TP 3+4
Kit de référence

NB : attention aux copier-coller, qui peuvent modifier les apostrophes et les guillemets. Si vous avez des erreurs de syntaxe après un copier-coller, vérifiez ces caractères.

Guide pour Big Query

Datasets

Tous les objets que vous référencez (table, UDF, modèle de ML) doit être “qualifié” avec le nom du projet, celui du dataset, et celui de l’objet lui-même.

Ainsi, pour une table `mytable` présente dans votre dataset `binome_XNN`, le nom qualifié est `` `but-tp34.binome_XNN.mytable` ``, entre apostrophes inversées.

Dans les modèles de requêtes présentés en exemple, pensez à adapter le nom du dataset Big Query à celui de votre binôme ! Attention, certains objets ne sont pas dans votre dataset, mais dans un qui s’appelle `utils`.

Mise au point des requêtes

Lorsqu’il est demandé de créer une table à partir du résultat d’une requête, il est plus simple de mettre au point la partie `SELECT`, d’ajuster en fonction des résultats, puis d’ajouter l’ordre `CREATE OR REPLACE TABLE …` lorsque ceux-ci sont satisfaisants. Exemple :

- Mise au point d’une requete :

```
SELECT * FROM `but-tp34.binome_XNN.source_table` WHERE a = 1
```

- Création de la table :

```
CREATE OR REPLACE `but-tp34.binome_XNN.final_table` AS
SELECT * FROM `but-tp34.binome_XNN.source_table` WHERE a = 1
```

Types de données

Dans la console Big Query, les types numériques apparaissent comme INT, FLOAT, mais dans les requêtes il faut être plus précis : `INT64` et `FLOAT64`, respectivement, sinon la requête sera refusée.

Les **types tableaux** se précisent ainsi dans les requêtes : `ARRAY<FLOAT64>` (pour un tableau de flottants).

Pour la question 3

Pour répondre à cette question, il faut “déplier” le tableau `vectors` de la table `raw_vectors`. Cela revient à considérer un tableau JSON comme une mini-table, que l’on joint (produit cartésien) à la table originelle pour en démultiplier les lignes.

Le “dépliage” du tableau se fait avec la fonction `UNNEST()` de Big Query, la jointure avec l’opérateur `CROSS JOIN` classique.

Voici un exemple simplifié, avec des vecteurs simples (des entiers et non des objets imbriqués). Si notre table de départ, `mytable`, contient ceci :

id	myarray
1	[‘a’, ‘b’, ‘c’, ‘d’]
2	[‘e’, ‘f’]

La requête suivante va faire le “dépliage” :

```
SELECT id, element
FROM mytable
CROSS JOIN
UNNEST(myarray) AS element;
```

Le résultat sera celui-ci :

id	element
1	‘a’
1	‘b’
1	‘c’

1	‘d’
2	‘e’
2	‘f’

Si on besoin de mettre en face la position de chaque élément dans le tableau, on peut étendre la requête :

```
SELECT id, element, position
FROM mytable
CROSS JOIN
UNNEST(vector) AS element
WITH OFFSET AS position;
```

Pour tester la requête rapidement sans créer une table mytable pour l’occasion :

```
WITH mytable AS (
  SELECT 1 AS id, ['a', 'b', 'c','d'] AS myarray
  UNION ALL
  SELECT 2 AS id, ['e', 'f'] AS myarray
)
SELECT id, element, position
FROM mytable
CROSS JOIN
UNNEST(myarray) AS element
WITH OFFSET AS position;
```

Pour répondre à la question 3, il faut en plus accéder au sous-tableau `vector` qui se situe à l’intérieur des éléments du tableau principal (analogue à `myarray` dans l’exemple). L’équivalent de `element` est alors un objet JSON, dont on peut extraire l’attribut `vector` grâce à la syntaxe suivante : `element.vector` (à utiliser dans la liste des colonnes après le SELECT).

Pour la question 5

Voici le code de la fonction `join_vectors()` du dataset `utils` :

```
CREATE OR REPLACE TABLE FUNCTION `but-tp34.utils.join_vectors`
  (vec1 ARRAY<FLOAT64>, vec2 ARRAY<FLOAT64>)
RETURNS TABLE<x FLOAT64, y FLOAT64> AS
SELECT
  x, vec2[SAFE_OFFSET(position)] AS y
```

```
  FROM UNNEST(vec1) AS x WITH OFFSET AS position
```

- C'est une fonction qui renvoie une table, d'où le `TABLE FUNCTION`
- Elle prend 2 paramètres, `vec1` et `vec2`, qui sont des vecteurs de flottants
- Elle renvoie une structure de table, avec 2 colonnes x et y, des flottants
- Elle utilise la fonction `UNNEST()` pour "déplier" `vec1` et parcourir `vec2` au même rythme, grâce à l'offset

Vous pouvez tester la fonction directement :

```
SELECT *
FROM `but-tp34.utils.join_vectors`([1., 2., 3.], [4., 5., 6.])
```

Comme c'est une fonction de table, il est logique de l'interroger avec `SELECT * FROM`.

La question 5 demande de créer une autre fonction (dans votre dataset cette fois), qui va appeler celle-ci, et utiliser les colonnes x et y du résultat pour calculer la distance. Cette fonction ne doit pas être une fonction de table car elle renvoie un flottant. En voici le squelette :

```
CREATE OR REPLACE FUNCTION `but-tp34.binome_XNN.euclidian2`
  (vec1 ARRAY<FLOAT64>, vec2 ARRAY<FLOAT64>)
RETURNS FLOAT64
AS ((
  SELECT …
))
```

Noter les 2 niveaux de parenthèses, nécessaires pour la syntaxe.

Pour la question 6.a

Il est suggéré d'utiliser une clause `WITH` pour avoir une "variable locale" de table correspondant au morceau d'ID 31.

La clause `WITH` s'utilise ainsi :

```
WITH song_31 AS (
  SELECT …
)
SELECT …
FROM une_table
JOIN song_31 …
WHERE …
```

song_31 n'est pas une vraie table mais se comporte comme telle dans la requête (et uniquement pour celle-ci). On peut lui appliquer des jointures (une jointure interne dans l'exemple), y faire référence dans les clauses SELECT et WHERE, etc.

Parfois il est commode d'avoir plusieurs "variables de table", la syntaxe est la suivante :

```
WITH variable1 AS (
  SELECT …
),
Variable2 AS (
  SELECT …
  FROM variable1…
),
Variable3 AS (
  SELECT …
  FROM variable2…
)
SELECT …
```

Ici on a 3 "variables", noter qu'elles se référencent les unes les autres.

Question 7

Voici la requête du pivot :

```
WITH unfolded AS (
  SELECT
  vmd.* except (vector),
  element,
  'V' || CAST(position AS STRING) AS position
  FROM `but-tp34.binome_XNN.vectors_with_metadata` AS vmd
  CROSS JOIN UNNEST(vmd.vector) AS element WITH OFFSET position
)
SELECT *
FROM unfolded
PIVOT (
  MIN(element)
  FOR position IN (
    'V0', 'V1', 'V2', 'V3', 'V4', 'V5', 'V6', 'V7', 'V8', 'V9', 'V10', 'V11', 'V12',
'V13', 'V14', 'V15'
  )
)
```

La pseudo-table `unfolded` est la version “dépliée” de `vectors_with_metadata` après éclatement des vecteurs eux-mêmes (au début, on avait juste déplié le niveau de tableau précédent). La position (offset) est transformée en chaîne de caractère `'Vxx'` car ses valeurs détermineront le nom des colonnes du résultat.

Ensuite le résultat est pivoté, grâce à l’opérateur `PIVOT` ; il implique une agrégation, comme pour un tableau croisé dynamique. Ici les valeurs sont toutes uniques, l’opération `MIN()` n’a pas d’effet si ce n’est satisfaire le besoin d’agrégation.

Pour entraîner un modèle de k-means (question 7.a), il faut respecter le protocole suivant :

```
CREATE OR REPLACE MODEL `but-tp34.binome_XNN.kmeans`
TRANSFORM (...)
OPTIONS (
…
)
AS SELECT …
```

- Dans la partie `TRANSFORM`, on met entre parenthèses une liste de colonnes qui servira effectivement pour l’entraînement. Ce peut être un sous-ensemble des colonnes du `SELECT`, mais aussi des expressions dérivées de celles-ci (c’est un comme si on créait une table intermédiaire, sauf que la logique de transformation est liée au modèle pour appliquer le feature engineering plus tard à l’inférence)
- Les options sont des listes de type *`PARAMETRE = valeur`*, séparées par des virgules ; ce sont les hyperparamètres du modèle par exemple
- Le `SELECT` fournit le jeu de données d’entraînement. Dans ce cas précis, il est plus simple d’encapsuler notre requête de pivot telle quelle dans un `SELECT * FROM (`*`requête pivot`*`)`

Pour l’inférence (question 7.b), la syntaxe est un peu différente et utilise la fonction `ML.PREDICT()` de Big Query :

```
SELECT *
FROM ML.PREDICT(
  MODEL `but-tp34.binome_XNN.kmeans`,
  (SELECT …)
)
```

Le `SELECT` interne doit être entre parenthèse, et sa structure correspondre à celle du jeu d’entraînement. Dans notre cas, il faut donc reproduire la requête pivot… (on aurait pu faire une fonction !).