Case Inflection in Koalib : discovering the Rules

Georgi Boychev

University of Lorraine

Master Thesis

Case Inflection in Koalib: Discovering the Rules

Author:
Georgi Boychev

Supervisors:
Claire Gardent
Gosse Bouma
Nicolas Quint

June 19, 2013

Contents

1 Introduction

The goal of this master thesis is to discover the rules that govern object-case inflection in Koalib. Koalib is an African language with two grammatical cases - subject case and object case. It seems that the object forms of nouns are derived from their subject forms by following a set of rules which involve the ending of the noun, its syllable schema and tonal pattern. However, due to the complexity of the rules, it is very difficult to discover all of them using manual analysis. This study demonstrates how machine learning methods can be applied to discover those rules.

The rules are discovered from a corpus of about 1,200 Koalib nouns, which contains their subject forms, object forms, root morphemes and tonal patterns. This corpus is used to train a classifier, which predicts the change from subject form to object form. The classifier is trained using a rule-learning algorithm, which produces a list of rules as a model. The trained model is then evaluated on unseen data, in order to estimate the prediction accuracy of the generated rules. These automatically discovered rules can correctly predict the exact object form of unfamiliar nouns in about 66% of the cases.

This thesis is organized as follows. The section *Background* gives an overview of the Koalib language and discusses the characteristics that are relevant to the study, then gives an introduction to machine learning and the algorithms used in the study. The section *Methods* describes the methodology used to discover the rules and explains the classes and features for the machine learning model. The section *Experiments* reports on the most important experiments, conducted during the thesis project, and discusses their results. Finally, the section *Conclusion* summarizes the findings of this study and offers points for future investigation.

2 Background

This section summarizes some of the most important previous studies on the Koalib language. It also gives an overview of the language and discusses some important characteristics of its grammar and phonology. After that, it introduces machine learning and some of its terminology, then describes the algorithms that are used in the study.

2.1 Koalib

Koalib (also spelled *Kwalib*) is spoken by approximately 100,000 people in the Nuba Mountains, which are located in the Republic of Sudan (Quint, 2010a). It belongs to the Niger-Congo family, which is considered to be the largest language family in the world in terms of distinct languages. The Niger-Congo family has many branches and one of them is the Kordofanian branch, which is composed of the languages spoken in the Nuba Mountains. Kordofanian languages themselves are divided into several smaller groups, including the Heiban, Talodi, Rashad, Katla (Schadeberg et al., 1989) and Lafofa (Stevenson, 1956) languages. Koalib is part of the Heiban group, together with several similar languages (Schadeberg, 1981). Thus, Koalib can be classified as belonging to the Niger-Congo family, Kordofanian branch, Heiban group. Furthermore, this study focuses on one particular dialect - *ŋèrɛ́ɛɽɛ̀* (also spelled *Rere*), which is the basis of written modern Koalib and could be regarded as the standard dialect (Quint, 2010b).

Kordofanian languages are some of the least studied African languages, due to a variety of factors. One of them is the sheer diversity of Kordofanian languages and Niger-Congo languages in general. Even though they are spoken in a relatively small geographical area - the Nuba Mountains, and by a small number of people, Kordofanian languages include an impressive number of distinct languages, many of them with a variety of dialects. This complexity itself presents a challenge to the linguists who wish to account for the entire Kordofanian branch. Due to the relatively few speakers, Kordofanian languages have long been neglected by the scientific community and this has led to the scarcity of reliable data about these languages. In addition to that, the wars

and isolated location have not made the Nuba Mountains a popular destination for international researchers.

Nevertheless, there are some studies on those languages and Koalib in particular. Some of the most important earlier studies are those by Roland Stevenson from the 1950s, a protestant priest who was supposed to christianize the Nuba people but subsequently became quite interested in their language as well, and the studies by Thilo Schadeberg from the 1970s, who is a professional linguist. Some more recent studies have been made by Nicolas Quint, who has investigated the phonology and grammar of the Koalib language.

2.1.1 Tonality

Koalib, like most Niger-Congo languages, is tonal (Quint, 2010b). In linguistics and phonology, the term *tone* refers to a specific change in the pitch of the speaker's voice when pronouncing a word. It is similar to *intonation*, which is used by almost all languages to express attitude, emotion or to distinguish a statement from a question. However, intonation cannot change the lexical meaning or the grammatical form of the word. Tones, on the other hand, are used exactly for that purpose - to distinguish words and word forms. Thus, a speaker of a tonal language can differentiate one word (or word form) from another merely by a change in the pitch.

Koalib distinguishes two pitch levels - *low* and *high*. These two pitch levels represent the *register tones* in Koalib (Quint, 2010b). In addition to the register tones, which correspond to different pitch levels, there are also two *contour tones* - *falling* and *raising*, which correspond to a specific change in the pitch, rather than just a level (Quint, 2010b). A falling tone is a tone that starts as a high tone, then ends in a low tone. A raising tone is the reverse - first low, then high. Most Koalib words contain only register tones and contour ones are quite rare. Furthermore, the low tone is generally held as the default tone, while the high tone is used to mark syllables. In the written language, tones are indicated by special accent marks above vowels:

- low tone **(L)**: *àо* as in *kwàо* (grave accent),

- high tone **(H)**: *á* as in *kwár* (acute accent),
- falling tone **(F)**: *â* as in *kwâm* (circumflex),
- raising tone **(R)**: *ǒ* as in *kwǒrmàn* (caron).

Note that in the first word - *kwào*, there are two vowels and only one accent mark. This does not mean that the vowel *o* does not have a tone. On the contrary, it means that the low tone extends to the vowel *o*, which belongs to the same syllable as the vowel *a*. It is merely a writing convention to indicate the tone only for the first vowel of a syllable with more than one vowels. This way it is easier to see whether two adjacent vowels belong to the same syllable or to two different ones.

This writing convention also implies that tones apply to individual syllables, as opposed to whole words. However, this is not entirely correct. It appears that tones in Koalib are not strictly bound to their corresponding syllables. In certain contexts, a contour tone may expand to cover two syllables, breaking down to two different register tones in the process:

(1) kwɛ̂ɛcɛ́
'he will see'

(2) nyí kwɛ̀ɛcɛ́
'I will see'

In example 1, we see that the syllable *kwɛɛ* has a falling tone. But when it follows the word *nyi*, the syllable *kwɛɛ* receives a low tone and the preceding syllable gets a high one. Considering that a falling tone indicates the change from high to low pitch, we could consider it to be a combination of a high and a low tone. Thus, the tone melody[1] of the word *kwɛ̂ɛcɛ́* is essentially HLH. But since it has only two syllables, the first two tones are forced onto one syllable - they are "squeezed" into a falling tone. However, when it appears after the word *nyi*, the falling tone expands to cover the preceding syllable as well. While doing so, the falling tone is split into a high tone, which goes to the syllable *nyi*, and a low tone, which remains for the syllable *kwɛɛ*. But the underlying tone melody remains the same - HLH. Therefore, I will consider contour

[1]Here, I use the term *tone melody* to refer to the change of the pitch for a whole word.

tones to be merely a combination of two different register tones and assume that tones can only be either low or high.

2.1.2 Vowel Harmony

Many languages, including Koalib, exhibit *vowel harmony* (Quint, 2010b). Vowel harmony is a phonological phenomenon that constrains which vowels can appear in a word. In Koalib, vowel harmony manifests itself by grouping all vowels into two sets and forcing every word to contain vowels from only one of those sets (Quint, 2010b). These sets are defined by the vowel height - the vertical position of the tongue when pronouncing the vowel. Thus, there are high vowels, for which the tongue needs to be positioned high in the mouth and closer to the roof, and low vowels, for which the tongue is lowered and there is more space between it and the roof of the mouth:

- high vowels = { i, ɐ, u },
- low vowels = { e, ɛ, a, ɔ, o }.

Because of vowel harmony, we can assume that every Koalib word contains either only high vowels or only low ones. In addition to their height, we can also categorize the vowels according to their backness. Vowel backness refers to the horizontal position of the tongue - front vowels are pronounced by moving the tongue forwards and for back vowels we need to place it towards the back of the mouth. Using the notions of vowel height and backness, we can categorize the vowels in Koalib as follows:

Height	Backness		
	Front	Central	Back
High	**i**	**ɐ**	**u**
Low	**e**	**a**	**o**
	ɛ		**ɔ**

Note that the low set contains two front vowels - *e* and *ɛ*, as well as two back ones - *o* and *ɔ*. Although all of these vowels belong to the same vowel harmony set, there are some subtle differences that separate the vowels *ɛ* and *ɔ* from the rest of the low set. First of all, they belong to a different category, according to vowel openness:

- open vowels = { ɛ, ɔ },
- mid vowels = { e, a, o }.

Secondly, almost all Koalib words that belong to the low vowel harmony set contain either only open vowels or only mid ones (Quint, 2010b). Thus, it appears that the open vowels *ɛ* and *ɔ* almost form their own vowel harmony set. So why group them with the mid vowels? The reason is that almost all words in the low set receive affixes with only mid vowels, regardless of whether they have mid or open vowels. Hence, the distinction between open and mid vowels is irrelevant from a morphological point of view. In contrast, the distinction between the low and the high vowel harmony set plays a crucial role in determining the vowels of the affix. Since this study is concerned with predicting the suffixes of nouns, it seems more logical to assume only two vowel harmony sets - low and high.

2.1.3 Noun Classes

Many Indo-European languages have noun classes which are used to express gender. Koalib also has a noun class system, but it is used to express different classes and not just gender. These noun classes are expressed with a prefix, so that Koalib nouns can be thought of as a combination of a nominal root and a class prefix:

(3) **kw**òoɽàm ⇒ **kw**òoɽámè
'thief'

(4) **l**òoɽàm ⇒ **l**òoɽámè
'thieves'

(5) **ŋ**òoɽàm ⇒ **ŋ**òoɽámè
'theft'

In these examples, the nominal root *-òoɽàm* combines with three different prefixes - *kw-*, *l-* and *ŋ-*, each giving the noun a different class. The forms for the object case, which is discussed in the next paragraph, are shown right of the arrow. As you can see, the noun class prefix does not affect the object form in any way - the noun is inflected in the same way, regardless of its class. This pattern is observed in almost all known Koalib nouns (Quint, unp).

2.1.4 Case

Another important characteristic of the Koalib language that is relevant to this study is its case declension. Koalib nouns have two cases - subject (S) and object (O) (Quint, 2006). Moreover, around 75% of the nouns are inflected for case (Quint, unp). The subject case is assumed to be the unmarked case, as there is usually no overt marker to indicate that a noun is in the subject case, while the object case is considered to be the marked case and is indicated by the addition of a suffix, a change in the tonal pattern of the noun, or a combination of both (Quint, 2006):

(6) **káaŋàl** ŋkó kè-pèetò
sheep.S DEM.CLF.PROX CLF-be.white.PFV
'This sheep is white.'

(7) Kwókkò kwèm-èɛcɛ́ **kàaŋálè** ŋkó
Kwókkò CLF.PRF-see **sheep.O** DEM.CLF.PROX
'Kwókkò has seen this sheep.'

In example 3, the word *káaŋàl* is the subject of the sentence, therefore it is in subject case. In example 4, the same word is in object case and this is indicated by a change in the form - *káaŋàl* becomes *kàaŋálè*. The object case is indicated by two overt markers - the word receives the suffix *-è* and its tonal pattern changes from HL to LH, plus an additional L tone for the suffix. This is an example of an object case form, marked by both suffix and tone change. However, other object forms are marked by only a suffix or only a tone change, and some are not marked for object case at all. Thus, in total there are four general ways to create the object form:

- **no change** (25%): ŋèráaɽà (sauce.S) ⇒ ŋèráaɽà (sauce.O),
- **suffix only** (35%): kòʈʈó (gourd.S) ⇒ kòʈʈóŋé (gourd.O),
- **tone only** (6%): kwìcì (person.S) ⇒ kwícì (person.O),
- **both** (34%): kwòtlòm (jackal.S) ⇒ kwótlòmá (jackal.O).

The percentages indicate the proportion of nouns that use this type of case marking and are estimated from a corpus of approximately 1,200 Koalib nouns, collected by Nicolas Quint Quint (unp).

Note that *tone change* here does not include the tone change introduced by a suffix, which usually adds another tone. By manually analyzing this corpus, Quint (unp) was able to discover some patterns, regarding case inflection:

Subject pattern		Object pattern		Reliability
Tones	Phonemes	Tones	Phonemes	
-LH	any	-LH**H**	any	252 / 267 = 94%
	-C		-**e**	125 / 136 = 92%
	-V		-**ŋe**	113 / 131 = 86%
$(H)_n$	any	$(L)_n$**H**	-**a**	117 / 167 = 70%
	-C		-**a**	108 / 124 = 87%
	-V	no change		23 / 43 = 53%
LL	any	no pattern		N/A
	-VC.CVC	HL**H**	-**a**	36 / 44 = 82%

Though these patterns are decently accurate, they only cover a small part of the corpus - around 40%. The present study uses this very same corpus and discovers rules that cover all of the nouns in it by employing machine learning methods.

2.2 Machine Learning

The goal of this study is to demonstrate how machine learning methods can be successfully used to automatically discover the rules behind case inflection in Koalib. Machine learning is a branch of artificial intelligence and is primarily concerned with creating systems that are able to learn from data. The data consists of examples (also called *instances*), which are represented as values of specific properties (also called *features* or *attributes*) which capture relevant information about the data. The choice of properties depends on what is available from the data and the problem that has to be solved by the system.

There are two basic types of machine learning - *supervised learning* and *unsupervised learning*. In supervised learning, the task is to assign categories (also called *labels*) to instances. To do this, the system needs training data - a set of labeled instances to learn from. The process of learning consists of building a formal representation (called a *model*) which generalizes the training data, capturing any patterns that involve the specified features. The

system can then use this model to label new unseen instances. How the model is built and how it is used to label new data, depends on the particular supervised learning algorithm.

Supervised learning algorithms can be of several types, depending on the nature of the task. The most common type of supervised learning is *classification*, where the goal is to simply assign the correct label to individual instances. In contrast, there is also *sequence labeling*, which is a different type of supervised learning. In sequence labeling, we assume that some instances belong together to form sequences of instances, where each label may depend on the other labels in the sequence.

An example of a classification task is diagnosing patients. The training data for this task could be a set of relevant observations about previous patients, such as their age, gender, blood pressure, presence or absence of specific symptoms and their correct diagnoses. Here, the properties *age*, *gender*, *blood pressure* and *symptoms* are the features that describe each instance, while the *diagnosis* is the class - what we want to predict.

Features can have different types of values. For example, the features *gender* and *symptoms* have nominal values (male or female; present or absent for each specified symptom), while *age* and *blood pressure* could have numeric values (the measure in years; the measure in millimetres of mercury) or ordinal values if we define some intervals (child: 0-18 years, adult: 19-49 years, elder: 50+ years; low: 0-89 mmHg, normal: 90-119 mmHg, high: 120+ mmHg). Classes typically have nominal values - the set of possible diagnoses, in this case. The values of the class are also referred to as labels.

In order to build a classifier that can predict the diagnoses of new patients, first we need to describe our observations about previous patients with the specified features by assigning the correct values for each patient. This way, each instance is composed of feature values, which are usually stored in a vector (the *feature vector*). With this training data, we can use a classification algorithm to build a classifier, which would capture any patterns that involve the specified features. Thus, it is important to choose the right features, in order to have accurate predictions. In general,

the more training data we have, the more reliable the classifier, though it is also important that this data is representative of the population.

A typical example of a sequence labeling task is part-of-speech tagging - predicting the parts-of-speech of words in texts. As with classification, the training data consists of observations - texts which are annotated with the correct part-of-speech tags. In this case, the labels are the possible part-of-speech tags and the features are the properties of words that are relevant for determining the part-of-speech in the language. Each sentence is a sequence with each instance being a word from the sentence. Although it is possible to use a classifier and treat words individually, it would be more appropriate to use a sequence labeler instead, because the labels may depend on each other - for example, the part-of-speech of a word often depends on the part-of-speech of the previous word in the sentence.

Aside from supervised, there is also unsupervised learning, where the task is to find some hidden structure in the data. The main type of unsupervised learning is clustering and the goal of clustering is to group similar instances together, based on the specified features. One way to do this is to represent instances as points in an abstract multi-dimensional space, where each dimension corresponds to a feature. The feature values for each instance determine its position in the space, so that similar instances would be placed closer to each other, clumping together and forming clusters (hence *clustering*). The system can then identify the clusters by measuring the distances between the points in the space.

2.2.1 Feature Selection

As mentioned, selecting the right features is an important factor in machine learning. Though we cannot always know for sure what features are the best, we can experiment with the ones that are likely to be useful and then check which ones actually help with the prediction. This process of feature selection is particularly important for tasks, where there are a lot of potential features but only a small fraction of them are actually useful, such as text classification, where each word could be a feature.

One way to measure which features are most useful is by using an algorithm called *InfoGain*. InfoGain works by measuring the information gain of each feature with respect to the class and then ranking the features from most useful to least useful. The information gain of a feature can be measured by calculating the reduction of entropy when this feature is introduced.

Entropy is a measure of uncertainty. The more the possible outcomes of an event, the greater the uncertainty, thus the greater the entropy of that event. Entropy can be measured in bits - one bit denotes two possible outcomes (0 or 1, true or false). For example, tossing a fair coin has two possible outcomes - heads or tails. Therefore, the entropy of the event is one bit. Tossing two fair coins has an entropy of two bits, as each coin has two possible outcomes, for a total of four possible outcomes. However, tossing an unfair coin has a smaller entropy than tossing a fair one. This is because the fair coin is equally likely to land on either side, thus we are more uncertain about the outcome. But an unfair coin is more likely to land on one side than the other, thus we are less uncertain about the outcome.

The entropy of a set of training examples with respect to the class can be calculated using the logarithmic probability of each class label. The probability of a class label is calculated by dividing the number of instances with that class label by the total number of instances. More formally, if E denotes the set of examples (instances), C_i is the set of examples with the i-th class label (i ranges over the possible class labels), then the probability of the i-th class label is this:

$$P_i = \frac{|C_i|}{|E|}$$

The entropy of the data set with respect to the class is calculated by summing up the logarithmic probabilities of all the class labels, where b is the base of the logarithm, which is usually 2:

$$H(C) = -\sum_i P_i . \log_b P_i$$

The information gain of an attribute (feature) A with respect to the class C is calculated by measuring the reduction of entropy in the data set when A is introduced:

$$InfoGain(C, A) = H(C) - H(C|A)$$

The entropy with respect to the class when an attribute is introduced, denoted as $H(C|A)$ is calculated by summing up the products of the probability of each attribute value times the entropy with respect to the class of the instances with that attribute value:

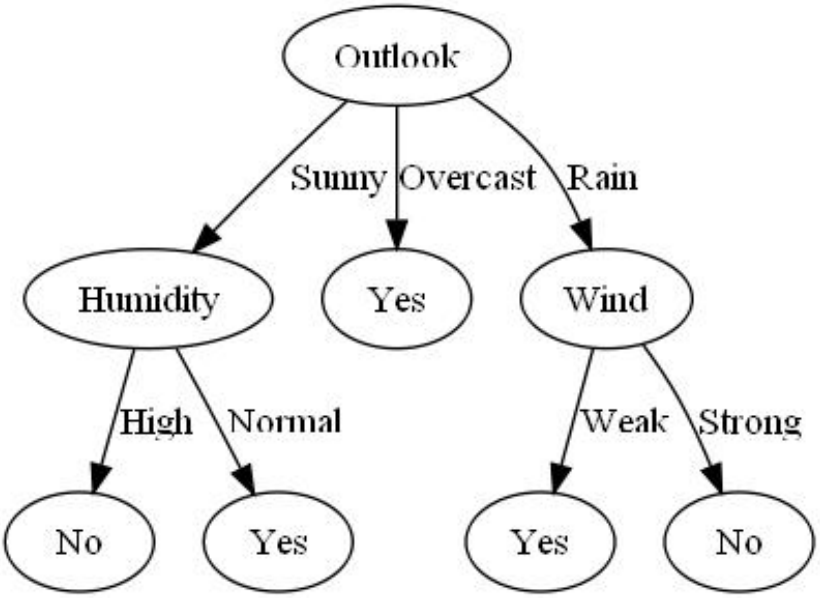
$$H(C|A) = \sum_j P_j.H(C|A_j)$$

In this formula, P_j is the probability of the j-th value of A (j ranges over the possible values of A), which is calculated similarly to the probability of a class label - by dividing the number of instances with the j-th value for A by the total number of instances. Here, $H(C|A_j)$ denotes the entropy with respect to the class, but only for the instances in the set A_j - that is, the instances with the j-th value for A. It is calculated just like $H(C)$, except that the set of instances E is limited to A_j, so $E = A_j$.

Thus, InfoGain measures how the entropy of the data set is reduced when a certain attribute is introduced. That is, how much relevant information does the attribute bring. It is a useful measure for feature selection, but also for building decision trees.

2.2.2 Decision Trees

Decision trees are tree-like graphs that visualize a model for making decisions. Each node in the tree represents a test and each edge from that node stands for a possible outcome of that test. The leaf nodes indicate the decisions that are made after following the tree from the root to a leaf. A simple tree for making the decision whether to go out or not, depending on the weather, could look like this:



In order to make a decision, we need to start from the root node and perform each test, then follow the edge that corresponds to the observed outcome. So, if the weather today is sunny with normal humidity, then we make the decision to go out. But if it rains and the wind is strong, we stay at home.

Decision trees are widely used as classifiers in machine learning. The training data is used to learn a tree, then the tree is applied to classify instances of data. There are different algorithms for learning decision trees. Some notable ones are ID3 and C4.5.

The *Iterative Dichotomiser 3* (ID3) uses the information gain of features to learn a tree (Quinlan, 1986). It works by measuring the information gain of each attribute and then choosing the best one - the attribute with the highest information gain. Then, it splits the data set into subsets - one subset for each value of that attribute. For each one of those subsets, the process is repeated recursively, until every subset belongs to only one class label or there are no more attributes to use:

- create the root node of the tree,
- if instances in data set belong to more than one class label, choose best attribute (feature),

- partition the data set into subsets - one for each value of the attribute,
- for each subset, repeat the process with another attribute.

After inventing ID3, Quinlan extended the algorithm and created an improved tree learner, called C4.5 (Quinlan, 1993). C4.5 works similarly to ID3 and also uses the information gain measure to evaluate attributes. It introduces several improvements, one of which is the pruning of the tree after creation. Pruning is a technique which is used to simplify a decision tree by removing nodes that are not very useful in classifying instances. The resulting tree is much smaller in size and also much more accurate at predicting new data.

Pruning is important, because it allows the decision tree to deal with the problem of "overfitting". Overfitting occurs when the features used for training are too many or too specific. This causes the trained model to be very specific to the data it was trained on and capture not only the general pattern in the data, but also the "noise" in it. Noise is a term used to refer to rare, exceptional instances which do not fit the general pattern of the data. Such exceptions usually occur due to random errors in the data and should be ignored by the model. But if the model is too complex and specific, it will also account for such exceptions. The problem is that this makes the model less predictive, because new instances are unlikely to be random errors, and even if they are, it's even more unlikely that they are the same random errors as those in the training data.

Thus, overfitting reduces the prediction accuracy of a model for unseen data by making it unnecessarily complex. However, decision trees can be "pruned" by removing the nodes that help little to increase the prediction accuracy. There are several pruning techniques. The simplest one is *reduced error pruning*, where each node, starting from the leaves, is replaced by its most probable class label. Then the prediction accuracy is checked and if it is not affected much, the change is kept.

2.2.3 Rule Learning

Decision trees encode a model for making decisions by showing what decision to make when certain tests are fulfilled. Such a model can also be represented as a list of rules, where each rule says what decision to make when certain conditions are met. The technique for building such rule lists is called rule learning. Rule learning in fact is similar to learning decision trees and the algorithms for that are derived from the earlier decision tree algorithms.

One of the earlier algorithms for learning rules was also invented by Quinlan (1987), who created the decision tree algorithms discussed before. This algorithm basically builds a decision tree and then extracts rules from that tree. First, it grows a very complex tree that overfits the data, then it prunes this tree and represents it as a list of production rules.

Many other rule learning algorithms also make a heavy use of pruning. They usually work by dividing the training data into a growing set and a pruning set. The growing set is used to build a very specific rule list that overfits this data, then this rule list is pruned using the pruning set. The pruning technique used by the earlier rule learners was reduced error pruning. However, later algorithms improved that technique and used *incremental reduced error pruning* (Furnkranz and Widmer, 1994).

Though incremental reduced error pruning (IREP) is very efficient, the rule lists that it produces are shown to have higher error rates than C4.5 rules (Cohen, 1995). Thus, Cohen (1995) proposed a modified version of IREP for rule learning, called RIPPER, which is competitive with C4.5 in terms of error rates, but also much more efficient for large data samples.

Both decision trees and rule lists have the advantage of being easy to interpret by human beings. This is a very important advantage for this study, because the ultimate goal is to discover the rules that govern case inflection in Koalib and not simply to build a system that predicts the object form. Therefore, I consider both the decision tree learner C4.5 and the rule learner RIPPER to be valid choices for the supervised machine learning algorithm

that will discover the rules. However, the rule learner RIPPER has a slight advantage over C4.5, because the model it produces is even more transparent than a C4.5 decision tree. Using a decision tree would require some additional postprocessing to collect the rules from it. Though this is not very difficult to do, it makes more sense to use an algorithm that already does it - RIPPER.

3 Methods

This study explores the use of machine learning methods for discovering the rules behind case inflection in Koalib. It exploits some of the information available from the corpus of Koalib nouns, collected by Nicolas Quint. This information includes the subject form and its tonal pattern, the root morpheme of the subject form and its tonal pattern, the object form, its suffix if there is any and its tonal pattern, too. The corpus contains some additional information, such as the types of phonemes (consonants or vowels) for the subject form, its root morpheme and the object form, as well as French translations for most of the nouns. However, this information is not exploited, because it is either automatically deduced during feature extraction or is not needed for the task.

The data extracted from the corpus is used for the training and evaluation of supervised machine learning models. The main idea is to represent the task as a classification problem, where the goal is to predict the object forms of nouns, given their subject forms. Then, experiment with various features and classes, until a good prediction accuracy is achieved. Finally, extract the rules from the model with the best prediction accuracy.

Since the ultimate goal is to extract human-readable rules, it is important to use a classifier, whose model is easy for human beings to interpret. The accuracy of the model is of a secondary concern - it is needed to select the optimal features and provide an estimate to the reliability of the rules. The performance of the model, in terms of time and space it takes to train or classify, is not an issue here, as the data set is quite small. Therefore, I consider that both decision trees and rule lists are good choices for a model, but I prefer to use rule lists, since they provide exactly the information I need.

3.1 Object-Form Prediction

During the project, I have experimented with various methods for predicting the object form and extracting the features and classes. This paper presents the results of the methods that I consider to be the optimal solutions, both in terms of prediction

accuracy and relevance to linguistic theory, and discusses some alternative solutions that yield similar accuracy.

The problem of predicting the object form can be reduced to predicting only the change from subject form to object form, because the subject form is given as input. Recall that object forms in Koalib are marked either by the addition of a suffix, a change in the tonal pattern or both. This means that object forms may differ from subject forms in two aspects - suffix and tonal pattern. If we can predict both of these changes correctly, then we can deduce the correct object form from the subject form.

Thus, one strategy for predicting the object form is to break down the whole problem into several smaller ones and predict each type of change individually. This means that the object form is predicted using two classifiers - one of them predicts the suffix and the other one predicts the tone change. The advantage of this strategy is that we can separate the rules that determine the suffix from those that determine the tone change. This is useful if we are only interested in one of those changes. From a technical point of view, another advantage is that the features and parameters of each classifier can be adjusted separately. However, with this strategy we are making the assumption that the suffix change is independent of the tone change, which may not be true.

A second strategy is to predict both the suffix and tone change at the same time. This way, the problem is solved with only one classifier, which predicts the combination of suffix and tone change. The advantage of this approach is that any dependencies between suffix and tone change are captured by the model. This study focuses on using these two strategies for predicting the object form.

There is also an alternative, third strategy, which will be discussed. It combines the advantages of the previous two to some extent, but makes the whole design slightly more complex. The idea is to use multiple classifiers - one for each type of change, similarly to the first strategy, and to add the predictions of the first classifier as additional information for the next one. This way, the classifiers run in a pipeline fashion - the output of each classifier

is input for the next one. With this approach, the features and parameters of each classifier can still be adjusted separately, while also capturing dependencies between the two types of change.

3.2 Classes

The classes encode the change from subject to object form or a particular aspect of that change. They are defined in such a way that knowing the correct class label makes it possible to deduce the correct object form from the subject form. The first two classes - suffix and tone change, are needed for the multiple-classifier strategy, while the third one - full change, is a combination of the first two and is used for the single-classifier strategy.

3.2.1 Suffix

One of the changes that subject forms undergo during inflection is the addition of a suffix. 69% of the nouns in the corpus receive a suffix in their object form. Therefore, one of the classes for object-form prediction is the suffix class. Each possible suffix in Koalib is encoded as a label of this class and there is an additional label for nouns that do not receive any suffix in the object form.

An important note about suffixes is that the vowels in them are not encoded directly, but are instead represented by abstract vowel categories. These vowel categories correspond to the phonological categories, specified by vowel backness. This means that each vowel falls into one of three categories: front, central or back. Each category is represented by a letter:

- front **(E)** = { i, e, ɛ },
- central **(A)** = { ɐ, a },
- back **(O)** = { u, o, ɔ }.

The main reason for this categorization is vowel harmony. In Koalib, the choice between, say an *i* or an *e* in an affix[2], depends solely on vowel harmony constraints (Quint, 2010b). That is, if

[2] I use the term *affix* here, because these statements apply not only to suffixes, but also to prefixes.

the other vowels in the word belong to the high VH set, then so does the vowel in the affix and we choose *i*. Otherwise, if they are in the low VH set, then we choose *e* for the affix. Therefore, it suffices to know if the vowel in an affix is front, central or back, as the differences between the vowels in each one of those sets are dependent on the context of the affix in the word and should not be regarded as a property of the affix itself. This also has the side effect of reducing the number of possible class labels for the suffix.

Note that the low VH set contains two front vowels (*e* and *ɛ*) and two back vowels (*o* and *ɔ*). As discussed earlier, the difference between the mid vowels (*e* and *o*) and the open ones (*ɛ* and *ɔ*) is irrelevant to morphology, as affixes almost always contain only mid vowels for the low VH set, regardless of whether the word has mid or open vowels.

With this definition, the class *suffix* has 8 possible labels. 4 of them are very common (frequency > 100), while the other 4 are quite rare (frequency < 10). Here are the 4 most common labels for *suffix* (the numbers indicate the frequencies with the total number of instances being 1206):

- -A (488)
- NONE (374)
- -E (211)
- -ŋE (122)

On a side note, some suffixes replace the last letter of the subject form when added, while others double the length of the first vowel in it. Thus, one could additionally try to predict whether a suffix replaces the last letter, prolongs the first vowel or is simply added without any other changes. However, I consider this information to be insignificant, as almost all the suffixes in the corpus are added without any other changes. In the whole corpus of 1206 nouns, there are only 12 instances of suffixes which replace the last letter of the subject form and only 18 instances of suffixes which double the length of the first vowel.

3.2.2 Tone Change

The other possible change that can occur when a noun is inflected is a change in its tonal pattern. Recall that Koalib is a tonal language with two register tones (*low* and *high*) and two contour tones (*falling* and *raising*). For the purposes of this study, I will treat each contour tone as a combination of two different register tones and assume that there are only two tones - *low* and *high*. I use the term *tone pattern* to refer to the sequence of tones in a word, where each tone is represented by a letter:

- L = low tone,
- H = high tone.

The class *tone change* encodes the difference between the tone pattern of the subject form and the tone pattern of the object form. 75% of the nouns in the corpus show a change in their tone pattern when inflected. Once again, there is a special label for cases where there is no change. In other cases, one or two tones are added to the tone pattern of the subject form without changing any of the original tones (probably caused by the addition of a suffix). In these cases, there is a label for each sequence of added tones. All other tone changes are encoded as a pair of a subject tone pattern and a corresponding object tone pattern, with a label for each possible pair. This way, the class *tone change* has 49 possible labels[3]. 14 of them are fairly common (frequency > 10), while the other 35 are rarer (frequency < 10). These are the 14 most frequent labels:

- ADD-H (372)
- NO-CHANGE (306)
- HH-LLH (101)
- HL-LHL (68)
- LL-HLH (42)

[3]There are actually several versions of the data set - in two of them the whole subject and object forms are taken into account, while in the other two prefixes are ignored. These are discussed in the section about experiments. The information reported here is derived from a data set that considers the whole subject and object forms.

- LHL-LLHL (40)
- L3H-LHH (38)[4]
- LLL-HLLH (31)
- HL-LL (29)
- LL-HL (24)
- LL-LHL (23)
- HL-LLH (21)
- ADD-L (16)
- LHL-LLL (14)

During the course of the project, I have experimented with many alternative ways to define the class *tone change*. Initially, the class labels were simply the tone patterns of the object form. Then, in order to reduce the number of possible labels, those tone patterns were shortened by reducing a sequence of the same tone, repeated more than once, to only one tone. This was limited to only one sequence per tone pattern, in order to be able to deduce the entire object tone pattern afterwards.

But there are many cases where there is no change in the tone pattern. Such cases were later grouped into the same label, as in the current definition of the *tone change* class. Furthermore, assuming that each tone is strictly related to a particular syllable in the word, added tones were predicted separately as a property of the suffix and were not taken into account in the *tone change* class. Then, in order to express the change from subject tone pattern to object tone pattern, rather than the result thereof, the cases where an actual change does occur were explicitly listed as pairs. Some of these pairs were grouped together to avoid having too many class labels. Initially, the grouping was done by assuming that a tone changes by "inverting" into the opposite tone (H inverts to L and L inverts to H) and the class labels encoded which

[4]In Koalib, there is a group of disyllabic nouns that have two possible tone patterns - LL and LH. These are treated as special cases and are assigned a unique tone pattern - L3H.

tones are inverted. Then, in order to have a more "natural" grouping, the data was clustered, according to the features that best predict the explicit tone-change pairs, and the class labels corresponded to the clusters, but without having more than one pair with the same subject tone pattern in a cluster.

Unfortunately, this cluster-based grouping did not lead to a significant improvement in the prediction accuracy. Therefore, that idea was dropped in favor of a simpler design with more intuitive rules as a result. Finally, the *tone change* class was redefined to include added tones as well, realizing that tones are not strictly related to syllables.

3.2.3 Full Change

The previous two classes - *suffix* and *tone change*, describe the two aspects of the change from subject to object form separately and are used for the multiple-classifier strategy. The alternative strategy - the single classifier, is to predict the combination of suffix and tone change in one go. The class for this classifier is called *full change* and simply combines the labels of the previous two classes, as defined above. This results in 76 possible class labels. 20 of those labels are common (frequency ≥ 10) and the remaining 56 are rare (frequency < 10). Here are the 20 most common labels:

- NO-CHANGE + NONE (297)
- ADD-H + A (161)
- ADD-H + E (110)
- HH-LLH + A (101)
- ADD-H + ŋE (98)
- HL-LHL + A (55)
- LL-HLH + A (42)
- LLL-HLLH + A (31)
- LHL-LLHL + A (24)

- L3H-LHH + ŋE (21)
- LL-HL + NONE (21)
- HL-LL + NONE (20)
- HL-LLH + A (18)
- L3H-LHH + E (17)
- LHL-LLHL + E (16)
- ADD-L + E (15)
- LHL-LLL + NONE (14)
- HL-LHL + E (13)
- LL-LHL + E (11)
- LL-LHL + A (10)

Though 76 class labels may seem like a lot, most of them are very rare and constitute only a tiny portion of the data. There are 31 unique labels (frequency = 1), but they are only 3% of the data. Such rare labels are likely to be nothing more than noise in the data and can safely be ignored by the model, using techniques like pruning.

3.3 Features

The features encode the properties of the nouns that are used for predicting the object-form change. They are probably the most important factor for the success of the machine learning model. Ideally, the features should capture only those properties of the data that are relevant to solving the task. They should be specific enough to separate most of the class labels, but not too specific to account for noise as well. Thus, finding the optimal selection of features is not always easy, as it is not immediately obvious what the most important properties are. This is why it is important to have a good understanding of the problem which has to be solved by the model.

In order to find the optimal selection, I have experimented with

various features and only kept the ones that help the prediction. The rules discovered by Quint (unp) provide a good starting point and I have also examined the phonology of Koalib, in order to develop a better understanding of the linguistic phenomena that take place. Finally, I have tried to define features that are not only accurate, but also relate to linguistic theory.

3.3.1 Phonemic Features

The first group of features encode information about the phonemes of the subject form. Though Koalib is a written language, I consider that phonemes are the basic building blocks of words, rather than letters, because it appears that the phenomena involved in Koalib case inflection are of a phonological nature. Similarly to suffixes, phonemes are not taken directly, but are replaced by abstract categories. The categories for vowel phonemes are the same as the ones for vowels in suffixes - front (E), central (A) and back (O), while consonants are categorized by manner of articulation:

- plain obstruent **(B)** = { b, ɗ },
- strong obstruent **(T)** = { pp, tt, ʈʈ, cc, kk, kkw },
- weak obstruent **(P)** = { p, t, ʈ, c, k, kw },
- prenasalized obstruent **(M)** = { mp, nt, nʈ, ny, ŋk, ŋkw },
- nasal non-obstruent **(N)** = { m, n, ŋ, ŋw },
- liquid non-obstruent **(L)** = { l, r, ɽ, j, w, y }.

This categorization is based on Quint (2010b) with some minor adjustments to the available data. Apart from the category, phonemic features also encode the type of the phoneme - consonant (C) or vowel (V), for cases where the category feature is too fine-grained. This way, I have defined four phonemic features:

- last phoneme
- last two phonemes
- last phoneme type

- last two phoneme types

All four features describe the ending of the subject form, because this is likely to play a role in the choice of the suffix.

3.3.2 Syllabic Features

While phonemes are the basic building blocks of words, they can combine to form larger phonological units - syllables. The next group of features describe the subject forms at the syllable level. Syllabic features encode information about the structure of the syllables in the noun.

In general, syllables are made up of three components - nucleus, onset and coda. The nucleus is a vowel and is usually the only obligatory component of a syllable. The onset consists of one or more consonants that precede the nucleus, while the coda comprises the consonants that follow the nucleus. The nucleus and coda combine to form the rhyme of the syllable.

Koalib syllables follow this general structure, with the only difference that the nucleus can be either one or two vowels. The structure of Koalib syllables is encoded by grouping them into 8 possible types, depending on whether the nucleus is one or two vowels and whether there is an onset and coda:

- **V** = single-vowel nucleus,
- **CV** = onset + single-vowel nucleus,
- **VC** = single-vowel nucleus + coda,
- **CVC** = onset + single-vowel nucleus + coda,
- **VV** = two-vowel nucleus,
- **CVV** = onset + two-vowel nucleus,
- **VVC** = two-vowel nucleus + coda,
- **CVVC** = onset + two-vowel nucleus + coda.

Note that although the system theoretically allows for the types VVC and CVVC, they are practically non-existant. I have only encountered one instance of a CVVC syllable in the entire corpus and not a single VVC syllable. Thus, from a practical point of view, we can assume that the last two types do not exist and there are only six syllable types in Koalib, as noted by Quint (2010b). The syllabic features that are used are the following:

- last syllable
- last syllable type
- last two syllable types
- first two syllable types

The feature *last syllable* consists of the phonemes (or more precisely, the phoneme categories) of the last syllable. For monosyllabic nouns, the features *last syllable type*, *last two syllable types* and *first two syllable types* are identical, while for disyllabic nouns, only the features *last two syllable types* and *first two syllable types* are identical. Most of the nouns in Koalib have two syllables. The noun with the most syllables in the subject form has 5 syllables. The mean number of syllables is 2.329 with a standard deviation of 0.629.

3.3.3 Tonal Features

The last group of features describe the tone pattern of the subject form. As with the *tone change* class, only the register tones - low (L) and high (H) are considered, and nouns with two possible tone patterns (LL and LH) are treated as a special case and are assigned the tone pattern L3H. The following tonal features are defined:

- last tone
- last two tones
- squeezed tone pattern
- complete tone pattern

The feature *squeezed tone pattern* is a shortened version of the *complete tone pattern*. It is obtained by reducing all sequences of the same tone, repeated more than once, to only a single instance of that tone. This way, the tone pattern is "squeezed", so that the resulting pattern never contains two identical adjacent tones. This "squeezed" tone pattern encodes the basic underlying tone melody of the subject form. In some cases, the basic tone melody is more important than the complete tone pattern, as tones are not strictly bound to syllables and sometimes there are variations in the tone pattern, while preserving the same basic melody.

4 Experiments

In order to test the methods proposed in the previous section and discover the actual rules, I conducted a number of experiments. This section describes the most important of those experiments, reports on their results and discusses their findings.

The data that was used for the experiments was extracted from the corpus of Koalib nouns, collected by Nicolas Quint. This corpus includes approximately 1,200 nouns. It does not contain any borrowings from English or Arabic, in order to focus only on traditional Koalib words, as these are more representative of the language and its grammar. Borrowings seem to follow different rules when it comes to case inflection and these rules are discussed in detail by Quint (unp). Furthermore, the corpus contains only nouns with distinct nominal roots - that is, it does not contain two nouns with the same root but different prefixes.

The corpus includes the subject forms, their root morphemes and the object forms of the nouns. For each one of those, it also provides the tone pattern and the phoneme types (vowel or consonant). It also contains the suffix of the object form, whenever there is any, and a translation of the noun into French. For some nouns, there is additional information, such as the plural form and a semantic category, but this information is too scarce to be used effectively.

4.1 Implementation

In order to conduct the experiments, I implemented a toolkit for manipulating the corpus, extracting classes and features for machine learning and evaluating some of the proposed methods. This toolkit is written in Java and uses Weka (Witten et al., 2011) for implementations of various machine learning algorithms. Weka is a popular open-source library for machine learning and is also written in Java, which makes it easy to integrate into my program. It also has a graphical user interface, but some of the experiments required a finer control, which the GUI cannot provide, therefore I also used some of its implementations directly from within my program.

4.1.1 Preprocessing

Before extracting the data for machine learning, it was necessary to transform the corpus into a more usable format and conduct some preprocessing steps to ensure that it is consistent. The corpus was originally in the format of an Excel spreadsheet. Though this format may be useful for viewing, it is not the most suitable format for automatic processing. Therefore, the first step was to transform the spreadsheet into a simple text format, where columns are delimited by tabs. To do this, I used another open-source library, written in Java - Apache POI, which enabled me to parse the Excel spreadsheet and export it in the format of my choice.

After transforming the corpus into a simple text format, some additional preprocessing was done to make it even more usable. The French tone labels were replaced by English ones, in order to avoid confusion and to comply with the notations used in Quint's paper on case marking. The data fields were checked for consistency and some small errors were fixed, such as incorrect or missing information. Additionally, some noun forms were normalized by making all letters lowercase and removing some extra notations that were not part of the actual form. The resulting corpus has exactly 1,206 nouns.

4.1.2 Syllabification

After preprocessing the corpus, the next step was to extract the data for machine learning. For each noun, the correct feature values and class labels had to be collected and stored in as a data set. The data set was exported in the Attribute Relation File Format (ARFF) which is used by Weka.

Most of the feature values and class labels were either directly available from the corpus or could easily be deduced automatically. Special care had to be taken while reading the letters of the words, as some of these letters are internally encoded by more than one character. The only features that required more sophisticated methods of extraction were the syllabic features, as the corpus contains no information about syllable boundaries.

In order to extract the values of syllabic features, I devised and implemented an algorithm for splitting words into syllables. The current implementation is tuned specifically for Koalib words, though the algorithm could probably be used for some other languages, for example Bulgarian, by adjusting its parameters accordingly. The idea for the algorithm came after trying to manually split some of the Koalib nouns into syllables and noticing that they follow a similar pattern. After careful consultations with my supervisor and expert on Koalib, Nicolas Quint, I defined the *syllable splitter* algorithm, making sure that it correctly splits all Koalib words into syllables.

The algorithm reduces the problem of splitting a word into syllables to finding the start index of each syllable. After knowing where each syllable starts from, it becomes trivial to split the word into syllables, as each start index represents a syllable boundary. Alternatively, it may also be possible to look for the end index of each syllable, but it seems easier to use start indices instead.

Thus, the essence of the algorithm is in the method for returning the start indices of the syllables in a word. As discussed before, a syllable is composed of three parts - onset, nucleus and coda. The onset and coda consist of one or more consonants and are optional. The nucleus in Koalib is either one or two vowels and is obligatory. Thus, a syllable starts either with an onset or with a nucleus.

Therefore, the problem of finding the start indices of syllables is solved by identifying the onsets and nuclei in the word. Recall that in written Koalib, accent marks are put on the first vowel of each syllable. This writing convention makes it easy to detect syllable nuclei - a valid nucleus consists of a vowel with an accent mark, which is optionally followed by a vowel without an accent mark, in case of a two-vowel nucleus.

But if there is a valid onset before the nucleus, then the syllable starts from that onset. Syllable onsets in Koalib can be divided into two types, which I have labeled *short onsets* and *long onsets*. Short onsets are composed of one or two consonant letters which behave as a single sound - they form a phoneme. Short onsets can never be split by syllable boundaries - if they contain

more than one letter, then these letters always go together.

On the other hand, long onsets are special combinations of two or more consonants that only go together under particular circumstances. Each long onset consists of a consonant letter, followed by a valid short onset. If the long onset is preceded by a consonant, then it behaves as a single unit and the syllable starts from the start index of that long onset. If it is preceded by a vowel, then it is split by a syllable boundary and the syllable starts from the start index of the short onset, which is within the long onset. Thus, when determining the start index of each syllable, the following conditions are checked for each valid nucleus (in this order):

- Is the nucleus preceded by a valid long onset?
- If yes, is the long onset preceded by a consonant or nothing[5]?
  - If yes, then the index of the syllable is the index of the long onset.
- If not, is the nucleus preceded by a valid short onset?
  - If yes, then the index of the syllable is the index of the short onset.
- If not, then the index of the syllable is the index of the nucleus.

Therefore, we need to know what are the valid nuclei, short onsets and long onsets before we can split a word into syllables. These are the parameters for the method and depend on the language for which it is used. In the case of Koalib, the syllable nuclei are either single vowels with an accent mark or combinations of two vowels - the first one with an accent mark and the second one without. But since we are actually only interested in finding the start of each nucleus, it suffices to know what the valid first vowels for each nucleus are. Thus, the set of valid nuclei contains all the accented vowels in Koalib. The set of valid long onsets contains all the combinations of a consonant and a short onset that behave as a single unit only under particular circumstances.

[5] In case the word starts with that long onset.

In the case of Koalib, this includes the prenasalized obstruents (nasal plus weak obstruent) and the strong obstruents (doubled weak obstruent). The set of valid short onsets consists of all other consonant phonemes in Koalib.

After defining the sets of valid nuclei (N), short onsets (S) and long onsets (L) for the particular language, we can use the syllable splitter to extract the syllables from words. It finds the start index of each syllable and then places syllable boundaries at those positions. The procedure for finding the start indices of syllables works as follows:

```
Data: a word
Result: a list of the start indices of the syllables in the word
initialize an empty list;
foreach letter in word do
    if letter ∈ N then
        prefix ← word.substring(0, index(letter));
        if prefix.endsWith(l ∈ L) then
            if index(l) = 0 or word.substring(0,
            index(l)).endsWith(s ∈ S) then
                add index(l) to list;
            else if prefix.endsWith(s ∈ S) then
                add index(s) to list;
        else if prefix.endsWith(s ∈ S) then
            add index(s) to list;
        else
            add index(letter) to list;
        end
    end
end
```

Algorithm 1: How to find the start indices of syllables in words

Note that S contains all the individual consonant letters, as every single consonant letter is also a phoneme. It is also important to keep in mind that certain letters are actually encoded as several characters, so iterating over the characters in the word may not always return the complete letters.

4.1.3 Evaluation

After extracting the classes and features from the corpus, the resulting data set is exported into an ARFF file. Then it is used to train and evaluate models with Weka. The main method for evaluation is stratified 10-fold cross-validation. This means that the entire data set is partitioned into 10 subsets. Then for each one of the 10 folds, a model is trained on 9 of the subsets and tested on the remaining one, so that in the end each subset has been used for testing once. Stratification ensures that the distribution of the class labels is roughly the same in each subset.

Cross-validation accuracy provides a good estimate of how good the model is at predicting the class label for unseen data. In our case, it shows how reliable the rules are when used for new nouns. In addition to cross-validation, the accuracy is also measured on the training set. This is done simply by training the model on the whole data set and testing it on the same. The train accuracy shows how accurate the rules are, relative to the nouns they were learned from. By comparing the cross-validation and the train accuracy, we can see whether the model "overfits" the data set. This happens when the features are too specific and results in a model that has a very high train accuracy but a poor cross-validation accuracy.

It should be noted that the data set is also randomized before stratification. This means that the instances are shuffled, so that they do not appear in any particular order. This shuffling has a slight effect on the accuracy of the model, as it determines which instances are used for training and which ones for testing. Since some instances are better training examples than others, the randomization of the data causes the reported accuracy to vary slightly, but these differences are never significant. The shuffling is performed according to a random number generator. The random number generator, as the name implies, generates random numbers that are used for reordering the instances. It can be given a number as an argument and this number is used as the "seed" for the number generation, so that two random number generators with the same seed always produce the same sequence of "random" numbers if used in exactly the same way.

4.1.4 Optimization

The final step was to optimize the model for each experiment. This involves adjusting the parameters of the machine learning algorithm and selecting the features that maximize the cross-validation accuracy. Features were selected with the help of the InfoGain algorithm. InfoGain evaluates each feature by measuring the information gain with respect to the class. This evaluation can then be used to rank the features according to their usefulness.

The features for each experiment were selected by measuring the cross-validation accuracy with the full feature set, then subsequently reranking those features and removing the least useful one, then measuring the new cross-validation accuracy. The process is repeated until the maximum accuracy has been reached. Finally, only the features that maximize the accuracy are used.

4.2 Results

The two methods for predicting the object form - multiple classifiers and single classifier, were thoroughly tested and compared to a baseline system which simply assigns the most common class label. Furthermore, four different versions of the data set were created. Two of them include all the nouns, while the other two include only the disyllabic nouns. The reason for this is that most of the nouns in the corpus are disyllabic and they may be more representative of case inflection than monosyllabic or other polysyllabic nouns. Furthermore, two versions of the corpus had their classes and features extracted from the complete subject and object forms, while the other two had them extracted from the root morphemes. The motivation for this distinction is that the prefixes of nouns do not seem to affect case inflection, so it may be better to consider only the root mopheme and ignore the prefix of the subject or object form. Thus, the following data sets were used for testing:

- all subjects
- all roots
- disyllabic subjects

- disyllabic roots

All experiments were performed using the RIPPER rule learner, whose implementation in Weka is called JRip. Though in some cases a higher accuracy can be achieved by using C4.5 decision trees instead, the difference is insignificant. I have decided to use a rule learner over a decision tree, because the model of a rule learner does not require addition postprocessing to extract rules.

In all experiments, JRip was set to do pruning of the data, in order to deal with very rare feature values and class labels that may lead to overfitting and lower the accuracy. The algorithm was set to use only rules that apply to at least 2 instances. Lastly, the seed used for the randomization of the data was 1.

4.2.1 Multiple Classifiers

The first method for predicting the object form is using multiple classifiers, where each classifier predicts a different aspect of the object-form change. This results in two prediction tasks - predicting the class *suffix* and predicting the class *tone change*. The features for each task are a subset of the features described in the previous section. They also vary depending on whether the data set includes all nouns or only disyllabic ones, with less features being needed for disyllabic nouns.

9 features are selected for predicting the *suffix* class for the data sets *all subjects* and *all roots*, and they are the following:

- last phoneme
- last two phonemes
- last syllable
- last two phoneme types
- last syllable type
- last two syllable types
- last two tones
- squeezed tone pattern

- complete tone pattern

For the data sets *disyllabic subjects* and *disyllabic roots*, the features for the *suffix* class are the following 6:

- last phoneme
- last two phonemes
- last syllable
- last two syllable types
- squeezed tone pattern
- complete tone pattern

As for the *tone change* class, 9 features for the data sets *all subjects* and *all roots* are chosen:

- last phoneme
- last two phonemes
- first two syllable types
- last syllable type
- last two syllable types
- last tone
- last two tones
- squeezed tone pattern
- complete tone pattern

And the features for the *tone change* class for the data sets *disyllabic subjects* and *disyllabic roots* are the following 8:

- last phoneme
- last two phonemes
- last syllable type
- last two syllable types

- last tone
- last two tones
- squeezed tone pattern
- complete tone pattern

The table below summarizes the resulting accuracies and compares them to the baseline system:

Class	Accuracy	Data set			
		all		disyllabic	
		subjects	roots	subjects	roots
suffix	base	40.46%	40.46%	39.67%	46%
	test	**81.09%**	**78.86%**	**75.14%**	**83.62%**
	train	83.08%	80.85%	81.39%	84.83%
tone	base	30.85%	34.25%	28.67%	31.19%
	test	**66.42%**	**66.25%**	**67.93%**	**67.84%**
	train	73.22%	69.65%	73.51%	72.69%

The results show that the classifiers are much more reliable than the baseline system. It seems that predicting the suffix is easier than predicting the tone change, perhaps due to the large number of possible tone changes.

4.2.2 Single Classifier

The second method for predicting the object form is to use a single classifier that predicts the whole change in one go. This way, the class *full change* is a combination of the classes for the two multiple classifiers. Once again, the feature sets for all nouns are different from those for disyllabic ones.

12 features are selected for predicting the *full change* for the data sets *all subjects* and *all roots*. These are all the features discussed in the previous section:

- last phoneme
- last two phonemes
- last syllable

- last phoneme type
- last two phoneme types
- first two syllable types
- last syllable type
- last two syllable types
- last tone
- last two tones
- squeezed tone pattern
- complete tone pattern

As for the data sets *disyllabic subjects* and *disyllabic roots*, only 11 features are selected for predicting the *full change*:

- last phoneme
- last two phonemes
- last syllable
- last phoneme type
- last two phoneme types
- last syllable type
- last two syllable types
- last tone
- last two tones
- squeezed tone pattern
- complete tone pattern

The table below reports the accuracies and compares them to the baseline system, as well as a system that uses the multiple-classifier method to predict the suffix and tone change separately, then combines them together:

Class	Accuracy	Data set			
		all		disyllabic	
		subjects	roots	subjects	roots
full	base	24.63%	25.12%	28.26%	20.87%
	multiple	62.52%	59.87%	60.05%	64.44%
	test	**64.59%**	**66.25%**	**63.99%**	**69.05%**
	train	72.06%	70.73%	72.55%	71.84%

The results show that the single-classifier method is more accurate than using the multiple classifiers and then combining their predictions. This is probably, because it accounts for any dependencies between the two types of change. Furthermore, it appears that concentrating on the roots of the nouns and ignoring their prefixes yields better results. This difference is not very big, because each nominal root appears only once in the corpus. However, some prefixes do affect the values of certain features.

4.2.3 Alternative Methods

In addition to the multiple- and single-classifier methods, the pipeline method was also implemented and tested. It consists of two classifiers - the first one predicts the suffix, just like the first classifier in the multiple-classifier method. The second one predicts the tone change, similarly to the multiple-classifier method, but it also uses the suffixes, predicted by the previous classifier as features. This allows it to capture some dependencies between the two classes when predicting the second one.

The classifiers in the pipeline are always trained on the correct features from the training data. This means that the second classifier trains on the correct suffixes. However, in the testing phase it does not have access to the correct values of this feature, therefore it uses the suffixes that are predicted by the previous classifier. Since Weka does not implement such a functionality, to the best of my knowledge, I have implemented the pipeline design myself.

The pipeline classifiers use exactly the same parameters and features as those in the multiple-classifier method, except that each successive classifier also uses the class of the previous one as an extra feature. The table below summarizes the accuracies for the

pipeline method and compares it to the two main methods:

Class	Accuracy	Data set			
		all		disyllabic	
		subjects	roots	subjects	roots
suffix	multiple	**81.09%**	**78.86%**	**75.14%**	**83.62%**
	pipeline	80.35%	78.52%	76.09%	83.74%
tone	multiple	**66.42%**	**66.25%**	**67.93%**	**67.84%**
	pipeline	80.18%	80.51%	79.48%	78.88%
full	multiple	62.52%	59.87%	60.05%	64.44%
	single	**64.59%**	**66.25%**	**63.99%**	**69.05%**
	pipeline	67%	66%	63.32%	68.93%

As the table shows, the pipeline is about as good at predicting the suffix, as the classifier in the multiple method. This is normal, because the features are exactly the same. The differences are probably due to a different way of randomizing data, but they are very small of course. However, the pipeline is very good at predicting the second class - the tone change. This is because the information from the suffix classifier seems to be helpful. However, when the two labels are combined and compared to the label, predicted by the single classifier, there is not much difference between the two. Apparently, the single classifier is equally good at capturing the dependencies between the two classes.

Since the pipeline method did not prove to be much more effective than the single-classifier method for predicting the whole change, I would not use it for discovering the rules, because it would only produce more complicated rules, that also require knowledge of the suffix of the object to predict its tone pattern. However, I consider it worthy of mentioning.

4.3 Discussion

Considering the complexity of the task and the limited amount of available data, it seems that the classifiers manage to achieve a reasonable accuracy for unseen data. The entire object-form change can be predicted correctly in about 66% of the cases. The rest of the object-form changes are either too irregular to predict or involve properties that are not available from the data.

As for discovering some actual rules, it seems that it is better to look at the root morphemes and ignore the prefixes of nouns. The object forms of disyllabic nouns are slightly easier to predict, but not much more different than predicting them for any nouns. For the experiments, the rule learner was set to use any rules that apply to at least two cases. This yields good accuracies, but some of the rules are not very interesting from a practical point of view. By setting the minimum instances per rule higher, we can obtain smaller models with fewer, but more interesting rules, at the cost of a slight decrease in accuracy.

Finally, I would like to present some discovered rules. These rules are all extracted from the *all roots* data set. The parameters and features used are exactly the same as those in the previously mentioned experiments, except that JRip was set to use only rules that cover at least 10 instances. This greatly reduces the size of the model for a small decrease in accuracy.

First, here are some rules for predicting the *suffix* only. They are able to predict it for unfamiliar nouns with 77.2% accuracy:

- (LAST-TWO-TONES = LH) and (LAST-TWO-PHONEME-TYPES = CV) => SUFFIX-CLASS=ŋe (104.0/12.0)
- (LAST-TWO-TONES = L3H) and (LAST-TWO-PHONEME-TYPES = CV) => SUFFIX-CLASS=ŋe (22.0/1.0)
- (LAST-TWO-TONES = LH) and (LAST-TWO-PHONEME-TYPES = VC) => SUFFIX-CLASS=e (109.0/11.0)
- (LAST-TWO-PHONEME-TYPES = VC) and (LAST-TWO-TONES = L3H) => SUFFIX-CLASS=e (18.0/1.0)
- (LAST-TWO-PHONEME-TYPES = VC) and (LAST-TWO-PHONEMES = AN) and (LAST-TWO-TONES = HL) => SUFFIX-CLASS=e (38.0/13.0)
- (LAST-PHONEME = A) => SUFFIX-CLASS=NONE (147.0/7.0)
- (LAST-TWO-PHONEME-TYPES = CV) and (SQUEEZED-TONE-PATTERN = L) => SUFFIX-CLASS=NONE (58.0/13.0)
- (LAST-TWO-TONES = L) => SUFFIX-CLASS=NONE (102.0/47.0)

- (LAST-TWO-PHONEME-TYPES = CV) and (LAST-TWO-SYLLABLE-TYPES = VC.CV) => SUFFIX-CLASS=NONE (43.0/18.0)
- (LAST-TWO-SYLLABLE-TYPES = VV.CV) => SUFFIX-CLASS=NONE (39.0/17.0)
- (LAST-TWO-TONES = H) and (LAST-SYLLABLE-TYPE = VC) => SUFFIX-CLASS=NONE (15.0/4.0)
- => SUFFIX-CLASS=a (511.0/110.0)

As you can see, the rules have two sides. The left side contains some conditions, which operate in conjunction with each other. Each condition checks if a specific feature has a specific value. If this is the case, then the condition is fulfilled. If all conditions are fulfilled, the rule predicts a class - in this case a suffix. Otherwise, we move on to the next rule and check if its conditions are fulfilled. Finally, if no rule is applied, we reach the default rule, which has no conditions, and apply it. The numbers in the brackets show the reliability of each rule. The first number indicates how many instances the rule applies to in total, while the second number indicates the number of instances for which the rule predicted a wrong class.

The following rules are used to predict the *tone change*. They have an accuracy of 64.01% on unseen data:

- (LAST-TWO-TONES = L3H) => TONE-CHANGE-CLASS=L3H-LHH (41.0/3.0)
- (COMPLETE-TONE-PATTERN = LL) and (FIRST-TWO-SYLLABLE-TYPES = VC.CVC) => TONE-CHANGE-CLASS=LL-HLH (45.0/9.0)
- (COMPLETE-TONE-PATTERN = HL) and (LAST-SYLLABLE-TYPE = CVC) => TONE-CHANGE-CLASS=HL-LHL (132.0/35.0)
- (COMPLETE-TONE-PATTERN = HH) and (LAST-SYLLABLE-TYPE = CVC) => TONE-CHANGE-CLASS=HH-LLH (98.0/8.0)
- (LAST-TONE = L) and (LAST-PHONEME = A) => TONE-CHANGE-CLASS=NO-CHANGE (137.0/47.0)
- (SQUEEZED-TONE-PATTERN = L) and (LAST-TWO-TONES = L) => TONE-CHANGE-CLASS=NO-CHANGE (108.0/51.0)

- => TONE-CHANGE-CLASS=ADD-H (645.0/282.0)

Finally, these rules predict the whole inflection. Their accuracy on unseen data is 64.43%:

- (LAST-TWO-TONES = L3H) and (LAST-PHONEME-TYPE = C) => FULL-CHANGE-CLASS=L3H-LHH + e (18.0/1.0)
- (LAST-TWO-TONES = L3H) => FULL-CHANGE-CLASS=L3H-LHH + ŋe (23.0/2.0)
- (LAST-TWO-PHONEMES = AN) and (SQUEEZED-TONE-PATTERN = HL) => FULL-CHANGE-CLASS=HL-LHL + e (33.0/12.0)
- (COMPLETE-TONE-PATTERN = LL) and (FIRST-TWO-SYLLABLE-TYPES = VC.CVC) => FULL-CHANGE-CLASS=LL-HLH + a (45.0/9.0)
- (COMPLETE-TONE-PATTERN = HL) and (LAST-SYLLABLE-TYPE = CVC) => FULL-CHANGE-CLASS=HL-LHL + a (103.0/35.0)
- (LAST-TWO-TONES = LH) and (LAST-PHONEME-TYPE = V) => FULL-CHANGE-CLASS=ADD-H + ŋe (111.0/15.0)
- (COMPLETE-TONE-PATTERN = HH) and (LAST-SYLLABLE-TYPE = CVC) => FULL-CHANGE-CLASS=HH-LLH + a (98.0/8.0)
- (LAST-TWO-TONES = LH) => FULL-CHANGE-CLASS=ADD-H + e (109.0/12.0)
- (SQUEEZED-TONE-PATTERN = LHL) and (LAST-PHONEME = E) => FULL-CHANGE-CLASS=ADD-H + a (22.0/2.0)
- (LAST-PHONEME = O) and (SQUEEZED-TONE-PATTERN = LHL) => FULL-CHANGE-CLASS=ADD-H + a (16.0/1.0)
- (LAST-PHONEME = O) and (SQUEEZED-TONE-PATTERN = HL) => FULL-CHANGE-CLASS=ADD-H + a (51.0/25.0)
- (FIRST-TWO-SYLLABLE-TYPES = CVC.CVC) => FULL-CHANGE-CLASS=ADD-H + a (29.0/9.0)
- (SQUEEZED-TONE-PATTERN = HL) and (LAST-PHONEME = E) => FULL-CHANGE-CLASS=ADD-H + a (58.0/25.0)
- => FULL-CHANGE-CLASS=NO-CHANGE + NONE (490.0/260.0)

Thus, even with so few rules, it is possible to correctly predict the object form for a good portion of nouns.

5 Conclusion

This thesis demonstrates how machine learning methods can be used to discover case inflection rules in Koalib. By knowing what properties of nouns are involved in the case inflection, it is possible to learn the rules for that automatically from training data. Though these rules are far from perfect, they are able to predict the change from subject to object form for unfamiliar nouns with about 66% accuracy, while covering all of the available data. Whether the rest of the object forms require additional information to predict or are simply too irregular to follow any rules, is uncertain.

The automatically discovered rules provide a good insight into the grammar of this remarkable language. However, there are some even stranger phenomena, regarding case inflection, that this study does not cover. For example, certain nouns in Koalib appear to have more than one object form (Quint, unp). This study does not attempt to predict which nouns have two object forms and which have one, as it focuses on predicting only one of them. However, it might be interesting to account for those cases as well, in future studies.

But such nouns are quite rare and its possible that only one of the object forms is preferred, while the other one is rare or archaic. What could be more important for discovering case inflection rules, would be to examine if there are other potentially useful features, beyond the ones that can be extracted from the current corpus, and collect an even larger corpus with such additional information, as that would provide more examples for machine learning and result in a more reliable model. But I have tried to create the optimal model, given the available data, and I think that the rules discovered from it are already quite useful in giving an insight into the grammar of the Koalib language.

References

Cohen, W. W. (1995). Fast effective rule induction. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 115--123. Morgan Kaufmann.

Furnkranz, J. and Widmer, G. (1994). Incremental reduced error pruning. In *International Conference on Machine Learning*, pages 70--77.

Quinlan, J. R. (1986). Induction of decision trees. *Machine learning*, 1(1):81--106.

Quinlan, J. R. (1987). Generating production rules from decision trees. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, volume 30107, pages 304--307. Citeseer.

Quinlan, J. R. (1993). *C4.5: Programs for machine learning*, volume 1. Morgan Kaufmann.

Quint, N. (2006). Do you speak Kordofanian? '*Fifty Years After Independence: Sudan's Quest for Peace, Stability and Identity*'.

Quint, N. (2010a). Benefactive and malefactive verb extensions in the Koalib verb system. *Benefactives and malefactives. Typological perspectives and case studies*, pages 295--315.

Quint, N. (2010b). *The phonology of Koalib*. Rüdiger Köppe.

Quint, N. (unp.). Case in Koalib (a Kordofanian language) and related Heibanian languages. To be published.

Schadeberg, T. C. (1981). *A survey of Kordofanian: The Heiban group*, volume 1. H. Buske.

Schadeberg, T. C. et al. (1989). Kordofanian. *The Niger-Congo Languages*, page 66.

Stevenson, R. C. (1956). *A survey of the phonetics and grammatical structure of the Nuba Mountain languages, with particular reference to Otoro, Katcha and Nyima*. Reimer.

Witten, I. H., Frank, E., and Hall, M. A. (2011). *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann.