UNIVERSIDADE FEDERAL DO AMAZONAS

INSTITUTO DE COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

MÁRCIO PALHETA

ANÁLISE DE ESTILO PARA ATRIBUIÇÃO DE AUTORIA DE PROGRAMAS EM CURSOS DE PROGRAMAÇÃO

Manaus

Março de 2013

UNIVERSIDADE FEDERAL DO AMAZONAS

INSTITUTO DE COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

MÁRCIO PALHETA

ANÁLISE DE ESTILO PARA ATRIBUIÇÃO DE AUTORIA DE PROGRAMAS EM CURSOS DE PROGRAMAÇÃO

Dissertação apresentada ao Programa de Pós-Graduação em Informática do Instituto de Computação da Universidade Federal do Amazonas como requisito parcial para a obtenção do grau de Mestre em Informática.

ORIENTADOR: PROF. MARCO ANTÔNIO PINHEIRO DE CRISTO

Manaus

Março de 2013

"Sempre adiante; sempre mais longe; sempre mais alto."
(Léon Denis)

Resumo

A atribuição de autoria de código fonte consiste em identificar o autor de um determinado programa. Esta tarefa pode ser útil em análise forense, por exemplo, para apoio em cursos de programação ao auxiliar o professor a identificar estudantes que não produzem os códigos que fornecem. Os métodos de melhor desempenho para esta tarefa na literatura, em geral, usam uma estratédia de recuperação de informação. Nela, cada autor é representado por um conjunto de características de estilo de escrita, extraídas de seu histórico de programas. Assim, dado um programa-consulta, o seu autor deve ser o mesmo de outro programa que possui o conjunto de características mais similar ao da consulta. As abordagens tradicionais, contudo, não levam em conta alguns fatores particularmente comuns em ambiente acadêmico, ou seja, (a) códigos-fonte usados para determinar as características dos autores, em geral, resolvem os mesmos problemas ou problemas muito similares, o que aumenta a probabilidade de casamento entre o programas de diferentes autores que resolvem o mesmo problema; (b) estudantes, especialmente iniciantes, têm baixo nível de maturidade em programação, o que se reflete na falta de um estilo próprio e bem definido e (c) métodos de detecção de autoria, especialmente para alunos iniciantes, devem ser capazes de extrair características de históricos com poucos programas. Neste trabalho, propomos novos métodos de atribuição de autoria para ambientes acadêmicos que lidem melhor com tarefas similares bem como estudamos o impacto destes métodos em cenários caracterizados por estudantes inciantes e avançados e diferentes tamanhos de histórico de programas. Como resultado, obtivemos modelos probabilísticos, baseados em BM25, que alcançaram ganhos da ordem de 15% e 23% sobre o estado da arte nas coleções que usamos neste estudo. Resultados tão bons ou maiores foram obtidos por modelos de linguagem (21% e 23%). Também observamos que a coleção de autores iniciantes representa um desafio maior que a formada por autores avançados, bem como mostramos que os modelos têm bom desempenho mesmo para pequenos históricos de programas.

Palavras-chave: Atribuição de autoria, código fonte, recuperação da informação.

Abstract

The task of source-code authorship attribution consists in determining the author of a program. This task can be useful in forensic analysis, for instance, to assist teachers in programming courses to identify students that do not write the code they provide. The state-of-the-art methods found in literature to this task normally employ an Information Retrieval approach. That is, each author is represented by a set of writing style features, extracted from his program history. So, given a program, its author must be the same of another program that has most of the same features. Traditional approches, however, do not take into consideration factors very common in academic environments such as (a) source codes used to extract author features, in general, were developed to solve the same problem or very similar problems, which increases the probability of matching between programs from different authors written to solve the same tasks; (b) students, especially novices, have a low programming maturity level which leads to a lack of a well defined style. (c) methods for author atribution, especially for novices, must be able to extract features from very short program history logs. In this work, we propose new methods for author attribution for academic environments which deal with programs written to solve similar tasks and study the performance of these methods in scenarios characterized by novices and advanced students as well as short program logs. As a result of this research, we proposed probabilistic modelos, based on BM25, which outperformed a state-of-the-art approch with gains of about 15% and 23% in our test collections. Similar or even better results (21% and 23%) were achieved by language model approaches using only profile information. We also observed that source codes written by novices are harder to assign authorship than source codes provided by advanced authors. We have also shown that the proposed models present satisfatory performance even for history logs with few programs.

Keywords: Authorship attribution, source code, information retrieval.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Em linguística computacional e domínios de mineração de textos, segundo Kim et al. [2011], existem três problemas tradicionais de classificação: classificação de tópicos, classificação de gênero e classificação de autores, também conhecida como atribuição de autoria.

A atribuição de autoria é o processo de identicação do autor de um determinado documento, a partir de uma base conhecida de autores e suas obras, onde os documentos classificados podem ser representados por textos livres ou códigos fonte desenvolvidos por programadores. No entanto, Krsul & Spafford [1997] afirma que a classificação de autoria em textos livres é diferente da classificação de códigos fonte, pois a implementação de um programa deve obedecer a um conjunto rígido de regras de sintaxe, imposto pela linguagem de programação utilizada pelo autor. Além disso, em ambientes de programação, é comum o reuso de trechos de códigos entre programas, bem como o uso de funções de autoformatação que melhoram a legibilidade do texto. Com isso, é comum que códigos criados por autores diferentes apresentem os mesmos componentes de escrita, como indentação, laços de repetição, estruturas condicionais e palavras reservadas, tornando mais complexo o processo de busca por características que representem o estilo de escrita de um autor específico.

No processo de atribuição de autoria de códigos fonte, precisamos extrair caractrísticas de estilo de escrita dos autores, a partir de um conjunto conhecido de seus códigos. Assim, dado um novo *programa consulta*, buscamos o autor com características de estilo mais similares às características da consulta [Frantzeskou et al., 2006, 2007; Burrows et al., 2012]. Para Burrows et al. [2012], a escolha de dessas características de estilo ocorre a partir de dois conjuntos: (1) métricas de software, onde o estilo de escrita do autor é definido a partir de métricas de layout, estilo e estrutura do código, ou (2) combinações de sequencias de caracteres ou palavras contidos em um

programa. Após a definição do tipo de características, realizamos sua extração a partir de códigos fonte do autor.

Segundo Lange & Mancoridis [2007], a atribuição de autoria de códigos fonte pode ser aplicada à solução de problemas como: (1) Identificação de responsáveis por crimes virtuais, como roubo de senhas bancárias, a partir de um conjunto de suspeitos; (2) Detecção de infrações contratuais corporativas, onde uma empresa que suspeite da existência de códigos mal intencionados em seus sistemas pode utilizar a atribuição de autoria para identificar o funcionário responsável pela violação; (3) Aumento de produtividade na manutenção de sistemas legados, onde empresas desse segmento podem utilizar a atribuição de autoria para escolher, a partir do seu grupo de desenvolvedores, aquele com perfil mais adequado à manutenção de um determinado programa; e (4) Detecção de plágio, onde a identificação de autor pode ajudar a determinar se um código suspeito pertence ou não ao aluno que o submeteu.

Esta última aplicação pode ser particularmente útil para o auxílio de instrutores em cursos de programação, onde um grupo de aluno submete programas em resposta às tarefas propostas em classe.

1.1 Problema e Questões de Pesquisa

Nos trabalhos pesquisados, observamos que as coleções de códigos usadas são relativamente grandes e constituídas de códigos de domínios diversos, ou seja, os autores estavam resolvendo problemas distintos. Em cursos de programação, a realidade é diferente, pois a coleção de códigos de um autor costuma ser pequena e é criada ao longo da execução das atividades em classe. Além disso, é muito comum que vários alunos resolvam os mesmos problemas ou problemas muito similares. Ou seja, além de pequenos, os históricos de códigos de dois diferentes alunos podem ser compostos de códigos muito semelhantes. Outros fatores que contribuem para isto são a influência do estilo do instrutor sobre os alunos, bem como o uso de ferramentas de auto formatação em ambientes de desenvolvimento.

Um método de atribuição de autoria em ambiente acadêmico deveria, portanto, ser robusto o suficiente para lidar com programas escritos para resolver as mesmas tarefas ou tarefas similares, pequenas bases de programas e cenários onde alunos podem ser inexperientes e não ter estilo muito bem definido. Este não é o caso de métodos do estado-da-arte que em geral falham nestes cenários. Por exemplo, o método considerado estado-da-arte para atribuição de autoria em coleções acadêmicas foi proposto por Burrows et al. [2012]. Ao aplicarmos este método em nossas coleções de teste,

observamos que em cerca de 67% dos seus casos de erro, o autor atribuido corresponde ao de um programa que foi criado para resolver a mesma tarefa do programa alvo.

Em suma, considerando que queremos determinar os autores de programas, dada uma coleção de programas em que muitos foram criados para resolver as mesmas tarefas ou tarefas similares, gostaríamos de responder as seguintes questões de pesquisa:

- Em um cenário onde é possível agrupar programas de acordo com as tarefas que estes resolvem, a informação da tarefa pode ser usada para melhorar o processo de atribuição de autoria? Como? Nossa hipótese é que a informação de tarefa pode ser usada para aprimorar a estimativa da importância das características de estilo dos autores. Mais especificamente, características comuns em tarefas devem ser desenfatizadas.

- Em um cenário onde *não* é possível agrupar programas de acordo com as tarefas que estes resolvem, como produzir um método mais robusto de atribuição de autoria? Nossa hipótese é que métodos que evitem a comparação direta entre programas devem ser mais robustos uma vez que minimizam a possibilidade de comparação de programas criados para resolver as mesmas tarefas.

- Qual o desempenho dos métodos a serem propostos quando consideramos cenários em que os alunos são iniciantes e cenários em que eles são experientes? Nossa hipótese é que os métodos funcionem melhor em cenários com alunos experientes e que, possivelmente, tenham estilo de escrita mais bem definido.

- Qual o impacto do tamanho do histórico de programas do aluno na atribuição de autoria nos métodos propostos, considerando tanto o cenário envolvendo tanto alunos iniciantes quanto experientes?

1.2 Objetivos

1.2.1 Objetivo Geral

Desenvolver métodos de atribuição de autoria que sejam eficientes em ambientes acadêmicos caracterizados por pequenas bases de programas por autor, com muitos destes programas escritos para a resolução de tarefas comuns ou muito similares e cujos autores podem ser programadores iniciantes, sem estilo de escrita bem definido.

1.2.2 Objetivos Específicos

1. Implementar novos métodos para a atribuição de autoria que sejam robustos em ambientes acadêmicos caracterizados por pequenas bases de programas por autor, com muitos destes programas escritos para a resolução de tarefas comuns ou muito similares e cujos autores podem ser programadores iniciantes, sem estilo de escrita bem definido;

2. Comparar os vários métodos propostos e em uso na literatura, considerando tanto o cenário de sala de aula em que os alunos são iniciantes quanto o cenário em que eles são experientes;

3. Estudar o impacto do tamanho do histórico de programas do aluno na detecção de estilo, considerando tanto o cenário de sala de aula em que os alunos são iniciantes quanto o cenário em que eles são experientes;

1.3 Metodologia

A realização desta pesquisa se deu com a execução das seguintes tarefas:

1. Levantamento bibliográfico.

2. Obtenção de coleções para experimentação.

3. Seleção e implementação de método estado-da-arte para comparação.

4. Proposta e implementação de métodos para a atribuição de autoria.

5. Análise de resultados e conclusão.

A pesquisa foi concentrada no tema de atribuição de autoria em cursos de programação, como uma forma de minimizar problemas de plágio. A estratégia de atribuição de autoria foi preferida em relação à de detecção de plágio devido à necessidade desta última de comparação com o fonte do plágio. Nem sempre é fácil obter o fonte do plágio pois é comum o uso de fontes externas, como a Internet. Uma vez definido o foco, a primeira fase do trabalho consistiu no levantamento bibliográfico sobre o estado da arte relacionado ao problema de atribuição de autoria.

A segunda fase do trabalho consistiu em criar bases de referência para o problema de atribuição de autoria característico do ambiente de sala de aula. Ou seja, (1) há poucos programas por autor, (2) é comum que estes programas tenham sido escritos para a resolução de tarefas comuns ou muito similares e (3) os autores destes programas

podem apresentar diferentes níveis de maturidade em programação que, para fins deste trabalho, classificamos como iniciantes e avançados. Estas bases foram coletadas a partir de programas feitos por alunos em disciplinas de ensino de programação no primeiro (iniciantes) e no quarto ano (avançados) do curso de ciência da computação. Nas duas bases é possível determinar a tarefa correspondente aos programas entregues por todos os alunos. Ou seja, para cada codigo fonte, sabemos quem é o seu autor e a que tarefa o código resolve.

A terceira fase do projeto consistiu na implementação do método de atribuição de autoria proposto por Burrows et al. [2012]. Naquele trabalho, é realizada uma análise comparativa dos principais métodos de autoria de código fonte, onde são avaliados os trabalhos de nove grupos de pesquisa, utilizando o mesmo conjunto de coleções, considerando códigos acadêmicos e códigos profissionais. Como resultado, a abordagem adaptada de Burrows et al. [2009] apresenta melhor acurácia em bases de dados acadêmicas e o modelo de Frantzeskou et al. [2007] apresenta melhores resultados em coleções de programas profissionais. Com base no exposto, o modelo proposto por Burrows et al. [2009] e adaptado em Burrows et al. [2012] será utilizado como *baseline* deste trabalho. Sucintamente, este método consiste na representação de programas através de combinações de caracteres e palavras. O problema de atribuição de autoria é visto como um problema de recuperação de informação onde o autor do programa-consulta é o autor do programa mais similar à consulta.

A quarta fase correspondeu à proposição de novos métodos de atribuição. Neste trabalho, não focamos no problema de extração de características. Assim, usamos as mesmas características propostas em Burrows et al. [2012]. Nossa contribuição para o problema consistiu em propor novos modelos de *ranking* dos programas mais similares ao programa-consulta. Assim, utilizamos uma técnica de recuperação da informação, onde: (1) códigos conhecidos de cada autor são indexados como documentos do modelo, individualmente ou agrupados em um único perfil; (2) Códigos de autores desconhecidos são usados como consulta; (3) para cada nova consulta, o sistema gera uma lista de documentos/perfis ordenados por grau de similaridade; (4) o autor do primeiro documento da lista é devolvido como resposta do modelo.

Em particular, nossa contribuição está no passo (3), onde propusemos modelos baseados em duas hipóteses. A primeira é que a informação da tarefa pode ser usada para aprimorar a estimativa da importância das características de estilo dos autores, se esta informação for conhecida. A segunda é que métodos que evitem a comparação direta entre programas devem ser mais robustos, uma vez que minimizam a possibilidade de comparação de programas criados para resolver as mesmas tarefas.

Com base na primeira hipóstese, formulamos diversas variantes de um modelo probabilístico de recuperação de informação para testar diferentes formas de ponderação de características usando informação de tarefa (grupo de programas que resolvem o mesmo problema) e de perfil (grupo de programas de um mesmo autor). Modelos baseados nesta hipótese exigem que a informação de tarefa seja fornecida. Como nem sempre esta informação está disponível, com base na segunda hipóste, propusemos quatro modelos de recuperação de informação baseados em modelos geração de linguagem, que exploram estatísticas de perfil.

Os vários métodos propostos foram comparados com o método de Burrows et al. [2012], usando as coleções criadas que podem ser vistas como de autores iniciantes e avançados. Além disso, os métodos foram estudados considerando variações no número de programas usados como base histórica dos autores. Desta forma, medimos o impacto do número de programas na efetividade do processo de atribuição de autoria.

1.4 Resultados

Neste trabalho, formulamos modelos de busca que levam em conta informações de tarefas (quando disponíveis) e perfis de autores na estimativa da importância das características de estilo de programação. Como resultado, obtivemos modelos probabilísticos, baseados em BM25, que alcançaram ganhos da ordem de 15% e 23% sobre o método de Burrows et al. [2012] nas coleções que usamos neste estudo. Ganhos de 21% e 23% foram obtidos com métodos baseados em modelos de linguagem que exploram apenas a informação de perfil. Também observamos indícios de que a coleção de autores iniciantes representa um desafio maior que a formada por autores avançados, bem como mostramos que os modelos têm bom desempenho mesmo quando há poucos programas no perfil histórico do autor.

1.5 Estrutura da Dissertação

Além deste capítulo introdutório, este trabalho está organizado como segue. No Capítulo 2, descrevemos os conceitos básicos necessários para compreensão deste trabalho, bem como uma revisão da literatura. No Capítulo 3, apresentamos os vários modelos propostos. No Capítulo 4, apresentamos os experimentos realizados e os resultados obtidos. Finalmente, no Capítulo 5, apresentamos nossas conclusões e sugestões de próximos passos na pesquisa.

Capítulo 2

Fundamentos

Neste capítulo, apresentamos conceitos básicos para a compreensão dos métodos estudados, bem como uma revisão da literatura relacionada ao nosso problema. Os conceitos discutidos incluem apresentação de técnicas para definição de estilo de autoria, fundamentos de recuperação da informação e seus modelos tradicionais de ranking.

2.1 Atribuição de autoria

A ideia principal da atribuição de autoria é que, a partir de algumas características textuais, podemos distinguir entre textos de autores diferentes [Stamatatos, 2009].

Para Juola [2006], atribuição de autoria é um dos antigos e novos problemas de recuperação da informação e pode ser definida como a ciência de inferir características do autor, a partir das características dos documentos escritos por ele.

Krsul & Spafford [1997] afirma que a classificação de autoria pode ser aplicada a textos livres e códigos fonte. No entanto, a classificação de códigos fonte possui algumas particularidades, como: (1) a implementação de um programa deve obedecer a um conjunto rígido de regras de sintaxe, imposto pela linguagem de programação utilizada pelo autor; (2) em ambientes de programação, é comum o reuso de trechos de códigos entre programas; (3) autores diferentes podem utilizar funções de auto-formatação para melhorar a legibilidade do texto; e, em abimentes acadêmicos, (4) o estilo de programação de um aluno pode ser influenciado pelo estilo de programação do professor.

Para Krsul & Spafford [1997], quando um grupo de autores desenvolve programas utilizado a mesma linguagem de programação (ou linguagens com sintaxes similares), é comum que os códigos gerados apresentem os mesmos componentes de escrita, como indentação, laços de repetição, estruturas condicionais e palavras reservadas, tornando

mais complexo o processo de busca por características que representem o estilo de escrita de um autor específico. Na seção 2.1.1, apresentamos as características de estilo estudadas em nossa pesquisa.

2.1.1 Características de estilo

No processo de classificação de autoria, é importante determinar quais características de estilo de escrita de autores devem ser utilizadas, considerando o grau de importância de cada uma para o modelo de classificação.

Desde a decada de 90, a pesquisa em atribuição de autoria tem sido dominada por estudos que buscam definir características para quantificar o estilo de escrita de um autor [Stamatatos, 2009].

No trabalho de Krsul & Spafford [1997], o autor afirma que em atribuição de autoria de código fontes o estilo de escrita de um autor é definido por um conjunto distinto de caracteríscas presentes em seus códigos fonte.

Nesta seção, apresentamos as principais características de estilo utilizadas para atribuição de autoria de códigos fonte.

2.1.1.1 Métricas de software

Segundo Harold [1986], métricas de software são características que auxiliam à medição de aspectos do código fonte como exatidão e eficiência. No entanto, Krsul & Spafford [1997] afirma que métricas de software podem ser utilizadas para definir o estilo de programação de um autor, em uma linguagem específica. Ainda segundo Krsul & Spafford [1997], as métricas de software podem ser utilizadas para comparação de códigos escritos em liguagens semelhantes, como C e C++, sem representar uma boa alternativa para comparação entre linguagens que apresentem estruturas muito diferentes, como Java e Prolog.

Em seu trabalho, Lange & Mancoridis [2007] define um grupo de 18 de métricas de software, que são combinadas em um algorítmo genético, a fim de encontrar a combinação que gere melhor acurácia ao modelo. Dentre as métricas apresentadas, podemos citar o tamanho médio de variáveis, número médio de palavras por linha de código e o percentual de nomes de variáveis que contêm o caracter *underline*.

Em MacDonell et al. [1999], é definida uma coleção de 26 métricas de software para classificação de autoria, distribuídas em cinco grupos: (1) métricas de formatação do código fonte, considerando indentação e espaços em branco no início de cada linha de código, (2) métricas de estilo de comentário, onde são consideradas características como o tamanho dos comentários e os percentuais de comentários em bloco ou

linha, (3) métricas de uso de caracteres especiais, como *cifrão* e *underline*, (4) métricas de nomenclatura de variáveis e métodos, considerando percentuais de nomes de identificadores que contêm letras maiúsculas e (5) métricas de uso de características da linguagem, como a escolha entre *for*, *while* ou *reapeat..until* para a definição de laços de repetição.

Em Krsul & Spafford [1997], o autor apresenta uma taxonomia de 60 métricas, divididas em (1) métricas de *layout*, considerando o uso de espaços em branco, chaves e parênteses, (2) métricas de estilo de programação, considerando características como o tamanho médio de nomes de variáveis e comentários, e (3) métricas de estrutura de programação, onde são avaliadas características como tamanho médio dos métodos, tipos de retorno e estruturas de dados comumente utilizados.

2.1.1.2 N-gramas

Segundo Cavnar & Trenkle [1994], um $N-grama$ é a uma sequência de $N-caracteres$ consecutivos, que fazem parte de uma string maior, onde N indica o tamanho da substring gerada, sendo comum seu uso em classificação de textos de linguagem natural. Como exemplo, dada a palavra "BANCO" e substituíndo espaços em branco por "_" poderíamos gerar as seguintes sequencias de *2-gramas*: _B, BA, AN, NC, CO e O_. Para Cavnar & Trenkle [1994], o uso de perfis baseados em frequências de N-gramas oferece uma forma simples e confiável para categorização de documentos, permitindo que as combinações de N-gramas sejam aplicadas a diversas tarefas de classificação.

Outras abordagens de uso de N-gramas em textos de linguagem natural são propostas por Barrón-Cedeño et al. [2009], onde o autor utiliza sequências de quaisquer palavras consecutivas e por Lioma & Blanco [2007], onde são utilizadas apenas palavras de grupos como pronomes, verbos e adjetivos.

Contudo, Burrows et al. [2012] afirma que, em classificação de autoria de código fonte, as estratégias que utilizam N-gramas são apresentadas por (1) Frantzeskou et al. [2006], que utiliza sequências de caracteres, (2) Kolter & Maloof [2004], utilizando sequências de bytes dos arquivos executáveis e (3) Burrows et al. [2007], que utiliza sequências de *tokens*, como palavras reservadas e operadores.

2.2 Recuperação da Informação

Para Baeza-Yates & Ribeiro-Neto [1999], a *Recuperação da Informação* (RI) atua na representação, armazenamento e acesso a informações de itens, permitindo que usuários tenham acesso facilitado a informações relevantes aos seus propósitos. Manning et al.

[2008] afirma que, em geral, os itens buscados em um sistema de RI são documentos que possuem textos não-estruturados, armazenados em grandes coleções, fisicamente aramzenadas em disco. Segundo Manning et al. [2008], o termo *não-estruturados* é usado para indicar que os dados não possuem uma semântica facilmente interpretada por um sistema computacional, necessitando mecanismos para sua extração a partir dos documentos.

Nas seções a seguir, apresentamos a arquitetura e os principais componentes utilizados em sistemas de RI.

2.2.1 Máquina de busca

Segundo Baeza-Yates & Ribeiro-Neto [1999], máquina de busca é um sistema computacional que recebe uma string de consulta de um usuário e busca em uma base indexada os documentos mais relevates a essa consulta. Em Brin & Page [1998], é apresentado um protótipo de arquitetura distribuída para uma máquina de busca de larga escala, projetada a partir de um conjunto de componentes capazes de extrair e indexar documentos web de forma eficiente, além de melhorar a qualidade das respostas geradas. Essa arquitetura pode ser observada na figura 2.1.

Em nossa pesquisa voltada ao ambiente acadêmico, onde o volume de documentos é pequeno, quando comparado ao montante de documentos espalhados na web, adotamos uma arquitetura mais simples, conforme proposta de Baeza-Yates & Ribeiro-Neto [1999], apresentada na figura 2.2, e implementada pelos seguintes componentes:

1. *Interface* é o componente responsável pela interação entre usuário e sistema, utilizando técnicas de visualização para que o usuário possa informar ao sistema um *texto-consulta* e visualize como resposta uma lista com os documentos mais relevantes à consulta realizada.

   Segundo Baeza-Yates & Ribeiro-Neto [1999], após receber lista com os documentos mais relevantes à consulta, o módulo de interface pode iniciar o processo de *feedback* do usuário, registrando que documentos da lista de respostas foram selecionados (ou clicados), a fim mapear a relação entre consultas e respostas, incrementando a acurácia do sistema em futuras consultas similares.

2. *Processador de consultas* é o componente que recebe um conjunto de palavras do módulo de interface e procura na base de documentos aqueles com maior relevância em relação à string de busca recebida. Para definição da relevência de cada documento, o módulo processador percorre a base calculando a similaridade de cada documento em relação à consulta. Para isso, são utilizadas funções de

similaridade, conforme descrito nas seções 2.2.4.1 e 2.2.4.2. Como resultado do processamento, segundo Arasu et al. [2001], o módulo processador devolve à interface uma lista contendo o *ranking* ordenado dos documentos mais relevantes à consulta, onde o documento no topo do ranking é o documento mais relavante e o último documento é o menos relevante. Na seção 2.2.3 detalhamos as atividades envolvidas no processamento de consultas.

3. *Rastreador* (ou *crawler*) é o módulo responsável por percorre a web coletando páginas que, após os processos de extração e indexação, descritos na seção 2.2.2, se tornam documentos do sistema de RI. No entando, assim como em Burrows et al. [2012] e Frantzeskou et al. [2008], esse módulo não foi utilizado em nosso trabalho, uma vez que a matéria-prima para indexação de nossos documentos são códigos fonte de acadêmicos de cursos de computação, recebidos em resposta às tarefas passadas em sala de aula.

4. *Indexador* é o módulo que, segundo Witten et al. [1999], é resposnável pela extração de palavras e suas respectivas frequências em documentos baixados pelo *crawler*. Baeza-Yates & Ribeiro-Neto [1999] afirma que é papel do indexador organizar os dados extraídos em arquivos salvos em disco, de forma a garantir acesso rápido e organizado aos mesmos. Contudo, Brin & Page [1998] afirma que, em ambientes empresariais, o indexador precisa se comunicar com aplicações que armazenam dados em formatos distintos, como texto (Microsoft Outlook) e bases de dados (Oracle e MySQL).

5. *Índice* é o módulo que, segundo Baeza-Yates & Ribeiro-Neto [1999], é responsável pelo armazenamento e acesso eficiente aos dados de ocorrências de termos na coleção de documentos. Manning et al. [2008] afirma que o módulo de índice é geralmente de chamado de *índice invertido* ou *arquivo invertido*. Este componente é detalhado na seção 2.2.2

Nas seções a seguir, detalhamos os processos de indexação e processamento de consultas em uma máquina de busca.

2.2.2 Indexação

Segundo Baeza-Yates & Ribeiro-Neto [1999], o processo de indexação é responsável pela criação da estrutura base de uma máquina de buscas, geralmente chamda de *índice* ou *arquivo invertido*. Para criação dessa estrutura, o módulo indexador percorre

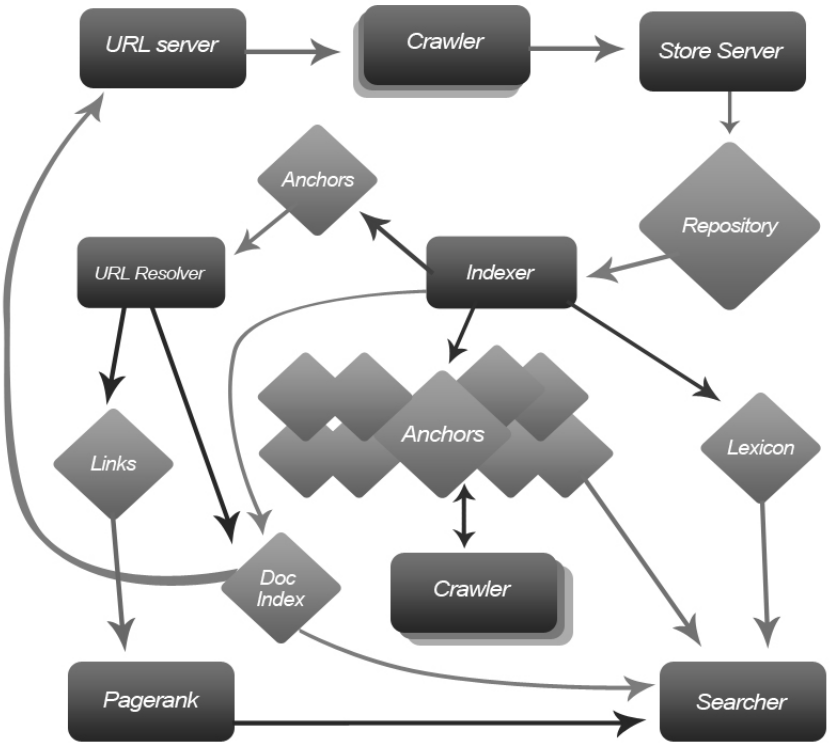


Figura 2.1: Arquitetura distribuída de máquina de busca em larga escala. Adaptado de Brin & Page [1998].

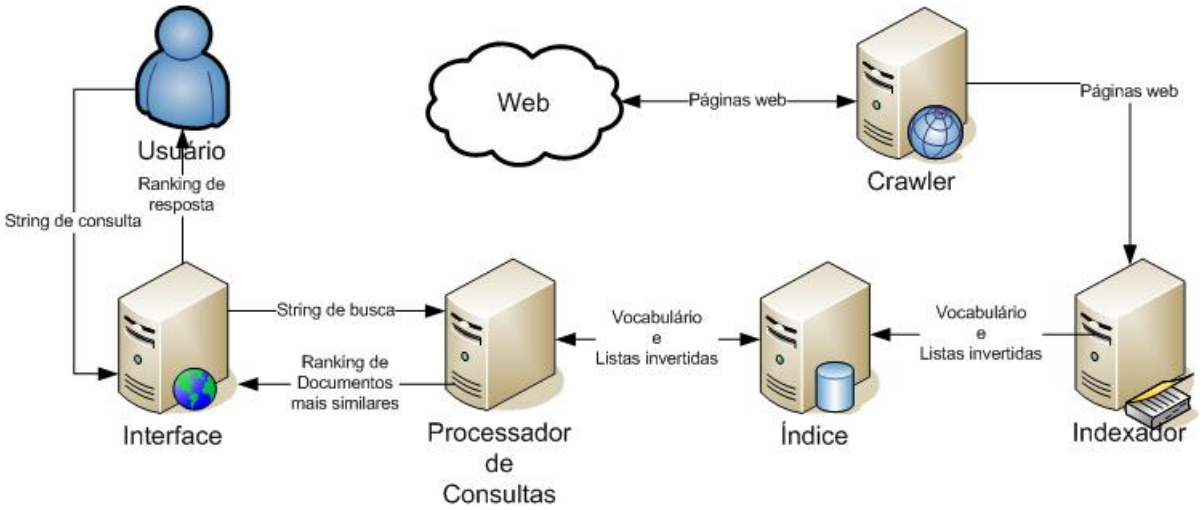


Figura 2.2: Arquitetura centralizada de um máquina de busca. Adaptado de Baeza-Yates & Ribeiro-Neto [1999].

os documentos coletados pelo *crawler* e extrai uma lista de termos e frequências de termos em documentos da coleção.

Durante o processo de indexação, são criados os dois principais componentes de um arquivo invertido: o *Vocabuário* e suas *Listas invertidas*. Para Baeza-Yates & Ribeiro-Neto [1999], o vocabulário de um sistema de RI contém uma coleção de todos os termos (palavras) encontrados na coleção de documentos e as listas invertidas indicam o número de ocorrências (frequência) de termos em cada documento.

A tabela 2.1 apresenta, como exemplo, uma coleção de documentos e seus respectivos termos, onde *D1* representa o documento de chave igual a 1, *D2* representa o documento de chave igual a 2 e assim sucessivamente. A coluna *Termos do documento* exibe o texto contido em cada documento.

Tabela 2.1: Coleção de documentos coletados

Documento	Termos do documento
D1	Abacate Abacate Abacate Banana
D2	Abacate Abacate Côco
D3	Abacate Abacate
D4	Banana Banana

A tabela 2.2 apresenta o resultado da indexação da base de documentos. Nessa tabela podemos verificar que o vocabulário do sistema de RI é composto pelos termos *Abacate*, *Banana* e *Côco*. A coluna *Listas invertidas* apresenta a frequência do termo em cada documento da coleção. Cada item de uma lista invertida é composto por uma combinação do tipo (d, tf), onde d indica a chave do documento e tf indica a frequência do termo no documento d. Com isso, podemos concluir que o termo *Abacate* ocorre 3 vezes no documento 1, 2 vezes no documento 2 e 2 vezes no documento 3.

Tabela 2.2: Resultado da indexação da coleção de documentos

Vocabulário	Listas invertidas
Abacate	(1,3) (2,2) (3,2)
Banana	(1,1) (4,2)
Côco	(2,1)

Para Manning et al. [2008], após a criação das listas invertidas, o indexador pode calcular estatísticas relacionadas aos termos do vocabulário e armazenar seus resultados em disco. Dentre as estatísticas de um termo, o indexador pode calcular a importância do um termo para a coleção de documentos (*idf*) ou, conforme proposto por Bahle

et al. [2002], processar informações sobre termos adjacentes, o que nos permite avaliar a sequência em que os termos de uma consulta ocorrem em um documento.

Anh & Moffat [2005] afirma que palavras que ocorrem em muitos documentos da coleção (conhecidas como *stopwords*) apresentam listas invertidas muito longas, o que aumenta o custo para o processamento de consultas que apresentem tais palavras. Assim, o sistema de RI precisa adotar estratégias para diminuição do impacto da presença de *stopwords* na coleção de documentos ou na consulta.

2.2.3 Processamento de consultas

O processamento de consultas em sistemas de RI inicia com uma requisição do usuário, onde é informada uma *string* de busca, contendo uma sequência de termos que representam o interesse do mesmo. Após o recebimento, o sistema extrai os termos existentes na string de busca executa uma função de similaridade, conforme descrito nas seções 2.2.4.1 e 2.2.4.2, a fim de gerar um ranking com os documentos mais relevantes à consulta.

Segundo Baeza-Yates & Ribeiro-Neto [1999], é comum que alguns documentos da base possuam mais termos pertencentes à consulta do que outros e que certos termos sejam mais importantes que outros, fazendo com que, em geral, funções de similaridade utilizem combinações de informações como: (1) importância do termo no documento (tf), onde o fato de um termo t ocorrer muitas vezes em um documento d indica que t é um termo importante para d; e (2) importância do termo na coleção, onde quanto mais raro for um termo na coleção de documentos, maior sua importância.

Baeza-Yates & Ribeiro-Neto [1999] define que sistemas de RI possuem duas abordagens tradicionais para percorrer a base indexada e calcular a similaridade dos documentos: (1) processamento *termo a termo*, onde o sistema percorre a lista invertida de cada termo t da consulta q, incrementando a similaridade dos documentos presentes na lista; e (2) processamento *documento a documento*, onde as listas invertidas de todos os termos de q são lidas simultaneamente, gerando a similaridade de apenas um documento por interação.

2.2.4 Ranking em Recuperação de Informação

Ao longo do tempo, foram desenvolvidas várias estratégias em Recuperação de Informação para ordenar os documentos de acordo com a sua relevância estimada para a consulta. Nesta seção, descrevemos dois modelos probabilísticos usados para esta tarefa. O primeiro é um dos mais usados em Recuperação de Informação e, em par-

ticular, foi adotado no método de atribuição de autoria proposto por Burrows et al. [2012]. O segundo é um modelo que tem atraído muita atenção nos últimos anos por ser fácil de estender.

2.2.4.1 Modelo Probabilístico Okapi BM25

Okapi BM25 é função de ranking baseada em um modelo probabilísto, usada para classificar documentos de acordo com suas relevâncias em relação a uma dada consulta. Nesse modelo, a relevância de um documento é calculada a partir da probabilidade de um documento d ser relevante para uma consulta q. [Jones et al., 2000a,b]

Dada uma consulta $q = q_1 q_2 ... q_n$ e um documento $d = d_1 d_2 ... d_m$, a similaridade de d em relação a q é dada pela equação 2.1:

$$Sim(d,q) = \sum_i IDF(q_i) * \frac{(k_1+1) * tf(q_i,d)}{K + tf(q_i,d)} * \frac{(k_3+1) * TF(q_i,d)}{k_3 + TF(q_i,d)}, onde \qquad (2.1)$$

$$IDF(q_i) = \ln\left(\frac{N - n(q_i) + 0.5}{n(q_i) + 0.5}\right), K = k1 * \left((1-b) + \frac{b * |d|}{|d_{avg}|}\right), e$$

- $TF(q_i, d)$ é a frequência do termo q_i no documento d;
- N é o número de documentos da coleção;
- $n(q_i)$ é o número de documentos da coleção em que q_i ocorre;
- $|d|$ é o tamanho do documento d;
- $|d_{avg}|$ é o tamanho médio de documentos da coleção;
- b, k_1 e k_3 são parâmetros configurados livremente que. Adotamos em nosso trabalho os valores padrões definidos em Burrows et al. [2012], com: $b = 0.75$, $k_1 = 1.2$ e $k_3 = 10^{10}$.

Em nossa proposta, vamos sugerir várias modificações nas estimativas das funções $IDF(q_i)$ e $TF(q_i, d)$. Em particular, para o $IDF(q_i)$ iremos sugerir diferentes valores para N e n. Para o $TF(q_i, d)$ iremos usar diferentes interpretações para o documento d.

2.2.4.2 Modelos de Linguagem

Nesta seção, apresentamos uma abordagem de modelos de linguagem aplicada à tarefa de ranking em recuperação de informação, com ênfase nas técnicas de suavização comparadas em [Zhai & Lafferty, 2004].

Nesta abordagem, nós assumimos que uma consulta é "gerada" por um modelo probabilístico baseado em um documento d. Dada uma consulta $q = q_1 q_2 ... q_n$ e um documento $d = d_1 d_2 ... d_m$, nós estamos interessados em estimar a probabilidade condicional $P(d|q)$, ou seja, a probabilidade de que d gere a consulta q observada. Após aplicar a regra de Bayes e eliminar a constante de normalização (dada que ela é irrelevante para o problema de ranking), obtemos:

$$P(d|q) \propto P(q|d)P(d) \tag{2.2}$$

Na equação 2.2, a probabilidade $P(d)$ é nossa crença a priori de que d é relevante para qualquer consulta e $P(q|d)$ é a verossimelhança da consulta dado o documento, o que captura o quão adequado o documento é para uma consulta q. No caso mais simples, podemos assumir $P(d)$ uniforme tal que ela não afeta o ranking. Com a hipótese de uniformidade, o problema de ordenação se resume à estimativa do seguinte modelo de unigramas:

$$P(d|q) = \prod_i P(q_i|d) \tag{2.3}$$

Note que a Eq. 2.3 pode ser re-escrita como:

$$\log P(d|q) = \sum_i \log P(q_i|d) \tag{2.4}$$

Um problema chave para estimar a probabilidade $P(q_i|d)$ é conhecido como *suavização*, ou seja, o ajuste do estimador de máxima verossimelhança de forma a corrigir erros resultantes da esparsidade dos dados. Muitas destas técnicas de suavização se baseiam em estratégias de interpolação onde são usadas duas distribuíções, um modelo usado para as palavras observadas e um modelo para as não observadas. Mais formalmente, podemos descrever esta ideia por meio da Eq 2.5.e

$$P(w|d) = \begin{cases} P_o(w|d), & \text{se palavra } w \text{ é observada em } d \\ \alpha_d P(w|\mathcal{C}), & \text{caso contrário} \end{cases} \tag{2.5}$$

Na equação 2.5, $P_o(w|d)$ é a probabilidade suavizada de uma palavra vista no documento, enquanto $\alpha_d P(w|\mathcal{C})$ corresponde à probabilidade suavizada de uma palavra

não vista no documento. Em particular, α_d é um coeficiente (dependente unicamente de d) que controla a massa de probabilidade associada com palavras não vistas, de forma que todas as probabilidades somem pra 1, e $P(w|\mathcal{C})$ é o modelo de linguagem da coleção (ou seja, a probabilidade da coleção gerar a palavra).

Como, em geral, a tarefa de ranking requer que computações sejam realizadas sobre uma grande coleção de documentos, é importante usar métodos de suavização eficientes. Em particular, nós usamos neste trabalho os métodos conhecidos como *suavização de Jelinek-Mercer* e *suavização Bayesiana com crença anterior de Dirichlet*. Para as dicussões que seguem, considere que a estimativa da probabilidade de máxima verossimelhança da palavra w, dado o documento d, é dada por:

$$P_{MV}(w|d) = \frac{c(w,d)}{\sum_w c(w,d)} \tag{2.6}$$

onde $c(w|d)$ é o número de vezes que a palavra w ocorre no documento d e $\sum_w c(w,d)$ é o número de palavras contidas em d.

O método de Jelinek-Mercer usa uma interpolação linear do modelo de máxima verossimelhança com o modelo da coleção, usando um coeficiente λ (um valor entre 0 e 1) para controlar a influência de cada modelo. Neste caso, o modelo para palavras vistas no documento é dado por:

$$P_o(w|d) = P_\lambda(w|d) = (1-\lambda)P_{MV}(w|d) + \lambda P(w|\mathcal{C}) \tag{2.7}$$

Para calcular o modelo para palavras não vistas, é suficiente determinar o valor de α_d que, neste caso, é dado por $\alpha_d = \lambda$. Neste trabalho, vamos chamar nossa implementação baseada em modelo de linguagem com suavização de Jelinek-Mercer de ML-λ.

A técnica de suavização Bayesiana, usando crença anterior de Dirichlet, assume que o modelo de linguagem segue uma distribuição multinomial para o qual a crença a priori para a análise Bayesiana é a distribuição de Dirichlet com parâmetros $(\mu P(w_1|\mathcal{C}), \mu P(w_2|\mathcal{C}), ..., \mu P(w_n|\mathcal{C}))$. Portanto, o modelo para as palavras observadas é dado por:

$$P_o(w|d) = P_\mu(w|d) = \frac{c(w,d) + \mu P(w|\mathcal{C})}{\sum_w c(w,d) + \mu} \tag{2.8}$$

Finalmente, para o caso das palavras não observadas, $\alpha_d = \frac{\mu}{\sum_w c(w,d)+\mu}$. Neste trabalho, vamos chamar nossa implementação baseada em modelo de linguagem com suavização de Dirichlet de ML-μ.

2.3 Atribuição de Autoria baseada em Recuperação de Informação

A fim de determinar o autor de um *código consulta*, os autores em Burrows & Tahaghoghi [2007] e Burrows et al. [2012] representam todos os códigos fonte conhecidos dos autores como documentos de um sistema de RI. Nessa abordagem, cada código fonte de um autor pode ser usado como consulta ao sistema de RI e esperamos que o topo do ranking de respostas seja composto por códigos produzidos pelo mesmo autor da consulta, conforme descrito na Seção 4.2.

Nesta seção descrevemos os processos de criação da estrutura de índices nesta estratégia, conforme definido em Burrows et al. [2012]. Mais especificamente, descrevemos as características usadas, o processo de extração (ou tokenização) dessas características e sua indexação.

2.3.1 Definição de Características

Em seu trabalho, Burrows & Tahaghoghi [2007] utilizam coleções de códigos fonte escritos em C, C++ e Java para identificar características que ajudem a identificar o estilo de programação de um autor. Em particular, eles declaram que a combinação das características *espaço em branco*, *operadores* e *palavras chave* melhora a acurácia de modelos de classificação de autoria baseada em RI.

No Anexo A, as Tabelas A.1 a A.6 apresentam as características de estilo definidas por Burrows & Tahaghoghi [2007] e utilizadas em nosso trabalho.

2.3.2 Extração de características de estilo

Na figura 2.3, apresentamos um exemplo de código fonte submetido por um aluno, em resposta a uma tarefa. Nesse exemplo, destacamos os comentários e strings que serão removidos do código, para fins de anonimização.

A fim de facilitar a visualização, apresentamos as características de estilo que podem ser extraídas da Figura 2.3 nas figuras seguintes. Em particular, a Figura 2.4 destaca os operadores e palavras-chaves e a Figura 2.5 destaca as sequências de espaços em branco.

A partir das características extraídas de um código fonte, realizamos a indexação do documento baseada em sequências de N-gramas.

```
1  int main(void)              /* Linked list driver.  */
2  {
3     IntList il;
4     int anInt;
5     ListMake(&il);
6     while (scanf("%d", &anInt) == 1)  /* All input.  */
7     {
8        if (!ListInsert(&il, anInt))
9        {
10          fprintf(stderr, "Error!\n");
11          break;
12       }
13    }
14    ListDisplay(&il);
15    ListFree(&il);
16    exit(EXIT_SUCCESS);
17 }
```

Figura 2.3: Código fonte oringinal, não indexado. Adaptado de Burrows & Tahaghoghi [2007].

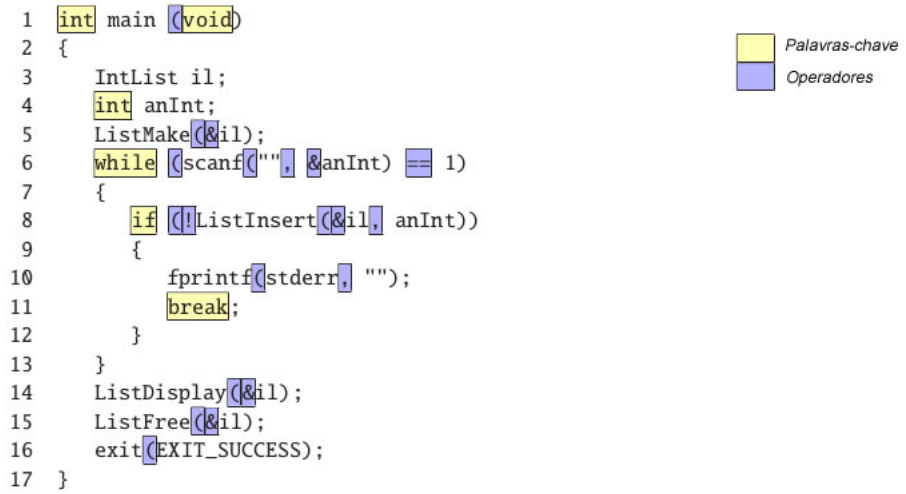


Figura 2.4: Definição de características para extração - Operadores e palavras-chave. Adaptado de Burrows & Tahaghoghi [2007].

2.3.3 Indexação baseada em N-gramas

Após a extração de características, realizamos o processo de indexação, que consiste em (1) converter todas as características em uma sequência de 2-caracteres; (2) concatenar todas as sequências de 2-caracteres em uma string S; (3) aplicar o algoritmo de *janela deslizante* à string S para gerar sequências de N-gramas, onde $N = 6$.

Ao final do processo, o código fonte original, submetido pelo autor, é convertido em um documento anonimizado, onde arranjos de 6-gramas de características de estilo representam *palavras* desse documento.

Na Figura 2.6, apresentamos o resumo do processo de indexação utilizado.

```
1  int main(void)
2  {
3      IntList il;
4      int anInt;
5      ListMake(&il);
6      while (scanf("", &anInt) == 1)
7      {
8          if (!ListInsert(&il, anInt))
9          {
10             fprintf(stderr, "");
11             break;
12         }
13     }
14     ListDisplay(&il);
15     ListFree(&il);
16     exit(EXIT_SUCCESS);
17 }
```

Figura 2.5: Definição de características para extração - Caracteres de espaço e nova linha. Adaptado de Burrows & Tahaghoghi [2007].

2.4 Avaliação

Neste trabalho, avaliamos o desempenho dos métodos propostos usando a métrica de Acurácia. Esta métrica é definida pela Eq. 2.9.

$$Acuracia = \frac{\text{Número de programas cuja a autoria é estimada corretamente}}{\text{Total de programas avaliados}} \tag{2.9}$$

Assim, avaliamos um método usando um conjunto de programas a serem consultados. Em nossos experimentos, o conjunto pode ser tanto formado pelo total de códigos na coleção (cf. Seções 4.3.1 e 4.3.2) quanto por uma amostra de códigos (cf. Seção 4.3.3). A avaliação usando amostras é conveniente para simular uma aplicação real, necessária na avaliação de tamanho de históricos.

Para estimar a confiabilidade estatística dos resultados, consideramos a acurácia como uma uma média tomada sobre N decisões binárias (1 para acerto na atribuição de autoria e 0 para erro), onde N é o total de programas avaliados. Assim, dados dois métodos, tomamos a acurácia média de cada um e calculamos o Teste-t, de Student, pareado e com duas caudas [Witten et al., 1999].

A opção pelo Teste-t se deve ao fato de que todas as nossas amostras terão um número razoável de consultas, o que dispensa a necessidade de testes de normalidade. Optamos pelo teste pareado porque a comparação dos métodos é dependente da consulta. Usamos o teste bi-caudal porque estamos interessados em estimar a confiabilidade da *diferença* entre os métodos. Ao longo deste trabalho, consideramos confiáveis

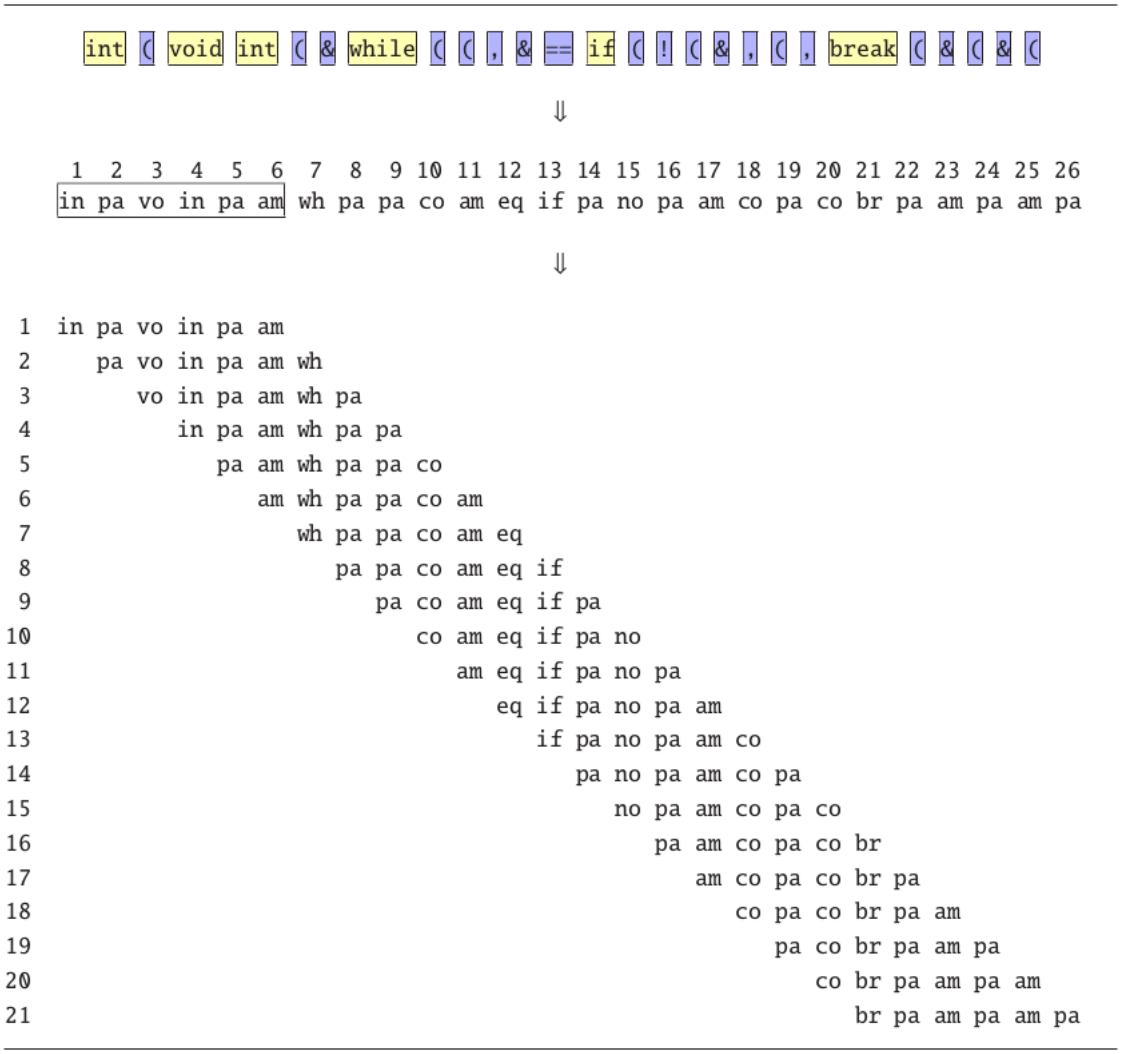


Figura 2.6: Indexação de documentos baseada em N-gramas. Adaptado de Burrows & Tahaghoghi [2007].

apenas resultados cujo o valor p do Teste-t for menor ou igual a 5% (grau de confiança de, pelo menos, 95%).

2.5 Trabalhos Relacionados

O problema de atribuição de autoria de código fonte é abordado na literatura de acordo com duas estratégias principais: classificação e busca. Em ambos os casos, os programas precisam ser representados através de atributos que podem ser agrupados em dois grupos distintos: baseados em métricas de escrita ou sequências de termos (n-gramas).

Exemplos de métodos que usam atributos baseados em métricas de escrita são os de Krsul & Spafford [1997] e de MacDonell et al. [1999]. No trabalho realizado em Krsul

& Spafford [1997], os autores estudam códigos fontes gerados em C e C++ e definem 49 métricas distintas para definição de estilo, separadas em três grupos: (1) métricas de layout de programação, onde são consideradas características como o percentual de vezes em que a abertura de chaves ocorre sozinha na linha e o percentual de linhas em branco no código; (2) métricas de estilo de programação, que incluem métricas como o tamanho médio das linhas do programa e o percentual de variáveis que iniciam com a letra maiúscula; (3) métricas de estrutura de programação, onde são avaliados itens como percentual de definição de funções que devolvem inteiros e o número médio de linhas por função.

O processo de classificação utiliza classificadores gaussianos e redes neurais. Em MacDonell et al. [1999], o autor apresenta uma coleção de 26 métricas para definição do perfil. Esse conjunto de métricas calcula itens como o número de ocorrências de uma determinada característica e o percentual de caracteres maiúsculos. A técnica de classificação modela o relacionamento entre uma série de variáveis independentes e uma ou mais variáveis dependentes.

Exemplos de métodos que usam n-gramas são os de Burrows et al. [2009] e o de Frantzeskou et al. [2007]. Estes dois métodos podem ser considerados métodos de busca, uma vez que tratam o programa a ter o autor atribuido como uma consulta. O autor do programa será definido de acordo com as respostas obtidas para a consulta.

Em Frantzeskou et al. [2007], o perfil de um determinado autor é criado a partir do ranque de frequências de n-gramas, onde n indica o número de caracteres da sequencia. O conjunto de programas conhecidos de um autor é dividido em bases de treino e teste. O conjunto de treino é concatenado em um único arquivo, de onde são extraídas as séries de n-gramas (para $n = 3$, exemplos de n-gramas seriam "_=_", "_i_" e "Usi") ordenadas pela frequência. Após a extração das sequencias de n-gramas, o perfil de um autor é definido por uma lista dos L n-gramas mais frequentes encontrados na sua base de treino. Dado um novo programa, calculamos seu perfil e a atribuição de autoria é feita ao autor com perfil que tenha maior numero de n-gramas comuns aos n-gramas do código consultado.

Em Burrows et al. [2009], a estratégia de tokenização também é utilizada para definição de perfis. No entanto, ao invés de considerar sequencias de caracteres consecutivos, o autor utiliza sequencias de palavras reservadas (como "void", "main" e "int"), operadores (como "!", "(" e "==") e espaços em branco, criando uma representação indexada de códigos de cada autor. Na abordagem proposta, cada programa de um autor conhecido é mapeado como um documento de um modelo de recuperação da informação. Nesse sentido, um código de autoria desconhecida é usado como consulta para o modelo, que utiliza a função de similaridade $OkapiBM25$ para gerar um ranque

contendo uma lista de documentos similares. O autor do documento que está no topo da lista é retornado como resposta do modelo.

Os trabalhos Burrows et al. [2009] e de Frantzeskou et al. [2007] são os mais próximos do nosso, no sentido em que também pretendemos usar um método baseado em busca. Em particular, o trabalho em Burrows et al. [2009] apresentou resultado superior a todos os demais e, comparado ao trabalho de Frantzeskou et al. [2007], foi especialmente melhor no cenário acadêmico. Por conta disso, usamos a última versão ques os autores propuseram (Burrows et al. [2012]) como nosso baseline de comparação neste trablho. Para uma revisão completa da literatura com comparação sistemática entre os vários métodos propostos, recomendamos ao leitor o trabalho de Burrows et al. [2012].

Nossas propostas se diferem das anteriores nos vários aspectos descritos a seguir. A partir do estudo realizado, identificamos que todos os métodos têm problemas com coleções de programas que foram escritos para resolver as mesmas tarefas ou tarefas similares. Eles usam sempre bases de programas onde são resolvidos problemas distintos. Em Lange & Mancoridis [2007], os autores informam explícitamente que as métricas adotadas por seus modelos podem apresentar resultados diferentes quando a base de perfis for composta por programas que resolvem os mesmos problemas. Este é um problema em cursos de programação, uma vez que vários alunos necessitem resolver o mesmo grupo de tarefas. Além disso, nenhum desses trabalhos estuda especificamente o impacto da maturidade dos alunos e do número de programas em seus históricos para a confiança dos métodos de atribuição de autoria. Essas informações são relevantes se pretendemos usar tais métodos nos primeiros períodos de cursos de programação, quando os alunos são imaturos e o histórico de códigos disponível é escasso.

2.6 Conclusões

Neste capítulo, apresentamos conceitos básicos para a compreensão dos métodos estudados, ou seja, (1) o problema de atribuição de autoria; (2) fundamentos de RI com destaque para os modelos de *ranking* probabilístico BM25 e baseados em geração de linguagem; (3) a estratédia de atribuição de autoria usando RI; e (4) estratégias de avaliação de métodos de atribuição de autoria. Finalmente, nós apresentamos uma revisão da literatura relacionada ao nosso problema, onde destacamos as nossas contribuições em relação aos trabalhos anteriormente propostos.

Capítulo 3

Atribuição de Autoria com Informação de Tarefa e Perfil

Neste capítulo, apresentamos os conjuntos de modelos e extensões de modelos que propomos para o problema de detecção de autoria. Nossa abordagem segue a ideia das estratégias mais bem sucedidas da literatura e trata este problema como um problema de *ranking* [Burrows et al., 2012; Frantzeskou et al., 2008]. Contudo, ao contrário destas abordagens, não focamos em métodos de seleção de características para a representação dos programas e sim no ranking dos programas. Nas próximas seções, descrevemos o método de detecção de autoria que empregamos e os modelos utilizados.

3.1 Atribuição de Autoria baseada em Busca

Para determinar o autor de um dado programa, usaremos a abordagem proposta na Seção 2.3, baseada em técnicas de recuperação da informação. Nesse modelo, uma coleção de treino é construída com todos os códigos conhecidos dos autores. Estes códigos são indexados em função de seus atributos. Eles podem ser indexados individualmente ou em conjunto. O conjunto de códigos de um autor é o que constitui o seu perfil. Códigos de autores desconhecidos são usados como consultas ao modelo, que gera uma lista de códigos ou perfis similares, ordenada pelo grau de similaridade entre o código-consulta e os códigos/perfis. O autor do código/perfil no topo da lista é retornado como resposta do modelo.

Neste trabalho, usamos como características os n-gramas de tokens, conforme a abordagem proposta em Burrows et al. [2012]. Em particular, coletamos os seguintes tokens: espaços em branco, operadores, e palavras-chave. Outra variável importante é o tamanho da sequência de palavras, ou seja, o valor de n. Como em Burrows et al.

[2012], estudamos variações do valor de n, o que nos levou a adotar $n = 6$.

Nossa contribuição para este problema está na forma como organizamos os códigos/perfis da coleção de treino. Uma hipótese em nosso trabalho é que o processo de *ranking* pode ser melhorado se algumas informações forem adicionadas diretamente à modelagem, em particular, informações indicando que tarefas os programas tentam resolver e que conjunto de programas pertence a cada autor (o que pode ser usado para definir o seu perfil de programação). Para testar essa hipótese, estendemos o modelo probabilístico BM-25 (cf. Seção 2.2.4.1) para incorporar as informações descritas e verificar seu impacto na tarefa de detecção de autoria. Este modelo é descrito na Seção 3.2.

Um inconveniente do primeiro conjunto de modelos é que ele requer que a informação sobre mesmas tarefas seja conhecida a priori. Como nem sempre essa informação está disponível, adotamos uma segunda estratégia de modelagem, baseada em modelos de linguagem (cf. Seção 2.2.4.2), em que incorporamos a ideia de perfil de programação à tarefa de detecção de autoria. A hipótese é que evitando a comparação direta entre programas, diminuimos a chance de atribuição para programas que resolvem a mesma tarefa. Este segundo conjunto de modelos é descrito na seção 3.3.

3.2 Detecção baseada em variantes do Modelo Probabilístico BM-25

Como visto na Seção 2.2.4.1, a importância de uma característica no modelo BM25 tradicional é dada por sua importância no documento (frequência do termo, TF) e a sua raridade na coleção (inverso da frequência na coleção, IDF). O modelo tradicional ignora que códigos possam ser agrupados em perfis e tarefas. Assim, nossa hipótese é que podemos incorporar tal informação de grupos ao modelo BM25 para melhorar a estimativa da importância do termo.

Intuitivamente, dado um programa relacionado a uma tarefa t, acreditamos que uma boa característica de estilo de um autor é uma comum no seu perfil e rara no perfil de outros autores e também na tarefa t. Se ela for comum na tarefa t, essa característica pode estar mais associada ao problema sendo resolvido que ao estilo do autor.

Para modificar o BM25 para testar essa hipótese, consideramos que a importância do termo w no documento deve considerar tanto sua frequência em códigos individuais d ($TF(w, d)$) quanto no perfil completo p ($TF(w, p)$). Da mesma forma, a importância do termo na coleção deve considerar as suas frequências na coleção de códigos ($idfd$),

na coleção de perfis ($idfp$) e na coleção de tarefas ($idft$). Note ainda que podemos considerar a coleção alvo formada por códigos ou perfis.

Assim, a Tabela 3.1 sumariza as quatro estratégias de busca possíveis se considerarmos apenas a importância do termo no documento e o tipo de documento considerado (código ou perfil).

Tabela 3.1: Estratégias de busca possíveis com BM25 considerando importância do termo tf e o tipo de documento

Estratégia	TF	Documento	Descrição
tfd/C	$TF(w,d)$	Código	Importância do termo no código usado para busca em códigos Esta é a estratégia usada por Burrows et al. [2009]
tfd/P	$TF(w,d)$	Perfil	Importância do termo no código usado para busca em perfis
tfp/C	$TF(w,p)$	Código	Importância do termo no perfil usado para busca em códigos
tfp/P	$TF(w,p)$	Perfil	Importância do termo no perfil usado para busca em perfis

No caso da importância do termo pra coleção devemos considerar a combinação de sua importância em três conjuntos distintos: os códigos ($idfd$), os perfis ($idfp$) e as tarefas ($idft$). Para combinar estas três estimativas em uma única estimativa de idf, nós usamos operadores tradicionalmente usados em probabilidade, ou seja, a conjunção (produto), a disjunção (soma), a disjunção com ruido, o máximo valor e o mínimo valor. Considerando estas variáveis e operadores combinatórios, temos na tabela 3.2 as combinações possíveis que serão avaliadas neste trabalho.

Considerando todas as estratégias descritas na Tabela 3.1 e os $idfs$ da Tabela 3.2, iremos avaliar 84 variantes do BM25 com o intuito de determinar se as informações de perfil e tarefa podem ser úteis para melhorar a atribuição de autoria em nossas bases acadêmicas. A seguir, descrevemos modelos de linguagem que consideram que a informação de tarefa não é disponível.

3.3 Detecção baseada em perfis usando Modelos de Linguagem

Uma das nossas estratégias para determinar o autor de um programa usa uma abordagem baseada em modelos de linguagem (cf. Seção 2.2.4.2). Neste caso, nós combinamos os programas de um mesmo autor e os tratamos como se fossem um único grande documento que chamamos de perfil. As ideias que apresentamos aqui são inspiradas nos modelos de recuperação baseada em grupos apresentados em Liu & Croft [2004]. Note que, ao contrário daquele trabalho, não estamos interessados em técnicas de agrupamento e suportamos duas variantes por modelo, de acordo com as técnicas

Tabela 3.2: Combinações heurísticas para cálculo da relevância de termos na coleção (*idf*)

Combinação de IDF's	Descrição
idfd	raridade na coleção de documentos (usado em Burrows et al. [2009])
idfp	raridade na coleção de perfis
idft	raridade na tarefa correspondente
idfd*idfp	conjunção de idfd e idfp
idfd*idft	conjunção de idfd e idft
idfp*idft	conjunção de idfp e idft
idfd*idfp*idft	conjunção de idfd, idft e idfp
idfd+idfp	disjunção de idfd e idfp
idfd+idft	disjunção de idfd e idft
idfp+idft	disjunção de idft e idfp
idfd+idfp+idft	disjunção de idfd, idft e idfp
max(idfd,idfp)	maior entre idfd e idfp
max(idfd,idft)	maior entre idfd e idft
max(idfp,idft)	maior entre idft e idfp
min(idfd,idfp)	menor entre idfd e idfp
min(idfd,idft)	menor entre idfd e idft
min(idfp,idft)	menor entre idft e idfp
1-(1-idfd)*(1-idfp)	disjunção com ruido entre idfd e idfp
1-(1-idfd)*(1-idft)	disjunção com ruido entre idfd e idft
1-(1-idfp)*(1-idft)	disjunção com ruido entre idft e idfp
1-(1-idfd)*(1-idfp)*(1-idft)	disjunção com ruido entre idfd, idft e idfp

de suavização aplicadas. Nesta seção, apresentamos as duas estratégias de modelagem aplicadas, às quais chamamos de Modelagem de Perfil e Modelagem de Programa, e suas variantes.

3.3.1 Modelagem de Perfil

A ideia neste caso é construir modelos para os perfis de programação dos autores e então recuperar/rankear os perfis com base na possibilidade do perfil $\mathcal{P}$ gerar o programa q. Em outras palavras, estamos interessados em determinar $P(q|\mathcal{P})$. Para tanto, estimamos $P(q|\mathcal{P})$ como fizemos para a Eq 2.4, o que se traduz como:

$$\log P(q|\mathcal{P}) = \sum_i \log P(q_i|\mathcal{P}) \tag{3.1}$$

onde $P(q_i|\mathcal{P})$ é especificado pelo modelo de linguagem:

$$P(w|\mathcal{P}) = \begin{cases} P_o(w|\mathcal{P}), & \text{se } w \text{ é observada no perfil } \mathcal{P} \\ \alpha_d P(w|\mathcal{C}), & \text{caso contrário} \end{cases} \tag{3.2}$$

Para a suavização de Jelinek-Mercer, temos que $P_o(w|\mathcal{P})$ é dado por:

$$P_o(w|\mathcal{P}) = P_\lambda(w|\mathcal{P}) = (1-\lambda)P_{MV}(w|\mathcal{P}) + \lambda P(w|\mathcal{C}) \tag{3.3}$$

onde:

$$P_{MV}(w|\mathcal{P}) = \frac{c(w,\mathcal{P})}{\sum_w c(w,\mathcal{P})} \tag{3.4}$$

No caso deste método, para o modelo de palavras não observadas, consideramos $\alpha_d = \lambda$, como visto anteriormente. Neste trabalho, chamamos este primeiro modelo de PERFIL-λ.

No caso da suavização Bayesiana com crença prévia de Dirichlet, o modelo é dado por:

$$P_o(w|\mathcal{P}) = P_\mu(w|\mathcal{P}) = \frac{c(w,\mathcal{P}) + \mu P(w|\mathcal{C})}{\sum_w c(w,\mathcal{P}) + \mu} \tag{3.5}$$

Neste caso, consideramos $\alpha_d = \frac{\mu}{\sum_w c(w,\mathcal{P})+\mu}$ para o cálculo do modelo de palavras não observadas. Neste trabalho, chamamos este segundo modelo de PERFIL-μ.

Finalmente, o autor do programa q de acordo com PERFIL-λ e PERFIL-μ é o autor associado ao perfil $\mathcal{P}$.

3.3.2 Modelagem de Programa

O Modelo de Programa recupera/rankeia o programa d de um autor que mais se assemelha ao programa q, baseado na possibilidade das palavras no programa d, pertencente ao perfil do autor, gerarem o programa q. Ou seja, estamos interessados em estimar $P(q|d)$. Em suma, este modelo suaviza as representações dos programas individuais usando os modelos dos perfis aos quais eles pertencem. Para tanto, o modelo é formulado para suportar duas suavizações, como segue:

$$P(w|d) = P_\beta(w|d) = (1-\beta)P_{MV}(w|d) + \beta P(w|\mathcal{P}) \tag{3.6}$$

onde β é o coeficiente de influência em uma suavização Jelinek-Mercer. Por sua vez, $P(w|\mathcal{P})$ é estimado usando a Eq. 3.2, podendo ser suavizado tanto usando Jelinek-Mercer (Eq. 3.3) quanto suavização Bayesiana com crença prévia de Dirichlet (Eq. 3.5). Ou seja, neste caso, há sempre dois parâmetros a estimar: β na Eq 3.6 e λ na Eq 3.3 ou μ na Eq 3.5.

Assim, primeiro aplicamos a suavização de Jelinek-Mercer (parâmetro λ) ou a suavização Bayesiana com crença prévia de Dirichlet (parâmetro μ) ao modelo de perfil

e, então, o modelo de programa é suavizado com o modelo de perfil, usando Jelinek-Mercer (parâmetro β). Note que o modelo de linguagem de programa é suavizado apenas no segundo estágio. Chamamos de PROGRAMA-λ ao modelo em que aplicamos Jelinek-Mercer no primeiro estágio. Da mesma forma, chamamos PROGRAMA-μ o modelo em que aplicamos suavização Bayesiana com crença prévia de Dirichlet no primeiro estágio.

Embora várias estratégias possam ser usadas para estimar o autor em PROGRAMA-λ e PROGRAMA-μ como, por exemplo, os k vizinhos mais próximos, optamos pela abordagem usada por Burrows et al. [2012], dada a sua simplicidade. Assim, o autor do programa q de acordo com PROGRAMA-λ e PROGRAMA-μ é simplesmente o autor do programa d, para o qual $P(q|d)$ é máximo.

3.4 Conclusões

Neste capítulo, apresentamos nossas abordagens para classificação de autoria em cursos de programação, considerando informações de tarefas e perfis, além de sugerir que a classificação de códigos fonte pode ser realizada apartir de coleções de documentos (estado-da-arte) ou perfis (onde os documentos são agrupados por autor). Em nossa primeira abordagem, assumimos que informações de tarefas e perfis são conhecidas e, com isso, propomos variações da função de similaridade Okapi BM25, a fim avaliar a importância de informações de tarefas e perfis em modelos em classificação de códigos fonte. Em nossa segunda abordagem, assumimos que informações de tarefa podem não ser conhecidas e, com isso, propomos o uso de modelos de linguagem baseados informações de documentos (códigos fonte) e perfis (códigos fonte agrupados por autor).

Capítulo 4

Experimentos

Neste capítulo, avaliamos os modelos que propusemos através de uma série de experimentos. Para tanto, apresentamos as coleções usadas nestes experimentos, nossa metodologia de experimentação e os experimentos realizados. Estes têm por objetivo (a) avaliar o impacto do uso de informações sobre tarefas e perfis em variantes do modelo BM25, (b) avaliar os modelos baseados linguagens propostos para usar apenas informação de perfil e (c) estudar o impacto do tamanho do histórico de programas na atribuição de autoria.

4.1 Coleções

Para avaliar nossa hipótese neste trabalho, usamos duas coleções de códigos fonte, coletadas em cursos de programação de instituições de ensino superior, conforme descrito a seguir:

- INICIANTE: composta por códigos de 31 alunos de uma faculdade particular, participantes da disciplina de Estrutura de Dados, que desenvolveram códigos para responder uma lista de exercícios com 36 questões. A base tem ao todo 918 programas. Os autores dessa base são, em sua maioria, alunos iniciantes do curso de computação, com pouca experiência em programação, que podem ter copiado os códigos de outros alunos ou trabalhado em pares.

- AVANÇADO: composta por código de 8 alunos da Universidade de São Paulo, participantes voluntários da turma de preparação para maratona de programação. Os códigos dessa base foram gerados para resolver dez listas de exercícios com um total de 40 questões diferentes. A base tem ao todo 270 programas. Os

autores dessa base são alunos veteranos e possuem um estilo de programação bem formado.

Note que ambas as coleções foram filtradas para para eliminar o conteúdo dos comentários dos programas. Esta é uma tarefa de pré-processamento comumente empregada em problemas de atribuição de autoria, uma vez que o problema pode se tornar trivial se comentários indentificarem os autores.

4.2 Metodologia

Nossos experimentos foram realizados a partir das coleções INICIANTE e AVANÇADO, (cf. Seção 4.1). A cada coleção, aplicamos os processos de extração de características (cf. Seção 2.3.2) e indexação (cf. Seção 2.3.3), gerando os documentos do sistema de RI, onde cada termo é uma sequencia de 6-gramas.

Para avaliação dos modelos propostos (cf. Seções 4.3.1, 4.3.2 e 4.3.3), cada documento da base foi utilizado como consulta à coleção. Nesse caso, o documento-consulta foi antes filtrado da coleção. Para avaliação dos modelos ao longo do tempo (cf. Seção 4.3.3), organizamos os documentos em grupos de tarefas, de acordo com as características de cada base: (1) na base INICIANTE não temos a informação de dia da tarefa, o que nos levou a dividir as 36 tarefas em 9 grupos de 4 tarefas, com média de 102 programas por grupo; (2) na base AVANÇADO, agrupamos as tarefas por dia de apresentação da tarefa, gerando 12 grupos com cerca de 3 tarefas, ou seja, com média de $22,5$ programas por grupo. Nesse experimento, a base inicia com um grupo de tarefas e, a cada iteração, um novo grupo é adicionado.

Todos os métodos foram avaliados usando a métrica de acurácia com a confiabilidade dos ganhos medida usando o Teste-t, conforme descrito na seção 2.4.

4.3 Resultados

Nesta seção apresentamos os resultados obtidos, considerando três conjuntos de experimentos. O primeiro conjunto de experimentos têm por objetivos avaliar o impacto do uso de informações sobre a raridade das características nas tarefas e perfis em variantes do modelo BM25. O segundo conjunto procura avaliar nossos modelos baseados linguagens usando apenas informação de perfil. O terceiro conjunto de experimentos visa determinar o impacto do tamanho do histórico de programas na tarefa de atribuição de autoria.

4.3.1 Modelos baseados em BM-25

As Tabelas 4.1 e 4.1 apresentam os resultados obtidos com as variantes estudadas do BM25 nas coleções INICIANTE e AVANÇADO, respectivamente. Nestes primeiros experimentos, consideramos que a informação de tarefa é conhecida, ou seja, nós sabemos determinar precisamente a que tarefa corresponde um certo código.

Em ambas as tabelas, as colunas representam as quatro estratégias de busca descritas na Tabela 3.1 e as linhas representam as diversas estimativas de *idf* que estudadas, conforme descrito anteriormente na Tabela 3.2. Para ambos os casos, tomamos como baseline a estratégia tfd/C combinada com idfd (ou seja, pesos TF-IDF calculados sobre códigos individuais com busca em uma coleção de códigos), o que corresponde ao método proposto por Burrows et al. [2009].

Tabela 4.1: Variantes do método BM25 na coleção INICIANTE

Combinação de IDF's	tfd/C	Ganho(%)	tfd/P	Ganho(%)	tfp/C	Ganho(%)	tfp/P	Ganho(%)
idfd	0,488	-	0,275	-43,65*	0,488	0,00	0,282	-42,21*
idfp	0,502	2,87	0,396	-18,85*	0,529	8,40*	0,405	-17,01*
idft	0,518	6,15*	0,262	-46,31*	0,541	10,86*	0,272	-44,26*
idfd*idfp	0,474	-2,87	0,501	2,66	0,505	3,48	0,502	2,87
idfd*idft	0,508	4,10*	0,421	-13,73*	0,525	7,58*	0,432	-11,48*
idfp*idft	0,511	4,71*	0,502	2,87	0,546	**11,89***	0,490	0,41
idfd*idfp*idft	0,488	0,00	0,556	**13,93***	0,525	7,58*	0,559	**14,55***
idfd+idfp	0,492	0,82	0,300	-38,52*	0,505	3,48	0,303	-37,91*
idfd+idft	0,497	1,84	0,273	-44,06*	0,511	4,71*	0,278	-43,03*
idfp+idft	0,518	6,15*	0,296	-39,34*	0,537	10,04*	0,299	-38,73*
idfd+idfp+idft	0,501	2,66*	0,287	-41,19*	0,516	5,74*	0,293	-39,96*
max(idfd,idfp)	0,488	0,00	0,275	-43,65*	0,488	0,00	0,282	-42,21*
max(idfd,idft)	0,489	0,20	0,267	-45,29*	0,496	1,64	0,275	-43,65*
max(idfp,idft)	0,519	**6,35***	0,263	-46,11*	0,540	10,66*	0,274	-43,85*
min(idfd,idfp)	0,502	2,87	0,396	-18,85*	0,529	8,40*	0,405	-17,01*
min(idfd,idft)	0,518	6,15*	0,273	-44,06*	0,535	9,63*	0,281	-42,42*
min(idfp,idft)	0,500	2,46	0,395	-19,06*	0,529	8,40*	0,403	-17,42*
1-(1-idfd)(1-idfp)	0,298	-38,93*	0,123	-74,80*	0,307	-37,09*	0,105	-78,48*
1-(1-idfd)(1-idft)	0,113	-76,84*	0,049	-89,96*	0,082	-83,20*	0,010	-97,95*
1-(1-idfp)(1-idft)	0,323	-33,81*	0,148	-69,67*	0,333	-31,76*	0,125	-74,39*
1-(1-idfp)(1-idfp)(1-idft)	0,522	6,97*	0,283	-42,01*	0,535	9,63*	0,279	-42,83*

O símbolo '*' indica ganhos estatisticamente significativos com, pelo menos, 95% de confiança.

Como observado nas Tabelas 4.1 e 4.2, em geral, a busca na coleção de códigos é mais efetiva que na coleção de perfis, o que confirma os resultados de Burrows et al. [2009]. Contudo, os melhores resultados na coleção INICIANTE foram obtidos na busca por perfis, para a combinação idfd*idfp*idft. Este, contudo, foi um resultado isolado, provavelmente relacionado com o fato de que esta coleção deve possuir mais códigos copiados entre os alunos e um número grande de tarefas. Isto também pode explicar porque a estratégia tfd/C foi a pior nesta coleção enquanto foi a melhor na coleção AVANÇADO.

Tabela 4.2: Variantes do método BM25 na coleção AVANÇADO

Combinação de IDF's	tfd/C	Ganho(%)	tfd/P	Ganho(%)	tfp/C	Ganho(%)	tfp/P	Ganho(%)
idfd	0,703	-	0,714	1,56	0,703	0,00*	0,688	-2,13
idfp	0,796	13,23*	0,766	8,96*	0,792	12,66*	0,744	5,83
idft	0,814	15,79*	0,703	0,00	0,800	13,80*	0,685	-2,56
idfd*idfp	0,777	10,53*	0,800	13,80*	0,785	11,66*	0,796	13,23*
idfd*idft	0,811	15,36*	0,755	7,40	0,770	9,53*	0,759	7,97
idfp*idft	0,862	**22,62***	0,807	14,79*	0,840	**19,49***	0,796	13,23*
idfd*idfp*idft	0,825	17,35*	0,822	**16,93***	0,829	17,92*	0,822	**16,93***
idfd+idfp	0,722	2,70*	0,722	2,70	0,714	1,56	0,711	1,14
idfd+idft	0,729	3,70*	0,714	1,56	0,725	3,13	0,688	-2,13
idfp+idft	0,820	16,64*	0,725	3,13	0,811	15,36*	0,692	-1,56
idfd+idfp+idft	0,737	4,84*	0,725	3,13	0,725	3,13	0,696	-1,00
max(idfd,idfp)	0,703	0,00	0,714	1,56	0,703	0,00	0,688	-2,13
max(idfd,idft)	0,700	-0,43	0,700	-0,43	0,714	1,56	0,662	-5,83
max(idfp,idft)	0,818	16,36*	0,707	0,57	0,800	13,80*	0,681	-3,13
min(idfd,idfp)	0,796	13,23*	0,766	8,96*	0,792	12,66*	0,744	5,83
min(idfd,idft)	0,825	17,35*	0,733	4,27	0,788	12,09*	0,711	1,14
min(idfp,idft)	0,807	14,79*	0,774	10,10*	0,800	13,80*	0,744	5,83
1-(1-idfd)(1-idfp)	0,674	-4,13	0,629	-10,53*	0,688	-2,13	0,614	-12,66*
1-(1-idfd)(1-idft)	0,559	-20,48*	0,474	-32,57*	0,551	-21,62*	0,414	-41,11*
1-(1-idfp)(1-idft)	0,774	10,10*	0,644	-8,39	0,770	9,53*	0,633	-9,96*
1-(1-idfp)(1-idfp)(1-idft)	0,714	1,56	0,596	-15,22*	0,751	6,83*	0,559	-20,48*

O símbolo '*' indica ganhos estatisticamente significativos com, pelo menos, 95% de confiança.

Também observamos, comparando as coleções, que é muito mais díficil atribuir autoria na coleção INICIANTE. Isto se deve a dois fatores. Primeiro, os programas nesta coleção foram implementados por alunos com pouca experiência de programação e, portanto, ainda sem um estilo bem definido. Segundo, a coleção INICIANTE conta com mais autores que a coleção AVANÇADO, o que torna o problema de atribuição mais difícil de resolver.

Quando consideramos os métodos de combinação usados para a estimativa de idf, observamos que o melhor foi a conjunção para ambas as coleções, com exceção de um caso isolado da combinação máximo na coleção INICIANTE. Entre as informações combinadas, a raridade no perfil e na tarefa foram as que produziram os resultados mais efetivos. No caso da busca em perfis, a raridade nos códigos é necessária para a obtenção de melhores resultados. De fato, o uso isolado da informação de perfil (idfp) e tarefa (idft) já é o suficiente para se obter ganhos significativos sobre o baseline, neste caso, da ordem de 3% (idfp) e 6% (idft) em INICIANTE e 13% (idfp) e 15% (idft) em AVANÇADO. Nestas coleções, a informação de tarefa foi levemente mais efetiva que a de perfil.

Destes resultados, concluimos que, para a coleção INICIANTE, uma boa característica de perfil de programação é uma comum no *perfil* mais similar à consulta enquanto na coleção AVANÇADO, é uma comum no *código* mais similar. Em am-

bos os casos, a característica deve ser rara nos demais perfis, bem como na tarefa correspondente ao código consulta.

Note que os melhores resultados obtidos para as variantes de BM25 serão referenciadas nas próximas seções como BM25dxpxt1 (ganho de cerca de 15% sobre Burrows et al. [2009] em coleção INCIANTE) e BM25pxt1 (ganho de cerca de 23% sobre Burrows et al. [2009] em coleção AVANÇADO).

4.3.2 Modelos de Linguagem

Ao contrário do primeiro conjunto de experimentos, nós agora não consideramos que a informação de tarefa é conhecida. Assim, testamos modelos que usam tão somente a informação de perfil para a atribuição de autoria.

Em particular, a Figura 4.1 mostra os resultados obtidos com métodos de PERFIL, quando variando os parâmetros λ e μ, nas coleções INICIANTE e AVANÇADO (Figures 4.1a–(d)). As mesmas figuras também reportam os resultados obtidos para modelos de linguagens simples (ML-λ e LM-μ), as melhores variantes BM25 (BM25dxpxt1 e BM25pxt1) e o método proposto por Burrows et al. [2009].

Como podemos observar, o método baseado em PERFIL-λ tem desempenho descrescente à medida que se aumenta o valor de λ. Para $\lambda = 0$, o método é sempre melhor que o proposto por Burrows et al. [2009]. Surpreendentemente, para coleção INCIANTE e $\lambda = 0$, ele é mesmo superior à melhor variante do BM25, mesmo embora não use informação de tarefa. Por sua vez, o método PERFIL-μ tem desempenho crescente ou idêntico à medida que se aumenta o valor de μ. Ele é sempre igaul (INICIANTE) ou melhor (AVANÇADO) aos baselines, incluindo a melhor variante do BM25. Comparando ambos os modelos, PERFIL-μ apresenta melhor desempenho, em geral.

Ao analisar os melhores parâmetros em detalhe, notamos que ambos os modelos têm melhor desempenho à medida em que desconsideram a estimativa dos termos observados e não observados na coleção de códigos. Ou seja, ambos os modelos conseguem bons resultados simplesmente por considerar a contribição para o perfil dos termos observados no código consulta. No caso do PERFIL-λ, isso é claro já que ao usar $\lambda = 0$, a Equação 3.2 se reduz à Equação 3.4. O mesmo se observa para a Equação 3.5, se observarmos que o valor μ é dominado pelo número de ocorrências de uma característica em um perfil (estimativa $c(w, P)$ na Equação 3.5).

A Figura 4.2 mostra os resultados obtidos com métodos de PROGRAMA, quando variando os parâmetros λ e μ, nas coleções INICIANTE e AVANÇADO (Figures 4.2a–(d)). Como antes, as figuras também reportam os resultados obtidos para modelos

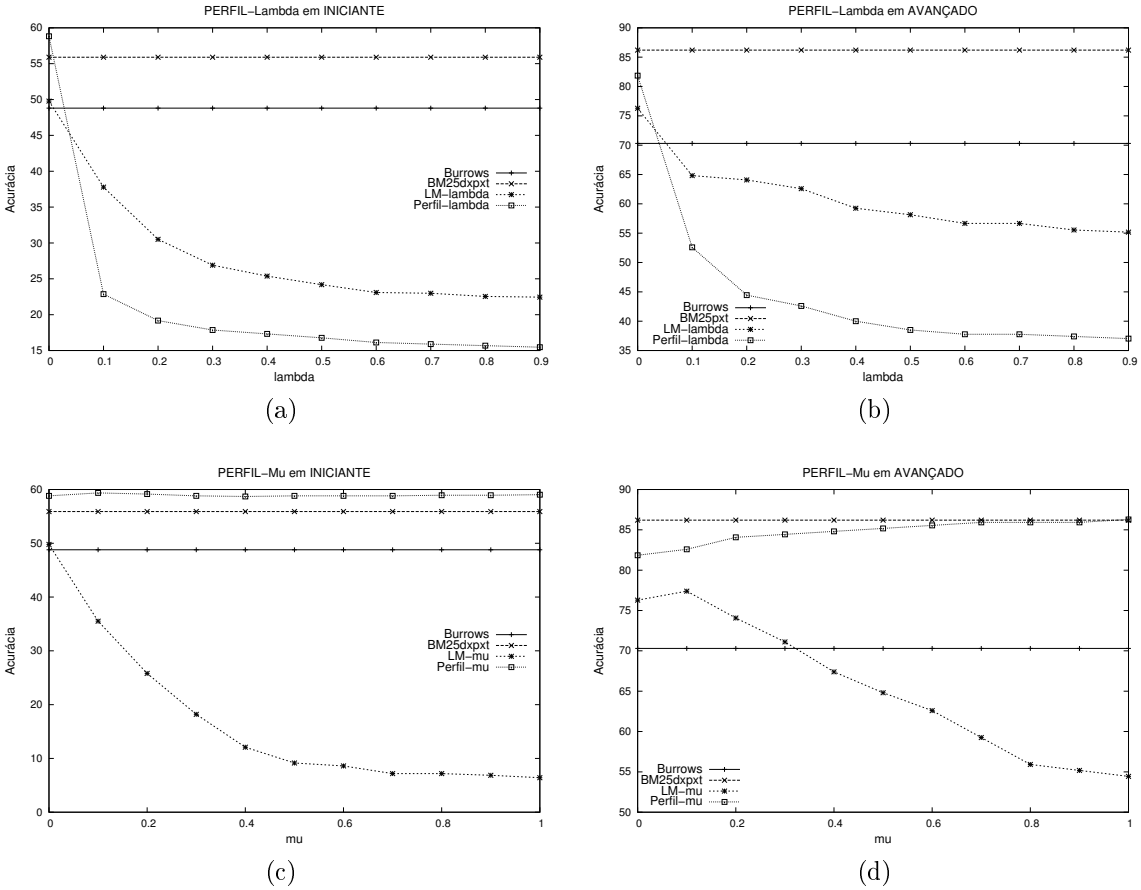


Figura 4.1: PERFIL-λ em INICIANTE (a); PERFIL-λ em AVANÇADO (b); PERFIL-μ em INICIANTE (c); e PERFIL-μ em AVANÇADO (d).

de linguagens simples (ML-λ e LM-μ), as melhores variantes BM25 (BM25dxpxt1 e BM25pxt1) e o método proposto por Burrows et al. [2009].

Como podemos observar, os métodos baseados em PROGRAMA-λ e PROGRAMA-μ têm desempenho crescente à medida que se aumenta o valor de β, alcançando maior desempenho com $\beta = 1$, exceto nno caso de PROGRAMA-λ em AVANÇADO. Para a coleção INICIANTE, ele apresenta desempenho superior à melhor variante do BM25. Novamente, comparando ambos os modelos, PROGRAMA-μ apresenta melhor desempenho que PROGRAMA-λ, em geral. Também observamos que os modelos baseados em PROGRAMA apresentam melhores resultados que os modelos baseados em PERFIL, com ganhos da ordem de 21% (INICIANTE) e 23% (AVANÇADO) sobre o método de Burrows et al. [2009]. Isto era de certa forma esperado, uma vez que o segundo conjunto de modelos usa a estimativa do primeiro.

Ao analisar os melhores valores de β em detalhe, notamos que ambos os modelos são mais efetivos à medida em que desconsideram a estimativa de importância das

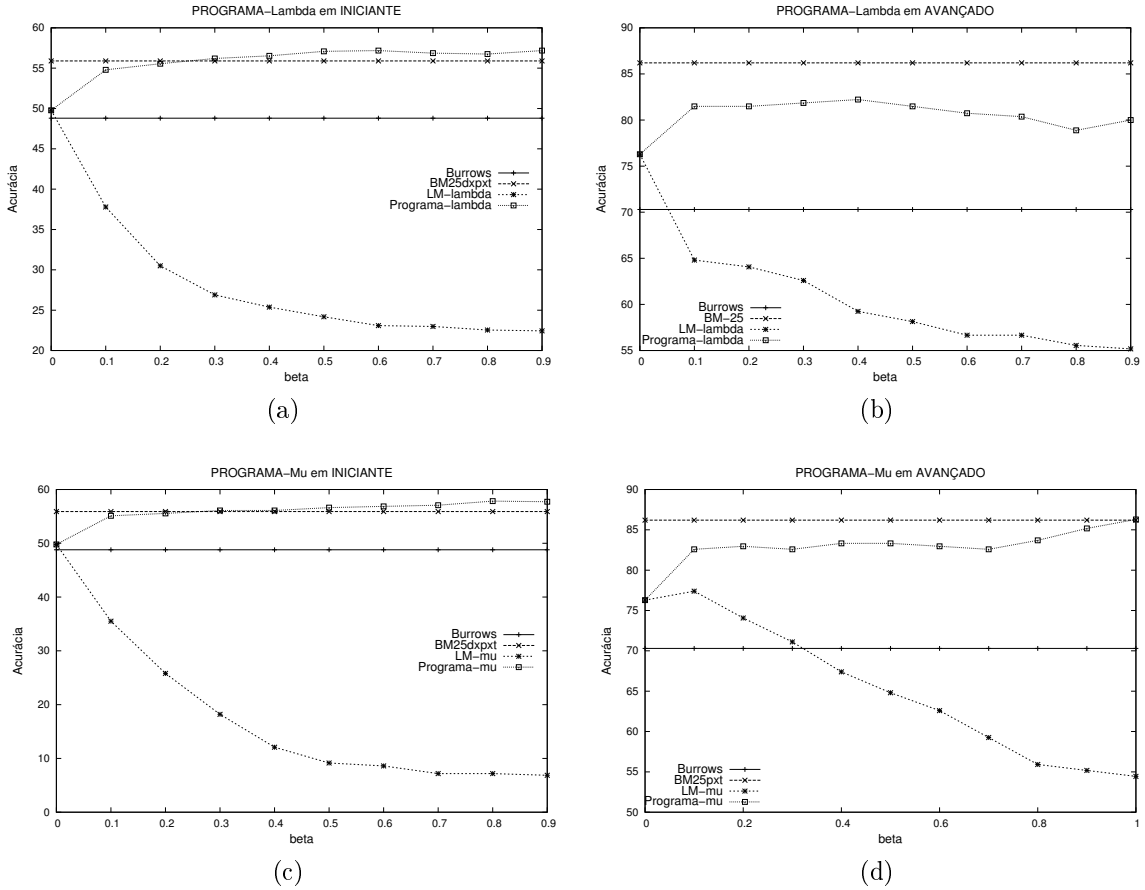


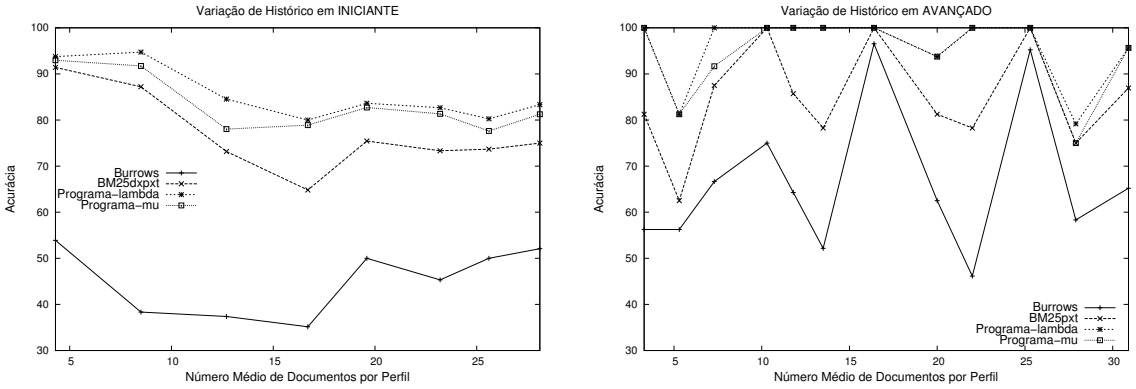
Figura 4.2: PROGRAMA-λ em INICIANTE (a); PROGRAMA-λ em AVANÇADO (b); PROGRAMA-μ em INICIANTE (c); e PROGRAMA-μ em AVANÇADO (d).

características nos códigos ($P_{ML}(w|d)$ na Equação 3.6). Com *beta* tendendo para 1, os modelos PROGRAMA consideram basicamente a estimativa de importância no perfil (conforme modelos de PERFIL) das características observadas e não observadas do programa-consulta.

Ambos os modelos apresentaram resultados surpreendentes quando consideramos o fato que superaram várias vezes as melhores variantes do BM-25, sem usar informação de tarefa. Também é surpreendente que o componente chave dos modelos foi a simples estimativa de importância no perfil (com a consideração de características não observados apenas em uma segunda suavização).

4.3.3 Impacto do tamanho do Histórico de Programas

Para verificar o impacto do tamanho do histórico de progrmas no desempenho dos métodos estudados, simulamos uma situação de uso real, em que a cada dia deve



(a) (b)

Figura 4.3: Resultados dos métodos ao variar o tamanho do histórico de programas nas coleções INICIANTE (a) e AVANÇADO (b).

ser determinada a autoria de um grupo de programas consultas, considerando todos os programas anteriormente entregues pelos autores. Assim, em cada dia, o número médio de programas por perfil de autor é maior que no dia anterior.

A Figura 4.3 mostra os resultados obtidos nas coleções INICIANTE (Figura 4.3a) e AVANÇADO (Figura 4.3b). Os métodos considerados neste estudo são os modelos de linguagens PROGRAMA e os baselines (variantes do BM25, BM25dxpxt1 e BM25pxt1, e o método proposto por Burrows et al. [2009]). Para tornar mais legíveis as figuras, não incluimos os desepenhos dos métodos mais simples baseados em modelos de linguagem (ML-λ e ML-μ) nem dos modelos PERFIL.

Como podemos observar na Figura 4.3a, os modelos PROGRAMA possuem desempenho sempre superior aos baselines e com menor variação de desempenho de dia para dia. Embora aparentemente surpreendente, a queda no desempenho, à medida que aumenta o tamanho do perfil, se deve à maior complexidade da tarefa à medida que mais programas que podem ser similares ao programa-consulta são adicionados à base. O importante neste experimento é que mesmo para um número pequeno de programas por perfil, os modelos são capazes de atribuir corretamente os autores. Na coleção AVANÇADO (Figura 4.3b), os baselines apresentaram desempenho bem mais variável. Os modelos PROGRAMA, neste caso, foram sepre muito efetivos independentemente do número de programas no perfil.

4.4 Conclusões

Neste capítulo, apresentamos os experimentos realizados para avaliação dos modelos propostos e comparação com diferentes baselines. No primeiro conjunto de experimentos, observamos que ao considerarmos a raridade das características nas tarefas e perfis, fomos capazes de atribuir autoria melhor que o estado-da-arte na literatura. Os resultados também sugerem que, como esperado, coleções com programas de alunos iniciantes representam um desafio maior que coleções de programas de alunos avançados. Em seguida, observamos que modelos baseados apenas em estimativas derivadas de perfis são capazes de resultados tão bons como nossas melhores variantes de BM25, mesmo sem usar informação de tarefa. Finalmente, observamos que os modelos estudados têm desempenho muito bom mesmo com históricos de programas relativamente pequenos.

Capítulo 5

Conclusões e Trabalhos Futuros

Neste capítulo, apresentamos as conclusões do nosso trabalho, incluindo limitações de nossa pesquisa e direções futuras que podem ser exploradas.

5.1 Resultados Obtidos

Nesta trabalho propusemos métodos para a resolução do problema de atribuição de autoria em coleções acadêmicas caracterizadas por (a) programas escritos para resolver os mesmos problemas (tarefas) ou problemas similares, (b) autores inexperientes e avançados e (c) históricos de programas possivelmente pequenos. Para tanto, utilziamos uma estratégia de atribuição de autoria baseada em busca dos códigos mais similares em uma coleção de treino. Nossa contribuição consistiu na formulação de modelos de busca que levassem em conta informações de tarefas (quando disponível) e perfis na estimativa da importância das características de perfil de programação. Como resultado, sugerimos dois conjuntos de modelos, um que considera que a informação de tarefa está disponível e um segundo que considera que apenas a informação de perfil pode ser usada. Adicionalmente, estudamos o impacto de coleções formadas por programadores iniciantes e avançados bem como históricos de diferentes tamanhos.

Como resultado deste trabalho, obtivemos modelos probabilísticos, baseados em BM25, que alcançaram ganhos da ordem de 15% e 23% sobre o estado da arte nas coleções que usamos neste estudo. Resultados similares e mesmo maiores forma obtidos com nossos modelos de linguagem (21% e 23%). Também observamos que a coleção de autores iniciantes representa um desafio maior que a formada por autores avançados, bem como mostramos que os modelos têm bom desempenho mesmo para pequenos históricos de programas.

Considerando as questões de pesquisa que motivaram este trabalho, fornecemos

agora as seguintes respostas:

- *Em um cenário onde é possível agrupar programas de acordo com as tarefas que estes resolvem, a informação da tarefa pode ser usada para melhorar o processo de atribuição de autoria? Como?* Sim, pode. Esta informação pode ser usada como peso de raridade na estimativa da contribuição da característica para a ordenação do código mais similar ao código consulta em um modelo BM25.

- *Em um cenário onde não é possível agrupar programas de acordo com as tarefas que estes resolvem, como produzir um método mais robusto de atribuição de autoria?* Esta informação pode ser usada como fonte para geração de características ("palavras") em um modelo de linguagem. Assim, estimativas de importância vão considerar o peso nos perfis diminuindo a chance de casamento direto entre o código consulta e um código particular de um autor. Desta forma, é menor a chance de casamento com códigos similares escritos para uma mesma tarefa.

- *Qual o desempenho dos métodos propostos quando consideramos cenários em que os alunos são iniciantes e cenários em que eles são experientes?* Nossos resultados sugerem que coleções de programas de alunos iniciantes representam um problema mais difícil de resolver. Contudo, novos experimentos devem ser realizados usando um mesmo número de autores nos dois cenários e dados mais confiáveis de autoria. Da forma como os experimentos foram feitos, o menor desempenho dos métodos no caso dos alunos iniciantes pode estar relacionado a características desta base como um maior número de autores e um maior número de trabalhos copiados.

- *Qual o impacto do tamanho do histórico de programas do aluno na atribuição e autoria, nos métodos propostos, considerando tanto o cenário envolvendo alunos iniciantes quanto experientes?* Todos os métodos propostos apresentaram resultados satisfatórios para números pequenos de programas por histórico de autor.

5.2 Limitações

As principais limitações deste trabalho estão relacionadas ao processo de avaliação, uma vez que não há como garantir que todos os códigos usados nas bases coletadas são efetivamente dos autores. Em particular, na base INICIANTE, é provável que haja informação de atribuição incorreta. Outras possíveis limitações da avaliação realizada incluem a não implementação de outro método baseado em perfis (como o proposto

por Frantzeskou et al. [2006]) e a não adoção de uma base de caráter mais geral. No segundo caso, infelizmente, não tivemos acesso a nenhuma das bases usadas em outros trabalhos.

5.3 Trabalhos Futuros

Uma variedade de trabalhos podem ser explorados na continuação desta pesquisa, como descrito a seguir:

- Implementar o método proposto por Frantzeskou et al. [2006] como base de comparação adicional, uma vez que ele também se baseia em uma idéia de perfis. Uma característica interessante da estratégia de Frantzeskou et al. [2006] é que ela mantém nos perfis apenas as características mais comuns, o que impede os perfis de crescerem indefinidamente. Esta ideia sugere que outras técnicas de redução de características poderiam ser estudadas como é o caso de fatoração matricial [Hofmann, 1999]. Esta técnica tem a vantagem adicional de criar perfis representados por fatores latentes relacionados com as características originais.

- Oferecer respostas mais precisas para as questões relacionadas aos cenários com alunos inexperientes e avançados e o tamanho do histórico. No primeiro caso, usar coleções de treino com o mesmo número de autores e com informação de autoria confiável. No segundo caso, testar um mesmo grupo de programas contra perfis de tamanho variável.

- Avaliar os métodos propostos com coleções de caráter mais geral (como bases profissionais) para verificar seu desempenho em quaisquer cenários de atribuição de autoria.

- Incorporar a informação de tarefa nos modelos de geração de linguagem, de maneira a se ter um modelo único que possa ser usado quando esta informação está disponível e quando não está. Uma possibilidade é considerar informação de agrupamento, como descrito no próximo item. Adicionalmente, estudar variações nas estimativas de raridade no perfil.

- Usar métodos de agrupamento para reconhecer conjuntos de tarefas similares, independente desta informação ser ou não disponível. Neste estudo, deve-se ter atenção especial para a distinção entre grupos de programas que se assemelham por compartilharem características da tarefa dos grupos que se assemelham dev-

ido às características de estilo dos autores (o que poderia indicar plágio na base de treino).

- Avaliar os métodos diante da presença de informação incorreta de autoria nas coleções de treino e avaliar o impacto deste ruído. Isso pode ocorrer no cenário real na medida em que os autores fornecem códigos que não são da sua autoria para compor o seu perfil histórico. Em particular, seria interessante usar códigos externos à base para simular plágios de programas oriundos da internet.

- Usar estratégias supervisionadas na fase de busca do código solução. Uma primeira possibilidade seria o aprendizado dos parâmetros dos modelos propostos nas próprias coleções. Um segunda possibilidade seria o uso de de aprendizagem de *ranking* [Joachims, 2002]. Neste caso, *rankings*, como os propostos na Seção 3.3, baseados em programas, perfis e tarefas (usando ou não métodos de agrupamento) poderiam ser usados como atributos de entrada pro método de aprendizado. Uma vantagem destas estratégias é que o método seria mais geral uma vez que, durante o aprendizado, ele poderia se adaptar naturalmente a cenários distintos, como o acadêmico ou o profissional.

- Adaptar os modelos propostos para o problema de detecção de autoria. Como os modelos são probabilísticos, uma estratégia simples seria obter a maior verossimelhança para a probabilidade do perfil (autor) sugerido, dado o programa consulta. Outra estratégia seria considerar a autoria sugerida como uma hipótese e aplicar testes estatísticos de validação de hipóteses. Neste caso, devemos estudar limiares para estabelecer a confiança da hipótese de autoria.

Referências Bibliográficas

Anh, V. N. & Moffat, A. (2005). Inverted index compression using word-aligned binary codes. *Inf. Retr.*, 8(1):151--166.

Arasu, A.; Cho, J.; Garcia-Molina, H.; Paepcke, A. & Raghavan, S. (2001). Searching the web. *ACM Trans. Internet Technol.*, 1(1):2--43.

Baeza-Yates, R. A. & Ribeiro-Neto, B. (1999). *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Bahle, D.; Williams, H. E. & Zobel, J. (2002). Efficient phrase querying with an auxiliary index. Em *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '02, pp. 215--221, New York, NY, USA. ACM.

Barrón-Cedeño, A.; Rosso, P. & Benedí, J.-M. (2009). Reducing the plagiarism detection search space on the basis of the kullback-leibler distance. Em *Proceedings of the 10th International Conference on Computational Linguistics and Intelligent Text Processing*, CICLing '09, pp. 523--534, Berlin, Heidelberg. Springer-Verlag.

Brin, S. & Page, L. (1998). The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107--117.

Burrows, S. & Tahaghoghi, S. M. M. (2007). Source Code Authorship Attribution using N-Grams. Em Spink, A.; Turpin, A. & Wu, M., editores, *Proceedings of the Twelfth Australasian Document Computing Symposium*, pp. 32--39, Melbourne, Australia. RMIT University.

Burrows, S.; Tahaghoghi, S. M. M. & Zobel, J. (2007). Efficient Plagiarism Detection for Large Code Repositories. *Software: Practice and Experience*, 37(2):151--175.

Burrows, S.; Uitdenbogerd, A. L. & Turpin, A. (2009). Application of Information Retrieval Techniques for Source Code Authorship Attribution. Em Zhou, X.; Yokota,

H.; Kotagiri, R. & Lin, X., editores, *Proceedings of the Fourteenth International Conference on Database Systems for Advanced Applications*, DASFAA '09, pp. 699--713, Brisbane, Australia. Springer.

Burrows, S.; Uitdenbogerd, A. L. & Turpin, A. (2012). Comparing techniques for authorship attribution of source code. *Software: Practice and Experience*, pp. n/a--n/a.

Cavnar, W. B. & Trenkle, J. M. (1994). N-gram-based text categorization. *Ann Arbor MI*, 48113(2):161--175.

Frantzeskou, G.; MacDonell, S.; Stamatatos, E. & Gritzalis, S. (2008). Examining the significance of high-level programming features in source code author classification. *J. Syst. Softw.*, 81(3):447--460.

Frantzeskou, G.; Stamatatos, E.; Gritzalis, S.; Chaski, C. E. & Howald, B. S. (2007). Abstract identifying authorship by byte-level n-grams: The source code author profile (scap) method.

Frantzeskou, G.; Stamatatos, E.; Gritzalis, S. & Katsikas, S. (2006). Effective identification of source code authors using byte-level information. Em *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pp. 893--896, New York, NY, USA. ACM.

Harold, F. G. (1986). Experimental evaluation of program quality using external metrics. Em *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pp. 153--167, Norwood, NJ, USA. Ablex Publishing Corp.

Hofmann, T. (1999). Probabilistic latent semantic indexing. Em *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '99, pp. 50--57, New York, NY, USA. ACM.

Joachims, T. (2002). Optimizing search engines using clickthrough data. Em *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '02, pp. 133--142, New York, NY, USA. ACM.

Jones, K. S.; Walker, S. & Robertson, S. E. (2000a). A probabilistic model of information retrieval: development and comparative experiments. *Inf. Process. Manage.*, 36(6):779--808.

Jones, K. S.; Walker, S. & Robertson, S. E. (2000b). A probabilistic model of information retrieval: development and comparative experiments. Em *Information Processing and Management*, pp. 779--840.

Juola, P. (2006). Authorship attribution.

Kim, S.; Kim, H.; Weninger, T.; Han, J. & Kim, H. D. (2011). Authorship classification: a discriminative syntactic tree mining approach. Em *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, SIGIR '11, pp. 455--464, New York, NY, USA. ACM.

Kolter, J. Z. & Maloof, M. A. (2004). Learning to detect malicious executables in the wild. Em *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '04, pp. 470--478, New York, NY, USA. ACM.

Krsul, I. & Spafford, E. H. (1997). Authorship analysis: identifying the author of a program. *Computers And Security*, 16(3):233--257.

Lange, R. C. & Mancoridis, S. (2007). Using code metric histograms and genetic algorithms to perform author identification for software forensics. Em *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pp. 2082--2089, New York, NY, USA. ACM.

Lioma, C. & Blanco, R. (2007). Part of speech based term weighting forinformation retrieval.

Liu, X. & Croft, W. B. (2004). Cluster-based retrieval using language models. Em *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '04, pp. 186--193, New York, NY, USA. ACM.

MacDonell, S.; Gray, A.; MacLennan, G. & Sallis, P. (1999). Software forensics for discriminating between program authors using case-based reasoning, feedforward neural networks and multiple discriminant analysis. Em *Proceedings of the 6th International Conference on Neural Information Processing*, ICONIP 99, pp. 66--71. IEEE.

Manning, C. D.; Raghavan, P. & Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA.

Stamatatos, E. (2009). A survey of modern authorship attribution methods. *J. Am. Soc. Inf. Sci. Technol.*, 60(3):538--556.

Witten, I. H.; Moffat, A. & Bell, T. C. (1999). *Managing Gigabytes : Compressing and Indexing Documents and Images*. Morgan Kaufmann, San Francisco, CA, 2 edição.

Zhai, C. & Lafferty, J. (2004). A study of smoothing methods for language models applied to information retrieval. *ACM Trans. Inf. Syst.*, 22(2):179--214.

Anexo A

Tabela de características de linguagem de programação

As tabelas A.1, A.2, A.3 e A.4 apresentam as listas de operadores e palavras chaves comuns em linguagens C e C++, conforme Burrows & Tahaghoghi [2007] e a tabela A.5 as palavras chave de funções de io e a tabela A.6 apresenta os tokens de espaço em branco utilizados na seleção de características do nosso trablho, conforme descrito na seção 4.2.

Tabela A.1: Operadores comuns a C e C++

OPERADORES			
(	parêntesis	!=	diferente
[	colchete	&	E bit a bit
− >	acesso indireto a membro	ˆ	XOR bit a bit
.	acesso a membro	|	OU bit a bit
++	incremento	&&	E lógico
–	decremento	||	OU lógico
!	negação	?	condição ternária
˜	complemento	=	atribuição
*	multiplicação	+=	mais igual
/	divião	-=	menos igual
%	módulo (resto	*=	vezes igual
+	adição	/=	dividido igual
-	subtração	%=	resto igual
«	deslocamento esquerda	«=	deslocamento esquerda igual
»	deslocamento direita	»=	deslocamento direita igual
<	menor que	&=	E igual
>	maior que	ˆ=	XOR igual
<=	menor ou igual	|=	OU igual
>=	maior ou igual	,	vírgula
==	igualdade		

Adaptado de Burrows & Tahaghoghi [2007]

Tabela A.2: Palavras chave comuns a C e C++

PALAVRAS CHAVE									
auto	do	goto	signed	unsigned	break	double	if	sizeof	void
case	else	int	static	volatile	char	enum	long	struct	while
const	extern	register	switch	continue	float	return	typedef	default	for
short	union								

Adaptado de Burrows & Tahaghoghi [2007]

Tabela A.3: Operadores específicos da linguagem C++

OPERADORES			
− > ∗	ligar ponteiro para membro por ponteiro	::	resolução de escopo
.∗	ligar ponteiro para membro por referência		

Adaptado de Burrows & Tahaghoghi [2007]

Tabela A.4: Palavras chave específicas da linguagem C++

PALAVRAS CHAVE					
asm	export	private	true	bool	false
try	catch	friend	public	typeid	class
reinterpret_cast	typename	const_cast	mutable	static_cast	using
namespace	template	virtual	dynamic_cast	new	this
explicit	operator	throw	protected	inline	delete
wchar_t					

Adaptado de Burrows & Tahaghoghi [2007]

Tabela A.5: Palavras chave de funções de entrada e saída

PALAVRAS CHAVE					
BUFSIZ	FILE	fscanf	_IONBF	rewind	stderr
clearerr	FILENAME_MAX	fseek	L_tmpnam	scanf	stdin
EOF	fopen	fsetpos	NULL	SEEK_CUR	stdout
fclose	FOPEN_MAX	ftell	perror	SEEK_END	tmpfile
feof	fpos_t	fwrite	printf	SEEK_SET	TMP_MAX
ferror	fprintf	getc	putc	setbuf	tmpnam
fflush	fputc	getchar	putchar	setvbuf	ungetc
fgetc	fputs	gets	puts	size_t	vfprintf
fgetpos	fread	_IOFBF	remove	sprintf	vprintf
fgets	freopen	_IOLBF	rename	sscanf	vsprintf

Adaptado de Burrows & Tahaghoghi [2007]

Tabela A.6: Caracteres de espaço em branco

CARACTERES DE ESPAÇO					
‘ ’	espaço em branco	‘\t’	tab	‘\r’	retorno de linha	‘\n’	nova linha

Adaptado de Burrows & Tahaghoghi [2007]