KUNUMI

REFRAMING THE PRESENT
FOR A REIMAGINED FUTURE

Banrisul

ROTEIRO DOS CICLOS DE MODELAGEM ASSISTIDOS

Fevereiro/2021

Objetivo 3

Estrutura dos Pipelines 3

Especificação dos Pipelines 3
Especificação dos Nodes 4
Configuração dos Pipelines 7
Parâmetros 7
Datasets 8
Credenciais 9

Execução de um Ciclo de Modelagem 10

Execução de Pipelines do Kedro 10
Execução do Pipeline de Backtest 11
Avaliação Técnica do Modelo em Backtest 12

Ciclos de Modelagem Sugeridos 19

Aspectos Práticos 19
Ajustes no Modelo 22
Método de Machine Learning 22
Hiperparâmetros 23
Otimização de Hiperparâmetros 23
Pré-processamento 24
Método e Parâmetros de Calibração 25
Remoção de Features 25
Adição de Features 26
Sugestões de Experimentos 26
Avaliar Configuração Atual 26
Experimento com Hiperparâmetros do XGBoost 26
Experimento Adicionando Padronização 27
Experimento Adicionando Imputação 27
Experimento com o Random Forest 27
Experimento com o Otimização do Random Forest 28
Experimento com Calibração 28
Experimento com o Tempo de Treino e OOS 28
Avaliação de Features por Remoção 28
Implementação e Adição de uma Feature Nova 29

1. Objetivo

O objetivo deste documento é prover uma sugestão de ciclos de modelagem para efetuar a transferência do conhecimento sobre os modelos desenvolvidos pela Kunumi para o Banrisul. A seção 2 contém uma explicação de como os pipelines de dados foram implementados e como estão estruturados. A seção 3 explica como executar os pipelines do Kedro, e mais especificamente, como executar os pipelines de backtesting e análise durante um ciclo de modelagem e como avaliar o modelo treinado. Por fim, a seção 4 contém uma sugestão de ciclos de modelagem, onde cada ciclo utiliza de um processo diferente de ajuste do modelo para aprimorar os resultados em backtest.

2. Estrutura dos Pipelines

As soluções implementadas pela Kunumi para os dois problemas propostos no projeto com o Banrisul foram feitas no formato de pipelines de dados. Um pipeline de dados consiste em um conjunto de operações encadeadas, cada uma tendo determinados conjuntos de dados de entrada e de saída, que juntos resultam em um fluxo de dados para produzir o resultado desejado. Especificamente, os dados de entrada consistem nas informações dos clientes a serem avaliados pelo modelo e a saída consiste na predição de probabilidade dada pelo modelo e a explicação.

Abaixo estão descritos como os pipelines são especificados e configurados utilizando o Kedro, ferramenta escolhida para a implementação dos pipelines. Recomendamos fortemente consultar o código ao longo da leitura para viabilizar o entendimento de como a solução está organizada, e consequentemente, como realizar um ciclo de modelagem. Para a consulta do código, favor utilizar a branch **`ciclos-assistidos`**.

Note que tudo o que está descrito a seguir foi implementado pela Kunumi e é passível de alteração, sendo modificado conforme os requisitos da solução. Portanto, é importante entender como cada elemento foi implementado caso seja necessário realizar alterações ou ajustes no futuro.

2.1. Especificação dos Pipelines

Há cinco pipelines de dados principais na solução, cada um correspondendo a um momento diferente do processo do modelo, que são: **`monthly`**, **`backtesting`**, **`analysis`**, **`training`** e **`inference`**.

- O pipeline **`monthly`** é utilizado para pré-calcular algumas informações mensais dos clientes e salvar os resultados no banco de dados para serem utilizados em outros pipelines. Especificamente, ele calcula o tempo de atraso dos clientes e as features de granularidade mensal. Esse pipeline está apenas no repositório do modelo de concessão pois os resultados dele são compartilhados pelo banco de dados.

- O pipeline **backtesting** utiliza dados históricos para treinar e avaliar os modelos de acordo com a parametrização especificada, simulando o modelo em funcionamento com o passado.
- O pipeline **analysis** é um pipeline complementar ao de backtesting, que é utilizado para realizar análises adicionais de qualidade dos modelos em backtest.
- Já o pipeline **training** treina um modelo para que o mesmo seja utilizado em produção, dada uma parametrização que tenha sido testada e validada em backtesting.
- Por fim, o pipeline **inference** utiliza o modelo treinado no pipeline de treinamento para escorar exemplos novos.

Os pipelines de dados foram implementados utilizando a biblioteca Kedro do Python. Esta biblioteca provê abstrações de pipelines, dados (datasets) e operações (nodes). O Kedro também provê uma interface para a execução dos pipelines por meio da linha de comando e pelo próprio Python.

Os pipelines que podem ser executados pelo Kedro são especificados dentro do arquivo `src/banrisul_concessao/hooks.py` (ou `src/banrisul_pendencias/hooks.py`) dentro da função `register_pipelines`. Esta função retorna um dicionário mapeando o nome de cada pipeline ao objeto que o representa. Além disso este dicionário contém uma entrada `__default__` que indica qual pipeline deve ser executado caso nenhum pipeline seja especificado.

A definição dos pipelines em si é feita dentro do módulo `src/banrisul_concessao/pipelines` (ou `src/banrisul_pendencias/pipelines`), onde cada cada pipeline é definido dentro de um arquivo Python específico.

Cada pipeline é representado pela classe `kedro.pipeline.Pipeline` e é definido dentro da função `create_pipeline`, sendo retornado ao final da mesma. A definição do pipeline consiste dos nodes que compõem o mesmo e tags opcionais que facilitam a execução.

2.2. Especificação dos Nodes

Cada node é instanciado pela função `kedro.pipeline.node` e recebe como argumento: a função Python que implementa a operação desejada, as entradas, as saídas e o nome do node. As funções dos nodes são definidas dentro dos demais módulos do repositório em `src/banrisul_concessao` (ou `src/banrisul_pendencias`) e importadas no arquivo de definição de cada pipeline. As entradas, identificadas pelo nome de cada, podem ser tanto parâmetros quanto datasets, ambos buscados da configuração do projeto (que será explicado na subseção seguinte). Já as saídas, também identificadas pelo nome de cada, devem consistir de datasets que também são buscados da configuração do projeto. Em geral, tanto as entradas quanto as saídas especificadas para cada node devem corresponder aos argumentos e retorno da função do node respectivamente. Como exemplo, se a função do node recebe três argumentos e possui dois retornos, é necessário especificar três entradas e duas saídas na definição do node.

Para facilitar o entendimento dos pipelines implementados, segue abaixo uma lista dos principais nodes dos pipelines mensal, backtesting e análise, e o objetivo de cada. Para o modelo de

pendências, alguns destes nodes tem duas versões, uma para o cálculo de Propensão de Pagamento por Cobrança (PPC) e outra para o cálculo de Propensão à Auto-Cura (PAC).

Pipeline	Node	Objetivo
Mensal	`preprocess_atraso`	Calcula o tempo de atraso por mês, cliente e conta BIB usando as tabelas do BIG (POSICAO_CLI_MES, POSICAO_PARCELA_ABERTA, ESTOQUE_BRW, PAGAMENTO_PARCELA e CONTA) e salva os resultados no banco de dados.
	`features_*` (mensal)	Calcula um determinado conjunto de features com base nos dados de entrada mensais e salva os resultados no banco de dados (múltiplos nodes).
Backtest (concessão)	`preprocess_bwa_proposta`	Pré-processa as tabelas PROPOSTA e CLI_PROD_PROPOSTA do BWA para o próximo node.
	`preprocess_primary`	Pré-processa as propostas que devem ser incluídas no modelo e calcula o target (inadimplência nos próximos 6 meses).
	`features_proposta`	Calcula as features baseadas nas informações da proposta.
Backtest (pendências)	`preprocess_pendencia_dia`	Pré-processa a tabela de COBRANCA_BJG_DIA.
	`preprocess_folha_pg`	Pré-processa os clientes que recebem pelo estado através da tabela BIG.HISTORICO_CLIENTE.
	`preprocess_primary_ppc`	Pré-processa os datasets primary com as pendências-datas a serem incluídas no modelo de Propensão a Pagamento de Cobrança (PPC).

	`preprocess_primary_pac`	Pré-processa os datasets primary com as pendências-datas a serem incluídas no modelo de Propensão a Auto Cura (PAC).
	`preprocess_primary_entities`	Separa uma tabela primary de pendências tanto do PAC quanto do PPC para facilitar os processamentos futuros.
	`features_pendencia_*`	Calcula as features baseadas nas informações da pendência (múltiplos nodes).
Backtest (ambos)	`walk_forward_split`	Aplica o algoritmo do Walk Forward para separar os dados em folds temporais de conjuntos de treino, validação e out of sample.
	`features_merge`	Realiza a junção das tabelas de features para os exemplos de entrada.
	`fit_model`	Ajusta o modelo de acordo com as informações de entrada (tabela de features, tabela de targets, resultado do walk forward e hiperparâmetros do modelo).
	`predict`	Calcula a predição do modelo para os exemplos recebidos como entrada.
	`calculate_metrics`	Calcula as métricas especificadas dado as predições de entrada.
Análise (ambos)	`calculate_shap_values`	Calcula os valores do SHAP para os modelos e exemplos recebidos como entrada.
	`lca`	Calcula a Learning Curve Analysis (LCA - Análise de Curva de Aprendizado) dado os parâmetros especificados para a análise.

	`msa`	Calcula o Modelo Stability Analysis (MSA - Análise de Estabilidade do Modelo) dado os parâmetros especificados para a análise.

2.3. Configuração dos Pipelines

Os pipelines de dados implementados possuem três componentes de configuração, seguindo o padrão do Kedro: parâmetros, datasets e credenciais. Os parâmetros são definições de valores utilizados na execução dos pipelines. As configurações de datasets especificam, para cada dataset, identificado por um nome, qual classe do Python deve tratar a leitura e a escrita daquele dataset e quais são os atributos dessa classe. Por fim, as credenciais especificam as informações necessárias para autenticação em bancos de dados ou outras fontes externas de dados.

No caso dos datasets, é possível não especificar a configuração de determinados datasets. Neste caso, eles serão tratados apenas em memória. Consequentemente, este dataset não será salvo após ser retornado pelo node; e não poderá ser usado como entrada em um node sem que ele tenha sido produzido por outro node na mesma execução.

Todos os três componentes de configuração são armazenados na pasta `conf` e sub-divididos em dois tipos: `base` e `local`. As configurações do `base` são compartilhadas entre todos os usuários que utilizam o repositório e são salvas no Git/SVN (exceto as credenciais). Já as configurações do `local` são específicas para a instância local do repositório e não são armazenadas no Git/SVN. As configurações local tem precedência sobre as configurações base (ou seja, se existir uma configuração com o mesmo nome em ambas, o local que é considerado).

2.3.1. Parâmetros

Abaixo estão listados os principais parâmetros utilizados atualmente no pipeline de backtesting, e consequentemente, nos ciclos de modelagem. Eles se encontram no arquivo `conf/base/backtest.yml`. Vale ressaltar que esta forma de organização foi uma escolha de implementação, e não é a única forma possível de parametrizar o pipeline de backtest.

- **`backtest_period`**: Lista com dois elementos, a data de início e de fim do período considerado para o backtest (no formato YYYY-MM-DD em string). Este parâmetro define o período que será utilizado para gerar exemplos, treinar modelos e avaliar os mesmos em backtest.

- **`backtest_features_drop`**: Lista dos nomes (em string) das features a serem removidas no node de `features_merge` e consequentemente do modelo. Este parâmetro facilita a remoção de uma ou mais features de forma rápida e sem necessitar a remoção do código daquela feature específica.

- **`metrics_names`**: Lista dos nomes (em string) das métricas a serem calculadas para avaliar a qualidade do modelo. As métricas disponíveis para serem adicionadas aqui são especificadas dentro do módulo `utils/metrics.py`.

- **`walk_forward`**: Dicionário de parâmetros do algoritmo de Walk Forward. Este dicionário é passado para o node que realiza o Walk Forward (`walk_forward_split`) e consiste dos seguintes valores:
    - **`train_size_min`**: Tamanho mínimo do conjunto de treino em número de datas únicas.
    - **`train_size_max`**: Tamanho máxima do conjunto de treino em número de datas únicas.
    - **`train_to_valid_gap_size`**: Tamanho do gap entre o treino e a validação em número de datas únicas.
    - **`valid_size`**: Tamanho da validação em número de datas únicas.
    - **`valid_to_oos_gap_size`**: Tamanho do gap entre a validação e o out of sample em número de datas únicas.
    - **`oos_size`**: Tamanho do out of sample em número de datas únicas.

- **`backtest_model`**: Dicionário utilizado para especificar os pré-processamentos a serem aplicados nas features, o método de Machine Learning (ML), o método de otimização de hiperparâmetros e o método de calibração. Este dicionário é convertido para um pipeline do sklearn dentro de `model/model.py`, que é então treinado pela função `fit_model` dentro de `model/nodes.py`. No dicionário são especificados os componentes (opcionais) de pré-processamento (imputação, padronização e seleção de features) assim como o classificador em si, que é dividido em três sub-componentes, o calibrador (calibrador), o otimizador de hiperparâmetros (opcional) e o estimador em si. Para cada componente, são especificados o nome do método, que identifica a classe do método, e um dicionário de parâmetros, que são passados na inicialização da classe. Os elementos que podem ser incluídos, e a ordem em que são adicionados ao pipeline do Scikit-learn, estão listados abaixo.
    - **`imputer` (opcional)**: método para preencher valores faltantes nas features.
    - **`scaler` (opcional)**: método para padronizar os valores das features.
    - **`feature_selection` (lista, opcional)**: um ou mais métodos para seleção de features, aplicados em sequência.
    - **`classifier`**:
        - **`calibrator`**: método de calibração.
        - **`optimizer`**: método de otimização de hiperparâmetros.
        - **`estimator`**: método de classificação usando ML.

- **`lca`**: Dicionário de parâmetros do LCA, que contém os seguintes parâmetros.
    - **`random_seed`**: Semente do gerador aleatório.
    - **`insample_sizes`**: Lista com as proporções do in sample que devem ser utilizadas no LCA.

- **`msa`**: Dicionário de parâmetros do MSA, que contém os seguintes parâmetros.
    - **`step_size`**: Tamanho do passo utilizado no cálculo do MSA, em número de datas únicas. Este passo define de quantos em quantos dias (ou meses) a performance é avaliada no conjunto de out of sample para medir a estabilidade do modelo.

- **walk_forward**: Parâmetros a serem passados para o `walk_forward_split` durante o MSA (possui o mesmo formato do `walk_forward` usado para treinar o modelo.

2.3.2. Datasets

De forma análoga, abaixo constam os principais datasets utilizados no backtesting. Eles se encontram na pasta `conf/base/catalog_dev/`, sendo que há uma pasta para cada pipeline e dentro de cada um arquivo para cada camada de dados. Datasets usados por mais de um pipeline não estão em sub-pastas. Como o próprio nome indica, tais datasets são usados apenas em desenvolvimento, sendo que os datasets usados em produção ficam na pasta `conf/base/_catalog_prod/` (apenas um catálogo deve estar habilitado por vez, sendo que o catálogo que está desabilitado deve ter um underscore no início do nome).

- **raw_big_*, raw_bjg_* e raw_bwa_***: Representam as tabelas dos schemas BIGI01, BJGI01 e BWAI01 respectivamente (múltiplos datasets).
- **primary_time_series_backtest**: Tabela da série temporal dos exemplos a serem utilizados no modelo, indexados pela entidade e data de referência.
- **primary_entities_backtest**: Tabela com as informações das entidades a serem utilizadas no modelo (propostas ou pendências).
- **primary_targets_backtest**: Valores dos targets do modelo para cada entidade e data de referência.
- **features_*_backtest**: Tabela com as features resultantes de um determinado node de cálculo de features (múltiplos datasets).
- **model_input_features_backtest**: Tabela com a junção das features para os exemplos a serem utilizadas no modelo.
- **model_input_temporal_folds_backtest**: Tabela contendo o resultado do Walk Forward, indicando a partição (treino, validação ou out of sample) para cada fold temporal e data de referência.
- **models_list_backtest**: Modelos treinados no node `fit_model`, salvos como uma lista de modelos (um modelo para cada fold temporal) como um Pickle do Python.
- **model_output_predictions_backtest**: Tabela com as predições do modelo para cada exemplo e fold temporal.
- **reporting_metrics_backtest**: Tabela com os valores calculados para cada métrica em cada partição e fold temporal.

- **reporting_shap_values_backtest**: Tabelas com os valores do SHAP para os exemplos utilizados no modelo para cada fold temporal.

- **reporting_lca_backtest**: Resultado do LCA, indicando o valor de cada métrica para cada iteração do LCA.

- **reporting_msa_backtest**: Resultado do MSA, indicando o valor de cada métrica para cada iteração do MSA.

2.3.3. Credenciais

Por fim, segue abaixo as credenciais que devem estar configuradas para a execução do backtest. As credenciais precisam ser especificadas no formato abaixo (exemplificado para o `bigi01_con`):

```
bigi01_con:
  con: oracle+cx_oracle://usuário:senha@BAIDBD.WORLD
```

- **bigi01_con**: Credencial de acesso aos dados do schema BIGI01
- **bjgi01_con** (apenas modelo de pendências): Credencial de acesso aos dados do schema BJGI01
- **bwa01_con** (apenas modelo de concessão): Credencial de acesso aos dados do schema BWAI01

3. Execução de um Ciclo de Modelagem

Tendo em vista a forma como os pipelines estão atualmente estruturados e configurados, esta seção tem como objetivo explicar aspectos práticos da modelagem, descrever como executar partes do pipeline ou pipelines completos e como avaliar o modelo treinado em uma execução completa dos pipelines de backtest e análise.

3.1. Aspectos Práticos

Antes de iniciar os ciclos de modelagem, de forma a viabilizar o trabalho colaborativo, é importante criar uma branch para você trabalhar. Especificamente para estes ciclos, favor criar uma nova branch a partir da branch **ciclos-assistidos**, que já foi preparada para a realização dos ciclos e para evitar conflitos com alterações sendo feitas na branch principal de desenvolvimento (develop).

Ainda relacionado ao versionamento de código, recomendamos utilizar a funcionalidade de ignorar arquivos do SVN para evitar que arquivos específicos, como arquivos de dados, sejam enviados por engano para o repositório. Isto pode ser feito no SVN executando o comando abaixo estando na pasta raiz do projeto.

```
svn propset svn:ignore svnignore.txt . --recursive
```

Um outro aspecto prático é sobre a utilização de recursos do servidor de desenvolvimento (LUD6). Dado que haverão múltiplas pessoas trabalhando em um servidor com recursos limitados para esse número de pessoas, lembre-se sempre de liberar os recursos (memória e processamento) quando terminar de utilizá-los. Especificamente, encerrar (shutdown) todos os notebooks que ainda estiverem como running no JupyterHub. Isto pode ser verificado na segunda aba do JupyterHub, conforme a figura abaixo. Pedimos que **sempre** façam isso ao terminarem o que estiverem fazendo.

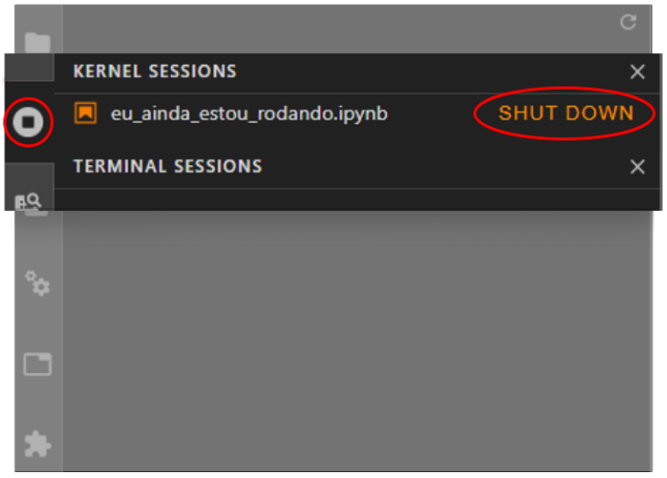

Figura 9 - Aba Kernel Sessions no JupyterHub

E um último aspecto prático é sobre a utilização do banco de dados. Algumas tabelas com resultados intermediários utilizados durante a modelagem são salvas no banco de dados. Dado que cada pessoa irá trabalhar separadamente, é importante utilizar tabelas separadas (exceto se for apenas para leitura de um informação já presente na tabela). Atualmente temos apenas uma tabela do pipeline no banco de dados no catálogo de desenvolvimento, a `BACKTEST_PRI_ENTIDADES_MODELO` (que está no schema BWAI01 para o modelo de concessão e no schema BJGI01 para o modelo de pendências). Portanto, antes de começar, é importante configurar a sua versão individual da tabela no catálogo do projeto.

Isto pode ser feito utilizando uma versão local da configuração. Para isto basta criar uma cópia dos arquivos da pasta `conf/base/catalog_dev/backtest/` na pasta `conf/local/catalog_dev/backtest/`, e então substituir a tabela original pela sua tabela. Os arquivos que precisam ser copiados são os listados abaixo:

Apenas no modelo de concessão:

```
De:   conf/base/catalog_dev/backtest/01_raw_big.yml
Para: conf/local/catalog_dev/backtest/01_raw_big.yml
```

Ambos:
```
De:   conf/base/catalog_dev/backtest/03_primary.yml
Para: conf/local/catalog_dev/backtest/03_primary.yml

De:   conf/base/catalog_dev/backtest/04_features.yml
Para: conf/local/catalog_dev/backtest/04_features.yml
```

E então substituir todas as ocorrências da tabela `backtest_pri_entidades_modelo` (mantendo se estava em minúsculo/maiúsculo) pela sua cópia da tabela, conforme a lista abaixo:

- **Risco** (schema BWAI01)
  - **Elise Aubin**: pri_entidades_modelos_risco_001
  - **Eder Silveira**: pri_entidades_modelos_risco_002
  - **Luis Schuck:** pri_entidades_modelos_risco_003
  - **Tiago Panerai**: pri_entidades_modelos_risco_004
  - **Vilquer Oliveira**: pri_entidades_modelos_risco_005

- **UREC** (schema BJGI01)
  - **Aparecida Mansur**: pri_entidades_modelos_urec_001
  - **Jorge Ullmann**: pri_entidades_modelos_urec_002
  - **Matheus Alcantara**: pri_entidades_modelos_urec_003

Abaixo estão alguns exemplos de como ficaria a troca.

```
# Antes
primary_entities_backtest:
  type: banrisul_concessao.io.datasets.DelayedFilteredSQLDataSet
  credentials: bwai01_con
  load_args:
    table_name: backtest_pri_entidades_modelo
    schema: BWAI01
  save_args:
    table_name: backtest_pri_entidades_modelo
    schema: BWAI01
    mode: “replace”
    index: False

# Depois
primary_entities_backtest:
  type: banrisul_concessao.io.datasets.DelayedFilteredSQLDataSet
  credentials: bwai01_con
  load_args:
    table_name: pri_entidades_modelo_001
    schema: BWAI01
  save_args:
    table_name: pri_entidades_modelo_001
    schema: BWAI01
```

```
    mode: “replace”
    index: False
```

```
# Antes
features_equifax_cliente_backtest:
  type: banrisul_concessao.io.datasets.DelayedFilteredSQLDataSet
  credentials: bigi01_con
  load_args:
    query: “SELECT * FROM BIGI01.FEAT_EQUIFAX_CLIENTE WHERE SEQ_CLIENTE
           IN (SELECT DISTINCT SEQ_CLIENTE FROM
           BWAI01.BACKTEST_PRI_ENTIDADES_MODELO”
  save_args:
    table_name: feat_equifax_cliente
    schema: BIGI01
    mode: “append”
    index: False

# Depois
features_equifax_cliente_backtest:
  type: banrisul_concessao.io.datasets.DelayedFilteredSQLDataSet
  credentials: bigi01_con
  load_args:
    query: “SELECT * FROM BIGI01.FEAT_EQUIFAX_CLIENTE WHERE SEQ_CLIENTE
           IN (SELECT DISTINCT SEQ_CLIENTE FROM
           BWAI01.PRI_ENTIDADES_MODELO_001”
  save_args:
    table_name: feat_equifax_cliente
    schema: BIGI01
    mode: “append”
    index: False
```

3.2. Execução de Pipelines do Kedro

O Kedro disponibiliza duas formas principais de executar os pipelines de dados especificados: utilizando a aplicação por linha de comando e diretamente pelo Python. Aqui o foco será na execução por linha de comando, porém a execução pelo Python é bastante simples e segue os mesmos princípios.

Para executar um pipeline pela linha de comando, é necessário primeiramente estar na pasta raiz do projeto (a mesma pasta que possui o arquivo `kedro_cli.py`). Estando nesta pasta, a execução é feita pelo comando:

```
kedro run
```

O comando acima irá executar o pipeline configurado como default (dentro de `hooks.py`) por completo, incluindo todos os nodes. A ordem de execução é determinada pelo Kedro através de uma ordenação topológica dos nodes do grafo (i.e. de forma que cada node X é executado após todos os nodes que tem como saída as entradas de X).

Para especificar um pipeline diferente do default, basta adicionar uma opção a mais:

```
kedro run --pipeline backtesting
```

Também é possível especificar exatamente quais nodes você deseja executar, por exemplo:

```
kedro run -n fit_model              # Executar apenas fit_model
kedro run -n fit_model -n predict   # Executar apenas fit_model e predict
```

Outra forma de execução é através das tags que foram especificadas para os pipelines e sub-pipelines no código, conforme o comando abaixo:

```
kedro run -t preprocessing          # Executa todos os nodes dentro de
                                      pipelines com a tag “preprocessing”
```

Também é possível especificar nodes de origem e/ou nodes de destino. No primeiro caso, o Kedro irá executar os nodes especificados e todos os nodes alcançáveis a partir deles. No segundo caso, o Kedro irá executar todos os nodes necessários para alcançar os nodes especificados, assim com os mesmos. Abaixo estão um exemplo de cada:

```
kedro run --from-nodes features_merge
kedro run --to-nodes calculate_metrics
```

Muitas das opções de execução listadas acima não são mutuamente exclusivas, sendo possível utilizar mais de uma opção por vez, por exemplo:

```
kedro run --pipeline training -n fit_model
```

Para visualizar uma lista completo de todas as opções de execução, basta executar o comando abaixo:

```
kedro run --help
```

3.3. Execução do Pipeline de Backtest

Tendo em vista as formas apresentadas de execução no Kedro, para executar o pipeline de backtesting e análise basta usar os comandos abaixo:

```
kedro run --pipeline backtesting
kedro run --pipeline analysis
```

Ou apenas `kedro run` se o pipeline default atualmente for o pipeline de backtest mais o pipeline de análise (é possível concatenar pipelines do Kedro).

Atualmente todos os datasets então configurados para serem salvos no disco ou no banco de dados. Com isso, depois da primeira execução completa, é possível aproveitar os resultados anteriores em execuções futuras, reduzindo o tempo necessário para executar novos ciclos de modelagem. Após realizar algum ajuste no modelo, é necessário apenas executar os nodes modificados e os subsequentes no pipeline.

Por exemplo, caso eu tenha feito uma alteração nos hiperparâmetros do modelo e desejasse ver as novas métricas, bastaria eu executar o seguinte comando:

```
kedro run -n fit_model -n predict -n calculate_metrics
```

3.4. Avaliação Técnica do Modelo em Backtest

Feita uma execução completa dos pipelines de backtesting e análise, o resultado do modelo em backtest é obtido, principalmente, através das métricas calculadas no node de `calculate_metrics`. O resultado deste node é uma tabela contendo o valor de cada métrica em cada partição e fold temporal do Walk Forward. Para facilitar a modelagem, o próprio node imprime na tela um log com o resumo das métricas, já calculando a média e desvio padrão por métrica e partição.

Além disto, para visualizar os resultados em maior detalhe, foi feito um notebook que executa um código que lê as métricas e gera algumas visualizações dos resultados obtidos, que está em `notebooks/reporting.ipynb`. O notebook contém as seguintes análises:

Média e desvio padrão das métricas: Nesta análise são mostrados os principais indicadores de qualidade do modelo. No contexto do projeto do Banrisul, as duas métricas principais selecionadas foram a área sob a curva ROC (AUC) e o KS. Outra métrica importante sendo avaliada é o Expected Calibration Error (ECE), que mede a qualidade da calibração.

Recapitulando o que foi explicado no workshop sobre validação de modelos, foi utilizada uma abordagem de validação cruzada utilizando o algoritmo do Walk Forward e a divisão de cada fold

temporal em três partições, treino, validação e out of sample (OOS). A partição de treino é utilizada para ajustar o modelo, a de validação é usada na calibração e para otimizar hiperparâmetros, e o OOS é utilizado para estimar a qualidade do modelo em exemplos novos. Cada métrica é calculada separadamente para cada partição de cada fold temporal, e então é feita a média e o desvio padrão dentre os folds temporais (exceto quando há apenas um fold temporal).

O resultado do modelo no OOS indica qual o valor médio esperado da métrica para exemplos novos no horizonte considerado (e.g. se foram utilizados 3 meses de OOS, o valor indica a média esperada nos próximos 3 meses após o treino do modelo). O resultado do modelo na validação também serve de estimativa para a qualidade do modelo, porém para exemplos que foram usados na calibração e/ou otimização de hiperparâmetros, e portanto influenciaram no modelo. Já o resultado do modelo no treino indica a qualidade do modelo para exemplos usados no seu ajuste.

Ao analisar o valor das métricas em cada partição, é importante observar a diferença entre o valor obtido no treino e o valor obtido no OOS (assim como na validação). Se o valor obtido no treino for significativamente melhor que o valor obtido no OOS, isto indica que o modelo pode estar sofrendo de *overfitting*. Ou seja, o modelo está "memorizando" exemplos de treino e não está generalizando para exemplos novos. Neste caso, duas abordagens bastante utilizadas para mitigar o problema são: aumentar o número de exemplos de treino, seja aumentando o período de tempo ou o número de entidades consideradas; e ajustando o modelo, seja utilizando um classificador mais simples, alterando os hiperparâmetros do classificador que afetam a sua complexidade, ou mesmo removendo features do modelo. A seção 4 irá tratar destas alterações que podem ser feitas no modelo com o objetivo de melhorar o modelo.

		mean	std
partition	metric_name		
oos	accuracy	0.931	0.004
	brier_loss	0.057	0.003
	ece	0.033	0.004
	ks	0.583	0.031
	roc_auc	0.869	0.010
train	accuracy	0.948	0.006
	brier_loss	0.043	0.005
	ece	0.033	0.005
	ks	0.671	0.008
	roc_auc	0.910	0.005
valid	accuracy	0.938	0.007
	brier_loss	0.052	0.007
	ece	0.025	0.004
	ks	0.570	0.055
	roc_auc	0.861	0.020

Figura 1 - Exemplo de resultado das métricas

Análise de score: Esta análise complementar tem como objetivo mostrar a distribuição das predições dadas pelo modelo para cada uma das classes consideradas. Isto é feito através do gráfico da distribuição acumulada do score do modelo para cada uma das classes. Ou seja, o gráfico mostra, para cada valor de score (no eixo x), a proporção dos exemplos daquela classe (no eixo y) que tem um score menor ou igual à aquele score de referência. Esta visualização ajuda a entender quão boa é a separação das classes feita pelo modelo e também na escolha de um limiar de classificação (quando necessário).

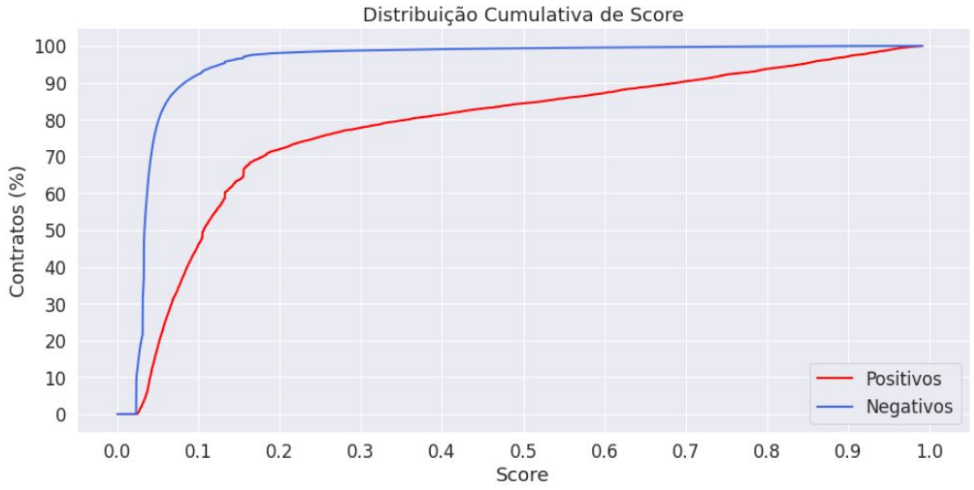

Figura 2 - Exemplo de resultado da análise de score

Curva ROC: Já a curva ROC permite entender como as taxas de Falso Positivo (FP) e Verdadeiro Positivo (TP) variam de acordo com a escolha do limiar. Para cada limiar escolhido, são obtidos valores diferentes para o FP e o TP. Quanto menor o limiar, maior a taxa de TP porém maior a taxa de FP. Quanto maior o limiar, menor a taxa de FP porém menor a taxa de TP. Portanto, existe um compromisso entre a escolha do limiar e o valor destas duas taxas. A área debaixo da curva é justamente o valor da AUC, que é usada como métrica de qualidade do modelo.

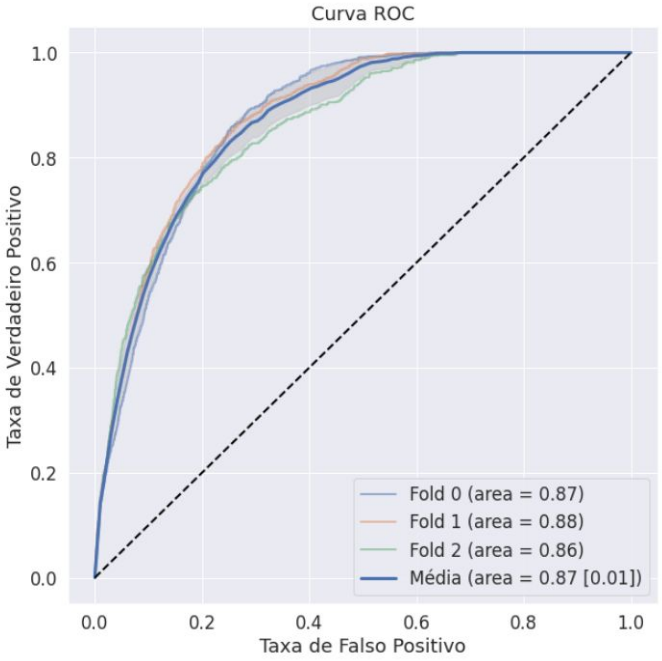

Figura 3 - Exemplo de resultado da curva ROC

Gráfico de calibração: Esta visualização tem como objetivo avaliar a qualidade da calibração através da curva de calibração. Esta curva estima a fração de exemplos positivos para cada valor de predição dada pelo modelo. Em uma cenário ideal, esta curva seguiria a função identidade, ou seja, a probabilidade dada pelo modelo corresponde à frequência com que a classe positiva ocorre. Como exemplo, dentre os casos em que o modelo atribui 70% de probabilidade, deseja-se que aproximadamente 70% deles sejam de fato positivos, e os demais 30% sejam negativos. Quanto mais longe a curva de calibração estiver da curva de identidade, pior a calibração.

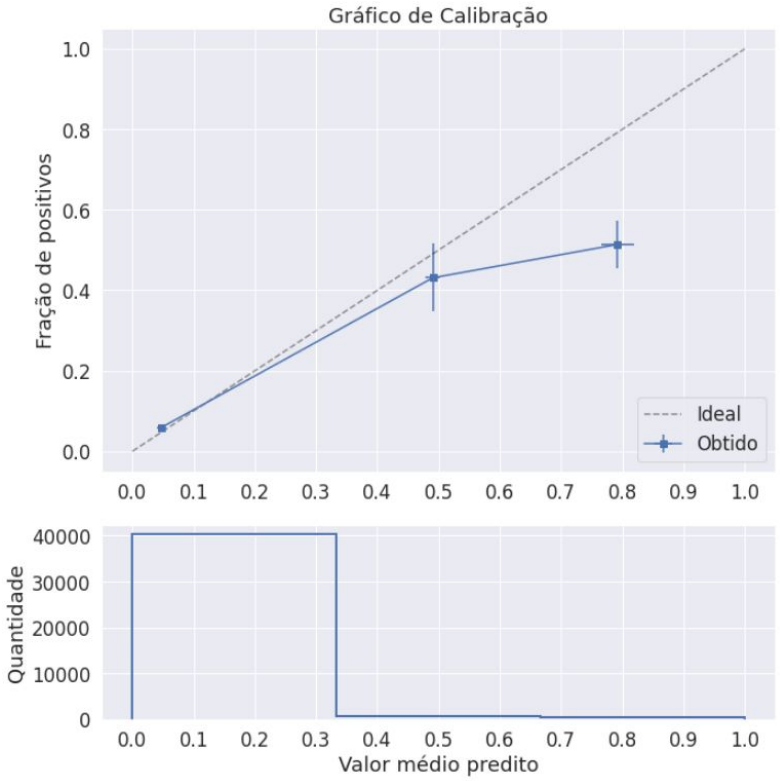

Figura 4 - Exemplo de resultado do gráfico de calibração

Complementar ao notebook de métricas estão as três análises adicionais, que são executadas no pipeline de análise: explicabilidade (utilizando o método SHAP), LCA e MSA.

A análise de explicabilidade, feita pelo método SHAP, consiste em estimar a importância de cada feature na predição de cada exemplo. Mais informações sobre esse método podem ser encontradas na trilha de capacitação na seção 3.2.1.

O resultado do node `calculate_shap_values` é uma estimativa do *shap value* para cada feature de cada exemplo predito no modelo em cada partição e fold temporal. Estes números são salvos em uma tabela e podem ser visualizados no notebook em `notebooks/shap_plot.ipynb`. Este notebook implementa duas visualizações:

SHAP summary: Este gráfico mostra um resumo dos valores do SHAP considerando todos os exemplos do OOS. Cada exemplo é representado como uma bolinha no gráfico e cada feature do modelo como uma linha no eixo vertical. A cor de cada bolinha indica o valor daquela feature para aquele exemplo. Valores mais altos são indicados em vermelho enquanto valores mais baixos são indicados em azul (no caso de features categóricas, o vermelho indica presença enquanto o azul indica ausência daquela categoria). Já a posição de cada bolinha no eixo horizontal indica o impacto que aquela feature teve para aquele exemplo, sendo que a barra vertical cinza indica o "zero" (ponto onde não há impacto na predição). Bolinhas à esquerda da barra reduziram a probabilidade dada pelo modelo, enquanto bolinhas à direita aumentaram a

probabilidade dada pelo modelo. As features estão ordenadas pelo impacto total no modelo de forma decrescente, de cima para baixo. Em geral são plotadas as N features com maior impacto, para evitar gráficos muito grandes.

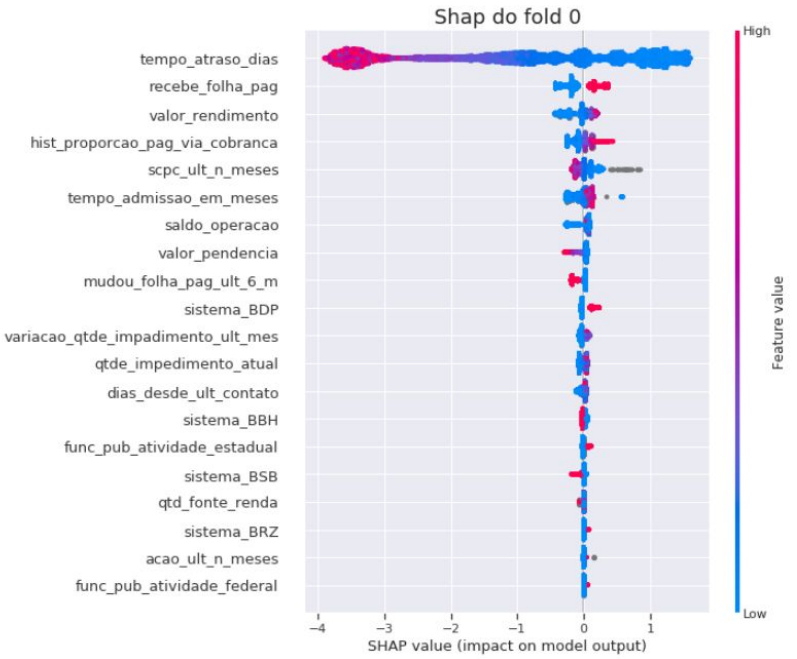

Figura 5 - Exemplo de resultado do SHAP summary

SHAP individual: Já o gráfico de SHAP individual mostra os valores do shap para um exemplo específico, indicando como cada feature impactou na predição de um exemplo. Neste caso, as features estão ordenadas pelo impacto naquele exemplo. A linha vertical cinza indica o valor da predição caso nenhuma feature fosse conhecida, que serve de referência. Já a linha vertical em azul indica o resultado da predição para aquele exemplo na parte superior e o quanto cada feature afetou a predição para mais ou para menos. Os valores entre parênteses são os valores das features em si. Um detalhe importante é que o valor mostrado neste gráfico corresponde à predição antes da calibração, portanto será diferente do valor da predição após a calibração.

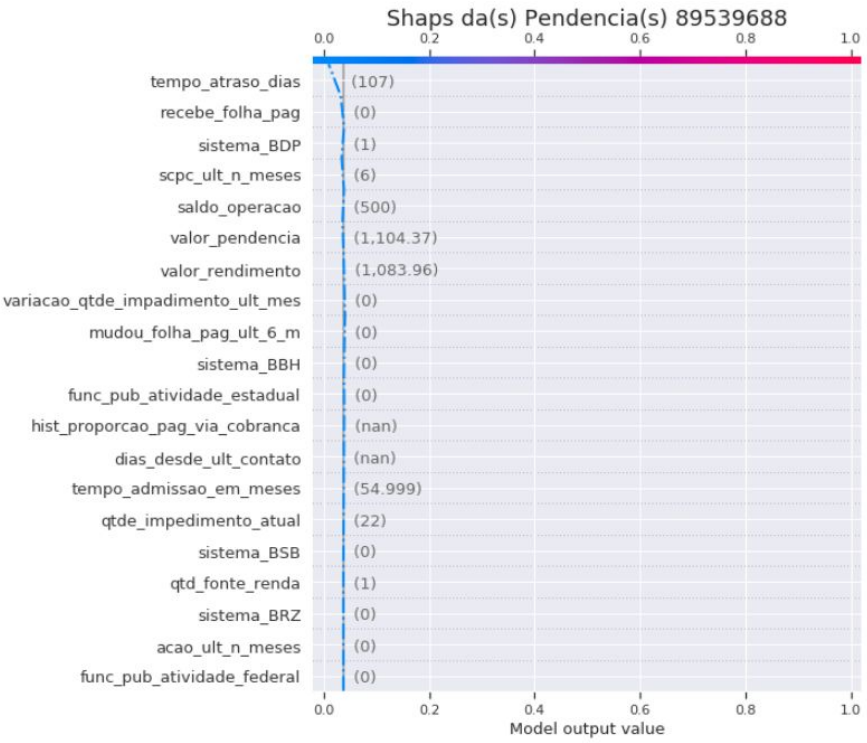

Figura 6 - Exemplo de resultado do SHAP individual

Análise complementar do LCA (Learning Curve Analysis): O objetivo do LCA é entender como o número de exemplos utilizados durante o treino afeta as métricas de qualidade do modelo em cada partição. Isto é feito treinando múltiplos modelos variando a porcentagem do in sample utilizada, por exemplo, aumentado de 10% em 10% por iteração. Esta análise traz alguns insights importantes, como por exemplo, se a adição de mais exemplos poderia melhorar o resultado do modelo, que irá depender da inclinação da curva do OOS em função da proporção usada no treino. Ela também permite visualizar o grau de *overfitting* do modelo, baseado na distância entre as curvas de treino e OOS. Outra observação importante que pode ser feita através do LCA é o número mínimo de exemplos necessários para se obter um resultado bom, em relação ao máximo obtido no OOS, que pode então ser usado como tamanho do grupo de controle. Mais detalhes sobre o LCA podem ser encontrados no curso de Machine Learning na seção 2.2.1 da trilha de capacitação.

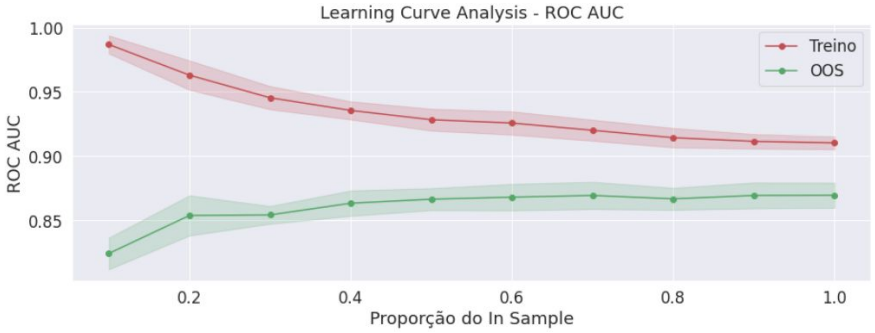

Figura 7 - Exemplo de resultado do LCA

Análise complementar do MSA (Model Stability Analysis): O objetivo do MSA é estimar como as métricas de qualidade do modelo variam à medida em que o período avaliado se afasta do período utilizado para treino. Ele é feito avaliando o resultado do modelo separadamente para períodos disjuntos consecutivos dentro do período completo de OOS. O resultado do MSA é então facilmente visualizado graficamente, no qual é possível entender se a performance cai de forma significativa quando os exemplos considerados se distanciam temporalmente da data do treino. A principal aplicação desta análise é estimar quanto tempo o modelo pode permanecer em produção sem que haja uma perda de qualidade significativa e também para planejar o re-treino.

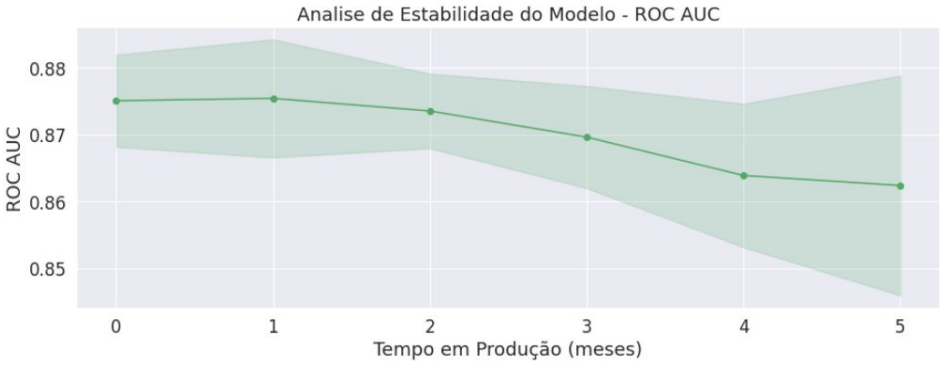

Figura 8 - Exemplo de resultado do MSA

Por fim, outra forma comum de avaliar o modelo é inspecionando os exemplos individualmente ou em grupos, principalmente os que foram classificados incorretamente. É interessante visualizar tanto a predição dada pelo modelo, quanto os valores das features daquele exemplo, quanto os valores individuais do SHAP. Isto pode ser feito facilmente de forma exploratória no notebook de resultados (`notebooks/reporting.ipynb`), que já possui a leitura das predições, features e SHAP. Com isto é possível perceber pontos fracos do modelo, possíveis erros de implementação, oportunidades de melhoria de features e criação de novas, identificar vieses do modelo ou mesmo insights de negócio sobre como melhor modelar o problema, qual target utilizar, qual horizonte de monitoramento utilizar, etc.

4. Ciclos de Modelagem Sugeridos

Esta seção tem como objetivo explicar como realizar ajustes no modelo e prover uma lista de sugestões de ciclos de modelagem assistidos para praticar as situações mais comuns de modelagem. Por ciclo de modelagem, entende-se o processo de realizar uma alteração no procedimento de backtest – seja adicionando features, mudando o modelo ou mudando a forma de avaliação – e re-avaliar os resultados por meio das métricas e análises de qualidade. Os ciclos sugeridos não irão necessariamente melhorar o modelo, dado que o objetivo principal é mostrar possíveis formas de melhorar os resultados em backtest e também pelo fato dos modelos já terem sido bastante trabalhados.

4.1. Ajustes no Modelo

A seguir serão explicados os principais ajustes que podem ser feitos no modelo com o objetivo de melhorar os resultados em backtest e como são feitos.

4.1.1. Método de Machine Learning

Um dos principais fatores que afetam o resultado de uma modelagem é a escolha do método de Machine Learning para realizar a classificação. Portanto, é importante conhecer os métodos existentes e as implementações disponíveis.

Em geral não é possível dizer a priori qual método terá o melhor resultado para determinado problema, sendo necessário realizar experimentos para verificar empiricamente. Na prática, para problemas de classificação de dados estruturados, alguns métodos mais avançados e mais robustos tendem a ter resultados melhores, mais especificamente, os métodos de ensemble baseados em árvores, como Random Forest e Gradient Tree Boosting.

Para configurar o método de Machine Learning no backtest, basta alterar o parâmetro `model.classifier.estimator.name`, que especifica o método. Os métodos disponíveis para uso ficam configurados dentro de `model/model.py` no dicionário `ESTIMATORS`, onde o `name` é a chave do dicionário. Cada método precisa estar implementado em uma classe que segue o padrão de API scikit-learn, de forma que eles possam ser utilizados dentro de pipelines do Scikit-learn. Para adicionar mais uma opção de método, basta adicionar mais uma entrada no `ESTIMATORS` colocando um nome e a classe que implementa o método. Mais informações sobre a API padrão do scikit-learn podem ser encontradas na documentação no link abaixo:

→ https://scikit-learn.org/stable/developers/develop.html#apis-of-scikit-learn-objects

Segue abaixo um exemplo de como ficaria a configuração para o método Random Forest.

```
backtest_model:
```

```
  classifier:
    estimator:
      name: “RandomForestClassifier”
```

4.1.2. Hiperparâmetros

Os hiperparâmetros são bastante importantes pois afetam significativamente a performance dos métodos de ML. Os hiperparâmetros dependem de cada método e da implementação usada, assim como o tipo de impacto que eles têm no modelo. Por isso é importante conhecer o método e a implementação utilizada.

Os hiperparâmetros são especificados em `model.classifier.estimator.params`, que são passados para o construtor da classe do método de ML escolhido. Os parâmetros específicos de cada classe podem ser encontrados na documentação da biblioteca de origem. Parâmetros não especificados terão o valor default de acordo com a biblioteca.

Segue abaixo um exemplo de como ficaria a configuração para hiperparâmetros do Random Forest.

```
backtest_model:
  classifier:
    estimator:
      name: “RandomForestClassifier”
      params:
        n_estimators: 50
        max_depth: 3
```

4.1.3. Otimização de Hiperparâmetros

Uma abordagem bastante utilizada para melhorar o resultado do modelo é utilizar um método de otimização de hiperparâmetros. Tais métodos testam várias combinações de hiperparâmetros de forma a encontrar aquela que tenha o melhor resultado no conjunto de validação.

O otimizador de hiperparâmetros, que é opcional, é especificado em `model.classifier.optimizer`, onde `name` especifica o método e `params` os parâmetros da classe que implementa o método. Os métodos de otimização são especificados em `model/model.py` no dicionário `OPTIMIZERS`, novamente seguindo a API padrão do Scikit-learn. Após a busca por hiperparâmetros, o modelo é reajustado utilizando os melhores valores de hiperparâmetros encontrados na busca. Os parâmetros específicos de cada classe podem ser encontrados na documentação da biblioteca de origem.

Segue abaixo um exemplo de como ficaria a configuração para a otimização de hiperparâmetros do Random Forest utilizando o Random Search.

```
backtest_model:
  classifier:
    optimizer:
      name: “RandomizedSearchCV”
      params:
        param_distributions:
          n_estimators: [20, 30, 40, 50, 60, 70, 80, 90, 100]
          max_depth: [2, 3, 4, 5, 6]
        n_iter: 20
    estimator:
      name: “RandomForestClassifier”
      params:
        min_samples_split: 4
```

4.1.4. Pré-processamento

Outro fator importante da modelagem são as etapas de pré-processamento das features. Estas etapas estão bastante ligadas à escolha do método de ML. Alguns métodos precisam de dados padronizados, como é o caso do SVM, enquanto outros não. Outro exemplo são dados faltantes, que são tratados por alguns métodos porém não por todos. E de forma geral, essas etapas de pré-processamento podem melhorar os resultados do modelo.

No pipeline desenvolvido foram adicionados três pré-processamentos possíveis, todos opcionais. De forma análoga à especificação do método de Machine Learning, cada pré-processamento é especificado pelo `name` do método e os `params` necessários. O primeiro é o `model.imputer`, o segundo o `model.scaler`, e o terceiro o `model.feature_selection`, no qual é possível especificar uma lista de seletores de features aplicados em sequência. Os pré-processamentos disponíveis estão configurados nos dicionários `IMPUTERS`, `SCALERS` e `FEATURE_SELECTORS` respectivamente dentro de `model/model.py`. Também de forma análoga ao classificador, eles seguem o padrão de API do scikit-learn.

Segue abaixo um exemplo de como ficaria a configuração das etapas de pré-processamento com o Random Forest.

```
backtest_model:
  imputer:
    name: “SimpleImputer”
    params:
      strategy: “mean”
  scaler:
    name: “StandardScaler”
    params:
  feature_selection:
```

```
    -  name: “VarianceThreshold”
       params:
         threshold: 0
  classifier:
    estimator:
      name: “RandomForestClassifier”
```

4.1.5. Método e Parâmetros de Calibração

O método e parâmetros de calibração são importantes para garantir que as predições sejam calibradas ao utilizar modelos que não tem predições calibradas por construção, como é o caso do XGBoost.

O método de calibração, que é opcional, é especificado em `model.classifier.calibrator`, onde `name` especifica o método e `params` os parâmetros da classe que implementa o método. Novamente de forma análoga, os métodos de calibração são especificados em `model/model.py` no dicionário `CALIBRATORS`, seguindo a API padrão do Scikit-learn. Os parâmetros específicos de cada classe podem ser encontrados na documentação da biblioteca de origem.

Segue abaixo um exemplo de como ficaria a configuração do calibrador com o Random Forest.

```
backtest_model:
  classifier:
    calibrator:
      name: “CalibratedClassifierCV”
      params:
        method: “sigmoid”
    estimator:
      name: “RandomForestClassifier”
```

4.1.6. Remoção de Features

Eventualmente pode ser necessário remover alguma feature do modelo. Isso pode ser necessário por exemplo quando uma feature perdeu relevância no modelo, quando uma regra de negócio mudou e a feature deixou de ter sentido, ou mesmo para avaliar o impacto daquela feature.

Uma forma mais rápida de remover uma feature é adicionando o nome da mesma no parâmetro `features_drop`, de forma que ela seja removida durante a função `features_merge`. Fora isto, também é possível obviamente remover o cálculo da feature específica da função correspondente ou mesmo removendo o node inteiro do pipeline.

Segue abaixo um exemplo de como remover features pelo parâmetro.

```
backtest_features_drop:
  - “scpc_ult_n_meses”
  - “sexo”
```

4.1.7. Adição de Features

Adicionar novas features pode melhorar o poder preditivo do modelo ao permitir que ele tenha acesso a mais informações que separem os exemplos de cada classe. O ideal é que as features sejam o mais discriminativas possível, ou seja, tenham faixas de valores bastante distintas para exemplos de classes diferentes (tanto individualmente quanto em conjunto). A escolha de quais features adicionar é orientada por vários fatores, como conhecimento e experiência no problema de negócio em questão, aprimoração de features já existentes e que têm relevância, análises de exemplos errados pelo modelo para identificar informações que poderiam distinguí-los, ou mesmo pesquisando na literatura.

Para adicionar uma feature no modelo é necessário primeiramente implementar o cálculo da mesma. Para isto sugerimos primeiro implementar e validar o cálculo em um notebook, adicionar esta feature no modelo, e verificar os resultados do backtest. A adição no modelo é feita adicionando o dataset com a features como entrada do node de `features_merge` e adicionando a junção do DataFrame com as features implementadas com as demais. Feito o teste da feature, o cálculo pode ser então passado para um arquivo Python, um node criado para ela e então incorporada no pipeline.

Vários exemplos de implementação de features podem ser encontrados nas funções dentro do modelo de `features` do repositório.

4.2. Sugestões de Experimentos

A seguir estão listadas algumas sugestões de experimentos para fixar os conceitos apresentados e entender na prática como realizar ciclos de modelagem. Fique à vontade para realizar outros experimentos que julgar relevantes.

4.2.1. Avaliar Configuração Atual

O primeiro passo seria apenas executar o pipeline de backtest utilizando a configuração atual salva no repositório e verificar os resultados. É importante ter em mente o resultado atual para que seja possível mensurar o impacto das modificações que serão feitas nas próximas etapas.

4.2.2. Experimento com Hiperparâmetros do XGBoost

Experimente alterar os valores dos hiperparâmetros `n_estimators` e `max_depth` do XGBoost, dois parâmetros que possuem bastante impacto no desempenho do método. Em particular, verifique como as métricas na validação e o resultado do LCA variam ao alterar estes dois parâmetros.

Informações sobre os parâmetros podem ser encontradas na documentação do XGBoost.

→ https://xgboost.readthedocs.io/en/latest/python/python_api.html#xgboost.XGBClassifier

4.2.3. Experimento Adicionando Padronização

Experimente adicionar uma etapa de padronização das features (scaler) para o XGBoost, como por exemplo a padronização por média e desvio padrão (StandardScaler), e verifique se houve impacto nos resultados. Repare que agora os valores que serão mostrados no gráfico do SHAP também estarão padronizados.

Informações sobre o StandardScaler podem ser encontradas na documentação do Scikit-learn.

→ https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

4.2.4. Experimento Adicionando Imputação

Experimente adicionar uma etapa de imputação das features (imputer) para o XGBoost, como por exemplo a imputação utilizando a média (SimpleImputer), e verifique se houve impacto nos resultados. Repare que agora o gráfico do SHAP não terá mais os valores faltantes, que foram substituídos pela média.

Informações sobre o SimpleImputer podem ser encontradas na documentação do Scikit-learn.

→ https://scikit-learn.org/stable/modules/generated/sklearn.impute.SimpleImputer.html

4.2.5. Experimento com o Random Forest

Experimente trocar o método de classificação para o Random Forest (RandomForestClassifier), outro método de Machine Learning baseado em ensembles de árvores de decisão. De forma análoga ao XGBoost, dois hiperparâmetros importantes do Random Forest são o `n_estimators` e o `max_depth`. Repare que para esta implementação é necessário adicionar imputação das features, dado que ela não lida com valores faltantes.

Informações sobre o RandomForestClassifier podem ser encontradas na documentação do Scikit-learn.

→
https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

4.2.6. Experimento com o Otimização do Random Forest

Experimente utilizar o Random Search (RandomizedSearchCV) juntamente com o Random Forest para fazer uma busca de hiperparâmetros com base nos resultados na validação. É necessário especificar no RandomizedSearchCV quais hiperparâmetros devem ser otimizados e quais são os valores possíveis para cada, além do número de iterações desejadas.

Informações sobre o RandomizedSearchCV podem ser encontradas na documentação do Scikit-learn.

→
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html

4.2.7. Experimento com Calibração

Experimente variar o método de calibração dentre as duas disponíveis para o CalibratedClassifierCV: “sigmoid” e “isotonic”. Para entender a necessidade de calibração, experimente também tirar a calibração. Verifique nos resultados principalmente o Expected Calibration Error e a curva de calibração.

Informações sobre o CalibratedClassifierCV podem ser encontradas na documentação do Scikit-learn.

→
https://scikit-learn.org/stable/modules/generated/sklearn.calibration.CalibratedClassifierCV.html

4.2.8. Experimento com o Tempo de Treino e OOS

Experimente aumentar e reduzir o tempo de treino e de OOS dentro dos parâmetros do Walk Forward, e verifique como isto irá afetar as métricas de qualidade do modelo. Verifique também como isto irá afetar os resultados do LCA e do MSA (note que também é necessário alterar os parâmetros do Walk Forward do MSA).

4.2.9. Avaliação de Features por Remoção

Experimente remover as 5 features mais relevantes de acordo com o SHAP, uma por uma, para verificar qual o impacto cada uma delas terá nas métricas de qualidade ou se há muita redundância entre as features (ou seja, remover uma feature não causa um impacto grande no

modelo pois a informação presente naquela feature também está presente em outra(s)). Este teste é bastante simples porém bastante útil para entender o impacto e a relação entre as features do modelo.

4.2.10. Implementação e Adição de uma Feature Nova

Para este último experimento, o objetivo é implementar e adicionar uma feature nova no modelo. Abaixo estão listadas algumas sugestões de features que poderiam ser adicionadas no modelo com potencial de melhorar os resultados. Porém são apenas sugestões, sinta-se à vontade para implementar outras features que possam ter alto poder preditivo no modelo.

- Adicionar uma feature baseada em alguma feature atual que possua relevância no modelo. Uma forma seria adicionando uma outra versão da feature porém com uma segmentação específica, como por exemplo segmentar a feature de número de inserções no SCPC nos últimos 6 meses para subgrupos específicos. Outra forma seria calculando a tendência de uma determinada feature, como por exemplo a tendência na variação do saldo da conta corrente. Outra forma interessante seria calcular a razão entre duas features já existentes, como por exemplo a razão entre o número de operações tomadas pelo cliente e o número de instituições financeiras que o cliente opera no SFN.

- Features adicionais utilizando as informações de fonte de renda (BIG.FONTE_RENDA): Indicador se o cliente foi demitido nos últimos 6 meses; quanto tempo o cliente permaneceu no emprego; variação ou tendência no valor da fonte de renda; valor esperado da renda nos próximos 6 meses.

- Razão entre a renda esperada e o valor total de parcelas esperadas para os próximos 6 meses, a partir dos dados da BIG.FONTE_RENDA e da BIG.VENCIMENTO_SFN_BMZ. A partir desta primeira feature, calcular a tendência desta razão nos próximos 6 meses.

- Dados de refinanciamentos do cliente: número de refinaciamentos nos últimos 6 meses; valor total de refinanciamentos; proporção do valor comprometido atual que corresponde a refinanciamentos.

- Dados de investimentos do cliente: número de aplicações de investimento do cliente; valor total de investimento e/ou valor em cada tipo de investimento; tendência no valor de investimento nos últimos 6 meses.

- (Concessão) Informações sobre garantias das propostas: indicador se a proposta possui garantia; tipo da garantia; valor da garantia.

- (Concessão) Rendimento médio do grupo de atividade do cliente a partir dos dados do IBGE, associados ao cliente através da tabela SETORIZACAO_EMPREGADOR_FONTE_RENDA pelo CNAE e PNAD.

- (Pendências) Taxa de auto-cura histórica do cliente nos últimos 6 meses. Ou seja, calcular a porcentagem de vezes em que o cliente pagou uma pendência sem ser cobrado nos últimos 6 meses em relação à data de referência.

- (Pendências) Para clientes com histórico de pagamento de pendências no banco, calcular o número médio de dias de atraso até o pagamento e subtrair do número atual de dias de atraso da pendência sendo avaliada. Ou seja, considerando o histórico do cliente, faltam quantos dias até ou quantos dias já se passaram desde a data esperada de pagamento (que pode ser positivo, negativo ou zero)?

Durante a implementação de features, é importante garantir que o cálculo da feature possa ser precisamente replicado com dados novos durante a inferência. Ou seja, o cálculo não pode utilizar nenhuma informação que não esteja disponível no momento em que uma proposta/pendência nova for avaliada pelo modelo. Atente-se principalmente em cálculos que envolvem janelas temporais para que dados do futuro não sejam usados para evitar o que é conhecido como data leak (vazamento de dados).

Como exemplo do processo de implementação de uma feature, segue abaixo um passo a passo que poderia ser utilizado para implementar a feature mensal de número de cheques devolvidos nos últimos 6 meses, a partir dos dados da BIG.CHEQUE_BACEN_CLIENTE. Neste caso o cálculo da feature é adicionado ao pipeline mensal, que está apenas no repositório do modelo de concessão.

1. Criar um notebook novo para implementar o cálculo da feature e carregar os dados da tabela BIGI01.CHEQUE_BACEN_CLIENTE para um mês qualquer de teste.

```
# notebooks/feature_cheque_bacen.ipynb

[1]
import pandas as pd
from pathlib import Path
from kedro.framework.context import load_context
from kedro.config import ConfigLoader
from sqlalchemy import create_engine

context = load_context(Path.cwd().parent)
credentials = ConfigLoader(["../conf/base",
"../conf/local"]).get("credentials*", "credentials*/**")

[2]
mes_referencia = pd.to_datetime("2018-06-01")
period = (
    mes_referencia - pd.DateOffset(months=6),
    mes_referencia - pd.DateOffset(months=1),
)

engine = create_engine(credentials["bigi01_con"]["con"])
```

```
df_cheque = pd.read_sql_query(
    "SELECT SEQ_CLIENTE, DATA_REFERENCIA, QTDE_INCLUSAO_MES"
    " FROM BIGI01.CHEQUE_BACEN_CLIENTE"
    f" WHERE DATA_REFERENCIA >= TO_DATE('{period[0]:%Y-%m-%d %H:%M:%S}',
'YYYY-MM-DD HH24:MI:SS')"
    f" AND DATA_REFERENCIA <= TO_DATE('{period[1]:%Y-%m-%d %H:%M:%S}',
'YYYY-MM-DD HH24:MI:SS')",
    engine
)
engine.dispose()
```

2. Tendo os dados carregados, o cálculo da feature, para aquele mês de referência, é então implementado conforme abaixo (neste caso a implementação é bastante simples).

```
# notebooks/feature_cheque_bacen.ipynb

[3]
df_cheque = (
    df_cheque.groupby(["seq_cliente"])
    .agg({"qtde_inclusao_mes": "sum"})
    .reset_index()
)
df_cheque["data_referencia"] = mes_referencia
```

3. Uma vez que o cálculo esteja implementado, testado e revisado, ele pode então ser adicionado ao pipeline. Para isto é necessário adicioná-lo dentro de um arquivo Python (dentro do módulo de features para manter a organização do código) e dentro de uma função. Nesta função também é importante adicionar os outros elementos de código necessários, como logging, docstring, tipagem estática, parametrizar nomes de colunas, parametrizar o cálculo em si e validação dos dados de entrada. Segue abaixo como ficaria o arquivo (exemplificado para o modelo de concessão).

```
# features/features_cheque_bacen.py

import logging
from typing import Dict

import pandas as pd

from banrisul_concessao.io.data_validation import check_dataframe
from banrisul_concessao.io.datasets import DelayedFilteredSQLDataSet

logger = logging.getLogger(__name__)
```

```
def features_cheque_bacen(
    ddf_cheque: DelayedFilteredSQLDataSet,
    mes_referencia: str,
    col_names: Dict[str, str],
    feat_params: Dict[str, str],
) -> pd.DataFrame:
    """
    Calcula a feature de numero de cheques devolvidos nos ultimos N meses a
partir da tabela BIG.CHEQUE_BACEN_CLIENTE.

    Args:
        ddf_cheque: DelayedFilteredSQLTableDataSet com a tabela
BIG.CHEQUE_BACEN_CLIENTE
        mes_referencia: Mes para o calculo das features
        col_names: Dicionario com os nomes das colunas da tabela
        feat_params: Dicionario de parametros das features

    Returns:
        DataFrame de features calculadas
    """

    # Seleciona o nome das colunas
    client_col = col_names["seq_cliente"]
    time_col = col_names["data_referencia"]
    qtde_inclusao_mes_col = col_names["qtde_inclusao_mes"]

    # Seleciona os parametros das featuers
    tamanho_janela = feat_params["tamanho_janela"]
    cheque_devol_ult_n_meses = feat_params["cheque_devol_ult_n_meses"]

    # Transforma strings de data em datetime e calcula o periodo
    mes_referencia = pd.to_datetime(mes_referencia)
    period = (
        mes_referencia - pd.DateOffset(months=tamanho_janela),
        mes_referencia - pd.DateOffset(months=1),
    )

    logger.info("Lendo dados de entrada da tabela CHEQUE_BACEN_CLIENTE.")
    logger.debug(f"Período: {period[0]} a {period[1]}")
    logger.debug(f"Dataset: {ddf_cheque._describe()}")

    cols = [client_col, time_col, qtde_inclusao_mes_col]

    # Le o dado do banco de dados filtrando por data e cliente
```

```
    df_cheque = ddf_cheque.load_query(cols=cols, time_col=time_col,
period=period)

    logger.info("Iniciando cálculo de features da CHEQUE_BACEN_CLIENTE.")
    logger.debug(f"Shape do DataFrame de entrada: {df_cheque.shape}")

    # Checa se existem dados
    if df_cheque.empty:
        logger.info(
            "Não foram lidos registros da CHEQUE_BACEN_CLIENTE, finalizando
cálculo de features."
        )

        return pd.DataFrame(columns=[client_col, time_col,
cheque_devol_ult_n_meses])

    # Verifica DataFrame de entrada
    check_dataframe(
        df_cheque,
        cols,
        expected_types={
            client_col: "int64",
            time_col: "datetime64",
            qtde_inclusao_mes_col: "int64",
        },
        cols_not_null=cols,
        cols_non_negative=[qtde_inclusao_mes_col],
    )

    logger.info("Calculando soma de inclusões de cheques devolvidos nos
ultimos N meses.")

    # Calcula soma de inclusoes por mes e cliente
    df_cheque = (
        df_cheque.groupby([client_col])
        .agg({qtde_inclusao_mes_col: "sum"})
        .reset_index()
    )
    df_cheque[time_col] = mes_referencia

    # Renomeia coluna para nome da feature
    df_cheque = df_cheque.rename(columns={qtde_inclusao_mes_col:
cheque_devol_ult_n_meses})

    logger.info("Finalizando cálculo de features da CHEQUE_BACEN_CLIENTE.")
```

```
    logger.debug(f"Shape do DataFrame de saída: {df_cheque.shape}")

    return df_cheque
```

4. Com a função implementada, basta então adicionar, na configuração do projeto, os nomes das colunas, os parâmetros das features, o(s) dataset(s) de entrada e o dataset de saída, mostrados abaixo respectivamente.

```
# conf/base/parameters/column_names.yml

col_names_big_cheque_bacen_cliente:
  seq_cliente: "seq_cliente"
  data_referencia: "data_referencia"
  qtde_inclusao_mes: "qtde_inclusao_mes"
```

```
# conf/base/parameters/features.yml

extraction_features_cheque_bacen:
  tamanho_janela: 6
  cheque_devol_ult_n_meses: "cheque_devol_ult_6_meses"
```

```
# conf/base/catalog_dev/mensal/01_raw_big.yml

raw_big_cheque_bacen_cliente:
  type: banrisul_concessao.io.datasets.DelayedFilteredSQLDataSet
  credentials: bigi01_con
  load_args:
    table_name: cheque_bacen_cliente
    schema: BIGI01
```

```
# conf/base/catalog_dev/mensal/04_features.yml

features_cheque_bacen_mensal:
  type: banrisul_concessao.io.datasets.DelayedFilteredSQLDataSet
  credentials: bigi01_con
  save_args:
    table_name: feat_cheque_bacen
    schema: BIGI01
    mode: "append"
    index: False
```

5. Depois de adicionar a configuração necessária, o node do cálculo pode então ser criado dentro do pipeline mensal conforme abaixo.

```
# pipelines/monthly.py

node(
    features_cheque_bacen,
    [
        "raw_big_cheque_bacen_cliente",
        "params:mes_referencia",
        "params:col_names_big_cheque_bacen_cliente",
        "params:extraction_features_cheque_bacen",
    ],
    "features_cheque_bacen_mensal",
    name="features_cheque_bacen",
),
```

6. Feito isso é necessário inserir as features dentro do node de features_merge para que elas sejam utilizadas no backtest para aferir a sua relevância, conforme abaixo (exemplificado para o modelo de concessão nos trechos de código relevantes; neste caso também seria necessário atualizar o conf local).

```
# features/features_merge.py
# ...

def features_merge(
    ddf_primary_entities: DelayedFilteredSQLDataSet,
    df_primary_timeseries: pd.DataFrame,
    ddf_features_cheque_bacen: DelayedFilteredSQLDataSet,

# ...

logger.info("Realizando junção de features de cheque bacen.")

df_features_cheque_bacen = ddf_features_cheque_bacen.load_query(
    time_col=time_col, period=period
)
df_features = df_features.merge(
    df_features_cheque_bacen, on=[client_col, time_col], how="left",
)
del df_features_cheque_bacen

# ...
```

```
# conf/base/catalog_dev/backtest/04_features.yml

features_cheque_bacen_backtest:
  type: banrisul_concessao.io.datasets.DelayedFilteredSQLDataSet
  credentials: bigi01_con
  load_args:
    query: "SELECT * FROM BIGI01.FEAT_CHEQUE_BACEN WHERE SEQ_CLIENTE IN
(SELECT DISTINCT SEQ_CLIENTE FROM BWAI01.BACKTEST_PRI_ENTIDADES_MODELO)"
```

```
# pipelines/backtesting.py
# ...

node(
    features_merge,
    [
        “primary_entities_backtest”,
        “primary_timeseries_backtest”,
        “features_cheque_bacen_backtest”,

# ...
```

7. E por fim, executar o cálculo da feature para os meses necessários no backtest e então executar o pipeline de backtest.

```
# bash

kedro run --pipeline monthly -n features_cheque_bacen --params
mes_referencia:2018-06
kedro run --pipeline monthly -n features_cheque_bacen --params
mes_referencia:2018-07
kedro run --pipeline monthly -n features_cheque_bacen --params
mes_referencia:2018-08

# ...

kedro run --pipeline monthly -n features_cheque_bacen --params
mes_referencia:2020-10


kedro run --pipeline backtesting
```