VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

BIG DATA ANALYTICS
AND BUSINESS INTELLIGENCE
Semester 221

Build a model to predict asteroid size using Spark and Google Cloud Service

---

Teacher: Assoc. Prof. Nguyen Thanh Binh
Student: Le Tien Loc - 1952328
Phan Tuan Khai - 1952780
Nguyen Van Quoc Chuong - 1950004

HO CHI MINH CITY, DECEMBER 2022

Contents

1 Project Description

1.1 Requirement

This project has to meet the following requirements as what the teacher Thanh Binh said in class:

- Make use of big data tools/frameworks to process, visualize data
- Datasets could be any size but should be as large as possible

1.2 Our topic

In this project, our group will train a model to predict asteroids size using a dataset of asteroids' observations provided by NASA. (about 1.2 million records)

1.3 Big data tools

After several discussions, we decided to use Google Cloud Services, which is a storage for big data and Spark, a very popular framework for data analyst to process data. Moreover, Spark also provides us a lbrary named SparkML to build some basic ML models, hence, we will make us of it for building model in our project. The tools we used will be summarized in the following table:

Tools/Frameworks	Purposes
Google Cloud Storage	Store Data
SparkSQL	Data visualization and processing
SparkML	Build ML model and evaluation

Table 1: Big data tools

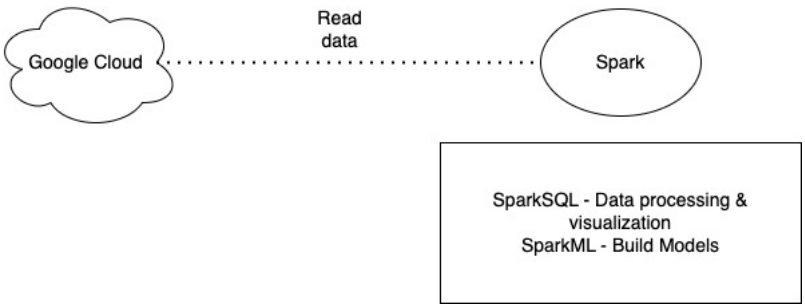

Figure 1

1.4 Notice

In this project, the path we use is absolute path which may not work in your workspace. To make the code run you have to make sure the path leading to correct required files. The secrets.JSON is the file for authentication so it will not be uploaded or the value will be changed. You can generate the secrets.JSON by yourself by creating a key for your Service Account in Google Cloud Service.

2 Google Cloud Service - Data preparation

2.1 Dataset

We use the datasets provided by NASA [NAS], which comprises of about 1.2 millions records with 31 fields in total. We think this datasets is large enough for us to practice working with big data tools/frameworks

The 31 fields information of dataset:

- a: semi-major axis (au)
- e: eccentricity
- i: inclination with respect to ecliptic plane
- om: longitude of the ascending node
- w: argument of perihelion
- q: perihelion distance (au)
- ad: aphelion distance (au)
- per_y: orbital period (years)
- data_arc: span of recorded data (days)
- condition_code: orbital condition code
- n_obs_used: number of observations used
- H: absolute magnitude parameter
- neo: near-earth object
- pha: physically hazardous object
- diameter: diameter (target variable)
- extent: Object bi/tri axial ellipsoid dimensions(Km)
- albedo: geometric albedo
- rot_per: rotation period (hours)
- GM: gravitational parameter. Product of mass and gravitational constant
- BV: Color index B-V magnitude difference
- UB: Color index U-B magnitude difference
- IR: Color index I-R magnitude difference
- spec_B: Spectral taxonomic type(SMASSII)
- spec_T: Spectral taxonomic type (Tholen)
- G: Magnitude slope parameter
- moid: Earth minimum orbit intersection distance

- class: asteroid orbit class
- n: mean motion (degrees/day)
- per: orbital period (days)
- ma: mean ananomly (degrees)

2.2 Uploading dataset to Google Cloud Storage

To use Google Cloud Service, we have to sign up with a credit card. Luckily, Google will give us $300 free credit for 3 months. After successful registry, we have to create a bucket (figure 2), which is a place to store data in Google Cloud Storage. We then create a folder and upload our dataset to that folder in bucket.

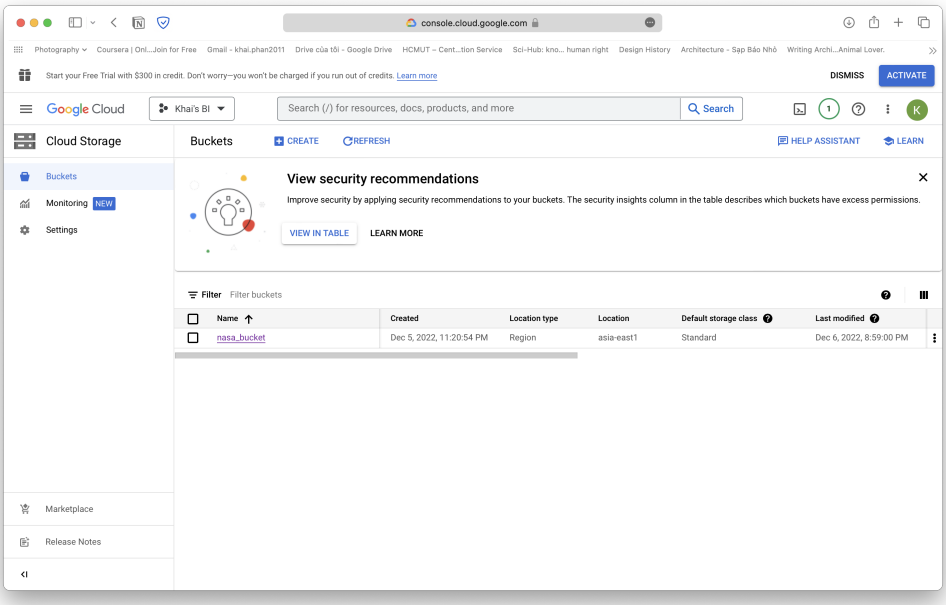

Figure 2: Created bucket

2.3 Create a Service Account to get access to the bucket

After creating a new bucket, we have to create a Service Account which will be use to get access to the data bucket.

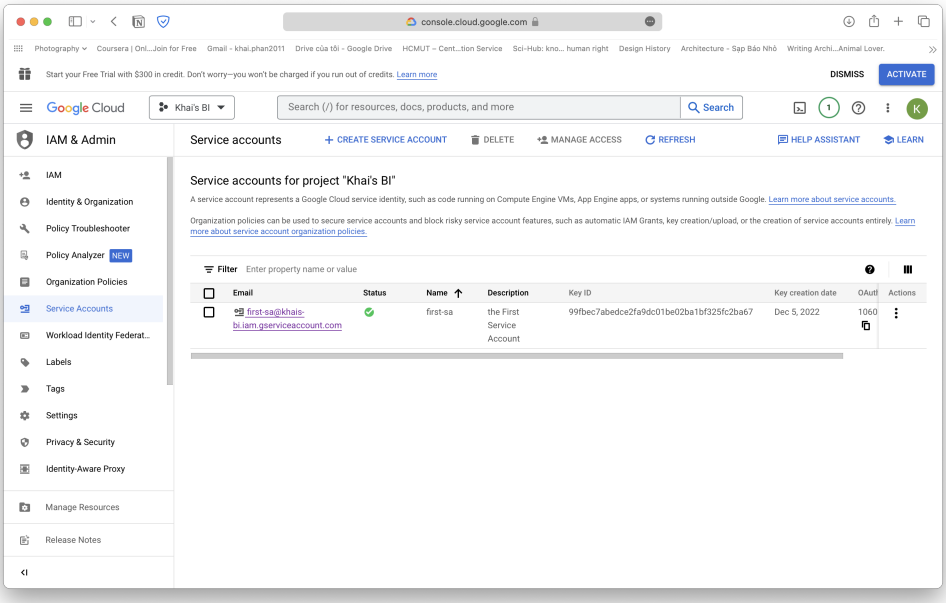

Figure 3: Service Account

With the created Service Account, we will create a key stored in a JSON file which will be renamed as secrets.JSON and we will not upload this file due to security. We will use this JSON file to for authentication and get access to the data bucket.

3 SparkSession - Google Cloud Storage connection

We now have the data in Google Cloud Storage but Spark have not had a connection to it yet. We have to configure the Spark Session so that it can connect to Google Cloud Storage. We will need a connector to connect Spark to Google Cloud Service which will be store in file gcs-connector-hadoop3-latest.jar. When we have the connector file, we have 2 main Spark configurations to make Spark work with Google Cloud.

- Setup hadoop fs configuration for schema gs://
- Setup hadoop fs configuration for Google Cloud authentication using secrets.JSON

```
#create a Spark Session with Google Cloud connector
spark = SparkSession \
    .builder \
    .appName("GCS_NASA") \
    .config("spark.jars", "drive/Shareddrives/Big_Data_and_BI/connector/
    gcs-connector-hadoop3-latest.jar") \
    .getOrCreate()

```

```
# Setup hadoop fs configuration for schema gs://
conf = spark.sparkContext._jsc.hadoopConfiguration()
conf.set("fs.gs.impl", "com.google.cloud.hadoop.fs.gcs.
    GoogleHadoopFileSystem")
conf.set("fs.AbstractFileSystem.gs.impl", "com.google.cloud.hadoop.fs.
    gcs.GoogleHadoopFS")

# Google Cloud Authentication
conf.set("fs.gs.auth.type", "SERVICE_ACCOUNT_JSON_KEYFILE")
conf.set("fs.gs.auth.service.account.json.keyfile","drive/Shareddrives/
    Big_Data_and_BI/key/secrets.json")
```

Now, we can read data from Google Cloud Storage bucket. If it works, the following code will not throw any error.

```
bucket_name="nasa_bucket"
path=f"gs://{bucket_name}/asteroid/Asteroid_Updated.csv"

df=spark.read.csv(path, sep=',', inferSchema=True, header=True)
```

4 SparkSQL - data visualization and processing

4.1 Data visualization

In this part, we will visualize data focusing on the asteroid diameter, we do not have much knowledge on this field so the way we make classes for diameter size may not practical in real life. This part is done after the data is cleaned by dropping na, which is on the Data processing part.

4.1.1 Bar Chart for counting number of asteroids base on its diameter

Firstly, I check for the range of diameter.

```
    astro_df.agg(min(col("diameter")), max(col("diameter"))).show()
```

And we got the following result, the range is from 0.0025 to 939.4

```
+-------------+-------------+
|min(diameter)|max(diameter)|
+-------------+-------------+
|       0.0025|        939.4|
+-------------+-------------+
```

Figure 4: Diameter Range

With the range above, we tried to divide diameter to 5 class (0,200), (200, 400), (400, 600), (600, 800), (800, 1000). However, we face the problem that the number of asteroid with diameter from 0 to 200 is considerable larger than the other classes. After several trial, we have specify another class as following.

```
count_list = []
label = []
for i in range (0, 10, 2):
  count_list += [astro_df.filter( (col("diameter") > (i)) & (col("
    diameter") < (i+ 2)) ).count()]
```

```
  label += [f"{i} - {i + 2}"]

for i in range (10, 50, 10):
  count_list += [astro_df.filter( (col("diameter") > (i)) & (col("
    diameter") < (i+ 10)) ).count()]
  label += [f"{i} - {i + 10}"]

for i in range (50, 200, 50):
  count_list += [astro_df.filter( (col("diameter") > (i)) & (col("
    diameter") < (i+ 50)) ).count()]
  label += [f"{i} - {i + 50}"]

for i in range (200, 1000, 400):
  count_list += [astro_df.filter( (col("diameter") > (i)) & (col("
    diameter") < (i+ 400)) ).count()]
  label += [f"{i} - {i + 400}"]

diameter_data = {'label': label, 'data': count_list}
diameter_df = pd.DataFrame(data=diameter_data)
diameter_df
```

With the data frame, we use Pandas to plot a bar chart

```
# set width of bar
barWidth = 0.5
fig = plt.subplots(figsize =(16, 8))

# Set position of bar on X axis
X_axis = np.arange(len(diameter_df["label"]))

# Make the plot
count = 0
color_list = ["#3981BF", "#38A649", "#F2AE30", "#F2594B", "#F2594C"]

plt.bar(X_axis + barWidth*(0.5+ count), diameter_df["data"], color =
    color_list[count], width = barWidth,
        edgecolor ='grey')

# Adding Xticks
plt.xlabel('Diameter', fontweight ='bold', fontsize = 15)
plt.ylabel('Number of asteroids', fontweight ='bold', fontsize = 15)
plt.xticks(X_axis + 0.5*barWidth,diameter_df["label"])

plt.legend()
plt.show()
```

And we got:

4.1.2 Bar Chart for average diameter by asteroid's classes

Firstly, we calculate the average diameter grouped by class column.

```
avrg_pd = astro_df.groupBy(col("class")).agg( avg(col("diameter")).alias
    ("average")).toPandas()
```

With the data frame, we use Pandas to plot a bar chart

```
# set width of bar
barWidth = 0.5
fig = plt.subplots(figsize =(16, 8))

# Set position of bar on X axis
X_axis = np.arange(len(avrg_pd["class"]))

# Make the plot
count = 0
```

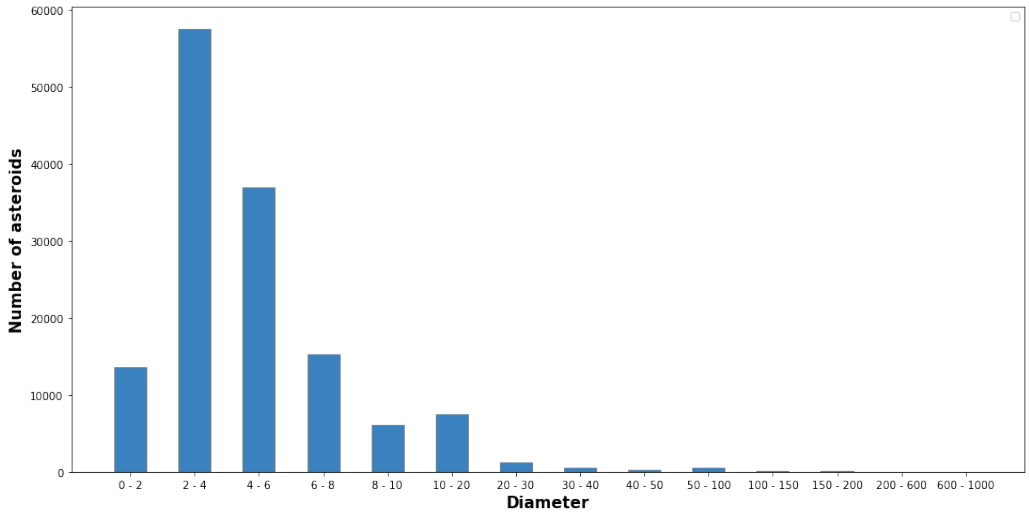

Figure 5: Bar Chart showing number of asteroids by Diameter ranges

```
10
11 plt.bar(X_axis + barWidth*(0.5+ count), avrg_pd["average"], color = "
       #3981BF", width = barWidth,
12         edgecolor ='grey')
13
14
15
16 # Adding Xticks
17 plt.xlabel('Class', fontweight ='bold', fontsize = 15)
18 plt.ylabel('Average Diameter', fontweight ='bold', fontsize = 15)
19 plt.xticks(X_axis + 0.5*barWidth,avrg_pd["class"])
20
21 plt.legend()
22 plt.show()
```

And we got:

From this chart, we know that the diameter of asteroid in TNO class is considerably larger than the others

4.2 Data processing

First of all, we need to handle the missing data by using dropna() function in pandas.

```
1
2 astro_df = df.dropna(subset=['diameter'])
3 drop_list = ["extent",
4             "rot_per",
5             "GM", "BV", "UB", "IR",
6             "spec_B",
7             "spec_T",
8             "G", 'data_arc', 'H', 'albedo']
9 astro_df = astro_df.drop(*drop_list)
```

Take note that we modified the True/False values in the 'neo' and 'pha' columns before converting them to boolean values and then casting them as integers. Simply put, this is binary encoding. These columns' True/False values are now shown as 1s

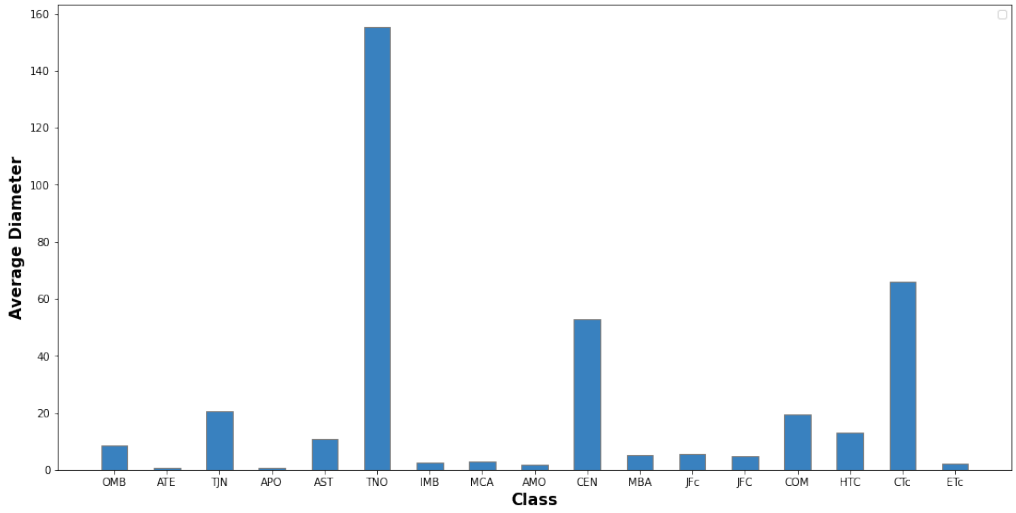

Figure 6: Bar Chart showing average diameter by Asteroids' Class

and 0s. Since many models in SparkML need 'double' datatypes as input, we also updated floating in this code to doubles.

```
1
2  astro_df = astro_df.withColumn('neo', regexp_replace('neo', 'Y', 'True')
       )
3  astro_df = astro_df.withColumn('neo', regexp_replace('neo', 'N', 'False'
       ))
4  astro_df = astro_df.withColumn('pha', regexp_replace('pha', 'Y', 'True')
       )
5  astro_df = astro_df.withColumn('pha', regexp_replace('pha', 'N', 'False'
       ))
6  astro_df = astro_df.withColumn('n_obs_used', astro_df['n_obs_used'].cast
       ('double'))
7  astro_df = astro_df.withColumn('diameter', astro_df['diameter'].cast('
       double'))
8
9  for column in ['neo', 'pha']:
10     astro_df = astro_df.withColumn(column, astro_df[column].cast('
       boolean').cast('int'))
11
12 astro_df = astro_df.dropna(subset=['diameter'])
```

Utilizing Spark's StringIndexer, OneHotEncoder, and Pipeline classes, we have to encode categorical variables, then, we can drop the original columns and re-order our dataframe.

```
1  condition_code_indexer = StringIndexer(inputCol="condition_code",
       outputCol="condition_codeIndex").setHandleInvalid("skip")
2  class_indexer = StringIndexer(inputCol="class", outputCol="classIndex").
       setHandleInvalid("skip")
3  onehotencoder_condition_code_vector = OneHotEncoder(inputCol="
       condition_codeIndex", outputCol="condition_code_vec")
4  onehotencoder_class_vector = OneHotEncoder(inputCol="classIndex",
       outputCol="class_vec")
5
6
7  encoding_pipeline = Pipeline(stages=[condition_code_indexer,
8                               class_indexer,
9                               onehotencoder_condition_code_vector,
```

```
                                onehotencoder_class_vector
                    ])

astro_df = encoding_pipeline.fit(astro_df).transform(astro_df)
```

```
astro_df = astro_df.drop('condition_code', 'class', 'condition_codeIndex
    ', 'classIndex')

astro_df = astro_df.select('full_name', "a","e","i",'om','w','q','ad', '
    per_y', 'n_obs_used', 'neo', 'pha', 'moid', 'n', 'per', 'ma', '
    condition_code_vec', 'class_vec', 'diameter')

features = astro_df.schema.names[1:-1]
```

After that, we split our data into training, validation, and testing sets. All of our features must be consolidated into a single vector as well. One of the stranger features of building models with Spark is this. To input all of the feature columns into a single vector for a Spark model, we need to use the VectorAssembler() class.

```
train, val, test = astro_df.randomSplit([0.6, 0.2, 0.2], seed=42)

train_df = train.drop('name')
val_df = val.drop('name')
test_df = test.drop('name')

assembler = VectorAssembler(inputCols=features, outputCol='features').
    setHandleInvalid("skip")

test_pack = assembler.transform(test_df)
train_pack = assembler.transform(train_df)
val_pack = assembler.transform(val_df)

for field in features:
    test_pack = test_pack.drop(field)
    train_pack = train_pack.drop(field)
    val_pack = val_pack.drop(field)
```

We use the MinMaxScaler to normalize our features for training section.

```
scaler = MinMaxScaler(inputCol="features", outputCol="scaledFeatures")
scaler_model = scaler.fit(train_pack)

train_pack = scaler_model.transform(train_pack)
val_pack = scaler_model.transform(val_pack)
test_pack = scaler_model.transform(test_pack)
```

5 SparkML - ML model and evaluation

Now that we're all ready for modelling, let's instantiate, fit, and make predictions with a few different regression methods available in SparkML:

```
gbt = GBTRegressor(featuresCol='features', labelCol='diameter', maxIter
    =100, maxDepth=5, seed=42, lossType='squared', stepSize=.1)

gbt_model = gbt.fit(train_pack)
gbt_pred = gbt_model.transform(val_pack)

rf = RandomForestRegressor(featuresCol='features', labelCol='diameter',
    maxDepth=5, seed=42, bootstrap=True, numTrees=100)

rf_model = rf.fit(train_pack)
rf_pred = rf_model.transform(val_pack)
```

```
lr = LinearRegression(featuresCol='features', labelCol='diameter',
    maxIter=100, loss='squaredError', elasticNetParam=0.5, regParam=0.1,
     fitIntercept=True, standardization=True, solver='auto', tol=.1)

lr_model = lr.fit(train_pack)
lr_pred = lr_model.transform(val_pack)
```

	rmse	r2
LR	13.708287	-1.376717
RF	6.683033	0.435117
GBT	6.590959	0.450575

Now let's evaluate the performance of these three models:

```
rmse = RegressionEvaluator(
    labelCol="diameter", predictionCol="prediction", metricName="rmse")

r2 = RegressionEvaluator(
    labelCol="diameter", predictionCol="prediction", metricName="r2")

metrics = [rmse, r2]
metric_labels = ['rmse', 'r2']

predictions = [lr_pred, rf_pred, gbt_pred]
predict_labels = ['LR', 'RF', 'GBT']

eval_list = list()

for pred in zip(predict_labels, predictions):
    name = pred[0]
    predict = pred[1]

    metric_vals = pd.Series(dict([(x[0], x[1].evaluate(predict))
                                  for x in zip(metric_labels, metrics)]),
                            name=name)
    eval_list.append(metric_vals)

eval_df = pd.concat(eval_list, axis=1).T
eval_df = eval_df[metric_labels]
eval_df
```

Based on the results, Gradient-Boosted Regression performed best with R2 score of .47, and a Root Mean Squared Error of 7.1 km, so let's try to optimize the model with GridSearch and cross-validation:

```
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator

# the GradientBoostedTree model
gbt2 = GBTRegressor(featuresCol='features',
                    labelCol='diameter',
                    predictionCol='prediction',
                    seed=13,
                    lossType='squared',
                    maxIter=100,
                    )

```

```
pipeline = Pipeline(stages=[gbt2])

paramgrid = (ParamGridBuilder().addGrid(gbt2.maxDepth, [2,4,6])
                               .addGrid(gbt2.stepSize, [0.0001, 0.001,
    0.01, 0.1, 1.0]).build())


evaluator = RegressionEvaluator(labelCol='diameter',
                                                predictionCol='prediction'
    ,
                                                metricName='r2')

crossval = CrossValidator(estimator=pipeline,
                          estimatorParamMaps=paramgrid,
                          evaluator=evaluator,
                          numFolds=3)

gbt2_model = crossval.fit(train_pack)
```

Now we need to extract our best model from the cross-validator. We can also see the feature importance of our model with the code below:

```

best_pipeline = gbt2_model.bestModel
best_gbt_model = best_pipeline.stages[0]

feature_importances = best_gbt_model.featureImportances.toArray()


feature_names = astro_df.columns[:-1]

feature_series = (pd.Series(dict(zip(features, feature_importances)))
                  .sort_values(ascending=True))

feature_series
```

```
class_vec            0.000000
condition_code_vec   0.000000
pha                  0.000085
w                    0.001119
neo                  0.001324
e                    0.002504
om                   0.005929
ma                   0.005932
moid                 0.007812
q                    0.030492
n                    0.031186
ad                   0.051679
per                  0.058555
per_y                0.069070
a                    0.074710
i                    0.089795
n_obs_used           0.481063
dtype: float64
```

Finally, let's make predictions on our test set using the optimized GBT model and see how it compares with our initial model results:

```

gbt_pred_test = best_pipeline.transform(test_pack)

predictions = [lr_pred, rf_pred, gbt_pred, gbt_pred_test]
predict_labels = ['LR', 'RF', 'GBT', 'GBT_GridSearch']

eval_list = []
```

```
for pred in zip(predict_labels, predictions):
    name = pred[0]
    predict = pred[1]

    metric_vals = pd.Series(dict([(x[0], x[1].evaluate(predict))
                                  for x in zip(metric_labels, metrics)]),
                            name=name)
    eval_list.append(metric_vals)


eval_df = pd.concat(eval_list, axis=1).T
eval_df = eval_df[metric_labels]
eval_df
```

	rmse	r2
LR	13.708287	-1.376717
RF	6.683033	0.435117
GBT	6.590959	0.450575
GBT_GridSearch	5.170815	0.482980

6 Summary

In this mini-project, I set out to provide a conceptual understanding of Spark and Google Cloud Storage, and a practical example of creating a machine learning model using both.

As the results of the model show, some additional work could be done on the data (hint: outliers), but the point was not to create an optimal model, merely to show the workflow for someone who might be starting out.

The way we approach working with large datasets is different than we might be used to if we're just making the transition from learning to practitioner. As we make that change, it's important that we start thinking about the optimal solution path for your specific problem — there are plenty of other ways we could have approached this task, such as sampling the data. Often, the tools you will have at your disposal will depend on the company you're working for, so it pays to be flexible, and have a working understanding of a wide range of tools

References

[NAS] NASA. *Orbital and/or physical parameters for asteroids and/or comets of interest from our small-body database (SBDB)*. URL: https://ssd.jpl.nasa.gov/tools/sbdb_query.html.