Enquanto lia o README do teste, fiquei na dúvida se a pipeline deveria ser executada sobre cada arquivo de dados separadamente ou aos dois ao mesmo tempo, então resolvi criar das duas maneiras.

Na root, vai ser encontrado 3 novas pastas e 1 script python, são elas:

pastas **AdultData** e **AdultTest:** irão conter os códigos necessários para rodar a pipeline para cada arquivo Adult.data e Adult.test, respectivamente.

pasta **Both**: possui um pipeline que executa os dois arquivos ao mesmo tempo.

script python ***get_mapping.py*** vai, baseando-se nas bases de dados, retornar uma string contendo a estrutura de dicionário para a qual será usado no Label Encoder, suas atribuições estão em *./*/Common/constants.py*.

as três pastas irão conter uma pasta chamada **Common**, e os arquivos *counter.txt*, *pipeline.sh* e *script.py*, além do database *DBClick.db*.

A lógica por trás da resolução da pipeline em cada pasta são análogas (com leves alterações que serão pontuadas). Primeiramente, a base de dados é lida usando a função *pd.read_csv()* do pandas. Eu não queria apagar os dados já ingeridos das bases, por isso, para evitar a ingestão de dados duplicados foi criado o *counter.txt*, ele será responsável por guardar o índice do próximo registro que será lido. O parâmetro *skiprows* da *read_csv()* irá fazer esse slice, retornando False para os registros que não devem ser pulados. Os dados das bases não são apagados, e se for adicionados mais linhas nas bases, desde que elas continuem a seguir o layout, a pipeline irá ingerir essas novas linhas.

Após a leitura, é buscado qual dicionário de mapeamento será utilizado. Depois, base e dicionário é passado para a função *transform()*, a qual irá realizar o mapeamento em cada uma das colunas pré-determinadas, para isso, é utilizado o Label Encoder do scikit-learn.

Por último, caso a transformação não retorne nenhum erro e o conjunto de dados não seja vazio o dataframe será ingerido no SQLite. Caso uma dessas condições aconteçam, mensagens customizadas serão geradas e o *counter.txt* não será atualizado. O shell script *pipeline.sh* irá executar a pipeline a cada dez segundos, para isso, basta executar: ´sh pipeline.sh´.

O *script.py* seria a pipeline em si (é o que o shell script chama), contendo a função *main()* que chama todas as funções auxiliares necessárias. Além da estrutura *if __name__ == __main__:* que irá chamar a função principal, iniciando a pipeline, além de informar o tempo de execução e a quantidade de linhas processadas e ingeridas por segundo em cada execução. Em **Both**, há mais uma função *get_dict_mapping()*, que basicamente irá retornar qual dicionário deve ser utilizado.

As pastas **Both**, **AdultData** e **AdultTest** possui a seguinte pasta:

pasta **Common**: contém 5 python scripts com atribuições e funções que serão usadas ao longo da pipeline. São eles:

constants.py: duas listas, *encode_col* e *all_col*, as quais contêm todas as colunas que irão passar pelo label encoder e todas as colunas da base, respectivamente. Um dicionário de mapeamento, *dict_map*, criado no *get_mapping.py*, em **Both**, há o *dict_map_data* e *dict_map_test*, para Adult.data e Adult.test, respectivamente. E um valor numérico, *row_to_map*, para definir quantas linhas vão ser processadas, em **AdultData** e **AdultTest** será 1630, em **Both** será 815.

counter.py: duas funções, *read()* e *write()*, elas irão ler e escrever o índice desejado no *counter.txt*. O índice inicial para o Adult.data é 0 e para o Adult.test é 1. Em **Both**, *counter.txt* possui duas linhas, a primeira referente ao Adult.data e a segunda ao Adult.test, as funções em *counter.py* são modificadas para lidar com essa linha extra.

label_encoder.py: três funções, *label_encode*, *convert_type* e *transform*.

A primeira receberá como parâmetro o dataframe, a coluna que sofrerá a ação e o dicionário de mapeamento. O dicionário é uma constante, então não importa qual a execução, ele sempre será o mesmo, por isso ele é usado para treinar a função *LabelEncoder()*, pois assim, os valores vão estar sempre na mesma ordem, ou seja, sempre terão o mesmo valor para representá-lo. Caso queira adicionar mais possíveis valores para um determinado campo, basta acrescentar o valor na lista referente a esse campo no dicionário de mapeamento.

A segunda irá apenas converter a tipagem dos dados para a ingestão no SQLite, garantir que as colunas transformadas sejam guardadas como *category* e não como *int*.

A última é a principal, irá executar, para cada coluna pré-determinada (*constants.encode_col*), a transformação; caso algum novo valor foi adicionado na base mas não no dicionário, esse erro é tratado e uma mensagem customizadas gerada.

load_data: uma função, *extract_data()*, que irá retornar o conjunto de dados que será processado. Para retornar apenas as linhas que queremos processar, é lido o índice inicial (contido no *counter.txt*) e o último índice (seria a soma do inicial e da *row_to_map*), e a partir deles, é criado um *range(first_idx, last_idx)*. Utilizando da seguinte forma o parâmetro *skiprows* da *read_csv()*: *skiprows = lambda x: x not in range()*, apenas as linhas que estiverem nesse intervalo não serão puladas.

sqlite.py: duas funções, *load()* e *read()*, a primeira irá carregar os dados processados no SQLite. Para ingerir, está sendo usando a função *pd.dataframe.to_sql()*, a tipagem e a nomenclatura das colunas do dataframe devem ser condizentes com os apresentados no bd. A segunda irá realizar uma query no database.