UNIVERSIDADE FEDERAL DE ALAGOAS
INSTITUTO DE COMPUTAÇÃO
PROGRAMA DE PÓS GRADUAÇÃO EM INFORMÁTICA

DIOGO CABRAL DE ALMEIDA

Pride: Uma ferramenta para detecção de similaridade em código-fonte

Maceió
2015

Diogo Cabral de Almeida

Pride: Uma ferramenta para detecção de similaridade em código-fonte

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

Orientador: Prof. Dr. Rodrigo de Barros Paes

Coorientador: Prof. Dr. Márcio de Medeiros Ribeiro

Maceió
2015

Catalogação na fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico
Bibliotecário: Valter dos Santos Andrade

A447p Almeida, Diogo Cabral de.
PRIDE: uma ferramenta para detecção de similaridade em código-fonte / Diogo Cabral de Almeida. – 2015.
79 f. : il.

Orientador: Rodrigo de Barros Paes.
Coorientador: Márcio de Medeiros Ribeiro.
Dissertação (Mestrado em Informática) – Universidade Federal de Alagoas. Instituto de Computação. Programa de Pós-Graduação em Informática. Maceió, 2015.

Bibliografia: f. 76-79.

1. Código-fonte. 2. Linguagem de programação (Computadores). 4. PRIDE (Linguagem de programação de computador). I. Título.

CDU: 004.43

UNIVERSIDADE FEDERAL DE ALAGOAS/UFAL
Programa de Pós-Graduação em Informática – PpgI
Instituto de Computação
Campus A. C. Simões BR 104-Norte Km 14 BL 12 Tabuleiro do Martins
Maceió/AL - Brasil CEP: 57.072-970 | Telefone: (082) 3214-1401

Membros da Comissão Julgadora da Dissertação de Mestrado de Diogo Cabral de Almeida, intitulada: "*PRIDE: Uma Ferramenta de Detecção de Similaridade em Código Fonte*", apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal de Alagoas em 31 de março de 2015, às 15h00min, no Miniauditório do Instituto de Computação da UFAL.

COMISSÃO JULGADORA

Prof. Dr. Rodrigo de Barros Paes
UFAL – Instituto de Computação
Orientador

Prof. Dr. Márcio de Medeiros Ribeiro
UFAL – Instituto de Computação
Orientador

Prof. Dr. Leandro Dias da Silva
UFAL – Instituto de Computação
Examinador

Prof. Dr. Leopoldo Motta Teixeira
UFPE – Universidade Federal de Pernambuco
Examinador

RESUMO

O plágio entre alunos de disciplinas introdutórias de programação vem aumentando ao longo do tempo. A facilidade na troca de informações trazida pela Internet pode ser um dos fatores responsáveis por esse aumento. Em muitos casos, os alunos tentam disfarçar o plágio fazendo algumas modificações no código-fonte. Porém, algumas técnicas de disfarce são extremamente complexas e podem não ser detectadas a olho nu.

Neste trabalho, foram analisadas as técnicas de detecção e, com base nelas, foi desenvolvido um sistema capaz de detectar plágio em código-fonte. Este sistema é baseado na representação do código como uma árvore sintática abstrata e no algoritmo *Karp-Rabin Greedy String Tiling*.

O sistema foi avaliado utilizando uma base de códigos-fonte de alunos de disciplinas programação. Foi realizada uma comparação baseada em oráculo para comparar o sistema com os demais. O oráculo foi criado a partir da análise do docente da disciplina, onde foi marcado se havia plágio ou não em cada par de código-fonte. Para representar os resultados, foram utilizadas curvas ROC e matrizes de confusão. O mesmo procedimento foi aplicado aos sistemas já existentes, o que permitiu a comparação direta entre os resultados. Mais especificamente, utilizamos o valor da área sob a curva e a distância mínima para o ponto (0, 1) do espaço ROC, uma vez que esses valores representam o desempenho de classificação.

A análise dos resultados indica que, para a amostra utilizada, o sistema desenvolvido obteve o maior valor da área sob a curva e também a menor distância para o ponto (0, 1) do espaço ROC. No entanto, concluímos que a escolha de uma ferramenta de detecção de similaridade em código-fonte dependerá bastante do perfil conservador ou liberal do docente.

Palavras-chaves: código-fonte, similaridade, plágio

ABSTRACT

Plagiarism among students of introductory programming courses has been increasing over time. The ease of exchange of information brought by the Internet can be the factor responsible for this increase. In many cases, students try to disguise the plagiarism making some modifications to the source code. However, some masking techniques are extremely complex to be detected and may not be seen with the naked eye.

In this dissertation, detection techniques were analyzed and, on this basis, was developed a system able to detect plagiarism in source code. This system is based on the representation code as an abstract syntax tree and Karp-Rabin Greedy String Tiling algorithm.

The system was evaluated using a source-code base of students of programming disciplines. Oracle based comparison was performed to compare the system with others. The oracle was created from the manual analysis of the teacher of the subject, which was marked if there was plagiarism or not in each pair of source code. To represent the results, ROC curves and confusion matrices were used. The same procedure was applied to existing systems, allowing direct comparison of results. More specifically, we use the value of the area under the curve and the minimum distance to point (0, 1) of the ROC space, since these figures represent the classification performance.

The analysis of results shows that, for the sample used, the developed system obtained higher area under the curve and also the shortest distance to the point (0, 1) of the space ROC. However, we find that the choice of similarity detection tool in source code will depend on conservative or liberal profile of teaching.

Keywords: source-code, similarity, plagiarism.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Muitos estudos sobre plágio foram realizados nos últimos anos. O consenso geral parece ser que a sua incidência tem crescido (DESRUISSEAUX, 1999). Nos Estados Unidos, por exemplo, uma pesquisa indicou que a fraude entre estudantes de graduação aumentou ao longo do último meio século de 23 por cento (DRAKE, 1941), para 90 por cento (JENSEN et al., 2002). Nos últimos anos vários casos foram descobertos e, segundo Ramasesha (2014), um dos fatores que influenciaram na descoberta foi o aperfeiçoamento das ferramentas de detecção de similaridade.

O plágio acontece devido a várias razões, tais como a pressão do tempo, desconhecimento sobre o que constitui plágio e a riqueza de recursos digitais disponíveis na Internet que tornam atividades de copiar e colar quase natural (BIRNBAUM; GOSCILO, 2001). Este último é particularmente verdadeiro no campo da Ciência da Computação e códigos-fonte (OHMANN, 2013).

Similaridade de código-fonte se torna grave nos casos de plágio principalmente nos contextos comercial e educacional (WHALE, 1990). No contexto comercial, muitos produtos similares competem por uma fatia do mercado. Já no contexto educacional, os alunos respondem problemas para obter aprovação em cursos de programação. Iremos abordar apenas o contexto educacional e não iremos considerar plágio de código-fonte em um âmbito geral, apenas em tarefas copiadas de alunos de programação.

Neste trabalho apresentamos o sistema PRIDE. Trata-se de uma ferramenta de detecção de similaridade em códigos-fonte, onde são utilizadas heurísticas baseadas em técnicas de disfarce de cópia de código-fonte. O sistema possui código-fonte fechado, entretanto sua utilização é gratuita.

A validação da ferramenta foi feita utilizando várias submissões de alunos de graduação. Cada par de submissões foi avaliado manualmente pelo docente da disciplina para indicar se havia plágio ou não. Logo após, a ferramenta foi executada e seus resultados comparados com a avaliação manual. Também foi realizado o mesmo procedimento para ferramentas já existentes (JPLAG e MOSS), a fim de compararmos a eficácia entre elas. Os resultados estão apresentados por matrizes de confusão e curvas ROC.

1.1 Motivação

O plágio de código-fonte em disciplinas de programação não pode ser determinado utilizando apenas a métrica de similaridade. É comum haver um valor considerável de similaridade entre códigos-fonte de alunos da mesma turma. Isto acontece devido aos exemplos trazidos pelo docente e também as soluções dadas pelos próprios alunos durante as aulas (COSMA; JOY, 2006). Há também outro fator importante que é a dificuldade e

o tamanho da resolução do problema a ser resolvido. Em geral, problemas para iniciantes em programação são de curta resolução e possuem baixo nível de dificuldade.

No curso de Ciência da Computação da Universidade Federal de Alagoas, uma turma de Programação 1 tem em média 40 alunos matriculados.[1] Isso significa que se um problema for resolvido por todos os alunos, o docente precisará verificar cada código-fonte com todos os outros. Nesses casos, uma ferramenta de detecção de similaridade é extremamente útil, visto que é possível indicar quais pares de códigos-fonte possuem similaridade suspeita.

1.2 Objetivos

Os seguintes objetivos foram definidos para este trabalho:

- Especificar e implementar uma ferramenta de detecção de similaridade em códigos-fonte;
- Avaliar a ferramenta desenvolvida em relação às ferramentas existentes.

1.3 Estrutura do Texto

Esta dissertação está organizada em sete capítulos e dois apêndices. Neste primeiro Capítulo (1) foi descrito o contexto geral do plágio, assim como o contexto do ensino em disciplinas de Programação.

No segundo Capítulo (2) estão apresentados conceitos básicos sobre detecção de similaridade em código-fonte e algumas ferramentas que os utilizam. Inclusive também serão apresentadas algumas listas de técnicas para disfarçar plágio que estão na literatura.

No Capítulo 3 está apresentada a ferramenta desenvolvida. A apresentação será composta por aspectos arquiteturais e funcionais. Serão detalhados, inclusive, os aspectos dos algoritmos em que a ferramenta desenvolvida foi baseada.

No Capítulo 4 estão apresentados os trabalhos relacionados. Além da apresentação, também haverá comparações entre a ferramenta desenvolvida e os trabalhos relacionados. A ideia é apresentar as diferenças do funcionamento de cada uma delas.

No Capítulo 5 está apresentado todo o método utilizado para a realização da validação e comparação das ferramentas. Inclusive uma breve apresentação dos elementos de validação que foram utilizados junto com ilustrações de exemplo para facilitar o entendimento de cada um deles.

No sexto Capítulo (6) estão apresentados os resultados, assim como comentários acerca destes. Teremos um sumário dos dados coletados e as curvas ROC do experimento. Cada um destes elementos será discutido.

[1] Informação obtida no Núcleo de Tecnologia da Informação da Instituição.

Apesar de fazerem parte dos resultados, as matrizes de confusão foram incluídas no Apêndice B, no entanto os comentários sobre estas estão no Capítulo 6.

Por fim, no Capítulo sete 7 são apresentadas as conclusões deste trabalho. Também neste mesmo capítulo, descrevemos as ideias para trabalhos futuros envolvendo a ferramenta.

2 DETECÇÃO DE PLÁGIO EM CÓDIGO-FONTE

Kleiman (2007) afirma que em geral, se um aluno está copiando do outro, é porque não tem conhecimento ou tempo suficiente para completar a tarefa por si próprio. Isso significa que esse mesmo aluno provavelmente não fará alterações extremamente complexas, devido a essa mesma falta de conhecimento ou tempo.

Existem diversas técnicas para efetuar plágio em tarefas de programação. Elas vão desde as mais simples, que acontece quando um aluno copia e cola trechos do programa de outra pessoa; até as mais elaboradas, tal como adicionar código morto que não vai alterar o resultado do programa. Diante dessa variedade, em 1986, após um experimento com alunos de um curso de programação, foram listadas treze técnicas mais utilizadas para disfarçar a origem do código-fonte copiado (WHALE, 1986). Poucos anos mais tarde elas foram revisadas e passaram a ser doze (WHALE, 1990). Estas doze técnicas estão listadas a seguir pelo nível de complexidade em ordem crescente:

1. Mudança de comentário;
2. Mudança de identificadores;
3. Mudança na ordem dos operadores das expressões;
4. Mudança nos tipos de dados;
5. Substituição de expressões por equivalentes;
6. Adição de sentenças redundantes;
7. Mudança na ordem de sentenças independentes;
8. Mudança na estrutura de sentenças de iteração;
9. Mudança na estrutura de sentenças de seleção;
10. Substituição de chamadas procedurais pelo corpo do procedimento;
11. Introdução de sentenças não estruturadas;
12. Combinação de fragmentos originais com copiados.

Em um artigo mais recente (LIU et al., 2006) foi criada uma nova listagem de técnicas. A lista possui cinco técnicas elencadas a seguir.

1. Alteração na formatação;
2. Mudança de identificadores;

3. Reordenação de comandos;
4. Mudança na estrutura das sentenças;
5. Inserção de código morto.

Apesar do artigo ser relativamente novo, a maioria destas já estão inclusas de alguma maneira na lista de Whale (1990). A regra 1, alteração na formatação, é a única que não foi listada anteriormente. Para todas as outras há uma regra equivalente ou um conjunto de regras que equivale. A regra 2 de Liu et al. (2006) equivale à regra 2 de Whale (1990). A regra 3 de Liu et al. (2006) equivale à regra 11 de Whale (1990). A regra 4 (LIU et al., 2006) equivale às regras 9 e 8 de Whale (1990). Por fim, temos a regra 5 de Liu et al. (2006) que equivale à regra 11 de Whale (1990). Outros autores também observaram o comportamento dos seus alunos e também elencaram técnicas. Ahmadzadeh, Mahmoudabadi e Khodadadi (2011), por exemplo, fez uma lista que envolve conceitos de orientação a objetos que possui onze técnicas. São elas:

1. Mudança de identificadores;
2. Mudança de comentário;
3. Trocar a posição das declarações de variáveis e funções;
4. Mudança no escopo da variável;
5. Mudança na assinatura do método;
6. Adição de classes abstratas e interfaces;
7. Transformação estrutural;
8. Substituição de classes por classes da API;
9. Mudança no modificador de acesso das variáveis de instância;
10. Separar em diferentes arquivos a super classe e a *interface*;
11. Mudança na indentação.

É possível observar uma grande semelhança entre as três listas, sendo a terceira a mais diferente, visto que engloba também o conceito de orientação a objetos. A elaboração dessas listas de técnicas foi e é extremamente importante para a criação de ferramentas que auxiliem na detecção de plágio em código-fonte. Algumas destas listas deram origem a ferramentas, como por exemplo: Plague (WHALE, 1988) e GPLAG (LIU et al., 2006).

Em geral, as ferramentas de detecção não especificam que regras estão utilizando para a detecção de similaridade. Isso pode dificultar a escolha de uma ferramenta, visto que,

por exemplo, em uma disciplina de programação orientada a objetos, não seria adequado utilizar o catálogo de Whale (1990). No entanto, Schleimer, Wilkerson e Aiken (2003) defende que as regras utilizadas pela ferramenta não devem ser divulgadas, pois alguém poderia descobrir uma maneira fácil de enganar o sistema.

2.1 Formas de Detecção

Uma ferramenta de detecção de similaridade precisa ser sensível às técnicas de disfarce utilizadas pelos alunos. As duas formas comuns para comparação de programas são: comparação de atributos e comparação de estruturas (JI; WOO; CHO, 2007). A seguir iremos descrevê-las e citar algumas ferramentas que as utilizam.

2.1.1 Comparação Baseada em Atributos

O primeiro sistema de comparação de atributos (OTTENSTEIN, 1976) utilizou as métricas de Halstead (1977) para medir o nível de similaridade entre códigos-fonte, utilizando dados básicos de um programa, como, por exemplo, número de operadores e número de operandos. As métricas sugeridas por Halstead (1977) foram:

$$n_1 = \text{número de operadores únicos} \tag{2.1}$$

$$n_2 = \text{número de operandos únicos} \tag{2.2}$$

$$N_1 = \text{número de ocorrência de operadores} \tag{2.3}$$

$$N_2 = \text{número de ocorrência de operandos} \tag{2.4}$$

Neste sistema, pares de programas com valores n_1, n_2, N_1 e N_2 idênticos eram considerados similares.

Robinson e Soffa (1980) também desenvolveram um sistema de comparação de atributos chamado de ITPAD (Institutional Tool for Program ADvising). Este sistema combinava contagem de estruturas de controle de fluxo com as métricas de Halstead (1977).

Alguns anos depois, Whale (1990) evidenciou que um sistema baseado em comparação de atributos não tem competência para detectar programas similares de maneira aceitável, pois é incapaz de detectar alterações na estrutura. Na época da realização do estudo, nenhum sistema que utilizasse comparação de atributos estava disponível, então, Whale foi forçado a reconstruí-los a partir das descrições presentes na literatura. Mais tarde, Verco e Wise (1996) reforçaram a evidência de Whale (1990) e, inclusive, também tiveram que reconstruir os sistemas que utilizavam comparação de atributos.

São também exemplos dessa técnica os sistemas desenvolvidos por Grier (1981), Berghel e Sallach (1984) e Rambally e Sage (1990). Nenhum desses sistemas foi encontrado disponível para que fosse possível analisar sua eficácia.

2.1.2 Comparação Baseada em Estruturas

Em 1981, Donaldson, Lancaster e Sposato (1981) propôs uma nova técnica chamada de comparação baseada em estruturas como um método adicional para melhorar o desempenho da comparação baseada de atributos. Essa abordagem identifica técnicas simples que estudantes novatos utilizam para esconder o plágio.

Em estratégias de análise de estrutura, o objetivo é buscar trechos de um programa que se apresentam como muito parecidos com trechos de outro programa. A vantagem da análise de estrutura é a possibilidade de encontrar trechos de plágio parcial — casos onde foi copiada apenas parte do programa de um aluno, algo que ocorre bastante na prática (VERCO; WISE, 1996).

Ferramentas que utilizam essa abordagem seguem o princípio da *tokenização*. Cada parte do código-fonte é substituída por um *token* pré-definido. No fim, cada programa é representado por um conjunto de *tokens* e comparados entre si através de algum algoritmo de detecção de padrões. Maiores detalhes sobre o funcionamento da *tokenização* serão apresentados posteriormente na seção 3.2.1.

Três exemplos de ferramentas que utilizam essa técnica são: JPlag (PRECHELT; MALPOHL; PHILIPPSEN, 2001), YAP3 (WISE, 1996) e SIM (GITCHELL; TRAN, 1999).

3 PRIDE

A ferramenta construída foi batizada de PRIDE. O nome é um acrônimo para *PlagiaRIsm DEtection*. Ela está disponível para uso em <http://diogo.4techlabs.com.br/plagiarism>.

A PRIDE tem como objetivo auxilar docentes de disciplinas de programação na detecção de plágio em código-fonte. Para tal, o docente terá que submeter o código-fonte dos alunos na ferramenta para que seja emitido um parecer sobre algum provável plágio. Além dos docentes, a ferramenta também poderá ser utilizada por ambientes de aprendizagem de programação, como o The Huxley.[1]

O sistema possui código fechado e sua apresentação é limitada a trechos de pseudo-códigos e uma breve noção de sua arquitetura. No entanto, seu uso é gratuito, o que garante a possibilidade de reprodução de qualquer um dos resultados apresentados nesta dissertação.

3.1 Arquitetura

A PRIDE foi dividido em três componentes: *pride-web*, *pride-worker* e *pride-core*. O componente *pride-web* é responsável por receber as submissões de código-fonte e prepará-las para processamento. Para tal, foi criada uma API e também uma *interface* em formato HTML.[2] A API foi desenvolvida para que seja possível integrar mais facilmente com outras aplicações e está no formato REST.[3] O componente *pride-core* é responsável por toda a parte do processamento de similaridade, que por sua vez foi dividida em três fases: pré-processamento, comparação e cálculo de similaridade. As três fases serão descritas posteriormente.

Por fim, temos o componente *pride-worker* que é responsável pela troca de mensagens com o componente *pride-web* utilizando o protocolo AMQP.[4] O componente *pride-worker* encapsula o componente *pride-core* para que suas funcionalidades fiquem expostas apenas pela troca de mensagens. A Figura 1 apresenta o diagrama de componentes do sistema.

O principal requisito não funcional foi mitigar problemas de desempenho que podem ser causados pela quantidade de comparações geradas quando submetemos muitos códigos-fonte. Optamos por trabalhar de forma assíncrona utilizando filas para não atrasar a resposta ao usuário ao submeter um código-fonte. No caso de muitos códigos-fonte submetidos, a quantidade de *workers* pode ser aumentada.

---

1 Mais informações: <http://www.thehuxley.com>
2 Mais informações: <http://en.wikipedia.org/wiki/Application_programming_interface>
3 Mais informações: <http://en.wikipedia.org/wiki/Representational_state_transfer>
4 Mais informações: <http://en.wikipedia.org/wiki/AMQP>

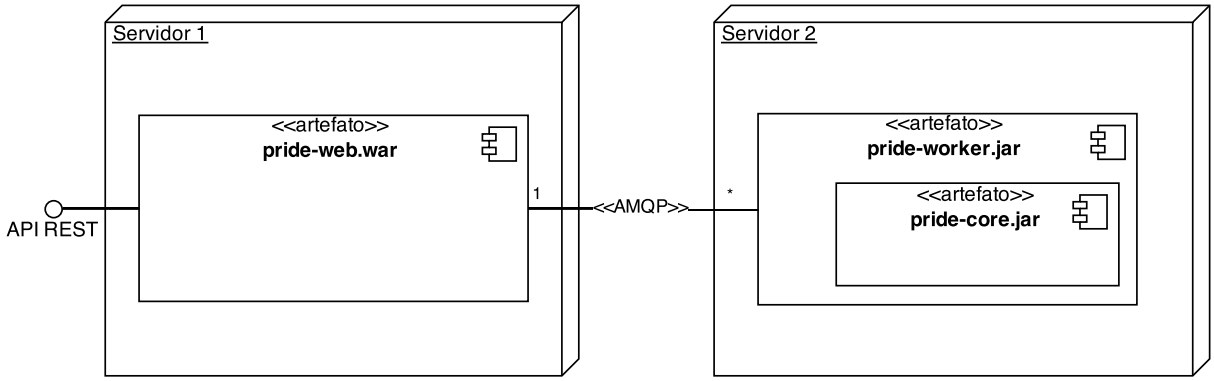

Figura 1 – Diagrama de Componentes do Sistema PRIDE

O sistema tem três principais entidades que estão exibidas na Figura 2. A entidade Grupo é responsável por agrupar as submissões, o agrupamento foi importante para separarmos as submissões por contexto, o interesse é comparar apenas submissões do mesmo problema. A entidade Submissão representa o código-fonte submetido e também contém as informações relativas ao pré-processamento. Por último, temos a entidade Comparação que representa a comparação entre duas Submissões.

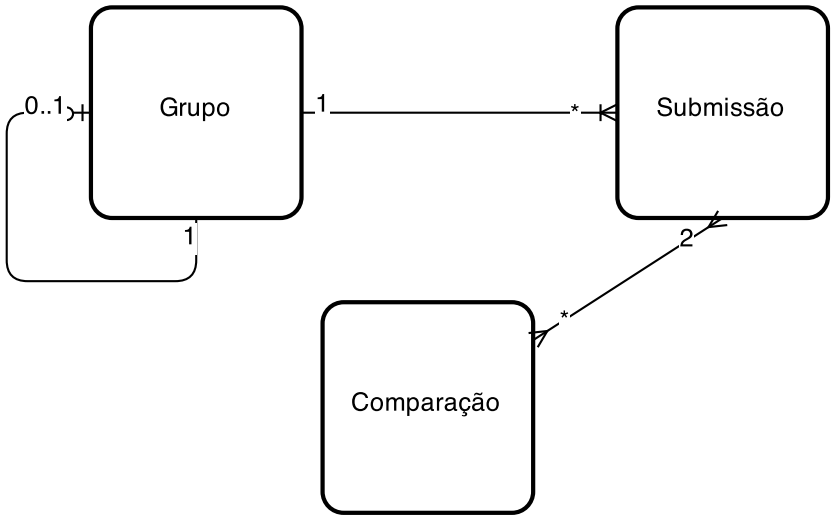

Figura 2 – Diagrama de Entidades do sistema PRIDE

3.2 Núcleo da Ferramenta

Nesta seção é descrito o funcionamento do núcleo da ferramenta (componente *pride-core*), que é responsável por toda parte do cálculo de similaride. O detalhamento foi dividido em três seções que tratam de funções bem específicas, são elas: pré-processamento, comparação e cálculo da similaridade.

3.2.1 Pré-processamento

Nesta primeira fase, recebemos como entrada um código-fonte e devemos retornar como saída uma lista de *tokens*. Em outros sistemas semelhantes, essa fase é chamada de tokenização. Kleiman (2007) inclusive sugere que esta é a fase mais importante na detecção de similaridade em código-fonte.

São realizados dois procedimentos: análise léxica e análise sintática. Os dois procedimentos são realizados pela ferramenta ANTLR que será descrita adiante. Para exemplificar o funcionamento, temos os Códigos 3.1 e 3.2. O primeiro é um programa escrito na linguagem C e representa a entrada recebida pela fase de pré-processamento. O segundo é a representação deste programa por *tokens* em alto nível e representa a saída da fase de pré-processamento.

Listing 3.1 – helloworld.c

```
int main (int argc, char const *argv[])
{
        printf(``Hello World!'');
        return 0;
}
```

A analisador léxico lê o Código 3.1 e verifica se cada lexema casa com algum padrão para um *token* da gramática. Além disso também removerá comentários, formatações e espaços em branco.

O analisador sintático irá construir a árvore sintática da estrutura; que por sua vez será percorrida para que sejam gerados *tokens* em nível mais alto do que os contidos na gramática da linguagem. O intuito é utilizar apenas informações relevantes para a detecção de plágio.

No caso de exemplo de saída representado no Código 3.2, tivemos algumas informações omitidas. Não há *token* representando os parâmetros de entrada da função *main*, também não há representação para o parâmetro "Hello World" na função *printf* e também não há representação para o valor retornado, o zero.

Listing 3.2 – tokens helloworld

```
BEGIN_FUNCTION_DEFINITION
        FUNCTION_CALL
        RETURN
END_FUNCTION_DEFINITION
```

Outro exemplo de entrada e saída são os Códigos 3.3 e 3.4. Neste exemplo temos dois comandos de repetição distintos que geram os mesmos tokens (*BEGIN_LOOP* e *END_LOOP*). Esta modificação pode detectar, por exemplo, a regra 8 (Mudança na estrutura de sentenças de iteração) de Whale (1990).

Listing 3.3 – loop.c

```
int main (int argc, char const *argv[])
{
        int foo = 0;
        while (foo < 10)
        {
                printf(``Hello World!'');
                foo = foo + 1;
        }
        for (int bar = 0; bar < 10; bar++)
        {
                printf(``Hello World!'');
        }
        return 0;
}
```

Nos dois exemplos de saída apresentados (Códigos 3.2 e 3.4), os *tokens* foram indentados para facilitar a leitura e comparação com o código-fonte de entrada. A saída na implementação concreta é apenas uma lista contendo os *tokens*.

Listing 3.4 – tokens loop

```
BEGIN_FUNCTION_DEFINITION
        IDENTIFIER = NUMBER
        BEGIN_LOOP
                FUNCTION_CALL
                IDENTIFIER = IDENTIFIER + NUMBER
        END_LOOP
        BEGIN_LOOP
                FUNCTION_CALL
        END_LOOP
        RETURN
END_FUNCTION_DEFINITION
```

3.2.1.1 Regras Utilizadas

No Capítulo 2 apresentamos alguns catálogos de técnicas de disfarçe de plágio em código-fonte. Para construir a ferramenta, foi necessário escolher quais técnicas iriam ser combatidas. Optamos pelo catálogo de Whale (1990) por entendermos que não há novidade em catálogos mais novos.

Na fase de pré-processamento, conseguimos eliminar a utilização das seguintes técnicas:

1. Mudança de comentário;
2. Mudança de identificadores;
3. Mudança nos tipos de dados;
4. Mudança na estrutura de sentenças de iteração;
5. Mudança na estrutura de sentenças de seleção;

Para a regra 1, adicionamos na gramática uma nova regra de produção que ignora todos os comentários do código-fonte. Para a regra 2, unificamos os nomes de todos os identificadores. Para a regra 3, ignoramos todos os tipos de dados. Por fim, para as regras 4 e 5, normalizamos os *tokens* de iteração e seleção, dessa forma todo comando de iteração gera o mesmo *token*, assim como os de seleção.

As demais regras não puderam ser eliminadas ou mitigadas durante esta fase.

3.2.1.2 ANTLR

O ANTLR é um gerador de *parsers* que pode ler, processar, executar, ou traduzir texto estruturado ou arquivos binários.[5] É utilizado para construir linguagens, ferramentas e *frameworks*. A partir de uma gramática, o ANTLR cria um *parser* que pode gerar e percorrer uma árvore de análise.

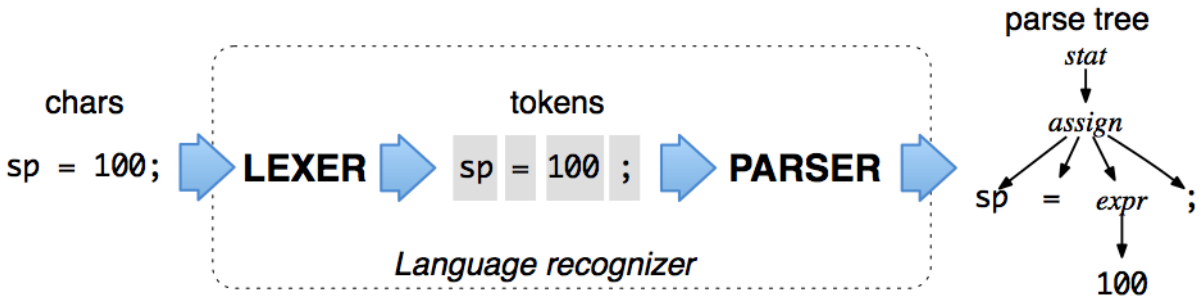

Figura 3 – Funcionamento do ANTLR (PARR, 2013)

A Figura 3 demonstra o funcionamento do ANTLR em nível macro. O fluxo é iniciado com a entrada representada pela *string* "sp = 100;". O próximo passo é a análise léxica, representada por *LEXER*. A saída deste passo é uma lista de *tokens* que serão utilizados como entrada para o próximo passo, o *PARSER*.

[5] Mais informações: <http://www.antlr.org/>

O *PARSER* é responsável por realizar a análise sintática. Neste momento é possível adicionar código-fonte customizado para alterar os *tokens* da árvore de saída. Os chamados *tokens* de alto nível são todos gerados pelo código-fonte customizado. No final desse processo, temos como saída uma árvore sintática de *tokens* de alto nível. Os componentes *LEXER* e *PARSER* são gerados pelo ANTLR. O *PARSER* gerado já possui um mecanismo para acoplar o código-fonte customizado, não necessitando intervenções diretas no componente.

No PRIDE, utilizamos a versão 4.1 do ANTLR para realizar a primeira parte do processamento dos códigos-fonte. Foi utilizada originalmente uma gramática C disponível no repositório do criador do ANTLR, Terence Parr.[6] Foram necessárias diversas modificações na gramática para chegar no resultado esperado. A gramática modificada pode ser encontrada no Apêndice A.

Para adicionar novas linguagens de programação, basta apenas a gramática da linguagem no formato ANTLR 4 e implementar o componente que irá percorrer a árvore gerada pelo *parser*.

3.2.2 Comparação

O objetivo é descobrir as maiores cadeias idênticas de *tokens*. São utilizadas como entrada duas cadeias de *tokens* resultantes do pré-processamento de dois códigos-fonte. Caso haja apenas um código-fonte, esta fase não chega a ser executada. O algoritmo utilizado para comparar os *tokens* é baseado no *Greedy String Tiling* (WISE, 1993). "A abordagem deste algoritmo é considerada avançada e confiável; a maioria dos sistemas de detecção de similaridade implementa uma variação deste algoritmo" (MOZGOVOY, 2008).

Para comparar duas *strings* A e B, este algoritmo tenta encontrar um conjunto de *substrings* que são iguais e satisfaz as seguintes regras:

1. Qualquer *token* de A deve apenas ter uma correspondência exata de um *token* de B.

2. *Substrings* podem ter correspondência independente da posição na *string*.

3. Correspondências de *substrings* maiores possuem prioridade sobre as menores.

É baseado na ideia de que correspondências longas são mais interessantes do que as curtas, pois são mais prováveis de representar semelhanças entre as *strings*.

A Figura 4 mostra o funcionamento do algoritmo com duas listas de *tokens* como entrada, lista A e B, e também a saída do algoritmo, representada pelas correspondências. A primeira correspondência encontrada está representada pela linha verde. Esta

[6] https://github.com/antlr/grammars-v4

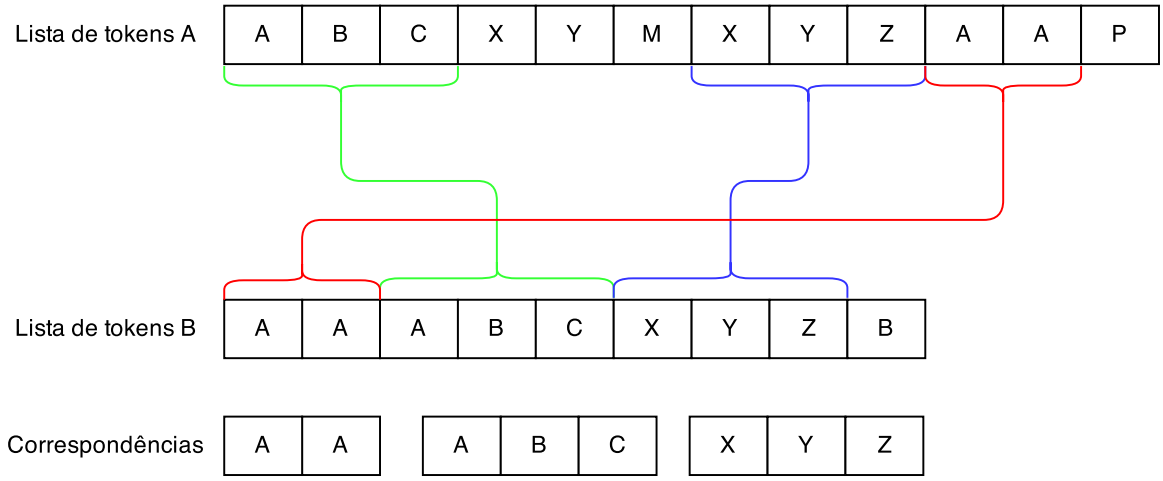

Figura 4 – Funcionamento do algoritmo *Greedy String Tiling*

correspondência é a aplicação direta da regra 2, onde a posição dos *tokens* idênticos é ignorada - a primeira correspondência inicia na posição 1 na lista de *tokens* A e, na lista de *tokens* B, inicia na posição 3. A segunda correspondência está representada pela linha azul. Neste caso são aplicadas as regra 2 e 3. Para a regra 2, temos as duas correspondências em posições diferentes, posição 7 para a lista A e 6 para a lista B. Para a regra 3, temos uma correspondência menor, posição 4 da lista de *tokens* A (X, Y), que é ignorada devido a uma maior correspondência (X, Y, Z). Por fim, temos a terceira correspondência representada pela linha vermelha, onde são aplicadas as regras 1 e 2. A aplicação da regra 1 se apresenta quando não é possível ter como correspondência o terceiro *token* da lista B, pois este já foi marcado na primeira correspondência. A regra 2 se apresenta nas diferentes posições da correspondência - posição 10 para a lista *tokens* A e posição 1 para a lista de *tokens* B.

A seguir serão descritos os algoritmos *Greedy String Tiling* e *Karp-Rabin* (KARP; RABIN, 1987), este último foi aplicado no *Greedy String Tiling* por Wise (1993) com o objetivo de otimizar o desempenho. Essa aplicação originou o algoritmo *Karp-Rabin Greedy String Tiling* (WISE, 1993). A apresentação será feita de forma separada para simplificar o entendimento.

3.2.2.1 *Greedy String Tiling*

O Algoritmo 1 mostra a implementação em pseudocódigo do *Greedy String Tiling* (WISE, 1993). Sua entrada possui dois parâmetros que são as duas listas de *tokens* de cada um dos códigos-fonte.

Para cada *token* A_a da lista A, a primeira fase inicia verificando se há um *token* B_b de

B idêntico a A_a (linhas 6 a 9). Em caso de sucesso, os *tokens* são comparados em ordem para ter a maior correspondência possível (linhas 8 a 11). Nenhum desses *tokens* pode ter sido previamente marcado e para verificar utilizamos a função *isUnmarked* (linha 9). Quando não houver correspondência entre os *tokens* A_{a+j} e B_{b+j} ou algum deles estiver marcado, o tamanho da correspondência máxima é encontrado (linha 12) e é verificado com o último tamanho de correspondência máximo encontrado ($maxMatch$). Caso seja igual, a correspondência é adicionada no conjunto de correspondências *matches* (linha 13). Caso o valor encontrado seja maior, o conjunto de correspondências é reinicializado, a nova correspondência é adicionada e o valor do tamanho da última correspondência máxima é atualizado (linhas 14 a 16).

O operador U utilizado na linha 13 adiciona uma correspondência no conjunto de correspondências se e somente se não sobrescrever alguma já existente no conjunto.

A segunda fase do algoritmo é iniciada na linha 20. Todas as correspondências são iteradas e cada *token* é marcado (linhas 20 a 24). Após a marcação, a correspondência é adicionada ao conjunto final de correspondências *tiles* (linha 25).

O processo é repetido até que não haja nenhuma correspondência maior que o tamanho mínimo definido por $minMaxLength$. A complexidade deste algoritmo em seu melhor caso é $O(n^2)$ e no pior $O(n^3)$ (PRECHELT; MALPOHL; PHILIPPSEN, 2001, p. 13), sendo n o tamanho da maior lista de *tokens*.

3.2.2.2 *Karp-Rabin*

O algoritmo Rabin-Karp (KARP; RABIN, 1987) apresenta uma evolução ao algoritmo de Força-Bruta (GUSFIELD, 1997), ao invés de comparar as *strings*, é proposto usar uma comparação numérica.

Wise (1993) aplicou este algoritmo no *Greedy String Tiling*, criando o *Karp-Rabin Greedy String Tiling*. Com essa modificação, o pior caso continua $O(n^3)$, visto que todas as *substrings* ainda precisam ser comparadas *token* por *token*. No entanto, no experimento prático realizado por Prechelt, Malpohl e Philippsen (2001, p. 14), o algoritmo usualmente apresentou complexidade $O(n^2)$.

No PRIDE, foi gerada uma tabela *hash* para cada uma das listas de *tokens* retornadas pela fase de pré-processamento.[7] Essa tabela permite que a busca por um *token* seja de complexidade $O(1)$ independente do número de registros. No entanto, apesar da redução da complexidade na busca por *tokens*, foi adicionada a construção das tabelas *hash* que possui complexidade $O(n)$.

O Algoritmo 2 mostra a implementação em pseudocódigo do Karp-Rabin (KARP; RABIN, 1987). Sua entrada possui dois parâmetros, o primeiro é o texto e o segundo é o padrão a ser encontrado no texto.

[7] Mais informações: <http://pt.wikipedia.org/wiki/Tabela_de_dispers%C3%A3o>

Algoritmo 1 *Greedy String Tiling*

```
1: função GreedyStringTiling(A, B)
2:     tiles = {}
3:     repita
4:         maxMatch = minimumMatchLength
5:         matches = {}
6:         para todo unmarked token A_a ∈ A faça
7:             para todo unmarked token B_b ∈ B faça
8:                 j = 0
9:                 enquanto A_{a+j} == B_{b+j} & isUnmarked(A_{a+j}) & isUnmarked(B_{b+j}) faça
10:                    j++
11:                fim enquanto
12:                se j == maxMatch então
13:                    matches = matchesUmatch(a, b, j)
14:                senão se j > maxMatch então
15:                    matches = match(a, b, j)
16:                    maxMatch = j
17:                fim se
18:            fim para
19:        fim para
20:        para todo match(a, b, maxMatch) ∈ matches faça
21:            para j = 0 → maxMatch − 1 faça
22:                mark(A_{a+j})
23:                mark(B_{b+j})
24:            fim para
25:            tiles = tiles ∪ match(a, b, maxMatch)
26:        fim para
27:    até (maxMatch > minimumMatchLength)
28:    retorne tiles
29: fim função
```

Algoritmo 2 *Karp-Rabin*

```
1: função KarpRabin(text, pattern)
2:     n = strlen(text)
3:     m = strlen(pattern)
4:     textHash = hash(text)
5:     patternHash = hash(pattern)
6:     para i = 0 → n − m + 1 faça
7:         se textHash == patternHash então
8:             retorne i
9:         fim se
10:        textHash = hash(substr(text, i, m))
11:    fim para
12:    retorne −1
13: fim função
```

Nas linhas 1 e 2 são calculados os tamanhos das duas *strings* de entrada. Na linhas 3 e 4, são calculados os *hashes* das duas *strings* de entrada. É verificado se os *hashes* calculados anteriormente são iguais (linha 7). Caso seja, é retornada a posição inicial do padrão encontrado no texto. Caso não seja, é calculado um novo *hash* (linha 9) avançando uma posição. Esse processo é repetido até encontrar o padrão ou até o tamanho do texto calculado chegue em um tamanho menor que o padrão a ser encontrado. Caso não encontre, é retornado o valor -1 (linha 11).

3.2.3 Cálculo da Similaridade

A medida de similaridade deve refletir a fração de *tokens* dos programas originais que foram copiados. Se quisermos um único valor de similaridade para os dois códigos-fonte, temos de considerar o tamanho das duas listas de *tokens* no cálculo. Se preferirmos que cada programa que foi copiado por completo (e, em seguida, talvez estendido) resulte em um valor de similaridade, temos que considerar apenas a lista de *tokens* mais curta. O PRIDE utiliza a primeira opção, o que resulta nas Equações 3.1 e 3.2.

Na Equação 3.1, temos o somatório do comprimento do conjunto de correspondências resultantes do Algoritmo 1 (*tiles*). Na Equação 3.2, temos o cálculo da similaridade. Este cálculo utiliza a função de cobertura representada pela Equação 3.1 utilizando como parâmetro o conjunto de correspondências encontradas (*tiles*). Também são utilizados como parâmetro o tamanho das listas de *tokens* utilizadas como entrada na fase de comparação (3.2.2). Na Equação 3.2 o tamanho das listas está representado pela norma do vetor A e B.

$$coverage(tiles) = \sum_{match(a;b;length) \ \in \ tiles} length \tag{3.1}$$

$$sim(A, B) = \frac{2 \cdot coverage(tiles)}{|A| + |B|} \tag{3.2}$$

Para o exemplo apresentado na Figura 4, encontramos as correspondências {A, A}, {A, B, C} e {X, Y, Z} nas listas de *tokens* A e B; {A, B, C, X, Y, M, X, Y, Z, A, A, P} e {A, A, A, B, C, X, Y, Z, B} respectivamente. Abaixo aplicamos as equações 3.1 e 3.2 para demostrar o valor de similaridade encontrado para este exemplo.

$$coverage(tiles) = 2 + 3 + 3 \tag{3.3}$$

$$coverage(tiles) = 8 \tag{3.4}$$

$$sim(A, B) = \frac{2 \cdot 8}{12 + 9} \tag{3.5}$$

$$sim(A, B) = 0,761 \tag{3.6}$$

A equação 3.1 soma o tamanho de todas as correspondências encontradas. Estes passos são apresentados em 3.3 e 3.4. A equação 3.2 apresenta o valor encontrado em 3.1 multiplicado por dois em seu numerador. Já em seu denominador, é apresentada a soma da norma do vetor A com a norma do vetor B, que significa a soma dos tamanhos das listas A e B do exemplo da Figura 4. O desenvolvimento dessa equação é apresentado em 3.5 e 3.6.

3.3 Processo de Submissão

Nesta seção descreveremos um processo simples de utilização do sistema. Não é obrigatório que todo usuário o utilize. Cada docente provavelmente deve ter sua sistemática de trabalho e, inclusive, o usuário pode ser também uma outra aplicação.

A Figura 5 mostra um processo utilização do PRIDE em notação BPM.[8] Ao todo teremos três autores neste exemplo: o sistema PRIDE, o Docente e o Discente. O processo inicia quando o Docente disponibiliza um problema para ser resolvido. O Discente resolve o problema e envia sua solução ao Docente. Após receber a solução, o Docente submete o código-fonte para o PRIDE, que por sua vez recebe o código-fonte e realiza o pré-processamento (3.2.1). Caso não haja nenhum outro código-fonte submetido, o processo é finalizado, pois não é possível emitir um percentual de similaridade com apenas um código-fonte. Caso haja outros códigos-fonte submetidos, é realizada a comparação (3.2.2) dos dados retornados pela fase de pré-processamento. Por fim, é realizado o cálculo de similaridade (3.2.3), e para cada par de código-fonte enviado é enviada uma lista de códigos-fonte suspeitos de plágio. Como não podemos afirmar que um código-fonte é plágio apenas pela sua similaridade, a responsabilidade de decidir passa para o Docente.

[8] Mais informações: <http://en.wikipedia.org/wiki/Business_process_modeling>

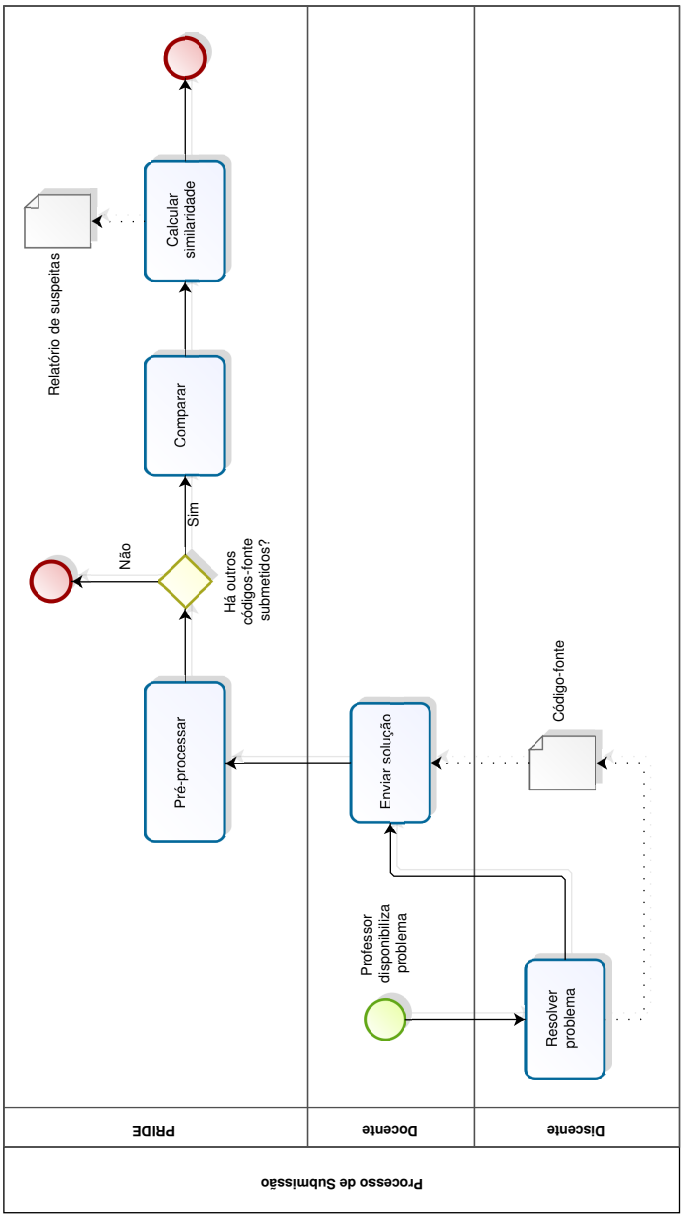

Figura 5 – Processo de Submissão

4 TRABALHOS RELACIONADOS

Nesta seção listaremos alguns trabalhos relacionados, bem como compararemos o funcionamento de cada um com o sistema PRIDE.

4.1 JPlag

A ferramenta JPlag foi implementada por Prechelt, Malpohl e Philippsen (2001) e está disponível em <https://jplag.ipd.kit.edu/>. Seu funcionamento é dividido em três fases. A primeira é o pré-processamento, onde o código-fonte é transformado em um conjunto de *tokens*. Na segunda fase é executado o algoritmo *Karp-Rabin Greedy String Tiling* (WISE, 1993) nos *tokens* gerados na primeira fase. Na última fase é gerado um relatório contendo o valor de similaridade para cada par de código-fonte. O relatório é disponibilizado no formato HTML.

O código-fonte do JPlag é fechado, portanto não foi possível ter acesso à sua implementação. Entretanto, há uma descrição completa do funcionamento da ferramenta no artigo de Prechelt, Malpohl e Philippsen (2001). Possui suporte às linguagens Java, C#, C, C++, Scheme e texto livre.

O JPlag funciona de forma muito semelhante ao PRIDE inclusive é baseado no mesmo algoritmo para detecção de padrões, o *Karp-Rabin Greedy String Tiling*. Também possui três fases: pré-processamento (*tokenização*), processamento e pós-processamento. No entanto, há uma junção entre a execução do algoritmo de comparação e a execução do cálculo de similaridade para formar a segunda fase. No PRIDE são fases distintas e a geração de relatórios não é contada como fase.

O PRIDE também utiliza as mesmas funções 3.1 e 3.2 para cálculo de similaridade que o JPlag. Portanto, a grande diferença entre os dois são os *tokens* gerados. Há um apêndice em seu artigo apresentando os *tokens* que podem ser gerados para a linguagem JAVA.

4.2 MOSS

O *Measure of Software Similarity* (MOSS) é um sistema que foi desenvolvido por Schleimer, Wilkerson e Aiken (2003). Seu código-fonte é fechado e é baseado no algoritmo *Winnowing* (SCHLEIMER; WILKERSON; AIKEN, 2003). A ferramenta está disponível em <http://theory.stanford.edu/~aiken/moss/>.

Detalhes de sua implementação não foram liberados ao público, sob o risco de que, ao conhecer o funcionamento do sistema, alguém poderia descobrir uma maneira de enganar o sistema. O artigo de Schleimer, Wilkerson e Aiken (2003) indica que há três fases: pré-processamento, processamento e pós-processamento; exatamente como o JPlag. Algumas

observações do artigo podem sugerir que não há grande manipulação sintática na fase de pré-processamento.

Atualmente possui suporte a várias linguagens de programação. São elas: C, C++, Java, C#, Python, Visual Basic, Javascript, FORTRAN, ML, Haskell, Lisp, Scheme, Pascal, Modula2, Ada, Perl, TCL, Matlab, VHDL, Verilog, Spice, MIPS assembly, a8086 assembly, a8086 assembly, MIPS assembly e HCL2.

4.3 SIM

O SIM utiliza o algoritmo de alinhamento local (GITCHELL; TRAN, 1999). O sistema, antes de processar algum documento, o quebra em cadeias de *tokens* - *tokenização*. É interessante notar que o SIM não descarta comentários do código, estes são considerados como um tipo de *token*. Também funciona de maneira semelhante ao PRIDE, tendo como principais diferenças a utilização algoritmo de Gitchell e Tran (1999) e o fato de não descartar comentários do código-fonte.

Esta ferramenta pode ser encontrada em <http://dickgrune.com/Programs/similarity_tester/> e dá suporte a C, Java, Pascal, Modula-2, Miranda e Lisp.

4.4 Sherlock

O Sherlock (JOY; LUCK, 1998; PIKE, 2012) é uma alternativa às ferramentas proprietárias, pois possui código aberto, permitindo modificações e melhorias. Para encontrar trechos duplicados de cada código-fonte, é gerada uma assinatura digital calculando valores hash para comandos e sequência de comandos. Ao final, comparam-se as assinaturas geradas e identifica-se o percentual de semelhança. A ferramenta está disponível em <http://sydney.edu.au/engineering/it/~scilect/sherlock/>.

O Sherlock não faz análise da estrutura do código-fonte. Isso interfere bastante quando, por exemplo, temos um código-fonte com vários comentários. Essa diferença pode produzir resultados extremamente diferente das ferramentas apresentadas até aqui.

4.5 YAP3

O *Yet Another Plague 3* (YAP3) foi desenvolvido por Wise (1996). Esta é a terceira versão da ferramenta, as anteriores foram: YAP1 e YAP2. O sistema YAP3 e seu código-fonte estão disponíveis em <http://luggage.bcs.uwa.edu.au/~michaelw/YAP.html>. Seu site não apresenta as linguagens suportadas.

O processamento do YAP é dividido em duas fases: pré-processamento e comparação. A primeira fase envolve várias operações, como, remoção de comentários e constantes, tradução de letras maiúsculas para minúsculas, mapeamento de operações de mesma finalidade para uma forma comum (Exemplo: *strcpy* e *strncpy*), reordenação das funções

na ordem de chamada e remoção de todos os *tokens* que não fazem parte das palavras-chave da linguagem. É interessante notar que a remoção dos *tokens* que não fazem parte das palavras-chave, remove, por exemplo, variáveis do corpo do programa.

Na fase de comparação, o método utilizado difere entre as versões do YAP. A versão original (YAP) utiliza uma comparação baseada na ferramenta "sdiff" do UNIX.[1] A segunda versão (YAP2) utiliza o algoritmo de Heckel (HECKEL, 1978). Já a última versão (YAP3) utiliza o algoritmo *Karp-Rabin Greedy String Tiling* (WISE, 1993).

[1] Mais informações: http://c2.com/cgi/wiki?DiffAlgorithm

5 METODOLOGIA DO EXPERIMENTO

O objetivo deste capítulo é descrever o experimento de análise da ferramenta construída. A análise será baseada na seguinte pergunta:

- A ferramenta apresenta melhor resultado que as outras ferramentas analisadas?

A metodologia foi particionada em cinco seções. São elas: Seleção dos Dados 5.1, Avaliação Manual 5.2, Escolha das Ferramentas 5.3, Execução das Ferramentas 5.4, Elementos para a Avaliação 5.5 e Avaliação 5.6.

5.1 Seleção dos Dados

Foi solicitado ao docente da disciplina Programação 1 do curso de Ciência da Computação da Universidade Federal de Alagoas acesso aos códigos-fonte dos alunos do último semestre realizado (2014/1). A princípio seriam utilizados códigos-fonte do semestre inteiro. No entanto, os problemas do início do curso são muito simples, como por exemplo, imprimir a mensagem "Olá mundo!". Esse tipo de problema não é útil para a análise, visto que a porcentagem de similaridade seria muito alta até mesmo para casos de pares de códigos-fonte originais. Assim, foram utilizados problemas que tivessem ao menos uma estrutura de decisão ou repetição. Os problemas utilizados foram:

- ***Ambrosio's Bus Company***: Ambrósio é sócio de muitas empresas. Uma das empresas de ambrósio é a Ambrósio's Bus Company que oferece o serviço de viagens de ônibus intermunicipais. A Figura 6 mostra a ficha de registro de cada passageiro. Os ônibus possuem 44 lugares numerados de 1 a 44. Você vai receber uma lista de passageiros com no máximo 44 elementos. Ao final imprima o nome de todos os passageiros que estiverem acima da média das idades e que estiverem sentados nas poltronas pares.

- **Características Físicas**: Uma pesquisa sobre algumas características físicas da população de uma determinada região coletou os seguintes dados, referentes a cada habitante, para serem analisados: sexo (masculino (m), feminino (f)); cor dos olhos (azuis (a), verdes (v), castanhos (c)); cabelos (louros (l), castanhos (c), pretos(p)); idade em anos. Para cada habitante, foram digitadas duas linhas, a primeira com a idade e a segunda os outros dados e a última linha, que não corresponde a ninguém, conterá o valor igual a -1. Escreva um programa que dê a idade do habitante mais velho e a porcentagem de mulheres que tenham idade maior ou igual a 18 e menor ou igual a 35 anos, sejam louras e tenham olhos verdes. O valor deve ser calculado considerando o total de pessoas e não somente o total de mulheres.

Figura 6 – Ficha de registro do passageiro (TEAM, 2013)

- **Colchão**: João está comprando móveis novos para sua casa. Agora é a vez de comprar um colchão novo, de molas, para substituir o colchão velho. As portas de sua casa têm altura H e largura L e existe um colchão que está em promoção com dimensões A B C. O colchão tem a forma de um paralelepípedo reto retângulo e João só consegue arrastá-lo através de uma porta com uma de suas faces paralelas ao chão, mas consegue virar e rotacionar o colchão antes de passar pela porta. Entretanto, de nada adianta ele comprar o colchão se ele não passar através das portas de sua casa. Portanto ele quer saber se consegue passar o colchão pelas portas e para isso precisa de sua ajuda.

- **Festa da Marilda**: Marilda é uma menina bem desorganizada, avessa aos computadores mas bastante festeira. Por isso, toda festa que Marilda organiza é uma confusão, não sabe direito quem convidou, uma bagunça só. Seu irmão, aluno do curso de computação, está tentando convencê-la que ela deve se organizar melhor e deve utilizar um computador para registrar a lista de convidados da sua festa. Porém, os convidados podem chegar a 300 pessoas e achar um nome dentre esses trezentos pode ser complicado. Então a sua função é escrever um programa que receba a lista de convidados de Marilda e a imprima em ordem alfabética. Porém, Marilda organiza várias festas e, portanto, você terá que ordenar várias listas de convidados diferentes.

- **Foto de Família**: Ambrosina é uma fotógrafa muito peculiar. Ela só aceita tirar fotos de pessoas se as pessoas estiverem em grupos de exatamente 4 pessoas. Tudo isso porque Ambrosina tem uma mania esquisita de ordenação. Para ela, a pessoa mais baixa deve ficar sempre do lado esquerdo, a segunda mais baixa do lado direito. Logo após a mais baixa, fica a terceira mais baixa e em seguida a mais alta. A Figura 7 é uma ilustração de uma foto tirada por Ambrosina.

- **Futebol**: Seu time de futebol favorito vai jogar um torneio de caridade, que é parte de um esforço de angariação de fundos em todo o mundo para ajudar as

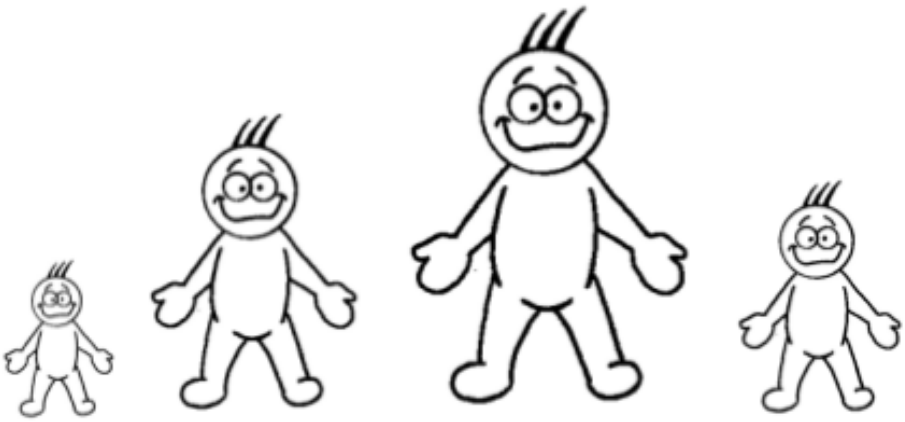

Figura 7 – Foto de família (TEAM, 2013)

crianças com deficiência. Como em um torneio normal, 3 pontos são concedidos para a equipe ganhar uma partida, 0 pontos para a equipe perdedora. Se a partida acaba empatada, cada equipe recebe 1 ponto. Seu time jogou N partidas durante a primeira fase do torneio, que acabou de terminar. Apenas algumas equipes, as únicos com mais pontos acumulados, vão avançar para a segunda fase do torneio. No entanto, como o principal objetivo do torneio é arrecadar dinheiro, antes dos jogo das equipes que passarão para a segunda fase serem determinados, para cada equipe é permitido comprar gols adicionais. Esses novos gols contarão como gols normalmente marcados, e podem ser usados para alterar o resultado de qualquer um dos jogos que o time jogou. Orçamento da sua equipe é suficiente para comprar até L gols. Você pode dizer o número máximo total de pontos de sua equipe pode obter depois de comprar os gols, supondo que os outros times não vão comprar gols?

- **Lanchonete da Ambrosina**: Ambrosina é de lua. Quase todo dia ela quer mudar o cardápio de sua lanchonete. Ambrósio, o programador do sistema, não agüenta mais alterar o código do programa todas as vezes que Ambrosina muda o humor. Então Ambrósio resolveu mudar o software de forma que Ambrosina possa cadastrar o cardápio todas as manhãs, ao iniciar o software. O software lê a quantidade de produtos a serem cadastrados, e depois o código de LED instalada na lanchonete seguidos pela descrição e preço do produto. Os clientes escolhem os produtos pelo código. Se o cliente pede um produto não cadastrado ou uma quantidade negativa o pedido é considerado inválido. O sistema calcula quantos itens o cliente escolheu de cada código e imprime o total da conta, sem descontos! Eita Ambrosina sovina!!

- **Letra Maiúscula**: Um professor de Português estava corrigindo resumos de textos que seus alunos fizeram, porém, encontrou uma falha grave em todos os textos. No início das frases e após os pontos finais, a primeira letra estava escrita em letra minúscula. Ao questionar seus alunos, descobriu que era um erro do editor de texto que todos usaram. Como ele é muito seu amigo, e você ficou sabendo desse problema

que ele enfrentava então se ofecereu para criar um algoritmo para corrigir esse erro. Crie um programa que troque os caracteres de início de frase e após ponto final em minúsculo para maiúsculo.

- **Lua Cheia**: Ambrósio resolveu perder a barriguinha dando uma corridinha na praia. Depois de alguns dias treinando, ele achou que já era hora de competir provas de curta distância. Mas onde ele mora, o sol é muito forte e, portanto, ele só corre a noite. Ele descobriu que existe uma corrida que ocorre a cada lua cheia. A lua cheia se repete a cada 30 dias, aproximadamente. Logo, se você souber a data da última lua cheia, conseguirá prever qual será a próxima. Escreva um programa que recebe a data da última lua cheia e imprime a data da próxima lua cheia.

- **Permutador de trem**: Em uma antiga estação ferroviária, você ainda pode encontrar um dos últimos remanescentes "Permutador de Trem". Um manobrista de trem é um empregado da estrada de ferro, cujo único trabalho é reorganizar as carruagens dos comboios. Uma vez que as carruagens são dispostas na ordem ideal, todo maquinista do comboio tem que retirar as carruagens, uma a uma, até as estações que a carga se destina. O título "permutador de trem" deriva da primeira pessoa que realizou essa tarefa, em um posto perto de uma ponte ferroviária. Em vez de abertura vertical, a ponte fica girando em torno de um pilar no centro do rio. Depois de girar a ponte de 90 graus, os barcos poderiam passar para a esquerda ou direita. O primeiro permutador de trem descobriu que a ponte poderia ser operada com, no máximo, duas carruagens nela. Ao girar a ponte de 180 graus, as carruagens trocavam de posição, permitindo-lhe reorganizar os carros (como um efeito colateral, as carruagens, ficam na direção oposta, mas vagões podem mover-se para qualquer direção, então isso realmente não importa). Agora que quase todos os permutadores de trem morreram, a empresa ferroviária gostaria de automatizar sua operação. Parte do programa a ser desenvolvido, é uma rotina que decide por um determinado trem o menor número de trocas de duas carruagens adjacentes necessárias para ordenar o trem. Sua tarefa é criar essa rotina. Lembre-se, a rotina usa uma lógica computacional, por isso, seguirá uma lógica de algoritmos de ordenação com função de troca dois a dois, causando um número maior de trocas do que aconteceria numa lógica normal.

O docente da disciplina permite a submissão em diferentes linguagens, no entanto, como a grande maioria submeteu códigos-fonte na linguagem C, iremos descartar as outras.

5.2 Avaliação Manual

O docente da disciplina avaliou manualmente cada código-fonte com todos os outros submetidos para verificar se havia indícios de plágio. Caso houvesse, o par deveria ser marcado como plágio.

Algumas informações foram omitidas para não interferir no julgamento do docente; foram elas: autor da submissão, data da submissão e hora da submissão. Essas informações poderiam interferir na avaliação, pois depois de um semestre inteiro, o docente conhece o perfil da maioria dos alunos da disciplina. A data e a hora poderiam também interferir, pois geralmente o aluno que faz plágio submete seu código-fonte após o original.

Foi desenvolvido um ambiente na ferramenta proposta para facilitar a avaliação do docente durante o experimento. A Figura 8 demonstra o ambiente de avaliação. Para todas as comparações foi mostrado 0% de similaridade e o docente da disciplina foi avisado sobre este detalhe para não interferir em seu julgamento. O ambiente utiliza o plugin *jsdifflib* para realizar *diff* em códigos-fonte.[1]

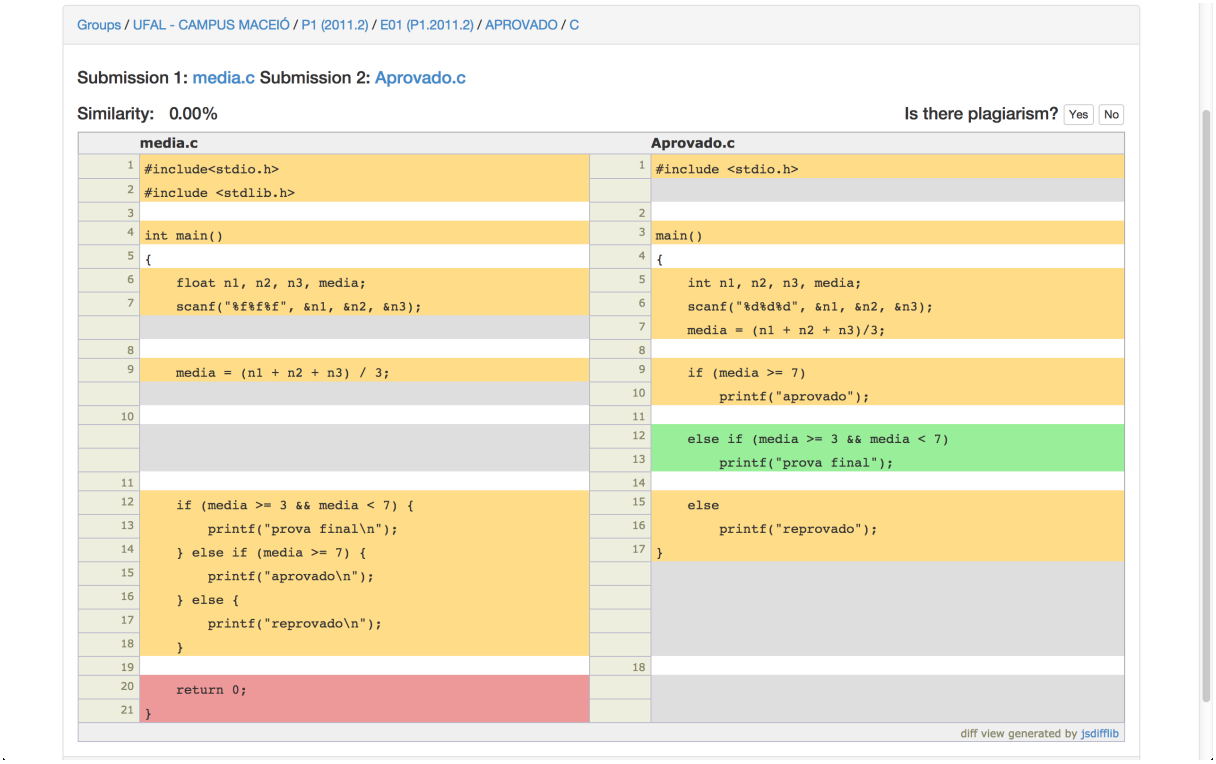

Figura 8 – Ferramenta para avaliação manual

[1] https://github.com/cemerick/jsdifflib

5.3 Escolha das Ferramentas

Vários estudos (PRECHELT; MALPOHL; PHILIPPSEN, 2001; CHEN et al., 2004; LIU et al., 2006; MOZGOVOY, 2008) sugerem que não existe ferramenta definitiva para a detecção de plágio em código-fonte. No entanto, o estudo qualitativo de Lancaster e Culwin (2004) demonstra que as ferramentas JPLAG (PRECHELT; MALPOHL; PHILIPPSEN, 2001) e MOSS (SCHLEIMER; WILKERSON; AIKEN, 2003) apresentam melhores resultados que as outras. Por ser de natureza qualitativa, Lancaster e Culwin (2004) descreve e compara algumas propriedades de cada ferramenta, como por exemplo: linguagens de programação que suporta, funcionamento é local ou baseado na web, algoritmo utilizado para comparação, etc. O resultado deste estudo foi o fator utilizado para a escolha dessas duas ferramentas.

5.4 Execução das Ferramentas

Executamos cada uma das ferramentas de detecção para os códigos-fonte analisados manualmente. Foram enviados um par por vez a fim de haver apenas uma porcentagem de similaridade para cada entrada.

Foram criados alguns *scripts* para automatizar a submissão dos pares de código-fonte tanto para o JPLAG (PRECHELT; MALPOHL; PHILIPPSEN, 2001) quanto para o MOSS (SCHLEIMER; WILKERSON; AIKEN, 2003). Já a ferramenta PRIDE foi integrada diretamente ao ambiente de submissão utilizado pelo docente da disciplina.

5.5 Elementos para a Avaliação

Para avaliar o resultado das ferramentas, foram utilizadas matrizes de confusão, espaços ROC (EGAN, 1975; FAWCETT, 2006) e curvas ROC (EGAN, 1975; FAWCETT, 2006). As matrizes de confusão são usadas para registrar observações independentes de duas ou mais variáveis aleatórias, normalmente qualitativas. O espaço ROC é utilizado para medir a eficácia de um classificador binário. A Curva ROC é usada para medir a eficácia relativa de um classificador.

5.5.1 Matriz de Confusão

A matriz de confusão mostra o número de classificações corretas em oposição às classificações preditas para cada classe. Pode ser representada por uma tabela com duas linhas e duas colunas que informa o número de falsos positivos, falsos negativos, verdadeiros positivos e verdadeiros negativos.[2] Na Figura 9 temos um exemplo de matriz de confusão. Na primeira caixa ficam as amostras que foram corretamente classificadas como verdadeiras (VP), abaixo desta caixa ficam as amostras que foram erroneamente classificadas como

[2] Para maiores detalhes (FAWCETT, 2006)

verdadeiras (FP). Seguindo a mesma ordem, no outro lado temos a primeira caixa com as amostras que foram classificadas erroneamente como negativas (FN) e abaixo desta temos as que foram corretamente classificadas como negativas (VN). O valor P′ informa o total de amostras positivas e o N′ o total de negativas. Os valores P e N informam os valores retornados pelo classificador, positivos e negativos respectivamente.

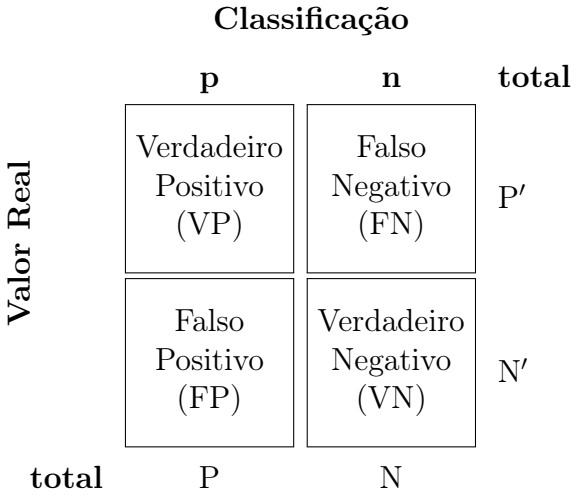

Figura 9 – Matriz de Confusão

A partir desta tabela podemos calcular várias métricas comuns, tais como: taxa de falso positivo (Equação 5.1), taxa de verdadeiro positivo (Equação 5.2), precisão (Equação 5.3), *recall* (Equação 5.4) e acurácia (Equação 5.5).

$$txFP = \frac{FP}{N} \tag{5.1}$$

$$txVP = \frac{VP}{N} \tag{5.2}$$

$$precisao = \frac{VP}{VP + FP} \tag{5.3}$$

$$recall = \frac{VP}{P} \tag{5.4}$$

$$acuracia = \frac{VP + VN}{P + N} \tag{5.5}$$

Para exemplificar, considere um classificador para identificar humanos infectados com uma doença Z. Considere também que este mesmo classificador foi executado em uma amostra composta por 100 pacientes; 23 estão infectados (P′) e 77 não estão (N′). Dos 23 infectados o classificador identificou 19 (VP) e deixou de identificar 4 (FN). Já os 77 não infectados, o classificador identificou 65 (VN) e errou 12 (FP). O valor P é igual a soma das classificações positivas que seriam os 19 infectados classificados como positivo

e os 12 não infectados classificados como positivo. Já o valor N é a soma dos 4 infectados não classificados como positivo e os 65 não infectados classificados como negativo. A Figura 10 mostra como ficaria a matriz confusão deste exemplo.

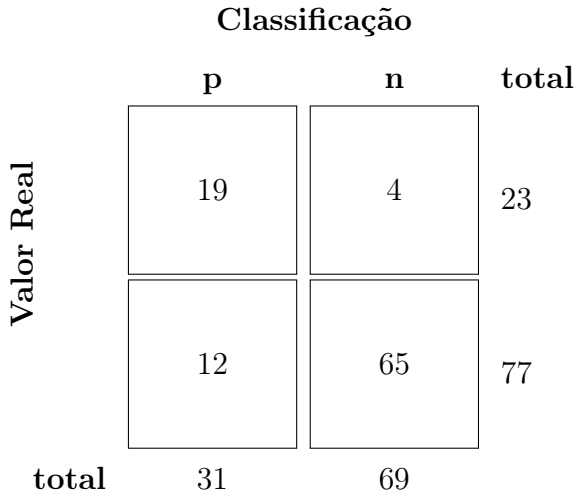

Figura 10 – Exemplo de utilização da Matriz de Confusão

5.5.2 Espaço ROC

O espaço ROC é uma representação gráfica que ilustra o desempenho de um classificador binário.[3] Seu gráfico é em duas dimensões, onde a taxa de verdadeiro positivos é plotada no eixo y e a taxa de falso positivos é plotada no eixo x. O ponto (0, 0) representa um classificador que classifica todos os casos como negativos e o ponto (1, 1) representa um classificador que classifica todos os casos como positivos.

Na Figura 11 temos um espaço ROC contendo a avaliação de cinco classificadores binários. Os pontos acima da linha cinza são considerados bons resultados, já os pontos abaixo são considerados resultados ruins. O classificador A atingiu um resultado perfeito, isso significa que todas as classificações foram corretas. O D é chamado de classificador conservador, pois só classifica como positivo quando há uma grande evidência. Ou seja, possui pouquíssimos falsos positivos e muitos falsos negativos. O classificador C é chamado de liberal; classifica muitos falsos positivos e poucos falsos negativos. Já o B é chamado de aleatório, pois não consegue classificar adequadamente. Por fim o classificador E, que apesar de ter um resultado ruim, pode se tornar um bom classificador bastando apenas inverter a saída para que o ponto encontrado fique acima da linha cinza.

Segundo Fawcett (2006), informalmente, um ponto no espaço ROC é melhor do que outro, se este estiver mais para o noroeste do primeiro.

[3] Para maiores detalhes, ver (EGAN, 1975; FAWCETT, 2006)

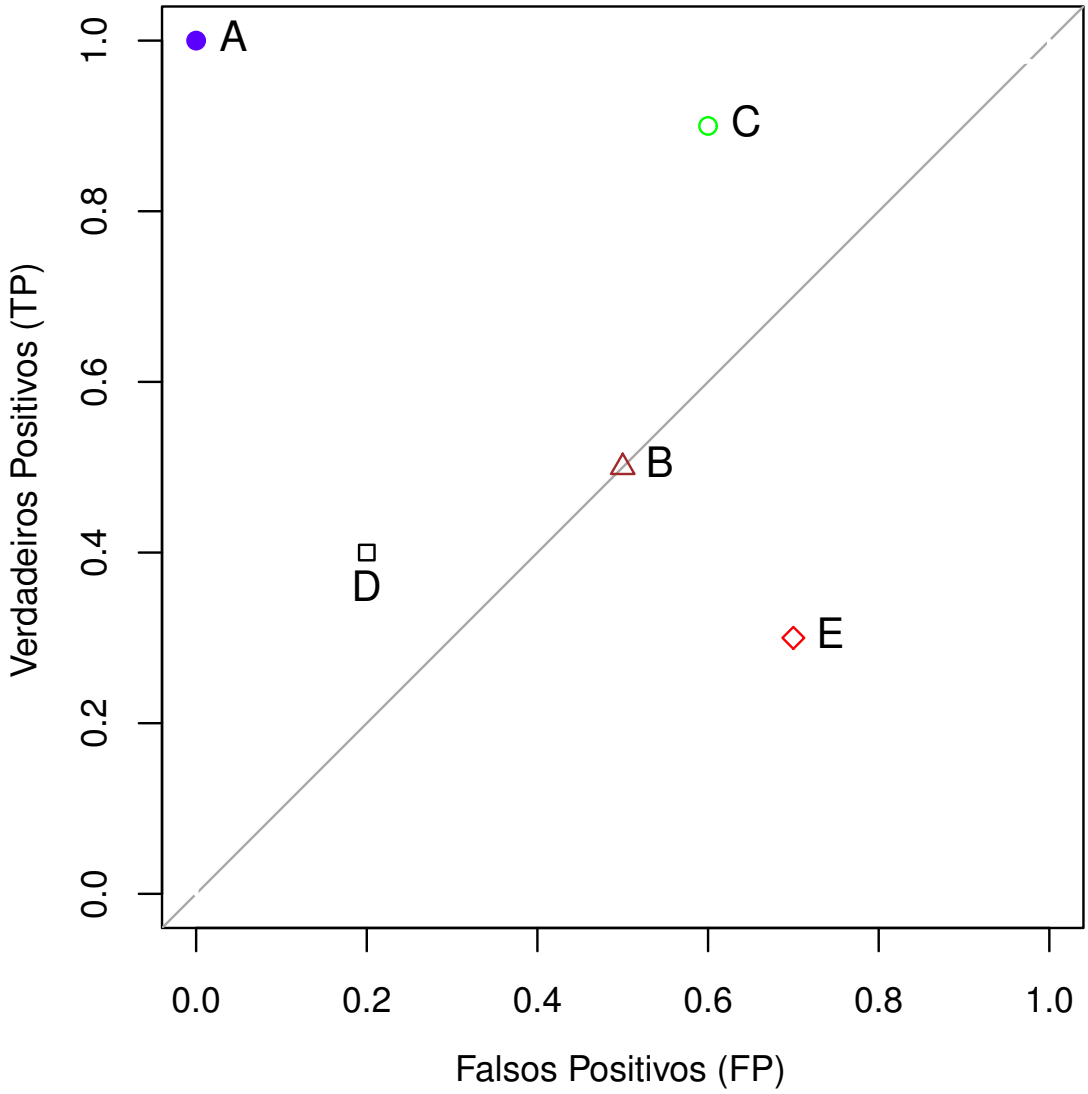

Figura 11 – Espaço ROC

5.5.3 Curva ROC

A curva ROC apresenta uma funcionalidade adicional ao espaço ROC, visto que podemos observar o comportamento do classificador quando seu limiar de discriminação é variado.[3] Essa variação é bastante importante, visto que em nosso experimento não temos um classificador binário e não podemos considerar que, por exemplo, um par de códigos-fonte é plágio se e somente se possuir uma similaridade acima de 80%. Este percentual precisa ser variado para cada uma das ferramentas, para não prejudicar ou ajudar qualquer uma delas. A Figura 12 mostra uma curva ROC. Cada ponto desta curva representa o desempenho do classificador em um limiar diferente.

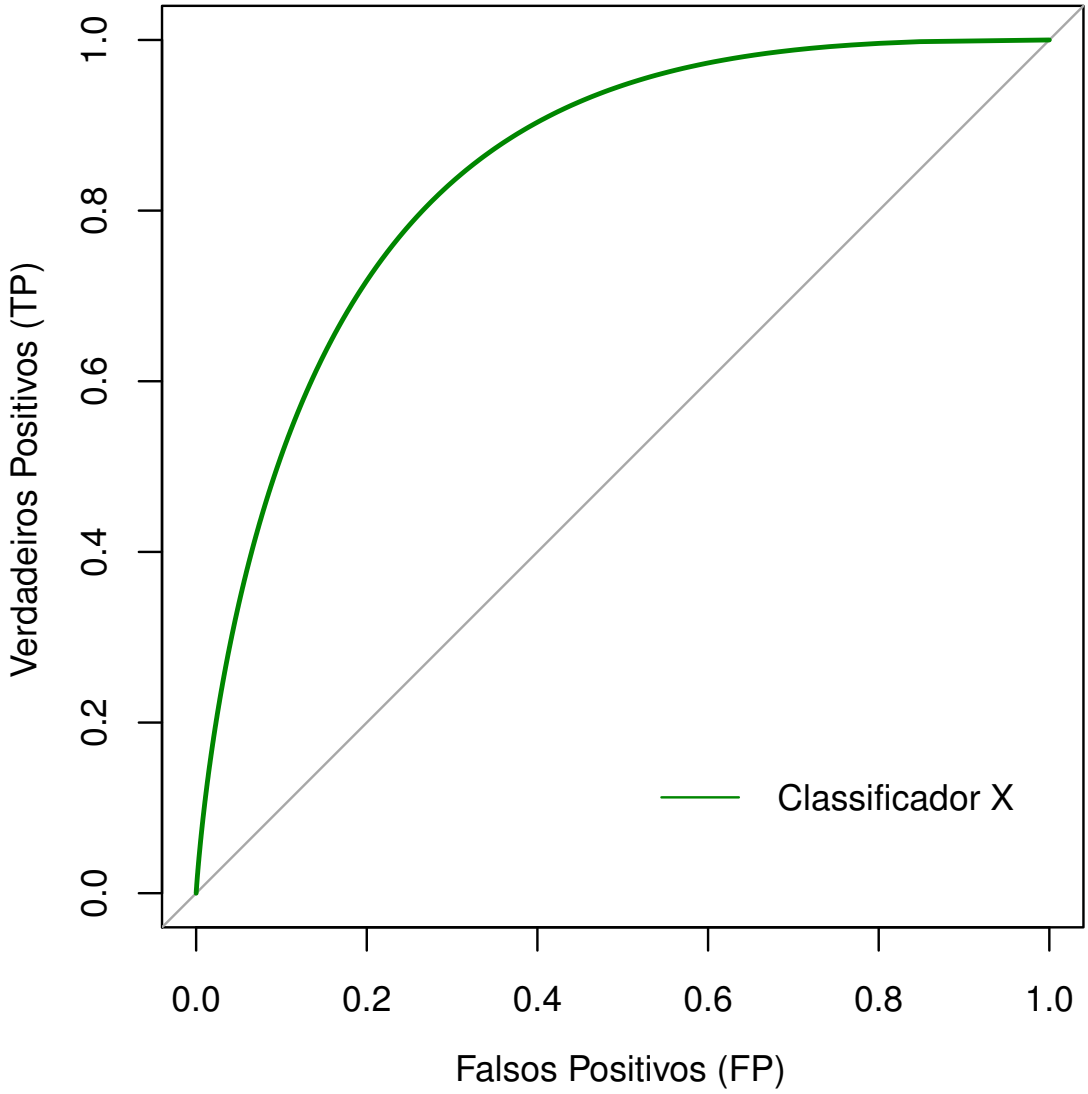

Figura 12 – Exemplo de Curva ROC para um classificador X

5.6 Avaliação

As ferramentas analisadas retornam um valor de similaridade no formato de porcentagem. Este valor foi dividido por 100 para termos um valor entre 0 e 1 e com no máximo duas casas decimais. Caso, por exemplo, uma ferramenta apresente o valor de 36%, o valor convertido será de 0,36.

A Tabela 1 mostra um exemplo de como foram interpretados os resultados das ferramentas. A primeira coluna mostra o valor de similaridade encontrado pelo classificador, a segunda coluna apresenta o limiar discriminatório e, por fim, a terceira representa o resultado da classificação. No exemplo da tabela, o classificador apresentou a similaridade de 56% para um par de códigos-fonte. Variamos o limiar discriminatório de 0 até 1 com intervalos de 0,1. Logo, temos resultados positivos (1) para os valores com limiar discriminatório menor ou igual a 0,56 e os maiores resultados negativos (0).

Programas Comparados	**Similaridade**	**Limiar discriminatório**	**Saída**
program1.c e program2.c	0,56	0,0	1
program1.c e program2.c	0,56	0,1	1
program1.c e program2.c	0,56	0,2	1
program1.c e program2.c	0,56	0,3	1
program1.c e program2.c	0,56	0,4	1
program1.c e program2.c	0,56	0,5	1
program1.c e program2.c	0,56	0,6	0
program1.c e program2.c	0,56	0,7	0
program1.c e program2.c	0,56	0,8	0
program1.c e program2.c	0,56	0,9	0
program1.c e program2.c	0,56	1,0	0

Tabela 1 – Exemplo de saída utilizando limiar discriminatório

A saída apresentada pela Tabela 1 é comparada com a análise manual do docente. A análise do docente também está em forma binária, em caso positivo é utilizado o valor 1 e em caso negativo o valor 0. Caso a análise do docente tenha o mesmo valor da saída apresentada na tabela, teremos um acerto, senão, um erro.

A eficácia de cada ferramenta foi analisada através de duas medidas. A primeira é chamada de área abaixo da curva, também conhecida como AUC (HANLEY; MCNEIL, 1982; BRADLEY, 1997) (do inglês *Area Under Curve*). Este é um dos métodos mais comuns para comparar dois classificadores (FAWCETT, 2006). O valor da área pode ir de 0 até 1 e quanto maior o valor, melhor a eficácia do classificador. Essa medida só pode ser utilizada em classificadores que utilizam limiar discriminatório. Os cálculos das AUCs foram realizados pela ferramenta R versão 3.0.3, assim como a plotagem das curvas ROC.

A segunda é a menor distância Euclidiana entre um ponto da curva ao ponto (0, 1) do espaço ROC. Quanto menor a distância, melhor o desempenho do classificador. Essa medida, diferente da primeira, também pode ser utilizada em classificadores binários,

aqueles que não utilizam limiar discriminatório e geram apenas um ponto no espaço ROC. As distâncias foram calculadas utilizando o R versão 3.0.3.

6 RESULTADOS

Nesta seção serão apresentados os resultados do experimento descrito no capítulo anterior. Foi feita uma análise quantitativa baseada nos elementos de avaliação descritos em 5.5.

6.1 Sumário dos Dados

A Tabela 2 apresenta a quantidade de submissões e comparações para cada um dos problemas selecionados. Ao todo foram realizadas 818 comparações, isso também significa que o docente avaliou manualmente 818 pares de códigos-fonte.

A maioria dos problemas seguiu a tendência de que quanto maior o nível de dificuldade, menor o número de submissões. Alguns casos como, por exemplo, o problema *Ambrosio's Bus Company* teve um baixo número de submissões mesmo com o nível de dificuldade baixo. Sugerimos que tal comportamento pode ser resultado da evasão da disciplina e também ao alto aproveitamento de alguns alunos. Os alunos com alto aproveitamento nem sempre fazem provas de reavaliação, visto que conseguem pontos suficientes em duas avaliações.

Problema	**Submissões**	**Comparações**	**Nível de Dificuldade**
Ambrosio's Bus Company	6	15	2
Características Físicas	11	55	3
Colchão	25	300	3
Festa da Marilda	15	105	3
Foto de Família	15	105	2
Futebol	3	3	8
Lanchonete da Ambrosina	10	45	5
Letra Maiúscula	11	55	6
Lua Cheia	6	15	6
Permutador de Trem	16	120	6

Tabela 2 – Sumário dos dados

6.2 Matrizes de Confusão

As matrizes de confusão estão apresentadas no Apêndice B. Foram criadas ao todo 30 tabelas sendo 10 para cada ferramenta. A partir delas foi possível calcular os valores da taxa de falso positivo (Equação 5.1) e da taxa de verdadeiro positivo (Equação 5.2) para a geração das curvas ROC.

Analisando as tabelas, percebe-se que em nenhum momento o JPlag conseguiu detectar todos os plágios. Em todos os testes, a ferramenta deixou de apontar plágio em pelo

menos 10 códigos-fonte. A ferramenta PRIDE conseguiu acertar todos os plágios para os limiares discriminatórios 10%, 20%, 30%, 40%, 50% e 60%. A ferramenta MOSS atingiu uma marca bem próxima a do PRIDE, deixando de classificar apenas um caso de plágio nos mesmos limiares discriminatórios.

A ferramenta PRIDE, no entanto, apresentou um número superior de falsos positivos em todos os limiares discriminatórios tanto em relação ao JPlag e ao MOSS. Isso indica que a ferramenta PRIDE é a mais liberal das três.

O valor 0 obtido na taxa de verdadeiro positivo para o limiar discriminatório 1 (100%) da ferramenta MOSS, pode sugerir que a modificação de nomes de variáveis e funções interfere no cálculo de similaridade. A mudança nos nomes das variáveis e/ou funções é uma das técnicas de disfarçe mais simples de ser aplicada e considerando que algum aluno utilizou apenas essa técnica de disfarçe, a ferramenta MOSS deveria ter indicado algum par de código-fonte com similaridade de 100%. Esta possibilidade é sugerida, pois não temos como afirmar que há um par de códigos-fonte que utilizou somente a técnica de mudança nos nomes das variáveis e/ou funções.

6.3 Curvas ROC

As Figuras 13, 14 e 15 apresentam as curvas ROC das ferramentas PRIDE, JPlag e MOSS respectivamente. As figuras também mostram os valores da AUC de cada uma - 0,976 (PRIDE), 0,932 (JPlag) e 0,943 (MOSS).

Em se tratando do valor da área sob a curva, o PRIDE obteve melhor resultado, sendo seguido pela ferramenta MOSS e logo após o JPlag. Em quase toda a região da curva o PRIDE apresenta um melhor resultado que as outras duas ferramentas; já a ferramenta MOSS apresenta uma melhor eficácia em relação ao JPlag. É possivel afirmar também que as três ferramentas obtiveram bom desempenho absoluto, visto que o valor máximo que pode ser obtido na AUC é 1.

A comparação entre as curvas está apresentada na Figura 16 e os pontos de menor distância para o ponto (0, 1) marcados com símbolos. A distância mínima obtida foi de 0.0964048 (PRIDE), 0.2051104 (JPlag) e 0.1873819 (MOSS). Na segunda medida não houve mudanças em relação à primeira, o resultado corroborou com o resultado da primeira medida.

6.4 Ameaças

Durante a execução do experimento observamos algumas ameaças à sua validade. A seguir descrevemos cada um destes.

- Oráculo com apenas um docente: A avaliação manual foi criada com apenas um docente. Foram feitos alguns convites informais, no entanto, apenas um docente aceitou participar o experimento.

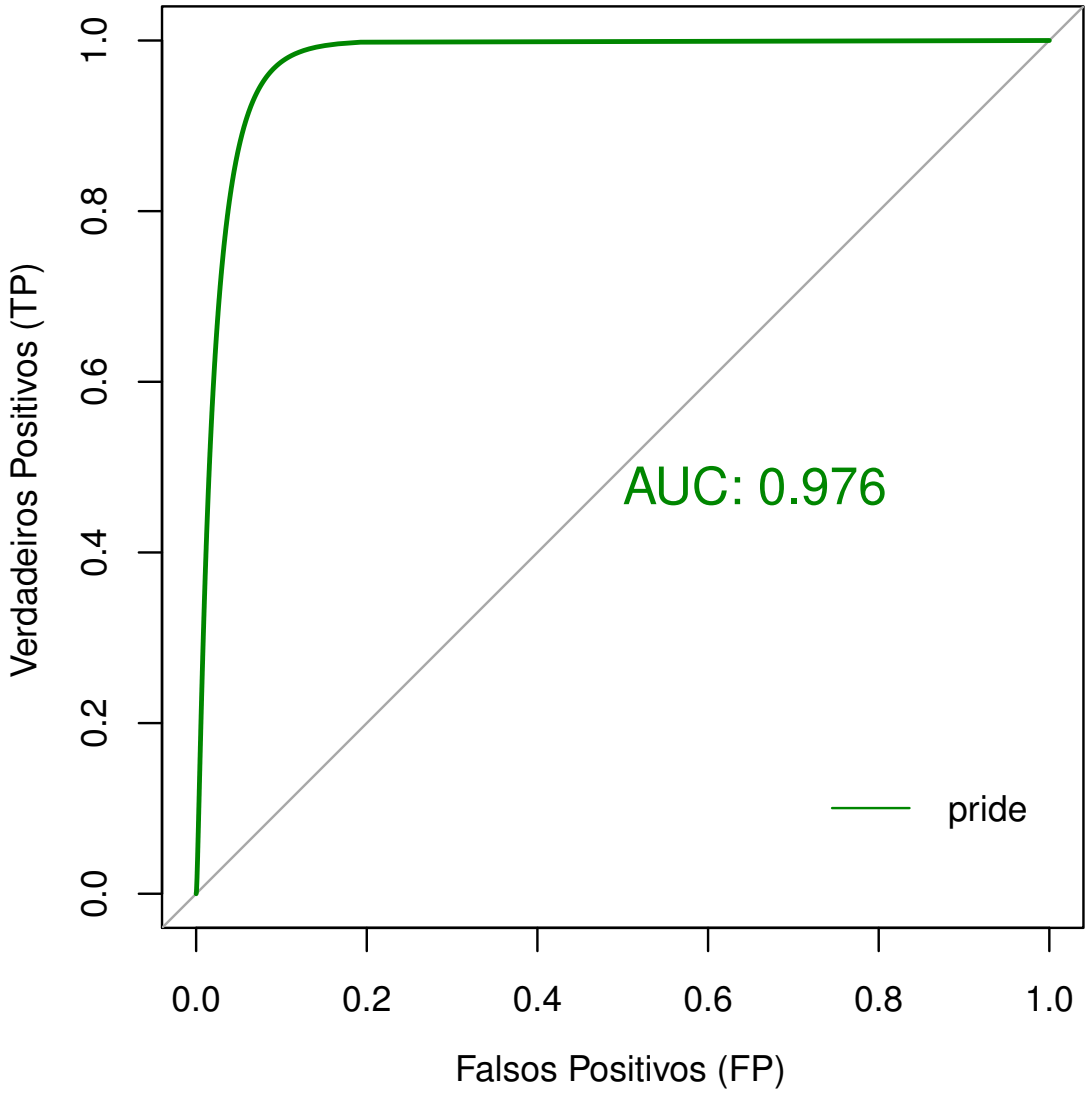

Figura 13 – Curva ROC da ferramenta PRIDE

- O docente é orientador do trabalho: O orientador deste trabalho foi o único docente que se dispôs a avaliar os códigos-fonte. Isso trouxe um grande risco, visto que o trabalho foi discutido unicamente com o orientador. É possível que, inconscientemente, o docente tenha procurado plágio com base nas técnicas discutidas durante a construção da ferramenta. Uma forma de mitigar este risco seria aumentando a quantidade de avaliadores. Infelizmente, como foi informado no item anterior, não foi possível aumentar esse número.

- Os problemas escolhidos podem não ser representativos: O sistema The Huxley (TEAM, 2013) possui mais de duzentas mil submissões em mais de quatrocentos problemas com diversos níveis de dificuldade. Estes problemas abrangem desde o primeiro

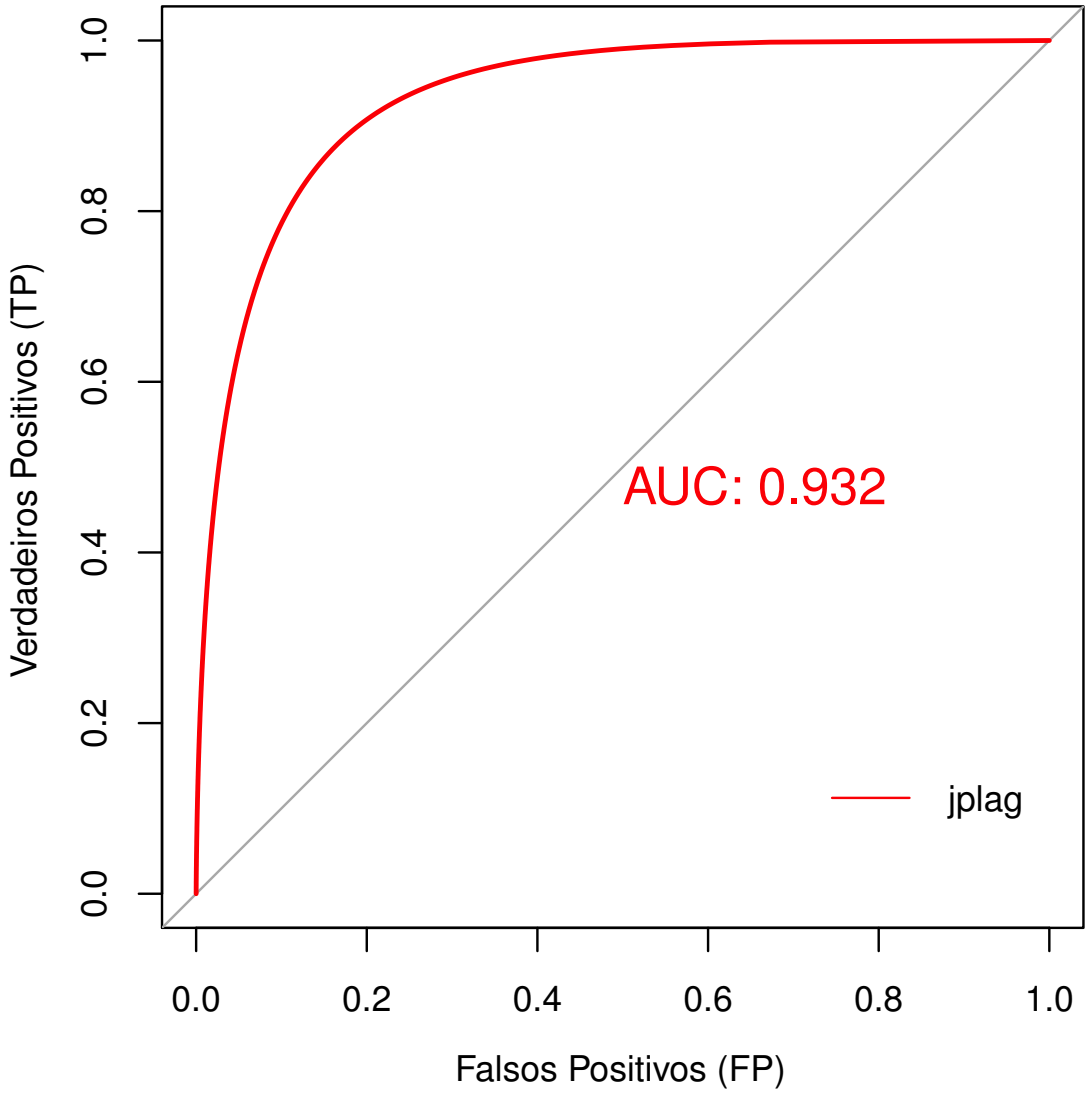

Figura 14 – Curva ROC da ferramenta JPLAG

programa criado por um iniciante até os de grande dificuldade, como problemas de maratona de programação. Infelizmente, devido ao grande número de verificações manuais a serem feitas pelo docente, não foi possível conseguir aumentar o número de problemas utilizados no experimento. Descartamos problemas com nível de dificuldade muito baixo, pois estes possuem uma curta e simples resolução. Isso iria provavelmente acarretar em uma alta similaridade para todos as submissões.

- O estudo de Lancaster e Culwin (2004) pode ter deixado alguma ferramenta importante de lado: O estudo observou apenas o caráter qualitativo de onze ferramentas. Não houve comparação de resultados. .

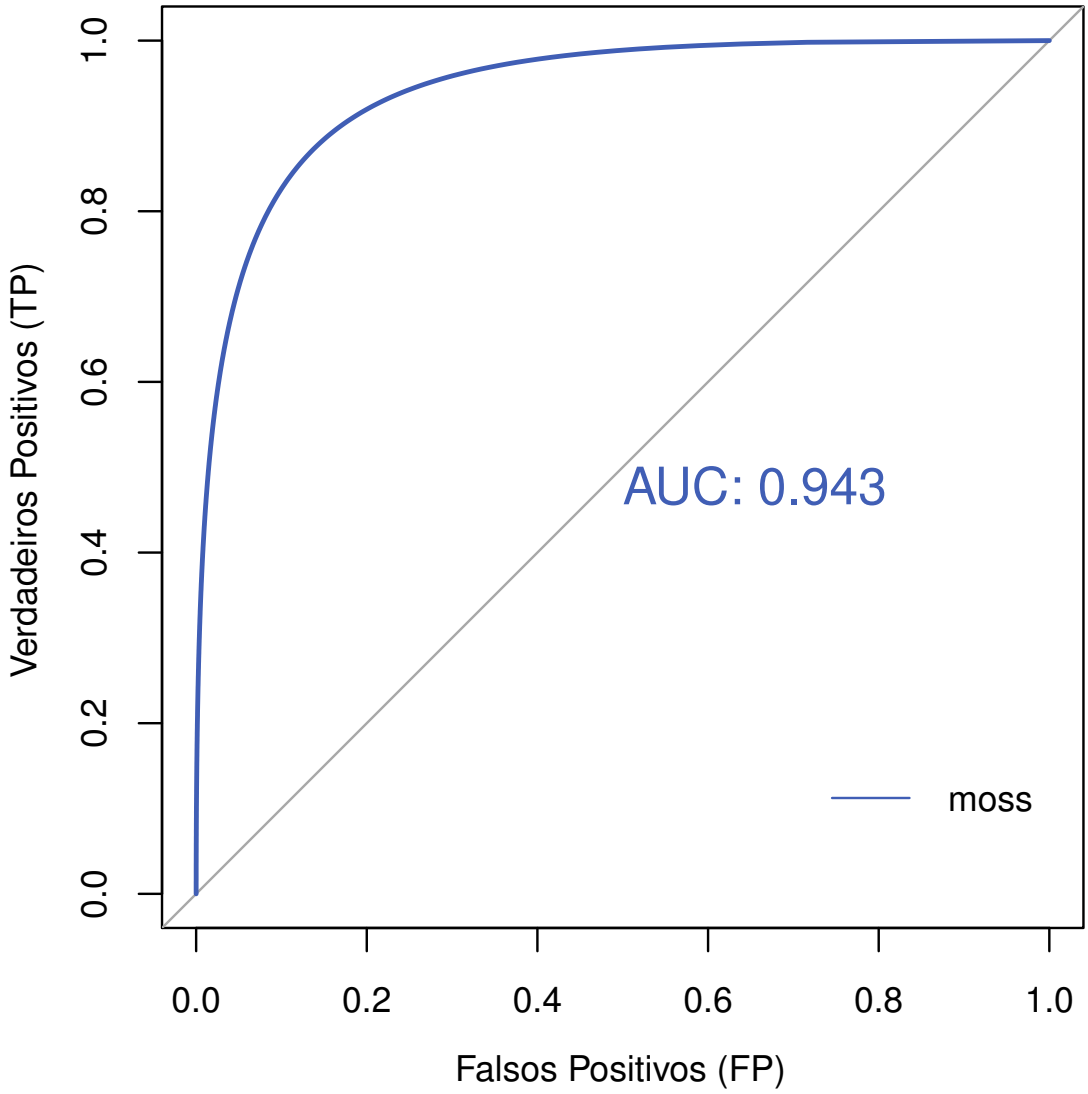

Figura 15 – Curva ROC da ferramenta MOSS

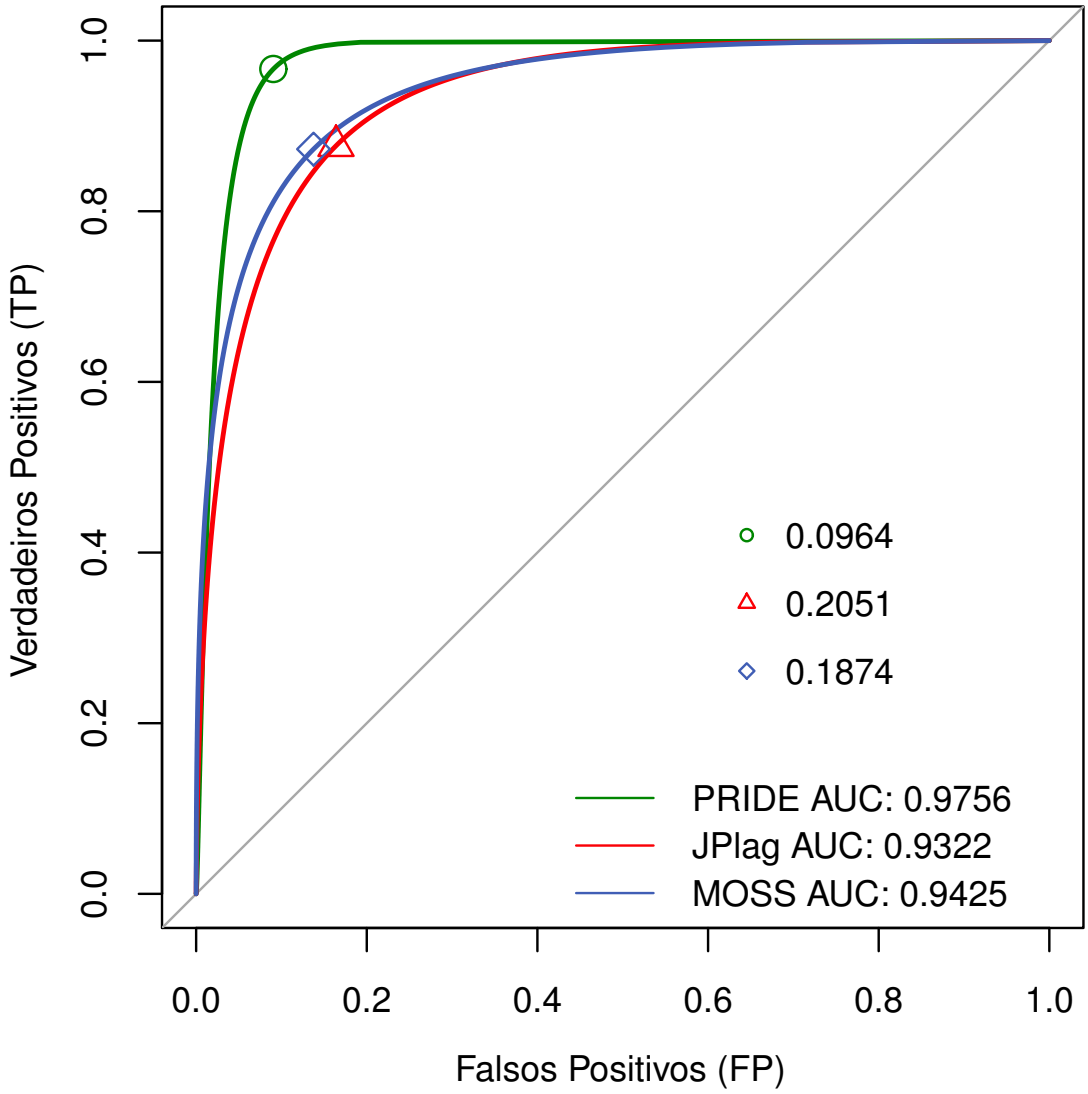

Figura 16 – Comparação entre as curvas geradas pelas ferramentas

7 CONCLUSÕES E TRABALHOS FUTUROS

Apresentamos neste trabalho a ferramenta PRIDE. Trata-se de um sistema computacional capaz de detectar similaridade em código-fonte. Foi definido um experimento para respondermos a seguinte pergunta:

- A ferramenta apresenta melhor resultado que as outras ferramentas analisadas?

Utilizamos duas medidas para determinar qual ferramenta possui melhor resultado para os códigos-fonte coletados. A primeira medida utilizada foi a AUC que possui valor mínimo de 0 e valor máximo de 1, sendo 1 o melhor resultado. Os valores obtidos foram 0,976 (PRIDE), 0,932 (JPlag) e 0,943 (MOSS). A segunda medida foi a menor distância para o ponto (0, 1) do espaço ROC. Os valores encontrados foram 0.0964048 (PRIDE), 0.2051104 (JPlag) e 0.1873819 (MOSS). Os dois métodos apresentaram o mesmo resultado o que sugere que, para os códigos-fonte coletados, a ferramenta PRIDE apresenta melhor resultado que as outras ferramentas analisadas.

Para o caso do JPlag, não podemos garantir que a implementação utilizada ainda segue a descrição do artigo publicado em 2001, visto que seu código-fonte é fechado. É possível supor que o sistema teve atualizações, devido a algumas publicações em seu site (<http://jplag.ipd.kit.edu/>).

Assim como o JPlag, o MOSS também tem código-fonte fechado. Em seu site (<http://theory.stanford.edu/~aiken/moss/>), podemos verificar que há contribuições em scripts para facilitar a submissão nos arquivos. Não há informações sobre atualizações no sistema.

Os três sistemas apresentaram bons resultados e a escolha da ferramenta dependerá do perfil conservador ou liberal do utilizador da ferramenta. Outro fator que será determinante é a linguagem de programação utilizada pelo docente. O PRIDE, neste primeiro momento, só realiza o processo de tokenização para a linguagem C, para as demais, o pré-processamento é resumido na remoção de espaços em branco. Esse último fator determinante citado é o principal trabalho futuro a ser realizado para o PRIDE, visto que nos dias de hoje temos um vasto número de linguagens de programação que podem ser utilizadas no aprendizado de programação. Ainda como trabalho futuro, serão aplicadas técnicas de clusterização de dados na tentativa de reduzir o número de comparações realizadas.

Por fim, destacamos as contribuições deste trabalho: estratégia de *tokenização* para potencializar a detecção, implementação do algoritmo *Karp-Rabin Greedy String Tiling*, apoio computacional assíncrono (PRIDE) e definição de um processo de detecção de similaridade.

REFERÊNCIAS

AHMADZADEH, M.; MAHMOUDABADI, E.; KHODADADI, F. Pattern of plagiarism in novice students' generated programs: An experimental approach. *Journal of Information Technology Education*, v. 10, 2011. ISSN 1547–9714.

BERGHEL, H. L.; SALLACH, D. L. Measurements of program similarity in identical task environments. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 19, n. 8, p. 65–76, Aug 1984. ISSN 0362-1340.

BIRNBAUM, D.; GOSCILO, H. *Avoiding Plagiarism*. 2001. Http://clover.slavic.pitt.edu/ tales/02-1/plagiarism.html. [Online; acesso em 19-Julho-2014].

BRADLEY, A. P. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern Recognition*, v. 30, p. 1145–1159, 1997.

CHEN, X. et al. Shared information and program plagiarism detection. *Information Theory, IEEE Transactions on*, v. 50, n. 7, p. 1545–1551, July 2004. ISSN 0018-9448.

COSMA, G.; JOY, M. *Source-code Plagiarism: a UK Academic Perspective*. Coventry, 2006. Disponível em: <http://eprints.dcs.warwick.ac.uk/52/>.

DESRUISSEAUX, P. Cheating is reaching epidemic proportions worldwide, researchers say. *Chronicle of Higher Education*, April 1999.

DONALDSON, J. L.; LANCASTER, A.-M.; SPOSATO, P. H. A plagiarism detection system. *SIGCSE Bull.*, ACM, New York, NY, USA, v. 13, n. 1, p. 21–25, Feb 1981. ISSN 0097-8418. Disponível em: <http://doi.acm.org/10.1145/953049.800955>.

DRAKE, C. A. Why students cheat. *Journal of Higher Education*, v. 12, p. 418—-420, 1941.

EGAN, J. P. *Signal Detection Theory and ROC Analysis*. [S.l.]: Academic Press, 1975. (Series in Cognition and Perception).

FAWCETT, T. An introduction to roc analysis. *Pattern Recognition Letters*, v. 27, n. 8, p. 861–874, JUN 2006. ISSN 0167-8655.

GITCHELL, D.; TRAN, N. Sim: A utility for detecting similarity in computer programs. *SIGCSE Bull.*, ACM, New York, NY, USA, v. 31, n. 1, p. 266–270, Mar 1999. ISSN 0097-8418. Disponível em: <http://doi.acm.org/10.1145/384266.299783>.

GRIER, S. A tool that detects plagiarism in pascal programs. *SIGCSE Bull.*, ACM, New York, NY, USA, v. 13, n. 1, p. 15–20, Feb 1981. ISSN 0097-8418. Disponível em: <http://doi.acm.org/10.1145/953049.800954>.

GUSFIELD, D. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. New York, NY, USA: Cambridge University Press, 1997. ISBN 0-521-58519-8.

HALSTEAD, M. H. *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Inc., 1977. ISBN 0444002057.

HANLEY, J.; MCNEIL, B. The meaning and use of the area under a receiver operating characteristic (roc) curve. *Radiology*, Radiological Society of North America, v. 143, n. 1, p. 29–36, 1982. ISSN 0033-8419.

HECKEL, P. A technique for isolating differences between files. *Commun. ACM*, ACM, New York, NY, USA, v. 21, n. 4, p. 264–268, Apr 1978. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/359460.359467>.

JENSEN, L. A. et al. It's wrong, but everybody does it: Academic dishonesty among high school and college students. *Contemporary Educational Psychology*, v. 27, p. 209–228, 2002.

JI, J.-H.; WOO, G.; CHO, H.-G. A source code linearization technique for detecting plagiarized programs. *SIGCSE Bull.*, ACM, New York, NY, USA, v. 39, n. 3, p. 73–77, Jun 2007. ISSN 0097-8418. Disponível em: <http://doi.acm.org/10.1145/1269900.1268807>.

JOY, M.; LUCK, M. *Plagiarism in Programming Assignments*. Coventry, 1998. Disponível em: <http://eprints.dcs.warwick.ac.uk/346/>.

KARP, R. M.; RABIN, M. O. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, IBM Corp., Riverton, NJ, USA, v. 31, n. 2, p. 249–260, mar. 1987. ISSN 0018-8646. Disponível em: <http://dx.doi.org/10.1147/rd.312.0249>.

KLEIMAN, A. *Análise e comparação qualitativa de sistemas de detecção de plágio em tarefas de programação*. Dissertação (Mestrado) — Universidade Estadual de Campinas, 2007.

LANCASTER, T.; CULWIN, F. A comparison of source code plagiarism detection engines. *Computer Science Education*, v. 14, n. 2, p. 101–112, 2004. Disponível em: <10.1080/08993400412331363843>.

LIU, C. et al. Gplag: Detection of software plagiarism by program dependence graph analysis. In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA: ACM, 2006. (KDD '06), p. 872–881. ISBN 1-59593-339-5. Disponível em: <http://doi.acm.org/10.1145/1150402.1150522>.

MOZGOVOY, M. *Enhancing Computer-Aided Plagiarism Detection*. Saarbrücken, Germany, Germany: VDM Verlag, 2008. ISBN 3639097246, 9783639097245.

OHMANN, A. *Efficient Clustering-based Plagiarism Detection using IPPDC*. Dissertação (Mestrado) — College of Saint Benedict and Saint John's University, 2013.

OTTENSTEIN, K. J. An algorithmic approach to the detection and prevention of plagiarism. *SIGCSE Bull.*, ACM, New York, NY, USA, v. 8, n. 4, p. 30–41, dez. 1976. ISSN 0097-8418. Disponível em: <http://doi.acm.org/10.1145/382222.382462>.

PARR, T. *ANTLR Documentation*. 2013. Https://theantlrguy.atlassian.net/wiki/download/attachments/1900546/process.png. Accessado em: 24/02/2015.

PIKE, R. *The Sherlock Plagiarism Detector*. 2012. Http://sydney.edu.au/engineering/it/ scilect/sherlock/. Acesso em 20 de abril de 2014.

PRECHELT, L.; MALPOHL, G.; PHILIPPSEN, M. Finding plagiarisms among a set of programs with jplag. *Journal of Universal Computer Science*, v. 8, p. 1016–1038, 2001.

RAMASESHA, S. Combating plagiarism in scientific research. *CURRENT SCIENCE*, INDIAN ACAD SCIENCES, v. 107, n. 1, p. 11, JUL 2014. ISSN 0011-3891.

RAMBALLY, G.; SAGE, M. L. An inductive inference approach to plagiarism detection in computerprograms. In: *Proceedings of the National Educational Computing Conference*. [S.l.: s.n.], 1990. p. 23–29.

ROBINSON, S. S.; SOFFA, M. L. An instructional aid for student programs. *SIGCSE Bull.*, ACM, New York, NY, USA, v. 12, n. 1, p. 118–129, feb 1980. ISSN 0097-8418. Disponível em: <http://doi.acm.org/10.1145/953032.804623>.

SCHLEIMER, S.; WILKERSON, D. S.; AIKEN, A. Winnowing: Local algorithms for document fingerprinting. In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2003. (SIGMOD '03), p. 76–85. ISBN 1-58113-634-X. Disponível em: <http://doi.acm.org/10.1145/872757.872770>.

TEAM, T. H. *The Huxley*. 2013. Http://www.thehuxley.com. Accessado em: 03/01/2015.

VERCO, K. L.; WISE, M. J. Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems. In: *Proceedings of the 1st Australasian Conference on Computer Science Education*. New York, NY, USA: ACM, 1996. (ACSE '96), p. 81–88. ISBN 0-89791-845-2. Disponível em: <http://doi.acm.org/10.1145/369585.369598>.

WHALE, G. Detection of plagiarism in student programs. *Proceedings of the Ninth Australian Computer Science Conference*, 1986.

WHALE, G. *Plague: plagiarism detection using program structure*. [S.l.], 1988.

WHALE, G. Identification of program similarity in large populations. *The Computer Journal*, v. 33, p. 140–146, Oct 1990.

WISE, M. J. *Running Karp-Rabin Matching and Greedy String Tiling*. [S.l.], 1993.

WISE, M. J. Yap3: Improved detection of similarities in computer program and other texts. In: *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education*. [S.l.]: ACM Press, 1996. p. 130–134.

APÊNDICE A – GRAMÁTICA DA LINGUAGEM C (ANTLR 4)

Gramática C utilizada no PRIDE em formato ANTLR 4. A original pode ser encontrada em <https://github.com/antlr/grammars-v4>.

C.g4

```
/*
 [The "BSD licence"]
 Copyright (c) 2013 Sam Harwell
 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions
 are met:
 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.
 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.
 3. The name of the author may not be used to endorse or promote products
    derived from this software without specific prior written permission.

 THIS SOFTWARE IS PROVIDED BY THE AUTHOR ‘‘AS IS’’ AND ANY EXPRESS OR
 IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
 IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
 INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
 NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
 THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/

/** C 2011 grammar built from the C11 Spec */
grammar C;

/*
@header {
package com.huxley.plagiarism.tokenizer.c;
}
*/
primaryExpression
    :   Identifier
    |   Constant
```

```
    |   StringLiteral+
    |   LeftParen expression RightParen
    |   genericSelection
    |   '__extension__'? LeftParen compoundStatement RightParen // Blocks (GCC extension)
    |   '__builtin_va_arg' LeftParen unaryExpression Comma typeName RightParen
    |   '__builtin_offsetof' LeftParen typeName Comma unaryExpression RightParen
    ;

genericSelection
    :   Generic LeftParen assignmentExpression Comma genericAssocList RightParen
    ;

genericAssocList
    :   genericAssociation
    |   genericAssocList Comma genericAssociation
    ;

genericAssociation
    :   typeName Colon assignmentExpression
    |   Default Colon assignmentExpression
    ;

postfixExpression
    :   primaryExpression
    |   postfixExpression LeftBracket expression RightBracket
    |   postfixExpression LeftParen argumentExpressionList? RightParen
    |   postfixExpression Dot Identifier
    |   postfixExpression Arrow Identifier
    |   postfixExpression PlusPlus
    |   postfixExpression MinusMinus
    |   LeftParen typeName RightParen LeftBrace initializerList RightBrace
    |   LeftParen typeName RightParen LeftBrace initializerList Comma RightBrace
    |   '__extension__' LeftParen typeName RightParen LeftBrace initializerList RightBrace
    |   '__extension__' LeftParen typeName RightParen LeftBrace initializerList Comma RightBrace
    ;

argumentExpressionList
    :   assignmentExpression
    |   argumentExpressionList Comma assignmentExpression
    ;

unaryExpression
    :   postfixExpression
    |   PlusPlus unaryExpression
    |   MinusMinus unaryExpression
    |   unaryOperator castExpression
    |   Sizeof unaryExpression
```

```
    |   Sizeof LeftParen typeName RightParen
    |   Alignof LeftParen typeName RightParen
    |   AndAnd Identifier // GCC extension address of label
    ;

unaryOperator
    :   And | Star | Plus | Minus | Tilde | Not
    ;

castExpression
    :   unaryExpression
    |   LeftParen typeName RightParen castExpression
    |   '__extension__' LeftParen typeName RightParen castExpression
    ;

multiplicativeExpression
    :   castExpression
    |   multiplicativeExpression Star castExpression
    |   multiplicativeExpression Div castExpression
    |   multiplicativeExpression Mod castExpression
    ;

additiveExpression
    :   multiplicativeExpression
    |   additiveExpression Plus multiplicativeExpression
    |   additiveExpression Minus multiplicativeExpression
    ;

shiftExpression
    :   additiveExpression
    |   shiftExpression LeftShift additiveExpression
    |   shiftExpression RightShift additiveExpression
    ;

relationalExpression
    :   shiftExpression
    |   relationalExpression Less shiftExpression
    |   relationalExpression Greater shiftExpression
    |   relationalExpression LessEqual shiftExpression
    |   relationalExpression GreaterEqual shiftExpression
    ;

equalityExpression
    :   relationalExpression
    |   equalityExpression Equal relationalExpression
    |   equalityExpression NotEqual relationalExpression
    ;
```

```
andExpression
    :   equalityExpression
    |   andExpression And equalityExpression
    ;

exclusiveOrExpression
    :   andExpression
    |   exclusiveOrExpression Caret andExpression
    ;

inclusiveOrExpression
    :   exclusiveOrExpression
    |   inclusiveOrExpression Or exclusiveOrExpression
    ;

logicalAndExpression
    :   inclusiveOrExpression
    |   logicalAndExpression AndAnd inclusiveOrExpression
    ;

logicalOrExpression
    :   logicalAndExpression
    |   logicalOrExpression OrOr logicalAndExpression
    ;

conditionalExpression
    :   logicalOrExpression (Question expression Colon conditionalExpression)?
    ;

assignmentExpression
    :   conditionalExpression
    |   unaryExpression assignmentOperator assignmentExpression
    ;

assignmentOperator
    :   Assign | StarAssign | DivAssign | ModAssign | PlusAssign | MinusAssign | LeftShiftAssign |
    ;

expression
    :   assignmentExpression
    |   expression Comma assignmentExpression
    ;

constantExpression
    :   conditionalExpression
    ;
```

```
declaration
    :   declarationSpecifiers initDeclaratorList? Semi
    |   staticAssertDeclaration
    ;

declarationSpecifiers
    :   declarationSpecifier+
    ;

declarationSpecifiers2
    :   declarationSpecifier+
    ;

declarationSpecifier
    :   storageClassSpecifier
    |   typeSpecifier
    |   typeQualifier
    |   functionSpecifier
    |   alignmentSpecifier
    ;

initDeclaratorList
    :   initDeclarator
    |   initDeclaratorList Comma initDeclarator
    ;

initDeclarator
    :   declarator
    |   declarator Assign initializer
    ;

storageClassSpecifier
    :   Typedef
    |   Extern
    |   Static
    |   ThreadLocal
    |   Auto
    |   Register
    ;

typeSpecifier
    :   (Void
    |   Char
    |   Short
    |   Int
    |   Long
```

```
    |   Float
    |   Double
    |   Signed
    |   Unsigned
    |   Bool
    |   Complex
    |   '__m128'
    |   '__m128d'
    |   '__m128i')
    |   '__extension__' LeftParen ('__m128' | '__m128d' | '__m128i') RightParen
    |   atomicTypeSpecifier
    |   structOrUnionSpecifier
    |   enumSpecifier
    |   typedefName
    |   '__typeof__' LeftParen constantExpression RightParen // GCC extension
    ;

structOrUnionSpecifier
    :   structOrUnion Identifier? LeftBrace structDeclarationList RightBrace
    |   structOrUnion Identifier
    ;

structOrUnion
    :   Struct
    |   Union
    ;

structDeclarationList
    :   structDeclaration
    |   structDeclarationList structDeclaration
    ;

structDeclaration
    :   specifierQualifierList structDeclaratorList? Semi
    |   staticAssertDeclaration
    ;

specifierQualifierList
    :   typeSpecifier specifierQualifierList?
    |   typeQualifier specifierQualifierList?
    ;

structDeclaratorList
    :   structDeclarator
    |   structDeclaratorList Comma structDeclarator
    ;
```

```
structDeclarator
    :   declarator
    |   declarator? Colon constantExpression
    ;

enumSpecifier
    :   Enum Identifier? LeftBrace enumeratorList RightBrace
    |   Enum Identifier? LeftBrace enumeratorList Comma RightBrace
    |   Enum Identifier
    ;

enumeratorList
    :   enumerator
    |   enumeratorList Comma enumerator
    ;

enumerator
    :   enumerationConstant
    |   enumerationConstant Assign constantExpression
    ;

enumerationConstant
    :   Identifier
    ;

atomicTypeSpecifier
    :   Atomic LeftParen typeName RightParen
    ;

typeQualifier
    :   Const
    |   Restrict
    |   Volatile
    |   Atomic
    ;

functionSpecifier
    :   (Inline
    |   Noreturn
    |   '__inline__' // GCC extension
    |   '__stdcall')
    |   gccAttributeSpecifier
    |   '__declspec' LeftParen Identifier RightParen
    ;

alignmentSpecifier
    :   Alignas LeftParen typeName RightParen
```

```
    |   Alignas LeftParen constantExpression RightParen
    ;

declarator
    :   pointer? directDeclarator gccDeclaratorExtension*
    ;

directDeclarator
    :   Identifier
    |   LeftParen declarator RightParen
    |   directDeclarator LeftBracket typeQualifierList? assignmentExpression? RightBracket
    |   directDeclarator LeftBracket Static typeQualifierList? assignmentExpression RightBracket
    |   directDeclarator LeftBracket typeQualifierList Static assignmentExpression RightBracket
    |   directDeclarator LeftBracket typeQualifierList? Star RightBracket
    |   directDeclarator LeftParen parameterTypeList RightParen
    |   directDeclarator LeftParen identifierList? RightParen
    ;

gccDeclaratorExtension
    :   '__asm' LeftParen StringLiteral+ RightParen
    |   gccAttributeSpecifier
    ;

gccAttributeSpecifier
    :   '__attribute__' LeftParen LeftParen gccAttributeList RightParen RightParen
    ;

gccAttributeList
    :   gccAttribute (Comma gccAttribute)*
    |   // empty
    ;

gccAttribute
    :   ~(Comma | LeftParen | RightParen) // relaxed def for "identifier or reserved word"
        (LeftParen argumentExpressionList? RightParen)?
    |   // empty
    ;

nestedParenthesesBlock
    :   (   ~(LeftParen | RightParen)
        |   LeftParen nestedParenthesesBlock RightParen
        )*
    ;

pointer
    :   Star typeQualifierList?
    |   Star typeQualifierList? pointer
```

```
    |   Caret typeQualifierList? // Blocks language extension
    |   Caret typeQualifierList? pointer // Blocks language extension
    ;

typeQualifierList
    :   typeQualifier
    |   typeQualifierList typeQualifier
    ;

parameterTypeList
    :   parameterList
    |   parameterList Comma Ellipsis
    ;

parameterList
    :   parameterDeclaration
    |   parameterList Comma parameterDeclaration
    ;

parameterDeclaration
    :   declarationSpecifiers declarator
    |   declarationSpecifiers2 abstractDeclarator?
    ;

identifierList
    :   Identifier
    |   identifierList Comma Identifier
    ;

typeName
    :   specifierQualifierList abstractDeclarator?
    ;

abstractDeclarator
    :   pointer
    |   pointer? directAbstractDeclarator gccDeclaratorExtension*
    ;

directAbstractDeclarator
    :   LeftParen abstractDeclarator RightParen gccDeclaratorExtension*
    |   LeftBracket typeQualifierList? assignmentExpression? RightBracket
    |   LeftBracket Static typeQualifierList? assignmentExpression RightBracket
    |   LeftBracket typeQualifierList Static assignmentExpression RightBracket
    |   LeftBracket Star RightBracket
    |   LeftParen parameterTypeList? RightParen gccDeclaratorExtension*
    |   directAbstractDeclarator LeftBracket typeQualifierList? assignmentExpression? RightBracket
    |   directAbstractDeclarator LeftBracket Static typeQualifierList? assignmentExpression RightB
```

```
    |   directAbstractDeclarator LeftBracket typeQualifierList Static assignmentExpression RightBr
    |   directAbstractDeclarator LeftBracket Star RightBracket
    |   directAbstractDeclarator LeftParen parameterTypeList? RightParen gccDeclaratorExtension*
    ;

typedefName
    :   Identifier
    ;

initializer
    :   assignmentExpression
    |   LeftBrace initializerList RightBrace
    |   LeftBrace initializerList Comma RightBrace
    ;

initializerList
    :   designation? initializer
    |   initializerList Comma designation? initializer
    ;

designation
    :   designatorList Assign
    ;

designatorList
    :   designator
    |   designatorList designator
    ;

designator
    :   LeftBracket constantExpression RightBracket
    |   Dot Identifier
    ;

staticAssertDeclaration
    :   '_Static_assert' LeftParen constantExpression Comma StringLiteral+ RightParen Semi
    ;

statement
    :   labeledStatement
    |   compoundStatement
    |   expressionStatement
    |   selectionStatement
    |   iterationStatement
    |   jumpStatement
    |   ('__asm' | '__asm__') (Volatile | '__volatile__') LeftParen (logicalOrExpression (Comma lo
    ;
```

```
labeledStatement
    :   Identifier Colon statement
    |   Case constantExpression Colon statement
    |   Default Colon statement
    ;

compoundStatement
    :   LeftBrace blockItemList? RightBrace
    ;

blockItemList
    :   blockItem
    |   blockItemList blockItem
    ;

blockItem
    :   declaration
    |   statement
    ;

expressionStatement
    :   expression? Semi
    ;

selectionStatement
    :   If LeftParen expression RightParen statement (Else statement)?
    |   Switch LeftParen expression RightParen statement
    ;

iterationStatement
    :   While LeftParen expression RightParen statement
    |   Do statement While LeftParen expression RightParen Semi
    |   For LeftParen expression? Semi expression? Semi expression? RightParen statement
    |   For LeftParen declaration expression? Semi expression? RightParen statement
    ;

jumpStatement
    :   Goto Identifier Semi
    |   Continue Semi
    |   Break Semi
    |   Return expression? Semi
    |   Goto unaryExpression Semi // GCC extension
    ;

compilationUnit
    :   translationUnit? EOF
```

```
    ;

translationUnit
    :   externalDeclaration
    |   translationUnit externalDeclaration
    ;

externalDeclaration
    :   RandomDirective
    |   functionDefinition
    |   declaration
    |   Semi // stray ;
    ;

functionDefinition
    :   declarationSpecifiers? declarator declarationList? compoundStatement
    ;

declarationList
    :   declaration
    |   declarationList declaration
    ;

Auto : 'auto';
Break : 'break';
Case : 'case';
Char : 'char';
Const : 'const';
Continue : 'continue';
Default : 'default';
Define : 'define';
Do : 'do';
Double : 'double';
Else : 'else';
Enum : 'enum';
Extern : 'extern';
Float : 'float';
For : 'for';
Goto : 'goto';
Hash : '#';
If : 'if';
Import : 'import';
Include        : 'include';
Inline : 'inline';
Int : 'int';
Long : 'long';
Register : 'register';
```

```
Restrict : 'restrict';
Return : 'return';
Short : 'short';
Signed : 'signed';
Sizeof : 'sizeof';
Static : 'static';
Struct : 'struct';
Switch : 'switch';
Typedef : 'typedef';
Union : 'union';
Unsigned : 'unsigned';
Void : 'void';
Volatile : 'volatile';
While : 'while';

Alignas : '_Alignas';
Alignof : '_Alignof';
Atomic : '_Atomic';
Bool : '_Bool';
Complex : '_Complex';
Generic : '_Generic';
Imaginary : '_Imaginary';
Noreturn : '_Noreturn';
StaticAssert : '_Static_assert';
ThreadLocal : '_Thread_local';

LeftParen : '(';
RightParen : ')';
LeftBracket : '[';
RightBracket : ']';
LeftBrace : '{';
RightBrace : '}';

Less : '<';
LessEqual : '<=';
Greater : '>';
GreaterEqual : '>=';
LeftShift : '<<';
RightShift : '>>';

Plus : '+';
PlusPlus : '++';
Minus : '-';
MinusMinus : '--';
Star : '*';
Div : '/';
Mod : '%';
```

```
And : '&';
Or : '|';
AndAnd : '&&';
OrOr : '||';
Caret : '^';
Not : '!';
Tilde : '~';

Question : '?';
Colon : ':';
Semi : ';';
Comma : ',';

Assign : '=';
// '*=' | '/=' | '%=' | '+=' | '-=' | '<<=' | '>>=' | '&=' | '^=' | '|='
StarAssign : '*=';
DivAssign : '/=';
ModAssign : '%=';
PlusAssign : '+=';
MinusAssign : '-=';
LeftShiftAssign : '<<=';
RightShiftAssign : '>>=';
AndAssign : '&=';
XorAssign : '^=';
OrAssign : '|=';

Equal : '==';
NotEqual : '!=';

Arrow : '->';
Dot : '.';
Ellipsis : '...';

Identifier
    :   IdentifierNondigit
        (   IdentifierNondigit
        |   Digit
        )*
    ;

RandomDirective
    :   '#' Whitespace? (Include|Import|Define) ~[\r\n]*
    ;

fragment
IdentifierNondigit
```

```
    :   Nondigit
    |   UniversalCharacterName
    //|   // other implementation-defined characters...
    ;

fragment
Nondigit
    :   [a-zA-Z_]
    ;

fragment
Digit
    :   [0-9]
    ;

fragment
UniversalCharacterName
    :   '\\u' HexQuad
    |   '\\U' HexQuad HexQuad
    ;

fragment
HexQuad
    :   HexadecimalDigit HexadecimalDigit HexadecimalDigit HexadecimalDigit
    ;

Constant
    :   IntegerConstant
    |   FloatingConstant
    //|   EnumerationConstant
    |   CharacterConstant
    ;

fragment
IntegerConstant
    :   DecimalConstant IntegerSuffix?
    |   OctalConstant IntegerSuffix?
    |   HexadecimalConstant IntegerSuffix?
    ;

fragment
DecimalConstant
    :   NonzeroDigit Digit*
    ;

fragment
OctalConstant
```

```
    :   '0' OctalDigit*
    ;

fragment
HexadecimalConstant
    :   HexadecimalPrefix HexadecimalDigit+
    ;

fragment
HexadecimalPrefix
    :   '0' [xX]
    ;

fragment
NonzeroDigit
    :   [1-9]
    ;

fragment
OctalDigit
    :   [0-7]
    ;

fragment
HexadecimalDigit
    :   [0-9a-fA-F]
    ;

fragment
IntegerSuffix
    :   UnsignedSuffix LongSuffix?
    |   UnsignedSuffix LongLongSuffix
    |   LongSuffix UnsignedSuffix?
    |   LongLongSuffix UnsignedSuffix?
    ;

fragment
UnsignedSuffix
    :   [uU]
    ;

fragment
LongSuffix
    :   [lL]
    ;

fragment
```

```
LongLongSuffix
    :   'll' | 'LL'
    ;

fragment
FloatingConstant
    :   DecimalFloatingConstant
    |   HexadecimalFloatingConstant
    ;

fragment
DecimalFloatingConstant
    :   FractionalConstant ExponentPart? FloatingSuffix?
    |   DigitSequence ExponentPart FloatingSuffix?
    ;

fragment
HexadecimalFloatingConstant
    :   HexadecimalPrefix HexadecimalFractionalConstant BinaryExponentPart FloatingSuffix?
    |   HexadecimalPrefix HexadecimalDigitSequence BinaryExponentPart FloatingSuffix?
    ;

fragment
FractionalConstant
    :   DigitSequence? '.' DigitSequence
    |   DigitSequence '.'
    ;

fragment
ExponentPart
    :   'e' Sign? DigitSequence
    |   'E' Sign? DigitSequence
    ;

fragment
Sign
    :   '+' | '-'
    ;

fragment
DigitSequence
    :   Digit+
    ;

fragment
HexadecimalFractionalConstant
    :   HexadecimalDigitSequence? '.' HexadecimalDigitSequence
```

```
    |   HexadecimalDigitSequence '.'
    ;

fragment
BinaryExponentPart
    :   'p' Sign? DigitSequence
    |   'P' Sign? DigitSequence
    ;

fragment
HexadecimalDigitSequence
    :   HexadecimalDigit+
    ;

fragment
FloatingSuffix
    :   'f' | 'l' | 'F' | 'L'
    ;

fragment
CharacterConstant
    :   '\'' CCharSequence '\''
    |   'L\'' CCharSequence '\''
    |   'u\'' CCharSequence '\''
    |   'U\'' CCharSequence '\''
    ;

fragment
CCharSequence
    :   CChar+
    ;

fragment
CChar
    :   ~['\\\r\n]
    |   EscapeSequence
    ;

fragment
EscapeSequence
    :   SimpleEscapeSequence
    |   OctalEscapeSequence
    |   HexadecimalEscapeSequence
    |   UniversalCharacterName
    ;

fragment
```

```
SimpleEscapeSequence
    :   '\\' ['"?abfnrtv\\]
    ;

fragment
OctalEscapeSequence
    :   '\\' OctalDigit
    |   '\\' OctalDigit OctalDigit
    |   '\\' OctalDigit OctalDigit OctalDigit
    ;

fragment
HexadecimalEscapeSequence
    :   '\\x' HexadecimalDigit+
    ;

StringLiteral
    :   EncodingPrefix? '"' SCharSequence? '"'
    ;

fragment
EncodingPrefix
    :   'u8'
    |   'u'
    |   'U'
    |   'L'
    ;

fragment
SCharSequence
    :   SChar+
    ;

fragment
SChar
    :   ~["\\\r\n]
    |   EscapeSequence
    ;

LineDirective
    :   '#' Whitespace? DecimalConstant Whitespace? StringLiteral ~[\r\n]*
        -> skip
    ;

PragmaDirective
    :   '#' Whitespace? 'pragma' Whitespace ~[\r\n]*
        -> skip
```

```
    ;

Whitespace
    :   [ \t]+
        -> skip
    ;

Newline
    :   (   '\r' '\n'?
        |   '\n'
        )
        -> skip
    ;

BlockComment
    :   '/*' .*? '*/'
        -> skip
    ;

LineComment
    :   '//' ~[\r\n]*
        -> skip
    ;
```

APÊNDICE B – MATRIZES DE CONFUSÃO

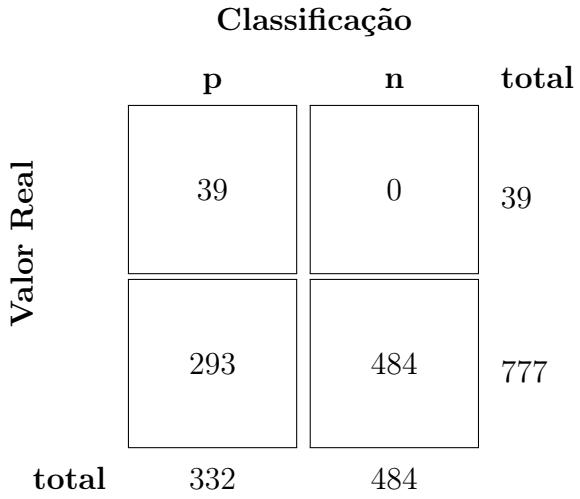

Figura 17 – Matriz de Confusão - Limiar Discriminatório 10% (PRIDE)

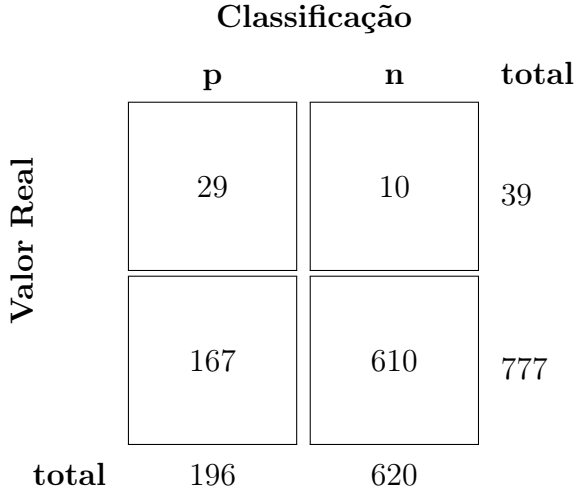

Figura 18 – Matriz de Confusão - Limiar discriminatório 10% (JPLAG)

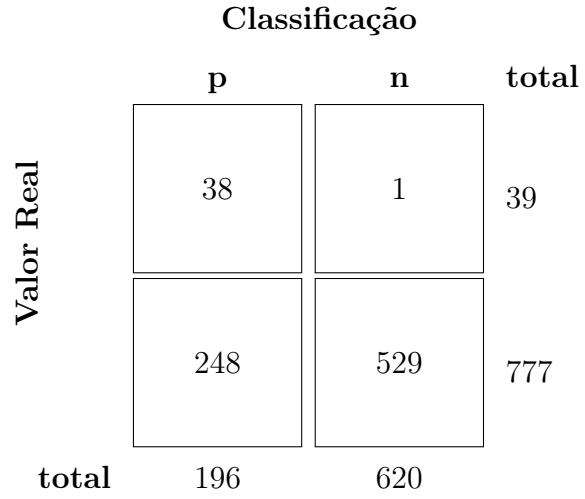

Figura 19 – Matriz de Confusão - Limiar discriminatório 10% (MOSS)

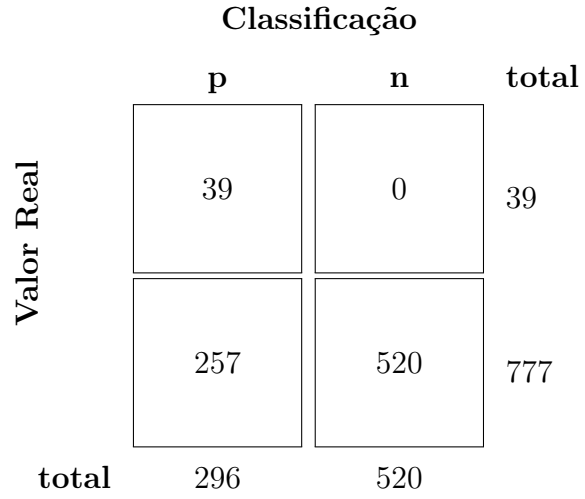

Figura 20 – Matriz de Confusão - Limiar discriminatório 20% (PRIDE)

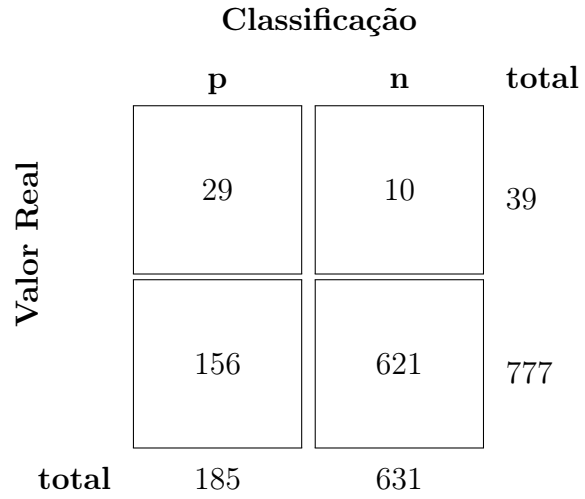

Figura 21 – Matriz de Confusão - Limiar discriminatório 20% (JPLAG)

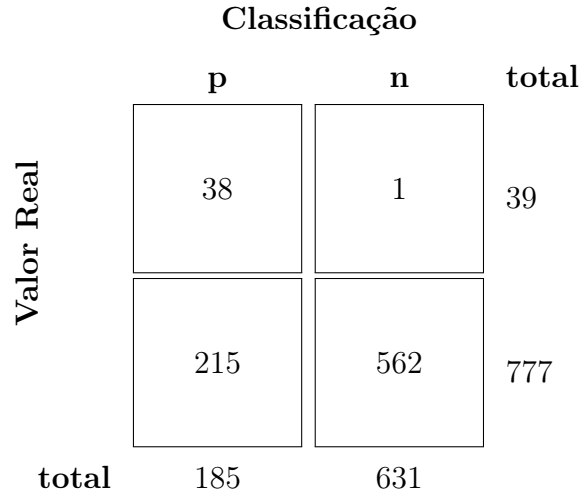

Figura 22 – Matriz de Confusão - Limiar discriminatório 20% (MOSS)

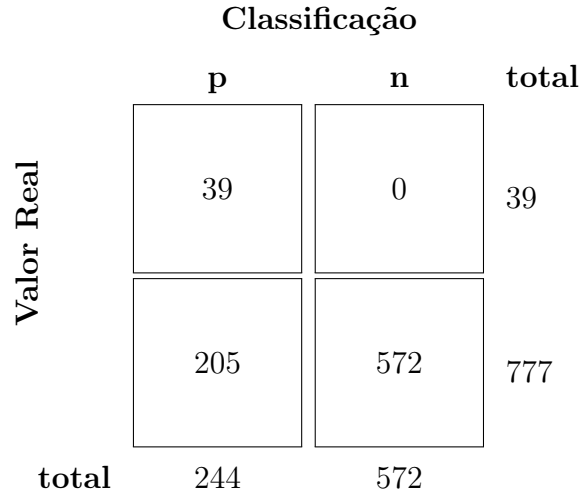

Figura 23 – Matriz de Confusão - Limiar discriminatório 30% (PRIDE)

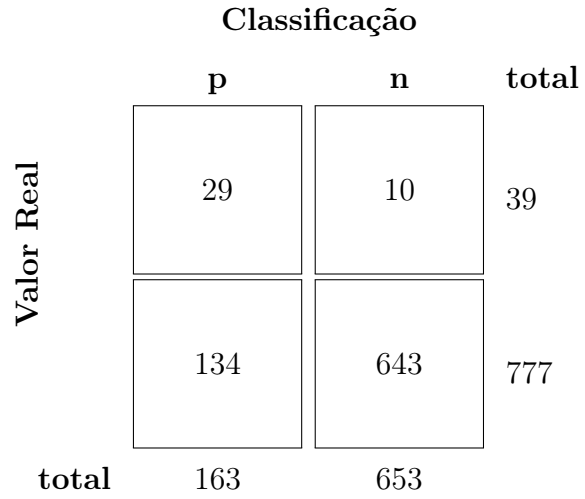

Figura 24 – Matriz de Confusão - Limiar discriminatório 30% (JPLAG)

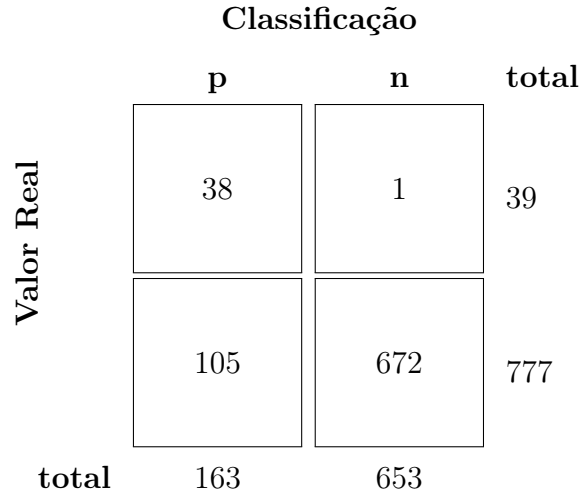

Figura 25 – Matriz de Confusão - Limiar discriminatório 30% (MOSS)

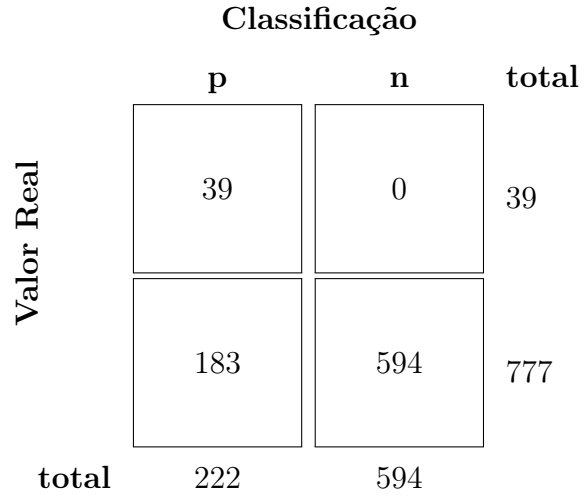

Figura 26 – Matriz de Confusão - Limiar discriminatório 40% (PRIDE)

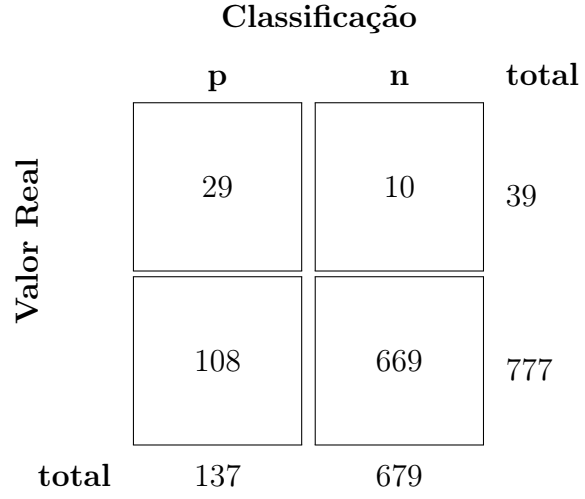

Figura 27 – Matriz de Confusão - Limiar discriminatório 40% (JPLAG)

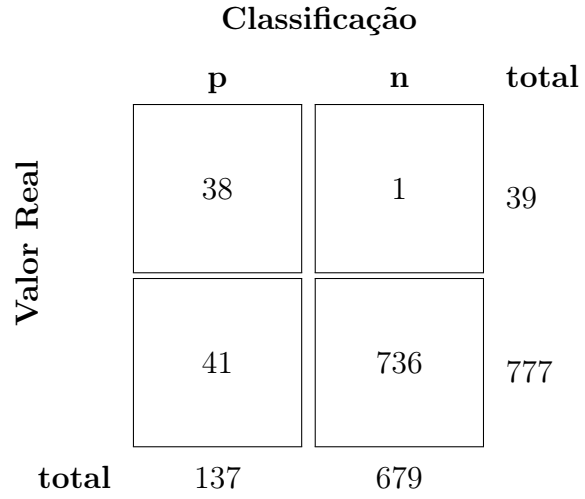

Figura 28 – Matriz de Confusão - Limiar discriminatório 40% (MOSS)

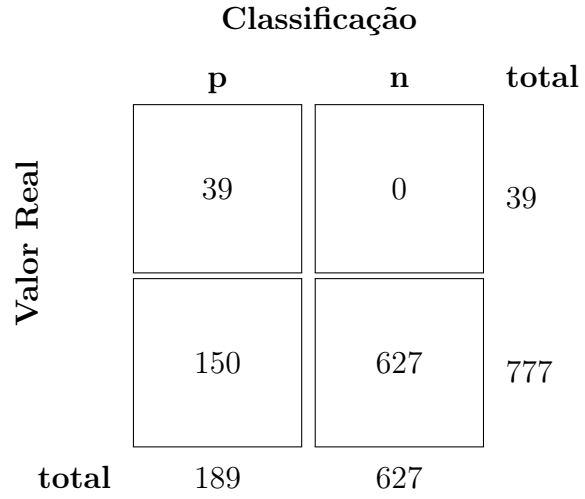

Figura 29 – Matriz de Confusão - Limiar discriminatório 50% (PRIDE)

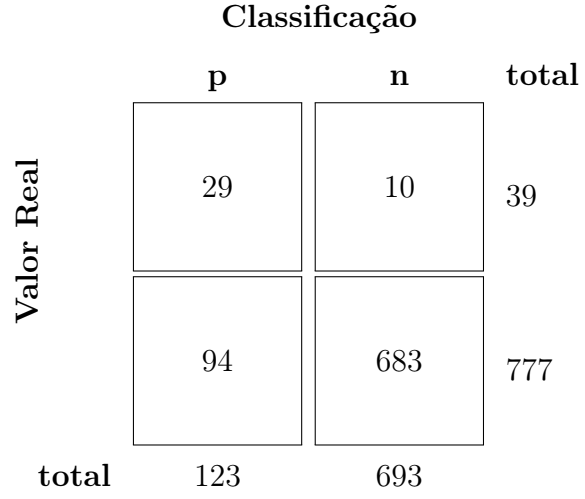

Figura 30 – Matriz de Confusão - Limiar discriminatório 50% (JPLAG)

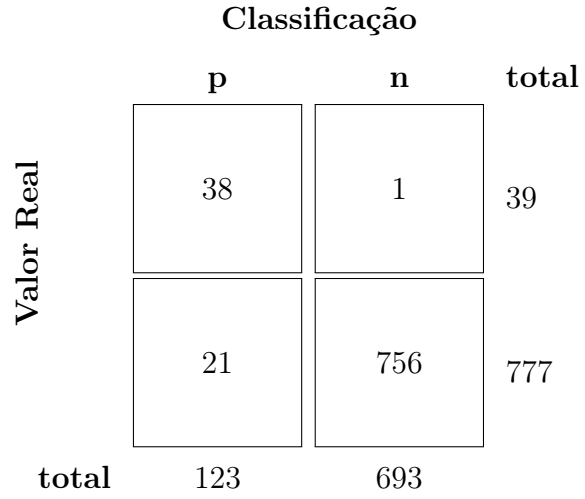

Figura 31 – Matriz de Confusão - Limiar discriminatório 50% (MOSS)

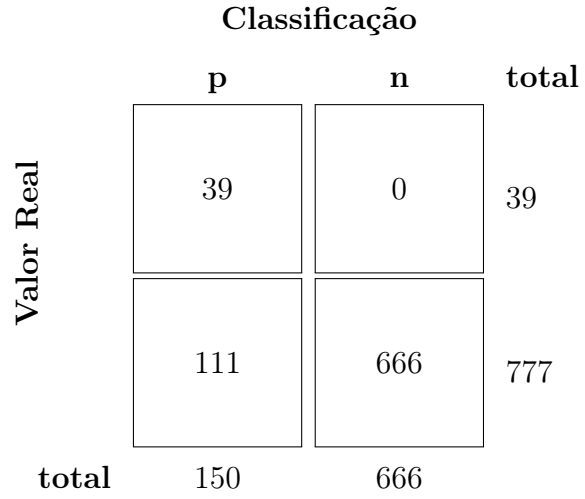

Figura 32 – Matriz de Confusão - Limiar discriminatório 60% (PRIDE)

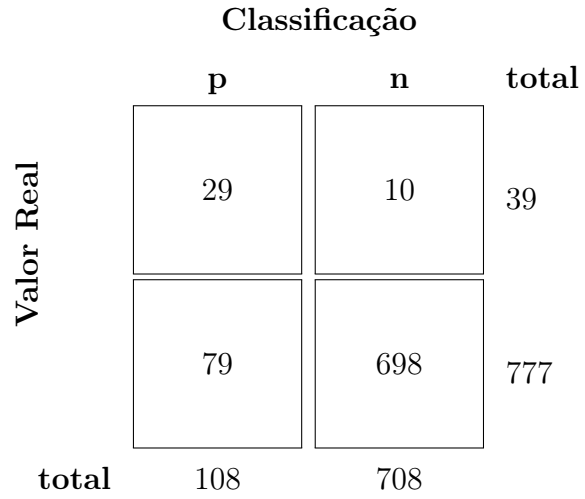

Figura 33 – Matriz de Confusão - Limiar discriminatório 60% (JPLAG)

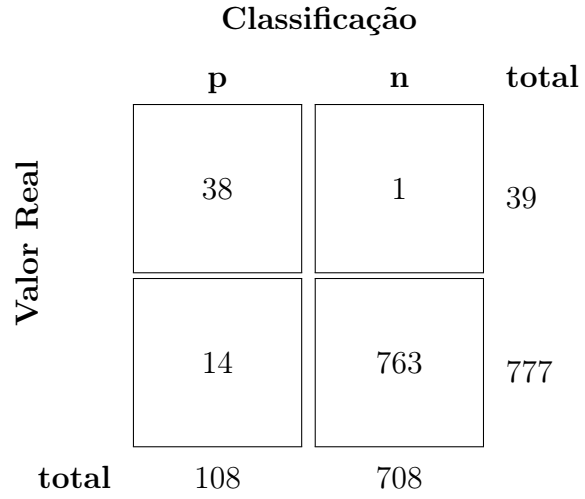

Figura 34 – Matriz de Confusão - Limiar discriminatório 60% (MOSS)

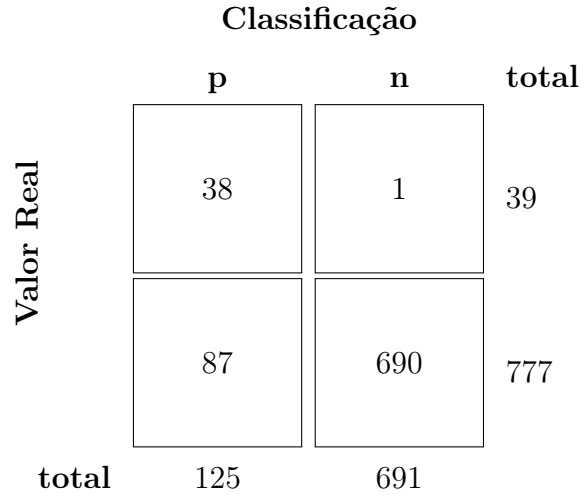

Figura 35 – Matriz de Confusão - Limiar discriminatório 70% (PRIDE)

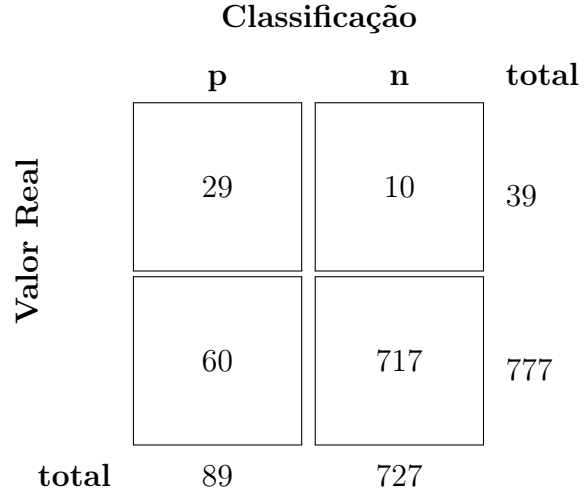

Figura 36 – Matriz de Confusão - Limiar discriminatório 70% (JPLAG)

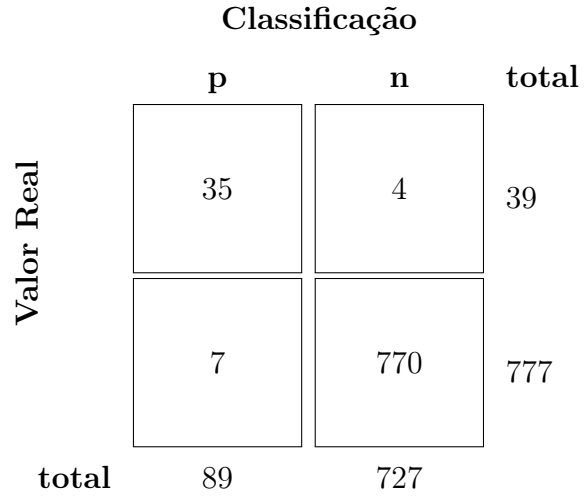

Figura 37 – Matriz de Confusão - Limiar discriminatório 70% (MOSS)

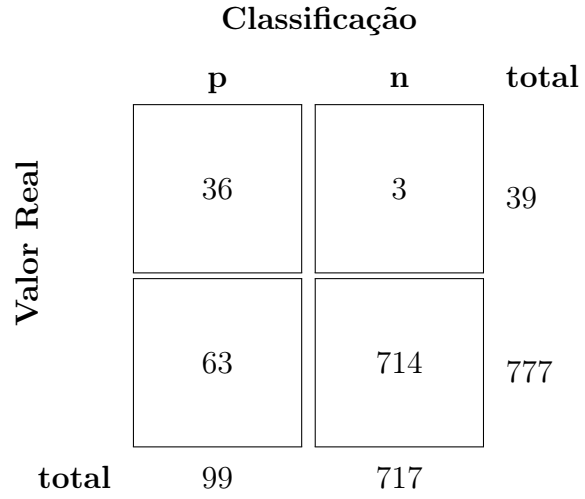

Figura 38 – Matriz de Confusão - Limiar discriminatório 80% (PRIDE)

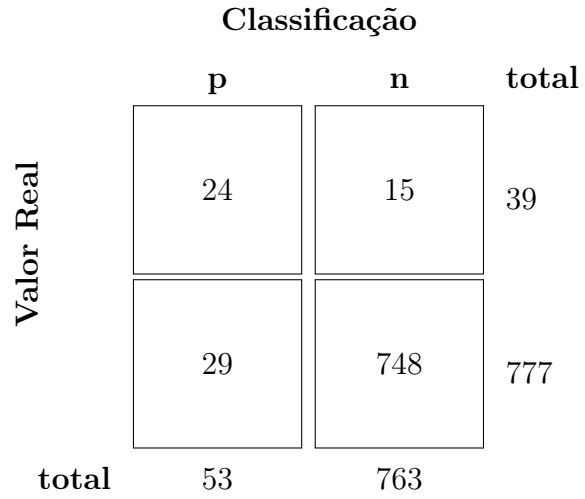

Figura 39 – Matriz de Confusão - Limiar discriminatório 80% (JPLAG)

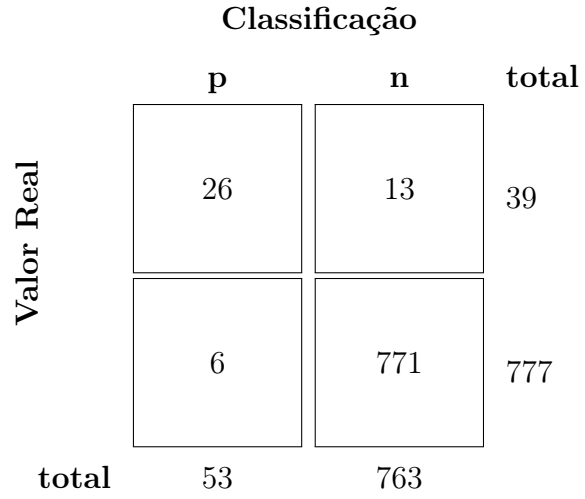

Figura 40 – Matriz de Confusão - Limiar discriminatório 80% (MOSS)

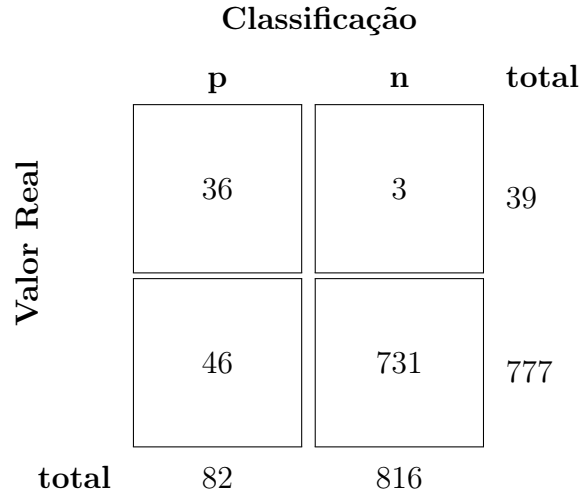

Figura 41 – Matriz de Confusão - Limiar discriminatório 90% (PRIDE)

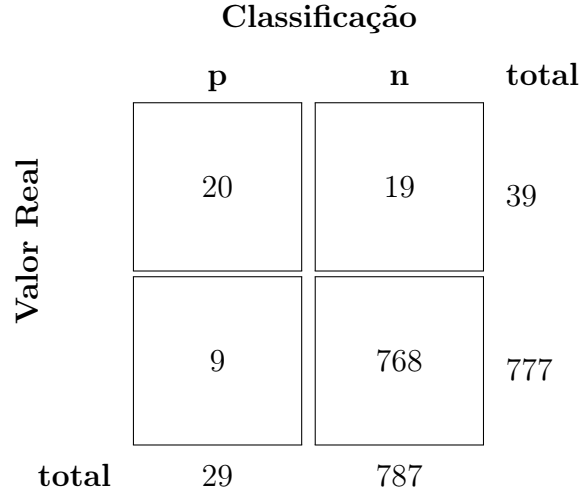

Figura 42 – Matriz de Confusão - Limiar discriminatório 90% (JPLAG)

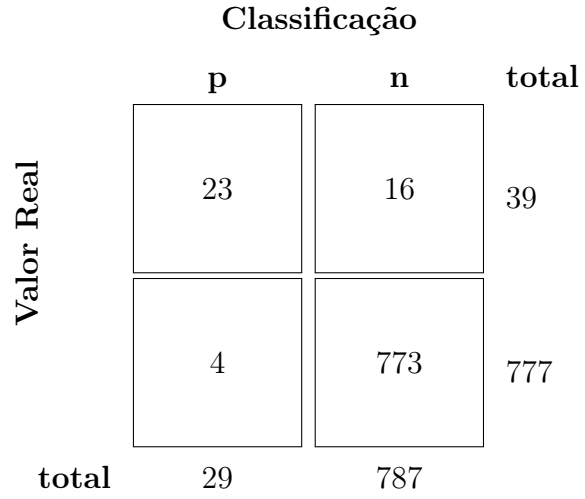

Figura 43 – Matriz de Confusão - Limiar discriminatório 90% (MOSS)

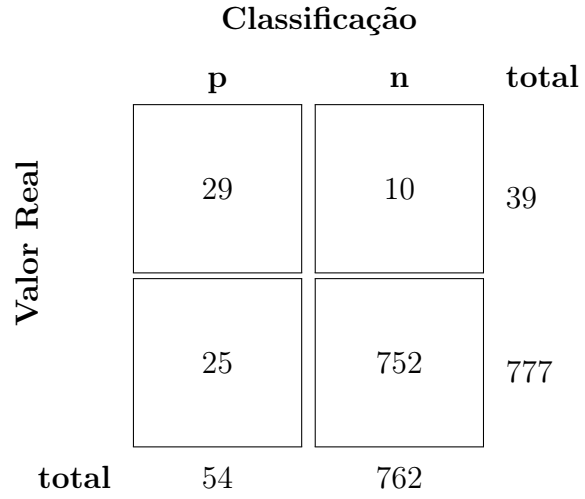

Figura 44 – Matriz de Confusão - Limiar discriminatório 100% (PRIDE)

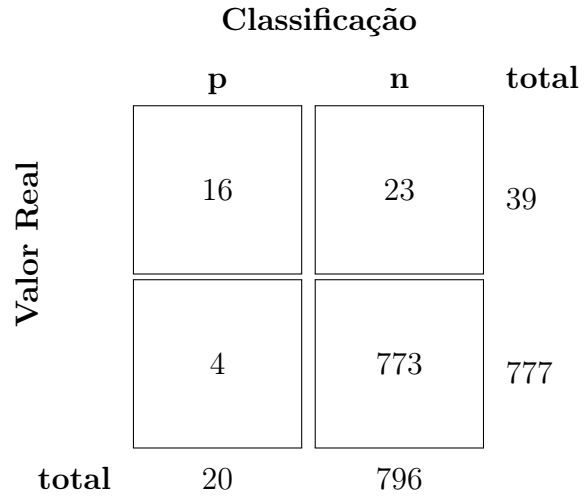

Figura 45 – Matriz de Confusão - Limiar discriminatório 100% (JPLAG)

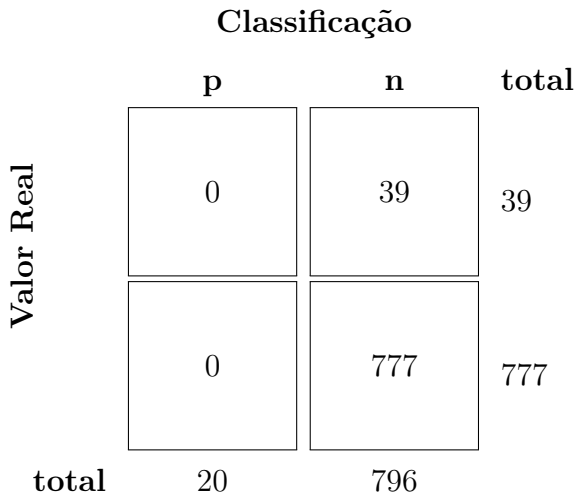

Figura 46 – Matriz de Confusão - Limiar discriminatório 100% (MOSS)