UNIVERSIDADE DE LISBOA

FACULDADE DE CIÊNCIAS

DEPARTAMENTO DE INFORMÁTICA

Optimization of Feature Learning through Grammar-Guided Genetic Programming

Leon Kornelis Ingelse

Mestrado em Ciência de Dados

Dissertação orientada por:

Professor Doutor Alcides Aguiar Fonseca

2022

“Que seja infinito enquanto dure”

— Vinicius de Moraes, *Soneto de Fidelidade*

Acknowledgements

As this is the only place where I can be poetic and dramatic, a dramatic poet I'll be.

This thesis was a journey with many hardships and pleasures, but mostly much fun and interesting discourses. The journey was smoothened, and made profoundly durable, by many, but, first and foremost, by the seemingly effortless availability of my supervisor, Alcides. His deep source of academic knowledge and relentless drive to push me into interesting waters has made it a challenging, but adventurous voyage. We share the excitement of Genetic Engine and believe in its potential. He helped me through tough obstacles and invited me to work on new methods and meet new, interesting people, some of which I was allowed to meet at EuroGP and GECCO. I'm grateful for his energy, availability and contagious excitement, but mostly for the responsibility he gave me by handing me all he could offer, without expecting anything in return.

The second person I want to thank is Guilherme Espada, for being a reliable source of advice, good advice, never unknowing. For his discussions on academic work, but also on politics, a well of information of the world's order, inequality and wrong power dynamics. And for going to EuroGP with me, and having an unforgettable time, interesting academically, open freedom socially.

Thanks to the others of the team, in alphabetic order. To Afonso, for being happy and unconforming, and being open to connect. To Catarina, for being a reliable strength, an inspiration of perseverance and believe in oneself. To Paulo, for opening up and showing interest, and his admirable conquest for freedom. To Pedro, for accepting my stress and his eagerness to learn and progress, openness to critically evaluate himself and surroundings and to not be afraid to criticize me. Together with others, like Miguel, Tiago Guerreiro, Rita, Pedro, Ziggy, João, Filipa, Nuno, Diogo, they made me feel welcome and at home at FCUL, and I will always remember the lunches, lanches, and procrastination sessions.

Thanks to Ruben, for being the cornerstone of FCUL society, even though he might not want to be that. For the night walks during the pandemic and the long philosophical talks on freedom and happiness. Even though our dreams do not align, we connect on dreaming itself, and that is beautiful.

Thanks to Castanheira for asking me to call during the pandemic, for being patient with me, and for trying to enjoy life. For sharing his house, friends, hardships and happiness with me.

Thanks to my parents and brother for never ceasing to support me, and for relentlessly trusting in me. For always being proud of me even if I do not want it. For always trying to understand what I am doing, and patiently listening to my endless mulling and doubting. For being there, whenever I need them, without asking anything in return.

Thanks to Mau, for supporting me, calling me, assuring me. For visiting me, and loving me, exploring Portugal with me, and finally allowing me to choose my own path, unconditionally.

Thanks to all the others in Lisbon that made these two years an amazing experience. Ole, for dragging me along on ridiculous journeys. Dharma, for talking and sharing. Chiara, for searching together. Fran, for being there, always. Cesar, for exploring. Fotis, for being real. Andrea, for the advice. Nina, for the talks. Lukas, for the energy. Clem, for the music. Bruno, for the drive of life. Akshay, for the calmness and perseverance. Flo, for dancing, still fresh. Nadia, for the art. Suzy, for the freedom. Giannis, for being the calm, wise rock. Eleanora, for bombing with excitement. Jessica, for the spirit. And all the others, Stefan, Hatty, Alice, Martina, Marc, Gonçalo, Robert, Mariana, Cleide, Rômelo, Maria, Eleanora, Giacomo, Mario, Mario, Alicia, Lourenzo, Suzano, Pedro, and in the Netherlands, Lodie, Thijs, Pelle, Selma, Aaron, Nizi, Jerry, Nawo, Bram, Bram, Swift.

And finally, thanks to magical Lisbon, its secrets, its alleys, its temper, its accent, its roughness, its timelessness, its paradoxes, the beaches, the nature, the miradouros, the Tejo, the music, the culture, the bitoques, the dancing, the dancing, the dancing.

And of course the money. This work was supported by *Fundação para a Ciência e Tecnologia (FCT)* in the LASIGE Research Unit under the ref. UIDB/00408/2020 and UIDP/00408/2020, the CMU–Portugal project CAMELOT (POCI-01-0247-FEDER-045915), and the RAP project under the reference (EXPL/CCI-COM/1306/2021).

Abstract

Machine Learning (ML) is becoming more prominent in daily life. A key aspect in ML is Feature Engineering (FE), which can entail a long and tedious process. Therefore, the automation of FE, known as Feature Learning (FL), can be highly rewarding. FL methods need not only have high prediction performance, but should also produce interpretable methods. Many current high-performance ML methods that can be considered FL methods, such as Neural Networks and PCA, lack interpretability.

A popular ML used for FL that produces interpretable models is Genetic Programming (GP), with multiple successful applications and methods like M3GP. In this thesis, I present two new GP-based FL methods, namely M3GP with Domain Knowledge (DK-M3GP) and DK-M3GP with feature Aggregation (DKA-M3GP). Both use grammars to enhance the search process of GP, in a method called Grammar-Guided GP (GGGP). DK-M3GP uses grammars to incorporate domain knowledge in the search process. In particular, I use DK-M3GP to define what solutions are humanly valid, in this case by disallowing operating arithmetically on categorical features. For example, the multiplication of the postal code of an individual with their wage is not deemed sensible and thus disallowed.

In DKA-M3GP, I use grammars to include a feature aggregation method in the search space. This method can be used for time series and panel datasets, to aggregate the target value of historic data based on a known feature value of a new data point. For example, if I want to predict the number of bikes seen daily in a city, it is interesting to know how many were seen on average in the last week. Furthermore, DKA-M3GP allows for filtering the aggregation based on some other feature value. For example, we can include the average number of bikes seen on past Sundays.

I evaluated my FL methods for two ML problems in two environments. First, I evaluate the independent FL process, and, after that, I evaluate the FL steps within four ML pipelines. Independently, DK-M3GP shows a two-fold advantage over normal M3GP; better interpretability in general, and higher prediction performance for one problem. DKA-M3GP has a much better prediction performance than M3GP for one problem, and a slightly better one for the other. Furthermore, within the ML pipelines it performed well in one of two problems. Overall, my methods show potential for FL.

Both methods are implemented in Genetic Engine an individual-representation-independent GGGP framework, created as part of this thesis. Genetic Engine is completely implemented in Python and shows competing performance with the mature GGGP framework PonyGE2.

Keywords: Feature Learning, Grammar-Guided Genetic Programming, Domain knowledge, Aggregation, Interpretability

Resumo alargado

A Inteligência Artificial (IA) e o seu subconjunto de Aprendizagem Automática (AA) estão a tornar-se mais importantes para nossas vidas a cada dia que passa. Ambas as áreas estão presentes no nosso dia a dia em diversas aplicações como o reconhecimento automático de voz, os carros autónomos, ou o reconhecimento de imagens e deteção de objetos. A AA foi aplicada com sucesso em muitas áreas, como saúde, finanças e marketing.

Num contexto supervisionado, os modelos de AA são treinados com dados e, posteriormente, são usados para prever o comportamento de dados futuros. A combinação de etapas realizadas para construir um modelo de AA, totalmente treinado e avaliado, é chamada um AA *pipeline*, ou simplesmente *pipeline*. Todos os pipelines seguem etapas obrigatórias, nomeadamente a recuperação, limpeza e manipulação dos dados, a seleção e construção de *features*, a seleção do modelo e a otimização dos seus parâmetros, finalmente, a avaliação do modelo. A construção de AA pipelines é uma tarefa desafiante, com especificidades que dependem do domínio do problema. Existem desafios do lado do design, otimização de hiperparâmetros, assim como no lado da implementação.

No desenho de pipelines, as escolhas devem ser feitas em relação aos componentes a utilizar e à sua ordem. Mesmo para especialistas em AA, desenhar pipelines é uma tarefa entediante . As escolhas de design exigem experiência em AA e um conhecimento do domínio do problema, o que torna a construção do *pipeline* num processo intensivo de recursos.

Após o desenho do *pipeline*, os parâmetros do mesmo devem ser otimizados para melhorar o seu desempenho. A otimização de parâmetros, geralmente, requer a execução e avaliação sequencial do *pipeline*, envolvendo altos custos. No lado da implementação, os programadores podem introduzir bugs durante o processo de desenvolvimento. Esses bugs podem levar à perda de tempo e dinheiro para serem corrigidos, e, se não forem detectados, podem comprometer a robustez e correção do modelo ou introduzir problemas de desempenho. Para contornar esses problemas de design e implementação, surgiu uma nova linha de investigação designada por AutoML (*Automated Machine Learning*). AutoML visa automatizar o desenho de AA pipelines, a otimização de parâmetros, e a sua implementação. Uma parte importante dos pipelines de AA é a maneira como os *features* dos dados são manipulados. A manipulação de dados tem muitos aspetos, reunidos sob o termo genérico Feature Engineering (FE). Em suma, FE visa melhorar a qualidade do espaço de solução selecionando as *features* mais importantes e construindo novas *features* relevantes. Contudo, este é um processo que consome muitos recursos, pelo que a sua automação é uma sub-área altamente recompensadora de AutoML. Nesta tese, defino Feature Learning (FL) como a área de FE automatizado.

Uma métrica importante de FE e, portanto, de FL, é a interpretabilidade das *features* aprendidas. Interpretabilidade, que se enquadra na área de Explainable IA (XIA), refere-se à facilidade de entender o significado de uma *feature*. A ocorrência de diversos escândalos em IA, como modelos racistas e sexistas, levaram a União Europeia a propor legislação sobre modelos sem interpretabilidade. Muitos métodos clássicos, e portanto amplamente usados, carecem de interpretabilidade, dando origem ao interesse recém-descoberto em XIA. A atual investigação em FL trata os valores de *features* existentes sem os relacionar com o seu significado semântico. Por exemplo, engenharia de uma *feature* que representa a multiplicação do código postal com a idade de uma pessoa não é um uso lógico do código postal. Embora os códigos postais possam ser representados como números inteiros, eles devem ser tratados como valores categóricos. A prevenção deste tipo de interações entre *features*, melhora o desempenho do *pipeline*, uma vez que reduz o espaço de procura de possíveis *features* ficando apenas com as que fazem semanticamente sentido. Além disso, este processo resulta em *features* que são intrinsecamente interpretáveis. Deste modo, o conhecimento sobre o domínio do problema, impede a engenharia de *features* sem significado durante o processo de FE..

Outro aspecto de FL normalmente não considerado nos métodos existentes, é a agregação de valores de uma única *feature* por várias entidades de dados. Por exemplo, vamos considerar um conjunto de dados sobre fraude de cartão de crédito. A quantidade média de transações anteriores de um cartão é potencialmente uma *feature* interessante para incluir, pois transmite o significado de uma transação ’normal’. No entanto, isso geralmente não é diretamente inferível nos métodos de FL existentes. Refiro-me a este método de FL como agregação de entidades, ou simplesmente agregação.

Por fim, apesar da natureza imprevisível dos conjuntos de dados da vida real, os métodos existentes exigem principalmente *features* que tenham dados homogêneos. Isso exige que os cientistas de dados realizem um pré-processamento do conjunto de dados. Muitas vezes, isso requer transformar categorias em números inteiros ou algum tipo de codificação, como por exemplo *one-hot encoding*. Contudo, conforme discutido acima, isso pode reduzir a interpretabilidade e o desempenho do *pipeline*.

A Programação Genética (GP), um método de ML, é também usado para FL e permite a criação de modelos mais interpretáveis que a maioria dos métodos tradicionais. GP é um método baseado em procura que evolui programas ou, no caso de FL, mapeamentos entre apresentas de espaços. Os métodos de FL baseados em GP existentes não incorporam os três aspectos acima mencionados: o conhecimento do domínio, a agregação e a conformidade com tipos de dados heterogêneos. Algumas abordagens incorporam algumas partes desses aspetos, principalmente usando gramáticas para orientar o processo de procura. O objetivo deste trabalho é explorar se a GP consegue usar gramáticas para melhorar a qualidade da FL, quer em termos de desempenho preditivo ou de interpretabilidade. Primeiro, construímos o Genetic Engine, uma *framework* de GP guiada por gramática (Grammar-Guided GP (GGGP)). O Genetic Engine é uma *framework* de GGGP fácil de usar que permite expressar gramáticas complexas. Mostramos que o Genetic Engine tem um bom desempenho quando comparado com a *framework* de Python do estado da arte, PonyGE2.

Em segundo lugar, proponho dois novos métodos de FL baseados em GGGP implementados no Genetic Engine. Ambos os métodos estendem o M3GP, o método FL do estado da arte baseado em GP. A primeira incorpora o conhecimento do domínio, denominado M3GP com conhecimento do domínio

(*M3GP with Domain Knowledge* (DK-M3GP)). O primeiro método restringe o comportamento das *features* permitindo apenas interações sensatas, por meio de condições e declarações. O segundo método estende X DK-M3GP, introduzindo agregação no espaço de procura, e é denominado DK-M3GP com Agregação (*DK-M3GP with Aggregation* (DKA-M3GP)). O DKA-M3GP usa totalmente a facilidade de implementação do Genetic Engine, pois requer a implementação de uma gramática complexa.

Neste trabalho, o DK-M3GP e DKA-M3GP foram avaliados em comparação com o GP Tradicional, M3GP e numerosos métodos clássicos de FL em dois problemas de ML. As novas abordagens foram avaliadas assumindo que são métodos autônomos de FL e fazendo parte de uma *pipeline* maior. Como métodos FL independentes, ambos os métodos demonstram boa previsão de desempenho em pelo menos um dos dois problemas. Como parte da *pipeline*, os métodos apresentam pouca vantagem em relação aos métodos clássicos no seu desempenho de previsão. Após a análise dos resultados, uma possível explicação encontra-se no *overfitting* dos métodos FL para a função de fitness e no conjunto de dados de treino. O

Neste trabalho, discuto também a melhoria na interpretabilidade após incorporar conhecimento do domínio no processo de procura. Uma avaliação preliminar do DK-M3GP indica que, utilizando a medida de complexidade *Expression Size* (ES), é possível obter uma melhoria na interpretabilidade. Todavia, verifiquei também que a medida de complexidade utilizada pode não ser a mais adequada devido a estrutura de características em forma de árvore das características construídas por DK-M3GP que potencia um ES. Considero que um método de avaliação de interpretabilidade mais complexo deve apontar isso.

Keywords: Feature Learning, Grammar-Guided Genetic Programming, Conhecimento do Domínio, Agregação, Interpretabilidade

Contents

List of Figures

List of Tables

Acronyms

AI	Artificial Intelligence
AutoML	Automated Machine Learning
BNF	Backus-Normal-Form
CFG	Context-Free Grammars
CFG-GP	Context-Free Grammars Genetic Programming
DK-M3GP	M3GP with Domain Knowledge
DKA-M3GP	M3GP with Domain Knowledge and Aggregation
DL	Deep Learning
DSGE	Dynamic Structured Grammatical Evolution
DT	Decision Tree
ES	Expression Size
FE	Feature Engineering
FEAT	Feature Engineering Automation Tool
FG	Feature Generation
FL	Feature Learning
FS	Feature Selection
FT	Feature Transformation
HC	Hill Climbing
GE	Grammatical Evolution
GGGP	Grammar-Guided Genetic Programming
GP	Genetic Programming
MFI	Multiple-Features-per-Individual
ML	Machine Learning
MLP	Multilayer Perceptron
MSE	Mean Squared Error
PCA	Principal Component Analysis
PI	Position-Independent
PGE	Probabilistic Grammatical Evolution
RF	Random Forest

RS	Random Search
SFI	Single-Feature-per-Individual
SGE	Structured Grammatical Evolution
STGP	Strongly Typed Genetic Programming
SVM	Support Vector Machine
VE-GP	Vectorial Genetic Programming
XAI	Explainable Artificial Intelligence

Chapter 1

Introduction

1.1 Motivation

Artificial Intelligence (AI), and its subset Machine Learning (ML), are becoming more important for our lives every day. They are present in our daily life, in various applications, such as automatic speech recognition, self-driving cars, image-recognition, and object-detection. ML has been successfully applied in areas such as health care [46], Natural Language Processing [12], finance [24], and marketing [13].

In a supervised context, ML models are trained with data, and thereafter used to predict behaviour of future data. The combination of steps undertaken to build a fully trained and evaluated ML model is called a *machine learning pipeline*, or simply *pipeline*. Necessary steps of these pipelines include retrieving data, cleaning and manipulating data, feature selection and feature construction, model selection and model-parameter optimization, and model evaluation. The construction of ML pipelines is a challenging task, with domain-dependent specificities. There are design-side challenges, hyperparameter optimization, and implementation-side challenges.

When designing pipelines, choices must be made regarding the components to use and their order. Even for ML experts, designing pipelines is a tedious matter [39]. Design choices require both ML expertise and domain knowledge, making the construction a resource-intensive process. Moreover, human choices introduce bias, which could deteriorate performance and reduce fairness.

After design, pipeline parameters should be optimized to enhance performance. Parameter optimization generally requires the sequential running and evaluation of the pipeline, incurring high computational costs.

On the implementation side, developers may introduce bugs during the construction process. These bugs can cost time and money to be fixed, but if undetected, they can compromise the robustness and the correctness of the model, or introduce performance issues. Some examples of these bugs occur when training and testing data are mixed [113], algorithms are wrongly implemented [99], local and ML library environments do not align [111, 48], ML library's frameworks are misused [23, 111, 48], software engineering standards are neglected [88], etc.

To bypass these design and implementation issues, a new field of research has appeared: Automated Machine Learning (AutoML). AutoML aims to automate ML pipeline design, parameter optimization, and implementation. Hutter et al. [43] define AutoML as the area that studies the automatization of the usage of ML technologies, and is, as such, "democratizing" ML.

An important part of ML pipelines is the manner in which the features of the data are manipulated. In this work, we consider *Feature Selection*, *Feature Transformation*, and *Feature Generation* as defined by Dong and Liu [26]. Dong and Liu [26] gather the used feature-manipulation techniques (and more) under the umbrella term *Feature Engineering (FE)*. Summarizing, FE aims to improve the quality of the solution space [97] by selecting important features and constructing new, relevant features.

1.2 Problem Statement

There is no standard procedure to perform FE, as it requires "domain knowledge, intuition, and [...] a lengthy process of trial and error" [26]. Altogether, it is a resource-intensive process, making automation of FE a highly rewarding subarea of AutoML. Moreover, "automation of the [FE] process" [28] is claimed as the reason for Deep Learning's (DL) success. Here, the trained weights in DL can be considered engineered features, and each network layer as a set of engineered features. In this sense, training the weights is an automated feature engineering process, where each layer is performing FE on the previous Layer, creating an immensely complex FE system. In this thesis, I define *Feature Learning (FL)* as the area of automated FE.

An important metric of FE, and thus FL, is the interpretability of the learned features [26]. Interpretability, which falls under Explainable AI (XAI), refers to how easy it is to understand what a feature means. Scandals in AI, such as racist and sexist models [114], parallel to more philosophical considerations [60], made the European Union move to propose legislation concerning models lacking explainability [94]. Many widely used, classical methods lack interpretability and explainability, giving rise to the new-found interest in XAI. For example, a drawback of using the layers of DL models is that they are not interpretable [19]. Other popular methods, like Support Vector Machines (SVM) and the feature-dimensionality reduction algorithm Principal Component Analysis (PCA) [110] also automatically engineer features that are not interpretable [19].

Current FL research treats existing feature values regardless of their semantic meaning. For example, engineering a feature that is the multiplication of the postal code with the age of a person is not a logical usage of the postal code. Even though postal codes can be represented as integers, they should be treated as categorical values. The prevention of senseless interactions between features enhances performance by reducing the search space of possible features to those that make sense, and results to features that are intrinsically interpretable. Encoding Domain knowledge in the FE process could prevent the engineering of senseless features.

Another aspect of FL normally not considered in existing methods is the aggregation of values of a single feature over multiple data entities. For instance, let us consider a dataset on credit card fraud. The average amount of previous transactions from a card is potentially an interesting feature to include, as it conveys a 'normal' transaction. Nevertheless, this is generally not directly inferable in existing FL approaches. I refer to this FE method as *entity aggregation*, or simply *aggregation*.

Lastly, despite the unpredictable nature of real-life datasets, existing methods mostly require input features to have homogeneous data types. This requires data scientists to perform preprocessing to the dataset. Often times, this requires transforming categories to integers, or some sort of one-hot encoding. As discussed above, this might reduce both interpretability and performance.

One ML method, Genetic Programming (GP), can create interpretable models [7], and can also be used for FL. GP is a search-based method that evolves programmes, or, in the case of FL, mappings between features spaces. GP has been applied to FL multiple times successfully [9, 4, 5, 19, 38, 102]. Existing GP-based FL approaches are mostly designed for a limited set of data types, typically scalars or vectors [6]. On the one hand, aggregation of the value of a single feature over multiple data entities is generally not possible for scalar representations, but would be beneficial for certain datasets, e.g., time-series and panels. On the other hand, inter-scalar functions are generally not available in vector representations. Overall, grammars have been used to incorporate domain knowledge into the search process, but traditional GP requires input data to be homogeneous, and aggregating over historic data has not been thoroughly studied.

1.3 Objective

The goal of this work is to explore if Genetic Programming can use grammars to improve the quality of FL, either in terms of predictive performance or in interpretability. We propose to design a customizable process in which practitioners can encode domain knowledge in the program, making the FL process problem-dependent. In particular, we intend to improve the predictive performance and the interpretability of feature types through the incorporation of domain knowledge. Furthermore, we introduce a grammar with the ability to aggregate over historic data.

There are three prospected advantages of incorporating domain knowledge to the search process: Firstly, feature interactions that do not make sense will be excluded from the search process, which I expect to improve prediction performance. As an example, categorical features will not be multiplied by continuous variables. Secondly, as learned features only conform to domain-specific rules, the learned features are anticipated to be more easily interpreted. Finally, our approach is time-series aware, and can perform aggregations over multiple data entities, assuming higher predictive performance due to higher expressive power. This extra expressive power does not come at the cost of interpretability, as aggregations are easily interpreted by domain experts.

In this work, I introduce two approaches; the first only incorporates domain knowledge; and the second incorporates both domain knowledge and the ability to perform aggregations. Since I based my approaches on M3GP [73], I name these methods M3GP with Domain Knowledge (DK-M3GP) and DK-M3GP with Aggregation (DKA-M3GP), respectively.

In this work, I also introduce Genetic Engine [29], a GP framework with a Strongly Typed GP (STGP) [70] frontend and a Grammar-Guided GP (GGGP) [106] backend. Genetic Engine is used to build a GP-based FL model that incorporates domain knowledge into the search space, allows for aggregation, and works for any data type. Genetic Engine is also a contribution in light of the need of GP to be "integrated into standard software development infrastructure" [91]. I introduce Genetic Engine more elaborately in chapter 5. In the context of FL, we implement both DK-M3GP and DKA-M3GP in Genetic Engine. As such, we use Genetic Engine to experiment with (1) implementing domain-specific knowledge to enhance performance and feature interpretability, (2) aggregation over multiple entities to improve performance and feature interpretability, and (3) combining different data types to allow heterogeneous features.

In summary, this work has the following contributions:

1. Genetic Engine, a GP framework aimed at allowing a larger audience the benefits of GP and its various adversaries.
2. A comparison between M3GP implemented in Genetic Engine and M3GP implemented by [9]. By comparing Genetic Engine with a respectable implementation, I justify using Genetic Engine as an acceptable tool for GGGP.
3. The application of Genetic Engine to the problem of FL on two heterogeneous, complex datasets.
4. The implementation of DK-M3GP and DKA-M3GP in Genetic Engine. These methods incorporate domain-specific knowledge, entity aggregation and heterogeneous data types. Given the domain-specific knowledge, this method is generalizable for other datasets.
5. Researching the impact of domain-knowledge incorporation, entity aggregation, and heterogeneous data types to GP-based FL.

1.4 Outline

The rest of this document is structured as follows. In chapter 2, I introduce the building blocks used within ML pipelines. Furthermore, I elaborate on the basic concepts of GP. In chapter 3, I discuss existing research on the area of FL, and, in particular, the area of GP-based FL, as well as different search-space-restricting techniques within GP. In chapter 4 I introduce a regression problem, through the BoomBikes dataset [10], and a classification problem, through the credit-g dataset [27]. Genetic Engine is introduced elaborately in chapter 5. Chapter 6 introduces Traditional GP, M3GP, and my proposed approaches DK-M3GP and DKA-M3GP. In chapter 7 I present the results of my thesis.

Chapter 2

Background

In this chapter, I introduce topics crucial to this thesis. Firstly, I introduce basic concepts of Machine Learning (ML) in section 2.1. In section 2.1.1, I introduce a subset of ML: search-based methods, with a focus on GP. Finally, I introduce formal grammars in section 2.2 which I will later use in GGGP.

2.1 Machine Learning

There are two main areas in ML, supervised and unsupervised. Unsupervised ML aims to extract unknown relations between individuals from a dataset. Supervised ML concerns learning models to predict unknown features from known features. These unknown features are called the *target* features. Models are learned through training on a subset for which the target features are known.

During training, the target features *supervise* the model on what predictions are correct and incorrect. Once trained, the model is tested on a test set, of which the target features are not given. A model that performs well on a training dataset does not necessarily perform well on test data. When the model is optimized for the training dataset, but does not generalize to the test data, the model is said to be *overfitted*.

FL is a key part of ML, which aims to create new features by combining existing features and operators. FL is the automation of FE, which is an umbrella name for various methods. Methods are *Feature Selection (FS)*, *Feature Transformation (FT)*, and *Feature Generation (FG)*, as defined by Dong and Liu [26]. FS addresses the reduction of the number of features in the data, by only selecting the most relevant ones. FT, also known as Feature Construction, combines different features through arithmetic operations to find new relevant features. An example of FT is the Body Mass Index (BMI), which combines sex, weight and height to create a new feature. The BMI is more compact than the separate features, and contains the necessary health information, allowing for quick consultation. FG extracts features from the whole data set to be assembled in new relevant features. An example of FG is the aggregation method discuss in chapter 1. There are other methods, but for this thesis, only FS, FT and FG are relevant.

The usefulness of a feature is measured by the improvement at solving the problem at hand, but its interpretability is also important [26]. Interpretability falls under Explainable AI (XAI), discussed in section 2.1.3.

2.1.1 Search-based methods

In comparison to other optimization methods, a benefit of search-based methods is that the shape of the solution must not necessarily be given beforehand. Only the solution space must be defined. The solution space consists of two disjoint sets, the *terminal* set and the *function* set. Functions operate on terminals and function results, whereas terminals do not operate on anything.

Let's look at the example of symbolic regression. Given a dataset D with features $x_0, \dots, x_n$ and target values Y, a symbolic regression solution is a combination of the features $x_0, \dots, x_n$, using a predefined set of operators and possibly scalar values. For this example, let the function set be a set of operators, say $\{+, *\}$, and let the terminal set be the set of features and scalar values, i.e., $\{x_0, \dots, x_n\} \cup \mathbb{R}$. Given an error function, the aim of symbolic regression is to find a solution S such that the error between $S(d)$ and $y_d \in Y$ is minimized, for all $d \in D$. Possible solutions are

$$\begin{aligned}
&S_1 : y = 2x_0x_1x_3 + x_4 - 3x_1x_5, \\
&S_2 : y = 11x_1^3 + 5.2x_2^6, \text{ and} \\
&S_3 : y = x_0^2 - x_0x_1 + x_1^2.
\end{aligned}$$

If there are finite possible solutions, one can simply generate all and see which one is best. However, this can be extremely expensive, especially for complex solution spaces. Moreover, the number of possible solutions are not necessarily finite.

Search-based methods aim to efficiently find the best possible solution. Checking all solutions is generally not feasible, so, in practise, finding a well-performing solution is good enough. The most basic search-based method is Random Search (RS). RS randomly selects solutions from the solution space until some criteria are met.

2.1.2 Genetic Programming

Genetic Programming (GP) [56] is a particular search-based method, inspired by the concept of natural selection. In the following, naming and explanation ideas are taken from *A Field Guide to Genetic Programming* [83].

Given a problem and a solution space S, a GP algorithm needs three components. Firstly, it needs a way of randomly generating *individuals* from S. Generally, individuals are represented as trees or as strings, representing a possible solution. Secondly, it must be provided with an evaluation method for individuals. This method is called the *fitness function*, and measures an individual's capability of solving the problem at hand. Thirdly, we require a method for randomly changing individuals within S.

There are two generally used operations for this, *mutation* of single individuals, and *crossovers* between individuals, i.e., combining two individuals to create a new one. The above operators are called *evolutionary operators*. The individuals that are used to construct new individuals are called the *parents*, and the obtained individuals are called the *children*. These operators can be used separately, or they can be combined.

A GP algorithm starts by generating a *population* of individuals. It then iterates over *generations*, where in each generation a new population is produced. The new individuals of the new population are created through three different methods. Firstly, some individuals are directly moved to the next generation. This is called *elitism*, and the number of individuals directly moved is called the *elitism size*. Secondly, some individuals are created through the evolutionary operators. Two probability parameters are used to determine whether two parents undergo crossover and/or mutation. These parameters are called *crossover probability* and *mutation probability*. Finally, the population is completed by adding new, randomly generated individuals. This is called *novelty*, and the number of newly created individuals is called the *novelty size*. After each generation, the goal is to achieve an improvement of the population's fitness.

Being a search-based method, GP can benefit from restricting the search space. There are plenty of methods to do so (more on this in section 3.1). In this thesis I use grammars to restrict the search space in a method called Grammar-Guided GP (GGGP). In section 2.2 I introduce grammars.

2.1.3 Metrics

2.1.3.1 Performance

In ML, prediction metrics (e.g., accuracy, mean-squared error (MSE), F_1 score, area under the curve) are used to evaluate the performance of solutions. More specifically, the prediction performance is evaluated against time, the number of evaluations, or the number of generations. In this thesis I use the MSE (definition 1) to evaluate regression solutions, and the F_1 score (definition 2) to evaluate classification solutions.

Definition 1. *Given a dataset with n data points, with ground truth y_{gt}. The MSE of solution S with predictions y_{pred} is then*

$$MSE_M = \frac{1}{n}\sum_{i=0}^{n-1}(y_{gt}[i] - y_{pred}[i])^2.$$

The F_1 score combines *precision* and *recall*. For a class C and a solution S, let the *true positives* (tp_C) be the set of data points that were correctly classified by S as being of C, and let the *false positives* (fp_C) be the set of data points that were wrongly classified by S as being of C. The precision of S for class C is then $p_{S,C} = \frac{tp_C}{tp_C+fp_C}$. Now, let the *false negatives* (fn_C) be the set of data points that were wrongly classified by S as not being of C. The recall of S for class C is then $r_{S,C} = \frac{tp_C}{tp_C+fn_C}$. A disadvantage of both precision and recall is that they focus on part of the predicted values, making it possible to have overall badly performing models with good prediction and recall scores. To remediate that, the F_1 score combines the two scores.

Definition 2. *Given a dataset with class C, and a classification solution S, with precision p_C and recall r_C, the F_1 score of S for class C is*

$$F_{1,S,C} = 2 * \frac{p_C * r_C}{p_C + r_C}.$$

The overall F_1 score of S can be defined in multiple manners. The *binary* method is only applicable to binary classification, in which one of the classes is chosen to be the positive class, and the overall F_1 score equals the F_1 score of that class. In the *weighted* method, the overall F_1 score is the weighted mean of the F_1 scores of all the classes. It is weighted on class size.

As our dataset has a binary class, we can use both methods. Both methods have advantages and disadvantages. The binary method does not include *true negatives* (the set of data points that were correctly classified as being from the negative class). The weighted mean might push for the F_1 score of the dominant class to be given too much importance. In line with the implementation of Batista and Silva [8], I used the weighted average. Oftentimes, I append results using the binary method to support the narrative.

To conclude, a lower MSE is better, as the error is smaller. The minimum and best MSE is 0. A higher F_1 score is better, with a perfect score of 1. The lowest F_1 score is 0.

2.1.3.2 Interpretability

In many domains where accountability is required, such as policy, medicine, defence and critical infrastructures, it is necessary to explain how decisions are made [60]. As such, solutions are required to be *interpretable*. Due to recent scandals in AI, such as racist and sexist solutions [114], there is a push for legislation concerning the use of non-interpretable solutions [94]. Furthermore, well-predicting solutions have learned something about the data, and thus the domain. The understanding of the decisions of the solution can thus contribute to domain knowledge.

Recently, there was a call for GP solutions to be evaluated based on their complexity and interpretability [105], as GP solutions are naturally interpretable. Interpretability, and XAI in general, is subjective [62, 115]. There are many stakeholders in XAI [96], making it hard to take a well-representing user study. One convenient metric to measure complexity, a key part of interpretability, is the expression size (ES), in this case, the number of nodes that make up an expression. GP solutions that are too complex have a reduced interpretability. ES is convenient as different solutions can easily be compared. Still, it does not convey the complexity of interpretability, but, in lack of a better metric, we will use this metric as a basis. In addition to ES, we will have a discussion on interpretability, in line with the proposal of Liao et al. [62].

ES was used by Batista et al. [9] and for the GECCO'22 Interpretable Symbolic Regression for Data Science Competition [55].

```
S      ::= <Sa>
         | <b>
<a>    ::= "ha"
<b>    ::= "ho"
         | "hi"
```

Listing 2.1: Example of a simple grammar.

2.2 Formal Grammars

Formal grammars, or just *grammars*, describe how to form *words*, or *strings*, according to the syntax of a language, given the *symbols* of that language, also known as its *alphabet*. Grammars can be used to formally define a solution space, to avoid exploring programs that are not valid in the domain. This way, a solution space can be more-efficiently traversed.

Grammars consist of a set of *production rules* that specify how a particular word on the left-hand side (LHS) of the production can be rewritten into another word on the right-hand side (RHS). Production rules are defined by a set of *terminals* symbols, a set of *non-terminal* symbols, a set of production rules, and a *starting* symbol. Rewriting starts from a starting symbol. Rewriting is straight-forward and can best be shown through an example. Let's look at the example grammar in listing 2.1 which is written in Backus-Normal-Form (BNF) [52].

From the starting symbol S, there are two production possibilities, either Sa or b. Productions are denoted by $::=$. This grammar is *recursive*, as one production possibility of S contains S. As the grammar is recursive, a production might never stop. Choosing the first possibility, Sa, leaves us with two symbols, that both have different production possibilities, S and a. We can rewrite S as Sa or as b, and a as ”ha”. As the production Sa of S is recursive, the production process will only end once b is chosen. Possible productions are $baaa$, ba, $baaaaaa$, and many more. $baaa$ would then produce either ”hohahaha” or ”hihahaha”.

In this thesis, I use Context-Free Grammars (CFGs). CFGs distinguish themselves with productions that do not rely on their *context*, i.e., surroundings. In other words, each production rule only sees a single non-terminal symbol at the LHS. CFGs are more restricted than general grammars, but are much faster in terms of computing time.

Chapter 3

Related Work

In this chapter, I discuss research related to this thesis. I start by introducing solution-space restriction methods in GP in section 3.1. In section 3.2, I introduce some basic AutoML concepts. I then move on to existing FL research in section 3.3, and I discuss GP-based FL in section 3.4. Finally, I introduce already-existing, GP-based FL frameworks in section 3.5.

3.1 Constraining the Solution Space of GP

As GP is a search-based method, its main drawback is the time needed to explore vast solution spaces. To restrict the solution space, multiple methods have been introduced [35, 16]. Most-commonly used are Strongly Typed GP (STGP) [70] and Grammar-Guided GP (GGGP) [106]. STGP uses types to restrict interaction between functions and terminals, thus only defining valid trees at tree construction. GGGP uses a grammar to define what are valid trees. To restrict the solution space in this work, I use the GGGP approach through Genetic Engine, which uses an STGP front-end for the grammar definition. Genetic Engine is individual-representation independent, allowing for the use of various GGGP methods, such as Context-Free Grammars GP (CFG-GP) [108], Grammatical Evolution (GE) [87], and Structured GE (SGE) [63]. A more elaborate discussion on different GGGP methods is given in section 5.2.

Solution-space restriction can also be done by validating the tree before fitness evaluation [67]. By assigning invalid trees the worst possible fitness, they are naturally not selected throughout the search process. STGP and GGGP are more efficient because they only generate valid trees at each branch, so there is no time wasted generating full invalid trees, only to be discarded right away. Another approach is not to discard invalid trees, but to repair them [87].

3.2 AutoML

AutoML, and more particularly FL, is a broad area. He et al. [39] split ML pipelines into four modules: data preparation, FL, model generation, and model evaluation. They, and others [116, 43], consider any automatization within any of these parts as AutoML. Extending their argumentation, libraries as scikit-learn [82] and Keras [20] could be considered AutoML, even though they only include limited pipeline-optimization functions. There is a broad selection of AutoML frameworks that aim to optimize the complete ML pipeline. Some use GP [76, 22, 69] and others use different optimization methods [33, 98, 37, 50], like *Bayesian optimization*, *meta-learning* and *ensemble construction*. Furthermore, there have been extensive surveys and other resources concerning AutoML recently, that encompass all AutoML subfields [39, 116, 43]. In this work, I focus only on FL and, more specifically, on GP-based FL.

3.3 Feature Learning

Feature Learning (FL)[1] can improve the quality of the solution space by “decreasing the dimensionality, speeding up the learning process, and enhancing [the] algorithm’s performance” [97]. Simply said, FL aims to create a ’better’ feature set from the *original features*. Generally, ’better’ refers to either a simpler feature set, a feature set that improves a prediction metric, or a feature set that is more easily interpretable. The automation of FL is considered “one of the holy grails of [ML]” [25], and it leads to the discovery of hidden relations between features [90].

A famous automated FL method is Principal Component Analysis (PCA) [110]. PCA is based on formatting data to a smaller dimension space. The variance of the data over each dimension expresses something over the importance of that dimension. Values for less important dimensions can be discarded, reducing the number of features. Similarly, Self-Organising Maps [53] create lower-dimensional representations of data. Both methods fail at allowing the implementation of domain-specific knowledge, the input to be heterogeneous, and entity aggregation. Furthermore, the obtained features are not very hard to interpret.

Two well-known FL frameworks are FeatureTools [51] and AutoFeat [42]. FeatureTools was created to easily combine different relational datasets and extract aggregated features from them. One interesting aspect of the framework is the possibility to aggregate past data based on a feature value, and incorporating it into a new feature. Automatic optimization of the feature set in FeatureTools is very basic, only incorporating a simple feature selection method. On the other hand, AutoFeat works on non-relational data, and doesn’t allow for aggregations. Interestingly, AutoFeat allows a minimal implementation of domain knowledge to disallow nonsensical features. AutoFeat supports some feature optimization functionalities based on sklearn’s Lasso method LassoLARS [100].

[1]For a formal definition and broad survey of FL, see [92].

3.4 Feature Learning using GP

Early examples of GP for FL used a simple function set and obtained promising results [90, 34]. Generally, the function set consists of addition, subtraction, multiplication, and division. The terminal set contains all the possible features, and possibly constants. Each individual represents a single learned feature, which is evaluated on its ability to represent the whole data point.

Later, more complex work on FL using GP showed GP's power to enhance classification performance [101] and its ability to construct complex relations [85]. In this section, we discuss different approaches to GP-based FL.

Individual shapes In GP-based FL individuals generally have either of two shapes, "Single-[Feature]-per-Individual" (SFI) or "Multiple-[Features]-per-Individual" (MFI) [36]. SFI individuals evolve a single feature from the original feature space, whereas MFI individuals evolve a set of features. Consisting of a single feature, SFI individuals are overall simpler to interpret, but cannot easily find complex inter-feature relations like MFI. Performance of SFI and MFI was compared by Tran et al. [103] and Ain et al. [3], and MFI was concluded to perform better.

Fitness functions Across GP applications, fitness functions differ vastly. Espejo et al. [30] identify two approaches to defining the fitness function, the *filter* approach and the *wrapper* approach. These approaches are distinguished by whether the fitness function is model-dependent, or not. Examples of the wrapper approach use decision trees [2, 103] and XGBoost[19]. The filter approach normally uses an independent statistic to measure fitness. Examples of statistics used in the filter approach are Information Gain [80, 72, 71], Fisher's criterion [38, 1], the Gini Index [72, 71], and class-separation [9]. The filter approach is said to be more efficient [74, 80], and, being model-independent, it introduces no bias to the FL process. Furthermore, Tran et al. [103] researched and compared a filter and a wrapper approach, returning better results for the filter approach.

Support of domain knowledge Incorporating domain knowledge in GP-based FL can enhance performances and interpretability. Domain knowledge for operator definition has improved performance [61]. By integrating domain knowledge into a grammar, Cherrier et al. [19] used GGGP to improve performance and interpretability for FL. Sadly, the method is not general, but domain specific.

Support of heterogeneous input data Interestingly, FL function and terminal sets are generally similar throughout research [30], only incorporating homogeneous input data. Breaking with the homogeneous-input-data convention is Vectorial GP (VE-GP) [6], a GP-based framework specifically created for *panel datasets*. They define panel datasets as "[collections] of observations for multiple subjects at different equal-spaced time intervals" [6]. VE-GP supports solving problems that consist of both scalar and vector data.

Support of entity aggregation Another interesting implementation detail of VE-GP, is that it allows aggregation of multiple values of a subject over time. Despite possible benefits, VE-GP does not extend to aggregating over other features than the subject, nor does it support other data types than scalars and vectors. It also requires the to-be-aggregated data to be available as a vector. If this data is historic data available in other parts of the dataset, then this requires a preprocessing step. Furthermore, the data must be available for each data point, which results to duplicate data.

Another attempt at automatically aggregating entities was done by Song [93] in AutoFE. Firstly, new features are exhaustively generated by combining terminals through a set of operators. AutoFE generates new features by (1) discretizing them, by (2) combining features through standard arithmetic operators, and by (3) feature aggregation after grouping on another discrete feature. The generated features then undergo an evolutionary-based FS process. This process indirectly optimizes a feature set by first expanding the feature set and then selecting relevant features. My method directly optimizes the aggregation operator itself. As such, not all feature combinations need to be generated beforehand, making the process more efficient. Furthermore, in AutoFE the aggregation can only be done over all data, whereas in my method I aggregate over historic data. Moreover, I evolve the number of data points included in the aggregation, making my method more dynamic.

3.5 Existing GP-based FL Frameworks

Many GP-based FL methods have been introduced. Muñoz et al. [73] proposed M3GP, a multi-class classification GP method that extends on previous work called M2GP [45]. It distinguishes itself with its individual representation, which is an MFI representation, and its wrapper fitness function. The fitness function first calculates the cluster centroids of each class in the new feature space. Using the Mahalanobis measure, each data point is assigned the class of the closest centroid. The fitness is the accuracy of this classification. M3GP has been shown to have excellent performance.

Two years later, Cava et al. [17] presented M4GP, improving M3GP. It introduces a stack-based program representation, multi-objective parent-selection techniques, and an archiving strategy. The stack-based program representation contributes to simpler models, making M4GP more efficient. The different multi-objective parent-selection techniques have different purposes. Where *lexicase* selection [95] rewards solutions that perform well on hard cases (generally data points), *age-fitness Pareto survival* prefers solutions in less explored areas of the search space, enhancing the diversity of populations. Above enhancements of M3GP implemented in M4GP were not part of this thesis.

Feature Engineering Automation Tool (FEAT) [58] is a multi-objective, GP-based FL method that learns features for a linear regression model. It combines six different perturbation methods, four mutation variations, and two crossover variations. Four objective functions are incorporated that aim to optimize three solution aspects; the mean square error; the complexity of the solution; and the disentanglement of the solution. Complexity is optimized based on a function that sums the complexity weights of operators within a subtree. Here, the authors assign the complexity weights. Disentanglement is measured through two functions, one measuring bivariate correlations; the other captures higher-order dependencies.

Notice that measuring complexity and interpretability are recurring aspects of GP-based FL methods, as they are paramount advantages of using GP for FL. In addition to above methods to improve on these aspects, adding a preference to smaller trees in the fitness function is a usual sight in literature, for example done by Tran et al. [102] and Li and Yang [61].

Like FEAT, there are other GP-based symbolic regression tools, such as Python's GP-GOMEA [104] and C++'s OPERON [15]. These tools perform very well on symbolic regression, but don't directly support FL.

Chapter 4

Case Studies

To guide the explanation of the method, we are considering two ML problems as case studies. The first problem is a regression problem for the prediction of bicycle usage. The second one is a classification problem to predict whether the credit risk of a person is high or low.

4.1 The BoomBikes dataset

The BoomBikes dataset [10] was created for the fictitious bike-sharing company BoomBikes after the Corona pandemic revenues dropped, and BoomBikes wanted to have a clearer view on the demand for shared bikes. With the dataset made available, BoomBikes aimed to obtain an interpretable prediction model that would (1) "predict demand with high accuracy", and (2) give "insight into the significant relationships that exist between demand and available predictors". The resulting analysis consists of a lengthy, manual process of, amongst other steps, data processing, outlier correction, and feature selection, before a linear regression model is trained. The final regression model is

$$\begin{aligned}\text{cnt } =& 2392.0791 + 1946.7864 \text{ yr } + 444.4907 \text{ Saturday } + 466.0136 \text{ winter}\\ &- 890.3115 \text{ july } - 1063.6669 \text{ spring } + 296.8008 \text{ workingday}\\ &- 1749.8275 \text{ hum } + 4471.6602 \text{ temp } - 1110.3191 \text{ windspeed}\\ &- 1273.7519 \text{ light snow/rain.}\end{aligned}$$

Notice that the model is not very complex and gives some insight into the effect of certain feature values. In this thesis, I am not further concerned with linear regression.

4.1.1 Dataset specifics

The dataset contains 730 instants and 16 features: *instant*, *dteday*, *season*, *yr*, *mnth*, *holiday*, *weekday*, *workingday*, *weathersit*, *temp*, *atemp*, *hum*, *windspeed*, *casual*, *registered*, *cnt*. Feature *cnt*, a discreet, positive, integer value, is the feature we want to predict (see it depicted over time in fig. 4.1). It is the sum of the features *casual* and *registered*. Because predicting *cnt* would be easy from those features, they are discarded, leaving us with 13 features and the target. The first 547 (~75%) instants are used as training set, and the last 183 (~25%) instants comprise the validation set.

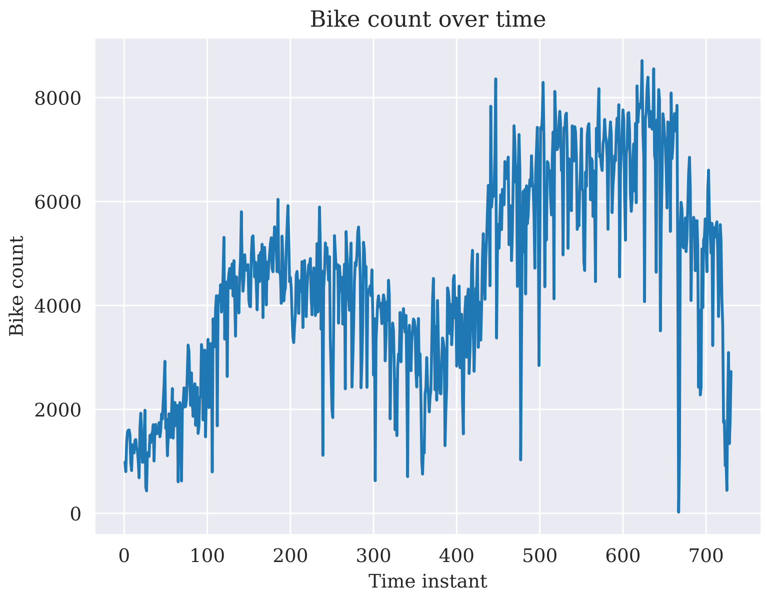


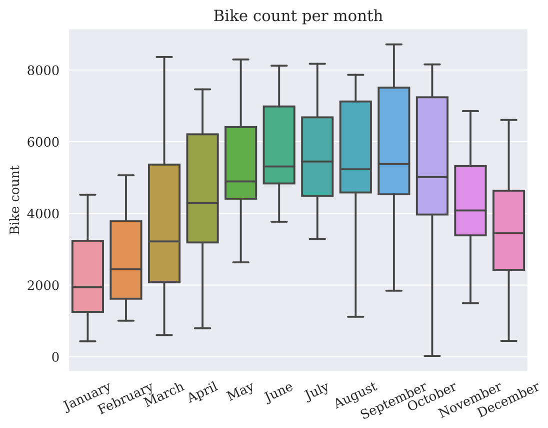
Figure 4.1: Bikes counted over time.

The BoomBikes dataset is a time series with seemingly homogeneous data, where all data are floats. As such, general FL frameworks can be applied to the dataset. Further inspection unfolds that certain features should be considered as categories or booleans, e.g., season, day of the week and holiday. Regarding this as domain knowledge, I can use my method to acknowledge these features as categories instead of floats, and in this way incorporate domain knowledge. Furthermore, my method can use historic data in the construction of new features, as it is a time series. Historic data can be aggregated with respect to any feature, or not. Altogether, the BoomBikes dataset allows us to test all aspects of my method, incorporating domain knowledge, and entity aggregation.

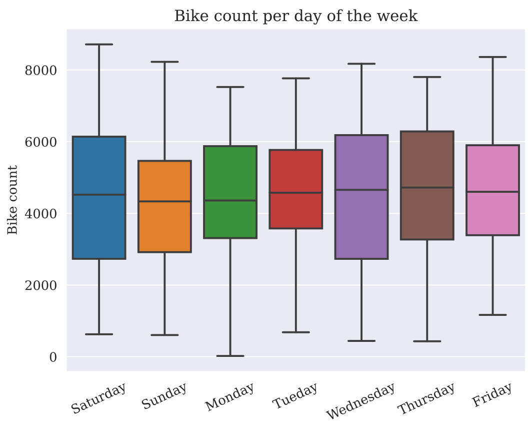
4.1.2 Data analysis

The data is very irregular through time, as you can see in fig. 4.1. Each time instant concerns a day. The data encompasses two years, starting on January first, 2018. A summer peak is visible twice, with a valley around the end of the year, the 365th day. Furthermore, there is a clear increase visible in the second year in comparison with the first.

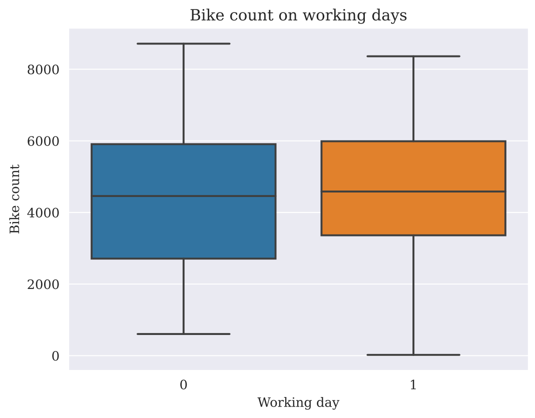
Let's look more in-depth at some features in fig. 4.2. Figure 4.2a repeats the information retrieved from fig. 4.1, namely more bikes counted during the summer. In fig. 4.2b we see relatively fewer bikes counted on Sunday. Figures 4.2c and 4.2d show the same information, more bikes are counted on working days and non-holidays. Differences are minimal, except for the difference in bikes counted on holidays and non-holidays.



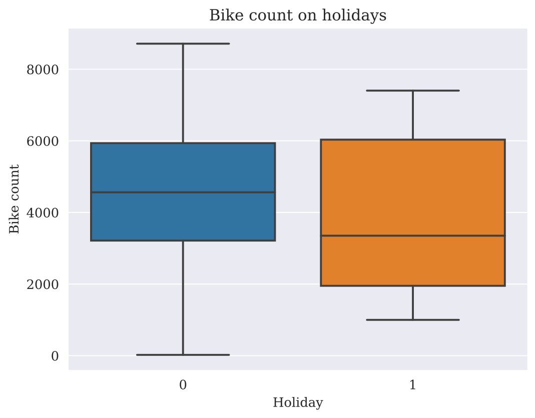
(a) Bikes counted in each month.



(b) Bikes counted on each day of the week.



(c) Bikes counted on (non-)working days.



(d) Bikes counted on (non-)holidays.

Figure 4.2: Bike count analysis for different kinds of days.

4.2 The credit-g dataset

The credit-g dataset [27] contains financial and general attributes of a group of people from Germany. People are classified as low or high credit risks. I took the dataset from the PMLB benchmark [77]. No solutions are proposed for the classification of this dataset.

4.2.1 Dataset specifics

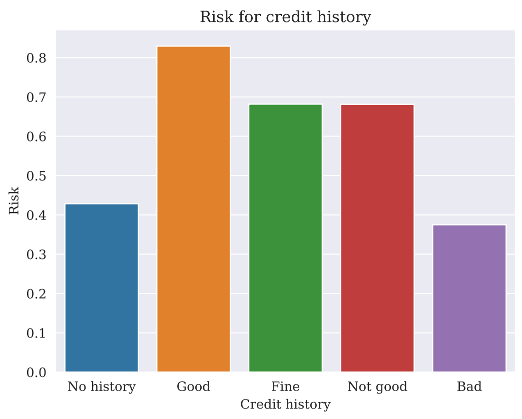
The dataset contains 1000 data points and 21 features: *checking_status, duration, credit_history, purpose, credit_amount, savings_status, employment, installment_commitment, personal_status, other_parties, residence_since, property_magnitude, age, other_payment_plans, housing, existing_credits, job, num_dependents, own_telephone, foreign_worker, target*. Feature *target* is a boolean value, stating whether a person has a high (0) or a low (1) credit risk. The dataset is imbalanced, as ~70% is classified as low-risk, versus ~30% being classified as high risk. The first 750 (75%) instants are used for training the models, and the last 250 (25%) is used for testing.

In the PMLB benchmark all data is numerical, so also this dataset. However, the original dataset source shows that most data is not numerical, but categorical. I define this information as domain knowledge, and, using my method, I incorporate the information in the search process. The dataset is a panel, as each data point specifies an individual. Aggregation is done on previously analysed individuals in the dataset.

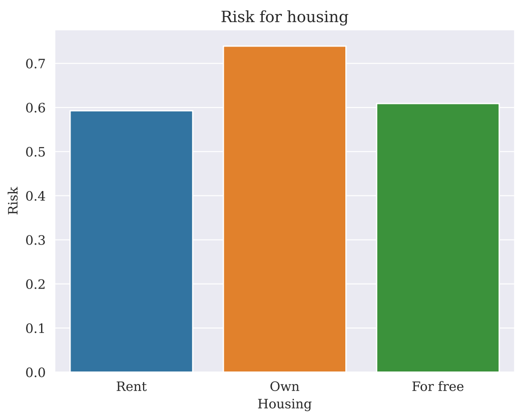
4.2.2 Data analysis

In fig. 4.3, I take a closer look at four features and the distribution of risk over its categories. In fig. 4.3a a clear correlation between one's credit history and risk score. The better is one's credit history, the lower is one's the risk. Similar sensible trends are seen in figs. 4.3b and 4.3c; owning a house results to lower credit risks in comparison to renting or free housing; and unemployed individuals tend to have higher risk in comparison to employed individuals.

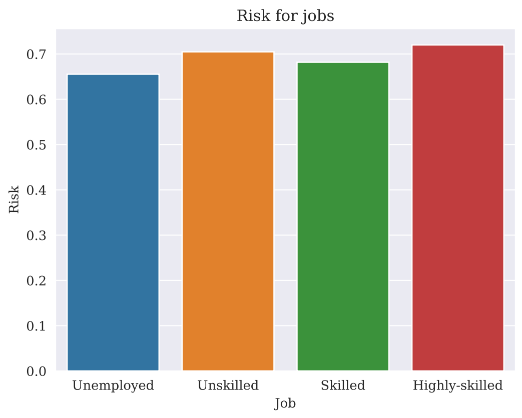
Figure 4.3d depicts something especially peculiar. Throughout the data, women face higher risks than men, on average. In a world where women are systematically underpaid in comparison to men [11], occupy less prestigious jobs, and have a harder time to be promoted [75], it is expected for women to be classified, on average, as being a bigger risk to provide credit to. Even so, one's sex has nothing to do with one's credit risk; the data is only the symptom of a sexist society. AI models that use the sex of an individual to determine their risk, indirectly inherit this sexism. In AI models that lack interpretability, these undesired side effects can go unnoticed. In my method these issues can be easily spotted and resolved.



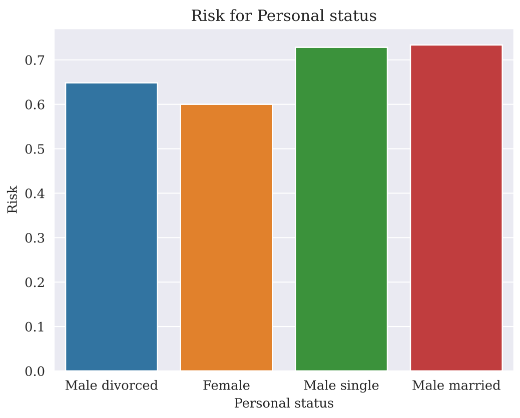
(a) Risk for each credit history.



(b) Risk for each housing.



(c) Risk for each job.



(d) Risk for each personal status.

Figure 4.3: Risk analysis for different individual features.

Chapter 5

Genetic Engine

Genetic Engine [29] is a GGGP framework that allows the user to encode the constraints of a problem independently of the individual representation. In this chapter, I will start with presenting the motivation for Genetic Engine and some relevant features of Genetic Engine in section 5.1. Then I will discuss search-space restrictions in section 5.2, and how to use the framework in section 5.3. In section 5.4 I show the results of comparing Genetic Engine to PonyGE2 [32], the most widely used GGGP framework in Python. Finally, I list my contributions to Genetic Engine in section 5.5.

5.1 Motivation and overview

Genetic Engine was developed as an all-purpose GGGP framework that has an only Python front-end. The aim of Genetic Engine is to create a high-performance GGGP framework that can easily be used by data scientists. Genetic Engine distinguishes itself from other GGGP frameworks because of three characteristics.

Firstly, users can define grammars using Python classes in a STGP way. As the fitness function is also defined in Python, Genetic Engine is a Python-native library. This allows people without knowledge of BNF, or any of its extensions, to implement GGGP easily. Moreover, using Python allows the use of standard development tools like linting and type checking, improving the usability of the framework, one of the key challenges for GP [89, 79].

Secondly, Genetic Engine is individual-representation independent, allowing users to choose different individual representations to their convenience. Users can even define new individual representations to their convenience.

The third notable feature is the implementation of Meta-Handlers, a dynamic refinement [35] that restricts the nodes of the tree.

5.2 Search-space Restrictions

Being a search-based method, GP gains from the restriction of the solution space. As discussed in section 3.1, there are various methods to do so. Even though the front end of Genetic Engine uses types, as in STGP, the actual algorithm works in a GGGP manner, i.e., it uses grammars to guide the search process. In this section, I introduce GGGP.

As discussed in section 2.2, a grammar is a set of rules for rewriting objects, conforming to some syntax. In GGGP, grammars are used to define what valid individuals look like. In other words, the grammar is a set of rules for creating individuals, and, as such, defines the validity of an individual. There are multiple GGGP methods. The most-widely used methods are the original Context-Free Grammars GP (CFG-GP) [108] and Grammatical Evolution (GE) [87]. Both use Context-Free Grammars to guide the search process, even though the names suggest otherwise. They mainly differ on the individual-representation method. The individuals in CFG-GP are syntax trees, of which the construction is guided by a grammar. GE uses a linear string to represent an individual, which is called the *genotype* of the individual. Starting from the grammar's starting symbol, the characters of the linear string can be used to select productions step by step. This results in a syntax tree, i.e., the actual individual, or the *phenotype*. The mapping of the genotype to the phenotype is called the *genotype-to-phenotype* mapping.

Research has shown that CFG-GP performs better than GE [107, 86, 65]. Furthermore, GE faces issues with the locality of its off-spring [86], the redundancy in its genotype [78, 86], and invalid individuals [80]. Still, GE is one of the most widely used GGGP methods[66]. Ingelse et al. [47] have argued that this is mainly because of the simplicity of implementation and performance gains of the evolutionary operations on linear strings, the representation of GE individuals. Another reason is that, due to the decoupling of the search algorithm and the problem at hand through the genotype-to-phenotype mapping, algorithms can be reused for different use cases. These implementation advantages come at a cost in performance. As part of this thesis, [47] discussed the advantages and disadvantages of both CFG-GP and GE, and introduced various advantages of CFG-GP not proposed before in research.

Recently, Structured GE (SGE) [63] was proposed to alleviate the above-mentioned issues of GE. SGE uses a different individual representation from GE. SGE structures the linear string, separating it into substrings specific to each production rule. If a production rule can be passed only once, the substring has a length of one, if it can be passed twice, the substring has a length of two, etc. This method requires a preprocessing step to find the maximum length of each substring. Furthermore, notice that for recursive grammars, one must predefine the maximum recursion for each recursive step, as to prevent a never-ending production to be constructed. Overall, SGE wins on locality and reduces redundancy, but loses on simplicity when compared with GE.

Lourenço et al. [64] later proposed Dynamic SGE (DSGE), in which the substrings are grown as needed while constructing individuals. DSGE gains on locality and reduces redundancy in comparison with SGE. It requires a user-defined maximum tree depth instead of a maximum recursion for each recursive production, making it more-easily comparable with other GGGP methods.

Genetic Engine focusses on CFG-GP, but also supports GE and an SGE version. As the user can easily switch between individual representations, the implementation is said to be individual-representation independent. The magic of individual-representation independence in Genetic Engine derives from the equivalence between, on the one hand, generating an individual in CFG-GP based on a sequence of randomly generated production choices, and, on the other hand, the phenotype-to-genotype mapping in (S)GE based on a beforehand-generated (set of) linear string(s) that define the choices. In other words, individual generation in CFG-GP and genotype-to-phenotype mapping in GE and SGE are implemented by the same code. As such, Genetic Engine can be used to fairly compare different individual representations in GGGP.

5.3 Genetic Engine usage

In this section, I elaborate on show to use Genetic Engine. Genetic Engine requires the user to define two things in Python; the grammar through Python classes; and a fitness function to guide the search process. Grammars can have more expressive productions, through the usage of Meta-handlers. Finally, I show all the features that Genetic Engine currently supports.

5.3.1 Grammar definition

The grammar is defined through Python classes. Classes represent terminal and non-terminal nodes, one of which needs to be identified as the starting symbol. From the starting symbol and the nodes, Genetic Engine automatically extracts the grammar. See listing 5.1 for a simple classification example. For simplicity, the example only contains the summation and multiplication operators ”+” and ”*” as non-terminals, and the dataset features and integer constants as terminals.

The *Number* node is the starting symbol and extends all other nodes. Through this extension, Genetic Engine denotes the productions of a class. Looking at the *Plus* node, we see it has two arguments, *left* and *right*, both *Numbers*. Evaluating the *Plus* node evaluates both arguments and sums them. The other function node *Multiply* is implemented equivalently. The terminal nodes, *Variable* and *Literal* both have single *Annotated* arguments, which implements Meta-Handlers. In section 5.3.3 I discuss Meta-Handlers.

Number is the starting symbol, and, together with the nodes *Plus, Multiply, Variable*, and *Literal*, the grammar is extracted by the *extract_grammar* function. The *extract_grammar* function iterates over all the nodes and registers its possible productions and the productions it extends. See listing 5.2 for the extracted grammar in BNF.

As is standard in GGGP, Genetic Engine now used the grammar to generate trees. When using CFG-GP, the grammar is used directly at individual generation, when generating tree individuals, and for the evolutionary operations. On the other hand, when using GE or SGE, it is only used for the phenotype-to-genotype mapping.

5.3.2 Fitness function definition

The fitness function definition is quite straight-forward. The fitness function should take in a candidate solution and return the fitness of that candidate. See listing 5.3 for an example of a fitness function for classification. First, the solution is evaluated on the preprocessed dataset of X, obtaining *y_predicted*, the predicted target values of each data point in X. Then *y_predicted* is compared to the ground truth y using some metric, in this case the F_1-score. Finally, the F_1-score is returned as the fitness.

Here we quickly return to the evaluation of the *Variable* node in listing 5.1, which returns a column of the *DATASET*. This is a simplified way notation of the true implementation. The data is actually passed through the tree with the evaluation functions. This is done using Python *kwargs*. The preprocessing in line 5 refers to creating a dictionary of the dataset that can be passed through the evaluation functions.

5.3.3 Meta-Handlers

A novel approach introduced in Genetic Engine is the use of Meta-Handlers. Meta-Handlers allow the user to overwrite the production rules to a more informative format, i.e., they allow the introduction of Dependent Types. For example, in line 27 of listing 5.1, you see the integer type Annotated with an *IntRange* Meta-Handler. The *IntRange* Meta-Handler allows the user to define maximum and minimum values for the integer type, in this case 0 and 9. At tree generation, the integers selected for this production will be in between 0 and 9. See listing 5.4 for the implementation of the *IntRange* Meta-Handler. Other Meta-Handlers are easily created by the user.

Another Meta-Handler can be found in line 21 of listing 5.1. Here, the user defines which columns of the data are included in the classification process. Meta-Handlers are not limited to defining *generate* methods, but can also specify *mutate* and *crossover* methods. These replace the standard evolutionary operations throughout the evolutionary process.

Meta-Handlers can also be used to implement specific grammars. For example, Meta-Handlers may define probabilistic grammars as used for Probabilistic GE (PGE) [68]. Furthermore, they may be used to introduce Refined Types in the grammar, which can be beneficial for GGGP [35]. Adding Refined Types is a prospected enhancement for Genetic Engine.

5.3.4 Genetic Engine features

Genetic Engine includes three search-based algorithms: Random Search, Hill Climbing, and GP (see section 2.1.1 for an introduction). The GP algorithm is the most elaborated. It is contained within a GP object, which can be configured with several options, shown in table 5.1. Furthermore, the CFG-GP implementation supports *in-node storage*, as proposed by Ingelse et al. [47] as part of this thesis. The in-node storage is currently used to keep track of the number of nodes and the depth of each individual.

In the future, we intend to boost the performance of Genetic Engine by saving node evaluations in each node.

Except for the possibility for users to implement their own GGGP models, Genetic Engine comes with some off-the-shelf features. Firstly, there are many grammar terminals and function predefined for symbolic regression, string matching, basic arithmetic and programme synthesis. Secondly, I implemented a GP-based symbolic regression function compatible with the scikit-learn [82] estimator API. This was done in light of the SRBench Competition at GECCO'22[1] in which we participated along with 12 other participants in two tracks; the synthetic track; and the real-world-application track. In the synthetic track we ended 7th, and in the real-world-application track we ended 3rd. The scikit-learn API contributes to the necessity of GP to be ”integrated into standard software development infrastructure”, as promoted by Sobania et al. [91] for programme synthesis, but as important for the area of symbolic regression.

5.4 Performance Comparison against PonyGE2

As part of the introduction of Genetic Engine, we compared its performance against PonyGE2 [32], a state of the art framework for Grammar-Guided GP in Python. PonyGE2 is a Python library that implements GE and where the user is required to write a fitness function in Python and a grammar in BNF.

We compared the performance (considering fitness) in five benchmarks: a classification task; a regression Task; VE-GP [6], a string matching problem, and Conway's game of life problem [21]. We also implemented a large variety of examples in Genetic Engine to compare with implementations in PonyGE2, but they sadly didn't work in PonyGE2. As part of that, we implemented a program synthesis framework based on an implementation in PonyGE2.

The algorithms were started under the same conditions considering initialization method, population size, operator probabilities, and a budget of 60 seconds. The code for replicating these experiments is available at `https://github.com/pcanelas/GeneticEngineEvaluation`.

Figure 5.1, shows the distribution of the final fitness in both frameworks, where higher is better. PonyGE2 and Genetic Engine have comparative performance, as neither one is clearly better than the other overall.

We also looked at expressive power of the grammar definition and usability of both Genetic Engine and PonyGE2. Because of Meta-Handlers, Genetic Engine can express more complex grammars than the BNF of PonyGE2. Firstly, probabilistic grammars can be completely and more easily expressed in Genetic Engine. Probabilistic grammars cannot be completely expressed in PonyGE2 as exact expression of irrational probabilities is not possible. Moreover, it is easier to implement in Genetic Engine as BNFs require repeating productions in the correct partitions to create probabilistic grammars, whereas Genetic Engine only needs a single number. Secondly, the future incorporation of Refinement Types gives Genetic Engine a clear advantage in expressivity regarding its grammars.

[1]https://cavalab.org/srbench/competition-2022/

Table 5.1: The variables of the GP object of Genetic Engine.

Feature	**Description**	**Type**
Grammar	This object contains all the grammar information necessary for the tree generation. See section 5.3.1 for more information.	Grammar
evaluation_function	A function that takes in a solution and returns the fitness of this solution. See section 5.3.2 for more information.	Solution $\rightarrow$ float
representation	The individual representation you wish to use. Currently, implemented are CFG-GP, GE, and SGE.	Representation
randomSource seed	A function that takes in an integer (the seed) and returns a random source. The seed of the random source	function integer
population_size	The population size used during evolution.	integer
n_elites	Number of individuals that are directly reproduced in the next generation.	integer
n_novelties	Number of novel individuals introduced in each following generation.	integer
number_of_generations	Number of generations to run the algorithm.	integer
max_depth	The maximum depth the trees that represent each individual may have.	integer
favor_less_deep_trees	Allows the user to favor less deep trees. This gives a small penalty to deeper trees so that simpler solutions are favored.	boolean
selection_method	The selection method used by the algorithm. Currently, only tournament selection is implemented.	tuple
probability_mutation probability_crossover	Probability of occurence of mutation and crossover.	float [0,1]
specific_type_mutation specific_type_crossover	Type given preference in mutation and crossover.	Node type
hill_climbing	Whether to use hill climbing in the mutation process.	boolean
minimize	Select whether the fitness function must be minimized.	boolean
target_fitness	Gives the target fitness. Evolution stops when reached.	float
force_individual	Allows the user to introduce a custom individual.	-
timer_stop_criteria timer_limit	Make the algorithm stop after the timer has stopped. Limit of the timer.	boolean integer (s)
save_to_csv	Given a csv path, individual trees are saved.	string

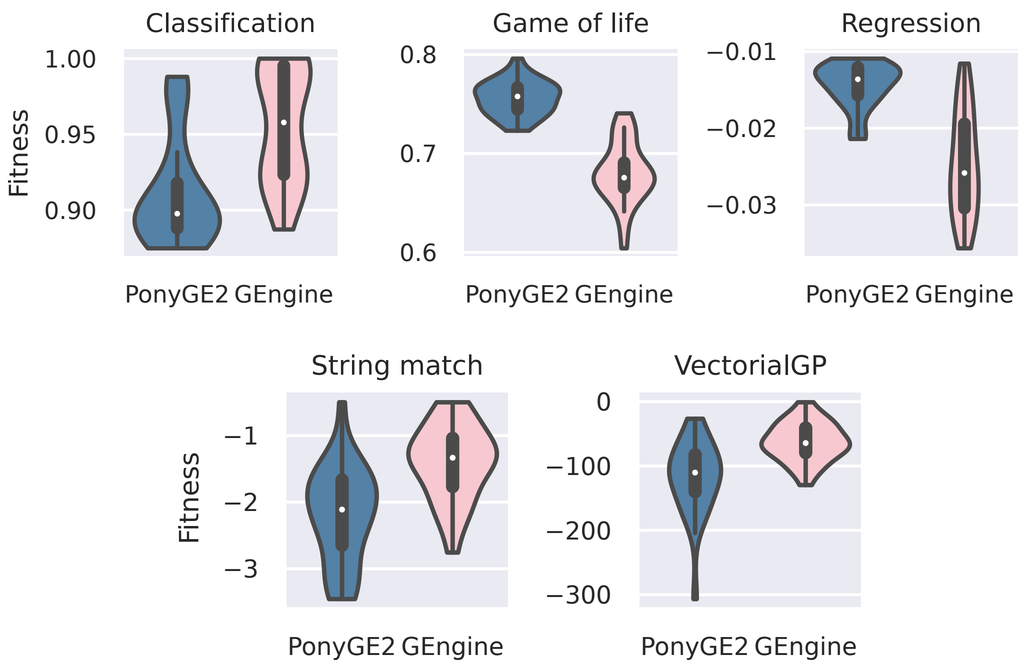


Figure 5.1: Comparison of Genetic Engine and PonyGE2 on five benchmark problems. A higher fitness corresponds to a better performing solution.

Concerning usability, we argued that Genetic Engine is more user-friendly than PonyGE2. Again, Meta-Handlers increase usability with easy implementation of complex grammars, and the possibility of implementation of customized mutation and crossover for certain nodes. Furthermore, the fact that Genetic Engine is completely Python native gives its users multiple advantages. For example, the user can use functionalities from its code editor directly at production time, such as linting, autocompletion, and error checking. A BNF file is only read at runtime, and thus does not have this advantage.

5.5 Personal contributions

Except for overall assisting in the backend implementation of Genetic Engine I had the following concise contributions:

- Implementation of various examples.
- Implementation of the HC mutation and stand-alone algorithm.
- Numerous functionalities of the GP object, namely,

  - favouring of less deep trees, to reduce tree size,

  - the either-mutation-or-crossover evolutionary-operations method, in line with Batista et al. [9],

  - specific type mutation and crossover, to allow the user to define node types to be favoured for mutation and crossover,

  - an adjustable timer limit, and

  - the possibility to save generation information to CSVs with all relevant sub-functionalities (genotype saving, solely recording of the best individual, saving test data results).
- The Structured GE representation.
- A large part of the ready-to-use grammar.
- Personalized mutation and crossover through Meta-Handlers.
- The Meta-Handlers IntList, FloatList, ListSizeBetween with personalized mutation and crossover, in line with Batista et al. [9].
- The off-the-shelf classifier and regressor methods of GP and HC.
- Helping with writing a paper on Genetic Engine.
- A large part of the documentation.

```python
@abstract
class Expr:
    pass

@dataclass
class Plus(Expr):
    left: Expr
    right: Expr
    def evaluate(self):
        return self.left.evaluate() + self.right.evaluate()

@dataclass
class Multiply(Expr):
    left: Expr
    right: Expr
    def evaluate(self):
        return self.left.evaluate() * self.right.evaluate()

@dataclass
class Variable(Expr):
    name: Annotated[str, VarRange(["x", "y", "z"])] # features of the dataset
    def evaluate(self):
        return DATASET[self.name]

@dataclass
class Float(Expr):
    val: float
    def evaluate(self):
        return self.val

grammar = extract_grammar(
                nodes = [Plus, Multiply, Variable, Literal],
                starting_symbol = Expr,
            )
```

Listing 5.1: Grammar definition of a simple grammar that can be used for classification.

```
<Number>    ::= <Plus>
              | <Multiply>
              | <Variable>
              | <Literal>
<Plus>      ::= <Number> + <Number>
<Multiply>  ::= <Number> * <Number>
<Variable>  ::= x1 | x2 | x3
<Literal>   ::= 0 .. 9
```

Listing 5.2: Extracted grammar from the grammar definition in listing 5.1.

```
    def fitness_function(solution: Solution):
        X = data
        y = target

        variables = preprocess(X)
        y_predicted = solution.evaluate(variables)

        fitness = f1_score(y_predicted, y)

        return fitness
```

Listing 5.3: Fitness function for classification.

```
class IntRange(MetaHandlerGenerator):
    def __init__(self, min, max):
        self.min = min
        self.max = max

    def generate(self, random_source)
        random_source.randint(self.min, self.max)
```

Listing 5.4: Integer range Meta-Handler implementation.

Chapter 6

Domain-Aware Interpretable Feature Learning

In this chapter, I will present the proposed approach to apply Domain-Aware and Aggregation-based GP-based FL. I start by discussing Traditional GP and its extension M3GP [73] in section 6.1. Finally, I introduce two new FL methods; M3GP with Domain Knowledge (DK-M3GP) which allows the incorporation of domain knowledge to restrict the search space (in section 6.2); and DK-M3GP with Aggregation (DKA-M3GP) in which aggregation over historical data is made possible (in section 6.3).

6.1 Traditional GP & M3GP for Regression

When applying Traditional GP (also known as standard GP) to FL, the GP algorithm evolves a mapping from the original feature set to a one-dimensional feature set. That mapping is a function that can be composed of a function and terminal set. The function set consists of the following functions: addition, subtraction, multiplication, and *safe* division. Safe division refers to a division method, that is slightly adapted to be ”safe” from division by zero. When a division by zero occurs, safe division replaces the result with a one: for example, $\frac{a}{0} = 1$. The terminal set consists of the original features of the dataset (see chapter 4). In line with the M3GP implementation of Batista et al. [9], I chose not to use constant values. The above is interpreted as a grammar in listing 6.1. Notice that the evolved one-dimensional feature is called a *BuildingBlock*, the standard name for (partial) features that are learned.

The fitness function differs for the two datasets. For the BoomBikes dataset with a numeric target, the fitness of the feature mapping is the quality of the prediction made by a Decision Tree (DT) regressor with depth 4 that was trained and applied using the resulting feature. As such, the fitness function is a wrapper function of a DT regressor with depth 4, similar to the method done by Ain et al. [2]. The simplified code of the fitness function is shown in listing 6.2. For the credit-g dataset with a class target, I use a DT classifier with the same depth. Implementation is mostly analogous.

```
<BuildingBlock> ::= <Plus>
                  | <Minus>
                  | <Multiply>
                  | <SafeDiv>
                  | <Variable>
<Plus>          ::= <Number> + <Number>
<Minus>         ::= <Number> - <Number>
<Multiply>      ::= <Number> * <Number>
<SafeDiv>       ::= <Number> / <Number>
<Variable>      ::= x1 .. xn # features of the dataset
```

Listing 6.1: Grammar interpretation of Tradtional GP.

```
def fitness_function(fs: Solution, data, target):
        transformed_data = mapping(data, fs)
        dt = DecisionTreeRegressor(max_depth=4)
        scores = cross_validate(dt, transformed_data, target)
        return np.mean(scores)
```

Listing 6.2: The wrapper fitness function of a DT regressor with depth 4.

In listing 6.2, each candidate solution is a mapping between the original feature space and a new space. Using the solution the original feature set is mapped into a new feature space. With the transformed data the DT regressor/classifier is trained in a cross-validation process using the transformed data and the target data. The cross-validation process uses a metric that differs for the DT regressor and classifier. For the regressor I use the *mean squared error* (MSE), and for the classifier I use the weighted F_1 *score*. Both are introduced in section 7.2.6. Cross validation for time series is not straight-forward, and is discussed in section 7.2.5.

In M3GP the GP algorithm evolves a mapping from the original feature set to a *multidimensional* feature set. See listing 6.3 for the grammar definition of this multidimensional feature set. A *FeatureSet* consists of a list of *BuildingBlock* nodes, with a length between 1 and 15. The length of 15 is chosen as a DT regressor with depth 4 has a maximum of $\sum_{i=0}^{3} 2^i = 15$ choices to make. Each *BuildingBlock* node represents a feature.

In their implementation, Batista et al. [9] defined special mutation and crossover operators. When mutating an individual, there is a $\frac{1}{3}$ probability that the algorithm removes an element from the feature set, and there is a $\frac{1}{3}$ probability that it adds a random element to the feature set. If neither of those possibilities occur, it mutates one feature of the feature set. Similarly, for crossover there are two possibilities. One randomly cuts the feature sets of two individuals in two parts and combines one splice of each individual. Otherwise, two features of different individuals undergo crossover.

I implemented the above evolutionary operators in Genetic Engine using the *ListSizeBetween* Meta-Handler. This only created the possibility for the evolutionary operator to happen. To align with Batista

```
@abstract
class Solution():
    pass

@dataclass
class FeatureSet(Solution):
    subset: Annotated[List[BuildingBlock], ListSizeBetween(1,15)]

    def evaluate(self):
        return [ el.evaluate() for el in self.subset ]

@abstract
class BuildingBlock():
    pass
```

Listing 6.3: Grammar definition of a *FeatureSet* node. A *FeatureSet* node contains a list of *BuildingBlock* nodes, which represent learned features.

et al. [9], I require the *FeatureSet* nodes to be operated upon more often. Using the *specific_type_mutation* and *specific_type_crossover* variables introduced in table 5.1, I give preference to the *FeatureSet* nodes to be operated on, resulting in at least half of the mutation and crossovers done on *FeatureSet* nodes. This deviates slightly from the implementation of Batista et al. [9]. I compare my M3GP implementation with the implementation of Batista et al. [9] in chapter 7.

Originally, M3GP was proposed as a classification method, with its fitness function being a wrapper function that involves clustering. Batista and Silva [8] showed that simpler and quicker wrapper functions like a DT regressor/classifier do not perform worse than the clustering method. As such, I chose the same fitness function as used for Traditional GP, found in listing 6.2.

6.2 Implementation of Domain Knowledge

Starting of from the M3GP method introduced above, I enhance my model by introducing domain knowledge. I did this by recognizing the intrinsic meaning of certain features of both the BoomBikes dataset, as the credit-g dataset. As a running example, I will use the feature *season* from the BoomBikes dataset. I name this method M3GP with Domain Knowledge (DK-M3GP). In this section, I discuss the implementation of DK-M3GP and prospected improvement of both prediction performance and interpretability.

Originally, the node *Season* can take on the values 1 to 4, referring to the seasons Winter to Fall, respectively. Without telling my models what these values mean, the values are considered integers, and treated as such. Because of this, it is possible to do integer operations on these values. For example, we can sum the season of a data point with its temperature, which is given as a float. To Python, summing an integer with a float makes sense, so this is a legitimate operation. However, to a human being, the

```python
@abstract
class Category:
    category: int
    column: Col

@abstract
class Col:
    col_name: str

    def evaluate(self):
        return DATASET[self.col_name]

@dataclass
class SeasonCol(Col):
    col_name = "season"

@dataclass
class Season(Category):
    category: Annotated[int, IntRange( 1, 4 )]
    column: SeasonCol
```

Listing 6.4: *Season* category-column combination defined in the grammar. Notice how a *Category* node consists of both a category value (in this case an integer between 1 and 4), and a column (when evaluated this returns the values of the season column).

summation of the season and the temperature does not make sense at all; e.g., what does $Summer + 25°C$ mean?. As such, the evolutionary process can result in feature spaces that are built up from nonsensical features. If we want to be able to easily interpret learned features, a basic requirement is for them to make sense. Furthermore, we hypothesize that nonsensical features represent the data worse, as they do not represent real relationships. Therefore, restricting the solution space by making the search through these unreal relations impossible, should improve prediction performance.

Essentially, the issue is that the season shouldn't be seen as an integer, but as a category. Being a category, I must operate on seasons distinctly in comparison to features of types integer and float. This operation should have a season category as input, and output a *BuildingBlock* node. To solve this, I introduce conditions that can be formed from categories, for example an equal condition: DATASET[season] = Winter. The resulting column of booleans can then be used in *control flows* such as an *if* statement, which I will discuss later in listing 6.6. This requires combining a possible value of season and the season column from the dataset. See listing 6.4 for the implementation of the *Season* category-column combination, and see listing 6.5 for the implementation of the *Equals* condition.

Season is not the only feature that shouldn't be interpreted as an integer. For the BoomBikes dataset, the other features are *month*, *weekday*, *year*, *holiday* and *workingday*, of which the last two are of type boolean. For the credit-g dataset, the features are *checking_status*, *credit_history*, *purpose*, *sav-*

```python
@abstract
class Condition:
    pass

@dataclass
class Equals(Condition):
    input: Category

    def evaluate(self):
        return np.apply_along_axis(
            lambda x: x == self.input.category,
            0,
            self.input.column.evaluate()
            )
```

Listing 6.5: *Equals* condition defined in the grammar. The numpy (np) function applies the equals statement along the season column. Evaluating a node of type *Category* therefore returns a column with boolean values stating whether or not the column value in that place equals the specified category value.

ings_status, *employment*, *personal_status*, *other_parties*, *property_magnitude*, *other_payment_plans*, *housing*, *job*, *own_telephone* and *foreign_worker*, of which the last two are of type boolean. These were all implemented as *Category* nodes.

Except for the *Equals* condition, I implemented *NotEquals* similarly. Furthermore, I implemented the *InBetween* condition. This evaluates categories to be between two values, requiring the implementation of *Category* nodes with two *category* values, implemented in the *IBCategory* node. Note that this is only possible for categories with an explicit order, so, in this case, the features of type boolean (*holiday* and *workingday* for the BoomBikes dataset, and *own_telephone* and *foreign_worker* for the credit-g dataset) are not considered. Implementation is straight-forward, and thus not discussed.

Conditions can be used in conditional statements, like an *if* statement. An if statement takes in three variables; a condition; and two possible outcomes, one for when the condition is *True*, and the other for when the condition is *False*. See the *IfThenElse* implementation that extends a *BuildingBlock* node in listing 6.6.

I expect the improvement of domain knowledge incorporation in DK-M3GP to be two-fold. First, I expect performance to improve, as nonsensical relations are disregarded, so that the search space is restricted to only consider sensible relations. Secondly, I expect the learned features to be more easily interpreted, as they are learned without considering nonsensical relations. As all building blocks considered make sense to human beings, the final learned features hopefully also make sense.

```python
@dataclass
class IfThenElse(BuildingBlock):
    cond: Condition
    then: BuildingBlock
    elze: BuildingBlock

    def if_statement(self,x):
        if x[0]:
            y = x[1]
        else:
            y = x[2]
        return y

    def evaluate(self):
        y = np.apply_along_axis(
                self.if_statement,
                0,
                np.array([
                    self.cond.evaluate(),
                    self.then.evaluate(),
                    self.elze.evaluate()
                    ])
                )
        return y
```

Listing 6.6: If statement defined in the grammar. The numpy (np) function applies the *if_statement* along the columns of the *Condition* node and the two possible outcomes, *then* and *elze*, both *BuildingBlock*. Evaluating a node of type *IfThenElse* therefore returns a column with *BuildingBlock* evaluations of either the *then* or the *elze* node, depending on the result of the *Condition* node.

6.3 Aggregation

Developing further on the implementation of domain knowledge, I added the possibility of aggregating historical data. As the BoomBikes datasets is a time series, where every data point records information of one day, I might suppose there is a relation between future data and historical data. Likewise, the credit-g dataset is a panel, with data points representing different individuals, and thus newly gathered data might resemble information on previously gathered data. For example, the target of the BoomBikes dataset, the number of bicycles (*cnt*) of each day, might resemble the average number of bicycles of previous days. Having constructed categories in section 6.2, I can even add a restriction to the average based on a category. For instance, for the BoomBikes dataset it might be interesting to look at the average of the same day in previous years, the same weekdays in previous weeks, or only average over days in the same season. For the credit-g dataset, we might want to consider the average risk assessment of individuals with the same job. I name this method DK-M3GP with Aggregation (DKA-M3GP).

```python
@dataclass
class Average(BuildingBlock):
    col: Col
    aggregation_col: Annotated[str, VarRange(["target"])]
    window_length: Annotated[int, IntList([10, 25, 50, 75, 100, 150, 200, 300, 400, 600, 800])]

    def assign_target_values_to_data(self, data, historical_data):
        ...
        return combined_data

    def aggregate(self, combined_data, instants, window_length):
        ...
        return aggregated_vals

    def evaluate(self):
        historical_data = DATASET['historic']
        historical_data = historical_data[[TIME_COL, self.col.col_name, self.aggregation_col]]
        instants = kwargs[TIME_COL]
        data = pd.DataFrame({TIME_COL:instants, self.col.col_name: self.col.evaluate()})

        combined_data = self.assign_target_values_to_data(data, historical_data)
        aggregates = self.aggregate(combined_data, instants, self.window_length)
        aggregates = np.nan_to_num(aggregates)
        return aggregates
```

Listing 6.7: Aggregation implementation of the *Average* node. The implementations of the functions *assign_target_values_to_data* and *aggregate* are shown in listing 6.8 and listing 6.9, respectively.

The implementation of the *Average* node is depicted in listing 6.7, and is not straight-forward. The node is initialized by three variables; a category column (*col*), to restrict the aggregation so that the aggregation is done over data points with that same category value; an aggregation column (*aggregation_col*), the values to be aggregated; and a window length (*window_length*), the number of historic data points to be considered in the aggregation. Evaluation of the *Average* node is split up into three parts; the *assign_target_values_to_data* function in listing 6.8; the *aggregate* function in listing 6.9; and the *evaluation* function itself in listing 6.7.

The *evaluate* function starts by collecting all necessary input. The *historical_data* is taken from the DATASET and then filter on the columns that matter, the *TIME_COLUMN*, the category *col*, and the *aggregation_col*. Then the current data (*data*) is obtained from the DATASET. Subsequently, I use the helper functions *assign_target_values_to_data* and *aggregate* to assign target values to the current data and aggregate all available historic data for each data point, respectively. After filling the *NaN* values of the *aggregates*, they are returned as the aggregated column. Most work is done in the helper functions, which I discuss below.

To aggregate, I must first construct a dataframe that combines historic data with input data. In list-

```python
def assign_target_values_to_data(self, data, historical_data):
    data = data.merge(historical_data,how='left')
    first_data_instant = data[TIME_COL].min()
    historical_data = historical_data[historical_data[TIME_COL] < first_data_instant]

    means = historical_data.groupby(self.col.col_name).mean()[self.aggregation_col]
    uns = data[self.col.col_name].unique()
    def missing_mean_zero(un):
        try:
            return means[un]
        except:
            return 0
    mapping = dict([ (un,missing_mean_zero(un)) for un in uns ])
    def fill_nans(row):
        if np.isnan(row[self.aggregation_col]):
            return mapping[row[self.col.col_name]]
        else:
            return row[self.aggregation_col]
    data[self.aggregation_col] = data.apply(lambda row: fill_nans(row), axis=1)

    combined_data = pd.concat([historical_data,data])
    return combined_data
```

Listing 6.8: The *assign_target_values_to_data* function in the *Average* node.

ing 6.8 this is implemented in the *assign_target_values_to_data* function. First, all current data is assigned target values from the historic data, if a target value is available. This is done by merging them, and following this, the historic dataset is trimmed from data points that occurred after the current data started.

Now a mapping is constructed for all current data that is not represented in the historical target values, based on the category value of each data point. For example, if the category is *season*, the target value of each unrepresented data point is filled with the average of the historic data from the same season. To start, I find the means for each value of the category. If no mean is found, I fill it with zero. Consecutively, the *fill_nans* function is built. Given a row, this function checks whether its target value is *NaN*, and if so, it returns the target value from the mapping based on its category value. After applying the *fill_nans* function to the current data I concatenate the historic data with the current data. Remember that I trimmed the historic data to not contain data from after that start of the current data, so there will not be any historic data from after the start of the input data included.

Listing 6.9 depicts the *aggregate* function, which is slightly more straight-forward, but has many small implementation details. The centrepiece to finding the historic mean is the *rolling* function of *pandas* [81]. I first reindex the data with the time instant, and sort it. Then, I group the data by the category column, as to only aggregate data based on its category value. Following, I reset the index again

```python
def aggregate(self, combined_data, instants, window_length):
    aggregated_vals = combined_data.set_index(TIME_COL).sort_index().groupby(self.col.col_name).
        ↪ rolling(window=window_length, min_periods=1, closed='left').mean()
    aggregated_vals = aggregated_vals.reset_index().set_index(TIME_COL).sort_index()
    aggregated_vals = aggregated_vals[self.aggregation_col].filter(instants)

    return aggregated_vals
```

Listing 6.9: The *aggregate* function in the *Average* node.

and sort the data. The data is now in the correct format, sorted, and with aggregation values. Remember that the data contains historic data to get correct aggregations, even though it is now redundant. So the final step consists of throwing the historic data out.

The addition of aggregation in DKA-M3GP enlarges the solution space, allowing for more solutions to be found. The main improvement that I expect to see in DKA-M3GP is a performance improvement, due to the enlarged solution space. Secondly, aggregation can unravel complex relationships from data, possibly contributing to domain knowledge. Lastly, these complex relationships might resemble human-engineered features more closely, enhancing interpretability.

Chapter 7

Evaluation

In this chapter, I evaluate the proposed methods. I compare my methods with a number of baselines, introduced in section 7.1. Then, I introduce the actual experiment details in section 7.2. After that, I show the results of the comparison between my M3GP implementation using Genetic Engine and the implementation as done by Batista et al. [9] in section 7.3. Finally, in section 7.4 I compare all the other FL methods.

7.1 Baselines

To evaluate my methods, I compare them to a number of baselines, most of which I introduced in chapter 3, and the last two I detailed in section 6.1:

No FL To be sure that FL had any effect at all, I compared to no FL, passing the input feature set directly.

PCA PCA implementation was directly taken from the decomposition library of Scikit-learn [82].

FS through FeatureTools I used the *remove_highly_correlated_features* method from FeatureTools [51] which, as the name suggests, removes highly correlated features.

Random Search FS Using Random Search (RS) as introduced in section 2.1.1, I implemented an FS method. For the implementation, I used the RS algorithm from Genetic Engine. The grammar consisted of *Var*, *FeatureSet*, *BuildingBlock*, and starting symbol *Solution*, all introduced above in section 6.1.

M3GP as implemented by Batista et al. [9] M3GP-JB was directly imported from the library provided by Batista et al. [9]. A slight adaptation had to be made for the method to allow for regression. The fitness function was adapted in line with the fitness function based on a DT, introduced in section 6.1.

M3GP as implemented by Batista et al. [9] is only used compared to my implementation of M3GP using Genetic Engine, to evaluate the performance of Genetic Engine. This is done in section 7.3. The other four FL methods together with Traditional GP, M3GP, DK-M3GP and DKA-M3GP are separately compared in section 7.4. Below, I introduce the experiment details.

7.2 Experiment details

I evaluate the FL methods mainly in two ways. The search-based algorithms are evaluated on their training fitness progression and test fitness progression, introduced in section 7.2.1. To compare the all FL methods, I evaluate them within one of 4 full ML pipeline, described in section 7.2.2. I then introduce the settings of the experiment and of the FL methods in section 7.2.3. To make sure performance is less dependent on specific model settings, I first pass each evaluation through a grid search, as elaborated upon in section 7.2.4. After that, in section 7.2.5 I discuss my implementation of cross-validation for time series, paramount in good evaluation techniques. As evaluation is not straight-forward, I finally define the evaluation techniques in section 7.2.6.

7.2.1 Search-based algorithm evaluation

For the search-based FL methods Traditional GP, M3GP (both implementations), DK-M3GP and DKA-M3GP I will separately compare the methods. I will compare the evolutionary process in each generation of these methods on (1) the training fitness of the best individual, (2) the test fitness of the best individual, and (3) the complexity of the best individual. Finally, I compare the methods on the time spend.

7.2.2 Pipeline

To evaluate each FL method, I contain them within full ML pipelines. To evaluate diversely, I compare the following four models also researched by Ain et al. [3]: a Decision Tree (DT), Random Forest (RF), a Multilayer Perceptron (MLP), and a Support Vector Machine (SVM). The other two models researched by Ain et al. [3] (Naive-Bayes and k-Nearest Neighbour) are not suitable for regression, and are therefore left out. Each model is instantiated using the experiment seed (see section 7.2.6 for details).

7.2.3 Experiment settings

The comparison is done using two different comparison methods. First, we compare the FL methods without restricting the resources of the methods. Secondly, we compare them with a restriction on the resources, namely allowing an hour for the algorithm to run, excluding the grid search. We call the comparison methods *free comparison* and *budgeted comparison* respectively.

Except for the algorithm parameters included in the grid search that are introduced in section 7.2.4, the FL methods rely on other parameters. See table 7.1 for an overview of all these parameters.

Table 7.1: Parameters of FL methods, excluding grid search parameters introduced in section 7.2.4.

Parameter	**Applicable to**	**Value**
Individual representation	All search-based methods	Treebased[1]
Max feature depth	All search-based methods	10
Population size	All search-based methods	200[2]
Number of generations	All search-based methods	500 or 200
Crossover probability	All search-based methods except RS FS	0.5[1]
Mutation probability	All search-based methods except RS FS	if no crossover[1]
Number of novelties	All search-based methods except RS FS	0[1]
Tournament size	All search-based methods except RS FS	5[1]
Favour less deep trees	All search-based methods except M3GP-JB	True

7.2.4 Grid search

To make sure performance is less dependent on specific model settings, each evaluation undergoes a very simple grid search. The goal of a grid search is exhaustively searching through all user-defined model parameters, and selecting the best performing ones. As our search-based methods are time-consuming, we only run the grid search over a part of the generations, in this case 25. Below, I outline the parameters I selected for each method.

Originally, I included many parameters in the grid search for the search-based FL methods, but in order to align as much with the M3GP implementation of Batista et al. [9], I had to exclude almost all. I ended up with only including the elitism size in the grid search, including 1 and 5 as possible sizes. The RS FS method doesn't use any of the above parameters, so no grid search is performed. PCA has a single parameter, namely the number of components constructed by the algorithm. The grid search analyses 1, 2, 3, 4 and 5 components. Finally, FS using FeatureTools has a correlation threshold which I include in the grid search for the parameters 0.6, 0.7, 0.8, 0.9 and 0.95.

7.2.5 Cross validation for time series

The grid search is done in a cross-validation-for-time-series manner, similar to the method proposed by Hyndman and Athanasopoulos [44] in Section 5.10. The training data is split up in parts, specifically at

[1] Aligned with implementation of Batista et al. [9].

[2] For testing with the BoomBikes dataset, I used a population size of 500. For testing with the credit-g dataset, I used a population size of 200.

$\frac{3}{6}$, $\frac{4}{6}$ and $\frac{5}{6}$ of the data. The first $\frac{3}{6}$ part is used to predict the next $\frac{1}{6}$, the first $\frac{4}{6}$ is used for the following $\frac{1}{6}$, and the first $\frac{5}{6}$ is used for the following $\frac{1}{6}$ (see fig. 7.1 for a visualisation).

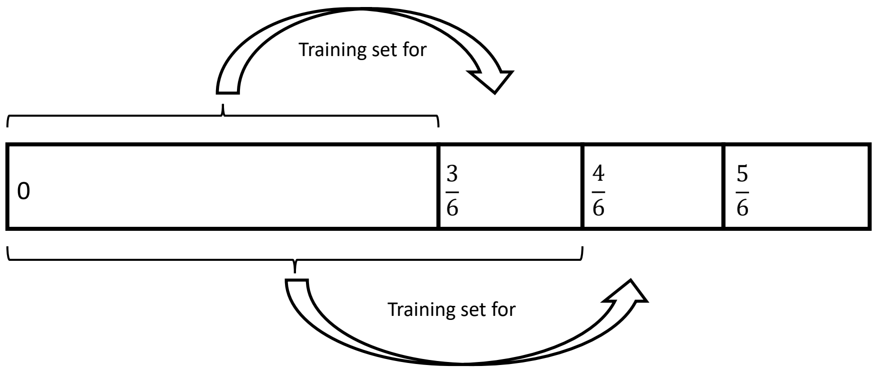


Figure 7.1: Here you see the training sets that are used for each test set. The last training set is not shown.

I run the grid search using cross-validation-for-time-series on 75 % of the data, the training set. After that, I train the model on the whole training set, and test it on the final 25 %. In this way, the grid search parameters are independent of the test data.

7.2.6 Evaluation

I evaluate my methods using the performance metrics MSE for evaluation of regression solutions, and F_1 score for the evaluation of classification solutions, as introduced in section 2.1.3. To show statistical significance when comparing methods, I use a statistical significance test, elaborated upon in section 7.2.6.1.

I primarily evaluate the interpretability of my methods using the ES, also introduced in section 2.1.3. Furthermore, for I discuss interpretability in a broader sense as suggested by Liao et al. [62].

7.2.6.1 Statistical significance

To show statistical significance, each FL method is trained for each model for 30 different seeds. For the statistical significance tests I use the Mann-Whitney-Wilcoxon test [109]. To do so, two experiments must be independent of each other, which they clearly are, as both consider different FL methods. The Mann-Whitney-Wilcoxon U statistic is defined as follows.

Definition 3. *Given two samples* $X = \{x_1, \ldots, x_n\}$ *and* $Y = \{y_1, \ldots, y_m\}$*, we calculate the* U_X *statistic as follows:*

$$U_X = \sum_{i=1}^{n}\sum_{j=1}^{m} S(x_i, y_j),$$

where

$$S(x_i, y_j) = \begin{cases} 0 & \textit{if}\, x_i < y_j \\ \frac{1}{2} & \textit{if}\, x_i = y_j \\ 1 & \textit{if}\, x_i > y_j. \end{cases}$$

The U_Y statistic is calculated in an analogous way.

Given a null hypothesis H_0 stating that X and Y come from the same distribution, we expect the two samples to have a similar median. As such,

$$H_0 : P(x < y) = \frac{1}{2}, \forall x \in X, y \in Y, \text{and}$$
$$H_1 : P(x < y) \neq \frac{1}{2}, \forall x \in X, y \in Y.$$

Now, the p-value is

$$p\text{-value} = P(U|H_0).$$

I reject the null hypothesis if the p-value is less than 0.05.

I visualize the p-values directly in the plots. For visualization, I make use of the *statannotations* [18] library. Whenever it is relevant, the p-values appear at the top of the plots.

7.3 Comparison of M3GP implementations.

To see whether our implementation of M3GP in Genetic Engine functions well, I compared it with the implementation presented by Batista et al. [9]. I compared the performance of the M3GP implementations in section 7.3.1. I separate the performance comparison for the two datasets. Finally, I briefly compare run times in section 7.3.2.

7.3.1 Performance comparison

7.3.1.1 The BoomBikes dataset

For both datasets, I looked at performance on training data and at performance on testing data. See fig. 7.2 and fig. 7.3 for the fitness per generation comparisons between the two M3GP implementations for the BoomBikes dataset. The max tree depth for the features of each individual is set to 10.

In fig. 7.2, the Genetic Engine implementation has a better fitness at initialization. Note that the fitness is based on the MSE, and the lower MSE, the better. As such, we are aiming to minimize the fitness. After only a few generations, the Batista et al. [9] implementation passes the Genetic Engine

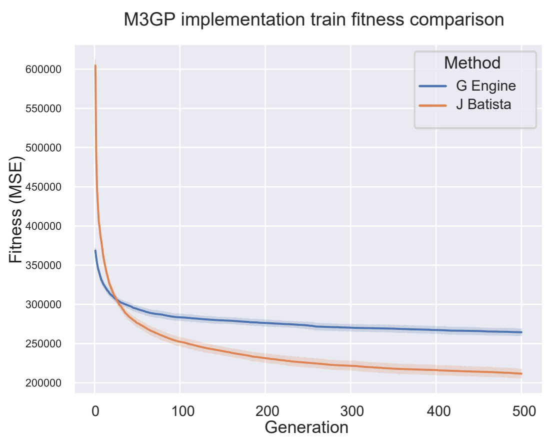


Figure 7.2: BoomBikes training fitness comparison.

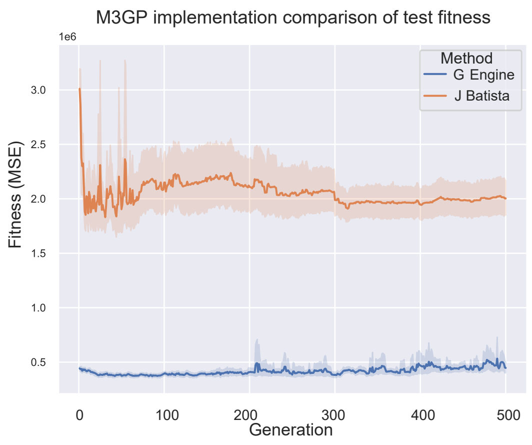


Figure 7.3: BoomBikes testing fitness comparison.

implementation and both stabilize. As such, the Batista et al. [9] implementation has a better training fitness, and thus shows better training performance. It is expected that both implementations give similar fitness progressions, as they implement the same algorithm. I found two reasons for this discrepancy.

First, as Genetic Engine is still immature, I suspect that our node generation method is worse. Further inspection shows that in the first generation there are multiple duplicate individuals. This is not expected of a purely random node generation, as there are millions of different individuals possible. Having multiple duplicate individuals in a population of 500 individuals is therefore highly unlikely. As part of future work, I will improve the diversity of the node generation method.

Secondly, Genetic Engine creates much more complex individuals at generation (see fig. 7.4). As deeper trees suffer from *failed disruption propagation*, i.e., evolutionary operations having no effect, because of more redundant information [59], it is likely that the Genetic Engine implementation suffers from this, reducing its performance.

On the other hand, looking at fig. 7.3 we see that, very soon after the start of the evolution, the training fitness improvement is not translated to an improvement in the test fitness. In other words, both implementations overfit the training data. Overall, the Genetic Engine implementation shows a better testing performance, and it thus generalizes better.

I also looked at both implementations within the ML pipelines. See fig. 7.5 for the comparison within ML pipelines with models DT, RF, MLP and SVM.

The comparison shows that for two models (RF and SVM) the Batista et al. [9] implementation performs significantly better than the Genetic Engine implementation. On the other hand, the Genetic Engine implementation performs better for MLP.

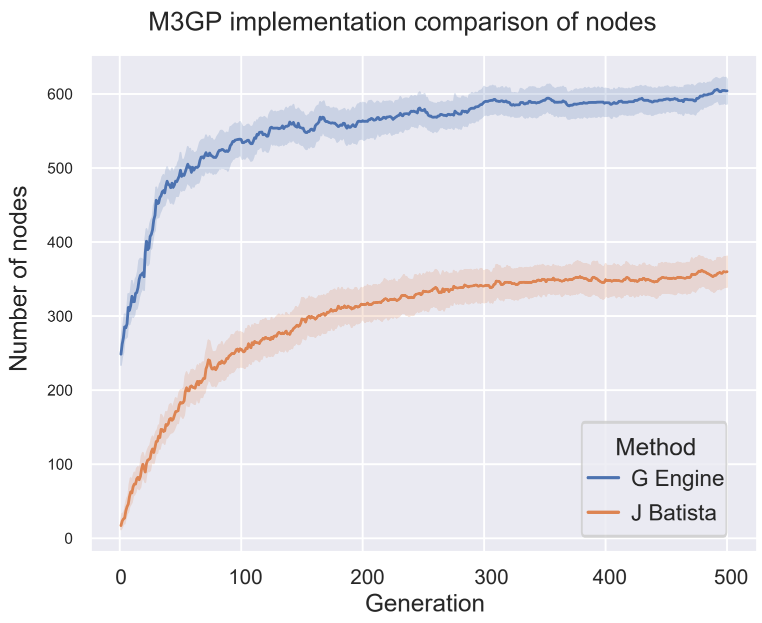


Figure 7.4: Comparing the complexity of solutions generated by each implementation for the BoomBikes dataset. Complexity is measured by the number of nodes in the best individuals of both implementations per each generation (as introduced in section 7.2.6).

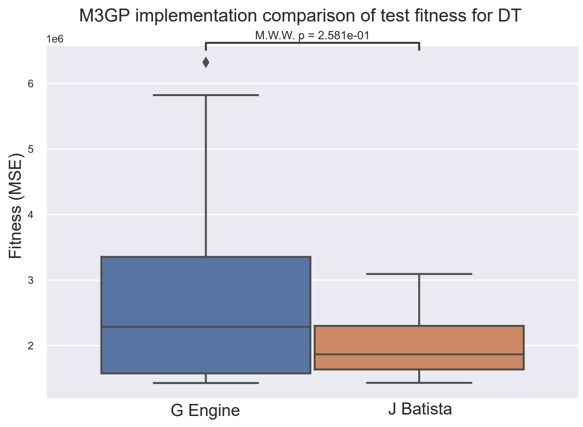
7.3.1.2 The credit-g dataset

I also compared the M3GP implementation for classifying the credit-g dataset. As discussed in section 7.2.6, I used the F_1 score with the weighted method in the fitness function in line with the implementation of Batista and Silva [8]. As you can see in fig. 7.6, the Batista et al. [9] implementation does not improve during the evolution. Further inspection shows that the best individual classifies everything as the positive class, obtaining an acceptable fitness (± 0.825), and does not improve on that. Obviously, as the Genetic Engine implementation does not suffer from this, my implementation performs better in this case.

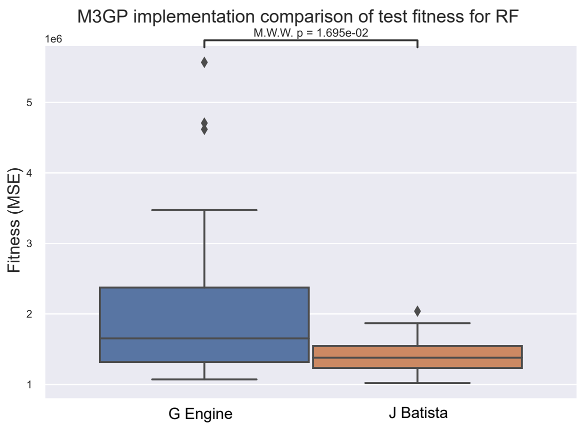
As there are other methods, I decided to also compare the implementations using the binary F_1 score. In figs. 7.7 and 7.8 the train and test fitness using the binary F_1 score are depicted respectively.

For credit-g, the Genetic Engine implementation outperforms the Batista et al. [9] implementation in both cases, even though the Genetic Engine implementation is much more complex (see fig. 7.9). Like for the BoomBikes dataset, both implementations overfit the train data, as the training fitness improvement is not translated to an improvement in the test fitness.

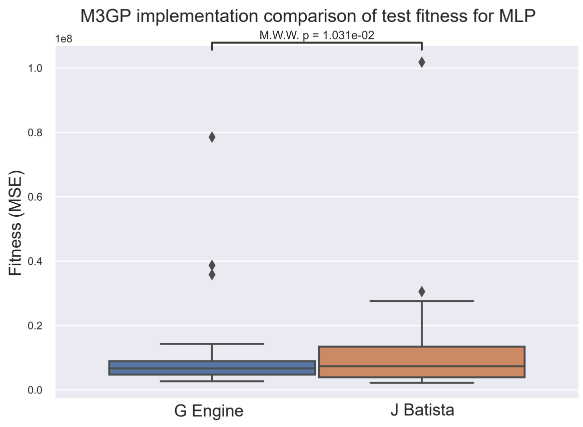
Now, let's analyse both implementations within the ML pipelines. See fig. 7.10 for the comparison within ML pipelines with models DT, RF, MLP and SVM. Both implementations do not significantly perform differently except for MLP, for which the Genetic Engine implementation significantly performs



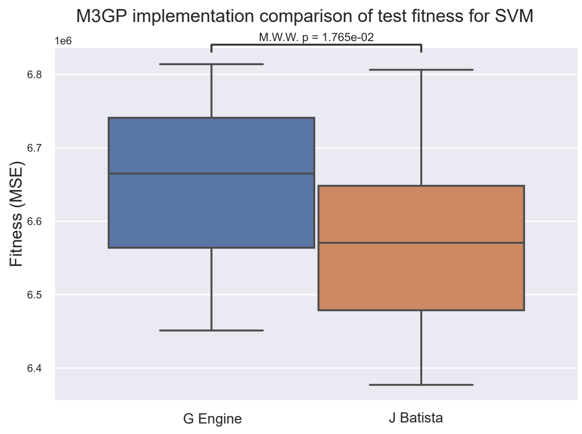
(a) ML pipeline with DT



(b) ML pipeline with RF



(c) ML pipeline with MLP



(d) ML pipeline with SVM

Figure 7.5: MSE comparison of the two M3GP implementations within ML pipelines for the BoomBikes dataset. The ML pipelines differ only on the models. In the figures, outliers were taken out for aesthetical reasons. At the top of each figure, statistical significance scores are portrayed. I included the outliers for the calculation of the statistical significance score. RF, and SVM show significant better performance for the Batista et al. [9] implementation, whereas MLP performs significantly better for the Genetic Engine implementation.

better.

Overall, my implementation using Genetic Engine performs quite well in comparison with the M3GP implementation implemented by Batista et al. [9]. However, there are many improvements possible for Genetic Engine. Amongst others, the node generation should be randomized further to improve diversity, and (ϵ-)lexicase selection [40, 57] can easily boost performance for both use cases above. I discuss this further in section 8.3.

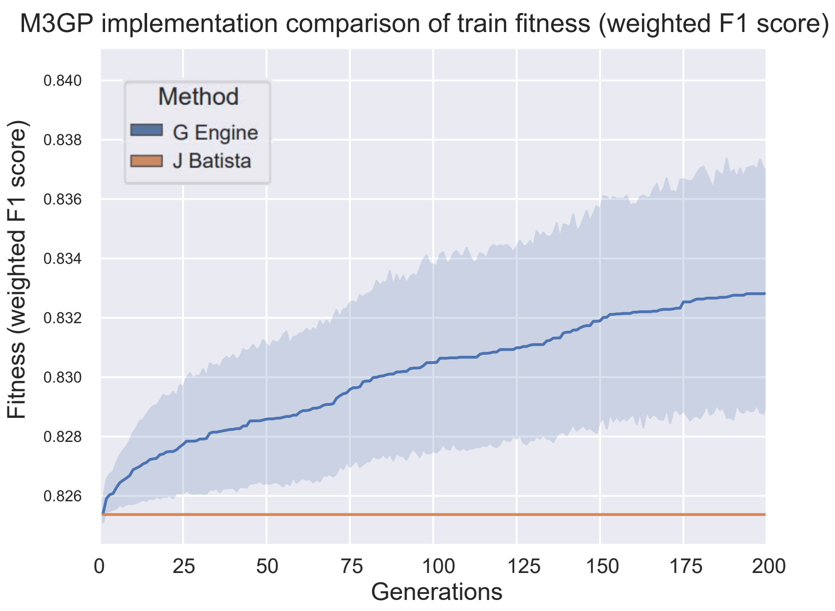


Figure 7.6: Comparison of the fitness using the weighted F_1 score.

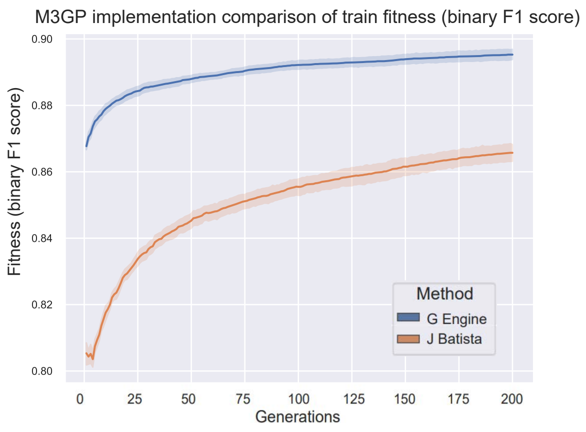


Figure 7.7: Credit-g training fitness comparison.

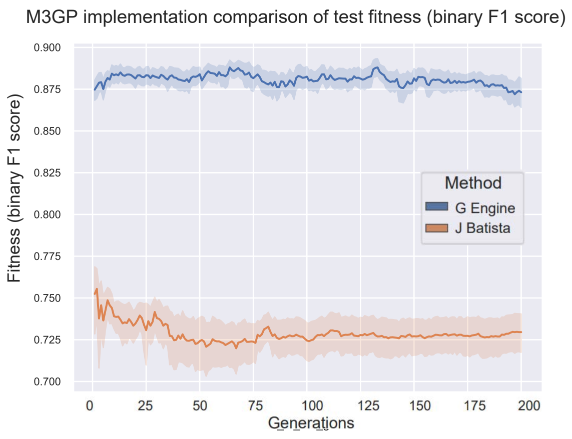


Figure 7.8: Credit-g testing fitness comparison.

7.3.2 Time comparison

Lastly, I compared the two implementations on the execution time. In fig. 7.11 the execution time is compared (excluding the time spend on the grid search) for both datasets. For each model and dataset, Genetic Engine is much faster and less variant.

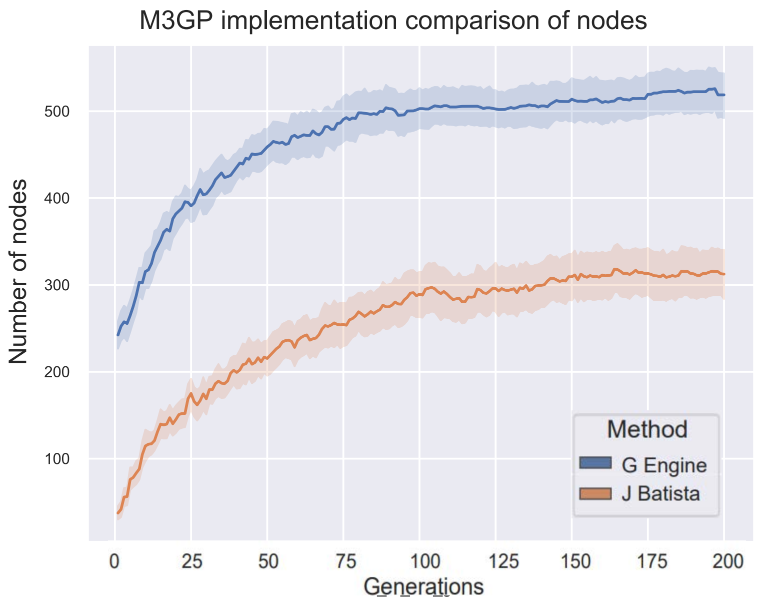
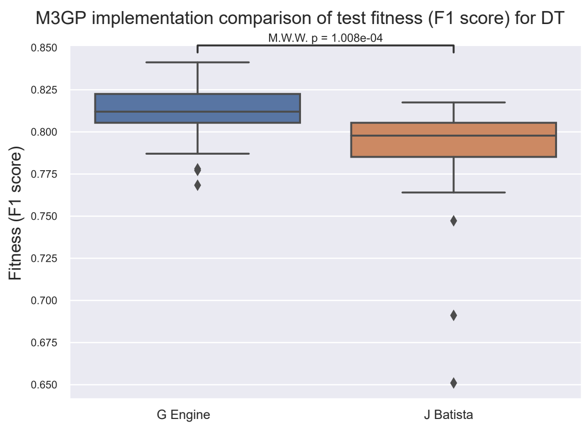
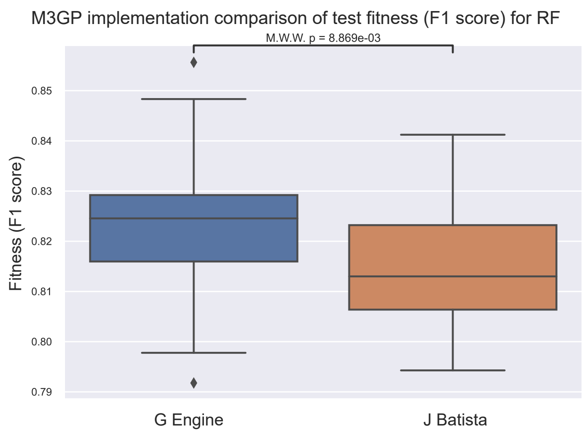


Figure 7.9: Comparing the complexity of solutions generated by each implementation for the credit-g dataset. Complexity is measured by the number of nodes in the best individuals of both implementations per each generation (as introduced in section 7.2.6).

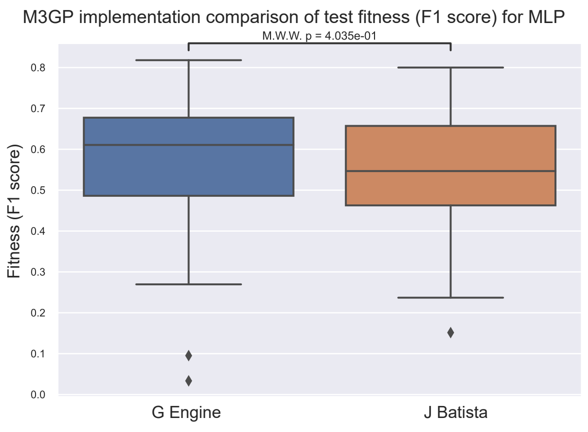
It would be interesting to compare both implementations with a budget. As the implementation of Batista et al. [9] does not include running with a time limit, this was out of scope for this work.



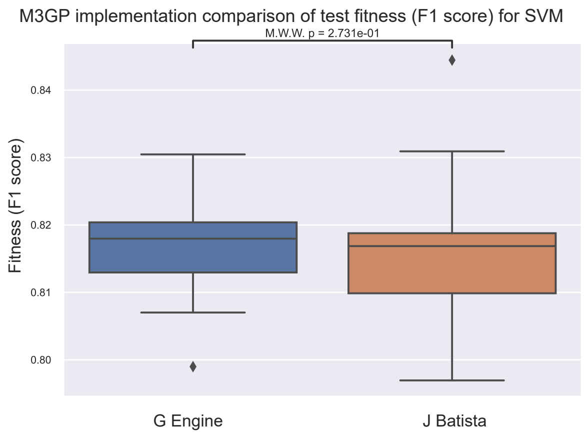
(a) ML pipeline with DT



(b) ML pipeline with RF

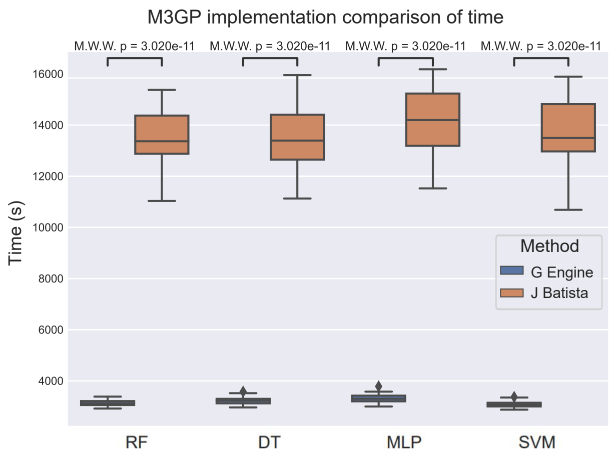


(c) ML pipeline with MLP

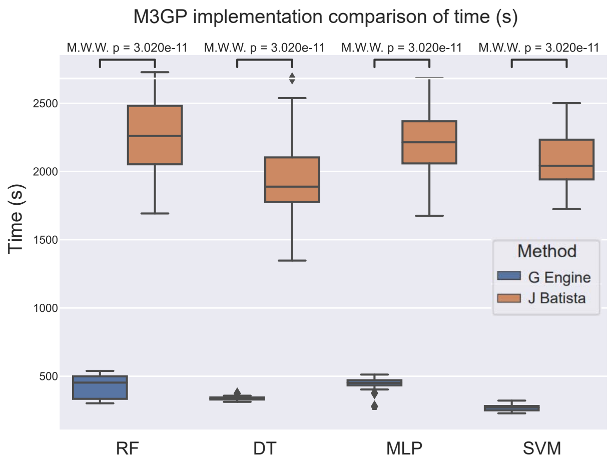


(d) ML pipeline with SVM

Figure 7.10: Binary F_1 score comparison of the two M3GP implementations within ML pipelines for the credit-g dataset. The ML pipelines differ only on the models. In the figures, outliers were taken out for aesthetical reasons. At the top of each figure, statistical significance scores are portrayed. I included the outliers for the calculation of the statistical significance score. RF, and SVM show significant better performance for the Batista et al. [9] implementation, whereas MLP performs significantly better for the Genetic Engine implementation.



(a) BoomBikes dataset



(b) Credit-g dataset

Figure 7.11: Time comparison of the Batista et al. [9] implementation and the Genetic Engine implementation for both datasets. the Genetic Engine implementation is significantly faster for both datasets across models. Grid search time is excluded.

7.4 Comparison of FL methods

In this section, I compare my novel FL approaches to the different baselines introduced in section 7.1. I start by analysing the results of the BoomBikes dataset in section 7.4.1. After that, I analyse the results of the credit-g dataset in section 7.4.2. Finally, in section 7.4.3, I compare the expressions of some solutions for both datasets.

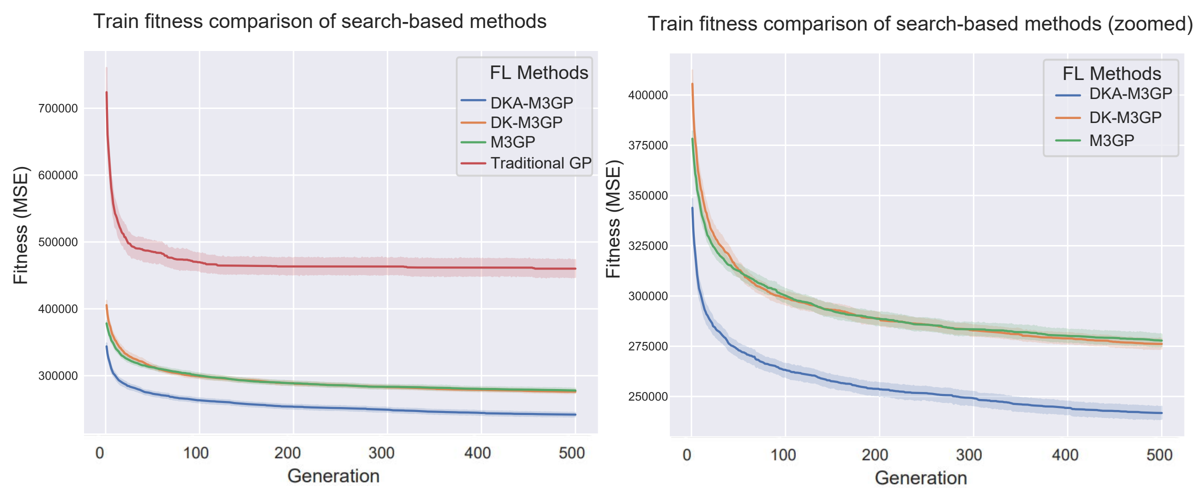
7.4.1 The BoomBikes dataset

First, I compare against the search-based methods (excluding RS FS) in section 7.4.1.1. In section 7.4.1.2, I include the classical methods. Finally, I compare the FL methods on time in section 7.4.1.3.

7.4.1.1 Comparison of search-based Feature Learning methods

In fig. 7.12 I compare the training fitness of the different search-based methods. Traditional GP performs significantly worse than the other methods. When zooming in on M3GP, DK-M3GP and DKA-M3GP. M3GP and DK-M3GP have a comparable fitness progression, reaching a similar final fitness. DKA-M3GP, on the other hand, has a much better fitness, indicating that adding aggregation to the grammar improves the solution space.

Figure 7.13 depicts the test fitness for the different methods. As for M3GP, Traditional GP, DK-M3GP and DKA-M3GP overfit the training data. The methods perform similarly on the test data, except for Traditional GP, which performs worse than the other methods. Overfitting could be caused by the



(a) Complete comparison (b) Zoomed in comparison

Figure 7.12: Comparison of Traditional GP, M3GP, DK-M3GP and DKA-M3GP.

methods not being suitable for the task at hand. Another reason for the methods to consistently overfit the training data could be that the training data does not represent the test data well. Looking back at fig. 4.1, we see that at roughly the last 25% (the size of the test data) the data makes a big sudden drop. Such a big drop had not occurred before in the data. As such, the test data seems not to be represented well by the training data.

Subsequently, I compared the methods within the different ML pipelines. See fig. 7.14 for the results. Across all models, DK-M3GP does not perform well within the pipeline, significantly worse than M3GP, even though training fitness is similar for DK-M3GP and M3GP. From this, I derive that DK-M3GP overfits the DT regressor of the fitness function and its particular training seed. Traditional GP performs well for all models except for MLP. That Traditional GP does not perform well on MLPs is expected, as Traditional GP generates a single feature and MLPs excel when combining features. Except for MLP and RF, Traditional GP significantly performs best. Apparently the learned feature generalizes the data best. DKA-M3GP performs quite similar to M3GP. For SVM it performs significantly better than M3GP, but for MLP significantly worse.

As for the M3GP implementation comparison, good training fitness does not translate to good performance within ML pipelines. Even more so, the worse performing model, Traditional GP, performs best within ML pipelines. In search for answers, I compared the complexity of the models in fig. 7.15. Most significantly, there is a negative correlation between complexity and performance on the test set. Traditional GP is the least complex and performs best. DK-M3GP is the most complex and performs worse. M3GP and DKA-M3GP have similar complexity progressions and perform comparably.

Hypothetically, this negative correlation can be explained by the training task not representing the testing task correctly. While a solution is less complex, it does not specialize on the specific task. Methods

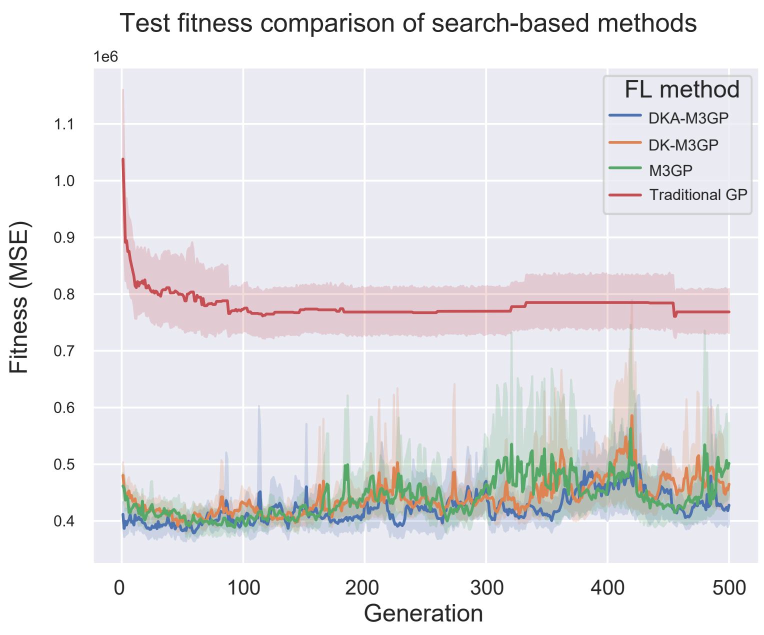


Figure 7.13: Comparison of Traditional GP, M3GP, DK-M3GP and DKA-M3GP on test data.

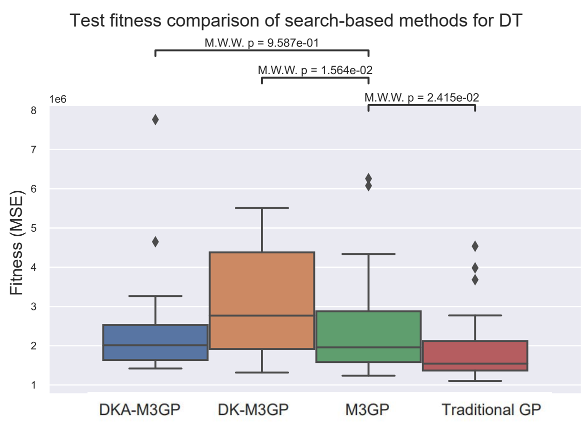
aim to become more complex for them to fit the task better. If that task, in this case training, does not represent the testing task well, this complexity would only be counter-productive. It would be interesting to research this hypothesis further.

7.4.1.2 Comparing to classical Feature Learning methods

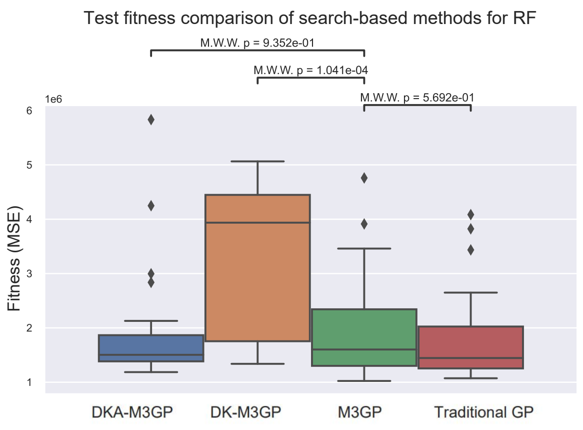
For a more general comparison of FL methods, I compare above search-based methods to classical FL approaches. See fig. 7.16 for a comparison of training performance of M3GP and DKA-M3GP and classical methods. The classical methods compared to are RS FS, FeatureTools FS, PCA, and No FL, as introduced in section 7.1. In terms of statistical significance, DKA-M3GP performs best for DT and ties for best for RF and SVM. Furthermore, for MLP it is only beaten by No FL (notice that fig. 7.16c is crooked due to outlier removal). The above shows that DKA-M3GP can be a good FL method.

In fig. 7.17 the test performance of M3GP and DKA-M3GP compared to classical methods is depicted. No FL performs best for all models except for DT, where it performs second best only to FeatureTools FS. Overall, high training performance is not necessarily translated to high test performance, just like we saw before.

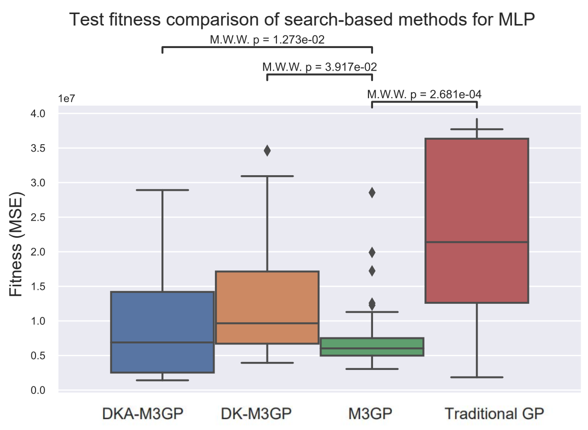
In line with the hypothesis I presented in section 7.4.1.1, the best performing methods learned the least complex models, and are not influenced by a DT in their fitness function, like for RS FS. As such, they were not overfitted during training.



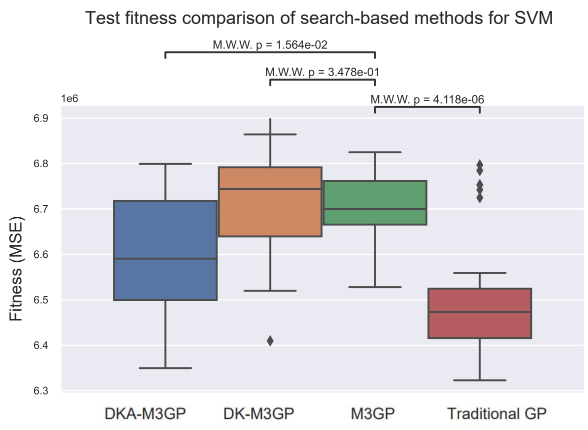
(a) ML pipeline with DT



(b) ML pipeline with RF



(c) ML pipeline with MLP



(d) ML pipeline with SVM

Figure 7.14: MSE comparison of the search-based methods within ML pipelines for the BoomBikes dataset. The ML pipelines differ only on the models. In the figures, outliers were taken out for aesthetical reasons. At the top of each figure, statistical significance scores comparing the models with M3GP are portrayed. I included the outliers for the calculation of the statistical significance score.

7.4.1.3 Time

Finally, I compare the running time of the different methods in fig. 7.18. There are no surprises here: DKA-M3GP is slowest, then DK-M3GP, then M3GP, and finally Traditional GP, all statistically significant. Classical methods were left out as their runtime is negligible.

As training time is sometimes limited in real-life applications, I also compared the methods on budget. In this case, I ran them with a time limit of 20 minutes. See the comparison of test performance with the budget in fig. 7.19. The performance of DKA-M3GP suffered from the limited budget for MLP. However, it did not suffer for DT, RF and SVM, and even improved slightly. As the DKA-M3GP is very slow it

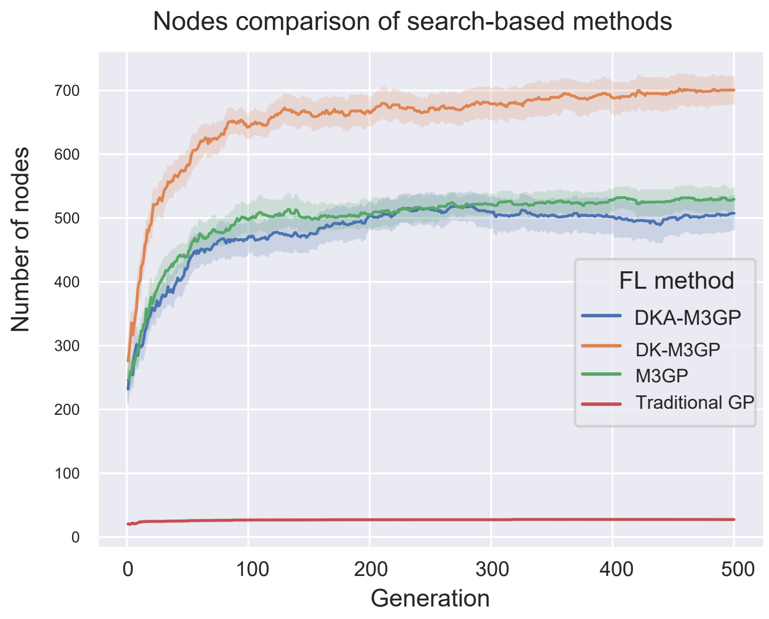
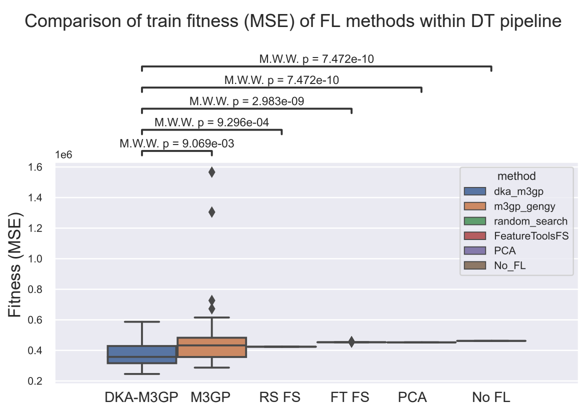
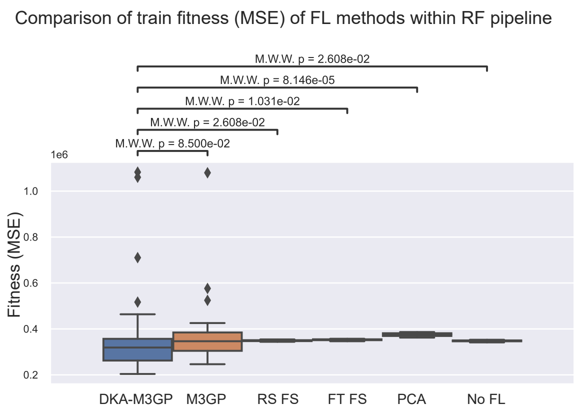


Figure 7.15: Complexity comparison of the best individuals of the FL methods over generations. DK-M3GP is most complex, whereas Traditional GP is least-complex by far.

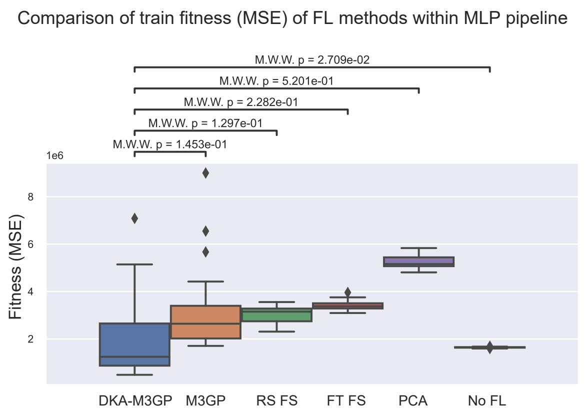
was only able to run a limited number of generations. On average, it only ran for 7.2 generations. The resulting individuals had an average of ±300 nodes, which is much less than the average of ±500 nodes for individuals after 500 generations. This again affirms the hypothesis of a negative correlation between complexity and performance.



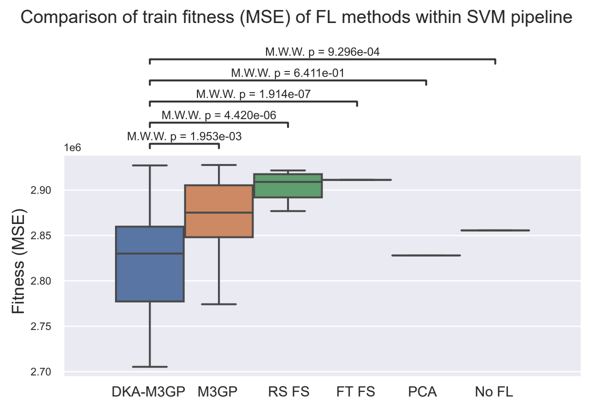
(a) ML pipeline with DT



(b) ML pipeline with RF

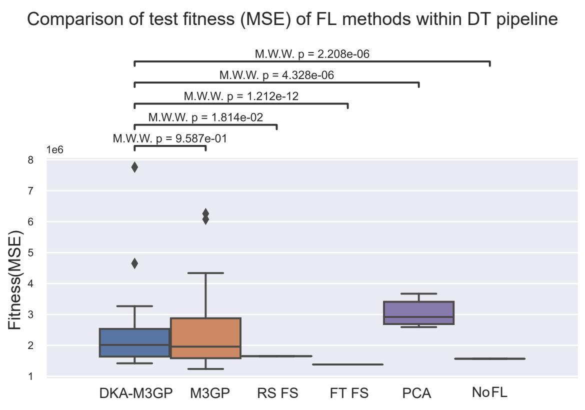


(c) ML pipeline with MLP

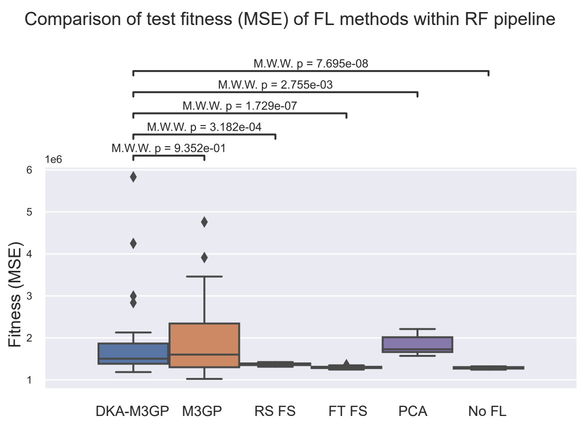


(d) ML pipeline with SVM

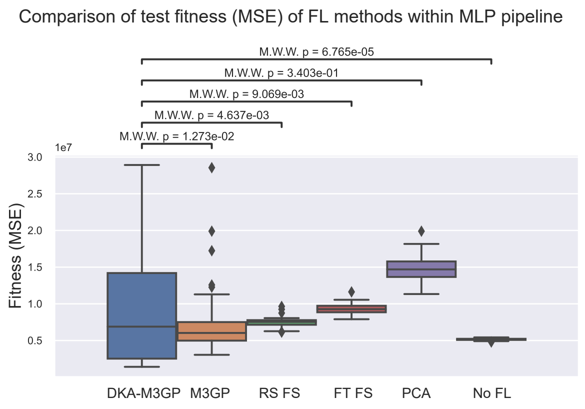
Figure 7.16: MSE comparison of training performance of M3GP and DKA-M3GP with classical methods within ML pipelines for the BoomBikes dataset. In the figures, outliers were taken out for aesthetical reasons.



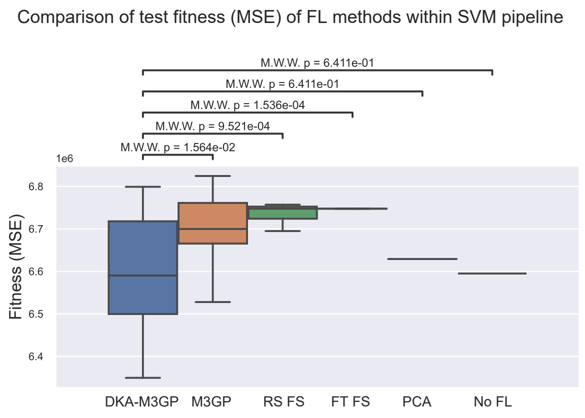
(a) ML pipeline with DT



(b) ML pipeline with RF



(c) ML pipeline with MLP



(d) ML pipeline with SVM

Figure 7.17: MSE comparison of test performance of M3GP and DKA-M3GP with classical methods within ML pipelines for the BoomBikes dataset. In the figures, outliers were taken out for aesthetical reasons.

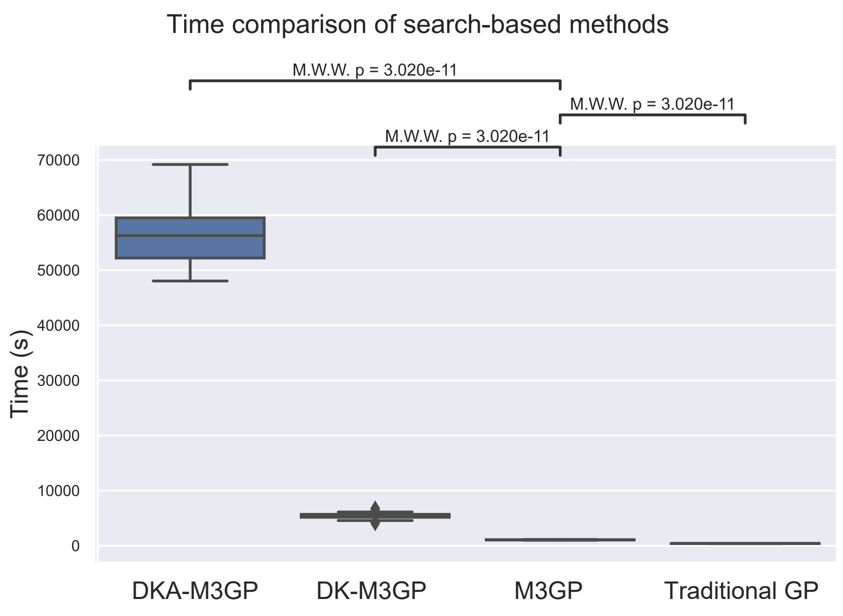


Figure 7.18: Comparison of the run time of the different methods for the BoomBikes dataset. DKA-M3GP is slowest, then DK-M3GP, then M3GP, and finally Traditional GP, all statistically significant. Note that the statistical significance score is the same because it is the highest possible.

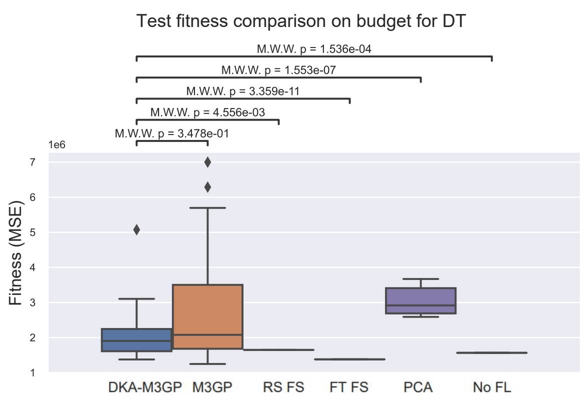
7.4.2 The credit-g dataset

First, I compare my methods against the other search-based methods (excluding RS FS) in section 7.4.2.1. In section 7.4.2.2, I include the classical methods. Finally, I compare the FL methods on time in section 7.4.2.3.

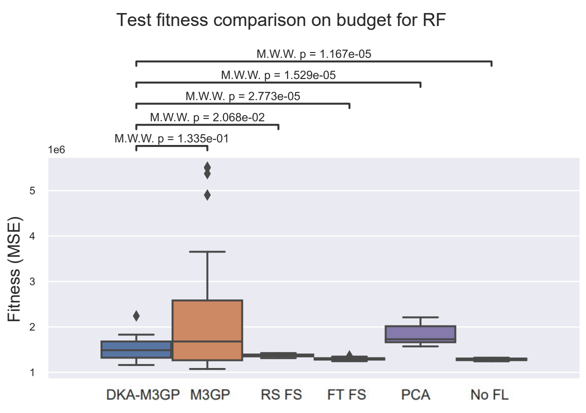
7.4.2.1 Comparison of search-based Feature Learning methods

In figs. 7.20 and 7.21 I compare the training fitness and test fitness of the different search-based methods, respectively.

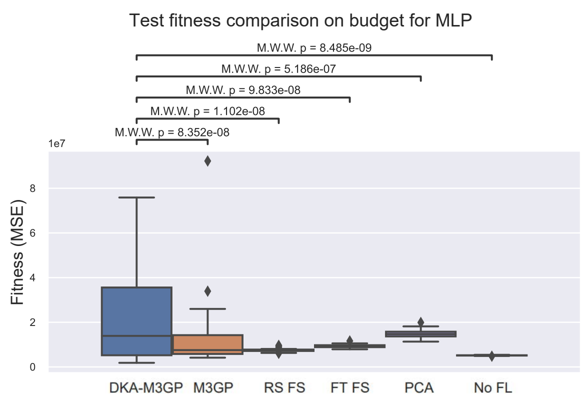
Consider the training fitness progression in fig. 7.20. Like the Batista et al. [9] implementation, Traditional GP does not evolve actual solutions but facilitates classifying everything as the positive class (low-risk). Classifying each data point as low-risk gives quite a high fitness (± 0.825). Furthermore, DKA-M3GP performs best, followed closely by DK-M3GP. Adding aggregation does not improve the evolutional process in the same proportions as it does for the BoomBikes dataset. Looking more closely at the credit-g dataset (see section 4.2) we see that aggregating over different categories generally does



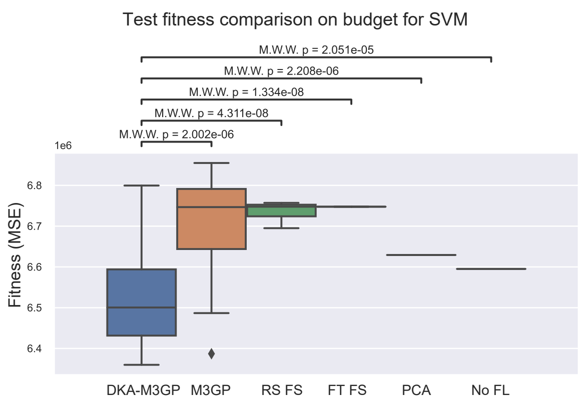
(a) ML pipeline with DT



(b) ML pipeline with RF



(c) ML pipeline with MLP



(d) ML pipeline with SVM

Figure 7.19: MSE comparison of test performance on a 20 minutes budget of M3GP and DKA-M3GP with classical methods within ML pipelines. In the figures, outliers were taken out for aesthetical reasons.

not give very insightful information. In fig. 4.3 the features show risks per unique category value, all averaging around 0.7, with only slight variances. For example, fig. 4.3c shows only slight differences between different job categories. Including the average for those categories as a building block adds little information. On the other hand, comparing my methods to standard M3GP shows an improvement. This suggests that the evolutional process benefits from the inclusion of domain knowledge in the grammar.

Now consider the test fitness progression in fig. 7.21. As expected, the fitness of Traditional GP does not change throughout the evolution. The other progressions show a close resemblance to the test fitness progression for the BoomBikes dataset. Again, it shows all the methods overfit the training data.

Following this, I compared the methods within the different ML pipelines. See fig. 7.22 for the results.

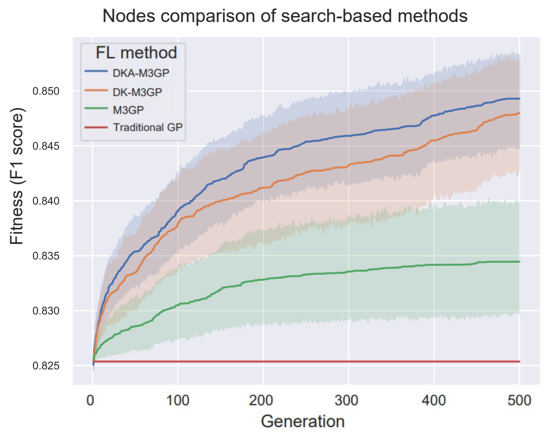


Figure 7.20: Comparison of training fitness of search-based methods.

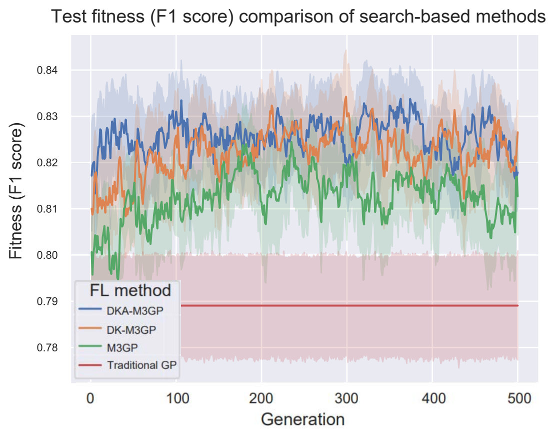


Figure 7.21: Comparison of test fitness of search-based methods.

Across all models, we see that Traditional GP performs best, by predicting all data as low-risk. The other models do not perform significantly different, except for the ML pipeline with an SVM classifier; standard M3GP outperforms my methods.

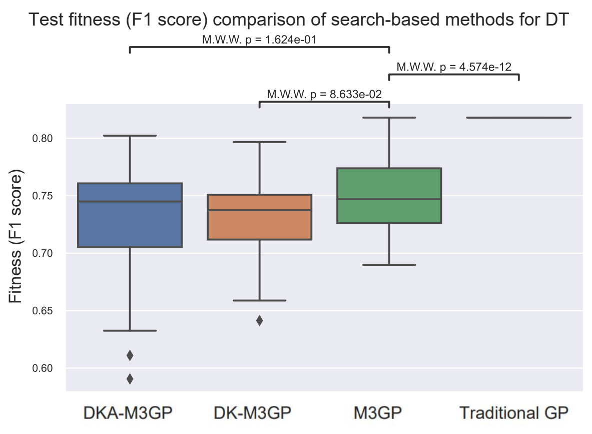
7.4.2.2 Comparing to classical Feature Learning methods

I continue with a general comparison of FL methods; I compare above search-based methods to classical FL approaches. See fig. 7.23 for a comparison of training performance of M3GP, DKA-M3GP and the classical baselines. The classical baselines used are RS FS, FeatureTools FS, PCA, and No FL, as introduced in section 7.1. RS FS consistently has a fitness of ± 0.825 by classifying all data as low-risk. It significantly performs best for DT, MLP and SVM. Clearly, RS FS does not evolve the best solution, but the fitness function does not reveal that. For RF No FL performs best on average. For all ML pipeline models the best models generated by DKA-M3GP outperform or perform on par with the best generated models of all other methods. This shows the potential of DKA-M3GP.

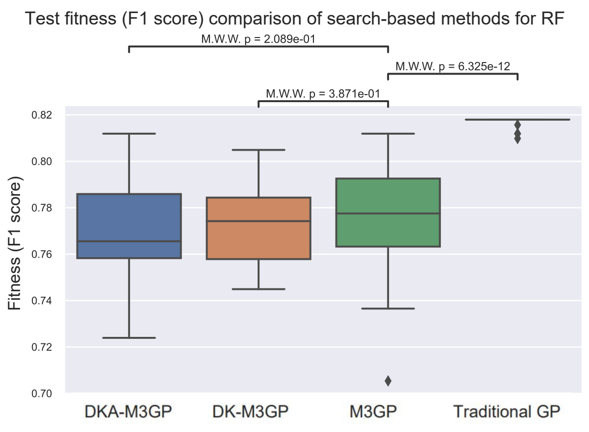
In fig. 7.24 the test performance of M3GP and DKA-M3GP compared to classical methods using the weighted F_1 score is depicted. The story is the same as for the training data, RS FS on average performs best by classifying all data as low-risk. Still, the best models evolved by DKA-M3GP perform well when compared to the other methods.

As the fitness function using the weighted F_1 score did not reveal the models evolved by RS FS as undesirable, I also ran the tests using the binary F_1 score. See the results in fig. 7.25. No method consistently predicts all data as a single class. On average, DKA-M3GP does not perform particularly well. Still, the best performing runs, generally outperform the classical methods, but DKA-M3GP shows little consistency.

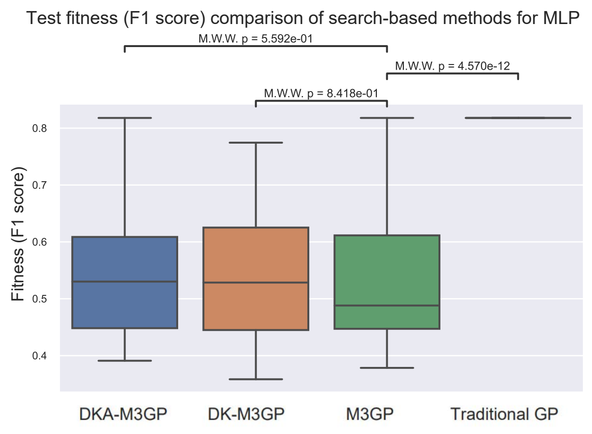
When comparing our methods during evolution, domain knowledge and aggregation clearly show to



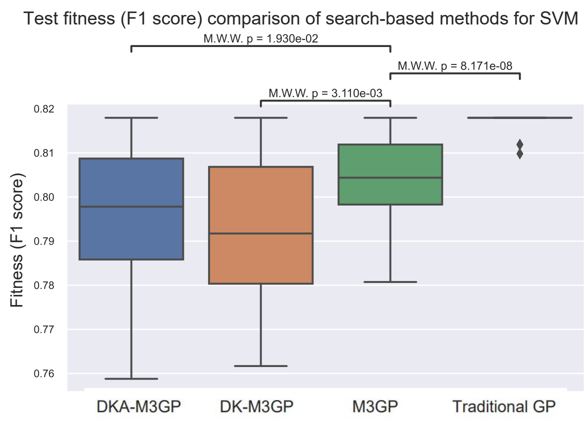
(a) ML pipeline with DT



(b) ML pipeline with RF



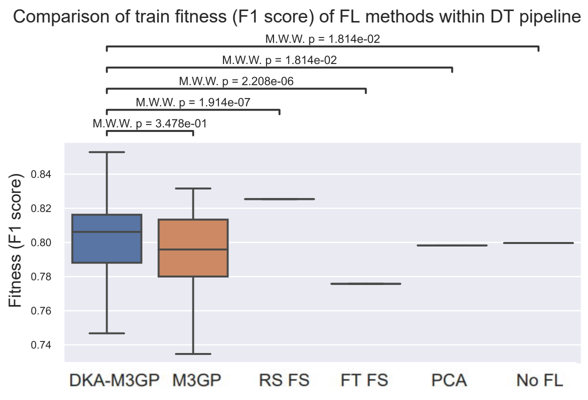
(c) ML pipeline with MLP



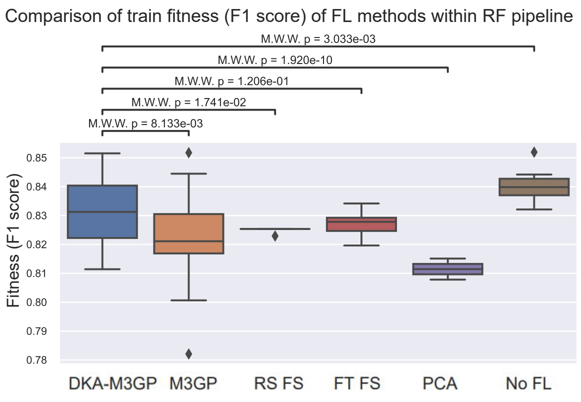
(d) ML pipeline with SVM

Figure 7.22: Weighted F_1 score comparison of the search-based methods within ML pipelines for the credit-g dataset. The ML pipelines differ only on the models. In the figures, outliers were taken out for aesthetical reasons. At the top of each figure, statistical significance scores comparing the models with M3GP are portrayed. I included the outliers for the calculation of the statistical significance score.

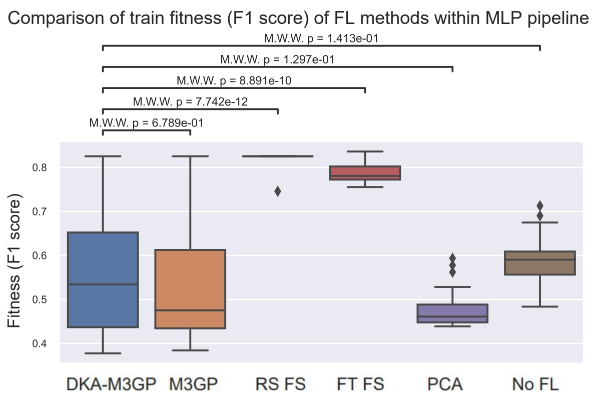
be beneficial. As for the BoomBikes dataset, this is not translated to a good performance within the ML pipelines. The learned features do not enhance the prediction performance of the models. From this I conclude that either the ML pipelines should be revisited, or the fitness function is ill-defined. Moreover, the fact that classifying the whole dataset as one class returns high fitness shows the need for an improved fitness function. I discuss this more elaborately in section 8.3.



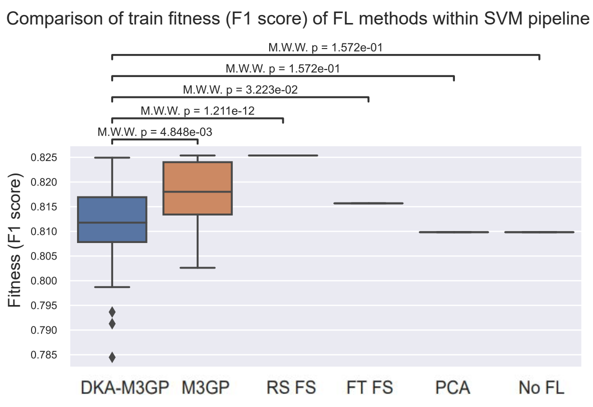
(a) ML pipeline with DT



(b) ML pipeline with RF



(c) ML pipeline with MLP



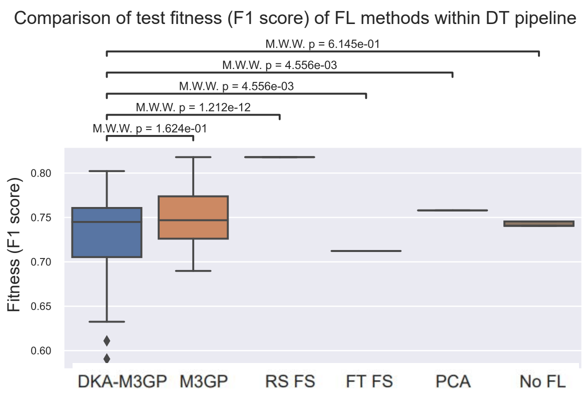
(d) ML pipeline with SVM

Figure 7.23: Weighted F_1 score comparison of training performance of M3GP and DKA-M3GP with classical methods within ML pipelines for the credit-g dataset. In the figures, outliers were taken out for aesthetical reasons.

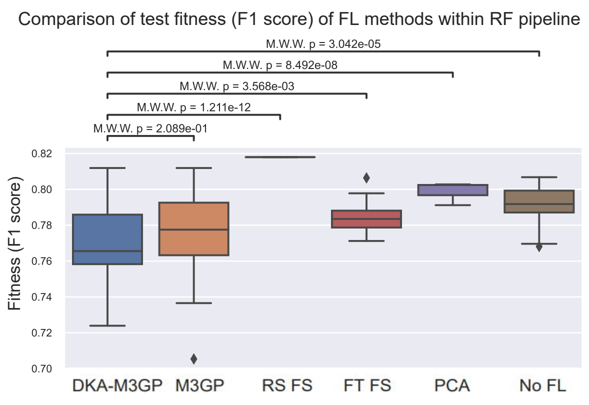
7.4.2.3 Time

Finally, I compare the running time of the different methods in fig. 7.26. Results resemble the results for the BoomBikes dataset. Classical methods were left out as their runtime is negligible.

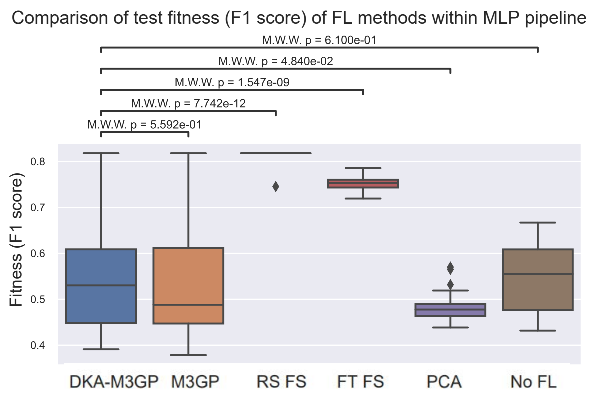
As with the BoomBikes dataset, I ran the FL methods with a budget of 20 minutes to see how well they performed with limited resources. See the results in fig. 7.27. The results show that on a budget, DKA-M3GP and DK-M3GP perform slightly worse, especially when looking at the best-found fitness.



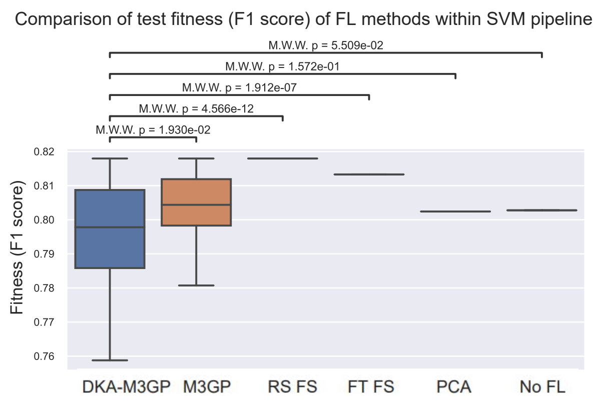
(a) ML pipeline with DT



(b) ML pipeline with RF

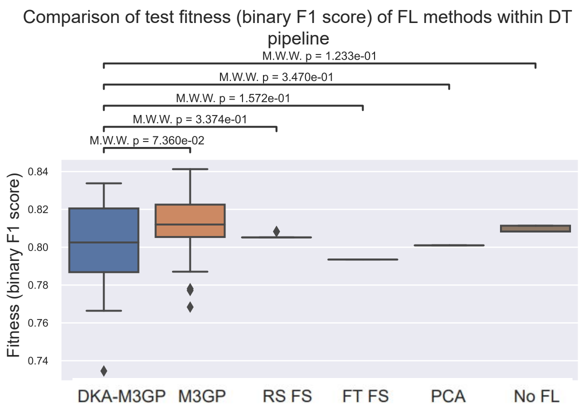


(c) ML pipeline with MLP

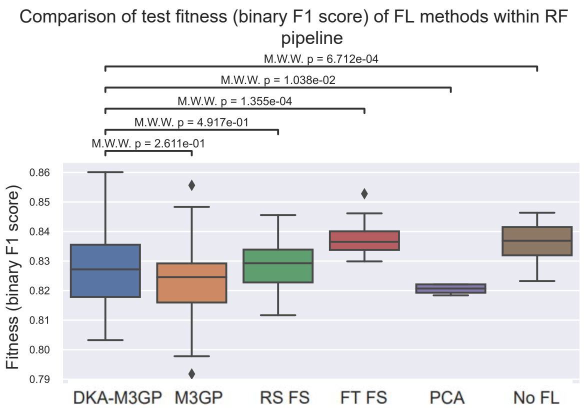


(d) ML pipeline with SVM

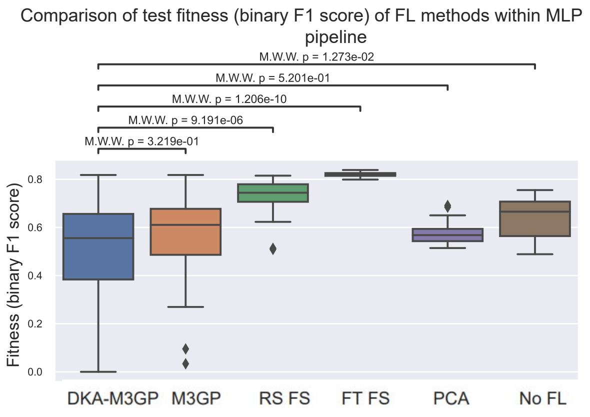
Figure 7.24: Weighted F_1 score comparison of test performance of M3GP and DKA-M3GP with classical methods within ML pipelines for the credit-g dataset. In the figures, outliers were taken out for aesthetical reasons.



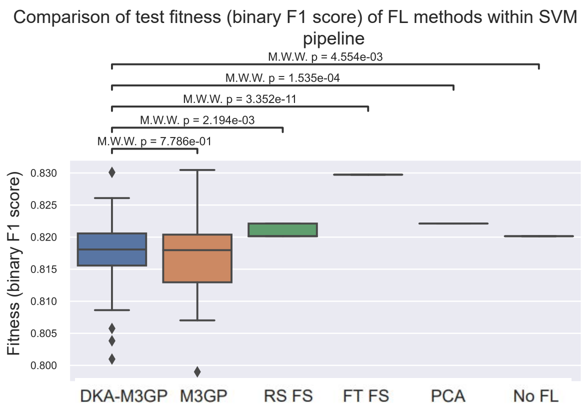
(a) ML pipeline with DT



(b) ML pipeline with RF



(c) ML pipeline with MLP



(d) ML pipeline with SVM

Figure 7.25: Binary F_1 score comparison of test performance of M3GP and DKA-M3GP with classical methods within ML pipelines for the credit-g dataset. In the figures, outliers were taken out for aesthetical reasons.

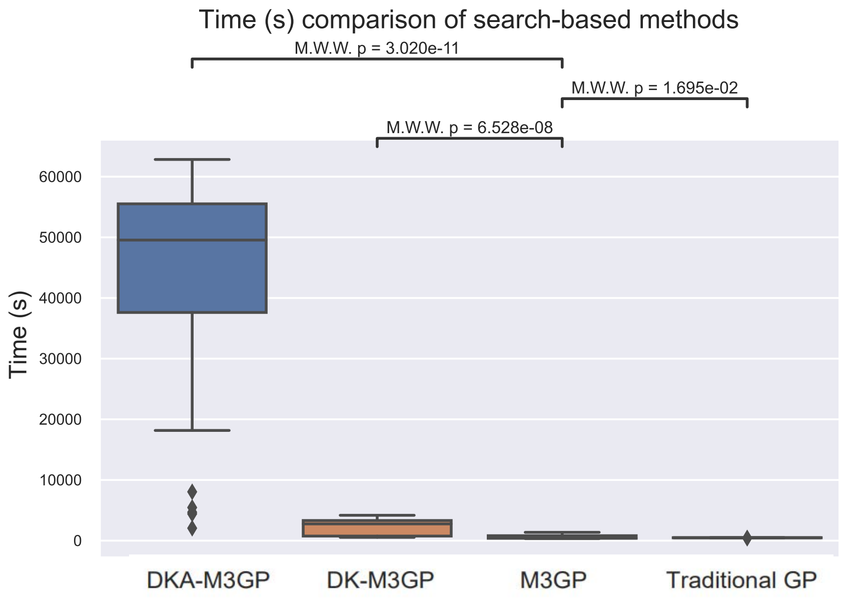
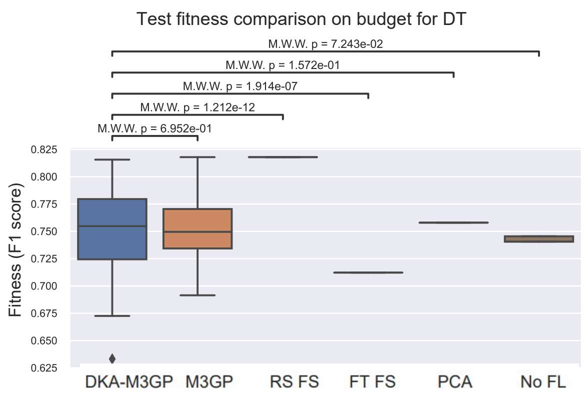
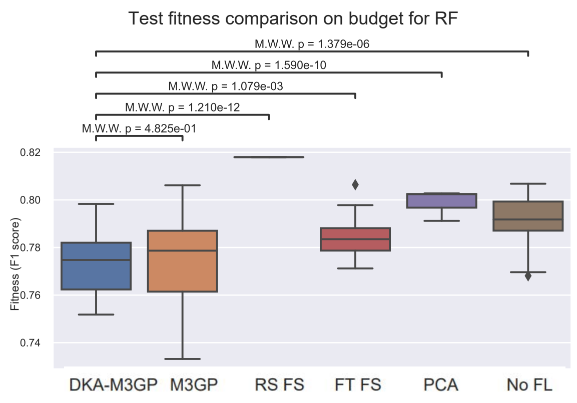


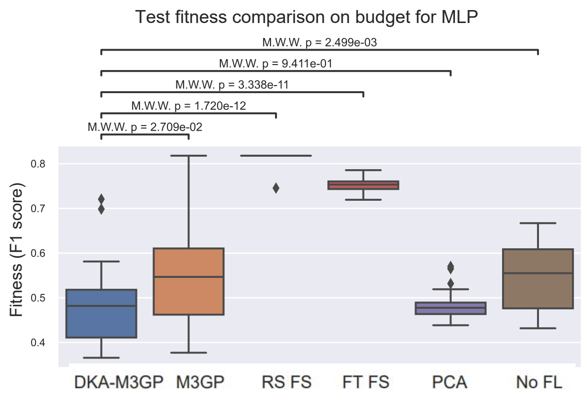
Figure 7.26: Comparison of the run time of the different methods for the credit-g dataset. DKA-M3GP is slowest, then DK-M3GP, then M3GP, and finally Traditional GP, all statistically significant. Note that the statistical significance score is the same because it is the highest possible.



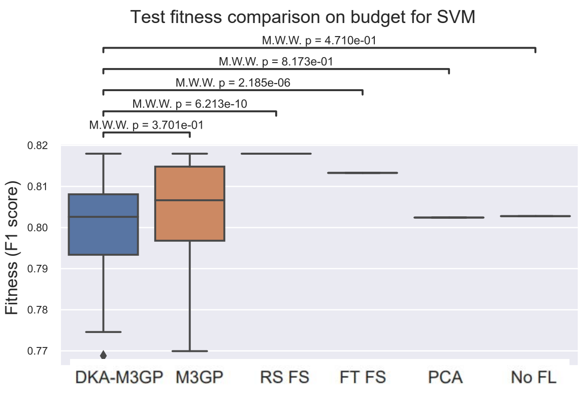
(a) ML pipeline with DT



(b) ML pipeline with RF



(c) ML pipeline with MLP



(d) ML pipeline with SVM

Figure 7.27: MSE comparison of test performance on a 20 minutes budget of M3GP and DKA-M3GP with classical methods within ML pipelines. In the figures, outliers were taken out for aesthetical reasons.

7.4.3 Expression comparison of domain knowledge incorporation

To get an idea of the interpretability improvement of incorporation of domain knowledge in the grammar, I now highlight an evolved feature from the feature sets produced by the FL methods M3GP and DK-M3GP, for the seed 0, for the BoomBikes dataset. See the full evolved feature sets for the above models for seed 0 in chapter A.

For M3GP, the evolved features are combinations of original features using standard arithmetic op-

```
workingday + temp - (weekday / season) - (mnth * temp) + windspeed + (((season * atemp) + (atemp
        ↪  / yr)) / mnth)
```

Listing 7.1: The first evolved feature by M3GP for seed 0. See full feature set in listing A.1.

```
if (mnth == December):
    if (mnth inbetween (October,November)):
            atemp / hum
    else:
            temp * temp
else:
    atemp + atemp
```

Listing 7.2: Last evolved feature by DK-M3GP for seed 0. See full feature set in listing A.2.

erators. For example, see the feature in listing 7.1. The feature combines multiple categorical features using arithmetic operators, making no sense. For example, see $workingday - temp$ and $weekday/season$.

Looking at DK-M3GP, we see no categorical features being combined through arithmetic operations, but only through if statements. See, for example, the feature presented in listing 7.2. The feature returns different values based on the month of a data point. This feature can be displayed in a tree-like format (see fig. 7.28 for an example).

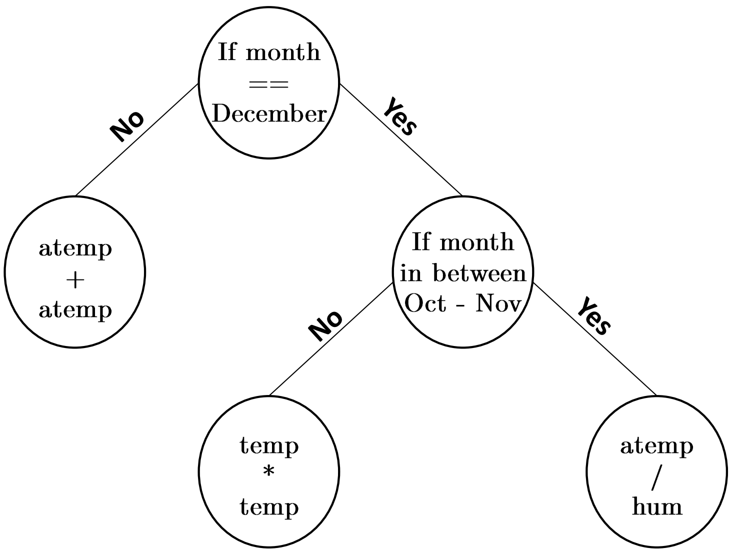


Figure 7.28: The feature from listing 7.2 in a tree-like format.

Showing the feature in a tree-like format improves the interpretability of the feature. In this case, I manually create the tree, but it could be automated, which would be a great enhancement of Genetic Engine. Notice that the ES of the M3GP-evolved feature in listing 7.1 and the DK-M3GP-evolved feature in listing 7.2 is both 23, even though the DK-M3GP-evolved feature is far more easily interpreted and less complex.

Chapter 8

Conclusion

In this chapter, I discuss the results and conclude on the thesis. I start with discussing the results in section 8.1. Then, I conclude on the thesis in section 8.2. Finally, in section 8.3 I discuss interesting directions for future research.

8.1 Results discussion

In chapter 7 I presented the results in two main parts. First, I compared M3GP implemented in Genetic Engine to M3GP as implemented by Batista et al. [9]. This was done to justify Genetic Engine as an acceptable tool for an M3GP GGGP-based implementation. I discuss the results of this evaluation in section 8.1.1. Following, I introduced DK-M3GP and DKA-M3GP and implemented them through Genetic Engine. These methods were applied to two complex datasets with heterogeneous data types. I discuss the evaluation of DK-M3GP and DKA-M3GP in section 8.1.2.

8.1.1 M3GP implementations comparison

The Genetic Engine and the Batista et al. [9] implementation resulted in different training progressions. For the BoomBikes dataset, the Genetic Engine implementation performed worse than the Batista et al. [9] implementation, but for the credit-g dataset, the Genetic Engine implementation performed better. Furthermore, the Genetic Engine implementation showed to be generalizing better over the training set. Lastly, when included in an ML pipeline, the Genetic Engine implementation showed to have a better overall performance. The difference in performance is not expected, as the algorithms are the same. There are multiple possible reasons for this discrepancy, found in differences in implementation.

Firstly, the random generation of nodes is done differently in both implementations. This results in differences in population diversity. Even though Genetic Engine showed to perform well enough, it would benefit tremendously from a population-diversity-analysis system to keep track of the diversity of the population throughout the evolutionary process. Furthermore, (ϵ-)lexicase selection [40, 57] should be implemented as it improves population diversity [41].

Secondly, the Batista et al. [9] implementation implements mutation and crossover differently from the Genetic Engine implementation. When operating on individuals the Batista et al. [9] implementation initially disregards the tree depth. Once a new individual is generated, the Batista et al. [9] implementation checks whether the individual has the correct depth. If it is too deep, the Batista et al. [9] implementation discards the individual. In the Genetic Engine implementation, operations are depth-safe. When operating on a node, the Genetic Engine implementation first filters the candidate replacements on their depth, making sure that candidates that are too deep cannot be selected. As such, each operation creates trees that are within the depth limit. Besides an evident performance improvement on the the Genetic Engine implementation side, there is an unexpected side effect. In the the Batista et al. [9] implementation implementation, there is a difference in the probability of trees being discarded after the evolutionary operation. More specifically, if the node that is selected for operation is deeper in the tree, there is a higher probability that the resulting tree is too big. The deeper the node is that is operated on, the bigger the chance of a failed disruption propagation [59], i.e., for the operation not to have effect. As the implementation the Batista et al. [9] implementation accidentally discards more trees that are operated on at a deeper level, there will be less failed disruption propagation, resulting in a higher diversity. To improve Genetic Engine, we should give less deep nodes a larger chance of being operated on.

Thirdly, the method of crossover is different between the two implementations. In the Batista et al. [9] implementation two trees are selected, and of each tree a node is selected. The subtrees of the selected nodes are then swapped, resulting in two trees that together contain all the same nodes of the original individuals. In the Genetic Engine implementation two trees undergo crossover independently. First, the Genetic Engine implementation selects a node of each tree, and creates a new tree, using the root of the first tree. Following the creation, it selects another new node from each tree, and, again, it creates a new tree, using the root of the second tree. The result is two new trees that do not necessarily contain all the same nodes of the original individuals. What the result is of the different crossover implementations, I do not know; further research should point that out.

Finally, there is a negligible difference in mutation and crossover methods. In the Batista et al. [9] implementation a mutation or crossover is done either (1) on the complete list of features (removing or deleting a feature, or swapping feature subsets), or (2) on an individual feature (standard tree mutation and crossover). In the Genetic Engine implementation this is almost the same. The difference is that the feature set has a parent node itself, and when the second of above options is chosen, the whole feature set can be mutated. This results in a completely new individual. The probability of this happening is $\frac{1}{2} * \frac{1}{\#nodes}$, which averages to one individual per generation for a population of 500 when the average individual size is 250 nodes, as is the case at initialization (see fig. 7.4). This can be interpreted as a cumbersome implementation of novelty, or as a mutation with extremely low locality. Again, the impact of this I do not know, but I expect it to be negligible.

Except for performance differences, there is also a clear difference in execution speed of the two im-

```
<A>     ::= <B>
<B>     ::= <C>
<C>     ::= 1
          | 2
```

Listing 8.1: Grammar with redundant information.

plementations. Foremost, the Genetic Engine implementation is much quicker than the Batista et al. [9] implementation. There are multiple reasons for this, of which the impact I did not analyse. One important aspect of Genetic Engine is that trees are directly evaluated while traversing the trees, allowing for tremendous speed-ups. Furthermore, throughout the evolution, trees are operated on in a depth-safe manner, so that no trees need to be discarded. Moreover, Genetic Engine implements in-node storage, making it easy to retrieve basic tree information. Genetic Engine can be made quicker by storing subtree evaluations in nodes alongside the basic information. Lastly, Genetic Engine can incorporate deduplication. All the above I elaborate on in a late-breaking abstract submitted to EuroGP [47].

Finally, there is a difference between the Genetic Engine implementation and the Batista et al. [9] implementation concerning the expression size (ES) of the evolved models. In line with PonyGE2 [32], Genetic Engine implements Position-Independent (PI) grow as defined by Fagan et al. [31], which influences the ES of individuals. The PI grow method is used to generate random nodes, always with the maximum depth possible. It can be changed to randomly select a depth smaller than the maximum and generate a random node for that depth, using PI grow.

Furthermore, Genetic Engine has followed PonyGE2 in the method of counting nodes. PonyGE2 counts nodes for every production step, even when a production does not materialize in an actual node. For example, see listing 8.1. Each individual produced by this grammar is a single literal node with either value 1 or 2. PonyGE2 counts A, B and C as nodes as well, making each individual four nodes in total. We implemented Genetic Engine to be comparable with PonyGE2, so it follows the same node counting method, contributing to the larger ES of the Genetic Engine implementation. Genetic Engine should implement both node-counting methods to allow for comparisons with a wider variety of methods.

The different method of counting nodes also impacts the way the depth of a tree is measured. For the Genetic Engine implementation, each production increase the depth by one. An individual formed from listing 8.1 would thus have a depth of 4. This impacts the depth of individuals in Genetic Engine throughout the evolutionary process. A better evaluation should be done where these methods are aligned for both implementations.

Altogether, Genetic Engine shows to be a reliable GGGP framework for FL, when compared to the GGGP implementation of Batista et al. [9]. Together with the performance evaluation of Genetic Engine compared to PonyGE2 in section 5.4 and the participation in the GECCO'22 SRBench competition, I conclude that Genetic Engine shows great promise as a general GGGP framework. We plan to submit a paper concerning its implementation details in the near future. Furthermore, we intend to conduct a survey concerning the usability of Genetic Engine.

8.1.2 Evaluation of DK-M3GP and DKA-M3GP

Incorporating domain knowledge in the search process, as done in DK-M3GP, had varying outcomes in terms of prediction performance. When comparing DK-M3GP with M3GP for the BoomBikes dataset, the effect on the training fitness progression was negligible. On the other hand, regarding the comparison for the credit-g dataset, the effect on the training fitness progression was plain; domain knowledge improved the performance. Moreover, DK-M3GP showed little success in generalizing to functioning within the ML pipelines, scoring worse than or on par with M3GP in almost all cases.

Except for a deterioration of performance, DK-M3GP had two other prospected advantages. Firstly, the categorization was necessary for aggregation to be added as part of the grammar. This advantage is very practical and functioned as desired.

Secondly, DK-M3GP was expected to evolve more easily interpreted individuals. We measured interpretability through feature complexity using expression size (ES) as defined in section 2.1.3. Looking at ES has the advantage of the implementation being straightforward, and of results being easily comparable. On the other hand, it does not cover delicate differences in interpretability. For example, Poursabzi-Sangdeh et al. [84] found that decreasing model size does not necessarily improve interpretability, even though predictions could be more easily simulated. Moreover, interpretability is highly subjective [112], even more so when models do not have the same structure. As DK-M3GP incorporates if statements, the models it evolves can be visualized in a tree-like structure (see section 7.4.3). This tree-like structure is supposed to be more easily interpreted, but comes at a cost in ES. For an if statement, we need a *Condition* node, either an *Equal* node or a *NotEqual* node, that take a *Category* node as input, or an *InBetween* node, that takes an *IBCategory* node as input. As such, a Condition node comprises at least a total of 7 nodes. The number of nodes needed to construct an if statement is not in line with the interpretability of its possible tree structure. Consequently, I have not been able to correctly assess the interpretability gains of DK-M3GP. To conclude, in the future I intend to measure interpretability using another method than comparing ES, such as the method used in the SRBench Competition of GECCO'22, where the trust rating of a domain expert was combined with the accuracy and complexity of the model.

DK-M3GP also played a key role in the development of DKA-M3GP, as categories are necessary to aggregate based on the values of other features. DKA-M3GP had drastic effects on the prediction results compared to M3GP and DK-M3GP for the BoomBikes dataset. For the credit-g dataset, there is only a slight improvement compared to DK-M3GP, and both show a large improvement over M3GP. The improvement of DKA-M3GP over standard M3GP can thus mostly be attributed to the incorporation of domain knowledge. DKA-M3GP showed to be more reliable in comparison with DK-M3GP as a FL method to be incorporated within ML pipelines.

DKA-M3GP was originally built using aggregations over the whole historic dataset. Firstly, this was slow and would not scale well for larger datasets. Secondly, for the BoomBikes dataset, this method did not perform well, so I chose to make the window size smaller. This performed well for certain features, but not for all. Finally, I decided to let the window size be evolved, as to allow the solution to select the best-performing size. This worked well, but there are other aggregation parameters I didn't experiment with. For example, the window size is currently taken regarding the complete dataset, and then the relevant data points that match the condition are filtered out. It would be interesting to research reversing this order, first filtering out relevant data points, and then select the window size. Then, for most windows, there would be a similar amount of relevant data points. Furthermore, I only implemented an averaging method. I would like to implement methods like the maximum and minimum, the median, the last or first historic value, and other functions implemented in Vectorial-GP by Azzali et al. [6].

From this, a question arises: why does one method perform well on one dataset, and one on the other dataset? Firstly, the BoomBikes dataset is less complex, with fewer features. Furthermore, the categorical features in the BoomBikes dataset all have an order. Rewriting ordered categorical features with numeric substitutes is more sensible than doing so for unordered categorical features, as numeric values have an inherent order. This might be the reason for domain knowledge to have an insignificant effect on performance for the BoomBikes dataset. On the other hand, the credit-g dataset is more complex, with a greater number of features, and with many unordered features. Representing these unordered features with numeric values is not logical, as it encodes a non-existent order into each feature. In this case, keeping the categorical values as they are, proves to be more beneficial, which is represented by the increased performance due to domain knowledge incorporation.

On the BoomBikes dataset, aggregating over the value of a feature drastically improves performance. Aggregating over historic data seems to be a good representative of the data. This might be because the data is simple, and aggregating over the value of a single feature already covers the data. For the credit-g dataset, on the other hand, aggregation does not improve performance significantly. The credit-g dataset is more complex, and there is no single feature that influences the target value as strongly as, for example, the *month* feature (see fig. 4.2a) does in the BoomBikes dataset. The difference in risk between the values of features is less obvious for the credit-g dataset. Furthermore, the data is imbalanced, so that aggregation will mostly result to values larger than 0.5, which, in a binary dataset, is rounded to the positive class. All things considered, the credit-g dataset does not seem to benefit from aggregation, as aggregating over other individuals does not seem to represent its data well. It would be interesting to experiment with aggregation over the values of multiple features instead of only one. This way, the more complex relationships between features are included in the aggregation process, allowing for more complex datasets to be represented by aggregations.

8.2 Conclusion

Altogether, improving the performance of the training fitness by incorporating domain knowledge or aggregation into the grammar is not reserved for all datasets. Some datasets will benefit from one or the other, and some will not. I hypothesize that the higher the number of unordered features in a dataset, the higher the impact of incorporating domain knowledge is. Aggregation in its current format benefits simpler datasets. The effects of introducing more complex aggregation methods is an interesting direction for future research. To conclude, DK-M3GP and DKA-M3GP have shown that they can improve the performance of FL methods. Given the domain-specific knowledge, these methods can be generally applied to any domain.

The experiments done in this thesis had two parts. First, FL methods were trained as FL methods only and analysed on both training and test data. Then, the FL method was analysed as part of an ML pipeline. In retrospect, I have spent unnecessary time on building the complete experiment, even though looking at the FL method is the first step. I assumed the need of analysing the methods within full ML pipelines would be important and interesting. Not only are the results incoherent due to overfitting of the DT in the fitness function, I also think there are other, more interesting paths I could have pursued after implementing DK-M3GP and DKA-M3GP. Reflecting on my choices, I would take smaller steps, and make less assumptions beforehand, were I to do the thesis again.

Because we were designing methods that could be applied to any dataset, I did not concern much with the selection of the use case. I selected them on few criteria besides the availability of additional domain information, and for them to be either a time series or a panel, so that aggregation would be applicable. In retrospect, I should have spent some time analysing to what extent recent data represented the historic data, in order for the chance of overfitting to be diminished. Additionally, we could have explored other fitness functions, such as the macro weighted F_1 score from scikit-learn [82], or changing the DT seed in the fitness function for each generation.

To conclude, in this thesis I proposed the usage of Genetic Engine, an individual-representation-independent GP framework completely implemented in Python, to allow a larger audience the benefits of GP and its various adversaries. Through Traditional GP and M3GP, I applied Genetic Engine to the problem of FL on two heterogeneous, complex datasets. I evaluated the performance of Genetic Engine by comparing the M3GP implementation of Genetic Engine to M3GP implemented by [9], and showed that Genetic Engine performs well, but also included multiple future improvements. I introduced two novel GGGP-based FL methods called DK-M3GP and DKA-M3GP and implemented them in Genetic Engine. Given the domain-specific knowledge, these methods are generalizable for other datasets.

Lastly, I researched the impact of domain-knowledge incorporation, entity aggregation, and heterogeneous data types to GP-based FL, by analysing the results of DK-M3GP and DKA-M3GP.

8.3 Future work

In this section, I discuss prospected future work. First, I discuss the improvements we want to implement in Genetic Engine in section 8.3.1. Then, in section 8.3.2, I propose future research concerning DK-M3GP and DKA-M3GP. Finally, I introduce some other research questions that came up in my thesis in section 8.3.3.

8.3.1 Improvements to Genetic Engine

As mentioned before, Genetic Engine can do with some improvements. Firstly, we would like to improve the execution time by implementing in-node storage of subtree evaluations, and deduplication as outlined by us [47]. Secondly, the node and depth counting methods should be enhanced to allow for both the PonyGE2 counting method to be used, and the one used by Batista et al. [9]. This will give us more freedom when comparing Genetic Engine with other GGGP frameworks. Thirdly, Genetic Engine currently uses standard PI grow with the user-defined max depth, resulting to trees being as deep as possible at initialization. A new method that does not always initialize the deepest-possible trees, but initializes trees with depths randomly smaller than the maximum depth, should be implemented and tested. Finally, the evolutionary operations should favour less deep nodes when selecting nodes for operations. As deeper nodes have a larger probability of failed disruption propagation Langdon et al. [59], operating on less deep nodes improves diversity.

Except for the above enhancements to the implementation, I would like to implement three features in Genetic Engine. First, lexicase selection [41, 57] should be implemented to allow the user the diversity-benefits of this selection method. Second, a local search method for numeric nodes (integers and floating points) as, for example, done by Kommenda et al. [54] can benefit the search process by optimizing numeric values in a more numeric-suitable way. Lastly, Genetic Engine would benefit immensely from a diversity-tracking interface or method. If we could see the diversity of the population throughout the evolutionary process, Genetic Engine would have a great advantage over other frameworks. One such interface is given by Burlacu et al. [14].

Finally, to analysis the usability of Genetic Engine, a user survey should be conducted. This allows us to compare the usability with PonyGE2, and see whether our hypotheses are true.

8.3.2 Continuing research of DK-M3GP and DKA-M3GP

The research in DK-M3GP and DKA-M3GP could see multiple extensions. First and foremost, the methods are mature enough to analyse their performance on a wide variety of benchmarks. Such an analysis can confirm their potential and reveal their shortcomings. To do so, the fitness function should be revisited. To start, I would like to research the impact of using the macro weighted F_1 score from scikit-learn [82] to improve the classification method. Moreover, experimenting with changing the seed of the DT in the fitness function each generation is interesting. Finally, I think that including the functioning of the FL methods within ML pipelines is initially not relevant, and can be left out until a later stage.

Another alluring aspect of a wide-scale analysis on a large benchmark would be to see how the difference between training and test set influences the model performance. I noticed that all methods overfitted the training data, also the respected Traditional GP and M3GP methods. I am curious to know whether that is because of the datasets or because of the methods themselves.

Other improvements were mentioned in section 8.1.2, such as introducing more aggregation functions, making the aggregation functions more versatile, and allowing for more complex conditions within the aggregation functions. Finally, the prospected improvement in interpretability for DK-M3GP is not analysable using the simple ES analysis. A more thorough interpretability analysis should be implemented, possibly including the opinion of domain experts.

8.3.3 Other research questions

During this thesis, I came across multiple interesting aspect of GGGP that I would like to explore more. One of them is comparing the different individual representations of GE, SGE, and CFG-GP with Genetic Engine. As Genetic Engine is individual-representation independent, as it uses the same method for phenotype-to-genotype mapping in GE and SGE as the node generation in CFG-GP, a comparison is fair. This would extend on our work on CFG-GP advantages [47], and comparison efforts by Whigham et al. [107] and by Lourenço et al. [65].

I also found interest in analysis standards within the GP community. I already discussed the different methods of counting nodes and depth between PonyGE2 [32] and Batista et al. [9]. Another important and non-uniformly implemented method of analysis is how to measure the prediction performance of new EAs. Most research includes either prediction performance per generation or prediction performance per evaluation, as their performance is independent of the machine used. There are many examples for which these metrics suffice, for example when the time per generation and the time per evaluation do not differ across the analysed EAs. Still, an analysis of prediction performance per set time would demonstrate whether an EA performs well within a certain budget. If two methods are compared per generation, but one method runs twice as quickly, should that method not be allowed to use the extra time to improve the solution? I would like to study some publications to see how the performance prediction metric is decided upon, and whether the results are similar when the prediction performance per set time is analysed.

Finally, together with Pedro Barbosa, I plan to use Genetic Engine for multiple DNA-related problems. We want to see if we can simulate SpliceAI [49] using GP, to make interpretable models that could contribute to domain knowledge in biology. Furthermore, we aim to research whether we can generate splicing data using GE in Genetic Engine. In both works, I have an assisting role.

References

[1] S. Ahmed, M. Zhang, L. Peng, and B. Xue. Multiple feature construction for effective biomarker identification and classification using genetic programming. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO '14, page 249–256, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326629. doi: 10.1145/2576768.2598292. URL https://doi.org/10.1145/2576768.2598292. 15

[2] Q. U. Ain, B. Xue, H. Al-Sahaf, and M. Zhang. Genetic programming for multiple feature construction in skin cancer image classification. In *2019 International Conference on Image and Vision Computing New Zealand (IVCNZ)*, pages 1–6, Paris, France, 12 2019. IEEE, IEEE. doi: 10.1109/IVCNZ48456.2019.8961001. 15, 35

[3] Q. U. Ain, B. Xue, H. Al-Sahaf, and M. Zhang. Multi-tree genetic programming with a new fitness function for melanoma detection. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, pages 880–887, Paris, France, 06 2019. IEEE, IEEE. doi: 10.1109/CEC.2019.8790282. 15, 46

[4] I. Arnaldo, U.-M. O'Reilly, and K. Veeramachaneni. Building predictive models via feature synthesis. In *Proceedings of the 2015 annual conference on genetic and evolutionary computation*, pages 983–990, New York, NY, USA, 2015. Association for Computing Machinery. 3

[5] P. Arroba, J. L. Risco-Martín, M. Zapater, J. M. Moya, and J. L. Ayala. Enhancing regression models for complex systems using evolutionary techniques for feature engineering. *Journal of Grid Computing*, 13(3):409–423, 2015. 3

[6] I. Azzali, L. Vanneschi, S. Silva, I. Bakurov, and M. Giacobini. A vectorial approach to genetic programming. In *European Conference on Genetic Programming*, pages 213–227. Springer, 2019. 3, 15, 29, 77

[7] J. Bacardit, A. Brownlee, S. Cagnoni, G. Iacca, J. McCall, and D. Walker. The intersection of evolutionary computation and explainable ai. In *Genetic and Evolutionary Computation Conference: GECCO'22*. ACM, 2022. 3

[8] J. E. Batista and S. Silva. Comparative study of classifier performance using automatic feature construction by m3gp, 2022. 10, 37, 51

[9] J. E. Batista, A. I. Cabral, M. J. Vasconcelos, L. Vanneschi, and S. Silva. Improving land cover classification using genetic programming for feature construction. *Remote Sensing*, 13(9):1623, 2021. XIII, XIV, 3, 4, 10, 15, 32, 35, 36, 37, 45, 46, 47, 49, 50, 51, 52, 54, 55, 56, 63, 73, 74, 75, 78, 79, 80

[10] J. Boddu. Boom bikes demand analysis, January 2022. URL `https://www.kaggle.com/code/jayantb1019/boom-bikes-demand-analysis/data`. 5, 19

[11] V. Bolotnyy and N. Emanuel. Why do women earn less than men? evidence from bus and train operators. *Journal of Labor Economics*, 40(2):283–323, 2022. 22

[12] R. Branco, A. Branco, J. Rodrigues, and J. Silva. Shortcutted commonsense: Data spuriousness in deep learning of commonsense reasoning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 1504–1521, 2021. 1

[13] V. A. Brei et al. Machine learning in marketing: Overview, learning strategies, applications, and future developments. *Foundations and Trends® in Marketing*, 14(3):173–236, 2020. 1

[14] B. Burlacu, M. Affenzeller, M. Kommenda, S. Winkler, and G. Kronberger. Visualization of genetic lineages and inheritance information in genetic programming. In *Proceedings of the 15th annual conference companion on Genetic and evolutionary computation*, pages 1351–1358, 2013. 79

[15] B. Burlacu, G. Kronberger, and M. Kommenda. *Operon C++: An Efficient Genetic Programming Framework for Symbolic Regression*, page 1562–1570. Association for Computing Machinery, New York, NY, USA, 2020. ISBN 9781450371278. URL `https://doi.org/10.1145/3377929.3398099`. 17

[16] T. Castle and C. G. Johnson. Evolving high-level imperative program trees with strongly formed genetic programming. In A. Moraglio, S. Silva, K. Krawiec, P. Machado, and C. Cotta, editors, *Genetic Programming - 15th European Conference, EuroGP 2012, Málaga, Spain, April 11-13, 2012. Proceedings*, volume 7244 of *Lecture Notes in Computer Science*, pages 1–12, New York, USA, 2012. Springer. doi: 10.1007/978-3-642-29139-5_1. 13

[17] W. L. Cava, S. Silva, L. Vanneschi, L. Spector, and J. Moore. Genetic programming representations for multi-dimensional feature learning in biomedical classification. In *European Conference on the Applications of Evolutionary Computation*, pages 158–173. Springer, 2017. 16

[18] F. Charlier. statannotations, 2022. URL `https://github.com/trevismd/statannotations`. 49

[19] N. Cherrier, J.-P. Poli, M. Defurne, and F. Sabatié. Consistent feature construction with constrained genetic programming for experimental physics. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, pages 1650–1658, Paris, France, 2019. IEEE, IEEE. 2, 3, 15

[20] F. Chollet et al. Keras, 2015. URL `https://github.com/fchollet/keras`. 14

[21] J. Conway et al. The game of life. *Scientific American*, 223(4):4, 1970. 29

[22] S. M. J. Dahl. *TSPO: an autoML approach to time series forecasting*. PhD thesis, NOVA IMS, Lisbon, 2020. 14

[23] M. Dilhara, A. Ketkar, and D. Dig. Understanding software-2.0: a study of machine learning library usage and evolution. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(4):1–42, 2021. 2

[24] M. F. Dixon, I. Halperin, and P. Bilokon. *Machine Learning in Finance*. Springer, New York, USA, 2020. 1

[25] P. Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012. 14

[26] G. Dong and H. Liu. *Feature engineering for machine learning and data analytics*. CRC Press, Florida, USA, 2018. 2, 7, 8

[27] D. Dua and C. Graff. UCI machine learning repository, 2017. URL `http://archive.ics.uci.edu/ml`. 5, 21

[28] T. Elsken, J. H. Metzen, and F. Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019. 2

[29] G. Espada, L. Ingelse, P. Santos, P. Barbosa, and A. Fonseca. Genetic Engine, 1 2022. URL `https://github.com/alcides/GeneticEngine`. 4, 25

[30] P. G. Espejo, S. Ventura, and F. Herrera. A survey on the application of genetic programming to classification. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 40(2):121–144, 2009. 15

[31] D. Fagan, M. Fenton, and M. O'Neill. Exploring position independent initialisation in grammatical evolution. In *2016 IEEE Congress on Evolutionary Computation (CEC)*, pages 5060–5067. IEEE, 2016. 75

[32] M. Fenton, J. McDermott, D. Fagan, S. Forstenlechner, E. Hemberg, and M. O'Neill. Ponyge2: grammatical evolution in python. In P. A. N. Bosman, editor, *Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15-19, 2017, Companion Material Proceedings*, pages 1194–1201. ACM, 2017. doi: 10.1145/3067695.3082469. 25, 29, 75, 80

[33] M. Feurer, K. Eggensperger, S. Falkner, M. Lindauer, and F. Hutter. Auto-sklearn 2.0: Hands-free automl via meta-learning. *arXiv preprint arXiv:2007.04074*, 2021. 14

[34] H. Firpi, E. Goodman, and J. Echauz. On prediction of epileptic seizures by computing multiple genetic programming artificial features. In M. Keijzer, A. Tettamanzi, P. Collet, J. van Hemert, and M. Tomassini, editors, *Genetic Programming*, pages 321–330, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31989-4. 15

[35] A. Fonseca, P. Santos, and S. Silva. The usability argument for refinement typed genetic programming. In T. Bäck, M. Preuss, A. H. Deutz, H. Wang, C. Doerr, M. T. M. Emmerich, and H. Trautmann, editors, *Parallel Problem Solving from Nature - PPSN XVI - 16th International Conference, PPSN 2020, Leiden, The Netherlands, September 5-9, 2020, Proceedings, Part II*, volume 12270 of *Lecture Notes in Computer Science*, pages 18–32, New York, USA, 2020. Springer. doi: 10.1007/978-3-030-58115-2_2. 13, 25, 28

[36] A. A. Freitas. *A Review of evolutionary Algorithms for Data Mining*, pages 79–111. Springer US, Boston, MA, 2008. ISBN 978-0-387-69935-6. doi: 10.1007/978-0-387-69935-6_4. 15

[37] Y. Gil, K.-T. Yao, V. Ratnakar, D. Garijo, G. Ver Steeg, P. Szekely, R. Brekelmans, M. Kejriwal, F. Luo, and I.-H. Huang. P4ml: A phased performance-based pipeline planner for automated machine learning, 2018. 14

[38] H. Guo and A. K. Nandi. Breast cancer diagnosis using genetic programming generated feature. *Pattern Recognition*, 39(5):980–987, 2006. ISSN 0031-3203. doi: https://doi.org/10.1016/j.patcog.2005.10.001. URL `https://www.sciencedirect.com/science/article/pii/S0031320305003511`. 3, 15

[39] X. He, K. Zhao, and X. Chu. Automl: A survey of the state-of-the-art. *Knowledge-Based Systems*, 212:106622, 2021. ISSN 0950-7051. doi: https://doi.org/10.1016/j.knosys.2020.106622. URL `https://www.sciencedirect.com/science/article/pii/S0950705120307516`. 1, 14

[40] T. Helmuth, L. Spector, and J. Matheson. Solving uncompromising problems with lexicase selection. *IEEE Transactions on Evolutionary Computation*, 19(5):630–643, 2014. 52, 74

[41] T. Helmuth, N. F. McPhee, and L. Spector. Lexicase selection for program synthesis: a diversity analysis. In *Genetic Programming Theory and Practice XIII*, pages 151–167. Springer, 2016. 74, 79

[42] F. Horn, R. Pack, and M. Rieger. The autofeat python library for automated feature engineering and selection. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 111–120, New York, USA, 2019. Springer, Springer. 14

[43] F. Hutter, L. Kotthoff, and J. Vanschoren. *Automated machine learning: methods, systems, challenges*. Springer Nature, New York, 2019. 2, 14

[44] R. J. Hyndman and G. Athanasopoulos. *Forecasting: principles and practice*. OTexts, 2018. 47

[45] V. Ingalalli, S. Silva, M. Castelli, and L. Vanneschi. A multi-dimensional genetic programming approach for multi-class classification problems. In *European Conference on Genetic Programming*, pages 48–60. Springer, 2014. 16

[46] L. Ingelse, D. Branco, H. Gjoreski, T. Guerreiro, R. Bouça-Machado, J. J. Ferreira, and C. P. S. Group. Personalised gait recognition for people with neurological conditions. *Sensors*, 22(11): 3980, 2022. 1

[47] L. Ingelse, G. Espada, and A. Fonseca. Benchmarking individual representation in grammar-guided genetic programming. Technical report, EasyChair, 2022. 26, 28, 75, 79, 80

[48] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 510–520, New York, NY, United States, 2019. Association for Computing Machinery. 2

[49] K. Jaganathan, S. K. Panagiotopoulou, J. F. McRae, S. F. Darbandi, D. Knowles, Y. I. Li, J. A. Kosmicki, J. Arbelaez, W. Cui, G. B. Schwartz, et al. Predicting splicing from primary sequence with deep learning. *Cell*, 176(3):535–548, 2019. 80

[50] H. Jin, Q. Song, and X. Hu. Auto-keras: An efficient neural architecture search system, 07 2019. 14

[51] J. M. Kanter and K. Veeramachaneni. Deep feature synthesis: Towards automating data science endeavors. In *2015 IEEE International Conference on Data Science and Advanced Analytics, DSAA 2015, Paris, France, October 19-21, 2015*, pages 1–10, Paris, France, 2015. IEEE, IEEE. 14, 45

[52] D. E. Knuth. Backus normal form vs. backus naur form. *Communications of the ACM*, 7(12): 735–736, 1964. 11

[53] T. Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, 1990. 14

[54] M. Kommenda, B. Burlacu, G. Kronberger, and M. Affenzeller. Parameter identification for symbolic regression using nonlinear least squares. *Genetic Programming and Evolvable Machines*, 21 (3):471–501, 2020. 79

[55] M. Kommenda, W. La Cava, M. Majumder, F. O. De França, and M. Virgolin. Srbench competition 2022 interpretable symbolic regression for data science, 2022. URL https://cavalab.org/srbench/competition-2022/. 10

[56] J. R. Koza and J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, Cambridge, 1992. 8

[57] W. La Cava, L. Spector, and K. Danai. Epsilon-lexicase selection for regression. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 741–748, 2016. 52, 74, 79

[58] W. La Cava, T. R. Singh, J. Taggart, S. Suri, and J. H. Moore. Learning concise representations for regression by evolving networks of trees. *arXiv preprint arXiv:1807.00981*, 2018. 16

[59] W. B. Langdon, A. Al-Subaihin, and D. Clark. Measuring failed disruption propagation in genetic programming. In *Proceedings of the 2022 Genetic and Evolutionary Computation Conference (GECCO'22), Alma Rahat et al.(Eds.). Association for Computing Machinery, Boston, USA. http://dx. doi. org/10.1145/3512290.3528738*, 2022. 50, 74, 79

[60] S. Lazar. Legitimacy, authority, and the political value of explanations. *arXiv preprint arXiv:2208.08628*, 2022. 2, 10

[61] Y. Li and C. Yang. Domain knowledge based explainable feature construction method and its application in ironmaking process. *Engineering Applications of Artificial Intelligence*, 100: 104197, 2021. ISSN 0952-1976. doi: https://doi.org/10.1016/j.engappai.2021.104197. URL https://www.sciencedirect.com/science/article/pii/S0952197621000440. 15, 17

[62] Q. V. Liao, Y. Zhang, R. Luss, F. Doshi-Velez, and A. Dhurandhar. Connecting algorithmic research and usage contexts: A perspective of contextualized evaluation for explainable ai. In *Proceedings of the AAAI Conference on Human Computation and Crowdsourcing*, volume 10, pages 147–159, 2022. 10, 48

[63] N. Lourenço, F. B. Pereira, and E. Costa. Sge: a structured representation for grammatical evolution. In *International Conference on Artificial Evolution (Evolution Artificielle)*, pages 136–148. Springer, 2015. 13, 26

[64] N. Lourenço, F. Assunção, F. B. Pereira, E. Costa, and P. Machado. Structured grammatical evolution: A dynamic approach. In C. Ryan, M. O'Neill, and J. J. Collins, editors, *Handbook of Grammatical Evolution*, pages 137–161. Springer, 2018. doi: 10.1007/978-3-319-78717-6_6. URL https://doi.org/10.1007/978-3-319-78717-6_6. 26

[65] N. Lourenço, J. Ferrer, F. Pereira, and E. Costa. A comparative study of different grammar-based genetic programming approaches. pages 311–325, 03 2017. ISBN 978-3-319-55695-6. doi: 10.1007/978-3-319-55696-3_20. 26, 80

[66] R. I. Mckay, N. X. Hoai, P. A. Whigham, Y. Shan, and M. O'neill. Grammar-based genetic programming: a survey. *Genetic Programming and Evolvable Machines*, 11(3-4):365–396, 2010. 26

[67] E. Medvet, A. Bartoli, A. De Lorenzo, and F. Tarlao. Designing automatically a representation for grammatical evolution. *Genetic Programming and Evolvable Machines*, 20(1):37–65, 2019. 13

[68] J. Mégane, N. Lourenço, and P. Machado. Probabilistic grammatical evolution. In T. Hu, N. Lourenço, and E. Medvet, editors, *Genetic Programming*, pages 198–213, Cham, 2021. Springer International Publishing. ISBN 978-3-030-72812-0. 28

[69] Microsoft. Neural Network Intelligence, 1 2021. URL https://github.com/microsoft/nni. 14

[70] D. J. Montana. Strongly typed genetic programming. *Evol. Comput.*, 3(2):199–230, 1995. doi: 10.1162/evco.1995.3.2.199. 4, 13

[71] M. Muharram and G. D. Smith. Evolutionary constructive induction. *IEEE transactions on knowledge and data engineering*, 17(11):1518–1528, 2005. 15

[72] M. A. Muharram and G. D. Smith. Evolutionary feature construction using information gain and gini index. In M. Keijzer, U.-M. O'Reilly, S. Lucas, E. Costa, and T. Soule, editors, *Genetic Programming*, pages 379–388, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24650-3. 15

[73] L. Muñoz, S. Silva, and L. Trujillo. M3gp–multiclass classification with gp. In *European Conference on Genetic Programming*, pages 78–91, New York, USA, 2015. Springer. 4, 16, 35

[74] K. Neshatian, M. Zhang, and P. Andreae. A filter approach to multiple feature construction for symbolic learning classifiers using genetic programming. *IEEE Transactions on Evolutionary Computation*, 16(5):645–661, 2012. 15

[75] M. T. Nuseir, B. H. Al Kurdi, M. T. Alshurideh, and H. M. Alzoubi. Gender discrimination at workplace: Do artificial intelligence (ai) and machine learning (ml) have opinions about it. In *The International Conference on Artificial Intelligence and Computer Vision*, pages 301–316. Springer, 2021. 22

[76] R. S. Olson and J. H. Moore. *TPOT: A Tree-Based Pipeline Optimization Tool for Automating Machine Learning*, pages 151–160. Springer International Publishing, Cham, 2019. ISBN 978-3-030-05318-5. doi: 10.1007/978-3-030-05318-5_8. URL https://doi.org/10.1007/978-3-030-05318-5_8. 14

[77] R. S. Olson, W. La Cava, P. Orzechowski, R. J. Urbanowicz, and J. H. Moore. Pmlb: a large benchmark suite for machine learning evaluation and comparison. *BioData Mining*, 10(1):36, Dec 2017. ISSN 1756-0381. doi: 10.1186/s13040-017-0154-4. URL `https://doi.org/10.1186/s13040-017-0154-4`. 21

[78] M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, 2001. 26

[79] M. O'Neill, L. Vanneschi, S. M. Gustafson, and W. Banzhaf. Open issues in genetic programming. *Genet. Program. Evolvable Mach.*, 11(3-4):339–363, 2010. doi: 10.1007/s10710-010-9113-2. URL `https://doi.org/10.1007/s10710-010-9113-2`. 25

[80] F. E. Otero, M. M. Silva, A. A. Freitas, and J. C. Nievola. Genetic programming for attribute construction in data mining. In *European Conference on Genetic Programming*, pages 384–393, New York, USA, 2003. Springer. 15, 26

[81] T. pandas development team. pandas-dev/pandas: Pandas, Feb. 2020. URL `https://doi.org/10.5281/zenodo.3509134`. 42

[82] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python, 2011. 14, 29, 45, 78, 79

[83] R. Poli, W. B. Langdon, and N. F. McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, Morrisville, 2008. ISBN 1409200736. 8

[84] F. Poursabzi-Sangdeh, D. G. Goldstein, J. M. Hofman, J. W. Wortman Vaughan, and H. Wallach. Manipulating and measuring model interpretability. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450380966. doi: 10.1145/3411764.3445315. URL `https://doi.org/10.1145/3411764.3445315`. 76

[85] E. Real, C. Liang, D. R. So, and Q. V. Le. Automl-zero: Evolving machine learning algorithms from scratch. *CoRR*, abs/2003.03384, 2020. URL `https://arxiv.org/abs/2003.03384`. 15

[86] F. Rothlauf and M. Oetzel. On the locality of grammatical evolution. In *European conference on genetic programming*, pages 320–330. Springer, 2006. 26

[87] C. Ryan, J. Collins, and M. O. Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 1391, pages 83–96. Springer Berlin Heidelberg, 1998. ISBN 3540643605. doi: 10.1007/BFb0055930. 13, 26

[88] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison. Hidden technical debt in machine learning systems. *Advances in neural information processing systems*, 28:2503–2511, 2015. 2

[89] M. Sipper and J. H. Moore. Genetic programming theory and practice: a fifteen-year trajectory. *Genet. Program. Evolvable Mach.*, 21(1-2):169–179, 2020. doi: 10.1007/s10710-019-09353-5. URL `https://doi.org/10.1007/s10710-019-09353-5`. 25

[90] M. G. Smith and L. Bull. Feature construction and selection using genetic programming and a genetic algorithm. In C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa, editors, *Genetic Programming*, pages 229–237, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-36599-0. 14, 15

[91] D. Sobania, M. Briesch, and F. Rothlauf. Choose your programming copilot: A comparison of the program synthesis performance of github copilot and genetic programming. *arXiv preprint arXiv:2111.07875*, 2021. 4, 29

[92] P. Sondhi. Feature construction methods: a survey. *sifaka. cs. uiuc. edu*, 69:70–71, 2009. 14

[93] H. Song. *AutoFE: efficient and robust automated feature engineering*. PhD thesis, Massachusetts Institute of Technology, 2018. 16

[94] F. Sovrano, S. Sapienza, M. Palmirani, and F. Vitali. Metrics, explainability and the european ai act proposal. *J*, 5(1):126–138, 2022. 2, 10

[95] L. Spector. Assessment of problem modality by differential performance of lexicase selection in genetic programming: a preliminary report. In *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*, pages 401–408, 2012. 16

[96] H. Suresh, S. R. Gomez, K. K. Nam, and A. Satyanarayan. Beyond expertise and roles: A framework to characterize the stakeholders of interpretable machine learning and their needs. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–16, 2021. 10

[97] I. M. A. O. Swesi and A. A. Bakar. Recent developments on evolutionary computation techniques to feature construction. In *Asian Conference on Intelligent Information and Database Systems*, pages 109–122, New York, USA, 2019. Springer. 2, 14

[98] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, page 847–855, New York, NY, USA, 2013. Association for Computing Machinery.

ISBN 9781450321747. doi: 10.1145/2487575.2487629. URL https://doi.org/10.1145/2487575.2487629. 14

[99] F. Thung, S. Wang, D. Lo, and L. Jiang. An empirical study of bugs in machine learning systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pages 271–280, 1730 Massachusetts Ave., NW Washington, DCUnited States, 2012. IEEE, IEEE Computer Society. 1

[100] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996. 14

[101] B. Tran, B. Xue, and M. Zhang. Genetic programming for feature construction and selection in classification on high-dimensional data. *Memetic Computing*, 8(1):3–15, 2016. 15

[102] B. Tran, B. Xue, and M. Zhang. Class dependent multiple feature construction using genetic programming for high-dimensional data. In *Australasian Joint Conference on Artificial Intelligence*, pages 182–194, New York, USA, 2017. Springer, Springer. 3, 17

[103] B. Tran, B. Xue, and M. Zhang. Genetic programming for multiple-feature construction on high-dimensional classification. *Pattern Recognition*, 93:404–417, 2019. 15

[104] M. Virgolin, T. Alderliesten, C. Witteveen, and P. A. N. Bosman. Improving model-based genetic programming for symbolic regression of small expressions. *Evolutionary Computation*, 29(2): 211–237, 2021. 17

[105] M. Virgolin, E. Medvet, T. Alderliesten, and P. A. Bosman. Less is more: A call to focus on simpler models in genetic programming for interpretable machine learning. *arXiv preprint arXiv:2204.02046*, 2022. 10

[106] P. A. Whigham. Search bias, language bias, and genetic programming. *Genetic Programming*, 1996:230–237, 1996. 4, 13

[107] P. A. Whigham, G. Dick, J. Maclaurin, and C. A. Owen. Examining the" best of both worlds" of grammatical evolution. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1111–1118, 2015. 26, 80

[108] P. A. Whigham et al. Grammatically-based genetic programming. In *Proceedings of the workshop on genetic programming: from theory to real-world applications*, volume 16, pages 33–41. Citeseer, 1995. 13, 26

[109] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945. ISSN 00994987. URL http://www.jstor.org/stable/3001968. 48

[110] S. Wold, K. Esbensen, and P. Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987. 2, 14

[111] R. Zhang, W. Xiao, H. Zhang, Y. Liu, H. Lin, and M. Yang. An empirical study on program failures of deep learning jobs. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1159–1170, 1730 Massachusetts Ave., NW Washington, DCUnited States, 2020. IEEE, IEEE Computer Society. 2

[112] Q. Zhou, F. Liao, C. Mou, and P. Wang. Measuring interpretability for different types of machine learning models. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 295–308. Springer, 2018. 76

[113] M. Zinkevich. Rules of machine learning: Best practices for ml engineering, 2017. URL `https://developers.google.com/machine-learning/guides/rules-of-ml`. 1

[114] J. Zou and L. Schiebinger. Ai can be sexist and racist—it's time to make it fair, 2018. 2, 10

[115] A. Zytek, I. Arnaldo, D. Liu, L. Berti-Equille, and K. Veeramachaneni. The need for interpretable features: Motivation and taxonomy. *arXiv preprint arXiv:2202.11748*, 2022. 10

[116] M.-A. Zöller and M. F. Huber. Benchmark and survey of automated machine learning frameworks, 2021. 14

Appendix A

Expressions of evolved solutions

Below I list the complete evolved feature sets by M3GP (listing A.1) and DK-M3GP (listing A.2) for seed 0. I add these as an appendix to section 7.4.3, in which I highlight the main advantage of using DK-M3GP in terms of interpretability; tree-like feature formats. The first seed (0) was chosen to avoid bias towards selecting a seed that benefits our approach.

```
workingday + temp - (weekday / season) - (mnth * temp) + windspeed + (((season * atemp) + (atemp
        ↪  / yr)) / mnth)

temp - ((mnth + holiday + weathersit - season) / ((mnth / weathersit) * season * mnth))

((mnth * temp * (workingday + yr)) + ((windspeed + mnth) / (season * windspeed))) * ((holiday -
        ↪ weathersit - holiday - atemp) / ((season / yr) + (hum * temp)))

holiday / (((temp - yr) / weathersit) / ((holiday * holiday) + atemp))

season * (season / (holiday - hum + atemp - weekday))

(((season + workingday) / (weekday * atemp)) + weathersit - mnth + holiday) * (season - mnth + (
        ↪ season * mnth) - windspeed)

((yr + atemp) / season) + ((workingday - weekday) / (mnth - atemp)) + hum

(windspeed / ((weathersit + holiday) * mnth)) + ((windspeed / windspeed) * (atemp + holiday)) -
        ↪ weathersit + hum + windspeed

yr / (((yr * mnth) / (windspeed * windspeed)) * ((atemp / atemp) - windspeed))

(season - weekday - (temp * hum) - ((season * workingday) + atemp - yr)) / (((weathersit +
        ↪ windspeed) / (atemp * weekday)) / (weathersit + atemp - (atemp * weathersit)))

(((hum / windspeed) / (holiday * temp)) / ((workingday / weekday) * (windspeed / season))) / hum

windspeed

((yr * mnth * (season + workingday)) / (mnth + holiday - (season * holiday))) / (yr - weekday +
        ↪ workingday + workingday + temp)

(((mnth * temp) + (mnth / hum)) * ((windspeed / atemp) / (mnth + weathersit))) / ((season + temp
        ↪  - season - temp) / ((weekday - weathersit) * season * temp))

atemp - weekday
```

Listing A.1: Evolved feature set by M3GP for seed 0.

```
(((hum / hum) - (temp * atemp)) / ((hum / atemp) / (windspeed + atemp))) / (windspeed - ((windspeed / atemp) / (atemp / windspeed)))

if (weathersit inbetween (Clear,Misty)):
        temp * atemp
else:
        windspeed

(((windspeed / hum) * (windspeed / atemp)) - (windspeed * (hum / atemp))) / (atemp - temp - windspeed - atemp)

(atemp * (windspeed / atemp)) - (((temp - atemp) / (hum - hum)) -
        (if (weekday inbetween (Sunday,Wednesday)):
                atemp + windspeed
            else:
                temp - windspeed))

((temp - atemp) / (windspeed * atemp + windspeed + temp)) * (temp * atemp * hum *
        (if (season inbetween (spring,summer)):
                hum + temp
            else:
                hum + windspeed))

if (workingday != 0):
    temp * atemp + temp
else:
    atemp - temp - hum - atemp + hum

(if (yr inbetween (0,1)):
        (temp * hum)
    else:
        (atemp - windspeed) / ((temp / atemp) * (windspeed / temp)))
/ ((windspeed - atemp + windspeed) /
        (if (weathersit inbetween (Clear,Misty)):
                temp + hum
            else:
                atemp / atemp))

if (mnth == December):
    if (mnth inbetween (October,November)):
            atemp / hum
    else:
            temp * temp
else:
    atemp + atemp
```

Listing A.2: Evolved feature set by DK-M3GP for seed 0.