Creating a Sampled Dataset

Learning Objectives

1. Setup up the environment
2. Sample the natality dataset to create train, eval, test sets
3. Preprocess the data in Pandas dataframe

Introduction

In this notebook, we'll read data from BigQuery into our notebook to preprocess the data within a Pandas dataframe for a small, repeatable sample.

We will set up the environment, sample the natality dataset to create train, eval, test splits, and preprocess the data in a Pandas dataframe.

Each learning objective will correspond to a **#TODO** in this student lab notebook -- try to complete this notebook first and then review the solution notebook.

Set up environment variables and load necessary libraries

In [ ]:
```
!sudo chown -R jupyter:jupyter /home/jupyter/training-data-analyst
```

In [ ]:
```
!pip install --user google-cloud-bigquery==1.25.0
```

Note: Restart your kernel to use updated packages.

Kindly ignore the deprecation warnings and incompatibility errors related to google-cloud-storage.

Import necessary libraries.

In [1]:
```
from google.cloud import bigquery
import pandas as pd
```

Lab Task #1: Set up environment variables so that we can use them throughout the notebook

In [2]:
```
%%bash
# TODO 1
export PROJECT=$(gcloud config list project --format "value(core.project)")
echo "Your current GCP Project Name is: "$PROJECT
```

```
Your current GCP Project Name is: qwiklabs-gcp-04-7cdacd129561
```

In [9]:
```
PROJECT = "qwiklabs-gcp-04-7cdacd129561"  # Replace with your PROJECT
```

Create ML datasets by sampling using BigQuery

We'll begin by sampling the BigQuery data to create smaller datasets. Let's create a BigQuery client that we'll use throughout the lab.

In [10]:

```
bq = bigquery.Client(project = PROJECT)
```

We need to figure out the right way to divide our hash values to get our desired splits. To do that we need to define some values to hash within the module. Feel free to play around with these values to get the perfect combination.

In [11]:

```
modulo_divisor = 100
train_percent = 80.0
eval_percent = 10.0

train_buckets = int(modulo_divisor * train_percent / 100.0)
eval_buckets = int(modulo_divisor * eval_percent / 100.0)
```

We can make a series of queries to check if our bucketing values result in the correct sizes of each of our dataset splits and then adjust accordingly. Therefore, to make our code more compact and reusable, let's define a function to return the head of a dataframe produced from our queries up to a certain number of rows.

In [12]:

```
def display_dataframe_head_from_query(query, count=10):
    """Displays count rows from dataframe head from query.

    Args:
        query: str, query to be run on BigQuery, results stored in dataframe.
        count: int, number of results from head of dataframe to display.
    Returns:
        Dataframe head with count number of results.
    """
    df = bq.query(
        query + " LIMIT {limit}".format(
            limit=count)).to_dataframe()

    return df.head(count)
```

For our first query, we're going to use the original query above to get our label, features, and columns to combine into our hash which we will use to perform our repeatable splitting. There are only a limited number of years, months, days, and states in the dataset. Let's see what the hash values are. We will need to include all of these extra columns to hash on to get a fairly uniform spread of the data. Feel free to try less or more in the hash and see how it changes your results.

In [13]:

```
# Get label, features, and columns to hash and split into buckets
hash_cols_fixed_query = """
SELECT
    weight_pounds,
    is_male,
    mother_age,
```

```
    plurality,
    gestation_weeks,
    year,
    month,
    CASE
        WHEN day IS NULL THEN
            CASE
                WHEN wday IS NULL THEN 0
                ELSE wday
            END
        ELSE day
    END AS date,
    IFNULL(state, "Unknown") AS state,
    IFNULL(mother_birth_state, "Unknown") AS mother_birth_state
FROM
    publicdata.samples.natality
WHERE
    year > 2000
    AND weight_pounds > 0
    AND mother_age > 0
    AND plurality > 0
    AND gestation_weeks > 0
"""

display_dataframe_head_from_query(hash_cols_fixed_query)
```

Out[13]:

	weight_pounds	is_male	mother_age	plurality	gestation_weeks	year	month	date	state	m
0	7.063611	True	32	1	37	2001	12	3	CO	
1	4.687028	True	30	3	33	2001	6	5	IN	
2	7.561856	True	20	1	39	2001	4	5	MN	
3	7.561856	True	31	1	37	2001	10	5	MS	
4	7.312733	True	32	1	40	2001	11	3	MO	
5	7.627994	False	30	1	40	2001	10	5	NY	
6	7.251004	True	33	1	37	2001	11	5	WA	
7	7.500126	False	23	1	39	2001	9	2	OK	
8	7.125340	False	33	1	39	2001	1	4	TX	
9	7.749249	True	31	1	39	2001	1	1	TX	

Using `COALESCE` would provide the same result as the nested `CASE WHEN`. This is preferable when all we want is the first non-null instance. To be precise the `CASE WHEN` would become `COALESCE(wday, day, 0) AS date`. You can read more about it here.

Next query will combine our hash columns and will leave us just with our label, features, and our hash values.

In [14]:

```
data_query = """
SELECT
    weight_pounds,
    is_male,
    mother_age,
    plurality,
```

```
    gestation_weeks,
    FARM_FINGERPRINT(
        CONCAT(
            CAST(year AS STRING),
            CAST(month AS STRING),
            CAST(date AS STRING),
            CAST(state AS STRING),
            CAST(mother_birth_state AS STRING)
        )
    ) AS hash_values
FROM
    ({CTE_hash_cols_fixed})
""".format(CTE_hash_cols_fixed=hash_cols_fixed_query)

display_dataframe_head_from_query(data_query)
```

Out[14]:

	weight_pounds	is_male	mother_age	plurality	gestation_weeks	hash_values
0	7.063611	True	32	1	37	4762325092919148672
1	4.687028	True	30	3	33	2341060194216507348
2	7.561856	True	20	1	39	-8842767231851202242
3	7.561856	True	31	1	37	7957807816914159435
4	7.312733	True	32	1	40	-5961624242430066305
5	7.627994	False	30	1	40	5493295634082918412
6	7.251004	True	33	1	37	-2988893757655690534
7	7.500126	False	23	1	39	-6735199252008114417
8	7.125340	False	33	1	39	-3514093303120687641
9	7.749249	True	31	1	39	2175328516857391398

The next query is going to find the counts of each of the unique 657484 `hash_values` . This will be our first step at making actual hash buckets for our split via the `GROUP BY` .

In [15]:

```
# Get the counts of each of the unique hash of our splitting column
first_bucketing_query = """
SELECT
    hash_values,
    COUNT(*) AS num_records
FROM
    ({CTE_data})
GROUP BY
    hash_values
""".format(CTE_data=data_query)

display_dataframe_head_from_query(first_bucketing_query)
```

Out[15]:

	hash_values	num_records
0	-6735199252008114417	6
1	-2818158671747967146	16
2	-4192703448845442406	610

	hash_values	num_records
3	8263154982898115196	1395
4	6227678821205992496	809
5	-492294964451713448	497
6	-3574477993584580214	4
7	160395265237815829	398
8	6709266860196047792	19
9	-8934255065242073897	318

The query below performs a second layer of bucketing where now for each of these bucket indices we count the number of records.

In [16]:

```
# Get the number of records in each of the hash buckets
second_bucketing_query = """
SELECT
    ABS(MOD(hash_values, {modulo_divisor})) AS bucket_index,
    SUM(num_records) AS num_records
FROM
    ({CTE_first_bucketing})
GROUP BY
    ABS(MOD(hash_values, {modulo_divisor}))
""".format(
    CTE_first_bucketing=first_bucketing_query, modulo_divisor=modulo_divisor)

display_dataframe_head_from_query(second_bucketing_query)
```

Out[16]:

	bucket_index	num_records
0	4	398118
1	17	222562
2	64	283091
3	39	224255
4	43	201054
5	98	374697
6	48	370308
7	45	265930
8	62	426834
9	38	338150

The number of records is hard for us to easily understand the split, so we will normalize the count into percentage of the data in each of the hash buckets in the next query.

In [17]:

```
# Calculate the overall percentages
percentages_query = """
SELECT
    bucket_index,
```

```
    num_records,
    CAST(num_records AS FLOAT64) / (
    SELECT
        SUM(num_records)
    FROM
        ({CTE_second_bucketing})) AS percent_records
FROM
    ({CTE_second_bucketing})
""".format(CTE_second_bucketing=second_bucketing_query)

display_dataframe_head_from_query(percentages_query)
```

Out[17]:

	bucket_index	num_records	percent_records
0	48	370308	0.011218
1	91	333267	0.010096
2	43	201054	0.006090
3	1	163893	0.004965
4	64	283091	0.008576
5	4	398118	0.012060
6	97	480790	0.014564
7	38	338150	0.010243
8	62	426834	0.012930
9	98	374697	0.011351

We'll now select the range of buckets to be used in training.

In [18]:

```
# Choose hash buckets for training and pull in their statistics
train_query = """
SELECT
    *,
    "train" AS dataset_name
FROM
    ({CTE_percentages})
WHERE
    bucket_index >= 0
    AND bucket_index < {train_buckets}
""".format(
    CTE_percentages=percentages_query,
    train_buckets=train_buckets)

display_dataframe_head_from_query(train_query)
```

Out[18]:

	bucket_index	num_records	percent_records	dataset_name
0	1	163893	0.004965	train
1	57	453019	0.013723	train
2	68	197797	0.005992	train
3	51	180001	0.005453	train
4	17	222562	0.006742	train

	bucket_index	num_records	percent_records	dataset_name
5	38	338150	0.010243	train
6	29	453175	0.013728	train
7	39	224255	0.006793	train
8	45	265930	0.008056	train
9	43	201054	0.006090	train

We'll do the same by selecting the range of buckets to be used evaluation.

In [19]:

```
# Choose hash buckets for validation and pull in their statistics
eval_query = """
SELECT
    *,
    "eval" AS dataset_name
FROM
    ({CTE_percentages})
WHERE
    bucket_index >= {train_buckets}
    AND bucket_index < {cum_eval_buckets}
""".format(
    CTE_percentages=percentages_query,
    train_buckets=train_buckets,
    cum_eval_buckets=train_buckets + eval_buckets)

display_dataframe_head_from_query(eval_query)
```

Out[19]:

	bucket_index	num_records	percent_records	dataset_name
0	85	368045	0.011149	eval
1	87	523881	0.015870	eval
2	83	411258	0.012458	eval
3	89	256482	0.007770	eval
4	81	233538	0.007074	eval
5	86	274489	0.008315	eval
6	84	341155	0.010334	eval
7	88	423809	0.012838	eval
8	80	312489	0.009466	eval
9	82	468179	0.014182	eval

Lastly, we'll select the hash buckets to be used for the test split.

In [20]:

```
# Choose hash buckets for testing and pull in their statistics
test_query = """
SELECT
    *,
    "test" AS dataset_name
FROM
    ({CTE_percentages})
```

```
WHERE
    bucket_index >= {cum_eval_buckets}
    AND bucket_index < {modulo_divisor}
""".format(
    CTE_percentages=percentages_query,
    cum_eval_buckets=train_buckets + eval_buckets,
    modulo_divisor=modulo_divisor)

display_dataframe_head_from_query(test_query)
```

Out[20]:

	bucket_index	num_records	percent_records	dataset_name
0	94	431001	0.013056	test
1	90	286465	0.008678	test
2	93	215710	0.006534	test
3	91	333267	0.010096	test
4	97	480790	0.014564	test
5	98	374697	0.011351	test
6	96	529357	0.016036	test
7	99	223334	0.006765	test
8	95	313544	0.009498	test
9	92	336735	0.010201	test

In the below query, we'll `UNION ALL` all of the datasets together so that all three sets of hash buckets will be within one table. We added `dataset_id` so that we can sort on it in the query after.

In [21]:

```
# Union the training, validation, and testing dataset statistics
union_query = """
SELECT
    0 AS dataset_id,
    *
FROM
    ({CTE_train})
UNION ALL
SELECT
    1 AS dataset_id,
    *
FROM
    ({CTE_eval})
UNION ALL
SELECT
    2 AS dataset_id,
    *
FROM
    ({CTE_test})
""".format(CTE_train=train_query, CTE_eval=eval_query, CTE_test=test_query)

display_dataframe_head_from_query(union_query)
```

Out[21]:

	dataset_id	bucket_index	num_records	percent_records	dataset_name

	dataset_id	bucket_index	num_records	percent_records	dataset_name
0	1	83	411258	0.012458	eval
1	1	89	256482	0.007770	eval
2	0	5	449280	0.013610	train
3	0	12	412875	0.012507	train
4	0	40	333712	0.010109	train
5	0	56	226752	0.006869	train
6	0	50	184434	0.005587	train
7	0	28	449682	0.013622	train
8	0	26	492824	0.014929	train
9	0	35	250505	0.007588	train

Lastly, we'll show the final split between train, eval, and test sets. We can see both the number of records and percent of the total data. It is really close to that we were hoping to get.

In [22]:

```
# Show final splitting and associated statistics
split_query = """
SELECT
    dataset_id,
    dataset_name,
    SUM(num_records) AS num_records,
    SUM(percent_records) AS percent_records
FROM
    ({CTE_union})
GROUP BY
    dataset_id,
    dataset_name
ORDER BY
    dataset_id
""".format(CTE_union=union_query)

display_dataframe_head_from_query(split_query)
```

Out[22]:

	dataset_id	dataset_name	num_records	percent_records
0	0	train	25873134	0.783765
1	1	eval	3613325	0.109457
2	2	test	3524900	0.106778

Now that we know that our splitting values produce a good global splitting on our data, here's a way to get a well-distributed portion of the data in such a way that the train, eval, test sets do not overlap and takes a subsample of our global splits.

Lab Task #2: Sample the natality dataset

In [28]:

```
# TODO 2
# TODO -- Your code here.
# every_n allows us to subsample from each of the hash values
```

```
every_n = 1000
splitting_string = "ABS(MOD(hash_values, {0} * {1}))".format(every_n, modulo_div
def create_data_split_sample_df(query_string, splitting_string, lo, up):
    """Creates a dataframe with a sample of a data split.

    Args:
        query_string: str, query to run to generate splits.
        splitting_string: str, modulo string to split by.
        lo: float, lower bound for bucket filtering for split.
        up: float, upper bound for bucket filtering for split.
    Returns:
        Dataframe containing data split sample.
    """

    query = "SELECT * FROM ({0}) WHERE {1} >= {2} and {1} < {3}".format(
        query_string, splitting_string, int(lo), int(up))
    df = bq.query(query).to_dataframe()
    return df

train_df = create_data_split_sample_df(
    data_query, splitting_string,
    lo=0, up=train_percent)
eval_df = create_data_split_sample_df(
    data_query, splitting_string,
    lo=train_percent, up=train_percent + eval_percent)
test_df = create_data_split_sample_df(
    data_query, splitting_string,
    lo=train_percent + eval_percent, up=modulo_divisor)
# This helps us get approximately the record counts we want
print("There are {} examples in the train dataset.".format(len(train_df)))
print("There are {} examples in the validation dataset.".format(len(eval_df)))
print("There are {} examples in the test dataset.".format(len(test_df)))
```

```
There are 7733 examples in the train dataset.
There are 1037 examples in the validation dataset.
There are 561 examples in the test dataset.
```

Preprocess data using Pandas

We'll perform a few preprocessing steps to the data in our dataset. Let's add extra rows to simulate the lack of ultrasound. That is we'll duplicate some rows and make the `is_male` field be `Unknown` . Also, if there is more than child we'll change the `plurality` to `Multiple(2+)` . While we're at it, we'll also change the plurality column to be a string. We'll perform these operations below.

Let's start by examining the training dataset as is.

In [29]:
```
train_df.head()
```

Out[29]:

	weight_pounds	is_male	mother_age	plurality	gestation_weeks	hash_values
0	7.312733	True	23	1	40	2798049222917800056
1	6.812284	True	38	1	38	-1466514356607500060
2	7.749249	True	35	1	38	-6784884401981100070

	weight_pounds	is_male	mother_age	plurality	gestation_weeks	hash_values
3	7.561856	False	44	1	40	-6784884401981100070
4	8.598028	True	23	1	39	-6865817661748100017

Also, notice that there are some very important numeric fields that are missing in some rows (the count in Pandas doesn't count missing data)

In [30]:

```
train_df.describe()
```

Out[30]:

	weight_pounds	mother_age	plurality	gestation_weeks	hash_values
count	7733.000000	7733.000000	7733.000000	7733.000000	7.733000e+03
mean	7.264415	28.213371	1.035691	38.691064	-2.984870e+17
std	1.303220	6.134232	0.201568	2.531921	5.590715e+18
min	0.562179	13.000000	1.000000	18.000000	-9.210618e+18
25%	6.624891	23.000000	1.000000	38.000000	-6.781866e+18
50%	7.345803	28.000000	1.000000	39.000000	5.057323e+17
75%	8.062305	33.000000	1.000000	40.000000	4.896699e+18
max	11.563246	48.000000	4.000000	47.000000	9.203641e+18

It is always crucial to clean raw data before using in machine learning, so we have a preprocessing step. We'll define a `preprocess` function below. Note that the mother's age is an input to our model so users will have to provide the mother's age; otherwise, our service won't work. The features we use for our model were chosen because they are such good predictors and because they are easy enough to collect.

Lab Task #3: Preprocess the data in Pandas dataframe

In [34]:

```
    # TODO 3
    # TODO -- Your code here.
def preprocess(df):
# Filter out the data we will not use, or is not usable
    df = df[df.weight_pounds>0]
    df = df[df.mother_age>0]
    df = df[df.gestation_weeks>0]
    df = df[df.plurality>0]

    # Modify plurality field to be a string
    twins_etc = dict(zip([1,2,3,4,5],
                  ["Single(1)",
                   "Twins(2)",
                   "Triplets(3)",
                   "Quadruplets(4)",
                   "Quintuplets(5)"]))
    df["plurality"].replace(twins_etc, inplace=True)

    # Clone data and mask certain columns to simulate lack of ultrasound
    no_ultrasound = df.copy(deep=True)
```

```
    # Modify is_male
    no_ultrasound["is_male"] = "Unknown"

    # Modify plurality
    condition = no_ultrasound["plurality"] != "Single(1)"
    no_ultrasound.loc[condition, "plurality"] = "Multiple(2+)"

    # Concatenate both datasets together and shuffle
    return pd.concat(
        [df, no_ultrasound]).sample(frac=1).reset_index(drop=True)
```

Let's process the train, eval, test set and see a small sample of the training data after our preprocessing:

In [35]:

```
train_df = preprocess(train_df)
eval_df = preprocess(eval_df)
test_df = preprocess(test_df)
```

In [36]:

```
train_df.head()
```

Out[36]:

	weight_pounds	is_male	mother_age	plurality	gestation_weeks	hash_values
0	6.459544	False	34	Single(1)	37	-6784884401981100070
1	7.749249	Unknown	32	Single(1)	39	-6784884401981100070
2	7.705156	False	29	Single(1)	39	830080012870100058
3	8.564959	Unknown	31	Single(1)	40	766709067980000060
4	8.743533	Unknown	21	Single(1)	39	-4614303140002600076

In [37]:

```
train_df.tail()
```

Out[37]:

	weight_pounds	is_male	mother_age	plurality	gestation_weeks	hash_values
15461	8.430477	Unknown	28	Single(1)	38	-4614303140002600076
15462	6.499227	True	23	Single(1)	40	2620860165093800008
15463	8.664167	True	33	Single(1)	37	780565305641800050
15464	8.375361	Unknown	22	Single(1)	43	-3058524906279500017
15465	7.251004	True	27	Single(1)	39	-2875790318525700041

Let's look again at a summary of the dataset. Note that we only see numeric columns, so `plurality` does not show up.

In [38]:

```
train_df.describe()
```

Out[38]:

	weight_pounds	mother_age	gestation_weeks	hash_values
count	15466.000000	15466.000000	15466.000000	1.546600e+04
mean	7.264415	28.213371	38.691064	-2.984870e+17

	weight_pounds	mother_age	gestation_weeks	hash_values
std	1.303178	6.134034	2.531839	5.590534e+18
min	0.562179	13.000000	18.000000	-9.210618e+18
25%	6.624891	23.000000	38.000000	-6.781866e+18
50%	7.345803	28.000000	39.000000	5.057323e+17
75%	8.062305	33.000000	40.000000	4.896699e+18
max	11.563246	48.000000	47.000000	9.203641e+18

Write to .csv files

In the final versions, we want to read from files, not Pandas dataframes. So, we write the Pandas dataframes out as csv files. Using csv files gives us the advantage of shuffling during read. This is important for distributed training because some workers might be slower than others, and shuffling the data helps prevent the same data from being assigned to the slow workers.

In [39]:

```
# Define columns
columns = ["weight_pounds",
           "is_male",
           "mother_age",
           "plurality",
           "gestation_weeks"]

# Write out CSV files
train_df.to_csv(
    path_or_buf="train.csv", columns=columns, header=False, index=False)
eval_df.to_csv(
    path_or_buf="eval.csv", columns=columns, header=False, index=False)
test_df.to_csv(
    path_or_buf="test.csv", columns=columns, header=False, index=False)
```

In [40]:

```
%%bash
wc -l *.csv
```

```
  2074 eval.csv
  1122 test.csv
 15466 train.csv
 18662 total
```

In [41]:

```
%%bash
head *.csv
```

```
==> eval.csv <==
8.62448368944,False,33,Single(1),40
6.0009827716399995,Unknown,22,Single(1),37
4.31224184472,Unknown,39,Multiple(2+),34
9.68711179228,False,31,Single(1),41
4.2328754304,True,37,Single(1),32
8.99926953484,Unknown,38,Single(1),40
7.1870697412,Unknown,25,Single(1),39
7.4626475687,True,31,Single(1),40
```

```
8.811876612139999,Unknown,18,Single(1),38
5.24920645822,True,27,Single(1),35

==> test.csv <==
6.8122838958,Unknown,21,Single(1),39
7.87491199864,True,27,Single(1),41
7.50012615324,False,27,Single(1),36
7.68751907594,Unknown,27,Single(1),40
10.00016820432,Unknown,32,Single(1),40
6.1244416383599996,True,21,Single(1),38
6.1244416383599996,Unknown,18,Single(1),38
9.18666245754,Unknown,41,Single(1),38
4.5635688234,True,31,Twins(2),33
8.928721611,Unknown,23,Single(1),40

==> train.csv <==
6.4595442766,False,34,Single(1),37
7.7492485093,Unknown,32,Single(1),39
7.7051560569,False,29,Single(1),39
8.5649588787,Unknown,31,Single(1),40
8.74353331092,Unknown,21,Single(1),39
7.605948039,True,28,Single(1),38
3.2187490251999997,Unknown,20,Multiple(2+),35
6.1883756943399995,Unknown,14,Single(1),37
8.2232423726,Unknown,26,Single(1),37
6.1068046574,False,24,Single(1),39
```

In [42]:

```
%%bash
tail *.csv
```

```
==> eval.csv <==
7.936641432,Unknown,29,Single(1),38
7.936641432,Unknown,28,Single(1),41
8.75014717878,Unknown,17,Single(1),39
8.062304921339999,True,20,Single(1),40
4.850169764,Unknown,39,Single(1),39
5.1257475915,False,34,Single(1),35
8.12623897732,Unknown,24,Single(1),39
8.28717642858,Unknown,24,Single(1),36
7.62578964258,True,19,Single(1),38
8.313631900019999,False,33,Single(1),40

==> test.csv <==
8.062304921339999,True,21,Single(1),37
5.5005334369,True,27,Single(1),35
8.50102482272,Unknown,38,Single(1),39
8.68841774542,True,26,Single(1),39
4.7840310854,True,34,Twins(2),38
8.437090766739999,Unknown,26,Single(1),38
6.8673994613,False,27,Single(1),40
10.37495404972,Unknown,22,Single(1),40
7.06140625186,Unknown,23,Single(1),40
3.196702799,Unknown,19,Single(1),29

==> train.csv <==
7.06140625186,True,22,Single(1),40
7.68751907594,Unknown,21,Single(1),39
8.313631900019999,Unknown,24,Single(1),39
6.250105127699995,True,33,Single(1),41
7.87491199864,False,20,Single(1),39
```

```
8.43047689888,Unknown,28,Single(1),38
6.4992274837599995,True,23,Single(1),40
8.6641668966,True,33,Single(1),37
8.375361333379999,Unknown,22,Single(1),43
7.25100379718,True,27,Single(1),39
```

Lab Summary:

In this lab, we set up the environment, sampled the natality dataset to create train, eval, test splits, and preprocessed the data in a Pandas dataframe.

In [ ]: