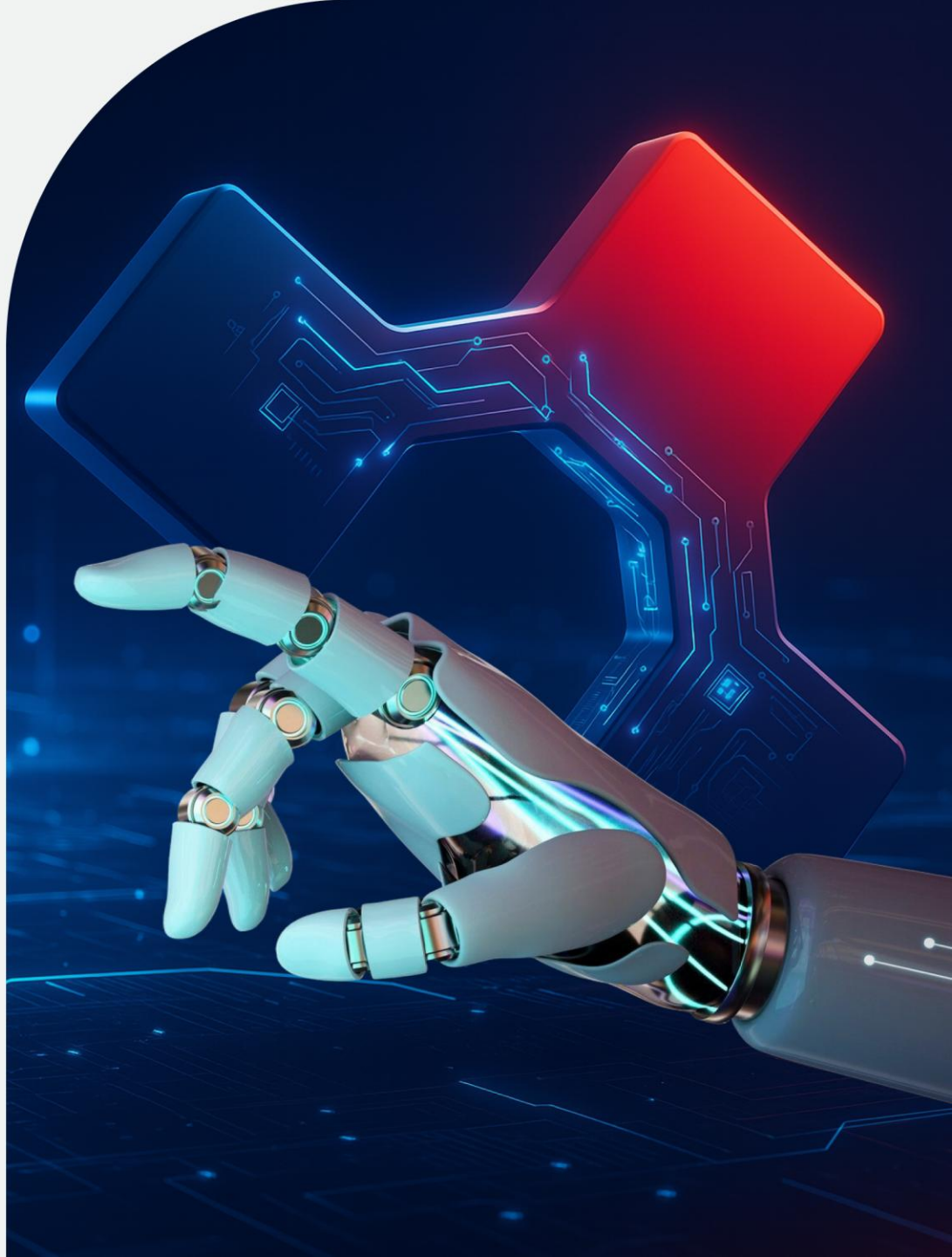
NCIA

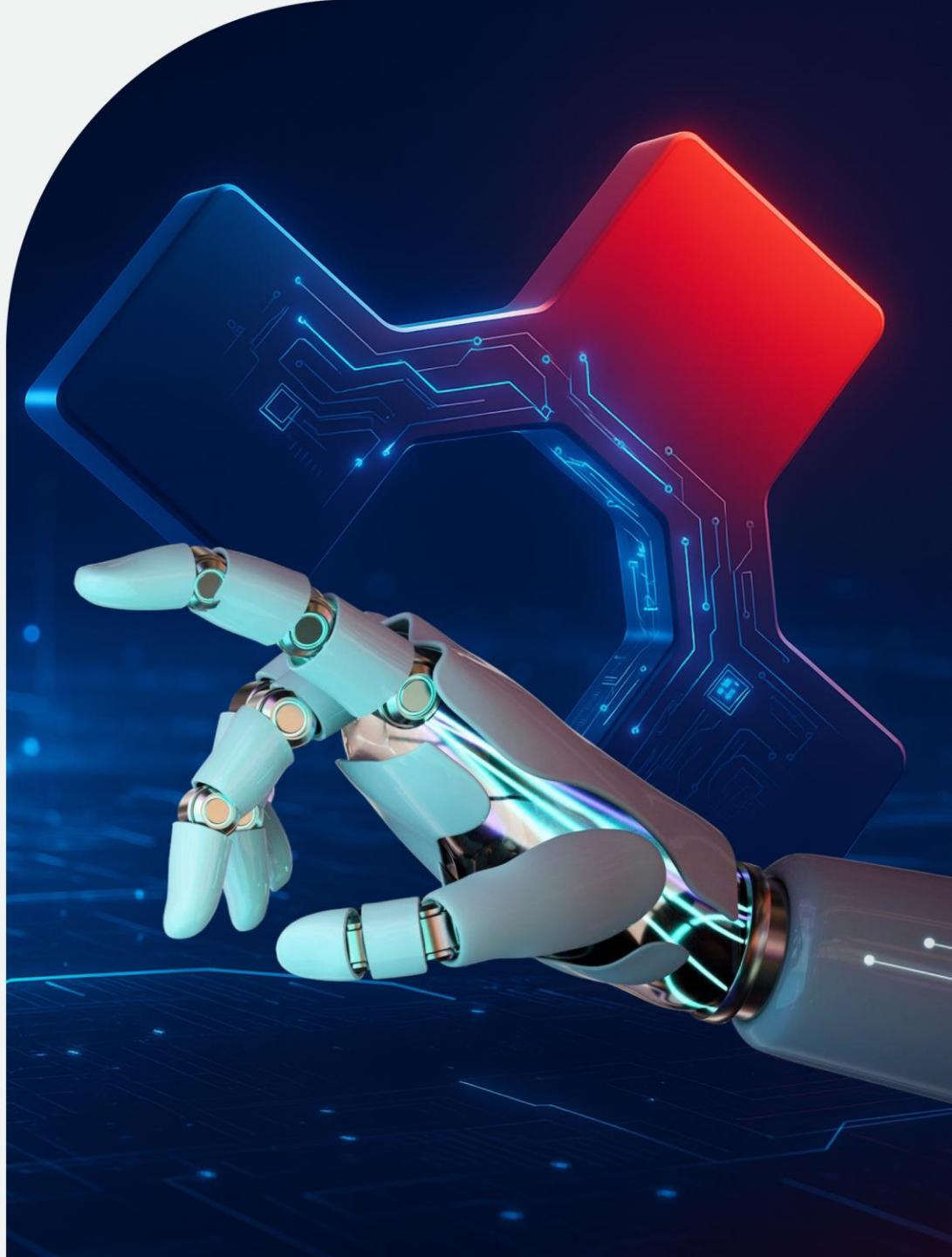
Núcleo de Capacitação em Inteligência Artificial



NCIA | FOXCONN FPF tech

Ensemble Learning and Random Forest

Voting Classifiers; Bagging and Pasting; Bagging and Pasting in Scikit-Learn; Out-of-Bag Evaluation; Random Patches and Random Subspaces; Random Forests; Extra-Trees; Feature Importance; Boosting; AdaBoost; Gradient Boosting; Histogram-Based Gradient Boosting; Stacking



NCIA | FOXCONN FPF tech

Ensemble Learning

Quando pedimos a opinião de muitas pessoas sobre uma questão complexa e depois juntamos todas as respostas, muitas vezes a média dessas opiniões é melhor do que a resposta de um especialista.

Esse fenômeno é chamado de *sabedoria coletiva*.

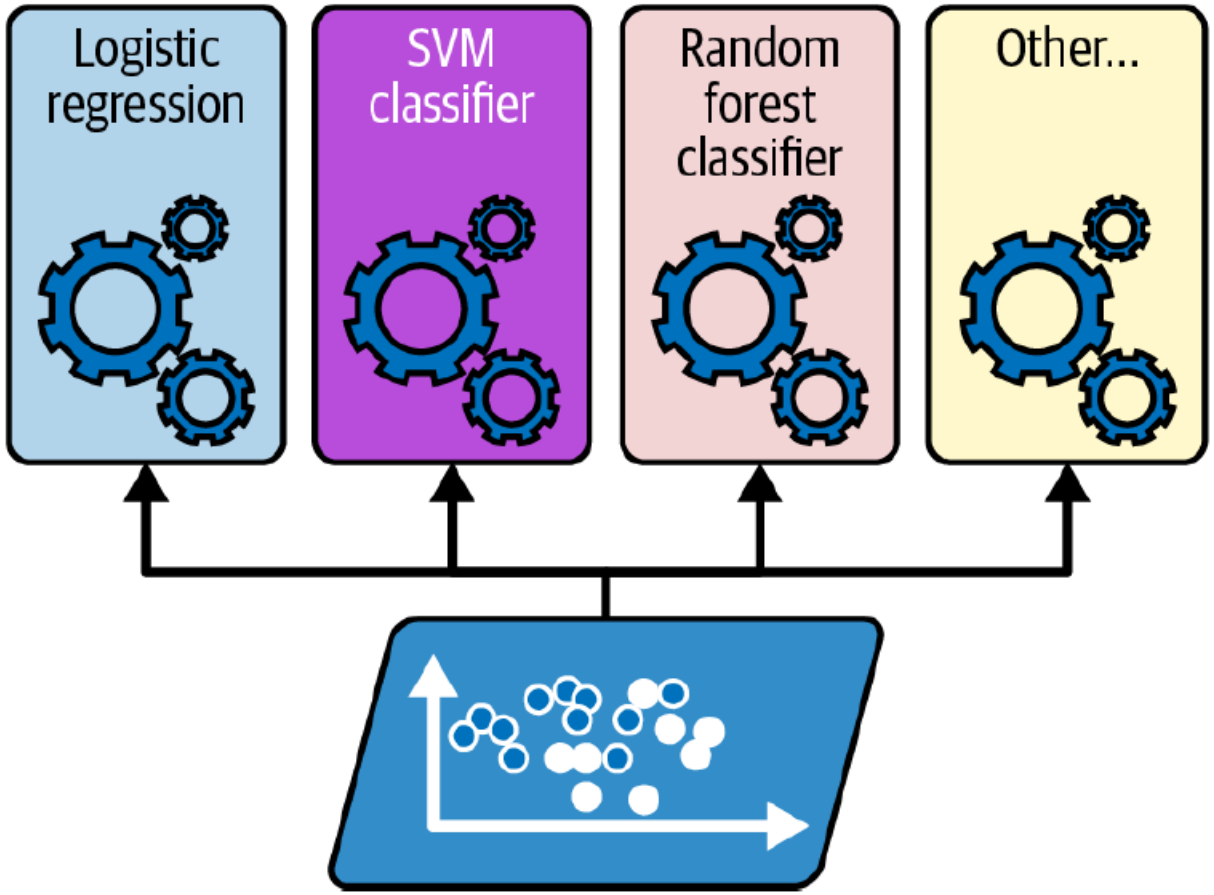
Em Machine Learning acontece o mesmo: ao invés de depender de um único modelo, podemos combinar vários modelos e obter resultados melhores. A esse conjunto de modelos damos o nome de *ensemble*, e o método que faz a combinação é chamado de *ensemble method*.

Um exemplo muito conhecido é o *Random Forest*, que combina várias árvores de decisão treinadas em subconjuntos diferentes dos dados para chegar a uma previsão final.

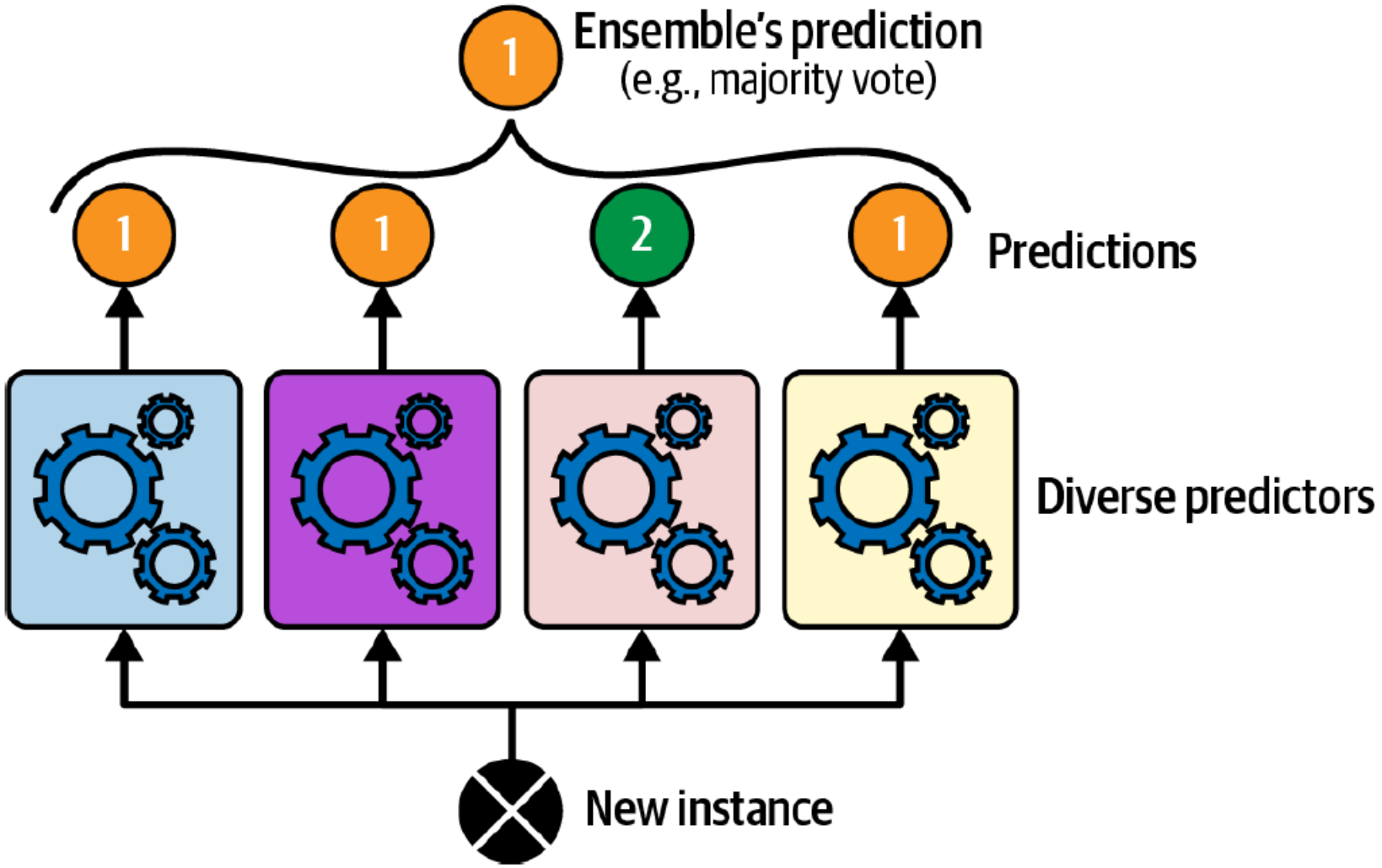
1. Classificadores de Votação (Hard Voting)

Podemos treinar vários modelos diferentes — por exemplo, uma Regressão Logística, um SVM e um Random Forest e usar todos eles juntos.

Cada modelo faz sua predição para a nova instância, e então escolhemos a classe que recebeu a maioria dos votos. Esse processo é chamado de hard voting classifier. A ideia é simples, mas muitas vezes o resultado final é melhor do que qualquer modelo individual isolado.



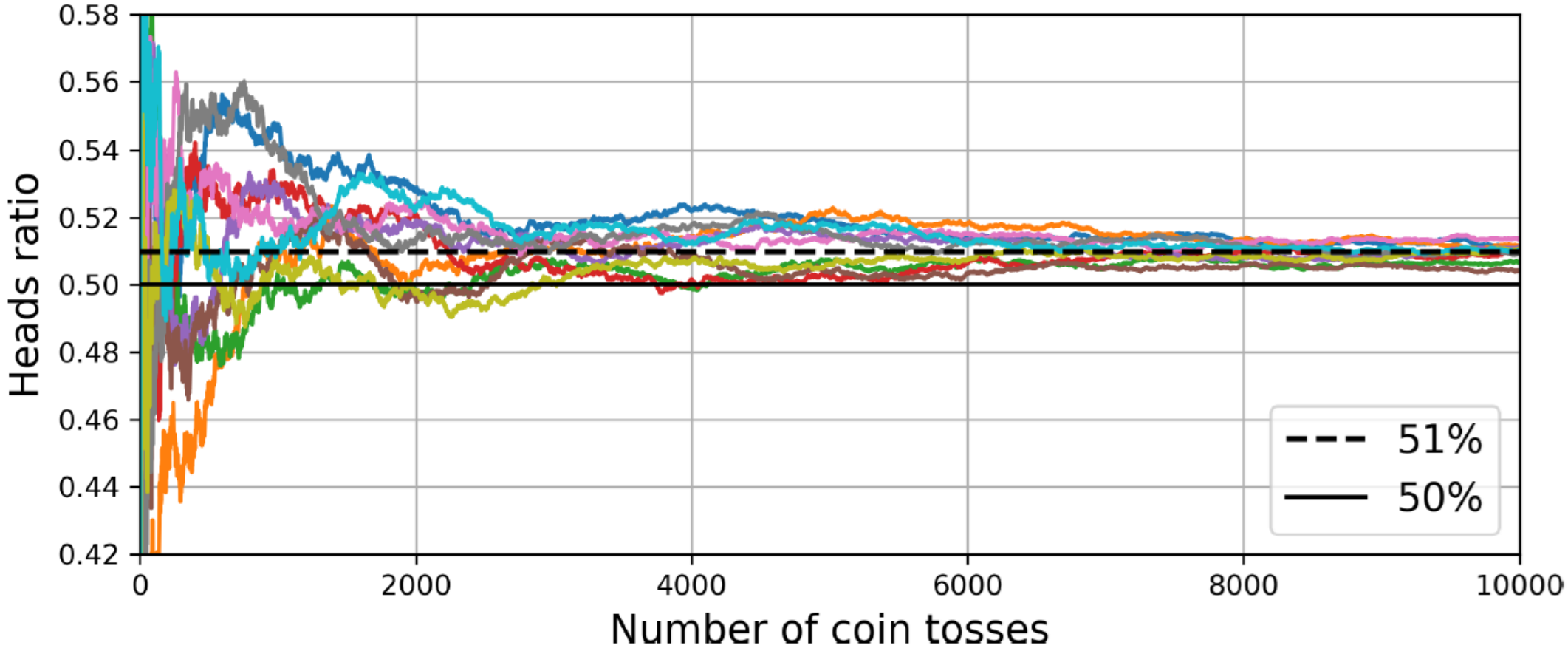
Diverse predictors


1
Ensemble's prediction
(e.g., majority vote)
1
1
2
1
Predictions
Diverse predictors
New instance

Mesmo quando usamos modelos considerados fracos, que acertam pouco mais do que o acaso (por exemplo, 51% de acerto), ao juntarmos muitos deles podemos criar um modelo forte.

Isso acontece por causa da Lei dos Grandes Números. Ela afirma que, quanto mais vezes repetimos um experimento, mais a média dos resultados se aproxima da probabilidade real.

Um exemplo simples: se jogarmos uma moeda viciada que tem 51% de chance de dar cara milhares de vezes, a proporção de caras vai se aproximando de 51%. E quanto mais jogadas, maior a chance de termos maioria de caras.


0.58
0.56
0.54
0.52
0.50
0.48
0.46
0.44
0.42
Heads ratio
0
2000
4000
6000
8000
10000
Number of coin tosses
51%
50%

Mesmo quando usamos modelos considerados fracos, que acertam pouco mais do que o acaso (por exemplo, 51% de acerto), ao juntarmos muitos deles podemos criar um modelo forte.

Isso acontece por causa da Lei dos Grandes Números. Ela afirma que, quanto mais vezes repetimos um experimento, mais a média dos resultados se aproxima da probabilidade real.

Um exemplo simples: se jogarmos uma moeda viciada que tem 51% de chance de dar cara milhares de vezes, a proporção de caras vai se aproximando de 51%. E quanto mais jogadas, maior a chance de termos maioria de caras.

Após importarmos os conjuntos de dados, os classificadores que comporão o ensemble, a classe VotingClassifier, o utilitário de divisão treino/teste e o SVM. Usaremos random_state=42 para reprodutibilidade.

```
from sklearn.datasets import make_moons
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
```

Criamos o dataset “make moons” com ruído moderado para ficar não linear e, em seguida, separamos em treino e teste.

```
# Gerando o dataset
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)
```

Definimos um ensemble com três modelos diversos (Logistic Regression, Random Forest e SVC).
Observação importante: ao chamar fit, o VotingClassifier clona cada estimador e treina os clones (acessíveis em estimators_ ou named_estimators_).

```
# Criando o VotingClassifier (hard voting por padrão)
voting_clf = VotingClassifier(
    estimators=[
        ('lr', LogisticRegression(random_state=42)),
        ('rf', RandomForestClassifier(random_state=42)),
        ('svc', SVC(random_state=42))
    ]
)
voting_clf.fit(X_train, y_train)
```

A seguir, medimos a acurácia de cada modelo separadamente no conjunto de teste. Isso nos dá a linha de base para comparar com o ensemble.

```
# Avaliando a acurácia de cada estimador individual
for name, clf in voting_clf.named_estimators_.items():
    print(name, "=", clf.score(X_test, y_test))
```

Output: Lr ≈ 0.864, rf ≈ 0.896, svc ≈ 0.896

Com *hard voting*, o ensemble retorna a *classe mais votada* (sem olhar as probabilidades). Veja o primeiro exemplo do teste: dois modelos votam na classe 1, um na classe 0 → o ensemble prevê 1.

```
# Testando o hard voting na primeira instância
print("VotingClassifier prediction (hard voting):", voting_clf.predict(X_test[:1]))
print("Individual predictions:", [clf.predict(X_test[:1]) for clf in voting_clf.estimators_])
```

Output:
array([1])
[array([1]), array([1]), array([0])]

Agora avaliamos o ensemble completo no conjunto de teste. A expectativa é superar cada modelo individual.

```
# Acurácia do ensemble com hard voting
print("VotingClassifier (hard voting) score:", voting_clf.score(X_test, y_test))
```

Output: 0.912

No *soft voting*, o ensemble escolhe a classe com *maior probabilidade média* entre os modelos (o que tende a funcionar melhor).
Para isso, *todos* os estimadores devem expor predict_proba(). O SVC *não* faz isso por padrão; precisamos ativar probability=True (ele usará validação cruzada internamente para estimar probabilidades, deixando o treino mais lento). Em seguida, refazemos o fit e medimos novamente a acurácia.

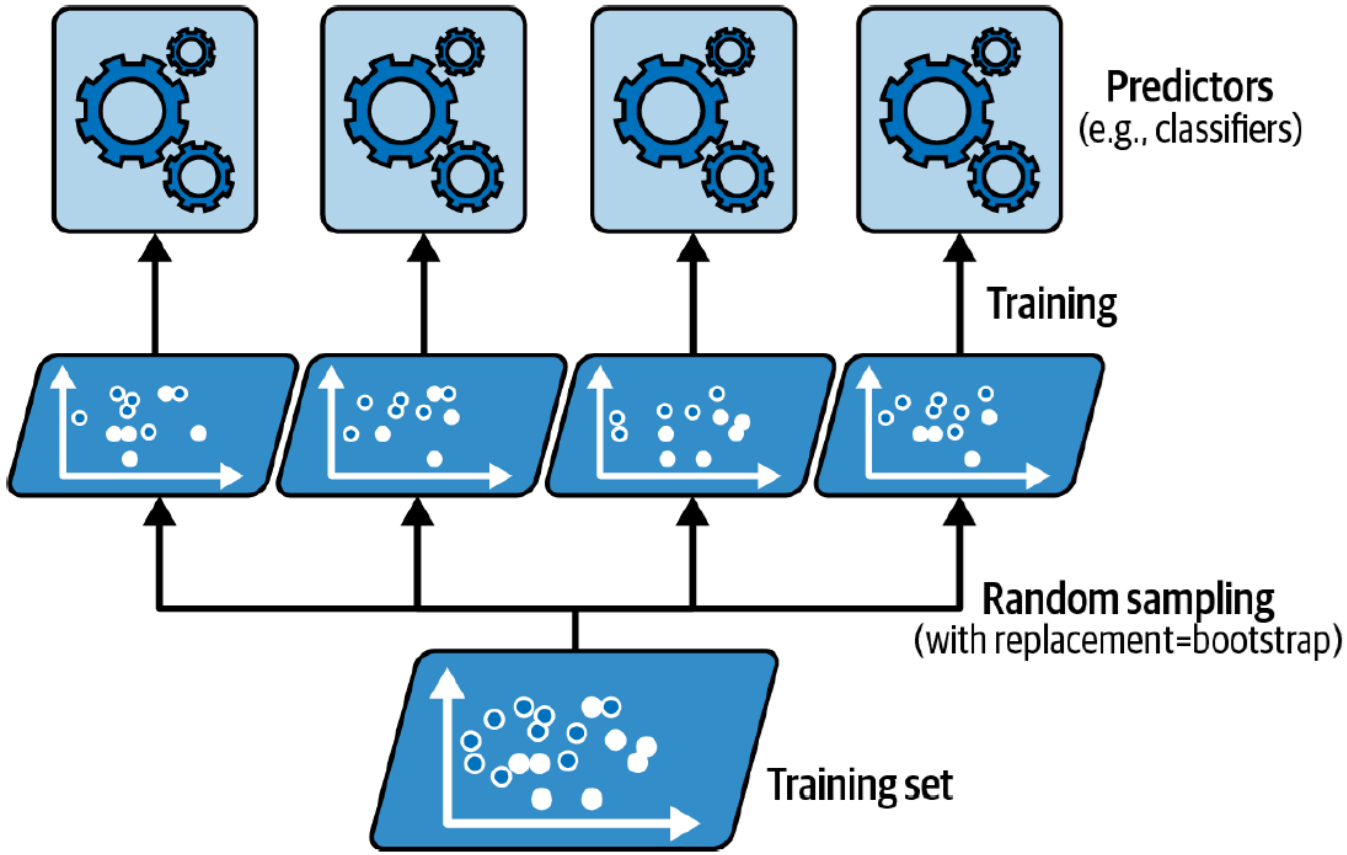
```
# Alterando para soft voting (necessário ativar probabilidade no SVC)
voting_clf.voting = "soft"
voting_clf.named_estimators["svc"].probability = True
voting_clf.fit(X_train, y_train)

# Acurácia do ensemble com soft voting
print("VotingClassifier (soft voting) score:", voting_clf.score(X_test, y_test))
```

Output: 0.92
— pequeno ganho sobre o hard voting

2. Bagging e Pasting

Outra forma de criar ensembles é usar o mesmo algoritmo várias vezes, mas treinando em subconjuntos diferentes do conjunto de treino.

Quando o subconjunto é gerado com reposição (uma mesma instância pode aparecer mais de uma vez no subconjunto), chamamos de Bagging (*Bootstrap Aggregating*).
Quando é sem reposição (cada instância é usada apenas uma vez no subconjunto escolhido), chamamos de Pasting.
Nos dois casos, cada preditor vê apenas parte dos dados, o que gera mais diversidade.


Predictors
(e.g., classifiers)
Training
Random sampling
(with replacement=bootstrap)
Training set

Depois que todos os preditores são treinados:

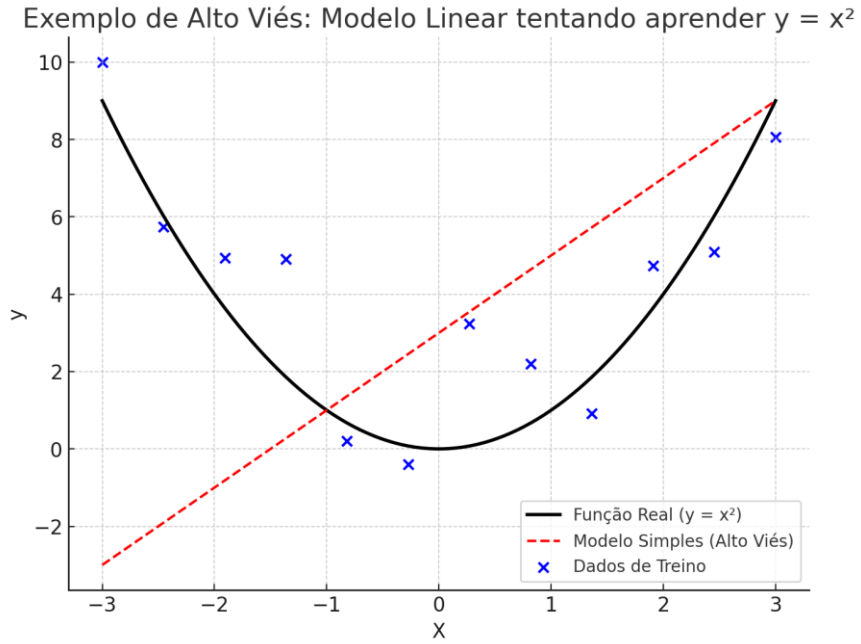
Para classificação: o ensemble usa a moda (classe mais votada).
Para regressão: o ensemble usa a média das previsões.

Cada modelo individual tende a ter maior viés, pois foi treinado em menos dados.
Mas, ao combinar todos, o ensemble consegue reduzir a variância e manter o viés parecido, o resultado é um modelo mais estável e robusto.
.

2.1 Sobre Viés e Variância

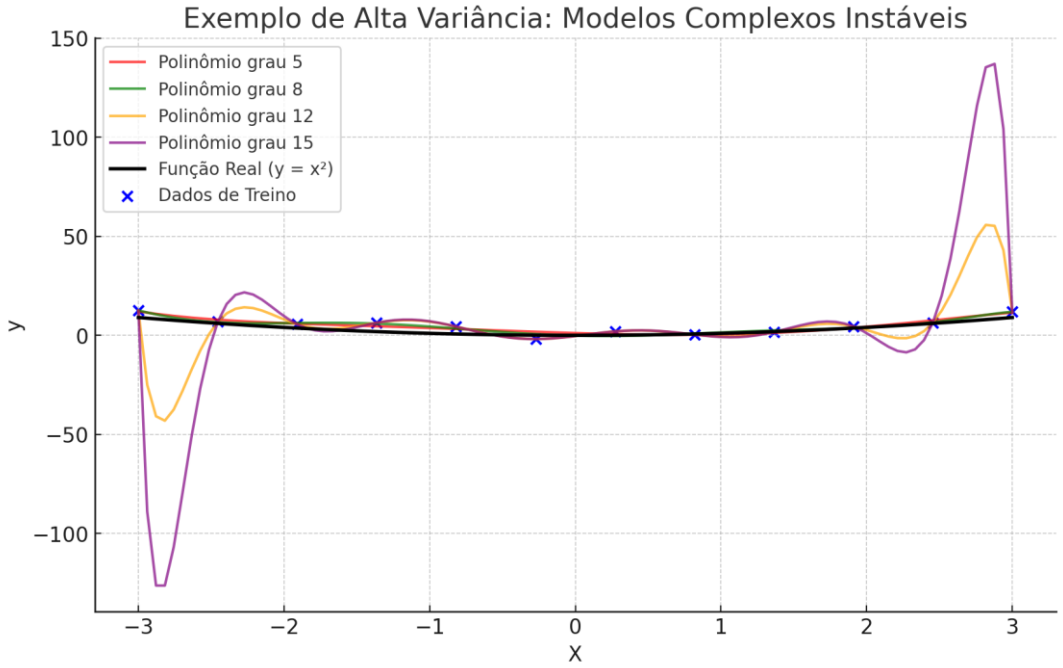
O viés representa o erro sistemático de um modelo.
Mede a diferença entre a previsão média e o valor real.
Modelos simples tendem a ter viés alto, pois não conseguem capturar a complexidade do problema.
Exemplo: usar regressão linear para prever algo parabólico

.



A variância representa a sensibilidade do modelo ao conjunto de dados de treino.
Mede o quanto o modelo muda quando os dados mudam um pouco.
Modelos muito complexos têm alta variância: se ajustam demais ao treino e não generalizam.
Exemplo: árvore de decisão profunda que memoriza o treino.

.

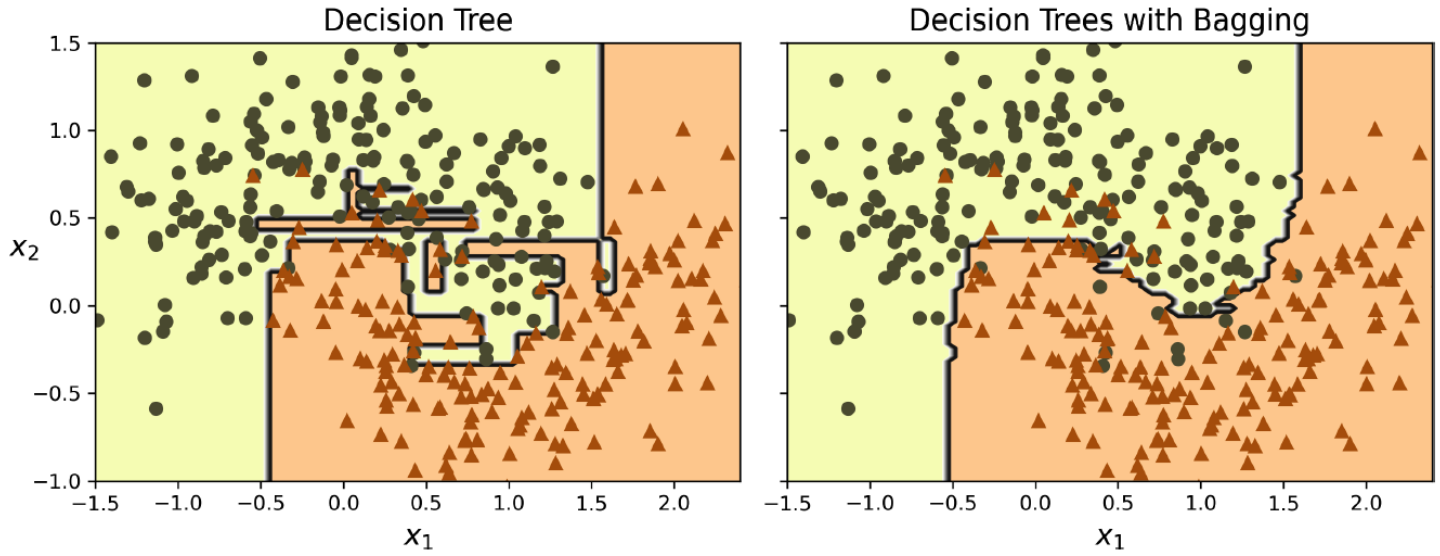


Sempre desejamos um equilíbrio entre viés e variância:

Modelos simples: alto viés, baixa variância.
Modelos complexos: baixo viés, alta variância.
O objetivo é encontrar um ponto intermediário que permita boa generalização.

2.2 Bagging no Scikit-Learn

O Scikit-Learn fornece a classe BaggingClassifier para ensembles de classificadores, e BaggingRegressor para regressão.
Exemplo: 500 árvores de decisão, cada uma treinada em 100 instâncias escolhidas aleatoriamente.
bootstrap=True → Bagging (com reposição).
bootstrap=False → Pasting (sem reposição).
n_jobs=-1 → uso de todos os núcleos da CPU.



Ambos criam subconjuntos de treino aleatórios.
Bagging: Mais diversidade porque permite repetição de instâncias → menor variância, mas viés ligeiramente maior.

Pasting → usa instâncias sem repetição.

Na prática, Bagging tende a funcionar melhor, mas é possível usar validação cruzada para decidir.

2.3 Out-of-Bag (OOB) Evaluation

Em média, 37% dos exemplos não entram no treino de um dado estimador (instâncias OOB). Podemos usar essas instâncias para avaliar o ensemble sem separar um conjunto de validação.

No Scikit-Learn: oob_score=True.
O resultado fica em oob_score_.
Também é possível obter as probabilidades previstas para cada instância em oob_decision_function_.

```python
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.metrics import accuracy_score
import numpy as np

# 1) Dataset e split
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)
# 2) Baseline: árvore única
tree_clf = DecisionTreeClassifier(random_state=42)
tree_clf.fit(X_train, y_train)
acc_tree = tree_clf.score(X_test, y_test)
print(f"Acurácia - Árvore única: {acc_tree:.3f}")
```

Output: Acurácia - Árvore única: 0.856

Bagging (com reposição): 500 árvores; cada uma treina com 100 amostras.
Conceitos: bootstrap=True por padrão; n_jobs=-1 usa todos os núcleos.

```
# 3) Bagging (com reposição) - 500 árvores, 100 amostras por
estimador
bag_clf = BaggingClassifier(
    base_estimator=DecisionTreeClassifier(),
    n_estimators=500,
    max_samples=100,
    n_jobs=-1,
    random_state=42
)
bag_clf.fit(X_train, y_train)
acc_bag = bag_clf.score(X_test, y_test)
print(f"Acurácia - Bagging (500 árvores): {acc_bag:.3f}")
```

Output: Acurácia - Bagging (500 árvores): 0.904

Pasting (sem reposição): troca para bootstrap=False.
Conceitos: pasting pode ter viés um pouco menor/maior; avalie com Cross Validation.

```
# 4) Pasting (sem reposição)
pasting_clf = BaggingClassifier(
    base_estimator=DecisionTreeClassifier(),
    n_estimators=500,
    max_samples=100,
    bootstrap=False,   # sem reposição
    n_jobs=-1,
    random_state=42
)
pasting_clf.fit(X_train, y_train)
acc_pasting = pasting_clf.score(X_test, y_test)
print(f"Acurácia - Pasting (500 árvores): {acc_pasting:.3f}")
```

Output: Acurácia - Pasting (500 árvores): 0.920

OOB evaluation: mede desempenho usando amostras não vistas por cada estimador.
Conceitos: ~63% das amostras entram por estimador; ~37% ficam OOB.

```
# 5) Bagging com avaliação OOB
bag_oob_clf = BaggingClassifier(
    base_estimator=DecisionTreeClassifier(),
    n_estimators=500,
    oob_score=True,    # ativa OOB
    n_jobs=-1,
    random_state=42
)
bag_oob_clf.fit(X_train, y_train)
print(f"OOB score (estimativa): {bag_oob_clf.oob_score_:.3f}")
```

Output: OOB score (estimativa): 0.896

```python
# 6) Verificação no teste
y_pred = bag_oob_clf.predict(X_test)
acc_test = accuracy_score(y_test, y_pred)
print(f"Acurácia no teste (Bagging com OOB): {acc_test:.3f}")

# 7) Probabilidades OOB para as 3 primeiras instâncias de treino
np.set_printoptions(precision=4, suppress=True)
print("OOB decision function (primeiras 3 instâncias):")
print(bag_oob_clf.oob_decision_function_[:3])
```

Output:

```
Acurácia no teste (Bagging com OOB): 0.920
OOB decision function (primeiras 3 instâncias):
[[0.3235 0.6765]
 [0.3375 0.6625]
 [1.     0.    ]]
```

```python
# 6) Verificação no teste
y_pred = bag_oob_clf.predict(X_test)
acc_test = accuracy_score(y_test, y_pred)
print(f"Acurácia no teste (Bagging com OOB): {acc_test:.3f}")

# 7) Probabilidades OOB para as 3 primeiras instâncias de treino
np.set_printoptions(precision=4, suppress=True)
print("OOB decision function (primeiras 3 instâncias):")
print(bag_oob_clf.oob_decision_function_[:3])
```

Output:

```
Acurácia no teste (Bagging com OOB): 0.920
OOB decision function (primeiras 3 instâncias):
[[0.3235 0.6765]
 [0.3375 0.6625]
 [1.     0.    ]]
```

2.4 Amostragem de Features: Random Patches & Subspaces

Além de amostrar instâncias, podemos amostrar features para treinar cada modelo do ensemble. Definimos quantas features cada estimador verá (max_features) e se essa amostragem é com reposição (bootstrap_features).

Chamamos de random patches quando amostramos instâncias e features; e de random subspaces quando mantemos todas as instâncias, mas amostramos apenas features. Essa estratégia aumenta a diversidade entre os modelos, o que reduz a variância do ensemble, com um pequeno aumento de viés — e costuma acelerar o treino em dados de alta dimensionalidade.

Random Forest

3. Random Forests: ideia e vantagens

Uma Random Forest é um conjunto de árvores de decisão treinadas com amostragem aleatória. Em cada nó, a árvore escolhe o melhor split dentro de um subconjunto aleatório de features, o que gera árvores diferentes entre si. Essa aleatoriedade extra normalmente mantém o viés próximo ao de uma árvore bem configurada, mas reduz a variância do modelo final, melhorando a generalização.

Na prática usamos RandomForestClassifier ou RandomForestRegressor, que são versões otimizadas do bagging com árvores. É comum treinar centenas de árvores, limitar a profundidade ou o número de folhas, e usar todas as CPUs disponíveis. Um ensemble de árvores com amostragem de features (por exemplo, sqrt(n_features) em classificação) é aproximadamente equivalente a um BaggingClassifier configurado com DecisionTreeClassifier(max_features="sqrt").

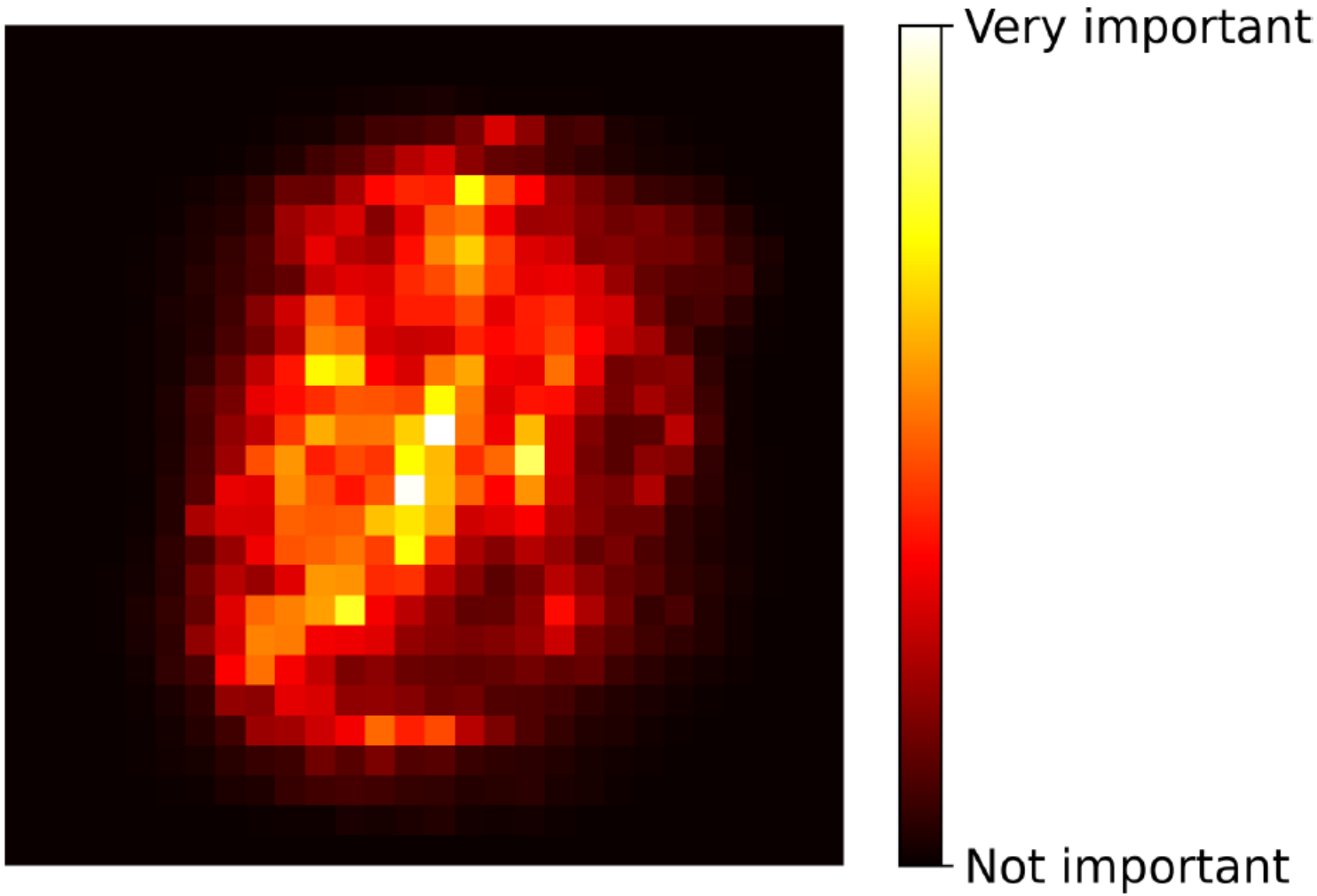
3.1 Extra-Trees (Extremely Randomized Trees)

As Extra-Trees tornam o processo ainda mais aleatório ao escolher limiares aleatórios para cada feature, sem buscar o melhor limiar. Isso deixa o treino mais rápido e o ensemble menos correlacionado, reduzindo variância. O custo é um leve aumento de viés. Em muitos cenários, Extra-Trees e Random Forest apresentam desempenho muito próximo, valendo testar ambos.

3.2 Importância de Features em Florestas

Florestas fornecem, após o treino, a importância relativa de cada feature, calculada pela redução média de impureza ponderada. Esse diagnóstico ajuda a entender o que o modelo usa para decidir e pode guiar seleção de atributos.

No conjunto Iris, as medidas da pétala são as mais importantes; no MNIST, o mapa de importância destaca as regiões dos dígitos.


Very important
Not important

Random Forest (500 árvores, 16 folhas máx.)

Treinamos uma floresta com 500 árvores; limitamos o nº de folhas para controlar complexidade; usamos todas as CPUs.

```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(
    n_estimators=500,
    max_leaf_nodes=16,
    n_jobs=-1,
    random_state=42
)
rnd_clf.fit(X_train, y_train)
y_pred_rf = rnd_clf.predict(X_test)
```

Random Forest ≈ Bagging + Árvore com amostragem de features

Um BaggingClassifier com árvore base usando max_features="sqrt" (e o mesmo limite de folhas) é aproximadamente equivalente à floresta acima.

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(max_features="sqrt", max_leaf_nodes=16),
    n_estimators=500,
    n_jobs=-1,
    random_state=42
)
bag_clf.fit(X_train, y_train)
```

Random Patches & Random Subspaces (exemplos)

Random patches: Amostra instâncias e features.
Random subspaces: Mantém todas as instâncias, amostra apenas features.

```
# Random patches: instâncias + features
patches_clf = BaggingClassifier(
    DecisionTreeClassifier(),
    n_estimators=300,
    max_samples=0.6,            # 60% das instâncias
    bootstrap=True,             # com reposição (bagging)
    max_features=0.5,           # 50% das features
    bootstrap_features=True,    # com reposição nas features
    n_jobs=-1,
    random_state=42
).fit(X_train, y_train)
```

```python
# Random subspaces: todas as instâncias, amostra só features
subspaces_clf = BaggingClassifier(
    DecisionTreeClassifier(),
    n_estimators=300,
    max_samples=1.0,          # 100% das instâncias
    bootstrap=False,          # sem reposição nas instâncias
    max_features=0.5,         # 50% das features
    bootstrap_features=True,  # com reposição nas features
    n_jobs=-1,
    random_state=42
).fit(X_train, y_train)
```

Extra-Trees (extremely randomized trees)

Árvores ainda mais aleatórias: limiares de split aleatórios; treino rápido; menor variância.

```python
from sklearn.ensemble import ExtraTreesClassifier

extra_clf = ExtraTreesClassifier(
    n_estimators=500,
    max_leaf_nodes=16,
    n_jobs=-1,
    random_state=42
)
extra_clf.fit(X_train, y_train)
```

Importância de Features (Iris)

Após treinar, avaliamos feature_importances_ para entender o que o modelo valoriza.

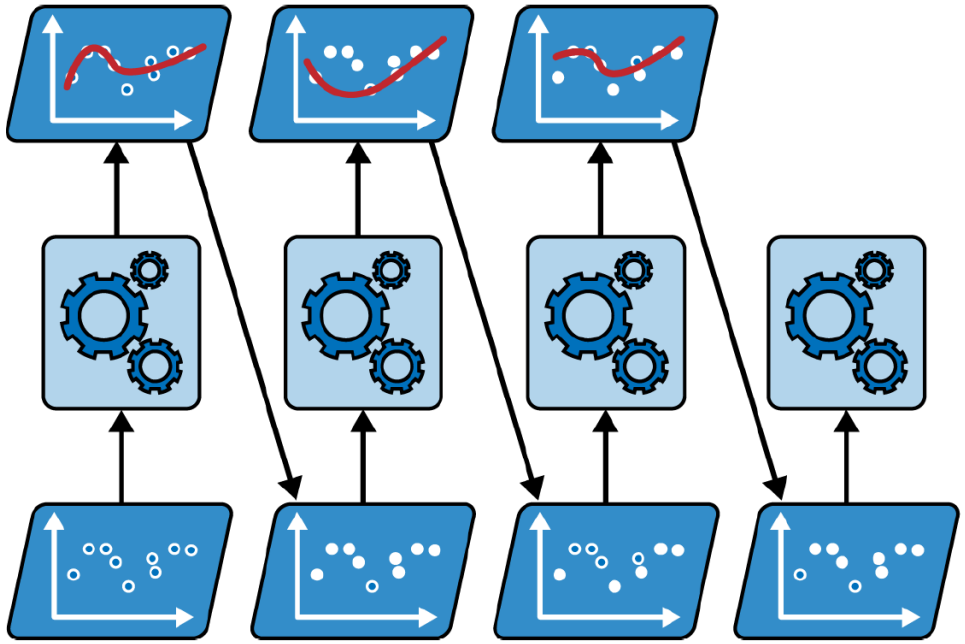
```
from sklearn.datasets import load_iris
iris = load_iris(as_frame=True)

rf_iris = RandomForestClassifier(n_estimators=500, random_state=42)
rf_iris.fit(iris.data, iris.target)

for score, name in zip(rf_iris.feature_importances_, iris.data.columns):
    print(round(score, 2), name)
```

4. Boosting

Boosting é um método de ensemble que combina vários aprendizes fracos para formar um aprendiz forte. A chave é treinar os modelos em sequência, de modo que cada novo modelo corrija os erros do anterior, focando nas instâncias difíceis. Diferente do bagging (que treina em paralelo e reduz variância), o boosting reduz principalmente o viés, refinando o modelo passo a passo.

4.1 AdaBoost: foco nos casos difíceis

No AdaBoost, começamos treinando um modelo simples (ex.: decision stump). Em seguida, aumentamos o peso das instâncias que foram mal classificadas e treinamos o próximo modelo com esses novos pesos. Repetimos o processo, acumulando modelos com pesos α proporcionais à sua qualidade. Ao final, a predição é feita por votação ponderada: quem acerta mais tem mais influência.



Inicialize $w_i = 1/m$ .Em cada iteração j, treine um preditor h_j com os pesos atuais. Meça o erro ponderado r_j. Em seguida, compute o peso do preditor α_j.

Intuição: se o preditor foi bom ($r_j < 0{,}5$), αj é positivo e aumenta seu voto; se foi ruim, $\alpha_j$tende a zero (ou negativo, em casos extremos).

$$r_j = \sum_{\substack{i=1 \\ \hat{y}_j^{(i)} \neq y^{(i)}}}^{m} w^{(i)} \quad \text{where } \hat{y}_j^{(i)} \text{ is the } j^{\text{th}} \text{ predictor's prediction for the } i^{\text{th}} \text{ instance}$$

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

Atualize os pesos das instâncias: mantenha w_i se o preditor acertou; multiplique por e^{α_j} se ele errou. Depois, normalize para que $\sum_i w_i = 1$ .Repita o ciclo: treine o próximo preditor, recalcule r_j ,α_j ,atualize w_i .Pare quando atingir o número de preditores desejado ou quando houver um preditor perfeito.

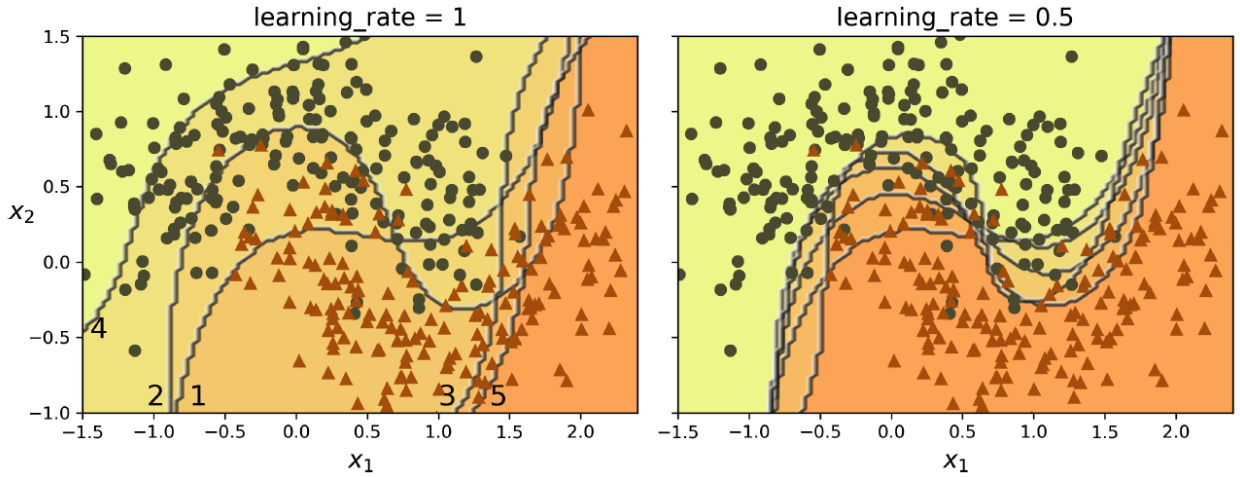
$$\text{for } i = 1, 2, \cdots, m$$

$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{if } \widehat{y_j}^{(i)} = y^{(i)} \\ w^{(i)} \exp\left(\alpha_j\right) & \text{if } \widehat{y_j}^{(i)} \neq y^{(i)} \end{cases}$$

Predição final e SAMME / SAMME.R

A predição do AdaBoost usa votação ponderada: escolhe a classe com maior soma de votos $\sum_j \alpha_j$ .No Scikit-Learn, a versão multiclasse é SAMME; quando os preditores fornecem probabilidades (predict_proba), usa-se SAMME.R, que normalmente performar melhor, pois pondera por confiança das previsões. A learning rate η controla a força de cada iteração: menor η → mais estável, porém exige mais estimadores.

Ao visualizar as fronteiras de decisão de preditores consecutivos (ex.: SVM RBF muito regularizado) no conjunto *moons*, vemos que o primeiro comete muitos erros; após aumentar o peso dessas instâncias, o segundo melhora nesses pontos, e assim por diante. Se reduzimos a learning rate pela metade, cada passo é mais cauteloso: a sequência de fronteiras evolui de forma mais suave, exigindo mais iterações para atingir desempenho semelhante, com tendência a melhor generalização.

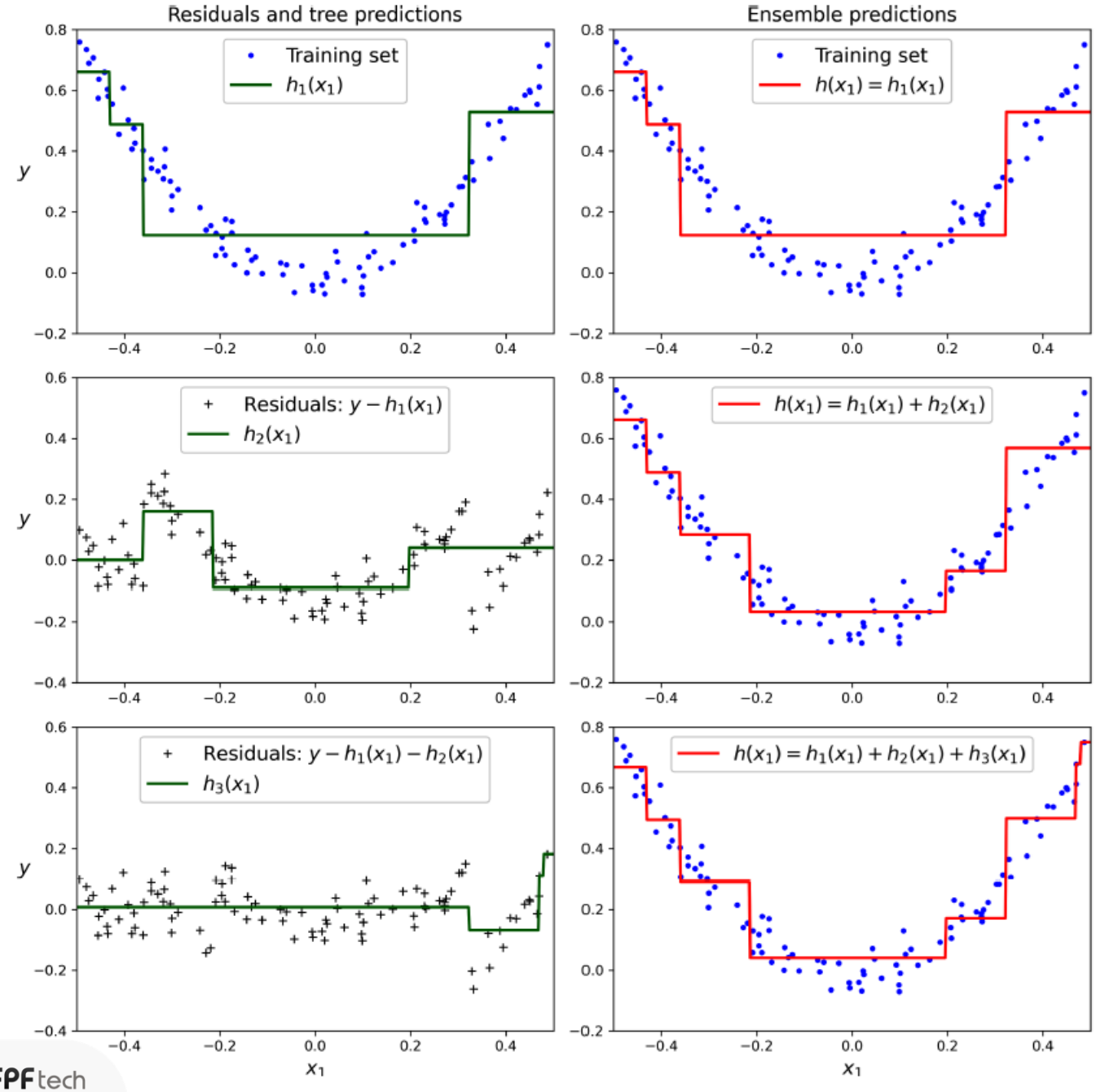


```python
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1),
    n_estimators=30,
    learning_rate=0.5,
    random_state=42
)
ada_clf.fit(X_train, y_train)
```

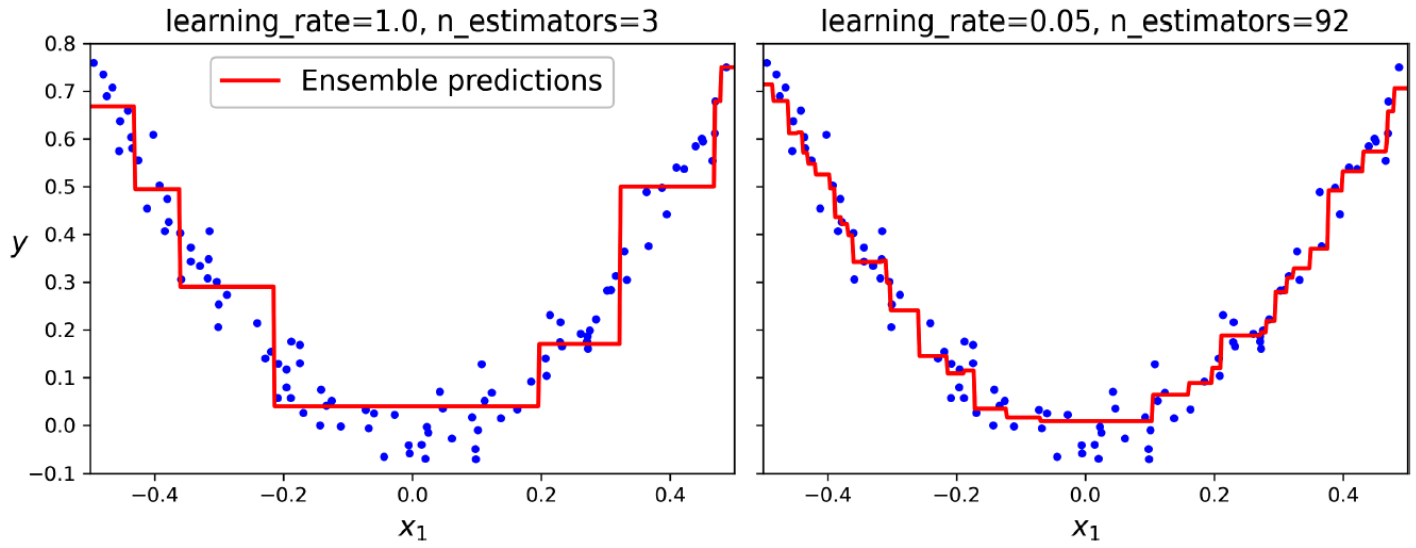
4.2 Intuição do Gradient Boosting

Gradient Boosting constrói o modelo passo a passo, treinando o próximo preditor nos resíduos do anterior. Em vez de reponderar instâncias (como o AdaBoost), ajustamos diretamente os erros remanescentes. Com árvores de decisão rasas (GBRT), a predição final é a soma das árvores: cada nova árvore corrige o que ficou faltando, melhorando gradualmente o ajuste.

No GBRT de regressão, treinamos a 1ª árvore para modelar y. Calculamos os resíduos $r_1 = y - \hat{y}_1$ e treinamos a 2ª árvore para modelar r_1 .Depois calculamos $r_2 = r_1 - \hat{r}_1$ e treinamos a 3ª árvore, e assim por diante. A predição do ensemble é a soma das previsões das árvores (possivelmente escaladas por uma *learning rate*). Esse processo refina o modelo a cada iteração.


Residuals and tree predictions
Ensemble predictions
Training set
$h_1(x_1)$
$h(x_1) = h_1(x_1)$
Residuals: $y - h_1(x_1)$
$h_2(x_1)$
$h(x_1) = h_1(x_1) + h_2(x_1)$
Residuals: $y - h_1(x_1) - h_2(x_1)$
$h_3(x_1)$
$h(x_1) = h_1(x_1) + h_2(x_1) + h_3(x_1)$
y
x_1

O hiperparâmetro learning_rate controla a força de cada passo. Com learning_rate pequeno, precisamos de mais árvores, porém generalizamos melhor (regularização por *shrinkage*). Se usarmos poucas árvores, subajustamos; se passarmos do ponto, sobreajustamos. O objetivo é encontrar um equilíbrio entre quantidade de árvores e *learning rate*.



Para evitar *overfitting* sem *grid search* completo, use early stopping: defina n_iter_no_change (ex.: 10). O treinamento para quando não há melhora por esse número de iterações. O método usa uma fração de validação (validation_fraction, padrão 10%) para monitorar o ganho, e tol define o que conta como melhora mínima. Na prática, isso escolhe automaticamente a quantidade adequada de árvores.

Definir subsample < 1.0 (ex.: 0.25) faz cada árvore treinar em uma amostra aleatória do conjunto de treino. Isso aumenta a diversidade entre árvores, reduz variância, acelera o treino, e funciona como regularização adicional. Em contrapartida, pode aumentar um pouco o viés.

AdaBoost: repondera instâncias; cada modelo recebe peso α; útil para reduzir viés com *stumps*; sensível a *outliers* se não regularizado.
GBRT: ajusta modelos aos resíduos; *learning rate* e *early stopping* são chaves; versão estocástica (subsample) reduz variância e acelera.
Ambos constroem o ensemble em sequência, mas corrigem o erro de formas diferentes.

4.3 Histogram-Based Gradient Boosting

O Histogram-Based Gradient Boosting acelera o treinamento ao agrupar cada feature contínua em bins e operar com inteiros, reduzindo o número de thresholds avaliados e eliminando a necessidade de ordenar as features por nó. Isso muda a complexidade para O(b × m) (nº de bins × nº de instâncias), permitindo ganhar ordens de grandeza em velocidade. O binning traz uma regularização implícita que pode reduzir o overfitting (ou, às vezes, causar underfitting, dependendo dos dados).

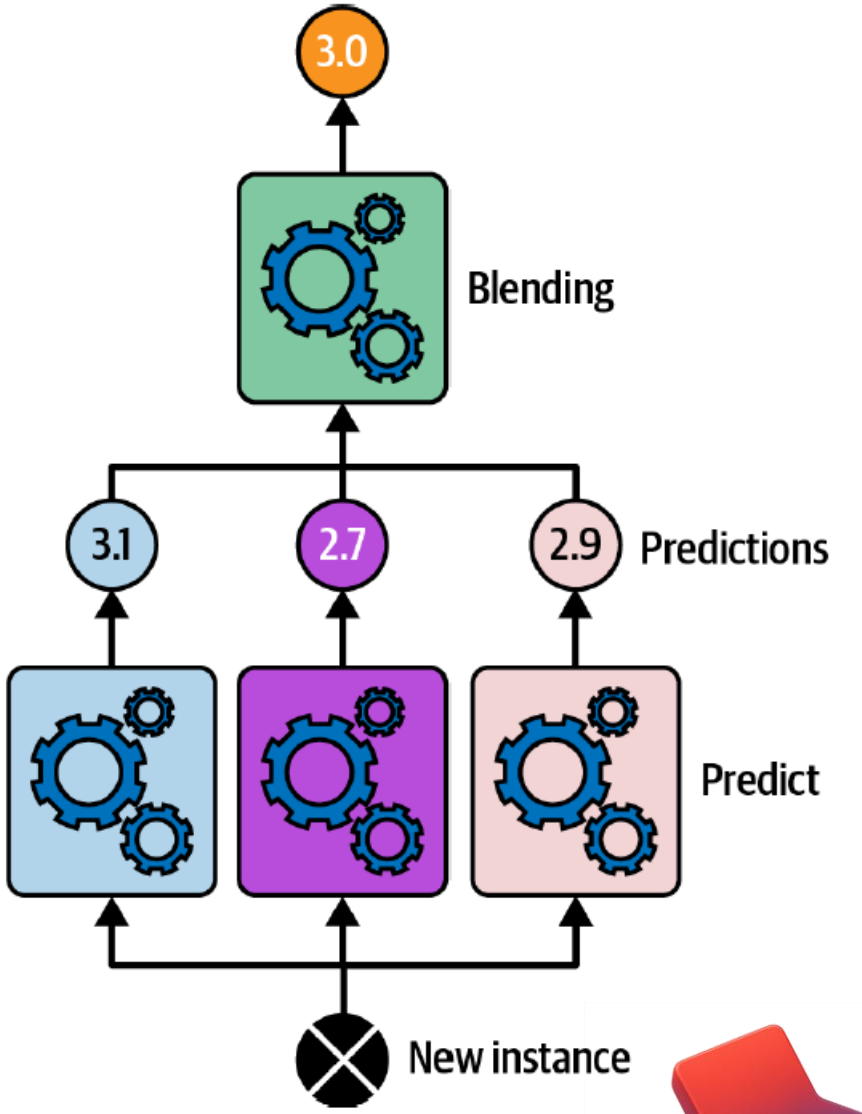
As classes são HistGradientBoostingRegressor e HistGradientBoostingClassifier. Elas habilitam *early stopping* automaticamente se houver mais de 10.000 instâncias (ajustável via early_stopping). n_estimators vira max_iter, não há subsample, e os hiperparâmetros de árvore expostos são max_leaf_nodes, min_samples_leaf e max_depth. Outro benefício é suportar faltantes e categóricas diretamente (após OrdinalEncoder para inteiros < max_bins).

Com HGB fica mais simples o pré-processamento: basta codificar a categórica como ordinal (inteiros) e passar as demais colunas “no passthrough”. Assim, não precisamos de imputer, scaler nem one-hot. O resultado é um fluxo curto e eficaz que já entrega um RMSE razoável sem tuning.

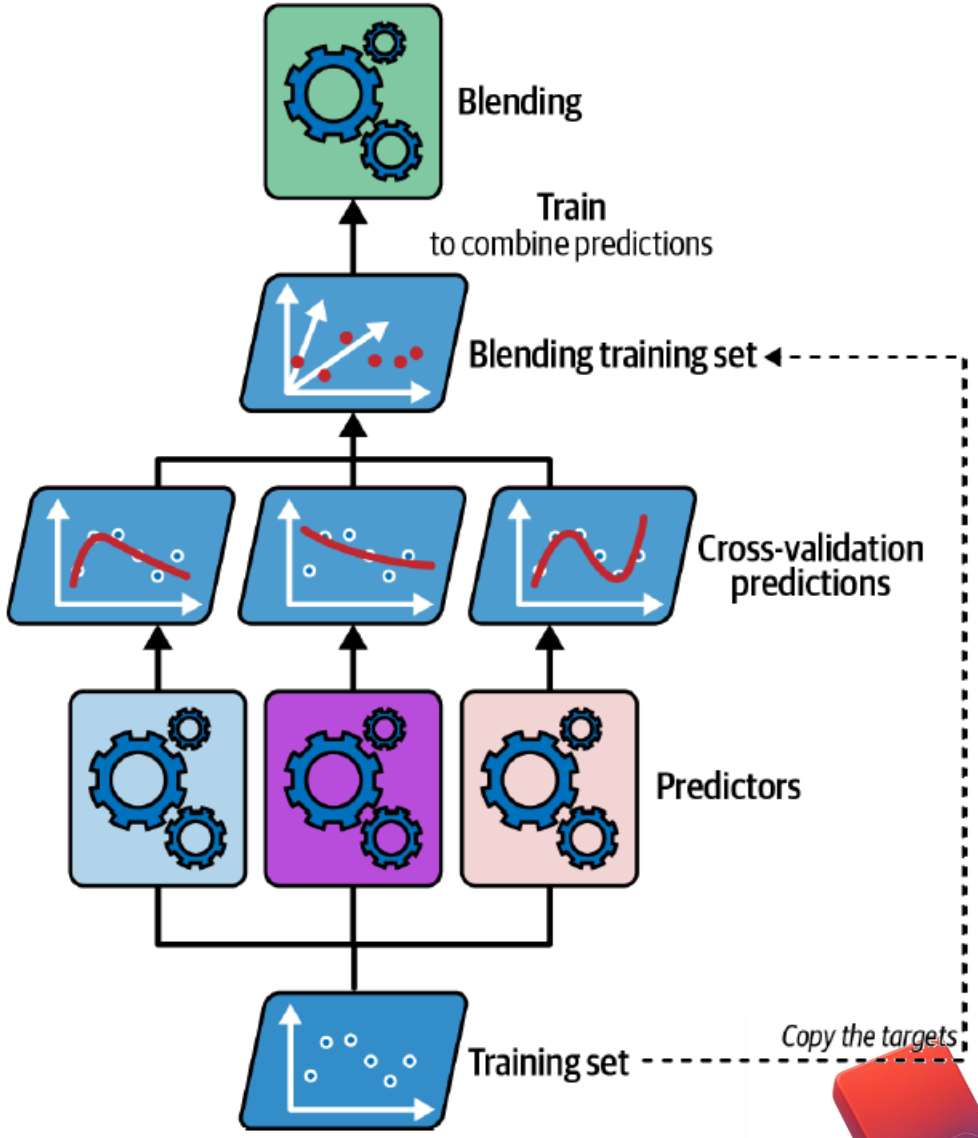
5. Stacking: agregando com um meta-modelo

Ao invés de combinar predições com regras simples (votação/média), o stacking treina um blender para aprender a melhor forma de combinar as saídas dos modelos base. Cada preditor de base gera uma predição para a nova instância; o blender recebe esses valores e produz a previsão final.

três modelos base geram 3.1, 2.7 e 2.9; o blender combina e prevê 3.0.

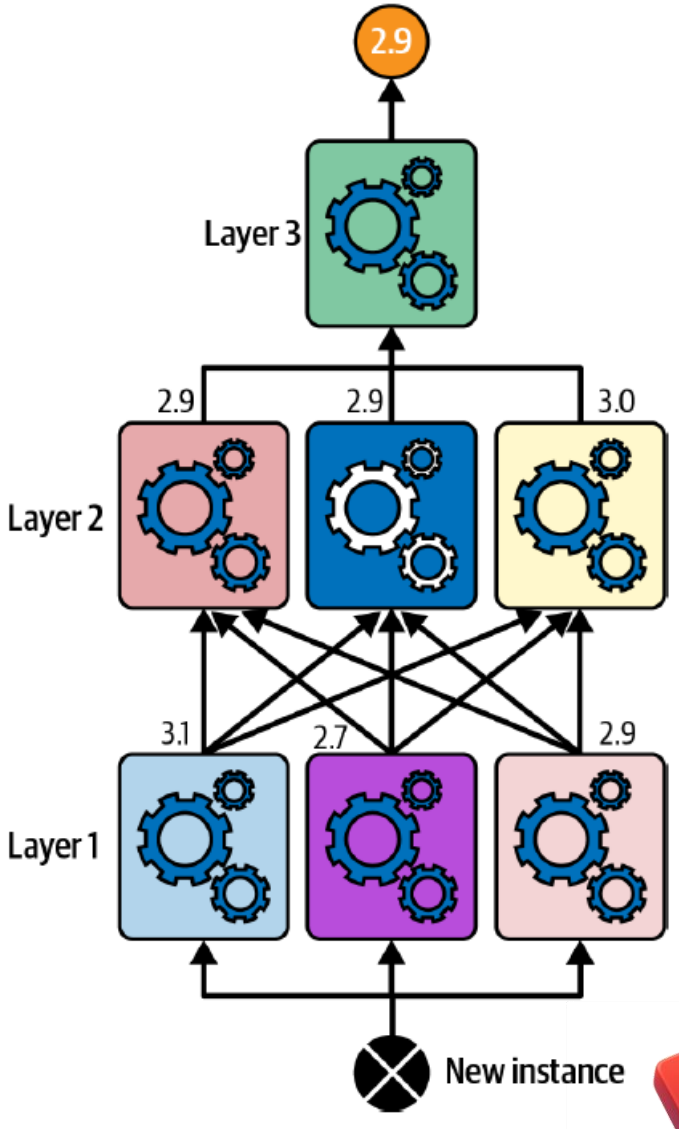


Usamos cross_val_predict para obter previsões fora da amostra de cada modelo base e construímos um novo conjunto de treino onde cada coluna é a predição de um modelo base. Esse conjunto treina o blender, enquanto os modelos base, ao final, são re-treinados no treino completo.



Podemos criar camadas de blenders (nível 1, nível 2...), espremendo um pouco mais de performance ao custo de treino mais demorado e maior complexidade operacional. Avaliar o ganho vs custo é crucial.

Com StackingClassifier/StackingRegressor, definimos os modelos base, o final_estimator (blender) e cv (folds). Por padrão, o stacking usa predict_proba se existir; senão, decision_function; senão, predict. Em um exemplo no *moons*, o stacking alcança ~92,8% de acurácia, superando o soft voting (~92%).



```python
from sklearn.pipeline import make_pipeline
from sklearn.compose import make_column_transformer
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.preprocessing import OrdinalEncoder

hgb_reg = make_pipeline(
    make_column_transformer(
        (OrdinalEncoder(), ["ocean_proximity"]),
        remainder="passthrough"
    ),
    HistGradientBoostingRegressor(categorical_features=[0], random_state=42)
)
hgb_reg.fit(housing, housing_labels)  # housing/housing_labels do Cap. 2
```

```python
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier, StackingClassifier
from sklearn.svm import SVC

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)

stacking_clf = StackingClassifier(
    estimators=[
        ('lr', LogisticRegression(random_state=42)),
        ('rf', RandomForestClassifier(random_state=42)),
        ('svc', SVC(probability=True, random_state=42))
    ],
    final_estimator=RandomForestClassifier(random_state=43),
    cv=5
)
stacking_clf.fit(X_train, y_train)
stack_acc = stacking_clf.score(X_test, y_test)
stack_acc
# Valor típico reportado no texto: ~0.928 (92,8%), acima do soft voting (~92%).
```

FPFtech