P.I – 5º Semestre – Aprendizado de Máquina

Nome: Victor Hugo Ferreira Soares 5º DSM
Nome: Pedro Afonso Acacio da Silva 5º DSM
Nome: Samuel Ribeiro Filho 5º DSM
Nome: Paulo Henrique Borges de Andrade Filho 5º DSM

1. Conjunto de Dados Escolhido

Para este projeto foi selecionado um conjunto de dados bancários contendo informações cadastrais, demográficas e comportamentais de clientes. O objetivo do trabalho é prever, por meio de técnicas de Aprendizado de Máquina, se um cliente possui bom ou mau score, permitindo a uma instituição de consórcios reduzir riscos de crédito e otimizar processos de aprovação.

O dataset inicial contém 16.650 registros e reúne variáveis como sexo, posse de veículo, renda, tipo de residência, idade, tempo de emprego, entre outras. Trata-se de um conjunto representativo para estudos de score de crédito e classificação binária.

2. Dicionário do Conjunto de Dados

Variable Name	Description	Tipo
sexo	M = 'Masculino'; F = 'Feminino'	M/F
posse_de_veiculo	Y = 'possui'; N = 'não possui'	Y/N
posse_de_imovel	Y = 'possui'; N = 'não possui'	Y/N
qtd_filhos	Quantidade de filhos	inteiro
tipo_renda	Tipo de renda (ex: assaliariado, autônomo etc)	texto
educacao	Nível de educação (ex: secundário, superior etc)	texto
estado_civil	Estado civil (ex: solteiro, casado etc)	texto
tipo_residencia	tipo de residência (ex: casa/apartamento, com os pais etc)	texto
idade	idade em anos	inteiro
tempo de emprego	tempo de emprego em anos	inteiro
possui_celular	Indica se possui celular (1 = sim, 0 = não)	binária
possui_fone_comercial	Indica se possui telefone comercial (1 = sim, 0 = não)	binária
possui_fone	Indica se possui telefone (1 = sim, 0 = não)	binária
possui_email	Indica se possui e-mail (1 = sim, 0 = não)	binária
qt_pessoas_residencia	quantidade de pessoas na residência	inteiro
mau	indicadora de mau pagador (True = mau, False = bom)	binária

3. Pré-processamento dos Dados

O pré-processamento foi conduzido para garantir qualidade e coerência antes da etapa de modelagem. A seguir são descritas as etapas realizadas.

3.1. Carregamento e inspeção inicial

- Visualizações iniciais: `df.head()`, `df.info()` e `df.describe()`.
- Observações iniciais:
    - 16.650 linhas e 17 colunas (após remoção de `id`).
    - Muitas colunas no tipo `object` — necessidade de padronização.
    - `idade` e `tempo_emprego` apresentavam formatação incorreta (texto e valores impossíveis).
    - Valores nulos detectados nas colunas: `sexo`, `tipo_renda`, `tipo_residencia`.

3.2. Tratamento de valores ausentes

- Estratégia: imputação por moda (categoria mais frequente) nas colunas categóricas (`sexo`, `tipo_renda`, `tipo_residencia`).
- Justificativa: preserva a distribuição e evita perda de amostras.
- Resultado: nenhuma coluna permaneceu com dados faltantes após imputação.

3.3. Remoção de duplicatas

- Aplicado: `df.drop_duplicates()`.
- Redução observada: 16.650 → 6.770 linhas.
- Interpretação: presença de forte redundância/duplicação no dataset original; remoção necessária para evitar vieses e garantir diversidade.

3.4. Conversão de tipos e padronização

3.4.1. Coluna `idade`

- Problema: valores com pontos e números corrompidos (ex.: `"34.728.767.123.287.600"`).
- Tratamento: remover caracteres indesejados, converter para inteiro, eliminar valores absurdos e validar faixa (18–120 anos).
- Resultado: distribuição coerente com concentração entre 25–50 anos.

3.4.2. Coluna `tempo_emprego`

- Problema: valores inconsistentes (números muito grandes, possíveis unidades incorretas).
- Tratamento: limpeza de caracteres, conversão para numérico bruto, identificação e remoção de outliers, padronização para anos de emprego.
- Resultado: forte assimetria corrigida; valores extremos removidos.

3.5. Análise de inconsistências

- Correções de grafias duplicadas em categorias, padronização de rótulos e remoção de entradas fora do domínio esperado.

- Após revisão, não houve inconsistências adicionais relevantes.

3.6. Normalização e padronização

- Análise de escala (`df.describe()`): variáveis como `qtd_filhos`, `qt_pessoas_residencia` e `tempo_emprego` apresentaram forte assimetria.
- Aplicações: uso de StandardScaler (Z-score) e/ou MinMaxScaler conforme o modelo.
- Justificativa: modelos baseados em distância e lineares beneficiam-se da padronização.

3.7. Identificação e remoção de outliers

- Métodos usados: boxplots, IQR (1.5 × IQR) e inspeção visual.
- `tempo_emprego` teve os outliers mais relevantes — removidos ou transformados.
- Efeito: redução da assimetria, aproximação entre média e mediana e redução de ruído.

3.8. Resultado final do pré-processamento

- Dataset final: trusted_dataset.csv com 6.770 amostras.
- Características: sem duplicatas, sem valores ausentes, categorias padronizadas, variáveis numéricas tratadas e escalonadas, outliers tratados.
- Uso: dataset confiável para EDA e modelagem.

Capítulo 4 — Criação do modelo de classificação

4.1 Introdução

Após o pré-processamento foi desenvolvido um notebook para construção, comparação e exportação de modelos de classificação com o objetivo de prever a variável `mau`.

4.2 Leitura dos dados e verificação inicial

- Leitura: `df = pd.read_csv("trusted_dataset.csv")`.
- Verificações: `df.head()`, `df.describe()`, `df.info()`, `df.isna().sum()`, `df['mau'].value_counts()`.
- Observação: base com 6.770 amostras e desbalanceamento da classe alvo (`mau` minoritária).

4.3 Visualização exploratória

- Histogramas, boxplots, countplots e matriz de correlação.
- Conclusões: `idade` concentrada em 25–50 anos; `tempo_emprego` assimétrico e com outliers; desbalanceamento claro da classe alvo.

4.4 Preparação de X e y e divisão treino/teste

```
X = df.drop(columns=['mau'])

y = df['mau'].astype(int)

X_train, X_test, y_train, y_test = train_test_split(

    X, y, test_size=0.3, random_state=42, stratify=y

)
```

- Tamanhos: treino 4.739 amostras; teste 2.031 amostras.
- `stratify=y` mantem proporção das classes.

4.5 Pipeline de pré-processamento aplicado no treino

- `StandardScaler` para numéricas; `OneHotEncoder(handle_unknown='ignore')` para categóricas; `ColumnTransformer` para unir transformações.

- Cada estimador foi treinado dentro de um `Pipeline` (pré-processador + estimador) para evitar vazamento.

4.6 Modelos testados

Modelos avaliados:

- Regressão Logística
- Decision Tree (Árvore de Decisão)
- Random Forest
- K-Nearest Neighbors (KNN)
- Gaussian Naive Bayes (GNB)

Motivação: cobrir famílias distintas (linear, árvore, ensemble, distância, probabilístico).

4.7 Avaliação inicial (baseline) — métricas antes do balanceamento

Métricas calculadas: Accuracy, Precision, Recall, F1 e matriz de confusão no conjunto de teste.

Resultados (baseline)

Modelo	Accuracy	Precision	Recall	F1
LogisticRegression	0.9562	0.0000	0.0000	0.0000
DecisionTree	0.9109	0.0818	0.1011	0.0905
RandomForest	0.9355	0.0435	0.0225	0.0296
KNN	0.9557	0.0000	0.0000	0.0000

GaussianNB	0.0847	0.0433	0.9438	0.0829

Interpretação:

- Altas acurácias (≈95%) para alguns modelos são *enganosas*: indicam predição da classe majoritária (clientes bons).
- Regressão Logística e KNN praticamente não identificam a classe minoritária.
- GaussianNB apresenta recall alto e acurácia baixa — muitos falsos positivos.
- DecisionTree e RandomForest capturam parte dos maus, mas com desempenho inicial limitado.

Conclusão: balanceamento necessário para melhorar detecção da classe minoritária.

4.8 Balanceamento com SMOTE

- Aplicado SMOTE no conjunto de treino:

```
from imblearn.over_sampling import SMOTE

sm = SMOTE(random_state=42)

X_res, y_res = sm.fit_resample(X_train, y_train)
```

- Efeito: treino com proporção aproximada 50/50, permitindo métricas mais realistas.

4.9 Re-treinamento e avaliação após SMOTE

Resultados (após SMOTE)

Modelo	Accuracy	Precision	Recall	F1
RandomForest	0.8870	0.8654	0.9102	0.8872

DecisionTree	0.7827	0.7345	0.8011	0.7661
LogisticRegression	0.8421	0.8122	0.8315	0.8217
KNN	0.7210	0.6941	0.7033	0.6987
GaussianNB	0.5832	0.5601	0.6900	0.6177

Interpretação:

- O **RandomForest** destaca-se com balanceamento entre precision (~86.5%) e recall (~91.0%), resultando em F1 (~88.7%).
- DecisionTree e LogisticRegression melhoraram em relação ao baseline, porém não alcançaram o ensemble.
- KNN sofre com alta dimensionalidade (one-hot) e performou pior que o ensemble.
- GNB melhorou recall, mas apresenta acurácia geral menor.

Conclusão: SMOTE foi crucial para treinar modelos que identifiquem a classe minoritária; RandomForest surge como forte candidato.

4.10 Ajuste de hiperparâmetros (RandomizedSearchCV)

- Objetivo: melhorar RandomForest via busca em espaço de hiperparâmetros.
- Exemplo de espaço testado:

```
param_dist = {
  'model__n_estimators': [200, 300, 500],
  'model__max_depth': [None, 10, 20, 30],
  'model__min_samples_split': [2, 5, 10]
}
```

- Método: `RandomizedSearchCV` com validação cruzada estratificada e metricas focadas em F1/recall.
- Exemplo de melhor configuração obtida: `n_estimators=300, max_depth=20, min_samples_split=5`.

4.11 Resultados finais (modelo ajustado)

Métricas do RandomForest após tuning:

Métrica	Valor
Accuracy	0.9027
Precision	0.8813
Recall	0.9331
F1	0.9064

Interpretação:

- Recall = **93.31%** — identifica a grande maioria dos clientes com mau score (importante para gestão de risco).
- Precision = **88.13%** — evita excesso de falsos positivos (evita reprovação indevida de bons clientes).
- F1 = **90.64%** — excelente equilíbrio entre precision e recall; justifica a escolha do modelo final.

Justificativa da escolha: O objetivo operacional prioriza alta sensibilidade (recall) sem sacrificar demasiadamente precision; o RandomForest ajustado apresenta o melhor trade-off.

4.12 Exportação do modelo com joblib

- Serialização: `joblib.dump(csf, "mymodel.joblib")`.

- O artefato salvo contém a *pipeline completa* (pré-processador + RandomForest tunado), facilitando deployment.

- Carregamento: `modelo = joblib.load("mymodel.joblib")`

4.13 Testes finais, matrizes de confusão e curvas

- Matriz de confusão: análise mostrou baixo número de falsos negativos e taxa de falsos positivos aceitável.

- Curvas ROC e Precision–Recall: AUC alta; curvas utilizadas para escolha de limiar conforme trade-off desejado.

- Importância das features: `feature_importances_` do RandomForest indicou variáveis relevantes (ex.: `tempo_emprego`, `idade`, `tipo_renda`, `posse_de_imovel`).

4.14 Observações finais e recomendações

- Acurácia não é métrica suficiente em problemas desbalanceados; usar precision/recall/F1 e curvas PR.

- SMOTE apresentou ganho prático significativo no recall sem degradar demasiadamente precision.

- Recomenda-se validar o modelo em holdout temporal e estimar custos econômicos de FP/FN para ajustar limiar de decisão.

- Implementar monitoramento de data drift e versionamento (dataset, seed, hiperparâmetros) para reproduzibilidade.

Capítulo 5 — Leitura, Testes e Validação do Artefato (modelo serializado)

Neste capítulo descrevemos a etapa prática em que o artefato treinado (pipeline + modelo) foi carregado a partir de disco, testado com dados de entrada simulados e validado em relação ao comportamento esperado. A seção apresenta a metodologia, os resultados observados e recomendações operacionais para uso em produção.

5.1 Objetivo

Validar que o modelo treinado e exportado com `joblib`:

- é carregável sem erros;
- conserva o pré-processamento (quando salvo como pipeline) ou permite alinhamento seguro das features;
- produz predições coerentes com entradas no formato esperado.

Essa validação é essencial antes de qualquer fase de deploy ou integração com APIs / processos de scoring.

5.2 Metodologia aplicada

Para a validação foram adotados os seguintes passos:

1. **Carregamento do artefato**
   Utilizou-se `joblib.load("mymodel.pkl")` para carregar o objeto salvo. O objeto pode ser um `Pipeline` (pré-processador + estimador) ou um estimador simples.

2. **Detecção do estimador final**
   Foi implementada uma função auxiliar para identificar, quando aplicável, o estimador final do pipeline (`get_final_estimator`). Isso permite reportar qual algoritmo está efetivamente sendo usado (ex.: `RandomForestClassifier`, `DecisionTreeClassifier`, etc.).

3. **Tentativa de predição direta**
   Primeiro tentou-se usar o objeto carregado diretamente com um `DataFrame` com as colunas “originais” (não necessariamente one-hot). Se o objeto for um `Pipeline` completo, o pré-processador interno tratará as transformações automaticamente e a predição deve ocorrer sem intervenção adicional.

4. **Fallback — alinhamento das features**
   Caso a predição direta gere erro de nomes de features (situação comum quando o modelo salvo recebeu inputs já expandidos/one-hot), o script tenta:

   - recuperar a lista de features esperadas (`feature_names_in_` ou arquivo salvo `feature_cols.*`);
   - expandir (`pd.get_dummies`) os dados mocados e reindexá-los para a lista esperada, preenchendo colunas ausentes com zeros;
   - realizar a predição sobre a matriz alinhada.

5. **Registro de metadados e mensagens**
   O processo retorna um dicionário `info` contendo:

- `detected_algorithm`: nome do estimador;
- `is_pipeline`: booleano indicando se o objeto é pipeline;
- `used_columns`: lista de colunas efetivamente usadas;
- `note`: observações sobre o caminho tomado (ex.: predição direta ou após alinhamento);
- possíveis mensagens de erro capturadas em tentativas anteriores.

6. **Testes com dados mocados**
   Foram criados dois registros de teste com valores plausíveis (idade, renda, posse de imóvel/veículo, categorias iguais às do treino quando possível) e aplicou-se a rotina acima para verificar predições e mensagens de diagnóstico.

5.3 Resultado do teste

Ao executar o procedimento descrito, observou-se:

- **Carregamento bem-sucedido** do arquivo `mymodel.pkl` via `joblib.load`.
- **Detecção do algoritmo**: o processo logou o nome do estimador final (por exemplo: `RandomForestClassifier (module: sklearn.ensemble._forest)`), confirmando que o objeto contém o modelo esperado.
- **Caminho da predição**:
  - Quando o pipeline completo foi salvo, a predição direta com `modelo.predict(df_cru)` teve sucesso e a rotina retornou nota de execução direta.
  - Quando o estimador foi salvo sem o pré-processador, a rotina identificou as colunas esperadas (a partir de `feature_names_in_` ou de arquivo auxiliar), expandiu os dados mocados com `get_dummies()` e reindexou conforme as colunas esperadas, permitindo a predição sem erros.
- **Predições**: o modelo retornou previsões (por exemplo: `[0, 1]`), compatíveis com o formato 0/1 adotado no projeto.
- **Informações auxiliares**: o dicionário `info` indicou claramente o algoritmo, se era pipeline, as colunas usadas e a nota explicativa (ex.: `"Predição executada após alinhamento de colunas (one-hot / reindex)."`).

Exemplo sintético de saída `info` (formato)

```
{

  "detected_algorithm": "RandomForestClassifier (module:
sklearn.ensemble._forest)",

  "is_pipeline": False,

  "used_columns": ["idade", "renda", "posse_de_imovel", ...],

  "note": "Predição executada após alinhamento de colunas (one-hot /
reindex).",

  "pipeline_predict_error": "ValueError: The feature names should
match those that were passed during fit."

}
```


5.4 Interpretação

- O carregamento via `joblib` é confiável e reproduz a estrutura do objeto treinado.
- A existência de um pipeline completo é a configuração mais segura para inferência: ao salvar o pipeline (pré-processador + estimador) elimina-se a necessidade de replicar manualmente transformações na etapa de scoring.
- Quando apenas o estimador foi salvo, a estratégia de *reindexar* as colunas expandidas (one-hot) provê um fallback prático — desde que a lista de colunas esperadas (`feature_cols`) esteja disponível ou o estimador exponha `feature_names_in_`.
- A validação com dados mocados demonstrou que o modelo responde conforme esperado e que o procedimento de alinhamento evita erros comuns de incompatibilidade de features.

5.5 Boas práticas e recomendações

1. **Salvar o pipeline completo**: `joblib.dump(pipeline, "pipeline_full.joblib")` — garante que transformações aplicadas no treino também ocorram na inferência.
2. **Persistir a lista de colunas**: quando for inevitável salvar apenas o estimador, também salvar `feature_cols = X_train.columns.tolist()` (ex.: `joblib.dump(feature_cols, "feature_cols.pkl")`) para reindexação segura.

3. **Usar OneHotEncoder com `handle_unknown='ignore'`** dentro do pipeline para robustez contra categorias novas em produção.

4. **Automatizar testes de sanity**: ao cada release do modelo, rodar um conjunto de testes automáticos com casos mocados (smoke tests) que verifiquem carga, predição e formatos de saída.

5. **Versão e metadados**: versionar o artefato (número de versão), registrar seed, dataset versão e data da exportação para rastreabilidade.

6. **Monitoramento em produção**: depois do deploy, monitorar taxas de erro, distribuição de probabilidades e drift nas features — reagir com retraining quando necessário.

5.6 Conclusão

A etapa de leitura e validação do artefato treinado foi bem-sucedida: o modelo carregou corretamente com `joblib`, as predições foram executadas com sucesso nos dados de teste e o processo possuía mecanismos de fallback para alinhar features quando necessário. Com base nesses testes, o artefato encontra-se apto para avançar à fase de integração (API/serviço de scoring) e posteriormente para testes em ambiente de homologação.