Functional and Non-Functional Requirements

Version Control

Version	Date	Update notes
v1	28 Apr 2021	• Extraction of functional and non-functional requirements from current specifications
v2 (Current)	02 May 2021	• Refactor based on client meeting, including recategorization of non-functional requirements

Functional Requirements

- **predictive monitor management**

1. a predictor monitor can be created by importing pickle files of one or many predictors, and can be deleted

2. details of a predictor monitor(e.g. its name) can be edited

3. predictor monitor generated can be read/viewed

- **Monitor Dashboard Management**

1. a dashboard can be created by inputting event log into its monitor,

and can be deleted; more than one dashboard can be created concurrently.

2. the progress of a monitor dashboard can be read/viewed

- **Monitor Dashboard view**

1. prediction results of the event logs and the aggregate data statistics(e.g. average case length, duration, and amount completed) can be read on the dashboard

2. the dashboard can be exported as a CSV file.

- **Case-level Predictions**

- **Predictor Training and training options**

1. an event log and a filter file(for binary predictors) can be imported for the training purpose.

2. a predictor can be named(with specific name).

3. a predictor can be trained with any advanced options and just the default training options (e.g. frequency encoding, no bucketing method, and XGBoost prediction) and advanced options (e.g. select the type of encoding, bucketing method, and prediction method).

4. a predictor can be deleted.

- **Predictor management**

1. information of a completed predictor , progress of a predictor being trained, can be read/viewed.

2. more than one predictor can be trained concurrently.

3. a predictor's name can be updated.

- **Predictor dashboard and storage**

1. MAE (for a remaining time predictor) and the F-score and the AUC (for a binary predictor) can be viewed in the dashboard.

2. completed predictor training jobs should be stored in memory.

Non-functional Requirements

- Maintainability:
    - Errors (if happen) should be located quickly and easily.

    - Bug fixes should not influence irrelevant parts of the system.
- Usability:
    - The plugin should be compatible with Apromore Community version
    - The code of queuing system should be decoupled from other parts, in case the queuing system is changed in the future.
- Recoverability:
    - Capable of recovering from sudden crashes and picking up work from where it left off
- Capacity/Scalability:
    - Be able to handle loads of up to one hundred million event logs at a time
    - The design should allow clients to add new types predictors and models without changing the code much easily
    - Be able to make predictions on tens of thousands of cases and training models on logs of up to one hundred million cases
    - Be able to handle dozens of concurrent requests.
    - A master node should manage the request queue and distribute tasks among a scalable number of worker nodes.
    - Be able to spawn new worker nodes or remove current ones, based on the load of the system, i.e., horizontal scaling
- Localization / Environmental:
    - ~~Integrate with ApromoreCore 7.20~~
    - Requires 2 different private images in docker when deployment: one for predictor and second for trainer services. For micro-service, be able to have two different artefacts i.e. docker-compose and Dockerfile
    - Docker swamp for orchestrating the containers
- Portability and compatibility:
    - Apromore license terms and use open source libraries and packages
    - Built using a master-slave architecture for the back-end.
    - Use a RESTful API to wrap around the predictor service
    - The back-end must follow good design and programming practices such as object oriented and design patterns when needed.
    - Built using a decoupled architecture with

        - a back-end micro-service implementing the core business logic written in Python using FastAPI
        - A front-end client built using the Angular framework
- Security:
    - Keep docker hub's vulnerability scanning on in order to monitor the security violations from the start
    - Type-safety needs to be considered in any part of code

- Reliability:
    - Unit tests & integration test coverage needs to be at least 70%
    - Good and detailed documentation of all the steps for understanding the project design and implementation details easily.
- Availability:
    - When main components crash, the recover should not be too long.
    - When non-essential components(such as queue system) crash, the main components should work so that system should be available to a large extent.
- Documentation:
    - Good and detailed documentation of all the steps for understanding the project design and implementation details easily.
    - Follow a good and uniform programming style (i.e. define data types of the variables when writing the Python code)
    - Codes should follow basic principles such as low-coupling and high cohesion.