CENTRO UNIVERSITÁRIO FEI
FÁBIO VILLAMARIN ARREBOLA

MODELO DE SUGESTÃO DE CÓDIGO-FONTE BASEADO EM MODELOS DE LINGUAGEM ORGANIZADOS EM UM CONTEXTO HIERÁRQUICO

São Bernardo do Campo
2018

FÁBIO VILLAMARIN ARREBOLA

MODELO DE SUGESTÃO DE CÓDIGO-FONTE BASEADO EM MODELOS DE LINGUAGEM ORGANIZADOS EM UM CONTEXTO HIERÁRQUICO

Tese de Doutorado apresentada ao Centro Universitário FEI como parte dos requisitos necessários para obtenção do título de Doutor em Engenharia Elétrica. Orientado pelo Prof. Dr. Plinio Thomaz Aquino Júnior.

São Bernardo do Campo

2018

Arrebola, Fabio Villamarin.
Modelo de sugestão de código-fonte baseado em modelos de linguagem organizados em um contexto hierárquico / Fabio Villamarin Arrebola. São Bernardo do Campo, 2018.
155 f. : il.

Tese - Centro Universitário FEI.
Orientador: Prof. Dr. Plinio Thomaz Aquino Júnior.

1. Sugestão de código-fonte. 2. Computação baseada em contexto. 3. Modelos de linguagem. I. Aquino Júnior, Plinio Thomaz, orient. II. Título.

Elaborada pelo sistema de geração automática de ficha catalográfica da FEI com os dados fornecidos pelo(a) autor(a).

centro universitário FEI

APRESENTAÇÃO DE TESE
ATA DA BANCA EXAMINADORA

Programa de Pós-Graduação Stricto Sensu em Engenharia Elétrica

Doutorado

PGE-10

Aluno: Fabio Villamarin Arrebola **Matrícula:** 514201-3

Título do Trabalho: Modelo de sugestão de código-fonte baseado em modelos de linguagem organizados em um contexto hierárquico.

Área de Concentração: Inteligência Artificial Aplicada à Automação e Robótica

Orientador: Prof. Dr. Plinio Thomaz Aquino Junior

Data da realização da defesa: 27/11/2018

ORIGINAL ASSINADA

Avaliação da Banca Examinadora

_____

_____

_____

_____

São Bernardo do Campo, / / .

MEMBROS DA BANCA EXAMINADORA	
Prof. Dr. Plinio Thomaz Aquino Junior	Ass.: _____
Prof. Dr. Eduardo Martins Guerra	Ass.: _____
Prof. Dr. Francisco de Assis Zampirolli	Ass.: _____
Prof. Dr. Reinaldo Augusto da Costa Bianchi	Ass.: _____
Prof. Dr. Guilherme Alberto Wachs Lopes	Ass.: _____

A Banca Examinadora acima-assinada atribuiu ao aluno o seguinte:

APROVADO ☒ REPROVADO ☐

VERSÃO FINAL DA TESE

ENDOSSO DO ORIENTADOR APÓS A INCLUSÃO DAS RECOMENDAÇÕES DA BANCA EXAMINADORA

_____

Aprovação do Coordenador do Programa de Pós-graduação

_____

Prof. Dr. Carlos Eduardo Thomaz

AGRADECIMENTOS

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001.

RESUMO

A sugestão de código-fonte é uma funcionalidade muito popular nos modernos ambientes de desenvolvimento integrado. Ela desempenha um papel de grande importância no âmbito da programação, pois permite a execução de atividades de maneira mais rápida e precisa. Há muitas maneiras de se construir um mecanismo de sugestão de código-fonte, incluindo algoritmos especializados, árvores de decisão e, principalmente, modelos de linguagem n-grama. Nesse sentido, a composição e a relevância do contexto de invocação na capacidade preditiva de um mecanismo de sugestão é assunto pouco estudado, e praticamente restrito a modelos com suporte a *cache*, que exploram a relação entre regularidades locais e globais para determinar a probabilidade de que um evento seja observado. No entanto, com inspiração nas definições de computação baseada em contexto, pode-se considerar que outros fatores sejam utilizados, tais como o escopo do projeto de software sendo manipulado, seus objetivos e funcionalidades principais e seu usuário mantenedor. Assim, o objetivo deste trabalho é aumentar a capacidade preditiva de um modelo de sugestão de código-fonte através da proposição de sugestões mais relevantes e assertivas. E para isso, considera-se a elaboração e validação de um modelo de sugestão de código-fonte cujo contexto de invocação seja definido pela segmentação hierárquica de modelos de linguagem n-grama, de tal maneira que essa segmentação considere múltiplos níveis, incluindo o projeto de software, o usuário mantenedor e os objetivos e funcionalidades do projeto de software em questão. O modelo proposto foi avaliado em um conjunto de experimentos controlados que observam a progressão de medidas intrínsecas e os resultados obtidos demonstraram que o modelo proposto é bem-sucedido até mesmo quando comparado a modelos com suporte a *cache*.

Palavras-chave: Sugestão de código-fonte. Computação baseada em contexto. Modelos de linguagem.

ABSTRACT

Code completion is a very popular feature in modern IDEs. It plays a key role in software engineering since it allows fast and precise programming. There is a number of ways of building a code completion engine, including specialized algorithms and decision trees, but the main focus of this dissertation is the n-gram language model. In such scenario, the importance of how to build an invocation context for a completion engine is an open problem, mostly studied by cache-based initiatives. Cache-based models explore how to combine local and global regularities to determine the probability that a given event is observed. However, inspired by context-aware computing definitions, one could expect that alternative factors would also be relevant and play a fundamental role in prediction. Thus, the aim of this dissertation is to design, build and validate a code completion model in which the invocation context is a direct result of a hierarchical organization of n-gram language models that considers scopes such as the software project, its owner and its main goals and functionalities. The proposed model was evaluated using an extended set of controlled experiments that observe the progression of an intrinsic language model measure called cross-entropy. The results obtained demonstrated that the proposed model is successful even when compared to cache-based language models.

Keywords: Code completion. Language models. Context-aware computing.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA ALGORITMOS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
AST	Abstract Syntax Tree
BNF	Backus-Naur Form
BOS	Beginning of Sentence
EOS	End of Sentence
IDE	Integrated Development Environment
LDA	Latent Dirichlet Allocation
LOC	Lines of Code
LSA	Latent Semantic Allocation
LSTM	Long Short-Term Memory
MRR	Mean Reciprocal Rank
kNN	k-Nearest Neighbors
UNK	Unknown

SUMÁRIO

1 INTRODUÇÃO

No contexto da engenharia de software, programadores consideram que a produtividade na execução de suas atividades é determinada, principalmente, pelo uso de ferramentas apropriadas (TRENDOWICZ, MÜNCH, 2009; PAIVA et al., 2010). Essas ferramentas, normalmente combinadas em um ambiente integrado de desenvolvimento, ou *Integrated Development Environment* (IDE), oferecem inúmeras funcionalidades, desde a simples edição de texto até a depuração de código-fonte em tempo de execução.

No entanto, para o contexto deste trabalho, a funcionalidade de maior interesse é a assistência de conteúdo para código-fonte, particularmente naquilo que, segundo as definições de Hindle et al. (2012), é chamado de sugestão de código-fonte[1]. Trata-se de uma funcionalidade tão popular que estudos como Murphy, Kersten, Findlater (2006) e Amann et al. (2016) observam de maneira empírica que a assistência de conteúdo para código-fonte é uma das funcionalidades mais utilizadas nas interações do programador com a IDE.

Em parte, essa popularidade pode ser justificada pelo próprio objetivo dos assistentes de conteúdo, que se resume a auxiliar o programador na conclusão de suas tarefas, oferecendo uma lista de sugestões que incluem constantes, variáveis e métodos que podem ser utilizados para complementar um determinado trecho de código-fonte. De fato, os trabalhos de Church, Nash e Blackwell (2010) e Marasoiu, Church e Blackwell (2015) suportam o fato de que os assistentes de conteúdo desempenham um papel importante como ferramenta de melhoria de produtividade. Além disso, eles também observam que assistentes de conteúdo podem ser vistos como mecanismos de *feedback* em tempo real, já que a não exibição de sugestões acaba, na grande maioria das vezes, por ser interpretada como um indicador de que o código-fonte original é incorreto[2].

[1] Note que as definições de Hindle et al. (2012) dividem a assistência de código-fonte em duas categorias intrinsicamente relacionadas; a categoria de complemento e a categoria de sugestão de código-fonte. Enquanto a primeira pode ser entendida como a tentativa de se completar um termo parcialmente digitado; a segunda pode ser entendida como a tentativa de apresentar um termo completo.

[2] Note que mesmo assim os estudos empíricos de Marasoiu, Church, Blackwell (2015), Amann et al. (2016) e Arrebola e Aquino Junior (2017) observam que aproximadamente metade das interações com os assistentes são canceladas sem que o programador realmente aceite uma das sugestões apresentadas.

O princípio geral de funcionamento de um assistente de conteúdo para código-fonte é bastante simples. Mediante ativação manual ou automática, o funcionamento de um assistente se resume na composição de uma lista de termos considerados válidos dentro de um contexto de invocação, que pode ser entendido como o conjunto de instruções encontradas no entorno do ponto em que o assistente é exibido. E uma vez que essa lista de termos é apresentada para o programador, este pode então optar por efetivar a sugestão que julgar mais adequada.

Para ilustrar essa explicação, considere que um programador tem em mãos o trecho de código-fonte em linguagem de programação Java encontrado na Figura 1.

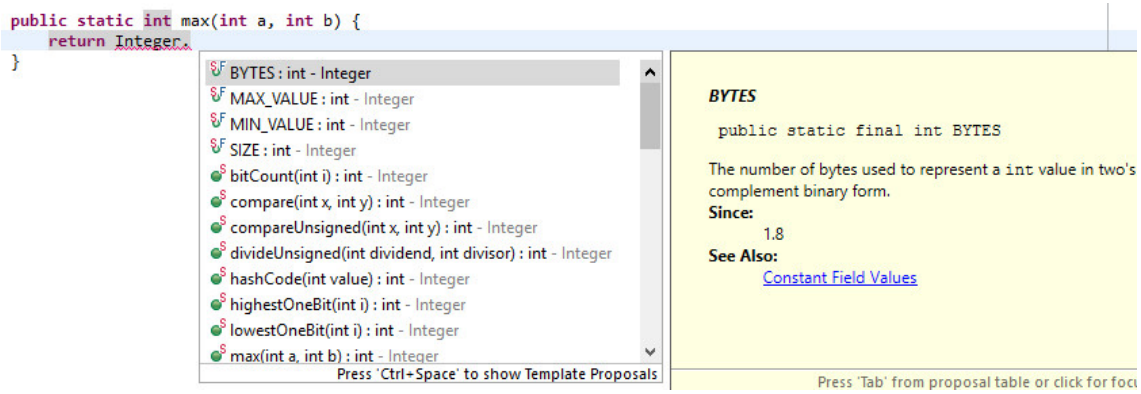
Figura 1 – Trecho de código-fonte para ilustrar o funcionamento de um assistente de conteúdo para sugestão de código-fonte.

```
public static int max(int a, int b) {
    return Integer.
}
```

Fonte: Autor.

No cenário ilustrado pela Figura 1, o objetivo do programador é completar o corpo do método `max`, responsável por determinar qual dos números inteiros de entrada é o maior. O programador deve então encontrar uma função que satisfaça essa restrição, e para isso ele pode acionar o assistente de sugestão de código-fonte. Ao fazê-lo, o assistente compõe a lista de sugestões aplicáveis conforme ilustrado na Figura 2, que representa um assistente na IDE Eclipse (ECLIPSE FOUNDATION, 2018), muito popular entre programadores Java.

Figura 2 – Ilustração da invocação de um assistente de conteúdo para sugestão de código-fonte na IDE Eclipse.

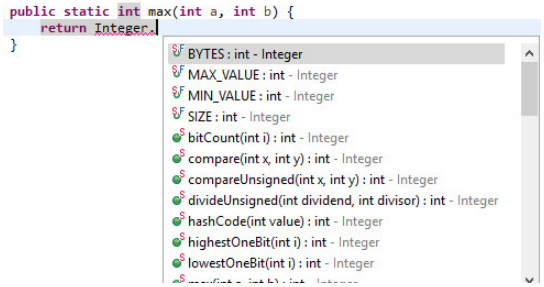


Fonte: Autor.

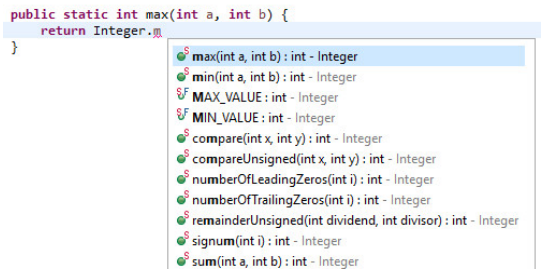
Agora, assim como a sequência de interações ilustrada na Figura 3, basta que o programador avalie as sugestões fornecidas, aplicando filtros adicionais conforme a necessidade e, por fim, opte por incorporar a sugestão mais adequada ao trecho de código-fonte original.

Figura 3 – Ilustração de uma possível sequência de interações de um programador com um assistente de conteúdo para sugestão e complemento de código-fonte na IDE Eclipse.

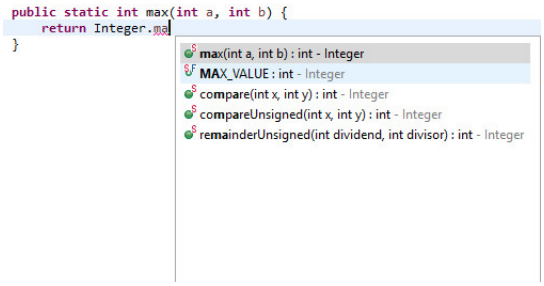
(a) Programador ativa o assistente.



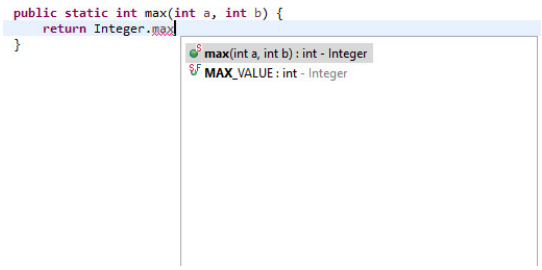
(b) Programador pressiona tecla ‘m’.



(c) Programador pressiona tecla ‘a’.



(d) Programador pressiona tecla ‘x’.



(e) Programador opta por aplicar a sugestão resultante.

```
public static int max(int a, int b) {
    return Integer.max(a, b);
}
```

Fonte: Autor.

No exemplo ilustrado na Figura 3, o objetivo do programador pode ser considerado relativamente simples. Isso porque a função `Integer.max` é significativa o suficiente dentro do objetivo da tarefa para que se considere cabível que um programador profissional a encontre quase de maneira independente, com pouca ou nenhuma interação com o assistente de conteúdo para código-fonte. Apesar disso, em situações como essa, os assistentes exercem um papel importante, pois permitem que o programador desempenhe suas tarefas com maior velocidade, já que basta selecionar e aplicar uma das sugestões apresentadas.

No entanto, os cenários reais nem sempre são assim tão favoráveis. Em cenários mais restritos, caracterizados, por exemplo, pela pouca familiaridade com o domínio do problema ou pela pouca familiaridade com as interfaces de programação envolvidas, o programador pode interagir com o assistente de conteúdo de código-fonte repetidas vezes, em um processo contínuo de filtragem e avaliação das eventuais sugestões oferecidas. Em situações como essa, a exemplo do trabalho de Arrebola e Aquino Junior (2017), podemos considerar que os assistentes também desempenham o papel de ferramenta de busca, por permitir ao programador explorar as características de constantes, tipos, variáveis e métodos.

Para fins de taxonomia, qualquer que seja o propósito de uso, os assistentes de conteúdo de código-fonte podem ser classificados nas categorias léxica e semântica. Na categoria léxica, a lista de sugestões é formada com base na similaridade textual entre o código-fonte original e as opções disponíveis. Já na categoria semântica, a lista de sugestões é composta levando em conta as restrições gramaticais da linguagem de programação. Isso significaria dizer que uma determinada instrução só seria apresentada como sugestão se a mesma fosse aplicável àquele contexto de invocação.

Na literatura relacionada, as técnicas puramente léxicas são estudadas extensivamente. No entanto, é possível observar que as implementações práticas são escassas, e isso se dá possivelmente devido à sua simplicidade e até mesmo à sua aplicabilidade limitada. Já as técnicas semânticas, embora estudadas com menor abrangência, apresentam uma quantidade e diversidade de implementações significante, conforme observado nas listagens não-exaustivas encontradas nos trabalhos de Robbes e Lanza (2010), Asaduzzaman et al. (2014) e Ording (2016). Independente da categoria de técnicas utilizada, a verdade é grande parte do que caracteriza um assistente de conteúdo de código-fonte reside na maneira como a lista de sugestões é composta antes de sua exibição para o programador. E, com o passar do tempo, o que se observa é que as técnicas utilizadas cada vez mais se baseiam em técnicas de inteligência artificial, especialmente no ramo do processamento de linguagem natural.

Sendo assim, a diversidade de técnicas e algoritmos disponíveis, aliada à importância e popularidade adquirida pelos assistentes de conteúdo de código-fonte ao longo dos anos, sugere que ainda há grandes oportunidades de pesquisa sobre o tema, especialmente quando se consideram as técnicas de composição da lista de sugestões que são apresentadas para o programador.

1.1 MOTIVAÇÃO

Há muitas maneiras de endereçar o problema da sugestão de código-fonte, incluindo algoritmos especializados, algoritmos baseados em instâncias, modelos de linguagem estatísticos puramente léxicos e suas extensões, gramáticas livres de contexto probabilísticas e suas adaptações, árvores de decisão combinadas a linguagens específicas de domínio e até mesmo redes neurais. E, embora cada uma delas possua características particulares que limitam sua aplicabilidade, quase todas exploram, mesmo que de maneira implícita, um dos elementos mais relevantes para a motivação deste trabalho, conhecido como contexto de invocação do assistente de sugestão de código-fonte.

Inicialmente, o contexto de invocação do assistente de sugestão de código-fonte deve ser visto como o conjunto de instruções e propriedades no entorno do ponto de invocação. Sua importância para o bom funcionamento de um assistente é considerável, e na literatura especializada é possível encontrar trabalhos que avaliam o impacto do contexto na capacidade preditiva do algoritmo proposto. Asaduzzman et al. (2014), por exemplo, é um trabalho que combina cálculos de distância de *Hamming* (HAMMING, 1950) e de similaridade baseados no algoritmo *simhash* (CHARIKAR, 2002) em uma técnica especializada, que considera um contexto de invocação formado pela linha em que a invocação do assistente ocorre acrescida de, no máximo, as quatro linhas precedentes.

Dependendo da técnica utilizada, o contexto de invocação pode ser definido a partir de análises estáticas de código-fonte, e assim incluir, por exemplo, o conjunto de métodos de um determinado tipo que tenham sido invocados previamente. Isso pode ser observado na categoria de algoritmos baseados em instâncias, particularmente nos trabalhos de Bruch et al. (2009) e Zhang et al. (2012), que adaptam o algoritmo *kNN* (COVER; HART, 1967) para, respectivamente, sugerir chamadas a métodos de um determinado tipo e inferir os eventuais parâmetros necessários para uma chamada.

O contexto de invocação também pode ser definido implicitamente, pela própria técnica utilizada para tratar o problema da sugestão de código-fonte. É o caso, por exemplo, da categoria de modelos de linguagem, uma categoria essencialmente léxica que merece destaque por ser relativamente extensa e propiciar algumas intuições interessantes para o contexto deste trabalho.

Tudo se inicia com Hindle et al. (2012), que são os primeiros a constatar que, em virtude das características de regularidade que projetos de software apresentam, o problema da sugestão de código-fonte pode ser reduzido à capacidade preditiva de modelos de linguagem explícitos baseados na representação n-grama, que podem ser entendidos como modelos probabilísticos que ajudam a prever o próximo elemento de uma sequência de termos.

Desde então, esses modelos de linguagem – que doravante serão simplesmente chamados de modelos de linguagem n-grama – têm sido estudados extensivamente. Allamanis e Sutton (2013) exploram como algumas classes de termos, especialmente os identificadores, influenciam a capacidade preditiva, e sugerem que o logaritmo da probabilidade do n-grama seja considerado como métrica complementar às métricas de complexidade de código tradicionais tais como a quantidade de linhas de código (LOC) ou ainda a complexidade ciclomática de McCabe (MCCABE, 1976). Raychev, Vechev e Yahav (2013) propõem a combinação de modelos de linguagem n-grama às redes neurais recorrentes para prever padrões de uso de *Application Programming Interfaces* (APIs). Tu, Su e Devanbu (2014) observam que os modelos tradicionais de linguagem n-grama são incapazes de capturar regularidades locais, e assim propõem a inclusão de um modelo de linguagem com suporte a *cache*, de maneira similar ao originalmente proposto por Kuhn e Mori (1990).

A lista de exemplos se estende até o recente trabalho de Hellendoorn e Devanbu (2017), que analisa a aplicação de modelos de linguagem explícitos e implícitos no contexto do problema da sugestão de código-fonte. O argumento principal é que modelos de linguagem n-grama, se bem configurados, podem apresentar capacidade preditiva superior a modelos implícitos baseados, por exemplo, em redes neurais recorrentes, assim como encontrado no trabalho de White et al. (2015)[3].

A única exceção para a questão da definição implícita do contexto de invocação em modelos de linguagem n-grama é o trabalho de Nguyen et al. (2013), que propõe a inclusão de informações sintáticas aos termos do vocabulário combinadas à segmentação de n-gramas de acordo com uma distribuição latente de tópicos, fazendo com que o contexto de invocação inclua os tópicos associados às unidades de código-fonte.

[3] Interessante notar que em um contexto mais amplo de aplicação, Jozelfowicz et al., (2016) argumentam justamente o oposto; que modelos de linguagem n-grama apresentam benefícios limitados quando comparados a redes neurais recorrentes.

Outro exemplo de definição implícita do contexto de execução pode ser encontrado na categoria de técnicas que utilizam gramáticas livres de contexto probabilísticas. Gvero e Kuncak (2015), por exemplo, propõem a utilização de gramáticas em um modelo híbrido de sugestão de código-fonte baseado em linguagem natural. Por outro lado, Bielik, Raychev, Vechev (2016) utilizam um contexto de invocação determinado dinamicamente, condicionando as produções gramaticais à execução de pequenos programas escritos em linguagem específica de domínio desenhada para proporcionar a navegação entre os nós de uma árvore sintática abstrata, ou *Abstract Syntax Tree* (AST). Raychev, Bielik, Vechev (2016) acabam por fazer o mesmo, mas com o auxílio de árvores de decisão.

Quase todas as iniciativas apresentadas até aqui consideram um contexto de invocação definido implicitamente pela técnica utilizada para a construção do modelo preditivo. Entretanto, acredita-se que o contexto de invocação, assim como observado nas limitações e sugestões futuras dos trabalhos de Bruch et al. (2009), Asaduzzman et al. (2014) e Proksch, Lerch, Mezini (2015), deva ser definido por mais do que um conjunto pré-determinado de termos ou linhas precedentes. Seguindo as definições de computação baseada em contexto de Dey (2001), que estabelecem que o contexto é qualquer informação que pode ser utilizada para caracterizar a situação de um objeto, é preciso considerar que o contexto de invocação inclua, minimamente, características do projeto de software sendo manipulado e de seus objetivos e funcionalidades principais. Nesse sentido, vale ressaltar que Saraiva, Bird e Zimmermann (2015) constatam empiricamente que programadores utilizam padrões de desenvolvimento ligeiramente diferentes entre os projetos de software em que trabalham.

Ademais, acredita-se que o problema da sugestão de código-fonte não possa ser tratado adequadamente por uma técnica isolada, especialmente em relação à determinação dos objetivos e funcionalidades principais de um projeto de software. Com inspiração no trabalho de Heilman et al. (2007), a combinação de técnicas variadas e a exploração de seus pontos positivos e tratamento de eventuais limitações de maneira apropriada parece ser um caminho promissor.

Portanto, este trabalho sugere que um assistente de sugestão de código-fonte pode apresentar maior capacidade preditiva ao considerar uma abordagem que combina técnicas de processamento de linguagem natural que inclua modelos de linguagem n-grama e modelos de tópicos latentes, desde que eles sejam segmentados hierarquicamente

de acordo com o projeto de software, com seu usuário mantenedor[4] e com seus objetivos e funcionalidades.

1.2 OBJETIVO E CONTRIBUIÇÕES

O objetivo deste trabalho é aumentar a capacidade preditiva de um modelo de sugestão de código-fonte através da proposição de sugestões mais relevantes e assertivas. Para isso, considera-se a elaboração e validação de um modelo de sugestão de código-fonte cujo contexto de invocação seja definido pela segmentação hierárquica dos modelos de linguagem n-grama e dos modelos de tópicos latentes utilizados para a previsão, de tal maneira que a segmentação inclua o âmbito do projeto de software, de seu usuário mantenedor e dos objetivos e funcionalidades do projeto em questão.

Para atingir esse objetivo, o método de pesquisa incluiu a execução de um experimento com programadores profissionais desenhado para coletar informações sobre o uso de assistentes de código-fonte durante a execução de tarefas de programação. Essa pesquisa, encontrada em Arrebola e Aquino Junior (2017), sugere que assistentes de código-fonte poderiam ser mais eficazes se, ao invés de considerar uma definição de contexto unicamente e diretamente acoplada com a estrutura do código-fonte, considerassem uma definição mais abrangente, que incluísse, por exemplo, o objetivo da tarefa sendo realizada.

Assim, a definição e construção de um modelo de sugestão de código-fonte com as características citadas anteriormente é a primeira contribuição deste trabalho. Embora seja possível encontrar na literatura especializada iniciativas que incorporem as mesmas técnicas utilizadas pelo modelo proposto, tais como modelos de linguagem n-grama ou modelos de tópicos latentes, este trabalho difere dos demais ao considerar a organização hierárquica dos modelos preditivos como determinante para o sucesso da sugestão de código-fonte.

[4] O termo usuário mantenedor é utilizado para representar um conjunto de um ou mais usuários finais que representam uma empresa da iniciativa privada ou um grupo da comunidade de código-fonte aberto.

Em caráter complementar, a segunda contribuição deste trabalho é a definição de um conjunto de dados particular para o problema estudado e a adaptação de um conjunto frequentemente utilizado na literatura e originalmente definido por Allamanis e Sutton (2013). A adaptação se faz necessária visto que o conjunto de dados original não contém informações explícitas referentes aos usuários mantenedores, cruciais para uma segmentação hierárquica que considere esse fator.

Devido à ausência de ferramentas apropriadas, uma contribuição adicional deste trabalho é a implementação de um arcabouço específico para o propósito da criação e validação de modelos de previsão de código-fonte[5] baseados em modelos de linguagem n-grama. Este arcabouço tem o potencial de minimizar o esforço de adaptação para utilização de arcabouços de propósito geral de processamento de linguagem natural tais como o SRILM[6] (STOLCKE, 2002) e o KenLM[7] (HEAFIELD, 2011), e pode ser estendido para incorporar diferentes algoritmos e técnicas de processamento. Ademais, o arcabouço em questão pode ser utilizado em avaliações com conjuntos de dados distintos daquele apresentado nesta pesquisa, de tal forma a considerar linguagens de programação variadas, ou ainda maior quantidade de projetos de software e de usuários mantenedores.

Por fim, a última contribuição deste trabalho é a validação do modelo proposto contra técnicas tradicionais de previsão de código-fonte que utilizam modelos de linguagem n-grama, particularmente os modelos baseados em *cache*, sempre através do arcabouço de criação e validação de modelos mencionado anteriormente.

[5] Disponível em https://gitlab.com/fvarrebola/lm4cc/.
[6] Disponível em http://www.speech.sri.com/projects/srilm/.
[7] Disponível em https://kheafield.com/code/kenlm/.

1.3 ORGANIZAÇÃO DO TRABALHO

O restante deste trabalho está organizado nos seguintes capítulos:

a) Capítulo 2. Este capítulo discute a fundamentação teórica, com foco na computação baseada em contexto, nos modelos de linguagem n-grama e na modelagem de tópicos, sempre que possível com indicações sobre o estado da arte no âmbito da sugestão de código-fonte;
b) Capítulo 3. Este capítulo formaliza o problema estudado e apresenta o modelo de sugestão de código-fonte proposto por este trabalho;
c) Capítulo 4. Este capítulo apresenta a metodologia de avaliação do modelo proposto, a composição dos corpos de dados considerados, as avaliações em si e os resultados obtidos com modelo apresentado no capítulo anterior; e
d) Capítulo 5. Este capítulo apresenta as conclusões do trabalho, consolidando as discussões observadas no capítulo anterior, discutindo as limitações do modelo apresentado e traçando possibilidades de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

O objetivo deste capítulo é proporcionar ao leitor a fundamentação teórica básica para a compressão dos objetivos propostos por este trabalho. Aqui, são discutidos de maneira não exaustiva os tópicos relacionados à computação baseada em contexto, aos modelos de linguagem n-grama e à modelagem de tópicos latentes, sempre que possível com indicações sobre o estado da arte no âmbito da sugestão de código-fonte.

2.1 COMPUTAÇÃO BASEADA EM CONTEXTO

A definição da computação baseada em contexto remonta ao trabalho de Schilit, Adams e Want (1994), com a introdução de conceito de um software capaz de examinar e responder às mudanças de contexto do usuário final. Em seu trabalho, os aspectos considerados na definição do contexto incluem a localização e os recursos computacionais disponíveis no entorno do usuário final.

Desde então, um grande conjunto de aspectos tem sido proposto para definição do contexto, incluindo, por exemplo, o papel que o usuário final assume e a atividade que ele desempenha (KALTZ; ZIEGLER; LOHMANN, 2005). Entretanto, acredita-se que a proposta de aspectos tenha atingido seu ápice com o trabalho de Dey (2001), quando este autor define contexto de maneira bem abrangente, mais especificamente como qualquer informação que possa ser utilizada para caracterizar a situação de uma entidade.

No âmbito da sugestão de código-fonte pode-se dizer que a aplicação dos conceitos de computação baseada em contexto se resume a considerar originalmente dois grandes aspectos; as características estruturais da unidade de código-fonte e do projeto de software sendo manipulado, e as restrições impostas pela gramática associada à linguagem de programação. Essa constatação advém da observação de que assistentes de sugestão de código-fonte, a exemplo dos assistentes embarcados com a IDE Eclipse (ECLIPSE FOUNDATION, 2018), apresentam sugestões concordantes com o contexto de invocação, listando como sugestões apenas aquelas opções tidas como válidas (isto é, opções que são acessíveis dentro daquela unidade de código-fonte ou projeto, e que não resultam em violações gramaticais). Por fim, pode-se observar a existência de um terceiro aspecto, particularmente presente na sugestão de métodos de uma determinada API, visto

que os assistentes de código-fonte acabam por considerar a própria API como elemento formador do contexto.

Na literatura relacionada, o trabalho Tu, Su e Devanbu (2014), em um estudo sobre a relevância das regularidades locais encontradas em código-fonte, considera uma definição de contexto de dois níveis; um nível específico e um genérico, conforme detalhado na seção 2.2. Hellendoorn e Devanbu (2017) têm visão similar, ainda que em um estudo sobre a eficácia da adoção de redes neurais profundas para modelagem de código-fonte em comparação a modelos de linguagem n-grama. No entanto, eles consideram três níveis na definição de contexto; um nível de agrupamento de arquivos em um mesmo diretório, um nível de projeto e um nível geral.

Este trabalho considera que a definição de contexto utilizada por um assistente de código-fonte deva abranger aspectos formadores tais como o projeto de software sendo manipulado, o usuário mantenedor deste projeto e os objetivos e funcionalidades do projeto em questão. Observe, no entanto, que essa proposta não limita ou impede que outros aspectos sejam considerados. Assim, até mesmo os aspectos originais mencionados inicialmente – características estruturais, restrições gramaticais e APIs em uso – poderiam ser considerados.

2.2 MODELOS DE LINGUAGEM N-GRAMA

Um modelo de linguagem n-grama, ou estimador n-grama, tem por objetivo determinar a probabilidade de uma sentença s composta por um conjunto de termos $w_i \dots w_n$ (e que também pode ser representando por w_1^n) de acordo com a Equação 1.

$$P(w_i \dots w_n) = P(w_1^n) = \prod_{i=1}^{n} P(w_i | w_1^{i-1}) \quad (1)$$

A dificuldade em se calcular $P(w_1^n)$ para longas sequências de termos requer a adoção de uma simplificação através da qual a probabilidade de observação de um termo é condicionada a um conjunto reduzido de termos anteriores. Essa simplificação, conhecida como premissa *markoviana*, pode ser observada no exemplo de bigrama, ou 2-grama, definido na Equação 2.

$$P(w_1^n) \approx \prod_{i=1}^{n} P(w_i | w_{i-1}) \quad (2)$$

Nesse cenário, a probabilidade de que o termo w_i seja observado após w_{i-1} é determinada de acordo com a Equação 3, com $C(x, y)$ representando uma função de contagem de termos.

$$P(w_i | w_{i-1}) = \frac{C(w_{i-1}, w_i)}{C(w_{i-1})} \quad (3)$$

Por tratar de uma distribuição de probabilidades, modelos n-grama precisam, de acordo com seu grau, ser aumentados com símbolos especiais que representem o início e o término de uma sentença (JURAFSKY; MARTIN, 2009). Esses termos, que podem assumir, por exemplo, os valores `<BOS>` e `<EOS>`, são geralmente acrescidos a um termo especial `<UNK>` que indica as palavras não observadas no vocabulário.

Aliás, umas das maiores limitações com modelos de linguagem n-grama é subestimar a probabilidade de sequências de termos que não tenham sido observadas no corpo de treinamento (JURAFSKY; MARTIN, 2009), em um problema conhecido como problema da frequência zero. Para que essa limitação seja tratada adequadamente, é comum reavaliar probabilidades zero e probabilidades muito baixas com o auxílio de algoritmos de suavização, ou *smoothing*.

Um dos exemplos mais simples de algoritmo de suavização é o *add-one*, uma suavização aditiva que credita uma observação adicional a todos os eventos de um corpo de dados. Assim, e considerando novamente o exemplo do 2-grama, a probabilidade $P(w_i|w_{i-1})$ de observação do termo w_i após w_{i-1} passa a ser definida de acordo com a Equação 4, com $|V|$ representando o tamanho do vocabulário em questão.

$$P(w_i|w_{i-1}) = \frac{C(w_{i-1}, w_i) + 1}{C(w_{i-1}) + |V|} \tag{4}$$

No entanto, Chen e Goodman (1999) observam em seu estudo sobre algoritmos de suavização que o algoritmo Kneser-Ney (KNESER; NEY, 1995) apresenta desempenho superior quando aplicado a modelos de linguagem. Seu funcionamento se baseia principalmente em probabilidades de continuação, que, novamente para o exemplo de 2-grama, são definidas de acordo com a Equação 5.

$$P_{cont}(w) = \frac{|\{w_{i-1}: C(w_{i-1}, w) > 0\}|}{\left|\{(w'_{i-1}, w'): C(w'_{i-1}, w') > 0\}\right|} \tag{5}$$

A probabilidade de continuação P_{cont} de um termo w é determinada pela razão entre a quantidade de diferentes termos que o precedem e a quantidade total de diferentes 2-gramas. Assim, a probabilidade $P(w_i|w_{i-1})$ de que o termo w_i seja observado após w_{i-1} é definida de acordo com a Equação 6, com δ representando o valor de desconto absoluto e λ representando uma constante de normalização utilizada para redistribuir a massa de probabilidade restante definida conforme a Equação 7.

$$P(w_i|w_{i-1}) = \frac{\max(C(w_{i-1,}w_{i,}) - \delta, 0)}{C(w_{i-1})} + \lambda(w_{i-1})P_{cont}(w_i) \tag{6}$$

$$\lambda = \frac{\delta}{C(w_{i-1})}|w: C(w_{i-1}, w) > 0| \tag{7}$$

Já a formulação do algoritmo de suavização Kneser-Ney para n-gramas de ordem superior requer a abordagem recursiva definida na Equação 8, considerando a substituição da notação $P(w_i|w_{i-1})$ por $P_{KN}(w_i|w_{i-n+1}^{i-1})$ e a subsituição da função de contagem C por $C_{KN}(\cdot)$, que passa a variar de acordo com a ordem do n-grama sob avaliação.

$$P_{KN}(w_i|w_{i-n+1}^{i-1}) = \frac{\max(C_{KN}(w_{i-n+1}^{i}) - \delta, 0)}{C_{KN}(w_{i-n+1}^{i-1})} + \lambda(w_{i-n+1}^{i-1})P_{KN}(w_i|w_{i-n+2}^{i-1})$$
$$C_{KN}(\cdot) = \begin{cases} C(\cdot) & \text{contagem de termos para a maior ordem} \\ C_{cont}(\cdot) & \text{contagem de termos para ordens inferiores} \end{cases} \tag{8}$$

Há ainda outras técnicas que podem ser utilizadas para minimizar as limitações de modelos n-grama, tais como o *backoff* e a interpolação. As técnicas de *backoff* utilizam n-gramas organizados hierarquicamente de acordo com sua ordem, ou seja, se a capacidade preditiva de um n-grama de ordem 3 for insuficiente, avalia-se a resposta de um n-grama de ordem 2, e assim sucessivamente. Já as técnicas de interpolação consideram uma combinação de estimadores n-gramas, cada qual com um peso associado.

2.2.1 Entropia cruzada

A capacidade preditiva de um modelo de linguagem n-grama é determinada através de medidas de entropia cruzada, derivada da medida de entropia originária da teoria da informação. A entropia determina o limite mínimo na quantidade de *bits* necessários para codificar uma decisão ou pedaço de informação em um esquema ótimo de codificação (JURAFSKY; MARTIN, 2009). Sua definição pode ser encontrada na Equação 9, considerando que a variável aleatória W define o conjunto de termos w a serem previstos e p define a distribuição de probabilidades associada.

$$H(W) = -\sum_{w \in W} p(w) \log_2 p(w) \tag{9}$$

No entanto, há situações em que a distribuição p é desconhecida, e portanto não se pode calcular o valor real de entropia. É o caso, por exemplo, de modelos de linguagem construídos e avaliados a partir de conjuntos de dados distintos. Em situações como essa é preciso considerar um modelo de aproximação q, o que resulta em uma medida de entropia cruzada definida de acordo com a Equação 10.

$$H(p, q) = -\sum_{w \in W} p(w) \log_2 q(w) \tag{10}$$

Na prática, a entropia cruzada acaba sendo determinada de acordo com a Equação 11, que considera as observações na distribuição p como igualmente prováveis, o que torna a entropia cruzada um limite superior para a entropia. E considerando que para qualquer modelo de aproximação q tem-se $H(p) \leq H(p, q)$, conclui-se que modelos com alta capacidade preditiva produzem medidas próximas à entropia real.

$$H(q) = -\frac{1}{N} \sum_{w \in W} \log_2 q(w) \tag{11}$$

Por fim, vale ressaltar uma questão sobre a interpretação das medidas de entropia cruzada. Ao considerar dois modelos de aproximação quaisquer, aquele com as menores medidas de entropia cruzada para a sequência de termos em questão representa o modelo com melhor capacidade preditiva; isto é, o modelo com maior acurácia.

2.2.2 Aplicações de modelos de linguagem n-grama

As aplicações de modelos de linguagem são variadas, e incluem, por exemplo, correção de grafia e correção automática de texto. Já no contexto de sugestão de código-fonte, a primeira aplicação prática pode ser observada a partir do trabalho Hindle et al. (2012). A partir daí, a quantidade de estudos é considerável, incluindo, por exemplo, os trabalhos de Allamanis e Sutton (2013), Nguyen et al. (2013), Raychev, Vechev e Yahav (2013) e Hellendoorn e Devanbu (2017).

O trabalho de Tu, Su e Devanbu (2014) é um exemplo importante para os objetivos desse trabalho, pois introduz o conceito de *cache* no âmbito de modelos n-grama ao observar que código-fonte, de maneira geral, costuma apresentar regularidades locais que modelos de linguagem n-grama tradicionais são incapazes de capturar. Assim, Tu, Su e Devanbu (2014) propõem a priorização de locais de acordo com a Equação 12, que considera que a probabilidade de ocorrência do termo t_i após a sequência de termos h seja determinada pela soma entre as probabilidades P_{n-gram} e P_{cache}, considerando λ como o fator utilizado para evidenciar o estimador global ou o *cache*.

$$P(t_i|h, cache) = \lambda * P_{n-gram}(t_i|h) + (1 - \lambda) * P_{cache}(t_i|s) \qquad (12)$$

Neste trabalho, a importância do conceito de *cache* deve ser traduzida para a organização hierárquica de modelos de linguagem. Assim como pode ser observado no capítulo 3, a probabilidade do termo a ser sugerido é determinada pelo resultado da combinação das probabilidades de cada um dos níveis da hierarquia do modelo proposto, que devem ser ativados separadamente.

2.3 MODELAGEM DE TÓPICOS

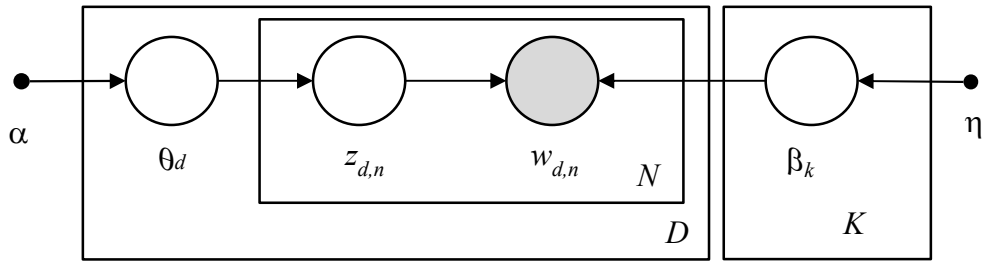
A modelagem de tópicos pode ser descrita como um método para que se determine um modelo estatístico generativo que possa ser utilizado para inferir as estruturas semânticas latentes[8] de uma coleção de documentos essencialmente não estruturados. Em outras palavras, trata-se de uma técnica que, geralmente a partir de uma contagem de frequência de termos, é capaz de determinar os tópicos aos quais um documento está associado. Isso porque a modelagem de tópicos considera que cada documento é um aglomerado de tópicos e que, por sua vez, cada tópico é um aglomerado de termos, todos eles distribuídos em diferentes proporções.

Historicamente, pode-se dizer que a origem da modelagem de tópicos remonta ao esquema *tf-idf*[9] no sentido calcular a importância de um termo em um determinado documento. Atualmente, os esquemas mais observados na literatura são o *Latent Semantic Allocation* (LSA) encontrado em Deerwester et al. (1990) e o *Latent Dirichlet Allocation* (LDA) proposto por Blei et al. (2003)[10].

Particularmente, o LDA determina a probabilidade *a posteriori* $p(\theta, z, \beta|w)$, sendo que θ representa a distribuição de tópicos por documento, z a distribuição de tópicos por termo, β representa a distribuição de tópicos no corpo de dados e w representa um termo. Devido à sua complexidade, é comum utilizar a notação gráfica encontrada na Figura 4 para a compreensão da composição da distribuição de probabilidade conjunta do modelo. De acordo com essa notação, os nós representam as variáveis aleatórias e os vértices representam as dependências entre variáveis, considerando que nós sombreados indicam variáveis observáveis e os demais as variáveis latentes, ou não-observáveis.

[8] Note que o termo latente é utilizado para indicar elementos que não são diretamente observáveis, mas sim inferidos a partir de variáveis observáveis.
[9] Note que definições mais extensivas do esquema *tf-idf* podem ser encontradas em Salton e McGill (1983).
[10] Note, a título de curiosidade, que ambos apresentam capacidade inferior, mas ainda assim comparável, à capacidade humana de classificação (ANAYA, 2011).

Figura 4 – Representação gráfica do LDA com suavização.



Fonte: Autor "adaptado de" Blei et al. (2003), pg. 1006.

Com isso, considerando um conjunto de tópicos K e um conjunto de documentos D, cada qual com um conjunto de termos N, o objetivo do LDA é inferir, com base nos termos observáveis $w_{d,n}$, a distribuição entre tópicos por termo dada por $z_{d,n}$, a distribuição entre tópicos e documentos dada por θ_d e a distribuição de tópicos ao longo de todo o vocabulário de termos dada por β_k. Note ainda que o LDA possui um grande conjunto de parâmetros que influenciam o seu funcionamento, a exemplo da quantidade de tópicos K e dos parâmetros de concentração α e η aplicáveis às distribuições como probabilidades *a priori*. Há uma série de estudos que avaliam a influência desses parâmetros, a exemplo do trabalho de Binkley et al. (2014).

No contexto da engenharia de software é possível observar o uso do LDA para agrupamento de código-fonte (MALETIC; VALLURI, 1999), detecção de similaridades (MALETIC; MARCUS, 2000), ferramentas de visualização (SAEIDI ET AL., 2015) e até mesmo modelagem de tópicos em larga escala de projetos de software (MARKOVTSEV; KANT, 2017). Já no contexto da sugestão de código-fonte, Nguyen et al. (2013) é o único exemplo a utilizar esse algoritmo para segmentar modelos de linguagem n-grama.

Na grande maioria dos casos é comum processar as unidades de código-fonte para que a modelagem de tópicos exclua palavras vazias, ou *stop words*, e seja realizada utilizando um vocabulário composto por comentários e identificadores. Além disso, também é comum decompor as palavras do vocabulário sempre que elas sejam *multi-termo*, isto é, sempre que elas sejam formadas pela combinação entre um ou mais termos[11]. Todas essas etapas, acrescidas ainda de uma conversão para caixa baixa, são

[11] Como exemplo considere o identificador `minhaVariavelDeControle`, que quando decomposto origina o conjunto de termos {'minha', 'Variavel', 'De', 'Controle'}.

necessárias para que os termos resultantes possam ser submetidos a processos de *stemming* e lematização, que reduzem as inflexões de uma palavra à sua raiz.

Nesse sentido, vale ressaltar que os processos de *stemming* e lematização utilizam técnicas distintas para alcançar o mesmo objetivo. Segundo Manning, Raghavan e Schütze (2008), enquanto o processo de *stemming* considera uma aproximação baseada em heurísticas para determinar a raiz de uma palavra, o processo de lematização utiliza um vocabulário e uma análise morfológica do termo, resultando na determinação da base ou versão de dicionário de uma palavra, que também é conhecido como *lemma*.

Este trabalho está em linha com Nguyen et al. (2013) ao utilizar o LDA, e também em linha com Baldi et al. (2008) e Nguyen et al. (2012) ao considerar que os tópicos latentes inferidos a partir do código-fonte correspondem bem aos objetivos e funcionalidades de um sistema. A diferença chave é que este trabalho considera o agrupamento dos arquivos de código-fonte em conjuntos intra e inter-usuário. O conjunto intra-usuário de arquivos de código-fonte é utilizado para representar os objetivos e funcionalidades inferidos a partir de todos os projetos de software do usuário mantenedor em questão. Já o conjunto inter-usuário de arquivos de código-fonte é utilizado para representar os objetivos e funcionalidades comuns, inferidos a partir de todos os usuários mantenedores, sem distinção.

Ademais, o vocabulário utilizado por este trabalho é composto exclusivamente por comentários e identificadores, processados para tratar a decomposição *multi-termos*. Os termos resultantes são filtrados de acordo com uma lista a exclusão de palavras vazias da língua inglesa e de palavras vazias no contexto da linguagem de programação estudada.

3 MODELO PROPOSTO

O objetivo deste capítulo é apresentar e discutir o modelo de sugestão de código-fonte proposto por este trabalho, iniciando com a formalização do problema e o estabelecimento das restrições impostas conforme observado na seção 3.1. Em seguida, durante a seção 3.2, há o registro da estratégia geral utilizada para endereçar o problema. Ao longo da seção 3.3 encontram-se a apresentação e a formalização do modelo proposto. Por fim, a seção 3.4 discute suas limitações.

3.1 DEFINIÇÃO DO PROBLEMA

Considere um projeto de software P como uma sequência de unidades de código-fonte[12] $U_1 U_2 \dots U_n$ com $|P| > 0$. Cada unidade de código-fonte U_i é composta de um trecho de código-fonte s, que por sua vez é representado por uma sequência de termos $w_1 w_2 \dots w_n$, com $|s| > 0$, $w_i \in \Sigma$ para $i = 1 \dots n$ e com Σ representando o conjunto de símbolos não-terminais da gramática G associada à linguagem de programação[13].

Considere também que cada termo w_i seja formado pela sequência c de caracteres $c_1 c_2 \dots c_j$ com $|c| > 0$. Considere que o termo w^{UNK} seja definido como o símbolo especial `<UNK>`[14] e que o termo w' seja definido como uma sequência c de caracteres com $|c| \geq 0$ acrescida do termo w^{UNK}, isto é, $w' = cw^{UNK}$.

Por fim, considere ainda que o termo w_i da sequência original s tenha sido substituído pelo termo especial w_i' para representar o ponto em que a sugestão de código-fonte deve ocorrer, originando assim a sequência derivada s' com $|s'| = |s|$.

---

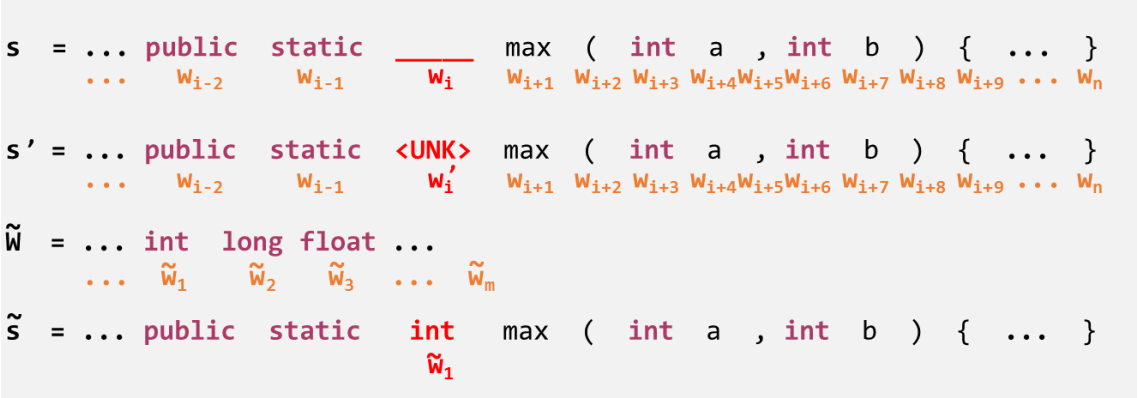
[12] Note que projetos de software geralmente contêm outros elementos além de unidades de código-fonte. No entanto, para o contexto dessa definição, esses elementos são irrelevantes.

[13] Maiores informações sobre gramáticas podem ser encontradas no Apêndice A.

[14] Note que o símbolo `<UNK>` é normalmente utilizado em modelos de linguagem para indicar palavras fora do vocabulário, mas aqui ele está sendo utilizado para representar o ponto em que a sugestão de código-fonte deve ocorrer. Qualquer outro símbolo poderia ser utilizado em seu lugar, desde que fora do vocabulário em questão.

O problema da sugestão de código-fonte, ilustrado na Figura 5, consiste em determinar um conjunto $\widetilde{W}$ de termos candidatos $\widetilde{w}_1\widetilde{w}_2 \ldots \widetilde{w}_m$ com $|\widetilde{W}| \geq 0$ que possam ser utilizados para substituir o termo w_i' na sequência s' e então produzir uma sequência candidata $\tilde{s}$, com $|\tilde{s}| = |s'|$, considerando a premissa de que $\widetilde{w}_i \in \Sigma$ para $i = 1 \ldots m$, exceto para a previsão de termos fora do vocabulário.

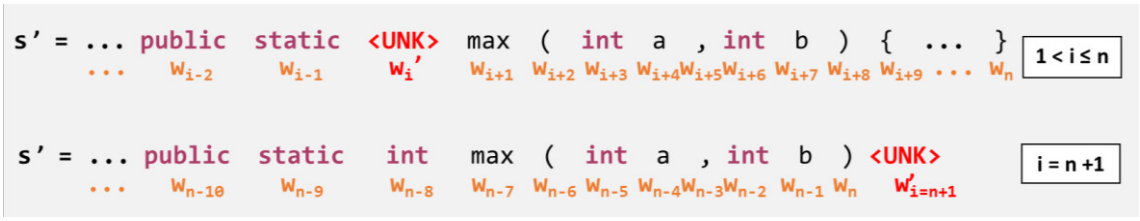
Figura 5 – Representação do problema da sugestão de código-fonte.



Fonte: Autor.

O termo candidato $\widetilde{w}_i$ pode ser incorporado na sequência s' de duas maneiras distintas, ilustradas na Figura 6. Na primeira, $\widetilde{w}_i$ substitui o termo w_i' com $1 < i \leq n$, e assim $|\tilde{s}| = |s|$. Na segunda, $\widetilde{w}_i$ substitui w_i' com $i = n + 1$, e portanto, $|\tilde{s}| > |s|$.

Figura 6 – Representação da incorporação do termo candidato à sequência original.



Fonte: Autor.

Ademais, vale ressaltar que as sequências s e $\tilde{s}$ podem ser completas, ou gramaticais, em um indicativo de que as sentenças podem ser interpretadas com sucesso pela gramática G. No entanto, essa não é uma situação comum ou mesmo uma premissa para a sugestão de código-fonte. O mais comum é que as sequências s e $\tilde{s}$ sejam incompletas. De maneira geral, uma sequência s, seja ela completa ou incompleta, pode originar tanto sequências $\tilde{s}$ completas quanto incompletas.

Por fim, as restrições adicionais ao problema da sugestão de código-fonte são as seguintes:

a) Em uma sequência s' há um único elemento que segue a estrutura de w', ou seja, há uma única previsão por sequência.
b) Se, para o termo $w' = cw^{UNK}$ tem-se $|c| > 0$, trata-se então de um caso especial do problema da sugestão de código-fonte conhecido como problema de complemento de uma expressão parcial. Nele, a lista de termos candidatos $\widetilde{w}_1\widetilde{w}_2 \ldots \widetilde{w}_m$ pode não seguir a premissa $\widetilde{w}_i \in \Sigma$ para $i = 1 \ldots m$.
c) O único objetivo da sugestão de código-fonte é produzir o conjunto $\widetilde{W}$ de termos candidatos $\widetilde{w}_1\widetilde{w}_2 \ldots \widetilde{w}_m$. Assim, excluem-se problemas adicionais, a exemplo da incorporação do termo candidato $\widetilde{w}_i$ na sequência s' para a produção da sequência candidata $\tilde{s}$.

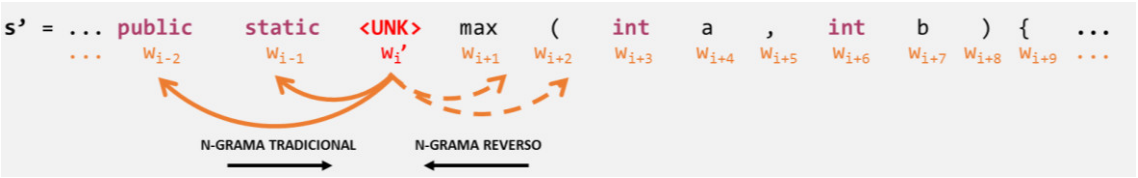
3.2 CONSIDERAÇÕES INICIAIS E VISÃO GERAL DO MODELO

Para endereçar o problema descrito na seção 3.1, a abordagem adotada por este trabalho considera, em linhas gerais, que a capacidade preditiva de um mecanismo de sugestão de código-fonte é influenciada diretamente pelo conjunto de fatores que definem o contexto de execução.

Normalmente, o contexto de execução é primariamente definido como um conjunto de termos prévios ou posteriores ao ponto de invocação, e essa definição pode ocorrer de maneira explícita ou implícita, sendo este último caso sempre concordante com a técnica preditiva utilizada.

Nesse sentido, a Figura 7 ilustra um exemplo de modelo de linguagem 3-grama em que a probabilidade de que um termo candidato $\widetilde{w}_i$ seja observado é determinada pela interpolação entre um modelo de linguagem n-grama tradicional e um modelo de linguagem reverso, isto é, $\lambda_1 P(\widetilde{w}_i | w_{i-1}w_{i-2}) + \lambda_2 P(\widetilde{w}_i | w_{i+1}w_{i+2})$, com $\sum_i \lambda_i = 1$.

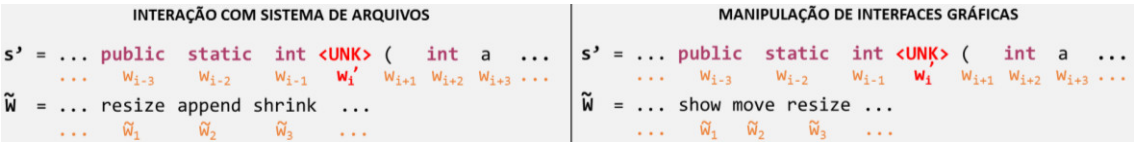
Figura 7 – Exemplo de definição do contexto de execução utilizando modelos de linguagem 3-grama.



Fonte: Autor.

Entretanto, a probabilidade de observação de um termo candidato $\widetilde{w}_i$ também pode ser influenciada pela unidade de código-fonte U em questão e pelo projeto de software P associado, visto que ambos podem conter características marcantes se comparados aos demais. Essa influência pode ainda ser extrapolada para agrupamentos interprojeto de unidades de código-fonte e para agrupamentos de projetos de software, considerando uma divisão de acordo com seus objetivos e funcionalidades. Por exemplo: unidades de código-fonte construídas para interagir com o sistema de arquivos podem apresentar características distintas de unidades de código-fonte construídas para a manipulação de interfaces gráficas, e essas características podem influenciar a capacidade preditiva de um mecanismo de sugestão de código-fonte, conforme ilustrado na Figura 8.

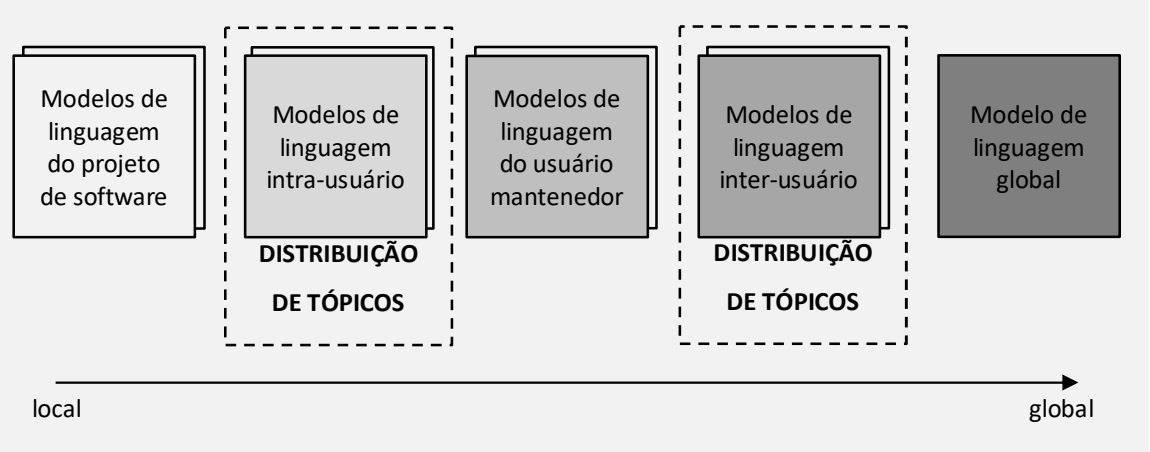
Figura 8 – Exemplo da relevância de características locais mediante características globais.



Fonte: Autor.

Adicionalmente, um mecanismo de sugestão de código-fonte deve ser capaz de tirar proveito das diferenças entre características locais mediante características globais. E para que isso seja possível, a abordagem adotada considera a segmentação hierárquica dos modelos preditivos utilizados de maneira a favorecer os modelos locais sempre que estes estiverem presentes, conforme ilustrado na visão geral da Figura 9. Além do modelo global, os modelos são organizados por projeto de software, por usuário mantenedor e por agrupamentos intra e inter-usuário de projetos de software que compartilham os mesmos objetivos e funcionalidades, considerando que estes tenham sido inferidos a partir de uma modelagem de tópicos latentes.

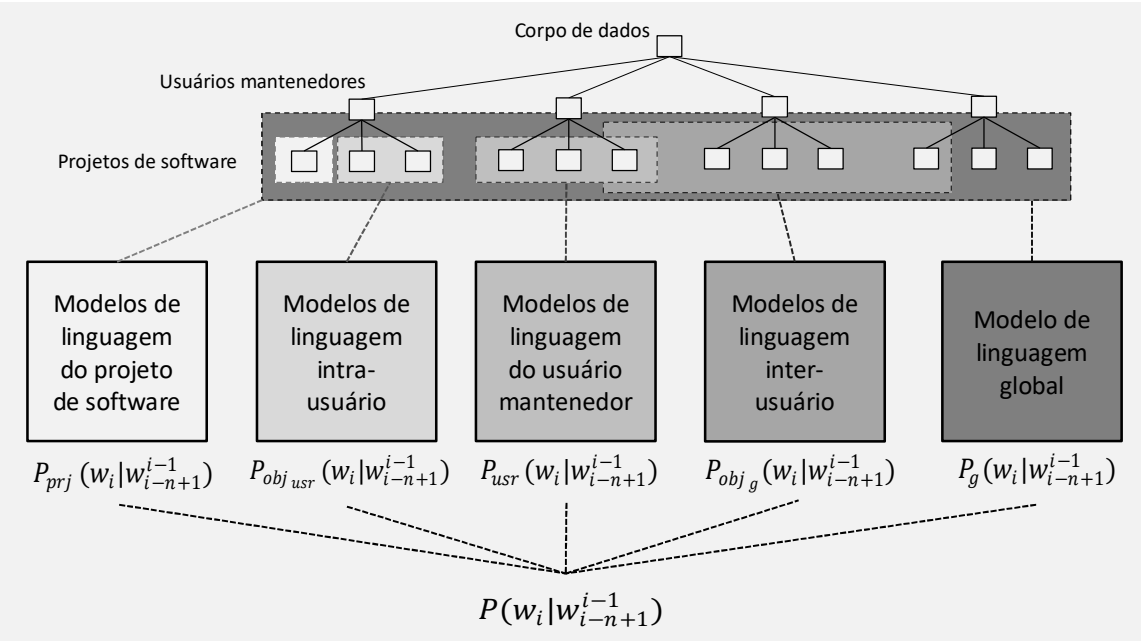
Figura 9 – Visão geral do modelo proposto – segmentação hierárquica.



Fonte. Autor.

Cada um dos modelos preditivos do modelo proposto por este trabalho é gerado de acordo com uma composição particular, ilustrada na Figura 10. Dado um corpo de dados composto por um conjunto de usuários mantenedores, cada qual com seu conjunto de projetos de software, a geração dos modelos de linguagem se dá de diferentes maneiras; ora considerando os arquivos de código-fonte de um único projeto de software, ora todos os arquivos de um determinado usuário mantenedor independente do projeto de software considerado.

Figura 10 – Visão geral do modelo proposto – a composição dos modelos preditivos;



Fonte. Autor.

As explicações detalhadas sobre os elementos formadores do modelo proposto, sobre os parâmetros de entrada e saída, e sobre os demais parâmetros que influenciam seu funcionamento são dadas na seção 3.3 a partir de sua formalização. Por enquanto, basta observar que os modelos de linguagem são segmentados em 5 níveis hierárquicos diferentes; um nível para o projeto de software, representado na Figura 10 pela probabilidade P_{prj}; um nível para os objetivos e funcionalidades intra-usuário mantenedor inferidos a partir de distribuições de tópicos latentes, representado por $P_{obj_{usr}}$; um nível para o usuário mantenedor, representado por P_{usr}; um nível para os objetivos e funcionalidades inter-usuário mantenedor novamente inferidos a partir de distribuições de tópicos latentes, representado por P_{obj_g}; e um nível geral, representado por P_g.

3.3 FORMALIZAÇÃO DO MODELO

A formalização do modelo tem início na seção 3.3.1 com a definição dos parâmetros de entrada e saída e os parâmetros adicionais que determinam seu funcionamento. Em seguida, ao longo das seções 3.3.2 e 3.3.3, são listados os algoritmos que compõem a fase de treinamento e a predição de termos. Por fim, a seção 3.3.4 registra a equação geral do modelo proposto neste trabalho para a predição de termos.

3.3.1 Parâmetros de entrada, saída e hiperparâmetros

O modelo proposto se baseia nos seguintes parâmetros de entrada:

a) A unidade de código-fonte U: sua presença é importante para que se determine a existência de modelos preditivos compartilhados entre unidades com objetivos similares, inferidos a partir de seus comentários e identificadores;
b) O projeto de software P que abriga U: a importância de sua presença é similar ao observado para a unidade de código-fonte U. A partir dele, identificam-se os eventuais modelos preditivos associados ao próprio projeto;
c) O usuário mantenedor do projeto de software P: sua presença é importante para que se determine a existência de modelos preditivos mais abrangentes que os modelos por projeto; e

d) A sequência de termos s': é a partir dessa sequência, concordante com as definições da seção 3.1, que o modelo deve produzir a sequência de termos candidatos.

Em relação aos parâmetros de saída, há um único parâmetro, a sequência de termos candidatos $\widetilde{W}$ com $\left|\widetilde{W}\right| \geq 0$.

Ademais, o modelo proposto também é influenciado por um conjunto de parâmetros diretamente associados às técnicas utilizadas. São eles:

a) Em relação à modelagem de tópicos, destacam-se os seguintes parâmetros: o conjunto de termos do código-fonte utilizados para a inferência de tópicos durante a fase de treinamento, inicialmente restritos aos comentários e aos identificadores; e a quantidade de tópicos K que devem ser inferidos, e que, por ser utilizada para posterior agrupamento dos modelos preditivos, influencia diretamente a quantidade de modelos presentes;
b) Na categoria de parâmetros relacionados aos modelos de linguagem, destacam-se a quantidade de termos considerada, isto é, a ordem no modelo de linguagem n-grama, e o algoritmo de suavização utilizado. Neste caso, e conforme observado no capítulo 4, a configuração padrão do modelo assume o padrão 3-grama com algoritmo de suavização Kneser-Ney.

3.3.2 Treinamento

O modelo proposto requer uma fase de treinamento composta de três etapas para a determinação de todos os modelos de linguagem n-grama utilizados. A primeira etapa, que envolve a determinação das distribuições de tópicos para os agrupamentos intra e inter-usuário, é definida no Algoritmo 1.

Algoritmo 1 – `determinarDistribuiçõesDeTópicos` – Algoritmo para determinação da distribuição de tópicos.

Entrada: C, corpo de dados treinamento formado por um conjunto de usuários mantenedores M cada qual com seu conjunto de projetos P, por sua vez composto por um conjunto de unidades de código-fonte U

Saída: TM_M, conjunto de distribuições de tópicos de cada usuário mantenedor $m \in M$

TM_G, a distribuição global de tópicos, isto é, de todas as unidades de código-fonte U, independente do usuário mantenedor M

início:
    $D_g \leftarrow \emptyset$, conjunto de documentos, cada qual representando a serialização de todas as ASTs disponíveis, independente do usuário mantenedor M

    para cada usuário mantenedor m encontrado em C **faça**
        $D_m \leftarrow \emptyset$, conjunto de documentos, cada qual representando a serialização de todas as ASTs do usuário mantenedor m
        tm_m, distribuição de tópicos do usuário mantenedor m
        para cada projeto $P \in P_m$ **faça**
            para cada unidade de código-fonte $U \in P$ **faça**
                $t \leftarrow$ carregue a AST da unidade de código-fonte U
                $d \leftarrow$ serialize t mantendo apenas os comentários e identificadores
                processe ɖ e filtre os termos (usando o Algoritmo 2)
                $D_m \leftarrow D_m \cup d$
                $D_g \leftarrow D_g \cup d$
            fim
        fim
        $tm_m \leftarrow$ modele tópicos a partir de D_m
        $TM_M \leftarrow TM_M \cup tm_m$
    fim
    $TM_G \leftarrow$ modele tópicos a partir de D_g
    retorne TM_M, TM_G
fim

O Algoritmo 1 considera que os documentos em uma modelagem de tópicos tm são representados pela serialização dos comentários e dos identificadores de uma unidade de código-fonte U. Essas serializações são cadeias de caracteres, ou *strings*, formadas pela concatenação dos valores dos nós de uma árvore sintática abstrata t que tenha sido produzida a partir da interpretação de uma unidade de código-fonte U, conforme linha 10. A leitura dos nós da árvore sintática abstrata t, que ocorre no sentido de cima para baixo e da esquerda para direita, acaba por produzir a serialização d, conforme linha 11. Adicionalmente, conforme linha 12, todas as serializações d são processadas de acordo com o Algoritmo 2 para remoção de palavras vazias, decomposição dos *multi-termos* e realização de processos de lematização, sempre considerando um limite mínimo de comprimento τ para os termos resultantes.

Algoritmo 2 – `serializarASTParaModelagemDeTópicos` – Algoritmo para processamento das serializações da árvores sintáticas considerando filtragem, decomposição de *multi-termos* e processos de lematização.

1 **Entrada**: d, documento que representa um conjunto de termos w gerado a partir da serialização de todos os comentários e identificadores encontrados em uma árvore sintática t

2 **Saída**: d', versão do documento original d após remoção de palavras vazias, decomposição dos multi-termos e realização de processos de lematização

```
3  início:
4      SW, conjunto pré-definido de palavras vazias
5      para cada termo w ∈ d, w ∉ SW faça
6          w' ← divida w em n termos, com n ≥ 1
7          para cada termo w' ∈ W', w' ∉ SW faça
8              se |w'| > τ então
9                  aplique processos de lematização em w'
10             fim
11             anexe w' em d'
12         fim
13     fim
14     retorne d'
15 fim
```

A segunda etapa do treinamento é definida no Algoritmo 3. Ela consiste em determinar os modelos de linguagem n-grama distribuídos de acordo com os tópicos latentes resultantes do Algoritmo 1.

Algoritmo 3 – `calcularModelosDeLinguagemNGramaSegmentadosPorTópicos` – Algoritmo para determinação dos modelos de linguagem n-grama distribuídos de acordo com os tópicos dos projetos de software.

1 **Entrada**: C, corpo de dados treinamento formado por um conjunto de usuários mantenedores M cada qual com seu conjunto de projetos P, por sua vez composto por um conjunto próprio de unidades de código-fonte U
TM_M, conjunto de distribuições de tópicos de cada usuário mantenedor M
TM_G, a distribuição de tópicos de todas as unidades de código-fonte U, independente do usuário mantenedor M

2 **Saída**: $LM_{k,M}$, conjunto de modelos n-grama indexados por tópico dos usuários mantenedores M
$LM_{k,G}$, conjunto de modelos n-grama indexados por tópico das unidades de código-fonte U independente do usuário mantenedor M

3 **início**:
4 **para cada** usuário mantenedor m e tópico $k \in TM_M$ **faça**
5 $S_k \leftarrow \emptyset$, conjunto de serializações das ASTs no tópico k e para o usuário m
6 $D_{k,m} \leftarrow$ carregue os documentos que se enquadram no tópico k do usuário m
7 **para cada** documento $d \in D_{k,m}$ **faça**
8 $U \leftarrow$ unidade de código-fonte que originou d
9 $t \leftarrow$ carregue a AST da unidade de código-fonte U
10 $s \leftarrow$ serialize t
11 $S_k \leftarrow S_k \cup s$
12 **fim**
13 $lm_{k,m} \leftarrow$ calcule modelo n-grama a partir de S_k
14 $LM_{k,M} \leftarrow LM_{k,M} \cup lm_{k,m}$
15 **fim**
16 **para cada** tópico $k \in TM_G$ **faça**
17 $S_k \leftarrow \emptyset$, conjunto de serializações das ASTs para o tópico k
18 $D_k \leftarrow$ carregue os documentos que se enquadram no tópico k
19 **para cada** documento $d \in D_k$ **faça**
20 $U \leftarrow$ unidade de código-fonte que originou d
21 $t \leftarrow$ carregue a AST da unidade de código-fonte U
22 $s \leftarrow$ serialize t
23 $S_k \leftarrow S_k \cup s$
24 **fim**
25 $lm_k \leftarrow$ calcule modelo n-grama a partir de S_k
26 $LM_{k,G} \leftarrow LM_{k,G} \cup lm_k$
27 **fim**
28 **retorne** $LM_{k,M}$, $LM_{k,G}$
27 **fim**

A terceira e última etapa do treinamento é definida no Algoritmo 4. Ela consiste em determinar os modelos de linguagem n-grama por projeto, por usuário mantenedor e global. Seu princípio de funcionamento consiste em interpretar as unidades de código-fonte U, gerar as árvores sintáticas t relacionadas e criar a serialização s, que novamente é produzida no sentido de cima para baixo e da esquerda para direita.

Algoritmo 4 – `calcularModelosDeLinguagemNGrama` – Algoritmo para determinação dos modelos de linguagem n-grama por projeto e global.

Entrada: C, corpo de dados treinamento formado por um conjunto de usuários mantenedores M cada qual com seu conjunto de projetos P, por sua vez composto por um conjunto próprio de unidades de código-fonte U

Saída: LM_P, os modelos n-gramas indexados por projeto P
LM_M, os modelos n-grama indexados por usuário mantenedor M
LM_G, o modelo n-grama para todos os projetos

início:
    $S_g \leftarrow \emptyset$, conjunto geral de serializações das ASTs
    para cada usuário mantenedor m encontrado em C **faça**
        $S_m \leftarrow \emptyset$, conjunto de serializações das ASTs para o usuário m
        para cada projeto $P \in P_m$ **faça**
            $S_p \leftarrow \emptyset$, conjunto de serializações das ASTs para o projeto P
            para cada unidade de código-fonte $U \in P$ **faça**
                $t \leftarrow$ carregue a AST da unidade de código-fonte U
                $s \leftarrow$ serialize t
                $S_p \leftarrow S_p \cup s$
                $S_m \leftarrow S_m \cup s$
                $S_g \leftarrow S_g \cup s$
            fim
            $lm \leftarrow$ calcule modelo n-grama a partir de S_p
            $LM_P \leftarrow LM_P \cup lm$
        fim
        $lm \leftarrow$ calcule modelo n-grama a partir de S_m
        $LM_M \leftarrow LM_M \cup lm$
    fim
    $LM_G \leftarrow$ calcule modelo n-grama a partir de S_g
    retorne LM_P, LM_M, LM_G
fim

3.3.3 Predição dos termos candidatos

A predição dos termos candidatos, ilustrada no Algoritmo 5, está diretamente relacionada ao formato do termo w', lembrando que $w' = c_1 c_2 \dots c_j w^{UNK}$ com $j \geq 0$, e deve ser entendida como o resultado da combinação de técnicas de modelagem de tópicos para determinação dos modelos de linguagem utilizados.

Algoritmo 5 – `preverTermosCandidatos` – Algoritmo geral para a predição de termos candidatos.

Entrada: s', a sequência de termos que deve ser completa, derivada da sequência original s, com $|s'| \geq 0$
U, a unidade de código-fonte que contém a sequência original s
P, o projeto de software que contém a unidade de código-fonte U
M, o usuário mantenedor
TM_M, conjunto de distribuições de tópicos para o usuário mantenedor M
TM_G, a distribuição de tópicos de todas as unidades de código-fonte U, independente do usuário mantenedor M

Saída: $\widetilde{W}$, a lista de termos candidatos, com $|\widetilde{W}| \geq 0$
início:
    LM_G ← carregue o modelo n-grama global
    LM_P ← carregue o modelo n-grama associado à P
    LM_M ← carregue o modelo n-grama associado à M
    t ←carregue a AST da unidade de código-fonte U
    d ← serialize t mantendo apenas os comentários e identificadores
    processe ɖ e filtre os termos (usando o Algoritmo 2)
    k_m ← determine o tópico mais relevante em TM_M com base em ɖ
    k_g ← determine o tópico mais relevante em TM_G com base em ɖ
    $LM_{k,M}$, $LM_{k,G}$ ← carregue os modelos n-grama associados à k_m e k_g
    $\widetilde{W}$ ← combine os termos previstos por LM_P LM_M, $LM_{k,M}$, $LM_{k,G}$ e LM_G
    retorne $\widetilde{W}$
fim

3.3.4 Equação geral para predição de termos

Em virtude do exposto nas seções anteriores, especialmente na seção 3.3.3, a probabilidade de que um termo w_i seja observado após os w_{i-n+1}^{i-1} termos anteriores é determinada pela combinação dos diferentes modelos de linguagem que compõem cada nível hierárquico, conforme linha 13 do Algoritmo 5.

Há, no entanto, muitas maneiras de combinar modelos de linguagem, desde a interpolação linear estudada no capítulo 4, a interpolação não-linear, e até mesmo mecanismos de *backoff*, nos quais os modelos com probabilidade abaixo de um limiar pré-determinado são descartados em favor dos demais modelos. Assim, a equação geral que descreve o modelo proposto no sentido da predição de termos é dependente da maneira escolhida para combinar os modelos de linguagem.

Para fins ilustrativos, a Equação 13 considera a alternativa de interpolação linear para combinar os modelos de linguagem, de tal forma que cada um dos termos da equação representa um nível da hierarquia de contextos.

$$P\left(w_i|w_{i-n+1}^{i-1}\right) = \lambda_{prj}P_{prj}\left(w_i|w_{i-n+1}^{i-1}\right) + \lambda_{obj_{usr}}P_{obj_{usr}}\left(w_i|w_{i-n+1}^{i-1}\right) + \lambda_{usr}P_{usr}\left(w_i|w_{i-n+1}^{i-1}\right) + \lambda_{obj_g}P_{obj_g}\left(w_i|w_{i-n+1}^{i-1}\right) + \lambda_g P_g\left(w_i|w_{i-n+1}^{i-1}\right) \quad (13)$$

com a restrição $\sum_k \lambda_k = 1$

O uso da interpolação linear permite que qualquer nível formador do contexto seja favorecido, novamente conforme estudado no capítulo 4. Ademais, essa abordagem é flexível o suficiente para permitir que uma quantidade diferente de níveis seja considerada, generalizando o modelo para a Equação 14.

$$P\left(w_i|w_{i-n+1}^{i-1}\right) = \sum_k \lambda_k P_k\left(w_i|w_{i-n+1}^{i-1}\right) \quad (14)$$

Por fim, vale ressaltar que qualquer algoritmo de suavização, a exemplo das opções listadas no capítulo 2 (seção 2.2), pode ser utilizado pelos modelos de linguagem n-grama formadores do modelo completo. Pode-se ainda considerar que cada nível da hierarquia utilize algoritmos de suavização distintos, ou ainda algoritmos iguais mas com hiperparâmetros distintos, tais como a ordem do modelo de linguagem n-grama. Essas opções são discutidas ao longo do capítulo 4.

3.4 LIMITAÇÕES DO MODELO PROPOSTO

A primeira limitação está relacionada ao fato de que o modelo proposto desconsidera a gramática da linguagem de programação em questão. Isso ocorre pois os modelos preditivos formadores do modelo proposto são os tradicionais modelos de linguagem n-grama, que são classificados como técnicas essencialmente léxicas. Trata-se, portanto, de uma limitação comum a trabalhos que utilizam modelos n-grama, tais como Allamanis e Sutton (2013), Tu, Su e Devanbu (2014) e Hellendoorn e Devanbu (2017). No entanto, as informações sintáticas notadamente desempenhariam papel fundamental em qualquer assistente de sugestão de código-fonte, pois auxiliariam na tarefa de sugerir termos que produzissem código-fonte sintaticamente inválido.

A segunda limitação trata da dificuldade preditiva de palavras fora do vocabulário. Essa limitação pode ser endereçada de diversas maneiras, incluindo os próprios algoritmos de suavização, que visam evitar que sequências de termos não observadas previamente sejam subestimadas (JURAFSKY; MARTIN, 2009), modelos de linguagem baseados em caracteres ou ainda redes neurais recorrentes com arquitetura *Long Short-Term Memory* (LSTM). Esta dificuldade preditiva de palavras fora do vocabulário é uma das características dos modelos de linguagem que ganha certa proporção em virtude de sua importância no contexto de linguagens de programação, uma vez que palavras fora do vocabulário podem ser facilmente mapeadas para uma classe especial de termos conhecida como identificadores, que são utilizados para definir nomes de tipos, variáveis ou mesmo métodos. De fato, a análise de um modelo de linguagem n-grama construído a partir de uma grande quantidade de projetos de software, conforme Allamanis e Sutton (2013), constata que grande parte do vocabulário é composta pela classe de termos de identificadores.

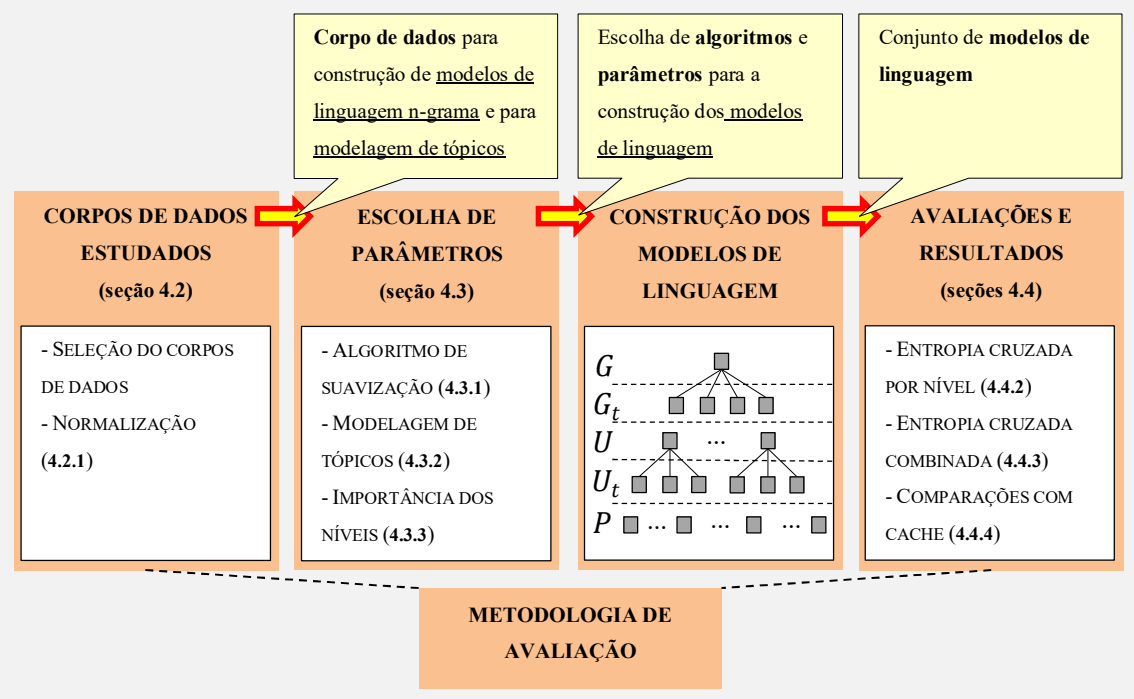
4 AVALIAÇÕES E RESULTADOS

O objetivo deste capítulo é apresentar e discutir as avalições e os resultados obtidos com o modelo de sugestão de código-fonte proposto por este trabalho, iniciando com a seção 4.1, que descreve a metodologia de avaliação utilizada. Em seguida, a seção 4.2 discute a composição do corpo de dados utilizados nas avaliações. Na seção 4.3 são registradas as questões relativas à validação e escolha de parâmetros utilizados para a composição do modelo. Durante a seção 4.3.3 são apresentadas as avaliações em si bem como seus resultados. A seção 4.5 registra notas importantes sobre as avaliações e, por fim, a seção 4.6 concentra uma discussão sobre os resultados obtidos.

4.1 METODOLOGIA DE AVALIAÇÃO

A metodologia de avaliação ilustrada na Figura 11 foi elaborada com o objetivo de atestar a importância da segmentação hierárquica dos modelos de linguagem n-grama para a capacidade preditiva de um modelo de sugestão de código-fonte. Nota-se a relação direta das etapas da metodologia com as seções encontradas neste capítulo.

Figura 11 – Representação da metodologia adotada por este trabalho.



Fonte: Autor.

Para atingir seu objetivo, a metodologia avalia a capacidade preditiva dos modelos de linguagem de maneira intrínseca, sempre considerando um particionamento 75%-25% do corpo de dados (75% para treinamento e 25% para validação) sem sobreposição ao longo dos níveis hierárquicos. Isso significa que ao particionar os arquivos de código-fonte pertencentes a um determinado projeto em um conjunto de dados de treinamento e validação, esse mesmo particionamento será propagado a todos os níveis superiores, seja no âmbito do usuário ou no âmbito global. Portanto, se um arquivo de código-fonte foi designado para o conjunto de dados de treinamento, não importa o nível hierárquico que se considere, ele sempre fará parte do conjunto de dados de treinamento.

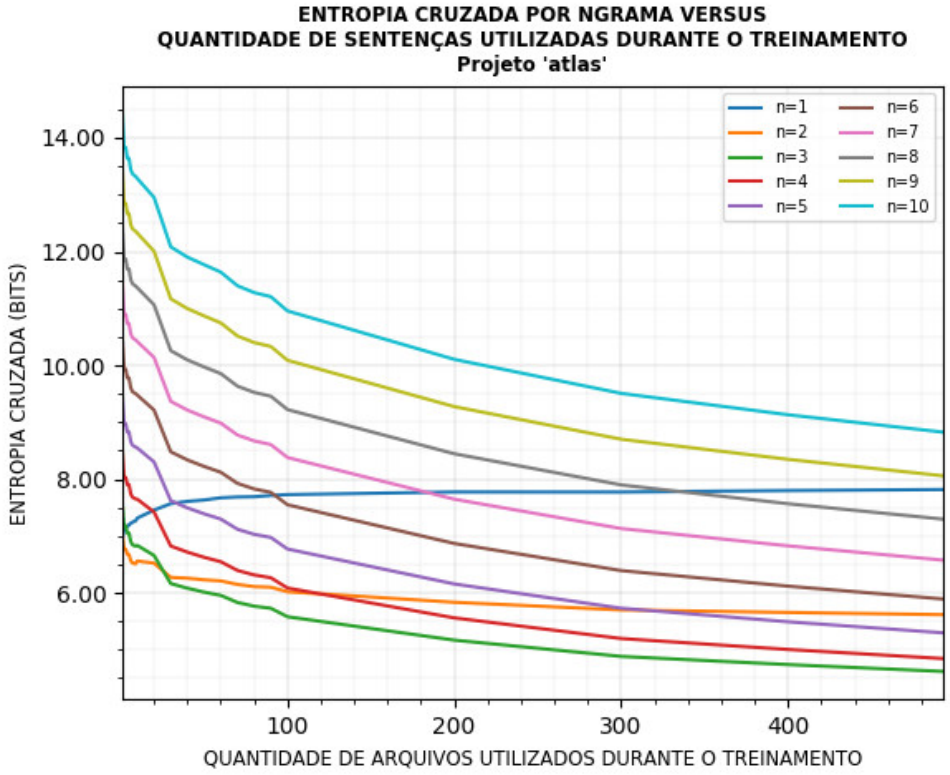
Sempre que aplicável, os modelos estudados são avaliados considerando uma adição gradativa de arquivos de código-fonte ao conjunto de dados de treinamento. Nessas situações, os arquivos são adicionados em lotes definidos pelo conjunto gerador $\{a * 10^d \mid 0 < a \leq 10, 0 \leq d < \text{digitos}(t)\}$ acrescido do total de arquivos disponíveis naquele conjunto t de dados de treinamento. Ademais, a avaliação dos modelos é sempre realizada a partir de todos os arquivos de teste disponíveis.

De maneira geral, as análises intrínsecas seguem o princípio utilizado para a avaliação de modelos de linguagem n-grama, e ajudam a determinar a capacidade preditiva de maneira independente de aplicação. Nesse sentido, considerando que um arquivo de código-fonte representa uma sequência de termos W de tamanho N, as avaliações intrínsecas utilizam medidas de entropia cruzada conforme a Equação 15, com h representando a sequência de termos prévios a w_i. Vale lembrar que, conforme observado na seção 2.2.1, valores baixos de entropia cruzada significam que os termos são facilmente previstos pelo modelo de linguagem.

$$H(W) = -\frac{1}{N}\sum_{i=1}^{N} \log_2 P(w_i|h) \tag{15}$$

A Figura 12, de caráter meramente ilustrativo, exibe a progressão da entropia cruzada à medida que mais arquivos são adicionados ao conjunto de dados de treinamento de um dos projetos encontrado no corpo de dados. Nesse sentido, o comportamento desejado considera medidas de entropia cruzada baixas ou gradativamente mais baixas à medida que mais arquivos são utilizados durante o treinamento.

Figura 12 – Exemplo de medição de entropia cruzada.

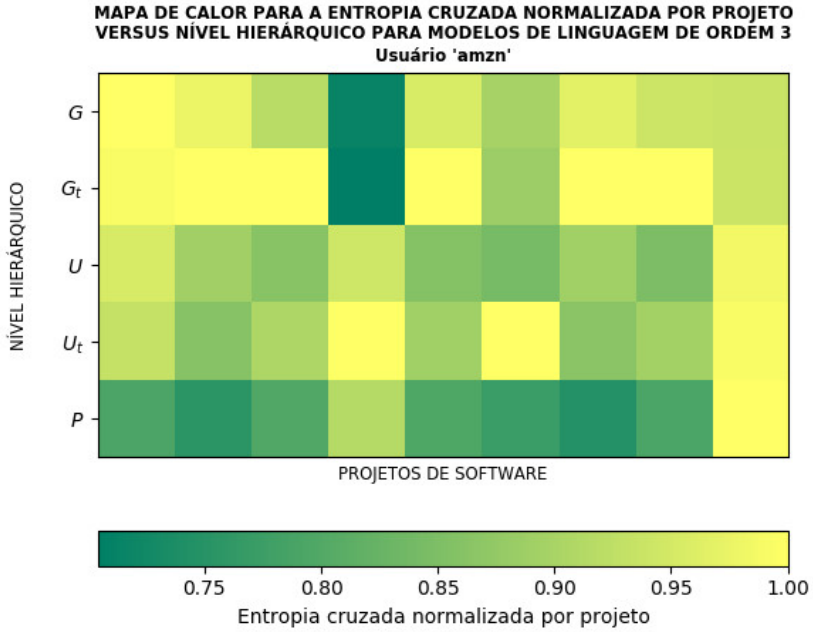


Fonte: Autor.

Por fim, para fins ilustrativos das eventuais melhorias na capacidade preditiva dos modelos estudados, são utilizados mapas de calor inspirados no trabalho de Allamanis e Sutton (2013). Esses mapas, ao invés de representar as probabilidades de um trecho de código-fonte, são utilizados para representar a medida de entropia cruzada em cada um dos projetos de software de cada um dos usuários mantenedores considerados de acordo com o nível hierárquico do modelo de linguagem n-grama analisado.

A Figura 13, de caráter meramente ilustrativo, exibe um exemplo de mapa de calor. Importante notar que, assim como em todos os demais mapas de calor observados a partir deste ponto, P representa o modelo de linguagem construído a partir de projeto de software considerado, U_t representa os modelos de linguagem intra-usuário mantenedor (isto é, divididos por uma distribuição de tópicos latentes dentro do contexto do usuário), U representa o modelo de linguagem construído a partir de todos os projetos daquele usuário, G_t representa os modelos de linguagem inter-usuário mantenedor (isto é, divididos por uma distribuição de tópicos latentes independentemente do usuário) e G representa o modelo de linguagem geral.

Figura 13 – Exemplo de mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico.



Fonte: Autor.

4.2 OS CORPOS DE DADOS

Este trabalho utiliza dois corpos de dados formados a partir dos projetos de software de código aberto em linguagem de programação Java[15] obtidos do repositório de código-fonte online GitHub.

O primeiro corpo de dados, doravante chamado de `CD1`, é composto por 6 grandes usuários mantenedores escolhidos arbitrariamente da iniciativa privada e da comunidade de código aberto, que inicialmente totalizam 490 projetos de software e 157.308 arquivos de código-fonte. Após o processo de normalização, discutido na seção 4.2.1, esses números caem para 489 projetos e 146.610 arquivos, distribuídos conforme observado no Apêndice B, e totalizando 100.702.054 termos.

O segundo corpo de dados, doravante chamado de `CD2`, é derivado do corpo de dados definido por Allamanis e Sutton (2013)[16] e é utilizado para que se determine a aplicabilidade da metodologia de avaliação e a capacidade de generalização da abordagem proposta por este trabalho. O corpo de dados original é frequentemente utilizado na literatura especializada e consiste de 14.807 projetos de software de código-aberto escritos por um conjunto variado de usuários (não definidos explicitamente, mas computado como superior a 9.000), em um total de 2.130.264 arquivos e pouco mais de 1 bilhão de termos. O processo de derivação do corpo de dados original consiste em uma etapa de inferência do usuário mantenedor e uma etapa filtragem para que assim se considerem apenas os usuários que possuam quantidade total de projetos de software igual ou superior à quantidade mínima de projetos do usuário menos expressivo encontrado no primeiro corpo de dados[17]. Após o processo de derivação e consequente normalização, o segundo corpo de dados passa a ser composto por 76 usuários mantenedores que concentram 1.739 projetos e 553.802 arquivos, distribuídos conforme observado no Apêndice B, em um total de 242.795.823 termos.

---

[15] Note que, conforme discutido na seção 5.1 sobre trabalhos futuros, seria possível considerar corpos de dados compostos de projetos de software escritos em linguagens de programação diferentes de Java.
[16] Disponível em http://groups.inf.ed.ac.uk/cup/javaGithub/.
[17] Na prática, esse limite mínimo é de 9 projetos de software. Assim, todos usuários mantenedores com 9 ou mais projetos são utilizados para compor o segundo corpo de dados.

4.2.1 Normalização

O processo de normalização foi aplicado a todos os arquivos de código-fonte coletados inicialmente. Ele consiste na remoção de caracteres inválidos[18] para o interpretador de código-fonte Java `javalang` (JAVALANG, 2018) e na remoção de todos os arquivos que representam unidades de compilação vazias ou não-interpretáveis. Unidades de compilação vazias devem ser entendidas como arquivos de código-fonte que não declaram um único tipo sequer, seja uma interface ou classe. Unidades de compilação não-interpretáveis devem ser entendidas como arquivos com erros de compilação, que não podem ser interpretados de maneira bem-sucedida pela gramática associada à linguagem de programação[19].

A partir desse ponto, o processo de normalização de arquivos se divide em dois segmentos, de acordo com sua aplicação. No primeiro segmento, utilizado para a elaboração dos modelos de linguagem n-grama, todos os eventuais comentários encontrados em uma unidade de compilação são removidos. Já no segundo segmento, utilizado para a elaboração das distribuições de tópicos, apenas os comentários e identificadores resultantes do processo de decomposição de *multi-termos* são mantidos. A exceção fica à cargo dos comentários que representam notificações sobre licenciamento de software geralmente encontradas no topo de uma unidade de compilação.

[18] A exemplo de caracteres nulos e não-ASCII.
[19] Para maiores informações sobre gramáticas e seu papel no contexto das linguagens de programação, vide o Apêndice A.

4.3 VALIDAÇÃO E ESCOLHA DE PARÂMETROS

Nesta seção são registradas as avaliações e os resultados referentes à escolha da ordem e do algoritmo de suavização utilizados pelos modelos de linguagem n-grama que compõem o modelo proposto, bem como as considerações sobre o algoritmo de modelagem de tópicos utilizado para segmentação por objetivos e funcionalidades, e sobre a importância dos níveis hierárquicos na capacidade preditiva. Todas as avaliações encontradas nesta seção são realizadas com o corpo de dados `CD1` definido na seção 4.2.

4.3.1 Algoritmo de suavização

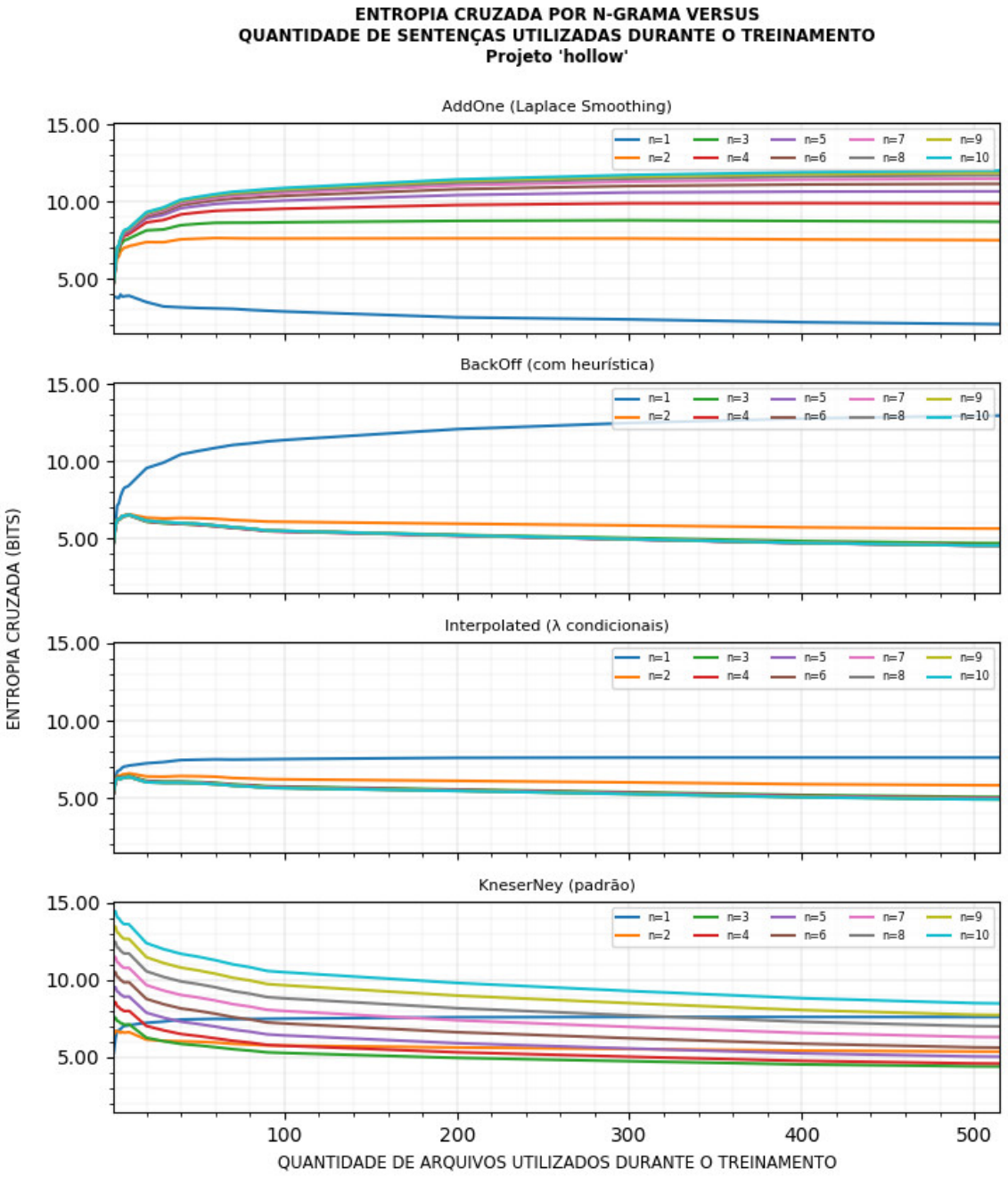
Para que as avaliações sejam realizadas adequadamente, há um conjunto de parâmetros importantes que precisam ser observados e definidos, e que incluem a ordem e o algoritmo de suavização a ser considerado para a construção dos modelos de linguagem n-grama descritos na seção 2.2, bem como seus eventuais hiperparâmetros.

Este trabalho opta por modelos de linguagem n-grama de ordem 3 e pelo algoritmo de suavização Kneser-Ney com taxa de desconto absoluto $\delta = 0.5$. Essa escolha é fundamentada não só pelas indicações bibliográficas, mas também pelas observações extraídas a partir das medições de entropia cruzada descritas a seguir. Ademais, sempre que aplicável, são fornecidas comparações com algoritmos de suavização que usam interpolação com pesos dependentes de contexto. Isso porque Hellendoorn e Devanbu (2017) constatam durante seus experimentos que o Kneser-Ney nem sempre proporciona os melhores resultados para modelos de linguagem n-grama construídos a partir de código-fonte.

Inicialmente, considera-se o projeto `hollow` de propriedade do usuário mantenedor `netflix` e selecionado aleatoriamente do corpo de dados. Ele apresenta as medidas de entropia cruzada ilustradas na Figura 14. Através dessas medidas, observa-se que o algoritmo *add-one smoothing* resulta em medidas altas de entropia cruzada para todas as ordens do modelo de linguagem exceto para ordem 1, ou unigrama. Já os algoritmos de *backoff* e de interpolação de pesos λ dependentes de contexto apresentam resultados muito similares, com medidas de entropia cruzada quase constantes mesmo considerando a progressão da quantidade de arquivos observados. Por fim, o algoritmo Kneser-Ney, também com a exceção do unigrama, resulta em melhorias consistentes e independentes

da ordem do n-grama à medida que mais arquivos são adicionados ao conjunto de dados de treinamento.

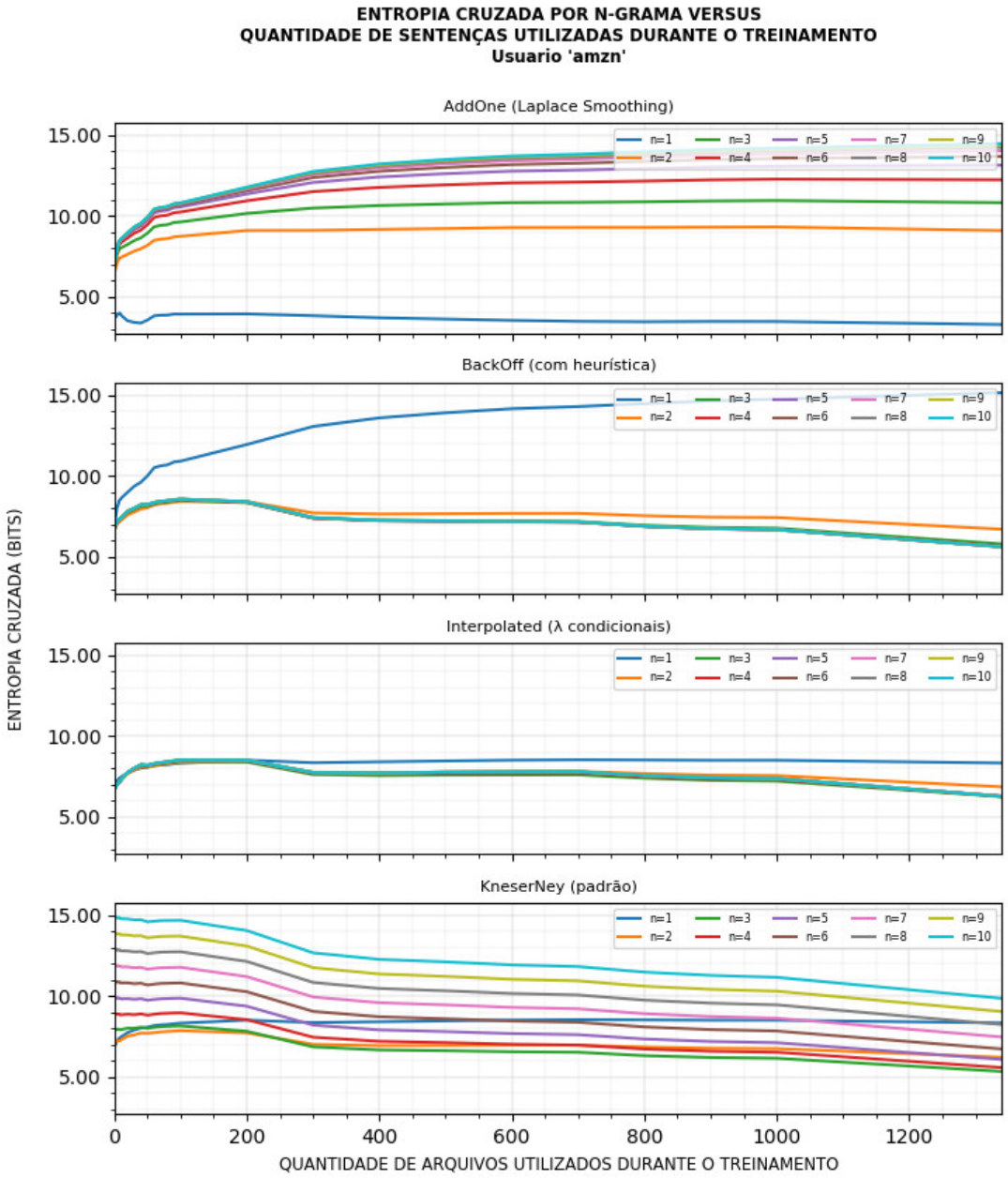
Figura 14 – Entropia cruzada por algoritmo para todos os n-gramas no contexto do projeto ‘hollow’.



Fonte: Autor.

Comportamento similar em relação às medidas de entropia cruzada também pode ser observado quando são considerados todos os projetos de um determinado usuário, a exemplo do ilustrado na Figura 15, que considera o usuário `amzn`.

Figura 15 – Entropia cruzada por algoritmo para todos n-gramas para o usuário ‘amzn’.

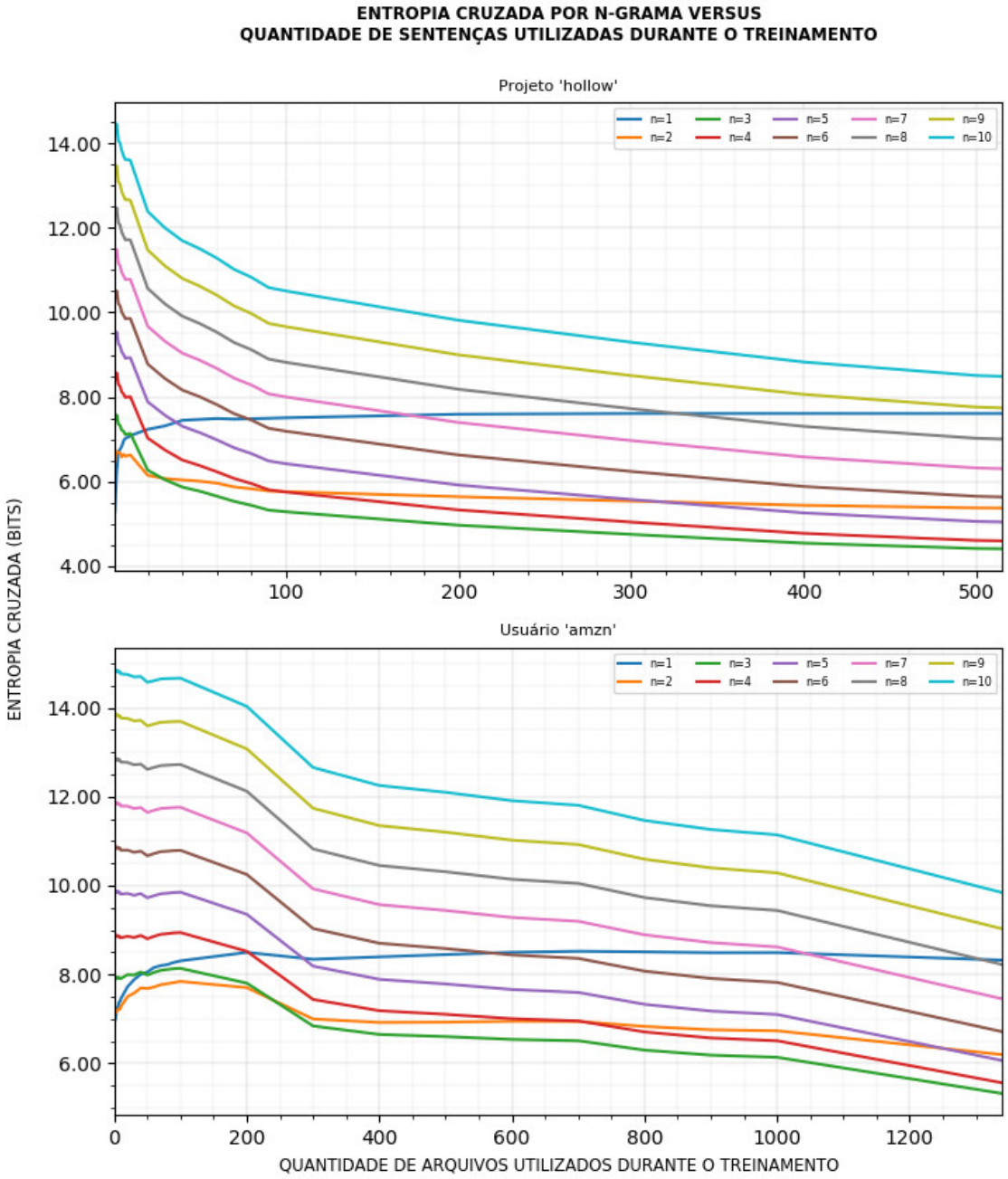


Fonte: Autor.

Todas essas observações são consistentes com as indicações encontradas em grande parte da literatura relacionada de que o algoritmo Kneser-Ney é o mais adequado para a suavização de modelos de linguagem n-grama.

Levando em consideração a escolha do algoritmo, o passo seguinte é a escolha da ordem mais adequada para o modelo de linguagem n-grama. Observações mais detalhadas de ambos os casos mencionados anteriormente sugerem que os n-gramas de ordem 3 apresentam os melhores resultados. Nesse sentido a Figura 16 ilustra as medidas de entropia cruzada com o algoritmo Kneser-Ney para projeto `hollow` pertence ao usuário `netflix` e para o usuário `amzn`.

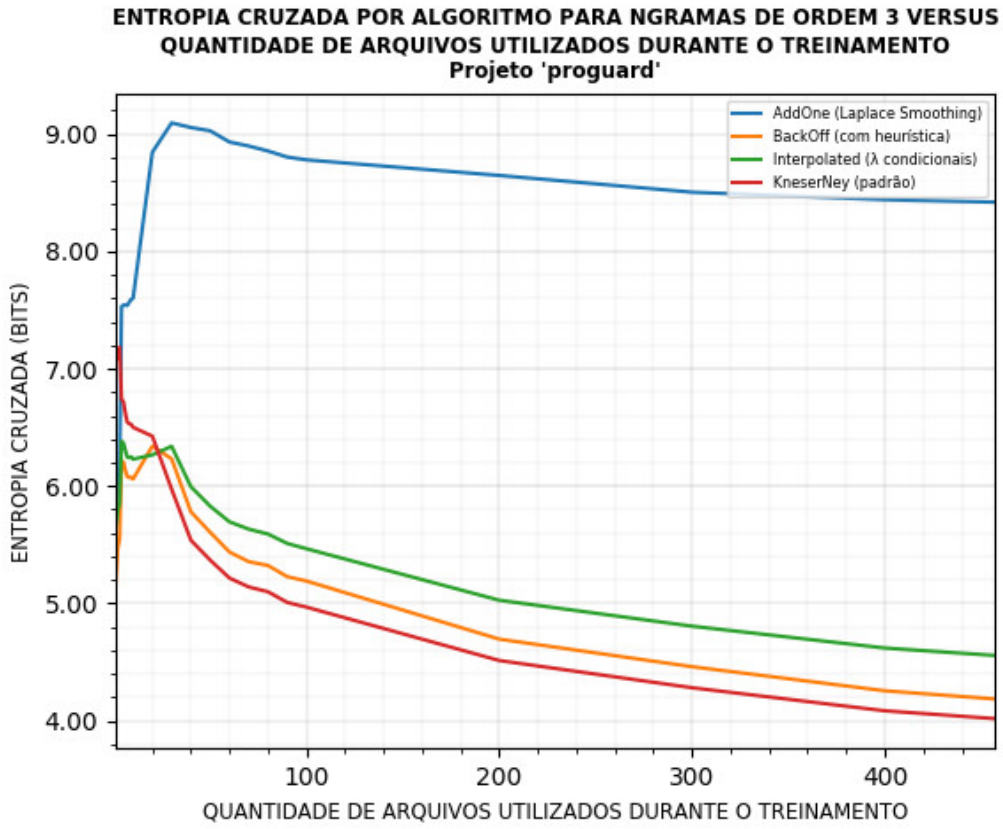
Figura 16 – Entropia cruzada por n-grama para o projeto ‘hollow’ e para o usuário ‘amzn’.



Fonte: Autor.

Evidências adicionais sobre a escolha de n-gramas de ordem 3 podem ser observadas em outros projetos, a exemplo do projeto `proguard`, selecionado aleatoriamente do conjunto de projetos do usuário `facebook` e ilustrado na Figura 17. Nesta figura observam-se comparações entre os diferentes algoritmos de suavização considerados para modelos de linguagem n-grama de ordem 3.

Figura 17 – Entropia cruzada por algoritmo para n-gramas de ordem 3 para o projeto 'proguard'.

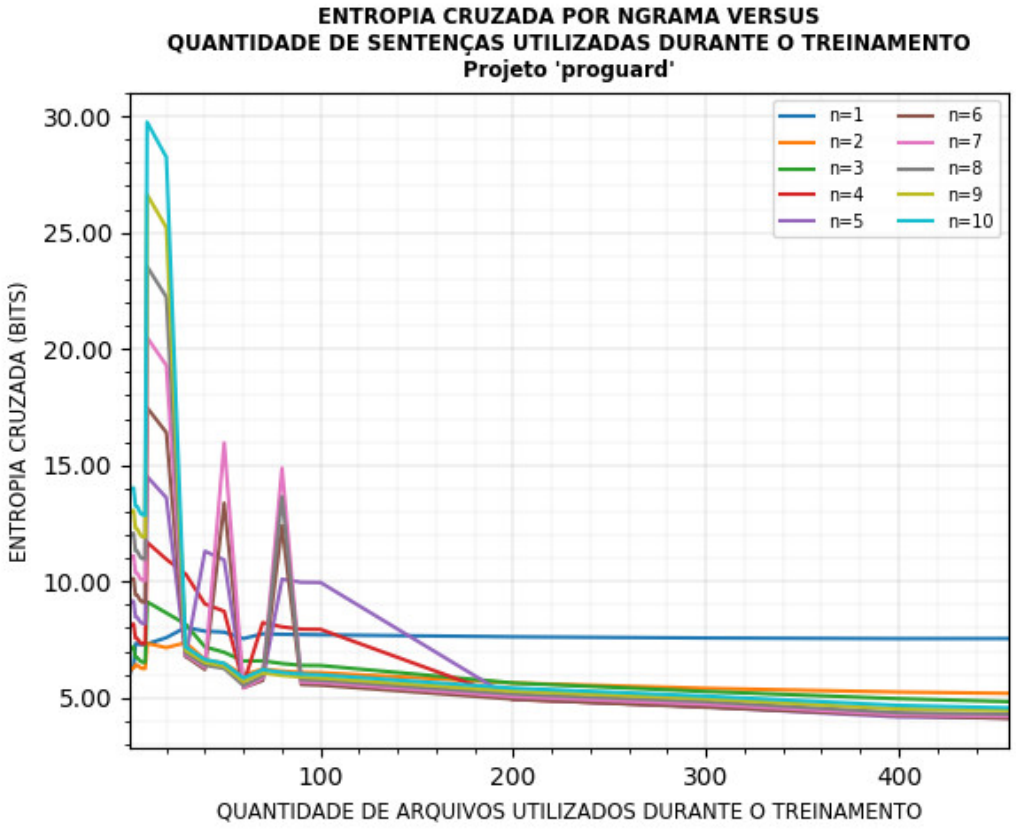


Fonte: Autor.

A última consideração referente à escolha do algoritmo tem relação com seus hiperparâmetros. Conforme indicado na Equação 8, o algoritmo Kneser-Ney tem seu funcionamento baseado em descontos absolutos δ, normalmente calculados com o auxílio da técnica de *held-out*, através da qual um segmento do conjunto de dados de treinamento é utilizado para avaliar o valor do desconto absoluto δ mais adequado, sempre considerando a restrição $0 < \delta \leq 1$.

No entanto, considerando como exemplo o mesmo projeto `proguard` mencionado anteriormente, é possível observar que o uso de *held-out* atinge resultados muito próximos ao uso de um valor pré-definido, conforme ilustrado na Figura 18.

Figura 18 – Entropia cruzada por n-grama considerando held-out para o projeto ‘proguard’.



Fonte: Autor.

4.3.2 Modelagem de tópicos

Para organizar os projetos de acordo com seus objetivos, este trabalho considera que os tópicos latentes extraídos de uma modelagem de tópicos apresentam grande correspondência com os objetivos e funcionalidades de um sistema, conforme observado em trabalhos tais como Baldi et al. (2008), Nguyen et al. (2013) e Markovtsev e Kant (2017).

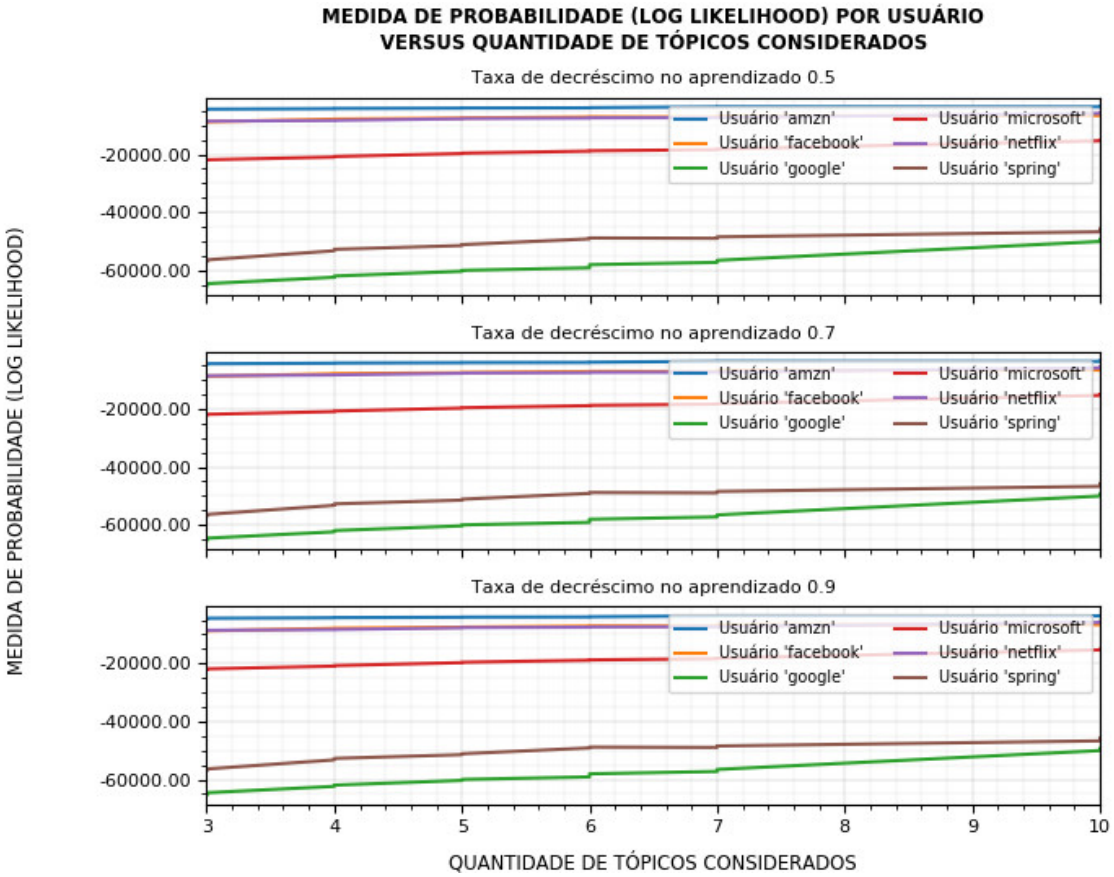
As distribuições de tópicos são construídas a partir de um conjunto de termos extraídos dos comentários e dos identificadores das unidades de código-fonte, que são normalizados e filtrados de acordo com um conjunto de palavras vazias[20]. A normalização é composta de duas etapas principais. Na primeira, motivada pelo trabalho de Markovtsev e Kant (2017), encontra-se a decomposição dos identificadores, em especial os identificadores *multi-termos*, comumente observados em código-fonte escrito na linguagem de programação Java. Nesse cenário, o identificador de exemplo `minhaVariavelDeControle` é decomposto no conjunto de termos $\{minha, variavel, de, controle\}$. Na segunda etapa da normalização encontra-se o processo de lematização, utilizado para determinar a forma inflexionada um termo. Todos os termos resultantes do processo de normalização são filtrados para a remoção de palavras vazias, que incluem palavras reservadas extraídas da sintaxe da linguagem de programação e palavras vazias da língua inglesa. Além disso, em virtude da quebra dos identificadores em um conjunto menor de termos, termos resultantes com comprimento inferior a 5 caracteres também são removidos.

A escolha do algoritmo LDA é motivada pela literatura relacionada, particularmente pelos trabalhos de Nguyen et al. (2013) e de Binkley et al. (2014), sendo que este último observa a importância dos hiperparâmetros utilizados pelo algoritmo, assim como os efeitos decorrentes de variações em seus valores. Com isso, e considerando o interesse particular em uma quantidade de tópicos reduzida, a Figura 19 ilustra a medida de probabilidade *log likelihood* fornecida pelo LDA como resultado de uma busca exaustiva para a modelagem de tópicos não-supervisionada com agrupamentos por

[20] O conjunto final de palavras vazias é definido pela união entre dois conjuntos. O primeiro conjunto considera as palavras vazias da língua inglesa, e é definido pela própria biblioteca de tokenização e lematização utilizada, conforme indicado na seção 4.5. Já o segundo conjunto considera palavras frequentes no contexto da linguagem de programação Java, e é definido pelas palavras-chave (*keywords*) encontradas na gramática associada e pelos termos frequentes {'java','javax', 'com', 'sun', 'object', 'instance', 'method', 'property', 'type', 'value', 'string', 'exception', 'test'}.

usuário e variações da quantidade de tópicos no conjunto $\{3; 4; 5; 6; 7\}$, da taxa de aprendizado no conjunto $\{0{,}5; 0{,}7; 0{,}9\}$ e da quantidade máxima de iterações no conjunto $\{10;\ 20\}$.

Figura 19 – Busca exaustiva pelos hiperparâmetros utilizados pelo algoritmo LDA para a modelagem de tópicos de acordo com o usuário.



Fonte: Autor.

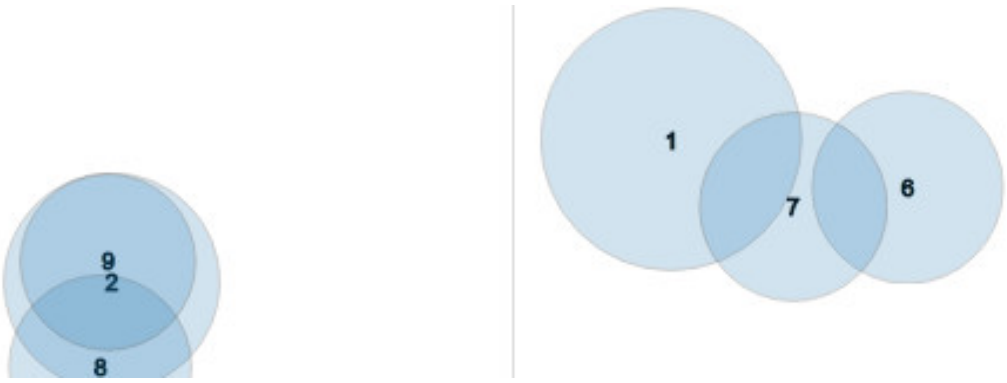
Observa-se que a progressão na quantidade de tópicos considerados é relevante para a medida de probabilidade apenas para os usuários `google` e `spring-projects`. Isso pode ser um indicativo de que o conjunto de dados considerado para os demais usuários é pouco representativo, que os algoritmos de normalização de termos são muito restritivos ou muito abrangentes, ou ainda que a quantidade de tópicos latentes é realmente muito pequena. Assim, em situações como essa, é comum realizar ajustes finos através da visualização dos modelos de tópicos resultantes da execução do algoritmo LDA com o auxílio de ferramentas tais como o LDAvis[21] (SIEVERT; SHIRLEY, 2014).

[21] Disponível em https://github.com/bmabey/pyLDAvis.

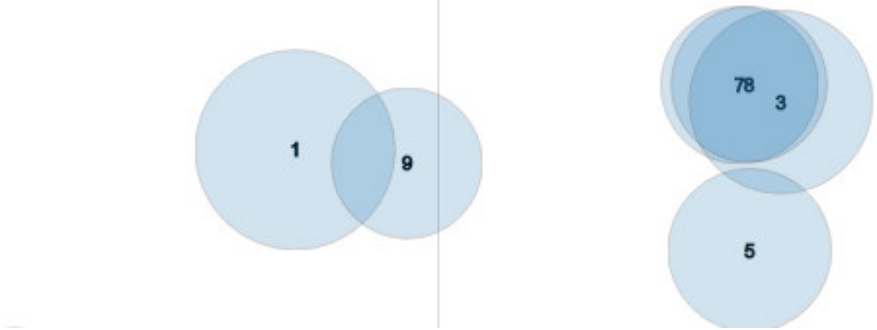
A Figura 20 ilustra o melhor modelo encontrado pela busca exaustiva para os usuários `google` e `spring-projects` no quesito medida de probabilidade. A ferramenta LDAvis produz um diagrama de distância inter-tópico que ajuda a determinar se as distribuições obtidas, ainda que resultantes de uma busca exaustiva, são adequadas. O comportamento esperado para esse diagrama é que os tópicos sejam bem distribuídos, isto é, espera-se que as circunferências tenham diâmetros próximos uns aos outros, sem grandes discrepâncias. Ademais, espera-se que haja pouca ou nenhuma sobreposição de tópicos.

Figura 20 – Exemplo de visualização dos tópicos resultantes do processo de modelagem de tópicos gerados a partir dos dados de treinamento dos usuários ‘google’ e ‘spring-projects’.

(a) Representação do mapa de distância inter-tópico para o usuário ‘google’ gerado pela ferramenta LDAvis.



(b) Representação do mapa de distância inter-tópico para o usuário ‘spring-projects’ gerado pela ferramenta LDAvis.



Fonte: Autor.

Entretanto, pode-se notar a existência de situações em que a alta medida de probabilidade resulta em uma sobreposição indesejada de tópicos. Na Figura 20.a o tópico #2 é um subconjunto do tópico #9 que, por sua vez, apresenta grande intersecção com o

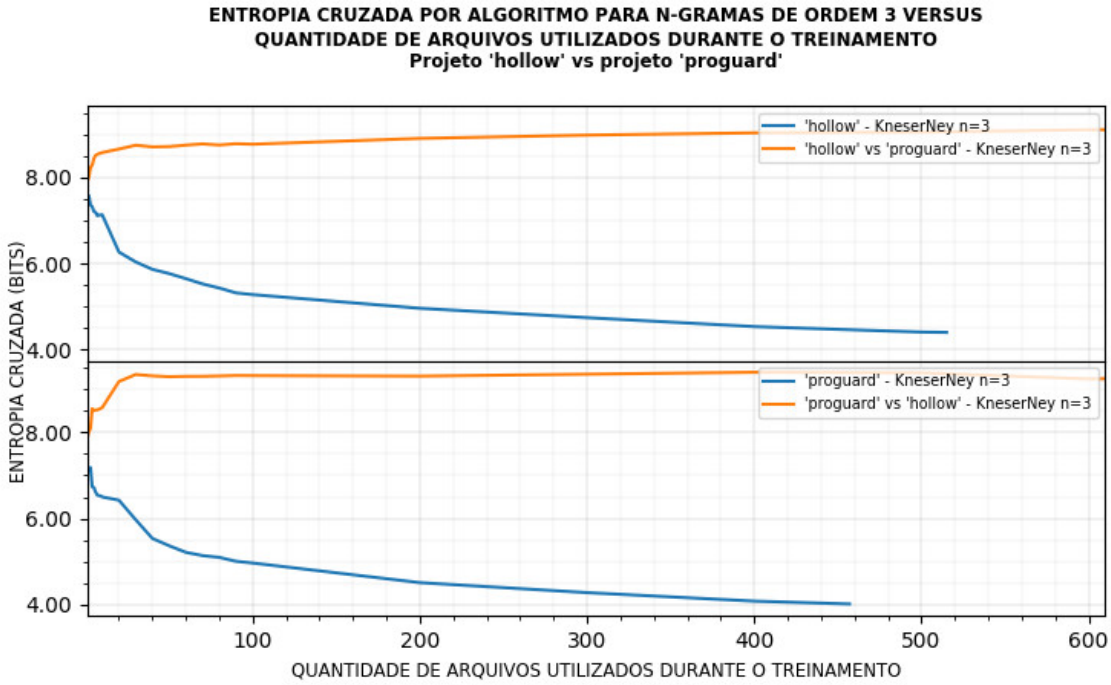
tópico #8. Já na Figura 20.b os tópicos #7 e #8 são muito similares no sentido do termos discriminantes. Assim, entende-se que é necessário abrir mão de medidas de probabilidade mais altas em favor de uma distribuição de tópicos mais enxuta. Portanto, exceto quando explícito, este trabalho considera uma distribuição pré-determinada de 3 tópicos para os modelos intra-usuário e uma distribuição pré-determinada de 4 tópicos para o modelo inter-usuário.

4.3.3 Níveis hierárquicos

Uma vez que um modelo de linguagem n-grama tenha sido construído a partir do conjunto de dados de treinamento de um determinado projeto, é interessante observar como ele se comporta quando aplicado ao conjunto de dados de validação de outro projeto, já que isso pode dar indicações sobre a importância que o elemento projeto tem na hierarquia de definição do contexto.

A Figura 21 ilustra um exemplo para esse cenário, comparando os projetos `hollow` do usuário `netflix` e `proguard` do usuário `facebook` através de duas curvas de medida de entropia cruzada.

Figura 21 – Entropia cruzada para n-gramas de ordem 3 comparando os projetos 'hollow' e 'proguard'.



Fonte: Autor.

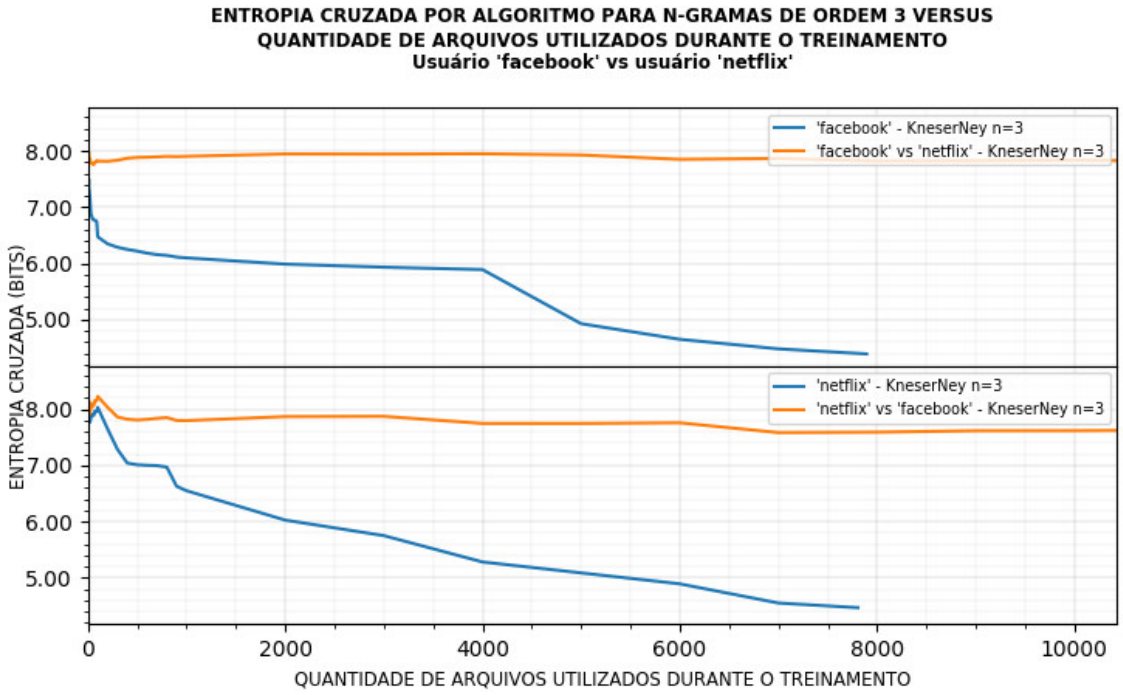
A primeira curva (em cor azul) indica a entropia cruzada intra-projeto (ou seja, os dados de treinamento e de validação são obtidos de um mesmo projeto). A segunda curva (em cor laranja) representa a entropia cruzada interprojeto (ou seja, os dados de treinamento são obtidos de um projeto e os de validação, do outro). Observa-se que as curvas intra-projeto indicam medidas de entropia cruzada mais baixas com a progressão

na quantidade de arquivos observados durante o treinamento. Já as curvas interprojeto indicam medidas de entropia relativamente constantes.

De maneira similar, é interessante observar como um modelo de linguagem construído a partir de todo o conjunto de dados de um usuário específico se comporta quando aplicado todo o conjunto de dados de outro usuário, já que isso pode dar indicações sobre a importância que o elemento usuário tem na hierarquia de definição do contexto.

A Figura 22 ilustra um exemplo para esse cenário, comparando os usuários `facebook` e `netflix` através de duas curvas de medida de entropia cruzada.

Figura 22 – Entropia cruzada para n-gramas de ordem 3 comparando os usuários ‘facebook’ e ‘netflix’.

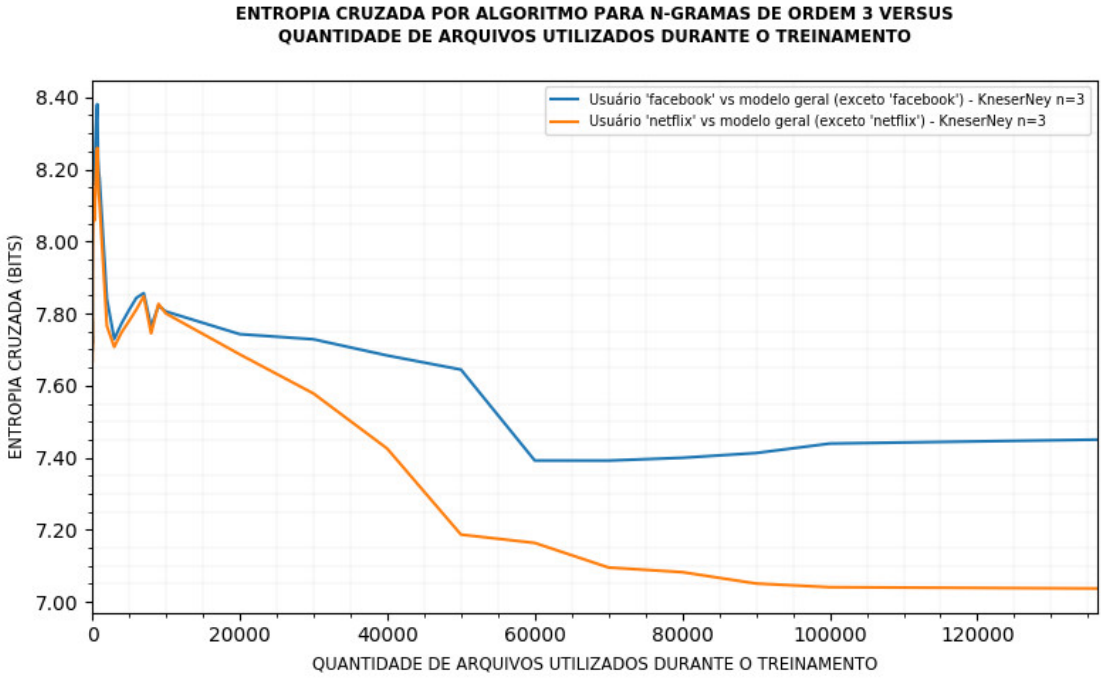


Fonte: Autor.

A primeira curva (em cor azul) indica a entropia cruzada intra-usuário (ou seja, os dados de treinamento e de validação são obtidos de todos os projetos de um mesmo usuário). A segunda curva (em cor laranja) a entropia cruzada inter-usuário (ou seja, os dados de treinamento são obtidos de um usuário e os de validação, do outro). Observa-se que as curvas intra-usuário indicam medidas de entropia cruzada mais baixas com a progressão na quantidade de arquivos observados durante o treinamento. Já as curvas inter-usuário indicam medidas de entropia relativamente constantes.

Em adição, a Figura 23 ilustra como um modelo de linguagem construído a partir de todos os usuários exceto o usuário sob avaliação se comporta quando aplicado ao seu conjunto de dados.

Figura 23 – Entropia cruzada para n-gramas de ordem 3 comparando os usuários ‘facebook’ e ‘netflix’ contra um modelo treinado a partir de todos os demais usuários.



Fonte: Autor.

Todos os casos ilustrados anteriormente sugerem que modelos de linguagem n-gramas mais específicos apresentam resultados de entropia cruzada menores, em um indicativo de capacidade preditiva superior.

4.4 AVALIAÇÕES E RESULTADOS SOBRE O MODELO PROPOSTO

Nesta seção são registradas as avaliações e resultados referentes ao modelo proposto por este trabalho considerando o corpo de dados `CD1` definido na seção 4.2 e os parâmetros definidos na seção 4.3. A seção 4.4.1 analisa a medida de entropia cruzada em cada um dos níveis hierárquicos considerados. Em seguida, a seção 4.4.2 registra as análises referentes à combinação dos modelos diversos de linguagem n-grama considerados na hierarquia. Por fim, na seção 4.4.3 são registradas as comparações do modelo proposto com uma versão adaptada do trabalho de Tu, Su e Devanbu (2014).

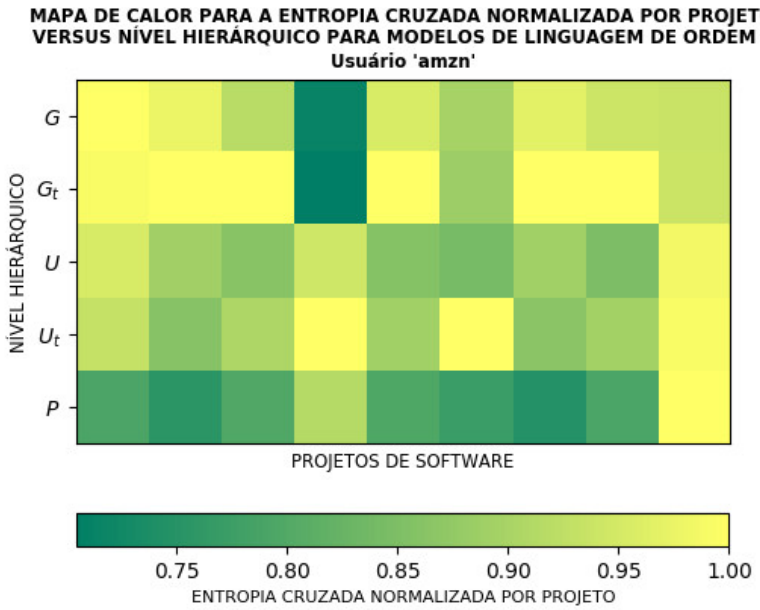
4.4.1 Entropia cruzada em cada um dos níveis hierárquicos

A primeira avaliação consiste em determinar o comportamento da medida de entropia cruzada em cada um dos níveis do contexto hierárquico proposto. Para isso, utilizam-se mapas de calor que representam a medida de entropia cruzada em cada um dos projetos de software do usuário mantenedor considerado de acordo com o nível hierárquico do modelo de linguagem n-grama analisado. Os projetos de software são analisados em sua totalidade, isto é, as medidas de entropia cruzada obtidas fazem referência à todos os arquivos de código-fonte encontrados no conjunto de dados de validação do projeto sob avaliação.

Também é importante notar que, visto que a quantidade de projetos por usuário é variável, apresenta-se um mapa de calor para cada usuário. Adicionalmente, também é importante destacar que a entropia cruzada indicada nestes mapas é normalizada por projeto, permitindo assim uma observação mais uniforme da importância do nível hierárquico. Por fim, vale lembrar que, conforme observado na seção 4.1, medidas de entropia cruzada mais baixas são indicações de um modelo de linguagem com melhor capacidade preditiva.

A Figura 24 representa o mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico considerando exclusivamente o usuário `amzn`. Nota-se que a região inferior do gráfico concentra grande parte das medidas de entropia cruzada baixa, em uma indicação de que os eventos utilizados para avaliação dos modelos de linguagem são mais frequentes em níveis mais específicos, tais como os níveis de projeto (P), de objetivos e funcionalidades intra-usuário (U_t) e usuário (U). Há, no entanto, exceções à essa observação, em uma indicação de que os eventos utilizados são tão frequentes em outros usuários e projetos que os níveis hierárquicos mais genéricos, tais como os níveis de objetivos inter-usuário (G_t) e geral (G), proporcionam as menores medidas de entropia cruzada.

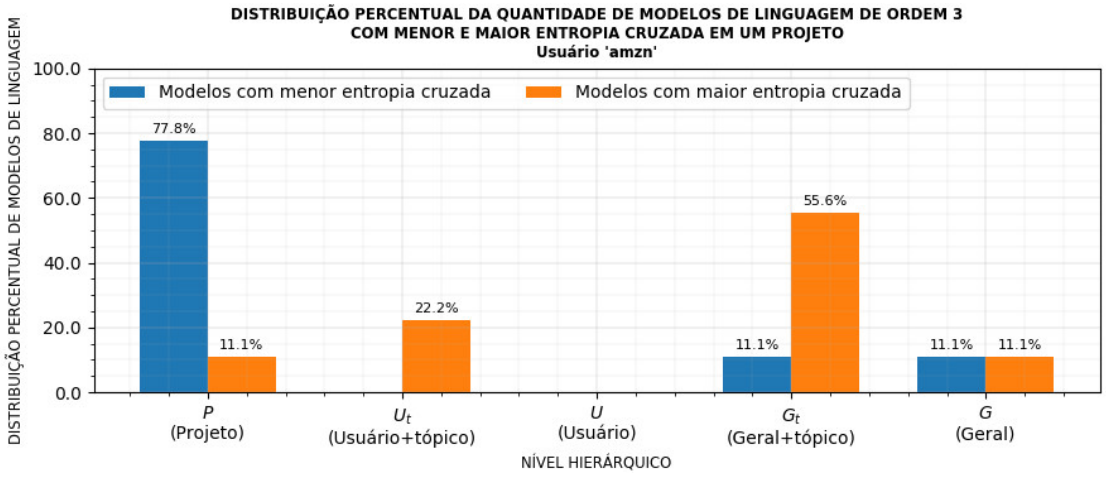
Figura 24 – Mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico – usuário ‘amzn’.



Fonte: Autor.

A Figura 25 representa uma alternativa de visualização ao mapa de calor, e ilustra a distribuição percentual da quantidade de modelos com menor e maior entropia cruzada em um projeto considerando o usuário `amzn`. Observa-se que, dentre todos os projetos avaliados, 77,8% das medidas obtidas a partir dos modelos de linguagem mais específicos (ou seja, construídos a partir do próprio projeto de software) representaram valores mais baixos de entropia cruzada.

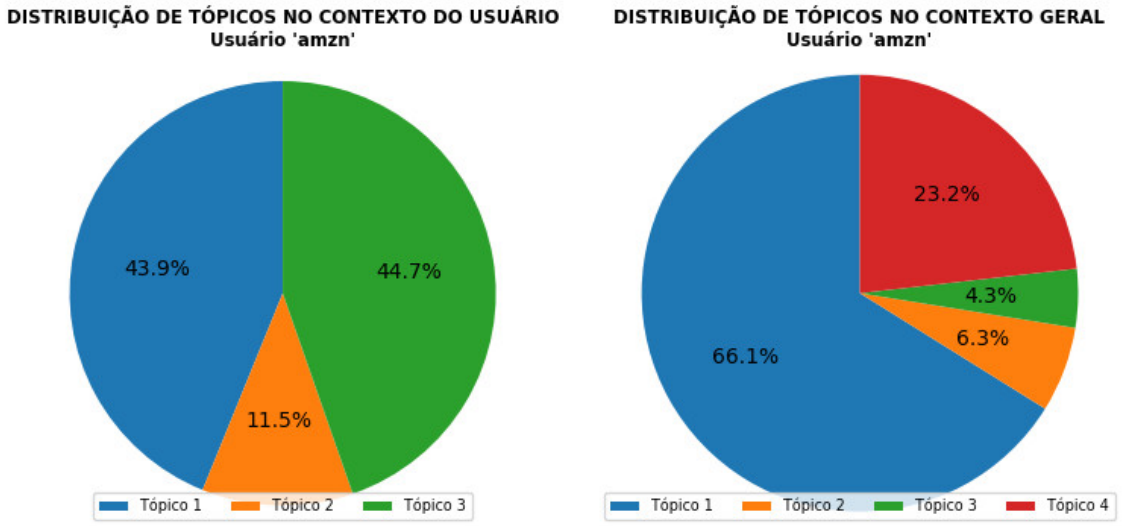
Figura 25 – Distribuição percentual da quantidade de modelos de linguagem de ordem 3 com menor e maior entropia cruzada em um projeto– usuário ‘amzn’.



Fonte: Autor.

Em caráter informacional, a Figura 26 representa a distribuição dos tópicos inferidos a partir do conjunto de dados de validação. No contexto intra-usuário nota-se que grande parte do conjunto de dados de validação se enquadra nos tópicos #1 (43,9%) e #3 (44,7%). Já no contexto geral, grande parte do conjunto de validação se enquadra nos tópicos #1 (66,1%) e #4 (23,2%).

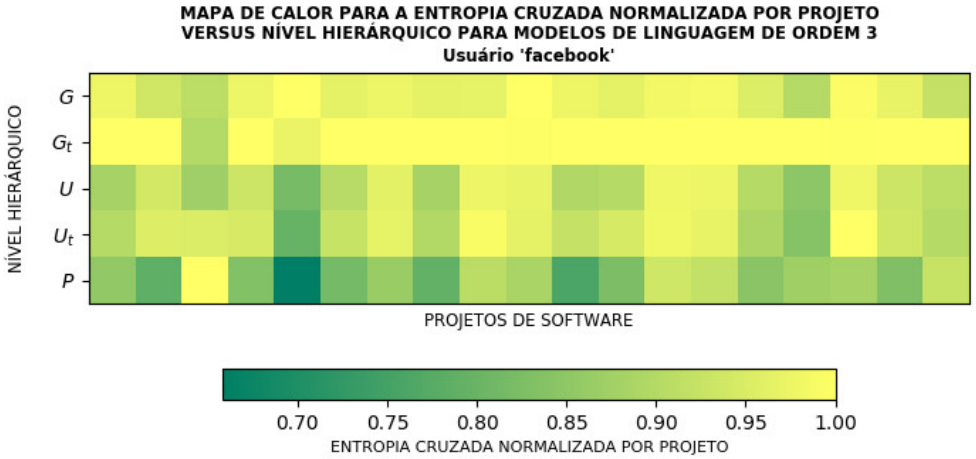
Figura 26 – Distribuição de tópicos inferidos a partir do conjunto de dados de validação para o usuário ‘amzn’.



Fonte: Autor.

A Figura 27 representa o mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para o usuário `facebook`. Novamente, é possível observar que na parte inferior do gráfico, que representa os níveis hierárquicos mais específicos, há uma concentração de medidas de entropia cruzada baixa.

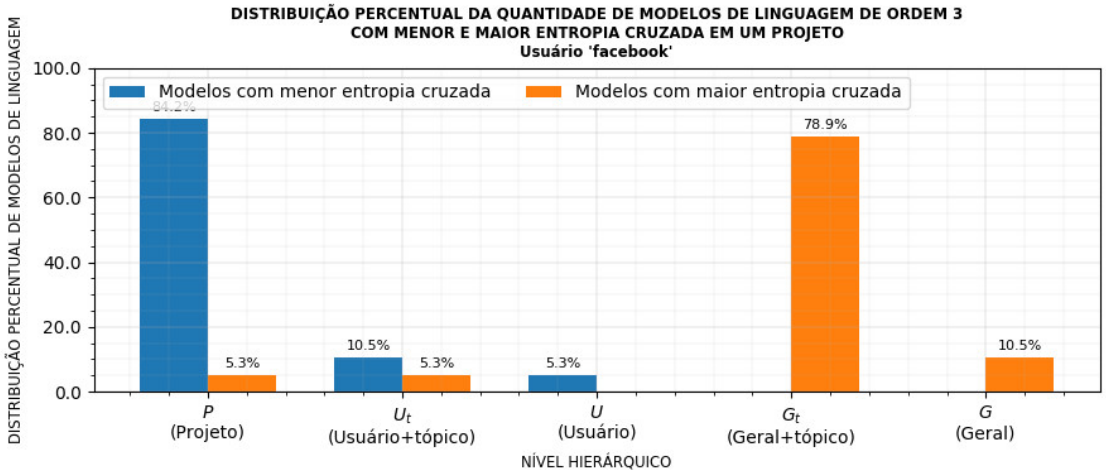
Figura 27 – Mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico – usuário ‘facebook’.



Fonte: Autor.

A Figura 28 evidencia as observações anteriores ao ilustrar a distribuição percentual da quantidade de modelos com menor e maior entropia cruzada em um projeto considerando o usuário `facebook`. Nota-se que, dentre todos os projetos avaliados, 84,2% das medidas obtidas a partir dos modelos de linguagem mais específicos apresentaram valores mais baixos de entropia cruzada. Adicionalmente, observa-se que modelos de linguagem mais genéricos, em especial aqueles divididos por uma distribuição de tópicos latentes inter-usuário, concentraram grande parte dos valores mais altos de entropia cruzada (78,9%).

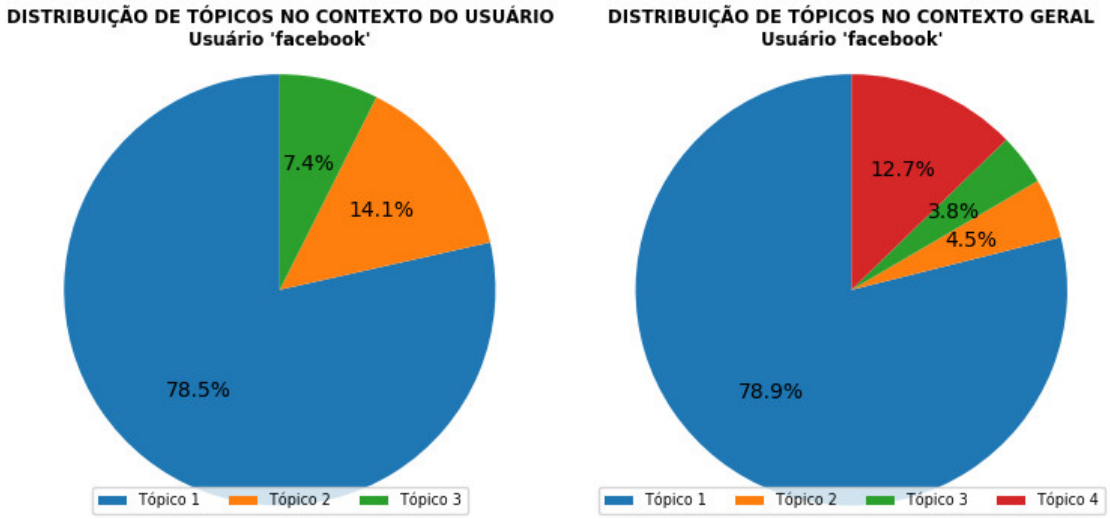
Figura 28 – Distribuição percentual da quantidade de modelos de linguagem de ordem 3 com menor e maior entropia cruzada em um projeto– usuário ‘facebook’.



Fonte: Autor.

A Figura 29 ilustra a distribuição dos tópicos inferidos a partir do conjunto de dados de validação para o usuário `facebook`. Nota-se que tanto no contexto intra-usuário quanto no contexto geral, grande parte do conjunto de dados de validação se enquadra em um único tópico, o tópico #1 (com valores respectivos de 78,5% e 78,9%).

Figura 29 – Distribuição de tópicos inferidos a partir do conjunto de dados de validação para o usuário ‘facebook’.

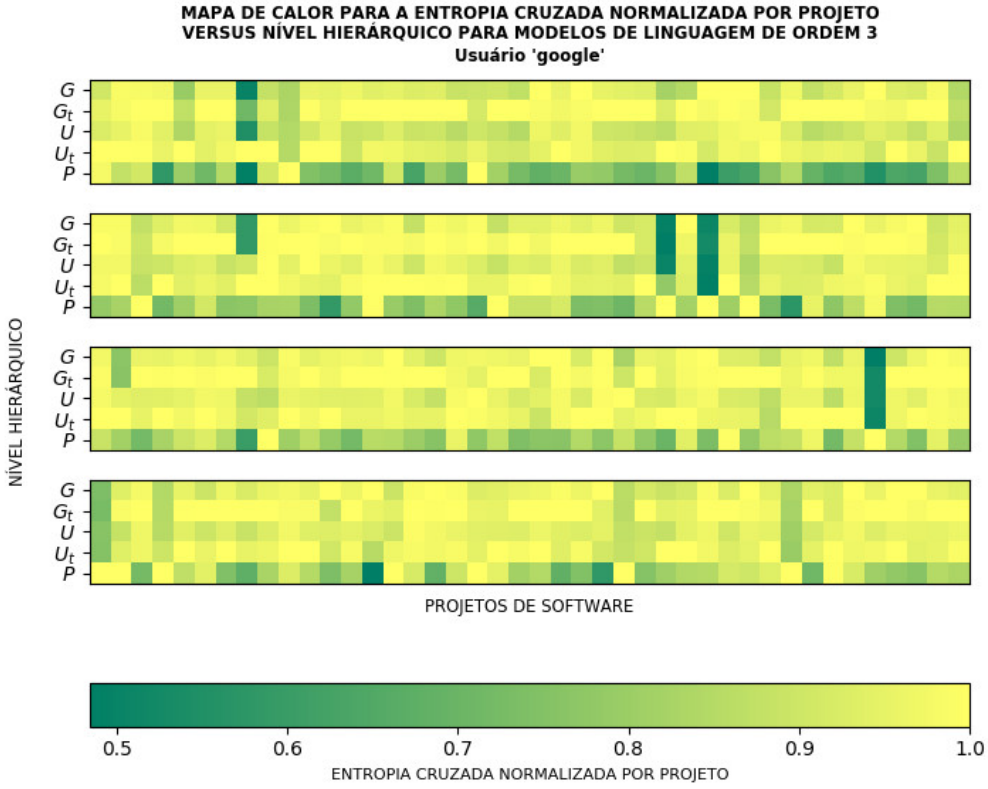


Fonte: Autor.

A Figura 30 representa o mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para o usuário `google`. Nota-se que, ainda que existam

picos de entropia cruzada baixa nos níveis hierárquicos mais gerais, a concentração de medidas de entropia cruzada baixa permanece na parte inferior do gráfico. Ademais, é importante observar que, devido à grande quantidade de projetos que este usuário possui, opta-se por dividir o mapa de calor em segmentos de mesmo comprimento.

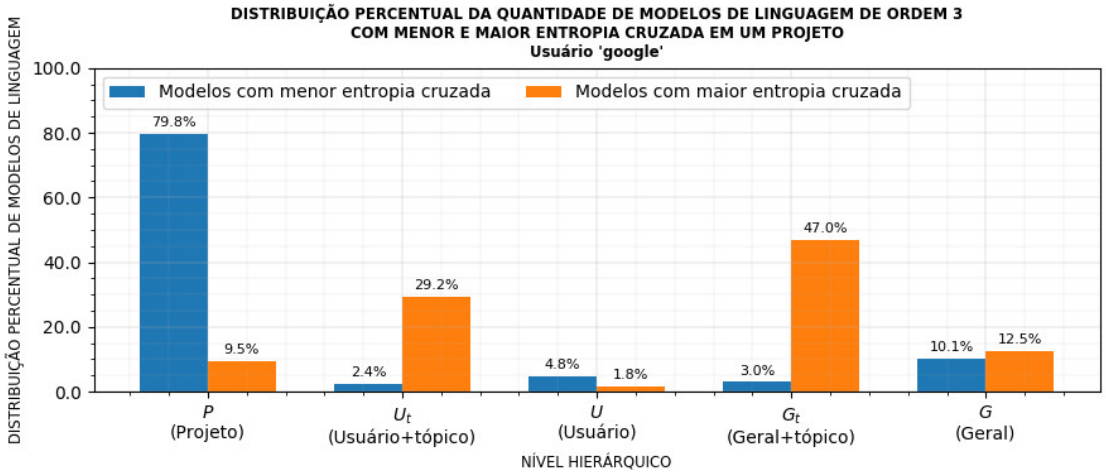
Figura 30 – Mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico – usuário ‘google’.



Fonte: Autor.

A Figura 31 ilustra a distribuição percentual da quantidade de modelos com menor e maior entropia cruzada em um projeto considerando o usuário `google`. Observa-se que, dentre todos os projetos avaliados, 79,8% da medidas obtidos a partir dos modelos de linguagem mais específicos apresentaram valores mais baixos de entropia cruzada.

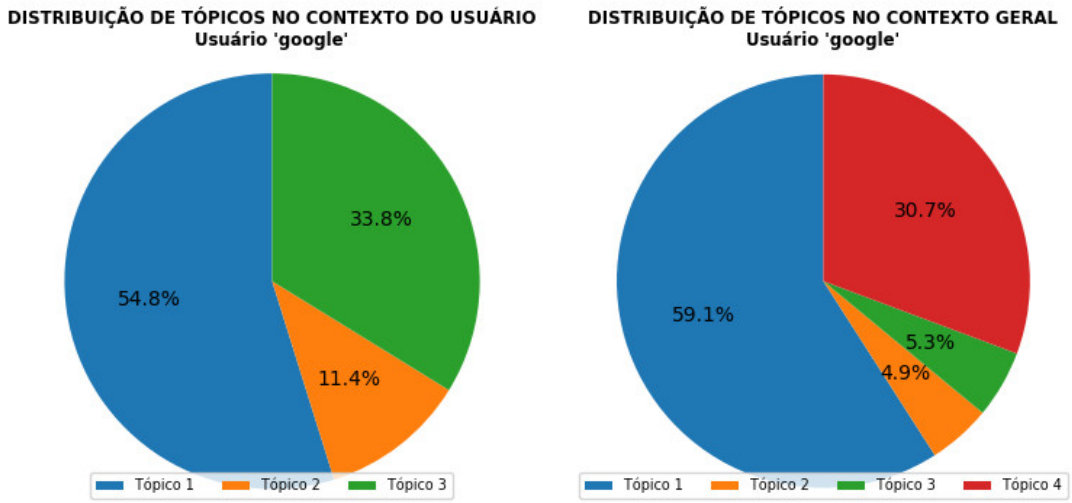
Figura 31 – Distribuição percentual da quantidade de modelos de linguagem de ordem 3 com menor e maior entropia cruzada em um projeto– usuário ‘google’.



Fonte: Autor.

A Figura 32 representa a distribuição dos tópicos inferidos a partir do conjunto de dados de validação. No contexto intra-usuário nota-se que grande parte do conjunto de dados de validação se enquadra nos tópicos #1 (54,8%) e #3 (33,8%). Já no contexto geral, grande parte do conjunto de validação se enquadra nos tópicos #1 (59,1%) e #4 (30,7%).

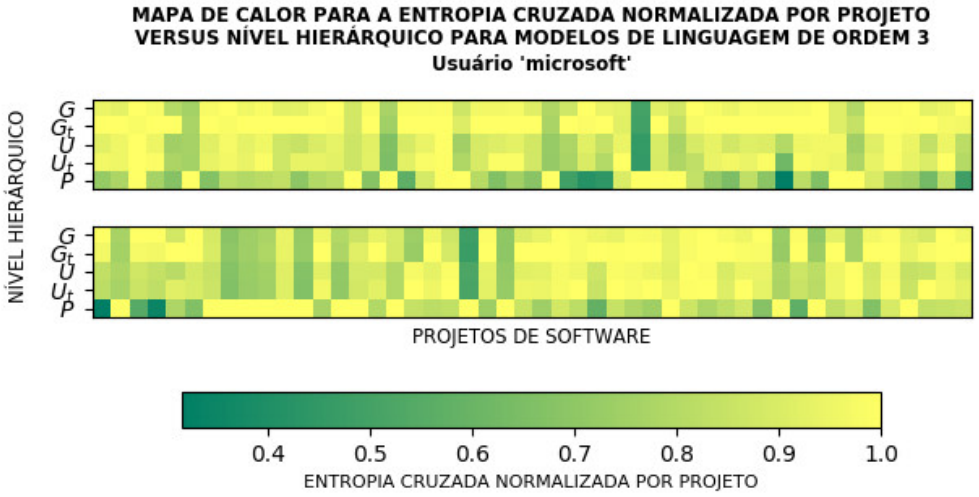
Figura 32 – Distribuição de tópicos inferidos a partir do conjunto de dados de validação para o usuário ‘google’.



Fonte: Autor.

A Figura 33 representa o mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para o usuário `microsoft`. Novamente, ainda que existam picos de entropia cruzada baixa nos níveis hierárquicos mais gerais, a concentração de medidas de entropia cruzada baixa permanece na parte inferior do gráfico.

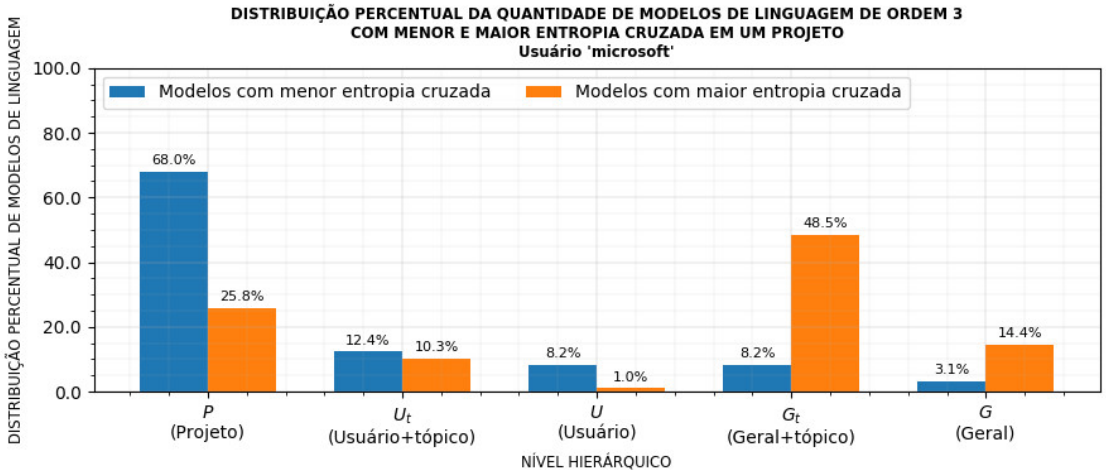
Figura 33 – Mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico – usuário 'microsoft'.



Fonte: Autor.

A Figura 34 evidencia as observações anteriores ao ilustrar a distribuição percentual da quantidade de modelos com menor e maior entropia cruzada para o usuário `microsoft`. Nota-se uma distribuição relativamente mais homogênea quando comparada aos demais usuários observados até o momento. De todos os projetos avaliados, 68,0% das medidas obtidas a partir dos modelos de linguagem mais específicos apresentaram valores mais baixos de entropia cruzada. Por outro lado, modelos de linguagem baseados no projeto concentraram 25,8% das medidas mais altas de entropia cruzada.

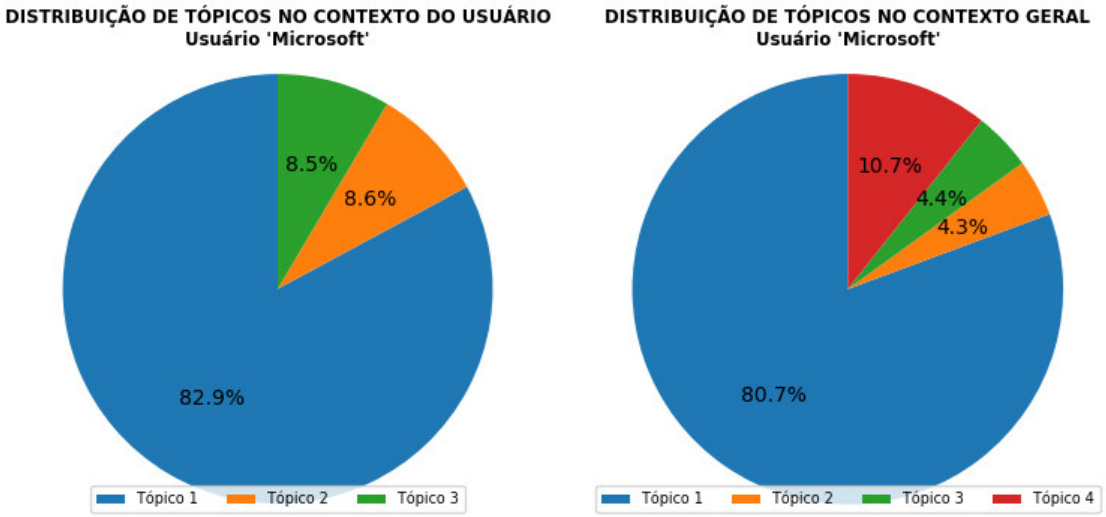
Figura 34 – Distribuição percentual da quantidade de modelos de linguagem de ordem 3 com menor e maior entropia cruzada em um projeto– usuário ‘microsoft’.



Fonte: Autor.

A distribuição dos tópicos inferidos a partir do conjunto de dados de validação para o usuário `microsoft` é observada na Figura 35. No contexto intra-usuário nota-se que grande parte do conjunto de dados de validação se enquadra nos tópicos #1 (54,8%) e #3 (33,8%). Já no contexto geral, grande parte do conjunto de validação se enquadra nos tópicos #1 (59,1%) e #4 (30,7%).

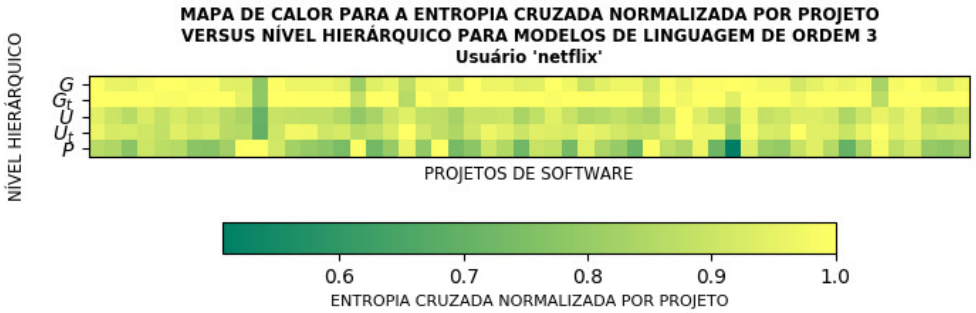
Figura 35 – Distribuição de tópicos inferidos a partir do conjunto de dados de validação para o usuário ‘microsoft’.



Fonte: Autor.

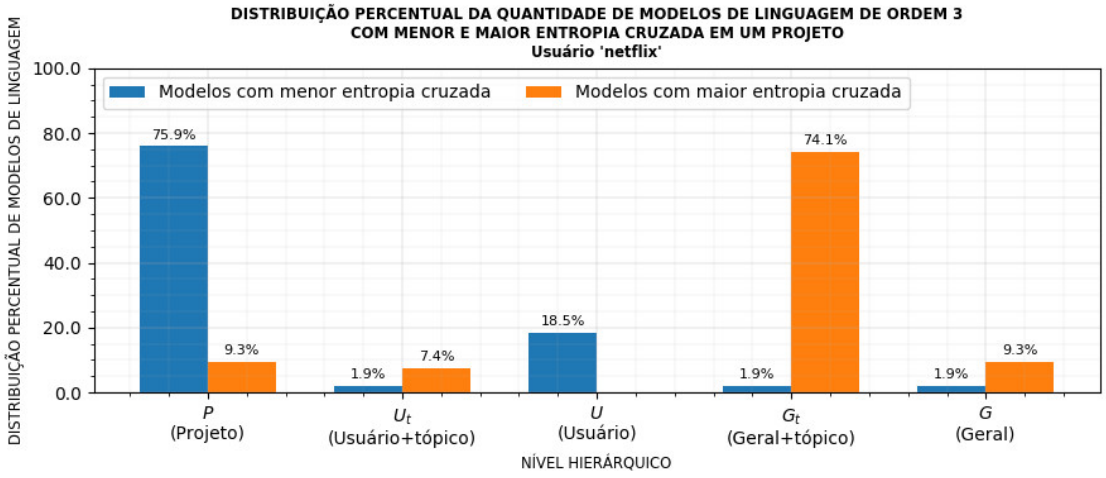
A Figura 36 representa o mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para o usuário `netflix`. Uma vez mais, nota-se que a parte inferior do gráfico concentra as menores medidas de entropia cruzada, assim como pode ser observado também na Figura 37.

Figura 36 – Mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico – usuário ‘netflix’.



Fonte: Autor.

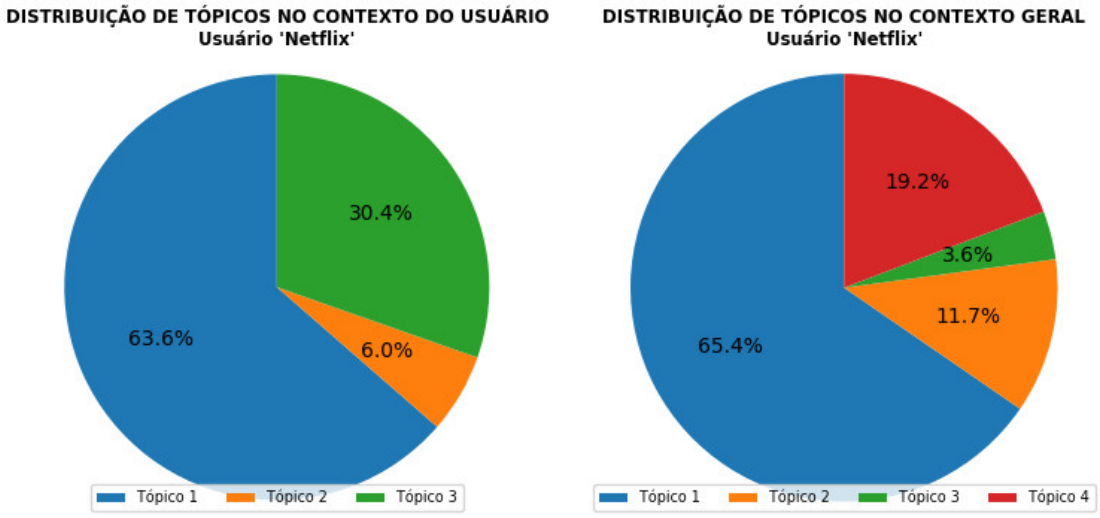
Figura 37 – Distribuição percentual da quantidade de modelos de linguagem de ordem 3 com menor e maior entropia cruzada em um projeto– usuário ‘netflix’.



Fonte: Autor.

A Figura 38 ilustra a distribuição dos tópicos inferidos a partir do conjunto de dados de validação para o usuário `netflix`. No contexto intra-usuário nota-se que grande parte do conjunto de dados de validação se enquadra nos tópicos #1 (63,6%) e #3 (30,4%). Já no contexto geral, grande parte do conjunto de validação se enquadra nos tópicos #1 (65,4%) e #4 (19,2%).

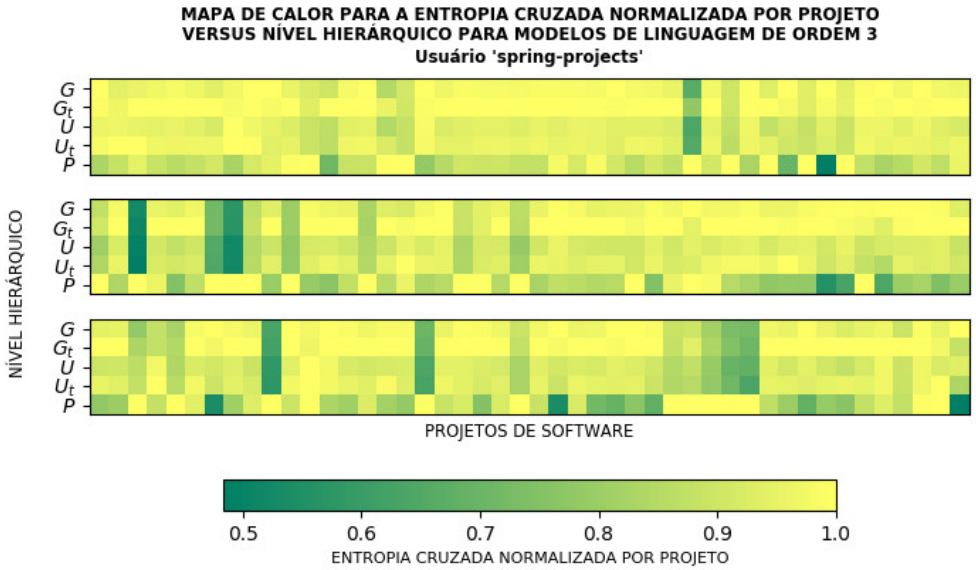
Figura 38 – Distribuição de tópicos inferidos a partir do conjunto de dados de validação para o usuário ‘netflix’.



Fonte: Autor.

A Figura 39 representa o mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para o usuário `spring-projects`. Assim como observado para o usuário `google` , nota-se que ainda que existam picos de entropia cruzada baixa nos níveis hierárquicos mais gerais, a concentração de medidas de entropia cruzada baixa permanece na parte inferior do gráfico.

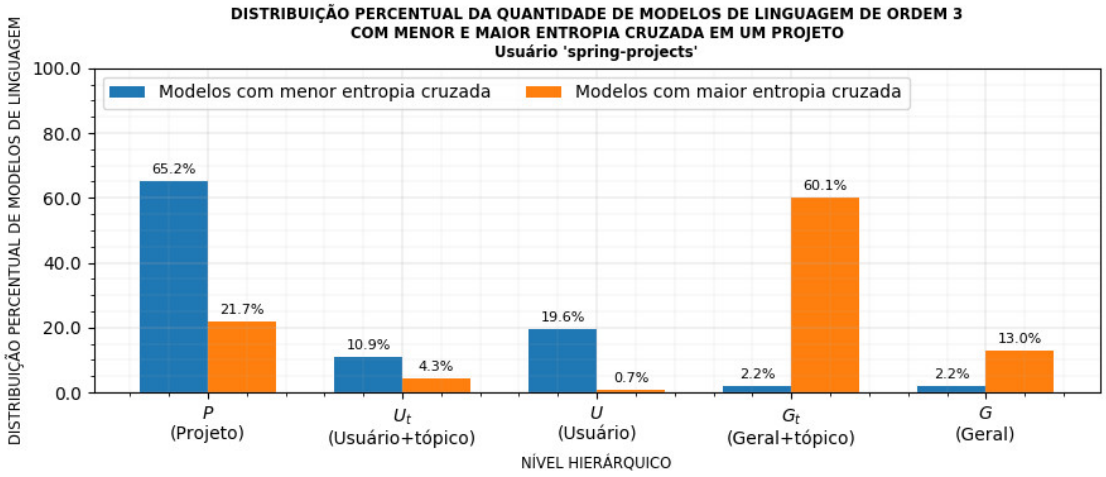
Figura 39 – Mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico – usuário ‘spring-projects’.



Fonte: Autor.

A Figura 40 evidencia as observações anteriores ao ilustrar a distribuição percentual da quantidade de modelos com menor e maior entropia cruzada para o usuário `spring-projects`. Novamente, nota-se uma distribuição relativamente mais homogênea quando comparada aos demais usuários observados. De todos os projetos avaliados, 65,2% das medidas obtidas a partir dos modelos de linguagem mais específicos apresentaram valores mais baixos de entropia cruzado. Por outro lado, modelos de linguagem baseados no projeto concentraram 21,7% das medidas mais altas de entropia cruzada.

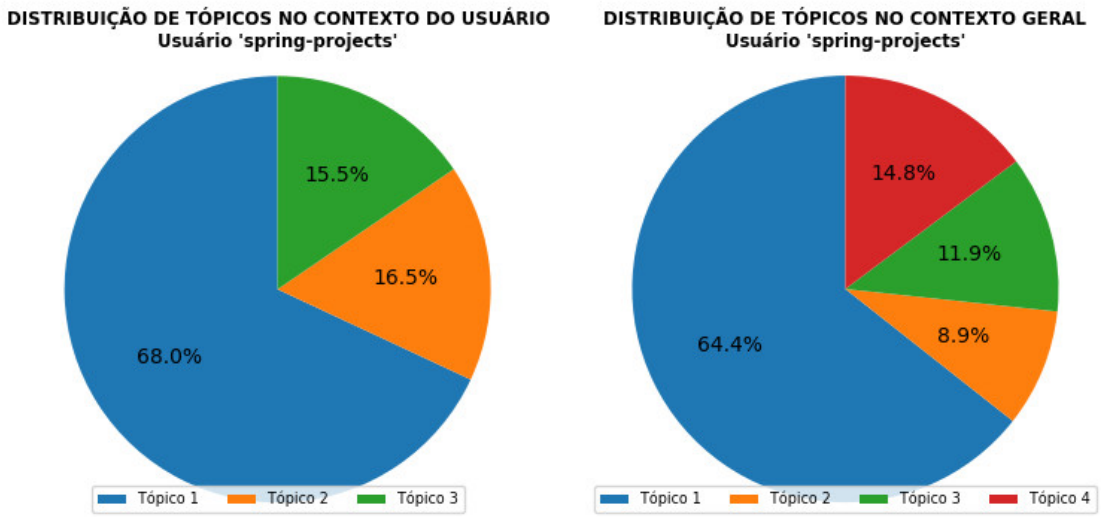
Figura 40 – Distribuição percentual da quantidade de modelos de linguagem de ordem 3 com menor e maior entropia cruzada em um projeto – usuário ‘spring-projects’.



Fonte: Autor.

Por fim, a distribuição dos tópicos inferidos a partir do conjunto de dados de validação para o usuário `spring-projects` é observada na Figura 41. No contexto intra-usuário nota-se que grande parte do conjunto de dados de validação se enquadra nos tópicos #1 (54,8%) e #3 (33,8%). No contexto geral, grande parte do conjunto de validação se enquadra nos tópicos #1 (64,4%) e #4 (14,8%).

Figura 41 – Distribuição de tópicos inferidos a partir do conjunto de dados de validação para o usuário ‘spring-projects’.



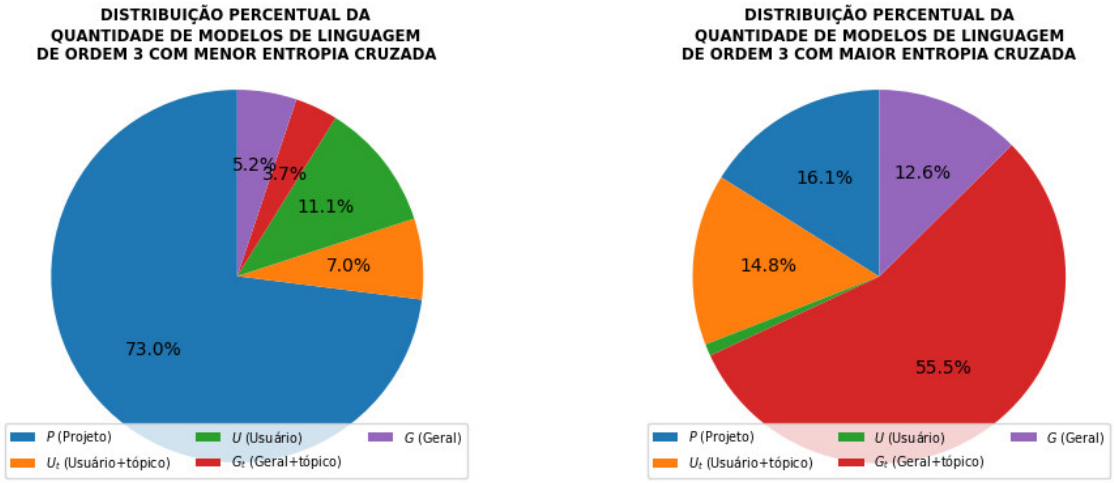
Fonte: Autor.

As avaliações realizadas ao longo dessa seção indicam que modelos de linguagem mais específicos, tais como aqueles construídos a partir de um conjunto de dados de treinamento obtidos a partir do contexto do projeto, ou ainda a partir do contexto do usuário mantenedor, apresentam medidas mais baixas de entropia cruzada que os demais. Essa é uma indicação em linha com as observações de Tu, Su e Devanbu (2014), de que as regularidades encontradas em arquivos de código-fonte têm maior relevância para a capacidade preditiva se consideradas em modelos de linguagem mais específicos. Isso pode ser observado de maneira resumida na Figura 42, que concentra todos os resultados apresentados nesta seção, sem qualquer segmentação por usuário.

No entanto, há situações – encontradas em praticamente todos os usuários estudados – em que os modelos específicos apresentaram desempenho inferior aos modelos mais abrangentes. Além disso, as avaliações observam que modelos de linguagem segmentados por objetivos apresentam resultados pouco significativos quando comparados aos demais, tanto para o caso de modelos de linguagem intra quanto inter-usuário, representados na Figura 42 por U_t e G_t, respectivamente. Modelos de linguagem inter-usuário resultam em medidas de entropia cruzada menores em apenas 3,7% das vezes, valor inferior até mesmo aos 5,2% obtidos com um modelo geral. Em complemento, modelos inter-usuário resultam em mais da metade (55,5%) das medidas de entropia cruzada mais altas. Essas constatações proporcionam reflexões, discutidas na

seção 4.6, sobre as relações entre o comportamento observado e a escolha de técnicas de modelagem de tópicos latentes para a segmentação de modelos mais abrangentes.

Figura 42 – Distribuição percentual da quantidade de modelos de linguagem de ordem 3 com menor e maior entropia cruzada em um projeto.



Fonte: Autor.

As avaliações também constatam que há uma tendência de polarização entre os modelos específicos, que geralmente proporcionam medidas de entropia cruzada mais baixas, e os modelos abrangentes, que geralmente proporcionam medidas mais altas. Isso indica que os modelos intermediários (divididos por uma distribuição de tópicos intra-usuário, por usuário e por uma distribuição de tópicos inter-usuário), embora com pouca representatividade nos valores médios apresentados Figura 42, podem ser combinados em um modelo que equilibre os diversos graus de predição específica e genérica. Assim, um estudo sobre o equilíbrio dos modelos de linguagem com o objetivo de obter medidas de entropia cruzada mais significativas é realizado na seção 4.4.2.

Adicionalmente, alguns questionamentos importantes para a validação da importância do princípio da organização hierárquica recaem sobre a escolha da ordem do modelo de linguagem e do algoritmo de suavização utilizado. Avaliar modelos de linguagem com ordem diferente de 3 e algoritmos de suavização distintos do Kneser-Ney é de grande relevância mesmo que em desacordo com as considerações estabelecidas na seção 4.3.1. Nesse sentido, as medidas de entropia cruzada normalizada por projeto com modelos de linguagem de ordem 4 e algoritmo de suavização Kneser-Ney podem ser observadas no Apêndice C. Já as medidas obtidas com o algoritmo de interpolação com pesos dependentes de contexto podem ser observadas no Apêndice D. Em ambos os casos

pode-se observar que as medidas obtidas corroboram as mesmas constatações dessa seção.

Por fim, considerando métricas extrínsecas, o Apêndice E registra um conjunto adicional de avaliações sobre o comportamento da métrica MRR nos diversos níveis hierárquicos considerados. Essas avaliações, ainda que reduzidas, são importantes para corroborar a importância dos níveis hierárquicos. Na grande maioria dos casos, modelos de linguagem mais específicos, em especial aqueles construídos a partir do projeto de software, concentram a maioria das medidas de MRR mais altas. A diferença principal em relação às medidas de entropia cruzada é que os valores mais altos estão mais bem distribuídos ao longo de todos os níveis hierárquicos considerados, em mais uma indicação de que um modelo mais apropriado deva equilibrar os diversos graus de predição específica e genérica, conforme estudo realizado na seção 4.4.2.

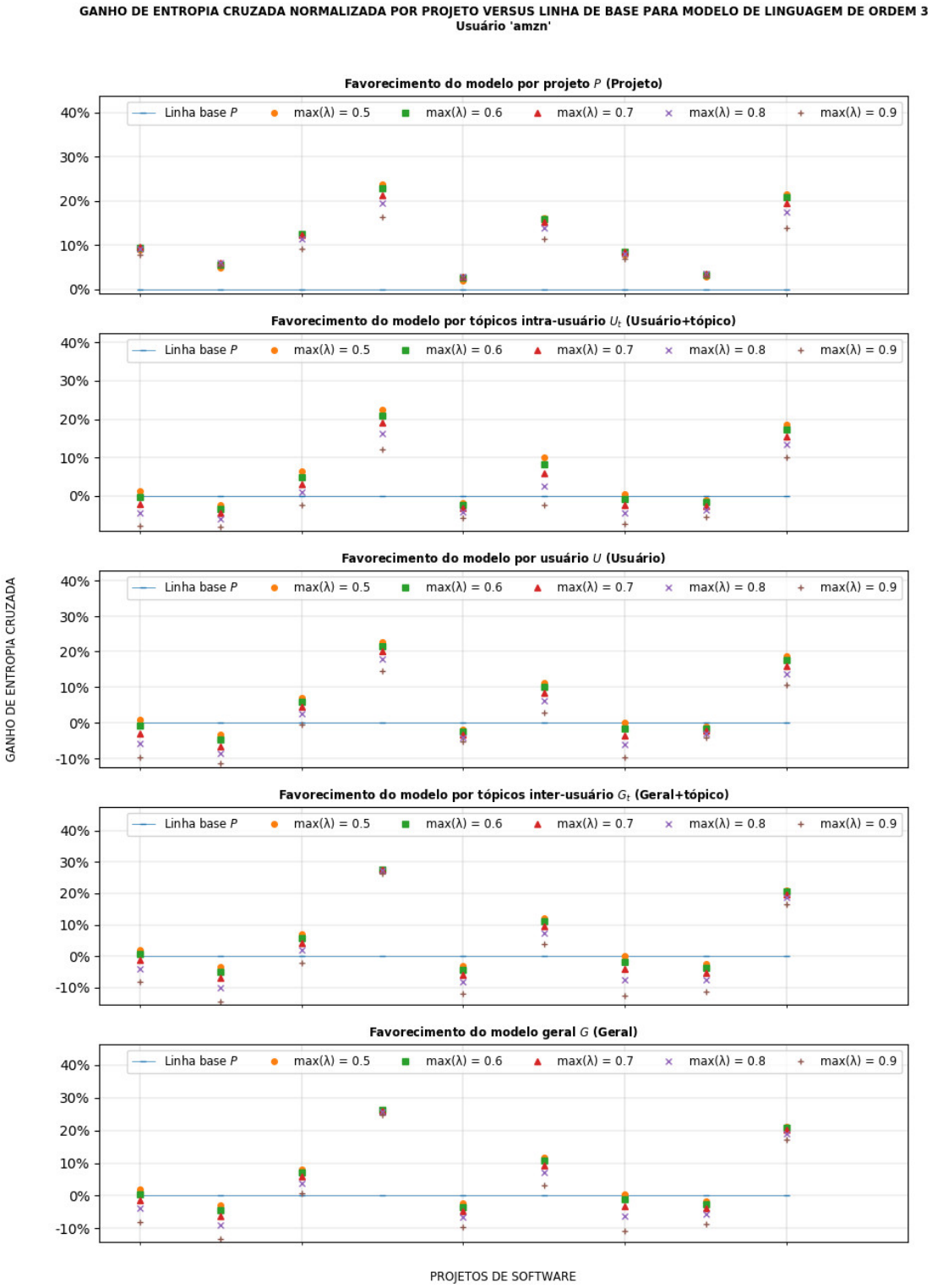
4.4.2 Combinando os modelos de linguagem

A segunda avaliação consiste em combinar os modelos de linguagem n-grama dos diferentes níveis hierárquicos e observar os efeitos dessas combinações nas medidas de entropia cruzada. E, visto que há muitas maneiras de combinar modelos de linguagem em um modelo composto, esta avaliação considera apenas o uso de técnicas de interpolação linear. Essas avaliações simulam o favorecimento de cada um dos cinco níveis que compõem a hierarquia de contexto, considerando que os pesos máximos variam no intervalo $0{,}5 \leq \lambda \leq 0{,}9$ e que os demais pesos são distribuídos igualmente entre os demais níveis hierárquicos. Assim, supondo que um nível da hierarquia referente ao usuário seja favorecido com um peso $\lambda = 0{,}6$, todos os demais níveis utilizarão pesos $\lambda = 0{,}1$. Em outras palavras, a probabilidade resultante do modelo proposto é determinada pela Equação 14.

A linha de base escolhida para a comparação dos resultados obtidos é o modelo de linguagem do nível de hierárquico de projeto, que apresentou os melhores resultados individuais na avaliação da seção 4.4.1. Os projetos de software são analisados em sua totalidade, isto é, as medidas de entropia cruzada obtidas fazem referência à todos os arquivos de código-fonte encontrados no conjunto de dados de validação do projeto sob avaliação.

A Figura 43 ilustra os ganhos de entropia cruzada com diferentes configurações de favorecimento de nível hierárquico para o usuário `amzn`.

Figura 43 – Ganho de entropia cruzada normalizada por projeto com favorecimento dos níveis hierárquicos versus linha de base – usuário ‘amzn’.



Fonte: Autor.

Nota-se na Figura 43 que o favorecimento de modelos de projeto resulta em ganhos de entropia acima da linha base em todas as variações de pesos λ consideradas. Conforme registrado na Tabela 1, o menor valor médio de ganho de entropia cruzada obtido com essa configuração de favorecimento independentemente do valor de λ é de $2{,}57\%$ e o maior valor de $20{,}72\%$, com média $10{,}41\%$, corroborando assim os resultados da seção 4.4.1. No entanto, é possível perceber que o favorecimento de modelos mais gerais proporciona ganhos progressivamente menores, eventualmente resultando em medidas de entropia cruzada até mesmo 10% inferiores à linha de base.

Tabela 1 – Ganhos de entropia cruzada normalizada por nível favorecido a partir da combinação de modelos de linguagem por interpolação linear – usuário ‘amzn’.

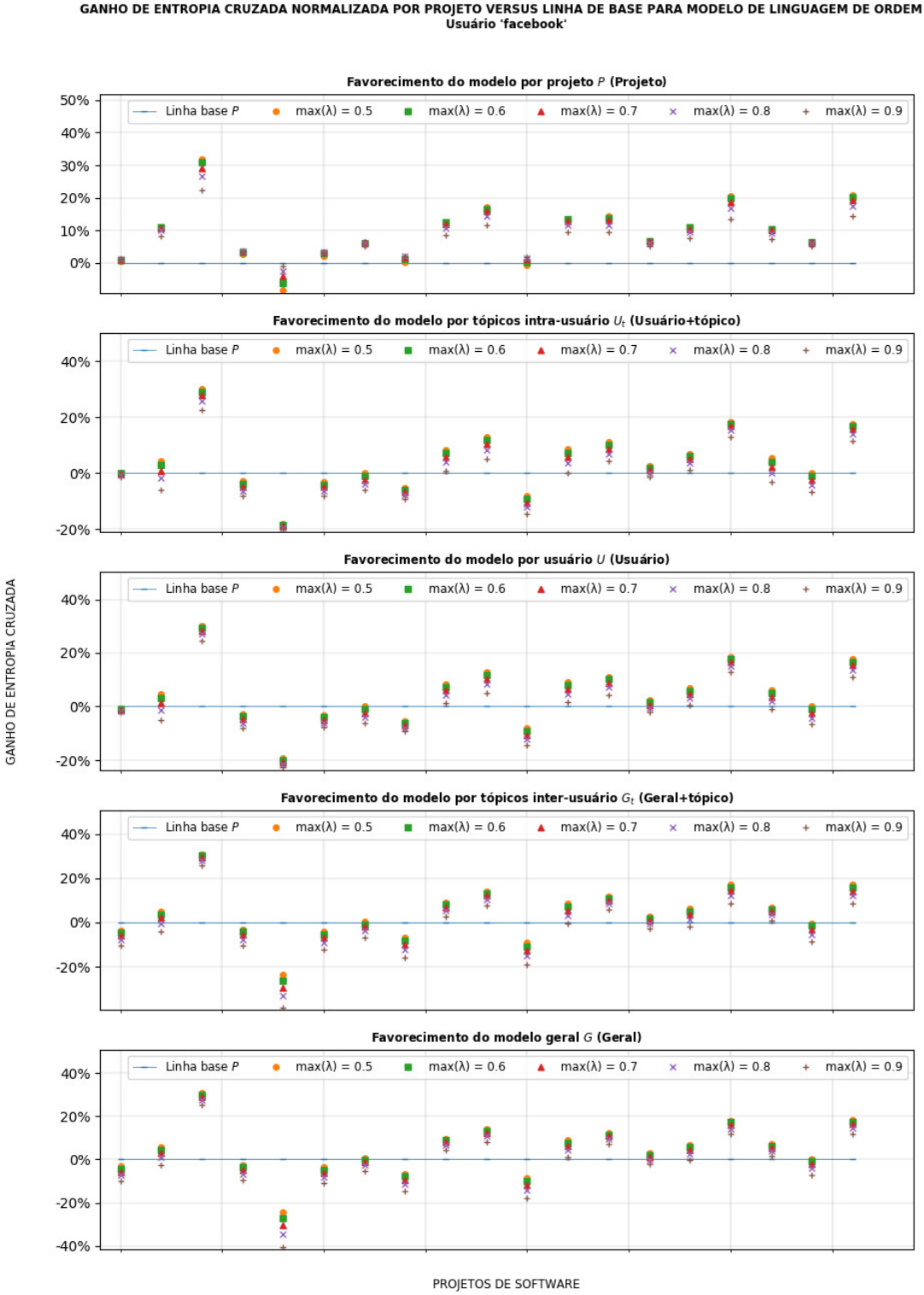
Usuário (1)	**Nível fav. (2)**	**Menor ganho % (3)**	**Maior ganho % (4)**	**Ganho % médio (5)**	**Mediana (6)**
amzn	P	2,57%	20,72%	10,41%	8,97%
	U_t	-4,93%	18,13%	2,62%	-2,70%
	U	-6,92%	19,28%	2,87%	-2,32%
	G_t	-7,95%	27,16%	3,39%	-2,21%
	G	-7,14%	25,82%	3,95%	-2,19%

Fonte: Autor.

Ademais, ainda na Figura 43, vale observar que o uso de pesos $\lambda \geq 0{,}8$, mesmo considerando o favorecimento de modelos de projeto, geralmente resulta em ganhos mais baixos se comparados aos demais.

A Figura 44 ilustra os ganhos de entropia cruzada com diferentes configurações de favorecimento de nível hierárquico para o usuário `facebook`.

Figura 44 – Ganho de entropia cruzada normalizada por projeto com favorecimento dos níveis hierárquicos versus linha de base – usuário ‘facebook’.



Fonte: Autor.

Na Figura 44 é possível verificar que o favorecimento de modelos de projeto, em todas as variações de pesos λ consideradas, quase sempre resulta em ganhos de entropia acima da linha base. Conforme registrado na Tabela 2, o menor valor médio obtido com essa configuração de favorecimento de modelos de projeto é de $-4{,}32\%$ e o maior valor de $28{,}17\%$, com média $8{,}88\%$.

Tabela 2 – Ganhos de entropia cruzada normalizada por nível favorecido a partir da combinação de modelos de linguagem por interpolação linear – usuário 'facebook'.

Usuário (1)	**Nível fav. (2)**	**Menor ganho % (3)**	**Maior ganho % (4)**	**Ganho % médio (5)**	**Mediana (6)**
facebook	P	-4,32%	28,17%	8,88%	9,43%
	U_t	-19,04%	27,14%	2,12%	0,77%
	U	-20,82%	28,07%	2,12%	0,46%
	G_t	-30,30%	28,90%	0,86%	1,27%
	G	-31,28%	28,57%	1,56%	2,36%

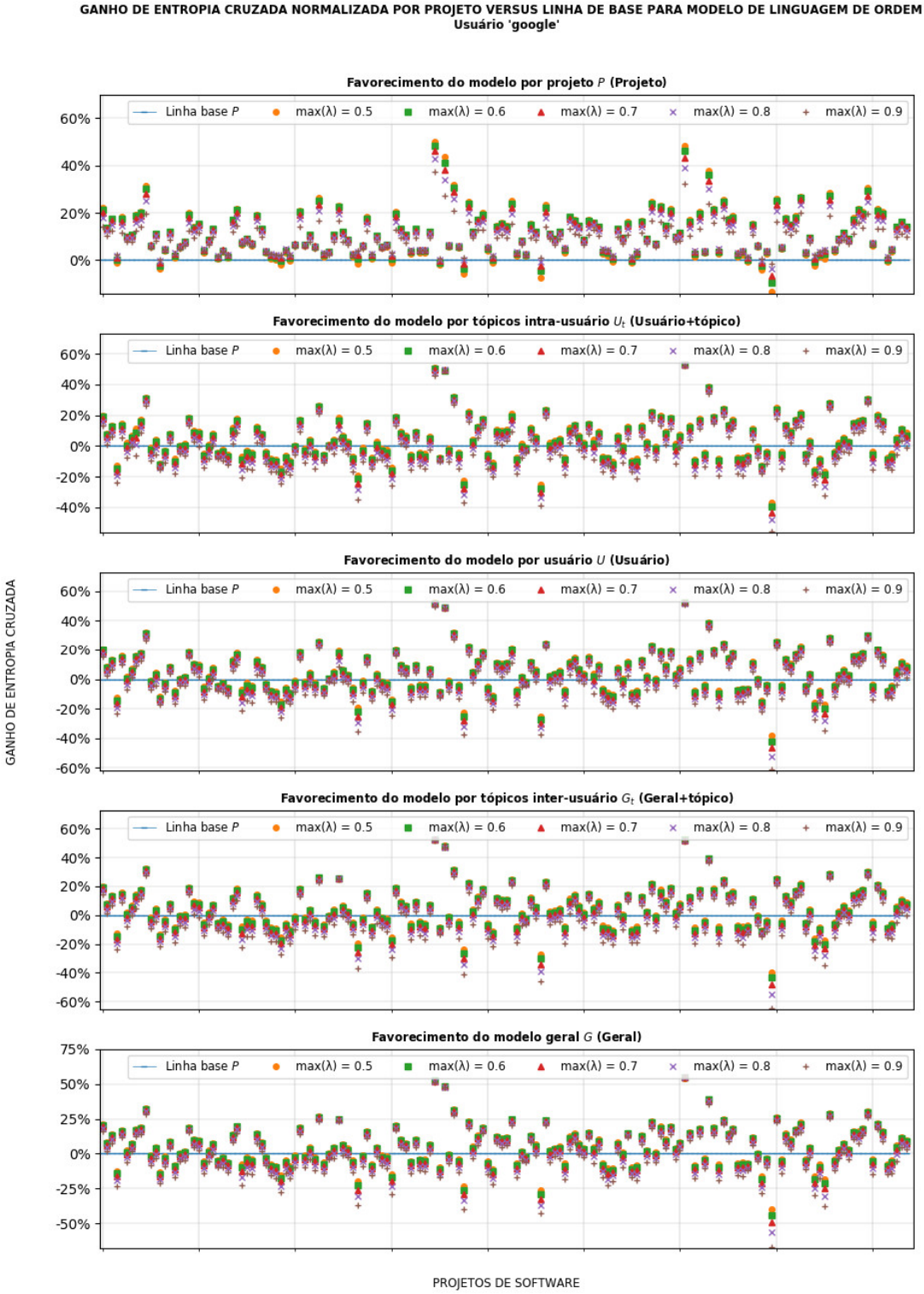
Fonte: Autor.

Ainda em relação à Figura 44, pode-se observar que há algumas poucas exceções em que o favorecimento de modelos de projeto resulta em medidas obtidas muito próximas ou ainda inferiores à linha de base, em uma indicação de que as regularidades encontradas no projeto em questão são frequentemente observadas nos modelos de linguagem dos demais níveis hierárquicos, resultando em valores de entropia cruzada próximos entre si.

Novamente, o uso de pesos $\lambda \geq 0{,}8$ geralmente resulta em ganhos mais baixos se comparados aos demais. Também é possível perceber que o favorecimento de modelos mais gerais proporciona ganhos progressivamente menores, eventualmente resultando em medidas de entropia cruzada até 20% inferiores à linha de base. Isso indica que as regularidades encontradas no projeto em questão são pouco observadas nos modelos de linguagem dos demais níveis hierárquicos e que, portanto, os valores absolutos de entropia cruzada são distantes entre si.

A Figura 45 ilustra os ganhos de entropia cruzada com diferentes configurações de favorecimento de nível hierárquico para o usuário `google`.

Figura 45 – Ganho de entropia cruzada normalizada por projeto com favorecimento dos níveis hierárquicos versus linha de base – usuário ‘google’.



Fonte: Autor.

Nota-se na Figura 45 que o favorecimento de modelos de projeto, em todas as variações de pesos λ consideradas, resulta, em grande maioria, em ganhos de entropia acima da linha base. Conforme registrado na Tabela 3, o menor valor médio obtido com essa configuração de favorecimento é de $-6{,}93\%$ e o maior valor de $44{,}82\%$, com média $10{,}54\%$.

Tabela 3 – Ganhos de entropia cruzada normalizada por nível favorecido a partir da combinação de modelos de linguagem por interpolação linear – usuário 'google'.

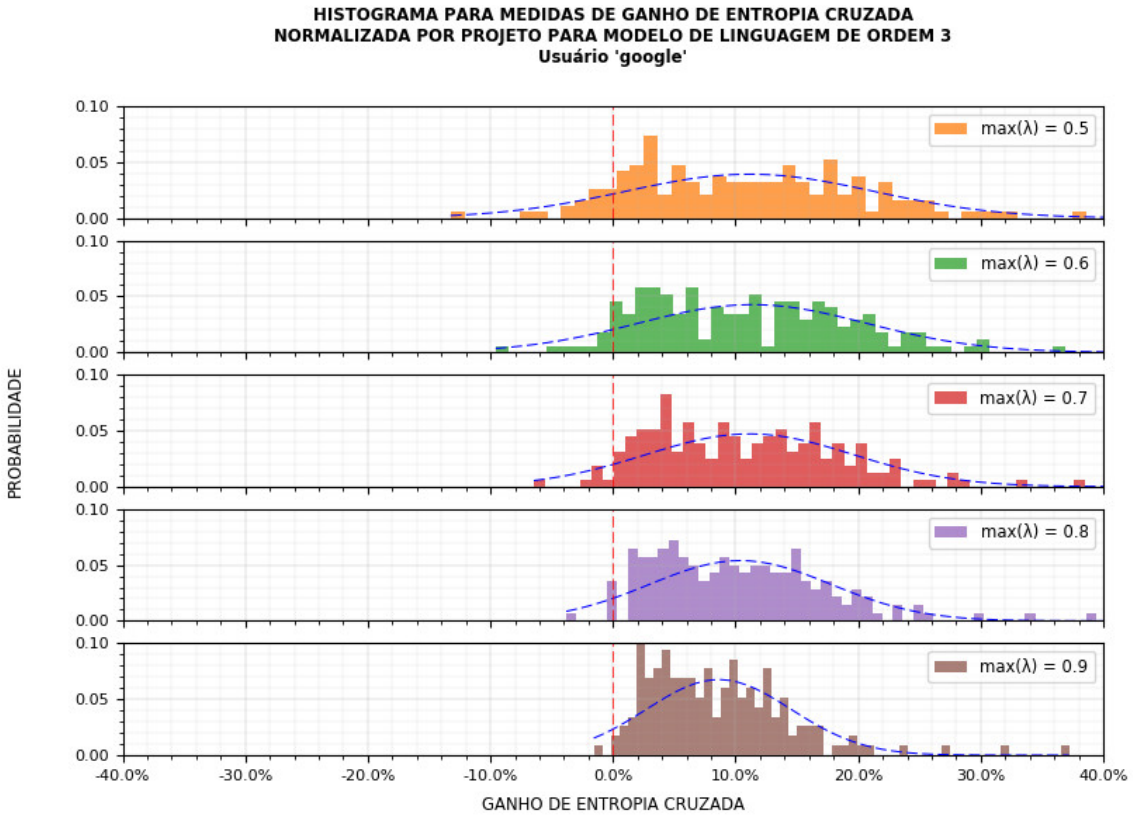
Usuário (1)	**Nível fav. (2)**	**Menor ganho % (3)**	**Maior ganho % (4)**	**Ganho % médio (5)**	**Mediana (6)**
google	P	-6,93%	44,82%	10,54%	9,74%
	U_t	-44,65%	52,85%	1,50%	-0,06%
	U	-47,88%	52,07%	2,27%	0,82%
	G_t	-50,04%	52,44%	1,66%	-0,02%
	G	-51,46%	54,74%	2,41%	1,25%

Fonte: Autor.

No entanto, ainda em relação à Figura 45, verifica-se a existência de algumas poucas exceções nas quais as medidas obtidas são muito próximas ou ainda inferiores à linha de base, em uma indicação de que as regularidades encontradas no projeto em questão são frequentemente observadas nos modelos de linguagem dos demais níveis hierárquicos. Nota-se também que os ganhos tornam-se menores à medida que modelos mais gerais são favorecidos, resultando em grandes quantidades de medidas de entropia cruzada inferiores à linha de base.

Uma vez que a quantidade de pontos exibidos na Figura 45 é significativa, opta-se por apresentar uma alternativa para a visualização através do histograma da Figura 46. A linha tracejada vermelha em $x = 0$ representa a linha de base, isto é, a medida de entropia cruzada normalizada para o nível hierárquico de projeto. A curva tracejada azul representa a função densidade de probabilidade ajustada a partir do conjunto de dados obtido.

Figura 46 – Histograma de ganhos de entropia cruzada normalizada por projeto para as diferentes configurações de favorecimento de modelos de projeto – usuário ‘google’.

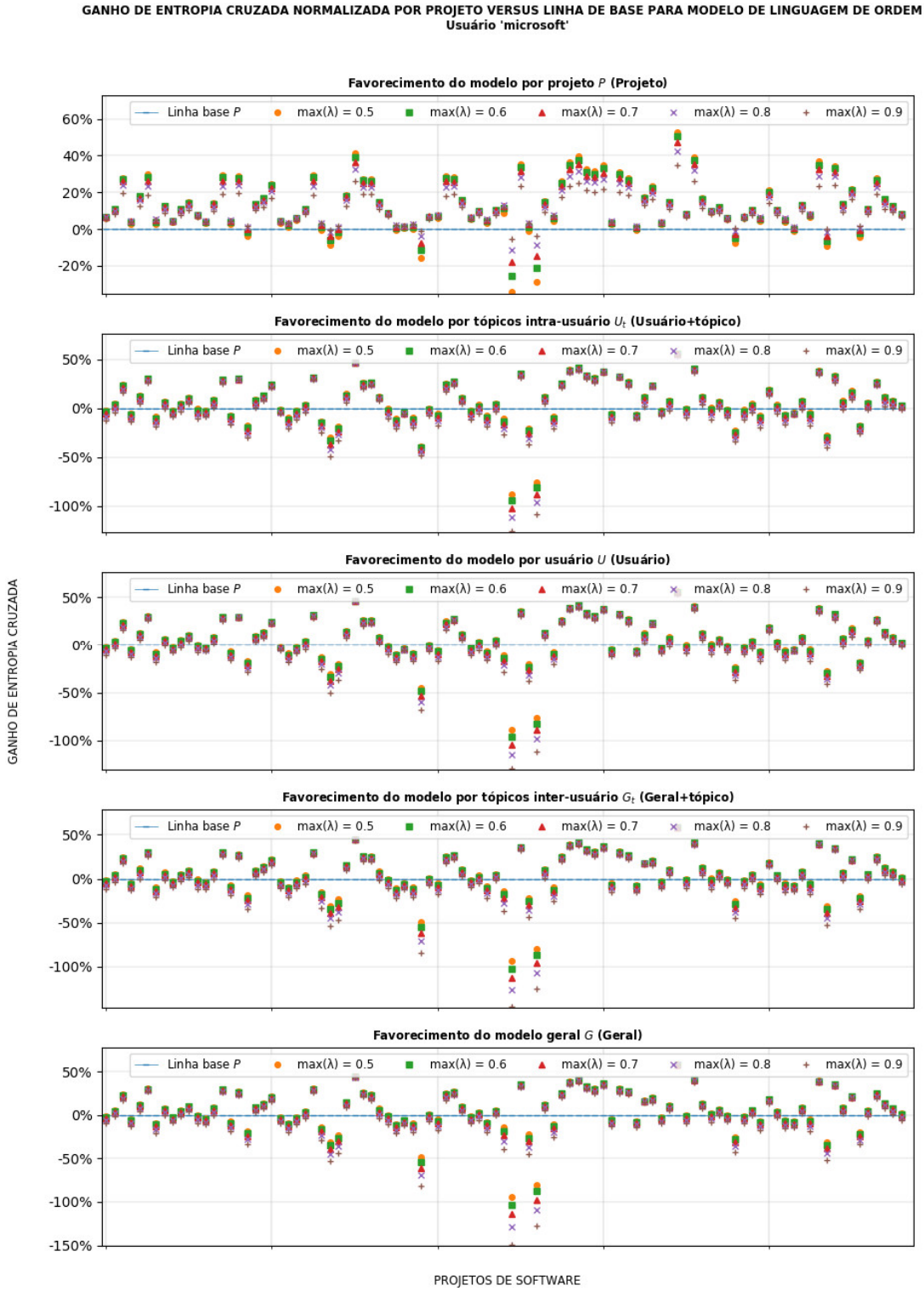


Fonte: Autor.

Verifica-se que, ainda que existam exceções, grande parte dos ganhos de entropia cruzada para as diferentes configurações de favorecimento de modelos de projeto é de valores positivos, em uma indicação de que modelos de linguagem mais específicos podem ser combinados a modelos de linguagem mais genéricos por meio de interpolação linear para obter ganhos nas medidas de entropia cruzada.

A Figura 47 ilustra os ganhos de entropia cruzada com diferentes configurações de favorecimento de nível hierárquico para o usuário `microsoft`.

Figura 47 – Ganho de entropia cruzada normalizada por projeto com favorecimento dos níveis hierárquicos versus linha de base – usuário ‘microsoft’.



Fonte: Autor.

Nota-se na Figura 47 que o favorecimento de modelos de projeto, em todas as variações de pesos λ consideradas, resulta, na grande maioria das vezes, em ganhos de entropia acima da linha base. Conforme registrado na Tabela 4, o menor valor médio obtido com essa configuração de favorecimento é de $-18{,}93\%$ e o maior valor de $45{,}52\%$, com média $12{,}28\%$.

Tabela 4 – Ganhos de entropia cruzada normalizada por nível favorecido a partir da combinação de modelos de linguagem por interpolação linear – usuário 'microsoft'.

Usuário (1)	**Nível fav. (2)**	**Menor ganho % (3)**	**Maior ganho % (4)**	**Ganho % médio (5)**	**Mediana (6)**
microsoft	P	-18,93%	45,52%	12,28%	9,99%
	U_t	-104,37%	55,06%	2,65%	1,22%
	U	-106,50%	54,75%	2,65%	2,07%
	G_t	-115,76%	58,60%	1,53%	2,14%
	G	-118,03%	58,02%	2,01%	2,99%

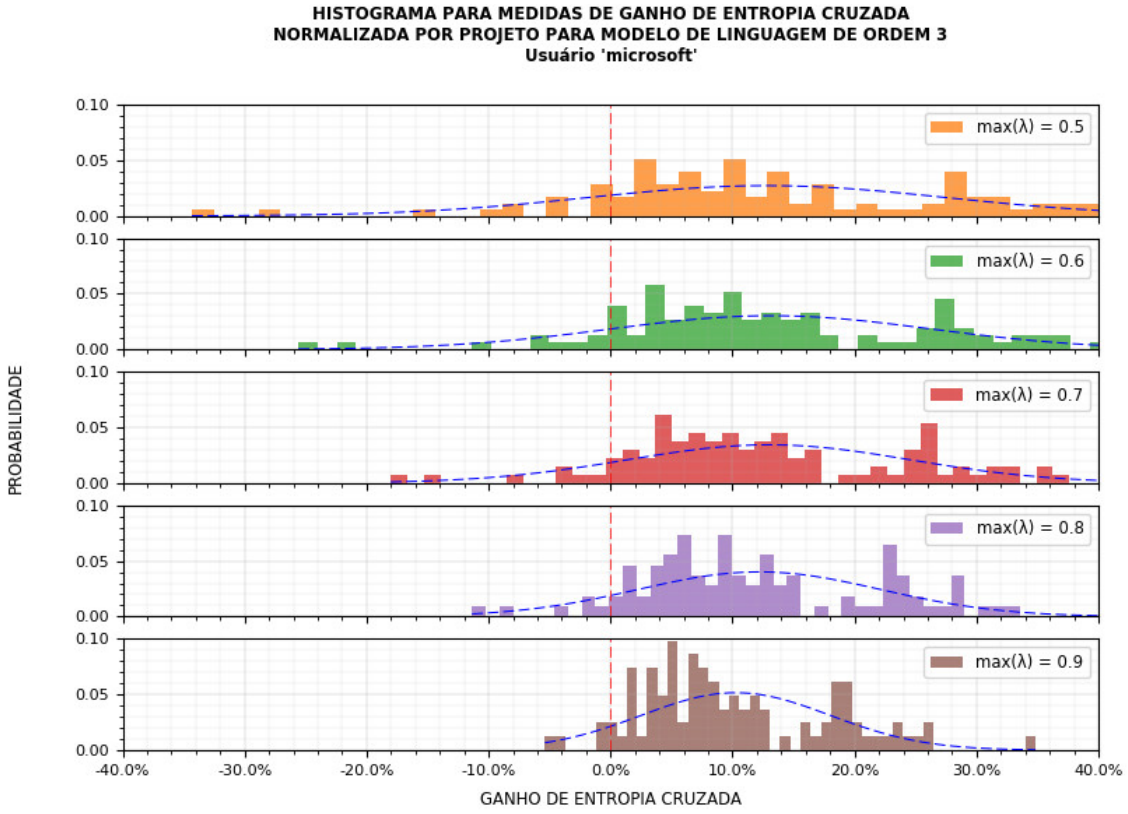
Fonte: Autor.

Ainda em relação à Figura 47, há algumas exceções nas quais as medidas obtidas são muito próximas ou ainda inferiores à linha de base, novamente em uma indicação de que as regularidades encontradas no projeto em questão são frequentemente observadas nos modelos de linguagem de dos demais níveis hierárquicos.

Ademais, os ganhos tornam-se menores à medida que modelos mais gerais são favorecidos, resultando em grandes quantidades de medidas de entropia cruzada inferiores à linha de base. Embora o comportamento geral das medidas de ganho obtidas com níveis mais altos seja similar, os valores absolutos sofrem degradação progressiva, e em alguns casos é possível observar valores 50% inferiores à linha base.

Dado que a quantidade de pontos exibidos na Figura 47 é significativa, opta-se por apresentar uma alternativa para a visualização através do histograma da Figura 48. A linha tracejada vermelha em $x = 0$ representa a linha de base, isto é, a medida de entropia cruzada normalizada para o nível hierárquico de projeto. A curva tracejada azul representa a função densidade de probabilidade ajustada a partir do conjunto de dados obtido.

Figura 48 – Histograma de ganhos de entropia cruzada normalizada por projeto para as diferentes configurações de favorecimento de modelos de projeto – usuário ‘microsoft’.

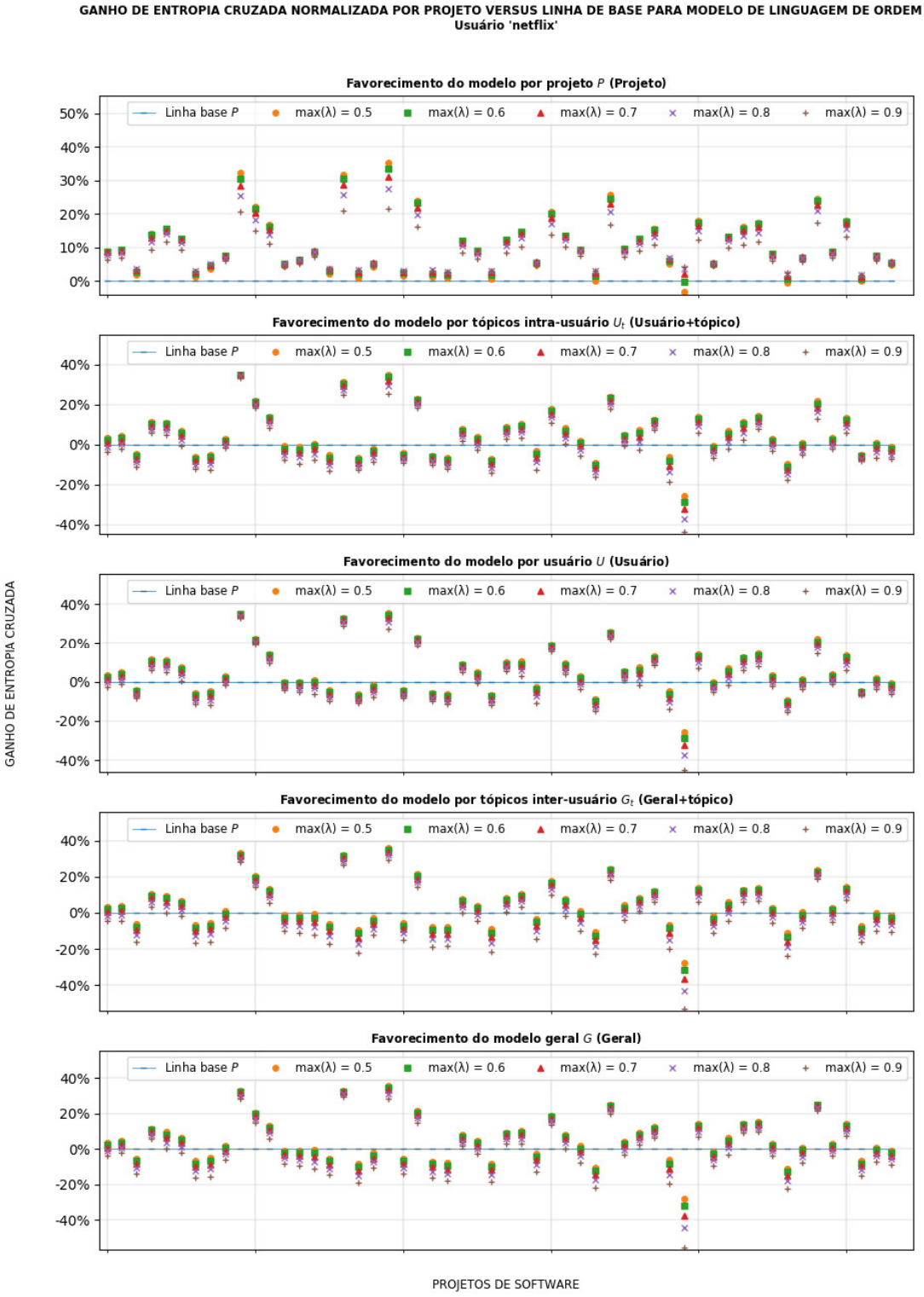


Fonte: Autor.

Verifica-se que, ainda que existam exceções (em quantidade ligeiramente maior se comparada ao usuário `google`, conforme Figura 46), grande parte dos ganhos de entropia cruzada para as diferentes configurações de favorecimento de modelos de projeto é de valores positivos, em uma indicação de que modelos de linguagem mais específicos podem ser combinados a modelos de linguagem mais genéricos por meio de interpolação linear para obter ganhos nas medidas de entropia cruzada.

A Figura 49 ilustra os ganhos de entropia cruzada com diferentes configurações de favorecimento de nível hierárquico para o usuário `netflix`.

Figura 49 – Ganho de entropia cruzada normalizada por projeto com favorecimento dos níveis hierárquicos versus linha de base – usuário ‘netflix’.



Fonte: Autor.

Na Figura 49 pode-se notar que o favorecimento de modelos de projeto, em todas as variações de pesos λ consideradas, quase sempre resulta em ganhos de entropia acima da linha base. Conforme registrado na Tabela 5, o menor valor médio obtido com essa configuração de favorecimento de modelos de projeto é de 1,09% e o maior valor de 29,87%, com média 10,01%.

Tabela 5 – Ganhos de entropia cruzada normalizada por nível favorecido a partir da combinação de modelos de linguagem por interpolação linear – usuário 'netflix'.

Usuário (1)	**Nível fav. (2)**	**Menor ganho % (3)**	**Maior ganho % (4)**	**Ganho % médio (5)**	**Mediana (6)**
netflix	P	1,09%	29,87%	10,01%	8,30%
	U_t	-33,40%	34,44%	2,31%	0,99%
	U	-33,78%	34,19%	3,23%	1,25%
	G_t	-38,43%	33,12%	0,71%	-0,33%
	G	-39,64%	32,62%	1,57%	0,26%

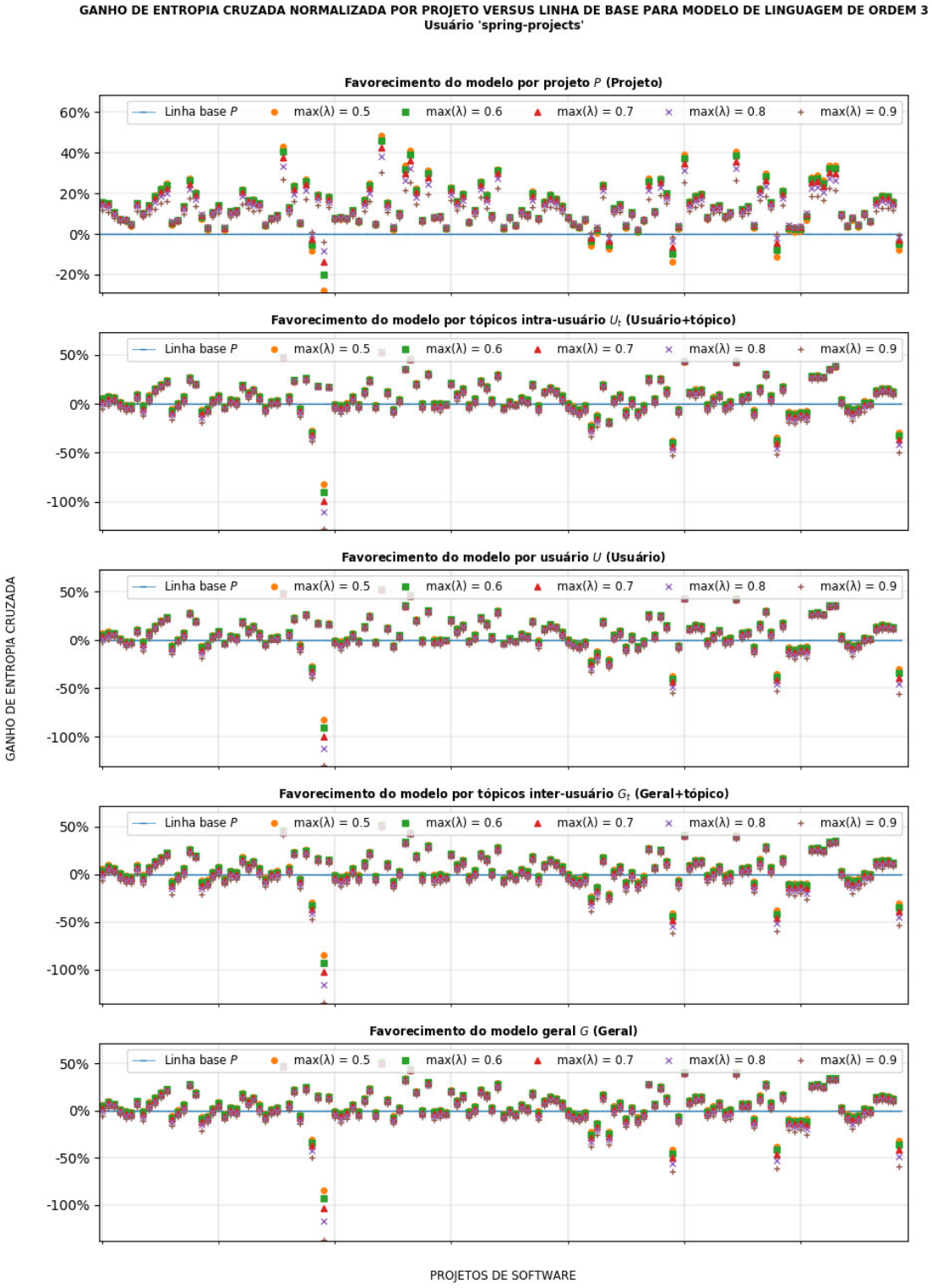
Fonte: Autor.

Há algumas exceções nas quais as medidas obtidas são muito próximas ou ainda inferiores à linha de base, uma vez mais indicando que as regularidades encontradas no projeto em questão são frequentemente observadas nos modelos de linguagem de dos demais níveis hierárquicos.

Entretanto, à medida que modelos mais gerais são favorecidos, os ganhos tornam-se menores, eventualmente resultando em medidas de entropia cruzada inferiores à linha de base. Novamente, embora o comportamento geral das medidas de ganho obtidas com níveis mais altos seja similar, os valores absolutos sofrem de uma degradação progressiva, e em alguns casos é possível observar valores 20% inferiores à linha base.

A Figura 50 ilustra os ganhos de entropia cruzada com diferentes configurações de favorecimento de nível hierárquico para o usuário `spring-projects`.

Figura 50 – Ganho de entropia cruzada normalizada por projeto com favorecimento dos níveis hierárquicos versus linha de base– usuário ‘spring-projects’.



Fonte: Autor.

Pode-se notar na Figura 50 que o favorecimento de modelos de projeto, em todas as variações de pesos λ consideradas, quase sempre resulta em ganhos de entropia acima da linha base. Conforme registrado na Tabela 6, o menor valor médio obtido com essa configuração de favorecimento de modelos de projeto é de $-14{,}69\%$ e o maior valor de $41{,}14\%$, com média $12{,}38\%$.

Tabela 6 – Ganhos de entropia cruzada normalizada por nível favorecido a partir da combinação de modelos de linguagem por interpolação linear – usuário 'spring-projects'.

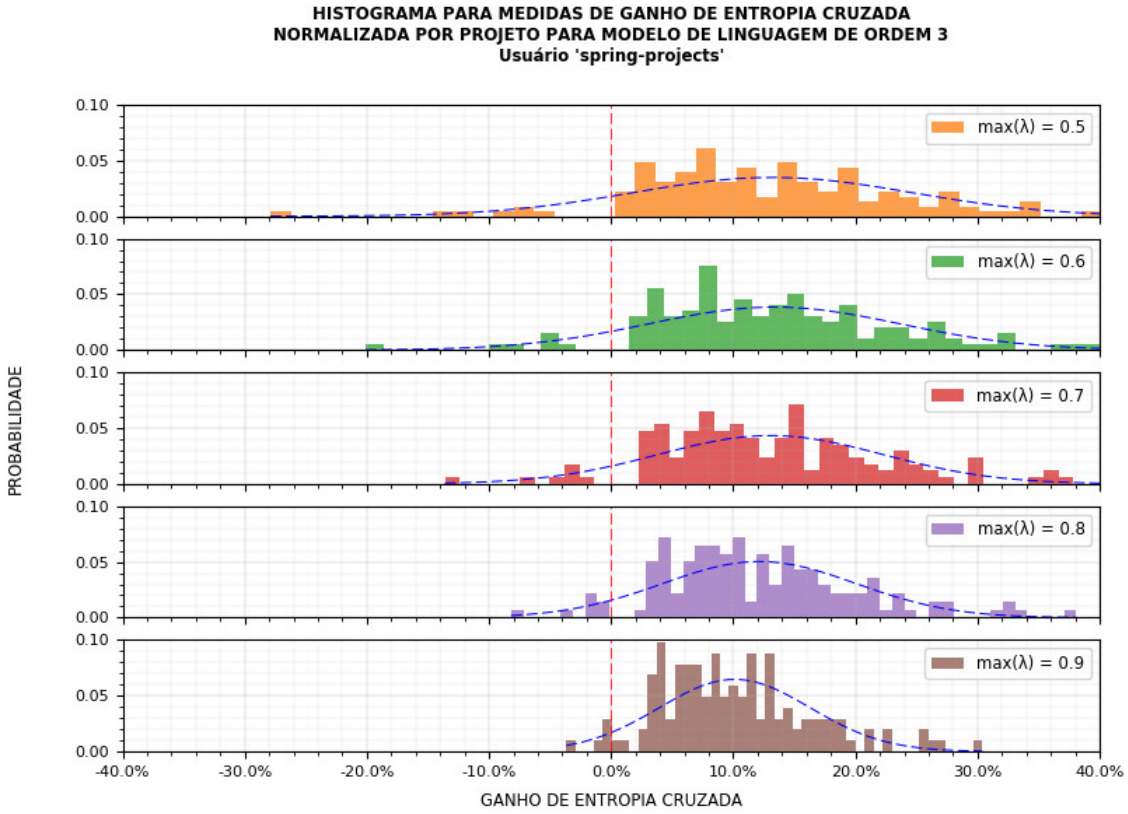
Usuário (1)	**Nível fav. (2)**	**Menor ganho % (3)**	**Maior ganho % (4)**	**Ganho % médio (5)**	**Mediana (6)**
microsoft	P	-14,69%	41,14%	12,38%	10,96%
	U_t	-101,66%	52,37%	4,95%	3,25%
	U	-102,59%	52,00%	5,21%	3,67%
	G_t	-105,82%	50,88%	3,42%	2,38%
	G	-107,19%	50,39%	3,75%	2,23%

Fonte: Autor.

Entretanto, há algumas exceções nas quais as medidas obtidas são muito próximas ou ainda inferiores à linha de base. Além disso, os ganhos tornam-se menores à medida que modelos mais gerais são favorecidos, o que eventualmente resulta em medidas de entropia cruzada inferiores à linha de base. Uma vez mais, o comportamento geral das medidas de ganho obtidas com níveis mais altos é similar, com a mesma degradação progressiva dos valores absolutos, em alguns casos 50% abaixo da linha base.

Devido à quantidade de pontos exibidos na Figura 50, opta-se, novamente, por apresentar uma alternativa para a visualização através do histograma da Figura 51. A linha tracejada vermelha em $x = 0$ representa a linha de base, isto é, a medida de entropia cruzada normalizada para o nível hierárquico de projeto. A curva tracejada azul representa a função densidade de probabilidade ajustada a partir do conjunto de dados obtido.

Figura 51 – Histograma de ganhos de entropia cruzada normalizada por projeto para as diferentes configurações de favorecimento de modelos de projeto – usuário 'spring-projects'.



Fonte: Autor.

Nota-se comportamento similar àquele observado para os usuários `google` e `microsoft`, com grande parte dos ganhos de entropia cruzada para as diferentes configurações de favorecimento de modelos de projeto distribuída ao longo de valores positivos.

As avaliações realizadas ao longo dessa seção fortalecem as observações da seção 4.4.1 sobre a importância de modelos mais específicos e também indicam que modelos desse tipo podem ser combinados a modelos de linguagem mais genéricos por meio de interpolação linear para que se obtenham ganhos nas medidas de entropia cruzada de um projeto de software. Esses ganhos, sumarizados em valores percentuais na Tabela 7, possuem um valor médio próximo a 10%, e podem atingir valores tão altos quanto 45%.

Tabela 7 – Ganhos de entropia cruzada normalizada por usuário considerando o favorecimento do nível de projeto.

Usuário (1)	**Menor ganho %** (2)	**Maior ganho %** (3)	**Ganho % médio** (4)	**Mediana** (5)
amzn	2,57%	20,72%	10,41%	8,97%
facebook	-4,32%	28,17%	8,88%	9,43%
google	-6,93%	44,82%	10,54%	9,74%
microsoft	-18,93%	45,52%	12,28%	9,99%
netflix	1,09%	29,87%	10,01%	8,30%
spring-projects	-14,69%	41,14%	12,38%	10,96%

Fonte: Autor.

Entretanto, há cenários de exceção – encontrados em praticamente todos os usuários estudados – em que, mesmo favorecendo os modelos mais específicos, a combinação de modelos proporciona resultados muito próximos ou inferiores à linha de base. Resultados próximos à linha de base indicam que as regularidades encontradas nos arquivos de código-fonte de um projeto são frequentemente observadas nos modelos de linguagem dos demais níveis hierárquicos, o que produz valores de entropia cruzada próximos entre si. Já resultados inferiores à linha base são motivados por regularidades que, embora frequentes em um projeto, são pouco observadas nos modelos dos demais níveis hierárquicos, o que produz valores de entropia cruzada distantes entre si. Situações como essas sugerem que o uso de interpolação adaptativa para combinar os modelos de linguagem n-grama dos diferentes níveis hierárquicos talvez seja mais adequado.

4.4.3 Comparação com modelos de linguagem com suporte a cache

A terceira e última avaliação consiste em comparar as medidas de entropia cruzada obtidas a partir do modelo proposto, em especial as medidas encontradas na seção 4.4.2, com uma versão adaptada do conceito de *cache* sugerido por Tu, Su e Devanbu (2014), que considera que a probabilidade resultante de observação de um termo t_i seja determinada pela Equação 16, sendo que γ representa um parâmetro de concentração e H representa o número de vezes que o prefixo h foi observado no *cache*.

$$P(t_i|h, cache) = \frac{\gamma}{\gamma + H} P_{n-gram}(t_i|h) + \frac{H}{\gamma + H} P_{cache}(t_i|h) \qquad (16)$$

Trata-se de uma versão adaptada, pois no trabalho original o *cache* é construído a partir do conteúdo de um único arquivo de código-fonte e, eventualmente, dos arquivos encontrados no mesmo diretório e subdiretórios próximos ao arquivo original. Nesta análise, o *cache* é substituído pelo modelo de linguagem n-grama construído a partir do projeto de software. Em complemento, e assim como o trabalho original, o parâmetro de concentração é configurado como $\gamma = 1{,}0$.

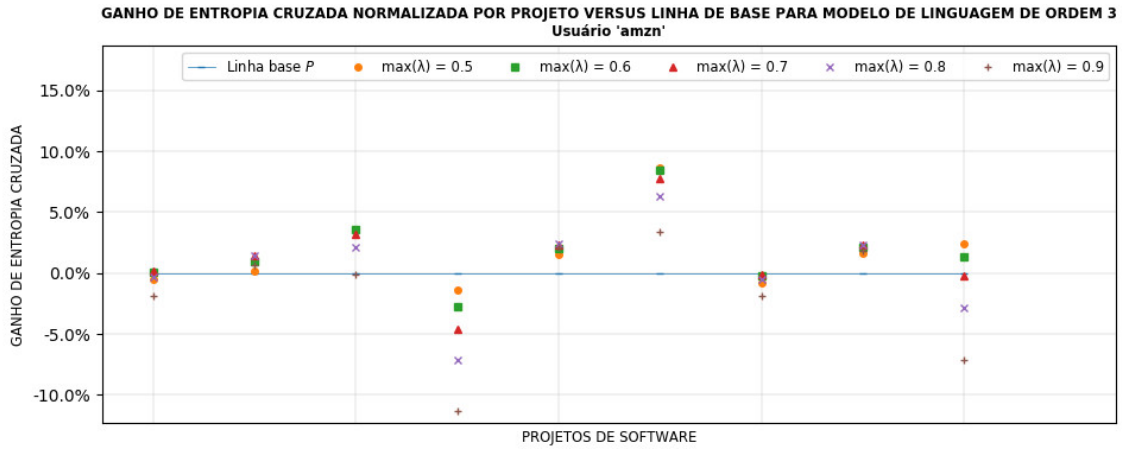
O trabalho de Hellendoorn e Devanbu (2017) também seria passível de comparação por utilizar o mesmo conceito de *cache* definido por Tu, Su e Devanbu (2014). No entanto, visto que Hellendoorn e Devanbu (2017) definem três níveis para o *cache* – um nível de agrupamento de arquivos em um mesmo diretório, um nível de projeto e um nível geral – e visto que não há nível hierárquico equivalente no modelo proposto que permita aproximação, a comparação torna-se inviável.

A linha de base escolhida para a comparação dos resultados obtidos é o próprio modelo de linguagem com *cache*, e as medidas do modelo proposto consideram apenas o nível de hierárquico de projeto, já que este apresentou melhor resultado mediante favorecimento via atribuição de pesos conforme seção 4.4.2.

Novamente, os projetos de software são analisados em sua totalidade, isto é, as medidas de entropia cruzada obtidas fazem referência a todos os arquivos de código-fonte encontrados no conjunto de dados de validação do projeto sob avaliação.

A Figura 52 ilustra o ganho na medida de entropia cruzada normalizada por projeto para o usuário `amzn`. Embora algumas das medidas obtidas com o modelo proposto sejam iguais ou superiores à linha de base, é possível notar uma redução nos ganhos obtidos em comparação com as análises da seção 4.4.2.

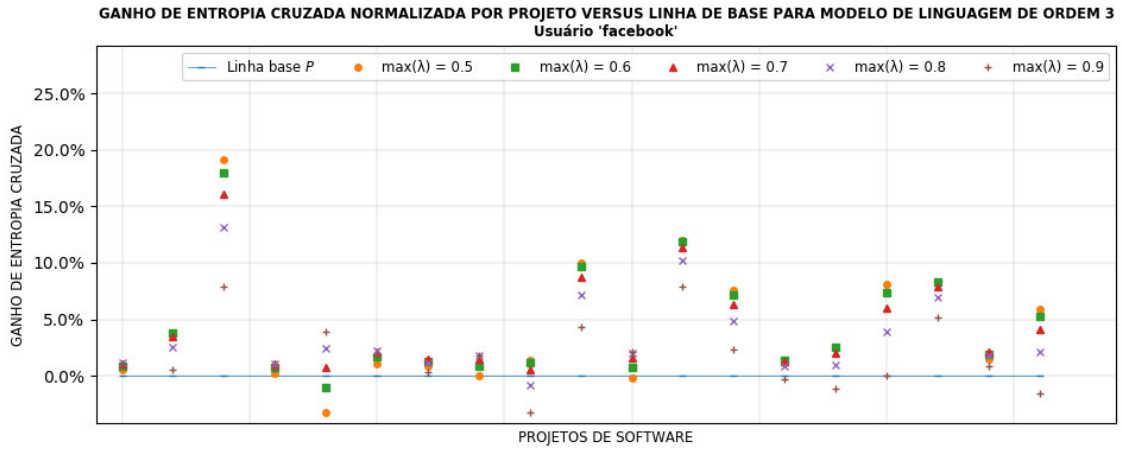
Figura 52 – Ganho de entropia cruzada normalizada versus linha de base definida a partir de um modelo com cache – usuário ‘amzn’.



Fonte: Autor.

A Figura 53 ilustra as diferenças na medida de entropia cruzada normalizada por projeto para o usuário `facebook`. Novamente, ainda que algumas das medidas obtidas com o modelo proposto sejam iguais ou superiores à linha de base, é possível notar uma redução nos ganhos obtidos em comparação com as análises da seção 4.4.2.

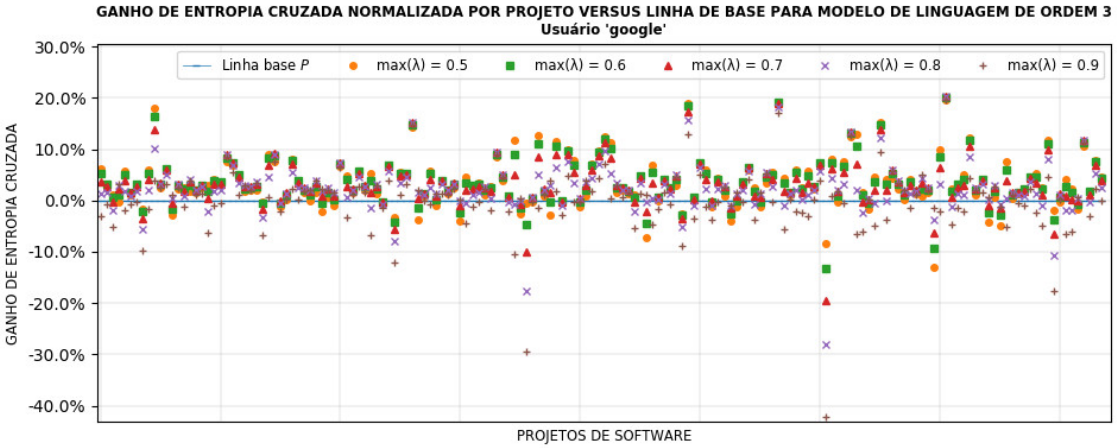
Figura 53 – Ganho de entropia cruzada normalizada versus linha de base definida a partir de um modelo com cache – usuário ‘facebook’.



Fonte: Autor.

A Figura 54 ilustra as diferenças na medida de entropia cruzada normalizada por projeto para o usuário `google`.

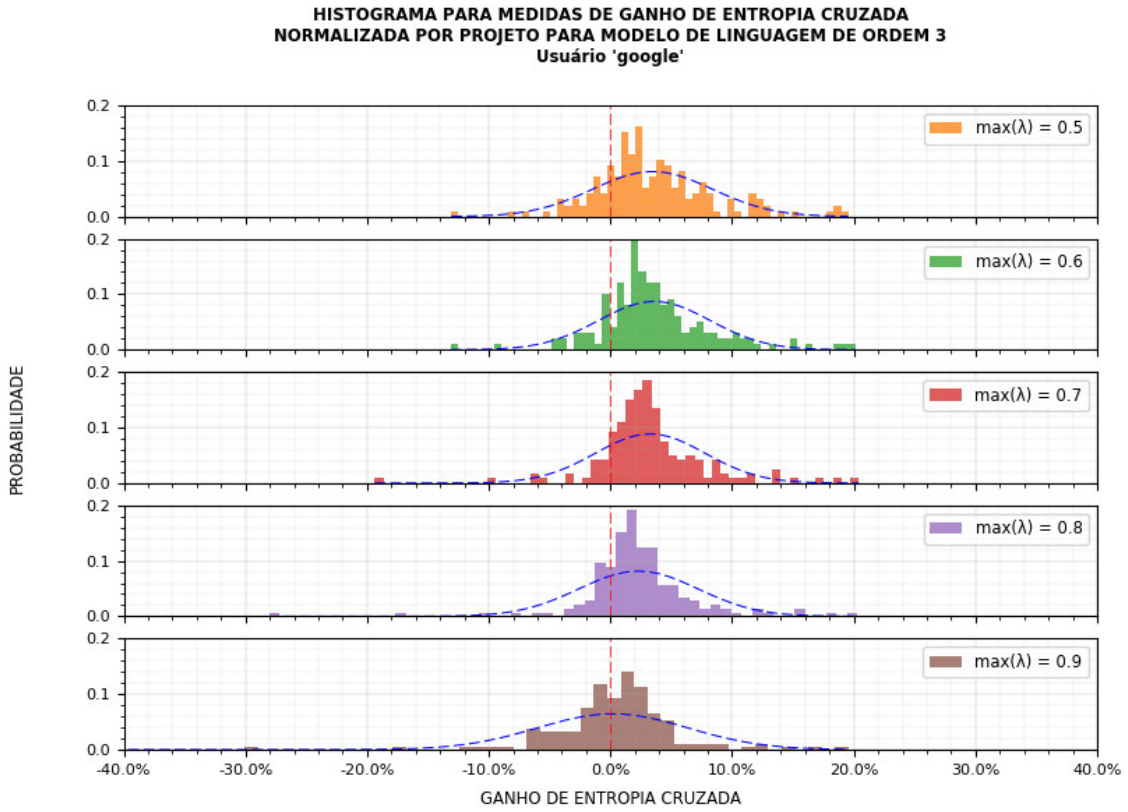
Figura 54 – Ganho de entropia cruzada normalizada versus linha de base definida a partir de um modelo com cache – usuário ‘google’.



Fonte: Autor.

Uma vez que a quantidade de pontos exibidos na Figura 54 é significativa, opta-se por apresentar uma alternativa para a visualização através do histograma da Figura 55. A linha tracejada vermelha em $x = 0$ representa a linha de base, isto é, a medida de entropia cruzada normalizada para o modelo com suporte a *cache* simulado. A curva tracejada azul representa a função densidade de probabilidade ajustada a partir do conjunto de dados obtido.

Figura 55 – Histograma de ganhos de entropia cruzada normalizada por projeto para as diferentes configurações de favorecimento de modelos de projeto – usuário ‘google’.

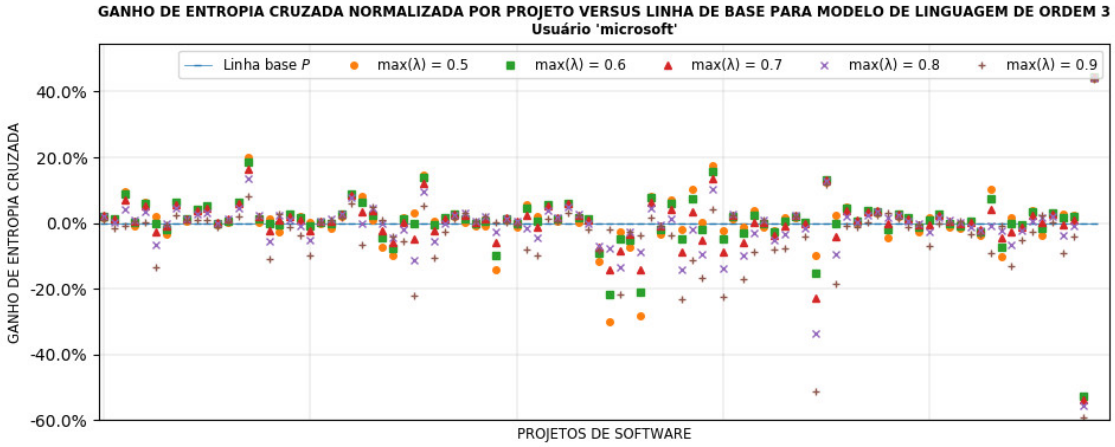


Fonte: Autor.

Nota-se que grande parte dos ganhos de entropia cruzada para as diferentes configurações de favorecimento de modelos de projeto ainda é de valores positivos. No entanto, além de uma redução nos ganhos obtidos em comparação com as análises da seção 4.4.2 (conforme Figura 46), observa-se que a distribuição de ganhos se torna cada vez menor à medida que o maior valor para o peso λ progride. Para o caso $\lambda = 0{,}9$, por exemplo, é possível observar que os ganhos estão relativamente bem distribuídos nas proximidades de $x = 0$, indicando que nessas configurações a adoção do modelo proposto proporcionaria pouco ou nenhum ganho em comparação ao modelo com suporte a *cache* simulado.

A Figura 56 ilustra as diferenças na medida de entropia cruzada normalizada por projeto para o usuário `microsoft`.

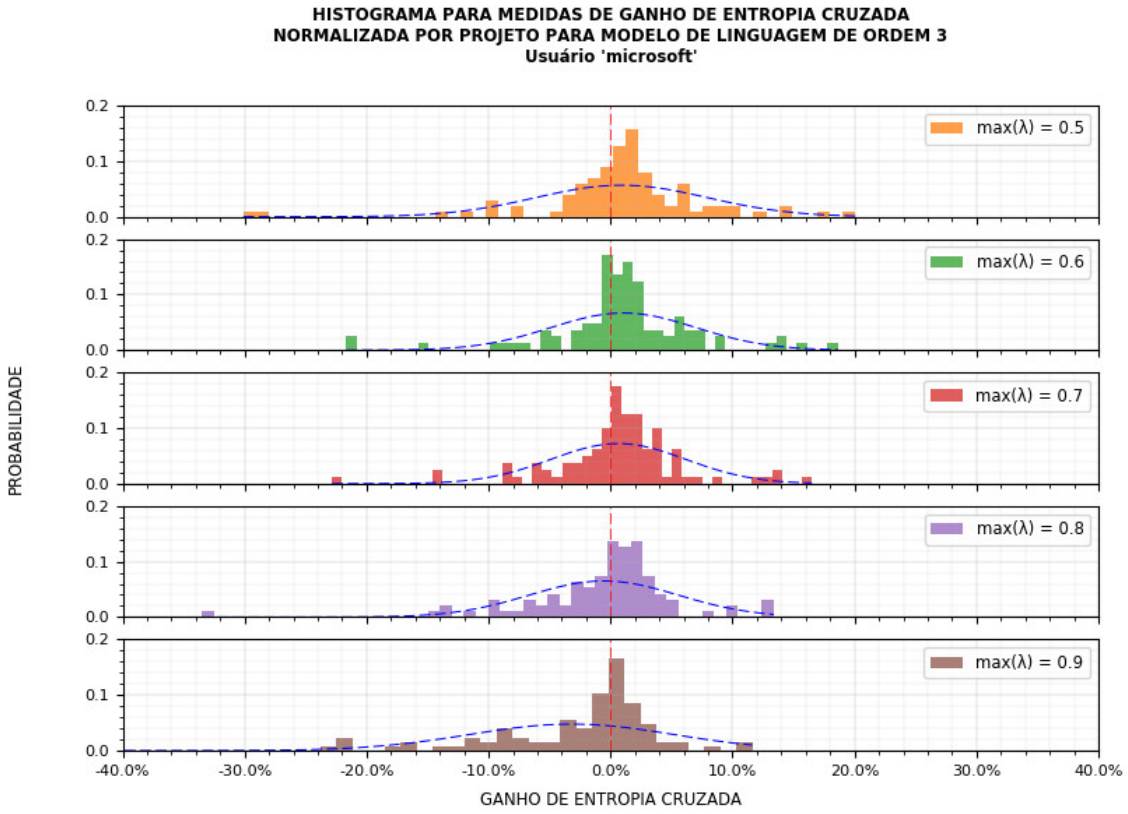
Figura 56 – Ganho de entropia cruzada normalizada versus linha de base definida a partir de um modelo com cache – usuário ‘microsoft’.



Fonte: Autor.

Em razão da quantidade significativa de pontos exibidos na Figura 56, opta-se por apresentar uma alternativa para a visualização através do histograma da Figura 57. A linha tracejada vermelha em $x = 0$ representa a linha de base, isto é, a medida de entropia cruzada normalizada para o modelo com suporte a *cache* simulado. A curva tracejada azul representa a função densidade de probabilidade ajustada a partir do conjunto de dados obtido.

Figura 57 – Histograma de ganhos de entropia cruzada normalizada por projeto para as diferentes configurações de favorecimento de modelos de projeto – usuário ‘microsoft’.

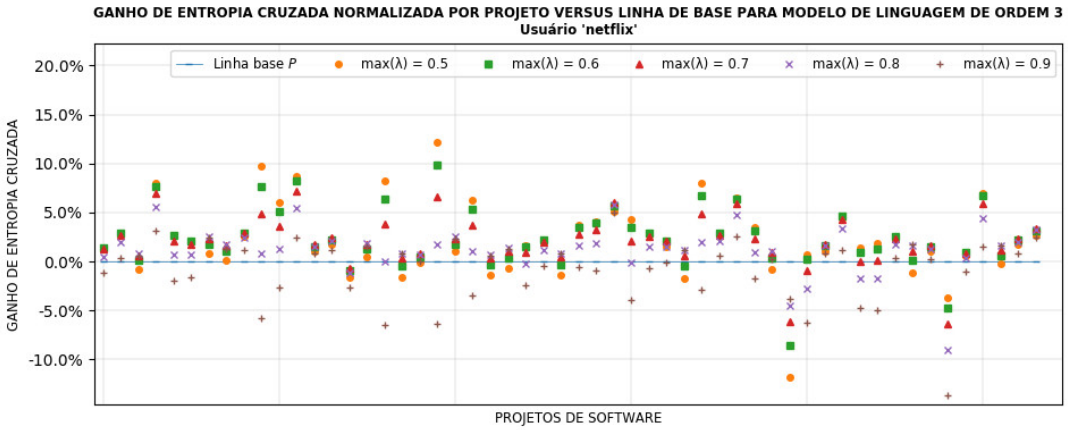


Fonte: Autor.

Novamente, verifica-se que grande parte dos ganhos de entropia cruzada para as diferentes configurações de favorecimento de modelos de projeto ainda é de valores positivos, e que eles estão relativamente bem distribuídos nas proximidades de $x = 0$. Logo, além de uma redução nos ganhos obtidos em comparação com as análises da seção 4.4.2 (conforme Figura 48), observa-se que a distribuição de ganhos se torna cada vez menor à medida que o maior valor para o peso λ progride. Especialmente para os casos com $\lambda \geq 0{,}8$, a adoção do modelo proposto proporcionaria ganhos inferiores a modelos com suporte a *cache* simulado.

A Figura 58 ilustra as diferenças na medida de entropia cruzada normalizada por projeto para o usuário `netflix`.

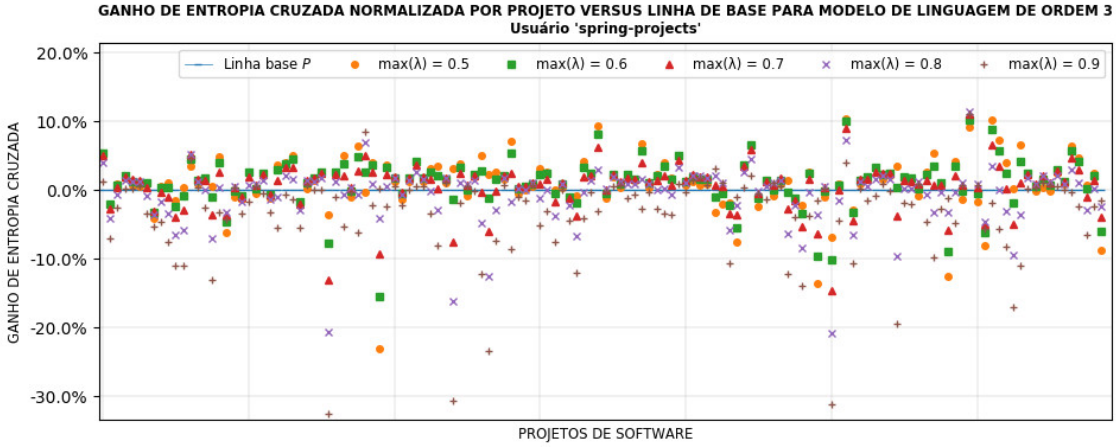
Figura 58 – Ganho de entropia cruzada normalizada versus linha de base definida a partir de um modelo com cache – usuário ‘netflix’.



Fonte: Autor.

A Figura 59 ilustra as diferenças na medida de entropia cruzada normalizada por projeto para o usuário `spring-projects`.

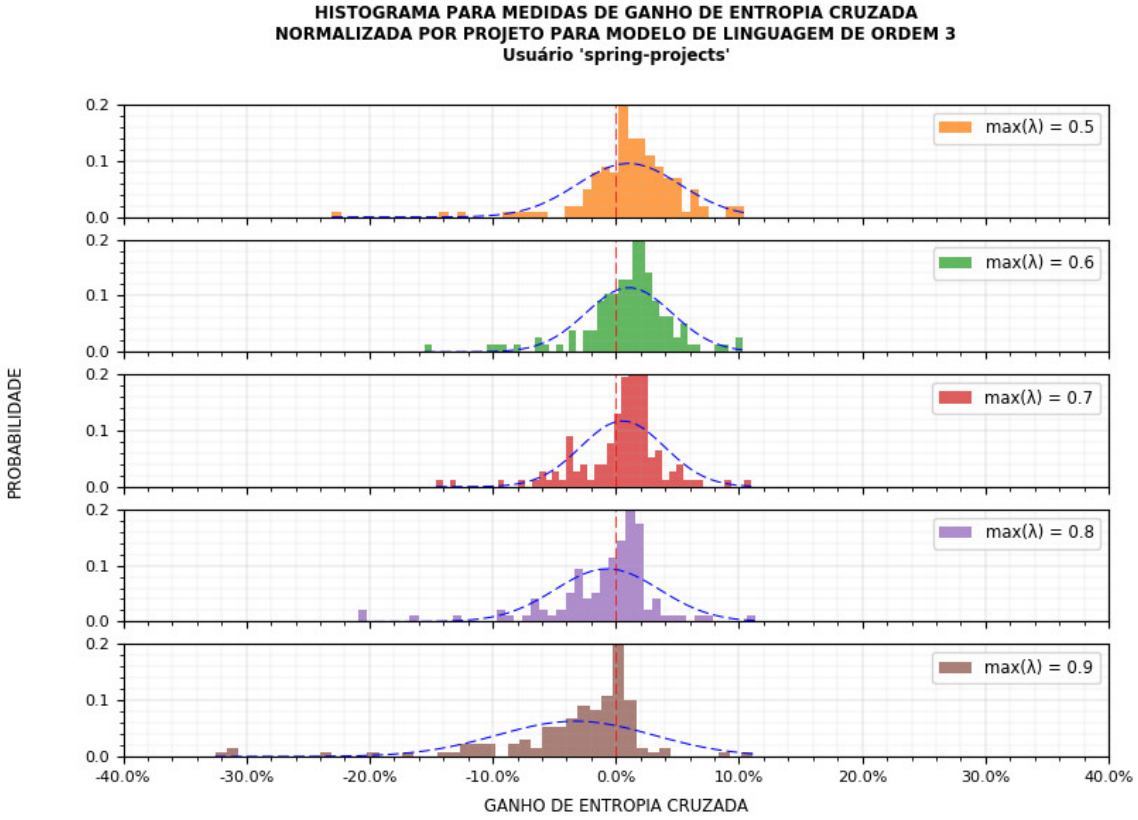
Figura 59 – Ganho de entropia cruzada normalizada versus linha de base definida a partir de um modelo com cache – usuário ‘spring-projects’.



Fonte: Autor.

Dado que a quantidade de pontos exibidos na Figura 59 é significativa, opta-se, uma vez mais, por apresentar uma alternativa para a visualização através do histograma da Figura 60. A linha tracejada vermelha em $x = 0$ representa a linha de base, isto é, a medida de entropia cruzada normalizada para o modelo com suporte a *cache* simulado. A curva tracejada azul representa a função densidade de probabilidade ajustada a partir do conjunto de dados obtido.

Figura 60 – Histograma de ganhos de entropia cruzada normalizada por projeto para as diferentes configurações de favorecimento de modelos de projeto – usuário 'spring-projects'.



Fonte: Autor.

Nota-se que grande parte dos ganhos de entropia cruzada para as diferentes configurações de favorecimento de modelos de projeto ainda é de valores positivos e relativamente bem distribuídos nas proximidades de $x = 0$. Logo, além de uma redução nos ganhos obtidos em comparação com as análises da seção 4.4.2 (conforme Figura 51), observa-se que a distribuição de ganhos se torna cada vez menor à medida que o maior valor para o peso λ progride.

As avaliações realizadas ao longo dessa seção indicam mais uma vez que a combinação de modelos de linguagem apresenta ganhos nas medidas de entropia cruzada de um projeto de software até mesmo se comparados a modelos de linguagem com suporte a *cache*, tais como os exemplos de Tu, Su e Devanbu (2014) e Hellendoorn e Devanbu (2017).

Entretanto, vale ressaltar que em todos os usuários estudados houve redução em todas as medidas de ganho de entropia cruzada em comparação às análises da seção 4.4.2, conforme observado na Tabela 8. Essa redução pode ser explicada pela relevância que os modelos mais específicos (no caso o modelo do nível hierárquico de projeto) apresentam para a capacidade preditiva em comparação a modelos mais genéricos.

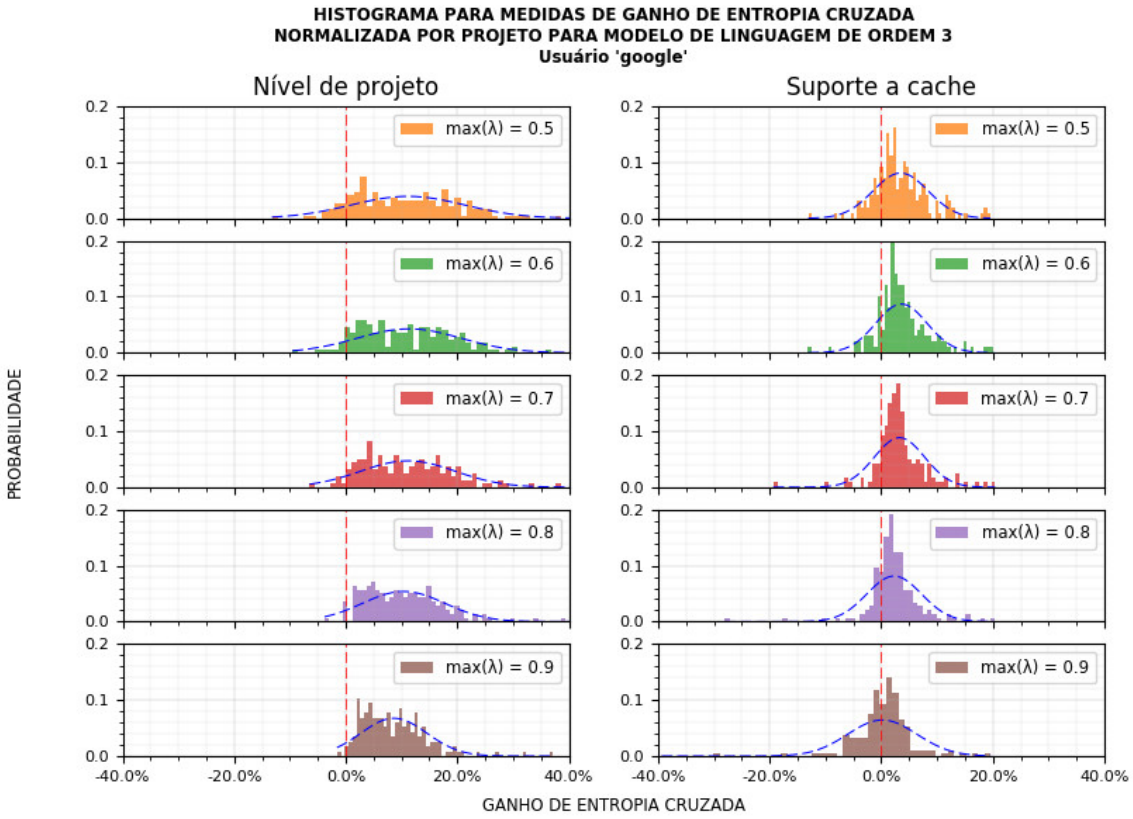
Tabela 8 – Comparação entre os ganhos de entropia cruzada normalizada considerando a linha de base de projeto e de modelos com suporte a cache.

Usuário (1)	**Menor ganho % (2)**		**Maior ganho % (3)**		**Ganho % médio (4)**		**Mediana (5)**	
	P (4.4.2)	***cache*** (4.4.3)	***P*** (4.4.2)	***cache*** (4.4.3)	***P*** (4.4.2)	***cache*** (4.4.3)	***P*** (4.4.2)	***cache*** (4.4.3)
amzn	2.57%	-5.45%	20.72%	6.91%	10.41%	0.72%	8.97%	0.91%
facebook	-4.32%	-0.17%	28.17%	14.86%	8.88%	3.63%	9.43%	1.63%
google	-6.93%	-22.22%	44.82%	20.00%	10.54%	2.57%	9.74%	2.16%
microsoft	-18.93%	-54.70%	45.52%	44.14%	12.28%	-0.48%	9.99%	0.24%
netflix	1.09%	-7.49%	29.87%	6.40%	10.01%	1.43%	8.30%	1.39%
spring-projects	-14.69%	-16.76%	41.14%	10.59%	12.38%	-0.26%	10.96%	0.68%

Fonte: Autor.

Um exemplo para essa redução está em destaque na Figura 61, que representa as distribuições dos ganhos comparados a uma linha de base definida pelo modelo de linguagem do nível de projeto (indicadas pelos histogramas do lado esquerdo obtidos da seção 4.4.2) e as distribuições dos ganhos comparados a uma linha base definida pelo modelo de linguagem com suporte a *cache* simulado (indicadas pelos histogramas do lado direito). Observa-se que o modelo proposto apresenta ganhos quando comparado a modelos de suporte a *cache* ainda que esses ganhos sejam menores e mais concentrados nas proximidades de $x = 0$.

Figura 61 – Comparação das distribuições dos ganhos de entropia cruzada para o usuário ‘google’.



Fonte: Autor.

4.5 NOTAS SOBRE A AVALIAÇÃO

O arcabouço de criação e validação de modelos de linguagem[22] utilizado para as avaliações registradas neste capítulo foi desenvolvido exclusivamente para este trabalho em linguagem Python. Ele utiliza uma série de bibliotecas de terceiros, dentre elas: `javalang`[23] para interpretação dos arquivos de código-fonte Java em árvores AST; `spacy`[24] para tokenização e lematização de texto em linguagem natural; `numpy`[25] e `scipy`[26] para manipulação numérica; `pandas`[27] para manipulação de vetores de dados; `scikit`[28] para a modelagem de tópicos LDA; e `marisa_trie`[29] para armazenamento das contagens dos modelos n-gramas em estruturas de dados enxutas conhecidas como *tries*, incluindo suporte a busca por prefixo, e `matplotlib`[30] para renderização de gráficos. Ademais, com base no trabalho de PLN-FAMAF (2018), o arcabouço implementa modelos n-gramas tradicionais, com suavização *add-one*, *backoff* em sua versão exaustiva e baseada em heurísticas, interpolação linear com pesos dependentes de contexto, além do Kneser-Ney padrão.

Todas as avaliações descritas nesta seção foram executadas em uma máquina virtual adquirida no serviço Google Cloud Platform[31] com 6 CPUs virtuais, 240GiB de armazenamento e 128GiB de memória RAM, em um ambiente operacional Linux CentOS 7 x64 com Python versão 3.6. Os picos de execução, particularmente observados durante a construção de modelos n-grama de ordens superiores (9 e 10) com o algoritmo de suavização Kneser-Ney padrão, atingiram 100% de consumo de CPU e mais de 64GiB de memória RAM.

[22] Disponível em https://gitlab.com/fvarrebola/lm4cc/.
[23] Disponível em https://github.com/c2nes/javalang.
[24] Disponível em https://spacy.io.
[25] Disponível em http://www.numpy.org.
[26] Disponível em https://scipy.org.
[27] Disponível em https://pandas.pydata.org.
[28] Disponível em http://scikit-learn.org.
[29] Disponível em https://github.com/pytries/marisa-trie.
[30] Disponível em https://matplotlib.org.
[31] Disponível em https://cloud.google.com.

4.6 DISCUSSÃO

Os resultados registrados ao longo deste capítulo indicam que considerar a segmentação hierárquica de modelos de linguagem é um caminho promissor para a tarefa de sugestão de código-fonte, uma vez que na grande maioria dos casos estudados os modelos mais específicos apresentaram medidas de entropia cruzada inferiores aos mais genéricos, em uma indicação de capacidade preditiva superior. Esses resultados estão em linha com o reportado por Tu, Su e Devanbu (2014), quando estes sugerem que código-fonte apresenta regularidades locais que podem ser melhor exploradas com a adoção do conceito de *cache*, no qual os termos recém-observados em um determinado arquivo de código-fonte ou diretório têm precedência sobre os demais. Vale observar, no entanto, que no modelo proposto por este trabalho, a segmentação é feita por níveis hierárquicos iniciados no escopo do projeto de software, que pode conter múltiplos arquivos e múltiplos diretórios. Em complemento, o modelo proposto por este trabalho também considera modelos de linguagem construídos a partir do usuário mantenedor, em um contraponto ao observado por Saraiva, Bird e Zimmermann (2015), que avaliam usuários individuais e constatam que suas regularidades são pouco significativas se comparadas às regularidades de projeto.

Os resultados também indicam que a combinação dos modelos de linguagem, particularmente através de técnicas de interpolação linear, desempenha papel fundamental para a obtenção de ganhos na capacidade preditiva, conforme estudo registrado na seção 4.4.2. Ao comparar o modelo proposto com alternativas que utilizam a mesma técnica para favorecer regularidades locais perante regularidades globais, percebe-se que os ganhos se mantêm, ainda que em grau menor, conforme estudo registrado na seção 4.4.3.

A escolha de técnicas de modelagem de tópicos latentes para a segmentação de modelos mais abrangentes é assunto que também merece atenção. Acredita-se haver espaço para ganhos ainda maiores com a adoção deste tipo de técnica, uma vez que o algoritmo utilizado por este trabalho – o LDA – é apenas uma das possíveis alternativas para segmentar as regularidades intra ou inter-usuário. Acredita-se ainda que os eventuais ganhos possam ter sido minimizados em virtude das pré-condições estabelecidas na seção 4.3.1, ou ainda em virtude das características não-supervisionadas inerentes ao algoritmo.

Um risco importante para os resultados apresentados por este trabalho é a aplicabilidade da metodologia de avaliação e a capacidade de generalização da abordagem proposta em outros conjuntos de dados. Nesse sentido, o Apêndice F, de maneira similar ao observado na seção 4.4, concentra os resultados das avaliações realizadas com o corpo de dados `CD2` definido na seção 4.2. Pode-se observar que apesar de se tratar de um corpo de dados mais abrangente em relação ao corpo de dados `CD1`, os resultados obtidos proporcionam as mesmas conclusões registradas até aqui; de que a combinação dos modelos de linguagem desempenha papel fundamental para a obtenção de ganhos na capacidade preditiva.

Outro risco importante para os resultados apresentados é a ausência de comparações diretas das medidas obtidas com eventuais trabalhos relacionados. Ainda que trabalhos tais como Nguyen et al. (2013), Tu Su e Devanbu (2014) e Hellendoorn e Devanbu (2017) utilizem modelos de linguagem n-grama, as comparações realizadas ao longo da seção 4.4.3 são adaptações, particularmente do trabalho de Tu Su e Devanbu (2014). Vale lembrar que se trata de uma adaptação pois no trabalho original o *cache* é construído a partir do conteúdo de um único arquivo de código-fonte e, eventualmente, dos arquivos encontrados no mesmo diretório e subdiretórios próximos ao arquivo original. Nas análises realizadas ao longo deste capítulo, o *cache* foi substituído pelo modelo de linguagem n-grama construído a partir do projeto de software.

Por fim, um risco adicional é a implementação proprietária de todos os algoritmos de suavização utilizados para as avaliações. No entanto, esse risco foi minimizado através de avaliações que asseguram uma precisão da ordem de 10^{-7} para todos os algoritmos avaliados, em especial o método de interpolação com pesos dependentes do contexto e o Kneser-Ney. Ainda em relação aos riscos relacionados à implementação, cabe ressaltar que a versão do algoritmo Kneser-Ney utilizada considera um único peso δ de desconto absoluto, quando a versão mais popular deste algoritmo utiliza, na verdade, um peso δ diferente para cada uma das ordens que compõem o modelo de linguagem final. Neste caso, a versão que utiliza múltiplos valores de δ é comumente chamada de Kneser-Ney modificado.

5 CONCLUSÃO E TRABALHOS FUTUROS

O estudo sobre a aplicação de modelos de linguagem n-grama ao problema da sugestão de código-fonte remonta ao trabalho de Hindle et al. (2012). Desde então, a quantidade de estudos sobre o tema ganhou certa proporção, com destaque para os trabalhos de Allamanis e Sutton (2013), Nguyen et al. (2013), Tu, Su e Devanbu (2014) e Hellendoorn e Devanbu (2017). No entanto, a importância do contexto na capacidade preditiva de um modelo n-grama é assunto pouco estudado, restrito aos trabalhos de Tu, Su e Devanbu (2014) e Hellendoorn e Devanbu (2017), que operam com modelos de linguagem com suporte a *cache*. Dessa forma, o trabalho desenvolvido contribui para os estudos sobre a aplicação de modelos de linguagem n-grama no âmbito do problema da sugestão de código-fonte comprovando que um modelo composto por modelos de linguagem segmentados de maneira hierárquica é capaz de proporcionar capacidade preditiva superior a modelos individuais ou ainda a modelos com suporte a *cache*.

Ao longo do capítulo 4 é possível observar uma série de avaliações e resultados que atestam essas afirmações. Através de um conjunto de avaliações majoritariamente intrínsecas – embora avaliações extrínsecas também tenham sido realizadas – observa-se que, embora as menores medidas de entropia cruzada geralmente sejam obtidas a partir de modelos de linguagem mais específicos, a combinação de diversos níveis hierárquicos proporciona resultados ainda melhores. Com isso, esse trabalho também contribui para a definição e utilização de um modelo baseado em um contexto de invocação definido por modelos de linguagem n-grama segmentados por projeto de software, por usuário mantenedor, por agrupamentos intra e inter-usuário baseados em distribuições de tópicos latentes e por um modelo geral.

Finalmente, e considerando que essa abordagem não limita ou impede que outras formas de segmentação sejam consideradas, este trabalho indica uma abordagem promissora para trabalhos futuros sobre o tema, conforme listado na seção 5.1.

5.1 TRABALHOS FUTUROS

Considerando a abordagem adotada pelo trabalho e os resultados obtidos, sugerem-se as seguintes possibilidades de trabalhos futuros:

a) Incluir na composição do modelo proposto o papel sintático que um termo assume, através, por exemplo, do uso de gramáticas livres de contexto probabilísticas. As gramáticas poderiam ser utilizadas para a segmentação de modelos por trecho de código-fonte – poderia haver modelos para métodos ou laços – e os principais desafios dessa alternativa estariam relacionados à interpretação parcial de programas (DAGENAIS; HENDREN, 2008);
b) Avaliar, em contrapartida à inferência de tópicos latentes a partir da frequência de termos observados, como uma eventual organização semântica de arquivos de código-fonte (MAHMOUD; BRADSHAW, 2016) influenciaria a capacidade preditiva do modelo proposto;
c) Incluir na composição do modelo proposto modelos de linguagem n-grama ortogonais àqueles considerados nesta pesquisa. Por exemplo: estabelecer modelos preditivos exclusivos de uma API poderia trazer impactos positivos para a capacidade preditiva;
d) Avaliar formas alternativas à interpolação linear para a combinação dos modelos de linguagem n-grama que compõem o contexto;
e) Estudar o comportamento do modelo proposto em conjuntos de dados que considerem linguagens de programação variadas, ou ainda conjunto de dados mais abrangentes, a exemplo do recente trabalho de Markovtsev e Long (2018);
f) Avaliar a aplicação do modelo proposto para fins de ensino e aprendizagem em programação;
g) Incorporar o modelo proposto a um assistente de código-fonte de tal maneira que este seja passível de uso em um ambiente integrado de desenvolvimento;
h) Avaliar o efeito que um assistente de código-fonte construído de acordo com as premissas do item anterior teria no desempenho de programadores profissionais no exercício de suas atividades. As dificuldades para um estudo dessa magnitude incluem a definição das atividades de programação a serem desempenhadas e a medida de conhecimento especializado dos sujeitos sob

avaliação para a devida segmentação em grupos. Estudos desse tipo, conforme observado em Arrebola e Aquino Junior (2017), tendem a despender grande quantidade de recursos por serem realizados de maneira longitudinal; e

i) Incorporar a experiência em programação do usuário final à hierarquia de contextos em uma segmentação por personas seguindo os princípios estabelecidos pelos trabalhos de De Araujo e Aquino (2014) e Masiero e Aquino (2015). A incorporação desse nível hierárquico seria similar à segmentação de projetos de software por objetivos e funcionalidades inter-usuário. A partir dessa incorporação também seria possível utilizar um modelo de aprendizagem por reforço capaz de adaptar-se a eventuais preferências do usuário final.

REFERÊNCIAS

ALLAMANIS, M.; SUTTON, C. **Mining source code repositories at massive scale using language modeling**. 2013 10th Working Conference on Mining Software Repositories (MSR). **Anais** ... IEEE, 2013. Disponível em: <https://doi.org/10.1109/MSR.2013.6624029>.

AMANN, S. et al. **A Study of Visual Studio Usage in Practice**. Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER '16). **Anais** ... 2016. Disponível em: <https://doi.org/10.1109/SANER.2016.39>.

ARREBOLA, F. V.; AQUINO JUNIOR, P. T. On source code completion assistants and the need of a context-aware approach. In: **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**. [s.l: s.n.]. v. 10274 LNCSp. 191–201.

ASADUZZAMAN, M. et al. CSCC: Simple, efficient, context sensitive code completion. **Proceedings - 30th International Conference on Software Maintenance and Evolution, ICSME 2014**, p. 71–80, 2014.

ASADUZZAMAN, M. et al. A Simple, Efficient, Context-sensitive Approach for Code Completion. **Journal of Software: Evolution and Process**, v. 28, n. 7, p. 512–541, jul. 2016.

BALDI, P. F. et al. A theory of aspects as latent topics. **ACM SIGPLAN Notices**, v. 43, n. 10, p. 543, 2008.

BIELIK, P.; RAYCHEV, V.; VECHEV, M. PHOG: Probabilistic Model for Code. **International Conference on Machine Learning (ICML'16)**, v. 48, p. 2933–2942, 2016.

BINKLEY, D. et al. Understanding LDA in source code analysis. **Proceedings of the 22nd International Conference on Program Comprehension**, v. undefined, n. undefined, p. 26–36, 2014.

BLEI, D. M. et al. Latent Dirichlet Allocation. **Journal of Machine Learning Research**, v. 3, p. 993–1022, 2003.

BRUCH, M. et al. Learning from examples to improve code completion systems. **Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium - E**, p. 213, 2009.

CHARIKAR, M. S. **Similarity estimation techniques from rounding algorithms**. Proceedings of the thiry-fourth annual ACM symposium on Theory of computing - STOC '02. **Anais** ... New York, New York, USA: ACM Press, 2002. Disponível em: <http://portal.acm.org/citation.cfm?doid=509907.509965>.

CHEN, S. F.; GOODMAN, J. An empirical study of smoothing techniques for language modeling. **Computer Speech & Language**, v. 13, n. 4, p. 359–393, out. 1999.

CHOMSKY, N. Three models for the description of language. **IRE Transactions on Information Theory**, v. 2, n. 3, p. 113–124, 1956.

CHURCH, L.; NASH, C.; BLACKWELL, A. F. Liveness in Notation Use: From Music to Programming. **In Proceedings of the 22nd Annual Workshop of the Psychology of Programming Interest Group (PPIG 2010)**, n. January 2010, p. 2–11, 2010.

COVER, T.; HART, P. Nearest neighbor pattern classification. **IEEE Transactions on Information Theory**, v. 13, n. 1, p. 21–27, 1967.

DAGENAIS, B.; HENDREN, L. **Enabling Static Analysis for Partial Java Programs**. Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications. **Anais** ... 2008. Disponível em: <http://doi.acm.org/10.1145/1449764.1449790>.

DE ARAUJO, C. F.; AQUINO, P. T. **Psychological personas for universal user modeling in human-computer interaction**. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). **Anais** ... 2014.

DEERWESTER, S. et al. Indexing by Latent Semantic Analysis. **Journal of the American Society for Information Science**, v. 41, p. 391–407, 1990.

DEY, A. K. Understanding and Using Context. **Journal of Personal Ubiquitous Computing**, v. 5, n. 1, p. 4–7, 2001.

ECLIPSE FOUNDATION, I. **Eclipse - The Eclipse Foundation open source community website**. Disponível em: <https://www.eclipse.org>. Acesso em: 15 maio. 2018.

GVERO, T.; KUNCAK, V. Synthesizing Java expressions from free-form queries. **Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2015**, p. 416–432, 2015.

HAMMING, R. W. Error Detecting and Error Correcting Codes. **Bell System Technical**

Journal, v. 29, n. 2, p. 147–160, 1950.

HEAFIELD, K. KenLM : Faster and Smaller Language Model Queries. **Proceedings of the Sixth Workshop on Statistical Machine Translation**, n. 2009, p. 187–197, 2011.

HEILMAN, M. et al. Combining Lexical and Grammatical Features to Improve Readability Measures for First and Second Language Texts. **Computational Linguistics**, p. 460–467, 2007.

HELLENDOORN, V. J.; DEVANBU, P. **Are deep neural networks the best choice for modeling source code?** Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017. **Anais** ... New York, New York, USA: ACM Press, 2017. Disponível em: <http://dl.acm.org/citation.cfm?doid=3106237.3106290>. Acesso em: 23 mar. 2018

HINDLE, A. et al. On the naturalness of software. **Proceedings - International Conference on Software Engineering**, p. 837–847, 2012.

javalang. Disponível em: <https://github.com/c2nes/javalang>. Acesso em: 9 fev. 2018.

JOZELFOWICZ, R. et al. Exploring the Limits of Language Modeling. **arxiv:1602.02410**, 2016.

JURAFSKY, D.; MARTIN, J. H. **Speech and language processing : an introduction to natural language processing, computational linguistics, and speech recognition**. London [u.a.]: Pearson Prentice Hall, 2009.

KALTZ, J. W.; ZIEGLER, J.; LOHMANN, S. Context-aware web engineering: Modeling and applications. **Revue d'Intelligence Artificielle**, v. 19, n. 3, p. 439–458, 1 jun. 2005.

KNESER, R.; NEY, H. **Improved backing-off for M-gram language modeling**. 1995 International Conference on Acoustics, Speech, and Signal Processing. **Anais** ... IEEE, 1995. Disponível em: <http://ieeexplore.ieee.org/document/479394/>. Acesso em: 31 mar. 2018

KUHN, R.; MORI, R. DE. A Cache-Based Natural Language Model for Speech Recognition. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, v. 12, n. 6, p. 570–583, 1990.

MAHMOUD, A.; BRADSHAW, G. Semantic topic models for source code analysis. **Empirical Software Engineering**, 2016.

MALETIC, J. I.; MARCUS, A. Using latent semantic analysis to identify similarities in

source code to support program understanding. **Tools with Artificial Intelligence, 2000. ICTAI 2000. Proceedings. 12th IEEE International Conference on**, p. 46–53, 2000.

MALETIC, J. I.; VALLURI, N. Automatic software clustering via Latent Semantic Analysis. **14th IEEE International Conference on Automated Software Engineering**, p. 251–254, 1999.

MANNING, C. D.; RAGHAVAN, P.; SCHÜTZE, H. **Introduction to Information Retrieval Introduction**. [s.l: s.n.]. v. 35

MARASOIU, M.; CHURCH, L.; BLACKWELL, A. **An empirical investigation of code completion usage by professional software developers**. Proceedings of the 26th Annual Workshop of the Psychology of Programming Interest Group ({PPIG} 2015). **Anais** ... 2015.

MARKOVTSEV, V.; KANT, E. **Topic modeling of public repositories at scale using names in source code**. Disponível em: <https://arxiv.org/abs/1704.00135>. Acesso em: 1 abr. 2018.

MARKOVTSEV, V.; LONG, W. **Public git archive**. Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18. **Anais** ... New York, New York, USA: ACM Press, 20 mar. 2018. Disponível em: <http://arxiv.org/abs/1803.10144>.

MASIERO, A. A.; AQUINO, P. T. **Creating personas to reuse on diversified projects**. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). **Anais** ... 2015.

MCCABE, T. J. A Complexity Measure. **IEEE Transactions on Software Engineering**, v. SE-2, n. 4, p. 308–320, 1976.

MURPHY, G. C.; KERSTEN, M.; FINDLATER, L. How are java software developers using the eclipse IDE? **IEEE Software**, v. 23, n. 4, p. 76–83, 2006.

NGUYEN, A. T. et al. **Duplicate bug report detection with a combination of information retrieval and topic modeling**. Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012. **Anais** ... 2012. Disponível em: <http://dl.acm.org/citation.cfm?doid=2351676.2351687>.

NGUYEN, T. T. et al. A Statistical Semantic Language Model for Source Code. **Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering**, p. 532–542, 2013.

ORDING, M. **Context-Sensitive Code Completion Improving Predictions with**

Genetic Algorithms. [s.l.] KTH ROYAL INSTITUTE OF TECHNOLOGY, 2016.

PAIVA, E. et al. Factors that Influence the Productivity of Software Developers in a Developer View. In: **Innovations in Computing Sciences and Software Engineering**. Dordrecht: Springer Netherlands, 2010. p. 99–104.

PLN-FAMAF. **Procesamiento de Lenguaje Natural - UBA 2018**. Disponível em: <https://github.com/PLN-FaMAF/PLN-UBA2018>. Acesso em: 6 abr. 2018.

PROKSCH, S. et al. **Evaluating the evaluations of code recommender systems: a reality check**. Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016. **Anais** ... New York, New York, USA: ACM Press, 2016. Disponível em: <http://dl.acm.org/citation.cfm?doid=2970276.2970330>.

PROKSCH, S.; LERCH, J.; MEZINI, M. Intelligent Code Completion with Bayesian Networks. **St.Informatik.Tu-Darmstadt.De**, v. 25, n. 1, p. 1–31, 2015.

RAYCHEV, V.; BIELIK, P.; VECHEV, M. Probabilistic model for code with decision trees. **ACM SIGPLAN Notices**, v. 51, n. 10, p. 731–747, 19 out. 2016.

RAYCHEV, V.; VECHEV, M.; YAHAV, E. Code completion with statistical language models. **Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '14**, p. 419–428, 2013.

ROBBES, R.; LANZA, M. Improving code completion with program history. **Automated Software Engineering**, v. 17, n. 2, p. 181–212, 2010.

SAEIDI, A. M. et al. ITMViz: Interactive Topic Modeling for Source Code Analysis. **IEEE International Conference on Program Comprehension**, v. 2015–Augus, p. 295–298, 2015.

SALTON, G.; MCGILL, M. J. **Introduction to modern information retrieval.** [s.l: s.n.].

SARAIVA, J.; BIRD, C.; ZIMMERMANN, T. **Products, Developers, and Milestones: How Should I Build My N-Gram Language Model**. Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. **Anais** ... 2015. Disponível em: <http://dl.acm.org/citation.cfm?doid=2786805.2804431>.

SCHILIT, B.; ADAMS, N.; WANT, R. Context-aware computing applications. **Mobile Computing Systems and Applications**, p. 1–7, 1994.

SIEVERT, C.; SHIRLEY, K. LDAvis: A method for visualizing and interpreting topics.

Proceedings of the Workshop on Interactive Language Learning, Visualization, and Interfaces, p. 63–70, 2014.

STOLCKE, A. Srilm — an Extensible Language Modeling Toolkit. **Interspeech**, v. 2, n. Denver, Colorado, p. 901–904, 2002.

TRENDOWICZ, A.; MÜNCH, J. Chapter 6 Factors Influencing Software Development Productivity-State-of-the-Art and Industrial Experiences. **Advances in Computers**, v. 77, n. 09, p. 185–241, 2009.

TU, Z.; SU, Z.; DEVANBU, P. On the localness of software. **Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014**, p. 269–280, 2014.

WHITE, M. et al. Toward deep learning software repositories. **2015 IEEE/ACM 12th Working Conference on Mining Software Repositories**, p. 334–345, 2015.

ZHANG, C. et al. **Automatic parameter recommendation for practical API usage**. Proceedings - International Conference on Software Engineering. **Anais** ... 2012.

APÊNDICE A – GRAMÁTICAS LIVRES DE CONTEXTO

Geralmente, as sentenças manipuladas pelos modelos de linguagem mencionados neste trabalho são produzidas de acordo com alguma regra gramatical. Para ilustrar essa afirmação, considere linguagens naturais com o português, nas quais é comum observar, por exemplo, a existência de sentenças concordantes com a estrutura de *sujeito-verbo-predicado*. Essa regra, quando generalizada e combinada à todas as demais regras gramaticais de uma linguagem, representa as maneiras através das quais os símbolos de uma linguagem podem ser agrupados e ordenados.

Considerando o estudo de linguagens formais, as gramáticas são divididas em quatro grandes categorias de acordo com a hierarquia de Chomsky (1956), e para os propósitos deste trabalho, as gramáticas de tipo-2, ou gramáticas livres de contexto, são as de maior interesse para esta explicação, já que elas podem ser consideradas os pilares da maioria das linguagens de programação[32].

De maneira geral, uma gramática formal, ao definir as regras produtivas, leva em consideração dois grandes conjuntos de símbolos; os símbolos terminais, que podem ser entendidos como os termos do vocabulário; e o símbolos não-terminais, que podem ser entendidos como as classes de termos ou categorias sintáticas.

Formalmente, uma gramática livre de contexto pode ser definida por um 4-tupla $G = (N, \Sigma, R, S)$, sendo N o conjunto finito de símbolos não-terminais, Σ o conjunto finito de símbolos terminais, R o conjunto finito de regras na forma $X \rightarrow Y_1 Y_2 \ldots Y_n$ com $X \in N$, $n \geq 0$ e $Y_i \in (N \cup \Sigma)$ para $i = 1 \ldots n$, e $S \in N$ como o símbolo inicial.

Como exemplo, considere uma gramática hipotética relativamente simples, com conjuntos bem reduzidos de símbolos terminais Σ, não-terminais N e produções R conforme ilustrado na Figura 62.a. Essa gramática define as regras necessárias para que uma sentença $s \in \Sigma^*$[33], comumente chamada de produto, ou *yield*, seja considerada válida, conforme o exemplo *the man took the book* ilustrado na Figura 62.b, que também ilustra a árvore sintática relacionada à sentença e todas as derivações consideradas.

[32] Note que é possível argumentar que algumas linguagens de programação requerem gramáticas sensíveis ao contexto, devido ao escopo de variáveis. No entanto, essa distinção é irrelevante para os propósitos desse trabalho.

[33] Note que o símbolo Σ^* é utilizado para indicar o conjunto de todas as possíveis sentenças construídas a partir dos termos encontrados em Σ.

Figura 62 – Exemplo de gramática livre de contexto.

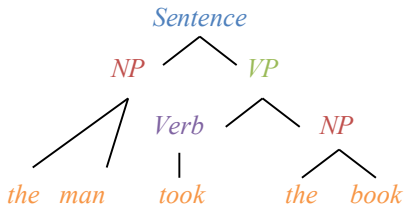
$$\boldsymbol{N} = \{Sentence, NP, VP, Verb\}$$

$$\boldsymbol{\Sigma} = \{the, man, took, book\}$$

$$\boldsymbol{R} = \left\{ \begin{array}{c} Sentence \rightarrow NP \quad VP, \\ VP \rightarrow Verb \quad NP, \\ NP \rightarrow the \mid man \mid book \\ Verb \rightarrow took \end{array} \right\}$$

$$\boldsymbol{S} = Sentence$$

(a) Exemplo de gramática livre de contexto.



(b) Sentença e árvore sintática associada.

Fonte: Autor “adaptado de” Chomsky (1956).

Em relação às árvores sintáticas, $\mathcal{T}_G$ representa o conjunto de todas as árvores possíveis em uma gramática G formadas a partir das regras R. E para cada $t \in \mathcal{T}_G$, a função $yield(t)$ representa uma sentença $s \in \Sigma^*$. Assim, dada uma sentença $s \in \Sigma^*$, $\mathcal{T}_G(s)$ representa o conjunto $\{t : t \in \mathcal{T}_G, yield(t) = s\}$, isto é, $\mathcal{T}_G(s)$ representa o conjunto de todas as árvores sintáticas associadas a s.

Considerando o problema da sugestão de código-fonte, as gramáticas têm grande importância, já que as linguagens de programação possuem ao menos uma definição formal que permite a transformação, ou interpretação, de um conjunto de caracteres de texto em uma árvore sintática.

Observe, por exemplo, o trecho diminuto da gramática da linguagem de programação Java versão 10[34] ilustrado na Figura 63, representado com uma notação proprietária[35].

Figura 63 – Trecho da gramática da linguagem de programação Java versão 10 – representação de uma unidade de compilação.

```
CompilationUnit:
    OrdinaryCompilationUnit
    ModularCompilationUnit
OrdinaryCompilationUnit:
    [PackageDeclaration] {ImportDeclaration} {TypeDeclaration}
```

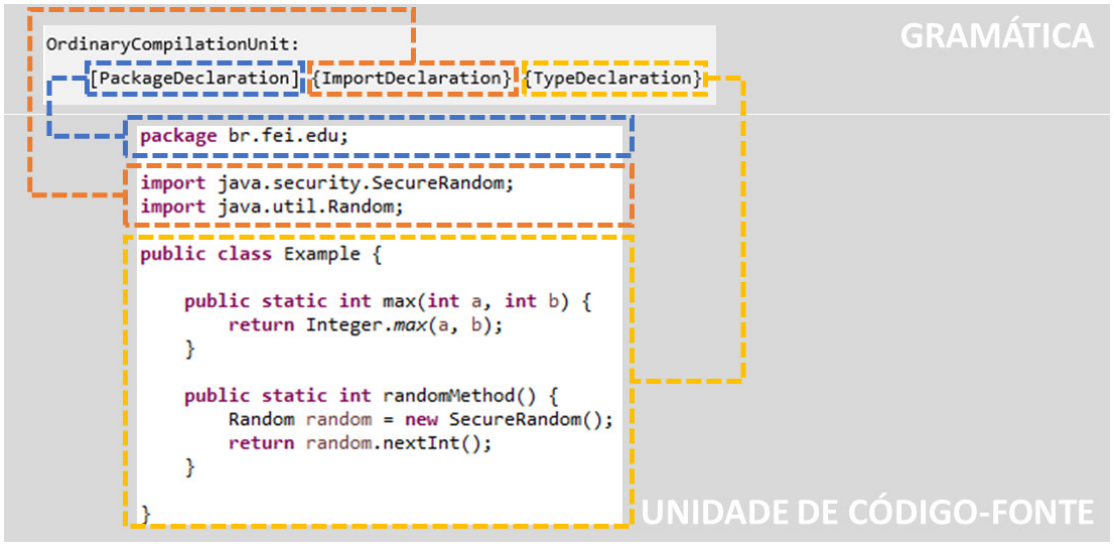
Fonte: Autor.

[34] Note que a especificação completa da linguagem de programação Java versão 10 pode ser encontrada em https://docs.oracle.com/javase/specs/jls/se10/html/jls-19.html.

[35] Note que o mais comum seria utilizar a notação *Backus-Naur Form* (BNF). No entanto, essa gramática proprietária é relativamente simples e intuitiva, e pode ser encontrada em https://docs.oracle.com/javase/specs/jls/se10/html/jls-2.html.

O trecho ilustrado na Figura 63 define que uma unidade de código-fonte é representada pelo símbolo `CompilationUnit` que pode ser expandido para os símbolos `OrdinaryCompilationUnit` ou `ModularCompilationUnit`. Particularmente, o símbolo `OrdinaryCompilationUnit` pode ser decomposto em uma combinação de símbolos[36] `PackageDeclaration`, `ImportDeclaration` e `TypeDeclaration`, conforme ilustrado na Figura 64, que representa ainda a associação entre a gramática e um exemplo concreto de unidade de código-fonte. Observe que o símbolo `PackageDeclaration`, que representa a declaração de um pacote, pode estar presente, no máximo, uma vez. Já o símbolo `ImportDeclaration`, que representa as declaração dos tipos externos que podem ser referenciados, pode estar presente muitas vezes, assim como `TypeDeclaration`, que representa uma definição de tipo.

Figura 64 – Representação da relação entre a gramática da linguagem de programação Java versão 10 e uma unidade de código-fonte.



Fonte: Autor.

[36] Note que o símbolo [é utilizado para encapsular um elemento opcional, que pode estar presente zero ou uma única vez. Já o símbolo { representa um elemento opcional que pode estar presente zero ou mais vezes.

APÊNDICE B – CARACTERÍSTICAS DOS CORPOS DE DADOS

O corpo de dados `CD1` está distribuído conforme a Tabela 9.

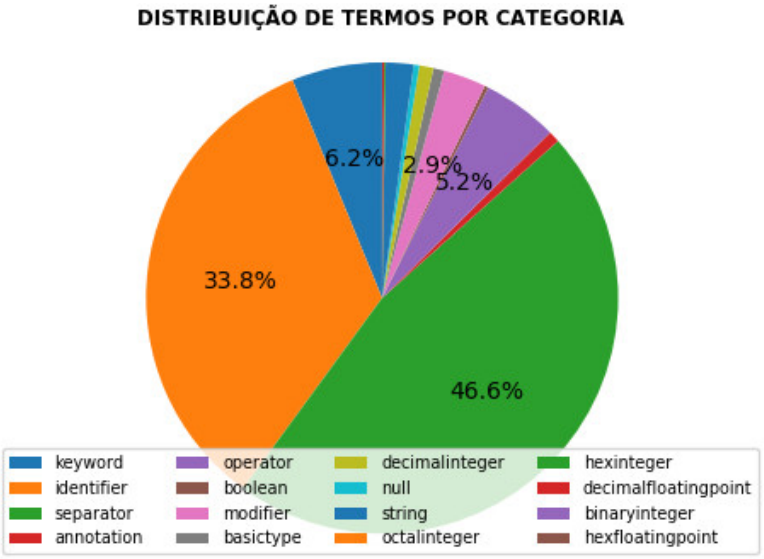
Tabela 9 – Distribuição de projetos e arquivos por usuário mantenedor para o corpo de dados CD1.

Usuário (1)	Projetos (2)	% do total (3)	Arquivos (4)	% do total (5)
amzn	9	1,84%	1.790	1,22%
facebook	19	3,89%	10.537	7,19%
google	172	35,17%	44.047	30,04%
microsoft	97	19,84%	25.294	17,25%
netflix	54	11,04%	10.435	7,12%
spring-projects	138	28,22%	54.507	37,18%
Total	**489**		**146.610**	

Fonte: Autor.

Os 100.702.054 termos estão distribuídos nas categorias ilustradas na Figura 65, e em grande parte concentrados nas categorias de separadores (46,6%) e identificadores (33,8%). Excluindo-se a categoria de separadores, que são inerentes à gramática da linguagem de programação, a alta concentração de identificadores no conjunto de termos observados está em linha com uma série de trabalhos relacionados, incluindo Allamanis e Sutton (2013).

Figura 65 – Distribuição de termos por categoria para o corpo de dados CD1.



Fonte: Autor.

O corpo de dados CD2 está distribuído conforme a Tabela 10.

Tabela 10 – Distribuição de projetos e arquivos por usuário mantenedor para o corpo de dados CD2.

Usuário (1)	Projetos (2)	% do total (3)	Arquivos (4)	% do total (5)
akquinet	11	0,63%	969	0,17%
allwinnerwk	41	2,36%	9.477	1,71%
android	35	2,01%	7.504	1,35%
apache	146	8,40%	154.408	27,88%
aptana	11	0,63%	12.950	2,34%
arquillian	27	1,55%	2.832	0,51%
caelum	20	1,15%	2.482	0,45%
cloudera	11	0,63%	1.255	0,23%
codjo	36	2,07%	2.856	0,52%
commonsguy	24	1,38%	1.385	0,25%
ctron	11	0,63%	2.686	0,49%
cyanogenmod	50	2,88%	8.003	1,45%
dawnscience	14	0,81%	2.748	0,50%
ddewaele	9	0,52%	62	0,01%
dvest	12	0,69%	65	0,01%
droolsjbpm	14	0,81%	18.828	3,40%
eclipse	78	4,49%	119.296	21,54%
elasticsearch	18	1,04%	2.657	0,48%
exoplatform	20	1,15%	32.085	5,79%
fasterxml	12	0,69%	1.085	0,20%
felixb	11	0,63%	147	0,03%
forge	10	0,58%	2.082	0,38%
fusebyexample	9	0,52%	378	0,07%
gatein	12	0,69%	4.242	0,77%
gckjdev	17	0,98%	550	0,10%
gemserk	10	0,58%	1.754	0,32%
greenlaw110	9	0,52%	404	0,07%
guardianproject	13	0,75%	539	0,10%
harism	9	0,52%	108	0,02%
heroku	14	0,81%	177	0,03%
hudson	10	0,58%	3.351	0,61%
hudson3-plugins	15	0,86%	658	0,12%

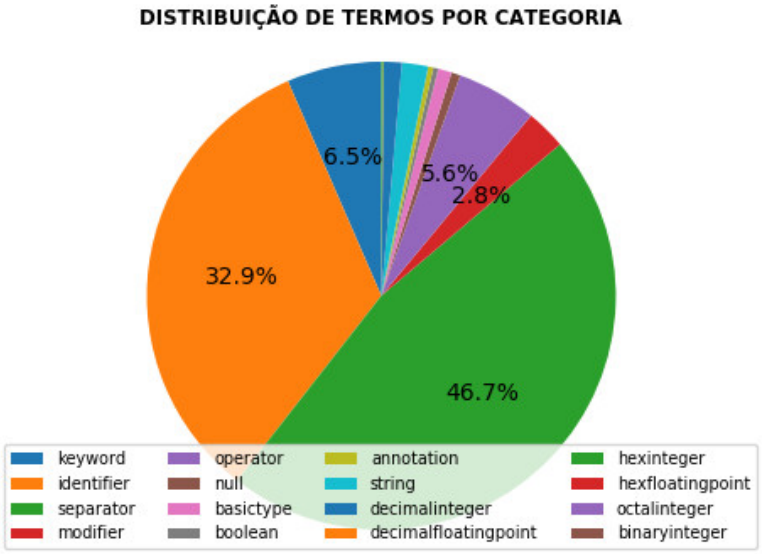
hudson-plugins	34	1,96%	911	0,16%
idega	12	0,69%	2.854	0,52%
jakewharton	18	1,04%	461	0,08%
jamesward	10	0,58%	53	0,01%
jasig	25	1,44%	3.026	0,55%
jayway	11	0,63%	982	0,18%
jboss	23	1,32%	4.066	0,73%
jbossas	16	0,92%	9.263	1,67%
jenkinsci	277	15,93%	6.922	1,25%
jetbrains	10	0,58%	50.300	9,08%
killme2008	10	0,58%	910	0,16%
krams915	9	0,52%	101	0,02%
marakana	10	0,58%	388	0,07%
mitchtech	19	1,09%	112	0,02%
mmominecraftdev	15	0,86%	70	0,01%
mulesoft	25	1,44%	5.933	1,07%
nathanmarz	9	0,52%	605	0,11%
neo4j	24	1,38%	2.897	0,52%
nesscomputing	20	1,15%	593	0,11%
nicolasgramlich	14	0,81%	839	0,15%
ning	12	0,69%	2.049	0,37%
njbartlett	9	0,52%	213	0,04%
novoda	9	0,52%	538	0,10%
nuxeo	20	1,15%	7.225	1,30%
openengsb	23	1,32%	867	0,16%
ops4j	11	0,63%	1.076	0,19%
pierre	10	0,58%	472	0,09%
powertac	14	0,81%	288	0,05%
propra12-orga	21	1,21%	647	0,12%
raphfrk	11	0,63%	248	0,04%
rtyley	10	0,58%	2.524	0,46%
salaboy	9	0,52%	1.083	0,20%
seam	19	1,09%	7.205	1,30%
sk89q	10	0,58%	1.581	0,29%
sonatype	51	2,93%	6.683	1,21%
spoutdev	12	0,69%	2.868	0,52%
springsource	71	4,08%	22.056	3,98%
stylingandroid	22	1,27%	42	0,01%

tdiesler	15	0,86%	873	0,16%
threerings	9	0,52%	4.030	0,73%
thymeleaf	9	0,52%	412	0,07%
twitter	9	0,52%	857	0,15%
waarp	10	0,58%	497	0,09%
webbukkit	13	0,75%	159	0,03%
Total	**1.739**		**553.802**	

Fonte: Autor.

Os 242.795.823 termos estão distribuídos nas categorias ilustradas na Figura 66, e em grande parte concentrados nas categorias de separadores (46,7%) e identificadores (32,9%). Excluindo-se a categoria de separadores, que são inerentes à gramática da linguagem de programação, nota-se novamente a alta concentração de identificadores.

Figura 66 – Distribuição de termos por categoria para o corpo de dados CD2.

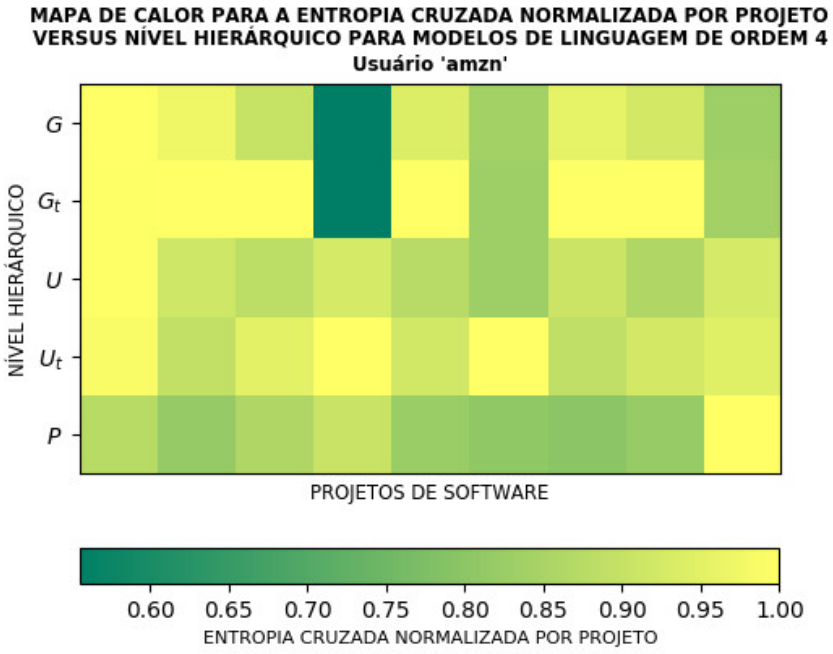


Fonte: Autor.

APÊNDICE C – MEDIDAS DE ENTROPIA CRUZADA EM CADA UM DOS NÍVEIS HIERÁRQUICOS PARA MODELOS DE LINGUAGEM DE ORDEM 4 COM ALGORITMO DE SUAVIZAÇÃO KNESER-NEY

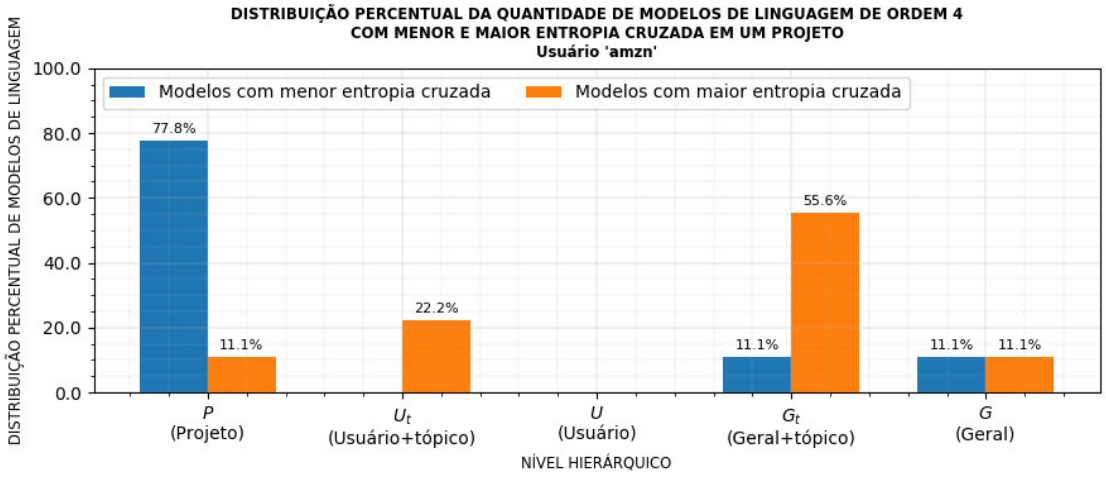
A Figura 67 representa o mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para o usuário `amzn`, e a Figura 68 ilustra a distribuição percentual da quantidade de modelos com menores e maiores medidas.

Figura 67 – Mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para modelos de linguagem de ordem 4 – usuário ‘amzn’.



Fonte: Autor.

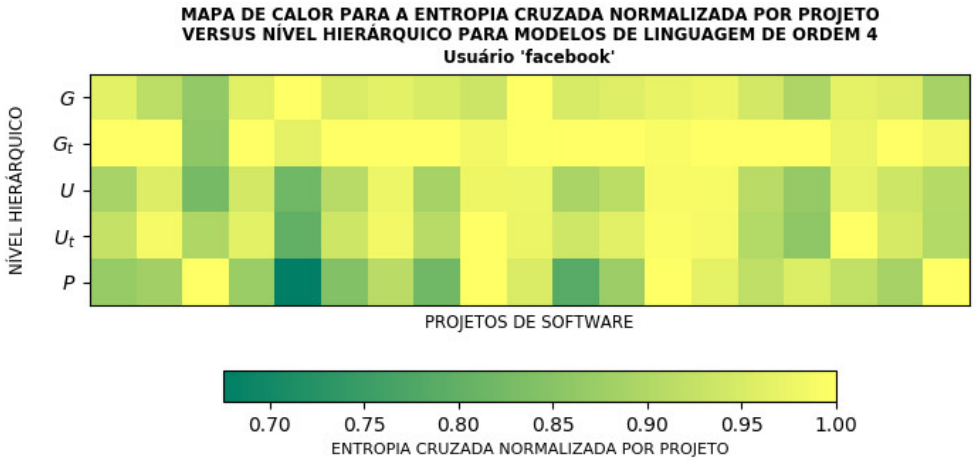
Figura 68 – Distribuição percentual da quantidade de modelos de linguagem de ordem 4 com menor e maior entropia cruzada em um projeto– usuário ‘amzn’.



Fonte: Autor.

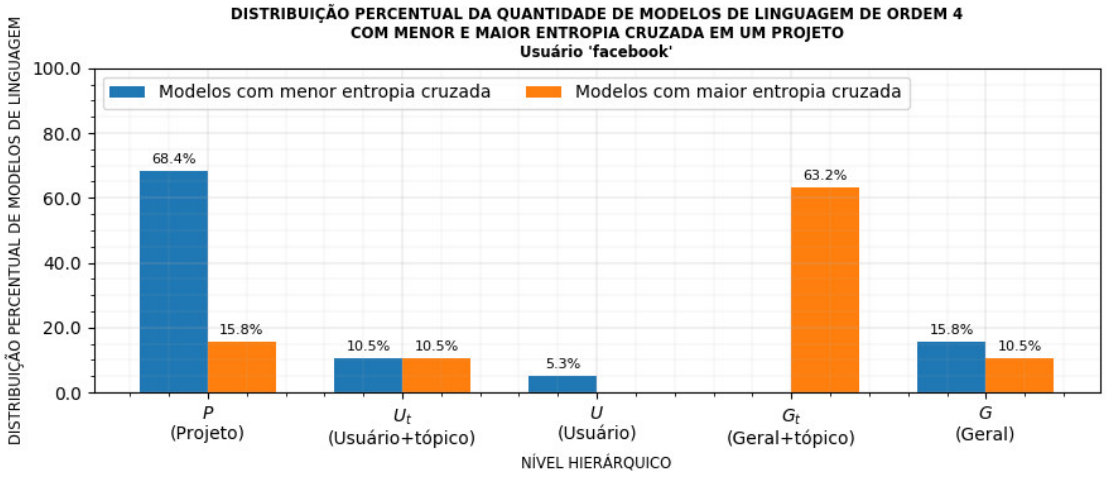
A Figura 69 representa o mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para o usuário `facebook`, e a Figura 70 ilustra a distribuição percentual da quantidade de modelos com menores e maiores medidas.

Figura 69 – Mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para modelos de linguagem de ordem 4 – usuário ‘facebook’.



Fonte: Autor.

Figura 70 – Distribuição percentual da quantidade de modelos de linguagem de ordem 4 com menor e maior entropia cruzada em um projeto– usuário ‘facebook’.



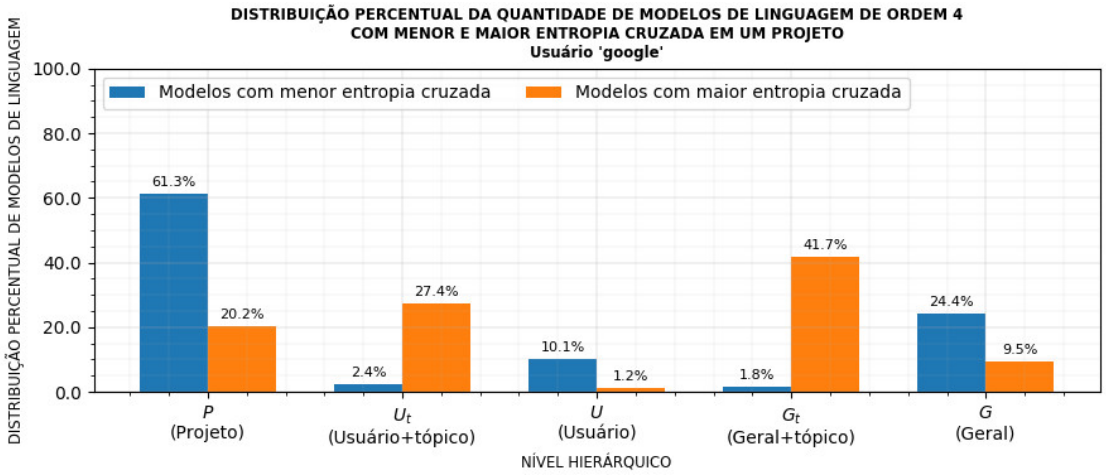
Fonte: Autor.

A Figura 71 representa o mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para o usuário `google`, e a Figura 72 ilustra a distribuição percentual da quantidade de modelos com menores e maiores medidas.

Figura 71 – Mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para modelos de linguagem de ordem 4 – usuário ‘google’.

Fonte: Autor.

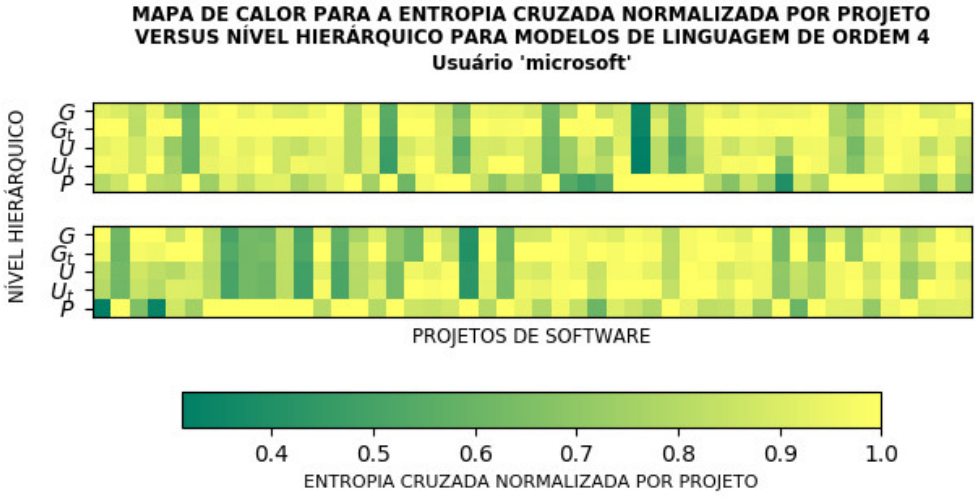
Figura 72 – Distribuição percentual da quantidade de modelos de linguagem de ordem 4 com menor e maior entropia cruzada em um projeto– usuário ‘google’.



Fonte: Autor.

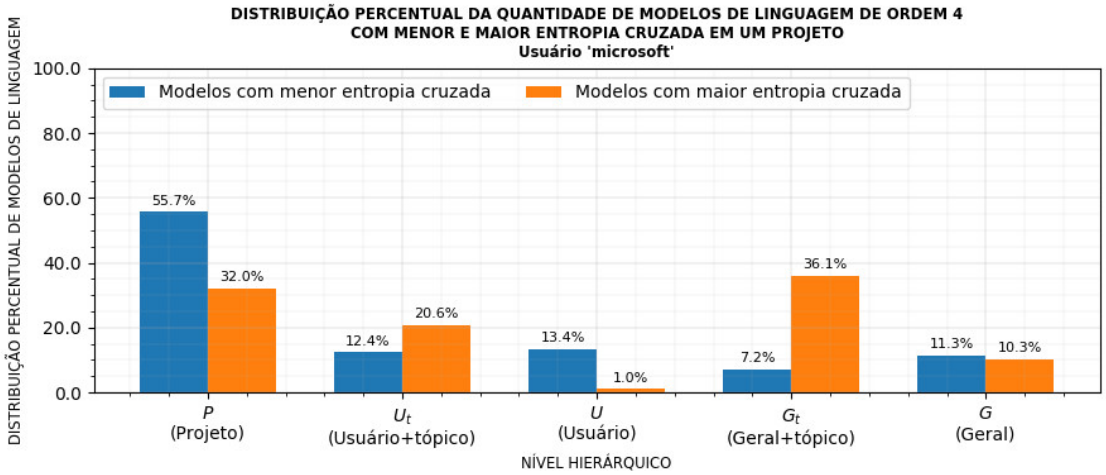
A Figura 73 representa o mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para o usuário `microsoft`, e a Figura 74 ilustra a distribuição percentual da quantidade de modelos com menores e maiores medidas.

Figura 73 – Mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para modelos de linguagem de ordem 4 – usuário ‘microsoft’.



Fonte: Autor.

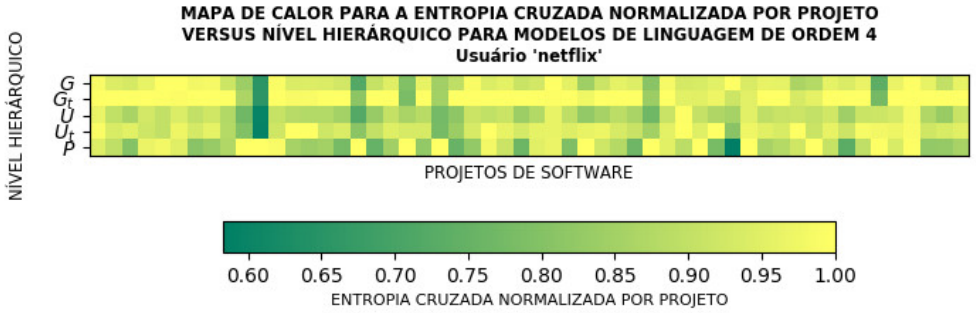
Figura 74 – Distribuição percentual da quantidade de modelos de linguagem de ordem 4 com menor e maior entropia cruzada em um projeto– usuário ‘microsoft’.



Fonte: Autor.

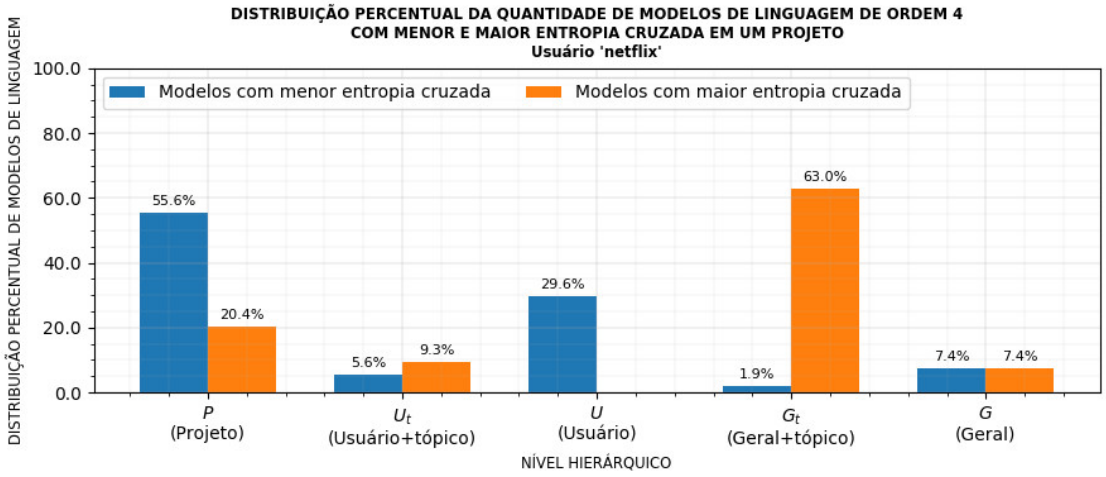
A Figura 75 representa o mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para o usuário `netflix`, e a Figura 76 ilustra a distribuição percentual da quantidade de modelos com menores e maiores medidas.

Figura 75 – Mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para modelos de linguagem de ordem 4 – usuário ‘netflix’.



Fonte: Autor.

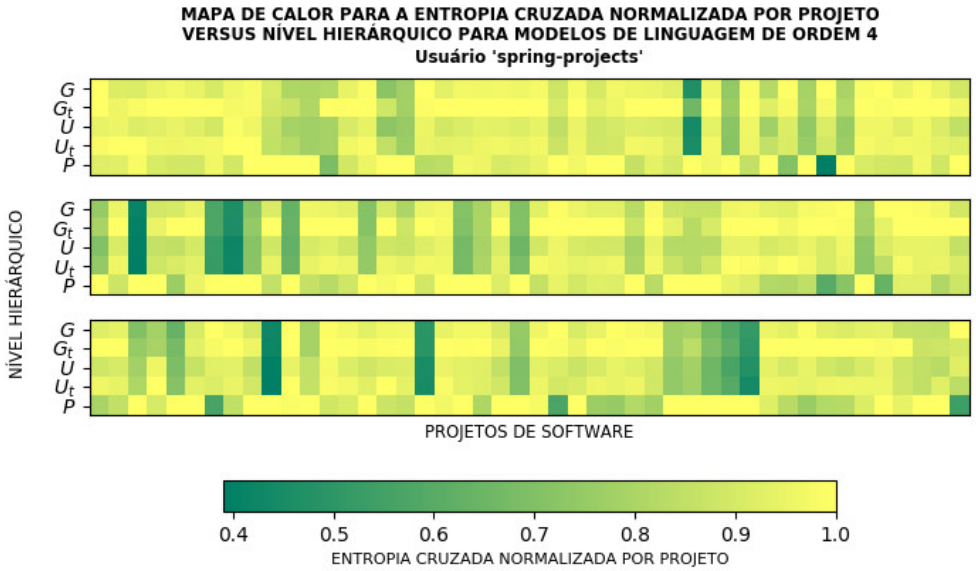
Figura 76 – Distribuição percentual da quantidade de modelos de linguagem de ordem 4 com menor e maior entropia cruzada em um projeto– usuário ‘netflix’.



Fonte: Autor.

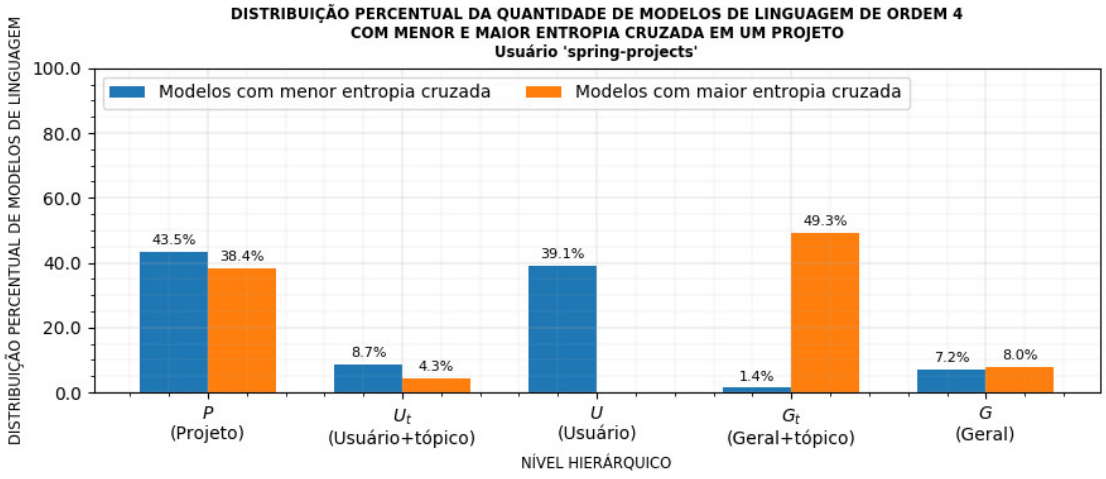
A Figura 77 representa o mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para o usuário `spring-projects`, e a Figura 78 ilustra a distribuição percentual da quantidade de modelos com menores e maiores medidas.

Figura 77 – Mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para modelos de linguagem de ordem 4 – usuário ‘spring-projects’.



Fonte: Autor.

Figura 78 – Distribuição percentual da quantidade de modelos de linguagem de ordem 4 com menor e maior entropia cruzada em um projeto– usuário ‘spring-projects’.

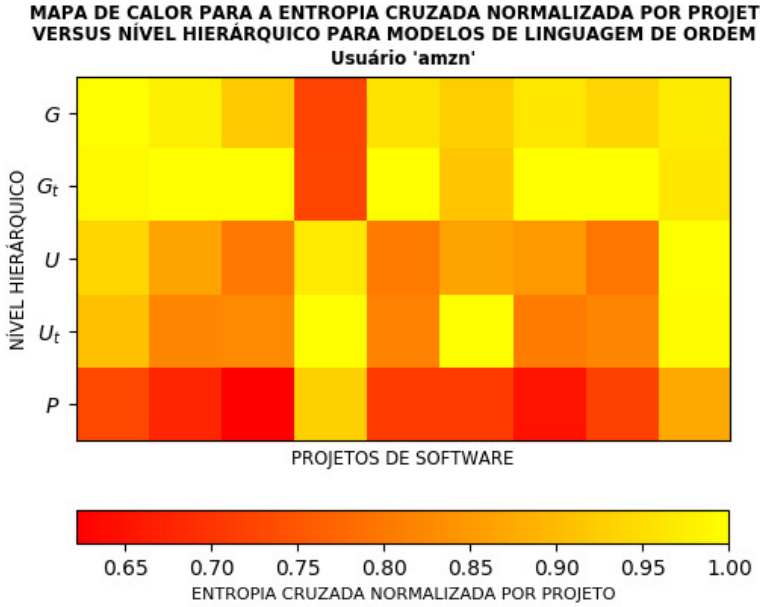


Fonte: Autor.

APÊNDICE D – MEDIDAS DE ENTROPIA CRUZADA EM CADA UM DOS NÍVEIS HIERÁRQUICOS PARA MODELOS DE LINGUAGEM DE ORDEM 3 COM ALGORITMO DE SUAVIZAÇÃO BASEADO EM INTERPOLAÇÃO

A Figura 79 representa o mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para o usuário `amzn`.

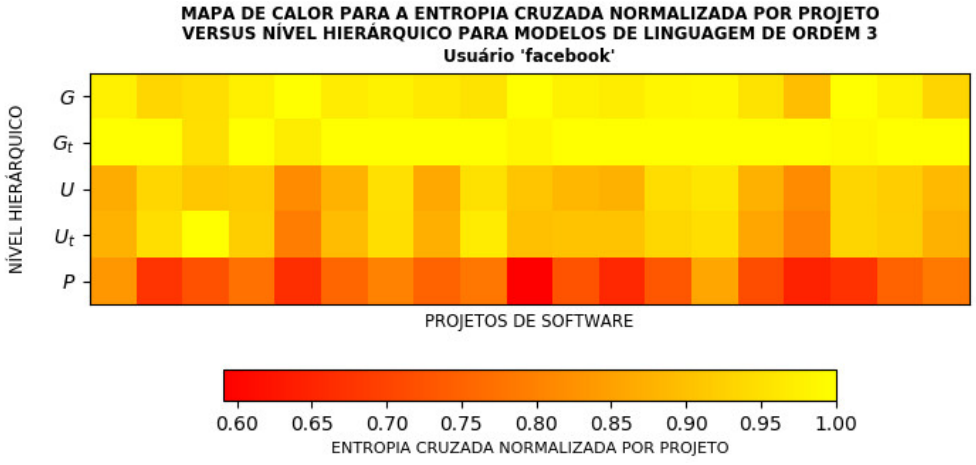
Figura 79 – Mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para modelos de linguagem de ordem 3 e algoritmo de suavização baseado em interpolação dependente de contexto – usuário ‘amzn’.



Fonte: Autor.

A Figura 80 representa o mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para o usuário `facebook`.

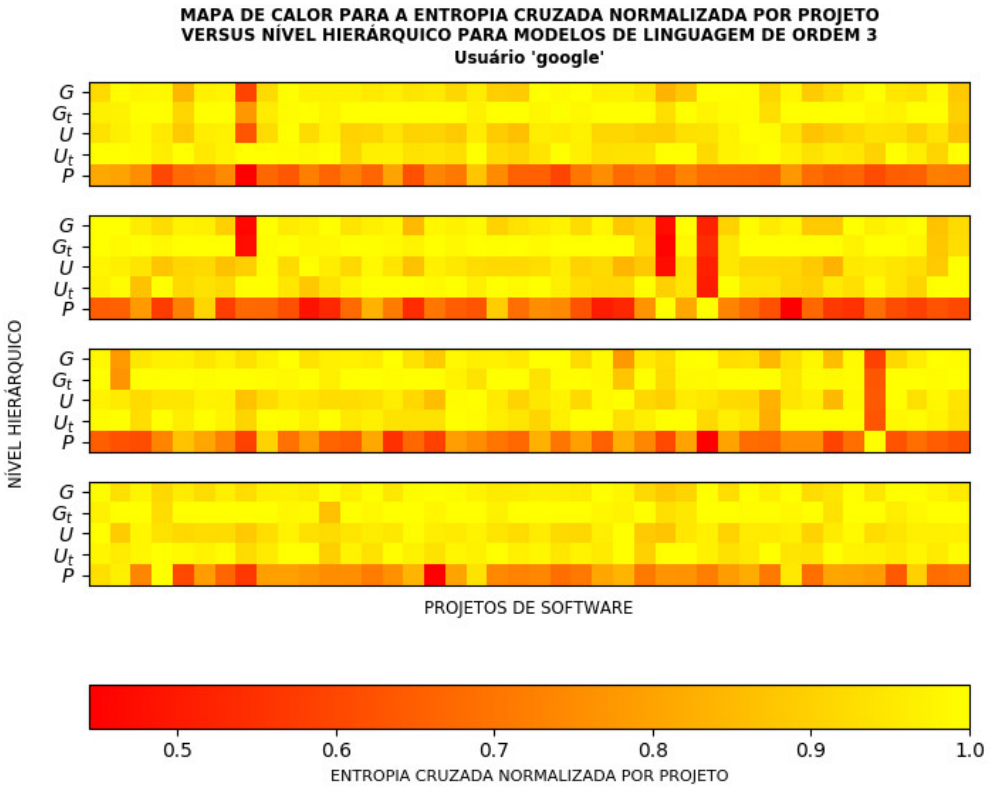
Figura 80 – Mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para modelos de linguagem de ordem 3 e algoritmo de suavização baseado em interpolação dependente de contexto – usuário ‘facebook’.



Fonte: Autor.

A Figura 81 representa o mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para o usuário `google`.

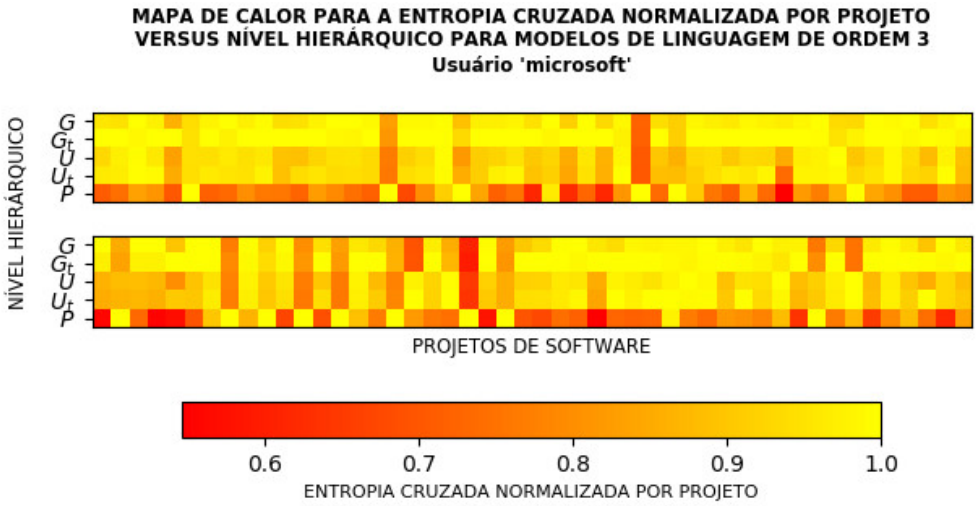
Figura 81 – Mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para modelos de linguagem de ordem 3 e algoritmo de suavização baseado em interpolação dependente de contexto – usuário ‘google’.



Fonte: Autor.

A Figura 82 representa o mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para o usuário `microsoft`.

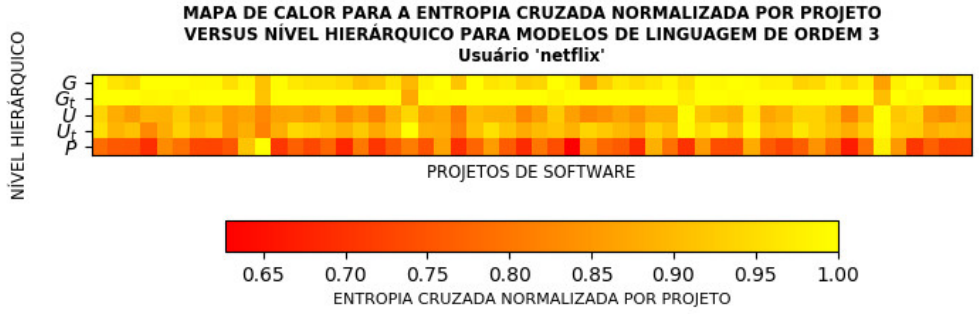
Figura 82 – Mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para modelos de linguagem de ordem 3 e algoritmo de suavização baseado em interpolação dependente de contexto – usuário ‘microsoft’.



Fonte: Autor.

A Figura 83 representa o mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para o usuário `netflix`.

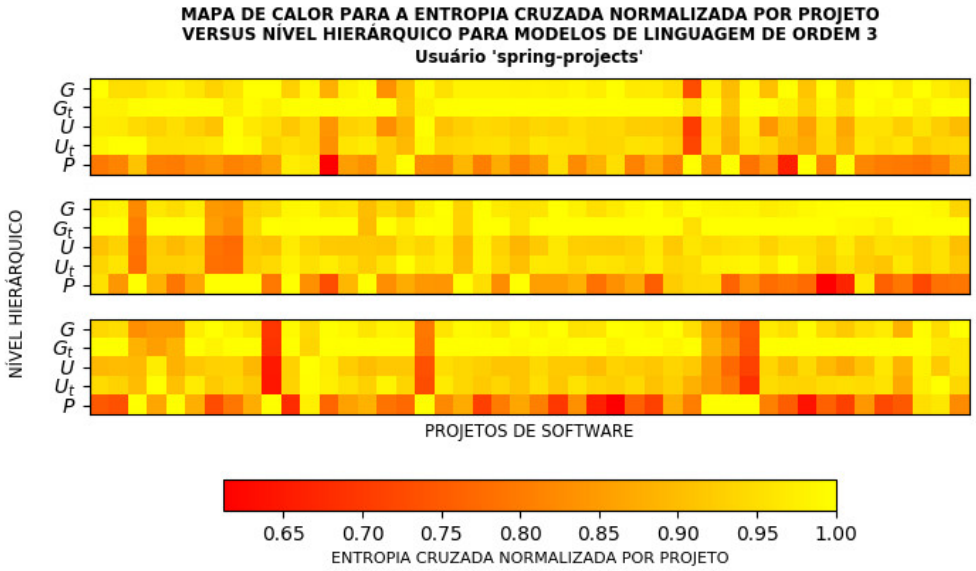
Figura 83 – Mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para modelos de linguagem de ordem 3 e algoritmo de suavização baseado em interpolação dependente de contexto – usuário ‘netflix’.



Fonte: Autor.

A Figura 84 representa o mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para o usuário `spring-projects`.

Figura 84 – Mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para modelos de linguagem de ordem 3 e algoritmo de suavização baseado em interpolação dependente de contexto – usuário ‘spring-projects’.



Fonte: Autor.

APÊNDICE E – AVALIAÇÕES COM A MEDIDA MRR

As análises extrínsecas realizadas nesse apêndice utilizam a métrica *Mean Reciprocal Rank* (MRR). Essa métrica, utilizada em trabalhos nos trabalhos de Tu, Su e Devanbu (2014) e Hellendoorn e Devanbu (2017), foi originalmente definida para sistemas de perguntas e respostas *web*, é definida a partir da Equação 17. De acordo com essas definições, a medida de um conjunto de consultas Q é dada pela posição em que a resposta esperada se encontra em uma lista de resultados[37].

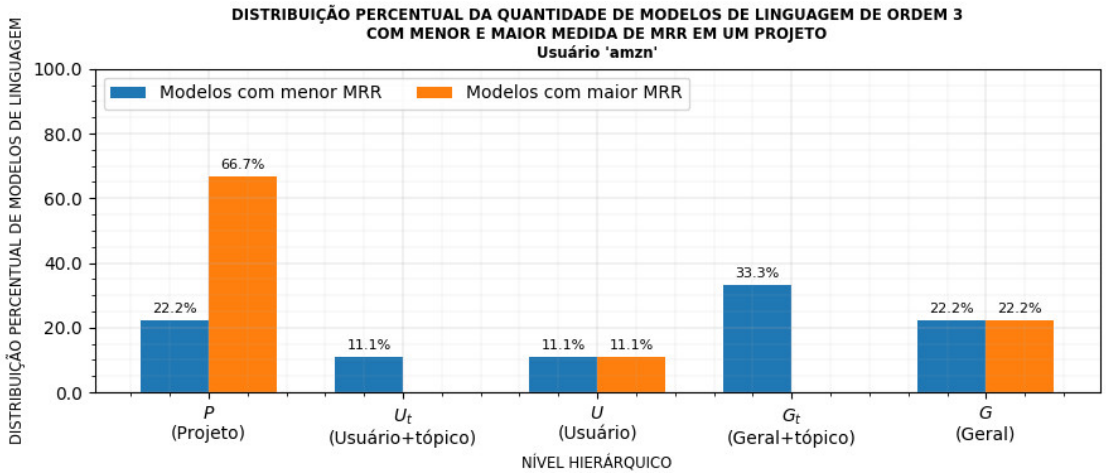
$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i} \tag{17}$$

Para realizar as avaliações, e diferentemente da seção 4.4.1 que analisa os projetos de software em sua totalidade, esse apêndice considera um conjunto de 110.511 consultas artificiais extraídas aleatoriamente do conjunto de dados de validação, mesmo embora Proksch et al. (2016) observem que a aplicabilidade desse tipo de consulta é limitada já que elas nem sempre estão relacionadas com os padrões de uso dos usuários finais. Esse limite prático, que considera 3 consultas para cada um dos arquivos de código-fonte analisados, é necessário pois avaliações extrínsecas utilizam grandes quantidades de recursos computacionais, e podem durar longos períodos de tempo para serem concluídas.

[37] Medidas MRR mais altas são indicações de um modelo de linguagem com melhor capacidade preditiva.

A Figura 85 ilustra a distribuição percentual da quantidade de modelos com menor e maior medida MRR em um projeto considerando o usuário `amzn`. Observa-se que, dentre todos os projetos avaliados, modelos de linguagem mais específicos concentram grande parte (66,7%) dos valores mais altos de MRR.

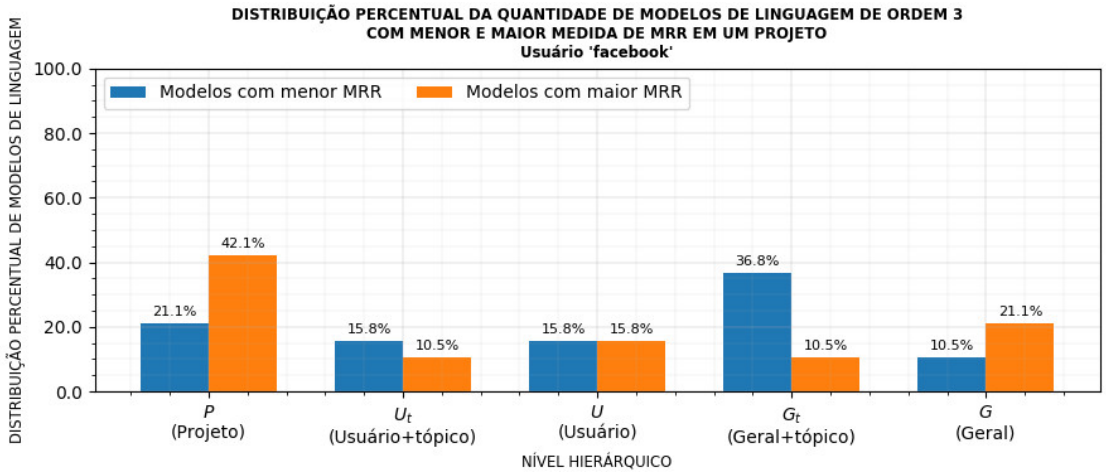
Figura 85 – Distribuição percentual da quantidade de modelos de linguagem com menor e maior medida MRR em um projeto – usuário ‘amzn’.



Fonte: Autor.

A Figura 86 ilustra a distribuição percentual da quantidade de modelos com menor e maior medida MRR em um projeto considerando o usuário `facebook`. Nota-se uma distribuição que concentra 42,1% dos valores mais altos de MRR em modelos de linguagem n-grama construídos a partir do projeto de software.

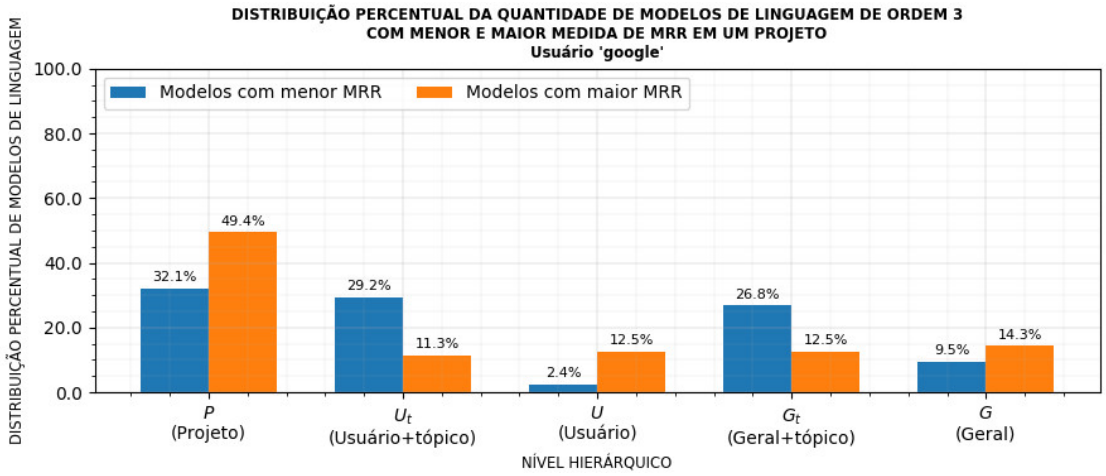
Figura 86 – Distribuição percentual da quantidade de modelos de linguagem com menor e maior medida MRR em um projeto – usuário ‘facebook’.



Fonte: Autor.

A Figura 87 ilustra a distribuição percentual da quantidade de modelos com menor e maior medida MRR em um projeto considerando o usuário `google`. Observa-se que 49,4% dos modelos de linguagem de projeto apresentaram valores mais altos de MRR.

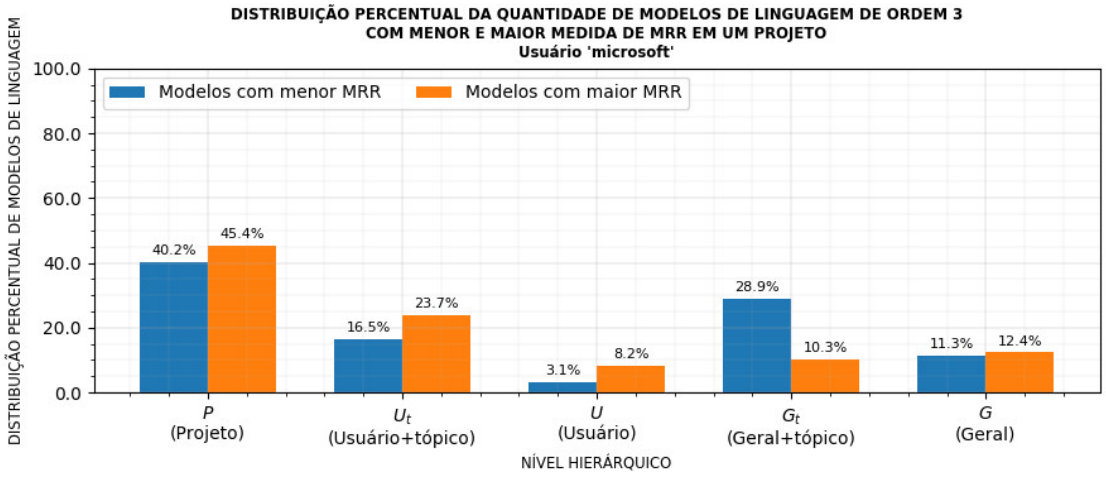
Figura 87 – Distribuição percentual da quantidade de modelos de linguagem com menor e maior medida MRR em um projeto – usuário ‘google’.



Fonte: Autor.

A Figura 88 ilustra a distribuição percentual da quantidade de modelos com menor e maior medida MRR em um projeto considerando o usuário `microsoft`. Nota-se que os valores mais altos de MRR se concentram em modelos de linguagem por projeto (45,4%) e intra-usuário (23,7%).

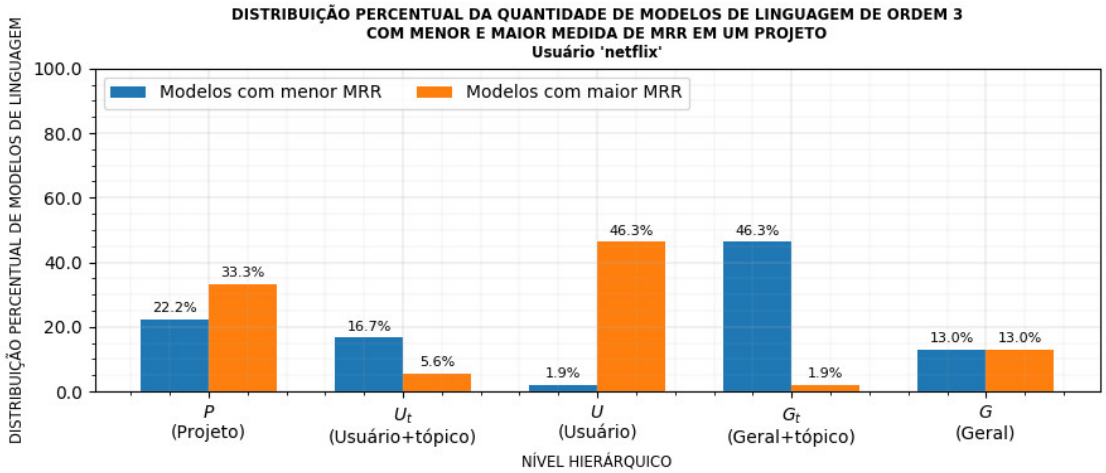
Figura 88 – Distribuição percentual da quantidade de modelos de linguagem com menor e maior medida MRR em um projeto – usuário ‘microsoft’.



Fonte: Autor.

A Figura 89 ilustra a distribuição percentual da quantidade de modelos com menor e maior medida MRR em um projeto considerando o usuário `netflix`. Observa-se que os valores mais altos de MRR se concentram no modelos de linguagem de projeto (33,3%) e de usuário (46,3%).

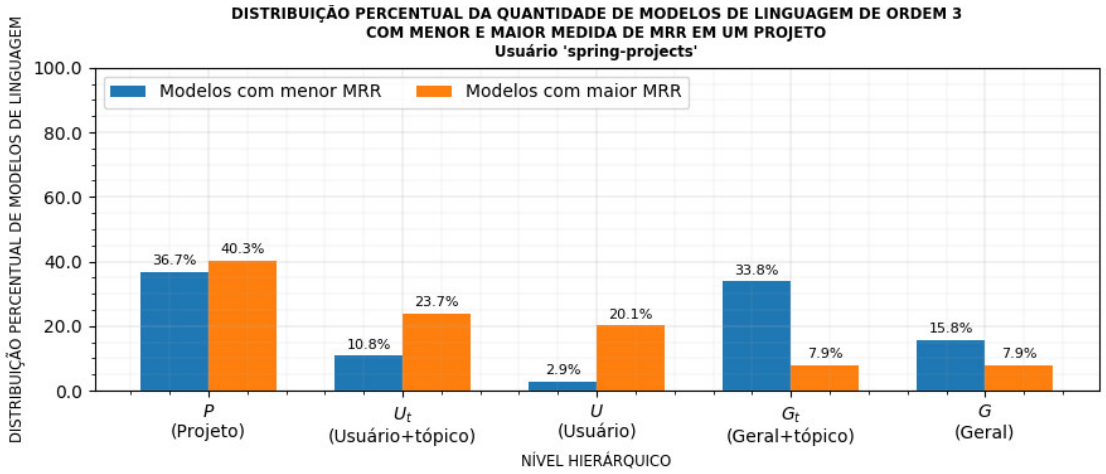
Figura 89 – Distribuição percentual da quantidade de modelos de linguagem com menor e maior medida MRR em um projeto – usuário ‘netflix’.



Fonte: Autor.

A Figura 90 ilustra a distribuição percentual da quantidade de modelos com menor e maior medida MRR em um projeto considerando o usuário `spring-projects`. Observa-se que grande parte dos valores mais altos de MRR se concentram nos modelos de projeto (40,3%), nos modelos intra-usuário (23,7%) e de usuário (20,1%).

Figura 90 – Distribuição percentual da quantidade de modelos de linguagem de ordem 3 com menor e maior medida MRR em um projeto – usuário ‘spring-projects’.



Fonte: Autor.

APÊNDICE F – AVALIAÇÕES E RESULTADOS SOBRE O MODELO PROPOSTO COM O CORPO DE DADOS CD2

Neste apêndice são registradas as avaliações e resultados referentes ao modelo proposto por este trabalho considerando o corpo de dados `CD2` definido na seção 4.2.

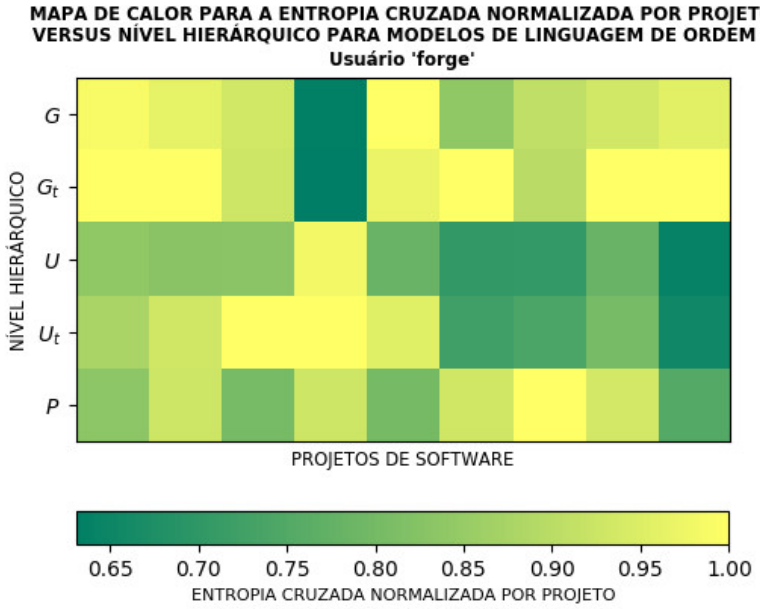
O formato de apresentação dos resultados segue a mesma estrutura encontrada na seção 4.3.3. Entretanto, a quantidade de usuários mantenedores encontrada no corpo de dados `CD2` é significativa, o que impede a exibição exaustiva dos resultados . Assim, opta-se por exibir os resultados de 4 usuários escolhidos aleatoriamente a partir de conjuntos organizados de acordo a representatividade perante a quantidade total de arquivos disponíveis no corpo de dados, encontrada na coluna (5) da Tabela 10 do Apêndice B.

No primeiro conjunto, que considera usuários com até 1% da quantidade total de arquivos, `forge` foi o usuário foi selecionado. No segundo conjunto, para representatividade entre 1% e 5%, `allwinnerwk` foi o usuário foi selecionado. No terceiro conjunto, para representatividade entre 5% e 10%, `jetbrains` foi o usuário foi selecionado. Por fim, para representatividade superior a 10% o usuário `apache` foi selecionado.

ENTROPIA CRUZADA EM CADA UM DOS NÍVEIS HIERÁRQUICOS

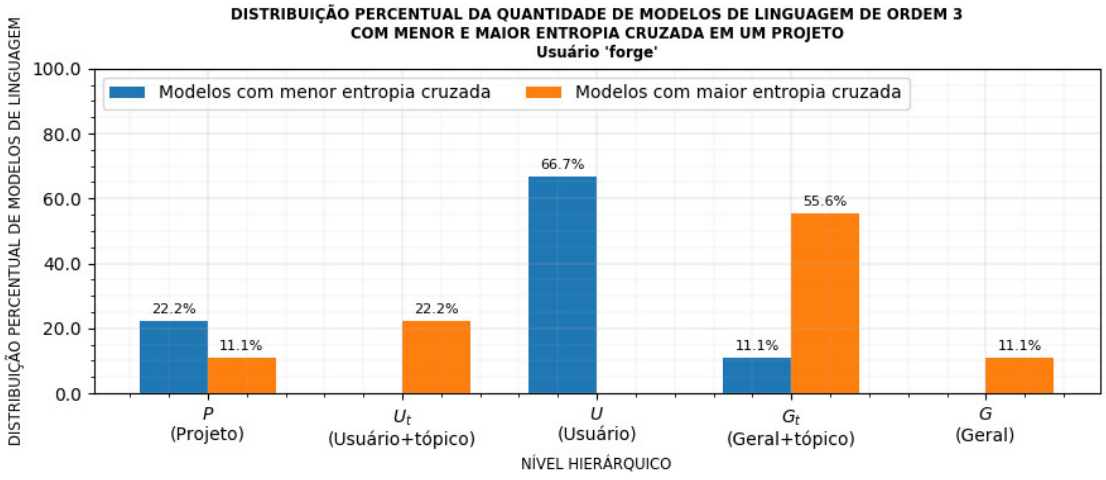
A Figura 91 representa o mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para o usuário `forge` e a Figura 92 evidencia a distribuição percentual da quantidade de modelos com menores e maiores medidas.

Figura 91 – Mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para modelos de linguagem de ordem 3 – usuário ‘forge’.



Fonte: Autor.

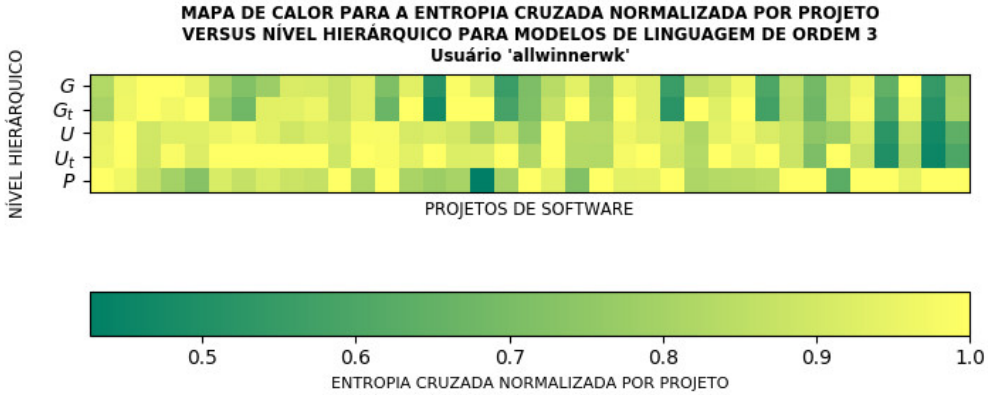
Figura 92 – Distribuição percentual da quantidade de modelos de linguagem de ordem 3 com menor e maior entropia cruzada em um projeto– usuário ‘forje’.



Fonte: Autor.

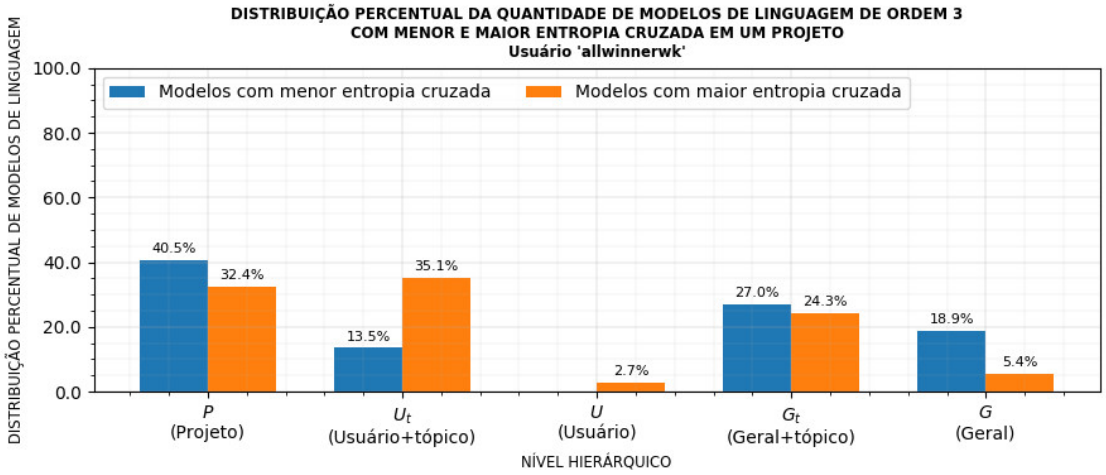
A Figura 93 representa o mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para o usuário `allwinnerwk` e a Figura 94 evidencia a distribuição percentual da quantidade de modelos com menores e maiores medidas.

Figura 93 – Mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para modelos de linguagem de ordem 3 – usuário ‘allwinnerwk’.



Fonte: Autor.

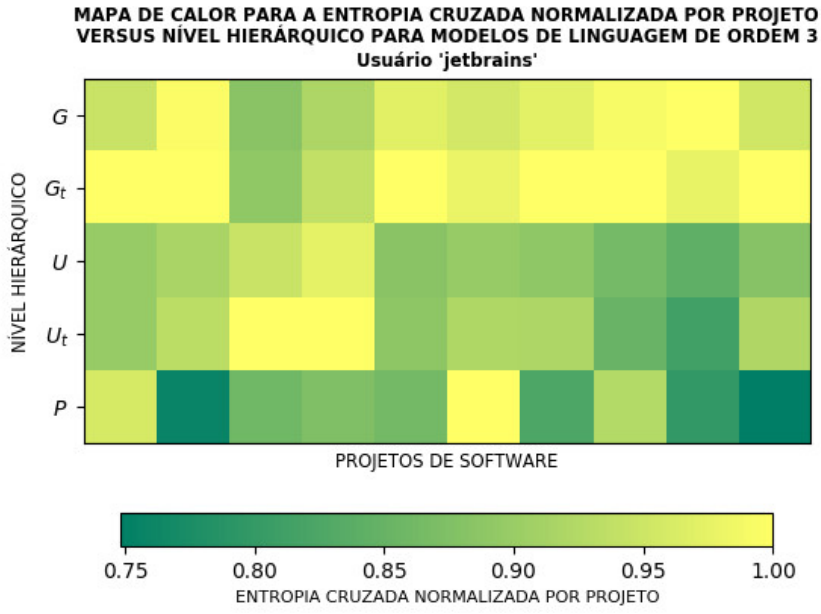
Figura 94 – Distribuição percentual da quantidade de modelos de linguagem de ordem 3 com menor e maior entropia cruzada em um projeto– usuário ‘allwinnerwk’.



Fonte: Autor.

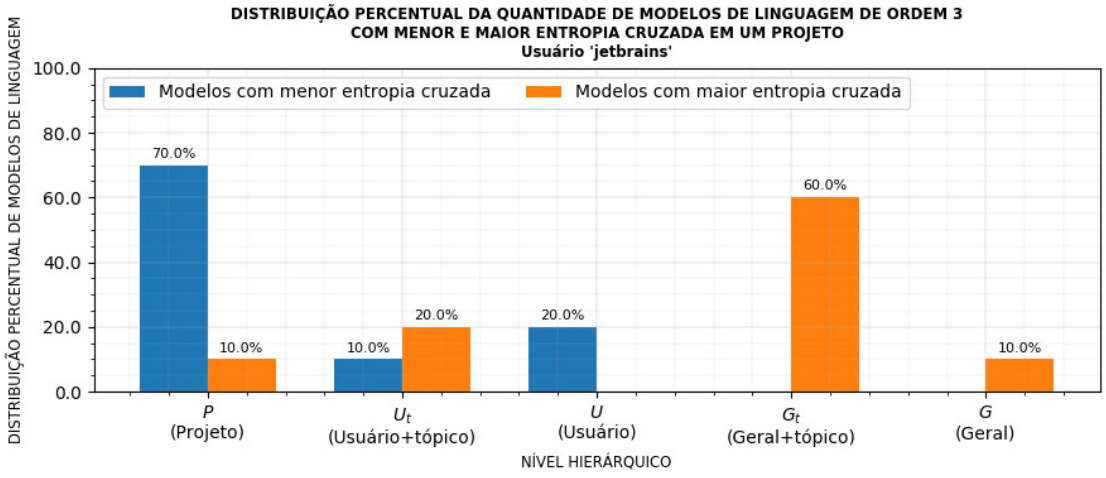
A Figura 95 representa o mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para o usuário `jetbrains` e a Figura 96 evidencia a distribuição percentual da quantidade de modelos com menores e maiores medidas.

Figura 95 – Mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para modelos de linguagem de ordem 3 – usuário ‘jetbrains’.



Fonte: Autor.

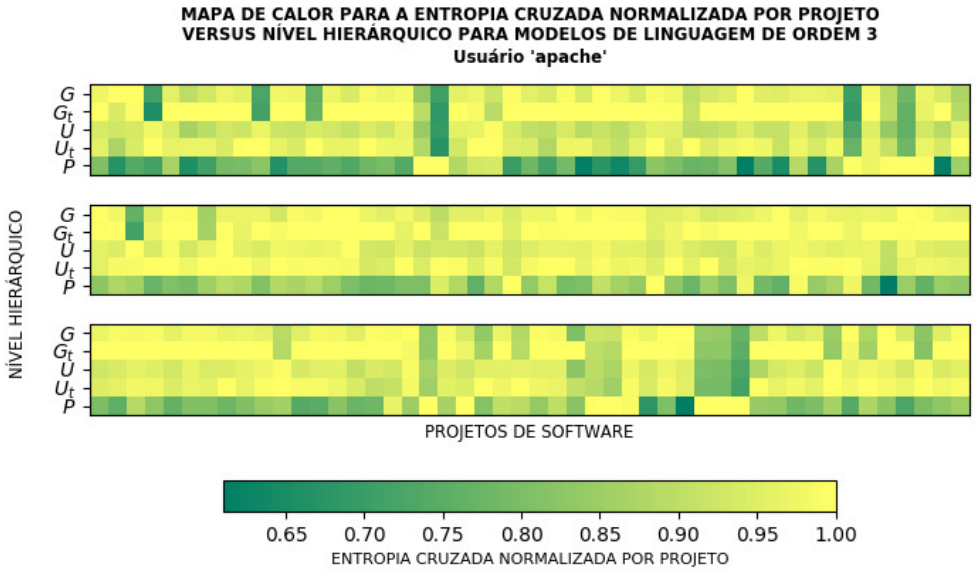
Figura 96 – Distribuição percentual da quantidade de modelos de linguagem de ordem 3 com menor e maior entropia cruzada em um projeto– usuário ‘jetbrains’.



Fonte: Autor.

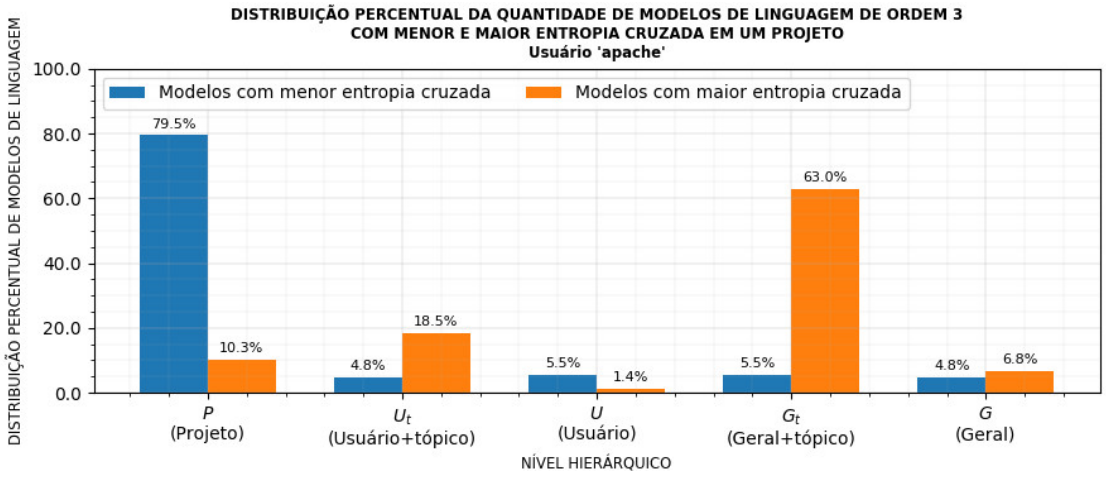
A Figura 97 representa o mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para o usuário `apache` e a Figura 98 evidencia a distribuição percentual da quantidade de modelos com menores e maiores medidas

Figura 97 – Mapa de calor para entropia cruzada normalizada por projeto versus nível hierárquico para modelos de linguagem de ordem 3 – usuário ‘apache’.



Fonte: Autor.

Figura 98 – Distribuição percentual da quantidade de modelos de linguagem de ordem 3 com menor e maior entropia cruzada em um projeto– usuário ‘apache’.

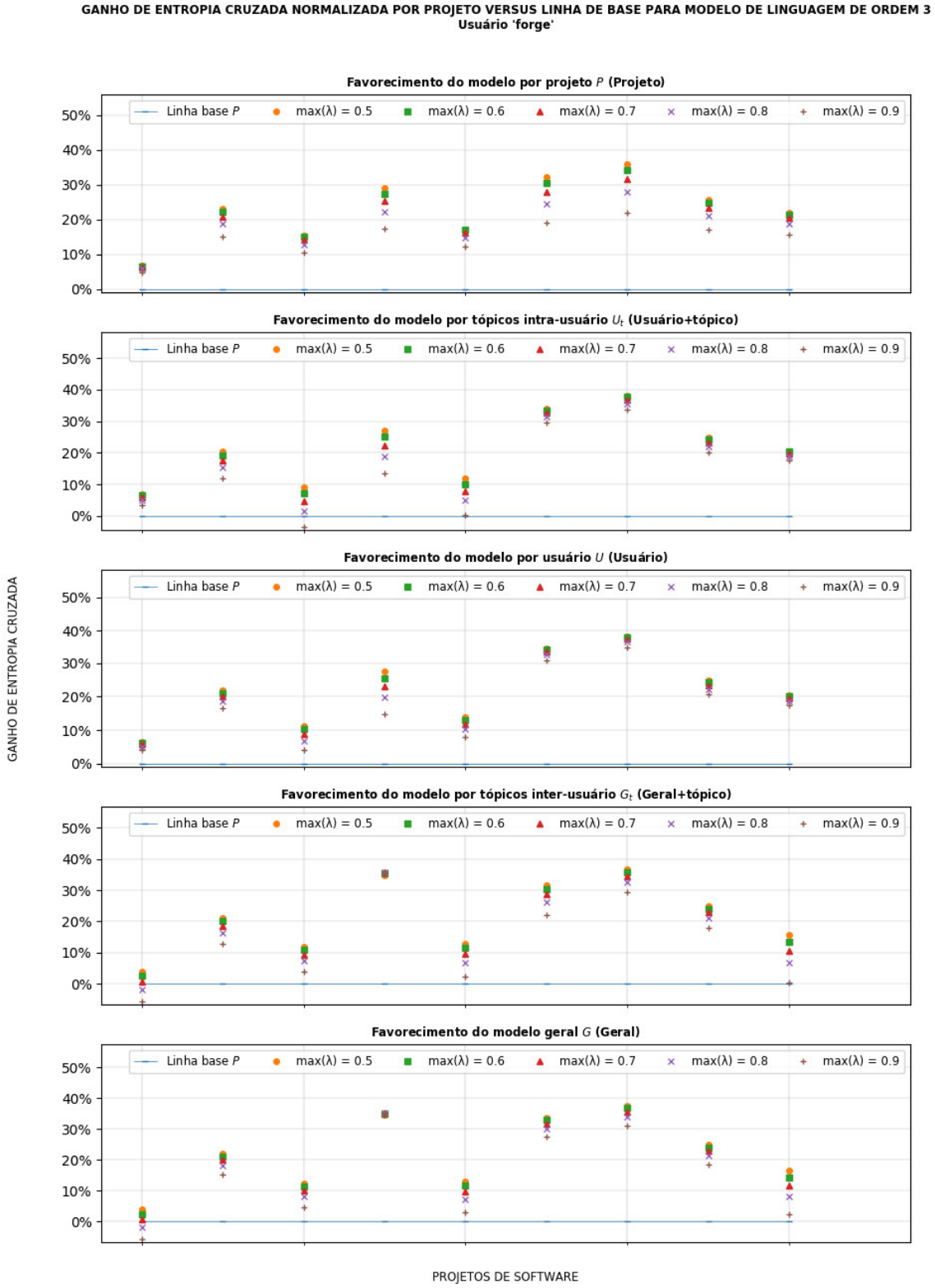


Fonte: Autor.

COMBINANDO OS MODELOS DE LINGUAGEM

A Figura 99 ilustra os ganhos de entropia cruzada com diferentes configurações de favorecimento de nível hierárquico para o usuário `forge`.

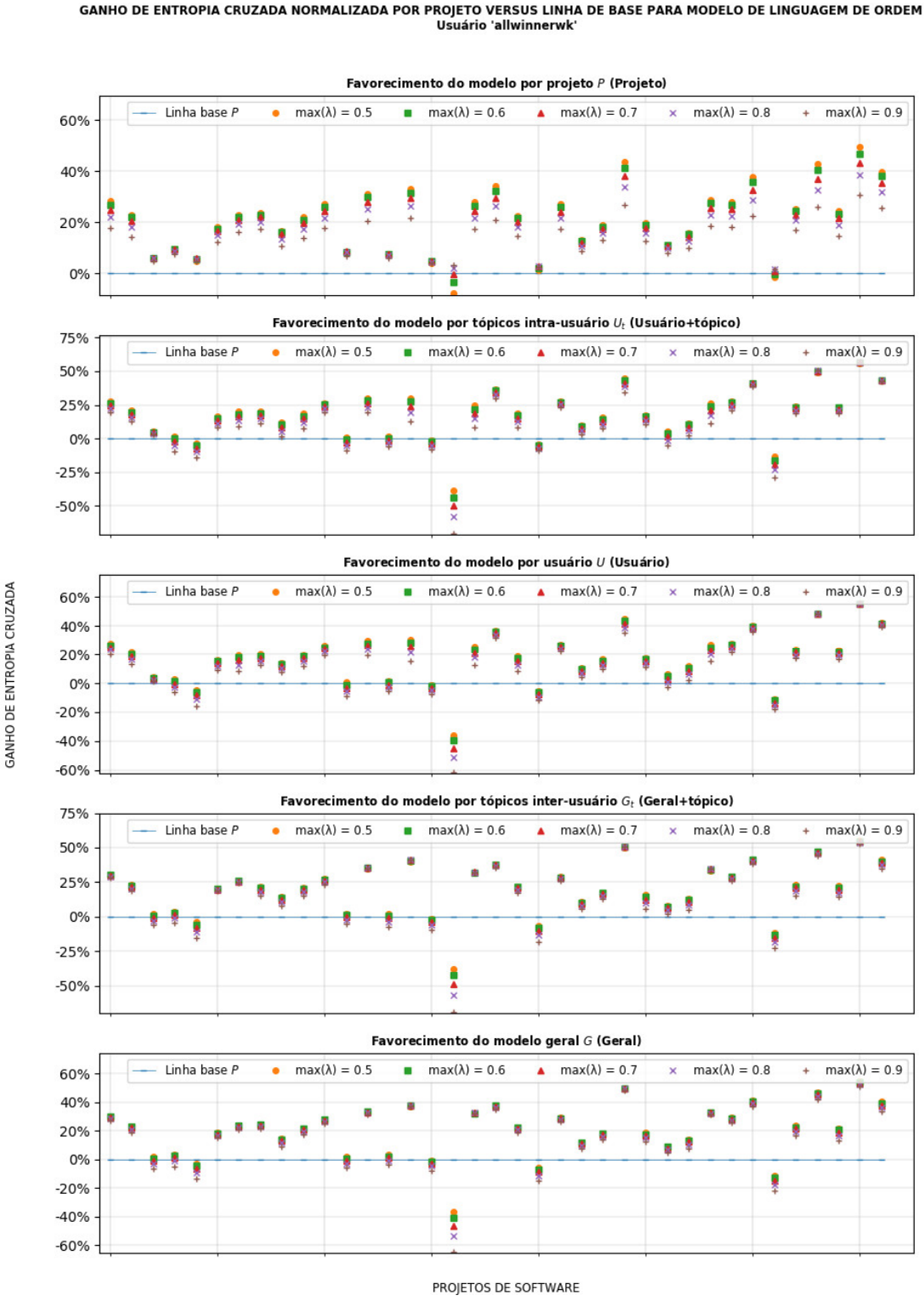
Figura 99 – Ganho de entropia cruzada normalizada por projeto com favorecimento dos níveis hierárquicos versus linha de base – usuário ‘forge’.



Fonte: Autor.

A Figura 100 ilustra os ganhos de entropia cruzada com diferentes configurações de favorecimento de nível hierárquico para o usuário `allwinnerwk`.

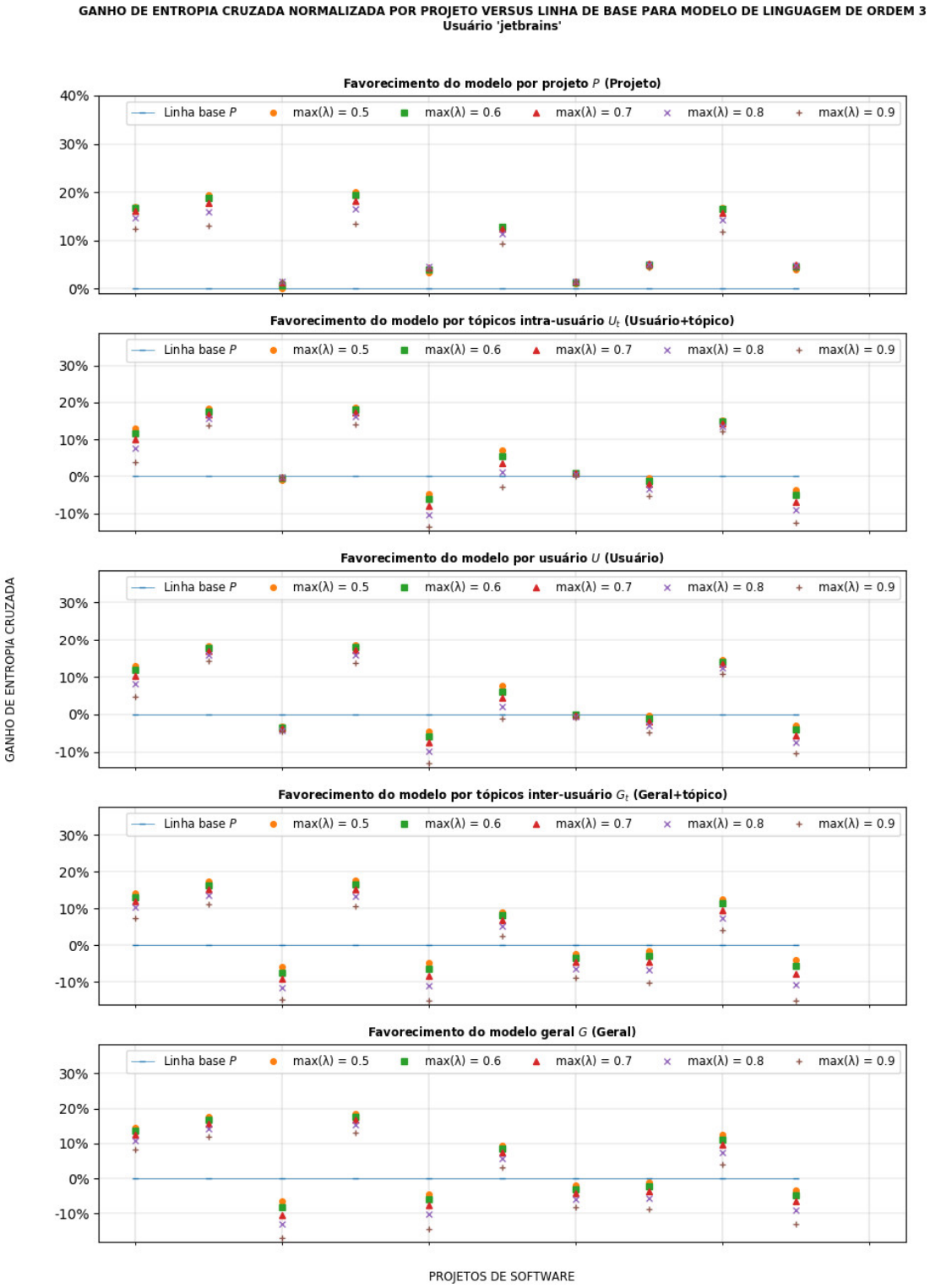
Figura 100 – Ganho de entropia cruzada normalizada por projeto com favorecimento dos níveis hierárquicos versus linha de base – usuário ‘allwinnerwk’.



Fonte: Autor.

A Figura 101 ilustra os ganhos de entropia cruzada com diferentes configurações de favorecimento de nível hierárquico para o usuário `jetbrains`.

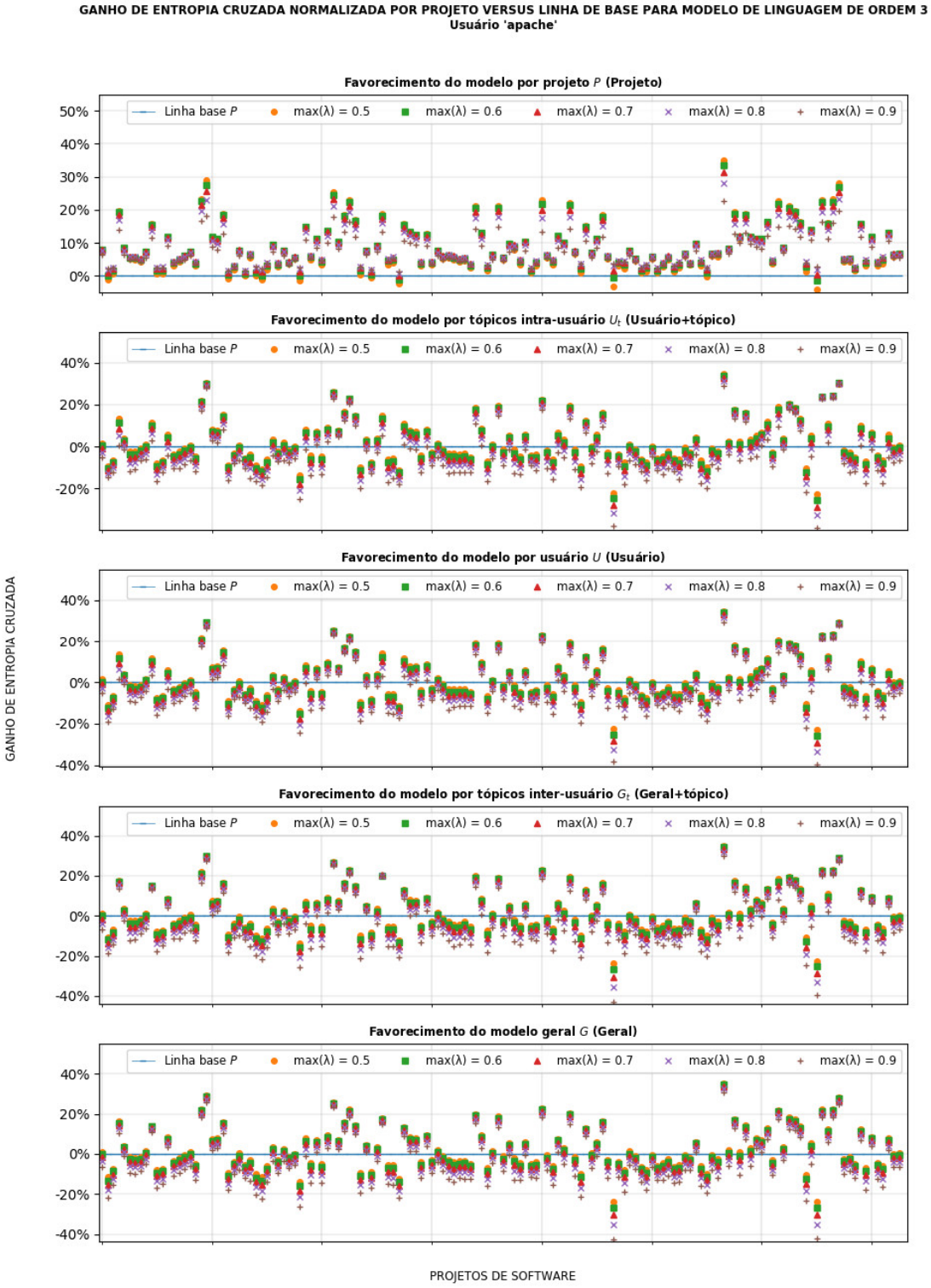
Figura 101 – Ganho de entropia cruzada normalizada por projeto com favorecimento dos níveis hierárquicos versus linha de base – usuário ‘jetbrains’.



Fonte: Autor.

A Figura 102 ilustra os ganhos de entropia cruzada com diferentes configurações de favorecimento de nível hierárquico para o usuário `apache`.

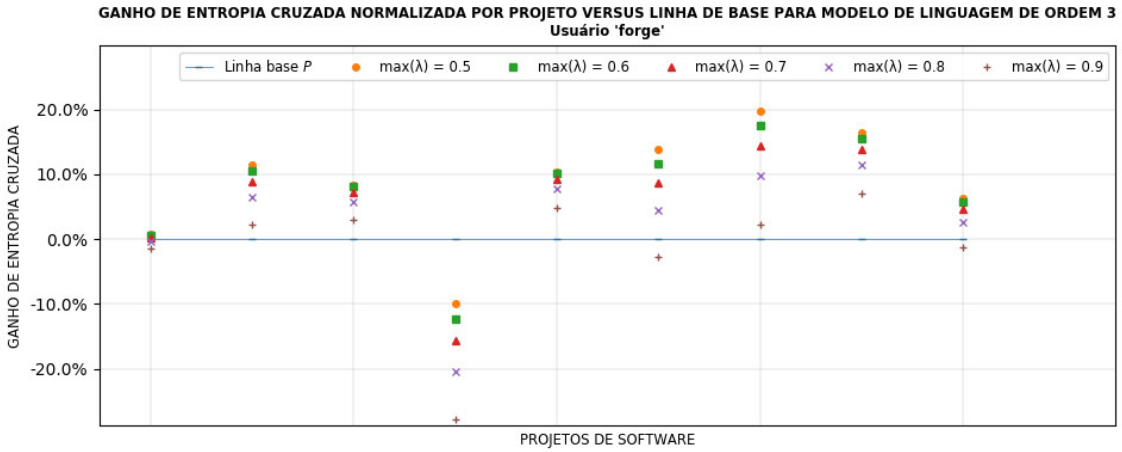
Figura 102 – Ganho de entropia cruzada normalizada por projeto com favorecimento dos níveis hierárquicos versus linha de base – usuário ‘apache’.



Fonte: Autor.

A Figura 103 ilustra o ganho na medida de entropia cruzada normalizada por projeto para o usuário `forge`.

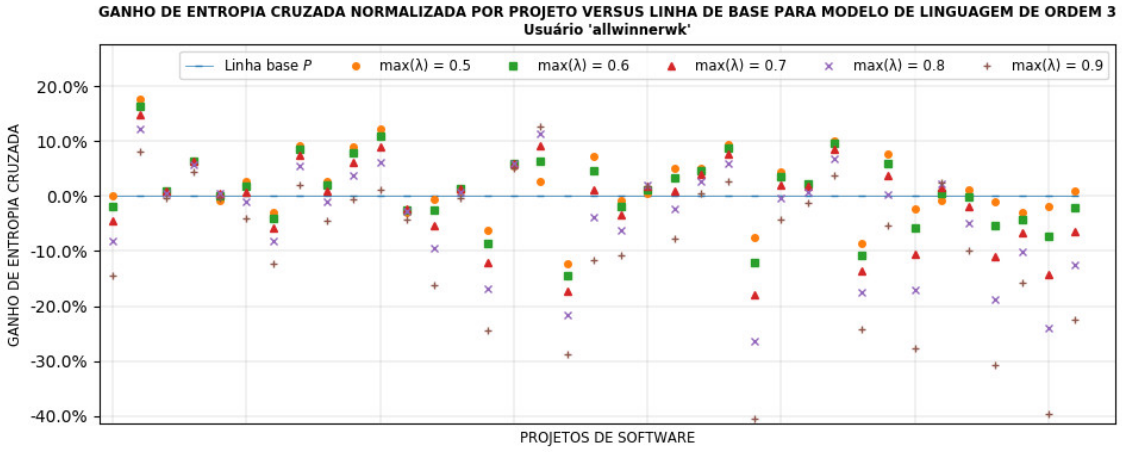
Figura 103 – Ganho de entropia cruzada normalizada versus linha de base definida a partir de um modelo com cache – usuário ‘forge’.



Fonte: Autor.

A Figura 104 ilustra o ganho na medida de entropia cruzada normalizada por projeto para o usuário `allwinnerwk`.

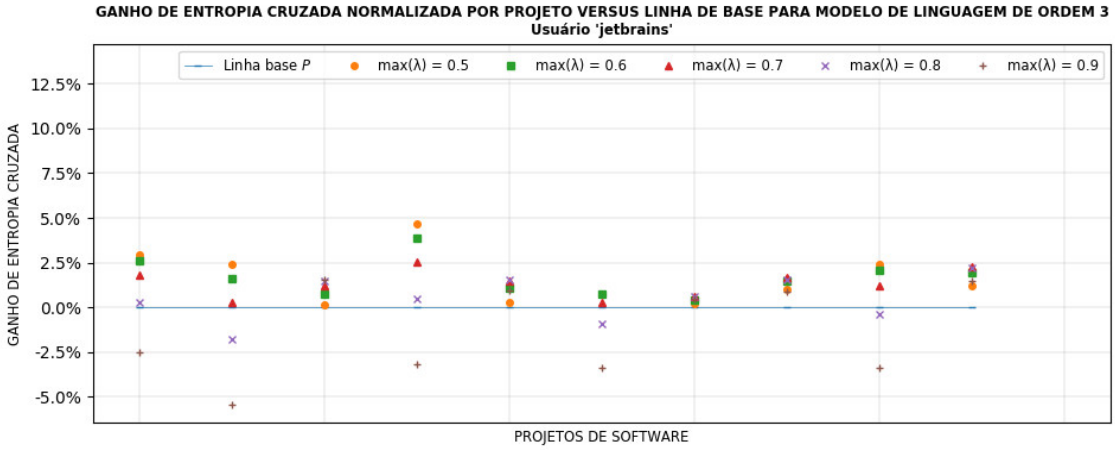
Figura 104 – Ganho de entropia cruzada normalizada versus linha de base definida a partir de um modelo com cache – usuário ‘allwinnerwk’.



Fonte: Autor.

A Figura 105 ilustra as diferenças na medida de entropia cruzada normalizada por projeto para o usuário `jetbrains`.

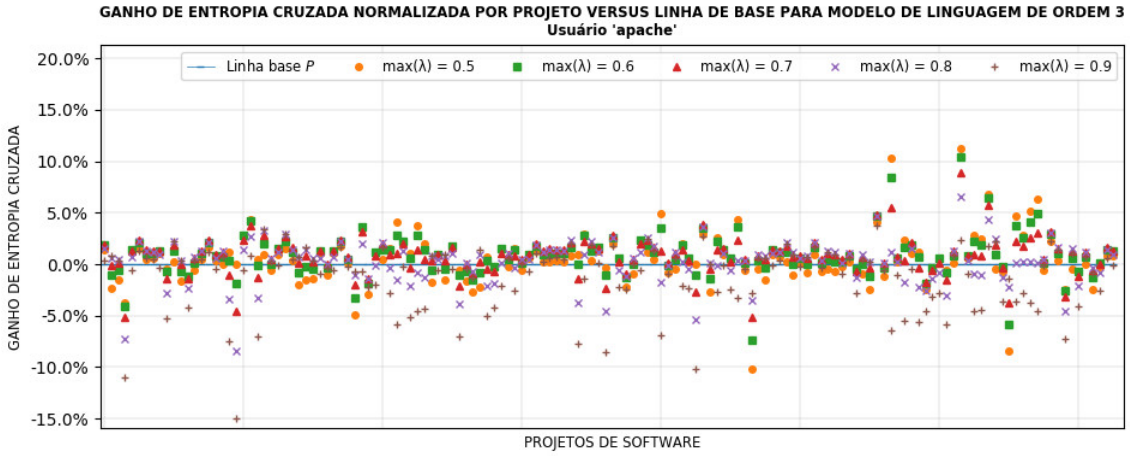
Figura 105 – Ganho de entropia cruzada normalizada versus linha de base definida a partir de um modelo com cache – usuário ‘jetbrains’.



Fonte: Autor.

A Figura 106 ilustra as diferenças na medida de entropia cruzada normalizada por projeto para o usuário `apache`.

Figura 106 – Ganho de entropia cruzada normalizada versus linha de base definida a partir de um modelo com cache – usuário ‘apache’.



Fonte: Autor.