CODE SMELLS NEI SISTEMI ABILITATI AL MACHINE LEARNING

Nei progetti software, è comune imbattersi in soluzioni di implementazione che, pur funzionando correttamente, presentano segnali di debolezza o cattive pratiche. Questi segnali sono conosciuti come *code smells*. Si tratta di pattern di codice che non causano errori immediati, ma che possono compromettere la qualità, la manutenibilità e l'evoluzione del sistema nel tempo.

Nei sistemi che integrano componenti di Machine Learning, i *code smells* assumono caratteristiche particolari. Le pipeline di ML, che combinano fasi di pre-processing dei dati, definizione di modelli, addestramento e validazione, espongono nuove tipologie di *code smells*. Questi possono emergere sia nella gestione dei dati che nelle operazioni di addestramento e inferenza dei modelli.

Un *code smell* in un progetto di Machine Learning si manifesta quando una pratica di sviluppo sub-ottimale viene adottata all'interno di una pipeline o di un componente. Tali pratiche possono rallentare l'esecuzione, rendere il comportamento del modello meno prevedibile, introdurre ambiguità nella gestione dei dati o aumentare la probabilità di errori nascosti difficili da rilevare e correggere.

Individuare questi *smells* e correggerli tempestivamente aiuta a migliorare la qualità del codice e a ridurre il rischio di accumulare debito tecnico che potrebbe ostacolare la futura manutenzione o estensione del sistema. Le sezioni che seguono descrivono alcuni tra i *code smells* più frequenti nei progetti di Machine Learning, spiegandone il contesto, il problema che introducono e le pratiche consigliate per evitarli.

Di seguito ecco alcune categorie di code smells, ciascuna strutturata come:

- CONTESTO
- PROBLEMA
- SOLUZIONE
- FASE ESISTENTE ED EFFETTO
- ESEMPIO PRATICO

CHAIN INDEXING

CONTESTO

In Pandas, `df["one"]["two"]` e `df.loc[:,("one", "two")]` restituiscono lo stesso risultato. `df["one"]["two"]` è chiamato **chain indexing**.

PROBLEMA

Il chain indexing esegue due operazioni separate. Prima, `df["one"]` recupera un sottoinsieme, e poi `["two"]` opera sul risultato. Questo approccio è più lento rispetto all'uso di `df.loc`, che esegue l'operazione in un unico passaggio.

Inoltre, assegnare valori tramite chain indexing può fallire perché Pandas non garantisce se `df["col"]` restituisce una **view** (modifica i dati originali) o una **copia** (crea un nuovo oggetto).

SOLUZIONE

Gli sviluppatori che usano Pandas dovrebbero evitare il chain indexing e utilizzare invece `df.loc` per accedere ai dati.

Fase esistente	Effetto
Importazione e pulizia dei dati	Soggetto a errori & inefficienza

ESEMPIO

Python

```python
### Pandas

import pandas as pd

df = pd.DataFrame([[1,2,3],[4,5,6]])

col = 1
```

```
x = 0

- df[col][x] = 42

+ df.loc[x, col] = 42
```

COLUMNS AND DATATYPE NOT EXPLICITLY SET

CONTESTO

In Pandas, tutte le colonne vengono selezionate per impostazione predefinita quando un DataFrame viene importato da un file o da altre fonti. Il tipo di dato per ciascuna colonna viene definito sulla base della conversione dtype predefinita.

PROBLEMA

Se le colonne non sono esplicitamente selezionate, diventa poco chiaro cosa aspettarsi nello schema dei dati per i processi successivi.

Se i tipi di dati non vengono impostati esplicitamente, il processo può procedere silenziosamente con input inattesi, portando potenzialmente a errori successivi.

Pertanto, ogni operazione di caricamento dati dovrebbe includere i nomi delle colonne e i tipi di dato di ciascuna colonna.

Fase esistente	Effetto
Importazione e pulizia dei dati	Leggibilità

SOLUZIONE

È consigliato specificare **i nomi delle colonne e i tipi di dato** quando si caricano i dati.

ESEMPIO

Python

```
### Pandas

import pandas as pd

df = pd.read_csv('data.csv')

+ df = df[['col1', 'col2', 'col3']]

### Pandas Set DataType

import pandas as pd

- df = pd.read_csv('data.csv')

+ df = pd.read_csv('data.csv', dtype={'col1': 'str', 'col2': 'int',
'col3': 'float'})
```

GRADIENTS NOT CLEARED BEFORE BACKWARD PROPAGATION

CONTESTO

In PyTorch, `optimizer.zero_grad()` azzera i gradienti accumulati, `loss_fn.backward()` esegue la retropropagazione e `optimizer.step()` aggiorna i pesi.

PROBLEMA

Se `optimizer.zero_grad()` non viene chiamato prima di `loss_fn.backward()`, i gradienti si accumulano portando a **gradient explosion**, che compromette l'addestramento.

SOLUZIONE

Gli sviluppatori non devono dimenticare di usare `optimizer.zero_grad()` prima di `loss_fn.backward()`.

Fase esistente	Effetto
Addestramento Modello	Soggetto a errori

ESEMPIO

Python

```
# 4. Train the network

for epoch in range(2): # training loop multiple times

running_loss = 0.0

for i, data in enumerate(trainloader, 0):

# get the inputs; data is a list of [inputs, labels]

inputs, labels = data

+ # zero the parameter gradients

+ optimizer.zero_grad()

# forward + backward + optimize

outputs = net(inputs)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()
```

IN-PLACE APIS MISUSED

CONTESTO

Le strutture dati possono essere manipolate in due modi principali:

1. Applicando le modifiche a una copia, lasciando intatto l'oggetto originale.
2. Modificando direttamente l'oggetto originale (in-place).

PROBLEMA

Alcuni metodi lavorano in-place, altri restituiscono una copia. Se uno sviluppatore assume erroneamente che l'operazione sia in-place e non assegna il risultato a una variabile, l'operazione non influirà sul risultato finale.

SOLUZIONE

Si suggerisce di verificare se il risultato di ogni chiamata API Pandas o Numpy viene raccolto in una variabile.

Fase esistente	Effetto
Pulizia dei Dati	Soggetto a errori

ESEMPIO

Python

```
### Pandas

import pandas as pd

df = pd.DataFrame([-1])

- df.abs()
```

```
+ df = df.abs()

### NumPy

import numpy as np

zhats = [2, 3, 1, 0]

- np.clip(zhats, -1, 1)
+ zhats = np.clip(zhats, -1, 1)
```

Esempi di API Pandas che NON operano in-place:

- `df.drop()`
- `df.rename()`
- `df.sort_values()`
- `df.fillna()`
- `df.replace()`
- `df.drop_duplicates()`
- `df.reset_index()`
- `df.set_index()`
- `df.groupby()`
- `df.agg()`
- `df.transform()`
- `df.abs()`
- `pd.merge()`
- `df.join()`
- `df.concat()`
- `df.pivot()`
- `df.melt()`
- `df.stack()`
- `df.unstack()`

Esempi di API NumPy che NON operano in-place:

- `np.copy()`
- `np.reshape()`
- `np.transpose()`

- `np.flatten()`
- `np.ravel()`
- `np.concatenate()`
- `np.stack()`
- `np.split()`
- `np.clip()`
- `np.add()`
- `np.subtract()`
- `np.multiply()`
- `np.divide()`
- `np.power()`
- `np.sqrt()`
- `np.exp()`
- `np.log()`
- `np.mean()`
- `np.median()`
- `np.std()`
- `np.var()`
- `np.sum()`
- `np.prod()`
- `np.min()`
- `np.max()`
- `np.sin()`
- `np.cos()`
- `np.tan()`
- `np.arcsin()`
- `np.arccos()`
- `np.arctan()`

PYTORCH CALL METHOD MISUSED

CONTESTO

In PyTorch, è possibile usare sia `self.net()` che `self.net.forward()` per eseguire la forward propagation.

PROBLEMA

Questi due approcci **non** sono equivalenti. `self.net()` gestisce anche i **register hooks**, mentre `self.net.forward()` li ignora.

SOLUZIONE

È raccomandato usare `self.net()` anziché `self.net.forward()`.

Fase esistente	Effetto
Addestramento Modello	Robustezza

ESEMPIO

Python

```
### PyTorch

# 1. Load and normalize CIFAR10

# 2. Define a Convolutional Neural Network

def forward(self, x):

- x = self.pool.forward(F.relu(self.conv1(x)))

+ x = self.pool(F.relu(self.conv1(x)))

x = self.pool(F.relu(self.conv2(x)))

x = torch.flatten(x, 1) # flatten all dimensions except batch

x = F.relu(self.fc1(x))

x = F.relu(self.fc2(x))

x = self.fc3(x)
```

```
return x
```

TENSORARRAY NOT USED

CONTESTO

Far crescere dinamicamente array in loop TensorFlow può essere problematico se non gestito correttamente. Usare `tf.constant()` porta a errori poiché i tensori creati con `tf.constant()` sono **immutabili**.

PROBLEMA

Usare `tf.constant()` nei loop per far crescere array comporta:

- Errore di modifica di tensori immutabili.

SOLUZIONE

Gli sviluppatori devono usare `tf.TensorArray()` anziché `tf.constant()` per array dinamici.

Fase esistente	Effetto
Addestramento Modello	Robustezza

ESEMPIO

Python

```
### TensorFlow
```

```
import tensorflow as tf

def fibonacci(n):

a = tf.constant(1)

b = tf.constant(1)

- c = tf.constant([1, 1])

+ c = tf.TensorArray(tf.int32, n)

+ c = c.write(0, a)

+ c = c.write(1, b)
```