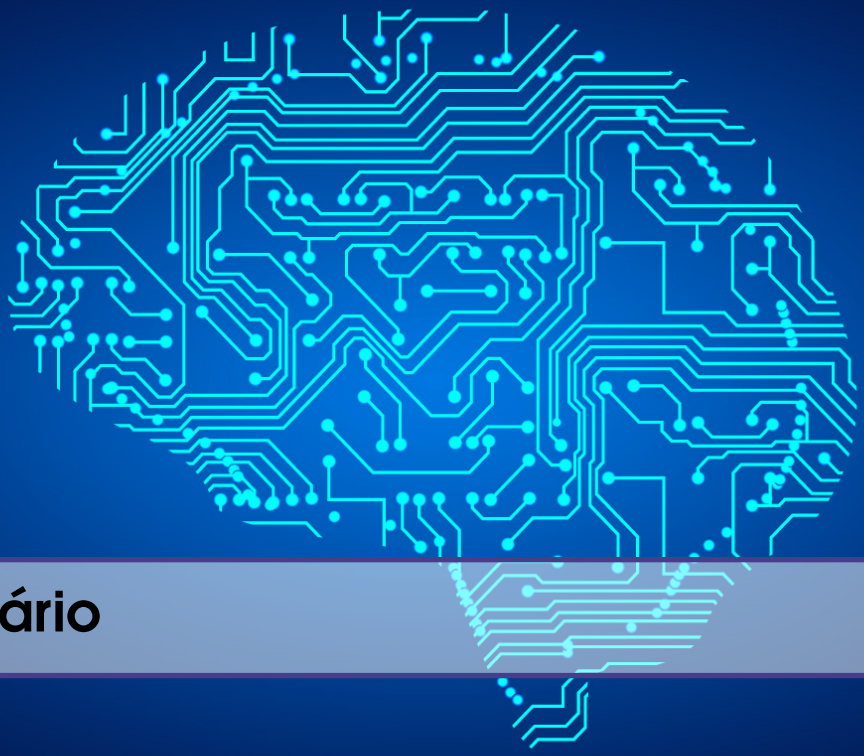
Lições de Machine Learning

Tópicos reunidos a partir de estudos

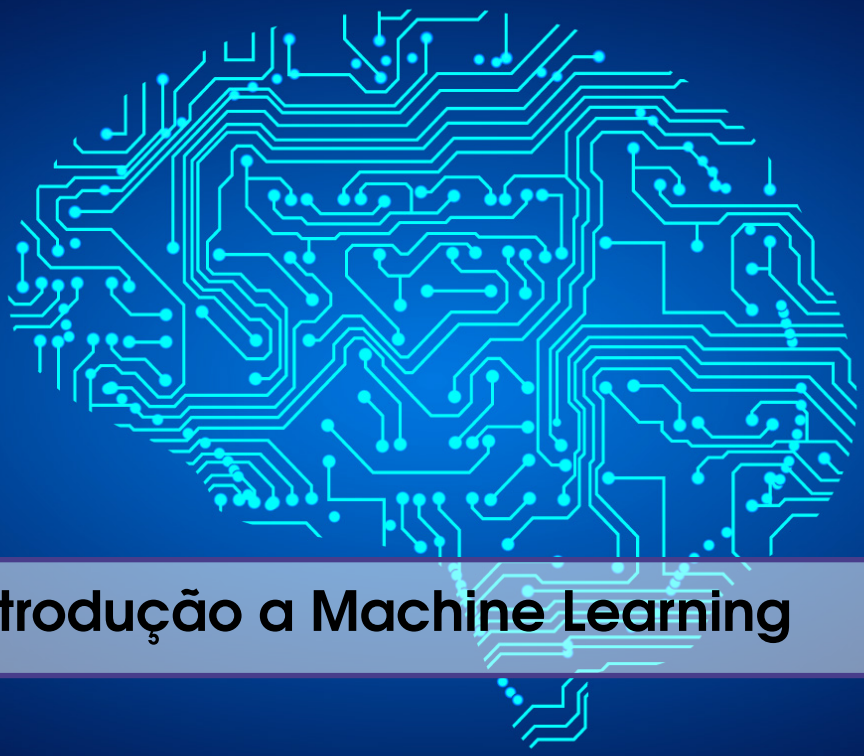
Thiago Panini

Versão 1, Agosto de 2019



Sumário

O Essencial em Aprendizado de Máquina



1. Introdução a Machine Learning

1.1 Motivação

Antes de dar início a essa jornada de aprendizado, é importante ressaltar que este é um documento criado com o objetivo de alocar os conhecimentos do autor aprendidos ao longo de cursos, estudos pessoais, eventos ou até mesmo conversas informais para facilitar a busca e a retomada de conceitos envolvendo Machine Learning e suas áreas relacionadas.

Neste documento, será possível encontrar definições teóricas, implementações práticas, experiências bem sucedidas e mal sucedidas, reunindo conteúdos úteis que descrevem toda uma trajetória de aprendizado.

O conteúdo aqui alocado terá como alicerce principal o curso *Aprendizagem Automática por Andrew Ng (Coursera)* e o livro *Hands-On Machine Learning with Scikit-Learn & TensorFlow por Aurélien Géron*. Demais referências utilizadas serão citadas ao longo da construção deste documento e poderão ser encontradas no tópico de Bibliografia.

1.2 Tipos de Aprendizado

O conceito de Aprendizagem Automática não é recente. Em 1959, Arthur Sammuel o define como sendo o campo de estudo que permite o aprendizazdo à computadores sem que estes sejam explicitamente programados.

A definição de Tom Mitchel, posta em 1977, defente que um programa de computador aprende com a experiência E, a respeito de uma tarefa T e com uma performance P, se sua performance em T, medida por P, melhora através da experiência E.

Isto pode soar um pouco confuso em uma primeira avaliação, porém é possível definir o conceito de Aprendizagem Automática como uma forma de determinar modelos capazes de aprender a partir de dados. Para aplicar tal conceito, tem-se quatro principais grupos:

1. Aprendizado Supervisionado
2. Aprendizado Não-Supervisionado
3. Aprendizado Semi-Supervisionado
4. Aprendizado por Reforço

1.2.1 Aprendizado Supervisionado

No aprendizado supervisionado, o target do problema de negócio a ser resolvido é fornecido ao algorítmo. Em outras palavras, ao receber um conjunto de dados contendo os atributos necessários para descrever o modelo de negócio ao qual o algoritmo de Machine Learning será aplicado, a variável resposta do modelo poderá ser identificada, fornecendo assim insumos suficientes para que o algoritmo literalmente aprenda com o conjunto de dados.

O resultado é um modelo treinado capaz de generalizar e gerar predições para quaisquer outros dados de entrada referentes ao contexto inserido no treinamento. Abaixo, serão destacados alguns exemplos de aplicações de aprendizado supervisionado:

- Predição de preços de casas
- Classificação de spam
- Diagnóstico de câncer
- Classificação de imagens
- Entre outros

Por fim, no contexto de aprendizagem supervisionada, um algorítmo é treinado através do fornecimento de *inputs* e *outputs* dentro de um conjunto de dados, ou seja, o treinamento é baseado em uma resposta conhecida. Para treinar estes algoritmos, é necessário fornecer os elemento preditores *(features)* e também as soluções *(labels)*. Alguns exemplos:

- Linear Regression
- Logistic Regression
- k-Nearest Neighbors
- Support Vector Machines (SVMs)
- Decision Trees and Random Forest
- Neural Networks

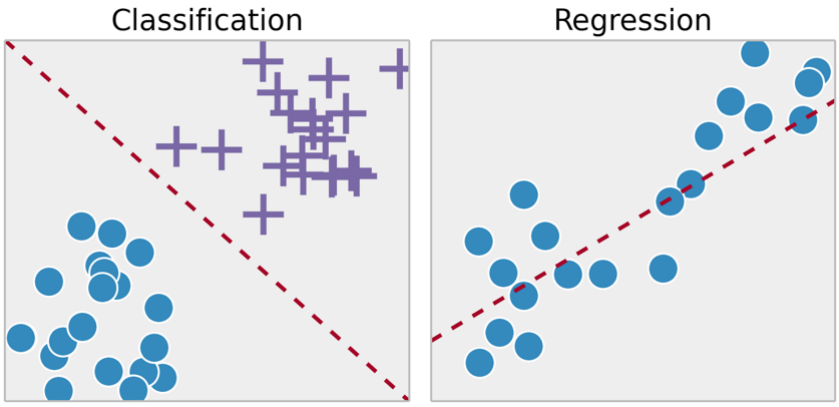


Figura 1.1: Exemplos de Aprendizado Supervisionado

1.2.2 Aprendizado Não-Supervisionado

Diferente do que foi visto na sessão anterior, no aprendizado não-supervisionado as saídas não são fornecidas ao modelo, tendo este a responsabilidade de identificar *padrões*. Neste modelo de aprendizagem, as soluções *(labels)* não estão bem definidas. Técnicas de *clustering* estão altamente ligadas ao aprendizado não supervisionado pois, dado um conjunto de dados, o algorítmo associado é capaz de identificar e agrupar padrões, sem que sejam fornecidos dados prévios e tampouco o nome desses grupos.

É possível separar o contexto de problemas com dados não rotulados em campos de aprendizagem, sendo estes:

- Clusterização
  - k-Means
  - Hierarchical Cluster Analysis (HCA)
  - Expectation Maximization
- Visualização e Redução de Dimensionalidade
  - Principal Component Analysis (PCA)
  - Kernel PCA
  - Locally-Linear Embedding (LLE)
  - t-distributed Stochastic Neighbor Embedding (t-SNE)
- Association Rule Learning
  - Apriori
  - Eclat

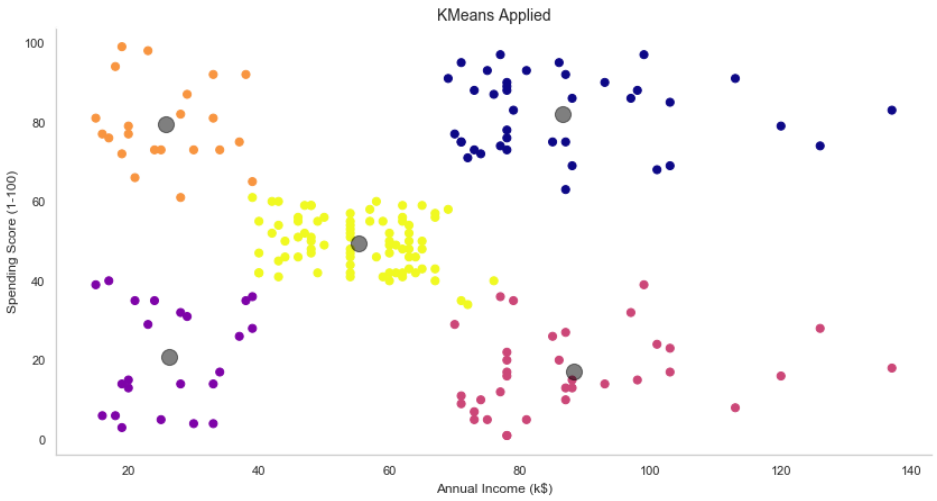


Figura 1.2: Aplicação K-Means: Aprendizado Não-Supervisionado

1.2.3 Aprendizado Semi-Supervisionado

O aprendizado semi-supervisionado, como o nome sugere, é aquele onde as soluções são dadas parcialmente, deixando a cargo do próprio algorítimo realizar identificações e correlações. Um bom exemplo deste tipo de aprendizado é o Google Photos, onde o algorítimo retorna, com eficiência, rostos semelhantes (clustering não-supervisionado), sendo somente necessário dizer a quem pertence a foto (label supervisionado).

Sendo assim, dentro do mesmo contexto, o marcador de fotos do Facebook também pode ser utilizado como exemplo. A parcela não-supervisionada do algorítimo é responsável pela identificação de pessoas na foto, sem necessariamente saber quem são essas pessoas. Já a parcela supervisionada se dá por conta da identificação, por parte do usuário, do rosto presente na foto.

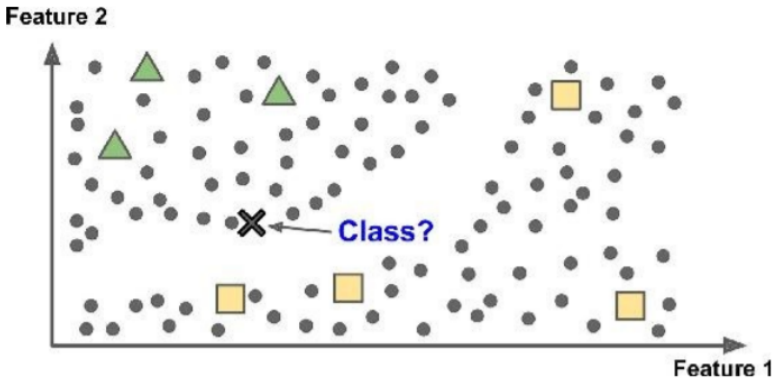


Figura 1.3: Exemplo de Aprendizado Semi-Supervisionado

1.2.4 Aprendizado por Reforço

Pode-se dizer que o aprendizado por reforço é o que mais se difere entre os tipos de aprendizado. Nele, o sistema de aprendizado (chamado de agente), consegue observar o ambiente, selecionar e performar ações, coletando *recompensas* ou *penalidades* de acordo com as ações tomadas. Por conta próprio, através do retorno obtido, o modelo aprende através de uma estratégia, chamada *policy*, cujo objetivo é coletar o maior número de recompensas. Exemplos:

- Programa alphaGo da DeepMind
- Carros autônomos
- Games/inteligências desenvolvidos para nunca serem derrotados

1.3 Aplicação de Modelos

Este tópico irá tratar, de forma breve e resumida, sobre aplicações de modelos de Machine Learning em situações práticas e bem definidas.

1.3.1 Batch Learning

No tipo de implementação conhecido como *Batch Learning*, o treinamento é realizado offline em grandes lotes de dados, gerando um elevado consumo de recursos computacionais. Nesse contexto, novos dados exigem novas versões do modelo, tornando o aprendizado inviável em casos dinâmicos e sendo aplicável em:

- Modelos fixos
- Treinamentos sem atualizações constantes

1.3.2 Online Learning

Diferente do aprendizado por lote, o termo *Online Learning* refere-se a modelos alimentados sequencialmente onde o algoritmo aprende conforme a entrada de novos dados em um fluxo contínuo de recebimento de insumos para o treinamento. Pode-se afirmar que, nessa abordagem, os recursos são otimizados. Algumas aplicações:

- Modelos que utilizam IoT
- Leitura de sensores
- Dados de vendas com rápidas atualizações

1.3.3 Instance-based Learning

Um sistema de predição baseado na instância *(Instance-based Model)* refere-se a um sistema que generaliza novos dados a partir de medições de similaridade. Graficamente, é possível imaginar que uma nova instância, ou seja, um novo ponto no gráfico, é classificada de acordo com a proximidade em relação aos pontos já classificados pelo modelo. Exemplificando através do contexto de classificação de *Spam*, é possível dizer que o modelo identifica e-mails maliciosos a partir da similaridade entre e-mails desse tipo.

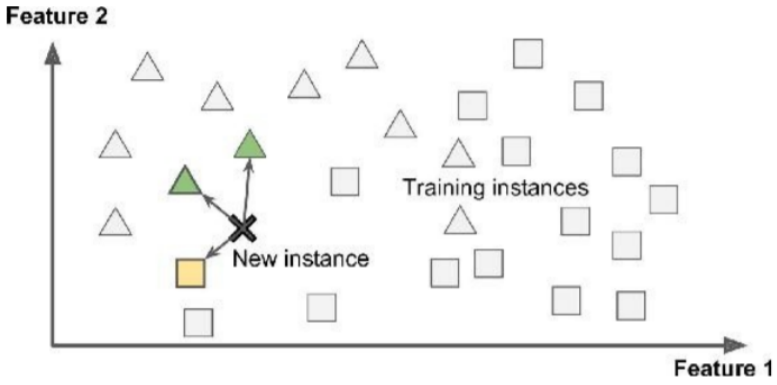


Figura 1.4: Instance-based Learning

1.3.4 Model-based Learning

Já uma implementação baseada no próprio modelo *(Model-based Learning)* diz respeito ao treinamento de um modelo e a definição de uma fronteira de decisão para generalização de novas instâncias. Dessa forma, as predições do modelo indicam classificações ou posições da amostra dentro do espaço.

1.4 Álgebra Linear e Probabilidade

Conceitos de cálculo, álgebra linear, estatística e teoria da probabilidade são essenciais no universo de Machine Learning e, portanto, visando uma maior facilidade de aprendizado, o link para o portal *CS229: Machine Learning* contempla o conteúdo necessário (e um pouco além) para a sequência dos tópicos.

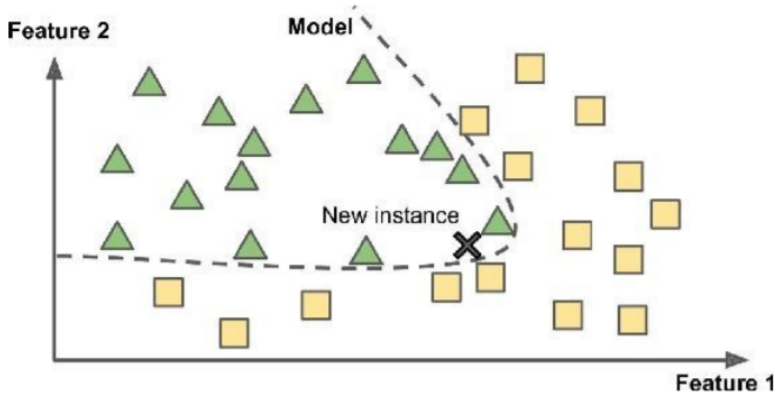


Figura 1.5: Instance-based Learning

1.5 Exercícios

Os exercícios a seguir foram retirados do livro *Hands-On Machine Learning with Scikit-Learn & TensorFlow*. A resposta para alguns deles irá depender de pesquisas em outras fontes, dado que este documento engloba um conteúdo resumido e compactado sobre estudos pessoais.

Exercício 1.1 Como você define Machine Learning? ■

Há várias definições plausíveis para Machine Learning porém, generalizando, é possível concluir que o aprendizado de máquina nada mais é que a habilidade programacional de computadores em reconhecer padrões, tomar decisões e aprender com os erros, sem que estas ordens sejam explicitamente dadas. De acordo com Tom Mitchell (1977), é dito que um programa de computador aprende com a experiência E a respeito de uma tarefa T com uma performance P, se sua performance em T, medida por P, melhora através da experiência E.

Exercício 1.2 Defina quatro problemas onde Machine Learning é aplicável ■

- Classificação de padrões para predição de tumores malignos
- Predição de preços de casas através de uma função contínua
- Processamento de linguagem natural NLP
- Sistemas de recomendação (Netflix)

Exercício 1.3 O que é um conjunto de treinamentos com solução *(labeled training set)*? ■

Trata-se de um conjunto de dados que possuem "solução"ou *target*, indicando assim qual a resposta almeada pelos dados de entradas, definidos pelas *features*. Tais conjuntos são utilizados em treinamentos de aprendizado de máquina supervisionado.

Exercício 1.4 Quais são os dois modelos mais comuns em aprendizado supervisionado? ■

Os modelos mais comuns de aprendizado supervisionado são modelos de regressão e de classificação. Neles, são preditos, respectivamente, valores que dependem de funções contínuas e discretas.

Exercício 1.5 Mencione quatro exemplos de aprendizado não-supervisionado (modelo e aplicação). ■

Os modelos mais comuns de aprendizado supervisionado são modelos de regressão e de classificação. Neles, são preditos, respectivamente, valores que dependem de funções contínuas e discretas.

Exercício 1.6 Que tipo de algorítimo de Machine Learning pode ser utilizado para agrupar clientes em múltiplos grupos? ■

Para resolver este problema, o mais adequado é um algorítimo de aprendizagem não supervisionada, mais especificamente de clusterização. Dessa forma, o modelo pode automaticamente reconhecer padrões específicos de clientes e agrupa-los sem que haja uma definição específica dos parâmetros selecionados.

Exercício 1.7 Que tipo de algorítimo pode ser utilizado para permitir que um robô caminhe em terrenos desconhecidos? ■

Neste caso, o mais adequado é utilizar os conceitos propostos pelo Aprendizado por Reforço ou simplesmente Reinforcement Learning. Dessa forma, o robô pode trabalhar com recompensas e penalidades, se policiando em sua estratégia para sempre buscar recompensas, mudando sua forma de atuar caso encontre uma penalidade. Ao longo do caminho o robô poderá aprender a seguir o melhor caminho possível.

Exercício 1.8 O problema de detecção de Spam pode ser classificado como um algoritmo do tipo aprendizado supervisionado ou não-supervisionado? ■

Um modelo de detecção de Spam pode ser descrito como uma situação de Aprendizagem Supervisionada, visto que os outputs (ou labels) são fornecidos para os e-mails, indicando quais são spam e quais não são.

Exercício 1.9 O que é um método de aprendizagem online (Online Learning)? ■

Um sistema de aprendizagem online é aquele treinado a partir de uma gigantesca massa de dados, dividida em pequenas partes que são inseridas no algorítimo pouco a pouco. Dessa forma, os dados de treinamento são divididos em mini-batches e inseridos no modelo sequencialmente. Com estes sistemas, há uma imensa facilidade na adaptação a novos dados e alterações futuras.

Exercício 1.10 O que é *out-of-core* learning? ■

Out-of-core learning é o nome dado ao processo de "economia"de memória proposto por sistemas de aprendizagem online. Isso se dá devido a partição do conjunto de dados em pequenos "pedaços"que são sequencialmente treinados sem que haja relevantes custos operacionais e de memória.

Exercício 1.11 Qual é a diferença entre um parâmetro do modelo e um hiperparâmetro do algorítmo? ■

Um hiperparâmetro é um parâmetro do algorítimo de machine learning (e não do modelo) e tem a função de mensurar a regularização a ser aplicada no modelo durante a fase de treinamento. Para clarear as definições, tem-se por regularização a simplificação do modelo (espécie de restrição do

escopo) para reduzir o risco de overfitting, ou seja, o treinamento demasiado.

- Parâmetros do modelo: θ_1 e θ_2, por exemplo;
- Hiperparâmetros: definem a regularização e o comportamento do modelo.

Exercício 1.12 Quais são as principais caracterísitcas de sistemas de aprendizado baseados em modelo? Qual a estratégia mais comum utilizada por eles? Como é realizada a predição? ■

Os modelos que não são baseados em instância, são conhecidos como Model-Based Learning e, entre suas principais características, encontra-se o treinamento padrão em um conjunto de dados através de um modelo ou algorítimo de Machine Learning. Dessa forma, são definidos o tipo de algorítimo, levantados os parâmetros do modelo, *Função Custo* (mede eficiência do modelo) e uma série de outras tarefas bem conhecidas. A diferença para o método Instance-Based Learning é que, neste caso, as predições são realizadas de acordo com o modelo (por exemplo, uma regressão linear) e não através das instâncias/pontos já presentes (por exemplo, kNN). Pode haver relativa diferença entre estes modelos aplicados a um mesmo problema. Em resumo, um aprendizado baseado em modelo segue:

- Estudo dos dados
- Seleção do modelo
- Treinamento dos dados (levantamento de parâmetros que minimizam a Função Custo)
- Aplicação do modelo para predições (inferência)

Exercício 1.13 Cite os quatro principais desafios de um modelo de Machine Learning ■

- Quantidade dos dados
  - Para resolver um problema utilizando Machine Learning com eficiência, deve-se ter em mãos uma quantidade de dados razoável, tornando o modelo aplicável e garantindo resultados através da generalização.
- Representatividade e qualidade dos dados
  - Em outra vertente, uma grande massa de dados de baixa qualidade também não se faz válida. Os dados devem representar o comportamento real do problema a ser analisado. Paralelamente, um conjunto de treinamento com diversos *outliers* ou ruídos dificultará o reconhecimento de padrões por parte do algorítmo, ocasionando resultados que não demonstram a realidade.
- Features irrelevantes
  - É extremamente importante que haja uma avaliação das variáveis preditoras de maior importância para o modelo. Atributos ou features sem relevância para o processo acabam atuando de forma negativa nos resultados, ocupando espaços e interferindo nas métricas resultantes.
- Overfitting
  - Este é um grande desafio enfrentado durante o treinamento de um modelo. A separação co conjunto de dados em treinamento e teste é uma maneira de se alcançar melhores resultados. Entretanto, a vaidade em se alcançar resultados magníficos durante o trenimaneto pode ocasionar o fenômeno conhecido como *overfitting* que, por sua vez, está relacionado à falta de generalização do modelo para dados que não fazem parte do conjunto utilizado no treinamdno. Em outras palavras, um modelo com *overfitting* é perfeitamente capaz de reconhecer padrões presentes no conjunto de treinamento, porém extremamnete ineficaz no reconhecimento de qualquer dado externo.

Exercício 1.14 Se o modelo treinado está obtendo uma boa performance nos dados de treino, porém má generalização em novas instâncias, o que está acontecendo? Cite três possíveis soluções. ■

Como mencionado na questão 14, trata-se do fenômeno conhecido como *overfitting*, onde o modelo é tão bem treinado no conjunto de treinamento que passa a atuar de forma efetiva apenas em dados que contém esse conjunto, performando de forma inadequada para dados externos. Algumas possívei soluções:

- Simplificação do modelo através da seleção de uma quantidade menor de parâmetros theta, ou seja, aplicando uma redução no grau polinomial da função
- Coleta de uma quantidade maior de dados de treinamento (auxilia na generalização)
- Redução de ruídos no set de treinamento (remoção de erros e outliers)

Exercício 1.15 O que é um conjunto de testes e por que ele deve ser usado? ■

Na fase de treinamento de um modelo de Machine Learning, normalmente o conjunto total de dados é dividido em dois subconjuntos: treinamento e teste. Isto é essencial para que uma estira de testes seja implementada com eficiência, sendo possível checar se o treinamento foi realmente efetivo através do levantamento de métricas utilizando tais dados externos. com isso, é possível verificar se o modelo está generalizando bem e se os resultados estão plausíveis. Normalmente, utiliza-se 80% do conjunto total para treinamento e 20% para testes, porém esse número pode variar de acordo com o problema a ser resolvido.

Exercício 1.16 Qual é o propósito de um conjunto de validação? ■

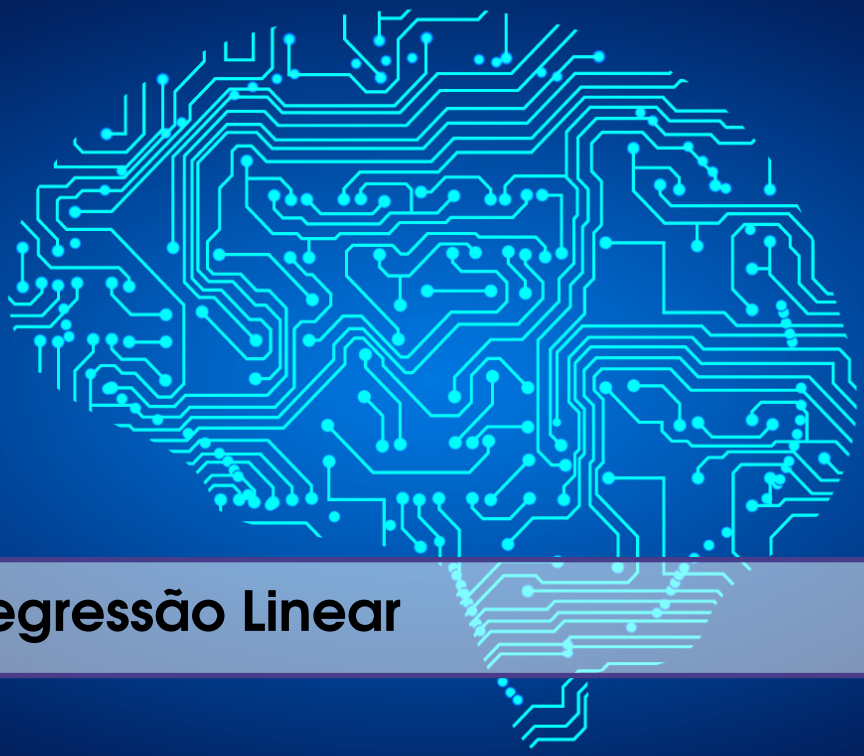
Um conjunto de validação é importante para que seja possível se certificar do nível de generalização do modelo de Machine Learning aplicado. Em outras palavras, é pssível medir como o algoritmo se comporta ao receber dados externos aos dados de treinamento, evitando assim possíveis problemas de *overfitting*.

Exercício 1.17 O que pode dar errado ao ajustar os hiperparâmetros do modelo utilizando o conjunto de testes? ■

Ajustar os hiperparâmetros do modelo repetidas vezes no set de treinamento ocasiona uma má generalização do modelo, fazendo com que este apresente performances consideradas ruins quando apresentado a novos conjuntos. Para sanar este tipo de problema, é comum criar um outro conjunto de dados chamado de *validation set* ou conjunto de validação. Este, por sua ve, nad amais é que a separação do conjunto de treino em partes complementares. Uma vez definidos os hiperparâmetros, o teste é realizado através do cruzamento destes pequenos conjuntos, onde cada um funciona como teste e treino pelo menos uma vez.

Exercício 1.18 O que é validação cruzada e por que este método é preferível ao utilizar um set de validação? ■

Como mencionado na questão anterior, a validação cruzada (ou *cross validation*) é o processo de "quebra"do dataset em pequenos conjuntos complementares, realizando o treinamento do modelo de modo a cruzar todos estes pequenos conjuntos, validando o resultado com a parte remanescente. É preferível realizar este processo pois assim os dados são aproveitados com mais eficiência.



2. Regressão Linear

Introduzindo os algorítmos de Machine Learning, neste capítulo serão abordados detalhes sobre Regressão Linear. Entre os ensinamentos, serão tratados tópicos referentes a implementação do modelo, funções e equações utilizadas para representá-lo e otimizá-lo, além do entendimento das métricas de avaliação de performance de um modelo treinado.

2.1 Representação do Modelo

Definição 2.1.1 — Equação Regressão Linear. Dadas as features x_i e os parâmetros θ_i do modelo, a variável target y é definida por:

$$\mathbf{y} = \theta_0 + \theta_1 x_1 + \theta_2 x + 2 + ... + \theta_n x_n \tag{2.1}$$

$$\mathbf{y} = h_\theta(x) = \theta^T.x \tag{2.2}$$

Definição 2.1.2 — MSE (Erro Médio Quadrático).

$$\mathbf{MSE}(\mathbf{X}, \mathbf{h}_\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(\theta^T.x^{(i)} - y^{(i)}\right)^2 \tag{2.3}$$

Definição 2.1.3 — Normal Equation.

$$\theta = \left(X^T.X\right)^{-1}X^T.y \tag{2.4}$$

As definições acima servem de base para o entendimento do modelo de Regressão Linear, desempenhando papel fundamental nos conceitos a serem abordados posteriormente.

2.2 Gradiente Descendente

O Gradiente Descendente tem como objetivo a minimização da função de custo definida, ajustando os parâmetros do modelo de forma iterativa utilizando derivadas parciais.

> **Corolário 2.2.1 — Montanha Nebulosa.** Imagine-se preso no pico de uma montanha onde só há névoa ao seu redor. Você somente é capaz de sentir o declive da montanha em seus pés e seu objetivo é chegar até o solo. O Gradiente Descendente atua de forma semelhante ao "caminhar", passo a passo, rumo ao mínimo da função.

Dependências: *learning rate* e derivadas parciais
Importante: normalização dos dados

Uma outra forma de minimizar a função custo de um modelo de Regressão Linear é utilizando a *Normal Equation* (definida em 2.1.3 e utilizada como padrão na classe *LinearRegression()* do Scikit-Learn). Algunas vantagens e desvantagens no uso dessas duas abordagens podem ser visualizadas na tabela abaixo:

Gradiente Descendente	Normal Equation
Necessário definir α	Não precisa definir α
Muitas iterações	Cálculo vetorizado
Custo O(kn^2)	Custo O(n^3) - matriz inversa
Adequado com n elevado	Lento para n elevado

Tabela 2.1: Comparação Gradiente Descendente e Normal Equation

Dentro do universo de Gradiente Descendente, existem certas variações relacionadas a forma de implementação e o alcance do mínimo global da função custo. A seguir, essas abordagens serão tratadas e os detalhes de cada uma delas serão evidenciados.

2.2.1 Batch Gradient Descendent

As *derivadas parciais* são fatores essenciais para a implementação do Gradiente Descendente. A ideia é calcular a mudança da função custo caso haja a alteração de um determinado parâmetro em uma magnitude extremamente baixa. Uma boa maneira de entender este conceito é questionar: "qual o declive da montanha abaixo dos meus pés se eu estiver olhando para o **leste**? E olhando para o **norte**?". E assim essa pergunta pode ser repetida em qualquer um dos xontextos para qualquer uma das n-dimensões.

> **Definição 2.2.1 — Derivada Parcial MSE.**

$$\frac{\partial}{\partial \theta_j}\mathbf{MSE}(\theta) = \frac{2}{m}\sum_{i=1}^{m}\left(\theta^T.x^{(i)} - y^{(i)}\right)x_j^{(i)} \tag{2.5}$$

Otimizando os cálculos, é possível obter um vetor com todos os gradientes de cada um dos parâmetros:

> **Definição 2.2.2 — Otimização Derivada Parcial MSE.**

$$\nabla_\theta \mathbf{MSE}(\theta) = \frac{2}{m}X^T.(X.\theta - y) \tag{2.6}$$

Corolário 2.2.2 — Cálculo em Lote. A fórmula para calcular as derivadas parciais com o Batch Gradient Descent envolve a utilização de todo o conjunto de dados X em cada um dos passos iterativos. Por esse motivo, esse algoritmo é chamado de "Batch"e, como resultado, temos um grande custo computacional. Apesar disso, o algoritmo de Batch Gradient Descent escala melhor com um grande número de features se comparado a Normal Equation.

Uma vez calculadas as derivadas parciais, para que o algoritmo cumpra seu objetivo e encontre os melhores parâmetros para função, basta realizar uma subtração entre os parâmetros theta e o resultado do cálculo, visto que as derivadas indicam a direção e a magnitude do passo a ser tomado em busca do mínimo local. É neste ponto que o *learning rate* entra em ação para definir o quão largo será este passo.

Definição 2.2.3 — Learning Rate.

$$\theta^{(\text{next step})} = \theta - \alpha \nabla_\theta MSE(\theta) \tag{2.7}$$

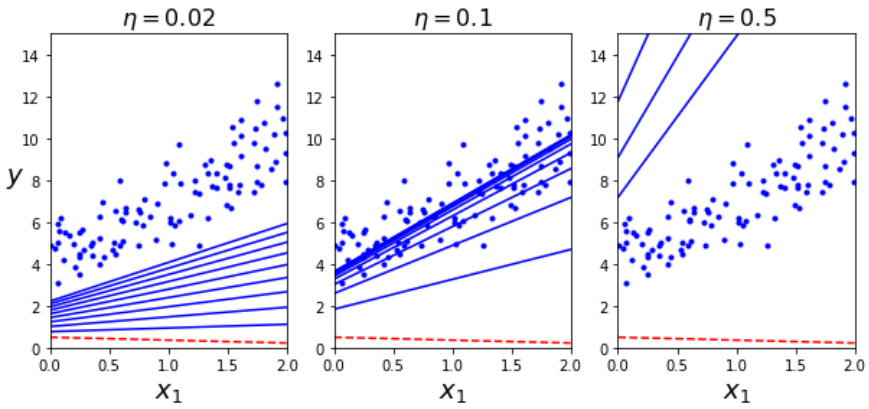


Figura 2.1: Comparação de Learning Rates

Caso 1 ($\alpha = 0.02$)**:** Learning rate muito brando. Convergência em velocidade reduzida.
Caso 2 ($\alpha = 0.1$)**:** Valor ótimo para o learning rate.
Caso 2 ($\alpha = 0.5$)**:** Learning rate muito alto. Algorítmo não converge.

2.2.2 Stochastic Gradient Descendent

Corolário 2.2.3 — Processo Estocástico. A principal ideia do Stochastic Gradient Descent é evitar o custo computacional causado pela utilização de todo o conjuto de dados em cada passo do gradiente (Batch Gradient Descent) e, para tal, coleta aleatoriamente um único exemplo do conjunto de treino e o utiliza para o cálculo do gradiente em cada passo.

2.2.3 Mini-Batch Gradient Descendent

Corolário 2.2.4 — Pequenos Lotes. De fato, a utilização de todo o conjunto de dados para o cálculo do gradiente é computacionalmente oneroso (Batch). Por outro lado, a utilização de um único exemplo dos dados para este cálculo também pode não ser preciso (Stochastic). Sendo

> assim, o Mini-Batch Gradient Descent propõe a utilização de um pequeno conjunto de dados para a realização deste cálculo.

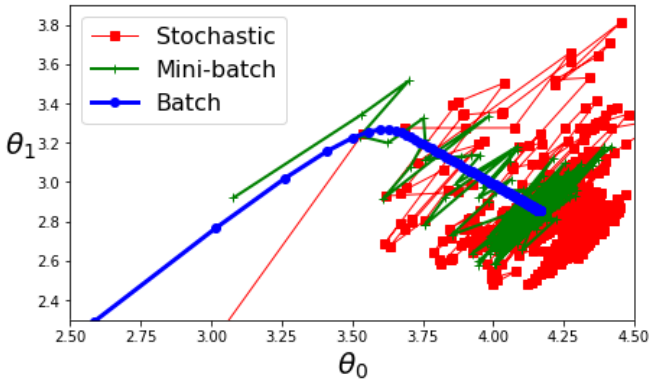


Figura 2.2: Batch, Stochastic e Mini-Batch Gradient Descent

2.3 Regressão Polinomial

Como o próprio nome sugere, a função que define um problema de Regressão Polinomial é composta por polinômios de grau n. Dessa forma, o curva resultante pode se ajustar a dados que não seguem necessariamente um comportamento linear.

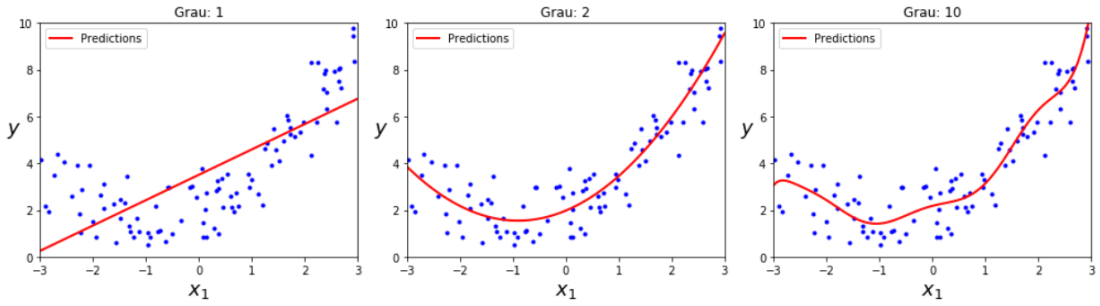


Figura 2.3: Modelos com Diferentes Graus Polinomiais

A imagem acima identifica os efeitos do aumento do grau polinomial em algoritmos de Regressão. É possível notar, extrapolando para equações com um elevado grau polinomial, a possível presença do fenômeno de *overfitting* do modelo, dado que este apresenta um super-ajuste nos dados.

Equação em sua forma geral:

$$y = h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + ... + \theta_m x^m \tag{2.8}$$

De fato, é possível obter diferentes curvas através do aumento do grau polinomial do modelo. Entretanto, é necessário atentar-se a presença de outliers que, eventualmente, farão com que a curva do modelo se ajuste de tal forma a causar um overfitting nos dados.

Utilizando o Scikit-learn, é possível aumentar o grau polinomial do modelo utilizando a classe PolynomialFeatures() (pode ser inserida em um Pipeline de preparação com diferentes graus). Um exemplo pode ser encontrado abaixo.

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

# Transformando dados em um grau polinomial = d
poly_features = PolynomialFeatures(degree=d)
X_poly = poly_features.fit_transform(X_train)

# Treinando um modelo
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y_train)
```

2.4 Regularização

Como visto na sessão sobre Regressão Polinomial, nem sempre a reta pode ser tratada como a curva adequada ao problema de negócio. Alguns conjuntos de dados possuem comportamentos característicos de curvas com graus polinomiais elevados. Entretanto, é preciso adaptar os parâmetros da equação para que, ao mesmo tempo que se desenvolva uma correta representação do modelo, este não perca a generalização para novas instâncias não utilizadas no treinamento.

Em um modelo linear, regularização é basicamente a restrição dos pesos dos parâmetros. Para isso, existem três formas diferentes de se aplicar tal procedimento:

- Ridge Regression
- Lasso Regression
- Elastic Net

2.4.1 Ridge Regression

Também conhecida como *Tikhonov Regulariation*, o modelo de Ridge Regression é uma forma regularizada da Regressão Linear com um termo de regularização igual a:

$$\alpha \sum_{i=1}^{n} \theta_i^2 = l_2 \text{ penalty} \tag{2.9}$$

Este termo, adicionado a função custo, força o modelo a não apenas se enquadrar aos dados, mas também manter os pesos do modelo tão baixos quanto possível. Este termo de regularização somente deve ser adicionado a função custo durante o treinamento. Uma vez treinado o modelo, sua performance deve ser medida sem regularização.

$$J(\theta) = MSE(\theta) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2 \tag{2.10}$$

$$\begin{cases} \alpha = 0 & \text{Ridge Regression = Linear Regression} \\ \alpha = alto & \text{Alta penalização nos pesos e redução no overfitting} \\ \alpha = baixo & \text{Baixa penalização nos pesos e aumento do overfitting} \end{cases} \tag{2.11}$$

É importante aplicar a normalização dos dados (Standard Scaler) antes de performar o modelo Ridge Regression. O termo de regularização aplicado, representa uma penalidade do tipo l_2 que, por sua vez, tem como principal característica, a penalização elevada (quadrado) a medida que a magnitude dos parâmetros cresce.

2.4.2 Lasso Regression

Também conhecido como *Least Absolut Shrinkage and Selection Operator Regression*, o modelo *Lasso Regression* é uma outra versão da Regressão Linear, tendo a esta uma adição de um termo de regularização sob a forma l_1.

$$J(\theta) = MSE(\theta) + \alpha \sum_{i=1}^{n} |\theta_i| \tag{2.12}$$

Diferente do Ridge Regression, o Lasso Regression aplica uma penalização do tipo l_1 que, por sua vez, faz com que os coeficientes das features de menor importância tenham tendência a zerar. Portanto, é possível dizer que este tipo de penalidade pode atuar bem como uma *feature selection* no caso de haver muitas features. Link de referência: towardsdatascience.com/l1-and-l2

2.4.3 Elastic Net

Pode-se dizer que o modelo Elatic Net é um meio termo entre o Ridge Regression e o Lasso Regression. Além do termo de regularização α presente nos outros dois modelos, o Elastic Net traz um parâmetro adicional r responsável por dosar essa mistura entre o Ridge e o Lasso Regression.

- Quando $r = 0$, o Elastic Net é equivalente ao Ridge Regression
- Quando $r = 1$, o Elastic Net é equivalente ao Lasso Regression

$$J(\theta) = MSE(\theta) + r\alpha \sum_{i=1}^{n} |\theta_i| + \frac{1-r}{2} \alpha \sum_{i=1}^{n} \theta_i^2 \tag{2.13}$$

Corolário 2.4.1 — Ridge, Lasso ou Elastic Net?. Normalmente, é preferível pelo menos um pouco de regularização e, portanto, o algoritmo de Regressão Linear pode ser descartado em um primeiro momento. Ridge é um bom modelo pra ser tratado como default mas, se houver uma suspeita de que apenas algumas features são úteis, é preferível partir pro Lasso ou Elastic Net. No geral, Elastic Net é preferível em relação ao Lasso, dado que este último pode se comportar erroneamente quando o número de features ultrapassa o número de exemplos ou então quando muitas features estão correlacionadas.

2.4.4 Early Stopping

O conceito de *Early Stopping* está relacionado a uma forma diferente de aplicar regularização de modelos iterativos como o Gradiente Descendente. A ideia é parar o treinamento assim que o erro de validação atinge um ponto mínimo.

Uma forma de entender esse conceito, é imaginar o debug do Gradiente Descendente, relacionando o erro RMSE do modelo nos dados de teste e validação a partir do aumento do número de iterações. Obviamente, a medida que as iterações crescem, o modelo vai aprendendo e o erro RMSE nos dados de treino e validação diminui gradativamente.

Contudo, existe um determinado momento onde o erro de validação começa a aumentar. Neste ponto, o modelo começa a sofrer com *overfitting*. Parar o treinamento neste ponto, significa aplicar o *early stopping*.

2.5 Métricas de Avaliação

Como visto nas sessões anteriores, o modelo de Regressão Linear possui uma função custo onde pode-se aplicar um fator de regularização α e penalidades seguindo as normas l_1 e l_2. Entretanto, como é possível avaliar a eficiência de um modelo treinado? Nos tópicos subsequentes, serão mostradas algumas métricas de avaliação de modelos de Regressão, sendo eventualmente colocados exemplos utilizando a linguagem Python e a biblioteca Scikit-Learn.

Figura 2.4: Definição de Erro

2.5.1 MAE - Mean Absolute Error

MAE é a média das diferenças absolutas entre as predições e o target real. Trata-se de um cálculo residual para cada ponto onde apenas o valor absolutado é coletado, evitando assim o cancelamento dos resíduos positivos e negativos. Efetivamente, o MAE descreve a magnitude média dos resíduos.

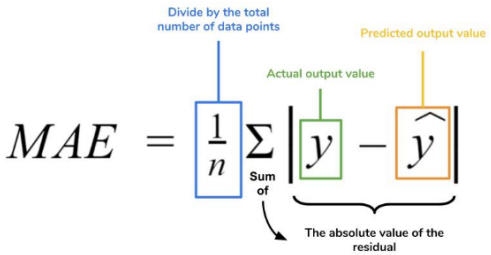


Figura 2.5: Equação - Mean Absolute Error

O erro médio absoluto (MAE) é uma das métricas de avaliação mais intuitivas, uma vez que se observa apenas a diferença entre o target e a predição. Entretanto, é importante citar que essa é

uma métrica linear onde cada erro, sendo este elevado ou não, contribui linearmente para a métrica final. Em outras palavras, não há nenhuma penalização adicional para *outliers*.

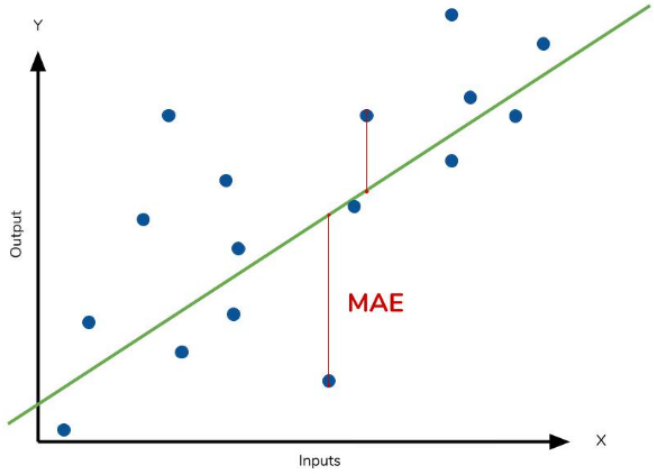


Figura 2.6: Interpretação Gráfica - Mean Absolute Error

Calculando MAE utilizando a linguagem Python (link:documentacaoMAE):

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error

lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train)

# Calculando Erro Medio Absoluto (MAE) no sklearn
test_pred = lin_reg.predict(X_test)
mae = mean_absolute_error(y_test, test_pred)

# Calculando MAE na unha
mae_sum = 0
for X, y in zip(X_test, y_test):
    prediction = lin_reg.pred(X)
    mae_sum += abs(y - prediction)
mae = mae_sum / len(y_test)
```

2.5.2 MSE - Mean Squared Error

O erro quadrático médio (MSE) tem uma filosofia semelhante ao erro médio absoluto (MAE), com a diferença de que os erros são elevados ao quadrado antes de aplicada a soma.

$$MSE = \frac{1}{n}\Sigma\underbrace{\left(y - \widehat{y}\right)}^{2}$$

The square of the difference between actual and predicted

Figura 2.7: Equação - Mean Squared Error

Diferente da contribuição proporcional vista no MAE, os erros contribuem de forma quadrática no MSE. Isto implica em uma maior importância dada aos *outliers* do modelo, aumentando assim a penalização a estes casos.

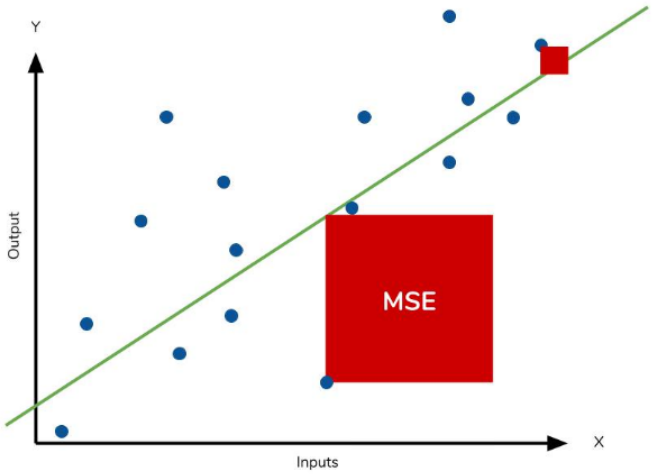


Figura 2.8: Interpretação Gráfica - Mean Squared Error

Calculando MSE utilizando a linguagem Python (link:documentacaoMSE):

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train)

# Calculando Erro Quadratico Medio (MSE) no sklearn
test_pred = lin_reg.predict(X_test)
mse = mean_squared_error(y_test, test_pred)

# Calculando MSE na unha
mse_sum = 0
for X, y in zip(X_test, y_test):
    prediction = lin_reg.pred(X)
    mse_sum += (y - prediction)**2
mse = mse_sum / len(y_test)
```

Corolário 2.5.1 — Outliers: MAE ou MSE?. Para responder essa pergunta, deve-se fazer outra: qual a importância dos outliers dentro do contexto do seu modelo? Sabemos que essa resposta varia em cada caso e, portanto, a escolha da métrica também irá depender do objetivo a ser alcançado. Um exemplo clássico diz respeito ao salário dos profissionais formados em Geografia. Michael Jordan é formado em geografia e seu salário, na época em que estava na ativa, era astronômico, levando assim a média salarial dos profissionais formados em geografia para as alturas. Isso deve ser levado em consideração?

2.5.3 RMSE - Root Mean Squared Error

A raíz do erro quadrático médio (RMSE) pode ser obtida simplesmente aplicando a raíz quadrada ao MSE, fazendo assim com que a unidade de medida do erro seja igual a unidade de

médida da classe target e, portanto, facilitando a interpretação do erro.

Estatisticamente, é possível dizer que o MSE é a variância e o RMSE o desvio padrão.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} (Predicted_i - Actual_i)^2}{N}}$$

Figura 2.9: Equação - Root Mean Squared Error

Calculando RMSE utilizando a linguagem Python:

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score
import numpy as np

lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train)

# Calculando Erro Quadratico Medio (MSE) no sklearn
test_pred = lin_reg.predict(X_test)
mse = mean_squared_error(y_test, test_pred)
rmse = np.sqrt(mse)

# Utilizando cross validation
scores = cross_val_score(lin_reg, X_train, y_train,
                         scoring='neg_mean_squared_error', cv=5)
cv_rmse = np.sqrt(-scores)
```

2.5.4 MAPE - Mean Absolute Percentage Error

O percentual do erro médio absoluto (MAPE) é o percentual equivalente do MAE. A equação é estritamente semellhante, porém um ajuste é realizado para converter tudo em porcentagem.

$$MAPE = \frac{100\%}{n} \sum \left| \frac{\overbrace{y - \hat{y}}^{\text{The residual}}}{y} \right|$$

Multiplying by 100% converts to percentage

Each residual is scaled against the actual value

Figura 2.10: Equação - Mean Absolute Percentage Error

Assim como o MAE, o MAPE também é robusto aos *outliers*, visto que também se trata de uma medida média da magnitude dos erros produzidos pelo modelo (valor absoluto).

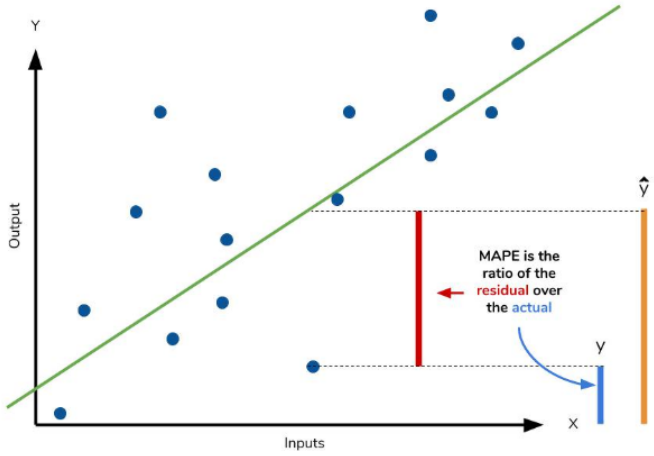


Figura 2.11: Interpretação Gráfica - Mean Absolute Percentage Error

Entretanto, existe uma limitação maior no uso do MAPE do que o MAE. Isso se dá pela utilização da divisão por y na fórmula, ou seja, a normalização pela classe target, fazendo assim com que o MAPE não seja definido para valores onde a classe target é 0. Outro ponto é que o MAPE pode gerar valores extremamente altos quando a classe target for extremamente pequena (divisão por um valor mínimo).

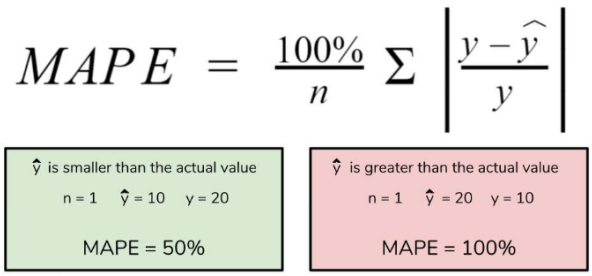


Figura 2.12: Aplicação - Mean Absolute Percentage Error

Calculando MAPE utilizando a linguagem Python:

```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train)

# Calculando MAPE na unha
mape_sum = 0
for X, y in zip(X_test, y_test):
    prediction = lin_reg.pred(X)
    mape_sum += (abs((y - prediction))/y)
mape = mape_sum / len(y_test)
```

2.5.5 MPE - Mean Percentage Error

O erro percentual médio (MPE) é muito similar ao MAPE, com exceção da parcela absoluta do cálculo. Como a parcela negativa é levada em consideração, os erros irão se cancelar e, portanto, não poderá ser realizada nenhuma análise sobre a performance do modelo treinado. Entretanto, o MPE irá identificar se há um maior número de erros negativos do que positivos e vice e versa, provendo a capacidade de concluir se o modelo está *subestimando* (mais erros negativos) ou *superestimando* (maios erros positivos).

$$MPE = \frac{100\%}{n} \Sigma \left(\frac{y - \widehat{y}}{y} \right)$$

Figura 2.13: Equação - Mean Percentage Error

A análise gráfica do MPE pode ser visualizada abaixo:

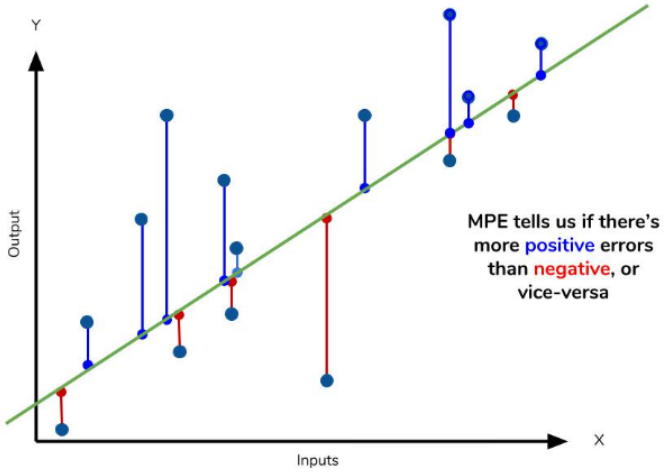


Figura 2.14: Interpretação Gráfica - Mean Percentage Error

Calculando MPE utilizando a linguagem Python.

```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train)

# Calculando MPE na unha
mpe_sum = 0
for X, y in zip(X_test, y_test):
    prediction = lin_reg.pred(X)
    mpe_sum += ((y - prediction)/y)
mpe = mpe_sum / len(y_test)
```

Corolário 2.5.2 — Medidas Relativas x Medidas Absolutas. Até o momento, vimos métricas com resultados absolutos (MAE, MSE e RMSE) e métricas como resultados relativos (MAPE e MPE). Nesse segundo campo, é preferível utilizar o MAPE do que o MPE por conta da fácil interpretação. O MPE pode ser útil para identificar erros sistemáticos do modelo.

2.5.6 R Squared Error

O erro R Squared (R^2) é utilizado para um melhor entendimento sobre quão bem as variáveis independentes explicam a variância no modelo. Para entender, de fato, o cálculo dessa métrica, é preciso realizar algumas definições prévias:

Definição 2.5.1 — RSE - Relative Squared Error.

$$\mathbf{RSE} = \frac{\sum_{j=1}^{n} \left(y_{pred} - y_{true}\right)^2}{\sum_{j=1}^{n} \left(y_j - y_{mean}\right)^2} \tag{2.14}$$

Dessa forma, o erro R Squared é dado por:

Definição 2.5.2 — R Squared Error.

$$\mathbf{R^2} = 1 - RSE = 1 - \frac{\sum_{j=1}^{n} \left(y_{pred} - y_{true}\right)^2}{\sum_{j=1}^{n} \left(y_j - y_{mean}\right)^2} \tag{2.15}$$

O numerador da equação que define o erro R Squared é dado pelo MSE (Mean Squared Error, ou erro quadrático médio). O denominador é dado pela variância da classe target Y. Quanto maior o MSE, menor o R Squared e pior o modelo. Analogamente, quanto menor o MSE, maior o R Squared e melhor o modelo.

2.5.7 Adjusted R Squared Error

Assim como o erro R^2, o Adjusted R Squared também mostra o quão bem os dados se enquadram na curva de predição, porém ajustado pela quantidade de observações do modelo.

Uma das características do R^2 é que este tende aumentar a medida que adiciona-se mais variáveis ao modelo. Entretanto, isto não pode ser levado como uma verdade absoluta. A métrica Adjusted R Squared conserta esse problema utilizando a fórmula:

$$R^2_{adj} = 1 - \left[\frac{\left(1 - R^2\right)(n-1)}{n-k-1}\right] \tag{2.16}$$

Onde n representa o número total de observações do modelo e k representa o número total de variáveis. Como exemplo prático, é possível levar em consideração a tabela abaixo e os resultados obtidos com as métricas R Squared e Adjusted R Squared.

Case 1		Case 2			Case 3		
Var1	Y	Var1	Var2	Y	Var1	Var2	Y
x1	y1	x1	2*x1	y1	x1	2*x1+0.1	y1
x2	y1	x2	2*x2	y1	x2	2*x2	y1
x3	y3	x3	2*x3	y3	x3	2*x3+0.1	y3
x4	y4	x4	2*x4	y4	x4	2*x4	y4
x5	y5	x5	2*x5	y5	x5	2*x5+0.1	y5

Tabela 2.2: Comparação R Squared e Adjusted R Squared

No Caso 1, tem-se 5 observações normais. No Caso 2, foi adicionada mais uma variável com correlação perfeita em relação a variável 1 (apenas foi multiplicada por 2). No Caso 3, foi aplicado um pequeno distúrbio na variável 2.

Ao aplicar o erro R Squared, espera-se que as métricas nos casos 2 e 3 não sejam menor que a métrica obtida no caso 1. Isto pois, como exemplificado acima, a adição de variáveis tende a aumentar (ou a manter) a métrica. No segundo caso, nenhuma informação adicional é fornecida ao modelo, visto que apenas foi adicionada uma variável com uma correlação perfeita com a variação já existente. A métrica não deveria aumentar para estes dois casos.

Entretanto, o erro Adjusted R Squared atua de forma mais assertiva e penaliza o segundo caso por conta do número de variáveis. A tabela abaixo mostra os resultados obtidos a partir de número gerados para as variáveis:

	Case 1	Case 2	Case 3
R Squared	0.985	0.985	0.987
Adj R Squared	0.981	0.971	0.975

Tabela 2.3: Resultados R Squared e Adjusted R Squared

Calculando R Squared utilizando a linguagem Python:

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score

lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train)

# Calculando R Squared Score
test_pred = lin_reg.predict(X_test)
r2 = r2_score(y_test, test_pred, multioutput='variance_weighted')
```

2.5.8 Resumo Sobre Métricas

As métricas MAE, MSE, RMSE, MAPE e MPE lidam com os resíduos produzidos pelo modelo. Para cada uma delas, utiliza-se a magnitude para decidir se o modelo está performance bem ou não. É extremamente importante conhecer o problema de negócio e a natureza dos dados para selecionar a melhor métrica de avaliação, dependendo da significância dada ao erro total.

As métricas mencionadas no parágrafo acima contemplam a **média** dos resíduos, entretanto também é possível realizar os cálculos a paratir da **mediana** dos resíduos. A figura abaixo traz um resumo rápido sobre as abordagens citadas:

acroynm	full name	residual operation?	robust to outliers?
MAE	Mean Absolute Error	Absolute Value	Yes
MSE	Mean Squared Error	Square	No
RMSE	Root Mean Squared Error	Square	No
MAPE	Mean Absolute Percentage Error	Absolute Value	Yes
MPE	Mean Percentage Error	N/A	Yes

Figura 2.15: Resumo de Métricas Absolutas e Relativas

Abaixo, um resumo das métricas mais utilizadas tanto em problemas de Regressão, quanto em Classificação e Aprendizado Não-Supervisionado:

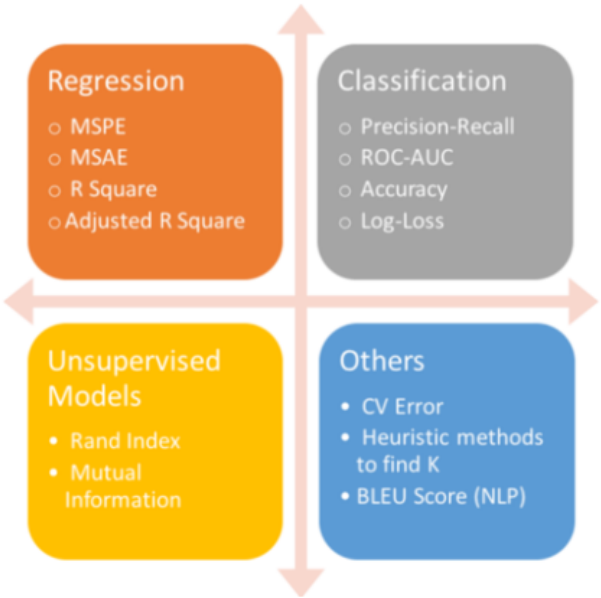


Figura 2.16: Métricas Mais Utilizadas

2.6 Links Úteis

- link-Understanding-Regression-Error-Metrics-in-Python
- link-Choosing-the-Right-Metric-for-Evaluating-Models
- link-Understand-Regression-Performance-Metrics
- link-Sklearn-Evaluation-Metrics
- link-Cap02-HandsOn-ML-LinearRegression
- link-Kaggle-Housing-Prices-Competition
- link-TowardsDS-Polynomial-Regression
- link-TowardsDS-Linear-and-Polynomial-Regression
- link-Repositorio-AndrewNG

2.7 Exercícios

Os exercícios a seguir foram retirados do livro *Hands-On Machine Learning with Scikit-Learn & TensorFlow*. A resposta para alguns deles irá depender de pesquisas em outras fontes, dado que este documento engloba um conteúdo resumido e compactado sobre estudos pessoais.

Exercício 2.1 Que função de minimização para Regressão Linear pode ser utilizada para um conjunto de treinos com milhões de features? ■

Com milhões de features, uma restrição que se faz presente é a utilização da Normal Equation, uma vez que esta envolve a inversão de matrizes de dimensões dependentes do número de features, se tornando assim um processo computacionalmente inviável (muito complexo). Dessa forma, o Stochastic Gradient Descent, Mini Batch Gradient Descent ou, quem sabe, o Batch Gradient Descent são tentativas válidas.

Exercício 2.2 Suponha que as features de um conjunto de dados tenha escalas totalmente diferentes. Quais algorítmos poderiam sofrer com essa diferença? O que poderia ser feito em relação a isso? ■

A grande maioria dos algoritmos poderá ser afetado com a diferença de escala das features. tal diferença afeta o gradiente no sentido de dificultar o alcance do mínimo global (imagine um dos eixos como uma elipse achatada), necessitando de um número maior de iterações, dado que os parâmetros de maior range (máximo e mínimo) são mais sensíveis às alterações em suas magnitudes.

Exercício 2.3 O Gradiente Descendente pode ficar preso em um mínimo local quando treinamos um modelo de Regressão Linear? ■

Não. A função custo em modelos de Regressão Linear possuem um shape convexo *(bowl shaped)* e, portanto, possuem um único mínimo global.

Exercício 2.4 Todos os algoritmos de Gradiente levam ao mesmo modelo se os deixarmos rodando por um tempo suficiente? ■

Não. Existe um parâmetro que influencia diretamente no comportamento do gradiente rumo ao mínimo global: learning rate (ou α). O parâmetro de regularizazção α tem a responsabilidade de fornecer ao modelo a magnitude da regularização aplicada ao treinamento. Em outras palavras, seu valor pode fazer com que o gradiente alcance o mínimo global de forma mais lenta, mais rápida, ou até mesmo fazer com que a função não alcance o mínimo global, divergindo do resultado esperado.

Mesmo que o fator de regularização não esteja tão elevado e, se tratando de Regressões Lineares e Logísticas (funções custo convexas), ainda sim não é possível afirmar que todos os algoritmos cheguem ao mesmo ponto em um longo período de tempo. Isto pois os algoritmos *Stochastic GD* e *Mini-Batch GD*, por suas funcionalidades, não seguem um caminho linear rumo ao mínimo, mas sim apresentam uma pequena oscilação próximo a região ótima.

Exercício 2.5 Imagine que você tenha utilizado o Batch Gradient Descend e tenha plotado a curva de erro nos dados de validação a cada época. Se você notar que o erro de validação consistentemente aumenta, o que pode estar acontecendo? Como consertar isso? ■

Nessa situação, é possível concluir que o modelo sofre de overfitting, ou seja, o modelo se enquadra tão bem nos dados de treino que, ao ser apresentado a dados que não fizeram parte do treinamento, resulta em consideráveis erros. Isto indica uma falta de generalização do modelo e pode estar vinculado aos seguintes fatores:

- Alto grau polinomial
- Muitas features presentes
- Baixo fator de regularização alpha

Para amenizar esse efeito, seria possível:

- Diminuir o grau polinomial
- Aumentar o fator de regularização alpha
- Aplicar feature selection para coletar somente as features relevantes

Exercício 2.6 É uma boa ideia parar o Mini-Batch Gradient Descent imediatamente quando o erro de validação aumenta? ■

Não. A parada imediata deste gradiente assim que o erro de validação cresce é um equívoco e pode ocasionar perfromances não esperadas. Neste caso, o ideal seria continuar o treinamento por mais algumas iterações para realmente se certificar de que a performance não pode ser melhorada. Neste momento, então, o treinamento pode ser parado. A própria natureza randomica dos algoritmos Mini-Batch GD e Stochastic GD causam certas oscilações no debug.

Tecnicamente, é possível salvar os melhores resultados gradativamente para que, quando o modelo não superar o melhor resultado depois de um certo período de tempo, seja possível retornar ao momento onde o modelo alcançou o melhor resultado e considerar os parâmetros obtidos.

Exercício 2.7 Qual algoritmo de Gradiente (entre os discutidos no capítulo) irá alcançar a vizinhança da solução ótima de maneira mais rápida? Quais convergem? Como é possível fazer com que os demais também alcancem os mesmos resultados? ■

O algoritmo que irá alcançar a vizinhança da solução ótima mais rápida é o Stochastic Gradient Descent, dado que considera apenas uma única amostra para o ajuste dos parâmetros a cada iteração. O algoritmo que, de fato, irá convergir para o mínimo exato é o Batch Gradient Descent, dado que este considera todo o conjunto de dados para o ajuste dos parâmetros a cada iteração. Por este mesmo motivo, o Batch GD é o algoritmo que leva o maior tempo para alcance do mínimo global.

Para fazer com que todos os algoritmos alcancem o mínimo global (ou pelo menos cheguem o mais próximo disso possível), pode-se diminuir gradativamente o parâmetro de regularização α. Com isto, mesmo os modelos vinculados a um viés aleatório (Stochastic GD e Mini-Batch GD) irão apresentar oscilações muito próximas a zero.

Exercício 2.8 (Será visto nos próximos capítulos). Suponha que você tenha terinado um modelo de Regressão Polinomial. Após a plotagem das curvas de aprendizado, você verifico que há um grande gap entre o erro calculado com os dados de treino e o erro de validação. O que pode estar acontecendo? Como resolver isso? ■

Este grande espaço entre as curvas de erro de treino e validação indicam um *overfitting* nos dados. Como mencionado em um exercício anterior, este tipo de fenômeno ocorre quando o modelo se enquadra tão bem nos dados de treino que perde poder de generalização, apresentando resultados ruins quando submetido a dados externos ao treinamento (validação e treino). Para mitigar o overfitting, é possível:

- Diminuir o grau polinomial do modelo
- Coletar mais dados para o treinamento
- Diminuir o set de features do modelo
- Aumentar o parâmetro de regularizazção alpha

Exercício 2.9 Suponha que você esteja usando o algoritmo Ridge Regression e notouo que o erro nos dados de treino e o erro nos dados de validação são praticamente idênticos porém em magnitude elevada. O que você poderia dizer? O modelo sofre de um bias alto ou de uma variância alta? Você deve aumenar ou diminuir o parâmetro de regularização alpha? ■

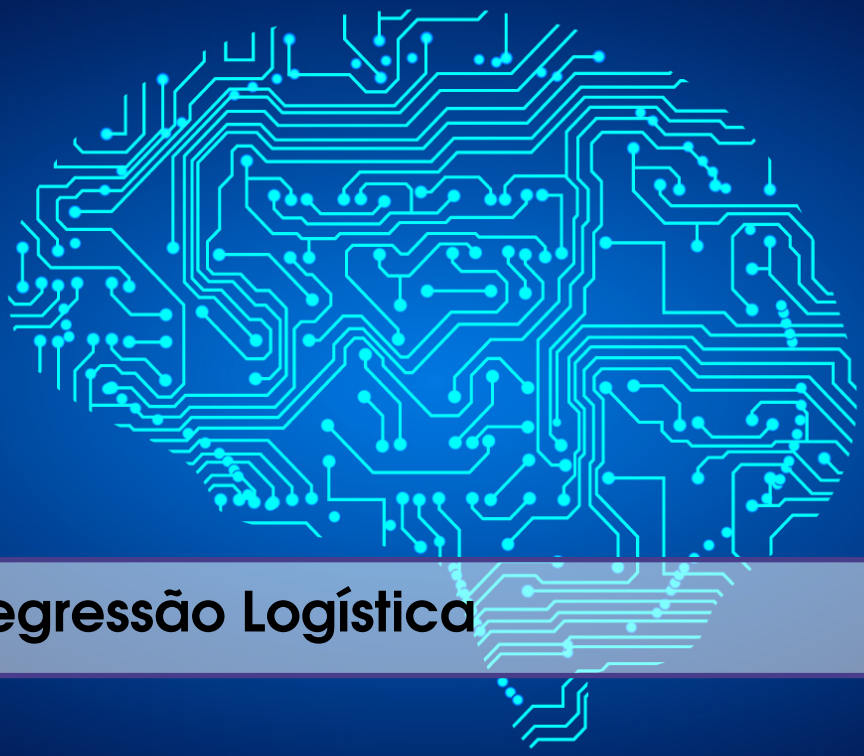
Este é um típico caso de *underfitting*, ou seja, o modelo sofre de um bias alto. Em modelos com bias alto, aumentar o número de dados de treinamento não irá funcionar. Para aumentar a performance, é possível diminuir o parâmetro de regularização alpha para que o modelo não penalize muito a magnitude dos parâmetros θ vinculados a graus polinomiais elevados.

Exercício 2.10 Qual a melhor opção entre?

- Ridge Regression *versus* Linear Regression (sem regularização)
- Lasso *versus* Ridge Regression
- Elatic Net *versus* Lasso

■

- É preferível usar Ridge Regression ao invés da clássica Regressão Linear, dado que fatalmente os dados não irão se comportar exatamente como uma curva linear, necessitando assim de alguma regularização/penalização no modelo.
- Lasso é preferível ao Ridge sempre que você suspeitar que, no modelo, existam algumas features irrelevantes para o resultado final. Isto pois Lasso traz a penalidade na norma l_1, praticamente eliminando a magnitude das features irrelevante e atuando como um verdadeiro feature selection.
- Neste caso, é preferível utilizar o algoritmo Elastic Net quando o número de features do modelo superar (ou chegar bem próximo) o número de dados de treinamento. Isto pois o algoritmo Lasso tende a se comportar de maneira inadequada quando o número de features ultrapassa os dados de treinamento. Também é preferível Elastic Net quando existem muitas features correlacionadas além da possibilidade de configurar o parâmetro $l_1 ratio$.



3. Regressão Logística

Alguns algoritmos de regressão também podem ser utilizados para classificação. A Regressão Logística é comumente usada para estimar a probabilidade que uma instância tem de pertencer a uma classe particular.

3.1 Representação do Modelo

Definição 3.1.1 — Estimando Probabilidades.

$$\mathbf{p} = h_\theta(x) = \sigma\left(\theta^T.x\right) \tag{3.1}$$

Definição 3.1.2 — Função Sigmoidal.

$$\sigma(\mathbf{t}) = \frac{1}{1+\exp(-t)} \tag{3.2}$$

A função sigmoidal é utilizada em problemas de classificação. O nome "sigmóide"vem justamente do formato em "S"do seu gráfico:

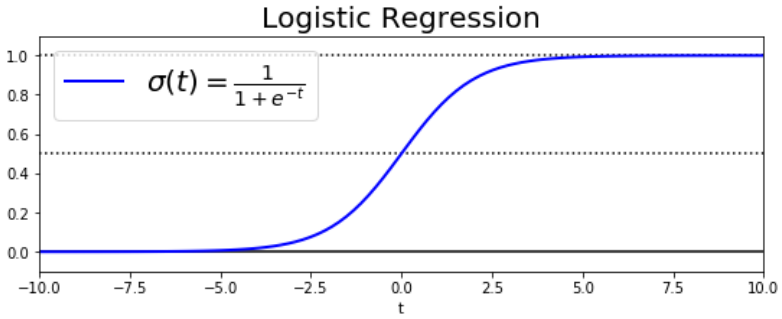


Figura 3.1: Função Sigmoidal

Nesse caso de exemplo, as predições poderiam ser dadas por:

$$y = \begin{cases} 0 \text{ if } p < 0.5 \\ 1 \text{ if } p >= 0.5 \end{cases} \tag{3.3}$$

A Função Custo em um problema de Regressão Logística pode ser definida por:

$$c(\theta) = \begin{cases} -\log(p) \text{ if } y = 1 \\ -\log(1-p) \text{ if } y = 0 \end{cases} \tag{3.4}$$

Definição 3.1.3 — Função Custo.

$$\mathbf{J}(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(p^{(i)}\right) + \left(1-y^{(i)}\right)\log\left(1-p^{(i)}\right)\right] \tag{3.5}$$

É interessante citar alguns exemplos mostrando o custo relacionado a cada predição $p^{(i)}$, dado o respectivo target $y^{(i)}$.

$$y^{(i)} = 1 \longrightarrow \mathbf{J}(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(p^{(i)}\right)\right] \tag{3.6}$$

$$y^{(i)} = 0 \longrightarrow \mathbf{J}(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[\left(1-y^{(i)}\right)\log\left(1-p^{(i)}\right)\right] \tag{3.7}$$

Portanto, caso $y^{(i)} = 1$ e $p^{(i)} = 1$, ou seja, uma classificação perfeitamente correta, a função custo se torna $log(1)$ e equivale a zero (custo mínimo). Caso $p^{(i)} = 0$ para o mesmo target, ou seja, uma classificação errada, a função custo se torna $log(0)$ e tende ao infinito (custo máximo).

A análise acima é análoga para o caso $y^{(i)} = 0$ e $p^{(i)} = 0$, retornando um custo mínimo com $log(1)$ e, por outro lado, um custo máximo de $log(1)$ para $p^{(i)} = 1$.

Definição 3.1.4 — Derivadas Parciais.

$$\frac{\partial}{\partial\theta_j}J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(\sigma\left(\theta^T.x^{(i)}\right) - y^{(i)}\right)x_j^{(i)} \tag{3.8}$$

A filosofia para o cálculo da função custo e também do gradiente descendente a partir das derivadas parciais é semelhante a utilizada na Regressão Linear. A única alteração é na função *hypothesis* $h(\theta)$, sendo esta agora baseada na função sigmoidal.

3.2 Métricas de Avaliação

Assim como a sessão 2.5 do capítulo anterior, onde foram detalhadas as principais métricas de avaliação para modelos de Regressão, nesta sessão a proposta é apresentar uma mesma análise, levando agora em consideração modelos de Classificação.

Para facilitar o entendimento, é importante definir alguns termos:

Descrição	Abrev	Definição
True Positives	TP	$y_{true} = 1; y_{pred} = 1$
False Positives	FP	$y_{true} = 0; y_{pred} = 1$
True Negatives	TN	$y_{true} = 0; y_{pred} = 0$
False Negatives	FN	$y_{true} = 1; y_{pred} = 0$

Tabela 3.1: Definições em Classificação

1. **True Positives (TP):** Mostra que o elemento da classe Positiva foi *corretamente* predito pelo modelo como sendo da classe Positiva. (exemplo: transação fraudulenta marcada como fraudulenta)
2. **False Positives (FP):** Mostra que o elemento da classe Negativa foi *incorretamente* predito pelo modelo como sendo da classe Positiva. (exemplo: transação normal marcada como fraudulenta - *Erro Tipo I*)
3. **False Negatives (FN):** Mostra que o elemento da classe Positiva foi *incorretamente* predito pelo modelo como sendo da classe Negativa. (exemplo: transação fraudulenta marcada como normal - *Erro Tipo II*)
4. **True Negatives (TN):** Mostra que o elemento da classe Negativa foi *corretamente* predito pelo modelo como sendo da classe Negativa. (exemplo: transação normal marcada como normal)

3.2.1 Matriz de Confusão

A Matriz de Confusão é usada para descrever a performance de modelos de classificação em um conjunto onde o target é conhecido. A Matriz, por si só, não pode ser considerada uma métrica de avaliação, entretando grande parte das demais métricas são baseadas na própria matriz e nos elementos que a compõe.

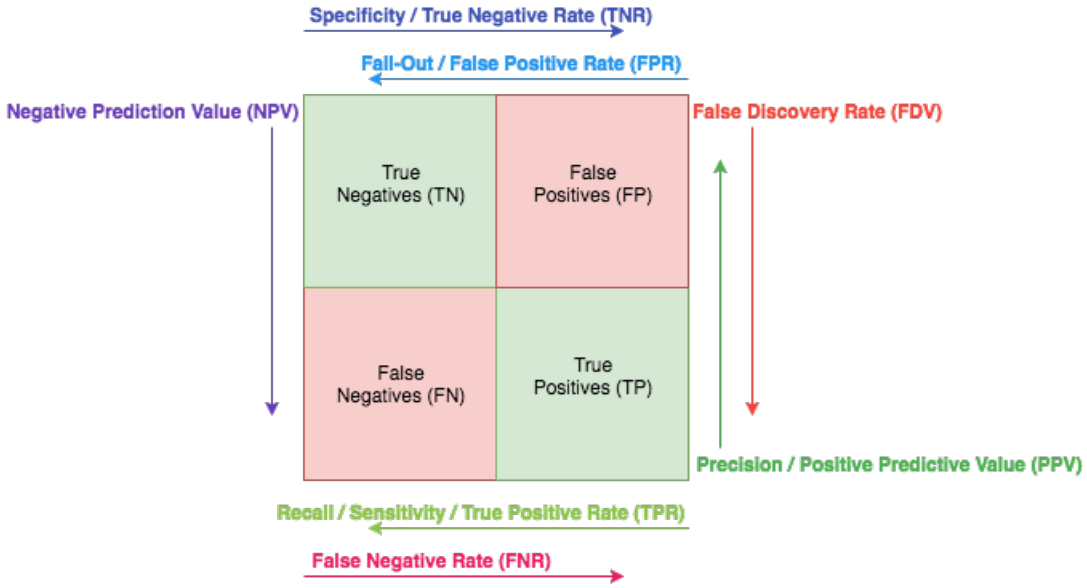


Figura 3.2: Matriz de Confusão

Para computar a Matriz de Confusão, é necessário possuir um set de *predictions* para que estes sejam comparados aos valores *target*. Para tal, seria possível realizar predições no set de teste, entretanto, a ideia é deixa-lo intocável até as definições finais do modelo. Dessa forma, os valores podem ser obtidos através de predições nos próprios dados de treinamento.

Para reunir predições utilizando os dados de treino aplicando validação cruzada, pode-se utilizar o método *cross_val_predict()*. O código abaixo é um exemplo de utilização e plotagem de uma Matriz de Confusão em um problema de classificação de Risco de Crédito:

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import confusion_matrix
import itertools
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# Treinando um modelo de classificacao
log_reg = LogisticRegression()
log_reg.fit(X_train, y_train)

# Reunindo predicoes
predictions = cross_val_predict(log_reg, X_train, y_train, cv=5)

# Definindo funcao para personalizar CfMx
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function sets up and plot a Confusion Matrix
    """
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title, fontsize=14)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
```

```
    plt.yticks(tick_marks, classes)

    # Format plot
    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 1.2
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j]),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.ylabel('True label')
    plt.xlabel('Predicted label')

# Plotando matriz de confusao
labels = ['Good', 'Bad']
cf_mx = confusion_matrix(y_train, predictions,
                         title='LogisticRegression Confusion Matrix')
plot_confusion_matrix(cf_mx, labels)
plt.show()
```

Figura 3.3: Exemplo de Matriz de Confusão

3.2.2 Accuracy

A acurácia de um modelo pode ser entendida como a quantidade de predições corretas (de ambas as classes) em relação ao total de elementos classificados. Em termos técnicos, podemos definir a acurácia como:

Definição 3.2.1 — Accuracy.

$$\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN} \tag{3.9}$$

Nem sempre um modelo com uma alta acurácia é um bom modelo. É interessante utilizar essa métrica quando o conjunto de dados possui classes balanceadas. Exemplos clássicos onde medir a performance do modelo através da acurácia não é recomendada:

- Modelo para identificação de transações fraudulentas
- Modelo para diagnóstico de câncer

- Qualquer modelo com a classe target *desbalanceada*

Isto pois, se o problema de negócio envolver dados com uma quantidade muito baixa da classe positiva, qualquer modelo que classifique toda e qualquer instância como sendo da classe negativa, terá uma acurácia muito alta.

Por exemplo, se o problema a ser resolvido envolve a identificação de transações fraudulentas em cartões e, no conjunto de dados, apenas 5% das instâncias são fraudulentas, o modelo que classificar toda e qualquer transação como não-fraudulenta, terá 95% de acurácia. Parece um bom número, mas nenhuma transação fraudulenta foi identificada. Existem métricas que lidam com esse tipo de problema e serão vistas logo a seguir.

Antes, vejamos como medir a acurácia de modelos utilizando validação cruzada:

```
from sklearn.model_selection import cross_val_score

model.fit(X_train, y_train)
acc_scores = cross_val_score(model, X_train, y_train, cv=5,
                             scoring='accuracy')
acc_score = acc_scores.mean()
```

3.2.3 Precision

A métrica Precision responde a seguinte pergunta: "De todas as predições da classe positiva, quantas realmente são da classe positiva?". Intuitivamente, significa dizer o quão preciso e acertivo é o modelo treinado, visto que a métrica mede os verdadeiros acertos da classe positiva em todas as classificações positivas do modelo. Talvez o entendimento será maior a partir da fórmula:

Definição 3.2.2 — Precision.

$$\text{Precision} = \frac{TP}{TP+FP} \tag{3.10}$$

Retomando o exemplo do problema de identificação de fraudes, significa dizer que a métrica mede a relação entre transações fraudulentas classificadas corretamente (TP) e todas as transações fraudulentas classificadas (TP+FP).

Aumentar Precision significa diminuir o número de *Falsos Positivos (FP)* (ex: transações normais erroneamente classificadas como fraudes). Um exemplo onde os Falsos Positivos podem ter mais importância que os Falsos Negativos é na classificação de Spams. Aumentar a precisão do modelo, significa minimizar e-mails normais classificados como Spam, o que pode ser mais vantajoso do que deixar alguns Spams passarem batido.

Porém, pode-se sempre querer um modelo preciso? No exemplo de identificação de fraude, seria mais vantajoso minimizar os Falsos Positivos (transações normais classificadas como fraude) ou os Falsos Negativos (fraudes classificadas como transações normais)? É melhor bloquear o cartão de clientes fiéis e pegar todos os fraudadores? Ou não bloquear o cartão de clientes fiéis a troco de deixar alguns fraudadores? Antes de conhecer a métrica que pode minimizar os Falsos Negatives, vejamos um exemplo de como medir a precisão de um modelo:

```
from sklearn.model_selection import cross_val_score

model.fit(X_train, y_train)
prec_scores = cross_val_score(model, X_train, y_train, cv=5,
                              scoring='precision')
precision = prec_scores.mean()
```

3.2.4 Recall

Como introduzido no tópico anterior, Recall (ou *Sensitivy*, ou *True Positives Rate*) responde a pergunta: "De todas as instâncias pertencentes a classe positiva, quantas o modelo classificou corretamente?". Comparando com Precision, aqui tem-se uma significância maior na Classe Atual das instâncias e não na Classe Predita.

Definição 3.2.3 — Recall.

$$\text{Recall} = \frac{TP}{TP+FN} \tag{3.11}$$

Retomando o exemplo do problema de identificação de fraudes, temos o Recall como medida da relação entre transações fraudulentas classificadas corretamente (TP) e todas as transações que deveriam ser classificadas como fraudulentas (TP+FN).

Aumentar o Recall significa diminuir o número de *Falsos Negativos (FN)* (ex: fraudes erroneamente classificadas como transações normais).

A mesma questão feita anteriormente é: o problema de negócio irá definir se é mais vantajoso minimizar os Falsos Positivos (aumentar Precision) ou os Falsos Negativos (aumentar Recall). O fato é que nunca será possível maximizar os dois simnultaneamente. Existe uma troca entre Precision e Recall onde aumentar um significa diminuir o outro. Antes de entrar em detalhes dessa relação, vejamos como calcular o recall de um modelo utilizando validação cruzada.

```
from sklearn.model_selection import cross_val_score

model.fit(X_train, y_train)
rec_scores = cross_val_score(model, X_train, y_train, cv=5,
                             scoring='recall')
recall = rec_scores.mean()
```

3.2.5 Specificity

Specificity é exatamente o oposto do Recall (ou Sensitivity). Também chamada de taxa de Falsos Positivos (FPR), a Specificity corresponde a proporção de amostras negativas erroneamente classificadas como positivas, dado todas as amostras da classe negativa.

Definição 3.2.4 — Specificity.

$$\text{Precision} = \frac{FP}{FP+TN} \tag{3.12}$$

De fato, a Taxa de Falsos Positivos dá maior importância a instâncias negativas classificadas de forma errada como positivas. Mais a frente, será possível analisar o Recall (Taxa de Verdadeiros Positivos) e a Specificity (Taxa de Falsos Positivos) em conjunto dentro da curva ROC.

3.2.6 F1-Score

Precision e Recall são duas métricas extremamente importantes para qualquer modelo de classificação. A relação entre ambas deve ser explorada ao limite (e isso será detalhado em um dos próximos tópicos). Entretanto, nem sempre será preciso carregar ambas as métricas e utiliza-las a todo momento. A F1-Score é uma métrica desenvolvida justamente para contemplar Precision e

Recall em um único número, através de uma *média harmônica*.

Definição 3.2.5 — F1-Score.

$$\text{f1} = 2 \times \frac{Precision * Recall}{Precision + Recall} \tag{3.13}$$

O score F1 pode ser utilizado como objetivo de minimização, visto que penaliza modelos onde a diferença entre Precision e Recall é muito ampla (um muito bom e o outro muito ruim), retornando bons índices caso os dois sejam próximos e em níveis razoáveis. Abaixo, uma forma de calculá-lo.

```
from sklearn.model_selection import f1_score

model.fit(X_train, y_train)
f1_scores = cross_val_score(model, X_train, y_train, cv=5,
                            scoring='f1')
f1 = f1_scores.mean()
```

3.2.7 Log-loss

A avaliação de um modelo de classificação com a métrica Log-loss é dada, em particular, quando a saída bruta do classificador é uma probabilidade numérica ao invés de um label da classe. Log-loss é uma métrica que leva em consideração essa probabilidade no momento da predição, e não apenas se a classe predita é positiva ou negativa.

Sua fórmula é muito semelhante a equação 3.5 definida para a Função Custo em 3.1.3.

Definição 3.2.6 — Log-loss.

$$\text{log-loss} = \sum_{i=1}^{N} y_i \log(p_i) + (1 + y_i) \log(1 - p_i) \tag{3.14}$$

Essa definição é interessante pois está diretamente relacionada com a teoria da informação: intuitivamente, a métrica log-loss mede a imprevisibilidade do "ruído extra"proveniente de um preditor em oposição a classe positiva. Minimizando a cross entropia garante uma grande acurácia ao classificador.

3.2.8 ROC Curve

A curva ROC é extremamente útil para visualizar o balanceamento entre a Taxa de Verdadeiros Positivos (TPR ou Recall) e a Taxa de Falsos Positivos (FPR ou Sensitivity). Basicamente, trata-se de um gráfico que mostra a performance de um classificador binário de acordo com a variação do **threshold** de predição.

Ao invés de predizer as classes diretamente em um problema de classificação, as vezes é mais interessante predizer as probabilidades de cada observação. Dessa forma, é possível ter uma maior flexibilidade na interpretação dessas probabilidades e, como consequência, será possível avaliar a performance do modelo de forma mais assertiva.

Para a plotagem da curva ROC, é preciso ter em mãos um vetor de probabilidades ao invés de um vetor de predições de classes puramente dito. Utilizando a linguagem Python, é possível retornar tais probabilidades através do método *cross_val_predict()* configurando o parâmetro *method="decision_function"* para classificadores, ou *method="predict_proba"* para classificadores do tipo árvore (Decision Trees e Random Forest). Vejamos um exemplo:

```
from sklearn.model_selection import cross_val_predict

try:
    y_scores = cross_val_predict(model, X_train, y_train, cv=10,
                                 method='decision_function')
except:
    # Arvores nao possuem o metodo decision_function
    y_probas = cross_val_predict(model, X_train, y_train, cv=10,
                                 method='predict_proba')
    y_scores = y_probas[:, 1]
```

Após isso, é possível retornar os elementos necessários para a construção da curva ROC através de funções builtins (roc_curve e roc_auc_score).

```
from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_test, y_scores)
fig, ax = plt.subplots(figsize=(13, 6))
ax.plot(fpr, tpr)
```

A função *roc_curve()* recebe o vetor label verdadeiro e um vetor de predições (gerado a partir do método *cross_val_predict()*). Como resultado, temos três elementos utilizados para a plotagem da curva ROC:

- **thresholds:** todas as probabilidades únicas de predições em ordem descrecente
- **fpr:** taxa de Falsos Positivos para cada threshold
- **tpr:** taxa de Verdadeiros Positivos para cada threshold

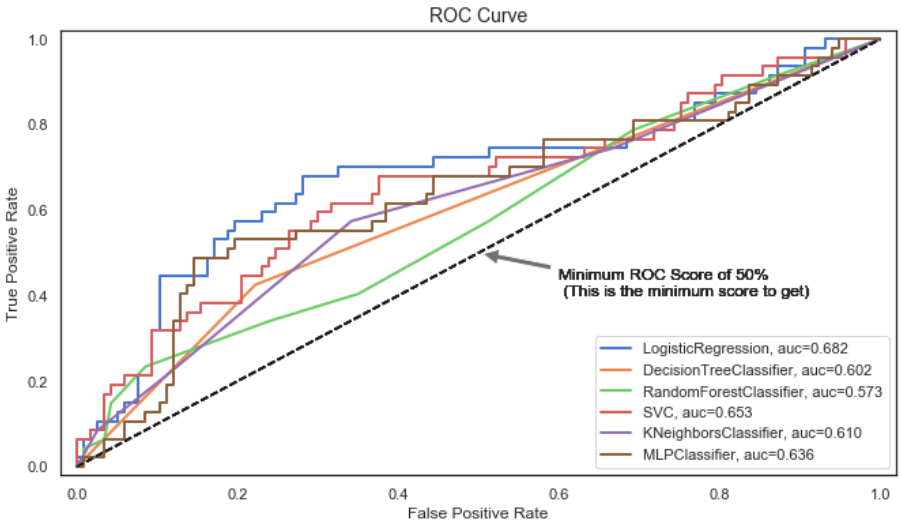


Figura 3.4: ROC Curve

A figura 3.4 mostra um exemplo de curva ROC para 6 diferentes classificadores. Espera-se de um bom classificador que sua curva ROC esteja tão distante da linha pontilhada quanto possível, uma vez que esta simplesmente representa um classificador de caráter aleatório.

Além disso, é possível afirmar, a partir da curva, que um alto threshold resulta em um ponto no canto inferior esquerdo da plotagem, enquanto um baixo threshold resulta em um ponto no canto superior direito da plotagem. Com isso, conclui-se que quando menor o threshold, menor o threshold, maior a taxa TPR, a custo de um aumento também na taxa FPR.

Corolário 3.2.1 — AUC Score: Area Under The Curve. Para medir a performance do modelo utilizando a curva ROC, existe uma métrica chamada AUC (Area Under The Curve) que, como o nome sugere, retorna a área abaixo da curva ROC (integral) entre os pontos [0, 0] e [1, 1]. Quanto mais próximo de 1.0 para a métrica AUC, melhor o modelo.

```
from sklearn.metrics import roc_auc_score

auc_score = roc_auc_score(y_train, y_scores)
```

3.3 Relação entre Precision e Recall

Até o momento, teve-se a impressão de que era possível minimizar os Falsos Positivos e os Falsos Negativos sem nenhuma troca equivalente (referência nerd: https://fma.fandom.com/pt-br/wiki/Troca_Equivalente), entretanto não é tão simples quanto parece.

No exemplo retirado do capítulo 3 do livro *Hands-On Machine Learning with Scikit-Learn & TensorFlow* de Aurélien Géron, o dataset de dígitos MNIST foi utilizado para montar um classificador binário do dígito 5. Em outras palavras, o modelo simplesmente recebia um dígito e tinha o papel de identificar se este era ou não um 5. Abaixo, uma matriz de confusão ilustrada:

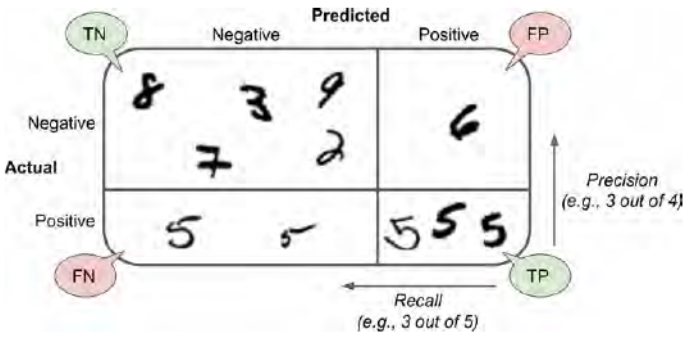


Figura 3.5: Classificador de Dígitos

Aqui, é possível perceber que algumas falhas aconteceram, como por exemplo, a classificação errada do dígito 6 como sendo o dígito 5, além de alguns dígitos 5 sendo classificados como "não-5". De fato isso foi mencionado, mas não definido: como maximizar Precision (diminuindo FP) ou como maximizar Recall (diminuindo FN)? Resposta: **thresholds**.

Corolário 3.3.1 — Threshold em Classificadores Probabilísticos. Se você está treinando seu modelo em um classificador probabilístico, ou seja, um classificador onde as predições são probabilidades entre 0 e 1, por padrão você pode dizer que qualquer probabilidade acima de 0.5 é parte de uma classe e, analogamente, probabilidades abaixo de 0.5 pertencem a outra classe. O limite 0.5 é, na verdade o threshold do modelo e este pode ser modificado para atender determinados requisitos.

A figura abaixo demonstra o efeito do ajuste do threshold no exemplo de classificador de digitos mencionado anteriormente. Comparando essa figura com a figura 3.5, é possível perceber que o ajuste do threshold permite:

- Não classificar nenhum 5 erroneamente (a custo de classificar outros dígitos de forma errada)
- Não classificador nenhum outro dígito erroneamente (a custo de classificar 5s de forma errada)

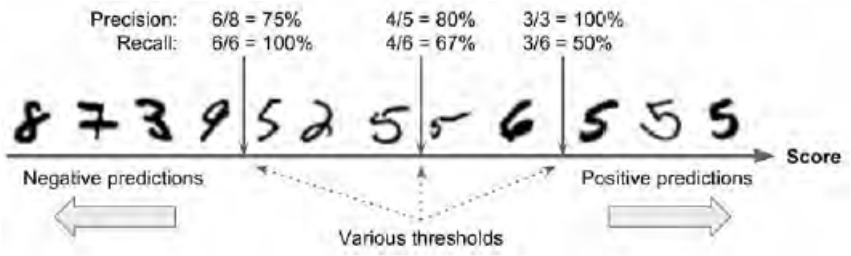


Figura 3.6: Thresholds de um Classificador Binário de Dígitos

Na definição da curva ROC em 3.2.8, foi criado o vetor *y_scores* contendo predições em forma de probabilidades (e não em forma de classes positivas e negativas). A partir desses dados, é possível plotar duas curvas extremamente importantes:

1. Precision/Recall *versus* Thresholds
   - Permite visualizar as métrica Precision e Recall de acordo com o Threshold configurado
2. Precision *versus* Recall
   - Permite validar a real troca entre Precision e Recall do modelo

```
from sklearn.metrics import precision_recall_curve

precisions, recalls, thresholds = precision_recall_curve(y_train, y_scores)
fig, ax = plt.subplots()
ax.plot(thresholds, precisions[:-1], 'b--', label='Precision')
ax.plot(thresholds, recalls[:, -1], 'g--', label='Recall')
```

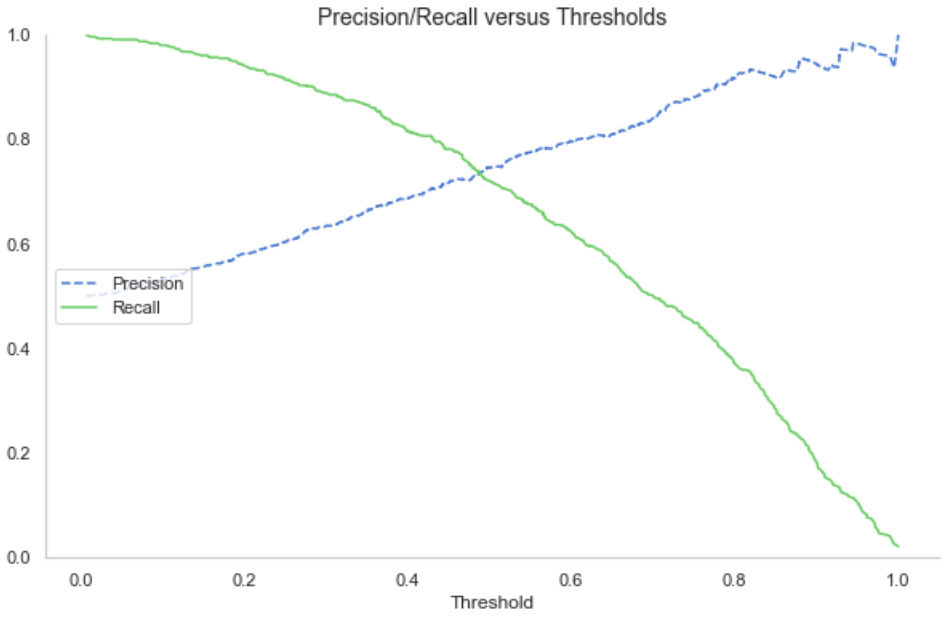


Figura 3.7: Efeitos do Threshold na Classificação

```
fig, ax = plt.subplots()
ax.plot(recalls, precisions)
ax.set_xlabel('Recall')
ax.set_ylabel('Precision')
plt.axis([0, 1, 0, 1])
```

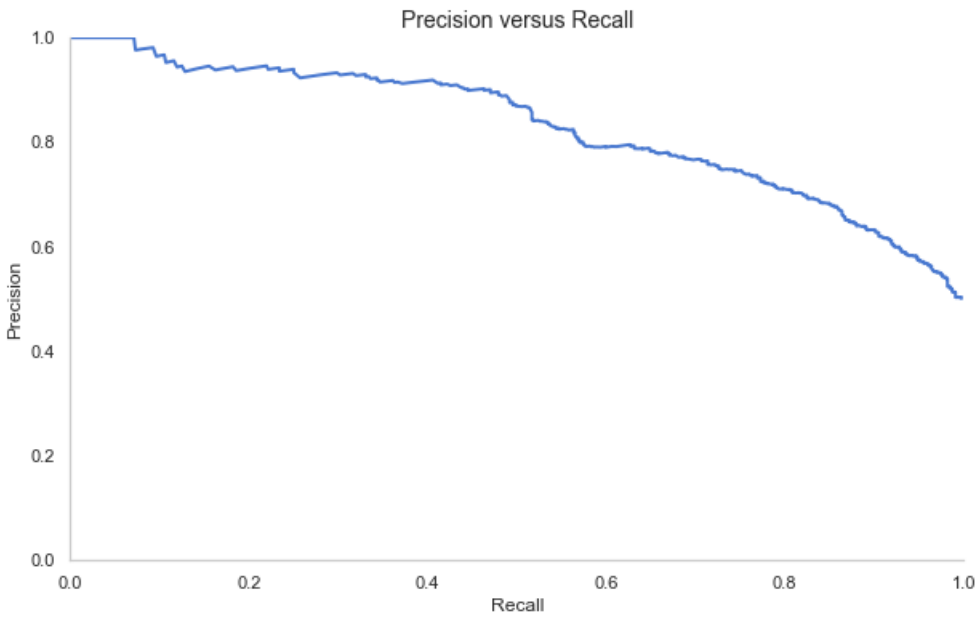


Figura 3.8: Precision/Recall Tradeoff

Corolário 3.3.2 — Precision/Recall Curve versus ROC Curve. Na sessão 3.2.8 vimos como a curva ROC pode ser um importante instrumento de avaliação de modelos de classificação. Nessa sessão, vimos a curva Precision/Recall e a relação entre ambas as métricas. Quando usar uma ou outra? Caso a classe positiva seja de rara aparição ou quando há um interesse maior em Falsos Positivos do que Falsos Negativos, prefere-se a curva Precision/Recall.

3.4 Thresholds

Até o momento, diversas métricas foram mencionadas, definidas e estudadas. Algumas delas, entretanto, envolviam limites de classificação responsáveis por definir predições das classes positivas e negativas. Tais limites são conhecidos como **thresholds**.

Ao treinar e avaliar um modelo, um threshold padrão é utilizado dentro das bibliotecas utilizadas. Porém, visando atingir o objetivo do negócio, é possível modificar esse limite de modo a modificar as predições das classes positivas e negativas.

Utilizando o NumPy, é possível criar um array de thresholds a partir de um limite inferior, superior e um passo. Após retornar o vetor de probabilidades *y_scores*, é possível realizar uma indexação booleana para retornar apenas os valores maiores que um determinado limite, sempre visando a maximização de alguma métrica (precision ou recall).

Na prática, o código abaixo retorna as predições em forma de probabilidades (já visto antriormente) e, na sequência, define um novo threshold para a criação de um novo vetor de predições baseado no limite ajustado:

```
from sklearn.model_selection import cross_val_predict

# Retornando predicoes em forma de probabilidades
try:
    y_scores = cross_val_predict(model, X_train, y_train, cv=10,
                                 method='decision_function')
except:
    # Arvores nao possuem o metodo decision_function
    y_probas = cross_val_predict(model, X_train, y_train, cv=10,
                                 method='predict_proba')
    y_scores = y_probas[:, 1]

# Definindo probabilidades com novo threshold
t = -3.16 # novo threshold
custom_scores = np.where(y_scores > t, 1, 0)
```

A partir do momento em que é definido um novo threshold **t**, pode-se gerar um novo vetor de scores customizados *custom_scores* utilizando o método *where(condition, if_true, if_false)* do NumPy onde o argumento *condition* é um resultado booleano que retorna True sempre que a condição for verdadeira e False caso contrário.

Com o novo vetor *custom_scores*, é possível realizar diversas análises. A figura abaixo, compara a utilização dessa abordagem na geração de uma matriz de confusão com um threshold pré estabelecido.

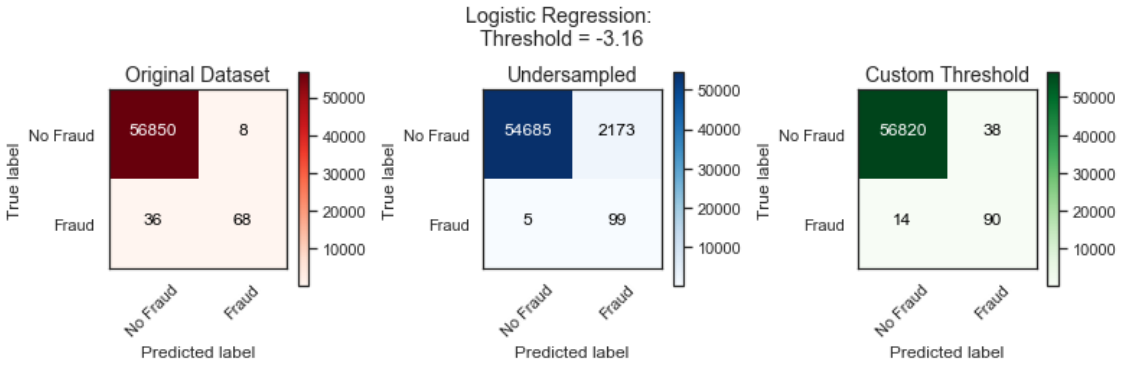


Figura 3.9: Matriz de Confusão com Ajuste de Threshold

No exemplo acima, o threshold foi modificado de forma a maximizar o Recall do modelo (diminuição dos Falsos Negativos). Observe que, com isso, há um aumento nos Falsos Positivos do modelo. Para visualizar de uma forma mais didática a relação entre essa troca e os thresholds, veja abaixo uma série de matrizes e os efeitos em sua composição de acordo com o threshold ajustado.

Seria possível então avaliar uma série de thresholds e seus respectivos efeitos no modelo de classificação? A resposta para essa pergunta é: sim! O código abaixo cria um array com alguns limites de classificação e avalia, para cada um deles, os resultados do modelo a partir da Matriz de Confusão.

```
cf_mx_thresholds = np.linspace(-5, 0, 9)
i = 1
plt.figure(figsize=(11, 10))
for t in cf_mx_thresholds:
    custom_scores = np.where(y_scores > t, 1, 0)
    cf_mx = confusion_matrix(y_test, custom_scores)
    plt.subplot(3, 3, i)
    plot_confusion_matrix(cf_mx, labels, title=f'Threshold={t}',
                          cmap=plt.cm.Greens)
    i += 1
plt.tight_layout()
plt.suptitle('Threshold Comparison')
plt.subplots_adjust(top=0.93)
plt.show()
```

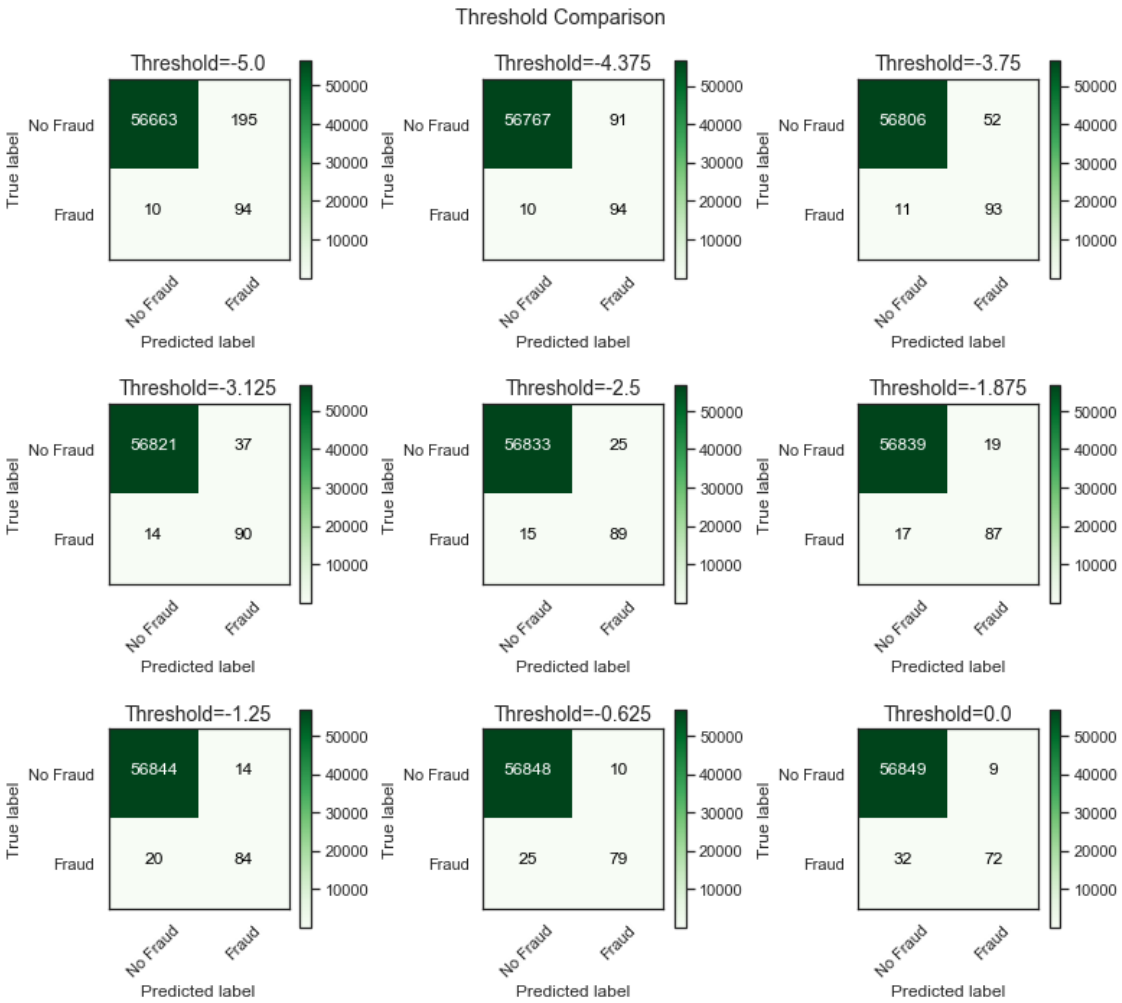


Figura 3.10: Efeitos do Threshold na Classificação

A figura 3.10 demonstra claramente como é possível ajustar as métricas de um modelo de acordo com o threshold. Analisando as matrizes, percebe-se que os números de TP, FP, FN e TN mudam de acordo com o ajuste do limiar, alterando assim métricas como Precision, Recall, Acurácia, F1-Score, entre outras.

Também é possível realizar um procedimento semelhante e avaliar, para cada threshold o resultado da curva Precision/Recall. Em Python, é possível fazer isso através do código abaixo:

```
# Precision x Recall
plt.figure(figsize=(10, 5))
i = 1
colors = ['darkgrey', 'gold', 'yellowgreen', 'azure', 'deepskyblue', '
    midnightblue',
          'blueviolet', 'violet', 'crimson']
for t in cf_mx_thresholds:
    custom_scores = np.where(y_scores > t, 1, 0)
    precisions, recalls, thresholds = precision_recall_curve(y_test,
    custom_scores)
    plot_precision_vs_recall(precisions, recalls, label=f'Threshold={t}',
                             color=colors[i-1])
    i+=1
```

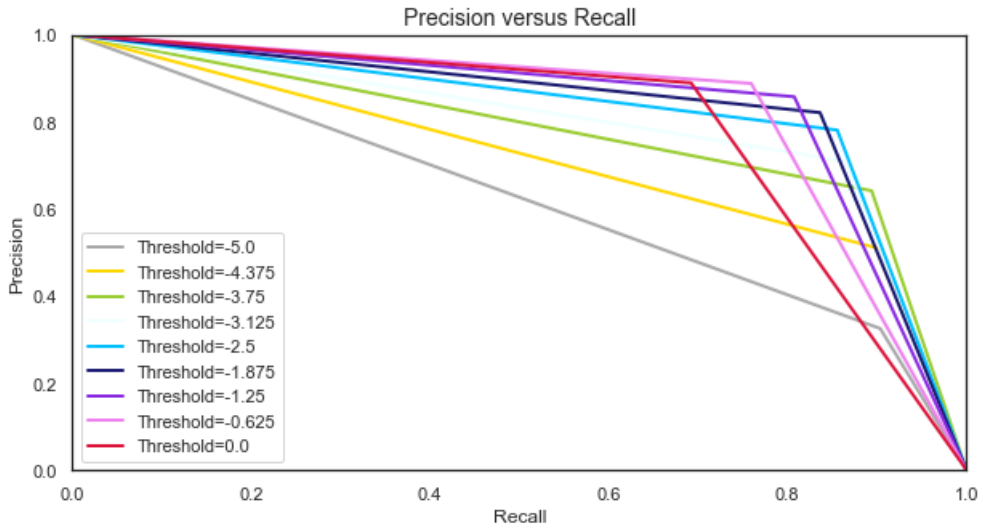


Figura 3.11: Thresholds e Curva Precision/Recall

3.5 Curvas de Calibração

Para introduzir a ideia das Curvas de Calibração, é preciso retomar alguns conceitos, já abordados neste capítulo, sobre a forma como classificadores normalmente retornam suas predições e como estas são analisadas através de métricas como Accuracy, Precision, Recall, F1-Score, entre outras.

Muitos modelos de classificação retornam predições em forma de probabilidades entre 0 e 1, sendo estas transformadas em predições de classes. Mesmo modelos que retornam somente *scores*, como SVMs, podem ser reformulados para produzirem predições em forma de probabilidades.

As métricas utilizadas em classificador binário avaliam a qualidade das saídas binárias 0s e 1s. Se as saídas não são binárias, mas sim pontos flutuantes entre 0 e 1, então estes podem ser utilizados como scores para um tipo de ranqueamento. Contudo, estes números remetem a probabilidades. Como confiar em tais probabilidades?

> **Corolário 3.5.1 — Exemplo na Meteorologia.** Imagine que um famoso site de clima mostra que, no próximo domingo, a chance de chuva é de 80%. O quão confiável são esses 80%? Se

entrarmos nesse site e verificar que 8 dos últimos 10 dias foram chuvosos quando a predição feita foi de 80%, então é possível dizer que o número é confiável. Analogamente, se olharmos 100 dias onde a predição foi de 80%, o esperado é que em 80 deles a chuva realmente aconteceu. Da mesma forma, se o site diz que a chance de chuva é de 30%, temos que, em 100 vezes onde a predição foi exatamente de 30%, a chuva aconteceu em 30. Se esse padrão se segue para outros ranges, diz-se que as predições estão **calibradas**. É o modo probabilístico de dizer "acertou na mosca".

3.5.1 Entendendo a Curva de Calibração

Como verificar se um modelo está ou não calibrado? Ao invés de sumarizar a calibração em uma única métrica, é preferível plotar as *curvas de calibração*. Abaixo, um exemplo:

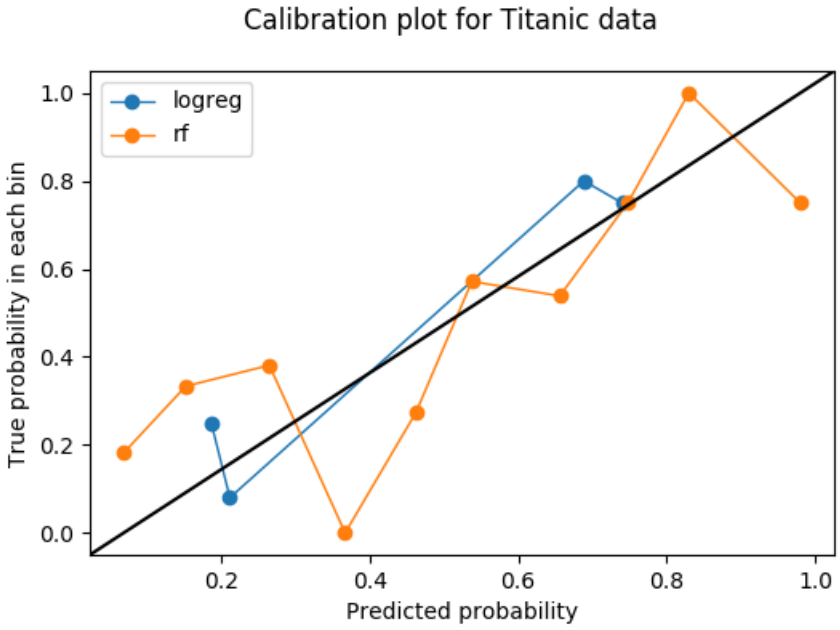


Figura 3.12: Exemplo de Curva de Calibração (Log Reg e Rnd Forest)

As curvas de calibração normalmente são formadas por *line plots* e dividida em um número n de *bins* (faixas). Cada faixa é convertida em um ponto da curva e expressa, no eixo y, a proporção de probabilidade das saídas do modelo. Já no eixo x tem-se o valor médio das probabilidades preditas pelo modelo. Um modelo bem calibrado possui curva de calibração próxima a reta diagonal $y = x$.

Uma outra definição defende que as curvas de calibração (ou Reliability Diagrams) são curvas lineares que mostram a frequência relativa do que foi observado (eixo y) versus a frequência de probabilidade predita (eixo x).

Corolário 3.5.2 — Bins (Faixas). As probabilidades preditas pelo modelo são divididas em um número fixo de buckets (bins) ao longo do eixo x. Após isso, é contado o número de resultados da classe positiva para cada faixa (i.e. a frequência relativa observada). Por fim, essa contagem é normalizada e o resultado é plotado em uma curva linear.

Veja o exemplo abaixo de uma curva de calibração plotada para diversos modelos de classificação:

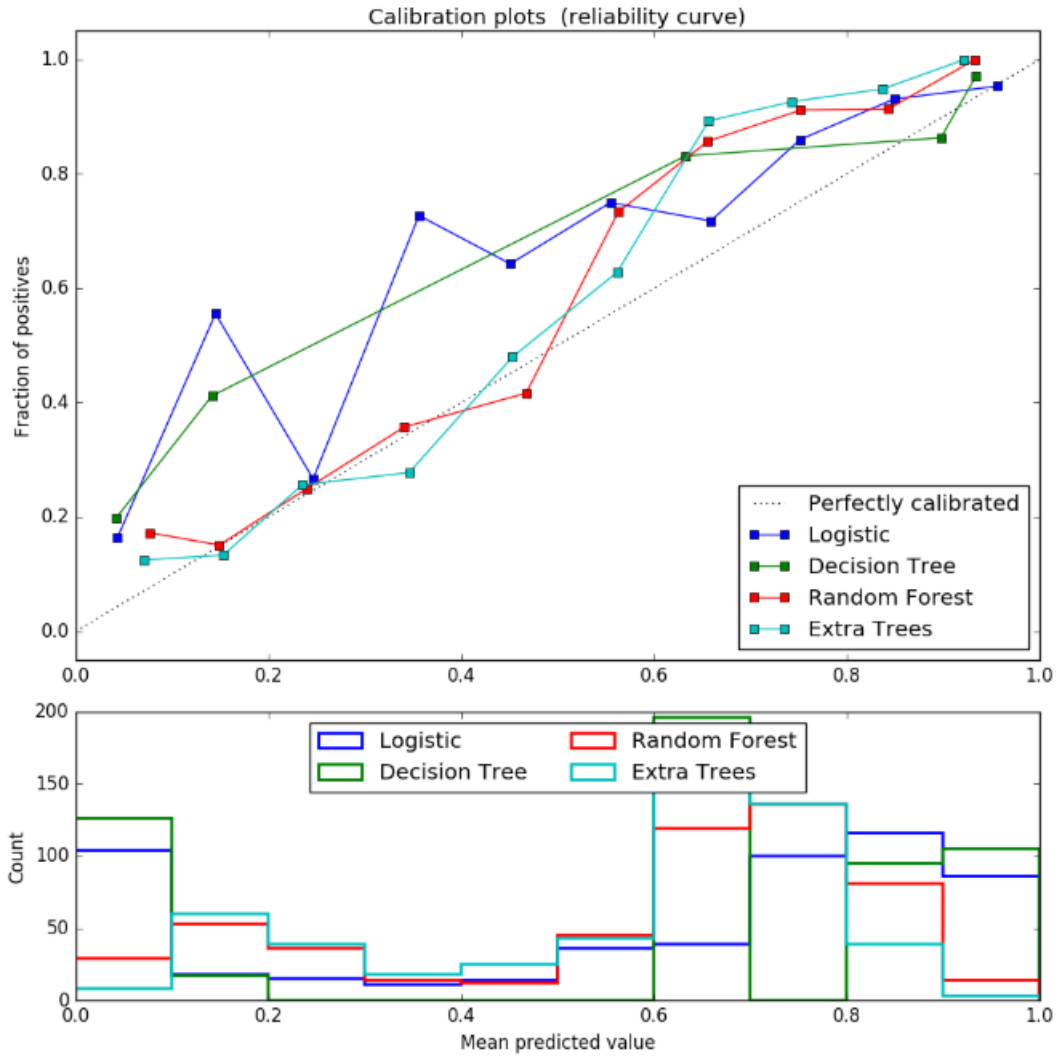


Figura 3.13: Exemplos de Curvas de Calibração

- **Eixo X - Mean Predicted Value:** indica a probabilidade média (divida em um número n de faixas)
- **Eixo Y - Fraction of Positives:** indica, pra faixa de probabilidade em questão, qual a fração de predição da classe positiva

Exemplificando utilizando a figura acima, o segundo ponto da curva que define o modelo Decision Trees (verde), tem-se uma fração da classe positiva de aproximadamente 0,4 para uma média de probabilidades próxima de 0,15. Em outras palavras, significa dizer que, para este modelo, 40% das instâncias com probabilidade 0,15 é da classe positiva. Não é um número satisfatório e, por conta disso, este ponto está bem longe do ideal (linha $y = x$ que indica um valor aproximado de 0,15 para essa mesma faixa).

Neste ponto:

- **Ideal:** espera-se que, para uma probabilidade média de 0,15, se tenha 15% das predições sendo da classe positiva
- **Verificado em DTree:** para uma probabilidade média de predição igual a 0,15, tem-se 40% das predições sendo da classe positiva

3.5.2 Características das Curvas

Continuando com a saga de entendimento das curvas de calibração, o exemplo abaixo mostra uma comparação mais justas de curvas plotadas para diferentes modelos. Para a plotagem, foram utilizadas 100.000 instâncias do dataset *make classification* importado diretamente do sklearn.

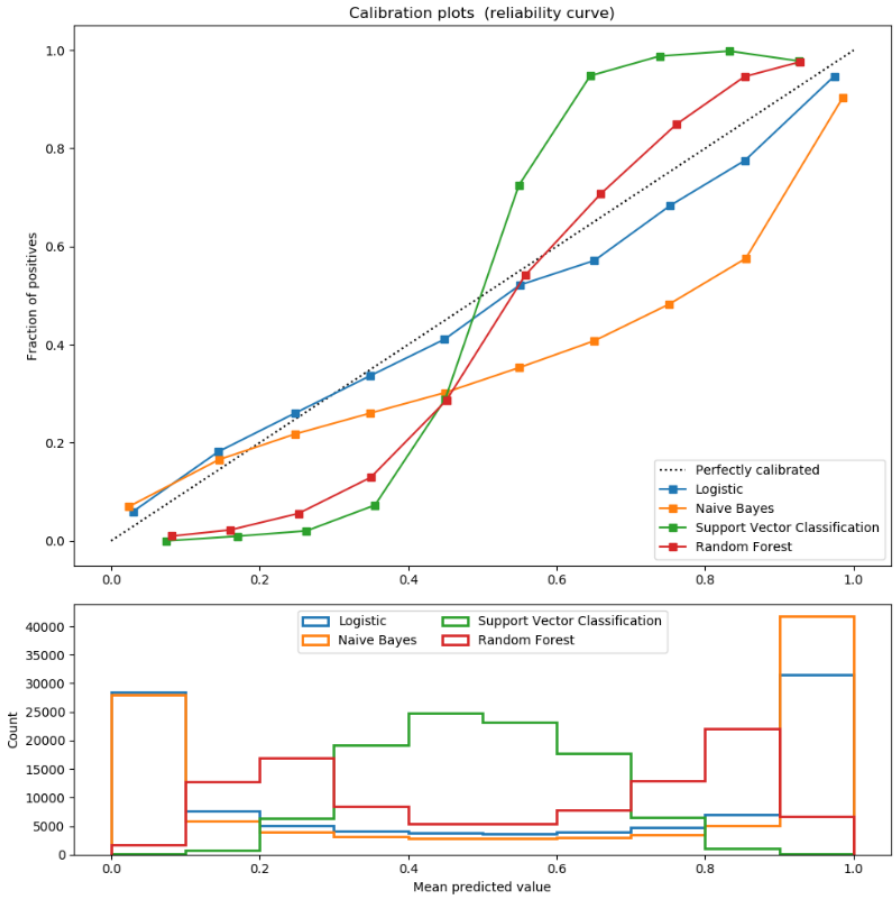


Figura 3.14: Curvas para Diferentes Modelos

- O modelo de **Regressão Logística** retorna predições bem calibradas, uma vez que sua otimização é baseada na função log-loss, não produzindo assim probabilidades enviesadas.
- O modelo **Naive Bayes (GaussianNB)** tende a retornar probabilidades próximas de 0 ou 1 (notar as contagem presentes no histograma). Muito disso devido as suposições feitas pelo modelo de que as features são condicionalmente independentes, o que não é verdade em casos onde há features redundantes ou com alta correlação.
- **Random Forest** apresentam um comportamento inverso: o histograma mostra picos em probabilidades próximas de 0.2 e 0.9, enquanto probabilidades 0 e 1 são bem raras. Existe uma explicação mais detalhada para isso, mas esta basicamente refere-se a dificuldade de modelos bagging e de florestas em predizer valores próximos de 0 ou 1 devido a veriância dos modelos base (árvores).
- Já o modelo **SVM** mostra uma curva ainda mais sigmoidal que a Random Forest, o que é típico deste modelo que utiliza maximização de margens de decisão, focando em amostras que estão próximas dos vetores de suporte (fronteiras de decisão).

Com relação aos pontos apresentados nas curvas, tem-se:

- **Pontos abaixo da diagonal:** o modelo apresentou um over-forecast; as probabilidades são muito elevadas
- **Pontos acima da diagonal:** o modelo apresentou um under-forecast; as probabilidades são muito baixas

3.5.3 Calibrando Probabilidades

Após a passagem por conceitos importantes sobre as curvas de calibração, é necessário entender como atuar de modo a "consertar"eventuais probabilidades diferentes da esperada.

> **Corolário 3.5.3 — Reforçando o Objetivo.** Um modelo de classificação binária bem calibrado deve classificar as amostras de tal forma que, entre aquelas que retornaram uma probabilidade de predição de, por exemplo, 0.8 (predict_proba()), aproximadamente 80% destas realmente pertençam a classe positiva.

Para plotar uma curva de calibração, em Python, é necessário utilizar o código abaixo:

```
from sklearn.calibration import calibration_curve

# Retornando as probabilidades
probs = model.predict_proba(X_test)[:, 1]
frac_of_positives, mean_pred_values = calibration_curve(y_test,
                                                        probs,
                                                        n_bins=10)
fig, ax = plt.subplots()
ax.plot(mean_pred_values, frac_of_positives)
```

Os modelos plotados nas figuras 3.14 e 3.13 estão descalibrados (uns mais e outros menos, lembrando que LogisticRegression é o que menos sofre com isso). Para consertar este bias, existem pelo menos duas abordagens: Platt Scaling e Isotonic Regression.

Platt Scaling

Nessa abordagem clássica, é aplicada uma transformação logarítima nas probabilidades de saída do classificador. Observando as curvas da figura 3.14 para os modelos SVM e Random Forest, percebemos um formato de S característico de uma curv a sigmoidal. A transformação logarítima é uma boa forma de tratar erros nesse formato, suavizando as probabilidades em uma distribuição linear e resultando em números próximos do esperado.

Isotonic Regression

Porém, nem sempre a curva de calibração de um modelo terá um formato sigmoidal. A abordagem não-paramétrica conhecida como Istonic Regression é mais complexa e requer uma maior quantidade de dados (caso contrário, corre risco de overfitting), entretanto suporta curvas dos mais diferentes shapes. Observando a curva do modelo Naive Bayes, é possível afirmar que houve um under-forecast para baixas probabilidades e um over-forecast para altas probabilidades. Em situações desse tipo, a calibração Platt não é tão eficiente quanto a Isotonic Regression.

Exemplos de Calibração

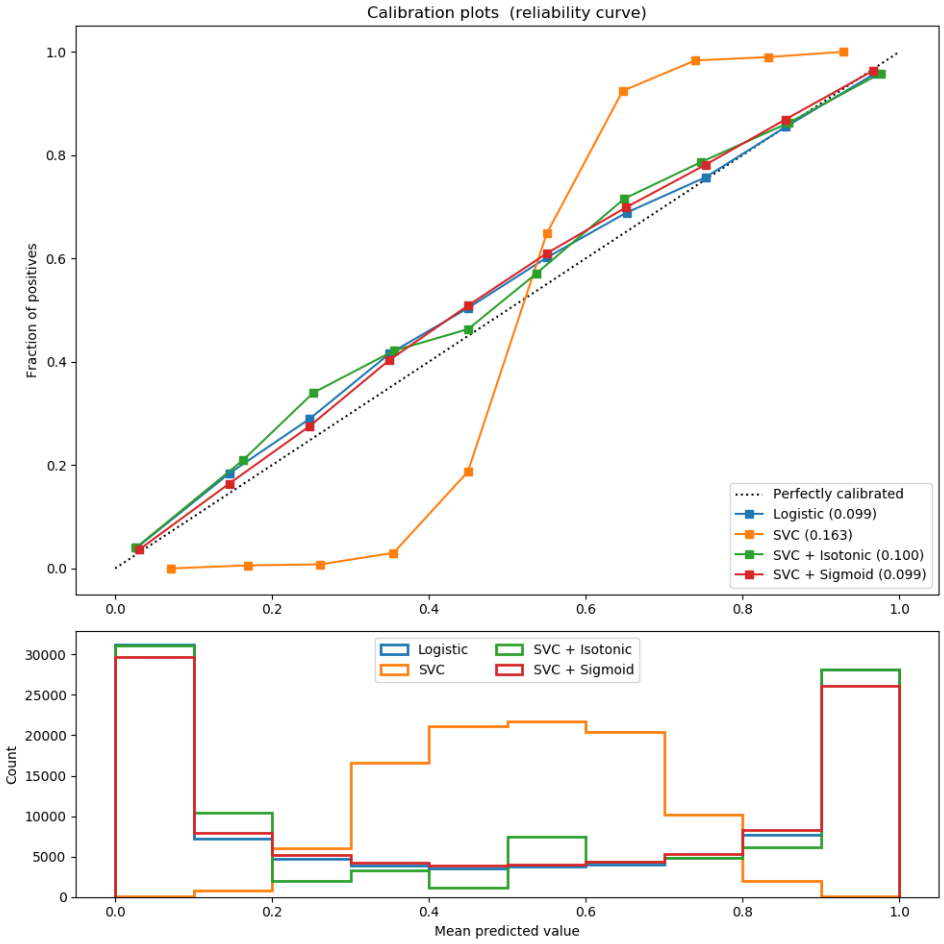


Figura 3.15: Calibrando Curva para LinearSVC

A figura 3.15 mostra um exemplo de calibração de curva para o modelo LinearSVC utilizando as duas abordagens tratadas neste capítulo: Platt (Sigmoid) e Isotonic. A partir dela, é possível perceber que a curva sigmoidal do modelo "original"foi calibrada utilizando as duas abordagens.

Na prática, o método Isotonic Regression funciona bem para grande parte dos casos e não possui restrições. Porém, é preciso atentar-se a possibilidade deste causar um overfitting nos dados. Por outro lado, como evidenciado na definição acima, o método Platt Scaling (Sigmoid) funciona muito bem para curvas de classificadores under-confident, como no caso do LinearSVC.

Abaixo, uma relação das métricas do modelo antes e após a calibração em cada abordagem.

Model	Brier	Precision	Recall	F1-Score
Logistic Regression	0.099	0.872	0.851	0.862
SVC	0.163	0.872	0.852	0.862
SVC + Isotonic	0.100	0.853	0.878	0.865
SVC + Sigmoid	0.099	0.874	0.849	0.861

Tabela 3.2: Calibração e Métricas para LinearSVC

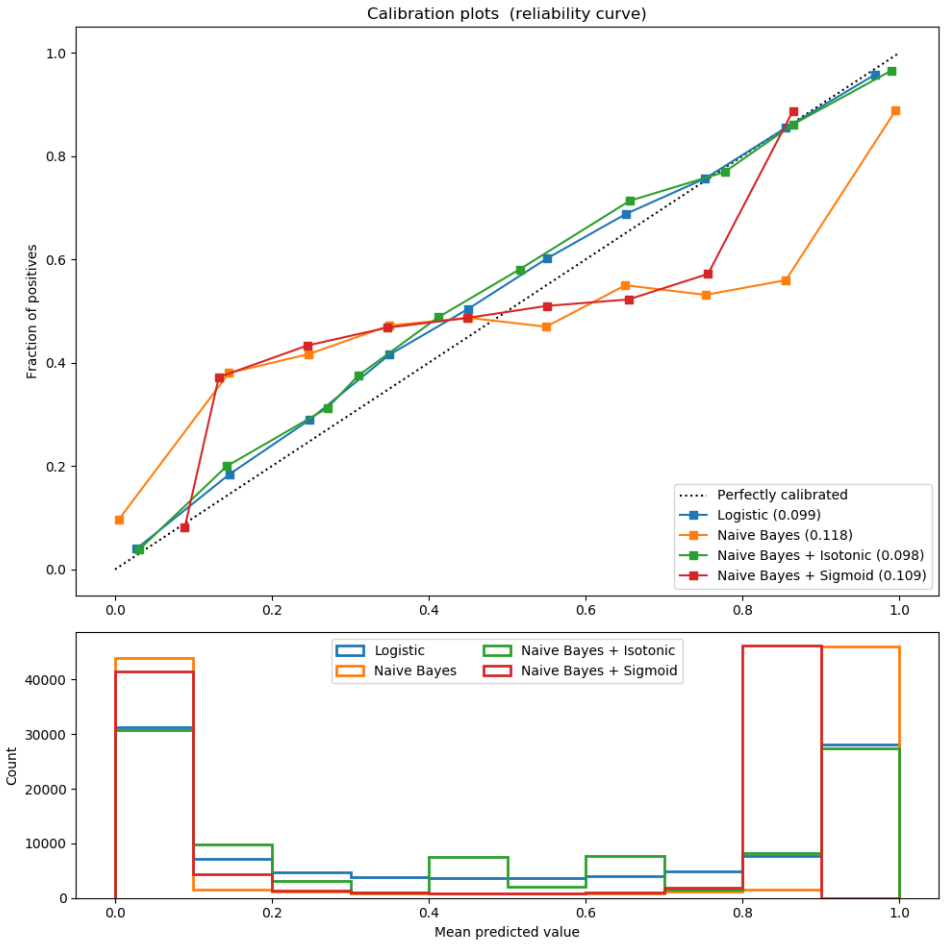


Figura 3.16: Calibrando Curva para GaussianNB

Acima, é possível visualizar a calibração de curva para um modelo Naives Bayes utilizando Platt Scaling e Isotonic Regression. Como mencionado na definição, o método Platt não surtiu efeito na calibração da curva deste modelo, dado que este apresenta um comportamento over-confident, causando um under-forecasting para baixas probabilidades e um over-forecasting para altas probabilidades.

Uma forma mais fácil de perceber isso é através do formato "não-sigmoidal"da curva de calibração original do modelo. Aqui, o método Isotonic Regression atuou de forma positiva na calibração da curva, sendo preferível para este tipo de modelo.

Model	Brier	Precision	Recall	F1-Score
Logistic Regression	0.099	0.872	0.851	0.862
Naive Bayes	0.118	0.857	0.876	0.867
Naive Bayes + Isotonic	0.098	0.883	0.836	0.859
Naive Bayes + Sigmoid	0.109	0.861	0.871	0.866

Tabela 3.3: Calibração e Métricas para Naive Bayes

A partir da obervação das métricas, percebe-se que a calibração da curva não é um meio direto de aumentar a performance de modelos. De fato, isso é relatado em documentos oficiais. Estudiosos dizem que os modelos que mais se beneficiam de calibrações são modelos Ensemble.

3.6 Links Úteis

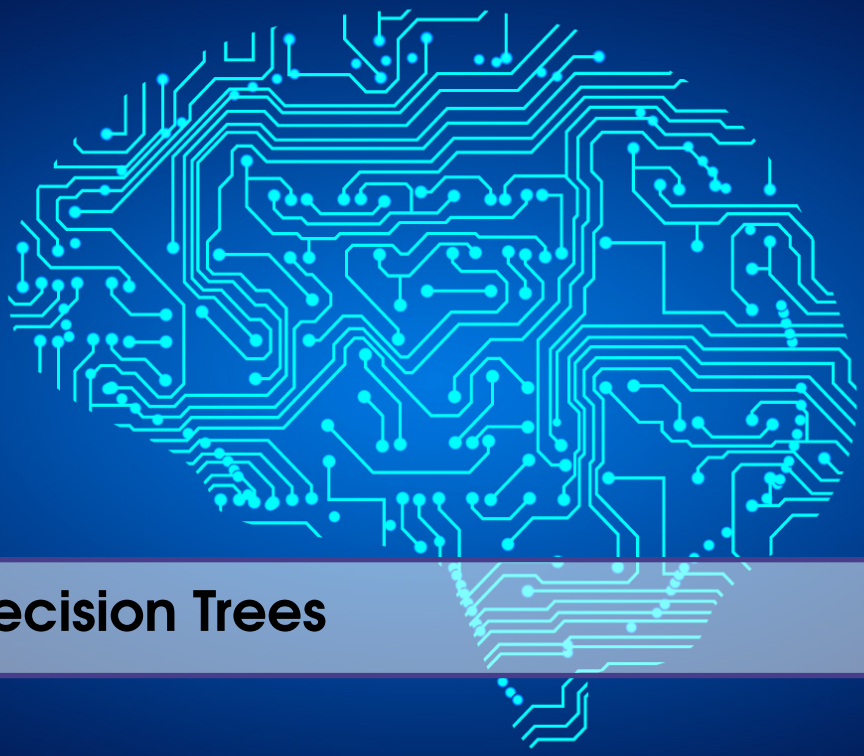
3.6.1 Métricas de Avaliação

- link-BecomingHuman-Understand-Classification-Performance-Metrics
- link-Medium-Performance-Metrics-for-Classification
- link-TowardsDS-Common-Classification-Model-Evaluation-Metrics
- link-Sklearn-Model-Evaluation

3.6.2 Curvas de Calibração

- link-Sklearn-Isotonic-and-Sigmoid-Calibration-GaussianNB-LinearSVC
- link-Kaggle-Notes-on-Classification-Probability-Calibration
- link-ChangHsinLee-A-Guide-to-Calibration-Plots
- link-Sklearn-Comparison-of-Calibration-of-Classifiers
- link-MLMastery-How-and-when-Calibrate-Classification-Model

II Modelos em Detalhes



4. Decision Trees

As Árvores de Decisão são algoritmos versáteis de Machine Learning que podem ser utilizados em problemas de classificação e regressão. Neste capítulo, serão discutidos alguns detalhes sobre o treinamento, visualizações e predições envolvendo Árvores de Decisão (ou Decision Trees).

4.1 Representação do Modelo

Como sugere a nomenclatura, as Árvores de Decisão podem ser definidas como estruturas baseadas em fluxos onde cada nó representa uma espécie de "teste"em um determinado atributo, resultando em diferentes caminhos de acordo com a resposta obtida.

- Um nó interno denota um teste a ser realizado sobre um determinado atributo
- Uma derivação representa um resultado do teste
- As folhas representam os rótulos da classe

Assim, tendo posse da variável target pré-definida, o algoritmo separa a população (ou amostra) em dois ou mais sets homogêneos (ou sub-populações) baseado em um nível de significância dessa separação. A figura abaixo irá auxiliar no entendimento desse fluxo bem como nos termos comumente utilizados na definição de elementos envolvendo Árvores de Decisão:

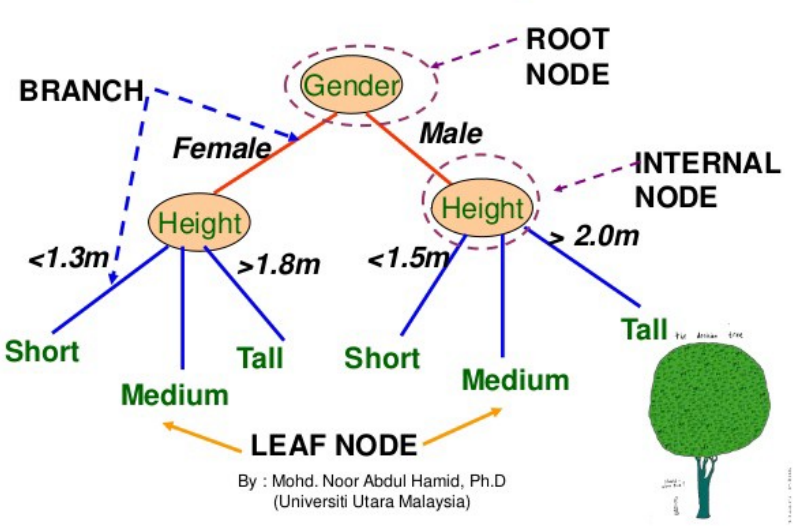


Figura 4.1: Diagrama Decision Trees

- **Root Node:** Representa a população ou amostra que será dividida em dois ou mais subsets
- **Splitting:** Processo de separação de um nó em dois ou mais sub-nós
- **Decision Node:** Quando um sub-nó é dividido em outros sub-nós, então este é chamado de Decision Node (nó de decisão)
- **Leaf/Terminal Node:** Representa um nó que não gera mais divisões (nó folha)
- **Pruning:** Poda da árvore. Geralmente trata-se da remoção de sub-nós de um nó de decisão cujos atributos não são representativos para o modelo
- **Branch/Sub Tree:** Representa uma sub-sessão da árvore
- **Parent and Child Node:** Um nó que é dividido em sub-nós é chamado de *parent node* (nó pai) destes. Os sub-nós gerados são *child node* (nós filho) do nó que gerou esta divisão

Corolário 4.1.1 — Geração das Árvores. Basicamente, é possível resumir o processo de geração de Árvores de Decisão em duas etapas:

- **Construção da Árvore**
  - No início, todos os exemplos de treinamento estão no nó raíz (root node)
  - Particionam-se os exemplos de maneira recursiva, baseado em determinados atributos
- **Poda da Árvore**
  - Identificar e remover ramos que refletem o ruído ou outliers

Na prática, é possível treinar e visualizar as decisões tomadas pelo algoritmo Decision Trees através do sklearn. No exemplo abaixo, o dataset iris será utilizado para treinar um modelo simples e visualizar o gráfico de decisão gerado.

```
# Importando bibliotecas
from sklearn.datasets import loead_iris
from sklearn.tree import DecisionTreeClassifier, export_graphviz
import os

# Lendo dataset
iris = load_iris()
X = iris.data[:, 2:] # utilizando apenas petal lenght e petal width
y = iris.target
```

```
10
11 # Treinando modelo
12 tree_clf = DecisionTreeClassifier(max_depth=2) # restricao
13 tree_clf.fit(X, y)
14
15 # Definindo diretorio para salvar figuras
16 root_dir = os.getcwd()
17 def image_path(fig_id):
18     return os.path.join(root_dir, fig_id)
19
20 # Gerando visualizacao da arvore
21 export_graphviz(
22     tree_clf,
23     out_file=image_path('iris_tree.dot'),
24     feature_names=iris.feature_names[2:],
25     class_names=iris.target_names,
26     rounded=True,
27     filled=True
28 )
29
30 # Visualizando arvore treinada
31 from IPython.display import Image
32 !dot -Tpng iris_tree.dot -o iris_tree.png -Gdpi=600
33 Image(filename='iris_tree.png', width=450, height=450)
```

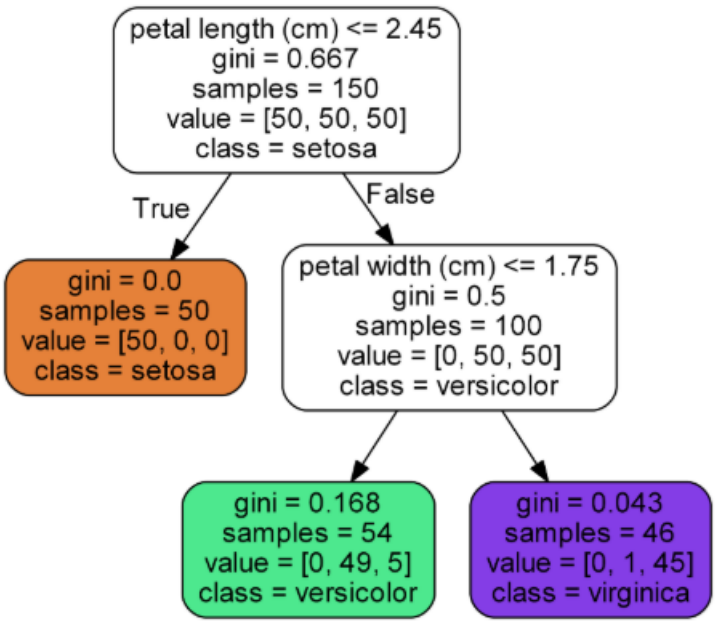


Figura 4.2: Árvore de Decisão Treinada

4.2 Predições

A árvore plotada acima será utilizada para exemplificar a maneira como o algoritmo Decision Trees realiza predições. Suponha então que uma nova flor seja recebida como exemplo e a intenção seja classificá-la entre as classes disponíveis: setosa, versicolor ou virginica.

- Iniciando no **nó raíz** (root node - *depth 0, top*): o teste proposto neste nó verifica se o elemento de entrada *(iris flower)* possui um valor para o atributo *petal length* menor ou igual a 2.45. Em caso positivo, então o fluxo segue para o **nó folha** (leaf node - *depth 1, left*). Por

característica, um nó folha não possui nenhuma ramificação e, portanto, não propõe novos testes. Assim, basta visualizar a classe predita (neste caso, class=setosa).

- Supondo um outro exemplo com um valor de *petal width* maior que 0.8, o deslocamento do nó raiz encontra um nó que não é um nó folha *(depth 1, right)*. Dessa forma, o novo teste proposto questiona se o valor de *petal width* é menor ou igual a 1.75. Se sim, trata-se de uma virginica. Se não, trata-se de uma versicolor.

Em cada nó é possível visualizar:

- **samples=** quantidade de amostras contidas no nó
- **value=** indica quantas amostras pertencem a classe do nó em questão
- **class=** classe predita pelo nó (aplicada apenas a nós folha)
- **gini=** índice gini para medir *impureza* (nó puro, gini=0)

Corolário 4.2.1 — Algoritmo de Hunt. Considerando uma árvore sem restrições, X_t como sendo o conjunto de dados de treinamento no nó t e $y = y_1, ..., y_c$ os rótulos das classes, eis como o modelo trabalha: **Passo 1:** Se todos os dados em X_t pertencem a uma mesma classe y_t, então t é um nó folha e deve ser rotulado como y_t **Passo 2:** Se X_t contém dados que pertencem a mais de uma classe:

- Selecione um atributo que será utilizado no teste para particionar os dados em subconjuntos menores
- Crie um nó filho para cada possível resultado do teste
- Aplique o algoritmo recursivamente para cada nó filho

4.3 Medidas de Impureza

Após o entendimento do funcionamento das Árvores de Decisão, uma pergunta plausível poderia ser relacionada forma como a árvore constitui os testes em cada nó. O modelo aplica divisões baseadas na melhor bifurcação possível. Porém, o que define uma boa separação? A resposta para essa pergunta é dada pelo índice de *impureza* Gini ou pela Entropia.

Definição 4.3.1 — Impureza Gini.

$$G_i = 1 - \sum_{k=1}^{n} p_{i,k}^2 \tag{4.1}$$

Definição 4.3.2 — Entropia.

$$H = -\sum_{k=1}^{n} p_{i,k} log\left(p_{i,k}\right) \tag{4.2}$$

O elemento $p_{i,k}$ também pode ser encontrado em outras fontes como $p(i|t)$ e indica a fração de dados da classe t (ou k) no nó i. No algoritmo DecisionTree do Scikit-Learn, a impureza Gini é utilizada como padrão, porém é possível utilizar a Entropia através do hiperparâmetro *criterion=entropy*.

Retomando a figura 4.2, é possível perceber a medida da impureza Gini em cada nó considerando as três classes utilizadas.

No nó raíz, a divisão de classes é dada por [50, 50, 50], ou seja, existem exatamente 50 elementos para cada uma das classes y_c. Nessas condições, a impureza Gini é igual a 0.667, pois:

$$G_0 = 1 - \left(\frac{50}{150}\right)^2 - \left(\frac{50}{150}\right)^2 - \left(\frac{50}{150}\right)^2 = 0.667$$

Da mesma forma, percebe-se que a impureza Gini é igual a 0 no nó folha a esquerda, dado que a divisão de amostras por classe resulta em [50, 0, 0], ou seja, um nó totalmente puro com apenas elementos de uma única classe (setosa).

Como um último exemplo, apenas para captar a ideia, o nó folha que identifica a classe *versicolor* possui os valores [0, 49, 5], indicando que, das 54 amostras disponíveis no nó (samples=54), 49 foram classificadas como versicolor e 5 classificadas como virginifica. O índice Gini nesse nó é:

$$G_0 = 1 - \left(\frac{0}{54}\right)^2 - \left(\frac{49}{54}\right)^2 - \left(\frac{1}{54}\right)^2 = 0.168$$

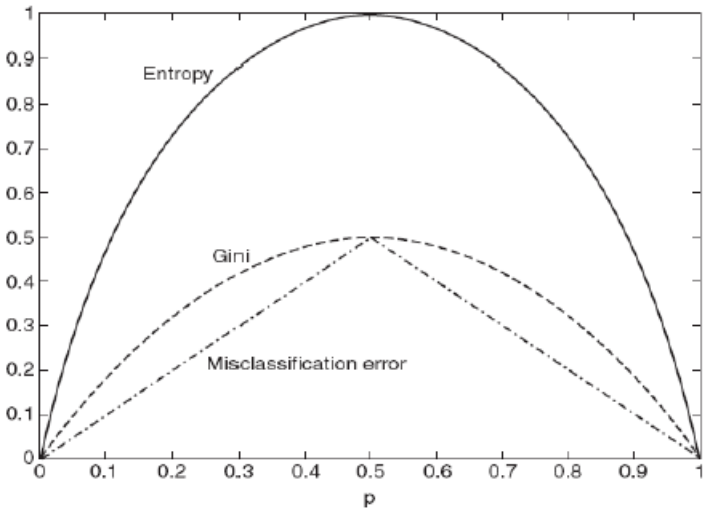


Figura 4.3: Medição de Impureza Gini e Entropia

Um exemplo de fronteira de decisão de um algoritmo de Árvore treinado:

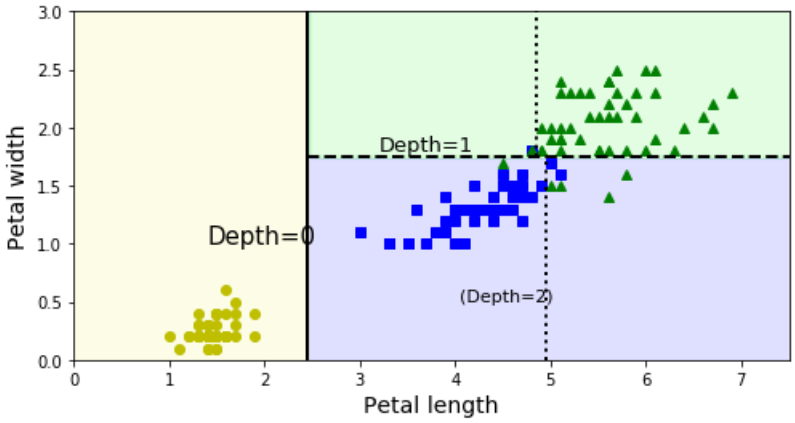


Figura 4.4: Fronteira de Decisão - Algoritmo Decision Trees

4.4 Estimando Probabilidades

Uma Árvore de Decisão pode estimar a probabilidade de uma instância pertencer a uma determinada classe k: primeiro, o modelo percorre a árvore para encontrar o nó folha referente a instância em questão (realizando todas as validações e testes necessários até que se chegue a um resultado mostrado por um nó folha), retornando assim a proporção entre o total de instâncias da classe k neste nó.

Por exemplo, se o mesmo modelo exemplificado pela figura 4.2 for utilizado para prever a classe referente a uma instância (iris flower) com *petal length* maior que 2.45 e *petal width* menor que 1.75, as probabilidades são:

- 0% para Iris-Setosa (0/54);
- 90.7% para Iris-Versicolor (49/54);
- 9.3% para Iris-Virginica (5/54)

Assim, o modelo irá retornar a classe Iris-Versicolor como sendo a mais provável para esta instância. Como visto em alguns breves momentos no Capítulo 3 - Regressão Logística, a predição de probabilidades pode ser dada através do método *predict_proba()* de um modelo de classificação já treinado.

```
# Retornando as probabilidades
probs = tree_clf.predict_proba([[5, 1.5]])
print(probs)
```

```
# Vejamos
tree_clf.predict_proba([[5, 1.5]])
```

executed in 6ms, finished 09:23:07 2019-05-20

```
array([[0.        , 0.90740741, 0.09259259]])
```

```
# Predição
tree_clf.predict([[5, 1.5]])
```

executed in 266ms, finished 09:23:07 2019-05-20

```
array([1])
```

Figura 4.5: Probabilidades e Classe Predita

4.5 O Algoritmo CART

O Scikit-Learn utiliza o algoritmo CART (Classification And Regression Tree) para treinar o modelo de Árvores de decisão (também chamado de "growing trees"). A ideia é bem simples: primeiro, o algoritmo separa o set de treino em dois usbsets usando uma única feature k e um threshold t_k (por exemplo, ***"petal width <= 0.8"***. Mas, como determinar k e t_k? O modelo procura o par (k, t_k) que produz o **mais puro** subset (ponderado pelo tamanho do subset). Para isso, é definida uma função custo:

Definição 4.5.1 — Custo Separação.

$$J(k,t_k) = \frac{m_{left}}{m} G_{left} + \frac{m_{right}}{m} G_{right} \tag{4.3}$$

$$\begin{cases} G_{\text{left/right}} \text{ mede a impureza do subset esquerdo e direito} \\ m_{\text{left/right}} \text{ é o numero de instâncias no subset esquerdo e direito} \end{cases} \tag{4.4}$$

Uma vez separado o set em dois, o modelo então separa os subsets gerados utilizando a mesma lógica, gerando mais dois subsets. E assim, recursivamente até que a limitação imposta pelo parâmetro ***max_depth*** seja atingida, ou até encontrar um split que não reduza a impureza. Alguns outros parâmetros atuam como "agentes reguladores"dessa separação":

- min_samples_split
- min_samples_leaf
- min_weight_fraction_leaf
- max_leaf_nodes

O algoritmo CART é "insaciável"(guloso também pode ser utilizado aqui): ele procura incansavelmente por um split ótimo no primeiro nível; e então repete o processo para cada nível subsequente. O algoritmo não verifica se o próximo split irá levar a uma possível impureza baixa nos níveis mais abaixos. Algoritmos desse tipo produzem boas soluções, mas sem a garantia de que essas boas soluções sejam realmente ótimas soluções.

4.6 Regularizando Hiperparâmetros

Este é um tópico extremamente útil e importante. Se a ideia do algoritmo Decision Trees foi bem entendida, foi possível perceber que, se nenhuma restrição for aplicada, o algoritmo simplesmente irá procurar incessantemente pela melhor solução de acordo com o input (dados de treino) fornecido. Isto soa familiar e a nomenclatura para tal é bem conheida: *overfitting*.

Normalmente, modelos baseados em árvores possuem tendências ao overfitting. Sem restrições, a adequação do modelo aos dados é feita de forma praticamente perfeita, tentando captar toda e qualquer nuance envolvendo atributos fora do padrão (os famosos *outliers*). Portanto, é de extrema importância conhecer e regularizar o treinamenteo com as Árvores de Decisão a partir de seus hiperparâmetros, sendo alguns deles:

- ***max_depth*=** restringe a profundidade máxima da árvore
- ***min_samples_split*=** número mínimo de amostras por nó antes deste poder ser separado
- ***min_samples_leaf*=** número mínimo de amostras um nó folha deve ter
- ***min_weight_fraction_leaf*=** semelhante ao *min_samples_leaf*, porém em porcentagem do total
- ***max_leaf_nodes*=** número máximo de leaf nodes
- ***max_features*=** número máximo de features avaliadas para a separação de cada nó

Para exemplificar a importância dessa regularização da árvore, abaixo é possível visualizar dois modelos treinados, sendo o primero sem nenhuma restrição e o segundo com o hiperparâmetro *min_samples_leaf=4* (lembrando que este determina o número mínimo de amostras que um nó deve ter).

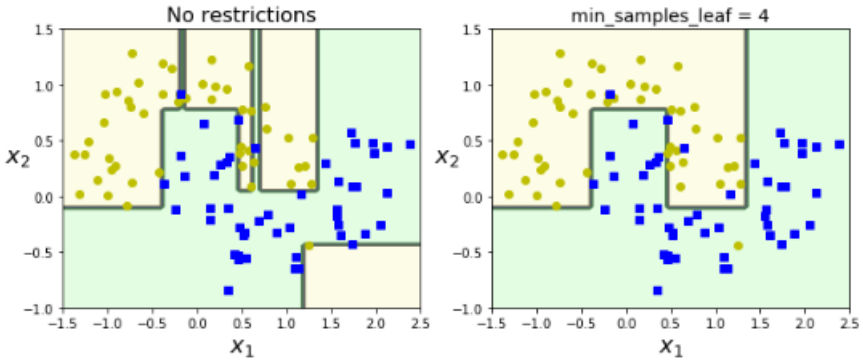


Figura 4.6: Árvores de Decisão com e sem Restrições

4.7 Regressão

Ao contrário do que muitos imaginam ao estudar Árvores de Decisão, não somente problemas de classificação podem ser resolvidos, mas também problemas de regressão, onde a variável target é, na verdade, quantitativa. De toda forma, ainda sim é possível reunir certa confusão ao tentar imaginar as árvores sendo usadas para predições numéricas (pelo menos com o que foi visto até este momento). Assim, a figura abaixo irá exemplificar a estrutura utilizada por algoritmos de Árvores de Decisão em problemas de Regressão.

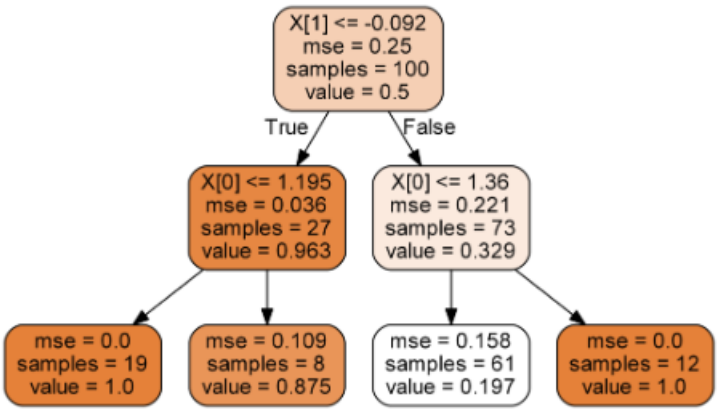


Figura 4.7: Decision Trees Regressor

A árvore gerada segue os mesmos princípios das árvores obtidas em modelos de classificação. Qualquer amostra deve atravessar as ramificações de acordo com as verificações impostas e, ao final, ser classificada dentro de um nó onde o atributo *samples* indica a quantidade de amostras presentes naquele nó com um **mse** (mean squared error) definido.

A figura abaixo mostra a curva obtida a partir do treinamento de um modelo Decision Tree Regressor utilizando dados sintéticos com uma característica de um grau polinomial quadrático.

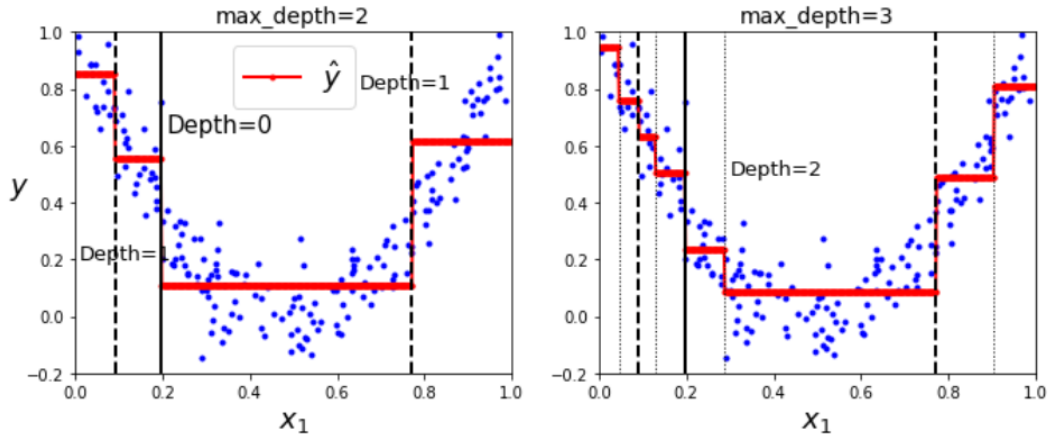


Figura 4.8: Curva Decision Trees Regressor

O algoritmo CART funciona de forma semelhante para problemas de regressão, exceto pelo fato de que, ao invés de tentar separar os dados de modo a minimizar a impureza, agora tenta separar os dados de forma a minimizar o MSE. Assim como eme modelos de classificação, Árvores de Decisão em regressão também são propensas ao overfitting. Sem restrições, isto é, utilizando os valores *default* para os hiperparâmetros, a curva obtida fatalmente terá um comportamento inadequado.

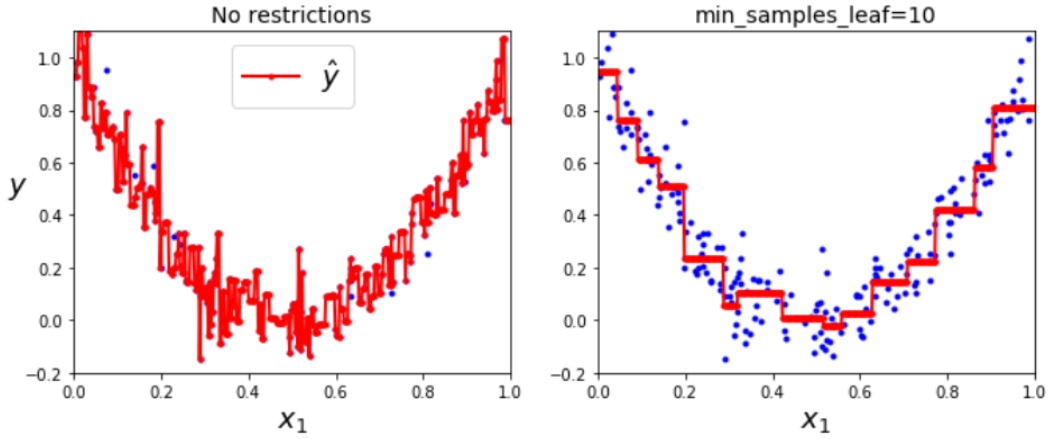


Figura 4.9: Overfitting Decision Trees Regressor

4.8 Vantagens e Desvantagens

Ao entrar em detalhes sobre algoritmos de Árvores de Decisão, foi possível notar algumas particularidades realmente interessantes dentro das possibilidades de se encontrar funções que descrevam comportamentos singulares. A seguir, serão pontuadas algumas das vantagens e desvantagens desse tipo de abordagem.

4.8.1 Vantagens

- **Fácil interpretação:** É nítido que modelos de Árvore de Decisão são fáceis de entender e interpretar dentro do contexto ao qual são aplicados. Intuitivos.

- **Pouca preparação dos dados:** Comparado com outros modelos, as Árvores de Decisão requerem apenas uma pouca dedicação a preparação dos dados.
- **Relevância das features:** Modelos baseados em árvores podem indicar as features mais importantes em problemas onde muitos atributos estão presentes.

4.8.2 Desvantagens

- **Overfitting:** Sem dúvida, este é um dos principais problemas envolvendo Árvores de Decisão. As relações complexas aprendidas durante o treinamento podem não generalizar tão bem para outros sets. A restrição e tunagem dos hiperparâmetros pode resolver este problema.
- **Instabilidade:** As árvores podem se tornar instáveis por conta de pequenas variações nos dados, podendo resultar em modelos totalmente diferentes ao serem treinadas novamente. A isto se dá o nome de variância, a qual pode ser mitigada em modelos como *bagging* ou *boosting*.
- **Gula:** Como mencionado anteriormente, modelos baseados em árvores são "gulosos"e, portanto, não há garantia de que o mínimo global será atingido.
- **Limitação:** É possível dizer que os algoritmos de árvores de decisão estão limitados a hipóteses a hiper-retângulos (linhas ortogonais).

4.9 Links Úteis

- link-githubThiagoPanini-DecisionTrees
- link-Medium-DecisionTrees
- link-TowardsDS-DecisionTrees
- link-UFABC-SistemasInteligentes-Aula2

4.10 Exercícios

Os exercícios a seguir foram retirados do livro *Hands-On Machine Learning with Scikit-Learn & TensorFlow*. As respostas para alguns deles irão depender de pesquisas em outras fontes.

Exercício 4.1 Qual a profundidade aproximada de uma Árvore de Decisão treinada (sem restrições) em um set de 1 milhão de instâncias? ■

A profundidade de uma árvore binária e balanceada contendo m amostras é igual a $log_2(m)$. Portanto, com 1 milhão de instâncias, a profundidade aproximada é $log_2(10^6) \approx 20$

Exercício 4.2 A impureza Gini de um nó é menor ou maior que o nó pai usado para gerá-lo? É *geralmente* menor/menor ou *sempre* menor/maior? ■

No geral, a impureza Gini dos nós filhos são menores do que a impureza Gini dos nós pais. isto devido as funcionalidades do algoritmo CART utilizado pelas Árvores de Decisão treinadas via Scikit-Learn. Entretanto, nem sempre isso é verdade. Existem alguns exemplos que mostram que as ramificações podem possuir índices Gini maiores (o que não é adequado, porém acontece).

Exercício 4.3 Se uma Árvore de Decisão causa overfitting no set de treino, é uma boa ideia diminuir o hiperparâmetro *max_depth*? ■

Sim, visto que o hiperparâmetro *max_depth* irá restringir o modelo e aplicar regularização na árvore.

Exercício 4.4 Se uma Árvore de Decisão causa underfitting no set de treino, é uma boa ideia normalizar as features? ■

Não. Árvores de Decisão não são sensíveis a normalização dos dados.

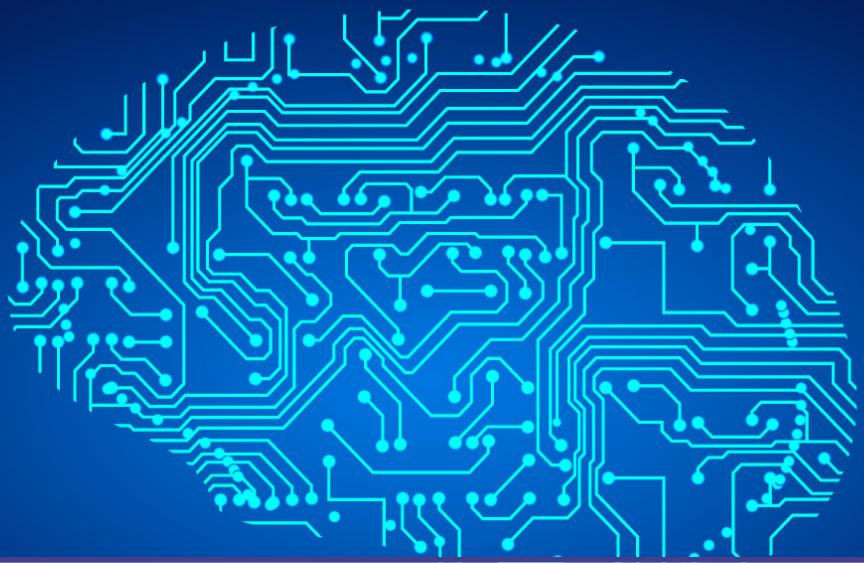
Exercício 4.5 Se um algoritmo de Árvore de Decisão leva 1 hora para ser treinado com 1 milhão de instâncias, quanto tempo levaria para treinar um outro modelo contendo 10 milhões de instâncias? ■

A complexidade computacional de um treinamento envolvendo Árvores de Decisão é dada por $(n \times 10mlog(10m))(n(m))$ e, se o conjunto de dados é multiplicado por 10 (10 milhões contra 1 milhão), temos $K = (n \times 10m\log(10m))/(n \times m\log(m))$ = $(10 \times \log(10m)/\log(m))$

Se $m = 10^6$, então temos que $K \approx 11.7$, levando assim 11.7 horas para o treinamento.

Exercício 4.6 Se o set de treino contém 100.000 instâncias, setar *presort=True* irá aumentar a velocidade do treinamento? ■

O parâmetro presort somente aumenta a velocidade do treinamento se o dataset for menor que 10.000 instâncias. Com 100.000 amostras, setar presort=True irá diminuir a velocidade do treinamento consideravelmente.



5. Ensemble Learning

Suponha que uma pergunta complexa seja feita para milhares de pessoas aleatórias com a agregação de todas essas respostas. Fatalmente, a resposta agragada de todas as pessoas valerá mais que a resposta de um único expert. A isso dá-se o nome de *Wisdom of the Crowd* (sabedoria do povo). Traduzindo essa pequena fábula ao universo de Machine Learning, se forem agregadas as **predições** de um grupo de **preditores** (mesmo que estes não sejam considerados "bons preditores"), provavelmente o resultado será **melhor** que o melhor preditor analisado de forma individual. Um grupo de preditores é chamado de *ensemble* e, a esta técnica, dá-se o nome de *Ensemble Learning*.

Por exemplo, é possível treinar um grupo de classificadores de Árvores de Decisão, cada qual em um subset aleatório dos dados de treino. Para realizar predições deste grupo, basta obter as predições de cada um dos classificadores individualmente e então predizer a classe que ganhar mais votos entre o grupo. Um conjunto de Árvores de Decisão é chamado de Florestas Aleatórias (ou *Random Forest*) e, apesar da aparente simplicidade, trata-se de um dos mais poderosos algoritmos conhecidos hoje.

Em geral, modelos Ensemble são discutidos e implementados no final de cada projeto, uma vez que a solução já foi construída e já foram testados e levantados bons modelos para a solução do problema. Neste ponto, é possível realizar combinações destes modelos com o objetivo de criar um novo modelo ainda mais poderoso para aquele problema em questão. Soluções vencedoras em competições de Machine Learning geralmente envolvem métodos Ensemble. Neste capítulo, serão tratados os métodos *bagging, boosting, stacking* e também *Random Forest*.

5.1 Voting Classifiers

Suponha que, em um problema de Machine Learning, tenham sidos testados os classificadores Regressão Logística, SVM, Random Forest KNN e talvez alguns outros, todos com alcançando uma acurácia próxima a 80%. Um jeito simples de criar um classificador ainda melhor é agregar as predições de cada classificador, retornando a classe que ganhar a maior quantidade de votos. Esta classe mais votada é chamada de *hard voting classifier*.

Surpreendentemente, esse *voting classifier* irá alcançar uma acurácia maior que o **melhor** classificador individual do conjunto. De fato, mesmo que se tenha um *weak learner*, ou seja, um

modelo com uma performance ligeiramente acima do que seria o aleatório, um conjunto formado por modelos desse tipo ainda pode se tornar um *strong learner*, alcançando bons resultados. Isso é possível desde que haja um número suficiente de weak learners no grupo e que estes sejam suficientemente diversos (se todos os modelos cometem os mesmos erros de predição, então não faz sentido).

Como isso é possível?

> **Corolário 5.1.1 — Flip a Coin; A Mágica da Moeda Viciada.** Suponha que você tenha uma moeda viciada com 51% de chance para *cara* e 49% de chance para *coroa*. Se esta moeda for jogada 1000 vezes, normalmente espera-se um resultado com 510 caras e 490 coroas ou, pelo menos, uma maioria de resultados cara. Pensando na maioria e não em quantidade, se você calcular matematicamente, verá que a probabilidade de obter maioria cara com esta moeda, após 1000 lançamentos, é de incríveis 75%. Quanto mais você joga a moeda, maior essa *"probabilidade da maioria"* (por exemplo, em 10.000 lançamentos, a probabilidade de dar maioria cara é de 97%!)

Para desvendar e provar a mágica da moeda viciada mostrada no corolário acima, tem-se um tópico conhecido como *law of large numbers*: quanto mais a moeda é lançada, mais a probabilidade de caras se assemelha ao vício da moeda (51%). No gráfico abaixo, será possível visualizar um teste para 10 séries diferentes. Em todas elas, há uma grande aproximação a linha tracejada, indicando os 51%. No limite, todas as séries terminam acima de 50%.

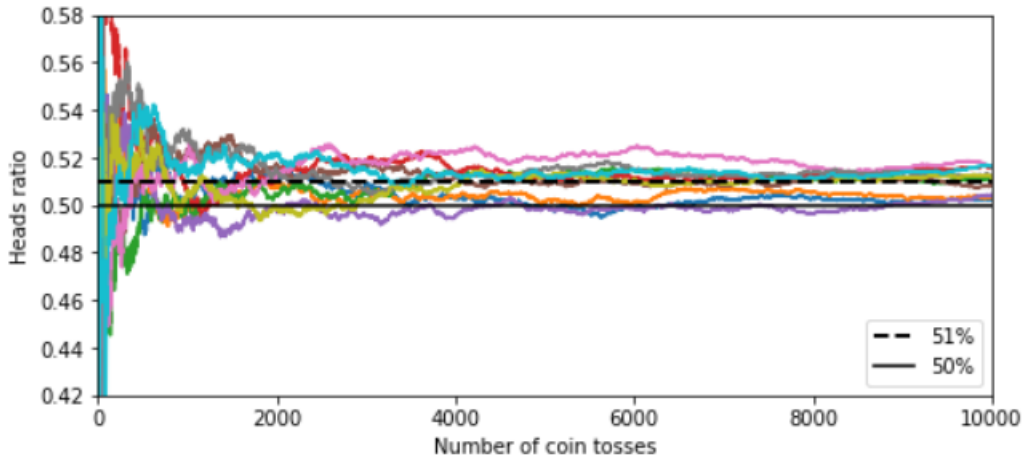


Figura 5.1: Wisdom of the Crowd - A Mágica da Moeda Viciada

Mais uma vez transpondo essa situação ao universo de Machine Learning, suponha um total de 1.000 classificadores que, individualmente, acertam em apenas 51% das oportunidades (acurácia extremamente baixa e bem próxima ao aleatório). Se as predições forem realizadas considerando o voto da maioria, espera-se que o conjunto final apresente uma acurácia em torno de 75%! Contudo, isso somente é verdade se os classificadores forem totalmente independentes, ou seja, contendo erros que não podem ser correlacionados. Isso é praticamente impossível de ser alcançado, dado que os modelos são treinados no mesmo conjunto de dados. Na prática, tem-se:

```
# Importando bibliotecas
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.linear_model import LogisticRegression
```

```
from sklearn.svm import SVC

# Preparando dados
X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

# Criando modelos
log_clf = LogisticRegression(random_state=42)
rnd_clf = RandomForestClassifier(n_estimators=10, random_state=42)
svm_clf = SVC(gamma='auto', random_state=42)

# Agrupando
voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard'
)

# Treinando modelo
voting_clf.fit(X_train, y_train)

# Visualizando resultado
for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(f'{clf.__class__.__name__}: {accuracy_score(y_test, y_pred)}')
```

Logistic Regression	Random Forest	SVC Classifier	Voting Classifier
0.864	0.872	0.888	0.896

Tabela 5.1: Acurácias Obtidas

Aqui está! O modelo agrupado, considerando apenas os votos da maioria, conseguiu uma acurácia melhor que o melhor modelo individual do conjunto. Ainda é possível melhorar a performance desse modelo agrupado.

Se todos os classificadores individuais possuírem o método *predict_proba()*, então é possível dizer ao scikit-learn para predizer a classe com maior probabilidade, medida a partir de todos os classificadores de modo individual. Isto é conhecido como *soft voting*. Normalmente, as performances são melhores se comparadas com o *hard voting* por conta do maior peso atribuído a predições mais confiáveis. No exemplo utilizado, o classificador SVM não tem o método *predict_proba()* por padrão (este deve ser configurado).

```
log_clf = LogisticRegression(random_state=42)
rnd_clf = RandomForestClassifier(n_estimators=10, random_state=42)
svm_clf = SVC(gamma='auto', random_state=42, probability=True) # hiperp
    probability=True

# Agrupando
voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='soft' # soft voting
)

# Treinando modelo
voting_clf.fit(X_train, y_train)

# Visualizando resultado
for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
    clf.fit(X_train, y_train)
```

```
    y_pred = clf.predict(X_test)
    print(f'{clf.__class__.__name__}: {accuracy_score(y_test, y_pred)}')
```

Logistic Regression	Random Forest	SVC Classifier	Voting Classifier
0.864	0.872	0.888	0.912

Tabela 5.2: Acurácias Obtidas

5.2 Bootstrap Aggregating

Como visto na sessão acima, uma forma de obter modelos diversos é treinar diferentes algoritmos de Machine Learning. Uma outra abordagem diz respeito a utilização do **mesmo algoritmo** para todos os preditores, realizando o treinamento em diferentes subsets escolhidos aleatoriamente do conjunto de treino. Quando essa coleta é realizada com reamostragem, tem-se o método *Bagging* (Bootstrap Aggregating). Quando a coleta é realizada sem reamostragem, tem-se o método *Pasting*.

Uma vez que todos os preditores são treinados, o conjunto ensemble realiza predições a partir da agregação das predições de todos os preditores. Normalmente, essa funçao de agregação é a moda para classificação (isto é, é considerada a predição mais frequente entre todas) e a média para a regressão. No geral, o resultado do conjunto ensemble possui um bias semelhante e variância mais baixa que um modelo individual treinado nos dados originais de treino.

Uma das justificativas que explicam a fama deste método entre os adoradores de Machine Learning, é a possibilidade de treinar cada preditor de forma paralela, através de cores de CPU ou até mesmo em servidores diferentes. Esse procedimento também é válido para realizar as predições.

5.2.1 Bagging na Prática

A seguir, vejamos como aplicar o Bagging em um problema de Machine Learning e comparar os resultados obtidos com um modelo individual de Árvore de Decisão.

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_moons
from sklearn.metrics import accuracy_score

# Separando dados e treinando modelo
X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1
)

bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)

# Medindo acuracia
acc_bag = accuracy_score(y_test, y_pred)
print(f'Acuracia Bagging: {acc_bag:.4f}')
```

No código acima, foi realizado o treinamento de um conjunto de 500 Árvores de Decisão (*_estimators*), cada qual treinada em 100 instâncias (*max_samples=100*) selecionadas aleatoriamente

do set de treino com reamostragem (*bootstrap=True*). A acurácia deste modelo atingiu um número entre 91 e 92%. Para validar o que foi dito até aqui, é possível treinar um modelo utilizando uma única Árvore de Decisão e comparar sua performance.

```
# Comparando com Decision Trees
tree_clf = DecisionTreeClassifier(random_state=42)
tree_clf.fit(X_train, y_train)
tree_pred = tree_clf.predict(X_test)
acc_tree = accuracy_score(y_test, tree_pred)
print(f'Acuracia Tree: {acc_tree:.4f}')
```

O modelo com uma única Árvore de Decisão alcançou uma performance próxima a 85%, ou seja, um nível inferior ao modelo de Bagging utilizado. A tabela abaixo agrega e evidencia os resultados obtidos.

Decision Trees	Bagging Classif
0.8560	0.9120

Tabela 5.3: Comparando Performance de Modelos

Para confirmar o que foi dito até o momento, serão plotadas as duas fronteiras de decisão geradas para ambos os casos acima exemplificados.

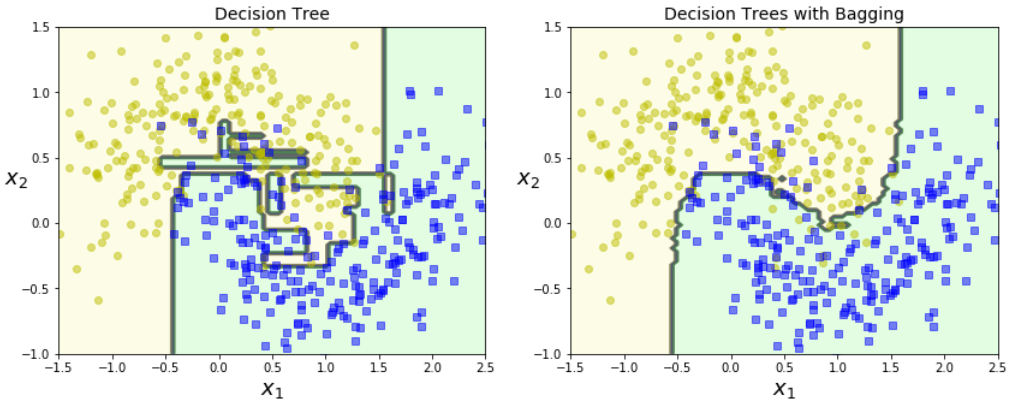


Figura 5.2: Comparação de Fronteiras de Decisão

A imagem acima mostra nitidamente que as predições do conjunto Bagging irão generalizar de forma melhor se comparadas ao modelo com uma única Árvore de Decisão: o conjunto geralmente possui bias semelhante e redução na variâncioa, tornando a fronteira de decisão menos irregular.

5.2.2 Out-of-Bag Evaluation

Em se tratando de Bagging, algumas instâncias serão amostradas repetidas vezes para qualquer um dos preditores envolvidos. Por outro lado, devido ao caráter randômico, é possível dizer que algumas dessas amostras jamais serão utilizadas pelos preditores. Por padrão, a classe BaggingClassifier amotra *m* instâncias com reamostragem *(bootstrap=True)*, onde *m* é o tamanho dos dados de treino; Isto significa que, em média, apenas 63% das instâncias de treinamento são amostradas para cada preditor. O 37% restantes não amostrados, referem-se a instâncias conhecidas como *out-of-bag* (oob). É preciso notar que estes 37% não são as mesmas instâncias para cada preditor.

Dessa forma, uma vez que o preditor não teve contando com essas instâncias oob durante o treinamento (não foram amostradas), é possível avaliar o algoritmo nessas instâncias sem a

necessidade de separar o conjunto em treino e validação, ou então aplicar validação cruzada. No Scikit-Learn, configurando o parâmetro *oob_score=True* ao criar a classe BaggingClassifier indica uma avaliação automática com as amostras oob após o treinamento. Abaixo, o código demonstrando como fazê-lo:

```
# Out-of-Bag evaluation
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    bootstrap=True, n_jobs=-1, oob_score=True
)

bag_clf.fit(X_train, y_train)
bag_clf.oob_score_
```

O resultado obtido indica que a acurácia do modelo nos dados out-of-bag é aproximadamente 89.6%. Isso indica que este modelo BaggingClassifier irá alcançar um resultado semelhante nos dados de treino, o que de fato foi comprovado após as predições.

Out-of-Bag Score	Accuracy on Test Set
0.896	0.896

Tabela 5.4: Avaliação oob e avaliação nos dados de teste

Existe também a possibilidade de visualizar a função de decisão *(decision_function())* para cada instância oob. Neste caso, como o preditor possui o método *predict_proba()*, a função *decision_function()* retorna a probabilidade de cada instância pertencer a determinada classe.

```
bag_clf.oob_decision_function_[:5]
```

executed in 6ms, finished 23:17:26 2019-09-03

```
array([[0.41573034, 0.58426966],
       [0.34782609, 0.65217391],
       [1.        , 0.        ],
       [0.        , 1.        ],
       [0.        , 1.        ]])
```

Figura 5.3: Oob decision function

5.2.3 Sampling Features

A classe BaggingClassifier também suporta a amostragem de features do modelo. Isso pode ser configurado por dois hiperparâmetros: *max_features* e *bootstrap_features*. Eles funcionam da mesma forma que os hiparparâmetros *max_samples* e *boostrap*, porém para amostragem de features ao invés de instâncias. dessa forma, o preditor será treinado em um subset aleatório das features de entrada.

Esse tipo de abordagem é útil quando o problema tratado possui alta dimensionalidade (como imagens, por exemplo). Amostrando tanto as instâncias quanto as features é um método conhecido como *Random Patches*. Mantendo todas as instâncias (isto é, *bootstrap=False* e *max_samples=1.0*) porém amostrando as features (isto é, *bootstra_features=True* e/ou *max_features* menor que 1) é chamado de *Random Subspaces*.

A amostragem de features resulta em um preditor ainda mais diverso, trocando um bias ligeiramente maior por uma variância ainda menor.

5.3 Random Forest

O algoritmo Random Forest (ou Floresta Aleatória) é um modelo ensemble de Árvores de Decisão, geralmente treinado via Bagging com *max_samples* configurado para o tamanho total do set de treino. Ao invés de construir um classificador através da classe BaggingClassifier e passar o preditor DecisionTreeClassifier, é possível utilizar diretamente a classe RandomForestclassifier, o que é mais conveniente e otimizada para Árvores de Decisão (existe também uma classe RandomForestRegressor para problemas de regressão).

Com algumas exceções, modelos Random Forest possuem todos os mesmos hiperparâmetros das Árvores de Decisão (para controlar o modo com as árvores é construída) e também todos os hiperparâmetros da classe BaggingClassifier (para controlar o conjunto ensemble por si só).

5.3.1 Extra Trees

Ao construir o algoritmo Random Forest, apenas um subset aleatório de features é considerado em cada nó de cada árvore para o splitting. É possível criar árvores ainda mais aleatórias utilizando thresholds aleatórios para cada feature ao invés de procurar pelo melhor threshold possível (como é feito normalmente pela classe Decision Trees).

Uma floresta criada com árvores extremamente aleatórias é chamada de Extra Trees. Mais uma vez, a troca realizada é mais bias por menos variância. Além disso, Extra-Trees é um modelo muito mais veloz que o Random Forest convencional, dado que a busca pelo melhor threshold possível para cada feature em cada nó pode ser extremamente oneroso. No Scikit-Learn, é possível utilizar a classe *ExtraTreesClassifier*, tendo essa as mesmas configurações da classe *RandomForestClassifier*.

5.3.2 Feature Importance

Uma outra excelente qualidade do algoritmo Random Forest é a facilidade provida para analisar a importância de cada feature para o modelo. No Scikit-Learn, essa medida é feita através da análise da redução de impureza em um nó que contém determinada feature (ao longo de todas as árvores da floresta). Mais precisamente, trata-se de uma média ponderada que considera o peso como sendo o número de instâncias associadas ao nó. No exemplo abaixo, tem-se o cálculo de *feature_importance* para o conjunto de dados iris.

```
# Importando e lendo dados
from sklearn.datasets import load_iris
iris = load_iris()

# Treinando RandomForest
rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1, random_state
    =42)
rnd_clf.fit(iris["data"], iris["target"])

# Avaliando feature importance
for name, score in zip(iris["feature_names"], rnd_clf.feature_importances_)
    :
    print(name, score)
```

Feature	Importance
sepal length (cm)	11.24%
sepal width (cm)	2.11%
petal length (cm)	44.10%
petal width (cm)	42.35%

Tabela 5.5: Importância das features

Um outro exemplo mostra uma visão da importância de cada feature dentro do dataset MNIST (dígitos manuscritos). Nesse caso, cada feature equivale a um pixel e, portanto, estamos falando da importância de cada pixel para o modelo.

```
# Importando e lendo dados
from sklearn.datasets import fetch_mldata
mnist = fetch_mldata('MNIST original')

# Treinando RandomForest
rnd_clf = RandomForestClassifier(n_estimators=10, random_state=42)
rnd_clf.fit(mnist['data'], mnist['target'])

# Definindo funcao de plotagem
def plot_digit(data):
    image = data.reshape(28, 28) # Dimensoes
    plt.imshow(image, cmap=mpl.cm.hot, interpolation='nearest')
    plt.axis('off')

# Plotando importancia das features
plot_digit(rnd_clf.feature_importances_)
cbar = plt.colorbar(ticks=[rnd_clf.feature_importances_.min(), rnd_clf.
    feature_importances_.max()])
cbar.ax.set_yticklabels(['Not important', 'Very important'])
```

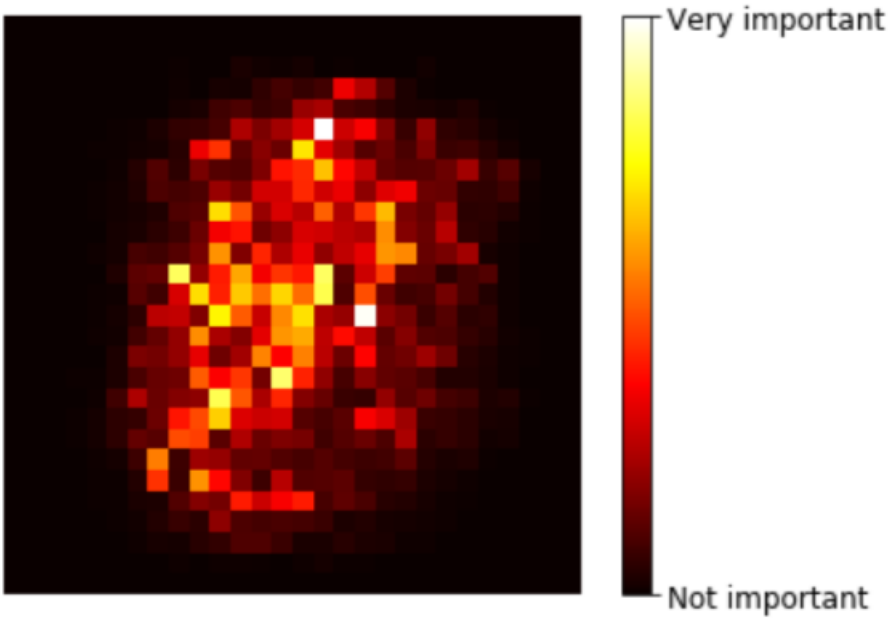


Figura 5.4: Importância das features no dataset MNIST

5.4 Boosting

O termo *Boosting* se refere a um método ensemble que combina diversos *weak learners* para formar um *strong learner*. A ideia geral é realizar treinamentos sequenciais, cada um corrigindo os erros dos antecessores. A seguir, iremos entrar em datalhes em dois dos principais algoritmos de Boosting: *AdaBoost* (Adaptative Boosting) e *Gradient Boosting*.

5.4.1 AdaBoost

Uma maneira de fazer com que o preditor corrija os erros de seu antecessor é dar mais atenção as instâncias que o antecessor obteve baixa performance. Por exemplo, na construção de um método AdaBoost, o primeiro classificador (DecisionTrees, por exemplo) é treinado e utilizado para as predições no próprio set de treinamento. O peso relativo das instâncias classificadas erroneamente são então aprimorados. Um segundo classificador é treinado utilizando os pesos atualizados dessas instâncias e, novamente, utilizado para predições no set de treinamento, gerando novos erros e novas atualizações nos pesos. E assim, cada classificador que entrar no pacote irá considerar os pesos atualizados de acordo com os erros cometidos pelos classificadores anteriores.

Uma vez que todos os preditores são treinados, o conjunto ensemble realiza as predições semelhante ao método bagging ou pasting, com exceção de que os preditores possuem pesos diferentes de acordo com a acurácia geral no set ponderado de treino.

Corolário 5.4.1 — Serialização e Paralelização. Pela própria filosofia de construção do modelo AdaBoost (e a maioria dos modelos do tipo Boosting), percebe-se que não é possível paralizar esse processamento, dado que cada preditor somente pode ser treinado após o resultado do preditor anterior. Dessa forma, Boosting não escala também quanto Bagging ou Pasting.

Considerando a matemática do problema, os pesos $w^{(i)}$ de cada instância são, inicialmente, definidos em $\frac{1}{m}$. O primeiro preditor possui uma taxa de erro r_j, computada no set de treinos através da fórmula:

Definição 5.4.1 — Taxa de Erro Ponderada para o Preditor $j^{(th)}$.

$$r_j = \frac{\sum_{\substack{i=1 \\ p_j^{(i)} \neq y^{(i)}}}^{m} w^{(i)}}{\sum_{i=1}^{m} w^{(i)}} \tag{5.1}$$

Onde $p_j^{(i)}$ é a predição do preditor j^{th} para a instância i^{th}.

Dessa forma, o peso α_j é computado de acordo com a equação abaixo, onde η é o hiperparâmetro *learning rate* (padrão=1). Quanto maior a performance do preditor, maior esse peso. Se o preditor está predizendo aleatoriamente, este peso estará próximo de 0. Se o preditor estiver pior que um palpite aleatório, o peso será negativo.

Definição 5.4.2 — Peso do Preditor.

$$\alpha_j = \eta log \frac{1 - r_j}{r_j} \tag{5.2}$$

Em seguida, os pesos das instâncias são atualizados utilizando a equação a seguir: instâncias

onde o preditor erra são "boostadas".

Definição 5.4.3 — Peso das Instâncias.

$$w^{(i)} \hookleftarrow \begin{cases} w^{(i)} & \text{if } p_j^{(i)} = y^{(i)} \\ w^{(i)} \exp(\alpha_j) & \text{if } p_j^{(i)} \neq y^{(i)} \end{cases} \tag{5.3}$$

E assim, todas os pesos são normalizados, isto é, divididos por $\sum_{i=1}^{m} w^{(i)}$.

Por fim, um novo preditor é treinado utilizando os pesos ataualizados, repetindo então todo o processo. O algoritmo para quando o número de preditores é atingido. Para as predições, o algoritmo AdaBoost computa as predições de todos os preditores e aplica os pesos aos mesmos utilizando os valores calculados de α_j. A classe predita é simplesmente aquela com a maioria dos votos (considerando a ponderação).

Definição 5.4.4 — Predições no AdaBoost.

$$p(x) = argmax_k \sum_{\substack{j=1 \\ p_j(x)=k}}^{N} \alpha_j \tag{5.4}$$

Onde N é o número de preditores.

Para classificação multi-classe, o Scikit-learn oferece uma versão do AdaBoost chamada SAMME *(Stagewise Additive Modeling using a Multiclass Exonential Loss Function*. Quando os preditores podem estimar as probabilidades (isto é, possuem o método *predict_proba*), o Scikit-Learn utiliza uma variante do SAMME chamada SAMME.R, que retorna as probabilidades da classe ao invés das predições propriamente dita, normalmente performando melhor que o SAMME. Abaixo, um exemplo:

```
# Importando ensemble
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=200,
    algorithm='SAMME.R', learning_rate=0.5
)
ada_clf.fit(X_train, y_train)
```

5.4.2 Gradient Boosting

Assim como foi visto no algoritmo AdaBoost, o Gradient Boosting também trabalha sequencialmente na adição de preditores em um conjunto, cada qual corrigindo o antecessor. Entretanto, ao invés de modificar os pesos de instâncias classificadas erroneamente (como faz o AdaBoost), o Gradient Boosting treina o próximo modelo utilizando os *erros residuais* do antecessor.

Para exemplificar a filosofia do Gradient Boosting, será utilizado o algoritmo Decision Trees em um problema de regressão de forma sequencial, considerando dados sintéticos com um pouco de ruído. Nesse tipo de problema, a nomenclatura correta para o modelo é *Gradient Tree Boosting* ou *Gradient Boosted Regression Trees (GBRT)*

```
from sklearn.trees import DecisionTreeRegressor

tree_reg1 = DecisionTreeRegressor(max_depth=2)
tree_reg1.fit(X, y)
```

Agora, um segundo modelo DecisionTreeRegressor é aplicado nos erros residuais gerados pelo primeiro modelo:

```
# Calculando erros
y2 = y - tree_reg1.predict(X)

tree_reg2 = DecisionTreeRegressor(max_depth=2)
tree_reg2.fit(X, y2)
```

Mais uma vez, um terceiro modelo será aplicado considerando os erros residuais gerados pelo anterior:

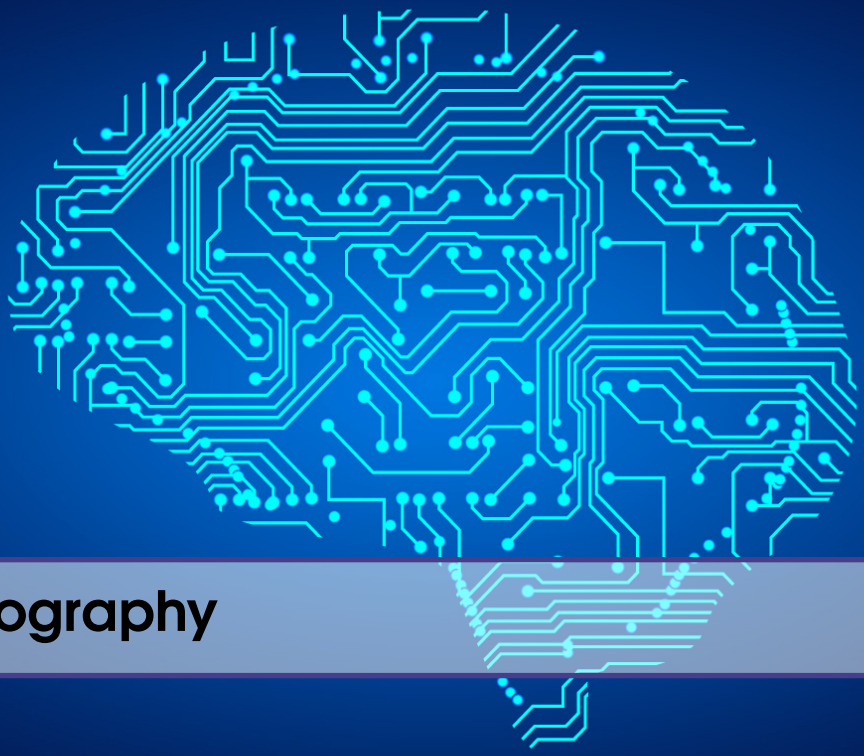
```
# Calculando erros
y3 = y2 - tree_reg1.predict(X)

tree_reg3 = DecisionTreeRegressor(max_depth=2)
tree_reg3.fit(X, y3)
```

Nesse momento, tem-se um ensemble contendo três Árvores de Decisão. As predições de uma nova instância podem ser realizadas a partir da soma das predições de cada um dos modelos.

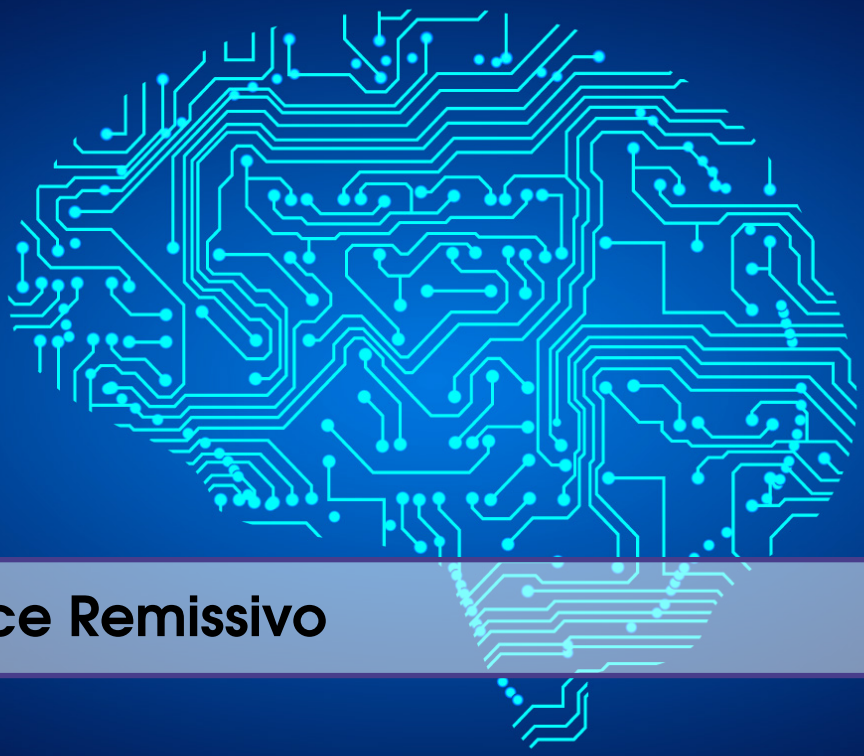
```
# Predizendo novos dados
y_pred = sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2,
    tree_reg3))
```

A figura a seguir ilustra o que de fato ocorreu nos três passos de treinamentos exemplificados acima. Na primeira linha de gráficos, tem-se apenas um único modelo de árvore e, portanto, as predições do ensemble serão exatamente as mesmas desse único modelo. A partir da segunda linha, são calculados os erros residuais do modelo anterior e, dessa forma, *Colocensemble serão rigorar figura.

Parte Três



Bibliography



Índice Remissivo

O

P

R

T

V

5.5 Curvas de Aprendizado

Até agora, tópicos sobre *overfitting* e *underfitting* foram citados sem necessariamente explicar, de fato, como identificar tais fenômenos e, o mais importante, o que fazer para evitar que eles se façam presentes. Um método extremamente eficaz para identificar se um modelo sofre de *overfitting* (variância alta) ou *underfitting* (bias alto) é através das curvas de aprendizado.

Basicamente, a curva de aprendizado plota um gráfico com o score (ou o erro) no eixo y avaliados nos dados de treino e de teste. No eixo x, tem-se as épocas de treinamento. Entende-se por época, pequenos pacotes gradativos de amostras do conjunto de dados.

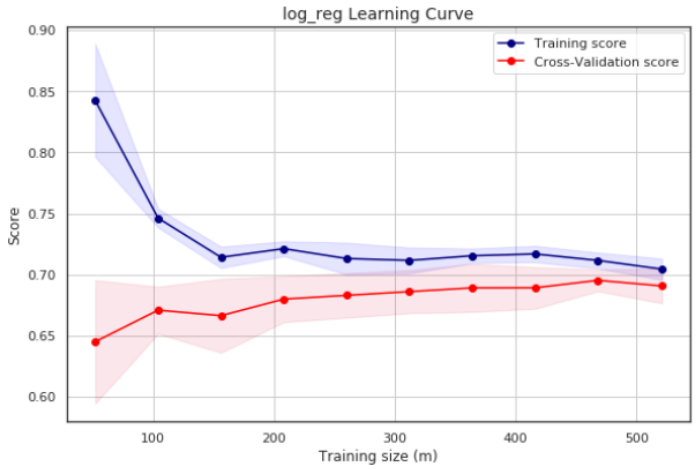


Figura 5.5: Exemplo de Curva de Aprendizado: Regressão Logística

Na figura 2.4, é apresentada a curva de aprendizado para um modelo treinado com o algorítmo de Regressão Logística. Nela, o score para os dados de treino e de validação estão em um patamar semelhante, indicando que o modelo não sofre de overfitting. Entretanto, verifica-se que os scores dos dois conjuntos de dados representam um baixo score, provavelmente gerado por um alto *bias* do modelo.

O que pode ser feito para melhorar a performance de modelos que possivelmente sofrem de overfitting:

- Aumentar o grau polinomial do modelo
- Considerar mais features para o modelo
- Diminuir o parâmetro de regularização

Já na figura 2.5, a curva de aprendizado de um modelo de Árvores de Decisão é mostrada. Nela, é possível perceber uma grande diferença entre os scores medidos com os dados de treino e de validação, indicando assim que o modelo performa extremamente bem nos dados de treino, porém de forma ineficaz nos dados de validação. Este é um problema típico de overfitting, visto que o modelo não generaliza bem para dados não utilizados no treinamento.

A alta *variância* do modelo treinado pode ser resolvida com as seguintes tentativas:

- Coletar mais dados de treinamento
- Diminuir o set de features
- Aumentar o termo de regularização

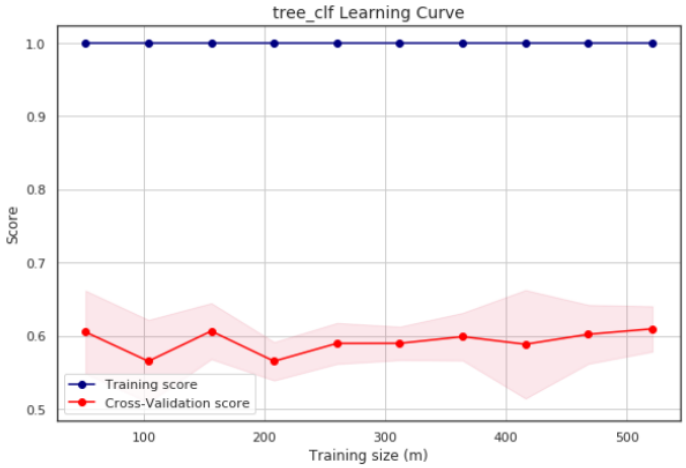


Figura 5.6: Exemplo de Curva de Aprendizado: Árvore de Decisão