

Código Fonte

Neste ficheiro vou apresentar o código que utilizei ao longo do projeto, em conjunto com anotações que justificam a aplicação do código. O projeto foi trabalhado desde o Jupyter Notebook.

Carga e visualização dos dados

Importei a biblioteca *Pandas* e carreguei de dados na variável *raw_data* para ter como backup ante qualquer inconveniente. A seguir visualizei os primeiros registos com *head()*

```
# importo biblioteca Pandas
import pandas as pd

# carrego os meus dados num DataFrame (df)
raw_data = pd.read_csv(
    "/Users/anamariarodrigues/Desktop/BIG_DATA/Projeto_Final/Air_Traffic_Passenger_Statistics.csv")

raw_data.head()
```

O seguinte código foi aplicado para entender a estrutura e tipo de dados do dataframe e de algumas colunas em específico.

```
# dados estatísticos das colunas numéricas
raw_data.describe()
```

```
# informação geral do df
raw_data.info()
```

```
raw_data["GEO Summary"].unique()
```

```
raw_data["Price Category Code"].unique()
```

```
raw_data["Boarding Area"].unique()
```

Para conhecer os valores nulos nas colunas apliquei a primeira linha de código. Depois as restantes linhas foram aplicadas para o tratamento desses valores nulos. Filtrando com a metodologia de indexação de listas, definindo condições. E a seguir preenchi os valores da forma mais simples com o comando *fillna()*

```
# visualização valores nulos
raw_data.isnull().sum()
```

```
# visualizo os registos que tem valor nulo na coluna referida
raw_data[raw_data["Operating Airline IATA Code"].isnull()]
```

```
# valores NC existentes
raw_data[raw_data["Operating Airline IATA Code"] == "NC"]
```

```
# preencho valores nulos
df = raw_data.fillna("NC")
df.isnull().sum()
```

Para transformar a coluna 'Month' em valores numéricos, primeiro visualizo os valores únicos, e a seguir com o comando *replace()* posso definir quais são as transformações que quero aplicar em cada condição.

A seguir apaguei a coluna duplicada com comando *drop()* e mudei o nome da nova coluna para que o df fique igual ao original. o argumento *inplace = True* define que a transformação vai ser aplicada na mesma localização.

```
df["Month"].unique()

df["Month Number"] = df["Month"].replace({"January": 1, "February": 2,
                                           "March": 3, "April": 4,
                                           "May": 5, "June": 6, "July": 7,
                                           "August": 8, "September": 9,
                                           "October": 10, "November": 11,
                                           "December": 12
                                           })

df["Month Number"] = df["Month Number"].astype("int")

# Apago coluna duplicada
df.drop(columns = ["Month"], inplace = True)
df.rename(columns = {"Month Number" : "Month"}, inplace = True)

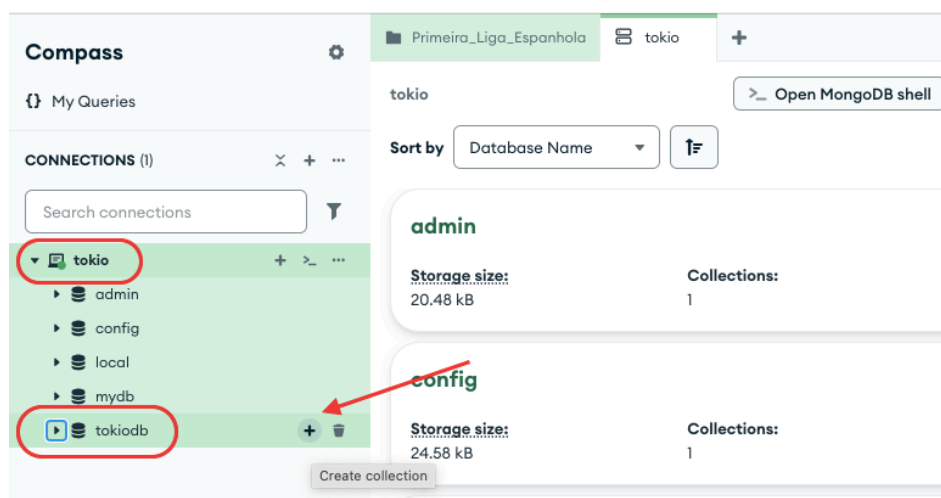
df["Month"].head()
```

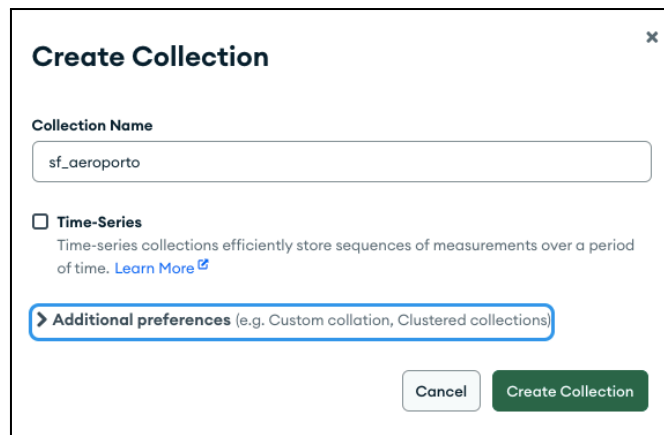
Criação de Collection no CosmosDB.

Para trabalhar com a Base de Dados *CosmosDB* é preciso ter uma coleção onde vão ser armazenados os documentos. Trabalhei com o aplicativo *CosmosDB Compass* porque é mais simples de utilizar, fazer consultas e visualizar os resultados.

Para criar a collection, primeiro faço uma conexão no aplicativo, e dentro da minha DB existentes com o nome “**tokiodb**”, cliquei no botão **+** e defino o nome da collection onde vou trabalhar, que coloquei o nome de “**sf_aeroporto**”.

A seguir colo imagens demonstrando os passos mencionados.





Inserção dos documentos na coleção

Para começar importei os módulos necessários das bibliotecas *json* e *pymongo* para fazer o processo de transformação e inserção de dados no MongoDB.

Com *MongoClient* crio a conexão com Mongo, e com *loads* carrego qualquer arquivo em formato JSON (formato de dicionário {key: value}), e permite a leitura e manipulação do objeto.

```
# importo bibliotecas
from pymongo import MongoClient
from json import loads
```

Primeiro transformo o meu DF Pandas para JSON com o comando *to_json()* definindo o parâmetro *orient = 'records'* para obter os resultados num formato de lista, onde cada objeto da lista, é um dicionário com os valores de cada registro.

A modo ilustrativo imprimo o 1º objeto da lista.

```
# transformo o DF para o formato JSON
df_json = df.to_json(orient = "records")
parsed = loads(df_json)
```

```
parsed[0]
```

```
{'Activity Period': 200507,
 'Operating Airline': 'ATA Airlines',
 'Operating Airline IATA Code': 'TZ',
 'Published Airline': 'ATA Airlines',
 'Published Airline IATA Code': 'TZ',
 'GEO Summary': 'Domestic',
 'GEO Region': 'US',
 'Activity Type Code': 'Deplaned',
 'Price Category Code': 'Low Fare',
 'Terminal': 'Terminal 1',
 'Boarding Area': 'B',
 'Passenger Count': 27271,
 'Adjusted Activity Type Code': 'Deplaned',
 'Adjusted Passenger Count': 27271,
 'Year': 2005,
 'Month': 7,
 '_id': ObjectId('67f8fe255435822a73f3ec8c')}
```

Tendo os dados no formato indicado para trabalhar, defini uma função, para ter um código mais simples e ágil.

Esta função vai trabalhar com quatro argumentos, o *URI* que representa o caminho de conexão a DB, o nome da DB, o nome da collection e os *docs* que vão ser os documentos a ser inseridos.

Na função;

client ⇒ faz a conexão através do endereço uri

database ⇒ defino com *get_database* onde estou a trabalhar

collection ⇒ defino a coleção com *get_collection*

insert_many ⇒ função que permite inserir mais de 1 documento

close ⇒ fecha a conexão com a DB

bloco try - except ⇒ os apliquei para testar se for possível correr os comandos designados, e no caso de não ser possível, devolver o erro como *Exception* e não gerar uma trava no programa. E quando o código correr bem devolva a mensagem de “sucesso”.

```
# função para inserir vários documentos
def insert_many(uri, db, collection, docs):
    client = MongoClient(uri)
    try:
        database = client.get_database(db)
        collection = database.get_collection(collection)
        collection.insert_many(docs)
        client.close()
        print("Dados inseridos com sucesso")
    except Exception as e:
        print(f"Erro durante a conexão: {e}")
```

Defino os parâmetros e corro a função para inserir os documentos.

```
# aplico função para inserir
uri = "mongodb://localhost:27017/"
db = "tokiodb"
collection = "sf_aeroporto"

insert_many(uri, db, collection, parsed)
```

Dados inseridos com sucesso

Para realizar consultas desde o aplicativo *MongoDB Compass* existe um campo para preencher com a pesquisa que quero e correr ele.

Como, por exemplo, na imagem seguinte apliquei um *find()* com o condicional *\$and* para devolver os valores que cumprirem as duas condições.

The screenshot shows the MongoDB Compass interface. At the top, the path 'tokio > tokiodb > sf_aeroporto' is displayed. Below the path, there are tabs for 'Documents', 'Aggregations', 'Schema', 'Indexes', and 'Validation'. The 'Documents' tab is selected, showing a query editor with a red box around the query:

```
{
  $and: [
    {
      "Operating Airline": "Air Berlin",
      "Boarding Area": "G"
    }
  ]
}
```

 To the right of the query editor are buttons for 'Generate query', 'Explain', 'Reset', 'Find', and 'Options'. Below the query editor, there are buttons for 'ADD DATA', 'EXPORT DATA', 'UPDATE', and 'DELETE'. The results are displayed in a table with columns: '_id ObjectId', 'Activity Period Int32', 'Operating Airline String', 'Operating Airline IATA Co...', and 'Pul'. The table contains 10 rows of data, all with 'Air Berlin' as the operating airline and 'AB' as the IATA code.

	_id ObjectId	Activity Period Int32	Operating Airline String	Operating Airline IATA Co...	Pul
1	ObjectId('67f8fe25543582...')	201005	"Air Berlin"	"AB"	"Ai"
2	ObjectId('67f8fe25543582...')	201005	"Air Berlin"	"AB"	"Ai"
3	ObjectId('67f8fe25543582...')	201006	"Air Berlin"	"AB"	"Ai"
4	ObjectId('67f8fe25543582...')	201006	"Air Berlin"	"AB"	"Ai"
5	ObjectId('67f8fe25543582...')	201007	"Air Berlin"	"AB"	"Ai"
6	ObjectId('67f8fe25543582...')	201007	"Air Berlin"	"AB"	"Ai"
7	ObjectId('67f8fe25543582...')	201008	"Air Berlin"	"AB"	"Ai"
8	ObjectId('67f8fe25543582...')	201008	"Air Berlin"	"AB"	"Ai"
9	ObjectId('67f8fe25543582...')	201009	"Air Berlin"	"AB"	"Ai"
10	ObjectId('67f8fe25543582...')	201009	"Air Berlin"	"AB"	"Ai"

Trabalhar com PySpark

Importei os módulos necessários para criar uma sessão de Spark, e poder trabalhar com os dados aplicando transformações e operações importantes para análises futura.

Criei o DF de Spark em base ao dataframe de Pandas.

```
# importação bibliotecas
from pyspark.sql import SparkSession
from pyspark.sql import functions as f

# crio sessão Spark
spark = SparkSession.builder.appName("Tokio_Airline").getOrCreate()

# crio df do spark, partindo do df criado de Pandas
df_sp = spark.createDataFrame(df)
```

Código para resolver ponto nº3 do projeto

O *select()* permitiu aplicar operações e só devolver essas operações, sem necessidade de visualizar o DF inteiro. Neste caso queria a contagem de valores únicos para conhecer quantas companhias diferentes existem dentro dos registos.

O comando *alias()* é muito conveniente, já que se não for aplicado o nome da coluna que devolve tem o título igual à operação aplicada *'count_distinct(...)'*.

```
# número de companhias existentes no df
df_sp.select(f.count_distinct("Operating Airline")\
            .alias("Companhias operacionais"),
            f.count_distinct("Published Airline")\
            .alias("Companhias comerciais"))\
            .show()
```

O seguinte código agrupa os valores pelas colunas definidas *groupBy()*, e com *agg()* cria uma coluna onde devolve o resultado de calcular o valor médio *avg()* e considerando que o resultado fala de pessoas, apliquei *round()* que tira os decimais do resultado, devolvendo o número inteiro mais próximo.

Busquei que os resultados da média foram ordenados de maior a menor, então apliquei *orderBy()* com a condição *ascending = False*.

PySpark permite salvar dados em diferentes formatos, e um deles é *.csv* através do comando *write()* onde posso definir o modo em que vai ser guardado, escrevendo sobre o ficheiro antigo *'overwrite'*. Também defino que os títulos das colunas fiquem com *header, true*. Só resta definir o nome do arquivo com o seu formato e pronto.

```
# média de passageiros por período e companhias
df_avg = df_sp.groupBy(["Activity Period", "Operating Airline"])\
            .agg(f.round(f.avg("Adjusted Passenger Count"))\
            .alias("Avg Passenger")\
            )\
            .orderBy("Activity Period", ascending = False)

df_avg.show(5)

# salvar resultados num ficheiro .csv
df_avg.write.mode("overwrite")\
            .option("header", "true")\
            .csv("media_passageiros.csv")
```

Para apagar registos duplicados tendo como referência uma coluna, existe o comando `dropDuplicates()` que trabalha fazendo um mapeio de cima para baixo, e os registos que se repetem os apaga, e deixa os primeiros que achou.

Por isso, primeiro ordenei os registos pelo número de passageiros, deixando os valores mais altos acima.

Depois, para visualizar se os duplicados foram apagados, fiz um `select()` da coluna em questão e observei que o processo foi bem-sucedido. A seguir salvei os resultados num `.csv` igual que no passo anterior.

```
# apagando duplicados pela região
geo_regiao = df_sp.orderBy("Adjusted Passenger Count", ascending = False)\
    .dropDuplicates(["GEO Region"])

geo_regiao.select("Operating Airline", "GEO Region", "Adjusted Passenger Count")\
    .orderBy("Adjusted Passenger Count", ascending = False)\
    .show()

geo_regiao.write.mode("overwrite")\
    .option("header", "true")\
    .csv("geo_regiao_unique.csv")
```

Análise descritiva

- Cálculos auxiliares para análise

Para realizar a análise descritiva apliquei diferentes operações e dependendo do objetivo fiz o agrupamento relevante para obter resultados significativos.

avg ⇒ calcula média.

stddev ⇒ devolve desvio padrão.

max - min ⇒ devolve valor máximo e valor mínimo respetivamente.

median ⇒ mede o valor na posição do médio de todos, previamente ordenados de menor a maior.

Aqui agrupei por companhia aérea, calculei media, desvio padrão, máximo e mínimo ordenados de maior a menor pela média de passageiros.

```
# média de passageiros agrupados por companhias
df_sp.groupBy("Operating Airline")\
    .agg(
        f.round(f.avg("Adjusted Passenger Count")).alias("Media de passageiros"),
        f.round(f.stddev("Adjusted Passenger Count")).alias("Desvio padrão"),
        f.max("Adjusted Passenger Count").alias("Max"),
        f.min("Adjusted Passenger Count").alias("Min")
    )\
    .orderBy("Media de passageiros", ascending = False)\
    .show(15)
```

No seguinte caso agrupei pela região do período e apliquei as mesmas operações, somando a mediana, e ordenando também de maior a menor pela média de passageiros.

```
# resultados agrupados por região
df_sp.groupBy("GEO Region")\
  .agg(
    f.round(f.avg("Adjusted Passenger Count")).alias("Media passageiros"),
    f.round(f.stddev("Adjusted Passenger Count")).alias("Desvio padrão"),
    f.max("Adjusted Passenger Count").alias("Max"),
    f.min("Adjusted Passenger Count").alias("Min"),
    f.median("Adjusted Passenger Count").alias("Mediana")
  )\
  .orderBy("Media passageiros", ascending = False)\
  .show()
```

Agora com *filter()* filtrei os valores onde a região contém 'US' e agrupei pelo mês para analisar sazonalidade. Calculei média, desvio padrão, máximo e mínimo, com uma ordem ascendente dos meses.

```
# cálculo de valores onde o destino é 'US'
df_sp.filter(df_sp["GEO Region"] == "US")\
  .groupBy("Month")\
  .agg(
    f.round(f.avg("Adjusted Passenger Count")).alias("Media passageiros"),
    f.round(f.stddev("Adjusted Passenger Count")).alias("Desvio padrão"),
    f.max("Adjusted Passenger Count").alias("Max"),
    f.min("Adjusted Passenger Count").alias("Min"),
  )\
  .orderBy("Month", ascending = True)\
  .show()
```

Fiz o agrupamento pelo tipo de voo e a seguir medi média, desvio padrão, máximo, mínimo e mediana de passageiros, com uma ordem descendente da média.

```
# análise descritiva agrupado por tipo de voo
df_sp.groupBy("GEO Summary")\
  .agg(
    f.round(f.avg("Adjusted Passenger Count")).alias("Media passageiros"),
    f.round(f.stddev("Adjusted Passenger Count")).alias("Desvio padrão"),
    f.max("Adjusted Passenger Count").alias("Max"),
    f.min("Adjusted Passenger Count").alias("Min"),
    f.median("Adjusted Passenger Count").alias("Mediana")
  )\
  .orderBy("Media passageiros", ascending = False)\
  .show()
```

O meu objetivo com o seguinte código foi criar um gráfico que auxilie a análise sobre o tipo de voos.

Achei interessante por ser um gráfico simples, utilizar a biblioteca Pandas para fazer o gráfico, então primeiro fiz o agrupamento e calculo de média com PySpark como foi anteriormente, com a diferença que apliquei o comando `toPandas()` para transformar o resultado num DF de Pandas.

Com o módulo `plot()` posso definir o tipo, e as configurações que vou querer no diagrama.

`kind = bar` ⇒ define o tipo de gráfico, este caso foi de barras.

`figsize` ⇒ tamanho da figura inteira.

`x - y` ⇒ defino parâmetros que utilizo para desenhar.

`xlabel - ylabel` ⇒ se quiser agregar uma descrição nos eixos x e y.

`legend` ⇒ informação adicional que posso somar ou não.

`title` ⇒ defino titulo da figura.

```
# gráfico auxiliar para compreensão no tipo de voos
avg_pass = df_sp.groupby("GEO Summary")\
    .agg(f.round(f.avg("Adjusted Passenger Count"))\
        .alias("Media Passageiros")).toPandas()

avg_pass.plot(kind = "bar",
              figsize = (5,4),
              x = "GEO Summary",
              xlabel = "Tipo de voo",
              y = "Media Passageiros",
              legend = False,
              title = "Media de passageiros por tipo de voo")
```

Criei uma tabela pivot para analisar o comportamento dos tipos de voos, agrupado por mês, já que o comando `pivot()` permite isso ao 'dividir' os dados sobre a variável designada, como se fosse aplicar um agrupamento com dois variáveis.

Mas para o meu objetivo de obter um gráfico derivado desses resultados, achei esta forma a mais apropriada.

No código agrupei por mês e criei o pivot baseado no tipo de voo, a seguir calculei a contagem de vezes que aparece cada tipo dentro de cada mês, ordenados por mês

```
# crio tabela pivot para dividir os dados pelo tipo de voo
pivot = df_sp.groupBy("Month")\
    .pivot("GEO Summary")\
    .agg(f.count("GEO Summary")\
        .alias("Contagem"))\
    .orderBy("Month")

pivot.show()
```

Por ser um gráfico onde quero demonstrar o padrão de duas variáveis escolhi trabalhar com a biblioteca `Matplotlib`. As configurações do gráfico são semelhantes ou iguais que Pandas, já que são Pandas tem certa 'integração' com Matplotlib.

Baseado na tabela pivot, criei duas listas diferenciadas pelo tipo de voo, para isso, o comando *value* devolve os valores de uma coluna, e com *toList()* transformo isso numa lista.

As diferenças entre Pandas e Matplotlib são que as configurações são feitas dentro de cada parâmetro, como a seguir.

plt.figure ⇒ funciona para limitar onde começa a figura que vou fazer, e também posso configurar parâmetros como o tamanho. Ou seja que a partir desta linha as configurações pertencem à figura.

plt.plot ⇒ define o tipo de gráfico, este caso é do tipo linha, configuro os valores, as cores, label ligado a esses valores.

plt.title ⇒ define o título da figura.

plt.legend ⇒ imprime os label de cada valor que defini antes.

plt.show ⇒ devolve a figura com todas as configurações que defini.

```
import matplotlib.pyplot as plt
# gráfico comparativo na frequência de voos
pivot_pd = pivot.toPandas()

y_values = pivot_pd["International"].values.tolist()
y_values2 = pivot_pd["Domestic"].values.tolist()

plt.figure(figsize = (6,4))
plt.plot(y_values, "red", label = "Internacionais")
plt.plot(y_values2, "blue", label = "Nacionais")
plt.title("Frequência de voos")
plt.legend()
plt.show()
```

● Análise da distribuição de dados

Com o código a seguir agrupei os dados pelo tipo de voo e calculei as frequências absoluta e relativa das variáveis, com o comando *count()* que devolve a contagem, e a seguir apliquei operações como multiplicação *** e divisão */* para calcular a frequência relativa. Sempre aplicando o comando *alias()* para ter encabeçados mais legíveis.

```
# distribuição de tipo de voos
dis_geo = df_sp.groupby("GEO Summary")\
    .agg(f.count("GEO Summary").alias("Freq Absoluta"),\
        f.round(f.count("GEO Summary") * 100 / 15007 ,2)\
        .alias("Freq Relativa"))

dis_geo.show()
```

O seguinte passo foi transformar esses resultados para Pandas e fazer o gráfico com o módulo *plot()*.

```
# grafico de barras
pandas_df = dis_geo.toPandas()
pandas_df.plot(kind = "bar",
                x= "GEO Summary",
                xlabel = "Tipo de voo",
                y= "Freq Relativa",
                figsize = (5,4),
                title = "Frequencia de tipo de voos",
                legend = False,
                ylim = (0, 75))
```

Agora filtrei os registos de voos internacionais, e agrupei pela região, agregando as frequências e dando uma ordem descendente em base a frequência absoluta.

```
dis_reg = df_sp.filter(df_sp["GEO Summary"] == "International")\
    .groupBy("GEO Region")\
    .agg(f.count("GEO Region").alias("Freq Absoluta"),\
        f.round(f.count("GEO Region") * 100 / 15007, 2)\
        .alias("Freq Relativa"))\
    .orderBy("Freq Absoluta", ascending = False)

dis_reg.show()
```

Para visualizar os resultados obtidos acima, num gráfico de barras, utilizei a biblioteca *Seaborn* por ser mais amigável para utilizar e também tem uma integração com *Pandas*. Tentei aplicar a biblioteca *Matplotlib* neste caso, mas devolvia sempre algum erro por falta de definição de parâmetros, enquanto com *Seaborn* foi só definir os eixos e a origem dos dados.

Fiz a transformação dos resultados para *Pandas* e configurei com *Matplotlib* os parâmetros 'externos' da figura igual que antes e o gráfico propriamente dito fiz com *sns.barplot()* onde defini os eixos e a fonte dos dados.

xticks ⇒ permite configurar os valores de referência do eixo X

```
import seaborn as sns
# gráfico da distribuição de regiões internacionais
reg_pd = dis_reg.toPandas()

plt.figure()
sns.barplot(x = "GEO Region", y = "Freq Absoluta", data = reg_pd)
plt.title("Distribuição voos Internacionais")
plt.ylabel("Frequência")
plt.xticks(rotation = 35, fontsize = "small")
plt.show()
```

Aplicação de matriz de correlação

Primeiro importe a biblioteca *StringIndexer* para transformar os valores do tipo string para o tipo numérico.

Para começar criei outro DF só com as variáveis que analisarei, para ser mais ordenado visualmente criei uma lista com os nomes das colunas a trabalhar, e a seguir criei o DF fazendo um *select()* com a lista criada. Fiz o print do esquema para ter certeza que foi bem aplicada a criação.

```
from pyspark.ml.feature import StringIndexer

#Defino colunas que vou utilizar na matriz de correlação
vars = ["Operating Airline", "Published Airline",
        "Activity Period", "GEO Summary", "GEO Region",
        "Activity Type Code", "Price Category Code",
        "Terminal", "Boarding Area", "Adjusted Passenger Count",
        "Year", "Month"]

# crio copia do df com as variáveis selecionadas
corr = df_sp.select(vars)
corr.printSchema()
```

Para aplicar a transformação do tipo de valores criei duas listas, uma com as colunas a transformar, e a outra com as colunas resultantes da transformação.

Criei o objeto *indexer* definindo as entradas e saídas, e logo treinei o modelo com *fit()*, passo seguinte é transformar o meu DF com o comando *transform()*.

```
input = ["Operating Airline", "Published Airline", "GEO Summary", "GEO Region",
        "Activity Type Code", "Price Category Code",
        "Terminal", "Boarding Area"]
output = ["operator_ind", "published_ind", "geo_summary_index", "region_index",
        "activity_index", "price_index",
        "terminal_index", "boarding_index"]

# defino o objeto StringIndexer e treino ele com os meus dados
indexer = StringIndexer(inputCols = input, outputCols = output)
si_model = indexer.fit(corr)

# transformo o df para valores numéricos
corr = si_model.transform(corr)
```

O que faz o código a seguir é imprimir uma lista por cada coluna transformada, com os valores que o modelo treina, na ordem de frequência que eles aparecem, de maior a menor. A função *labelsArray* do módulo *StringIndexer* permite conhecer qual número foi designado a cada valor, 0 para o mais frequente, 1 para o seguinte e assim vai.

```
# visualizo frequência de variáveis segundo StringIndexer
for i in si_model.labelsArray:
    print (i)
```

Para criar a matriz de correlação importei a biblioteca *Numpy*. Apaguei as colunas com valores originais, assim aplico a correlação em toda a matriz, e transformei para Pandas e poder trabalhar.

Para calcular a matriz esta a função *corrcoef()* onde dou os dados onde vai ser calculada a correlação (DF de pandas), e o parâmetro *rowvar = False* definindo que as variáveis estão divididas por colunas e não por filas.

```
import numpy as np
# apago colunas com valores em formato string
corr = corr.drop("Operating Airline", "Published Airline",
                 "Activity Period", "GEO Summary", "GEO Region",
                 "Activity Type Code", "Price Category Code",
                 "Terminal", "Boarding Area")

corr_pd = corr.toPandas()

# calculo matriz de correlação
matriz = np.corrcoef(corr_pd, rowvar = False)
```

Para visualizar e entender os resultados da matriz, fiz um mapa de calor com *Seaborn* onde as configurações da figura são aplicadas com *plt = Matplotlib*. Os parâmetros do gráfico são definidos dentro do *sns.heatmap()*

matriz ⇒ e a variável que representa os dados que vai utilizar

cmap ⇒ defino a cor do gráfico

annot ⇒ para os valores resultantes estarem ou não em cada quadrante

fmt ⇒ tamanho dos valores

linewidths ⇒ define tamanho da linha que separa cada quadrante.

```
# crio mapa de calor para visualizar correlações
plt.figure(figsize=(11,6))
sns.heatmap(matriz, cmap="coolwarm", annot = True, fmt = ".2f", linewidths = 0.5)
plt.title("Mapa calor de correlação entre variáveis")
plt.xlabel("Passageiros - Ano - Mês - Operação - Publicidade - Tipo - Região - Estado - Categoria - Terminal - Sala")
plt.show()
```

Algoritmo de Machine Learning

- **Pre-processamento dos dados**

O primeiro é importar as bibliotecas necessárias como *BisectingKMeans* é o algoritmo que vou aplicar, *ClusteringEvaluator* para ponderar os resultados do modelo e *VectorAssembler* para criar um vetor para o algoritmo poder treinar.

```
# importo as bibliotecas necessárias
from pyspark.ml.clustering import BisectingKMeans
from pyspark.ml.evaluation import ClusteringEvaluator
from pyspark.ml.feature import VectorAssembler
```

Criei e treinei o modelo *StringIndexer* igual que anteriormente, com os mesmos inputs e outputs, mas agora apliquei a transformação num novo DF *indexer_df*.

```
inputs = ["Operating Airline", "Published Airline", "GEO Summary", "GEO Region",
          "Activity Type Code", "Price Category Code",
          "Terminal", "Boarding Area"]
outputs = ["operator_ind", "published_ind ", "geo_summary_index", "region_index",
           "activity_index", "price_index",
           "terminal_index", "boarding_index"]

indexer = StringIndexer(inputCols = inputs, outputCols = outputs)
ind_modelo = indexer.fit(df_sp)

indexer_df = ind_modelo.transform(df_sp)
```

Criei o vetor partindo de uma lista de variáveis com as que quero que o modelo treine, e a seguir criei o objeto *assembler* onde forneço a lista como input, e a saída será o vetor, na coluna chamada *features*.

Logo aplico a transformação no DF com as colunas categorizadas por número, e crio *va_df* contendo as colunas numéricas, as colunas originais, e a coluna com o vetor.

```
features = ["Activity Period", "operator_ind", "published_ind ",
            "geo_summary_index", "region_index", "activity_index", "price_index",
            "terminal_index", "boarding_index", "Month", "Adjusted Passenger Count"]

assembler = VectorAssembler(inputCols = features, outputCol = "features")
va_df = assembler.transform(indexer_df)
```

- **Treino do modelo com voos internacionais**

Ao trabalhar com os voos internacionais, o primeiro é filtrar os dados

```
inter = va_df.filter(f.col("GEO Summary") == "International")
```

Para poder medir a eficácia do modelo criei o objeto *evaluator*.

predictionCol ⇒ coluna com valores resultantes do modelo

featuresCol ⇒ coluna com vetor de entrada do modelo

metricName ⇒ defino qual vai ser o cálculo pelo qual mede a eficácia

```
# objeto para calcular o Silhouette
evaluator = ClusteringEvaluator(predictionCol = "prediction",
                                featuresCol = "features",
                                metricName = "silhouette")
```

Como quero conhecer qual é o número de agrupamentos mais adequado para os meus dados, sem ser preciso escrever o mesmo código várias vezes, criei um *loop for* para iterar com diferentes números de K (agrupamentos).

No ciclo repetitivo vou definir as ações a fazer para obter os resultados do *evaluator* dependendo do número de agrupamentos.

for k in range (3,7) ⇒ quer dizer que vai iterar com k tendo valores de 3 até 6

bkm ⇒ crio o modelo e defino;

setK ⇒ defino numero de agrupamento desejado *k*

setSeed ⇒ para o modelo partir sempre do mesmo lugar

fit ⇒ treino do algoritmo com os dados filtrados

transform ⇒ aplica o modelo sobre os dados filtrados

evaluate ⇒ aplica o cálculo de eficácia sobre os resultados do modelo

print ⇒ imprimo os resultados para poder analisar

```
# loop para conhecer eficácia de diferentes K
for k in range(3, 7):
    bkm = BisectingKMeans().setK(k).setSeed(1)
    modelo = bkm.fit(inter)
    prediction = modelo.transform(inter)
    silhouette = evaluator.evaluate(prediction)
    print(f"K={k}, Silhouette Score={silhouette}")
```

Após conhecer a eficácia vou aplicar um algoritmo com 3 agrupamentos, aplicando os mesmos parâmetros que no loop for, mas definindo K = 3.

Logo a seguir imprimo os valores dos centroides para analisar os meus resultados. A função *clusterCentres()* devolve uma matriz com os centroides correspondente de cada variável.


```

bkm = BisectingKMeans().setK(3).setSeed(1)
modelo = bkm.fit(inter)
prediction_inter = modelo.transform(inter)
silhouette = evaluator.evaluate(prediction_inter)
print(f"K=3, Silhouette Score={silhouette}")

print("Cluster Centers: ")
centers = modelo.clusterCenters()
for center in centers:
    print(center)

```

Para uma melhor interpretação dos centroides calculei a média de todas as colunas que o algoritmo considerou, agrupando e ordenando por grupo.

```

# tabela com centroides dos clusters
summary_df = prediction_inter.groupby("prediction")\
    .agg(f.round(f.avg("Activity Period")).alias("Periodo"),
         f.round(f.avg("operator_ind"),2).alias("Operador"),
         f.round(f.avg("region_index"),2).alias("Regiao"),
         f.round(f.avg("activity_index"),2).alias("Estado"),
         f.round(f.avg("price_index")).alias("Categoria"),
         f.round(f.avg("terminal_index"),2).alias("Terminal"),
         f.round(f.avg("boarding_index"),2).alias("Embarque"),
         f.round(f.avg("Month")).alias("Mês"),
         f.round(f.avg("Adjusted Passenger Count")).alias("Media Passageiros"),
    ).orderBy("prediction").toPandas()

summary_df

```

Auxiliando a interpretação do funcionamento do modelo, criei um gráfico que representa as conexões dos grupos e as suas distâncias, conhecido como *Dendograma*. Consegui fazer com ajuda da biblioteca *SciPy* e o seu módulo *linkage* para definir as conexões entre os grupos, de forma simples *single*. E o módulo *dendrogram* para fazer o gráfico baseado nessas ligações.

Logo defino os parâmetros da figura inteira com *Matplotlib*.

```

# gráfico dendograma
from scipy.cluster.hierarchy import dendrogram, linkage

linkage = linkage(modelo.clusterCenters(), "single")

plt.figure(figsize =(5,3))
dendrogram(linkage)
plt.title("Agrupamento Hierárquico Bisecting KMeans")
plt.xlabel("Clusters")
plt.ylabel("Distância")
plt.show()

```


- **Exploração dos resultados**

O objetivo dos códigos a seguir, foi aplicar diferentes cálculos para conhecer distribuições, ou frequências dos agrupamentos. E obter o gráfico desses resultados para analisar o trabalho do modelo de ML.

Agrupei por agrupamento e fiz a contagem de registos dentro de cada um deles.

```
clusters = prediction_inter.groupBy("prediction")\
    .count()\
    .orderBy("prediction")\

clusters.show()
```

Transformei os resultados para Pandas e fiz um gráfico de barras com ajuda de *Matplotlib* e *Seaborn*.

```
df_pandas = clusters.toPandas()

plt.figure(figsize = (5,3))
sns.barplot(x= "prediction", y= "count", data = df_pandas)
plt.title("Tamanho dos Clusters")
plt.xlabel("Clusters")
plt.ylabel("Registos")
plt.show()
```

Medi a média e desvio padrão de passageiros dentro de cada agrupamento, e visualize os resultados.

```
prediction_inter.groupBy("prediction")\
    .agg(f.round(f.avg("Adjusted Passenger Count"))\
        .alias("média passageiros"),
        f.round(f.stddev("Adjusted Passenger Count"),2)\
        .alias("desvio"))\
    .orderBy("prediction").show()
```

Fiz a seleção das colunas *prediction* e *nº passageiros* para poder obter um gráfico do tipo *boxplot*, que precisa de todos os valores para devolver as amplitudes de cada agrupamento, baseado na coluna de passageiros.

Com *Seaborn* consegui fazer de uma forma simples, configurando alguns parâmetros;

x - y ⇒ defino eixos partindo das colunas

data ⇒ origem dos dados

hue ⇒ define que variável define a separação de cores

palette ⇒ paleta de cores que quero utilizar

`plt.grid` ⇒ habilito a grelha na figura

```
df_pandas = prediction_inter.select("prediction", "Adjusted Passenger Count").toPandas()

plt.figure(figsize = (10,5))
sns.boxplot(x = "prediction",
            y="Adjusted Passenger Count",
            data = df_pandas,
            hue = "prediction",
            palette = "Set2")
plt.xlabel("Agrupamentos")
plt.ylabel("Numero de Passageiros")
plt.title("Número de Passageiros por Agrupamento")
plt.grid(True)
plt.show()
```

Calculei a frequência de cada mês dentro dos agrupamentos, juntando os valores pelo grupo e por mês. Com uma ordem ascendente para obter resultados com uma lógica temporal.

```
sazonal = prediction_inter.groupBy("prediction", "Month")\
    .agg(f.count("Month").alias("Frequencia"))\
    .orderBy("prediction", "Month")
sazonal.show(5)
```

Transformei os resultados para Pandas e crio um gráfico de barras com Matplotlib e Seaborn.

```
df_pandas = sazonal.toPandas()

plt.figure(figsize = (6,5))
sns.barplot(x= "Month",
            y= "Frequencia",
            hue="prediction",
            palette = "Set2",
            data= df_pandas)
plt.title("Frequência de voos mensais")
plt.xlabel("Mês")
plt.ylabel("Frequência")
plt.legend(title = "Clusters")
plt.show()
```

Aqui dividi por grupo e região, calculando a frequência de cada região nos grupos

```
regiao = prediction_inter.groupBy("prediction", "GEO Region")\
    .agg(f.count("GEO Region").alias("Contagem"))\
    .orderBy("prediction")

regiao.show(5)
```

Igual que antes, transformei para Pandas e criei um gráfico de barras, neste caso defini na etiqueta do eixo X *xticks* que a rotação seja 35° e o tamanho da fonte *small* para conseguir uma melhor visualização.

```
df_pandas = regioao.toPandas()

plt.figure(figsize = (6,5))
sns.barplot(x= "GEO Region",
            y= "Contagem",
            data = df_pandas,
            hue="prediction",
            palette = "Set2")
plt.title("Regiões por cluster")
plt.xlabel("Região")
plt.xticks(rotation = 35, fontsize = "small")
plt.legend(title = "Cluster")
plt.show()
```