Google Cloud

Kaggler のための BigQuery 活用手法

佐藤 貴海
カリフォルニア大学アーバイン校
計算機科学専攻博士課程在籍

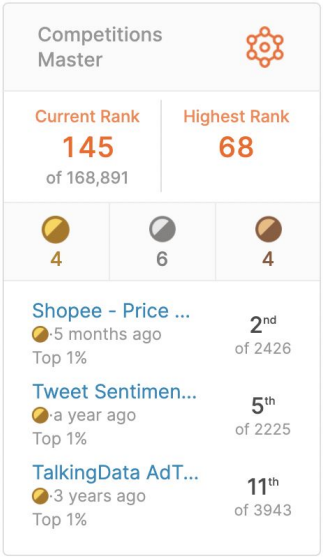
About Me: 佐藤 貴海 (SATO, Takami)

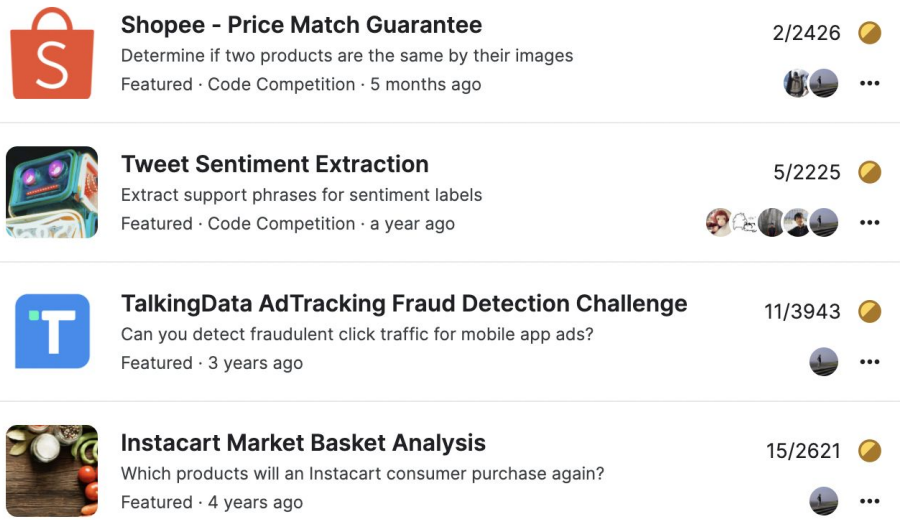
- 東京工業大学工学部経営工学科卒業 (2011 年)・修士修了 (2013 年)
  - 専門は数理最適化
- データサイエンティスト・機械学習エンジニアとして日本で 5 年程勤務
- 現在はカリフォルニア大学アーバイン校計算機科学専攻博士課程に在籍
  - 専門分野は機械学習のセキュリティ、特に自動運転システムを対象に研究

Published in Usenix Security '21

私と Kaggle

- KDD Cup の盛り上がりを見て興味をもつ（2014 年）
- 2016 年の Bosch Production Line Performance より本格参戦
- 以降、年に数コンペ参加、金メダル 1 個ぐらいのペースで参戦中
- Kaggler-ja slack (現在 11,850 人)参加者募集中！





Kaggleとは？

- **機械学習モデルのスコアを競うコンペを開催しているサイト**
  - 2017 年に Google が買収
- **数年前まで:**
  - 予測結果の CSV を提出する形式が主流
  - テストデータも配布されリーク発生もしばしば
  - ローカルでやりたい放題なので 100 モデルアンサンブルとかもあった
- **近年:**
  - Kaggle 上で GPU 付き Notebook が使える様に
  - Notebook を提出するコンペが主に
    - 計算時間制約のため無茶はできない
    - 時系列 API などでより現実に則した評価が可能に

Kaggle は業務に役に立つか？

直接は、

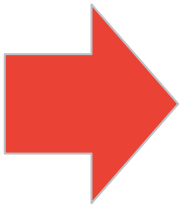
ほぼ役に立たない

入賞したからと言って高年収 DS 職が君を待っているとかは無い

Kaggle は業務に役に立つか？

- 大体の業務では。。。
  - 普通は機械学習が全く無いところに導入する。
  - 何もないから別に線形回帰でも成果が結構でる。
  - それよりデータ整備や実現場（工場）での運用整備が大変

そして9 割以上の Kaggler がよく目にする



「Kaggle は実業務に使えない」
「問題解くよりも Kaggle コンペに落とし込むことが数百倍大事」

につながる。

なぜ Kaggle をやるのか

- **ペアプロと似たような効果**
  - 実業務で同じ問題を複数人が別々に解くことは無い
  - 自分が挑戦した同じ問題の他人のコードは
    読みやすい & 超勉強になる。

- **実務上は Kaggle 上位モデルを抑えておけば大体 OK**
  - 最新論文を追うのは企業人にはつらい

- **設計するにも問題がある程度解けるべき**
  - Kaggle 入賞は良い客観的証明になる。
  - 知らない領域のデータの性質に詳しくなれる。

なので、

Kaggle は

Not for キャリアアップ

Good for スキルアップ & 知的探求

そうだ、

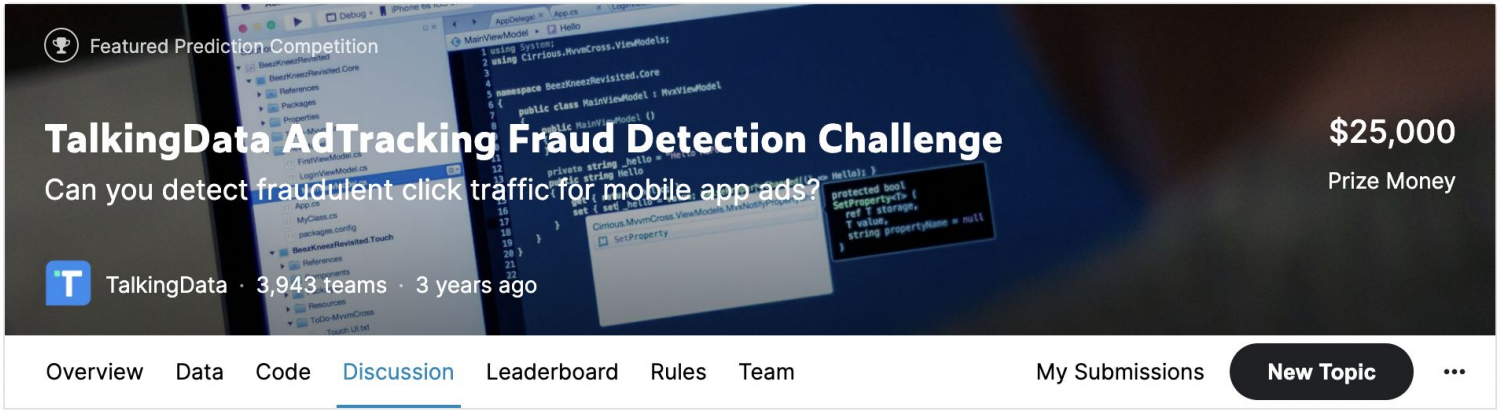
Kaggleで、
BigQueryを学ぼう！

BigQuery

BigQuery の有用性

- **魔法かと思うほど速い**
  - 数億件の処理が一瞬で帰ってくる。
- **企業のデータはほぼすべてデータベースに入っている**
  - DS 職に SQL は必須知識
  - というか最近は各社大体 BigQuery にデータが入ってる。
- **無料枠＋従量課金制**
  - インスタンス立てるタイプだと月 3 万円以上は必要
  - Kaggle で使うレベルは従量課金 & 普通初期クーポン（$300）に収まる。
- **DS としては分散処理のオプションは持っておくべき**
  - モデルが処理出来る形にするのは結構計算が重い。
  - SQL なら誰かに引き継ぎやすい。（Python・R よりは）

Kaggle での BigQuery の活用事例 (TalkingData)



- web のログから広告の不正なクリックを見つけるコンペ
- なんと訓練データのレコード数 184,903,890 件（1.8 億件）

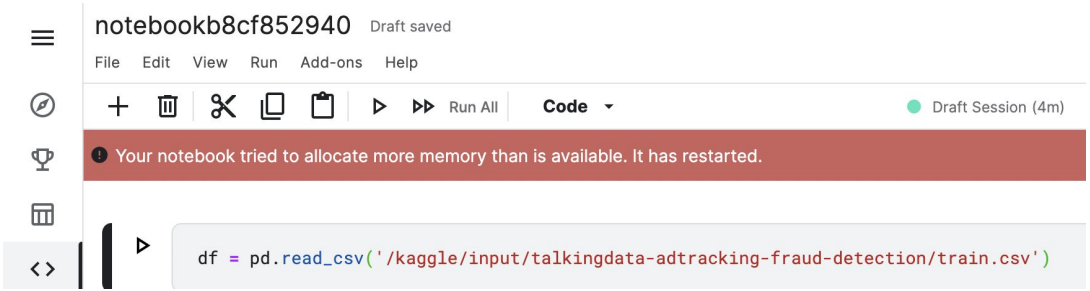
Data fields

Each row of the training data contains a click record, with the following features.

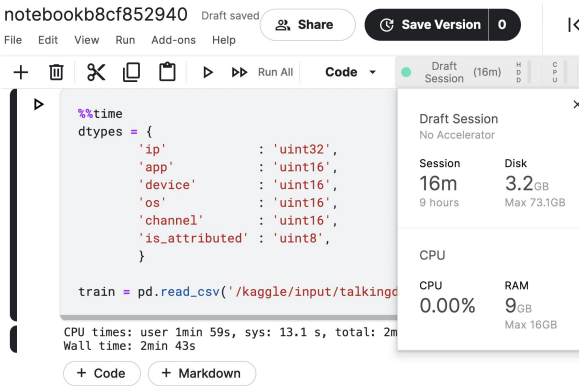
- `ip` : ip address of click.
- `app` : app id for marketing.
- `device` : device type id of user mobile phone (e.g., iphone 6 plus, iphone 7, huawei mate 7, etc.)
- `os` : os version id of user mobile phone
- `channel` : channel id of mobile ad publisher
- `click_time` : timestamp of click (UTC)
- `attributed_time` : if user download the app for after clicking an ad, this is the time of the app download
- `is_attributed` : the target that is to be predicted, indicating the app was downloaded

Kaggle での BigQuery の活用事例 (TalkingData)

メモリ 16 GBの Notebook では簡単に読むことすらできない



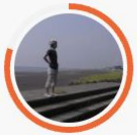
型指定すると読めるが、さらなる分析には厳しい



そうだ、

BigQuery を使おう！

私の 11 位の解法 with BigQuery



tkm2261

Topic Author

11th place

11th place features

Posted in talkingdata-adtracking-fraud-detection 3 years ago

97

Add Tags

https://www.kaggle.com/c/talkingdata-adtracking-fraud-detection/discussion/56250

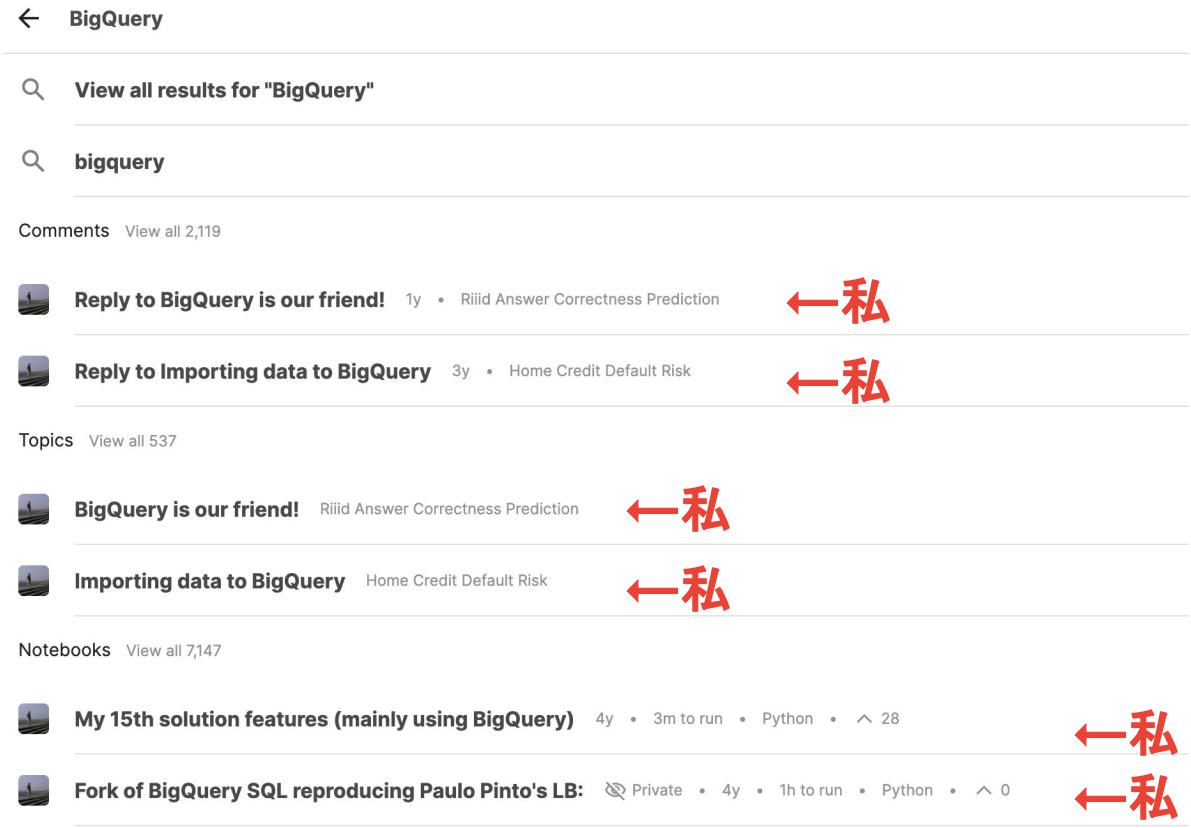
- BigQuery 上での処理は約 20 分！
  - Python でやろうと思うと半日以上かかったはず。
  - 実際にこのスレッドで 1 人が 1 日で再現できた。
  - 褒められた

Izmaylov Konstantin • (213th in this Competition) •

great solution! most suitable for production

さぞかし BigQuery は Kaggle で大人気なのだろう

Kaggle で BigQuery と検索すると。。。



正直、

私以外 BigQuery を Kaggle で使ってるのを見たことはない。

Kaggle で BigQuery いる？

- **実際必須ではない**
  - 近年は Notebook 提出が増えて、
    実際学習用のコードも Python で書いておいたほうが何かと便利
- **お金の無い学生でもコンペで勝てる可能性**
  - BigQuery は高額請求がバズって危険視されているが、
    実際 3 ヶ月コンペ参加しても数千円ぐらいなもの
  - Google Cloud でインスタンス借りたら数万は簡単に行く
- **SQL でデータ処理する経験を積める**
  - 私は業務経験から受領したデータは DB に入れるよう叩き込まれている
  - 学生（未経験）のうちに SQL でのデータ処理の経験を積める
- **さらに Window 関数が便利で速い（後述）**
  - 正直これが私が BigQuery を Kaggle で使う理由の 9 割

実際便利なので、

これから実コンペでのBigQueryの活用事例を紹介します。

その前に、分析 SQL 超入門

“分析とはGroup Byである”

Takami Sato

例えばマーケティング系のデータだと
ユーザと商品をどうGroup By（集約）して見るかとなる

- **ユーザー特徴**
  - 年代でGroup Byした RFM (**R**ecency **F**requency **M**onetary)
  - 性別でGroup Byした RFM
  - 店舗でGroup Byした RFM
  - ・・・
- **商品特徴**
  - カテゴリでGroup Byした平均価格
  - 価格帯でGroup Byした購買割合
  - ・・・
- **ユーザーx商品特徴**
  - 各年代の商品カテゴリの購買割合 etc.

更にどう集約するかも（平均 / 最大 / 最小 / 標準偏差等）

そのため基本となる SQL は

```sql
SELECT
    user_id,
    COUNT(1) as cnt_user,
    AVG(amount) as avg_user_amount,
    MIN(amount) as min_user_amount,
    MAX(amount) as max_user_amount,
    STDDEV(amount) as std_user_amount,
FROM
    train_transaction
GROUP BY
    user_id
```

※結果をdmt_userとして保存

Group by で集約して。

```sql
SELECT
    t.*,
    u.avg_user_amount,
    u.min_user_amount,
    u.max_user_amount,
    u.std_user_amount
FROM
    test_transaction as t
LEFT OUTER JOIN
    dmt_user as u
GROUP BY
    t.user_id = u.user_id
```

LEFT OUTER JOIN でくっつける

でも時系列的なのも考慮したいよね

- **ユーザーの直近 10 回の平均勾配金額**
- **商品の直近 3 ヶ月の販売個数**
- **そもそも全体で Group by したら未来の情報はいっちゃう**

でも時系列的なのも考慮したいよね

- **ユーザーの直近 10 回の平均勾配金額**
- **商品の直近 3 ヶ月の販売個数**
- **そもそも全体で Group by したら未来の情報はいっちゃう**

そう、簡単にできる Window 関数なら。

BigQuery では分析関数（Analytic function）となっている

分析関数の構文

```
analytic_function_name ( [ argument_list ] ) OVER over_clause

over_clause:
  { named_window | ( [ window_specification ] ) }

window_specification:
  [ named_window ]
  [ PARTITION BY partition_expression [, ...] ]
  [ ORDER BY expression [ { ASC | DESC }  ] [, ...] ]
  [ window_frame_clause ]

window_frame_clause:
  { rows_range } { frame_start | frame_between }

rows_range:
  { ROWS | RANGE }
```

https://cloud.google.com/bigquery/docs/reference/standard-sql/analytic-function-concepts

そんなに複雑ではなく「直近 10 回のユーザの平均購買額」の場合

```
SELECT
    AVG(amount) OVER(partition by user_id order by timestamp ROWS BETWEEN 10 PRECEDING AND 1 PRECEDING)
```

平均購買額を　ユーザごとに　　timestamp で過去 10 個について求める。

もし前回から過去全てを対象にしたい場合は

```
AVG(amount) OVER(partition by user_id order by timestamp ROWS BETWEEN UNBOUNDED PRECEDING AND 1 PRECEDING)
```

もし今回も対象にしたい場合は

```
AVG(amount) OVER(partition by user_id order by timestamp ROWS BETWEEN 10 PRECEDING AND CURRENT ROW)
```

さらに、もし過去 30 日の平均を計算したい場合

```
AVG(amount) OVER(partition by user_id order by timestamp RANGE BETWEEN 3600 * 24 * 30 PRECEDING AND 1 PRECEDING)
```

これを Pandas でやろうとするとかなり大変

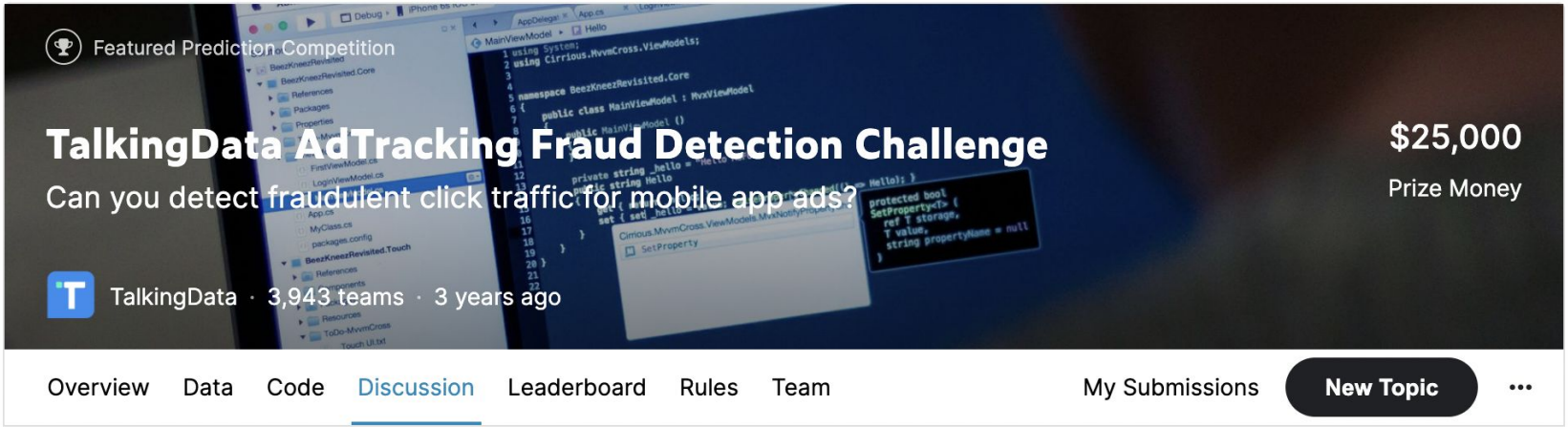
```
def agg_func(df):
    df['avg_user_amount_visit_10'] = df.sort_values('timestamp')['amount'].rolling(window=10).mean()
    return df
df.groupby('user_id').apply(agg_func)
```

- User ごとの DF の中で処理を色々やるので激遅
- 速くする Tips は色々あるが可読性を損ないがち
- ↑みたいなのを書きたくないので BigQuery をよく使う

改めて、

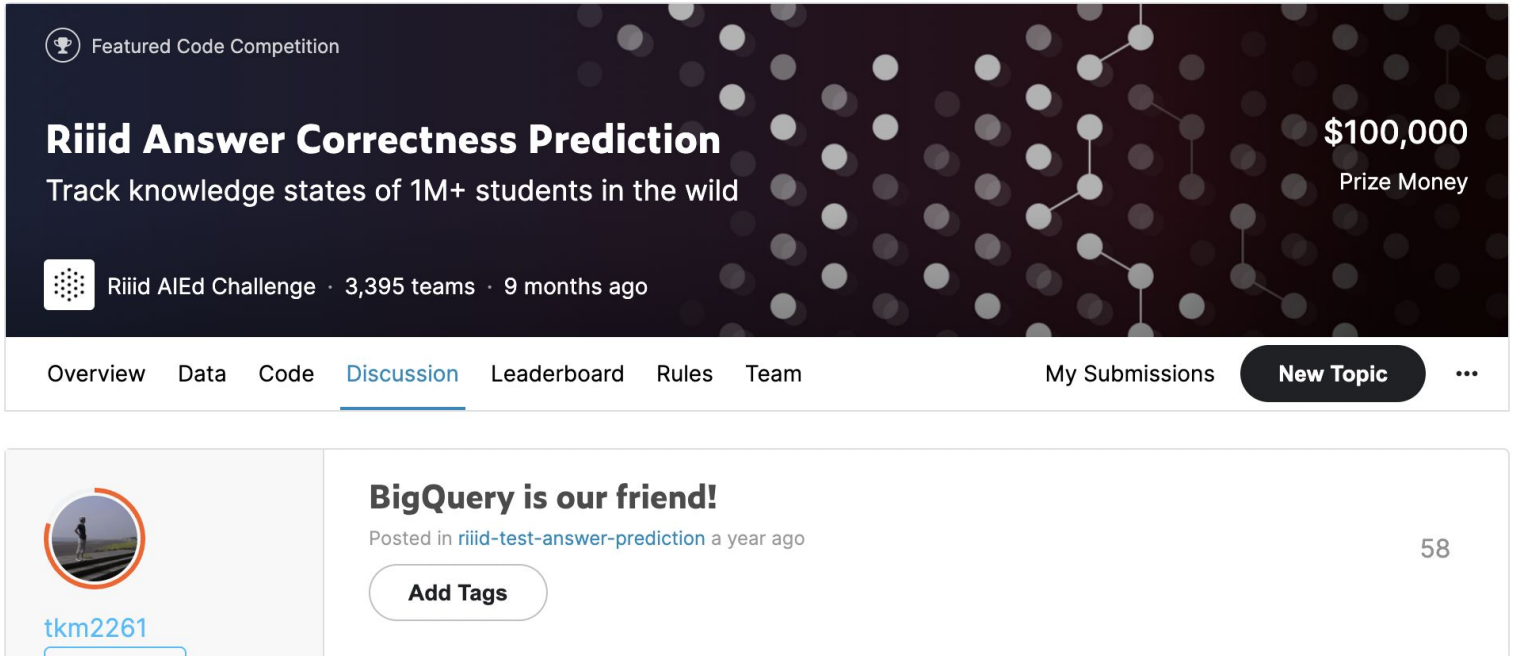
これから実コンペでのBigQueryの活用事例を紹介します。

Kaggle での BigQuery の活用事例 (TalkingData)



- このコンペでは機械的に特徴作成して試すのが鍵だった
- 遅めに参戦したのに勝てたのは BigQuery のおかげ
- 結局ひたすら Window 関数を変数を変えて試しまくった

Kaggle での BigQuery の活用事例 (Riild)



- こちらもレコードが1億件あるので、当初試したGBDT 用の特徴は BigQuery で作成
- その後 Transformer が強いコンペになったのでその特徴もBigQuery で準備（次ページ）

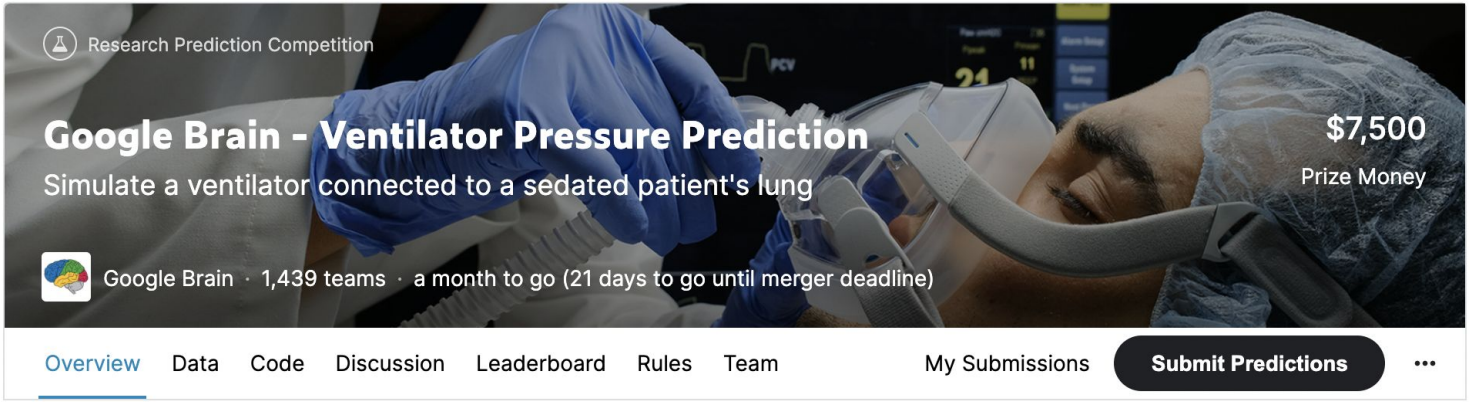
力技だが、リストの文字列を作ることが出来る

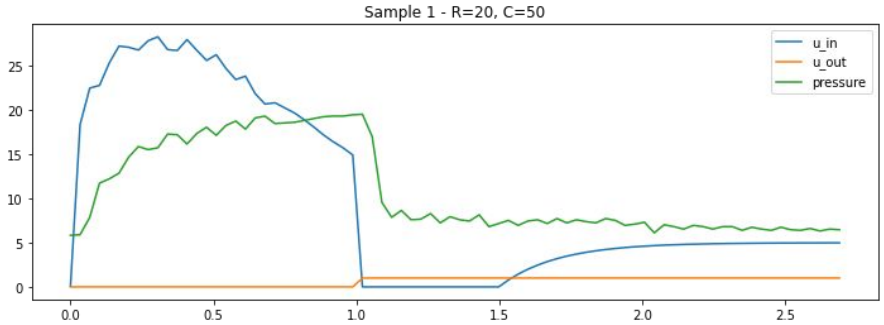
```
SELECT
    user_id,
    '[' || STRING_AGG(CAST(timestamp as STRING), ',' ORDER BY t.timestamp) || ']' timestamp,
    '[' || STRING_AGG(CAST(content_id as STRING), ',' ORDER BY t.timestamp) || ']' content_id,
    '[' || STRING_AGG(answered_correctly, ',' ORDER BY t.timestamp) || ']' answered_correctly,
    '[' || STRING_AGG(user_answer, ',' ORDER BY t.timestamp) || ']' user_answer,
FROM
    `train` as t
GROUP BY
    user_id
```

- Transformer に入れるのにユーザ毎のレコードにしたかった
- STRING_AGG で並べたのをコンマ区切りで結合
- 前後にカッコをつければ完成
- Python で読むときは eval() は遅いので json.loads() を使うと良い
- こんなのでも 1 億件あると Python だと超遅い

Kaggle での BigQuery の活用事例 (Ventilator Pressure Prediction)

- 現行コンペだけど公開 Notebook にちょうど良い例があったので紹介





- 3 秒間の人工呼吸器の圧力を諸々の変数から予測

Improvement base on Tensor Bidirect LSTM (0.173)

Python · Google Brain - Ventilator Pressure Prediction https://www.kaggle.com/kensit/improvement-base-on-tensor-bidirect-lstm-0-173

```
%%time
def add_features(df):
    df['area'] = df['time_step'] * df['u_in']
    df['area'] = df.groupby('breath_id')['area'].cumsum()
    df['cross']= df['u_in'] * df['u_out']
    df['cross2']= df['time_step'] * df['u_out']

    df['u_in_cumsum'] = (df['u_in']).groupby(df['breath_id']).cumsum()
    df['one'] = 1
    df['count'] = (df['one']).groupby(df['breath_id']).cumsum()
    df['u_in_cummean'] =df['u_in_cumsum'] /df['count']

    df['breath_id_lag'] = df['breath_id'].shift(1).fillna(0)
    df['breath_id_lag2'] = df['breath_id'].shift(2).fillna(0)
    df['breath_id_lagsame'] = np.select([df['breath_id_lag']==df['breath_id']],[1],0)
    df['breath_id_lag2same'] = np.select([df['breath_id_lag2']==df['breath_id']],[1],0)
    df['u_in_lag1'] = df['u_in'].shift(1).fillna(0)
    df['u_in_lag1'] = df['u_in_lag1'] * df['breath_id_lagsame']
    df['u_in_lag2'] = df['u_in'].shift(2).fillna(0)
    df['u_in_lag2'] = df['u_in_lag2'] * df['breath_id_lag2same']
    df['u_out_lag2'] = df['u_out'].shift(2).fillna(0)
    df['u_out_lag2'] = df['u_out_lag2'] * df['breath_id_lag2same']

    df['R'] = df['R'].astype(str)
    df['C'] = df['C'].astype(str)
    df['RC'] = df['R']+df['C']
    #df = pd.get_dummies(df)
    return df

train = add_features(train_ori)
```

```
CPU times: user 16.8 s, sys: 1.46 s, total: 18.2 s
Wall time: 17.9 s
```

- ここでは既存の特徴を組み合わせて新しい特徴を作っている。
- 赤枠の処理では 1 個前のと 2 個前の ‘u_in’ と ’u_out’ を求めている。
- ただ breath_id の境目の処理で複雑になっており可読性が低い
- ただ Pandas 君はこうやってベクトル演算風に書かないと著しく遅い

KEN SIT · COPIED FROM TOLGA DINCER +42,-9 · 10D AGO · 6,188 VIEWS

156 Edit My Copy 542

Improvement base on Tensor Bidirect LSTM (0.173)

Python · Google Brain - Ventilator Pressure Prediction https://www.kaggle.com/kensit/improvement-base-on-tensor-bidirect-lstm-0-173

- もしわかりやすくuser_id 毎の group_by で実装すると物凄く遅い
  - 17.9 秒が 11 分に悪化、36 倍遅くなる
  - Pandas はこういった例が度々あるので注意が必要

```
%%time
def map_breath(_df):
    df = _df.copy()
    df['area'] = df['time_step'] * df['u_in']
    df['area'] = df['area'].cumsum()
    df['cross']= df['u_in'] * df['u_out']
    df['cross2']= df['time_step'] * df['u_out']

    df['u_in_cumsum'] = df['u_in'].cumsum()
    df['count'] = np.arange(df.shape[0]) + 1
    df['u_in_cummean'] = df['u_in_cumsum'] / df['count']

    for i in range(1, 3):
        df[f'u_in_lag{i}'] = df['u_in'].shift(i).fillna(0)
        df[f'u_out_lag{i}'] = df['u_out'].shift(i).fillna(0)

    df['R'] = df['R'].astype(str)
    df['C'] = df['C'].astype(str)
    df['RC'] = df['R'] + df['C']

    return df

train = train_ori.groupby('breath_id').apply(map_breath).reset_index(drop=True)
```

```
CPU times: user 10min 52s, sys: 8.74 s, total: 11min 1s
Wall time: 11min 1s
```

TKM2261 · 1Y AGO · 8,629 VIEWS

144 Edit

FAST PANDAS LEFT JOIN (357x faster than pd.merge)

ちょうど良い例なので、このデータで BigQueryのデモをします。

どうやってデータを出し入れするか知らないと使えないですしね。

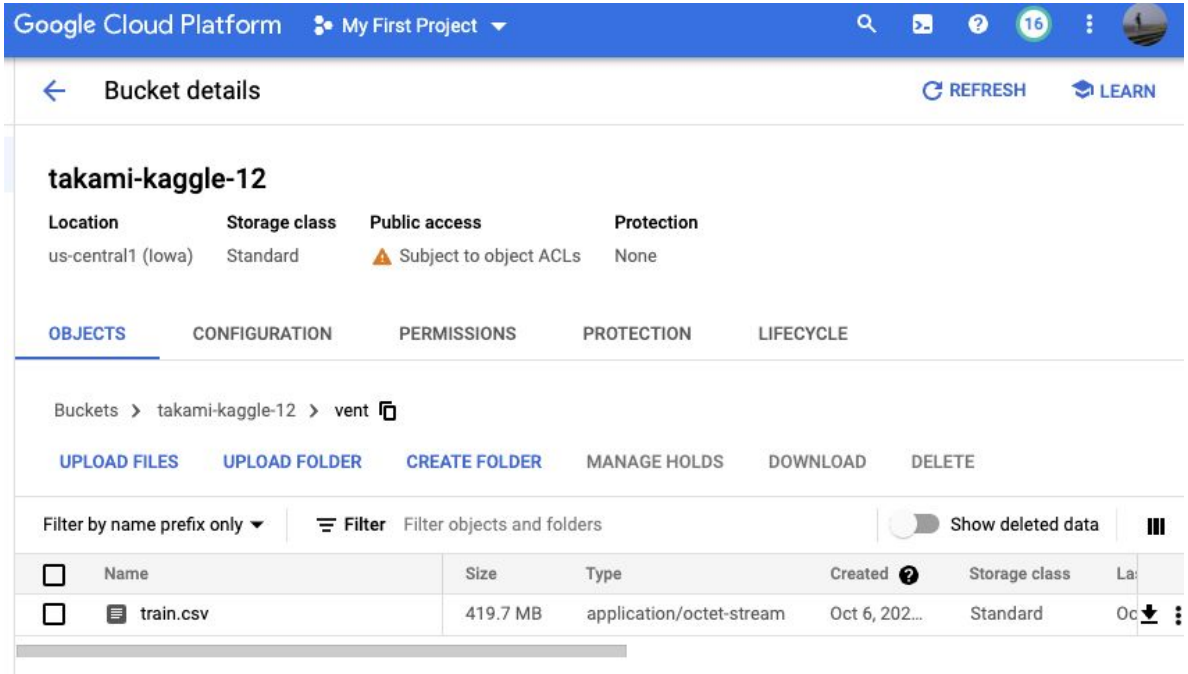
Kaggle Notebook からでも連携は出来るが

おそらくKaggler はモデルの学習は手元だと思うので 直接 Google Cloud から触る方法を解説します。

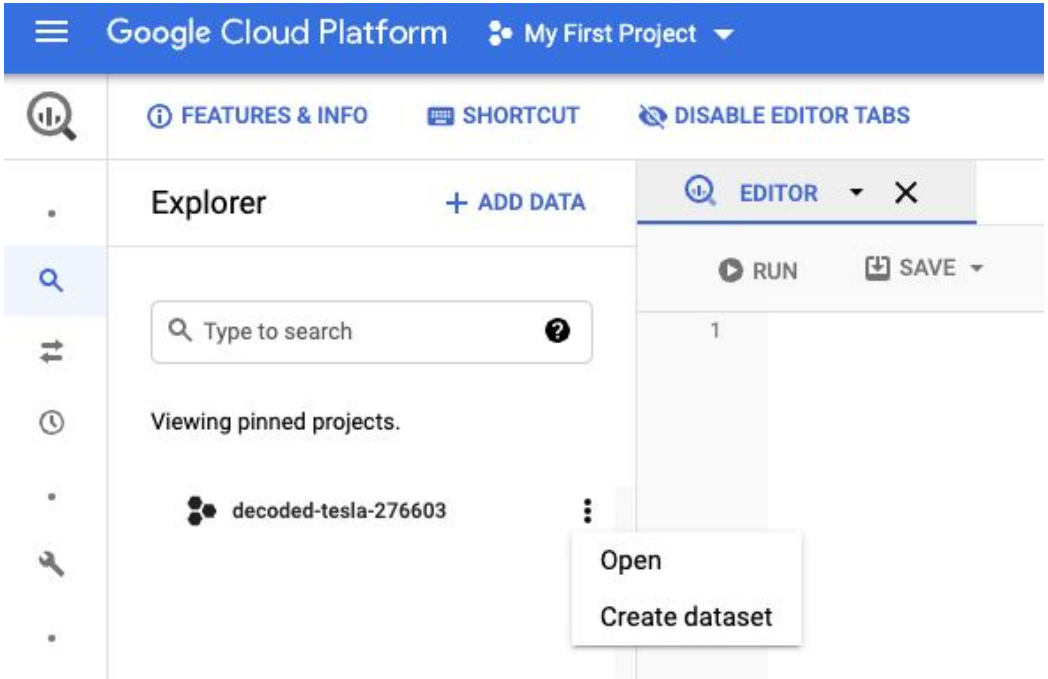
先日発表された Vertex AI Workbench とかでナウでヤングにやる方法もありそう。

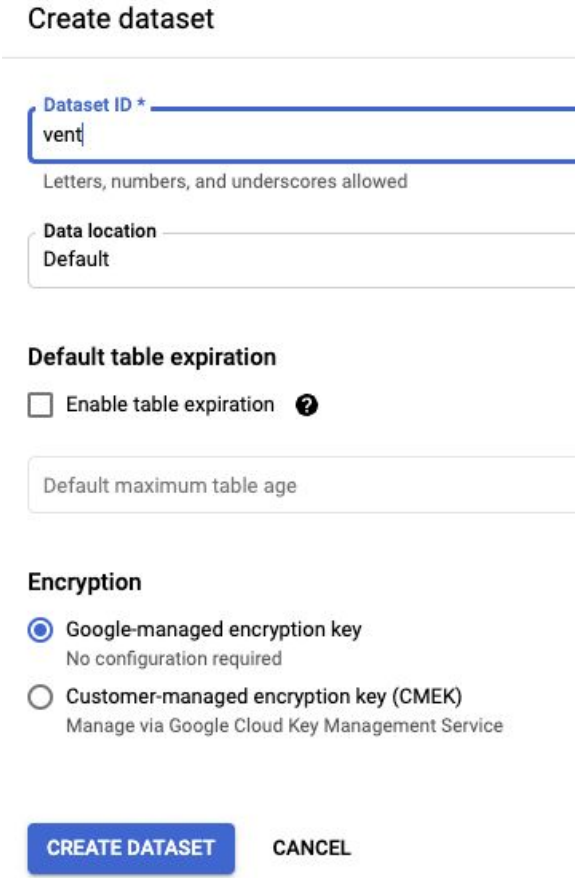
まずインポートしたいデータを適当な GCS に置く

- UI で Drug & Drop するか gsutil コマンドなどで置きましょう
- Kaggle Notebook から直接 gsutil などで上げることもできます
- Gzip 圧縮でも BigQuery は読めるのでファイルが大きいときは活用



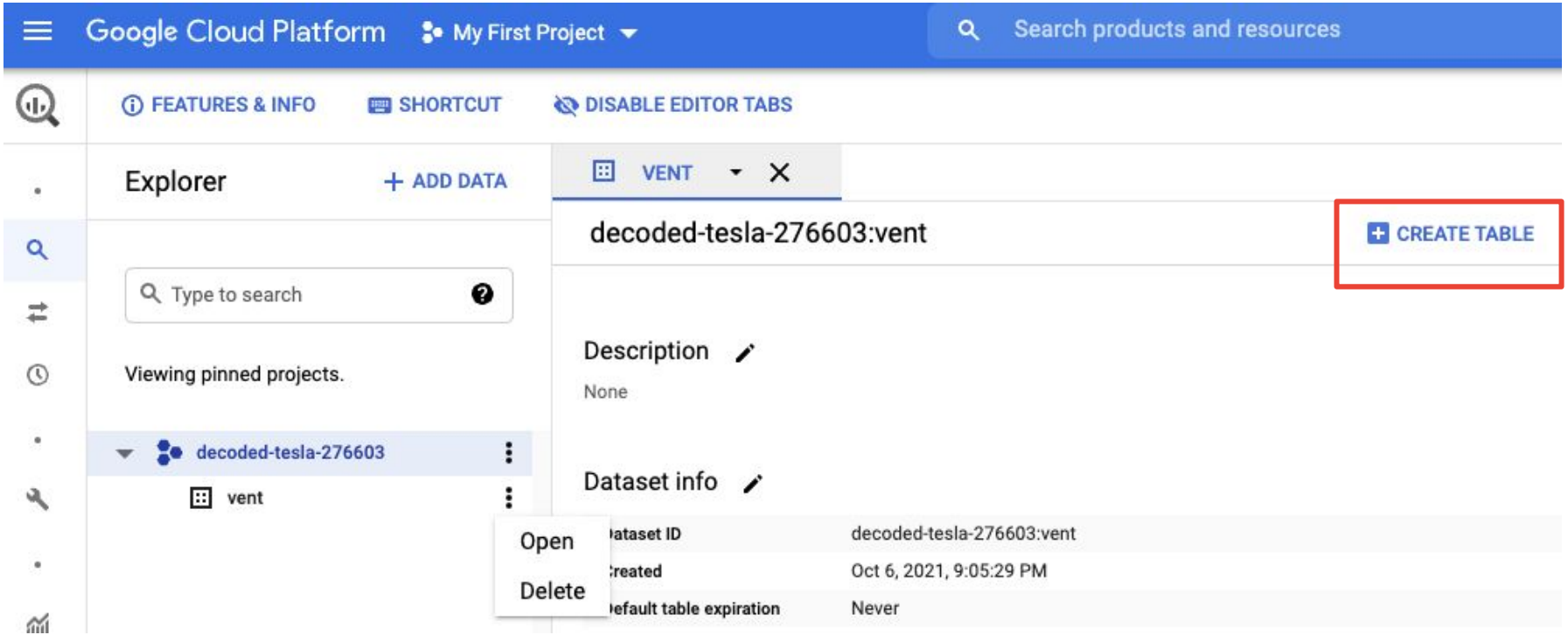
BigQuery の画面でデータセットを作成



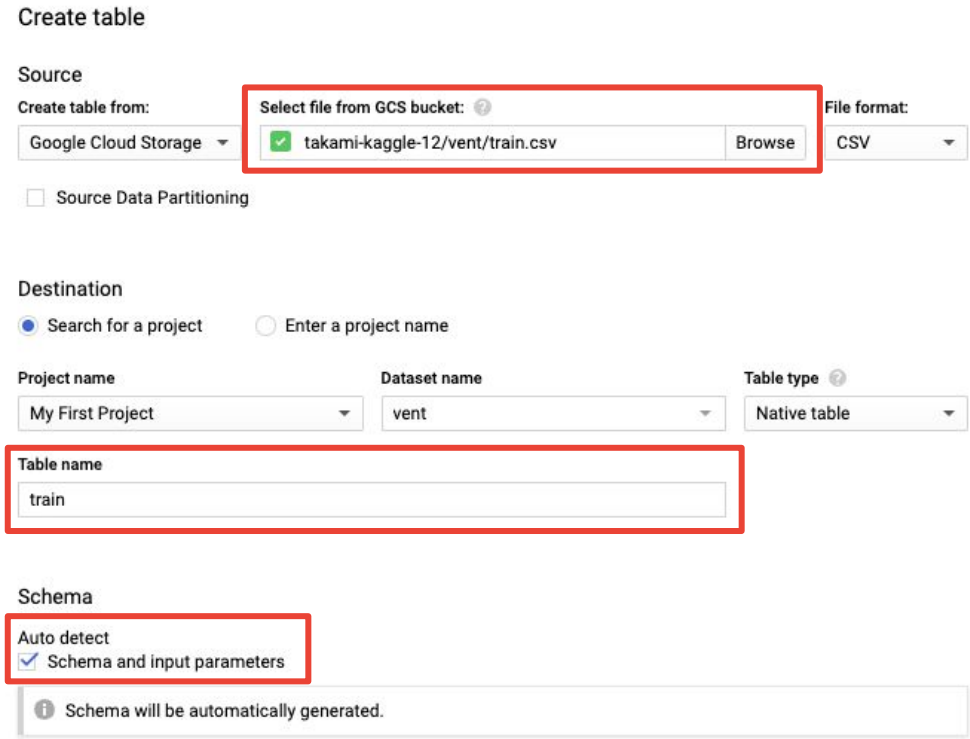


オプションはデフォルトで OK

作ったデータセットにテーブルを作成

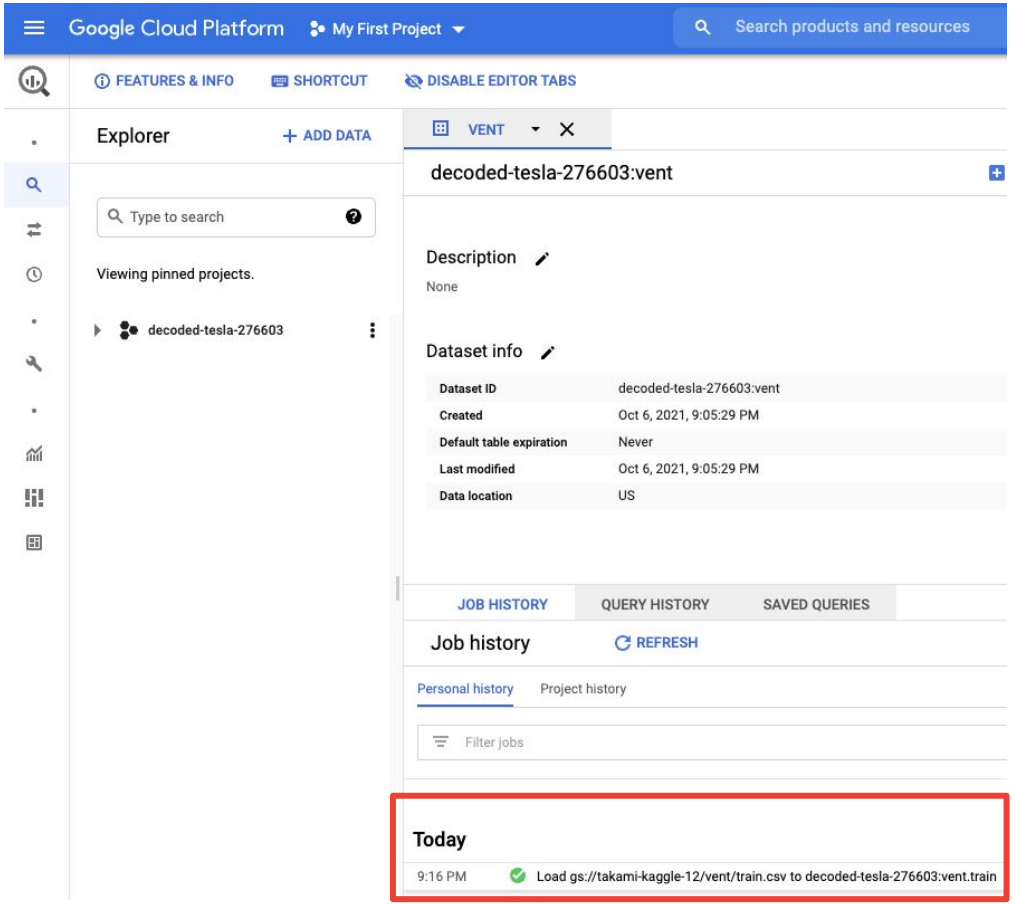


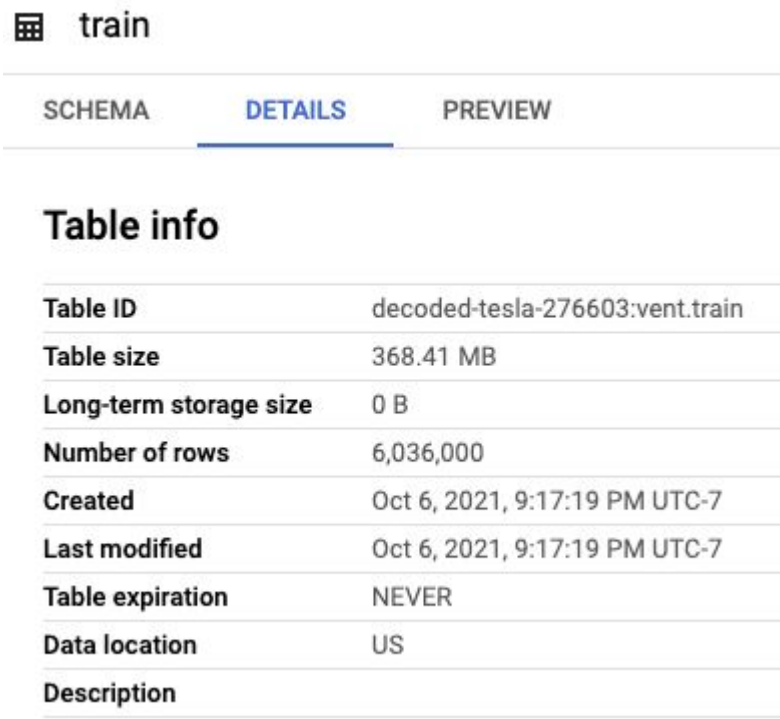
GCS からテーブルをインポート



- Schema Auto Detection で勝手に型推定して入れてくれます。
- Kaggle のきれいなデータで失敗することはほぼない
- 他のオプションは超大規模データようなので Kaggle では触らなくて OK
- 昔は業務で受領したデータのスキーマ書くだけで １日使ってたので良い時代です。

下の Job history が緑になってれば完了





先程の処理を BigQuery (SQL) で書くと（LAG で発生した null 以外）、

```
%%time
def add_features(df):
    df['area'] = df['time_step'] * df['u_in']
    df['area'] = df.groupby('breath_id')['area'].cumsum()
    df['cross']= df['u_in'] * df['u_out']
    df['cross2']= df['time_step'] * df['u_out']

    df['u_in_cumsum'] = (df['u_in']).groupby(df['breath_id']).cumsum()
    df['one'] = 1
    df['count'] = (df['one']).groupby(df['breath_id']).cumsum()
    df['u_in_cummean'] =df['u_in_cumsum'] /df['count']

    df['breath_id_lag'] = df['breath_id'].shift(1).fillna(0)
    df['breath_id_lag2'] = df['breath_id'].shift(2).fillna(0)
    df['breath_id_lagsame'] = np.select([df['breath_id_lag']==df['breath_id']],[1],0)
    df['breath_id_lag2same'] = np.select([df['breath_id_lag2']==df['breath_id']],[1],0)
    df['u_in_lag1'] = df['u_in'].shift(1).fillna(0)
    df['u_in_lag1'] = df['u_in_lag1'] * df['breath_id_lagsame']
    df['u_in_lag2'] = df['u_in'].shift(2).fillna(0)
    df['u_in_lag2'] = df['u_in_lag2'] * df['breath_id_lag2same']
    df['u_out_lag2'] = df['u_out'].shift(2).fillna(0)
    df['u_out_lag2'] = df['u_out_lag2'] * df['breath_id_lag2same']

    df['R'] = df['R'].astype(str)
    df['C'] = df['C'].astype(str)
    df['RC'] = df['R']+df['C']
    #df = pd.get_dummies(df)
    return df

train = add_features(train_ori)
```

```
CPU times: user 16.8 s, sys: 1.46 s, total: 18.2 s
Wall time: 17.9 s
```

RUN　SAVE　SCHEDULE　MORE　This query will proces

```
SELECT
    *,
    SUM(time_step * u_in) OVER(partition by breath_id order by time_step ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS area,
    u_in * u_out AS cross1,
    time_step * u_out AS cross2,
    ROW_NUMBER() OVER(partition by breath_id order by time_step) AS count1,
    SUM(u_in) OVER(partition by breath_id order by time_step ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS u_in_cumsum,
    AVG(u_in) OVER(partition by breath_id order by time_step ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS u_in_cummean,
    LAG(u_in, 1) OVER (PARTITION BY breath_id ORDER BY time_step) AS u_in_lag1,
    LAG(u_in, 2) OVER (PARTITION BY breath_id ORDER BY time_step) AS u_in_lag2,
    LAG(u_out, 1) OVER (PARTITION BY breath_id ORDER BY time_step) AS u_out_lag1,
    LAG(u_out, 2) OVER (PARTITION BY breath_id ORDER BY time_step) AS u_out_lag2,
    CAST(R as string) || CAST(C as string) AS RC
FROM
    `decoded-tesla-276603.vent.train`
ORDER BY
    breath_id, time_step
```

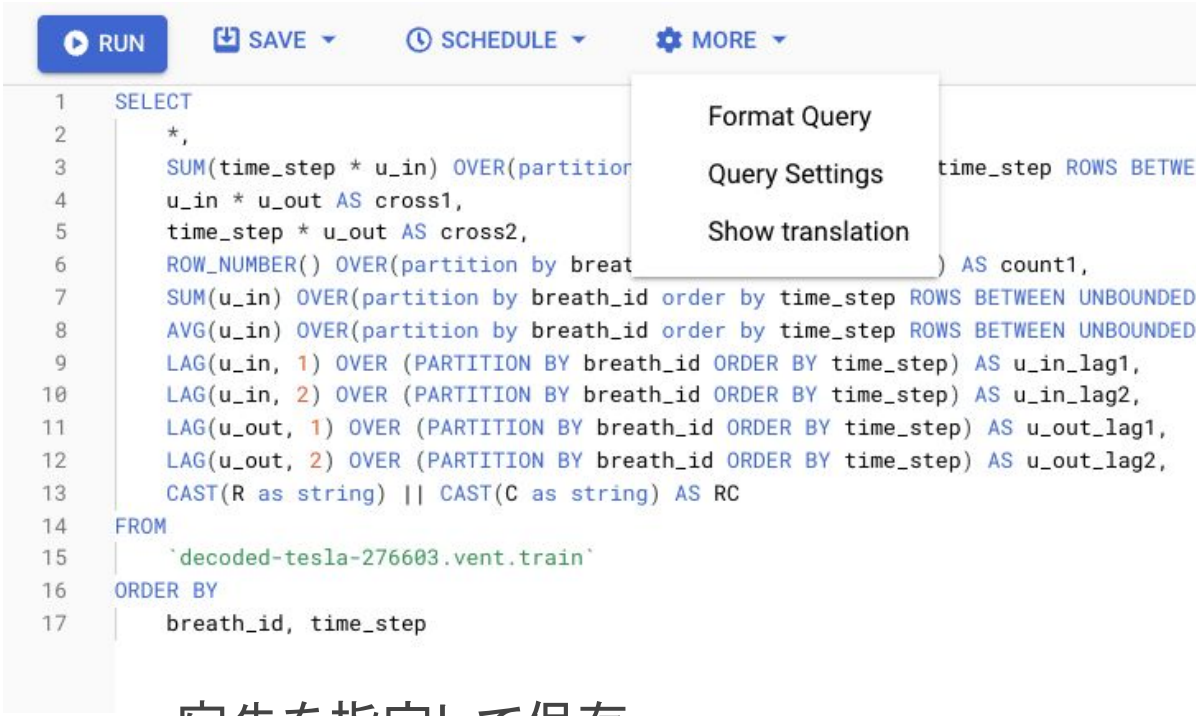
個人的にはとてもとても読みやすい

拡大版

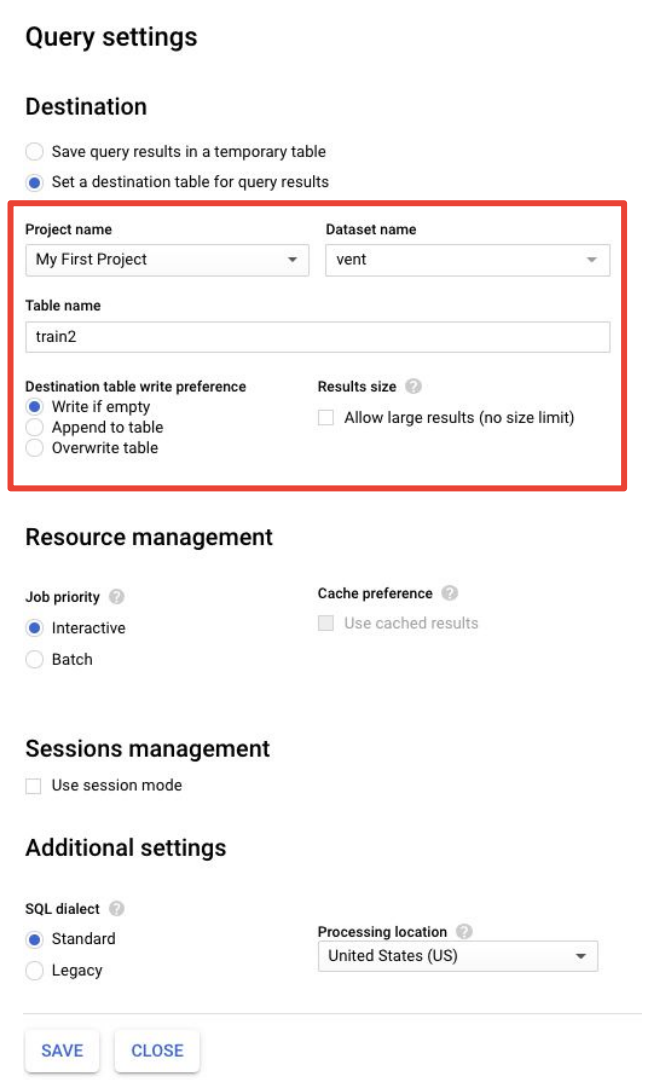
RUN SAVE SCHEDULE MORE This query will proces

```sql
SELECT
    *,
    SUM(time_step * u_in) OVER(partition by breath_id order by time_step ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS area,
    u_in * u_out AS cross1,
    time_step * u_out AS cross2,
    ROW_NUMBER() OVER(partition by breath_id order by time_step) AS count1,
    SUM(u_in) OVER(partition by breath_id order by time_step ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS u_in_cumsum,
    AVG(u_in) OVER(partition by breath_id order by time_step ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS u_in_cummean,
    LAG(u_in, 1) OVER (PARTITION BY breath_id ORDER BY time_step) AS u_in_lag1,
    LAG(u_in, 2) OVER (PARTITION BY breath_id ORDER BY time_step) AS u_in_lag2,
    LAG(u_out, 1) OVER (PARTITION BY breath_id ORDER BY time_step) AS u_out_lag1,
    LAG(u_out, 2) OVER (PARTITION BY breath_id ORDER BY time_step) AS u_out_lag2,
    CAST(R as string) || CAST(C as string) AS RC
FROM
    `decoded-tesla-276603.vent.train`
ORDER BY
    breath_id, time_step
```

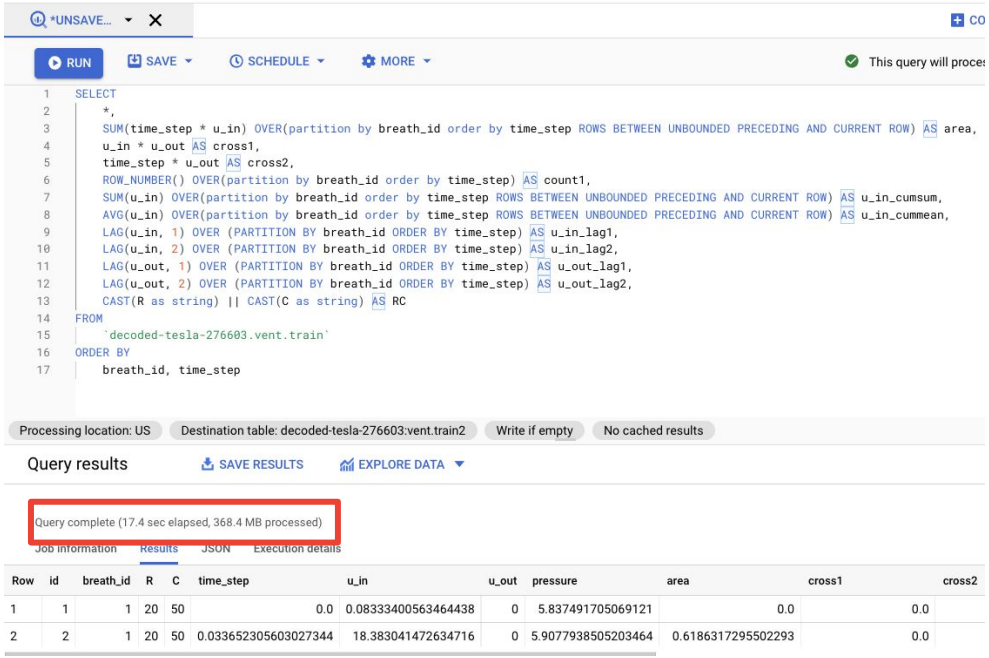
この結果を保存したいので Query Settings を開く



- 宛先を指定して保存
- 他のオプションはデフォルトで OK

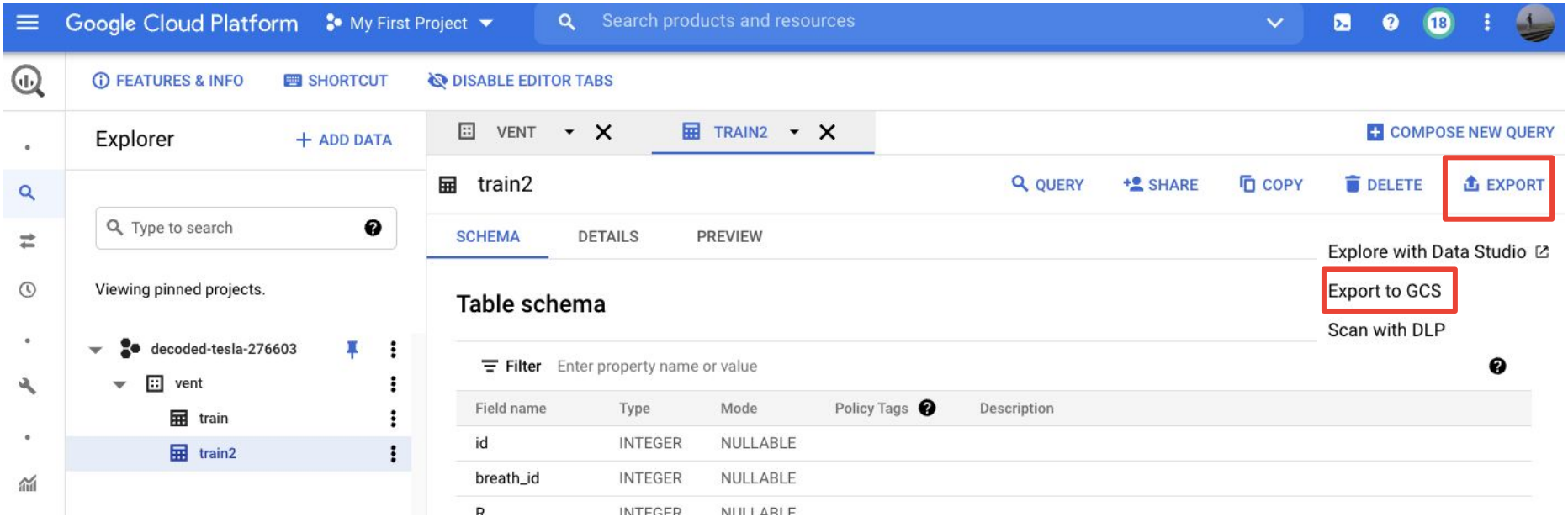


そして実行！



- 実行時間は 17.4 秒と元の 17.9 秒に比べてほぼ同程度
  - 11 分と比べれば雲泥の差、可読性も高い。
- 小さいデータなので差はほぼないが、BigQuery はメモリに乗らないデータも高速に処理できるので色々試してみてください。

出力結果を手元に DL



- テーブルを選んで GCS に出力
- 10 MB以下だと直接 DL できるが GCS に書いておいたほうが無難
  - Kaggle だと大体 CUI の Linux マシンで gsutil コマンドで DL
- 一連の流れは bq コマンドと gsutil コマンドで自動化もできる

まとめ

- Kaggle はデータ分析の勉強＆キャッチアップに最適
- 業務でよく使われる BigQuery に Kaggle で慣れよう！
- BigQuery は魔法の様に速い＆
  時系列処理が Window 関数で書きやすい
- Kaggle で BigQuery 使う仲間が欲しい。

Thank you!

質問などはTwitterの@tkm2261まで