UNIVERSIDADE FEDERAL DE SERGIPE

DEPARTAMENTO DE COMPUTAÇÃO

DOCENTE: HENDRIK TEIXEIRA MACEDO

RELATÓRIO DO PROJETO 02
IMPLEMENTAÇÃO DO N-GRAMAS

DISCENTES:

BRENO SILVA DO NASCIMENTO

DAVI ARAÚJO DO NASCIMENTO

LUCAS EMANUEL SIQUEIRA COSTA

PEDRO JOAQUIM SILVA SILVEIRA

WILLIAM GABRIEL YCKSON ARAÚJO BRAGA

São Cristóvão - SE

22/02/2025

INTRODUÇÃO

A modelagem estatística de linguagem baseada em n-gramas constitui uma das abordagens mais fundamentais e didáticas para compreensão de modelos probabilísticos aplicados a texto. Neste relatório, apresenta-se a implementação de um modelo de linguagem n-grama aplicado a código-fonte em C, bem como sua utilização em duas tarefas distintas: predição de próximos tokens e detecção de similaridade (plágio).

Um n-grama é definido como uma sequência contígua de *N* elementos (tokens) extraídos de um corpus textual. A ideia central do modelo consiste em estimar a probabilidade condicional de ocorrência de um token dado um histórico composto pelos *(N−1)* tokens anteriores. Formalmente, o modelo estima:

$$P(w_i | w_{i-(n-1)}, \ldots, w_{i-1})$$

utilizando Estimativa de Máxima Verossimilhança (Maximum Likelihood Estimation — MLE), baseada exclusivamente em contagens de co-ocorrência observadas no corpus de treinamento.

Diferentemente de aplicações tradicionais voltadas para linguagem natural, esta implementação opera sobre código-fonte, o que implica particularidades no processo de tokenização e pré-processamento. Operadores, delimitadores e símbolos sintáticos são tratados como tokens relevantes, preservando a estrutura léxica da linguagem C. O modelo é treinado a partir de amostras extraídas do dataset Hugging Face Hub — especificamente do conjunto *BigCode / The Stack*, permitindo que as probabilidades reflitam padrões reais de escrita de código.

Além da modelagem probabilística, o mesmo mecanismo de geração de n-gramas é reutilizado para cálculo de similaridade via Índice de Jaccard, tratando conjuntos de n-gramas como representações estruturais do texto. Dessa forma, o projeto demonstra como uma mesma base matemática pode sustentar aplicações distintas: modelagem de linguagem e detecção de similaridade estrutural.

Este relatório detalha os fundamentos teóricos do modelo, as decisões de projeto adotadas na implementação, as adaptações realizadas em relação ao pseudocódigo formal e a análise do comportamento prático do sistema. O objetivo é evidenciar tanto a correção conceitual quanto a coerência entre especificação algorítmica e implementação efetiva.

DETALHAMENTO DO PROBLEMA

A implementação de um modelo baseado em n-gramas neste projeto está orientada à resolução de dois problemas computacionais distintos, porém conceitualmente relacionados: detecção de plágio em código-fonte e recomendação (predição) de código. Ambos exploram padrões estatísticos extraídos de sequências de tokens, mas com objetivos e critérios de avaliação diferentes.

1. PLÁGIO EM CÓDIGO-FONTE

O plágio em ambientes acadêmicos e profissionais representa um problema recorrente, especialmente em disciplinas de programação. Diferentemente de textos em linguagem natural, o código-fonte possui uma estrutura sintática rígida, um vocabulário limitado e alta probabilidade de similaridade estrutural legítima. Isso torna a detecção de plágio mais complexa do que simplesmente comparar textos literais.

Natureza do problema

O desafio consiste em determinar se dois códigos compartilham estrutura lógica substancialmente semelhante, mesmo quando há alterações superficiais como em renomeação de variáveis, alteração de espaçamento ou formatação, pequenas modificações sintáticas e inserção ou remoção de comentários

Métodos ingênuos baseados em comparação textual direta (string matching) são facilmente burlados. Portanto, é necessário um mecanismo que capture padrões estruturais locais, e não apenas igualdade textual.

Uso de N-Gramas na Detecção

Neste projeto, o problema é modelado como um problema de similaridade entre conjuntos de n-gramas.

2. Cada código é transformado em:
3. Uma sequência de tokens;
4. Um conjunto de n-gramas;
5. Um conjunto matemático representando sua "impressão digital estrutural".

A similaridade é então calculada via Índice de Jaccard:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

onde:

A = conjunto de n-gramas do código 1

B = conjunto de n-gramas do código 2

Se dois códigos compartilham muitos n-gramas idênticos, é provável que eles possuam estrutura semelhante.

O problema, portanto, não é apenas comparar textos, mas comparar padrões locais de contexto, o que torna o método robusto contra alterações superficiais.

2. RECOMENDAÇÃO DE CÓDIGO (PREDIÇÃO DE TOKENS)

O segundo problema abordado é o da modelagem probabilística de código, com o objetivo de sugerir automaticamente o próximo token mais provável dado um contexto.

Natureza do problema

Dado um trecho parcial de código, deseja-se prever qual token tem maior probabilidade de ocorrer em seguida.

Exemplo conceitual:

```
int main ( )
{
   return
```

O modelo deve estimar qual token é mais provável após *return*.

Esse problema é central em ferramentas modernas de desenvolvimento assistido, como sistemas de autocompletar presentes em IDEs e modelos mais avançados como o GitHub Copilot.

Entretanto, o código possui dependências sintáticas rígidas, pequenos contextos podem ser insuficientes para capturar intenção semântica e o modelo n-grama considera apenas dependências locais (memória limitada a $n-1$ tokens).

Ainda assim, para padrões frequentes (como estruturas de função, loops e retornos), o modelo captura regularidades estatísticas significativas.

COMPARAÇÕES COM PSEUDO-CÓDIGO

O pseudo-código utilizado como base para a construção da parte solicitada do projeto (implementação da predição de tokens com recomendação de código) foi o fornecido pelo professor da disciplina, Hendrik Teixeira Macedo.

Dessa forma, esta seção então apresenta uma análise sistemática da correspondência entre o pseudocódigo formal do modelo n-grama e a implementação prática realizada. A comparação é estruturada por etapas do algoritmo para evidenciar equivalências funcionais, diferenças estruturais e justificativas técnicas.

Geração de N-Gramas

Especificação Formal:

```
for i = 1 to LENGTH(tokens) - n + 1 do
   window <- SUBSEQUENCE(tokens, i, i + n - 1)
```

A operação central consiste em extrair, para cada posição válida, uma subsequência contígua de tamanho *N*.

Implementação:

```
def ngrams(listOfWords, window=3):
    listNgrams = []
    for i in range(0, len(listOfWords) - window + 1):
        windowSlice = listOfWords[i:i+window]
        listNgrams.append(tuple(windowSlice))
    return listNgrams
```

Correspondência Conceitual:

Elemento	**Pseudo-código**	**Implementação**	**Equivalência**
Limite do laço	LENGTH(tokens) - n + 1	len(listOfWords) - window + 1	Idêntico
Extração da janela	SUBSEQUENCE(...)	tokens[i:i+window]	Idêntico
Armazenamento	Implícito	append(tuple(...))	Equivalente

A implementação reproduz exatamente a lógica matemática do pseudocódigo. A única diferença é sintática (indexação iniciando em 0 em Python).

Pré-processamento e tokenização.

Especificação Formal:

```
tokens <- PREPROCESS-AND-TOKENIZE(corpus)
```

A etapa envolve:

1. Normalização (ex.: lowercase)
2. Remoção ou tratamento de pontuação
3. Segmentação em tokens

Implementação:

```
def codeTokenizer(code):
    return re.findall(r"[\w']+|[.,!?;(){}\[\]=+-/*]", code)
```

No contexto de linguagem natural, o pseudocódigo assume normalização textual. Entretanto, para código-fonte, os operadores (+, =, {) são semanticamente relevantes, a sensibilidade a maiúsculas pode ser relevante (C é case-sensitive) e a estrutura léxica deve ser preservada.

Assim, a implementação adapta o conceito de *PREPROCESS* ao domínio de programação, preservando tokens sintáticos essenciais.

Treinamento do Modelo

Especificação Formal

```
function TRAIN-NGRAM-MODEL(corpus, n)
   model <- new NGRAM-MODEL
   tokens <- PREPROCESS-AND-TOKENIZE(corpus)
   tokens <- ADD-PADDING(tokens, n-1)

   for i = 1 to LENGTH(tokens) - n + 1 do
      window  <- SUBSEQUENCE(tokens, i, i + n - 1)
      history <- window[1 ... n-1]
      word    <- window[n]

      INCREMENT-COUNT(model.counts, history, word)
      INCREMENT-COUNT(model.context_totals, history)
```

Implementação

```
model = defaultdict(lambda: defaultdict(int))

for sample in ds:
   tokens = codeTokenizer(code)
   for i in range(len(tokens) - n + 1):
      history = tuple(tokens[i:i + n - 1])
      nextW   = tokens[i + n - 1]
      model[history][nextW] += 1
```

Diferenças Estruturais:

O total do contexto é obtido dinamicamente por:

```
sum(model[history].values())
```

Isso elimina redundância e mantém equivalência matemática:

$$P(w|h) = \frac{count(h, w)}{\sum_{w'} count(h,w')}$$

Além disso, o pseudo-código inclui uma função de ADD-PADDING. Na implementação, essa etapa não é aplicada. Os arquivos de código contêm grande número de tokens, então a ausência de padding afeta apenas os primeiros (n-1) n-gramas. O impacto estatístico é desprezível no corpus total.

Por fim, o pseudo-código assume um corpus abstrato. Na implementação, foram utilizadas amostras carregadas do Hugging Face Hub, especificamente do dataset *The Stack*, mantido pela iniciativa *BigCode*. Isso não altera a lógica algorítmica, apenas a origem dos dados.

Cálculo da Probabilidade

Especificação Formal:

```
function GET-PROBABILITY(model, word, history)
  numerator   <- GET-COUNT(model.counts, history, word)
  denominator <- GET-COUNT(model.context_totals, history)
  if denominator = 0 then return 0
  return numerator / denominator
```

Implementação:

O numerador é retornado diretamente:

```
predictions = model.get(currHistory, {})
```

O denominador é calculado na camada de visualização:

```
total = sum(count)
prob  = c / total
```

Equivalência Matemática:

A implementação realiza a divisão fora da função principal, mas o cálculo é idêntico:

$$P(w|h) = \frac{model[h][w]}{\sum model[h][w']}$$

A condição denominator = 0 é tratada implicitamente por:

```
model.get(currHistory, {})
```

Se o histórico não existir, retorna *conjunto vazio*, evitando divisão por zero.

Geração de Texto

Especificação Formal:

```
for i = 1 to length do
   next_word <- SAMPLE-FROM-DISTRIBUTION(...)
   APPEND(output, next_word)
   UPDATE(current_history, next_word)
```

Implementação:

```
currHistory = tuple(tokens[-(n-1):])
predictions = model.get(currHistory, {})
orderedWords = sorted(predictions.items(), key=lambda
item: item[1], reverse=True)
```

A seleção do próximo token é feita pelo usuário, entre os cinco mais prováveis. A janela deslizante é implementada implicitamente e garante atualização correta do histórico a cada nova predição.

RESULTADOS

A avaliação do sistema foi conduzida considerando separadamente as duas funcionalidades implementadas: (1) detecção de similaridade entre códigos e (2) predição de tokens para recomendação de código.

1. Resultados — Detecção de Plágio

A aplicação do índice de Jaccard sobre conjuntos de n-gramas demonstrou comportamento coerente com o esperado teoricamente.

1.1. Comparação sem Normalização

Quando aplicada diretamente sobre tokens brutos os códigos idênticos produziram similaridade próxima de 100%, pequenas alterações sintáticas (ex.: renomeação de variáveis) reduziram significativamente o índice. Além disso, mudanças estruturais maiores resultaram em queda acentuada da similaridade.

Observou-se que o método é altamente sensível a modificações superficiais quando não há normalização estrutural.

1.2. Comparação com Normalização via AST

Com a ativação da normalização utilizando tree-sitter, os resultados tornaram-se mais robustos, pois renomeações de variáveis não afetaram tanto como antes o índice de semelhança. Alterações cosméticas (espaçamento, comentários) foram completamente ignoradas.

A similaridade passou a refletir a estrutura lógica do programa, e não apenas a forma textual. Esse comportamento confirma que a normalização semântica reduz falsos negativos na detecção de plágio estrutural.

2. Resultados — Recomendação de Código

O modelo de linguagem n-grama foi treinado com amostras do dataset The Stack, disponível no Hugging Face Hub.

2.1. Coerência das Predições

Para contextos frequentes, como:

O modelo sugeriu tokens sintaticamente válidos, como:

"*0*", "*EXIT_SUCCESS*", "*;*".

O comportamento demonstra que o modelo capturou padrões estatísticos comuns na escrita de código C.

2.2. Limitações Observadas

Apesar da coerência local, observou-se a incapacidade de manter dependências de longo alcance, a ausência de compreensão semântica e predições limitadas ao contexto imediato (n−1 tokens).

Isso é consistente com a limitação teórica dos modelos n-grama, que assumem a hipótese de Markov de ordem fixa.

2.3. Impacto do Tamanho do Corpus

O aumento do número de amostras de treinamento resultou em uma maior diversidade de sugestões, distribuições de probabilidade mais estáveis e na redução de históricos não encontrados no modelo.

Entretanto, o custo computacional de treinamento cresce linearmente com o volume de dados processados.

CONCLUSÃO

Este trabalho demonstrou que modelos estatísticos baseados em n-gramas, apesar de sua simplicidade conceitual, são capazes de resolver problemas relevantes em análise e geração de código-fonte.

No contexto de detecção de plágio, a abordagem baseada em conjuntos de n-gramas e índice de Jaccard mostrou-se eficaz para identificar similaridade estrutural, especialmente quando combinada com normalização sintática via AST. A introdução da etapa de normalização elevou significativamente a robustez do método, aproximando a comparação da lógica do programa em vez de sua forma textual.

No contexto de recomendação de código, o modelo de linguagem implementado foi capaz de capturar regularidades estatísticas locais, produzindo sugestões coerentes para padrões comuns da linguagem C. Embora limitado por sua memória de curto alcance e ausência de modelagem semântica, o sistema ilustra claramente o princípio de estimativa de máxima verossimilhança aplicado à modelagem de sequência.