INFO-H-515 Project 2022–2023, Part II

2022 - 2023

Massa ALSAFADI MAHMALJI

Oussama MIFDAL

Ali Manzer

Sami Abdul Sater

Group-6

Data Exploration

We have discussed the data we are working in the report of phase 1, now to realise this next phase we have made some changes:

We are utilizing a dataset that brings together the counts from all sensors for the same t_gap. Previously, the counts for each sensor were stored in separate tables. By joining these individual tables, we create a new dataset that allows us to work with all sensor counts for a specific time gap in a unified manner.

The decision to use this consolidated dataset was made to streamline the prediction process. Instead of handling each sensor's counts separately, we can now perform predictions for all sensors simultaneously. This approach simplifies the workflow and enables us to leverage the collective information from all sensors to make predictions.

By working with this unified dataset, we can access and analyze the counts of multiple sensors in a cohesive manner, facilitating the prediction task and potentially improving the accuracy of our predictions.

Task 1: Persistence Model

In task 1, the "Stateless" method refers to the approach we used to make predictions. We relied on the current value, y(t), to predict the value at the next time gap, y(t+1), using a "persistence model." The underlying principle of this method is that the value of y(t+1), which represents the count at the next time gap, is assumed to be the same as the current count, y(t).

For each line received by the producer, we follow these steps:

- Increment the value of t_gap by 1, indicating the next time gap.
- Return the same count values for all 18 sensors, as we assume that the count remains constant from one time gap to the next.

Essentially, we are using the current count as a prediction for the count at the next time gap, assuming no significant change or variation in the count values.

```
--------------------------------------------
Time: 2023-05-13 23:15:54
--------------------------------------------
[98, '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0',
'0.0']
[99, '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0',
'0.0']
[100, '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0',
'0.0']
[101, '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0',
'0.0']
[102, '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0',
'0.0']
[103, '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0',
'0.0']
[104, '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0',
'0.0']
[105, '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0', '0.0',
'0.0']
```

Task 2: Weighted Persistence Model

In task 2, we employ the "stateful" method, which means our predictions for the count value at the next time gap are not solely reliant on the current t_gap value. Instead, we take into consideration previous count values along with the current t_gap. The number of previous count values we consider is determined by the 'State' variable, and its quantity depends on the value of V specified in the code.

To make predictions using the stateful method, we follow these steps:

- Collect the relevant previous count values based on the value of V (where V can be 2, 3, or 4).

- Calculate the average of these previous count values.
- Update the state by incorporating the latest count value.
- Set the value of the count at the next time gap to be equal to the calculated average.

By incorporating historical count values, our goal is to capture underlying patterns or trends within the data. This approach allows us to make more accurate predictions for the count at the next time gap, taking advantage of the information contained in previous counts.

```
------------------------------------------
Time: 2023-05-14 16:13:52
------------------------------------------
('state', ((array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0.]), array([0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       5.]), array([0., 0., 3., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       2.])), 4))
```

```
V=  4
Time gap : 578.0  | Sensor 1 : 0.0 | Sensor 2 : 0.0 | Sensor 3 : 1.0 | Sensor 4 : 0.25 | Sensor 5 : 0.25 | Sensor 6 : 0.0 | Se
nsor 7 : 0.0 | Sensor 8 : 0.5 | Sensor 9 : 0.5 | Sensor 10 : 0.0 | Sensor 11 : 0.0 | Sensor 12 : 0.0 | Sensor 13 : 0.0 | Senso
r 14 : 0.25 | Sensor 15 : 0.0 | Sensor 16 : 0.0 | Sensor 17 : 0.0 | Sensor 18 : 2.0
V=  4
Time gap : 579.0  | Sensor 1 : 0.0 | Sensor 2 : 0.0 | Sensor 3 : 1.0 | Sensor 4 : 0.0 | Sensor 5 : 0.75 | Sensor 6 : 0.0 | Sen
sor 7 : 0.0 | Sensor 8 : 0.0 | Sensor 9 : 0.75 | Sensor 10 : 0.0 | Sensor 11 : 0.0 | Sensor 12 : 0.0 | Sensor 13 : 0.0 | Senso
r 14 : 0.0 | Sensor 15 : 0.0 | Sensor 16 : 0.0 | Sensor 17 : 0.0 | Sensor 18 : 2.25
V=  4
Time gap : 580.0  | Sensor 1 : 0.0 | Sensor 2 : 0.0 | Sensor 3 : 1.0 | Sensor 4 : 0.0 | Sensor 5 : 0.25 | Sensor 6 : 0.0 | Sen
sor 7 : 0.0 | Sensor 8 : 0.25 | Sensor 9 : 0.25 | Sensor 10 : 0.0 | Sensor 11 : 0.0 | Sensor 12 : 0.0 | Sensor 13 : 0.0 | Sens
or 14 : 0.0 | Sensor 15 : 0.0 | Sensor 16 : 0.0 | Sensor 17 : 0.0 | Sensor 18 : 2.0
V=  4
Time gap : 581.0  | Sensor 1 : 0.0 | Sensor 2 : 0.0 | Sensor 3 : 1.0 | Sensor 4 : 0.0 | Sensor 5 : 0.5 | Sensor 6 : 0.0 | Sens
or 7 : 0.0 | Sensor 8 : 0.0 | Sensor 9 : 0.25 | Sensor 10 : 0.0 | Sensor 11 : 0.0 | Sensor 12 : 0.0 | Sensor 13 : 0.0 | Sensor
14 : 0.0 | Sensor 15 : 0.0 | Sensor 16 : 0.0 | Sensor 17 : 0.0 | Sensor 18 : 2.25
V=  4
Time gap : 582.0  | Sensor 1 : 0.0 | Sensor 2 : 0.0 | Sensor 3 : 1.0 | Sensor 4 : 0.0 | Sensor 5 : 0.5 | Sensor 6 : 0.0 | Sens
or 7 : 0.0 | Sensor 8 : 0.0 | Sensor 9 : 0.0 | Sensor 10 : 0.0 | Sensor 11 : 0.0 | Sensor 12 : 0.0 | Sensor 13 : 0.0 | Sensor
14 : 0.25 | Sensor 15 : 0.0 | Sensor 16 : 0.0 | Sensor 17 : 0.0 | Sensor 18 : 2.25
V=  4
Time gap : 583.0  | Sensor 1 : 0.0 | Sensor 2 : 0.0 | Sensor 3 : 1.25 | Sensor 4 : 0.0 | Sensor 5 : 0.25 | Sensor 6 : 0.0 | Se
nsor 7 : 0.0 | Sensor 8 : 0.0 | Sensor 9 : 0.0 | Sensor 10 : 0.0 | Sensor 11 : 0.0 | Sensor 12 : 0.0 | Sensor 13 : 0.0 | Senso
r 14 : 0.0 | Sensor 15 : 0.0 | Sensor 16 : 0.0 | Sensor 17 : 0.0 | Sensor 18 : 2.25
V=  4
Time gap : 584.0  | Sensor 1 : 0.0 | Sensor 2 : 0.0 | Sensor 3 : 1.0 | Sensor 4 : 0.0 | Sensor 5 : 0.25 | Sensor 6 : 0.0 | Sen
sor 7 : 0.0 | Sensor 8 : 0.0 | Sensor 9 : 0.0 | Sensor 10 : 0.0 | Sensor 11 : 0.0 | Sensor 12 : 0.0 | Sensor 13 : 0.0 | Sensor
14 : 0.0 | Sensor 15 : 0.0 | Sensor 16 : 0.0 | Sensor 17 : 0.0 | Sensor 18 : 2.0
V=  4
Time gap : 585.0  | Sensor 1 : 0.0 | Sensor 2 : 0.0 | Sensor 3 : 1.25 | Sensor 4 : 0.0 | Sensor 5 : 0.25 | Sensor 6 : 0.0 | Se
nsor 7 : 0.0 | Sensor 8 : 0.0 | Sensor 9 : 0.0 | Sensor 10 : 0.0 | Sensor 11 : 0.0 | Sensor 12 : 0.0 | Sensor 13 : 0.0 | Senso
r 14 : 0.5 | Sensor 15 : 0.0 | Sensor 16 : 0.0 | Sensor 17 : 0.0 | Sensor 18 : 1.75
```

Tasks 3 and 4: Description of RLS models

Task 3 asks to implement a linear model using the RLS methods for different values for the **embedding order (EO)**, defined as `n` in the formula `y(t+1) = f(y(t), y(t-1), ... y(t-n))`. The model that we implemented takes as inputs a scalar `y` containing the measure for the current time, and a vector of features `x`. The features that will have an impact on the predicted value depend on the EO. For the simple model asked in task 3, `x` will only contain the measures of the last `n` time gaps for the desired sensor. For this model, the number of features is exactly EO.

>x_task3 = [y_(t-1), y_(t-2), ... y_(t-n)]

Starting from this, the RLS method can be applied with the dimensions of `x` being `n`. One value is added to the vector of features to take into account the intercept. The update value for `beta` is then computed as usual to perform **one step of RLS**. To make this model more precise, this step is to be iterated for the different time gaps : for each measure, consider the actual model `beta, V`, actual measure `y` and actual former measures `x` (up to `n`), and update iteratively the model.

```python
def RLSstep(y, x, n, beta, V, nu):
    # y = count for the current time ;  x = vector of features
    # n is here the number of features, not the embedding order!
    x.shape = (1,n)
    x1 = np.append(1,x)

    V = [...] # Update value of X (see code)
    alpha = V.dot(x1.T)
    yhat = x1.dot(beta)
    err = y - yhat

    beta = beta + alpha * err
    return (beta, V, err, yhat)
```

Task 4 is essentially based on the same logic, as it implements a RLS model except more features are taken into account. There is no difference in the `RLSstep` above, only in the dimensions of the features vector `x`. In task 4, vector `x` contains the past measures and information about the current time, namely the hour of the day and the day in the week. Hence, `x` has a size `n+t`, where `t=2`, the number of temporal features. Temporal features for task 4 are the following, and computed in a quite intuitive Python script in the implementation.

- Day in the week, read from the timegap, considering that the original timegap = 0 is a Thursday
- Hour of the day, read from the timegap as well

Updating models in parallel

Each model needs to be updated throughout the process. In the distributed framework of Spark, the `updateStateByKey` exactly allows states to be updated : we specify the states to be the models. More specifically, we chose to keep track of the current model with `beta,V`, the current MSE, the current error history `E`, and the number of steps done `N`. `D` is a history that has the size of the **EO**, to make sure that the previous `n` counts are taken into account when predicting the next output.

There is one state per model, labeled by their **key**, which contains the information about the device, so that each device has its associated model. For incoming data of `s` sensors, this indeed allows to update the `s` states in parallel at each batch arrival. Also, this allows efficient **scalability** in the number of sensors : adding or removing sensors just requires to increase the `s` parameter once in the code.

```
s = 18 # number of sensors
# ... (Task 3)
# Define initial values for model parameters
model = (beta, V, nu, mse, N, D, E)
initialStateRDD = sc.parallelize([(f'device{i+1}', (model)) for i in range(s)])
# ... (Task 4)
# Define initial values for model parameters
embedding_orders = [1, 2, 3, 4]
model = [beta, V, nu, mse, N, D, E]
initialStateRDD = sc.parallelize([(f'device{i+1}_embed{embed}', model+[embed])\
    for i in range(s)
    for embed in embedding_orders ])
```

For task 4, we decided to compare the models with different values of the embedding order, from 1 to 4, by running them in parallel as well. This implies a change in the definition of the states, as now a key must precise the number of the device and the embedding order for the model.

The incoming data is thus mapped into a list of key-value pairs, where the value is the relevant data for one sensor (namely for sensor `i`, time gaps and the corresponding counts for sensor `i`) and the key is the name of the associated model. For task 3, we simply map each batch to a list of key-value pairs as explained, and for task 4 we create one copy for each embedding order. Indeed, the models for sensor `i` all need the same data regardless of the embedding order. We make sure to create each value as an array of 2 columns : one with the time gaps `x[:,0]` and one with the counts of sensor `i`, `x[:,i]`.

```
# DStream manipulation for task 4
dataS = dataS.flatMap(lambda x: [(f'device{i}_embed{j}', x[:, 0] x[:, i])
                                 for i in range(1, x.shape[1])
                                 for j in embedding_orders])
```

We highlight that the way the states are defined allows perfect **scalability** in the number of sensors and different values of the embedding order to try. Regardless of what the producer sends, the number of states is parametrized by `s` solely, which can be easily modified, same goes for `embedding_orders`, which contains the different values to

launch in parallel. As a consequence, once we have found the best embedding order `e`, the list can be modified by `[e]` simply to only run one model per sensor, with `e` as embedding order.

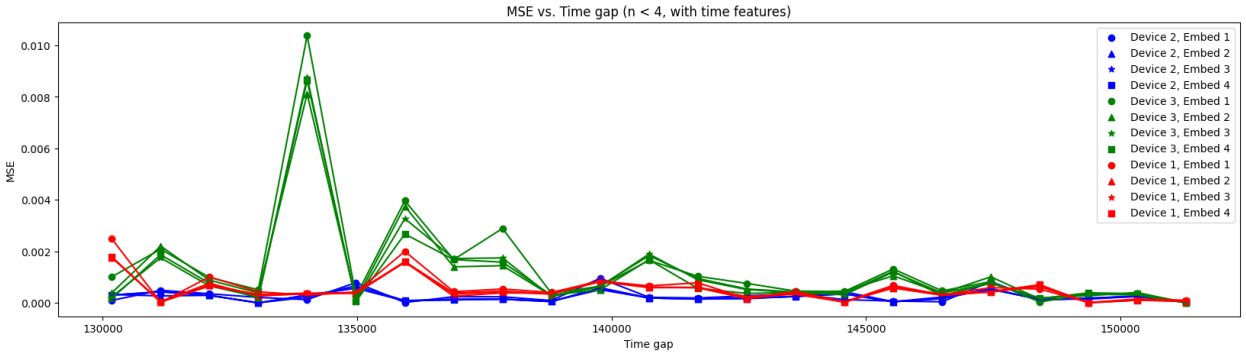
Evaluation of the RLS models

For each incoming batch, the state is updated using the `updateFunction` defined in our implementation. Its first input is made by the incoming batch so to process it as a whole, an iteration over it is required. Each row is a new record and will launch a step of RLS to update the model with the current measure. To do so, we store the current count in a variable `y`, and we build `x` the vector of features that consists in the last most recent elements in the history `D` of the state (see above), AND eventually some time-related features. The RLS step is performed on these and the new values of `beta, mse` are computed, and the history `D` is updated with the latest measure before the state is finally returned.

Each time a batch of `M` records is sent, the `new_values` object will contain 1 object of size `M`. For each record in `new_values[0]`, the RLS model of the according sensor embedding order will be updated, so there is a total of `M` updates for each incoming batch, which explains that our model is updated in a discontinue way. We **evaluate our models by looking at the MSE for the last iterations, analyzing batches from 15/04/2022 to 31/03/2023**, as asked in the assignment.

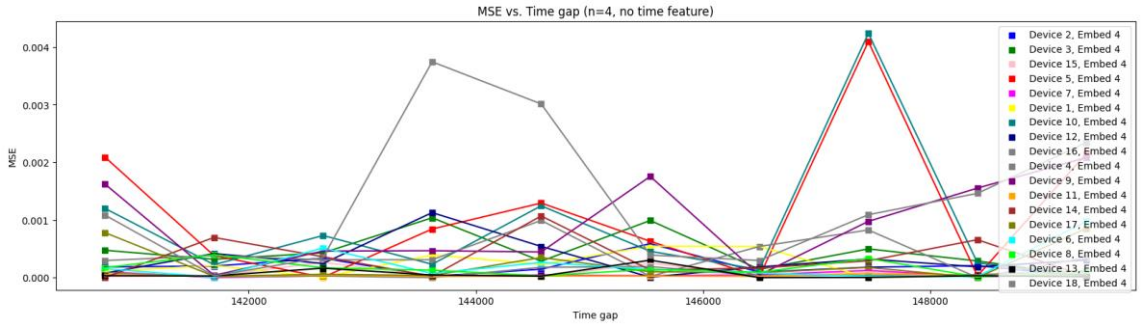
Value for the embedding order

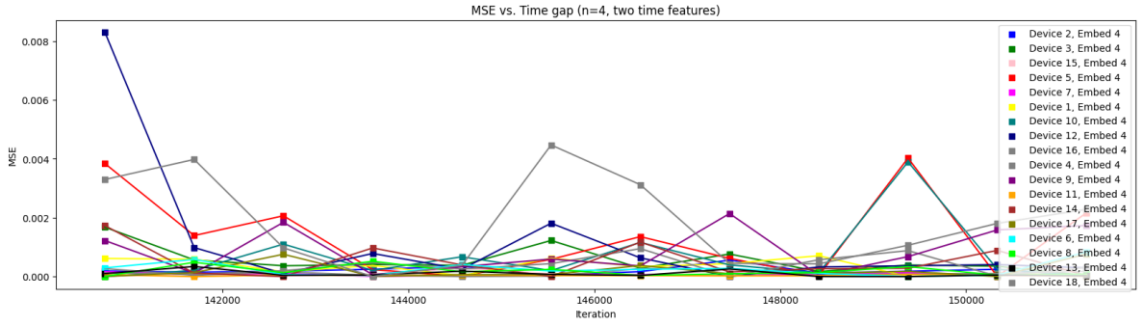
Running the models of 18 sensors for embedding order values of 1, ..., 4, we analyzed that there was no miraculous value for the embedding order among those. Indeed, results show some preference for a high embedding order, but the general tendance of the MSE still looks the same for all of them. Largest "anomalies" in the MSE are dominated by the embedding order = 1, and embedding order = 4 gave the "most satisfactory" results among the different values, which is hence our final choice. Moreover, the MSE converges to small values towards the end, which is the sign of a performant model.



Relevance of time-related features

Quite counter-intuitively, adding time-related features decreases the overall precision of the model. This can be observed by comparing the different plots for n=4 : we can see that the maximum MSE for the last iterations is 2 times smaller when no time features are observed. Other than that, the two models give quite similar results. **Our next steps** would be to study those added features more in detail to understand this behaviour





Parallelized architecture

In our way of doing, the state list is distributed across the different nodes of our (local) cluster. Then, each incoming batch is converted to a list of key-value pairs, where the key indicates to which state the value refers. Using the `updateStateByKey` method redirects the key-value pair to the node in which the dedicated state is held. This is the logic behind our distributed architecture, and we trust Spark to do the necessary. In addition, printing the output to the device results in the creation of 8 files, one for each partition. Gathering the results of all the parts give the full state of all the models.

```
~$ tree outputs/part-1684170810000
outputs/part-1684170810000
├── part-00000
├── part-00001
├── part-00002
├── part-00003
├── part-00004
├── part-00005
├── part-00006
├── part-00007
└── _SUCCESS
```

Link to our video: https://www.dropbox.com/s/ohv2b3uwvzn0tay/Phase2_final.mov?dl=0

References

[1] https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#creating-streaming-dataframes-and-streaming-datasets
[2] https://www.analyticsvidhya.com/blog/2020/11/what-is-the-difference-between-rdds-dataframes-and-datasets/
[3] https://stackoverflow.com/questions/59240277/insert-missing-date-rows-and-insert-old-values-in-the-new-rows-pyspark
[4] Spark and RLS lecture notes
[5] RLS exercise session