Task 3: fitting simple models

Study the Wine Quality Data Set from UCI Machine Learning repository. Build various linear and non-linear models and report the performance of each model. Also, report if any of the models is over- or underfitting.

Task 4: research

Suppose you are working with heterogeneous graph data, where nodes can be either customers, devices, recipients or phone numbers. The edges represent their relationship with additional attributes like `link_creation_date`, `number_of_links` and `last_link_date`. The nodes are also labelled based on whether they have been reported as fraudulent or not. Your task is to give a brief overview of metrics (`hint: betweenness, centrality etc.`) that can be calculated for nodes, edges or networks which could be used in machine learning models to detect similar patterns. Please also choose at least one novel method for learning the network representation and compare it to the ones pointed out earlier.

Task 5: writing SQL

In this task you have to write a sample SQL query for extracting some features potentially relevant for machine learning models. TransferWise has collected information about all payments that are carried out through its platform and the analytics team has stored payment information into a table called `PAYMENTS`.

The `PAYMENTS` table includes the following fields:

- `payment_id` (integer type) - the payment unique incremental id (larger id means more recent payment)
- `user_profile_id` (integer type) - the user unique id
- `source_currency_id` (integer type) - the source currency code of the payment
- `target_currency_id` (integer type) - the target currency code of the payment
- `payment_submit_time` (timestamp type) - the time the payment was submitted

The following provides a snapshot of the `PAYMENTS` table.

payment_id	user_profile_id	source_currency_id	target_currency_id	payment_submit_time
65932183	11111	2	1	2019-03-10 15:21:36.000
62216425	11111	1	2	2019-02-07 17:25:28.000
62025098	11111	2	1	2019-02-06 08:36:24.000
58611332	11111	2	1	2019-01-08 21:39:55.000
51330943	11111	2	1	2018-11-02 07:03:36.000
51194858	11111	1	2	2018-11-01 08:06:16.000
51139634	11111	1	2	2018-10-31 18:16:39.000
50972085	11111	1	2	2018-10-30 14:56:51.000
50864981	11111	2	1	2018-10-29 19:47:07.000
50624842	11111	2	1	2018-10-27 06:16:54.000
1200821	22222	1	2	2014-11-21 18:30:46.000
329699	22222	2	3	2014-03-03 20:25:06.000
192984	22222	2	1	2013-11-21 00:15:43.000
135023	22222	2	3	2013-09-10 10:31:23.000
97727	22222	2	3	2013-07-08 06:29:01.000
81688	22222	1	2	2013-06-04 07:01:46.000

Notes:

- One `user_profile_id` can have multiple `payment_id`'s. This records the payment history of the user
- Every payment has a different submit timestamp
- `source_currency_id` is the source currency code for that payment e.g. 1 relates to EUR and 2 relates to GBP
- `target_currency_id` is the target currency code for that payment e.g. 1 relates to EUR and 2 relates to GBP

Your challenge

1. Write a SQL query that performs the following aggregations per payment_id:

- `payment_cnt` - for each `payment_id` count the number of previous payments submitted by the user
- `target_ccy_cnt` - for each `payment_id` count the number of target currencies used by the user
- `source_ccy_cnt` - for each `payment_id` count the number of source currencies used by the user
- `oldest_payment_age` - for each `payment_id` calculate the age in days of the oldest payment made by the user
- `target_ccy_payments_cnt` - for each `payment_id` count the number of payments previously made on the same target currency code by the user
- `source_ccy_payments_cnt` - for each `payment_id` count the number of payments previously made on the same source currency code by the user
- `same_route_pmnts_cnt` - for each `payment_id` count the number of payments previously made on the same route by the user
- `backward_route_pmnts_cnt` - for each `payment_id` count the number of payments previously made on the backward route by the user

2. With focus on query performance, please explain different approaches you would use to make this query faster than a simple self-join. Note that it is OK to provide a solution to part 1 that relies on a self-join, but what else can we do (if anything) to make it performant and be able to execute millions of aggregations in a timely manner?

Hint: snippet of expected results

payment_id	payment_cnt	target_ccy_cnt	source_ccy_cnt	oldest_payment_age
65932183	10	2	2	134
62216425	9	2	2	103
62025098	8	2	2	102
58611332	7	2	2	73
51330943	6	2	2	6

Task 6: simple SQL

In this task you have to analyse the two following SQL snippets. Plese give an overview of the expected results and compare them in terms of computational complexity.

Snippet 1

```
SELECT
  id_user,
  recip_account,
  MIN(date_request_submitted) OVER (PARTITION BY recip_account, id_user)
FROM recipient rec
LEFT JOIN request req ON req.recipient_id = rec.id
WHERE rec.receiver_type = 'BUSINESS'
AND req.flag_cancelled <> 1
ORDER BY 3 DESC;
```

Snippet 2

```
SELECT
  id_user,
  recip_account,
  MIN(date_request_submitted)
FROM recipient rec
LEFT JOIN request req ON req.recipient_id = rec.id AND flag_cancelled <> 1
WHERE rec.receiver_type = 'BUSINESS'
GROUP BY 1,2
ORDER BY 3 DESC;
```