Configuration Notes:

In a project like this, where we utilize a lot of RAM on a device, it is essential to set up the right configuration. We have tried multiple configurations with a lot of them giving Java heap space memory error. The below setting has worked best for us yet. We set the driver memory to the maximum possible in our Colab notebook and set the garbage collectors as well.

```
  .config("spark.driver.memory", "15g") \
  .config("spark.driver.extraJavaOptions", "-XX:+UseG1GC") \
  .config("spark.executor.extraJavaOptions", "-XX:+UseG1GC")\
```

Notes on Data Engineering :

Dataset 1 :
Dataset 1 is the latest Medicare Part D Prescriber dataset which has all the information of the provider and the medicine prescription on individual drugs.

Number of columns in the dataset : 22
Columns in part D dataset : ['Prscrbr_NPI', 'Prscrbr_Last_Org_Name', 'Prscrbr_First_Name', 'Prscrbr_City', 'Prscrbr_State_Abrvtn', 'Prscrbr_State_FIPS', 'Prscrbr_Type', 'Prscrbr_Type_Src', 'Brnd_Name', 'Gnrc_Name', 'Tot_Clms', 'Tot_30day_Fills', 'Tot_Day_Suply', 'Tot_Drug_Cst', 'Tot_Benes', 'GE65_Sprsn_Flag', 'GE65_Tot_Clms', 'GE65_Tot_30day_Fills', 'GE65_Tot_Drug_Cst', 'GE65_Tot_Day_Suply', 'GE65_Bene_Sprsn_Flag', 'GE65_Tot_Benes']

We compute basic statistics on the above columns and find out that to figure out patterns of individual providers we need columns related to total claims, daily supplies, drug costs, number of beneficiaries, last 30 days behavior to find out anomalous patterns. We take all the relevant columns and make a new dataframe.
After doing so, we take the numeric columns and perform basic operations on them like max, mean, sum of values and add them as new columns to identify patterns in max-mean and sum variations.

We join the newly formed columns with the categorical columns like first name, last name, city, state to identify geographical patterns as well for a prescriber.

Dataset 2 :
Dataset 2 is an open payment dataset provided by the government which has more information on prescription payments in the country.
]
This massive dataset has 91 columns related to the payment which covers the exact location, form of payment, details on the hospital. We are only interested in numerical columns for now to understand payment information. Therefore we extract NPI number, total payment value for the NPI, total number of payment installations for those payments in this dataset.

We group these by NPI and join the dataset with dataset 1 to form a more informative dataset.

Dataset 3 :

Dataset 3 is an exclusion dataset that marks all payments and NPIs reported by the government. They are reported for different reasons and under different exclusion lists. For simplicity, we consider any NPI to appear on this dataset to be a fraudulent set.

Number of columns in the dataset : 18

Columns in part D dataset : ['LASTNAME', 'FIRSTNAME', 'MIDNAME', 'BUSNAME', 'GENERAL', 'SPECIALTY', 'UPIN', 'NPI', 'DOB', 'ADDRESS', 'CITY', 'STATE', 'ZIP', 'EXCLTYPE', 'EXCLDATE', 'REINDATE', 'WAIVERDATE', 'WVRSTATE']

We take NPI, 'EXCLTYPE' for our use case and make a new dataframe out of it. We mark all NPIs with non zero and valid 'EXCLTYPE' as a new column "LABEL " with value 1.

Final Dataset

We create the final dataset by joining dataset 3 with the results of joining dataset 1 and dataset 2. All the records with no label resulting in this dataframe is marked with LABEL 0 which means non fraudulent data.

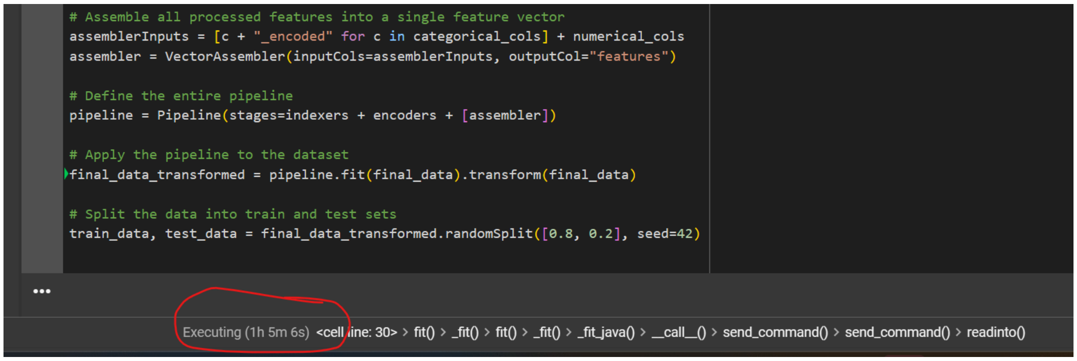
We take a subset of the entire dataset to work on classification problem for two reasons :

1. We are restricted to limited memory driver space due to repetitive Java Heap Space Error. We try training the model with the entire dataset but we face space error everytime.
2. Since we are working with a real fraudulent dataset, it is a highly skewed dataset with only about 11131 records that are fraudulent and all other records(more than 1 million) as normal data records. Therefore, we take all fraud datasets, take a subset of 200000 rows and mix them to form a somewhat balanced dataset.
3.

Modeling -

This part is a bit open ended. We experimented with 2 ways of doing this part.

1. One was with MLlib so that we could continue with using Pyspark df but running models on MLlib took a significantly high amount of time. Attached is a screenshot below which uses pyspark.ml.feature import VectorAssembler, StringIndexer, OneHotEncoder to make a pipeline



```
# Assemble all processed features into a single feature vector
assemblerInputs = [c + "_encoded" for c in categorical_cols] + numerical_cols
assembler = VectorAssembler(inputCols=assemblerInputs, outputCol="features")

# Define the entire pipeline
pipeline = Pipeline(stages=indexers + encoders + [assembler])

# Apply the pipeline to the dataset
final_data_transformed = pipeline.fit(final_data).transform(final_data)

# Split the data into train and test sets
train_data, test_data = final_data_transformed.randomSplit([0.8, 0.2], seed=42)
```

```
Executing (1h 5m 6s) <cell line: 30> > fit() > _fit() > fit() > _fit() > _fit_java() > __call__() > send_command() > send_command() > readinto()
```

The code above ran for more than an hour just to make the pipeline.

2. We tried experimenting with other tech stack. We converted the final data to pandas df which took only about 15 mins! Therefore, we proceeded to test our data using sklearn and pandas Dataframe.

The performance differences between Random Forest, Linear Regression, and Gradient Boosted Trees in a Medicare fraud detection task can be attributed to how each algorithm deals with the complexities and specific characteristics of the dataset.

Random Forest excels due to its ability to handle large and complex datasets with interdependent features, its robustness against overfitting, and its capacity to model non-linear relationships without needing any transformation of features. This makes it particularly effective for datasets with diverse and complex structures, such as those involving healthcare claims, where interactions between variables can be significant and not necessarily linear.

Linear Regression, while efficient and straightforward, often falls short in such tasks due to its assumption of linear relationships between variables. It struggles with outlier sensitivity and cannot capture the complexity of interactions between features as effectively as tree-based methods.

Gradient Boosted Trees are powerful for similar reasons as Random Forest, particularly in their ability to model non-linearities and interactions. However, they are generally more prone to overfitting and can be sensitive to noise and outliers, which might explain why they underperform compared to Random Forest in scenarios where data quality and overfitting control are crucial.

In summary, Random Forest's superiority in this context likely stems from its ensemble approach, providing a balance between accuracy and robustness, and its effectiveness in handling diverse features without extensive preprocessing. This makes it particularly suited for complex detection tasks like fraud analysis in Medicare data, where these capabilities directly address the challenges posed by the dataset's nature.