INTRODUÇÃO À MACHINE LEARNING

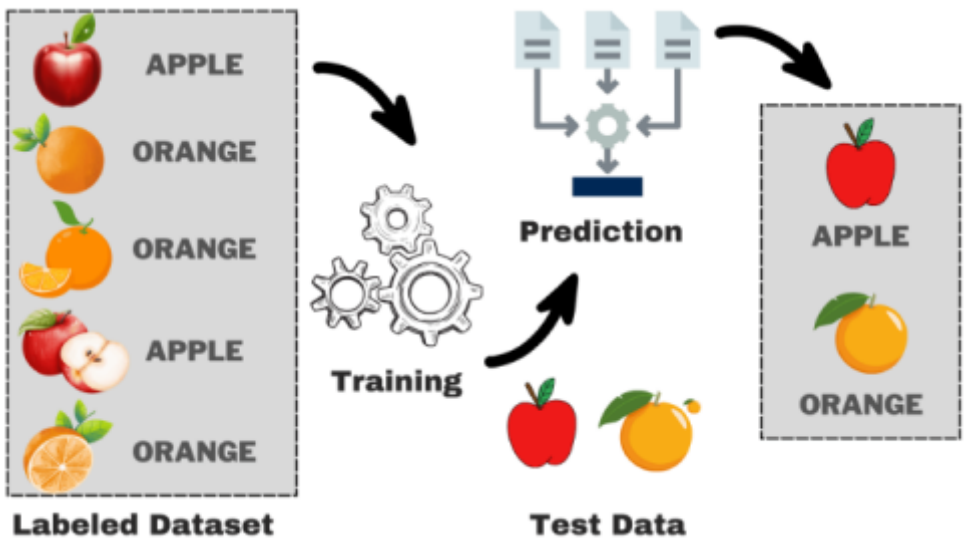
Leticia da Luz
www.linkedin.com/in/leticiadluz
https://github.com/leticiadluz
leticiadluz@gmail.com

O que é Aprendizado de Máquina?

- Aprendizado de máquina é a ciência (e arte) de programar computadores para que possam aprender a partir de dados. (Aurélien Géron, 2023).
- Seu filtro de spam é um programa de aprendizado de máquina que, com base em exemplos de e-mails de spam (identificados por usuários) e e-mails normais (não-spam), pode aprender a identificar e marcar mensagens como spam. Os exemplos utilizados pelo sistema para aprendizado são chamados de conjunto de treinamento. Cada exemplo individual nesse conjunto é chamado de instância de treinamento (ou amostra). A parte do sistema de aprendizado de máquina responsável por aprender e fazer previsões é chamada de modelo. Redes neurais e florestas aleatórias são exemplos de modelos de aprendizado de máquina.
- Desta forma, podemos definir Aprendizado de Máquina (Machine Learning) como uma subárea da Inteligência Artificial (IA) dedicada ao desenvolvimento de algoritmos e modelos que permitem aos computadores aprenderem a partir de dados. **Em vez de seguir regras explícitas programadas manualmente, os sistemas de aprendizado de máquina identificam padrões e relações nos dados, o que lhes permite fazer previsões ou tomar decisões com base em novas informações.**
- **Os sistemas de aprendizado de máquina podem ser classificados de acordo com a quantidade e o tipo de supervisão que recebem durante o treinamento. Existem muitas categorias, mas vamos discutir as principais: aprendizado supervisionado, não supervisionado e semissupervisionado.**

Aprendizado Supervisionado

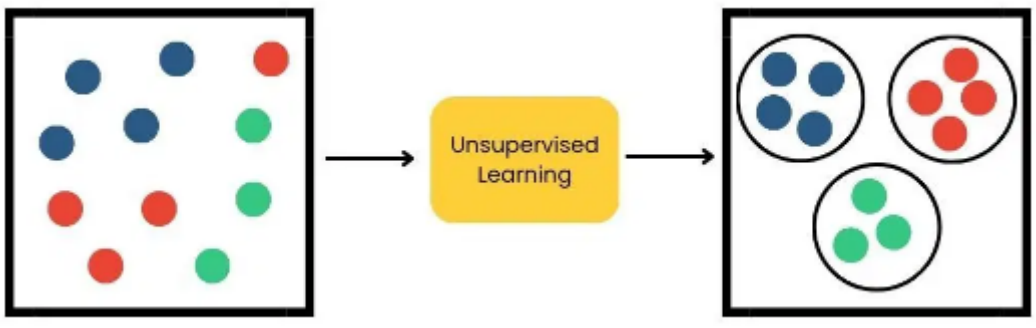
- No aprendizado supervisionado, o conjunto de treinamento fornecido ao algoritmo inclui as soluções desejadas, chamadas de rótulos. Isso significa que cada exemplo no conjunto de dados vem com uma entrada e a saída correta correspondente (também chamada de rótulo). O objetivo do modelo é aprender a mapear as entradas para as saídas de modo que possa prever a saída correta para novos exemplos não vistos.



- Durante o treinamento, o algoritmo ajusta seus parâmetros internos para minimizar a diferença entre as saídas previstas e as saídas reais, geralmente usando uma função de custo ou uma medida de erro.
- Uma tarefa típica de aprendizado supervisionado é a classificação. O filtro de spam é um bom exemplo disso: ele é treinado com muitos exemplos de e-mails juntamente com sua classe (spam ou ham), e deve aprender a classificar novos e-mails.
- Outra tarefa típica é prever um valor numérico alvo, como o preço de um carro, dado um conjunto de características (quilometragem, idade, marca, etc.). Esse tipo de tarefa é chamado de regressão. Para treinar o sistema, você precisa fornecer muitos exemplos de carros, incluindo tanto suas características quanto seus alvos (ou seja, seus preços).

Aprendizado Não Supervisionado

- No aprendizado não supervisionado, o conjunto de treinamento fornecido ao algoritmo não inclui os rótulos desejados. O sistema deve aprender a partir dos dados de entrada sem orientação explícita. O objetivo é explorar os dados e encontrar padrões ou estruturas ocultas sem qualquer supervisão ou orientação explícita sobre quais são os resultados corretos. Algumas técnicas comuns de aprendizado não supervisionado incluem:
  - Agrupamento (Clustering): Agrupar instâncias em clusters baseados em suas semelhanças. Um exemplo é o algoritmo K-means.
  - Redução de Dimensionalidade: Reduzir o número de variáveis sob consideração, encontrando uma representação simplificada dos dados. Um exemplo é a Análise de Componentes Principais (PCA).



- Outra tarefa importante de aprendizado não supervisionado é a detecção de anomalias. Por exemplo, detectar transações de cartão de crédito incomuns para prevenir fraudes, identificar defeitos de fabricação ou remover automaticamente outliers de um conjunto de dados antes de alimentá-lo a outro algoritmo de aprendizado. O sistema é mostrado principalmente instâncias normais durante o treinamento, de

modo que ele aprende a reconhecê-las; então, quando vê uma nova instância, ele pode dizer se parece uma instância normal ou se é provavelmente uma anomalia

Aprendizado Semissupervisionado

- O aprendizado semissupervisionado utiliza uma combinação de dados rotulados e não rotulados para treinamento. Este tipo de aprendizado é útil quando a rotulação de dados é cara ou demorada, mas os dados não rotulados são abundantes. Um exemplo é o uso de um pequeno conjunto de dados rotulados para orientar o aprendizado de um grande conjunto de dados não rotulados.
- Alguns serviços de hospedagem de fotos, como o Google Photos, são bons exemplos disso. Quando você faz upload de todas as suas fotos de família no serviço, ele automaticamente reconhece que a mesma pessoa A aparece nas fotos 1, 5 e 11, enquanto outra pessoa B aparece nas fotos 2, 5 e 7. Esta é a parte não supervisionada do algoritmo (agrupamento). Agora, tudo o que o sistema precisa é que você diga quem são essas pessoas. Basta adicionar um rótulo por pessoa e o sistema é capaz de nomear todos em cada foto, o que é útil para buscar fotos.
- A maioria dos algoritmos de aprendizado semi-supervisionado são combinações de algoritmos não supervisionados e supervisionados. Por exemplo, um algoritmo de clustering pode ser usado para agrupar instâncias semelhantes, e então cada instância não rotulada pode ser rotulada com o rótulo mais comum em seu grupo. Uma vez que todo o conjunto de dados esteja rotulado, é possível usar qualquer algoritmo de aprendizado supervisionado.
- Em resumo, o aprendizado semi-supervisionado é uma abordagem eficaz para lidar com a escassez de dados rotulados, combinando técnicas supervisionadas e não supervisionadas para maximizar o uso dos dados disponíveis e melhorar o desempenho dos modelos de aprendizado de máquina.

Aprendizado em Lote versus Aprendizado Online

- Outro critério usado para classificar sistemas de aprendizado de máquina é se o sistema pode aprender incrementalmente a partir de um fluxo de dados de entrada.

Aprendizado em lote:

- No aprendizado em lote, o sistema é incapaz de aprender incrementalmente: ele deve ser treinado usando todos os dados disponíveis. Isso geralmente leva muito tempo e recursos computacionais, então é tipicamente feito offline. Primeiro, o sistema é treinado e, em seguida, é lançado em produção e continua sem aprender mais; ele apenas aplica o que aprendeu. Isso é chamado de aprendizado offline.
- Infelizmente, o desempenho de um modelo tende a diminuir lentamente ao longo do tempo, simplesmente porque o mundo continua a evoluir enquanto o modelo permanece inalterado. Esse fenômeno é frequentemente chamado de degradação do modelo ou mudança nos dados. A solução é re-treinar regularmente o modelo com dados atualizados. Com que frequência você precisa fazer isso depende do caso de uso: se o modelo classifica imagens de gatos e cachorros, seu desempenho vai degradar muito lentamente, mas se o modelo lida com sistemas de evolução rápida, por exemplo, fazendo previsões no mercado financeiro, então é provável que decaia bastante rápido.
- Se você deseja que um sistema de aprendizado em lote saiba sobre novos dados (como um novo tipo de spam), você precisa treinar uma nova versão do sistema do zero no conjunto de dados completo (não apenas nos novos dados, mas também nos dados antigos), e então substituir o modelo antigo pelo novo.
- Essa solução é simples e geralmente funciona bem, mas o treinamento usando o conjunto completo de dados pode levar muitas horas, então você normalmente treinaria um novo sistema apenas a cada 24 horas ou até mesmo semanalmente. Se o seu sistema precisa se adaptar a dados que mudam rapidamente (por exemplo, para prever preços de ações), então você precisa de uma solução mais reativa.

Aprendizado online:

- No aprendizado online, você treina o sistema incrementalmente alimentando-o com instâncias de dados sequencialmente, seja individualmente ou em pequenos grupos chamados mini-lotes. Cada passo de aprendizado é rápido e barato, então o sistema pode aprender sobre novos dados conforme eles chegam. O aprendizado online é útil para sistemas que precisam se adaptar a mudanças extremamente rápidas (por exemplo, para detectar novos padrões no mercado de ações). Também é uma boa opção se você tiver recursos computacionais limitados.
- Além disso, algoritmos de aprendizado online podem ser usados para treinar modelos em conjuntos de dados enormes que não cabem na memória principal de uma máquina (isso é chamado de aprendizado fora do núcleo). O algoritmo carrega parte dos dados, executa uma etapa de treinamento nesses dados e repete o processo até ter executado em todos os dados.
- Um grande desafio com o aprendizado online é que se dados ruins forem alimentados para o sistema, o desempenho do sistema diminuirá, possivelmente rapidamente. Para reduzir esse risco, você precisa monitorar seu sistema de perto e desativar o aprendizado rapidamente (e possivelmente voltar a um estado anterior de funcionamento) se detectar uma queda no desempenho

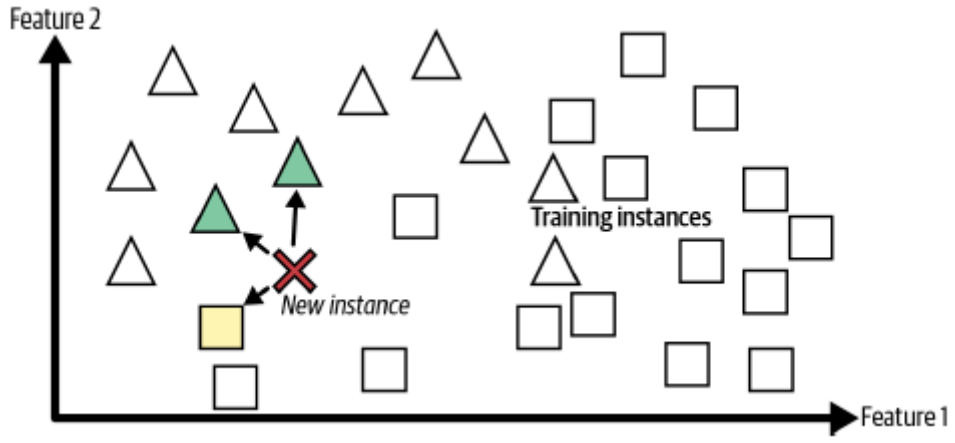
Aprendizado Baseado em Instância versus Aprendizado Baseado em Modelo

- Uma outra maneira de categorizar sistemas de aprendizado de máquina é pela forma como eles generalizam. A maioria das tarefas de aprendizado de máquina trata de fazer previsões. Isso significa que, dadas várias exemplos de treinamento, o sistema precisa ser capaz de fazer boas previsões (generalizar) para exemplos que nunca viu antes. Ter uma boa medida de desempenho nos dados de treinamento é bom, mas insuficiente; o verdadeiro objetivo é ter um bom desempenho em novas instâncias. Existem duas abordagens principais para generalização: aprendizado baseado em instância e aprendizado baseado em modelo.

Aprendizado baseado em instância

- Possivelmente a forma mais trivial de aprendizagem é simplesmente decorar. Se você fosse criar um filtro de spam dessa maneira, ele simplesmente marcaria todos os e-mails que são idênticos aos e-mails que já foram marcados pelos usuários - não é a pior solução, mas certamente não é a melhor.

- Em vez de apenas marcar e-mails que são idênticos aos e-mails de spam conhecidos, seu filtro de spam poderia ser programado para também marcar e-mails que são muito semelhantes aos e-mails de spam conhecidos. Isso requer uma medida de similaridade entre dois e-mails. Uma medida de similaridade (muito básica) entre dois e-mails poderia ser contar o número de palavras que eles têm em comum. O sistema marcaria um e-mail como spam se ele tiver muitas palavras em comum com um e-mail de spam conhecido.
- Isso é chamado de aprendizado baseado em instância: o sistema aprende os exemplos de cor e, em seguida, generaliza para novos casos usando uma medida de similaridade para compará-los aos exemplos aprendidos (ou a um subconjunto deles). Por exemplo, na imagem, a nova instância seria classificada como um triângulo porque a maioria das instâncias mais similares pertence a essa classe.



- Exemplos comuns incluem o k-vizinhos mais próximos (k-NN).

Aprendizado baseado em modelo

- O aprendizado baseado em modelo é uma abordagem de aprendizado de máquina onde um modelo matemático é criado a partir dos dados de treinamento. Esse modelo captura as relações subjacentes entre as variáveis de entrada e a variável alvo (ou saída), permitindo que ele faça previsões sobre novos dados.
- O processo envolve duas etapas principais: treinamento e predição. Durante o treinamento, um algoritmo de aprendizado é aplicado aos dados de treinamento para ajustar os parâmetros do modelo, geralmente minimizando uma função de erro ou perda. Exemplos comuns de algoritmos baseados em modelo incluem regressão linear, redes neurais, máquinas de vetores de suporte (SVM) e árvores de decisão. Após o treinamento, o modelo é utilizado para fazer previsões, aplicando as operações matemáticas definidas durante a fase de ajuste.

Problemas de Regressão Versus Problemas de Classificação

- As variáveis podem ser caracterizadas como quantitativas ou qualitativas (também conhecidas como categóricas). Variáveis quantitativas assumem valores numéricos. Exemplos incluem a idade, altura ou renda de uma pessoa, o valor de uma casa e o preço de uma ação. Em contraste, variáveis qualitativas assumem valores em uma das K diferentes classes ou categorias. Exemplos de variáveis qualitativas incluem o estado civil de uma pessoa (casado ou não), a marca do produto comprado (marca A, B ou C), se uma pessoa está inadimplente em uma dívida (sim ou não) ou um diagnóstico de câncer (Leucemia Mieloide Aguda, Leucemia Linfoblástica Aguda ou Sem Leucemia).
- Tendemos a nos referir aos problemas com uma resposta quantitativa como problemas de regressão, enquanto aqueles que envolvem uma resposta qualitativa são frequentemente chamados de problemas de classificação. No entanto, a distinção nem sempre é tão nítida. A regressão linear de mínimos quadrados é usada com uma resposta quantitativa, enquanto a regressão logística é tipicamente usada com uma resposta qualitativa (de duas classes, ou binária). Assim, apesar de seu nome, a regressão logística é um método de classificação. Mas, como ela estima as probabilidades das classes, também pode ser considerada um método de regressão.
- Alguns métodos estatísticos, como os K-vizinhos mais próximos e o boosting, podem ser usados tanto no caso de respostas quantitativas quanto qualitativas.

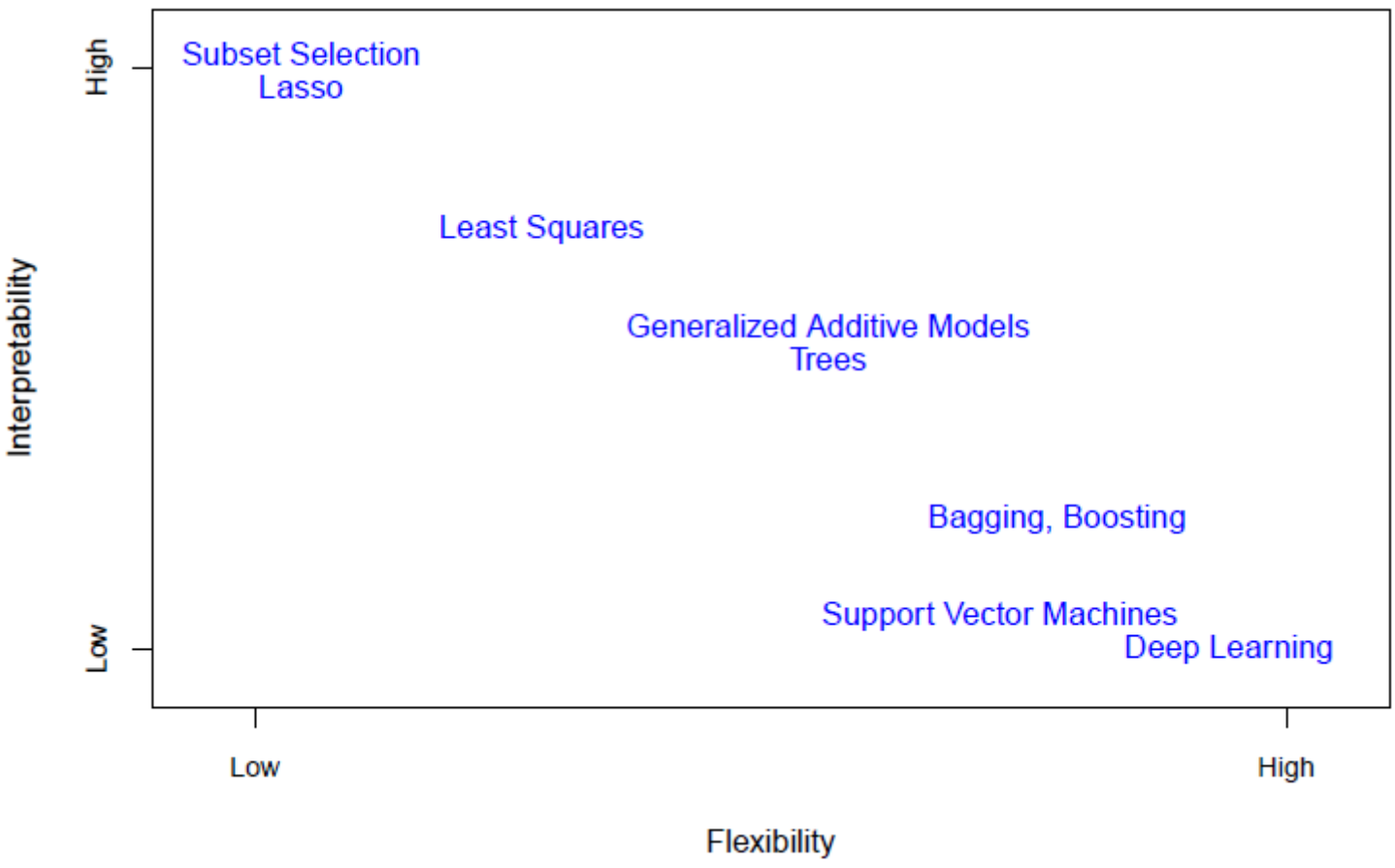
Estimadores

- Em machine learning, um estimador é um algoritmo ou modelo que é utilizado para inferir informações a partir de dados. O objetivo principal de um estimador é fazer previsões ou estimativas sobre dados não vistos com base em dados observados anteriormente. Exemplos: Regressão Linear Simples e Múltipla, Regressão Logística, Árvores de Decisão, Gradient Boosting, KNN (K-Nearest Neighbors), etc.

Trade-Off Entre Precisão da Previsão e Interpretabilidade do Modelo

- Dos muitos métodos usados em Machine Learning, alguns são menos flexíveis, ou mais restritivos, no sentido de que podem produzir apenas uma gama relativamente pequena de formas para estimar y. Por exemplo, a regressão linear é uma abordagem relativamente inflexível, pois pode gerar apenas funções lineares ou planos. Outros métodos são consideravelmente mais flexíveis, pois podem gerar uma gama muito mais ampla de formas possíveis para estimar y.
- Pode-se razoavelmente fazer a seguinte pergunta: por que escolheríamos usar um método mais restritivo em vez de uma abordagem muito flexível? Existem várias razões pelas quais podemos preferir um modelo mais restritivo. Se estivermos principalmente interessados em inferência, então modelos restritivos são muito mais interpretáveis. Por exemplo, quando a inferência é o objetivo, o modelo linear pode ser uma boa escolha. Em contraste, abordagens muito flexíveis, como boostings, podem levar a estimativas tão complicadas que é difícil entender como qualquer preditor individual está associado à resposta.
- A imagem fornece uma ilustração do trade-off entre flexibilidade e interpretabilidade para alguns dos métodos. A regressão linear de mínimos quadrados é relativamente inflexível, mas é bastante interpretável. O lasso, baseia-se no modelo linear, mas usa um procedimento de ajuste alternativo para estimar os coeficientes. Assim, nesse sentido, o lasso é uma abordagem menos flexível do que a regressão linear. Ele também é mais interpretável do que a regressão linear, porque no modelo final a variável de resposta estará relacionada apenas a um pequeno subconjunto dos preditores.

- Os modelos aditivos generalizados (GAMs) estendem o modelo linear para permitir certas relações não lineares. Consequentemente, os GAMs são mais flexíveis do que a regressão linear. Eles também são um pouco menos interpretáveis do que a regressão linear, porque a relação entre cada preditor e a resposta agora é modelada usando uma curva. Finalmente, métodos totalmente não lineares, como bagging, boosting, máquinas de vetor de suporte com kernels não lineares, e redes neurais (aprendizado profundo), são abordagens altamente flexíveis que são mais difíceis de interpretar.
- **Estabelecemos que, quando o objetivo é a interpretabilidade do modelo, há claras vantagens em usar métodos de aprendizado estatístico simples e relativamente inflexíveis. No entanto, em alguns cenários, estamos interessados apenas em previsão, e a interpretabilidade do modelo preditivo simplesmente não é de interesse.** Por exemplo, se buscamos desenvolver um algoritmo para prever o preço de uma ação, nosso único requisito para o algoritmo é que ele faça previsões precisas, a interpretabilidade não é uma preocupação. Nesse cenário, poderíamos esperar que fosse melhor usar o modelo mais flexível disponível. Surpreendentemente, isso nem sempre é o caso. Muitas vezes, obtemos previsões mais precisas usando um método menos flexível. Esse fenômeno, que pode parecer contraintuitivo à primeira vista, tem a ver com o potencial de overfitting em métodos altamente flexíveis.



Por que é necessário separar os dados?

- A única maneira de saber o quão bem um modelo irá generalizar para novos casos é realmente testá-lo em novos casos. Uma maneira de fazer isso é colocar seu modelo em produção e monitorar seu desempenho. Isso pode funcionar bem, mas se o seu modelo for terrivelmente ruim, seus usuários vão reclamar - não é a melhor ideia.
- Uma opção melhor é dividir seus dados em dois conjuntos: o conjunto de treinamento e o conjunto de teste. Como esses nomes sugerem, você treina seu modelo usando o conjunto de treinamento e o testa usando o conjunto de teste. A taxa de erro em novos casos é chamada de erro de generalização (ou erro fora da amostra), e ao avaliar seu modelo no conjunto de teste, você obtém uma estimativa desse erro. Esse valor indica o quão bem seu modelo irá performar em instâncias que nunca viu antes.
- É comum usar 80% dos dados para treinamento e reservar 20% para teste. No entanto, isso depende do tamanho do conjunto de dados: se ele contiver 10 milhões de instâncias, reservar 1% significa que seu conjunto de teste conterá 100.000 instâncias, provavelmente mais do que suficiente para obter uma boa estimativa do erro de generalização.

Por que devemos realizar o Split em conjuntos de treinamento, validação e teste antes de qualquer pré-processamento ?

- **Considere os tipos de processamento:** feature scaling, inputing de valores missing e encoding

Prevenir Vazamento de Dados (Data Leakage):

- Se aplicarmos técnicas de pré-processamento, sobre todo o conjunto de dados antes de dividir em conjuntos de treinamento e teste, você pode 'vazar' informações do conjunto de teste para o conjunto de treinamento. Isso pode superestimar a performance do modelo.

Avaliação mais Realista:

- Ao separarmos os dados antes do pré-processamento, simulamos um cenário mais realista. O modelo não terá acesso às informações do conjunto de teste durante o treinamento, o que ajuda a avaliar melhor sua capacidade de generalização para novos dados.

Evitar Viés no Pré-processamento:

- Pré-processar todo o conjunto de dados antes da divisão pode introduzir um viés nos dados de teste, pois estamos influenciando a distribuição dos dados com base em informações do conjunto de treinamento. Dividir antes do pré-processamento ajuda a evitar esse viés

Desta forma, qualquer processamentos entre instâncias deve ser feita após a divisão dos dados entre o conjunto de treinamento e o conjunto de teste, usando apenas os dados do conjunto de treinamento. Isso ocorre porque o conjunto de testes desempenha o papel de dados novos e invisíveis, portanto, não deveria estar acessível no estágio de treinamento. Usar qualquer informação proveniente do conjunto de testes antes ou durante o treinamento é um potencial viés na avaliação do desempenho.

Exemplos de processamento:

1 - Inputing de valores missing

- Se **usarmos a média do conjunto de dados completo (treinamento + teste) para preencher os valores ausentes, esteremos introduzindo informações dos dados de teste no processo de treinamento**, o que pode levar a um viés nos resultados. O conjunto de teste deve ser mantido como uma representação pura de dados invisíveis ao modelo durante o treinamento. Desta forma, é importante usar apenas estatísticas calculadas a partir do conjunto de treinamento, por exemplo, calculamos a media apenas do conjunto de treinamento para imputar valores ausentes tanto nos dados de treino quanto de teste.
- Também podemos destacar que em produção ao nos depararmos com valores faltantes, também devemos seguir a mesma analogia. Os valores faltantes serão preenchidos com base nas informasções obtidadas dos dados de treino.

2 - Encoding

- Durante o processo de encoding, especialmente em problemas de aprendizado supervisionado, é comum experimentar várias técnicas para representar os dados de forma mais eficaz:
  - Por exemplo, o **target encoding** envolve substituir as categorias de uma variável categórica pela média da variável de destino (ou outra estatística, como a mediana) para cada categoria. Desta forma, ao usar o target encoding, é importante lembrar de calcular as estatísticas (como a média da variável de destino) com base nos dados de treinamento e, em seguida, aplicar esses cálculos aos dados de teste.
  - A mesma premissa vale para **codificação de frequência**, a codificação de frequência envolve substituir cada categoria por sua contagem de ocorrências no conjunto de dados. Se essa contagem for calculada utilizando todo o conjunto de dados, incluindo os dados de teste, isso pode resultar em vazamento de dados, já que o modelo pode aprender informações sobre o conjunto de teste durante o treinamento.
  - Para outras técnicas como **codificação dummy** que envolve transformar as variáveis categóricas em variáveis binárias. Se criarmos essas variáveis dummy usando informações de todo o conjunto de dados, incluindo tanto os dados de treinamento quanto os de teste, as variáveis dummy serão criadas com base nas categorias presentes em todo o conjunto de dados. Por exemplo, suponha que no conjunto de teste haja uma categoria que não esteja presente no conjunto de treinamento. Se criarmos a variável dummy para essa categoria com base em todo o conjunto de dados, você estará introduzindo informações dos dados de teste no processo de treinamento, o que é indesejado. Além disso, é fundamental que o modelo seja capaz de lidar com categorias que ele nunca viu durante o treinamento, pois isso é uma situação realista que pode ocorrer em produção.

3 - Feature scaling

- Exemplos:

Scaling

- A fórmula para o Min-Max Scaling é dada por:

$$X_{\text{norm}} = \frac{X - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}}$$

- X é o valor original.
- Xmin é o valor mínimo no conjunto de dados.
- Xmax é o valor máximo no conjunto de dados.
- Xnorm é o valor normalizado.
  - Se calcularmos os valores mínimos e máximos usando todo o conjunto de dados (treinamento + teste) para normalizar os dados, estaremos introduzindo informações do conjunto de teste no cálculo desses valores. Isso pode levar a uma situação em que o modelo de machine learning tenha acesso a informações do conjunto de teste durante o treinamento, o que é um vazamento de informações indesejado.

Normalização

- A fórmula para a Z-score Normalization é dada por:

$$X_{\text{norm}} = \frac{X - \mu}{\sigma}$$

- X é o valor original.
- μ é a média do conjunto de dados.
- σ é o desvio padrão do conjunto de dados.
- Xnorm é o valor normalizado.
  - Se calcularmos a média e o desvio padrão usando todo o conjunto de dados (treinamento + teste) para normalizar os dados, estaremos

Valores Ausentes

- É recomendável preencher valores ausentes apenas ao treinar um modelo de Machine Learning. Para análises exploratórias, é preferível estudar a natureza dos dados nulos e compreender se há algum padrão subjacente, uma prática também fundamental durante a etapa de modelagem.
- Excluir observações pode enviesar o modelo, portanto, não é recomendável fazer isso indiscriminadamente, pois pode resultar em viés e perda de informações valiosas. Como então lidar com dados ausentes? Se uma coluna apresenta muitos dados ausentes, pode ser apropriado removê-la, especialmente se ela não contribuir significativamente para a análise.
- O limite para essa exclusão pode variar conforme o contexto; em geral, pode-se estabelecer um limite de cerca de 30%. Para os dados restantes, recomenda-se realizar imputação, mas somente após a divisão dos dados em conjuntos de treinamento e teste, evitando-se a imputação antes dessa divisão.
- **Tratar dados ausentes nos dados de teste: Se métodos de imputação foram aplicados aos dados de treinamento para lidar com dados ausentes, os mesmos métodos devem ser aplicados aos dados de teste para manter a coerência na análise.**
- Desta forma, ao aplicar transformações consistentes nos dados de teste: Ao preparar os dados de teste, é essencial aplicar as mesmas transformações e tratamentos que foram aplicados aos dados de treinamento, garantindo a consistência e a validade dos resultados.
- Para variáveis categóricas, valores ausentes podem ser substituídos pela categoria mais frequente.
- Embaixo está um exemplo de código para tratamento de valores ausentes:

In [ ]:
```
# Criando um pipeline com imputação e escalonamento
pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='mean')),  # Imputando usando a média
])

# Aprender e aplicar transformações nos dados de treino
X_train_transformed = pipeline.fit_transform(X_train) # 'fit_transform'

# Aplicar as mesmas transformações aos dados de teste
X_test_transformed = pipeline.transform(X_test) # apenas 'transform', desta forma, não vazamos dados para o teste
```

Remoção de Outliers

- Quanto aos outliers, se decidir removê-los, isso deve ser feito apenas nos dados de treinamento. Outliers nunca devem ser removidos antes de dividir os dados, pois isso pode comprometer a capacidade preditiva do modelo.
- Não remover antes da divisão dos dados (split): É crucial não remover outliers antes de dividir os dados em conjuntos de treinamento e teste, pois isso pode resultar na validação do modelo com dados que não representam a realidade.
- Validação com dados não representativos: Se os outliers forem removidos antes do split, a validação do modelo será feita com dados que não refletem a complexidade do mundo real.

Verificação de Multicolinearidade em Machine Learning

- A multicolinearidade ocorre quando certas variáveis nos dados podem ser representadas como combinações lineares de outras variáveis.

Inferência vs. Predição:

- Quando estamos interessados em **inferência**, nosso objetivo é entender a relação entre as variáveis independentes (features) e a variável dependente (target). Nesse caso, é crucial validar as premissas, como a multicolinearidade, e considerar todas as etapas rigorosas da análise.
- Por outro lado, quando o foco é na **predição**, como em modelos de regressão XGBoost, o processo difere significativamente. Aqui, a ênfase está nos dados de teste, onde avaliamos o desempenho prático do modelo. A análise de multicolinearidade e coeficientes pode ser desnecessária para a capacidade preditiva do modelo.
- **Portanto, se o interesse está na capacidade preditiva do modelo em dados novos, a análise de multicolinearidade e coeficientes torna-se desnecessária.**

Impacto da Multicolinearidade:

- A multicolinearidade não altera o poder preditivo do modelo nos dados de treinamento, mas pode atrapalhar nossas estimativas de coeficientes.
- Na maioria das aplicações de Machine Learning, nos preocupamos mais com as previsões do modelo do que com os coeficientes em si.
- **Entretando, se duas features são altamente correlacionadas, elas fornecem informações redundantes. Por exemplo, se as features A e B são altamente correlacionadas, ter ambas no modelo é desnecessário, mas é importante destacar que isso não afeta a capacidade preditiva, que já está sendo validada na etapa de teste.**

Checagem de multicolinearidade para Inferência:

- **O Fator de Inflação da Variância (VIF)** é uma medida que quantifica o quanto a variância de um coeficiente estimado em uma regressão é aumentada devido à multicolinearidade entre as variáveis explicativas no modelo. Ele é calculado para cada variável independente no modelo e nos ajuda a entender se a presença de multicolinearidade pode estar influenciando negativamente a precisão das estimativas dos coeficientes.

Cálculo do VIF:

- Para cada variável independente Xi em um modelo de regressão com múltiplas variáveis independentes, o VIF é calculado seguindo estes passos:
  - Regressão de Xi sobre todas as outras variáveis independentes:
    - Considerando uma variável Xi realize uma regressão linear onde Xi é a variável dependente e todas as outras variáveis independentes no modelo são os preditores.

$$X_i = \beta_0 + \beta_1 X_1 + \cdots + \beta_{i-1} X_{i-1} + \beta_{i+1} X_{i+1} + \cdots + \beta_k X_k + \epsilon$$

- - Determinar o R² dessa regressão:
    - Calcule o coeficiente de determinação (R²) para essa regressão. o R² é uma medida de quão bem as variações de Xi podem ser explicadas pelas outras variáveis independentes no modelo.
  - Calcular o VIF:
    - O VIF é então calculado usando a fórmula:

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

Interpretação do VIF:

- VIF = 1: Não há multicolinearidade entre Xi e as outras variáveis.
- 1 < VIF < 5: Multicolinearidade moderada, geralmente não é preocupante.
- VIF \geq 5: Multicolinearidade pode ser problemática (alguns usam um limite mais conservador de 10).
- Um VIF alto indica que a variável independente Xi pode ser linearmente prevista por outras variáveis independentes no modelo com um alto grau de precisão. Isso pode levar a estimativas de regressão que são instáveis ou imprecisas.

Exemplo:

$$\text{VIF}_{X_1} = \frac{1}{1 - 0.90} = 10$$

- Um VIF de 10 sugere que 90% da variação em Xi pode ser explicada pelas outras variáveis no modelo, indicando uma alta multicolinearidade.

VIF X Correlação de Pearson

- A correlação de Pearson é calculada apenas entre dois conjuntos de variáveis por vez. Isso significa que ela pode perder complexidades quando três ou mais variáveis estão inter-relacionadas de maneiras que não são diretamente visíveis em uma análise par-a-par.

- **A Correlação de Pearson não consegue capturar casos onde uma variável é uma combinação linear de múltiplas outras variáveis**. Por exemplo, se uma variável é igual à soma de outras três variáveis, a correlação de Pearson entre cada par individual pode não ser suficientemente alta para indicar um problema, mas a multicolinearidade ainda existe.
- **Ao contrário da correlação de Pearson, o VIF considera a relação de uma variável com todas as outras variáveis no modelo simultaneamente**. Isso o torna muito útil para detectar se uma variável pode ser linearmente prevista com precisão por outras variáveis, um sinal de multicolinearidade que pode afetar a interpretação e a precisão dos resultados de um modelo de regressão para fins de inferência.

Classes desbalanceadas

- O desequilíbrio de classes ocorre quando há uma discrepância significativa no número de amostras entre as classes em um conjunto de dados, resultando em previsões tendenciosas. Considere, por exemplo, um cenário de classificação binária onde a classe majoritária possui 10.000 amostras e a classe minoritária apenas 100. Isso resulta em uma proporção de 100:1, indicando que para cada 100 amostras da classe majoritária, existe apenas uma da classe minoritária. Esse fenômeno é conhecido como desequilíbrio de classes.
- Por que é essencial lidar com o desequilíbrio de classes?
  - A maioria dos algoritmos de aprendizado de máquina parte do pressuposto de que as classes são igualmente representadas nos dados. Quando enfrentamos desequilíbrio de classes, os modelos tendem a ser enviesados, favorecendo a previsão da classe majoritária devido à sua maior frequência. Com isso, a classe minoritária, que geralmente pode ser mais crítica, não é adequadamente representada, levando a um aprendizado insuficiente dos padrões que a caracterizam.
- **Uma abordagem eficaz para mitigar esse problema é o ajuste estratégico dos pesos de classe.** Ao atribuir pesos maiores à classe minoritária, incentivamos o modelo a prestar mais atenção aos padrões menos frequentes, equilibrando assim o tratamento entre as classes. Esse método não só ajuda a melhorar a precisão da classificação para a classe minoritária, mas também contribui para reduzir o viés do modelo em favor da classe majoritária.
- **Em resumo, ao desenvolver modelos de machine learning, é essencial que eles sejam capazes de generalizar para novos dados. Uma prática comum nesse processo é o balanceamento de classes, mas é importante realizar isso somente nos dados de treino. Isso porque o conjunto de teste deve refletir as condições reais que o modelo encontrará após ser implantado, onde frequentemente as classes estão desbalanceadas.**

In [ ]:
```python
#Atribuindo pesos
class_weights = compute_class_weight('balanced', classes=np.unique(y_train), y=y_train)
weights_dict = dict(zip(np.unique(y_train), class_weights))
weights_dict

pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', LogisticRegression(class_weight=weights_dict))
])
```

sklearn.utils.class_weight é usada para calcular os pesos para as classes de modo a equilibrar o conjunto de dados. Ela é frequentemente utilizada quando temos um desequilíbrio significativo entre as classes em um conjunto de treinamento.
O **parâmetro 'balanced'** instrui a função a ajustar os pesos inversamente proporcionais às frequências das classes no dado de entrada. Ou seja, classes minoritárias recebem pesos maiores para garantir que o modelo não seja enviesado em favor das classes majoritárias.

Redução de dimensionalidade

- **A redução de dimensionalidade é uma técnica utilizada em machine learning para reduzir o número de variáveis aleatórias sob consideração, obtendo um conjunto de variáveis principais. Existem duas abordagens principais para a redução de dimensionalidade: seleção de variáveis (feature selection) e extração de variáveis (feature extraction). Vamos focar na seleção de variáveis, que é dividida em três métodos principais: métodos de filtro, métodos incorporados e métodos wrapper.**
- Aqui estão algumas razões pelas quais a seleção de variáveis é importante:
  - A redução no número de variáveis diminui a complexidade do modelo, tornando-o mais simples e fácil de interpretar. Modelos menos complexos são menos propensos a overfitting, onde o modelo se ajusta demais aos dados de treinamento e tem um desempenho ruim em novos dados.
  - A remoção de variáveis irrelevantes ou redundantes pode melhorar significativamente a performance do modelo, tanto em termos de precisão quanto em eficiência computacional. Variáveis irrelevantes podem adicionar ruído aos dados, prejudicando a capacidade do modelo de aprender padrões úteis.
  - Com menos variáveis para processar, o tempo necessário para treinar o modelo é reduzido. Isso é especialmente importante para conjuntos de dados grandes e modelos complexos, onde o tempo de treinamento pode ser significativo.
- **Quando estamos lidando com um dataset que possuí muitas features, é essencial adotar estratégias adequadas para seleção de features, pois a análise exploratória pode ser complexa e a simples observação de correlações pode não ser suficiente.**

Seleção de variáveis:

1 - Métodos de Filtro:

- Os métodos de filtro avaliam as características dos dados sem a necessidade de um modelo preditivo. Eles usam critérios estatísticos para selecionar variáveis relevantes.

1.1 - Limiar de Variância:

- A Variance Threshold (limiar de variância) é um método de filtro simples e eficaz utilizado na seleção de variáveis. Este método remove todas as variáveis cuja variância não atinge um determinado limiar
- A ideia por trás do método Variance Threshold é que uma variável com baixa variância não contribui significativamente para a diferenciação entre os dados. Por exemplo, se uma variável assume quase sempre o mesmo valor, ela provavelmente não será útil para distinguir entre diferentes classes ou prever uma variável de interesse.
- **Ou seja, features com variância zero ou muito baixa não possuem valor preditivo. Isso ocorre porque essas features não oferecem informação adicional que ajude a diferenciar as classes ou fazer predições.**

- Exemplo: Previsão de Aposentadoria de Funcionários:

Idade	Anos de Serviço	Salário Anual	Departamento	Aposentadoria
65	30	90000	Financeiro	1
60	25	85000	Financeiro	0
70	40	100000	Financeiro	1
55	20	80000	Financeiro	0
67	35	95000	Financeiro	1
50	15	75000	Financeiro	0

- No exemplo acima, a feature 'Departamento' possui variância zero, pois todos os funcionários pertencem ao mesmo departamento ('Financeiro').
- Como o valor é constante, essa feature não tem poder preditivo para a variável alvo 'Aposentadoria'. Não ajuda a distinguir entre quem se aposentou e quem não se aposentou.
- Desta forma, para seleção de features, podemos adotar as seguintes estratégias:
  - **Análise Individual de Features:** Examinar cada feature isoladamente e remover aquelas com baixa variância, pois elas tendem a ter menor poder explicativo.

In [ ]:
```
from sklearn.feature_selection import VarianceThreshold

# Definindo o limiar de variância
selector = VarianceThreshold(threshold=0.5)

# Ajustando e transformando no conjunto de treinamento
X_train_reduced = selector.fit_transform(X_train)

# Transformando o conjunto de teste com os parâmetros ajustados no conjunto de treinamento
X_test_reduced = selector.transform(X_test)
```

1.2 - DropConstantFeatures

- O método DropConstantFeatures pode ser considerado uma variação específica dos métodos de filtro na seleção de variáveis. Mais especificamente, ele é uma forma simplificada do método Variance Threshold, focando em remover variáveis que são constantes (ou quase constantes) em todo o conjunto de dados
- **Exclusão de features constantes:** Features constantes não fornecem nenhuma informação útil para a modelagem preditiva, pois não variam entre as amostras e, portanto, não podem ajudar a distinguir entre diferentes classes ou valores da variável alvo.
- A classe DropConstantFeatures da biblioteca feature_engine é especificamente projetada para remover features constantes, ou seja, aquelas com variância zero.
- Enquanto o Variance Threshold remove variáveis com variância abaixo de um certo limiar, DropConstantFeatures especificamente remove variáveis com variância zero (ou próximo de zero)

In [ ]:
```python
from feature_engine.selection import DropConstantFeatures

# Instanciando o método DropConstantFeatures
dcf = DropConstantFeatures()

# Ajustando e transformando no conjunto de treinamento
X_train_reduced = dcf.fit_transform(X_train)

# Transformando o conjunto de teste com os parâmetros ajustados no conjunto de treinamento
X_test_reduced = dcf.transform(X_test)
```

1.3 - Correlação Par-a-Par e correlação com a variável alvo

- **Correlação entre Features:** Avaliar como as features se relacionam entre si e remover aquelas com alta correlação. Se duas features são altamente correlacionadas, elas fornecem informações redundantes. Por exemplo, se as features A e B são altamente correlacionadas, ter ambas no modelo é desnecessário.
  - **Relação com a Variável Alvo:** Observar como cada feature se relaciona com a variável alvo. Features que têm uma forte relação com a variável alvo devem ser mantidas, pois são úteis para a predição.

1.3.1 DropCorrelatedFeatures

- O DropCorrelatedFeatures avalia a correlação entre todas as variáveis no conjunto de dados e remove aquelas que têm uma correlação acima de um limiar especificado (threshold). Ele utiliza diferentes métodos de correlação, sendo o método de Pearson o mais comum.

In [ ]:
```python
# Instanciando o método DropCorrelatedFeatures
dcor = DropCorrelatedFeatures(method='pearson', threshold=0.8)

# Ajustando e transformando no conjunto de treinamento
X_train_reduced = dcor.fit_transform(X_train)

# Transformando o conjunto de teste com os parâmetros ajustados no conjunto de treinamento
X_test_reduced = dcor.transform(X_test)
```

1.3.2 SmartCorrelatedSelection

- O SmartCorrelatedSelection é outra ferramenta poderosa para seleção de variáveis, disponível na biblioteca Feature-engine. Este método vai além da simples remoção de variáveis altamente correlacionadas. Ele tenta selecionar as variáveis que são mais informativas, levando em conta a correlação entre elas e a variável alvo, além da correlação entre as variáveis independentes
- No SmartCorrelatedSelection da biblioteca Feature-engine, o parâmetro selection_method define o critério para selecionar a variável principal dentro de cada grupo de variáveis correlacionadas. Aqui estão os métodos de seleção disponíveis:
  - Variance (Variância): Este método mantém a variável com a maior variância dentro de cada grupo de variáveis correlacionadas.
  - Correlation with Target (Correlação com o Alvo):
  - Calcula a correlação (usando o método especificado, como Pearson) entre cada variável no grupo correlacionado e a variável alvo, mantendo aquela com a maior correlação com o Target.
  - Importance (Importância): Treina um modelo de aprendizado (como uma árvore de decisão) e usa as importâncias das variáveis calculadas pelo modelo para selecionar a variável mais importante dentro de cada grupo correlacionado.

In [ ]:
```python
from feature_engine.selection import SmartCorrelatedSelection

# Instanciando o método SmartCorrelatedSelection
scs = SmartCorrelatedSelection(
    method='pearson',
    threshold=0.8,
    selection_method='variance'  # ou 'correlation_with_target' se a variável alvo estiver disponível
)

# Ajustando e transformando no conjunto de treinamento
X_train_reduced = scs.fit_transform(X_train)

# Transformando o conjunto de teste com os parâmetros ajustados no conjunto de treinamento
X_test_reduced = scs.transform(X_test)
```

2 - Métodos Incorporados (Embedded Methods)

- Os métodos incorporados selecionam variáveis durante o processo de treinamento do modelo. Esses métodos consideram a importância das variáveis como parte integrante do algoritmo de aprendizado. Eles são mais lentos que os métodos de filtro, mas tendem a produzir melhores resultados.

2.1 - LASSO (Least Absolute Shrinkage and Selection Operator):

- A **penalização L1 (regularização L1 ou Lasso)** é especialmente útil para a seleção de features (variáveis) devido à sua capacidade única de forçar alguns coeficientes a zero, resultando em modelos esparsos. Isso é útil para seleção de características.
- Se o peso de uma feature não contribui muito para melhorar a precisão do modelo, a penalização L1 pode 'forçar' esse peso a ser zero. Quando o peso de uma feature é zero, essa feature é essencialmente removida do modelo.
- **Ao forçar alguns pesos a zero, a penalização L1 reduz o número de features que o modelo realmente usa. Isso torna o modelo mais simples e mais fácil de interpretar.**
- Os métodos incorporados realizam a seleção de features enquanto o modelo está sendo treinado. A penalização L1 faz exatamente isso: durante o processo de ajuste do modelo, ela decide quais features manter e quais remover.

In [ ]:
```python
from sklearn.linear_model import LogisticRegression

# Modelo de regressão logística com penalização L1
modelo = LogisticRegression(penalty='l1', solver='liblinear')
modelo.fit(X_train, y_train)
```

- Entretanto, a ideia de usar a regressão logística diretamente para a seleção de features (recursos) com base nos respectivos pesos ou coeficientes tem algumas suposições e limitações importantes.
- Quando usamos a regressão logística diretamente para a seleção de features com base nos coeficientes, estamos implicitamente assumindo que todas as features são independentes umas das outras. Essa suposição pode não ser verdadeira em muitos casos.
- A presença de multicolinearidade pode causar instabilidade nos coeficientes estimados, tornando difícil interpretar a importância individual das features.
- A importância de uma feature pode depender do valor de outra feature. A regressão logística tradicional pode não capturar essas interações complexas.

2.2 Árvores de Decisão (Decision Trees) e Florestas Aleatórias (Random Forests)

- Árvores de decisão são modelos de machine learning usados tanto para tarefas de classificação quanto de regressão. Elas dividem os dados em subconjuntos baseados nos valores das features, criando uma estrutura de árvore onde cada nó representa uma feature e cada ramo representa um valor ou intervalo de valores.
- Durante o treinamento, as árvores de decisão calculam a importância de cada feature com base em como ela melhora a pureza dos nós (por exemplo, usando a redução da impureza de Gini ou o ganho de informação). Features que mais contribuem para a separação dos dados recebem maior importância.
- Já as Florestas aleatórias são um conjunto (ensemble) de muitas árvores de decisão treinadas de maneira independente. Elas combinam as previsões de múltiplas árvores para melhorar a precisão e reduzir o overfitting.
- Random Forests calculam a importância das features de maneira similar às árvores de decisão, mas agregam as importâncias através de todas as árvores na floresta. Ao usar muitas árvores e amostragem aleatória, Random Forests fornecem uma estimativa mais robusta da importância das features, mitigando problemas de variância que podem afetar uma única árvore de decisão.

In [ ]:
```python
from sklearn.tree import DecisionTreeClassifier

# Treinando o modelo de Árvore de Decisão
model = DecisionTreeClassifier(random_state=42)
model.fit(X_train, y_train)

# Extraindo a importância das features
importances = model.feature_importances_
feature_importance = pd.DataFrame({'Feature': X_train.columns, 'Importance': importances})

# Ordenando por importância
feature_importance = feature_importance.sort_values(by='Importance', ascending=False)
feature_importance
```

In [ ]:
```python
from sklearn.ensemble import RandomForestClassifier

# Treinando o modelo de Random Forest
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# Extraindo a importância das features
importances = model.feature_importances_
feature_importance = pd.DataFrame({'Feature': X_train.columns, 'Importance': importances})

# Ordenando por importância
feature_importance = feature_importance.sort_values(by='Importance', ascending=False)
feature_importance
```

- Ressalvas da Importância das Features em Random Forests

- A importância das features baseada em impureza, como a usada em Random Forests, pode ser inflada para features categóricas que têm muitos valores únicos. Isso acontece porque essas features podem facilmente dividir os dados em muitos pequenos grupos, reduzindo a impureza (Gini ou entropia) de maneira significativa, mesmo que a feature não seja realmente informativa.
- Quando duas features são altamente correlacionadas, ambas podem estar fornecendo informações muito similares para o modelo. Em uma Random Forest, a importância baseada em impureza será distribuída entre essas features correlacionadas. Isso demonstra como a importância das features pode ser influenciada pela correlação entre elas, Porém, se uma dessas features for removida, a importância que estava sendo compartilhada entre as duas pode ser atribuída integralmente à feature restante, dependendo do contexto e da natureza dos dados.

3 - Métodos Wrapper (Wrapper Methods)

- Os métodos wrapper utilizam um modelo de machine learning para avaliar a combinação de variáveis e selecionam aquelas que melhoram o desempenho do modelo. Esses métodos são computacionalmente mais caros, mas tendem a ser mais precisos na seleção de variáveis relevantes.

3.1 Eliminação Recursiva - Recursive Feature Elimination (RFE)

- Recursive Feature Elimination (RFE) é uma técnica de seleção de features que se encaixa na categoria de métodos wrapper. Ela é usada para selecionar um subconjunto de features mais relevantes, eliminando as menos importantes de maneira recursiva.
- Processo de Funcionamento:
  - Primeiro, um modelo é ajustado ao conjunto de dados completo.
  - As features são ordenadas de acordo com sua importância no modelo.
  - A feature com menor importância é removida.
  - O modelo é treinado novamente com as features restantes.
  - O desempenho do novo modelo é calculado.
  - Se a performance não cair significativamente, a feature é removida.
  - Caso contrário, ela é mantida.
  - Os passos são repetidos recursivamente até que o número desejado de features seja alcançado. A cada iteração, o modelo é ajustado novamente ao conjunto de dados reduzido e a feature menos importante é eliminada.

- O RFE ajuda a identificar e manter apenas as features que são mais relevantes para o modelo.
- Ao reduzir o número de features, o modelo resultante tende a ser mais simples e fácil de interpretar.
- Eliminando features irrelevantes ou redundantes, o RFE pode ajudar a melhorar a generalização do modelo para novos dados.

3.1.1 RecursiveFeatureElimination da biblioteca feature_engine

- Permite escolher um modelo base, como o RandomForestClassifier.
- A eliminação das features é baseada em um limiar de aumento na pontuação de uma métrica especificada (por exemplo, precisão).
- threshold: Define o limiar para a diferença de desempenho que é aceitável para a remoção de uma feature. Se a remoção de uma feature causa uma diminuição na métrica de avaliação menor do que este limiar, a feature é removida.
  - Exemplo: Se utilizarmos um limiar (threshold) de 0.01, e o desempenho do modelo cair apenas 0.005 ao remover uma feature (de 0.69 para 0.685, por exemplo), a feature seria removida.
- scoring: Métrica para avaliar a performance do modelo durante a seleção das features.
- cv: Número de folds na validação cruzada para avaliar a performance.

In [ ]:
```
from feature_engine.selection import RecursiveFeatureElimination

rfe = RecursiveFeatureElimination(
    RandomForestClassifier(random_state=42),
    cv=3,
    threshold=0.01,  # limiar
    scoring='precision'  # métrica avaliada
)

X_train_rfe = rfe.fit_transform(X_train, y_train)
X_test_rfe = rfe.transform(X_test)
```

3.1.2 Recursive Feature Elimination da biblioteca scikit-learn

- O RFE do scikit-learn permite escolher um modelo base (como regressão logística, SVM, etc.) para ajustar e selecionar features.
- A eliminação das features é baseada nos coeficientes do modelo ou na importância das features.
- n_features_to_select: Número de features a serem selecionadas.
- step = quantidade de features a serem eliminadas por vez em cada iteração.

In [ ]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.feature_selection import RFE

# Criando uma instância do modelo de regressão logística
lr = LogisticRegression(solver='liblinear', random_state=123)

# Criando uma instância do RFE
rfe = RFE(estimator=lr, n_features_to_select=5, step=1)
```

```
# Ajustando e transformando os dados de treinamento
X_train_sub = rfe.fit_transform(X_train, y_train)
```

3.1.3 RFE como parte de um Pipeline

- Pode ser uma boa prática usar um modelo mais simples para seleção de variáveis e um modelo mais complexo para a modelagem final

In [ ]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import make_pipeline

pipe = make_pipeline(RFE(estimator=lr, step=1),
                     KNeighborsClassifier())

parameters = {'rfe__n_features_to_select': range(1, 13), # Número de features a serem selecionadas pelo RFE, 1 a 12.
              'kneighborsclassifier__n_neighbors': range(1, 10)}

grid = GridSearchCV(pipe, param_grid=parameters, cv=10, n_jobs=-1)
grid.fit(X_train, y_train)

print('Best params:', grid.best_params_)
print('Best accuracy:', grid.best_score_)
```

- Prós e Contras da Eliminação Recursiva de Features (Recursive Feature Elimination):
  - Pode selecionar explicitamente o número de features.
  - Não é super caro (se for usado um modelo linear).
  - Leva em conta a interação entre as features.
- Contras:
  - Assume separabilidade linear (se for usado um modelo linear).
  - Não otimiza diretamente a métrica de performance.
  - Necessita de um método de busca para encontrar um bom número de features.

3.2 Adição Recursiva

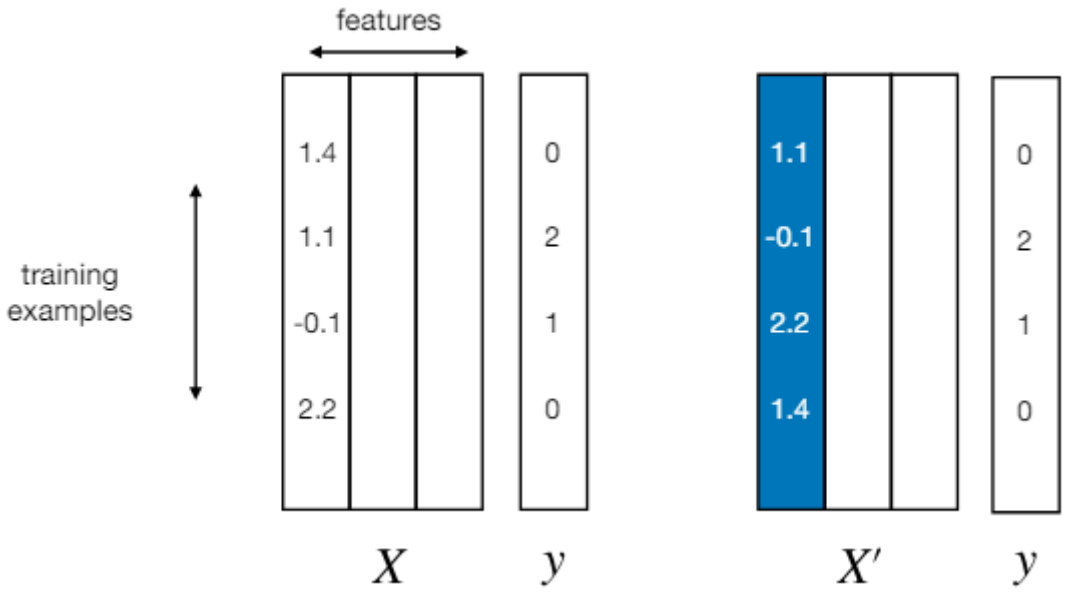
- A Adição Recursiva de Features (Recursive Feature Addition, RFA) é uma técnica de seleção de features que, ao contrário da eliminação recursiva, começa com um conjunto vazio de features e adiciona iterativamente as features que mais melhoram a performance do modelo, até que um critério de parada seja atingido.
  - Treina um modelo usando todas as features para obter a importância de cada uma.
  - Ordena as features de acordo com sua importância obtida pelo estimador.
  - Começa treinando um modelo com a feature mais importante e calcula o desempenho.
  - Adiciona a segunda feature mais importante e treina um novo modelo.
  - Calcula a diferença de desempenho entre os modelos.
  - Se o desempenho aumentar além do limiar determinado (threshold), a feature é mantida.
  - Caso contrário, a feature não é adicionada.

In [ ]:
```
from feature_engine.selection import RecursiveFeatureAddition

rfa = RecursiveFeatureAddition(
    RandomForestClassifier(random_state=42),
    cv=5,
    threshold=0.01,  # quanto precisa aumentar no score
    scoring='roc_auc'  # métrica avaliada
)

X_train_rfa = rfa.fit_transform(X_train, y_train)
X_test_rfa = rfa.transform(X_test)
```

- **threshold:** Define o limiar de aumento no desempenho necessário para adicionar uma feature.
  - Exemplo: Treine o modelo com a feature mais importante e obtenha uma métrica de avaliação inicial. Digamos que o modelo com essa feature tenha uma AUC de 0.71. Adicione uma nova feature e reavalie o modelo. Por exemplo, se a AUC aumentar de 0.71 para 0.713, o aumento é de 0.003. Como o aumento de 0.003 é menor que o limiar de 0.01, a feature não será adicionada.

3.3 Permutation Importance

- A importância por permutação é uma técnica para avaliar a importância de cada feature em um modelo. Ela mede a mudança no desempenho do modelo quando os valores de uma feature específica são embaralhados. Se a performance do modelo cair significativamente ao embaralhar uma feature, isso indica que a feature é importante.
- Para cada coluna de feature, embaralhamos aleatoriamente os valores dessa coluna.
- Isso quebra a associação entre a feature e o target, simulando a ausência dessa feature.



- Avaliamos a performance do modelo com a feature embaralhada.
- Comparamos a performance com a performance original do modelo (sem a feature embaralhada).
- A diferença na performance indica a importância da feature. Maior diferença significa maior importância.

- **A importância por permutação frequentemente fornece resultados semelhantes aos da importância baseada em impureza do Random Forest, mas é independente do modelo.**
- A importância por permutação é uma técnica que mede a relevância de cada feature em termos de impacto na performance do modelo, mas não seleciona ativamente ou elimina features. Em vez disso, fornece uma indicação de quais features são mais importantes para o modelo.
- Assim como na importância baseada em impureza do Random Forest, a importância das features pode ser subestimada se duas features forem altamente correlacionadas.

In [ ]:
```python
from sklearn.inspection import permutation_importance

# Criação e Treinamento do Modelo
forest = RandomForestClassifier(n_estimators=100,random_state=0)
forest.fit(X_train, y_train)

# Calculando a importância da permutação para cada variável no modelo treinado.
result = permutation_importance(estimator=forest,
                                X=X_test,
                                y=y_test,
                                scoring='accuracy',
                                n_repeats=50, # número de repetições para calcular importância da permutação p/ cada variável
                                random_state=0)


# Visualização dos resultados
for i in result.importances_mean.argsort()[::-1]:
    print(f'Feature: {X_test.columns[i]} ''
          f'Importância Média: {result.importances_mean[i]:.3f}'
          f'Desvio Padrão: {result.importances_std[i]:.3f}')
```

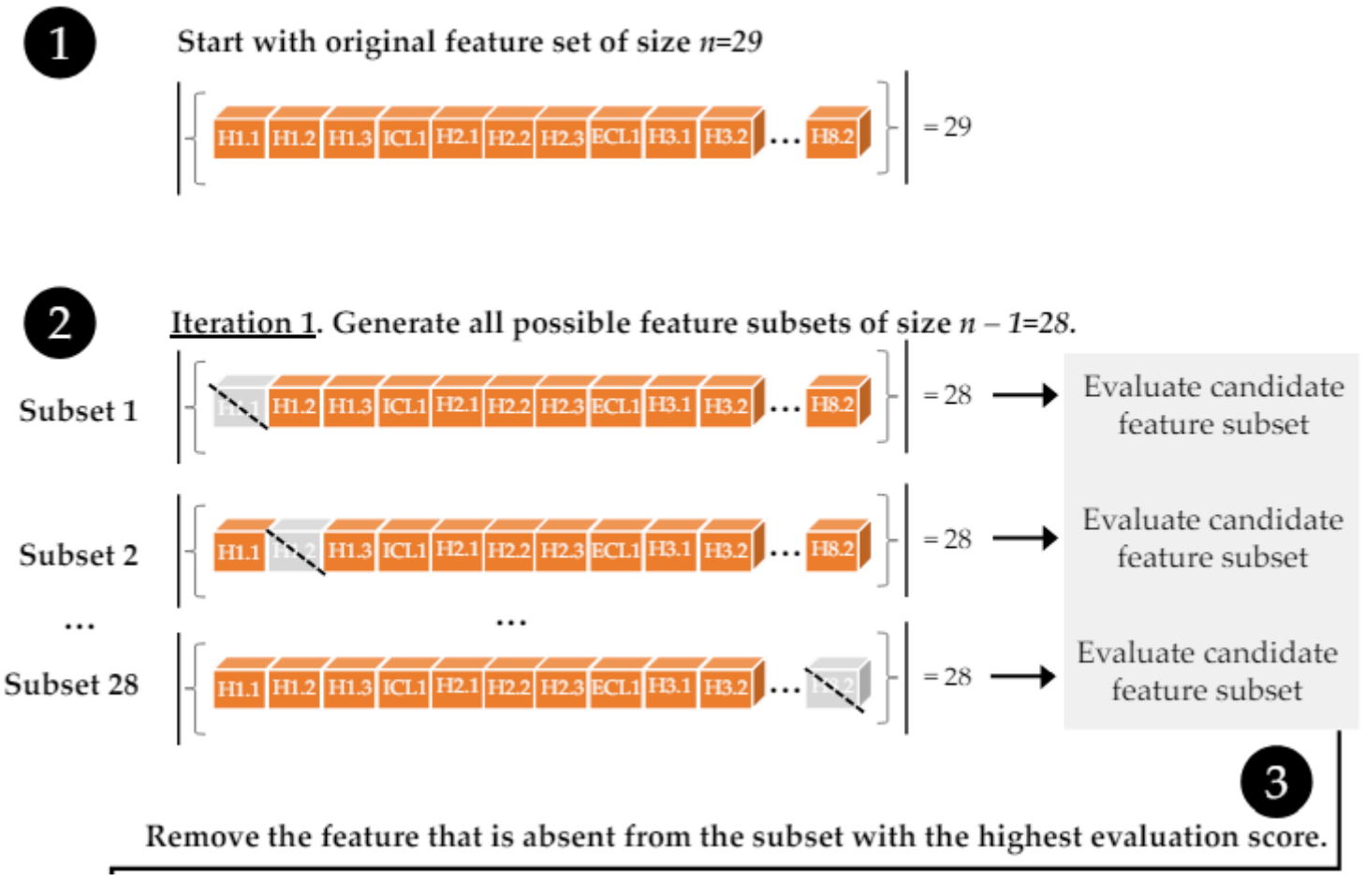
- A importância por permutação pode ter problemas com features correlacionadas, similar à importância baseada em impureza no Random Forest. Quando duas features são altamente correlacionadas, a importância de cada uma pode ser subestimada, pois permutar uma delas não altera muito a performance do modelo, já que a outra feature correlacionada ainda está presente.

3.4 Seleção Sequencial de Recursos (Sequential Feature Selection, SFS)

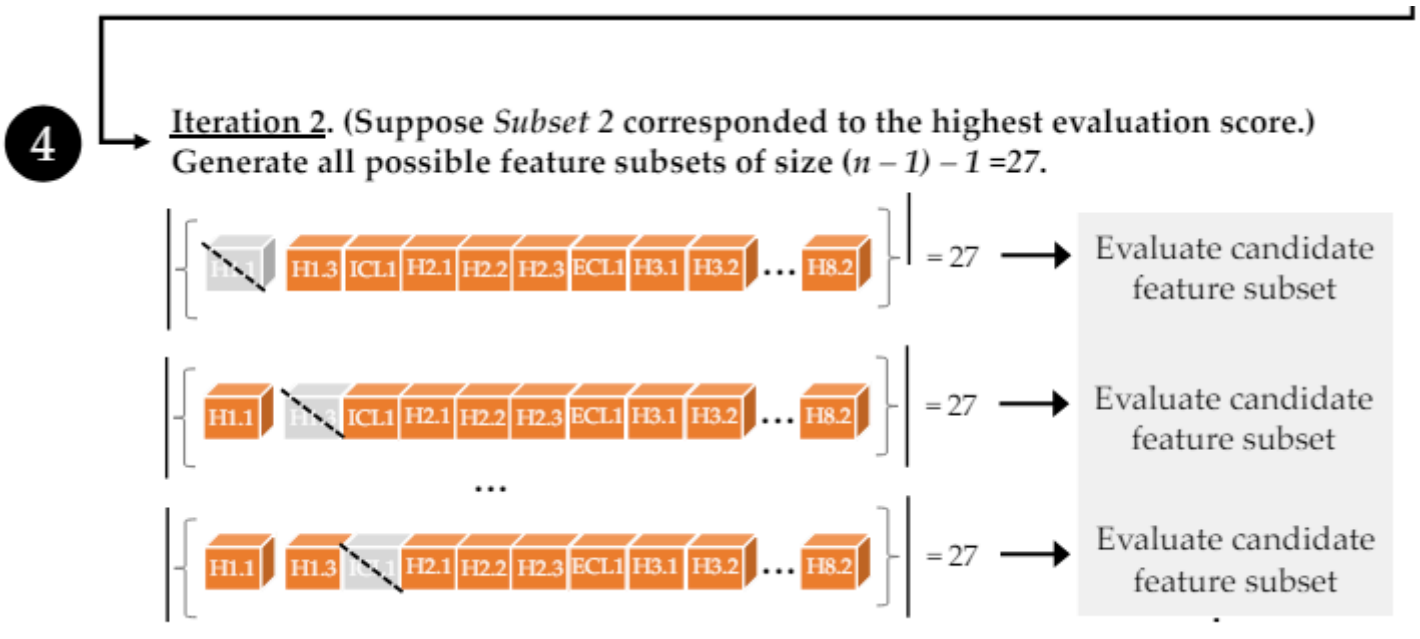
- A Seleção Sequencial de Recursos (Sequential Feature Selection, SFS) é uma técnica de seleção de features que envolve a adição ou remoção sequencial de features de um modelo com base em um critério de performance. Existem duas abordagens principais para a seleção sequencial de features:

3.4.1 Seleção Sequencial Para Trás (Sequential Backward Selection, SBS):

- Começa com todas as features disponíveis e remove sequencialmente as features que menos impactam a performance do modelo até que um critério de parada seja atingido.
  - Inicialização: Começa com o conjunto completo de features: Inicialmente, temos todas as features disponíveis no nosso conjunto de dados. Por exemplo, se tivermos 29 features, começamos com todas essas 29 features.
    - Primeira Iteração: Em cada iteração, removemos temporariamente uma feature do conjunto atual de features e avaliamos o modelo usando as features restantes.
    - Repetição para Todas as Features: Repetimos esse processo para cada feature, removendo uma de cada vez e avaliando a performance do modelo com as 28 features restantes
    - **Seleção da Melhor Remoção: Identificamos qual remoção de feature resulta na menor diminuição (ou maior melhoria) na performance do modelo. A feature cuja remoção tem o menor impacto negativo ou o maior impacto positivo é permanentemente removida do conjunto de features.**



- Segunda Iteração:
  - **Novo Conjunto de Features: Agora, começamos com 28 features (uma já foi removida na primeira iteração).**
  - Remoção de Features: Repetimos o processo de remoção e avaliação, removemos a primeira feature e avaliamos o modelo com as 27 features restantes.
  - Avaliação da Performance: Medimos a performance do modelo sem cada uma das features restantes.
  - Repetição para Todas as Features: Repetimos esse processo para todas as features restantes.



- Os passos 3 e 4 são repetidos até que o subconjunto de features contenha apenas uma única feature.
- **Considerando todas as iterações de 1 até n−1, o subconjunto com a maior pontuação de avaliação é selecionado como o subconjunto final de features. Em caso de empate, o menor subconjunto de features é selecionado.**

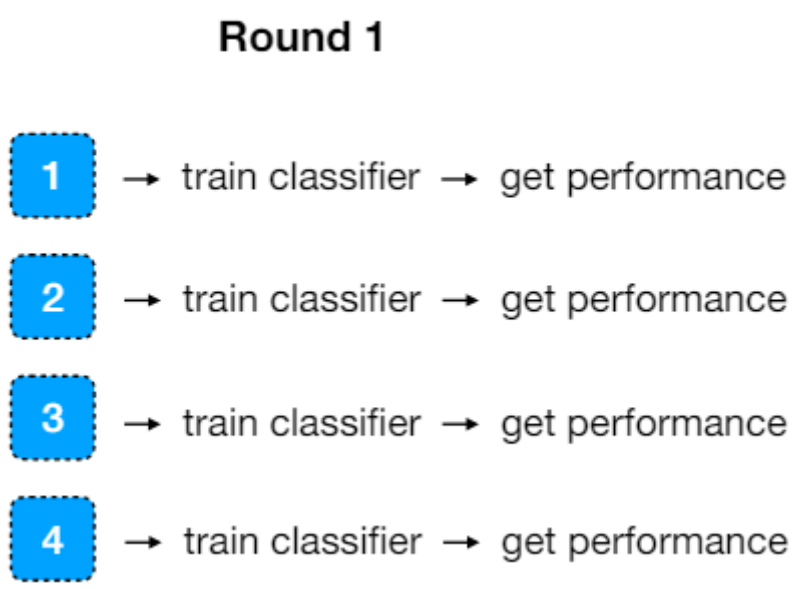
In [ ]:
```
from sklearn.feature_selection import SequentialFeatureSelector as SFS

model = KNeighborsClassifier(n_neighbors=5)

sfs = SFS(model,
          n_features_to_select=5,
          direction='backward',
          scoring='accuracy',
          n_jobs=-1,
          cv=5)

# Ajustar o SequentialFeatureSelector aos dados de treinamento e transformar os dados
X_train_sfs = sfs.fit_transform(X_train, y_train)

# Transformar os dados de teste utilizando as features selecionadas
X_test_sfs = sfs.transform(X_test)
```

- **Resumo das Diferenças entre Sequential Backward Selection e Eliminação Recursiva de Features:**
  - Sequential Backward Selection: Remove a feature que, ao ser removida, resulta na menor diminuição da performance.
  - RFE: Remove a feature com a menor importância segundo o modelo preditivo treinado.
  - Sequential Backward Selection: Avalia diretamente a performance do modelo após a remoção de cada feature.
  - RFE: Utiliza a importância das features determinada por um modelo preditivo.
  - Sequential Backward Selection: Pode ser computacionalmente intensivo, especialmente com grandes conjuntos de dados.

- RFE: Geralmente mais eficiente, pois utiliza a importância das features calculada pelo modelo.

3.4.2 Seleção Sequencial Para Frente (Sequential Forward Selection, SFS):

- Começa com um modelo vazio e adiciona sequencialmente as features que mais melhoram a performance do modelo até que um critério de parada seja atingido:
  - Inicialização: Começa com um conjunto vazio de features.
  - Primeira iteração: Treinamos o classificador com cada feature individualmente e medimos a performance do modelo.



- Seleção da Melhor Feature:
  - Comparando as performances, selecionamos a feature que resultou na melhor performance do modelo.
  - Suponhamos que a Feature 2 apresentou a melhor performance.
  - Adicionamos a Feature 2 ao conjunto de features selecionadas.
- Segunda Iteração:
  - Com a Feature 2 já selecionada, agora treinamos o classificador adicionando cada uma das outras features ao conjunto atual (Feature 2)

e avaliamos a performance.

- **O processo continua até que todas as features tenham sido adicionadas ou até que um critério de parada predefinido seja alcançado (por exemplo, um número desejado de features ou uma performance mínima).**

- **Diferenças entre Adição Recursiva de Features (Recursive Feature Addition, RFA) e Seleção Sequencial Para Frente (Sequential Forward Selection, SFS)**
  - Adição Recursiva de Features (RFA): RFA: Usa a importância das features fornecida pelo modelo inicial para guiar o processo de adição. A ordem de adição é baseada na importância das features.
  - Adição Recursiva de Features (RFA): Critério de Adição: A feature é adicionada ao conjunto de features selecionadas se a adição resultar em uma melhoria na performance que exceda um limiar predefinido (threshold).
  - Seleção Sequencial Para Frente (SFS): Começa com um conjunto vazio de features. Em cada iteração, adiciona-se uma feature de cada vez ao conjunto atual de features. A feature que resulta na maior melhoria da performance do modelo é mantida no conjunto de features selecionadas.
  - Seleção Sequencial Para Frente (SFS): Critério de Adição: A feature que melhora mais a performance do modelo é selecionada e adicionada ao conjunto atual de features.

3.5 Seleção por desempenho da variável

- A Seleção por Desempenho da Variável é uma abordagem de seleção de features onde o desempenho de cada feature é avaliado individualmente usando um modelo preditivo. A ideia é treinar vários modelos diferentes, cada um usando apenas uma única feature do dataset, e então avaliar o desempenho de cada modelo para determinar a utilidade de cada feature.

    - Um modelo é treinado usando cada feature individualmente. Por exemplo, se o conjunto de dados tiver 10 features, 10 modelos diferentes serão treinados, cada um utilizando apenas uma das features.
    - O desempenho de cada modelo é avaliado utilizando uma métrica específica.
- Desvantagens:
    - Não considera possíveis interações entre features. Uma feature que é fraca individualmente pode ser muito útil em combinação com outras features.

In [ ]:
```
from sklearn.linear_model import LinearRegression
from feature_engine.selection import SelectBySingleFeaturePerformance

# Selecionamos utilizando uma regressão linear
sel = SelectBySingleFeaturePerformance(
    estimator=LinearRegression(),
    scoring='r2', # Critério de avaliação
    cv=3,
    threshold=0.01 #limiar de desempenho, features que resultam em um R² abaixo desse valor serão excluídas
)

X_train_sel = sel.fit_transform(X_train, y_train)
X_test_sel = sel.transform(X_test)
```

Três Etapas para Seleção de Features

- Uma abordagem promissora poderia envolver três etapas:
- Remoção de Features com baixa Variância
- Remoção de Features Correlacionadas
- Eliminação Recursiva de Features

Encoder

- O encoding das variáveis é uma etapa essencial no processo de preparação de dados para machine learning.
- **A maioria dos algoritmos de machine learning requerem dados numéricos para funcionar. Variáveis categóricas em seu estado bruto não podem ser diretamente usadas nesses modelos. O encoding converte essas categorias em um formato numérico que os algoritmos podem processar.**
- Encoding adequado garante que a informação contida nas variáveis categóricas seja preservada e utilizada de forma eficaz

1 - Label Encoding

- Label Encoding (codificação de rótulos) é uma técnica de codificação para lidar com variáveis categóricas. Nesta técnica, a cada rótulo é atribuído um número inteiro exclusivo com base na ordem alfabética.
- Na codificação de rótulos em Python, substituímos o valor categórico por um valor numérico entre 0 e o número de classes menos 1. Se o valor da variável categórica contiver 5 classes distintas usamos (0, 1, 2, 3 e 4).
- Os dados são modificados, mas sem perder a informação que eles representam. Isso pode ser feito de maneira manual (usando dicionários em Python) ou com o algoritmo LabelEncoder do pacote Scikit-Learn.
- **O LabelEncoder do scikit-learn é projetado para ser usado especificamente na variável alvo (target) de tarefas de classificação, não nas características de entrada (features) que formam o conjunto de dados, neste caso, conseguimos aplicá-lo apenas em uma única coluna por vez.**

In [ ]:
```python
from sklearn.preprocessing import LabelEncoder

# Inicializando o LabelEncoder
label_encoder = LabelEncoder()

# Ajustando o encoder e transformando a variável target de treino
y_train_encoded = label_encoder.fit_transform(y_train)

# Transformando a variável target de teste com o mesmo encoder
y_test_encoded = label_encoder.transform(y_test)
```

2 - Ordinal Encoding

- O OrdinalEncoder transforma cada valor categórico em um número inteiro. Ao contrário do LabelEncoder que é normalmente usado na variável alvo e trata cada valor de forma independente, o OrdinalEncoder pode ser aplicado a várias colunas ao mesmo tempo e é frequentemente usado nas características de entrada.
- Você deve usar o OrdinalEncoder quando as categorias têm uma ordem natural, pois o modelo de aprendizado de máquina interpretará que as categorias representadas por números maiores são de alguma forma "maiores" ou "mais" que aquelas representadas por números menores.

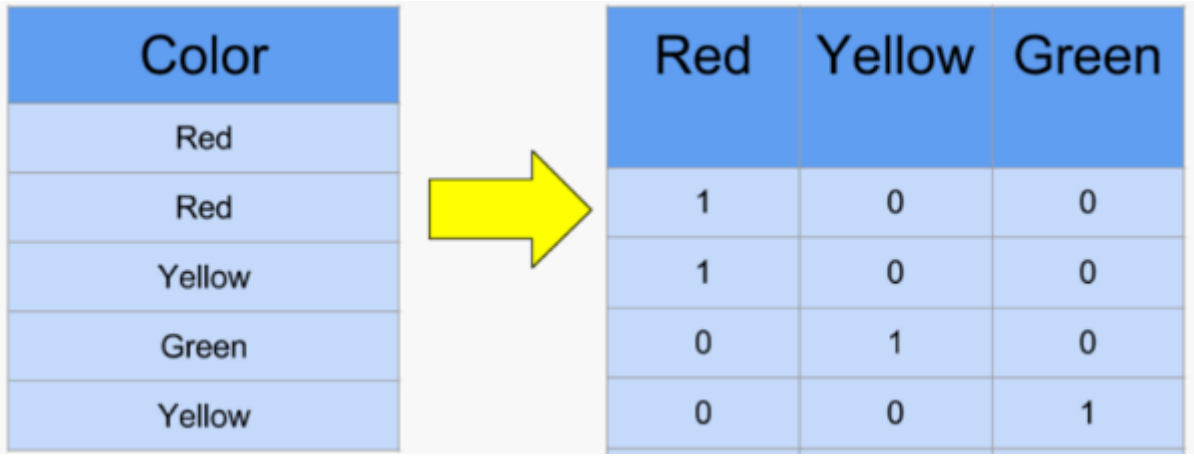
Original Encoding	Ordinal Encoding
Poor	1
Good	2
Very Good	3
Excellent	4

In [ ]:
```python
from sklearn.preprocessing import OrdinalEncoder

# Definindo a ordem das categorias para o OrdinalEncoder
categories_order = [
    ['fundamental', 'médio', 'superior'],  # Ordem para nível educação
]

# Criando o pipeline
pipeline = Pipeline([
    ('ordinal_encoder', OrdinalEncoder(categories=categories_order))  # Codificador ordinal
])

# Aprender e aplicar transformações nos dados de treino
X_train_transformed = pipeline.fit_transform(X_train)

# Aplicar as mesmas transformações aos dados de teste
X_test_transformed = pipeline.transform(X_test)
```

3 - One-Hot Encoding

- One-Hot Encoding é outra técnica popular para tratar variáveis categóricas. Ele simplesmente cria recursos adicionais com base no número de valores exclusivos no recurso categórico.
- Cada valor exclusivo na categoria será adicionado como um recurso(uma nova variável).

- Nessa abordagem, para cada categoria de um recurso, criamos uma nova coluna (às vezes chamada de variável fictícia) com codificação binária (0 ou 1) para indicar se uma determinada linha pertence a essa categoria.
- Vamos considerar a imagem abaixo. Observe que a variável Color possui 3 categorias (Red, Yellow e Green). Aplicando One-Hot Encoding 3 novas variáveis são criadas, sendo o valor 1 quando a ocorrência daquela cor e 0 quando não há ocorrência.



- One-Hot Encoding é o processo de criação de variáveis fictícias(dummy variables).
- Por padrão, a saída de um OneHotEncoder é uma matriz esparsa. Uma matriz esparsa é uma representação para matrizes que contêm principalmente zeros
- Quando um atributo categórico tem centenas de categorias, codificá-lo com one-hot resulta em uma matriz muito grande, cheia de 0s, exceto por um único 1 por linha. Nesse caso, uma matriz esparsa é exatamente o que você precisa: ela economiza muita memória e acelera os cálculos.
- **Uma potencial desvantagem desse método é um aumento significativo na dimensionalidade do conjunto de dados (que é chamado de Curse of Dimensionality.**

Por que evitamos codificação one-hot em árvores de decisão?

- Quando codificamos uma variável categórica usando one-hot encoding, cada categoria se torna uma nova variável binária (dummy variable). Por exemplo, se tivermos uma variável com 4 categorias (A, B, C, D), ela se transformará em 4 variáveis binárias:
- A: [1, 0, 0, 0]
- B: [0, 1, 0, 0]
- C: [0, 0, 1, 0]
- D: [0, 0, 0, 1]

- **Impacto na Crescimento da Árvore:** No exemplo, acima cada categoria tem uma chance de 25% de aparecer e a medida que aumentamos o número de categorias esta proporção diminuí mais ainda, desta forma, a proporção de valores 1 em relação aos valores 0 é pequena, especialmente quando o número de categorias aumenta. Isso significa que a maioria dos valores em cada variável dummy será 0. Essa distribuição desigual pode afetar a seleção de splits (divisões) na árvore de decisão, pois o algoritmo de divisão (como Gini ou entropia) pode não achar essas variáveis muito informativas.

- **Distância do Nó Raiz:** Como as variáveis dummies são independentes entre si, uma divisão em uma dessas variáveis geralmente não resulta em um ganho significativo de pureza. Por isso, é improvável que o algoritmo escolha essas variáveis para dividir próximo ao nó raiz. As árvores tendem a crescer em direções que minimizam a impureza, e variáveis dummies podem não proporcionar essa minimização significativa, resultando em splits que ocorrem mais longe do nó raiz. **Desta forma, Árvores de decisão podem ter dificuldade em encontrar splits significativos em variáveis dummies, resultando em splits mais afastados do nó raiz.**
  - Divisões próximas do nó raiz são responsáveis por grandes reduções na impureza (como Gini ou entropia)
  - Variáveis que causam divisões perto do nó raiz são geralmente mais importantes.
- **Impacto na Impureza:** A impureza em uma árvore de decisão é medida pela heterogeneidade dos dados nos nós. Em uma variável dummy, a maioria dos valores será 0, o que pode levar a uma divisão que não reduz suficientemente a impureza. Por exemplo, se temos um nó com muitas variáveis dummies, a divisão baseada em uma delas não criará nós filhos significativamente mais puros, pois a maioria dos dados continuará sendo 0 em outras variáveis.**A presença predominante de valores 0 em variáveis dummies faz com que as divisões resultem em ganhos marginais de pureza, impactando negativamente a eficiência da árvore.**

Esses fatores juntos podem fazer com que a árvore de decisão subestime a importância de variáveis codificadas em one-hot, mesmo que essas variáveis sejam relevantes para o modelo e assim a árvore de decisão tende a crescer longe dessas variáveis quando usamos a codificação one-hot.

In [ ]:
```
from sklearn.preprocessing import OneHotEncoder

# Criando o pipeline
pipeline = Pipeline([
    ('onehot_encoder', OneHotEncoder())  # Codificador One-Hot
])

# Aprender e aplicar transformações nos dados de treino
X_train_transformed = pipeline.fit_transform(X_train)

# Aplicar as mesmas transformações aos dados de teste
X_test_transformed = pipeline.transform(X_test)
```

4 - Frequency Encoding

- O Frequency Encoding é uma técnica de codificação de variáveis categóricas onde cada categoria é substituída pela sua frequência (número de vezes que aparece) ou pelo seu percentual de ocorrências no conjunto de dados. Esta abordagem pode ser particularmente útil em situações onde a frequência de categorias específicas está correlacionada com o alvo que se deseja prever, pois captura informações sobre a prevalência de cada categoria
- o contrário de técnicas como One-Hot Encoding, o Frequency Encoding não aumenta o número de características, pois cada categoria é substituída por um único número.
- **O Scikit-Learn não possui uma função padrão para aplicar Frequency Encoding diretamente, por isso a necessidade de criar uma classe personalizada.**

Numerical value	Animal
1.5	cat
3.6	cat
42	dog
7.1	crocodile

Frequency encoding →

Numerical value	Animal_freq
1.5	0.5
3.6	0.5
42	0.25
7.1	0.25

---

5 - Target Encoding

- O Target Encoding é uma técnica de codificação de variáveis categóricas onde cada categoria é substituída por uma medida que reflete o efeito que a categoria tem sobre a variável alvo. Isso é feito calculando o valor esperado da variável alvo dentro de cada categoria. As especificidades dependem do tipo de problema:

Regressão:

- As categorias são substituídas pelo valor médio da variável alvo para essa categoria.

$$E_i = \frac{1}{n_i} \sum_{k=1}^{n_i} Y_k$$

- Ei é o valor a ser substituído para a categoria i.
- Ni é o número de exemplos na categoria i.
- Yk é o valor da variável alvo para o exemplo K na categoria i.

- Vamos supor que temos os seguintes dados de treinamento:

Bairro	Preço do Aluguel
Centro	2000
Centro	2200
Norte	1800
Sul	1900
Norte	1700
Centro	2100

- Para o bairro Centro: 2000+2200+2100/3 = 2100

- Para o bairro Norte: 1800+1700/2 = 1750
- Para o bairro Sul: 1900

- Agora, substituímos cada categoria pela média dos valores alvo correspondentes:

Bairro	Preço do Aluguel Codificado
Centro	2100
Centro	2100
Norte	1750
Sul	1900
Norte	1750
Centro	2100

- Neste ponto, podemos usar esses valores codificados como entradas para treinar nosso modelo de regressão, que aprenderá a prever o preço do aluguel com base nos bairros. Durante a fase de previsão, quando um novo bairro é apresentado ao modelo, ele substituirá o bairro pelo valor correspondente na codificação de destino, ou seja, a média dos preços do aluguel para esse bairro. Isso ajudará o modelo a fazer previsões mais precisas com base nas informações disponíveis.
- **Embora isso possa nos dar uma ideia de como funciona a codificação de destino, usar esse método de maneira simples pode introduzir o risco de overfitting, especialmente para categorias raras e como nesses casos a codificação alvo fornecerá essencialmente o valor alvo ao modelo há o risco de vazamento de dados. Além disso, o método acima só pode lidar com categorias vistas, portanto, se seus dados de teste tiverem uma nova categoria, ele não será capaz de lidar com ela. Para evitar esses erros, é necessário tornar o transformador de codificação de destino mais robusto.**

Classificação:

- As categorias são substituídas pela proporção de casos positivos daquela categoria:
- Em primeiro lugar, precisamos medir a frequência das categorias para as classes-alvo positivas e negativas:

Cliente	Cor	Target
1	Azul	Sim
2	Vermelho	Não
3	Azul	Sim
4	Verde	Não
5	Vermelho	Não
6	Azul	Não
7	Azul	Sim
8	Verde	Sim
9	Azul	Não
10	Vermelho	Sim

- Temos um total de 6 azuis e 3 deles pertencem a classe positiva. Portanto, precisamos considerar a fração da classe positiva para esta categoria específica. Então, para "Azul", a média é 3/5 = 0.6
- Para "Vermelho", a média da classe positiva é 1/3 ≈ 0.333;
- Para "Verde", a média da classe positiva é 1/2 = 0.5
- Então, substituímos cada categoria pelo valor da proporção correspondente à classe positiva:

Cliente	Cor
1	0.6
2	0.333
3	0.6
4	0.5
5	0.333
6	0.6
7	0.6
8	0.5
9	0.6
10	0.333

In [ ]:
```python
from category_encoders import TargetEncoder

# Criando o pipeline
pipeline = Pipeline([
    ('target_encoder', TargetEncoder())  # Codificador Target
])

# Aprender e aplicar transformações nos dados de treino
X_train_transformed = pipeline.fit_transform(X_train, y_train)

# Aplicar as mesmas transformações aos dados de teste
X_test_transformed = pipeline.transform(X_test)
```

6 - Bayesian Target Encoding

- O Bayesian Target Encoding é uma forma de codificação de alvo que combina a média global com a média específica da categoria, ponderando de acordo com o número de observações na categoria, especialmente útil quando há categorias com poucos dados, pois para categorias com poucas observações, essa média pode ser altamente variável e propensa a overfitting.
- **Codificação de Alvo Bayesiana:**
  - Introduz um parâmetro de suavização (α) que controla a influência da média global em comparação com a média específica da categoria.
  - Essa suavização reduz a variância das categorias com poucas observações, tornando o modelo mais robusto e menos propenso a overfitting.

$$\text{Codificação}_c = \frac{N_c \cdot \mu_c + \alpha \cdot \mu}{N_c + \alpha}$$

- Fórmula:
- Onde:
  - Nc é o número de amostras na categoria c.
  - μc é a média específica da categoria c.
  - μ é a média global do alvo.
  - α é o parâmetro de suavização

- **Vamos demonstrar o cálculo com a tabela anterior:**
  - Calcular a Média Global: μ = 5/10 = 0.5 (5 'sim' em 10)
  - Calcular a Média Específica da Categoria:
    - Azul: 3/5 = 0.6 (3 'sim' em 5)
    - Vermelho: 1/3 = 0.333 (1 'sim' em 3)
    - Verde: 1/2 = 0.5(1 'sim' em 2)
  - Usando um parâmetro de suavição α igual a 2:
    - Azul: (5 *0.6* + *2* 0.5) / 5 + 2 = 0.57
    - Vermelho: (3 *0.333* + *2* 0.5) / 3 + 2 = 0.4
    - Verde: (2 *0.5* + *2* 0.5) / 2 + 2 = 0.5

Resultados:

Cliente	Cor	Target	Codificação de Alvo Bayesiana
1	Azul	Sim	0.57
2	Vermelho	Não	0.4
3	Azul	Sim	0.57
4	Verde	Não	0.5
5	Vermelho	Não	0.4
6	Azul	Não	0.57
7	Azul	Sim	0.57
8	Verde	Sim	0.5
9	Azul	Não	0.57
10	Vermelho	Sim	0.4

In [ ]:
```
from category_encoders import TargetEncoder

# Criando o pipeline com TargetEncoder configurado para Bayesian Target Encoding
pipeline = Pipeline([
    ('target_encoder', TargetEncoder(smooth='auto'))
]) # O parâmetro smooth ajusta a suavização Bayesiana, corresponde ao nosso parâmetro alfa (a)

# Aprender e aplicar transformações nos dados de treino
X_train_transformed = pipeline.fit_transform(X_train, y_train)

# Aplicar as mesmas transformações aos dados de teste
X_test_transformed = pipeline.transform(X_test)
```

- Os valores do parâmetro alpha (ou smooth no TargetEncoder) dependem da quantidade de suavização desejada e das características do seu conjunto de dados.
  - Quando smooth é definido como "auto", o valor de suavização é definido como uma estimativa Bayesiana empírica. Isso significa que a biblioteca estima automaticamente um valor adequado para a suavização com base nos dados disponíveis.
  - Quando smooth é definido como um valor de ponto flutuante, você pode especificar diretamente a quantidade de suavização. Valores maiores resultam em maior peso na média global do alvo.
    - Pequeno α: Quando você tem muitas amostras por categoria (grande Nc), um valor pequeno de α pode ser suficiente. Isso faz com que a média da categoria μc tenha mais influência.
    - Grande α: Quando você tem poucas amostras por categoria (pequeno Nc), um valor maior de α é desejável para evitar overfitting. Isso faz com que a média global μ tenha mais influência.
- Mais informações em: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.TargetEncoder.html

7 - K-fold Target Encoding

- K-fold target encoding é uma técnica utilizada para transformar variáveis categóricas em variáveis numéricas com base na média dos valores-alvo (target) da variável de interesse.
- **O K-fold target encoding ajuda a prevenir o data leakage (vazamento de dados) através da separação cuidadosa das amostras de treino e validação durante o processo de encoding.**
- Processo:
  - O conjunto de dados é dividido em K partes (ou folds) aproximadamente iguais. Essa divisão é feita de forma aleatória e cada fold será utilizado como conjunto de validação uma vez.
  - Para cada fold, calcula-se a média dos valores-alvo (target) para cada categoria da variável categórica, mas utilizando apenas os dados que estão fora do fold atual. Isso significa que, para o fold i, a média do target para a categoria c é calculada usando apenas os dados presentes nos outros K−1 folds.
  - Por exemplo, se tivermos 5 folds, ao calcular a média do target para a categoria A no fold 1, utilizamos os dados dos folds 2, 3, 4 e 5.

$$\text{Codificação}_c = \frac{N_c \cdot \mu_c + \alpha \cdot \mu}{N_c + \alpha}$$

  - Utilizamos a mesma fórmula da medía ponderada:
- Vamos exemplificar com a mesma tabela usada anteriormente.
- Vamos assumir que α=1 para simplificar os cálculos.
- Vamos dividir apenas em 2 Folds, devido ao número pequeno de amostras:
  - **Fold 1 (Clientes 1, 2, 3, 4, 5), usamos a média dos clientes fora do Fold 1:Clientes 6, 7, 8, 9, 10:**
    - Azul: (Não, Sim, Não), Média = (0+1+0)/3 = 0.33
    - Vermelho: (Sim) -> (1) -> Média = 1.00
    - Verde: (Sim) -> (1) -> Média = 1.00

- Média global (fora do Fold 1): Clientes 6, 7, 8, 9, 10 (Não, Sim, Sim, Não, Sim) -> (0, 1, 1, 0, 1)/5 = 0.6
  - Usando um parâmetro de suavição α igual a 1:
  - Azul: (3 *0.33* + *1* 0.6) / 3 + 1 = 0.40
  - Vermelho: (1 * 1.0 + 1 0.60) / 1 + 1 = 0.80
  - Verde: (1 * 1.0 + 1 0.60) / 1 + 1 = 0.80
  - Estas médias serão substituídas no Fold 1.
- **Fold 2 (Clientes 6, 7, 8, 9, 10), usamos a média dos clientes fora do Fold 2:Clientes 1, 2, 3, 4, 5:**
  - Azul: (Sim, Sim), Média = (1+1)/2 = 1
    - Vermelho: (Não, Não) -> (0 + 0)/2 -> Média = 0
    - Verde: (Não) -> (0) -> Média = 0
    - Média global (fora do Fold 1): Clientes 1, 2, 3, 4, 5 (Sim, Não, Sim, Não, Não) -> (1, 0, 1, 0, 0)/5 = 0.4
- Usando um parâmetro de suavição α igual a 1:
  - Azul: (2 *1* + *1* 0.4) / 2 + 1 = 0.8
  - Vermelho: (2 *0* + *1* 0.4) / 2 + 1 = 0.133
  - Verde: (1 *0* + *1* 0.4) / 1 + 1 = 0.20
- Estas médias serão substituídas no Fold 2.

Resultados:

Cliente	Cor	Target	Target Encoding
1	Azul	Sim	0.40
2	Vermelho	Não	0.80
3	Azul	Sim	0.40
4	Verde	Não	0.80
5	Vermelho	Não	0.80
6	Azul	Não	0.80
7	Azul	Sim	0.80
8	Verde	Sim	0.20
9	Azul	Não	0.80
10	Vermelho	Sim	0.13

No scikit-learn, não existe uma implementação direta para K-fold Target Encoding.

8 - CatBoost Encoder

- O CatBoost Encoder é uma técnica de codificação de variáveis categóricas usada em algoritmos de aprendizado de máquina. Foi desenvolvida como parte do framework CatBoost. **A codificação CatBoost é uma forma de Target Encoding, mas com melhorias para evitar overfitting e vazamento de dados (data leakage).**
- Um artigo detalhado pode ser encontrado em: (https://readmedium.com/en/https:/towardsdatascience.com/how-catboost-encodes-categorical-variables-3866fb2ae640)
- Vamos utilizar a mesma tabela usada anteriormente como exemplo:

Cliente	Cor	Target
1	Azul	Sim
2	Vermelho	Não
3	Azul	Sim
4	Verde	Não
5	Vermelho	Não
6	Azul	Não
7	Azul	Sim
8	Verde	Sim
9	Azul	Não
10	Vermelho	Sim

- Vamos detalhar como os valores de codificação são calculados para cada categoria

$$ctr = \frac{curCount \ + \ prior}{maxCount \ + \ 1}$$

- Considere a seguinte equação:
  - curCount: Número total de objetos no conjunto de dados de treinamento com o valor do recurso categórico atual.
  - maxCount: Número de objetos no conjunto de dados de treinamento com o valor de recurso mais frequente
  - prior: Um valor constante que influencia a estimativa do CTR para categorias com poucas ocorrências. Ele pode ser visto como um 'valor de suavização'. Definido como 0.05 na documentação do Catboost (https://catboost.ai/en/docs/concepts/algorithm-main-stages_cat-to-numberic)
- **Cada linha é tratada de maneira indepedente para esta codificação, a linha atual não é levada em conta para codificação, portanto começamos em zero para cada recurso:**
  - **Primeira linha:'Azul':** 0 + 0.05 / 0 + 1 = 0.05
    - 0: Não vimos nenhum recurso 'Azul' como 1 no Target até esta linha
    - 0.05: Constante
    - 0: Não Há linhas anteriores de 'Azul'
  - **Segunda linha 'Vermelha':** 0 + 0.05 / 0 + 1 = 0.05
    - 0: Não vimos nenhum recurso 'Vermelho' como 1 no Target até esta linha
    - 0.05: Constante
    - 0: Não Há linhas anteriores de 'Vermelho'
  - **Terceira linha 'Azul':** 1 + 0.05 / 1 + 1 = 0.525
    - 1: Vimos um recurso 'Azul' com 1 no Target até esta linha
    - 0.05: Constante
    - 1: Há uma linha 'Azul' anteriormente
      - **Quarta linha 'Verde':** 0 + 0.05 / 0 + 1 = 0.05
    - 0: Não vimos nenhum recurso 'Verde' com 1 no Target até esta linha
    - 0.05: Constante
    - 0: Não Há linhas anteriores de 'Verde'
  - **Quinta linha 'Vermelha':** 0 + 0.05 / 1 + 1 = 0.025
    - 0: Não vimos nenhum recurso 'Vermelho' com 1 no Target até esta linha
    - 0.05: Constante
    - 0: Há uma linha 'Vermelha' anteriormente
      - **Sexta linha 'Azul':** 2 + 0.05 / 2 + 1 = 0.683
    - 2: Vimos dois recursos 'Azul' com 1 no Target até esta linha
    - 0.05: Constante
    - 2: Há dois linhas 'Azul' anterior
      - **Sétima linha 'Azul':** 2 + 0.05 / 3 + 1 = 0.5125
    - 2: Vimos dois recursos 'Azul' com 1 no Target até esta linha
    - 0.05: Constante
    - 2: Há três linhas 'Azul' anteriormente
  - **Oitava linha 'Verde':** 0 + 0.05 / 1 + 1 = 0.025
    - 0: Não vimos nenhum recurso 'Verde' com 1 no Target até esta linha
    - 0.05: Constante
    - 0: Há duas linhas anteriores de 'Verde'
      - **Nona linha 'Azul':** 3 + 0.05 / 4 + 1 = 0.61
    - 2: Vimos três recursos 'Azul' com 1 no Target até esta linha
    - 0.05: Constante
    - 2: Há 4 linhas 'Azul' anteriormente
      - **Décima linha 'Vermelha':** 0 + 0.05 / 2 + 1 = 0.0166
    - 0: Não vimos nenhum recurso 'Vermelho' com 1 no Target até esta linha
    - 0.05: Constante
    - 0: Há duas linhas 'Vermelha' anteriormente

Tabela de Resultados:

Linha	Cor	Target	curCount	maxCount	CTR
1	Azul	Sim	0	0	0.05
2	Vermelho	Não	0	0	0.05
3	Azul	Sim	1	1	0.525
4	Verde	Não	0	0	0.05
5	Vermelho	Não	0	1	0.025
6	Azul	Não	2	2	0.683
7	Azul	Sim	2	3	0.5125
8	Verde	Sim	0	1	0.025
9	Azul	Não	3	4	0.61
10	Vermelho	Sim	0	2	0.0166

In [ ]:
```python
from category_encoders import CatBoostEncoder

# Criando o pipeline
pipeline = Pipeline([
    ('catboost_encoder', CatBoostEncoder())  # Codificador CatBoost
])

# Aprender e aplicar transformações nos dados de treino
X_train_transformed = pipeline.fit_transform(X_train, y_train)

# Aplicar as mesmas transformações aos dados de teste
X_test_transformed = pipeline.transform(X_test)
```

9 - Codificação WoE

- A codificação WoE (Weight of Evidence) é uma técnica utilizada principalmente em modelos de regressão logística e em modelos de pontuação de crédito e análise de risco. Ela transforma variáveis categóricas em variáveis contínuas ao atribuir um valor de peso de evidência para cada categoria. Este método é especialmente útil para problemas de classificação binária.

$$\text{WoE} = \ln\left(\frac{\text{Proporção de Bons Resultados}}{\text{Proporção de Maus Resultados}}\right)$$

- A codificação WoE utiliza a seguinte fórmula para transformar cada categoria:
- Onde:
- Proporção de Bons Resultados: é a % de bons resultados (por exemplo, não inadimplentes em pontuação de crédito) em uma determinada categoria.
- Proporção de Maus Resultados: é a % de maus resultados (por exemplo, inadimplentes em pontuação de crédito) na mesma categoria.
- ln: logaritmo natural

Interpretação do WoE

- WoE Positivo: Significa que a Distribuição de Bons é maior do que a Distribuição de Maus.
  - Como o logaritmo natural de um número maior que 1 é positivo, se a proporção de bons resultados é maior que a de maus resultados, o WoE será positivo.
- WoE Negativo: Significa que a Distribuição de Bons é menor do que a Distribuição de Maus.
  - Como o logaritmo natural de um número menor que 1 é negativo, se a proporção de bons resultados é menor que a de maus resultados, o WoE será negativo.
- Exemplo de cálculo, suponha que temos uma variável categórica 'Estado Civil' com as categorias 'Solteiro', 'Casado' e 'Divorciado'. E temos uma variável dependente binária 'Inadimplente' (0 = Não, 1 = Sim).

Estado Civil	Bons (0)	Maus (1)	Total
Solteiro	100	50	150
Casado	300	30	330
Divorciado	150	20	170

- Total de Bons (Não Inadimplentes): 100 + 300 + 150 = 550
- Total de Maus (Inadimplentes): 50 + 30 + 20 = 100

- Calculando a % de eventos (maus) e % de não eventos (bons) em cada grupo:
  - Para 'Solteiro':
    - % de Bons (não eventos) = 100/550 = 0.1818

        - % de Maus (eventos) = 50/100 = 0.5
    - Para 'Casado':
        - % de Bons (não eventos) = 300/550 = 0.5455
        - % de Maus (eventos) = 30/100 = 0.3
            - Para 'Divorciado':
        - % de Bons (não eventos) = 150/550 = 0.2727
        - % de Maus (eventos) = 20/100 = 0.2
- Calculando WoE tomando o logaritmo natural da divisão de % de não eventos e % de eventos:
    - Para 'Solteiro': WoE= ln(0.1818/0.5) = ln(0.3636) = −1.0116
    - Para 'Casado': WoE= ln(0.5455/0.3) = ln(1.8183) = 0.5983
    - Para 'Divorciado': WoE= ln(0.2727/0.2) =ln(1.3635)= 0.3102

Resultados:

Estado Civil	% Bons	% Maus	WoE
Solteiro	0.1818	0.5	-1.0116
Casado	0.5455	0.3	0.5983
Divorciado	0.2727	0.2	0.3102

- Solteiro: Tem um WoE negativo, indicando que ser solteiro está associado a uma maior probabilidade de inadimplência.
- Casado: Tem um WoE positivo, sugerindo que ser casado está associado a uma menor probabilidade de inadimplência.
- Divorciado: Também tem um WoE positivo, mas menor que o dos casados, indicando uma menor probabilidade de inadimplência em comparação aos solteiros, mas maior que a dos casados.

Dimensionamento de recursos (feature scaling)

- Muitos algoritmos de aprendizado de máquina funcionam melhor quando as variáveis de entrada são dimensionadas para um intervalo padrão. Esse processo de dimensionamento ou mudança de escala é chamado de Feature Scaling (dimensionamento de recursos).
- Isso inclui algoritmos que usam uma soma ponderada da entrada, como regressão linear, e algoritmos que usam medidas de distância, como k-vizinhos mais próximos (KNN).
- As duas técnicas mais populares para dimensionar dados numéricos antes da modelagem são **Scaling e normalização.**
- O dimensionamento de recursos é usado para evitar que os modelos de aprendizado supervisionados sejam tendenciosos em relação a um intervalo específico de valores. Por exemplo, se seu modelo é baseado em regressão linear e você não dimensiona recursos, alguns recursos podem ter um impacto maior do que outros, o que afetará o desempenho das previsões, dando vantagem indevida a algumas variáveis sobre outras. Isso coloca certas classes em desvantagem durante o treinamento do modelo.
- É por isso que se torna importante usar algoritmos de dimensionamento para que você possa padronizar seus valores de recursos. Esse processo de dimensionamento de recursos é feito para que todos os recursos possam compartilhar a mesma escala e, portanto, evitar problemas como: perda de precisão e aumento no custo computacional à medida que os valores dos dados variam amplamente em diferentes ordens de magnitude. A ideia é transformar o valor dos recursos em um intervalo semelhante como outros para que os algoritmos de aprendizado de máquina se comportem melhor, resultando em modelos ideais.

1 - Normalização (Min-Max Scaling):

- A normalização, também conhecida como Min-Max Scaling, ajusta os valores de uma variável para um intervalo específico, geralmente entre 0 e 1.

A fórmula para o Min-Max Scaling é dada por:

$$X_{\text{norm}} = \frac{X - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}}$$

- X é o valor original.
- Xmin é o valor mínimo no conjunto de dados.
- Xmax é o valor máximo no conjunto de dados.
- Xnorm é o valor normalizado.

- A normalização é útil quando os dados não seguem uma distribuição gaussiana e quando a escala dos dados é desconhecida ou variável.

- **Ela é mais sensível a outliers, justamente por ajustar os valores extremos dentro do intervalo entre 0 e 1, podendo condensar os demais valores em um intervalo mais estreito.**
- O transformador MinMaxScaler possui um hiperparâmetro feature_range que permite alterar o intervalo se, por algum motivo, você não quiser que seja de 0 a 1 (por exemplo, redes neurais funcionam melhor com entradas de média zero, então um intervalo de -1 a 1 é preferível)

In [ ]:
```python
from sklearn.preprocessing import MinMaxScaler

# Criando o pipeline
pipeline = Pipeline([
    ('min_max_scaler', MinMaxScaler())  # Escala as features para o intervalo [0, 1]
])

# Aprender e aplicar transformações nos dados de treino
X_train_transformed = pipeline.fit_transform(X_train)

# Aplicar as mesmas transformações aos dados de teste
X_test_transformed = pipeline.transform(X_test)
```

2 - Padronização

- A padronização, ou Standard Scaling, ajusta os valores de uma variável para ter uma média de zero e um desvio padrão de 1.

A fórmula para a Z-score Normalization é dada por:

$$X_{\text{norm}} = \frac{X - \mu}{\sigma}$$

- X é o valor original.
- μ é a média do conjunto de dados.
- σ é o desvio padrão do conjunto de dados.
- Xnorm é o valor normalizado.

- Ao contrário da Normalização, a padronização mantém informações úteis sobre valores discrepantes e torna o algoritmo menos sensível a eles.

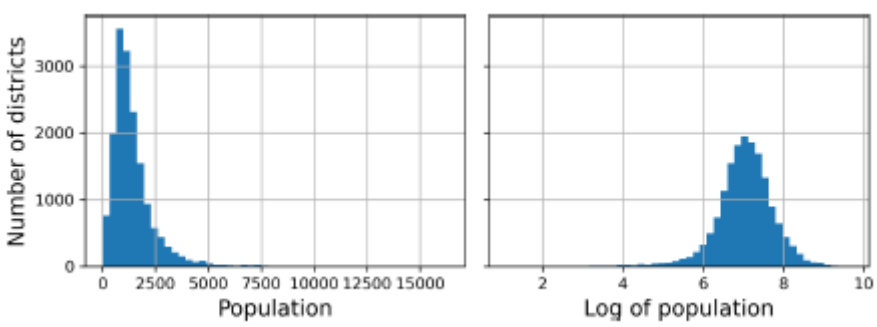
- **A padronização é mais robusta para dados com distribuição normal ou aproximadamente normal e é menos sensível a outliers do que a normalização.**
- Em resumo, a escolha entre normalização e padronização depende das características dos dados e das suposições do modelo. Se os dados não seguem uma distribuição normal e a escala dos dados é importante, a normalização pode ser preferível.
- No entanto, se a distribuição dos dados é normal ou aproximadamente normal, a padronização pode ser mais apropriada. É sempre recomendável experimentar ambas as técnicas e avaliar qual funciona melhor para o seu conjunto de dados e modelo

In [ ]:
```python
from sklearn.preprocessing import StandardScaler

# Criando o pipeline
pipeline = Pipeline([
    ('scaler', StandardScaler())  # Escalonador de features
])

# Aprender e aplicar transformações nos dados de treino
X_train_transformed = pipeline.fit_transform(X_train)

# Aplicar as mesmas transformações aos dados de teste
X_test_transformed = pipeline.transform(X_test)
```

- **Note que, enquanto os valores do conjunto de treinamento serão sempre escalonados para a faixa especificada, se novos dados contiverem valores atípicos, estes podem acabar escalonados fora da faixa. Se você quiser evitar isso, basta definir o hiperparâmetro clip como 'True'.**

Dica

- **Quando a distribuição de uma característica tem uma cauda longa (ou seja, quando valores distantes da média não são exponencialmente raros), tanto o escalonamento min-max quanto a padronização irão comprimir a maioria dos valores em um pequeno intervalo. Modelos de aprendizado de máquina geralmente não gostam disso.**
- Portanto, antes de escalar a característica, você deve primeiro transformá-la para reduzir a cauda longa e, se possível, tornar a distribuição aproximadamente simétrica. Por exemplo, uma maneira comum de fazer isso para características positivas com uma cauda longa à direita é substituir a característica por sua raiz quadrada (ou elevar a característica a uma potência entre 0 e 1).
- Se a característica tiver uma cauda realmente longa e pesada, então substituí-la pelo seu logaritmo pode ajudar. A imagem abaixo mostra como essa característica fica muito melhor quando você calcula seu logaritmo: fica muito próxima de uma distribuição Gaussiana (ou seja, em forma de sino).



- **Os valores alvo também podem precisar ser transformados**.Se a distribuição do alvo tiver uma cauda longa, você pode optar por substituir o alvo pelo seu logaritmo. Mas se fizer isso, o modelo de regressão agora preverá o log do valor mediano da variável, não o valor mediano da variável em si. Você precisará calcular o exponencial da previsão do modelo se quiser o valor mediano previsto da variável. Felizmente, a maioria dos transformadores do Scikit-Learn tem um método inverse_transform(), o que torna fácil calcular o inverso de suas transformações.

Robust Scaler

- O RobustScaler é uma técnica de pré-processamento de dados usada para normalizar ou padronizar os dados. É especialmente útil quando se trabalha com dados que contêm outliers. Diferentemente de outros métodos de escalonamento, como StandardScaler ou MinMaxScaler, o RobustScaler usa estatísticas robustas para calcular os parâmetros de escalonamento, o que o torna menos sensível a outliers.
- O RobustScaler remove a mediana dos dados e os escala de acordo com o intervalo interquartil (IQR), que é a diferença entre o terceiro

$$X_{\text{scaled}} = \frac{X - \text{mediana}(X)}{\text{IQR}(X)}$$

  quartil (Q3) e o primeiro quartil (Q1). A fórmula básica é:
- Desta forma:
  - Para cada valor no conjunto de dados, subtrai-se a mediana dos dados.
  - Em seguida, divide-se o valor resultante pelo IQR dos dados.

- **Como a mediana e o IQR não são afetados por valores extremos, o escalonamento é mais estável e representativo dos dados centrais.**

In [ ]:
```
from sklearn.preprocessing import RobustScaler

# Criando o pipeline
pipeline = Pipeline([
    ('rscaler', RobustScaler())  # RobustScaler
])

# Aprender e aplicar transformações nos dados de treino
X_train_transformed = pipeline.fit_transform(X_train)

# Aplicar as mesmas transformações aos dados de teste
X_test_transformed = pipeline.transform(X_test)
```

Dados de treino, validação e teste

- A divisão dos dados em treinamento, validação e teste é crucial para o desenvolvimento de modelos de machine learning. No estágio de treinamento, o modelo é exposto aos padrões nos dados, e é nessa fase que a otimização dos hiperparâmetros ocorre.
- No entanto, existe sempre o perigo do overfitting, onde o modelo pode se tornar excessivamente complexo e capturar ruídos específicos do conjunto de treinamento em vez de aprender padrões verdadeiramente generalizáveis. Para evitar isso, é introduzido o conjunto de validação, que serve como uma checagem preliminar do desempenho do modelo com dados que ele não aprendeu diretamente.
- O conjunto de teste é deliberadamente mantido à parte e não é usado durante a fase de ajuste do modelo para prevenir o vazamento de dados. Isso mantém a integridade do teste como um meio de avaliar como o modelo se comportará com dados totalmente novos, simulando um ambiente de produção real.
- Técnicas adicionais como a validação cruzada K-fold elevam a robustez do processo de validação. Através dela, os dados de treinamento e validação são divididos em K segmentos. O modelo é então treinado K vezes, cada vez com um segmento diferente atuando como validação e os outros como treinamento. Isso assegura um uso eficiente dos dados disponíveis e minimiza o risco de vazamento de dados.
- O processo de validação de modelos, portanto, é meticulosamente projetado para assegurar que o modelo seja capaz de funcionar de forma confiável e eficaz em qualquer novo cenário que possa encontrar no futuro.

- **Parâmetros:** Os parâmetros de um modelo de machine learning são os valores que o modelo aprende durante o processo de treinamento. Esses parâmetros são ajustados para minimizar uma função de custo, que mede a discrepância entre as previsões do modelo e os valores reais dos dados de treinamento.
  - Em muitos casos, os parâmetros são as ponderações ou coeficientes associados às características de entrada no modelo. Por exemplo, em uma regressão linear, os parâmetros são os coeficientes que multiplicam as características de entrada.
  - **Esses parâmetros são ajustados automaticamente pelo algoritmo de treinamento durante o processo de aprendizado do modelo.**

- **Os hiperparâmetros**, por outro lado, são configurações do modelo que são definidas antes do treinamento e não são ajustadas durante o treinamento. Eles controlam aspectos do processo de treinamento ou do próprio modelo que não são aprendidos automaticamente. A seleção adequada dos hiperparâmetros pode ter um grande impacto no desempenho do modelo, incluindo sua capacidade de generalização para novos dados.
  - A escolha dos hiperparâmetros pode afetar significativamente o desempenho do modelo, e geralmente é feita por tentativa e erro ou por meio de técnicas de otimização específicas.
  - Exemplos comuns de hiperparâmetros incluem o número de vizinhos no algoritmo KNN, a profundidade máxima da árvore de decisão no algoritmo de árvore de decisão, a taxa de aprendizado em algoritmos de gradiente descendente, entre outros.
  - **Normalmente, usamos o conjunto de validação para ajustar hiperparâmetros.**

Métodos de Tunagem de Hiperparâmetros

- Agora, vamos explorar os métodos de ajuste de hiperparâmetros disponíveis para refinar nossos modelos de machine learning, alterando esses valores especiais que não são diretamente aprendidos durante o treinamento.
- Tomando a Random Forest como exemplo, destacamos dois hiperparâmetros cruciais: o número de estimadores e a profundidade máxima das árvores. O número de estimadores refere-se ao total de árvores de decisão no modelo. Em geral, um maior número de árvores pode aumentar a acurácia, mas também amplia o tempo de processamento.
- Já a profundidade máxima determina quão 'profundas' as árvores podem crescer. A escolha ideal desse parâmetro varia conforme o problema e uma escolha inadequada pode levar ao overfitting, resultando em um modelo que performa bem nos dados de treino mas não generaliza para novos dados.
- Para encontrar a melhor configuração de hiperparâmetros, recorremos a estratégias de busca, visto que o espaço de hiperparâmetros é vasto e desconhecido:
  - **Grid Search:** Testa todas as combinações possíveis de hiperparâmetros dentro de um conjunto predefinido. Embora sistemático, pode ser inviável com um grande número de combinações.
  - **Random Search:** Seleciona aleatoriamente combinações de hiperparâmetros para teste. Menos metódico que o Grid Search, mas muitas vezes mais eficiente e capaz de descobrir configurações surpreendentemente eficazes.
- Cada técnica tem seus méritos e a decisão sobre qual usar depende do tempo disponível, da extensão do espaço de hiperparâmetros e da complexidade do modelo em questão. O objetivo comum é experimentar até identificar a configuração que maximiza o desempenho do modelo. Iniciar com o Grid Search pode ser uma boa estratégia para estabelecer um ponto de partida na jornada de tunagem.

Grid Search

- O Grid Search é uma técnica fundamental de ajuste de hiperparâmetros. Ela envolve testar todas as combinações possíveis de valores pré-determinados para cada hiperparâmetro, avaliando cada modelo gerado para determinar a melhor configuração. Tomando, por exemplo, os hiperparâmetros 'número de estimadores' e 'profundidade máxima', com quatro valores distintos para cada um, o Grid Search exploraria todas as 16 combinações possíveis, treinando um modelo para cada uma delas para identificar qual oferece o desempenho mais otimizado.
  - **Escalabilidade:** Como testa todas as combinações possíveis de hiperparâmetros, o Grid Search se torna computacionalmente inviável quando o número de hiperparâmetros e seus possíveis valores aumenta.
  - **Granularidade:** Se a grade de valores de hiperparâmetros não for suficientemente densa, pode-se perder o conjunto ótimo de hiperparâmetros que não está na grade especificada.

    - **Eficiência:** Pode gastar muito tempo avaliando hiperparâmetros que têm pouco impacto no desempenho do modelo.

Random Search

- A Random Search se diferencia do Grid Search por sua abordagem probabilística na seleção de hiperparâmetros. Em vez de testar um conjunto específico de valores, ela define distribuições de onde os valores são escolhidos aleatoriamente. Por exemplo, ao invés de limitar o número de árvores em uma Random Forest a valores fixos, a Random Search pode escolher qualquer número dentro de uma faixa, como de 1 a 100, seguindo uma distribuição média de 100 árvores.
- Esta técnica é eficaz porque nem todos os hiperparâmetros influenciam igualmente o desempenho do modelo. Ao permitir uma exploração mais abrangente do espaço de hiperparâmetros, a Random Search pode identificar conjuntos de valores impactantes mais rapidamente do que o Grid Search. Definindo um número de iterações, o usuário limita a quantidade de combinações aleatórias a serem testadas, permitindo que cada iteração avalie o modelo com uma nova seleção de valores.
- Essencialmente, a Random Search é mais flexível e, frequentemente, mais eficiente, pois não presume quais hiperparâmetros são mais importantes, oferecendo assim oportunidades iguais para explorar todas as variáveis em jogo.
    - **Cobertura:** Pode não explorar adequadamente o espaço de hiperparâmetros se o número de iterações for muito baixo.
    - **Direção:** Falta de um método sistemático para guiar a busca; depende da sorte para encontrar a combinação ótima.
    - **Repetição:** Há um risco de selecionar a mesma combinação mais de uma vez, especialmente se o espaço de hiperparâmetros não for bem definido.

HalvingRandomSearchCV e HalvingGridSearchCV

- O Scikit-Learn também possui as classes HalvingRandomSearchCV e HalvingGridSearchCV para busca de hiperparâmetros
- Normalmente, ao fazer busca de hiperparâmetros usando GridSearchCV ou RandomizedSearchCV, todas as combinações de hiperparâmetros são avaliadas usando validação cruzada. Isso pode ser extremamente caro em termos de tempo e recursos computacionais, especialmente quando há muitas combinações para testar.
- HalvingRandomSearchCV e HalvingGridSearchCV introduzem uma abordagem mais eficiente, onde os recursos computacionais são usados de forma progressiva para explorar um espaço de hiperparâmetros maior ou treinar mais rápido. Aqui está como funcionam em detalhes:
- **Primeira Rodada:**
    - Geração de Candidatos: Na primeira rodada, um grande número de combinações de hiperparâmetros (chamados de 'candidatos') é gerado. No caso do HalvingRandomSearchCV, os candidatos são gerados de forma aleatória, enquanto no HalvingGridSearchCV, eles são gerados usando a abordagem de grade.
    - Treinamento com Recursos Limitados: Cada modelo candidato é treinado com recursos limitados. 'Recursos limitados' geralmente significa usar apenas uma pequena parte do conjunto de treinamento ou reduzir o número de iterações de treinamento. Isso acelera o processo de treinamento inicial.
- **Seleção e Avanço para a Próxima Rodada:**
    - Avaliação e Seleção: Após o treinamento inicial, todos os candidatos são avaliados usando validação cruzada. Apenas os melhores candidatos (com base em suas pontuações) são selecionados para avançar para a próxima rodada.
    - Mais Recursos para os Melhores Candidatos: Na próxima rodada, os candidatos selecionados recebem mais recursos para treinamento. Isso pode significar usar uma parte maior do conjunto de dados de treinamento ou aumentar o número de iterações de treinamento.
    - **Rodadas Subsequentes:**
    - Iteração: O processo de avaliação, seleção e alocação de mais recursos continua em rodadas subsequentes. Em cada rodada, o número de candidatos diminui, mas os recursos alocados a cada candidato aumentam.
    - Rodada Final: Após várias rodadas, os melhores candidatos restantes são avaliados usando todos os recursos disponíveis. Isso permite uma avaliação completa e precisa dos modelos mais promissores
- Ao limitar os recursos nas primeiras rodadas, economiza-se tempo e poder computacional, permitindo que um número maior de combinações de hiperparâmetros seja testado.
- Essa abordagem permite explorar um espaço de hiperparâmetros maior sem a necessidade de treinar exaustivamente cada combinação desde o início.
- Concentra os recursos computacionais nos candidatos mais promissores à medida que a busca avança.

Validação

- A validação em machine learning é crucial porque ajuda a avaliar a capacidade de generalização de um modelo, ou seja, quão bem o modelo se comporta em novos dados, não vistos durante o treinamento.
- Há diversas técnicas de validação, mas as mais comuns são a validação cruzada e o método hold-out. Aqui está uma visão geral de cada uma:

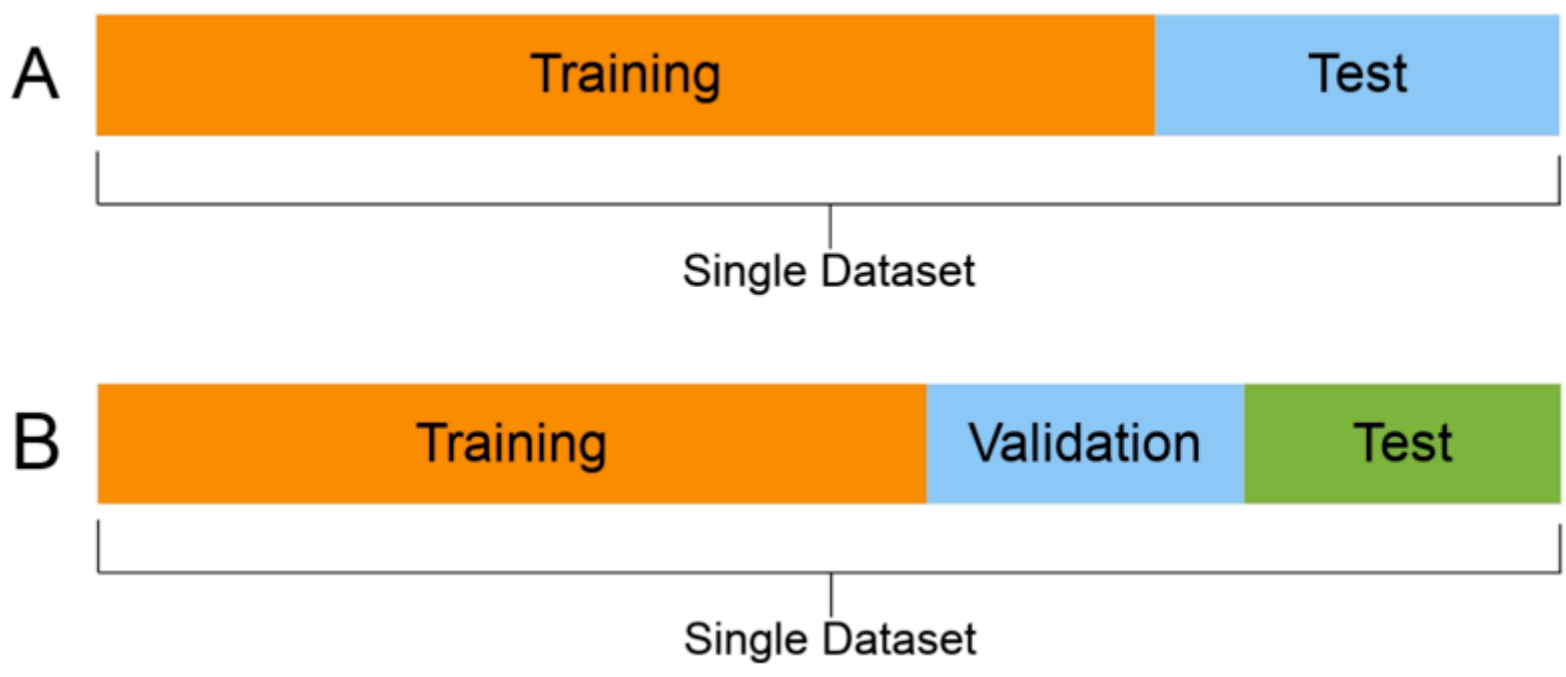
Método Hold-out

- O método hold-out é mais simples e envolve dividir o conjunto de dados em duas partes: uma para treino e outra para teste. Geralmente, essa divisão é feita uma única vez. O modelo é treinado no conjunto de treino e depois avaliado no conjunto de teste.

- **Embora mais rápido e menos computacionalmente intensivo que a validação cruzada, o hold-out pode ser menos confiável se o conjunto de dados não for suficientemente grande ou se a divisão não representar bem a variação total dos dados. Isso pode levar a estimativas de desempenho que dependem muito de como os dados são divididos.**

- A técnica de hold-out, embora útil, pode ter uma limitação importante relacionada exatamente à sorte (ou ao azar) na divisão dos dados. Dependendo de como os dados são divididos entre os conjuntos de treino e teste, o modelo pode apresentar um desempenho muito bom ou muito pobre. Isso pode acontecer se, por exemplo, algumas características ou padrões incomuns dos dados se concentrarem em apenas um dos conjuntos.
- Já quando temos um conjunto de dados pequeno, qualquer divisão aleatória pode levar a uma representação não balanceada das características dos dados. Isso pode resultar em uma estimativa de desempenho pessimista, porque os modelos poderiam ser melhores se tivessémos mais dados.
- Além disso, o método hold-out pode introduzir dependências indiretas entre o conjunto de dados de treino e teste devido à variabilidade na divisão dos dados em conjunto de dados pequenos.

- **Conjunto de dados de treino, validação e teste:**
  - **Dados de Treinamento (Training Data):** Os dados de treinamento são usados para treinar o modelo durante o processo de aprendizado. Representam a parte dos dados disponíveis que será utilizada para ensinar o modelo. Geralmente, são a maior parte do conjunto de dados disponível, pois o modelo precisa de uma quantidade suficiente de exemplos para aprender padrões e generalizar bem para novos dados.
  - **Dados de Validação (Validation Data):** Os dados de validação são usados para ajustar os hiperparâmetros do modelo e realizar a seleção do modelo. Por isto, representam uma parte dos dados de treinamento que é reservada para ajustar os hiperparâmetros do modelo e realizar a seleção do modelo. São usados para estimar o desempenho do modelo em dados não vistos durante o treinamento.
  - **Dados de Teste (Test Data):** Os dados de teste são usados para avaliar o desempenho do modelo depois que ele foi treinado. Representam uma parte dos dados que o modelo nunca viu durante o treinamento e são usados para avaliar sua capacidade de generalização.



Exemplo em python:

In [1]:
```
#Importando a biblioteca pandas
import pandas as pd

#Importando a função train_test_split para divisão dos dados
from sklearn.model_selection import train_test_split
```

In [2]:
```
# Carregando o conjunto de dados
df = pd.read_csv('insurance.csv')
```

In [3]:
```
# Separando as features (x) e a variável alvo(y)
X = df.drop('charges', axis =1)
```

```
y = df['charges']
```

In [4]:
```
# Separando em treino e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

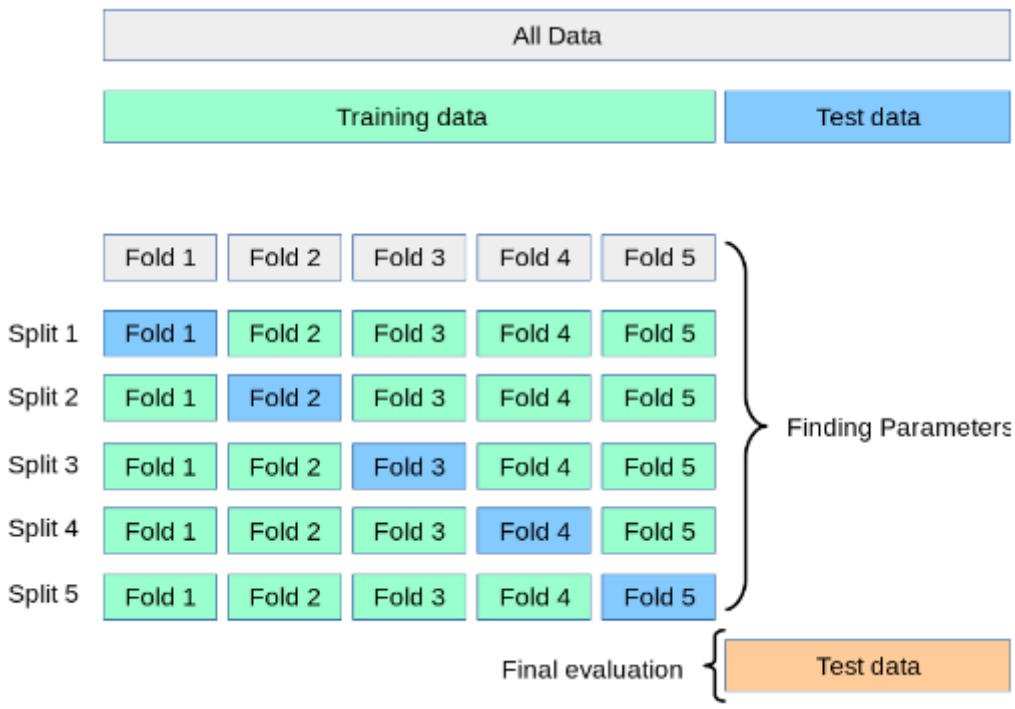
In [5]:
```
# Verificando as dimensões
print(X_train.shape, y_train.shape, X_test.shape, y_test.shape)
```

```
(936, 6) (936,) (402, 6) (402,)
```

Posteriormente seguiríamos para demais pré-processamento como scaling, normalização e/ou encoding, etc.

Validação Cruzada (Cross-validation)

- A validação cruzada é uma técnica robusta e amplamente usada. Ela envolve dividir o conjunto de dados em múltiplos 'folds' ou partes. Por exemplo, na validação cruzada k-fold, o conjunto de dados de treino é dividido em k subconjuntos. O modelo é treinado k vezes, cada vez usando k-1 subconjuntos como dados de treino e o subconjunto restante como dados de teste. Isso é feito de forma que cada subconjunto seja usado uma vez como dados de teste.
- Essa técnica é benéfica porque:
  - Maximiza tanto os dados de treino quanto os de teste, o que é especialmente útil em conjuntos de dados pequenos. Fornece uma medida mais abrangente de quão bem o modelo pode generalizar em novos dados, já que cada ponto de dados é usado tanto para treino quanto para teste.
  - Na validação cruzada, como cada parte dos dados de treino é usada tanto para treino quanto para teste em algum momento, você consegue minimizar o risco de que uma divisão específica influencie demais a avaliação de desempenho do modelo. Isso ajuda a garantir que a performance refletida pelo modelo é verdadeiramente representativa da sua capacidade de generalização em diferentes subconjuntos dos dados.



Para evitar o vazamento de dados na validação cruzada, é crucial garantir que qualquer pré-processamento dos dados seja feito dentro do loop de validação cruzada, ou seja, que o pré-processamento seja aplicado separadamente em cada fold de treino e teste.

- Aqui está o que acontece passo a passo para prevenir o vazamento:
  - **Separação dos Folds:** Na validação cruzada, os dados de treino são divididos em k folds. Cada fold atua como um conjunto de teste em um ponto, enquanto os demais k-1 folds são usados para treinamento.
  - **Isolamento de Preprocessamento e Treinamento:** Com o uso do Pipeline, quando um fold específico é usado como conjunto de teste, os passos de pré-processamento (como a normalização com StandardScaler) são ajustados (fit) apenas nos dados de treino dos k-1 folds e aplicados (transform) tanto aos folds de treino quanto ao fold de teste. Este processo é repetido para cada um dos k folds, garantindo que o pré-processamento e o treinamento do modelo são feitos de maneira isolada para cada combinação de treino/teste.
  - **Prevenção de Vazamento:** Como o ajuste do pré-processamento não inclui dados do fold de teste em cada iteração, não há informações do conjunto de teste sendo usadas para influenciar o modelo durante o treinamento. Isso previne o vazamento de dados, pois cada teste é realizado em dados "não vistos" pelo modelo durante o ajuste dos parâmetros de pré-processamento e treinamento.

- Quando se utiliza GridSearchCV ou RandomizedSearchCV para a otimização de hiperparâmetros, o processo envolve treinar e avaliar o modelo em múltiplos folds, usando validação cruzada.
- Para cada combinação única de hiperparâmetros, o GridSearchCV exaustivamente avalia todas as combinações possíveis dentro do grid especificado. Já o RandomizedSearchCV seleciona aleatoriamente um número fixo de combinações de hiperparâmetros a partir de um espaço de busca.
- **Durante a validação cruzada, para cada combinação única de hiperparâmetros, o modelo é treinado e avaliado em todos os folds. A métrica de avaliação (por exemplo, recall) é calculada para cada fold, e a média dessas métricas é então determinada para essa**

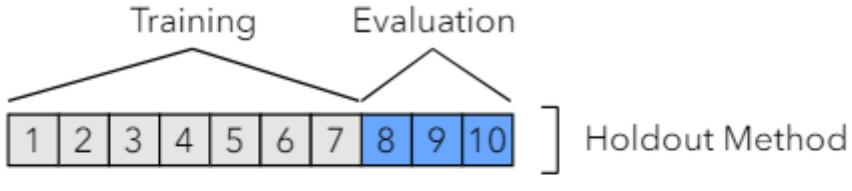
combinação de hiperparâmetros. Este processo garante que a avaliação do modelo seja robusta e menos suscetível a variações nos dados de treino e teste.

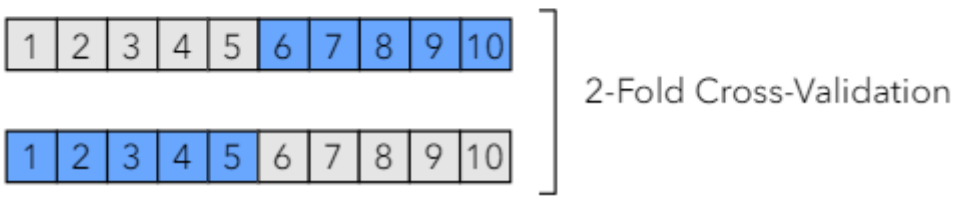
- Esse processo é repetido para todas as combinações de hiperparâmetros no caso do GridSearchCV, ou para as combinações selecionadas aleatoriamente no caso do RandomizedSearchCV. A combinação de hiperparâmetros que obtém a maior média dos scores de validação cruzada é selecionada como a melhor. Isso permite identificar a configuração de hiperparâmetros que proporciona o melhor desempenho esperado do modelo ao generalizar para novos dados.

---

k pequeno

- Quando se utiliza um pequeno número de folds (k pequeno), há menos dados disponíveis em cada fold de treinamento, já que uma parte maior dos dados é reservada para validação em cada iteração. Isso pode levar a estimativas mais pessimistas da performance do modelo, pois ele é treinado com menos dados a cada vez.
- Portanto, ao diminuir o número de folds (k pequeno), o modelo tende a ser mais pessimista porque a quantidade de dados para treinamento é reduzida, o que pode limitar a capacidade do modelo de aprender padrões adequados dos dados.





Exemplo completo usando pipeline em python:

In [6]:
```
# Carregando o conjunto de dados
df = pd.read_csv('insurance.csv')
```

In [7]:
```
df.head()
```

Out[7]:

	age	sex	bmi	children	smoker	region	charges
0	19	female	27.900	0	yes	southwest	16884.92400
1	18	male	33.770	1	no	southeast	1725.55230
2	28	male	33.000	3	no	southeast	4449.46200
3	33	male	22.705	0	no	northwest	21984.47061
4	32	male	28.880	0	no	northwest	3866.85520

In [8]:
```
# Importanto pipeline para encadear múltiplas etapas de processamento e modelagem em um fluxo
from sklearn.pipeline import Pipeline

# Importando demais classes para exemplicar o uso de validação cruzada
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from category_encoders import TargetEncoder

#Importando a classe ColumnTransformer para realizar diferentes transformações em conjuntos de dados numéricos e categóricos
from sklearn.compose import ColumnTransformer

#Importando a classe StratifiedKFold e a função cross_validate para realizar a validação cruzada
from sklearn.model_selection import KFold, StratifiedKFold, cross_validate
```

In [9]:
```
# Separando as features (x) e a variável alvo(y)
X1 = df.drop('charges', axis =1)
y1 = df['charges']
```

In [10]:
```
# Separando em treino e teste
X_train1, X_test1, y_train1, y_test1 = train_test_split(X1, y1, test_size=0.3, random_state=42)
```

In [11]:
```
# Verificando as dimensões
print(X_train1.shape, y_train1.shape, X_test1.shape, y_test1.shape)
```

```
(936, 6) (936,) (402, 6) (402,)
```

In [12]:
```
# Identificando suas colunas categóricas e numéricas
categorical_features = ['sex', 'smoker', 'region']
numeric_features = ['bmi', 'age']

# Pré-processamento com ColumnTransformer
# As colunas numéricas são precisariam passar por transformação, mas aqui é apenas um exemplo de uso
```

```
# Criando o transformador de colunas com TargetEncoder para categóricas e StandardScaler para numéricas
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numeric_features),
        ('cat', TargetEncoder(), categorical_features)
    ])

# Criando o pipeline com o pré-processador e o modelo de regressão linear
pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('regressor', LinearRegression())
])

# Configuração do KFold para a validação cruzada
KF = KFold(n_splits=5, shuffle=True, random_state=42)

# O parâmetro shuffle=True indica que o conjunto de dados será embaralhado antes de ser dividido em folds
```

In [13]:
```
# Executando a validação cruzada com o pipeline, vamos realizar validação cruzada nos dados de treino
results = cross_validate(pipeline, X_train1, y_train1, cv=KF, scoring='neg_root_mean_squared_error', return_train_score=True)
```

In [14]:
```
# Verificando os resultado:
# fit_time: tempo em segundos para treinamento
# score_time: tempo em segundos levado para calcular a métrica de desempenho
# test_score e train_score: Pontuações do modelo nos dados de teste e treino em cada fold para RMSE, neste caso
results_cv = pd.DataFrame(results)
results_cv
```

Out[14]:

	fit_time	score_time	test_score	train_score
0	0.151907	0.033979	-6599.510869	-6071.253847
1	0.212871	0.039973	-5947.559879	-6231.682287
2	0.355780	0.071956	-6739.193321	-6029.593705
3	0.214864	0.051968	-5848.866415	-6261.980209
4	0.144911	0.044973	-5887.673216	-6251.058627

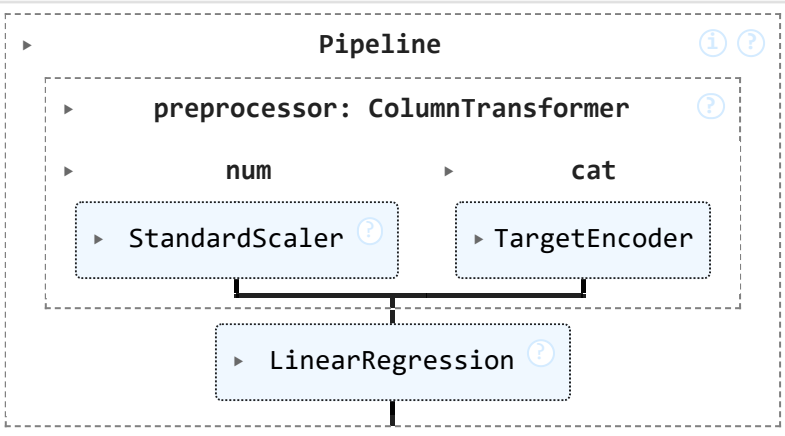
In [15]:
```
# Resumo estatístico das métricas
results_cv.describe()
```

Out[15]:

	fit_time	score_time	test_score	train_score
count	5.000000	5.000000	5.000000	5.000000
mean	0.216067	0.048570	-6204.560740	-6169.113735
std	0.084722	0.014647	428.603313	109.882173
min	0.144911	0.033979	-6739.193321	-6261.980209
25%	0.151907	0.039973	-6599.510869	-6251.058627
50%	0.212871	0.044973	-5947.559879	-6231.682287
75%	0.214864	0.051968	-5887.673216	-6071.253847
max	0.355780	0.071956	-5848.866415	-6029.593705

In [16]:
```
# Treinando o pipeline com todos os dados de treino
pipeline.fit(X_train1, y_train1)
```

Out[16]:



Ao chamar pipeline.predict(X_test1), o pipeline aplica os transformadores que foram ajustados apenas aos dados de treino aos dados de teste. Isso significa que ele transforma X_test1 usando os parâmetros (como média, desvio padrão para StandardScaler e mapeamentos para TargetEncoder) que foram aprendidos exclusivamente a partir de X_train1.

In [17]:
```
# Fazendo previsões com os dados de teste
y_pred_test1 = pipeline.predict(X_test1)
```

In [18]:
```
# Importando métricas de avaliação do scikit-learn
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
# Importando o pacote numpy
import numpy as np
```

```python
# Calculando o Mean Squared Error (MSE)
mse = mean_squared_error(y_test1, y_pred_test1)
# Calculando o Mean Absolute Error (MAE)
mae = mean_absolute_error(y_test1, y_pred_test1)
# Calculando o Root Mean Squared Error (RMSE)
rmse = np.sqrt(mse)
# Calculando o coeficiente de determinação, também conhecido como R^2
r2 = r2_score(y_test1, y_pred_test1)

# Imprimindo os resultados
print(f'Mean Squared Error (MSE): {mse:.4f}')
print(f'Root Mean Squared Error (RMSE): {rmse:.4f}')
print(f'Mean Absolute Error (MAE): {mae:.4f}')
print(f'R-squared (R2): {r2:.4f}')
```

```
Mean Squared Error (MSE): 34392115.0521
Root Mean Squared Error (RMSE): 5864.4791
Mean Absolute Error (MAE): 4202.6714
R-squared (R2): 0.7654
```

StratifiedKFold

- O StratifiedKFold é uma variação do KFold que é usada em validação cruzada. A principal característica do StratifiedKFold é que ele divide os dados de forma a preservar a porcentagem de amostras para cada classe. Isso é especialmente útil em conjuntos de dados onde pode haver um desequilíbrio significativo nas classes. Ele garante que cada fold de treino e teste tenha uma boa representação de todas as classes, o que ajuda a avaliar melhor a performance do modelo.
- **StratifiedKFold é adequado apenas para variáveis alvo categóricas (tipos 'binary' ou 'multiclass')**

In [19]:
```python
df1 = pd.read_csv('diabetes_prediction_dataset.csv')
```

In [20]:
```python
df1.head()
```

Out[20]:

	gender	age	hypertension	heart_disease	smoking_history	bmi	HbA1c_level	blood_glucose_level	diabetes
0	Female	80.0	0	1	never	25.19	6.6	140	0
1	Female	54.0	0	0	No Info	27.32	6.6	80	0
2	Male	28.0	0	0	never	27.32	5.7	158	0
3	Female	36.0	0	0	current	23.45	5.0	155	0
4	Male	76.0	1	1	current	20.14	4.8	155	0

In [21]:
```python
# Separando as features (x) e a variável alvo(y)
X2 = df1.drop('diabetes', axis =1)
y2 = df1['diabetes']
```

In [22]:
```python
# Separando em treino e teste
X_train2, X_test2, y_train2, y_test2 = train_test_split(X2, y2, test_size=0.3, random_state=42)
```

In [23]:
```python
# Verificando as dimensões
print(X_train2.shape, y_train2.shape, X_test2.shape, y_test2.shape)
```

```
(70000, 8) (70000,) (30000, 8) (30000,)
```

In [24]:
```python
# Para transformar as variáveis categóricas em variáveis dummy
from sklearn.preprocessing import OneHotEncoder

#Importando Logistic Regression
from sklearn.linear_model import LogisticRegression
```

In [25]:
```python
# Identificando suas colunas categóricas
categorical_features = ['gender', 'smoking_history']

# Pré-processamento com ColumnTransformer
preprocessor = ColumnTransformer(
    transformers=[
        ('cat', OneHotEncoder(), categorical_features)
    ])

# Criando o pipeline com o pré-processador e o modelo de regressão linear
pipeline1 = Pipeline([
    ('preprocessor', preprocessor),
    ('regressor', LogisticRegression())
])

# Configuração do StratifiedKFold para a validação cruzada
SF = StratifiedKFold(n_splits=20, shuffle=True, random_state=42)
```

In [26]:
```python
# Executando a validação cruzada com o pipeline
results_1 = cross_validate(pipeline1, X_train2, y_train2, cv=SF, scoring='recall_weighted', return_train_score=True)
```

In [27]:
```python
results_cv_1 = pd.DataFrame(results_1)
results_cv_1
```

Out[27]:

	fit_time	score_time	test_score	train_score
0	0.260839	0.043973	0.915143	0.914947
1	0.257844	0.014991	0.915143	0.914947
2	0.369772	0.026982	0.915143	0.914947
3	0.166897	0.012990	0.915143	0.914947
4	0.181890	0.013990	0.915143	0.914947
5	0.195878	0.020987	0.915143	0.914947
6	0.303814	0.028980	0.915143	0.914947
7	0.276828	0.025984	0.914857	0.914962
8	0.257843	0.023983	0.914857	0.914962
9	0.266838	0.021985	0.914857	0.914962
10	0.212871	0.016987	0.914857	0.914962
11	0.249846	0.017989	0.914857	0.914962
12	0.260838	0.021985	0.914857	0.914962
13	0.192881	0.017987	0.914857	0.914962
14	0.251844	0.021986	0.914857	0.914962
15	0.263835	0.016989	0.914857	0.914962
16	0.217868	0.025979	0.914857	0.914962
17	0.197884	0.010989	0.914857	0.914962
18	0.132921	0.011986	0.914857	0.914962
19	0.152906	0.026981	0.914857	0.914962

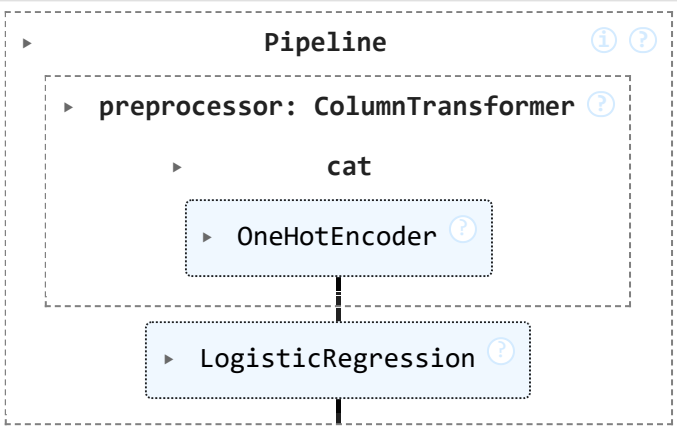
In [28]:
```
# Resumo estatístico das métricas
results_cv_1.describe()
```

Out[28]:

	fit_time	score_time	test_score	train_score
count	20.000000	20.000000	20.000000	20.000000
mean	0.233607	0.021235	0.914957	0.914957
std	0.055923	0.007625	0.000140	0.000007
min	0.132921	0.010989	0.914857	0.914947
25%	0.195129	0.016488	0.914857	0.914947
50%	0.250845	0.021486	0.914857	0.914962
75%	0.261588	0.025980	0.915143	0.914962
max	0.369772	0.043973	0.915143	0.914962

In [29]:
```
# Treinando o pipeline com todos os dados de treino
pipeline1.fit(X_train2, y_train2)
```

Out[29]:



In [30]:
```
df1['diabetes'].value_counts()
```

Out[30]:
```
0    91500
1     8500
Name: diabetes, dtype: int64
```

In [31]:
```
# Fazendo previsões com os dados de teste
y_pred_test2 = pipeline1.predict(X_test2)
```

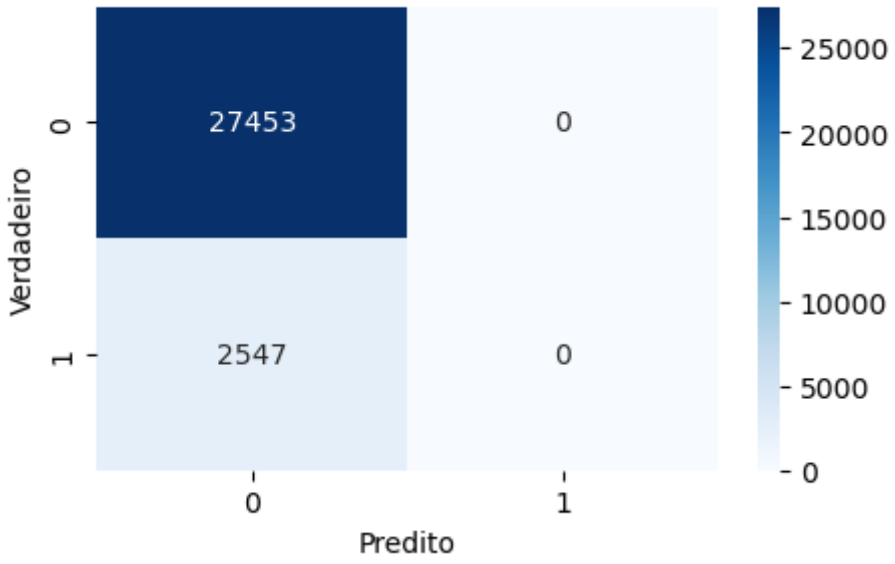
In [32]:
```
# Avaliação do modelo de classificação
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
from sklearn.metrics import recall_score, precision_score, f1_score
import seaborn as sns
import matplotlib.pyplot as plt

# Calculando métricas individuais
accuracy = accuracy_score(y_test2, y_pred_test2)
```

```
precision = precision_score(y_test2, y_pred_test2, average='macro')
recall = recall_score(y_test2, y_pred_test2, average='macro')
f1 = f1_score(y_test2, y_pred_test2, average='macro')

# Imprimindo métricas
print(f'Acurácia: {accuracy:.4f}')
print(f'Precisão: {precision:.4f}')
print(f'Recall: {recall:.4f}')
print(f'F1-score: {f1:.4f}')
```

```
Acurácia: 0.9151
Precisão: 0.4576
Recall: 0.5000
F1-score: 0.4778
```

```
C:\Users\Leticia\anaconda3\Lib\site-packages\sklearn\metrics\_classification.py:1517: UndefinedMetricWarning: Precision is ill-d
efined and being set to 0.0 in labels with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

In [33]:
```
#Matriz de confusão
cm = confusion_matrix(y_test2, y_pred_test2)

# Plotando a matriz de confusão com Seaborn
plt.figure(figsize=(5,3))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
plt.title("Matriz de Confusão")
plt.ylabel("Verdadeiro")
plt.xlabel("Predito")
plt.show()
```



Métricas usada com a função cross_validate:

- **A avaliação durante o processo de validação cruzada difere da avaliação realizada no Hold-out:**
  - Passamos a métrica de interesse diretamente no parâmetro **scoring**
  - Não importamos diretamente as métricas como r2_score, mean_absolute_error, mean_squared_error
  - Os nomes das métricas podem ser diferentes

Classificação:

- **accuracy:** A fração de previsões corretas entre o total de casos.
- **roc_auc, roc_auc_ovr, roc_auc_ovo, roc_auc_ovr_weighted, roc_auc_ovo_weighted:** Área sob a curva ROC (Receiver Operating Characteristic). Avalia a capacidade do classificador de discriminar entre classes. As variantes ovr (one-vs-rest) e ovo (one-vs-one) são para problemas multiclasse.
- **average_precision:** Média de precisões calculadas em cada ponto de corte da curva precision-recall.
- **neg_log_loss:** Log loss negativo, uma medida de performance onde as previsões são probabilidades; valores menores são melhores.
- **precision_micro, precision_macro, precision_weighted, precision_samples:** Precisão calculada globalmente (micro) ou para cada classe e depois média (macro), ponderada pelo suporte (weighted) ou por amostra.
- **recall_micro, recall_macro, recall_weighted, recall_samples:** Recall calculado de forma similar às variantes de precisão.
- **f1, f1_micro, f1_macro, f1_weighted, f1_samples:** F1 score é a média harmônica de precisão e recall, com variantes semelhantes às de precisão e recall.
- **jaccard, jaccard_micro, jaccard_macro, jaccard_weighted, jaccard_samples:** Pontuação de Jaccard, ou índice de interseção sobre união para verdadeiros positivos vs. união de verdadeiros e falsos positivos.
- **balanced_accuracy:** Acurácia que leva em conta o desbalanceamento de classes.
- **matthews_corrcoef:** Coeficiente de correlação de Matthews, uma medida de qualidade para classificações binárias.
- **top_k_accuracy:** Acurácia onde a previsão correta está entre os top k scores mais altos.

Métricas de Regressão:

- **neg_mean_absolute_error:** Erro absoluto médio negativo.
- **neg_mean_squared_error, neg_root_mean_squared_error:** Erro quadrático médio negativo e sua raiz, respectivamente.
- **neg_mean_squared_log_error:** Erro logarítmico quadrático médio negativo.

- **neg_median_absolute_error:** Mediana do erro absoluto negativo.
- **explained_variance:** A fração da variância que é explicada pelo modelo.
- **r2:** Coeficiente de determinação R^2, proporciona uma indicação de quão bem os resultados observados são replicados pelo modelo.
- **max_error:** O maior erro entre a previsão e o valor verdadeiro.
- **neg_mean_absolute_percentage_error:** Erro percentual absoluto médio negativo.
- **neg_mean_gamma_deviance, neg_mean_poisson_deviance:** Deviance para distribuições Gamma e Poisson em regressão.

Você pode passar mais de um scoring por teste:

- **Por exemplo: scoring=['recall_weighted', 'precision_weighted', 'f1_weighted']**

Utilizando GridSearch com validação cruzada:

- Vamos utilizar o dataset diabetes

In [34]:
```
#Importando KNN
from sklearn.neighbors import KNeighborsClassifier

#Seleção dos melhores parâmetros
from sklearn.model_selection import GridSearchCV
```

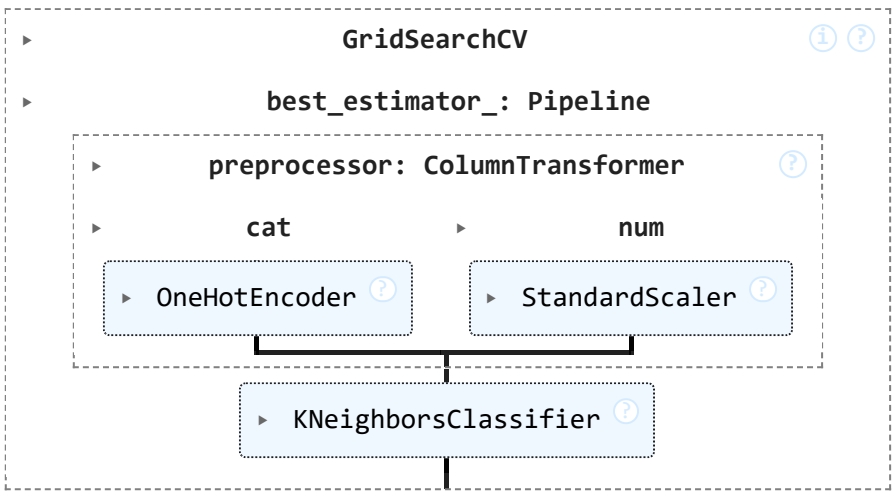
In [35]:
```
# Separando em treino e teste
X_train3, X_test3, y_train3, y_test3 = train_test_split(X2, y2, test_size=0.3, random_state=42)

# Verificando as dimensões
print(X_train3.shape, y_train3.shape, X_test3.shape, y_test3.shape)
```

```
(70000, 8) (70000,) (30000, 8) (30000,)
```

In [36]:
```
# Identificando suas colunas categóricas
categorical_features = ['gender', 'smoking_history']
numeric_features = ['bmi', 'age', 'HbA1c_level', 'blood_glucose_level']

# Pré-processamento com ColumnTransformer
preprocessor = ColumnTransformer(
    transformers=[
        ('cat', OneHotEncoder(), categorical_features),
        ('num', StandardScaler(), numeric_features)
    ])

# Criando o pipeline com o pré-processador e o modelo de regressão linear
pipeline1 = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', KNeighborsClassifier())
])

# Parâmetros para o GridSearchCV, entre 1 e 3 vizinhos:
param_grid = {
    'classifier__n_neighbors': range(1, 4)
}

# Configuração do StratifiedKFold para a validação cruzada
SF = StratifiedKFold(n_splits=4, shuffle=True, random_state=42)

# Criando o GridSearchCV
grid_search = GridSearchCV(pipeline1, param_grid, cv=SF, scoring=['recall_weighted', 'recall_macro', 'recall_micro'],
                           refit = 'recall_macro', return_train_score=True)
```

In [37]:
```
# Executando o GridSearchCV
grid_search.fit(X_train3, y_train3)
```

Out[37]:



In [38]:
```
# Imprimindo os melhores parâmetros e o melhor score
print(f'Melhores parâmetros: {grid_search.best_params_}')
print(f'Melhor score de validação cruzada com recall: {grid_search.best_score_:.3f}')
```

```
Melhores parâmetros: {'classifier__n_neighbors': 1}
Melhor score de validação cruzada com recall: 0.826
```

In [39]:
```
results_grid = pd.DataFrame(grid_search.cv_results_)
results_grid
```

Out[39]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_classifier__n_neighbors	params	split0_test_recall_weighted	split1_te
0	0.598681	0.032853	2.590146	0.187432	1	{'classifier__n_neighbors': 1}	0.947657	
1	0.571395	0.032021	2.979655	0.153538	2	{'classifier__n_neighbors': 2}	0.963600	
2	0.687920	0.107863	3.664183	0.541429	3	{'classifier__n_neighbors': 3}	0.959429	

3 rows × 45 columns

In [40]:
```
# Fazendo previsões com o melhor modelo encontrado pelo GridSearch
y_pred_test3 = grid_search.predict(X_test3)
```

In [41]:
```
# Calculando métricas individuais
accuracy = accuracy_score(y_test3, y_pred_test3)
precision = precision_score(y_test3, y_pred_test3, average='macro')
recall = recall_score(y_test3, y_pred_test3, average='macro')
f1 = f1_score(y_test3, y_pred_test3, average='macro')

# Imprimindo métricas
print(f'Acurácia: {accuracy:.4f}')
print(f'Precisão: {precision:.4f}')
print(f'Recall: {recall:.4f}')
print(f'F1-score: {f1:.4f}')

# Gerando um relatório completo
report = classification_report(y_test3, y_pred_test3)
print('Relatório de Classificação:')
print(report)
```

```
Acurácia: 0.9520
Precisão: 0.8519
Recall: 0.8300
F1-score: 0.8405
Relatório de Classificação:
              precision    recall  f1-score   support

           0       0.97      0.98      0.97     27453
           1       0.73      0.68      0.71      2547

    accuracy                           0.95     30000
   macro avg       0.85      0.83      0.84     30000
weighted avg       0.95      0.95      0.95     30000
```

Utilizando RandomSearch com validação cruzada:

- Vamos utilizar o dataset diabetes

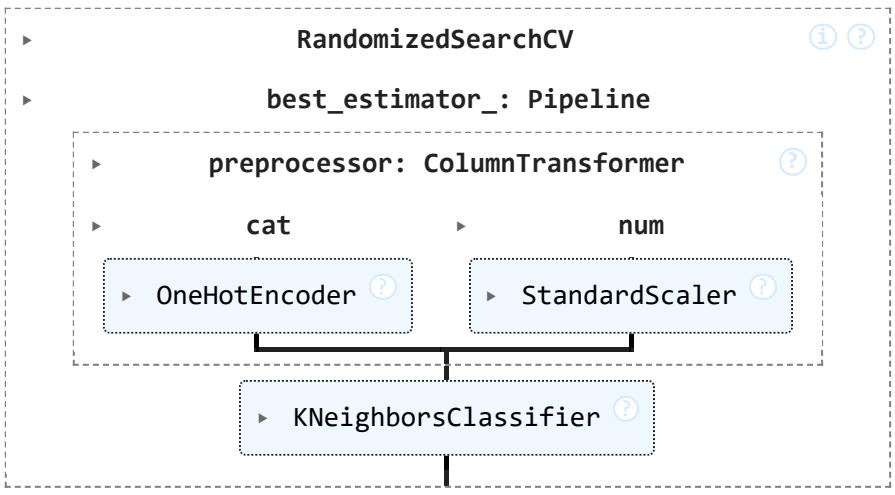
In [42]:
```
# Seleciona aleatoriamente combinações de hiperparâmetros
from sklearn.model_selection import RandomizedSearchCV
```

In [43]:
```
# Identificando suas colunas categóricas e numéricas
categorical_features = ['gender', 'smoking_history']
numeric_features = ['bmi', 'age', 'HbA1c_level', 'blood_glucose_level']

# Pré-processamento com ColumnTransformer
preprocessor = ColumnTransformer(
    transformers=[
        ('cat', OneHotEncoder(), categorical_features),
        ('num', StandardScaler(), numeric_features)
    ])

# Criando o pipeline com o pré-processador e o modelo de classificação KNeighbors
pipeline1 = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', KNeighborsClassifier())
])

# Parâmetros para o RandomizedSearchCV
param_distributions = {
    'classifier__n_neighbors': range(1, 5)
}

# Configuração do StratifiedKFold para a validação cruzada
SF = StratifiedKFold(n_splits=4, shuffle=True, random_state=42)

# Criando o RandomizedSearchCV
random_search = RandomizedSearchCV(
    pipeline1,
    param_distributions=param_distributions,
    n_iter=2,  # Número total de diferentes combinações de hiperparâmetros
    cv=SF,
    scoring='recall_weighted',
    random_state=42,
```

```
    return_train_score=True
)
```

In [44]:
```
# Executando o RandomizedSearchCV
random_search.fit(X_train3, y_train3)
```

Out[44]:



In [45]:
```
# Imprimindo os melhores parâmetros e o melhor score
print(f'Melhores Parâmetros: {random_search.best_params_}')
print(f'Melhor score de validação cruzada recall ponderado: {random_search.best_score_:.3f}')
```

```
Melhores Parâmetros: {'classifier__n_neighbors': 4}
Melhor score de validação cruzada recall ponderado: 0.963
```

In [46]:
```
results_random = pd.DataFrame(random_search.cv_results_)
results_random
```

Out[46]:

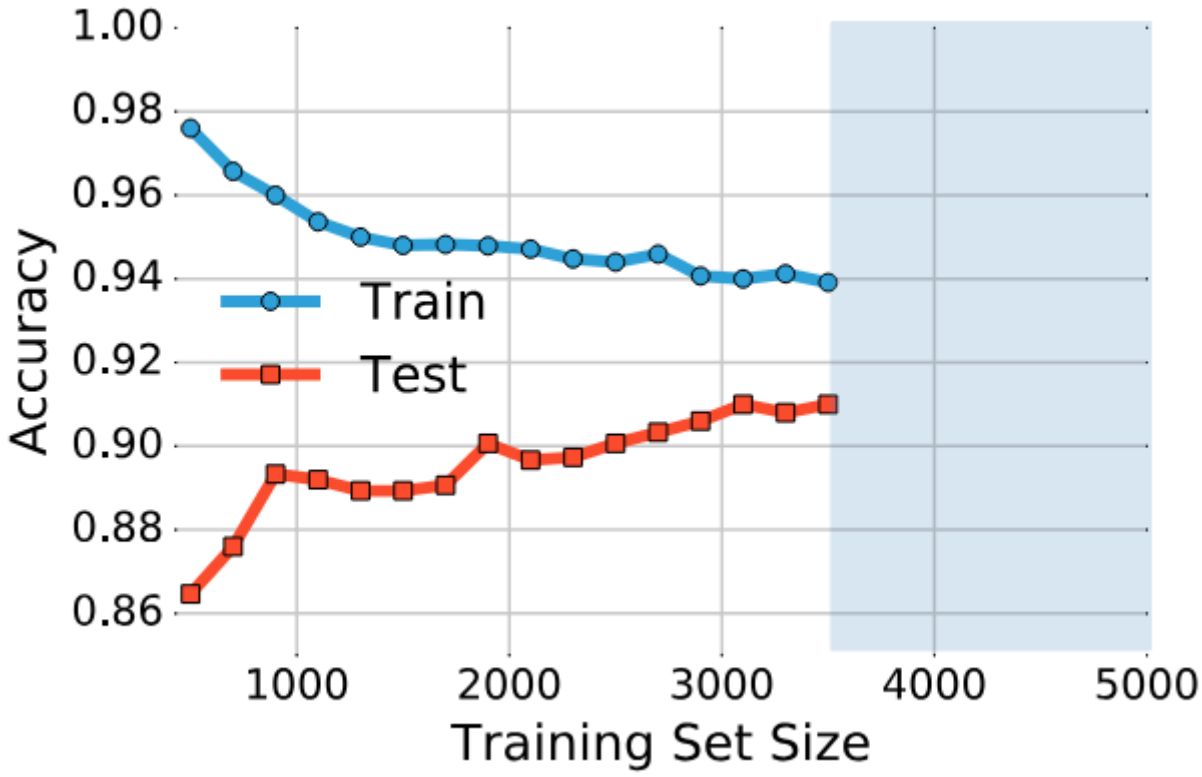
	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_classifier__n_neighbors	params	split0_test_score	split1_test_score
0	0.649233	0.108251	3.259413	0.489667	2	{'classifier__n_neighbors': 2}	0.963600	0.961600
1	0.579139	0.038308	3.606706	0.472565	4	{'classifier__n_neighbors': 4}	0.963829	0.962286

In [47]:
```
# Fazendo previsões com o melhor modelo encontrado pelo RandomSearch
y_pred_test4 = grid_search.predict(X_test3)
```

In [48]:
```
# Calculando métricas individuais
accuracy = accuracy_score(y_test3, y_pred_test4)
precision = precision_score(y_test3, y_pred_test4, average='macro')
recall = recall_score(y_test3, y_pred_test4, average='macro')
f1 = f1_score(y_test3, y_pred_test4, average='macro')

# Imprimindo métricas
print(f'Acurácia: {accuracy:.4f}')
print(f'Precisão: {precision:.4f}')
print(f'Recall: {recall:.4f}')
print(f'F1-score: {f1:.4f}')
```

```
Acurácia: 0.9520
Precisão: 0.8519
Recall: 0.8300
F1-score: 0.8405
Relatório de Classificação:
              precision    recall  f1-score   support

           0       0.97      0.98      0.97     27453
           1       0.73      0.68      0.71      2547

    accuracy                           0.95     30000
   macro avg       0.85      0.83      0.84     30000
weighted avg       0.95      0.95      0.95     30000
```

Considerações



- A imagem mostra uma curva de aprendizado que ilustra o comportamento do desempenho do modelo em função do tamanho do conjunto de treinamento.
- A linha azul representa a precisão no conjunto de treinamento.
    - Conforme o tamanho do conjunto de treinamento aumenta, a precisão no conjunto de treinamento tende a diminuir ligeiramente. Isso ocorre porque com um conjunto de treinamento maior, o modelo encontra mais diversidade nos dados, incluindo ruídos e outliers, tornando mais difícil para o modelo decorar (memorizar) esses dados específicos.
- A linha vermelha representa a precisão no conjunto de teste.
- Conforme o tamanho do conjunto de treinamento aumenta, a precisão no conjunto de teste geralmente aumenta. Isso indica que o modelo é capaz de generalizar melhor os dados, pois aprendeu a capturar os padrões subjacentes nos dados de treinamento, em vez de apenas memorizar os exemplos específicos.

Lidando com dados limitados

Bootstrap Sampling

- O uso da técnica bootstrap é especialmente importante em cenários onde os dados são limitados, pois permite uma melhor estimativa da performance do modelo e de sua variabilidade.
- **Bootstrap é um método de reamostragem que envolve repetidamente amostrar dados com reposição para estimar a precisão de um modelo ou estimativa estatística.**

- Como Funciona o Bootstrap?
    - A partir de um conjunto de dados original com n observações, criam-se várias amostras bootstrap. Cada amostra bootstrap é gerada selecionando n observações do conjunto de dados original com reposição, o que significa que a mesma observação pode ser selecionada várias vezes.
    - Para cada amostra bootstrap, um modelo é treinado. O desempenho do modelo é então avaliado usando as observações que não foram incluídas na amostra bootstrap (conhecidas como 'out-of-bag' samples ou OOB).
    - Estatísticas de desempenho são calculadas para cada amostra bootstrap. As estimativas finais são obtidas calculando a média dessas estatísticas ao longo de todas as amostras bootstrap.

- A imagem ilustra o processo de amostragem bootstrap. Vamos detalhar cada parte da imagem:
    - Conjunto de Dados Original:
        - Representado pela linha azul na parte superior da imagem.
        - Contém as observações xi, x2, ..., x10.

        - Este é o conjunto de dados completo a partir do qual faremos as amostras bootstrap.
            - Amostras Bootstrap:
            - Bootstrap 1, Bootstrap 2, Bootstrap 3
            - Cada uma dessas linhas representa uma amostra bootstrap gerada a partir do conjunto de dados original.
            - A amostragem bootstrap é feita com reposição, o que significa que uma mesma observação pode ser selecionada mais de uma vez.
            - Amostras Out-of-Bag (Fora do Saco):
            - Out-of-Bag samples.
            - Estas são as observações do conjunto de dados original que não foram selecionadas na amostra bootstrap correspondente.
        - Estas amostras são usadas para validar o modelo treinado na amostra bootstrap, fornecendo uma estimativa de desempenho que não é otimista, pois o modelo não foi treinado nesses dados.
- Desta forma:
    - Cada amostra bootstrap é criada reamostrando com reposição do conjunto de dados original.
    - Isso significa que uma observação pode aparecer várias vezes ou não aparecer de forma alguma em uma amostra específica.
    - Por exemplo, em Bootstrap 1, a observação x8 aparece três vezes e a observação x3 não aparece.
    - As amostras out-of-bag são compostas pelas observações que não foram selecionadas na amostra bootstrap correspondente.
- **Este processo é amplamente utilizado para avaliar a precisão de modelos de aprendizado de máquina, especialmente quando o conjunto de dados é pequeno e queremos obter uma estimativa confiável da performance do modelo.**
- No scikit-learn, podemos utilizar o método resample, ele permite gerar novas amostras a partir de um conjunto de dados existente, com ou sem reposição: from sklearn.utils import resample

Pipelines

- Quando trabalhamos com machine learning, um dos desafios críticos é evitar o vazamento de dados entre as etapas de treino e teste. O vazamento de dados ocorre quando a informação do conjunto de teste influencia o processo de treinamento do modelo, resultando em uma avaliação excessivamente otimista do desempenho do modelo. Uma maneira eficaz de evitar esse problema é utilizar pipelines, especialmente durante a validação cruzada.
- Transformações como escalonamento e codificação devem ser ajustadas exclusivamente nos dados de treino e, em seguida, aplicadas aos dados de teste. Caso contrário, estamos deixando que o modelo 'veja' informações dos dados de teste durante o treinamento, comprometendo a validade da avaliação do modelo.
- Além disso, existem muitas etapas de transformação de dados que precisam ser executadas na ordem correta. Felizmente, o Scikit-Learn fornece a classe Pipeline para ajudar com essas sequências de transformações.
- O construtor Pipeline leva uma lista de pares nome/estimador que definem uma sequência de etapas. Os nomes podem ser qualquer coisa que você desejar, contanto que sejam únicos e não contenham duplo sublinhado (__).
- **Durante a validação cruzada, o Pipeline garante que as transformações sejam ajustadas apenas nos dados de treino de cada dobra e, em seguida, aplicadas aos dados de teste dessa mesma dobra. Isso evita que informações dos dados de teste influenciem as transformações aprendidas durante o treinamento.**

Transformando Todas as Colunas com ColumnTransformer

- Para ter um único transformador capaz de lidar com todas as colunas, aplicando as transformações apropriadas a cada coluna, usamos o 'ColumnTransformer'

Exemplos de pipelines

In [1]:
```
#Importando a função train_test_split para divisão dos dados
from sklearn.model_selection import train_test_split

# Importanto pipeline para encadear múltiplas etapas de processamento
from sklearn.pipeline import Pipeline

#Importando a classe ColumnTransformer para realizar diferentes transformações em conjuntos de dados numéricos e categóricos
from sklearn.compose import ColumnTransformer

# Importando classes necessárias para pré-processamento
from sklearn.preprocessing import StandardScaler
from category_encoders import TargetEncoder, OneHotEncoder

# Importa o SimpleImputer, para tratar valores faltantes nos dados
from sklearn.impute import SimpleImputer

#Importando estimadores
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression

#Importando a classe StratifiedKFold  e Kfold para realizar a validação cruzada
from sklearn.model_selection import KFold, StratifiedKFold, cross_validate

#Seleção dos melhores parâmetros
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV

# Avaliação do modelo de classificação
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
from sklearn.metrics import recall_score, precision_score, f1_score
import seaborn as sns
import matplotlib.pyplot as plt

# Avaliação modelo de regressão
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Para calcular os pesos das classes
from sklearn.utils.class_weight import compute_class_weight
```

In [ ]:
```
# Separando as features (x) e a variável alvo(y)
X = df.drop('target', axis =1)
y = df['target']
```

In [ ]:
```
# Separando em treino e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

In [ ]:
```
# Verificando as dimensões
print(X_train.shape, y_train.shape, X_test.shape, y_test.shape)
```

Pipeline para um estimador

In [ ]:
```
#Pipeline
```

```
# Identificando suas colunas categóricas e numéricas
categorical_features = ['', '', '']
numeric_features = ['', '']

# Criando um pipeline para processamento de características numéricas
numerical_transformer = Pipeline(steps=[
    ('scaler', StandardScaler())
])

# Criando um pipeline para processamento de características categóricas
categorical_transformer = Pipeline(steps=[
    ('encoder', OneHotEncoder())
])

# Criando um transformador de colunas com transformações apropriadas para cada tipo de dados
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)
    ])

# Criando o pipeline completo com o pré-processador e o classificador
pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', KNeighborsClassifier())
])

# Configuração do KFold para a validação cruzada, verificar balanceamento dos dados
KF = KFold(n_splits=5, shuffle=True, random_state=42)

#Ou

# Configuração do StratifiedKFold para a validação cruzada
SF = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Parâmetros para o RandomizedSearchCV
param_distributions = {
    'classifier__n_neighbors': range(1, 50),
    'classifier__weights': ['distance', 'uniform'],
    'classifier__algorithm': ['ball_tree', 'kd_tree', 'brute'],
    'classifier__metric': ["manhattan", "euclidean", "minkowski"],
}

# RandomSearch
random_search = RandomizedSearchCV(pipeline, param_distributions=param_distributions,
    n_iter=30,  # Número total de diferentes combinações de hiperparâmetros
    cv=KF, scoring='precision_micro',
    random_state=42, return_train_score=True
)
#Ou

#GridSearch
grid_search = GridSearchCV(pipeline, param_distributions, cv=KF, scoring='precision_micro', return_train_score=True)
```

In [ ]:
```
# Executando o RandomSearch
random_search.fit(X_train, y_train)
```

In [ ]:
```
#Ou
# Executando o GridSearchCV
grid_search.fit(X_train, y_train)
```

In [ ]:
```
# Imprimindo os melhores parâmetros e o melhor score, se grid grid_search.best_params_, grid_search.best_score_:.3f
print(f'Melhores Parâmetros: {random_search.best_params_}')
print(f'Melhor score de validação cruzada: {random_search.best_score_:.3f}')
```

In [ ]:
```
# Resultados, se grid grid_search.cv_results_
results_random = pd.DataFrame(random_search.cv_results_)
results_random
```

In [ ]:
```
# Fazendo previsões com o melhor modelo encontrado pelo RandomSearchCV, grid_search.predict(X_test)
y_pred_test = random_search.predict(X_test)
```

In [ ]:
```
#Classificação
# Calculando métricas individuais
accuracy = accuracy_score(y_test, y_pred_test)
precision = precision_score(y_test, y_pred_test, average='macro')
recall = recall_score(y_test, y_pred_test, average='macro')
f1 = f1_score(y_test, y_pred_test, average='macro')

# Imprimindo métricas
print(f'Acurácia: {accuracy:.4f}')
print(f'Precisão: {precision:.4f}')
print(f'Recall: {recall:.4f}')
print(f'F1-score: {f1:.4f}')

# Gerando um relatório completo
report = classification_report(y_test, y_pred_test)
print('Relatório de Classificação:')
print(report)
```

In [ ]:
```python
#Matriz de confusão
cm = confusion_matrix(y_test, y_pred_test)

# Plotando a matriz de confusão com Seaborn
plt.figure(figsize=(5,3))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Matriz de Confusão')
plt.ylabel('Verdadeiro')
plt.xlabel('Predito')
plt.show()
```

In [ ]:
```python
# Regressão
# Calculando o Mean Squared Error (MSE)
mse = mean_squared_error(y_test, y_pred_test)
# Calculando o Mean Absolute Error (MAE)
mae = mean_absolute_error(y_test, y_pred_test)
# Calculando o Root Mean Squared Error (RMSE)
rmse = np.sqrt(mse)
# Calculando o coeficiente de determinação, também conhecido como R^2
r2 = r2_score(y_test, y_pred_test)

# Imprimindo os resultados
print(f'Mean Squared Error (MSE): {mse:.4f}')
print(f'Root Mean Squared Error (RMSE): {rmse:.4f}')
print(f'Mean Absolute Error (MAE): {mae:.4f}')
print(f'R-squared (R2): {r2:.4f}')
```

Pipeline com balanceamento de classes

In [ ]:
```python
#Atribuindo pesos
class_weights = compute_class_weight('balanced', classes=np.unique(y_train), y=y_train)
weights_dict = dict(zip(np.unique(y_train), class_weights))
weights_dict
```

In [ ]:
```python
# Identificando suas colunas categóricas e numéricas
categorical_features = ['', '', '']
numeric_features = ['', '']

# Criando um pipeline para processamento de características numéricas
numerical_transformer = Pipeline(steps=[
    ('imputer_num', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

# Criando um pipeline para processamento de características categóricas
categorical_transformer = Pipeline(steps=[
    ('imputer_cat', SimpleImputer(strategy='most_frequent')),
    ('encoder', OneHotEncoder())


# Criando um transformador de colunas com transformações apropriadas para cada tipo de dados
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)
    ])


# Criando o pipeline completo com o pré-processador e o classificador
pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', LogisticRegression(class_weight=weights_dict)) #Balanceamento de dados
])

# Configuração do StratifiedKFold para a validação cruzada
SF = StratifiedKFold(n_splits=20, shuffle=True, random_state=42)

# Parâmetros para o GridSearch
param_grid = {
    'classifier__C': [0.001],
    'classifier__penalty': ['l2']
}

#GridSearch
grid_search = GridSearchCV(pipeline, param_grid, cv=SF, scoring='recall_macro', return_train_score=True)
```

In [ ]:
```python
# Executando o GridSearch
grid_search.fit(X_train, y_train)
```

'passthrough'

- As colunas que não são especificadas nas transformações são deixadas inalteradas e passadas adiante

In [ ]:
```python
# Identificando suas colunas numéricas
numeric_features = ['', '']
```

```
# Lista de colunas que não serão transformadas
unchanged_features = ['', '']

# Criando um pipeline para processamento de características numéricas
numerical_transformer = Pipeline(steps=[
    ('min_max_scaler', MinMaxScaler())
])

# Criando um transformador de colunas com transformações apropriadas para cada tipo de dados
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numeric_features),
        ('passthrough', 'passthrough', unchanged_features)
    ])

pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', SVC(class_weight=weights_dict))
])

# Configuração do StratifiedKFold para a validação cruzada
SF = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Parâmetros para o RandomSearch focado nos kernels linear e RBF
param_distributions = {
    'classifier__C': [],
    'classifier__kernel': ['rbf'],
    'classifier__gamma': []
}

# Configuração do RandomizedSearchCV
random_search = RandomizedSearchCV(pipeline, param_distributions, cv=SF, scoring='recall_macro', return_train_score=True)
```

'remainder'

- **Por padrão, as colunas restantes (ou seja, aquelas que não foram listadas) serão descartadas.**
- O hiperparâmetro remainder no ColumnTransformer é útil quando você deseja definir um comportamento padrão **para todas as colunas que não foram explicitamente mencionadas nas transformações**. Ele permite especificar como tratar essas colunas restantes: seja descartando-as (='drop'), passando-as sem alterações(='passthrough'), ou aplicando uma transformação padrão(=StandardScaler()).

- **Para não precisar explicitar múltiplicas colunas em, por exemplo, um dataset com muitas variáveis, podemos separar as características do conjunto de dados X_train em colunas numéricas e categóricas, com as seguintes linhas de código:**

In [ ]:
```
numeric_features = X_train.select_dtypes(include=['number']).columns
categorical_features = X_train.select_dtypes(include=['object']).columns
```

Uso correto de predict() no scikit-learn e ajuste do ponto de corte

- Quando usamos métodos de classificação no scikit-learn, três métodos principais são comumente utilizados:
  - fit(): Treina o modelo com os dados fornecidos, ajustando seus parâmetros internos.
  - predict_proba(): Retorna as probabilidades das classes para cada amostra.
  - predict(): Retorna a classe prevista para cada amostra.

Problema com o Limiar de 0,5

- Por padrão, o método predict() usa um limiar de 0,5 para decidir entre as classes. Isso significa que, se a probabilidade prevista para a classe positiva for maior que 0,5, a amostra será classificada como pertencente à classe positiva. Caso contrário, será classificada como pertencente à classe negativa. Esse limiar pode não ser adequado para todos os casos.

Problema de Dados Desbalanceados

- Em conjuntos de dados desbalanceados, onde uma classe é muito mais comum do que a outra, o uso do limiar padrão de 0,5 pode levar a um desempenho insatisfatório do modelo de classificação. Vamos entender por que isso acontece e como o limiar de 0,5 pode não ser adequado.

1 - Comportamento do Modelo com Dados Desbalanceados

- Em um conjunto de dados desbalanceado, a classe majoritária (a classe que ocorre com mais frequência) domina o treinamento do modelo. Isso ocorre porque o modelo é treinado para minimizar o erro global, e uma maneira fácil de fazer isso é simplesmente prever a classe majoritária na maioria das vezes.
- Exemplo: Se 95% dos dados pertencem à classe negativa (não fraude) e apenas 5% à classe positiva (fraude), o modelo pode aprender a sempre prever 'não fraude' para minimizar o erro, resultando em alta precisão, mas baixa sensibilidade para detectar fraudes.
- **Quando usamos um limiar de 0,5, estamos implicitamente assumindo que as classes são balanceadas, ou seja, que a probabilidade de ocorrência de cada classe é aproximadamente igual. Em um cenário desbalanceado, essa suposição é falsa, e o limiar de 0,5 pode levar a uma alta taxa de falsos negativos ou falsos positivos, dependendo da distribuição das classes.**

- **Viés para a Classe Majoritária:** O modelo tende a aprender mais sobre a classe majoritária (não fraude) porque há mais exemplos dessa classe. Isso pode levar o modelo a prever que a maioria dos novos exemplos também pertencem à classe majoritária.
- **Poucos Exemplos da Classe Minoritária:** Com poucos exemplos da classe minoritária (fraude), o modelo não aprende bem os padrões e características que definem essa classe. Isso resulta em uma menor capacidade de detectar fraudes.
- Como o modelo tem um forte viés ele não vai saber separar as classes adequadamente.
- **Probabilidades Concentradas:** As probabilidades previstas para a classe minoritária (fraude) podem ser próximas, mas muitas vezes abaixo do limiar padrão de 0.5, resultando em classificações incorretas.

O problema principal com dados desbalanceados é que o modelo tende a aprender melhor a classe majoritária e, consequentemente, pode errar ao prever a classe minoritária. Ajustar o limiar de decisão é uma maneira de mitigar esse problema, melhorando a capacidade do modelo de detectar a classe minoritária.

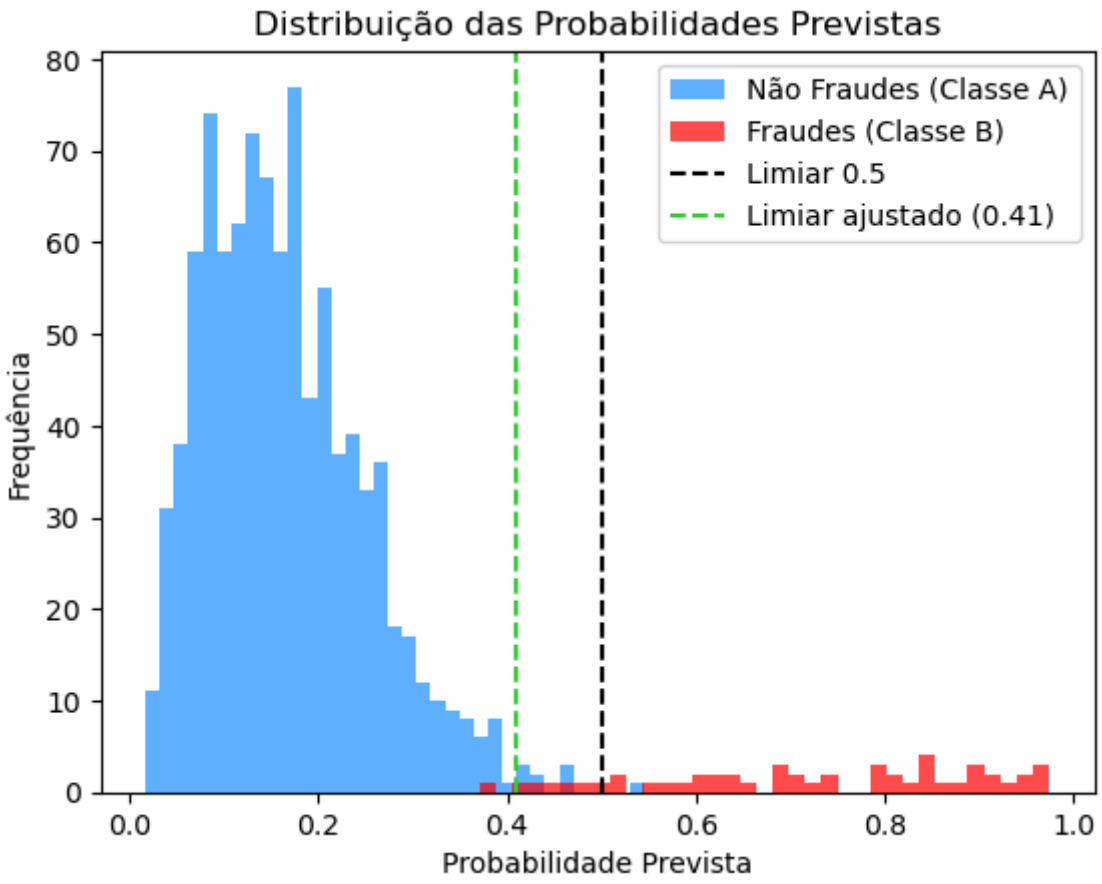
2 - Ilustração do Problema com um Exemplo Prático

- Vamos considerar um exemplo prático para ilustrar por que o limiar de 0,5 pode não ser adequado em dados desbalanceados.
- Vamos supor que temos um conjunto de dados com 1000 exemplos:
  - 950 exemplos da classe A (negativa, não fraude).
  - 50 exemplos da classe B (positiva, fraude).
- O modelo é treinado com essas 1000 transações. Durante o treinamento, o modelo vai ajustar seus parâmetros para prever a classe correta para o máximo de exemplos possível. Se o modelo prever a classe A (negativa) para todos os exemplos, ele terá 95% de precisão, já que 950 de 1000 exemplos são da classe A.
- Após o treinamento, o modelo pode prever probabilidades para cada classe. Por exemplo, ele pode prever que um exemplo tem 0,43 de chance de ser da classe A (negativa) e 0,57 de chance de ser da classe B (positiva). Com um limiar de 0,5:
  - O modelo classifica o exemplo como A (negativa) porque $0,43 < 0,5$.
- Agora, considere que a maioria dos exemplos tem probabilidades previstas como:
  - menor que 0,40 para a classe A (negativa).
  - maior que 0,41 para a classe B (positiva).
- Com um limiar de 0,5, muitos exemplos da classe B (fraude) que têm uma probabilidade prevista entre 0,41 e 0,49 serão incorretamente classificados como pertencentes à classe A (não fraude). Isso resulta em muitas previsões corretas para a classe majoritária, mas muitas previsões erradas para a classe minoritária, reduzindo a eficácia do modelo na detecção de fraudes.

- O problema principal com dados desbalanceados é que o modelo não aprende bem a classe minoritária devido ao pequeno número de exemplos. Isso resulta em probabilidades previstas para fraudes que são muitas vezes mais baixas do que deveriam ser. O modelo tende a prever probabilidades (perto de 0) para a maioria dos exemplos, porque a maioria dos exemplos que viu são de não fraudes. Ajustar o limiar de decisão é uma estratégia para melhorar a detecção da classe minoritária, compensando o aprendizado desbalanceado do modelo.

- **O Gráfico abaixo tenta ilustrar este comportamento em que alguns exemplos seriam incorretamente definidos como não fraude se utilizássemos um limiar arbitrário de 0.5:**

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0)

prob_nfraudes = np.random.beta(3, 15, 950)
prob_fraudes = np.random.beta(6.5, 2, 50)
plt.hist(prob_nfraudes, bins=35, alpha=0.7, label='Não Fraudes (Classe A)', color='#1E90FF')
plt.hist(prob_fraudes, bins=35, alpha=0.7, label='Fraudes (Classe B)', color='red')
plt.axvline(x=0.5, color='black', linestyle='--', label='Limiar 0.5')
plt.axvline(x=0.41, color='#32CD32', linestyle='--', label='Limiar ajustado (0.41)')
plt.xlabel('Probabilidade Prevista')
plt.ylabel('Frequência')
plt.title('Distribuição das Probabilidades Previstas')
plt.legend()
plt.show()
```



- Desta forma, o uso de um limiar fixo de 0,5 em conjuntos de dados desbalanceados pode ser inadequado porque favorece a classe majoritária e não otimiza corretamente para a detecção da classe minoritária. Ajustar o limiar de decisão é uma maneira eficaz de melhorar a sensibilidade do modelo para a classe minoritária, equilibrando melhor as métricas de avaliação, como precisão e recall. Em cenários desbalanceados, é crucial considerar diferentes limiares para otimizar o desempenho do modelo.
- **Felizmente, recentemente foi implementada uma nova funcionalidade na nova versão scikit-learn 1.5 que consegue ajustar esse ponto de corte de decisão após o treinamento do modelo.** Utilizando validação cruzada, este método busca o melhor ponto de corte que maximiza a métrica escolhida. Esse ajuste é crucial porque permite focar mais efetivamente em minimizar erros específicos, como os falsos positivos ou falsos negativos, de acordo com as necessidades do seu projeto e também lidar melhor com problemas desbalanceados.
- A documentação pode ser acessada em: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.TunedThresholdClassifierCV.html

In [ ]:
```
#Atualize o scikit-learn:
#pip install --upgrade scikit-learn==1.5
```

In [2]:
```
import sklearn
print(sklearn.__version__)
```

```
1.5.0
```

Implementação em duas etapas:

- Treina o modelo inicial usando técnicas padrão como busca em grade.
- Utilizar TunedThresholdClassifierCV para ajustar o limiar de decisão do modelo treinado.
- Permite usar qualquer modelo base e depois ajustar o limiar de decisão especificamente.

In [ ]:
```
#Treinando o modelo base
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import StratifiedKFold, GridSearchCV
from sklearn.linear_model import LogisticRegression

# Identificando suas colunas categóricas e numéricas
categorical_features = ['', '', '']
numeric_features = ['', '']
```

```
# Criando um pipeline para processamento de características numéricas
numerical_transformer = Pipeline(steps=[
    ('scaler', StandardScaler())
])

# Criando um pipeline para processamento de características categóricas
categorical_transformer = Pipeline(steps=[
    ('encoder', OneHotEncoder())
])

# Criando um transformador de colunas com transformações apropriadas para cada tipo de dados
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)
    ]
)

# Criando o pipeline completo com o pré-processador e o classificador
pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', LogisticRegression())
])

# Configuração do StratifiedKFold para a validação cruzada
SF = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Parâmetros para o GridSearch
param_grid = {
    'classifier__C': [0.001, 0.01, 0.1],
    'classifier__penalty': ['l2']
}

# Executando o GridSearch
grid_search = GridSearchCV(pipeline, param_grid, cv=SF, scoring='recall_macro', return_train_score=True, n_jobs=-1)
```

In [ ]:
```
# Executando o GridSearchCV
grid_search.fit(X_train, y_train)
```

In [ ]:
```
# Imprimindo os melhores parâmetros e o melhor score
print(f'Melhores Parâmetros: {grid_search.best_params_}')
print(f'Melhor score de validação cruzada: {grid_search.best_score_:.3f}')
```

In [ ]:
```
# Obtendo o melhor estimador
best_estimator = grid_search.best_estimator_
```

- A documentação de base pode ser acessada em: https://scikit-learn.org/stable/auto_examples/model_selection/plot_cost_sensitive_learning.html

In [ ]:
```
# Ajustando o limiar de decisão

from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import TunedThresholdClassifierCV

tuned_model = TunedThresholdClassifierCV(
    estimator=best_estimator,
    scoring='recall_macro',  # Métrica de otimização
    random_state=42,
    store_cv_results=True
)
```

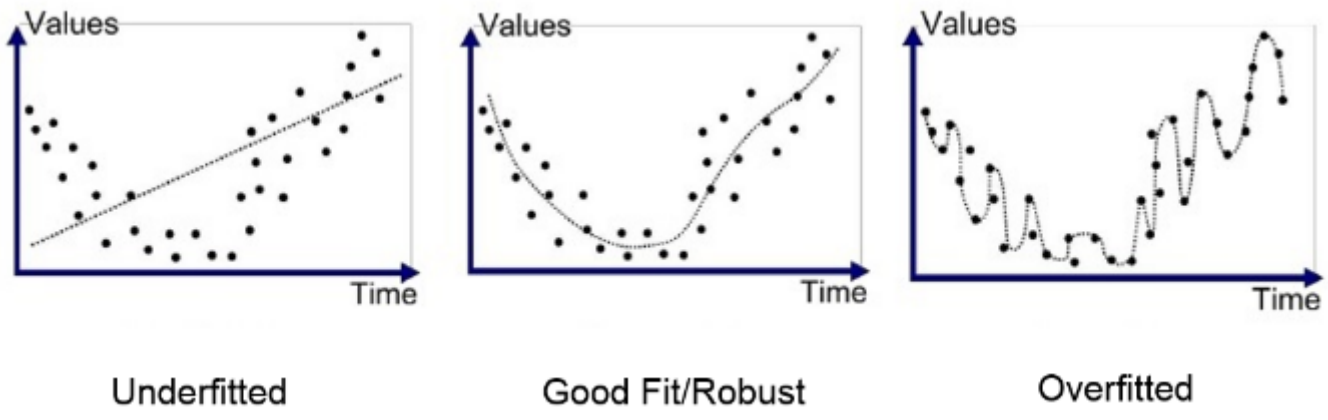
In [ ]:
```
# Ajuste do limiar de decisão
tuned_model.fit(X_train, y_train)
```

In [ ]:
```
print(f'Ponto de corte: {tuned_model.best_threshold_:.3f}')
```

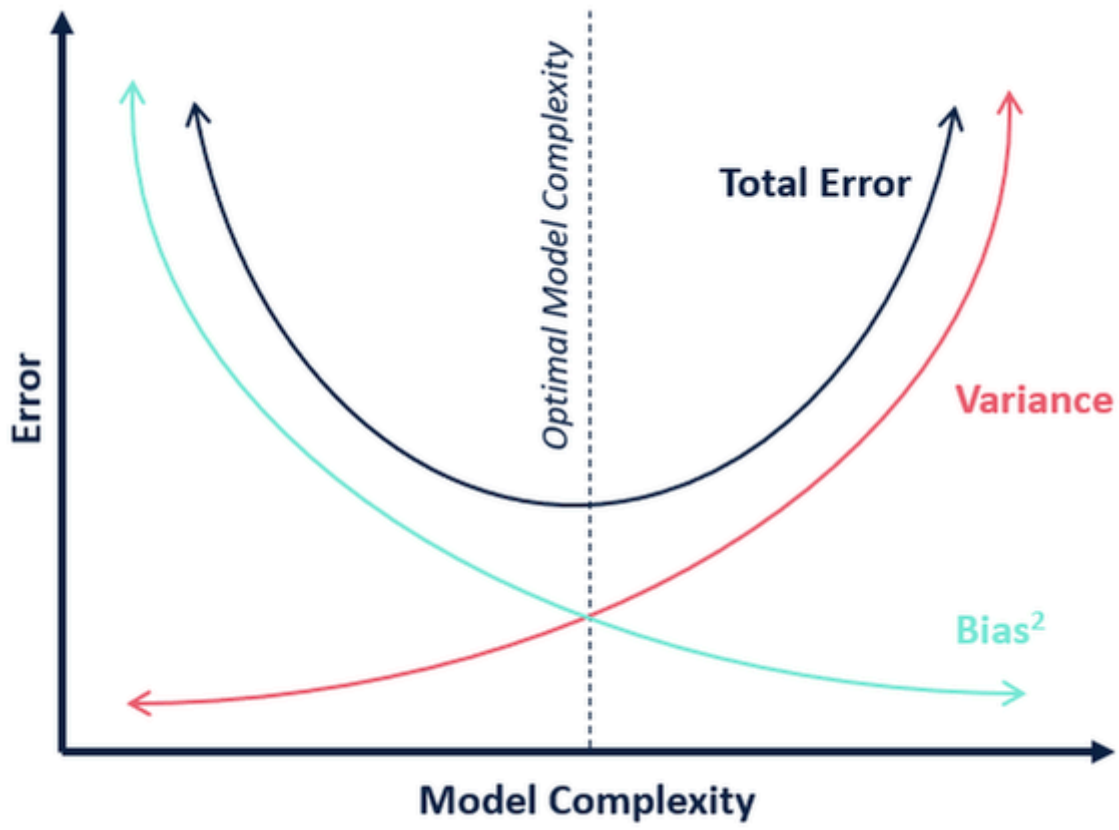
In [ ]:
```
from sklearn.metrics import classification_report
classification_report(y_test, tuned_model.predict(X_test))
```

Underfitting e Overfitting

- **Underfitting (sub-ajuste)**: ocorre quando o modelo não consegue capturar a complexidade dos dados durante o treinamento. Em outras palavras, ele não aprende o suficiente sobre os padrões presentes nos dados. **O resultado é um desempenho ruim tanto nos dados de treino quanto nos dados de teste.** Características típicas de underfitting incluem alta viés e baixa variância, significando que o modelo é incapaz de capturar a complexidade dos dados.
- **Overfitting (sobreajuste):** é o oposto. Nesse cenário, o modelo se ajusta excessivamente aos dados de treinamento, incluindo até mesmo ruídos e variações aleatórias. **Como resultado, ele tem um desempenho excelente nos dados de treino, mas falha ao lidar com novos dados.** O modelo 'decorou' os dados de treino, mas não consegue generalizar bem para situações diferentes. Características típicas de overfitting incluem baixa viés e alta variância, indicando que o modelo é excessivamente sensível às variações dos dados de treinamento.
- Por outro lado, um bom modelo de machine learning é aquele que equilibra adequadamente a complexidade e a capacidade de generalização, **proporcionando um bom desempenho nos dados de treinamento e nos dados de teste. Isso significa que ele tem medidas de erro próximas em ambos os conjuntos.**

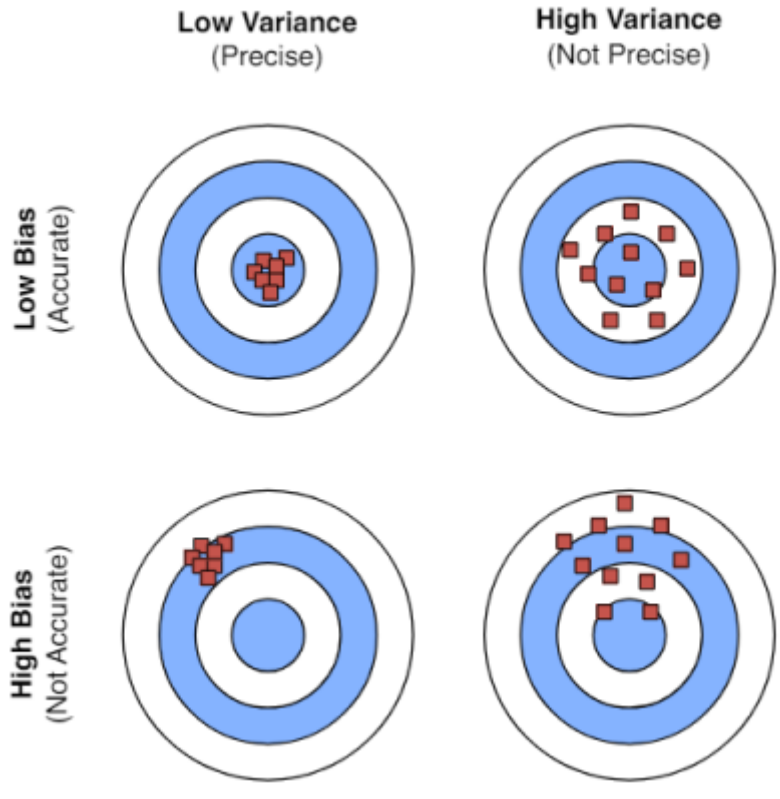


- O gráfico mostra a relação entre a complexidade do modelo e o erro:



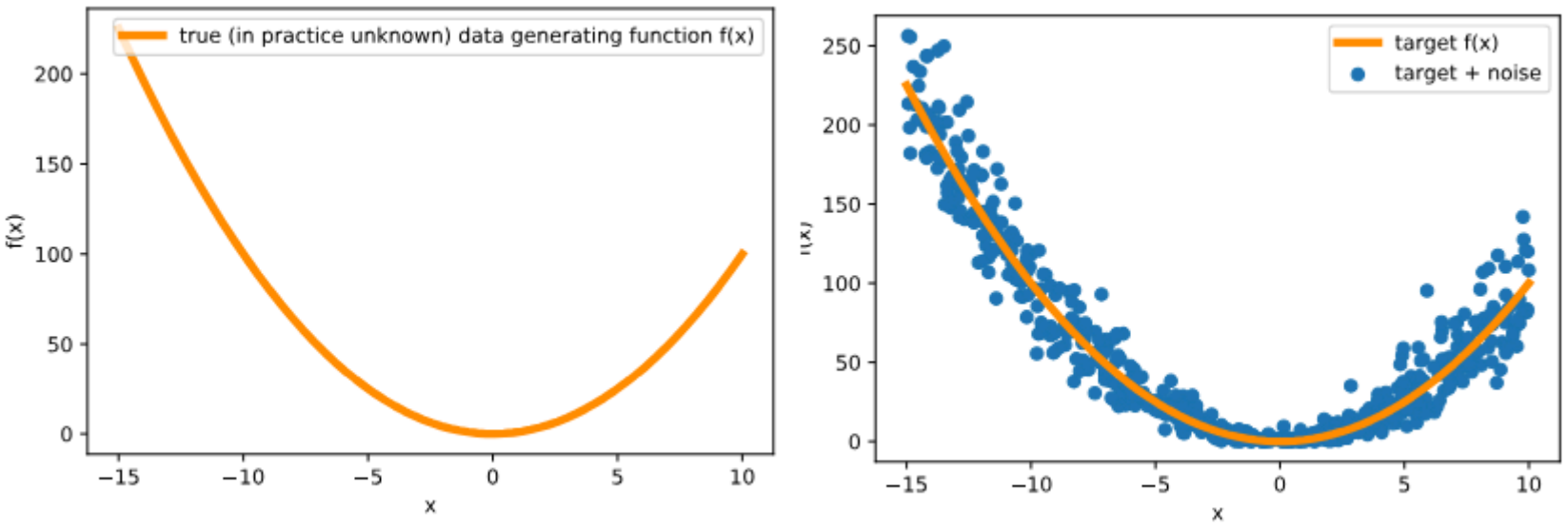
- No lado esquerdo do gráfico, a complexidade do modelo é baixa. Isso resulta em um alto viés e um baixo erro de variância. Modelos simples não conseguem capturar a complexidade dos dados, resultando em underfitting, onde o modelo tem um desempenho ruim tanto nos dados de treinamento quanto nos dados de teste.
- No meio do gráfico, a complexidade do modelo é ótima. Neste ponto, o erro total é minimizado, indicando um bom ajuste. O viés é reduzido e a variância ainda não é alta, resultando em um modelo que generaliza bem.
- o lado direito do gráfico, a complexidade do modelo é alta. Isso reduz o viés ao mínimo, mas aumenta a variância significativamente. Modelos complexos capturam não apenas os padrões subjacentes nos dados, mas também o ruído, resultando em overfitting. Aqui, o modelo tem um desempenho excelente nos dados de treinamento, mas um desempenho ruim nos dados de teste.

Viés e Variância

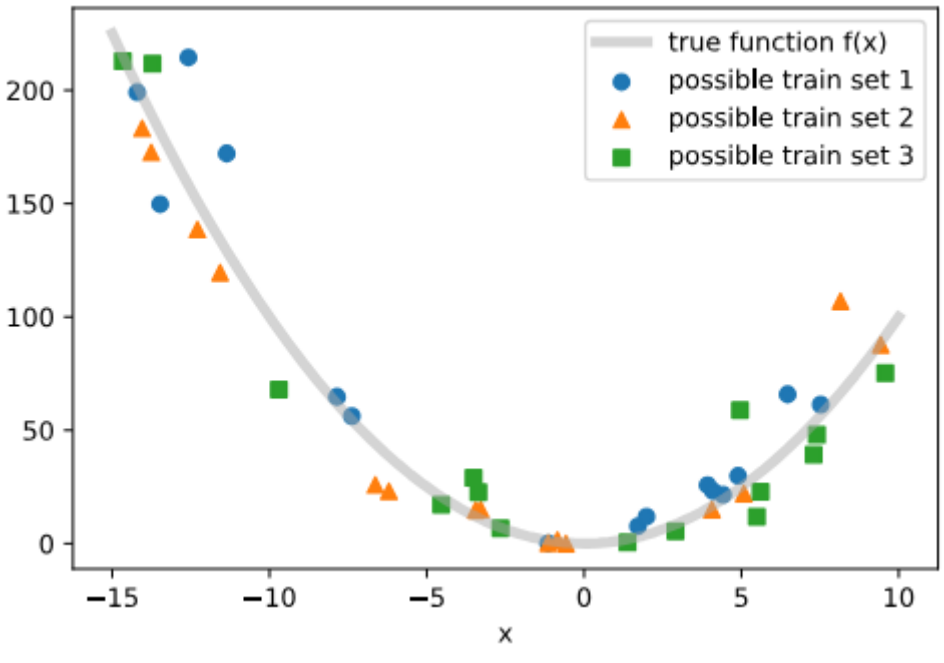


- **Quadrante Superior Esquerdo (Low Bias, Low Variance):**
  - Baixo Viés (Low Bias): O modelo é preciso. Ele faz boas previsões em relação ao alvo real.
  - Baixa Variância (Low Variance): O modelo é consistente. Suas previsões não variam muito com diferentes conjuntos de dados de treinamento.
  - Interpretação: Os pontos (previsões) estão próximos do centro (valor verdadeiro) e muito próximos uns dos outros, indicando que o modelo é preciso e consistente.
- **Quadrante Superior Direito (Low Bias, High Variance):**
  - Baixo Viés (Low Bias): O modelo ainda é preciso.
  - Alta Variância (High Variance): O modelo não é consistente. Suas previsões variam muito com diferentes conjuntos de dados de treinamento.
  - Interpretação: Os pontos (previsões) estão próximos do centro (valor verdadeiro), mas espalhados, indicando que o modelo é preciso, mas não consistente. Isso é típico de um modelo que sofre de overfitting.
- **Quadrante Inferior Esquerdo (High Bias, Low Variance):**
  - Alto Viés (High Bias): O modelo não é preciso. Ele faz previsões que estão longe do alvo real.
  - Baixa Variância (Low Variance): O modelo é consistente. Suas previsões são semelhantes para diferentes conjuntos de dados de treinamento.
  - Interpretação: Os pontos (previsões) estão concentrados em uma área específica, mas essa área está longe do centro (valor verdadeiro), indicando que o modelo é consistentemente impreciso. Isso é típico de um modelo que sofre de underfitting.
- **Quadrante Inferior Direito (High Bias, High Variance):**
  - Alto Viés (High Bias): O modelo não é preciso. Ele faz previsões que estão longe do alvo real.
  - Alta Variância (High Variance): O modelo não é consistente. Suas previsões variam muito com diferentes conjuntos de dados de treinamento.
  - Interpretação: Os pontos (previsões) estão espalhados e longe do centro (valor verdadeiro), indicando que o modelo é impreciso e inconsistente. Isso é o pior cenário possível, onde o modelo sofre tanto de high bias quanto de high variance.

- Considere as imagens abaixo:
- Vamos analisar as duas imagens com a perspectiva de que elas representem um fenômeno natural como a temperatura:

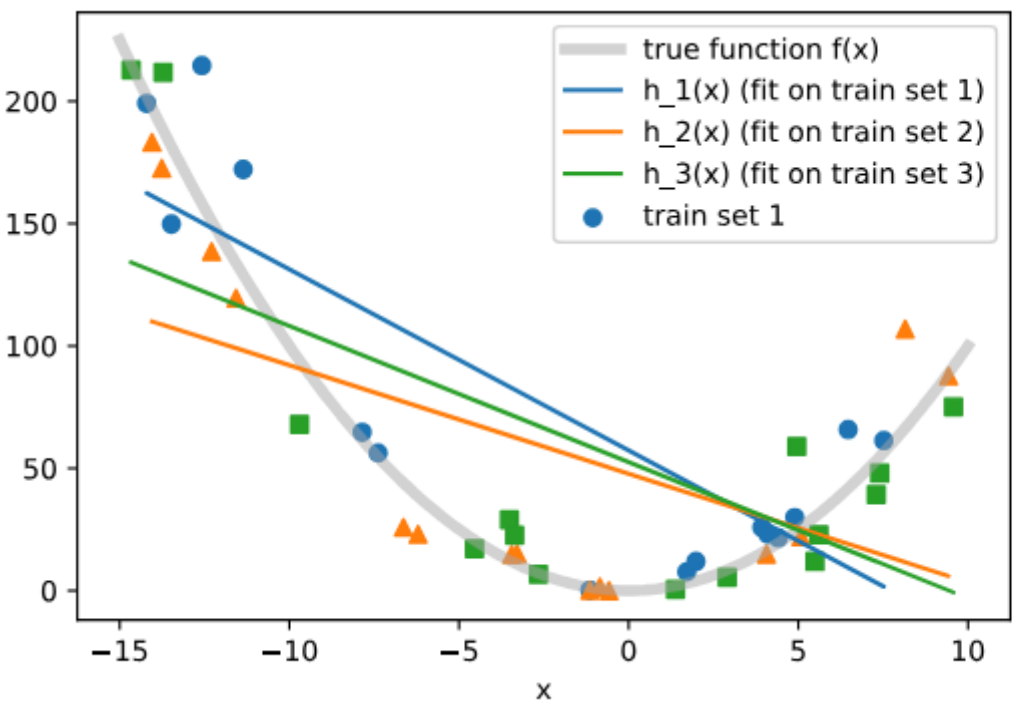


- A primeira imagem mostra uma função teórica f(x) que gera os dados, que seria a verdadeira função que descreve como a temperatura varia em relação a x.
- A segunda imagem adiciona ruído aos dados gerados pela função f(x):
  - Os pontos azuis representam os dados observados da temperatura com ruído. O ruído pode representar variações diárias, medições imperfeitas ou outros fatores imprevisíveis que afetam a temperatura. Isso é mais representativo dos dados que realmente coletamos.



- A terceira imagem mostra a função verdadeira f(x) (a curva cinza) e três conjuntos de treinamento possíveis que são amostras da população.
    - **A curva Cinza (Função Verdadeira f(x)):** Representa a verdadeira relação entre x e y (temperatura, por exemplo).
    - **Pontos Azuis (Conjunto de Treinamento 1):** Um possível conjunto de treinamento amostrado aleatoriamente da população.
    - **Triângulos Laranja (Conjunto de Treinamento 2):** Outro possível conjunto de treinamento amostrado aleatoriamente da mesma população.
    - **Quadrados Verdes (Conjunto de Treinamento 3):** Um terceiro conjunto de treinamento amostrado aleatoriamente da mesma população.
- Cada conjunto de treinamento (azul, laranja, verde) é uma amostra aleatória da população total. Isso reflete a realidade de que, quando coletamos dados para treinar nossos modelos, geralmente temos acesso apenas a uma amostra dos dados totais.
- **Embora todos os três conjuntos de treinamento sejam amostras da mesma população, eles mostram variação significativa. Isso ilustra a variabilidade que pode ocorrer devido à amostragem aleatória. Cada conjunto de treinamento contém pontos que seguem a tendência geral da função verdadeira f(x), mas com diferentes distribuições de ruído e variações.**

- A imagem mostra três modelos de regressão linear ajustados a três diferentes conjuntos de treinamento, sobrepostos à função verdadeira f(x) e aos pontos dos conjuntos de treinamento.

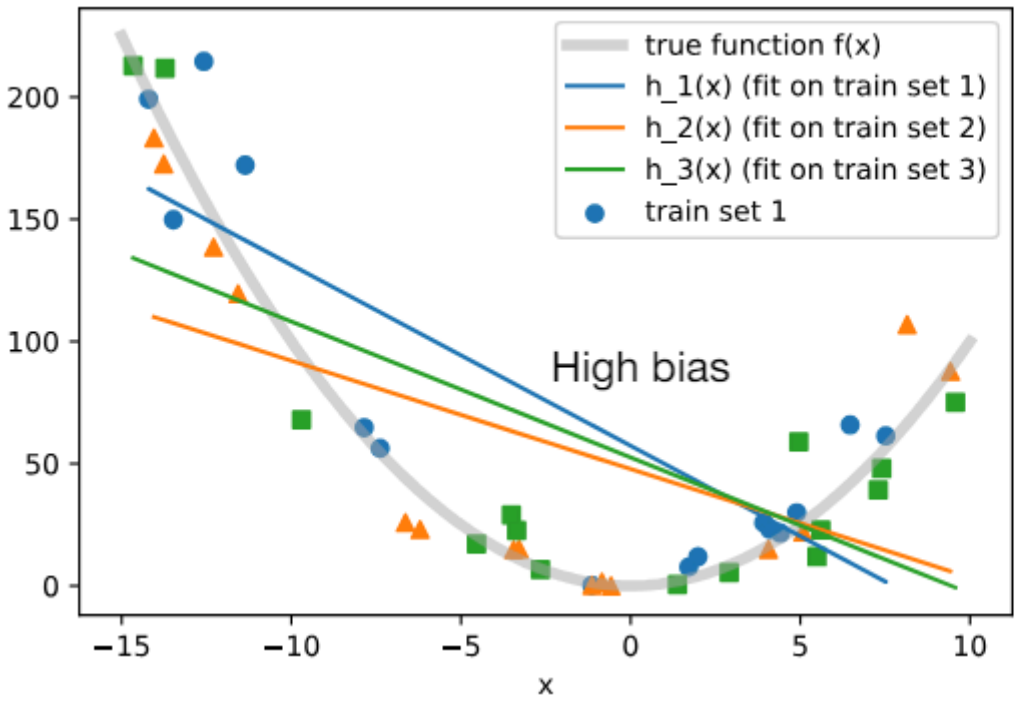


linear regression models

- Linha Azul (h_1(x)): Modelo de regressão linear ajustado ao Conjunto de Treinamento 1.
- Linha Laranja (h_2(x)): Modelo de regressão linear ajustado ao Conjunto de Treinamento 2.
- Linha Verde (h_3(x)): Modelo de regressão linear ajustado ao Conjunto de Treinamento 3.
- Cada uma das linhas (azul, laranja, verde) representa um modelo de regressão linear ajustado a um dos conjuntos de treinamento.
- A regressão linear tenta encontrar a melhor linha reta que se ajusta aos dados de treinamento fornecidos.

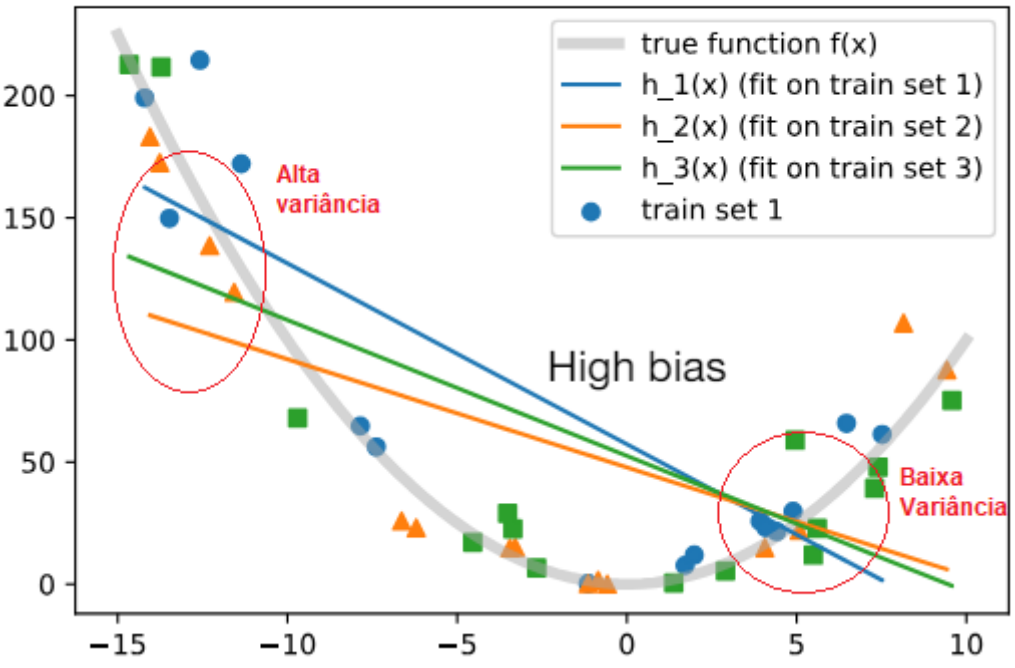
Alto viés:

- A função verdadeira f(x) é uma curva não linear, mas os modelos de regressão linear são apenas linhas retas. Isso mostra uma limitação dos modelos de regressão linear ao tentar capturar relações complexas não lineares.
- **Desta forma, cada modelo de regressão linear tem um viés alto porque não consegue capturar a curvatura da função verdadeira f(x).**

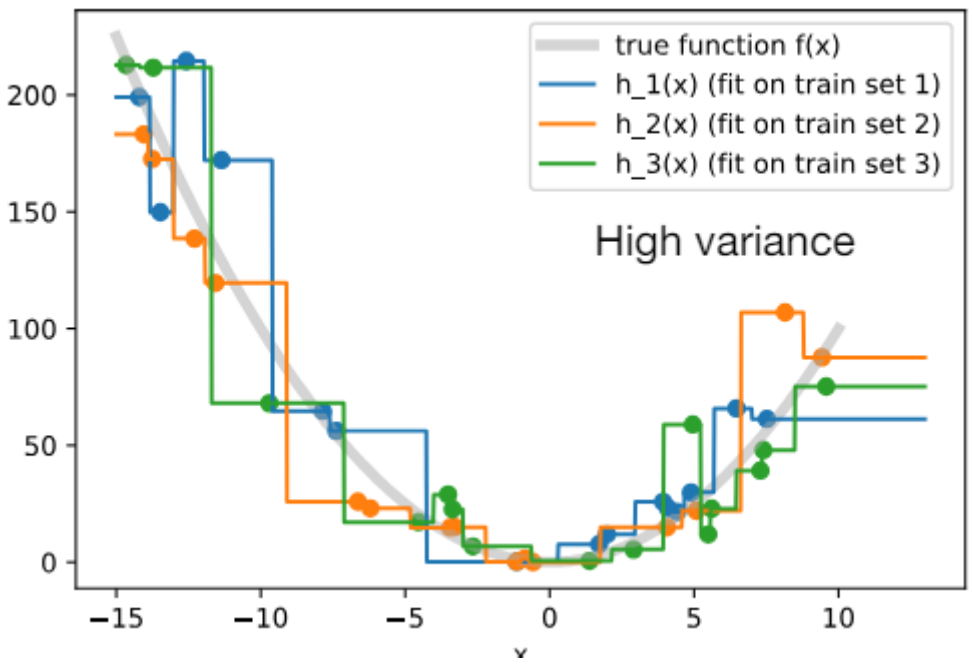


linear regression models

Alta variância



- **Supondo que estejamos utilizando um modelo mais complexo do que a regressão linear, como uma árvore de decisão não podada, ele se ajustaria muito bem aos dados de treinamento. No entanto, se tivermos diferentes conjuntos de treinamento, mesmo que provenientes da mesma população, isso pode afetar significativamente o modelo. Se a simples alteração do subconjunto de treinamento fizer com que o modelo seja muito diferente dos outros, teremos alta variância. Isso significa que as previsões seriam muito dispersas dependendo da aparência do conjunto de treinamento.**



- Modelos ajustados aos conjuntos de treinamento 1, 2 e 3, respectivamente. Estes são modelos complexos, como árvores de decisão não podadas.
- Cada modelo (h_1(x), h_2(x), h_3(x)) se ajusta de maneira bastante diferente aos diferentes conjuntos de treinamento, embora todos os conjuntos sejam amostras da mesma população.
- Isso resulta em modelos que são muito diferentes entre si, dependendo do conjunto de dados de treinamento específico usado.
- As árvores de decisão não podadas representadas na imagem ajustam-se perfeitamente aos conjuntos de treinamento individuais, resultando em modelos que são muito diferentes entre si. Isso demonstra como a alta variância pode levar ao overfitting.

Definição formal de Viés:

$$\text{Bias} = E[\hat{\theta}] - \theta$$

- O viés de um estimador é a diferença entre o valor esperado do estimador θ^ e o verdadeiro valor do parâmetro θ.
- O parâmetro θ é o valor verdadeiro
- E[θ^] é o valor esperado do estimador.

Valor Esperado E[θ^]:

- O valor esperado de θ^ é a média de todas as possíveis estimativas de θ^. Especificadamente, em machine learning, a média das estimativas é a média das previsões feitas por diferentes modelos treinados em diferentes conjuntos de dados da mesma população.
- Desta forma, ao treinar o mesmo estimador em diferentes subconjuntos de treinamento da mesma população, a média das previsões para um ponto específico x é o valor esperado.

Interpretação do Viés:

- Se o viés de um estimador é zero E[θ^]= θ, o estimador é chamado de 'não enviesado' ou 'não tendencioso'. Isso significa que, em média, o estimador acertará o valor verdadeiro do parâmetro θ.
- Se o viés de um estimador é diferente de zero E[θ^] ≠ θ, o estimador é chamado de 'enviesado'. Isso significa que o estimador sistematicamente subestima ou superestima o valor verdadeiro do parâmetro.
  - Um viés positivo indica que o estimador tende a superestimar o parâmetro, enquanto um viés negativo indica que o estimador tende a subestimar o parâmetro.
- **Em resumo, o viés é a diferença entre a média das estimativas de diferentes modelos treinados em diferentes conjuntos de dados da mesma população e o valor real do parâmetro que estamos tentando estimar.**

Definição formal de Variância:

$$\text{Var}[\hat{\theta}] = E\left[(E[\hat{\theta}] - \hat{\theta})^2\right]$$

- Valor Esperado do Estimador E[θ^]): é a média de todas as possíveis estimativas θ^ que o estimador pode produzir.
- θ^ é a estimativa do parâmetro θ obtida a partir de uma amostra específica.
- E[θ^]− θ^ é a diferença entre essa estimativa e o valor esperado do estimador. Representa o desvio da estimativa individual em relação à média de todas as estimativas.
- Elevamos a diferença ao quadrado para garantir que estamos medindo a magnitude do desvio, independentemente da direção (se é maior ou menor que o valor esperado).

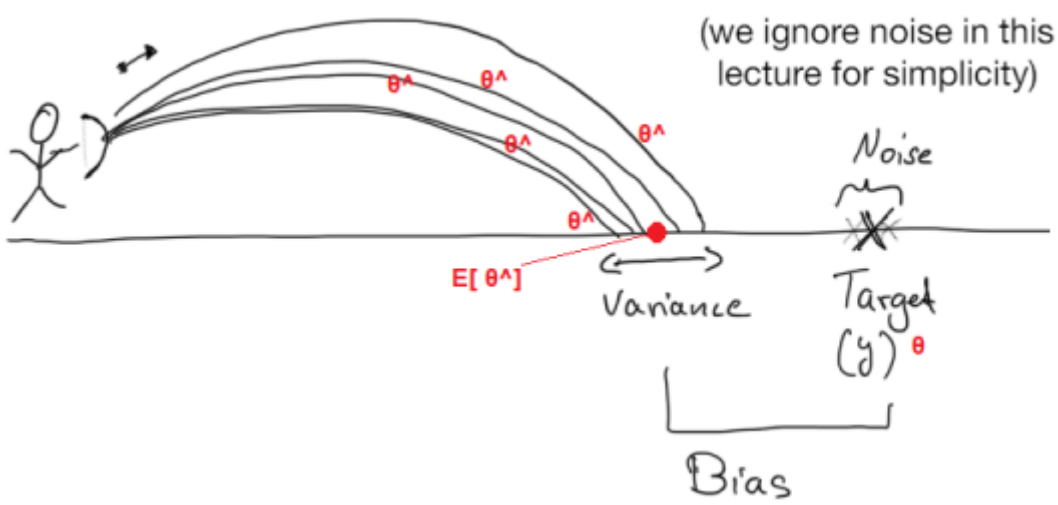
- Quando treinamos o mesmo modelo em diferentes subconjuntos de dados de treinamento, obtemos várias versões do modelo. Cada versão é ligeiramente diferente porque os dados de treinamento são diferentes.
- Cada modelo faz suas próprias previsões para um ponto específico x. Essas previsões podem variar de um modelo para outro.
- Calculamos a média de todas essas previsões. Essa média é o valor esperado das estimativas, conforme já abordado.
- Para medir a variabilidade das previsões, verificamos a diferença entre cada previsão individual e a média das previsões

Interpretação da variância:

- Baixa Variância: Indica que as previsões dos diferentes modelos são bastante consistentes entre si. O modelo é estável.
- Alta Variância: Indica que as previsões dos diferentes modelos variam significativamente. O modelo é sensível às flutuações nos dados de treinamento.
- **Em resumo, a variância é a diferença entre cada estimativa individual de um modelo treinado em um conjunto específico de treino de uma população e a média das estimativas de todos os modelos. É uma medida de quão dispersa cada estimativa está em relação à média das estimativas.**

Intuição

- Desenho adaptado do material de Sebastian Raschka:



Decomposição Viés-Variância

- **A decomposição viés-variância divide o erro esperado de um modelo em três componentes: Viés, Variância e Erro Irredutível.**
- Erro Irredutível: O erro que não pode ser reduzido por nenhum modelo, devido à variabilidade inerente nos dados.
- A decomposição de viés-variância é um conceito fundamental em aprendizado de máquina que ajuda a entender e analisar a performance de um modelo preditivo.

Erro Quadrático

- Mede a diferença ao quadrado entre o valor alvo verdadeiro y e a predição y^. $S = (y - \hat{y})^2$
- Em que:
  - y^ = f^(x)=h(x): Este é o valor previsto pelo modelo.
    - y^ é a predição feita pelo modelo f^(x), que pode ser representado também como h(x).
  - y = f(x): Este é o valor alvo ou verdadeiro que estamos tentando prever. y é a saída real, e f(x) é a função subjacente que mapeia as entradas x para o valor alvo y.

Passos da decomposição:

- 1º Passo: $(y - \hat{y})^2 = (y - \mathbb{E}[\hat{y}] + \mathbb{E}[\hat{y}] - \hat{y})^2$
- **Adicionar e subtrair E[y ^] no termo (y− e y ^) é uma técnica matemática que nos permite separar o erro total em componentes que podemos interpretar como viés e variância.**
- E[y^] é a expectativa (média) das predições do modelo.
- 2º passo:
  - Vamos expandir o termo (y−E[y ^]+E[y ^]− y^)² usando a fórmula do quadrado da soma (ou diferença):
    $(a + b)^2 = a^2 + b^2 + 2ab$
  - onde a= y−E[y ^] e b= E[y ^]− y ^.
  - Aplicando isso, temos: $(y - \mathbb{E}[\hat{y}])^2 + (\mathbb{E}[\hat{y}] - \hat{y})^2 + 2(y - \mathbb{E}[\hat{y}])(\mathbb{E}[\hat{y}] - \hat{y})$
- Entendendo Cada Termo:
  - (y−E[y^])²: Este termo mede a diferença entre o valor verdadeiro y e a média das predições E[y ^]. É o componente do viés ao quadrado.
  - (E[y^]−y^)²: Este termo mede a diferença entre a média das predições E[y^] e a predição individual y ^. É o componente da variância.
  - 2(y−E[y^])(E[y^]−y^): Este termo é o produto entre o viés e a variância, a expectativa deste termo é zero. Isto ocorre porque o termo estamos multiplicando:
    - y−E[y^] que é uma constante.
    - E[y ^]− y ^ ´que é uma variável que pode ser positiva ou negativa. E ao longo de muitos conjuntos de dados, as diferenças positivas e negativas se cancelam. Desta forma, ao tomar a expectativa deste termo sobre muitos conjuntos de dados, ele se torna zero devido ao cancelamento das variações positivas e negativas.
    - Portanto o resultado será zero, porque estamos multiplicando uma constante por zero e por dois e estermo é cancelado.

Expectativa do Erro Quadrático Médio (MSE)

- Agora temos a decomposição do MSE:
- MSE=(Viés)²+Variância $\mathbb{E}[(y - \hat{y})^2] = \mathbb{E}[(y - \mathbb{E}[\hat{y}])^2] + \mathbb{E}[(\mathbb{E}[\hat{y}] - \hat{y})^2]$
- Essa decomposição nos ajuda a entender o trade-off entre viés e variância e a analisar como melhorar o modelo.

Decomposição do erro no contexto da classificação

- A perda 0-1 é uma forma simples de medir a precisão de um modelo de classificação. Ela verifica se a previsão do modelo está correta ou não:
  - Se a previsão do modelo y^ for igual ao valor verdadeiro y, a perda é 0 (nenhuma perda).
  - Se a previsão y ^ for diferente do valor verdadeiro y, a perda é 1 (perda total).
- **Previsão Principal (Moda)**
  - A previsão principal é a classe que o modelo mais frequentemente prevê. Por exemplo, se o modelo faz várias previsões e a maioria delas é para a classe 'A', então 'A' é a previsão principal (moda).
  - Desta forma, a decomposição viés-variância utiliza a moda das previsões do modelo. A moda é a classe mais frequentemente predita pelo modelo, porque isso maximiza a precisão.

Viés:

- **No contexto da classificação, o viés é uma medida de quão frequentemente a classe mais predita pelo modelo (moda das previsões) está incorreta em relação ao valor verdadeiro.** Um alto viés indica que o modelo está sistematicamente errando, prevendo sempre uma classe que frequentemente não é a correta.

$$\text{Viés} = \begin{cases} 1 & \text{se } y \neq \hat{y}_{\text{moda}} \\ 0 & \text{caso contrário} \end{cases}$$

- O viés é uma função indicadora que é 1 se o valor verdadeiro e a previsão forem diferentes, e 0 caso contrário.

Variância:

- A variância mede a probabilidade de que uma previsão y^ seja diferente da previsão mais frequente. **Isso significa verificar o quão diferentes as previsões individuais do modelo são em comparação com a previsão mais comum (moda).** Alta variância indica que o modelo está muito sensível às mudanças nos dados de treinamento e faz previsões muito variadas.

$$\text{Variância} = P(\hat{y} \neq \hat{y}_{\text{moda}})$$

- Quando o viés é igual a zero, a perda total (Loss) é igual à variância:
  - Loss= Bias + Variance
  - Loss= 0 + Variance
  - Loss= Variance
  - Loss = P(y^ ≠ y^moda)
- Desta forma, se um modelo tem viés zero, sua perda é definida inteiramente pela variância.

- Quando o viés é igual a 1:

$$P(\hat{y} \neq y)$$

- Isso mede a probabilidade de a previsão y^ ser diferente do valor verdadeiro y.
- **A função de perda pode ser reescrita como:**

$$\text{Loss} = P(\hat{y} \neq y) = 1 - P(\hat{y} = y)$$

- Apenas eventos complementares, que medem a proporção de previsões erradas.

- Se y^moda é a previsão principal errada (com viés 1), temos que y ≠ y^ moda.
- Se a previsão principal (y^ moda) está errada, a probabilidade de uma previsão individual (y ^) ser diferente da previsão principal está relacionada à variância, ou seja a probabilidade de uma previsão estar correta pode ser definida como:
  - P(y^=y) ≈ P(y^≠y^moda)
- Usando o 'inverso' ('1 menos'), podemos então escrever a perda como:
  - Loss = P(y^ ≠ y) = 1 P(y^ = y) = 1 - P(y^ ≠ ymoda)
- Como o viés é 1, a perda total pode ser expressa como:

$$\text{Loss} = 1 - \text{Variance}$$

- **Desta forma, se o viés for igual a 1, aumentar a variância pode diminuir a perda, o que é uma observação interessante.**
- Exemplificando:
  - Imagine que estamos tentando classificar se uma imagem contém um gato ou um cachorro. Suponha que temos um modelo de classificação que faz previsões (y^) para várias imagens.
  - Sendo que y = cachorro
  - ymoda = gato
  - Portanto, o viés é 1, porque a previsão principal ymoda é diferente de y.
  - Suponha que inicialmente o modelo tem baixa variância. Ele sempre prevê 'gato', o que está errado para a maioria das imagens (que na verdade são 'cachorro').
  - Se aumentarmos a variância, o modelo começa a fazer previsões diferentes para diferentes imagens, em vez de prever sempre 'gato'.
  - Com maior variância, o modelo agora prevê 'gato' para algumas imagens e 'cachorro' para outras.
- **Desta forma, aumentar a variância pode mudar a fronteira de decisão do modelo, levando a algumas previsões corretas por 'sorte'.**