Metodo de Conjuntos em Python

November 11, 2025

1 Introdução

Bagging, boosting e stacking pertencem a uma classe de algoritmos de aprendizado de máquina conhecidos como algoritmos de aprendizado em conjunto. O aprendizado em conjunto envolve combinar as previsões de múltiplos modelos em um só para aumentar o desempenho da previsão.

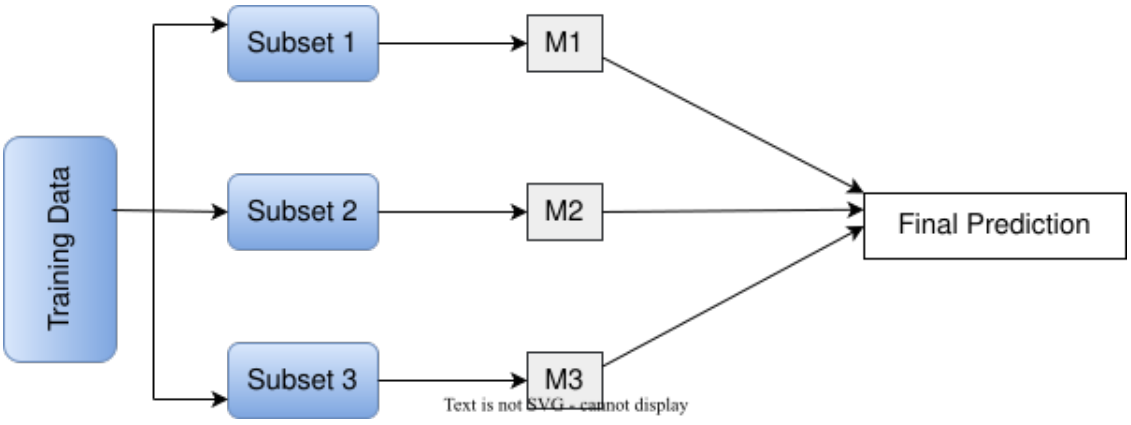
1.1 Bagging

Bagging, também conhecido como agregação bootstrap, é uma técnica de aprendizado em conjunto que combina os benefícios do bootstrapping e da agregação para produzir um modelo estável e melhorar o desempenho da previsão de um modelo de aprendizado de máquina.

No bagging, primeiro amostramos subconjuntos de dados de tamanho igual de um conjunto de dados com bootstrapping, ou seja, amostramos com reposição. Em seguida, usamos esses subconjuntos para treinar vários modelos fracos independentemente. Um modelo fraco é aquele com baixa precisão de previsão. Em contraste, modelos fortes são muito precisos. Para obter um modelo forte, agregamos as previsões de todos os modelos fracos:

[25]:
```
from IPython.display import Image
Image(filename='img_1.png')
```

[25]:



1.2 Passos do Bagging

1. Amostrar subconjuntos de tamanho igual com reposição.

2. Treinar modelos fracos em cada um dos subconjuntos de forma independente e em paralelo.
3. Combinar os resultados de cada um dos modelos fracos por meio de média ou votação para obter um resultado final.

Os resultados são agregados pela média dos resultados para tarefas de regressão ou pela escolha da classe majoritária em tarefas de classificação.

1.3 Algoritmos que Usam Bagging

A ideia principal por trás do bagging é reduzir a variância em um conjunto de dados, garantindo que o modelo seja robusto e não influenciado por amostras específicas no conjunto de dados.

Por essa razão, o bagging é principalmente aplicado a modelos de aprendizado de máquina baseados em árvores, como árvores de decisão e florestas aleatórias.

1.3.1 Prós e Contras do Bagging

Aqui está um resumo rápido do bagging:

Prós	Contras
Reduz a variância geral	Alto número de modelos fracos pode reduzir a interpretabilidade do modelo
Aumenta a robustez dos modelos ao ruído nos dados	

1.4 Implementando Bagging do zero

Para simplificar, usaremos o Scikit-Learn para acessar um conjunto de dados de aprendizado bem conhecido, o modelo base learner e algumas funções para tarefas como dividir o conjunto de dados. Também usaremos o NumPy para lidar com os dados como arrays, incluindo a seleção aleatória de um subconjunto dos dados para treinar os learners fracos.

[1]:
```
from sklearn.base import clone
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.tree import DecisionTreeClassifier
import numpy as np
from scipy.stats import mode

# Define a classe SimpleBag sem herança
class SimpleBag:
    def __init__(self, base_estimator=None, n_estimators=10, subset_size=0.8):
        """
        Inicializa a classe SimpleBag com os parâmetros fornecidos.

        Parâmetros:
```

```
        base_estimator (objeto): O estimador base a ser usado. Se não for␣
↪fornecido, usa DecisionTreeClassifier.
        n_estimators (int): O número de estimadores base no conjunto.
        subset_size (float): A fração do conjunto de dados de treinamento usada␣
↪para bootstrap de cada modelo fraco.
        """
        self.base_estimator = base_estimator if base_estimator else␣
↪DecisionTreeClassifier(max_depth=1, max_features=1)
        self.n_estimators = n_estimators
        self.subset_size = subset_size
        self.base_learners = []
        self.is_fitted = False

    def fit(self, X, y):
        """
        Treina os base learners no conjunto de dados de treinamento fornecido.

        Parâmetros:
        X (array-like): Conjunto de dados de treinamento.
        y (array-like): Rótulos de treinamento.
        """
        n_samples = X.shape[0]
        subset_size = int(n_samples * self.subset_size)
        self.base_learners = []

        for _ in range(self.n_estimators):
            # Seleciona um subconjunto aleatório com reposição
            indices = np.random.choice(range(n_samples), size=subset_size,␣
↪replace=True)
            X_subset, y_subset = X[indices], y[indices]
            # Clona o estimador base e treina no subconjunto
            cloned_estimator = clone(self.base_estimator)
            cloned_estimator.fit(X_subset, y_subset)
            self.base_learners.append(cloned_estimator)

        self.is_fitted = True

    def predict(self, X):
        """
        Gera previsões para novos dados usando os base learners treinados.

        Parâmetros:
        X (array-like): Conjunto de dados de teste.

        Retorna:
        array: Previsões finais do conjunto.
        """
```

```
        if not self.is_fitted:
            raise Exception("Esta instância de SimpleBag ainda não foi treinada.
↪")

        # Gerar previsões de cada base learner
        predictions = np.array([learner.predict(X) for learner in self.
↪base_learners]).T
        # Agrega as previsões usando a moda
        final_predictions, _ = mode(predictions, axis=1, keepdims=True)
        return final_predictions.ravel()
```

1.5 Inicializando o SimpleBag

Inicializamos o SimpleBag com os seguintes parâmetros:

base_estimator: Usamos o DecisionTreeClassifier por padrão, mas qualquer modelo fraco que siga a interface do SciKit Learn pode ser usado. As árvores de decisão são conhecidas por sua alta variância. No entanto, ao criar múltiplos modelos em diferentes subconjuntos dos dados, reduzimos a variância geral graças ao teorema central do limite.

n_estimators: O número de estimadores base no conjunto. Mais estimadores geralmente levam a um melhor desempenho, mas aumentam o custo computacional e o risco de overfitting.

subset_size: A fração do conjunto de dados de treinamento usada para bootstrap de cada modelo fraco. Isso controla o tamanho dos subconjuntos e é um parâmetro chave no bagging, pois afeta a diversidade dos modelos no conjunto.

Método fit() No método fit(), usamos amostragem aleatória com reposição (np.choice()) para criar um subconjunto bootstrap do conjunto de dados de treinamento. Em seguida, clonamos o estimador base e treinamos a nova instância no subconjunto bootstrap. Finalmente, adicionamos o modelo recém-treinado à lista de base_learners.

Método predict() No método predict(), usamos cada modelo treinado para prever uma entrada. Em seguida, agregamos as previsões para fazer uma previsão final. Neste caso, adicionamos todas as previsões a um array e escolhemos a mais comum como nosso resultado geral. Usamos a função mode do Scipy para simplificar a busca pela previsão mais comum.

1.5.1 Comparação de Desempenho

Agora, usando o conjunto de dados Iris, vamos comparar o desempenho do nosso modelo de conjunto completo com o de uma única árvore de decisão.

[2]:
```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.tree import DecisionTreeClassifier

# Carregar os dados
X, y = load_iris(return_X_y=True)
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Treinar e avaliar uma única Árvore de Decisão
single_tree = DecisionTreeClassifier(max_depth=1, max_features=1)
single_tree.fit(X_train, y_train)
single_tree_predictions = single_tree.predict(X_test)
single_tree_accuracy = accuracy_score(y_test, single_tree_predictions)

# Inicializar, treinar e avaliar o modelo SimpleBag
simple_bag = SimpleBag(n_estimators=100, subset_size=0.5)
simple_bag.fit(X_train, y_train)
simple_bag_predictions = simple_bag.predict(X_test)
simple_bag_accuracy = accuracy_score(y_test, simple_bag_predictions)

# Imprimir a precisão dos modelos
print(f'Precisão do modelo de Árvore de Decisão única: {single_tree_accuracy:.
 ↪2f}')
print(f'Precisão do modelo de conjunto SimpleBag: {simple_bag_accuracy:.2f}')
```

```
Precisão do modelo de Árvore de Decisão única: 0.47
Precisão do modelo de conjunto SimpleBag: 0.63
```

O modelo DecisionTreeClassifier, configurado com max_depth=1 e max_features=1, apresenta uma precisão aceitável de 0,63. Esse desempenho é notável, mas típico para um conjunto de dados pequeno como o que estamos usando. No entanto, as árvores de decisão são conhecidas por sua alta variância e tendência ao overfitting.

No nosso conjunto SimpleBag, usamos múltiplas árvores de decisão treinadas em diferentes subconjuntos do conjunto de dados de treinamento, o que reduz a variância geral de nossas previsões, graças ao teorema central do limite. Se executarmos o código várias vezes, notaremos que nossos resultados variam significativamente, desde igualar um modelo fraco até alcançar uma precisão perfeita. Isso se deve ao pequeno conjunto de dados e ao nosso processo de amostragem simplista, mas é suficiente para ilustrar a abordagem de bagging.

1.6 Usando Modelos de Bagging Existentes

Em vez de implementar o algoritmo do zero, devemos aproveitar os modelos de bibliotecas bem estabelecidas. Isso não só reduz o trabalho, mas uma biblioteca com uma base de usuários ampla terá maior qualidade, será mais rápida e terá menos bugs do que um código desenvolvido internamente.

O Scikit-learn é a biblioteca mais popular para algoritmos básicos de aprendizado de máquina em Python. Ela oferece um conjunto abrangente de ferramentas e implementações de algoritmos, incluindo uma para bagging conhecida como BaggingClassifier.

[5]:
```
from sklearn.datasets import load_iris
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
```

```python
from sklearn.metrics import accuracy_score, classification_report,␣
 ↪confusion_matrix

# Carregar o conjunto de dados Iris
X, y = load_iris(return_X_y=True)

# Dividir o conjunto de dados em conjuntos de treinamento e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Inicializar o BaggingClassifier com um DecisionTreeClassifier como estimador␣
 ↪base
bagging_model = BaggingClassifier(
    estimator=DecisionTreeClassifier(max_depth=2, max_features=1),
    n_estimators=10,
    random_state=42
)

# Treinar o modelo nos dados de treinamento
bagging_model.fit(X_train, y_train)

# Fazer previsões nos dados de teste
predictions = bagging_model.predict(X_test)

# Avaliar o modelo
accuracy = accuracy_score(y_test, predictions)

# Imprimir a precisão do BaggingClassifier do scikit-learn
print(f'Precisão do BaggingClassifier do scikit-learn: {accuracy:.2f}')

print(classification_report(y_test, predictions))

print(confusion_matrix(y_test, predictions))
```

```
Precisão do BaggingClassifier do scikit-learn: 1.00
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       1.00      1.00      1.00         9
           2       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30

[[10  0  0]
 [ 0  9  0]
```

```
 [ 0  0 11]]
```

Neste exemplo, replicamos a configuração da nossa abordagem de bagging personalizada, mas utilizamos o BaggingClassifier fornecido pelo scikit-learn. Especificamos um DecisionTreeClassifier com max_depth=2 e max_features=1 como o estimador base para corresponder de perto ao nosso experimento anterior. O BaggingClassifier lida com as complexidades de treinar cada estimador base em amostras bootstrap e agregar suas previsões, simplificando o processo em algumas linhas de código.

1.7 Boosting

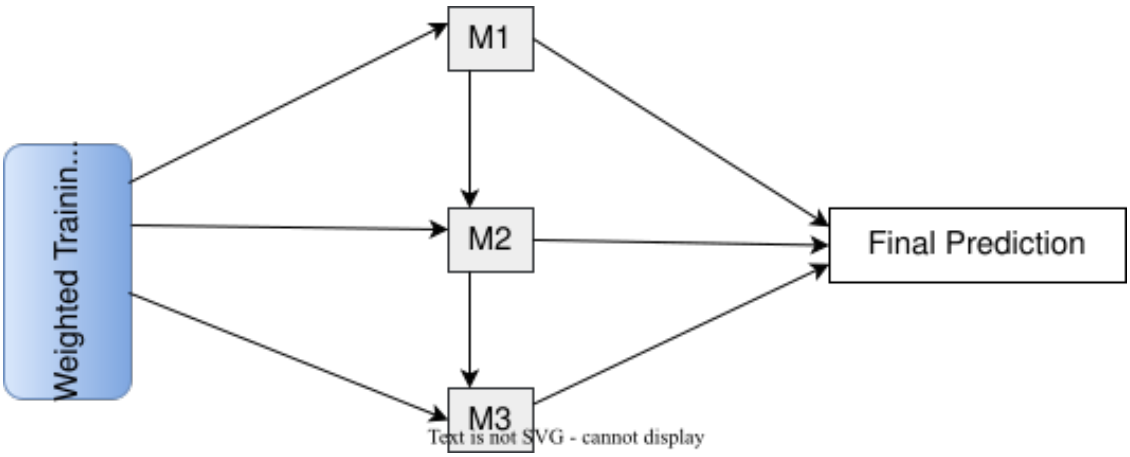
No boosting, treinamos uma sequência de modelos. Cada modelo é treinado em um conjunto de treinamento ponderado. Atribuímos pesos com base nos erros dos modelos anteriores na sequência.

A ideia principal por trás do treinamento sequencial é fazer com que cada modelo corrija os erros de seu predecessor. Isso continua até que o número predefinido de modelos treinados ou algum outro critério seja atendido.

Durante o treinamento, instâncias que são classificadas incorretamente recebem pesos mais altos para dar alguma forma de prioridade quando treinadas com o modelo seguinte:

[29]:
```
Image(filename='img_2.png')
```

[29]:



1.7.1 Passos no Boosting

Inicializar os pesos dos dados com o mesmo valor.

1. Treinar um modelo em todas as instâncias.
2. Calcular o erro na saída do modelo em todas as instâncias.
3. Atribuir um peso ao modelo (alto para bom desempenho e vice-versa).
4. Atualizar os pesos dos dados: dar pesos mais altos às amostras com altos erros.
5. Repetir os passos anteriores se o desempenho não for satisfatório ou se outras condições de parada forem atendidas.

Finalmente, combinar os modelos no que usamos para previsão.

1.7.2 Algoritmos que Usam Boosting

O boosting geralmente melhora a precisão de um modelo de aprendizado de máquina ao melhorar o desempenho dos learners fracos. Tipicamente usamos XGBoost, CatBoost e AdaBoost.

Esses algoritmos aplicam diferentes técnicas de boosting e são mais conhecidos por alcançar excelente desempenho.

1.7.3 Prós e Contras do Boosting

O boosting tem muitas vantagens, mas não está isento de desvantagens:

Prós	Contras
Melhora a precisão geral Reduz o viés geral ao melhorar a fraqueza do modelo anterior	Pode ser computacionalmente caro Sensível a dados ruidosos A dependência do modelo pode permitir a replicação de erros

1.7.4 Implementando Boosting quase do zero

Existem vários algoritmos de boosting, todos baseados no ajuste do treinamento, que são usados para prever o desempenho do próximo learner com base nos learners treinados anteriormente.

Gradient Boosting, eXtreme Gradient Boosting e Light Gradient Boosting ajustam os novos learners aos erros residuais, usando uma abordagem de descida de gradiente na função de perda para minimizar outros. No entanto, eles fazem diferentes trade-offs entre velocidade e desempenho, uso de regularização e sua capacidade de lidar com dados esparsos ou em grande quantidade. Eles são utilizáveis para cenários de classificação e regressão.

Categorical Boosting se especializa em problemas de categorização. Pode ser usado sem exigir pré-processamento de dados, como converter as categorias para codificação one-hot. Também é resistente ao overfitting.

Adaptive Boosting foi inicialmente desenvolvido para tarefas de classificação, mas foi modificado para problemas de regressão. Ele usa pesos de amostra variáveis para direcionar o treinamento de novos learners para um melhor desempenho nos dados de treinamento em que os learners anteriores tiveram um desempenho ruim.

Agora, implementaremos um algoritmo básico de boosting adaptativo usando modelos do scikit-learn.

Como precisamos passar os pesos das amostras para os modelos subjacentes, só podemos usar modelos base que suportem a passagem de pesos para seu método de ajuste; algoritmos como Decision Trees, Logistic Regression, Ridge Classifier e Support Vector Machines no scikit-learn permitem que passemos pesos para o processo de treinamento.

[30]:
```
from sklearn.base import BaseEstimator, ClassifierMixin, clone
from sklearn.datasets import load_iris
```

```python
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import LabelEncoder
import numpy as np

class SimpleMultiClassBoosting(BaseEstimator, ClassifierMixin):
    def __init__(self, base_estimator=None, n_estimators=50):
        """
        Inicializa a classe SimpleMultiClassBoosting com os parâmetros␣
 ↪fornecidos.

        Parâmetros:
        base_estimator (objeto): O estimador base a ser usado. Se não for␣
 ↪fornecido, usa DecisionTreeClassifier.
        n_estimators (int): O número de estimadores base no conjunto.
        """
        self.base_estimator = base_estimator if base_estimator is not None else␣
 ↪DecisionTreeClassifier(max_depth=1)
        self.n_estimators = n_estimators
        self.learners = []
        self.learner_weights = []
        self.label_encoder = LabelEncoder()

    def fit(self, X, y):
        """
        Treina os learners no conjunto de dados de treinamento fornecido.

        Parâmetros:
        X (array-like): Conjunto de dados de treinamento.
        y (array-like): Rótulos de treinamento.
        """
        # Converte os rótulos para [0, n_classes-1]
        y_encoded = self.label_encoder.fit_transform(y)
        n_classes = len(self.label_encoder.classes_)

        # Inicializa os pesos uniformemente
        sample_weights = np.full(X.shape[0], 1 / X.shape[0])

        for _ in range(self.n_estimators):
            learner = clone(self.base_estimator)
            learner.fit(X, y_encoded, sample_weight=sample_weights)
            learner_pred = learner.predict(X)

            # Calcula a taxa de erro ponderada (taxa de erro de classificação)
            incorrect = (learner_pred != y_encoded)
```

```
            learner_error = np.mean(np.average(incorrect,␣
↪weights=sample_weights))

            # Calcula o peso do learner usando o algoritmo SAMME
            learner_weight = np.log((1 - learner_error) / (learner_error +␣
↪1e-10)) + np.log(n_classes - 1)
            if learner_error >= 1 - (1 / n_classes):
                break  # Para se o learner não for melhor do que uma␣
↪adivinhação aleatória

            # Aumenta os pesos das amostras classificadas incorretamente
            sample_weights *= np.exp(learner_weight * incorrect *␣
↪(sample_weights > 0))
            sample_weights /= np.sum(sample_weights)  # Normaliza os pesos

            # Salva o learner atual
            self.learners.append(learner)
            self.learner_weights.append(learner_weight)

    def predict(self, X):
        """
        Gera previsões para novos dados usando os learners treinados.

        Parâmetros:
        X (array-like): Conjunto de dados de teste.

        Retorna:
        array: Previsões finais do conjunto.
        """
        # Coleta previsões de cada learner
        learner_preds = np.array([learner.predict(X) for learner in self.
↪learners])

        # Votação ponderada para a previsão de cada amostra em todos os learners
        weighted_preds = np.zeros((X.shape[0], len(self.label_encoder.
↪classes_)))
        for i in range(len(self.learners)):
            weighted_preds[np.arange(X.shape[0]), learner_preds[i]] += self.
↪learner_weights[i]

        # A previsão final é a que tem o maior voto ponderado
        y_pred = np.argmax(weighted_preds, axis=1)
        # Converte de volta para os rótulos de classe originais
        return self.label_encoder.inverse_transform(y_pred)
```

Na implementação acima, começamos codificando os rótulos usando o LabelEncoder do scikit-learn e, em seguida, procedemos para treinar os learners fracos, ajustando os pesos em cada etapa:

1. No início, atribuímos um peso igual a cada amostra. Para um conjunto de dados com N amostras, o peso inicial de cada amostra é definido como 1/N.

Em cada iteração, treinamos um learner fraco, calculamos um peso para o learner e recalculamos os pesos das amostras:

- O learner fraco atual é treinado nos dados de treinamento usando os pesos das amostras atuais.
- Após o treinamento, avaliamos o desempenho do learner, calculando sua taxa de erro ponderada.
- Em seguida, calculamos a taxa de erro para esse learner e a usamos para calcular o peso que daremos às previsões desse learner ao fazer a previsão geral do nosso conjunto. Se o erro do learner mostrar um desempenho pior do que uma adivinhação aleatória, paramos o treinamento e não adicionamos mais learners ao conjunto.
- Finalmente, atualizamos os pesos com base nas previsões do learner atual. Amostras que o learner classificou incorretamente recebem mais peso, e amostras que foram classificadas corretamente têm seus pesos reduzidos.

Para encontrar a previsão final do nosso conjunto, precisamos agregar as previsões individuais dos learners fracos, mas devemos pesar os votos e não simplesmente contar, como fizemos no exemplo de bagging:

1. Fizemos previsões usando todos os learners fracos e armazenamos o resultado em um array NumPy.
2. Em seguida, calculamos a previsão geral. Começamos criando um array com dimensões NxC, onde N é o número de entradas que estamos classificando e C é o número de categorias possíveis. Para cada amostra, adicionamos o peso do learner à categoria que ele previu.
3. Escolhemos a classe que recebe o maior peso total entre todos os learners como a previsão final para a entrada.

Ao comparar nosso conjunto com um único learner fraco, apreciamos o desempenho melhorado. Outro ponto interessante é que o desempenho é mais estável em várias execuções do que nossa implementação simples de bagging.

[31]:
```
# Load data
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Train and evaluate a single Decision Tree
single_tree = DecisionTreeClassifier(max_depth=1, max_features=1)
single_tree.fit(X_train, y_train)
single_tree_predictions = single_tree.predict(X_test)
single_tree_accuracy = accuracy_score(y_test, single_tree_predictions)

# Initialize, fit, and evaluate the SimpleBag model
simple_bag = SimpleMultiClassBoosting(n_estimators=100)
```

```
simple_bag.fit(X_train, y_train)
simple_bag_predictions = simple_bag.predict(X_test)
simple_bag_accuracy = accuracy_score(y_test, simple_bag_predictions)

print(f'Accuracy of the single Decision Tree model: {single_tree_accuracy:.2f}')
print(f'Accuracy of the SimpleMultiClassBoosting ensemble model:␣
 ↪{simple_bag_accuracy:.2f}')
```

```
Accuracy of the single Decision Tree model: 0.63
Accuracy of the SimpleMultiClassBoosting ensemble model: 0.93
```

O modelo base alcança uma precisão de 0,43 a 0,63, enquanto nosso conjunto de boosting simples alcança uma precisão consistente de 0,93. Esse desempenho consistente é um excelente contraste com a variabilidade da nossa implementação de bagging; em termos simples, nossa implementação de boosting não usa aleatoriedade ao treinar o modelo.

1.8 Usando Modelos de Boosting Existentes

Assim como com o algoritmo de bagging, quase sempre é melhor usar uma biblioteca que implemente esses algoritmos. O scikit-learn inclui implementações para diferentes estratégias de boosting: AdaBoostClassifier, AdaBoostRegressor, GradientBoostingClassifier, GradientBoostingRegressor.

Como nossa implementação simples é um algoritmo de Boosting Adaptativo simplificado, podemos compará-la com o AdaBoostClassifier do scikit-learn.

[6]:
```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# Carregar o conjunto de dados Iris
X, y = load_iris(return_X_y=True)

# Dividir o conjunto de dados em conjuntos de treinamento e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=42)

# Inicializar um Classificador de Árvore de Decisão rasa
shallow_decision_tree = DecisionTreeClassifier(max_depth=1)

# Inicializar o Classificador AdaBoost usando a Árvore de Decisão rasa como␣
 ↪estimador base
ada_clf = AdaBoostClassifier(estimator=shallow_decision_tree, n_estimators=100,␣
 ↪random_state=42, algorithm="SAMME")

# Treinar o modelo AdaBoost no conjunto de treinamento
ada_clf.fit(X_train, y_train)
```

```
# Fazer previsões no conjunto de teste
predictions = ada_clf.predict(X_test)

# Avaliar e imprimir a precisão do modelo no conjunto de teste
print("Precisão:", accuracy_score(y_test, predictions))

print(classification_report(y_test, predictions))

print(confusion_matrix(y_test, predictions))
```

```
Precisão: 1.0
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45

[[19  0  0]
 [ 0 13  0]
 [ 0  0 13]]
```

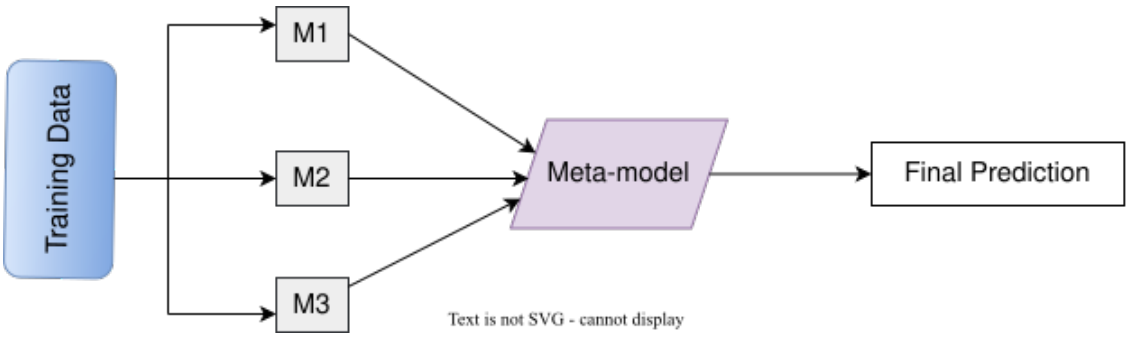
Executando repetidamente, obtemos uma precisão estável de 1.0. Além da maior precisão, também obtemos um modelo extensivamente testado e mais opções que podemos usar para ajustar como treinamos nosso modelo.

1.9 Stacking

No stacking, as previsões dos modelos base são usadas como entrada para um meta-modelo (ou meta-learner). O trabalho do meta-modelo é pegar as previsões dos modelos base e fazer uma previsão final:

[33]:
```
Image(filename='img_3.png')
```

[33]:



1.9.1 Passos no Stacking

1. Construir modelos base em diferentes porções dos dados de treinamento.
2. Treinar um meta-modelo nas previsões dos modelos base.

1.9.2 Prós e Contras do Stacking

Podemos resumir o stacking da seguinte forma:

Prós	Contras
Combina os benefícios de diferentes modelos em um só Aumenta a precisão geral	Pode levar mais tempo para treinar e agregar as previsões de diferentes tipos de modelos Treinar vários modelos base e um meta-modelo aumenta a complexidade

1.10 Implementando Stacking quase do zero

Como antes, podemos usar os modelos base no scikit-learn para escrever uma implementação simples de stacking.

[34]:
```
import numpy as np
from sklearn.base import clone
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.model_selection import cross_val_predict, train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC

class SimpleStacking:
    def __init__(self, base_learners, meta_learner):
        """
        Inicializa a classe SimpleStacking com os base learners e o meta␣
 ↪learner.

        Parâmetros:
        base_learners (list): Lista de modelos base.
        meta_learner (object): Modelo meta que será treinado nas previsões dos␣
 ↪modelos base.
        """
        self.base_learners = base_learners
        self.meta_learner = meta_learner
        self.fitted_base_learners = []

    def fit(self, X, y):
        """
        Treina os base learners e o meta learner.
```

```
        Parâmetros:
        X (array-like): Conjunto de dados de treinamento.
        y (array-like): Rótulos de treinamento.
        """
        meta_features = []
        self.fitted_base_learners = []

        # Treinar base learners e gerar previsões com validação cruzada para␣
↪servir como meta-features
        for base_learner in self.base_learners:
            fitted_learner = clone(base_learner).fit(X, y)
            self.fitted_base_learners.append(fitted_learner)
            preds = fitted_learner.predict(X)
            meta_features.append(preds)

        # Empilhar meta-features horizontalmente
        meta_features = np.array(meta_features).T

        # Treinar o meta-learner nas meta-features
        self.meta_learner.fit(meta_features, y)

    def predict(self, X):
        """
        Gera previsões para novos dados usando o meta learner.

        Parâmetros:
        X (array-like): Conjunto de dados de teste.

        Retorna:
        array: Previsões do meta learner.
        """
        # Gerar meta-features para novos dados
        meta_features = [learner.predict(X) for learner in self.
↪fitted_base_learners]
        meta_features = np.array(meta_features).T
        # Previsão final do meta-learner
        return self.meta_learner.predict(meta_features)
```

Semelhante aos conjuntos anteriores, treinamos um conjunto de base learners. Neste caso, exigimos que o usuário da nossa classe passe um array de base learners instanciados. Treinamos cada learner em paralelo, como fizemos no bagging, mas neste exemplo usamos todos os dados para treinar cada modelo. A principal diferença é que usamos a previsão de cada base learner como uma característica para treinar um meta-learner.

[35]:
```
# Carregar o conjunto de dados Iris
X, y = load_iris(return_X_y=True)
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=42)

# Definir base learners
base_learners = [
    DecisionTreeClassifier(max_depth=1, max_features=1),
    LogisticRegression(random_state=42)
]

# Definir meta-learner
meta_learner = SVC(probability=True, random_state=42)

# Inicializar e treinar o modelo SimpleStacking
stacking_model = SimpleStacking(base_learners, meta_learner)
stacking_model.fit(X_train, y_train)

# Fazer previsões e avaliar o modelo
predictions = stacking_model.predict(X_test)
accuracy = accuracy_score(y_test, predictions)
print(f"Precisão do Modelo de Stacking: {accuracy}")
```

```
Precisão do Modelo de Stacking: 0.9555555555555556
```

Nosso learner simples alcança uma precisão de 1.0 no conjunto de dados Iris, mas tem fraquezas que limitarão o desempenho em problemas mais complexos. Como estamos treinando o meta-learner diretamente na saída dos base learners, podemos acabar ajustando demais os dados de treinamento. Uma maneira comum de resolver esse problema é usar uma abordagem de previsão com validação cruzada (cross_val_predict) para evitar que os dados de entrada vazem para o meta-learner.

Outro problema é que usamos a previsão final da classe dos base learners; isso elimina informações de incerteza porque assume que os base learners têm certeza sobre a previsão. Isso é mais fácil de corrigir, então precisamos usar predict_prob(), que atribui uma probabilidade a cada categoria possível e usa isso como entrada para o meta-learner.

Prevemos como adicionar essas melhorias complicará lentamente nosso código, por isso devemos preferir usar os modelos de uma biblioteca na prática.

1.11 Stacking com SciKit-Learn

Usar o StackingClassifier é tão simples quanto com qualquer um dos outros conjuntos. Precisamos instanciar os base learners e o meta-learner, pois precisamos deles para instanciar a classe de stacking:

[7]:
```
from sklearn.datasets import load_iris
from sklearn.ensemble import StackingClassifier
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
```

```
from sklearn.metrics import accuracy_score

# Carregar o conjunto de dados Iris
X, y = load_iris(return_X_y=True)

# Dividir o conjunto de dados em conjuntos de treinamento e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=42)

# Definir os base learners
base_learners = [
    ('decision_tree', DecisionTreeClassifier(max_depth=1)),  # Árvore de␣
 ↪Decisão com profundidade máxima de 1
    ('lr', LogisticRegression())  # Regressão Logística
]

# Definir o meta-learner
meta_learner = SVC(probability=True, random_state=42)  # Máquina de Vetores de␣
 ↪Suporte (SVM) com probabilidade

# Inicializar o Stacking Classifier com os base learners e o meta-learner
stack_clf = StackingClassifier(estimators=base_learners,␣
 ↪final_estimator=meta_learner, cv=5)

# Treinar o stacking classifier
stack_clf.fit(X_train, y_train)

# Fazer previsões no conjunto de teste
predictions = stack_clf.predict(X_test)

# Avaliar e imprimir a precisão do modelo
print("Precisão do Modelo de Stacking:", accuracy_score(y_test, predictions))

print(classification_report(y_test, predictions))

print(confusion_matrix(y_test, predictions))
```

```
Precisão do Modelo de Stacking: 1.0
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```

```
[[19  0  0]
 [ 0 13  0]
 [ 0  0 13]]
```

```
/usr/lib/python3/dist-packages/sklearn/linear_model/_logistic.py:458:
ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
/usr/lib/python3/dist-packages/sklearn/linear_model/_logistic.py:458:
ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
```

[8]:
```python
from sklearn.datasets import load_iris
from sklearn.ensemble import StackingClassifier
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Carregar o conjunto de dados Iris
X, y = load_iris(return_X_y=True)

# Dividir o conjunto de dados em conjuntos de treinamento e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=42)

# Definir os base learners
base_learners = [
    ('decision_tree', DecisionTreeClassifier(max_depth=1)),  # Árvore de␣
 ↪Decisão com profundidade máxima de 1
    ('lr', LogisticRegression(max_iter=200))  # Regressão Logística com número␣
 ↪máximo de iterações aumentado
]
```

```python
# Definir o meta-learner
meta_learner = SVC(probability=True, random_state=42)  # Máquina de Vetores de␣
 ↪Suporte (SVM) com probabilidade

# Inicializar o Stacking Classifier com os base learners e o meta-learner
stack_clf = StackingClassifier(estimators=base_learners,␣
 ↪final_estimator=meta_learner, cv=5)

# Treinar o stacking classifier
stack_clf.fit(X_train, y_train)

# Fazer previsões no conjunto de teste
predictions = stack_clf.predict(X_test)

# Avaliar e imprimir a precisão do modelo
print("Precisão do Modelo de Stacking:", accuracy_score(y_test, predictions))


print(classification_report(y_test, predictions))

print(confusion_matrix(y_test, predictions))
```

```
Precisão do Modelo de Stacking: 1.0
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45

[[19  0  0]
 [ 0 13  0]
 [ 0  0 13]]
```

Este classificador não é apenas mais sofisticado do que nossa abordagem direta, mas também é flexível. Podemos usar os argumentos do construtor para alterar se os base learners devem se conectar ao meta-learner usando probabilidades de previsão ou a classe. Além disso, podemos decidir usar os base learners como características extras, alimentando o meta-learner com a entrada bruta e os resultados dos base learners.

Diferenças Entre Bagging, Boosting e Stacking As principais diferenças entre bagging, boosting e stacking estão na abordagem, modelos base, seleção de subconjuntos, objetivos e combinação de modelos:

Critério	Bagging	Boosting	Stacking
Abordagem	Treinamento paralelo de modelos fracos	Treinamento sequencial de modelos fracos	Agrega as previsões de múltiplos modelos em um meta-modelo
Modelos Base	Homogêneos	Homogêneos	Podem ser heterogêneos
Seleção de Subconjuntos	Amostragem aleatória com reposição	Subconjuntos não são necessários	Subconjuntos não são necessários
Objetivo	Reduzir variância	Reduzir viés	Reduzir variância e viés
Combinação de Modelos	Votação majoritária ou média	Votação majoritária ponderada ou média	Usando um modelo de aprendizado de máquina

A seleção da técnica a ser usada depende do objetivo geral e da tarefa em questão. O bagging é melhor quando o objetivo é reduzir a variância, enquanto o boosting é a escolha para reduzir o viés. Se o objetivo é reduzir a variância e o viés e melhorar o desempenho geral, devemos usar o stacking.