Universidad ORT Uruguay

Facultad de Ingeniería

Active learning techniques for probabilistic deterministic finite automata extraction from language models

Entregado como requisito para la obtención del título de Ingeniería en Sistemas

Federico Pan - 199657

Tutores: Franz Mayr y Sergio Yovine

2020

Declaración de Autoría

Yo, Federico Pan Suarez declaro que el trabajo que se presenta en esta obra es de mi propia mano. Puedo asegurar que:

- La obra fue producida en su totalidad mientras realizaba el Proyecto;
- Cuando he consultado el trabajo publicado por otros, lo he atribuido con claridad;
- Cuando he citado obras de otros, he indicado las fuentes. Con excepción de estas citas, la obra es enteramente mía;
- En la obra, he acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, he explicado claramente qué fue contribuido por otros, y qué fue contribuido por mi;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Federico Pan Suarez

10/9/2020

Agradecimientos

Agradezco a mis tutores, Dr. Sergio Yovine y MSc. Franz Mayr, por la guía y ayuda brindada en la realización de este trabajo.

Abstract Español

El presente trabajo surge como una investigación motivada por el trabajo realizado en [1], con el objetivo de profundizar en el problema del entendimiento del funcionamiento interno de las redes neuronales recurrentes (RNN por su sigla en inglés). Para ello se enfoca en la extracción de autómatas de peso que permitan aproximar el modelo de lenguaje aprendido por la RNN, presentando dos variantes del algoritmo de extracción de autómatas L^* propuesto en [2], adaptado para la extracción de autómatas de peso en lugar de autómatas finitos deterministas.

Abstract

This work arises as a tangential investigation to the work in [1], its goal is to delve in the problem of understanding the inner working of recurrent neural networks (RNN). It focuses on the extraction of weighted automata that approximate the language model learned by the RNN, presenting two variants of the automata extraction algorithm L^* proposed in [2], adapted for the extraction of weighted automata instead of deterministic finite automata.

Palabras clave

Redes neuronales recurrentes; Probabilidad de ocurrencia de secuencias; Automata de peso; Extracción de caja negra; Extracción de reglas; Inferencia regular; Inteligencia Artificial; Explicabilidad, Modelos de lenguaje

Keywords

Recurrent neural networks; Sequence occurrence probability; Deterministic finite automaton; Black box extraction; Rule extraction; Regular inference; Artificial Intelligence; Explainability, Language models

Contents

1 Introduction

Artificial Intelligence, Machine Learning and in particular Deep Learning have been some of the more cutting-edge fields in computing in recent years, expanding quickly from the the academic field to practical applications in business, science, entertainment, cyber-secutity, real world security (face detection in real time), etc. [4]

Despite the great success of deep learning techniques in several fields, it remains as a rather obscure learning method, thus the processes and inner working behind its decisions are difficult to interpret and explain, making it hard to understand which aspects of the input data are behind those results [5]. In [6], DARPA advocates for more interpretable deep learning models.

This work is set in a particular field inside explainable AI, grammatical inference from language models, extracting Probabilistic Deterministic Finite Automatons (PDFA) from black box models. This type of automata are particularly suitable to describe RNN used for learning grammars and for sequence classification, since their output is a probability distribution over the symbols of the alphabet, which can be represented by PDFAs.

We will use active learning techniques as opposed to the more commonly used passive learning ones. Active and Passive are two learning paradigms. In active learning, the learner interacts with the environment at training time, while the passive learner only observes the information provided without influencing or directing it [7].

In our particular problem, training RNNs would be an example of passive learning, they are given sequences that have already been branded, and the RNNs consume this sequences, not being able to change them or perform other queries about them. On the other hand, L^* and all its adaptations, including the one presented in this work, fall in the active learning paradigm, since we have a learner that actively queries the teacher (which knows the target language model), building the model in this fashion.

The objective of this work is to develop an algorithm that given a language model, such as the ones represented by RNNs learning sequences, is capable of extracting an equivalent PDFA through active learning techniques.

Outline

In Chapter 2 we present the necessary previous knowledge and an analysis of the problem in question. In Chapter 3 we show the solution found to the problem, presenting two versions of the algorithms, and showing how they work through a full run of each one. In Chapter 4 we test the algorithm both against PDFAs and RNN based language models. In Chapter 5 the lines of investigation that will be pursued next, stemming from this work are presented. Finally we present the conclusions and lessons learned.

2 Preliminaries

2.1 Hankel matrix

Let Σ be a finite alphabet and σ an arbitrary symbol in Σ. The set of all finite strings (or words) over Σ is denoted Σ^*, where ρ is the empty string.

We can define functions over strings, like $f : \Sigma^* \to \mathbb{R}$ or $g : \Sigma^* \to \mathbb{B}$. Using f as an example, we have that the Hankel Matrix of f is a bi-infinite matrix $H_f \in \mathbb{R}^{\Sigma^* \times \Sigma^*}$, which entries are defined as $H_f(u, v) = f(uv)$ for any $u, v \in \Sigma^*$. That is, rows are indexed by prefixes and columns by suffixes [8].

2.2 Deterministic Finite Automaton (DFA) extraction

This investigation stems from the work in the field of explainable AI that is being conducted by the AI research group. The work in [1] starts the extraction framework in which this work is inscribed, by extracting DFAs from RNNs.

2.2.1 DFAs and regular languages

In this work we will consider DFA and regular language as two analogous expressions, since a regular language is defined as one that can be recognized by a DFA [9], which are formally defined as a computational model consisting of:

1. A finite set of states denoted Q.
2. A finite set of input symbols denoted Σ.
3. A transition function that takes as arguments a state and an input symbol and returns a state, denoted δ.

4. A start state belonging to Q denoted q_0.

5. A set of final or accepting states F included in Q.

A DFA can also be represented by a transition diagram, as in Fig. 2.1, or a transition table, as in Table 4.1, where $\rightarrow$ and $*$ indicate initial and final states, respectively.

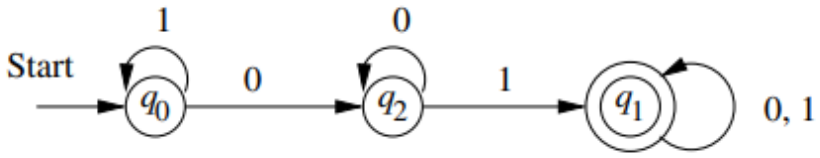


Figure 2.1: DFA accepting all strings with a 01 substring

	0	1
$\rightarrow q_0$	q_2	q_o
$*q_1$	q_1	q_1
q_2	q_2	q_1

Table 2.1: Transition table corresponding to the DFA of Fig, 2.1

2.2.2 L^* and bounded L^* algorithms

The L^* [2] and Bounded L^* [1] algorithms fall in the category of active learning algorithms, in which a learner interacts with a teacher. In this algorithms the learner performs membership and equivalence queries to a minimum adequate teacher in order to infer a DFA that represents the regular language known by the teacher.

2.2.3 L^*

The L^* algorithm is presented in [2], who also proves that it can learn a DFA in polynomial time in the number of states of the minimum DFA representative of the regular language and the maximum length of the counterexample presented by the teacher. This is a huge improvement from the passive learning of a language which is proven in [10] to be NP-Complete.

This algorithm learns the regular language by building an Observation Table (OT) through queries to a so-called Minimum Adequate Teacher (MAT). Two types of queries are used, Membership Queries (MQ) which consist of asking the teacher if a given sequence belongs to the language or not, and Equivalence Queries (EQ) that consist of asking if the constructed DFA is equivalent to the regular language the teacher knows.

The observation table used by the learner to construct the DFA is separated into two parts, the upper part denoted RED and the lower part denoted BLUE following the nomenclature in [11]. In RED rows the learner keeps all words that have already being tested by an MQ and the BLUE rows contain every word in RED followed by every symbol in the alphabet. The columns represent the suffixes of the words represented in the rows, the table is initiated with the empty string as the first row in RED and also as the first column representing the empty suffix. A table has no holes if for every cell there is a 0 or 1 value.

To be able to build a DFA from the observation table, it has to fulfill two conditions: to be closed, for every row in BLUE there is an equivalent row in RED, and consistent, for each pair of equivalent rows in RED, when adding the same symbol to both rows they have the same membership result.

Once the table is both closed and consistent, the learner can translate it to a DFA by extracting a different state q_v for each unique row v in RED, and adding the transitions following that $\delta(q_v, \sigma) = q_w$ if $OT[v\sigma] = OT[w]$, the final states are represented by the rows where $OT[w][\rho] = 1$.

Once the DFA is extracted from the observation table, the learner performs an EQ, if the result is positive the algorithms finishes and returns the DFA, if the result is negative, the teacher also provides a counterexample (a word in which the membership query is different for the language known by the teacher and the language represented by the DFA). This counterexample along with all its prefixes are added to RED and all concatenations with every symbol in the alphabet are added to BLUE. Then the table is made closed and consistent again repeating the process until the equivalent DFA is constructed.

	OT_0	ρ
RED	$\rightarrow \rho$	0
BLUE	0	0
BLUE	1	0

(a)

OT_1	ρ	b
$\rightarrow \rho$	0	0
0	0	1
* 01	1	1
1	0	0
00	0	1
010	1	1
011	1	1

(b)

Table 2.2: Observation tables during an L^* run for DFA presented in Figure 2.1

2.2.4 Bounded L^*

Providing that the language known by the teacher is regular, L^* is guaranteed to finish and find an equivalent DFA. This follows from the fact that the matrix built in RED is the binary Hankel matrix representing the regualar language which has been proven to have a finite row space of size equal to the minimal number of states of a DFA recognizing the language represented by the matrix [12]. But since we are working with a black box we do not know the language known by the teacher and can therefore make no assumptions about its Hankel matrix. In this case, the algorithm may run forever, never finding an equivalent DFA.

In this particular line of investigation, our goal is to focus on language models based in RNNs. Artificial Neural Networks have been proven to be Turing Complete, meaning that, given the proper architecture, they can simulate any Turing Machine [13] [14]. Since we are approaching this problem as black box extraction we do not know if the RNN known by the teacher is in fact equivalent to some automaton definition, making the use of bounds all the more important.

To make sure the algorithms finish, there are mechanisms in place to stop the algorithm from running forever, these are bounds in the maximum number of states and the maximum length of the membership query. this new algorithm is named Bounded L^* in [1].

2.3 Weighted Finite Automaton

A Weighted Finite Automaton over the semiring S is a structure $A = (Q, \Sigma, \theta, \mu, \gamma)$ where [15]:

1. Q is the nonempty finite set of states.
2. Σ is the input alphabet.
3. $\mu : Q \times \Sigma \times Q \to S$ is the transition-weight function.
4. $\theta : Q \to S$ is the initial-weight function.
5. $\gamma : Q \to S$ is the final-weight function.

A semiring $(R, +, .)$ is a nonempty set on which we have defined operations of addition and multiplication satisfying the following conditions [16]:

1. $(R, +)$ is a commutative monoid with identity element 0.
2. $(R, .)$ is a monoid with identity element $1 \neq 0$.
3. $a.(b + c) = a.b + a.c$ and $(a + b).c = a.c + b.c, \forall a, b, c \in R$.
4. $0.a = 0 = a.0$, $\forall a \in R$.

A monoid is a set that is closed under an associative binary operation and has an identity element $I \in S$ such that $\forall a \in S, Ia = aI = a$. Note that unlike a group, its elements need not have inverses. A monoid must contain at least one element [17].

Examples of semirings are $B = (0, 1, \vee, \wedge, 0, 1)$, $(N, +, ., 0, 1)$ being the boolean and the natural numbers semirings respectively or $(R, +, ., 0, 1)$ the real numbers ring (by definition all rings are also semirings plus the additive inverse).

To calculate the weight of a word w, we need to perform the sum of the product of all weights following w over all the possible paths, starting from initial weights and ending with the final weights.

A WFA can be represented with graphs like DFAs indicating the weights on each transition, and the initial and final weight on each state.

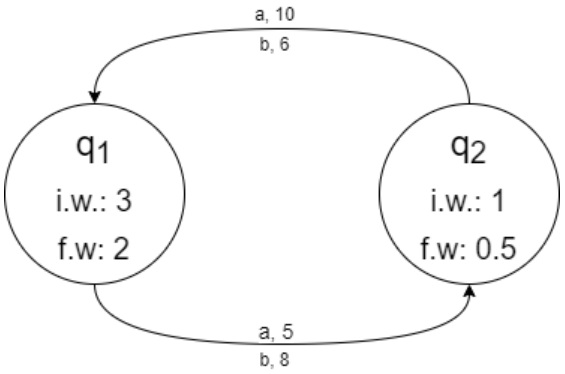


Figure 2.2: WFA example

The DFAs mentioned earlier are just WFAs over the boolean semiring in which a restriction has been imposed that there can only exist one transition per symbol per state.

2.3.1 Probabilistic Determinisitic Finite Automaton

Probabilistic Deterministic Finite Automata are WFAs over the semiring $Prob = (R \geq 0, +, ., 0, 1)$ which also fulfill the following requirements [15]:

1. There is a single state $p \in Q$ such that $\theta(p) = 1$ and $\forall q \in Q / q \neq p, \theta(p) = 0$.
2. $\forall p \in Q, \gamma(p) \in [0, 1]$.
3. $\forall p, q_1 \in Q, a \in \Sigma,$ if $\mu(p, a, q_1) \neq 0 \Rightarrow \mu(p, a, q) = 0$, $\forall q \in Q \neq q_1$
4. $\forall p, q \in Q, \sum_{a \in \Sigma} \mu(p, a, q) = 1$.

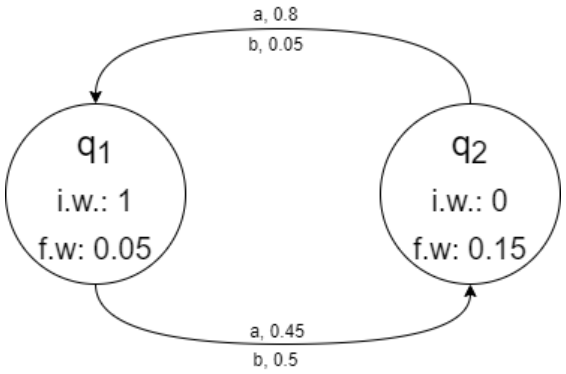


Figure 2.3: PDFA Example

In the previous example we can see that only one state q_1 has an initial weight of one while the other state q_2 has a initial weight of zero. If we consider state q_1 we can see that the transition probabilities are 0.45 for "a" and 0.5 for "b", and the final weight is 0.05, summing these values we get 1 so it represents a probability distribution over all transitions; the same occurs in state q_2. Also, we can observe that for each state there is only one transition for each symbol.

2.4 Problem analysis

The goal of this work is to learn WFAs from language models [18] [19] in black-box setting. Here, we define a language model to be given by two functions, namely P and P'. $P : \Sigma_{\$}^* \to [0, 1]$, with $\Sigma_{\$} = \Sigma \cup \{\$\}$, assigns to each string $w \in \Sigma_{\$}^*$ a probability of occurrence, where \$ is a special *terminal* symbol. The so-called *next symbol* probability $P' : \Sigma^* \to [0, 1]^{|\Sigma_{\$}|}$, assigns to each string w the conditional probability distribution $P'[\sigma|w]$, representing the probability of w being continued by a symbol $\sigma \in \Sigma_{\$}$. The relationship between P and P' is that $P(\sigma_1 \dots \sigma_n) = \prod_{i=1}^{n} P'[\sigma_i \mid \sigma_1 \dots \sigma_{i-1}]$. The role of \$ is explained later.

In order to be able to learn WFAs from a language model as defined, we set the task of adapting Angluin's well known L^* algorithm, the idea is to be able to construct a real Hankel matrix as opposed to the boolean Hankel matrix extracted for DFAs. For real Hankel matrices it has been proven that the row space size is equal to the minimal number of states of a WFA representative of the weighted language [20] [21].

As such, it is possible to translate from a real Hankel matrix to the WFA

generating the weighted language. In practice, since the language model (specially when working with RNNs) is not guaranteed to give exact values, arriving at the real Hankel matrix might not be entirely possible. A solution given in [3] is to use a tolerance level when comparing rows from the matrix.

When looking at the definition of language model given earlier, we can see that it is a function from strings to probability distributions for the next symbol, including all symbols of the alphabet and the terminal symbol. As such, these language models represent probabilistic languages, which are represented by PDFAs. Taking this fact into consideration, we adapt L^* to extract PDFAs from language models known by the teacher. This language model can be a PDFA or an RNN which have learnt a probabilistic language.

Having transitions weights limited to the [0, 1] interval greatly facilitates the use of a tolerance level when comparing observations

3 Probabilistic deterministic finite automaton Extraction

In this chapter we will explain the two versions of the PDFA extraction algorithm. Both algorithms are adaptations of L^* and as such make use of an observation table in the same style of the one proposed in [2] to construct the real Hankel matrix, a learner in charge of filling the observation table through queries to a teacher that knows the language model, and a translation strategy to transform the RED part of the observation table (Hankel matrix) to a PDFA.

After explaining each version to be able to show the reader the stages of the algorithm we will perform the extraction directly from a PDFA. For this example we will borrow the PDFA used in [3], which can easily exemplify the differences in both strategies. A tolerance of 0.1 will be used.

To simplify PDFAs representations, we will use a diamond shape to represent the state with an initial weight of one and a circle for the rest, also, in each state will only be represented the final weight since the name is not relevant

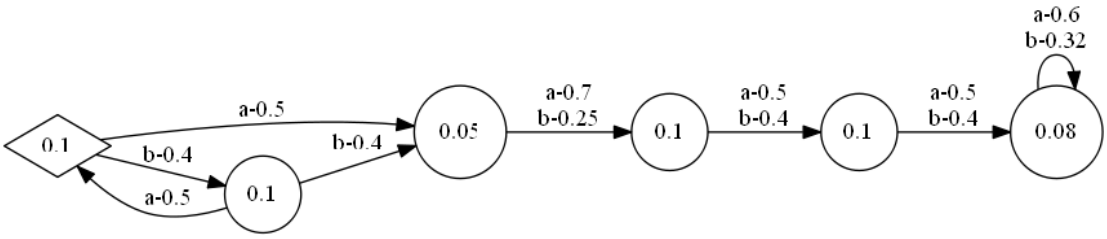


Figure 3.1: Target PDFA borrowed from [3]

3.1 Observation table

The observation table is formed, as in the case of DFA extraction, by a RED part consisting of sequences that have already been checked and a BLUE part filled with the sequences in RED followed by each symbol in the alphabet. Columns represent the suffixes of the words in each row.

In each cell of the resulting matrix we put the probability of the last symbol after traversing the word resulting of the concatenation of the prefix (in the row) and the suffix (in the column).

The sequences in BLUE are stored in a priority queue, with the priorities being the weight of the sequence without the terminal symbols, this serves as a heuristic to represent the probability that sequence has as prefix of any other sequence as proven in [3].

OT	\$	a	b
ρ	0.05	0.45	0.5
a	0.15	0.8	0.05
b	015	0.8	0.05

Table 3.1: Initial Observation Table for the automaton in Figure 2.3 using "$" as the terminal symbol

The terminal symbol $ is used to represent a sequence that ends in the symbol prior to the terminal symbol. When calculating the probability of a sequence if it ends in this symbol $, we multiply also by the final weight of the ending state.

We exemplify how to calculate sequence probability and symbol token probability using the PDFA from Figure 3.1. To calculate the sequence probability we multiply the transitions labeled with each letter from the sequence, following the successive states. To calculate the next token probability we traverse the word, ending in one state and return for each symbol the transition probability from that state and for the terminal symbol the final probability of the state.

Let us consider sequence "bab", we have for sequence probability $0.4 \times 0.5 \times 0.4 = 0.08$, for next token probabilities we can see that we end up in the second state from left to right, so we get {(a, 0.5), (b, 0.4), ($, 0.1)}.

If we now consider sequence "bab$" we get the probability $0.4 \times 0.5 \times 0.4 \times 0.1 = 0.008$, we need to multiply also the final weight of the terminating state. In this case we do not have next token probabilities since the sequence ends there.

3.2 Single hypothesis PDFA extractions

In this first version of the extraction algorithm we use a greedy strategy when translating the observation table to the automaton.

3.2.1 Single hypothesis learner

Just like in L^*, the learning task focuses on building an observation table, making it consistent and closed, the learner fills the observation cells by performing queries to the teacher; once the table is closed and consistent an equivalence query is performed to find if the representative PDFA has been found.

Algorithm 1: Single Hypothesis Learner

```
   Input  : Tolerance
   Output: PDFA P, EQCount
 1 Initialize-Observation-Table;
 2 while Learnt = False do
 3     while OT is not closed or not consistent do
 4         if OT is not closed then
 5             OT ← CloseTable(OT);
 6         if OT is not consistent then
 7             OT ← MakeTableConsistent(OT);
 8     P ← BuildAutomaton(OT);
 9     Learnt, Counterexample ← EQ(P);
10     if Learnt ≠ True then
11         OT ← AddCounterExample(OT, Counterexample);
12 return P;
```

The Single Hypothesis Learner is shown in Algorithm 1. The learner begins by initializing the observation table, adding the empty sequence to RED and each of the alphabet symbols to BLUE, then the terminal symbol and each of the alphabet's symbols are added as columns, then the observations are filled by querying the teacher for each sequence in RED and BLUE, asking for the weight of each suffix (this would be the equivalent to the membership queries in L^*).

The learner, then proceeds to close the table (Algorithm 2), meaning that there are no observation rows in BLUE without an equivalent in RED. As we have

already mentioned, it is virtually impossible to obtain two identical rows from the RNN, so we need to compare the observations using the tolerance defined. We consider two rows to be equivalent when for each suffix the absolute value of the difference is within the tolerance level.

Algorithm 2: Close Observation Table

```
Input  : Observation Table OT, Alphabet A
Output: Observation Table OT
ViolatingSequences ← GetBlueSequencesWithoutEquivalentInRed(OT);
while Len(ViolatingSequences) > 0 do
    foreach VS ∈ ViolatingSequences do
        MoveFromBlueToRed(OT, VS);
        VSCont ← GetSequenceContinuations(VS, A);
        AddToBlue(OT, VSCont);
    ViolatingSequences ←
     GetBlueSequencesWithoutEquivalentInRed(OT);
return OT;
```

Consider Table 3.1, we have three rows, one in RED and two in BLUE, if we want to compare these rows we compare the cells where the real values for the suffix following the prefix is. If we compare the row for ρ and "a" we have two vectors (0.05, 0.45, 0.5) and (0.15, 0.8, 0.05), the absolute value of the differences being (0.1, 0.35, 0.45), which means that with a tolerance of 0.1 these two rows are not equivalent, so "a" needs to be added to RED, "aa" and "ab" to BLUE.

To find sequences violating closeness, the learner asks the observation table to return all sequences in BLUE that have no equivalent in RED, which are returned ordered by priority as explained in the previous section. When the algorithm terminates, this priority is of no use, but if restrictions are imposed in the table expansion, as they were in [3], this heuristic allows us to take the sequences with the highest probabilities, building the Hankel matrix partition that is most representative of the target probabilistic language model.

These sequences from BLUE are then added to RED, and each of its continuations are added to BLUE. Then the learner asks again for sequences violating the closeness property, iterating in this way until an empty list is returned.

Algorithm 3: Make Observation Table Consistent

Input : Observation Table OT, Alphabet A
Output: Observation Table OT

```
1 InconsistentObservation, Symbol, Suffix ← FindInconsitency(OT);
2 while InconsitentObservation ≠ Null do
3 |   NewSuffix ← Concatenate(Symbol, Suffix);
4 |   AddNewSuffixToColumns(NewSuffix, OT);
5 |   CloseTable(OT);
6 |   InconsistentObservation, Symbol, Suffix ← FindInconsitency(OT);
7 return OT;
```

Once the table is closed, the learner makes it consistent (Algorithm 3). To achieve consistency, the learner controls that for each pair of equivalent rows in RED, if a symbol is added to the prefix, the observation rows for the resulting sequences are also equivalent. The learner asks the observation table to find an inconsistency, and the observation table looks for the first row violating this condition, returning the sequences, the symbol that generates the discrepancy, and the suffix with the greatest difference between the two observations.

Then, the learner adds to the suffixes (new column), the sequence resulting from concatenating the symbol causing the discrepancy with the suffix with the greatest discrepancy. The table is completed again by asking the teacher for the weight of this new suffix for each observation in RED and BLUE.

Consider for example the following table:

OT	\$	a	b
ρ	0.05	0.45	0.5
a	0.15	0.8	0.05
aa	0.05	0.45	0.5
b	015	0.8	0.05
ab	0.05	0.45	0.5
aaa	0.05	0.3	0.65
aab	0.05	0.3	0.65

Table 3.2: Example table

We can observe that the rows for ρ and "aa" are equivalent since both are (0.05, 0.45, 0.5). If we add the symbol "a" to both prefixes we now have to look at the "a" and "aaa" rows which are (0.15, 0.8, 0.05) and (0.05, 0.3, 0.65) respectively. If we calculate the absolute value of the differences we obtain (0.1, 0.5, 0.6), we can see that the two resulting rows are not equivalent, and the biggest difference of 0.6 is in the "b" symbol; the idea of using the suffix with the greatest difference comes from [3]. So we need to add "ab" to the suffixes.

After making the table consistent, it is checked again for closedness, iterating this process until the table is both closed and consistent. Once the table is closed and consistent, the learner translates the table into an automaton.

When the learner has a possible PDFA it performs an equivalence query to the teacher (as explained in the next section), which will answer whether the automaton is equivalent modulo the tolerance level to the target language model. If it results that the PDFA is indeed equivalent the learner returns this automaton and the learning process is complete.

On the other hand, if the learner has not found an equivalent PDFA, the teacher will return a counterexample, a sequence for which the automaton and the target model differ. This sequence, and all its prefixes will be added to RED, and all continuations to BLUE (provided they are not already in RED). This iterative process will be repeated until an equivalent automaton is found.

3.2.2 Single hypothesis teacher

The teacher is in charge of answering the queries performed by the learner, returning the needed weights to fill the observation table and answering the equivalence queries.

The teacher receives a sequence and a list of required suffixes, returning a list of the weights of the last symbol after traversing the sequences resulting of adding each suffix to the sequence received.

The second task of the teacher is to answer equivalence queries (Algorithm 4). For this, the teacher takes a fixed amount of sequences at random, and for each prefix of each sequence takes the weight of each symbol of the alphabet and the terminal symbol after traversing the prefix, for both the target language and the proposed automaton, comparing results using the same tolerance used when building the table. If a sequence is found for which the observations differ in more than the tolerance value for any of the symbols, this sequence is returned as a counter

example. Sequences and then prefixes are ordered by length so the shortest differing sequence is returned. If no sequence is found, the automaton is considered equivalent to the target language model.

Algorithm 4: Single Hypothesis Equivalence Query

Input : PDFA A, TestData TD, Tolerance T
Output: Learned, Counterexample

```
1 foreach w ∈ TD do
2     Obs1 ← GetNextTokenProbas(A, w);
3     Obs2 ← GetNextTokenProbas(TargetLanguage, w);
4     if not AreEquivalent(Obs1, Obs2, T) then
5         return False, w;
6 return True, Null;
```

3.2.3 Single hypothesis automaton translation

In this part of the extraction process the observation table is translated to a PDFA, using the observation rows in RED.

When we take into account the mechanism to build the table, we see that the property of rows being equal is not transitive. To take a simple example, we define equivalence with tolerance T as $=_T$ and consider the numbers 1, 2 and 3 and a tolerance $T = 1$, we then have that:

1. $1 =_T 2$
2. $2 =_T 3$
3. $1 \neq_T 3$

This fact needs to be taken into account when grouping sequences to form states and when adding transitions between states.

The Single Hypothesis Automaton translation procedure for generating a PDFA is shown in Algorithm–5. It starts by iterating over the sequences in RED and adding said sequences to candidate states, then transitions are added to this candidate states. These states are then translated to weighted states and weighted transitions are added, finally building automatons with this states and transitions.

Algorithm 5: Single Hypothesis Observation Table Translation

Input : Tolerance T, ObservationTable OT
Output: PDFA A

```
1  CandidateStates ← MakeStates(OT,T);
2  AddTransitions(CandidateStates, T);
3  IsDeterministic ← MakeDeterministic(CandidateStates, T);
4  while not IsDeterministic do
5  |  DeleteTransitions(CandidateStates);
6  |  AddTransitions(CandidateStates, T);
7  |  IsDeterministic ← MakeDeterministic(CandidateStates, T);
8  PDFAStates ← MakePDFAStates(CandidateStates);
9  AddPDFATransitions(PDFAStates, CandidateStates);
10 A ← MakePDFA(CandidateStates);
11 return A;
```

Taking into account the non transitivity of the equivalence as defined in this work, when adding observations to candidate states, there are sequences that could be added to more than one state, each containing observations equivalent to the observation in consideration but not between each other. The same may happen when adding transitions, since we want to make a PDFA, only one transition per state per symbol is allowed.

This version of the algorithm solves the problem of adding an observation to a candidate state by taking into consideration whether or not it is equivalent to all observations already in the state. In the case that it is equivalent to all observations in more than one state, the sequence is added to the one for which the distance to its centroid is minimal. When a new observation is added to a candidate state, its centroid is recalculated using all observations including the one just added.

The centroid is the multivariate equivalent of the mean. Even though a mass can be given to each vector when calculating the centroid, we use the approach of equal masses, arraving at the following definition.

Being V a set of L vectors we calculate the centroid as:

$$c = \frac{1}{L} \sum_{1}^{L} v_i, \tag{3.1}$$

Centroids are widely used in non hierarchical clustering techniques [22] [23], of which the most known is K nearest neighbour (KNN).

Once the candidate states are created, we iterate over them and for each state we take each observation belonging to it, and check for each symbol of the observation including the terminal symbol if the sequence resulting of adding the symbol to the prefix is found in another state, if it is we add a transition to it. If the sequence is not found in any state, we look for states having all sequences equivalent to the resulting one, if there is more than one state for which a transition could be added, the distance to the centroid is again taken into consideration to select the state to which we add the transition.

With this mechanism we avoid adding more than one transition for a pair observation-symbol, nevertheless it could happen that for two different observations in the same state, different transitions for the same symbol exist. To solve this issue, after adding transitions all candidate states are again iterated over, splitting observations that generate indetermination, generating thus, new candidate states. Then transitions are added again and the states are again checked for indetermination, repeating this process until the resulting automaton is deterministic.

Once we have all states with its correspondent transitions, weighted states are generated and weighted transitions added. If the candidate states contains the empty sequence among the prefixes of its observations, an initial weight of one is given, otherwise, the initial weight is zero. The final weight is calculated as the average of the weights of the terminal symbol for each observation. For this we take each observation row in the candidate state, sum the value for the terminal symbol column and divide by the amount of observation rows in the candidate state.

3.2.4 Single hypothesis PDFA extraction run

Consider the target PDFA of Fig. 3.1. The learner begins by initializing the table, adding ρ to the prefixes in RED, "a" and "b" to the prefixes in BLUE, and "$" (terminal symbol), "a" and "b" as the suffixes, and fills the cells obtaining the table:

$OT1$	$	a	b
ρ	0.1	0.5	0.4
a	0.05	0.7	0.25
b	0.1	0.5	0.4

Table 3.3: Initial table

It then proceeds to make the table closed, first it looks at the row with prefix "a", and as we can see is not equivalent to the row for ρ, as they are not equal the row for "a" is added to RED. It then considers "b" which row is equal to the row for ρ, so "b" remains in BLUE along with the continuations for "a", which are "aa" and "ab".

$OT2$	\$	a	b
ρ	0.1	0.5	0.4
a	0.05	0.7	0.25
b	0.1	0.5	0.4
aa	0.1	0.5	0.4
ab	0.1	0.5	0.4

Table 3.4: First closed table

As we can see the preceding table is closed since there is no observation in BLUE without an equivalent in RED. It is also consistent, the only two observation rows in RED are not equivalent, so there is no need to check for consistency.

The learner takes the RED part of the observation table and uses the translation strategy to extract the first automaton. Given the fact that there are only two rows in RED and that they are not equivalent, each row goes to a candidate state, which we will call ρ-state and a-state. Then transitions are mapped; if we add an "a" to ρ we have a transition through "a" from ρ-state to a-state, since "a" is part of a-state. If we add a "b" to ρ we get a transition from ρ-state to ρ-state since the row for "b" is equivalent to the row for ρ. When adding an "a" and a "b" to "a" we obtain "aa" and "ab" which rows are equivalent to the ρ row, thus we map two transitions from a-state to ρ-state through both symbols.

Then we add the initial and final weight to the states, the ρ-state is the one containing ρ so it has an initial weight of 1 while a-state has an initial weight of 0. Then we take the value for the terminal symbol as the final weight for each state, being 0.1 for ρ-state and 0.05 for a-state. We then complete the translation by creating the actual PDFA:

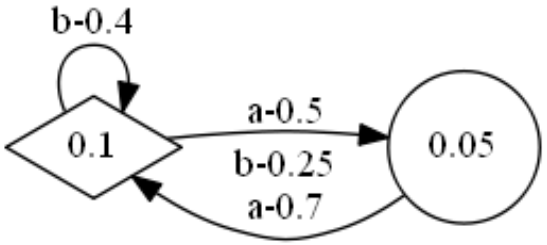


Figure 3.2: First PDFA Extracted

This PDFA is then passed to the teacher which finds it not to be equivalent to the target one, yielding the "ba" counterexample, which is added to RED along with its only prefix "b", resulting in:

$OT3$	$	a	b
ρ	0.1	0.5	0.4
a	0.05	0.7	0.25
b	0.1	0.5	0.4
ba	0.1	0.5	0.4
aa	0.1	0.5	0.4
ab	0.1	0.5	0.4
bb	0.05	0.7	0.25
baa	0.05	0.7	0.25
bab	0.1	0.5	0.4

Table 3.5: Observation table updated with "ba" counterexample

As we can see this table is already closed, so we proceed to check it for consistency. We find that for ρ and "b", which are equivalent, if we add an "a" we get "a" and "ba" which are not equivalent, and the greatest difference is produced for the suffix "a", so we add "aa" to the suffixes and fill all observations. The resulting table is both closed and consistent.

$OT4$	$	a	b	aa
ρ	0.1	0.5	0.4	0.7
a	0.05	0.7	0.25	0.5
b	0.1	0.5	0.4	0.5
ba	0.1	0.5	0.4	0.7
aa	0.1	0.5	0.4	0.5
ab	0.1	0.5	0.4	0.5
bb	0.05	0.7	0.25	0.5
baa	0.05	0.7	0.25	0.5
bab	0.1	0.5	0.4	0.5

Table 3.6: Second closed and consistent observation table

When translating this table we end up with the following candidate states {{b}, {ρ, ba}, {a}}, as we can see each observation row is t equivalent only to the ones in the same candidate state, so the clustering is trivial. When adding the transitions no indetermination arises so the candidate states with its transitions are mapped resulting in the following PDFA:

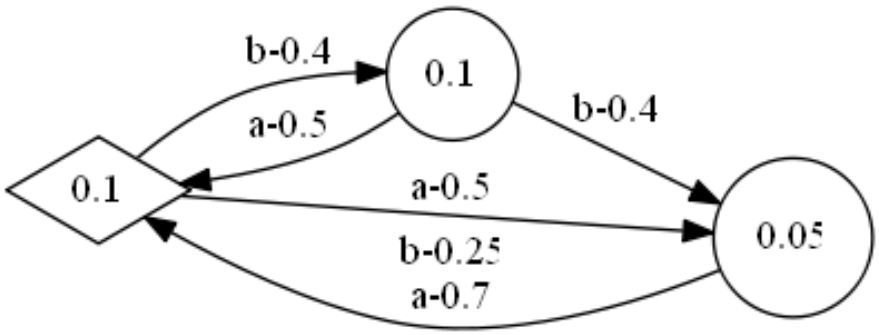


Figure 3.3: Second PDFA Extracted

This PDFA is again found not to be equivalent to the target one, and the "aab" counterexample is returned, we then add "aa" and "aab" to RED and its continuations to BLUE, and proceed to make the table closed and consistent again, resulting in the table:

$OT5$	$	a	b	aa	ba
ρ	0.1	0.5	0.4	0.7	0.5
a	0.05	0.7	0.25	0.5	0.5
b	0.1	0.5	0.4	0.5	0.7
aa	0.1	0.5	0.4	0.5	0.5
ba	0.1	0.5	0.4	0.7	0.5
aab	0.1	0.5	0.4	0.6	0.6
ab	0.1	0.5	0.4	0.5	0.5
bb	0.05	0.7	0.25	0.5	0.5
aaa	0.1	0.5	0.4	0.6	0.6
baa	0.05	0.7	0.25	0.5	0.5
bab	0.1	0.5	0.4	0.5	0.7
aaba	0.08	0.6	0.32	0.6	0.6
aabb	0.08	0.6	0.32	0.6	0.6

Table 3.7: Third closed and consistent observation table

When translating this table we end up with candidate states {{a},{aa}, {b}, {ρ, ba}, {aab}}. At first "aab" was added to the same state as ρ and "ba", but it generated indetermination so it was removed to another state. When concatenating an "a" to ρ, "ba" and "aab" we en up with "a", "baa" and "aaba", so both ρ and "ba" have a transition to the state containing "a", but "aab" has a transition to the state containing it, since "aaba" observation is closer to the centroid (0.1, 0.5, 0.4, 0.6666667, 0.5333333) than to the observation of "a" which matches the centroid.

After separating this observation to a new state and mapping the transitions again, the indetermination is broken and we translate to the following PDFA:

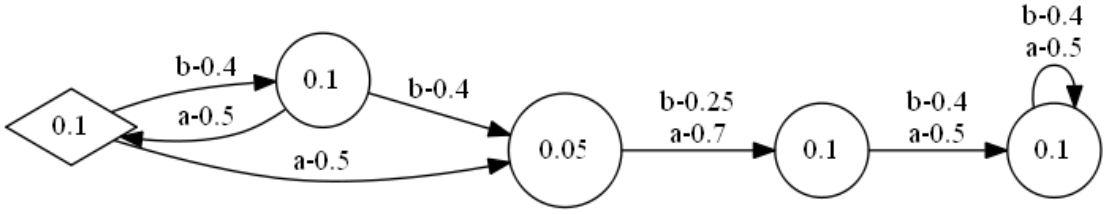


Figure 3.4: Equivalent PDFA

This PDFA while not identical to the target PDFA known by the teacher is indeed equivalent to it with a tolerance of 0.1 and found so by the teacher, it is then returned and the extraction is completed.

3.3 Multi hypothesis PDFA extraction

After considering the greedy strategy used by the Single Hypothesis version, we thought that a new strategy could be created in which all Hypothesis are considered when translating an observation table. With this motivation we created this second version of the algorithm.

3.3.1 Learner

The learner behaves almost identically to the learner in the previous version, the only difference is that it receives a list of PDFAs from the translation strategy that then passes to the teacher, which returns a counter example if none of the PDFAs are equal, or one of equivalent PDFAs in the list otherwise.

3.3.2 Teacher

As in the previous version, the teacher answers the queries performed by the learner. The main difference is in the equivalence query, since in this case, a list of PDFAs is received by the teacher.

Algorithm 6: Multi Hypothesis Equivalence Query

```
   Input  : list(PDFA) As, TestData TD, Tolerance T
   Output: Learned, Counterexample, Model
1  TestedPDFAs ← list;
2  foreach A ∈ As do
3  |  Error, Counterexample, Learned ← TestPDFA(A, TargetModel,
   |    TD, T);
4  |  AddToList(TestedPDFAs, { Error, Counterexample, Learned, A};
5  Learned, Model ←
     FindEquivalentModelWithSmallestError(TestedPDFAs);
6  if Learned = False then
7  |  Counterexample ← FindShortestCounterexample(TestedPDFAs);
8  return Learned, Counterexample, Model
```

In this case (Algorithm 6), the teacher receives a list of models for comparison with the target language model. The same comparison done in the first version

is performed for each automaton of the list, adding also the sum of the error of the logarithm of the probabilities for each tried sequence. If more than one model results to be equivalent, then the one with the smaller error is returned. If no automaton is equivalent, the shortest counterexample not used before is returned.

We define the error of PDFA A_1 with respect of the target PDFA A_0 over sequence w of length l as:

$$\mathrm{LPE}(A_1, A_0, w) = |\log(P_{A_0}(w)) - \log(P_{A_1}(w))| \tag{3.2}$$

$$\log(P_A(\sigma_1 \dots \sigma_l)) = \sum_{i=1}^{l} \log(P'_A[\sigma_i|\sigma_1 \dots \sigma_{i-1})] \tag{3.3}$$

where P_A and P'_A are the probabilities corresponding to the language model defined by A. By using the logarithm of the probability we avoid the underflow of multiplying probabilities, also accounting for the relevance of errors in longer sequences [24].

3.3.3 Multi hypothesis translation strategy

In this strategy we also iterate over each observation in RED, adding each one to the corresponding candidate state. The difference with the first strategy is that if an observation can belong to more than one candidate state (for the reasons mentioned in the previous section), a model is generated for each state to which it may belong, adding the observation under consideration to each possible state, one state for each model generated. Then we proceed to the next observation, doing the same procedure for each model generated in the previous step.

After adding all observations, generating several models in the process, we proceed to add the transitions. As in the previous strategy, more than one transition possibility may arise for a state-symbol pair. Each time this happens, new models are generated and all transitions are added, one to each model. Then we keep iterating for all models over the next states still not visited. This process continues until no new models are generated and all transitions are mapped.

Then, for each model, candidate states are translated into weighted states, weighted transitions are added and PDFAs are created in the same way that in the previous strategy. Despite having different candidate states, when translated

to PDFAs, we can end up with models that are identical (because the differences were in the sequences within candidate states and not because of the actual states and transitions), so the list is filtered, keeping only unique models that are then returned to the learner.

The objective of this strategy is that, by considering all possible hypothesis the learner can reach the PDFA closer to the target language model. Since we are working with a tolerance level, more than one distinct PDFA could result equivalent to the target, so we return the one that has the smaller difference from the target when predicting the word probability for a given sample.

3.3.4 Multi Hypothesis PDFA Extraction Run

This strategy proceeds identically to the previous one until the translation of Table 3.7. In each one of the previous iterations, the translation returns a single automaton since no indeterminations are present in the observation table.

When generating the intermediate states this version generates a state for ρ and “ba” together, one for “a”, one for ”b”, and one for “aa”. When considering “aab” its observation row is equal to all other observation rows except for ”a”, meaning that it can be added to any of the three other states, and so three different candidate models are generated with “aab” added to one of the three possible candidate states.

After generating all states, transitions are mapped solving the indetermination by adding a possible model for each possibility, this results in some cases in an explosion of possible models, since each time a new model is added its indeterminations also need to be solved. Also, different models, which have at least one different prefix in at least one state or one different transition for at least one state, may end up translating to the same final PDFA, so at the end of the process we filter the list of PDFAs returning those that are actually different.

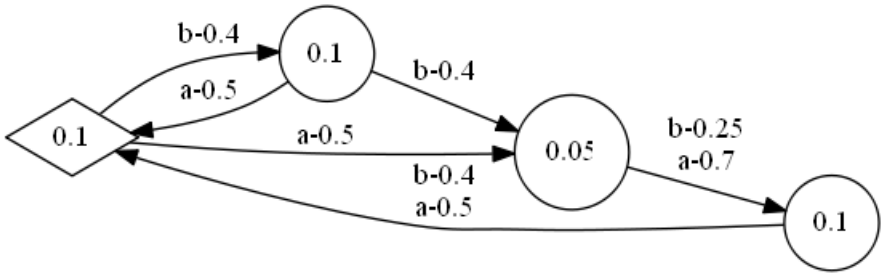


Figure 3.5: Equivalent PDFA

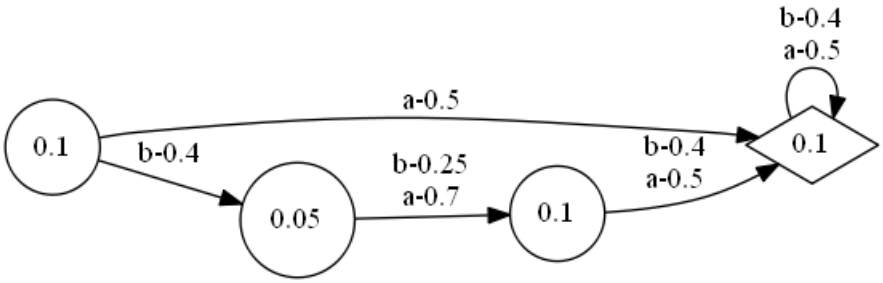


Figure 3.6: PDFA 2

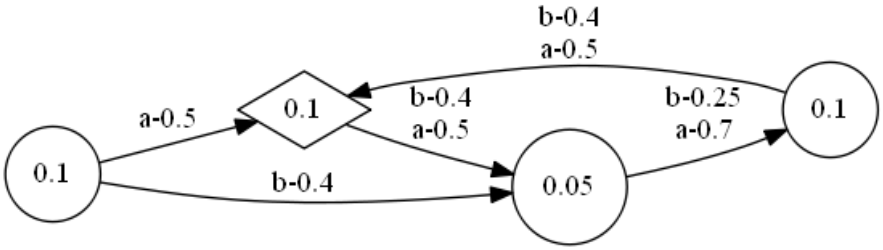


Figure 3.7: PDFA 3

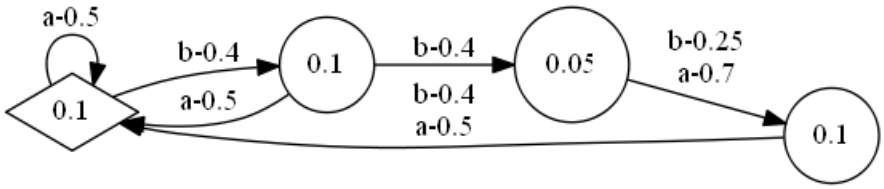


Figure 3.8: PDFA 4

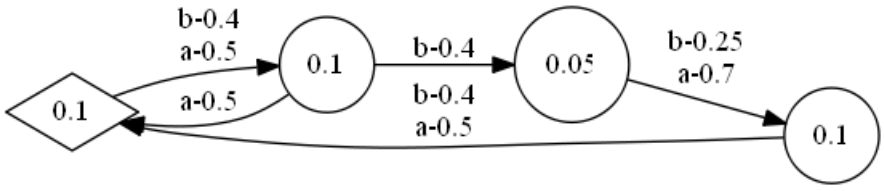


Figure 3.9: PDFA 5

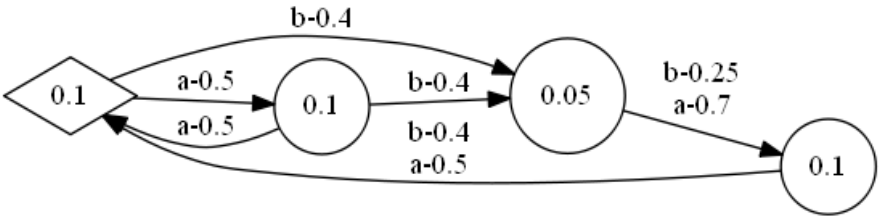


Figure 3.10: PDFA 6

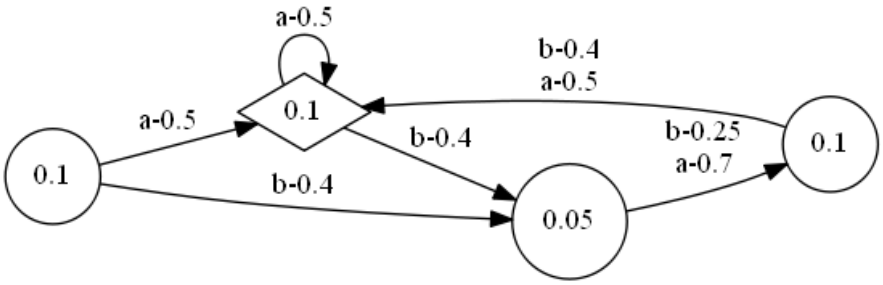


Figure 3.11: PDFA 7

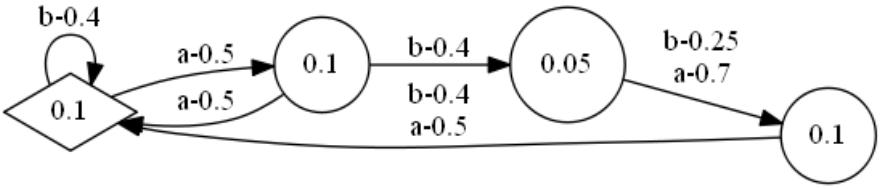


Figure 3.12: PDFA 8

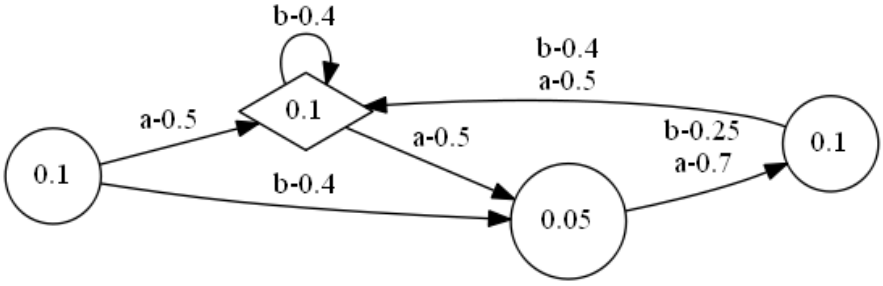


Figure 3.13: PDFA 9

Figures 3.5 to 3.13 above show the nine PDFAs translated from the observation table after the last iteration. When passed to the teacher it found only PDFA from Figure 3.9 to be equivalent to the target model. This PDFA is returned and the process is completed.

We can see again that this PDFA is not identical to the target one and neither to the one learned with the single hypothesis strategy, but all three are equivalent with a tolerance of 0.1 meaning that there is no sequence for which, after traversing it, the probabilities of occurrence of every symbol in the alphabet and the terminal symbol differ in more than 0.1 from one another.

3.4 Automaton implementation

We made a general WFA implementation, comprising a non-empty set of states and an alphabet. Each state has an initial and final weight and a transitions dictionary with the symbol of the transition as key, and as a value a list of tuples weight, next-state.

The restrictions to PDFA extraction is done during the translation, giving only one state an initial weight of one while the rest have a weight of zero, also, only one transition per state per symbol is allowed.

3.5 Key differences with the work in [3]

The work in [3] is closely related to this one, so it results pertinent to compare both approaches. As it was already mentioned, we used the tolerance level based on their work.

As both works are based on Angluin's L^* algorithm, many similarities can be found, in the use of a learner and a teacher, an observation table comprising a RED and a BLUE part, the use of queries and counterexamples, etc.. In the translation part is where most differences can be found.

During the translation, [3] executes a clustering algorithm (from sklearn), called DBScan. This algorithm dos not respect equality in terms of tolerance, so it may end up adding observations that are not within the given tolerance level inside the same candidate state. Therefore, [3] must proceed to refining the partition returned

by the clustering algorithm. In this way, non-equivalent words are separated by generating new candidate states. Moreover, after adding transitions, the algorithm proposed in [3] splits candidate states that generate non-determinism, in order to generate a deterministic automaton.

An important aspect of our algorithm is that it always clusters observations respecting equivalence modulo tolerance, therefore avoiding the need for splitting clusters afterwards. Nevertheless, in the case of single hypothesis, when adding transitions, we do have to separate when different observations within a candidate state point to different candidate states.

On the other hand, in the multi-hypothesis version of our algorithm no splitting is ever done. In this case, indetermination is solved by adding a new candidate model for every possible transition. Here, the goal is to consider all hypothesis, aiming to obtain the closest PDFA to the target language that may not be found when translating using a greedy clustering technique, either DBScan or our T-equivalence preserving one.

4 Experimental results

In this chapter we will show the results of applying the two versions of the PDFA extraction algorithm to different language models. We will start extracting directly from PDFAs that will be known by the teacher and then we will train RNNs with samples from these PDFAs and extract from language models constructed with these RNNs.

Since the first work regarding PDFA extraction from a black box is the one in [3], one of the goals was to compare ourselves with their results, so we borrowed the PDFAs used by them, which you can see in the next images. We will use the PDFA from Section 3 (which will be the basic automaton) along with seven adaptations of the Tomitas Grammars to transform them into PDFAs (Figure 4.1), and what [3] refers to as "Unbounded History Languages" comprising three cyclical PDFAs, two of which iterate over all states regardless of the input symbol and one PDFA similar to Tomitas 5 but with closer probabilities for different transitions (Figure 4.2).

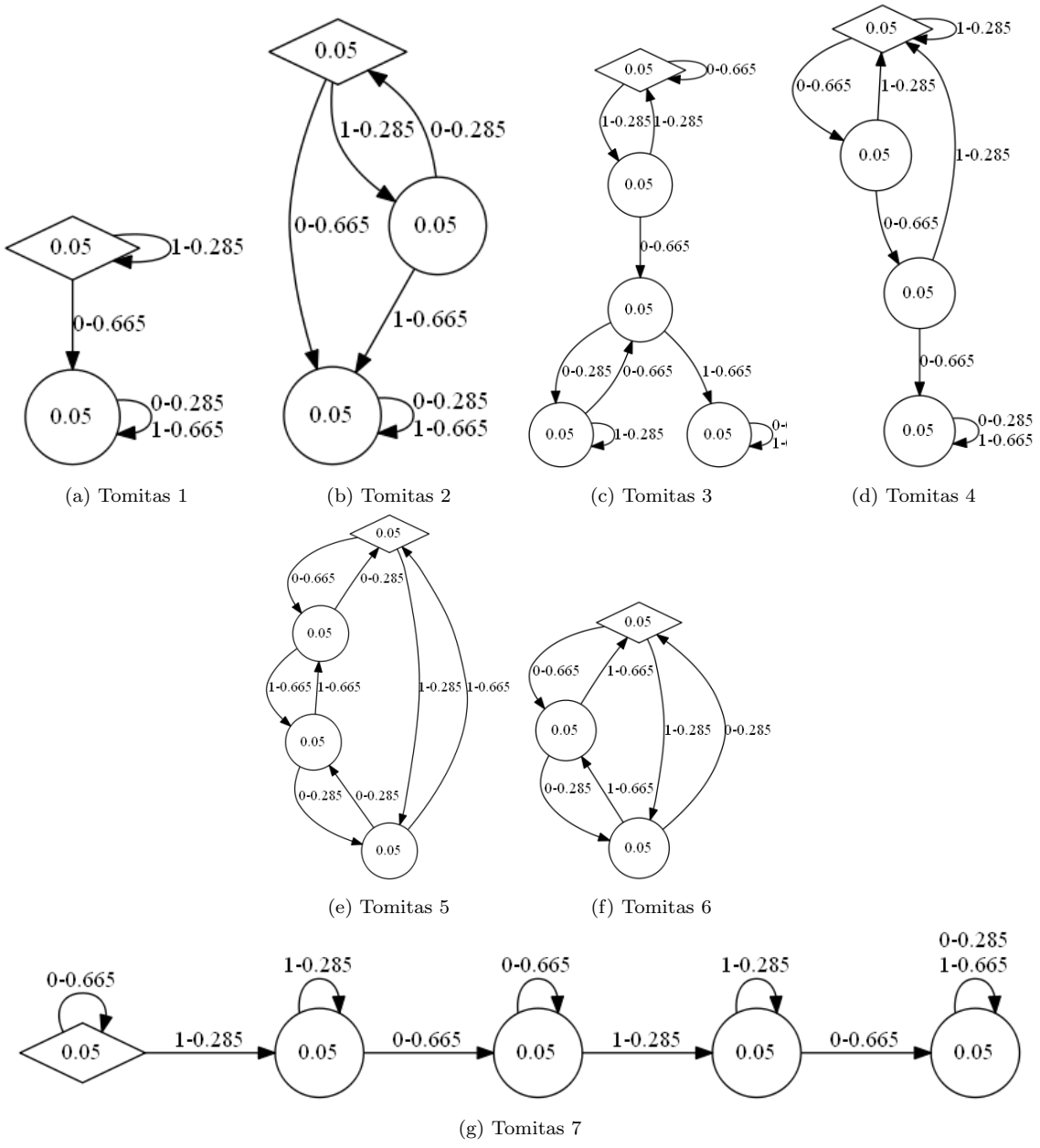
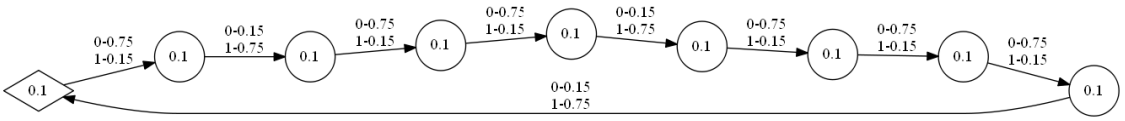
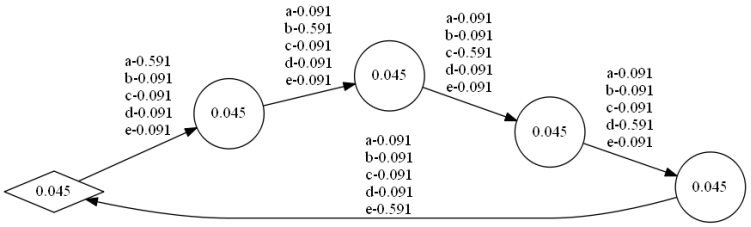


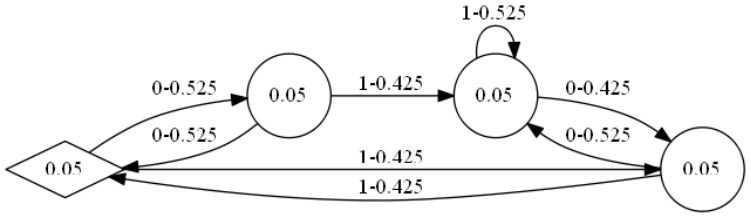
Figure 4.1: PDFAs addaptations of Tomitas Grammars borrowed from [3]



(a) UHL 1



(b) UHL 2



(c) UHL 3

Figure 4.2: UHL PDFAs borrowed from [3]

4.1 Metrics

We use three different metrics to compare the PDFA learned to the language model known by the teacher. These are, Word Error Rate (WER), Normalized Discounted Cumulative Gain (NDCG), metrics used in [3] to be able to compare with them and the average of the absolute difference of the logarithm of the word probability.

4.1.1 WER

We will use the WER definition given in [3] instead of the classic one used for text recognition and translation (WER = (Substitutions + Insertions + Deletions) / (Original Number of Words)). In [3] WER is calculated as the average amount of times the model does not predict the most likely next symbol correctly, when compared to the target language model.

Let P'_M be a set of probabilities distributions for the next symbol over all symbols in the alphabet plus the terminal symbol for language model M, $P'_M(w)$ the probability distribution after traversing word w, and the most likely symbol $\text{MLS}(P'_M(w))$, that is, the symbol with the greatest probability, we can define:

$$L(M_0, M_1, w) = \begin{cases} 0 & \text{if } \text{MLS}(P'_{M_0}(w)) = \text{MLS}(P'_{M_1}(w)) \\ 1 & \text{otherwise} \end{cases} \tag{4.1}$$

Then the WER of a model M_1 against a target model M_0 over the sample S of words is:

$$\text{WER}(M_0, M_1, S) = \frac{\sum_{w \in S} L(M_0, M_1, w)}{|S|} \tag{4.2}$$

4.1.2 NDCG

NDCG is a metric for ranking systems, in this case it is used to measure how well a model ranks next symbols, from most probable to least probable, after traversing a word, when compared to the actual rank given by the target language model.

We will explain this metric following the definitions in [25], applied to our own case scenario.

We start by defining Cumulative Gain of model M as the sum of the relevance of each next symbol after traversing word w.

$$CG_M(w) = \sum_{\sigma \in \Sigma_\$} P'_M[\sigma \mid w] \tag{4.3}$$

When working with probability distribution this metric will always account to one. We now add the position of each symbol as a discount, arriving at the definition of Discounted Cumulative Gain:

$$DCG_M(w) = \sum_{\sigma \in \Sigma_\$} \frac{P'_M[\sigma \mid w]}{\log_2(pos_{M,w}(\sigma) + 1)} \tag{4.4}$$

where $pos_{M,w}(\sigma)$ is the position of σ in the ordered array of probabilities $P'_M(w)$.

This metric does not take into account the correct ranking given by the target language model, for this we define Normalized Discounted Cumulative Gain for model M_1 compared to target model M_0 as:

$$NDCG(M_1, M_0, w) = \frac{DCG_{M_1}(w)}{DCG_{M_0}(w)} \tag{4.5}$$

NDCG yields a result between zero and one, being one the perfect score, meaning that M_1 ranks every symbol exactly as M_0.

For our work we use the NDCG scikit function that takes an array of predictions (taken from a data set of words), an array of the correct predictions for the same data set, and returns the average NDCG over the array.

4.1.3 LPE

This metric has already been explained in Subsection 3.3.2.

4.2 Methodology

For each language model we run the algorithm one hundred times for each version of the algorithm, and use a test sample of five thousand words to obtain the different metrics. The test sample is always the same for each of the runs.

For each run we present a table with each language model in the rows and the results for each version of the algorithm in the columns (SH for single hypothesis and MH for multi hypothesis). The results shown are the extraction time in seconds, number of equivalence queries, number of states in the equivalent PDFA and the three metrics presented earlier.

4.3 Extraction from PDFAs

In this section we show the results of applying both versions of the algorithm to extract from automata. Given the nature of the algorithm and the exact results obtained when querying a teacher with a PDFA as language model, if a low enough tolerance (possibly zero) is given, both versions of the algorithm arrive at the same PDFA known by the teacher.

4.3.1 First Run

For our first run we used a tolerance of 0.1 as in the manual run, in the next table we show the results for both versions of the algorithm.

	Time		E. Q.		State Count		WER		NDCG		LPE	
	SH	MH	SH	MH	SH	MH	SH	MH	SH	MH	SH	MH
Basic	0.30	0.62	3.00	3.00	2.00	4.00	0.00	0.00	1.00	1.00	0.23	0.22
T1	0.25	0.34	1.00	1.00	2.00	2.00	0.00	0.00	1.00	1.00	0.00	0.00
T2	0.32	0.35	1.00	1.00	3.00	3.00	0.00	0.00	1.00	1.00	0.00	0.00
T3	0.38	0.40	3.00	3.00	5.00	5.00	0.00	0.00	1.00	1.00	0.00	0.00
T4	0.35	0.36	2.00	2.00	4.00	4.00	0.00	0.00	1.00	1.00	0.00	0.00
T5	0.32	0.35	2.00	2.00	4.00	4.00	0.00	0.00	1.00	1.00	0.00	0.00
T6	0.35	0.35	1.00	1.00	3.00	3.00	0.00	0.00	1.00	1.00	0.00	0.00
T7	0.36	0.39	2.00	2.00	5.00	5.00	0.00	0.00	1.00	1.00	0.00	0.00
UHL1	0.48	0.50	3.00	3.00	9.00	9.00	0.00	0.00	1.00	1.00	0.00	0.00
UHL2	39.54	30.24	1.00	1.00	5.00	5.00	0.00	0.00	1.00	1.00	0.00	0.00
UHL3	0.18	0.19	3.00	3.00	4.00	4.00	0.00	0.00	1.00	1.00	0.00	0.00

Table 4.1: First Run Results

We can see from the results table that all target models, aside from the Basic one, where perfectly learnt. The graphs for each of the PDFA extracted are identical to the ones shown in the beginning of this chapter, and the ones for the Basic PDFA are identical to the ones shown in the manual run, so they are not added again here.

Another run with a zero tolerance was performed and all language models were perfectly learnt including the Basic one.

There are a few things to highlight, firstly, the Single Hypothesis version is faster than the Multi Hypothesis one except in UHL2 where MH is faster than SH, but the Multi Hypothesis one scored better in the only different metric, the LPE for the Basic PDFA. Even when the extraction of the Basic PDFA was not exact, the WER and NDCG scored perfectly because the errors were small enough.

These runs were designed to prove that the algorithms work consistently, and are capable of learning a PDFA from a language model, which was the main goal of this work.

4.4 Extraction from Recurrent Neural Network language models

In this section we first make a short presentation of Recurrent Neural Networks (RNNs) and the language model based on them, used in this work. We trained these language models which each of the PDFAs presented earlier and extracted a PDFA from the trained model, utilizing each of the algorithm versions.

Since the definition of the language model was outside of the scope of this work, we used the language model already present in the extraction framework. without much tuning for the architecture or the hyperparameters, this resulted in less than optimal RNN based language models, and constitute one of the main issues to focus on in the future as it will be discussed in chapter "Future Work".

4.4.1 Recurrent Neural Networks

RNNs are a type of Artificial Neural Network that apply to time series data, using the outputs of networks units at time t as input to other units at time t + 1, supporting directed cycles in the network graph [26].

In [27] RNN are characterized as a dynamical system:

$$s^{(t)} = f(s^{(t-1)}, \theta), \tag{4.6}$$

where $s^{(t)}$ represents the state in time t which, in this basic example depends only in the state of the system in t - 1.

If we add external input at time t we get the more general equation:

$$s^{(t)} = f(s^{(t-1)}, x^{(t)}; \theta), \tag{4.7}$$

where $x^{(t)}$ represents the input in time t.

A normal way of training such RNN is by stating a finite number of recurrent steps τ, and unfolding the network applying the definition τ times.

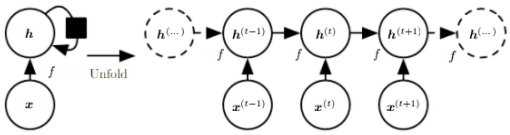


Figure 4.3: Unfold of recurrent network with no output, borrowed from [27]

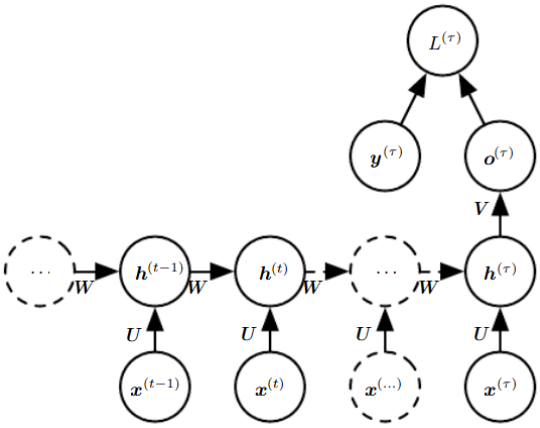


Figure 4.4: Unfold of recurrent network with one output, borrowed from [27]

4.4.2 Long Short-Term Memory (LSTM)

Earlier methods for sequence classification and prediction such as simple RNNs tend to be tailored to particular use cases and do not scale well to long time dependencies. LSTMs on the other hand are general, effective at capturing long term temporal dependecies and do not suffer from optimization hurdles that simple RNNs do. As such have been effectively used to tackle several problems involving sequences, handwriting recognition, language modeling, language translation, acoustic modeling of speech, speech recognition, etc [28].

LSTMs units are composed of a memory cell, an input gate used to protect the memory content from perturbations from irrelevant inputs and an output gate which protects other units from irrelevant memory content of the present unit [29]. Later, a forget gate was introduced which learn to reset memory blocks once the memory cell's information becomes outdated [30].

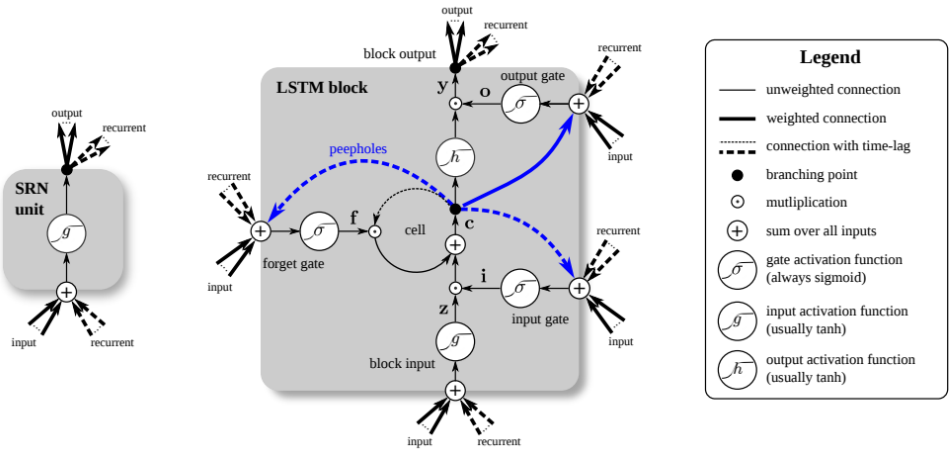


Figure 4.5: Comparison between a Simple Recurrent Network unit and a LSTM unit borrowed from [28]

4.4.3 Language model implemented for this Work

The language model used in this work, is based on a LSTM Neural Network with two LSTM layers with a Dense layer in between and a last Dense layer with a softmax activation function that given a word, returns the probability of each symbol including the terminal symbol. It uses a learning rate of 0.01, an Adam optimizer, for the loss it uses categorical crossentropy and it calculates the accuracy.

4.4.4 Run with the base RNN architecture and hyperparameters

For this first run we train a LSTM with each of the PDFAs. To achieve this, we sample the PDFA 100.000 times according with its probability distribution, and use this sample to train the RNN. The results of the training were subpar, with an accuracy on the last symbol of around 65% for all PDFAs, except for UHL2 which the accuracy was even worse, 40% which means that the RNN never converged and its outputs are hardly useful.

Despite this results we extracted PDFAs from these language models to measure the ability of the algorithms to extract a PDFA from a RNN based language model. The results are presented in the following table.

	Time		E. Q.		State Count		WER		NDCG		LPE	
	SH	MH	SH	MH	SH	MH	SH	MH	SH	MH	SH	MH
Basic	1.70	1.76	2.00	2.00	4.00	4.00	0.00	0.00	1.00	1.00	0.07	0.07
T1	1.67	1.65	1.00	1.00	1.00	1.00	0.00	0.00	1.00	1.00	0.12	0.12
T2	1.71	1.72	2.00	0.00	4.00	4.00	0.00	0.00	1.00	1.00	0.02	0.02
T3	2.26	2.30	3.00	3.00	11.00	11.00	0.00	0.00	1.00	1.00	0.08	0.08
T4	2.28	2.33	4.00	4.00	10.00	10.00	0.00	0.00	1.00	1.00	0.74	0.74
T5	1.92	1.94	4.00	2.00	4.00	4.00	0.00	0.00	1.00	1.00	0.40	0.40
T6	2.27	2.52	4.00	4.00	10.00	10.00	0.00	0.00	1.00	1.00	0.15	0.15
T7	2.47	2.52	3.00	3.00	11.00	11.00	0.00	0.00	1.00	1.00	0.23	0.23
UHL1	2.32	2.34	3.00	3.00	7.00	7.00	0.00	0.00	1.00	1.00	0.21	0.21
UHL2	447.78	-	1.00	-	131.00	-	0.03	-	0.99	-	0.07	-
UHL3	1.94	3.01	2.00	2.00	4.00	4.00	0.00	0.00	1.00	1.00	0.01	0.01

Table 4.2: RNN Language Model First Run Results

As we can see in the previous table, the PDFAs almost perfectly represent the language models known by the RNNs, except for the case of UHL2 that the results from the RNN learning never achieved convergence, but even in that case the WER is close to zero and the NDCG close to one, meaning that the PDFA extracted closely resembles the language model known by the RNN. When extracting from the RNN trained with UHL2, the multi hypothesis version could not complete a full run. We think that the divergence in the RNN train, produced a myriad of different observation rows that when combined produced an unmanageable amount of candidate models.

Nevertheless, when we analyze the graphs of the PDFAs extracted, we can see that they are not equal to the ones the RNNs were supposed to learn. Before presenting the graphs, it is important to highlight that the difference is not a result of the extraction process from the RNN language model, but of the RNN training process itself, which could not learn the correct next symbol probabilities.

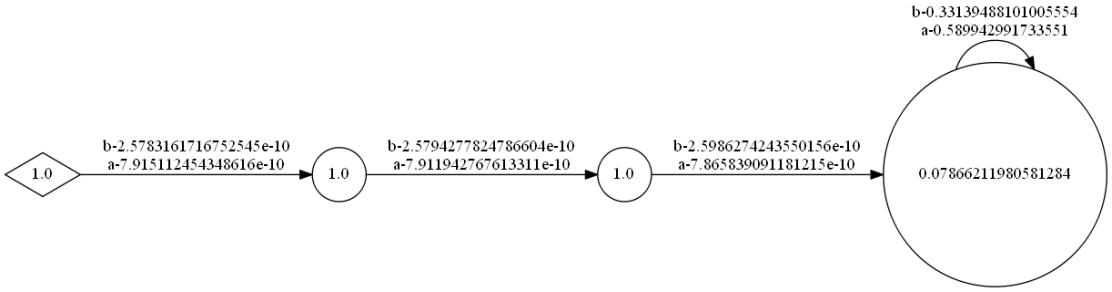


Figure 4.6: RNN Basic

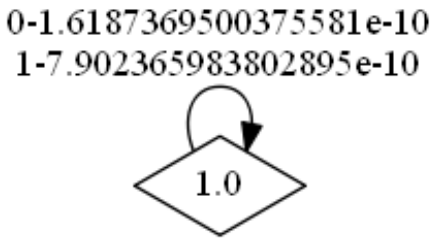


Figure 4.7: RNN Tomitas1

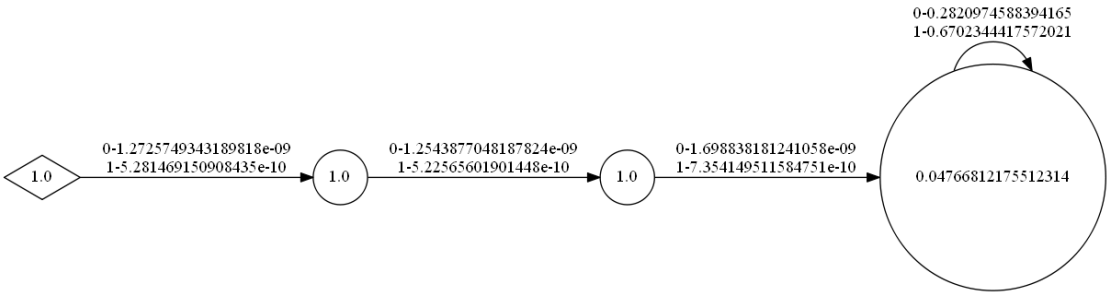


Figure 4.8: RNN Tomitas2

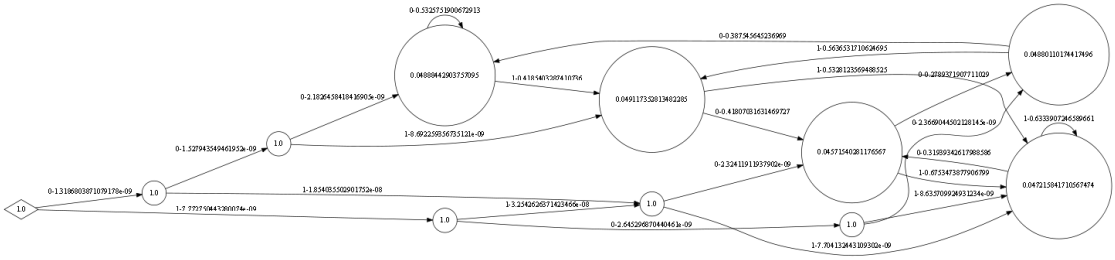


Figure 4.9: RNN Tomitas3

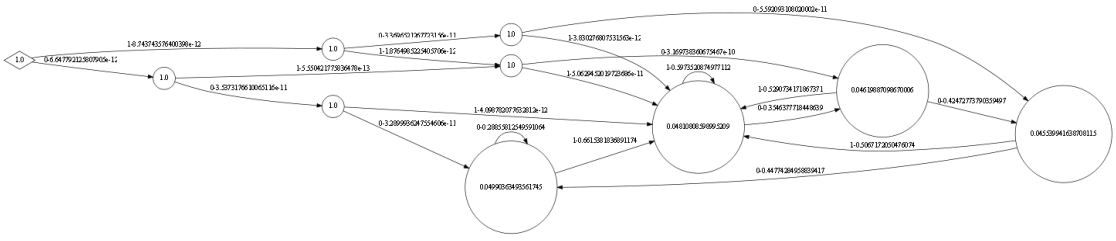


Figure 4.10: RNN Tomitas4

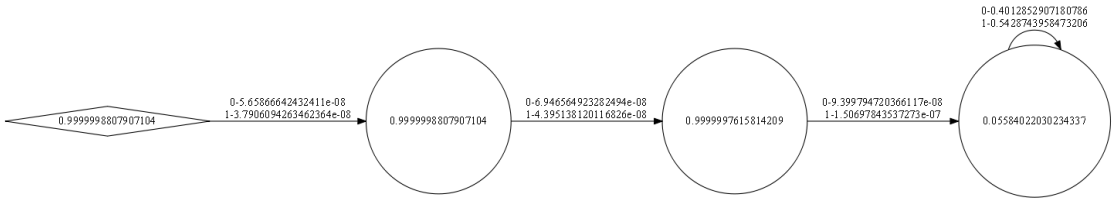


Figure 4.11: RNN Tomitas5

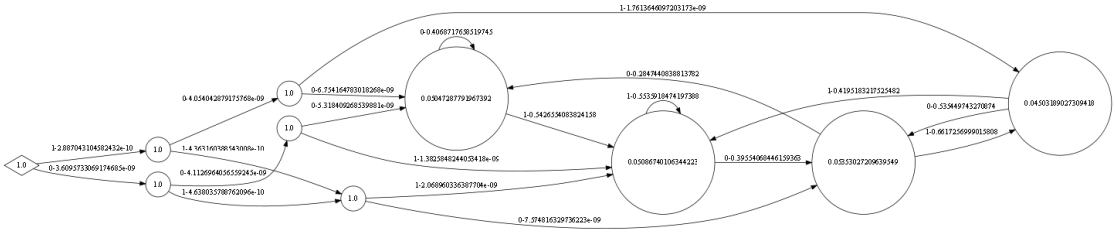


Figure 4.12: RNN Tomitas6

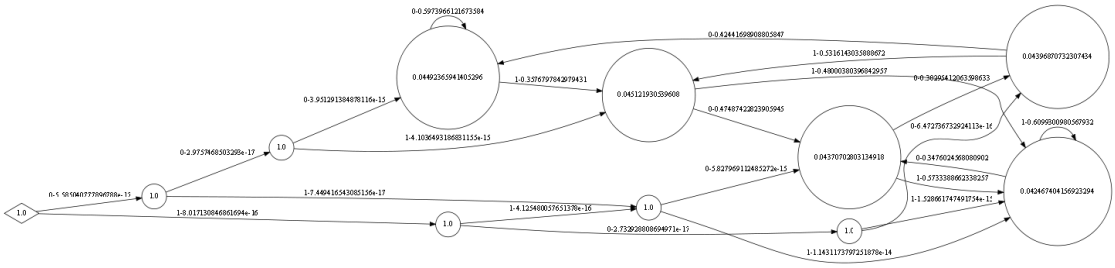


Figure 4.13: RNN Tomitas7

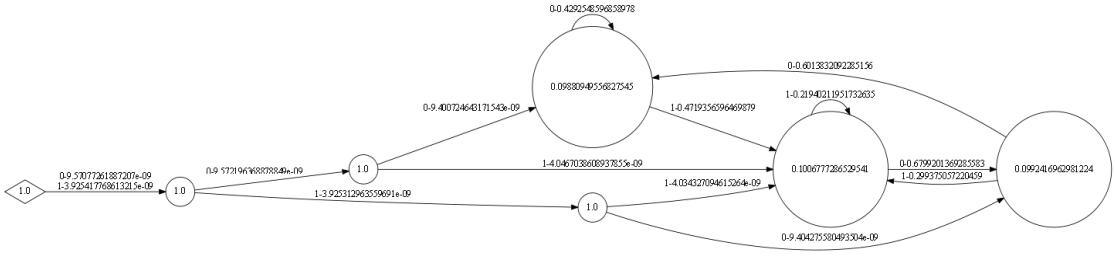


Figure 4.14: RNN UHL1

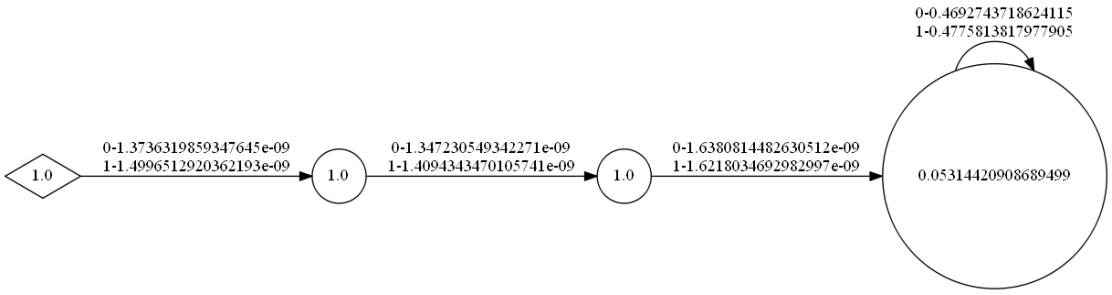


Figure 4.15: RNN UHL3

We can see in the figures that the RNNs usually learn that several short words have last symbol probabilities of one with transitions with probabilities of almost zero, this happens because of the inability of computers to operate with really small margins in floating point arithmetic. This fact was corroborated by directly querying the RNN language model for the empty sequence which returned a probability of one.

4.4.5 Representative run

We tried several modifications of the architecture, hyperparameters and sampling method, and usually the results were not good from the RNN training point of view. In a few cases we managed to achieve 100% accuracy or close during training, but after extracting the PDFAs we could see that the RNN had not learnt the probabilities correctly. Presently we will show a sample of these cases and the PDFAs extracted.

By sampling the Tomitas1 and Tomitas2 with words of max length equal to the number of states of the underlying PDFA we managed to obtain 100% accuracy and 0% loss. Then we extracted from these RNN language models and obtained the following results:

	Time		E. Q.		State Count		WER		NDCG		LPE	
	SH	MH	SH	MH	SH	MH	SH	MH	SH	MH	SH	MH
Tomitas1	2.79	2.85	1.00	1.00	1.00	1.00	0.00	0.00	1.00	1.00	0.03	0.03
Tomitas2	4.33	2.73	1.00	1.00	14.00	14.00	0.00	0.00	1.00	1.00	0.09	0.09

Table 4.3: Results from RNNs with 100% Accuracy

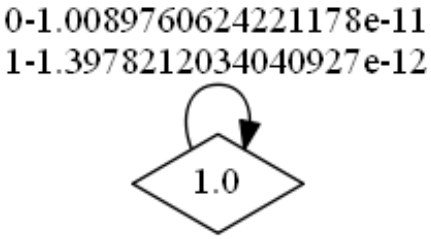


Figure 4.16: RNN Tomitas1

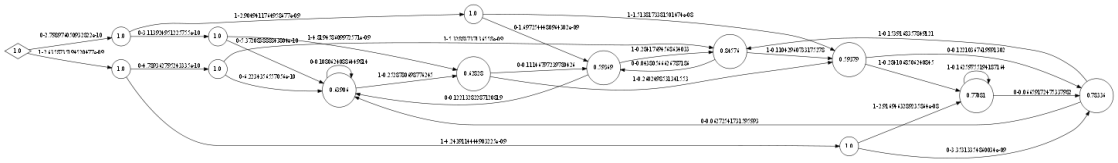


Figure 4.17: RNN Tomitas2

In these results we can see that a RNN that achieved perfect metrics, still did not learn the correct probability distributions. This result shows the importance of this line of investigation towards a more understandable and explainable AI.

5 Future work

After testing the correctness of the PDFA extraction algorithms, several lines of investigation remain open. In this chapter we make a brief account of the areas were we will focus from now on.

5.1 RNN based language model

We need to improve the RNN architecture and correctly tune the hyperparameters in order to reach a RNN that can learn the probabilities of the underlying PDFA. With the help of the algorithms proposed in this work we can easily compare what the RNN has learnt with the PDFA from where the samples came.

5.2 Compare our work with [3]

Being the first publication on active learning techniques applied to PDFA learning, one goal is compare the results of our algorithms with the ones obtained by [3]. For this we need to fulfill the previous point in order to measure time and correctness against their work when extracting from RNNs based language models. In [3] they also compare themselves with other extraction techniques, such as spectral learning and n-gram, another objective we have for future investigation is to be able to compare our work with those techniques.

5.3 Optimize the multi hypothesis version of the algorithm

As we mentioned when describing the algorithm, in the multi hypothesis version, we can have an explosion of possible models that when translated to actual PDFAs

end up being the same in most cases. We need to optimize the algorithm to add some mechanism to prune the candidate model tree and candidate state tree when iterating over all models.

5.4 Apply Probably Approximately Correct (PAC) framework to PDFA extraction

In [1], PAC framework is applied to DFA extraction from RNNs, our goal is to be able to adapt their findings to the stochastic nature of PDFA extraction, considering the many hypothesis we take into consideration in the multi hypothesis version of the algorithm.

In order to answer the EQ, the teacher needs to perfectly know the regular language, but this is not the case when working with RNNs. For cases like this, Angluin proposes the use of the Probably Approximately Correct framework developed by Leslie Valiant in [31].

The PAC framework states that a concept class C is PAC learnable if there exists a polynomial time algorithm A and a polynomial $p(n, 1/\delta, 1/\epsilon)$ with n being the number of Hypothesis and δ and $\epsilon \in (0, 1)$ such that for all $n > 1$, A returns a hypothesis $h \in H_n$ that with a probability of $1 - \delta$ incurs in an error smaller than ϵ, defining $error(h)$ as the proportion of times h disagrees with C [32].

When applied to L^*, a δ and an ϵ need to be provided in order to calculate for each iteration (which generates a new hypothesis increasing n) the number of samples we need in order to perform an statistical equivalence test that ensures that the DFA learnt incurs in an error smaller that ϵ with a probability of $1 - \delta$ over a whole run of L^*.

The difficulty in adding a PAC equivalence test in the context of learning a PDFA lies in the definition of an appropriate error measure. Possible candidates for such error are WER and LPE.

6 Conclusions and lessons learned

In this work we presented two novel algorithms for PDFA extraction from language models, and tested them for correctness. We also used these algorithms to show that the language model defined in the framework was not correctly learning probability distributions.

During the development of this work we faced the challenge of addressing a problem that had only been tackled in [3]. We took some ideas from their work but decided to come with new strategies to try to achieve better results, either in processing time or correctness. We ended up adapting the L^* algorithm already implemented for DFA extraction to our own problem using the tolerance concept proposed in [3]. The most complex part to adapt was the translation from the closed and consistent observation table to the PDFA, and we decided to implement two different and completely new versions of it.

We still have ahead of us the task of measuring our work against similar works, spectral learning, n-gram and especially the work of [3], for this we need to work in the direction mentioned in Chapter 5.

The process of creating and testing these algorithms allowed us to dive into the field of grammatical inference and learn about equivalences between RNNs and different automata. One of the most important lessons we take from this is the usefulness of techniques aiming to make the inner workings of RNNs more understandable. It was really eye opening to see the PDFAs extracted from RNNs which achieved an accuracy of close to 100% and realizing that the probability distributions learnt by the RNN were not at all correct with respect to the PDFA that generated the training sample.

Even though in these cases we knew the underlying PDFAs, the results are no less useful when working with black boxes. If we look at the extracted PDFAs, a last symbol probability of one on several states should at least be a red flag that something may not be right.

We will continue this line of investigation, to keep on contributing with the framework developed by the AI research group which we are trying to transform in a reference tool for understanding sequence process, explainable AI, etc..

7 Bibliographic references

[1] F. Mayr and S. Yovine, "Regular inference on artificial neuralnetworks," in *Machine Learning and Knowledge Extraction*, A. Holzinger *et al.*, Eds. Cham: Springer International Publishing, 2018, pp. 350–369.

[2] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, Nov. 1987.

[3] G. Weiss, Y. Goldberg, and E. Yahav, "Learning deterministic weighted automata with queries and counterexamples," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8560–8571.

[4] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. USA: Prentice Hall Press, 2009.

[5] N. Xie, G. Ras, M. van Gerven, and D. Doran, "Explainable deep learning: A field guide for the uninitiated," 04 2020. [Online]. Available: https://arxiv.org/pdf/2004.14545v1.pdf

[6] D. Gunning, "Darpa's explainable artificial intelligence (xai) program," in *Proceedings of the 24th International Conference on Intelligent User Interfaces*, ser. IUI '19. New York, NY, USA: Association for Computing Machinery, 2019, p. ii. [Online]. Available: https://doi.org/10.1145/3301275.3308446

[7] S. Shalev-Shwartz and S. Ben-David, "Understanding machine learning - from theory to algorithms," 2014. [Online]. Available: https://www.cs.huji.ac.il/~shais/UnderstandingMachineLearning/

[8] B. Balle, X. Carreras, F. M. Luque, and A. Quattoni, "Spectral learning of weighted automata: a forward-backward perspective," *Machine learning*, no. October, p. 1–31, Oct 2013. [Online]. Available: http://hdl.handle.net/2117/21075

[9] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed. USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[10] E. M. Gold, "Complexity of automaton identification from given data," *Inf. Control.*, vol. 37, pp. 302–320, 1978.

[11] C. de la Higuera, *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.

[12] A. Nerode, "Linear automaton transformations," *Proceedings of the American Mathematical Society*, vol. 9, no. 4, pp. 541–544, 1958. [Online]. Available: http://www.jstor.org/stable/2033204

[13] W. R. de Oliveira, M. C. P. de Souto, and T. B. Ludermir, "Turing's analysis of computation and artificial neural networks," *J. Intell. Fuzzy Syst.*, vol. 13, pp. 85–98, 2002.

[14] P. Indyk, "Optimal simulation of automata by neural nets," in *STACS 95*, E. W. Mayr and C. Puech, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 337–348.

[15] B. Bollig and Z. Marc, "Weighted automata," in *Weighted Automata*, 2011. [Online]. Available: https://www.cmi.ac.in/~madhavan/courses/qath-2015/reading/bollig-zeitoun-lecture-notes.pdf

[16] J. S. Golan, *Semirings*. Dordrecht: Springer Netherlands, 2003, pp. 1–26. [Online]. Available: https://doi.org/10.1007/978-94-017-0383-3_1

[17] E. W. Weisstein, "Monoid." [Online]. Available: https://mathworld.wolfram.com/Monoid.html

[18] Y. Goldberg and G. Hirst, *Neural Network Methods in Natural Language Processing*. Morgan amp; Claypool Publishers, 2017.

[19] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. USA: Cambridge University Press, 2008.

[20] J. Carlyle and A. Paz, "Realizations by stochastic finite automata," *J. Comput. Syst. Sci.*, vol. 5, pp. 26–40, 1971.

[21] M. Fliess, "Matrices de hankel," *Journal de Mathematiques Pures et Appliquees*, 1974.

[22] R. E. Bonner, "On some clustering techniques," *IBM Journal of Research and Development*, vol. 8, no. 1, pp. 22–32, 1964.

[23] A. Saxena, M. Prasad, A. Gupta, N. Bharill, O. P. Patel, A. Tiwari, M. J. Er, W. Ding, and C.-T. Lin, "A review of clustering techniques and developments," *Neurocomputing*, vol. 267, pp. 664 – 681, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231217311815

[24] N. A. Smith, "Probabilistic language models 1.0," 2017.

[25] Y. Wang, L. Wang, Y. Li, D. He, T. Liu, and W. Chen, "A theoretical analysis of NDCG type ranking measures," *CoRR*, vol. abs/1304.6480, 2013. [Online]. Available: http://arxiv.org/abs/1304.6480

[26] T. M. Mitchell, *Machine Learning*, 1st ed. USA: McGraw-Hill, Inc., 1997.

[27] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[28] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A search space odyssey," *CoRR*, vol. abs/1503.04069, 2015. [Online]. Available: http://arxiv.org/abs/1503.04069

[29] S. Hochreiter and J. Schmidhuber, "Long short-term memory," 1997. [Online]. Available: https://www.bioinf.jku.at/publications/older/2604.pdf

[30] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: continual prediction with lstm," in *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*, vol. 2, 1999, pp. 850–855 vol.2.

[31] L. Valiant, *Probably Approximately Correct: Nature's Algorithms for Learning and Prospering in a Complex World*. USA: Basic Books, Inc., 2013.

[32] D. Haussler, "Part 1: Overview of the probably approximately correct (pac) learning framework," 1995. [Online]. Available: http://web.cs.iastate.edu/~honavar/pac.pdf