Code Smells for Machine Learning Applications

Carolina Dias
Claudio Fortier

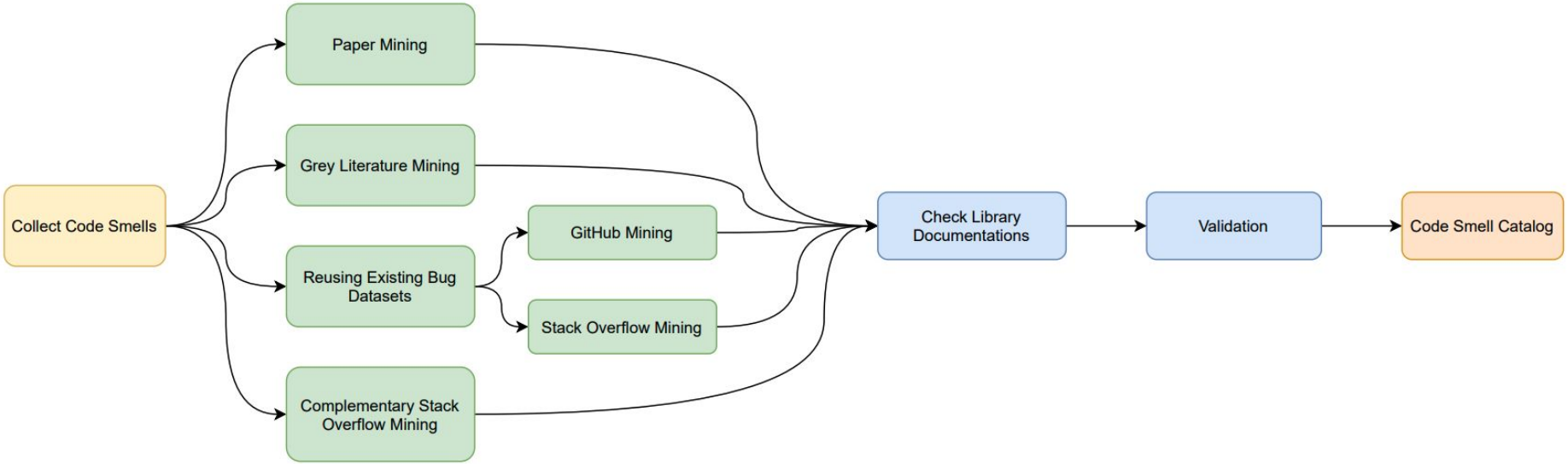
O que veremos hoje:

- Introdução
- Metodologia
- Resultados: Code Smells
- Implicações

Introdução

- Crescimento acelerado do uso de técnicas de Machine Learning na indústria e academia
- Falta de critérios e metodologias para a qualidade do código criado
- Cientistas de dados sem conhecimento adequado de engenharia de software
- Code Smells específicos de códigos de aprendizado de máquina
- A qualidade do código de aprendizado de máquina é mais difícil de avaliar e controlar
- Proposta: eliminar code smells e antipadrões de design
- Resultado: catálogo de code smells

Metodologia



Code Smell 01 - Iteração Desnecessária

- **Contexto:** Uso excessivo de loops ao tratar de dados utilizando Pandas
- **Problema:** Loops são lentos e geralmente desnecessários para realizar operações em linhas e colunas de arquivos de dados
- **Solução:** Utilizar uma função vetorizada ao invés de loops. Já existem várias funções substitutas aos loops na biblioteca Pandas, e essas soluções vetorizadas, além de serem mais rápidas, também geram código mais legível.

Code Smell 02 - Comparação de equivalência *NaN* mal utilizada

- **Contexto:** Realizar a comparação de valores nulos (*NaN*) se comporta diferente do que é esperado.
- **Problema:** *None* == *None* retorna *True*, mas *np.nan* == *np.nan* retorna *False* no Numpy.
- **Solução:** Evitar utilizar esse tipo de comparação ou saber bem quais os resultados que serão retornados.

Code Smell 03 - Indexação encadeada (Chain indexing)

- **Contexto:** Para a biblioteca Pandas, *df["one"]["two"]* e *df.loc[:,("one", "two")]* retornam o mesmo resultado. O primeiro caso é conhecido como indexação encadeada.
- **Problema:** A indexação encadeada leva a diminuição de performance do código, pois são duas operações em uma só, e diminui o entendimento do mesmo, podendo trazer resultados inesperados.
- **Solução:** Evitar o uso de indexação encadeada.

Code Smell 04 - Colunas e tipo dos dados não definidos explicitamente

- **Contexto:** Na biblioteca Pandas, ao importar dados, todas as colunas têm seu tipo selecionado por padrão.
- **Problema:** Nem sempre os tipos selecionados por padrão na biblioteca são os tipos reais dos dados. Por exemplo, 1.0 tem seu tipo como *float*, mas o Pandas lê como inteiro. Dependendo da aplicação, isso pode acarretar erros.
- **Solução:** Definir explicitamente o tipo de todas as colunas durante a importação dos dados.

Code Smell 05 - Erro de inicialização de coluna vazia

- **Contexto:** Necessidade de criar uma coluna vazia em uma tabela de dados.
- **Problema:** Utilizar uma string vazia ("") ou o número 0 para inicializar essa coluna não é a mesma coisa que inicializá-la com valores nulos do tipo *NaN*.
- **Solução:** Utilizar sempre valores nulos (*np.nan*) para inicializar as colunas, ao invés de valores de preenchimento.

Code Smell 06 - Parâmetro não explicitado para o merge de dados

- **Contexto:** Uso da função *df.merge()* do Pandas para juntar dois conjuntos de dados.
- **Problema:** Não explicitar como e onde deve ocorrer o merge (por exemplo, em quais colunas e se será um join pela esquerda ou direita) faz com que sempre seja usado o comportamento padrão que, muitas vezes, não é o que se quer no código.
- **Solução:** Sempre explicitar os parâmetros da função *df.merge()*.

Code Smell 07 - Mal uso do parâmetro *inplace* das funções

- **Contexto:** As estruturas de dados podem ser manipuladas de duas maneiras: 1) aplicando as mudanças em uma cópia dos dados e mantendo o original, ou 2) modificando a estrutura já existente (mudança inplace).
- **Problema:** Alguns métodos utilizam a mudança inplace como padrão, enquanto outros criam uma cópia dos dados. Não saber quais são quais métodos afeta o resultado final esperado.
- **Solução:** Conferir sempre qual é o padrão para o método utilizado.

Code Smell 08 - Erro de conversão de dataframes

- **Contexto:** Transformar um dataframe do Pandas em um array do Numpy pode ser feito utilizando *df.to_numpy()* ou *df.values()*.
- **Problema:** O valor retornado por *df.values()* é inconsistente e essa função está para ser deprecada.
- **Solução:** Utilizar sempre o *df.to_numpy()* para realizar essa conversão.

Code Smell 09 - Erro (semântico) de multiplicação de matrizes

- **Contexto:** Para multiplicar matrizes de duas dimensões, *np.matmul()* e *np.dot()* retornam o mesmo valor, que é também uma matriz.
- **Problema:** Matematicamente, np.dot() deveria retornar uma escalar, para manter a consistência semântica no código.
- **Solução:** Utilizar apenas *np.matmul()* para multiplicar matrizes de duas dimensões.

Code Smell 10 - Não redimensionamento dos dados antes de operações que necessitam do redimensionamento

- **Contexto:** O escalonamento das features é um modo de alinhar vários valores diferentes para uma mesma escala.
- **Problema:** Existem operações que devem ser feitas, obrigatoriamente, em dados já escalonados, como a PCA, SVM, etc. Caso os dados não estejam escalonados, o resultado será influenciado pela feature com maior valor.
- **Solução:** Checar sempre se está ocorrendo o escalonamento das features antes de utilizar operações que necessitem disso.

Code Smell 11 - Hiperparâmetro não definido explicitamente

- **Contexto:** Os hiperparâmetros são setados antes do processo de treinamento de um modelo e influenciam no comportamento do mesmo.
- **Problema:** Os hiperparâmetros padrões das bibliotecas podem não ser ideais para o conjunto de dados utilizado e podem não gerar um modelo satisfatório.
- **Solução:** Sempre setar explicitamente os hiperparâmetros utilizados para o treinamento do modelo.

Code Smell 12 - Memória não liberada

- **Contexto:** O treinamento consome muita memória e a memória é limitada.
- **Problema:** Se a memória lotar durante o treinamento o mesmo falhará.
- **Solução:** Utilizar recursos específicos das APIs utilizadas com a finalidade de liberar memória durante o processo de treinamento. Ex.: clear_session() em loops do TensorFlow e usar .detach() com Pytorch.

Code Smell 13 - Opção de algoritmo determinístico não utilizada

- **Contexto:** O uso de algoritmos determinísticos pode melhorar a reprodutibilidade.
- **Problema:** Algoritmos não determinísticos complicam a depuração por não produzir resultados repetíveis.
- **Solução:** Sempre que as bibliotecas permitirem, selecionar a opção de utilizar algoritmos determinísticos. Ex.: arch.use_-deterministic_algorithms(True) no PyTorch

Code Smell 14 - Aleatoriedade Descontrolada

- **Contexto:** Em alguns algoritmos, a aleatoriedade está inerentemente envolvida no processo de treinamento.
- **Problema:** Se a semente aleatória não for definida, o resultado será irreprodutível , o que aumenta o esforço de depuração. Além disso, será difícil replicar o estudo com base no anterior.
- **Solução:** Definir a semente aleatória global primeiro para obter resultados reproduzíveis

Code Smell 15 - Faltando a Máscara de Valor Inválido

- **Contexto:** No aprendizado profundo, o valor da variável muda durante o treinamento. A variável pode se tornar um valor inválido para outra operação neste processo.
- **Problema:** Não é simples detectar quando um variável de entrada reduz ao ponto de zerar causando erro na aplicação de funções como tf.log() durante o treinamento.
- **Solução:** Utilizar estratégias que evitem os valores inválidos. Ex.: tf.log(tf.clip_by_value(x,1e-10,1.0)). Neste caso, se x zerar será calculado log do valor mínimo 1e-10.

Code Smell 16 - Recurso de transmissão não usado

- **Contexto:** Bibliotecas de aprendizado profundo, como PyTorch e TensorFlow, suportam a operação de transmissão element-wise.
- **Problema:** Sem transmissão, colocar lado a lado um tensor primeiro para corresponder a outro tensor consome mais memória devido à criação e armazenamento de um resultado de operação de lado a lado intermediário.
- **Solução:** Utilizar a transmissão para otimizar o uso de memória.

Code Smell 17 - TensorArray não usado

- **Contexto:** Os desenvolvedores podem precisar alterar o valor da matriz nos loops do TensorFlow.
- **Problema:** Se o desenvolvedor inicializar um array usando tf.constant() e tentar atribuir um novo valor a ele no loop para mantê-lo crescendo, o código apresentará um erro.
- **Solução:** Usar tf.TensorArray() para aumentar o array no loop.

Code Smell 18 - Alternância imprópria do modo de treinamento/avaliação

- **Contexto:** No PyTorch, chamar .eval() significa que estamos entrando no modo de avaliação e a camada Dropout será desativada.
- **Problema:** Se o modo de treinamento não voltasse no tempo, a camada Dropout não seria usada em algum treinamento de dados e, portanto, afetaria o resultado do treinamento.
- **Solução:** chamar o modo de treinamento no local apropriado no código de aprendizado profundo para evitar o esquecimento de alternar o modo de treinamento novamente após a etapa de inferência.

Code Smell 19 - Método de chamada do Pytorch mal utilizado

- **Contexto:** Ambos self.net() e self.net.forward() podem ser usados para encaminhar a entrada para a rede no PyTorch.
- **Problema:** No PyTorch, self.net() e self.net.forward() não são idênticos. O self.net() também lida com todos os itens de registro, que não seriam considerados ao chamar o simples .forward().
- **Solução:** Usar self.net() em vez de self.net.forward().

Code Smell 20 - Gradientes não limpos antes da retropropagação

- **Contexto:** No PyTorch, optimizer.zero_grad() limpa os gradientes antigos da última etapa, loss_fn.backward() faz a propagação de volta e optimizer.step() executa atualização de peso usando os gradientes.
- **Problema:** Se optimizer.zero_grad() não for usado antes de loss_-fn.backward(), os gradientes serão acumulados de todas as chamadas loss_-fn.backward() e isso levará à explosão de gradiente, que falha no treinamento.
- **Solução:** Usar optimizer.zero_grad(), loss_fn.backward(), optimizer.step() juntos na ordem e usar optimizer.zero_grad() antes de loss_fn.backward().

Code Smell 21 - Vazamento de dados

- **Contexto:** O vazamento de dados ocorre quando os dados usados para treinar um modelo de aprendizado de máquina contêm informações de resultado de previsão.
- **Problema:** Vazamento de dados frequentemente leva a resultados experimentais excessivamente otimistas e baixo desempenho no uso do mundo real.
- **Solução:** Uma prática recomendada no Scikit- Learn é usar a API Pipeline() para evitar vazamento de dados.

Code Smell 22 - Validação dependente de limite

- **Contexto:** O desempenho do modelo de aprendizado de máquina pode ser medido por diferentes métricas, incluindo métricas dependentes de limite (por exemplo, F-measure) ou métricas independentes de limite (por exemplo, Área sob a curva (AUC)).
- **Problema:** Escolher um limite específico é complicado e pode levar a um resultado menos interpretável
- **Solução:** As métricas independentes de limite de solução são mais robustas e devem ser preferidas às métricas dependentes de limite.

Implicações da existência de um catálogo de code smells

- Para os cientistas de dados:
  - Diretrizes claras para verificação de qualidade do código de aprendizado de máquina
  - Entendimento inicial dos smells e referência para aprofundamento
  - Cientistas com pouca experiência em engenharia de software podem usar o catálogo para "escapar" de problemas e conhecer boas práticas.
- Para os desenvolvedores de bibliotecas de aprendizado de máquina:
  - Alguns smells decorrem do design das APIs, direcionando possíveis melhorias
  - A documentação das bibliotecas alertam para más práticas, mas isso não é suficiente para evitar que ocorram
- Para os desenvolvedores de ferramentas de análise de código:
  - Fornece um guia para problemas possivelmente detectáveis por meio de análise estática de código
  - Alerta para a necessidade de contextualização da análise, como, por exemplo, a análise do teste ser diferente da análise do código em produção.