Stone Case

Cleyton Farias

September, 2023

Pré-requisitos

Para acessar os scripts e a base de dados utilizados neste exercício, você pode fazer o download do repositório ou abrir um terminal e inserir o seguinte comando:

```
git clone https://github.com/cleytonfar/fraud_detection_case.git
```

Este repositório contém scripts escritos em Python. Para configurar todas asdependências, é essencial que você utilize a ferramenta de gerenciamento de ambiente virtual do Python chamada `pipenv`.

Após clonar este repositório, acesse o diretório correspondente. Haverá um arquivo chamado `Pipfile.lock` especificando todas as dependências utilizadas. Abra um terminal e execute o seguinte comando:

```
pipenv sync
```

Este comando irá instalar todos os pacotes e suas respectivas versões necessárias para executar os scripts.

O repositório conta com os seguintes arquivos:

- `simulate_data.py`: script que irá gerar os dados utilizados neste exercício;
- `baseline_estimation.py`: script que contendo toda a modelagem estatística;
- `api.py`: script para transformar o modelo em API;
- `utils.py`: script com funções auxiliares;
- `data/`: diretório com os dados;
- `output/`: diretório com modelo e dicionário utilizados no API;

Credit Card Fraud Detection

Este exercício consiste no desenvolvimento de um sistema simples para detectar fraudes em transações de cartão de crédito, utilizando dados simulados.

Por meio da aplicação de algoritmos de *machine learning* e uma estratégia de estimação que considera o aspecto temporal do problema, foi possível criar uma abordagem sólida e confiável para estimar o desempenho dos algoritmos e prever casos futuros.

Por fim, foi explorado como esse sistema pode ser implementado em produção, transformando-o em uma RESTful API.

Conteúdo

- Data
  - Data Understanding
  - Feature Engineering
- Data Preparation
- Modelling
- Evaluation
- Deployment

Data

Dados públicos sobre transações reais de cartões de crédito são bastante raros. Existe apenas uma base de dados disponível na plataforma kaggle disponibilizada pelo Machine Learning Group. Apesar de suas limitações em representar situações do mundo real, essa base tem sido amplamente utilizada por pesquisadores, profissionais e entusiastas da modelagem estatística.

Com o objetivo de disponibilizar ainda mais informações sobre esse tema, o Machine Learning Group desenvolveu um simulador de dados de transações de cartões de crédito. Esse simulador permite a criação de dados sintéticos que incorporam características autênticas desse tipo de informação.

Em particular, o simulador[1] cria um conjunto de dados que inclui classes (transações fraudulentas ou não) desbalanceadas, variáveis numéricas e categóricas (inclusive características categóricas com um grande número de valores possíveis). Além disso, as variáveis são configuradas de modo a possuir uma característica de *time-dependent*.

Para esse exercício foi gerado um amostra de transações de cartão de crédito para 5000 clientes, em 10000 terminais, durante 100 dias, iniciado na data de `2023-06-01`.

```
from utils import generate_dataset, add_frauds

# simulate data for:
# 5000 clientes
# 10000 terminais
# 100 dias de transação.
customer_profiles_table, terminal_profiles_table, transactions_df=generate_dataset(
    n_customers = 5000,
    n_terminals = 10000,
    nb_days=100,
    start_date="2023-06-01",
    r=5
)

# Generation of fraud scenarios
```

[1] Para a replicação dos dados simulados, há um script chamado `simulate_data.py` com os passos necessários. O script `utils.py` contem as funções auxiliares necessárias para a execução da simulação. Para mais detalhes teóricos, acesse esse link.

```
transactions_df = add_frauds(customer_profiles_table, terminal_profiles_table, transactions_df)
transactions_df.head(5)
```

```
   TRANSACTION_ID        TX_DATETIME CUSTOMER_ID TERMINAL_ID  TX_AMOUNT  \
0               0 2023-06-01 00:00:31         596        3156      57.16
1               1 2023-06-01 00:02:10        4961        3412      81.51
2               2 2023-06-01 00:07:56           2        1365     146.00
3               3 2023-06-01 00:09:29        4128        8737      64.49
4               4 2023-06-01 00:10:34         927        9906      50.99

  TX_TIME_SECONDS TX_TIME_DAYS  TX_FRAUD  TX_FRAUD_SCENARIO
0              31            0         0                  0
1             130            0         0                  0
2             476            0         0                  0
3             569            0         0                  0
4             634            0         0                  0
```

Data Understanding

A base de dados contém 959,229 transações de cartão de crédito, durante o período `2023-08-01` até `2023-09-08`, com as seguintes variáveis[2]:

- `TRANSACTION_ID`: identificador para cada transação;
- `TX_DATETIME`: data e horário da transação;
- `CUSTOMER_ID`: identificador do cliente/usuário do cartão de crédito;
- `TERMINAL_ID`: identificador do terminal onde ocorreu a transação;
- `TX_AMOUNT`: valor da transação;
- `TX_FRAUD`: variável binária com valor 1 se a transação é fraudulenta; 0 caso conctrário.

Neste exercício, será desenvolvido um sistema simples de detecção de fraudes utilizando algoritmos de *machine learning*.

Uma das etapas iniciais envolve a análise da distribuição da variável dependente, `TX_FRAUD`. O trecho de código a seguir demonstra como conduzir essa análise, fornecendo um contexto importante para entendermos a abordagem que estamos adotando.

```
transactions_df["TX_FRAUD"].mean()
```

```
0.00793762490500183
```

Conforme podemos notar, o conjunto de dados exibe uma distribuição significativamente desequilibrada em relação aos casos de fraudes. Apenas 0,08% das transações em nossa amostra são rotuladas como transações fraudulentas. Essa característica é bastante representativa de cenários do mundo real, onde as fraudes geralmente são eventos raros e excepcionais em relação às transações legítimas.

[2] As variáveis `TX_TIME_SECONDS`, `TX_TIME_DAYS` são variáveis extraídas a partir da variável de `TX_DATETIME` e `TX_FRAUD_SCENARIO` é uma variável *by-product* do processo de simulação. Elas não serão importantes para o exercício.

Feature Engineering

Ao examinarmos as variáveis disponíveis, percebemos que temos à nossa disposição apenas quatro delas (`TX_DATETIME`, `TX_AMOUNT`, `CUSTOMER_ID`, `TERMINAL_ID`) para construir um sistema de detecção de fraudes. É essencial investigar como essas variáveis estão relacionadas ao indicador de fraude, a fim de compreender o impacto delas na identificação de transações fraudulentas.

```
transactions_df.groupby("TX_FRAUD")["TX_AMOUNT"].mean()
```

```
TX_FRAUD
0     52.978073
1    133.035867
Name: TX_AMOUNT, dtype: float64
```

Como podemos observar, as transações rotuladas como fraudes apresentam um valor de transação médio aproximadamente 2,5 vezes maior do que as transações legítimas. Essa diferença é significativa, sugerindo que a variável `TX_AMOUNT` pode desempenhar um papel crucial na previsão de fraudes.

Uma característica que pode estar relacionada a transações suspeitas é o horário em que a transação ocorre. É plausível supor que transações fraudulentas ocorram mais frequentemente durante a noite e nos finais de semana. Para investigar essa hipótese, podemos criar variáveis que capturem essas características nos dados, permitindo-nos avaliar sua influência na detecção de fraudes:

```
# - flag se transação ocorre durante o fim de semana ou nao:
transactions_df["TX_DURING_WEEKEND"] = (transactions_df["TX_DATETIME"].dt.weekday >= 5).astype(

# - flag se transação ocorre a noite:
# definição de noite: 22 <= hour <= 6
transactions_df["TX_DURING_NIGHT"] = ((transactions_df["TX_DATETIME"].dt.hour <= 6) | (transact
```

```
transactions_df.groupby("TX_FRAUD")[["TX_DURING_WEEKEND", "TX_DURING_NIGHT"]].mean()
```

TX_FRAUD	TX_DURING_WEEKEND	TX_DURING_NIGHT
0	0.278996	0.194978
1	0.279879	0.198976

Os resultados indicam que esses indicadores não demonstram uma correlação substancial com a probabilidade de uma transação ser considerada uma fraude, com uma diferença de apenas um pouco mais de 0,1 ponto percentual entre os grupos. No entanto, é importante notar que essas características ainda podem ser valiosas, pois é razoável supor que os padrões de fraudes possam variar entre dias úteis e fins de semana, bem como entre o período diurno e noturno. Portanto, embora a correlação direta seja modesta, essas variáveis ainda podem desempenhar um papel significativo na detecção de fraudes, capturando nuances temporais que podem ser críticas para o processo de identificação.

Outro aspecto crucial em problemas de detecção de fraudes reside na análise dos padrões de gastos de cada cliente. Cada consumidor possui um histórico de consumo único, e qualquer desvio significativo desse padrão pode indicar um potencial risco de fraude. Para capturar e caracterizar esses padrões individuais de consumo, será criado variáveis que representam o número de transações e a média de gastos do cliente ao longo de três janelas de tempo distintas: o dia anterior, a semana anterior e o mês anterior. Essas variáveis nos permitirão avaliar o comportamento do cliente em relação ao seu próprio histórico e identificar variações que possam indicar atividades suspeitas. Para isso será utilizado a função auxiliar `feature_engineering_customer_spending_behavior` para a criação dessas variáveis:

```
def feature_engineering_customer_spending_behaviour(customer_transactions, windows_size_in_days
    # ordenando de acordo com data
    customer_transactions=customer_transactions.sort_values('TX_DATETIME')

    # colocando data como index:
    # isso ajudará para utilizar as funcionalidades de operacoes de 'moving average':
    customer_transactions.index=customer_transactions.TX_DATETIME

    # loop sobre cada janela de tempo (em dias)
    for window_size in windows_size_in_days:
        # calculo da soma de transações:
        SUM_AMOUNT_TX_WINDOW=customer_transactions['TX_AMOUNT'].rolling(str(window_size)+'d').s
        # Calculo do número:
        NB_TX_WINDOW=customer_transactions['TX_AMOUNT'].rolling(str(window_size)+'d').count()
        # calculo da média:
        AVG_AMOUNT_TX_WINDOW=SUM_AMOUNT_TX_WINDOW/NB_TX_WINDOW
        # Save feature values
        customer_transactions['CUSTOMER_ID_NB_TX_'+str(window_size)+'DAY_WINDOW']=list(NB_TX_WI
        customer_transactions['CUSTOMER_ID_AVG_AMOUNT_'+str(window_size)+'DAY_WINDOW']=list(AVG

    # set transaction como index:
    customer_transactions.index=customer_transactions.TRANSACTION_ID

    return customer_transactions
```

```
# calculando as variáveis que caracterizam o padrao do cliente:
transactions_df = transactions_df.groupby("CUSTOMER_ID").apply(lambda x: feature_engineering_cu
transactions_df.reset_index(drop=True, inplace=True)
transactions_df.sort_values(by="TX_DATETIME", inplace=True)
transactions_df
```

	TRANSACTION_ID	TX_DATETIME	CUSTOMER_ID	TERMINAL_ID	TX_AMOUNT	TX
112991	0	2023-06-01 00:00:31	596	3156	57.16	31
951176	1	2023-06-01 00:02:10	4961	3412	81.51	130
534	2	2023-06-01 00:07:56	2	1365	146.00	476
789906	3	2023-06-01 00:09:29	4128	8737	64.49	569
177280	4	2023-06-01 00:10:34	927	9906	50.99	634

	TRANSACTION_ID	TX_DATETIME	CUSTOMER_ID	TERMINAL_ID	TX_AMOUNT	TX
...	...	...	...	...	...	...
707910	959224	2023-09-08 23:53:37	3698	6046	40.14	863
684872	959225	2023-09-08 23:53:39	3565	9889	13.42	863
647282	959226	2023-09-08 23:57:39	3373	5963	67.22	863
35655	959227	2023-09-08 23:58:24	187	9772	47.29	863
657060	959228	2023-09-08 23:59:24	3422	9951	35.60	863

Vamos checar como essas variáveis estão relacionadas ao risco de fraude:

```
transactions_df.groupby("TX_FRAUD")[['CUSTOMER_ID_NB_TX_1DAY_WINDOW',
                'CUSTOMER_ID_AVG_AMOUNT_1DAY_WINDOW',
                'CUSTOMER_ID_NB_TX_7DAY_WINDOW',
                'CUSTOMER_ID_AVG_AMOUNT_7DAY_WINDOW',
                'CUSTOMER_ID_NB_TX_30DAY_WINDOW',
                'CUSTOMER_ID_AVG_AMOUNT_30DAY_WINDOW']].mean()
```

TX_FRAUD	CUSTOMER_ID_NB_TX_1DAY_WINDOW	CUSTOMER_ID_AVG_AMOUNT_1DAY_WI
0	3.555907	53.257462
1	3.546231	99.325549

Conforme podemos ver, as variáveis que descrevem o padrão de gastos dos clientes exibem uma relação significativa. Isso sugere que essas variáveis estão relacionadas de forma relevante e podem ser utilizadas como indicadores importantes para a previsão de transações fraudulentas.

Além disso, podemos derivar um conjunto adicional de variáveis que caracterizam os terminais de pagamento. Seguindo a mesma lógica utilizada anteriormente com os clientes, o objetivo aqui é calcular um índice de risco que avalie a vulnerabilidade de um determinado terminal a transações fraudulentas. Esse índice de risco será calculado como a média das ocorrências de fraudes registradas em um terminal durante três intervalos de tempo distintos.

Contudo, diferentemente das variáveis criadas baseadas no comportamento de gasto do cliente, as janelas de tempo para os terminais não estarão diretamente antes de uma transação específica. Em vez disso, elas serão ajustadas para trás por um período de *delay*. O motivo disso é que em um cenário real, as transações fraudulentas só são descobertas após uma investigação sobre o caso. Essa descoberta leva um certo tempo até acontecer. Portanto, as transações fraudulentas que são usadas para calcular o índice de risco, só estarão disponíveis após esse período de *delay*. Por exemplo, para calcular o índice de risco de um terminal em uma data t para uma janela dos últimos 7 dias, precisamos olhar para os últimos 7 dias de transações com a tag de fraude conhecida a partir de t. Para um período de delay com 7 dias, então a índice seria calculado entre `[t-14; t-7]`. Para uma aproximação inicial, esse período de *delay* será definido como de 1 semana (d = 7) ao longo desse exercício. As janelas serão as mesmas utilizadas anteriormente (1, 7, 30):

```
# definindo uma função auxiliar para calcular as variáveis de risco em diferentes janelas:
def feature_engineering_terminal_risk(terminal_transactions,
                                      delay_period=7,
                                      windows_size_in_days=[1,7,30],
                                      feature="TERMINAL_ID"):
    # ordering por data
    terminal_transactions=terminal_transactions.sort_values('TX_DATETIME')

    # set index to date:
    terminal_transactions.index=terminal_transactions.TX_DATETIME

    # calculando o número de fraudes no periodo de delay
    NB_FRAUD_DELAY=terminal_transactions['TX_FRAUD'].rolling(str(delay_period)+'d').sum()
    # calculando o número de total de transacoes no periodo de delay
    NB_TX_DELAY=terminal_transactions['TX_FRAUD'].rolling(str(delay_period)+'d').count()

    # calculando o número no periodo de (delay + windows_size)
    for window_size in windows_size_in_days:
        # calculando o número de fraudes
        NB_FRAUD_DELAY_WINDOW=terminal_transactions['TX_FRAUD'].rolling(str(delay_period+windo
        # calculando o número de total de transacoes
        NB_TX_DELAY_WINDOW=terminal_transactions['TX_FRAUD'].rolling(str(delay_period+window_si
        # difenca entre (delay+windows_size) - delay
        NB_FRAUD_WINDOW=NB_FRAUD_DELAY_WINDOW-NB_FRAUD_DELAY
        NB_TX_WINDOW=NB_TX_DELAY_WINDOW-NB_TX_DELAY
        # calculando o risco:
        RISK_WINDOW=NB_FRAUD_WINDOW/NB_TX_WINDOW

        terminal_transactions[feature+'_NB_TX_'+str(window_size)+'DAY_WINDOW']=list(NB_TX_WINDO
        terminal_transactions[feature+'_RISK_'+str(window_size)+'DAY_WINDOW']=list(RISK_WINDOW)

    terminal_transactions.index=terminal_transactions.TRANSACTION_ID

    # Replace NA values with 0 :
    terminal_transactions.fillna(0,inplace=True)

    return terminal_transactions
```

```
transactions_df=transactions_df.groupby('TERMINAL_ID').apply(lambda x: feature_engineering_term
transactions_df.sort_values('TX_DATETIME', inplace=True)
transactions_df.reset_index(drop=True, inplace=True)
transactions_df
```

	TRANSACTION_ID	TX_DATETIME	CUSTOMER_ID	TERMINAL_ID	TX_AMOUNT	TX
0	0	2023-06-01 00:00:31	596	3156	57.16	31
1	1	2023-06-01 00:02:10	4961	3412	81.51	130

	TRANSACTION_ID	TX_DATETIME	CUSTOMER_ID	TERMINAL_ID	TX_AMOUNT	TX
2	2	2023-06-01 00:07:56	2	1365	146.00	476
3	3	2023-06-01 00:09:29	4128	8737	64.49	569
4	4	2023-06-01 00:10:34	927	9906	50.99	634
...	...	...	...	...	...	...
959224	959224	2023-09-08 23:53:37	3698	6046	40.14	863
959225	959225	2023-09-08 23:53:39	3565	9889	13.42	863
959226	959226	2023-09-08 23:57:39	3373	5963	67.22	863
959227	959227	2023-09-08 23:58:24	187	9772	47.29	863
959228	959228	2023-09-08 23:59:24	3422	9951	35.60	863

Vamos checar como essas variáveis estão relacionadas ao risco de fraude:

```
transactions_df.groupby("TX_FRAUD")[[ 'TERMINAL_ID_NB_TX_1DAY_WINDOW',
                'TERMINAL_ID_RISK_1DAY_WINDOW',
                'TERMINAL_ID_NB_TX_7DAY_WINDOW',
                'TERMINAL_ID_RISK_7DAY_WINDOW',
                'TERMINAL_ID_NB_TX_30DAY_WINDOW',
                'TERMINAL_ID_RISK_30DAY_WINDOW']].mean()
```

TX_FRAUD	TERMINAL_ID_NB_TX_1DAY_WINDOW	TERMINAL_ID_RISK_1DAY_WINDOW	TE
0	0.926344	0.002493	6.26
1	0.973076	0.264802	6.76

As estatísticas mostram que as transações fraudulentas podem possuir uma relaçao com as variáveis que representam o nível de risco de cada terminal de pagamento. Essa relação sugere que as características dos terminais podem desempenhar um papel significativo na identificação de transações suspeitas.

Data Preparation

Agora que possuímos a base de dados de transações, analisamos a distribuição da nossa variável de interesse e derivamos novas características a partir das informações disponíveis, podemos avançar para a etapa de preparação dos dados necessários para treinar nosso sistema de detecção de fraudes.

Primeira coisa que precisamos definir é nossa base de treinamento e base de teste. Como mencionado anteriormente, os dados exibem uma relação de dependência temporal entre as variáveis, o que requer uma consideração cuidadosa na estratégia de treinamento. Em muitas situações, problemas que envolvem dependência temporal são resolvidos sem essa devida atenção, o que pode levar a problemas de *data leakage* no processo de treinamento de algoritmos de *machine learning*. Portanto, é essencial garantir que a estratégia de estimação escolhida leve em consideração essa temporalidade para obter resultados robustos e confiáveis.

Para evitar esse problema, optei por usar as transações ocorridas no período de `2023-08-11` a `2023-08-17` como conjunto de treinamento e as transações entre `2023-08-25` e `2023-08-31` como conjunto de teste.

Em outras palavras, foi selecionado uma semana de transações para treinamento e outra semana de transações para teste.

É importante destacar um aspecto crucial nessa estratégia escolhida: a base de teste está situada uma semana após a última transação da base de treinamento. Esse intervalo de tempo que separa o conjunto de treinamento do conjunto de teste é conhecido como *delay*. Esse período reflete o fato de que, em um cenário real de detecção de fraudes, a classificação de uma transação (fraudulenta ou não) só é determinada após uma investigação de fraude, o que acarreta esse atraso temporal.

Outro ponto crucial em cenários de detecção de fraudes diz respeito à exclusão de cartões fraudulentos da base de treinamento durante o período de teste. Devido à natureza temporal da separação entre a base de treinamento e a de teste, cartões que já foram identificados com transações fraudulentas no conjunto de treinamento serão removidos da base de teste. Além disso, quaisquer cartões que se tornem conhecidos como fraudulentos durante o período de atraso *delay* também serão excluídos da base de teste. Isso assegura que o modelo seja avaliado de forma justa, refletindo a realidade de que a detecção de fraudes em transações já conhecidas como fraudulentas é geralmente trivial e não representa um desafio real.

Para efetuar essa divisão temporal dos conjuntos de treinamento e teste, bem como realizar a exclusão das transações fraudulentas identificadas na base de treinamento e durante o período de atraso da base de teste, foi criado uma função auxiliar denominada `get_train_test_set`.

```
def get_train_test_set(transactions_df,
                       start_date_training,
                       delta_train=7,delta_delay=7,delta_test=7):

    # training set data
    train_df = transactions_df[(transactions_df.TX_DATETIME>=start_date_training) &
                               (transactions_df.TX_DATETIME<start_date_training+datetime.timede

    # test set data
    test_df = []

    # Note: cartoes fraudulentos da base de treino serão removidos da base de test.
    # também, cartões que vierem a ser descobertos como fraudulentos ao longo
    # do período de delay, tbm serão excluidos da base de test.

    # pegando os cartoes fraudulentos do training set:
    known_defrauded_customers = set(train_df[train_df.TX_FRAUD==1].CUSTOMER_ID)

    # pegando a data relativa:
    start_tx_time_days_training = train_df.TX_TIME_DAYS.min()

    # Then, for each day of the test set
    for day in range(delta_test):

        # Get test data for that day
        test_df_day = transactions_df[transactions_df.TX_TIME_DAYS==start_tx_time_days_training
                                                                    delta_train+delta_delay+
```

```
                                                                    day]

        # Os cartões comprometidos do dia de teste, subtraindo o período de esperadelay, são ac
        test_df_day_delay_period = transactions_df[transactions_df.TX_TIME_DAYS==start_tx_time_
                                                                                delta_train+
                                                                                day-1]

        new_defrauded_customers = set(test_df_day_delay_period[test_df_day_delay_period.TX_FRAU
        known_defrauded_customers = known_defrauded_customers.union(new_defrauded_customers)

        test_df_day = test_df_day[~test_df_day.CUSTOMER_ID.isin(known_defrauded_customers)]

        test_df.append(test_df_day)

    test_df = pd.concat(test_df)

    # Sort data sets by ascending order of transaction ID
    train_df=train_df.sort_values('TRANSACTION_ID')
    test_df=test_df.sort_values('TRANSACTION_ID')

    return (train_df, test_df)
```

Como dito anteriormente, nosso conjunto de treinamento será composto por transações ocorridas de 2023-08-11 a 2023-08-17, e as transações entre 2023-08-25 e 2023-08-31 irá compor a base de teste:

```
import datetime
# setting start date:
start_date = datetime.datetime.strptime("2023-08-11", "%Y-%m-%d")
#print("Start date: {start_date}")

# separating train and test set sequentially with a delay period:
train_df, test_df=get_train_test_set(
    transactions_df,
    start_date_training=start_date,
    delta_train=7,
    delta_delay=7,
    delta_test=7
)
```

```
train_df.shape[0]
```

67180

```
test_df.shape[0]
```

58330

O training set consiste em 67,180 observações e o test set com 58,330 observações.

```
output_feature="TX_FRAUD"
input_features=['TX_AMOUNT',
                'TX_DURING_WEEKEND', 'TX_DURING_NIGHT',
                'CUSTOMER_ID_NB_TX_1DAY_WINDOW',
                'CUSTOMER_ID_AVG_AMOUNT_1DAY_WINDOW',
                'CUSTOMER_ID_NB_TX_7DAY_WINDOW',
                'CUSTOMER_ID_AVG_AMOUNT_7DAY_WINDOW',
                'CUSTOMER_ID_NB_TX_30DAY_WINDOW',
                'CUSTOMER_ID_AVG_AMOUNT_30DAY_WINDOW',
                'TERMINAL_ID_NB_TX_1DAY_WINDOW',
                'TERMINAL_ID_RISK_1DAY_WINDOW',
                'TERMINAL_ID_NB_TX_7DAY_WINDOW',
                'TERMINAL_ID_RISK_7DAY_WINDOW',
                'TERMINAL_ID_NB_TX_30DAY_WINDOW',
                'TERMINAL_ID_RISK_30DAY_WINDOW']

y_train = train_df[output_feature]
X_train = train_df[input_features]

y_test = test_df[output_feature]
X_test = test_df[input_features]
```

Modelling

A detecção de fraudes é comumente abordada como um problema de classificação binária, em que um sistema de detecção de fraudes recebe transações e tem como objetivo determinar se essas transações são provavelmente legítimas ou fraudulentas.

Um aspecto intrínseco a problemas de detecção de fraude é a presença de um desequilíbrio significativo na distribuição das classes, em que a proporção de casos de fraude é consideravelmente menor em relação às transações legítimas. Durante o processo de modelagem, é de extrema importância levar em conta esse desequilíbrio para garantir uma abordagem adequada à detecção de fraudes.

Frequentemente, em problemas de classificação binária, a métrica de acurácia é amplamente empregada para avaliar o desempenho dos algoritmos. No entanto, essa métrica, apesar de sua interpretação simples, não se mostra apropriada quando se lida com problemas que apresentam um desequilíbrio significativo na distribuição das classes da variável dependente. Em tais cenários, outras medidas de avaliação são mais adequadas.

O F1-score é uma métrica particularmente útil para abordar problemas de desequilíbrio em problemas de classificação, como a detecção de fraudes. Sua utilidade deriva do fato de que o F1-score leva em consideração tanto a precisão (precision) quanto o recall, tornando-se assim uma métrica equilibrada e sensível para situações em que existe uma desproporção significativa entre as classes.

O G-mean, ou média geométrica, surge como outra métrica relevante na avaliação de modelos em cenários com classes desbalanceadas. Essa métrica é definida como a raiz quadrada da multiplicação das taxas de verdadeiros negativos (TNR) e verdadeiros positivos (TPR). O TNR mede a proporção de casos

corretamente identificados como a classe majoritária, enquanto o TPR representa a proporção de casos corretamente identificados como a classe minoritária.

A *AUC-ROC* mede a capacidade do modelo de classificar corretamente instâncias positivas em relação às negativas em diveros thresholds de classificação. Isso é especialmente útil em problemas desbalanceados, onde a classe majoritária (transações legítimas) pode dominar a métrica de acurácia.

Por fim, também será utilizado a *média de precisão (Average Precision)*. Essa métrica é a area sobre a curva Precision-Recall e é uma métrica valiosa para avaliar o desempenho de modelos em cenários de classes desbalanceadas, como a detecção de fraudes.

Para começar com o processo de modelagem, será criado um *dummy classifier* e usá-lo como uma referência inicial (*baseline*) para os modelos subsequentes. Esse classificador adotará a média de `TX_FRAUD` na base de treinamento como previsão para determinar se uma transação é suspeita ou não. Essa classificador será avaliado utilizando as métricas mencionadas anteriormente.

```
from sklearn.metrics import accuracy_score, roc_auc_score
from sklearn.metrics import f1_score, precision_score, recall_score
from sklearn.metrics import average_precision_score
from imblearn.metrics import geometric_mean_score

# dummy model:
acc_dummy = accuracy_score(y_test, [y_train.mean() > .5]*len(y_test))
f1_dummy = f1_score(y_test, [y_train.mean() > .5]*len(y_test))
avg_prec_dummy = average_precision_score(y_test,[y_train.mean()]*len(y_test))
gmean_dummy = geometric_mean_score(y_test,[y_train.mean() > .5]*len(y_test))
auc_dummy = roc_auc_score(y_test, [y_train.mean()]*len(y_test))

# convert to DF
dummy_res = pd.DataFrame(
    {"clf": ["dummy"],
     "acc": [acc_dummy],
     "f1_score": [f1_dummy],
     "gmean": [gmean_dummy],
     "auc": [auc_dummy],
     "average_precision": [avg_prec_dummy]}
)

# creating a empty list:
res = []

# append first result:
res.append(dummy_res)
```

A partir deste ponto, vamos realizar a estimativa de seis algoritmos de *machine learning* e compará-los entre si, bem como com o *dummy classifier*, a fim de avaliar como nossas previsões estão melhorando. Os algoritmos que serão estimados são os seguintes:

- Logistic Regression;
- *K-nearest neighbors*

- Decision Tree;
- Bagging of Decision trees;
- Random Forest;
- Gradient Boosting;

Todos os algoritmos serão estimados com configuração *default*. O objetivo aqui é avaliar diferentes modelos de forma rápida e checar se é possível aprender algo com os dados.

Como estratégia de pré-processamento, será utilizado um processo de imputação e de *scaling* nas variáveis explicativas. Essas etapas serão agrupadas em um `Pipeline`. O uso de *pipelines* permite uma organização mais estruturada e legível do fluxo de trabalho de modelagem, além de prevenir a contaminação do processo de treinamento com informações do conjunto de teste, evitando assim problemas como *data leakage*.

```
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, MinMaxScaler

# candidates:
clfs = {
    "knn": KNeighborsClassifier(),
    "logistic": LogisticRegression(max_iter=2000),
    "tree": DecisionTreeClassifier(),
    "bagging": BaggingClassifier(estimator=DecisionTreeClassifier()),
    "rf": RandomForestClassifier(),
    "GradientBoost": GradientBoostingClassifier()
}

for name, clf in clfs.items():
    # instantiate a pipeline:
    pipe = Pipeline(steps=[
        ("imputer", SimpleImputer()),
        ("scaling", StandardScaler()),
        ("clf", clf)
    ])
    # fitting to training data:
    pipe.fit(X_train, y_train)
    # predicting on test set:
    y_pred = pipe.predict_proba(X_test)
    # Evaluating:
    # threshold-based metrics:
    ## accuracy
    acc = accuracy_score(y_test, y_pred[:,1]>.5)
```

```
    ## f1-score
    f1 = f1_score(y_test, y_pred[:,1]>.5)
    ## g-mean
    g_mean = geometric_mean_score(y_test, y_pred[:,1]>.5)
    # threshold-free metrics:
    auc = roc_auc_score(y_test, y_pred[:,1])
    # average precision
    avg_prec = average_precision_score(y_test, y_pred[:, 1])
    # organizing:
    eval = pd.DataFrame(
        {"clf": [name],
         "acc": [acc],
         "f1_score": [f1],
         "gmean": [g_mean],
         "auc": [auc],
         "average_precision": [avg_prec]
         }
    )
    res.append(eval)
```

```
# concatenate results
res = pd.concat(res)
res = res.sort_values("average_precision", ascending=True).reset_index(drop=True)
print(res)
```

```
             clf       acc  f1_score     gmean       auc  average_precision
0          dummy  0.992628  0.000000  0.000000  0.500000           0.007372
1           tree  0.994205  0.617647  0.795550  0.815879           0.384461
2            knn  0.995903  0.631741  0.690383  0.792873           0.532729
3  GradientBoost  0.996417  0.724638  0.799336  0.866950           0.592378
4       logistic  0.995868  0.637594  0.702017  0.875138           0.627553
5        bagging  0.997120  0.766017  0.799619  0.853360           0.674439
6             rf  0.996777  0.728324  0.765471  0.876376           0.698653
```

A tabela anterior apresenta o desempenho dos algoritmos na base de teste. Conforme observado, o *dummy classifier* apresenta o desempenho mais fraco entre todos os classificadores, enquanto o random forest alcança o melhor desempenho em termos de *average precision*.

É fundamental ressaltar que a acurácia não é uma métrica adequada em cenários com uma distribuição desequilibrada da variável `Y`. O dummy classifier atingiu uma taxa de acurácia de 99.26%, um resultado que, em outras circunstâncias, poderia ser considerado excepcional. No entanto, ao analisar as outras métricas, fica evidente que esse mesmo classificador demonstra um desempenho notadamente inferior em comparação com outros modelos.

Improvement

Uma limitação da última estratégia de estimação (train-test split) é que estamos obtendo apenas uma única medida de desempenho para cada algoritmo. Apesar de ser uma abordagem valiosa, é crucial contar uma estratégia de estimação que nos permite quantificar a incerteza em relação ao desempenho do algoritmo.

Uma maneira de obter estimativas mais precisas é adotar uma estratégia de *cross validation*(CV). CNo entanto, o problema de detecção de fraudes apresenta um componente temporal que precisa ser levado em consideração. Portanto, será necessário adaptar a estratégia de CV para lidar com essa temporalidade.

Uma forma de fazer isso é adaptar a estratégia de timeSeriesSplit às características do problema de detecção de fraud (delay period). Em outras palavras, iremos aplicar a mesma função de separação de train-test set (`get_train_test_data`) tal que as bases de treino e teste sempre são deslocadas por um bloco de tempo.

A função `sequential_train_test_split` irá realizar esse separação. Essa função irá receber a base de dados (*transactions_df*), a data que irá começar o treinamento (start_date_training), o periodo para a base de treino (delta_train), base de validacao (delta_val) e o delay entre essas bases (delta_delay). Vale destacar que essa função irá retornar um *tuple* com os índices das observações que irão compor cada base. Isso é importante porque será possível integrar essa função a biblioteca do `sklearn` e dessa forma utilizar toda sua funcionalidade.

```
# cv strategy:
def sequential_train_test_split(transactions_df,
            start_date_training,
            n_folds=5,
            delta_train=7,
            delta_delay=7,
            delta_val=7):
    sequential_split_indices = []
    # For each fold
    for fold in range(n_folds):
        # Shift back start date for training by the fold index times the validation period
        start_date_training_fold = start_date_training-datetime.timedelta(days=fold*delta_val)
        start_date_training_fold
        # Get the training and test (assessment) sets
        (train_df, val_df)=get_train_test_set(
            transactions_df,
            start_date_training=start_date_training_fold,
            delta_train=delta_train,
            delta_delay=delta_delay,
            delta_test=delta_val
        )
        # Get the indices from the two sets, and add them to the list of sequential splits
        indices_train = list(train_df.index)
        indices_val = list(val_df.index)
        sequential_split_indices.append((indices_train,indices_val))
```

```
    return sequential_split_indices
```

Vamos utilizar estratégia de estimacao como utilizando 5 folders, e 1 semana de dados para treinamento, 1 semana de dados para o base de validacao e 1 semana de delay. Irei manter a base de teste para que possamos testar se nossa estimativa de performance irá bater como a performance na base de teste.

```
# determing the folders:
n_folds=5
delta_train=7
delta_delay=7
delta_val=7

# defining start date in the sequential cv:
start_date_training_seq = start_date+datetime.timedelta(days=-(delta_delay+delta_val))

# creting the indexes for each sequential cv:
sequential_split_indices = sequential_train_test_split(
    transactions_df,
    start_date_training = start_date_training_seq,
    n_folds=n_folds,
    delta_train=delta_train,
    delta_delay=delta_delay,
    delta_val=delta_val
)
```

Vamos checar as datas de cada base de treinamento e validacao nessa estratégia:

```
# checking dates of training:
for i in list(range(n_folds))[::-1]:
    train_data1 = transactions_df.loc[sequential_split_indices[i][0]]["TX_DATETIME"].min()
    train_data2 = transactions_df.loc[sequential_split_indices[i][0]]["TX_DATETIME"].max()

    val_data1 = transactions_df.loc[sequential_split_indices[i][1]]["TX_DATETIME"].min()
    val_data2 = transactions_df.loc[sequential_split_indices[i][1]]["TX_DATETIME"].max()

    print("train range: "+datetime.datetime.strftime(train_data1, "%Y-%m-%d")+" - "+datetime.da
    print("val range: "+datetime.datetime.strftime(val_data1, "%Y-%m-%d")+" - "+datetime.dateti
    print("\n")
```

```
train range: 2023-06-30 - 2023-07-06
val range: 2023-07-14 - 2023-07-20


train range: 2023-07-07 - 2023-07-13
val range: 2023-07-21 - 2023-07-27
```

```
train range: 2023-07-14 - 2023-07-20
val range: 2023-07-28 - 2023-08-03


train range: 2023-07-21 - 2023-07-27
val range: 2023-08-04 - 2023-08-10


train range: 2023-07-28 - 2023-08-03
val range: 2023-08-11 - 2023-08-17
```

Cada base de validação está separada por 1 semana da base de treinamento. Como a base de teste possui transações entre `2023-08-25` e `2023-08-31`, então todas as bases estão de acordo.

Com a estratégia de CV definida, agora é possível extender a estimação dos algoritmos e testar algumas configurações de hyperparâmetros para cada algoritmo, de modo otimizar a performance do sistema de detecção de fraude.

Para isso, irei definir um conjunto de algoritmos e hyperparâmetros e testá-los utilizando a estratégia de `GridSearchCV`. Os algoritmos serão avaliados utilizando a métrica *average precision*.

```
# creating a pipeline:
myPipe = Pipeline(steps=[
    ("imputer", SimpleImputer(strategy="mean")),
    ('scaling', StandardScaler()),
    ("clf", LogisticRegression(max_iter=2000))
])

# Hyperparameters to test
myGrid = [
    {
        "clf": [LogisticRegression(solver="liblinear", max_iter=2000)],
        "clf__penalty": ["l1", "l2"],
        "clf__C": [0.5, 5]
    },
    {
        "clf": [BaggingClassifier(n_estimators=200,random_state=10) ],
        "clf__estimator": [DecisionTreeClassifier(), LogisticRegression()],
        "clf__max_features": [1, .5]
    },
    {
        "clf": [RandomForestClassifier(n_estimators=200,random_state=10)]
    },
    {
        "clf": [GradientBoostingClassifier(learning_rate=.05,random_state=10)],
        "clf__n_estimators": [100, 200]
    }
```

```
]

# Let us instantiate the GridSearchCV
# set refit=False. Do retrained the best model on the whole data.
from sklearn.model_selection import GridSearchCV
grid = GridSearchCV(
    myPipe,
    param_grid=myGrid,
    scoring="average_precision",
    cv=sequential_split_indices,
    refit=False,
    n_jobs=-1,
    verbose=0
)

# fitting:
grid.fit(transactions_df[input_features], transactions_df[output_feature])
```

Conforme definido no `GridSearchCV`, foi configurado `refit=False` para que a melhor configuração do modelo não fosse reestimada em todo o conjunto de dados. Como resultado, não serão retornados os atributos `.best_params_`, `.best_score_` e `.best_estimator_`. Essa configuração permite um maior controle sobre o processo de treinamento e avaliação do modelo, especialmente quando se deseja ajustar a configuração do modelo posteriormente com base em resultados específicos.

Para avaliar o desempenho dos modelos, será extraído o atributo `.cv_results_` e converter os resultados em um `DataFrame`. Em seguida, será calculado um intervalo de confiança para cada estimativa de desempenho. Essa abordagem nos proporcionará uma medida de incerteza em relação à previsão do desempenho dos nossos algoritmos na base de teste. Isso é essencial para compreender a variabilidade e a confiabilidade das estimativas de desempenho obtidas por meio da validação cruzada.

```
# getting the results:
results = pd.DataFrame(grid.cv_results_)
# assuming normal distribution:
results["lower_bd"] = results["mean_test_score"]-2*results["std_test_score"]
results["upper_bd"] = results["mean_test_score"]+2*results["std_test_score"]
# sorting:
results = results.sort_values("rank_test_score", ascending=True).reset_index(drop=True)
```

Para selecionar o modelo final, adoto a seguinte abordagem: escolho o modelo que se encontra a um desvio padrão da melhor performance encontrada no CV. Essa escolha visa reduzir ainda mais a dependência do modelo em relação às bases de validação, diminuindo assim o risco de *overfitting*. Para implementar esse processo, desenvolvi uma função auxiliar chamada `get_bestModel`. Essa função recebe o `DataFrame` contendo as métricas de desempenho e retorna um *tuple* com o índice do modelo selecionado e sua configuração correspondente.

```
# Defining a Custom best model criteria
def get_bestModel(cv_results):
```

```
    # copying
    cv_results = results.copy()
    # sorting by ranking:
    cv_results = cv_results.sort_values("rank_test_score").reset_index()
    # threshold: max(mean_test_score) - 1*std(mean_test_score)
    threshold = cv_results["mean_test_score"].max() - 1*cv_results["mean_test_score"].std()
    # filtering candidates with score greater than threshold:
    cv_results = cv_results[cv_results["mean_test_score"] > threshold]
    # From the candidates, select the one with the smallest score:
    # get index:
    best_model_idx = cv_results["mean_test_score"].idxmin()
    # get model:
    best_params = cv_results.loc[best_model_idx]["params"].copy()
    for name in list(best_params.keys()):
        newkey = name.split("__")[-1]
        best_params[newkey]=best_params.pop(name)
    best_model = best_params.pop("clf")
    best_model.set_params(**best_params)

    return best_model_idx, best_model
```

Aplicando a função anterior e obtendo o melhor modelo (segundo regra de 1 standard rule):

```
# get the best configuration:
idx, best_clf_config = get_bestModel(results)
results.loc[idx]
```

```
mean_fit_time                                                2.239888
std_fit_time                                                 0.099521
mean_score_time                                              0.069477
std_score_time                                               0.008442
param_clf                 LogisticRegression(C=0.5, max_iter=2000, solve...
param_clf__C                                                        0.5
param_clf__penalty                                                   12
param_clf__estimator                                                NaN
param_clf__max_features                                             NaN
param_clf__n_estimators                                             NaN
params                    {'clf': LogisticRegression(C=0.5, max_iter=200...
split0_test_score                                            0.652561
split1_test_score                                            0.601248
split2_test_score                                            0.612506
split3_test_score                                            0.610476
split4_test_score                                            0.584905
mean_test_score                                              0.612339
std_test_score                                               0.022347
rank_test_score                                                     8
lower_bd                                                     0.567645
```

```
upper_bd                                                    0.657033
Name: 7, dtype: object
```

Os resultados de CV demonstram que o algoritmo com o melhor desempenho (segundo a regra de 1SD) é uma regressão logística, com penalidade do tipo L2 (ridge) e *penalty term* = .5.

Agora podemos reestimar a configuração final do algoritmo utilizando a base de treino (`X_train`, `y_train`) e testar esse modelo sobre a base de teste (`X_test`, `y_test`):

```
# final pipeline:
mdl = Pipeline(steps=[
    ("imputer", SimpleImputer(strategy="mean")),
    ('scaling', StandardScaler()),
    ("clf", best_clf_config)
])

# fitting:
mdl.fit(X_train, y_train)

# predicting on test set:
y_pred = mdl.predict_proba(X_test)
# average precision score:
avg_prec_test = average_precision_score(y_test, y_pred[:, 1])

print(f"CV prediction: {np.round(results.loc[idx, 'mean_test_score'], 2)}")
print(f"CV confidence interval: [{np.round(results.loc[idx, 'lower_bd'], 2)}; {np.round(results
print(f"Test score: {np.round(avg_prec_test, 2)}")
```

```
CV prediction: 0.61
CV confidence interval: [0.57; 0.66]
Test score: 0.63
```

A estimativa de CV para a previsão de performance no base de teste foi de 0.61, com intervalo entre 0.57 - 0.66. A performance na base de teste do algoritmo foi de 0.63. Como é possível observar, a estratégia de CV foi uma boa alternativa para estimar algoritmos em suas diversas configurações e também para obter uma estimativa confiável acerca da performance futura do algoritmo.

Evaluation

Após desenvolver um modelo com um bom nível de qualidade (na perspectiva técnica), é fundamental conduzir uma avaliação abrangente para assegurar que o modelosistema atinge efetivamente os objetivos de negócio. Um ponto crucial é determinar se há alguma questão de relevância empresarial que não tenha recebido a devida consideração ao longo do processo.

Quando se trata de um sistema de detecção de fraudes em transações de cartão de crédito, é fundamental avaliar sua relevância operacional para o cenário empresarial. O objetivo central de tal sistema é emitir alertas sobre transações suspeitas, as quais são então submetidas à análise por parte dos investigadores.

Esse procedimento consome tempo. Em consequência, a quantidade de alertas que pode ser verificada durante um período específico é limitada.

Uma forma de medirmos o benefício operacional um sistema de detecção de fraude é atráves da mensuração de quanto ele ajuda os investigadores a confirmarem as fraudes. Suponha que investigadores conseguem analisar k transações suspeitas em um dia. *Precision top-k* é uma medida de performance de que maximiza a precisão da estimação de um sistema de detecção de fraude sobre o número k de alertas que um investigador consegue analisar.

Precision top-k é calculada da seguinte forma. Para um determinado dia d:

1. Ordene as transações suspeitas de acordo com a probabilidade;
2. Selecione as top k transações mais suspeitas;
3. Compute a precisão sobre essas k transações.

```
def precision_top_k_day(df_day, top_k=100):
    # sorting by the highest chance of fraud:
    df_day = df_day.sort_values(by="predictions", ascending=False)
    # Get the top k most suspicious transactions
    df_day_top_k=df_day.head(top_k)
    # Compute precision top k
    precision_top_k = df_day_top_k["TX_FRAUD"].mean()
    return precision_top_k
```

Podemos aplicar essa métrica ao nosso modelo e calcular a performance operacional de para cada dia da base de teste:

```
test_df["predictions"] = y_pred[:, 1]
nb_frauds = []
prec_top_k_test = []
for day in test_df["TX_TIME_DAYS"].unique():
    prec_top_k_test.append(precision_top_k_day(test_df.query("TX_TIME_DAYS == @day")))
    nb_frauds.append(test_df.query("TX_TIME_DAYS == @day")["TX_FRAUD"].sum())

# média de frauds diária
print(f"avg nb_frauds: {np.mean(nb_frauds)}")
# media de precision top100
print(f"avg precision top100: {np.mean(prec_top_k_test)}")
```

```
avg nb_frauds: 61.42857142857143
avg precision top100: 0.4042857142857143
```

Esse resultado nos diz que o algoritmo de detecção de fraude corretamente detectou uma média de 40 transações fraudulentas por dia. Ou seja, como uma média de 61 fraudes por dia, esse resultado sugere que cerca de 65% das transações fraudulentas foram detectadas pelo algoritmo.

Uma vez que tenhamos uma forma de medir a performance operacional do sistema, podemos utilizar essa métrica como medida para otimizar os algoritmos durante o processo de estimação.

Vamos repetir o processo de estimação. Dessa vez utilizando a métrica de média diária *precision-top-k*. Para isso, precisamos criar uma função *custom scorer* com a função `sklearn.metrics.make_scorer` para que seja junto com o `sklearn`:

```
# create a function that receivies y_true, y_pred and computes the daily precision:
def daily_avg_precision_top_k(y_true, y_pred, top_k, transactions_df):
    #y_true = y_test
    #y_pred = y_pred[:, 1]
    # get the test data:
    df = transactions_df.loc[y_true.index]
    # adding prediction
    df["predictions"] = y_pred
    # computing daily avg precision top-k:
    avg = df.groupby("TX_TIME_DAYS").apply(precision_top_k_day).mean()
    return avg

from sklearn.metrics import make_scorer
daily_avg_precision_top_k_score = make_scorer(
    daily_avg_precision_top_k,
    greater_is_better=True,
    needs_proba=True,
    top_k=100,
    transactions_df=transactions_df[['CUSTOMER_ID', 'TX_FRAUD','TX_TIME_DAYS']]
)
```

Uma vez criado a função para ser utilizada junto a biblioteca do `sklearn`, podemos repetir o processo de estimação com a média *precision-top-k*:

```
# using the same pipe and grid:
myPipe
myGrid

# Let us instantiate the another GridSearchCV. This time with
# our custom scorer:
grid2 = GridSearchCV(
    myPipe,
    param_grid=myGrid,
    scoring=daily_avg_precision_top_k_score,
    cv=sequential_split_indices,
    refit=False,
    n_jobs=-1,
    verbose=4
)

# fitting:
grid2.fit(transactions_df[input_features], transactions_df[output_feature])
```

```
# getting the results:
results2 = pd.DataFrame(grid2.cv_results_)
# assuming normal distribution:
results2["lower_bd"] = results2["mean_test_score"]-2*results2["std_test_score"]
results2["upper_bd"] = results2["mean_test_score"]+2*results2["std_test_score"]
results2 = results2.sort_values("rank_test_score", ascending=True).reset_index(drop=True)

# get the best configuration:
idx2, best_clf_config2 = get_bestModel(results2)
results2.loc[idx2]

# final pipeline:
mdl2 = Pipeline(steps=[
    ("imputer", SimpleImputer(strategy="mean")),
    ('scaling', StandardScaler()),
    ("clf", best_clf_config2)
])

# fitting:
mdl2.fit(X_train, y_train)
```

```
# predicting on test set:
y_pred2 = mdl2.predict_proba(X_test)

# daily average precision top_100 score:
avg_prec_top_100 = daily_avg_precision_top_k(y_test, y_pred2[:, 1], top_k=100, transactions_df=

# average precision score:
avg_prec_test = average_precision_score(y_test, y_pred2[:, 1])

print(f"CV prediction: {np.round(results2.loc[idx2, 'mean_test_score'], 2)}")
print(f"CV confidence interval: [{np.round(results2.loc[idx2, 'lower_bd'], 2)}; {np.round(resul
print(f"Test score: {np.round(avg_prec_top_100, 2)}")
```

```
CV prediction: 0.38
CV confidence interval: [0.36; 0.4]
Test score: 0.4
```

Assim como anterior, a estratégia de CV foi uma boa alternativa para estimar algoritmos em suas diversas configurações e também para obter uma estimativa confiável acerca da performance operacional do algoritmo.

Deployment

A criação de um sistema de detecção de fraude não é o fim de um projeto. Um vez que o sistema está estimado e os objetivos comerciais estão alinhados, chega a hora de disponibilizar esse sistema para que

ele seja integrado em diferentes plataformas, aplicativos web, entre outros. A forma mais comum é tornar o sistema disponível é em formato de REST API.

Primeira coisa a ser feita é salvar o modelo estimado para que ele seja reutilizado:

```
# getting the terminal and customer dict:
# save final model:
from joblib import dump
dump(mdl2, "output/model.joblib")
```

Algumas considerações precisam ser feitas. Por exemplo, ao desenhar um API, o desenvolvedor precisa imaginar como será as informações de *input* para que o modelo gere previsões.

No caso de presente problema, é provável que dados de transações de cartão de crédito contenha apenas as seguintes variáves:

- TRANSACAO_ID
- TX_DATETIME
- CUSTOMER_ID
- TERMINAL_ID
- TX_AMOUNT

Um primeiro obstáculo que surge é como realizar o *encode* das variáveis de cliente e terminal. Como vimos ao longo da estimação, essas variáveis foram *encodadas* com valores históricos calculados. Em um modelo em produção, não será possível fazer isso a extração dessas informações a partir de dados históricos. Utilizaremos como dicionário, o último valor de cada variável calculada para cada customer e terminal:

```
# save dict

# get the last info from terminal
dict_terminal = transactions_df[["TX_DATETIME", "TERMINAL_ID",
                                 'TERMINAL_ID_NB_TX_1DAY_WINDOW', 'TERMINAL_ID_RISK_1DAY_WINDOW
                                 'TERMINAL_ID_NB_TX_7DAY_WINDOW', 'TERMINAL_ID_RISK_7DAY_WINDOW
                                 'TERMINAL_ID_NB_TX_30DAY_WINDOW', 'TERMINAL_ID_RISK_30DAY_WIND
dump(dict_terminal, "output/dict_terminal.joblib")

# get the last info from customer:
dict_customer = transactions_df[["TX_DATETIME",
                                 "CUSTOMER_ID",
                                 'CUSTOMER_ID_NB_TX_1DAY_WINDOW',
                                 'CUSTOMER_ID_AVG_AMOUNT_1DAY_WINDOW',
                                 'CUSTOMER_ID_NB_TX_7DAY_WINDOW', 'CUSTOMER_ID_AVG_AMOUNT_7DAY_
                                 'CUSTOMER_ID_NB_TX_30DAY_WINDOW', 'CUSTOMER_ID_AVG_AMOUNT_30DA
dump(dict_customer, "output/dict_customer.joblib")
```

Uma limitação dessa estratégia é que as novas transações pode ser feitas por clientes e/ou terminais que não estejam contemplados nos dicionários. Para esses casos, o modelo foi criado como um pipeline onde está contido uma pré-processamento de imputação. Dessa forma, o algoritmo ainda retornará uma previsão para transações com essas entidades desconhecidas, porém com pouco menos de precisão.

Uma vez que tenhamos os dicionários das variáveis categóricas e o modelo salvos, podemos escrever nosso API usando a biblioteca `Flask`:

```
from flask import Flask, request
from joblib import load
import pandas as pd

app = Flask(__name__)

@app.route("/fraud_alert", methods=["POST"])
def fraud_alert():
    # catching json:
    json_ = request.get_json()
    # convert to DF:
    df = pd.DataFrame(json_)
    # convert to datetime
    df["TX_DATETIME"] = pd.to_datetime(df["TX_DATETIME"])
    # creating features:
    # - whether a transaction occurs during a weekday or a weekend
    df["TX_DURING_WEEKEND"] = (df["TX_DATETIME"].dt.weekday >= 5).astype("int")
    # - whether a transaction is at night: night definition: 22 <= hour <= 6
    df["TX_DURING_NIGHT"] = ((df["TX_DATETIME"].dt.hour <= 6) | (df["TX_DATETIME"].dt.hour >= 2
    # customer features
    df = pd.merge(df, dict_customer, on="CUSTOMER_ID", how="left")
    # terminal features
    df = pd.merge(df, dict_terminal, on="TERMINAL_ID", how="left")
    # input features:
    input_features = [
    'TX_AMOUNT', 'TX_DURING_WEEKEND', 'TX_DURING_NIGHT',
    'CUSTOMER_ID_NB_TX_1DAY_WINDOW', 'CUSTOMER_ID_AVG_AMOUNT_1DAY_WINDOW',
    'CUSTOMER_ID_NB_TX_7DAY_WINDOW', 'CUSTOMER_ID_AVG_AMOUNT_7DAY_WINDOW',
    'CUSTOMER_ID_NB_TX_30DAY_WINDOW', 'CUSTOMER_ID_AVG_AMOUNT_30DAY_WINDOW',
    'TERMINAL_ID_NB_TX_1DAY_WINDOW', 'TERMINAL_ID_RISK_1DAY_WINDOW',
    'TERMINAL_ID_NB_TX_7DAY_WINDOW', 'TERMINAL_ID_RISK_7DAY_WINDOW',
    'TERMINAL_ID_NB_TX_30DAY_WINDOW', 'TERMINAL_ID_RISK_30DAY_WINDOW'
    ]
    # prediction:
    df["prob_fraud"] = mdl.predict_proba(df[input_features])[:, 1]

    return df[["TRANSACTION_ID", "prob_fraud"]].to_dict(orient="records")


if __name__ == "__main__":
    # loading dicts:
    dict_customer = load("output/dict_customer.joblib")
    dict_terminal = load("output/dict_terminal.joblib")
    # loading model:
```

```
    mdl = load("output/model.joblib")
    # running debug mode:
    app.run(debug=True)
```

A partir de um terminal, podemos levantar a API com o seguinte comando:

```
pipenv run python api.py
```

Endpoints

- **URL**: `/fraud_alert`
- **Method**: `POST`
- **Descrição**: Esse endpoint irá calcular a probabilidade de uma transação ser considerada fraudulenta.
- **Input**: `JSON` com `TRANSACAO_ID`, `TX_DATETIME`, `CUSTOMER_ID`, `TERMINAL_ID`, `TX_AMOUNT`. Exemplo:


```
[{"TRANSACTION_ID":892153,"TX_DATETIME":1693612359000,"CUSTOMER_ID":3465,"TERMINAL_ID":2982,"TX
```


- **Resposta**: `JSON` com o seguinte formato:


```
[
  {
    "TRANSACTION_ID": 892154,
    "prob_fraud": 0.0007436351351168843
  },
  {
    "TRANSACTION_ID": 892155,
    "prob_fraud": 0.0019805953668296816
  },
  {
    "TRANSACTION_ID": 892156,
    "prob_fraud": 0.00022534077315471119
  }
]
```


Há um conjunto de transações referente ao dia `2023-09-01` no endereço `data/transactions_20230901.json`. Para realizar um `request` com esses dados, podemos proceder da seguinte forma:

```
curl -XPOST -H "Content-Type: application/json" -d @data/transactions_20230901.json http://127.
```