

Tool-assisted induction of subregular languages and mappings

A Dissertation Presented

by

Alëna Aksënova

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Linguistics

Stony Brook University

May 2020

Abstract of the Dissertation

Tool-assisted induction of subregular languages and mappings

by

Alëna Aksënova

Doctor of Philosophy

in


Linguistics

Stony Brook University

2020

The last decade was very fruitful in the field of subregular research. New classes of subregular languages and mappings were uncovered for modeling natural language phenomena, and new learning algorithms were developed for these classes. The subregular approach has been successfully applied to phonotactics (Heinz, 2010a), rewrite processes in phonology and morphology (Chandlee, 2014), and even syntactic constraints over tree structures (Graf, 2018b). However, the rapid pace of the theoretical research has not been matched when it comes to engineering considerations. Many of the proposed learning algorithms have not been implemented yet, and as a result, their performance on concrete data sets is not known.

In my dissertation, I implement and experiment with some of the learners available for subregular languages and mappings. I test these learners on data that is modeled after linguistic phenomena such as word-final devoicing and

various types of harmony systems. The code for these evaluations is available as part of my Python package *SigmaPie*  (Aksënova, 2020b).


The findings of my thesis allow linguists and formal language theorists to assess possible applications of subregular techniques and approaches, in particular typology, cognitive science, and natural language processing.

Contents

1	Introduction	1
1.1	Subregular grammars	2
1.2	Linguistic motivation behind the subregular approach	4
1.3	Structure of the dissertation	6
2	Background	9
2.1	Modeling well-formedness conditions	10
2.1.1	Regular nature of natural language patterns	10
2.1.2	Subregular languages and their linguistic importance	15
2.1.3	Local restrictions as SL languages	19
2.1.4	Long distance restrictions as SP languages	24
2.1.5	Long-distant dependencies with blocking as TSL languages	29
2.1.6	Multiple long-distant dependencies with blocking as MTSL languages	36
2.1.7	Models of well-formedness conditions: summary	40
2.2	Modeling transformations	41
2.2.1	Formalizing transformations	41
2.2.2	Subsequential mappings	44
2.2.3	Left and right subsequential mappings	47
2.2.4	ISL and OSL mappings	49
2.2.5	Models of transformations: summary	54

2.3	Learning grammars from data	56
3	Learning languages	59
3.1	The experimental setup	60
3.1.1	Experimental pipeline	61
3.1.2	Natural languages	61
3.1.3	Artificial languages	62
3.1.4	Target patterns	64
3.2	Strictly local models	79
3.2.1	SL learning algorithm	79
3.2.2	Successful experiments	80
3.2.3	Unsuccessful experiments	84
3.2.4	SL experiments: interim summary	90
3.3	Strictly piecewise models	91
3.3.1	SP learning algorithm	91
3.3.2	Successful experiments	92
3.3.3	Unsuccessful experiments	96
3.3.4	SP experiments: interim summary	99
3.4	Tier-based strictly local models	100
3.4.1	TSL learning algorithm	101
3.4.2	Successful experiments	102
3.4.3	Unsuccessful experiments	109
3.4.4	TSL experiments: interim summary	111
3.5	Multiple tier-based strictly local models	112
3.5.1	MTSL learning algorithm	112
3.5.2	Successful experiments	115
3.5.3	Unsuccessful experiments	121
3.5.4	MTSL experiments: interim summary	123
3.6	Learning languages: summary	123

4	Learning mappings	128
4.1	The OSTIA algorithm	129
4.1.1	The pipeline	129
4.1.2	The successful example	133
4.1.3	The unsuccessful example	136
4.2	Learning experiments	142
4.2.1	Experimental setup	143
4.2.2	Target patterns	144
4.2.3	Experiment 1: word-final devoicing	146
4.2.4	Experiment 2: a single vowel harmony without blocking . . .	148
4.2.5	Experiment 3: a single vowel harmony with blocking	150
4.2.6	Experiment 4: several vowel harmonies without blocking . . .	151
4.2.7	Experiment 5: several vowel harmonies with blocking	152
4.2.8	Experiment 6: vowel and consonant harmonies without blocking	155
4.2.9	Experiment 7: vowel and consonant harmonies with blocking	157
4.2.10	Experiment 8: unbounded tone plateauing	157
4.2.11	Experiment 9: a “simple” first-last harmony	159
4.2.12	Experiment 10: a “complex” first-last harmony	161
4.2.13	Summary of the results	161
4.3	Beyond OSTIA	164
4.3.1	Specifying OSTIA	165
4.3.2	Fixing outputs of some input symbols	166
4.3.3	Other transduction learners	168
4.3.4	Learning groups of transducers	169
4.4	Learning processes: summary	171
5	Conclusion and future work	173
5.1	Summary of the results	173

5.1.1	<i>SigmaPie</i> 	174
5.1.2	Tool-assisted learning experiments	174
5.2	Future directions	178

List of Figures

1.1	Flow of chapters of this dissertation: Introduction, Background, Learning languages, Learning mappings, and Conclusion.	8
2.1	FSA for Russian compounding.	11
2.2	The extended Chomsky hierarchy from (Jäger and Rogers, 2012). . .	13
2.3	Some of the classes of the subregular hierarchy; the subregular classes discussed further in this chapter and in Chapter 3 are boxed. .	15
2.4	Evaluation of strings <i>mozg</i> , <i>mozk</i> , <i>mosg</i> and <i>mosk</i> by an SL grammar capturing obstruent cluster assimilation and word-final devoicing. .	20
2.5	Evaluation of strings <i>zəntəz</i> and <i>zəntəʒ</i> by an SP grammar capturing sibilant harmony in voicing and anteriority.	25
2.6	SP grammar incorrectly rules out Imdlawn Tashlhiyt word <i>smʁazaj</i> . .	26
2.7	SP grammar captures the UTP pattern.	27
2.8	Evaluation of strings <i>budε</i> , <i>bədu</i> and <i>rebõre</i> by a TSL grammar capturing Karajá vowel harmony in ATR.	32
2.9	Evaluation of strings <i>to:ro:d</i> , <i>to:ru:le:d</i> , <i>to:re:d</i> and <i>to:ru:lɔ:s</i> by a TSL grammar capturing Buryat vowel harmony in ATR and rounding. . .	34
2.10	Evaluation of strings <i>zbruz:a</i> , <i>ʒbruz:a</i> , <i>smʃazaj</i> and <i>zmʃazaj</i> by a MTSL grammar capturing Imdlawn Tashlhiyt sibilant harmony in voicing and anteriority.	39
2.11	An example of the FST.	43
2.12	Transducer for Buryat vowel harmony.	44

2.13	Subsequential FST for word-final devoicing.	46
2.14	Right subsequential FST for Tuareg regressive sibilant harmony. . . .	48
2.15	Relationship among left subsequential, right subsequential, OSL, and ISL functions; adapted from (Chandlee, 2014).	49
2.16	ISL application of the rule $a \rightarrow b/a _ a$ to <i>aaaaa</i>	52
2.17	OSL application of the rule $a \rightarrow b/a _ a$ to <i>aaaaa</i>	55
2.18	Relationship between a language \mathcal{L} and a grammar \mathcal{G}	57
3.1	The extracted TSL grammar evaluating strings (Experiment 3)	106
3.2	Experiment 7: the extracted MTSL grammar evaluating the ungrammatical strings <i>aabbotoob</i> and <i>aabbbaaap</i>	122
4.1	The main steps of OSTIA: BUILD, ONWARD, FOLD and PUSHBACK.	130
4.2	Non-onward and onward PTTs that are otherwise equivalent.	131
4.3	OSTIA pushes back the suffix <i>v</i>	132
4.4	FST for word-final devoicing obtained by OSTIA.	148
4.5	FST for a single vowel harmony without blocking obtained by OSTIA. . . .	149
4.6	The expected FST for a single vowel harmony with blocking.	151
4.7	The expected FST for several vowel harmonies without blocking. . . .	153
4.8	FST for vowel and consonant harmonies without blocking obtained by OSTIA.	156
4.9	FST for a “simple” first-last harmony obtained by OSTIA.	160
4.10	Some of the FSTs that can be built from the pair (<i>sim</i> , <i>seen</i>) in the “unbiased” way; to be contrasted with the following figure.	166
4.11	Some of the FSTs that can be built from the pair (<i>sim</i> , <i>seen</i>) if the output of the input symbol <i>s</i> is fixed to the output symbol <i>s</i>	167
4.12	Possible guesses of the transition that can be built after observing the pair (<i>ab</i> , <i>aab</i>).	170

List of Tables

2.1	Types of dependencies captured by some of the subregular classes. . .	16
2.2	Subregular patterns attested in natural languages and discussed in sections 2.1.3-6.	18
2.3	2-SL grammar for Russian obstruent voicing assimilation and word- final devoicing.	21
2.4	2-SP grammar for Tuareg sibilant harmony in voicing and anteriority.	25
2.5	3-SP grammar for Luganda unbounded tone plateauing.	27
2.6	2-TSL grammar for Karajá vowel harmony in ATR.	31
2.7	2-TSL grammar for Buryat vowel harmony in ATR and rounding. . .	33
2.8	2-MTSL grammar for Imdlawn Tashlhiyt sibilant harmony in voicing and anteriority.	38
3.1	German: <i>raw</i> \rightarrow <i>masked</i> representation.	66
3.2	German: <i>raw</i> \rightarrow <i>abstract</i> representation.	66
3.3	Finnish: <i>raw</i> \rightarrow <i>masked</i> representation.	68
3.4	Finnish: <i>raw</i> \rightarrow <i>abstract</i> representation.	69
3.5	A harmony in $[\alpha]$ exhibiting blocking effect; <i>abstract</i> representation. .	70
3.6	A harmony in $[\alpha]$ and $[\beta]$; <i>abstract</i> representation.	71
3.7	Turkish: <i>raw</i> \rightarrow <i>masked</i> & <i>abstract</i> representations.	73
3.8	A harmony in $[\alpha]$ and a harmony in $[\beta]$; <i>abstract</i> representation. . . .	75
3.9	A harmony in $[\alpha]$ and a harmony in $[\beta]$ with blockers; <i>abstract</i> representation.	76

3.10	The expected results of the language learning experiments.	78
3.11	SL learning of the word-final devoicing; abstract representation. . . .	82
3.12	SL learning of the word-final devoicing; masked representation. . . .	83
3.13	SL learning of the word-final devoicing; raw representation.	83
3.14	SL learning of a single harmony without blockers; abstract representation.	85
3.15	SL learning of a single harmony without blockers; masked representation.	85
3.16	SL learning of a single harmony with blockers; abstract representation.	86
3.17	SL learning of several vowel harmonies without blockers; abstract representation.	87
3.18	SL learning of several harmonies with blockers; abstract representation.	87
3.19	SL learning of vowel and consonant harmonies without blockers; abstract representation.	88
3.20	SL learning of vowel and consonant harmonies with blockers; abstract representation.	89
3.21	SL learning of unbounded tone plateauing; abstract representation. .	89
3.22	SL learning of first-last harmony; abstract representation.	90
3.23	SP learning of a single harmony without blockers; abstract representation.	93
3.24	SP learning of a single harmony without blockers; masked representation.	94
3.25	SP learning of a single harmony without blockers; raw representation.	94
3.26	SP learning of several vowel harmonies without blockers; abstract representation.	95
3.27	SP learning of vowel and consonant harmonies without blockers; abstract representation.	95

3.28	SP learning of unbounded tone plateauing; abstract representation. .	96
3.29	SP learning of the word-final devoicing; abstract representation. . . .	97
3.30	SP learning of a single harmony with blockers; abstract representation.	97
3.31	SP learning of several harmonies with blockers; abstract representation.	98
3.32	SP learning of vowel and consonant harmonies with blockers; abstract representation.	99
3.33	SP learning of first-last harmony; abstract representation.	100
3.34	TSL learning of the word-final devoicing; abstract representation. . .	103
3.35	TSL learning of the word-final devoicing; raw representation.	103
3.36	TSL learning of a single harmony without blockers; abstract representation.	104
3.37	TSL learning of a single harmony without blockers; masked representation.	105
3.38	TSL learning of a single harmony without blockers; raw representation.	105
3.39	TSL learning of a single harmony with blockers; abstract representation.	106
3.40	TSL learning of several vowel harmonies without blockers; abstract representation.	107
3.41	TSL learning of several harmonies with blockers; abstract representation.	107
3.42	TSL learning of several harmonies with blockers; masked representation.	108
3.43	TSL learning of vowel and consonant harmonies w/o blockers; abstract representation.	109
3.44	TSL learning of vowel and consonant harmonies with blockers; abstract representation.	110
3.45	TSL learning of unbounded tone plateauing; abstract representation.	111


3.46	TSL learning of first-last harmony; abstract representation.	111
3.47	MTSL learning of the word-final devoicing; raw representation. . . .	116
3.48	MTSL learning of a single harmony without blockers; abstract representation.	117
3.49	MTSL learning of a single harmony without blockers; raw representation.	117
3.50	MTSL learning of a single harmony with blockers; abstract representation.	118
3.51	MTSL learning of several vowel harmonies without blockers; abstract representation.	119
3.52	MTSL learning of several harmonies with blockers; abstract representation.	119
3.53	MTSL learning of several harmonies with blockers; raw representation.	120
3.54	MTSL learning of vowel and consonant harmonies w/o blockers; abstract representation.	120
3.55	MTSL learning of vowel and consonant harmonies with blockers; abstract representation.	121
3.56	MTSL learning of first-last harmony; abstract representation.	122
3.57	The expected results of the language learning experiments; repeated from the end of the section 3.1.4.	125
3.58	The expected vs. the actual results of the subregular language learning experiments; the experiment 8 cannot be conducted using MTSL learner because it is currently not available for $k > 2$; all other learners are used with $k = 2$	127
4.1	Parameters of the explored natural language patterns.	146
4.2	Results of OSTIA learning word-final devoicing.	147
4.3	Results of OSTIA learning a single vowel harmony without blocking.	149
4.4	Results of OSTIA learning a single vowel harmony with blocking. . .	151

4.5	Results of OSTIA learning several vowel harmonies without blocking.	152
4.6	Results of OSTIA learning several vowel harmonies with blocking.	154
4.7	Results of OSTIA learning vowel and consonant harmonies without blocking.	156
4.8	Results of OSTIA learning vowel and consonant harmonies with blocking.	158
4.9	Results of OSTIA learning UTP.	158
4.10	Results of OSTIA learning a “simple” first-last harmony.	160
4.11	Results of OSTIA learning a “complex” first-last harmony.	162
4.12	Results of the learning experiments using OSTIA.	163
4.13	Predicted results (marked as *) of the learning experiments using SOSFIA, OSLFIA and ISLFLA learning algorithms.	169
5.1	Learning results that were experimentally obtained in this dissertation. Black cells indicate that the experiments were not conducted due to the reasons discussed in 5.1.2.	176

Chapter 1

Introduction

The availability of tools greatly impacts the future of ideas. Charles Babbage was the first to conceptualize the design of a computer in 1837, however, he could not implement it because the required funding and technologies needed for the production of his *Analytical Engine* were not yet available. Only in 1941, technological progress allowed for the first general-purpose computer named Z3 to be assembled. Even though the field of genetics originated in the early 18th century, it was the development of the photo techniques that allowed Rosalind Franklin to take a picture of the crystallized fibers in 1952 that ultimately led to the discovery of DNA sequencing. Frequently, the development of tools for a certain scientific area helps to make progress in that area.

In my dissertation, I implement and experiment with some of the tools available for the formal classes of *subregular languages and mappings* that recently proved themselves to be extremely useful for modeling natural language dependencies. Namely, I discuss the results of the automatic extraction of subregular grammars from data exhibiting various linguistic patterns, such as word-final devoicing, harmony systems of different types, and others. The code behind the inference algorithms is available as a part of my package *SigmaPie*  (Aksënova, 2020b). This package is open source, available via pip and

implemented in Python 3. *SigmaPie* allows linguists and formal language theorists to assess possible applications of those tools and ideas in the areas of typology, cognitive linguistics, and natural language processing. This dissertation implements subregular learning algorithms and demonstrates their performance on various linguistic datasets. I focus exclusively on string representations, as it is a necessary step before extending such type of research to trees or graphs.

1.1 Subregular grammars

Formal tools help to generalize natural language patterns and study them independently of linguistic theories, therefore allowing researchers to focus on one of the core questions of linguistics: *what is the complexity of natural language?* Although this question is not yet answered, we already know to some extent the complexity of the restrictions that phonotactics and morphotactics impose on the surface forms of their objects. We also came closer to understanding what types of changes are involved in phonological and morphological processes. Recently, researchers started to target areas of linguistics such as syntax and semantics, and we already see interesting updates from that research front as well.

However, when machine learning techniques are applied to natural language data, all the knowledge we have about the types of language dependencies is usually ignored. It makes for a weird situation when people who study the core principles of natural language and people who model natural languages do not communicate. And even if they do, it is unclear how to incorporate this knowledge in modern machine learning systems. Neural networks that are widely employed nowadays in the field of natural language processing learn patterns in an uninterpretable fashion therefore not giving a way for the linguists to look inside those networks and understand *how* and *what exactly* was learned. On the contrary, *subregular* learning algorithms (further referred to as “subregular

learners”) are fully transparent and interpretable.

The *SigmaPie* package implements subregular learners that efficiently extract grammars after observing a finite number of well-formed strings of the target language. This package also implements sample generators, scanners, and some other tools. The generation of a data sample of the required complexity is needed during the design of artificial learning experiments. Scanners and re-writers verify the well-formedness of strings regarding some grammar or modify the input according to a specified set of rules. Additionally, the toolkit provides functions such as changing the polarity of the grammar or removing uninformative elements from the grammar.

Apart from the interpretability of the grammar per se, subregular learning algorithms guarantee the interpretability of the way the grammar was discovered. In other words, it is always *possible to look inside the algorithm*. Observing the behavior of the algorithm and studying its properties is necessary for understanding which configurations in the training data make it possible to find the pattern. If we are dealing with natural language data, transparent learning algorithms can help to explore the way humans learn languages. Interestingly, the discussed classes of subregular languages are learnable only from positive data.

In the field of formal languages, theoretical achievements are not always followed by their practical applications. As a result, a frequent situation arises where a learning algorithm is proposed in the literature but is not implemented. Although it is important to prove theorems about the convergence of such algorithms, it is also important to subject them to empirical testing. As of now, not a lot of such algorithms are implemented, and even fewer of them are employed in practice. Sometimes, as Gildea and Jurafsky (1996) show in their paper, the grammatical inference algorithms need to be modified and *linguistically “biased”* to work with raw language data. The last few decades brought us a lot of new knowledge about the complexity of human language patterns, and the majority of

this knowledge is still waiting to be incorporated into these algorithms.

1.2 Linguistic motivation behind the subregular approach

What is the minimum generative capacity of the grammar that is capable of encoding human language-like patterns? In other words, what types of dependencies must that grammar take into account? Answering these questions might give us a powerful insight into human cognition. The first step must be uncovering what *types* of patterns do human languages exhibit.

To describe and generalize phenomena observed in natural languages, we need to build their computational or mathematical models. Formal languages provide a way to do it. Their object can be any structured object formed from a finite collection of discrete elements, where those objects can be strings, trees, or graphs. In this thesis, however, I will only focus on string representations.

Subregular modeling provides two perspectives: modeling well-formedness conditions as languages, and modeling transformations as mappings. A **well-formedness condition** can be encoded as a language, or a potentially infinite collection of strings satisfying that condition. For example, in Russian, voiced obstruents become voiceless at the end of the word. It results in words such as *lo[b]* being excluded from the collection of well-formed Russian strings, whereas their voiceless counterparts such as *lo[p]* ‘forehead’ are grammatical. When augmented with a word-final marker \times , the corresponding grammar rules out all cases when a voiced obstruent is followed by that marker: $b\times$, $g\times$, $d\times$, etc. In contrast, a **transformation** can be formalized as a collection of pairs of strings, where those strings represent the states “before” and “after” the rule application. In other words, those pairs demonstrate the underlying representations and the corresponding surface forms. From this perspective, Russian word-final devoicing

can be viewed as a collection of pairs, where the final obstruent of the first string can be either voiced or voiceless, but it is always voiceless in the second one: $(lo[b], lo[p])$ ‘forehead’, $(lu[g], lu[k])$ ‘meadow’, $(lu[k], lu[k])$ ‘onion’, etc. The corresponding grammar then looks at every symbol of the underlying representation and rewrites it as is, unless that symbol is the word-final voiced obstruent: then it is substituted by its voiceless counterpart. In such a way, subregular models can capture well-formedness conditions, and encode the mapping of underlying representations to the corresponding surface forms.

In the domain of string languages, *regular languages and mappings* provide a reasonable upper bound for phonology and morphology (Johnson, 1972; Koskenniemi, 1983; Kaplan and Kay, 1994; Beesley and Karttunen, 2003). The class of regular languages, however, can be further subdivided into a nested hierarchy of weaker *subregular* languages. Closer research of phonological and morphological patterns shows that in fact, these patterns do not require the whole power of regular languages. Several subregular classes express well-formedness conditions imposed by phonotactics and morphotactics (Heinz et al., 2011; Aksënova et al., 2016; Heinz, 2018). Subregular – namely, *subsequential* mappings describe a multitude of morphological and phonological processes (Chandlee, 2017; Chandlee and Heinz, 2018). Subregular grammars found their applications even in the areas of syntax and semantics. (De Santo et al., 2017; Graf and Shafiei, 2019; Graf, 2019). In this thesis, I focus on modeling phonological dependencies of different kinds.

Nowadays, researchers work on many aspects of subregular languages. There has been significant progress in the understanding of their underlying mathematical structures (Fu et al., 2011; Heinz and Rogers, 2013). Multiple papers show how different linguistic phenomena can be accounted for in terms of subregular models (Heinz et al., 2011; Heinz and Lai, 2013; Chandlee, 2014; Aksënova et al., 2016; Dolatian and Heinz, 2018; Graf, 2019; Karakaş, 2020). The

approach was extended to trees and now can express such complicated dependencies as c-command or case assignment as well (Graf and Shafiei, 2019; Vu et al., 2019). The works cited above are all very recent. To help accelerate this currently growing direction of research, I implemented a package that provides the subregular functionality and explored *practical* capabilities of those algorithms.

1.3 Structure of the dissertation

Subregular learners capture different types of natural language dependencies. In my dissertation, I aimed to explore how well the theorems about the correctness of those learners carry over to real-world performance. To do so, I designed multiple artificial learning experiments and scored the subregular learners on datasets exhibiting patterns such as local assimilations, multiple long-distant harmonies of different types, some typologically unattested patterns, and others. The datasets ranged from artificial automatically generated samples to real-language datasets such as German, Finnish, and Turkish wordlists. While the artificially generated datasets explored if a pattern was learnable in general, the raw data showed what issues the learners have when faced with the raw natural language data. Every target pattern was approached from two perspectives: as a well-formedness condition on the surface forms, and as a transformational rule changing values of some elements.

While chapters 3 and 4 explore subregular modeling from a practical point of view, **Chapter 2. Background** gives a theoretical perspective on such languages and mappings. I demonstrate the modeling capacities of subregular grammars and transformations by capturing attested natural language patterns such as word-final devoicing, unbounded tone plateauing, and several different types of harmonies. Namely, the reviewed formal classes are strictly local, strictly

piecewise, tier-based strictly local, and multi-tier strictly local languages; and subsequential transformations. Additionally, in that chapter, I also discuss the problem of inferring grammars from data, and list the useful properties shared by the subregular learners.

In **Chapter 3. Learning languages**, I target modeling well-formedness conditions. To do so, I employ four subregular language classes that can theoretically express the target generalizations such as attested and unattested harmony systems, and other local and long-distant generalizations. Given this perspective, a learning dataset is a list of well-formed strings: it does not include words that violate the target well-formedness dependency. So, for example, for a target pattern of vowel harmony, the training dataset is a sample of harmonic words. Some of those subregular classes model local dependencies, while others encode long-distance dependencies. I also discuss the architectures of the learners originally introduced in (Heinz, 2010b; Jardine and McMullin, 2017; McMullin et al., 2019).

Chapter 4. Learning mappings is concerned with modeling transformations changing the underlying representations into the corresponding surface forms. According to Chandlee (2014), many of phonological and morphological processes belong to the class of subsequential mapping. Thus, I give an overview of a subsequential learner (Oncina et al., 1993; de la Higuera, 2010), and use it to extract generalizations from the datasets exhibiting linguistic dependencies similar to the ones listed above. In this case, patterns are represented as pairs of strings. So, for example, if a pair demonstrates a vowel harmony, then the first string shows the underlying or underspecified representation, while the vowels in the second word are fully specified and harmonic. Additionally, in that chapter, I discuss other algorithms that learn mappings and can be employed for similar tasks in the future.

Finally, **Chapter 5. Conclusion** summarizes the obtained results and proposes

directions for future research. Figure 1.1 gives an overview of the flow of this thesis, starting from the theory of modeling well-formedness conditions and transformations, and then followed by a practical part discussing the experimental setup and the executed learning experiments.

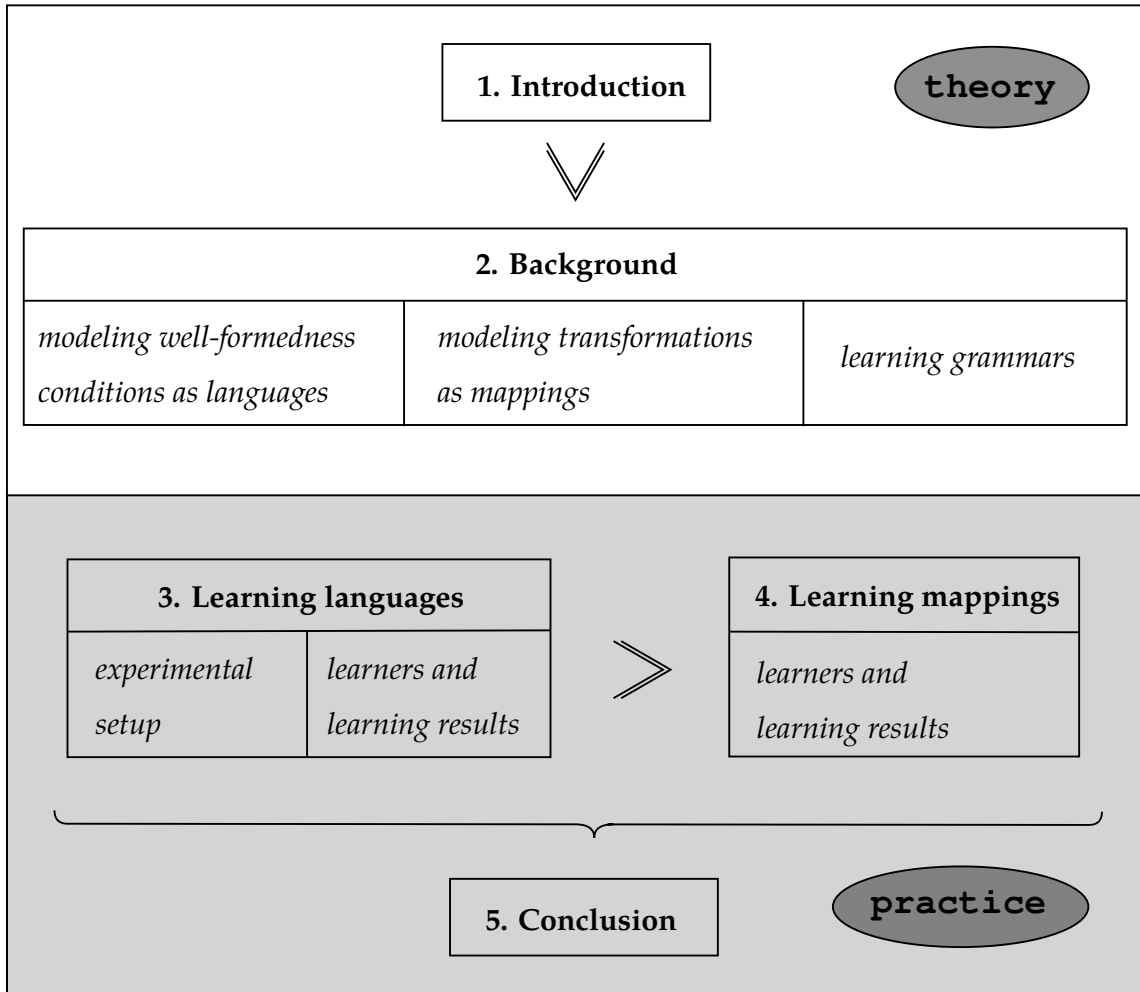


Figure 1.1: Flow of chapters of this dissertation: Introduction, Background, Learning languages, Learning mappings, and Conclusion.

Chapter 2

Background

Linguistic rules capture two types of generalizations: **well-formedness conditions**, i.e. the requirements for a word to be well-formed, and **transformations**, i.e. the rules of re-computing the given underlying representation into the corresponding surface form. The former ones restrict the word's form itself, such as “two vowels should never be adjacent to each other”, while the latter ones describe the change, such as “insert [j] in-between two adjacent vowels”. For example, transformations map the word *dlinnosheee* ‘long-necked’ (Russian) into its pronunciation *dlinnosh[ejeje]*, and the well-formedness conditions ensure that the pronunciation *dlinnosh[eee]* is not allowed since it contains two vowels adjacent to each other.

Grammars describe how to build well-formed words from the elements of the *alphabet*. A *language* of the grammar is a potentially infinite collection of all well-formed strings of that grammar. Thus in a formal sense, it simply refers to a collection of words, or *strings*. Transformations are functions from the *input language*, i.e. a collection of “underlying representations”, onto the *output language*, or a collection of “surface forms”.

In this chapter, I discuss *subregular grammars* and *subsequential functions* as they seem to be a good fit for natural language dependencies (Heinz, 2011; Heinz et al.,

2011; Gainor et al., 2012; Heinz and Lai, 2013; Aksënova et al., 2016; Graf, 2017a; Chandlee and Heinz, 2018, i.a.). In the two following chapters, I show the results of the **automatic extraction** of subregular grammars and subsequential functions given the learning framework defined in Section 2.3.

2.1 Modeling well-formedness conditions

To model well-formedness conditions means to find a way to discriminate between well-formed and ill-formed words of a language. In other words, it implies finding a *grammar* that only builds well-formed strings, and that can recognize which strings are ill-formed. For example, given the alphabet of vowels and consonants, a grammar can prohibit vowel hiatus by penalizing adjacent vowels.

Subregular grammars provide an interpretable and succinct way to encode such rules. Interestingly, the subregular nature of linguistic generalizations allows us to explain the absence of some theoretically possible yet typologically unattested patterns (Gainor et al., 2012). Also, this approach gives insights into human cognition since there is evidence that only some subregular language classes are learnable (Lai, 2015).

2.1.1 Regular nature of natural language patterns

Consider a pattern of Russian compounding, where a morpheme $-o$ ¹ is located in-between compounding stems. For example, the stems *vod(-a)*² ‘water’ and *voz* ‘carrier’ can be combined to obtain a complex word *vod-o-voz* ‘water carrier’. If the compound is composed of multiple stems, the marker is added in-between every one of them: *vod-o-voz-o-voz* ‘carrier of water carriers’.

¹This marker is also sometimes realized as *-e*.

²*-a* is a suffix marking nominative case, singular form for some nominal classes.

This pattern can be viewed as a language of well-formed sequences of stems and compounding affixes. Strings such as *stem*, *stem-o-stem*, *stem-o-stem-o-stem* belong to the target language, but *stem-stem* and *stem-o* do not. This can be rephrased a rule “a well-formed form cannot start or end with a compounding marker, and within a word, two markers or two stems should not be adjacent to each other”. Generalizations like this can be conveniently expressed as **finite state automata**.

A **finite state automaton** (FSA) is a type of an abstract machine that is defined by a finite list of states and the transitions between those states (Lawson, 2003). In the case of string-based automata, these transitions are annotated with characters. An automaton reads a string of characters (the *input string*), and every new character changes the current state. Some of the states are *initial*, meaning that the first character of the input string can be read from those states. *Final*, or *accepting* states are the ones that indicate that the string is accepted. The input string is accepted by an automaton when the first character of that string can be read from the initial state, and this string is a path from the initial state to the final one.

Consider the automaton in Figure 2.1. The numbered circles represent states, and the arrows are the transitions between those states. States are usually referred to as q , therefore the states of that machine are q_0 , q_1 and q_2 . The initial state is represented with an incoming “start” arc. The state q_1 is marked with a double circle, meaning that it is final.

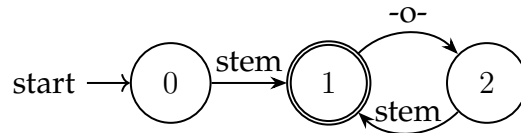


Figure 2.1: FSA for Russian compounding.

A language of an FSA is a potentially infinite set of strings, every member of which can be recognized by that automaton. For example, in Figure 2.1, the only possible transition from the initial state q_0 reads a stem and moves the machine to

q_1 . The state q_1 is the accepting state, and any string that brings the automaton to the accepting state is well-formed with respect to the rules encoded in the automaton. A single stem is therefore considered well-formed. A compounding marker *-o-* moves the machine from the state q_1 to q_2 . But q_2 is not final, so strings cannot be accepted if they end up in that state: the compounding marker cannot be the final element of the word. The machine then necessarily returns to the state q_1 , therefore accepting *stem-o-stem*. If more markers and stems follow, it takes the loop $q_1 \rightarrow q_2 \rightarrow q_1$ again. The complexity of the language recognized by an FSA is not more than *regular*. Formally, regular languages are ones of the least complex formal languages; they are discussed further in this section.

Formal languages are potentially infinite collections of strings produced according to the rules of some *grammar*. Different language classes can be recognized by different automata. These automata can also be referred to as *abstract machines*, a more general name for theoretically possible computers encoding the rules of those languages. These machines, and therefore languages corresponding to them, can be ordered with respect to the complexity of the dependencies that they encode. The first version of such hierarchy was introduced in Chomsky (1956) and is therefore known as *the Chomsky hierarchy*. Nowadays, we usually use an extended version of the hierarchy that includes *mildly context-sensitive* and *finite* language classes (Jäger and Rogers, 2012), see Figure 2.2.

This hierarchy represents nested classes of formal languages aligned with respect to their expressive complexity. On the very top of the hierarchy, there are **recursively enumerable** languages. Those are the languages that can be physically computed, i.e. realized by a computer in the universe³ (Chomsky, 1956). For example, a language a^n , where n is a prime number, is recursively

³The current definition is based on the physical Church-Turing thesis (Church, 1936; Turing, 1937b,a).

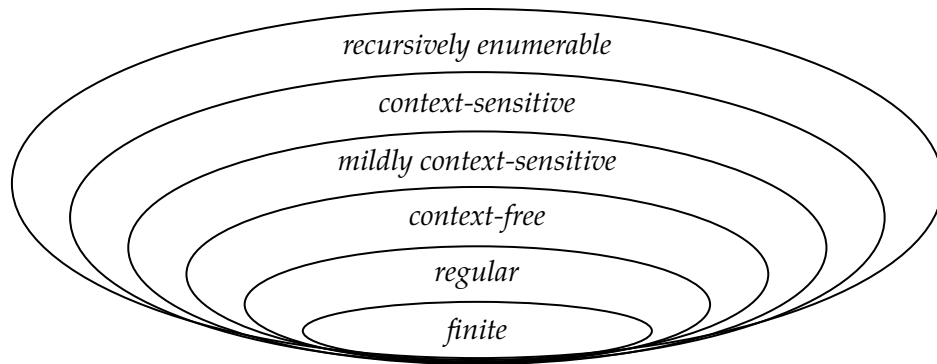


Figure 2.2: The extended Chomsky hierarchy from (Jäger and Rogers, 2012).

enumerable. Below, there are **context-sensitive** languages that recognize non-linear⁴ patterns such as a^{2n} , where n is greater than 0. There are subclasses of context-sensitive languages which are a better fit for natural language syntax, such as **mildly context-sensitive** languages. They are a good fit for syntactic dependencies as they handle cross-serial dependencies such as some cases of copying (Joshi, 1985; Shieber, 1985; Kallmeyer, 2010). The machine corresponding to **context-free** languages uses a stack of a potentially infinite size: in such a way, it recognizes patterns such as $a^n b^n$, “have as many a as b ” (Hopcroft et al., 2006). **Regular** languages are limited to the dependencies that can be recognized by an FSA (Hopcroft et al., 2006). It is commonly assumed that morphology and phonology are regular (Johnson, 1972; Kaplan and Kay, 1994; Beesley and Karttunen, 2003; Roark and Sproat, 2007), and I will come back to this topic in the following paragraphs. Finally, at the very bottom of the hierarchy, one can see a class of **finite** languages that refer to a finite number of strings. Classes that are more complex than mildly context-sensitive dependencies are rarely discussed in connection with natural languages: they are too powerful.

Finite-state models correspond to regular languages, and were introduced in

⁴The non-linearity refers to the growth of the number of a : every following number is *much* larger than the previous.

1940s by McCulloch and Pitts. Chomsky (1956), however, theorized that this type of modeling does not seem to be suitable for natural languages. Despite that, there was a significant number of applications of finite-state machines to language-related tasks, such as text search (Thompson, 1968), machine translations (Oncina et al., 1994; Knight and Al-Onaizan, 1998; Bangalore and Riccardi, 2002), speech recognition (Caseiro, 2003; Mohri et al., 2002, 2008), semantic parsing (Jones et al., 2011, 2012), and others. The restrictiveness of finite-state models is frequently used to balance the robustness of neural networks; see, for example, a FST-based pronunciation learning model by Bruguier et al. (2017). At the same time, linguists and computer scientists started to employ regular languages and finite-state models as a tool to research the complexities of patterns in human languages, see Hulden (2014) discussing the main milestones.

The regular nature of phonology was examined several decades ago by (Johnson, 1972; Kaplan and Kay, 1981, 1994). Importantly, Kaplan and Kay (1994) show that all SPE-style transformational rules (Chomsky and Halle, 1968), can be represented as finite-state machines. Since all attested phonological patterns can be modeled as SPE-style rules, and since SPE-style rules are regular, the complexity of regular languages is a good upper bound for phonological dependencies. In the same paper, they also presented a set of modeling tools and used them to capture patterns such as nasal assimilation and epenthesis.

Koskenniemi (1983), and later Beesley and Karttunen (2003) show that finite-state machinery is sufficient for encoding morphological dependencies as well. Even the non-concatenative morphology can be modeled in such a way (Kay, 1987; Beesley, 1996; Kiraz, 1996). For more examples of applications of finite-state methods in linguistics, see (Gildea and Jurafsky, 1996; Roche and Schabes, 1997; Hetherington, 2001; Jurafsky and Martin, 2009). Although regular languages are a good fit for linguistic patterns, research shows that the full power

of regular languages is not necessary, and *subregular* languages that are discussed further provide a tighter fit for phonology and morphology.

2.1.2 Subregular languages and their linguistic importance

The class of regular languages can be subdivided into a nested hierarchy of *subregular* classes – *subregular hierarchy*. “Parent” classes are more powerful than their “children” classes, and therefore properly include them. The “sibling” relation implies that the classes are not known to subsume each other. Some of the classes of the subregular hierarchy are presented below in Figure 2.3.

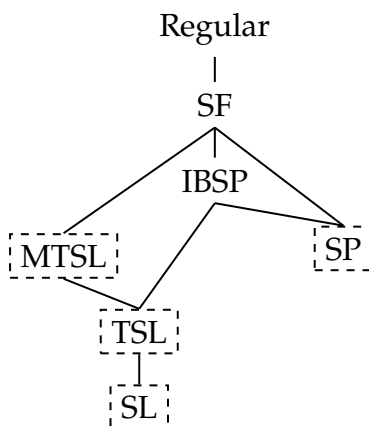


Figure 2.3: Some of the classes of the subregular hierarchy; the subregular classes discussed further in this chapter and in Chapter 3 are boxed.

Figure 2.3 shows some of the subregular classes, namely the ones that are crucially important for modeling linguistic dependencies, and therefore extensively used in this dissertation. Namely, those are strictly local (SL), strictly piecewise (SP), tier-based strictly local (TSL) and multi-tier strictly local (MTSL) languages. The crucial difference between these languages lays in the types of dependencies they can capture, see Table 2.1. The combined functionality of those classes covers local dependencies and long-distance dependencies with or

without a blocker.

Language	Dependencies it can handle
<i>SL</i>	only local dependencies
<i>SP</i>	only multiple long-distance dependencies without blocking
<i>TSL</i>	long-distance dependencies with blocking
<i>MTSL</i>	multiple long-distance dependencies with blocking

Table 2.1: Types of dependencies captured by some of the subregular classes.

Subregular languages are encoded by *subregular grammars*. These subregular grammars operate by blocking some types of substrings or subsequences in well-formed strings of their languages. A **substring** is a consecutive part of the string. For example, *ab*, *aba*, and *abacd* are substrings of the string *abacd*, whereas *aa* and *bcd* are not, because these symbols are not adjacent in the string *abacd*. **Subsequences** can be seen as a non-consecutive counterpart of substrings. A string *u* is a subsequence of *w* if all elements of *u* can be found in *w*, and the order of those elements is preserved. Continuing the previous example, both *aa* and *bcd* are indeed subsequences of *abacd*. Importantly, the elements of substrings and subsequences cannot violate the order in which the elements appeared in the original string: *ca* is neither a substring nor a subsequence of the string *abacd*. Subregular grammars are always defined for some particular locality, so a 2-local grammar operates with substrings or subsequences of the length 2. Formally, substrings and subsequences are defined in Definitions 2.1.1 and 2.1.4, and in (Elzinga et al., 2008; Rogers et al., 2010; Fu et al., 2011, a.o.).

While *positive grammars* list all allowed substructures of their languages, the *negative* ones list the substructures that must not be encountered in well-formed strings of their languages. Moreover, these grammars are equivalent, i.e. for every negative grammar, it is possible to construct a positive grammar that generates the same language, and vice versa.

The above mentioned subregular grammars are negative, so they prohibit certain substructures in well-formed strings of their languages. **Strictly local** (SL) grammars filter strings that violate some local dependency in the string, i.e. contain ill-formed *substrings* (Heinz, 2010b). For example, a language *ab, abab, ababab, etc.* contains any possible string of *a* and *b* that does not contain the substrings *aa* and *bb*. **Tier-based strictly local** (TSL) grammars project a potentially smaller string from the input string by using a tier alphabet. These grammars evaluate the relatively local dependencies among the elements of the *tier alphabet*, whereas all other symbols are “transparent” for the grammar (Heinz et al., 2011). For example, if the tier contains *a* and *b* and the string is *bccacbc*, the *tier image* of that string is *bab*. TSL constraints such as *ba* or *ab* would rule out that string. **Multi-tier strictly local** (MTSL) grammars can have more than just a single tier, and, therefore, multiple tier images are evaluated with respect to multiple local grammars (De Santo and Graf, 2019). **Strictly piecewise** (SP) grammars restrict certain *subsequences* in well-formed strings of their languages (Rogers et al., 2010; Heinz, 2010a). In such a way, SL, SP, TSL and MTSL grammars model a wide range of local and long-distant processes (Heinz, 2011; Heinz et al., 2011; Heinz and Lai, 2013; Aksënova et al., 2016; Chandlee and Heinz, 2018). There are other subregular classes not listed here such as star free, interval-based strictly piecewise, input-output tier-based strictly local, piecewise testable, etc., but they are out of scope of this dissertation (Lawson, 2003; Graf, 2018a). Additionally, I will only discuss grammars working with strings, but this approach is currently extended to other representations as well (Chandlee et al., 2019; Chandlee and Jardine, 2019).

A *strong subregular hypothesis* suggests that a regular bound is too powerful for natural language phonology and phonotactics and that subregular languages are a better fit (Heinz, 2010a). In line with phonotactics, Aksënova et al. (2016) shows that subregular languages are a good fit for morphotactic dependencies. The

well-formedness conditions imposed on languages of generalized and monomorphemic quantifiers are also subregular. There are likewise applications of subregular grammars to syntax (Graf, 2017c; De Santo et al., 2017; Vu et al., 2019).

Further in this section, I focus on SL, SP, TSL and MTSL languages and grammars, and provide linguistically-motivated examples of those. Additionally, I also define those classes mathematically and describe the corresponding classes of finite-state automata. Later, in chapter 3, I will model those and some other dependencies, and show how their subregular grammars can be learned from real data. Table 2.2 summarizes patterns that are discussed further and subregular classes to which they belong.

SL	SP	TSL	MTSL
<i>word-final devoicing and obstruent voicing assimilation</i>			
✓	×	✓	✓
<i>unbounded tone plateauing</i>			
×	✓	×	×
<i>sibilant harmony in voicing and anteriority, no blockers</i>			
×	✓	✓	✓
<i>vowel harmony in ATR, nasalized vowels are blockers</i>			
×	×	✓	✓
<i>vowel harmony in ATR and rounding, some vowels are blockers for rounding</i>			
×	×	✓	✓
<i>sibilant harmony in voicing and anteriority, voiceless obstruents are blockers for voicing</i>			
×	×	×	✓

Table 2.2: Subregular patterns attested in natural languages and discussed in sections 2.1.3-6.

2.1.3 Local restrictions as SL languages

The subregular class of strictly local (SL) languages captures local dependencies, and many restrictions in phonology have a purely local nature (Rogers and Pullum, 2011). Indeed, most of the patterns of phonological assimilation affect adjacent segments. In what follows, I exemplify SL languages using several purely local processes, and then formally introduce them using the notion of k -factor and the property of suffix substitution closure (Rogers et al., 2013).

Intuitive definition

In what follows, the SL languages are demonstrated through two local processes happening in Russian: one of them prohibits voiced obstruents in the word-final position, and another one enforces adjacent obstruents to agree in voicing. Additionally, I show the interaction between these two patterns.

Russian obstruent assimilation and word-final devoicing The examples (1-2) below show that the consonant of the preposition *iz* ‘from’ agrees in voicing with the obstruent of the following word.

- (1) i[z B]erlina ‘from Berlin’
- (2) i[s P]ragi ‘from Prague’

Additionally, in Russian, as well as in other languages such as German, there is a process of word-final devoicing that prohibits the appearance of a voiced obstruent in a word-final position (Brockhaus, 1995; Padgett, 2002). For example, *lug* ‘field’ is realized as *lu[k]*. In this case, the same cluster of obstruents might appear voiceless word-finally, and voiced in other positions of the word, see the pairs of examples in (3-4) and (5-6).

- (3) mo[sk] ‘brain’ ~ (4) mo[zg]i ‘brains’
- (5) dro[st] ‘thrush’ ~ (6) dro[zd]y ‘thrushes’

These two generalizations can be captured in a *strictly local* way, namely, by prohibiting illicit substrings. Assume that the inventory of obstruents is $\{z, s, b, p, g, k\}$, where z, b , and g are voiced, and s, p and k are voiceless. It is never possible to see two (or more) disagreeing obstruents adjacent to each other, i.e. the target SL grammar needs to prohibit zs, zp, zk, sz, sb, sg , etc.

To target voiced obstruents at the end of the word, the grammar needs to be able to differentiate between the word-final and other positions. For this reason, strings are usually annotated with the markers \times and \times denoting the beginning and the end of the string (Rogers and Pullum, 2011). Then, the SL grammar capturing word-final devoicing needs to rule out $z\times, g\times$, and $b\times$.

$$\begin{array}{cc} \times m o s k \times & \times m o z \boxed{g} \times \\ \times m o \boxed{z} \boxed{k} \times & \times m o \boxed{s} \boxed{g} \times \end{array}$$

Figure 2.4: Evaluation of strings *mozg*, *mozk*, *mosg* and *mosk* by an SL grammar capturing obstruent cluster assimilation and word-final devoicing.

Figure 2.4 shows how the SL grammar outlined above evaluates the pronunciations *mozg*, *mozk*, *mosg* and *mosk*. The string *mozg* has its obstruents agree in voicing, but the final obstruent is voiced, and therefore ruled out by the restriction $g\times$. The final obstruent in *mozk* is voiceless, but now the cluster disagrees and therefore ruled out by zk . In *mosg*, both violations are present. Finally, the form *mosk* contains no violations, and indeed, this is the correct pronunciation of the corresponding Russian word.

The illicit substrings such as $g\times$ and zk are the *restrictions* defined by the grammar. A set of restrictions R of a negative grammar lists all substrings that *cannot* be found in well-formed strings of the language. An *alphabet* of the language, usually denoted as Σ , includes the list of symbols the language uses. In this case, Σ includes all Russian phonemes. Finally, every grammar defines its

locality, namely, the size of the longest string prohibited by that grammar; it is usually referred to as k (McNaughton and Papert, 1971; Rogers and Pullum, 2011). In the case of the SL grammar capturing Russian well-formedness conditions, all the prohibited strings are of length 2 ($k = 2$); such substrings are also called *bigrams* or *factors*. The Russian SL grammar is then strictly 2-local, or 2-SL. These three components – the alphabet Σ , the set of restriction R , and the locality window k – define SL grammars, see Grammar 2.5.

SL grammar Russian obstruent voicing assimilation and word-final devoicing

$$\begin{aligned}\Sigma &= \{a, b, v, g, d \dots z, s, p, k \dots \varepsilon, {}^j u, {}^j a\} \\ R &= \langle zs, zp, zk, sz, sb, sg \dots z\bowtie, g\bowtie, b\bowtie, d\bowtie \rangle \\ k &= 2\end{aligned}$$

Grammar 2.3: 2-SL grammar for Russian obstruent voicing assimilation and word-final devoicing.

Indeed, SL models are useful in modeling the well-formedness conditions arising from word-final devoicing, intervocalic voicing, consonant cluster assimilation, and other local processes. However, SL models cannot model long-distance dependencies.

Tuareg sibilant harmony Long-distance dependency affects segments that can be located far from each other. For instance, in Tuareg (Berber), sibilants regressively agree in voicing and anteriority (Hansson, 2010b). In the examples below, a causative prefix agrees with the sibilant in the root (8-11) but is realized as *s-* if no other sibilant is present (7). This agreement is long-distant, and therefore it can happen across an arbitrary number of intervening elements.

- (7) s-əlməd 'CAUS-learn'
- (8) s-əq:usət 'CAUS-inherit'
- (9) z-əntəz 'CAUS-extract'
- (10) ʃ-əm:əʃən 'CAUS-be.overwhelmed'
- (11) ʒ-ək:uʒət 'CAUS-saw'

SL grammars capture only local generalizations, but, for example, in (9), there are 4 elements separating the agreeing sibilants. It would imply that the required locality of the SL grammar is at least 6 to accommodate the two sibilants and everything in-between them. However, this would not be enough for other cases since there is no upper bound on the number of the intervening segments in-between two agreeing sibilants. As a result, the power of SL grammars is not enough to capture patterns such as Tuareg sibilant harmony.

Formal definition

SL grammars define languages by listing substrings that cannot appear in well-formed words of those languages. These substrings are often referred to as k -factors, in order to be distinguished from the more NLP-oriented use of the term n -gram, that frequently implies the use of probabilistic models (Rogers and Pullum, 2011; Rogers et al., 2013). Further in this section, I follow De Santo and Graf (2019) in their algebraic definition of this class.

While Σ , as previously in this section, denotes the alphabet, Σ^k is a k -long word that uses symbols of that alphabet. Σ^* generalizes Σ^k , it employs the *Kleene star* (Kleene, 1956) to define a word of any length. A length of the string w is denoted as $|w|$.

Definition 2.1.1 (k -factors)

A string u is a k -factor of a string w iff $\exists x, y \in \Sigma^$ such that $w = xuy$ and $|u| = k$. The function F_k maps words to the set of k -factors within them:*

$$F_k(w) = \{u : u \text{ is a } k\text{-factor of } w \text{ if } |w| \geq k, \text{ else } u = w\}$$

For example, the 2-factors of the word abc are $\{ab, bc\}$. Strictly k -local grammars list the k -factors that cannot be used in the well-formed strings of their languages, i.e. they can be viewed as collections of illicit k -factors.

Definition 2.1.2 (SL languages and grammars)

A language L is strictly k -local (SL_k) iff there exists a finite set $S \subseteq F_k(\bowtie^{k-1}\Sigma^*\bowtie^{k-1})$ such that

$$L = \{w \in \Sigma^* : F_k(\bowtie^{k-1}w\bowtie^{k-1}) \cap S = \emptyset\}.$$

We call S a strictly k -local grammar, and use $L(S)$ to indicate the language recognized by S . A language L is strictly local iff it is SL_k for some $k \in \mathbb{N}$.

Consider a language described by a regular expression $(ab)^*$. Its language includes strings such as ϵ , ab , $abab$, $ababab$, etc. A negative 2-SL grammar that describes this language is $S = \{\bowtie b, a\bowtie, aa, bb\}$. Indeed, the well-formed strings of that language cannot start with b , end with a , and have two a or two b adjacent to each other. Importantly, a language is strictly k -local if it satisfies k -local *suffix substitution closure* (Rogers and Pullum, 2011).

Definition 2.1.3 (Suffix substitution closure)

For any $k \geq 1$, a language L satisfies k -local suffix substitution closure iff for all strings u_1, v_1, u_2, v_2 , for any string x of length $k - 1$ if both $u_1 \cdot x \cdot v_1 \in L$ and $u_2 \cdot x \cdot v_2 \in L$, then $u_1 \cdot x \cdot v_2 \in L$.

For instance, a language $(ab)^*$ is 2-SL, and it satisfies the suffix substitution closure. Both strings ab and $ababab$ contain the 1-local substring a , and it correctly predicts that the string $abab$ is also in the language. However, a language a^*ba^* is not SL. It is not 2-SL since the closure of the strings aba and baa contains $baba$, and it is not in the language. It is not 3-SL, because the closure of $aaba$ and $baaa$ contains the illicit form $baaba$; and so on. The language a^*ba^* is not closed under the suffix substitution, and therefore it is not SL.

Apart from the algebraic perspective, strictly local languages can be characterized in automata-theoretic terms. Rogers and Pullum (2011) describe SL languages as those that can be recognized by FSAs scanning a k -symbol window across the input string, and failing on strings that contain factors prohibited by the corresponding grammar.

2.1.4 Long distance restrictions as SP languages

While SL grammars can only capture local dependencies, strictly piecewise (SP) grammars generalize exclusively long-distance patterns: SL grammars prohibit sequences of adjacent segments within words, while SP grammars do not have the requirement of adjacency. An SP restriction prohibits a certain *order* of elements (Rogers et al., 2010; Fu et al., 2011). While an SL restriction VV prohibits two vowels adjacent to each other thus avoiding hiatus, the same SP restriction means that nowhere in the string can there be a vowel followed by another vowel. The language of such an SP grammar would only include words with no more than a single V .

Intuitive definition

In this section, I demonstrate how SP grammars can capture patterns of sibilant harmony and unbounded tone plateauing. SP grammars encode restrictions on the order of elements, and thus cannot differentiate between disharmonic stems and grammatical words exhibiting a blocking effect.

Tuareg sibilant harmony Coming back to the pattern of Tuareg sibilant harmony exemplified before in (7-11), it can be modeled with an SP grammar by prohibiting subsequences of disagreeing sibilants. The bigrams sz and $\text{ʃ}z$ are prohibited because their elements disagree in voicing, $\text{ʃ}s$ and $z\text{ʃ}$ disagree in anteriority, etc. In total, this grammar contains 12 restrictions R that are listed in

Grammar 2.5.

SP grammar Tuareg sibilant harmony in voicing and anteriority

$$\Sigma = \{s, \text{ʒ}, z, \text{ʃ}, \text{ə}, d, l, m, t \dots\}$$

$$R = \langle sz, s\text{ʃ}, s\text{ʒ}, zs, \text{ʃ}s, \text{ʒ}s, z\text{ʃ}, z\text{ʒ}, \text{ʃ}s, z\text{ʃ}, z\text{ʒ}, \text{ʃ}z, \text{ʃ}z, \text{ʃ}\text{ʒ}, \text{ʃ}z \rangle$$

$$k = 2$$

Grammar 2.4: 2-SP grammar for Tuareg sibilant harmony in voicing and anteriority.

Such an SP grammar has an alphabet that includes all Tuareg phonemes, and its list of restrictions includes all pairs of sibilants disagreeing in voicing or anteriority. The locality of such grammar is 2. Figure 2.5 shows that there are no violations in the word *zəntəz*: indeed, no substructure of that string is prohibited. However, the word *zəntəʒ* is ruled out because the subsequence *zʒ* is ill-formed.

z ə n t ə z z ə n t ə ʒ



Figure 2.5: Evaluation of strings *zəntəz* and *zəntəʒ* by an SP grammar capturing sibilant harmony in voicing and anteriority.

Imdlawn Tashlhiyt sibilant harmony Now, consider a language closely related to Tuareg, namely, Imdlawn Tashlhiyt (Berber), in which affixal sibilants also regressively harmonize with the stem in voicing and anteriority (Hansson, 2010b; McMullin, 2016). The difference is that in Imdlawn Tashlhiyt, the spreading of the voicing feature can be blocked by any intervening voiceless obstruent. At the same time, similarly to Tuareg, the anteriority harmony exhibits no blocking effect. Consider the data from (Elmedlaoui, 1995; Hansson, 2010a) in (12-18), where the causative prefix *s-* illustrates the harmonic pattern.

- (12) s:-uga 'CAUS-evacuate'
 (13) s-as:twā 'CAUS-settle'
 (14) ʃ-fiaʃr 'CAUS-be.full.of.straw'
 (15) z-bruz:a 'CAUS-crumble'
 (16) ʒ-m:ʒdawl 'CAUS-stumble'
 (17) s-mχazaj 'CAUS-loathe.each.other'
 (18) ʃ-quʒ:i 'CAUS-be.dislocated'

In (12), there are no sibilants in the root, so the prefix is realized as *s-*. Examples (13-16) show that as previously, the causative affix agrees with the stem sibilant in voicing and anteriority. However, the voicing harmony can be blocked, and it is exemplified in (17) and (18). In (17), the sibilants are both anterior, but χ is stopping the regressive spreading of [+voice], so the prefix is realized as voiceless. Similarly, in (18), both sibilants are non-anterior, but the voicing spreading is also blocked, this time by *q*.

In Imdlawn Tashlhiyt, voiceless obstruents are *blockers* for the voicing harmony, and this prevents SP grammars from being able to model the generalization. The rules of the harmony are the same as before, therefore all restrictions discussed earlier in Grammar 2.5 are still valid. However, there is no way to express a blocking effect in an SP grammar. An SP restriction is a restriction of the precedence of one segment by another, and therefore the presence of the bigram *sz* is necessary to rule out disharmonic words such as *saz:twā*. But the same restriction will necessarily rule out grammatical words such as *smχazaj*: it will simply “miss” the blocker, see Figure 2.7.

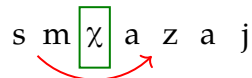


Figure 2.6: SP grammar incorrectly rules out Imdlawn Tashlhiyt word *smχazaj*.

Unbounded tone plateauing The ability of SP grammars to see substructures independently of other elements of the string gives them the power to encode *unbounded tone plateauing* (UTP) attested in Luganda (Niger-Congo). In that language, low tones are realized as high if they are surrounded by high tones (Hyman and Katamba, 2010). This makes it impossible for a well-formed word in Luganda to have low tones surrounded by high tones, see data in (19) cited by (Hyman, 2011; Jardine, 2016a). Accented vowels indicate high tones, and other vowels are low.

- (19) bikópo byaa-walúsiimbi → bikópó byáá-wálúsiimbi
‘the cups of Walusimbi’

Using letters H and L to indicate high and low tones, we can express the generalization as “never have one or more L in-between two H ”. This allows for strings such as HHL and $LLLHHHL$, but prohibits ones such as $HHLLLHH$. Due to the long-distant nature of SP grammars, a 3-local SP grammar can capture UTP by ruling out words that contain a subsequence HLH , see Figure 2.7.

L L H H L L H H L L L H


Figure 2.7: SP grammar captures the UTP pattern.

SP grammar Luganda unbounded tone plateauing
 $\Sigma = \{H, L\}$
 $R = \langle HLH \rangle$
 $k = 3$

Grammar 2.5: 3-SP grammar for Luganda unbounded tone plateauing.

Strictly piecewise grammars capture one or more long-distance processes that do not exhibit blocking effects. They prohibit subsequences of strings, therefore

ruling out all words that contain the illicit substructure. For example, the restriction zs means that nowhere in the string, z can be followed by s . However, blocking effects cannot be captured via an SP grammar, because the grammar is not sensitive to the presence of the blockers that make the banned substructure acceptable. Purely short-distant restrictions such as the word-final devoicing also cannot be expressed by an SP grammar: the restriction $b\bowtie$ prohibits *any* string containing b , because \bowtie always follows b in a word annotated with the word-final marker \bowtie .

Formal definition

An SP grammar is defined as a list of subsequences prohibited in well-formed strings of its language. Further, I define the notion of the subsequence and the SP languages formally following Rogers et al. (2010), and then provide an alternative definition in automata-theoretic terms.

Definition 2.1.4 (Subsequences)

A subsequence relation $v \sqsubseteq w$ is a partial order on the elements of Σ^* . A string v is a subsequence of w , $v \sqsubseteq w$, if v is an empty string, or if $v = \sigma_1\sigma_2\ldots\sigma_n$, and there is a collection of substrings $x_1, \ldots, x_n \in \Sigma^*$, such that those substrings can be placed between the elements of v thus obtaining $w = w_0\sigma_1w_1\ldots\sigma_nw_n$.

Then all k -long subsequences $P_k(w)$ of a word $w \in \Sigma^*$ can be computed as

$$P_k(w) = \{v \in \Sigma^k : v \sqsubseteq w\}$$

Similarly, $P_{\leq k}(w)$ lists all subsequences of $w \in \Sigma^*$ of the length up to k .

$$P_{\leq k}(w) = \{v \in \Sigma^{\leq k} : v \sqsubseteq w\}$$

For example, consider a string $w = abcd$. Then $P_3(w) = \{abc, abd, acd\}$, and $P_{\leq 3}(w) = P_3(w) \cup \{\epsilon, a, b, c, d, ab, ac, ad, bc, bd, cd\}$, where ϵ is the empty string.

Definition 2.1.5 (SP languages and grammars)

A k -SP grammar is a pair $\mathcal{G} = \langle \Sigma, S \rangle$ where $S \subseteq \Sigma^k$. The language licensed by a k -SP grammar is

$$L(\mathcal{G}) = \{w \in \Sigma^* : P_{\leq k}(w) \subseteq P_{\leq k}(T)\}.$$

The exact reason why SP grammars cannot capture the blocking effect is their *closure under subsequence*; see (Rogers et al., 2010) for other properties of SP languages and grammars.

Definition 2.1.6 (Subsequence closure)

Given a word $w \in L$, all strings v that are subsequences of w , $v \sqsubseteq w$, also belong to the language L : $v \in L$.

Alternatively, an SP language can be defined as a deterministic finite automaton (DFA) of a particular shape. Such a machine is a quintuple $\mathcal{M} = \langle Q, \Sigma, q_0, \delta, F \rangle$, where Q is a finite set of states, Σ is the alphabet, q_0 is the unique initial state, δ is the transition function, and F is the set of accepting states. The following properties are true for the automata recognizing SP languages. All states of \mathcal{M} are accepting, i.e. $F = Q$. If q_2 is reachable from q_1 and if there is no transition reading $\sigma \in \Sigma$ from q_1 , there will be no transition reading σ from q_2 (missing edges propagate down). All cycles are self-edges. If such a machine accepts a string $w \cdot v \cdot u : w, v, u \in \Sigma^*$, it necessarily accepts a string $w \cdot u$. This dependency, however, is not true in the other direction: if such a DFA accepts $w \cdot u$, there could be $v \in \Sigma^*$ such that $w \cdot v \cdot u$ is not an acceptable input sequence. A DFA with these properties accepts only SP languages.

2.1.5 Long-distant dependencies with blocking as TSL languages

Earlier, I showed that SL and SP grammars cannot capture long-distance harmonies with a blocking effect. SL grammars can only express local

generalizations, and SP restrictions target certain subsequences and therefore are not sensitive to the presence of blockers. Tier-based strictly local (TSL) grammars capture long-distance dependencies by *making them local* over the tier (Heinz et al., 2011).

Intuitive definition

I demonstrate the capacities of TSL grammars using the examples of vowel harmony in Karajá (ATR harmony with nasalized blockers) and Buryat (ATR and rounding harmony without blockers).

Karajá vowel harmony Consider a vowel harmony in Karajá (Macro-Jê), where a tense vowel spreads the advanced tongue root (ATR) feature leftwards. It makes it impossible to have a lax vowel followed by a tense one. This spreading can be blocked by intervening nasalized vowels; they are opaque for this harmony (Ribeiro, 2002).

(20)	woku	[woku]	‘inside’
(21)	dɔɾɛ	[dɔɾɛ]	‘parrot’
(22)	budɛ	[budɛ]	‘little, few’
(23)	brɔɾɛdĩ	[broɾɛɲĩ]	‘cow (<i>lit.</i> deer-similar.to)’
(24)	dɔɾɛ de	[dorede]	‘parrot’s wing’
(25)	rakɔhɔdɛkõre	[rakɔhɔdɛkõre]	‘He/she didn’t hit it.’
(26)	ɾɛbõre	[ɾɛmõre]	‘I caught (it).’

The data in (20-26) exemplifies the rule of harmony and is discussed in more detail in (Ribeiro, 2002). Note, that the harmony is reflected in the transcriptions and not in the orthography of the language. Stems in Karajá can contain tense (20) or lax (21) vowels, and the lax vowels can only follow the tense ones (22). A tense vowel starts its harmonic domain and spreads the [+ATR] feature regressively (23-

24). However, nasalized vowels such as \tilde{o} and \tilde{e} are opaque for this spreading: they do not enforce the agreement of the vowel thus allowing lax vowels to precede the nasal ones, even if a tense vowel follows it (25-26).

A TSL grammar is defined for a *tier alphabet* T that includes all segments that are relevant for the long-distance dependency. All vowels are relevant for the harmony, i.e. they are either undergoers (lax vowels), or blockers (nasalized vowels), or start the harmonic domain (tense vowels). Therefore in this case, the tier alphabet contains all vowels, $T = \{\varepsilon, o, e, u, \text{ɔ}, \tilde{o}, \tilde{e}, a, \dots\}$. A *tier image* of the string is a representation of that string where only the elements of T are preserved. For example, the tier image of *rakɔhɔdɛkõre* is *aɔɔεðe*. Finally, the set of restrictions R is defined for the tier representations of the string. In other words, the prohibited elements are the substrings that must not be observed in the tier representations of well-formed words of the language.

To construct the tier grammar for the Karajá vowel harmony, one needs to prohibit all combinations of a lax vowel followed by a tense one, i.e. $\varepsilon e, \varepsilon o, \text{ɔ} o, \text{ɔ} u$, etc. The presence of the nasalized vowels on the tier allows for sequences such as $\varepsilon \dots \tilde{e} \dots e$, where the opaque element blocks spreading of the ATR feature. Indeed, in such cases, the lax vowel ε and the tense e are not tier adjacent because of the intervening \tilde{e} . This makes TSL grammars a good fit for many harmonic patterns, even if they exhibit blocking effects.

$$\begin{aligned}
 \text{TSL grammar} \quad & \text{Karajá vowel harmony in ATR} \\
 \Sigma = & \{ \varepsilon, \tilde{o}, \tilde{e}, o, e, u, \text{ɔ}, a \dots b, d, r \dots \} \\
 T = & \{ \varepsilon, \tilde{o}, \tilde{e}, o, e, u, \text{ɔ}, a \dots \} \\
 R = & \langle \varepsilon e, \varepsilon o, \text{ɔ} o, \text{ɔ} u, \varepsilon u \dots \rangle \\
 k = & 2
 \end{aligned}$$

Grammar 2.6: 2-TSL grammar for Karajá vowel harmony in ATR.

See Figure 2.8 for the visualization of how the TSL grammar evaluates strings.

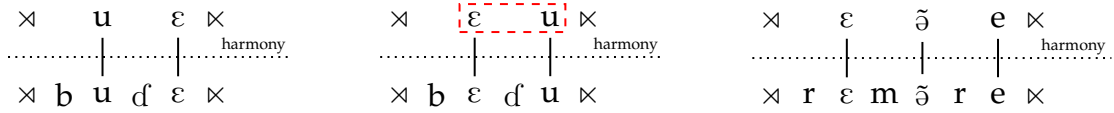


Figure 2.8: Evaluation of strings *budε*, *bεdu* and *reb̃re* by a TSL grammar capturing Karajá vowel harmony in ATR.

The word *budε* is well-formed, since the tense vowel is not preceded by any lax vowels, however, *bεdu* contains the violation *εu* and therefore is ruled out. In *reb̃re*, there is a lax vowel followed by a tense one, but there is a blocker *ã* in-between them, and its presence on the tier breaks the locality between *ε* and *e* therefore allowing such a configuration.

Buryat vowel harmony Now, let us consider a vowel harmony in Buryat (Mongolian) that spreads both ATR and rounding features. All vowels within a word must agree in ATR. Consecutive non-high vowels agree in rounding unless there is an intervening high vowel that blocks this assimilation (Poppe, 1960). The set of transparent items is the same for both agreements: it includes /i/ and all consonants (van der Hulst and Smith, 1987; Skribnik, 2003; Svantesson et al., 2005).

- (27) ɔr-ɔ:d 'enter-PERF'
- (28) ɔr-ʊ:l-a:d 'enter-CAUS-PERF'
- (29) to:r-o:d 'wander-PERF'
- (30) to:r-u:l-e:d 'wander-CAUS-PERF'
- (31) mɔrin-ɔ: 'horse-POSS'
- (32) o:rin-go: 'group-POSS'

Examples (27-32) illustrate the harmony using causative and perfective suffixes. The causative suffix *-ʊ:l* (*-u:l*) has its vowel specified as high, therefore it agrees with the stem only in ATR. A non-high vowel of the perfective affix *-a:d* (*-ɔ:d*, *-e:d*, *-o:d*) agrees with the preceding segment in ATR and, if that segment is non-high,

in rounding. In (27), the non-high perfective affix agrees with the non-high root vowel in ATR and rounding: both vowels are lax and rounded. But adding the high causative affix in-between them, as in (28), results in the blocking of the labial spreading: the perfective affix no longer agrees with the stem in rounding, because they are separated from each other by the intervening high vowel. Examples (29-30) show the same effect for the tense roots, and (31-32) demonstrate the transparency of the vowel /i/.

All vowels except /i/ harmonize, and therefore in this case, $T = \{a, e, \text{ɔ}, o, \text{ʊ}, u\}$. For example, the tier image of *to:ru:le:d* is *oue*.⁵ To create a list of tier restrictions, we need to understand what sequences of harmonizing vowels need to be ruled out. First, such a TSL grammar includes all bigrams where vowels disagree in tense because the tense harmony cannot be blocked by anything. It rules out 18 tier restrictions of the type $[\alpha\text{tense}][-\alpha\text{tense}]$, i.e. $\text{ɔ}o, o\text{ɔ}, \text{ʊ}u, u\text{ʊ}$, etc. Then, we enforce the agreement of tier-adjacent non-high vowels by prohibiting bigrams such as $[-\text{high}, \alpha\text{round}][-\text{high}, -\alpha\text{round}]$. It rules out 8 combinations such as $\text{ɔ}a, a\text{ɔ}, eo$, and others. Finally, we block rounded vowels from following high vowel, i.e. $[+\text{high}][-\text{high}, +\text{round}]$. That results in prohibiting $\text{ʊ}\text{ɔ}, uo, u\text{ɔ}$, and $\text{ʊ}o$. In such a way, we encode Buryat vowel harmony in ATR and rounding using a TSL grammar in 2.7.

$$\begin{aligned}
\text{TSL grammar} \quad & \text{Buryat vowel harmony in ATR and rounding} \\
\Sigma = \quad & \{a, b, t, o, \text{ɔ}, e, d, l, \dots\} \\
T = \quad & \{a, e, \text{ɔ}, o, \text{ʊ}, u\} \\
R = \quad & \langle \text{ɔ}o, o\text{ɔ}, \text{ʊ}u, u\text{ʊ} \dots \text{ɔ}a, a\text{ɔ}, eo, oe, ao \dots \text{ʊ}\text{ɔ}, uo, u\text{ɔ}, \text{ʊ}o \rangle \\
k = \quad & 2
\end{aligned}$$

Grammar 2.7: 2-TSL grammar for Buryat vowel harmony in ATR and rounding.

Figure 2.9 shows that the tier images of *to:ro:d* and *to:ru:le:d* are *oo* and *oue*,

⁵The length of vowels is ignored since it is not relevant for the rules of the harmony.

respectively. These tiers are well-formed: both words are tense, and in both cases, the second vowel inherits its rounding feature from the first non-high vowel, however, its value cannot be passed from a high vowel to a non-high one. The word *to:re:d* is illicit since its tier image *oe* is prohibited: vowels must agree with the preceding non-high vowel in rounding. The form *to:ru:lɔ:s* is also ruled out, since its tier *ouɔ* contains the prohibited bigram *uɔ*, since *ɔ* cannot inherit its rounding value from the preceding high vowel *u*, and they also disagree in ATR. In such a way, TSL grammar captures a pattern where a certain set of elements exhibits a long-distance dependency.

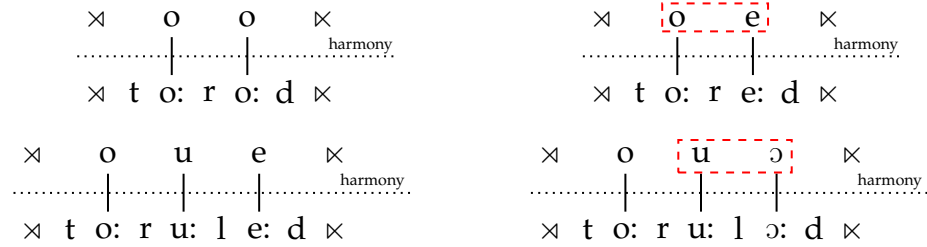


Figure 2.9: Evaluation of strings *to:ro:d*, *to:ru:le:d*, *to:re:d* and *to:ru:lɔ:s* by a TSL grammar capturing Buryat vowel harmony in ATR and rounding.

Apart from the long-distant patterns, TSL grammars can easily capture local dependencies since they are a proper extension of SL languages, see section 2.1.2. If a purely local dependency such as obstruent word-final devoicing or obstruent cluster voicing assimilation is expressed via a TSL grammar, its tier alphabet T will be the same as Σ .

As of the patterns discussed above, the Tuareg sibilant harmony in voicing and anteriority can also be described by a TSL grammar. In this case, the tier includes all sibilants. However, a similar harmony of Imdlawn Tashlhiyt, where only the voicing assimilation can be blocked by voiced obstruents, is not TSL. The anteriority harmony cannot be blocked, and the appearance of the voiced obstruents on the tier would break the locality relation between the sibilants. The

absence of the voiceless obstruents on the tier, however, makes it impossible to model the blocking of the voicing harmony. This creates two sets of items involved in a long-distance dependency: sibilants that are relevant for the anteriority harmony, and both sibilants and voiceless obstruents that are involved in the voicing harmony. It would thus require two tiers to capture the Imdlawn Tashlhiyt pattern; see McMullin (2016) and Aksënova and Deshmukh (2018) for the discussion of harmonies in more than a single feature, and tier properties of those harmonies.

Similarly, it is impossible to express a UTP generalization “no low tones in-between high tones” using a TSL grammar. Both L and H tones are important for the pattern, but including them both on the tier would make it impossible to notice the HLH configuration: the presence of the additional Ls, such as in *HHLLLLLLH*, makes the dependency non-local.

To sum up, TSL languages capture long-distance dependencies that can potentially include blocking or licensing effects. However, several long-distant assimilations cannot be expressed by a single TSL grammar if they affect different sets of segments.

Formal definition

TSL languages are a proper extension of the SL ones, but the k -local constraints are imposed on the *tier* symbols $T \subseteq \Sigma$. De Santo and Graf (2019) define tier-locality using the notion of the *erasing function* E , also called the *projection function*. Its purpose is to delete all symbols that are not included in the tier alphabet T . Given some string $\sigma \in \Sigma$, the erasing function E_T maps σ to itself if $\sigma \in T$ and to ϵ otherwise. In such a way, under the erasing function, a tier image of a word $w = \sigma_0 \dots \sigma_n$ is obtained by substituting non-tier elements $\sigma \notin T$ by ϵ .

Definition 2.1.7 (TSL languages and grammars)

A language L is tier-based strictly k -local (k -TSL) iff there exists a tier $T \subseteq \Sigma$ and a finite

set $S \subseteq F_k(\bowtie^{k-1}T^*\bowtie^{k-1})$ such that

$$L = \{w \in \Sigma^* : F_k(\bowtie^{k-1}E_T(w)\bowtie^{k-1}) \cap S = \emptyset\}$$

Additionally, S the set of forbidden k -factors on tier T , and $\langle T, S \rangle$ is a k -TSL grammar.

De Santo and Graf (2019) show that a language L is TSL iff it is strictly k -local on tier T for some $T \subseteq \Sigma$ and $k \in \mathbb{N}$. Indeed, SL properties such as suffix substitution closure (see the definitions in 2.1.3) can be generalized, as it was done by Lambert and Rogers (2020). For example, a language $x^*a^*x^*b^*x^*$ is not SL since it is not closed under suffix substitution. However, if only a and b are included in the tier alphabet, it becomes 2-TSL, with the shape of the tier restricted to a^*b^* .

In their paper, Lambert and Rogers (2020) show how to construct a DFA for a given TSL grammar. Namely, they start by encoding every allowed k -TSL factor in its own automaton, uniting all the automata obtained this way, and then adding loops reading non-tier symbols to every state. In such a way, a TSL-representing DFA is a DFA expressing the local restrictions on the tier symbols, and looping on the non-tier ones.

2.1.6 Multiple long-distant dependencies with blocking as MTSL languages

Multi-tier strictly local (MTSL) grammars are conjunction of several TSL grammars. A language of an MTSL grammar is the intersection of languages of multiple TSL grammars (De Santo and Graf, 2019). An MTSL grammar lists k tier alphabets $T_1 \dots T_k$, and for every tier alphabet T_i , there is a corresponding set of restrictions R_i . A string is well-formed with respect to a given MTSL grammar if it is well-formed on every tier.

Intuitive definition

Imdlawn Tashlhiyt sibilant harmony exhibits a blocking effect for only one feature out of two that are spreading (voicing and anteriority). In this subsection, I show that this pattern is MTSL.

Imdlawn Tashlhiyt Consider a regressive sibilant harmony in Imdlawn Tashlhiyt discussed earlier in 2.1.4. Sibilants in that language agree in voicing and anteriority. For example, in *zbruz:a* ‘CAUS-crumble’, both sibilants are voiced and anterior, and in *sas:twa* ‘CAUS-settle’, they are voiceless and anterior, in *[fia]r* ‘CAUS-be.full.of.straw’, they are voiceless and non-anterior. However, while the anteriority harmony cannot be blocked by anything, the voicing harmony can be blocked by intervening voiceless obstruents such as χ , k or q , resulting in the well-formedness of words such as *smʃazaj* ‘CAUS-loathe.each.other’.

This pattern is not SL, SP or TSL. It is not SL since it involves a long-distance dependency. An SP grammar cannot capture it either because it cannot model blockers. A TSL grammar is not a good fit either: both sibilants and voiceless obstruents need to be present on the tier to express the voicing harmony; however, the presence of non-sibilants on the tier breaks the tier locality required to model the anteriority assimilation. More than a single tier is required to model this generalization.

The power of MTSL grammars allows projecting multiple tiers, and this helps to model the Imdlawn Tashlhiyt pattern. Sibilants and voiceless obstruents are projected on one tier, let us call it T_{voice} , whereas only sibilants are visible on the second tier T_{ant} . The tier capturing the voicing harmony restricts all combinations of sibilants disagreeing in voicing (sz , zs , $ʃʒ$, $ʒʃ$), and also voiced sibilants followed by voiceless obstruents (zk , zf , $z\chi$, etc.). On the tier of anteriority, the combinations of sibilants of different anteriority are not allowed ($sʃ$, $zʃ$, $ʃs$, etc.). The obtained grammar is summarized in 2.8.

MTSL grammar Imdlawn Tashlhiyt sibilant harmony in voicing and anteriority

$$\begin{aligned}
\Sigma &= \{a, b, m, u, g, r, s, z, \text{ʃ}, \text{ʒ}, h, k, f, \chi, q, \dots\} \\
T_{ant} &= \{s, z, \text{ʃ}, \text{ʒ}\} \\
R_{ant} &= \{s\text{ʃ}, z\text{ʃ}, \text{ʃ}s, \text{ʃ}z, s\text{ʒ}, z\text{ʒ}, \text{ʒ}s, \text{ʒ}z\} \\
T_{voice} &= \{s, z, \text{ʃ}, \text{ʒ}, h, k, f, \chi, q\} \\
R_{voice} &= \{sz, zs, \text{ʃ}\text{ʒ}, \text{ʒ}\text{ʃ}, zh, zk, zf, z\chi, zq, \text{ʒ}h, \text{ʒ}k, \text{ʒ}f, \text{ʒ}\chi, \text{ʒ}q\} \\
k &= 2
\end{aligned}$$

Grammar 2.8: 2-MTSL grammar for Imdlawn Tashlhiyt sibilant harmony in voicing and anteriority.

This MTSL grammar correctly models the generalization behind the Imdlawn Tashlhiyt pattern. Ill-formed strings such as *ʒbruz:a* are illicit because the tier of the anteriority harmony contains a prohibited bigram *ʒz*. The combinations of sibilants that agree in voicing are allowed on that tier, and it is the case in well-formed words such as *smʃazaj*, where *ʃ* blocks the voicing agreement. Voiced sibilants cannot precede voiceless obstruents, so forms such as *zmʃazaj* are ruled out on the tier of voicing by the restriction *zʃ*. Figure 2.10 visualizes the discussed examples.

MTSL grammars model several agreements within the same language, even when they target different sets of segments. Interestingly, those sets are either in the set-subset relation, or are disjoint, but never only partially overlap. Imdlawn Tashlhiyt discussed in this section is an example of a harmony where the tier alphabets are in the set-subset relation. In Bukusu (Bantu), vowels agree in height along with the assimilation of nasals in height, therefore exemplifying the case of the disjoint tier alphabets (Odden, 1994; Elmedlaoui, 1995; Hansson, 2010a). Aksënova and Deshmukh (2018) summarize this restriction and propose its possible explanation.

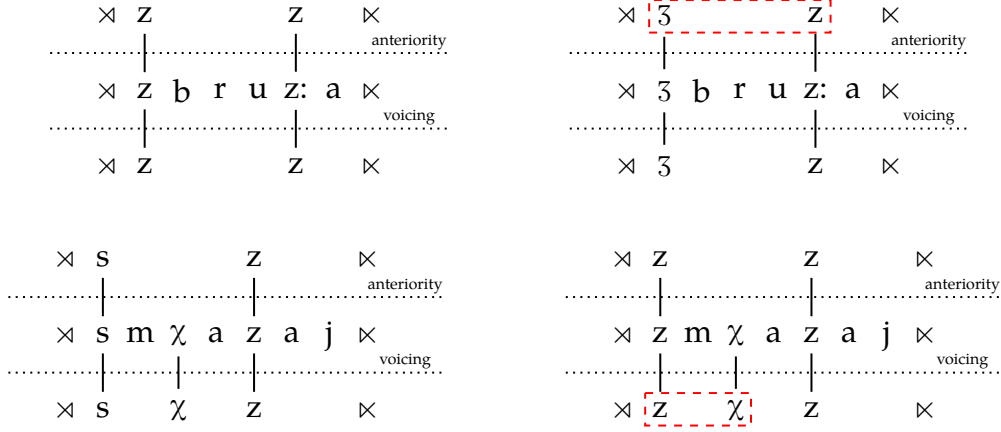


Figure 2.10: Evaluation of strings *zbruz:a*, *ʒbruz:a*, *smʃazaj* and *zmʃazaj* by a MTSL grammar capturing Imdlawn Tashlhiyt sibilant harmony in voicing and anteriority.

Formal definition

The language at the intersection of several TSL grammars can be viewed as a single grammar projecting multiple tiers, i.e. a multi-TSL, or MTSL grammar (De Santo and Graf, 2019).

Definition 2.1.8 (MTSL languages)

An n -tier strictly k -local (n -MTSL $_k$) language L is the intersection of n distinct k -TSL languages ($k, n \in \mathbb{N}$).

Since the language of an MTSL grammar is the intersection of several TSL languages, one can imagine a construction of the corresponding DFA by intersecting several DFAs representing those TSL languages. According to Lambert and Rogers (2020), it is possible to create DFAs for any languages definable by Boolean combinations of SL, SP, and TSL automata. Consequently, it includes MTSL languages.

2.1.7 Models of well-formedness conditions: summary

Subregular grammars describe well-formedness conditions imposed on words in natural languages. In this section, I focused on SL, SP, TSL, and MTSL grammars since they capture a vast majority of phonological patterns. SL grammars capture only local processes, therefore they are a good fit for word-final devoicing and obstruent cluster voicing assimilation. SP grammars generalize long-distance dependencies by prohibiting certain orders of elements, thus they are a good fit for harmonies without blockers and unbounded tone plateauing. Only a certain subset of elements of the alphabet is projected on a tier by TSL grammars, therefore they capture a long-distance process locally by ignoring irrelevant segments. This allows them to model a harmony process that can potentially involve blockers. Finally, MTSL grammars project several tiers, and this helps to represent several independent yet simultaneous harmonies. In such a way, these subregular grammars can capture most of the dependencies imposed by phonological well-formedness conditions.

Some known natural language patterns, however, require powers of different subregular language classes. Among them, there is a pattern of Sanskrit /n/-retroflexion and Yaka harmony triggered by local conditions (Walker, 2000; McMullin, 2016; Karakaş, 2020). IO-TSL and IBSP subregular language classes can model those cases (Graf, 2017b, 2018a). Interestingly, none of the subregular language classes mentioned above can express such theoretically possible but unattested patterns as first-last harmony, where the first vowel of the word needs to harmonize with the last one (Avcu, 2017).

Further in chapter 3 of this dissertation, I discuss several subregular patterns and show how the corresponding grammars can be learned from the real data.

2.2 Modeling transformations

The previous section explored modeling of natural language well-formedness conditions using subregular languages. Here, I discuss formalizing and modeling *transformations*, or rewrite rules, that apply to underlying representations and yield corresponding surface forms. Namely, I show that subsequential finite-state *transducers* can be employed to model transformations, and discuss their sub-types. Subsequential mappings belong to a class of regular functions, and therefore subsequential functions are also subregular.

2.2.1 Formalizing transformations

Earlier in Section 2.1.5, I discussed Buryat progressive vowel harmony in ATR and rounding. Vowels agree in these two features, while consonants and /i/ are transparent. While all harmonizing vowels are undergoers for the ATR agreement, high vowels *ʊ* and *u* block the spreading of the rounding feature, thus prohibiting the appearance of *ɔ* and *o*. So, for example, all vowels in a word *to:r-u:l-e:d* ‘wander-CAUS-PERF’ are tense, and the high vowel *u* blocks spreading of the rounding feature. In *to:r-o:d* ‘wander-PERF’, all vowels are low, and therefore they agree in rounding as well. Values of non-initial vowels only depend on their height and the value of the previous vowel.

Now, consider this harmony as a set of pairs exemplifying the mapping from the underlying representations (UR) to the surface forms (SF), see data in (33-38).

A UR of the initial vowel is fully specified, whereas all non-initial vowels are only specified for the height, and therefore are denoted as **Low** or **High**. To obtain the SFs, all *L* and *H* of the URs need to be substituted starting from the leftmost one. The rules of this substitution are listed below.

- | | | | | |
|------|--------------|---|--------------|--------------------|
| | UR | → | SF | |
| (33) | ɔr-L:d | → | ɔr-ɔ:d | ‘enter-PERF’ |
| (34) | ɔr-H:l-L:d | → | ɔr-ʊ:l-a:d | ‘enter-CAUS-PERF’ |
| (35) | to:r-L:d | → | to:r-o:d | ‘wander-PERF’ |
| (36) | to:r-H:l-L:d | → | to:r-u:l-e:d | ‘wander-CAUS-PERF’ |
| (37) | mɔrin-L: | → | mɔrin-ɔ: | ‘horse-POSS’ |
| (38) | o:rin-gL: | → | o:rin-go: | ‘group-POSS’ |

$$L = \left\{ \begin{array}{l} \text{'ɔ' if the previous harmonizing vowel is 'ɔ'} \\ \text{'a' if the previous harmonizing vowel is 'a' or 'ʊ'} \\ \text{'o' if the previous harmonizing vowel is 'o'} \\ \text{'e' if the previous harmonizing vowel is 'e' or 'u'} \end{array} \right\}$$

$$H = \left\{ \begin{array}{l} \text{'ʊ' if the previous harmonizing vowel is 'ɔ', 'a' or 'ʊ'} \\ \text{'u' if the previous harmonizing vowel is 'o', 'e' or 'u'} \end{array} \right\}$$

Previously, Section 2.1.1 introduced finite-state automata (FSA) that are defined via a finite number of states and transitions between them. Those transitions are annotated with symbols, and strings are read by following the corresponding transitions. If such transitions exist and the last portion of the string brings the machine to the final state, the string is *accepted*, i.e. considered well-formed with respect to the rules encoded by the FSA. Otherwise, the string is *rejected*. In such a way, an FSA reads a string and evaluates its well-formedness.

The transitions of the FSA have the following shape: $q_a \xrightarrow{b} q_b$. Such a transition indicates that after reading b , the FSA will move from the state q_a to q_b . The core difference of a **finite-state transducer** (FST) is that it can *write* an output string while reading the input string (Schützenberger, 1961). Transitions of the FST indicate what portion of the input is read, and what is added to the output string, also called the *translation*. For example, the transition $q_a \xrightarrow{b:x} q_b$ means “to go from q_a to q_b , read b and write x ”. Consider a simple example of an FST in Figure 2.11. While FSAs

are functions mapping *strings* to *boolean values*, FSTs map *strings* to *strings*.

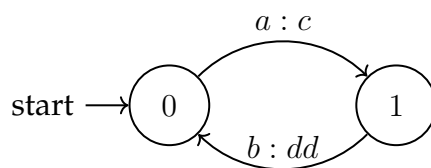


Figure 2.11: An example of the FST.

A transducer in Figure 2.11 accepts strings that start with *a* and alternate *a* and *b*. Every time *a* is read, *c* is written, and every time *b* is read, *dd* is written. It translates *aba* as *cddc*, and *abab* as *cddcdd*. I employ **rational** transducers in which all states can be final.

We can construct an FST mapping Buryat URs to the corresponding SFs, as depicted in Figure 2.12. For simplicity, *N* represents all elements that are not involved in the harmony – consonants and the transparent vowel /i/. Such a transducer has 5 states, with q_0 being the initial state. Every state has a loop $N:N$ on it, and it means that the irrelevant symbols for the harmony are left as is and do not move the machine to another state. If the initial vowel is *e* or *u*, it moves the machine to state q_1 , and all the following low and high vowels are rewritten as *e* and *u*, respectively. If the initial vowel is *o*, state q_2 is activated, so all the following low vowels agree in tense and rounding, and therefore are realized as *o*. However, reading a high vowel *u* moves the machine to q_1 thus blocking the rounding spreading. States q_3 and q_4 similarly encode the rounding harmony for lax stems. In such a way, the transducer in Figure 2.12 takes a Buryat UR as input and returns the corresponding SF as output, where the underspecified segments *H* and *L* are rewritten with respect to the rules of the harmony.

For example, assume that the word *ɔrLd* is given as input. The initial *ɔ* brings the machine to the state q_4 , and *r* is rewritten as *r* because of the loop annotated with $N:N$. The following *L* is now specified: the reflexive loop on the state q_4 changes its value to *ɔ*. Finally, *d* is left without modifications. The FST in 2.12

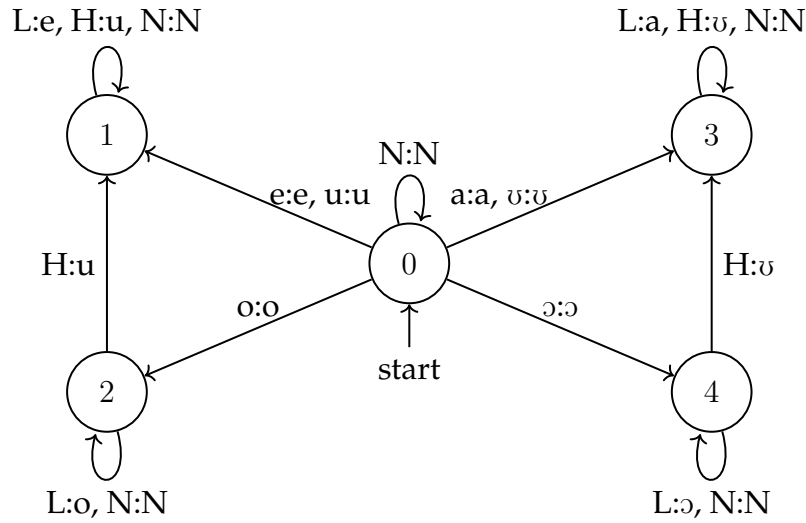


Figure 2.12: Transducer for Buryat vowel harmony.

rewrites that input form as *ɔrɔd* and it is indeed the expected output, see the example (33). Alternatively, consider the UR *torHILd*. The initial vowel *o* brings the machine to the state q_2 , meaning that all the consecutive vowels will be tense. The following underspecified vowel is high, so it is rewritten as *u*. Reading that high vowel moves the FST to q_1 . The spreading of the rounding feature is blocked, and given that the following vowel is low, it is realized as *e*. This results in the translation *toruled*, and the example (36) confirms it. In such a way, FSTs model phonological processes that change URs to the corresponding SFs.

2.2.2 Subsequential mappings

In line with the regularity of languages satisfying the well-formedness conditions, phonological mappings are also regular (Johnson, 1972; Koskeniemi, 1983; Kaplan and Kay, 1994). A *regular mapping* is one that can be represented using a FST, similarly to the way the pattern of Buryat vowel harmony is represented as a FST in figure 2.12.

Similar to the results discussed in the previous sections, resent research shows

that phonological mappings also do not require a full power of regular languages. Instead, subsequential mappings including input and output strictly local transformations were shown to be a good fit for natural language phonological and morphological processes (Chandlee, 2014; Chandlee and Heinz, 2018). Therefore, here, as well as in Chapter 4, I focus on *subsequential transducers* representing subsequential mappings.

The term “subsequential”, however, can be confusing. It can be interpreted as *is below sequential* or *subsumes sequential*, and is used in the literature in both of these meanings. Here following Roche and Schabes (1997) and de la Higuera (2010), I use “subsequential” to mean a subclass of sequential machines. A **sequential** FST reads symbols of the input string one by one. Crucially, for every input symbol, there is at most one transition outgoing from any state of such FST that reads that symbol. Additionally, **subsequential** machines implement a *state outputting function* that allows for the final portion of the translation to be generated at the end of the parse, depending on the state in which the FST stopped after reading the last input symbol.

In Section 2.1.3, I described the pattern of word-final devoicing in Russian, where voiced obstruents are realized as voiceless at the end of the word. For example, *lob* ‘forehead’ is pronounced as $lo[p]$, and l^jod ‘ice’ as $l^jo[t]$. For simplicity, assume that all voiced obstruents are encoded as B , all voiceless ones are P , and other segments that are not relevant for the process of word-final devoicing as N . The main rule of the target transducer is then to re-write all word-final B s as P s. This simplified alphabet is used in Figure 2.13 showing a subsequential FST for Russian word-final devoicing.

Consider how the transducer in Figure 2.13 rewrites the Russian word *pedagog* ‘pedagogue’. This word corresponds to the sequence $PNBNBNB$ according to the simplified representation outlined above. The FST reads symbols of that sequence one by one starting from the initial state q_0 . At first, it reads P and N and outputs

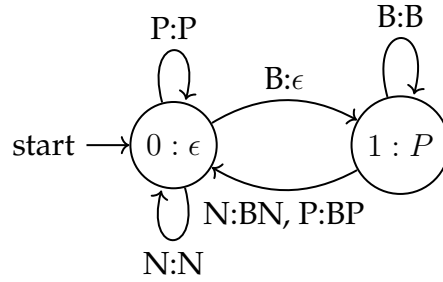


Figure 2.13: Subsequential FST for word-final devoicing.

the same symbols. The following *B* leads to the state q_1 , and that *B* is written together with the following *N* when the machine moves back to the state q_0 . It was delaying the output of *B* to make sure that only the word-final *B* is rewritten as *P*. Finally, when the FST reads the final *B*, it outputs *P* instead by the state output of q_1 , because it is the final activated state. The translation of *PNBNBNB* is *PNBNBNP*, that corresponds to correctly rewriting *pedagog* as *pedago[k]* since *[k]* is the voiceless counterpart of *[g]*.

Chandlee (2014), and later Chandlee and Heinz (2018) show that the majority of phonological processes can be modeled in similar ways. Chandlee (2017) later extends these results to morphology and models different types of affixation, word boundary processes and some types of reduplication. Other patterns that are shown to be subregular include metathesis, epenthesis, flapping, deletion, harmonies, and many others (Chandlee, 2014). Even some of the suprasegmental processes, such as stress, seem to exhibit subregular properties (Rogers, 2018). For a survey of vowel harmonies and their computational complexities, see (Gainor et al., 2012). However, some processes, such as reduplication, are not subregular (Dolatian and Heinz, 2018).

The limitations imposed by subregular models provide insights into typology and cognition. They allow to detect and explain typological gaps, such as the absence of first-last and sour grapes harmonic patterns⁶ (Heinz and Lai, 2013;

⁶The first-last vowel harmony enforces the agreement of the first and the last vowels within a

Luo, 2017), and give insights into human cognitive system (Rogers and Pullum, 2011; Lai, 2015; Avcu, 2017).

Although subsequential mappings fit a large portion of natural language patterns, there are phonological processes that are not subsequential. For example, bidirectional feature spreadings cannot be captured by a subsequential machine because the latter ones can read a string either left-to-right or right-to-left: it cannot pass the value in *both* directions. A transducer applying the rules of UTP to the underlying tonal representations is not subsequential either (see Sections 2.1.4 and 4.1.3): for a sequence of low tones to become high, it needs triggers on both sides. Such mappings are still regular, namely, weakly deterministic and circumambient (Heinz and Lai, 2013; Jardine, 2016a; Lamont et al., 2019). However, I will focus on a class of subsequential mapping since it subsumes a large portion of attested phonological processes.

2.2.3 Left and right subsequential mappings

Subsequential transducers are a good fit for local and long-distance phonological processes such as intervocalic voicing, word-final devoicing, assimilations, harmonies, and many others. In all the examples discussed so far (Buryat vowel harmony in Section 2.2.1 and Russian word-final devoicing in Section 2.2.2), the transducer reads the underlying representation left-to-right, correctly capturing the progressive feature spreading. Such transducers are **left subsequential**, in contrast to **right subsequential** FSTs that read the input string right-to-left.

Whereas a left subsequential transducer captures progressive harmonies, it fails to generalize regressive ones. In the latter, the *last* harmonizing segment contains the information about the feature value that needs to be spread. For example, in Tuareg, sibilants regressively assimilate in voicing and anteriority; see Section 2.1.3

word, and the sour grapes harmony spreads a harmonizing feature only if a blocker is not present (Heinz and Lai, 2013).

for details. The data below repeats the SFs presented in (7-11), together with the URs, where /S/ represents the underspecified sibilant.

- (39) S-əlməd → s-əlməd 'CAUS-learn'
(40) S-əq:usə → s-əq:usət 'CAUS-inherit'
(41) S-əntəz → z-əntəz 'CAUS-extract'
(42) S-əm:əfən → ʃ-əm:əfən 'CAUS-be.overwhelmed'
(43) S-ək:uʒət → ʒ-ək:uʒət 'CAUS-saw'

The right subsequential FST in Figure 2.14 implements the regressive Tuareg sibilant harmony. For simplicity, *N* refers to any segment that is not a sibilant. Every state has a loop reading *N* and writing the same *N*, therefore non-sibilants are not affected by the harmony in any way. Additionally, the loop on the state q_0 rewrites *S* as *s*, because an underspecified sibilant is realized as /s/ if there is no other sibilant to its right. For example, in (39), there is no sibilant in the root, and therefore the causative prefix is realized as /s/. In other cases (40-43), /S/ agrees with the root sibilant in voicing and anteriority. For example, in (43), the third segment from the end is ʒ that moves the FST to the state q_4 , and all following underspecified sibilants are rewritten as ʒ.

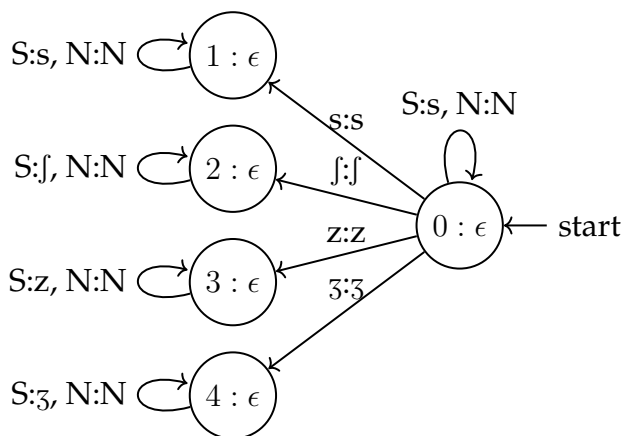


Figure 2.14: Right subsequential FST for Tuareg regressive sibilant harmony.

2.2.4 ISL and OSL mappings

In this section, I look at two different ways transformational rules can be applied to the underlying representations. Thus, instead of discussing linguistic patterns, I focus on the *manner of the rule application*. For instance, consider an SPE-style rule $a \rightarrow b/a_a$ from (Chandlee, 2014). There are two ways it can be applied. Given the UR *aaaaa*, the simultaneous application of this rule to all positions yields the SF *abbba*. However, if this rule was applied step-by-step, the obtained SF would be *ababa*, since the change of the second *a* to *b* makes the context of the third *a* incompatible with the one specified by the transformation. To reflect this difference, Chandlee (2014) introduces smaller subclasses of subsequential mappings: *input strictly local* (ISL) and *output strictly local* (OSL) ones. The ISL functions encode simultaneous rule application of strictly local functions, while the OSL ones reflect the iterative one. Figure 2.15 shows the relationship among left/right subsequential, ISL and OSL functions.

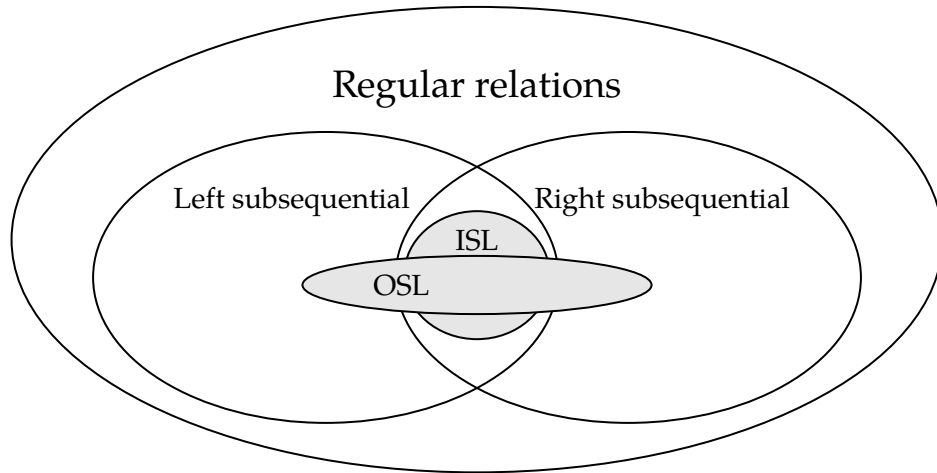


Figure 2.15: Relationship among left subsequential, right subsequential, OSL, and ISL functions; adapted from (Chandlee, 2014).

Chandlee et al. (2014) define both ISL and OSL mappings using the notion of **tails**. Tails show the dependency between the possible continuations of input

strings, and portions of the output contributed by those continuations. Formally, tails are defined as $\text{tails}(x) = \{(y, v) : f(x \cdot y) = u \cdot v \text{ and } u = \text{lcp}(f(x \cdot \Sigma^*))\}$, where f is the function mapping the input strings to the corresponding outputs, \cdot is the concatenation operator, and lcp is the longest common prefix. For example, the longest common prefix of abc and $abde$ is ab , since it is the longest prefix shared by those two strings. A slightly extended notation introduced below changes that definition to $\text{tails}(\vec{x}) = \{(y, v) : f(\vec{x} \cdot y) = \mathbf{u} \cdot v \text{ and } u = \text{lcp}(f(\vec{x} \cdot \Sigma^*))\}$.

Assume that Σ and Γ are (possibly different) alphabets used to represent the strings before and after application of some rule. Additionally, strings \vec{x} and y belong to Σ^* , and \mathbf{u} and v belong to Γ^* , i.e. these strings consist only of symbols included in Σ^* or Γ^* , respectively. A tail of the prefix \vec{x} is a list of all pairs (y, v) , where y is a possible continuation of the input prefix \vec{x} , i.e. $\vec{x} \cdot y$ is the input string. The corresponding to $\vec{x} \cdot y$ output string is $\mathbf{u} \cdot v$, where \mathbf{u} is the longest prefix shared by all outputs corresponding to inputs starting with \vec{x} .

As an example, let us find a list of tails of the input prefix $\overrightarrow{\times aa}$ in the mapping M . Note, that only the input strings are annotated with the edges \times and \times .

$$M = (\times aa \times, aa), (\times aaa \times, aba), (\times aaaa \times, abba), (\times aaaaa \times, abbba), \\ (\times aaaaaa \times, abbbbba), (\times aaaaaaa \times, abbbbbbba) \dots$$

All input strings of that mapping start with $\overrightarrow{\times aa}$. The longest common prefix of the corresponding output strings is \mathbf{a} : for example, the second symbol is different in aa and aba . Below, I mark the selected input prefix $\overrightarrow{\times aa}$ and the longest common prefix \mathbf{a} .

$$M_{\text{marked}} = (\overrightarrow{\times aa} \times, \mathbf{a}a), (\overrightarrow{\times aa} a \times, \mathbf{a}ba), (\overrightarrow{\times aa} aa \times, \mathbf{a}bba), (\overrightarrow{\times aa} aaa \times, \mathbf{a}bbba) \dots$$

The list of tails of $\overrightarrow{\times aa}$ can be computed by removing $\overrightarrow{\times aa}$ and \mathbf{a} from the input and output strings of M_{marked} , respectively.

$$tails(\overrightarrow{\times aa}) = (\times, a), (a \times, ba), (aa \times, bba), (aaa \times, bbba) \dots$$

The obtained list of tails implies that after observing $\overrightarrow{\times aa}$, the input continuation \times introduces a to the translation, $a \times$ contributes ba , and so on. Further in this subsection, I show how the notion of tails is used to define ISL and OSL mappings (Chandlee, 2014; Chandlee et al., 2014, 2015).

Input strictly local mappings

The k -ISL functions encode simultaneous rule application, i.e. when a transformational rule applied to all positions at the same time. In mapping M , for example, a is substituted by b if it is surrounded by a in the input string. It creates pairs such as $(aaaaa, abbba)$: three internal a are changed to b since their context in the input string is the same as specified by the rule $a \rightarrow b/a _ a$.

Chandlee et al. (2014) define an ISL mapping as follows. If two input strings u_1 and u_2 share the same $k - 1$ -local suffix, their set of tails is identical as well.

$$suff^{k-1}(u_1) = suff^{k-1}(u_2) \Rightarrow tails(u_1) = tails(u_2)$$

If the mapping M is indeed ISL, the set of tails is the same for all strings ending with the same $k - 1$ local prefix. Let us assume that $k = 3$, since the rule targets an item in the context of two other elements. Input strings $\times aa \times$ and $\times aaa \times$ both have the 2-local suffix $a \times$, and that definition states that their sets of tails must be identical as well. To confirm this, let us compare tails of $\times aa$ and $\times aaa$.

$$\begin{aligned} tails(\overrightarrow{\times aa}) &= (\overrightarrow{\times aa} \times, aa), (\overrightarrow{\times aa} a \times, aba), (\overrightarrow{\times aa} aa \times, abba) \dots = \\ &(\times, a), (a \times, ba), (aa \times, bba) \dots \end{aligned}$$

$$\begin{aligned} \text{tails}(\overrightarrow{\times a a a}) &= (\overrightarrow{\times a a a} \times, \mathbf{a b a}), (\overrightarrow{\times a a a} a \times, \mathbf{a b b a}), (\overrightarrow{\times a a a} a a \times, \mathbf{a b b b a}) \dots = \\ &(\times, a), (a \times, b a), (a a \times, b b a) \dots \end{aligned}$$

Their tails are indeed the same, and it confirms that the *simultaneous application* of the rule $a \rightarrow b/a _ a$ is an *ISL* function. The corresponding transducer encodes a 3-local window reading the input string. Knowledge of the previous 2 input symbols informs the transducer about the next action regarding the following symbol that it reads. Figure 2.16 demonstrates these steps.

<i>Input:</i>	$\times \ a \ a \ a \ a \ a \ \times$
<i>Output 1:</i>	\times
<i>Input:</i>	$\boxed{\times} \boxed{a} \boxed{a} \ a \ a \ a \ \times$
<i>Output 2:</i>	$\times \ a$
<i>Input:</i>	$\times \ \boxed{a} \boxed{a} \boxed{a} \ a \ a \ \times$
<i>Output 3:</i>	$\times \ a \ b$
<i>Input:</i>	$\times \ a \ \boxed{a} \boxed{a} \boxed{a} \ a \ \times$
<i>Output 4:</i>	$\times \ a \ b \ b$
<i>Input:</i>	$\times \ a \ a \ \boxed{a} \boxed{a} \boxed{a} \ \times$
<i>Output 5:</i>	$\times \ a \ b \ b \ b$
<i>Input:</i>	$\times \ a \ a \ a \ \boxed{a} \boxed{a} \boxed{\times}$
<i>Output 6:</i>	$\times \ a \ b \ b \ b \ a$
<i>Input:</i>	$\times \ a \ a \ a \ a \ a \ \times$
<i>Output 7:</i>	$\times \ a \ b \ b \ b \ a \ \times$

Figure 2.16: ISL application of the rule $a \rightarrow b/a _ a$ to $aaaaa$.

Output strictly local mappings

Now, let us consider the iterative application of the same rule $a \rightarrow b/a _ a$. In this case, every time the rule is applied, it changes the form that the same rule produced earlier, and therefore $aaaaa$ is changed to $ababa$. This type of transformation can be visualized as a 3-local window moving through the input string and rewriting the middle item if the contexts match. The steps below show the application of the rule; the underlined segments are the contexts, and the boxed items show how the target element was changed. The list of pairs in M' shows the corresponding mapping.

$$\times \boxed{a:a} \underline{aaaa} \times \rightarrow \times \underline{a} \boxed{a:b} \underline{aaa} \times \rightarrow \times \underline{ab} \boxed{a:a} \underline{aa} \times \rightarrow \times \underline{aba} \boxed{a:b} \underline{a} \times \rightarrow \times \underline{abab} \boxed{a:a} \times$$

$$M' = (\times aa \times, aa), (\times aaa \times, aba), (\times aaaa \times, abaa), (\times aaaaa \times, ababa), \\ (\times aaaaaa \times, ababaa), (\times aaaaaaa \times, abababa) \dots$$

Chandlee et al. (2015) shows that such mappings are k -OSL since the previous application of this rule affects the following one. Their definition of OSL mappings is below, where the function f , as previously, maps its argument (input string) to the corresponding output. For a mapping to be OSL, the following needs to be true: if two output strings share the same $k - 1$ -local suffix, the tails of the corresponding input strings are the same.

$$suf f^{k-1}(f(u_1)) = suf f^{k-1}(f(u_2)) \Rightarrow tails(u_1) = tails(u_2)$$

That would imply that in the mapping M' , tails of $\overrightarrow{\times aaa}$ and $\overrightarrow{\times aaaaa}$ are the same because the translations of $\times aaa \times$ and $\times aaaaa \times$ share the same $k - 1$ -local suffix ba (those translations are aba and $ababa$, respectively).

$$\begin{aligned} \text{tails}(\overrightarrow{\times aaa}) &= (\overrightarrow{\times aaa} \times, aba), (\overrightarrow{\times aaa} a \times, abaa), (\overrightarrow{\times aaa} aa \times, ababa) \dots = \\ &(\times, \epsilon), (a \times, a), (aa \times, ba) \dots \end{aligned}$$

$$\begin{aligned} \text{tails}(\overrightarrow{\times aaaaa}) &= \\ (\overrightarrow{\times aaaaa} \times, ababa), (\overrightarrow{\times aaaaa} a \times, ababaa), (\overrightarrow{\times aaaaa} aa \times, abababa) \dots &= \\ (\times, \epsilon), (a \times, a), (aa \times, ba) \dots \end{aligned}$$

Indeed, since their tails are the same, this shows that the mapping is OSL. The corresponding transducer encodes a 3-local window that keeps track of the last 2 output symbols. Based on those symbols and the current symbol it decides how that current symbol is changed. The iterative rule application is demonstrated in Figure 2.17.

To sum up, k -ISL and k -OSL mappings encode dependencies affecting k -local windows. ISL functions apply a rule simultaneously to all the positions of the input string, whereas the OSL functions apply a rule step-by-step. While in the former case, the changes are independent from each other, the latter uses information about the previous change to inform the following one. Chandlee (2014) argues that linguistic strictly local processes are ISL or OSL, and demonstrates it using a variety of linguistic examples such as Greek fricative deletion, English flapping, and others (Joseph and Philippaki-Warbuton, 1987).

2.2.5 Models of transformations: summary

FSTs encode regular mappings that are well-known to be a good fit for natural language phonology and morphology (Johnson, 1972; Kaplan and Kay, 1994; Beesley and Karttunen, 2003). However, a particular class of functions, namely,

<i>Input 1:</i>	× a a a a a ×
<i>Output 1:</i>	×
<i>Input 2:</i>	× a a a a a ×
<i>Output 2:</i>	× a
<i>Input 3:</i>	× a a a a a ×
<i>Output 3:</i>	× a b
<i>Input 4:</i>	× a a a a a ×
<i>Output 4:</i>	× a b a
<i>Input 5:</i>	× a a a a a ×
<i>Output 5:</i>	× a b a b
<i>Input 6:</i>	× a a a a a ×
<i>Output 6:</i>	× a b a b a
<i>Input 7:</i>	× a a a a a ×
<i>Output 7:</i>	× a b a b a ×

Figure 2.17: OSL application of the rule $a \rightarrow b/a_a$ to *aaaaa*.

subsequential, includes the major part of natural language patterns. Transducers implementing subsequential mappings read the symbols of the underlying representations one by one and output the corresponding surface forms. This allows one to model local and long-distance processes, such as word-final devoicing and vowel harmony.

It should be noted that some attested phonological processes are not subsequential. Among them, there are circumambient pattern of unbounded tonal plateauing and reduplication requiring the power of two-way FSTs (Jardine, 2016a; Dolatian and Heinz, 2018). Those patterns, however, are out of the scope of this dissertation.

In Chapter 4, I discuss results of the tool-assisted learning experiments targeting various phonological processes such as tone plateauing, local processes,

and different types of harmony systems with and without blockers.

2.3 Learning grammars from data

Previous sections showed that subregular grammars can model natural language dependencies. However, all the previously presented grammars were constructed manually. In this section, I discuss the possibility of building those models *automatically*. It not only allows the researchers to avoid the burden of manual grammar construction, but also gives us insights into the mechanisms helping to discover those patterns.

The *grammatical inference* is a sub-field of machine learning that is concerned with the extraction of grammars from data. As Colin de la Higuera formulates in his book “Grammatical Inference” (2010), this field submerged at the intersection of linguistics, inductive analysis, and pattern recognition. *Linguistics*, and computational linguistics, in particular, brings the core idea of the existence of a *formal grammar*, or a set of rules defining a *language*. A *language learning* can then be viewed as a process of discovering a language’s grammar by the learner. The field of *inductive inference* aims at a problem of inferring the underlying grammar that consistently predicts what is grammatical and what is not after observing a set of elements of the language, where those elements can be strings, trees, or other structured objects. Finally, *pattern recognition* describes the best model and its properties that would explain the data; it *analyzes* the pattern.

If the task is to model a language, then the goal is to find a grammar that describes that language. If the grammar is not known a priori, it might be possible to *learn* its rules by observing and exploring the language. *Grammatical inference algorithms* require a finite sample of usually positive data drawn from the target language as input, and return a grammar hypothesis as output. That grammar solves a *membership problem* for that language, or, in other words, it correctly

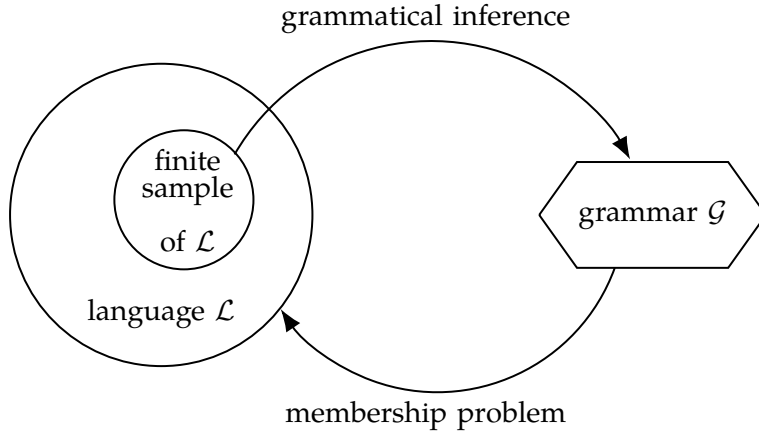


Figure 2.18: Relationship between a language \mathcal{L} and a grammar \mathcal{G} .


predicts for any given string if that string belongs to the target language, see Figure 2.18. If instead a mapping needs to be learned, grammatical inference algorithms require a sufficient sample of the input-output pairs as input and construct a transducer that generalizes that mapping.

The learning algorithms for SL, SP, TSL and MTSL languages, and also the algorithm inferring subsequential mappings, will be discussed in details in Chapters 3 and 4. These algorithms share several common properties, namely, they all require only positive data to find the grammar, they are fully interpretable, and work in polynomial time and data. Indeed, *learning only from the positive data* is the desired characteristic, since human learners do not have access to the information of what is not possible in their languages (Chomsky, 1986). Only some of the subregular languages have this property: the full class of regular languages cannot be learned from positive data. *Interpretability* of an algorithm means that both the learning process and the outcome are transparent: it is possible to trace how the learner came to a certain conclusion, and explain the obtained results. These learners extract grammars in *polynomial time*, so they can be computed in practice. (The running time of polynomial algorithms is n^c , where n is the size of the training sample, and c is some constant (Sipser, 2013).) Finally,

learning in the limit guarantees that after a finite number of errors, the learner will start making only correct predictions (Gold, 1967).

There were several successful applications of grammatical inference algorithms in the previous decades. For example, Alexander Clark won the Tenjinno competition in 2006 by using a modified version of OSTIA, a subsequential learner discussed further in Chapter 4 (Oncina et al., 1993; Clark, 2006). Also, see a paper by Chandlee et al. (2012) discussing the integration of FSA-based grammatical inference techniques into robotic planning.

However, the subregular learners are *structural* and not probabilistic, and therefore frequently, the absence of some particular configuration in the training sample results in the algorithm failing to learn simple patterns. For instance, Gildea and Jurafsky (1996) show that a corpus of English pronunciations is not enough for OSTIA to generalize a rule of English flapping.

As a part of my dissertation, I implemented the *SigmaPie* package  for working with subregular languages and mappings. It provides learners for SL, TSL, MTSL, SP languages and subsequential mappings (Aksënova, 2020b). In Chapter 3, I explore how well those subregular learners extract *well-formedness conditions* from artificial automatically generated datasets exhibiting human language-like patterns such as one or more harmonies with or without blockers, word-final devoicing, tone plateauing, and others. The training sample for those experiments is a collection of words well-formed according to one of those generalizations. Later in Chapter 4, I explore modeling of *processes* similar to the ones listed above. In this case, the training sample contains pairs of underlying representations and the corresponding surface forms. In my thesis, I model of processes and well-formedness conditions using tools implemented as a part of *SigmaPie*.

Chapter 3

Learning languages



The previous chapter of this dissertation showed that subregular languages are a good fit for phonology and morphology. Indeed, local processes can be expressed with strictly local (SL) grammars and long-distance processes with strictly piecewise (SP) grammars. If a long-distance process uses blockers (e.g., opaque vowels in vowel harmony), then we use tier-based strictly local (TSL) grammars. If the language shows multiple long-distance processes, then we may also need multiple tier-based strictly local (MTSL) grammars. This perspective on morphotactics and phonotactics also rules out typologically unattested patterns such as first-last harmony, majority harmony, or embedded circumfixation.

In this chapter, I discuss algorithms which extract SL, SP, TSL and MTSL grammars from the data. Apart from the training sample, these algorithms require knowing the class of the target grammar, and the locality window of that grammar, i.e. the length of substructures with which it operates. In other words, we need to know the formal complexity of the target language in order to learn it efficiently using subregular learning algorithms. Given a sufficient and representative training sample and proper specifications of the target grammar, the subregular learners discover the pattern in polynomial time and data.

I start by introducing the datasets that I will use throughout the chapter to

perform the learning experiments. These datasets vary from automatically generated artificial languages to real wordlists exemplifying concrete linguistic phenomena, such as word-final devoicing in German, vowel harmony in Turkish, and others. I then use these datasets as training samples for the subregular learners, discuss the obtained grammars, and automatically evaluate the predictions of those grammars based on the well-formedness of the strings they generate. At the end of the chapter, I provide a table that summarizes the results of the learning experiments.

The exemplified learners work exclusively with *string representations*. These algorithms focus on structural properties; they are limited to non-probabilistic algorithms which evaluate the well-formedness of input strings. As of now, I have not implemented statistical versions of these algorithms, or algorithms which work with non-string-based representations.

The subregular learners, scanners and sample generators are available as a module of my Python package *SigmaPie*  (Aksënova, 2020b). All of the code and datasets are available on GitHub  (Aksënova, 2020d).

3.1 The experimental setup

I use both artificial and natural languages to explore the performance of the SL, SP, TSL and MTSL learners. The artificial languages imitate concrete natural language phenomena. The learning of these artificial languages shows us if the extraction of those patterns is possible *conceptually*, whereas the performance of those algorithms on the natural language datasets shows us what is currently possible *in practice*. Due to the transparency and interpretability of the subregular learners, we can always see the path the learner took to extract the target grammar. If the learning experiment was unsuccessful, we can always look inside those algorithms and see what obstructed the convergence.




3.1.1 Experimental pipeline

The **experimental pipeline** involved 3 steps: learning, generation, and evaluation. **Learning** included automatic extraction of subregular grammars from the given training samples. I then used those extracted grammars to **generate** samples of strings.¹ Finally, during the **evaluation** step, I computed the percentage of the strings from the automatically generated datasets that are well-formed with respect to the target grammars.

Despite of what could be expected, I am *not* testing the performance of the grammars on the held out parts of the training sample to detect the cases in which the learner “overgeneralized” the pattern. For example, some learning experiments resulted in the learner incorrectly converging on the empty negative grammar: such a learner simply assumed that “anything goes”. Trivially, it will score 100% on all the held-out data. Only by looking at the predictions of the grammar it is possible to see if the learner generalized the language of the training sample.

The results showing the performance of the subregular models are presented in 3.6. Only negative grammars were employed for the experiments due to the succinctness of their grammars.

3.1.2 Natural languages


The real data used for the experiments comes from German, Finnish and Turkish. The German dataset is a wordlist for wordgames posted by enz  (Enzenberger, 2019). The Finnish data is taken from a collection of wordlists scraped by douglasbuzatto  (Buzatto, 2016). The collection of Turkish words was uploaded by Harrison et al. (2004) from Swarthmore College as a part of his project *The Vowel Harmony Calculator* . Due to the non-probabilistic nature of the

¹The functionality of subregular generators comes from the *SigmaPie* toolkit.

algorithms, I removed the data items that violate the target generalizations, such as disharmonic stems.

The target patterns exemplified by natural language datasets involved three levels of abstraction. The *raw representations* contained the strings from the wordlists; it allows us to explore the problems that the learner faces when it is given realistic data. The *masked representation* is more abstract and involved substituting all the symbols in the original data that are not relevant for the target pattern by a single symbol of choice. It allows the learner to focus on the generalization since all the “irrelevant” material is masked. It lets us explore if there is enough information in the simplified strings of the language to notice the pattern behind the behavior of the dependent elements. For example, if Turkish vowel harmony was explored under this perspective, all consonants were masked as x . Finally, the *abstract representation* represented the pattern with the highest degree of generality, therefore compelling the learner to discover only the “core” part of the generalization. This allows us to carefully examine the learning process: the levels of abstraction help remove the “concreteness” of the natural language dependencies and focus on the general properties of the target patterns.

3.1.3 Artificial languages

I used 5 artificial language generators for the experiments in this and the following chapters. Their code is available on GitHub  (Aks nova, 2020d). In all of these generators, the number of strings (the default value is 10) and the length of those strings (the default value is also 10) can be defined a priori.

The **Simple Harmony Generator** implements long-distance processes such as vowel harmony and consonant harmony. The generator lets us define several harmonic classes, where a *harmonic class* is a collection of elements that cannot co-occur within the same well-formed word of the language. For example, if there are two harmonic classes $A = \{a, o\}$ and $B = \{b, p\}$, the well-formed words of this

language can use at most 1 element of these class, unless a blocker intervenes. These classes A and B define strings such as *apappa*, *oppooo*, *bbaab*, and so on; but the ones such as *abbobb* cannot be produced by this generator. The blockers are defined as $\{f:a, s:p\}$, meaning that the occurrence of f in the string allows only a to be seen after itself. Other elements of a 's harmonic class are now prohibited, e.g., f blocks any subsequent o . Additionally, b is prohibited after an s . This generator now produces words such as *ababbsappp* and *obbofbasaapp*. Transparent elements can be expressed as single-item harmonic classes, such as $X = \{x\}$; it lets x be freely inserted in different parts of any string. Additionally, the minimal and maximal length of every harmonic class is also parametrized (the default values are $\min = 1$ and $\max = 3$), as well as the emission probability of every blocker (the default probability is $p(s) = p(f) = 0.2$).

The **Fake Turkish Generator** is much more specialized in comparison to the generator outlined above. It produces sequences of vowels that are well-formed with respect to the rules of Turkish vowel harmony, with all the consonants simplified to a single symbol. Like that, it produces harmonic sequences such as *öxüüxeei* and *oalxxxll*. The “choice” of the consonant, as well as the minimal and maximal lengths of vowel and consonant clusters, can be specified in the generator. The Turkish dataset cannot be defined by the Simple Harmony Generator because the same set of segments participates in two types of harmonies (backness and rounding), and some of the vowels that are undergoers for the backness harmony serve as blockers for the rounding one.

The **Word-Final Devoicing Generator** simply produces a set of strings that follow the rule of the word-final devoicing. For this, we define a list of voiced and voiceless segments, as well as the alphabet of the language. By default, the voiced and voiceless segments are $\{b\}$ and $\{p\}$ respectively, and the alphabet of the language is $\{a, b, p\}$. For example, this generator produces strings such as *apabaa* and *abbap*, but never the ones such as *paab*.

The **UTP Generator** produces strings of low (*L*) and high (*H*) tones that are well-formed concerning the unbounded tonal plateauing (UTP) generalization in Luganda. This generalization prohibits a low tone from appearing in a string if surrounded by high tones. For example, it produces strings such as *LLHH* and *LHHHL*, but is incapable of generating the ones such as *HHLHH* and *HLLHH*.

The **First-Last Harmony Generator** generates a language with first-last harmony. The language enforces agreement between the first and the last elements within the string. A list of agreeing elements needs to be specified (the default value is $\{a, o\}$), as well as the list of all other elements that can appear in the well-formed strings of this language ($\{a, o, x\}$ is the default). Strings such as *axoxoxa* and *ooaxo* are then grammatical, but *axaxxo* or *oaa* are not. Importantly, harmonic systems of this type are unattested in natural languages.

These generators were employed to produce data that was then fed to the SL, SP, TSL and MTSL learners. The next subsection provides the detailed description of the datasets used as training samples during the subregular experiments.


3.1.4 Target patterns

The learning experiments in this chapter target typologically widespread linguistic patterns such as word-final devoicing, tone plateauing and harmonic systems of different kinds. Here, I explain how I obtained the artificial training samples for every one of these patterns. For German, Finnish and Turkish natural language datasets, I also discuss the preprocessing steps that I did to get the data in the shape appropriate for the experiments.

Pattern 1: word-final devoicing

The phenomenon of word-final devoicing is attested in languages such as Russian and German. It prohibits underlyingly voiced obstruents from being pronounced voiced at the end of the word. In German, word-final */b/*, */d/*, and */g/* are

realized as [p], [t], and [k] (Brockhaus, 1995). For example, the word for ‘children’ is *Kinder*, but its singular form is pronounced as *Kin[t]*, i.e. the underlyingly voiced segment is realized as voiceless at the end of the word.

For this experiment, I used a German wordlist  (Enzenberger, 2019), its masked version, and an abstract representation of this pattern. During the masking step, all irrelevant segments were simply represented as *a*, and the abstract representation of this pattern only included 3 types of elements: voiced obstruents (*b*), voiceless obstruents (*p*), and others (*a*).

Raw representation This wordlist in its original form contains 685,618 words written in German orthography. However, German orthography does not reflect the process of the word-final devoicing and therefore the preprocessing of this corpus was necessary. It included two steps: incorporating the effect of the word-final devoicing and filtering words that contain non-German symbols. Firstly, I substituted every occurrence of the word-final /g/, /b/ and /d/ by their voiceless counterparts /k/, /p/, and /t/, respectively. In total, there were 1,599 words that end with /b/ (0.2% of the total number of words); 15,294 words that end with /d/ (2.2%); and 17,098 words with a word-final /g/ (2.4%). This step resulted in words such as *Kind* being changed to *Kint*, *Rad* to *Rat*, etc. Secondly, I excluded all strings that use letters that do not belong to the German alphabet, such as *złoty*, *château*, and some others. After those words were excluded, the size of the German wordlist became 685,147 words.

```
1 print(german_wfd)
2 # ['hochjagende', 'zugebliebener', 'verbricht', 'besuchszimmer', '
   beschneien', ...]
```

Masked representation The next step was to simplify German wordlist. Namely, for the phenomenon of the word-final devoicing, segments other than $\{b, p, g, k, d, t\}$ are not important, and therefore all of them can be simply masked

as *a*. The rules of the masking are summarized in table 3.1. Since none of the words were deleted during this step, the size of that sample was the same as before: 685,147 words.

$$\begin{aligned} b, g, d &\leftarrow b, g, d \\ p, k, t &\leftarrow p, k, t \\ a &\leftarrow a, \ddot{a}, c, e, f, h, i, j, l, m, n, o, \ddot{o}, q, r, s, u, \ddot{u}, v, w, x, y, z, \text{ß} \end{aligned}$$

Table 3.1: German: *raw* \rightarrow *masked* representation.

```
1 print(german_wfd_masked)
2 # ['aakaabaaaa', 'aakaabaaak', 'aakaa', 'aakat', 'aaa, ...']
```

Abstract representation Finally, the pattern can be simplified even further to only three classes of elements: voiced obstruents, voiceless obstruents, and items that are irrelevant for the word-final devoicing. Let us then refer to these classes as *b*, *p* and *a*, respectively. The rules of this simplification for the German alphabet are presented in table 3.2.

$$\begin{aligned} b &\leftarrow b, g, d \\ p &\leftarrow p, k, t \\ a &\leftarrow a, \ddot{a}, c, e, f, h, i, j, l, m, n, o, \ddot{o}, q, r, s, u, \ddot{u}, v, w, x, y, z, \text{ß} \end{aligned}$$

Table 3.2: German: *raw* \rightarrow *abstract* representation.

To generate such a pattern, I used the word-final devoicing generator previously discussed in section 3.1.3. Like this, I obtained a sample of 1,000 strings that represent the target pattern abstractly. The length of every word of this sample is 10 symbols.

```
1 toy_wfd = generate_wfd(n = 1000)
2 print(toy_wfd)
```



```
3 # ['aaabbbppbbp', 'pbapbapapa', 'apabaappap', 'bbbbbaabbbp', '
    pbbpabppap', ...]
```

Pattern 2: a single vowel harmony without blocking

The next targeted phenomenon was a simple case of a vowel harmony that does not exhibit a blocking effect. In Finnish, vowels can be sub-divided into 3 categories: front (*ä, ö, y*), back (*a, o, u*) and neutral (*e, i*). Vowels within a word must agree with respect to their fronting or backness features, and the initial vowel controls the spreading (Rose and Walker, 2011). The neutral vowels are transparent regarding the harmonic process, and therefore they can occur in both types of words. For example, a word *puisto* ‘park’ contains back and neutral vowels, whereas *ikänttyvien* ‘older’ contains front and neutral ones.

For this experiment, I used a Finnish wordlist (Buzatto, 2016), its masked version, and a simplified abstract representation of this pattern. The masked representation of the Finnish sample included masking all elements transparent for the harmony as *x*, and the alphabet of the abstract pattern only included 3 elements: vowels of one harmonic class (*a*), vowels of another harmonic class (*o*), and transparent elements (*x*).

Raw representation Originally, the Finnish wordlist contained 287,699 words. Three preprocessing steps were necessary: removing the words with non-Finnish letters from the sample, filtering the disharmonic words, and making changes to the representation of some letters. Firstly, I eliminated words that contain symbols that are not included in the Finnish alphabet, such as digits and punctuations. I also removed words such as *långsamt* ‘slowly’, that are in fact Swedish and therefore use the Swedish letter *å* represented as } in this dataset. The behavior of such words is not clear regarding Finnish vowel harmony. A total of 331 such words were removed from the dataset (0.1% of all words). Then, I substituted {

and | that stand in this wordlist for *ä* and *ö*, respectively, by their more legible counterparts *A* and *O*. Finally, I filtered the disharmonic stems such as *etukäteen* ‘in advance’ and *juhlapäivä* ‘holiday’, there were 36,563 of such words in total (12,7%). 250,805 words remained after the preprocessing steps were completed.

```
1 print(finnish_harmony)
2 # ['mathilden', 'lisAnimen', 'macmillanin', 'urquhartin', '
   ilmeneviksi', ...]
```

Masked representation Next, I created a dataset where every segment transparent for the Finnish vowel harmony was masked. The vowels {*ä*, *ö*, *y*, *a*, *o*, *u*} were left intact, whereas the other elements were rewritten as *x*, see the table 3.3. The obtained dataset also contains 250,805 words.

$$\begin{aligned} \text{ä, ö, y} &\leftarrow \text{ä, ö, y} \\ \text{a, o, u} &\leftarrow \text{a, o, u} \\ \text{x} &\leftarrow \text{b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s, t, v, w, x, z} \end{aligned}$$

Table 3.3: Finnish: *raw* \rightarrow *masked* representation.

```
1 print(finnish_harmony_simplified)
2 # ['xaxxixxex', 'xixAxixex', 'xaxxixxaxix', 'uxxuxaxxix', '
   ixxexexixxi', ...]
```

Abstract representation Lastly, I simplified the pattern even more by generalizing the harmonic and transparent elements of Finnish to three classes: *a* for the class of front vowels, *o* for the class of back vowels², and *x* for all other elements that make this dependency long-distant, see table 3.4.

²Although *a* and *o* are both back vowels, this is just an abstraction to keep the alphabet as simple as possible.

a	←	ä, ö, y
o	←	a, o, u
x	←	b, c, d, e, f, g, h, i, j, k, l, m, n, p, q, r, s, t, v, w, x, z

Table 3.4: Finnish: *raw* \rightarrow *abstract* representation.

To generate artificial data, I used the harmonic generator discussed in 3.1.3. I defined a single harmonic class $A = \{a, o\}$ with $X = \{x\}$ representing the transparent elements. Any word was able to contain x , however, a and o could not co-occur within the same word. I generated a sample of 1,000 words that were well-formed regarding the rules of this simplified vowel harmony.

```

1 generator = Harmony({"a", "o": "A", ("x"): "X"})
2 toy_vhnb = generator.generate_words(n = 1000)
3 print(toy_vhnb)
4 # ['oxxxooxxxx', 'ooxxxooxxo', 'aaxxxaaxxx', 'oxxxxoxxxo', '
   xxxaaxxaxx', ...]
```

Pattern 3: a single vowel harmony with blocking

The previous example concerns the case when there is a single vowel harmony that does not exhibit a blocking effect. However, blocking is a frequent phenomenon in harmonic systems. Consider Assamese, where vowels regressively harmonize in the advanced tongue root (ATR) feature (Mahanta, 2007). In the word *polox* ‘silt’ all vowels are lax, however, when the tense suffix *-uwe* is applied to that stem, all the vowels become tense: *poloxuwe* ‘fertile land’. Nasals, as well as some other segments, block this long-distance spreading. In *zomoni* ‘humorous’, lax vowels precede the nasal *n*, whereas the tense one follows it. Unfortunately, I found no available dataset exhibiting such a harmony. However, patterns of this type are widespread among the languages of the world, so in this experiment, I model the blocking effect.

Abstract representation In the harmonic system without blockers, there were 3 classes of elements: undergoers expressing one value of the harmonic feature (*a*), undergoers expressing the other value (*o*), and segments irrelevant for the harmony (*x*) that are present in the abstract representation just to make the dependency long-distant. Now, let us model *blockers* that enforce some particular value of the harmonic feature further in the string. Continuing the previous example, let us introduce a blocker *f* that only allows for *a* to be seen after itself thus allowing for well-formed sequences such as *ooxoxofxaxa* and *aaxaxaffaaa*.

a ← $[+\alpha]$ vowels
o ← $[-\alpha]$ vowels
x ← transparent elements
f ← blockers enforcing $[+\alpha]$ specification

Table 3.5: A harmony in $[\alpha]$ exhibiting blocking effect; *abstract* representation.

Strings representing this pattern are automatically produced by the harmonic generator. Its setup is similar to the one discussed in the previous experiment, but also includes the definition of the blocker *f* that prohibits *o* after itself, i.e. only *a* can be seen further in the string. The blocker together with the value that it licenses must be provided to the generator as the parameter *blocker*. In this case, this parameter is set to $\{f:a\}$. I generated 1,000 strings that follow the target pattern.

```

1 generator = Harmony({("a", "o"): "A", ("x"): "X"}, blockers = {"f": "a"
   })
2 toy_vhwb = generator.generate_words(n = 1000)
3 print(toy_vhwb)
4 # ['oxxxooxxxx', 'xxooxfaaax', 'aaxxxaaxxx', 'ofxxafxxaa', '
   aafaaaxaaf', ...]
```

Pattern 4: several vowel harmonies without blocking

In some languages, harmonic systems involve the spreading of more than one feature. For example, in Kirghiz, vowels agree in fronting and rounding (Nanaev, 1950; Kaun, 1995). All vowels within a word can be of 4 types: back and unrounded (*kiz-da* ‘girl-LOC’), back and rounded (*ot-to* ‘fire-LOC’), fronted and unrounded (*kim-de* ‘who-LOC’), and fronted and rounded (*üj-dö* ‘house-LOC’). Modeling this type of harmony involves increasing the inventory of the harmonic class: now there are 4 options of feature specifications that are available for vowels within words.

Abstract representation To have an abstract picture of this pattern, let us assume that we are dealing with the long-distant vowel agreement in features $[\alpha]$ and $[\beta]$. Then it is possible to generalize $[-\alpha, -\beta]$ vowels as *a*, $[-\alpha, +\beta]$ ones as *e*, $[+\alpha, -\beta]$ ones as *o*, and, finally, $[+\alpha, +\beta]$ vowels as *u*. I will again use *x* as a transparent element. This abstract harmony will enforce all vowels within the same word to agree in both features $[\alpha]$ and $[\beta]$: as the result, only one element out of the set $\{a, e, o, u\}$ can appear in the well-formed words of such language.

a	←	$[-\alpha, -\beta]$ vowels
e	←	$[-\alpha, +\beta]$ vowels
o	←	$[+\alpha, -\beta]$ vowels
u	←	$[+\alpha, +\beta]$ vowels
x	←	transparent elements

Table 3.6: A harmony in $[\alpha]$ and $[\beta]$; *abstract* representation.


Such a harmonic pattern can be encoded by extending the harmonic class of the generator: now, instead of two elements, it includes four. 1,000 strings of such language were generated.

```

1 generator = Harmony({"a", "o", "e", "u"): "A", ("x"): "X"})
2 toy_mhnb = generator.generate_words(n = 1000)
3 print(toy_mhnb)
4 # ['xuuxxxuuu', 'xxeeexeee', 'xxaaxxaa', 'xooxooxox', ...]

```

Pattern 5: several vowel harmonies with blocking

In this experiment, I target another type of a harmonic system spreading several features, but in this case, it also involves the blocking effect. For example, in Turkish, vowel harmony enforces vowels to agree in backness and rounding. For backness, all vowels within a word need to agree in this feature. However for roundness, only high vowels acquire the rounding value of the previous vowel; therefore, the non-high vowels are always realized unrounded in the non-initial syllables (Levi, 2001; Krämer, 2003). For example, the word *son-lar-un* ‘end-PL-GEN’ exemplifies that a non-high vowel from the plural suffix cannot acquire a rounding feature from the previous vowel, and therefore cannot further transmit it to the following high vowel. However, in *son-un* ‘end-GEN’, the high vowel is realized rounded because it is preceded by a rounded vowel. In both words, all vowels agree in backness. In such a system, non-high vowels have a double nature: they are undergoers for the backness harmony, however, they are blockers for the rounding one. Thus to test the performance of subregular models with this complex type of harmonic system, I will be using the Turkish wordlist  (Harrison et al., 2004).

I start by preprocessing the Turkish corpus by eliminating the words that contain non-Turkish characters and filtering the disharmonic stems. Then I generalize the pattern by masking all the consonants since they are irrelevant for the harmonic pattern³. Finally, I generate an artificial dataset exhibiting Turkish harmony.

³I ignore the cases when stem-final palatalized λ starts its backness harmony domain

Raw representation The wordlist that I use contains 23,501 lexemes. However, it also required several preprocessing steps. Firstly, I eliminated all words that used non-Turkish characters and nonce-words, such as *bungalow* and *lx*; there were 890 of such words (3.8% of total words). Secondly, I removed the stems that violate the rules for backness and rounding harmony (such as *koreografi* and *kümülatif*). Since Turkish vowel harmony is frequently violated (Pöchrager, 2010)⁴, there were 10,545 such words (44.9%). After those forms were eliminated, the corpus contained 14,434 Turkish harmonizing words.

```
1 print(turkish_harmony)
2 # ['som', 'lafazan', 'konuk', 'kekti', 'lafzan', ...]
```

Masked representation At the next step, I masked all the symbols that are not relevant to the Turkish harmony. Namely, all the consonants are transparent, and therefore were substituted as *x*, whereas the vowels {a, e, o, ö, u, ü, i, ı} were left intact, see table 3.7. The obtained dataset contains 14,434 words as well.

$$\begin{aligned} a, e, o, \ddot{o}, u, \ddot{u}, i, \ddot{i} &\leftarrow a, e, o, \ddot{o}, u, \ddot{u}, i, \ddot{i} \\ x &\leftarrow \text{ç, ğ, ş, b, c, d, f, g, h, j, k, l, m, n, p, r, s, t, v, y, z} \end{aligned}$$

Table 3.7: Turkish: *raw* \rightarrow *masked* & *abstract* representations.

```
1 print(turkish_harmony_simplified)
2 # ['xixxix', 'xuxx', 'xaaxa', 'xexxe', 'xaxIx', ...]
```

Abstract representation The vowel inventory of Turkish cannot be further simplified: all 3 features – backness, rounding, and height – are important for the choice of the following vowel, and this yields exactly $2^3 = 8$ vowels. However, it is highly likely that the Turkish data contains “accidental” gaps: some subregular

⁴It is hard to imagine a better name for a paper about Turkish disharmony than Pöchrager’s *Does Turkish diss harmony?*

learners require to observe all symbols of the alphabet adjacent to each other to make a decision, but this is rarely the case with natural language data.

I implemented a generator of fake Turkish, that imitates Turkish vowel sequences and uses *x* as a transparent element. In this way, it is possible to be sure that given enough data, the learner will always observe all the combinations of elements that it needs. In this case, 1,000 strings were not enough since the alphabet of the artificial language is larger in comparison to the previous cases, and the generalization is more complex, therefore I used a wordlist of 15,000 fake Turkish words.

```
1 print(toy_mhwb)
2 # ['xx0exxix', 'xxUUxUUx', 'exxiixee', 'iiexxxex', 'xuuxxuux', ...]
```

Pattern 6: vowel harmony and consonant harmony without blocking

There are also typologically attested cases where there are several independent harmonies within the same language. For instance, in several Bantu languages (Kikongo, Kiyaka, Bukusu a.o.), a vowel harmony co-occurs with long-distance consonant assimilation. As a case study, consider Bukusu, where vowels agree in height, and /l/ assimilates to /r/ if it is preceded by /r/ within the same word (Odden, 1994; Hansson, 2010a). This can be demonstrated by a causative affix that includes both a high vowel and a liquid. As a result, the affix /il/ can be realized in four different ways: *il*, *ir*, *el*, and *er*. In *teex-el* ‘cook-APPL’, the affixal vowel is low due to the low vowel in the stem, and the liquid surfaces as /l/; the version of this affix with a high vowel is *lim-il* ‘cultivate-APPL’. However, if the rhotic liquid precedes the affix, it is realized with /r/: *reeb-er* ‘ask-APPL’ and *rum-ir* ‘send-APPL’. To model this pattern, we need to imitate two harmonies: one affects vowels, and another one targets the consonants.

Abstract representation The abstract representation of such harmonic pattern exhibits two harmonic classes, one of them includes vowels $\{a, o\}$, and another one includes consonants $\{b, p\}$, see table 3.8. This language then allows for the words such as *poppooo* and *aaabba*, but not for the ones such as *babboo* or *opobp*. As previously, there are no transparent elements because vowels make the consonant harmony long-distant, and vice versa.

a	←	$[-\alpha]$ vowels
o	←	$[+\alpha]$ vowels
b	←	$[-\beta]$ consonants
p	←	$[+\beta]$ consonants

Table 3.8: A harmony in $[\alpha]$ and a harmony in $[\beta]$; *abstract* representation.

We can encode this pattern by increasing the number of harmonic classes. Now, the harmonic class $A = \{a, o\}$ represents vowel harmony, and $B = \{b, p\}$ depicts the consonant one. Given these specifications, I generated a sample of 1,000 words of such abstract harmonic language.

```

1 generator = Harmony({("a", "o"): "A", ("b", "p"): "B"})
2 toy_dhnb = generator.generate_words(n = 1000)
3 print(toy_dhnb)
4 # ['bbbbaabbbaa', 'bbooboboob', 'pppooppop', 'appaaappaa', ...]
```

Pattern 7: vowel harmony and consonant harmony with blocking

We can also imagine a harmony that affects vowels and consonants, and additionally exhibits a blocking effect. Although such a pattern, to the best of my knowledge, is unattested, it is a typologically plausible one.

Abstract representation Let us build on the harmonic system discussed before, and add one modification to it: now, the consonant harmony can be blocked. The

blocker is then represented as t , and it prohibits b to be seen after itself. This blocker will not affect the simultaneous vowel harmony in any way, see table 3.9. Words such as *abbabab*, *abataapap* and *oobobtoop* are well-formed regarding the rules of this harmony, but *ababtab* or *obbotppaap* are not.

a	←	$[-\alpha]$ vowels
o	←	$[\alpha]$ vowels
b	←	$[-\beta]$ consonants
p	←	$[\beta]$ consonants
t	←	blocks spreading of $[-\beta]$

Table 3.9: A harmony in $[\alpha]$ and a harmony in $[\beta]$ with blockers; *abstract* representation.

Again, it is possible to employ the harmonic generator to generate this language. The setup is similar to the one discussed in the previous subsection, with the blocker $\{t:p\}$ introduced as well. The generated dataset contains 1,000 strings of this language.

```

1 generator = Harmony({("a", "o"):"A", ("b", "p"):"B"}, blockers = {"t
    ":"p"})
2 toy_dhwb = generator.generate_words(n = 1000)
3 print(toy_dhwb)
4 # ['pppoootopt', 'obbbtpooot', 'aabbbaatat', 'pppaapappp', '
    bbbtpppaap', ...]
```

Pattern 8: unbounded tone plateauing

Next, I will target the phenomenon of *unbounded tone plateauing* (UTP). This pattern is observed in some Niger-Congo languages such as Luganda, where all low tones (L) are realized as high (H) if they are surrounded by high tones. For example, tonal sequences such as *LLHH*, *HHLLL* and *LLHL* are well-formed, whereas the ones such

as *HLLLH* are not (Hyman, 2011; Jardine, 2016a). The learner would then need to induce that after observing a high tone followed by a low tone, the appearance of another high tone is impossible.

Abstract representation Such a pattern required implementing a separate generator. The output of that generator is a sample of well-formed sequences of high and low tones in languages such as Luganda. For the subregular experiment, I used a wordlist of 1,000 such sequences.

```
1 toy_utp = generate_utp_strings(n = 1000)
2 print(toy_utp)
3 # ['HHHHH', 'LHLL', 'LHHHH', 'LHHHH', 'HHHHH', ...]
```

Pattern 9: first-last harmony

The final challenge for the subregular learners is the one they shall *not* meet: learning a language that is neither SL, nor SP, TSL or MTSL. As an example of one, I will be using the *first-last harmony* pattern discussed by Lai (2015). The artificial pattern of the first-last harmony requires that the first element of the word to agree with the last one, therefore considering words such as *aaooxoxoa* and *oxoaxo* well-formed, and rejecting ones such as *oxxoa*. This language requires a generator that handles more complicated dependencies than SL, SP, TSL, and MTSL grammars can express, and therefore learners for those classes are expected to fail to generalize patterns such as first-last harmony.

Abstract representation I used another generator to obtain a sample of this unattested language. Its alphabet included symbols *a*, *o*, and *x*, and the generalization was simplified to *words can either start and end with a, or with o, but cannot have the initial and final symbols different*. A first-last harmony generator produced a sample of 5,000 strings that I used to show that subregular learners indeed cannot capture this pattern.

Target patterns	SP	SL	TSL	MTSL
<i>word-final devoicing</i>	✗	👍	👍	👍
<i>a single vowel harmony without blocking</i>	👍	✗	👍	👍
<i>a single vowel harmony with blocking</i>	✗	✗	👍	👍
<i>several vowel harmonies without blocking</i>	👍	✗	👍	👍
<i>several vowel harmonies with blocking</i>	✗	✗	👍	👍
<i>vowel harmony and consonant harmony without blocking</i>	👍	✗	✗	👍
<i>vowel harmony and consonant harmony with blocking</i>	✗	✗	✗	👍
<i>unbounded tone plateauing</i>	👍	✗	✗	✗
<i>first-last harmony</i>	✗	✗	✗	✗

Table 3.10: The expected results of the language learning experiments.

```

1 first_last_data = first_last_words(n = 5000)
2 print(first_last_data)
3 # ['axoaxaxaa', 'aaxaaxxxoa', 'ooaxaoaooo', 'axxoaxaaaa', '
   oxaoaxxxao', ...]

```

Expected results

In this section, I discuss the 9 types of patterns that I modeled using automatically learned subregular grammars. This list exhausts the options for the modeling possibilities by SP, SL, TSL and MTSL grammars: every one of those phenomena can be represented by a different combination of those four subregular classes. For example, *word-final devoicing* can be modeled by SL, TSL, and MTSL grammars; *a single vowel harmony without blocking* can be expressed via SP, TSL, and MTSL grammars; *tone plateauing* can only be generalized as SP, *first-last harmony* cannot be modeled by either of them, etc. The list of patterns and the corresponding grammars can be found in table 3.10.

3.2 Strictly local models

Strictly local grammars express local generalizations by listing substrings that *cannot* be present in well-formed words of their languages. In linguistics, they are employed to model local dependencies such as intervocalic voicing, consonant cluster assimilation, or others. In this section, I show that the SL inference algorithm is capable of automatically extracting the pattern of *word-final devoicing* from the German dataset. SL grammars cannot handle long-distance dependencies, and therefore their performance on harmonic datasets is extremely poor.

3.2.1 SL learning algorithm

The **intuition** behind the learning algorithm is that it simply assumes that every k -gram that was not observed in the training sample is prohibited. The alphabet of the SL grammar and its locality window need to be defined a priori. As a pre-processing step, all words of the training sample are annotated with the start- and end-markers \bowtie and \bowtie ; this allows to distinguish word-initial and word-final positions from any other position in the string.⁵ The learner records all k -grams that are attested in the input data, therefore, constructing the *positive* SL grammar. The *negative* SL grammar then lists all the unattested k -grams. For example, if the alphabet is $\{a, b\}$, the locality of the grammar is 2, and the observed bigrams are $P = \{\bowtie a, a \bowtie, ab, ba\}$, the negative 2-local SL grammar is $R = \{\bowtie \bowtie, \bowtie b, b \bowtie, aa, bb\}$.

The **pseudocode** of the k -SL learner can be found below. I is the training sample, i.e. it contains the collection of the well-formed strings of the language. $|w|$ refers to the length of the word w , $w[i]$ targets the i th character of the string w , and \cdot is a concatenation operator.

⁵Grammars can be represented as 3 sets: a set of n -grams that can occur word-initially, a set of n -grams that can appear word-internally, and a set of n -grams that can be word-final (Heinz, 2010a).

Algorithm 1 Extracts G_{SL_k} from I

```
 $G \leftarrow \emptyset$ 
for  $w$  in  $I$  do
   $w \leftarrow \bowtie \cdot w \cdot \bowtie$ 
  if  $|w| \geq k$  then
    for  $i$  in  $k \dots |w|$  do
       $G \leftarrow G \cup \{w[i-k] \dots w[i]\}$ 
    end for
  end if
end for
```

This algorithm extracts a positive grammar from the data. The equivalent negative grammar is then obtained by simply subtracting the set of allowed k -grams from a list of all possible k -grams that can be built from the grammar's alphabet Σ . The positive SL grammar and its equivalent negative SL grammar recognize the same language: $L(NEG_G_{SL_k}) = L(\Sigma^k \cap POS_G_{SL_k})$, where Σ^k refers to all k -long strings that can be generated based on the elements of Σ .

3.2.2 Successful experiments

Strictly local grammars are capable of expressing local dependencies, and therefore the only experiment in which the SL learner performed very well (100%) was for word-final devoicing. The runners-up were some cases of artificially generated datasets of simple vowel harmonies (83 ~ 89%) and tone plateauing (85%), but these numbers are unsurprising given that the alphabets of those grammars are very small and the generated strings are usually not very long; hence, the chance of “guessing” a grammatical word is significant. The decent performance of the SL grammars on artificial harmonic datasets comes from the shape of the generated data itself, giving insight into the additional parameters

that need to be controlled for in the subsequent experiments. The performance of the SL learners is extremely bad (30 ~ 70%) on more complex cases of harmonic systems, especially the ones that involve several harmonies.

Experiment 1: word-final devoicing This experiment involved testing the learner using three types of training samples: artificially generated, masked German, and raw German data. Since the pattern has a local nature, the performance of the learner in all of these cases was 100%.

```
1 s1 = SL(polar = "n")
2 s1.data = toy_wfd
3 s1.extract_alphabet()
4 s1.learn()
```

The first line initializes a negative SL grammar `s1`. The locality of the grammar is not specified, and therefore by default it is set to 2. Then, the artificial dataset for word-final devoicing `toy_wfd` is passed into the `data` attribute of the class `s1`. To avoid the burden of listing the elements of the alphabet manually, I am using the function `extract_alphabet` that does it automatically. Finally, the last line invokes the learning algorithm.

```
1 print(s1.alphabet)
2 # ['a', 'b', 'p']
3 print(s1.grammar)
4 # [('b', '<'), ('>', '<')]
```

The automatically extracted alphabet is $\Sigma = \{a, b, p\}$, and the set of unattested bigrams is $R = \{b\ltimes, \ltimes\ltimes\}$. The learner indeed saw that it is impossible to have a voiced obstruent b in a word-final position. Additionally, it did not observe an empty string in the training sample, and therefore assumed that it needs to be ruled out as well.

```
1 sample = s1.generate_sample(n = 1000)
2 print(sample)
```

```
3 # ['pbpababa', 'apaapa', 'ap', 'bbp', 'bbbabpbbp', ...]
```

After the grammar was extracted, I generated a sample of well-formed strings with respect to the rules of the learned grammar using the function `generate_sample`. The parameter `n` indicates the number of words that need to be generated.

```
1 evaluate_wfd_words(sample)
2 # Percentage of well-formed words: 100%.
```

Then I used an evaluative function `evaluate_wfd_words` to compute the percentage of words generated by `s1` that comply with the rule of the word-final devoicing. Indeed, none of the generated words violated the target pattern. Notice how it was possible to “look inside” the algorithm and explore the exact generalizations that it made due to the *interpretability* of the subregular grammars.

Pattern	<i>word-final devoicing</i>
Type of data	artificial data
Example of data	aaabbbpbbbp, pbapbapapa, apabaappap, bbbbaabbbp, ...
Learned 2-SL grammar	$b\bowtie, \bowtie\bowtie$
Generated sample	pbpababa, apaapa, ap, bbp, bbbabpbbp, ...
Evaluation	<code>evaluate_wfd_words(sample)</code>
Score	100%

Table 3.11: SL learning of the word-final devoicing; abstract representation.

I used a similar setup for all of the consequent experiments in what follows. From now on, for a more succinct representation, I will represent the experimental setup as a table.

Now, let us explore the performance of the SL learner using the masked German dataset. From the corpus of masked German words, the SL grammar again extracted the correct rules: none of the voiced obstruents $/b/$, $/d/$, and $/g/$

can appear at the end of the well-formed words. As before, all of the words generated by this grammar are well-formed.

Pattern	<i>word-final devoicing</i>
Type of data	masked German data
Example of data	aakaabaaaa, aakaabaaak, aakaa, aakat, aaa, ...
Learned 2-SL grammar	$b\bowtie, d\bowtie, g\bowtie, \bowtie\bowtie$
Generated sample	btddta, gpttk, batktbtgbktbba, gatdkbgkgdp, ...
Evaluation	<code>evaluate_wfd_words(sample)</code>
Score	100%

Table 3.12: SL learning of the word-final devoicing; masked representation.

This learner was also able to extract the correct rule from the German dataset. But in this case, the size of the learned grammar was very large: 109 bigrams. In other words, 109 bigrams are unattested in the German wordlist. Indeed, among others, the learner extracted the bigrams representing the word-final devoicing, namely $*b\bowtie$, $*g\bowtie$, and $*k\bowtie$. Apart from the target grammar, the learner induced lots of other unattested restrictions such as $*cj$, $*d\beta$, $*\ddot{a}\ddot{a}$, $*\ddot{u}\ddot{z}$ and so on.

Pattern	<i>word-final devoicing</i>
Type of data	raw German data
Example of data	hochjagende, zugebliebener, verbricht, besuchszimmer, ...
Learned 2-SL grammar	$b\bowtie, cj, cv, cw, cx, \dots$
Generated sample	piüüdki ζ fhr, nzjbpcböpfbniabga...
Evaluation	<code>evaluate_wfd_words(sample)</code>
Score	100%

Table 3.13: SL learning of the word-final devoicing; raw representation.

We could also explore the dataset generated by the learned grammar. It includes words such as *piüüdkizfhr* and *nzjbpcböpfbniabga*. We can increase the locality of the grammar, and learn the prohibited trigrams. The 3-SL grammar generates words such as *känckhacix* and *flaw*. Finally, the 4-local grammar generates words that look more “German”, such as *Eipotfeigsucktbohnt* and *luxmetie*. What is the most important for the current experiment, is that all of the words generated by this grammar follow the rule of the word-final devoicing. We can conclude that SL grammars are capable of learning this pattern even when facing raw data as the training sample.

3.2.3 Unsuccessful experiments

All conducted experiments show that SL grammars were only able to successfully extract the pattern of word-final devoicing. In this subsection, I list the results of the experiments that showed negative results and numerically evaluate the learning outcomes.

Experiment 2: a single vowel harmony without blocking SL grammars are only suited to model local dependencies, and therefore the SL learner is not expected to generalize a long-distant pattern like vowel harmony. Given an artificial dataset, the grammar indeed learned that vowels *a* and *o* cannot be adjacent to each other *locally*, so it is never the case in the strings generated by the extracted grammar. But it does not see a problem with disagreeing vowels at a distance from each other. Therefore in the output of the generator, we see ungrammatical strings such as *xoxaxxxaa* and *oxaa*. The performance of the SL model on this task is 83%, mostly due to the learned local generalization and the fact that the majority of the generated strings are pretty short.

The grammar performed worse on the masked Finnish corpus, because only 72% of the generated words were grammatical. As before, it succeeded in learning

Pattern	<i>one vowel harmony, no blockers</i>
Type of data	artificial data
Example of data	oxxxoxxxxx, oxxxoxxxo, aaxxxaaxxx, oxxxoxxxo, ...
Learned 2-SL grammar	ao, oa, x x
Generated sample	xoxxxxxaa, oo, oxaa, x, xo, ...
Evaluation	harmonic_evaluator(sample, single_harmony_no_blockers)
Score	83%

Table 3.14: SL learning of a single harmony without blockers; abstract representation.

the local version of the restriction. However, it could not generalize it thus predicting the well-formedness of words such as *aooooooooäyyöö*. An SL grammar trained on the raw Finnish data performed much worse: only 41% words that were predicted to be well-formed by this grammar, were, in fact, well-formed. I omit the table with those results.

Pattern	<i>one vowel harmony, no blockers</i>
Type of data	masked Finnish data
Example of data	xaxxixxex, xixäxixex, xaxxixxaxix, uxxuxaxxix, ...
Learned 2-SL grammar	äa, äo, äu, öa, öo, ...
Generated sample	yxy, yö, uuuux, oaa, aooooooooäyyöö, ...
Evaluation	harmonic_evaluator(sample, front_harmony)
Score	72%

Table 3.15: SL learning of a single harmony without blockers; masked representation.

Experiment 3: a single vowel harmony with blocking Adding a blocker to the vowel harmony pattern resulted, as previously, in the learner capturing the local generalization, but failing to generalize it to the long-distant pattern. The performance of the model is 89%. But again, this is mostly due to the small size of the alphabet and the short average length of the generated strings; thus, this increased the chances of getting a harmonizing word simply by chance.

Pattern	<i>one vowel harmony with blockers</i>
Type of data	artificial data
Example of data	oxxxooxxxx, xxooxfaaax, aaxxxaaxxx, ofxxafxxaa, ...
Learned 2-SL grammar	ao, fo, oa, ××
Generated sample	x, fafafxfa, axooxo , ox, x, ...
Evaluation	harmonic_evaluator(sample, single_harmony_with_blockers)
Score	89%

Table 3.16: SL learning of a single harmony with blockers; abstract representation.

Experiment 4: several vowel harmonies without blocking Similarly to the cases before, when challenged with several vowel harmonies without a blocking effect, the SL learner only captured the local version of the generalization. But in this case, the performance of the learner is worse than with the previously discussed artificial grammars: indeed, there are now 4 choices of vowels instead of 2, and only one of them can be chosen and used throughout the word. It is now harder to get the harmonizing words “by chance”, and therefore such a model has a performance of only 69%.

Experiment 5: several vowel harmonies with blocking Expectedly, SL grammars performed even worse when trying to learn a pattern of several long-distance assimilation among vowels, where some of these assimilations exhibit blocking effect. The learner predicted the correct backness harmony in

Pattern	<i>several vowel harmonies, no blockers</i>
Type of data	artificial data
Example of data	xuuuxxxuuu, xxxeeexeee, xxxaaxxxaa, xooxxooxox, ...
Learned 2-SL grammar	ae, ao, ua, ue, uo, ...
Generated sample	xaaaa, u, oxuuxeexux , a, aaxxu, ...
Evaluation	harmonic_evaluator(sample, double_harmony)
Score	69%

Table 3.17: SL learning of several vowel harmonies without blockers; abstract representation.

67% of the words, and the rounding harmony was correct in 70%. However, overall, only 59% of the words predicted by the grammar were, in fact, grammatical with respect to the rules of the Turkish harmonic system.

Pattern	<i>several vowel harmonies with blockers</i>
Type of data	artificial data
Example of data	xxöexxix, xxüüxüüx, exxiixee, iiexxxex, xuuxxuüu, ...
Learned 2-SL grammar	iö, iü, ie, ii, io, ...
Generated sample	öxi, oaxüüee , iaia, uaiaaax, ü, ...
Evaluation	harmonic_evaluator(sample, backness_and_rounding)
Score	59% overall (67% backness only, 70% rounding only)

Table 3.18: SL learning of several harmonies with blockers; abstract representation.

When given a masked Turkish dataset as input, the accuracy of the learner increased to 70%. In the future, I would be interested in understanding why the learner performs on masked Turkish words better than on the artificially generated language. Only 30% of the predicted words were grammatical when trained on the actual Turkish data.

Experiment 6: vowel harmony and consonant harmony without blocking This experiment tests the SL learner on data showcasing two harmonies that affect different sets of segments. The learner is not successful since the elements participating in one harmony make the other harmony non-local, and vice versa. As a result, it predicts the generated words correctly only in 64% of the cases.

Pattern	<i>vowel and consonant harmonies, no blockers</i>
Type of data	artificial data
Example of data	bbbaaabbaa, bbooboboob, pppoopppop, appaaappaa, ...
Learned 2-SL grammar	ao, oa, bp, pb, $\times \times$
Generated sample	b, p, ppp, poobap , obbboboopp , ...
Evaluation	harmonic_evaluator(sample, double_harmony_no_blockers)
Score	64%

Table 3.19: SL learning of vowel and consonant harmonies without blockers; abstract representation.

Experiment 7: vowel harmony and consonant harmony with blocking The performance of the SL learner on the dataset exhibiting vowel and consonant harmonies with blocking effect is not different from the one before: only 64% of the words generated by the grammar are well-formed.

Experiment 8: unbounded tone plateauing The phenomenon of UTP prohibits the occurrence of low tones in-between high tones. Because this process looks at minimally 3 segments at any time, I use a window of $k = 3$. However, it does not learn this pattern. Indeed, it learns it locally, but as before, it does not induce its long-distant nature. The performance of the generator is 85%, but it is mostly due to frequently occurring short words and a small alphabet that contains only two elements.

Pattern	<i>vowel and consonant harmonies with blockers</i>
Type of data	artificial data
Example of data	pppoootopt, obbbtpooot, aabbbaatat, pppaapappp, ...
Learned 2-SL grammar	ao, oa, bp, pb, tb, $\times \times$
Generated sample	p, oobobbbaattapapot, a, aa, ota, ...
Evaluation	harmonic_evaluator(sample, double_harmony_with_blockers)
Score	64%

Table 3.20: SL learning of vowel and consonant harmonies with blockers; abstract representation.

Pattern	<i>unbounded tone plateauing</i>
Type of data	artificial data
Example of data	HHHHH, LHHLL, LHHHH, LHHHH, HHHHH, ...
Learned 3-SL grammar	HLH
Generated sample	HHHLLH, LLHL, LHLLLH, HL, LHH, ...
Evaluation	evaluate_utp_strings(sample)
Score	85%

Table 3.21: SL learning of unbounded tone plateauing; abstract representation.

Experiment 9: first-last harmony As expected, the SL learner fails to generalize the pattern of the first-last harmony. Apart from this pattern not being SL/TSL/MTSL, it involves a long-distance dependency between the beginning and the end of the word. The sole long-distance nature makes this phenomenon already impossible to model with SL grammars. The learner only induces that x cannot be the first or the last symbol in the well-formed strings of that language because all well-formed strings start and end with either a or o . The performance of such model is 51%.

Pattern	<i>first-last harmony</i>
Type of data	artificial data
Example of data	axoaaxaxaa, aaxaaxxxoa, ooaxaoaooo, axxoaxaaaa, ...
Learned 2-SL grammar	⋈x, x⋈
Generated sample	oxa, aaaaaxxooaoaoa, ooo, oxxo, oao, a, ooa, ...
Evaluation	<code>evaluate_first_last_words(sample)</code>
Score	51%

Table 3.22: SL learning of first-last harmony; abstract representation.

3.2.4 SL experiments: interim summary

The nature of SL models is to capture local generalizations. As a result, the only challenge that SL grammars successfully passed was learning *word-final devoicing* that can be rephrased as “do not have voiced obstruents /b/, /d/ and /g/ at the end of the word”. This experiment involved learning the pattern from 3 different representations of data: artificially generated sample, simplified German data, and the raw German wordlist. SL succeeded in generalizing this local pattern from all three representations, scoring 100% on every one of them.

The rest of the experiments included different versions of long-distant harmonies or other non-local patterns such as unbounded tone plateauing, and unattested pattern of the first-last harmony. Needless to say, the SL learner is not suited for those types of patterns, and therefore its average score was relatively low. Section 3.6 shows the chart of the performance of SL models in comparison to the results obtained by other subregular learners.

3.3 Strictly piecewise models

The previous section shows that SL grammars are not capable of modeling long-distance dependencies such as tone plateauing and vowel harmonies of different types. SP grammars, on the contrary, are not able to model local generalizations. They prohibit *subsequences* and not *substrings*: if a bigram ab is prohibited, it means that a cannot be followed by b anywhere further in the string. For example, an SL language prohibiting ab will rule out a word $baaabb$ but will consider $acccb$ grammatical. However, an SP language with the same bigram listed in its grammar will rule out both words: in both of them, a is followed by b . Therefore, in linguistics, SP models are used to model long-distance processes that do not exhibit a blocking effect, such as some cases of harmonies and unbounded tone plateauing.

3.3.1 SP learning algorithm

The **intuition** behind the SP learner is similar to the one behind the SL induction algorithm. But instead of recording all the attested substrings, the SP learner memorizes all the subsequences that were observed in the training sample. In such a way, it constructs a hypothesis of a positive grammar describing the input data. If a negative grammar needs to be constructed instead, it generates all possible k -local subsequences based on the given alphabet and then removes from that set the ones that were attested in the training data. Read more about such learners in (Heinz, 2010a).

The **pseudocode** of the k -SP learner is given below. It iteratively expands a set of k -long subsequences P based on the word w contained in the training sample.

Algorithm 2 Extracts G_{SP_k} from w

Require: $|w| \geq k$ $P \leftarrow \{w[0] \dots w[k-1]\}$ $C \leftarrow \emptyset$ **for** s in $\{w[k] \dots w[|w|]\}$ **do** **for** p in P **do** **for** i in $\{0 \dots k-1\}$ **do** $sq \leftarrow \{p[0] \dots p[i-1]\} \cup \{p[i+1] \dots p[|p|]\} \cup \{s\}$ $C \leftarrow C \cup \{sq\}$ **end for** **end for** $P \leftarrow P \cup C$ $C \leftarrow \emptyset$ **end for****return** P

3.3.2 Successful experiments

The experiments in which the SP learner performed exceptionally well were the ones that included long-distance dependencies that cannot be blocked by anything. Those were some of the harmonic systems and the pattern of unbounded tone plateauing. However, in cases where the dependency was local or a blocking effect was involved, the learner failed to capture the pattern. As a result, the SP learner did not model the patterns of the word-final devoicing, and even the simplest cases of harmonic systems that included a blocker.

Experiment 2: a single vowel harmony without blocking The SP learner performed extremely well when challenged with the task of learning a single vowel harmony pattern without a blocker. For this experiment, I used 3 different datasets: automatically generated words imitating the harmonic pattern, masked

Finnish words, and the raw Finnish data. On all these datasets, the output of the SP models consisted exclusively of well-formed words therefore scoring 100%.

When faced with the abstract representation of the pattern, the SP learner extracted the grammar $\{ao, oa\}$ that can be interpreted as *after a occurred in the string, o cannot be seen, and vice versa*. This is exactly the correct generalization, and therefore the output of the generator contained only well-formed words: *axxaax*, *xooxoo*, and so on. The accuracy of such a model is 100%.

Pattern	<i>one vowel harmony, no blockers</i>
Type of data	artificial data
Example of data	oxxxooxxxx, ooxxxooxxo, aaxxxaaxxx, oxxxooxxxo, ...
Learned 2-SP grammar	ao, oa
Generated sample	axxaax, aa, oo, ooxxxox, ooxxxo, xooxoo, 'xaxa, ...
Evaluation	harmonic_evaluator(sample, single_harmony_no_blockers)
Score	100%

Table 3.23: SP learning of a single harmony without blockers; abstract representation.

Not much changed when the learner was challenged with the masked Finnish dataset: it observed that front vowels are never followed by back vowels in Finnish words and vice versa. As a consequence, it extracted all the combinations of the type $[\alpha\text{front}][-\alpha\text{front}]$: *äa*, *äo*, *äu*, and others. Even with the larger vowel vocabulary, the performance of such a learner was still 100%.

Finally, even when the learner was given raw Finnish data, it saw the same pattern of fronting harmony. Apart from the rules of the harmony it also extracted the irrelevant subsequences that were never observed in the training sample such as *äw* or *wq*. All of the words generated with the obtained grammar were grammatical regarding the rules of Finnish vowel harmony.

Pattern	<i>one vowel harmony, no blockers</i>
Type of data	masked Finnish data
Example of data	xaxxixxex, xixäxixex, xaxxixxaxix, uxxuxaxxix, ...
Learned 2-SP grammar	äa, äo, äu, öa, öo, öu, ...
Generated sample	ua, xouuaxu, ooa, axu, äyxxä, a, ää ...
Evaluation	harmonic_evaluator(sample, front_harmony)
Score	100%

Table 3.24: SP learning of a single harmony without blockers; masked representation.

Pattern	<i>one vowel harmony, no blockers</i>
Type of data	raw Finnish data
Example of data	mathilden, lisänimen, macmillanin, urquhartin, ...
Learned 2-SP grammar	äq, äu, äw, öa, öo, vq, ...
Generated sample	cykqäkpjprpbhftä, yesxöven, hägvgs, dtvza, ...
Evaluation	harmonic_evaluator(sample, front_harmony)
Score	100%

Table 3.25: SP learning of a single harmony without blockers; raw representation.

Experiment 4: several vowel harmonies without blocking The SP learner correctly constructed a grammar for an artificial language where vowels harmonize for two features, therefore creating 4 choices of vowels: $[-\alpha, -\beta]$, $[-\alpha, +\beta]$, $[+\alpha, -\beta]$, and $[+\alpha, +\beta]$. The obtained SP grammar that represents this pattern is *au*, *ae*, *ao*, *ua*, etc. 100% of the words generated by the learner are harmonic and well-formed.

Pattern	<i>several vowel harmonies, no blockers</i>
Type of data	artificial data
Example of data	xuuuxxxuuu, xxxeeexeee, xxxaaxxxaa, xooxxooxox, ...
Learned 2-SP grammar	ae, ao, ua, ue, uo, ...
Generated sample	'a', 'ooxooo', 'ooo', 'oxoxoo', 'exexee', ...
Evaluation	harmonic_evaluator(sample, double_harmony)
Score	100%

Table 3.26: SP learning of several vowel harmonies without blockers; abstract representation.

Experiment 6: several vowel harmonies without blocking The learner also successfully learned the pattern involving two independent spreadings, such as consonant harmony for $[\alpha]$ and vowel harmony for $[\beta]$. It inferred the exactly correct rules: don't have consonants disagree (*pb*, *bp*) and don't have vowels disagree (*ao*, *oa*). Again, the performance of the learner is 100%.

Pattern	<i>vowel and consonant harmonies, no blockers</i>
Type of data	artificial data
Example of data	bbbaaabbaa, bbooboboob, pppoopppop, appaaappaa, ...
Learned 2-SP grammar	ao, oa, bp, pb, $\times \times$
Generated sample	pp, bobbbb, pppapp, ppa, ...
Evaluation	harmonic_evaluator(sample, double_harmony_no_blockers)
Score	100%

Table 3.27: SP learning of vowel and consonant harmonies without blockers; abstract representation.

Experiment 8: unbounded tone plateauing The learner also succeeded in learning the UTP pattern. Notice, that this requires examining three elements,

since the low tones (*L*) are prohibited if they are *in-between* two high tones (*H*). SP generalization *HLH* exactly describes this pattern and correctly rules out strings such as *HHHLLLLHH* that contain the illicit subsequence.

Pattern	<i>unbounded tone plateauing</i>
Type of data	artificial data
Example of data	HHHHH, LHHLL, LHHHH, LHHHH, HHHHH, ...
Learned 3-SP grammar	HLH, $\times\times\times$, $\times\times\times$, ...
Generated sample	HHHLL, LLL, LHHH, HH, LLLHHLLL ...
Evaluation	<code>evaluate_utp_strings(sample)</code>
Score	100%

Table 3.28: SP learning of unbounded tone plateauing; abstract representation.

3.3.3 Unsuccessful experiments

SP models failed to learn other patterns such as word-final devoicing and all cases of harmonies with a blocking effect. Strictly local generalizations, as well as the notion of a blocker, cannot be expressed in a strictly piecewise way. Also, as expected, the algorithm failed to learn first-last harmony.

Experiment 1: word-final devoicing The phenomenon of word-final devoicing cannot be expressed via SP grammars. More generally, the core notion of word-initial and word-final markers is not relevant for SP. For example, prohibiting the subsequence *b \times* would rule out any word in which *b* is present since if the words are annotated with the end markers, then any word containing *b* contains *b \times* as a subsequence. The extracted grammar is empty, and given that the grammar is negative, it translates to the generalization *anything goes*. The learner fails to acquire the pattern and produces well-formed strings only in 68% of the cases.

Pattern	<i>word-final devoicing</i>
Type of data	artificial data
Example of data	aaabbbpbbbp, pbapbapapa, apabaappap, bbbbaabbbp, ...
Learned 2-SP grammar	\emptyset
Generated sample	bapaaab, bpbpababa, pppabbpbba, bpaaap, pb, ...
Evaluation	evaluate_wfd_words(sample)
Score	68%

Table 3.29: SP learning of the word-final devoicing; abstract representation.

Experiment 3: a single vowel harmony with blocking Strictly piecewise constraints target subsequences at *any distance* from each other within a word. Therefore, if an SP grammar prohibits *oa*, it will simply miss a blocker that could license such configurations. Under the perspective of such an SP grammar, both strings *ooxoxaa* and *xxooxfaaax* contain *oa*, and therefore need to be ruled out. However, if given a dataset of harmony with a blocker that includes words such as *xxooxfaaax*, the learner assumes that the sequence *oa* is observed and, in fact, possible. Since *ao* is not prohibited, words such as *ooaa* are generated by the learned grammar. SP grammars, therefore, cannot express blockers, so the performance of the learner on this artificial dataset is 84%.

Pattern	<i>one vowel harmony with blockers</i>
Type of data	artificial data
Example of data	oxxxooxxxx, xxooxfaaax, aaxxxaaxxx, ofxxafxxaa, ...
Learned 2-SP grammar	ao, fo
Generated sample	fafa, oa, x, ooa, aafxxfaxafxfxfxfax ...
Evaluation	harmonic_evaluator(sample, single_harmony_with_blockers)
Score	84%

Table 3.30: SP learning of a single harmony with blockers; abstract representation.

Experiment 5: several vowel harmonies with blocking As shown before, SP grammars cannot model a blocking effect. Therefore they do not perform well even on the artificial dataset exhibiting Turkish harmony, where non-high vowels serve as blockers for the rounding harmony. The fronting harmony cannot be blocked by anything, and therefore fronting harmony can in fact be modeled in a strictly piecewise way. However, the violations of the rounding harmony cause the accuracy of the predictions of the grammar to not be higher than 76%.

Pattern	<i>several vowel harmonies with blockers</i>
Type of data	artificial data
Example of data	xxöexxix, xxüüxüüx, exxiixee, iiexxxex, xuuxxuüu, ...
Learned 2-SP grammar	iö, ü, ie, ii, io, ...
Generated sample	ux, e, axi, ixxxix, uax, oxi , ai, ...
Evaluation	harmonic_evaluator(sample, backness_and_rounding)
Score	76% overall (100% backness only, 76% rounding only)

Table 3.31: SP learning of several harmonies with blockers; abstract representation.

Since the SP learner fails on the artificial dataset, its performance is also far from ideal on the masked and raw Turkish data. Interestingly, however, it performed the best (89%) on the raw Turkish data, while scoring 76% percent on the masked corpus. This case is similar to German, where the SP model performed badly on the artificial and masked corpora (68% and 58%, correspondingly), but was able to achieve the accuracy of 89% on the raw data.

Experiment 7: vowel harmony and consonant harmony with blocking This dataset included blockers as well, therefore making it impossible to model the target generalization using an SP grammar. This double harmonic pattern included two types of assimilations: vowel and consonantal. The vowel harmony cannot be blocked by anything, and therefore SP grammars model it well.

However, the consonant harmony included a blocker. Thus, the SP grammar cannot model the consonant harmony since it predicts such ungrammatical strings as *bptottpppttp*. The overall accuracy of the model is 83%.

Pattern	<i>vowel and consonant harmonies with blockers</i>
Type of data	artificial data
Example of data	pppoootopt, obbbtpooot, aabbbaatat, pppaapappp, ...
Learned 2-SP grammar	ao, oa, pb, tb
Generated sample	bptottpppttp, ppp, obtp, o, bttoop, babpapp , ...
Evaluation	harmonic_evaluator(sample, double_harmony_with_blockers)
Score	83%

Table 3.32: SP learning of vowel and consonant harmonies with blockers; abstract representation.

Experiment 9: first-last harmony SP grammars cannot capture the pattern of first-last harmony. The learner extracts an empty grammar, so it allows for any sequence of *x*, *a* and *o*: clearly, it does not enforce the first and the last vowels to match. In fact, as the German final devoicing experiments show, SP grammars cannot distinguish between word-internal and word-final/initial positions. Only 32% of the strings predicted by the learned grammar were grammatical with respect to the rules of the first-last harmony.

3.3.4 SP experiments: interim summary

The nature of strictly piecewise grammars allows them to capture long-distant generalizations. The SP learner thus performs incredibly well (100%) on challenges such as single and multiple long-distance harmonies without blockers, and also on the pattern of unbounded tone plateauing.

However, the effect of blocking cannot be expressed in an SP way since as soon

Pattern	<i>first-last harmony</i>
Type of data	artificial data
Example of data	axoaaxaxaa, aaxaaxxxoa, ooaxaoaooo, axxoaxaaaa, ...
Learned 2-SP grammar	\emptyset
Generated sample	oxa, aaaaaxxooaoaoa, ooo, oxxo, oao, a, ooa, ...
Evaluation	<code>evaluate_first_last_words(sample)</code>
Score	32%

Table 3.33: SP learning of first-last harmony; abstract representation.

as the learner observes the disagreeing elements across the blocker, it assumes that those elements can disagree in general. Both word-final devoicing and first-last harmony patterns rely on the notion of being word-initial or word-final, and they cannot be represented with a SP perspective: an SP constraint $x\bowtie$ would simply rule out all strings where x is present. The long-distant nature of the SP generalizations makes it impossible for them to capture local generalizations.

3.4 Tier-based strictly local models

Previous sections discussed strictly local and strictly piecewise languages. The first ones were *only* able to model local dependencies, whereas the latter ones *only* expressed the long-distance ones, without the ability to be sensitive to intervening material. As a result, none of these two classes were able to handle long-distance harmonies with blockers. The subregular class of tier-based strictly local grammars has a way of representing long-distance constraints locally. TSL grammars project some characters of a string on a *tier* therefore making elements from that set local, and ignoring the elements that are not included in that set. For example, a grammar with tier symbols a and b and the restriction ab rules out

words such as *acccbb*: its *tier image* is *abb*, and it is ill-formed since it contains the prohibited substring *ab*. In this way, we get another perspective on modeling long-distance dependencies.

3.4.1 TSL learning algorithm

To learn a TSL grammar, simply extracting factors from the data is not sufficient. The goal is to uncover the *tier alphabet*, or a set of elements exhibiting a long-distance dependency. Currently, the most powerful learner for the TSL class is *kTSLIA* designed by Jardine and McMullin (2017).⁶ **Intuitively**, it initially assumes that every symbol of the alphabet (Σ) is also a member of a tier alphabet (T), and then looks for evidence if it can remove that symbol from T . For an item to be removed from T , it needs to satisfy two conditions. First, it can be freely removed from anywhere; and second, it needs to be able to be inserted everywhere. This algorithm is interpretable, and learns the k -TSL class of languages in polynomial time and data for any value of k .

The **pseudocode** of this algorithm uses two auxiliary functions – *ngrams* and *tier*, where *ngrams*(I , k) extracts all k -local substrings from the given set of strings I , and *tier*(s) creates a tier representation of a string s . So, for example, if the string is $s = cacbcca$ and the tier is $T = \{a, b\}$, *tier*(s) evaluates to *aba*. Σ^k stands for all k -grams that can be build using the elements of Σ . This algorithm starts by assuming that every member of Σ needs to be included in T . Then for every symbol x , it explores all $(k-1)$ -grams of the observed data sample and tries to insert x in all positions of those $(k-1)$ -grams: like this, it yields a set of k -grams, let us call it A . It also explores all $(k+1)$ -grams containing x and removes x from those therefore yielding another set of k -grams, let's call it B . If the sets A and B are subsets of k -grams found in the input sample, the symbol x is removed from the tier alphabet because its behavior is not crucial for the target language. Finally,

⁶Its earlier versions were presented in (Jardine, 2016b) and (Jardine and Heinz, 2016).

Algorithm 3 Extracts G_{TSL_k} from I

Require: a finite input sample $I \in \Sigma^*$, a positive integer k

```

 $L \leftarrow \text{ngram}(I, k - 1)$ 
 $N \leftarrow \text{ngram}(I, k)$ 
 $M \leftarrow \text{ngram}(I, k + 1)$ 
 $T \leftarrow \Sigma$ 
for  $x$  in  $T$  do
    if  $\forall uv \in L, uv \in N$  and  $\forall uv \in M, uv \in N$  then
         $T \leftarrow T \cap \{x\}$ 
    end if
end for
 $R \leftarrow T^k \cap \{\text{tier}(s) : s \in I\}$ 
return  $T, R$ 

```

it constructs a list of restricted bigrams R by collecting n -grams that were *not* observed in tier images of the training sample.

3.4.2 Successful experiments

TSL grammars can learn patterns in which a set of items exhibits some local or long-distant dependency. However, when several different long-distant dependencies are affecting different sets of elements, such as the independent vowel and consonant harmonies, the power of TSL grammars is not enough.

Experiment 1: word-final devoicing TSL grammars are a superset of the SL ones, and therefore it is following from the subregular hierarchy itself that TSL grammars can express patterns that are expressible using the SL ones. I represent tiers as tuples of the form $(a, b, p)_T$ followed by a list of restrictions that are imposed on that tier.

The performance of the TSL learner remained 100% on the dataset of masked

Pattern	<i>word-final devoicing</i>
Type of data	artificial data
Example of data	aaabbbpbbbp, pbapbapapa, apabaappap, bbbbaabbbp, ...
Learned 2-TSL grammar	$(a, b, p)_T: b^\times, \times^\times$
Generated sample	bppaapbaabp, aabbbabbbp, abbpaba, a, babpp ...
Evaluation	<code>evaluate_wfd_words(sample)</code>
Score	100%

Table 3.34: TSL learning of the word-final devoicing; abstract representation.

and raw German data as well, therefore I am only presenting the results of the latter experiment. In these cases, the TSL learner learned the tier that is the same as the alphabet of the language, thus simply learning the SL grammar.

Pattern	<i>word-final devoicing</i>
Type of data	raw German data
Example of data	hochjagende, zugebliebener, verbricht, besuchszimmer, ...
Learned 2-TSL grammar	$(a, b, c, d, e, f, \dots)_T: b^\times, cj, cv, cw, cx, \dots$
Generated sample	mlqftorjoiäxäzmölnmt, nlca, uoüßmakoöräzum...
Evaluation	<code>evaluate_wfd_words(sample)</code>
Score	100%

Table 3.35: TSL learning of the word-final devoicing; raw representation.

Experiment 2: a single vowel harmony without blocking The TSL learner extracted a grammar describing a single harmony pattern without blocking from two datasets: the masked version of Finnish harmony, and the abstract representation of Finnish. On both representations, it performed superbly: 100%.

The abstract representation is based on an artificial language dataset. When this is given as the input, the learner extracted the tier $T = \{a, o\}$. Notice, that it

correctly excluded x from the tier. Based on the set of unigrams of the data $\{a, o, x, \times, \bowtie\}$, it constructed a list of bigrams $\{xa, ax, ox, xo, xx, \times x, x \times\}$, and all these bigrams appeared in the input sample. Then, it tried to remove x from all observed 3-grams, and again, the obtained list of bigrams was found in the input data. Therefore, x was considered not a tier element. This TSL grammar rules out ungrammatical strings such as $axaxxxxoo$, since the tier image of that string is $aaoo$: the bigram that is seen on the tier image.

Pattern	<i>one vowel harmony, no blockers</i>
Type of data	artificial data
Example of data	oxxxoxxxxx, oxxxoxxxo, aaxxxaaxxx, oxxxoxxxo, ...
Learned 2-TSL grammar	$(a, o)_T$: $ao, oa, \times \times$
Generated sample	xaxxxaxx, xxoxxxxox, xxaxax, xxox, xxxoxxox, ...
Evaluation	<code>harmonic_evaluator(sample, single_harmony_no_blockers)</code>
Score	100%

Table 3.36: TSL learning of a single harmony without blockers; abstract representation.

The learner also extracted the TSL grammar from masked Finnish data: it also conjectured that x is not a tier element. On a tier of all vowels, that grammar prohibited all combinations of vowels that disagree in backness.

However, it failed to learn the rule of the Finnish vowel harmony from a raw data: only 42% of the words that that grammar generated, were, in fact, well-formed regarding the rules of Finnish vowel harmony. The majority of the Finnish letters, for the exception of l and n , were unable to be freely inserted into any $n - 1$ -gram, or deleted from any $n + 1$ -gram of the data, and therefore they were not excluded from the tier alphabet.

Pattern	<i>one vowel harmony, no blockers</i>
Type of data	masked Finnish data
Example of data	xaxxixxex, xixäxixex, xaxxixxaxix, uxxuxaxxix, ...
Learned 2-TSL grammar	$(a, o, u, \text{ä}, \text{ö}, y)_T$: äa, äo, äu, öa, öo, ...
Generated sample	xyyyäxyxyxäxx, öxöäxyxx, yäxyxyxyxxäxöxxäxxxä, ...
Evaluation	harmonic_evaluator(sample, front_harmony)
Score	100%

Table 3.37: TSL learning of a single harmony without blockers; masked representation.

Pattern	<i>one vowel harmony, no blockers</i>
Type of data	raw Finnish data
Example of data	mathilden, lisänimen, macmillanin, urquhartin, ...
Learned 2-TSL grammar	$(a, b, c, d, e, f, g, \dots)_T$: äq, äu, äw, öa, öo, ...
Generated sample	nololdlnölllyn, tlnynlpll, leldznnleln ...
Evaluation	harmonic_evaluator(sample, front_harmony)
Score	42%

Table 3.38: TSL learning of a single harmony without blockers; raw representation.

Experiment 3: a single vowel harmony with blocking Among the learners which i discussed so far, the TSL learner was the only one that was able to learn the rules of a single vowel harmony with blocking effect. In doing so, it achieved the accuracy of 100%. The learned grammar correctly excluded x from the tier, and noticed that the blocker f is crucial for this language. It prohibited disagreeing vowels adjacent on the tier; and if a blocker intervenes, the following vowel must not be o . Thus this grammar correctly rules out words such as $ooxxfxxo$ and $ooxxxxa$ since tiers of these strings contain the prohibited bigrams fo

and *oa*, correspondingly, see figure 3.1.

Pattern	<i>one vowel harmony with blockers</i>
Type of data	artificial data
Example of data	oxxooxxxx, xxooxfaaax, aaxxxaaxxx, ofxxafxxaa, ...
Learned 2-TSL grammar	$(a, f, o)_T$: ao, fo, oa
Generated sample	xxoxfx, xxa, xxxoxoxfxaxaxfx, xxaxaxxx, ...
Evaluation	<code>harmonic_evaluator(sample, single_harmony_with_blockers)</code>
Score	100%

Table 3.39: TSL learning of a single harmony with blockers; abstract representation.

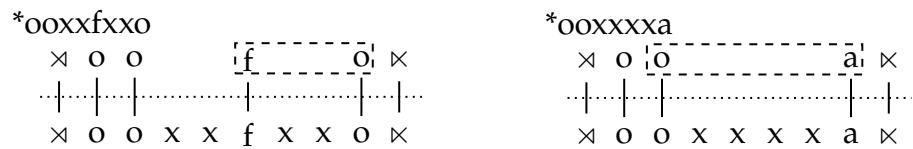


Figure 3.1: The extracted TSL grammar evaluating strings (Experiment 3)

Experiment 4: several vowel harmonies without blocking The learner induced that the “consonant” *x* is not relevant for the vowel harmony system, and it also learned that on the tier of vowels, no disagreeing vowels can appear next to each other. This learning outcome is therefore similar to the one of the second experiment, but with 4 harmonic classes inferred instead of 2. This TSL grammar only generates words that are well-formed with respect to the rules of the vowel harmony thus scoring 100% accuracy.

Experiment 5: several vowel harmonies with blocking The TSL learner correctly built a model for a Turkish-style harmonic system. One tier is, indeed, able to capture both spreadings at the same time. Given the tier consisting of all the vowels, the backness harmony can be expressed by a set of constraints of the

Pattern	<i>several vowel harmonies, no blockers</i>
Type of data	artificial data
Example of data	xuuuxxxuuu, xxeeexeee, xxxaaxxxaa, xooxxooxox, ...
Learned 2-TSL grammar	$(a, e, o, u)_T$: ae, ao, ua, ue, uo, ...
Generated sample	xxxaxaxaxax, xxxxxexx, xxuxx, oxoxxox, ...
Evaluation	harmonic_evaluator(sample, double_harmony)
Score	100%

Table 3.40: TSL learning of several vowel harmonies without blockers; abstract representation.

type $[\alpha\text{front}][-\alpha\text{front}]$ (*üu, uü, ae, ea, üa, ...*), and the rounding harmony that is blocked by non-high vowels is generalized as restrictions of the shape $[\alpha\text{round}][-\text{high}, +\text{round}]$ (*oo, uo, eo, ...*) and $[\alpha\text{round}][-\alpha\text{round}, +\text{high}]$ (*öi, oi, au, ...*). The performance of such model is 100%.

Pattern	<i>several vowel harmonies with blockers</i>
Type of data	artificial data
Example of data	xxöexxix, xxüüxüüx, exxiixee, iiexxxex, xuuxxuuu, ...
Learned 2-TSL grammar	$(i, ö, ü, a, e, i, o, u)_T$: iö, iü, ie, ii, io, ...
Generated sample	xxöxxexx, xixxx, axixxx, xaxxaxx, ...
Evaluation	harmonic_evaluator(sample, backness_and_rounding)
Score	100% overall (100% backness only, 100% rounding only)

Table 3.41: TSL learning of several harmonies with blockers; abstract representation.

This result is theoretically expected since, in this pattern, there are two vowel harmonies happening at the same and affecting the same set of segments. If we take two TSL grammars G_1 and G_2 that have the same tier alphabet but different

sets of prohibited n -grams, taking a union of the prohibited n -grams would yield us another TSL grammar G_3 . Importantly, G_3 generates a language that is the intersection of the languages of G_1 and G_2 . To re-iterate this in more linguistic terms, *two harmonies can fit on the same tier if the same sets of elements are involved in them*.

However, the learner did not perform well on the masked Turkish data. It failed to remove x from the tier since it did not observe $\ddot{o}\ddot{u}$ and ou adjacent to each other, even though they are well-formed regarding the harmonic rules. Its hypothesis, therefore, was not different from the one postulated by the SL grammar, and the accuracy of the model is 67%.

Pattern	<i>several vowel harmonies with blockers</i>
Type of data	masked Turkish data
Example of data	xixxix, xuxx, xaaxa, xexxe, xaxix, ...
Learned 2-TSL grammar	$(i, \ddot{o}, \ddot{u}, a, e, i, o, u, x)_T$: i \ddot{o} , i \ddot{u} , ie, ii, io, ...
Generated sample	uux \ddot{u} ix, \ddot{u} , iaaxeix, xiiix \ddot{u} e, ...
Evaluation	harmonic_evaluator(sample, backness_and_rounding)
Score	67% overall (74% backness only, 74% rounding only)

Table 3.42: TSL learning of several harmonies with blockers; masked representation.

The artificial dataset freely allowed vowel hiatus. Harmonic vowels could be adjacent, such as in *xxouxx*. However, Turkish has gaps in what pairs of harmonic vowels can be adjacent in vowel hiatus. Consequently, the performance of the TSL learner on the raw Turkish data was even worse, namely, 30%.

3.4.3 Unsuccessful experiments

TSL grammars can model patterns when a single set of elements is involved in a long-distance dependency. However, if there is more than one long-distant process affecting different sets of elements, such as independent vowel and consonant harmonies, one tier is not enough.

Experiment 6: vowel harmony and consonant harmony without blocking It is impossible to model independent vowel and consonant harmonies using TSL grammars. Vowels and consonants are involved in different long-distant processes, and therefore neither of them can be removed from the tier. However, the presence of consonants does not allow us to represent vowels in a tier-based local fashion, and vice versa. Therefore, neither vowel nor consonant harmony can be enforced: only 74% of the words generated by such grammar are well-formed. This model is the same as its SL counterpart.

Pattern	<i>vowel and consonant harmonies, no blockers</i>
Type of data	artificial data
Example of data	bbbaaabbaa, bbooboboob, pppooppop, appaaappaa, ...
Learned 2-TSL grammar	$(a, b, o, p)_T$: ao, oa, bp, pb
Generated sample	apapoopop, oppoobbbooppob, abab, baaaap, ...
Evaluation	<code>harmonic_evaluator(sample, double_harmony_no_blockers)</code>
Score	74%

Table 3.43: TSL learning of vowel and consonant harmonies w/o blockers; abstract representation.

Experiment 7: vowel harmony and consonant harmony with blocking Since TSL grammars failed to model the previous pattern with the independent vowel and consonant harmonies, they also fail to learn a similar pattern with a blocking

effect. Thus the performance of the TSL grammar, in this case, is again similar to its SL counterpart: 69%.

Pattern	<i>vowel and consonant harmonies with blockers</i>
Type of data	artificial data
Example of data	pppoootopt, obbbtpooot, aabbbaatat, pppaapappp, ...
Learned 2-TSL grammar	$(a, b, o, p, t)_T$: ao, oa, bp, pb, tb
Generated sample	ptopaba, btopttpt, a, pppa, ob, ...
Evaluation	harmonic_evaluator(sample, double_harmony_with_blockers)
Score	69%

Table 3.44: TSL learning of vowel and consonant harmonies with blockers; abstract representation.

Experiment 8: unbounded tone plateauing There is no choice of a tier alphabet that would allow a TSL grammar to capture the *no H...L...H* generalization. If *H* and *L* are both present on the tier, such TSL grammar behaves like the SL one. Otherwise either *H* or *L* needs to be omitted from the tier, but both of them are crucially important for the generalization. Hence the pattern of UTP is neither SL nor TSL. The learned grammar performs with the accuracy of 90% exclusively due to a small alphabet and the majority of the generated strings being short.

Experiment 9: first-last harmony As expected, the TSL learner cannot learn the unattested pattern of the first-last harmony. In fact, the obtained grammar is the same as the one proposed by the SL learner, and therefore it makes exactly the same types of mistakes.

Pattern	<i>unbounded tone plateauing</i>
Type of data	artificial data
Example of data	HHHHH, LHHLL, LHHHH, LHHHH, HHHHH, ...
Learned 3-TSL grammar	$(H, L)_T$: HLH
Generated sample	HHHLLH, HLL, HHLHLLLLHLLLLLHHL, LLLLH, ...
Evaluation	<code>evaluate_utp_strings(sample)</code>
Score	90%

Table 3.45: TSL learning of unbounded tone plateauing; abstract representation.

Pattern	<i>first-last harmony</i>
Type of data	artificial data
Example of data	axoaaxaxaa, aaxaaxxxoa, ooaxaoaooo, axxoaxaaaa, ...
Learned 2-TSL grammar	$(a, o, x)_T$: $\bowtie x, x \bowtie$
Generated sample	aaoxaxxoxao, aaxaooxa, axooaao, oxxo, ...
Evaluation	<code>evaluate_first_last_words(sample)</code>
Score	50%

Table 3.46: TSL learning of first-last harmony; abstract representation.

3.4.4 TSL experiments: interim summary

A TSL learner, if given a representative sample of data, extracts a *tier alphabet* that represents a set of elements involved in a long-distance dependency. If every item of that set is also involved in another dependency, it can capture such cases as well, as it did in case of the abstract pattern of Turkish harmony. Overall, TSL learner succeeded in building a grammar for every pattern that exhibited either a local dependency or a long-distance dependency among a single set of elements if given a representative sample.

However, if there is more than a single set of items involved in different long-

distance dependencies, this cannot be modeled by TSL grammars. Therefore TSL learner failed on a challenge that included learning separate vowel and consonant harmonies: for those cases, one tier is not enough.

3.5 Multiple tier-based strictly local models

Previous two sections explore two different perspectives on modeling long-distance dependencies. Strictly piecewise grammars prohibit subsequences elements of which can be *arbitrarily far* from each other. SP models thus handle cases of multiple long-distance dependencies; however, none of them can include blockers. Also, SP models can *only* model long-distance dependencies: they cannot handle locally bounded patterns. TSL grammars can encode local patterns and also blocking effects; however, they are limited to a *single* set of items involved in a long-distance dependency. Hence they cannot encode such cases as independent vowel and consonant harmonies within the same language. In this section, I explore the performance of multiple tier-based strictly local (MTSL) models: namely, models that employ several TSL grammars at the same time.

3.5.1 MTSL learning algorithm

The subregular class of MTSL grammars is a proper extension of TSL. However, the k TSLIA algorithm introduced earlier cannot be simply extended from a single tier to multiple ones since its initial assumption is that *all members of Σ* belong to a tier alphabet: it implicitly assumes the existence of just a single tier.

The MTSL learning algorithm *MTSL2IA* proposed by McMullin et al. (2019) relies on the assumption that we can first detect all the prohibited k -grams, and then learn a tier for every one of them. Thus, we learn *a tier for every negative bigram* of the MTSL grammar. Currently, this algorithm only works with 2-local restrictions, and the work of extending it to k is ongoing.

Crucially, the MTSL2IA algorithm relies on the notion of a *path* denoted as $\langle \rho_1, X, \rho_2 \rangle$. It can be thought of as a subsequence $(\rho_1 \dots \rho_2)$ accompanied by a set of symbols X that occurred in-between ρ_1 and ρ_2 in the training sample. For example, the following paths can be extracted from a string *abac*: $\langle a, \{\}, b \rangle$, $\langle a, \{b\}, a \rangle$, $\langle a, \{b, a\}, c \rangle$, $\langle b, \{\}, a \rangle$, $\langle b, \{a\}, c \rangle$, and $\langle b, \{\}, c \rangle$.

Algorithm 4 Extracts G_{MTSL_2} from I

Require: a finite input sample $I \in \Sigma^*$

```

 $B \leftarrow \Sigma^2 \cap \text{ngram}(I, 2)$ 
 $i \leftarrow 1$ 
for  $\rho_1 \rho_2 \in B$  do
     $R_i \leftarrow \rho_1 \rho_2$ 
     $T_i \leftarrow \Sigma$ 
    for  $\sigma \in \Sigma \cap \{\rho_1, \rho_2\}$  do
        if  $\forall \langle \rho_1, X, \rho_2 \rangle \in \text{path}(I)$  s.t.  $\sigma \in X, \langle \rho_1, X - \{\sigma\}, \rho_2 \rangle \in \text{path}(I)$  then
             $T_i \leftarrow T_i - \{\sigma\}$ 
        end if
    end for
     $G_i \leftarrow \langle T_i, R_i \rangle$ 
     $i \leftarrow i + 1$ 
end for
 $G \leftarrow G_1 \wedge G_2 \dots G_{|B|-1} \wedge G_{|B|}$ 
return  $G$ 

```

Intuitively, this algorithm works as follows. At first, it detects a list of bigrams B that is unattested in the training sample I . Then it loops over all elements of B , and for every bigram $\rho_1 \rho_2 \in B$, it assumes that the tier for that bigram is Σ . Afterwards it collects a set of all paths of the form $\langle \rho_1, X, \rho_2 \rangle$, and finds all symbols $\sigma \in \Sigma$ that can be removed from X so that the newly obtained path $\langle \rho_1, X \setminus \{\sigma\}, \rho_2 \rangle$ is still attested in the list of paths of I . It then removes such σ

from a tier associated with $\rho_1\rho_2$. After all members of B were processed, the algorithm outputs a grammar G that is a collection of all unattested bigrams with the tiers corresponding to those bigrams; see the **pseudocode** above. Similarly to the learners discussed in the previous subsections, MTSL2IA learns the grammar from a positive sample in polynomial time and data (McMullin et al., 2019).

Example Imagine having a dataset that exhibits long-distance sibilant assimilation between ʃ and s unless blocked by f . Additionally, it also has vowel harmony affecting a and o . This dataset includes the following strings: *saasa*, *ʃaʃaa*, *sooos*, *oʃoʃo*, *ʃofoʃ*, *ʃafas*, *sofoʃ*, *safaʃ*, *sʃʃ*, *sʃʃ*, and so on. Of course, strings violating the rules of sibilant (such as *ʃaaʃas* or *soʃooa*) or vowel (*aʃoo*, *sosoa*) harmony are not included in the training sample. As soon as the sample is given as input to the learner, the learner notices the absence of bigrams *sʃ*, *ʃs*, *ao* and *oa*. When it explores the bigram *sʃ*, one of the paths to consider is $\langle s, \{a\}, \text{ʃ} \rangle$. However, $\langle s, \{\}, \text{ʃ} \rangle$ is also a valid path, so a is not a tier element for the bigram *sʃ*, and neither is o . When the unattested bigram *ao* is explored, the learner does not detect any paths that would involve the symbols $\{s, \text{ʃ}, f\}$. The condition of the if-statement is then trivially satisfied, and therefore s , ʃ and f are removed from the tier of that bigram. A concise representation of the grammar that the learner induced is the following:

- $G_1 = \langle T_1 = \{a, o\}, R_1 = \{ao, oa\} \rangle$;
- $G_2 = \langle T_2 = \{s, \text{ʃ}, f\}, R_2 = \{\text{ʃ}s, s\text{ʃ}\} \rangle$.

However, the *tier-per-bigram* assumption comes with a caveat. It results in the algorithm failing to capture patterns where the same bigram is present on several different tiers. For example, consider an MTSL grammar where the bigram *xx* is prohibited on two tiers: $T_1 = \{x, a\}$ and $T_2 = \{x, b\}$. Instead, the MTSL2IA learner would converge on the incorrect tier $T = \{x, a, b\}$. Tier configurations that cannot be learned by this learner is a sub-case of a general case when two tier alphabets

have a non-empty intersection that does not overlap with either of the alphabets. Interestingly, we show in Aks nova and Deshmukh (2018) that in natural languages, if two agreements require two different tiers, those tiers never overlap unless one of them is properly contained within the other one. Therefore if applied to phonological data, this learner could be more efficient in comparison to the learner that would also explore the typologically unattested class of tier configurations.

As noted previously, this algorithm learns 2-local MTSL grammars, but we are currently working on extending it to arbitrary k . Intuitively, this can be done by extending the notion of a path. Its shape could be generalized as $\langle \rho_1, X_1, \rho_2, X_2, \dots, \rho_{n-1}, X_{n-1}, \rho_n \rangle$, where $\rho_1 \rho_2 \dots \rho_n$ is a k -long sequence, and X_i is the set of symbols that occurred in-between ρ_i and ρ_{i+1} in the training sample. The condition of the if-statement needs to also be adjusted to accommodate for longer paths; but otherwise, the logic of the algorithm stays the same.

3.5.2 Successful experiments

In this subsection, I show that MTSL grammars can be used to successfully model all of the discussed types of local processes and long-distant harmonies. Even when challenged with the raw data of German, Finnish, and Turkish, the MTSL learner extracts the corresponding MTSL grammars with the impressive accuracies of 100%, 100%, and 95%, correspondingly. The patterns of unbounded tone plateauing and the first-last harmony are not MTSL in their nature, and therefore cannot be learned using the MTSL inference algorithm.

Experiment 1: word-final devoicing Since MTSL grammars are a proper superclass of TSL grammars, and, consequently, of the SL ones, the MTSL learning algorithm acquires the pattern of word-final devoicing. The performance of the MTSL model on the raw German dataset is 100%, as well as on the other

representations of that pattern.

Pattern	<i>word-final devoicing</i>
Type of data	raw German data
Example of data	hochjagende, zugebliebener, verbricht, besuchszimmer, ...
Learned 2-MTSL grammar	<i>too large: 294 tiers!</i>
Generated sample	mugoftkuhämpo, kisizkkokgüp, rkümsübtal...
Evaluation	<code>evaluate_wfd_words(sample)</code>
Score	100%

Table 3.47: MTSL learning of the word-final devoicing; raw representation.

Experiment 2: a single vowel harmony without blocking Similarly, the MTSL learner extracted the grammar representing a single vowel harmony pattern without a blocking effect. The learner induced a single tier containing symbols *a* and *o*. The grammar for this tier was the same one as in the TSL version of this experiment: *ao, oa*. 100% of the words generated by the obtained grammar were well-formed. The success of the MTSL learner on this and further experiments where the TSL learner performed well follows from the fact that the class of MTSL languages subsumes TSL languages.

The MTSL inference algorithm also successfully learned the generalization from raw and masked Finnish datasets. Indeed, 100% of the words generated by the grammar, such as *rjegovnj* or *läyömppl*, are harmonic. Although the grammar is transparent and fully interpretable, it postulates 266 tiers. An open question is to explain *how exactly* increasing the number of tiers helped the learner to tackle this challenge.

Experiment 3: a single vowel harmony with blocking Again, the success of the MTSL learner in this experiment follows from the fact that TSL grammars are a

Pattern	<i>one vowel harmony, no blockers</i>
Type of data	artificial data
Example of data	oxxxooxxxx, oxxxooxxo, aaxxxaaxxx, oxxxoxxxo, ...
Learned 2-MTSL grammar	$(a, o)_T$: ao, oa
Generated sample	xxoox, xxaxaxa, axaaax, xooox, ...
Evaluation	harmonic_evaluator(sample, single_harmony_no_blockers)
Score	100%

Table 3.48: MTSL learning of a single harmony without blockers; abstract representation.

Pattern	<i>one vowel harmony, no blockers</i>
Type of data	raw Finnish data
Example of data	mathilden, lisänimen, macmillanin, urquhartin, ...
Learned 2-MTSL grammar	<i>too large: 266 tiers!</i>
Generated sample	rjegovnj, läyömppl, axiflt, silöäämydv, ...
Evaluation	harmonic_evaluator(sample, front_harmony)
Score	100%

Table 3.49: MTSL learning of a single harmony without blockers; raw representation.

proper subset of the MTSL ones. However, what is surprising is that the MTSL inference algorithm did not converge on a single tier: instead, it postulated 3 different tiers.

The learner discovered 3 unattested bigrams: *ao*, *fo*, and *oa*. However, in none of the data points *a* was ever followed by *o*, so there were no paths of the type $\langle a, X, o \rangle$: trivially, elements *f* and *x* were considered irrelevant for this restriction. Similarly, *a* was excluded from a tier induced for the restriction *fo* because *f* is never followed by *o*. But when considering the unattested bigram *oa*, paths such as

$\langle o, \{f\}, a \rangle$ and $\langle o, \{f, x\}, a \rangle$ were found in the data, while the ones such as $\langle o, \{\}, a \rangle$ were not. A symbol x was hence removed from the tier of the bigram oa , but the blocker f was not. In such a way, MTSL learner constructs 3 different tiers, but the language of the obtained grammar is equivalent to the one of the TSL grammar $\{oa, ao, fo\}$ with the tier $T = \{a, o, f\}$.

Pattern	<i>one vowel harmony with blockers</i>
Type of data	artificial data
Example of data	oxxxxxxxx, xxooxfaaax, aaxxxaaxxx, ofxxafxxaa, ...
Learned 2-MTSL grammar	$(a, o)_T$: ao; $(f, o)_T$: fo; $(a, f, o)_T$: oa.
Generated sample	oxffxax, faaaxffa, ofxaaf, fa, oooooof, ...
Evaluation	harmonic_evaluator(sample, single_harmony_with_blockers)
Score	100%

Table 3.50: MTSL learning of a single harmony with blockers; abstract representation.

Experiment 4: several vowel harmonies without blocking This experiment was successful since the MTSL learner extracted the grammar for the pattern of several vowel harmonies without a blocking effect. However, similarly to the previous example, the resulting grammar was not the same as the one extracted by the TSL learner. The MTSL algorithm constructed a separate tier for every pair of the potentially disagreeing elements, while correctly noticing that the transparent element x is irrelevant for the generalization.

Experiment 5: several vowel harmonies with blocking The MTSL induction algorithm found a way to model several vowel harmonies with a blocking effect. Also, similarly to the examples discussed above, more than a single tier was inferred. All of the words generated by the extracted MTSL grammar were well-formed regarding the rules of Turkish harmony.

Pattern	<i>several vowel harmonies, no blockers</i>
Type of data	artificial data
Example of data	xuuuxxxuuu, xxxeeexeee, xxxaaxxxaa, xoooxooxox, ...
Learned 2-MTSL grammar	$(u, o)_T$: uo, ou; $(a, u)_T$: au, ua; $(o, e)_T$: eo, oe; etc.
Generated sample	xuxuuuxu, xoox, aaa, exexxe ...
Evaluation	harmonic_evaluator(sample, double_harmony)
Score	100%

Table 3.51: MTSL learning of several vowel harmonies without blockers; abstract representation.

Pattern	<i>several vowel harmonies with blockers</i>
Type of data	artificial data
Example of data	xxöexxix, xxüüxüüx, exxiixee, iiexxxex, xuuxxuux, ...
Learned 2-MTSL grammar	$(ü, e, i)_T$: üi; $(ü, e)_T$: eü; $(ı, ö)_T$: ıö, öı; etc.
Generated sample	ıxxaaaa, üexi, üxeexe, öüüüe, öüexe, ...
Evaluation	harmonic_evaluator(sample, backness_and_rounding)
Score	100% overall (100% backness only, 100% rounding only)

Table 3.52: MTSL learning of several harmonies with blockers; abstract representation.

Some of the configurations that the MTSL learner was looking for were missing in the masked and raw representations of the Turkish data, and therefore the accuracies of those models were slightly worse than ideal: both of them scored 95%. As of the MTSL grammar inferred from the raw data, it relies on 266 tiers, and the way it is able to perform so well is worth further investigation.

Experiment 6: vowel harmony and consonant harmony without blocking The pattern of independent vowel and consonant harmonies can be captured using

Pattern	<i>several vowel harmonies with blockers</i>
Type of data	raw Turkish data
Example of data	som, lafazan, konuk, kekti, lafzan, ...
Learned 2-MTSL grammar	<i>too large: 266 tiers!</i>
Generated sample	apnıcırisaa, telçitçeeriden, mbezkdenic, ...
Evaluation	harmonic_evaluator(sample, backness_and_rounding)
Score	95% overall (100% backness only, 95% rounding only)

Table 3.53: MTSL learning of several harmonies with blockers; raw representation.

two tiers: one for vowels, and another one for consonants. The tier of vowels prohibits *oa* and *ao*, and the tier of consonants rules out *pb* and *bp*. To evaluate the well-formedness of a word, both of its tiers need to be inspected individually. For example, consider a word *ababbabb*. Its consonant tier contains *bbbbbb*, and the vowel tier is *aaa*: both of them are well-formed. However, words such as *ababbbob* are not grammatical: even though the consonant tier does not contain violations, the vowel tier is *aaao*, and it violates the rules of the vowel harmony. The language of this MTSL grammar is the intersection of two TSL grammars, one per every harmony. The accuracy of this model is 100%.

Pattern	<i>vowel and consonant harmonies, no blockers</i>
Type of data	artificial data
Example of data	bbbbaabbaa, bbooboboob, pppoopppop, appaaappaa, ...
Learned 2-MTSL grammar	$(a, o)_T$: <i>ao, oa</i> ; $(b, p)_T$: <i>pb, bp</i> .
Generated sample	oapapaaa, obbb, babbba, poop, ...
Evaluation	harmonic_evaluator(sample, double_harmony_no_blockers)
Score	100%

Table 3.54: MTSL learning of vowel and consonant harmonies w/o blockers; abstract representation.

Experiment 7: vowel harmony and consonant harmony with blocking MTSL learner performs 100% accurate on the pattern with vowel and consonant harmonies even if they include blockers, and it is the only subregular model among the discussed ones that is able to do so. In this case, the learner extracts 4 tiers, and a total of 5 prohibited bigrams. The choice of the tiers can be explained in the same way it was done for the third experiment. On the tier of vowels, the grammar prohibits their disagreeing combinations *ao* and *oa*. The consonant-related restrictions are located across 3 different tiers due to the inference steps of the algorithm, but these restrictions, in fact, can be expressed on a single tier containing *p*, *b*, and *t*. Figure 3.2 shows the MTSL evaluation of strings *aabbotoob* and *aabbbaaap* using a simplified yet equivalent MTSL grammar containing only 2 tiers: one for vowels, and another for consonants.

Pattern	<i>vowel and consonant harmonies with blockers</i>
Type of data	artificial data
Example of data	pppoootopt, obbbtpooot, aabbbaatat, pppaapapp, ...
Learned 2-MTSL grammar	$(b, p, t)_T$: bp; $(a, o)_T$: ao, oa; $(b, p)_T$: pb; $(b, t)_T$: tb.
Generated sample	obtpoppo, totoo, ap, ooptpp, abtatat, ...
Evaluation	<code>harmonic_evaluator(sample, double_harmony_with_blockers)</code>
Score	100%

Table 3.55: MTSL learning of vowel and consonant harmonies with blockers; abstract representation.

3.5.3 Unsuccessful experiments

I was not able to test the performance of the MTSL learner on the UTP pattern since this learner currently exists only for 2-local dependencies, and UTP requires postulating a 3-local restriction. However, this pattern is not MTSL expressible since there is no tier or a combination of tiers that would be able to express that

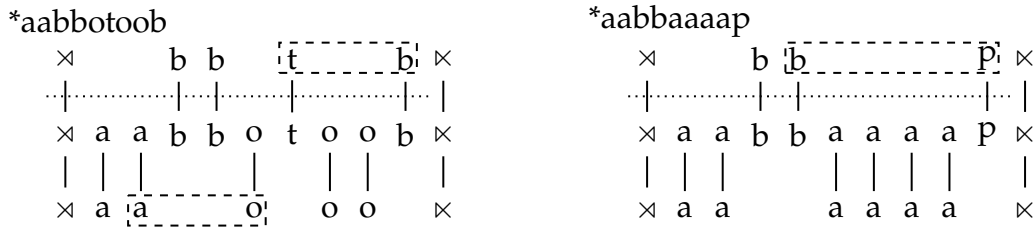


Figure 3.2: Experiment 7: the extracted MTSL grammar evaluating the ungrammatical strings *aabbotoob* and *aabbaaaap*.

generalization. Hence the only unsuccessful experiment that I present in this subsection is the expected inability of MTSL grammars to express the first-last harmony.

Experiment 9: first-last harmony MTSL grammars cannot encode the pattern of the first-last harmony. The MTSL learner extracts exactly the same grammar as TSL and SL learners: it only notices that the “non-agreeing” item x cannot occur string-initially and string-finally. It fails to generalize that the string-initial and string-final symbols need to match, and therefore the accuracy of this model is 50%.

Pattern	<i>first-last harmony</i>
Type of data	artificial data
Example of data	axoaaxaxaa, aaxaaxxxoa, ooaxaoaooo, axxoaxaaaa, ...
Learned 2-MTSL grammar	$(a, o, x)_T: \times x, x \times$
Generated sample	ooaaa, aaoa, ooooaxxa , oaaaxao, ...
Evaluation	<code>evaluate_first_last_words(sample)</code>
Score	50%

Table 3.56: MTSL learning of first-last harmony; abstract representation.

3.5.4 MTSL experiments: interim summary

The MTSL learner successfully extracted MTSL grammars corresponding to all types of harmonic systems present in the list of the experiments, performing equally well on cases with or without the blocking effect. While being able to capture long-distance dependencies, it also performed extremely well on the local pattern of word-final devoicing. Importantly, apart from learning the patterns from the artificially generated datasets, it also was able to generalize the rules from the raw data, scoring 100% on German and Finnish, and 95% on Turkish datasets. The experiment using the non-MTSL pattern of unbounded tone plateauing is not discussed due to the unavailability of the 3-local MTSL learner at the current moment. Finally, as expected, the first-last harmony is not learnable by either of the discussed subregular learners.

However, as explained before in section 3.5.1, there is a type of MTSL grammars that the current learner cannot induce due to its tier-per-bigram assumption. Namely, it cannot learn an MTSL grammar where one bigram belongs to two different tiers. Interestingly, according to Aksënova and Deshmukh (2018), languages with multiple harmonies typologically lack this type of tier configuration. Therefore this learner could be more efficient for language-related tasks than the one that would investigate typologically unattested possibilities.

3.6 Learning languages: summary

In this chapter, I discussed possibilities of modeling natural language patterns using subregular methods. Namely, I explored the performance of strictly piecewise (SP), strictly local (SL), tier-based strictly local (TLS), and multiple tier-based strictly local (MTSL) learning algorithms using different datasets exhibiting natural language dependencies. The experiments ranged from ones

that were using artificially generated samples imitating linguistic patterns, to extracting grammars from raw language data. Artificial language learning shows if the modeling of those generalizations is possible *conceptually*, whereas using raw data shows what is possible *in practice*.

The discussed learning experiments targeted following patterns: word-final devoicing, a single vowel harmony pattern with/without a blocking effect, several vowel harmonies with/without a blocking effect, independent vowel and consonant harmonies with/without blocking effect, and the unbounded tone plateauing. Additionally, I also challenged the learners with a typologically unattested pattern of first-last harmony. Every one among these 9 patterns can be theoretically modeled using different set of subregular classes. Namely, word-final devoicing can be captured by SL, TSL, and MTSL grammars; several vowel harmonies with blocking are expressible by TSL and MTSL grammars, and the unbounded tone plateauing can only be encoded using a SP grammar. Finally, none of the subregular languages should be able to capture the first-last harmony since it is not SL, SP, TSL, or MTSL. In 3.57, I repeat the table with the expected results of the language learning experiments that was previously shown in 3.10.

Every experiment included 4 steps: data collection or generation, subregular learning, sample generation using the constructed grammar, and model evaluation. At first, I prepared the *training samples*. They range from the automatically generated artificial languages, to simplified (masked) representations of the natural language data, to wordlists of German, Finnish and Turkish. During the *learning* step, a grammar was obtained by the inference algorithm based on the provided training data. Then I *generated* a large set of strings that are grammatical according to the extracted grammar. Finally, I computed the number of strings of the generated sample that are well-formed according to the target generalization thus numerically *evaluating* the performance of the model. The table 3.58 summarizes how the automatically

extracted subregular grammars performed on those experiments.

There are two important results of this work. First, every artificial language learning experiment that was predicted to be successful given some particular subregular model was, in fact, successful. It confirms that the implemented algorithms are indeed implemented correctly, and therefore can be reliably used in future. Second, the MTSL learner performed extremely well on raw language data: it learned German word-final devoicing and Finnish harmonic system from a raw data with an accuracy of 100%, and scored 95% on a challenge of learning Turkish harmony.

This line of research needs to be further investigated, as many questions are yet to be answered. These models need to be challenged with more data exhibiting different linguistic dependencies. In case of a successful learning outcome, we need to understand *how exactly* the learner came to the convergence. Otherwise, we need to know *what exactly* prevented the learner from discovering the pattern. Also, there

Target patterns	SP	SL	TSL	MTSL
<i>word-final devoicing</i>	✗	👍	👍	👍
<i>a single vowel harmony without blocking</i>	👍	✗	👍	👍
<i>a single vowel harmony with blocking</i>	✗	✗	👍	👍
<i>several vowel harmonies without blocking</i>	👍	✗	👍	👍
<i>several vowel harmonies with blocking</i>	✗	✗	👍	👍
<i>vowel harmony and consonant harmony without blocking</i>	👍	✗	✗	👍
<i>vowel harmony and consonant harmony with blocking</i>	✗	✗	✗	👍
<i>unbounded tone plateauing</i>	👍	✗	✗	✗
<i>first-last harmony</i>	✗	✗	✗	✗

Table 3.57: The expected results of the language learning experiments; repeated from the end of the section 3.1.4.

are other subregular classes, such as IO-TSL and IBSP, that are important for natural language modeling: learning algorithms for those classes need to be implemented and explored as well.

This chapter, however, is only concerned with modeling the *well-formedness conditions*. In the next chapter, I discuss ways to model *processes* that apply to strings, and transform them according to some set of rules. Namely, similarly to this chapter, I will focus on the ways to infer those rules automatically. Since subregular grammars are interpretable, and subregular learning algorithms are fully transparent, this type of research can in a long run give us larger insights in understanding how human language works.

Data	SP	SL	TSL	MTSL
<i>Experiment 1: word-final devoicing</i>				
Theoretical expectations	✗	👍	👍	👍
Artificial (1,000)	68%	100%	100%	100%
German simplified (658,147)	58%	100%	100%	100%
German (658,147)	89%	100%	100%	100%
<i>Experiment 2: a single vowel harmony without blocking</i>				
Theoretical expectations	👍	✗	👍	👍
Artificial (1,000)	100%	83%	100%	100%
Finnish simplified (250,805)	100%	72%	100%	100%
Finnish (250,805)	100%	41%	42%	100%
<i>Experiment 3: a single vowel harmony with blocking</i>				
Theoretical expectations	✗	✗	👍	👍
Artificial (1,000)	84%	89%	100%	100%
<i>Experiment 4: several vowel harmonies without blocking</i>				
Theoretical expectations	👍	✗	👍	👍
Artificial (1,000)	100%	69%	100%	100%
<i>Experiment 5: several vowel harmonies with blocking</i>				
Theoretical expectations	✗	✗	👍	👍
Artificial (15,000)	76%	59%	100%	100%
Turkish simplified (14,434)	76%	70%	67%	95%
Turkish (14,434)	89%	30%	30%	95%
<i>Experiment 6: vowel harmony and consonant harmony without blocking</i>				
Theoretical expectations	👍	✗	✗	👍
Artificial (1,000)	100%	64%	74%	100%
<i>Experiment 7: vowel harmony and consonant harmony with blocking</i>				
Theoretical expectations	✗	✗	✗	👍
Artificial (1,000)	83%	64%	69%	100%
<i>Experiment 8: unbounded tone plateauing</i>				
Theoretical expectations	👍	✗	✗	✗
Artificial (1,000)	100%	85%	90%	
<i>Experiment 9: first-last harmony</i>				
Theoretical expectations	✗	✗	✗	✗
Artificial (5,000)	32%	51%	50%	50%

Table 3.58: The expected vs. the actual results of the subregular language learning experiments; the experiment 8 cannot be conducted using MTSL learner because it is currently not available for $k > 2$; all other learners are used with $k = 2$.

Chapter 4

Learning mappings

Finite-state transducers are a convenient way to represent natural language processes: they rewrite strings according to the rules they encode. For example, Roark and Sproat (2007) claim that nearly all morphological processes can be modeled using FSTs. Chandlee (2014) in her dissertation shows that subregular functions are a good fit for phonology, and later extends the results to also include morphology (Chandlee, 2017). Heinz and Lai (2013) argue that subsequentiality is crucially important for long-distant phonological processes such as different types of harmonies. Subsequential transducers encode subsequential transformations: they read the input string symbol-by-symbol and output the translation, or a modified representation of that string. Thus, automatically extracting subsequential transducers from data allows to computationally model natural language processes. The learning algorithms analyze the provided pairs of underlying representations (UR) and surface forms (SF), therefore inducing the changes applied to the URs.

In his chapter, I explore the extraction of patterns that occur in natural languages using a popular transduction learning algorithm OSTIA (Oncina et al., 1993). Previously, Gildea and Jurafsky (1996) showed that a corpus of English pronunciations was not enough for OSTIA to generalize the rule of English



flapping. However, they further proceeded to test modified versions of the algorithm on the same corpus yielding improved accuracy. My aim here is to explore what generalizations are possible to model, and which ones cannot be extracted given the current version of the learner. The focus of the chapter is thus understanding what *types* of patterns OSTIA can learn from samples of automatically generated data.

4.1 The OSTIA algorithm

The name *OSTIA* stands for **O**nward **S**ubsequential **T**ransducer **I**nterference **A**lgorithm. Discussed in Oncina et al. (1993) and de la Higuera (2010), this algorithm infers subsequential functions mapping input strings to output strings from a finite sample of such input-output pairs. It identifies any subsequential function in the limit. In other words, given a finite sample of pairs of strings before and after the application of some rule, it extracts a subsequential transducer representing that rule. The property of *the identification in the limit* means that the learner would need a *finite* number of such pairs to induce the target machine. Below I discuss the main steps of this algorithm in 4.1.1, and then present a walk-through of examples in 4.1.2 and 4.1.3, one successful and one unsuccessful.

4.1.1 The pipeline

This algorithm requires a sample of input-output pairs of strings for training, and returns a finite-state subsequential transducer as the output. The algorithm consists of two main parts: creating a representation of data as an onward prefix tree transducer (PTT), thus *structuring* the input data, and *merging* the states of the PTT, therefore, formulating the hypothesis about the underlying rule. The structuring step includes building a PTT for the input sample and making that

PTT onward. Folding sub-trees into one another results in pairs of states being *merged* into a single state. For the pseudocode of the algorithm, refer to Oncina et al. (1993) and de la Higuera (2010). The implementation of OSTIA which I used to obtain the results is a part of the SigmaPie toolkit  (Aksënova, 2020c), and the discussion of that implementation is available on GitHub  (Aksënova, 2019). The main steps are presented in figure 4.1.

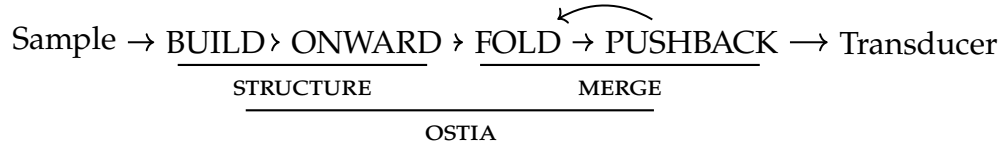


Figure 4.1: The main steps of OSTIA: BUILD, ONWARD, FOLD and PUSHBACK.

BUILD The first step is to represent the input data using a transducer-like data structure. For this purpose, we can build a *prefix-tree transducer* that reads input strings of the training sample symbol-by-symbol, with the common prefixes of those strings stored in the states. The initial state q_ϵ of such a PTT refers to the only common prefix of all the input strings: ϵ . The names of the later states refer to the common prefix those strings are sharing: the states accessible from the state q_ϵ correspond to different first symbols of the input strings. So, for example, a state q_{aba} reads a prefix *aba* by passing through the following states: q_ϵ , q_a , q_{ab} , and q_{aba} . State outputs are set to the translations of the input strings that end up in that state. For example, given the input pair $(ab, 01)$, we save 01 in the state output of the state q_{ab} .

If the state output is not known, it is marked as \perp , or *unknown*. The unknown state output has two properties: *absorbency* and *neutrality*. It is absorbent since its concatenation with any other string returns the same “unknown” output \perp . It is neutral because the longest common prefix of any set of strings W and \perp is the same as the one of W by itself, i.e. \perp is transparent for this operation. In such a way, the

training sample provided to OSTIA is represented as a PTT.

ONWARD The outputs of the PTT are then modified to be *onward*: such a PTT outputs translations as early as possible. During this step, common prefixes of state outputs are pushed closer to the initial state. For example, assume that the intermediate state of the PTT is the one as pictured in 4.2 on the left side, with the onward version of that PTT on the right side. In the input PPT, the state output of the state q_a is 1, and the translations on all edges coming out of q_a ($q_a \xrightarrow{a:10} q_{aa}$ and $q_a \xrightarrow{b:11} q_{ab}$) also contain 1 as their prefix. Therefore, this prefix can be removed from the state output and transitions, and be introduced in the transducer earlier, namely, on the transition incoming into the state q_a . Onwarding starts from the *leaves* of the PTT (the nodes that do not have any outgoing arcs), and percolates to the initial state q_ϵ .

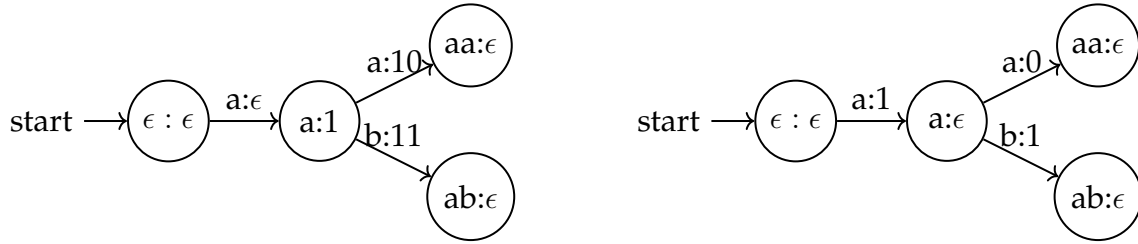


Figure 4.2: Non-onward and onward PTTs that are otherwise equivalent.

FOLD Then, we try to merge every pair of states of the PTT. If (a) the state outputs of q and q' are the same or are \perp , (b) all the incoming branches of q' can be redirected to q , and (c) all outgoing branches from q' are consistent with the outgoing branches of q , states q and q' are merged. Consistency implies either having matching outgoing branches, a possibility to add a missing branch, or, if required, being able to successfully delay a part of the output during the *pushback* step. Folding one state into another decreases the size of the transducer, and shows that the learner generalized the pattern.

PUSHBACK The pushback operation checks if a part of the output can be delayed and therefore removed from some transitions. If pushing back a portion of the output is possible, states q and q' considered during the previous *merge* step are combined, otherwise, their merge is rejected. For example, consider the FST on the left side in figure 4.3. Reading a from the state q_ϵ yields the translation uv . However, as the machine on the right shows, the translation's suffix v can be delayed to the state output of q_a and all the transitions outgoing from q_a . It could let the state q_ϵ be merged with some other state in the FST. After the pushback, OSTIA returns to the merging step and checks if there are other pairs of states that could be merged. When no such pairs remain, OSTIA outputs the FST. In some sense, *pushback* is the operation opposite to *onward* since it delays the outputs, but the resulting FST is always onward.

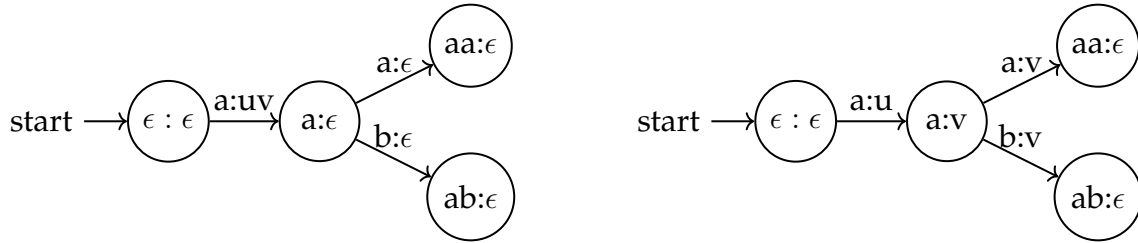


Figure 4.3: OSTIA pushes back the suffix v .

In such a way, OSTIA constructs a subsequential FST that generalizes the mapping from the input strings into their output representations. Note, that as well as the subregular learning algorithms discussed earlier in Chapter 3, OSTIA requires a sample of only positive data.¹ The next subsection presents the inference steps of this algorithm given a concrete example.

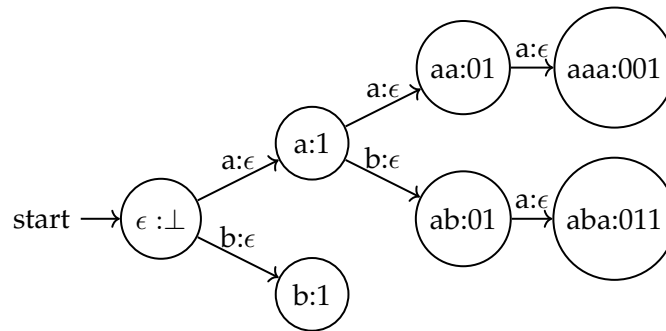
¹The algorithmic complexity of OSTIA is $\mathcal{O}(n^3(m + |\Sigma|) + nm|\Sigma|)$, where n is the sum of the input string lengths, m is the length of the longest output string, and Σ is the input alphabet (de la Higuera, 2010).

4.1.2 The successful example

Here, I discuss a slightly modified example of the OSTIA inference steps originally presented in de la Higuera (2010). The task is to learn the following mapping: word-final a is rewritten as 1, non-word-final a corresponds to 0, and b is always translated as 1. Notice, that this pattern can be viewed as a generalization of a linguistically-motivated process of word-final devoicing since it involves a segment changing its value to the opposite at the end of the word. The training sample that I use in this example is enlarged in comparison to the one presented by de la Higuera (2010): it provides *all* the necessary pairs that guarantee the extraction of the pattern.

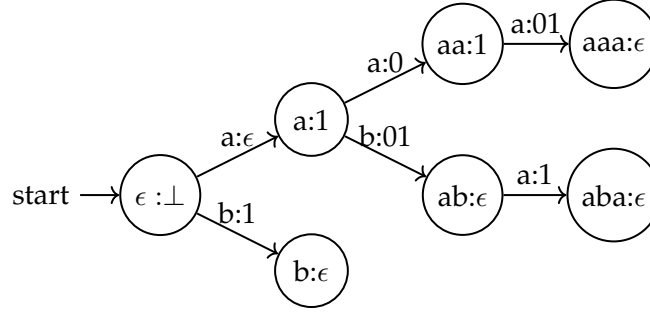
Sample = [(b, 1), (a, 1), (aa, 01), (ab, 01), (aba, 011), (aaa, 001)]

Step I. At first, OSTIA constructs a PTT representing the input sample. This PTT reads the left sides of the training sample one symbol at a time. For every string w of the input pair (w, o) , there exists a state q_w with the state output o . All transitions of this PTT output an empty string. If there is a state $q_{w'}$ that does not correspond to any input string of the training sample, its state output is \perp . For example, there is no empty string in the given sample, so the state output of q_ϵ is \perp .

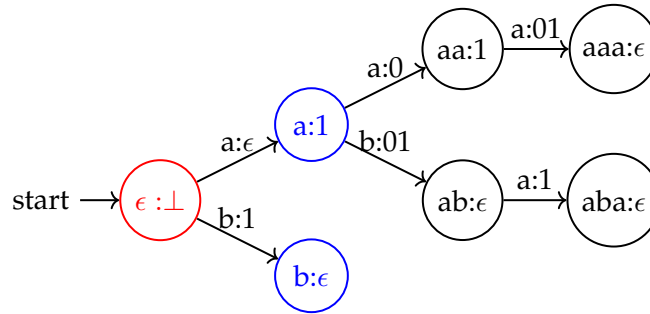


Step II. Then, this PTT is onwarded. For example, consider the state q_{aaa} with the state output 001. We can replace it by ϵ , and instead move 001 to the incoming arc therefore obtaining a transition $q_{aa} \xrightarrow{a:001} q_{aaa}$. The longest common prefix of the

modified transition and the state output of q_{aa} is the longest common prefix of 01 and 001, and that 0 can be moved to the output of the arc $q_a \xrightarrow{a:0} q_{aa}$. Other leaves of the FST are processed similarly. After this step, the input sample is represented as an onward PTT.

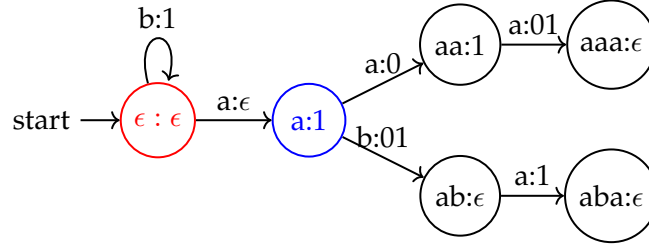


Step III. Next, we start the process of generalizing the obtained PTT by trying to merge pairs of its states. States are colored in two colors: red and blue. *Red states* cannot be eliminated from the FST: they are crucial and therefore cannot be folded into any other state. At first, only the initial state q_ϵ is colored red. All states that can be reached in one step from the red states are colored blue. The status of *blue states* is unclear: either they will be folded into some red states, or they will eventually be re-colored red. After a state was colored red, its immediate children are automatically added to the list of blue states. In our example, two states are colored blue – q_a and q_b – since they can be reached from q_ϵ in one step.

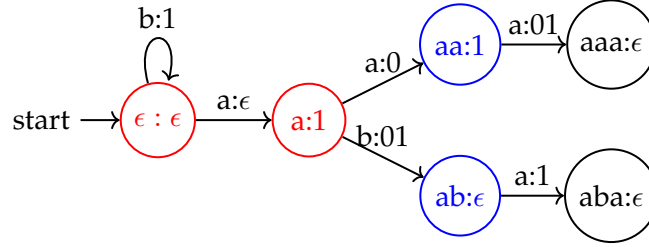


Step IV. Then, the algorithm considers pairs where one state is red and another one is blue and tries to fold the blue state into the red one. Let us then fold the

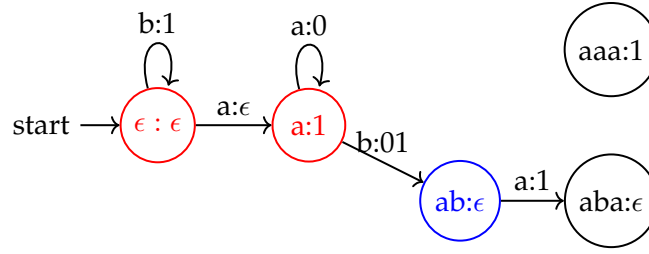
state q_b into q_ϵ . At first, we check if the state outputs of q_b and q_ϵ are compatible. They are \perp and ϵ , and therefore could be merged: they are *not different* due to the transparency of \perp , so we assign ϵ to the state output of q_ϵ . The transition coming to the state q_b is re-directed into the state q_ϵ thus yielding a loop on that state. There is no other sub-tree rooted in q_b , so folding q_b into q_ϵ can be finalized, and q_b is removed from the FST.



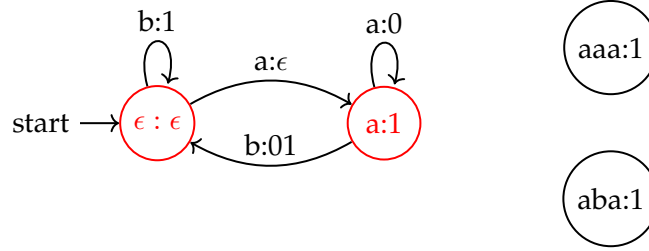
Step V. After the state q_b is eliminated, q_a is the only blue state left. We therefore consider merging q_a into q_ϵ . However, these two states have different state outputs, and therefore it is impossible. As the result, q_a is re-colored red, and q_{aa} and q_{ab} accessible in one step from q_a are colored blue.



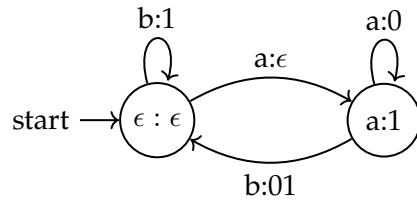
Step VI. We then try to merge q_ϵ and q_{aa} , but it is not possible since they have different state outputs. States q_a and q_{aa} could be merged because they have the same state output: 1. The outgoing arcs reading a from these two states are different: one outputs 0, and another outputs 01. However, the difference is the suffix 1 that can be pushed further to the state output of q_{aaa} , therefore making those two transitions identical. The arrow incoming to q_{aa} is then re-directed to q_a , and q_{aa} is eliminated from the list of states.



Step VII. The state q_{ab} is then folded into q_ϵ . The incoming arrow is re-directed to q_ϵ , and 1 is pushed back to state state output of q_{aba} . Since q_{ab} was merged with another state, it is eliminated from the machine. This leaves no other blue states in the machine, and it signifies that OSTIA completed the inference.



Step VIII. All blue states are now eliminated from the machine. However, the states that were never colored are still present. In SigmaPie, the last step included in the *OSTIA* algorithm is the elimination of the unaccessible states from the machine. After those steps are completed, we obtain the FST visualized below.



4.1.3 The unsuccessful example

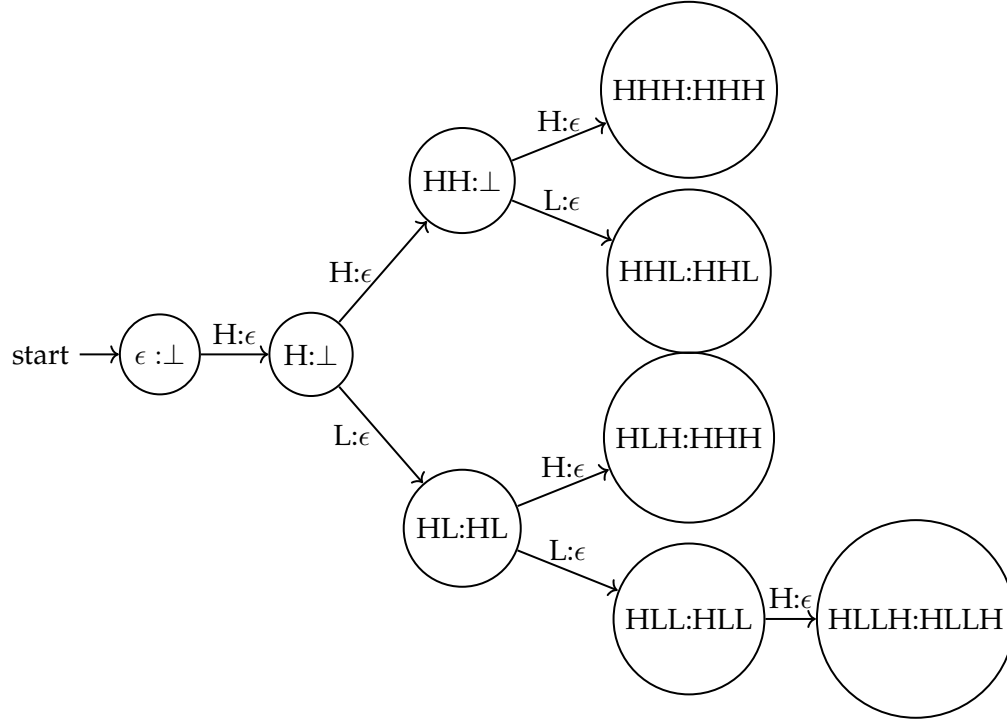
Now, consider a pattern of *unbounded tone plateauing* (UTP). In a pattern like that, a sequence of low tones is converted to high if surrounded by high tones. For

example, inputs HLH and $HLLLH$ are mapped to the outputs HHH and $HHHHH$, correspondingly. When a low tone L follows a high tone, it might be written as either L or H depending on the presence of another H anywhere further in the input. In other words, it requires an *unbounded lookahead*.

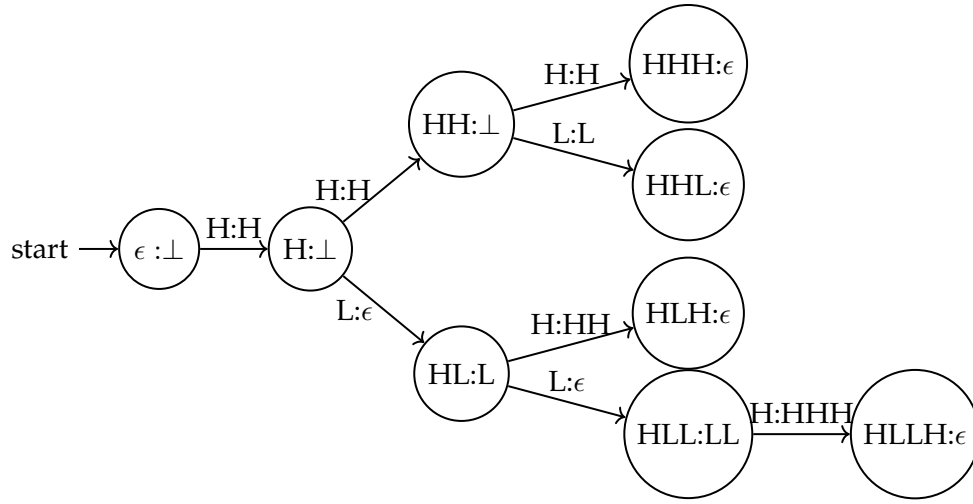
Patterns requiring lookahead, such as UTP, are called *unbounded circumambient processes* since the triggers are located on both sides of the undergoer, and they can be arbitrary far from it. Unbounded circumambient processes are not subsequential (Jardine, 2016a), and therefore it is expected that OSTIA is unable to capture UTP. I show that OSTIA fails to learn UTP with the following sample.

Sample = [(HHH, HHH), (HHL, HHL), (HL, HL), (HLH, HHH), (HLL, HLL), (HLLH, HHHH)]

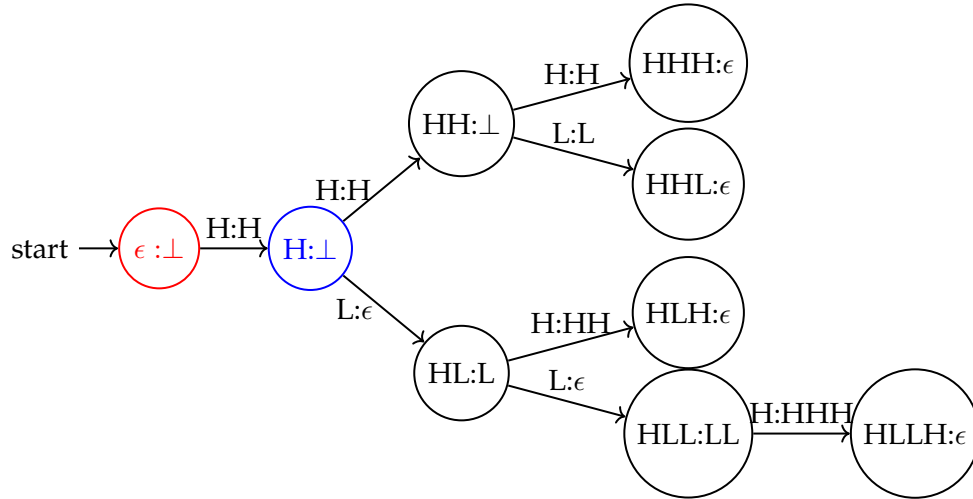
Step I. At first, consider a PTT corresponding to the given training sample S .



Step II. Then, as previously, let us push the state outputs from the leaf nodes closer to the initial state q_ϵ . The resulting PTT is onward.

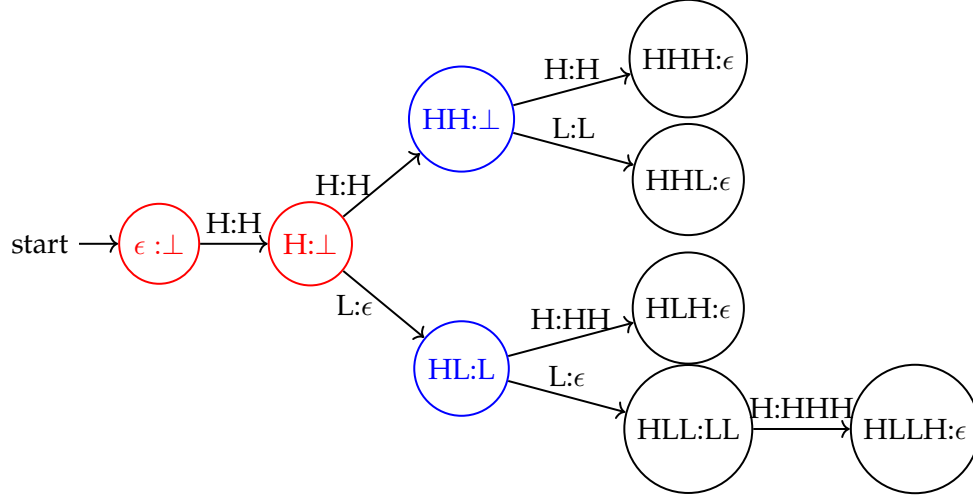


Step III. Next, we prepare to start folding states and sub-trees of the PTT into each other by coloring the states in red and blue. As previously, the initial state q_ϵ is colored red, and the state q_H available from q_ϵ in one step is colored blue.

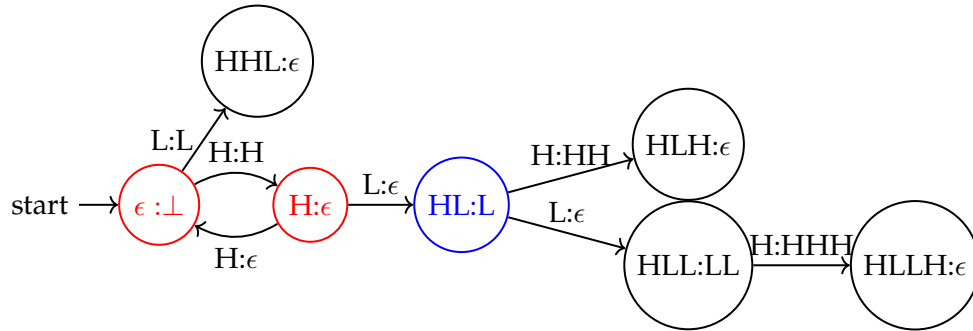


Step IV. The algorithm attempts to merge the blue state into the red state. However, it is not possible to fold the sub-tree of q_H into q_ϵ . It would cause the arrow $q_{HH} \xrightarrow{L:L} q_{HHL}$ to be changed to $q_\epsilon \xrightarrow{L:L} q_{HHL}$. It means that states q_{HHL} and q_{HL} need to be merged since both of them would be available from q_ϵ by reading L , but it is not possible because the state outputs of q_{HHL} and q_{HL} are different: ϵ

and L , correspondingly. Therefore the merge is rejected, and q_H is colored red. Its daughter nodes q_{HH} and q_{HL} are now blue.

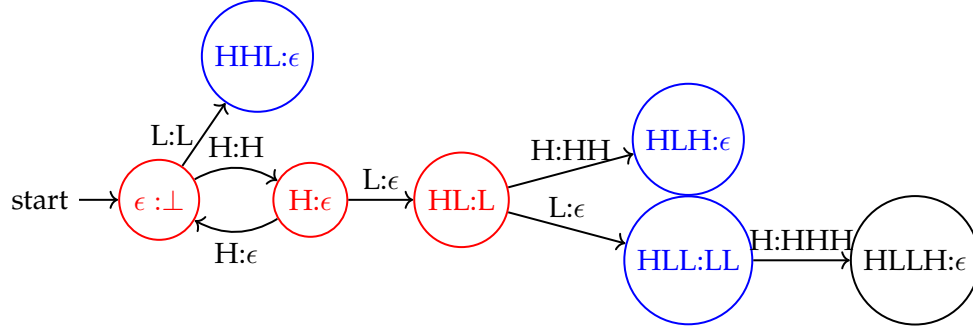


Step V. The state q_{HH} can be merged with q_ϵ . Indeed, the outgoing arcs reading and writing H are present in both of these states, and the outgoing arc from q_{HH} to q_{HHL} is now re-directed and originates from q_ϵ . All the incoming arcs into the state q_{HH} are re-directed to target q_ϵ . It created two outgoing from the state q_ϵ arrows reading H : $q_\epsilon \xrightarrow{H:H} q_H$ and $q_\epsilon \xrightarrow{H:H} q_{HHH}$, so q_{HHH} needed to be folded into q_H . Their state outputs are compatible since they are \perp and ϵ , therefore, the output of the state q_H was rewritten to ϵ .

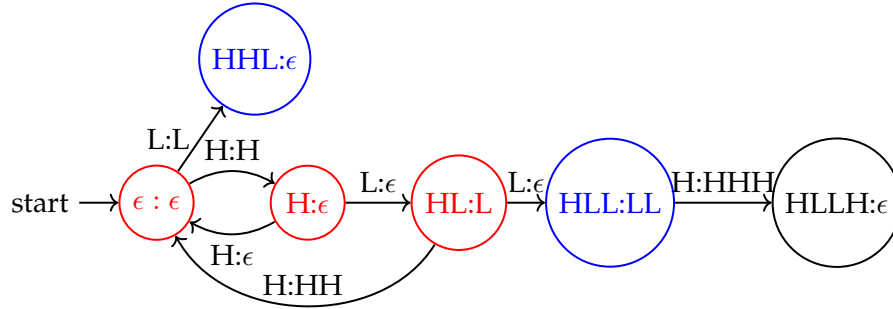


Step VI. The state q_{HL} can be merged with neither q_ϵ nor q_H because the arrows reading L bring the machine to states with different state outputs. While the state

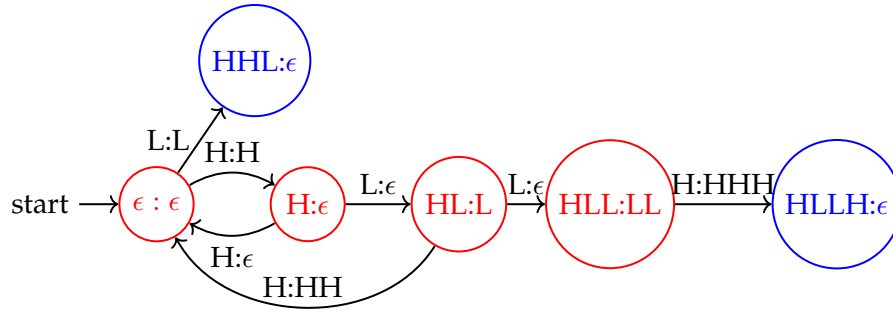
output of q_{HLL} is LL , q_{HL} and q_{HHL} output L and ϵ , correspondingly. Therefore q_{HL} is colored red, and its children q_{HLH} and q_{HLL} are now blue. Additionally, q_{HHL} is also blue because it originates from the red state q_ϵ .



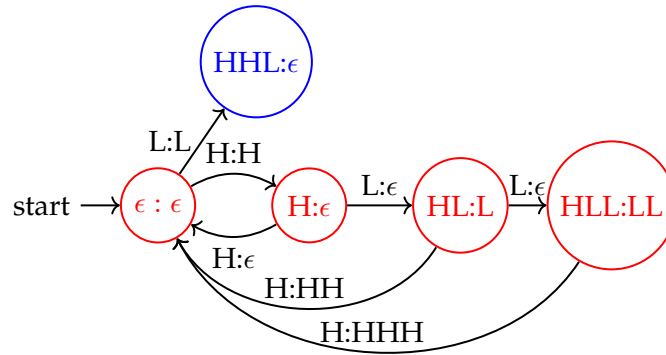
Step VII. The state q_{HLLH} is merged with q_ϵ , and the only arrow incoming in q_{HLLH} from q_{HLL} is now targeting q_ϵ . Since the state output of q_{HLLH} was ϵ and the one of q_ϵ was unknown, it is now re-defined as ϵ .



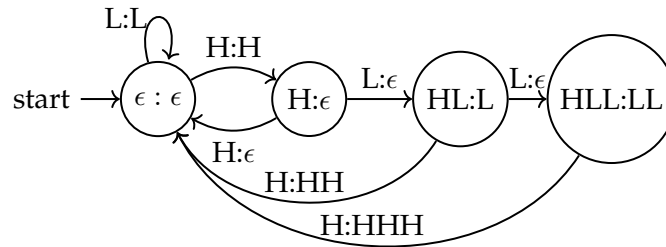
Step VIII. The state q_{HLL} cannot be merged with any red state. Its state output differs from those of all the red states. The state q_{HLL} is thus colored red, and its daughter state q_{HLLH} is blue.



Step IX. The state q_{HLLH} is then merged with q_ϵ , and the arrow incoming in it from the state q_{HLL} is re-directed. At this moment it is already clear that the algorithm failed to learn the UTP pattern. The learner memorized that one or two L need to be substituted by H if they are followed by H , but did not generalize it to an unbounded number of L s.




Step X. Finally, the only remaining state q_{HHL} is merged with q_ϵ , therefore, creating a loop that reads and writes L on that state. There are no other remaining blue states, and therefore the learner outputs the FST.



As one can see, this transducer does not represent the UTP pattern. For example, it would re-write an input string *HLHLH* as *HHHLH* by passing through the following states: $q_\epsilon \xrightarrow{H:H} q_H \xrightarrow{L:\epsilon} q_{HL} \xrightarrow{H:HH} q_\epsilon \xrightarrow{L:L} q_\epsilon \xrightarrow{H:H} q_H$. Indeed, UTP cannot be expressed using a subsequential function: it requires an unbounded lookahead (Jardine, 2016a).

4.2 Learning experiments

Here, I discuss the learning experiments that I used to explore the capacities of the OSTIA algorithm. Interestingly, OSTIA performed extremely well (100%) on one or more harmonies without blockers. However, the blocking effect showed itself as a challenge for the learner: the accuracy was falling drastically with the increasing length of the evaluated strings.

In this chapter, in contrast to the previous one, the learner extracts the “rewrite rules” instead of acquiring well-formedness conditions. Namely, the learner is given a pair of strings imitating the underlying representation (UR) and the surface form (SF), and its goal is to learn how to map one into another. This allows us to explore the capacities of OSTIA when challenged with natural language-like patterns discussed in the previous chapter: single and double harmonic systems with or without blockers, and others. The code behind the experiments is available on GitHub  (Aksënova, 2020d).

Note, however, the critical importance of the concrete implementation for the results of OSTIA. For example, the condition activating pushback is slightly different in various OSTIA’s pseudocodes. Oncina et al. (1993) in the original formulation of OSTIA execute the pushback step depending on the output of q_a transitions being a prefix of the corresponding q_b transitions. The core architecture of their learner, however, is a bit different from the one described in this section since their learner necessarily annotates the data with the

end-markers and uses this information for the inference steps. de la Higuera (2010) executes the pushback and folds the states q_a and q_b if all outgoing transitions from those states carry the same output. As one can see, in this case, the pushback step would not be useful: if the outputs of the transitions are identical, there is no disagreeing affix to be pushed further in the FST. In the errata for his book, de la Higuera (2011) changes that condition to depend on the color of the states accessible from q_b . The OSTIA version implemented and used in this chapter has its core architecture following de la Higuera (2010), but the condition is in line with Oncina et al. (1993). There is a multitude of different versions of pseudocodes, and, therefore, even more versions of the possible implementations. Thus the results obtained in this chapter are specific to the concrete interpretation of the fold and pushback conditions and need to be tested with different perspectives as well.

4.2.1 Experimental setup

As before, the experiments include 3 main steps: *data generation*, *learning*, and *model evaluation*.

At first, the training pairs were generated. For example, assume that we are trying to learn a single vowel harmony without blockers, where vowels are $A = \{a, o\}$, and the only consonant is x that is making this harmony long-distant. The training sample will then contain pairs such as $(xoxxAxAxxxA, xoxxoxxxxo)$ and $(axAxAxxA, axaxaxxa)$, where A refers to an underspecified harmonizing vowel. The left side of every pair contains the “underlying representation”, where only the value of the first vowel is established, and all the consecutive vowels are hidden and represented as the name of their harmonic set, in this case, A . The right side contains the “surface forms”, where all the vowels harmonize with each other. Such training pairs were then fed to OSTIA that outputted an FST representing the pattern.

To evaluate the performance of the obtained FST, following Gildea and Jurafsky (1996), I generated another set of pairs that are used as a testing sample. I provided the left-hand sides of those pairs to the FST as input and observed if the output strings were the same as the right-hand sides of the testing sample.

I used two types of testing samples: one includes strings of the same length as the ones that were used for training, and the second one contains strings that are twice longer than the latter. The test sets containing a longer strings helped me to explore how well OSTIA generalized the target pattern beyond memorization of the concrete shapes found in the training sample.

4.2.2 Target patterns

Among the patterns that I targeted while exploring the capacities of the learner, there were some local patterns such as word-final devoicing, different types of long-distance harmonies, and some circumambient and unattested patterns. Below I explain the parameters that I considered when choosing the learning experiments, see the summary in figure 4.1.

Long-distant The difference between local and long-distant processes is one of the very important distinctions for phonology. In the first case, the process is locally bounded, whereas, in the second one, it involves a potentially unbounded amount of the intervening material. As I show in chapter 3, this difference is crucial for strictly local models. They can evaluate local dependencies, but cannot capture long-distant ones. As an example of a purely local process, I use the phenomenon of word-final devoicing.

Includes blockers The blocking effect is widely discussed in the phonological literature. Harmony systems with and without blockers cannot always be modeled in the same way. For example, in chapter 3, I show that strictly piecewise models can

express multiple well-formedness conditions, but they cannot capture even simple cases of blocking effects. In the sample of explored datasets, 3 out of the 7 employed harmonies involve blockers.

Multiple processes A model cannot always handle multiple processes at the same time. For example, a single tier-based strictly local grammar can express a vowel or a consonant harmony, but it cannot capture both of them at the same time. There are a total of 4 harmonies that involve several harmonic spreadings, and 2 of them exhibit a blocking effect.

Different undergoers Some models can express several spreadings at once if all the harmonizing features are spread among the same sets of elements. For example, the only case when a tier-based strictly local grammar can capture the agreement in two features is when both features are affecting vowels. Therefore, 2 of the explored datasets present this type of a harmonic system, with and without a blocking effect.

Unbounded lookahead In subsection 4.1.3, I showed that processes that require an unbounded lookahead are not subsequential, and therefore cannot be learned by OSTIA. Therefore, I use 2 automatically generated datasets that exhibit circumambient patterns, and show that they cannot be learned by a subsequential learner.

Typologically attested Finally, I explore both typologically attested and unattested patterns. All types of harmonic processes are indeed attested in natural languages, as well as the unbounded tone plateauing. As an example of a typologically unattested pattern, I use the first-last harmony enforcing the agreement among the initial and the final vowels. Two datasets are exhibiting the first-last harmony: one of them includes an unbounded lookahead, and the other one does not.

long-distant	includes blockers	multiple processes	different undergoers	unbounded lookahead	typologically attested
<i>word-final devoicing</i>					
×	×	×	×	×	✓
<i>a single vowel harmony without blocking</i>					
✓	×	×	×	×	✓
<i>a single vowel harmony with blocking</i>					
✓	✓	×	×	×	✓
<i>several vowel harmonies without blocking</i>					
✓	×	✓	×	×	✓
<i>several vowel harmonies with blocking</i>					
✓	✓	✓	×	×	✓
<i>vowel harmony and consonant harmony without blocking</i>					
✓	×	✓	✓	×	✓
<i>vowel harmony and consonant harmony with blocking</i>					
✓	✓	✓	✓	×	✓
<i>unbounded tone plateauing</i>					
✓	×	×	×	✓	✓
<i>simple first-last harmony</i>					
✓	×	×	×	×	×
<i>complex first-last harmony</i>					
✓	×	×	×	✓	×

Table 4.1: Parameters of the explored natural language patterns.

4.2.3 Experiment 1: word-final devoicing

The rule of word-final devoicing prohibits underlyingly voiced obstruents from being voiced at the end of the word. So, for example, in German, word-final /b/, /d/, and /g/ are realized as [p], [t], and [k] (Brockhaus, 1995). While the word for ‘children’ is *Kinder*, its singular form is *Kin[t]*, i.e. the underlyingly voiced segment is voiceless at the end of the word.

Encoding As previously in Chapter 3, I used 3 elements to encode this pattern: *b* corresponding to voiced obstruents, *p* to their voiceless counterparts, and *a*

standing for any other sound. Like this, I generated pairs such as (*apab*, *apap*), (*aba*, *aba*) and (*app*, *app*), where every *b* of the first word of the pair is rewritten as *p* in the second one.

Results I produced 1,500 pairs exhibiting word-final devoicing and used them to build an FST using OSTIA. The obtained FST performed excellently on both testing samples. The first testing sample contained strings 1 to 5 characters long, similar to the training sample. The second testing sample contained longer strings, namely 5 to 10 characters. The perfect score of the FST on both tests signifies that it correctly acquired the pattern.

Pattern:	word-final devoicing
Training sample (info):	1500 pairs, 1 to 5 characters long
Testing sample 1 (info):	1000 pairs, 1 to 5 characters long
Testing 1, accuracy:	100%
Testing 1, predictions:	('apaap', 'apaap'), ('bpaab', 'bpaap'), ('abppp', 'abppp'), ...
Testing sample 2 (info):	1000 pairs, 5 to 10 characters long
Testing 2, accuracy:	100%
Testing 2, predictions:	('pappbab', 'pappbap'), ('aapppbapbb', 'aapppbapbp'), ...
Number of states:	2
Number of transitions:	5

Table 4.2: Results of OSTIA learning word-final devoicing.

The FST outputted by OSTIA is fully interpretable. It has 2 states and 5 transitions in-between them. In such a machine, the first state corresponds to the state of not observing *b*, and the second state to keeping *b* in memory. If *a* or *p* follow the memorized *b*, that *b* is outputted together with *a* or *p*. If another *b* follows a *b*, only one *b* is written. If *b* is a final character of the input sequence, *p* is written instead by the state output of q_b . The obtained machine exactly

corresponds to the target generalization, see figure 4.4.

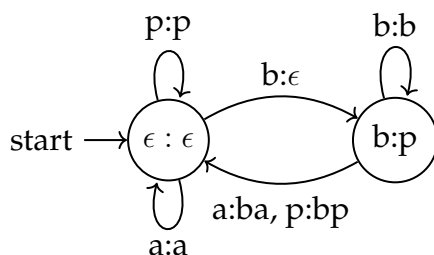


Figure 4.4: FST for word-final devoicing obtained by OSTIA.

4.2.4 Experiment 2: a single vowel harmony without blocking

The next challenge for OSTIA is to learn a pattern of a single vowel harmony without blocking. For example, in Finnish, vowels harmonize in fronting. Let us generalize a harmonic system where a single feature is spread without the possibility of being blocked.

Encoding Such a pattern can be generalized to a system where among vowels $A = \{a, o\}$, either o or a can occur in a surface representation, but not both. We can then refer to the underspecified vowel of the underlying representation as A . In order to make the spreading long-distant, x encodes the transparent element. This encoding defines pairs such as $(axAxxAAxx, axaxxaaxx)$ and $(xxoxxAxAxx, xxoxxoxx)$.

Results OSTIA induces the FST that correctly generalizes the pattern, therefore, scoring 100% on both testing samples. The inferred FST has 4 states and 14 transitions in-between them. It is depicted in figure 4.5: note, that this machine is not minimal, i.e. there are states that do not express any information significant for the rule of harmony. For example, q_x and q_{xx} only keep track of one or two x . After a was observed and written in the translation, the machine moves to the state q_a and any A from the input side is written as a on the output side. Instead,

observing o keeps the FST in the initial state q_e , and any following A is then re-written as o . Such an FST, although not minimal, correctly captures the intended harmonic system.

Pattern:	a single vowel harmony without blocking
Training sample (info):	5000 pairs, 1 to 10 characters long
Testing sample 1 (info):	1000 pairs, 1 to 10 characters long
Testing 1, accuracy:	100%
Testing 1, predictions:	('oAxxxAxxx', 'ooxxxoxxx'), ('xxaAxxAAxA', 'xxaaxxaaxa'), ...
Testing sample 2 (info):	1000 pairs, 15 to 20 characters long
Testing 2, accuracy:	100%
Testing 2, predictions:	('oAxAxAxxAAxxxAxAAxA', 'ooxooxxooxxxooxoo'), ...
Number of states:	4
Number of transitions:	14

Table 4.3: Results of OSTIA learning a single vowel harmony without blocking.

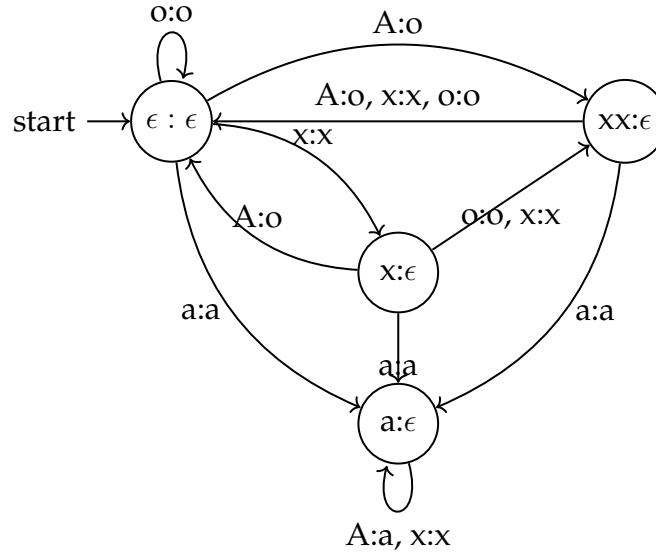


Figure 4.5: FST for a single vowel harmony without blocking obtained by OSTIA.

4.2.5 Experiment 3: a single vowel harmony with blocking

While the previous experiment explores the performance of OSTIA on a dataset exhibiting no blocking effect, this one adds it to the picture. In this case, while vowels within a word need to agree with respect to a certain feature, a blocker stops the spreading and only allows for its particular value after itself.

Encoding This pattern can be encoded as earlier, by using a class of vowels $A = \{a, o\}$, but now only a can be observed after the blocker f . As before, x stands for a transparent element. This defines pairs such as $(oxAA, oxoo)$, $(xaxxAxxA, xaxxaxxa)$ and $(xoxxAxfxxAx, xoxxoxxax)$.

Results In this case, the performance of the learner was not perfect. Namely, if faced with a testing sample where the length of the words is 1 to 10 characters, it performs with the accuracy of 99.2%. While mostly predicting the correct forms, for example, it rewrites the underlying form $oxAxAA$ as $oxoxoxxf$, incorrectly adding two extra characters to the end of the word. The accuracy falls when the length of the testing sample is increased to 15-20 characters: it only predicts 91.2% of the correct transformations.

The obtained machine is not correct since some of the surface forms that it predicts are wrong. However, the machine encoding the target generalization can be represented as a simple FST with 3 states, see figure 4.6 for the expected result. It raises a question of what obstructed the inference of that machine, and why it happened in any consecutive experiment targeting a harmonic system involving a blocking effect. I leave this issue aside for now, and come back to it further in the very end of the section 4.2.13.

Pattern:	a single vowel harmony with blocking
Training sample (info):	5000 pairs, 1 to 10 characters long
Testing sample 1 (info):	1000 pairs, 1 to 10 characters long
Testing 1, accuracy:	99.2%
Testing 1, predictions:	('oxAxAA', 'oxoxooxf'), ('xfxxaxAAx', 'xfxxaxaaxx'), ...
Testing sample 2 (info):	1000 pairs, 15 to 20 characters long
Testing 2, accuracy:	91.2%
Testing 2, predictions:	('xxxoxfAxxxffxAxxx', 'xxxoxfxxaaxxxffxaxxx'), ...
Number of states:	74
Number of transitions:	247

Table 4.4: Results of OSTIA learning a single vowel harmony with blocking.

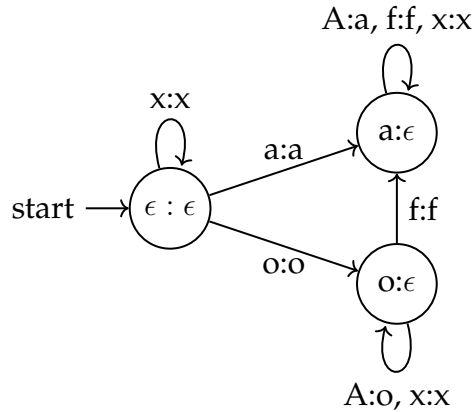


Figure 4.6: The expected FST for a single vowel harmony with blocking.

4.2.6 Experiment 4: several vowel harmonies without blocking

While the previous two experiments modeled spreadings of a single feature, this experiment targets the agreement in several features. For example, in Kyrgyz, vowels agree in backness and rounding. Abstractly, these features can be referred to as $[\alpha]$ and $[\beta]$. Thus all possible stems can be of 4 different types $[-\alpha, -\beta]$, $[-\alpha, +\beta]$, $[+\alpha, -\beta]$, and $[+\alpha, +\beta]$.

Encoding Now, let us assume that the class of vowels includes 4 elements $A = \{a, e, o, u\}$. Every one of these options encodes a possible type of vowel in a well-formed surface form. As previously, x is a transparent element. Such encoding defines pairs $(xxoxxAxAx, xxoxxoxxox)$ and $(xxxaxAxxx, xxxaxaxxx)$, among others.

Results Although the inferred machine is large (37 states and 90 transitions), it performs perfectly on both testing samples. In both cases, none of the forms predicted by the FST were disharmonic.

Pattern:	several vowel harmonies without blocking
Training sample (info):	5000 pairs, 1 to 10 characters long
Testing sample 1 (info):	1000 pairs, 1 to 10 characters long
Testing 1, accuracy:	100%
Testing 1, predictions:	('xxoxxAxAx', 'xxoxxoxxox'), ('xxxexAxxA', 'xxxexexxe'), ...
Testing sample 2 (info):	1000 pairs, 15 to 20 characters long
Testing 2, accuracy:	100%
Testing 2, predictions:	('xxoAxxxAAxxAxxxAAx', 'xxooxxxooxxoxxxoox'), ...
Number of states:	37
Number of transitions:	90

Table 4.5: Results of OSTIA learning several vowel harmonies without blocking.

Interestingly, the learned machine has 37 states, whereas smaller versions can easily be constructed, see figure 4.7. It requires additional investigation to understand the conditions that were not satisfied during OSTIA execution, therefore, yielding the machine with a greater number of states than possible.

4.2.7 Experiment 5: several vowel harmonies with blocking

The next task included learning Turkish vowel harmony. It enforces vowels to agree in backness and rounding. While all vowels within a word agree in backness, only

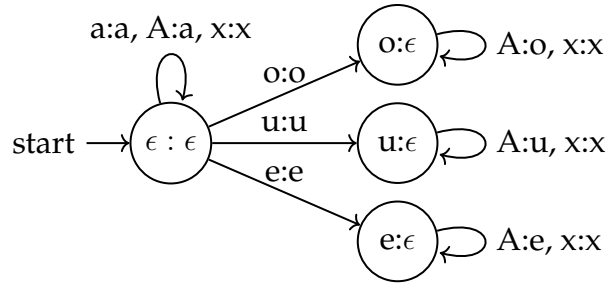


Figure 4.7: The expected FST for several vowel harmonies without blocking.

high vowels acquire the rounding value of a previous vowel. For example, the word *son-lar-ıın* ‘end-PL-GEN’ exemplifies that a non-high vowel from the plural suffix cannot acquire a rounding feature from the previous vowel, and therefore cannot transmit it to the following high vowel. However, in *son-un* ‘end-GEN’, the high vowel is realized rounded because it is preceded by a rounded vowel. In both words, all vowels agree in backness. In such a system, non-high vowels have a double nature: they are undergoers for the backness harmony, however, behave as blockers for the rounding one.

Encoding In the abstract representation of this pattern, it is impossible to use fewer vowels than already employed by Turkish. This harmony depends on 3 features (backness, rounding, and height), and therefore the minimum amount of vowels to encode it is $2^3 = 8$. Now, there are two classes of underspecified vowels in the URs of the strings: high (*H*) and low (*L*). The training pairs look like (*uxHxLxxLLxxHxxL*, *uxuxaxxaaxxuuxxa*): the initial underspecified vowel is high, and therefore it agrees in rounding with a previous back rounded vowel, becoming *u*. The next underspecified vowel is low and therefore is realized as unrounded *a*. After several other low vowels, another high one follows, but it is not rounded (*ı*) since it follows a non-round vowel. Similarly to the earlier experiments, *x* is transparent and makes this harmony long-distant. The rules below summarize how exactly the underspecified segments *H* and *L* are realized

in the SFs.

$$L = \left\{ \begin{array}{l} \text{'a' if the previous vowel is 'o', 'u', 'a' or 'u'} \\ \text{'e' if the previous vowel is 'ö', 'ü', 'e' or 'i'} \end{array} \right\}$$

$$H = \left\{ \begin{array}{l} \text{'o' if the previous vowel is 'a' or 'u'} \\ \text{'a' if the previous vowel is 'e' or 'i'} \\ \text{'o' if the previous vowel is 'o' or 'u'} \\ \text{'e' if the previous vowel is 'ö' or 'ü'} \end{array} \right\}$$

Pattern:	several vowel harmonies with blocking
Training sample (info):	5000 pairs, 1 to 10 characters long
Testing sample 1 (info):	1000 pairs, 1 to 10 characters long
Testing 1, accuracy:	97.9%
Testing 1, predictions:	(<i>'exLxHxL', 'exexixe'</i>), (<i>'xüxxLxHxxL', 'xüxxexüxxe'</i>), ...
Testing sample 2 (info):	1000 pairs, 15 to 20 characters long
Testing 2, accuracy:	87%
Testing 2, predictions:	(<i>'uxHxLxxLLxxHxxL', 'uxuxaxxaaxxuxxa'</i>), ...
Number of states:	107
Number of transitions:	318

Table 4.6: Results of OSTIA learning several vowel harmonies with blocking.

Results Given the data sample of 5,000 words 1 to 10 characters long, OSTIA did not learn this harmonic pattern completely. When evaluated using the test data of the same length as the one in the training sample, 97.9% of the predicted surface forms were as expected. On a test sample with longer words, the accuracy decreased to 87%. Such a model consistently makes incorrect predictions such as (*uxHxLxxLLxxHxxL*, *uxuxaxxaaxxuxxa*), where a rounded high vowel *u* occurs after a non-rounded low vowel *a*, instead of the expected high vowel *ü*.

Interestingly, the performance of the algorithm did not increase significantly with the increased size of the training sample.

4.2.8 Experiment 6: vowel and consonant harmonies without blocking

Now, let us consider two simultaneous yet independent harmonies. A frequent case is when there are two classes of harmonizing elements: consonants and vowels. Such a pattern is attested in several Bantu languages (Kikongo, Kiyaka, Bukusu a.o.).

Encoding Let us assume that vowels agree in $[\alpha]$, and consonants agree in $[\beta]$. Such system would need at least 2 vowels and 2 consonants: $A = \{a, o\}$ and $B = \{b, p\}$. Note that a special transparent element is not necessary since the presence of vowels makes the consonant harmony long-distant, and vice versa. This encoding defines pairs such as $(aApBAA, aappaa)$ and $(boABBABA, boobbobo)$, where in URs, every non-initial value of consonants and vowels is hidden under the name of the corresponding harmonic set.

Results The learner easily inferred the simultaneous vowel and consonant harmonies and scored 100% on both tests. The FST has 4 states and 16 transitions in-between them. Notice, that so far, the performance of OSTIA is 100% in every case when harmony does not include the blocking effect since it also performed extremely well on a single or double vowel harmonic systems earlier.

The inferred FST is visualized in figure 4.8. It has 4 states, and every state corresponds to a type of vowel and consonant in a stem. In particular, q_e corresponds to stems where the vowel and consonant values are a and p ; q_o corresponds to o and p ; q_b to a and b ; and, finally, q_{ob} encodes stems with o and b .

Pattern:	vowel and consonant harmonies without blocking
Training sample (info):	5000 pairs, 1 to 10 characters long
Testing sample 1 (info):	1000 pairs, 1 to 10 characters long
Testing 1, accuracy:	100%
Testing 1, predictions:	('oApBAA', 'ooppoo'), ('boABBABA', 'boobbobo'), ...
Testing sample 2 (info):	1000 pairs, 15 to 20 characters long
Testing 2, accuracy:	100%
Testing 2, predictions:	('opBAABAABBAABBAA', 'oppoopoopppoop'), ...
Number of states:	4
Number of transitions:	16

Table 4.7: Results of OSTIA learning vowel and consonant harmonies without blocking.

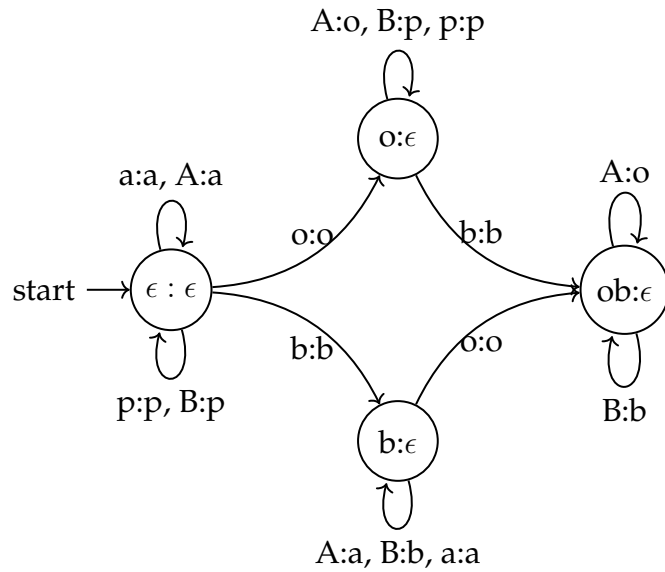


Figure 4.8: FST for vowel and consonant harmonies without blocking obtained by OSTIA.

4.2.9 Experiment 7: vowel and consonant harmonies with blocking

Let us add a blocking effect to the pattern from the previous experiment. Earlier we saw that OSTIA performs extremely well on data exhibiting a harmonic system that does not involve blockers. However, adding a blocking effect in all cases resulted in the obtained FST not scoring 100% on either of the training datasets.

Encoding Let us use the encoding as in the previous experiment, where a stem is able to “pick” one vowel from the set $A = \{a, o\}$ and one consonant from $B = \{b, p\}$. Additionally, I will introduce a blocker t , after which b cannot occur, and needs to be realized as p instead. Along with all pairs that are well-formed for the same pattern without the blocker, it also introduces pairs such as $(abABAtAABAB, ababataapap)$, where B is rewritten as b before the blocker since it inherited its value from the initial consonant b , however, B is realized as p after the blocker t .

Results The performance of OSTIA on this dataset is far from perfect. While it scores 96.4% on the testing sample that includes strings of the same length as the training ones, the accuracy falls to 77.4% when the length of the test words is doubled. It goes along with the previous results showing that OSTIA fails to generalize a blocking effect.

4.2.10 Experiment 8: unbounded tone plateauing

Now, consider a circumambient pattern of *unbounded tone plateauing* (UTP). This pattern is observed in some Niger-Congo languages such as Luganda, where all low tones (L) are realized as high (H) if they are surrounded by high tones. As section 4.1.3 shows, that pattern is not learnable by OSTIA since circumambient dependencies require an unbounded lookahead, and therefore cannot be expressed as subsequential transducers (Jardine, 2016a).

Pattern:	vowel and consonant harmonies with blocking
Training sample (info):	5000 pairs, 1 to 10 characters long
Testing sample 1 (info):	1000 pairs, 1 to 10 characters long
Testing 1, accuracy:	96.4%
Testing 1, predictions:	(‘aApABB’, ‘aapapp’), (‘pBtaA’, ‘pptaa’), (‘tpaBBAtAt’, ‘tpappatattppa’), ...
Testing sample 2 (info):	1000 pairs, 15 to 20 characters long
Testing 2, accuracy:	77.4%
Testing 2, predictions:	(‘pBaBABBABttttABABBAB’, ‘ppapappapttttopoppop’), ...
Number of states:	101
Number of transitions:	406

Table 4.8: Results of OSTIA learning vowel and consonant harmonies with blocking.

Encoding This process involves low tones (*L*) changing their value to high (*H*) if they are surrounded by high tones. To encode this pattern, we can simply create pairs where the underlying stretches of *L*s are realized as *H* in the surface forms if surrounded by high tones, such as (*LHHLHH*, *LHHHHH*). In all other cases, the underlying and the surface representations match.

Pattern:	unbounded tone plateauing
Training sample (info):	5000 pairs, 1 to 10 characters long
Testing sample 1 (info):	1000 pairs, 1 to 10 characters long
Testing 1, accuracy:	100%
Testing 1, predictions:	(‘HHHHL’, ‘HHHHL’), (‘LLLHL’, ‘LLLHL’), (‘LHHLHH’, ‘LHHHHH’), ...
Testing sample 2 (info):	1000 pairs, 15 to 20 characters long
Testing 2, accuracy:	94.9%
Testing 2, predictions:	(‘HLLLLLLLLHHLHHH’, ‘HLLLLLLLLHHHHHH’), ...
Number of states:	32
Number of transitions:	64

Table 4.9: Results of OSTIA learning UTP.

Results The obtained FST performs extremely well on the test set where the length of the words is the same as in the training sample. However, the accuracy falls to 94.9% when the length of the test words is doubled. This suggests that instead of capturing the pattern, the resulting machine simply memorized long substrings of tones that can be observed in the input. Indeed, UTP cannot be captured as a subsequential machine.

4.2.11 Experiment 9: a “simple” first-last harmony

Now, let us consider learning a typologically unattested pattern such as *first-last harmony* (Lai, 2015). This pattern enforces agreement among the first and the last vowels, while nothing else needs to agree. First, let us consider a case when vowels are always the first and last elements of the word.

Encoding The encoding involves vowels a and o , and a transparent element x . In this representation of the pattern, words always start and end with a vowel, therefore the sample pairs look as follows: $(oaoxaa, oaoxao)$, $(axooxa, axooxa)$, etc. If a final vowel of the underlying representation disagrees with the initial vowel, it is rewritten to match the initial vowel.

Results Such a pattern is easily induced by OSTIA. The obtained model scores 100% on both test datasets. The FST is pretty small and therefore interpretable: it has only 5 states and 14 transitions.

This FST is visualized below. After starting to process the input string in q_{ϵ} , it moves to either q_a or q_o depending on the first vowel that it reads. States q_o and q_{ao} handle strings that start and end with o ; similarly, the agreement within words that start with a is enforced by states q_a and q_{oa} . Note the similarity of these two branches with the way the word-final devoicing was encoded earlier in FST 4.4. While the disagreeing vowel is deleted from the transitions incoming to the states

Pattern:	a “simple” first-last harmony
Training sample (info):	5000 pairs, 1 to 6 characters long
Testing sample 1 (info):	1000 pairs, 1 to 6 characters long
Testing 1, accuracy:	100%
Testing 1, predictions:	(‘oaoxaa’, ‘oaoxao’), (‘axooxa’, ‘axooxa’), (‘oo’, ‘oo’), ...
Testing sample 2 (info):	1000 pairs, 10 to 15 characters long
Testing 2, accuracy:	100%
Testing 2, predictions:	(‘aoaxoaaaoaaaxaa’, ‘aoaxoaaaoaaaxaa’), ...
Number of states:	5
Number of transitions:	14

Table 4.10: Results of OSTIA learning a “simple” first-last harmony.

q_{oa} and q_{ao} , that vowel is returned if it is not final. If that vowel was, in fact, the last element of the input word, the other, “agreeing” vowel is written instead by the corresponding state output.

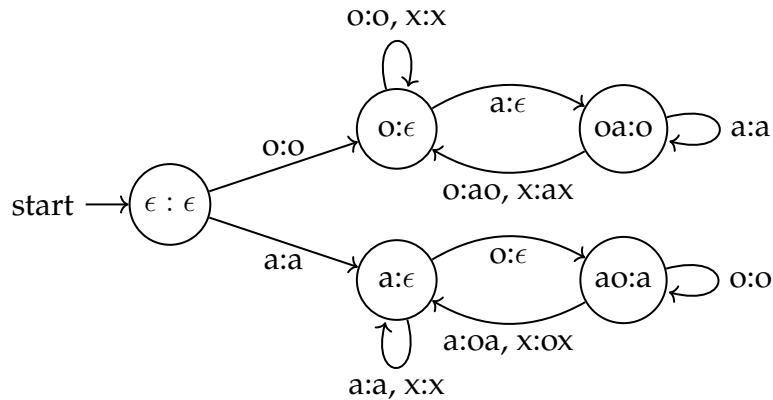


Figure 4.9: FST for a “simple” first-last harmony obtained by OSTIA.

4.2.12 Experiment 10: a “complex” first-last harmony

The previous experiment targeted a pattern of the first-last harmony, where all words started and ended with a vowel. Indeed, OSTIA learned that pattern with the accuracy of 100%. Now, let us challenge the learner with a more complicated version of this rule: in this case, words are also able to start and end with a transparent element, however, the first vowel of the word still agrees with the final vowel.

Encoding The training sample, apart from including everything possible for the previous experiment, also includes pairs where the forms have a sequence of initial or final *x*. So, for example, pairs such as (*xxoxoaooaax*, *xxoxoaooaax*) and (*axxoxoxxx*, *axxoxaxxx*) are added to the training sample. Now, not all input-output pairs begin and end with a vowel.

Results The performance of OSTIA is not perfect on either of the tests. The score is 98.6% on the test set that includes strings of the same length as the training sample, while it falls to 28.1% on a test set that includes longer words. While OSTIA learned the previous version of this pattern, it failed to generalize a more complicated one. Indeed, this result is expected since this case of first-last harmony requires an unbounded lookahead: there can be an unbounded amount of intervening material between the final vowel and the end of the word. Patterns like this are not subsequential.


4.2.13 Summary of the results

In this section, I explored automatic extraction of different types of harmonic or other assimilation processes, both attested and non-attested in natural languages. These patterns were word-final devoicing, single and several vowel harmonies with and without blocking, independent vowel and consonant harmonies with

Pattern:	“complex” first-last harmony
Training sample (info):	5000 pairs, 1 to 12 characters long
Testing sample 1 (info):	1000 pairs, 10 to 29 characters long
Testing 1, accuracy:	98.6%
Testing 1, predictions:	(‘oxaxooxx’, ‘oxaxoo oxxaxx ’), (‘xxaoxxaaxx’, ‘xxaoxxa axx ’). ...
Testing sample 2 (info):	1000 pairs, 15 to 20 characters long
Testing 2, accuracy:	28.1%
Testing 2, predictions:	(‘ooxaxoxoo axxxxxx ’, ‘ooxaxoxoo x ’), ...
Number of states:	79
Number of transitions:	235

Table 4.11: Results of OSTIA learning a “complex” first-last harmony.

and without blocking, unbounded tone plateauing, and different kinds of first-last harmony.

In the previous chapter I evaluated how learners acquire the well-formedness conditions imposed on the given training sample. Here, I discuss learning the observed *transformations* using the OSTIA algorithm. The training sample then consists of pairs of examples, where the first element of the pair stands for the underlying representation, and the second element is the corresponding surface form. Given such automatically generated sets of examples (see the code on GitHub  (Aksénova, 2020d)), I explored if OSTIA is capable of extracting the generalized rule from it.

As the first step, I generated the training data using the extension of the codebase explained in the previous chapter in 3.1.3. Then, I provided those pairs as a training sample to OSTIA. To test the performance of the learned FST, I generated another set of pairs and provided the left sides of those pairs as input to that FST. The accuracy of the learned model is the percentage of times the FST outputted the same surface form as the right-hand side of the generated test pair.

I tested the obtained model on two types of test samples: in the first test sample, the strings are of the same length as in the training sample, and in the second one, they are approximately twice as long. The first test sample evaluates the “baseline” performance of the learned models, while the second one uses longer strings to see how well the pattern was generalized.

Experiments	$ \mathbf{w}_{\text{train}} = \mathbf{w}_{\text{test}} $	$ \mathbf{w}_{\text{train}} = 2 \times \mathbf{w}_{\text{test}} $
E1: word-final devoicing	100%	100%
E2: a single vowel harmony without blocking	100%	100%
E3: a single vowel harmony with blocking	99.2%	91.2%
E4: several vowel harmonies without blocking	100%	100%
E5: several vowel harmonies with blocking	97.9%	87%
E6: vowel and consonant harmonies without blocking	100%	100%
E7: vowel and consonant harmonies with blocking	96.4%	77.4%
E8: unbounded tone plateauing	100%	94.9%
E9: “simple” first-last harmony	100%	100%
E10: “complex” first-last harmony	98.6%	28.1%

Table 4.12: Results of the learning experiments using OSTIA.

Table 4.12 provides a summary of the results discussed in this section. The first column describes the results of testing on the same-length testsets, while the second one summarizes the performance of the models on the strings that are approximately twice longer. Out of 10 experiments, OSTIA generalized 5 patterns extremely well, so its performance is 100% on both test sets. These experiments were word-final devoicing, single and several vowel harmonies, independent vowel and consonant harmony, and a “simple” version of the first-last harmony. Interestingly, none of the successful experiments included a pattern with a blocking effect. In fact, on the datasets with blocking, OSTIA performed significantly worse, scoring 91.2%, 87%, and 77.4% during the evaluations that used longer words. Unbounded circumambient processes are not subsequential

(Jardine, 2016a), so OSTIA expectedly did not learn UTP. Finally, the “complex” version of first-last harmony is also beyond the capacities of this learner because it requires an unbounded lookahead: only 28.1% of the predicted transformations are indeed correct.

The current implementation of OSTIA cannot capture the blocking effect, as the experiments 3, 5 and 7 show. Theoretically, this can be caused by 3 different reasons: the inability of the algorithm to learn the blocking effect, the absence of the crucially important data points, and the inability of the current implementation of the algorithm to learn the blocking effect. The algorithm behind OSTIA is proven to be correct by Oncina et al. (1993). The average error was not decreasing with the increased number of the training examples, and that shows that the problem is not rooted in the absence of some pairs of examples. It only leaves one source of the issue, namely, the concrete implementation of the algorithm. Indeed, as I discuss in the beginning of the section 4.2, different pseudocodes of OSTIA have different conditions behind the activation of the pushback module. In future work, re-implementing OSTIA with different versions of that condition is necessary to find the concrete implementation that can learn the full class of subsequential functions in practice².

4.3 Beyond OSTIA

So far, I presented OSTIA as the only way to learn mappings. Alternatively, one might want to either specify OSTIA given some concrete assumptions or to use other transduction learners. OSTIA learns *total* functions, i.e. functions that are defined for all possible values of their input. However, natural language input to

²For example, one can attempt the pushback and fold of the states q_a and a_b if the outputs of the transitions with the same input symbol originating in those states have a non-empty common prefix. Preliminary results show that OSTIA implementing this condition is capable of learning the blocking effect; however, it does not perform well on local processes.

output mappings are not total: not all inputs can be mapped to their output counterparts because some forms simply do not exist. This results in the learner never having a chance to satisfy a certain condition, and therefore it might not converge on the target FST. Learners can be redefined with respect to the natural language restrictions in ways that use negative data, require a deterministic finite-state acceptor (DFA) corresponding to the input or output, or define different types of locality. In this section, I discuss available extensions of OSTIA (OSTIA-D/R, OSTIA-N), other learning algorithms (SOSFIA, ISLFLA, OSLFIA), and propose some ideas for further learners. To the best of my knowledge, the performance of other transduction learners on different datasets was not explored as of now, and it would be an interesting project to be carried out in future.

4.3.1 Specifying OSTIA

Earlier in this chapter, I discussed the main version of the OSTIA algorithm: it builds a prefix tree using the input sides of the training sample, and then folds some states one into another, therefore generalizing the pattern. However, the FST extraction can be greatly simplified by providing some extra information about the shape of the input or output strings, as it is done in OSTIA-D, OSTIA-R, and OSTIA-DR, or by giving a sample of negative strings, as in OSTIA-N.

A transducer encodes a mapping. But in some cases, this mapping is not defined for *any* input string, but rather for a subset of the possible strings. If the constraints on the input are available a priori, one might use a form of OSTIA that encodes **Domain** knowledge, or OSTIA-D (Oncina and Varó, 1996). It takes as input not only the training sample but also the DFA describing the language of the input strings. If that information is available for the output strings of the intended mapping, then the **Range** is defined a priori. OSTIA-R requires a DFA describing the output language (Castellanos et al., 1998). Consequently, OSTIA-DR takes advantage of both domain and range DFAs (Oncina and Varó, 1996).

As another type of prior knowledge, OSTIA-N uses the Negative data (Oncina and Varó, 1996). Such a learner is given a set of well-formed pairs, as well as a set of input strings that should *not* be translated by the learned machine. While merging the states, OSTIA-N checks that none of the prohibited inputs obtained a translation.

4.3.2 Fixing outputs of some input symbols

We might have information about some of the outputs. Namely, the outputs of some input symbols can be fixed thus accelerating the convergence if the learner explores all possible options. For example, assume observing a pair $(sim, seen)$ in the training sample. Based on exclusively this pair, we can construct a total of 35 prefix tree-shaped FSTs with different output values of input symbols s , i and m , see some of the machines in 4.10. Some of these FSTs output the translation *seen* as soon as they read s , some of them distribute the string among different transitions, some of them only have this string as the state output of q_{sim} , and so on.

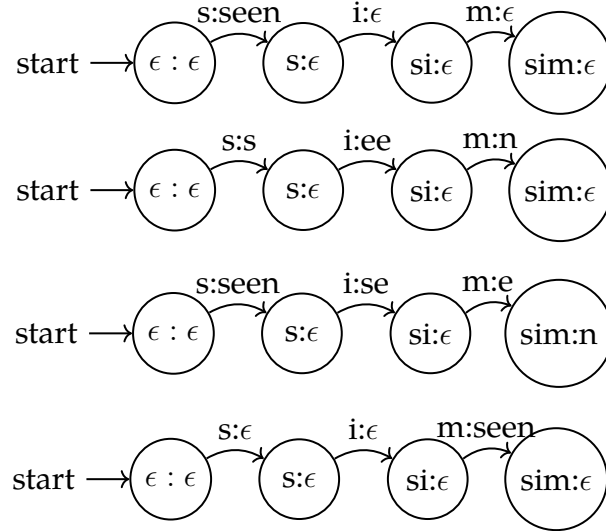


Figure 4.10: Some of the FSTs that can be built from the pair $(sim, seen)$ in the “unbiased” way; to be contrasted with the following figure.

However, if it is known in advance that some of the outputs can be fixed, the number of such 4-state FSTs processing $(sim, seen)$ decreases significantly. For example, if the output of the input symbol s is fixed to s , only 10 machines are possible. In all of them, a transition $q_\epsilon \xrightarrow{s:s} q_s$ is fixed to only read and output s . The number of possible machines then decreases to 10, see figure 4.11.

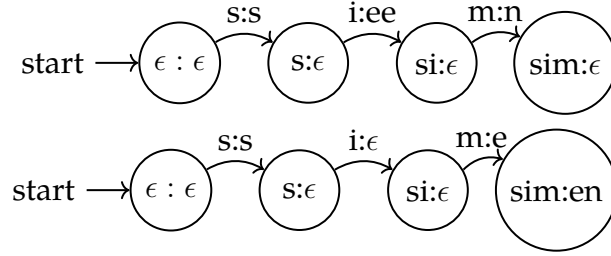



Figure 4.11: Some of the FSTs that can be built from the pair $(sim, seen)$ if the output of the input symbol s is fixed to the output symbol s .

Instead, if we fix the output of the symbol i to ee , there will be only 2 possible machines that employ 4 states. The output of s would be s in both cases, but the way of obtaining n in the translation will differ: in one case, its source is the transition $q_{si} \xrightarrow{m:n} q_{sim}$, and in another case, it is the state output of q_{sim} that yields n .

See the implementation of OSTIA with the possibility of fixing outputs of some input symbols on GitHub  (Aksënova, 2020e). However, this solution is not applicable if there are processes happening across the “fixed” symbol. If a metathesis occurs across some segment the output of which is fixed, extraction of this pattern becomes tricky. For example, in an Austronesian language Hawu, some vowels undergo metathesis across a consonant (Blust, 2012). In this case, fixing the output of that consonant will make it more complicated for the learner to discover the pattern.

4.3.3 Other transduction learners

In this chapter, I only discussed the learning results obtained by conducting toy learning experiments using OSTIA. However, there are several other subsequential learners available in the literature. One of them presents a different from OSTIA approach to the induction of subsequential transducers from the set of training pairs, and the other two rely on the particular assumptions about the shape of the dependencies encoded in the mapping.

Structured Onward Subsequential Function Inference Algorithm (SOSFIA) proposed by Jardine et al. (2014) assumes the existence of the k -local DFA that helps to navigate through the input sides of the training pairs. The learner then induces a subsequential function that reads the input strings one-by-one and outputs their translations. The obtained FST is onward, i.e. at any step, the biggest known part of the output string is produced.

ISLFLA and OSLFIA encode assumptions about the sources of dependencies. **Input Strictly Local Function Learning Algorithm** (ISLFLA) assumes that the translations only depend on the input string, i.e. there is enough information in the input itself to construct the translation (Chandlee et al., 2014). **Output Strictly Local Function Inference Algorithm** (OSLFIA), on the contrary, considers both the input string and the currently known output prefix as sources of the information for the prediction of the next output symbol (Chandlee et al., 2015). See chapter 2 for the examples of input/output local functions. Both learners require some integer k to be known a priori, and their first step is a construction of k -local DFA accepting input or output strings.

An enlarged set of experiments needs to be used when exploring SOSFIA, ISLFLA and OSLFIA. The expected result is that SOSFIA will learn all subsequential mappings (word final devoicing, all types of harmonies, and the “simple” case of vowel harmony). However, OSLFIA and ISLFLA cannot learn processes where a potentially unbounded number of segments can intervene

in-between two agreeing elements, because such processes are neither ISL nor OSL (Chandlee et al., 2015). It would imply that those two learners only capture the word-final devoicing, and fail on all other experiments, see figure 4.13. Extending the list of the experiment by different local dependencies, such as metathesis, epenthesis, and deletion, would allow one to explore the practical capabilities of those learning algorithms better.

Experiments	SOSFIA	OSLFIA	ISLFLA
<i>E1: word-final devoicing</i>	👍*	👍*	👍*
<i>E2: a single vowel harmony without blocking</i>	👍*	✖*	✖*
<i>E3: a single vowel harmony with blocking</i>	👍*	✖*	✖*
<i>E4: several vowel harmonies without blocking</i>	👍*	✖*	✖*
<i>E5: several vowel harmonies with blocking</i>	👍*	✖*	✖*
<i>E6: vowel and consonant harmonies without blocking</i>	👍*	✖*	✖*
<i>E7: vowel and consonant harmonies with blocking</i>	👍*	✖*	✖*
<i>E8: unbounded tone plateauing</i>	✖*	✖*	✖*
<i>E9: “simple” first-last harmony</i>	👍*	✖*	✖*
<i>E10: “complex” first-last harmony</i>	✖*	✖*	✖*

Table 4.13: Predicted results (marked as *) of the learning experiments using SOSFIA, OSLFIA and ISLFLA learning algorithms.

4.3.4 Learning groups of transducers

Currently, the transduction learners proposed in the literature have a goal of constructing a working generalized transducer based on the finite data sample. Different algorithms perform different strategies of pattern recognition. However, all of them have an assumption that the input sample is sufficiently representative. This is a very strong requirement: a fully representative data sample is not always

available. For this reason, it is possible to think of an algorithm that instead of extracting a single machine, builds a class of machines that behave equally with respect to the training sample but are non-equivalent otherwise.

For example, consider a learning algorithm for a group of equivalent yet not identical input local FSTs. Such FSTs only rely on the information available in the input to predict the output. A learner could start by taking an input k -local subsequential transducer template with unfilled outputs of the transitions. Then the transducer reads the input strings and saves all substrings of the corresponding translation string in the outputs of the transitions taken to read the input string. If the transition was taken before, the algorithm intersects the set of substrings of the current translation with the set of saved candidates, therefore, leaving only the candidates that are consistent throughout the whole training sample. After the training data is processed, the algorithm builds a set of transducers based on the obtained guesses for every transition.

For example, when the pair (ab, aab) is provided as the input to the learner, two out of many guesses about the target FST would be either (1) outputting ab if we read b after an a , or (2) a is translated to aa , and b stays intact, see figure 4.12. However, as soon as the training pair (a, aa) is encountered, the first machine from Figure 4.12 is rejected, leaving the second machine as the applicable candidate.

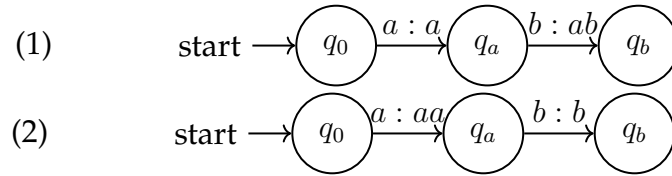



Figure 4.12: Possible guesses of the transition that can be built after observing the pair (ab, aab) .

Similarly to OSTIA-D and SOSFIA, such a learner needs to know a priori the DFA corresponding to the input side of the language. Therefore, assume that we have the input k -local DFA pre-initialized. Every transition q of that DFA

corresponds to the set Ω_q , and that set is empty upon the initialization. Then, we read one pair from the training sample. If Ω_q of that transition is empty, we fill every state activated while reading the input string with all the substrings of the right side of the training pair. If Ω_q is not empty, we intersect all substrings of the newly encountered output string with the guesses that those transitions contain. After the whole training sample is processed, we can build all possible transducers based on the remaining guesses in Ω of the transitions. Finally, we can run the input sides of the training pairs through the transducers again to validate that it predicts only correct outputs.

At the moment, the algorithm is implemented  (Aksënova, 2020a), but it has not been extensively tested or optimized yet. Further work on this algorithm includes elaborate presentation of the pseudocode, evaluating its time complexity, improving the implementation, and testing its performance on language data, as well as presenting the proof of its correctness.

4.4 Learning processes: summary

In this chapter, I discussed the automatic extraction of subsequential finite-state transducers from a sample of input-output pairs. Namely, I summarized the main steps of the OSTIA algorithm, presented two walk-through examples, demonstrated the results of the learning experiments, and proposed ways to go beyond OSTIA's performance.

The learning experiments targeted extraction of natural language-like patterns. The training samples exhibited in simplified ways such typologically attested processes as word-final devoicing, one or multiple harmonies with and without blocking effect, and unbounded tone plateauing. Additionally, OSTIA was presented with a dataset exemplifying the unattested pattern of first-last harmony. OSTIA successfully learned the generalization behind word-final devoicing, all

harmonies without a blocking effect, and a “simple” version of first-last harmony. If the target pattern exhibited a blocking effect, however, it significantly decreased the accuracy of OSTIA. Finally and expectedly, the generalizations behind tone plateauing and the “complex” version of first-last harmony remained unlearned since they involve an unbounded lookahead, so these patterns are not subsequential. I provided the obtained FSTs where applicable, but that was not enough to explain the learning outcomes of some of the experiments.

A lot of questions remain unanswered. Firstly, it is unclear what makes it impossible for OSTIA to learn even a simple case of harmony with a blocking effect. Second, it is also important to test OSTIA on non-length preserving processes and other phonological phenomena such as epenthesis, deletion, and metathesis. Finally, there are learners such as ISLFLA and OSLFIA that induce input and output-local functions (Chandlee et al., 2014, 2015), and their performance on natural language patterns needs to be evaluated as well.

In this chapter, I discussed learning *transformations* that capture processes which change underlying representations into the corresponding surface forms. Previously, chapter 3 discussed learners that are capable of modeling *well-formedness conditions* imposed on languages. Following this line of research helps us to explore the ways to create computational models of language, therefore giving an opportunity for insights into how languages work.


Chapter 5

Conclusion and future work

The last decade was very fruitful in the field of subregular research. New classes of subregular languages and mappings were uncovered for modeling natural language phenomena, and new learning algorithms were developed for these classes. The subregular approach has been successfully applied to phonotactics (Heinz, 2010a), rewrite processes in phonology and morphology (Chandlee, 2014), and even syntactic constraints over tree structures (Graf, 2018b). However, the rapid pace of the theoretical research has not been matched when it comes to engineering considerations. Many of the proposed learning algorithms have not been implemented before, and as a result, their performance on concrete data sets was not known.

In this final chapter, I summarize the impact of my dissertation and propose several different ideas and directions that can help to improve the balance between theory and practice in the subregular field.

5.1 Summary of the results

In my dissertation, I targeted the above problem of the theory-practice misbalance from two perspectives. First, I implemented a package *SigmaPie*  that includes

different subregular learners, scanners, sample generators, and some other functions useful for the subregular research (Aksënova, 2020b). Second, I explored the performance of several learning algorithms on datasets illustrating such widespread linguistic patterns as word-final devoicing, harmonies of different types, tone plateauing, and others.

5.1.1 *SigmaPie*

This package mostly focuses on tools for working with subregular languages and grammars, but also includes *OSTIA*, a learner for subsequential transducers (Oncina et al., 1993; de la Higuera, 2010). The functionality of the package includes various functions that can be used to simplify the practical work with subregular languages. *Learners* extract subregular grammars from the provided dataset. *Scanners* evaluate the well-formedness of data items with respect to the given grammar. *Sample generators* produces a dataset of the required size that follows the given grammar. *Polarity switchers* convert the grammars from positive to negative, and the other way around. Finally, *FSM constructors* build a finite state machine based on the given grammar. The implemented learning algorithms for strictly piecewise, strictly local, tier-based strictly local, and multi-tier strictly local languages are proposed in (Heinz, 2010b; Jardine and McMullin, 2017; McMullin et al., 2019).

The package is implemented in Python 3, and uses the copyleft open-source GNU General Public License v3.0. It is available on PyPI and in pip.

5.1.2 Tool-assisted learning experiments

Apart from the software component, this dissertation also includes the experimental part discussed in chapters 3 and 4. Namely, I designed artificial datasets exhibiting different linguistic patterns, and extracted the underlying

grammars using tools available in *SigmaPie*. The conducted experiments were of two types: the first type targeted learning well-formedness conditions, while the second one explored generalizing the rules of rewrite processes.

The datasets exhibiting the effect of well-formedness rules are lists of words, where those words never violate the encoded well-formedness condition. For example, if the condition is vowel harmony, the generated dataset included only the forms where all vowels agreed in the harmony feature. Alternatively, in case of the word-final obstruent devoicing, the dataset only included words that did not violate that rule.

The training samples for the processual learners included pairs of strings, representing the underlying representation (UR), and the corresponding surface form (SF) after the rule was applied. As a toy example, assume that the inventory of vowels only contains *a* and *o*, and the progressive vowel harmony enforces the agreement in rounding. Valid SFs then include either only *o* vowels or only *a* vowels. The corresponding URs have all the non-initial vowels hidden (for example, represented as *A*), and only the initial vowel is specified as *a* or *o* to trigger the agreement. In the used encoding scheme, the number of representations of vowels in the URs depends on the number of features that are important for the behavior of the vowels regarding the harmony. Namely, the number of vowel representations in URs is 2^f , where f is the number of such features. For example, in the mentioned toy pattern, no features were significant, and therefore there is only $2^0 = 1$ underspecified vowel. In some patterns, such as Turkish (see section 4.2.7), height plays a role in the behavior of the undergoers, and therefore there are $2^1 = 2$ underspecified vowels. In such a way, these pairs encode the URs that contain the underspecified elements, and the SFs where those elements are specified.

The results of the learning experiments are summarized in the figure 5.1. According to de la Higuera (2010), the task of grammatical inference algorithms is

to *constantly* predict the next correct element. Therefore, the algorithm did not fully learn the pattern if it is still making errors, no matter how negligibly rare those errors are. Figure 5.1 only indicates successful learning of the pattern if the achieved accuracy was 100%. A gradient representation of the experimental outcomes can be found in figures 3.58 and 4.12.

Experiments	Well-formedness				Transformations
	SP	SL	TSL	MTSL	OSTIA
<i>E1: word-final devoicing</i>	✗	👍	👍	👍	👍
<i>learning from raw German data</i>	✗	👍	👍	👍	
<i>E2: a single vowel harmony without blocking</i>	👍	✗	👍	👍	👍
<i>learning from raw Finnish data</i>	👍	✗	✗	👍	
<i>E3: a single vowel harmony with blocking</i>	✗	✗	👍	👍	✗
<i>E4: several vowel harmonies without blocking</i>	👍	✗	👍	👍	👍
<i>E5: several vowel harmonies with blocking</i>	✗	✗	👍	👍	✗
<i>learning from raw Turkish data</i>	✗	✗	✗	✗	
<i>E6: vowel and consonant harmonies without blocking</i>	👍	✗	✗	👍	👍
<i>E7: vowel and consonant harmonies with blocking</i>	✗	✗	✗	👍	✗
<i>E8: unbounded tone plateauing</i>	👍	✗	✗		✗
<i>E9: “simple” first-last harmony</i>	✗	✗	✗	✗	👍
<i>E10: “complex” first-last harmony</i>					✗

Table 5.1: Learning results that were experimentally obtained in this dissertation. Black cells indicate that the experiments were not conducted due to the reasons discussed in 5.1.2.

Learning well-formedness conditions To learn well-formedness conditions, I used learning algorithms for strictly piecewise (SP), strictly local (SL), tier-based strictly local (TSL), and multi-tier strictly local (MTSL) languages. SP grammars encode long-distance dependencies that prohibit certain substructures, while the distance between the elements of that substructure, as well as the type of

intervening material, plays no role Heinz and Rogers (2010); Heinz (2010b). Due to this nature, the SP learner was able to correctly generalize all long-distance patterns that do not involve blocking. The SP model achieved the accuracy of 100% on all harmonies that do not exhibit a blocking effect, including the pattern of Finnish harmony learned from raw Finnish data, and the unbounded tone plateauing. An SL learner models exclusively local processes, and therefore it succeeded on the word-final devoicing data, even when presented with the German wordlist. A TSL grammar captures a single long-distance dependency by enforcing the agreement among the selected subset of its alphabet (so-called *tier alphabet*) (Heinz et al., 2011; Jardine and McMullin, 2017). It captures patterns involving local or non-local dependencies with or without the blocking effect. However, if different agreements affect different sets of elements, such as in the case of independent vowel and consonant harmonies, one needs several tiers. It is indeed the capacity of MTSL models that showed successful learning of all types of typologically attested patterns presented to the learners (De Santo and Graf, 2019; McMullin et al., 2019). None of the learners captured the unattested pattern of first-last harmony that does not belong to any of those language classes. In some cases, although the learner would have “theoretically” able to learn the pattern, it could not extract the pattern from raw natural language data. So, for example, the TSL learner does not perform well on raw Finnish data, although the pattern is TSL in its nature. The Turkish vowel harmony is also TSL, but neither TSL nor MTSL learning algorithms could converge on the correct grammar. On the artificially generated data, the subregular learners correctly generalized all theoretically expected patterns.

Learning rewrite rules Due to the subsequential nature of many phonological and morphological processes discussed in section 2.2.2, I focused on the OSTIA inference algorithm for subsequential mappings (Oncina et al., 1993; de la

Higuera, 2010). The results show that it learned the local process of word-final devoicing, as well as harmonies that do not exhibit a blocking effect. Additionally, it also learned a simple first-case harmony, enforcing the agreement of the initial and final elements of the word. However, harmony systems including a blocking effect are also subsequential, but they were not generalized by OSTIA. As section 4.2.13 suggests, the reason for this might be rooted in the choice of OSTIA’s “pushback” condition: several versions of it are proposed in the literature (Oncina et al., 1993; de la Higuera, 2010, 2011). As expected, OSTIA did not learn the non-subsequential patterns such as unbounded tone plateauing or the more complicated case of first-last harmony. Other versions of OSTIA, as well as other transduction learners, would need to be explored in the future.

Some of the experiments were not conducted due to several reasons. The experiment targeting the learning of a “complex” first-last harmony was omitted since none of the subregular language learners could model even its simplified version. Running OSTIA using raw German, Finnish, or Turkish data would include the step of applying masking to those wordlists. It will also greatly increase memory use due to a large alphabet. Finally, upon the availability of the k -MTSL learning algorithm, one could confirm its inability to capture unbounded tone plateauing. Future work would help to fill those missing cells.


5.2 Future directions

There are several future directions that this type of work can take. Mainly, they can be classified as the implementation of other proposed algorithms, the design of the novel learning algorithms targeting classes of languages and mappings that do not have learners yet, and conducting learning experiments and analyzing their outcomes.

Implementation of other algorithms In the literature, other learning algorithms are not yet implemented and therefore their practical applications are not explored. Among them, there are the learning algorithms ISLFLA and OSLFIA which extract two subclasses of subsequential mappings that are especially useful for local phonological processes (Chandlee et al., 2014, 2015). A learner for the class of Output 2-TSL functions is available in (Burness and McMullin, 2019) and needs to be implemented as well. Chandlee et al. (2019) also proposes a transduction learner for feature-based representations learning long-distance dependencies.

Design and improvement of algorithms Several subregular language classes capture long-distance linguistic dependencies in other ways as well. Among them, there are TSL languages where the tier projection function is sensitive to the local context (ITSL), a full class of MTSL languages, and several other classes such as MITSL, OTSL, IOTSL, and IBSP (Graf, 2017b; De Santo and Graf, 2019). The current 2-MTSL learner does not always induce the minimal number of tiers, and this can be a focus of improvement as well. Some other ideas for the design of the learning algorithms are listed in section 4.3 and could be explored as well. Additionally, the topic of adding more “naturalness” to the learning algorithms needs to be further explored, such as learning the feature systems of the language or finding a way to encode linguistic notions such as natural classes.

Algorithm exploration via learning experiments Learning experiments show how the learner generalizes the pattern from real data. The shape of that data can be manipulated to explore how it affects the learning algorithm. For example, patterns exhibiting epenthesis, deletion, metathesis, and different types of dissimilations need to be added to the list of experiments as well. Studying the outcomes of learning experiments can help to improve the algorithms, and to better understand their scope.

During the last decade, the field of subregular research grew in its popularity, with theoretical advancements showing that it could be used for modeling different phonological, morphological, and even syntactic patterns. However, tools to use those theoretical results in practice were not available. To address this problem, I implemented the *SigmaPie*  package and practically explored the capacities of the available subregular learners. This type of work can lead to theoretical and practical improvements in the subregular field, which, in turn, can bring insights into understanding the structure of human language.

Appendix: code of *SigmaPie*

Grammar class

```
1  """A module with the definition of the grammar class. Copyright (C)
   2019 Alena
2  Aksenova.
3
4  This program is free software; you can redistribute it and/or modify
   it
5  under the terms of the GNU General Public License as published by
   the
6  Free Software Foundation; either version 3 of the License, or (at
   your
7  option) any later version.
8  """
9
10 from itertools import product
11 from sigmapie.helper import *
12
13
14 class L(object):
15     """A general class for grammars and languages.
16
17     Implements methods that
18     are applicable to all grammars in this package.
19     Attributes:
20         alphabet (list): alphabet used in the language;
21         grammar (list): the list of substructures;
22         k (int): locality window;
23         data (list): input data;
24         edges (list): start- and end-symbols for the grammar;
25         polar ("p" or "n"): polarity of the grammar.
26     """
```

```

27
28     def __init__(
29         self, alphabet=None, grammar=None, k=2, data=None, edges=[">
", "<"], polar="p"
30     ):
31         """Initializes the L object."""
32         if polar not in ["p", "n"]:
33             raise ValueError(
34                 "The value of polarity should be either "
35                 "positive ('p') or negative ('n')."
36             )
37         self.__polarity = polar
38         self.alphabet = alphabet
39         self.grammar = [] if grammar is None else grammar
40         self.k = k
41         self.data = [] if data is None else data
42         self.edges = edges
43
44     def extract_alphabet(self):
45         """Extracts alphabet from the given data or grammar and
46         saves it into
47         the 'alphabet' attribute.
48
49         CAUTION: if not all symbols were used in the data or grammar
50         ,
51
52         the result is not correct: update manually.
53         """
54         if self.alphabet is None:
55             self.alphabet = []
56         symbols = set(self.alphabet)
57         if self.data:
58             for item in self.data:
59                 symbols.update({j for j in item})
60         if self.grammar:

```

```

58         for item in self.grammar:
59             symbols.update({j for j in item})
60         symbols = symbols - set(self.edges)
61         self.alphabet = sorted(list(symbols))
62
63     def well_formed_ngram(self, ngram):
64         """Tells if the given ngram is well-formed. An ngram is ill-
65         formed if:
66
67         * there is something in-between two start- or end-symbols
68           ('>a>'), or
69
70         * something is before start symbol or after the end symbol
71           ('a>'), or
72
73         * the ngram consists only of start- or end-symbols.
74         Otherwise it is well-formed.
75         Arguments:
76             ngram (str): The ngram that needs to be evaluated.
77         Returns:
78             bool: well-formedness of the ngram.
79         """
80         start, end = [], []
81         for i in range(len(ngram)):
82             if ngram[i] == self.edges[0]:
83                 start.append(i)
84             elif ngram[i] == self.edges[1]:
85                 end.append(i)
86
87         start_len, end_len = len(start), len(end)
88         if any([start_len == len(ngram), end_len == len(ngram)]):
89             return False
90
91         if start_len > 0:
92             if ngram[0] != self.edges[0]:
93                 return False

```

```

91         if start_len > 1:
92             for i in range(1, start_len):
93                 if start[i] - start[i - 1] != 1:
94                     return False
95
96         if end_len > 0:
97             if ngram[-1] != self.edges[1]:
98                 return False
99             if end_len > 1:
100                 for i in range(1, end_len):
101                     if end[i] - end[i - 1] != 1:
102                         return False
103
104         return True
105
106     def generate_all_ngrams(self, symbols, k):
107         """Generates all possible ngrams of the length k based on
108         the given
109         alphabet.
110
111         Arguments:
112             alphabet (list): alphabet;
113             k (int): locality window (length of ngram).
114         Returns:
115             list: generated ngrams.
116         """
117         symb = symbols[:]
118         if not ((self.edges[0] in symb) or (self.edges[1] in symb)):
119             symb += self.edges
120
121         combinations = product(symb, repeat=k)
122         ngrams = []
123         for ngram in combinations:
124             if self.well_formed_ngram(ngram) and (ngram not in

```

```

ngrams):
124         ngrams.append(ngram)
125
126     return ngrams
127
128     def opposite_polarity(self, symbols):
129         """Returns the grammar opposite to the one given.
130
131         Arguments:
132             symbols (list): alphabet.
133         Returns:
134             list: ngrams of the opposite polarity.
135         """
136         all_ngrams = self.generate_all_ngrams(symbols, self.k)
137         opposite = [i for i in all_ngrams if i not in self.grammar]
138
139         return opposite
140
141     def check_polarity(self):
142         """Returns the polarity of the grammar ("p" or "n")."""
143         if self.__polarity == "p":
144             return "p"
145         return "n"
146
147     def change_polarity(self, new_polarity=None):
148         """Changes the polarity of the grammar.
149
150         Warning: it does not rewrite the grammar!
151         """
152         if new_polarity is not None:
153             if new_polarity not in ["p", "n"]:
154                 raise ValueError(
155                     "The value of polarity should be either "
156                     "positive ('p') or negative ('n')."

```

```

157         )
158         self.__polarity = new_polarity
159     else:
160         if self.__polarity == "p":
161             self.__polarity = "n"
162         elif self.__polarity == "n":
163             self.__polarity = "p"

```

Strictly local class

```

1  """A class of Strictly Local Grammars. Copyright (C) 2019  Alena
   Aksenova.
2
3  This program is free software; you can redistribute it and/or modify
   it
4  under the terms of the GNU General Public License as published by
   the
5  Free Software Foundation; either version 3 of the License, or (at
   your
6  option) any later version.
7  """
8
9  from random import choice
10 from sigmapie.helper import *
11 from sigmapie.fsm import *
12 from sigmapie.grammar import *
13
14
15 class SL(L):
16     """A class for strictly local grammars and languages.
17
18     Attributes:
19         alphabet (list): alphabet used in the language;
20         grammar (list): collection of ngrams;
21         k (int): locality window;

```



```

22     data (list): input data;
23     edges (list): start- and end-symbols for the grammar;
24     polar ("p" or "n"): polarity of the grammar;
25     fsm (FSM): corresponding finite state machine.
26     """
27
28     def __init__(
29         self, alphabet=None, grammar=None, k=2, data=None, edges=[">
30         ", "<"], polar="p"
31     ):
32         """Initializes the SL object."""
33         super().__init__(alphabet, grammar, k, data, edges, polar)
34         self.fsm = FSM(initial=self.edges[0], final=self.edges[1])
35
36     def learn(self):
37         """Extracts SL grammar from the given data."""
38         self.grammar = self.ngramize_data()
39         if self.check_polarity() == "n":
40             self.grammar = self.opposite_polarity(self.alphabet)
41
42     def annotate_string(self, string):
43         """Annotates the string with the start and end symbols.
44
45         Arguments:
46             string (str): a string that needs to be annotated.
47
48         Returns:
49             str: annotated version of the string.
50         """
51         return ">" * (self.k - 1) + string.strip() + "<" * (self.k -
52         1)
53
54     def ngramize_data(self):
55         """Creates set of n-grams based on the given data.

```

```

54     Returns:
55         list: collection of ngrams in the data.
56     """
57     if not self.data:
58         raise ValueError("The data is not provided.")
59
60     ngrams = []
61     for s in self.data:
62         item = self.annotate_string(s)
63         ngrams.extend(self.ngramize_item(item))
64
65     return list(set(ngrams))
66
67 def ngramize_item(self, item):
68     """This function n-gramizes a given string.
69
70     Arguments:
71         item (str): a string that needs to be ngramized.
72     Returns:
73         list: list of ngrams from the item.
74     """
75     ng = []
76     for i in range(len(item) - (self.k - 1)):
77         ng.append(tuple(item[i : (i + self.k)]))
78
79     return list(set(ng))
80
81 def fsmize(self):
82     """Builds FSM corresponding to the given grammar and saves
83     it in the
84     fsm attribute."""
85     if not self.grammar:
86         raise (IndexError("The grammar must not be empty."))
87     if not self.alphabet:

```

```

87         raise ValueError(
88             "The alphabet is not provided. " "Use 'grammar.
extract_alphabet()' ."
89         )
90
91     if self.check_polarity() == "p":
92         self.fsm.sl_to_fsm(self.grammar)
93     else:
94         opposite = self.opposite_polarity(self.alphabet)
95         self.fsm.sl_to_fsm(opposite)
96
97     def scan(self, string):
98         """Checks if the given string is well-formed with respect to
the given
99         grammar.
100
101         Arguments:
102             string (str): the string that needs to be evaluated.
103         Returns:
104             bool: well-formedness value of a string.
105         """
106         if not self.fsm.transitions:
107             self.fsmize()
108
109         string = self.annotate_string(string)
110         return self.fsm.scan_sl(string)
111
112     def generate_sample(self, n=10, repeat=True, safe=True):
113         """Generates a data sample of the required size, with or
without
114         repetitions depending on 'repeat' value.
115
116         Arguments:
117             n (int): the number of examples to be generated;

```

```

118         repeat (bool): allows (rep=True) or prohibits (rep=False
    )
119
120         repetitions within the list of generated items;
121         safe (bool): automatically breaks out of infinite loops,
122         for example, when the grammar cannot generate the
123         required number of data items, and the repetitions
124         are set to False.
125
126     Returns:
127         list: generated data sample.
128     """
129     if not self.alphabet:
130         raise ValueError("Alphabet cannot be empty.")
131     if not self.fsm.transitions:
132         self.fsmize()
133
134     statemap = self.state_map()
135     if not any([len(statemap[x]) for x in statemap]):
136         raise (
137             ValueError(
138                 "There are ngrams in the grammar that are"
139                 " not leading anywhere. Clean the grammar "
140                 " or run 'grammar.clean_grammar()'."
141             )
142         )
143
144     data = [self.generate_item(statemap) for i in range(n)]
145
146     if not repeat:
147         data = set(data)
148         useless_loops = 0
149         prev_len = len(data)
150
151         while len(data) < n:
152             data.add(self.generate_item(statemap))

```

```

151
152         if prev_len == len(data):
153             useless_loops += 1
154         else:
155             useless_loops = 0
156
157         if safe and useless_loops > 500:
158             print(
159                 "The grammar cannot produce the requested "
160                 "number of strings. Check the grammar, "
161                 "reduce the number, or allow repetitions."
162             )
163             break
164
165         return list(data)
166
167     def generate_item(self, statemap):
168         """Generates a well-formed string with respect to the given
169         grammar.
170
171         Arguments:
172             statemap (dict): a dictionary of possible transitions in
173             the
174             corresponding fsm; constructed inside
175             generate_sample.
176
177         Returns:
178             str: a well-formed string.
179         """
180         word = self.edges[0] * (self.k - 1)
181         while word[-1] != self.edges[1]:
182             word += choice(statemap[word[-(self.k - 1) :]])
183         return word[(self.k - 1) : -1]

```

```

182     """
183     Generates a dictionary of possible transitions in the FSM.
184     Returns:
185         dict: the dictionary of the form
186             {"keys":[list of possible next symbols]}, where
187             keys are (k-1)-long strings.
188     """
189     local_alphabet = self.alphabet[:] + self.edges[:]
190     poss = product(local_alphabet, repeat=(self.k - 1))
191
192     smap = {}
193     for i in poss:
194         for j in self.fsm.transitions:
195             if j[0] == i:
196                 before = "".join(i)
197                 if before in smap:
198                     smap[before] += j[1]
199                 else:
200                     smap[before] = [j[1]]
201     return smap
202
203     def switch_polarity(self):
204         """Changes polarity of the grammar, and changes the grammar
205         to the
206         opposite one."""
207         if not self.alphabet:
208             raise ValueError("Alphabet cannot be empty.")
209
210         self.grammar = self.opposite_polarity(self.alphabet)
211         self.change_polarity()
212
213     def clean_grammar(self):
214         """Removes useless ngrams from the grammar.

```

```

215         If negative, it just removes duplicates. If positive, it
detects
216         bigrams to which one cannot get         from the initial symbol
and
217         from which one cannot get         to the final symbol, and
removes
218         them.
219         """
220         if not self.fsm.transitions:
221             self.fsmize()
222
223         if self.check_polarity() == "n":
224             self.grammar = list(set(self.grammar))
225         else:
226             self.fsm.trim_fsm()
227             self.grammar = [j[0] + (j[1],) for j in self.fsm.
transitions]

```

Strictly piecewise class

```

1  """A class of Strictly Piecewise Grammars. Copyright (C) 2019  Alena
Aksenova.
2
3  This program is free software; you can redistribute it and/or modify
it
4  under the terms of the GNU General Public License as published by
the
5  Free Software Foundation; either version 3 of the License, or (at
your
6  option) any later version.
7  """
8
9  from random import choice
10 from itertools import product
11

```

```

12 from sigmapie.grammar import *
13 from sigmapie.fsm import *
14 from sigmapie.fsm_family import *
15 from sigmapie.helper import *
16
17
18 class SP(L):
19     """A class for strictly piecewise grammars and languages.
20
21     Attributes:
22         alphabet (list): alphabet used in the language;
23         grammar (list): collection of ngrams;
24         k (int): locality window;
25         data (list): input data;
26         polar ("p" or "n"): polarity of the grammar;
27         fsm (FSM): corresponding finite state machine.
28     """
29
30     def __init__(self, alphabet=None, grammar=None, k=2, data=None,
31 polar="p"):
32         """Initializes the SP object."""
33         super().__init__(alphabet, grammar, k, data, polar=polar)
34         self.fsm = FSMFamily()
35
36     def subsequences(self, string):
37         """Extracts k-long subsequences out of the given word.
38
39         Arguments:
40             string (str): a string that needs to be processed.
41         Returns:
42             list: a list of subsequences out of the string.
43         """
44         if len(string) < self.k:
45             return []

```



```

45
46     start = list(string[: self.k])
47     result = [start]
48
49     previous_state = [start]
50     current_state = []
51
52     for s in string[self.k :]:
53         for p in previous_state:
54             for i in range(self.k):
55                 new = p[:i] + p[i + 1 :] + [s]
56                 if new not in current_state:
57                     current_state.append(new)
58             result.extend(current_state)
59             previous_state = current_state[:]
60             current_state = []
61
62     return list(set([tuple(i) for i in result]))
63
64 def learn(self):
65     """Extracts k-long subsequences from the training data.
66
67     Results:
68         self.grammar is updated.
69     """
70     if not self.data:
71         raise ValueError("The data must be provided.")
72     if not self.alphabet:
73         raise ValueError(
74             "The alphabet must be provided. To "
75             "extract the alphabet automatically, "
76             "run 'grammar.extract_alphabet()'."
77         )
78

```

```

79     self.grammar = []
80     for i in self.data:
81         for j in self.subsequences(i):
82             if j not in self.grammar:
83                 self.grammar.append(j)
84
85     if self.check_polarity() == "n":
86         self.grammar = self.opposite_polarity()
87
88     def opposite_polarity(self):
89         """Returns the grammar opposite to the current one."""
90         all_ngrams = product(self.alphabet, repeat=self.k)
91         return [i for i in all_ngrams if i not in self.grammar]
92
93     def fsmize(self):
94         """Creates FSM family for the given SP grammar by passing
95 every
96 encountered subsequence through the corresponding automaton.
97 """
98
99         if not self.grammar:
100             self.learn()
101
102         if self.check_polarity() == "p":
103             data_subseq = self.grammar[:]
104         else:
105             data_subseq = self.opposite_polarity()
106
107         # create a family of templates in fsm attribute
108         seq = product(self.alphabet, repeat=self.k - 1)
109         for path in seq:
110             f = FSM(initial=None, final=None)
111             f.sp_build_template(path, self.alphabet, self.k)
112             self.fsm.family.append(f)

```

```

111     # run the input/grammar through the fsm family
112     for f in self.fsm.family:
113         for r in data_subseq:
114             f.sp_fill_template(r)
115
116     # clean the untouched transitions
117     for f in self.fsm.family:
118         f.sp_clean_template()
119
120 def scan(self, string):
121     """Tells if the input string is well-formed.
122
123     Arguments:
124         string (str): string to be scanned.
125     Returns:
126         bool: True is well-formed, otherwise False.
127     """
128     subseq = self.subsequences(string)
129     found_in_G = [(s in self.grammar) for s in subseq]
130
131     if self.check_polarity == "p":
132         return all(found_in_G)
133     else:
134         return not any(found_in_G)
135
136 def generate_item(self):
137     """Generates a well-formed string.
138
139     Returns:
140         str: the generated string.
141     """
142     if not self.alphabet:
143         raise ValueError("The alphabet must be provided.")
144

```

```

145     string = ""
146     while True:
147         options = []
148         for i in self.alphabet:
149             if self.scan(string + i):
150                 options.append(i)
151
152         add = choice(options + ["EOS"])
153         if add == "EOS":
154             return string
155         else:
156             string += add
157
158     def generate_sample(self, n=10, repeat=False, safe=True):
159         """Generates data sample of desired length.
160
161         Arguments:
162             n (int): the number of examples to be generated,
163                     the default value is 10;
164             repeat (bool): allow (rep=True) or prohibit (rep=False)
165                           repetitions, the default value is False;
166             safe (bool): automatically break out of infinite loops,
167                         for example, when the grammar cannot generate the
168                         required number of data items, and the repetitions
169                         are set to False.
170
171         Returns:
172             list: a list of generated examples.
173         """
174
175         sample = [self.generate_item() for i in range(n)]
176
177         if not repeat:
178             useless_loops = 0
179             sample = set(sample)
180             prev_len = len(sample)

```

```

179
180         while len(list(set(sample))) < n:
181             sample.add(self.generate_item())
182             if prev_len == len(sample):
183                 useless_loops += 1
184             else:
185                 useless_loops = 0
186
187             if safe and useless_loops > 100:
188                 print(
189                     "The grammar cannot produce the requested
190                     number" " of strings."
191                 )
192                 break
193
194         return list(sample)
195
196     def switch_polarity(self, new_polarity=None):
197         """Changes the polarity of the grammar.
198
199         Arguments:
200             new_polarity ("p" or "n"): the new value of the polarity
201             .
202
203         """
204         old_value = self.check_polarity()
205         self.change_polarity(new_polarity)
206         new_value = self.check_polarity()
207
208         if old_value != new_value:
209             self.grammar = self.opposite_polarity()
210
211     def clean_grammar(self):
212         """Removes useless ngrams from the grammar.

```

```

211         If negative, it just removes duplicates. If positive, it
        detects
212         bigrams to which one cannot get         from the initial symbol
        and
213         from which one cannot get         to the final symbol, and
        removes
214         them.
215         """
216         self.grammar = list(set(self.grammar))

```

Tier-based strictly local class

```

1  """A class of Tier-based Strictly Local Grammars. Copyright (C) 2019
    Alena
2  Aksenova.
3
4  This program is free software; you can redistribute it and/or modify
    it
5  under the terms of the GNU General Public License as published by
    the
6  Free Software Foundation; either version 3 of the License, or (at
    your
7  option) any later version.
8  """
9
10 from random import choice, randint
11 from sigmapie.sl_class import *
12
13
14 class TSL(SL):
15     """A class for tier-based strictly local grammars and languages.
16
17     Attributes:
18         alphabet (list): alphabet used in the language;
19         grammar (list): the list of substructures;

```

```

20         k (int): locality window;
21         data (list): input data;
22         edges (list): start- and end-symbols for the grammar;
23         polar ("p" or "n"): polarity of the grammar;
24         fsm (FSM): finite state machine that corresponds to the
grammar;
25         tier (list): list of tier symbols.
26     """
27
28     def __init__(
29         self,
30         alphabet=None,
31         grammar=None,
32         k=2,
33         data=None,
34         edges=[">", "<"],
35         polar="p",
36         tier=None,
37     ):
38         """Initializes the TSL object."""
39         super().__init__(alphabet, grammar, k, data, edges, polar)
40         self.tier = tier
41         self.fsm = FSM(initial=self.edges[0], final=self.edges[1])
42
43     def learn(self):
44         """Learns tier and finds attested (if positive) or
unattested (if
45         negative) ngrams of the tier images of the data."""
46         if not self.alphabet:
47             raise ValueError("Alphabet cannot be empty.")
48         if not self.data:
49             raise ValueError("Data needs to be provided.")
50
51         self.learn_tier()

```

```

52     tier_sequences = [self.tier_image(i) for i in self.data]
53     self.grammar = TSL(k=self.k, data=tier_sequences).
ngramize_data()
54
55     if self.check_polarity() == "n":
56         self.grammar = self.opposite_polarity(self.tier)
57
58     def learn_tier(self):
59         """This function determines which of the symbols used in the
language
60         are tier symbols, algorithm by Jardine & McMullin (2017).
61
62         Updates tier attribute.
63         """
64         self.tier = self.alphabet[:]
65         ngrams = self.ngramize_data()
66
67         ngrams_less = TSL(data=self.data, k=(self.k - 1)).
ngramize_data()
68         ngrams_more = TSL(data=self.data, k=(self.k + 1)).
ngramize_data()
69
70         for symbol in self.alphabet:
71             if self.test_insert(symbol, ngrams, ngrams_less) and
self.test_remove(
72                 symbol, ngrams, ngrams_more
73             ):
74                 self.tier.remove(symbol)
75
76     def test_insert(self, symbol, ngrams, ngrams_less):
77         """Tier presense test #1.
78
79         For every (n-1)-gram ('x','y','z'),
80         there must be n-grams of the type ('x','S','y','z') and

```



```

81     ('x','y','S','z').
82     Arguments:
83         symbol (str): the symbol that is currently being tested;
84         ngrams (list): the list of n-gramized input;
85         ngrams_less (list): the list of (n-1)-gramized input.
86     Returns:
87         bool: True if a symbol passed the test, otherwise False.
88     """
89     extension = []
90     for small in ngrams_less:
91         for i in range(len(small) + 1):
92             new = small[:i] + (symbol,) + small[i:]
93             if self.well_formed_ngram(new):
94                 extension.append(new)
95
96     # needs to be here: otherwise no local WF/WE processes
97     edgecase1 = tuple(self.edges[0] * (self.k - 1) + symbol)
98     edgecase2 = tuple(symbol + self.edges[1] * (self.k - 1))
99     extension.extend([edgecase1, edgecase2])
100
101     return set(extension).issubset(set(ngrams))
102
103     def test_remove(self, symbol, ngrams, ngrams_more):
104         """Tier presense test #2.
105
106         For every (n+1)-gram of the type
107         ('x','S','y'), there must be an n-gram of the type ('x', 'y
108         ').
109
110         Arguments:
111             symbol (str): the symbol that is currently being tested;
112             ngrams (list): the list of n-gramized input;
113             ngrams_more (list): the list of (n+1)-gramized input.
114
115         Returns:
116             bool: True if a symbol passed the test, otherwise False.

```

```

114     """
115     extension = []
116     for big in ngrams_more:
117         if symbol in big:
118             for i in range(len(big)):
119                 if big[i] == symbol:
120                     new = big[:i] + big[i + 1 :]
121                     if self.well_formed_ngram(new):
122                         extension.append(new)
123
124     return set(extension).issubset(set(ngrams))
125
126 def tier_image(self, string):
127     """Function that returns a tier image of the input string.
128
129     Arguments:
130         string (str): string that needs to be processed.
131     Returns:
132         str: tier image of the input string.
133     """
134     return "".join(i for i in string if i in self.tier)
135
136 def fsmize(self):
137     """Builds FSM corresponding to the given grammar and saves
138     in it the
139     fsm attribute."""
140     if not self.grammar:
141         raise (IndexError("The grammar must not be empty."))
142     if not self.tier:
143         raise ValueError(
144             "The tier is not extracted or empty. "
145             "Switch to SL or use 'grammar.learn()'"
146         )

```

```

147         if self.check_polarity() == "p":
148             self.fsm.sl_to_fsm(self.grammar)
149         else:
150             opposite = self.opposite_polarity(self.tier)
151             self.fsm.sl_to_fsm(opposite)
152
153     def switch_polarity(self):
154         """Changes polarity of the grammar, and rewrites grammar to
155         the
156         opposite one."""
157         if not self.tier:
158             raise ValueError(
159                 "Either the language is SL, or the tier "
160                 "is not extracted, use 'grammar.learn()'".
161             )
162
163         self.grammar = self.opposite_polarity(self.tier)
164         self.change_polarity()
165
166     def generate_sample(self, n=10, repeat=True, safe=True):
167         """Generates n well-formed strings, with or without
168         repetitions.
169
170         Arguments:
171             n (int): the number of examples to be generated;
172             repeat (bool): allow (rep=True) or prohibit (rep=False)
173             repetitions of the same data items;
174             safe (bool): automatically break out of infinite loops,
175             for example, when the grammar cannot generate the
176             required number of data items, and the repetitions
177             are set to False.
178
179         Returns:
180             list: generated data sample.
181         """

```

```

179         if not self.alphabet:
180             raise ValueError("Alphabet cannot be empty.")
181         if not self.tier:
182             raise ValueError(
183                 "Either the language is SL, or the tier "
184                 "is not extracted, use 'grammar.learn()'."
185             )
186
187         if len(self.alphabet) == len(self.tier):
188             sl = SL(polar=self.check_polarity())
189             sl.alphabet = self.alphabet
190             sl.grammar = self.grammar
191             sl.k = self.k
192             sl.edges = self.edges
193             sl.fsmize()
194             return sl.generate_sample(n, repeat, safe)
195
196         if not self.fsm.transitions:
197             self.fsmize()
198
199         statemap = self.state_map()
200         data = [self.generate_item() for i in range(n)]
201
202         if not repeat:
203             data = set(data)
204             useless_loops = 0
205             prev_len = len(data)
206             while len(data) < n:
207                 data.add(self.generate_item())
208                 if prev_len == len(data):
209                     useless_loops += 1
210                 else:
211                     useless_loops = 0
212

```

```

213         if safe and useless_loops > 100:
214             print(
215                 "The grammar cannot produce the requested
number" " of strings."
216             )
217             break
218
219         return list(data)
220
221     def generate_item(self):
222         """Generates a well-formed sequence of symbols.
223
224         Returns:
225             str: a well-formed string.
226         """
227         if not self.fsm.transitions:
228             self.fsmize()
229
230         statemap = self.state_map()
231         if not any([len(statemap[x]) for x in statemap]):
232             raise (
233                 ValueError(
234                     "There are ngrams in the grammar that are"
235                     " not leading anywhere. Clean the grammar "
236                     " or run 'grammar.clean_grammar()'."
237                 )
238             )
239
240         tier_seq = self.annotate_string(super().generate_item(
statemap))
241         ind = [x for x in range(len(tier_seq)) if tier_seq[x] not in
self.edges]
242         if not ind:
243             tier_items = []

```

```

244         else:
245             tier_items = list(tier_seq[ind[0] : (ind[-1] + 1)])
246
247             free_symb = list(set(self.alphabet).difference(set(self.tier
248 )))
249
250             new_string = self.edges[0] * (self.k - 1)
251             for i in range(self.k + 1):
252                 if randint(0, 1) and free_symb:
253                     new_string += choice(free_symb)
254
255             if not tier_items:
256                 return "".join([i for i in new_string if i not in self.
257 edges])
258
259             for item in tier_items:
260                 new_string += item
261                 for i in range(self.k + 1):
262                     if randint(0, 1) and free_symb:
263                         new_string += choice(free_symb)
264
265             return "".join([i for i in new_string if i not in self.edges
266 ])
267
268     def state_map(self):
269         """
270         Generates a dictionary of possible transitions in the FSM.
271         Returns:
272             dict: the dictionary of the form
273                 {"keys": [list of possible next symbols]}, where
274                 keys are (k-1)-long strings.
275         """
276         if self.fsm is None:
277             self.fsmize()

```

```

275
276     local_alphabet = self.tier[:] + self.edges[:]
277     poss = product(local_alphabet, repeat=(self.k - 1))
278
279     smap = {}
280     for i in poss:
281         for j in self.fsm.transitions:
282             if j[0] == i:
283                 before = "".join(i)
284                 if before in smap:
285                     smap[before] += j[1]
286                 else:
287                     smap[before] = [j[1]]
288     return smap
289
290 def scan(self, string):
291     """Checks if the given string is well-formed with respect to
292     the given
293     grammar.
294
295     Arguments:
296         string (str): the string that needs to be evaluated.
297     Returns:
298         bool: well-formedness value of a string.
299     """
300     tier_img = self.annotate_string(self.tier_image(string))
301     matches = [(n in self.grammar) for n in self.ngramize_item(
302         tier_img)]
303
304     if self.check_polarity() == "p":
305         return all(matches)
306     else:
307         return not any(matches)

```

Multi-tier strictly local class

```
1 """A class of Multiple Tier-based Strictly Local Grammars. Copyright
   (C) 2019
2 Alena Aksenova.
3
4 This program is free software; you can redistribute it and/or modify
   it
5 under the terms of the GNU General Public License as published by
   the
6 Free Software Foundation; either version 3 of the License, or (at
   your
7 option) any later version.
8 """
9
10 from copy import deepcopy
11 from random import choice, randint
12 from itertools import product
13 from sigmapie.tsl_class import *
14 from sigmapie.fsm_family import *
15
16
17 class MTSL(TSL):
18     """A class for tier-based strictly local grammars and languages.
19
20     Attributes:
21         alphabet (list): alphabet used in the language;
22         grammar (list): the list of substructures;
23         k (int): locality window;
24         data (list): input data;
25         edges (list): start- and end-symbols for the grammar;
26         polar ("p" or "n"): polarity of the grammar;
27         fsm (FSMFamily): a list of finite state machines that
28             corresponds to the grammar;
```



```

29         tier (list): list of tuples, where every tuple lists
elements
30         of some tier.
31     Learning for  $k > 2$  is not implemented: requires more theoretical
work.
32     """
33
34     def __init__(
35         self, alphabet=None, grammar=None, k=2, data=None, edges=[">
", "<"], polar="p"
36     ):
37         """Initializes the TSL object."""
38         super().__init__(alphabet, grammar, k, data, edges, polar)
39         self.fsm = FSMFamily()
40         if self.k != 2:
41             raise NotImplementedError(
42                 "The learner for k-MTSL languages is " "still being
designed."
43             )
44         self.tier = None
45
46     def learn(self):
47         """
48         Learns 2-local MTSL grammar for a given sample. The
algorithm
49         currently works only for  $k=2$  and is based on MTSL2IA
designed
50         by McMullin, Aksenova and De Santo (2019). We are currently
51         working on lifting the locality of the grammar to arbitrary
 $k$ .
52         Results:
53         self.grammar is updated with a grammar of the following
shape:
54         {(tier_1):[bigrams_for_tier_1],

```

```

55         ...
56         (tier_n):[bigrams_for_tier_n]}
57     """
58     if not self.data:
59         raise ValueError("Data needs to be provided.")
60     if not self.alphabet:
61         raise ValueError(
62             "The alphabet is empty. Provide data or "
63             "run 'grammar.extract_alphabet'."
64         )
65
66     possible = set(self.generate_all_ngrams(self.alphabet, self.
k))
67     attested = set()
68     for d in self.data:
69         bigrams = self.ngramize_item(self.annotate_string(d))
70         attested.update(set(bigrams))
71     unattested = list(possible.difference(attested))
72
73     paths = self.all_paths(self.data)
74     grammar = []
75
76     for bgr in unattested:
77         tier = self.alphabet[:]
78
79         for s in self.alphabet:
80             rmv = True
81
82             # condition 1
83             if s in bgr:
84                 rmv = False
85                 continue
86
87             # condition 2

```

```

88         relevant_paths = []
89         for p in paths:
90             if (p[0] == bgr[0]) and (p[-1] == bgr[-1]) and (
177             s in p[1]):
91                 relevant_paths.append(p)
92                 for rp in relevant_paths:
93                     new = [rp[0], set(i for i in rp[1] if i != s),
178                     rp[2]]
94                     if new not in paths:
95                         rmv = False
96                         break
97
98                 # remove from the tier if passed both conditions
99                 if rmv:
100                     tier.remove(s)
101
102                 grammar.append((tier, bgr))
103                 gathered = self.gather_grammars(grammar)
104
105                 self.grammar = gathered
106                 self.tier = [i for i in self.grammar]
107
108                 if self.check_polarity() == "p":
109                     self.grammar = self.opposite_polarity()
110
111     def scan(self, string):
112         """Scan string with respect to a given MTSL grammar.
113
114         Arguments:
115             string (str): a string that needs to be scanned.
116         Returns:
117             bool: well-formedness of the string.
118         """
119         tier_evals = []

```

```

120
121     for tier in self.grammar:
122         t = tier
123         g = self.grammar[tier]
124
125         delete_non_tier = "".join([i for i in string if i in t])
126         tier_image = self.annotate_string(delete_non_tier)
127         ngrams = self.ngramize_item((tier_image))
128
129         this_tier = [(ngr in g) for ngr in ngrams]
130
131         if self.check_polarity() == "p":
132             tier_evals.append(all(this_tier))
133         else:
134             tier_evals.append(not any(this_tier))
135
136     return all(tier_evals)
137
138 def gather_grammars(self, grammar):
139     """Gathers grammars with the same tier together.
140
141     Arguments:
142         grammar (list): a representation of the learned grammar
143             where there is a one-to-one mapping between tiers
144             and bigrams.
145     Returns:
146         dict: a dictionary where keys are tiers and values are
147             the restrictions imposed on those tiers.
148     """
149     G = {}
150     for i in grammar:
151         if tuple(i[0]) in G:
152             G[tuple(i[0])] += [i[1]]
153         else:

```

```

154         G[tuple(i[0])] = [i[1]]
155     return G
156
157     def path(self, string):
158         """Collects a list of paths from a string.
159
160         A path is a
161         triplet <a, X, b>, where 'a' is a symbol, 'b' is a symbol
162         that follows 'a' in 'string', and 'X' is a set of symbols
163         in-between 'a' and 'b'.
164
165         Arguments:
166             string (str): a string paths of which need to be found.
167
168         Returns:
169             list: list of paths of 'string'.
170         """
171
172         string = self.annotate_string(string)
173         paths = []
174
175         for i in range(len(string) - 1):
176             for j in range(i + 1, len(string)):
177                 path = [string[i]]
178                 path.append(set(k for k in string[(i + 1) : j]))
179                 path.append(string[j])
180
181                 if path not in paths:
182                     paths.append(path)
183
184         return paths
185
186     def all_paths(self, dataset):
187         """Finds all paths that are present in a list of strings.
188
189         Arguments:
190             dataset (list): a list of strings.

```

```

188     Returns:
189         list: a list of paths present in 'dataset'.
190     """
191     paths = []
192     for item in dataset:
193         for p in self.path(item):
194             if p not in paths:
195                 paths.append(p)
196
197     return paths
198
199 def opposite_polarity(self):
200     """Generates a grammar of the opposite polarity.
201
202     Returns:
203         dict: a dictionary containing the opposite ngram lists
204             for every tier of the grammar.
205     """
206     if not self.grammar:
207         raise ValueError(
208             "Grammar needs to be provided. It can also "
209             "be learned using 'grammar.learn()'."
210         )
211     opposite = {}
212     for i in self.grammar:
213         possib = self.generate_all_ngrams(list(i), self.k)
214         opposite[i] = [j for j in possib if j not in self.
grammar[i]]
215
216     return opposite
217
218 def switch_polarity(self):
219     """Changes polarity of the grammar, and rewrites grammar to
the

```

```

220         opposite one."""
221         self.grammar = self.opposite_polarity()
222         self.change_polarity()
223
224     def map_restrictions_to_fsms(self):
225         """Maps restrictions to FSMs: based on the grammar, it
226         creates a list
227         of lists, where every sub-list has the following shape:
228
229         [tier_n, restrictions_n, fsm_n]. Such sub-list is
230         constructed
231         for every single tier of the current MTSL grammar.
232         Returns:
233         [list, list, FSM]
234             list: a list of current tier's symbols;
235             list: a list of current tier's restrictions;
236             FSM: a FSM corresponding to the current tier.
237         """
238         if not self.grammar:
239             raise (IndexError("The grammar must not be empty."))
240
241         restr_to_fsm = []
242
243         for alpha, ngrams in self.grammar.items():
244             polarity = self.check_polarity()
245             tsl = TSL(
246                 self.alphabet,
247                 self.grammar,
248                 self.k,
249                 self.data,
250                 self.edges,
251                 polar=polarity,
252             )
253             if not tsl.alphabet:

```

```

252         tsl.extract_alphabet()
253         tsl.tier = list(alpha)
254         tsl.grammar = list(ngrams)
255         tsl.fsmize()
256         restr_to_fsm.append([tsl.tier[:], tsl.grammar[:], tsl.
fsm])
257
258     return restr_to_fsm
259
260     def fsmize(self):
261         """Builds FSM family corresponding to the given grammar and
saves in it
262         the fsm attribute."""
263         restr_to_fsm = self.map_restrictions_to_fsms()
264         self.fsm.family = [i[2] for i in restr_to_fsm]
265
266     def generate_sample(self, n=10, repeat=True, safe=True):
267         """Generates a data sample of the required size, with or
without
268         repetitions depending on 'repeat' value.
269
270         Arguments:
271             n (int): the number of examples to be generated;
272             repeat (bool): allows (rep=True) or prohibits (rep=False
)
273
274             repetitions within the list of generated items;
275             safe (bool): automatically breaks out of infinite loops,
276             for example, when the grammar cannot generate the
277             required number of data items, and the repetitions
278             are set to False.
279
280         Returns:
281             list: generated data sample.
282         """
283         if not self.alphabet:

```



```

282         raise ValueError("Alphabet cannot be empty.")
283     if not self.fsm.family:
284         self.fsmize()
285
286     tier_smap = self.tier_state_maps()
287     if not any([len(tier_smap[x]) for x in tier_smap]):
288         raise (
289             ValueError(
290                 "There are ngrams in the grammar that are"
291                 " not leading anywhere. Clean the grammar "
292                 " or run 'grammar.clean_grammar()'."
293             )
294         )
295
296     data = [self.generate_item(tier_smap) for i in range(n)]
297
298     if not repeat:
299         data = set(data)
300         useless_loops = 0
301         prev_len = len(data)
302
303         while len(data) < n:
304             data.add(self.generate_item(tier_smap))
305
306             if prev_len == len(data):
307                 useless_loops += 1
308             else:
309                 useless_loops = 0
310
311         if safe and useless_loops > 500:
312             print(
313                 "The grammar cannot produce the requested "
314                 "number of strings. Check the grammar, "
315                 "reduce the number, or allow repetitions."

```

```

316         )
317         break
318
319     return list(data)
320
321     def tier_image(self, string):
322         """
323         Creates tier images of a string with respect to the
324         different
325         tiers listed in the grammar.
326         Returns:
327             dict: a dictionary of the following shape:
328                 { (tier_1): "string_image_given_tier_1",
329                   ...,
330                   (tier_n): "string_image_given_tier_n"
331                 }
332         """
333         tiers = {}
334         for i in self.grammar:
335             curr_tier = ""
336             for s in string:
337                 if s in self.edges or s in i:
338                     curr_tier += s
339             tiers[i] = curr_tier
340         return tiers
341
342     def generate_item(self, tier_smap):
343         """Generates a well-formed string with respect to the given
344         grammar.
345
346         Returns:
347             str: a well-formed string.
348         """
349         word = self.edges[0] * (self.k - 1)

```

```

348     main_smap = self.general_state_map(tier_smap)
349     tier_images = self.tier_image(word)
350
351     while word[-1] != self.edges[1]:
352         maybe = choice(main_smap[word[-(self.k - 1) :]])
353         good = True
354         for tier in tier_smap:
355             if maybe in tier:
356                 old_image = tier_images[tier]
357                 if maybe not in tier_smap[tier][old_image[-(self
358 .k - 1) :]]:
359                     good = False
360             if good:
361                 word += maybe
362                 tier_images = self.tier_image(word)
363
364     newword = word[(self.k - 1) : -1]
365     if self.scan(newword):
366         return newword
367     else:
368         return self.generate_item(tier_smap)
369
370 def tier_state_maps(self):
371     """
372     Generates a dictionary of transitions within the FSMs
373     that correspond to the tier grammars.
374     Returns:
375         dict: the dictionary of the form
376         {
377             (tier_1): {"keys": [list of next symbols]},
378             (tier_2): {"keys": [list of next symbols]},
379             ...
380             (tier_n): {"keys": [list of next symbols]},
381         }, where keys are (k-1)-long tier representations.

```

```

381     Warning: the list of next symbols is tier-specific,
382           so this estimates the rough options: refer to
383           generate_item for the filtering of wrongly
384           generated items.
385     """
386     restr_to_fsm = self.map_restrictions_to_fsms()
387     tier_smaps = {}
388
389     for curr_tier in restr_to_fsm:
390         sl = SL()
391         sl.change_polarity(self.check_polarity())
392         sl.edges = self.edges
393         sl.k = self.k
394         sl.alphabet = curr_tier[0]
395         sl.grammar = curr_tier[1]
396         sl.fsm = curr_tier[2]
397         tier_smaps[tuple(sl.alphabet)] = sl.state_map()
398
399     return tier_smaps
400
401 def general_state_map(self, smaps):
402     """
403     Generates a dictionary of transitions within all
404     FSMs of the FSM family.
405     Returns:
406         dict: the dictionary of the form
407             {"keys": [list of next symbols]}, where
408             keys are (k-1)-long strings.
409     Warning: the list of next symbols is tier-specific,
410           so this estimates the rough options: refer to
411           generate_item for the filtering of wrongly
412           generated items.
413     """
414     local_smaps = deepcopy(smaps)

```

```

415
416     for tier in local_smmaps:
417         non_tier = [i for i in self.alphabet if i not in tier]
418         for entry in local_smmaps[tier]:
419             local_smmaps[tier][entry].extend(non_tier)
420
421     local_smmaps = list(local_smmaps.values())
422     main_smap = deepcopy(local_smmaps[0])
423
424     for other in local_smmaps[1:]:
425         for entry in other:
426
427             if entry not in main_smap:
428                 main_smap[entry] = other[entry]
429             else:
430                 inter = [i for i in main_smap[entry] if i in
other[entry]]
431                 main_smap[entry] = inter
432
433     free_ones = []
434     for i in self.alphabet:
435         for j in self.grammar:
436             if i in j:
437                 break
438             free_ones.append(i)
439
440     ext_alphabet = deepcopy(self.alphabet) + [self.edges[1]]
441     for x in free_ones:
442         main_smap[x] = ext_alphabet
443
444     return main_smap
445
446     def clean_grammar(self):
447         """Removes useless ngrams from the grammar.

```

```

448
449         If negative, it just removes duplicates. If positive, it
detects
450         ngrams to which one cannot get         from the initial symbol
and
451         from which one cannot get         to the final symbol, and
removes
452         them.
453         """
454         for tier in self.grammar:
455             sl = SL()
456             sl.change_polarity(self.check_polarity())
457             sl.edges = self.edges
458             sl.alphabet = list(tier)
459             sl.k = self.k
460             sl.grammar = self.grammar[tier]
461             sl.fsmize()
462             sl.clean_grammar()
463             self.grammar[tier] = deepcopy(sl.grammar)

```

FSM class

```

1  """A class of Finite State Machines. Copyright (C) 2019 Alena
Aksenova.
2
3  This program is free software; you can redistribute it and/or modify
it
4  under the terms of the GNU General Public License as published by
the
5  Free Software Foundation; either version 3 of the License, or (at
your
6  option) any later version.
7  """
8
9

```

```

10 class FSM(object):
11     """This class implements Finite State Machine.
12
13     Attributes:
14         initial (str): initial symbol;
15         final (str): final symbol;
16         transitions (list): triples of the form [prev_state,
17             transition, next_state].
18     """
19
20     def __init__(self, initial, final, transitions=None):
21         if transitions == None:
22             self.transitions = []
23         else:
24             self.transitions = transitions
25
26         self.initial = initial
27         self.final = final
28
29     def sl_to_fsm(self, grammar):
30         """Creates FSM transitions based on the SL grammar.
31
32         Arguments:
33             grammar (list): SL ngrams.
34         """
35         if not grammar:
36             raise ValueError("The grammar must not be empty.")
37         self.transitions = [(i[:-1], i[-1], i[1:]) for i in grammar]
38
39     def scan_sl(self, string):
40         """Scans a given string using the learned SL grammar.
41
42         Arguments:
43             string (str): a string that needs to be scanned.

```

```

44     Returns:
45         bool: well-formedness value of the string.
46     """
47     if string[0] != self.initial or string[-1] != self.final:
48         raise ValueError("The string is not annotated with " "
the delimiters.")
49     if not self.transitions:
50         raise ValueError(
51             "The transitions are empty. Extract the"
52             " transitions using grammar.fsmize()."
53         )
54
55     k = len(self.transitions[0][0]) + 1
56     for i in range(k - 1, len(string)):
57         move_to_next = []
58         for j in self.transitions:
59             can_read = string[(i - k + 1) : (i + 1)] == "".join(
j[0]) + j[1]
60             move_to_next.append(can_read)
61
62         if not any(move_to_next):
63             return False
64
65     return True
66
67     def trim_fsm(self):
68         """This function trims useless transitions.
69
70         1. Finds the initial state and collects the set of states to
which one
71             can come from that node and the nodes connected to it.
72         2. Changes direction of the transitions and runs algorithm
again to
73             detect states from which one cannot get to the final

```



```

state.
74     As the result, self.transitions only contains useful
transitions.
75     """
76     if not self.transitions:
77         raise ValueError("Transitions of the automaton must " "
not be empty.")
78     can_start = self.accessible_states(self.initial)
79     self.transitions = [(i[2], i[1], i[0]) for i in can_start]
80     mirrored = self.accessible_states(self.final)
81     self.transitions = [(i[2], i[1], i[0]) for i in mirrored]
82
83     def accessible_states(self, marker):
84         """Finds accessible states.
85
86         Arguments:
87             marker (str): initial or final state.
88         Returns:
89             list: list of transitions that can be made from
90                 the given initial or final state.
91         """
92         updated = self.transitions[:]
93
94         # find initial/final transitions
95         reachable = []
96         for i in self.transitions:
97             if i[0][0] == i[0][-1] == marker:
98                 reachable.append(i)
99                 updated.remove(i)
100
101         # to keep copies that can be modified while looping
102         mod_updated = updated[:]
103         mod_reachable = []
104         first_time = True

```

```

105
106     # find transitions that can be reached
107     while mod_reachable != [] or first_time:
108         mod_reachable = []
109         first_time = False
110         for p in updated:
111             for s in reachable:
112                 if p[0] == s[2]:
113                     mod_reachable.append(p)
114                     mod_updated.remove(p)
115         updated = mod_updated[:]
116         reachable.extend(mod_reachable)
117
118     return reachable
119
120 def sp_build_template(self, path, alphabet, k):
121     """Generates a template for the given k-SP path.
122
123     Arguments:
124         path (str): the sequence for which the template is
125 generated;
126         alphabet (list): list of all symbols of the grammar;
127         k (int): window size of the grammar.
128     """
129
130     # creating the "skeleton" of the FSM
131     for i in range(k - 1):
132         # boolean shows whether the transition was accessed
133         self.transitions.append([i, path[i], i + 1, False])
134
135     # adding non-final loops
136     newtrans = []
137     for t in self.transitions:
138         for s in alphabet:

```

```

138         if s != t[1]:
139             newtrans.append([t[0], s, t[0], False])
140
141     # adding final loops
142     for s in alphabet:
143         newtrans.append(
144             [self.transitions[-1][2], s, self.transitions
145 [-1][2], False]
146         )
147
148     self.transitions += newtrans
149
150 def sp_fill_template(self, sequence):
151     """Runs the input sequence through the SP automaton and
152 marks
153 transitions if they were taken.
154
155 Cleans
156 transitions that were not taken afterwards.
157 Arguments:
158     sequence (str): sequence of symbols that needs to be
159 passed through the automaton.
160 """
161     state = 0
162     for s in sequence:
163         for t in self.transitions:
164             if (t[0] == state) and (t[1] == s):
165                 state = t[2]
166                 t[3] = True
167                 break
168
169 def sp_clean_template(self):
170     """Removes transitions that were not accessed."""
171     self.transitions = [i[:3] for i in self.transitions if i[3]]

```

```

170 == True]
171
172 def scan_sp(self, string):
173     """Runs the given sequence through the automaton.
174
175     Arguments:
176         string (str): string to run through the automaton.
177     Returns:
178         bool: True if input can be accepted by the automaton,
179             otherwise False.
180     """
181     state = 0
182     for s in string:
183         change = False
184         for t in self.transitions:
185             if (t[0] == state) and (t[1] == s):
186                 state = t[2]
187                 change = True
188                 break
189
190         if not change:
191             return False
192
193     return True

```

FSM family class

```

1 """A class of Families of Finite State Machines. Copyright (C) 2019
2 Alena
3 Aksenova.
4
5 This program is free software; you can redistribute it and/or modify
6 it
7 under the terms of the GNU General Public License as published by
8 the

```

```

6 Free Software Foundation; either version 3 of the License, or (at
   your
7 option) any later version.
8 """
9
10 from sigmapie.fsm import *
11
12
13 class FSMFamily(object):
14     """
15     This class encodes Family of Finite State Machines. Used for
16     a simple encoding of FSMs corresponding to SP languages.
17     Attributes:
18         transitions(list): triples of the form
19             [prev_state, transition, next_state].
20     """
21
22     def __init__(self, family=None):
23         """Initializes the FSMFamily object."""
24         if family is None:
25             self.family = []
26         else:
27             self.family = family
28
29     def run_all_fsm(self, string):
30         """Tells whether the given string is accepted by all the
31         automata of
32         the family.
33
34         Arguments:
35             string (str): the input string.
36
37         Returns:
38             bool: True if the string is accepted by all the
39                 fsms, otherwise False.

```

```

38         """
39         return all([f.scan_sp(string) for f in self.family])

```

FST class

```

1  """A class defining the Finite State Transducer. Copyright (C) 2019
    Alena
2  Aksenova.
3
4  This program is free software; you can redistribute it and/or modify
    it
5  under the terms of the GNU General Public License as published by
    the
6  Free Software Foundation; either version 3 of the License, or (at
    your
7  option) any later version.
8  """
9
10 from copy import deepcopy
11
12
13 class FST:
14     """A class representing finite state transducers.
15
16     Attributes:
17         Q (list): a list of states;
18         Sigma (list): a list of symbols of the input alphabet;
19         Gamma (list): a list of symbols of the output alphabet;
20         qe (str): name of the unique initial state;
21         E (list): a list of transitions;
22         stout (dict): a collection of state outputs.
23     """
24
25     def __init__(self, Sigma=None, Gamma=None):
26         """Initializes the FST object."""

```

```

27     self.Q = None
28     self.Sigma = Sigma
29     self.Gamma = Gamma
30     self.qe = ""
31     self.E = None
32     self.stout = None
33
34     def rewrite(self, w):
35         """Rewrites the given string with respect to the rules
36         represented in
37         the current FST.
38
39         Arguments:
40             w (str): a string that needs to be rewritten.
41         Outputs:
42             str: the translation of the input string.
43         """
44         if self.Q == None:
45             raise ValueError("The transducer needs to be constructed
46             .")
47
48         # move through the transducer and write the output
49         result = ""
50         current_state = ""
51         moved = False
52         for i in range(len(w)):
53             for tr in self.E:
54                 if tr[0] == current_state and tr[1] == w[i]:
55                     result += tr[2]
56                     current_state, moved = tr[3], True
57                     break
58             if moved == False:
59                 raise ValueError(
60                     "This string cannot be read by the current

```

```

transducer."
59         )
60
61     # add the final state output
62     if self.stout[current_state] != "*":
63         result += self.stout[current_state]
64
65     return result
66
67 def copy_fst(self):
68     """Produces a deep copy of the current FST.
69
70     Returns:
71         T (FST): a copy of the current FST.
72     """
73     T = FST()
74     T.Q = deepcopy(self.Q)
75     T.Sigma = deepcopy(self.Sigma)
76     T.Gamma = deepcopy(self.Gamma)
77     T.E = deepcopy(self.E)
78     T.stout = deepcopy(self.stout)
79
80     return T

```

OSTIA

```

1  """An implementation of the learning algorithm OSTIA. Copyright (C)
   2019 Alena
2  Aksenova.
3
4  This program is free software; you can redistribute it and/or modify
   it
5  under the terms of the GNU General Public License as published by
   the
6  Free Software Foundation; either version 3 of the License, or (at

```



```

    your
7 option) any later version.
8 """
9
10 from sigmapie.fst_object import *
11 from sigmapie.helper import *
12
13
14 def ostia(S, Sigma, Gamma):
15     """This function implements OSTIA (Onward Subsequential
16     Transduction
17     Inference Algorithm).
18
19     Arguments:
20         S (list): a list of pairs (o, t), where 'o' is the original
21         string, and 't' is its translation;
22         Sigma (list): the input alphabet;
23         Gamma (list): the output alphabet.
24
25     Returns:
26         FST: a transducer defining the mapping.
27     """
28     # create a template of the onward PTT
29     T = build_ptt(S, Sigma, Gamma)
30     T = onward_ptt(T, "", "")[0]
31
32     # color the nodes
33     red = [""]
34     blue = [tr[3] for tr in T.E if tr[0] == "" and len(tr[1]) == 1]
35
36     # choose a blue state
37     while len(blue) != 0:
38         blue_state = blue[0]
39
40         # if exists state that we can merge with, do it

```

```

39         exists = False
40         for red_state in red:
41
42             # if you already merged that blue state with something,
stop
43             if exists == True:
44                 break
45
46             # try to merge these two states
47             if ostia_merge(T, red_state, blue_state):
48                 T = ostia_merge(T, red_state, blue_state)
49                 exists = True
50
51             # if it is not possible, color that blue state red
52             if not exists:
53                 red.append(blue_state)
54
55             # if possible, remove the folded state from the list of
states
56             else:
57                 T.Q.remove(blue_state)
58                 del T.stout[blue_state]
59
60             # add in blue list other states accessible from the red ones
that are not red
61             blue = []
62             for tr in T.E:
63                 if tr[0] in red and tr[3] not in red:
64                     blue.append(tr[3])
65
66             # clean the transducer from non-reachable states
67             T = ostia_clean(T)
68             T.E = [tuple(i) for i in T.E]
69

```

```

70     return T
71
72
73 def build_ptt(S, Sigma, Gamma):
74     """Builds a prefix tree transducer based on the data sample.
75
76     Arguments:
77         S (list): a list of pairs (o, t), where 'o' is the original
78             string, and 't' is its translation;
79         Sigma (list): the input alphabet;
80         Gamma (list): the output alphabet.
81     """
82
83     # build a template for the transducer
84     T = FST(Sigma, Gamma)
85
86     # fill in the states of the transducer
87     T.Q = []
88     for i in S:
89         for j in prefix(i[0]):
90             if j not in T.Q:
91                 T.Q.append(j)
92
93     # fill in the empty transitions
94     T.E = []
95     for i in T.Q:
96         if len(i) >= 1:
97             T.E.append([i[:-1], i[-1], "", i])
98
99     # fill in state outputs
100    T.stout = {}
101    for i in T.Q:
102        for j in S:
103            if i == j[0]:

```

```

104         T.stout[i] = j[1]
105     if i not in T.stout:
106         T.stout[i] = "*"
107
108     return T
109
110
111 def onward_ptt(T, q, u):
112     """Function recursively pushing the common parts of strings
113     towards the
114     initial state therefore making the machine onward.
115
116     Arguments:
117         T (FST): a transducer that is being modified;
118         q (str): a state that is being processes;
119         u (str): a current part of the string to be moved.
120
121     Returns:
122         (FST, str, str)
123         FST: the updated transducer;
124         str: a new state;
125         u: a new string to be moved.
126
127     """
128     # proceed as deep as possible
129     for tr in T.E:
130         if tr[0] == q:
131             T, qx, w = onward_ptt(T, tr[3], tr[1])
132             if tr[2] != "*":
133                 tr[2] += w
134
135     # find lcp of all ways of leaving state 1 or stopping in it
136     t = [tr[2] for tr in T.E if tr[0] == q]
137     f = lcp(T.stout[q], *t)
138
139     # remove from the prefix unless it's the initial state

```

```

137     if f != "" and q != "":
138         for tr in T.E:
139             if tr[0] == q:
140                 tr[2] = remove_from_prefix(tr[2], f)
141                 T.stout[q] = remove_from_prefix(T.stout[q], f)
142
143     return T, q, f
144
145
146 def ostia_outputs(w1, w2):
147     """Function implementing a special comparison operation:
148
149     it returns a string if two strings are the same and if
150     another string is unknown, and False otherwise.
151     Arguments:
152         w1 (str): the first string;
153         w2 (str): the second string.
154     Returns:
155         bool | if strings are not the same;
156         str | otherwise.
157     """
158     if w1 == "*":
159         return w2
160     elif w2 == "*":
161         return w1
162     elif w1 == w2:
163         return w2
164     else:
165         return False
166
167
168 def ostia_pushback(T_orig, q1, q2, a):
169     """Re-distributes lcp of two states further in the FST.
170

```

```

171 Arguments:
172     T_orig (FST): a transducer;
173     q1 (str): the first state;
174     q2 (str): the second state;
175     a (str): the lcp of q1 and q2.
176 Returns:
177     FST: an updated transducer.
178 """
179 # to avoid rewriting the original transducer
180 T = T_orig.copy_fst()
181
182 # states where you get if follow a
183 q1_goes_to = None
184 q2_goes_to = None
185
186 # what is being written from this state
187 from_q1, from_q2 = None, None
188 for tr in T.E:
189     if tr[0] == q1 and tr[1] == a:
190         from_q1 = tr[2]
191         q1_goes_to = tr[3]
192     if tr[0] == q2 and tr[1] == a:
193         from_q2 = tr[2]
194         q2_goes_to = tr[3]
195 if from_q1 == None or from_q2 == None:
196     raise ValueError("One of the states cannot be found.")
197
198 # find the part after longest common prefix
199 u = lcp(from_q1, from_q2)
200 remains_q1 = from_q1[len(u) :]
201 remains_q2 = from_q2[len(u) :]
202
203 # assign lcp as current output
204 for tr in T.E:

```

```

205         if tr[0] in [q1, q2] and tr[1] == a:
206             tr[2] = u
207
208         # find what the next state writes given any other choice
209         # and append the common part in it
210         for tr in T.E:
211             if tr[0] == q1_goes_to:
212                 tr[2] = remains_q1 + tr[2]
213             if tr[0] == q2_goes_to:
214                 tr[2] = remains_q2 + tr[2]
215
216         # append common part to the next state's state output
217         if T.stout[q1_goes_to] != "*":
218             T.stout[q1_goes_to] = remains_q1 + T.stout[q1_goes_to]
219         if T.stout[q2_goes_to] != "*":
220             T.stout[q2_goes_to] = remains_q2 + T.stout[q2_goes_to]
221
222         return T
223
224
225 def ostia_merge(T_orig, q1, q2):
226     """Re-directs all branches of q2 into q1.
227
228     Arguments:
229         T_orig (FST): a transducer;
230         q1 (str): the first state;
231         q2 (str): the second state.
232
233     Returns:
234         FST: an updated transducer.
235     """
236     # to avoid rewriting the original transducer
237     T = T_orig.copy_fst()
238
239     # save which transition was changed to revert in case cannot

```

```

merge the states
239     changed = None
240     for tr in T.E:
241         if tr[3] == q2:
242             changed = tr[:]
243             tr[3] = q1
244
245     # save the state output of the q1 originally
246     changed_stout = T.stout[q1]
247
248     # check if we can merge the states
249     can_do = ostia_fold(T, q1, q2)
250
251     # if cannot, revert the change
252     if can_do == False:
253         for tr in T.E:
254             if tr[0] == changed[0] and tr[1] == changed[1] and tr[2]
255             == changed[2]:
256                 tr[3] = changed[3]
257                 T.stout[q1] = changed_stout
258                 return False
259
260     # if can, do it
261     else:
262         return can_do
263
264 def ostia_fold(T_orig, q1, q2):
265     """Recursively folds subtrees of q2 into q1.
266
267     Arguments:
268         T_orig (FST): a transducer;
269         q1 (str): the first state;
270         q2 (str): the second state.

```



```

271 Returns:
272     FST: an updated transducer.
273     """
274     # to avoid rewriting the original transducer
275     T = T_orig.copy_fst()
276
277     # compare the state outputs
278     w = ostia_outputs(T.stout[q1], T.stout[q2])
279     if w == False:
280         return False
281
282     # rewrite * in case it's the output of q1
283     T.stout[q1] = w
284
285     # look at every possible subtree of q_2
286     for a in T.Sigma:
287         add_new = False
288
289         for tr_2 in T.E:
290             if tr_2[0] == q2 and tr_2[1] == a:
291
292                 # if the edge exists from q1
293                 edge_defined = False
294                 for tr_1 in T.E:
295                     if tr_1[0] == q1 and tr_1[1] == a:
296                         edge_defined = True
297
298                 # fail if inconsistent with output of q2
299                 if tr_1[2] not in prefix(tr_2[2]):
300                     return False
301
302                 # move the mismatched suffix of q1 and q2
303                 further

```

```

T = ostia_pushback(T, q1, q2, a)

```

```

304         T = ostia_fold(T, tr_1[3], tr_2[3])
305         if T == False:
306             return False
307
308         # if the edge doesn't exist from q1 yet, add it
309         if not edge_defined:
310             add_new = [q1, a, tr_2[2], tr_2[3]]
311
312         # if the new transition was constructed, add it to the list
313         of transitions
314         if add_new:
315             T.E.append(add_new)
316
317         return T
318
319 def ostia_clean(T_orig):
320     """Removes the disconnected branches from the transducer that
321     appear due to
322     the step folding the sub-trees.
323
324     Arguments:
325         T_orig (FST): a transducer.
326     Returns:
327         FST: an updated transducer.
328     """
329     # to avoid rewriting the original transducer
330     T = T_orig.copy_fst()
331
332     # determine which states are reachable, i.e. accessible from the
333     initial state
334     reachable_states = [""]
335     add = []
336     change_made = True

```

```

335 while change_made == True:
336     change_made = False
337     for st in reachable_states:
338         for tr in T.E:
339             if tr[0] == st and tr[3] not in reachable_states and
tr[3] not in add:
340                 add.append(tr[3])
341                 change_made = True
342
343     # break out of the loop if after checking the list once
again, no states were added
344     if change_made == False:
345         break
346     else:
347         reachable_states.extend(add)
348         add = []
349
350     # clean the list of transitions
351     new_E = []
352     for tr in T.E:
353         if tr[0] in reachable_states and tr[3] in reachable_states:
354             new_E.append(tr)
355     T.E = new_E
356
357     # clean the dictionary of state outputs
358     new_stout = {}
359     for i in T.stout:
360         if i in reachable_states:
361             new_stout[i] = T.stout[i]
362     T.stout = new_stout
363
364     # clean the list of states
365     new_Q = [i for i in T.Q if i in reachable_states]
366     T.Q = new_Q

```

367

368 `return T`

Additional functions

```
1 """Module with general helper functions for the subregular package.
   Copyright
2 (C) 2019 Alena Aksenova.
3
4 This program is free software; you can redistribute it and/or modify
   it
5 under the terms of the GNU General Public License as published by
   the
6 Free Software Foundation; either version 3 of the License, or (at
   your
7 option) any later version.
8 """
9
10
11 def alphabetize(data):
12     """Detects symbols used in the input data.
13
14     Arguments:
15         data (list): Input data.
16     Returns:
17         list: Symbols used in these examples.
18     """
19     alphabet = set()
20     for item in data:
21         alphabet.update({i for i in item})
22     return sorted(list(alphabet))
23
24
25 def get_gram_info(ngrams):
26     """Returns the alphabet and window size of the grammar.
```

```

27
28 Arguments:
29     ngrams (list): list of ngrams.
30 Returns:
31     (list, int)
32         list: alphabet;
33         int: locality window.
34
35 """
36 alphabet = list(set([i for i in "".join(ngrams) if i not in [ ">"
37 , "<" ]]))
38
39 k = max(len(i) for i in ngrams)
40 return alphabet, k
41
42
43 def prefix(w):
44     """Returns a list of prefixes of a given string.
45
46     Arguments:
47         w (str): a string prefixes of which need to be extracted.
48     Returns:
49         list: a list of prefixes of the given string.
50     """
51     return [w[:i] for i in range(len(w) + 1)]
52
53
54 def lcp(*string):
55     """
56     Finds the longest common prefix of an unbounded number of
57     strings.
58
59     Arguments:
60         *string (str): one or more strings;
61     Returns:
62         str: a longest common prefix of the input strings.
63     """

```

```

59     w = list(set(i for i in string if i != "*"))
60     if not w:
61         raise IndexError("At least one non-unknown string needs to
be provided.")
62
63     result = ""
64     n = min([len(x) for x in w])
65     for i in range(n):
66         if len(set(x[i] for x in w)) == 1:
67             result += w[0][i]
68         else:
69             break
70
71     return result
72
73
74 def remove_from_prefix(w, pref):
75     """Removes a substring from the prefix position of another
string.
76
77     Arguments:
78         w (str): a string that needs to be modified;
79         pref (str): a prefix that needs to be removed from the
string.
80     Returns:
81         str: the modified string.
82     """
83     if w.startswith(pref):
84         return w[len(pref) :]
85     elif w == "*":
86         return w
87
88     raise ValueError(pref + " is not a prefix of " + w)

```

Package initialization

```
1 """
2     SigmaPie: a toolkit for subregular grammars and languages.
3     Copyright (C) 2019   Alena Aksenova
4
5     This program is free software; you can redistribute it and/or
6     modify
7     it under the terms of the GNU General Public License as published
8     by
9     the Free Software Foundation; either version 3 of the License, or
10    (at your option) any later version.
11 """
12
13 from sigmapie.sl_class import *
14 from sigmapie.tsl_class import *
15 from sigmapie.mtsl_class import *
16 from sigmapie.sp_class import *
17 from sigmapie.ostia import *
18
19 print(
20     "\nYou successfully loaded SigmaPie. \n\n"
21     "Formal language classes and grammars available:\n"
22     "\t* strictly piecewise: SP(alphabet, grammar, k, data, polar);\n"
23     "\n"
24     "\t* strictly local: SL(alphabet, grammar, k, data, edges, polar\n"
25     ");\n"
26     "\t* tier-based strictly local: TSL(alphabet, grammar, k, data,\n"
27     "edges,\n"
28     "    polar, tier);\n"
29     "\t* multiple tier-based strictly local: MTSL(alphabet, grammar,\n"
30     "k, "\n"
31     "data, edges, polar).\n\n"
32     "Alternatively, you can initialize a transducer: "
```

```

27     "FST(states, sigma, gamma, initial, transitions, stout).\n"
28     "Learning algorithm:\n"
29     "\tOSTIA: ostia(sample, sigma, gamma).\"
30 )

```

Appendix: unit tests

Unit test for Grammar

```

1  #!/bin/python3
2
3  """A module with the unittests for the grammar module. Copyright (C)
4      2019 Alena
5      Aksenova.
6
7  This program is free software; you can redistribute it and/or modify
8  it
9  under the terms of the GNU General Public License as published by
10 the
11 Free Software Foundation; either version 3 of the License, or (at
12 your
13 option) any later version.
14 """
15
16 import sys, os
17
18 sys.path.insert(0, os.path.join(os.path.abspath(".."), ""))
19
20 import unittest
21 from grammar import L
22
23 class TestGeneralLanguages(unittest.TestCase):
24     """Tests for the L class."""

```



```

22
23 def test_good_ngram_standard_edges(self):
24     """Checks if ill-formed ngrams are correctly recognized, and
25     that the
26
27     well-formed ones are not blocked.
28
29     Tests standard edge-markers.
30     """
31     l = L()
32     self.assertTrue(l.well_formed_ngram(("a", "b", "a")))
33     self.assertTrue(l.well_formed_ngram((">", "a", "b")))
34     self.assertTrue(l.well_formed_ngram((">", "a", "<")))
35     self.assertTrue(l.well_formed_ngram((">", "<")))
36     self.assertTrue(l.well_formed_ngram(("b", "<")))
37     self.assertTrue(l.well_formed_ngram(("a", "a", "a", "a", "a"
38     )))
39
40     self.assertFalse(l.well_formed_ngram(("a", ">")))
41     self.assertFalse(l.well_formed_ngram(("?", "d", "<", ">")))
42     self.assertFalse(l.well_formed_ngram(("a", ">", "a")))
43     self.assertFalse(l.well_formed_ngram((">", ">")))
44     self.assertFalse(l.well_formed_ngram(("<")))
45
46
47 def test_good_ngram_non_standard_edges(self):
48     """Checks if ill-formed ngrams are correctly recognized, and
49     that the
50
51     well-formed ones are not blocked.
52
53     Tests user-provided edge markers.
54     """
55     l = L()
56     l.edges = ["$", "#"]
57     self.assertTrue(l.well_formed_ngram(("$", "a", "b")))
58     self.assertTrue(l.well_formed_ngram(("$", "a", "#")))

```

```

53     self.assertTrue(l.well_formed_ngram(("$", "#")))
54     self.assertTrue(l.well_formed_ngram(("b", "#")))
55
56     self.assertFalse(l.well_formed_ngram(("a", "$")))
57     self.assertFalse(l.well_formed_ngram(("$", "d", "#", "$")))
58     self.assertFalse(l.well_formed_ngram(("a", "$", "a")))
59     self.assertFalse(l.well_formed_ngram(("$", "$")))
60     self.assertFalse(l.well_formed_ngram(("#")))
61
62     def test_ngram_gen(self):
63         """Checks if ngram generation method produces the expected
64         results."""
65         l = L(alphabet=["a", "b"])
66         ngrams = l.generate_all_ngrams(l.alphabet, l.k)
67
68         ng = {
69             ">", "<"),
70             ">", "a"),
71             "a", "<"),
72             ">", "b"),
73             "b", "<"),
74             "a", "a"),
75             "b", "b"),
76             "b", "a"),
77             "a", "b"),
78         }
79         self.assertTrue(set(ngrams) == ng)
80
81     def test_switch_same_alpha(self):
82         """Checks if the generated grammar is correct when all
83         alphabet symbols
84         are used in the grammar, also checks that polarity was
85         changed."""
86         g = [(">", "a"), ("b", "<"), ("a", "b"), ("b", "a")]

```

```

84         l = L(grammar=g)
85         l.extract_alphabet()
86
87         old_polarity = l.check_polarity()
88
89         g_opp = {(">", "<"), ("a", "<"), (">", "b"), ("b", "b"), ("a",
90         "a")}
91         self.assertTrue(set(l.opposite_polarity(l.alphabet)) ==
92         g_opp)
93         self.assertFalse(old_polarity == l.check_polarity)
94
95     def test_switch_different_alpha(self):
96         """Checks if the generated grammar is correct when not all
97         alphabet
98         symbols are used in the grammar; also checks that polarity
99         was
100         changed."""
101         g = [(">", "b"), ("b", "<"), (">", "<")]
102         l = L(grammar=g)
103         l.alphabet = ["a", "b"]
104
105         old_polarity = l.check_polarity()
106
107         g_opp = {(">", "a"), ("a", "<"), ("a", "a"), ("b", "a"), ("a",
108         "b"), ("b", "b")}
109         self.assertTrue(set(l.opposite_polarity(l.alphabet)) ==
110         g_opp)
111         self.assertFalse(old_polarity == l.check_polarity)
112
113     def test_change_polarity(self):
114         """Tests the correctness of change_polarity."""
115         a = L(polar="n")
116         a.change_polarity(new_polarity="n")
117         self.assertTrue(a.check_polarity() == "n")

```

```

112         a.change_polarity()
113         self.assertFalse(a.check_polarity() == "n")
114
115         b = L()
116         old_polarity = b.check_polarity()
117         b.change_polarity()
118         self.assertTrue(b.check_polarity() != old_polarity)
119
120
121 if __name__ == "__main__":
122     unittest.main()

```

Unit test for SL

```

1  #!/bin/python3
2
3  """A module with the unittests for the SL module. Copyright (C) 2019
   Alena
4  Aksenova.
5
6  This program is free software; you can redistribute it and/or modify
   it
7  under the terms of the GNU General Public License as published by
   the
8  Free Software Foundation; either version 3 of the License, or (at
   your
9  option) any later version.
10 """
11
12 import unittest
13 from sl_class import *
14
15
16 class TestSLLanguages(unittest.TestCase):
17     """Tests for the SL class."""

```

```

18
19     def test_scan_pos(self):
20         """Checks if well-formed strings are detected correctly
21         given the
22         provided positive grammar."""
23         slp = SL()
24         slp.grammar = [(">", "a"), ("b", "a"), ("a", "b"), ("b", "<")
25         ])
26         slp.alphabet = ["a", "b"]
27         self.assertTrue(slp.scan("abab"))
28         self.assertTrue(slp.scan("ab"))
29         self.assertTrue(slp.scan("ababababab"))
30         self.assertFalse(slp.scan("abb"))
31         self.assertFalse(slp.scan("a"))
32         self.assertFalse(slp.scan(""))
33
34     def test_scan_neg(self):
35         """Checks if well-formed strings are detected correctly
36         given the
37         provided negative grammar."""
38         sln = SL(polar="n")
39         sln.grammar = [("<b", "a"), ("a", "<b")]
40         sln.alphabet = ["a", "b"]
41         self.assertFalse(sln.scan("abab"))
42         self.assertFalse(sln.scan("ab"))
43         self.assertFalse(sln.scan("ababababab"))
44         self.assertTrue(sln.scan("bbbb"))
45         self.assertTrue(sln.scan("aaaaa"))
46         self.assertTrue(sln.scan(""))
47
48     def test_ngramize_2(self):
49         """Checks if ngramize() correctly constructs bigrams."""
50         sl = SL()
51         sl.data = ["aaa", "bbb"]

```

```

49         ngrams = set(sl.ngramize_data())
50         goal = {(">", "b"), (">", "a"), ("a", "a"), ("b", "b"), ("a"
, "<"), ("b", "<")}
51         self.assertTrue(ngrams == goal)
52
53     def test_ngramize_3(self):
54         """Check if ngramize() correctly constructs trigrams."""
55         sl = SL()
56         sl.k = 3
57         sl.data = ["aaa", "bbb"]
58         ngrams = set(sl.ngramize_data())
59         goal = {
60             (">", "a", "a"),
61             (">", "b", "b"),
62             ("b", "b", "<"),
63             ("b", "<", "<"),
64             ("a", "<", "<"),
65             (">", ">", "a"),
66             ("a", "a", "a"),
67             ("a", "a", "<"),
68             (">", ">", "b"),
69             ("b", "b", "b"),
70         }
71         self.assertTrue(ngrams == goal)
72
73     def test_learn(self):
74         """Checks if positive and negative grammars are learned
correctly."""
75         data = ["abab", "ababab"]
76         gpos = {(">", "a"), ("b", "a"), ("a", "b"), ("b", "<")}
77         gneg = {(">", "<"), ("a", "<"), (">", "b"), ("b", "b"), ("a"
, "a")}
78
79         a = SL(data=data, alphabet=["a", "b"])

```

```

80         a.learn()
81         self.assertTrue(set(a.grammar) == gpos)
82
83         a.change_polarity()
84         a.learn()
85         self.assertTrue(set(a.grammar) == gneg)
86
87     def test_fsmize_pos(self):
88         """Checks if the transitions of the fsm corresponding to the
89         positive
90         grammar are constructed correctly."""
91         sl = SL(polar="p")
92         sl.alphabet = ["a", "b"]
93         sl.grammar = [(">", "a"), ("b", "a"), ("a", "b"), ("b", "<")]
94
95         sl.fsmize()
96
97         f = FSM(initial=">", final="<")
98         f.sl_to_fsm([(">", "a"), ("b", "a"), ("a", "b"), ("b", "<")
99
100         ])
101
102         self.assertTrue(set(sl.fsm.transitions) == set(f.transitions
103
104         ))
105
106     def test_fsmize_neg(self):
107         """Checks if the transitions of the fsm corresponding to the
108         negative
109         grammar are constructed correctly."""
110         sl = SL()
111         sl.change_polarity("n")
112         sl.alphabet = ["a", "b"]
113         sl.grammar = [(">", "<"), ("a", "<"), (">", "b"), ("b", "b")
114
115         , ("a", "a")]
116
117         sl.fsmize()

```

```

108
109     f = FSM(initial=">", final="<")
110     f.sl_to_fsm([(">", "a"), ("b", "a"), ("a", "b"), ("b", "<")
111 ])
112
113
114     self.assertTrue(set(sl.fsm.transitions) == set(f.transitions
115 ))
116
117
118     def test_generate_sample(self):
119         """Checks if all generated data points are actually well-
120         formed with
121         respect to the given grammar, and that the number of
122         generated data
123         points is correct."""
124         sl = SL()
125         sl.alphabet = ["a", "b"]
126         sl.grammar = [(">", "a"), ("b", "a"), ("a", "b"), ("b", "<")
127 ]
128
129         sl.fsmize()
130
131
132         sample = sl.generate_sample(n=10)
133         self.assertTrue(all([sl.scan(i) for i in sample]))
134         self.assertTrue(len(sample) == 10)
135
136
137     def test_switch_polarity(self):
138         """Makes sure that switch_polarity actually switches the
139         grammar to the
140         opposite, and that switching it again will result in the
141         original
142         grammar."""
143         gpos = {(">", "a"), ("b", "a"), ("a", "b"), ("b", "<")}
144         gneg = {(">", "<"), ("a", "<"), (">", "b"), ("b", "b"), ("a",
145 , "a")}
146         sl = SL(polar="n")

```



```

134     sl.alphabet = ["a", "b"]
135     sl.grammar = list(gneg)
136
137     sl.switch_polarity()
138     self.assertTrue(set(sl.grammar) == gpos)
139     self.assertTrue(sl.check_polarity() == "p")
140
141     sl.switch_polarity()
142     self.assertTrue(set(sl.grammar) == gneg)
143     self.assertTrue(sl.check_polarity() == "n")
144
145     def test_clean_grammar_2_pos(self):
146         """Tests if clean_grammar correctly cleans 2-local positive
147         SL
148         grammar."""
149         goal = {(">", "a"), ("b", "a"), ("a", "b"), ("b", "<")}
150         s = SL()
151         s.grammar = [
152             (">", "a"),
153             ("b", "a"),
154             ("a", "b"),
155             ("b", "<"),
156             (">", "g"),
157             ("f", "<"),
158             ("t", "t"),
159         ]
160         s.extract_alphabet()
161         s.clean_grammar()
162         self.assertTrue(set(s.grammar) == goal)
163
164     def test_clean_grammar_2_neg(self):
165         """Tests if clean_grammar correctly cleans 2-local negative
166         SL
167         grammar."""

```

```

166     goal = {(">", "<"), ("a", "<"), (">", "b"), ("b", "b"), ("a"
, "a")}
167     a = SL(polar="n")
168     a.alphabet = ["a", "b"]
169     a.grammar = [
170        (">", "<"),
171        ("a", "<"),
172        (">", "b"),
173        ("b", "b"),
174        ("a", "a"),
175        (">", "<"),
176        ("b", "b"),
177     ]
178     a.clean_grammar()
179     self.assertTrue(set(a.grammar) == goal)
180
181     def test_clean_grammar_3_pos(self):
182         """Tests if clean_grammar correctly cleans 2-local positive
SL
183         grammar."""
184         goal = {
185            (">", "a", "a"),
186            (">", "b", "b"),
187            ("b", "b", "<"),
188            ("b", "<", "<"),
189            ("a", "<", "<"),
190            (">", ">", "a"),
191            ("a", "a", "a"),
192            ("a", "a", "<"),
193            (">", ">", "b"),
194            ("b", "b", "b"),
195         }
196         s = SL()
197         s.grammar = [

```

```

198        (">", "a", "a"),
199        (">", "b", "b"),
200        ("b", "b", "<"),
201        ("b", "<", "<"),
202        ("a", "<", "<"),
203        (">", ">", "a"),
204        ("a", "a", "a"),
205        ("a", "a", "<"),
206        (">", ">", "b"),
207        ("b", "b", "b"),
208        (">", ">", "f"),
209        ("b", "d", "c"),
210     ]
211     s.extract_alphabet()
212     s.clean_grammar()
213     self.assertTrue(set(s.grammar) == goal)
214
215
216 if __name__ == "__main__":
217     unittest.main()

```

Unit test for SP

```

1 #!/bin/python3
2
3 """A module with the unittests for the SP module. Copyright (C) 2019
4     Alena
5     Aksenova.
6
7 This program is free software; you can redistribute it and/or modify
8 it
9 under the terms of the GNU General Public License as published by
10 the
11 Free Software Foundation; either version 3 of the License, or (at
12 your

```

```

9 option) any later version.
10 """
11
12 import unittest
13 from sp_class import *
14
15
16 class TestSPLanguages(unittest.TestCase):
17     """Tests for the SP class."""
18
19     def test_subsequences_2(self):
20         """Tests extraction of 2-subsequences."""
21         str1 = "abab"
22         ssq1 = {("a", "a"), ("a", "b"), ("b", "a"), ("b", "b")}
23         str2 = "a"
24         ssq2 = set()
25         str3 = "abcde"
26         ssq3 = {
27             tuple(i)
28             for i in ["ab", "ac", "ad", "ae", "bc", "bd", "be", "cd",
29 , "ce", "de"]
30         }
31         sp = SP()
32         self.assertTrue(set(sp.subsequences(str1)) == ssq1)
33         self.assertTrue(set(sp.subsequences(str2)) == ssq2)
34         self.assertTrue(set(sp.subsequences(str3)) == ssq3)
35
36     def test_subsequences_3(self):
37         """Tests extraction of 3-subsequences."""
38         str1 = "abab"
39         ssq1 = {tuple(i) for i in ["aba", "abb", "bab", "aab"]}
40         str2 = "abcde"
41         ssq2 = {
42             tuple(i)

```

```

42         for i in [
43             "abc",
44             "abd",
45             "abe",
46             "acd",
47             "ace",
48             "ade",
49             "bcd",
50             "bce",
51             "bde",
52             "cde",
53         ]
54     }
55     sp = SP(k=3)
56     self.assertTrue(set(sp.subsequences(str1)) == ssq1)
57     self.assertTrue(set(sp.subsequences(str2)) == ssq2)
58
59     def test_learn_pos(self):
60         """Tests learning of the positive grammar."""
61         data = ["abab", "abcde"]
62         goal = {
63             tuple(i)
64             for i in [
65                 "aba",
66                 "abb",
67                 "bab",
68                 "aab",
69                 "abc",
70                 "abd",
71                 "abe",
72                 "acd",
73                 "ace",
74                 "ade",
75                 "bcd",

```

```

76         "bce",
77         "bde",
78         "cde",
79     ]
80 }
81 sp = SP(k=3)
82 sp.data = data
83 sp.alphabet = ["a", "b", "c", "d", "e"]
84 sp.learn()
85 self.assertTrue(set(sp.grammar) == goal)
86
87 def test_learn_neg(self):
88     """Tests learning of the negative grammar."""
89     data = ["aaaaabbbb", "abbbb", "aaab"]
90     goal = {tuple("ba")}
91     sp = SP(polar="n")
92     sp.data = data
93     sp.alphabet = ["b", "a"]
94     sp.learn()
95     self.assertTrue(set(sp.grammar) == goal)
96
97 def test_change_polarity(self):
98     """Tests change_polarity function."""
99     sp1 = SP(polar="p")
100    sp1.change_polarity()
101    self.assertTrue(sp1.check_polarity() == "n")
102
103    sp2 = SP()
104    sp2.change_polarity("p")
105    sp2.change_polarity()
106    self.assertTrue(sp2.check_polarity() == "n")
107
108    sp3 = SP(polar="n")
109    sp3.change_polarity()

```

```

110     self.assertTrue(sp3.check_polarity() == "p")
111
112     sp4 = SP()
113     sp4.change_polarity("n")
114     sp4.change_polarity()
115     self.assertTrue(sp4.check_polarity() == "p")
116
117     sp5 = SP()
118     sp5.change_polarity("p")
119     self.assertTrue(sp5.check_polarity() == "p")
120
121     def test_scan_neg(self):
122         """Tests if automata correctly recognize illicit
123         substructures."""
124
125         sp = SP(polar="n")
126         sp.grammar = [tuple("aba")]
127         sp.k = 3
128         sp.extract_alphabet()
129         sp.fsmize()
130
131         self.assertTrue(sp.scan("aaaa"))
132         self.assertTrue(sp.scan("aaabbbbb"))
133         self.assertTrue(sp.scan("baaaaaabbbbb"))
134         self.assertTrue(sp.scan("a"))
135         self.assertTrue(sp.scan("b"))
136
137         self.assertFalse(sp.scan("aaaabaabbbba"))
138         self.assertFalse(sp.scan("abababba"))
139         self.assertFalse(sp.scan("abbbbabbaababab"))
140
141     def test_generate_item(self):
142         """Tests string generation."""
143
144         sp = SP(polar="n")
145         sp.grammar = [tuple("aba")]

```

```

143         sp.k = 3
144         sp.extract_alphabet()
145         sp.fsmize()
146
147         for i in range(30):
148             self.assertTrue(sp.scan(sp.generate_item()))
149
150     def test_generate_sample_pos(self):
151         """Tests sample generation when the grammar is positive."""
152         sp = SP()
153         sp.grammar = [tuple(i) for i in ["ab", "ba", "bb"]]
154         sp.extract_alphabet()
155         sp.fsmize()
156
157         a = sp.generate_sample(n=10)
158         self.assertTrue(len(a) == 10)
159
160     def test_generate_sample_neg(self):
161         """Tests sample generation when the grammar is negative."""
162         sp = SP(polar="n")
163         sp.grammar = [tuple("aba")]
164         sp.k = 3
165         sp.extract_alphabet()
166         sp.fsmize()
167
168         a = sp.generate_sample(n=15, repeat=False)
169         self.assertTrue(len(set(a)) == 15)
170
171
172 if __name__ == "__main__":
173     unittest.main()

```

Unit test for TSL

```

1 #!/bin/python3

```



```

2
3 """A module with the unittests for the TSL module. Copyright (C)
   2019 Alena
4 Aksenova.
5
6 This program is free software; you can redistribute it and/or modify
   it
7 under the terms of the GNU General Public License as published by
   the
8 Free Software Foundation; either version 3 of the License, or (at
   your
9 option) any later version.
10 """
11
12 import unittest
13 from tsl_class import *
14
15
16 class TestTSLLanguages(unittest.TestCase):
17     """Tests for the TSL class."""
18
19     def test_tier_learning(self):
20         """Tests the tier learning function."""
21         a = TSL()
22         a.data = ["aaaab", "abaaaa", "b"]
23         a.alphabet = ["a", "b"]
24         a.learn_tier()
25         self.assertTrue(a.tier == ["b"])
26
27         b = TSL()
28         b.data = ["ccaccaccbc", "acbbaababc", "ababbab"]
29         b.alphabet = ["a", "b", "c"]
30         b.learn_tier()
31         self.assertTrue(set(b.tier) == {"a", "b"})

```

```

32
33     def test_tier_learning_raised_issue(self):
34         """Checks a specific case related to GitHub issue #6."""
35         tsl = TSL()
36         tsl.data = [
37             "aa", "ab", "ax", "ay",
38             "ba", "bb", "bx", "by",
39             "xa", "xb", "xx",
40             "ya", "yb", "yx", "yy"
41         ]
42         tsl.alphabet = ["a", "b", "x", "y"]
43         tsl.learn_tier()
44         self.assertTrue(set(tsl.tier) == {"x", "y"})
45
46     def test_tier_image(self):
47         """Tests the erasing function."""
48         a = TSL()
49         a.tier = ["a"]
50         self.assertTrue(a.tier_image("cvamda") == "aa")
51
52     def test_learn_pos(self):
53         """Tests learning of the positive TSL grammar."""
54         a = TSL()
55         a.data = [
56             "o",
57             "oko",
58             "a",
59             "aka",
60             "oo",
61             "aa",
62             "kak",
63             "kok",
64             "kk",
65             "kkakka",

```

```

66         "akk",
67         "kkokko",
68         "okk",
69     ]
70     a.extract_alphabet()
71     a.learn()
72     goal = {
73        (">", "<"),
74        (">", "a"),
75        ("a", "a"),
76        (">", "o"),
77        ("o", "o"),
78        ("a", "<"),
79        ("o", "<"),
80     }
81     self.assertTrue(set(a.grammar) == goal)
82     self.assertTrue(set(a.tier) == {"a", "o"})
83
84     def test_learn_neg(self):
85         """Tests learning of the negative TSL grammar."""
86         a = TSL(polar="n")
87         a.data = [
88             "o",
89             "oko",
90             "a",
91             "aka",
92             "oo",
93             "aa",
94             "kak",
95             "kok",
96             "kk",
97             "kkakka",
98             "akk",
99             "kkokko",

```

```

100         "okk",
101     ]
102     a.extract_alphabet()
103     a.learn()
104     goal = {("a", "o"), ("o", "a")}
105     self.assertTrue(set(a.grammar) == goal)
106     self.assertTrue(set(a.tier) == {"a", "o"})
107
108     def test_scan_pos(self):
109         """Tests recognition of strings."""
110         a = TSL(polar="p")
111         a.data = [
112             "o",
113             "oko",
114             "a",
115             "aka",
116             "oo",
117             "aa",
118             "kak",
119             "kok",
120             "kk",
121             "kkakka",
122             "akk",
123             "kkokko",
124             "okk",
125         ]
126         a.extract_alphabet()
127         a.learn()
128         self.assertTrue(a.scan("akkaka"))
129         self.assertTrue(a.scan("kkk"))
130         self.assertTrue(a.scan("okoko"))
131         self.assertTrue(a.scan("ookokkk"))
132         self.assertFalse(a.scan("okoak"))
133         self.assertFalse(a.scan("okakok"))

```

```

134         self.assertFalse(a.scan("kakokak"))
135
136     def test_scan_neg(self):
137         """Tests recognition of strings."""
138         a = TSL(polar="n")
139         a.data = [
140             "o",
141             "oko",
142             "a",
143             "aka",
144             "oo",
145             "aa",
146             "kak",
147             "kok",
148             "kk",
149             "kkakka",
150             "akk",
151             "kkokko",
152             "okk",
153         ]
154         a.extract_alphabet()
155         a.learn()
156         self.assertTrue(a.scan("akkaka"))
157         self.assertTrue(a.scan("kkk"))
158         self.assertTrue(a.scan("okoko"))
159         self.assertTrue(a.scan("ookokkk"))
160         self.assertFalse(a.scan("okoak"))
161         self.assertFalse(a.scan("okakok"))
162         self.assertFalse(a.scan("kakokak"))
163
164     def test_generate_item_pos(self):
165         """Tests that the generated items are grammatical."""
166         a = TSL(polar="p")
167         a.data = [

```

```

168         "o",
169         "oko",
170         "a",
171         "aka",
172         "oo",
173         "aa",
174         "kak",
175         "kok",
176         "kk",
177         "kkakka",
178         "akk",
179         "kkokko",
180         "okk",
181     ]
182     a.extract_alphabet()
183     a.learn()
184     gen_items = [a.generate_item() for i in range(15)]
185     for i in gen_items:
186         self.assertTrue(a.scan(i))
187
188     def test_generate_item_neg(self):
189         """Tests that the generated items are grammatical."""
190         a = TSL(polar="n")
191         a.data = [
192             "o",
193             "oko",
194             "a",
195             "aka",
196             "oo",
197             "aa",
198             "kak",
199             "kok",
200             "kk",
201             "kkakka",

```

```

202         "akk",
203         "kkokko",
204         "okk",
205     ]
206     a.extract_alphabet()
207     a.learn()
208     gen_items = [a.generate_item() for i in range(15)]
209     for i in gen_items:
210         self.assertTrue(a.scan(i))
211
212     def test_change_polarity_pos_to_neg(self):
213         """Checks that the polarity switching works."""
214         a = TSL(polar="p")
215         a.grammar = [
216            (">", "o"),
217            ("a", "<"),
218            ("a", "a"),
219            ("o", "o"),
220            ("o", "<"),
221            (">", "a"),
222            (">", "<"),
223         ]
224         a.tier = ["a", "o"]
225         a.switch_polarity()
226         self.assertTrue(set(a.grammar) == {("a", "o"), ("o", "a")})
227         self.assertTrue(a.check_polarity() == "n")
228
229         b = TSL(polar="p")
230         b.data = ["aaaab", "abaaaa", "b"]
231         b.extract_alphabet()
232         b.learn()
233         b.switch_polarity()
234         self.assertTrue(set(b.grammar) == {("b", "b"), (">", "<")})
235         self.assertTrue(b.check_polarity() == "n")

```

```

236
237 def test_change_polarity_neg_to_pos(self):
238     """Checks that the polarity switching works."""
239     a = TSL(polar="n")
240     expected = {
241        (">", "o"),
242        ("a", "<"),
243        ("a", "a"),
244        ("o", "o"),
245        ("o", "<"),
246        (">", "a"),
247        (">", "<"),
248     }
249     a.grammar = [("a", "o"), ("o", "a")]
250     a.tier = ["a", "o"]
251     a.switch_polarity()
252     self.assertTrue(set(a.grammar) == expected)
253     self.assertTrue(a.check_polarity() == "p")
254
255     b = TSL(polar="n")
256     b.data = ["aaaab", "abaaaa", "b"]
257     b.extract_alphabet()
258     b.learn()
259     b.switch_polarity()
260     self.assertTrue(set(b.grammar) == {(">", "b"), ("b", "<")})
261     self.assertTrue(b.check_polarity() == "p")
262
263 def test_polarity_raised_issue(self):
264     """Checks a specific case from the GitHub issue."""
265     a = TSL(polar="p")
266     a.grammar = [(">", "a"), ("a", "b"), ("b", "<"), ("b", "a")]
267     a.tier = ["a", "b"]
268     a.switch_polarity()
269     expected = {("a", "a"), ("a", "<"), ("b", "b"), (">", "b"),

```



```

270         (">", "<")}
271
272         self.assertTrue(set(a.grammar) == expected)
273         self.assertTrue(a.check_polarity() == "n")
274
275     def test_generate_sample(self):
276         a = TSL(polar="p")
277         a.grammar = [("a", "a"), ("a", "b"), ("b", "<"), ("b", "a")]
278         a.tier = ["a", "b"]
279         a.alphabet = ["a", "b", "c"]
280
281         sample = a.generate_sample(n=10, repeat=False)
282         for i in sample:
283             self.assertTrue(a.scan(i))
284
285 if __name__ == "__main__":
286     unittest.main()

```

Unit test for MTSL

```

1  #!/bin/python3
2
3  """A module with the unit tests for the MTSL module. Copyright (C)
4      2019 Alena
5      Aksenova.
6
7      This program is free software; you can redistribute it and/or modify
8      it
9      under the terms of the GNU General Public License as published by
10     the
11     Free Software Foundation; either version 3 of the License, or (at
12     your
13     option) any later version.
14
15     """

```

```

12 import unittest
13 import unittest.mock
14 from mtsl_class import *
15
16
17 class TestMTSLLanguages(unittest.TestCase):
18     """Tests for the MTSL class."""
19
20     def test_grammar_learning_neg(self):
21         """Tests the learner."""
22         a = MTSL(polar="n")
23         VC = [
24             "aabbaabb",
25             "abab",
26             "aabbab",
27             "abaabb",
28             "aabaab",
29             "abbabb",
30             "ooppoopp",
31             "opop",
32             "ooppop",
33             "opoopp",
34             "oopoop",
35             "oppopp",
36             "aappaapp",
37             "apap",
38             "aappap",
39             "apaapp",
40             "aapaap",
41             "appapp",
42             "oobboobb",
43             "obob",
44             "oobbob",
45             "oboobb",

```

```

46         "ooboob",
47         "obbobb",
48         "aabb",
49         "ab",
50         "aab",
51         "abb",
52         "oopp",
53         "op",
54         "oop",
55         "opp",
56         "oobb",
57         "ob",
58         "oob",
59         "obb",
60         "aapp",
61         "ap",
62         "aap",
63         "app",
64         "aaa",
65         "ooo",
66         "bbb",
67         "ppp",
68         "a",
69         "o",
70         "b",
71         "p",
72         "",
73     ]
74     expected = {
75         ("a", "o"): [("a", "o"), ("o", "a")],
76         ("b", "p"): [("b", "p"), ("p", "b")],
77     }
78     a.data = VC[:]
79     a.extract_alphabet()

```

```

80         a.learn()
81
82         correct = True
83         for i in a.grammar:
84             if not (i in expected and set(a.grammar[i]) == set(
expected[i])):
85                 correct = False
86             if len(a.grammar) != len(expected):
87                 correct = False
88
89         self.assertTrue(correct)
90
91     def test_grammar_learning_pos(self):
92         """Tests the learner."""
93         b = MTSL(polar="p")
94         VC = [
95             "aabbaabb",
96             "abab",
97             "aabbab",
98             "abaabb",
99             "aabaab",
100             "abbabb",
101             "oopppopp",
102             "opop",
103             "oppop",
104             "oppopp",
105             "oopoop",
106             "oppopp",
107             "aappaapp",
108             "apap",
109             "aappap",
110             "apaapp",
111             "aapaap",
112             "appapp",

```

```
113         "oobboobb",
114         "obob",
115         "oobbob",
116         "oboobb",
117         "ooboob",
118         "obbobb",
119         "aabb",
120         "ab",
121         "aab",
122         "abb",
123         "oopp",
124         "op",
125         "oop",
126         "opp",
127         "oobb",
128         "ob",
129         "oob",
130         "obb",
131         "aapp",
132         "ap",
133         "aap",
134         "app",
135         "aaa",
136         "ooo",
137         "bbb",
138         "ppp",
139         "a",
140         "o",
141         "b",
142         "p",
143         "",
144     ]
145     expected2 = {
146         ("a", "o"): [
```

```

147         (">", "a"),
148         ("a", "<"),
149         ("a", "a"),
150         (">", "o"),
151         ("o", "o"),
152         ("o", "<"),
153         (">", "<"),
154     ],
155     ("b", "p"): [
156         (">", "b"),
157         ("b", "b"),
158         ("b", "<"),
159         (">", "p"),
160         ("p", "p"),
161         ("p", "<"),
162         (">", "<"),
163     ],
164 }
165
166 b.data = VC[:]
167 b.extract_alphabet()
168 b.learn()
169
170 correct = True
171 for i in b.grammar:
172     if not (i in expected2 and set(b.grammar[i]) == set(
173         expected2[i])):
174         correct = False
175         if len(b.grammar) != len(expected2):
176             correct = False
177
178 self.assertTrue(correct)
179
180 @unittest.mock.patch(

```

```

180         # Artificially enforce a particular case of list(set())'s
naturally-
181         # occurring non-determinism with respect to ordering:
182         # make it ascending if odd number of elements, descending if
even.
183
184         # While impractical, this re-implementation of list(set())
is perfectly
185         # legal. It could be discarded, but that way, the test
becomes
186         # non-deterministic and reveals the bug only in some 10% of
runs.
187
188         "mtsl_class.list",
189         new=lambda x: sorted(x, reverse=len(x) % 2 == 0) \
190             if type(x) == set else list(x)
191     )
192     def test_grammar_learning_raised_issue(self):
193         """Checks a specific case related to GitHub issue #6."""
194         mtsl = MTSL(k=2, polar="n")
195         mtsl.data = ["axb", "ayxb", "azxb", "azxyb"]
196         mtsl.extract_alphabet()
197         mtsl.learn()
198         self.assertTrue(all({*tier} == {"a", "b", "x"} for tier,
restrict \
199                             in mtsl.grammar.items() if ("a", "b") in
restrict))
200
201     def test_convert_pos_to_neg(self):
202         """Tests conversion of a positive grammar to a negative one.
"""
203         z = MTSL(polar="p")
204         z.grammar = {
205             ("a", "o"): [

```

```

206         (">", "a"),
207         ("a", "<"),
208         ("a", "a"),
209         (">", "o"),
210         ("o", "o"),
211         ("o", "<"),
212         (">", "<"),
213     ],
214     ("b", "p"): [
215         (">", "b"),
216         ("b", "b"),
217         ("b", "<"),
218         (">", "p"),
219         ("p", "p"),
220         ("p", "<"),
221         (">", "<"),
222     ],
223 }
224 z.switch_polarity()
225 expected = {
226     ("a", "o"): [("a", "o"), ("o", "a")],
227     ("b", "p"): [("b", "p"), ("p", "b")],
228 }
229 self.assertTrue(z.grammar == expected)
230
231 def test_scan_pos(self):
232     """Tests scanning using a positive grammar."""
233     c = MTSL(polar="p")
234     c.grammar = {
235         ("a", "o"): [
236             (">", "a"),
237             ("a", "<"),
238             ("a", "a"),
239             (">", "o"),

```



```

240         ("o", "o"),
241         ("o", "<"),
242         (">", "<"),
243     ],
244     ("b", "p"): [
245         (">", "b"),
246         ("b", "b"),
247         ("b", "<"),
248         (">", "p"),
249         ("p", "p"),
250         ("p", "<"),
251         (">", "<"),
252     ],
253 }
254 for s in ["apapappa", "ppp", "appap", "popo", "bbbooo"]:
255     self.assertTrue(c.scan(s))
256 for s in ["aoap", "popa", "pbapop", "pabp", "popoa"]:
257     self.assertFalse(c.scan(s))
258
259 def test_scan_neg(self):
260     """Tests scanning using a positive grammar."""
261     d = MTS�(polar="n")
262     d.grammar = {
263         ("a", "o"): [("a", "o"), ("o", "a")],
264         ("b", "p"): [("b", "p"), ("p", "b")],
265     }
266     for s in ["apapappa", "ppp", "appap", "popo", "bbbooo"]:
267         self.assertTrue(d.scan(s))
268     for s in ["aoap", "popa", "pbapop", "pabp", "popoa"]:
269         self.assertFalse(d.scan(s))
270
271
272 if __name__ == "__main__":
273     unittest.main()

```

Unit test for FSA

```
1 #!/bin/python3
2
3 """A module with the unittests for the fsm module. Copyright (C)
4     2019 Alena
5     Aksenova.
6
7 This program is free software; you can redistribute it and/or modify
8 it
9 under the terms of the GNU General Public License as published by
10 the
11 Free Software Foundation; either version 3 of the License, or (at
12 your
13 option) any later version.
14 """
15
16 import unittest
17 from fsm import FSM
18
19 class TestFSM(unittest.TestCase):
20     """Tests for the FSM class."""
21
22     def test_sl_to_fsm_2(self):
23         """Checks if a 2-SL grammar translates to FSM correctly."""
24         f = FSM(initial=">", final="<")
25         grammar = [(">", "a"), ("b", "a"), ("a", "b"), ("b", "<")]
26         f.sl_to_fsm(grammar)
27
28         tr = {
29             (("a",),), "a", ("a",),),
30             (("b",),), "a", ("a",),),
31             (("a",),), "b", ("b",),),
```

```

29         (("b",), "<", ("<")),
30     }
31     self.assertTrue(set(f.transitions) == tr)
32
33     def test_sl_to_fsm_3(self):
34         """Checks if a 3-SL grammar translates to FSM correctly."""
35         f = FSM(initial=">", final="<")
36         grammar = [
37             (">", "a", "b"),
38             ("a", "b", "a"),
39             ("b", "a", "b"),
40             ("a", "b", "<"),
41             (">", ">", "a"),
42             ("b", "<", "<"),
43         ]
44         f.sl_to_fsm(grammar)
45
46         tr = {
47             ((">", "a"), "b", ("a", "b")),
48             (("a", "b"), "a", ("b", "a")),
49             (("b", "a"), "b", ("a", "b")),
50             (("a", "b"), "<", ("b", "<")),
51             ((">", ">"), "a", (">", "a")),
52             (("b", "<"), "<", ("<", "<")),
53         }
54         self.assertTrue(set(f.transitions) == tr)
55
56     def test_scan_sl_2(self):
57         """Checks if a FSM for 2-SL grammar can correctly recognize
58 strings."""
59         f = FSM(initial=">", final="<")
60         f.transitions = [
61             ((">",), "a", ("a",)),
62             (("b",), "a", ("a",)),

```

```

62         (("a",), "b", ("b",)),
63         (("b",), "<", ("<",)),
64     ]
65
66     self.assertTrue(f.scan_sl(">abab<"))
67     self.assertTrue(f.scan_sl(">ab<"))
68     self.assertTrue(f.scan_sl(">abababab<"))
69
70     self.assertFalse(f.scan_sl("><"))
71     self.assertFalse(f.scan_sl(">a<"))
72     self.assertFalse(f.scan_sl(">ba<"))
73     self.assertFalse(f.scan_sl(">ababbab<"))
74
75     def test_scan_sl_3(self):
76         """Checks if a FSM for 3-SL grammar can correctly recognize
77         strings."""
78         f = FSM(initial=">", final="<")
79         f.transitions = [
80             (">", "a", "b", ("a", "b")),
81             ("a", "b", "a", ("b", "a")),
82             ("b", "a", "b", ("a", "b")),
83             ("a", "b", "<", ("b", "<")),
84             (">", ">", "a", (">", "a")),
85             ("b", "<", "<", ("<", "<")),
86         ]
87
88         self.assertTrue(f.scan_sl(">>abab<<"))
89         self.assertTrue(f.scan_sl(">ab<"))
90         self.assertTrue(f.scan_sl(">>abababab<<"))
91
92         self.assertFalse(f.scan_sl(">><<"))
93         self.assertFalse(f.scan_sl(">>a<<"))
94         self.assertFalse(f.scan_sl(">>ba<<"))
95         self.assertFalse(f.scan_sl(">>ababbab<<"))

```

```

95
96     def test_trim_fsm_2(self):
97         f = FSM(initial=">", final("<"))
98         f.transitions = [
99             ((" ">,), "a", ("a",)),
100             (("b",), "a", ("a",)),
101             (("a",), "b", ("b",)),
102             (("b",), "<", ("<")),
103             ((" ">,), "c", ("c",)),
104             (("d",), "<", ("<")),
105         ]
106         goal = {
107             ((" ">,), "a", ("a",)),
108             (("b",), "a", ("a",)),
109             (("a",), "b", ("b",)),
110             (("b",), "<", ("<")),
111         }
112         f.trim_fsm()
113         self.assertTrue(set(f.transitions) == goal)
114
115     def test_trim_fsm_3(self):
116         f = FSM(initial=">", final("<"))
117         f.transitions = [
118             ((" ">, "a"), "b", ("a", "b")),
119             (("a", "b"), "a", ("b", "a")),
120             (("b", "a"), "b", ("a", "b")),
121             (("a", "b"), "<", ("b", "<")),
122             ((" ">, ">"), "a", (" ">, "a")),
123             (("b", "<"), "<", ("<", "<")),
124             ((" ">, "b"), "j", ("b", "j")),
125             ((" ">, ">"), "j", (" ">, "j")),
126             (("j", "k"), "o", ("k", "o")),
127         ]
128         goal = {

```

```

129         (( ">", "a"), "b", ("a", "b")),
130         (("a", "b"), "a", ("b", "a")),
131         (("b", "a"), "b", ("a", "b")),
132         (("a", "b"), "<", ("b", "<")),
133         (( ">", ">"), "a", (">", "a")),
134         (("b", "<"), "<", ("<", "<")),
135     }
136     f.trim_fsm()
137     self.assertTrue(set(f.transitions) == goal)
138
139
140 if __name__ == "__main__":
141     unittest.main()

```

Unit test for OSTIA

```

1  #!/bin/python3
2
3  """A module with the unittests for the fsm module. Copyright (C)
4      2020  Alena
5      Aksenova.
6
7  This program is free software; you can redistribute it and/or modify
8  it
9  under the terms of the GNU General Public License as published by
10 the
11 Free Software Foundation; either version 3 of the License, or (at
12 your
13 option) any later version.
14 """
15
16 import unittest
17 from ostia import ostia

```

```

16 class TestOSTIA(unittest.TestCase):
17     """Tests for the OSTIA learner.
18
19     Warning: updated versions of the learner might require updating
20     the unittests.
21     """
22
23     def test_ostia_success(self):
24         """Checks if OSTIA can learn a rule rewriting "a" as "1" if
25         "a" is
26         final and as "0" otherwise, and always mapping "b" to "1".
27         """
28         S = [
29             ("a", "1"),
30             ("b", "1"),
31             ("aa", "01"),
32             ("ab", "01"),
33             ("aba", "011"),
34             ("aaa", "001"),
35         ]
36         t = ostia(S, ["a", "b"], ["0", "1"])
37
38         transitions = {
39             ("", "a", "", "a"),
40             ("", "b", "1", ""),
41             ("a", "a", "0", "a"),
42             ("a", "b", "01", ""),
43         }
44         stout = {"": "", "a": "1"}
45
46         self.assertTrue(set(t.E) == transitions)
47         self.assertTrue(stout == t.stout)
48
49     def test_ostia_fail(self):

```

```

48     """Checks that OTSIA cannot learn an unbounded tone
plateauing."""
49     S = [
50         ("HHH", "HHH"),
51         ("HHL", "HHL"),
52         ("HLH", "HHH"),
53         ("HLL", "HLL"),
54         ("HLLH", "HHHH"),
55         ("HL", "HL"),
56     ]
57     t = ostia(S, ["H", "L"], ["H", "L"])
58
59     transitions = {
60         ("", "H", "H", "H"),
61         ("H", "H", "H", ""),
62         ("H", "L", "", "HL"),
63         ("HL", "H", "HH", ""),
64         ("HL", "L", "", "HLL"),
65         ("HLL", "H", "HHH", ""),
66         ("", "L", "L", ""),
67     }
68     stout = {"": "", "H": "", "HL": "L", "HLL": "LL"}
69
70     self.assertTrue(set(t.E) == transitions)
71     self.assertTrue(stout == t.stout)
72
73
74 if __name__ == "__main__":
75     unittest.main()

```


Bibliography

Alëna Aksënova. 2019. Ostia algorithm: implementation and tests. <https://github.com/alenaks/OSTIA/blob/master/ostia.ipynb>. [Online; accessed 22-March-2020].

Alëna Aksënova. 2020a. Brute force extraction of k-local isl transducers. https://github.com/alenaks/subregular-experiments/blob/master/ISL_group_learner.ipynb. [Online; accessed 22-March-2020].

Alëna Aksënova. 2020b. Sigmapie. <https://pypi.org/project/SigmaPie/>. SigmaPie for subregular and subsequential grammar induction. Python package available on PyPI.

Alëna Aksënova. 2020c. Sigmapie for subregular and subsequential grammar induction. <https://github.com/alenaks/SigmaPie>. [Online; accessed 22-March-2020].

Alëna Aksënova. 2020d. Subregular experiments. <https://github.com/alenaks/subregular-experiments>. [Online; accessed 22-March-2020].

Alëna Aksënova. 2020e. Tokenization with transducers. https://github.com/alenaks/OSTIA/blob/master/ostia_biased_outputs.ipynb. [Online; accessed 22-March-2020].

Alëna Aksënova and Sanket Deshmukh. 2018. Formal restrictions on multiple tiers. In *Proceedings of the Society for Computation in Linguistics (SCiL) 2018*, pages 64–73. Association for Computational Linguistics.

Alëna Aksënova, Thomas Graf, and Sedigheh Moradi. 2016. Morphotactics as tier-based strictly local dependencies. In *Proceedings of the 14th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pages 121–130. Association for Computational Linguistics.

Enes Avcu. 2017. Experimental investigation of the subregular hierarchy. In *Proceedings of the 35th West Coast Conference on Formal Linguistics at Calgary, Alberta Canada*.

- Srinivas Bangalore and Giuseppe Riccardi. 2002. Stochastic finite-state models for spoken language machine translation. *Machine Translation*, 17(3):165–184.
- Kenneth Beesley. 1996. Arabic finite-state morphological analysis and generation. In *Proceedings of COLING-96*, pages 89–94, Copenhagen, Denmark.
- Kenneth Beesley and Lauri Karttunen. 2003. *Finite state morphology*. CSLI Publications, Stanford, CA.
- Robert Blust. 2012. Hawu vowel metathesis. *Oceanic Linguistics*, 51(1):207–233.
- Wiebke Brockhaus. 1995. *Final Devoicing in the Phonology of German*. Max Niemeyer.
- Antoine Bruguier, Danushen Gnanapragasam, Leif Johnson, Kanishka Rao, and Francoise Beaufays. 2017. Pronunciation learning with rnn-transducers. In *INTERSPEECH*.
- Phillip Burness and Kevin McMullin. 2019. Efficient learning of output tier-based strictly 2-local functions. In *Proceedings of the 16th Meeting on the Mathematics of Language*, pages 78–90. Association for Computational Linguistics.
- Douglas Buzatto. 2016. Wordlists. <https://github.com/douglasbuzatto/WordLists>. [Online; accessed 28-March-2020].
- Diamantino Caseiro. 2003. *Finite-state methods in automatic speech recognition*. Ph.D. thesis, Instituto Superior Técnico, Universidade Técnica de Lisboa.
- Antonio Castellanos, Enrique Vidal, Miguel A. Varó, and José Oncina. 1998. Language understanding and subsequential transducer learning. *Computer Speech and Language*, 12:193–228.
- Jane Chandlee. 2014. *Strictly Local Phonological Processes*. Ph.D. thesis, University of Delaware.
- Jane Chandlee. 2017. Computational locality in morphological maps. *Morphology*, 27:599–641.
- Jane Chandlee, Remi Eyraud, and Jeffrey Heinz. 2014. Learning strictly local subsequential functions. *Transactions of the Association for Computational Linguistics*, 2:491–503.
- Jane Chandlee, Rémi Eyraud, and Jeffrey Heinz. 2015. Output strictly local functions. In *Proceedings of the 14th Meeting on the Mathematics of Language (MoL 2015)*, pages 112–125, Chicago, USA. Association for Computational Linguistics.

- Jane Chandlee, Remi Eyraud, Jeffrey Heinz, Adam Jardine, and Jonathan Rawski. 2019. Learning with partially ordered representations. In *Proceedings of the 16th Meeting on the Mathematics of Language*, pages 91–101, Toronto, Canada. Association for Computational Linguistics.
- Jane Chandlee, Jie Fu, Konstantinos Karydis, Cesar Koirala, Jeffrey Heinz, and Herbert G. Tanner. 2012. Integrating grammatical inference into robotic planning. In *Proceedings of the Eleventh International Conference on Grammatical Inference (ICGI 2012)*, volume 21, pages 69–83. JMLR Workshop and Conference Proceedings.
- Jane Chandlee and Jeffrey Heinz. 2018. Strict locality and phonological maps. *Linguistic Inquiry*, 49(1):23–60.
- Jane Chandlee and Adam Jardine. 2019. Autosegmental input strictly local functions. *Transactions of the Association for Computational Linguistics*, 7:157–168.
- Noam Chomsky. 1956. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124.
- Noam Chomsky. 1986. *Knowledge of Language: Its Nature, Origin, and Use*. Praeger, New York, NY.
- Noam Chomsky and Morris Halle. 1968. *The Sound Pattern of English*. Harper and Row, New York.
- Alonzo Church. 1936. A note on the entscheidungsproblem. *Journal of Symbolic Logic*, 1(1):40–41.
- Alexander Clark. 2006. Large scale inference of deterministic transductions: Tenjinno problem 1. In *Proceedings of the 8th International Colloquium on Grammatical Inference (ICGI)*, pages 227–239, Tokyo, Japan.
- Aniello De Santo and Thomas Graf. 2019. Structure sensitive tier projection: Applications and formal properties. In *Formal Grammar*, pages 35–50, Berlin, Heidelberg. Springer.
- Aniello De Santo, Thomas Graf, and John Drury. 2017. Evaluating subregular distinctions in the complexity of generalized quantifiers. Poster presented at the ESSLLI Workshop on Quantifiers and Determiners (QUAD 2017), July 17 – 21, University of Toulouse, France.
- Hossep Dolatian and Jeffrey Heinz. 2018. Modeling reduplication with 2-way finite-state transducers. In *Proceedings of the 15th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pages 66–77.

- Mohamed Elmedlaoui. 1995. *Aspects des représentations phonologiques dans certaines langues chamito-sémitiques*. Ph.D. thesis, Université Mohammed V.
- Cees Elzinga, Sven Rahmann, and Hui Wang. 2008. Algorithms for subsequence combinatorics. *Theoretical Computer Science*, 409:394–404.
- Markus Enzenberger. 2019. `german-wordlist`. <https://github.com/enz/german-wordlist>. [Online; accessed 28-March-2020].
- Jie Fu, Jeffrey Heinz, and Herbert G. Tanner. 2011. An algebraic characterization of strictly piecewise languages. In *Theory and Applications of Models of Computation*, volume 6648 of *Lecture Notes in Computer Science*, pages 252–263. Springer Berlin/Heidelberg.
- Brian Gainor, Regine Lai, and Jeffrey Heinz. 2012. Computational characterizations of vowel harmony patterns and pathologies. In *The Proceedings of the 29th West Coast Conference on Formal Linguistics*, pages 63–71.
- Daniel Gildea and Daniel Jurafsky. 1996. Learning bias and phonological-rule induction. *Computational Linguistics*, 22(4):497–530.
- E. Mark Gold. 1967. Language identification in the limit. *Information and Control*, 10:447–474.
- Thomas Graf. 2017a. It’s a (sub-)regular conspiracy: Locality and computation in phonology, morphology, syntax, and semantics. Slides of the CLS invited talk, May 26.
- Thomas Graf. 2017b. The power of locality domains in phonology. *Phonology*, 34:1–21.
- Thomas Graf. 2017c. Subregular morpho-semantics: The expressive limits of monomorphemic quantifiers. Invited talk, December 15, Rutgers University, New Brunswick, NJ.
- Thomas Graf. 2018a. Locality domains and phonological c-command over strings. In *Proceedings of NELS 2017*. To appear.
- Thomas Graf. 2018b. Why movement comes for free once you have adjunction. In *Proceedings of CLS 53*, pages 117–136.
- Thomas Graf. 2019. A subregular bound on the complexity of lexical quantifiers. In *Proceedings of the 22nd Amsterdam Colloquium*, pages 455–464.
- Thomas Graf and Nazila Shafiei. 2019. C-command dependencies as TSL string constraints. In *Proceedings of the Society for Computation in Linguistics (SCiL) 2019*, pages 205–215.

- Gunnar Olafur Hansson. 2010a. *Consonant Harmony: Long-Distance Interaction in Phonology*. University of California Press, Los Angeles.
- Gunnar Olafur Hansson. 2010b. Long-distance voicing assimilation in berber: spreading and/or agreement? In *Proceedings of the 2010 annual conference of the Canadian Linguistic Association*, Ottawa, Canada. Canadian Linguistic Association.
- K. David Harrison, Emily Thomforde, and Michael O’Keefen. 2004. The vowel harmony calculator.
- Jeffrey Heinz. 2010a. Learning long-distance phonotactics. *Linguistic Inquiry*, 41(4):623–661.
- Jeffrey Heinz. 2010b. String extension learning. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 897–906, Uppsala, Sweden. Association for Computational Linguistics.
- Jeffrey Heinz. 2011. Computational phonology part I: Grammars, learning, and the future. *Language and Linguistics Compass*, 5(4):140–152.
- Jeffrey Heinz. 2018. The computational nature of phonological generalizations. In Larry Hyman and Frans Plank, editors, *Phonological Typology*, Phonetics and Phonology, chapter 5, pages 126–195. De Gruyter Mouton.
- Jeffrey Heinz and Regine Lai. 2013. Vowel harmony and subsequentiality. In *Proceedings of the 13th Meeting on the Mathematics of Language (MoL 13)*, pages 52–63, Sofia, Bulgaria.
- Jeffrey Heinz, Chetan Rawal, and Herbert G. Tanner. 2011. Tier-based strictly local constraints for phonology. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics*, pages 58–64, Portland, USA. Association for Computational Linguistics.
- Jeffrey Heinz and James Rogers. 2010. Estimating strictly piecewise distributions. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 886–896, Uppsala, Sweden. Association for Computational Linguistics.
- Jeffrey Heinz and James Rogers. 2013. Learning subregular classes of languages with factored deterministic automata. In *Proceedings of the 13th Meeting on the Mathematics of Language (MoL 13)*, pages 64–71, Sofia, Bulgaria. Association for Computational Linguistics.
- Lee Hetherington. 2001. An efficient implementation of phonological rules using finite-state transducers. In *Proceedings of Eurospeech 2001*, Aarhus, Denmark.

- Colin de la Higuera. 2010. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, New York, NY, USA.
- Colin de la Higuera. 2011. Errata to grammatical inference: Learning automata and grammars. Errata. LINA, University of Nantes.
- John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- Mans Hulden. 2014. Finite state languages. In Mark Aronoff, editor, *Oxford Bibliographies in Linguistics*. Oxford University Press, New York, NY.
- Harry van der Hulst and Noval Smith. 1987. Vowel harmony in khalkha and buriat (east mongolian). *Linguistics in the Netherlands*, pages 81–89.
- Larry Hyman. 2011. Tone: is it different? In John Goldsmith, Jason Riggle, and Alan Yu, editors, *The handbook of phonological theory*, pages 197–239. Wiley-Blackwell, Oxford.
- Larry Hyman and Francis Katamba. 2010. Tone, syntax, and prosodic domains in luganda. In Laura J. Downing, editor, *ZAS Papers in Linguistics 53*, pages 69–98. ZASPil, Berlin.
- Gerhard Jäger and James Rogers. 2012. Formal language theory: Refining the chomsky hierarchy. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 367(1598):1956–1970.
- Adam Jardine. 2016a. Computationally, tone is different. *Phonology*, 33(2):247–283.
- Adam Jardine. 2016b. Learning tiers for long-distance phonotactics. In *Proceedings of the 6th Conference on Generative Approaches to Language Acquisition North America (GALANA 2015)*, pages 60–72, Somerville, MA. Cascadilla Proceedings Project.
- Adam Jardine, Jane Chandlee, Rémi Eyraud, and Jeffrey Heinz. 2014. Very efficient learning of structured classes of subsequential functions from positive data. *JMLR: Workshop and Conference Proceedings*, 34:94–108.
- Adam Jardine and Jeffrey Heinz. 2016. Learning tier-based strictly 2-local languages. *Transactions of the Association for Computational Linguistics*, 4:87–98.
- Adam Jardine and Kevin McMullin. 2017. Efficient learning of tier-based strictly k -local languages. *Lecture Notes in Computer Science*, 10168:64–76.

- Charles Douglas Johnson. 1972. *Formal Aspects of Phonological Description*. The Hague, Mouton.
- Bevan K. Jones, Mark Johnson, and Sharon Goldwater. 2011. Formalizing semantic parsing with tree transducers. In *Proceedings of Australasian Language Technology Association Workshop*, pages 19–28.
- Bevan K. Jones, Mark Johnson, and Sharon Goldwater. 2012. Semantic parsing with bayesian tree transducers. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics*, pages 488–496.
- Brian D. Joseph and Irene Philippaki-Warbuton. 1987. *Modern Greek*. Croom Helm, Wolfeboro, NH.
- Aravind K. Joshi. 1985. Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions? In David R. Dowty, Lauri Karttunen, and Arnold M. Zwicky, editors, *Natural Language Parsing*, pages 206–250. Cambridge University Press.
- Daniel Jurafsky and James Martin. 2009. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. Second Edition*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
- Laura Kallmeyer. 2010. On mildly context-sensitive non-linear rewriting. *Research on Language and Computation*, 8(4):341–363.
- Ronald M. Kaplan and Martin Kay. 1981. Phonological rules and finite-state transducers.
- Ronald M. Kaplan and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378.
- Ayla Karakaş. 2020. An ibsp description of sanskrit /n/-retroflexion. In *Proceedings of the Society for Computation in Linguistics*, volume 3, pages 160–169, New Orleans, LA.
- Abigail Rhoades Kaun. 1995. *The typology of rounding harmony: an optimality theoretic approach*. Ph.D. thesis, UCLA.
- Martin Kay. 1987. Nonconcatenative finite-state morphology. In *Proceedings of the Third Conference of the European Chapter of the ACL (EACL-87)*, pages 2–10, Copenhagen, Denmark. ACL.
- George Anton Kiraz. 1996. *Computational Approach to Non-Linear Morphology*. Ph.D. thesis, University of Cambridge.

- Stephen Cole Kleene. 1956. Representation of events in nerve nets and finite automata. In Claude Shannon and John McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, NJ.
- Kevin Knight and Yaser Al-Onaizan. 1998. Translation with finite-state devices. In *Lecture Notes in Computer Science 1529*. Springer Verlag.
- Kimmo Koskenniemi. 1983. *Two-Level Morphology: A general computational model for word-form recognition and production*. Ph.D. thesis, University of Helsinki.
- Martin Krämer. 2003. *Vowel Harmony and Correspondence Theory*. Mouton de Gruyter.
- Regine Lai. 2015. Learnable vs. unlearnable harmony patterns. *Linguistic Inquiry*, 46(3):425–451.
- Dakotah Lambert and James Rogers. 2020. Tier-based strictly local stringsets: Perspectives from model and automata theory. In *Proceedings of the Society for Computation in Linguistics: vol. 3*, pages 330–337, New Orleans, Louisiana.
- Andrew Lamont, Charlie O’Hara, and Caitlin Smith. 2019. Weakly deterministic transformations are subregular. In *Proceedings of the 16th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pages 196–205. Association for Computational Linguistics.
- Mark V. Lawson. 2003. *Finite Automata*. Taylor & Francis.
- Susannah V. Levi. 2001. Tier-based strictly local constraints for phonology. In *CLS 37: The Main Session*, pages 379–393, Chicago. Chicago Linguistic Society.
- Huan Luo. 2017. Long-distance consonant agreement and subsequentiality. *Glossa*, 2(1):52.
- Shakuntala Mahanta. 2007. *Directionality and locality in vowel harmony*. LOT, Utrecht, Netherlands.
- Warren McCulloch and Walter Pitts. 1943. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5(4):115–133.
- Kevin McMullin, Alëna Aksënova, and Aniello De Santo. 2019. Learning phonotactic restrictions on multiple tiers. *Proceedings of the Society for Computation in Linguistics*, 2:377–378.
- Kevin James McMullin. 2016. *Tier-based locality in long-distance phonotactics: learnability and typology*. Ph.D. thesis, University of British Columbia.

- Robert McNaughton and Seymour A. Papert. 1971. *Counter-Free Automata (M.I.T. Research Monograph No. 65)*. The MIT Press.
- Mehryar Mohri, Fernando Pereira, and Michael Riley. 2002. Weighted finite-state transducers in speech recognition. *Computer Speech and Language*, 16(1):69–88.
- Mehryar Mohri, Fernando Pereira, and Michael Riley. 2008. Speech recognition with weighted finite-state transducers. In Larry Rabiner and Fred Juang, editors, *Handbook on Speech Processing and Speech Communication, Part E: Speech recognition*. Springer-Verlag, Heidelberg, Germany.
- Kemelbek Nanaev. 1950. *Uchebnik kirgizskogo yazika*. Kirgizgosizdat, Frunze.
- David Odden. 1994. Adjacency parameters in phonology. *Language*, 70(2):289–330.
- José Oncina, Antonio Castellanos, Enrique Vidal, and Víctor Jimenez. 1994. Corpus-based machine translation through subsequential transducers. In *Proceedings of the Third International Conference on the Cognitive Science of Natural Language Processing*, Dublin, Ireland.
- José Oncina, Pedro García, and Enrique Vidal. 1993. Learning subsequential transducers for pattern recognition tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15:448–458.
- José Oncina and Miguel A. Varó. 1996. Using domain information during the learning of a subsequential transducer. In *Lecture Notes in Computer Science – Lecture Notes in Artificial Intelligence*, pages 313–325.
- Jaye Padgett. 2002. Russian voicing assimilation, final devoicing, and the problem of [v] (or, the mouse that squeaked)*. Manuscript.
- Markus Pöchtrager. 2010. Does turkish diss harmony? *Acta Linguistica Hungarica*, 57(4):458–473.
- Nicholas Poppe. 1960. *Buriat grammar. Uralic and Altaic series*. Indiana University, Bloomington.
- Eduardo Rivail Ribeiro. 2002. Directionality in vowel harmony: The case of karaja (macro-je). In *Proceedings of BLS 28*, pages 475–485.
- Brian Roark and Richard Sproat. 2007. *Computational approaches to syntax and morphology*. Oxford University Press, Oxford.
- Emmanuel Roche and Yves Schabes. 1997. Introduction. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*, chapter 1, pages 2–66. The MIT Press, Cambridge, MA.

- James Rogers. 2018. On the cognitive complexity of phonotactic constraints. Slides of a Stony Brook Colloquium. March 23.
- James Rogers, Jeffrey Heinz, Gil Bailey, Matt Edlefsen, Molly Visscher, David Wellcome, and Sean Wibel. 2010. On languages piecewise testable in the strict sense. In *The Mathematics of Language*, volume 6149 of *Lecture Notes in Artificial Intelligence*, pages 255–265. Springer.
- James Rogers, Jeffrey Heinz, Margaret Fero, Jeremy Hurst, Dakotah Lambert, and Sean Wibel. 2013. *Cognitive and Sub-regular Complexity*, chapter Formal Grammar. Springer.
- James Rogers and Geoffrey Pullum. 2011. Aural pattern recognition experiments and the subregular hierarchy. *Journal of Logic, Language and Information*, 20:329–342.
- Sharon Rose and Rachel Walker. 2011. Harmony systems. In John Goldsmith, Jason Riggle, and Alan C. L. Yu, editors, *The Handbook of Phonological Theory*, chapter 8, pages 240–290. Wiley-Blackwell, Cambridge, MA.
- Marcel-Paul Schützenberger. 1961. A remark on finite transducers. *Information and Control*, 4:185–196.
- Stuart Shieber. 1985. Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8:333–345.
- Michael Sipser. 2013. *Introduction to the Theory of Computation. Third edition.* Cengage Learning, Boston, MA.
- Elena Skribnik. 2003. Buryat. In Juha Janhunen, editor, *The Mongolic Languages*, chapter 5, pages 102–128. Routledge, London.
- Jan-Olof Svantesson, A. Tsendina, A. Karlsson, and V. Franzen. 2005. *The Phonology of Mongolian.* Oxford University Press, Oxford.
- Ken Thompson. 1968. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422.
- Alan Turing. 1937a. Computability and λ -definability. *Journal of Symbolic Logic*, 2(4):153–163.
- Alan Turing. 1937b. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265.

- Mai Ha Vu, Nazila Shafiei, and Thomas Graf. 2019. Case assignment in TSL syntax: A case study. In *Proceedings of the Society for Computation in Linguistics (SCiL) 2019*, pages 267–276.
- Rachel Walker. 2000. Yaka nasal harmony: Spreading or segmental correspondence? In *Proceedings of the Annual Meeting of the Berkeley Linguistics Society 26*, pages 321–332.