Uso de geração de valores aleatórios em Python para aprendizado de máquina

Sumário

Introdução

A geração e escolha de valores aleatórios é parte essencial do processo de *Aprendizado de máquina*, conforme indicado a seguir.

Divisão de Dados em Treino e Teste	A divisão aleatória dos dados é importante para garantir que ambas as partes representem a distribuição original dos dados. • Imparcialidade: A randomização garante que a divisão dos dados não seja enviesada. Sem randomização, é possível que o conjunto de treino ou teste seja dominado por certos padrões, o que pode levar a um modelo que não generaliza bem. • Validação Cruzada: Técnicas como *k-fold cross-validation* dependem de uma boa randomização para criar diferentes subconjuntos de dados para treinamento e teste.
Redução de *Overfitting*	*Overfitting* ocorre quando um modelo aprende os detalhes e o ruído do conjunto de treinamento ao ponto de não funcionar bem em novos dados. A randomização ajuda a reduzir o *overfitting*: • Generalização: Ao garantir que os dados de treino e teste sejam representativos de toda a distribuição, a randomização ajuda o modelo a aprender padrões gerais em vez de detalhes específicos. • Robustez: A randomização pode ajudar a criar um modelo mais robusto que lida melhor com dados novos e desconhecidos.
Balanceamento de Classes	Em problemas de classificação, é comum ter classes desbalanceadas. A randomização é essencial ao aplicar técnicas de balanceamento, como *oversampling* e *undersampling*, por causa da representatividade. A randomização garante que as instâncias de diferentes classes sejam tratadas de maneira justa, ajudando a criar um conjunto de dados balanceado que representa melhor todas as classes.
Embaralhamento de Dados	Embaralhar os dados é uma etapa importante no pré-processamento, especialmente em problemas de séries temporais e processamento em lote (*batch*): • Quebra de Sequência: Embaralhar os dados quebra qualquer sequência ou padrão temporal, evitando que o modelo aprenda dependências indesejadas. • Estabilidade do Treinamento: Em processamento em lote, embaralhar os dados antes de dividi-los em lotes ajuda a criar lotes que representam melhor a distribuição geral dos dados, levando a um treinamento mais estável e eficaz.

O Python oferece várias ferramentas para randomização no contexto de aprendizado de máquina, principalmente através das bibliotecas `random` e `numpy`. Além disso, bibliotecas como `scikit-learn` fornecem funções convenientes para randomização de dados.

```
#### Usando random e numpy
import random
import numpy as np
```

```
# Embaralhando uma lista
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
random.shuffle(data)
print("Lista Embaralhada:", data)

# Dividindo dados em treino e teste
data = np.array(data)
train_size = int(0.8 * len(data))
np.random.shuffle(data)
train_data, test_data = data[:train_size], data[train_size:]
print("Dados de Treino:", train_data)
print("Dados de Teste:", test_data)
```

```
# Usando scikit-learn

from sklearn.model_selection import train_test_split

# Criando um conjunto de dados fictício
data = np.arange(1, 101)
labels = np.random.choice([0, 1], size=100)

# Dividindo os dados em treino e teste
X_train, X_test, y_train, y_test = train_test_split(data, labels,
test_size=0.2, random_state=42)
print("Dados de Treino:", X_train)
print("Labels de Treino:", y_train)
print("Dados de Teste:", X_test)
print("Labels de Teste:", y_test)
```

Operações com valores aleatórios em Python – módulo `random`

1. Importação do módulo `random`

Para usar as funções de randomização, primeiro você precisa importar o módulo `Random`. O randomizador padrão, PNG64, é chamado `default_rng`:

```
from numpy.random import default_rng
rng = default_rng()
vals = rng.standard_normal(10)
more_vals = rng.standard_normal(10)
```

2. Geração de Números Aleatórios

a. Números Inteiros

```
# Inicializar o gerador de números aleatórios com PCG64 e uma semente
fixa (opcional)
rng = np.random.default_rng(seed=42)
```

```
# Gerar um número aleatório inteiro entre 1 e 10 (inclusive)
random_number = rng.integers(1, 11)

numero_aleatorio = random.randint(1, 10)
print(numero_aleatorio)
```

b. Números de Ponto Flutuante

Para gerar um número de ponto flutuante aleatório entre 0.0 e 1.0, use a função random():

```
# Gera um número de ponto flutuante aleatório entre 0.0 e 1.0
numero_aleatorio = random.random()
print(numero_aleatorio)
```

Se você quiser um número de ponto flutuante em um intervalo diferente, use a função `uniform(a, b)`:

```
# Gera um número de ponto flutuante aleatório entre 1.5 e 10.5
numero_aleatorio = random.uniform(1.5, 10.5)
print(numero_aleatorio)
```

3. Escolhendo Aleatoriamente de uma Lista

Se você tem uma lista e quer escolher um item aleatório, use a função `choice(lista)`:

```
# Lista de exemplo
frutas = ['maca', 'banana', 'laranja', 'uva']

# Escolhe uma fruta aleatoriamente
fruta_aleatoria = random.choice(frutas)
print(fruta_aleatoria)
```

4. Embaralhando uma Lista

Para embaralhar os itens de uma lista aleatoriamente, use a função `shuffle(lista)`:

```
# Lista de exemplo
cartas = ['A', '2', '3', '4', '5']

# Embaralha a lista de cartas
random.shuffle(cartas)
print(cartas)
```

5. Amostragem

Para selecionar aleatoriamente um determinado número de itens únicos de uma lista, use a função `sample(lista, k)`, onde `k` é o número de itens que você quer selecionar:

```
# Lista de exemplo
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Seleciona 3 números aleatórios sem repetição
amostra = random.sample(numeros, 3)
print(amostra)
```

Pseudoaleatoriedade

Os números gerados pelo módulo `random` são chamados de pseudoaleatórios porque eles são produzidos por um algoritmo determinístico e não por um processo realmente aleatório. No entanto, para a maioria dos propósitos, especialmente em programação e simulações, eles são suficientemente aleatórios.
Se você quiser que a sequência de números aleatórios seja previsível (por exemplo, para testes), defina uma semente com a função `seed()`:

```
# Define a semente
random.seed(95)

# Gera um número aleatório com a semente definida
numero_aleatorio = random.randint(1, 10)
print(numero_aleatorio)
```

Usar uma semente garante que a mesma sequência de números aleatórios será gerada toda vez que o código for executado.

Período

O período de um gerador de números pseudoaleatórios é a quantidade de números que ele pode gerar antes de entrar em um ciclo e começar a repetir a mesma sequência de números. Quanto mais longo for o período, maior a quantidade de números únicos que podem ser gerados antes da repetição. Isso é importante, por exemplo, para:

1. Análise Exploratória de Dados: nos algoritmos de tratamento de desbalanceamento de dados e de divisão de *datasets* (em conjuntos de dados de treinamento e de teste), a geração de números aleatórios é parte primordial. Um período longo evita viés.
2. Simulações: Em simulações que requerem muitos números aleatórios, um período longo garante que os resultados não sejam afetados pela repetição dos números.

3. Jogos e Animações: Jogos e animações que usam números aleatórios para gerar eventos ou comportamentos imprevisíveis beneficiam-se de um período longo para evitar padrões repetitivos.
4. Criptografia: Na criptografia, a segurança pode ser comprometida se os números aleatórios começarem a se repetir, então um período longo é crucial.

Algoritmos de randomização

Algoritmo Gerador Linear Congruente (LCG, *Linear Congruential Generator*)

$$X_{n+1} = (a.X_n + c)\ mod\ m$$

Onde:

X: sequência de números pseudoaleatórios
a: multiplicador
c: incremento
m: módulo
X_0: valor inicial chamado de **semente**

Como Funciona:

1. Semente (Seed): O algoritmo começa com um valor inicial chamado de semente (X_0). Esse valor pode ser definido pelo usuário ou gerado pelo sistema (geralmente usando o tempo atual do sistema).
2. Cálculo Recorrente: Cada número subsequente na sequência é calculado usando a fórmula acima. A semente inicial passa pelo cálculo para produzir X_1, então X_1 é usado para calcular X_2, e assim por diante.
3. Módulo (mod m): O uso do módulo garante que os números gerados estejam dentro de um intervalo específico (geralmente de 0 a m-1).

Em Python:

```
class LinearCongruentialGenerator:
    def __init__(self, seed):
        self.a = 475698
        self.c = 949654675
        self.m = 1000
        self.seed = seed

    def random(self):
        self.seed = (self.a * self.seed + self.c) % self.m
        return self.seed / self.m

# Usando o gerador
lcg = LinearCongruentialGenerator(seed=42)
print(lcg.random())  # Gera um número pseudoaleatório
print(lcg.random())  # Gera outro número pseudoaleatório
```

Mersenne Twister

O nome "Mersenne Twister" tem origem na matemática e na informática, especificamente no campo dos geradores de números pseudoaleatórios:

1. **Mersenne**: Refere-se aos números de Mersenne, que são números primos da forma 2^{n-1}, onde n também é um número primo. Esses números são importantes na teoria dos números e possuem propriedades matemáticas interessantes que os tornam úteis em algoritmos de computação.
2. **Twister**: Este termo pode ser interpretado como "giro" ou "torção", e é frequentemente usado em contextos que envolvem movimento ou mudança de estado. No contexto do Mersenne Twister, o termo pode se referir à ideia de torção ou transformação dos números gerados de forma aleatória.

O método de Mersenne Twister é, portanto, um gerador de números pseudoaleatórios que utiliza números de Mersenne como parte fundamental de seu algoritmo para produzir sequências de números que se aproximam de comportamento aleatório.

Também existe a variante MT19937 deste algoritmo, que é caracterizada pelo tamanho do seu estado interno de 19937 bits, garantindo um período longo e números pseudoaleatórios com boa qualidade.

As principais características dos algoritmos desta categoria são:

- **Período Longo**: antes de repetir a sequência completa de números pseudoaleatórios, o algoritmo Mersenne Twister tem um período longo, igual a (2^{2532}-1) iterações. MT19937, por sua vez, possui um período maior ainda, de (2^{19937}–1) iterações. Isso significa que eles podem gerar números únicos em grande quantidade antes de começar a repetir.
- **Boa Distribuição**: Os números gerados pelos algoritmos Mersenne Twister têm distribuição uniforme adequada para a maioria das aplicações práticas.
- **Eficiência**: os algoritmos são rápidos e eficientes em termos de consumo de recursos computacionais.

O algoritmo Mersenne Twister MT19937 já foi usado por padrão no módulo `random` do Python; pode ser chamado . Já o algoritmo é implementado pela classe **`numpy.random.MT19937(seed)`**

Em Python:

```
import random

# Gera um número de ponto flutuante aleatório entre 0.0 e 1.0
numero_aleatorio = random.MT19937()
print(numero_aleatorio)

# Gera um número inteiro aleatório entre 1 e 10
numero_inteiro_aleatorio = random.randint(1, 10)
print(numero_inteiro_aleatorio)
```

Aplicação: scikit-learn para ML supervisionada com o dataset Iris

O `scikit-learn` oferece várias funções e utilitários para facilitar a randomização no contexto de aprendizado de máquina. Vamos explorar algumas das principais funções e métodos que você pode usar:

- `train_test_split`: Divide os dados em treino e teste.
- `shuffle`: Embaralha os dados.
- `KFold` e `StratifiedKFold`: Realizam validação cruzada.
- `resample`: Realiza amostragem com substituição.
- `ShuffleSplit`: Realiza splits aleatórios.
- `StratifiedShuffleSplit`: Realiza splits aleatórios, preservando a distribuição das classes.
- `RandomizedSearchCV`: Realiza busca aleatória de hiperparâmetros.

Métodos e funções do `scikit-learn` que usam randomização

1. Divisão de Dados: `'train_test_split'`

A função train_test_split é usada para dividir um conjunto de dados em conjuntos de treino e teste. Ela permite a randomização da divisão dos dados para garantir que cada divisão seja representativa e imparcial.

```
from sklearn.model_selection import train_test_split

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

- `X`: Conjunto de dados de entrada.
- `y`: Conjunto de rótulos ou valores alvo.
- `test_size`: Fração do conjunto de dados a ser usada como conjunto de teste (por exemplo, 0.2 para 20%).
- `random_state`: Semente para a geração de números aleatórios. Define a semente para garantir a reprodutibilidade.

2. Amostragem Aleatória: `shuffle`

A função `shuffle` é usada para embaralhar os dados no próprio conjunto de dados, sem preservar uma versão anterior à sua aplicação.

```
from sklearn.utils import shuffle

X, y = shuffle(X, y, random_state=42)
```

- `X`: Conjunto de dados de entrada.
- `y`: Conjunto de rótulos ou valores alvo.
- `random_state`: Semente para garantir a reprodutibilidade.

3. Validação Cruzada: `KFold`, `StratifiedKFold`

Essas classes são usadas para realizar validação cruzada, dividindo os dados em vários folds para treinamento e teste. `StratifiedKFold` é especialmente útil quando as classes estão desbalanceadas.

```
from sklearn.model_selection import KFold, StratifiedKFold

kf = KFold(n_splits=5, shuffle=True, random_state=42)
for train_index, test_index in kf.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
for train_index, test_index in skf.split(X, y):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
```

- `n_splits`: Número de folds.
- `shuffle`: Se `True`, embaralha os dados antes de dividir.
- `random_state`: Semente para garantir a reprodutibilidade.

4. Amostragem Aleatória de Dados: `resample`

A função `resample` é usada para amostrar com substituição, útil para lidar com dados desbalanceados.

```
from sklearn.utils import resample

# Suponha que X e y sejam os dados originais
X_resampled, y_resampled = resample(X, y, replace=True, n_samples=100,
random_state=42)
```

- `replace`: Se `True`, realiza amostragem com substituição.
- `n_samples`: Número de amostras desejadas.
- `random_state`: Semente para garantir a reprodutibilidade.

5. Validação cruzada: `ShuffleSplit`

`ShuffleSplit` é uma estratégia de validação cruzada que realiza divisões aleatórias do conjunto de dados em múltiplos splits.

```
from sklearn.model_selection import ShuffleSplit

splitter = ShuffleSplit(n_splits=5, test_size=0.2, random_state=42)
for train_index, test_index in splitter.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
```

- `n_splits`: Número de splits.
- `test_size`: Fração do conjunto de dados a ser usado como teste.

- `random_state`: Semente para garantir a reprodutibilidade.

6. Divisão de dados: `StratifiedShuffleSplit`

`StratifiedShuffleSplit` é uma estratégia de divisão de dados que preserva a distribuição das classes nos conjuntos de treino e teste, garantindo que ambas as partes tenham uma proporção similar das classes originais. Isso é especialmente útil quando se trabalha com dados desbalanceados.

```
from sklearn.model_selection import StratifiedShuffleSplit

sss = StratifiedShuffleSplit(n_splits=5, test_size=0.2, random_state=42)
for train_index, test_index in sss.split(X, y):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
```

- `n_splits`: Número de splits.
- `test_size`: Fração do conjunto de dados a ser usado como teste.
- `random_state`: Semente para garantir a reprodutibilidade.

7. Busca aleatória: `RandomizedSearchCV`

`RandomizedSearchCV` realiza uma busca aleatória para encontrar os melhores hiperparâmetros do modelo.

```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import RandomForestClassifier

param_dist = {
    'n_estimators': [10, 50, 100, 200],
    'max_features': ['auto', 'sqrt', 'log2'],
    'max_depth': [None, 10, 20, 30],
    'criterion': ['gini', 'entropy']
}

random_search = RandomizedSearchCV(RandomForestClassifier(),
param_distributions=param_dist, n_iter=100, cv=5, random_state=42)
random_search.fit(X, y)
```

- `param_distributions`: Dicionário de parâmetros a serem testados.
- `n_iter`: Número de combinações de parâmetros a serem testadas.
- `cv`: Número de folds para validação cruzada.
- `random_state`: Semente para garantir a reprodutibilidade.

Consolidação: ML supervisionada com o dataset Iris usando scikit-learn

Aqui está um exemplo completo que demonstra como usar várias dessas funções para preparar os dados e treinar um modelo:

```
import numpy as np
```

```
from sklearn.model_selection import train_test_split, KFold,
StratifiedKFold, StratifiedShuffleSplit, RandomizedSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.utils import shuffle, resample
from sklearn.datasets import load_iris

# Carregar dados
data = load_iris()
X, y = data.data, data.target

# Dividir os dados em treino e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Embaralhar os dados
X, y = shuffle(X, y, random_state=42)

# Validação cruzada com KFold
kf = KFold(n_splits=5, shuffle=True, random_state=42)
for train_index, test_index in kf.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
    print(f"Fold: Train indices {train_index}, Test indices
{test_index}")

# Validação cruzada estratificada com StratifiedKFold
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
for train_index, test_index in skf.split(X, y):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
    print(f"Stratified Fold: Train indices {train_index}, Test indices
{test_index}")

# Divisão estratificada com StratifiedShuffleSplit
sss = StratifiedShuffleSplit(n_splits=5, test_size=0.2, random_state=42)
for train_index, test_index in sss.split(X, y):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
    print(f"Stratified Shuffle Split: Train indices {train_index}, Test
indices {test_index}")

# Amostragem com substituição
X_resampled, y_resampled = resample(X, y, replace=True, n_samples=100,
random_state=42)

# RandomizedSearchCV para ajuste de hiperparâmetros
param_dist = {
    'n_estimators': [10, 50, 100, 200],
    'max_features': ['auto', 'sqrt', 'log2'],
    'max_depth': [None, 10, 20, 30],
    'criterion': ['gini', 'entropy']
}
random_search = RandomizedSearchCV(RandomForestClassifier(),
param_distributions=param_dist, n_iter=100, cv=5, random

_state=42)
random_search.fit(X_train, y_train)

print(f"Melhores parâmetros encontrados: {random_search.best_params_}")
print(f"Melhor score: {random_search.best_score_}")
```

```
# Treinar o modelo final com os melhores parâmetros
best_model = random_search.best_estimator_
best_model.fit(X_train, y_train)

# Avaliar o modelo
score = best_model.score(X_test, y_test)
print(f"Modelo testado com score: {score}")
```

Uso de numpy para preparação de dados para ML

Algumas funções e métodos do `numpy` que utilizam randomização na preparação dos dados para aprendizado de máquina são:

- `train_test_split`: Divide os dados em treino e teste.
- `shuffle`: Embaralha os dados.
- `KFold` e `StratifiedKFold`: Realizam validação cruzada.
- `resample`: Realiza amostragem com substituição.
- `ShuffleSplit`: Realiza splits aleatórios.
- `StratifiedShuffleSplit`: Realiza splits aleatórios, preservando a distribuição das classes.
- `RandomizedSearchCV`: Realiza busca aleatória de hiperparâmetros.

Preparação do dataset Iris para ML supervisionada usando `numpy`

Aqui é descrito um processo de randomização e preparação do dataset Iris utilizando essencialmente o módulo `numpy`.

Passo 1: Carregar o Dataset
Passo 2: Dividir Dados em Treino e Teste
Passo 3: Embaralhar os Dados
Passo 4: Validação Cruzada (K-Fold)
Passo 5: Validação Cruzada Estratificada (Stratified K-Fold)
Passo 6: Amostragem com Substituição

```
import numpy as np
from sklearn.datasets import load_iris

# Carregar o dataset Iris
iris = load_iris()
X = iris.data
y = iris.target

# 1. Dividir dados em treino e teste
def train_test_split(X, y, test_size=0.2, random_state=None):
    if random_state is not None:
        np.random.seed(random_state)
    indices = np.random.permutation(len(X))
    test_size = int(len(X) * test_size)
```

```
    train_indices, test_indices = indices[:-test_size], indices[-
test_size:]
    return X[train_indices], X[test_indices], y[train_indices],
y[test_indices]

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

# 2. Embaralhar os dados
def shuffle_data(X, y, random_state=None):
    if random_state is not None:
        np.random.seed(random_state)
    indices = np.random.permutation(len(X))
    return X[indices], y[indices]

X_shuffled, y_shuffled = shuffle_data(X, y, random_state=42)

# 3. Validação cruzada (K-Fold)
def k_fold_cross_validation(X, y, k, random_state=None):
    if random_state is not None:
        np.random.seed(random_state)
    indices = np.random.permutation(len(X))
    fold_size = len(X) // k
    for i in range(k):
        test_indices = indices[i*fold_size:(i+1)*fold_size]
        train_indices = np.concatenate([indices[:i*fold_size],
indices[(i+1)*fold_size:]])
        yield X[train_indices], X[test_indices], y[train_indices],
y[test_indices]

k = 5
for X_train, X_test, y_train, y_test in k_fold_cross_validation(X, y, k,
random_state=42):
    print(f"Train size: {len(X_train)}, Test size: {len(X_test)}")

# 4. Validação cruzada estratificada (Stratified K-Fold)
def stratified_k_fold_cross_validation(X, y, k, random_state=None):
    if random_state is not None:
        np.random.seed(random_state)
    unique_classes, y_indices = np.unique(y, return_inverse=True)
    class_counts = np.bincount(y_indices)
    fold_indices = [[] for _ in range(k)]

    for cls in range(len(unique_classes)):
        cls_indices = np.where(y_indices == cls)[0]
        np.random.shuffle(cls_indices)
        cls_fold_sizes = (class_counts[cls] // k) * np.ones(k,
dtype=int)
        cls_fold_sizes[:class_counts[cls] % k] += 1
        current = 0
        for i in range(k):
            fold_indices[i].extend(cls_indices[current:current +
cls_fold_sizes[i]])
            current += cls_fold_sizes[i]

    for i in range(k):
        test_indices = np.array(fold_indices[i])
        train_indices = np.concatenate([np.array(fold_indices[j]) for j
in range(k) if j != i])
```

```
        yield X[train_indices], X[test_indices], y[train_indices],
y[test_indices]

for X_train, X_test, y_train, y_test in
stratified_k_fold_cross_validation(X, y, k, random_state=42):
    print(f"Stratified Train size: {len(X_train)}, Test size:
{len(X_test)}")

# 5. Amostragem com substituição
def resample_with_replacement(X, y, n_samples, random_state=None):
    if random_state is not None:
        np.random.seed(random_state)
    indices = np.random.choice(len(X), n_samples, replace=True)
    return X[indices], y[indices]

X_resampled, y_resampled = resample_with_replacement(X, y,
n_samples=100, random_state=42)
```