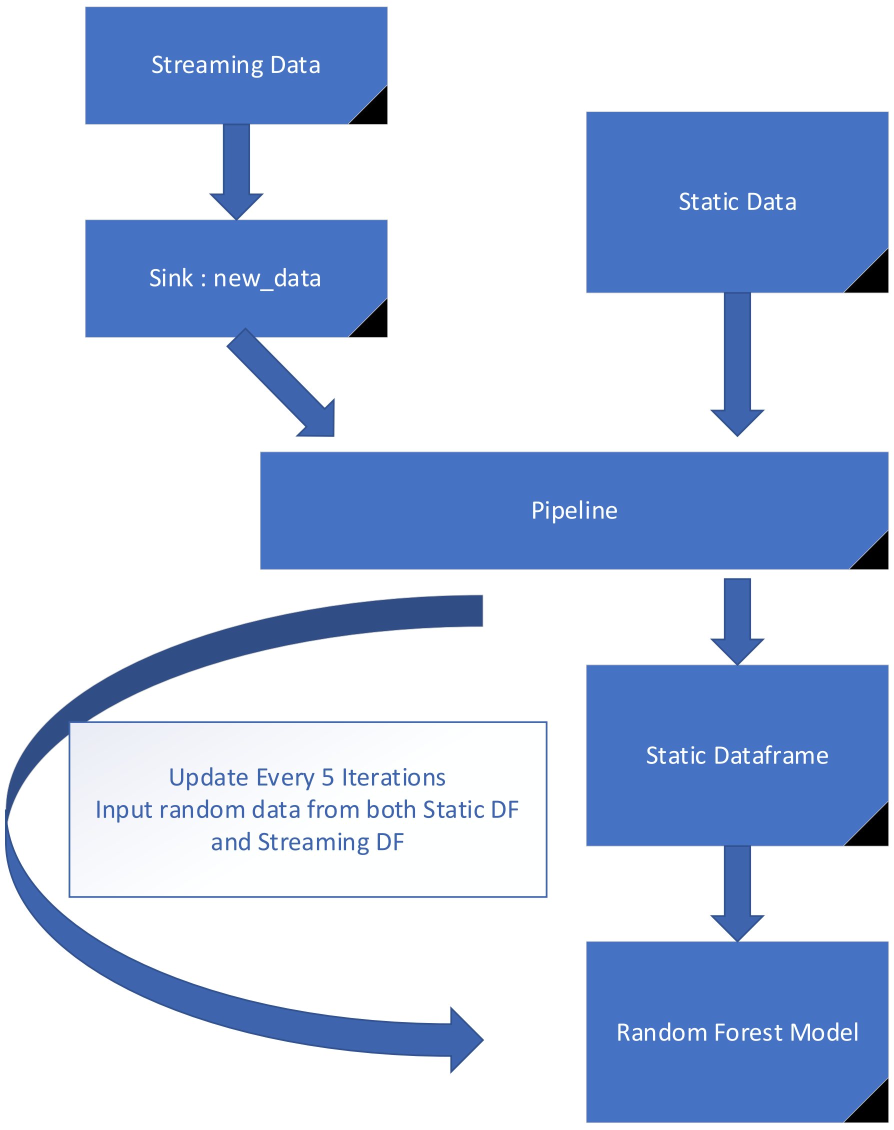
Project Part 2 Question 3

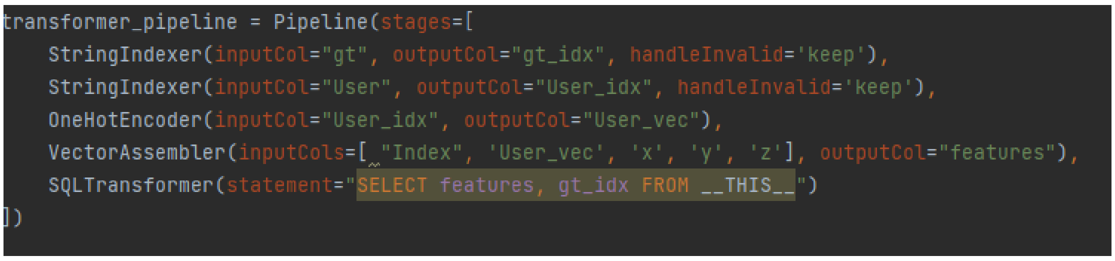
We chose to use the same model from question 2, apart from its number of trees – but we changed some of the processes along the way. Mainly, we changed the data manipulation on the dataframes.

Our General Model Scheme:



For Debugging - Initialy we didn't know how difficult debugging will be. After we found out, we imported the time library and printed the time it took to reach every check point along the code.

Pre Proccessing – We used a similar pipeline as in question 2 but modified it.



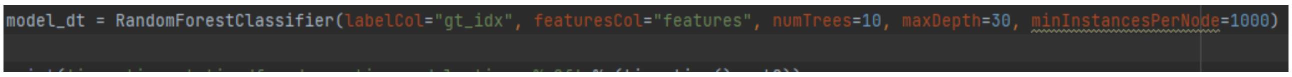
```
transformer_pipeline = Pipeline(stages=[
    StringIndexer(inputCol="gt", outputCol="gt_idx", handleInvalid='keep'),
    StringIndexer(inputCol="User", outputCol="User_idx", handleInvalid='keep'),
    OneHotEncoder(inputCol="User_idx", outputCol="User_vec"),
    VectorAssembler(inputCols=[ "Index", 'User_vec', 'x', 'y', 'z'], outputCol="features"),
    SQLTransformer(statement="SELECT features, gt_idx FROM __THIS__")
])
```

first, we kept the string indexers for the User and gt columns, but removed the Device one.

That is because there were too many devices as new data kept coming and the Vector assembler was creating sparse and big data frames, which both slowed down the model and took a lot of space.

Next, for space complexity, we used the SQLTransformer to select only the "features" column – because it was the only column needed for the Random Forest model.

The Model- We then sampled 50% of the data, and we chose our model to be:

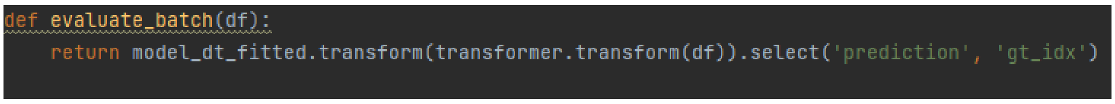


```
model_dt = RandomForestClassifier(labelCol="gt_idx", featuresCol="features", numTrees=10, maxDepth=30, minInstancesPerNode=1000)
```

The difference is the number of trees is bigger – the accuracy dropped when there weren't enough trees. And so, we chose to make the model bigger.

We then made it too big for the server to handle, but found out that 10 trees with 1000 rows per leaf minimum, was a good enough space complexity for the server.

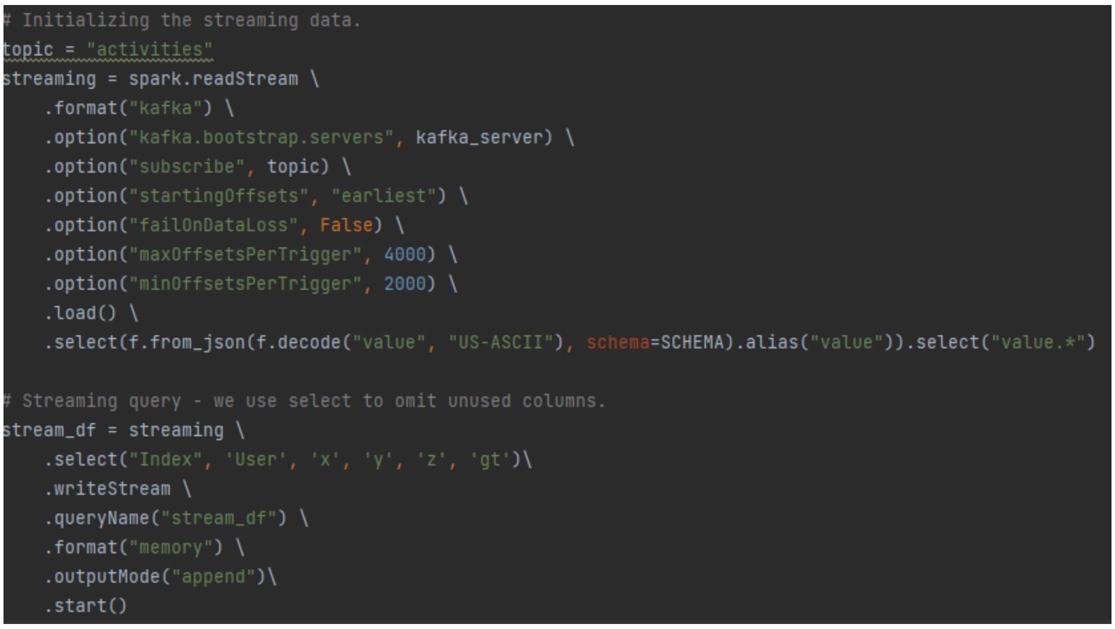
Utilities – we created a function to make the code cleaner, and utilize the space in the Stack rather then only the heap:



```
def evaluate_batch(df):
    return model_dt_fitted.transform(transformer.transform(df)).select('prediction', 'gt_idx')
```

The function take incoming streaming data, transforms it, and the trained model returns its predictions.

Streaming -



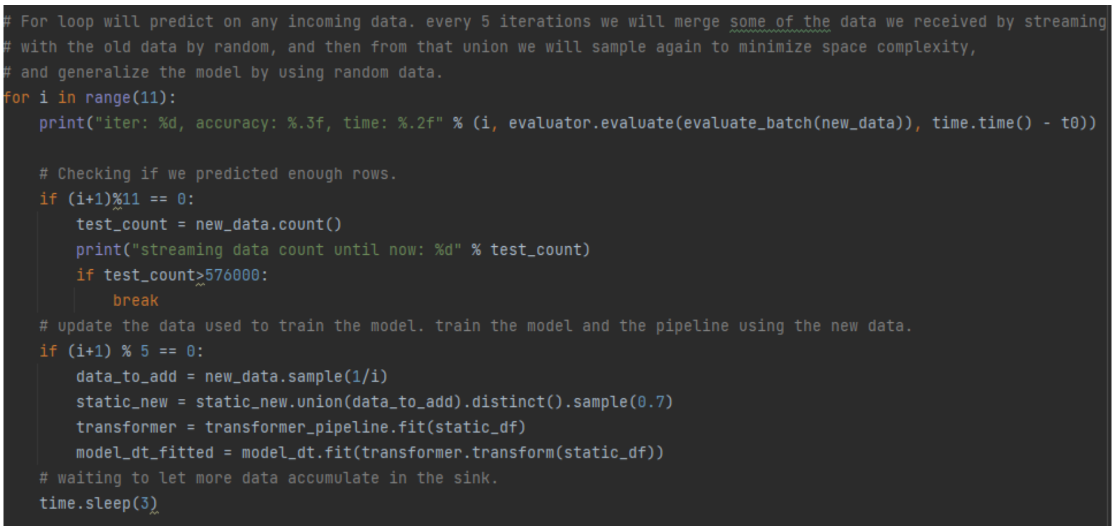
```python
# Initializing the streaming data.
topic = "activities"
streaming = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", kafka_server) \
    .option("subscribe", topic) \
    .option("startingOffsets", "earliest") \
    .option("failOnDataLoss", False) \
    .option("maxOffsetsPerTrigger", 4000) \
    .option("minOffsetsPerTrigger", 2000) \
    .load() \
    .select(f.from_json(f.decode("value", "US-ASCII"), schema=SCHEMA).alias("value")).select("value.*")

# Streaming query - we use select to omit unused columns.
stream_df = streaming \
    .select("Index", 'User', 'x', 'y', 'z', 'gt')\
    .writeStream \
    .queryName("stream_df") \
    .format("memory") \
    .outputMode("append")\
    .start()
```

It took a lot of tries to find out the best size of the batchs, and then we used the select function on the query in order to save space while the data is not yet written into the memory (instead of waiting for the pipeline to do it).

We then created a while loop that waits until the stream_df dataframe will be full enough (we chose 10,000 records).

The process itself –



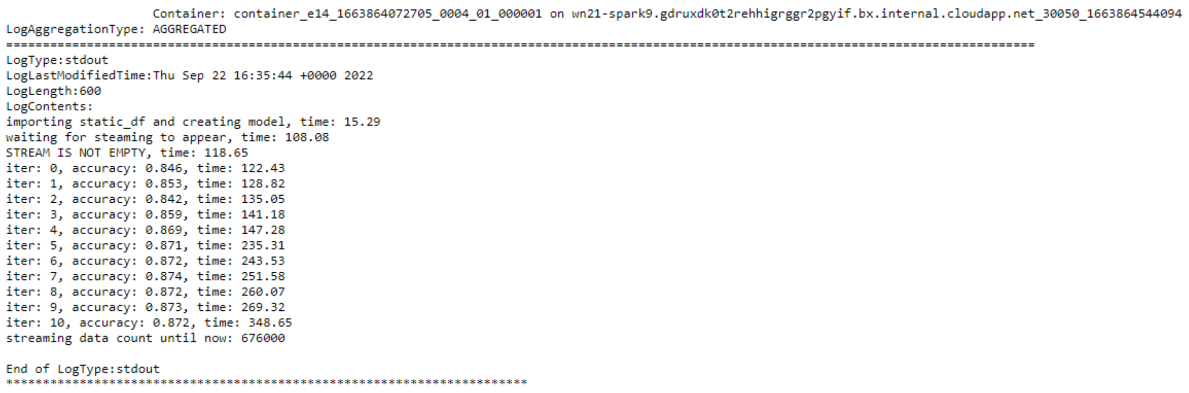
```python
# For loop will predict on any incoming data. every 5 iterations we will merge some of the data we received by streaming
# with the old data by random, and then from that union we will sample again to minimize space complexity,
# and generalize the model by using random data.
for i in range(11):
    print("iter: %d, accuracy: %.3f, time: %.2f" % (i, evaluator.evaluate(evaluate_batch(new_data)), time.time() - t0))

    # Checking if we predicted enough rows.
    if (i+1)%11 == 0:
        test_count = new_data.count()
        print("streaming data count until now: %d" % test_count)
        if test_count>576000:
            break
    # update the data used to train the model. train the model and the pipeline using the new data.
    if (i+1) % 5 == 0:
        data_to_add = new_data.sample(1/i)
        static_new = static_new.union(data_to_add).distinct().sample(0.7)
        transformer = transformer_pipeline.fit(static_df)
        model_dt_fitted = model_dt.fit(transformer.transform(static_df))
    # waiting to let more data accumulate in the sink.
    time.sleep(3)
```

The remarks pretty much explain the process and our intention in every block of code. Basically it shows the same model showed in the first page.

Notice that the IF statement , which is the update of the model, is located below the prediction part, to avoid predicting a batch the model has already trained on.

For conclusion our output was:



```
                    Container: container_e14_1663864072705_0004_01_000001 on wn21-spark9.gdruxdk0t2rehhigrggr2pgyif.bx.internal.cloudapp.net_30050_1663864544094
LogAggregationType: AGGREGATED
============================================================================================================================================
LogType:stdout
LogLastModifiedTime:Thu Sep 22 16:35:44 +0000 2022
LogLength:600
LogContents:
importing static_df and creating model, time: 15.29
waiting for steaming to appear, time: 108.08
STREAM IS NOT EMPTY, time: 118.65
iter: 0, accuracy: 0.846, time: 122.43
iter: 1, accuracy: 0.853, time: 128.82
iter: 2, accuracy: 0.842, time: 135.05
iter: 3, accuracy: 0.859, time: 141.18
iter: 4, accuracy: 0.869, time: 147.28
iter: 5, accuracy: 0.871, time: 235.31
iter: 6, accuracy: 0.872, time: 243.53
iter: 7, accuracy: 0.874, time: 251.58
iter: 8, accuracy: 0.872, time: 260.07
iter: 9, accuracy: 0.873, time: 269.32
iter: 10, accuracy: 0.872, time: 348.65
streaming data count until now: 676000

End of LogType:stdout
***********************************************************************
```

A lot of times we would have reached around 900,000 records but decided the server was too crowded to continue.

We were able to minimize the time complexity from 12 minutes to just about 6! And by no means was that easy keeping the static dataframe at sufficient size, but it was needed in order to reach this level of accuracy.

And thus the journey ends, thank you for reading!