PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

NÚCLEO DE EDUCAÇÃO A DISTÂNCIA

Pós-graduação *Lato Sensu* em Ciência de Dados e Big Data

Fábio Daros de Freitas

PREDIÇÃO DE ATRASO NA ENTREGA DE DECLARAÇÕES: UM ESTUDO NA DCTF

Belo Horizonte

2022

Fábio Daros de Freitas

PREDIÇÃO DE ATRASO NA ENTREGA DE DECLARAÇÕES: UM ESTUDO NA DCTF

Trabalho de Conclusão de Curso apresentado ao Curso de Especialização em Ciência de Dados e Big Data como requisito parcial à obtenção do título de especialista.

Belo Horizonte
2022

SUMÁRIO

Página

1 Introdução

1.1 Contextualização

Nações modernas valem-se da arrecadação de tributos como uma das suas principais fontes de financiamento do estado. Nesse contexto, podemos considerar o pacto social que viabiliza a aceitação de uma tributação adequada como uma importante e valiosa conquista destas sociedades.

No Brasil, o Sistema Tributário Nacional é organizado pelo Código Tributário Nacional (CTN), na forma da Lei nº 5.172, de 25 de outubro de 1966, que "define as normas gerais de direito tributário aplicáveis à União, aos Estados, ao Distrito Federal e aos Municípios, sem prejuízo da respectiva legislação complementar, supletiva ou regulamentar." (BRASIL, 1966)

O art. 3º do CTN define que "tributo é toda prestação pecuniária compulsória, em moeda ou cujo valor nela se possa exprimir, que não constitua sanção de ato ilícito, instituída em lei e cobrada mediante atividade administrativa plenamente vinculada." Os tributos estão divididos em (i) impostos, (ii) taxas, (iii) contribuições de melhoria, e (iv) contribuições especiais.

O art. 16 do CTN dispõe que *imposto* "é o tributo cuja obrigação tem por fato gerador uma situação independente de qualquer atividade estatal específica, relativa ao contribuinte." As *taxas* são definidas no art. 77 do CTN como tendo "como fato gerador o exercício regular do poder de polícia, ou a utilização, efetiva ou potencial, de serviço público específico e divisível, prestado ao contribuinte ou posto à sua disposição." As *contribuições de melhoria* são determinadas neste mesmo artigo como sendo "decorrente de obras públicas." Por fim, as *contribuições especiais* são definidas no art. 149 da Constituição Federal de 1988 (CF/88), e dividem-se em contribuições sociais, de intervenção no domínio econômico, e de interesse das categorias profissionais ou econômicas; ainda, o art. 195 da CF/88 dispõe que a seguridade social também será financiada pelas contribuições sociais do empregador (incidentes sobre a folha de salários, a receita ou faturamento, e o lucro), do trabalhador e dos demais segurados

da previdência social, sobre a receita de concursos de prognósticos, e sobre o valor das importações de bens e serviços (BRASIL, 2022).

O art. 113 do CTN define as obrigações tributárias como sendo do tipo *principal*, quando decorre do fato gerador do tributo; ou *acessória*, quando decorre da legislação tributária e tem por objeto as prestações, positivas ou negativas, nela previstas no interesse da arrecadação ou da fiscalização dos tributos. O mesmo artigo estabelece também que a obrigação acessória, pelo simples fato da sua inobservância, converte-se em obrigação principal relativamente à penalidade pecuniária.

As Administrações Tributárias (AT) dos entes federativos das três esferas de governo são o conjunto de órgãos e entidades estatais responsáveis pela gestão do cumprimento das obrigações tributárias, incluindo a realização de atividades de cobrança e fiscalização do pagamento de tributos (MAZZA, 2018).

Uma gestão eficaz do cumprimento das obrigações tributárias contribui para minimizar o chamado *gap tributário*[1] (*tax gap*) e aumentar a eficiência do estado, beneficiando toda a sociedade (MATOS, 2021; PEDROSA; MOURA, 2019; SIQUEIRA; RAMOS, 2005).

1.2 O Problema Proposto

A abordagem clássica empregada pelas AT na busca da conformidade tributária é a chamada *política tributária de imposição*, derivada do modelo proposto por Allingham e Sandmo (1972). Nesta abordagem, o agente econômico comporta-se racionalmente diante da possibilidade de pagar ou não determinado tributo, adotando a conduta que maximize seu benefício individual. Contudo, este modelo clássico não é capaz de explicar determinados comportamentos tributários de muitos agentes econômicos (GOUVÊA, 2020).

Como consequência, a gestão contemporânea das AT têm cada vez mais adotado o modelo denominado *Pirâmide de Conformidade* (*Compliance Pyramid*), proposto pela Organização para a Cooperação e Desenvolvimento Econômico (OCDE) no ano de

[1] A diferença entre a arrecadação potencial, aquela que Estado poderia estar arrecadando, e a arrecadação efetiva, aquela que realmente ingressa em seus cofres.

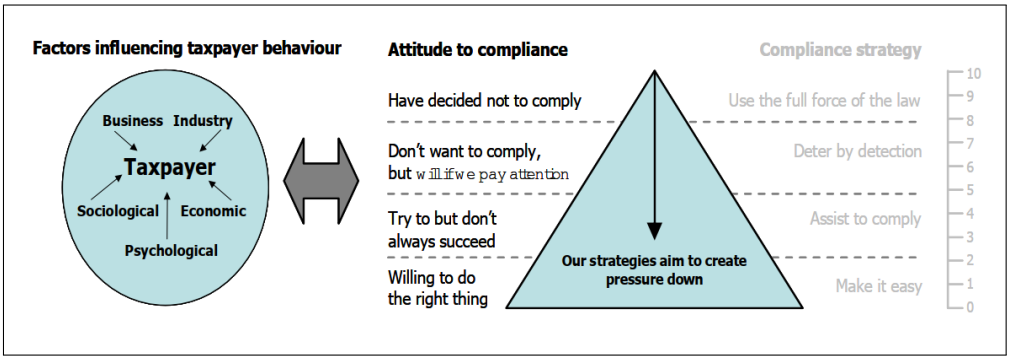


Figura 1: Pirâmide de Conformidade (OCDE, 2004)

2004 (OCDE, 2004). Trata-se de uma abordagem qualitativa, que emprega conceitos da Psicologia e Sociologia comportamental. Nesse modelo, os contribuintes são classificados em quatro grupos comportamentais, de acordo suas atitudes em relação às suas obrigações tributárias, com o objetivo de produzir respostas das AT que sejam adequadas ao comportamento característico de cada grupo, conforme ilustrado na Figura 1. Estes grupos são formados pelos seguintes perfis, correspondentes aos níveis da pirâmide: (1) *apoiadores* (*supporters*, base da pirâmide), aqueles que realmente querem fazer a coisa certa e vão se esforçar para isso; (2) *experimentadores* (*triers*), os que tentam cumprir com suas obrigações mas às vezes encontram dificuldades em fazê-lo; (3) *resistentes* (*resisters*), os que não querem cumprir suas obrigações mas o farão se o fisco os dissuadir; e (4) os *desatrelados* (*the disengaged*, topo da pirâmide), os que definitivamente decidiram não cumprir com suas obrigações.

Para cada grupo, as respostas das AT devem incluir, respectivamente, (1) facilitação; (2) assistência no cumprimento da obrigação; (3) dissuasão pela detecção; e (4) emprego de toda a força da lei (OCDE, 2004).

Desta forma, as AT enfrentam as diferentes atitudes em relação ao cumprimento das obrigações tributárias (*attitude to compliance*) por meio de estratégias de conformidade (*compliance strategy*) distintas, com o objetivo de pressionar a maior quantidade de contribuintes para a base da pirâmide, onde as soluções têm menor custo e, portanto, resultam em ganho de eficiência na gestão tributária (GOUVÊA, 2020). Melhorar a gestão tributária é um caminho eficaz para e aumentar a eficiência do estado.

Na esfera federal, a Secretaria Especial da Receita Federal do Brasil (RFB) é o órgão responsável pela administração dos tributos de competência da União, inclusive

os previdenciários e aqueles incidentes sobre o comércio exterior, abrangendo, assim, parte significativa das contribuições sociais do País (RFB, 2022e).

O modelo de arrecadação da RFB tipicamente emprega obrigações acessórias na forma de declarações a serem preenchidas e entregues ao fisco pelos contribuintes em prazos específicos. Nesse contexto, a *declaração* é a prestação de informações relevantes inerentes a um ou mais tributos cujo contribuinte é o sujeito passivo[2] da obrigação principal, tipicamente preenchida e enviada por meio de sistema eletrônico disponibilizado pela RFB. Se a declaração for enviada após o prazo legal, será cobrada a *Multa por Atraso na Entrega da Declaração* (MAED).

1.2.1 Predição de Atraso na Entrega de Declarações

Predizer se um contribuinte não entregará uma declaração no prazo legal pode ajudar as Administrações Tributárias a implementarem estratégias bem sucedidas para dissuasão por detecção, como resposta aos indivíduos classificados no grupo resistentes do modelo da Pirâmide de Conformidade. Iniciativas desta natureza podem produzir impactos positivos em indicadores e processos de trabalho envolvendo a conformidade dos contribuintes sob a sua administração.

Nesse contexto, o problema de predição de atraso na entrega de declarações pode ser definido como:

> *Dado o conjunto de dados* $\mathbf{X}(t)$ *disponível no tempo* t*, relacionado a um tipo de declaração* $\mathcal{D}$ *de obrigação do contribuinte, com prazo de entrega* $p, p > t$*, obtenha a predição* $\hat{y}(p)$ *do valor* $y(p) = \{0, 1\}$*, que indique se o contribuinte irá atrasar,* $\hat{y}(p) = 1$*, ou não,* $\hat{y}(p) = 0$*, a entrega da declaração com vencimento no prazo* p*.*

Este é um problema de *aprendizado supervisionado* denominado *classificação binária*, onde o objetivo é atribuir a cada entrada os rótulos (*labels*) binários "Sim" (1) ou "Não" (0), de forma a identificar corretamente a classe a qual os dados de entrada pertencem (HASTIE; TIBSHIRANI; FRIEDMAN, 2009). Portanto, no problema de predição de atraso na entrega de declarações, cada entrada será classificada como

[2] A pessoa obrigada ao pagamento de tributo ou penalidade (art. 121 do CTN).

"Sim" (1) ou "Não" (0), indicando antecipadamente (predizendo) se a próxima declaração será entregue com atraso.

1.2.2 Declaração de Débitos e Créditos Tributários Federais (DCTF)

Neste trabalho, estudaremos o problema de predição de atraso na entrega de declarações com dados da Declaração de Débitos e Créditos Tributários Federais (DCTF), que é uma das principais declarações administradas pela RFB. A DCTF é uma declaração mensal das pessoas jurídicas que informa os tributos federais devidos e os correspondentes créditos para cada tributo, e constitui legalmente uma declaração e confissão de dívida tributária (RFB, 2022b; CASTRO, 1996).

Como regra geral, o prazo mensal para entrega da DCTF é o 15º (décimo quinto) dia útil do 2º (segundo) mês subsequente ao mês de ocorrência dos fatos geradores. Por exemplo, os débitos e créditos decorrentes do mês de janeiro, devem ser declarados no mês de março. Se o contribuinte obrigado por lei a entregar a declaração entregá-la após o prazo, será cobrada a Multa por Atraso na Entrega da Declaração–MAED (RFB, 2022c)

Nossos estudos abrangeram um subconjunto dos contribuintes pessoas jurídicas obrigados a entregar a DCTF mensal no período de janeiro de 2015 a outubro de 2021. Utilizamos modelos de *Machine Learning* para conduzir diversos experimentos, e seus resultados foram avaliados no escopo do problema de predição de atraso na entrega de declarações.

1.3 Objetivos

O principal objetivo deste trabalho é verificar se é possível atacar o problema de predição de atraso na entrega de declarações por meio de modelos de *Machine Learning* (ML), a partir de um conjunto de dados reais.

Para isso, empregaremos modelos de ML para realizar a predição da ocorrência de MAED na DCTF, que será utilizada como um *proxy* do atraso na entrega— uma predição positiva indica atraso na entrega—, e verificaremos seus resultados

de desempenho por meio de um conjunto de métricas adequadas aos problemas de classificação.

Este trabalho está organizado como a seguir. Após esta introdução, os conjuntos de dados (*datasets*[3]) que foram utilizados serão descritos na Seção 2, relatando suas características fundamentais, tais como origem e formatos. Em seguida, os tratamentos que foram necessários aplicar aos conjuntos de dados para adequá-los à utilização pelos modelos de ML serão apresentados na Seção 3. Na Seção 4 será apresentada uma análise exploratória do *dataset* final, que será utilizado pelos modelos de ML. Nossos modelos de ML e os métodos experimentais utilizados serão descritos na Seção 5, e nossos resultados serão apresentados e interpretados nas Seções 6 e 7, respectivamente. Por fim, nossas conclusões e apontamentos para trabalhos futuros serão apresentados na Seção 8.

2 Coleta de Dados

Conforme apresentado na Seção 1.3, este trabalho tem por objetivo empregar modelos de *Machine Learning* (ML) no problema de predição de atraso na entrega de declarações (Seção 1.2.1) aplicado à Declaração de Débitos e Créditos Tributários Federais–DCTF (Seção 1.2.2).

Nesta seção, descreveremos os dados fundamentais utilizados neste trabalho, incluindo seus processos de coleta e mescla (*merging*) para formação do nosso *dataset* inicial.

2.1 Sensibilidade dos Dados

Todos os dados que foram utilizados neste trabalho são dados administrados pela RFB, incluindo dados de declarações entregues pelos contribuintes pessoas jurídicas e de outras fontes ou sistemas, ou derivados destes, e, portanto, são protegidos por sigilo fiscal (RFB, 2022a). Desta forma, os referidos dados somente serão apresenta-

[3]Neste trabalho, os termos *conjunto de dados* e *dataset* serão utilizados indistintamente.

Data Source	Descrição	Origem
DCTF	Declaração de Débitos e Créditos Tributários Federais (DCTF)	Receitadata
EFD	Escrituração Fiscal Digital Contribuições (EFD-Contribuições)	Receitadata
EFIN	e-Financeira	Receitadata
NFE	Nota Fiscal Eletrônica (NF-e)	Receitadata
MAED	Multa por Atraso na Entrega da Declaração (MAED)	e-SICODEC

Tabela 1: *Data Sources* utilizados na coleta de dados.

dos neste trabalho, e nos arquivos do repositório, de forma descaracterizada, que não permita se identificar os contribuintes nem valores reais declarados ou registrados.

Por este mesmo motivo, também não apresentaremos os códigos-fonte completos das consultas SQL e programas desenvolvidos para extração de dados, e que operaram diretamente nas bases de dados e sistemas internos da RFB, nos reservando a apenas descrever seus funcionamentos de forma adequada quando oportuno.

2.2 Fontes de Dados

Para construir os *datasets* deste trabalho, nós utilizamos diversas fontes de dados (*data sources*) tributários e econômico-financeiros, disponibilizados pelo *lago de dados* da RFB, denominado *Receitadata*, e pelo Sistema de Controle de Declarações (e-SICODEC). Estes *data sources* são mostrados na Tabela 1. Conforme mostrado na tabela, foram utilizados cinco *data sources*, que são descritos a seguir.[4]

O *data source* DCTF contém dados relativos às DCTF entregues pelos contribuintes.

O *data source* EFD disponibiliza dados da Escrituração Fiscal Digital das Contribuições incidentes sobre a Receita (EFD-Contribuições), incluindo a Contribuição para o PIS/Pasep, Cofins, e Contribuição Previdenciária incidente sobre a Receita.

O *data source* EFIN disponibiliza dados da e-Financeira, que são um conjunto de arquivos digitais contendo a prestação de informações relativas às operações financeiras de interesse da RFB.

[4]Referências sobre EFD-Contribuições, e-Financeira, e NF-e podem ser encontradas no Sistema Público de Escrituração Digital–Sped (SPED, 2022)

O *data source* NFE disponibilizada dados do sistema Nota Fiscal Eletrônica (NF-e), que contém o conjunto de notas fiscais (compra e venda) relativas aos contribuintes.

Por fim, o *data source* MAED fornece as ocorrências da Multa por Atraso na Entrega da Declaração na DCTF.

2.3 Ambiente de Desenvolvimento

A identificação dos *data sources* que forneceram os dados para este estudo norteou a escolha de um ambiente híbrido para o seu desenvolvimento.

O ambiente Receitadata (RD) é um *data lake* baseado nas plataformas Hadoop, Jupyter e Hue (APACHE, 2022; JUPYTER, 2022; HUE, 2022), rodando Python versão 3.6.8 com Anaconda versão 4.6.9. É neste ambiente que estão abrigados quase a totalidade do volume de dados que utilizamos neste trabalho.

Contudo, apesar de oferecer todo o poder computacional e memória de um *cluster* Hadoop de grande porte, o ambiente Receitadata não nos permitiu prontamente efetuar a instalação de pacotes Python, devido a aspectos corporativos. Desta forma, optamos por utilizar este ambiente apenas para a extração dos *data sources* DCTF, EFD, EFIN, e NFE (ver Tabela 1) e geração do nosso *dataset* inicial.

Nosso segundo ambiente de desenvolvimento foi um notebook com processador Intel(R) Core(TM) i5-8350U CPU @ 1.90GHz, 8 GB de RAM, HD SSD de 256GB, e sistema operacional Windows 10 com Python versão 3.7.3 e Anaconda versão 4.11.0.

Este ambiente foi utilizado para desenvolver todos nossos modelos de *Machine Learning* em Python. Também utilizamos este ambiente para extrair o *data source* MAED, por meio de um *web crawler* para o sistema e-SICODEC que implementamos em Python no ambiente ContaÁgil (FIGUEIREDO, 2010), uma poderosa e importante plataforma de dados desenvolvida pela RFB.

Todos os programas desenvolvidos para a execução deste trabalho foram implementados na linguagem Python, e as principais bibliotecas (*packages*) utilizadas estão listadas na Tabela 2.

	Versão			
Package	RD	Win10	Aplicação	Página Web
Pandas	1.1.5	0.24.2	Manipulação e análise de dados	`https://pandas.pydata.org/`
Numpy	1.18.0	1.16.2	Computação numérica e científica	`https://numpy.org/`
scikit-learn	0.20.4	0.20.3	*Machine Learning*	`https://scikit-learn.org/`
seaborn	0.11.2	0.9.0	Estatística e visualização de dados	`https://seaborn.pydata.org/`
statsmodels	0.12.2	0.9.0	Modelagem estatística	`https://www.statsmodels.org/`
PyTorch	—	1.10.1	*Machine Learning* (redes neurais)	`https://pytorch.org/`
Matplotlib	3.3.0	3.0.3	Visualização de dados	`https://matplotlib.org/`

Tabela 2: Principais bibliotecas Python (*packages*) utilizadas neste trabalho.

2.4 Geração do *Dataset* Inicial

A construção dos *datasets* deste trabalho foi conduzida em duas etapas: primeiro, realizamos a geração do *dataset* inicial, contendo o *merge* dos *datasets* parciais gerados a partir da extração individual dos *data sources* da Seção 2.2, que será apresentada nesta seção. Em seguida, realizamos a geração do *dataset* final, que foi utilizado pelos modelos de *Machine Learning*, que será apresentada mais adiante na Seção 3.

Como consequência, as etapas do processo de ETL (*extract, transform, and load*) deste trabalho também ficaram distribuídas nestas duas seções.

Esta escolha se deu principalmente por conta do nosso ambiente híbrido de desenvolvimento (Seção 2.3), onde, uma vez que não executaríamos nossos modelos no ambiente Receitadata, optamos por transportar o menor volume de dados possível para o ambiente Windows 10.

2.4.1 Contextualização dos Dados

Neste trabalho, delineamos nosso universo inicial de estudo a 8392 grandes contribuintes, ou *contribuintes diferenciados*, que foram obrigados a declarar a DCTF mensal no ano de 2020.

Estes contribuintes são selecionados anualmente segundo vários critérios, e recebem acompanhamento específico da RFB por meio do monitoramento da arrecadação de tributos (impostos, taxas e contribuições), análise dos setores e grupos econômicos, e tratamento prioritário das situações encontradas (RFB, 2022d).

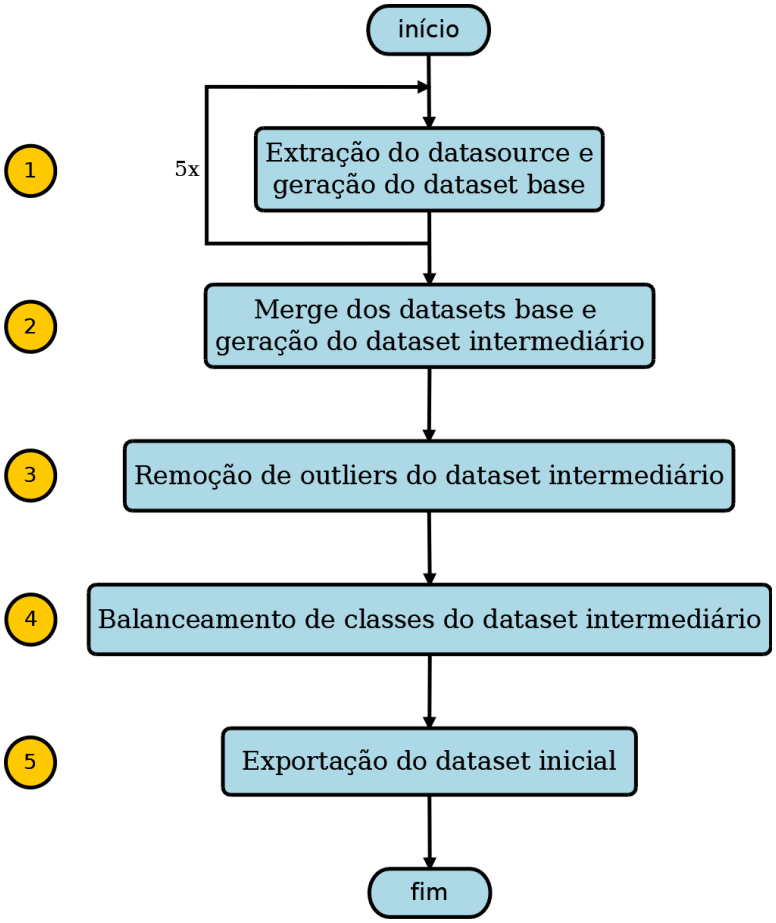


Figura 2: Fluxograma da geração do *dataset* inicial

Os CNPJ destes contribuintes foram salvos com o formato CSV (*comma-separated values*) no arquivo `TCC/CNPJ/tcc_cnpj.csv` contendo 8392 linhas. Este arquivo foi utilizado para extração inicial dos rótulos MAED do sistema e-SICODEC—este conjunto de rótulos irá nortear o restante do processo de ETL.

O processo de geração do nosso *Dataset* Inicial é ilustrado na Figura 2, e suas etapas são descritas a seguir. A saída final do processo é o *dataset* inicial. Todos os *datasets* foram implementados como objetos da classe DataFrame da biblioteca Pandas.[5]

Os conjuntos de dados que utilizamos neste trabalho estão relacionados ao projeto Raios-X DCTF, que está sendo desenvolvido na RFB, no âmbito da 7ª Região Fiscal.

[5]O DataFrame é uma classe para manipulação de dados tabulares.

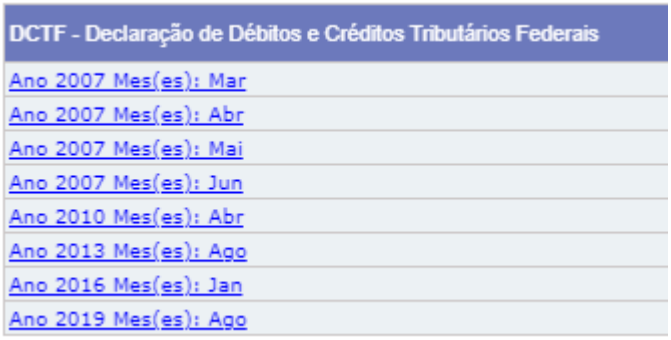


Figura 3: Extrato de tela de resposta do sistema e-SICODEC contendo as ocorrências de MAED na DCTF de um contribuinte.

2.4.2 Extração dos *Data Sources* e Geração dos *Datasets* Base

Neste etapa, como regra geral, cada *data source* da Tabela 1 dará origem a um *dataset* base homônimo; e cada *dataset* base gerado será armazenado em um arquivo CSV, para utilização pela próxima etapa do processo de ETL. O *data source* MAED será uma exceção, pois produzirá um segundo *dataset* que será utilizado apenas nesta etapa de extração.

1. ***Data Source* MAED**

Primeiramente, realizamos a extração do *data source* MAED para gerar os rótulos binários (*labels*) que indicam se contribuinte recebeu Multa por Atraso na Entrega da Declaração na DCTF de um determinado ano e mês. Esta extração foi implementada por meio de um *web crawler* desenvolvido em Python na plataforma de dados ContÁgil, que comandou 8392 consultas no sistema e-SICODEC (uma para cada contribuinte do arquivo inicial `TCC/CNPJ/tcc_cnpj.csv`).

Cada página de resposta recebida continha todos os anos e meses de ocorrência de MAED na DCTF do contribuinte. O programa, então, realizou o *parsing* do fonte HTML da página no formato da Figura 3, extraindo os dados de interesse.

Por fim, o extrator gerou como saída o arquivo `TCC/MAED/tcc_maed.csv` no formato CSV, contendo 52.582 linhas de dados no formato apresentado na Tabela 3[6]. Ou seja, os dados extraídos contém 52.582 rótulos da classe MAED “Sim”.

[6]Sempre que adequado, descreveremos os tipos dados pela sua nomenclatura elementar, tal como “string”, “float”, e “integer”.

MAED			
Gerado pelo *web crawler* do sistema e-SICODEC			
Arquivo `TCC/MAED/tcc_maed.csv`			
Linhas(52.582): duplicadas=0 nulas=0 metricas zero=0 metricas negativo=0			
Período: 1999-01 a 2021-11			
Coluna/campo	*Descrição*	*Tipo*	*Exemplo*
cnpj	CNPJ do contribuinte	string	99.999.999/9999-99
maed	Ano-mês da ocorrência de MAED	string	2021-11

Tabela 3: Formato do *dataset* MAED.

MAED_CNPJ			
Gerado pelo Jupyter Notebook Python `TCC/DATALAKE/tcc_maed.ipynb`			
Arquivo `TCC/DATALAKE/tcc_maed_cnpj.csv`			
Linhas(6558) Colunas(3)			
Tabela no Receitadata `tcc_maed_cnpj`			
Linhas(6558) Colunas(3)			
Coluna/campo	*Descrição*	*Tipo*	*Exemplo*
cnpj	CNPJ do contribuinte	string	99.999.999/9999-99
cnpj14	CNPJ do contribuinte com 14 dígitos	string	99999999999999
cnpj8	CNPJ do contribuinte com 8 dígitos	string	99999999

Tabela 4: Formato do *dataset* MAED_CNPJ.

Na sequência, fizemos *upload* deste arquivo `TCC/MAED/tcc_maed.csv` para o Receitadata, e criamos o Jupyter Notebook Python `TCC/DATALAKE/tcc_maed.ipynb`[7] para produzir uma relação de CNPJ únicos.

Este Notebook, então, importou o arquivo `TCC/MAED/tcc_maed.csv` para um DataFrame Pandas, e gerou um novo DataFrame contendo a coluna de CNPJ únicos e suas duas versões com 8 e 14 dígitos (sem pontuação), totalizando 6558 linhas, equivalente ao total de CNPJ únicos presentes no conjunto de rótulos MAED “Sim”.

As principais linhas deste trecho do Notebook são:

```
df = df_maed.copy()
df = pd.DataFrame(df.cnpj.unique())
df.rename(columns={0: 'cnpj'}, inplace=True)
df['cnpj14'] = df.cnpj.str.replace('[^\w\s]','')
df['cnpj8'] = df.cnpj14.str[:8]
```

[7] O diretório TCC/DATALAKE é o repositório dos *datasets* base, e se localiza no sistema de arquivos do ambiente Receitadata.

Algorítmo 1: *Jupyter Notebook Python de Extração dos data sources no Receitadata*

Data: tabela `tcc_maed_cnpj`
1 importa *packages*;
2 lê credenciais para autenticação no kerberos;
3 conecta ao ambiente de consultas (Impala);
4 **while** *houver consultas intermediárias a executar* **do**
5 monta SQL da consulta;
6 executa SQL e salva o resultado em tabela temporária (ctas);
7 **end**
8 lê a tabela temporária para o DataFrame;
9 salva o DataFrame para arquivo CSV;

Este novo Datafame foi enão salvo no arquivo `TCC/DATALAKE/tcc_maed_cnpj.csv` com formato CSV, e suas características são mostradas na Tabela 4.

Após isso, este *dataset* foi então importado para a tabela SQL `tcc_maed_cnpj` no *schema* do ambiente Receitadata, com o mesmo formato da Tabela 4. Esta tabela será utilizada como filtro nas demais consultas SQL que produzirão os outros *datasets* base deste processo, conforme a seguir.

2. ***Data Sources* DCTF, EFD, EFIN, e NFE**

A extração de dados dos demais *data sources* DCTF, EFD, EFIN, e NFE (ver Tabela 1) ocorreu conforme a seguir.

Implementamos um Jupyter Notebook Python para extração de dados de cada um destes *data sources*. Cada *data source* deu origem a uma diferente tabela fonte no *schema* do Receitadata. Desta forma, uma extração contou com uma sequência de consultas SQL executadas, eventualmente utilizando armazenagem parcial de dados em tabelas temporárias, e uma consolidação da saída final.

O Algoritmo 1 apresenta o procedimento geral de extração dos *data sources*. Os dados de entrada consistem na tabela `tcc_maed_cnpj`, que é utilizada para seleção dos CNPJ.

O padrão das consultas SQL de extração (linhas 5 e 6 do Algoritmo 1) é mostrado abaixo.

```
1 sql = '''
2     SELECT filtro.cnpj8, <demais colunas de tab1, tab2,...>
3     FROM {schema}.tcc_maed_cnpj AS filtro
4         INNER JOIN tabela1 AS tb1
5             ON <creiterios da relacao>
6         LEFT JOIN tabela2 AS tb2
```

```
            ON <creiterios da relacao>
        (...)
    WHERE
        FLOOR(tb1.dt_dia_inic_per/100) >= {dt_inicio_analise} AND
        FLOOR(tb1.dt_dia_inic_per/100) <= {dt_fim_analise} AND
        (...)
    GROUP BY cnpj8, ano, mes
    UNION
    (...)
'''
sql = sql.format(schema=rd.schema, dt_inicio_analise=ano_mes_ini,
    dt_fim_analise=ano_mes_fim)
tabela = 'tcc_dctf_tmp'
rd.executar_ctas(tabela=tabela,
                 sort_by=['cnpj8', 'ano', 'mes'],
                 sql_select=sql)
```

Nesse extrato de código, o SELECT é realizado na tabela filtro `tcc_maed_cnpj` (cláusula FROM), e as demais tabelas contendo os dados de interesse são acessadas pelos relacionamentos (cláusulas JOIN). Por fim, é aplicado o filtro de datas para o período de janeiro de 1999 a novembro de 2021 (1999-01 a 2021-11, ver Tabela 3) na cláusula WHERE, eventualmente com outros critérios, e os resultados são ordenados.

Após a execução das consultas intermediárias necessárias para se obter os dados de interesse de cada *data source*, o resultado final é salvo na sua respectiva tabela no *schema* do Receitadata. Em seguida, exportamos esta tabela para um arquivo CSV.

Por exemplo, para o *data source* DCTF, a tabela final no Receitadata foi a `tcc_dctf` e o arquivo CSV foi o `TCC/DATALAKE/tcc_dctf.csv`, gerado pelo código abaixo.

```
 # Salva a tabela final no formato CSV
sql = f'select * from {rd.schema}.tcc_dctf'
df = rd.read_sql(sql)
df.to_csv('~/TCC/DATALAKE/tcc_dctf.csv', header=True, index=False)
```

Este foi o processo de extração, que foi executado para cada um dos *data sources* DCTF, EFD, EFIN, e NFE. Vale ressaltar que todos os registros resultantes da extração são referentes a valores apurados mensalmente para as respectivas declarações e escriturações.

Uma vez concluída a fase de extração dos *data sources*, nós implementamos no Jupyter Notebook Python `TCC/DATALAKE/tcc_dataset.ipynb` a geração dos *datasets*

DCTF

Gerado pelo Jupyter Notebook Python `TCC/DATALAKE/tcc_dctf.ipynb`
Tabela no Receitadata: `tcc_dctf`
Arquivo CSV: `TCC/DATALAKE/tcc_dctf.csv`
Linhas(1.137.971): duplicadas=0 nulas=0 metricas zero=15.014 metricas negativo=0
Colunas(3): colunas com alguma metrica zero=deb_apurados_mes(15.014)
cnpj8 únicos=6558 período=2004-01 a 2021-11

Coluna/campo	*Descrição*	*Tipo*	*Exemplo*
cnpj8	CNPJ do contribuinte	string	99999999
ano_mes	Ano e mês da declaração	string	2021-11
deb_apurados_mes	Total de débitos apurados	float	999999999,99

Tabela 5: Formato do *dataset* DCTF.

EFD

Gerado pelo Jupyter Notebook Python `TCC/DATALAKE/tcc_efd.ipynb`
Tabela no Receitadata: `tcc_efd`
Arquivo CSV: `TCC/DATALAKE/tcc_efd.csv`
Linhas(656.154): duplicadas=0 nulas=0 metricas zero=648.801 metricas negativo=0
Colunas(8): colunas com alguma metrica zero=pis_n_cumulativo(348.976)
pis_cumulativo(535.718) cofins_n_cumulativo(346.097) cofins_cumulativo(534.798)
cprb(584.693) rb(3457)
cnpj8 únicos=6269 período=2011-04 a 2021-11

Coluna/campo	*Descrição*	*Tipo*	*Exemplo*
cnpj8	CNPJ do contribuinte	string	99999999
ano_mes	Ano e mês da declaração	string	2021-11
pis_n_cumulativo	PIS não cumulativo	float	999999999,99
pis_cumulativo	PIS cumulativo	float	999999999,99
cofins_n_cumulativo	COFINS não cumulativo	float	999999999,99
cofins_cumulativo	COFINS cumulativo	float	999999999,99
cprb	Contrib. Social sobre a Receita	float	999999999,99
rb	Receita bruta	float	999999999,99

Tabela 6: Formato do *dataset* EFD.

base, e também o restante do processo de ETL inicial. Os resultados obtidos são mostrados nas Tabelas 5, 6, 7, e 8 a seguir.

Na parte superior de cada tabela de resultados do *data source*, são identificados seus respectivos Jupyter Notebook Python gerador, tabela SQL no *schema* Receitadata resultante da extração de dados, e arquivo CSV produzido. Na sequência são

EFIN			
Gerado pelo Jupyter Notebook Python TCC/DATALAKE/tcc_efin.ipynb Tabela no Receitadata: tcc_efin Arquivo CSV: TCC/DATALAKE/tcc_efin.csv Linhas(443.248): duplicadas=0 nulas=0 metricas zero=10.191 metricas negativo=0 Colunas(4): colunas com alguma metrica zero=total_creditos(9625) total_debitos(6505) cnpj8 únicos=6554 período=2014-12 a 2021-11			
Coluna/campo	*Descrição*	*Tipo*	*Exemplo*
cnpj8	CNPJ do contribuinte	string	99999999
ano_mes	Ano e mês da declaração	string	2021-11
total_creditos	Movimentação financeira em créditos	float	999999999,99
total_debitos	Movimentação financeira em débitos	float	999999999,99

Tabela 7: Formato do *dataset* EFIN.

NFE			
Gerado pelo Jupyter Notebook Python TCC/DATALAKE/tcc_nfe.ipynb Tabela no Receitadata tcc_nfe Arquivo TCC/DATALAKE/tcc_nfe.csv Linhas(935.657): duplicadas=0 nulas=0 metricas zero=362.367 metricas negativo=0 Colunas(4): colunas com alguma metrica zero=vr_compras(18) vr_vendas(362.364) cnpj8 únicos=6507 período=2007-01 a 2021-11			
Coluna/campo	*Descrição*	*Tipo*	*Exemplo*
cnpj8	CNPJ do contribuinte	string	99999999
ano_mes	Ano e mês da declaração	string	2021-11
vr_compras	Total em notas fiscais de compras	float	999999999,99
vr_vendas	Total em notas fiscais de vendas	float	999999999,99

Tabela 8: Formato do *dataset* NFE.

mostrados alguns contadores obtidos após a limpeza inicial do *dataset* base, seguido da sua relação de colunas.[8]

A limpeza inicial realizada no final da geração dos *datasets* base envolveu a remoção de linhas nulas, duplicadas, e com valores negativos. Os valores zero foram preservados, por serem situações plausíveis no contexto do problema.

Assim, ao término desta etapa, geramos seis *datasets* com um total de 3.278.206 linhas.

[8]Em todos os *datasets* base, as colunas originais ano e mes foram compactadas numa única coluna ano_mes.

2.4.3 *Merge* dos *Datasets* Base e Geração do *Dataset* Intermediário

Nesta etapa, realizamos o *merge* dos *datasets* base para a geração do *dataset* intermediário. Todas as demais implementações do restante deste processo de ETL estão no Jupyter Notebook Python `TCC/DATALAKE/tcc_dataset.ipynb`.

Primeiramente, realizamos o equivalente ao *merge* do *dataset* MAED (Tabela 3) com o *dataset* DCTF (Tabela 5), adicionando a coluna `maed_dctf`, que indica se o contribuinte incorreu em MAED no próprio mês da DCTF.

Após isso, adicionamos os rótulos MAED ao *dataset* DCTF, por meio da coluna `maed_prox_mes`, que indica se o contribuinte incorreu em MAED no mês subsequente ao mês da DCTF (próximo mês)—esta é a variável que vamos predizer neste trabalho. Por exemplo, para a DCTF com `ano_mes` igual a 2021-10, o valor de `maed_prox_mes` indica se ocorreu MAED em 2021-11.

A adição destas duas colunas foi realizada com o extrato de código abaixo.

```
1  # (I) Merge DCTF + MAED. Adicionar colunas MAED no mes e no proximo mes (rortulo MAED)
2  maed_dict = df_maed_dict(df_maed_base)
3  df = df_dctf_base.copy()
4  df['maed_dctf'] = df.apply(lambda row: df_column_apply_maed_dctf(maed_dict, row),
       axis=1)
5  df['maed_dctf_prox_mes'] = df.apply(lambda row:
       df_column_apply_maed_dctf_prox_mes(maed_dict, row), axis=1)
```

Na linha 2 do código, criamos um dicionário com o conteúdo do *dataset* MAED, por meio da chamada à função `df_maed_dict()`; na linha 3 copiamos o DataFrame do *dataset* DCTF original para o DataFrame `df`, que irá abrigar o *dataset* intermediário; e nas linhas 4 e 5 criamos as colunas `maed_dctf` e `maed_prox_mes`, aplicando ao DataFrame (`df.apply()`) as funções `df_column_apply_maed_dctf()` e `df_column_apply_maed_dctf_prox_mes()`, respectivamente, que recebem o dicionário MAED como parâmetro. Estas funções auxiliares são listadas no Apêndice II deste texto, que mostra o código `TCC/COMMON/tcc_common.py`, que contém os artefatos de uso geral que implementamos para este trabalho.

Em seguida, realizamos o *merge* dos demais *datasets*, EFD, EFIN, e NFE, conforme o extrato de código abaixo.

```
# (II) Merge dos demais datasets base
# df = df + efd
left = df_dctf_maed
right= df_efd_base
df = pd.merge(left, right, how='inner', left_on=['cnpj8', 'ano_mes'], right_on=['cnpj8', 'ano_mes'])
df_clean(df)
# df = df + efin
left = df
right= df_efin_base
df = pd.merge(left, right, how='inner', left_on=['cnpj8', 'ano_mes'], right_on=['cnpj8', 'ano_mes'])
df_clean(df)
# df = df + nfe
left = df
right= df_nfe_base
df = pd.merge(left, right, how='inner', left_on=['cnpj8', 'ano_mes'], right_on=['cnpj8', 'ano_mes'])
df_clean(df)
```

No trecho de código acima, vale notar a chamada da função `df_clean(df)` após cada *merge*. Esta função remove linhas nulas, duplicadas, e com valores negativos do DataFrame.

Encerrando esta etapa de criação do *dataset* intermediário, renomeamos as colunas para nomes mais curtos e as reposicionamos para facilitar a visualização, e também ofuscamos a coluna `cnpj8` para não permitir a identificação dos contribuintes.

Assim, ao término desta etapa, geramos o *dataset* intermediário com um total de 390.714 linhas e 15 colunas. O detalhamento das colunas e os valores de alguns contadores obtidos são mostrados na Tabela 9.

2.4.4 Remoção de *Outliers* do *Dataset* Intermediário

Nesta etapa, tratamos da remoção de valores extremos (*outliers*) do *dataset* intermediário.

Iniciamos a etapa gerando os *boxplots* dos dados originais, conforme mostrado na Figura 4, onde pudemos verificar a forte presença de *outliers* em diversas colunas.

Em seguida, optamos por remover o último percentil dos dados, conforme o código a seguir. Esta escolha se deu como um compromisso entre a eliminação dos *outliers* e

Dataset Intermediário			
Gerado pelo Jupyter Notebook Python `TCC/DATALAKE/tcc_dataset.ipynb`			
Pandas DataFrame			
Linhas(390.714): duplicadas=0 nulas=0 metricas zero=388.991 metricas negativo=0			
Colunas(15): colunas com alguma metrica zero=dctf(656)			
pis_n(169.356) pis_c(304.571) cfs_n(167.069) cfs_c(303.881)			
cprb(364.606) rb(1123) crd(5663) deb(4435) cpras(6) vndas(115.276)			
cnpj8 únicos=6123 período=2014-12 a 2021-11			
Coluna/campo	*Descrição*	*Tipo*	*Exemplo*
cnpj8	CNPJ do contribuinte	string	99999999
ano_mes	Ano e mês da declaração	string	2021-11
maed_dctf	MAED no mês corrente	integer	{0, 1}
maed_dctf_prox_mes	MAED no próximo mês	integer	{0, 1}
dctf	Total de débitos apurados em DCTF	float	999999999,99
pis_n	PIS não cumulativo	float	999999999,99
pis_c	PIS cumulativo	float	999999999,99
cfs_n	COFINS não cumulativo	float	999999999,99
cfs_c	COFINS cumulativo	float	999999999,99
cprb	Contrib. Social sobre a Receita	float	999999999,99
rb	Receita bruta	float	999999999,99
crd	Movimentação financeira em créditos	float	999999999,99
deb	Movimentação financeira em débitos	float	999999999,99
cpras	Total em notas fiscais de compras	float	999999999,99
vndas	Total em notas fiscais de vendas	float	999999999,99

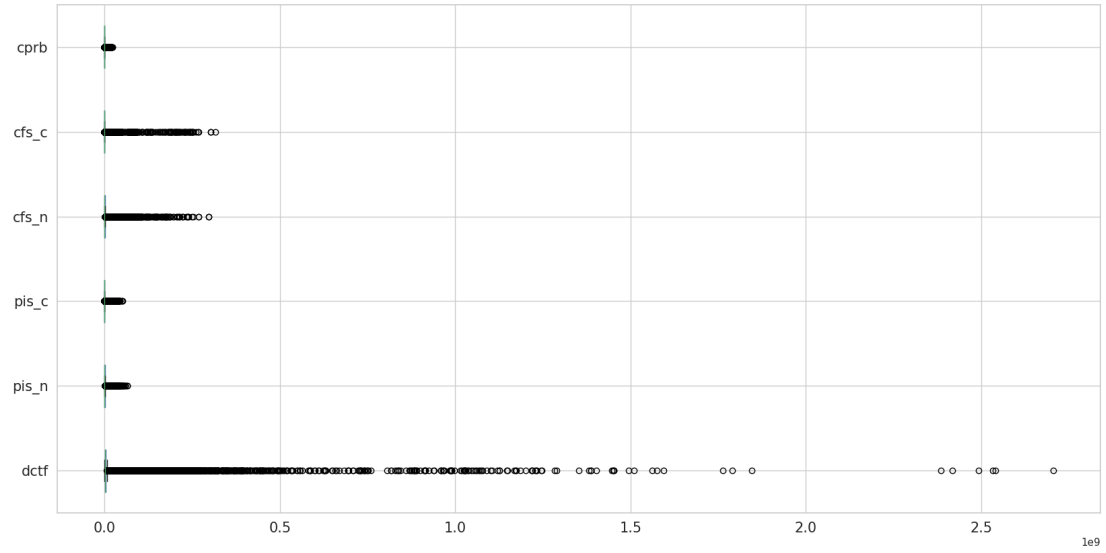
Tabela 9: Formato inicial do *dataset* intermediário.

a preservação da maior quantidade de linhas possíveis no *dataset*. Ainda, vale notar a grande diferença de escala de valores entre algumas colunas do *dataset*.

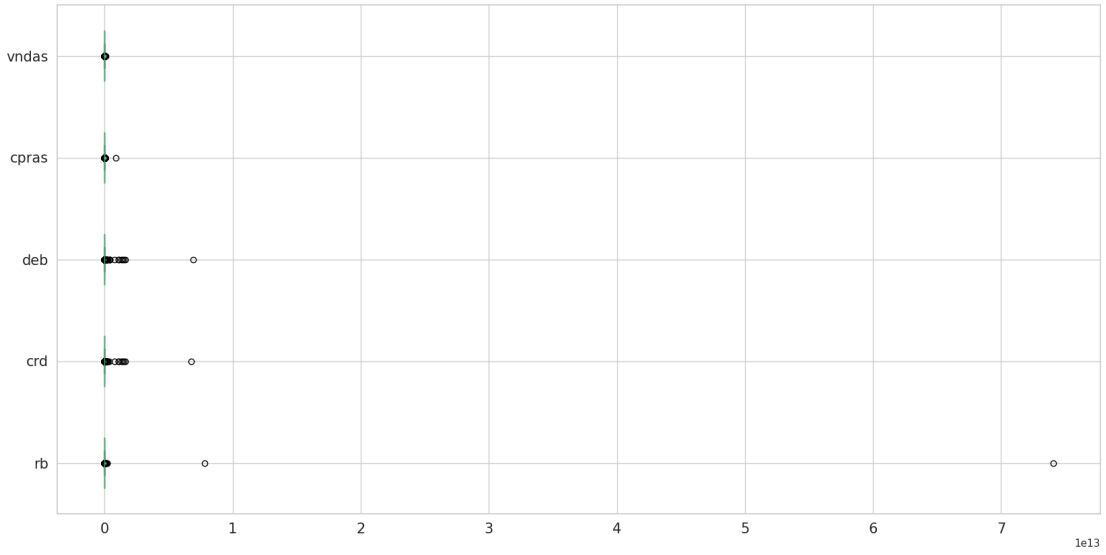
```
# Remove o ultimo percentil (99 percentil) das colunas numericas
D = df.quantile(0.99)
df = df[~(df > D).any(axis=1)].copy()
```

Os *boxplots* após a remoção de *outliers* são mostrados na Figura 5, onde ainda podemos observar uma forte variabilidade e assimetria nos valores de diversas colunas, realçando as diferenças no perfil econômico dos contribuintes, também presentes no universo de contribuintes diferenciados.

Ao término desta etapa, o *dataset* intermediário agora contém 372.148 linhas—18.566 linhas foram removidas—e as mesmas 15 colunas.



(a)

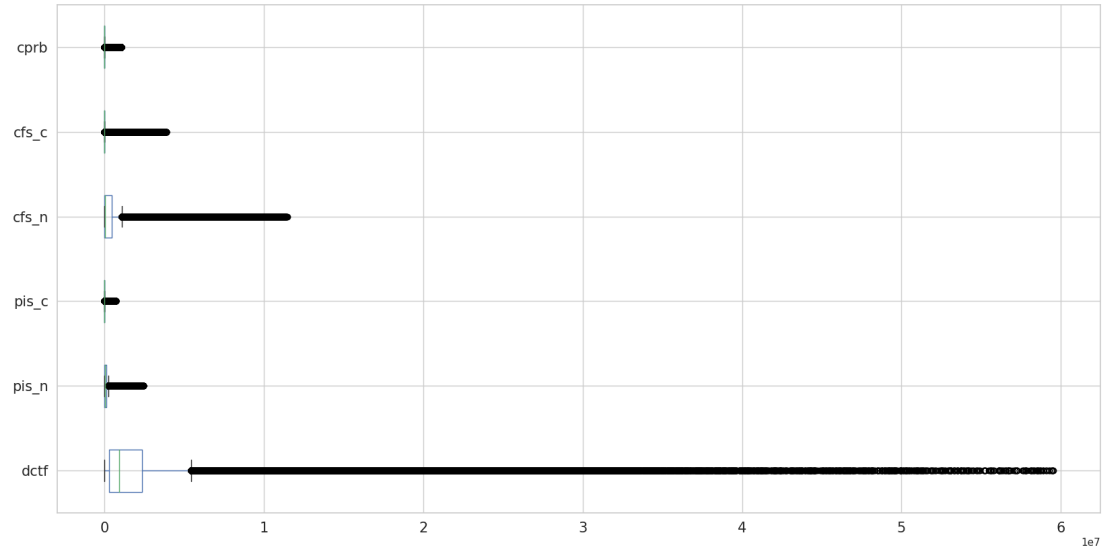


(b)

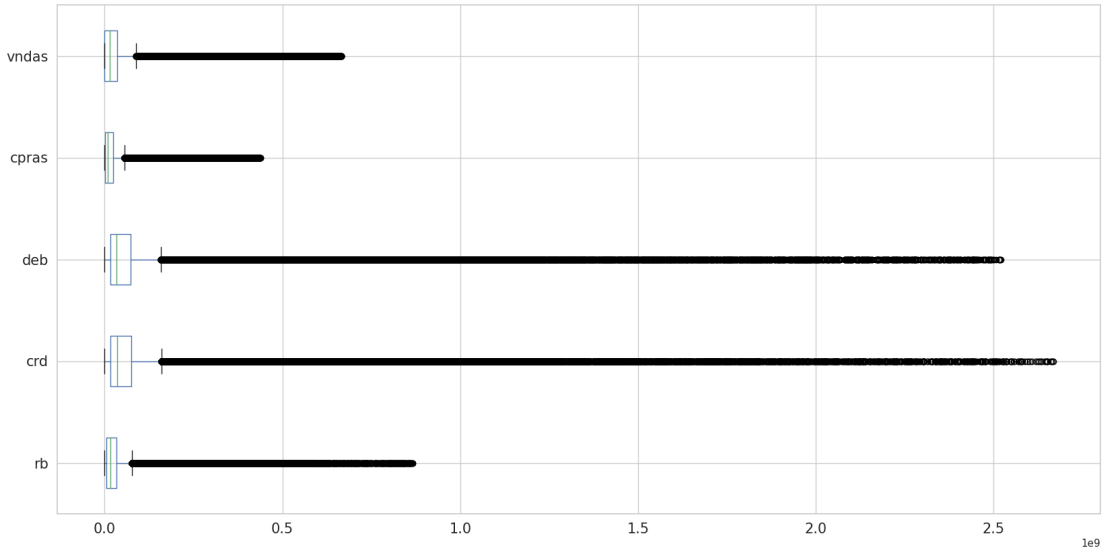
Figura 4: *Boxplot* do *dataset* intermediário com *outliers*.

2.4.5 Balanceamento de Classes do *Dataset* Intermediário

A próxima etapa foi avaliar o balanceamento das classes MAED das colunas `maed_dctf` e `maed_dctf_prox_mes` do *dataset* intermediário, sendo esta última coluna a mais crítica pois é aquela que iremos predizer.



(a)



(b)

Figura 5: *Boxplot* do *dataset* intermediário com remoção de *outliers*.

Inicialmente, verificamos que o *dataset* intermediário era do tipo *severamente não balanceado* (BROWNLEE, 2022), com relação Sim/Não de aproximadamente 1:71, conforme distribuições de classes mostradas abaixo e na Figura 6.

`maed_dctf_prox_mes` = 0 $\Rightarrow$ 367.018 (98,62% da classe)

`maed_dctf_prox_mes` = 1 $\Rightarrow$ 5130 (1,38% da classe)

`maed_dctf` = 0 $\Rightarrow$ 366.972 (98,61% da classe)

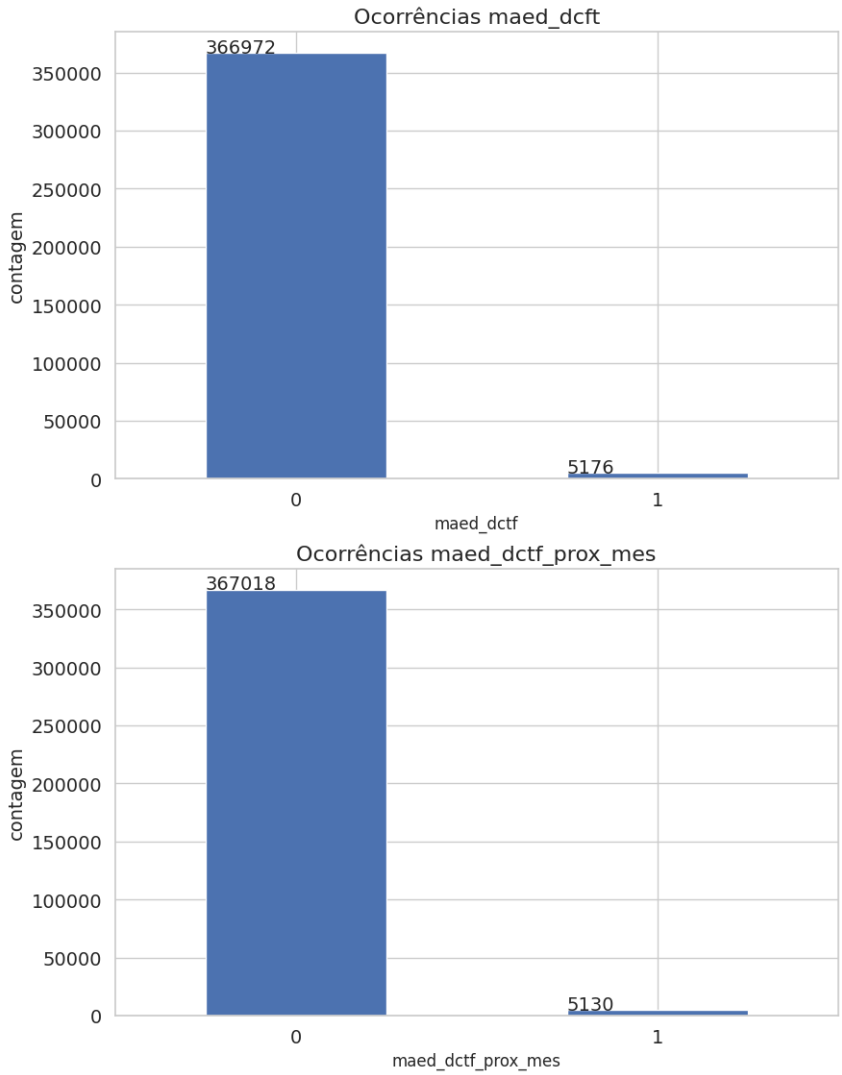


Figura 6: Distribuição das classes no *dataset* intermediário não balanceado

`maed_dctf` = 1 $\Rightarrow$ 5176 (1,39% da classe)

Nossa estratégia para contornar este problema foi extrair uma amostra balanceada do *dataset*, e para isso, utilizamos a função `df_profile_maed(df)`, que subdividiu o *dataset* não balanceado em quatro DataFrames:

`f_00` = linhas com `maed_dctf` = 0 e `maed_dctf_prox_mes` = 0;

`f_01` = linhas com `maed_dctf` = 0 e `maed_dctf_prox_mes` = 1;

`f_10` = linhas com `maed_dctf` = 1 e `maed_dctf_prox_mes` = 0;

`f_11` = linhas com `maed_dctf` = 1 e `maed_dctf_prox_mes` = 1;

Os tamanhos resultantes para cada um desses DataFrames é mostrado abaixo, nas respectivas variáveis `m_ij`:

```
Profile MAED: m_ij (i==maed_dctf, j=maed_dctf_prox_mes)
m_00=362948 m_01=  4024
m_10=  4070 m_11=  1106
```

Em seguida, criamos um novo DataFrame priorizando o balanceamento da coluna `maed_dctf_prox_mes` da seguinte forma:

(1) Aproveitamos todos os rótulos `maed_dctf_prox_mes` = 1 disponíveis, utilizando completamente os DataFrames `f_01` e `f_11`, (`n_s`= 5130);

(2) Utilizamos completamente o DataFrame `f_10`, que contém todas as linhas disponíveis com rótulos `maed_dctf` = 1 e `maed_dctf_prox_mes` = 0 (`n_n`= 4070); e

(3) Amostramos aleatoriamente o DataFrame `f_00` (o mais abundante), que contém todas as linhas disponíveis com rótulos `maed_dctf` = 0 e `maed_dctf_prox_mes` = 0, na quantidade faltante `n_s - n_n` = 1060.

Assim, obtivemos um novo *dataset* com as características de balanceamento mostradas abaixo e na Figura 7.

`maed_dctf_prox_mes` = 0 $\Rightarrow$ 5130 (50,00% da classe)
`maed_dctf_prox_mes` = 1 $\Rightarrow$ 5130 (50,00% da classe)
`maed_dctf` = 0 $\Rightarrow$ 5084 (49,55% da classe)
`maed_dctf` = 1 $\Rightarrow$ 5176 (50,45% da classe)

O novo particionamento de rótulos MAED do *dataset* ficou como:

```
Profile MAED: m_ij (i==maed_dctf, j=maed_dctf_prox_mes)
m_00=  1060 m_01=  4024
m_10=  4070 m_11=  1106
```

Este balanceamento descrito acima foi implementado com o código abaixo.

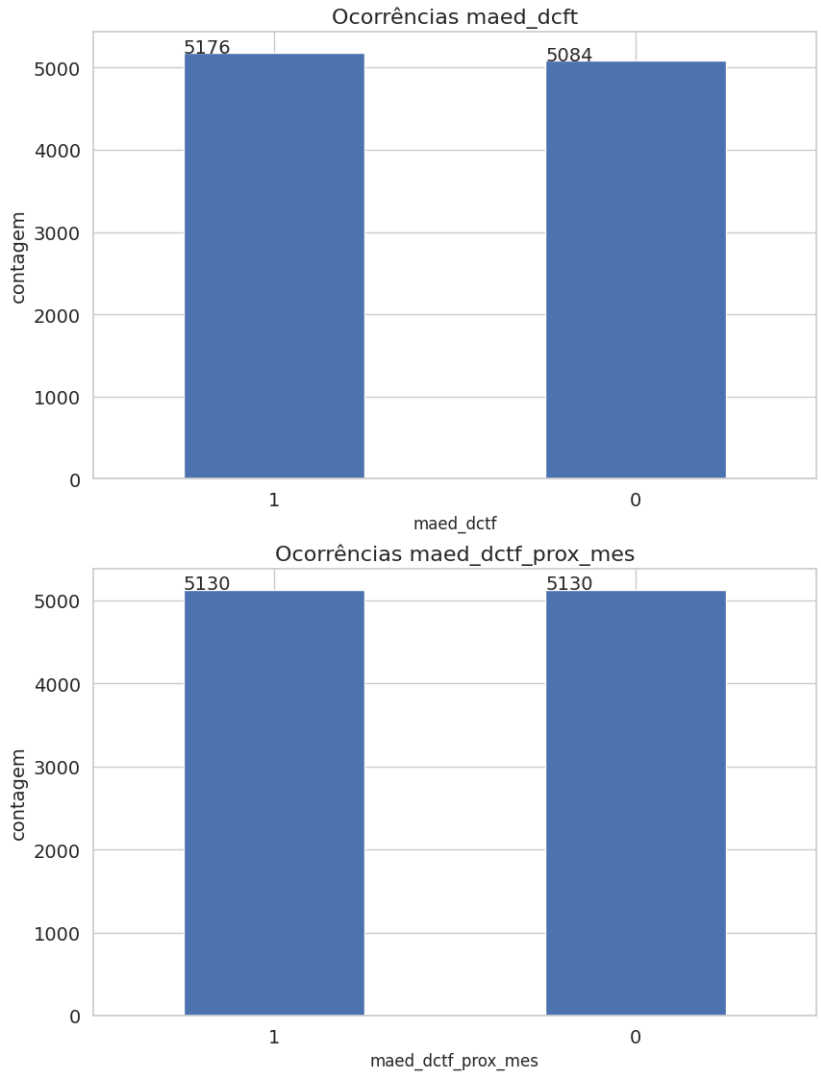


Figura 7: Distribuição das classes no *dataset* intermediário balanceado

```
 # Balanceando o dataset inicial
df = df_base_sem_outliers.copy()
 # profile das classes MAED
f00, f01, f10, f11 = df_profile_maed(df)
print('seleciona maed_dctf_prox_mes == 1 (f01 + f11): n_s=', end=' ')
df_s = pd.concat([df[f01], df[f11]]) # df[df["maed_dctf_prox_mes"] == 1].copy()
n_s = len(df_s)
print('seleciona maed_dctf==1 e maed_dctf_prox_mes==0 (f10): n_n=', end=' ')
df_n = df[f10]
n_n  = len(df_n)
print('seleciona amostras faltantes maed_dctf==0 e maed_dctf_prox_mes==0 (f00) para
    igualar n_n a n_s: n=', end=' ')
n = n_s - n_n
df_n00 = df[f00].sample(n=n, random_state=1)
df_n = pd.concat([df_n, df_n00])
 # concatena os datasets
df = pd.concat([df_s, df_n])
```

Ao término desta etapa, o *dataset* intermediário foi bastante reduzido, e agora contém 10.260 linhas—361.888 linhas foram removidas—com as mesmas 15 colunas, 3160 CNPJ únicos, e dados no período de janeiro de 2015 a outubro de 2021.

2.4.6 Exportação do *Dataset* Inicial

Concluindo o ETL dos *data sources*, geramos o *dataset* inicial a partir do *dataset* intermediário com remoção de *outliers* e balanceado, e o exportamos para o arquivo CSV `TCC/DATALAKE/tcc_dataset_inicial_balanceado.csv`, que será utilizado na próxima etapa.

3 Processamento/Tratamento de Dados

Na seção anterior, apresentamos nosso processo inicial de ETL dos *data sources*, que produziu o nosso *dataset* inicial. Este *dataset* passou por tratamentos e processamentos para torná-lo adequado para utilização pelos modelos de *Machine Learning*.

Isso ocorreu em duas etapas neste trabalho. A primeira etapa será apresentada nesta seção, onde tratamos a multicolinearidade das *features*,[9] e produzimos um *dataset* aumentado, adicionando algumas colunas que verificamos empiricamente ter contribuído com a capacidade preditiva dos modelos. A segunda etapa ocorreu durante o treinamento dos modelos de *Machine Learning*, onde utilizamos o método *K-Fold Cross-Validation* e foi aplicado pré-processamento aos *datasets* de cada *fold*.

3.1 Tratamento da Multicolinearidade

Em modelos de regressão, o efeito da *multicolinearidade* ocorre quando há uma forte correlação linear entre dois ou mais pares de variáveis explanatórias (ou preditoras). Como consequência geral, isso dificulta identificar o efeito de uma ou outra variável na variável resposta, mas, quando a multicolinearidade é severa, ela pode afetar fortemente a variância dos coeficientes estimados para o modelo, tornando estas

[9] Utilizaremos o termo *feature* como sinônimo das variáveis (colunas) preditoras tributárias e econômico-financeiras, e ocasionalmente as colunas `maed_dctf` e `maed_dctf_total`.

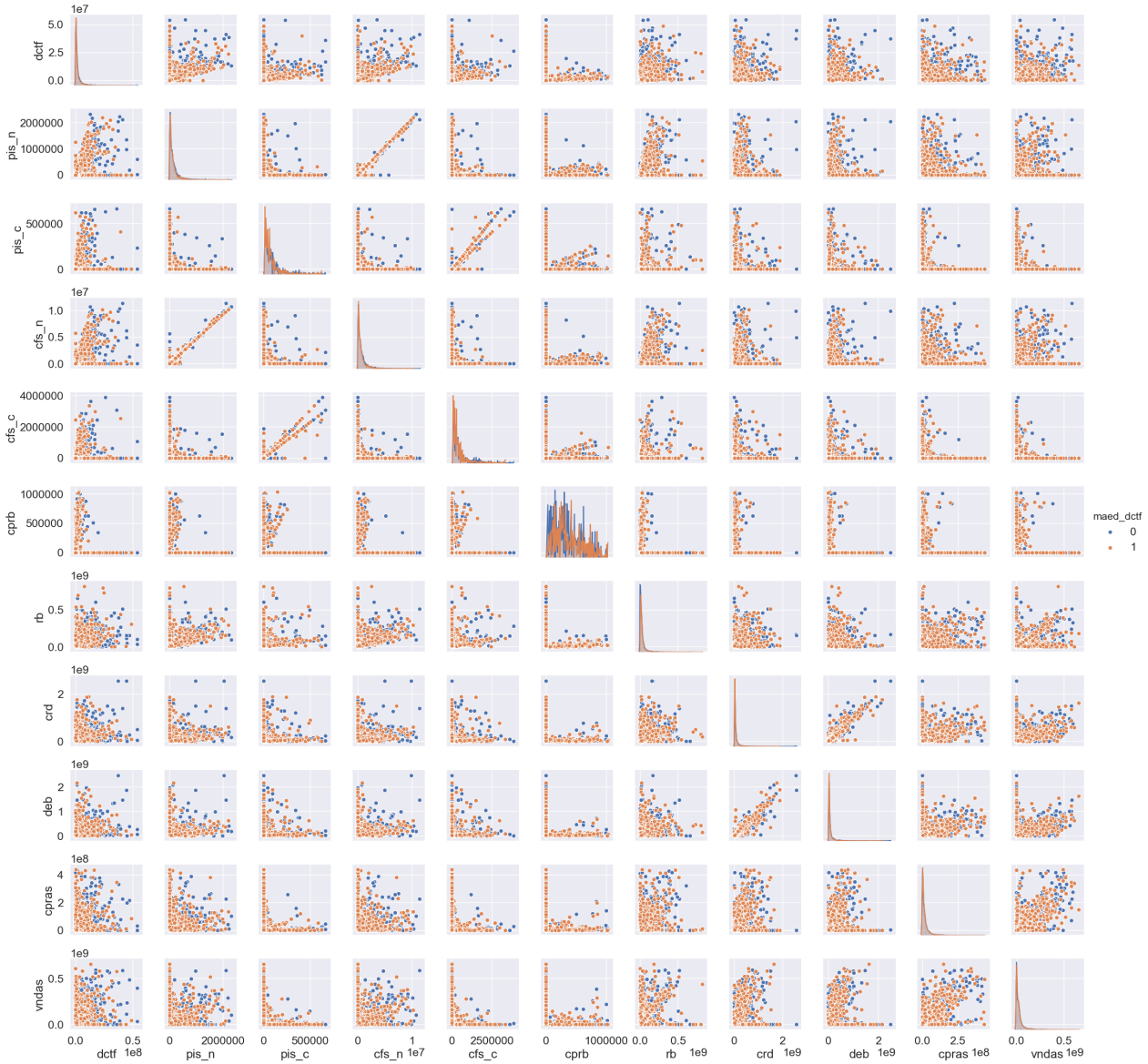


Figura 8: Gráficos de dispersão das *features* do *dataset* inicial

estimativas instáveis. Estes erros de estimação prejudicam o desempenho final do modelo, sendo uma boa prática evitar a multicolinearidade mesmo em modelos não lineares (KESAVULU et al., 2016).

Iniciamos esta etapa de tratamento da multicolinearidade dos dados verificando indícios de correlação por meio dos *gráficos de dispersão* (*scatter plot*) dos pares de colunas do *dataset* inicial, conforme mostrado na Figura 8.

O gráfico de dispersão tipicamente utiliza coordenadas Cartesianas para mostrar pontos correspondentes aos valores de duas variáveis, sendo que o valor de uma variável determina a posição no eixo horizontal, e o valor da outra a posição no eixo vertical.

O padrão de dispersão dos pontos plotados no gráfico sugere a presença ou não de correlação, sendo que no caso de existir correlação entre as variáveis, estes padrões assemelham-se a retas—quanto mais aglomerados mais forte é a correlação, e o sentido da inclinação da reta determina o sinal da correlação.

A Figura 8 foi obtida por meio do método seaborn `sns.pairplot()`, que produz uma matriz quadrada de gráficos de dispersão com tamanho igual ao número de colunas numéricas do DataFrame, sendo que a diagonal principal apresenta o gráfico de densidade da coluna.

Como podemos observar na Figura 8, diversos pares de colunas apresentaram padrão de variáveis correlacionadas, o que nos levou a aprofundar nossa verificação de multicolinearidade na próxima etapa.

3.2 *Variance Inflation Factor* (VIF)

Para mensurar mais precisamente o grau de multicolinearidade das colunas do *dataset* inicial, calculamos o *fator de inflação da variância* (*variance inflation factor*–VIF) para cada *feature*.

O VIF é uma medida do acréscimo da variância das estimativas dos coeficientes quando se adiciona uma variável a um modelo de regressão linear. Ou seja, é uma medida da quantidade de multicolinearidade em um conjunto de variáveis de regressão, que é calculada para cada variável do modelo, como sendo a relação entre a variância total do modelo e a variância de um outro modelo que inclui somente aquela variável (JAMES et al., 2021). O VIF é definido para cada variável como:

$$VIF = \frac{1}{1 - R^2} \tag{1}$$

onde R^2 é o coeficiente de determinação do modelo, dado por

$$R^2 = 1 - \frac{\sum_i e_i^2}{\sum_i (y_i - \bar{y})^2} \tag{2}$$

onde o somatório do numerador é a soma dos quadrados dos resíduos da regressão envolvendo somente aquela variável, e o somatório do denominador é a soma total dos quadrados, que é proporcional à variância dos dados.

	Feature	VIF
0	dctf	2,43
1	pis_n	65,53
2	pis_c	12,16
3	cfs_n	67,86
4	cfs_c	12,61
5	cprb	1,02
6	rb	2,51
7	crd	10,73
8	deb	10,90
9	cpras	2,78
10	vndas	3,08

Tabela 10: *Variance inflation factor* (VIF) do *dataset* inicial

Os VIF do nosso *dataset* inicial foram calculados com o extrato de código abaixo, que utilizou a função `variance_inflation_factor()` da biblioteca statsmodels.

```
feats = df_columns_metric(df)
X = df[feats]
vif = pd.DataFrame()
vif["feature"] = X.columns
vif["VIF"] = [variance_inflation_factor(X.values, i) for i in range(len(X.columns))]
display(vif)
```

Os resultados obtidos são mostrados na Tabela 10, e a prática para o critério de decisão do VIF é eliminar variáveis com valores acima de 5 ou 10.

Como a natureza dos nossos dados (tributários e econômico-financeiros) traz alguma correlação implícita (por exemplo, empresas de um mesmo setor econômico), optamos por adotar o critério de corte de 10.

Desta forma, verificamos que as colunas `pis_n`, `pis_c`, `cfs_n`, e `cfs_c` obtiveram valores de VIF acima de 10. De fato, estes tributos incidem sobre a mesma base de cálculo tributária, o que justifica seus elevados valores de VIF.

Assim, a nossa estratégia de contorno foi agregar estas colunas numa nova coluna `pis_cfs = pis_n + pis_c + cfs_n + cfs_c`, produzindo um *dataset* reduzido em relação aos dados originais extraídos, com seus gráficos de dispersão e valores de VIF mostrados, respectivamente, na Figura 9 e Tabela 11.

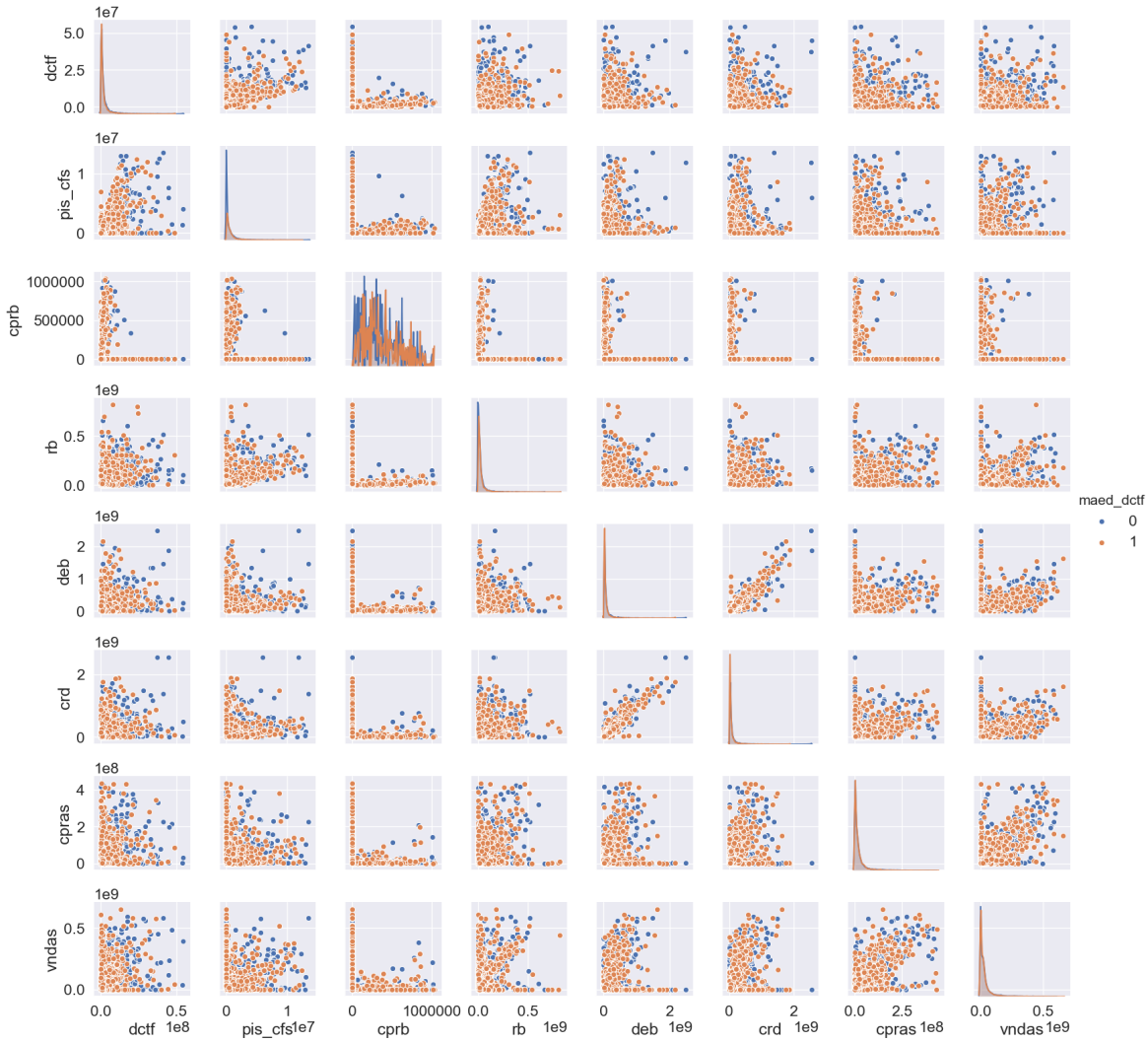


Figura 9: Gráficos de dispersão das *features* do *dataset* reduzido

Quanto à coluna `deb`, apesar do seu valor ter ultrapassado levemente o limiar de corte do VIF, optamos por preservá-la no *dataset* porque ela será empregada na criação da nova coluna `rbc`, conforme será apresentado na próxima seção.

Já a coluna `crd`, optamos por preservá-la por ser um informação complementar à coluna `deb`, e seu valor de VIF ter ficado próximo ao desta coluna.

Conforme mostram a Figura 9 e a Tabela 11, as colunas `crd` e `deb` exibem correlação mais acentuada entre elas, e moderada com algumas outras colunas, e apresentam os valores de VIF mais altos do conjunto.

	Feature	VIF
0	dctf	2,39
1	pis_cfs	2,08
2	cprb	1,01
3	rb	2,40
4	deb	10,87
5	crd	10,71
6	cpras	2,78
7	vndas	3,01

Tabela 11: *Variance inflation factor* (VIF) do *dataset* reduzido

3.3 *Dataset* Aumentado

Concluindo esta primeira etapa de tratamentos e processamentos dos dados, nós produzimos um *dataset* aumentado a partir da versão reduzida, adicionando algumas colunas que verificamos empiricamente ter contribuído com a capacidade preditiva dos modelos, conforme a seguir.

As novas colunas foram adicionadas com a função `df_augment(df)`, que é listada no Apêndice II deste texto, e suas descrições são:

1. **Receita Bruta Calculada** (`rbc`), obtida como o maior valor entre a receita bruta da EFD-Contribuições e NF-e vendas, quando ambos são maiores que zero; ou o valor da e-Financeira débitos. Trata-se de uma estimativa mais robusta da receita bruta real do contribuinte, que utilizaremos no lugar da coluna `rb`.

2. **Carga Tributária** (`ct`), obtida como sendo os débitos em DCTF divididos pela receita bruta calculada, `ct = dctf / rbc`.

3. **MAED DCTF Total** (`maed_dctf_total`), acumula o número de vezes que o contribuinte incorreu em MAED. Trata-se de um indicador comportamental do contribuinte na DCTF.

Com este *dataset* aumentado, criamos nosso *dataset* final, cujas características são mostradas na Tabela 12, e seus gráficos de dispersão e valores de VIF são mostrados nas Figuras 10 e Tabela 13, respectivamente. Este *dataset* final foi salvo no arquivo CSV `TCC/EXP/tcc_dataset_final_balanceado_completo.csv`, e uma outra versão do *dataset* sem as colunas `cnpj8` e `ano_mes` foi salva no arquivo CSV `TCC/EXP/tcc_dataset_final_balanceado.csv`.

Dataset Final			
Gerado pelo Jupyter Notebook Python `TCC/EXP/tcc_exploracao.ipynb` Pandas DataFrame Linhas(10.260): duplicadas=0 nulas=0 metricas zero=10.100 metricas negativo=0 Colunas(14): colunas com alguma metrica zero=dctf(170) pis_cfs(4011) cprb(9756) crd(110) deb(79) cpras(1) vndas(2639) rbc(2) ct(171) cnpj8 únicos=3160 período=2015-01 a 2021-10			
Coluna/campo	*Descrição*	*Tipo*	*Exemplo*
cnpj8	CNPJ do contribuinte	string	99999999
ano_mes	Ano e mês da declaração	string	2021-11
dctf	Total de débitos apurados em DCTF	float	999999999,99
pis_cfs	Total geral de PIS e COFINS	float	999999999,99
cprb	Contrib. Social sobre a Receita	float	999999999,99
crd	Movimentação financeira em créditos	float	999999999,99
deb	Movimentação financeira em débitos	float	999999999,99
cpras	Total em notas fiscais de compras	float	999999999,99
vndas	Total em notas fiscais de vendas	float	999999999,99
rbc	Receita bruta calculada	float	999999999,99
ct	Carga tributária da DCTF	float	9999,99
maed_dctf	MAED no mês corrente	integer	{0, 1}
maed_dctf_total	Total de ocorrências de MAED	integer	5
maed_dctf_prox_mes	MAED no próximo mês	integer	{0, 1}

Tabela 12: Formato e características do *dataset* final.

	Feature	VIF
0	dctf	2,41
1	pis_cfs	1,91
2	cprb	1,01
3	crd	10,73
4	deb	10,89
5	cpras	2,82
6	vndas	5,59
7	rbc	7,00
8	ct	1,00

Tabela 13: *Variance inflation factor* (VIF) do *dataset* final

Conforme mostrado nos resultados dos VIF da Tabela 13, a coluna `deb` ficou com valor de 10,89, levemente maior que o seu valor de 10,87 no *dataset* reduzido (ver Figura 11). Uma explicação para este acréscimo foi a adição do campo `rbc`, que utiliza condicionalmente o valor de `deb`. O perfil da correlação entre estas duas colunas pode ser verificado na Figura 10.

Por fim, o particionamento dos rótulos MAED no *dataset* final não foi alterado, permanecendo conforme mostrado ao término da Seção 2.4.5 (ver Figura 7).

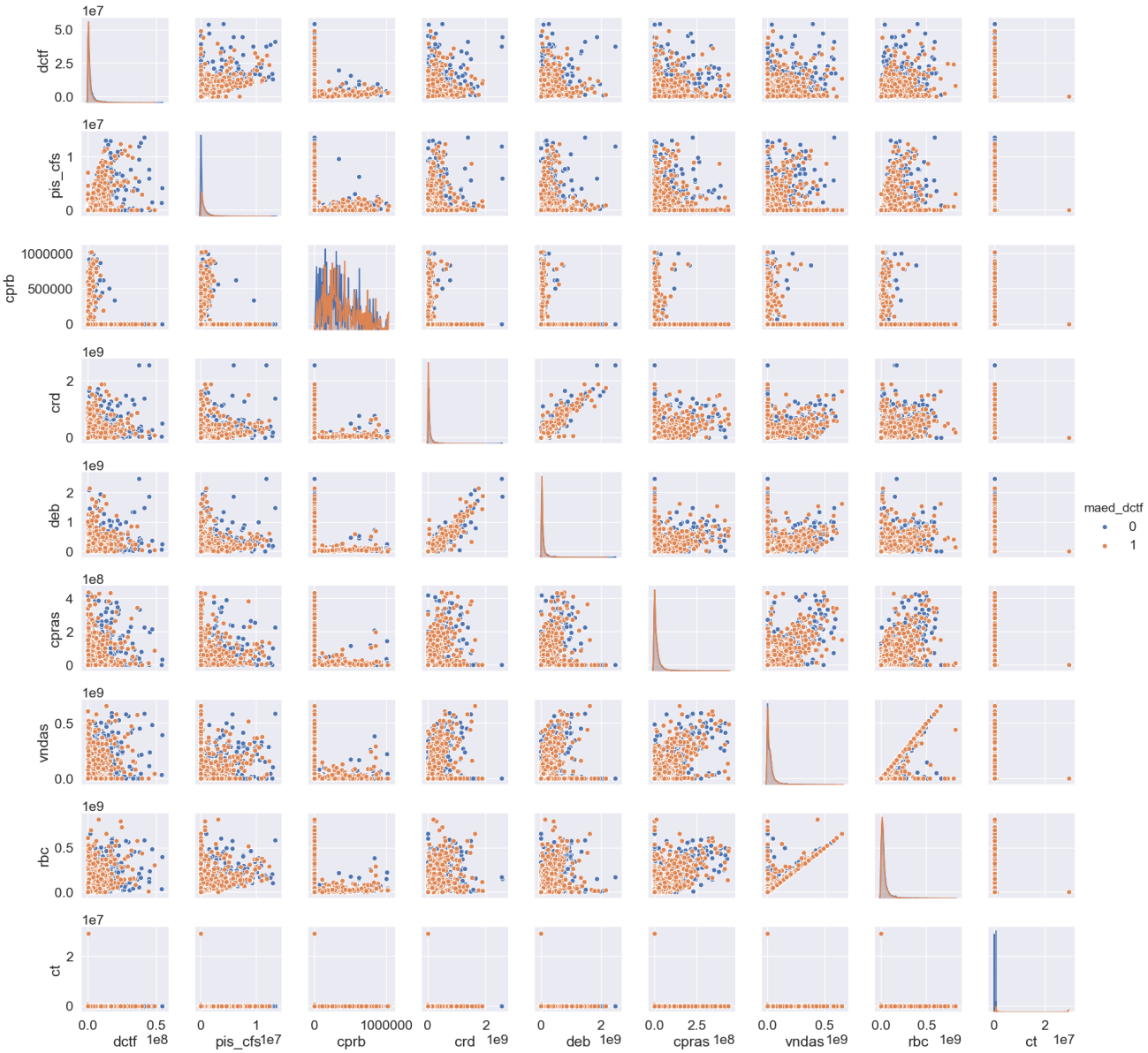


Figura 10: Gráficos de dispersão das *features* do *dataset* final

Ao término desta etapa, o *dataset* final ficou com 10.260 linhas e 14 colunas (Tabela 12), e será utilizado neste formato pelos modelos de *Machine Learning* da Seção 5.

4 Análise e Exploração dos Dados

Nesta seção, apresentaremos algumas análises exploratórias do dados obtidos nas Seções 2 e 3[10].

[10]Deste ponto em diante, utilizaremos, quando conveniente, o termo *dados* para nos referir ao *dataset* final e demais dados deste estudo.

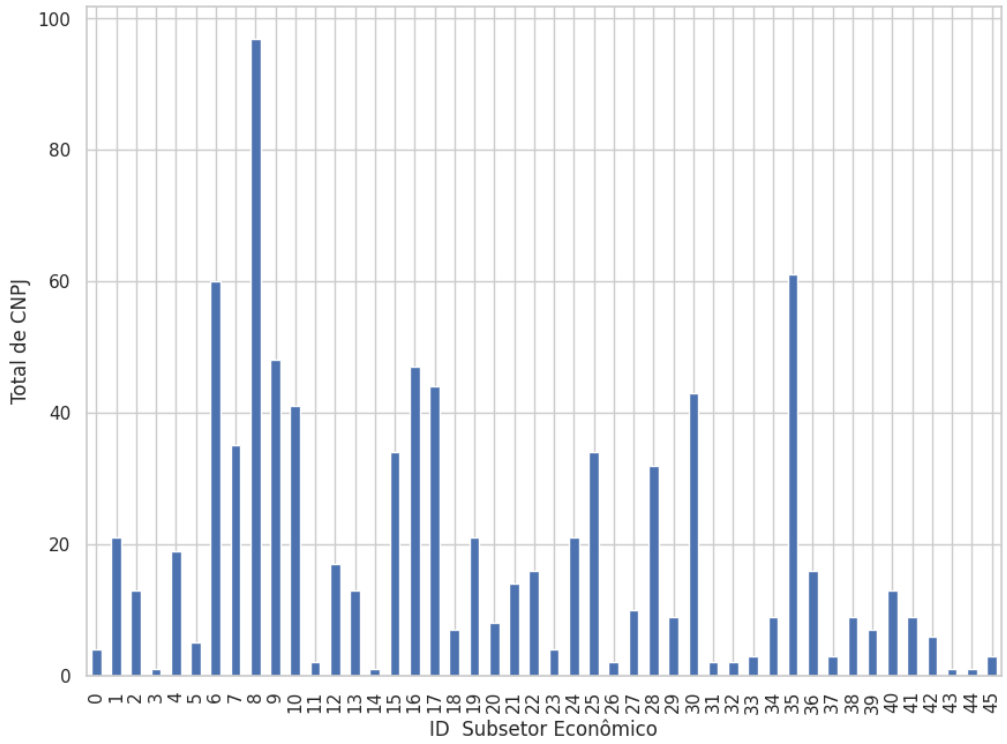


Figura 11: Distribuição de Contribuintes por Subsetor Econômico no ano de 2021

4.1 Distribuição de Contribuintes por Subsetor Econômico

Realizamos uma análise da distribuição dos contribuintes dos nossos dados por subsetor econômico, conforme a seguir.

A área de gestão de grandes contribuintes da RFB classifica os seus contribuintes em uma lista própria de segmentos econômicos, denominada *subsetor econômico*. Esta classificação é realizada anualmente, e iremos basear nossas análises nos resultados para o ano de 2021, que contaram com 1195 contribuintes classificados em 46 subsetores econômicos.

Assim, as análises desta e da próxima seção serão realizadas com um subconjunto dos 3160 contribuintes do nosso universo, que foram os 868 (27,50%) que integraram a lista para o ano de 2021.

A distribuição de contribuintes por subsetor econômico é mostrada na Figura 11, onde o identificador do subsetor econômico (0 a 45) é mostrado no eixo x, e o total de contribuintes é mostrado no eixo y.

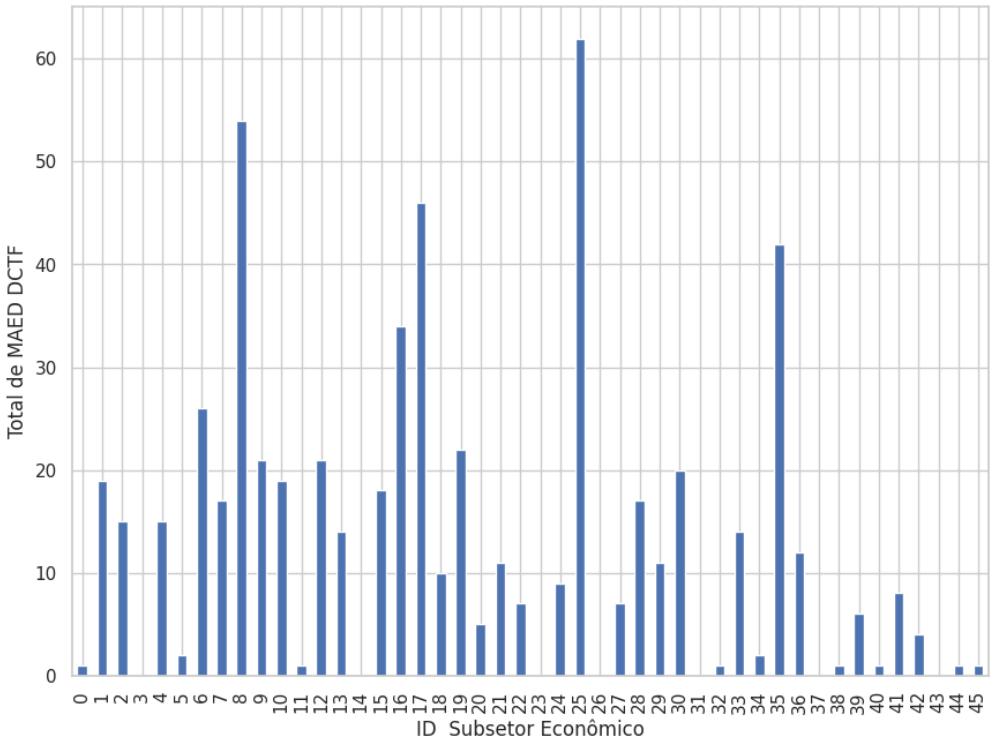


Figura 12: Distribuição de MAED na DCTF por Subsetor Econômico

Conforme mostrado na figura, os contribuintes estão razoavelmente bem distribuídos, com exemplares em todos os subsetores econômicos, sendo que dois deles se destacam. A média foi de 18 CNPJ por subsetor em 2021.

Com isso, podemos considerar, por extrapolação, que não foi verificado viés de subsetor econômico aparente nas nossas análises.

4.2 Distribuição de MAED na DCTF por Subsetor Econômico

Analisamos a distribuição da ocorrência de MAED na DCTF por subsetor econômico, utilizando a mesma classificação descrita na seção anterior.

Nossos resultados são mostrados na Figura 12, onde o identificador do subsetor econômico (0 a 45) é mostrado no eixo x, e o total de ocorrências de MAED (coluna `maed_dctf`) é mostrada no eixo y.

Conforme mostrado na figura, as ocorrência de MAED se distribuem razoavelmente por todos os subsetores econômicos, com apenas sete subsetores com zero ocorrências, quatro que se destacaram mais que os demais, e uma média de 13

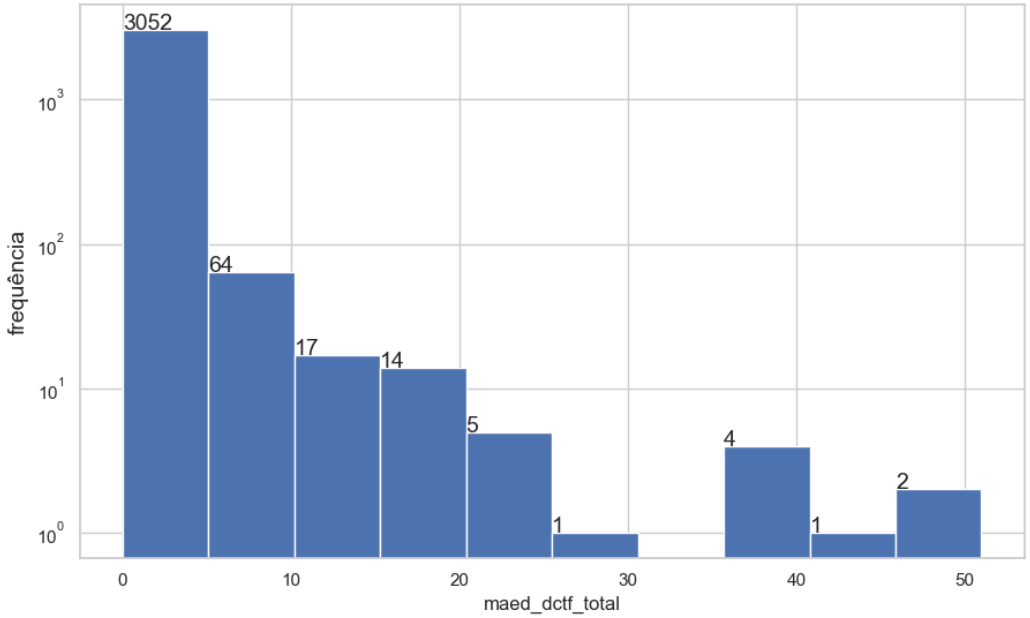


Figura 13: Histograma do total de ocorrências de MAED na DCTF dos contribuintes

ocorrências MAED por subsetor em 2021. Isso complementa nossa avaliação de que não há viés de subsetor econômico aparente nas nossas análises.

4.3 Histograma do total de ocorrências de MAED na DCTF por contribuinte

Analisamos o número de vezes que os contribuintes incorreram em MAED na DCTF (ver Seção 3.3) por meio de um histograma do valor máximo da coluna `maed_dctf_total` agrupada por contribuinte, e os resultados são mostrados na Figura 13. Na figura, as 10 faixas (*bins* de [0 a 4], [5 a 9], etc.) do valor máximo de `maed_dctf_total` são mostradas no eixo x, e a frequência (total de contribuintes) é mostrada no eixo y.

Conforme mostrado na figura, a grande maioria dos contribuintes, 3052 dos 3160—96,58%—ficou com `maed_dctf_total` entre zero e quatro. Isso nos mostra que a conformidade na entrega da DCTF é o comportamento predominante dos contribuintes do universo deste estudo. Vale notar, ainda, a ausência de contribuintes na faixa de 30 a 34.

4.4 Sumário dos Dados

O sumário do nosso *dataset* final não pode ser exibido por conter dados protegidos por sigilo fiscal, conforme descrito na Seção 2.1. Nele, podemos destacar a presença de elevadas faixas de valores das colunas das variáveis tributárias e econômico-financeiras. Isso se deve ao critério de seleção dos contribuintes deste estudo, que envolveu apenas grandes contribuintes (ver Seção 2.4.1).

Outro destaque para estas colunas é valor do desvio padrão bastante superior ao valor médio, mostrando uma forte assimetria na distribuição de valores destas colunas—a presença dos zeros explica, em parte, essa variabilidade.

A média e o desvio padrão da coluna carga tributária (`ct`) revelam a ocorrência de valores de `rbc` menores que `dctf` na expressão `ct = dctf / rbc` (Seção 3.3). Valores de `ct` próximos ou superiores a um não são esperados numa situação de normalidade, e a presença destes altos valores pode ser explicada por erros de preenchimento ou outros motivos. Não obstante, optamos por utilizá-los desta forma por representarem um comportamento com potencial preditivo.

Outra análise que fizemos foi agrupar o *dataset* final por classe `maed_dctf` e calcular os valores médios das *features*, para avaliar a presença de viés de classe nestes valores.

Os resultados obtidos mostraram que as médias das colunas das variáveis tributárias e econômico-financeiras para as duas classes são próximas, não exibindo viés de classe. Porém, a coluna `ct` apresentou média bastante superior para a classe Sim(1).

4.5 *Heatmap* das Correlações do *dataset* Final

Analisamos as correlações das colunas do *dataset* final por meio do *heatmap* dos seus valores. Trata-se de um gráfico da matriz de correlações, onde os seus valores são representados em uma escala de cores, e o resultado é mostrado na Figura 14.

Conforme mostrado na figura, as correlações entre as colunas das variáveis tributárias e econômico-financeiras são relevantes, com exceção da `cprb` e `ct`, que

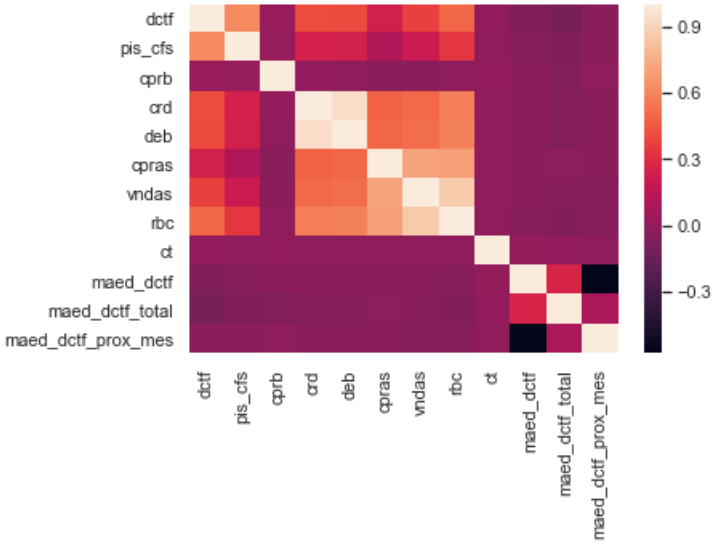


Figura 14: *Heatmap* das correlações do *dataset* final

apresentam valores de correlação mais modestos—esta última, possivelmente devido à prevalência de baixos valores (ver Seção 3.3).

Podemos observar ainda, uma correlação negativa entre as colunas `maed_dctf` e `maed_dctf_prox_mes`, que pode ter sua capacidade preditiva explorada pelos modelos de *Machine Learning*.

5 Criação de Modelos de *Machine Learning*

Na seção anterior, apresentamos todo nosso processo de obtenção dos dados e construção do nosso *dataset*, incluindo a caraterização dos dados, todo o detalhamento do nosso ETL (*extract, transform, and load*), e alguns *insights* obtidos com a análise e exploração dos dados.

Nesta seção, apresentaremos nossos resultados experimentais com os modelos de *Machine Learning* que foram treinados para atacar o problema de predição de atraso na entrega de declarações aplicado à DCTF. Para isso, utilizaremos a definição do problema da Seção 1.2.1 e os dados da Tabela 12.

Neste trabalho, estudamos o desempenho dos modelos *Naïve Bayes*, máquina de vetor de suporte (*Support Vector Machine*), e rede neural (*Neural Network*), para predizer,

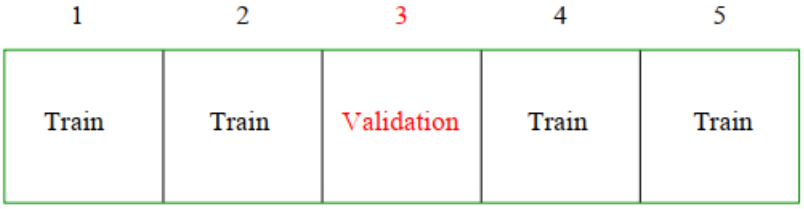


Figura 15: Diagrama de particionamento dos dados para *5-Fold Cross-Validation*. Fonte: (HASTIE; TIBSHIRANI; FRIEDMAN, 2009).

com dados de um determinado mês, se um contribuinte atrasará a entrega da DCTF no próximo mês (ver Seção 1.3).

Escolhemos estes modelos por serem amplamente empregados como classificadores e adequados ao tratamento de problemas não-lineares, que são tipicamente encontrados em muitos problemas reais de aplicação. Outro fator que determinou nossa escolha foi a disponibilidade de implementações eficientes destes modelos em bibliotecas Python, que nos permitiu executá-los no nosso ambiente de desenvolvimento.

Para conduzir nossos experimentos, definimos uma metodologia de treino e validação dos modelos que inclui a técnica *K-Fold Cross-Validation*, conforme a seguir.

5.1 *K-Fold Cross-Validation*

Modelos de *Machine Learning* aprendem a partir de dados, e, portanto, quanto maior a quantidade de dados disponíveis para treinamento, maior a chance de se obter um bom modelo preditivo.

Conforme mostrado na Tabela 12, nosso *dataset* balanceado para treinamento dos modelos tem 10.260 linhas e 14 colunas. Esta quantitativo de linhas é considerado modesto para fins de treinamento de modelos de *Machine Learning*, e foi consequência de optarmos pelo balanceamento das classes do *dataset* intermediário, que era severamente não balanceado (ver Seção 2.4.5). Uma técnica utilizada para contornar esta situação é a chamada *validação cruzada* (*Cross-Validation*—CV), e mais especificamente a *K-Fold Cross-Validation* (HASTIE; TIBSHIRANI; FRIEDMAN, 2009; JAMES et al., 2021).

Na técnica *K-Fold Cross-Validation*, nós dividimos os dados em K partes com o mesmo tamanho, conforme mostrado na Figura 15. Em seguida, selecionamos uma parte para validação do modelo[11](realizar predições com dados não vistos no treinamento), e as outras $K-1$ partes são selecionadas para treinar o modelo. Desta forma, conseguimos maximizar o aproveitamento do *dataset* disponível, e com isso conseguir avaliar o modelo por K vezes.

Neste trabalho, nós empregamos *5-Fold Cross-Validation* em todos os experimentos que realizamos, e assim pudemos comparar seus resultados de forma adequada.

Para isso, nós criamos uma função para orquestrar o processo de treinamento e avaliação dos modelos conforme abaixo (a listagem completa está no Apêndice II deste texto).

```
1  # Model cross validation
2  def model_cross_validation(model, df, K, fold_data, train_model, **kwargs):
3      '''
4      Client application must implement the following functions:
5      - fold_data(df, trn_ids, tst_ids)
6      receives current fold training and testing indexes of df dataframe,
7      and returns trn_data and tst_data (in suitable format) to be used by train_model()
8      - train_model(model, trn_data, tst_data, fold=fold, **kwargs)
9      receives model current fold trn_data and tst_data, and returns
10     trn_mts and tst_mts fold metrics PredResult objects for aggregating
11     '''
```

A função `model_cross_validation()` recebe os seguintes parâmetros:

- `model`, modelo que será treinado e avaliado;
- `df`, Pandas DataFrame contendo o *dataset* com os dados;
- `K`, indicador da ordem do *K-Fold Cross-Validation*;
- `fold_data`, função de *callback* a ser implementada no módulo cliente, que recebe os índices de treino e validação do *fold* atual, referentes ao DataFrame dos dados, e retorna duas variáveis contendo os dados de treino e validação, respectivamente, no formato que será utilizado pela função de treinamento do modelo;
- `train_model`, função de *callback* a ser implementada no módulo cliente, que recebe o modelo e as duas variáveis contendo os dados de treino e validação do *fold*, conduz

[11] Neste trabalho, utilizaremos os termos *validação* e *teste* indistintamente.

o treinamento e avaliação, e retorna dois objetos da classe `PredResult` contendo os resultados obtidos para os dados de treino e validação, respectivamente;

- `**kwargs`, parâmetros opcionais que serão repassados para as funções de *callback*.

O fluxo de trabalho para se treinar um modelo é como a seguir. Primeiro, implementamos a função que irá fornecer os dados de treinamento e validação do *fold* atual para os respectivos índices recebidos. Em seguida, implementamos a função de treinamento dos *folds*. Por fim invocamos o orquestrador para conduzir o treinamento, invocando, para cada *fold*, as funções de *callback* e recebendo seus retornos na forma abaixo.

```
1 trn_data, tst_data = fold_data(df, trn_ids, tst_ids, **kwargs)
2 trn_mts, tst_mts = train_model(model, trn_data, tst_data, fold=fold, **kwargs)
```

Ao término do processo, os resultados do treinamento e validação para cada *fold* e o sumário geral serão produzidos como saída.

Todos os experimentos conduzidos neste trabalho foram treinados por meio da função `model_cross_validation()` com $K = 5$, e os *folds* ficaram com o tamanho de 2.052 linhas ($\frac{10260}{5} = 2052$).

5.2 Preditor Ingênuo

Inicialmente, construímos um modelo de predição ingênuo, com o propósito de verificar a capacidade preditiva da coluna `maed_dctf` dos nossos dados.

Este modelo utiliza o próprio valor da coluna `maed_dctf` como predição de `maed_dctf_prox_mes`. Ou seja, o preditor ingênuo implementa a relação $\hat{y}(p) = y(t)$, com $p > t$, e nosso objetivo foi avaliar o quanto de capacidade preditiva os modelos de *Machine Learning* iriam agregar a esta abordagem ingênua.[12]

[12]Neste preditor, o termo *ingênuo* é empregado no sentido de se esperar que o comportamento do contribuinte no mês atual irá se repetir no mês subsequente.

O código fonte da implementação do modelo é mostrado abaixo.

```
1  # Preditor ingenuo
2  # model_cross_validation() callback handlers implementation
3  # prepare the dataset for a single fold
4  def fold_data(df, train_ids, test_ids):
5      return df.iloc[train_ids], df.iloc[test_ids]
6  # train the model
7  def train_model(model, trn_df, tst_df, fold):
8      print(f'Training model for fold {fold+1} ...')
9      trn_metrics = PredResult(name='Train trn', index_name='epoch')
10     tst_metrics = PredResult(name='Train tst', index_name='epoch', latex=True)
11     # fit model
12     model = lambda df: (df['maed_dctf_prox_mes'], df['maed_dctf'])
13     # evaluate model: trn
14     y_true, y_pred = evaluate_model(model, trn_df)
15     trn_metrics.update(y_true, y_pred)
16     # evaluate model: tst
17     y_true, y_pred = evaluate_model(model, tst_df)
18     tst_metrics.update(y_true, y_pred)
19     return trn_metrics, tst_metrics
20 # Training auxiliary functions
21 def evaluate_model(model, df):
22     return model(df)
23 # Run experiment
24 K=5
25 model_cross_validation(None, df, K, fold_data, train_model)
```

O código define as funções `fold_data()` (linha 4) e `train_model()` (linha 7) para o modelo, e comanda a sua execução na linha 25.

O modelo é criado na linha 12, onde é declarada uma função *lambda* que retorna os valores das colunas `maed_dctf_prox_mes` e `maed_dctf`.

Em seguida, o modelo é executado na linha 17, com a chamada da função `evaluate_model()`, que retorna os valores destas colunas para as variáveis `y_true` e `y_pred`, respectivamente.

Os resultados deste modelo serão apresentados mais adiante na Seção 6.2.

5.3 Preditor *Naïve Bayes*

O classificador *Naïve Bayes* é um método de aprendizado supervisionado baseado na teoria da probabilidade estatística. Ele é fundamentado no *Teorema de Bayes*, que

pode ser definido para o problema de classificação como:

$$P(y\,|\,x_1,\ldots,x_n) = \frac{P(y)\;P(x_1,\ldots,x_n\,|\,y)}{P(x_1,\ldots,x_n)} \tag{3}$$

onde, $P(y\,|\,x_1,\ldots,x_n)$ é a probabilidade do vetor de *features* (uma entrada do classificador) $x_1,\ldots,x_n$ pertencer à classe y (denominada *posterior*); $P(y)$ é a probabilidade incondicional de classe y (frequência relativa da classe y no *dataset*—denominada *prior*); $P(x_1,\ldots,x_n\,|\,y)$ é a probabilidade do vetor de *features* ocorrer dado a classe y (denominada *likehood*); e $P(x_1,\ldots,x_n)$ é a probabilidade incondicional do vetor de *features* ocorrer (denominada *marginalization* ou *evidência*).

O *Naïve Bayes* assume a independência condicional dos pares de *features* (daí o termo *Naïve*), o que alivia a necessidade de calcular as probabilidades conjuntas, e resulta no seguinte modelo de classificação (SCIKIT-LEARN, 2022):

$$\hat{y} = \arg\max_i\; P(y)\;\Pi_{i=1}^{n} P(x_i\,|\,y) \tag{4}$$

onde, a classe predita $\hat{y}$ para o vetor de entrada $x_1,\ldots,x_n$ é aquela que produz o maior valor do produto da frequência relativa de classe y com a multiplicação das probabilidades individuais das *features* condicionadas a esta classe.

Assumir a independência condicional dos pares de *features* pode não ser o mais adequado em muitas situações, mas mesmo assim, o classificador *Naïve Bayes* produz bons resultados, especialmente quando não há dados suficientes para se estimar as probabilidades condicionais. Como resultado, o classificador *Naïve Bayes* exibe uma boa relação entre viés (*bias*) e variância, que leva a um bom desempenho na prática (JAMES et al., 2021).

Nós implementamos o nosso preditor *Naïve Bayes* utilizando a classe `GaussianNB` da biblioteca scikit-learn, que implementa o *Naïve Bayes* calculando as probabilidades $P(x_i\,|\,y)$ com a distribuição Gaussiana.

O modelo *Naïve Bayes* não tem parâmetros gerais, e o código fonte do seu treinamento é mostrado a seguir.

```
1  # Preditor Naive Bayes
2  # model_cross_validation() callback handlers:
3  # prepare the dataset for a single fold
4  def fold_data(df, trn_ids, tst_ids):
5      # preprocess features
6      scaler      = StandardScaler()
7      cols        = df_columns_metric(df)
8      feats       = scaler.fit_transform(df[cols])
9      df[cols] = pd.DataFrame(feats, index=df.index, columns=cols)
10     trn_df = df.iloc[trn_ids]
11     tst_df = df.iloc[tst_ids]
12     return trn_df, tst_df
13 # train the model
14 def train_model(model, trn_df, tst_df, fold):
15     print(f'Training model for fold {fold+1} ...')
16     trn_metrics = PredResult(name='Train trn', index_name='epoch')
17     tst_metrics = PredResult(name='Train tst', index_name='epoch', latex=True)
18     x_trn, y_trn = dataset_input_and_target(trn_df)
19     x_tst, y_tst = dataset_input_and_target(tst_df)
20     # fit model
21     fit_model(model, x_trn, y_trn)
22     # evaluate model: trn
23     y_pred = evaluate_model(model, x_trn)
24     trn_metrics.update(y_trn, y_pred)
25     # evaluate model: tst
26     y_pred = evaluate_model(model, x_tst)
27     tst_metrics.update(y_tst, y_pred)
28     return trn_metrics, tst_metrics
29 # Training auxiliary functions
30 def fit_model(model, x, y):
31     model.fit(x, y)
32 def evaluate_model(model, x):
33     y_pred = model.predict(x)
34     return y_pred
35 # Run experiment
36 df = df_data_original.copy()
37 K=5
38 model = GaussianNB()
39 model_cross_validation(model, df, K, fold_data, train_model)
```

O código define as funções `fold_data()` (linha 4) e `train_model()` (linha 14) para o modelo, e comanda a sua execução na linha 39.

O modelo é criado na linha 38 sem parâmetros de entrada.

Apesar do *Naïve Bayes* não exigir pré-processamento dos dados de entrada, conseguimos melhores resultados normalizando, em cada *fold*, as *features* de entrada que apresentam altos valores numéricos. Isso foi feito por meio da função `StandardScaler()`

da biblioteca scikit-learn, que produziu os z-scores $z = \frac{x-\mu}{\sigma}$ dos valores de cada coluna (linha 8).

Os resultados deste modelo serão apresentados mais adiante na Seção 6.3.

5.4 Preditor *Support Vector Machine*

As *Máquinas de Vetores de Suporte* (*Support Vector Machine*– SVM) pertencem a uma categoria de classificadores denominados classificadores de máxima margem.

Nestes modelos, busca-se encontrar um hiperplano de separação ótimo, que seja o mais distante possível de cada classe dos dados de treinamento—representados como pontos em um espaço multidimensional, com o número de dimensões dado pelo número de *features* (JAMES et al., 2021).

Um exemplo é mostrado na Figura 16, onde temos um hiperplano de separação ótimo obtido para duas classes (azul e lilás) de dados bidimensionais (duas *features*). A *margem* é a distância perpendicular entre a linha sólida e qualquer uma das duas linhas pontilhadas, sendo que os pontos que se encontram nas linhas pontilhadas são denominados *vetores de suporte* (*support vectors*).

Desta forma, o treinamento destes classificadores envolve a busca do hiperplano de separação ótimo por meio de modelos de otimização.

As SVM generalizam e flexibilizam esta abordagem, permitindo tratar problemas não linearmente separáveis por meio da expansão do espaço de *features* utilizando funções denominadas *kernels*, e admitindo que alguns vetores de suporte sejam classificados erradamente. Como resultado, as SVM apresentam boa capacidade de generalização, grande robustez às observações individuais dos dados e a grandes dimensões, e fundamentação teórica bem definida (JAMES et al., 2021; LORENA; CARVALHO, 2003).

Desta forma, os hiperparâmetros básicos de uma SVM são a função de *kernel* e o *coeficiente de regularização* (C), que ajusta o compromisso de troca entre a largura da margem e a quantidade de erros de classificação permitidos para os vetores de suporte.

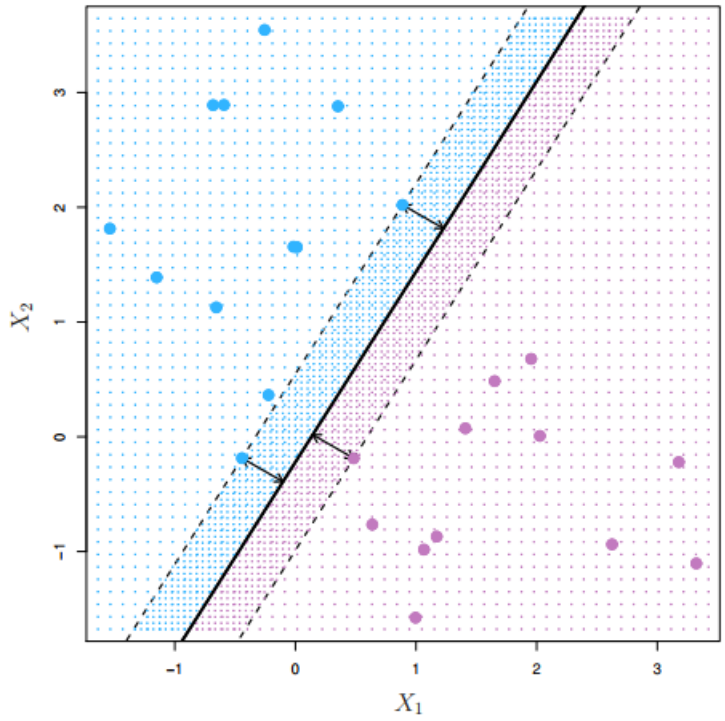


Figura 16: Exemplo de hiperplano de separação ótimo para duas classes (azul e lilás) de dados bidimensionais (duas *features*). Fonte: (JAMES et al., 2021).

Um exemplo de curva de separação da SVM é mostrado na Figura 17, onde temos curvas de separação obtidas para dados não-lineares bidimensionais (duas *features*) de duas classes (azul e lilás) com *kernel* polinomial de grau 3 (esquerda), e *kernel* de função de base radial (direita).

Nós implementamos o nosso preditor SVM utilizando a classe `svm.SVC` da biblioteca scikit-learn. Os detalhes do nosso modelo são descritos a seguir, por meio do código fonte do seu treinamento.

```
 # Preditor Support Vectore Machine (SVM)
 # model_cross_validation() callback handlers:
 # prepare the dataset for a single fold
def fold_data(df, trn_ids, tst_ids):
    # preprocess features
    scaler      = StandardScaler()
    cols        = df_columns_metric(df)
    feats       = scaler.fit_transform(df[cols])
    df[cols] = pd.DataFrame(feats, index=df.index, columns=cols)
    trn_df = df.iloc[trn_ids]
    tst_df = df.iloc[tst_ids]
    return trn_df, tst_df
 # train the model
def train_model(model, trn_df, tst_df, fold):
    print(f'Training model for fold {fold+1} ...')
    trn_metrics = PredResult(name='Train trn', index_name='epoch')
```

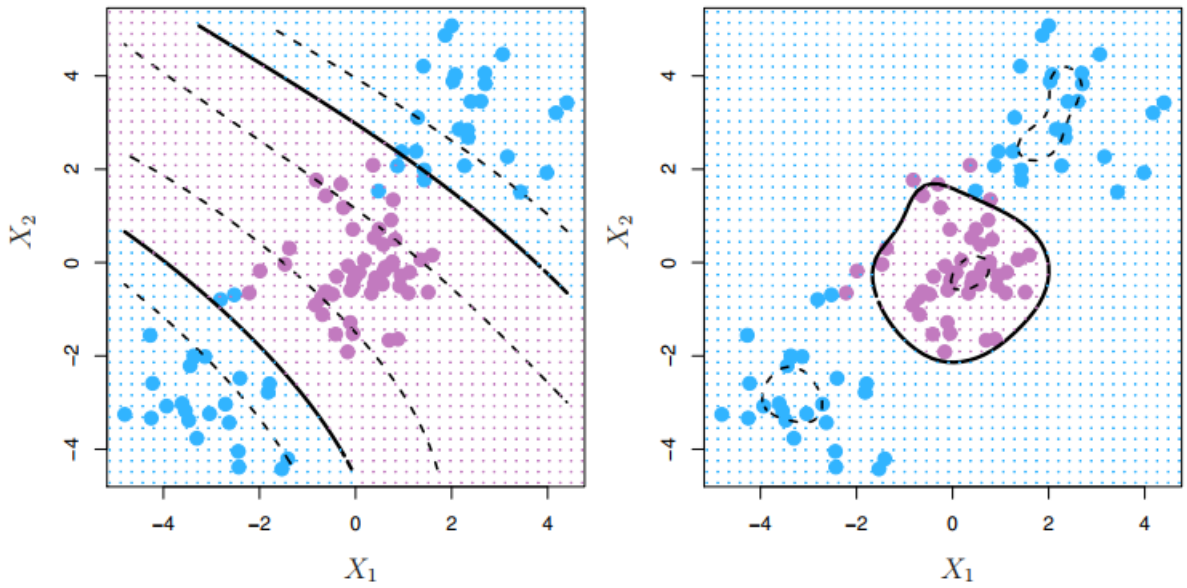


Figura 17: Exemplo de curva de separação da *Support Vector Machine* para dados não-lineares bidimensionais (duas *features*) de duas classes (azul e lilás) com *kernel* polinomial de grau 3 (esquerda), e *kernel* de função de base radial (direita). Fonte: (JAMES et al., 2021).

```
17      tst_metrics = PredResult(name='Train tst', index_name='epoch', latex=True)
18      x_trn, y_trn = dataset_input_and_target(trn_df)
19      x_tst, y_tst = dataset_input_and_target(tst_df)
20      # fit model
21      fit_model(model, x_trn, y_trn)
22      # evaluate model: trn
23      y_pred = evaluate_model(model, x_trn)
24      trn_metrics.update(y_trn, y_pred)
25      # evaluate model: tst
26      y_pred = evaluate_model(model, x_tst)
27      tst_metrics.update(y_tst, y_pred)
28      return trn_metrics, tst_metrics
29   # Training auxiliary functions
30  def fit_model(model, x, y):
31      model.fit(x, y)
32  def evaluate_model(model, x):
33      y_pred = model.predict(x)
34      return y_pred
35   # Run experiment
36  K=5
37  model = svm.SVC(kernel='rbf', C=1.0)
38  model_cross_validation(model, df, K, fold_data, train_model)
```

O código define as funções `fold_data()` (linha 4) e `train_model()` (linha 14) para o modelo, e comanda a sua execução na linha 38.

O modelo é criado na linha 37, e utilizamos a função de base radial (`rbf`) como *kernel*, e um coeficiente de regularização de 1.

As colunas das (*features*) dos dados de entrada apresentam altos valores numéricos, o que dificulta a convergência do algoritmo de treinamento da SVM, gerando soluções sub-ótimas. Assim, para cada *fold*, aplicamos um pré-processamento a estas colunas, que tiveram seus valores normalizadas (coluna por coluna) por meio da função `StandardScaler()` da biblioteca scikit-learn, que produziu os z-scores $z = \frac{x-\mu}{\sigma}$ dos valores de cada coluna (linha 8).

Os hiperparâmetros do nosso preditor SVM foram determinados empiricamente, a partir de escolhas padrão para classificadores binários encontradas na literatura, e da condução de diversos experimentos exploratórios.

Os resultados deste modelo serão apresentados mais adiante na Seção 6.4.

5.5 Preditor baseado em Rede Neural

De forma simplificada, o modelo de computação do cérebro humano utiliza unidades elementares de processamento, denominadas *neurônios*, que são massivamente interconectadas segundo determinadas topologias (HAYKIN, 1999).

Um modelo de neurônio artificial é mostrado na Figura 18. Conforme mostrado na figura, o neurônio recebe um conjunto de valores x_i na sua entrada, calcula a soma ponderada v destes valores pelos seus respectivos pesos w_i, e aplica uma função não linear $\phi(v)$, denominada *função de ativação*, para gerar a sua saída.

Dentre as funções de ativação mais utilizadas, temos a ReLU (*rectified linear unit*) e a sigmóide, que são mostradas na Figura 19.

Diversos neurônios podem ser interconectados e organizados em camadas para formar uma rede neural, conforme mostrado na Figura 20, onde temos uma rede com uma camada interna[13], ou *escondida*, cujas saídas y_j são produzidas para as entradas x_i.

Os algoritmos de treinamento têm por finalidade utilizar os dados para ajustar os pesos da rede de forma a produzir as saída desejadas.

[13]Tipicamente, o número de camadas da rede considera apenas as camadas internas, não considerando as entradas nem as saídas.

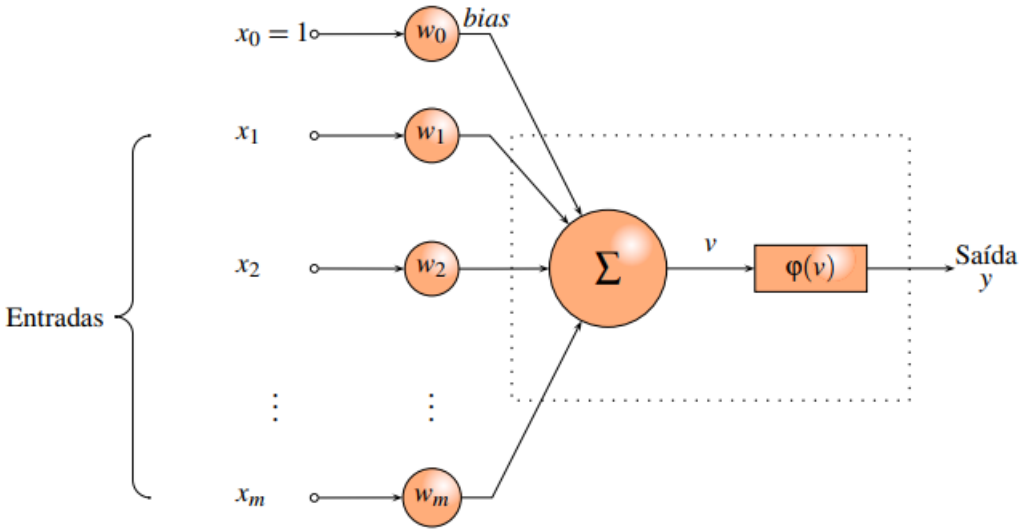


Figura 18: Modelo de neurônio artificial. Fonte: (FREITAS, 2010).

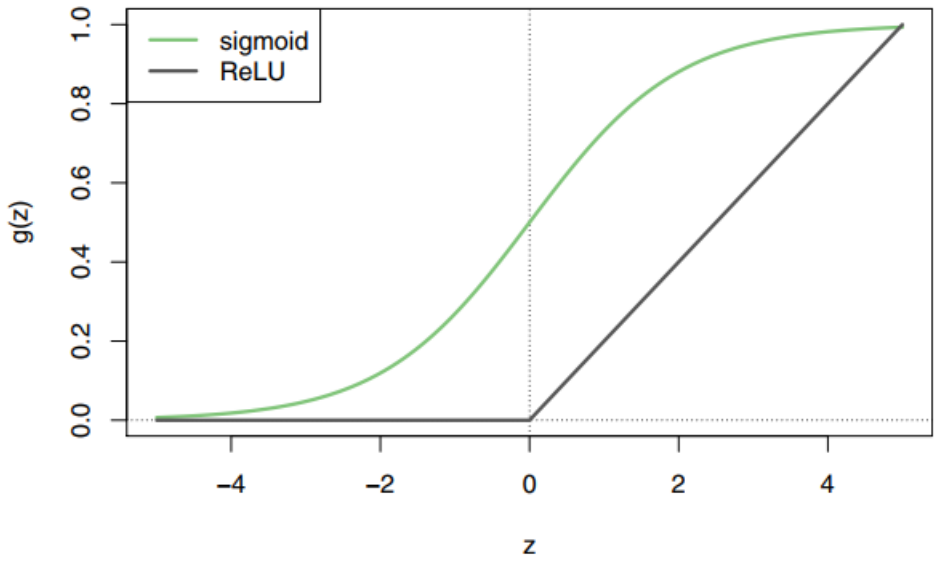


Figura 19: Funções de ativação sigmóide e ReLU. Fonte: (JAMES et al., 2021).

Cada problema de interesse irá definir as escolhas de topologia de rede, o tipo de aprendizado—supervisionado, semi-supervisionado, ou não-supervisionado—, e o algoritmo de treinamento utilizado.

Os algoritmos de treinamento supervisionado sucessivamente ajustam os pesos da rede em função do erro observado na saída, denominado *perda* (*loss*), para uma entrada apresentada. A intensidade deste ajuste é basicamente determinada pela *taxa de aprendizado* (α), sendo que determinados algoritmos podem utilizar parâmetros adicionais como *momento* e *regularização*. Um ciclo de treinamento que emprega todos

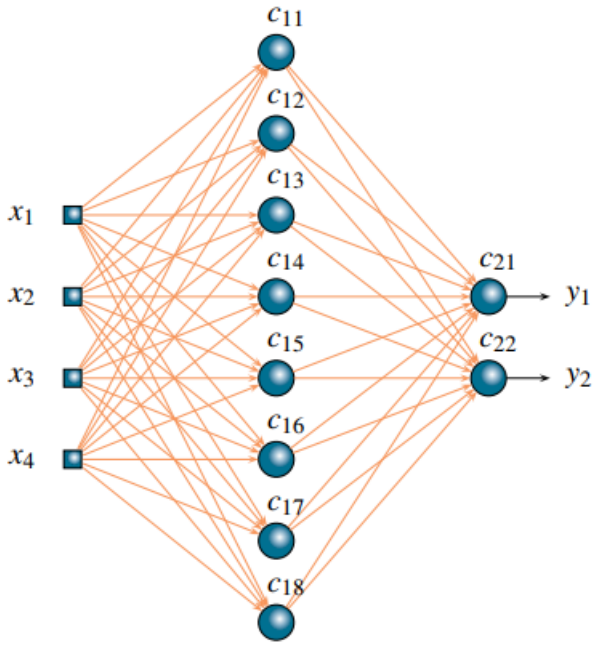


Figura 20: Exemplo de Rede Neural com uma camada interna. Fonte: (FREITAS, 2010).

os pares de entrada e saída do *dataset* de treinamento é denominado *época* (HAYKIN, 1999).

Desta forma, podemos considerar a topologia, a taxa de aprendizado, e número de épocas de treinamento como sendo os hiperparâmetros básicos do modelo de rede neural.

Implementamos o nosso preditor baseado em rede neural utilizando a biblioteca PyTorch, por meio da classe `MLP` e um conjunto de classes e métodos auxiliares.

Nossa classe `MLP` recebe como entrada uma estrutura de dados com a representação da topologia da rede como um conjunto de strings, e instancia um modelo de rede neural do tipo *multilayer perceptron* totalmente conectada. O código fonte da classe `MLP` e seus artefatos auxiliares é listado no Apêndice I.

Os detalhes do nosso modelo são descritos a seguir, por meio do código fonte do seu treinamento.

```
 # Preditor baseado em Rede Neural
 # model_cross_validation() callback handlers:
 # prepare the dataset for a single fold
def fold_data(df, trn_ids, tst_ids, epochs, lr, momentum, verbose, plot):
    # preprocess features
    scaler  = StandardScaler()
    cols    = df_columns_metric(df)
    feats   = scaler.fit_transform(df[cols])
    df[cols] = pd.DataFrame(feats, index=df.index, columns=cols)
    # adjusts data types
    df['maed_dctf']     = df['maed_dctf'].astype(np.float64)
    df['maed_dctf_total'] = df['maed_dctf_total'].astype(np.float64)
    # load the dataset
    dataset = DataSet(df)
    # Sample elements randomly from a given list of ids, no replacement.
    trn_subsampler = torch.utils.data.SubsetRandomSampler(trn_ids)
    tst_subsampler = torch.utils.data.SubsetRandomSampler(tst_ids)
    # Define data loaders for training and testing data in this fold
    trn_dl = torch.utils.data.DataLoader(dataset, batch_size=100, sampler=trn_subsampler)
    tst_dl = torch.utils.data.DataLoader(dataset, batch_size=100, sampler=tst_subsampler)
    return trn_dl, tst_dl
 # train the model
def train_model(model, trn_dl, tst_dl, fold, epochs, lr, momentum, verbose, plot):
    print(f'Training model during {epochs} epochs ...')
    losses = []
    accuracies = []
    # Set model inits
    torch.manual_seed(0)
    model.reset_weights()
    # define the optimization
    criterion = BCELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    trn_metrics = PredResult(name='Train trn', index_name='epoch')
    tst_metrics = PredResult(name='Train tst', index_name='epoch', latex=True)
    loss_trn = []
    # enumerate epochs
    for epoch in range(epochs):
        # train mini-batches
        current_loss = []
        count = 0
        for i, (inputs, targets) in enumerate(trn_dl):
            # clear the gradients
            optimizer.zero_grad()
            # compute the model output
            yhat = model(inputs)
            # calculate loss
            loss = criterion(yhat, targets)
            # credit assignment
            loss.backward()
            # update model weights
            optimizer.step()
            # Print statistics
            current_loss.append(loss.item())
            count += 1
        if epochs < 50 or not epoch \% 49:
            trn_epoch_loss = sum(current_loss)/len(current_loss)
            loss_trn.append(trn_epoch_loss)
```

```
            # evaluate mini-batches training and record performance
            y_true, y_pred, trn_epoch_accuracy = evaluate_model(model, trn_dl)
            trn_metrics.update(y_true, y_pred, epoch+1)
            y_true, y_pred, tst_epoch_accuracy = evaluate_model(model, tst_dl)
            tst_metrics.update(y_true, y_pred, epoch+1)
        if not epoch \% 99:
            if verbose:
                print(f'Epoch {epoch+1} partial result: average loss={trn_epoch_loss:.4f}')
                trn_metrics.display_metrics()
                tst_metrics.display_metrics()
    if plot:
        plot_training(loss_trn, trn_metrics, tst_metrics, fold)
    return trn_metrics, tst_metrics
 # Training auxiliary functions
 # evaluate the model
def evaluate_model(model, test_dl):
    predictions, actuals = list(), list()
    for i, (inputs, targets) in enumerate(test_dl):
        # evaluate the model on the test set
        yhat = model(inputs)
        # retrieve numpy array
        yhat = yhat.detach().numpy()
        actual = targets.numpy()
        actual = actual.reshape((len(actual), 1))
        # round to class values
        yhat = yhat.round()
        # store
        predictions.append(yhat)
        actuals.append(actual)
    predictions, actuals = vstack(predictions), vstack(actuals)
    # calculate accuracy
    accuracy = accuracy_score(actuals, predictions)
    return actuals, predictions, accuracy
 # predict one row of data
def predict(row, model):
    # convert row to data
    row = Tensor([row])
    # make prediction
    yhat = model(row)
    # retrieve numpy array
    yhat = yhat.detach().numpy()
    return yhat
 # Run experiment
 #TOPOLOGIA 1
topology = Dict()
topology.name             = None
topology.input_size       =  df.shape[1]-1 # remove target column count
topology.layers           = 'inp : 64          : 32         : 2             :1'
topology.activations      = 'inp : relu        : relu       : relu          : sigmoid'
topology.initializations = 'inp : kaiming_relu : kaiming_relu : kaiming_relu : xavier'
K       = 5
epochs  = 500
lr      = 0.01
momentum = 0 # unused with Adam (default SGD 0.95)
model = MLP(topology)
model_cross_validation(model, df, K, fold_data, train_model, epochs=epochs, lr=lr,
    momentum=momentum, verbose=False, plot=True)
```

O código define as funções `fold_data()` (linha 4) e `train_model()` (linha 23) para o modelo, e comanda a sua execução na linha 113.

O modelo é criado na linha 112, onde é instanciado o objeto `MLP` com a nossa topologia de rede neural.

A topologia é definida nas linhas 102 a 107 como sendo uma rede totalmente conectada, com 13 entradas (número de colunas do *dataset* menos a coluna do rótulo `maed_dctf_prox_mes`), três camadas internas, com 64, 32, e 2 neurônios, respectivamente, e saída com um neurônio.

Nós treinamos nosso preditor com taxa de aprendizado de 0,01 durante 500 épocas, utilizando o algoritmo Adam (KINGMA; BA, 2014)—linha 32—, e função de perda *Binary Cross Entropy*–BCE (linha 31).

O Adam tem sido extensivamente aplicado em *deep learning*, exibindo vantagens em relação aos métodos tradicionais de otimização estocástica (BROWNLEE, 2022). A BCE é uma escolha padrão para função de perda em classificadores binários (GODOY, 2018), e computa o log das probabilidades dos exemplos pertencerem às suas respectivas classes.

A função de ativação escolhida para os neurônios das camadas internas foi a ReLU, que é a escolha padrão para camadas internas com os algoritmos de treinamento modernos, por conta da sua eficiência computacional; e a função de ativação do neurônio de saída foi a sigmóide, que é necessária quando se utiliza a função de perda `BCELoss()` (linha 31).

A inicialização dos pesos foi realizada com a função `kaiming_uniform_()` para as camadas internas, e com a função `xavier_uniform_()` para a camada de saída. Estas escolhas se deram por conta dos tipos de função de ativação utilizadas nestas camadas.

As colunas das *features* dos dados de entrada apresentam altos valores numéricos, o que gera saturação nas saídas dos neurônios da rede e prejudica a sua convergência. Assim, para cada *fold*, aplicamos um pré-processamento a estas colunas, que tiveram seus valores normalizadas (coluna por coluna) por meio da função

`StandardScaler()` da biblioteca scikit-learn, que produziu os z-scores $z = \frac{x-\mu}{\sigma}$ dos valores de cada coluna (linha 8).

Todos os hiperparâmetros de treinamento do nosso preditor baseado em rede neural foram determinados empiricamente, a partir de escolhas padrão para classificadores binários com redes neurais encontrada na literatura, e da condução de diversos experimentos exploratórios.

Os resultados deste modelo serão apresentados mais adiante na Seção 6.5.

6 Apresentação dos Resultados

Na seção anterior, apresentamos os detalhes da criação dos nossos preditores para o problema de predição de atraso na entrega de declarações aplicado à DCTF.

Nesta seção, apresentaremos os resultados da execução dos modelos. Inicialmente, definiremos as métricas de avaliação que utilizamos, e em seguida apresentaremos os resultados de validação (ver Seção 5.1) obtidos para cada modelo. Por fim, será apresentado um sumário do desempenho dos preditores.

6.1 Métricas de Desempenho

Os resultados dos preditores foram avaliados por diversas métricas padrão para problemas de classificação, que serão descritas nesta seção.

Conforme apresentado na Seção 1.2.1, nossos preditores implementaram modelos de aprendizado supervisionado denominados de classificação binária, onde o objetivo é atribuir a cada entrada os rótulos binários “Sim” (positivo, 1) ou “Não” (negativo, 0), de forma a identificar corretamente a classe a qual os dados de entrada pertencem—no nosso caso, se atrasará ou não a entrega da próxima DCTF.

Desta forma, os modelos receberam como entrada o vetor $\mathbf{X}$ com os dados de um determinado mês (linha do DataFrame com os valores das *features*), e produziram como saída o valor predito $\hat{y} = \{0, 1\}$, sendo que o valor 1 indica que o contribuinte irá atrasar a entrega da declaração, e 0 indica o contrário. Esta saída foi, então,

		Real	
		Positivo	*Negativo*
Predito	*Positivo*	**verdadeiro positivo (tp)**	**falso positivo (fp)**
	Negativo	**falso negativo (fn)**	**verdadeiro negativo (tn)**

Tabela 14: Matriz de confusão

comparada com o valor real observado $y = \{0, 1\}$ da coluna maed_dctf_prox_mes, e contabilizado um acerto ou um erro.

Os erros acertos e erros podem ser divididos nas seguintes possibilidades:

$\hat{y} = 1$ e $y = 1$, denominado *verdadeiro positivo* (*true positive*–tp);
$\hat{y} = 1$ e $y = 0$, denominado *falso positivo* (*false positive*–fp);
$\hat{y} = 0$ e $y = 1$, denominado *falso negativo* (*false negative*–fn);
$\hat{y} = 0$ e $y = 0$, denominado *verdadeiro negativo* (*true negative*–tn).

Estes erros são organizados em forma matricial, na denominada *matriz de confusão* (*confusion matrix*–CM), conforme apresentado na Tabela 14. Os valores das células da CM tipicamente são a contagem dos resultados, ou seus percentuais em relação ao total de resultados. Conforme mostrado na tabela, o preditor ideal tem valores diferentes de zero somente na sua diagonal principal (tp e tn)—as demais variáveis (fp e fn) são zero.

Assim, as métricas de desempenho utilizadas neste estudo são definidas nas equações a seguir, onde também mostramos o sentido de melhoria dos seus valores como sendo ⇧ para quanto maior melhor, e ⇩ para quanto menor melhor.

$$FPR = \frac{fp}{fp+tn} = \frac{fp}{N} \quad \Downarrow \tag{5}$$

$$FNR = \frac{fn}{tp+fn} = \frac{fn}{P} \quad \Downarrow \tag{6}$$

$$TNR = \frac{tn}{tn+fp} = \frac{tn}{N} \quad \Uparrow \tag{7}$$

$$NPV = \frac{tn}{tn+fn} = \frac{tn}{\hat{N}} \quad \Uparrow \tag{8}$$

$$FDR = \frac{fp}{tp+fp} = \frac{fp}{\hat{P}} \quad \Downarrow \tag{9}$$

$$Recall = \frac{tp}{tp+fn} = \frac{tp}{P} \quad \Uparrow \tag{10}$$

$$Precision = \frac{tp}{tp+fp} = \frac{tp}{\hat{P}} \quad \Uparrow \tag{11}$$

$$Accuracy = \frac{tp+tn}{tp+fp+fn+tn} = \frac{tp+tn}{P+N} \quad \Uparrow \tag{12}$$

Nas equações acima, temos que P é o total de exemplos positivos, N é o total de exemplos negativos, $\hat{P}$ é o total de predições positivas, $\hat{N}$ é o total de predições negativas, e:

- FPR é a *taxa de falsos positivos* (*False Positive Rate/Type I error*), que representa razão de falsos positivos para o total de exemplos negativos (Eq. 5);

- FNR é a *taxa de falsos negativos* (*False Negative Rate/Type II error*), que representa a razão de falsos negativos para o total de exemplos positivos (Eq. 6);

- TNR é a *taxa de verdadeiros negativos*, ou, *especificidade* (*True Negative Rate/Specificity*), que representa a razão de verdadeiros negativos para o total de exemplos negativos (Eq. 7);

- NPV é o *valor preditivo negativo* (*Negative Predictive Value*), que representa a razão de verdadeiros negativos para o total de predições negativas (Eq. 8);

- FDR é a *taxa de falsa descoberta* (*False Discovery Rate*), que representa a razão de falsos positivos para o total de predições positivas (Eq. 9);

- Recall, ou TPR, ou Sensibilidade, é a *revocação* (*True Positive Rate/Recall/Sensibility*), que representa a razão de verdadeiros positivos para o total de exemplos positivos (Eq. 10);

- Precision, ou PPV, é a *precisão* (*Positive Predictive Value*), que representa a razão de verdadeiros positivos para o total de predições positivas (Eq. 11); e

- Accuracy é a *acurácia*, que representa a razão das predições corretas pelo total de exemplos ou predições realizadas (Eq. 12), portanto, avalia o desempenho global do preditor. É uma métrica de desempenho bastante adequada para *datasets* balanceados.

Os valores das métricas acima definidas também podem ser apresentados como percentuais, quando são multiplicados por 100.

6.2 Resultados do Preditor Ingênuo

Esta seção mostra os resultados obtidos para o preditor ingênuo, conforme os métodos de criação e execução do modelo apresentados na Seção 5.2.

Os resultados de validação do preditor obtidos para cada *fold* do *5-Fold Cross Validation* são apresentados na Tabela 15, o sumário destes resultados é apresentado na Tabela 16, e a matriz de confusão para o *fold* (2) com a maior acurácia é mostrada na Tabela 17.

Conforme mostrado nas tabelas, o preditor ingênuo apresentou desempenho ruim em todas as métricas, alcançando uma acurácia média de 23%—um desempenho pior que o acaso (50%), com valores similares de precisão e revocação.

Desta forma, podemos verificar que a coluna `maed_dctf` isoladamente não apresentou capacidade preditiva, justificando, assim, a exploração de modelos de *Machine Learning* no problema proposto.

fold	FPR	FNR	TNR	NPV	FDR	Recall	Precision	Accuracy
1	0,81	0,78	0,19	0,21	0,80	0,22	0,20	0,21
2	0,78	0,75	0,22	0,21	0,75	0,25	0,25	0,23
3	0,79	0,80	0,21	0,20	0,80	0,20	0,20	0,20
4	0,79	0,79	0,21	0,21	0,80	0,21	0,20	0,21
5	0,79	0,80	0,21	0,21	0,79	0,20	0,21	0,21

Tabela 15: Resultados de validação do preditor ingênuo.

	FPR	FNR	TNR	NPV	FDR	Recall	Precision	Accuracy
count	5,00	5,00	5,00	5,00	5,00	5,00	5,00	5,00
mean	0,79	0,78	0,21	0,21	0,79	0,22	0,21	0,21
std	0,01	0,02	0,01	0,00	0,02	0,02	0,02	0,01
min	0,78	0,75	0,19	0,20	0,75	0,20	0,20	0,20
25%	0,79	0,78	0,21	0,21	0,79	0,20	0,20	0,21
50%	0,79	0,79	0,21	0,21	0,80	0,21	0,20	0,21
75%	0,79	0,80	0,21	0,21	0,80	0,22	0,21	0,21
max	0,81	0,80	0,22	0,21	0,80	0,25	0,25	0,23

Tabela 16: Sumário dos resultados de validação do preditor ingênuo.

	1	0
1	262	774
0	800	216

Tabela 17: Matriz de confusão do preditor ingênuo para o *fold* (2) com a maior acurácia.

6.3 Preditor *Naïve Bayes*

Esta seção mostra os resultados obtidos para o preditor *Naïve Bayes*, conforme os métodos de criação e execução do modelo apresentados na Seção 5.3.

Os resultados de validação do preditor obtidos para cada *fold* do *5-Fold Cross Validation* são apresentados na Tabela 18, o sumário destes resultados é apresentado na Tabela 19, e a matriz de confusão para o *fold* (5) com a maior acurácia é mostrada na Tabela 20.

Conforme mostrado nas tabelas, o preditor *Naïve Bayes* alcançou resultados médios muito modestos para acurácia (57%) e precisão (57%). Contudo, alcançou ótimos resultados de revocação, com valor médio de 93% e máximo de 98%.

fold	FPR	FNR	TNR	NPV	FDR	Recall	Precision	Accuracy
1	0,93	0,04	0,07	0,66	0,51	0,96	0,49	0,50
2	0,92	0,04	0,08	0,63	0,47	0,96	0,53	0,53
3	0,93	0,03	0,07	0,71	0,49	0,97	0,51	0,52
4	0,91	0,02	0,09	0,78	0,49	0,98	0,51	0,52
5	0,21	0,21	0,79	0,79	0,21	0,79	0,79	0,79

Tabela 18: Resultados de validação do preditor *Naïve Bayes*.

	FPR	FNR	TNR	NPV	FDR	Recall	Precision	Accuracy
count	5,00	5,00	5,00	5,00	5,00	5,00	5,00	5,00
mean	0,78	0,07	0,22	0,72	0,43	0,93	0,57	0,57
std	0,32	0,08	0,32	0,07	0,13	0,08	0,13	0,12
min	0,21	0,02	0,07	0,63	0,21	0,79	0,49	0,50
25%	0,91	0,03	0,07	0,66	0,47	0,96	0,51	0,52
50%	0,92	0,04	0,08	0,71	0,49	0,96	0,51	0,52
75%	0,93	0,04	0,09	0,78	0,49	0,97	0,53	0,53
max	0,93	0,21	0,79	0,79	0,51	0,98	0,79	0,79

Tabela 19: Sumário dos resultados de validação do preditor *Naïve Bayes*.

	1	0
1	822	214
0	214	802

Tabela 20: Matriz de confusão do preditor *Naïve Bayes* para o *fold* (5) com a maior acurácia.

Os resultados médios para as demais métricas não foram consistentes com seus sentidos de melhoria, apresentando viés de classe (FPR de 78% *versus* FNR de 7%).

Vale notar ainda, o desvio-padrão (std) relevante para diversas métricas, que pode ser verificado na variabilidade dos resultados individuais dos *folds*—o *fold* 5 apresentou resultados bem melhores que os demais.

6.4 Preditor *Support Vector Machine*

Esta seção mostra os resultados obtidos para o preditor *Support Vector Machine* (SVM), conforme os métodos de criação e execução do modelo apresentados na Seção 5.4. Os resultados de validação do preditor obtidos para cada *fold* do *5-Fold Cross Validation* são apresentados na Tabela 21, o sumário destes resultados é apresentado

fold	FPR	FNR	TNR	NPV	FDR	Recall	Precision	Accuracy
1	0,25	0,13	0,75	0,86	0,23	0,87	0,77	0,81
2	0,27	0,17	0,73	0,80	0,23	0,83	0,77	0,78
3	0,25	0,11	0,75	0,87	0,22	0,89	0,78	0,82
4	0,26	0,11	0,74	0,87	0,23	0,89	0,77	0,81
5	0,25	0,13	0,75	0,85	0,22	0,87	0,78	0,81

Tabela 21: Resultados de validação do preditor *Support Vector Machine*.

	FPR	FNR	TNR	NPV	FDR	Recall	Precision	Accuracy
count	5,00	5,00	5,00	5,00	5,00	5,00	5,00	5,00
mean	0,26	0,13	0,74	0,85	0,23	0,87	0,77	0,81
std	0,01	0,02	0,01	0,03	0,01	0,02	0,01	0,01
min	0,25	0,11	0,73	0,80	0,22	0,83	0,77	0,78
25%	0,25	0,11	0,74	0,85	0,22	0,87	0,77	0,81
50%	0,25	0,13	0,75	0,86	0,23	0,87	0,77	0,81
75%	0,26	0,13	0,75	0,87	0,23	0,89	0,78	0,81
max	0,27	0,17	0,75	0,87	0,23	0,89	0,78	0,82

Tabela 22: Sumário dos resultados de validação do preditor *Support Vector Machine*.

	1	0
1	913	259
0	116	764

Tabela 23: Matriz de confusão do preditor *Support Vector Machine* para o *fold* (3) com a maior acurácia.

na Tabela 22, e a matriz de confusão para o *fold* (3) com a maior acurácia é mostrada na Tabela 23.

Conforme mostrado nas tabelas, o preditor SVM alcançou bons resultados médios, com acurácia de 81%, precisão de 77%, e revocação de 87%.

Todas as demais métricas apresentaram resultados condizentes com seus sentidos de melhoria. Vale notar também, o baixo desvio-padrão (std) de todas as métricas, como consequência da consistência do desempenho do preditor em todos os *folds*.

6.5 Preditor baseado em Rede Neural

Esta seção mostra os resultados obtidos para o preditor baseado em rede neural, conforme os métodos de criação e execução do modelo apresentados na Seção 5.5.

fold	FPR	FNR	TNR	NPV	FDR	Recall	Precision	Accuracy
1	0,22	0,16	0,78	0,82	0,21	0,84	0,79	0,81
2	0,23	0,18	0,77	0,81	0,22	0,82	0,78	0,79
3	0,25	0,16	0,75	0,82	0,22	0,84	0,78	0,80
4	0,26	0,16	0,74	0,82	0,24	0,84	0,76	0,79
5	0,24	0,17	0,76	0,83	0,23	0,83	0,77	0,80

Tabela 24: Resultados de validação do preditor baseado em rede neural.

	FPR	FNR	TNR	NPV	FDR	Recall	Precision	Accuracy
count	5,00	5,00	5,00	5,00	5,00	5,00	5,00	5,00
mean	0,24	0,17	0,76	0,82	0,22	0,83	0,78	0,80
std	0,02	0,01	0,02	0,01	0,01	0,01	0,01	0,01
min	0,22	0,16	0,74	0,81	0,21	0,82	0,76	0,79
25%	0,23	0,16	0,75	0,82	0,22	0,83	0,77	0,79
50%	0,24	0,16	0,76	0,82	0,22	0,84	0,78	0,80
75%	0,25	0,17	0,77	0,82	0,23	0,84	0,78	0,80
max	0,26	0,18	0,78	0,83	0,24	0,84	0,79	0,81

Tabela 25: Sumário dos resultados de validação do preditor baseado em rede neural.

	1	0
1	872	242
0	163	775

Tabela 26: Matriz de confusão do preditor baseado em rede neural para o *fold* (1) com a maior acurácia.

Os resultados de validação do preditor obtidos para cada *fold* do *5-Fold Cross Validation* são apresentados na Tabela 24, o sumário destes resultados é apresentado na Tabela 25, e a matriz de confusão para o *fold* (1) com a maior acurácia é mostrada na Tabela 26.

Conforme mostrado nas tabelas, o preditor baseado em rede neural alcançou bons resultados médios, com acurácia de 80%, precisão de 78%, e revocação de 83%.

Todas as demais métricas apresentaram resultados condizentes com seus sentidos de melhoria. Vale notar ainda, o baixo desvio-padrão (std) de todas as métricas, como consequência da consistência do desempenho do preditor em todos os *folds*.

A convergência do treinamento para o *fold* 1 é mostrada na Figura 21, onde podemos verificar que a rede convergiu mais fortemente nas primeiras 50 épocas, e após isso iniciou uma convergência mais assintótica.

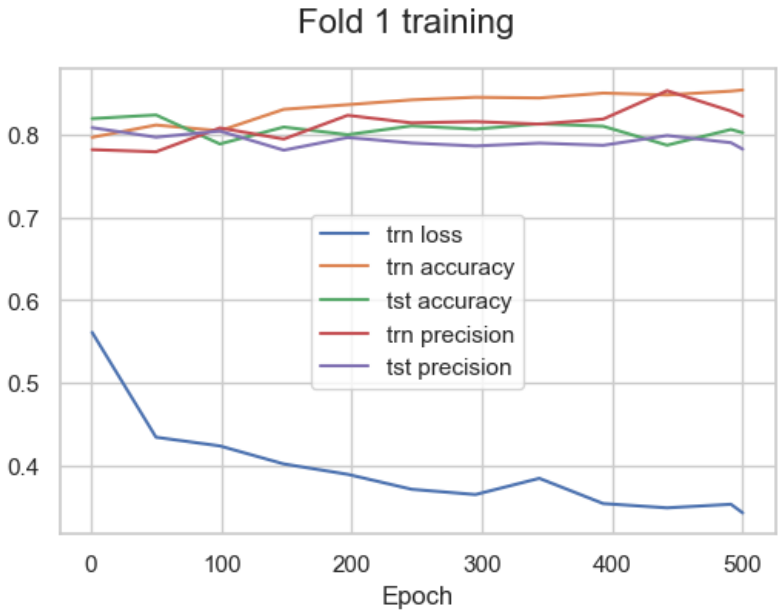


Figura 21: Evolução do treinamento da Rede Neural para o *fold* 1.

Investigamos estre padrão de convergência treinando o modelo com números maiores de épocas (até 10.000). Esse treinamento adicional gerou *overfitting* nos pesos da rede e os resultados das métricas não melhoraram significativamente. Outras topologias de rede `MLP` foram investigadas, mas, também não alcançaram melhorias significativas nos resultados.

7 Interpretação dos Resultados

Na seção anterior, apresentamos os resultados experimentais que obtivemos com os modelos que estudamos neste trabalho, que foram os preditores Ingênuo (Seções 5.2 e 6.2), *Naïve Bayes* (NB—Seções 5.3 e 6.3), *Support Vector Machine* (SVM—Seções 5.4 e 6.4),e baseado em Rede Neural (NN—Seções 5.5 e 6.5).

Nesta seção, apresentaremos uma avaliação interpretativa destes resultados.

Os resultados médios obtidos pelos preditores nas métricas de avaliação da Seção 6.1 são mostrados na Tabela 27, onde os melhores valores alcançados para cada métrica estão destacados.

Conforme podemos verificar na tabela, não houve hegemonia de um método de *Machine Learning*, sendo que todos os três métodos obtiveram o melhor resultado

	pred	FPR	FNR	TNR	NPV	FDR	Recall	Precision	Accuracy
1	Ingênuo	0,79	0,78	0,21	0,21	0,79	0,22	0,21	0,21
2	*Naïve Bayes*	0,78	0,07	0,22	0,72	0,43	0,93	0,57	0,57
3	*Support Vector Machine*	0,26	0,13	0,74	0,85	0,23	0,87	0,77	0,81
4	Rede Neural	0,24	0,16	0,76	0,82	0,22	0,84	0,78	0,80

Tabela 27: Resultados médios das métricas dos preditores

	1	0
1	984	954
0	25	89

Tabela 28: Matriz de confusão do preditor *Naïve Bayes* para o *fold* (4) com a maior revocação.

em ao menos uma das métricas de avaliação—a Rede Neural foi melhor em quatro métricas (precisão, FDR, TNR, e FPR); a SVM foi melhor em três métricas (acurácia, NPV, e FNR); e o NB foi melhor na revocação. Vale destacar ainda, que a Rede Neural e a SVM exibiram resultados bastante similares.

O NB exibiu altos valores de revocação em quatro dos seus cinco *folds* (ver Tabela 18), e atingiu um valor médio de 93%, contudo, acompanhado de um FPR médio de 78%. Analisamos este comportamento por meio do resultado do seu *fold* (4), que obteve a maior revocação (98%), cuja matriz de confusão é mostrada na Tabela 28. Conforme mostrado na matriz, a saída do preditor foi positiva para 1938 das 2052 predições (94,44%), o que aumenta a chance de se conseguir verdadeiros positivos, contudo, acompanhados também de falsos positivos—as predições positivas exibiram $tp = 984$ e $fp = 954$.

Na Figura 22, apresentamos um *gráfico de radar* com os resultados dos preditores. Neste gráfico, os valores das métricas são mostrados em coordenadas polares. As métricas foram organizadas de forma que as quatro métricas com sentido de melhoria maior melhor (⇧) são mostradas na parte superior do gráfico, e as três métricas com sentido de melhoria menor melhor (⇩) são mostradas na parte inferior. Desta forma, o gráfico do preditor perfeito ($fp = 0$ e $fn = 0$) ocuparia todo o semi-círculo superior.

Conforme mostrado no gráfico, a inconsistência no desempenho do *Naïve Bayes* ficou evidenciada, assim como a similaridade dos desempenhos da Rede Neural e

Figura 22: Radar plot dos resultados médios das métricas dos preditores

da *Support Vector Machine*, que exibiram um formato condizente com a ocupação do semi-círculo superior. Estes dois métodos são bastante poderosos e adequados para modelar superfícies de separação não-lineares.

Examinando os gráficos de dispersão das *features* do *dataset* final (Figura 10), podemos notar uma aparente separação linear entre alguns pares de *features* (em especial envolvendo `dctf` e `pis_cfs`). Porém, ampliando a visualização do gráfico, é revelada a sobreposição de boa parcela dos pontos, indicando uma separação não-linear nos planos dos pares de *features*. Além disso, a inferência acerca da forma da superfície de separação dos dados deve considerar o espaço dimensional da totalidade das *features*.

Para podermos ter uma melhor noção de como a complexidade da superfície aumenta com o acréscimo de dimensões, criamos um gráfico da superfície formada pelas *features* `dctf`, `pis_cfs` e `rbc`, que é mostrado na Figura 23. Muito embora o gráfico não mostre a que classe pertence cada região da superfície, podemos verificar

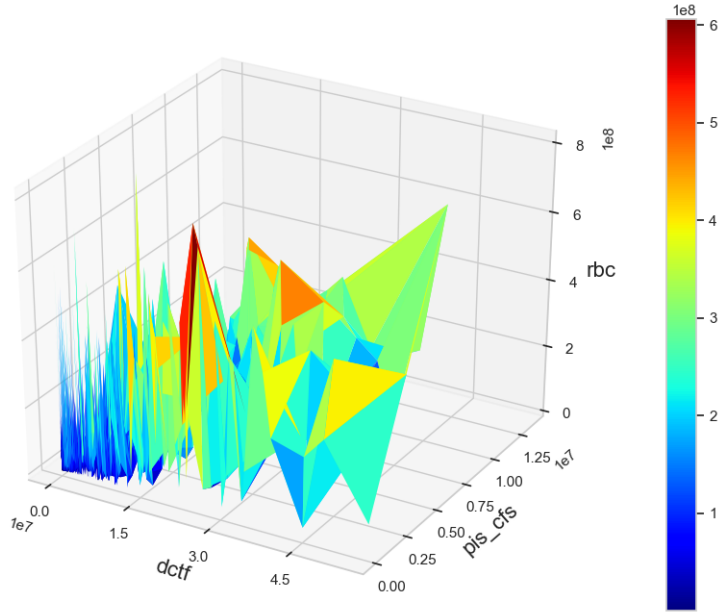


Figura 23: Superfície formada pelas *features* `dctf`, `pis_cfs` e `rbc`.

claramente o aumento da complexidade do seu formato se comparado com os gráficos bidimensionais da Figura 10.

8 Conclusão e Trabalhos Futuros

Neste trabalho, estudamos o problema de predição de atraso na entrega de declarações com dados da Declaração de Débitos e Créditos Tributários Federais (DCTF), que é uma das principais declarações administradas pela Receita Federal do Brasil.

Para isso, empregamos três modelos de *Machine Learning*, que foram o *Naïve Bayes*, a *Support Vector Machine*, e a Rede Neural, para realizar a predição da ocorrência de Multa por Atraso na Entrega da Declaração (MAED) da DCTF, que foi utilizada como um *proxy* do atraso na entrega.

Nossos resultados iniciais alcançaram acurácia de 81%, precisão de 77%, e revocação de 87%, e demonstraram que é possível atacar este problema por meio de modelos de *Machine Learning*.

Nossos trabalhos futuros incluem:

- aumentar o espaço de *features* por meio de novas variáveis com capacidade preditiva no problema;

- estudar outras abordagens para lidar com *datasets* severamente desbalanceados;
- investigar hiperparâmetros e heurísticas da rede neural, tais como funções de perda mais adequadas ao problema, e emprego de elitismo e outros métodos de melhoria da convergência do treinamento;
- investigar *kernels* da SVM que sejam mais adequadas ao problema; e
- pesquisar estratégias para a melhoria da precisão dos modelos, que é uma métrica importante no contexto de respostas do tipo dissuasão pela detecção no modelo Pirâmide de Conformidade (Seção 1.2). Nesse contexto, deve-se evitar alertar um contribuinte que já vai cumprir com as suas obrigações.

Declaração de Isenção de Responsabilidade

Este é um trabalho com propósito acadêmico, e as opiniões aqui expressas, os resultados, as interpretações, bem como as conclusões, são do próprio autor e não refletem, necessariamente, posições da Receita Federal do Brasil.

9 *Links*

Abaixo estão relacionados os endereços eletrônicos para o material complementar a este trabalho.

- Vídeo: `https://youtu.be/kqFpYeGGwDo`
- Repositório: `https://github.com/fdfreitas/tcc_pos_bigdata_pucmg`

REFERÊNCIAS

ALLINGHAM, M. G.; SANDMO, A. Income tax evasion: a theoretical analysis. *Journal of Public Economics*, v. 3–4, n. 3, p. 323–338, nov 1972.

APACHE. *Apache Hadoop*. 2022. Disponível em: <https://hadoop.apache.org/>.

BRASIL. *Código Tributário Nacional — Lei nº 5.172, de 25 de Outubro de 1966*. out. 1966. Disponível em: <http://www.planalto.gov.br/ccivil_03/leis/l5172compilado.htm>.

BRASIL. *Constituição da República Federativa do Brasil : texto constitucional promulgado em 5 de outubro de 1988, compilado até a Emenda Constitucional no 116/2022*. Brasília, DF: Senado Federal, Coordenação de Edições Técnicas, 2022. 435 p. Disponível em: <https://www2.senado.leg.br/bdsf/handle/id/596093>.

BROWNLEE, J. *A Gentle Introduction to Imbalanced Classification*. 2022. Disponível em: <https://machinelearningmastery.com/what-is-imbalanced-classification/>.

CASTRO, A. A. Declaração e confissão de dívida tributária — realizadas pelo sujeito passivo nos tributos submetidos à sistemática de lançamento por homologação. *Tributação em Revista*, Sindicato dos Aauditores-Fiscais da Receita Federal, v. 4, n. 16, abr–jun 1996. Disponível em: <http://www.aldemario.adv.br/artigo3.pdf>.

FIGUEIREDO, G. H. de B. *Um novo paradigma na auditoria em meio digital*. 2010. Disponível em: <http://repositorio.enap.gov.br/handle/1/4580>.

FREITAS, F. D. de. *Modelo de Seleção de Carteiras Baseado em Erros de Predição*. 1. ed. São Paulo: Editora Edgard Blücher Ltda, 2010. ISBN 9788561209933. Disponível em: <http://www.blucher.com.br/livro.asp?Codlivro=09933>.

GODOY, D. *A Gentle Introduction to Imbalanced Classification*. 2018. Disponível em: <https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a>.

GOUVÊA, M. T. Aplicabilidade do modelo pirâmide de conformidade fiscal à cobrança dos tributos no brasil: Um estudo de caso na rfb/rs. *Revista Científica do Sindreceita*, v. 2020, n. 01, p. 3–26, 2020.

HASTIE, T.; TIBSHIRANI, R.; FRIEDMAN, J. *The elements of statistical learning: data mining, inference and prediction*. 2. ed. Springer, 2009. Disponível em: <http://www-stat.stanford.edu/~tibs/ElemStatLearn/>.

HAYKIN, S. *Neural Networks: A Comprehensive Foundation*. 2. ed. [S.l.]: Prentice-Hall, Inc., 1999.

HUE. *Hue Open-source software*. 2022. Disponível em: <https://gethue.com/>.

JAMES, G. et al. *An Introduction to Statistical Learning: With Applications in R*. 2. ed. [S.l.]: Springer, 2021.

JUPYTER. *Project Jupyter*. 2022. Disponível em: <https://jupyter.org/>.

KESAVULU, P. et al. The effect of multicollinearity in nonlinear regression models. *International Journal of Applied Research*, v. 2, n. 12, p. 506–509, 2016. Disponível em: <https://www.allresearchjournal.com/archives/2016/vol2issue12/PartH/2-12-105-626.pdf>.

KINGMA, D. P.; BA, J. *Adam: A Method for Stochastic Optimization*. arXiv, 2014. Disponível em: <https://arxiv.org/abs/1412.6980>.

LORENA, A. C.; CARVALHO, A. C. P. L. F. de. *Introdução às Máquinas de Vetores Suporte*. 2003. Disponível em: <https://repositorio.usp.br/directbitstream/a7ed198b-f6a3-4cec-b132-7e113bd51424/BIBLIOTECA_113_RT_192.pdf>.

MATOS, T. O. de. *Interação entre órgãos fiscais na redução do "tax gap": regime de trocas e aproveitamentos, eficiência tributária e combate à ilicitude fiscal na experiência do Estado de São Paulo*. São Paulo: Dialética, 2021. 188 p.

MAZZA, A. *Manual de direito tributário*. São Paulo: Saraiva, 2018.

OCDE. *Compliance Risk Management: Managing and Improving Tax Compliance*. 2004. Disponível em: <https://www.oecd.org/tax/administration/33818656.pdf>.

PEDROSA, L. S.; MOURA, F. R. de. Eficiência na arrecadação de icms dos estados brasileiros com base no pib estadual: uma análise do gap tributário e da hipótese do icms em relação ao pib como fato estilizado (2002-2017). *Revista Debate Econômico*, v. 7, n. 1, jan-jun 2019.

RFB. *Aspectos gerais sobre o sigilo fiscal*. 2022. Disponível em: <https://www.gov.br/receitafederal/pt-br/assuntos/orientacao-tributaria/sigilo-fiscal/aspectos-gerais-sobre-o-sigilo-fiscal>.

RFB. *DCTF (Declaração de débitos e créditos tributários federais)*. 2022. Disponível em: <https://www.gov.br/receitafederal/pt-br/acesso-a-informacao/legislacao/legislacao-por-assunto/dctf>.

RFB. *Declarar débitos e créditos tributários federais (DCTF)*. 2022. Disponível em: <https://www.gov.br/pt-br/servicos/declarar-debitos-e-creditos-tributarios-federais>.

RFB. *Grandes Contribuintes*. 2022. Disponível em: <https://www.gov.br/receitafederal/pt-br/servicos/cadastro/maco>.

RFB. *Secretaria Especial da Receita Federal do Brasil — Institucional*. 2022. Disponível em: <https://www.gov.br/receitafederal/pt-br/acesso-a-informacao/institucional>.

SCIKIT-LEARN. *Naive Bayes*. 2022. Disponível em: <https://scikit-learn.org/stable/modules/naive_bayes.html>.

SIQUEIRA, M. L.; RAMOS, F. S. A economia da sonegação – teorias e evidências empíricas. *Revista de Economia Contemporânea*, v. 9, n. 3, p. 555–581, set./dez. 2005.

SPED. *Sistema Público de Escrituração Digital (Sped)*. 2022. Disponível em: <http://sped.rfb.gov.br/>.

APÊNDICE

I - Código Fonte do Peditor Baseado em Rede Neural

```
 # Importa modulos PyTorch
from numpy import vstack
from pandas import read_csv
from sklearn.metrics import accuracy_score
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torch.utils.data import random_split
from torch import Tensor
from torch.nn import Linear
from torch.nn import ReLU
from torch.nn import Sigmoid
from torch.nn import Module
from torch.optim import SGD
from torch.nn import BCELoss, BCEWithLogitsLoss
from torch.nn.init import kaiming_uniform_
from torch.nn.init import xavier_uniform_

 # Neural Network class
'''
Network topology definition dictionary in the form:

topology = Dict()
topology.name         = f'Topology name used as abse filename for saving model
    parameters'
topology.layers       = 'inp : 64         : 32         : 2             :1'
topology.activations  = 'inp : relu       : relu       : sigmoid       : sigmoid'
topology.initializations = 'inp : kaiming_relu : kaiming_relu : kaiming_sigmoid : xavier'

'''
 # model definition
class MLP(Module):
    # define model elements
    def __init__(self, topology):
        super(MLP, self).__init__()
        torch.manual_seed(0)
        self.layers = []
        self.activations = []
        # parse topology and activations
        print('Creating neural network:')
        layers        = topology.layers.replace(' ', '').split(':')[1:]
        activations   = topology.activations.replace(' ', '').split(':')[1:]
        initializations = topology.initializations.replace(' ', '').split(':')[1:] if
            'initializations' in topology else None
        n_input = topology.input_size
        for i in range(len(layers)):
            # layers
            n_output = int(layers[i])
            print(f'-layer({i+1}): Linear({n_input:3d}, {n_output:3d})', end='\t')
            layer = torch.nn.Linear(n_input, n_output)
```

```
            setattr(self, f'layer{i}', layer) # necessary that the layer be a member of
                the class
            self.layers.append(layer)

            # activations
            activation = activations[i]
            if activation == 'relu':
                print(f'activation {activation}', end='\t')
                self.activations.append(torch.relu)
            elif activation == 'sigmoid':
                print(f'activation {activation}', end='\t')
                self.activations.append(torch.sigmoid)

            # initializations
            if initializations is not None:
                initialization = initializations[i]
                if initialization == '':
                    print(f'no weights initialization', end='\t')
                if initialization.startswith('kaiming'):
                    nl = initialization.split('_')
                    if len(nl) == 1:
                        nonlinearity = 'relu'
                    if len(nl) == 2:
                        nonlinearity = initialization.split('_')[1]
                        torch.nn.init.kaiming_uniform_(getattr(self, f'layer{i}').weight,
                            nonlinearity=nonlinearity)
                    else:
                        nonlinearity = None

                    if nonlinearity is None:
                        print(f'**ERROR*: {initialization} not in format
                            kaiming_nonlinearity', end='\t')
                    else:
                        torch.nn.init.kaiming_uniform_(getattr(self, f'layer{i}').weight,
                            nonlinearity=nonlinearity)
                        print(f'weights initialization kaiming_uniform_ {nonlinearity} ',
                            end='\t')

                elif initialization.startswith('xavier'):
                    torch.nn.init.xavier_uniform_(getattr(self, f'layer{i}').weight)
                    print(f'weights initialization xavier_uniform_', end='\t')
            print('')
            # next n_input
            n_input = n_output
        print('')

    # forward propagate input
    def forward(self, X):
        for layer, activation in zip(self.layers, self.activations):
            X = activation(layer(X))
        return X

    # reset all weights
    def reset_weights(self):
        #print(f'Reset all network trainable parameters')
        for layer in self.children():
            if hasattr(layer, 'reset_parameters'):
                #print(f'Reset trainable parameters of layer = {layer}')
```

```
                layer.reset_parameters()

 # dataset definition
class DataSet(Dataset):
    # load the dataset
    def __init__(self, df):
        # store the inputs and outputs
        self.X = df.values[:, :-1]
        self.y = df.values[:, -1]
        # ensure input data is floats
        self.X = self.X.astype('float32')
        self.y = self.y.astype('float32')
        self.y = self.y.reshape((len(self.y), 1))

    # number of rows in the dataset
    def __len__(self):
        return len(self.X)

    # get a row at an index
    def __getitem__(self, idx):
        return [self.X[idx], self.y[idx]]

    # get indexes for train and test rows
    def get_splits(self, n_test=0.20):
        # determine sizes
        test_size = round(n_test * len(self.X))
        train_size = len(self.X) - test_size
        # calculate the split
        return random_split(self, [train_size, test_size])
```

II - Código Fonte dos Artefatos Auxiliares e de Uso Geral

Arquivo `COMMON/tcc_common.py`

```
#!/usr/bin/env python
# coding: utf-8

# Trabalho de Conclusao de Curso
# Pos-graduacao em Ciencia de Dados e Big Data (2020) - PUC Minas
# Fabio Daros de Freitas

#
#  Common use artifacts and Jupyter Notebooks stuff
#

import builtins as __builtin__
import time
from datetime import datetime, timedelta
from IPython.display import Markdown, display, HTML
display(HTML("<style>.container { width:100
import pandas as pd
pd.options.display.float_format = '{:,.2f}'.format
print(f'Pandas version......: {pd.__version__}')

import numpy as np
print(f'Numpy version.......: {np.__version__}')

import sklearn as sk
from sklearn import preprocessing
from sklearn.preprocessing import LabelEncoder, MinMaxScaler, StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn.model_selection import KFold, LeaveOneOut
print(f'Scikit-learn version: {sk.__version__}')

import seaborn as sns
print(f'Seaborn version.....: {sns.__version__}')
sns.set(style="white")
sns.set(style="whitegrid", color_codes=True)

import statsmodels as sm
from statsmodels.stats.outliers_influence import variance_inflation_factor
import statsmodels.api as sm_api
print(f'Stats models version: {sm.__version__}')

import matplotlib as mpl
print(f'Matplotlib version..: {mpl.__version__}')
import matplotlib.pyplot as plt
plt.rc("font", size=14)

#-------------------------------------------------------------------------
# Classes and general functions
```

```
52  #--------------------------------------------------------------------------------
53
54  class Dict(dict):
55      __getattr__= dict.__getitem__
56      __setattr__= dict.__setitem__
57      __delattr__= dict.__delitem__
58
59  class ETime():
60      def __init__(self):
61          self.start = datetime.now()
62      def check(self):
63          e = datetime.now() - self.start
64          print(f'Elapsed time: {e}')
65
66  def print(*args, **kwargs):
67      '''
68      print() override to treat behavorial kw
69      '''
70      verbose = kwargs.pop('verbose', True)
71      if not verbose:
72          return
73      ret = __builtin__.print(*args, **kwargs)
74      return ret
75
76  def print_md(string, verbose=True):
77      if verbose:
78          display(Markdown(string))
79
80  #--------------------------------------------------------------------------------
81  # Pandas dataframe functions
82  #--------------------------------------------------------------------------------
83
84  def df_display(df, name=None, rows=2):
85      if name is not None:
86          print(f'{name}:')
87      max_rows = pd.options.display.max_rows
88      print(df.shape)
89      if hasattr(df, 'columns'):
90          print(df.columns)
91      print(df.dtypes)
92      pd.options.display.max_rows = rows
93      display(df)
94      pd.options.display.max_rows = max_rows
95
96  # Print df as LaTeX table
97  def df_latex(df, dec=2, **kwargs):
98      def num(x):
99          x = round(x, dec)
100         return f'\\num{{{x}}}'
101
102     s  = '\\begin{table}[TABPOS]\n'
103     s += '\\centering\n'
104     s += '\\small\n'
105     s += df.to_latex(float_format=num, escape=False, **kwargs)
106     s += '\\caption{}\n'
107     s += '\\label{}\n'
108     s += '\\end{table}\n'
109     s = s.replace('_', '\\_')
```

```
    s = s.replace('\\\\', '\\\\%')
    print(s)


#
# dtype functions
#
def df_metric_dtypes():
    #np.number
    return ['float16', 'float32', 'float64']

def df_category_dtypes():
    return ['int32', 'int64']

#
# Counting functions
#
def df_count_rows_duplicated(df):
    if 'cnpj8' in df.columns and 'ano_mes' in df.columns:
        return df.duplicated(['cnpj8','ano_mes']).sum()
    return df.duplicated().sum()

def df_count_rows_nan(df):
    return df.shape[0] - df.dropna().shape[0]

def df_count_rows_metric_zero(df):
    zero = 0
    a = df.select_dtypes(include=df_metric_dtypes())
    if not a.empty:
        b = df[a.eq(0).any(axis=1)]
        if not b.empty:
            zero = b.shape[0]
    return zero
def df_count_rows_metric_negative(df):
    neg = 0
    a = df.select_dtypes(include=df_metric_dtypes())
    if not a.empty:
        b = df[a.lt(0).any(axis=1)]
        if not b.empty:
            neg = b.shape[0]
    return neg

def df_count_columns_binary(df, column):
    s =0
    n = 0
    if column in df:
        s = len(df[df[column] == 1])
        n = len(df[df[column] == 0])
    return s, n

#
# Rows functions
#

#
# Columns functions
#

```

```
168 def df_metric_dtypes():
169     #np.number
170     return ['float16', 'float32', 'float64']
171 
172 def df_category_dtypes():
173     return ['int32', 'int64']
174 
175 def df_columns_metric(df):
176     a = df.select_dtypes(include=df_metric_dtypes())
177     return a.columns
178 
179 def df_columns_metric_zero(df):
180     cols = []
181     a = df.select_dtypes(include=df_metric_dtypes())
182     for c in a.columns:
183         n = a[c].eq(0).sum(axis=0)
184         if n > 0:
185             cols.append(f'{c}({n}) ')
186     return cols
187 
188 #
189 # Profile functions
190 #
191 def df_profile(df, name=''):
192     print_md(f'**Sumario do DataFrame:** {name}')
193     l      = df.shape[0]
194     c      = df.shape[1]
195     ldup   = df_count_rows_duplicated(df)
196     lnan   = df_count_rows_nan(df)
197     cnpj8 = len(df['cnpj8'].unique()) if 'cnpj8' in df else 0
198     periodo= f'{df["ano_mes"].min()} a {df["ano_mes"].max()}' if 'ano_mes' in df else
            '-'
199     neg   = df_count_rows_metric_negative(df)
200     zero = df_count_rows_metric_zero(df)
201     zcols = df_columns_metric_zero(df)
202     maed_s, maed_n = df_count_columns_binary(df, 'maed_dctf_prox_mes')
203 
204     print(f'-Linhas({l}): duplicadas={ldup} nulas={lnan} metricas zero={zero} metricas
            negativo={neg}')
205     print(f'-Colunas({c}): colunas com alguma metrica zero={"".join(zcols)}')
206     print(f'-cnpj8 unicos={cnpj8} periodo={periodo}')
207     tt = maed_s + maed_n
208     if tt == 0:
209         s = '-'
210         n = '-'
211         r = '-'
212     else:
213         s = f'{maed_s} ({100*maed_s/tt:.2f}    n = f'{maed_n} ({100*maed_n/tt:.2f}    r
                = f'1:{int(1/(maed_s/maed_n))}' if maed_n > 0 else f'1:-'
214     print(f'-maed_dctf_prox_mes: Sim[1]={s} Nao[0]={n} ratio(S:N)={r})')
215     df_profile_maed(df)
216     print(df.dtypes)
217     print_md('---')
218 
219 
220 # MAED specific profile
221 def df_profile_maed(df):
222     if not ('maed_dctf' in df and 'maed_dctf_prox_mes' in df):
```

```
223         return
224     f00 = (df["maed_dctf"] == 0) & (df["maed_dctf_prox_mes"] == 0)
225     f01 = (df["maed_dctf"] == 0) & (df["maed_dctf_prox_mes"] == 1)
226     f10 = (df["maed_dctf"] == 1) & (df["maed_dctf_prox_mes"] == 0)
227     f11 = (df["maed_dctf"] == 1) & (df["maed_dctf_prox_mes"] == 1)
228     m00 =  len(df[f00])
229     m01 =  len(df[f01])
230     m10 =  len(df[f10])
231     m11 =  len(df[f11])
232     print('-Profile MAED: m_ij (i==maed_dctf, j=maed_dctf_prox_mes)')
233     print(f'\tm_00={m00:6d}\tm_01={m01:6d}')
234     print(f'\tm_10={m10:6d}\tm_11={m11:6d}')
235     print('')
236     return f00, f01, f10, f11
237
238 def df_profile_maed_plot(df, width=940, height=1024, dpi=100, log=False, fname=None):
239     fig = plt.figure(figsize=(width/dpi, height/dpi), dpi=dpi)
240     #fig.suptitle(f'Ocorrencias maed_dcft', fontsize=16)
241     plt.subplots_adjust(left=0.1, bottom=-0.1, right=0.9, top=0.9, wspace=0.1,
            hspace=0.2)
242     #plt.xticks(fontsize=10) # for xticks
243     #plt.yticks(fontsize=10) # for yticks
244
245     axes = fig.subplots(2, 1, sharex=False, gridspec_kw={'height_ratios': [1, 1]})
246     ax1, ax2 = axes
247
248     ax = ax1
249     if log:
250         ax.set_yscale('log')
251     ax.set_title(f'Ocorrencias maed_dcft', fontsize=16)
252     ax.set_xlabel('maed_dctf')
253     ax.set_ylabel('contagem', fontsize=14)
254     df['maed_dctf'].value_counts().plot(ax=ax, kind='bar', fontsize=14, rot=0) # ,
            color=['blue', 'red'])
255     for p in ax.patches:
256         ax.annotate(str(p.get_height()), (p.get_x() * 1.0001, p.get_height() * 1.0001))
257
258     ax = ax2
259     if log:
260         ax.set_yscale('log')
261     ax.set_title(f'Ocorrencias maed_dctf_prox_mes', fontsize=16)
262     ax.set_xlabel('maed_dctf_prox_mes')
263     ax.set_ylabel('contagem', fontsize=14)
264     df['maed_dctf_prox_mes'].value_counts().plot(ax=ax, kind='bar', fontsize=14, rot=0)
            # , color=['blue', 'red'])
265     for p in ax.patches:
266         ax.annotate(str(p.get_height()), (p.get_x() * 1.0001, p.get_height() * 1.0001))
267     if fname is not None:
268         plt.savefig(fname, dpi=dpi, bbox_inches='tight')
269
270 #
271 # Changing and augmentation functions
272 #
273
274 def df_clean(df, name='', drop_metrica_negativo=True, drop_metrica_zero=False,
        fill_zeros=False):
275     '''Remove negativos e zerados apenas das colunas das metricas float'''
276     print(f'df_clean: {name} total linhas={df.shape[0]}')
```

```
277
278     n = df_count_rows_duplicated(df)
279     if n > 0:
280         if 'cnpj8' in df.columns and 'ano_mes' in df.columns:
281             print(f'Removendo linhas duplicadas [cnpj8, ano_mes]({n})...')
282             df.drop_duplicates(['cnpj8','ano_mes'],keep= 'last', inplace=True)
283         else:
284             print(f'Removendo linhas duplicadas({n})...')
285             df.drop_duplicates(inplace=True)
286
287     n = df_count_rows_nan(df)
288     if n > 0:
289         print(f'Removendo linhas nulas({n})...')
290         df.dropna(inplace=True)
291
292     n = df_count_rows_metric_negative(df)
293     if n > 0 and drop_metrica_negativo:
294         print(f'Removendo linhas com metricas com valores negativos({n})...')
295         a = df.select_dtypes(include=df_metric_dtypes())
296         if not a.empty:
297             b = a.lt(0).any(axis=1)
298             if not b.empty:
299                 df.drop(df[b].index, inplace=True)
300
301     n = df_count_rows_metric_zero(df)
302     if n > 0 and drop_metrica_zero:
303         print(f'Removendo linhas com metricas com valores zero({n})...')
304         a = df.select_dtypes(include=df_metric_dtypes())
305         if not a.empty:
306             b = a.eq(0).any(axis=1)
307             if not b.empty:
308                 df = df.drop(df[b].index, inplace=True)
309
310     n = df_count_rows_metric_zero(df)
311     if n > 0 and fill_zeros:
312         print(f'Preenchendo metricas zero({n})...')
313         a = df.select_dtypes(include=df_metric_dtypes())
314         cols = a.columns
315         df[cols] = df[cols].replace(0, df[cols].mean(skipna=True, axis=0))
316
317     print(f'total linhas final={df.shape[0]}')
318     return df
319
320 def df_augment(df, remove_colinear=True):
321     '''Augment dataframe columns with:
322     - rbc : receita bruta calculada
323     - ct  : carga tributaria
324     - rec : receitas
325     - desp: despesas
326     - maed_dctf_total : quantidade de vezes que o contribuinte recebeu MAED DCTF
327
328     Remove colinear columns:
329     deb, vndas, crd, cpras
330
331     '''
332     df = df.copy()
333     df['rbc'] = df.apply(df_column_apply_rbc, axis=1)
334     df['ct'] = df.apply(df_column_apply_ct, axis=1)
```

```
335     #df['rec'] = df.deb + df.vndas
336     #df['desp'] = df.crd + df.cpras
337 
338     df = df.sort_values(['cnpj8', 'ano_mes'])
339     df['maed_dctf_total'] = (df['maed_dctf']).groupby(df['cnpj8']).cumsum()
340     #df = df.astype({'maed_dctf_total': np.float64})
341 
342     # Final dataset columns: must have target at last column
343     cols = ['cnpj8',
344            'ano_mes',
345            'dctf',
346            'pis_cfs',
347            'cprb',
348            'crd',
349            'deb',
350            'cpras',
351            'vndas',
352            'rbc',
353            'ct',
354            'maed_dctf',
355            'maed_dctf_total',
356            'maed_dctf_prox_mes']
357 
358     if remove_colinear:
359         pass
360 
361     df = df[cols]
362     df = df_clean(df) # protect nan
363     # debug
364     df_display(df)
365     df.describe()
366     #df.maed_dctf_prox_mes.describe()
367     return df
368 
369 # Columns changing and augmentation functions
370 
371 def df_column_create_ano_mes(df):
372     df['ano_mes'] = df['ano'] + '-' + df['mes']
373     df.drop(['ano', 'mes'], axis=1, inplace=True)
374 
375 # apply func para criacao da coluna maed_dctf
376 def df_column_apply_maed_dctf(maed_dict, row):
377     cnpj8 = row.cnpj8
378     ano_mes = row.ano_mes
379     key = (cnpj8, ano_mes)
380     return 1 if key in maed_dict else 0
381 
382 # apply func para criacao da coluna maed_dctf_prox_mes
383 def df_maed_dict(df):
384     df = df.set_index(['cnpj8', 'ano_mes'], inplace=False)
385     return df.T.to_dict()
386 
387 def df_column_apply_maed_dctf_prox_mes(maed_dict, row):
388     cnpj8 = row.cnpj8
389     ano_mes = ano_mes_next(row.ano_mes)
390     key = (cnpj8, ano_mes)
391     return 1 if key in maed_dict else 0
392 
```

```
393 def df_column_apply_rbc(row):
394     '''Maior valor entre EFD Reeceita Bruta e NFe Vendas, quando ambos > 0, ou valor
            Efinancera Debitos'''
395     v = max(row.rb, row.vndas) if row.rb > 0.0 or row.vndas > 0.0 else row.deb
396     return v if v >= 0 else np.nan
397 
398 def df_column_apply_ct(row):
399     '''ct = dctf / rbc'''
400     v = row.dctf / row.rbc if row.rbc > 0 else 0
401     return v
402 
403 # Auxiliary functions
404 
405 # returns next ano_mes from current ano_mes
406 def ano_mes_next(am):
407     aml = am.split('-')
408     a = int(aml[0])
409     m = int(aml[1])
410     a = a+1 if m == 12 else a
411     m = 1 if m == 12 else m+1
412     return(f'{a}-{m:02d}')
413 
414 # returns previous ano_mes from current ano_mes
415 def ano_mes_previous(am):
416     aml = am.split('-')
417     a = int(aml[0])
418     m = int(aml[1])
419     a = a-1 if m == 1 else a
420     m = 12 if m == 1 else m-1
421     return(f'{a}-{m:02d}')
422 
423 #------------------------------------------------------------------------------
424 # Training support functions
425 #------------------------------------------------------------------------------
426 
427 # Get inputs and target from dataframe
428 def dataset_input_and_target(df):
429     inp = df.select_dtypes(include=df_category_dtypes() +
            df_metric_dtypes()).drop('maed_dctf_prox_mes', axis=1).copy()
430     tgt = df['maed_dctf_prox_mes'].copy()
431     return inp, tgt
432 
433 # Model cross validation
434 def model_cross_validation(model, df, K, fold_data, train_model, **kwargs):
435     '''
436     Client application must implement the following functions:
437 
438     - fold_data(df, trn_ids, tst_ids)
439     receives current fold training and testing indexes of df dataframe,
440     and returns trn_data and tst_data (in suitable format) to be used by train_model()
441 
442     - train_model(model, trn_data, tst_data, fold=fold, **kwargs)
443     receives model current fold trn_data and tst_data, and returns
444     trn_mts and tst_mts fold metrics PredResult objects for aggregating
445     '''
446 
447     cv_type = f'{K}-fold' if K > 0 else 'leave-one-out'
448     print_md(f'**Model {cv_type} Cross Validation:**')
```

```
449
450     metrics = []
451
452     # Define the K-fold Cross Validator
453     if K == 0:
454         folds = LeaveOneOut()
455     else:
456         folds = KFold(n_splits=K, shuffle=True, random_state=10)
457
458     # Cross Validation model evaluation
459     trn_metrics = PredResult(name=f'Folds trn', index_name='fold')
460     tst_metrics = PredResult(name=f'Folds tst', index_name='fold')
461     for fold, (trn_ids, tst_ids) in enumerate(folds.split(df)):
462
463         print_md('\n---\n')
464         print_md(f'**Fold {fold+1}**')
465         rows_df, rows_trn, rows_tst = df.shape[0], len(trn_ids), len(tst_ids)
466         print(f'trn_ids={rows_trn} ({100*rows_trn/rows_df:.2f}
              print(f'tst_ids={rows_tst} ({100*rows_tst/rows_df:.2f}
467         trn_data, tst_data = fold_data(df, trn_ids, tst_ids, fold=fold, **kwargs)
468
469         trn_mts, tst_mts = train_model(model, trn_data, tst_data, fold=fold, **kwargs)
470
471         print(f'Fold {fold+1} performance and last test set prediction result:')
472         trn_mts.display_metrics()
473         tst_mts.prediction_summary()
474
475         # aggreagte fold result
476         trn_metrics.latex = trn_mts.latex
477         trn_metrics.append(trn_mts.get(row_id=fold+1))
478
479         tst_metrics.latex = tst_mts.latex
480         tst_metrics.append(tst_mts.get(row_id=fold+1))
481
482     print_md('\n---\n')
483     print_md(f'**Folds overall performance:**')
484     trn_metrics.describe_metrics()
485     #display(trn_metrics.get_history())
486     trn_metrics.display_metrics_history()
487     tst_metrics.describe_metrics()
488     #display(tst_metrics.get_history())
489     tst_metrics.display_metrics_history()
490
491 #------------------------------------------------------------------------------
492 # Predictions evalaution classes and functions
493 #------------------------------------------------------------------------------
494
495 def pred_discretize(y):
496     y[y > 0.5] = 1
497     y[y <= 0.5] = 0
498     return y
499
500 class PredResult():
501
502     metrics_names = { 'FPR'     : 'False Positive Rate (Type I error) - FPR',
503                       'FNR'     : 'False Negative Rate (Type II error)- FNR',
504                       'TNR'     : 'True Negative Rate (Specificity): TNR',
505                       'NPV'     : 'Negative Predictive Value - NPV',
```

```
                    'FDR'      : 'False Discovery Rate - FDR',
                    'Recall'   : 'True Positive Rate (Recall/Sensibility) - TPR/Recall',
                    'Precision' : 'Positive Predictive Value (Precision) -
                        PPV/Precision',
                    'Accuracy' : 'Accuracy'}




    def __init__(self, name='', index_name='', latex=False):

        self.name    = name
        self.pp      = f'{self.name} - ' if self.name != '' else '' # print prtefix
        self.latex   = latex # Output describe and history also a LaTeX version

        # initialize metrics dataframe
        c = Dict((m, float()) for m in self.metrics_names.keys())
        self.df_metrics = pd.DataFrame(c, index=[])
        if index_name != '':
            self.df_metrics.index.name = index_name
        self.y_true = None
        self.y_pred = None

    def update(self, y_true, y_pred, row_id=None):

        if y_true is None or y_pred is None:
            return

        self.y_true = np.array(y_true)
        self.y_pred = np.array(y_pred)

        # metrics calculation

        # Confusion matrix: (C_ij: observations in group i predicted as group j)
        cm = confusion_matrix(self.y_true, self.y_pred)
        cm_res = cm.ravel()

        # not binary classification result...
        if len(cm_res) != 4:
            return
        tn, fp, fn, tp = cm_res
        row = dict() # Pandas dislike Dict()...

        def div(a, b):
            return a/b if b != 0 else 0

        row['FPR']     = div( fp,  fp + tn )
        row['FNR']     = div( fn,  tp + fn )
        row['TNR']     = div( tn,  tn + fp )
        row['NPV']     = div( tn,  tn + fn )
        row['FDR']     = div( fp,  tp + fp )
        row['Recall']  = div( tp,  tp + fn )
        row['Precision'] = div( tp, tp + fp )
        row['Accuracy'] = div( tp + tn, tp + fp + fn + tn )


        if row_id is None:
            self.df_metrics = self.df_metrics.append(row, ignore_index=True)
```

```
563         else:
564             ser = pd.Series(row)
565             ser.name = row_id
566             self.df_metrics = self.df_metrics.append(ser, ignore_index=False)
567 
568     def append(self, df):
569         self.df_metrics = self.df_metrics.append(df, ignore_index=False)
570 
571     def get(self, func=np.mean, row_id=None):
572         ser = self.df_metrics.apply(func, axis=0)
573         if row_id is not None:
574            ser.name = row_id
575         return ser
576 
577     def get_history(self, iloc=None):
578         if iloc == None:
579             return self.df_metrics if len(self.df_metrics) > 0 else None
580 
581         return self.df_metrics.iloc[iloc] if len(self.df_metrics) > 0 else None
582 
583     def display_metrics_history(self):
584         print(f'{self.pp}Metrics history:')
585         display(self.df_metrics)
586         if self.latex:
587             df_latex(self.df_metrics)
588 
589 
590     def display_metrics(self, func=np.mean, T=True):
591 
592         ag = {np.mean : 'avg',
593               np.sum : 'sum',
594               np.min : 'min',
595               np.max : 'max'}
596 
597         print_md(f'**{self.pp}Metrics values ({ag.get(func, "(agg. function not
                found)")}) values:**')
598         df = pd.DataFrame(self.get(func), columns = ['value'])
599         if T:
600             display(df.T)
601         else:
602             display(df)
603 
604     def describe_metrics(self):
605         print(f'{self.pp}Metrics summary:')
606         display(self.df_metrics.describe())
607         if self.latex:
608             df_latex(self.df_metrics.describe())
609 
610 
611     def explain_metrics(self):
612         print(f'{self.pp}Metrics descriptions:')
613         for m in self.metrics_names.items():
614             print(f'{m[0]:30s} : {m[1]}')
615 
616     def prediction_summary(self):
617 
618         self.display_metrics(T=True)
619 
```

```
620         if self.y_true is None or self.y_pred is None:
621             return
622 
623         print_md('**Result of the last updated prediction:**')
624         y = self.y_true
625         s = len(y[y == 1])
626         n = len(y[y == 0])
627         r = s/n if n > 0 else 0
628         t = len(y)
629         print(f'TRUE: Sim[1]={s:6d} Nao[0]={n:6d} ratio(S/N)={r:.2f} Total={t:6d}')
630 
631         y = self.y_pred
632         s = len(y[y == 1])
633         n = len(y[y == 0])
634         r = s/n if n > 0 else 0
635         t = len(y)
636         print(f'PRED: Sim[1]={s:6d} Nao[0]={n:6d} ratio(S/N)={r:.2f} Total={t:6d}')
637 
638         cm = confusion_matrix(self.y_true, self.y_pred)
639 
640         accuracy = cm.diagonal().sum()/cm.sum()
641 
642         print_md(f'Accuracy: {accuracy:.2f}')
643 
644         print_md('**Prediction Summary:**')
645 
646        #By definition a confusion matrix C is such that C_ij is equal to the number of
               observations known to be in group i and predicted to be in group j .
647        #Thus in binary classification, the count of true negatives is C_00, false
               negatives is C_10, true positives is C_11 and false positives is C_01.
648        print('Confusion matrix: (literature format)')
649        tn, fp, fn, tp = cm.ravel()
650        d = { 1 : [fn, tp], 0 : [tn, fp] } # create conf mat. with inverted rows
651        dfcm = pd.DataFrame(d)
652        dfcm = dfcm.iloc[::-1]           # revert rows
653        print(dfcm)
654 
655        if self.latex:
656            df_latex(dfcm)
657 
658        print(classification_report(self.y_true, self.y_pred))
659 # eof
```