OBAIC

- "AI ODBC for BI"
- Once this is defined and developed, BI vendors can connect to any AI framework (TensorFlow, PyTorch... etc) freely without concerning about the underlying implementation, just like what we used to have for database with ODBC
- As a BI platform, what specific call do I need this standard to provide so that I can better leverage any underlying AI/ML framework?
- As an AI practitioner, what should be the common denominator an AI framework should provide to support this standard?

**Decision to be made**

- Server VS server-less Or just define the protocol?: Protocol, and up to the vendor to take care of the actual implementation, server or server-less
- Data file type: What type of data we are supporting: e.g. for Delta needs to be parquet, RDBMS? Can modify the Jeffrey init cut below to support multiple data types, depending on the use case.
  - Inference: Pass by value should be good enough if it's only for predicting
  - Train: not immediate, maybe later in Phase 2
- Do we upload the data to AI (*passed by value*) and keep the data in the same location (*pass by reference*)?
- Metadata structure, what kind of JSON schema do we need
- Do we support training or just inference?
- Do we only support a specific model type (ONNX) or arbitrary number of framework
- Decouple model (asking the model to predict and train) and data (listing, upload, download)
- Tableau version of OBAIC: https://tableau.github.io/analytics-extensions-api/docs/ae_example_tabpy.html
- Qlik version of OBAIC: https://github.com/qlik-oss/server-side-extension
- Finalize Logo

**Question to clarify**

**FAQ**

Why should I share our model to you?

Ownership? Model and Data?

Security?

How can the data be accessed mechanically, for training?

This is a short doc illustrating a sample skeleton OBAIC protocol. This proposal envisions a data-centric workflow:

1. BI vendor has some data on which predictive analytics would be valuable.
2. BI vendor requests AI vendor (through OBAIC) to train/prepare a model that accepts features of a certain type (numeric, categorical, text, etc.)
3. BI vendor gives AI vendor a token to allow access to the training data with the above features. A SQL statement is a natural way to specify how to retrieve data from the datastore.
4. When model is trained, BI vendor can see the results of training (e.g., accuracy).
5. AI vendor provides predictions on data shared by BI vendor, again using an access token.

Format inspired by the Delta Sharing protocol doc.

**API**

**Train a New Model**

```plaintext
function TrainModel(inputs, outputs, modelOptions, dataConfig) -> UUID
```

*Example params:*
Model configuration is based on configs from the open-source Ludwig project. At a minimum, we should be able to define inputs and outputs in a fairly standard way. Other model configuration parameters are subsumed by the options field.

The data stanza provides a bearer token allowing the ML provider to access the required data table(s) for training. The provided SQL query indicates how the training data should be extracted from the source.

Example response:

```
{
  "modelUUID": "abcdef0123"
}
```

Consider also a fully SQL-like interface taking BigQuery ML model creation as an example and generalizing:

```
CREATE MODEL
    customerAge WITH ENCODING (type=numeric),
    activeInLastMonth WITH ENCODING (type=binary),
    canceledMembership WITH DECODING (type=binary)
FROM myData (sourceType=snowflake, endpoint="some/endpoint",
  bearerToken="...",
  query="SELECT foo FROM bar WHERE baz"
) AS (SELECT foo FROM BAR) WITH OPTIONS ();
```

List Models

```
function ListModels() -> List[UUID, Status]
```

Example response:
Show Model Config

function GetModelConfig(UUID) -> Config

Example response:

```json
{
  "models": [
    { "modelUUID": "abcdef0123", "status": "deployed" },
    { "modelUUID": "1234567890", "status": "training" }
  ]
}
```

The response here is essentially a pared-down version of the original training configuration.

Get Model Status

function GetModelStatus(UUID) -> Status

Example response:

```json
{
  "status": "errored",
  "message": "Failed to train"
}
```

Get Model Metrics

Get core evaluation metrics for a trained model.

function GetModelMetrics(UUID) -> Metrics

Example response:
Predict Using Trained Model

function PredictWithModel(UUID, dataConfig) -> Predictions

Example params

```json
{
  "uuid": "abcdef12345",
  "data": {
    "sourceType": "snowflake",
    "endpoint": "some/endpoint",
    "bearerToken": "...",
    "query": "SELECT foo FROM bar WHERE baz"
  }
}
```

A very similar data stanza to the train request, designating the feature data on which to predict.

Example response (as JSON here for convenience, not necessarily for large responses):

```json
{
  "data": [
    {
      "customerAge": 2,
      "activeInLastMonth": "false",
      "predicted__canceledSubscription": "true"
    },
    {
      "customerAge": 9,
      "activeInLastMonth": "true",
      "predicted__canceledSubscription": "false"
    }
  ]
}
```

Note that directly returning a large response set is not a good idea. In practice, the results could be streamed through something like a persistent socket connection.