Evolving Regular Expression Features for Text Classification with Genetic Programming

by

Robin Bakker
10548017

December 5, 2018

36 EC
March - December 2018

 Supervisor:  Assessor:
Maarten Marx    Fabian Jansen
Abstract

Text classification algorithms often rely on vocabulary counters like bag-of-words or character n-grams to represent text as a vector appropriate for use in machine learning algorithms. In this work, automatically generated regular expressions are proposed as an alternative feature set. The proposed algorithm uses genetic programming to evolve a set of regular expression features based on labeled text data and train a classifier in an end-to-end fashion. Though a comparison of the generated features and traditional text features indicates a classifier using generated features is not able to make better predictions, the generated features are able to capture patterns that cannot be found with the traditional features. As a result, a classifier combining traditional methods with generated features is able to improve significantly.
## Contents

1 Introduction 5

2 Theoretical Background 7
   2.1 Regular Expressions .............................................. 7
      2.1.1 Syntax ......................................................... 7
      2.1.2 Catastrophic backtracking .................................. 8
   2.2 Evolutionary Algorithms ........................................... 9
      2.2.1 Individuals .................................................... 9
      2.2.2 Variation Operators ......................................... 10
      2.2.3 Survivor Selection .......................................... 11
   2.3 Genetic Programming ............................................... 11
      2.3.1 Individuals ................................................... 11
      2.3.2 Variation Operators ......................................... 12

3 Related Work 15
   3.1 Regex Generation .................................................. 15
   3.2 Feature Extraction ............................................... 16
   3.3 Classification of Text ............................................ 16

4 End-To-End Classification of Text Data 17
   4.1 Evolving Regular Expressions ...................................... 17
      4.1.1 Individuals ................................................... 18
      4.1.2 Population Initialization .................................... 21
      4.1.3 Fitness Calculation ......................................... 21
      4.1.4 Offspring Generation ....................................... 24
      4.1.5 Survivor Selection .......................................... 28
      4.1.6 Separate-and-Conquer ...................................... 29
      4.1.7 Distributing with Multiprocessing .......................... 29

5 Experiments 31
   5.1 Datasets ........................................................... 31
      5.1.1 Regex 1 .......................................................... 31
      5.1.2 Regex 2 .......................................................... 32
      5.1.3 Spam ............................................................ 33
      5.1.4 PPI .............................................................. 34
   5.2 Experimental Setup ................................................ 36
      5.2.1 Q1: Comparing Regex Features and Traditional Features .. 36
      5.2.2 Q2: Combining Regex Features With Traditional Features . 37
      5.2.3 Q3: Comparing Automated and Hand-made Regex Features . 37

6 Results 39
   6.1 Comparing regular expressions and traditional features .......... 39
      6.1.1 Regex 1 .......................................................... 39
      6.1.2 Regex 2 .......................................................... 41
      6.1.3 Spam ............................................................ 43
      6.1.4 PPI .............................................................. 46
   6.2 Combining regular expressions with traditional features .......... 48
6.2.1 Spam ................................................................. 48
6.2.2 PPI ................................................................. 52
6.3 Comparing manually and automatically generated features ........... 56
  6.3.1 Stacked Word Features ....................................... 57
  6.3.2 Stacked Character Features ................................. 58

7 Conclusion ............................................................. 61
  7.1 Summary ........................................................ 61
  7.2 Future Work ...................................................... 62
    7.2.1 Improvement of the Genetic Programming Algorithm ...... 62
    7.2.2 Experimentation on Additional Datasets .................. 62
    7.2.3 Using Traditional Features Rather than Predictions ...... 62
Chapter 1

Introduction

Banks have vast databases of information on their account holders. However, as this information is private, it is not shared between banks. During a transaction between users of different banks, only a small portion of this information is exchanged. The wholesale banking division of ING aims to increase its user base with accounts of companies encountered in transactions. However, many of the accounts in transactions belong to private individuals (PI’s) and are out of scope for wholesale banking.

PPI (Possible Private Individual detection) is an algorithm created by ING to filter PI accounts from the data by automatically classifying an account as a private individual or business based on the limited information available in the transaction, i.e., the name belonging to the account. A variety of features are extracted from the account name, such as a Bag-of-words vector representation, length features, and text pattern features based on manually constructed regular expressions. However, creation of a well performing regular expression is no straightforward task, especially for large datasets, as it is strenuous to inspect the behavior for all examples.

Various tools and algorithms exist to aid users in creating well performing regular expressions, each with their own approach:

- SEER is a tool that proposes extraction rules, based on the data, which the user can add to the expression[16].
- Regex synthesis algorithms are capable of generating regular expressions from data without further input from the user[14][3]. In recent years, methods using genetic programming(gp) to drive the search have seen significant improvements in performance[4][6][18].
- Regex improvement algorithms take an initial regular expression and attempt to improve their performance by evaluating variations of the expression[22][11].

However, these methods are created for string extraction tasks and assume the desired extraction is provided as ground truth. PPI, on the other hand, is built to solve a classification problem, for which information on desired matching is not available. In other words, it is not known what part of the name is indicative of the correct class. In this sense, private individual detection is only a specific example of a broader type of classification problem for natural language texts. Yet another example of classification for text is the automatic detection of Spam messages. The use of genetic programming to generate a regular expression has been proposed by Basto-Fernandes et al. in their literature study on Spam detection[7]. Ruano-Ordás et al. recently implemented an algorithm for automatic regex generation for Spam[26], though this method did not make use of genetic programming. Moreover, as the approach was tailored more to Spam detection, the algorithm became less suitable for use in the general text classification problem. This observation leads to the formation of the following research question:

Can an end-to-end text classifier benefit from features based on regular expressions that have been learned from the data automatically with genetic programming?
From this research question several sub-questions have been derived:

1. Can a classifier with learned regex features outperform a similar classifier with traditional text features?
2. Does the addition of learned regex features to traditional features increase predictive power of a classifier?
3. Is it possible for an algorithm to construct regular expressions of higher quality than those made by a human expert?

In this paper, we propose a new end-to-end algorithm for the classification of text as shown in Figure 1.1. The raw data is fed to a genetic programming algorithm that evolves regular expressions corresponding to patterns in the data. These regular expressions are then matched against the data to transform the raw data to a new feature space based. The resulting match features can be used by any out-of-the-box classifier. For simplicity, a logistic regression model was chosen to evaluate the results. As the quality of extracted features is automatically evaluated within the genetic programming algorithm, only regular expressions giving indicative features are maintained and used to train the classifier. This results in a classification algorithm with features created directly from the available data and a set of common patterns in the data described by regular expressions as a byproduct.

The remainder of this work is structured as follows: Chapter 2 will provide the reader with background knowledge on topics crucial to this research. Chapter 3 lists related work in the fields of regex generation, feature selection, and more. A detailed description of the genetic programming algorithm is given in chapter 4. Chapter 5 contains the experimental setup. Chapter 6 shows the results obtained in the experiments. Finally, chapter 7 highlights the conclusions that can be drawn from these results.
Chapter 2

Theoretical Background

This chapter provides fundamental information on the topics that will be discussed in the rest of this work. Section 2.1 covers the uses, properties, and potential problems of regular expressions. Section 2.2 describes the process followed by Evolutionary Algorithms with which the search space is explored. Lastly, section 2.3 highlights the deviations made by Genetic Programming from the standard evolutionary algorithm.

2.1 Regular Expressions

Regular expressions (regexes) are a tool for finding, extracting, and replacing substrings in text that have existed for several decades. Due to their versatility and power, regexes are a common occurrence in many code projects[9]. A regular expression is comprised of a string of literal and special characters that describes a pattern in text. This string is interpreted by a search engine that attempts to locate pieces of text structured as described by the pattern. Text found by the search engine is said to "match" the regular expression.

Assume one needs to quickly locate all the mentioned days of the week within a large text file. An automated search can be performed by constructing a regular expression capturing some structure within the desired strings. Note that the name of every day of the week ends in 'day', with an unknown amount of letters in front. This pattern can be exploited with the regex '.+day', consisting of the character class '.', the operator '+', and the literal string 'day', as follows:

- . matches any character
- + repeats the previous character at least once
- day matches day

Monday, Tuesday, today

As can be seen, the designed regular expression is capable of extracting the desired strings. However, the difficulty of defining good regular expressions also becomes apparent, as the previous regex matches unwanted strings like 'today' as well.

2.1.1 Syntax

A regular expression string generally consists of a combination of literal characters and special characters. The special characters can be categorized as character classes and operators. Though there exist multiple syntaxes for these special characters, the Perl syntax has become standard for regexes and will be discussed here.

Character classes

Character classes are a concise way of representing a choice between multiple characters. These classes are:
Operators

Operators provide additional flexibility to the patterns described by regular expressions. Though a large amount of operators is available, this work uses only common operators to limit model complexity. Below, these operators are explained in further detail.

- \[\] A set of possible characters. `[abc]` matches strings 'a', 'b', and 'c'
- \[\] A range of characters. `[A-Z]` matches any uppercase alphabetic character
- \[^\] Complement of a character set. `[^abc]` matches any character but 'a', 'b', and 'c'
- + Kleene plus quantifier. `x+` matches one or more x's
- * Kleene star quantifier. `x*` matches zero or more x's
- ? Question mark quantifier. `x?` matches zero or one x's
- { } Exact match quantifier. `x{3}` matches exactly 3 x's
- {,} Range match quantifier. `x{2,4}` matches between 2 and 4 x's inclusive
- ^ Start of string assertion. ^ matches the first character in the string
- $ End of string assertion. $ matches the last character in the string
- | Or operator. cat|dog matches if either 'cat' or 'dog' matches
- \b Word boundary. i.e. `(\w|\w$|\W\w|\w\W)`. `\w\b` matches the last character of a word
- \B The reverse of \b. matches any but the first and last character of a word
- ( ) Capturing group. The phrase inside the brackets can be referenced with \r\r
  r being the number of the group. `D(\w)\1r` matches 'Door' but does not match 'Dear'

Table 2.1: List of regex operators used in this work

2.1.2 Catastrophic backtracking

Though evaluating regexes is generally fast, some deceptively simple regexes can take a long time to complete their search. Regular expressions that fall into this category are said to suffer from catastrophic backtracking.

Catastrophic backtracking is a problem in regular expressions leading to extensive evaluation time, which in turn causes the program using the regular expression to halt until the evaluation is finished. This weakness can even be exploited to cause denial-of-service attacks and should therefore be avoided if possible[30]. Catastrophic backtracking occurs when a regex tries and fails to match a string in many possible ways, usually due to nested quantifiers. An example of catastrophic backtracking can be seen by evaluating the strings 'xxx' and 'xxxy' with the following regex:

```
(\p{P}\p{P}\p{P})+\p{P}
```

For this regular expression, 'xxx' causes catastrophic backtracking, whereas 'xxxy' does not. As the kleene plus is greedy, meaning it will match as many characters as possible, the regex will fail once for 'xxxy'. First, \p{P} matches 'xxx', leaving no x's for \p{P} and causing the search engine to backtrack. Next, \p{P} gives up one x, leaving 'xx' for \p{P} and 'x' for \p{P}, which come together as 'xxx' in \p{P}. The engine now matches 'y' with y and the match is a success. However, if the same procedure is followed for 'xxx', many more possibilities have to be explored before the match can fail. After \p{P} has given up one x, which is matched by \p{P}, the match fails as no y is found. Now \p{P} gives up another x, \p{P} matches 'xx' and the match fails again on y. This time \p{P} can backtrack.
and give up one x, still resulting in a failed match. This process continues until no more x's can be given up by either kleene plus. Thus, the amount of steps necessary to assure the string cannot be matched increases exponentially with the length of the string.

Though catastrophic backtracking can be avoided if the maker is aware of this issue, it is difficult to detect whether a constructed regular expression suffers from catastrophic backtracking beforehand. As the regular expressions in this work are generated automatically, patterns leading to catastrophic backtracking could occur and slow down the algorithm. There are, however, search engines that do not depend on backtracking to evaluate regular expressions. RE2[1] is a regex search engine developed by Google that guarantees time performance linear to the length of the regex. As RE2 relies on multi-threaded finite automata rather than backtracking, operators requiring backtracking, e.g. backreferences of capturing groups, cannot be supported. Therefore, our algorithm uses RE2 by default, falling back on the Python search engine, decorated with a timeout to avoid catastrophic backtracking, when backtracking is necessary.

### 2.2 Evolutionary Algorithms

Evolutionary algorithms are guided random search algorithms inspired by the workings of evolution in nature[13]. These algorithms do not require any domain knowledge and are therefore applicable to many different problems. Evolutionary Algorithms work by maintaining a population of individuals, where each individual represents a possible solution to the stated problem. Individuals have a fitness that describes how well it is adapted to its environment, i.e. the problem. New individuals are added to the population through crossover between well adapted individuals and through random mutations. Finally, the need for individuals to adapt to the environment is achieved through survivor selection. These steps are repeated until the stopping criterion is met. A visual representation of this process is shown in Figure 2.1.

![Flowchart of an evolutionary algorithm](https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_fundamentals)

#### 2.2.1 Individuals

In evolutionary algorithms, an individual is a candidate solution to the search problem. Individuals are made up of a phenotype and a genotype, which represent the individual in separate spaces. The phenotype represents the individual in the solution space, meaning it is not only necessary to
evaluate performance but is also the most interesting representation for the user. The genotype, on the other hand, represents the individual in the search space and enables the algorithm to perform evolutionary operations, such as crossover and mutation, on the individual. A variety of genotype encodings exist, including bit-strings, arrays, sets, etc. Generally, each type will work, though some representations are more sensible, depending on the problem. Though in most genotype structures there can exist multiple encodings for a single phenotype, a genotype representation will always map to a single phenotype representation. In other words, though the phenotype can always be derived from the genotype, the opposite is not true, as there are many possible genotype representations resulting in the same phenotype.

The fitness of an individual indicates how likely the individual is to create offspring and survive. Fitness is determined by evaluating the individual’s performance with a fitness function. This fitness function applies the phenotype to the problem and rewards desired behavior. For instance, if the phenotype represents the weights given to a set of features in a classifier, the fitness function could perform classification on test data and return the accuracy as fitness.

2.2.2 Variation Operators

Variation operators, i.e. crossover and mutation, are operations that push the algorithm towards exploration of the search space. Every iteration, new individuals, called offspring, are generated from individuals in the current population, called parents. Parent selection schemes are stochastic methods that select individuals from the population with higher preference for fitter individuals. Once two parent individuals have been selected, a recombination of the parent genes, called crossover, is performed to create offspring. Afterwards, slight variations are made to the genes of the offspring through random mutations.

Crossover

Crossover creates offspring by recombining the genes of selected parents. Many crossover methods exist, depending on the chosen genotype structure. A common crossover operation that can be applied to bit-strings, among others, is one-point-crossover. In one-point-crossover, an index of the bit-string, named the crossover point, is selected at random. Parts of the gene following the crossover point are then exchanged by the parents, as demonstrated in Figure 2.3.

Figure 2.2: The genotype-phenotype mapping of an individual\textsuperscript{2}

\textsuperscript{2}https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_fundamentals
Mutation

Mutation introduces small changes in the genes of an individual. Generally, the mutation operation is performed on offspring created by the crossover operation. Like crossover, mutation can be performed in many ways, depending on the genotype. In bit-strings, mutation is generally performed by a bit-flip, where the allele(value) of a bit can switch between 0 and 1 with a given probability \( p \). Usually, the value of \( p \) is such that on average one allele in the genome is mutated. An example of a bit-flip mutation can be seen in Figure 2.4.

![Figure 2.4: Visual representation of a bit-flip mutation](https://www.researchgate.net/figure/Example-of-a-gene-value-flip-On-top-is-the-original-solution-on-bottom-is-the-mutated_fig5_276170294)

2.2.3 Survivor Selection

Once all offspring has been created and evaluated, the final step in the iteration is survivor selection, which reduces the population to a predetermined number of individuals. To avoid the loss of good individuals, survivor selection is the only step in evolutionary algorithms that is completely deterministic. Individuals are sorted by their fitness and only the strongest are kept. Those remaining after survivor selection form the new population and become possible parents in the next iteration if the stopping criterion has not yet been met.

2.3 Genetic Programming

Genetic programming is a type of evolutionary algorithm. Though it largely follows the approach of evolutionary algorithms, there are several significant differences between these algorithms, caused primarily by the difference of the search problems they are applied to. As genetic programming is often used for problems that require a solution of unknown length such as mathematical equations, strings, or program code, a distinct genotype representation is required for the individuals. This, in turn, causes a change in the evolutionary operators.

2.3.1 Individuals

Genotypes used in evolutionary algorithms are not well suited for problems addressed with genetic programming due to their inherent static structure. Instead, a more flexible structure is required,
for which tree structures have generally been employed. To decode the genotype tree into a phenotype string, the nodes of the tree can be evaluated recursively. Trees are built from a combination of terminal and non-terminal nodes. Terminal nodes represent the regular characters in the language, such as the numeric characters in math, whereas non-terminal nodes, also called operators, represent functions on the terminal nodes, e.g., the multiplication sign or square root in equations.

2.3.2 Variation Operators

Like other evolutionary algorithms, genetic programming uses crossover and mutation to explore the search space. However, rather than performing crossover and mutation in succession, as is common in evolutionary algorithms, a choice is made between the two operations at random as indicated in Figure 2.5. If crossover is chosen, two parents are selected and two new individuals are created with recombination. In the case of mutation, a single parent is selected and one new individual is created.

Figure 2.5: Offspring creation scheme for evolutionary algorithms vs genetic programming

Crossover

As genetic programming uses tree structure genotypes, the crossover operation as described for evolutionary algorithms is not applicable. Instead, crossover in genetic programming is achieved by exchanging subtrees between parents as shown in Figure 2.6. A subtree is selected for each parent by randomly selecting a node in the tree. This node, together with its descendants forms the subtree that will be exchanged. Crossover is then performed by swapping the positions of the two subtrees, resulting in two new individuals.

---

5 A.E. Eiben and J.E. Smith, Introduction to Evolutionary Computing, Chapter 6 Genetic Programming
Figure 2.6: Crossover in tree structures. First, a subtree is selected randomly for each parent. Next, the subtrees swap their positions.\(^7\)

**Mutation**

Mutation only accounts for a small percentage of created offspring in genetic programming. Some early work has suggested mutation can be removed from the algorithm completely, as the crossover operation already provides a lot of variation. However, recent approaches have settled on using mutation sparingly, still relying on crossover for most of the variation[13].

As mutation in genetic programming serves to create new offspring, rather than introducing variation offspring created by crossover, a single parent is selected. To create a new individual, a node in the parent’s tree is replaced, as shown in Figure 2.7. A node is selected at a random point in the tree. Additionally, a new tree is generated randomly. Finally, the subtree with the mutation point at the root is replaced by the randomly generated tree.

Figure 2.7: Mutation in tree structures. A randomly generated tree is inserted in place of the node at the random mutation point.\(^9\)

---

\(^7\)https://www.semanticscholar.org/paper/Deterministic-Crossover-Based-on-Target-Semantics-Hara-Kushida/ecd815e50f1a1f5d2be02bb107426e80e210c2d/figure/2

\(^9\)https://dces.essex.ac.uk/staff/rpoli/gp-field-guide/24RecombinationandMutation
Chapter 3

Related Work

In this chapter, the current work is put into perspective with other research that has been performed in similar fields. A large part of this work is based on the automatic generation of regular expressions from data. Section 3.1 is therefore dedicated to the progress of research on automatic regex generation algorithms. However, unlike the methods discussed in Section 3.1, the regular expressions created by our algorithm are not used directly for classification or extraction of text, but rather as a means of transforming the data to a feature space that can be used by other classifiers. As this approach is comparable to feature extraction, section 3.2 highlights some works that also employ genetic programming for feature extraction tasks. Lastly, section 3.3 explores work on text classification using more traditional methods.

3.1 Regex Generation

Automatic regex generation gained attention when Li et al. noted that the manual construction of a regular expression is a tedious task requiring domain knowledge, yet very few algorithms existed to automate this process. Thus, they proposed Re-LIE, a hill climbing search algorithm that automatically improves a given regular expression[22]. Though this approach could find a good regular expression, it still required the creation of an initial regex by a human expert. Several approaches for the generation of regular expression without providing an initial regex were explored, such as grammatical evolution by Cetinkaya[8], sequence alignment by Wang et al.[32], and graph evolution by Gonzalez et al.[14].

Another algorithm aiming to overcome the need for an initial regex was proposed by Bartoli et al. This algorithm, based on genetic programming, automatically evolves a population of regular expressions to extract desired strings from the data[2]. Fitness of the individuals is determined by the Levenshtein distance between the label, indicating the desired string, and the string extracted by the regex. Additionally, to promote short expressions, a penalty is added based on the length of the string. Results were highly promising, even when the set of regex operators was limited and a small fraction of the data was used for training.

The algorithm was further improved and applied to many more problems in later work[3][4][6]. A notable improvement has been the introduction of the separate-and-conquer technique[5]. Rather than allowing the algorithm to freely use the OR operator (‘|’) by adding it to the set of non-terminals, the separate-and-conquer scheme governs at what time an OR operator is added. With separate-and-conquer, multiple regular expressions are evolved, which are later appended with the OR operator to form one large expression. To avoid learning and appending similar expressions, matched positive examples are discarded from the data after a regular expression has been learned. Thus, the algorithm expands the full regular expression with new expressions capturing parts of the data missed by the other regexes. Through the introduction of the separate-and-conquer approach, alongside a multi-layered optimization criterion, optimizing precision before recall and length, and the addition of more regex operators to the non-terminal set, the algorithm was able to outperform the author's previous approaches.

An evolutionary algorithm for classification of Spam messages with a generated regular expres-
sion was proposed by Conrad some years before Bartoli et al. showed the advantage of genetic programming for regex generation[12]. Though evolution was successful, the approach had some issues regarding algorithm speed, memory usage, and efficiency of found regular expressions. Ruano-Ordás et al. aimed to improve the algorithm proposed by Conrad by addressing these problems[26]. Since mistakenly classifying real emails as Spam is a costly mistake, the authors focused on avoiding false positive errors. To do this, the original fitness function was updated to give higher scores to individuals avoiding false positives. Furthermore, changes were made to the original evolutionary operators to promote diversity in the population. Though results indicated false positives were successfully avoided, overall performance in terms of accuracy decreased.

3.2 Feature Extraction

Feature extraction is the process of transforming raw data into a set of features for use in a machine learning algorithm. Though much research has been done on methods for feature extraction, those using genetic programming are of particular interest for this work.

Feature extraction aims to reduce redundancy in the data by creating a new set of informative features that can be used in place of the raw data. Examples of these types of data are time series, images, and text. Guo et al. showed how genetic programming can be used to transform time series data into a set of features that could be used by machine learning classifiers like the SVM and Neural Network[15]. More recently, Harvey & Todd used genetic programming to extract features for numeric sequence classification[17]. Features created by this algorithm were not only close to optimal, but also compact and easily interpretable by humans. Image classification is another interesting field for genetic programming based feature extraction, as shown by Shao et al.[28]. In their paper, genetic programming is used to transform raw images into a significantly smaller feature vector. Operators for this problem consists of several image filters, such as the Gaussian or Laplace filter, and pixel arithmetics, such as pixel addition or division. To score an individual’s fitness, the fitness function evaluates the classification of an SVM model on test data with the feature set created by the individual. The authors also note that the SVM used in their work can be exchanged for any classifiers, such as KNN. The authors further note that, though evaluating the classification score for every individual during training is time consuming, the transformation of the data during further use is actually quite fast.

3.3 Classification of Text

The classification of sentences or documents is no new problem. SVM’s have been used to categorize documents by performing a binary classification for each possible category, predicting whether the text belongs to the category or not[19]. Since the SVM requires a numeric feature vector rather than text, a bag-of-words(BOW) model, yielding an unordered word count vector, was used to represent the text. Both words and character n-grams were tried as vocabulary for the BOW, of which character n-grams had the highest performance. Though the bag-of-words model is commonly used, it is limited in its representation. In their work, Wang et al. aimed to expand the BOW representation with semantic knowledge, which showed to improve text classification [31]. Later, the introduction of word embeddings would automate this process as semantic relations between words could be learned end-to-end[23].

By using word embeddings, neural methods for text classification like the Long Short-Term Memory network(LSTM), have become state of the art. A bi-directional LSTM for sentiment analysis has been proposed by Ruder et al.[27]. The model is capable of using pre-trained Glove word embeddings[25] or learning embeddings on its own during training. Another method, operating at the character level, portrays text as a source of raw data similar to images. This method, proposed by Zhang et al., introduces a Convolutional Neural Network(CNN) for text classification[33]. Through convolutions and pooling of characters, patterns in the text can be found, regardless of language or semantics. Results indicate the model is indeed capable of finding patterns in the ‘raw’ character data, outperforming BOW and LSTM methods when enough data is available.
Chapter 4

End-To-End Classification of Text Data

In this chapter, the proposed end-to-end algorithm, able to classify raw text data directly, will be described in detail. Figure 4.1 visualizes the complete algorithm once again, illustrating the separate phases of the algorithm. Creation of the regular expressions that transform the text data into a set of usable features is achieved in the feature generation step. In this step, the genetic programming algorithm evolves a population of regex individuals and returns a set of regular expressions capturing predictive patterns in the data. The design of this genetic programming algorithm is discussed in section 4.1. Once the feature generation step is completed and a set of regular expressions has been proposed, the algorithm can proceed to training and evaluation of the classifier. In this work, a logistic regression classifier is chosen, as it is fast to train due to its simplicity. The logistic regression classifier is trained on the same data as provided to the GP algorithm to learn features from. The training and evaluation data is transformed into a set of binary features using the proposed regular expressions to yield match features which are ready for use by the classifier. Finally, the performance of the classifier, driven by the learned features, can determined by comparing classifier predictions with the true labels. As the GP algorithm follows a similar process in its fitness calculation step, a more detailed description of this process is provided in subsection 4.1.3

Figure 4.1: Visualization of steps in the end-to-end text classification algorithm. Regular expressions are created during the feature generation step. The created regular expressions are then used during training and evaluation of the classifier to transform the text into a set of usable features.

4.1 Evolving Regular Expressions

During feature generation, a set of regular expressions that can be used to transform the data into a set of features must be learned from the data. A genetic programming algorithm is used to evolve
regular expressions suited for this task. This section describes the genetic programming algorithm in further detail. First, the individuals in the population, representing a set of features, are defined. Next, the steps of the evolutionary process in genetic programming, shown in Figure 4.2, are explained. Lastly, a multiprocessing extension of the algorithm, yielding significant speed improvements, is presented.

4.1.1 Individuals

Like with any evolutionary algorithm, the first step in the design of a genetic programming algorithm is to find a fitting representation for the individuals in the population. Defining the phenotype is relatively simple, as it has the same form as the desired outcome of the algorithm. As the goal of the algorithm is to find a set of regular expressions to use in matching, the phenotype should also be a set of regular expressions. However, to allow for a straightforward conversion between this set and the genotype, an abstraction is introduced. Rather than being modeled as a set of regex strings directly, the separate regexes are concatenated to form one large string, where regular expressions are separated by the SPLIT character (‘||’). When evaluation of the phenotype is necessary, a string split is performed using the SPLIT character as the delimiter, yielding a list of separate regular expressions. For example, the phenotype string ‘example||string’ yields a set of two separate regular expressions: ‘example’ and ‘string’, each of which is assigned a different weight during training.

The genotype of the individual is defined as a tree structure, consisting of terminal and non-terminal nodes. Traditionally, the set of non-terminal nodes is made up of the operators in the syntax, with the remaining literal characters as terminal nodes. Though this division is sufficient for some syntaxes, for instance, mathematics, regular expressions benefit from the ability to concatenate literal characters to form words, or parts thereof. To facilitate this, a special operator that concatenates two characters can be introduced, which is the approach of Bartoli et al.[4]. However, this approach makes the formation of longer strings unlikely and increases the amount of nodes in the tree dramatically, as many concatenation nodes are required. This work proposes to give characters the ability to concatenate other characters directly, without the need for a special operator, by adding the character as a child node. A result of this change is, however, that all characters are now non-terminals. A new terminal node is therefore defined: the empty string node (‘’). This node, which is added as a descendant of literal characters automatically, indicates that a leaf node has been reached, while also maintaining the option to be exchanged for other char-
acter nodes by variation operators. Besides literal characters, character sets and capturing groups also receive the concatenation option. As quantifier operators in regular expressions influence the character directly to the left, it is important to keep this relation clear in the tree structure. For this reason, characters are concatenated from right to left, so the functionality of the operator on the character is maintained.

Nodes in the non-terminal set have different requirements when it comes to the position and amount of child nodes, changing how the tree is parsed. Four mutually exclusive categories of nodes can be distinguished: enclosing nodes, right-bound nodes, left-bound nodes, and center nodes.

**Enclosing Nodes**

Enclosing nodes contain multiple characters that surround characters from child nodes. These nodes are used for set operators, range quantifiers, and capturing groups. Though the amount of children varies per operator, the left-most child is always the concatenation node. During interpretation, this value is placed left of the operator. The remaining child nodes are placed inside the characters of the operator, with the exception of the right-most child node of the capturing group operator. This node, saved for backreferencing the capturing group, is only able to take the empty string or \( \backslash r \). If the \( \backslash r \) character is chosen, it is replaced in a later stage by a number based on the amount of capturing groups in the string.

![Figure 4.3: Subtrees with enclosing node operators. From left to right: an exact match quantifier with a concatenated x, a character range with a concatenated x, and a capturing group with a back reference and no concatenation](image)

**Right-bound Nodes**

Right-bound nodes put their character on the right of their child nodes. These nodes are used for the literal characters, as well as the assertion operators, with the exception of \( \backslash ^{\wedge} \), which must be put to the left of a string. Examples of right-bound nodes can be seen in Figure 4.4.

![Figure 4.4: Subtrees with right-bound nodes. On the left, two concatenated literal characters. On the right, an end-of-string assertion operator with child node ‘a’](image)
Left-bound Nodes

As mentioned in the previously, the start-of-string assertion operator (‘\^’) needs a string on the right in order to be valid. If a string were to be placed to the left of this operator, the regular expression becomes impossible to match, i.e., invalid. The start-of-string assertion operator is the only operator requiring a left-bound node. An example of its parsing behavior is shown in Figure 4.5.

![Figure 4.5: A subtree with a left-bound node. The value of the operator is placed left of its child node](image1)

Center Nodes

The value of center nodes is placed in between the values of their children. Operators using a center node are the OR operator (‘|’) and the new SPLIT operator (‘||’). Whereas the OR operator is always binary, the SPLIT operator, which can only be used as a root node in order to keep separation between regexes clear in the tree structure, can take any amount of children. Child nodes can be added and removed through variation operators. When parsed, the SPLIT character is placed between each of the child nodes. If the SPLIT operator has a single child node, no SPLIT character is used, as can be seen in Figure 4.6.

![Figure 4.6: Subtrees with center node operators. From left to right: the binary OR operator, a SPLIT operator with three child nodes, and a SPLIT operator with a single child node.](image2)

Quantifiers

So far, several quantifiers have been left unmentioned: the kleene plus, kleene star, and question mark. Though these operators are right-bound and could be implemented as such, no special nodes have been allocated to these operators in this work. Instead, the operators have been rewritten into their equivalent range match quantifier representations to decrease model complexity while maintaining an equal level of functionality. When a range match quantifier node is generated, a choice is made at random between the initializations shown in Table 4.1.
\{0,1\} equal to \? \\
\{1,\} equal to + \\
\{0,\} equal to * \\
\{d,d\} a range quantifier with two random digits

Table 4.1: Possible initializations of a range quantifier

### 4.1.2 Population Initialization

In the first step of the genetic programming algorithm, a starting population is generated randomly. To create an individual, a new genotype tree must first be generated. Next, before the individual is added to the population, the phenotype of the individual undergoes several tests, ensuring validity and uniqueness. When the individual is accepted and added to the population, a small amount of new individuals based on subtrees of the original individual can also be added. The process of generating, evaluating, and adding good individuals continues until the population has reached a predefined size, indicated by the hyper parameter ‘population_size’.

#### Individual Generation

Random generation of the individual’s genotype tree is achieved through ramped half-and-half initialization\[20\]. In ramped half and half initialization, 50% of the trees are grown naturally by randomly generating child nodes for non-terminal nodes until a maximum depth is reached or no more non-terminal nodes are available. The other 50% of trees are developed by only selecting terminal nodes once the maximum depth has been reached, forcing the tree to fully expand all of its branches.

#### Individual Acceptation

An individual must match several criteria before it is ready to be added to the population. These criteria apply not only to randomly generated individuals, but also to those created by the crossover and mutation operations. First, the validity of the individual’s phenotype string is evaluated, as the random operations in the algorithm could result in syntactically incorrect regular expressions. Next, if the individual has parent individuals, its regex string is compared to those of its parents with the Levenshtein distance to ensure a degree of uniqueness\[21\]. In order to be allowed, a distance of at least 5 operations is required between the individual and its parent(s). However, the latter constraint does not restrict the algorithm from creating similar individuals completely. To ensure no duplicate individuals exist within the population, individuals are maintained in a dictionary, using the individual’s phenotype as the key. When both criteria are satisfied and the string of the individual cannot be found in the dictionary, the individual is ready to be added to the population.

#### Subtree Individual Generation

Individuals that have been accepted into the population have the chance of creating ‘subtree individuals’ if the parameter subtree_generation is set to true. A subtree is created by selecting a random node in the tree of the individual and denoting this as the root of a new tree. A new individual is then instantiated with the subtree and added to the population, promoting the search for shorter regular expressions. Lastly, a fitness is calculated for the original individual and any subtree individual that may have been created.

### 4.1.3 Fitness Calculation

In fitness calculation, the performance of an individual is measured by applying its proposed solution to the problem. Based on the performance of this solution, a fitness score is assigned to the individual. To evaluate the performance of a proposed set of regular expressions, conditions should be as similar as possible to those during final predictions. Therefore, the fitness calculation procedure consists of the same steps as described in the classifier training/evaluation phase of Figure 4.1. First, the text is transformed into a set of features using the proposed regular expressions.
Next, the features are used to train a classifier and predict labels on a test set. Finally, the quality of the predictions is calculated by the fitness function, determining the fitness of the individual.

Feature Transformation

In the feature transformation step, shown in Figure 4.7, the text requiring classification is transformed into a feature vector $\vec{v}$. Rather than representing the text as a vector directly, as is done by approaches like bag-of-words vectors, this vector is based on the match results of regular expressions on the text. Each entry in $\vec{v}$ represents the result of a single regular expression search when applied to the text. A boolean value indicates whether the regex was able to find a matching string in the text or not. As the proposed set of regular expressions is generally quite small, match feature vectors are much more compact than feature vectors based on a vocabulary, such as the bag-of-words vectors.

A simple example of this conversion from text to match features is given below, where the set of regular expression (‘aa’, ‘bb’, ‘cc’) is applied to the text ‘aaaa’.

<table>
<thead>
<tr>
<th>String</th>
<th>Regex</th>
<th>Match</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>aaaa</td>
<td>aa</td>
<td>aa</td>
<td>1</td>
</tr>
<tr>
<td>aaaa</td>
<td>bb</td>
<td>None</td>
<td>0</td>
</tr>
<tr>
<td>aaaa</td>
<td>cc</td>
<td>None</td>
<td>0</td>
</tr>
</tbody>
</table>

$\rightarrow \vec{v} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$

As matching needs to be performed for every regular expression and every example in the dataset, it can be quite costly for large data. In order to avoid unnecessary computations, a dictionary of previously evaluated regular expressions is maintained. Once a match vector is constructed for a regular expression, it is saved in the dictionary using the string of the regex as the key. In doing such, match features of previously evaluated regular expressions can be retrieved instantly, saving large amounts of time. However, as maintaining such a dictionary requires large quantities of memory, only 1000 match feature vectors are stored at once, removing older vectors as new vectors are stored.

Classifier Training

In order to evaluate the quality of the match features created with the set of regular expressions, a classifier is trained and evaluated. In order to avoid overfitting, the training data is split randomly into a training set and a model validation set, as shown in Figure 4.8.
Match features are generated for each text example in the training set, resulting in a $D \times V$ matrix, with $D$ as the size of the dataset and $V$ indicating the amount of regular expressions in the set. The classifier can then be trained by providing the $D \times V$ features and the $D \times 1$ labels as visualized in Figure 4.9. Though it is advised to train a model similar to the model that will be used during final evaluation to keep variation at a minimum, the out-of-the box logistic regression classifier used for final evaluation did not work well with the multiprocessing approach used during evolution (discussed in subsection 4.1.7). As a result, a decision tree classifier is trained during evolution instead, with similar final outcomes compared to logistic regression.

Once the classifier has been trained on the data, its predictions are evaluated on the model validation set as can be seen in Figure 4.10. As performance on this dataset is used to determine the fitness, which in turn influences the population, it can be seen as another training set, with the first training set in charge of the weights of the classifier and the second training set influencing the regexes that are found in the population. The predictions of the classifier, as well as the correct labels, are provided to the fitness function which then calculates the fitness of the individual based on the quality of the predictions.
Fitness Function

The fitness function assigns a fitness to an individual based on the quality of the predictions a classifier can make with the proposed regular expressions. Additionally, the algorithm should prefer concise expressions, which is effectuated with a penalty based on the total length of the regular expressions string. The combination of performance and penalty is captured in the following formula:

\[ f(i) = MCC(y, \hat{y}) - \frac{\text{len}(i)}{10000} \]

Here, \( i \) symbolizes the regular expressions string of the individual, \( y \) symbolizes the predictions made by the classifier, and \( \hat{y} \) the correct labels. The length penalty is divided by 10000 as this value showed to provide a good balance between exploration and penalization of longer strings during development. Lastly, MCC stands for Matthews correlation coefficient, a metric for binary classification problems that takes into account all four values of the confusion matrix and has been shown to be more informative than other widely used metrics like accuracy or F1-score\(^\text{[10]}\). Figure 4.11 shows the formula for the MCC using the confusion matrix of the predictions. Following this formula, the outcome of Matthews Correlation Coefficient ranges from -1 to 1, with -1 indicating the classifier has learned a negative correlation, 0 indicating the classifier performs randomly, and 1 indicating positive correlation. As machine learners operate based on positive correlation, the MCC range is effectively limited from 0 to 1.

\[
MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}
\]

Figure 4.11: The formula of Matthews Correlation Coefficient uses all parts of the confusion matrix for binary classification problems\(^2\)

4.1.4 Offspring Generation

Once the fitness of every individual in the population is known, the algorithm can proceed to generate offspring. In order to generate offspring, several steps are taken. First, a choice is made randomly between crossover and mutation, with mutation being chosen only 10% of the time. Next, the parent individuals required for the operation are selected in parent selection. Once the parents have been selected, the crossover or mutation operation is executed and the resulting offspring is evaluated for validity. Finally, fitness of the valid offspring is calculated and the individuals are

\(^2\)https://rasbt.github.io/mlxtend/user_guide/evaluate/confusion_matrix
added to the list of offspring. This process is repeated until the list of offspring is of the same size as the population.

**Parent Selection**

In parent selection, one or more individuals from the population are chosen to generate offspring. Generally, individuals are selected proportionally to their fitness and can be selected multiple times.

For this work, tournament selection has been chosen as parent selection method due to its fast execution and stability[24]. In tournament selection, a subset of individuals is drawn uniformly from the population without replacement. Individuals in the subset are ranked by fitness and the fittest individual is returned as the new parent. If multiple parents are required the complete process is repeated. As the subset size is generally much smaller than the size of the population, much time can be saved compared to methods like the roulette-wheel selection procedure, which must calculate a selection probability for each individual in the population, whereas in tournament selection, many individuals in the population can be ignored after the initial random sampling. Additionally, tournament selection is more stable compared to roulette-wheel selection when differences in fitness between individuals becomes relatively small. As the fitness of individuals ranges from 0 to 1, individuals are likely to obtain comparable fitness values after a few generations. In roulette-wheel selection, this would lead to all individuals having approximately the same probability of being selected as a parent. As tournament selection uses an absolute ranking for selection, stronger individuals will always be selected, even when fitness values are closely grouped.

**Crossover**

In crossover, subtrees from two parents are exchanged to create two new individuals. Subtrees are taken by randomly selecting a node and all its descendants in the parent tree with uniform probability. The root node of this subtree is called the crossover point. As SPLIT nodes can only exist at the root of a tree for simplicity, it is not possible to perform regular crossover with these nodes. Instead, if a SPLIT node is selected, a random child node of the SPLIT is selected to form the new root of the subtree. However, the SPLIT node can still receive the subtree from the other parent. Subtrees that perform crossover with a SPLIT node will be added as a child of the node, rather than replacing one of the children. An example of this behavior can be seen in Figure 4.12.

![Figure 4.12: Example of the crossover variation operation. Subtrees of the crossover points are exchanged, creating two child trees. As the split operator is selected for parent 2, the subtree of parent 1 is added instead of swapping. Parent 2 falls back on the subtree of one of the child nodes to exchange with parent 1.](image-url)
Mutation

The mutation variation operation is most useful to introduce characters that are not yet represented in the population and could therefore not be found using crossover alone.Mutation takes a single parent and produces a single new individual. Three varieties of mutation have been defined in this work: point mutation, subtree mutation and insert mutation.

In point mutation, a random mutation point is chosen by selecting a node in the parent tree uniformly. Point mutation aims to replace the node at the mutation point while keeping the remainder of the tree intact. To this end, a new node is first created randomly. Next, the random node takes place at the mutation point in the tree. Lastly, children of the original node are added to the new node. However, the new node often requires a different amount of children than the original node. Therefore, an extra step is taken to repair the tree in case the amount of children is not equal for both the original and new node. In this step, children are shifted randomly from the original node to the new node until the new node has the required amount of child nodes. If the new node requires additional child nodes after all children have been shifted, the missing nodes are generated randomly. A visual example of point mutation is given in Figure 4.13.

![Figure 4.13: Example of the point mutation operation. The node at the mutation point is replaced but its child node is kept. As the new node requires an additional child node, a random child is generated.](image)

Subtree mutation is designed to replace full subtrees of the parent tree, compared to only a single node in point mutation. In subtree mutation, a mutation point is again selected uniformly from the nodes. Next, a new subtree is randomly generated. This subtree is then added to the tree at the mutation point, effectively dropping the mutation point node and all its descendants. An example of subtree mutation can be seen in Figure 4.14.
Matching results of regular expressions are quite rigid as a single missed character can fail the entire regex. Insert mutation was designed to introduce possibly beneficial variations without influencing the structure of the existing string. If, for example, the optimal regex would be ‘ab?’ and the string ‘ab’ was found, none of the previously mentioned mutations would be likely to introduce the variation that moves the solution towards the optimum. Point mutation would only be able to generate the optimal solution by randomly generating the combination b? and replacing b. Similarly, subtree mutation would only be able to find the correct solution by generating ab? exactly and replacing b. To increase the likelihood of finding the correct solution, insert mutation is capable of introducing the ? operator directly into the tree, leaving ‘ab’ unchanged.

As the name suggests, insert mutation works by inserting a node rather than replacing one. After the mutation point is selected, a random node is generated. The new node is then placed in between the mutation point node and its parent node in the tree. If additional child nodes are required by the insert node they are generated randomly. Figure 4.15 shows an example of this behavior.
Figure 4.15: Example of the insert mutation operation. A newly generated node is inserted at the
mutation point. The original node is added as a child node of the insert node. If more child nodes
are required they are automatically generated

4.1.5 Survivor Selection

Once all offspring has been created, the amount of individuals is now twice as big as the pre-
determined population_size parameter. Survivor selection reduces this amount to the set pop-
ulation_size so that each generation of parents is of equal size. Two types of survivor selection
have been implemented: $(\mu + \lambda)$ selection and $(\mu, \lambda)$ selection with elitism. By default, the algo-
rithm uses $(\mu + \lambda)$ selection as this is the safest method and converges faster. However, in the
multiprocessing approach described in subsection 4.1.7, $(\mu, \lambda)$ can also be used.

$(\mu + \lambda)$ selection

In $(\mu + \lambda)$ selection, the parent population($\mu$) and the offspring($\lambda$) are first merged into one large
population. Next, individuals are ranked by their fitness score. Finally, the population is reduced to
the original population_size, keeping only the fittest individuals. The remaining set of individuals
is called the survivor set and will be used as parent population in the next generation. An overview
of $(\mu + \lambda)$ selection can be seen in Figure 4.16.

Figure 4.16: In $(\mu + \lambda)$ selection, all individuals are ranked based on their fitness. In order to
retain a set of survivors as big as the original population, the part of individuals with the lowest
fitness are cut.

$(\mu, \lambda)$ selection

In $(\mu, \lambda)$ selection, only the offspring is eligible to advance to the next generation. As offspring
is not necessarily fitter than its parents, this could lead to a decrease in the overall fitness of the
population in some generations. As a result, the population is more likely to overcome local optima and move toward the global optimum. Through the addition of elitism, it is possible to guarantee the fittest individual will not be lost while also maintaining the previous concept. In elitism, the parent population is first ranked in order to find the top scoring individuals. These individuals are then merged with the offspring, after which the remaining process is similar to \((\mu + \lambda)\) selection. As the highest scoring individual from the parent population is added to the list of possible survivors, it is guaranteed that the fittest individual will never be less fit than previous generations. However, as the amount of elites added is small compared to the size of the population, overall fitness of the population can still decrease and local optima can still be overcome. In Figure 4.17 the process of \((\mu, \lambda)\) selection with elitism is visualized.

![Figure 4.17](image)

Figure 4.17: In \((\mu, \lambda)\) selection, the fittest individual of the parent population are selected and merged with the offspring. The merged individuals are then ranked and reduced to yield a set of survivors.

### 4.1.6 Separate-and-Conquer

To aid the algorithm in finding missed examples, a variant of Bartoli’s Separate-and-Conquer method is implemented in the search algorithm[5]. Though the original approach was meant to guide usage of the OR operator, our algorithm is capable of adding and removing OR operators on its own. Instead, the adjusted version of separate-and-conquer works by adding expressions to the SPLIT node. As these expressions can be adjusted after they are added, unlike the expressions in the original separate and conquer, the original requirement for perfect precision before separation can be ignored. Instead, the algorithm decides to start separating when the best individual remains unchanged for 40 generations. To separate the data, a classifier is trained with the features of the best individual. Positive examples that can be correctly predicted with the existing regexes are removed, allowing the algorithm to focus on new parts of the data. Additionally, the current regexes are saved and a new population is initialized randomly. The algorithm then continues the evolutionary process, evolving the new population until the best individual again remains unchanged for 40 generations. The regular expressions of the new best individual are then concatenated with the regular expressions saved before the separation, yielding an individual that is possibly better suited for the complete dataset. Next, the training data is restored to its original state and another new population is initialized. The original best individual, new best individual, and concatenated individual are also added to the new population.

### 4.1.7 Distributing with Multiprocessing

Since obtaining match features and training a classifier requires iterating over all examples in the train set, the algorithm can become quite slow for large amounts of data and big population sizes. Through a multiprocessing approach, the algorithm is capable of significantly decreasing completion time by parallelizing much of the work in the genetic programming algorithm. To manage parallelization of the algorithm, an island model approach is implemented.
Island Model

The island model is a parallelized approach for evolutionary algorithms in which the population is divided into subpopulations called ‘islands’ [29]. Island subpopulations follow the complete evolutionary process described in previous sections. Since the population of an island is isolated from other islands, crossover is only possible between individuals located on the same island. Besides allowing for easy parallelization of the evolutionary process, this separation also allows for each island to explore different parts of the search space, leading to more variation in the full population. However, as the optimal solution could lie in a combination of these separated individuals, some shuffling of the individuals is wanted. This shuffling is facilitated through the migration of individuals. Every 100 iterations, evolution on the islands pauses and a few individuals are selected to transfer to another island with uniform probability. In order to retain equal population sizes on the islands, migrating individuals are swapped with one of the individuals on the other island. After migration has introduced new genes to the populations, the islands continue the evolutionary process and evolve their populations further.

In the island model, settings on the islands do not need to be identical. For instance, the parameters determining whether subtree individuals can be generated and which type of survivor selection is performed is varied between islands. Furthermore, to increase the speed of the algorithm for large datasets, data is also distributed over the islands. In order to prevent overfitting on these subsets of the data, data is re-distributed randomly over the islands whenever the islands resume evolution of the population.

Solution Selection

Whenever an island pauses its evolution, its best individual is reported. However, as there are multiple islands, each proposing their best individual as the optimal solution, a method of finding the absolute best is required. In order to find the individual best suited for the test data, a validation dataset is constructed. The prediction score of each island’s best individual, measured in MCC, is measured and the absolute best is saved in a list. Once the maximum amount of iterations has been reached, the algorithm finishes and the individual scoring highest on the validation set is returned to the user as the optimal solution.
Chapter 5

Experiments

In this chapter, a description of the experiments, performed to help answer the research question, will be provided. A variety of experiments have been conducted on a variety of datasets. In section 5.1, the origin and content of each dataset will be discussed. Next, in section 5.2, the settings of the algorithm regarding hyperparameters are specified and the setup of the different experiments is explained.

5.1 Datasets

In order to ensure that the algorithm works well in general rather than on a select set of problems, a variety of datasets have been used for testing. In this section, a short description of each dataset is provided along with a few examples found in the data.

5.1.1 Regex 1

Regex 1 is a dataset constructed for the development of the regular expression generation algorithm. A regular expression was created manually in order to evaluate whether the algorithm was able to re-create it. Example strings are constructed randomly with a non-deterministic finite state automaton generating sequences from the following set of characters: ('a', 'b', 'c', 'x', 'y', ':', ' ', '0', '1', '3', '7', '8', '9'). Labels are assigned to the examples by matching the regular expression with the strings, yielding a positive example if the pattern can be found in the string and a negative example otherwise. The following regular expression was constructed for the regex 1 dataset:

\[\text{[abc:]\+}\{a?\}\{d\{2,4\}\}\{\text{\(\text{\$}\)}\}\|\text{xy}\]

Following this regular expression, a string is a positive example if it contains either of two sequences:

1. a sequence consisting of at least one ‘a’, ‘b’, ‘c’ or ‘:’, possibly a whitespace, 2, 3 or 4 digits and finally a non-digit or nothing at all
2. a sequence consisting of ‘xy’

In Table 5.1, a few of the generated examples and their accompanying labels are presented. As can be seen in Figure 5.1, roughly one third of the examples in the dataset contain the pattern described by the regular expression.

<table>
<thead>
<tr>
<th>String</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>aacbcabcbab 000abaab</td>
<td>1</td>
</tr>
<tr>
<td>a30389bb 73:ya</td>
<td>1</td>
</tr>
<tr>
<td>:y::acc 7088cbb01</td>
<td>1</td>
</tr>
<tr>
<td>yx9388bba</td>
<td>0</td>
</tr>
<tr>
<td>978x:c</td>
<td>0</td>
</tr>
<tr>
<td>bababc</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.1: Example strings from the Regex 1 dataset. Positive examples contain a string matching with the mentioned regular expression.
5.1.2 Regex 2

The Regex 2 dataset was created to further test the performance of the algorithm on regex patterns. Example strings in the Regex 2 dataset were generated using the same non-deterministic finite state automaton as used for Regex 1. For this dataset, examples were matched on the following regular expression:

\((\w)\1\{2\}\)\(a(\w)a\)

This regex matches strings with either of the following sequences:

1. a sequence of any word character repeated exactly 3 times
2. a sequence of ‘a’ followed by any word character and another ‘a’

For this dataset, example strings and labels can be found in Table 5.2. Classes are distributed evenly in the data, as can be seen in Figure 5.2.

<table>
<thead>
<tr>
<th>String</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>xbyxcacac</td>
<td>1</td>
</tr>
<tr>
<td>8ycccb</td>
<td>1</td>
</tr>
<tr>
<td>bbababcabcbbx7</td>
<td>1</td>
</tr>
<tr>
<td>8cbbc::a793</td>
<td>0</td>
</tr>
<tr>
<td>:cac</td>
<td>0</td>
</tr>
<tr>
<td>78397783139y:1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.2: Example strings from the Regex 2 dataset. Positive examples contain a string matching with the mentioned regular expression.
5.1.3 Spam

The Spam classification dataset was taken from the work by Ruano-Ordás et al. on Spam detection with regular expressions[26]. In this dataset, email headers are used as example strings. Headers belonging to Spam messages are labeled as positive examples, whereas Ham email headers form negative examples. Positive and negative examples are presented in Table 5.3. Data in this dataset is slightly skewed toward Spam messages, which form roughly 60% of the data, as can be seen in Figure 5.3.

<table>
<thead>
<tr>
<th>String</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>We cure any disease!</td>
<td>1</td>
</tr>
<tr>
<td>re: Can’t be a lover anymore?</td>
<td>1</td>
</tr>
<tr>
<td>chief pills at low down worth.</td>
<td>1</td>
</tr>
<tr>
<td>svn commit: samba r22322 - in</td>
<td>0</td>
</tr>
<tr>
<td>When sending a HUP signal isn’t enough?</td>
<td>0</td>
</tr>
<tr>
<td>Optimizing Genco Assets in New ERCOT Nodal Market May 9-10 San Antonio</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.3: Example strings from the Spam dataset. Positive examples have been identified as Spam emails. Negative examples are called Ham.

Figure 5.3: Class distribution in the Spam dataset. Approximately two thirds of the data consists of positive examples.

As this dataset consists of examples in natural language rather than random strings, analysis on frequently occurring word and character n-grams can provide further insight into the composition of the dataset. The top 50 most occurring word n-grams can be found in Figure 5.4. As may not be surprising, short and common words like prepositions and articles are found at the top of the distribution. However, some n-grams specifically belonging to one of the two classes, like ‘avis important’ for Spam emails or ‘svn commit’ for Ham emails, can be found in the most frequent n-grams.
5.1.4 PPI

The PPI dataset for private individual detection has been provided by ING. Contradictory to its name, the dataset labels companies as positive examples. As the data contains personal information about ING clients, examples cannot be discussed without anonymizing account names. Table 5.4 shows examples from the PPI dataset that have been modified to guarantee client anonymity while also leaving intact important patterns. As can be seen in Figure 5.5, the PPI dataset is skewed heavily towards private individuals.

<table>
<thead>
<tr>
<th>String</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>Het *** *** Nederland B.V.</td>
<td>1</td>
</tr>
<tr>
<td>Stichting *** ***</td>
<td>1</td>
</tr>
<tr>
<td>*** *** BV</td>
<td>1</td>
</tr>
<tr>
<td>P. ***</td>
<td>0</td>
</tr>
<tr>
<td>***. *** van de</td>
<td>0</td>
</tr>
<tr>
<td>R.W.J. van ***</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.4: Example strings from the PPI dataset. Positive examples belong to company accounts. Negative examples are private individuals. Client data has been anonymized for privacy.
Figure 5.5: Class distribution in the PPI dataset. The dataset mainly consists of private individuals with roughly 10% of the data labeled as company.

Similarly to the Spam dataset, common words can be extracted from the PPI data as well. The top 50 frequent word n-grams, counting unigrams and bigrams, are displayed in Figure 5.6. Frequent words include infixes of personal names like ‘van’ and ‘de’ as well as titles like ‘dhr’ and ‘heer’ and several common surnames like ‘visser’ and ‘bakker’. Additionally, several company titles, such as ‘stichting’, ‘vereniging’, and ‘bv’, can be found in the top 50 as well.

Figure 5.6: Most frequent word n-grams in the PPI dataset. Both unigrams and bigrams have been recorded.
5.2 Experimental Setup

In this section, the performed experiments will be described. Experiments are conducted for each of the sub-questions of the research question. Since the algorithm uses many random events for its evolutionary procedures, a high variance could be observed in the results when the experiment is repeated with another random seed. In order to verify whether this is the case and overcome this problem if it were to occur, the algorithm is ran multiple times. In total, five runs are performed for each experiment with differing seeds for the random operations in the algorithm. Hyperparameter settings for the algorithm are equal for the different experiments and have been listed in Table 5.5.

Some experiments require the use of traditional text features, i.e. bag-of-word n-grams and bag-of-character n-grams, which have been generated with the CountVectorizer from scikit-learn’s feature extraction module\(^1\). Though most of the parameters were kept at default values, a grid search was performed on the PPI dataset to find the appropriate values for three parameters: ngram_range, min_df, and lowercase. For the parameter ‘ngram_range’, determining the amount of words or characters in the n-grams, the ranges of ((1,1),(1,2),(2,2)) and ((1, 1), (1,2), (2, 2), (3, 3), (2, 3)) were explored for the words and characters respectively. The minimum frequency of documents a token must be found in, given by ‘min_df’, was tried for values ranging from 1 to 5 for both words and characters. Lastly, ‘lowercase’, indicating whether text should be pre-processed into lowercase only, was tried with True and False for words and characters as well. The optimal parameters found by the grid search for both traditional features out of all combinations are reported in Table 5.6.

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nr of iterations</td>
<td>3000</td>
</tr>
<tr>
<td>Nr of islands</td>
<td>45</td>
</tr>
<tr>
<td>Island population size</td>
<td>100</td>
</tr>
<tr>
<td>Iterations between migration</td>
<td>100</td>
</tr>
<tr>
<td>Individual migration probability</td>
<td>0.01</td>
</tr>
<tr>
<td>Tournament size</td>
<td>5</td>
</tr>
<tr>
<td>Mutation probability</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Table 5.5: Choice of hyperparameter values for the genetic programming algorithm

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Value</th>
<th>Hyperparameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>word ngram range</td>
<td>1-2</td>
<td>char ngram range</td>
<td>2-3</td>
</tr>
<tr>
<td>min doc freq</td>
<td>2</td>
<td>min doc freq</td>
<td>2</td>
</tr>
<tr>
<td>lowercase</td>
<td>True</td>
<td>lowercase</td>
<td>True</td>
</tr>
</tbody>
</table>

Table 5.6: Hyperparameter values for the traditional features. Left: a bag-of-word n-grams vectorizer. Right: a bag-of-character n-grams vectorizer.

Data for the experiments is split randomly to create a training set, validation set, and test set. Of the data, 60% is attributed to the training set. Validation and test each consist of 20% of the remaining data.

5.2.1 Q1: Comparing Regex Features and Traditional Features

In the first sub-question, we asked whether a classifier with learned regex features would be able to outperform a classifier with traditional text features. In order to answer this question, the algorithm creates a set of regular expression features for use in a classifier. For the experiments, the logistic regression classifier from scikit learn’s linear model module is used with default parameters\(^2\). The quality of the generated features is determined by calculating the Mcc score of the predictions achieved by the classifier on a test set. This score is compared to the Mcc achieved by a classifier

---

\(^1\)https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer

\(^2\)https://scikit-learn.org/0.18/modules/generated/sklearn.linear_model.logistic_regression.html
using the traditional text features, word n-grams or character n-grams, on the same data to serve as a baseline.

5.2.2 Q2: Combining Regex Features With Traditional Features

The second sub-question concerned whether a classifier could benefit from a combination of learned regular expressions and traditional text features. To answer this question, the algorithm creates a set of regular expressions that work well in combination with the traditional text features. In order to combine the regexes with traditional features, match features from the learned regular expressions are concatenated with, or ‘stacked’ on, the vector of the traditional text features. A logistic regression classifier using these stacked features is then trained and evaluated. A classifier with only traditional text features is used as the baseline to observe whether the addition of regular expression features can predictive performance.

To ensure the regular expressions do not learn examples already found with the traditional features, the traditional features have to be included during training. However, as the traditional feature set is very large, using all features to train the classifier is too costly during the evolutionary process. Therefore, rather than using all the traditional features, a classifier is used to make predictions with the traditional features. These predictions are then used in the stacked features in stead of the full set of traditional features. As this prediction already captures the predictive capabilities of the traditional features, only individuals enhancing the original predictions will achieve higher fitness scores, meaning that learned features will automatically focus on examples that cannot be predicted correctly with traditional features.

5.2.3 Q3: Comparing Automated and Hand-made Regex Features

For the final sub-question, a comparison is made between automatically learned regular expression features and manually crafted features. Several regular expressions have been created for the PPI dataset by ING engineers to be used as features in combination with traditional text features. These features, based on domain knowledge, are described in Table 5.7.

<table>
<thead>
<tr>
<th>Regex</th>
<th>Captured strings</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\w(2,}\s)+</td>
<td>Company titles: B.V</td>
</tr>
<tr>
<td>.+,\s?\w(2,}$</td>
<td>Comma followed by one word: ‘Company, Paris’</td>
</tr>
<tr>
<td>^\w(2,})$</td>
<td>Abbreviated names: R.J Surname</td>
</tr>
<tr>
<td>^\w(2,}$</td>
<td>Words followed by a comma: ‘Name, Surname’</td>
</tr>
</tbody>
</table>

Table 5.7: Manually created regular expression features for the PPI algorithm

Additionally, six more features have been designed to improve performance of the PPI algorithm. These features and their description are given in Table 5.8.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Boolean value indicating whether a ‘-’ exists in the string</td>
</tr>
<tr>
<td>&amp;</td>
<td>Boolean value indicating whether a ‘&amp;’ exists in the string</td>
</tr>
<tr>
<td>)</td>
<td>Boolean value indicating whether a ‘)’ exists in the string</td>
</tr>
<tr>
<td># = 1</td>
<td>Boolean value indicating whether the string consists of a single word</td>
</tr>
<tr>
<td># &gt; 5</td>
<td>Boolean value indicating whether the string consists of more than five words</td>
</tr>
<tr>
<td>len &gt; 9</td>
<td>Boolean value indicating whether a word in the string is more than 9 characters long</td>
</tr>
</tbody>
</table>

Table 5.8: Additional manually created features for the PPI algorithm

In order to evaluate whether the algorithm can create stronger features than one would make by hand, the stacked classifier from the second sub-question is compared to classifiers combining traditional features with manually created features. One of the classifiers only stacks the four
regular expression features with the traditional features, whereas another classifier evaluates the performance of all ten features combined with traditional features.
Chapter 6

Results

In this chapter, the results of the experiments described in the previous chapter will be presented. First, section 6.1 shows the results of the first experiment comparing features generated by the algorithm to the traditional text features on a variety of datasets. Next, section 6.2 compares results of the stacked regex features with traditional text features to examine whether the learned features can detect patterns that cannot be captured by the traditional features to improve performance. Lastly, section 6.3 discusses the results of the final experiment, comparing the features made by the algorithm to manually designed features to conclude whether using algorithm could be practical in real situations.

6.1 Comparing regular expressions and traditional features

By comparing a classifier using regular expression created by the algorithm with a classifier using traditional features, the quality of the generated features can be determined. The experiment has been performed on all four datasets: Regex 1, Regex 2, Spam, and PPI.

6.1.1 Regex 1

As the Regex 1 dataset is created with regular expressions, the algorithm is very well suited for the data. This becomes evident in Figure 6.1, which reveals the algorithm was able to find a perfect solution, predicting 100% of the examples correctly in the training, validation, and test data, in every repetition of the algorithm.

![Figure 6.1: Aggregated scores of all runs on the Regex 1 dataset. Each run found features that enable the classifier to predict perfectly.](image)

As all runs of the algorithm scored equally on the predictions, the run yielding the shortest regex string was considered the best and selected for further analysis. The progress of the evolutionary process over multiple iterations is provided in Figure 6.2. It becomes clear that the algorithm required only 500 iterations to find a set of regular expression features achieving perfect predictions on the validation data. As the algorithm used a pre-defined number of iterations, further iterations were spent creating more compact versions of the found regular expressions. The final features of the best individual, shown in Figure 6.3, are similar to the regular expression used to create the data though not as compact.

39
Figure 6.2: Progression of validation set score over time. A perfect fitness score is achieved after 500 iterations.

![Figure 6.2: Progression of validation set score over time.](image1)

Figure 6.3: Features and weights of the best classifier for the Regex 1 dataset. Generated features are similar to the regular expression used to create the data.

![Figure 6.3: Features and weights of the best classifier for the Regex 1 dataset.](image2)

A comparison of the generated and traditional features, given in Figure 6.4, shows the generated features drastically outperform traditional methods. Even though performance of the word features seems decent when evaluating predictions accuracy, Mcc scores indicate the classifier could not learn to predict effectively with the features. Character n-grams, performing much better, were also unable to compete with the generated regex features. In Figure 6.5, the confusion matrices of the classifiers with the varying sets of features are displayed. A classifier with word features, unable to find patterns, heavily favors predictions of the most common class, resulting in a decent accuracy but low Mcc score. Contrastingly, a classifier with character n-grams is capable of learning from the data but makes several mistakes, whereas the regular expression features perform flawlessly.

![Figure 6.4: Comparison of generated and traditional features.](image3)

![Figure 6.5: Confusion matrices of classifiers.](image4)
Figure 6.4: Test set prediction scores for the different features. The highest score was achieved by the regular expression features.

Figure 6.5: Confusion matrices for the different features on test data. A classifier with word features heavily favors the negative class. Character n-grams are informative, but only regular expressions yield perfect results.

6.1.2 Regex 2

The second development dataset, Regex 2, was also created using regular expressions. On this dataset, the algorithm was able to generate features resulting in perfect predictions as well, as can be seen in Figure 6.6.

Figure 6.6: Aggregated scores for the Regex 2 dataset. All runs resulted in perfect predictions on the data.

Again, the most compact feature set was selected for evaluation. This set of features, shown in Figure 6.7, is nearly identical to the expression used to create the dataset. From the perspective of the algorithm, the generated feature is even fitter than the regular expression used to create the data, as \( (\w)\{1\} \) is one character shorter than \( (\w)\{2\} \), resulting in a lower penalty.
As can be seen in Figure 6.8, the algorithm was able to find a set of features yielding perfect predictions within the first 100 iterations. The shortest set of features was found after approximately 300 iterations. Though perhaps more challenging datasets would have provided additional insight, results prove the algorithm can quickly and consistently find patterns in string data with class labels, which was the goal when using the two development datasets.

Figure 6.7: Features and weights of the best classifier for the Regex 2 dataset. Only a single feature was required to capture all patterns in the data.

Figure 6.8: Progression of validation set scores for the second development dataset. Optimal features were found within the first 100 iterations.

Comparison of the scores in Figure 6.9 and confusion matrices in Figure 6.10 shows that character n-grams were almost able to achieve perfect classification as well. Word features led the classifier to predict the most common class on most occasions, similarly to the behavior on the Regex 1 dataset. However, once again, only the generated regular expression features enabled the classifier to make flawless predictions.
Figure 6.9: Test set prediction scores for the different features. Character n-grams allow for an almost perfect score, but are still outperformed by regular expression features.

Figure 6.10: Confusion matrices for the different features on test data. Generated regular expression features outperform traditional text features.

6.1.3 Spam

In contrast with the Regex datasets, the Spam dataset was created from real email, meaning the strings are natural language rather than randomized. On the Spam data, the algorithm was not able to find a perfect solution. However, as can be seen in Figure 6.11, results over multiple repetitions were quite stable, as only small differences can be found in the performance on average. Additionally, performance over the various data splits was also stable, indicating the classifier with generated regular expression features does not suffer from overfitting issues and, thus, generalizes well.

Figure 6.11: Aggregated scores on the Spam dataset. On average the runs achieve similar scores. Moreover, similar scores on train, validation, and test indicate the features generalize well.

Further analysis will focus on the set of features resulting in the highest score on the test set.
These features and their accompanying weights are presented in Figure 6.12. The two strongest features, $\{M\}$ and $[\$-\%]$, are responsible for the recognition of Spam emails. Many Spam headers in the dataset start with $[Mhln]$ which is recognized by the regular expression $\{M\}$. Other common occurrences in Spam emails are the $\$ and $\%$ signs, seen in headers like ‘Come grab your $1000’ and ‘Saving up to 70% on the meds’. Next, come two long regular expression using parts of Ham email headers help correctly classify Ham emails. For example, one of the parts in the expression is ‘vn’ which occurs often in Ham email headers like ‘svn commit: samba r22775 - in branches: SAMBA...’.

![Figure 6.12: Features and weights of the best classifier for the Spam dataset. Some features are used to recognize Spam emails, whereas other features focus on the recognition of Ham emails.](image)

In Figure 6.13, the prediction score on the validation set is shown for every 100 iterations. Performance on the validation set still seemed to be increasing when the algorithm terminated. However, a run with more iterations, shown in Figure 6.14 shows a flattening line after the default 3000 iterations, indicating the algorithm would not have been able to detect any further patterns.

![Figure 6.13: Validation set score for every 100 iterations. Performance increases until the algorithm terminates.](image)
By examining the comparison between features, shown in Figure 6.15, it becomes clear that the generated regular expression features are not quite as good as the traditional features on the Spam data. The highest score in both accuracy and Mcc is achieved by the classifier with character n-grams. The regular expression features created by the algorithm seem to function as an approximation character n-gram features, as only a little regular expression syntax is used. Though the generated features are not able to outperform the traditional features, their predictive performance is quite impressive, as the set of features is significantly smaller than those employed by the word and character n-gram features. Moreover, n-grams ranges do not have to be specified by the user, making the feature set more flexible than those of traditional features.

In Figure 6.16, the confusion matrices of the classifiers are presented. As the majority of examples in the data are Spam headers, the classifier is biased towards classifying emails as Spam. Unfortunately, the regex features are not able to capture many patterns belonging to Ham emails, resulting in many false positive errors. The traditional text features are better suited in this regard, resulting in significantly fewer false positives and, thus, higher prediction scores.
Figure 6.16: Test set confusion matrices for the different features. Regular expression features result in many false positive errors as patterns in Ham emails are missed.

6.1.4 PPI

Similarly to the Spam dataset, PPI consists of examples from natural language. The algorithm seems to have difficulty finding many patterns in this type of data, as the Mcc scores in Figure 6.17 are not that high. However, reported scores show a low variance between the multiple runs and strong generalization of the features even in more difficult datasets, which is beneficial to the credibility of the algorithm.

Figure 6.17: Aggregated scores on the PPI dataset. Even on more difficult datasets like PPI the algorithm maintains its stability across repetitions and generated features generalize well.

The generated features of the best run, visualized in Figure 6.18, reveal the features focused on the detection of companies, while the classifier is bias towards private individuals due to the large class imbalance. Text patterns captured by the algorithm include various company titles and parts thereof like BV/B V/B.V./bv/b v/b.v., V.O.F/V O F, St/St., Stg/Stg., VV (part of VVE), STIC (part of STICHTING), and nig(part of vereniging).
As can be seen in Figure 6.19, the algorithm converges after approximately 1500 iterations. A flattening of the line indicates the algorithm struggles to find further patterns in the data even though predictions of the classifier are nowhere near perfect at this point.

Analysis of the scores in Figure 6.20 indicate that PPI is a difficult dataset to do well on regarding predictions. Generated regular expression features come close to the traditional features in terms of performance but are unable to outperform them. However, predictions made with the traditional features are also subject to many mistakes. Similarly to the Spam dataset, the best Mcc score is achieved by the character n-gram features. It is however interesting to note the character n-grams score worst in terms of accuracy. The cause of these results becomes clear by evaluating the confusion matrices of the features in Figure 6.21. A classifier with character n-gram features makes many false positive mistakes as the n-grams are often shared by both classes. However, by overestimating the probability of a name belonging to a company, the amount of true positive predictions made by the classifier is significantly higher than for the other features. As true positives account for a higher increase in Mcc due to the steep class imbalance in the data, character n-grams achieve the highest Mcc while the total number of mistakes is slightly higher. In comparison, the classifier with word features is more balanced in its predictions, with roughly 45% of the mistakes being false positive. Regular expression features make the fewest false positive mistakes, yielding the highest prediction precision, but misclassify many company accounts resulting in the lowest Mcc.
Figure 6.20: Test set prediction scores for the different features on the PPI dataset. Character n-grams achieve the highest Mcc score, whereas word and regular expression features are tied for best accuracy.

Figure 6.21: Confusion matrices for the different features on test data. Classification mistakes are very distinct per feature: regular expression features result in many false negatives, character features yield many false positives, and word features balance both types of mistakes.

The clear difference in behavior between the features indicates the classifier could benefit from a combination of the features. Such a method, combining traditional and generated features, is described in the next section.

### 6.2 Combining regular expressions with traditional features

In this section, the question whether stacking regular expression features and traditional features improves performance of a classifier will be explored. As the regular expression features were able to predict 100% correctly on both development datasets, performing the experiment on these sets would not yield any additional insights. Therefore, this experiment is only performed on the remaining two datasets: Spam and PPI.

#### 6.2.1 Spam

Previous experiments showed classifiers using traditional features are capable of achieving high scores on the Spam data, outperforming regular expression features. New regular expression features were generated for use in combination with the traditional text features in a classifier. For the experiment, an evaluation of the classifier with stacked features has been performed for both sets of traditional text features: word and character n-grams.
Stacked Word Features

Multiple runs of the algorithm were performed, each generating a set of stacked word and regular expression features. Scores of the runs are provided in Figure 6.22. Though the features are stable over multiple runs, similarly to results in the previous experiment, the combination of word and regex features seems to suffer a bit from overfitting on the training data. As can be seen in Figure 6.23, the algorithm had difficulty improving on the features found after the first 100 iterations.

![Figure 6.22: Aggregated scores of all runs on the Spam dataset with stacked word features. The stacked word features seem to suffer slightly from overfitting during training.](image)

The difficulty experienced by the algorithm to find better features over time was in part caused by the strong predictive capabilities of the word features on the Spam data. As can be seen in Figure 6.24, predictions based on the word features received a significant share of the weights in the classifier. Regardless, the algorithm managed to find several regular expression features. Though most generated features do not seem to add much to the classification process, the strongest of the regular expression features stands out as it captures a pattern that could never have been modeled with the word features: `\s`. Though whitespace characters can be found by the word features, the features lack the means of detecting the position of said whitespace character in the string and are therefore unable to attain an equivalent feature. Intuitively, strings starting with whitespace characters do not sound like strong indicators for the classification of Spam or, if anything, like they would belong to Spam emails. Nevertheless, this feature is very useful in the detection of Ham emails in the data, as many Ham example headers start with a whitespace character, making it an easy to recognize feature for the algorithm.

![Figure 6.23: Achieved score on the validation set per 100 iterations. Over the full duration of the algorithm, only small improvements on the score were made.](image)
Figure 6.24: Stacked features and weights of the best classifier on the Spam dataset. Word predictions account for most of the weight in the classifier. However, several regular expression features were found by the algorithm and attributed to the predictions as well.

As can be seen in Figure 6.25, a classifier with stacked word features was able to find a decent improvement over the classifier using only traditional text features. Comparison of the confusion matrices indicates the additional regex features allowed the classifier to reduce its false positives drastically. However, the regular expression features were not flawless, resulting in a slight increase in false negative.

Figure 6.25: Comparison of the test set prediction scores for the traditional and stacked features. The highest score is achieved by the stacked features.

Figure 6.26: Confusion matrices for the features on test data. The addition of regular expression features decreases the amount of false positive errors drastically.

**Stacked Character Features**

In previous experiments, character n-grams were shown to outperform word n-grams on the Spam dataset, meaning even fewer patterns are left for the algorithm to find. As can be seen in Figure 6.27, stacked n-gram features also suffer slightly from overfitting on the training data. Furthermore, scores of the classifier with stacked character features were, on average, similar to those achieved with the stacked word features on validation and test data. Like with stacked words features, improvements made by the algorithm during the evolutionary process were tiny, as displayed in Figure 6.28.
As can be seen in Figure 6.29, the set of regular expression features stacked with the character predictions was reduced to a single regular expression: `\S`. This feature is similar to one of the features found in the experiment with stacked word features. However, as `\S` is the exact opposite of `\s`, the weight of the feature has swapped from negative to positive. Of course, as character n-grams features are, like word n-grams, unable to determine the position of a whitespace character in the string, it is sensible the algorithm would find the feature for stacked character features as well.
Figure 6.29: Features and weights of the best classifier using stacked character features on the Spam dataset. A single regular expression is found by the algorithm, capturing a pattern that cannot be described with character n-grams.

The comparison of prediction scores in Figure 6.30 shows the addition of the \S feature increases performance of the classifier slightly. Analysis of the confusion matrices in Figure 6.31 indicates this improvement is achieved through similar means as for the stacked word features, i.e., a decrease of false positive mistakes at the cost of a small increase in false negative mistakes, boosting the Mcc. Interestingly, when the scores in Figure 6.27 are re-evaluated, it becomes clear that, though the algorithm is capable of finding missing patterns, on average the classifier performs slightly worse when the features character are combined with regex features. This contrasts with the stacked word features, where even the worst performance was able to outperform the standalone word features.

Figure 6.30: Test set prediction scores for traditional and stacked character features. Stacked character features outperform the traditional features slightly.

Figure 6.31: Test set confusion matrices for traditional and stacked character features. The classifier with stacked character features makes fewer false positive errors but slightly increases the amount of false negatives.

6.2.2 PPI

Regular expression features were unable to outperform traditional features on the PPI dataset. In order to evaluate whether stacking of regular expressions and traditional features can improve
prediction performance on the PPI data, an experiment is performed on stacked word features and stacked character features.

Stacked Word Features

In order to evaluate whether patterns exist in the PPI dataset that cannot be captured by word n-gram features, the algorithm was used to create stacked word features. Results of the multiple runs of the algorithm are presented in Figure 6.32. Similarly to the classifier on Spam data, the classifier with stacked word features overfits on the train data, as average scores on the validation and test sets are much lower.

Figure 6.32: Aggregated scores of all runs on the PPI dataset with stacked word features. As validation and test scores are much worse than training scores, the classifier likely overfits during training.

Progress during the evolutionary process of the best classifier is shown in Figure 6.33. As indicated by the flat line, the algorithm had difficulty improving on the regular expression features found during the early iteration of the algorithm.

Figure 6.33: Progression of validation set score over time

In Figure 6.34 the weights of the features in the classifier with stacked word features are presented. The word predictions and bias account for most of the weight in the classifier. However, several interesting regex features are found in the set. First, the features \V\.$ and \V\. capture strings like B.V. and N.V. which cannot be captured by the word features, requiring at least two consecutive alphanumeric characters to qualify as a word, but are highly indicative of companies. Additionally, \V\. is capable of capturing V.O.F, another common title for companies. An empty regex was created by the algorithm which always finds a match in a string and therefore functions as another
bias toward private individuals. The final interesting feature is `\.`. Though this feature does not make much sense at first glance, analysis of the data indicates quite a few private individual account names follow the format `. "Family name"`, making this a unexpected but useful feature.

With these additional regular expression features, the classifier finds significant improvements over the classifier using tradition features, as can be seen in Figure 6.35.

Comparison of the confusion matrices in Figure 6.36 shows that the classifier with combined words and regex features drastically decreases the amount of false positive mistakes. Even though false negatives are increased by the additional regular expression, the overall change in Mcc is positive.

Stacked Character Features

Character n-grams achieved the highest score on the PPI dataset. The algorithm was used with stacked character features to determine whether the data contained patterns that could not be captured by the character n-grams. Performance of the algorithm on the PPI dataset is provided in Figure 6.37. Stacked character features seems to suffer less from overfitting compared to stacked word features.
Figure 6.37: Aggregated scores of all runs on the PPI dataset with stacked character features. Scores of the stacked features hardly vary over the multiple runs but seem to overfit slightly on training data.

As can be seen in Figure 6.38, performance of the classifier was not able to increase significantly after the first 100 iterations.

Figure 6.38: Score of the best classifier on the validation set per 100 iterations. Improvements after the first 100 iteration were too minute to be noticeable in the figure.

Features of the best classifier are shown in Figure 6.39. Though most regular expression features seem to provide no real benefit and have small weights, one specific regular expression stands out: \(|c|BA\). As there is nothing in front of the first OR operator in the expression, it matches on the empty string, which can always be found in an example. Because of this, the feature is always found and uses its negative weight to serve as an additional bias against the character prediction features. Lastly, it is interesting to note the other found regular expressions all consist of a single character, a pattern that cannot be captured by character n-grams as they only take into account bigrams and trigrams.
Traditional character n-grams features are overly confident when it comes to classifying examples as companies. As a result, many false positives mistakes are made by the classifier, as demonstrated in Figure 6.41. Through the use of the additional bias feature, false positive predictions are decreased dramatically by the classifier. Even though the additional bias towards private individuals results in quite a few false negatives, the decrease in false positives is of such magnitude that a significant increase in Mcc is witnessed over the traditional features, as can be seen in Figure 6.40. Unlike the performance of the stacked character features on the Spam dataset, the stacked features always outperform the standalone character features on PPI, even in the worst case.

Figure 6.39: Features and weights of the best classifier with stacked character features on the PPI dataset. The strongest regular expression feature captures the empty string and will always match, serving as an additional bias feature.

Figure 6.40: Prediction scores of the traditional and stacked character features on test data. Stacked character features drastically outperform the traditional character n-gram features.

Figure 6.41: Classifier confusion matrices for the different features. Stacked character features make significantly fewer false positive errors compared to traditional character n-gram features.

6.3 Comparing manually and automatically generated features

In this section, the results of the experiment comparing generated features with the manually created features described in subsection 5.2.3 are presented. As manually created features were only available for PPI data, the experiment is only performed on the PPI dataset.
6.3.1 Stacked Word Features

To compare the performance of the generated features with manually created features, a classifier has been trained and tested with each of the four feature sets: traditional word features, words + 4 manual regex features, words + 10 manual features, and words + generated features. A comparison of test scores, shown in Figure 6.42, reveals that both classifiers with manual features manage to outperform the standalone traditional features. The best performance was, however, achieved by the classifier using word features with stacked regular expression features generated by the algorithm. Interestingly, the 6 features added to the 4 manual regular expression features cause performance of the classifier to decrease slightly on training, validation and test data.

![Figure 6.42](image_url) Test set prediction scores of the classifier with the different features. The highest score is achieved by classifier with automatically generated features.

In Figure 6.43 and Figure 6.44, the feature weights of the classifiers with 4 and 10 manually created features can be observed. From the features listed in Table 5.7, the feature recognizing company title patterns receives the highest weight of the manual features in both classifiers. This division of weights is congruent with the findings on stacked word features in subsection 6.2.2, where the highest rated regex features were also indicative of company titles. Strings consisting of a single word seem to be indicative of company accounts as well.

![Figure 6.43](image_url) Features and weights of the classifier using stacked manual regex features. Of the regular expression, the feature capturing company titles receives the most weight.
6.3.2 Stacked Character Features

As character n-grams consistently outperformed word features on all datasets, the influence of stacking manually and automatically generated features with them was also explored. As can be seen in Figure 6.45, only slight improvements can be made by stacking the character features. However both manual and automatically generated features were able to improve performance of the classifier in terms of Mcc. The best score is achieved by the regular expression features created by the algorithm, improving both Mcc and accuracy slightly.

Figure 6.45: Test set prediction scores of the classifier with the different features. The highest score is achieved by classifier with automatically generated features.

Feature weights of the four manual regular expression features, shown in Figure 6.46, indicate only the regex capturing company title patterns is used actively during predictions. The other regex features are given only tiny weights, making them less useful. Most likely, the patterns captured by these features have already been captured by the character n-grams predictions feature, being the primary feature of the classifier. A similar story is portrayed in Figure 6.47, with only the company title regex receiving any significant weight.
Figure 6.46: Features and weights of the classifier using character features with four stacked manual regular expressions. Predictions are based largely on the predictions of the character n-grams, though the regular expression capturing company title patterns receives some weight as well.

Figure 6.47: Features and weights of the classifier using character features with ten stacked manual features. Similarly to the stacked regex features, predictions are based mainly on the character n-grams. Of the stacked features, the regular expression capturing company titles patterns receives the highest weight.
Chapter 7

Conclusion

In this work, an algorithm has been proposed to automatically generate regular expression features based on labeled text data. In section 7.1, a summary of the algorithm and experimental results is provided. Lastly, section 7.2 presents an outline for additional experiments that can be conducted and possible improvements to the algorithm that can be made in future work.

7.1 Summary

Though other work on the automatic generation of regular expression has been performed, these approaches are generally not capable of training a classifier end-to-end or require the user to specify the substring of the examples that must be captured.

Candidate feature sets are generated and evaluated within a genetic programming algorithm, returning the optimal feature set once the evolutionary process completes. The introduction of a “SPLIT” operator to the genetic programming algorithm allows for the flexible generation of multiple cooperating regular expression features. The features transform example strings into a binary vector representation appropriate for common machine learning algorithms like logistic regression. Fitness of a candidate feature set is calculated based on the quality of predictions, measured in Mcc, that have been made by a classifier using the feature set.

Large datasets and candidate population sizes are handled efficiently through distribution over multiple processes in an island model approach. This approach yields significant speed improvements compared to a single population approach and allows the algorithm to train on much larger dataset than are commonly used for regular expression generation.

The quality of the final features has been evaluated in several experiments and on a variety of datasets. In comparison, the automatically generated features are able to outperform the traditional text features, i.e. word and character n-grams, on datasets with clear patterns based on regular expressions. However, on datasets consisting of natural language examples like Spam detection and Private Individual detection, the algorithm struggles to find enough high quality features and achieves lower performance than the traditional features. However, regular expressions are able to achieve similar scores using a small set of features (approximately 8) compared to traditional features (approximately 20000). As a result, regex features might be faster during predictions, at the cost of a longer training time beforehand. Additionally, if one is familiar with regular expression syntax, the learned features might provide insight into common patterns in the data, which might be less the case for word features and certainly less for character n-grams.

Although the generated features are not able to outperform the traditional features in direct comparison, they are capable of finding patterns in the data that cannot be represented with the traditional features. Experiments on stacking traditional feature predictions with learned regex features, meaning the features are concatenated into one larger feature set, show the algorithm is able to find patterns in the data that remain undiscovered by the traditional features. By capturing these otherwise ignored patterns, classifiers with stacked features are able to consistently perform better predictions than the classifiers using only traditional features.
Stacking features is not reserved specifically for generated features, however, as manually created features can be stacked on top of the traditional features as well. Experiments comparing manually and automatically generated stacked features indicate the algorithm is able to create better features than human experts, making the algorithm a valuable asset for regular expression generation.

In conclusion, the algorithm is capable of detecting recurring patterns in labeled string data. However, for natural language data the features created by the algorithm are not up to par with traditional text features. As the algorithm prefers literal string features for this kind of data and does not find many additional features when stacking the features, it is likely natural language strings like Spam messages and company/individual names do not contain many patterns that cannot be captured directly by traditional text features. Or, conversely, that any additional rules followed by Spam headers and names do not sufficiently fit in the rules of regular expressions. Regardless, as the algorithm was able to find a few unexpected features and its features outperformed those created by the human expert on the PPI dataset, the algorithm could be a useful tool to detect whether recurring patterns exist in text data and generate regex features for them accordingly.

7.2 Future Work

Though the current work shows the proposed algorithm is successful as a proof of concept, many extension can still be made to the algorithm. In this section, several suggestions for continuation of this work will be discussed.

7.2.1 Improvement of the Genetic Programming Algorithm

Evaluation of the regular expressions generated by the genetic programming algorithm reveals the best expressions generally contain little regular expression syntax, especially for the datasets containing natural language. Instead, the algorithm relies mainly on short literal strings, even these strings could be captured by a few simple regular expressions. For example, the algorithm might create the expression $Bv|B.\backslash v|b.\backslash v$, while the more compact regex $[Bb]\.?v$ would be equivalent. Future work could focus on improving the algorithm by, for example, increasing the probability of selecting operator nodes during offspring generation rather than using uniform probabilities.

7.2.2 Experimentation on Additional Datasets

Experimental results on the performance of the regular expression features shown in the current work varied significantly per dataset. Regular expression features created by the algorithm allowed for perfect predictions on the Regex dataset, whereas predictions on the natural language datasets, Spam and PPI, seemed to be lacking. Additional experiments on new datasets of varying difficulty could help further identify the strengths and weaknesses of the algorithm on natural language classification tasks.

7.2.3 Using Traditional Features Rather than Predictions

In the current work, stacked features were created by concatenating classifier predictions, using the traditional features, with generated regular expression features. Though an approach directly stacking the traditional features and regular expressions is more natural, it is also very costly in the evolutionary process as a classifier must be trained for every candidate feature set and the time required to train the classifier scales dramatically with the amount of features. Future work could focus on a method to efficiently combine traditional and regex features during training, providing more flexibility to the algorithm.
Bibliography


