GERAÇÃO AUTOMÁTICA DE FLUXOS DE TAREFAS PARA PROBLEMAS DE APRENDIZADO DE MÁQUINA

WALTER JOSÉ GONÇALVES DA SILVA PINTO

GERAÇÃO AUTOMÁTICA DE FLUXOS DE TAREFAS PARA PROBLEMAS DE APRENDIZADO DE MÁQUINA

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: GISELE LOBO PAPPA

Belo Horizonte

Agosto de 2018

Ficha catalográfica elaborada pela Biblioteca do ICEx - UFMG

Pinto, Walter José Gonçalves da Silva.

P659g Geração automática de fluxos de tarefas para problemas de aprendizado de máquina/ Walter José Gonçalves da Silva Pinto — Belo Horizonte, 2018.
xxii, 73 p. il.; 29 cm.

Dissertação (mestrado) - Universidade Federal de Minas Gerais – Departamento de Ciência da Computação.

Orientadora: Gisele Lobo Pappa

1. Computação – Teses. 2. Programação Genética (Computação) – Teses. 3. Aprendizado do computador – Teses. I. Orientadora. II. Título.

CDU 519.6*82 (043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Geração Automática de Fluxos de Tarefas para Problemas de Aprendizado de Máquina

WALTER JOSÉ GONÇALVES DA SILVA PINTO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROFA. GISELE LOBO PAPPA - Orientadora
Departamento de Ciência da Computação - UFMG

PROFA. ANA PAULA COUTO DA SILVA
Departamento de Ciência da Computação - UFMG

PROF. LUIS ENRIQUE ZÁRATE GÁLVEZ
Departamento de Ciência da Computação - PUCMG

PROF. SANDRO CARVALHO IZIDORO
Instituto de Ciências Tecnológicas - UNIFEI

Belo Horizonte, 6 de agosto de 2018.

Dedico esse trabalho aos meus pais Marleth e Walter por sempre acreditarem em minha capacidade. E ao eu irmão Wagner por ser sempre uma competição saudável.

Agradecimentos

Eu agradeço primeiramente aos meus pais por terem me ensinado desde cedo que a busca pelo conhecimento é um caminho tortuoso, trabalhoso, cheio de frustrações e de falhas, mas é o único capaz de produzir resultados verdadeiramente duradouros. Agradeço a minha orientadora Gisele pela paciência, direcionamento e assertividade que tornaram possível não só esse trabalho mas também por proporcionar crescimento em meu crescimento como pesquisador e como indivíduo. Sou grato ao meu amigo Sandro Izidoro, por ter me ajudado e me influenciado nos meus caminhos iniciais da pesquisa que tornaram esse mestrado uma possibilidade. Agradeço a todos aqueles companheiros de laboratório: Derick, Denise, Paulo, Júlio, Camila, Samuel, Alex, Vinícius e todos aqueles que sempre me ajudaram e acompanharam esse processo.

"A life spent making mistakes is not only more honorable, but more useful than a life spent doing nothing."
(George Bernard Shaw)

Resumo

A área de Aprendizado de Máquina Automático tem como objetivo recomendar automaticamente fluxos de tarefas que devem ser seguidas para criar algoritmos de aprendizado personalizados para uma dada base de dados. Essas tarefas incluem métodos de pré-processamento de dados, algoritmos de aprendizado e seus parâmetros e técnicas de pós-processamento. A grande vantagem dos métodos dessa área está em sua capacidade de gerar fluxos sem dependência de conhecimento especializado do usuário realizando a tarefa. Esta dissertação propõe o RECIPE (REsilient ClassifIcation Pipeline Evolution), um método que faz uso de programação genética baseada em gramática para buscar por esses fluxos de tarefa considerando problemas de classificação. O RECIPE é flexível o suficiente para receber diferentes gramáticas e pode ser facilmente estendido para outras tarefas de aprendizado. Os resultados da medida F1 obtidos pelo RECIPE em 10 bases de dados são comparados a dois métodos estado da arte nessa tarefa, e são tão bons ou melhores do que os relatados anteriormente na literatura.

Abstract

Automatic Machine Learning is a growing area of machine learning that has a similar objective to the area of hyper-heuristics: to automatically recommend optimized pipelines, algorithms or appropriate parameters to specific tasks without much dependency on user knowledge. The background knowledge required to solve the task at hand is actually embedded into a search mechanism that builds personalized solutions to the task. Following this idea, this thesis proposes RECIPE (REsilient ClassifIcation Pipeline Evolution), a framework based on grammar-based genetic programming that builds customized classification pipelines. The framework is flexible enough to receive different grammars and can be easily extended to other machine learning tasks. RECIPE overcomes the drawbacks of previous evolutionary-based frameworks, such as generating invalid individuals, and organizes a high number of possible suitable data pre-processing and classification methods into a grammar. Results of f-measure obtained by RECIPE are compared to those two state-of-the-art methods, and shown to be as good as or better than those previously reported in the literature. RECIPE represents a first step towards a complete framework for dealing with different machine learning tasks with the minimum required human intervention.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

É perceptível o aumento na capacidade de geração de dados pela sociedade atual, incluindo dados provenientes de smartphones, smartwatchs, smarTVs, da Web 2.0, entre outros. Porém, dados não nos trazem nada de interessante se não forem interpretados, classificados, e contextualizados. Uma das formas de extrair conhecimento dos dados e obtenção de nova informação útil é através de técnicas de aprendizado de máquina. De acordo com Witten et al. [2011], aprendizado de máquina engloba um conjunto de técnicas que tornam possível a extração de informações relevantes de uma base de dados brutos, gerando descrições estruturadas. As descrições encontradas podem ser utilizadas para gerar um modelo de previsão, um classificador, ou um agrupamento.

O uso de técnicas de aprendizado de máquina se estende para uma gama muito ampla de áreas de conhecimento. Temos exemplos de métodos de aprendizado de máquina utilizados na verificação de falhas estruturais em aviões [Farrar & Worden, 2012], na classificação de sentimento de documentos [Pang et al., 2002], no setor financeiro para análise de risco para liberação de crédito em cartões [Khandani et al., 2010], na realização de diagnósticos médicos [Kononenko, 2001], entre muitos outros. Porém, é difícil encontrar especialistas em aprendizado de máquina nessas áreas, e o aumento do número de usuários não especialistas [Thornton et al., 2013] gera a necessidade de tornar o uso de métodos de aprendizado de máquina mais acessível, escalável e flexível (capaz de apresentar bons resultados para dados de diferentes domínios) [Olson et al., 2016b].

Uma solução para esse problema seria recomendar sempre o mesmo algoritmo para todos os problemas, sendo que esse algoritmo apresenta em geral uma boa eficácia na maioria das aplicações. Porém, o teorema de otimização *No Free Lunch* (NFL) [Ho & Pepyne, 2002] afirma que uma técnica genérica para resolver um problema sempre será superada por uma técnica especializada para o respectivo problema.

Por muito tempo, a área de meta-aprendizado, vem trabalhando para resolver o problema de recomendação do melhor algoritmo para bases de dados específicas [Vilalta & Drissi, 2002]. Existem três abordagens principais para o meta-aprendizado: combinação, seleção e geração de algoritmos. Na seleção de algoritmos um ranking é gerado com a recomendação do melhor algoritmo para a entrada [Smith-Miles, 2009]. A abordagem baseada em combinação parte do pressuposto de que um conjunto de métodos de aprendizado combinados apresentam melhores resultados que cada um isoladamente [Vilalta & Drissi, 2002]. Na abordagem baseada em geração, o algoritmo é montado através de combinação de componentes de baixo nível [Vilalta & Drissi, 2002].

A área de meta-aprendizado foca na escolha do melhor algoritmo para a base de dados em questão. Porém, em média, 70% do tempo gasto ao resolver um problema utilizando técnicas de aprendizado está na estruturação dos dados, seleção, criação e extração de atributos, normalização, entre outros métodos de pré-processamento Fernandez [2010]. Nesse contexto, o principal objetivo desse trabalho é gerar automaticamente uma sequência de passos capaz de produzir um modelo especializado para uma base de dados, capaz de obter uma acurácia superior àquelas obtidos por um algoritmo "genérico"não otimizado para a tarefa, como Support Vector Machines ou *Naive Bayes*.

Já existem iniciativas na área de ML cujo principal objetivo é recomendar automaticamente fluxos, algoritmos e/ou parâmetros apropriados para tarefas sem muita dependência do conhecimento do usuário [Olson et al., 2016b]. Essa linha de pesquisa é denominada Auto-ML [Thornton et al., 2013]. Dentre os métodos anteriormente propostos na literatura para lidar com esta tarefa estão Auto-WEKA [Thornton et al., 2013], Auto-SKLearn [Feurer et al., 2015a] e Ferramenta de otimização de fluxos baseada em árvores (TPOT) [Olson et al., 2016a]. Enquanto os dois primeiros usam um método de otimização Bayesiano para construir um fluxo de classificação, o TPOT usa um algoritmo de programação genética baseado em árvore para resolver o problema.

Todos os métodos anteriormente mencionados usam uma lista de componentes predefinidos (tarefas) que podem ser considerados durante a pesquisa. O Auto-WEKA e o Auto-SKLearn realizam uma pesquisa local para explorar esses componentes, e adicionam algumas restrições para poder evitar combinações inválidas. O TPOT, por sua vez, explora as vantagens de uma busca global, mas considera uma busca irrestrita, onde os recursos podem ser gastos para gerar e avaliar soluções inválidas. Este último pode ser considerado uma das principais desvantagens do TPOT.

Para atacar problema de geração automática de fluxos de classificação, nesse trabalho utilizaremos um algoritmo de programação genética (PG). A programação genética faz parte da área de computação evolutiva, que usa métodos inspirados na

teoria da evolução de Darwin para resolver problemas de otimização [De Castro, 2006]. Nesse tipo de algoritmo, uma população inicial é gerada, e cada indivíduo avaliado de acordo com uma função de aptidão, que mede a capacidade da solução em resolver o problema em questão. Os indivíduos são selecionados, de acordo com essa função de aptidão, para serem modificados através de operações genéticas, que incluem cruzamento e mutação. Esse processo é repetido por um número pré-definido de gerações, ao fim do qual uma solução para o problema normalmente próxima da ótima é retornada.

Quando algum tipo de conhecimento prévio está disponível para orientar a busca (como é o caso da construção de fluxos), os algoritmos de programação genética baseados em gramática (GGP) [McKay et al., 2010] aparecem como uma alternativa melhor do que a PG padrão. A principal diferença entre a PG padrão e a baseada em gramática, como o nome indica, é a definição de uma gramática para guiar a busca. Nessa direção, este trabalho propõe um algoritmo de GGP para evoluir automaticamente fluxos de classificação personalizados para o conjunto de dados de interesse, denominado RECIPE (REsilient ClassifIcation Pipeline Evolution). Embora o foco da pesquisa esteja na tarefa de classificação, o *framework* pode ser facilmente adaptado para lidar com uma série de tarefas, incluindo regressão, classificação ou agrupamento.

1.1 Objetivos

O principal objetivo desse trabalho é criar um algoritmo baseado em programação genética para a geração automática de um fluxo de aprendizado de máquina personalizado para uma base, com foco em classificação. Para isso os seguintes objetivos específicos são listados:

- Fazer um estudo completo dos métodos considerados estado-da-arte em cada uma das etapas que farão parte do fluxo.
- Estruturar esse conhecimento na forma de uma gramática.
- Investigar a melhor forma de utilizar esse conhecimento para criar a busca de um algoritmo evolucionário.
- Criar uma ferramenta Web de código aberto que possa ser usada pela comunidade.
- Realizar testes comparativos entre os fluxos gerados pelo algoritmo e os estado-da-arte.

1.2 Estrutura do Texto

O restante deste trabalho é dividido da seguinte forma:

- Capítulo 2 - Traz uma breve definição da computação evolucionária e o seu *framework* básico de funcionamento;
- Capítulo 3 - Apresenta os trabalhos relacionados à área de Auto-ML;
- Capítulo 4 - Introduz a metodologia utilizada para a criação do algoritmo de geração automática de fluxos de aprendizado de máquina;
- Capítulo 5 - Reporta a análise experimenta;
- Capítulo 6 - Discute as conclusões do nosso trabalho e apresenta as possibilidades de trabalhos futuros.

Capítulo 2

Computação Evolucionária

A computação evolucionária faz uso da teoria de Darwin para modelar algoritmos capazes de resolver problemas complexos [De Castro, 2006]. Parte-se do princípio que as soluções arbitrariamente geradas são capazes de evoluir até uma solução ótima, ou um conjunto de soluções muito próximas do ótimo. A Figura 2.1 mostra o funcionamento básico dos algoritmos evolucionários.

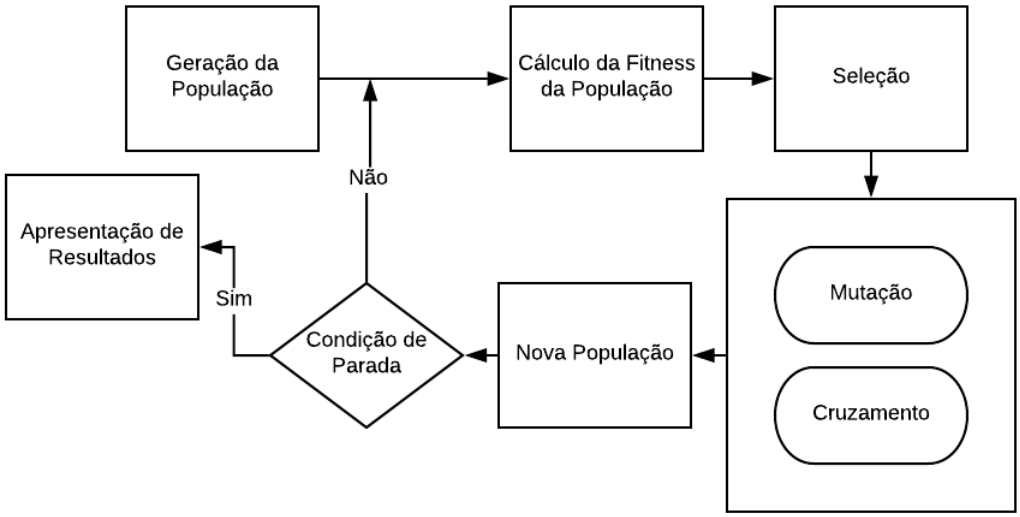


Figura 2.1: Funcionamento básico de um algoritmo evolucionário.

O primeiro passo é a criação de uma população inicial que servirá de ponto de partida da busca. A geração desses indivíduos é realizada de forma aleatória. Cada indivíduo é então avaliado de acordo com uma métrica de aptidão conhecida como *fitness* que define o quão bem esse indivíduo soluciona o problema em questão. Essa aptidão é então utilizada para selecionar os indivíduos. Os indivíduos selecionados

serão modificados através de operadores de mutação, cruzamento ou ambos, de acordo com uma probabilidade pré-definida. Todo esse processo gera uma nova população, que pode ser considerada melhor que a anterior se a função de *fitness* e operadores forem corretamente definidos. Se a condição de parada for atingida, o resultado final da execução é apresentado, caso o contrário o algoritmo executa o processo novamente a partir do cálculo da *fitness* de cada indivíduo [Back, 1996].

Toda a solução do problema por meio da computação evolucionária gira em torno de tornar um indivíduo arbitrário em uma solução quase ótima para o problema em questão. Isso torna a modelagem do indivíduo um passo muito importante nesse tipo de algoritmo. As duas estruturas mais utilizadas nas representações de indivíduos em algoritmos evolutivos são vetores e árvores [De Castro, 2006]. O que a estrutura do indivíduo guarda, é também conhecido como genótipo, e o valor semântico do indivíduo como fenótipo.

Um exemplo é a modelagem do indivíduo para o problema de máximos e mínimos de uma função. No problema em questão, o algoritmo deve retornar o valor de x que maximiza ou minimiza o resultado de uma função $f(x)$. Um indivíduo é uma provável solução para o problema, e nesse caso específico, a solução do problema é um possível valor para a variável x. Uma forma de modelar os indivíduos é através de um vetor de zeros e uns, no qual a transformação desse vetor de bits em um número se torna fácil através de um simples processo de conversão de binário para decimal. A Figura 2.2 apresenta dois indivíduos modelados como descrito. O genótipo nessa modelagem serão os vetores de zeros e uns, e o fenótipo o valor decimal que esses vetores representam.

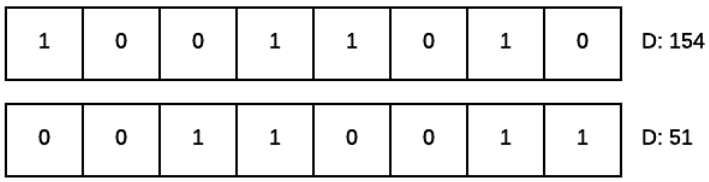


Figura 2.2: Exemplo de indivíduo para a solução do problema de maximização de uma função através de algoritmos evolucionários modelado em vetores de bits. O 'D' representa a conversão do indivíduo em decimal.

É importante que seja definida o que é uma boa solução e o que é uma solução ruim para o problema de forma que o algoritmo seja capaz de avaliar qual caminho tomar durante o processo de busca da solução. A etapa responsável por essa tarefa é conhecida como cálculo da fitness. Como neste passo os indivíduos serão avaliados de acordo com a definição do problema, quanto mais apropriada a modelagem da fitness, melhor serão os resultados encontrados pela técnica [Back, 1996]. Ainda fazendo uso do

problema de maximização dos valores de uma função, um exemplo de fitness, pode ser o resultado do valor decimal do indivíduo aplicado a função $f(x)$, o melhor indivíduo será aquele com o maior valor. Agora avaliando o caso de minimização do problema, o melhor indivíduo será aquele com o menor valor. A Figura 2.3 mostra o cálculo da fitness para os dois indivíduos da Figura 2.2 para duas funções de fitness distintas: na primeira $F1(x) = x$ e na segunda $F2(x) = 1/x + sen(x)$.

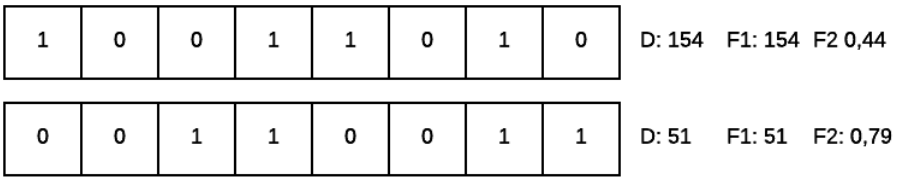


Figura 2.3: Valores de fitness para os indivíduos da Figura 2.2 considerando duas funções de fitness distintas: $F1$ e $F2$.

Na Figura 2.3, para o problema de maximização do valor de uma função, o primeiro indivíduo é mais apto se olharmos para $F1$, enquanto o segundo é mais apropriado se considerarmos $F2$.

Na teoria da evolução de Darwin, os indivíduos mais adaptados apresentam uma chance maior de sobrevivência e de passar suas características para os seus descendentes. Na computação evolucionária isso também ocorre através da etapa de seleção. A etapa de seleção tem a função de escolher probabilisticamente os indivíduos com os melhores valores de aptidão para o problema em questão. Existem vários métodos para a realização dessa etapa, sendo os mais conhecidos a seleção por torneio e o método da roleta [De Castro, 2006]. No método do torneio, K indivíduos são selecionados e suas fitness comparadas entre si. O indivíduo cuja aptidão for melhor será mantido na população e passará para a próxima etapa do processo evolutivo, onde são aplicados os operadores. Na seleção por roleta, cada indivíduo da população recebe uma proporção da roleta de acordo com o seu valor de aptidão. Um número então é sorteado, e o indivíduo que ocupa o espaço equivalente ao sorteado na roleta é mantido na nova população.

A Figura 2.4 mostra um exemplo da seleção por torneio e por roleta para os indivíduos da Figura 2.2. Ao se escolher um método de seleção, devemos ter em mente a pressão seletiva. Na natureza, a pressão seletiva são as condições ambientais que direcionam a evolução dos indivíduos para um ou outro caminho [Camargo, 2006]. Em seu artigo, Whitley et al. [1989] mostra os benefícios de se utilizar os métodos de seleção baseados em ranking (torneio) em relação a outros métodos em termos de pressão seletiva. A roleta, em contrapartida, trabalha com uma probabilidade muito

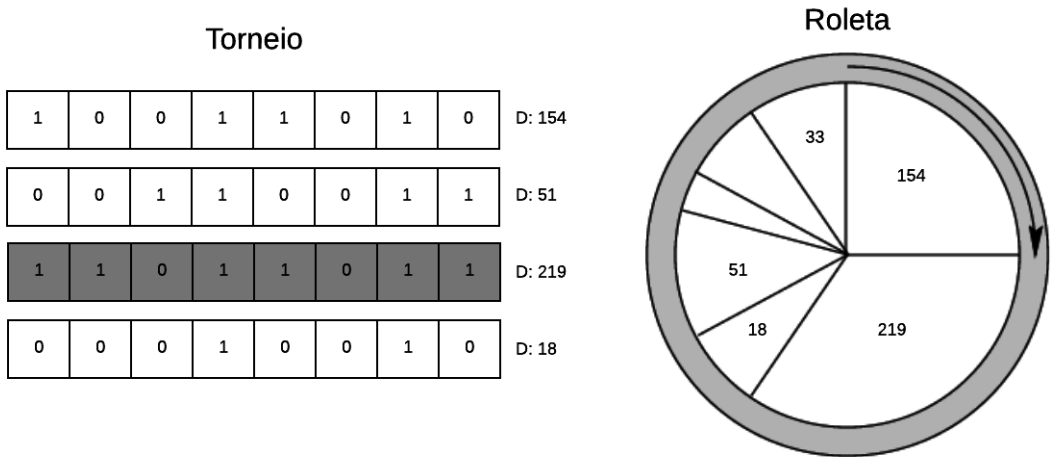


Figura 2.4: Exemplos da seleção por torneio e por roleta para os indivíduos da Figura 2.2.

grande de que o melhor indivíduo seja selecionado, porém, resguarda uma chance para os indivíduos que não possuem uma aptidão tão grande, o que reduz nossa pressão seletiva [Whitley et al., 1989].

Para a evolução dos indivíduos são aplicados operadores genéticos. Os operadores mais comumente utilizados são os de mutação e cruzamento [Back, 1996]. A função da mutação é adicionar uma diversidade ao indivíduo criando um distúrbio aleatório em seu genótipo. Já o cruzamento tem como função principal a criação de um novo indivíduo através do genótipo dos pais. O cruzamento parte do pressuposto de que pais bem adaptados irão gerar filhos melhores, e a mutação, por sua vez, se baseia na possibilidade de que a melhora desejada não ter ainda aparecido na população. A forma como esses operadores funcionam é estritamente relacionada ao problema. Por exemplo, tanto a mutação quanto o cruzamento podem ser realizados escolhendo-se um ou dois pontos do genótipo do indivíduo, dentre outras nuances particulares para cada problema. Um exemplo de mutação para o problema de máximo da função é selecionar uma posição do vetor e negar o seu valor, já o cruzamento pode ocorrer com a troca de pedaços do genótipo entre os pais. A Figura 2.5 exemplifica a aplicação de cruzamento e mutação em indivíduos representados como vetores binários.

Todo esse processo de avaliação, seleção e aplicação de operadores genéticos se repete até que o critério de parada seja satisfeito (número de gerações, limiar de solução desejado atingido, tempo de execução do algoritmo, entre outros).

Essa seção apresentou a metodologia de funcionamento de um algoritmo na área de computação evolucionária: geração da população, avaliação dos indivíduos, seleção,

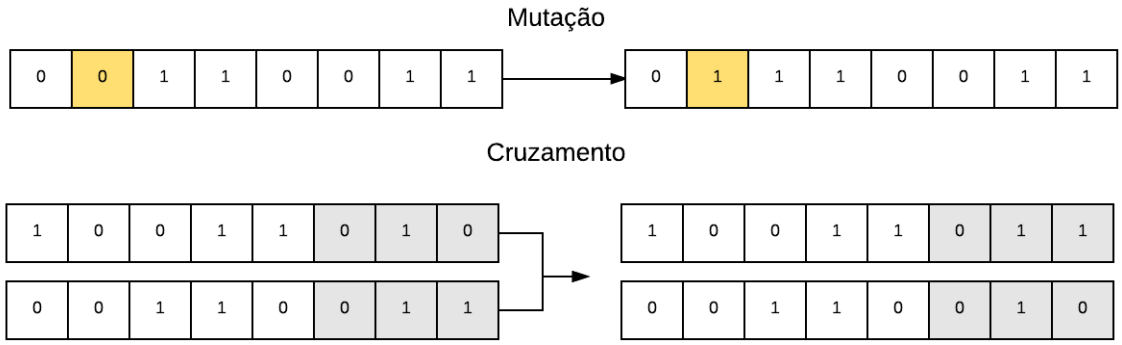


Figura 2.5: Aplicação de cruzamento e mutação nos indivíduos da Figura 2.2.

aplicação dos operadores evolucionários e nova população até que o critério de parada seja atingido. Algoritmos genéticos (GA), programação evolucionária (EP) e programação genética (GP) são exemplo de algoritmos da computação evolutiva. Eles se distinguem pela forma como os indivíduos são modelados, como os operadores evolucionários irão agir, a forma de geração da população inicial e, como consequência, como e onde os algoritmos são aplicados. A próxima seção foca na programação genética e suas nuances.

2.1 Programação Genética

De acordo com [Poli et al., 2008], a programação genética é uma técnica onde são gerados automaticamente algoritmos capazes de resolver problemas computacionais, sem que o usuário tenha a necessidade de especificar ou saber de antemão a forma ou a estrutura necessária da solução do problema.

As características mais marcantes nesse tipo de algoritmo são as formas de representação do indivíduo e de geração da população inicial. A representação geralmente ocorre através de árvores, e para a geração de cada indivíduo da população são utilizados um conjunto de não-terminais (funções) e terminais (parâmetros dessas funções). Duas propriedades muito importantes foram definidas por Koza [1994] para que a PG seja aplicada de maneira correta: Fechamento e Suficiência. O fechamento implica que todo elemento do conjunto de não-terminais deve aceitar como argumento qualquer terminal ou valor que possa ser retornado de qualquer função. Uma operação matemática que apresenta problema com a propriedade do fechamento é a divisão, onde não é possível realizar a divisão por zero, e caso o zero seja um terminal existente no conjunto, existe a possibilidade de se gerar um indivíduo inválido. A suficiência diz que os elementos existentes no conjunto de terminais e não-terminais são suficientes para

gerar uma solução para o problema e questão.

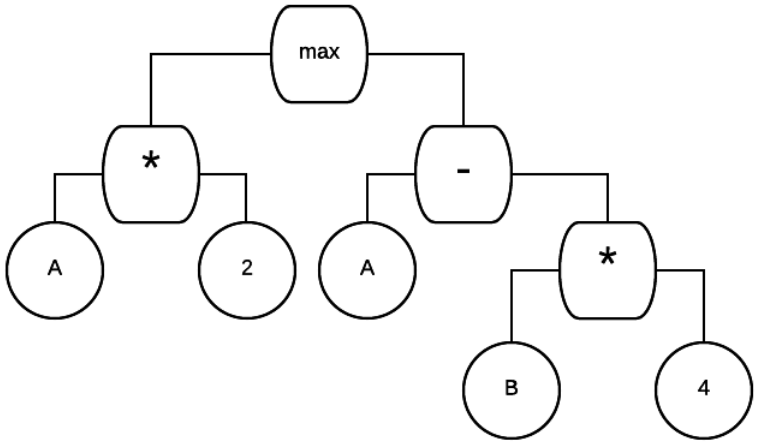


Figura 2.6: Exemplo do indivíduo representado como uma árvore.

A Figura 2.6 mostra um exemplo simples de uma árvore para representar a chamada da função max, no caso mostrado, o resultado da transcrição da árvore será $max(A + 2, A - B * 4)$. O conjunto de terminais são as folhas ($A, 2, B$ *e* 4) da árvore e os não-terminais são as nós internos ($*, -, max$). Ao se modificar o conjunto de terminas e funções é possível definir subrotinas e programas inteiros através da programação genética. Na Figura 2.7 é mostrado um exemplo mais complexo de uma árvore e do algoritmo resultante da sua transcrição. Nesse caso podemos observar a formação de uma subrotina.

No GP o operador de cruzamento tem o seu funcionamento muito similar ao apresentado na seção 2. Um ponto de cruzamento é escolhido aleatoriamente entre os pais, e um novo indivíduo é gerado a partir da modificação dos nós da árvore. Na Figura 2.8 um exemplo do cruzamento entre a árvore sintática mostrada na Figura 2.6 é apresentado e uma árvore arbitrária fazendo uso dos mesmos terminais e não-terminais que a árvore apresentada na Figura 2.6.

É interessante observar que, no exemplo apresentado, o cruzamento foi bastante destrutivo. O filho 2, por exemplo, teve toda uma sub-árvore com terminais e não-terminais substituída por um terminal. Algo que também pode ocorrer é o cruzamento não introduzir mudança na árvore.

Existem várias maneiras de se realizar a mutação em programação genética [Piszcz & Soule, 2006]: criar uma sub-árvore, remover um pedaço interno ao indivíduo, inserir elementos internamente ao indivíduo, reduzir o tamanho do indivíduo, entre outros. Nesse trabalho, o operador de mutação vai fazer uso dos elementos terminais e não-terminais para gerar uma sub-árvore da solução e substituir o ponto de mutação pelo

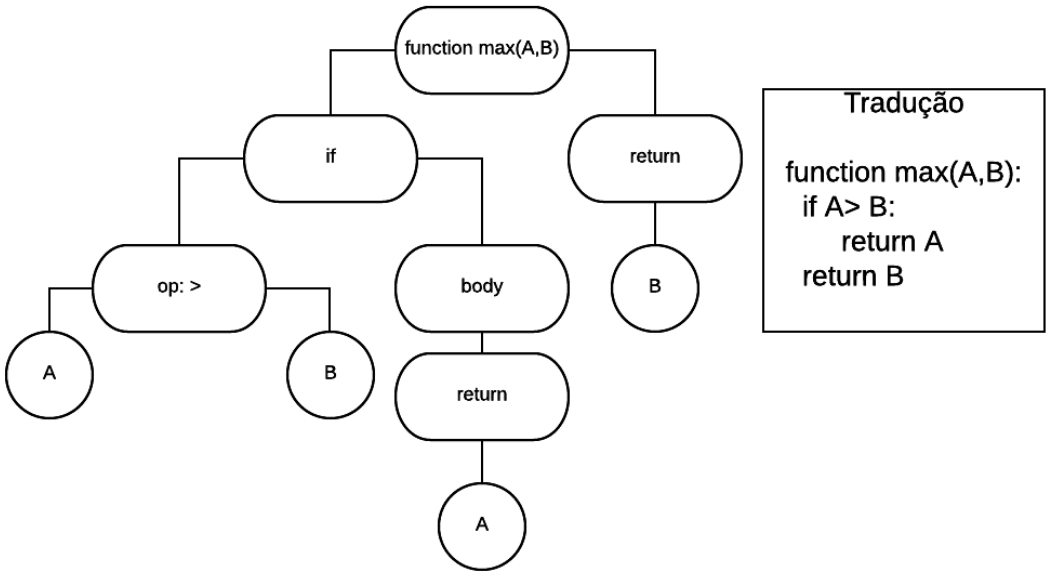


Figura 2.7: Exemplo de árvore de uma subrotina para encontrar o maior valor entre dois números A e B.

nó raiz dessa nova sub-árvore. A Figura 2.9 mostra um exemplo de uma mutação em um algoritmo de programação genética. Nesse caso em especial, uma lógica totalmente nova foi adicionada a expressão.

Os operadores de cruzamento e mutação devem ser bem compreendidos para que se encontre rapidamente a melhor configuração para a execução do algoritmo evolucionário. A escolha correta das taxas de mutação e de cruzamento não só influenciam na solução mas podem também influenciar no tempo de processamento e uso de recursos. Luke & Spector [1998] mostram que indivíduos que sofrem mutação são, em média, processados mais rapidamente do que indivíduos que passam pelo cruzamento.

2.2 Programação Genética baseada em Gramática

Um dos maiores problemas do GP canônico é a geração de indivíduos inválidos. A medida que a dificuldade do problema vai aumentando, os termos terminais e não-terminais se tornam mais complexos e por consequência os indivíduos gerados também possuem sua complexidade elevada. Se torna mais fácil a geração de indivíduos inválidos a partir da combinação dos valores terminais e não terminais, e isso significa gasto de recurso computacional, já que esses indivíduos serão avaliados. Uma solução para esse problema foi a adição de gramáticas na técnica evolucionária. A programação genética baseada em gramática é uma alternativa criada para se adicionar restrições aos operadores genéticos (geração do indivíduo, mutação e cruzamento), tornando o

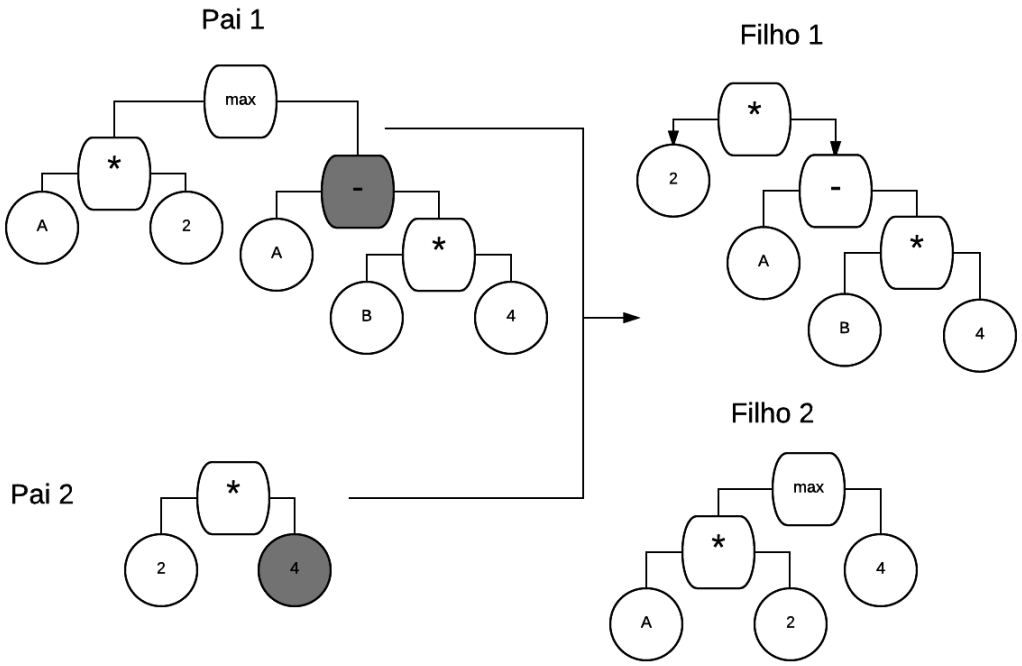


Figura 2.8: Exemplo do cruzamento entre dois indivíduos em um algoritmo de programação genética.

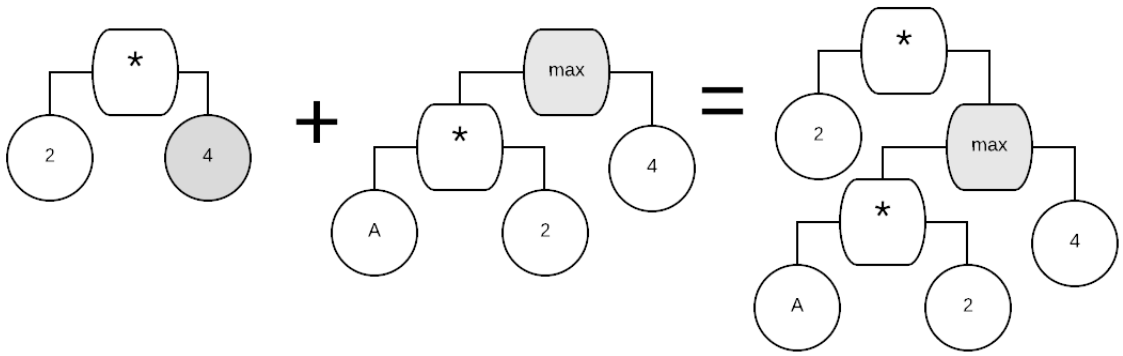


Figura 2.9: Exemplo da mutação em um algoritmo de programação genética.

algoritmo capaz de evitar a formação de indivíduos inválidos [McKay et al., 2010]. É importante salientar que ao se utilizar a gramática, evita-se também o uso de estratégias custosas para reparar resultados de operadores genéticos [McKay et al., 2010] fazendo melhor uso dos recursos.

Problemas como o da Figura 2.6 apresentam um conjunto de regras de formação mais complexas e mesmo uma simples modificação na árvore sintática já torna o indivíduo inválido. Nesses casos se torna cada vez mais difícil satisfazer a propriedade de fechamento para que apenas indivíduos sintaticamente válidos sejam gerados pelo

algoritmo [Whigham et al., 1995].

A gramática livre de contexto (CFG) é uma solução que satisfaz a propriedade de fechamento evitando a geração de indivíduos inválidos. Uma gramática G é representada por um 4-tupla $<N, T, P, S>$, onde N representa um conjunto de não-terminais, T um conjunto de terminais , P o set de regras de produção e S (um membro de N) o símbolo de início. As regras de produção definem o idioma que a gramática representa ao combinar os símbolos gramaticais [Whigham et al., 1995].

O funcionamento da programação genética baseada em gramática é facilmente visto na formação automática de frases sintaticamente válidas. Duas simples frases: "Derick é mestre"e "Samuel é doutor"podem ser formadas a partir da gramática apresentada na Figura 2.10.

```
<Start> ::= <oracao>
<oracao> :: = <subs> <verb> [<adjt>]
<subs> ::= Samuel | Derick
<ver> ::= é
<adjt> ::= doutor | mestre
```

Figura 2.10: Amostra da gramática livre de contexto para gerar frases sintaticamente corretas.

Assim como no GP canônico, no GP baseado em gramática a população inicial é composta por árvores de derivação geradas a partir da CFG onde cada indivíduo é uma dessas árvores. A Figura 2.11 mostra as frase "Derick é mestre"e "Samuel é doutor"representadas como indivíduos de uma população gerada a partir da gramática da Figura 2.10.

Com uma pequena alteração na gramática da Figura 2.10 podemos gerar frases mais complexas mantendo a validade sintática dos indivíduos, essa é uma das grandes vantagens da programação genética baseada em gramática. A Figura 2.12 mostra uma gramática capaz disso. A partir dessa gramática mais complexa o indivíduo da Figura 2.13 pode ser gerado.

É importante ter em mente que os indivíduos gerados serão sintaticamente válidos, a sua validação semântica é feita pela função de fitness.

No cruzamento, dois nós internos da árvore de derivação, rotulados com o mesmo símbolo não-terminal da gramática, são escolhidos aleatoriamente e as duas sub-árvores de derivação são trocadas entre os indivíduo. Em outras palavras, o cruzamento funciona da mesma forma que o cruzamento de sub-árvore do GP padrão, com a restrição adicional de que os pontos de cruzamento devem necessariamente ter o mesmo rótulo na gramática [McKay et al., 2010].

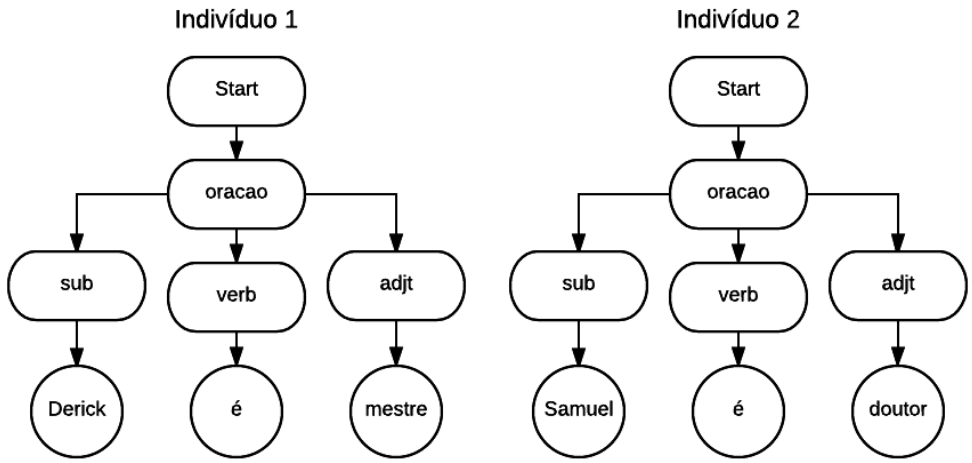


Figura 2.11: Exemplo de árvores geradas pela gramática da Figura 2.10

```
<Start> ::= <oracao> [<conj> <oracao>]
<oracao> :: = <subs> <verb> [<adjt>]
<subs> ::= Samuel | Derick | Walter
<ver> ::= é | for | era
<adjt> ::= doutor | mestre | feliz | triste
<conj> ::= e | se
```

Figura 2.12: Amostra da gramática livre de contexto para gerar frases mais complexas que se mantêm sintaticamente corretas.

A mutação em programação genética baseada em gramática utilizada apresenta uma pequena variação em relação ao operador no GP canônico. O operador irá selecionar um nó interno de forma aleatória e gerar uma sub-árvore a partir desse nó. A sub-árvore que nasce a partir desse nó é excluída e substituída por uma nova, assim como no GP canônico. A principal diferença esta no fato da sub-árvore ser gerada aleatoriamente de acordo com a gramática, começando pelo mesmo não-terminal escolhido como ponto da mutação, isso garante que os indivíduos gerados na mutação obedecem as restrições da gramática e são sempre válidos para o problema em questão.

2.3 Elitismo

O elitismo em computação natural é uma operação onde parte da população é passada para a geração seguinte sem sofrer nenhuma modificação. O uso incorreto desse operador pode exercer uma pressão seletiva muito grande causando uma convergência a um

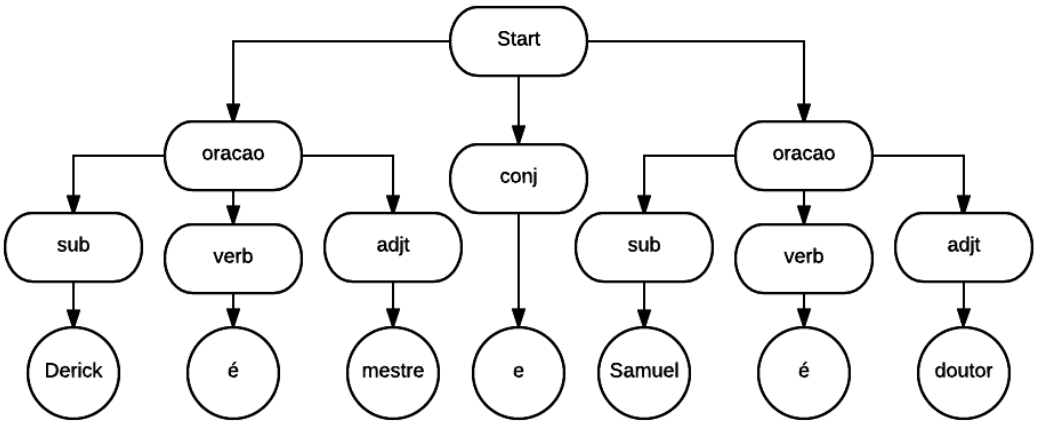


Figura 2.13: Exemplo indivíduo gerado pela gramática da Figura 2.12

máximo local ao invés do global [De Castro, 2006]. Nesse trabalho, o operador impacta positivamente no tempo de execução do algoritmo. É custoso calcular a aptidão de cada indivíduo, assim, o elitismo garante que os indivíduos mais aptos não tem sua fitness recalculada.

Capítulo 3

Trabalhos Relacionados

Embora o termo Auto-ML tenha sido recentemente cunhado, a área em si não é nova e recebeu nomes diferentes, incluindo hiper-heurísticas, otimização de hiper-parâmetro e meta-aprendizagem construtiva [Pappa et al., 2014]. Isso aconteceu porque a principal ideia por trás do Auto-ML apareceu nos campos do aprendizado de máquina e otimização em diferentes épocas, e foram desenvolvidos de forma independente.

Neste trabalho, estamos particularmente interessados em Auto-ML para problemas de classificação, focaremos em trabalhos propostos para resolver esta tarefa. A maioria da literatura se concentra em buscar os componentes (algoritmos de processamento de dados e classificação) para otimizar fluxos de aprendizado de máquina ao invés de criar fluxos completos com métodos de pré-processamento, classificação e pós-processamento. Identificamos seis modelos de classificação diferentes que tiveram seus componentes otimizados usando a abordagem de otimização de fluxos já conhecidos: (i) Redes neurais artificiais [Mendoza et al., 2016; Stanley & Miikkulainen, 2002; Yao, 1999], (ii) Algoritmos de indução de regras [Pappa & Freitas, 2009], (iii) Máquinas de vetor de suporte [Dioşan et al., 2012; Mantovani et al., 2015], (iv) Árvores de decisão [Barros et al., 2013], (v) Classificadores de rede Bayesiana [Sá & Pappa, 2013, 2014], (vi) Redes neurais Bayesianas [Springenberg et al., 2016].

No entanto, este trabalho não está centrado em apenas um tipo de algoritmo (por exemplo, algoritmos de árvores de decisão). Ao invés disso, buscamos uma abordagem mais geral para o usuário. Portanto, a principal ideia deste trabalho é gerar fluxos de aprendizado de máquinas completos, ou seja, um fluxo de tarefas de ML que pode conter etapas de pré-processamento (por exemplo, construção de recurso ou seleção de características), deve ter um algoritmo de classificação (por exemplo, árvores de decisão ou K-vizinhos próximos) e possivelmente tendo uma estratégia de pós-processamento (por exemplo, votação ou *stacking*). Ao selecionar os métodos nestes três subcompo-

nentes (pré-processamento, classificação e pós-processamento), devemos considerar a otimização de seus respectivos hiper-parâmetros. Embora o fluxo gerado seja específico para um conjunto de dados, ele é suficientemente geral para ser executado em qualquer outro.

Por este motivo, este capítulo detalha mais os cinco métodos principais que resolvem essa tarefa:

1. Auto-WEKA [Thornton et al., 2013; Kotthoff et al., 2017],
2. Auto-SKLearn [Feurer et al., 2015a],
3. Probabilistic Matrix Factorization for Automated Machine Learning (PMF-Auto-ML) [Fusi & Elibol, 2017],
4. Tree-based Pipeline Optimization Tool (TPOT) [Olson et al., 2016a,c],
5. Genetic Programming for Machine Learning (GP-ML) [Křen et al., 2017].

Nas próximas seções detalhamos os métodos que lidam com esta tarefa e serão utilizados como *baseline* para esse trabalho, incluindo o Auto-WEKA [Thornton et al., 2013], o Auto-SKLearn [Feurer et al., 2015a] e o Tree-based Pipeline Optimization Tool (TPOT) [Olson et al., 2016a].

3.1 Otimizadores Bayesianos

O Auto-WEKA e o Auto-SKLearn são métodos baseados na otimização Bayesiana e seu principal objetivo é encontrar a melhor combinação entre fluxos completos de aprendizado de máquina e seus respectivos parâmetros. Thornton et al. [2013] formaliza o aprendizado de máquina automático como seleção de algoritmo combinado com otimização de hiperparâmetros, i.e. CASH (*Combined Algorithm Selection and Hyperparameter optimization*) e dois problemas importantes são citados: (i) nenhum método de aprendizagem de máquina único funciona melhor em todos os conjuntos de dados [Pappa et al., 2014] (*No Free Lunch*), para cada bases de dados é necessário encontrar o método que mais se adequa e (ii) alguns métodos de aprendizagem de máquina são muito dependentes da otimização dos seus hiperparâmetros. Como um método de aprendizado com seus parâmetros otimizados se torna uma possível instância da solução, esses problemas estão interligados. AutoSK e o Auto-Weka exploram essa interconexão e tratam os problemas (i) e (ii) como um problema único para resolver o problema CASH.

Ambos os métodos implementaram a versão baseada em floresta aleatória do SMAC que fornece um desempenho melhor do que o estimador de parzen estruturado em árvore [Hutter et al., 2011; Thornton et al., 2013]. A otimização bayesiana encontra um modelo probabilístico capaz de capturar a relação entre configurações de hiperparâmetros e seu desempenho medido, em seguida, usa este modelo para selecionar a configuração de hiperparâmetro mais promissora, avalia essa configuração de hiperparâmetro, atualiza o modelo com o resultado e itera. Para o caso da geração automática de fluxos de aprendizado de máquina, o método Auto-ML primeiro escolhe o algoritmo de classificação (ou o método de pré-processamento) e, somente após essa etapa, seus parâmetros são otimizados. O uso dessa abordagem de otimização pode ser vantajoso no sentido de dividir o espaço de busca em dois, mas também pode excluir algoritmos que, com os parâmetros certos, possam gerar melhores resultados do que os selecionados.

Observamos algumas melhorias no Auto-SKLean quando comparamos isso com o Auto-WEKA. Por exemplo, o Auto-SKLearn pode ser inicializado via *meta-learning* [Feurer et al., 2015b] e também pode construir um conjunto para combinar os resultados da classificação em uma fase de pós-processamento. Além disso, o Auto-SKLearn utiliza o modelo *Random Online Adaptive Racing* (ROAR) [Hutter et al., 2011] como uma opção para buscar fluxos adequados.

O Auto-WEKA automatiza o processo de seleção do melhor fluxo ML usando o WEKA [Hall et al., 2009], enquanto o Auto-SKLearn visa otimizar as fluxos fazendo uso da biblioteca SciKit-Learn [Pedregosa et al., 2011]. A escolha desta biblioteca pelos métodos Auto-ML atuais é motivada pelo grande número de métodos já implementados e a popularidade desta biblioteca em Python. Além disso, Auto-SKLearn tem algumas melhorias quando comparado ao Auto-WEKA. Por exemplo, Auto-SKLearn pode ser inicializado via *meta-learning* [Feurer et al., 2015b] e também pode construir um conjunto para combinar os resultados da classificação em uma fase de pós-processamento. Por isso, usaremos o Auto-SKLearn nas comparações com o método proposto ao invés do Auto-WEKA.

3.2 Baseados em GP

Ao contrário dos métodos apresentados na seção 3.1, o TPOT aplica um algoritmo de programação genética canônica (GP) para buscar o fluxo mais adequado para um problema de aprendizado de máquina, realizando uma abordagem de busca global. Ele também procura métodos disponíveis na biblioteca SciKit-Learn, mas tem um espaço

de busca menor do que Auto-SKLearn. Uma diferença no espaço de busca do TPOT em comparação com os métodos acima mencionados e o proposto aqui é que permite o uso de muitas cópias do conjunto de dados, que são processadas em paralelo por diferentes métodos de pré-processamento e posteriormente combinados. Por exemplo, um fluxo pode ter dois ou mais métodos de seleção de atributos e, em seguida, um método de combinação é usado para verificar quais são os recursos comuns e distintos encontrados pelas técnicas. Na sua nova implementação ele também suporta uma versão projetada especificamente para estudos de bioinformática, chamada TPOT-MDR [Sohn et al., 2017]. Nesta versão, o TPOT implementa dois novos operadores de pré-processamento que são utilizados na análise genética de doenças humanas: *Multifactor Dimensionality Reduction* (MDR) e *Expert Knowledge Filter* (EKF). Além disso, o TPOT realiza busca multi-objetivo usando a seleção de Pareto através do algoritmo NSGA-II [Deb et al., 2002]. Neste caso, são considerados dois objetivos distintos: maximizar a medida final de precisão do fluxo, bem como minimizar a complexidade geral do fluxo, dada pelo número total de operadores selecionados, a fim de evitar *overfitting*. Um exemplo de individuo gerado pelo TPOT é mostrado na Figura 3.1

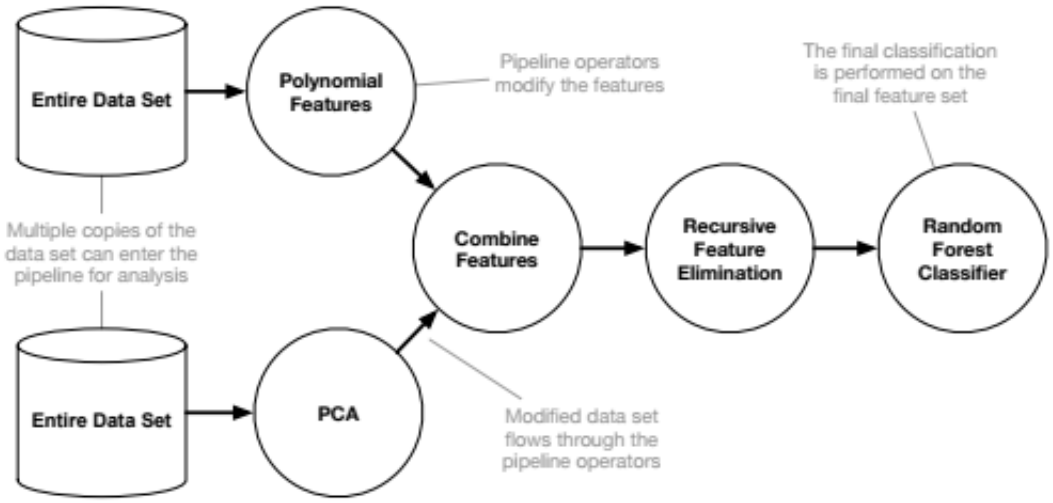


Figura 3.1: Indivíduo gerado pelo TPOT e seu fluxo de processamento.

Fonte: [Olson et al., 2016b]

A Figura 3.1 mostra que o TPOT pode fazer uso de múltiplos métodos de pré processamento para modificar os atributos da base de dados, e ao final ocorre a etapa de combinação de atributos para geração da base de dados que será utilizada pelos métodos de classificação. Em casos onde dois ou mais métodos de classificação são selecionados, o TPOT utiliza uma técnica de *stacking* para combiná-los. Em outras palavras, o resultado da primeira classificação é adicionado a base de dados original.

Essa nova base, com um atributo extra, é então utilizada pelo próximo método de classificação. A Figura 3.1 [Olson et al., 2016b] mostra que apenas o último elemento do fluxo é utilizado como método de classificação das bases de dados. Para avaliar o fluxo gerado, o TPOT realiza a separação da bases de dados em dois conjuntos, um de teste (25%) e um de treino (75%), de forma que o fluxo é avaliado apenas no conjunto de testes.

Uma das principais desvantagens do TPOT é que ele pode criar fluxos que são arbitrários/inválidos, ou seja, pode criar um fluxo que não consegue resolver um problema de classificação, uma vez que não há restrições em que tipo de componentes podem ser combinados. Por exemplo, o TPOT pode criar um fluxo sem um algoritmo de classificação [Olson et al., 2016a]. Isso também leva a um desperdício de recursos computacionais, uma vez que esses indivíduos são identificados como inválidos e recebem um valor físico muito baixo durante a avaliação dos fluxos.

O GP-ML [Křen et al., 2017] supera esta desvantagem usando um método de programação genética fortemente tipado (STGP). Um método STGP restringe os fluxos SciKit-Learn de tal forma que eles são aceitáveis a partir do ponto de vista do aprendizado da máquina, ou seja, o método faz com que o GP-ML gere apenas fluxos viáveis. Além disso, GP-ML usa um algoritmo evolutivo assíncrono ao invés de um geracional. Conforme mencionado por Scott & De Jong [2016], a evolução assíncrona é tendenciosa para as partes de avaliação mais rápidas do espaço de busca. Křen et al. [2017] considera isso uma vantagem para a tarefa aprendizado automático de máquina porque um fluxo mais rápido geralmente é preferível a um mais lento, quando ambos apresentam valores de precisão semelhantes.

3.3 Outros

O Auto-WEKA e Auto-SKLearn são métodos que dependem de um algoritmo de otimização Bayesiano chamado Configuração de Algoritmo Baseado em Modelo Sequencial (SMAC). A ideia desse algoritmo é encontrar a melhor combinação entre os fluxo completos e seus respectivos (hiper) parâmetros. Ambos os métodos seguem uma abordagem hierárquica para encontrar o "melhor"fluxo de classificação ao conjunto de dados em mãos. Nesse caso, o método Auto-ML primeiro escolhe o algoritmo de classificação (ou o método de pré-processamento) e, somente após essa etapa, seus parâmetros são otimizados. O uso desta abordagem de otimização hierárquica pode ser vantajoso no sentido de dividir o espaço de busca em dois, mas também pode excluir algoritmos que, com os parâmetros certos, possam gerar melhor resultados que os selecionados.

No entanto, Li et al. [2017] e Fusi & Elibol [2017] observaram que os métodos baseados em SMAC tendem a sofrer em espaços de busca de hiper-parâmetro de alta dimensão e geralmente apresentam performances, em média, semelhantes ou piores do que uma busca aleatória. Isso é observado experimentalmente por ambos os trabalhos e ocorre devido à necessidade de amostrar configurações hiper-parâmetro suficientes para obter uma estimativa adequada da probabilidade predicativa posterior em um espaço de alta dimensão.

O PMF-Auto-ML [Fusi & Elibol, 2017] tenta superar esta questão ao atacar o problema de aprendizado de máquina automático no SciKit-Learn como um problema de filtragem colaborativa, que pode ser resolvido com algoritmos de fatoração de matrizes. A ideia é que se dois conjuntos de dados tiverem resultados semelhantes para alguns fluxos, é provável que os fluxos restantes produzam resultados semelhantes. Isso se assemelha a um problema de filtragem colaborativa em sistemas de recomendação.

3.4 Considerações Finais

O RECIPE tem três melhorias importantes quando comparado ao Auto-SKLearn e o TPOT. Em primeiro lugar, ele usa uma gramática para organizar o conhecimento adquirido da literatura sobre o quão bem sucedidos os fluxos de aprendizado de máquina se parecem. A gramática evita a geração de fluxos inválidos e pode acelerar a busca. Em segundo lugar, ele funciona com um maior espaço de busca de fluxos do que o Auto-SKLearn e o TPOT. Embora isso torne a busca mais desafiadora, também dá a oportunidade de encontrar uma maior variedade de fluxos. Finalmente, a busca global guiada nos permite avaliar simultaneamente todo o fluxo em vez de buscar primeiramente os parâmetros discretos e, em seguida, os parâmetros contínuos como o Auto-SKLearn faz.

Capítulo 4

Metodologia

Este capítulo apresenta o RECIPE, um método baseado em GGP proposto para gerar automaticamente fluxos de algoritmos de classificação, e ilustrado na Figura 4.1. O RECIPE recebe como entrada um conjunto de dados e uma gramática, que é usada para inicializar a população. Cada indivíduo é representado por uma árvore de derivação construída a partir da gramática livre de contexto (CFG), que abrange todo o conhecimento obtido de especialistas sobre como gerar efetivamente um fluxo de classificação. Os indivíduos são mapeados em fluxos implementados pela biblioteca SciKit-Learn, que são executados em uma amostra de dados do problema que está sendo resolvido e avaliado de acordo com uma métrica de precisão. Os operadores de cruzamento e mutação são aplicados após uma seleção de torneio e garantem que os novos indivíduos gerados também respeitem as regras de produção da gramática. O elitismo também é usado, e a evolução continua por um número máximo de gerações.

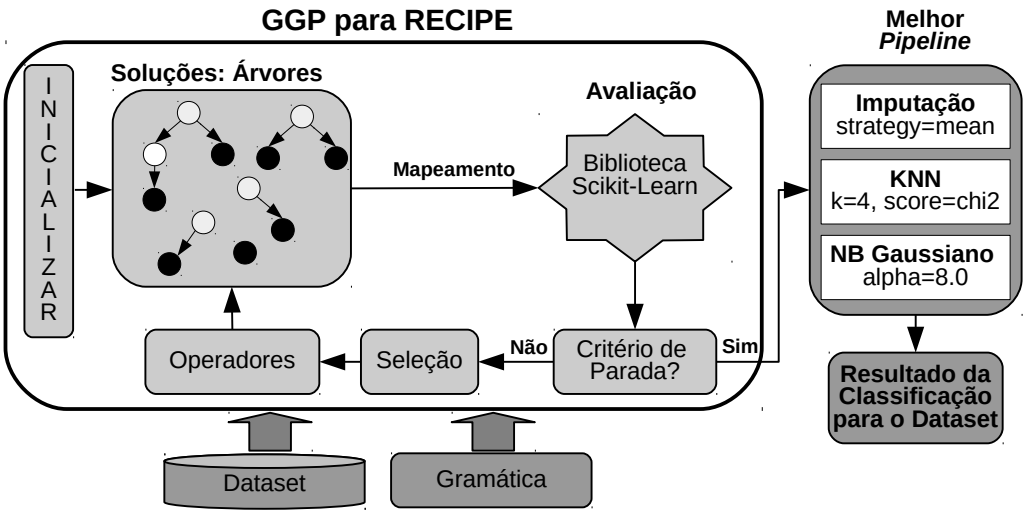


Figura 4.1: Framework utilizado pelo RECIPE.

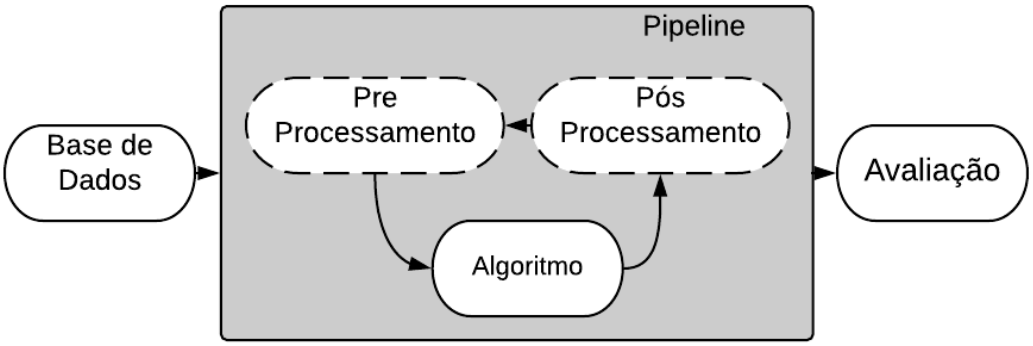


Figura 4.2: Principais componentes de um fluxo, as linhas tracejadas representam componentes opcionais.

4.1 Descrição da Gramática

Em métodos baseados em GGP, a gramática aparece como uma forma eficaz de representar o conhecimento sobre o problema a ser resolvido. Como apresentado na seção 2.2, uma gramática G é representada por uma 4-tupla $<N, T, P, S>$, onde N representa um conjunto de não-terminais, T um conjunto de terminais , P o conjunto de regras de produção e S (um membro de N) o símbolo inicial. As regras de produção definem a linguagem que a gramática representa ao combinar os símbolos gramaticais.

No RECIPE, a gramática representa um conjunto de componentes de fluxos que pode ser usado para resolver um problema de classificação. Conforme mencionado anteriormente, um fluxo é uma sequência de tarefas que devem ser aplicadas a um determinado conjunto de dados para produzir soluções para um determinado problema alvo. Note que os fluxos produzidos são específicos para os dados de entrada.

O primeiro passo para criação da gramática é definir quais etapas do processo de aprendizado de máquina devem ser incluídas na gramática. Os sistemas propostos anteriormente dividiram os fluxos em três etapas principais: pré-processamento de dados, processamento de dados e pós-processamento de dados [Olson et al., 2016b]. Nós também seguimos esses passos básicos, ilustrados na Figura 4.2. Esta abordagem inclui várias opções de componentes que podem ser considerados na criação dos fluxos. Embora seja sempre bom ter escolhas dentro de uma estrutura Auto-ML, quanto maior o número de componentes, maior o tamanho do espaço de busca. Ao mesmo tempo, um número limitado de opções pode não refletir em fluxos apropriados. As próximas seções abordam com mais detalhe as três etapas definidas pela nossa gramática.

4.1.1 Pré Processamento

As etapas de pré-processamento de dados incluem fazer transformações nos dados de entrada para torná-los mais adequado para a tarefa de aprendizado, uma vez que a grande maioria dos dados não é gerada com a tarefa de aprendizado em mente. Ao todo, o pré- processamento conta com 25 componentes que incluem normalização de dados, seleção de atributos, imputação de dados, entre outros. Nós dividimos a fase de pré-processamento em quatro grandes partes:

- **Imputação(1)**: algoritmos utilizados para transformar a base de dados completando os valores faltantes [Pedregosa et al., 2011].
- **Normalizadores(5)**: algoritmos para limitar os limites dos atributos da base de dados [Pedregosa et al., 2011].
- **Modificadores de espaço de entrada (18)**: algoritmos utilizados para se modificar o espaço de atributos da base de dados, incluindo algoritmos de construção e seleção de atributos [Pedregosa et al., 2011].
- **Binarizador(1)**: algoritmo utilizado para definir os atributos da base de dados como 0 ou 1, dependendo de um limiar [Pedregosa et al., 2011].

4.1.2 Classificação

O segundo passo constitui o núcleo do fluxo e o único componente obrigatório. Ele envolve a escolha do algoritmo de classificação e seus parâmetros. Entre os métodos que podem ser utilizados estão aqueles que geram diferentes tipos de modelos de conhecimento, como Naive Bayes, SVM, árvores de decisão, redes neurais, entre outros. O RECIPE soma ao final 22 componentes que podem ser utilizados para a classificação. Para definir os parâmetros a serem otimizados pelo algoritmo, nós fizemos uma combinação dos parâmetros utilizados pelo TPOT e pelo AutoSKLearn, introduzindo pequenas modificações. As próximas seções apresentam todos os classificadores que podem ser utilizados pelo RECIPE, categorizados de acordo com o tipo de modelo que produzem.

4.1.2.1 Árvores

Nessa categoria temos os classificadores que usam árvores de decisão e suas variantes para realizar a tarefa de classificação. Uma árvore de decisão tem por objetivo criar um modelo que prevê o valor de uma variável, aprendendo regras de decisão simples

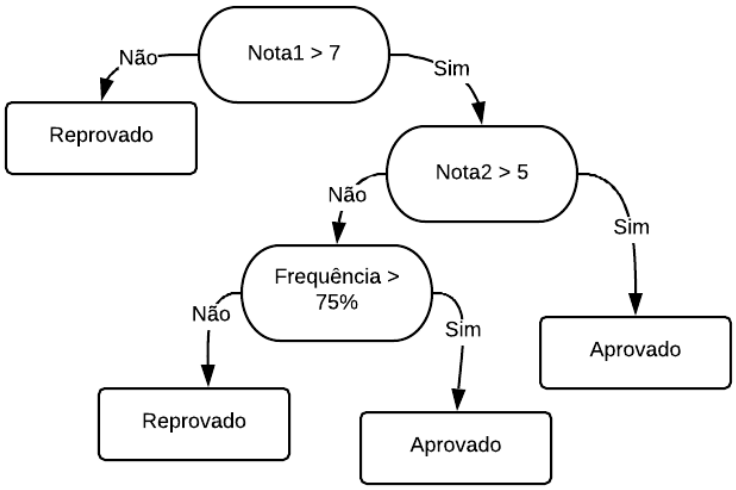


Figura 4.3: Exemplo de árvore de decisão para definir a aprovação ou não de um aluno.

inferidas a partir da base de dados [Song & Ying, 2015]. A Figura 4.3 exemplifica uma árvore de decisão.

Ao total, temos cinco algoritmos que geram árvores de decisão implementadas no ScikitLearn e que são utilizadas pelo RECIPE, o classificador clássico utilizando árvores de decisão (DecisionTree), uma versão da árvore de decisão randomizada (ExtraTree), uma floresta de árvores de decisão (RandomForest), uma versão que representa uma floresta de árvores de decisão randomizadas (ExtraTrees) e uma outra técnica que também faz uso de árvores de decisão (GradientBoost).

4.1.2.2 Bayesianos

Os classifiadores bayesianos são um conjunto de algoritmos de aprendizado baseados na aplicação do teorema de Bayes com a suposição "*naive*" de independência entre cada par de atributos [Pedregosa et al., 2011].

O RECIPE faz uso de cinco versões distintas de implementação desse tipo de classificador que podem ser encontrados na biblioteca do ScikitLearn: Gaussian Naive Bayes, Multinomial Naive Bayes, Bernoulli Naive Bayes, Linear Discriminant Analysis e Quadratic Discriminant Analysis.

4.1.2.3 SVM

Nesse algoritmo, plotamos cada elemento da base de dados como um ponto no espaço n-dimensional (onde n é o número de atributos que a base possui), com o valor de

cada atributo sendo o valor de uma determinada coordenada. Então, realizamos a classificação encontrando o hiperplano que diferencia as classes [Pedregosa et al., 2011].

O RECIPE implementa duas versões de SVM em seu código: SVC (*Support Vector Classifier*) e NuSVC. O NuSVC tem seu funcionamento similar ao SVC, porém, ele possui um parâmetro que torna possível controlar o número de vetores de suporte.

4.1.2.4 Modelos Lineares

Um conjunto de métodos no qual o valor objetivo deve ser uma combinação linear das variáveis de entrada. A maioria desse modelos são comumente utilizados na regressão, porém, o ScikitLearn implementa também alguns desses modelos como classificadores [Pedregosa et al., 2011].

São implementados cinco modelos como componentes do RECIPE: LogisticRegression, LogisticCV (*Cross-Validation*), RidgeClassifier, RidgeCV (*Cross-Validation*), PassiveAggressive e Perceptron.

4.1.2.5 Modelos baseados em Vizinhança

Nos algoritmos de classificação que utilizam a vizinhança como base, a classe é calculada a partir do voto da maioria simples dos vizinhos mais próximos de cada ponto: um ponto de consulta é atribuído à classe de dados que possui mais representantes dentro dos vizinhos mais próximos do ponto [Pedregosa et al., 2011].

Nesse contexto o RECIPE apresenta três componentes: KNeighbors Classifier, Radius Neighbors Classifier e Centroid Classifier.

4.1.3 Pós Processamento

Finalmente, o estágio de pós-processamento pode ser aplicado quando mais de um algoritmo de classificação é escolhido no segundo passo, e seus resultados podem ser combinados. Tanto o AutoSklearn quanto o TPOT realizam uso de técnicas de pós-processamento que, quando utilizadas, aumentam consideravelmente o espaço de busca do algoritmo. No caso do AutoSklearn temos o comitê de classificadores (*ensemble*), e no TPOT uma técnica conhecida como empilhamento (*stacking*). Para aumentar o espaço de busca do RECIPE decidimos implementar ambas as técnicas em nosso *framework*. O funcionamento de ambas sera melhor exemplificado ao decorrer dessa seção.

4.1.3.1 Empilhamento

O empilhamento (*stacking*) é utilizado pelo TPOT e pelo RECIPE quando gera-se um fluxo com dois ou mais métodos de classificação, conforme mencionado anteriormente. Como escolha de implementação, Olson et al. [2016a] utiliza apenas o último elemento do fluxo para realizar a classificação da base de dados, enquanto os outros métodos de classificação do fluxo são utilizados no *stacking*. A Figura 4.4 demonstra o funcionamento do *stacking* em um fluxo, independente do pré-processamento dos dados.

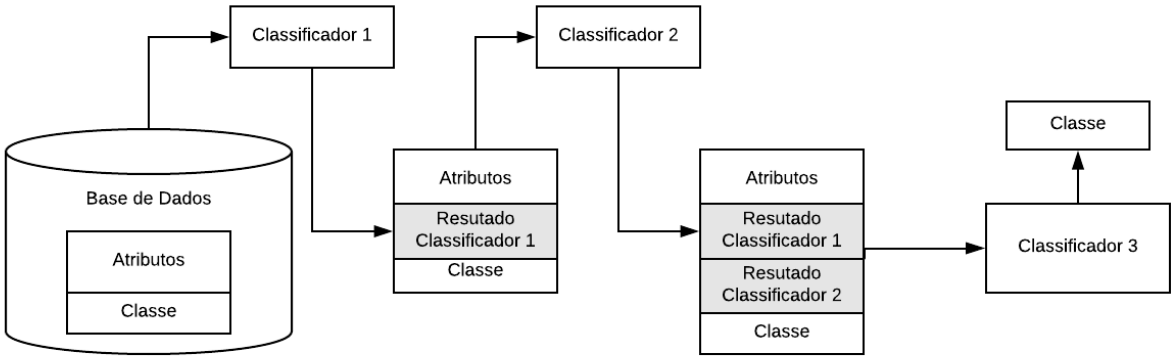


Figura 4.4: Exemplo de *stacking* sendo aplicado para um fluxo com 3 métodos de classificação.

Na Figura 4.4, o Classificador 1 recebe a base de dados com seus atributos e classe para classificação. O algoritmo é treinado e executado e para cada exemplo da base de dados, gerando um resultado que dá origem a uma nova coluna a base de dados, anexada na posição anterior à coluna indicando a classe. A base de dados modificada é dada como entrada para o Classificador 2, que no caso do exemplo da Figura 4.4 repete o mesmo processo que ocorreu com o Classificador 1 e gera uma nova base de dados modificada para servir de entrada ao método de classificação seguinte. O último classificador terá como entrada uma base de dados contendo, além dos atributos originais da base de dados, o resultado da classificação de cada chamada de método de classificação precedente como um novo atributo.

A grande vantagem do *stacking* é adicionar ao fluxo o conceito de meta-aprendizado [Witten et al., 2011], onde o algoritmo aprende levando em consideração os resultados dos classificadores anteriores. Uma desvantagem do *stacking* é o crescimento da base de dados. Como a cada chamada de classificador a base de dados cresce, se o *stacking* for realizado com muitos métodos de classificação, a demanda por

recursos necessários para a execução do fluxo (memória e tempo de processamento) irá crescer sem trazer um ganho efetivo no resultado. Atualmente o RECIPE aceita em sua gramática no máximo 5 algoritmos usando *stacking*.

4.1.3.2 Comitê de Classificadores

Um comitê de classificadores é um conjunto de classificadores cujas decisões individuais são combinadas (tipicamente por votação ponderada ou não ponderada) para classificar novos exemplos [Dietterich et al., 2000]. No RECIPE, a decisão do comitê é tomada através do voto de maioria [Pedregosa et al., 2011]. A Figura 4.5 mostra um exemplo da utilização do voto majoritário como métrica para definição de classe.

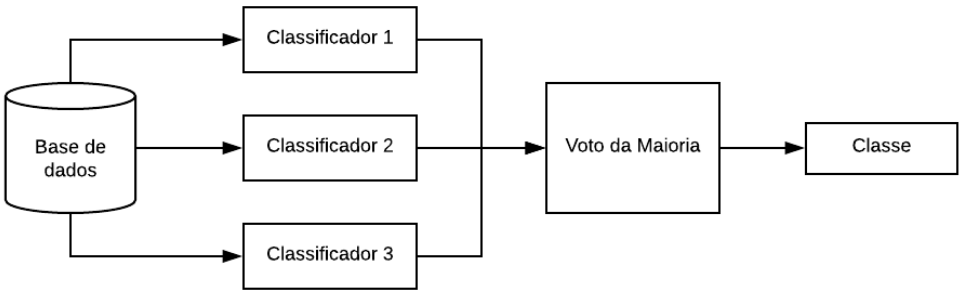


Figura 4.5: Exemplo do voto da maioria sendo aplicado para um fluxo com 3 métodos de classificação.

Atualmente o RECIPE admite no máximo 15 classificadores como parte do comitê, e faz uso apenas do voto da maioria para realizar a combinação no pós processamento.

O RECIPE ainda apresenta outros quatro métodos baseados em comitês de classificadores que são usados como componentes individuais da gramática, mas não são utilizados no pós processamento. Três foram citados anteriormente na sub-seção 4.1.2.1 (RandomForestClassifier e ExtraTressClassifier) e o outro é o AdaBoostClassifier (nele você cria um comitê de classificadores com apenas um algoritmo base e o classificador define os pesos) [Pedregosa et al., 2011].

4.2 Representação da Gramática

A Figura 4.6 apresenta uma amostra da gramática produzida usando o forma de Backus-Naur (BNF), onde cada uma das regras de produção tem a forma $<Start>::=$ $[<Pre\ -\ processing>]\ <Algorithm>$. Os símbolos envolvidos em "$<\ >$"representam não-terminais, e os símbolos especiais "|", “ [] "e"() "representam uma escolha, um

símbolo opcional e um conjunto de símbolos agrupados que devem ser usados em conjunto. A gramática proposta tem 147 regras de produção, num total de 146 terminais e 239 não-terminais. Uma versão completa da gramática pode ser encontrada no Anexo A ou on-line[1]. Detalhes sobre os métodos implementados são definidos na API do SciKit-Learn[2].

```
<Start> ::= [<Pre-processing>] <Algorithm>
<Pre-processing> ::= [<Imputation>] <DimensionalityDefinition>
<Imputation> ::= Mean | Median | Max
<DimensionalityDefinition> ::= <FeatureSelection>   [<FeatureConstruction>]
                        [<FeatureSelection>] <FeatureConstruction>
<FeatureSelection> ::= <Supervised> | <Unsupervised>
<Supervised> ::= SelectKBest <K> <score> | VarianceThreshold | [...]
<score> ::= f-classification | chi2
<K> ::= 1 | 2 | 3 | [...] | NumberOfFeatures - 1
<perc> ::= 1 | 2 | 3 | [...] | 99
<Unsupervised> ::= PCA | FeatureAgglomeration <affinity> | [...]
<affinity> ::= Euclidian | L1 | L2 | Manhattan | Cosine
<FeatureConstruction> ::= PolynomialFeatures
<Algorithm> ::=  <NaiveBayes> | <Trees> | [...]
<NaiveBayes> ::= GaussianNB | MultinomialNB | BernoulliNB
<Trees> ::= DecisionTree | RandomForest | [...]
[...]
```

Figura 4.6: Amostra da gramática definida.

Nossa abordagem proposta, RECIPE, vai na mesma direção do GP-ML, no sentido de que ele só permite a geração de fluxos válidos [Křen et al., 2017]. No entanto, em vez de usar uma programação genética fortemente tipada (STGP) [Montana, 1995] , o RECIPE realiza sua busca através de um algoritmo de programação genética baseada em gramática. Assim, o RECIPE usa uma gramática para organizar o conhecimento adquirido da literatura sobre como são parecidos fluxos bem sucedidos para a tarefa de aprendizado de máquina. A gramática evita a geração de fluxos inválidos e pode acelerar a busca.

Além disso, o RECIPE trabalha com um maior espaço de busca de fluxos maior que os métodos mencionados no capítulo 3.

A Tabela 4.1 resume o número total de componentes principais de cada método relacionado e compara-os com a versão mais atual do RECIPE, mostrando também a interseção (usando o símbolo $\bigcap$) e as diferenças (usando o símbolo #) entre os métodos e a nova versão do RECIPE. A maior diferença é encontrada entre o RECIPE e o Auto-WEKA, 23 componentes distintos. Isso ocorre porque o Auto-WEKA utiliza o WEKA como biblioteca base enquanto o RECIPE utiliza o Scikit-Learn. Já comparando apenas métodos que fazem uso do Scikit-Learn, o Auto-SKLearn é o que apresenta uma

[1]https://github.com/RecipeML/Recipe/tree/master/grammars
[2]http : //scikit-learn.org/stable/modules/classes.html

Tabela 4.1: Comparando trabalhos relacionados ao RECIPE em termos de componentes principais (pré-processamento, classificação e pós-processamento).

Algoritmos	Pre-Process	Process	Post-Process	Total	$\bigcap$	#
RECIPE	25	22	2	49	47	-
Auto-WEKA	2	28	2	32	9	23
Auto-SKLean	24	16	1	41	31	7
PMF-Auto-ML	2	11	0	13	13	0
TPOT	19	11	1	31	28	3
GP-ML	2	9	0	11	9	2

diferença maior de componentes em relação ao método proposto, 7 componentes. Observe que o RECIPE possui o maior número de componentes em relação a todos os outros métodos (49), que inclui métodos de pré-processamento, algoritmos de classificação (processamento) e estratégias pós-processamento (que foram incluídas para este trabalho. Embora isso torne a busca mais desafiadora, também dá a oportunidade de encontrar uma maior variedade de fluxos.

Quando comparado a outros métodos propostos na literatura, o RECIPE não usa qualquer esquema de inicialização sofisticado, como Auto-SKLearn, e os resultados mostram que o algoritmo encontra seu caminho a partir de uma inicialização aleatória. Ele também considera uma função de aptidão de objetivo único, e não parece ser propenso à superposição. O TPOT, por sua vez, adiciona a complexidade do fluxo à sua *fitness*. Finalmente, embora possa usar métodos de pré-processamento de dados diferentes em sequência, o RECIPE não os considera em paralelo, como o TPOT faz.

4.3 Representação do Indivíduo

Conforme mencionado anteriormente, os indivíduos representam fluxos de aprendizado de máquina focados na tarefa de classificação. Esses indivíduos são gerados a partir da gramática usando um conjunto de etapas de derivação. A Figura 4.7 mostra um exemplo de um indivíduo criado derivando as regras de produção da gramática apresentada na Figura 4.6. Nesse caso, a partir do símbolo *Start* da gramática, que inicializa o *Pipeline*, O não-terminal opcional *Pre-processing* é selecionado juntamente com a não-terminal de classificação *Algorithm*. Na etapa seguinte, o opcional *Imputation* é adicionado à árvore, juntamente com $DimensionalityDefinition$. $Imputation$ é substituído pelo terminal $Mean$, e do $DimensionalityDefinition$ selecionamos $FeatureSelection$ seguido de *Supervised* e $VarianceThreshold$. Da $DimensionalityDefinition$, o símbolo opcional $FeatureConstruction$ também é escolhido com $PolynomialFeatures$ como seu terminal. Com a etapa de pré-processamento

definida, o algoritmo $NaiveBayes$ é selecionado, seguido do terminal $GaussianNB$.

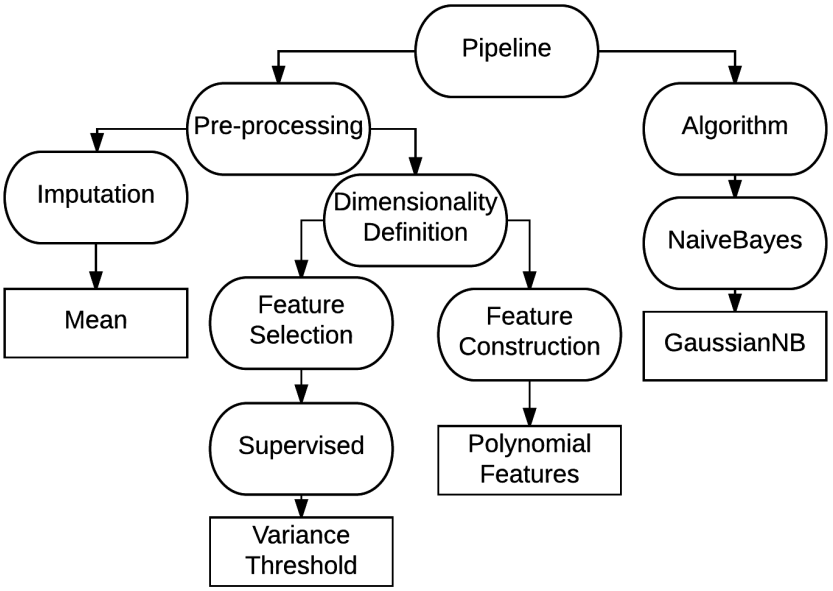


Figura 4.7: Exemplo de indivíduo gerado seguindo as regras de produção da gramática desenvolvida.

4.4 Operadores Genéticos

A seção 2 apresentou os principais operadores genéticos de um algoritmo evolucionário. O RECIPE implementa o cruzamento, a mutação e o elitismo.

A Figura 4.8 mostra as duas maneiras de relacionar os operadores de mutação e cruzamento que foram testadas nesse trabalho. Uma das maneiras foi ter o operador de mutação dependente do operador de cruzamento, isto é, a mutação ocorria apenas se o cruzamento não ocorresse. Essa forma de relação entre os operadores levou a uma convergência prematura e uma deficiência em relação a quantidade de indivíduos distintos gerados em toda a execução. Para solucionar esse problema de busca, a relação entre esses operadores foi modificada, de forma que os indivíduos possam passar tanto pelo cruzamento quanto pela mutação.

O funcionamento e um exemplo dos operadores de cruzamento e mutação são apresentado nas sub-seções a seguir. O elitismo funciona da maneira convencional que foi apresentada na seção 2.

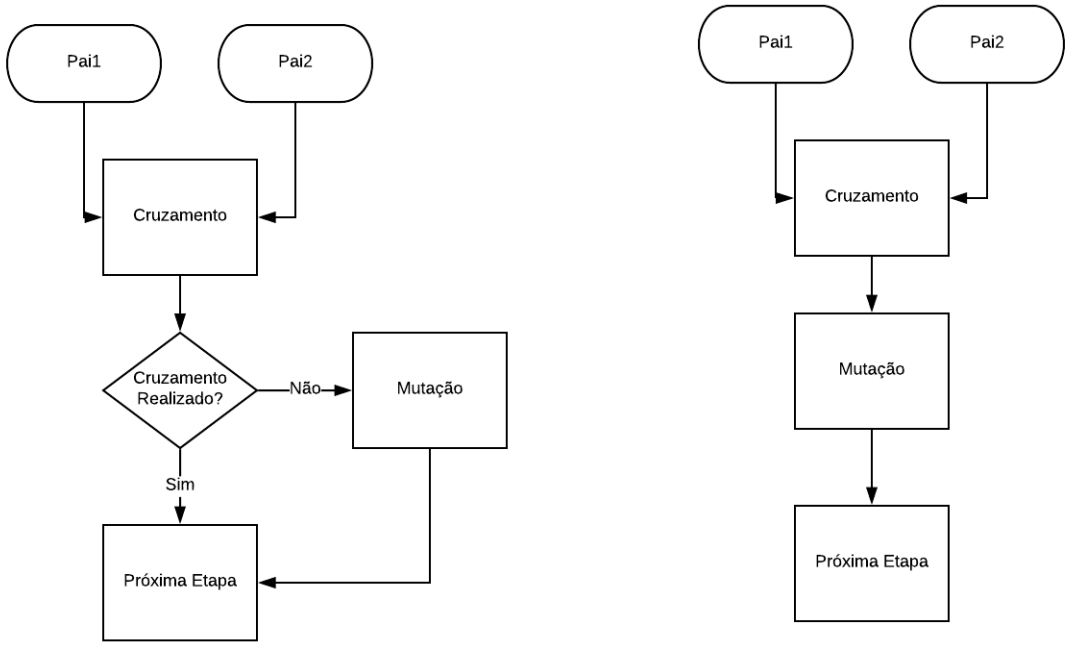


Figura 4.8: Relação de cruzamento e mutação testados pelo RECIPE.

4.4.1 Cruzamento

O cruzamento segue o funcionamento básico de um cruzamento de programação genético baseado em gramática. O algoritmo seleciona aleatoriamente um nó da árvore do primeiro pai, e busca por esse mesmo símbolo (terminal ou não-terminal) na árvore do segundo pai. Se ele não existir, um novo símbolo é selecionado. Caso contrário, ocorre a troca das sub-árvores entre os pais selecionados, gerando dois novos indivíduos.

A Figura 4.9 mostra dois indivíduos que foram selecionados para o cruzamento. Os operadores na cor cinza são os nós da árvore que são comuns entre os dois pais e aqueles que podem servir como ponto de cruzamento. Caso o ponto de cruzamento selecionado seja o nó *Algorithm*, os indivíduos apresentados na Figura 4.10 são gerados.

O cruzamento realizado dessa maneira garante que as restrições da gramática serão obedecidas, e apenas indivíduos válidos serão gerados.

4.4.2 Mutação

A mutação irá escolher um nó aleatoriamente de um indivíduo selecionado para, a partir dele, gerar uma nova sub-árvore respeitando as regras da gramática. A Figura 4.11 mostra um exemplo da mutação ocorrendo em um indivíduo dentro do RECIPE.

Nesse caso o ponto de mutação escolhido foi o $DimensionalityDefinition$. A partir desse ponto, foi gerada uma nova sub-árvore respeitando as regras de formação

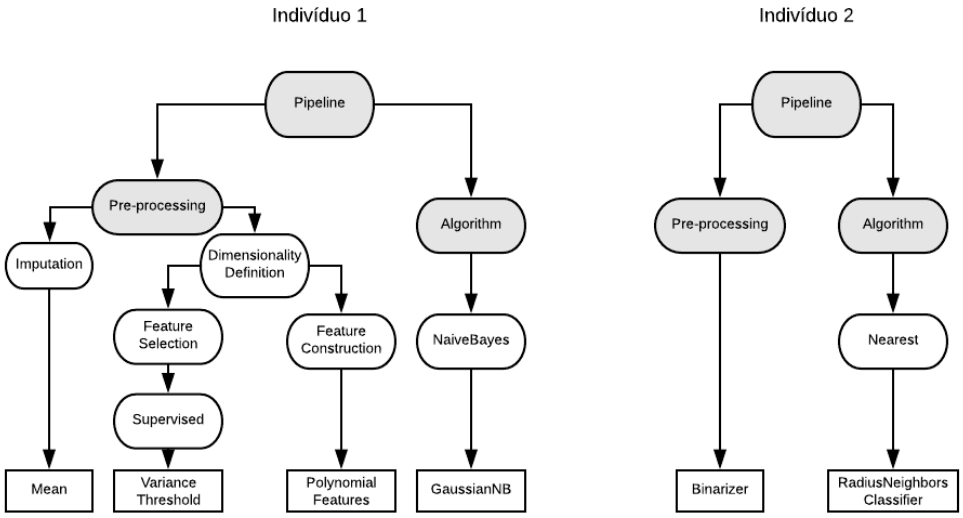


Figura 4.9: Dois indivíduos selecionados pelo RECIPE para o cruzamento.

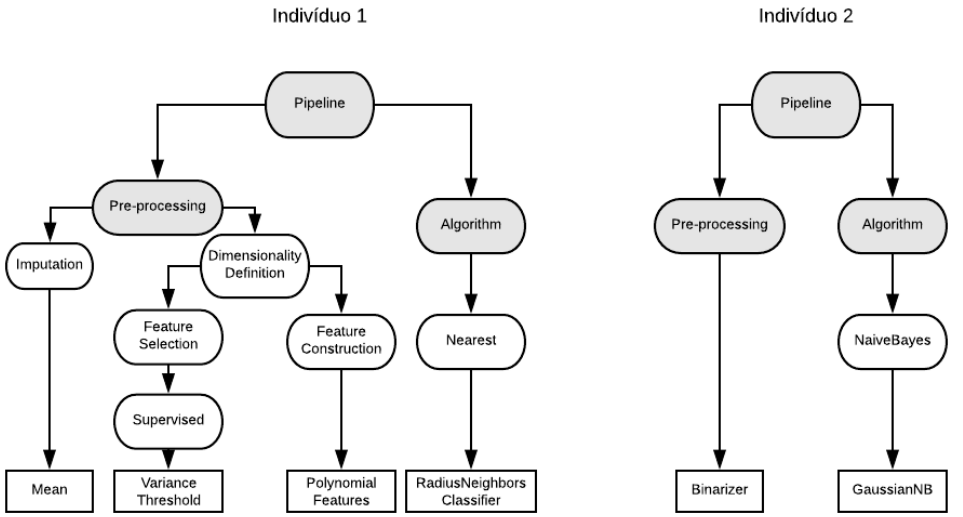


Figura 4.10: Resultado do cruzamento dos indivíduos da Figura 4.9 realizado com o ponto de cruzamento *Algorithm*.

da gramática, e essa nova sub-árvore substituiu a anterior, gerando um novo indivíduo.

4.5 Avaliação do Indivíduo

Lembre-se de que cada indivíduo é uma representação do fluxo de classificação de dados, e sua avaliação envolve a execução do fluxo em uma amostra do conjunto de dados de interesse. Portanto, o primeiro passo do processo de avaliação é gerar um fluxo exe-

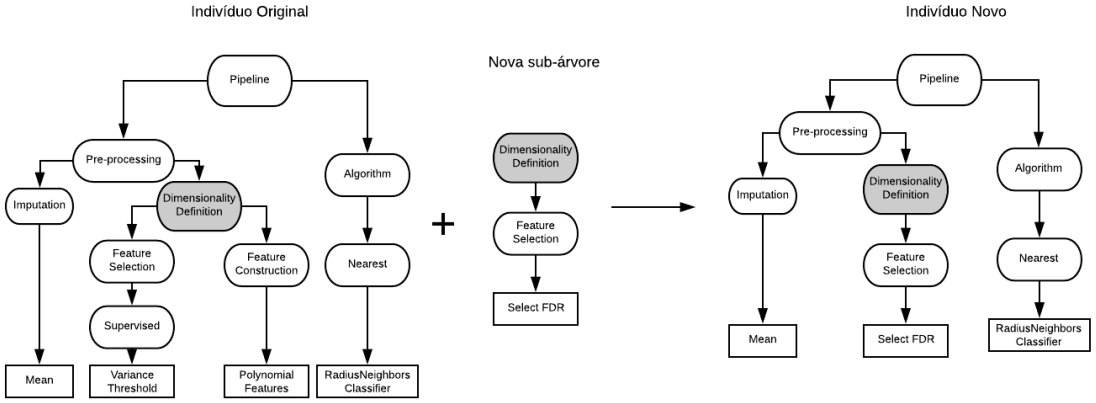


Figura 4.11: Exemplo da mutação de um indivíduo executada pelo RECIPE.

cutável a partir da representação do indivíduo. Este fluxo é gerado usando a biblioteca Python SciKit-Learn [Pedregosa et al., 2011]. Esta biblioteca é interessante porque oferece um amplo número de métodos para todas as fases incluídas na gramática, e nos permite gerar fluxos fáceis de ler que podem ser executados posteriormente por não especialistas para extrair informações de seus dados. Os fluxos são gerados lendo os nós folha do indivíduo, mostrados dentro dos quadrados na Figura 4.7. A Figura 4.12 mostra como ocorre a tradução do indivíduo da representação de árvore para código em Python.

Após este processo de tradução da árvore de derivação de gramática para um código executável, o fluxo é executado usando um procedimento de validação onde, a partir da base de dados de entrada, dados de treinamento e validação são gerados. Durante esta fase, cada conjunto de treinamento é usado para gerar o modelo representado pelo fluxo. Além disso, os conjuntos de treinamento e validação são reamostrados a cada cinco gerações para evitar a sobreposição.

A métrica F1, usada como função de *fitness*, é a média harmônica entre as métricas de precisão e de revocação definidas em um problema de classificação. Tomemos P como a precisão e R como revocação, e a métrica F1 é definida como:

$$F1 = 2 * \frac{P * R}{P + R}$$

A precisão é calculada dividindo o número de exemplos corretamente classificados para uma determinada classe em todos os exemplos que foram atribuídos a essa classe. Revocação, por sua vez, divide o número de exemplos corretamente classificados para uma determinada classe pelo número de exemplos que realmente pertencem a essa classe. Esta métrica foi escolhida porque lida bem com desbalanceamento de classes.

```
#Pipeline automatically generated using RECIPE

from sklearn.pipeline import make_pipeline

from sklearn.preprocessing import LabelEncoder

import numpy as np
import pandas as pd

from sklearn.preprocessing import Imputer
from sklearn.feature_selection import VarianceThreshold
from sklearn.preprocessing import PolynomialFeatures
from sklearn.naive_bayes import GaussianNB

def pipeline(dataTraining,dataTest):


        #Load the training and test datasets:
        training_df = pd.read_csv(dataTraining, header=0, delimiter=",")
        test_df = pd.read_csv(dataTest, header=0, delimiter=",")

        #Apply a filter if the data has categorical data
        #(sklean does not accept this type of data):
        objectList = list(training_df.select_dtypes(include=['object']).columns)
        if ('class' in objectList and len(objectList)>=1):
                training_df = training_df.apply(LabelEncoder().fit_transform)
                test_df = test_df.apply(LabelEncoder().fit_transform)

        #Get the feature data and the class for training:
        train_data = training_df.ix[:,:-1].values
        train_target = training_df["class"].values

        step0 = Imputer (strategy="mean")
        step1 = VarianceThreshold()
        step2 = PolynomialFeatures(degree=2, interaction_only=False, include_bias=True)
        step3 = GaussianNB()

        methods = [step0,step1,step2,step3]

        pipeline = make_pipeline(*methods)

        return pipeline
```

Figura 4.12: Tradução do indivíduo da Figura 4.7 em código.

4.6 Detalhes de Implementação

O RECIPE foi implementado usando a biblioteca *Libgges* [Whigham et al., 2015], e está disponível para download no GitHub [3].

É importante notar que o RECIPE sempre irá gerar fluxos válidos, mas alguns fatores de execução podem propositalmente tornar esse fluxo inválido. A única forma do RECIPE pode gerar uma solução inválida é se um fluxo atingir um tempo limite definido pelo usuário, onde o valor de fitness 0 será atribuído a esse indivíduo. Assim,

[3]https://github.com/laic-ufmg/Recipe

a geração de fluxos válidos porém altamente complexos, que demandam um tempo de execução extremamente alto, pode gerar indivíduos com fitness 0.

A biblioteca *Libgges* foi modificada para aceitar novos parâmetros de entrada: tempo limite de execução, semente do gerador aleatório, arquivo de treino e de teste e qual a métrica de avaliação será utilizada. A estrutura da chamada dos operadores genéticos também foi modificada.

Capítulo 5

Resultados Experimentais

Este capítulo relata os resultados da execução de experimentos do RECIPE em duas fases. Na primeira fase foram utilizados 10 base de dados para realizar uma prova de conceito do algoritmo e comparar seus resultados com outros dois algoritmos estado-da-arte. Na segunda fase, nós utilizamos o conhecimento adquirido a partir dos resultados da fase anterior para modificar o *framework*, corrigindo os problemas percebidos nos primeiros experimentos, aumentando a versatilidade do algoritmo e melhorando os resultados. Nessa etapa, além das 10 base de dados já testadas, também foram adicionadas 21 novas bases.

5.1 Primeira Fase

Na primeira fase os experimentos foram realizados em 10 bases de dados, incluindo cinco benchmarks clássicos da UCI (University of California Irvine) [Asuncion & Newman, 2007] e cinco conjuntos de dados de bioinformática introduzidos em [Freitas et al., 2011; Souto et al., 2008; Wan et al., 2015]. Esses conjuntos de dados de bioinformática trazem problemas reais sobre a longevidade, o reparo do DNA e a expressão genética carcinogênica. A Tabela 5.1 apresenta as principais características dos conjuntos de dados, incluindo o número de instâncias (inst.), o número de atributos (Atrib.), a quantidade de classes, os tipos de atributos e a presença de valores ausentes.

Nessa fase todos os experimentos foram executados usando uma validação cruzada de 10 partições (*folds*) com três repetições, cada execução reporta a medida F1 do melhor fluxo encontrado. Os resultados relatados nesta seção correspondem à média e aos desvios padrões obtidos para as 30 execuções no conjunto de teste. Todos os resultados foram comparados usando um teste Wilcoxon Signed-Rank [Wilcoxon et al., 1970] com 5% de significância.

Tabela 5.1: Bases de Dados utilizadas na primeira fase dos experimentos.

Base	**# Inst.**	**# Atrib.**	**Tipos Atrib.**	**# Classes**	**Ausente?**
Breast-Cancer (BC)	286	9	Numérico (Inteiro)	2	Sim
Car Evaluation (CAR)	1,728	6	Nominal	4	Não
CaeNãorhabditis Elegans (CE)*	478	765	Binário	2	Não
Chen-2002 (CHEN)*	179	85	Real	2	Não
Chowdary-2006 (CHOW)*	104	182	Real	2	Não
Credit-G (CRED)	1,000	20	Numérico/Nominal	2	Não
Drosophila Melanogaster (DM)*	104	182	Real	2	Não
DNA-No-PPI-T11 (DNA)*	135	104	Numérico/Nominal	2	Sim
Glass (GLS)	214	9	Real	7	Não
Wine Quality-Red (WQR)	1,599	11	Real	10	Não

O simbolo * indica base de dados da bioinformática.

Para esses experimentos, o RECIPE foi executado usando os seguintes parâmetros: população de 100 indivíduos que evoluíram por 100 gerações, seleção por torneio de tamanho dois, elitismo de cinco indivíduos e probabilidades de cruzamento e mutação uniformes de 0,9 e 0,1, respectivamente. Se o melhor indivíduo permanece o mesmo durante mais de cinco gerações, interrompemos o processo evolutivo e retornamos seu respectivo fluxo. O TPOT também foi executado com esses mesmos parâmetros, exceto o número de gerações que foi configurado para 40, o que corresponde ao maior número de gerações executadas pelo RECIPE nos experimentos. O tempo limite para execução de cada fluxo no TPOT foi de 5 minutos. Para o Auto-SKLearn um único parâmetro foi definido: o tempo máximo de execução, que foi configurado para uma hora para cada uma das 30 execuções, resultando em 30 horas por conjunto de dados. Este número foi definido após medirmos o tempo médio dos outros dois algoritmos em cada conjunto de dados.

A fim de fazer uma comparação justa com os sistemas anteriormente propostos na literatura, para os testes originais, também organizamos os componentes presentes nos espaços de busca Auto-SKLearn e TPOT em duas outras gramáticas. A Tabela 5.2 resume o número de componentes considerados em cada uma das três etapas principais dos fluxos (i.e, pré processamento, processamento e pós processamento), destacando suas semelhanças e diferenças. Observe que todos os componentes do TPOT estão presentes na gramática usada pelo RECIPE, enquanto dois dos componentes do Auto-SKLearn não estão presentes no RECIPE: a extração do autovetor generalizado e o *OneHotEncoder* por não serem componentes que eram encontrados na versão do SciKitLearn utilizada nesses experimentos. O tamanho do espaço de busca foi calculado considerando o número de combinações possíveis de todos os parâmetros discretos da gramática. Os parâmetros contínuos, que podem gerar um número infinito de combinações, foram desconsiderados nesta análise. Observe que o espaço de busca cresce

Tabela 5.2: Comparação da gramática proposta com outras geradas a partir dos blocos de construção usados por métodos anteriores, organizados em uma gramática.

Componentes	RECIPE	Auto-SKLearn	TPOT
Pre-processamento	33	20	20
Processamento	23	17	12
Post-processamento	0	1	0
Interseção	-	35	32
Diferenças	-	3	0
Tam. Espaço de Busca	$\approx 4.10 \times 10^{34}$	$\approx 2.47 \times 10^{17}$	$\approx 7.53 \times 10^{7}$

significativamente do TPOT para Auto-SKLearn e RECIPE.

5.1.1 Análise do Processo Evolutivo do RECIPE

Os resultados na seção anterior mostraram que os valores de F-measure obtidos pelo RECIPE podem ser tão bons quanto ou melhores do que os obtidos pelos métodos estado-da-arte, com garantia de gerar sempre soluções válidas. Esta seção investiga o processo evolutivo e soluções gerados pelo RECIPE. Conforme discutido anteriormente, os resultados da fitness da evolução de diferentes fluxos não variam significativamente e, com poucas exceções, estão dentro de um pequeno intervalo. O RECIPE apresentou uma média de execução de 15 gerações para as execuções nas bases de dados da Tabela 5.1.

A Figura 5.1 mostra a medida F1 dos melhores e piores indivíduos da população no conjunto de treinamento, bem como a média de aptidão para todos os indivíduos. Observe que os valores das médias podem ser reduzidos por alguns indivíduos com fitness 0, que são aqueles que excedem o limite de tempo para o modelo de classificação a ser construído. As flutuações nos valores do melhor indivíduo são devidas à estratégia de reamostragem de dados, implementada para evitar o *overfitting*.

Ao observar a Figura 5.1 podemos notar que existe uma convergência prematura do valor médio da medida F1 entre todos os indivíduos um pouco antes da geração 10. Isso mostra que existe uma convergência prematura e mesmo a reamostragem evitando *overfitting* o problema da diversidade existe. Esse fato é outra confirmação que o RECIPE não esta explorando de maneira adequada o espaço de busca. Nas gerações finais podemos perceber que até o pior indivíduo passa a ter o mesmo valor de *fitness* do melhor e a média dos valores é equivalente ao melhor, isso nos leva a crer que após a geração 10 a população é formada em grande parte por apenas um fluxo.

Além disso, a Figura 5.2 mostra a distribuição de algoritmos e métodos de pré-

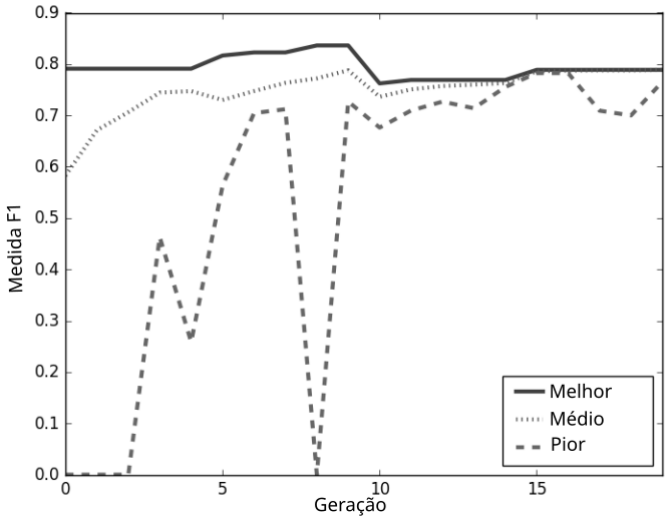
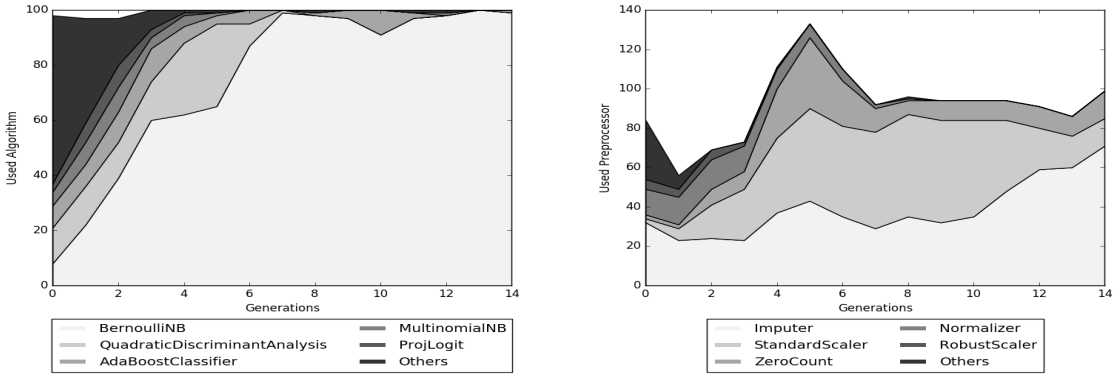


Figura 5.1: Curva de convergência do RECIPE para a base de dados DNA ao longo das gerações.



(a) Classificação (b) Pré-processamento

Figura 5.2: Cobertura do RECIPE em estágios de classificação e pré-processamento ao longo das gerações para o conjunto de dados DNA.

processamento ao longo das gerações. Observe que, muito rapidamente, um método de classificação acaba ultrapassando os outros, enquanto os métodos de pré-processamento têm uma maior variação. Embora o ponto de evolução em que isso acontece varie de um conjunto de dados para o outro, observamos que a primeira combinação de pré-processamento, algoritmo e parâmetros que funciona um pouco melhor do que as outras provavelmente estará presente na solução final. Mais uma vez, isso abre oportunidades para incluir no método uma variada gama de mecanismos de diversidade e especiação. Essa perda de diversidade é uma das razões pelas quais os métodos apresentam resultados às vezes também semelhantes aos obtidos por uma busca aleatória.

5.1.2 Resultados e comparações com outros métodos estado-da-arte

Antes de entrar em detalhes nos resultados da medida F1 obtida por todos os métodos considerados, mostramos os números que motivaram o desenvolvimento de RECIPE: a geração de soluções inválidas pelo TPOT. A Figura 5.3 mostra o número de soluções inválidas geradas pelo TPOT em cada geração, considerando uma população de 100 indivíduos para três conjuntos de dados: *car*, *glass* e *breast cancer*. Observe que, nas primeiras gerações, pelo menos 10 % das soluções são inválidas para todos os conjuntos de dados, com a *glass* atingindo 30 %. À medida que a evolução avança, esses números variam substancialmente, mas as soluções inválidas nunca desaparecem. O RECIPE, por outro lado, sempre garante que todas as soluções geradas sejam válidas de acordo com a gramática fornecida. A única maneira do método gerar uma solução inválida é se um fluxo atingir um tempo limite definido pelo usuário, onde o valor de fitness 0 será atribuído a esse indivíduo.

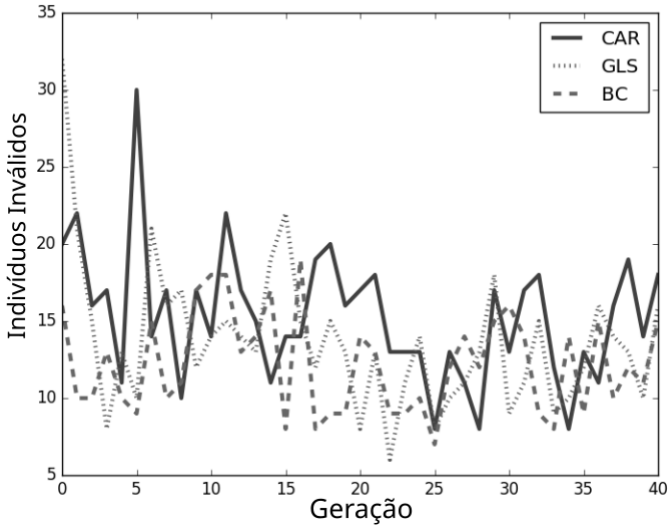


Figura 5.3: fluxos inválidos produzidos pelo TPOT a cada geração do GP.

A Tabela 5.3 mostra os resultados da medida F1 obtida pelo RECIPE, e sua comparação com os resultados do TPOT, Auto-SKLearn, e com uma estratégia completamente aleatória (RAND) usando a gramática proposta. Nessa estratégia, o número de fluxos gerados é equivalente ao número máximo de indivíduos avaliados durante o processo de busca do RECIPE (1500). Esta última comparação foi motivada pelos estudos de Mantovani et al. [2015], Bergstra et al. [2011] e Bergstra & Bengio [2012], eles mostraram que uma busca aleatória é suficiente para otimizar os componentes dos

algoritmos de aprendizado de máquina em vários domínios. Nessa comparação testamos a vericidade da afirmação e quão melhor é o resultado de uma busca aleatória quando uma "direção"é adicionada.

Na Tabela 5.3, o símbolo ↑ denota uma variação positiva estatisticamente significativa para o método nessa coluna em relação ao RECIPE e ↓ uma variação negativa estatisticamente significativa em relação ao RECIPE de acordo com o teste Wilcoxon Signed Rank.

Tabela 5.3: Comparação dos valores obtidos da medida F1 pelo RECIPE e os baselines no conjunto de teste.

Bases de Dados	RECIPE	Auto-SKLearn	TPOT	RAND
BC	0.939 (0.04)	0.940 (0.04)	0.942 (0.04)	0.936 (0.05)
CAR	0.991 (0.01)	0.990 (0.01)	0.999 (0.00)↑	0.994 (0.01)↑
CE	0.486 (0.08)	0.444 (0.12)	0.475 (0.09)	0.484 (0.08)
CHEN	0.959 (0.04)	0.933 (0.06)↓	0.951 (0.05)	0.942 (0.06)↓
CHOW	0.993 (0.03)	0.977 (0.05)	1.000 (0.00)	0.986 (0.03)
CRED	0.793 (0.03)	0.806 (0.04)↑	0.802 (0.03)↑	0.789 (0.03)
DM	0.758 (0.12)	0.756 (0.09)	0.759 (0.10)	0.754 (0.11)
DNA	0.827 (0.07)	0.824 (0.04)	0.788 (0.07)↓	0.786 (0.18)
GLS	0.741 (0.09)	0.734 (0.08)	0.789 (0.09)↑	0.745 (0.08)
WQR	0.642 (0.05)	0.642 (0.03)	0.679 (0.03)↑	0.636 (0.04)

Os resultados na Tabela 5.3 mostram que em 7 de 20 comparações os resultados do RECIPE apresentam diferenças estatísticas quando comparado ao Auto-SKLearn e ao TPOT. Os dois casos em que RECIPE foi melhor do que esses métodos estão nos conjuntos de dados de bioinformática CHEN e DNA, respectivamente. Considerando os conjuntos de dados da UCI, o TPOT apresentou os melhores resultados em quatro casos, CAR, CRED, GLS e WQR. O Auto-SKLearn, por sua vez, foi estatisticamente melhor no conjunto de dados do CRED. Ao comparar o RECIPE a abordagem aleatória, ambos com um espaço de busca idêntico, em 8 de cada 10 comparações não vemos diferença estatística, enquanto a busca aleatória teve um resultado melhor em um conjunto de dados UCI (CAR) e o RECIPE melhorou a saída de classificação em um conjunto de dados do mundo real (CHEN).

Esses resultados preliminares, embora interessantes, podem ser enviesados. Isso ocorre porque os espaços de busca dos métodos (com exceção do aleatório e do RECIPE) são diferentes, e isso pode representar uma vantagem para o TPOT, que possui o menor espaço de busca. Pode parecer contra intuitivo, mas um espaço de busca maior não necessariamente é melhor. Um quantidade maior de fluxos inadequados podem ser gerados para uma base de dados específica o que significa um aumento considerável de complexidade na criação da gramática e otimização dos parâmetros de execução, entre

outros problemas advintos da escalabilidade.

Assim, em um segundo experimento, comparamos a eficácia do mecanismo de busca do RECIPE ao usar gramáticas que abrangem o conjunto de soluções válidas que podem ser geradas pelos baselines (ver Tabela 4.1), tornando a comparação entre os métodos mais natural.

A Tabela 5.4 mostra os resultados do RECIPE com os espaços de busca do TPOT (RECIPE-TP) e Auto-SKLearn (RECIPE-AS), bem como os resultados de soluções geradas aleatoriamente com esses mesmos espaços de busca. Podemos observar que os resultados do RECIPE melhoraram quando comparados aos relatados na Tabela 5.3. Quando comparado ao Auto-SKLearn, o RECIPE apresenta resultados estatisticamente superiores em 4 de 10 conjuntos de dados. No caso do TPOT, é melhor no conjunto de dados de DNA. Os resultados mais surpreendentes, no entanto, são novamente os obtidos pela busca aleatória. Em apenas dois casos os resultados do RECIPE foram estatisticamente melhores com a gramática do TPOT, e em apenas um caso ao se utilizar a gramática do Auto-SKLearn. Uma maneira de olhar para isso é que os componentes da gramática são sempre adequados para a tarefa, portanto, as variações na medida F1 devem estar dentro de um intervalo predefinido, que não é muito grande.

Tabela 5.4: Resultados da medida F1 obtidos pelo RECIPE e o método aleatório ao usar gramáticas que englobam espaços de pesquisa do TPOT e do Auto-SKLearn.

Bases	**TPOT**			**Auto-SKLearn**		
	RECIPE-TP	**TPOT**	**RAND**	**RECIPE-AS**	**Auto-SKLearn**	**RAND**
BC	0.944 (0.04)	0.942 (0.04)	0.906 (0.09)↓	0.949 (0.04)	0.940 (0.04)	0.935 (0.04)↓
CAR	1.000 (0.00)	0.999 (0.00)	1.000 (0.00)	0.998 (0.00)	0.990 (0.01)↓	0.999 (0.00)
CE	0.463 (0.08)	0.475 (0.09)	0.451 (0.11)	0.482 (0.07)	0.444 (0.12)↓	0.495 (0.08)
CHEN	0.949 (0.05)	0.951 (0.05)	0.964 (0.05)	0.937 (0.05)	0.933 (0.06)	0.939 (0.06)
CHOW	1.000 (0.00)	1.000 (0.00)	0.988 (0.04)	0.997 (0.01)	0.977 (0.05)↓	0.997 (0.01)
CRED	0.807 (0.04)	0.802 (0.03)	0.803 (0.03)	0.801 (0.03)	0.806 (0.04)	0.799 (0.03)
DM	0.734 (0.09)	0.759 (0.10)	0.709 (0.09)	0.732 (0.11)	0.756 (0.09)	0.743 (0.12)
DNA	0.818 (0.08)	0.788 (0.07)↓	0.798 (0.09)	0.823 (0.08)	0.824 (0.04)	0.806 (0.08)
GLS	0.779 (0.10)	0.789 (0.09)	0.778 (0.09)	0.734 (0.09)	0.734 (0.08)	0.738 (0.12)
WQR	0.685 (0.03)	0.679 (0.03)	0.675 (0.03)↓	0.664 (0.04)	0.642 (0.03)↓	0.660 (0.07)

Para sumarizar os resultados, a Tabela 5.5 mostra uma comparação entre o melhor resultado de classificação absoluta para o RECIPE e todos os outros métodos. Nesta tabela, o símbolo ◇ indica que não há significância estatística entre os métodos comparados. O RECIPE é estatisticamente melhor em 6 dos 20 casos quando comparado aos métodos do estado da arte, e em 15 casos de 30 quando comparado à pesquisa aleatória. Observe também que diferentes versões da gramática apresentaram resultados diferentes. A gramática proposta, que tem o maior espaço de busca, foi melhor em 4 de 10 conjuntos de dados, enquanto a gramática do TPOT foi melhor em

Tabela 5.5: Comparação entre o melhor resultado do RECIPE usando diferentes versões da gramática e os baselines.

Bases			Auto-SK	TPOT	Random		
	Grammar	F-measure			Auto-SK	TPOT	RECIPE
BC	Auto-SKLearn	0.949 (0.04)	◇	◇	↓	↓	↓
CAR	TPOT	1.000 (0.00)	↓	◇	◇	◇	↓
CE	RECIPE	0.486 (0.08)	◇	◇	◇	↓	◇
CHEN	RECIPE	0.959 (0.04)	↓	◇	↓	◇	↓
CHOW	TPOT	1.000 (0.00)	↓	◇	◇	◇	↓
CRED	TPOT	0.807 (0.04)	◇	◇	◇	◇	↓
DM	RECIPE	0.758 (0.12)	◇	◇	◇	↓	◇
DNA	RECIPE	0.827 (0.07)	◇	↓	◇	◇	◇
GLS	TPOT	0.779 (0.10)	↓	◇	↓	◇	↓
WQR	TPOT	0.685 (0.03)	↓	◇	↓	↓	↓

5 e o Auto-SKLearn em um único conjunto de dados. Isso pode indicar que o método ainda não está conseguindo explorar adequadamente um espaço de busca maior. Outra coisa interessante a observar é que, à medida que o espaço de busca aumenta, também aumenta a diferença de desempenho da abordagem aleatória para os outros métodos. Observe que para todas as gramáticas, menos a proposta, os métodos obtiveram resultados melhores do que o aleatório em 4 de 10 conjuntos de dados. À medida que aumentamos o tamanho do espaço de busca com a gramática completa, esse número aumenta de 4 para 7 em 10.

5.2 Segunda Fase

Nessa fase, inicialmente nós tentamos modificar o framework fazendo uso da experiência adquirida através dos experimentos passados. Foram implementadas e testadas técnicas para aumentar a diversidade, o algoritmo foi modificado para aceitar novos parâmetros de entrada por parte do usuário, e foi realizado uma análise mais aprofundada dos componentes de aprendizado de máquina para redefinir a gramática.

Para essa nova etapa de experimentos, foram mantidas as 10 base de dados da etapa anterior e selecionados novas 21. As novas bases foram selecionadas de acordo com o estudo apresentado em [Bischl et al., 2017], onde foi criado um repositório com 100 base de dados da literatura. A Tabela 5.6 apresenta as características principais apresentadas pelas bases de dados, incluindo o número de instâncias (#Inst), o número de atributos (#Atrib), o número de classes (#Classes), a quantidade de valores faltantes (#Ausentes), e o índice de desbalanceamento de classes (Desbalanc.), que é definido pelo número de exemplos da classe minoritária dividido pelo número de exemplos da classe majoritária).

Tabela 5.6: 31 bases de dados utilizadas na segunda fase dos experimentos

id	Base	**# Inst.**	**# Atrib.**	**# Classes**	**#Ausentes**	**Desbalanc.**
1	breast-cancer	699	9	2	16	0.53
2	car evaluation	1,728	6	4	0	0.05
3	caenorhabditis elegans*	478	765	2	0	0.66
4	chen-2002*	179	85	2	0	0.72
5	chowdary-2006*	104	182	2	0	0.68
6	credit-g	1,000	20	2	0	0.43
7	dna-no-ppi-t11*	135	104	2	103	0.32
8	drosophila melanogaster*	119	585	2	0	0.49
9	glass	214	9	7	0	0.12
10	wine quality red	1,599	11	6	0	0.01
11	australian	690	15	2	0	0.80
12	balance-Scale	625	5	3	0	0.17
13	climate-model	540	21	2	0	0.09
14	cmc	1,473	10	3	0	0.53
15	diabetes	768	9	2	0	0.54
16	eucalyptus	736	20	5	448	0.49
17	hill-valley	1,212	101	2	0	1.00
18	ilpd	583	11	2	0	0.40
19	irish	500	6	3	32	0.80
20	kc1	2,109	22	2	0	0.18
21	kr-vs-kp	3,196	37	2	0	0.91
22	monks-problems-2	601	7	2	0	0.52
23	mushroom	8,124	23	2	2,480	0.93
24	ozone-level-8hr	2,534	73	2	0	0.07
25	pc4	1,458	38	2	0	0.14
26	sick	3,772	30	2	6,064	0.07
27	steel-plates-fault	1,941	34	2	0	0.53
28	tic-tac-toe	958	10	2	0	0.53
29	vehicle	846	19	4	0	0.91
30	vowel	990	13	11	0	1.00
31	waveform-5000	5,000	41	3	0	0.98

O simbolo * indica base de dados da bioinformática.

5.2.1 Impacto da Diversidade

O grande problema que o RECIPE apresentou nos experimentos reportados na seção 5.1 foi a baixa diversidade (quantidade de indivíduos únicos em toda a execução do GGP). Para economizar recursos computacionais, definimos uma condição de parada caso o melhor indivíduo se mantivesse o mesmo por um número fixo de gerações. Os experimentos iniciais foram executados com taxas de cruzamento e mutação como 0.9 e 0.1, respectivamente, e população de 100 indivíduos. O RECIPE apresentou uma média de execução de 15 gerações para as execuções nas bases de dados da Tabela 5.1. Isso significa um total de 1500 indivíduos gerados para cada execução. A versão aleatória da execução do RECIPE utilizou essa informação de 1500 indivíduos (15 gerações em média para 100 indivíduos em cada) em uma única geração para realizar os experimentos e comparações.

A Tabela 5.7 mostra os dados de diversidade entre a execução do RECIPE e do aleatório. O alto percentual de indivíduos únicos para a execução do aleatório é

Tabela 5.7: Diversidade da execução usando a gramática completa pelo RECIPE e para o Aleatório nas bases de dados da Tabela 5.1.

Bases	**RECIPE**		**Aleatório**	
	#Unic. Ind.	**%Diversidade**	**#Unic. Ind.**	**%Diversidade**
BC	299	19,3%	996	66,4%
CAR	193	12,8%	992	66,1%
CE	313	20,8%	1002	66,8%
CHEN	249	16,6%	990	66%
CRED	220	14,6%	993	66,2%
DM	388	25,8%	995	66,3%
DNA	376	25,1%	990	66%
GLS	274	18,2%	1002	66,8%
WQR	256	17,1%	996	66,4%

esperado, dado o tamanho do espaço de busca. Porém, essa diversidade também é uma explicação para os bons resultados apresentados. Já a baixa diversidade apresentada pelo RECIPE mostra que o algoritmo pode explorar mais o espaço de busca.

Na tentativa de melhorar a capacidade de busca do *framework*, a condição de parada para caso o melhor indivíduo não se modificasse foi removida. Outras modificações foram feitas com o intuito de gerar diversidade ao algoritmo, incluindo a atualização da gramática e otimização da taxa de mutação. As seções a seguir irão descrever as técnicas utilizadas para auxiliar no aumento da diversidade da busca.

5.2.1.1 Atualização da Gramática

A gramática tem um papel fundamental na formação dos indivíduos e nos operadores genéticos, e a forma como ela é estruturada pode beneficiar algumas regras em detrimento de outras. A gramática foi reformulada tendo como objetivo equilibrar a probabilidade de escolha dos métodos de processamento e pré-processamento. Nessa etapa, também foram adicionadas ao espaço de busca duas técnicas de pós-processamento, encontradas no TPOT e no AutoSklearn, que foram deixados de fora na primeira versão: o empilhamento dos classificadores e o comitê de classificadores.

Fazendo alterações na gramática e adicionando o pós processamento sem modificar as taxas de mutação e de cruzamento, já obtivemos um aumento de diversidade em relação aos valores da Tabela 5.7. A Tabela 5.8 apresenta os novos valores de diversidade encontrados.

A atualização da gramática aumentou o espaço de busca e ajudou na diversidade, e essa mudança foi mantida para os experimentos realizados nas 31 base de dados.

Tabela 5.8: Comparação entre valores de diversidade para a execução do RECIPE com a gramática sem modificações e com a gramática atualizada.

Bases	**RECIPE**		**RECIPE v2**	
	#Unic. Ind.	**%Diversidade**	**#Unic. Ind.**	**%Diversidade**
BC	299	19,3%	411	27,4%
CAR	193	12,8%	517	34,4%
CE	313	20,8%	376	25,1%
CHEN	249	16,6%	375	25,0%
CRED	220	14,6%	426	28,4%
DM	388	25,8%	405	27,0%
DNA	376	25,1%	433	28,8%
GLS	274	18,2%	358	23,9%
WQR	256	17,1%	503	33,5%

5.2.1.2 Otimização de Parâmetros

Mudar a taxa de mutação e cruzamento pode causar um aumento ou queda na diversidade da população. Nessa etapa, escolhemos 5 bases de dados para fazer a otimização dos parâmetros de mutação e cruzamento. As 5 bases de dados escolhidas foram: *DM*, *irish*, *diabetes*, *chen-2002* e a *monks-problem-2*.

Nesse experimento, um valor fixo para a taxa de cruzamento foi definido em 0.7, e a taxa de mutação sofreu variação de 0.1 até 0.5 em intervalos de 0.1. O experimento foi executado em uma população de 100 indivíduos que evoluíram por 100 gerações, seleção por torneio de tamanho dois, elitismo de cinco indivíduos, sem condição de parada caso o melhor indivíduo se mantivesse o mesmo durante uma quantidade pré-definida de gerações.

Alterações iniciais na taxa de mutação não mostraram impacto na diversidade do algoritmo. Assim, realizamos uma análise mais aprofundada de como os operadores genéticos estavam implementados na biblioteca Libgges [Whigham et al., 2015], que foi utilizada para a parte da programação genética do *framework*.

Com a análise do código conseguimos descobrir que o Libgges, na sua implementação original, considera a possibilidade de aplicar a mutação apenas se o cruzamento não ocorrer. E mesmo que ele não ocorra, não é garantido que a mutação vá ocorrer, a não ser que a taxa desse operador seja 1.0. Isto é, com as taxas de mutação e cruzamento atual (cruzamento 0.7 e mutação 0.6), a probabilidade de se ocorrer uma mutação é de 0.18 caso o cruzamento não ocorra, e não existe a possibilidade do indivíduo sofrer cruzamento e mutação simultaneamente.

O código foi modificado de forma a tornar possível as operações de mutação e cruzamento independentes. O resultado é apresentado na Figura 5.4.

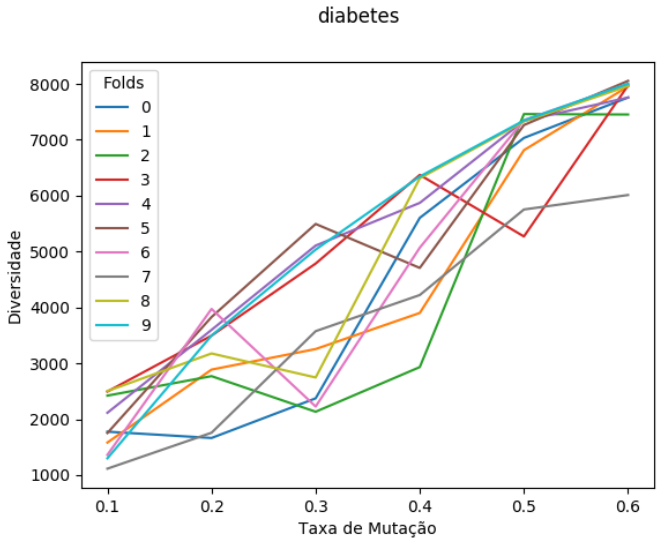


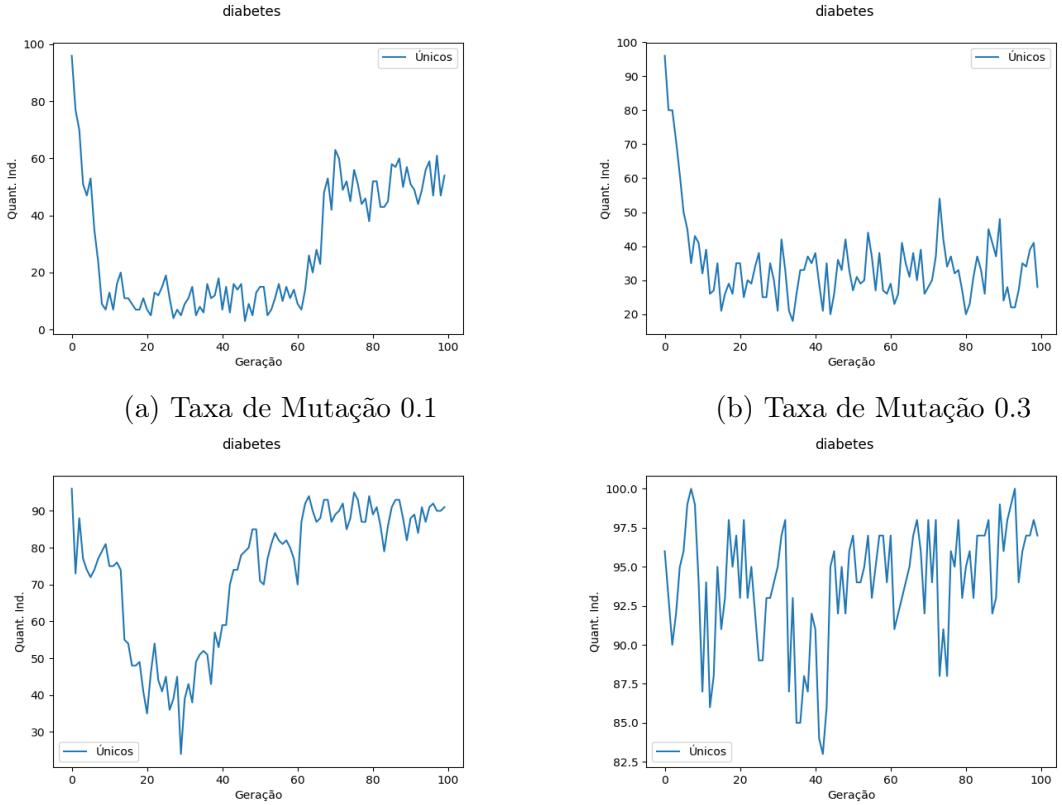
Figura 5.4: Curva da quantidade de indivíduos distintos na execução de um algoritmo do RECIPE ao se variar o valor da mutação na base *diabetes*.

É fácil observar que a mudança do funcionamento do código gerou um aumento considerável na diversidade do algoritmo. A taxa de mutação inicial de 0.1 já apresenta valores de diversidade muito próximos aos encontrados antes de se modificar a estrutura dos operadores, e com uma taxa de mutação de 0.6, em algumas partições são gerados oito mil indivíduos distintos de dez mil possíveis soluções (o que também não é desejado, pois nesse caso a busca passa a ser quase aleatória).

A Figura 5.5 mostra o novo comportamento da população de indivíduos do RECIPE para cada geração com 4 taxas de mutação distintas. Quando se compara a Figura **??** a Figura 5.5a, já é possível perceber um aumento na diversidade e na capacidade que o RECIPE adquiriu de recomeçar a explorar melhor o espaço de busca.

5.2.2 Resultados

Com a mudança da interação entre os operadores, foram definidos os parâmetros para a execução dos testes com as 26 bases restantes. O RECIPE foi executado usando os seguintes parâmetros: 100 indivíduos evoluíram por 100 gerações, seleção por torneio de tamanho três, elitismo de cinco indivíduos e probabilidades uniformes de cruzamento e mutação de 0,7 e 0,3, respectivamente. A versão utilizada do TPOT foi a mais atual do momento (0.9.0) e também foi executado com esses mesmos parâmetros do RECIPE, exceto para as taxas de cruzamento e mutação que foram respectivamente definidas para 0,05 e 0,90 [Olson et al., 2016a,c; Sohn et al., 2017]. O Auto-SKLearn requer um único parâmetro - um máxima de tempo de execução - que foi definido para duas



(a) Taxa de Mutação 0.1

(b) Taxa de Mutação 0.3

(c) Taxa de Mutação 0.4

(d) Taxa de Mutação 0.6

Figura 5.5: Comportamento do RECIPE para a base *diabetes* com 4 taxas de mutação distintas em cada geração.

hora para cada uma das 30 execuções, resultando em 60 horas por conjunto de dados. Também fizemos um experimento com os 129 fluxos que o Auto-SKlearn utiliza na inicialização com *metalearning* executando cada um desses fluxos juntamente com os algoritmos de busca.

Os novos resultados de F1, seguidos de seu desvio padrão, estão listados na Tabela 5.9. O teste de Friedman [Friedman, 1940] foi realizado para comparar os resultados e concluiu que não existe diferença estatística entre os três *frameworks*. Isso mostra que o RECIPE continua obtendo resultados similares as últimas versões dos algoritmos do estado da arte. Porém, quando fazemos uma análise par a par para cada base de dados, o RECIPE é melhor em 8 bases de dados, o TPOT em 5 bases e o AutoSKlearn em 12. A vantagem do AutoSKlearn em relação aos outros algoritmos é exatamente a inicialização por *meta-learning*. Dois resultados interessantes ocorrem para as bases *vehicle* e *vowel*. Nessas bases os resultados do RECIPE são significativamente piores que os apresentados pelo TPOT e pelo AutoSK. Um estudo aprofundado mostra que temos *overfitting* na base *vehicle*. No treino, os fluxos gerados mostram resultados semelhan-

Tabela 5.9: Resultados da médida F1 para 31 bases de dados.

id	Base	RECIPE	TPOT	AutoSKLearn	Meta
1	breast-cancer	**0.9568 (0.037)**	0,9504 (0.030)	0.9501 (0.031)	0.9601 (0.035)
2	car evaluation	**0.9967 (0.003)**	0.9948 (0.002)	0.9965 (0.007)	0,8935 (0.297)
3	caenorhabditis elegans*	0.6089 (0.048)	0.5945 (0.047)	**0.6396 (0.068)**	0.5722 (0.200)
4	chen-2002*	0.9238 (0.045)	0.9605 (0.043)	**0.9656 (0.045)**	0.9605 (0.035)
5	chowdary-2006*	0.9923 (0.023)	0.9911 (0.026)	**1.0000 (0.000)**	1.0000 (0.000)
6	credit-g	0.6775 (0.047)	0.6719 (0,065)	**0.6890 (0.052)**	0.7172 (0.055)
7	dna-no-ppi-t11*	**0.7334 (0.104)**	0.7283 (0.119)	0.6931 (0.114)	0.7611 (0.107)
8	drosophila melanogaster*	**0.6546 (0.115)**	0.6038 (0.135)	0.5643 (0,096)	0.6853 (0.153)
9	glass	0.7578 (0.067)	0.7578 (0.099)	**0.7833 (0.095)**	0.8024 (0.085)
10	wine quality red	**0.6797 (0.024)**	0.6734 (0.032)	0.6788 (0.039)	0.6930 (0.031)
11	australian	0.8565 (0.055)	**0.8739 (0.051)**	0.8682 (0.056)	0.8797 (0.054)
12	balance-Scale	0.9770 (0.018)	0.9931 (0.011)	**0.9964 (0.010)**	0.9845 (0.016)
13	climate-model	**0.8936 (0.029)**	0.8931 (0.027)	0.8894 (0.023)	0.9026 (0.027)
14	cmc	0.5531 (0.035)	0.5502 (0.042)	**0.5628 (0.039)**	0.5562 (0.026)
15	diabetes	0.5192 (0.011)	**0.5234 (0.018)**	0.5134 (0.004)	0.5797 (0.033)
16	eucalyptus	**0.1359 (0.038)**	0.1119 (0.021)	0.1088 (0.014)	0.2338 (0.051)
17	hill-valley	1.0000 (0.000)	1.0000 (0.000)	1.0000 (0.000)	1.0000 (0.000)
18	ilpd	0.6832 (0.046)	0.6785 (0.056)	**0.7160 (0.049)**	0.7485 (0.041)
19	irish	**0.7970 (0.181)**	0.7824 (0.203)	0.7366 (0.257)	0.7887 (0.160)
20	kc1	0.8251 (0.027)	**0.8357 (0.023)**	0.8326 (0.018)	0.8511 (0.027)
21	kr-vs-kp	0.9956 (0.003)	0.9956 (0.004)	**0.9959 (0.002)**	0.9974 (0.001)
22	monks-problems-2	1.0000 (0.000)	1.0000 (0.000)	1.0000 (0.000)	1.0000 (0.000)
23	mushroom	1.0000 (0.000)	1.0000 (0.000)	1.0000 (0.000)	1.0000 (0.000)
24	ozone-level-8hr	0.9289 (0.014)	0.9338 (0.012)	**0.9369 (0,009)**	0.9370 (0.008)
25	pc4	0.9024 (0.029)	**0.9044 (0.020)**	0.9017 (0.025)	0.9139 (0.023)
26	sick	0.9881 (0.006)	0.9879 (0.006)	**0.9899 (0.006)**	0.9901 (0.003)
27	steel-plates-fault	**1.0000 (0.000)**	**1.0000 (0.000)**	0.9994 (0.001)	1.0000 (0.000)
28	tic-tac-toe	**1.0000 (0.000)**	**1.0000 (0.000)**	1.0000 (0.000)	1.0000 (0.000)
29	vehicle	0.2450 (0.095)	0.4004 (0.117)	**0.5011 (0.046)**	0.6067 (0.049)
30	vowel	0.0191 (0.011)	0.1281 (0.260)	0.1062 (0.201)	0.5480 (0.081)
31	waveform-5000	0.8632 (0.015)	**0.8695 (0.012)**	0.8682 (0.008)	0.8704 (0.009)

O simbolo * indica base de dados da bioinformática.

tes ou melhores que os resultados de outros métodos. Porém, quando o fluxo gerado é aplicado nos dados de teste, ele não é capaz de generalizar e apresenta resultados muito inferiores ao esperado.

Outro ponto interessante de se notar é que os algoritmos utilizados pelo AutoSklearn na inicialização por *meta-learning* já apresentam resultados da medida f muito bons, quando não melhores, do que os resultados apresentados pelos algoritmos de busca. Isso nos mostra que os bons resultados do AutoSKlearn em relação aos outros algoritmos estado da arte estão vinculados à essa inicialização, e não ao processo de busca do algoritmo em si.

Para identificar o impacto das mudanças e a evolução do RECIPE, foram comparados os resultados das 10 bases utilizadas nos primeiros experimentos na versão antiga do RECIPE (RecipeV1) e na versão após as modificações (RecipeV2). Para que seja realizada uma comparação mais justa, o indivíduo do RecipeV2 utilizado para a comparação foi aquele da geração 20, já que o RECIPE dos primeiros experimentos

parava sua execução geralmente nessa geração. Os resultados relatados nesta seção correspondem à média e aos desvios padrões obtidos para as 30 execuções no conjunto de teste.

Por se tratarem de apenas dois métodos, os resultados foram comparados usando um teste Wilcoxon Signed-Rank [Wilcoxon et al., 1970] com 5 % de significância em uma comparação par a par, para determinar se houve ou não uma melhora significativa ao RECIPE.

Tabela 5.10: Comparação dos valores da medida F1 obtidos pela primeira versão do RECIPE (RecipeV1) e a versão mais atual (RecipeV2)

Bases de Dados	**RecipeV1**	**RecipeV2**
BC	0.939 (0.04)	0.956 (0.03) ↑
CAR	0.991 (0.01)	0.994 (0.01)
CE	0.586 (0.07)	0.627 (0.07) ↑
CHEN	0.959 (0.04)	0.960 (0.03)
CHOW	0.993 (0.03)	0.997 (0.04)
CRED	0.703 (0.03)	0.694 (0.05) ↓
DM	0.655 (0.12)	0.654 (0.09)
DNA	0.727 (0.07)	0.724 (0.04)
GLS	0.741 (0.09)	0.757 (0.06) ↑
WQR	0.642 (0.05)	0.679 (0.03) ↑

Como podemos observar na Tabela 5.10, as mudanças no RECIPE tiveram impacto positivo nos resultados de 4 bases de dados (BC, CE, GLS e WQR) e em apenas uma base o RECIPE sem as modificações realizadas obtém melhores resultados (CRED). Isso é um indicativo que as nossas modificações conseguem explorar de maneira mais eficaz o espaço de busca.

5.3 Recipe WEB

O principal objetivo por trás da criação do RECIPE é facilitar o uso do aprendizado de máquina por parte de profissionais de outras áreas que não sejam experts em ML. Ainda que o RECIPE já faça a criação do automática do pipeline de aprendizado de máquina, o usuário deve ter uma noção básica de programação para realizar a execução correta do framework.

RECIPE Web é uma iniciativa para auxiliar aqueles usuários que não possuem conhecimento nenhum na área de programação e gostariam de fazer uso do RECIPE. No serviço, basta o usuário ter em mãos a base de dados que deseja classificar em formato CSV e um pipeline será gerado e executado automaticamente após a execução do algoritmo. A Figura 5.6 mostra o caso de uso simplificado do RECIPE Web.

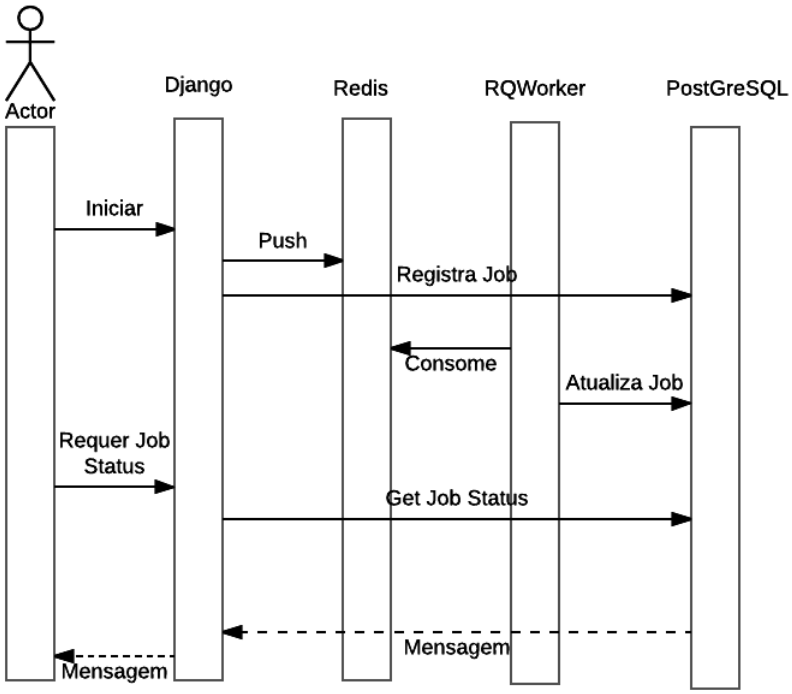


Figura 5.6: Caso simplificado de uso do RECIPE Web

O usuário faz uma requisição de serviço para o servidor *back-end*, que por sua vez coloca o serviço em uma fila de tarefas e registra o serviço no banco de dados. Um robô assíncrono irá checar a fila de tarefas e executar o RECIPE quando houver algum trabalho para ser executado. Enquanto o algoritmo é executado, ele atualiza as informações da tarefa no banco de dados para que o usuário tenha um feedback do andamento da execução. Ao término da execução, o robô faz uma última atualização no banco de dados em relação a tarefa e retorna a para checar a fila de execução na espera da próxima tarefa.

A arquitetura do RECIPE Web foi pensada de forma a ser escalável e caso ocorram grandes mudanças no framework do RECIPE o serviço consiga acompanhar sem grandes dificuldades. A experiência do usuário é um fator determinante para implementação desse serviço, pois deve-se levar em consideração que o público alvo do RECIPE são aqueles usuários com pouca ou nenhuma experiência na área de aprendizado de máquina. Todas as tecnologias utilizadas devem ser preferencialmente, assim como o RECIPE, de código aberto. Quando é levado em consideração as informações apresentadas, é possível criar a arquitetura apresentada na Figura 5.7.

O Django foi escolhido por ser um framework de desenvolvimento rápido para web e altamente escalável. O Postgre e o Redis são bancos de dados que se comunicam muito

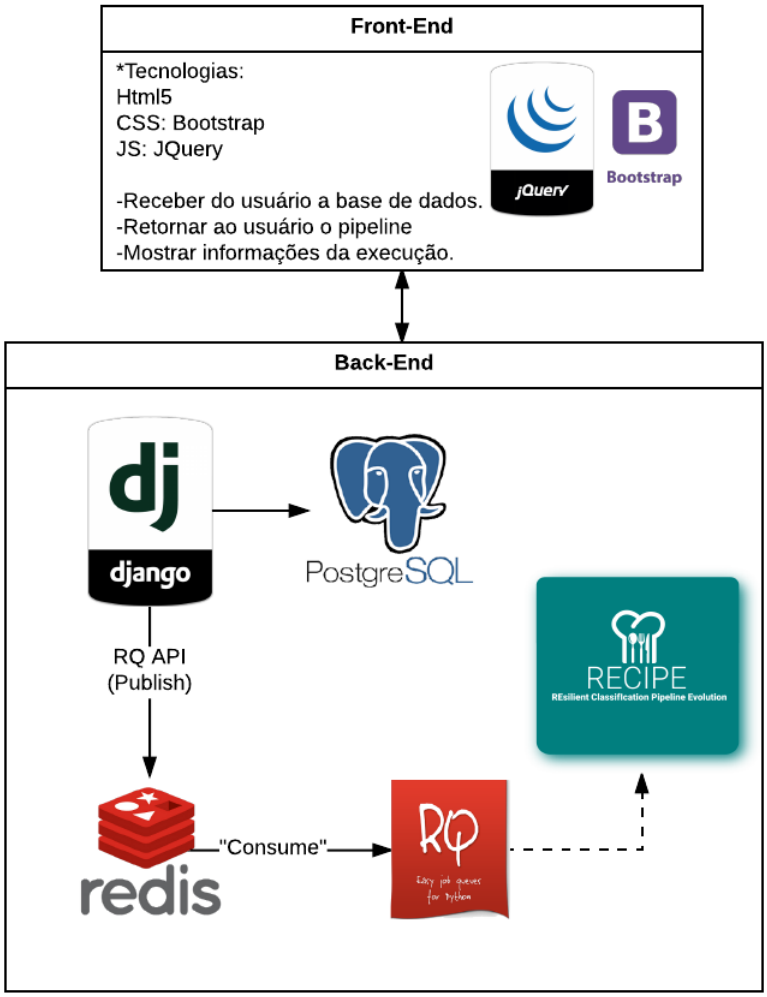


Figura 5.7: Arquitetura e tecnologias utilizadas no RECIPE Web

bem com o Django e foram uma escolha de projeto para facilitar o desenvolvimento de forma que o foco da implementação fosse mantido na experiência o usuário.

Em sua forma mais simples de execução o RECIPE Web requer apenas um arquivo contendo a base de dados utilizada na geração do algoritmo de classificação. Para usuários avançados, é possível configurar outros parâmetros de execução como entrar com uma base de dados de treino e outra de teste, incluir a semente para as chamadas das funções geradoras de dados aleatórios, a taxa de mutação, a taxa de cruzamento, o tamanho da população e a quantidade de gerações. Ao final da execução o usuário terá acesso ao resultado do melhor pipeline encontrado para a sua base de dados, os valores das métricas de precisão, acurácia, revocação e a medida F1 para o treino e teste, um gráfico mostrando o comportamento do GP durante o treino e o teste com informação do melhor, pior e a média dos indivíduos por geração. O RECIPE Web

pode ser encontrado no link http://recipe-web.speed.dcc.ufmg.br.

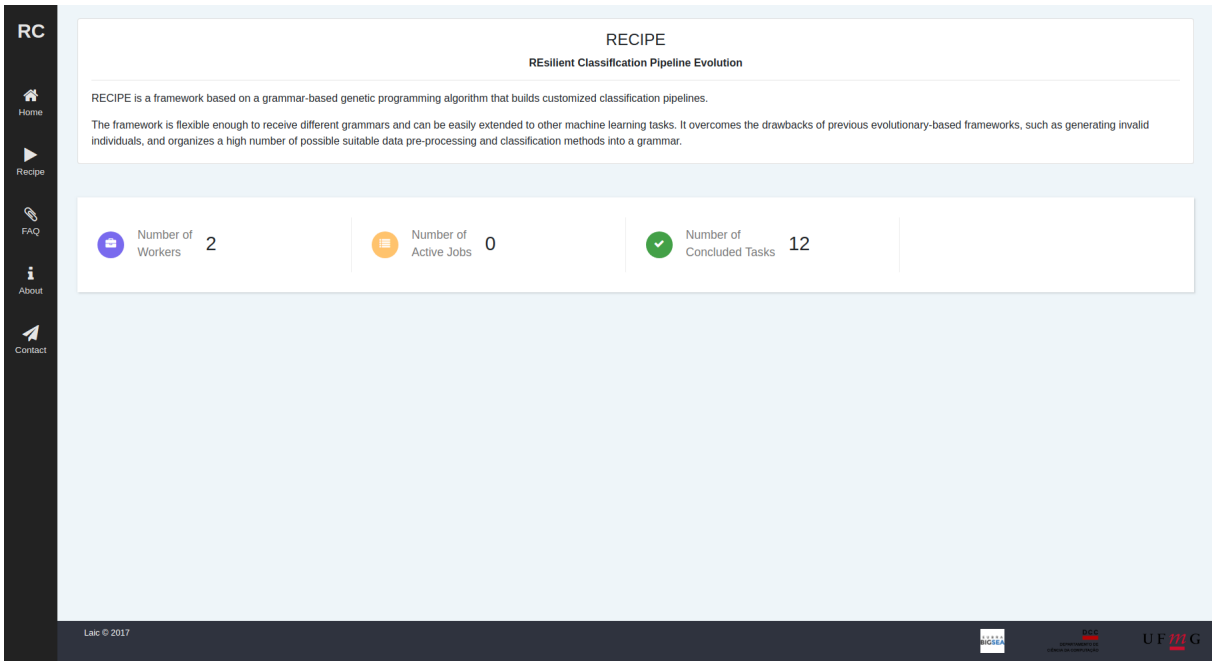


Figura 5.8: Página inicial do RECIPE Web

A Figura 5.8 mostra a página inicial do serviço web. Ao clicarmos na opção "Recipe"no menu lateral a esquerda somos redirecionados para a página da chamada do serviço.

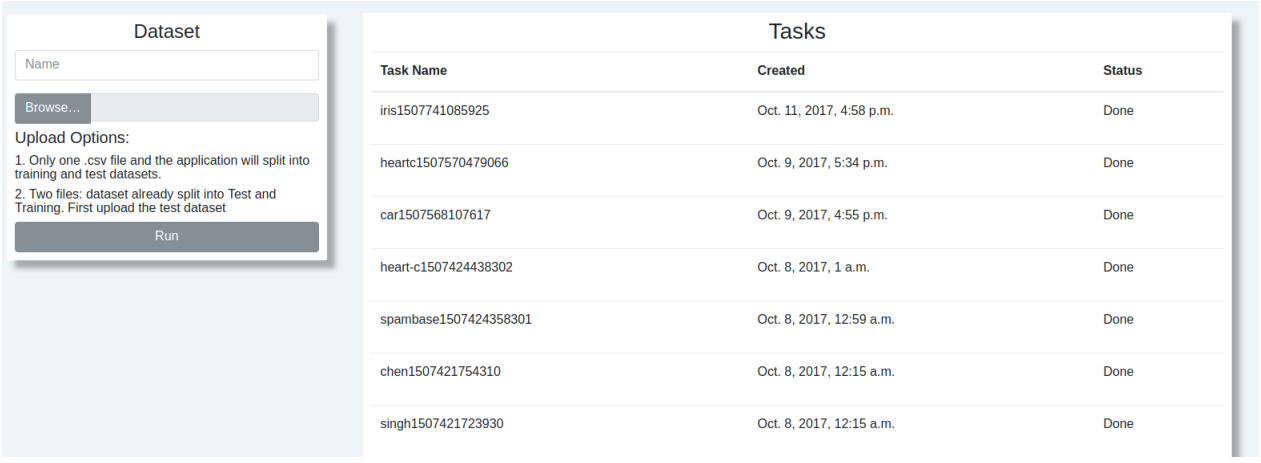


Figura 5.9: Página de execução do RECIPE no RECIPE Web

Na Figura 5.9 é apresentada a página para a chamada e execução do RECIPE. A esquerda temos o formulário para a entrada de dados. Nessa versão inicial, são aceitos como entrada um ou dois arquivos em formato ".csv". Se for inserido apenas um arquivo o próprio algoritmo realiza a divisão da base de dados entre treino e teste. Foi assumido

que a última coluna do arquivo é o atributo da classe que desejamos classificar. Na direita temos uma lista de tarefas que estão sendo realizadas pelo RECIPE Web, é nessa parte onde o usuário é capaz de ver o andamento do trabalho. Quando a tarefa for marcada como concluída, o usuário pode clicar na linha correspondente a mesma para ter acesso as informações da execução e o *pipeline* gerado.

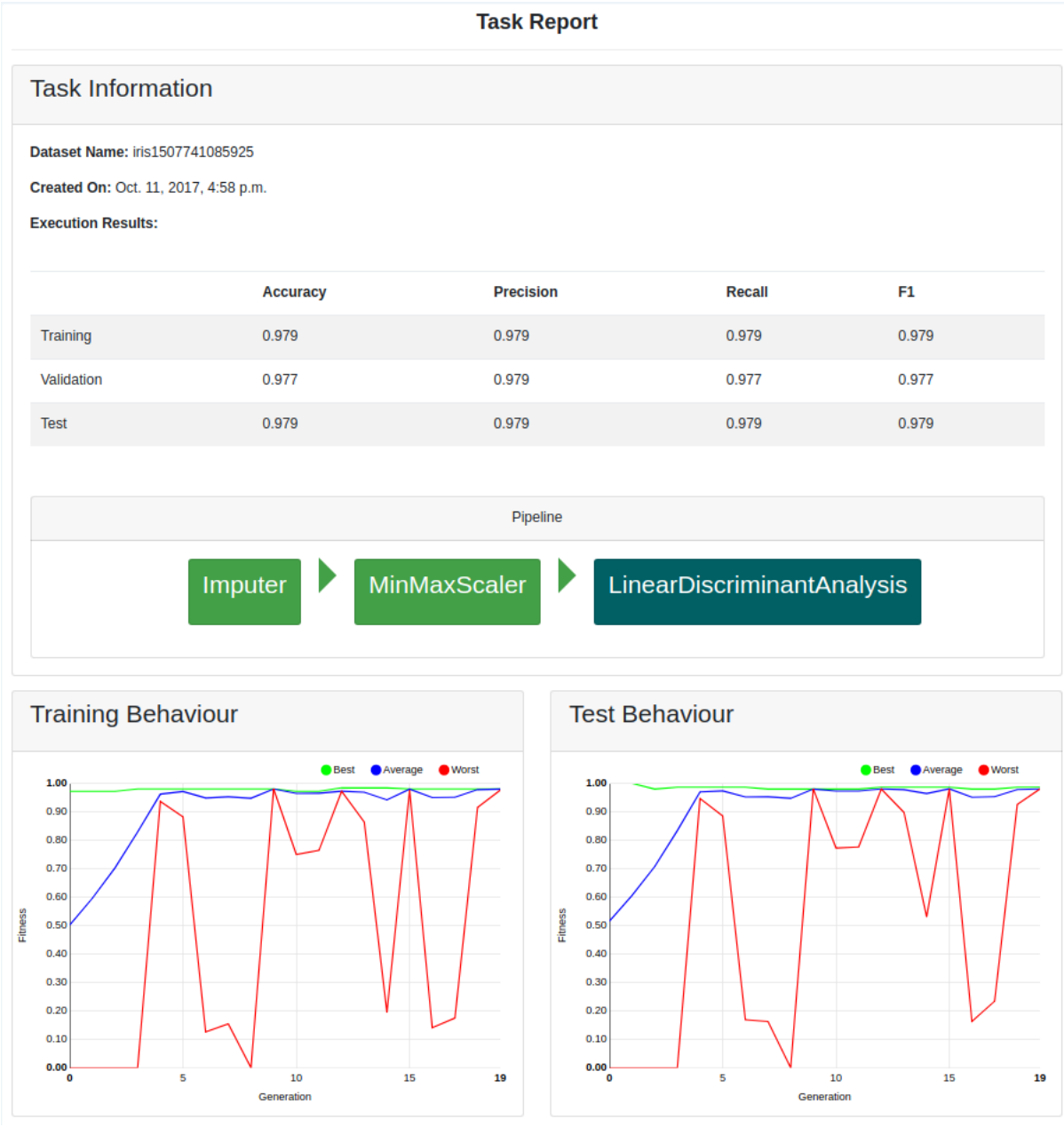


Figura 5.10: Relatório da tarefa executada pelo RECIPE Web

Ao fim de cada tarefa, o RECIPE Web gera um relatório para o usuário. Nesse relatório temos o nome da base de dados, quando a tarefa foi criada, o resultado das execuções para as partições de treino, validação e teste, é apresentado o melhor *pipeline*

que o RECIPE criou para essa base de dados e o comportamento tanto no treino quanto no teste. A Figura 5.10 mostra um exemplo do relatório da tarefa para a base de dados *iris*.

Capítulo 6

Conclusões e Trabalhos Futuros

Este trabalho introduziu o RECIPE, um *framework* de programação genética baseada em gramática concebido para gerar fluxos completos de tarefas de aprendizado de máquina para resolver problemas de classificação. O método introduz uma gramática ao processo de busca do algoritmo, que organiza conhecimento prévio na literatura e evita a geração e avaliação de soluções inválidas. A estrutura do *framework* é flexível o suficiente para lidar com diferentes conjuntos de gramáticas, e pode ser facilmente estendida para resolver outras tarefas de aprendizado de máquina além da classificação, como regressão e agrupamento.

Os resultados mostram que o método resolve o problema de gerar soluções inválidas agregando conhecimento de especialistas à busca através da gramática. Porém, ainda são gerados uma quantidade pequena de fluxos muito complexos que acabam fazendo uso irregular de recursos de memória e tempo de execução, o que ao final pode ser considerado uma solução inválida. Em torno de 1% de todos os indivíduos gerados pelo RECIPE em todas as execuções acabam sendo inválidos.

O RECIPE mostrou ser tão bom ou melhor do que as abordagens propostas anteriormente na literatura e, ainda que o *framework* no momento não consiga substituir totalmente um especialista na hora de se gerar fluxos de aprendizado de máquina, já é possível fazer uso do mesmo para dar uma direção ao tipo de modelo necessário para resolver o problema em mãos.

Percebemos que, ao se utilizar gramáticas diferentes baseadas nos componentes do TPOT e do AutoSK, conseguimos uma melhora nos resultados mesmo reduzindo o espaço de busca. Os resultados indicam que o crescimento substancial do espaço de busca introduzido pela gramática proposta no novo método exige a implementação de mecanismos mais sofisticados dentro do RECIPE para aumentar a diversidade e garantir uma cobertura mínima do espaço de busca. Isso mostra também a dependência

que alguns algoritmos de aprendizado de máquina apresentam em relação aos seus parâmetros de execução. Um estudo mais aprofundado dos operadores mostrou a importância das taxas de mutação e de cruzamento, assim como a forma como eles ocorrem tem um impacto muito grande no funcionamento desse tipo de algoritmo.

Com a criação do RECIPE Web foi possível identificar os desafios existentes ao se implementar um sistema de Auto-ML para usuários não experientes. Deve-se considerar não apenas quais informações serão mostradas ao usuário, mas também a forma de se mostrar os resultados para que esses possam ser facilmente compreendidos por um leigo na área. Ao mesmo tempo, os resultados devem continuar sendo significantes para um especialista.

6.1 Trabalhos Futuros

Como trabalhos futuros é primordial a atualização e manutenção do RECIPE para se manter no nível dos métodos estado da arte. Entre as tarefas que podemos citar, estão: adicionar mais algoritmos e técnicas de pós-processamento ao espaço de busca, revisar a gramática, experimentar técnicas para ajudar no aumento da diversidade.

Com a evolução do RECIPE, é inerente a evolução do RECIPE Web. Além de ter como objetivo facilitar o uso do *framework* por profissionais sem experiência na área de aprendizado de máquina, O RECIPE Web tem um segundo objetivo: agregar informações sobre características de bases de dados executadas pelos usuários e os melhores fluxos gerados para cada uma delas. Essa informação poderia enriquecer o RECIPE, que poderia recomendar fluxos para novas base de dados baseado em similaridade quando o usuário tem pouco tempo computacional disponível, sem a necessidade de reexecutar o RECIPE. Também é possível evoluir o serviço adicionando novos parâmetros de execução avançados, como por exemplo, tornando possível que o usuário escolha quais algoritmos ele quer ou não na gramática utilizada na execução.

Uma outra melhoria seria adicionar ao RECIPE uma fase de pré-avaliação da base de dados. A partir dessa análise, poderíamos gerar dinamicamente a gramática mais apropriada. Isso é interessante porque alguns métodos do scikit-learn se mostraram incapazes de lidar com algumas características das bases de dados. Entre elas, podemos citar: trabalhar com múltiplas classes, alguns parâmetros são limitados pelo número de atributos da base, e assim por diante. Se for possível avaliar a base de dados e depois disso criar a gramática é possível otimizar o uso dos recursos computacionais e a busca.

Além disso, implementar a fase de inicialização por meta-aprendizado ao RECIPE pode ser extremamente importante. Os resultados da última versão do AutoSKlearn

mostram que a inicialização por meta-aprendizado é potencialmente benéfica para a busca do melhor fluxo. É interessante verificar o quanto essa inicialização pode trazer benefícios para o RECIPE.

Por último, atualmente o RECIPE oferece suporte para a versão do Python 2.7. É interessante que ele também ofereça suporte para a nova versão do Python (3.6). Porém, esse processo de conversão não é simples.

Referências Bibliográficas

Asuncion, A. & Newman, D. (2007). UCI machine learning repository.

Back, T. (1996). *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press.

Barros, R. C.; Basgalupp, M. P.; de Carvalho, A. C. P. L. F. & Freitas, A. A. (2013). Automatic design of decision-tree algorithms with evolutionary algorithms. *Evolutionary Computation*, 21(4):659–684.

Bergstra, J.; Bardenet, R.; Bengio, Y. & Kégl, B. (2011). Algorithms for hyper-parameter optimization. Em *Proc. of the International Conference on Neural Information Processing Systems*, pp. 2546--2554.

Bergstra, J. & Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(1):281--305.

Bischl, B.; Casalicchio, G.; Feurer, M.; Hutter, F.; Lang, M.; Mantovani, R. G.; van Rijn, J. N. & Vanschoren, J. (2017). Openml benchmarking suites and the openml100. *arXiv preprint arXiv:1708.03731*.

Camargo, G. d. M. (2006). *Controle da pressão seletiva em algoritmo genético aplicado a otimização de demanda em infra-estrutura aeronáutica.* Tese de doutorado, Universidade de São Paulo.

De Castro, L. N. (2006). *Fundamentals of natural computing: basic concepts, algorithms, and applications*. CRC Press.

Deb, K.; Pratap, A.; Agarwal, S. & Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182--197.

Dietterich, T. G. et al. (2000). Ensemble methods in machine learning. *Multiple classifier systems*, 1857:1--15.

Dioşan, L.; Rogozan, A. & Pecuchet, J.-P. (2012). Improving classification performance of support vector machine by genetically optimising kernel shape and hyperparameters. *Applied Intelligence*, 36(2):280--294. ISSN 1573-7497.

Farrar, C. R. & Worden, K. (2012). *Structural health monitoring: a machine learning perspective*. John Wiley & Sons.

Fernandez, G. (2010). *Data mining using SAS applications*. CRC press.

Feurer, M.; Klein, A.; Eggensperger, K.; Springenberg, J.; Blum, M. & Hutter, F. (2015a). Efficient and robust automated machine learning. Em *Proc. of the International Conference on Neural Information Processing Systems*, pp. 2755--2763.

Feurer, M.; Springenberg, J. T. & Hutter, F. (2015b). Initializing bayesian hyperparameter optimization via meta-learning. Em *Proc. of the AAAI Conference on Artificial Intelligence*, pp. 1128--1135.

Freitas, A. A.; Vasieva, O. & de Magalhães, J. P. (2011). A data mining approach for classifying dna repair genes into ageing-related or non-ageing-related. *BMC Genomics*, 12(1).

Friedman, M. (1940). A comparison of alternative tests of significance for the problem of m rankings. *The Annals of Mathematical Statistics*, 11(1):86--92.

Fusi, N. & Elibol, H. M. (2017). Probabilistic matrix factorization for automated machine learning. *arXiv preprint*, arXiv:1705.05355v1.

Hall, M.; Frank, E.; Holmes, G.; Pfahringer, B.; Reutemann, P. & Witten, I. H. (2009). The WEKA data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11:10--18.

Ho, Y.-C. & Pepyne, D. L. (2002). Simple explanation of the no-free-lunch theorem and its implications. *Journal of optimization theory and applications*, 115(3):549--570.

Hutter, F.; Hoos, H. H. & Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. Em *Proceedings of the International Conference on Learning and Intelligent Optimization (LION)*, pp. 507--523. Springer-Verlag.

Khandani, A. E.; Kim, A. J. & Lo, A. W. (2010). Consumer credit-risk models via machine-learning algorithms. *Journal of Banking & Finance*, 34(11):2767--2787.

Kononenko, I. (2001). Machine learning for medical diagnosis: history, state of the art and perspective. *Artificial Intelligence in medicine*, 23(1):89--109.

Kotthoff, L.; Thornton, C.; Hoos, H. H.; Hutter, F. & Leyton-Brown, K. (2017). Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka. *Journal of Machine Learning Research*, 18(25):1--5.

Koza, J. R. (1994). Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4(2):87--112.

Křen, T.; Pilát, M. & Neruda, R. (2017). Automatic creation of machine learning workflows with strongly typed genetic programming. *International Journal on Artificial Intelligence Tools*, 26(05):1760020.

Li, L.; Jamieson, K.; DeSalvo, G.; Rostamizadeh, A. & Talwalkar, A. (2017). Hyperband: Bandit-based configuration evaluation for hyperparameter optimization. Em *Proceedings of the International Conference on Learning Representations (ICLR)*.

Luke, S. & Spector, L. (1998). A revised comparison of crossover and mutation in genetic programming. *Genetic Programming*, 98(208-213):55.

Mantovani, R. G.; Rossi, A. L. D.; Vanschoren, J.; Bischl, B. & de Carvalho, A. C. P. L. F. (2015). Effectiveness of random search in svm hyper-parameter tuning. Em *Proc. of the International Joint Conference on Neural Networks*, pp. 1--8.

McKay, R.; Hoai, N.; Whigham, P.; Shan, Y. & O'Neill, M. (2010). Grammar-based genetic programming: a survey. *Genetic Programming and Evolvable Machines*, 11(3):365--396.

Mendoza, H.; Klein, A.; Feurer, M.; Springenberg, J. & Hutter, F. (2016). Towards automatically-tuned neural networks. Em *Proc. of the ICML AutoML Workshop*.

Montana, D. J. (1995). Strongly typed genetic programming. *Evolutionary computation*, 3(2):199--230.

Olson, R.; Urbanowicz, R.; Andrews, P.; Lavender, N.; Kidd, L. & Moore, J. H. (2016a). Automating biomedical data science through tree-based pipeline optimization. Em *Proc. of the European Conference on the Applications of Evolutionary Computation*, pp. 123--137.

Olson, R. S.; Bartley, N.; Urbanowicz, R. J. & Moore, J. H. (2016b). Evaluation of a tree-based pipeline optimization tool for automating data science. Em *Proc. of the Genetic and Evolutionary Computation Conference*, pp. 485--492.

Olson, R. S.; Bartley, N.; Urbanowicz, R. J. & Moore, J. H. (2016c). Evaluation of a tree-based pipeline optimization tool for automating data science. Em *Proceedings of the Genetic and Evolutionary Computation Conference (2016)*, pp. 485--492. ACM.

Pang, B.; Lee, L. & Vaithyanathan, S. (2002). Thumbs up?: sentiment classification using machine learning techniques. Em *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*, pp. 79--86. Association for Computational Linguistics.

Pappa, G. L. & Freitas, A. A. (2009). *Automating the Design of Data Mining Algorithms: An Evolutionary Computation Approach*. Springer.

Pappa, G. L.; Ochoa, G.; Hyde, M. R.; Freitas, A. A.; Woodward, J. & Swan, J. (2014). Contrasting meta-learning and hyper-heuristic research: The role of evolutionary algorithms. *Genetic Programming and Evolvable Machines*, 15(1):3--35. ISSN 1389-2576.

Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; Vanderplas, J.; Passos, A.; Cournapeau, D.; Brucher, M.; Perrot, M. & Duchesnay, E. (2011). SciKit-Learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825--2830.

Piszcz, A. & Soule, T. (2006). A survey of mutation techniques in genetic programming. Em *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pp. 951--952. ACM.

Poli, R.; Langdon, W. B. & McPhee, N. F. (2008). *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd. ISBN 1409200736, 9781409200734.

Sá, A. G. C. & Pappa, G. L. (2013). Towards a method for automatically evolving Bayesian network classifiers. Em *Proc. of the Conference Companion on Genetic and Evolutionary Computation*, pp. 1505--1512.

Sá, A. G. C. & Pappa, G. L. (2014). A hyper-heuristic evolutionary algorithm for learning Bayesian network classifiers. Em *Proc. of Ibero-American Conference on Artificial Intelligence*, pp. 430--442.

Scott, E. O. & De Jong, K. A. (2016). Evaluation-time bias in quasi-generational and steady-state asynchronous evolutionary algorithms. Em *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pp. 845--852. ACM.

Smith-Miles, K. A. (2009). Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Computing Surveys (CSUR)*, 41(1):6.

Sohn, A.; Olson, R. S. & Moore, J. H. (2017). Toward the automated analysis of complex diseases in genome-wide association studies using genetic programming. Em *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pp. 489--496. ACM.

Song, Y.-Y. & Ying, L. (2015). Decision tree methods: applications for classification and prediction. *Shanghai archives of psychiatry*, 27(2):130.

Souto, M.; Costa, I.; Araujo, D.; Ludermir, T. & Schliep, A. (2008). Clustering cancer gene expression data: a comparative study. *BMC Bioinformatics*, 9(1):497.

Springenberg, J. T.; Klein, A.; S.Falkner & Hutter, F. (2016). Bayesian optimization with robust bayesian neural networks. Em *Proc. of the Conference on Neural Information Processing Systems*.

Stanley, K. O. & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99--127.

Thornton, C.; Hutter, F.; Hoos, H. H. & Leyton-Brown, K. (2013). Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. Em *Proc. of the International Conference on Knowledge Discovery and Data Mining*, pp. 847--855.

Vilalta, R. & Drissi, Y. (2002). A perspective view and survey of meta-learning. *Artificial Intelligence Review*, 18(2):77--95.

Wan, C.; Freitas, A. A. & De Magalhães, J. a. (2015). Predicting the pro-longevity or anti-longevity effect of model organism genes with new hierarchical feature selection methods. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 12(2):262--275.

Whigham, P. A.; Dick, G.; Maclaurin, J. & Owen, C. A. (2015). Examining the "best of both worlds" of grammatical evolution. Em *Proc. of the Conference on Genetic and Evolutionary Computation*, pp. 1111--1118.

Whigham, P. A. et al. (1995). Grammatically-based genetic programming. Em *Proceedings of the workshop on genetic programming: from theory to real-world applications*, volume 16, pp. 33--41.

Whitley, L. D. et al. (1989). The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. Em *ICGA*, volume 89, pp. 116--123. Fairfax, VA.

Wilcoxon, F.; Katti, S. & Wilcox, R. A. (1970). Critical values and probability levels for the Wilcoxon rank sum test and the wilcoxon signed rank test. *Selected tables in mathematical statistics*, 1:171--259.

Witten, I. H.; Frank, E. & Hall, M. A. (2011). *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., 3rd edição.

Yao, X. (1999). Evolving artificial neural networks. *Proc. of the IEEE*, 87(9):1423--1447.

Anexo A

Versão Completa da Gramática

```
<Start> ::= <preprocessing> '#' 'SA* ' <algorithm> | 'SA* ' <algorithm> |
      <preprocessing> '#' 'SA* ' <algorithm> '#' <algorithm> '#' 'Stacking' | 'SA* '
          <algorithm> '#' <algorithm> '#' 'Stacking' |
      <preprocessing> '#' 'SA* ' <algorithm> '#' <algorithm> '#' <algorithm> '#' '
          Stacking' | 'SA* ' <algorithm> '#' <algorithm> '#' <algorithm> '#' '
          Stacking' |
      <preprocessing> '#' 'SA* ' <algorithm> '#' <algorithm> '#' <algorithm> '#' <
          algorithm> '#' 'Stacking' | 'SA* ' <algorithm> '#' <algorithm> '#' <
          algorithm> '#' <algorithm> '#' 'Stacking' |
      <preprocessing> '#' 'SA* ' <algorithm> '#' <algorithm> '#' <algorithm> '#' <
          algorithm> '#' <algorithm> '#' 'Stacking' | 'SA* ' <algorithm> '#' <
          algorithm> '#' <algorithm> '#' <algorithm> '#' 'Stacking'
          |
      <preprocessing> '#' 'SA* ' <algorithm_e> '#' <algorithm_e> #' <algorithm_e>
          '#' 'Ensemble' | 'SA* ' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e>
          '#' 'Ensemble' |
      <preprocessing> '#' 'SA* ' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e>
          '#' <algorithm_e> '#' <algorithm_e> '#' 'Ensemble' | 'SA* ' <algorithm_e>
          '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e>
          '#' 'Ensemble' |
      <preprocessing> '#' 'SA* ' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e>
          '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e>
          '#' 'Ensemble' | 'SA* ' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e>
          '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e>
          '#' 'Ensemble' |
      <preprocessing> '#' 'SA* ' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e>
          '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e>
          '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e>
          '#' 'Ensemble' | 'SA* ' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e>
          '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e>
          '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e>
```

```
            '#' 'Ensemble' |
        <preprocessing> '#' 'SA* ' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e>
            '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e>
            '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e>
            '#' <algorithm_e> '#' <algorithm_e> '#' 'Ensemble' | 'SA* ' <algorithm_e>
            '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e>
            '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e>
            '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e>
            '#' 'Ensemble' |
        <preprocessing> '#' 'SA* ' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e>
            '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e>
            '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e>
            '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e>
            '#' 'Ensemble' | 'SA* ' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e>
            '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e>
            '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e>
            '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e> '#' <algorithm_e>
            '#' 'Ensemble'

##PREPROCESSING
<preprocessing> ::=  <imputation> | <bounding> | <dimensionality> | <binarizer> |
    <imputation> '#' <bounding> |
    <imputation> '#' <binarizer> |
    <imputation> '#' <dimensionality> |
    <bounding> '#' <dimensionality> |
    <imputation> '#' <bounding> '#' <dimensionality>

<imputation> ::= 'Imputer' <strategy_imp>

<binarizer> ::= 'Binarizer' <threshold_bin>

<bounding> ::= 'Normalizer' <norm> | 'MinMaxScaler' | 'MaxAbsScaler' |
    'RobustScaler' <boolean> <boolean> | 'StandardScaler' <boolean> <boolean>

<dimensionality> ::= <feature_selection> | <feature_construction> | <feature_selection
    > '#' <feature_construction>

<feature_selection> ::= 'VarianceThreshold' |
    'SelectKBest' <features> <score_function> |
    'SelectPercentile' <percentile> <score_function> |
    'SelectFpr' <value_rand_1> <score_function> |
    'SelectFwe' <value_rand_1> <score_function> |
    'SelectFdr' <value_rand_1> <score_function> |
    'RFE' <featuresToSelect> <value_rand_1> |
    'RecursiveFE_CV' <cv_rfe> <scoring_rfe> <value_rand_1> |
    'SelectFromModel' <threshold_sfm> <boolean> |
    'IncrementalPCA' <boolean> <featuresToSelect> |
    'TraditionalPCA' <boolean> <featuresToSelect> |
    'FastICA' <algorithm_fastica> <funct> <max_iter_fastica> <tol> <boolean> <
        featuresToSelect> |
    'GaussianRandomProjection' <value_rand_1> <featuresToSelect> |
    'SparseRandomProjection' <density> <boolean> <value_rand_1> <featuresToSelect> |
    'FeatureAgglomeration' <affinity> <boolean> <featuresToSelect> |
    'RBFSampler' <gamma_kernelApprox> <featuresToSelect> |
    'Nystroem' <kernel> <gamma_kernelApprox> <degree_1>  <coef0> <featuresToSelect>
```

```
<feature_construction> ::= 'PolynomialFeatures' <degree_1> <boolean> <boolean>

##ALGORITHM
<algorithm> ::= <strong> | <weak> | <tree_ensemble>
<algorithm_e> ::= <strong> | <weak>

<strong> ::= <Trees> | <NaiveBayes> | <SVM>
<weak> ::= <nearest> | <discriminant> | <logistic> | <passive> | <gradient_options> |
    <ridge>

<tree_ensemble> ::= 'AdaBoostClassifier' <algorithm_ada> <n_estimators> <
    learning_rate_ada> |
        'GradientBoostingClassifier' <loss_gradient> <tol> <value_rand_1> <presort>
            <n_estimators> <boolean> <max_features> <max_depth> <
            min_weight_fraction_leaf> <max_leaf_nodes> |
        'RandomForestClassifier' <criterion> <bootstrap_and_oob> <class_weight_Trees>
            <n_estimators> <boolean> <max_features> <max_depth> <
            min_weight_fraction_leaf> <max_leaf_nodes> |
        'ExtraTreesClassifier' <criterion> <bootstrap_and_oob> <class_weight_Trees> <
            n_estimators> <boolean> <max_features> <max_depth> <
            min_weight_fraction_leaf> <max_leaf_nodes>

<Trees> ::= 'ExtraTreeClassifier' <criterion> <splitter> <class_weight> <presort> <
    max_features> <max_depth> <min_weight_fraction_leaf> <max_leaf_nodes> |
    'DecisionTreeClassifier' <criterion> <splitter> <class_weight> <presort> <
        max_features> <max_depth> <min_weight_fraction_leaf> <max_leaf_nodes>

<NaiveBayes> ::= 'GaussianNB' |
    'BernoulliNB' <value_rand_1> <alpha> <boolean>|
    'MultinomialNB' <alpha> <boolean>

<SVM> ::= 'SVC' <kernel> <degree_1> <tol> <max_iter> <class_weight> |
    'NuSVC' <kernel> <degree_1> <tol> <max_iter> <class_weight>

<nearest> ::= 'KNeighborsClassifier' <k> <weights> <k_algorithm> <leaf_size> <p> <
    d_metric> |
    'RadiusNeighborsClassifier' <radius> <weights> <k_algorithm> <leaf_size> <p> <
        d_metric> |
    'CentroidClassifier' <shrinking_threshold> <d_metric>

<discriminant> ::= 'LinearDiscriminantAnalysis' <solver_discrim> <tol> <boolean> |
    'QuadraticDiscriminantAnalysis' <value_rand_1> <tol> <boolean>

<logistic> ::= 'LogisticRegression' <boolean> <max_iter_lr> <solver_lr_options> <
    class_weight> <boolean> <tol> |
    'LogisticCV' <cv> <boolean> <max_iter_lr> <solver_lr_options> <class_weight> <
        boolean> <tol>

<ridge> ::= 'RidgeClassifier' <max_iter> <boolean> <solver_ridge> <tol> <alpha> <
    class_weight> <boolean> <boolean> |
    'RidgeCCV' <cv> <alpha> <class_weight> <boolean> <boolean>

<passive> ::= 'PassiveAggressive ' <boolean> <n_iter> <boolean> <loss_sgd> <boolean> <
    class_weight>
```

```
<gradient_options> ::= 'Perceptron' <penalty> <alpha> <boolean> <n_iter> <boolean> <
    value_rand_1> <class_weight> <boolean>

##ATRIBUTES DEFINITION
<strategy_imp> ::= 'mean' | 'median' | 'most_frequent'
<criterion> ::= 'entropy' | 'gini'
<splitter> ::= 'best' | 'random'
<class_weight> ::= 'balanced' | 'None'
<presort> ::= 'True' | 'False' | 'auto'
<max_features> ::= 'RANDFLOAT(0.01,1.0)' | 'sqrt' | 'log2' | 'None'
##Depend on the number of features:
<max_depth> ::= 'RANDINT(1,100000)' | 'None'
<min_weight_fraction_leaf> ::= 'RANDFLOAT(0.0,0.5)'
##Depend on the number of features:
<max_leaf_nodes> ::= 'RANDINT(2,100000)' | 'None'
<threshold_bin> ::= 'RANDFLOAT(0.000001,100)'
<norm> ::= 'l1'| 'l2' | 'max'
<boolean> ::= 'True' | 'False'
<degree_1> ::= 'RANDINT(2,10)'
<score_function> ::= 'f_classif' | 'chi2'
<features> ::= 'RANDINT(2,3)'
<percentile> ::= 'RANDINT(5,95)'
<value_rand_1> ::= 'RANDFLOAT(0.0,1.0)'
<featuresToSelect> ::= 'RANDINT(1,3)'
<cv_rfe> ::= '2' | '3' | '5' | '10'
<scoring_rfe> ::= 'mean_absolute_error' | 'mean_squared_error' | '
    median_absolute_error' | 'r2' | 'None'
<threshold_sfm> ::= 'None' | 'median' | 'mean' | 'RANDFLOAT(0.0,1.0)'
<algorithm_fastica> ::= 'parallel' | 'deflation'
<funct> ::= 'logcosh' | 'exp' | 'cube'
<max_iter_fastica> ::= '10' | '100' | '250' | '500' | '750' | '1000'
<tol> ::= 'RANDFLOAT(0.0000000001,0.1)'
<density> ::= 'auto' | 'RANDFLOAT(0.00001,1.0)'
<affinity> ::= 'euclidean' <linkage_type0>| <affinity_options> <linkage_type1>
<affinity_options> ::= 'l1' | 'l2' | 'manhattan' | 'cosine'
<linkage_type0> ::= 'ward' | <linkage_type1>
<linkage_type1> ::= 'complete' | 'average'
<gamma_kernelApprox> ::= 'RANDFLOAT(0.000030518,8.0)'
<kernel> ::= 'linear' | 'poly' | 'rbf'| 'sigmoid'
<coef0> ::= 'RANDFLOAT(0.0,1000.0)'
<algorithm_ada> ::= 'SAMME.R' | 'SAMME'
<n_estimators> ::= '5' | '10' | '15' | '20' | '25' | '30' | '35' | '40' | '45' | '50'
<learning_rate_ada> ::= 'RANDFLOAT(0.01,2.0)'
<loss_gradient> ::= 'deviance' | 'exponential'
<bootstrap_and_oob> ::= 'True' <boolean> | 'False False'
<class_weight_Trees> ::= 'balanced' | 'balanced_subsample' | 'None'
<alpha> ::= 'RANDFLOAT(0.0,9.0)'
<max_iter> ::= '10' | '100' | '500' | '1000'
<k> ::= 'RANDINT(1,30)'
<weights> ::= 'uniform' | 'distance'
<k_algorithm> ::= 'auto' | 'brute' | 'kd_tree' | 'ball_tree'
<leaf_size> ::= 'RANDINT(5,100)'
<d_metric> ::= 'euclidean' |'manhattan' |'chebyshev' | 'minkowski'
<p> ::= 'RANDINT(1,15)'
```

```
<radius> ::= 'RANDFLOAT(1.0,30.0)'
<shrinking_threshold> ::= 'RANDFLOAT(0.0, 30.0)' | 'None'
<solver_discrim> ::= 'svd' | 'lsqr'
<C> ::= 'RANDFLOAT(0.03125,32768.0)'
<intercept_scaling> ::= 'RANDFLOAT(0.0,100.0)'
<max_iter_lr> ::= '10' | '100' | '150' | '300' | '350' | '400' | '450' |'500'
<solver_lr_options> ::= 'liblinear' | 'sag' | 'newton-cg' | 'lbfgs'
<cv> ::= 'RANDINT(2,10)' | 'None'
<solver_ridge> ::= 'auto' | 'svd' | 'cholesky' | 'lsqr' | 'sparse_cg' | 'sag'
<loss_sgd> ::= 'hinge'| 'log'| 'modified_huber'| 'squared_hinge'| 'perceptron'| '
    squared_loss'| 'huber'| 'epsilon_insensitive'| 'squared_epsilon_insensitive'
<n_iter> ::= '5' | '10' | '25' | '50' | '100' | '250' | '500' | '750'
<penalty> ::= 'l1' | 'l2'
<power_t> ::= 'RANDFLOAT(0.1, 5.0)'
<learning_rate_sgd> ::= 'constant'| 'invscaling' | 'optimal'
<average> ::= 'True' | 'False' | 'RANDINT(1,100)'
```