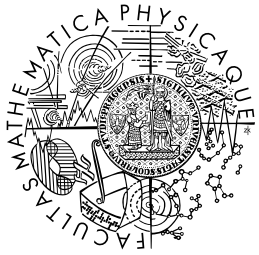

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Ivan Rychtera

Learning picture languages using picture-to-string transformations

Department of Software and Computer Science Education

Supervisor of the master thesis: RNDr. František Mráz, CSc.

Study programme: Computer science

Study branch: Artificial Intelligence

Prague 2023

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague date ....

Author's signature

I would like to thank my supervisor František Mráz for his patience and help with the thesis and the experiments.

Title: Learning picture languages using picture-to-string transformations

Author: Ivan Rychtera

Department: Department of Software and Computer Science Education

Supervisor: RNDr. František Mráz, CSc., Department of Software and Computer Science Education

Abstract: We define a new model for recognizing picture languages using picture-to-string transformations – transcriptor-evaluator machine for picture languages (TEMPL). The model consists of a transcriptor that rewrites an input picture into a string and an evaluator that accepts or rejects the obtained string. We use TEMPL as a representation of picture languages that can be learned from positive and negative examples. The attached implementation of TEMPL and several algorithms for learning regular languages were used for benchmarking the accuracy of training TEMPLs on selected picture languages.

Keywords: picture languages four-way finite automaton machine learning formal languages

Contents

Introduction

In comparison to well-known string (one-dimensional) languages, formal two-dimensional languages are sparsely discussed in the literature. But understanding two-dimensional languages might aid us greatly when dealing with problems concerning images, such as pattern recognition or object detection, or any other data with a regular two-dimensional structure that has some pattern regularity Pradella and Crespi Reghizzi [2008].

In some literature and also throughout this paper, the terms of two-dimensional and *picture* languages will be used interchangeably. To distinguish them from pictures in the wider, common sense, formal picture languages have formally exact mathematical descriptions and are not defined as sets of, for example, images of cars, which cannot be defined exactly. Therefore, also the usage of deep neural networks, which is typically very efficient with recognizing objects in images Krizhevsky et al. [2012], mostly fails to learn picture languages in the formal sense. However, there exist powerful models of automata that work on picture languages but lack the efficiency and determinism needed for more practical applications.

Several works have already been published (Kuboň and Mráz [2020] or Krtek [2014]) that focus on finding methods to learn picture languages from positive and negative examples. This process of learning a model (a grammar) for a target language based on some information about the words of the language is called grammatical inference De la Higuera [2010]. There are multiple known algorithms of grammatical inference for several classes of languages in the one-dimensional space of the problem, but almost no knowledge in two (or more) dimensions.

For grammatical inference of picture languages, we need a suitable representation for pictures and picture languages. The manner in which pictures can be formally represented differs in the literature. One option is generative, which describes the way a picture can be generated from a string. An 8-letter alphabet with moves representing all the 8 directions (north, south, east, west, northeast, southeast, northwest, and southwest) was introduced by Freeman Freeman [1961] and later simplified into a 4-letter (up, down, left, right) alphabet Maurer et al. [1982], that can represent a way in which a picture is drawn. That one was extended by Costagliola et al. [2003] in order to generate colored pictures including labels. In either of these representations, a picture language is a set of strings describing all pictures in the language.

A second way to represent a picture is closer to the common form – it is a rectangular array of symbols that could be interpreted as colors of pixels in the image. In this representation, a picture language is the set of pictures accepted by an automaton working on two-dimensional inputs. An example of such could be the non-deterministic online tessellation automaton Giammarresi and Restivo [1997], the even more powerful sgraffito automaton Průša et al. [2014] and two-dimensional limited context restarting automaton Krtek [2014]. As outlined earlier, the problem of these automata is their high complexity as the problem of deciding whether an input image is accepted by any of them is NP-complete.

In this thesis, we introduce a new representation for picture languages that generalizes the method from the paper Kuboň and Mráz [2020]. Then we use the

new representation for learning picture languages by using algorithms for learning (string) regular languages.

The thesis is structured as follows: Chapter 1 introduces basic definitions for formal languages and methods for recognizing one-dimensional and two-dimensional languages. We introduce also several methods for learning regular languages from positive and negative samples. In Chapter 2 we present our new transcription-evaluation framework that represents a picture language as a transcriptor that rewrites an input picture into a string and the string is accepted or rejected by a one-dimensional (string) automaton. Chapter 3 describes the implementation of the framework and a generator of some benchmark languages. Chapter 4 describes the experiments and the obtained results.

1. Theoretical background

In this chapter, we define the basics of formal languages and automata theory. Then we discuss some of the approaches for representing and learning picture languages.

In the first section, we define the fundamental concepts of formal languages, both one-dimensional and two-dimensional ones. The second section is dedicated to representing string languages using various automata, including ways to obtain an automaton from samples of language. The last section then contains a survey of similar representations for picture languages.

1.1 Formal languages

This section presents basic definitions of strings, pictures and languages and some useful operations we can do with them with an example for the pictures.

Informally, a word in a formal sense is a sequence of symbols. A formal language is then a set of words.

Let Σ be a finite alphabet. A *word* over Σ is any finite sequence of symbols from Σ. The empty sequence of symbols is called *the empty word* and is denoted as ε. We denote the n-th letter in a word w as w_n, starting the sequence at w_0.

We will denote the set of all words over Σ as Σ^*. A set of words of length k is then denoted as Σ^k and a set of words shorter than k is denoted as $\Sigma^{<k}$. A *language* is a subset of Σ^*. Let v and w be words. A *concatenation* of v and w is an operation that produces a new word such that the sequence of letters of v is directly followed by the sequence of letters of w. The concatenation of words v and w we denote as vw.

We say a word y is a *substring* of word w if there exist words x and z such that $w = xyz$. Word y is a *prefix* of w if $x = \varepsilon$. Word y is a *suffix* of w if $z = \varepsilon$.

A formal picture is a table of symbols. There is the same number of symbols in each row and the same number of symbols in each column. Symbols in a formal picture can be thought of as pixels. A *picture language* is a set of pictures.

A *picture* over Σ is a two-dimensional matrix of symbols from Σ. The picture p with a non-negative number of rows m and a non-negative number of columns n is of *dimension* (m, n). A picture with zero rows and zero columns is called *the empty picture* and denoted λ. Pictures with zero rows and non-zero columns or vice-versa are undefined.

A subset is denoted as $\subseteq$, a proper subset is denoted as $\subset$ and a finite subset as $\subset_{fin}$.

In what follows, $\mathbb{N}^0$ and $\mathbb{N}^+$ denote the set of non-negative and positive integers, respectively. $\mathbb{Z}$ denotes the set of all integers. A set of all pictures over the alphabet Σ of dimension (m, n) is denoted as $\Sigma^{m,n}$.

The set of all non-empty pictures $\bigcup_{m,n\in\mathbb{N}^+} \Sigma^{m,n}$ is denoted as $\Sigma^{+,+}$.

The set of all pictures $\bigcup_{m,n\in\mathbb{N}^+} \Sigma^{m,n} \cup \{\lambda\}$ is denoted as $\Sigma^{*,*}$.

The set of all pictures with $n > 0$ columns $\bigcup_{m\in\mathbb{N}^+} \Sigma^{m,n}$ is denoted as $\Sigma^{+,n}$.

The set of all pictures with $m > 0$ rows $\bigcup_{n\in\mathbb{N}^+} \Sigma^{m,n}$ is denoted as $\Sigma^{m,+}$.

In a picture $p \in \Sigma^{+,+}$, we refer to a symbol at the position i, j as $p_{i,j}$. The top-left corner has the position $0, 0$, see Figure 1.1.

We introduce a special border symbol $\# \notin \Sigma$ for any Σ. *The boundary picture* of a picture p of dimension (m, n) is the picture $\hat{p}$ over $(\Sigma \cup \{\#\})$ of dimension $(m+2, n+2)$. $\hat{p}$ has p as a sub-matrix in the rectangle defined by $\hat{p}_{1,1}$ and $\hat{p}_{m+1,n+1}$ and the rest of the symbols are $\#$. An example can be seen in Figure 1.1.

The boundary is useful for automata working on a picture to indicate they are on the edge of the picture and should act accordingly. Without the boundary, we would need to take great care in order to handle cases where such an automaton would go out of the bounds of the picture.

$p_{0,0}$	$p_{0,1}$	$p_{0,2}$
$p_{1,0}$	$p_{1,1}$	$p_{1,2}$
$p_{2,0}$	$p_{2,1}$	$p_{2,2}$

#	#	#	#	#
#	$p_{0,0}$	$p_{0,1}$	$p_{0,2}$	#
#	$p_{1,0}$	$p_{1,1}$	$p_{1,2}$	#
#	$p_{2,0}$	$p_{2,1}$	$p_{2,2}$	#
#	#	#	#	#

Figure 1.1: Picture p of dimension (3,3) on the left and its boundary picture $\hat{p}$ on the right.

1.2 Automata for the recognition of string languages

This section is dedicated to the role of automata in recognizing string languages and grammatical inference. *Grammatical inference* is a problem of how to find (infer) a representation of a target language from some information about the language, e.g., a set of sample words from the language. We look at some algorithms that allow us to estimate an automaton that represents the target language well.

A significant portion of discussed methods for inferring the automata recognizing picture languages (Section 1.3) utilise approaches used for one-dimensional regular languages. Therefore, we recall some core definitions of this domain. Further related definitions can be found in Rozenberg and Salomaa [1997].

Finite automata or finite state machines are one of the simplest computational models used to decide whether a word belongs to a given language. As the name implies, they only have a finite number of possible internal states. Such an automaton reads a word from left to right, letter by letter and uses its transition function to determine the next internal state.

For deterministic automata, if an automaton is in an accepting state after reading the whole input, the input word belongs to the language accepted by the automaton, otherwise, the input word is rejected and belongs to the complement of the accepted language. In general, an automaton needs not have a transition defined for each pair of state and input, implying that if such a pair occurs, the word is automatically rejected by the automaton. However, we will define a deterministic finite automaton that has a complete transition function defined for all pairs of symbols and states.

Definition 1. A deterministic finite automaton *(DFA) is a 5-tuple* $A = (\Sigma, Q, \delta, q_0, Q_a)$*, where* Σ *is an alphabet,* Q *is a finite set of states,* $q_0 \in Q$ *is the initial state,* $Q_a \subseteq Q$ *is a set of accepting states, and* $\delta : Q \times \Sigma \to Q$ *is a transition function.*

An input to a DFA is a word $w \in \Sigma^*$. A configuration of a DFA is a pair (q, s) of a state q and a word s over Σ. The word s represents an unread suffix of the input word.

A computation over an input word w is a sequence of configurations that starts with the initial configuration (q_0, w). Each subsequent configuration is obtained by a step from the previous one.

A step in a computation of the automaton A is a pair of configurations $(q_i, as) \rightarrow (q_{i+1}, s)$ such that $q_{i+1} = \delta(q_i, a); a \in \Sigma, s \in \Sigma^*$. We say that the step is from the configuration (q_i, as) to the configuration (q_{i+1}, s). Note that, for any DFA, from each configuration with a non-empty suffix there exists exactly one possible step.

An iteration of the transition function δ from a state q over a string w of length n is denoted as $\delta^*(q, w)$ and equals to $\delta(\ldots\delta(\delta(q, w_0), w_1)\ldots, w_{n-1})$ if w has a length at least 1, otherwise it equals to q.

A DFA accepts a word $w \in \Sigma^*$ if there exists a computation that starts in the initial configuration (q_0, w) and ends with the last configuration of the form (q, s) where $q \in Q_a$ and $s = \varepsilon$.

1.2.1 Regular inference

A grammatical inference is a process, where we obtain some type of formal language representation from some information about the target language, like a set of sample words. A regular inference is then a special type of grammatical inference, where the inferred language is a regular language, in our case represented by a DFA. Therefore, regular inference is often called learning DFA or learning regular languages. In the whole thesis, we apply regular inference only. We impose this restriction in order to simplify the learning process and to reduce the space of admissible solutions.

We may not always have the exact knowledge of the target language $L \subseteq \Sigma^*$. Therefore, we need to establish a way how to obtain a DFA from two finite sets of sample words (L_+, L_-), where $L_+ \subset_{fin} \Sigma^*$ is a set of words from the language L and $L_- \subset_{fin} \Sigma^*$ is a subset of words from $\Sigma^* \backslash L$. L_+ is called a *positive sample* of L and L_- is called a *negative sample* of L. Obviously, L_+ and L_- are disjoint.

We say that language $L' \subseteq \Sigma^*$ is *consistent* with the sample (L_+, L_-) if $L_+ \subseteq L'$ and $L_- \cap L' = \varnothing$

The idea is that we start with a trivial prefix automaton such as a prefix tree acceptor (PTA, see below) and then progressively merge states of the automaton which are somehow similar. For a given sample (L_+, L_-), the corresponding prefix tree acceptor accepts all words from L_+ and rejects all words from L_-. Hence, PTA is consistent with (L_+, L_-). However, we suppose that the language contains more words. By merging states of PTA we obtain an automaton that is still consistent with (L_+, L_-), but smaller. This not only helps to reduce the complexity of the automaton but ideally should help to find similarities in the sample words in order to discover a more general pattern among them.

The danger here is that the data provided may not hold the crucial information needed to distinguish the language, so the merging algorithm may overgeneralise, in extreme cases even turning the automaton into a constant function.

Derived automata

As previously stated, in many algorithms for learning regular languages the core operation for training an automaton is state merging. If two states serve a very similar role in an automaton, it can be useful to merge them into one in order to help combine the information from each state to generalize the automaton.

We can imagine a scenario where we start with a large automaton with relatively little interconnectivity in its transitions and iteratively merge similar states to get a more consolidated version of the original automaton. This new version should hopefully combine the information from previously unrelated states so that it is able to process words that the original automaton would have rejected on the basis of an undefined transition.

The merges can be formalized as a partition of the automaton. This means defining sets of states which behave as one in a new more consolidated automaton. Splitting or combining these sets gives us a larger or smaller automaton, respectively. This creates a partial ordering where on one side there is a partition where each state is in its own set and on the other is a partition where all states are in a single set.

We will use formal definitions of the search space described in Dupont et al. [1994].

For learning DFA, we need to represent partial knowledge of a target automaton, where we know that the target automaton has a state reachable from the initial state, but we do not know whether the state is accepting or rejecting.

E.g., let $L = \{a, b\}$. The automaton should reach an accepting state after reading ab. But we do not know whether the initial state or the state reached by reading the word a is accepting or rejecting. Such partial knowledge of a DFA can be represented by an incomplete automaton.

Definition 2. *An* incomplete automaton *(IA) a 6-tuple* $A = (\Sigma, Q, \delta, q_0, Q_a, Q_r)$*, where* Σ *is an alphabet,* Q *is a finite set of states,* $q_0 \in Q$ *is the initial state,* $Q_a \subset Q$ *is a set of accepting states,* $Q_r \subset Q$ *is a set of rejecting states and* $\delta : Q \times \Sigma \to Q$ *is a partial transition function. If a state belongs to* Q_a*, we say it has a positive label. Conversely, if it belongs to* Q_r*, we say it has a negative label.*

An incomplete automaton can be reduced to a DFA by disregarding Q_r *and defining all undefined transitions as a transition into a new special non-accepting state. We call the resulting automaton the* complete *DFA of* A*.*

For any set S, a partition π is a set of pairwise disjoint non-empty subsets of S whose union is S. We call each of these subsets a *block*. Let s be an element of S, then we denote the block in π containing s as $B(s, \pi)$. We say that π_i *refines*, or is finer than, π_j if and only if every block of π_j is a union of one or several blocks of π_i.

Now we will introduce a quotient automaton, which is an incomplete automaton obtained by merging some states of an incomplete automaton. The following definition can be considered as a generalization of the quotient automaton used in the reduction of DFAs (Rozenberg and Salomaa [1997]).

Definition 3. *Let* $A = (\Sigma, Q, \delta, q_0, Q_a, Q_r)$ *be an IA and* π *be a partition on* Q *such that for all* $B \in \pi$*, for all* $q, q' \in B$ *and each* $a \in \Sigma$ *it holds that* $B(q', \pi) = B(q, \pi)$ *implies* $B(\delta(q', a), \pi) = B(\delta(q, a), \pi)$ *and no partition can contain states*

from both Q_a and Q_r. Then the quotient *IA $A/\pi = (\Sigma, Q', B(q_0, \pi), Q'_a, \delta')$ is derived from A with respect to the partition π of Q.*
The quotient automaton is then determined as follows:

$$Q' = Q/\pi = \{B(q, \pi) | q \in Q\},$$

$$Q'_a = \{B \in Q' | B \cap Q_a \neq \varnothing\},$$

$$Q'_r = \{B \in Q' | B \cap Q_r \neq \varnothing\},$$

δ' is defined as follows: for all $B, B' \in Q'$ for all $a \in \Sigma$ we define $\delta'(B, a) = B'$ where $B' = B(\delta(q, a), \pi)$ for some $q \in Q$.

In the above definition, the transition function δ' is well defined, as if q_1, q_2 are states from the same block B of π, then $\delta'(B(q_1, \pi), a) = B(\delta(q_1, a), \pi) = B(\delta(q_2, a), \pi) = \delta'(B(q_2, \pi), a)$ for all $a \in \Sigma$.

We say that states from Q that belong in the same block in π are *merged*.

It is easy to see, that the above construction of a quotient automaton preserves consistency with a sample.

Proposition 1. *If IA A is consistent with a sample (L_+, L_-), then A/π is consistent with (L_+, L_-) as well.*

Definition 4 (Dupont et al. [1994])**.** *A positive sample L_+ is called* structurally complete *with respect to an automaton A accepting L_+ if $A = (\Sigma, Q, q_0, Q_a, \delta)$ accepts all elements of L_+ in such a way that:*

1. *every transition of A is used at least once, and*
2. *every element of Q_a (the accepting state set of A) is used as an accepting state for at least one word from L_+.*

Definition 5. *A prefix tree automaton (PTA) $A = (\Sigma, Q, \delta, q_0, Q_a, Q_r)$ for a sample $(L_+, L_-), L_+, L_- \subset_{fin} \Sigma^*$ is an incomplete automaton such that:*

- *Q is the set of all unique prefixes from L_+ and L_- including the empty string.*
- *$q_0 = \varepsilon$,*
- *$Q_a = L_+$, $Q_r = L_-$,*
- *δ is defined in the following way: $\forall w \in Q, \forall a \in \Sigma$, if $wa \in Q$ then $\delta(w, a) = wa$.*

The complete DFA of PTA A accepts a word w if and only if it belongs to L_+.

For processing words, the complete DFA is always used. The main use of the PTA is to obtain an initial IA consistent with (L_+, L_-) that accepts exactly the words from L_+ and rejects all words from L_-. PTA is then used for obtaining a better IA consistent with (L_+, L_-), which then results in a better reduced DFA.

An example of a prefix tree automaton of a target language can be seen in Figure 1.2.

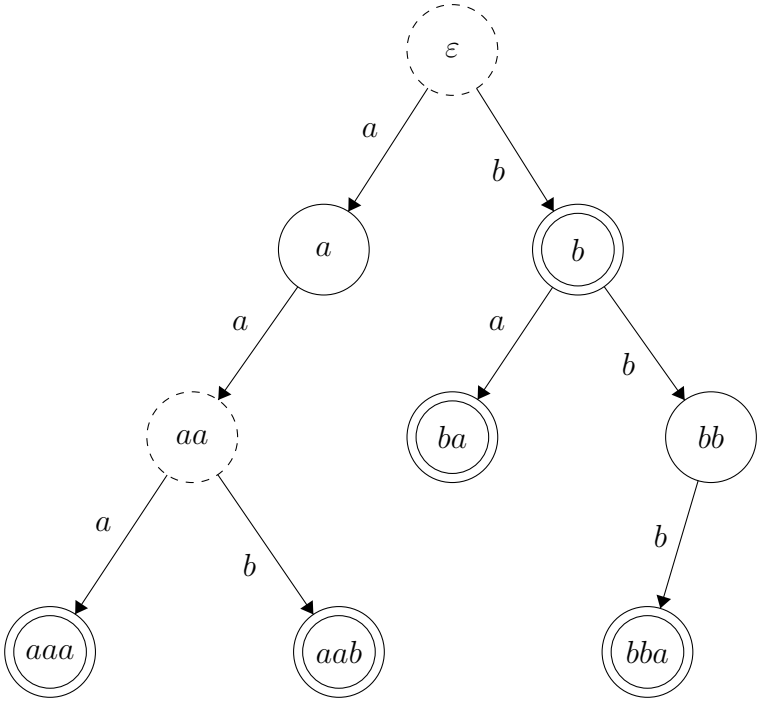

Figure 1.2: The prefix tree automaton for the sample ($\{b, ba, aaa, aab, bba\}$, $\{a, bb\}$). Accepting states are represented by double circles, rejecting states are represented by single circles, and the remaining states are marked by dashed circles. All transitions are drawn as orientated arrows marked by the read symbol.

Definition 6 (Dupont et al. [1994]). *Universal automaton A_u has a single state and accepts all the strings defined over the alphabet Σ, i.e. $L(A_U) = \Sigma^*$, and it is the smallest automaton with respect to which every sample of Σ^* is structurally complete.*

Let $P(A)$ denote the set of all partitions of the state set of an automaton A. Let $r(\pi_i)$, or simply r_i, denote the number of blocks of the partition π_i. Let $\pi_1 = \{B_1, \ldots, B_{r_1}\}$ and π_2 be two partitions from $P(A)$. We say that π_2 directly derives from π_1 if the partition π_2 can be constructed from π_1 as follows: $\pi_2 = \{B_j \cup B_k\} \cup (\pi_1 \backslash \{B_j, B_k\})$, for some j, k between 1 and r_1, $j \neq k$. Consequently, $r_2 = r_1 - 1$.

This derivation operation defines a partial order relation on $P(A)$, which we shall denote $\preceq$. In particular, we have $\pi_1 \preceq \pi_2$. Let $\ll$ denote its transitive closure. In other words, $\pi_i \ll \pi_j$ if and only if π_i is finer than π_j. By extension, we say that A/π_i is finer than A/π_j and that A/π_j derives from A/π_i. By construction of a quotient automaton, we have the language inclusion property which may be reformulated as follows: if $\pi_i \ll \pi_j$ then $L(A/\pi_i) \subseteq L(A/\pi_j)$ and we write $A/\pi_i \ll A/\pi_j$.

Proof. *Let w be a word accepted by $L(A/\pi_i)$. It must hold that the computation of A on w ends in a state q that belongs to an accepting block $B \in \pi_i$. If B is in π_j, then w is accepted by $L(A/\pi_j)$. If B is not in π_j, then it could only be merged with other accepting blocks from π_i, therefore q belongs in an accepting block of π_j and $w \in L(A/\pi_j)$.* □

The set of automata partially ordered by the relation $\ll$ is similar to a lattice,

that we shall denote $Lat(A)$, of which A and A_u (the universal automaton) are the null and universal elements, respectively.

The depth of an automaton A/π in $Lat(A)$ is given by $N - r(\pi)$, where N is the number of states of A. Consequently, the depth of the automaton A in $Lat(A)$ is equal to 0 while the depth of the universal automaton A_U is equal to N.

This defines a search space that we can use for generalization.

Definition 7. *Let A be a DFA. An antistring $\overline{as} \subset Lat(A)$ is a set of automata such that any element of $\overline{as}$ is not related by $\preceq$ with any other element of $\overline{as}$ in $Lat(A)$.*

Definition 8. *An automaton is said to be at a maximal depth in a lattice of automata, if there is no automaton A' that may be derived from A such that $L(A') \cap L_- = \varnothing$.*

Definition 9. *The border set $BSP_{PTA}(L_+; L_-)$ is defined as the antistring in $Lat(PTA(L_+))$, of which each element is at a maximal depth.*

Consequently, the border set of a lattice is the set of automata that correspond to the theoretical limit of the generalization we can reach while consistently rejecting the negative sample.

1.2.2 Algorithms for regular inference

This subsection is dedicated to algorithms for learning finite automata described by Tîrnăucă [2012]. The algorithms start with prefix automata in order to obtain a good DFA.

For many of the algorithms, it is useful first to define the merging operation. The aim of the merge is to eliminate a state from the automaton in a way that preserves as much of the original structure as possible.

Let a state p be merged into state q. This is possible only if one of the two states is neither accepting nor rejecting or if both p, q are accepting or both p, q are rejecting. All transitions that would lead to p are changed so that they now result in q. Next, if the p has a label (positive or negative), we copy it onto q, which always either has the same label or none at all. Finally, we handle the outgoing transitions of p. Each outgoing transition must have a different symbol that triggers it. If there is a corresponding transition triggered by reading the same symbol in q, we recursively merge the states resulting from the transitions, and the eliminated and the target states are assigned with respect to the originating state of the transition.

Special considerations are unlabeled nodes and undefined transitions. The issue is that the sub-tree rooted at p might contain information that is not present in the sub-tree of q. This might either be transitions that are undefined or states which do not classify an example from the sample. Discarding such information can easily lead to a degraded automaton.

Definition 10. *Given an incomplete automaton $A = (\Sigma, Q, \delta, q_0, Q_a, Q_r)$, we say that two states p and q are* distinguishable *in A if there exists a word $u \in \Sigma^*$ such that ($\delta(p, u) \in Q_a$ and $\delta(q, u) \in Q_r$) or ($\delta(p, u) \in Q_r$ and $\delta(q, u) \in Q_a$). Otherwise, p and q are* not distinguishable *in A.*

There are two modifications to the original algorithm. If p is not distinguishable from q, then:

- labels of labeled nodes in the sub-tree rooted at p must be copied over their respective unlabeled nodes in the sub-tree rooted at q, and
- transitions in any of the nodes in the sub-tree rooted at p which do not exist in their respective node in the sub-tree rooted at q must be spliced in the sub-tree rooted at p.

A pseudocode for merging states p and q of an incomplete automaton A can be seen in Algorithm 1.

Algorithm 1 Merge

```
function MERGE(p, q, A = (Σ, Q, δ, q_0, Q_a, Q_r))        ▷ A is an incomplete automaton, p and q are states from Q
    for all t ∈ Q, a ∈ Σ incoming transitions to p do
        if δ(t, a) = p then
            δ(t, a) ← q
        end if
    end for
    if (p ∈ Q_a & q ∈ Q_r) or(p ∈ Q_r & q ∈ Q_a) then
        abort merging
    end if
    if p ∈ Q_a then
        insert q into Q_a
    else
        if p ∈ Q_r then
            insert q into Q_r
        end if
    end if
    for all a ∈ Σ do
        if δ(p, a) ≠ ∅ then
            if δ(q, a) ≠ ∅ then
                A ← merge(δ(p, a), δ(q, a), A)
            else
                δ(q, a) ← δ(p, a)
            end if
        end if
    end for
    remove p from A
    return A
end function
```

The result of the merge operation is either rejection of the merge (the states p and q are distinguishable and cannot be merged) or a new IA A' with the number of states lower than the number of states of A. The merge operation has the following important property: if A is consistent with (L_+, L_-), then A' is consistent with (L_+, L_-) as well.

The Trakhtenbrot and Barzdin algorithm

The Trakhtenbrot and Barzdin algorithm produces a DFA from a sample set of words in time $O(mn^2)$ where m is the initial number of states of the PTA consistent with (L_+, L_-) and n is the final number of states of the resulting DFA. The sample needs to be *complete*, which means it has to include all words up to a certain length. This ensures the prefix tree does not contain any undefined transitions, apart from the leaves, since if a word is in the sample, then all its prefixes are in the sample too and we know their labels.

The algorithm merges eligible – not distinguishable – states in the breadth-first order.

We define U as a set of unique nodes; nodes that are pairwise distinguishable. The Trakhtenbrot and Barzdin algorithm starts by adding the root of the automaton A to the set of unique nodes. Then, it visits each proceeding node p of the prefix tree automaton in breadth-first order and compares the sub-tree rooted at p with the sub-tree rooted at each node from the unique nodes set. If p is pairwise distinguishable from each node from U, the algorithm adds p to U. Otherwise, if a state $q \in U$ is found so that p and q are not distinguishable the algorithm updates the automaton A by merging state p into state q. The algorithm halts when all states are inspected. The final content of U is the set of states of the learned final automaton obtained as the complete automaton of A.

Traxbar

Traxbar is an extension of the Trakhtenbrot and Barzdin algorithm so that it can be used with an incomplete set of sample words and still maintain consistency. Some inner nodes of PTA can be without label, i.e. they correspond to a word for which we do not know whether they belong to the target language or not.

The algorithm produces an automaton consistent with the training set. However, it may have some nodes unlabeled. In the experiments (Chapter 4), we consider the unlabeled states as rejecting. This is done under the assumption that any language in question is sparse and there is a much greater likelihood that a randomly chosen input will not belong to the language.

It is worth noting that during its run, the algorithm slowly increases the density of transitions and labelled nodes in the automaton. This results in the merges being increasingly rarer as the constraints for two states to be not distinguishable get stricter with time. If the greedy algorithm makes poor decisions early on, it is not possible to remedy them. Conversely, if the algorithm is able to guess good initial merges, the resulting automaton can be highly accurate.

Traxbar has no way of knowing which merges can lead to a robust result. This concern is addressed in the next algorithms.

For the complete pseudo-code of the Traxbar algorithm see Algorithm 2. In the algorithm, the Boolean function distinguishable(p, q, A) returns true, if states p and q are distinguishable in the automaton A.

Evidence-driven state merging

Evidence-driven state merging (EDSM) is an attempt to better distinguish between possible merges. It computes a score for each pair of states in the current

Algorithm 2 Traxbar algorithm

```
function TRAXBAR(L+, L−)        ▷ L+, L− is a pair of sets of positive and negative samples of an unknown language over alphabet Σ.
    A ← PTA(L+, L−)
    U ← {ε}                     ▷ ε is the initial state of A
    while p visits each proceeding node of A in the breadth-first order do
        dist ← true
        for all q ∈ U do
            dist ← distinguishable(p, q, A)
            if not dist then
                break
            end if
        end for
        if dist then
            append p to U
        else
            A ← merge (p, q, A)
        end if
    end while
    return A
end function
```

set of states in order to determine the best candidates for each merge, rather than simply merging the first admissible pair it finds.

The score is computed by crawling down the subtrees rooted at the two states p and q in consideration. If two nodes x and y are found, that have the same path to their respective node p or q, then the algorithm looks at the labels of x and y. If they match, the score is increased by 1. If they are different, then p and q are distinguishable and the score is set to $-\infty$.

The number of matching labels should be the evidence that the merge is a correct one.

There are two potential problems with the EDSM algorithm. The first is that computing the score for each pair during each iteration is computationally expensive. The algorithm has a time complexity $O(m^4n)$ where m is the initial number of states and n is the final number of states. We need $O(m^3)$ steps to find the best score for each merge and we will do $O(m)$ merges. And in an edge case, the process can be executed for each of the n final states.

The second problem is using sparse data. When language is more complex, it means that we need more words in the sample to capture this complexity. In case we do not have an adequate number of samples, sub-trees of the prefix tree automaton will rarely have a similar structure and capture the same suffix. And when the suffixes rarely match, it is hard for the algorithm to find any evidence of whether or not to merge two states. Thus the scores might be almost always zero making the evidence component of the algorithm much less significant.

For the complete pseudo-code of the EDSM algorithm see Algorithm 3.

Algorithm 3 EDSM algorithm

```
function EDSM(L+, L−)   ▷ L+, L− is a pair of sets of positive and negative samples of an unknown language over alphabet Σ.
    A ← PTA(L+, L−)
    repeat
        max ← −1
        for all pairs (p, q) ∈ (Q × Q) do        ▷ Q is the set of states of A
            compute score(p, q)
            if score(p, q) > max then
                max ← score(p, q)
                p_max = p
                q_max = q
            end if
        end for
        if max > −1 then
            A ← merge(p_max, q_max, A)
        end if
    until no merge is possible
    return A
end function
```

Windowed EDSM

Due to the complexity of EDSM, it is unsuitable for use with larger datasets. One possibility to tackle this problem is to introduce a window of states in which the scores will be computed. The window is a subset of states that go from the root in breadth-first order up to a limit, which is specified at the beginning and can increase if there are no possible merges. The initial window size can be adjusted so that it balances the computational time demand and the thoroughness of the search for the best merge.

The introduction of the window restricts the search space and decreases the time complexity of the algorithm so that it is closer to $O(m^3n)$ rather than $O(m^4n)$, assuming the window size will not increase with the size of the input for the same language. However, this approach still has the weakness to a prefix tree automaton, in which sub-trees have vastly different structures.

For the complete pseudo-code of the Windowed EDSM algorithm see Algorithm 4.

Locally k-testable languages

This method assumes that a given language is k-testable, for some positive integer k. The class of k-testable languages is a subclass of regular languages that is learnable from positive samples (De la Higuera [2010]).

A k-testable language L is characterized by a set T of strings of length k, a set of prefixes and a set of suffixes with length $k-1$, and a set of accepted words shorter than k. This means that each word from L longer than k has only substrings from T and a prefix and a suffix from the limited sets of possibilities. Therefore any word can simply be checked using the sets if it fulfills these

Algorithm 4 Windowed EDSM algorithm

function WEDSM($L_+, L_-, winsize$) ▷ L_+, L_- is a pair of sets of positive and negative samples of an unknown language over alphabet Σ.
$winsize$ is the initial window size
  $A \leftarrow PTA(L_+, L_-)$
  $W \leftarrow \{\varepsilon\}$ ▷ ε is the initial state of A
  repeat
    while size(W) $< winsize$ and $\{W \neq Q\}$ **do**
      get the next node q of A in the breadth-first order
      add q to W
    end while
    $max \leftarrow -1$
    for all pairs $(p, q) \in (W \times W)$ **do**
      compute $score(p, q)$
      if $score(p, q) > max$ **then**
        $max \leftarrow score(p, q)$
        $p_{max} = p$
        $q_{max} = q$
      end if
    end for
    if $max > -1$ **then**
      $A \leftarrow$ merge(p_{max}, q_{max}, A)
    else
      $winsize \leftarrow 2 * winsize$
    end if
  until no merge is possible
  return A
end function

conditions.

Definition 11 (Kuboň and Mráz [2020]). *A (string) language $L \subseteq \Sigma^*$ is called k-testable if there exist four finite sets of words: $I \subseteq \Sigma^{k-1}, F \subseteq \Sigma^{k-1}, C \subseteq \Sigma^{<k}$, and $T \subseteq \Sigma^k$ such that a word belongs to L if it is from C, or its prefix of length $k-1$ is in I, its suffix of length $k-1$ is in F, and all substrings of length k belong to T.*

Given a set of positive samples, we can extract the set of all present prefixes and suffixes of length $k-1$ as I and F respectively, the set of substrings of length k as T and a set of words shorter than k letters as C. We can then check any new word using this knowledge base to decide if it belongs to the language in question.

This method does not utilize the automata derivation method. However, the various sets can easily be thought of as a nondeterministic finite automaton that is equivalent to the following regular expression: $C \cup \{I\Sigma^* \cap \Sigma^* F \cap (\Sigma^* - \Sigma^*(\Sigma^k - T)\Sigma^*)\}$. Therefore it is equivalent to a DFA.

1.3 Automata recognizing picture languages

There are many known models of automata recognizing picture languages. Most of them work directly on two-dimensional inputs. In this section, we introduce several such models that were published in various papers.

Four-way finite automaton

One of the first models of automata for recognizing picture languages is the deterministic four-way finite automaton (4DFA) (Blum and Hewitt [1967]).

The main difference in operation between 4DFA and DFA is the set of possible movements for the head. The head position determines which symbol will be read next. The 4DFA does not read the input in a sequence but determines the head movement with its transition function. Given the state and symbol, the transition function returns a new state and a direction where the head will move. During its computation on an input picture p, the head must not leave the boundary picture $\hat{p}$.

Definition 12 (Giammarresi and Restivo [1997]). *A* deterministic four-way automaton *(4DFA) is a 7-tuple $A = (\Sigma, Q, \delta, \Delta, q_0, q_a, q_r)$, where Σ is an alphabet, Q is a finite set of states, $\Delta = \{\mathrm{r}, \mathrm{l}, \mathrm{u}, \mathrm{d}\}$ is the set of directions, $q_0 \in Q$ is the initial state, $q_a \in Q$ is the accepting state, $q_r \in Q$ is the rejecting state, and $\delta : (Q \backslash \{q_a, q_r\}) \times \Sigma \to Q \times \Delta$ is the transition function.*

An input to a 4DFA is a picture $p \in \Sigma^{*,*}$. The automaton works on the boundary picture $\hat{p}$ of p. A configuration of a 4DFA is a pair of a state and the position (x, y) on the boundary picture $\hat{p}$.

A transition is a pair of configurations $(q_i, (x_i, y_i)) \to (q_{i+1}, (x_{i+1}, y_{i+1}))$ such that $\delta(q_i, p_{x_i, y_i}) = (q_{i+1}, d)$ for some $d \in \Delta$, and the position (x_{i+1}, y_{i+1}) is determined by d and equals to:

- $(x_i + 1, y_i)$ if $d = \mathrm{d}$,

- $(x_i - 1, y_i)$ if $d = \mathrm{u}$,
- $(x_i, y_i - 1)$ if $d = \mathrm{l}$, and
- $(x_i, y_i + 1)$ if $d = \mathrm{r}$.

In case the configuration position reaches the boundary, the next transition has to happen in the opposite direction.

A computation of 4DFA on an input picture p is a sequence of configurations such that the initial configuration is always $(q_0, (1, 1))$ on the boundary picture $\hat{p}$. Each subsequent configuration is obtained by a transition from the previous one. The computation ends when the automaton reaches either the state q_a or q_r because there are no possible steps from any of these states. If the final state is q_a, the automaton accepts p, otherwise, it rejects p.

Note that the 4DFA does not have to visit every position in the picture in order to accept it.

There is a possibility that a computation may reach neither the accepting nor the rejecting state. This happens if the computation reaches the same configuration twice. The total number of different possible configurations can be obtained by multiplying the number of possible states and positions of the head. So if the computation for an automaton with s states on a boundary picture $\hat{p}$ of size (m, n) takes longer than $s \cdot m \cdot n$ steps, we terminate the computation and say the automaton rejects the picture p.

Unfortunately, such a condition cannot be checked by the 4DFA itself. The steps need to be counted externally.

Returning finite automaton and boustrophedon automaton

A returning finite automaton and boustrophedon automaton presented by Fernau et al. [2018] work on a picture p' which is a picture p with added special symbols $\#$ on both sides of each row. The returning finite automaton reads each row left to right and then top to bottom, while otherwise working similarly to a DFA. The boustrophedon automaton then alternates between left-to-right and right-to-left movements.

Definition 13 (Fernau et al. [2018])**.** *A deterministic* returning finite automaton *(RFA) is a 6-tuple $M_r = (\Sigma, Q, \delta, q_0, Q_f, \#)$, where Σ is an alphabet, Q is a finite set of states, $q_0 \in Q$ is the initial state, $Q_f \subset Q$ is a set of accepting states, and $\delta : Q \times (\Sigma \cup \{\#\}) \to Q$ is a transition function and $\#$ is a special symbol not in Σ.*

The definition differs from the source since we find the original definition needlessly complicated for our purpose, but our definition results in an equivalent class of accepted picture languages. The original definition includes rewriting symbols visited by the automaton with a special symbol $\square \notin \Sigma \cup \{\#\}$. However, the rewritten symbols serve just for determining the next visited position of the picture. Our definition avoids rewriting.

A configuration of an RFA is a triple $(q, (i, j), p')$, where q is the current state, (i, j) is the current position in the picture p'. The initial configuration is $(q_0, (0, 1), p')$.

A computation is a sequence of configurations that starts with the initial configuration and given a configuration $(q, (i, j), p')$ the next one is always $(\delta(q, \#), (i + 1, 0), p')$ if $j + 2$ is equal to the width of p', or $(\delta(q, a), (i, j + 1), p')$, where a is the symbol at (i, j) of p', otherwise. Note that the RFA does not read the right border.

M_r accepts picture p, if any computation on p' ends with configuration $(q, (m-1, n-2), p')$, where $q \in Q_f$, m is the height of p' and n is the width of p'.

Definition 14 (Fernau et al. [2018]). *A deterministic* boustrophedon automaton *(BFA) is a 6-tuple $M_r = (\Sigma, Q, \delta, q_0, Q_f, \#)$, where Σ is an alphabet, Q is a finite set of states, $q_0 \in Q$ is the initial state, $Q_f \subset Q$ is a set of accepting states, and $\delta : Q \times (\Sigma \cup \{\#\}) \to Q$ is a transition function and $\#$ is a special symbol not in Σ.*

A configuration of a BFA is a triple $(q, (i, j), p')$, where q is the current state, (i, j) is the current position in the picture p'. The initial configuration is $(q_0, (0, 1), p')$.

A computation is a sequence of configurations that starts with the initial configuration. For a current configuration $(q, (i, j), p')$:

- For an even i, the next is always $(\delta(q, \#), (i + 1, j), p')$ if $j + 2$ is equal to the width of p', or $(\delta(q, a), (i, j + 1), p')$, where a is the symbol at (i, j) of p', otherwise.
- For an odd i, the next configuration is $(\delta(q, \#), (i + 1, 0), p')$ if $j = 1$, or $(\delta(q, a), (i, j - 1), p')$ otherwise.

Again, note that the BFA skips the terminating # on each line.

M_r accepts picture p, if any computation on p' ends with configuration $(q, (m-1, j), p')$, where $q \in Q_f$, m is the height of p' and $j = 1$ if m is even, or $j + 1$ equals the width of p' for m odd.

Restarting tiling automaton

The restarting tiling automaton was introduced by Průša and Mráz [2013]. It combines two approaches. The first is rewriting tiles of the picture according to non-deterministic rewriting rules. The second is restarting, which aims to iteratively shorten an input until either there is a "correct" short input, or the algorithm gets stuck and rejects the input. This shortening is done by repeatedly scanning the input and doing one rewrite before restarting. Each rewrite must somehow shrink the input (which we define later).

A restarting tiling automaton utilizes a scanning strategy to determine the order in which all positions of an input picture are visited. A scanning strategy must satisfy a requirement that each position on the picture must be visited exactly once.

A scanning strategy for this automaton consists of a starting point, which is one of the corners of the picture, and a function $f(i, j, m, n)$, where (i, j) is a current position on the picture and (m, n) is the size of the picture, that determines the next position to be visited.

Definition 15 (Průša and Mráz [2013]). *A* two-dimensional restarting tiling automaton *(2RTA) is a 6-tuple* $M = (\Sigma, \Gamma, \Theta_F, \delta, \nu, \mu)$*, where* Σ *is a finite input alphabet,* Γ *is a finite working alphabet* $(\Sigma \subset \Gamma)$*,* $\Theta_F \subset (\Gamma \cup S)^{2,2}$ *is a set of accepting tiles,* $\nu = (c_s, f)$ *is a scanning strategy,* $\mu : \Gamma \to \mathbb{N}^+$ *is a weight function and* $\delta \subset \{(U \to V) | U, V \in (\Gamma \cup S)^{2,2}\}$ *is a set of rewriting rules such that in every rule* $u \to v$ *only a single position of* u *is changed and moreover if* $a \in \Gamma$ *is rewritten into* $b \in \Gamma$*, then* $\mu(b) < \mu(a)$*. A* deterministic *2RTA (2DRTA) is a 2RTA* $M = (\Sigma, \Gamma, \Theta_F, \delta, \nu, \mu)$ *with the set of rewriting rules* δ *satisfying the additional condition that for every tile* $T \in (\Gamma \cup S)^{2,2}$ *there exists at most one rule in* δ *with the left-hand side* T*.*

Symbols from $\Gamma \backslash \Sigma$ are called auxiliary symbols and must not be contained in any input picture.

The automaton M works in phases and restarts after each phase. In each phase, it starts at a staring position (i_0, j_0) on the boundary picture $\hat{p}$. The starting position is one of the corners determined by the scanning strategy ν. M scans this position using a 2-by-2 window and determines whether there is a rewriting rule for the contents of the scanning window. If there is none, the automaton moves to the next position specified by the scanning strategy. Once a suitable rule is found, the whole window is rewritten according to one of the rules applicable to the scanned tile, and the automaton restarts.

If the whole boundary picture has been scanned and no applicable rule exists for any of the tiles, the final verification is done by checking whether each 2-by-2 tile of the picture belongs in Θ_F, i.e. if the boundary picture belongs in the local language of Θ_F. If so, M accepts the picture $\hat{p}$, otherwise, it rejects it.

Two-dimensional online tesselation automaton

The two-dimensional online tesselation automaton can be thought of as filling out a table with the size of the picture. It starts in the top-left corner of the picture and works its way in diagonal lines to the bottom-right corner. The state of each cell in the table is determined solely by the contents of the cells directly above and to the left and the symbol at the corresponding place in the picture. The input which would be out of bounds is replaced by a default state.

Definition 16 (Giammarresi and Restivo [1997]). *A* deterministic two-dimensional online tesselation automaton *(2DOTA) is a 5-tuple* $A = (\Sigma, Q, \delta, q_0, F)$*, where* Σ *is an alphabet,* Q *is a finite set of states,* $q_0 \in Q$ *is the initial state,* $F \subseteq Q$ *is the set of accepting states, and* $\delta : (Q \times Q \times \Sigma \to Q)$ *is the transition function.*

A computation of 2DOTA on a picture $p \in \Sigma^{m,n}$ consists of assigning a state from Q to each position (i, j) on p. The state is determined according to the transition function from the states assigned to positions $(i-1, j)$ and $(i, j-1)$, and the symbol $p_{i,j}$.

The computation starts by assigning the initial state q_0 to each position in the first row and the first column in the boundary picture $\hat{p}$. Then follow the $m+n-1$ steps. In the first step, the state $\delta(q_0, q_0, p(0,0))$ is assigned to $(0,0)$. In the next step, the states are assigned to positions $(0,1)$ and $(1,0)$ simultaneously. In the k-th step, a state is assigned to each position (i, j) such that $i + j + 1 = k$.

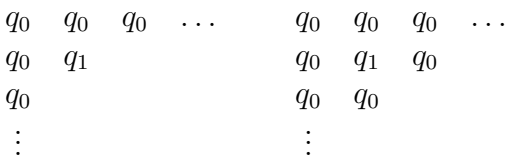

Figure 1.3: The first two steps of a 2DOTA automaton.

A 2DOTA accepts a picture if there is an accepting state assigned to position $(m-1, n-1)$ on p.

Example 1. *Let* $A = (\{1\}, \{q_0, q_1\}, q_0, \{q_1\}, \delta)$*, where* $\delta(q_0, q_0) = q_1$ *and* $\delta(q_1, q_0) = \delta(q_0, q_1) = \delta(q_1, q_1) = q_0$*, be a 2DOTA. Then the first two steps of the computation for any picture can be seen in Figure 1.3.*

2. Our model

This chapter is dedicated to methods that first transform pictures into strings, and then use a string automaton to recognize them. The idea is to leverage our understanding in tackling problems in the domain of one-dimensional languages to help us in the more complex domain of picture languages.

In Section 2.1 we review the motivation for the framework. In Section 2.2 we define the framework and in Section 2.3 we show some properties of the framework.

2.1 Preliminary developments

A string representation for pictures was already proposed in the past. Maurer et al. [1982] represented a picture as a sequence of movements that a pen would make on a plane to draw the picture. The word would then use the alphabet $\Pi = \{u, d, l, r\}$, representing the up, down, left, and right movements, respectively. Such a picture will always be connected, so a possible extension to the alphabet can be made so that it contains symbols for raising and lowering the pen.

Another approach was introduced by Kuboň and Mráz [2020]. This approach is further developed in this thesis. Some results obtained in this thesis are included in the paper Kuboň et al. [2023]. In the article, various methods of transcribing the pictures into strings are used to then utilize an algorithm to construct a DFA to classify the resulting strings.

Rather than a simple row-by-row or column-by-column transcription into a string, the proposed transcription methods have used a peculiar order, which concatenated together the contents of 3-by-3 windows of the picture separated by a special symbol. This, the authors theorized, should have led to a better generalization of other simpler approaches. Nevertheless, the authors suggest more research should be done on more complex strategies for mapping.

2.2 Transcription-evaluation framework

This thesis aims to expand on the mapping from picture to string. This section is therefore dedicated to presenting a model that can accommodate various approaches to first transcribing a picture into a string and then using an algorithm that works with one-dimensional languages.

The most general model for the two-step approach that we use here is the TEMPL.

Definition 17. *Let Σ and Γ be alphabets and Σ does not contain the symbol $\#$. Then* transcriptor-evaluator machine for picture languages *(TEMPL) is a pair $M = (T, E)$, where T is a map from $(\Sigma \cup \{\#\})^{**}$ to Γ^* and E is a string automaton accepting a language $L(E) \subset \Gamma^*$.*

We say that M accepts a picture language $L(M) = L(T, E) = \{p \in \Sigma^{**} \mid T(p) \in L(E)\}$.

There are two components in TEMPL $M = (T, E)$ that correspond to the steps in the approach. The transcriptor T takes a picture and outputs a string,

and the evaluator E decides if a string belongs to a string language. An arbitrary string automaton, like a DFA, can serve as the evaluator. In the next section, we will discuss transcriptors.

A similar picture-string transducer was studied by Otto and Mráz [2015]. The paper studied mainly models that scan an input picture row-by-row in a left-to-right manner using a head that scans a single symbol at a time. Here, we consider more general rewriting of pictures into strings.

2.2.1 Transcriptor

In simple terms, the transducer is an arbitrary picture-to-string mapping. Further developments that improve the performance may be discovered in the transcription process (such as more complex scanning strategies suitable for a given task).

In this thesis, we will limit the transcription to a two-part scanner. First, a scanning strategy is devised using a simple automaton, and then a constant dictionary is used to map fixed-size factors of the picture into fragments of a string in the determined order. We call these parts *scanning strategy* and *sequence dictionary*. Thus scanner is a pair $N = (S, D)$, where S is a scanning strategy and D is a sequence dictionary.

Scanning strategy

A scanning strategy is a mechanism that provides a scanning sequence, i.e. the order in which various pixels should be processed. The scanning sequence is a sequence of positions, around which the scanning window is centered. For this thesis, we limit the scope of possible scanning strategies to four-way scanner automatons (4SA, see below).

4SA works similarly to a 4DFA, but in each step, it also decides whether to add the current position to the output scanning sequence.

Formally, a scanning sequence for a picture p of dimension (m, n) is a finite sequence, where each element is a pair of non-negative integers $(i, j); 0 \leqslant i \leqslant m + 1, 0 \leqslant j \leqslant n + 1$.

Definition 18. *A four-way scanner automaton (4SA) is a system* $M_s = (Q, \Sigma, \Delta, q_0, q_h, \delta)$*, where* Q *is a set of states,* Σ *is an input alphabet,* $\Delta = \{\ell, \mathrm{r}, \mathrm{u}, \mathrm{d}\}$*,* q_0 *is the starting state,* q_h *is the halting state and* $\delta \subset Q \times (\Sigma \cup \{\#\}) \to Q \times \Delta \times \{position, \varepsilon\}$ *is the transfer function.*

Whenever the 4SA M_s is in the state q and scans a symbol $a \in (\Sigma \cup \{\#\})$ and $\delta(q, a) = (q', d, position)$ for some $q' \in Q, d \in \Delta$, the automaton appends its current position on the border picture $\hat{p}$ to its output. If $\delta(q, a) = (q', d, \epsilon)$, nothing is added to the output.

In this thesis, we will place some additional constraints on the 4SA. First, we require that the transfer function δ does not depend on the contents of the input picture. For all $q \in Q$ and for all $a, b \in \Sigma$ it holds that $\delta(q, a) = \delta(q, b)$. The 4SA can only differentiate if it reads a symbol from the picture or a border symbol. This restriction is in place to preserve the function of the scanning strategy. A more powerful automaton would be able to do some precomputing on the picture, which is beyond the scope of this thesis. And second, the output scanning

sequence must contain each position of the picture p exactly once. Hence, it can contain also some positions on the border picture $\hat{p}$.

Configuration and transition relations of the 4SA are similar to the ones of 4DFA. However, the 4SA in each transition determines whether to output the current position on the boundary picture $\hat{p}$. The output is part of the configuration as well.

A computation in 4SA is a sequence of configurations where the initial configuration is always $(q_0, (1, 1))$, the whole input, and an empty sequence representing the output. Each subsequent configuration is obtained by a step from the previous one. The computation ends when the automaton reaches the state q_h. The output of the computation of 4SA M_s on an input picture p, denoted as $M_s(p)$, is a scanning sequence. We call the elements of $M_s(p)$ *anchors*.

Evidently, the scanning sequence produced by M_s on an input picture p depends only on the dimensions of the picture p.

Sequence dictionary

A sequence dictionary is responsible for using the scanning sequence to transform a picture into a string. It is a map that takes as an argument some part of the picture and outputs a substring that is to be appended to the resulting string. The part of the picture is specified by a window w, that is a sequence of positions $(r_1, d_1) \dots (r_\ell, d_\ell)$. If an anchor has a position (a_r, a_d) in the picture, the symbols from positions$(a_r+r_1, a_d+d_1)...(a_r+r_\ell, a_d+d_\ell)$ are concatenated and then mapped to words over an output alphabet using a dictionary.

Since the dictionary is working with a finite alphabet $\Sigma \cup \{\#\}$ and a finite window, there exist only finitely many possible combinations of symbols in the window, so the map itself can be a simple table.

Definition 19. *A sequence dictionary is a tuple $D = (\Sigma, \Gamma, w, t, k)$ where Σ is the input alphabet, Γ is the output alphabet, $w = ((r_1, d_1) \dots (r_\ell, d_\ell))$ for $\ell \geqslant 0, r_i, d_i \in \mathbb{Z}$ is a sequence of relative positions of length ℓ and $t : (\Sigma \cup \{\#\})^\ell \to \Gamma^k$ is a map. k is a constant.*

The sequence dictionary takes as an input a picture p and a scanning sequence S. For each element (i, j) of S, it takes the symbols of p specified in w relatively to the anchor (i, j) (takes # if the position is outside p) and creates the word $p_{i+r_1,j+d_1}, \dots, p_{i+r_\ell,j+d_\ell}$ from $(\Sigma \cup \{\#\})^\ell$. Then it maps this word using t to a string from Γ^k and appends the string to the output. The output transcription of dictionary D for a picture p and scanning sequence S is a word over Γ denoted as $D(p, S)$.

Perhaps the most apparent relative position sequence to try is one that encompasses a 3-by-3 window around the anchor. In theory, this arrangement should retain the information about the neighboring symbols in all directions, while still keeping the size of the output alphabet manageable. For instance, a sequence dictionary can simply concatenate all symbols from the 3-by-3 window and output them as a word of length 9. Alternatively, such contents of a 3-by-3 window can be mapped to a single symbol from an alphabet of size $(|\Sigma| + 1)^9$.

2.3 TEMPL properties

In this section, we show the TEMPL working on an example language (Section 2.3.1) and we demonstrate some properties of the model (Section 2.3.2).

2.3.1 Example

Example 2. *Suppose a sequence dictionary* $D = (\{a\}, \{0,1\}, ((-1,-1),(-1,0),(0,-1),(0,0)), t, 4)$*, where for every word* $v \in \{a,\#\}^4$ *it holds* $t(v) = 1$ *if* $v = \#\#\#a$ *and* $t(v) = 0$ *otherwise. The sequence of the dictionary corresponds to an upper-left 2-by-2 square relative to the anchor.*

Then suppose p is the picture of dimensions (2,2) containing the letter a. (See Figure 2.1.) Let ((1, 1), (2, 2)) be a prefix of a scanning sequence. The sequence dictionary maps the corresponding words ###a and aaaa into the output word '10'.

$\#_1$	$\#_2$	#	#	#	#	#	#
$\#_3$	a_4	a	#	#	a_1	a_2	#
#	a	a	#	#	a_3	a_4	#
#	#	#	#	#	#	#	#

Figure 2.1: On the left the window $w = ((-1,-1),(-1,0),(0,-1),(0,0))$ anchored at position $(1,1)$ on the boundary picture for $p = a^{2,2}$. On the right, the window is anchored at $(2,2)$ on the same picture. The indices denote the order in which the symbols will be concatenated for the sequence dictionary.

Example 3. *Let* $L \subset \{0,1\}^{*,*}$ *be the chessboard language where each picture is a rectangle with the chessboard pattern. Each pair of neighboring positions (horizontally and vertically) contains different symbols and the symbol in the top left corner is 1. A visualization can be seen in Figure 2.2. Let us construct a TEMPL accepting the picture language L.*

We define a 4SA S, which will return a scanning sequence going left to right on odd rows and right to left on even rows.

$$M_s = (\{C_l, D_l, F_l, C_r, D_r, F_r, H\}, \{0,1\}, \Delta, F_r, H, \delta).$$

$$\delta(F_r, 0) = \delta(F_r, 1) = (C_r, \mathrm{r}, pos),$$

$$\delta(F_r, \#) = (H, \mathrm{r}, \varepsilon),$$

$$\delta(F_l, 0) = \delta(F_l, 1) = (C_l, \ell, pos),$$

$$\delta(F_l, \#) = (H, \ell, \varepsilon),$$

$$\delta(C_r, 0) = \delta(C_r, 1) = (C_r, \mathrm{r}, pos),$$

$$\delta(C_r, \#) = (D_r, \ell, \varepsilon),$$

$$\delta(C_l, 0) = \delta(C_l, 1) = (C_l, \ell, pos),$$

$$\delta(C_l, \#) = (D_l, \mathrm{r}, \varepsilon),$$

$$\delta(D_r, 0) = \delta(D_r, 1) = \delta(D_r, \#) = (F_l, \mathrm{d}, \varepsilon),$$

$$\delta(D_l, 0) = \delta(D_l, 1) = \delta(D_l, \#) = (F_r, \mathrm{d}, \varepsilon).$$

Further we can define a simple sequence dictionary $D = (\{0,1\}, \{a,b\}, ((0,0)), t, 1)$ *where* $t(0) = a, t(1) = b$. *Let* $E = (\{a,b\}, \{O, I, reject\}, \delta, O, \{O, I\})$ *be a deterministic finite automaton with the following transition function:*

$$\delta(O, b) = I,$$

$$\delta(O, a) = reject,$$

$$\delta(I, a) = O,$$

$$\delta(I, b) = reject,$$

$$\delta(reject, a) = \delta(reject, b) = reject.$$

Using 4SA S, *sequence dictionary* D *and* E *as an evaluator, we can build TEMPL* $M = ((M_s, D), E)$.

Claim: The TEMPL $M = ((M_s, D), E)$ *accepts the picture language* L.

Proof. *First, we will prove that each picture from* L *is accepted by* $M = ((M_s, D), E)$. *For an input chessboard picture* p *from* L, *the 4SA* M_s *always adds a position to a scanning sequence, unless it reads* # *or unless it has moved in the exact opposite direction than in the previous step. Therefore, each element of the scanning sequence will be a direct neighbor of the previous element. Due to the definition of* L *and the fact that the map in* D *is a bijection, no two symbols can repeat in a resulting string from* $D(p, M_s(p))$ *for any word from* L. E *rejects a word if and only if it reads two identical symbols in a row. Thus* $D(p, M_s(p))$ *is accepted by* E *and* $L \subset L(M)$.

Next, we will show that each picture accepted by M *belongs to* L. *For a contradiction, let us assume a picture* p *is accepted by* M *and it contains two identical symbols next to each other horizontally. Since* S *moves left to right or right to left from border to border before doing only one step down, any two horizontal neighbors would result in a substring* aa *or* bb *in* $D(p, M_s(p))$. *Because* E *is going to read two identical symbols in a row, it rejects* p, *which is a contradiction with* $p \in L(M)$.

Let us now assume a picture p *is accepted by* M *and it contains two identical symbols next to each other vertically. Since we have already established that if* $p \in L(M)$, *no two horizontal neighbors are identical, the same vertical neighbors imply the two rows with identical vertical neighbors have to be identical as a whole.*

Because S *makes a step down on the picture only after coming back to the position it has added to the scanning sequence last, the next added position will be a direct vertical neighbor. And since the computation ends only when the lower border is reached, there is at least one pair of vertical neighbors next to each other in the resulting string* $D(p, M_s(p))$ *for each pair of neighboring rows. Thus* $D(p, M_s(p))$ *is going to contain two identical symbols in a row, therefore it will be rejected by* E, *which is again a contradiction with* $p \in L(M)$.

Altogether, we have shown that in a picture $p \in L(M)$, *no two neighboring (horizontally or vertically) symbols are identical. The top left corner of* p ($p_{0,0}$)

is mapped to b if and only if $p_{0,0} = 1$. *As E only accepts words with prefix b, in the picture p there are no identical neighboring symbols and the top left corner of p is 1. Hence p is a chessboard pattern and always belongs to L.* □

1	0	1	...	0	1
0	1	0	...	1	0
1	0	1	...	0	1
⋮	⋮	⋮	⋱	⋮	⋮
0	1	0	...	1	0
1	0	1	...	0	1

Figure 2.2: A template for the chessboard language.

2.3.2 Computational power

Here we show that any returning finite automaton or boustrophedon automaton can be simulated by a TEMPL.

Equivalent TEMPL

Let $M_r = (\Sigma, Q, \delta_r, q_0, Q_f, \#)$ be a RFA. We will construct a TEMPL $T = ((M_s, D), E)$ such that it accepts the same picture language as RFA M_r.

The scanner M_s needs to transcribe each row from left to right, top to bottom. Rows need to be separated by the # symbol because it signals the RFA that it has reached the end of a row.

Let $M_s = (\{B, G, W, C, L, H\}, (\Sigma \cup \{\#\}), \Delta, W, H, \delta)$ be a 4SA, where δ is defined as follows:

$$\delta(W, a) = (W, \mathrm{r}, pos); \forall a \in \Sigma,$$

$$\delta(W, \#) = (B, \mathrm{l}, \varepsilon),$$

$$\delta(B, \#) = (L, \mathrm{d}, \varepsilon),$$

$$\delta(C, a) = (W, \mathrm{r}, pos); \forall a \in \Sigma,$$

$$\delta(C, \#) = (H, \mathrm{r}, \varepsilon),$$

$$\delta(L, \#) = (C, \mathrm{r}, \varepsilon), \text{ and}$$

$$\delta(B, a) = (B, \mathrm{l}, \varepsilon); \forall a \in \Sigma.$$

$D = (\{(\Sigma \cup \{\#\}), (\Sigma \cup \{\#\}), (0, 0), e, 1)$ is a sequence dictionary, where e is the identity map.

$E = (\Sigma, Q, \delta_r, q_0, Q_f)$ is a DFA with the same components as RFA M_r.

Theorem 1. *For each RFA* M_r, *there exists a TEMPL* T *such that* $L(T) = L(M_r)$.

Proof. *Because D is a direct one-to-one mapping, E will process input in the order specified by M_s. And since E uses the same parameters as M_r, we only need to prove that the scanning sequence provided by S on $\hat{p}$ is the same as the scanning order of an RFA on p.*

M_s starts in the position $(1,1)$ of $\hat{p}$, which is equal to the position $(0,1)$ on the picture p'. If the picture is not empty it scans the whole second (first non-border) row excluding the borders. It then returns to the first column and moves down to the state L.

In each row, if S is in state L at the left border, it returns one # symbol and then each non-border position on that row if there are any, otherwise it halts and outputs nothing. And because the only vertical movement is from B, it always arrives at a new row in L.

Because B can only be reached through a move to the left, it will always move M_s down in the leftmost column (at the left border of $\hat{p}$). That is the only position M_s can read # in B. Therefore M_s will always arrive at a new row in the leftmost column and will output the first position if the next symbol is not #.

Because W can only be reached through a move to the right, it will only transfer to B in the rightmost column. Therefore M_s will read each line it arrives on completely from left to right and has to report each position. And because M_s can only move one line at a time, it will read each line beside the first in order from top to bottom.

The last line of $\hat{p}$ is not output, because a line is only output if it does not begin with two #s.

Therefore the input of E is exactly p' row-by-row with exactly one # symbol between each row, the same as the reading order of an RFA. And thus $L(T) = L(M_r)$. □

Corollary 1. *For each BFA M_b, there exists a TEMPL T such that $L(T) = L(M_r)$.*

Proof (Theorem 1)**.** *According to Fernau et al. [2018], for every BFA there exists an equivalent RFA. Therefore, we can infer from the Theorem 1, that there exists an equivalent TEMPL.* □

3. Implementation

In this chapter, we describe the implementation of the TEMPL model presented in the previous chapter. We need an implementation to experimentally investigate the model. We require our implementation to be flexible and highly modular to allow for easy swapping of various components to compare them and even add new ones.

All experiments in this thesis have been done using this Python framework, which can be found in the electronic attachment. The framework allows the user to evaluate custom models for learning picture languages using picture-to-string transformation. The framework consists of two parts, the TEMPL itself and a picture language generator. The framework is built to resemble the theoretical model as much as possible to allow for maximum modularity and also includes algorithms for learning regular languages for evaluators. The picture language generator can be used to generate positive and negative samples for selected benchmark languages.

The first section of this chapter is dedicated to the TEMPL model and its components. The second chapter discusses the language generator and the last is an add-on for experiments.

3.1 TEMPL

This section is a top-down overview of the model (Section 2.2) implementation itself and its components: the transcriptor and the evaluator.

The basic functionality allows the user to make a custom combination of a transcriptor and an evaluator. It can either be used for constructing acceptors for picture languages or for inferring TEMPLs from sample pictures similarly to machine learning libraries such as `skicit-learn` [sckit-learn website]. Using `fit_evaluate` it can be utilized for benchmarking experiments.

The design is modular. It requires a transcriptor, an evaluator, and optionally a training strategy with minimal restrictions. This design is chosen so that parts of the framework can be easily replaced for testing more varied approaches.

The TEMPL class itself has methods `fit` and `predict`.

`fit` takes a training method and a language description in the form of a `yaml` configuration file for the language generator. This configuration is used to call the generator. Since transforming the picture into a string creates another intermediate dataset file, which can be re-used if the transformation process is computationally demanding, it makes sense for the TEMPL object to handle work with its files directly. It also allows the TEMPL class to work with datasets from other sources than the generator. The proper format is discussed in Section 3.2.

After creating or reading the proper dataset, the method fits the evaluator and returns the time it took for the evaluator to be trained.

A diagram of this process can be seen in Figure 3.1.

1. The generator generates sets of samples of a language together with their labeling specifying whether a given picture belongs to the selected picture language or not.

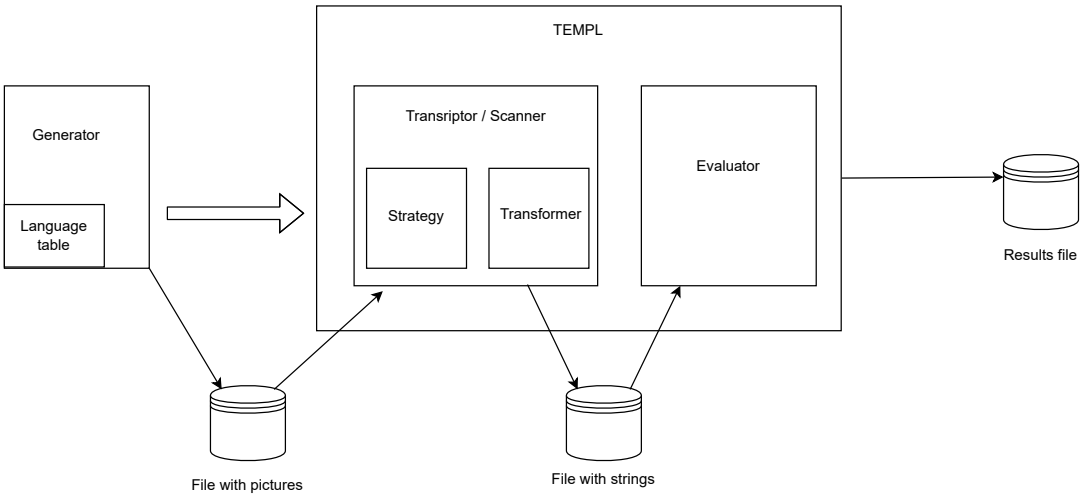

Figure 3.1: A diagram of the TEMPL pipeline for learning picture languages. The generator stores its output in a text file, where TEMPL can read it. After transcription, the obtained strings are once again stored in a text file. This allows for skipping the generation and transcription upon repeated calls.

2. The file with pictures is processed by Transcriptor/Scanner that transforms pictures into strings and stores them in "File with strings".

3. The evaluator reads the string file and produces a representation of the string language using the strings from the file as a sample.

`predict` takes a string and calls the `predict` method of the evaluator.

An example usage can be seen in Listing 3.1. Line 1 is creating an example picture, line 2 is defining the location of the configuration file, line 3 defines a window, and line 4 creates a Scanner object using an implemented strategy and the transformer class. Line 5 creates the TEMPL class with the Scanner object and the default Evaluator object, line 6 fits the model using the parameters specified in the configuration file. In line 8 we use the model to decide if the picture belongs in the language accepted by the TEMPL object.

The various components are described in more detail in the following parts. The configuration can either be a `.yaml` file or a Python dictionary and contains various parameters for the generator and the training process.

Listing 3.1: A basic usage of the TEMPL class. In this example, a Templ instance is created, trained and used to classify a 2-by-2 picture.

```
1 picture = [[0,1],[1,0]]
2 language = "config.yaml"
3 window = [(-1,0),(0,0),(1,0)]
4 scanner = Scanner(Row_by_row(), Transformer(window))
5 model = Templ(scanner,Evaluator())
6 train time = model.fit(language)
7 #language is configuration filename or dictionary
8 prediction = model.predict(picture)
9 #picture is a two-dimensional Python list
```

3.1.1 Trancsriptor

The transcriptor is an abstract class, which handles the conversion from picture to string. This class allows an arbitrary picture-to-string mapping in case the user would want a more complex transcription. However, in our proposed less general model, the transcriptor can be implemented as a scanner (Section 2.2.1).

3.1.2 Scanner

The scanner is an implementation of the transcriptor. It handles the picture-to-string transcription as described in the TEMPL model. It contains two modular components – a strategy and a transformer – to allow easy modifications and extensions.

Strategy

Strategy (Section 2.2.1) is an interface that only requires implementing a method for getting an anchor sequence for a given picture. There are no further limitations, but all our strategies are equivalent to 4SAs (Section18).

There are four implemented strategies:

1. row-by-row – each row is read left to right from top to bottom,
2. snake-by-row – each row is read, alternating between left to right and right to left, from top to bottom,
3. column-by-column – each column is read top to bottom from left to right,
4. snake-by-column – each column is read, alternating between top to bottom and bottom to top, from left to right.

None of these strategies returns positions on the border of the border pictures.

Transformer

The transformer is a wrapper class for the dictionary representing the map t as well as the size k of the output string of the sequence dictionary (Section 2.2.1) in the theoretical model. It handles the correct translation of the anchor sequence through a Python dictionary. The user can either provide a dictionary, or the class can use the Horner scheme to create a dictionary dynamically if arbitrary symbols can be used.

The only required parameter for the transformer is the window, which is a list of pairs of integers, specifying the relative positions of the pixels relative to the anchor, that the dictionary reads.

3.1.3 Evaluator

Evaluator provides a class for a string classifier. It has a simple interface with two methods. Its aim is again to imitate the interface of models in common libraries. The Evaluator is automaton based, but the interface can be used for any string classifier.

`fit` takes two sets of strings as input (the sets of positive and negative examples) and in its default version builds a prefix tree classifier and then uses a given training method to train it. The default training method is traxbar (Definition 1.2.2).

Prediction takes a string and uses the trained automaton to output a label. An exception is thrown if no model has been trained.

The simplified code of the evaluator class and an example can be seen in Listing 3.2. Line 2 is the definition of the prediction method, line 4 defines a method to get the size of the evaluator DFA. Lines 8 to 14 define the fit function, where line 12 first constructs the PTA (Definition 5) and uses the specified method (or traxbar) to fit the complete DFA. Lines 17 to 19 are the definition of positive and negative samples and lines 20, 21, and 22 are the definition, fitting of the samples and predicting an example word using the model of the evaluator object, respectively.

Listing 3.2: A simplified code of the Evaluator class and a simple usage example.

```
1  class Evaluator:
2      def predict(self,string):
3          return eval(self.automaton,string)
4
5      def get_model_size(self):
6          return self.automaton.number_of_nodes()
7
8      def fit(self,data, training_method = None):
9          if training_method is None:
10             training_method = traxbar
11         self.automaton = \
12             construct_tree(data["pos"], data["neg"])
13         training_method(self.automaton)
14         return
15
16
17 data = {"pos": {"00", "11"},
18         "neg": {"01", "10"},
19         }
20 evaluator = Evaluator()
21 evaluator.fit(data)
22 prediction = evaluator.predict("00")
```

Automaton

The automaton is represented by a directed graph using the networkx library. This choice has been done due to the flexibility of nodes and edges in the implementation as well as various options to access them. Each node needs to be able to hold a label, while edges have to represent a very complex transition function, all while the graph has to be open to any potential manipulation by the training algorithm.

During the construction of the prefix tree automaton, the name of each node corresponds to the prefix it represents. Each node has a `label` attribute that

indicates the presence and the value of the label. The transition function is then realized through `trans` attribute, which is a set of letters on which this transition should be followed. The python set has been chosen due to the need to quickly merge them and check for the presence of the letters.

Training methods

The training methods currently implemented in the framework are traxbar, windowed EDSM, and locally testable languages. The original EDSM is a special case of the windowed EDSM, where the initial window size is equal to the size of the automaton.

Traxbar and windowed EDSM are functions, that take one argument: the networx prefix three. Windowed EDSM has a second optional argument specifying the initial window size. They return the trained automaton.

The LTL is a child class of the evaluator. It can either be provided with the four sets of strings (respective to I, F, C, T defined in Section 1.2.2) during construction or can build its representation from a provided dataset.

3.2 Generator

In order to simplify testing applications, the implementation includes a language generator. The generator can provide datasets of various properties for some pre-defined languages of binary pictures. It serves as an interface for inputting custom datasets. It can either be run on its own or as a part of the whole TEMPL pipeline. The output of the generator are already bordered pictures from the desired language in order to remove this responsibility from the model itself.

We differentiate between two types of languages: crisp and noisy. The generator behaves slightly differently depending on the type of language being generated. A crisp language tends to be smaller and strictly defined, e.g., in a picture from the chessboard language, no two neighboring pixels have the same symbol.

Noisy languages are derived from crisp ones. Each picture belonging to a noisy language must not differ from a crisp example by more than a given noise threshold. The threshold indicates a percentage of pixels that can be different compared to the closest crisp sample.

The generator first establishes a sufficiently large set of possible examples to be included in the final datasets. We will call the set of sample pictures a pool. The pool includes examples for all height and width combinations within the limits specified in a configuration file; each combination has a different number of its representatives. This number is a function of an area of the picture of that size. The function is defined as two times the area of the picture to ensure compatibility with previous experiments. The function, however, can be redefined.

For crisp languages, only negative samples are randomly generated.

Each negative (and other positive for noisy languages) example is generated by selecting an appropriate number of flips from a uniform distribution. Two randomly selected pictures of the same size will most likely differ in about half of the pixels. The uniform distribution ensures that there is a significant representation of pictures that differ in almost no pixels or almost all of them. Then

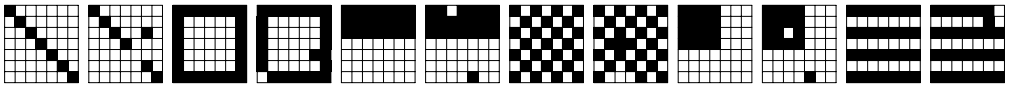

Figure 3.2: Sample pictures from languages $L_1, \ldots, L_6$ showing crisp and noisy versions.

a crisp positive example is taken and the pixels to be flipped are again selected uniformly.

Negative examples are discarded if they accidentally become positive. For crisp languages, this means simply comparing the picture to each positive example of given dimensions. For noisy languages, the number of differing pixels is counted for each of the crisp positive pictures of the same dimensions and then compared to whether it exceeds the noise threshold.

From this pool, train and test sets are uniformly randomly chosen. If there are too few positive examples, negative examples are used to preserve the total number of examples. The complementary problem of too few negative examples can easily be overcome by using the complement of the original language for the generation. It is reasonable to assume that for each sufficiently large m, n the target language is likely going to have much fewer possible pictures of dimension (m, n) than its complement in the vast majority of cases, hence we prefer lower code complexity.

The generator offers the following languages:

L_1 is the set of all white rectangles containing a black diagonal till the border of the picture. The diagonal can start in either of the top corners.

L_2 is the set of all white pictures of dimensions at least $(3, 3)$ with a black border of one-pixel width.

L_3 is the set of all pictures with a positive number of black rows followed by a positive number of white rows.

L_4 is the set of all pictures with a regular chessboard pattern of black and white pixels. The top-left corner of such a picture can contain either black or white pixels, but the whole picture must have the chessboard pattern.

L_5 is the set of all pictures where the top left quadrant (rounded up) is black, while the rest are white.

L_6 is the set of all pictures with alternating black and white rows. The first row can be either black or white.

An example from each language is shown in Figure 3.2.

The architecture allows an easy addition of new languages due to unified wrappers at each step of the generation.

Furthermore, the generator enables the application of some operations, rotations, and flips, to the language if the operations are permitted for the language. Languages resulting from the operations can be arbitrarily combined to allow for great flexibility in the tasks. The permitted operations are set in a configuration

CSV file (`language.csv`) and can be changed. For the languages currently supported by the generators, all operations are admissible. In the experiments, some equivalent operations were disabled. Otherwise, the distributions of potential examples would be skewed.

A sample of the language operations configuration follows. The first line is a header, the second line starts with the name of the language followed by Boolean values, where True means that the operation from that column can be applied to the language in that line.

```
language,include_noise,rot90,rot180,rot270,mirror,updown,transpose
chessboard,True,False,False,False,True,False,True
```

In the case of adding a new language to the generator, permitted operations can be configured by adding a corresponding line to the CSV configuration file.

3.2.1 Usage

As stated, the generator can either be used within the Templ pipeline, in which case it is handled by the Templ class, or run on its own from the command line. In the second case, it takes one parameter, which is a path to a `yaml` configuration file. An example follows:

```
python generator_with_noise.py --config configs\my_config.yaml
```

The `yaml` configuration file contains parameters of a given task, such as picture size limits, noise threshold, and language operations to perform and include in the sets. The format is a key value dictionary separated by a colon. A commented example can be found in the electronic attachment.

The output of the generator consists of two text files containing a train set and a test set. Each picture begins with a line indicating the membership in the target language and an ID. Then follows the picture with symbols divided by spaces and rows divided by line breaks. Example output follows. It shows the picture with ID 1 that belongs to the target language and the picture with ID 2 that does not belong to the target language.

```
1 ID: 1
# # # # # # #
# 1 0 1 0 1 #
# 0 1 0 1 0 #
# 1 0 1 0 1 #
# # # # # # #


0 ID: 2
# # # # # # #
# 1 1 1 1 1 #
# 0 0 0 0 0 #
# 0 1 1 1 0 #
# # # # # # #
```

3.3 Running experiments

The basic experimental usage is facilitated by the `fit_evaluate` function. The user must provide a Scanner and Evaluator implementation, a language configuration, and optionally a learning algorithm to `fit_evaluate`. The function creates a TEMPL from the components Transformer and Evaluator, fits it to a training set, tests the trained model on a test set, and outputs various metrics into a CSV file, all specified in the `yaml` configuration file. The metrics are F1 score on both train and test datasets, accuracies for both positive samples and for negative samples on both train and test datasets, train and test time, and the number of states in the final automaton provided the Evaluator uses one.

A simple example of how to call `fit_evaluate` can be seen in Listing 3.3.

Listing 3.3: main

```
1 def main():
2     window = [(-1,0),(0,0),(1,0))]
3     scanner = Scanner(Row_by_row(), Transformer(window))
4     fit_evaluate(scanner,Evaluator(),config_file,edsm)
```

The resulting CSV file can be further processed for evaluating the performance of the given combination of components. We will present such an evaluation in the next chapter.

4. Experiment setup

Here we present the results of experiments where we compare different TEMPL models with different sets of parameters on learning benchmark languages.

We prepared two sets of experiments. The first part is dedicated to comparing performance using different string alphabets, the second aims to compare different combinations of scanning strategies and algorithms.

4.1 Alphabet size

Some early experiments have shown that rather than simply rewriting the symbols in the original picture into one dimension, we can achieve higher performance by transcribing the picture into another, larger alphabet. This alphabet should reflect the contents of the 3-by-3 windows and substantially reduce the depth of the prefix tree for a DFA learning algorithm to process.

The first set of experiments is thus dedicated to comparing scanning the picture using the 3-by-3 window and scanning it symbol by symbol.

4.1.1 Datasets

For our experiments, we selected sample picture languages. The set of sample picture languages comprised the six languages from the generator introduced in the previous chapter. For each language, we generated a pool of positive and negative samples.

The data for each experiment were randomly selected from the randomly generated pool. The pool consists of pictures of various sizes; pictures of larger sizes are more common in order to accommodate the larger space needed to be sampled. Positive and negative examples are handled separately. Positive examples with zero noise are always included in the pool.

The specific training and testing sets required for our experiments were generated using the generator described in Chapter 3 and are fully reproducible. Sample sets comprised 100, 200, 400, 800, 1600, and 3200 pictures for each sample language, ranging from dimension 5 to dimension 10. Each of the sets contained the same number of positive and negative examples if possible. Of course, for experimenting with k-locally testable (string) languages we have used only positive examples in the training sets.

4.1.2 Learning Setup

In the first set of experiments, we stick to a single scanning strategy – Row-by-row and a scanning window of size 3-by-3.

For learning k-locally testable language we use learning positive examples inspired by Akram et al. [2010]. For learning from sets containing both positive and negative examples we use the state merging algorithm `traxbar`.

Once the learning is finished, the resulting automata are tested on independent test sets of pictures, which are rewritten into strings using the same transcriptor. As our sample languages are rather sparse, we do not report accuracy as it usually

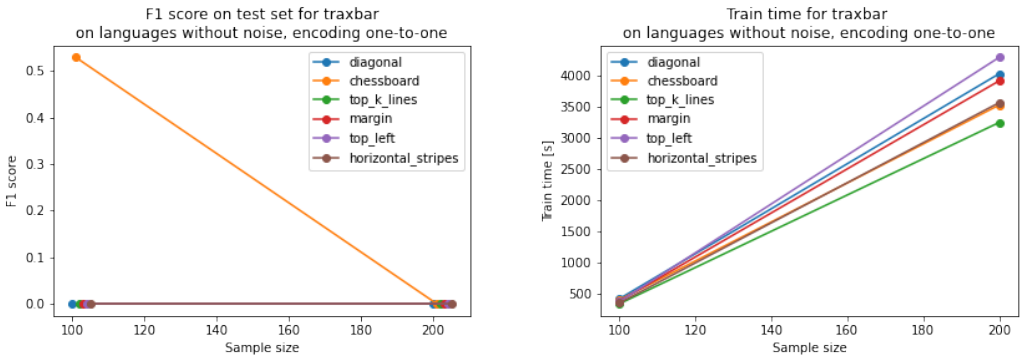

Figure 4.1: Results of training finite automata using `traxbar` on sample sets with 100 and 200 samples in train/test sets with the one-to-one encoding of the contents of the scanning window. In the left plot, the markers are shifted along the x axis to help readability.

considerably differs between positive and negative samples. Instead, we use F_1-score defined as

$$\frac{Precision \cdot Recall}{Precision + Recall}, \text{ where } Precision = \frac{TP}{TP + FP} \text{ and } Recall = \frac{TP}{TP + FN},$$

where TP, FP, and FN stand for the number of true positive, false positive, and false negative samples, respectively.

4.1.3 One-to-one encoding of window contents

At first, we considered the same encoding of the contents of a scanning window into a string as in Kuboň and Mráz [2020]. All symbols within the window are rewritten into a string row-by-row. That way on each step of the scanning the contents of the scanning window produced a string of length 9. Repeatedly, the window moved by one position to the right and the current contents of the scanning window were rewritten into a string of length 9. In contrast to Kuboň and Mráz [2020], we did not separate the consecutive contents of the scanning window by any separator. Further, we will call this encoding one-to-one.

Unsurprisingly, such encoding of the contents of the scanning window produces long strings and training `traxbar` on a set of strings obtained from quite small samples of pictures was prohibitively slow (see Figure 4.1) with a terrible accuracy (F_1-score close to zero). From the plots, we can see that the combination of one-to-one encoding and `traxbar` is unusable.

Conversely, combining one-to-one encoding and the learning of k-locally testable languages is feasible (see Figure 4.2). When using k-locally testable languages, we must choose also a proper order k of the locally testable languages. In the figure, there are plotted results of experiments with $k = 2, 4, 6, \ldots, 20$ with the language L_1 of diagonals. The best F_1-score was achieved for $k = 18$ and $k = 20$. The results for the other sample languages were also promising and were obtained in a very short training time, too.

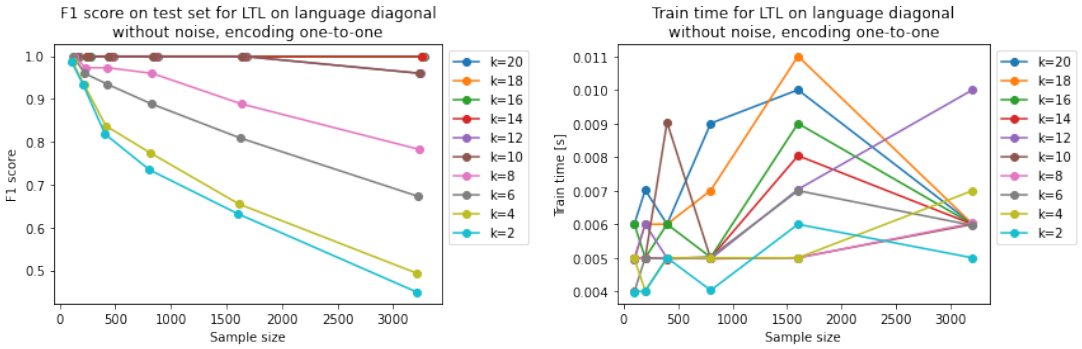

Figure 4.2: Results of training finite automata using k-locally testable languages on sample sets with 100, 200, 400, 800, 1600, and 3200 samples in train/test sets with the one-to-one encoding of the contents of the scanning window. F_1-score on test sets on the left and time for training on train sets on the right. The markers for different values of k are shifted a little to show overlapping marks.

4.1.4 Many-to-int encoding of window contents

In the rest of our experiments, we use an encoding that converts the contents of the whole scanning window (9 symbols) into a single symbol from a bigger alphabet. In our experiments, the number of possible contents of the scanning window has an upper limit of $3^9 = 19683$, as one field of the window can contain a white pixel, black pixel, or the border marker #. Of course, not all combinations of pixels and border markers are possible, but still, the alphabet is quite large. Therefore, we encode one symbol of the alphabet simply as an integer. Further, we will call this encoding many-to-int.

As the train and test sets are generated randomly, the resulting F_1-score and training time are not constant. Therefore in the following, we will plot results obtained by averaging F_1-score and train time from 10 randomly generated train and test sets for each size. For illustrating variance in the achieved results, we use error bars of length equal to the sample standard deviation of the measurements.

Using many-to-int encoding, `traxbar` produced a reasonable F_1-score for all tested crisp languages and also for noisy languages. See the plots in Figure 4.3, 4.4. We can observe that except for the noisy version of the language L_5, the training time for `traxbar` is not higher than 300 seconds up to the sample size of 3200. F_1-score is between 0.5 and 0.8 which is quite good for mostly sparse languages in our samples.

Next, we experimented with learning k-locally testable languages when using many-to-int encoding. The obtained results are surprising. At first, we examined how the F_1-score is influenced by the value of k. In Figure 4.5 and Figure 4.6, we can see that for the sample language L_1 and $k = 2$, the resulting average F_1-score is the highest.

Similarly, for other sample languages, the value $k = 2$ together with many-to-int encoding is the best combination. Interestingly, the value $k = 2$ with many-to-int encoding corresponds exactly to $k = 18$ with one-to-one encoding.

For the rest of our sample languages, we can see the performance of learning crisp sample picture languages and noisy sample picture languages using 2-locally testable languages in Figures 4.7 and 4.8. For all crisp sample languages, the F_1-

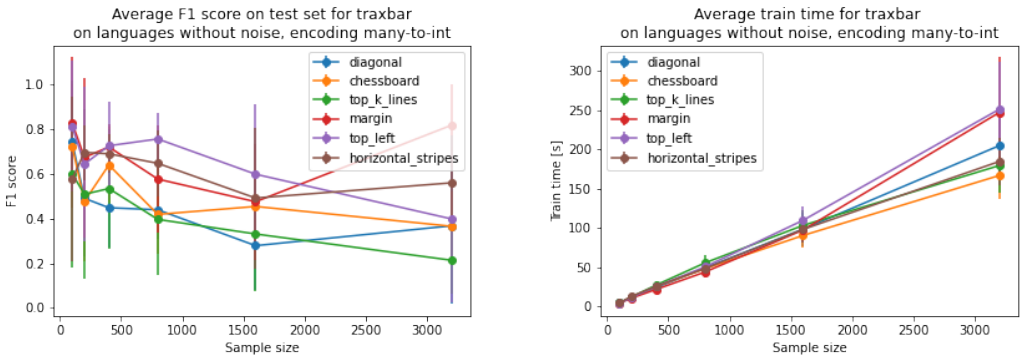

Figure 4.3: Results of training finite automata for crisp sample languages using `traxbar` with the many-to-int encoding of the contents of the scanning window. Average F_1-score on test sets on the left and average time for training on train sets on the right. The length of the error bars is the sample standard deviation.

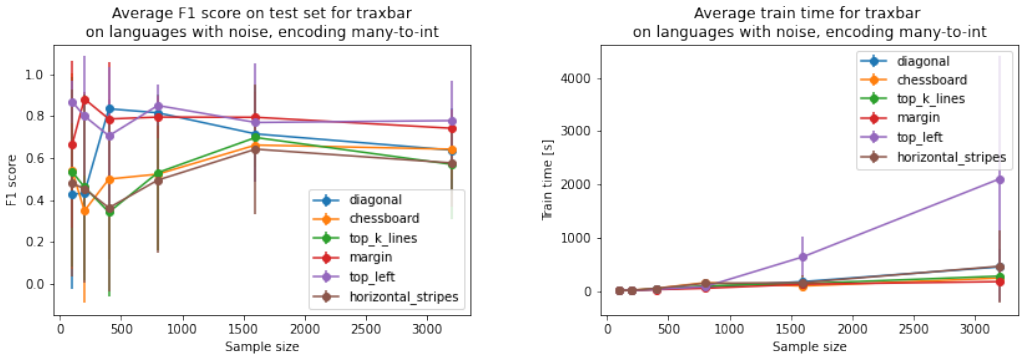

Figure 4.4: Results of training finite automata for noisy sample languages using `traxbar` with the many-to-int encoding of the contents of the scanning window. Average F_1-score on test sets on the left and average time for training on train sets on the right. The length of the error bars is the sample standard deviation.

score is close to 1, except for the sample language L_1 (of diagonals) for which it achieves 0.88 only. Probably it is caused by the very low number of positive samples. For all noisy sample languages including the noisy version of L_1, the F_1-score converges to a value around 0.95 for the growing size of the training sample. The convergence is very stable which can be seen from very short error bars.

Additionally, we can see a linear growth of the time required for training 2-locally testable languages with respect to the growing size of the training sample. Simultaneously, the training time is low. Lower than 0.03 seconds even for sample sets of size 3200.

4.2 Algorithm comparison

The second part of the experiments is dedicated to exploratory analysis of the algorithms. We use the same data as in the first set of experiments to compare various combinations of scanning strategies and learning algorithms. This should give us some insight into how the components influence the behavior of the model.

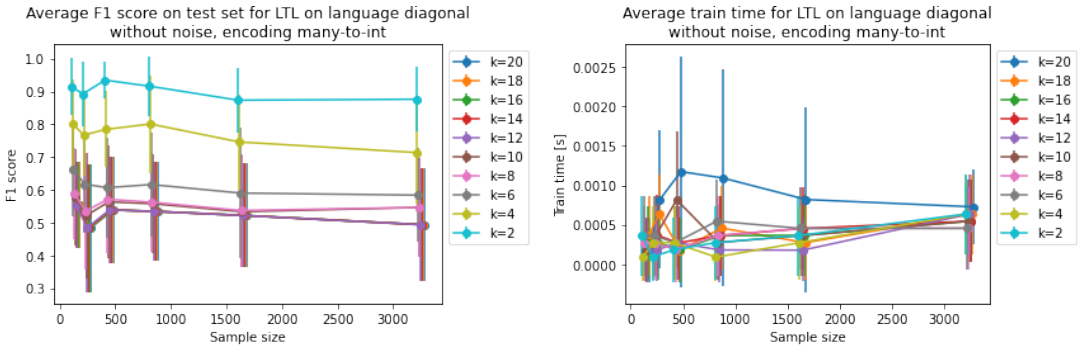

Figure 4.5: Results of training finite automata using k-locally testable languages for the crisp sample language L_1 with the many-to-int encoding of the contents of the scanning window. Average F_1-score on test sets on the left and average time for training on train sets on the right. The markers for different values of k are shifted a little to show overlapping marks.

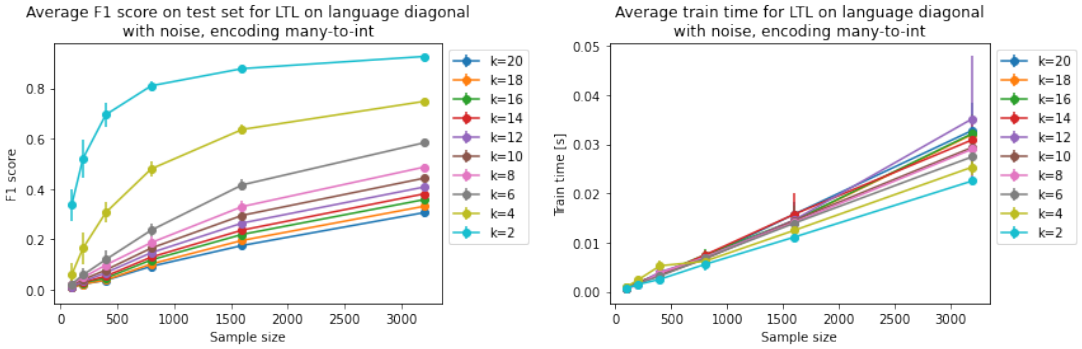

Figure 4.6: Results of training finite automata using k-locally testable languages for the noisy sample language L_1 with the many-to-int encoding of the contents of the scanning window. Average F_1-score on test sets on the left and average time for training on train sets on the right.

Since the many-to-int encoding has shown such a major improvement, all the experiments will exclusively be using this encoding.

This second set of experiments utilized all operations facilitated by the generator, so the languages included all possible flips and rotations (thus, e.g., the horizontal stripes language also included vertical stripes).

First, we compare different training methods to obtain the evaluator. Then we compare different scanning strategies and asses their compatibility with the languages.

4.2.1 Training methods

In this set of experiments, we look at three training algorithms – Traxbar, Windowed EDSM and Locally testable language. We average over all languages and look at the model performance curves for each of the scanning strategies. The initial window size for Windowed EDSM is 200. The number was chosen by estimating the number of states in the first two layers of the prefix tree.

In Figure 4.9, we can see there is just a small difference in performance be-

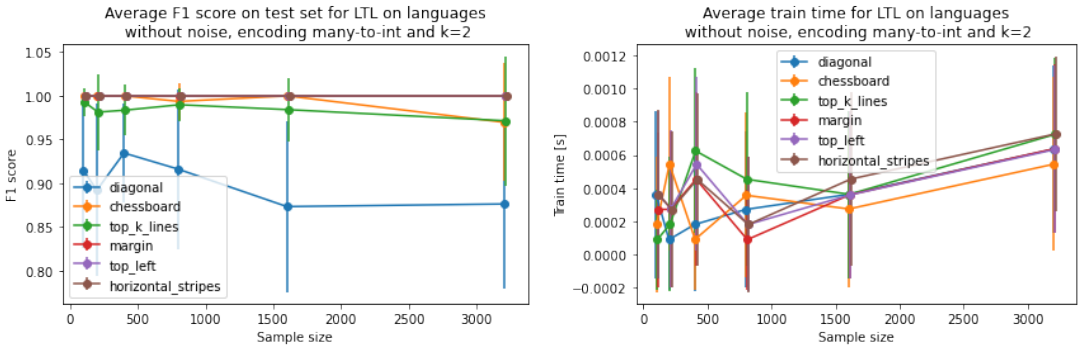

Figure 4.7: Results of training finite automata using 2-locally testable languages with the many-to-int encoding of the contents of the scanning window. Average F_1-score on test sets on the left and average time for training on train sets on the right. The markers for different values of k are shifted a little to show overlapping marks.

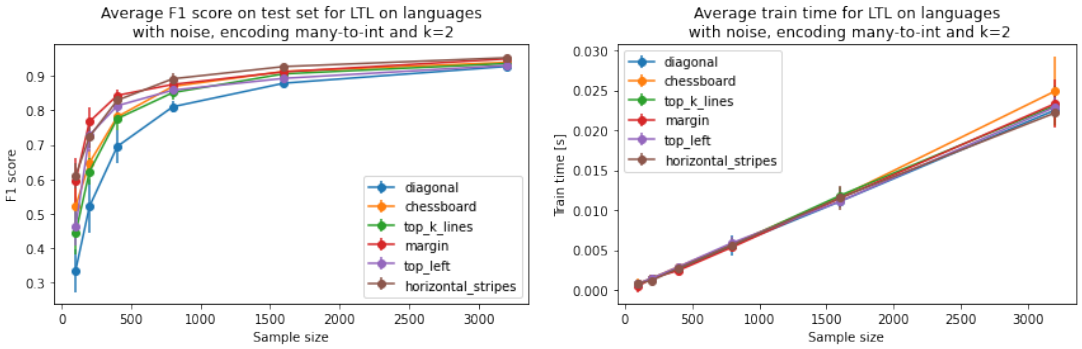

Figure 4.8: Results of training finite automata using 2-locally testable languages with the many-to-int encoding of the contents of the scanning window. Average F_1-score on test sets on the left and average time for training on train sets on the right.

tween Traxbar and Windowed EDSM. This can probably be explained by the fact that with the large alphabet, the scores computed for EDSM are initially often zero. EDSM, therefore, merged states mostly in the breadth-first order – same as Traxbar – and only made some minor adjustments.

In the same Figure, we can also more clearly see that the performance suffers noticeably more when fitting the automaton on a larger dataset of a crisp language compared to a noisy one. This is likely due to the dataset being heavily skewed towards the negative sample. The small amount of accepting states could mean that the merging algorithm is more likely to make a bad merge early on because most of the states can be merged. The final automaton is, therefore, larger (Figure 4.11), less general, but with no additional knowledge about the target language.

Similarly, the comparison in Figure 4.10 shows that even the time used for training the automaton shows a difference that could just be attributed to hardware factors. This indicates that the computing of the scores did not slow down the EDSM algorithm significantly. It could be the case that sub-trees in the automaton had very small intersections and thus both state merging algorithms performed mostly the same operations.

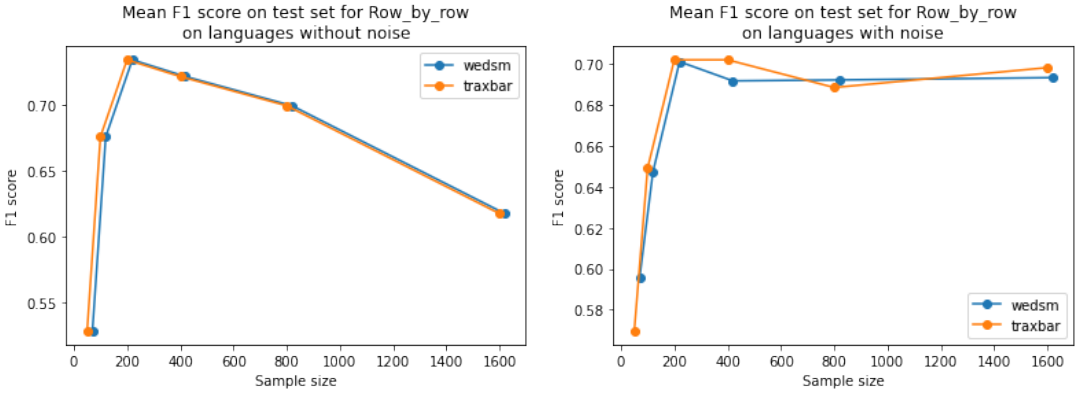

Figure 4.9: A comparison of F1 test performance between Traxbar and Windowed EDSM using Row-by-row strategy across all languages. Performance on crisp languages on the left and performance on noisy languages on the right.

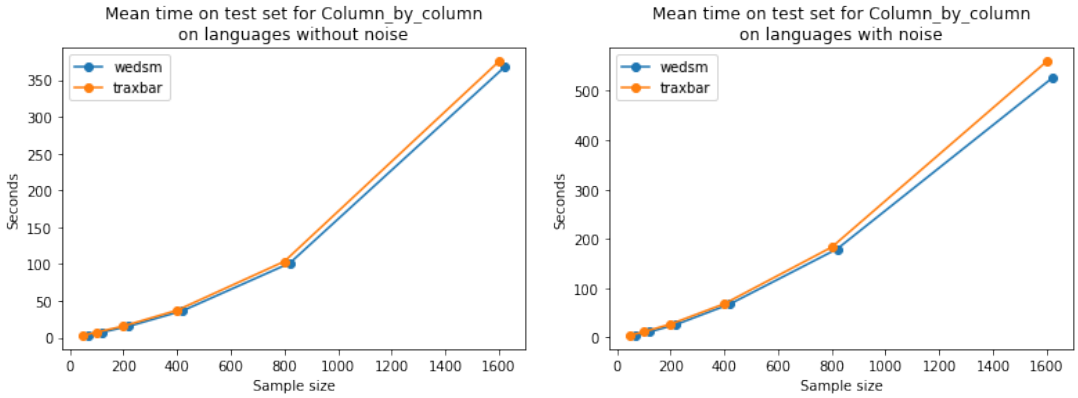

Figure 4.10: A comparison of time to train between Traxbar and Windowed EDSM using Column-by-column strategy across all of the tested languages. Time to train on crisp languages on the left and time to train on noisy languages on the right.

In Figure 4.11, we can see an interesting phenomenon. The figure shows the mean number of states on a final automaton. While the number grows steadily with the dataset size on the noisy languages, it initially drops on the crisp ones. This phenomenon is present in all strategies, it is only most pronounced in the Snake-by-row strategy.

We can also see that the Windowed EDSM algorithm was able to find automata with fewer states (hence more general) on noisy languages, which can be beneficial in more complex applications.

Looking at the time complexity paints a similar picture. There is no significant difference between the runtimes, thus we can assume the algorithms behaved the same. We only include some examples of the plots, all plots are however included in the attachment.

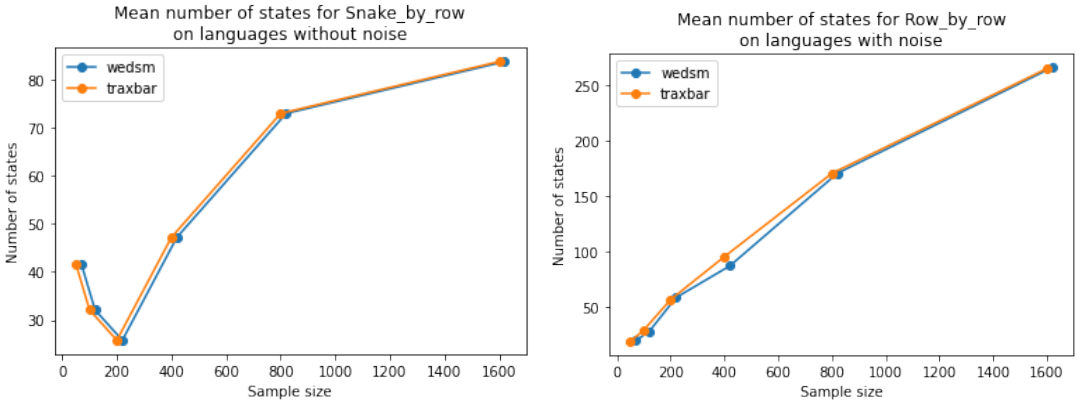

Figure 4.11: A comparisons of the number of states in the final automaton of Traxbar and Windowed EDSM across all sample languages. The number of states for the Snake-by-row strategy on crisp languages on the left and the number of states for the Row-by-row strategy on noisy languages on the right.

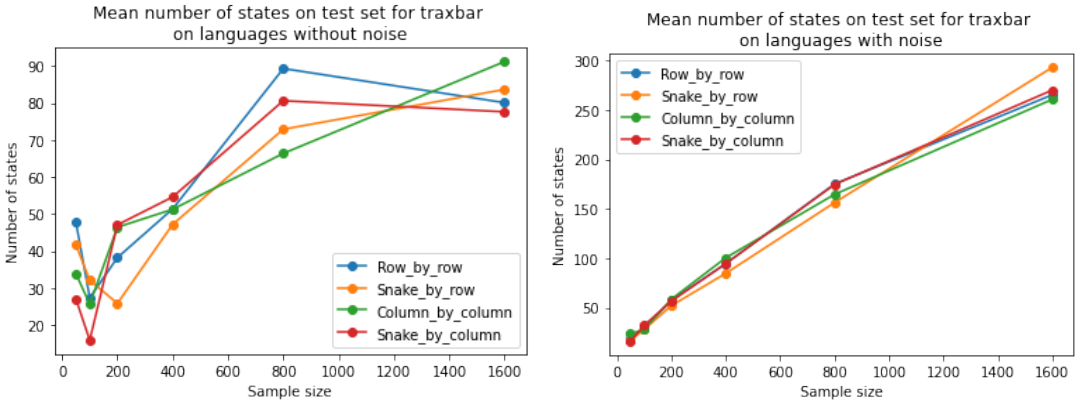

Figure 4.12: Two examples of the number of states in the final automaton of Traxbar and Windowed EDSM using Snake-by-row strategy across all languages. The number of states on crisp languages on the left and the number of states on noisy languages on the right.

4.3 Scanner behavior

This set of experiments is dedicated to surveying the various scanning strategies: Row-by-row, Column-by-column, Snake-by-row, and Snake-by-column. The aim is to infer some properties that could inform which of the strategies should be used for a given language.

As you can see in Figure 4.13, the performance of the scanners can vary greatly, but there is no clear best or worst approach.

We do a more in-depth analysis, where we look at each of the languages separately. Some interesting plots can be seen in Figure 4.14. We can again observe erratic behavior and see that there is no clear best strategy, not even a clear pattern. Strategies working the best on one size can do badly on another. Another interesting observation is that in the left plot Row-by-row and Snake-by-column – strategies that have the least in common – have very similar results.

This leads us to conclude that the chosen scanning strategy can have a very

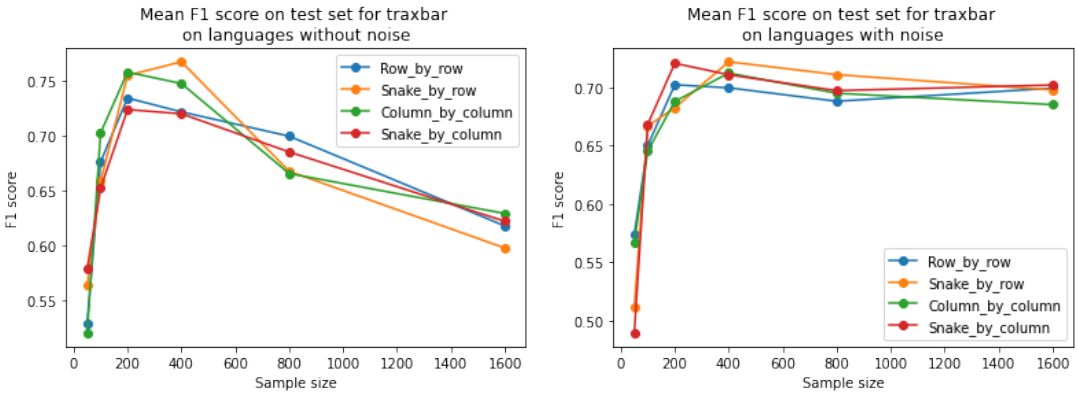

Figure 4.13: Results of training finite automata using k-locally testable languages with the many-to-int encoding of the contents of the scanning window. Average F_1-score on test sets on crisp languages on the left and the average F_1-score on test sets on noisy languages on the right.

significant impact on the performance of the model, but it can be hard to choose the correct one for a given task. The differences could be explained by the datasets being sampled in different ways, which might shift differences in the classes to be more apparent when we are scanning horizontally to when we are scanning vertically and vice versa. It, therefore, makes sense to put more focus on researching scanning strategies in the future.

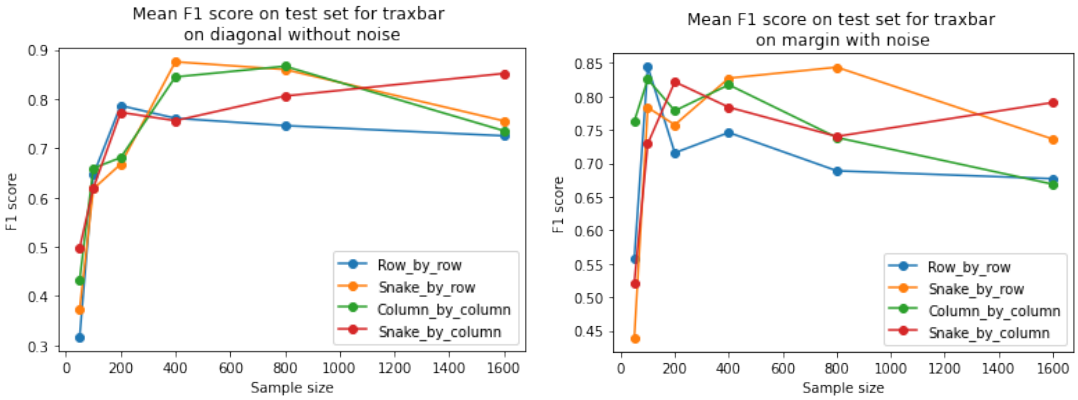

Figure 4.14: Comparing scanning strategies of training finite automata using Traxbar for the noisy sample language L_1 with the many-to-int encoding of the contents of the scanning window and different scanning strategies. Average F_1-score on test set of crisp L_1 on the left and average F_1-score on test set of noisy L_3 on the right.

From Figures 4.15 and 4.16, we can see two examples of how the scanning strategy does not have a significant impact on the performance of the k-locally testable languages. This is likely due to the relative simplicity of the languages; a different scanning strategy is not going to substantially change the substrings appearing in the transcribed language.

Overall, from the plots, we can see that even though the k-locally testable languages method works well on simple crisp languages, on smaller noisy datasets (up to size 400) it is outperformed by the state merging algorithms. It is only once

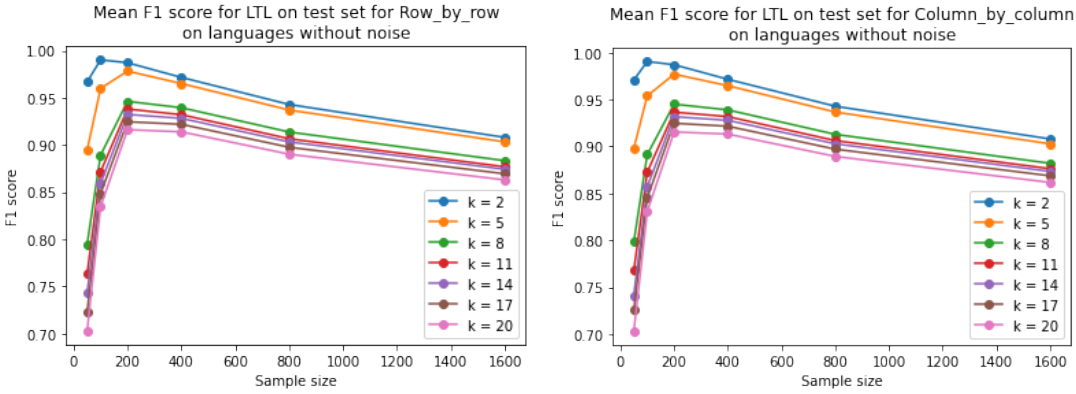

Figure 4.15: Results of training finite automata using k-locally testable languages for the crisp sample languages for two scanning strategies. Average F_1-score on test sets for Row-by-row on the left and Column-by-column the right.

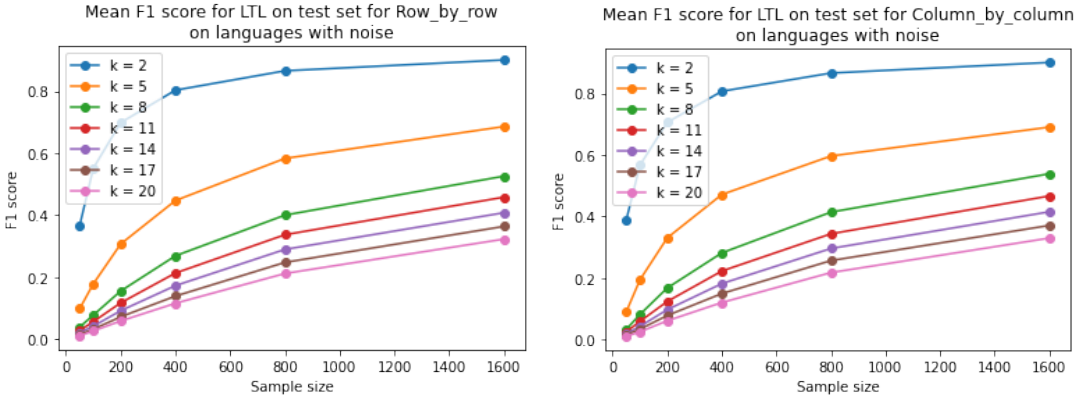

Figure 4.16: Results of training finite automata using k-locally testable languages for the noisy sample languages for two scanning strategies. Average F_1-score on test sets for Row-by-row strategy on the left and Column-by-column strategy the right.

the datasets are large enough to make the state merging algorithms overfit, that the k-locally testable languages get enough data to improve on the knowledge of the automata.

4.4 Summary

The above experiments shwo that our model TEMPL is suitable for representing picture languages and for learning picture languages.

Using locally testable languages as evaluators enables to learn our sample languages with a good F_1 score with any encoding. Of course the performance on noisy languages was a little lower.

Encoding one-to-one is not suitable when combined with traxbar due to a very slow training and poor performance. Using a many-to-int provides an order of magnitude improvement.

A scanning strategy contributes significantly to the performance of the model. We theorize the differences are connected to the sampling of the languages.

Conclusion

The thesis deals with the problem of learning picture languages and the notion that we can leverage the knowledge from learning string languages for it by rewriting pictures into strings.

We have proposed the TEMPL model that formalizes the process of picture language recognition using a picture-to-string transformation. TEMPL is composed of two parts, one for rewriting the picture to a string and the other to classify the string. The model is very general and allows for different levels of complexity. We have demonstrated some desirable properties that a restricted version of the TEMPL model has and that it can simulate other methods for recognizing picture languages.

We implemented a framework that is the realization of the TEMPL model. This framework offers the ability to learn picture languages from a set of positive and negative examples of a target picture language. The system supports multiple scanning strategies for variable window sizes and supports adding new strategies. There are three different learning methods in the system with the option to add more.

The system also offers a generator part, that can deterministically generate samples of selected languages. This generator allows for the reproducibility of the experiments via configuration files, that can unambiguously define the sets that will be generated, which allows easy benchmarking.

We experimented using this system with six languages and various scanning strategies and learning algorithms in order to determine the performance of the model with different parameters.

Our findings have shown that the for the state merging algorithms, many-to-int encoding clearly outperforms one-to-one. The second aspect of interest was the behavior of the learning algorithms on our sample languages. The performance of `traxbar` with many-to-int encoding on crisp languages varied significantly, but on the languages with noise, it showed a decent performance with F_1-scores mainly in the 0.6 to 0.8 range.

As for the locally testable language with the best performing value of $k = 2$ and many-to-int encoding, the learning algorithm succeeded for all of them, however, the diagonal language without noise turned out to be slightly more difficult to learn than others.

Comparing the different scanning strategies has shown that the choice of scanning strategy can have a significant impact on the model performance, but without any clear indication of how this choice should be made. Using a different state merging algorithm yields comparable results.

Some of the results were already published in Kuboň et al. [2023] and the findings were well received. There will be an extended version of the paper published in the future that will include more of the results from this thesis.

Further research can be focused on more advanced scanning strategies or developing a method to robustly determine an optimal scanning strategy.

Bibliography

Hasan Ibne Akram, Colin De La Higuera, Huang Xiao, and Claudia Eckert. Grammatical inference algorithms in matlab. In *ICGI 2020*, pages 262–266. Springer, 2010.

Manuel Blum and Carl Hewitt. Automata on a 2-dimensional tape. In *8th Annual Symposium on Switching and Automata Theory (SWAT 1967)*, pages 155–160. IEEE, 1967.

Gennaro Costagliola, Vincenzo Deufemia, Filomena Ferrucci, and Carmine Gravino. On regular drawn symbolic picture languages. *Information and Computation*, 187(2):209–245, 2003.

Colin De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

Pierre Dupont, Laurent Miclet, and Enrique Vidal. What is the search space of the regular inference? In *International Colloquium on Grammatical Inference*, pages 25–37. Springer, 1994.

Henning Fernau, Meenakshi Paramasivan, Markus L Schmid, et al. Simple picture processing based on finite automata and regular grammars. *Journal of Computer and System Sciences*, 95:232–258, 2018.

H. Freeman. On the encoding of arbitrary geometric configurations. *IRE Transactions on Electronic Computers*, EC-10(2):260–268, June 1961. ISSN 0367-9950.

Dora Giammarresi and Antonio Restivo. Two-dimensional languages. In *Handbook of Formal Languages*, pages 215–267. Springer Berlin Heidelberg, 1997. doi: 10.1007/978-3-642-59126-6_4. URL https://doi.org/10.1007/978-3-642-59126-6_4.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

Lukáš Krtek. Learning picture languages using restarting automata. master thesis, Charles University, Faculty of Mathematics and Physics, 2014.

David Kuboň and Frantisek Mráz. Learning picture languages represented as strings. In Roman Barták and Eric Bell, editors, *Proceedings of the Thirty-Third International Florida Artificial Intelligence Research Society Conference*, pages 529–532. AAAI Press, 2020.

David Kuboň, František Mráz, and Ivan Rychtera. Learning automata using dimensional reduction. In *Advances in Artificial Intelligence – IBERAMIA 2022*, volume 13788 of *LNCS*. Springer, 2023. In print.

Hermann A Maurer, Grzegorz Rozenberg, and Emo Welzl. Using string languages to describe picture languages. *Information and Control*, 54(3):155–185, 1982.

Friedrich Otto and František Mráz. Deterministic ordered restarting automata for picture languages. *Acta Informatica*, 52(7):593–623, 2015.

Matteo Pradella and Stefano Crespi Reghizzi. A sat-based parser and completer for pictures specified by tiling. *Pattern Recogn.*, 41(2):555–566, February 2008. ISSN 0031-3203.

Daniel Průša, František Mráz, and Friedrich Otto. Two-dimensional sgraffito automata. *RAIRO-Theoretical Informatics and Applications*, 48(5):505–539, 2014.

Daniel Průša and František Mráz. Restarting tiling automata. *International Journal of Foundations of Computer Science*, 24(06):863–878, 2013.

Grzegorz Rozenberg and Arto Salomaa, editors. *Handbook of Formal Languages, Vol. 3: Beyond Words*. Springer-Verlag, Berlin, Heidelberg, 1997. ISBN 3540606491.

sckit-learn website. scikit-learn, 2023. URL `https://scikit-learn.org/`. Accessed: 04/01/2023.

Cristina Tîrnăucă. A survey of state merging strategies for dfa identification in the limit. *Triangle: llenguatge, literatura, computació*, (8):121–136, 2012.

List of Figures

List of Abbreviations

DFA – deterministic finite automaton

IA – incomplete automaton

PTA – prefix tree automaton

4DFA – four-way deterministic finite automaton

2RTA – two-dimensional restarting tiling automaton

2DOTA – deterministic two-dimensional online tesselation automaton

4SA – four-way scanner automaton

EDSM – evidence-driven state merging

TEMPL – Transriptor-Evaluator machine for picture languages

A. Attachments

A.1 Electronic Attachment

- `experiment` contains the CSV files with raw experiment results and a `plots.ipynb` Jupyter notebook with all relevant plots.
- `implementation` contains the source codes for the framework and the experiments.
- `info.txt` contains library versions and some execution information.
- `implementation/example_generator/configs/my.yaml` conatins a commented example of a yaml configuration file.