ChatGPT

Plano de Execução do Projeto de Previsão de Churn Bancário

1. Análises Exploratórias Extras

Para aprofundar o entendimento dos dados de churn bancário, serão conduzidas análises exploratórias adicionais nos dados. Essas etapas devem ser realizadas no notebook `eda_completo.ipynb`, complementando a análise exploratória inicial.

1.1 Clusterização (K-Means) e PCA

Objetivo: Identificar agrupamentos naturais de clientes e visualizar padrões em relação ao churn. A ideia é aplicar **PCA (Principal Component Analysis)** para reduzir a dimensionalidade dos dados a 2 componentes principais, facilitando a visualização dos grupos em 2D [1]. Em seguida, usar **K-Means** para segmentar os clientes em clusters com características similares, minimizando a variância dentro de cada cluster [2]. Isso pode revelar perfis distintos de clientes (por exemplo, grupos com maior propensão a churn).

Notebook: `eda_completo.ipynb` (adicionar seção de *Clusterização*).

Ferramentas: - Biblioteca **scikit-learn**: utilizar `sklearn.decomposition.PCA` para extrair 2 componentes principais, e `sklearn.cluster.KMeans` para agrupar os dados. - É recomendável padronizar ou normalizar os dados antes da PCA/K-Means (usar `StandardScaler` do scikit-learn), garantindo que variáveis em escalas diferentes não distorçam os resultados. - Pode-se definir um número de clusters inicial (ex.: 3 a 5 clusters) e avaliar heurísticas como *inertia* ou o método do cotovelo para escolher o número adequado de clusters.

Visualizações: - Gerar um **scatter plot 2D** dos dois primeiros componentes principais, plotando os pontos dos clientes e colorindo cada ponto pelo cluster atribuído pelo K-Means. Assim, visualizamos a separação dos clusters no espaço reduzido. Adicionalmente, pode-se usar diferentes símbolos ou cores de borda para indicar se o cliente deu churn ou não, identificando se algum cluster específico concentra mais churners. - Opcional: criar um **gráfico 3D** (com PCA de 3 componentes) para visualização interativa dos clusters, usando bibliotecas como *plotly* ou *matplotlib* (projeção 3D), embora 2D geralmente seja suficiente. - Calcular e exibir a **taxa de churn por cluster**: por exemplo, uma tabela ou gráfico de barras mostrando, para cada cluster, a porcentagem de clientes que churnaram. Isso evidenciará se algum grupo de clientes apresenta churn substancialmente maior [3] [4]. - Adicionar comentários interpretando os clusters: descrever perfis médios de cada cluster (médias de variáveis principais por cluster) e se esses grupos possuem alta ou baixa propensão a churn.

1.2 Correlações entre Variáveis Derivadas e Principais

Objetivo: Avaliar como as variáveis derivadas (criadas pela função `criar_variaveis_derivadas`) se relacionam com as variáveis originais e entre si. Isso ajuda a verificar se as variáveis derivadas adicionam informações novas ou se são altamente correlacionadas com variáveis existentes (o que indicaria redundância). Além disso, observar correlações com a variável alvo *Churn* pode revelar quais

atributos (derivados ou originais) têm relação mais forte com a saída. Por exemplo, em problemas de churn de telecom já foi identificado que *tempo de contrato*, *tipo de contrato* e *serviços usados* têm correlação significativa com o churn [5].

Notebook: `eda_completo.ipynb` (seção de *Análise de Correlação*).

Ferramentas: - **Pandas/NumPy:** usar `DataFrame.corr()` para calcular a matriz de correlação de Pearson entre variáveis numéricas. Incluir tanto as variáveis principais quanto as derivadas no cálculo. - Para variáveis categóricas ou mix de categórica e numérica, considerar técnicas adequadas: por exemplo, usar coeficiente de correlação *bisserial* para variáveis binárias vs contínuas, ou a biblioteca **dython** (método `compute_associations`) para obter medidas de associação envolvendo variáveis categóricas [6]. - **Seaborn:** utilizar `sns.heatmap` para visualizar a matriz de correlação. Configurar uma máscara para metade superior, se necessário, e anotar coeficientes acima de um limiar para destacar correlações fortes (positivas ou negativas).

Visualizações: - **Mapa de calor (heatmap)** de correlações: incluir todas as variáveis derivadas e principais. Destacar com cores mais fortes as correlações acima de, por exemplo, 0.5 ou 0.7 em magnitude. Isso permite identificar grupos de variáveis altamente correlacionadas. - **Matriz de dispersão (pairplot) focada:** se viável, plotar scatter plots de algumas variáveis derivadas vs. principais para visualizar relações não-lineares ou outliers. Por exemplo, se uma variável derivada for uma razão ou diferença de atributos, verificar graficamente sua relação com os atributos originais. - Calcular a **correlação de cada variável com o churn** (convertendo churn em {0,1}): apresentar possivelmente um pequeno gráfico de barras ordenado mostrando quais variáveis (originais e derivadas) têm maior correlação absoluta com a variável alvo. Embora correlação linear não capture totalmente relação com variável binária, pode dar indícios iniciais. - Anotar insights: e.g., "Variável derivada X tem alta correlação com variável original Y, indicando que X pode estar capturando a mesma informação que Y. Já a variável derivada Z apresentou correlação baixa com todos originais, sugerindo que adiciona informação nova possivelmente útil." Essas observações guiam a seleção de features e evitam multicolinearidade excessiva.

1.3 Verificação do Equilíbrio das Classes (Churn vs Não Churn)

Objetivo: Quantificar o desbalanceamento da classe alvo e avaliar a necessidade de técnicas de balanceamento. Em muitos conjuntos de churn, a classe "churn" (clientes que saíram) é bem menor que "não churn". Identificar a proporção exata no dataset bancário ajudará a planejar estratégias de modelagem (por exemplo, uso de SMOTE, ajuste de pesos). Um forte desbalanceamento exige atenção, pois modelos tendem a ignorar a classe minoritária se nenhuma medida for tomada [7].

Notebook: `eda_completo.ipynb` (seção inicial ou de distribuição da variável alvo).

Ferramentas: - **Pandas:** usar `value_counts()` na coluna de churn para obter contagens de cada classe. Calcular também a proporção percentual de churn. - **Seaborn/Matplotlib:** `sns.countplot` ou `plt.bar` para visualizar o número de exemplos em cada classe. - Estatísticas: número total de clientes, número de churners, % churn (churn rate). Se houver conjuntos de treino/teste já separados, verificar o balanceamento em cada um para garantir que a separação manteve a proporção.

Visualizações: - **Gráfico de barras da distribuição de classes:** um gráfico simples mostrando a barra de "Churn = Sim" vs "Churn = Não". Adicionar rótulos com os valores absolutos e percentuais em cada barra para clareza. Por exemplo, se 20% dos dados são churn, destacar esse valor. - Se o desequilíbrio for muito grande (ex: < 10% de churn), enfatizar com uma anotação que o dataset é desbalanceado e

que isso **justifica o uso de técnicas de balanceamento de dados na etapa de modelagem**. - Além do gráfico, incluir comentário sobre as implicações: por exemplo, "*Observa-se um desbalanceamento significativo: apenas X% dos clientes representam churn. Sem tratamento, um modelo poderia obter alta acurácia simplesmente prevendo 'não churn' para todos, ignorando os churners.*". Esse entendimento reforça a importância das etapas de balanceamento e métricas adequadas (como *precision*, *recall*, *f1-score* para a classe churn) na avaliação dos modelos.

1.4 Distribuição das Variáveis Transformadas (derivadas)

Objetivo: Examinar a distribuição de cada variável criada por `criar_variaveis_derivadas` para entender seu comportamento, detectar **outliers** ou **skews** significativos e verificar se as transformações atingiram o efeito desejado (por exemplo, normalização ou redução de skew). Isso garante que as novas features estejam prontas para uso em modelos (muitas técnicas de modelagem assumem distribuições aproximadamente normais ou se beneficiam de features escalonadas). A análise de distribuição faz parte fundamental da EDA, mostrando tendências, dispersão e variabilidade dos dados [8].

Notebook: `eda_completo.ipynb` (seção de *Distribuição de Variáveis*).

Ferramentas: - **Matplotlib/Seaborn:** utilizar histogramas (`plt.hist` ou `sns.histplot`) e/ou densidade (`sns.kdeplot`) para visualizar a distribuição de cada variável derivada. Se forem muitas variáveis, pode-se fazer uma grid de subplots, ou focar nas principais derivadas que serão usadas nos modelos. - **Boxplot e violin plot:** úteis para ver outliers e distribuição resumida. Um `sns.boxplot` por variável (ou vários boxplots lado a lado) pode destacar valores atípicos nas features derivadas. - Comparar **distribuições antes e depois** (se a variável derivada for, por exemplo, um log de uma original, ou uma normalização): pode-se sobrepor no mesmo gráfico a distribuição da variável original vs derivada para ilustrar a mudança. - **SciPy** (opcional): calcular medidas de skewness ou testes de normalidade, se for relevante documentar que a transformação reduziu assimetria.

Visualizações: - **Histogramas individuais:** para cada variável derivada, mostrar a frequência de valores. Caso as escalas variem muito, considerar usar escala logarítmica no eixo x ou bins adequados. - **Histogramas segmentados por churn:** Uma abordagem informativa é plotar dois histogramas sobrepostos de uma variável derivada, separando clientes churn e não churn (por exemplo, usando cores/transparência ou em dois painéis). Isso revela se a distribuição dessa feature difere entre os grupos – possivelmente indicando poder preditivo. Por exemplo, se a variável derivada "número de produtos ativos" tiver distribuição mais baixa para churners do que para não churners, isso surgirá nesses gráficos. - **Boxplots por classe:** para cada variável derivada, fazer um boxplot comparando a distribuição nos grupos churn vs não churn. Isto destaca medianas e outliers em cada grupo. - Adicionar anotações sobre cada variável: ex: "*A variável derivada X apresenta distribuição altamente assimétrica à direita, indicando alguns clientes com valores excepcionalmente altos. Poderá ser necessário aplicar uma transformação logarítmica para esta variável.*" ou "*A variável Y (derivada) tem distribuição aproximadamente normal e similar entre churners e não churners, sugerindo que talvez não diferencie muito o churn.*" - Essa inspeção garante qualidade nas features utilizadas nos modelos e pode inspirar ajustes adicionais de engenharia de atributos, se necessário.

2. Explicabilidade dos Modelos

Para tornar os resultados do modelo acionáveis e confiáveis, será dedicada uma etapa à explicação e interpretação dos modelos de churn treinados. Conforme as boas práticas de *Machine Learning*, além de prever corretamente, é crucial entender **por que** o modelo faz certas predições – especialmente em

churn, onde ações de negócio dependem de saber quais fatores contribuem para a saída do cliente. Vamos focar em duas abordagens complementares de explicabilidade: **importância de features** e **valores SHAP**. Idealmente, isso será feito **para cada modelo** (XGBoost, LightGBM, Random Forest) em seus notebooks dedicados (por exemplo: `XGBoost_Model_Analysis.ipynb`, etc), seguindo a estrutura das aulas de Regressão/Classificação (Aulas 03 e 04) onde após o treinamento avalia-se a interpretabilidade.

2.1 Importância de Features (Feature Importance)

Objetivo: Identificar quais variáveis têm maior influência nas predições de cada modelo, em nível global. A importância de features ajuda a resumir o comportamento do modelo – atributos com *feature importance* alta são os que mais contribuíram para as decisões do modelo no conjunto de treino [9]. Isso responde, por exemplo: "quais fatores mais contribuem para o cliente churnar segundo o modelo X?". Comparar a importância entre modelos também permite verificar consistência (se diferentes algoritmos concordam que certa variável é crucial).

Notebook: Nos notebooks específicos de cada modelo (ex.: inserir seção *Feature Importance* em `XGBoost_Model_Analysis.ipynb`, `LightGBM_Model_Analysis.ipynb`, etc). Opcionalmente, no `comparacao_modelos.ipynb` pode-se compilar um resumo das top features de cada modelo lado a lado.

Ferramentas: - Para modelos de árvores, usar os atributos nativos: por exemplo, **XGBoost/LightGBM** possuem `model.feature_importances_` (importâncias por ganho, por padrão) [9]. **RandomForestClassifier** também fornece `feature_importances_` baseada na redução de impureza (Gini) acumulada. - Alternativamente (ou complementarmente), utilizar **Permutation Importance** do scikit-learn (`sklearn.inspection.permutation_importance`) para uma visão agnóstica do modelo – isso embaralha cada feature e mede o impacto na métrica, podendo ser mais interpretável para modelos complexos. - Ordenar as features pela importância e selecionar, por exemplo, as top 10 para destacar em gráficos. - **Pandas:** criar um DataFrame com duas colunas: nome da feature e importância, e ordenar. - **sklearn.metrics:** mesmo na explicação, será útil ter em mente métricas de desempenho por feature, mas aqui foca-se mais na contribuição do recurso do que na métrica final.

Visualizações: - **Gráfico de barras da importância das features:** plotar as importâncias em ordem decrescente. Geralmente, usar as top 10 ou 15 variáveis para não poluir (caso muitas variáveis). Incluir rótulos no gráfico com o valor numérico de importância para facilitar a leitura. Por exemplo, usar `sns.barplot` horizontal, onde o eixo x é o score de importância e o y são os nomes das features. - **Comparação entre modelos:** pode-se apresentar múltiplos gráficos lado a lado (um para cada modelo) ou um gráfico combinado (por exemplo, cores diferentes para cada modelo, embora isso fique melhor em um relatório do que no notebook). Essa comparação mostrará, por exemplo, se *Idade*, *Renda* e *Número de produtos* são consistently importantes em todos os modelos, ou se algum modelo deu peso incomum a alguma variável derivada específica. - Interpretar os resultados no contexto do negócio: ex: "*No modelo XGBoost, as três features mais importantes foram Tempo_de_Relacionamento, Saldo_médio e Reclamações_no_SAC (fictícias). Isso sugere que clientes com pouco tempo de casa e saldo baixo, ou que registraram muitas reclamações, tiveram grande influência nas predições de churn.*" Esse tipo de insight conecta o modelo aos fatores de negócio. - **Dica:** Confirmar se a escala das importâncias do XGBoost/LightGBM está normalizada (algumas libs dão importância relativa somando 1, outras em valores absolutos). Normalizar ou expressar em % pode facilitar a explicação ("Feature A contribuiu com 30% da importância total").

2.2 Valores SHAP (Shapley Additive Explanations)

Objetivo: Fornecer explicações detalhadas **globais e locais** para os modelos usando valores SHAP. O SHAP é uma técnica baseada em teoria dos jogos que explica a predição de um modelo atribuindo a cada feature um valor que representa sua contribuição para aquela predição. Diferente da importância global tradicional, o SHAP mostra exatamente **como cada feature impacta cada previsão individual**, além de permitir visualizar a influência média de cada feature [10]. Assim, podemos entender *por que* um cliente específico foi previsto como churn, e quais características geralmente aumentam ou diminuem a probabilidade de churn no modelo.

Notebook: Também nos notebooks de cada modelo (ex.: seção *SHAP Analysis* em cada um). SHAP pode ser computacionalmente custoso, então sugere-se aplicá-lo após ter o modelo final treinado. Caso o dataset de treino seja muito grande, pode-se amostrar um subconjunto para análise SHAP (ex.: 1000 pontos) para viabilizar os gráficos, mantendo representatividade.

Ferramentas: - Biblioteca **SHAP**: instalar via pip se necessário (`!pip install shap`). Utilizar o objeto `shap.Explainer` ou específico `shap.TreeExplainer` (otimizado para modelos de árvore como XGBoost, LightGBM e RandomForest). Ex: `explainer = shap.TreeExplainer(model)` seguido de `shap_values = explainer.shap_values(data)`. - Para modelos binários, o shap geralmente retorna um array de valores shap por classe ou apenas para a classe positiva dependendo da função utilizada – focar nos valores correspondentes à classe churn. - **Visualizações SHAP:** a biblioteca shap possui funções prontas: `shap.summary_plot`, `shap.dependence_plot`, `shap.force_plot`, etc. Usar essas funções para gerar gráficos interpretativos (ver abaixo). - **Matplotlib**: os gráficos do shap podem ser incorporados nos notebooks; certificar-se de configurar `plt.show()` se necessário. Em casos de ambientes restritos, converter gráficos shap em imagens estáticas pode ser útil. - (Opcional) **SHAP interactions:** shap também permite calcular interações entre features (`shap_interaction_values`), mas isso é avançado – usar somente se houver interesse explícito em analisar interações.

Visualizações (por modelo): - **SHAP Summary Plot (Beeswarm)**: esse gráfico mostra, em uma única visão, a importância *e* efeito de todas as features. Cada ponto representa um valor SHAP de uma instância para uma feature; no eixo y listam-se as features ordenadas por importância média, e no eixo x o valor SHAP. A cor do ponto indica o valor da feature (baixo a alto). Assim, é possível ver, por exemplo, que valores altos de uma feature específica deslocam a previsão para churn positivo (SHAP > 0) ou negativo (SHAP < 0) [11]. Gerar este gráfico para cada modelo – espera-se que as features mais importantes identificadas acima apareçam no topo, confirmando a análise de importância global. - **Gráfico de barras de importância global SHAP:** semelhante à feature importance tradicional, o `shap.summary_plot` pode ser chamado com `plot_type="bar"` para mostrar a contribuição média absoluta de cada feature. Isso pode complementar a importância nativa do modelo com uma métrica mais consistente entre diferentes algoritmos. - **SHAP Dependence Plots:** escolher 2-3 features principais e plotar seus gráficos de dependência (`shap.dependence_plot(feature, shap_values, data)`). Esse gráfico mostra como o valor da feature (eixo x) se relaciona com o valor SHAP (impacto no output, eixo y), possivelmente revelando tendências (por exemplo, "*à medida que a variável X aumenta, o impacto no churn torna-se mais positivo*"). Pontos são coloridos por outra feature para evidenciar interações (a biblioteca escolhe automaticamente uma feature correlacionada). Esses gráficos ajudam a explicar *como* cada feature influencia o modelo: ex.: "Clientes com **Saldo baixo** tendem a ter SHAP positivo para churn – ou seja, saldo menor contribui para aumentar a probabilidade de churn no modelo". - **Fornecer exemplos locais:** utilizar o **force plot** ou **waterfall plot** do shap para exemplos individuais. Por exemplo, pegar um cliente real/ficcional que o modelo prevê churn e mostrar uma visualização waterfall: a predição do modelo (ex.: 0.8 probabilidade de churn) é decomposta como a soma das contribuições de cada feature

em relação à média. O gráfico force plot (em Jupyter) interativo mostra setas vermelhas e azuis empurrando a predição para cima ou para baixo [12]. Em relatório final, pode-se converter isso em um gráfico estático tipo waterfall. Esse passo ilustra: "*Para o cliente A, as features B e C aumentaram significativamente a chance de churn (valores SHAP de +0.20 e +0.15), enquanto a feature D reduziu a chance (-0.10).*". - **Comparar comportamento entre modelos:** comentar diferenças, se houver. Ex: "O modelo Random Forest atribui impacto fortemente positivo para churn a idade baixa do cliente, enquanto o XGBoost mostra um efeito mais neutro da idade nos valores SHAP. Isso sugere que o Random Forest capturou algum padrão que associa idade com churn que o XGBoost não considerou tão importante." Análises assim ajudam a decidir modelos ou gerar hipóteses de negócio. - Ressaltar que os valores SHAP fornecem explicações mais robustas e consistentes: eles garantem propriedades de somatória (os SHAP values de um exemplo somam para a diferença da predição em relação à base). Isso aumenta a confiança de stakeholders nos resultados, pois podemos apontar *quais fatores levaram à predição de churn de cada cliente* de forma quantitativa.

3. Refinamento e Investigação Adicional

Após realizar a EDA e treinar modelos base, esta etapa foca em **refinar o modelo e investigar melhorias**. Consiste em abordar o desbalanceamento da base e otimizar hiperparâmetros dos algoritmos. Isso deve seguir uma ordem lógica: primeiro, tratar o desbalanceamento (se necessário) para garantir que a otimização de hiperparâmetros e avaliação de modelos seja justa em relação à classe minoritária; depois, realizar *fine-tuning* dos modelos para extrair o máximo desempenho. As atividades podem ser conduzidas no notebook de **comparação de modelos (**`comparacao_modelos.ipynb`**)**, visto que envolvem testes comparativos, mas partes específicas (como tuning) podem ser feitas em notebooks separados de cada modelo dependendo da conveniência computacional e estrutura do projeto.

3.1 Balanceamento de Dados (SMOTE, Undersampling, Oversampling)

Objetivo: Tratar o desbalanceamento identificado na seção 1.3 para evitar que os modelos ignorem a classe churn (minoritária). Testaremos diferentes técnicas de reamostragem dos dados de treino e avaliaremos o impacto na performance. Estratégias incluem **oversampling** (aumentar a classe minoritária duplicando ou sintetizando exemplos), **undersampling** (reduzir a classe majoritária) ou combinações. Em particular, aplicar o **SMOTE (Synthetic Minority Oversampling Technique)** para gerar exemplos sintéticos de churners pode melhorar a aprendizagem do modelo sobre essa classe, ao invés de apenas duplicar dados existentes [13]. Também podemos testar *undersampling* aleatório da classe não-churn para ver se resultados melhoram, embora isso jogue fora dados.

Notebook: Principalmente em `comparacao_modelos.ipynb`. Inserir uma seção de *Experimentos de Balanceamento* onde os modelos escolhidos são treinados/avaliados com dados balanceados de formas distintas. Se for muito extenso treinar todos os modelos para cada técnica, poderíamos primeiro focar no modelo melhor performando (por ex., XGBoost) para experimentar as técnicas, e depois aplicar a melhor aos demais. Alternativamente, usar pipelines para aplicar o mesmo procedimento de reamostragem a todos os modelos dentro de loops.

Ferramentas: - Biblioteca **imbalanced-learn (imblearn)**: especialmente a classe `imblearn.over_sampling.SMOTE` para oversampling sintético [14], `imblearn.over_sampling.RandomOverSampler` para oversampling simples, e `imblearn.under_sampling.RandomUnderSampler` para undersampling. Também existe `imblearn.combine.SMOTETomek` e outros métodos combinados que podem ser explorados. - **Pipeline do scikit-learn:** imblearn oferece `Pipeline` compatível, permitindo encadear o

oversampling *apenas nos dados de treino* dentro do fluxo de cross-validation. Ex: `Pipeline([('smote', SMOTE()), ('model', RandomForestClassifier()) ])` – assim, em cada split de CV o SMOTE é aplicado somente no treino, evitando vazamento para teste. - **Parâmetros de modelo:** outra abordagem é ajustar pesos de classe (ex.: `class_weight='balanced'` no RandomForest ou parâmetro `scale_pos_weight` no XGBoost) em vez de reamostrar. Vale mencionar e possivelmente testar essa via, pois é menos custosa computacionalmente (os modelos internamente penalizam erros na classe minoritária) [7] . - **sklearn.metrics:** utilizar métricas adequadas para avaliar o efeito do balanceamento. Além de acurácia, dar ênfase em **Recall da classe churn**, **Precision**, **F1-score**, e **AUC-ROC** – pois o objetivo é melhorar a detecção de churners sem inflar muito falsos positivos. `classification_report` e matriz de confusão ajudarão.

Procedimento: - Re-dividir dados de treino/validação se necessário (garantir que as técnicas de oversampling não apliquem no teste). Em seguida, para cada técnica: 1. Aplicar a técnica no conjunto de treino. 2. Treinar um modelo (por ex., usar um modelo fixo ou um conjunto de modelos). 3. Avaliar no conjunto de validação ou teste não balanceado. 4. Registrar métricas. - Comparar os resultados entre técnicas e com o baseline sem reamostragem.

Visualizações: - **Tabela comparativa de métricas:** montar uma tabela mostrando, para cada abordagem (Sem balanceamento, SMOTE, Oversample, Undersample, etc), as métricas principais (Accuracy, Recall_churn, Precision_churn, F1_churn, AUC). Destacar em negrito ou cor as melhores métricas de recall/F1 para churn. - **Gráfico de barras ou linhas:** por exemplo, um gráfico de barras agrupadas mostrando o *Recall* da classe churn para cada modelo sob diferentes métodos de balanceamento – isso visualiza facilmente qual técnica mais aumentou a sensibilidade. Outro gráfico similar poderia mostrar o efeito no *Precision* (pois muitas vezes há trade-off). - **Matrizes de confusão:** plotar a matriz de confusão normalizada antes e depois do balanceamento (pelo melhor método escolhido) para o modelo principal. Espera-se ver aumento na detecção de verdadeiros positivos (churners identificados) após o balanceamento, ainda que com possível aumento de falsos positivos. - Incluir comentários interpretando: "*Aplicando SMOTE, o recall de churn aumentou de 0.50 para 0.75 no modelo X, indicando que o modelo passou a recuperar bem mais churners, embora com leve queda na precisão.*" e "*O undersampling puro mostrou pior desempenho geral, possivelmente por descartar muita informação (o modelo perdeu precisão na classe não-churn).*". Dessa forma, justificar a técnica escolhida para o modelo final (por ex., usar SMOTE nos experimentos seguintes, ou um híbrido). - **Observação:** Documentar também o tempo de execução ou complexidade, se relevante – SMOTE aumenta o tempo de treino, mas em datasets moderados isso é manejável. Se o projeto envolver implantação, notar que técnicas de oversampling seriam aplicadas apenas em retrain offline, não no scoring em produção (onde se usaria o modelo já treinado).

3.2 Tunagem de Hiperparâmetros

Objetivo: Otimizar os modelos ajustando seus hiperparâmetros para melhorar performance e evitar overfitting. Os modelos inicialmente foram treinados com parâmetros padrão ou configurados manualmente; agora faremos uma busca sistemática pelos melhores hiperparâmetros. Um bom ajuste de hiperparâmetros pode **aumentar significativamente a acurácia e robustez do modelo** [15] , bem como melhorar a capacidade de generalização em dados novos. Focaremos nos três algoritmos principais (XGBoost, LightGBM, RandomForest), procurando o conjunto ótimo de parâmetros para cada um.

Notebook: A tarefa pode ser dividida por modelo, para facilitar: por exemplo, realizar a busca de hiperparâmetros de XGBoost dentro do notebook `XGBoost_Model_Analysis.ipynb`, e similar para LightGBM e RandomForest nos seus notebooks. Isso organiza melhor os resultados e permite rodar em paralelo se necessário. Posteriormente, os resultados dos melhores modelos podem ser resumidos no

`comparacao_modelos.ipynb` (comparando *baseline vs tunado*). Caso os recursos sejam limitados, pode-se optar por fazer toda a busca no `comparacao_modelos.ipynb` de forma sequencial, mas isso pode tornar o notebook pesado – dividir costuma ser mais prático.

Ferramentas: - **GridSearchCV** e **RandomizedSearchCV** do scikit-learn: São as ferramentas principais para busca. A escolha entre grid ou random depende do espaço de hiperparâmetros: para muitos parâmetros ou valores possíveis, RandomizedSearchCV é mais eficiente, amostrando combinações aleatórias [16]. GridSearchCV é viável para espaços menores e para garantir avaliação de combinações específicas. - **Optuna/Hyperopt/Sklearn HalvingGridSearch**: mencionar como opção avançada. Por exemplo, **Optuna** permite otimização Bayesiana de forma mais automática. Se o projeto já possui essas dependências, pode ser uma alternativa para melhor eficiência. Porém, dado o contexto educacional, manteremos o foco em Grid/Random search, a menos que o handout da aula tenha apresentado Optuna. - **Parâmetros a tunar**: - *XGBoost*: learning_rate (eta), max_depth das árvores, n_estimators (número de árvores), subsample, colsample_bytree, gamma (mínimo ganho), e scale_pos_weight (se não usar SMOTE, ajustar este para lidar com desbalanceamento). Focar nos mais influentes primeiro (learning_rate, max_depth, n_estimators) e depois fazer ajuste fino nos demais. - *LightGBM*: similar ao XGBoost (learning_rate, num_leaves que equivale a complexidade da folha, max_depth se aplicável, n_estimators, feature_fraction, bagging_fraction, reg_lambda etc.). - *RandomForest*: n_estimators, max_depth, min_samples_split, min_samples_leaf, max_features. O RandomForest tende a ter menos hiperparâmetros críticos, mas buscar um bom n_estimators e constraints de profundidade pode melhorar generalização. - **sklearn.model_selection**: usar 5-fold cross-validation (ou stratified k-fold se for classificação) durante a busca para avaliar desempenho médio. Definir *scoring* apropriado – por exemplo, usar `scoring='f1'` ou `'roc_auc'` focado na classe churn (pode usar `make_scorer` se precisar dar peso maior à classe minoritária). - **Recursos computacionais**: limitar o número de combinações para evitar buscas muito longas. Por exemplo, começar com RandomizedSearchCV 50 iterações, depois afunilar o grid em torno dos melhores. Utilizar `n_jobs=-1` se possível para paralelizar. - **Guardar resultados**: salvar o melhor estimador (`best_estimator_`) e `best_params_` para futura referência. Opcional: persistir em arquivos pickle dentro de `models/` (pasta já existente no projeto), garantindo reprodutibilidade.

Visualizações: - **Evolução da busca**: embora o GridSearchCV/RandomizedSearchCV execute internamente, podemos extrair os resultados do atributo `cv_results_` para análise. Por exemplo, plotar um gráfico da métrica vs. número de árvores, ou heatmap de duas dimensões de hiperparâmetros vs. score, se fizer grid 2D. Isso ajuda a visualizar a sensibilidade do modelo a certos parâmetros. - **Tabela de melhores parâmetros**: apresentar uma pequena tabela com os hiperparâmetros escolhidos para cada modelo e o respectivo desempenho obtido (ex.: "RandomForest: best_params = {...}, AUC = 0.84; XGBoost: best_params = {...}, AUC = 0.87; ..."). Isso documenta as escolhas finais. - **Comparação antes e depois (bar chart)**: por modelo, mostrar a métrica de teste antes do tuning vs depois do tuning. Por exemplo, um gráfico de barras para cada algoritmo com "F1 Score baseline" e "F1 Score tuned". Esperamos ver melhoria (mesmo que pequena, já é válida) – se algum caso não melhorar, comentar possivelmente overfitting ou que o default já era bom. - **Curvas de validação** (se aplicável): por exemplo, plotar a curva de validation score conforme aumenta `n_estimators` para o modelo boosted, para ilustrar escolha de estimadores ótimo (evitar overfitting). Ou curva de *learning rate* vs desempenho. - Ao final, **re-treinar o modelo final**: após determinar os melhores hiperparâmetros, treinar o modelo em todo o conjunto de treino (aplicando balanceamento se decidido) e então avaliar uma última vez no conjunto de teste para obter a métrica final do projeto. Apresentar essa métrica final (ex.: "Modelo XGBoost tunado com SMOTE alcançou AUC-ROC de 0.88 no teste, Recall de 0.80, Precision de 0.60..."). - Inclua discussões do resultado: "*A tunagem de hiperparâmetros elevou o desempenho do LightGBM em ~5 pontos percentuais de AUC, indicando que ajustes como redução do learning_rate e aumento do num_leaves permitiram capturar melhor os padrões de churn.*" e "*Notamos que além de melhorar a métrica global, o tuning ajudou a equilibrar melhor precision/recall para*

churn." - Ressalte a importância: hiperparameter tuning bem feito resulta em modelo mais forte segundo métricas do problema [17], porém tome cuidado com overfitting na validação – por isso usamos cross-validation rigorosa. Somente após esse processo consideramos o modelo pronto para produção/uso, com expectativa de generalização sólida.

Conclusão: Seguindo este plano estruturado – primeiro explorando dados e clusters (EDA aprofundada), depois desenvolvendo interpretabilidade (feature importance, SHAP) e finalmente refinando modelos (balanceamento e hiperparametrização) – teremos não apenas um modelo de churn bancário com boa acurácia, mas também *entendível* e *otimizado*. Todas as etapas acima devem ser documentadas nos notebooks correspondentes, alinhadas ao handout das aulas (EDA e Clusterização na Aula 02; modelagem e validação nas Aulas 03 e 04), usando as ferramentas recomendadas (pandas, scikit-learn, imblearn, SHAP, matplotlib/seaborn, sklearn.metrics, etc.). Assim, o projeto ficará completo, didático e preparado tanto para apresentar insights aos stakeholders quanto para entregar previsões de churn confiáveis para o negócio. [9] [10] [13] [15]

---

[1] [2] [3] [4] [9] Build a customer churn prediction model (with examples) | Hex
https://hex.tech/templates/data-science/churn-prediction/

[5] [6] [7] Aplicação de Churn Prediction com Machine Learning | by André Almeida | Medium
https://medium.com/@andre-almd/aplica%C3%A7%C3%A3o-de-churn-prediction-com-machine-learning-cca6164033ec

[8] Exploratory Data Analysis and Feature Engineering in Machine ...
https://medium.com/@meena.08.ch/eda-and-feature-engineering-in-machine-learning-3ec0d1e5026d

[10] A Gentle Introduction to SHAP for Tree-Based Models - MachineLearningMastery.com
https://machinelearningmastery.com/a-gentle-introduction-to-shap-for-tree-based-models/

[11] 18 SHAP – Interpretable Machine Learning
https://christophm.github.io/interpretable-ml-book/shap.html

[12] Be careful when interpreting predictive models in search of causal insights — SHAP latest documentation
https://shap.readthedocs.io/en/latest/example_notebooks/overviews/Be%20careful%20when%20interpreting%20predictive%20models%20in%20search%20of%20causal%20insights.html

[13] [14] SMOTE for Imbalanced Classification with Python - MachineLearningMastery.com
https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/

[15] [16] Hyperparameter Tuning - GeeksforGeeks
https://www.geeksforgeeks.org/machine-learning/hyperparameter-tuning/

[17] What Is Hyperparameter Tuning? | IBM
https://www.ibm.com/think/topics/hyperparameter-tuning