Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Data Mining Research Group

Auto-cleaning Dirty Data: the Data Encoding Bot

2IMM00: Seminar Data Mining

Angelo Majoor - 1030843
A.R.Majoor@student.tue.nl

Supervisor:
dr. ir. J. (Joaquin) Vanschoren

Final Report

Eindhoven, January 17, 2018

Contents

Contents ii

1 Introduction 1

2 Problem Formulation 3

3 Literature Analysis 5
- 3.1 Auto-Detecting Data Types . . . 5
  - 3.1.1 Brute-Force Guessing . . . 5
  - 3.1.2 Python Pandas . . . 6
- 3.2 Numeric, Categorical or Ordinal Data? . . . 7
  - 3.2.1 Simple Approach . . . 7
  - 3.2.2 Bayesian Method . . . 7
- 3.3 Auto-Selecting Encoding Techniques . . . 8

4 Proposed Solution 9
- 4.1 Auto-Detecting Data Types . . . 9
- 4.2 Numeric, Categorical or Ordinal Data? . . . 11
- 4.3 Auto-Selecting Encoding Techniques . . . 13

5 Results 14
- 5.1 Auto-Detecting Data Types . . . 14
- 5.2 Numeric, Categorical or Ordinal Data? . . . 15
- 5.3 Auto-Selecting Encoding Techniques . . . 15

6 Conclusion 16

7 Discussion 17

Bibliography 18

Appendix 19

A Predictions & Ground Truths 19

B Code Implementation 22

Chapter 1

Introduction

Context — Machine Learning (ML) is growing more popular by the day. Training computers to make them think and act autonomously is one of the new holy grails. A well-trained ML model can discover knowledge that people would have never found out about manually. These models are trained on data, to be able to act on (predict) other data as well. ML algorithms are at the core of creating such models, but these algorithms need preprocessed, clean data as an input. Most of this preprocessing is done on an individual basis per data set and takes a significant amount of time. Unfortunately, with the ever increasing amount of data that is being produced in this modern era, so is the raw data becoming messier and messier and as such, preprocessing data manually is very costly.

Problem — One of the major problems of cleaning raw, dirty data, is the manual interference, which takes time and money. Therefore, a valuable contribution would be to automate this entire process. Such a process could consist of four different steps [9]: *automatic data encoding*, *automatic outlier detection*, *automatic value imputation* and *automatic feature selection*. Unfortunately, all four of these steps combined together form a too big scope for this paper and as such, let us focus on the first part of such an automated process: *the automatic data encoding bot*. Say we have an input data set, raw and dirty, containing a number of columns, but without any further knowledge on the provided data. Optimally, for each of the columns our bot will

1. be able to tell what the data type of each particular column is.
2. be able to tell whether any of the columns contains categorical and / or ordinal data.
3. be able to find and return the best way to encode the data as input for the algorithms.

State of the Art — For the first step of our problem, a number of solutions have been proposed. One approach proposed by [6] tries to convert every single cell in a column from a data set into every possible data type, aggregating the number of successful conversions of a column to each of these types, weighing them by a fixed type priority and selecting the most probable type based on this information. Another approach is the one introduced in the Python language, which already has automatic data type inference in their Pandas library [5]. The second step of the problem already is a lot more difficult, with fewer existing solutions. The basic approach by [1] simply checks features for their data type and uses a critical value to determine whether any of these features can be seen as categorical. A more involved solution by [8] has been proposed only very recently, which uses a more statistical orientation in the form of a likelihood selection method to differentiate between numeric, categorical and ordinal features. The last step of the problem has nearly no proposed solutions, indicating that the difficulty of solving this problem is severe (and only few people have tried so far), leaving the opportunity to create novel approaches for finding an optimal encoding for the input data.

Solution — The solution we have created takes as an input a raw data set and turns it into an encoded version of the particular data set. For the first step of our problem we have used the functionality of the Python Pandas *read_csv()* function, which already automatically infers both integers and floating-point numbers (floats). We have chosen to use this function, because it is very easy to use and has already been used (and therefore tested) extensively by people all over the world. The other data types, boolean (bool), date and string, are then checked for based on a few simple rules: when a feature contains exactly two unique values, it is considered boolean, when the checked values in a feature have the '-' or '/' symbol at specific positions in the values, it is considered a date and everything else is considered to be of the string data type.

The second step, checking for categorical, as well as ordinal features, uses an approach that is based on human nature. When a certain feature contains less than a specific (user-determined) number of unique values, calculate either a similarity score (for strings) or a distance score (for integers), indicating whether these values in a particular feature are related. If a feature 'passes' this test, it is marked as categorical. Unfortunately, checking for ordinal features was and is an extremely difficult problem, which we were unable to solve in the limited time provided.

The last step, finally, creates an encoding for the raw data set, based on the information found in the two previous steps. Both categorical string and integers are *one-hot encoded* (creating a feature for every unique value in a particular feature and indicating occurrence), with an option to create an 'other' feature for values that occur less than a predefined number of times. The remaining bool, date and non-categorical string features are then optionally *factorized* (turning values into integers, with every unique value being assigned a unique integer) to provide an encoding entirely consisting of numbers, which can serve as the input to any possible next step.

Results — After testing the proposed solution, an accuracy score can be calculated for both the first and the second step. This accuracy score is calculated by taking the number of correctly predicted features, divided by the total number of checked features, multiplied by 100% to get a percentage indicating the performance on actual raw, test data sets. For step 1, automatically inferring feature data types, we show that our solution predicts all data types right 100% of the time, on the used test data sets. For step 2, checking for categorical, as well as ordinal data types, we show that the proposed solution is scoring between 70% to 100% accuracy, with an average of 91.7% accuracy. The last step, finding an optimal encoding for the raw data set, has no quantitative or qualitative measures, since it is very hard to express such an encoding in terms of 'successful' and 'unsuccessful' aspects. An encoding is created, with several options for the user to be specified, but whether it is optimal is for the specific user to decide, based on the consequent actions the user wants to perform on the encoding.

Chapter 2 discusses the problem statement in more detail, while chapter 3 dives into the existent solutions to the problem and subproblems. After that, chapter 4 extensively describes our created solution, with the produced results shown in chapter 5. Lastly, chapter 6 draws a number of conclusions on the research performed, whilst chapter 7 discusses points of improvement and any further research.

Chapter 2

Problem Formulation

Creating a data encoding bot can be split into three different steps:

1. Auto-detecting data types per feature (column).
2. Auto-detecting numeric, ordinal, categorical features.
3. Auto-selecting encoding techniques for all features.

Step 1 — The problem of automatically detecting data types narrows down to creating a model that is able to tell us which data types are involved in the dirty data that is being provided. The model is given a CSV file as input, containing multiple columns with data, without any further information on this data. Now, for a human being it is reasonably easy to tell what type of information is contained in this dirty data. That is, we human beings can easily differentiate between several types of data and different formats of the same data type (i.e. a date can be written in several different formats, depending on the part of the world someone comes from). However, for a computer this is not as easy. A computer has to be told how to be able to distinguish these different types of data. Therefore, one of the aims of this paper is to provide a solution to the auto-detecting data types problem. As such, the first research questions that is being addressed in this paper is:

1. How can we optimally auto-detect data types from a dirty data set?

Step 2 — After we have performed the first step and have automatically detected the different data types that make up the data set, one major problem remains: for integer and string features, do the values in a column represent a numeric value, a categorical value, or an ordinal value? This is very important for an autonomous bot to know to be able to find optimal models and learn over time. If we do not know anything about this aspect, our bot could interpret categorical values as mere integers, possibly leading to dramatically incorrect models and predictions. For instance, a date might be one of the easier examples, but what if a feature contains the numbers 1, 2 and 3? Do these numbers represent categories (categorical data), an order (ordinal data), or are these values literally just representing numbers (numeric data)? Let us take a look at the second column in our example table 2.1, where we have eight integer values between 1 and 5. These values could be the answers to some very simple calculations, which all happen to be between 1 and 5, but they could also represent five very different categories, maybe even representing an order (e.g. grades A, B, C, D and F). Therefore, one of the aims of this paper is to provide a solution to the problem of determining whether an integer is numeric, categorical or ordinal. As such, the second research question that is being addressed in this paper is:

2. How can we optimally auto-differentiate between numeric, categorical and ordinal features?

Step 3 — Finally, after performing steps 1 and 2, our bot has enough information about the different data types that the features have, to be able to encode the data set in such a way that it can move on to the next phase of the auto-cleaning process and eventually be ready as input to the algorithm. This encoding, however, can be a tricky problem in itself. To be able to create an optimal encoding for finding models within the data set, one has to know the algorithm that will be applied. Decision trees, Support Vector Machines and the Nearest Neighbor principles, for example, all have very different ways to find an optimal model from the input data and as such, the encoding can and needs to be very different for all of these algorithms. In the end, this all comes down to deciding whether features need to be one-hot encoded or not and more importantly, which features. Therefore, one of the aims of this paper is to provide a solution to the auto-selecting encoding techniques problem. As such, the third and last research question that is being addressed in this paper is:

3. How can we optimally auto-select encoding techniques, based on the different algorithms that exist?

Table 2.1: Example input - raw, dirty data

R / C	**1**	**2**	**3**	**4**	**5**
1	31	1		Seminar	c
2	6	5	09/12/1992	technical	g
3	120	4	1958/05/23	Mining	a
4	78	5	11/10/2001	msc	a
5	65	2	10-09-2005	data	d
6		1	02-03-1961	eindhoven	b
7	27	3	1974-12-10	Fall	g
8	112	3	29/03/1998	University	d

Chapter 3

Literature Analysis

Similar to the problem formulation, this chapter covers all three aspects of the data encoding bot, pointing out a number of solutions that exist as of today.

3.1 Auto-Detecting Data Types

Auto-detecting data types might probably be the easiest aspect of the data encoding bot. Several solutions to the problem already exist, some of them more sophisticated than others, but at least a variety in approaches has been proposed. Two different approaches will be covered in the next sections: one very logical to understand, the other an already integrated one.

3.1.1 Brute-Force Guessing

A first solution, proposed by [6], tries perhaps the most obvious way of type detection: brute-force guessing. Brute force guessing implies converting every single element in a particular column into all possible data types and counting the number of successful conversions per data type. A majority vote then determines the most probable data type for that particular column. Several data types are checked against (taken from [6]):

- String (weight 1): denotes a string or other converted type.
- Integer (weight 6): denotes an integer.
- Decimal (weight 4): denotes a decimal number.
- Bool (weight 7): denotes a boolean value.
- Date (weight 3): denotes a date.

What the algorithm then does, is aggregating the number of successful conversions of a column to each of these types, weighing them by a fixed type priority and selecting the most probable type based on this information.

Let us have a look at our example table 2.1 once more to familiarize ourselves with this approach. The first column of our table shows several 'numbers', with one empty cell in the sixth row. Except for the empty cell, all of the values can be converted into a string, into an integer and into a decimal value. As a result, the scores for these three data types will be equal to 7 (= the number of successful conversions). By multiplying this number with the corresponding weight we get the scores as shown in the last column of table 3.1. As can be seen from this table, the integer data type has the highest weighted score and therefore the first column of the example data set (table 2.1) is most likely to be of the integer data type.

Table 3.1: Brute-force guessing on the first column of the example input

Successful conversion	Unweighted	Weight	Weighted
String	7	1	7
Integer	7	6	42
Decimal	7	4	28
Bool	0	7	0
Date	0	3	0

Advantages & Disadvantages

One of the main advantages of this approach is that is produces relatively good results, compared to the 'ground truth', since the approach takes into account every single cell in the data set. However, taking into account all of the cells in the provided data set is also one of the disadvantages of this approach, simply because this takes a lot of time, especially for the bigger data sets that have to be dealt with nowadays. Also, this approach does not take into account differences in numeric data, such as categorical data versus ordinal data, which is another major disadvantage.

3.1.2 Python Pandas

Python already has an automatic data type inferer included in their Pandas library [5]. This means that whenever you read a CSV file with Pandas, it will automatically decide on the type of every single column. Again, let us have a look at the example table 2.1 to see how the Pandas library works. For example, by executing the following very simple code snippet, we can retrieve the data types for all 5 columns from our example data set. Note that the first column is inferred as a float, instead of an integer, due to the missing value.

```
import numpy as np
import pandas as pd

raw_data = pd.read_csv('Example_input.csv',delimiter=';',header=None)
raw_data.dtypes

0     float64    <class 'numpy.float64'>
1       int64    <class 'numpy.int64'>
2      object    <class 'str'>
3      object    <class 'str'>
4      object    <class 'str'>
dtype: object
```

Advantages & Disadvantages

The main advantage of using the Pandas library is that it has already been implemented and is very easy to use: only basic knowledge of Python (and Pandas) is sufficient to perform this task. Another advantage is that it seems to work reasonably well for most columns containing numbers. However, note that all the other columns are simply identified as 'strings', which does not do a very good job on our columns containing dates. Also, the last column in our example might represent categorical or ordinal data, but since this is represented by a single letter, Pandas recognizes these values as strings as well, implying that we have a problem with categorical and ordinal data that is not represented by numbers.

3.2 Numeric, Categorical or Ordinal Data?

Several methods have been proposed, although automatic data 'type' (numeric, categorical, ordinal) detection, as well as machine learning in general, is a relatively untrodden path. Similar to the previous section, two approaches will be covered: a very basic implementation and a more complicated one, based on a statistical approach.

3.2.1 Simple Approach

A first solution to the problem is the approach by [1], which is very simple and easy to understand. Note that this approach only checks for categorical data, not ordinal data. A column from an input data set has to fulfill two criteria to qualify for the title of categorical data:

1. A column can only contains either integers, or strings.
2. A column cannot have more than a predefined number of different values.

This means that, as a first step, all columns have to be checked for their data type and if and only if all of the cells within a column have either the integer data type or all of the cells have the string data type, then can we move on to the next step. The second step then checks whether or not all of the columns that satisfied the criterion from the first step have less than or equal to n unique values, where n can be specified by the user. So, if we take a look at the example input 2.1 again, then we can see that we have several columns that could pass the test: columns 2, 3, 4 and 5. Column 1 has an empty cell and as such does not fulfill step 1. Column 3 also has an empty cell, but this is considered an empty string and therefore the entire column can be considered to be of the string data type (since [5] showed that dates are inferred as strings as well). Now, if we set our $n = 5$, then only the second column passes the criterion of step 2, meaning that this column could possibly contain categorical data. Note that this clearly depends on the value that we set for n.

Advantages & Disadvantages

The major advantage of using this approach is the ability to specify the value for the critical value n. However, no guarantees can be made on the true interpretation of a particular column that has earned the label 'categorical'. Another disadvantage is that the method used only takes into account categorical data, whilst ordinal data should be of great importance as well. Also, checking every cell for the first step of the approach is time-consuming and seems to be unnecessary and more specifically the concern for another step during the auto-cleaning process.

3.2.2 Bayesian Method

A very recent advance in the particular field of this kind of data type detection is the solution as proposed by [8]. This approach has a more statistical nature, where the proposed method can be seen as a likelihood selection method. The paper distinguishes both continuous as well as discrete data and splits discrete data into three different types of data (taken from [8]):

- Count data: each observation takes a non-negative integer value.
- Categorical data: each observation takes a value in an unordered index set, such as blue, red, black.
- Ordinal data: each observation takes a value in an ordered index set, such as never, sometimes, often, usually, always.

For each of these data types, a likelihood function is created that determines (calculates) per class how likely it is that an observation belongs to that particular class. As a result, every attribute is assigned a 'most probable' data type, as well as the number of different categories that exist for a discrete attribute.

Advantages & Disadvantages

Since this is a very recent proposal, it is difficult to determine how well this approach works for raw, dirty data. According to their own paper, they 'can conclude that i) [their] model accurately discovers the true statistical type of the data, which might not be easily extracted from its documentation; and by doing so, ii) it provides a better fit of the data. Moreover, apparent failures are in fact sensible when data histograms are carefully examined' [8]. This leads us straight to the shortcomings of this approach. As they describe themselves, the likelihood models do not guarantee a 100% correct results for every attribute in a data set and need further (manual) examination to check for flaws in the data. Something we are explicitly trying to avoid, since we eventually want to create a fully autonomous data encoding bot.

3.3 Auto-Selecting Encoding Techniques

This problem is still very new within Machine Learning and as such, little to no solutions to the problem have been proposed (yet). One of the more interesting posts of recent years concerning an optimal encoding for categorical data (alas nothing about ordinal data), is the one by [4]. In this paper a mini benchmark is performed to compare different representations for categorical data, of which the two most interesting ones are *one-hot encoding* and *binary encoding*. The former transform a categorical feature into as many different features as there are unique values (categories) in the categorical feature, with a 1 representing a row belonging to that particular category. The latter, on the other hand, first translates the different categories into numeric values, to convert those numeric values into binary code. After that, a column is created for every digit that the binary string contains. 'This encodes the data in fewer dimensions than one-hot, but with some distortion of the distances.' [4]

Since there is a wide variety in algorithms that are used to generate a model, deciding on an optimal encoding for the general purpose is very hard and no solution to this specific problem exists as of today. Possible approaches for this problem will be explained further in the upcoming sections.

Chapter 4

Proposed Solution

Again, similar to the previous chapters, this chapter covers all three aspects of the data encoding bot, extensively describing the proposed solution for each of these steps. The actual implementation in code (Jupyter Notebook with Python) can be found in appendix B. To see how the actual implementation works, we use one of the last data sets from the test data, based on Spotify song rankings, to show the different functionalities. The first five columns of this data set are shown in table 4.1, with the URL column (column 5) removed due to spatial issues within this report.

Table 4.1: Spotify song rankings test data set

Position	Track Name	Artist	Streams	Date	Region
1	Reggaetón Lento (Bailemos)	CNCO	19272	2017-01-01	ec
2	Chantaje	Shakira	19270	2017-01-01	ec
3	Otra Vez (feat. J Balvin)	Zion & Lennox	15761	2017-01-01	ec
4	Vente Pa' Ca	Ricky Martin	14954	2017-01-01	ec
5	Safari	J Balvin	14269	2017-01-01	ec

4.1 Auto-Detecting Data Types

For the first step of the data encoding bot, we propose a solution that is basically a combination of the two solutions that have been discussed in chapter 3, but worked out in more detail to serve the purpose of detecting all of the data types that we would like to detect: bools, dates, floats, integers and strings. As mentioned before, the input of the entire process is a raw, dirty data set in the form of a CSV file. As a start, this CSV file is imported with the read_csv() function from Python Pandas, which is already able to automatically infer both integers and floats. Chapter 5 on results discusses how well this function performs, but for now let us assume that this function is able to perfectly identify the two data types mentioned. Unfortunately, since we are dealing with raw data, Python Pandas is not always able to infer integer or float features as their corresponding type due to, for example, missing values. As a result, we still have to check the remaining features for all five different data types. To detect all of these different data types we have implemented a few simple rules.

First of all, bools are very easy to detect, since these features have exactly two unique values. Therefore, any feature that is not inferred as the integer or float data type by Python Pandas itself and has exactly two unique values, is assigned the bool data type. Some people might note that integers can be of the bool data type as well, which is very true. To detect this, also features that are already inferred as the integer data type by Python Pandas are being checked on their number of unique values. When the result of this check is equal to two, this integer feature is then also assigned the bool data type (overruling the integer data type).

Now that we have detected our bool features, we would also like to do this for the four remaining data types. However, contrary to one of the solutions mention before in chapter 3, we want to overcome the problem of checking every single value in a feature. As such, we have implemented a solution that checks only 100 random indices in a feature and for each of these 100 values tries to determine the data type. Similar to the existing solution, a majority vote then decides on the data type of the entire feature.

Let us continue with date data types. Dates can be written in various notations and to be able to deal with this, we have created the following rules to check for this data type. If the first criterion, as well as one of the last two criteria is true, a value is considered to be a date. The last two criteria represent the fact that dates can have various different notations in different parts of the world.

- A value can only have a maximum length of ten symbols.
- A value needs to have either a dash symbol '-' or a slash symbol '/' at positions 3 and 6 in the value.
- A value needs to have either a dash symbol '-' or a slash symbol '/' at positions 5 and 8 in the value.

Floats and integers, on the other hand, are numbers and detecting numbers can be easily done by trying to convert these values into integers with the particular integer conversion function int(). When this is possible, we know that the value should be a number and by checking this number for the occurrence of a dot (.) we can differentiate between integers and floats.

Lastly, the string data type represents basically everything that remains. When a value is not considered either a bool, a date, a float or an integer, the value is assigned the string data type. This choice has been made, because the string data type can represent many different things, including words, sentences and even numbers.

After the function is applied to the test data set from table 4.1 we get the following results, together with the actual ground truth that has been created by checking the test data set manually. As can be seen, both the 'Position' and 'Streams' features are inferred as integers, while the 'Date' column ins inferred as a date and the remaining columns are inferred as the string data type, as should be the case.

```
Date: 0, Float64: 0, Int64: 0, String: 100
Date: 0, Float64: 0, Int64: 0, String: 100
Date: 0, Float64: 0, Int64: 0, String: 100
Date: 100, Float64: 0, Int64: 0, String: 0
Date: 0, Float64: 0, Int64: 0, String: 100

Predicted data types:
 ['int64', 'string', 'string', 'int64', 'string', 'date', 'string']

Provided ground truth:
 ['int64', 'string', 'string', 'int64', 'string', 'date', 'string']
```

4.2 Numeric, Categorical or Ordinal Data?

After we have found out which feature has which data type, it is time to move on to the next step. In this step we want to decide whether a feature contains categorical or ordinal data, or if the feature just represents values that have no relation to each other. To check for these special cases, we only take into consideration features that have either the string or the integer data type. This seems obvious, since bool data types are already categorical and cannot be ordinal (which is the definition of a bool), the date data type simply represents dates (which can be considered both categorical and ordinal, but are a kind of special case) and the float data type is too specific to be categorical or ordinal (why would the author of the data set have picked floating point numbers over integers, if these decimals do not matter in the end?). Thus, in our solution only features of the string or integer data type can be categorical or ordinal and both the string and the integer data type have their own function to decide on this matter.

Let us start with features of the string data type, which seems to be easier to understand from the human perspective. The idea is that a provided feature is checked for the number of unique values, as well as the relationship between these unique values. This represents the way in which we human beings try to determine if a list of values (a feature from the data set) contains categorical data. First, we scan through the list of values and unknowingly find the unique values in the list and if this number of unique values seems 'decent' we try to relate all of the unique values to each other. If we can find a relationship between these unique values, we consider them as categories. Similarly, the function we have created that checks whether or not a string feature is categorical takes as an input the raw feature and a user-defined parameter k and then performs these same two steps as humans do.

First of all, if a string feature contains less than or equal to 25 unique values, the feature is considered to be categorical immediately. A small number of unique strings means that the possible values that can be inserted in the feature are limited, most of the time indicating that these strings represent categories. However, when the provided feature contains more than 25 unique values, then we move on to the second human approach: relations.

When a string feature has more than 25 unique values, but less than or equal to k unique values, we would like to know how well all of these unique values are related. To find out, we introduce natural language processing and more particularly, the *Wu-Palmer algorithm* [3]. For every unique string in the feature we try to find the representation of that string in the *WordNet* library [10] (from the NLTK corpus) and compare this representation with a representation for every other unique string. That is, we calculate a similarity score between these representations, resulting in $n-1+n-2+...+1 = \frac{(n-1)n}{2}$ similarity scores. After that, we take the mean of these similarity scores, which needs to be bigger than the critical value of 0.5 for a string feature to be considered categorical. This critical value of 0.5 has been chosen to represent the fact that these categories have at least some relationship with each other, with a lower score than 0.5 being closer to 0, indicating no relationship at all. But how are these similarity scores calculated?

WordNet uses the NLTK corpus reader, containing synsets, which are sets of synonyms that share a common meaning. Comparing these synsets (words) can be done by calculating a similarity score between these words. WordNet first tries to find the most comming meaning for a word that is provided and then, as mentioned before, uses the Wu-Palmer algorithm, which uses the weights of the edges of the synsets in the hierarchy. What the Wu-Palmer algorithm basically does, is that it "calculates relatedness by considering the depths of the two synsets in the WordNet taxonomies, along with the depth of the LCS (Least Common Subsumer)" (taken from [2]). Figure 4.1 (adapted from [7]) shows this approach in more detail, with the formula used for this being:

$$score = \frac{2*depth(LCS)}{depth(synset1)+depth(synset2)}$$

The output is a score between 0 and 1, indicating how well these two words are related.

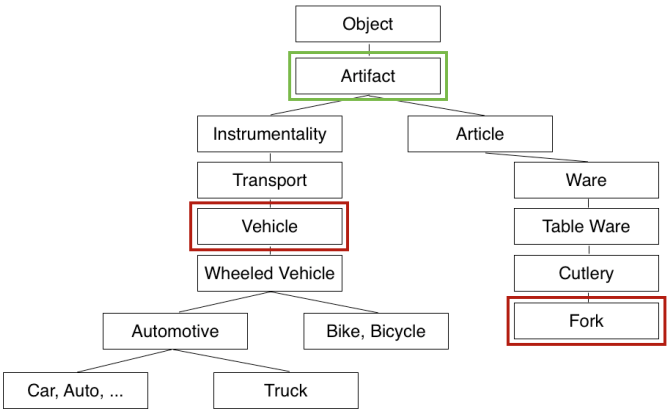

Figure 4.1: Wu-Palmer algorithm with LCS in green for synsets (words) in red

Deciding whether a string feature is categorical is already quite hard, even for humans, but deciding whether or not an integer feature is categorical is even more difficult, since there is no way to measure 'relationship' between numbers, because they have no meaning. A solution to this problem is calculating a distance score for these integers, but let us start from the beginning again. When an integer features contains less than or equal to 10 unique values, this feature is considered categorical. Numbers can represent anything, but if there are only a maximum of 10 unique values in a feature, this usually indicates that we are dealing with categories. Now, if an integer feature contains more than 10 unique values, but less than the user-defined k unique values, we have to find out some sort of 'relationship' between these numbers, similar to the string feature. A logical form of relationship between numbers would be to calculate a distance score for these numbers and that is what we have implemented. Again, for every combination of unique numbers $\frac{(n-1)n}{2}$ times, we calculate a distance between these two numbers based on the following very simple formula:

$$score = |number1 - number2|$$

The result is the absolute distance between the two numbers and calculating the mean for all these distance scores gives us the average absolute distance for the feature, based on all unique values in the feature. If this average score is smaller than or equal to the average of all unique values, than the feature is considered to be categorical as well. By taking the average of all values as the critical value, we make sure that all of the unique integers in a feature have a relatively close distance to each other, indicating these numbers represent categories.

After the function is applied to the test data set from table 4.1 we get the following results, together with the actual ground truth that has been created by checking the test data set manually. Note that the scores being returned are calculated for all features that are inferred in the previous step as either string or integer. We can see that in this case the 'Region' column is predicted as non-categorical (just a regular string), while it should actually be predicted as categorical, because the calculated similarity score is too low (0.38, which should be 0.5).

```
Predicted data types:
 ['int64', 'string', 'string', 'int64', 'string', 'date', 'string']

Provided ground truth:
 ['int64', 'string', 'string', 'int64', 'string', 'date', 'cat_string']
```

Note that the proposed solutions only detect categorical data and do not consider ordinal data. Both chapters 6 and 7 will discuss why this is the case and why no working solutions have been implemented to deal with this particular problem.

4.3 Auto-Selecting Encoding Techniques

After determining both which data type each feature has and which of these features are categorical, we would like to propose an encoding for the raw data set. However, since we do not know about any further plans the user has with the encoded data set, we have created a function that has several parameters that can be set by the user to have more control over the output.

Let us first recap a bit: we have discovered bool, date, float, integer and string data types and have decided on both integer categorical features and string categorical features. For each of these data types we propose a specific encoding.

Categorical features, that is categorical string features and categorical integer features, should be *one-hot encoded* to adequately represent the different categories of the feature. As mentioned before, one-hot encoding transforms a categorical feature into as many different features as there are unique values (categories) in the categorical feature, with a 1 representing a row belonging to that particular category. The function we have created for the categorical features has two optional parameters, that can be set by the user:

- numberOfOccurences (default=infinite)
- other (default=False)

The first parameter, *numberOfOccurences*, implies the number of times a unique value in a categorical feature has to occur to be represented as a feature. That is, if a certain unique value occurs less than this parameters value times, no feature for this particular category will be created during the one-hot encoding process.

The second parameter has a direct relationship to the first one, indicating whether or not those unique values (categories) that will not get their own column due to their number of occurrences, should be represented in a column named 'Other'. If this parameter is true, all categories with a number of occurrences smaller than the previous parameter will be combined and represented by one single feature, making sure there is no loss of data.

Now that we have encoded the categorical features, we also want to encode the non-categorical ones. For the integer and float features this is simple: these features are already represented by numbers and as such, need no further treatment. The other data types however (bool, date, string) have two options: either *factorize* all of these features, that is turn them into numbers with every unique value being represented by a unique number, or keep them as they are. Factorizing the remaining features results in an all-numeric encoding, while it might be beneficial to keep for example the dates in their original notation.

Once more, after the function is applied to the test data set from table 4.1 we get the encoding as shown for the first five rows in table 4.2. Note that the 'Position' and 'Streams' features were predicted as integers and as such, remain untouched. However, the other columns have been predicted as strings and date respectively and therefore these features are factorized.

Table 4.2: Encoding for the Spotify song rankings test data set

Position	**Track Name**	**Artist**	**Streams**	**Date**	**Region**
1	0	0	19272	0	0
2	1	1	19270	0	0
3	2	2	15761	0	0
4	3	3	14954	0	0
5	4	4	14269	0	0

Chapter 5

Results

As expected, similar to the previous chapters, this chapter covers all three aspects of the data encoding bot, showing results for the proposed solutions. The predicted values for each of the steps, as well as the ground truths can be found in appendix A.

5.1 Auto-Detecting Data Types

The first step of the process can be measured by comparing the number of correct predictions with the total number of features. Note that, because we are dealing with raw data, checking for the number of correct predictions implies comparing the predicted data types with the ground truth. However, dirty data does not contain any ground truth and as such, this has to be done manually. Since this is a tedious work, we have tested our solution on ten different, random data sets, mostly collect from *Kaggle* (eight out of ten). Table 5.1 shows for each of these data sets the accuracy score, as well as the number of features that has been checked. As can be seen, the proposed solution made zero mistakes in all ten different test data sets, indicating that the first step of the process works flawless based on this test data.

Table 5.1: Results for step 1, data type detection.

Data Set	**Accuracy (%)**	**Features Checked**
Solar Panel Energy Production Eindhoven	100.0	7
Weather Measurements Eindhoven Airport	100.0	9
TED Talks	100.0	17
Wine Reviews	100.0	11
Fake News	100.0	20
Electronic Music	100.0	14
Crypto Currency	100.0	13
Google Job Skills	100.0	7
Spotify Song Rankings	100.0	7
Kickstarter Projects	100.0	13

5.2 Numeric, Categorical or Ordinal Data?

Similar to the previous step, the second step of the process can be measured by comparing the number of correct predictions with the number of checked features. Note that we do not take into account the total number of features here, because we do not take into account any feature that has the bool, date or float data type in this step. Again, because we are dealing with raw data, checking for the number of correct predictions implies comparing the predicted data types with the ground truth. As such, we have tested our solution on the same ten data sets as in the previous step. Table 5.2 shows for each of these data sets the accuracy score, as well as the number of features that has been checked, based on a predefined value *k=100* (which was the maximum number of unique values for a feature to be allowed for categorical data). By looking at the table we can see that this step of the process performs quite well, but does make the occasional mistake. One of the reasons for this, is that the detection of categorical strings is only based on words that can be found in the corpus, such that a lot of categories created in the raw data sets cannot be used to calculate a similarity score for, resulting in incorrect predictions.

Table 5.2: Results for step 2, categorical data checking, based on a user-specified value *k=100*.

Data Set	**Accuracy (%)**	**Features Checked**
Solar Panel Energy Production Eindhoven	100.00	3
Weather Measurements Eindhoven Airport	100.00	2
TED Talks	88.24	17
Wine Reviews	70.00	10
Fake News	94.44	18
Electronic Music	90.91	11
Crypto Currency	100.00	6
Google Job Skills	100.00	6
Spotify Song Rankings	83.33	6
Kickstarter Projects	90.00	10

5.3 Auto-Selecting Encoding Techniques

There is no clear way to measure the efficiency of a created encoding and only the person that is going to use the encoding for further processing can determine how 'optimal' the particular encoding is. For this reason, several parameters have been created to make sure it is possible to adjust the encoding that is created to fit the aims and needs of the user. Possible outcomes are:

- An encoding that one-hot encodes all categorical features and factorizes all non-categorical bool, date and string features.
- An encoding that one-hot encodes values in a categorical feature that occur more than a predefined number of times and factorizes all non-categorical bool, date and string features.
- An encoding that one-hot encodes values in a categorical feature that occur more than a predefined number of times, creates an 'Other' feature for the remaining values in the particular feature and factorizes all non-categorical bool, date and string features.
- An encoding that one-hot encodes all categorical features and leaves all other features intact.
- An encoding that one-hot encodes values in a categorical feature that occur more than a predefined number of times and leaves all other features intact.
- An encoding that one-hot encodes values in a categorical feature that occur more than a predefined number of times, creates an 'Other' feature for the remaining values in the particular feature and leaves all other features intact.

Chapter 6

Conclusion

After having performed this project, it is clear that even such a topic that looks very easy on the eye, takes a lot of time and effort to perfectly complete. Furthermore, Machine Learning is an area that is under constant development and it is such a new field of interest, that there are no 'perfect' solutions just yet.

Personally, given the time constraints that we had (7.5 weeks spread over four courses), it is hard to determine what more could have been done. Much easier it is to determine what could have been done better or more efficiently. Overall, when looking back at the actual research questions, we can say that we have found a solution to the first problem, we have partially found a solution to the second problem and we have partially found a solution to the third problem.

The first step of the process has worked out very well, with results above expectation for all of the test data sets. The proposed solution to step 1 was relatively easy to implement, since it is a combination of several existing approaches, combined with a human approach of detecting data types.

One of the major problems with this research has been trying to find a solution to the second part, detecting categorical and ordinal data. As a first solution, we basically tried applying Machine Learning to be able to apply Machine Learning. That is, Machine Learning needs clean data as an input and simply cannot deal with dirty data just yet. However, we tried to train a Machine Learning model on this dirty data, to prepare this dirty data for the actual Machine Learning: Machine Learning for Machine Learning. Unfortunately, detecting categorical and ordinal features with this method proved to be very hard, because of multiple problems: training data for this cause is very rare to find, creating this training data is even harder and more time-consuming and the true difference between categorical and ordinal data only has a very narrow separation, which even human beings cannot distinguish in most cases.

Even though the results for the second step are reasonably good as well, the approach is based on the human concept of defining categorical data, instead of a Machine Learning approach. Also, we have been unable to detect ordinal data, which is a major downside of this project.

The final step creates and encoding based on several parameters that can be defined by the user. It is then up to the user to decide how 'optimal' the created encoding is, based on his consequent actions with the created encoding. It is impossible for us to decide how well the resulting encoding is, since there are no clear metrics for the sake of this.

Chapter 7

Discussion

There are several points of improvement for this project and a lot of further research can be performed in this direction. Let us take a look at the 'flaws' of this project in more detail for every step of the process.

Step 1 has shown to give us the best results, but still there are several points of improvement. First of all, dates that are represented as numbers, such as formats where date and time are being combined, cannot be detected, because it is unknown whether a value represents a date, or an actual number.

Also, dates can only be recognized when their day and month indicators are represented by two digits instead of just one. That is, when there is a day or month with numbers between 1 to 9, this number should be represented as 01 to 09 respectively (so with two digits) for the function to be able to detects this as a date.

Step 2 has been a major difficulty to correctly predict, as mentioned before in chapter 6. Several improvements can be made, starting with the inclusion of Machine Learning. Unfortunately, we were unable to apply any Machine Learning algorithm to train a model to predict whether features are categorical or ordinal and as such, a whole lot of further research can be performed in this area.

Concerning our own implementation, there are several aspects that could have been worked out more carefully. The first problem is that determining the ground truth of a raw data set can already be hard for human beings in a lot of cases and as such, testing for an accuracy score in this step becomes quite subjective (since the ground truth is our personal idea of what the ground truth should be for a particular data set).

As mentioned before, another problem is that we are dealing with raw data, which means that the used approach for categorical string detection fails when words, or certain strings, are being used that are not part of the common language. Further research should be performed in this field to make sure that also user-defined strings will be recognized as categorical.

One of the aspects that is easiest to investigate, but takes a considerable amount of time, is to check the influence of all of the critical values used. As stated in chapter 4 on the solution we have implemented, we use a critical value for strings, a critical value for integers, a user-defined critical value (determining the maximum number of unique values), as well as a critical value for similarity for strings and distance for integers (determining the relationship). All of these values have only been looked into minimally and need further research to optimize.

Lastly, finding an order in raw data is even harder than finding categories for human beings, meaning that a lot of further research has to be done on detecting ordinal data, as well as distinguishing categorical and ordinal data.

Bibliography

[1] Pieter Gijsbers. Arff-tools, 2017. https://github.com/openml/ARFF-tools/blob/master/csv-to-arff.py. 1, 7

[2] Sagar Gole. Words similarity/relatedness using wupalmer algorithm, 2015. http://blog.thedigitalgroup.com/sagarg/2015/06/10/words-similarityrelatedness-using-wupalmer-algorithm/. 11

[3] Daniel Jurafsky and James H. Martin. *Speech and Language Processing (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2009. 11

[4] Will McGinnis. Beyond one-hot: An exploration of categorical variables, 2015. http://www.willmcginnis.com/2015/11/29/beyond-one-hot-an-exploration-of-categorical-variables/. 8

[5] NumFocus. Pandas, Latest release: october 2017. https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html. 1, 6, 7

[6] The Open Knowledge Foundation Ltd. (okfn). messytables, 2013. http://messytables.readthedocs.io/en/latest/. 1, 5

[7] Troy Simpson and Thanh Dao. Wordnet-based semantic similarity measurement, 2016. https://www.codeproject.com/Articles/11835/WordNet-based-semantic-similarity-measurement. 11

[8] Isabel Valera and Zoubin Ghahramani. Automatic discovery of the statistical types of variables in a dataset. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3521–3529, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. 1, 7, 8

[9] Joaquin Vanschoren. Auto-clean - auto-ml, 2016. https://canvas.tue.nl/courses/3121/files/folder/example%20assignments?preview=456756. 1

[10] Princeton University "About WordNet." WordNet. Princeton university, 2010. http://wordnet.princeton.edu. 11

Appendix A

Predictions & Ground Truths

```
predicted_energy_step_1 =
['date', 'string', 'int64', 'float64', 'float64', 'int64', 'float64']

ground_truth_energy_step_1 =
['date', 'string', 'int64', 'float64', 'float64', 'int64', 'float64']

predicted_energy_step_2 =
['date', 'string', 'int64', 'float64', 'float64', 'int64', 'float64']

ground_truth_energy_step_2 =
['date', 'string', 'int64', 'float64', 'float64', 'int64', 'float64']
```

```
predicted_weather_step_1 =
['string', 'float64', 'float64', 'float64', 'float64', 'string', 'float64', '
    float64', 'float64']

ground_truth_weather_step_1 =
['string', 'float64', 'float64', 'float64', 'float64', 'string', 'float64', '
    float64', 'float64']

predicted_weather_step_2 =
['string', 'float64', 'float64', 'float64', 'float64', 'cat_string', 'float64', '
    float64', 'float64']

ground_truth_weather_step_2 =
['string', 'float64', 'float64','float64', 'float64', 'cat_string', 'float64', '
    float64', 'float64']
```

```
predicted_TED_step_1 =
['int64', 'string', 'int64', 'string', 'int64', 'int64', 'string', 'string', 'int64
    ', 'int64', 'string', 'string', 'string', 'string', 'string', 'string', 'int64'
    ]

ground_truth_TED_step_1 =
['int64', 'string', 'int64', 'string', 'int64', 'int64', 'string', 'string', 'int64
    ', 'int64', 'string', 'string', 'string', 'string', 'string', 'string', 'int64'
    ]

predicted_TED_step_2 =
['int64', 'string', 'int64', 'string', 'int64', 'cat_int64', 'string', 'string', '
    cat_int64', 'int64', 'string', 'string', 'string', 'string', 'string', 'string'
    , 'int64']

ground_truth_TED_step_2 =
['int64', 'string', 'int64', 'cat_string', 'int64', 'int64', 'string', 'string', '
    cat_int64', 'int64', 'string', 'string', 'cat_string', 'string', 'string', '
    string', 'int64']
```

```
predicted_wine_step_1 =
['int64', 'string', 'string', 'string', 'int64', 'float64', 'string', 'string', '
    string', 'string', 'string']

ground_truth_wine_step_1 =
['int64', 'string', 'string', 'string', 'int64', 'float64', 'string', 'string', '
    string', 'string', 'string']

predicted_wine_step_2 =
['int64', 'cat_string', 'string', 'string', 'cat_int64', 'float64', 'string', '
    string', 'cat_string', 'string', 'string']

ground_truth_wine_step_2 =
['int64', 'cat_string', 'string', 'string', 'cat_int64', 'float64', 'cat_string', '
    string', 'string', 'cat_string', 'string']
```

```
predicted_fake_step_1 =
['string', 'int64', 'string', 'string', 'string', 'string', 'string', 'string', '
    string', 'string', 'float64', 'string', 'float64', 'string', 'int64', 'int64',
    'int64', 'int64', 'int64', 'string']

ground_truth_fake_step_1 =
['string', 'int64', 'string', 'string', 'string', 'string', 'string', 'string', '
    string', 'string', 'float64', 'string', 'float64', 'string', 'int64', 'int64',
    'int64', 'int64', 'int64', 'string']

predicted_fake_step_2 =
['string', 'cat_int64', 'string', 'string', 'string', 'string', 'cat_string', '
    string', 'string', 'cat_string', 'float64', 'string', 'float64', 'string', '
    int64', 'int64', 'int64', 'int64', 'int64', 'cat_string']

ground_truth_fake_step_2 =
['string', 'cat_int64', 'string', 'string', 'string', 'string', 'cat_string', '
    string', 'string', 'cat_string', 'float64', 'string', 'float64', 'string', '
    int64', 'int64', 'int64', 'int64', 'int64', 'cat_string']
```

```
predicted_music_step_1 =
['int64', 'bool', 'string', 'string', 'string', 'string', 'int64', 'string', 'int64
    ', 'float64', 'date', 'string', 'string', 'string']

ground_truth_music_step_1 =
['int64', 'bool', 'string', 'string', 'string', 'string', 'int64', 'string', 'int64
    ', 'float64', 'date', 'string', 'string', 'string']

predicted_music_step_2 =
['int64', 'bool', 'string', 'string', 'string', 'cat_string', 'cat_int64', 'string'
    , 'int64', 'float64', 'date', 'string', 'string', 'string']

ground_truth_music_step_2 =
['int64', 'bool', 'string', 'string', 'string', 'cat_string', 'cat_int64', '
    cat_string', 'int64', 'float64', 'date', 'string', 'string', 'string']
```

```
predicted_crypto_step_1 =
['string', 'string', 'string', 'date', 'int64', 'float64', 'float64', 'float64', '
    float64', 'int64', 'int64', 'float64', 'float64']

ground_truth_crypto_step_1 =
['string', 'string', 'string', 'date', 'int64', 'float64', 'float64', 'float64', '
    float64', 'int64', 'int64', 'float64', 'float64']

predicted_crypto_step_2 =
['string', 'string', 'string', 'date', 'int64', 'float64', 'float64', 'float64', '
    float64', 'int64', 'int64', 'float64', 'float64']

ground_truth_crypto_step_2 =
['string', 'string', 'string', 'date', 'int64', 'float64', 'float64', 'float64', '
    float64', 'int64', 'int64', 'float64', 'float64']
```

```
predicted_job_step_1 =
['bool', 'string', 'string', 'string', 'string', 'string', 'string']

ground_truth_job_step_1 =
['bool', 'string', 'string', 'string', 'string', 'string', 'string']

predicted_job_step_2 =
['bool', 'string', 'cat_string', 'cat_string', 'string', 'string', 'string']

ground_truth_job_step_2 =
['bool', 'string', 'cat_string', 'cat_string', 'string', 'string', 'string']
```

```
predicted_spotify_step_1 =
['int64', 'string', 'string', 'int64', 'string', 'date', 'string']

ground_truth_spotify_step_1 =
['int64', 'string', 'string', 'int64', 'string', 'date', 'string']

predicted_spotify_step_2 =
['int64', 'string', 'string', 'int64', 'string', 'date', 'string']

ground_truth_spotify_step_2 =
['int64', 'string', 'string', 'int64', 'string', 'date', 'cat_string']
```

```
predicted_kickstarter_step_1 =
['int64', 'string', 'string', 'string', 'string', 'string', 'float64', 'string', '
    float64', 'string', 'int64', 'string', 'float64']

ground_truth_kickstarter_step_1 =
'int64', 'string', 'string', 'string', 'string', 'string', 'float64', 'string', '
    float64', 'string', 'int64', 'string', 'float64']

predicted_kickstarter_step_2 =
['int64', 'string', 'string', 'cat_string', 'cat_string', 'string', 'float64', '
    string', 'float64', 'cat_string', 'int64', 'cat_string', 'float64']

ground_truth_kickstarter_step_2 =
['int64', 'string', 'cat_string', 'cat_string', 'cat_string', 'string', 'float64',
    'string', 'float64', 'cat_string', 'int64', 'cat_string', 'float64']
```

Appendix B

Code Implementation

```
# Make sure to also output the intermediary steps
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

```
# Import all relevant libraries
%matplotlib inline
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import random

from nltk.corpus import wordnet as wn
```

Step 0: Import the data

```
#raw_data = pd.read_csv("ENTER_YOUR_CSV_FILE_NAME_HERE.csv")
raw_data = pd.read_csv("Data/kickstarter.csv") #Example

# Now simply run the entire Jupyter notebook!

raw_data.head()

print ("The provided data set consists of",raw_data.shape[0],"rows and",raw_data.
    shape[1],"columns (features.")
```

Step 1: Auto-Detecting Data Types

```
# Function to automatically infer data types for a specific feature that has the
    standard 'object' data type
# Data types that we want to infer: boolean, date, float, integer, string
# Note that every feature that is not either a boolean, a date, a float or an
    integer, is classified as a string
# Input: Pandas Dataframe consisting of one single feature (so n*1 in size)
# Output: Data type of the feature (in string format)

def autoInferObject(raw_data_feature):
    dataType = ""
    types = ["date","float64","int64","string"]
    weights = [0,0,0,0] #Weights corresponding to the data types

    featureLength = len(raw_data_feature)

    numberOfIndices = 100 #Number of different values to check in a feature
    randomIndices = random.sample(range(0,featureLength), min(numberOfIndices,
        featureLength)) #Array of random indices

    # If the feature only contains two different unique values, then infer it as
        boolean
    if len(pd.unique(raw_data_feature)) == 2:
        dataType = "bool"
    else:
        for i in randomIndices:
            try:
                if (len(raw_data_feature[i]) <= 10
                    and (((raw_data_feature[i][2:3]=='-' or raw_data_feature[i][2:3
                        ]=='/')
                    and (raw_data_feature[i][5:6]=='-' or raw_data_feature[i][5:6
                        ]=='/'))
                    or ((raw_data_feature[i][4:5]=='-' or raw_data_feature[i][4:5
                        ]=='/')
                    and (raw_data_feature[i][7:8]=='-' or raw_data_feature[i][7:8
                        ]=='/')))):
                    weights[0] += 1 #Date
                else:
                    weights[3] += 1 #String
            except (TypeError,ValueError,IndexError):
                try:
                    int(raw_data_feature[i])
                    if ('.' in str(raw_data_feature[i])):
                        weights[1] += 1 #Float
                    else:
                        weights[2] += 1 #Integer
                except (TypeError,ValueError,IndexError):
                    weights[3] += 1 #String

        #print ("Date: {}, Float64: {}, Int64: {}, String: {}".format(weights[0],
            weights[1],weights[2],weights[3])) #For debugging purposes
        dataType = types[weights.index(max(weights))]

    return dataType
```

```
# Function to automatically infer data types for every single feature in a raw data
    set
# Input: Pandas Dataframe created directly from the raw data with the pd.read_csv
    function
# Output: List of data types, one data type for each feature

def autoDetectDataTypes(raw_data):
    result = []

    for column in raw_data:
```

```
        if (raw_data.dtypes[column] == "object"):
            #print ("Trying to automatically infer the data type of the",column,"
                feature...") #For debugging purposes
            inferredType = autoInferObject(raw_data[column])
            result.append(inferredType)
            #print ("Result:",inferredType) #For debugging purposes
        elif (raw_data.dtypes[column] == "int64"):
            if (len(pd.unique(raw_data[column])) == 2):
                result.append("bool")
            else:
                result.append("int64")
        else:
            # The only remaining data type is 'float64', which needs no special
                treatment
            result.append("float64")

    return result
```

```
predicted = autoDetectDataTypes(raw_data)
print ("\nPredicted data types:\n",predicted)
```

```
# Manually check for the ground truth, since a dirty data set has no ground truth
    included

#ground_truth = ["ENTER_GROUND_TRUTH_DATA_TYPES_FOR_THE_PROVIDED_DATA_SET"]
ground_truth = ['int64', 'string', 'string', 'string', 'string', 'string', 'float64
    ', \
                'string', 'float64', 'string', 'int64', 'string', 'float64'] #
                    Example

print ("Provided ground truth:\n",ground_truth)
```

```
# Function to calculate an accuracy score for the implemented solution
# Input: Array containing the (self-made) ground truth
#        Array containing the predicted data types
# Output: Accuracy score based on the number of correct predictions

def score(ground_truth, predicted):
    correctPredictions = 0

    for i in range(0,len(ground_truth)):
        if ground_truth[i] == predicted[i]:
            correctPredictions += 1

    return correctPredictions / len(ground_truth)
```

```
print ("Accuracy:",score(ground_truth, predicted)*100,"%\nNumber of features
    checked:",len(predicted))
```

```
scoreResults = [("Solar Panel Energy Production Eindhoven",100.00,7,"https://canvas
    .tue.nl/files/508283"), \
                ("Weather Measurements Eindhoven Airport",100.00,9,"https://canvas.
                    tue.nl/files/508283"), \
                ("TED Talks",100.00,17,"https://www.kaggle.com/rounakbanik/ted-
                    talks"), \
                ("Wine Reviews",100.00,11,"https://www.kaggle.com/zynicide/wine-
                    reviews"), \
                ("Fake News",100.00,20,"https://www.kaggle.com/mrisdal/fake-news"),
                    \
                ("Electronic Music",100.00,14,"https://www.kaggle.com/marcschroeder
                    /17-years-of-resident-advisor-reviews"), \
                ("Crypto Currency",100.00,13,"https://www.kaggle.com/jessevent/all-
                    crypto-currencies"), \
                ("Google Job Skills",100.00,7,"https://www.kaggle.com/niyamatalmass
                    /google-job-skills"), \
```

```
                ("Spotify Song Ranking",100.00,7,"https://www.kaggle.com/edumucelli
                    /spotifys-worldwide-daily-song-ranking"), \
                ("Kickstarter Projects",100.00,13,"https://www.kaggle.com/kemical/
                    kickstarter-projects")]
pd.DataFrame(scoreResults, columns=["Data Set", "Accuracy", "Features Checked", "
    Retrieved From"])

print ("Results for step 1, data type detection.")
```

Step 2: Numeric, Categorical or Ordinal Data?

```
# Function to check if a feature contains categorical data, specifically concerning
    strings
# If a feature contains at most 25 unique elements, this feature is always
    considered as categorical
# If a feature contains between 26 and k (where k is user specified) unique values,
    then the function
# calculates a similarity score between all of the different values. If this score
    is higher than 0.70
# a feature is also considered to be categorical (since the values at least have
    some relationship).
# Input: Pandas Dataframe consisting of one single feature (so n*1 in size)
#        A user-determined value k (the critical value: more than k unique values
    cannot be categorical)
# Output: A boolean stating whether the supplied feature is categorical or not

def autoCheckCategoricalString(raw_data_feature, k=100):
    categorical = False

    allWords = pd.unique(raw_data_feature) #All unique strings in the feature
    similarityScores = []

    if (len(allWords) <= 25): #If there are less than 25 unique strings, it is
        categorical
        categorical = True
        #print ("Less than or equal to 25 unique strings (in this case",len(
            allWords),")") #For debugging purposes

    elif (len(allWords) <= k): #Else if there are less than k unique strings, check
         for similarity

        for i in range(0,len(allWords)-1):
            for j in range(i+1,len(allWords)):
                if (pd.isnull(allWords[i])): #If a string has no value (NaN), turn
                    it into some nonsense
                    allWords[i] = "abcdef"
                elif (pd.isnull(allWords[j])):
                    allWords[j] = "abcdef"

                word_1 = wn.synsets(allWords[i])
                word_2 = wn.synsets(allWords[j])

                if (word_1 != [] and word_2 != []): #Calculate similarity between
                    two non-empty words
                    similarity = wn.wup_similarity(word_1[0], word_2[0])
                    #print ("Similarity between",word_1[0],"and",word_2[0],":",
                        similarity) #For debugging purposes
                    if (not pd.isnull(similarity)):
                        similarityScores.append(similarity)

        #print ("Similarity Scores:\n",similarityScores) #For debugging purposes
        #print ("Mean Similarity Score:",np.mean(similarityScores)) #For debugging
            purposes

        if (np.mean(similarityScores) > 0.50): #0.50 = Critical similarity value
            categorical = True

    #print ("Categorical Feature?",categorical) #For debugging purposes

    return categorical
```

```
# Function to check if a feature contains categorical data, specifically concerning
    integers
# If a feature contains at most 10 unique elements, this feature is always
    considered as categorical
```

```
# If a feature contains between 11 and k (where k is user specified) unique values,
    then the function
# calculates a distance score between all of the different values. If this score is
    smaller than 0.1
# times the mean off all the integers a feature is also considered to be
    categorical (since the
# values have a relatively similar distance between each other).
# Input: Pandas Dataframe consisting of one single feature (so n*1 in size)
#        A user-determined value k (the critical value: more than k unique values
    cannot be categorical)
# Output: A boolean stating whether the supplied feature is categorical or not

def autoCheckCategoricalInt(raw_data_feature, k=100):
    categorical = False

    allInts = pd.unique(raw_data_feature) #All unique integers in the feature
    distanceScores = []

    if (len(allInts) <= 10): #If there are less tahn 10 unique integers, it is
        categorical
        categorical = True
        #print ("Less than or equal to 10 unique integers (in this case",len(
            allInts),")") #For debugging purposes

    elif (len(allInts) <= k): #Else if there are less than k unique integers, check
         for distance

        for i in range(0,len(allInts)-1):
            for j in range(i+1,len(allInts)):
                distance = abs(allInts[i] - allInts[j]) #Calculate absolute
                    distance between two integers
                #print ("Distance between integer",allInts[i],"and",allInts[j],":",
                    distance) #For debugging purposes
                distanceScores.append(distance)

        #print ("Distance Scores:\n",distanceScores) #For debugging purposes
        #print ("Mean Distance Score:",np.mean(distanceScores),",should be lower
            than:",np.mean(allInts)) #For debugging purposes

        if (np.mean(distanceScores) < (np.mean(allInts))):
            categorical = True

    #print ("Categorical Feature?",categorical) #For debuggin purposes

    return categorical
```

```
# Function to automatically decide on categoricals for every single feature in a
    raw data set
# Input: Pandas Dataframe created directly from the raw data with the pd.read_csv
    function
#        Array of data types, which is the output of step 1: autoDetectDataTypes
#        [Optional] Integer specifying the critical value for which features will
    be checked
#           on categoricals. That is, if the unique number of elements in a feature
     is higher
#           than the critical value, a feature cannot be categorical. (Default
    value is 100)
# Output: List of data types, one data type for each feature

def autoCheckCategoricals(raw_data,predicted,k=100):

    #print ("Checking if any of the features has categorical data...") #For
        debugging purposes

    for j in range(0,len(predicted)):
        if (predicted[j] == 'int64'):
            if (autoCheckCategoricalInt(raw_data.iloc[:,j],k)):
```

```
                predicted[j] = 'cat_int64'
        elif (predicted[j] == 'string'):
            if (autoCheckCategoricalString(raw_data.iloc[:,j],k)):
                predicted[j] = 'cat_string'

    return predicted
```

```
predicted = autoCheckCategoricals(raw_data,predicted,100)
print ("\nPredicted data types:\n",predicted)
```

```
# Manually check for the ground truth, since a dirty data set has no ground truth
    included

#ground_truth = ["ENTER_GROUND_TRUTH_DATA_TYPES_FOR_THE_PROVIDED_DATA_SET"]
ground_truth = ['int64', 'string', 'cat_string', 'cat_string', 'cat_string', '
    string', 'float64', \
                'string', 'float64', 'cat_string', 'int64', 'cat_string', 'float64'
                    ] #Example

print ("Provided ground truth:\n",ground_truth)
```

```
#ground_truth[:] = [x for x in ground_truth if x != 'bool']
#ground_truth[:] = [x for x in ground_truth if x != 'date']
#ground_truth[:] = [x for x in ground_truth if x != 'float64']

#predicted[:] = [x for x in predicted if x != 'bool']
#predicted[:] = [x for x in predicted if x != 'date']
#predicted[:] = [x for x in predicted if x != 'float64']

print ("Accuracy:",score(ground_truth, predicted)*100,"%\nNumber of features
    checked:",len(predicted))
```

```
scoreResults = [("Solar Panel Energy Production Eindhoven",100.00,3,"https://canvas
    .tue.nl/files/508283"), \
                ("Weather Measurements Eindhoven Airport",100.00,2,"https://canvas.
                    tue.nl/files/508283"), \
                ("TED Talks",88.24,17,"https://www.kaggle.com/rounakbanik/ted-talks
                    "), \
                ("Wine Reviews",70.00,10,"https://www.kaggle.com/zynicide/wine-
                    reviews"), \
                ("Fake News",94.44,18,"https://www.kaggle.com/mrisdal/fake-news"),
                    \
                ("Electronic Music",90.91,11,"https://www.kaggle.com/marcschroeder
                    /17-years-of-resident-advisor-reviews"), \
                ("Crypto Currency",100.00,6,"https://www.kaggle.com/jessevent/all-
                    crypto-currencies"), \
                ("Job Skills",100.00,6,"https://www.kaggle.com/niyamatalmass/google
                    -job-skills"), \
                ("Spotify Song Ranking",83.33,6,"https://www.kaggle.com/edumucelli/
                    spotifys-worldwide-daily-song-ranking"), \
                ("Kickstarter Projects",90.00,10,"https://www.kaggle.com/kemical/
                    kickstarter-projects")]
pd.DataFrame(scoreResults, columns=["Data Set", "Accuracy", "Features Checked", "
    Retrieved From"])

print ("Results for step 2, categorical data checking, based on a user-specified k
    value of 100.")
```

Step 3: Auto-Selecting Encoding Techniques

```
# Function to automatically enocde a features from a raw data set
# Input: Pandas Dataframe consisting of one single feature (so n*1 in size)
#        [Optional] A user-determined value numberOfOccurrences, which states the
#           minimum number of occurrences for a value in the provided feature. For
#           example, when numberOfOccurrences equals 3, all unique values that
#           occur less than three times will not be One Hot Encoded and are removed
#           from the data set. (Default is infinite, so all unique values are OHE.)
#        [Optional] A boolean stating whether the removed values should be
    represented
#           in a One Hot Encoded feature 'other'. (Default is False)
# Output: Pandas Dataframe representing the One Hot Encoded feature

def autoEncodeFeature(raw_data_feature, numberOfOccurrences=np.inf, other=False):

    oheFeature = pd.get_dummies(raw_data_feature)

    if (numberOfOccurrences != np.inf):

        dropColumns = []

        for i in range(0, len(oheFeature.columns)):
            column = oheFeature.iloc[:,i].value_counts()

            if (column[1] < numberOfOccurrences):
                dropColumns.append(i)

        if (other):
            otherValues = []
            for j in range(0, len(raw_data_feature)):
                for k in dropColumns:
                    if (oheFeature.iloc[:,k][j] == 1):
                        otherValues.append(1)
                        break
                else:
                    otherValues.append(0)
            oheFeature = pd.concat([oheFeature, pd.DataFrame(otherValues, columns=[
                "Other"])], axis=1)

        oheFeature.drop(oheFeature.columns[dropColumns], axis=1, inplace=True)

    #print ("Number of features after One Hot Encoding:",len(oheFeature.columns)) #
        For debugging purposes

    return oheFeature
```

```
# Function to automatically encode all features in a raw data set
# Input: Pandas Dataframe created directly from the raw data with the pd.read_csv
    function
#        Array of data types, which is the output of step 2: autoCheckCategoricals
#        [Optional] A user-determined value numberOfOccurrences, which states the
#           minimum number of occurrences for a value in the provided feature. For
#           example, when numberOfOccurrences equals 3, all unique values that
#           occur less than three times will not be One Hot Encoded and are removed
#           from the data set. (Default is infinite, so all unique values are OHE.)
#        [Optional] A boolean stating whether the removed values should be
    represented
#           in a One Hot Encoded feature 'other'. (Default is False)
# Output: Pandas Dataframe with all categorical features One Hot Encoded

def autoEncodeFeatures(raw_data, predicted, numberOfOccurrences=np.inf, other=False
    ):

    dropColumns = []
```

```
    for i in range(0, len(predicted)):
        if (predicted[i] == 'cat_string' or predicted[i] == 'cat_int64'):
            dropColumns.append(i)
            raw_data = pd.concat([raw_data,autoEncodeFeature(raw_data.iloc[:,i],
                numberOfOccurrences,other)], axis=1)

    raw_data.drop(raw_data.columns[dropColumns], axis=1, inplace=True)

    return raw_data
```

```
# Function to automatically encode all features in a raw data set
# Exactly similar to 'autoEncodeFeatures', with the only difference that this
    version
# also factorizes the non-categorical string and boolean features, whilst the
    previous
# version keeps the original values within these features.

def autoEncodeFeatures2(raw_data, predicted, numberOfOccurrences=np.inf, other=
    False):

    dropColumns = []

    for i in range(0, len(predicted)):
        if (predicted[i] == 'cat_string' or predicted[i] == 'cat_int64'):
            dropColumns.append(i)
            raw_data = pd.concat([raw_data,autoEncodeFeature(raw_data.iloc[:,i],
                numberOfOccurrences,other)], axis=1)

        elif (predicted[i] == 'string' or predicted[i] == 'bool' or predicted[i] ==
             'date'):
            factorize = pd.factorize(raw_data.iloc[:,i])
            raw_data.iloc[:,i] = factorize[0]

    raw_data.drop(raw_data.columns[dropColumns], axis=1, inplace=True)

    return raw_data
```

Using the Auto-Encoding Bot

```
# Function to create an encoding for an input data set
# Input: Pandas Dataframe created directly from the raw data with the pd.read_csv
    function
#        [Optional] Integer specifying the critical value for which features will
    be checked
#           on categoricals. That is, if the unique number of elements in a feature
     is higher
#           than the critical value, a feature cannot be categorical. (Default
    value is 100)
#        [Optional] A user-determined value numberOfOccurrences, which states the
#           minimum number of occurrences for a value in the provided feature to be
#           One Hot Encoded. For example, when numberOfOccurrences equals 3, all
#           unique values that occur less than three times will not be One Hot
#           Encoded and are removed from the data set. (Default is infinite)
#        [Optional] A boolean stating whether the removed values should be
    represented
#           in a One Hot Encoded feature 'other'. (Default is False)
#        [Optional] A boolean stating whether to use version1 or version2 for the
    encoding
#           step. Version2 factorizes all non-categorical string, boolean and date
    columns,
#           while version1 leaves non-categorical string, boolean and date features
     as they are.
# Output: Pandas Dataframe with all categorical features One Hot Encoded

def theAutoEncodingBot(raw_data, categoricalUniques=100, numberOfOccurrences=np.inf
    , other=False, version2=True):
    predicted = autoDetectDataTypes(raw_data)
    predicted = autoCheckCategoricals(raw_data,predicted,categoricalUniques)
    if (version2==False):
        encoding = autoEncodeFeatures(raw_data,predicted,numberOfOccurrences,other)
    else:
        encoding = autoEncodeFeatures2(raw_data,predicted,numberOfOccurrences,other
            )

    return encoding
```

```
print ("Encoding for the provided data set:")
theAutoEncodingBot(raw_data)
```