From Strings to Data Science

a Practical Framework for Automated String Handling

van Lith, John

Award date:
2021

Department of Mathematics and Computer Science
Data Mining Research Group

From Strings to Data Science: a Practical Framework for Automated String Handling

Master Thesis

John van Lith

Supervisors:
dr. ir. Joaquin Vanschoren
dr. Marcos De Paula Bueno

Assessment Committee Members:
dr. ir. Joaquin Vanschoren
dr. Yulong Pei
dr. Odysseas Papapetrou

version 1.0

Eindhoven, July 2021

Abstract

In machine learning, one of the issues that data scientists face is improving the quality of their datasets. Datasets of high quality are required to ensure that machine learning algorithms perform optimally. Furthermore, many machine learning models require that the input consists exclusively of numerical data, implying that string data be converted to a numerical representation for the models to work as intended. In comparison to numerical data, categorical string data can represent various types of features (e.g., zip codes, names, marital status). Because of this wide variety of string features, it is relatively difficult to automate data cleaning without sacrificing data quality. This thesis focuses on dealing with different varieties of string features to be automatically preprocessed and encoded. As a result, a Python framework[1] is developed that automatically identifies different types of string features, processes them accordingly, and encodes them into numerical representations using a combination of best practices and novel techniques. Evaluation of all components demonstrates promising results that suggest that automated string handling and cleaning is feasible while ensuring high-quality standards. These results indicate that the proposed techniques and this field of research are worth investigating further.

[1]The framework developed during this thesis can be found on GitHub at https://github.com/ml-tue/automated-string-cleaning.

Preface

First and foremost, I would like to thank Joaquin Vanschoren for his supervision and guidance throughout this thesis. His advice helped me steer this work into a suitable direction and grasp the context and possible solutions. Furthermore, I would like to thank Marcos De Paula Bueno, who provided helpful feedback and suggestions, giving me further insights into specific topics. Additionally, I would like to express my gratitude to my colleagues and friends, whose encouragement and feedback kept me motivated. Finally, I would like to express my gratitude to my parents, who have cared for me throughout my entire academic career.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In machine learning, one of the issues that data scientists face is improving the quality of their datasets. Datasets acquired from the real world come from various sources and serve different purposes but are often unrefined (dirty), and their quality might be sub-optimal. These datasets may contain missing values, outliers, and various non-numerical data types such as categorical (string) data. Datasets of high quality are required to ensure that machine learning algorithms perform optimally. The quality of the dataset highly depends on criteria such as validity, accuracy, completeness, consistency, and uniformity [71].

Furthermore, many machine learning models require that the input consist exclusively of numerical data, implying that string data be converted to a numerical representation for the models to work as intended. In comparison to numerical data, categorical string data can represent various features (e.g., zip codes, names, marital status). Each string feature requires specific processing to ensure that the model outputs optimal results. For example, it could be beneficial in terms of interpretability and predictive performance to encode geographical string data as numerical latitude and longitude representations instead of a categorical encoding. In order to increase data quality and handle various string data effectively, data scientists are required to manually preprocess unrefined data using suitable methods, heuristics, and data cleaning applications.

1.1 Motivation

According to a report by CrowdFlower (2016), data scientists can spend up to 60% of their day cleaning data [30]. Data cleaning is required to ensure that potential bias factors, such as missing values or outliers, are addressed. Furthermore, it is required for most machine learning models that string data are correctly represented as numerical values, which may require different strategies for each unique string feature. However, sufficient data cleaning comes at the cost of spending a significant amount of time to ensure that a reasonable quality is achieved. Even though data cleaning tools can reduce some workload, they still require time and effort from their users. Automated data cleaning tools exist; however, such tools are sub-optimal for robustly preprocessing and encoding different string feature types. Consequently, developing a method that attempts to ease and automate data cleaning and string feature handling from data scientists is essential.

1.2 Objective

This thesis is concerned with the automation of handling and processing string data in the cleaning process. We address the problem by developing a Python framework that performs a series of procedures to ensure that various string features in datasets are properly handled. Considering that numerous data formats are available, we focus on developing a framework that aims at preprocessing tabular data.

1.3 Outline

The remainder of this work is presented as follows. In Chapter 2, the problem statement is defined as well as the different challenges and components that need to be addressed and evaluated. In Chapter 3, a broad literature review is provided on all relevant topics that can be used to develop a solution to the problem statement. In Chapter 4, the proposed methodology aimed at addressing the problem statement is established and thoroughly described. In Chapter 5, the proposed methodology is evaluated against different criteria. Finally, Chapter 6 summarizes the problem, discusses the findings, and presents future directions in this subject.

Chapter 2

Problem statement

Surveys have shown that a significant amount of an analysts' time is spent on preparing and cleaning data [30, 46]. Extensive research has been done on optimizing and automating data cleaning steps and its potential to be a feasible solution to the time-consuming nature [18, 75, 84]. There are also tools available that automate a great deal of the data cleaning steps [53, 54, 55, 63, 83]. Current solutions are sub-optimal because automated solutions do not offer specific cleaning steps for various categorical string features in the data. String features refer to categories represented by strings, such as zip codes, names, and addresses. Users are still required to manually recognize, process, and encode string data based on their unique features. Research on this topic is still ongoing and remains relatively open.

This thesis aims to address the following problem:

How can we deal with different varieties of string features in tabular datasets such that they can be automatically preprocessed and encoded?

Given that the scope of machine learning is broad, we decided to limit our scope to cleaning strings in tabular data. To adequately address the problem, it is essential to investigate all sub-challenges that need to be addressed to transform 'dirty' string data into processed, interpretable data. We review relevant current state-of-the-art techniques for each of these components. After a preliminary study, we have discovered four key components that can be used to address the main problem.

Inference of string feature types In general, various data types require different handling and processing. These data types can usually be determined using various techniques. Hence, to handle each string type accordingly, it is essential to have a technique that can recognize different string features in the data. The preprocessing of string data remains fairly general without such a technique, as each string feature is only recognized as a 'standard' string type in programming languages such as Python. Furthermore, since this step most likely occurs at the beginning of the data cleaning process, type inference should be robust against missing values and outlying data types. Therefore, it is essential to research and implement a technique that allows us to achieve string feature inference, preferably using state-of-the-art techniques related to this topic. For this step in the process, we aim at answering the following question:

- *How can we robustly and accurately infer various string features in the data?*

Preprocessing inferred string features String data can often be further preprocessed to remove redundancies and to extract additional information. As each string feature requires different handling, it is necessary to investigate what types of processing and encoding techniques are relevant for the inferred string features. Furthermore, it is also necessary to investigate the potential benefits of these processing steps for each feature. For this step in the process, we aim at answering the following questions:

- *Which processing type has the highest impact on performance?*
- *Which of the inferred string features are the most relevant to process?*

Detecting ordinality in string data A wide, if not infinite, variety of string feature types exist, implying that an inference technique cannot distinguish all of them. If a string feature is not of a known type, we still need to apply an appropriate encoding technique. Thus, for these unknown string feature types, there is a need for a technique that extracts other properties from the set of so-called 'standard' string data whose feature type could not be inferred. In this work, we decided to focus on classifying the ordinality of standard string data by identifying whether a column contains ordered (ordinal) or unordered (nominal) string data. The difference between both is that ordinal data is categorical data where the variables have a natural ordering associated to them (e.g., `cold`, `warm`, `hot`), whereas nominal data is categorical data without a natural ordering (e.g., `red`, `green`, `blue`). Distinguishing string data based on this property allows for assigning a suitable encoding strategy. For this step in the process, we aim at answering the following question:

- *How can we detect the ordinality of a string column, and how well does this method perform?*

Robust and scalable categorical encoding As stated in Chapter 1, most machine learning models require all the input data to be numerical values. As a result, it is relevant to assign proper encoding techniques to each inferred string feature and other string data based on ordinality of the data. Since the data can be of any size and any cardinality, it is important to investigate both commonly used and state-of-the-art encoding techniques to allow for robust and scalable encoding. Furthermore, for ordinal data, it is desired that the natural ordering is maintained after encoding. While this is relatively trivial in a manual data cleaning setting, it is much more challenging to do so in an automated setting. This challenge is related to interpretability, which can play a significant role in assigning the order. For this step in the process, we aim at answering the following questions:

- *How can we encode string data of arbitrary length and cardinality in a robust and scalable manner without sacrificing performance?*
- *How can we determine the natural ordering of entries in ordinal data automatically, how well can this be done, and is there a noticeable difference in terms of performance?*

The developed solution to address the main challenge will be evaluated based on how well it performs compared to currently applied techniques of dealing with string data. Additionally, each sub-component is evaluated using either ground truth values or a control group (e.g., comparing the performance between preprocessing and not preprocessing with a particular technique), whichever is most relevant to the sub-component.

Chapter 3

Literature analysis

There exist literature and techniques covering several aspects of string handling in general or introducing approaches to handle specific string features. In order to automate string handling for automated data cleaning, a system needs to be able to identify and process string feature types, classify ordinality of 'standard' string (i.e., string features of an unknown type) columns, and encode categorical features correctly. This chapter provides a literature review on several topics that are relevant to the problem statement. More specifically, this literature review covers type detection, classification models for ordinality prediction, string categorization and processing, encoding strategies, other relevant data cleaning techniques, and currently existing state-of-the-art data cleaning tools.

3.1 Type detection

This technique refers to stating or predicting the data type of each column for a given dataset. It is essential to review the literature on this technique, as it can potentially allow the automation of data cleaning by automatically determining which cleaning techniques should be applied based on the inferred data types. In the context of our problem, the sub-problem of type inference for simple types, such as integers, floats, and general strings, is solved quite well and would most likely only have to be applied as is. However, it could be possible to adapt the described techniques to address detection of string feature types.

3.1.1 Data type inference (dtype)

The Pandas library contains two methods to detect which type of data is present in a column, namely (`infer_dtype()`) and (`infer_objects()`) [89]. The former method takes a list of values, checks the data type of each value, and returns a corresponding result. Despite its efficiency, it is highly error-prone. For example, if a column consists of ten strings and one integer, it is inferred as 'mixed' by the function, whereas it is more likely for a human observer to conclude that the integer entry is anomalous. The latter method attempts soft conversion of object-dtyped columns, leaving non-object and unconvertible columns unchanged. Soft conversion is the process of finding the most suitable dtype for a given column, whereas hard conversion forces a given column into a given dtype.

3.1.2 Type inference using Probabilistic Finite-State Machines

Ceritli et al. (2020) recently developed a type inference method that makes use of Probabilistic Finite-State Machines (PFSMs) to predict column types [27]. Besides its ability to infer data types of a given column, this approach also allows users to detect missing and anomalous data in each entry by using weighted predictions.

According to Paz (1971) and Rabin (1963), PFSMs are a class of mathematical models that represent a system consisting of a finite number of states, where transitions between states occur w.r.t. probability distributions [76, 81]. PFSMs can be defined as a tuple $A = (\theta, \Sigma, \delta, I, F, T)$, where θ is a finite set of states, Σ is a set of observed symbols, $\delta \subseteq \theta \times \Sigma \times \theta$ is a set of transitions among states w.r.t. observed symbols, $I, F : \theta \to \mathbb{R}^+$ are the initial-state and final-state probabilities respectively, and $T : \delta \to \mathbb{R}^+$ is the transition probabilities for elements of δ. A graphical representation of a PFSM is depicted in Figure 3.1. Note that for each transition, the left value denotes the probability of firing the transition, and the character in brackets represents what is emitted upon firing the corresponding transition.

Furthermore, PFSMs have to adhere to two conditions: (1) the sum of the initial-state probabilities has to be equal to 1, and (2) at each state $q \in \theta$, either a transition is done to state $q' \in \theta$ and emit a symbol $\alpha \in \Sigma$, or the process is stopped at state q without emitting any symbol. Based on the definition and conditions, PFSMs can generate a set of characters Σ^*. Likewise, for a given set of characters, we are now able to calculate the probability for a given PFSM to generate this set of characters [27].

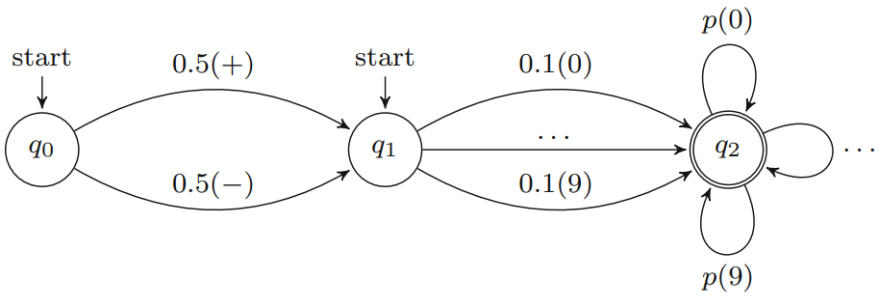


Figure 3.1: Representation of a PFSM with $\theta = \{q_0, q_1, q_2\}$ and $\Sigma = \{+, -, 0, ..., 9\}$ with $p = \frac{1-P_{stop}}{10}$, where P_{stop} represents the stopping probability [27].

Ceritli et al. (2020) developed a mixture of PFSMs that represents data types and missing values, which are used to construct a noisy observation model. This model is a generative model with a set of latent variables that essentially generates a column for each data type that can possibly contain outliers (i.e., entries of other data types) and missing values. To be precise, assuming a column of data $\mathbf{x} = \{x_i\}_{i=1}^N$ is given where each x_i denotes the characters in the i-th row and N is the total number of rows, the generative model consists of latent variables $t \in \{1, 2, .., K\}$ and $\mathbf{z} = \{z_i\}_{i=1}^N$, where t and z_i respectively denote the data type of a column and its i-th row and K is the total number of data types possible. The model has the following generative process:

$$
\begin{aligned}
&\text{column type } t \sim \mathcal{U}(1, K) \\
&\text{row type } z_i = \begin{cases} t & \text{with probability } \pi_t^t \\ m & \text{with probability } \pi_t^m \text{ , } m = \text{missing} \\ a & \text{with probability } \pi_t^a \text{ , } a = \text{anomalous} \end{cases} \\
&\text{row value } x_i \sim p(x_i | z_i)
\end{aligned}
$$

Where $\mathcal{U}$ represents a discrete Uniform distribution, $\pi_t^t, \pi_t^m, \pi_t^a$ represent the probability that a row type is a certain data type given the current data type with $\pi_t^t + \pi_t^m + \pi_t^a = 1$, and $p(x_i|z_i)$ represents the observation model of the generative process.

The inference model that is set up using the generative process is aimed at inferring the column type t, which is cast to the problem of calculating the posterior distribution of t given $\mathbf{x}$, denoted by

$p(t|\mathbf{x})$. It is then assumed that each row is either the same data type as the column type, a missing value, or an anomalous data type. The posterior distribution of column type t is determined as follows:

$$p(t=k|\mathbf{x}) \propto p(t=k)\prod_{i=1}^{N}\Big(\pi_k^k p(x_i|z_i=k)+\pi_k^m p(x_i|z_i=m)+\pi_k^a p(x_i|z_i=a)\Big)$$

Essentially, a posterior probability of each data type is calculated for a given data column, and the data type with the highest posterior probability most likely corresponds to the data type of the given column. After inference of the column type, each entry of $\mathbf{x}$ is also evaluated in terms of how likely it is that they represent missing or anomalous values. This inference is is done by comparing the posterior probabilities for each row type to the posterior probability of the missing and anomalous values as follows:

$$p(z_i=j|t=k,x_i)=\frac{\pi_k^j p(x_i|z_i=j)}{\sum_{\ell\in\{k,m,a\}}\pi_k^\ell p(x_i|z_i=\ell)}$$

In the context of string feature inference, the work of Ceritli et al. can be adapted to also consider specific string feature types in the posterior distribution of column type t. This goal can be achieved by constructing PFSMs based on string feature types and to include them in the generative process of the model.

3.2 Classification models for ordinality detection

Classification tasks exist in most machine learning domains, with the aims varying from e-mail spam detection to object detection in images. In the context of tabular data, classification models are used to predict which class is most likely associated with several characteristics for a given sample. Therefore, in the context of classifying ordinality for standard string columns, it might be relevant to use a classification model to predict whether the standard string column contains ordinal or nominal data. This section describes several classification models that could be used to predict the ordinality of a standard string column.

3.2.1 Statistical type inference

Valera et al. (2017) proposed a Bayesian approach to automatically discover the statistical data type of a column (e.g., ordinal, categorical, or real-valued) [90]. Their approach exploits the key ideas that there is a latent structure in the data that captures statistical dependencies among the different objects and that the observation model for each attribute can be expressed as a mixture of likelihood models. From these two ideas, an efficient Markov chain Monte Carlo (MCMC) inference algorithm was derived to jointly infer both the low-rank representation and the weight of each likelihood model for each attribute in the observed data. This approach would then exploit the latent structure in the data to automatically distinguish among categorical, ordinal, and count data as types of discrete variables. A disadvantage to this method is that it uses the latent distribution of the input data for classification, giving ambiguous results when the distribution of different classes is similar (e.g., some columns in the paper are inferred as "categorical or ordinal"). This ambiguity implies that a domain expert would still have to classify the attributes themselves as either nominal or ordinal, which is not necessarily ideal in an automated setting.

3.2.2 Decision tree learning

Decision tree learning is a simple predictive modeling approach commonly used in statistics, data mining, and (supervised) machine learning. As the name suggests, a decision tree is used as a predictive model to predict a specific target value based on observations. These trees have a flowchart-like structure where each node represents an evaluation on a (set of) attribute(s) of the

data, and each leaf represents a class label. This technique is efficient at picking global features with the most statistical information gain [37], which makes it useful for tabular data containing different statistical types (e.g., discrete categorical variables and continuous numerical variables).

In the context of ordinality detection, a decision tree could be trained to classify whether a column contains ordinal data or not. The decision tree would be trained on a dataset that exclusively consists of nominal and ordinal columns by constructing splits based on the contents of each column. Additionally, a meta-dataset could be constructed based on features of various nominal and ordinal columns, which are then used to train a decision tree to distinguish between the two statistical types accurately. An example of a decision tree that could be constructed based on ordinality features is depicted in Figure 3.2.

A limiting factor of decision tree learning is that the trees tend to be non-robust, meaning that a slight change in the training data can significantly affect the final prediction. Furthermore, decision tree learners are prone to overfitting.

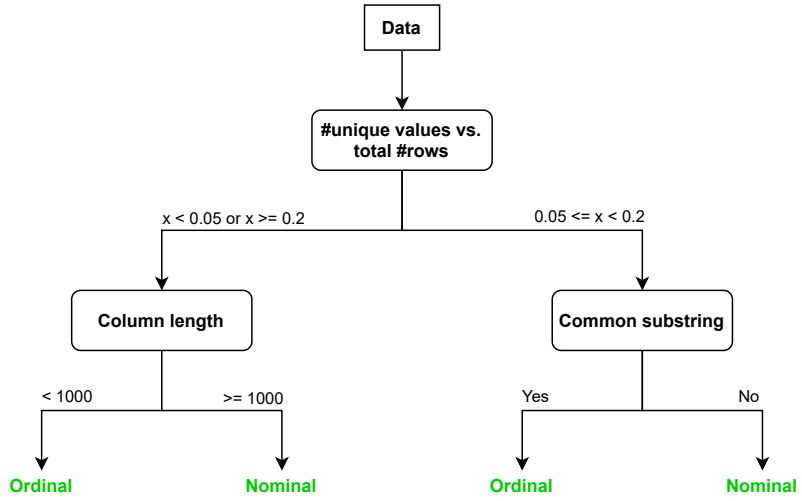


Figure 3.2: An example of a decision tree on determining ordinality based on features of the columns. The decision tree consists of nodes representing features of the data, directed edges representing the evaluation from the feature, and the leaf nodes representing the predicted class label.

3.2.3 Random forest

This is an ensemble learning method for tasks such as classification and regression which makes use of decision trees. During training, multiple decision trees are constructed from subsets of random samples of the data to form the random forest. The random forest predicts the data by taking the mean or mode of the output class labels of all decision trees on the same data. An example of a random forest classifier is depicted in Figure 3.3. An advantage of this strategy is that it counters the overfitting nature of decision trees [33]. In general, random forests outperform decision trees in most settings, but their performance can be affected due to various data characteristics. Similar to decision trees, a random forest learner could be trained to classify whether a column contains ordinal data or not. The learner constructs multiple decision trees based on features of the data and holds a majority vote to classify whether the given dataset is nominal or ordinal.

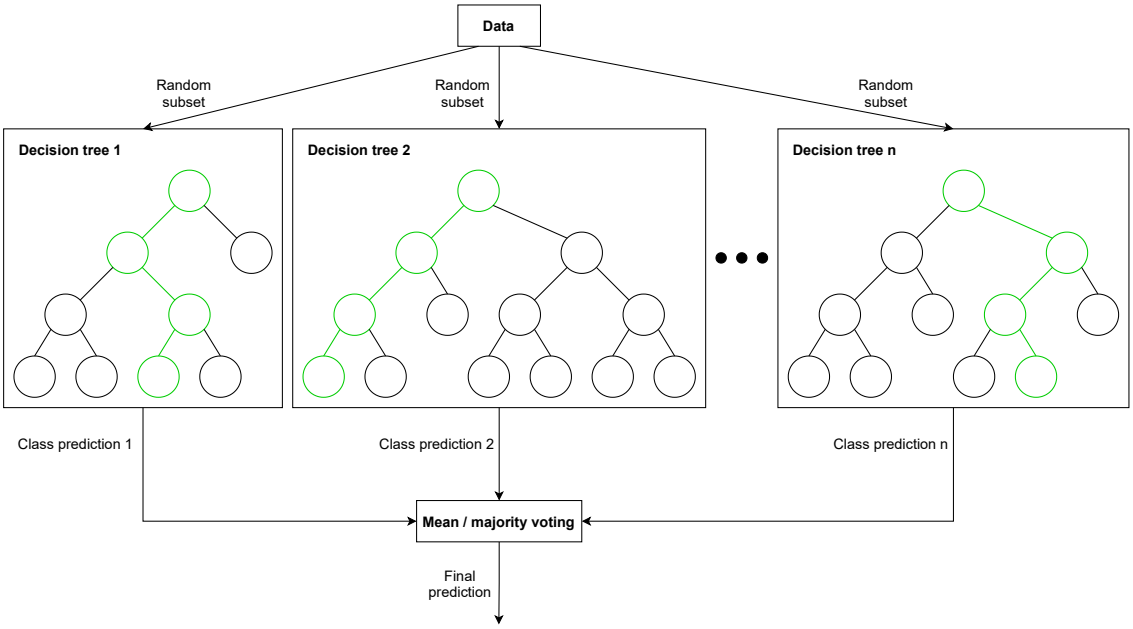


Figure 3.3: An example of a random forest. A multitude of decision trees are constructed based on random subsets, and each tree outputs a class prediction.

3.2.4 Gradient boosting

This is also an ensemble learning method for classification and regression tasks similar to random forest. However, gradient boosting takes a weak learner[1] (e.g., a decision tree) and iteratively modifies this learner to improve its performance instead of creating a multitude of learners that produce different class predictions. Gradient boosting is therefore not only more memory efficient, but it has also shown that it usually outperforms random forests on the same data [33]. Similar to decision trees and random forests, gradient boosting classifiers can be used to classify ordinality in the data based on the most relevant features in the given (meta-)dataset.

Gradient boosting is made up of three components. Firstly, a proper loss function needs to be defined and optimized for the given problem. Secondly, a weak learner is selected to be used as the base of the predictor. This weak learner is almost always a decision tree. Finally, an additive model is specified, which adds trees to the existing tree one at a time. When adding these trees, a gradient descent procedure is usually applied to minimize the loss and update the weights.

In literature, Friedman (2002) designed a stochastic gradient boosting algorithm which incorporates randomization to increase the robustness against overcapacity of the base learner and to improve the execution speed [34]. More recently, Dorogush et al. (2018) created a novel gradient boosting library that successfully handles categorical features and reduces computation times even further by incorporating both the GPU and CPU upon execution [31].

3.2.5 Neural networks for tabular data

Several (deep) neural network architectures have recently been developed to incorporate neural network solutions for tabular data. These solutions are known to equal or outperform gradient boosting algorithms for classification tasks. One of these neural network solutions is TabNN, as proposed by Ke et al. [50]. Their approach is to leverage feature groups obtained from a gradient boosting decision tree (GBDT) and feed them into a recursive neural network architecture with shared embeddings (RESE). More specifically, all feature groups obtained from the GBDT are

[1] A weak learner is a predictive model which performs slightly better than random guessing.

merged into k sets while maximizing the minimum number of common features in one set. Then, each set is processed in its own RESE (giving us k RESEs in total) to allow more critical features in each set to contribute more to the overall result. Finally, the final result of all RESEs are concatenated as inputs of a final fully connected layer. An illustration of the architecture is depicted in Figure 3.4.

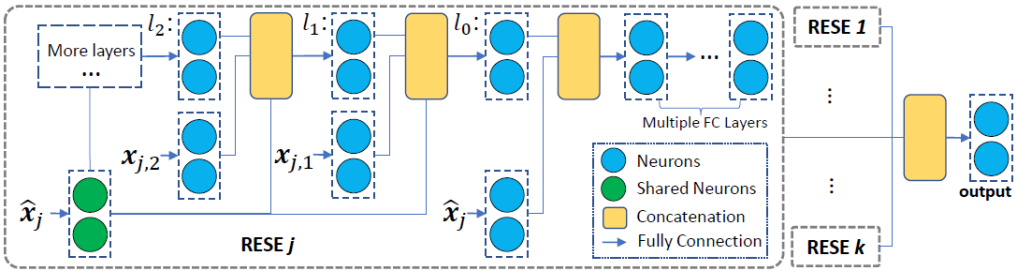


Figure 3.4: The architecture of TabNN, containing k RESEs where the embedding of common features $\hat{x}_j$ is shared in these layers (shared neurons) [50].

In the context of ordinality detection, a neural network used for classification in tabular data could be trained exclusively on nominal and ordinal data, such that it may discover latent relations between the data and the ordinality that can be leveraged in production.

3.3 String categorization and processing

This task refers to recognizing, categorizing, and processing different string entities in the dataset. In a non-automated scenario, a data scientist can (sub)consciously make this distinction for most strings based on their intuition. Without techniques that address this part in an automated scenario, choosing the best categorical encoding for a given dataset would be random at best, and processing remains relatively minimal. Listed below are a few methods that could deliver promising results for categorizing strings and further processing them.

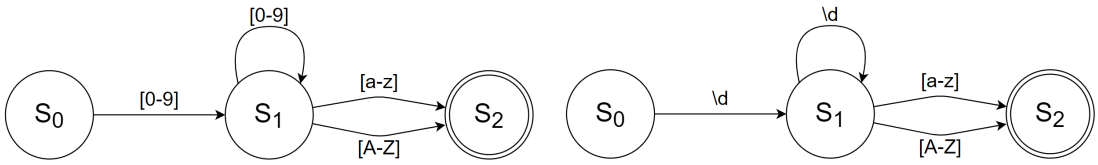
3.3.1 Regular expressions (Regex)

Regular expressions are a powerful tool in computer science. Regexes consist of a sequence of characters that allows for pattern matching, location, and management[2]. The applications of regexes in natural language processing (NLP) models include web searching, word processing, field validation in databases, and information extraction [45]. According to Shahbaz et al. (2012), this method is proven to be able to extract identifiers in strings [87]. This property could potentially allow further categorization of strings by, e.g., automatically developing complex PFSMs using regexes [27].

As an example, consider that a regular expression needs be constructed to match strings containing one or more digits followed by a single string character. There are various syntaxes available for creating regex rules, but in this example we only consider the basic concepts[3]. Given the basic regex rules, we can design the regular expression `[0-9]+[a-zA-Z]`, which states that we want to match any strings that start with one or more characters (`+`) from the set `0` to `9` (`[0-9]`), followed by exactly one character from the set `a` to `z` or `A` to `Z` (`[a-zA-Z]`). An alternative regex that gives the same result is defined as `\d+[a-zA-Z]`, where `\d` stands for a single digit. It is possible to visualize these regexes in the form of finite automata [42]. An illustration of the aforementioned regexes is depicted in Figure 3.5.

[2] https://www.computerhope.com/jargon/r/regex.htm

[3] An inexhaustive, but introductory, list of basic regex rules can be found at https://www.rexegg.com/regex-quickstart.html.



(a) `[0-9]+[a-zA-Z]` (b) `\d+[a-zA-Z]`

Figure 3.5: Finite automata of the example regular expressions

From the example, it becomes apparent that constructing slightly more complex expressions can lead to rules that are hard to interpret, which is also the major disadvantage of using regexes.

3.3.2 Named-entity recognition and classification (NERC)

This technique is a subtask of information extraction[4] and is mainly used for unstructured text. Using NLP, NERC can recognize strings and categorize them as, for example, a name or a location. In the context of string feature processing in tabular data, NERC or a similar technique can be applied to reduce the complexity of sentences in columns by only considering keywords. It could also be considered as a technique to identify string feature types using a trained model.

The working of a NERC system can be seen as follows. First, the system extracts information from the given data and selects a set of candidate entities based on pattern matching, linguistics, syntax, semantics, or a combination of approaches[5]. After this, the system tries to classify each entity based on either linguistic grammar-based techniques or statistical models.

Research has suggested that linguistic grammar-based techniques have a higher precision compared to statistical models [48]. However, this high precision comes at the cost of a lower recall and the time it takes for experts in the field of computational linguistics to design the semantics [48]. On the other hand, statistical models such as machine learning algorithms often require a large amount of training data that are manually annotated. As a suggestion to avoid some of this manual effort, semisupervised models were suggested [59, 74]. Statistical systems can also be constructed using unsupervised learning. In this setting, the typical approach for NERC is clustering [73]. Studies have suggested that this approach can be used to link input words with appropriate named-entity types [12] or to classify named entities under a given type using Pointwise Mutual Information and Information Retrieval (PMI-IR) [32]. An example of how NERC can be applied on a dataset can be seen in Figure 3.6. In the context of string feature type inference, the annotations can be used to identify similar entries in a string column and to classify the string feature type of the column as the annotation that occurs the most frequent. As for preprocessing string features, the annotations can be used to extract the most relevant features in a string entry consisting of sentences to reduce the overall complexity while retaining most of the information.

3.3.3 Automated feature engineering

In feature engineering, data scientists apply their domain knowledge to extract certain attributes or properties (features) from raw datasets. By determining which features are relevant for the problem to be solved, data scientists can process the data to maximize the performance of a machine learning algorithm. Research on optimal automation of this process is still ongoing. Previous work suggests using features obtained from decision trees [37] and multi-relational decision tree learning, which makes use of supervised learning [52]. However, these techniques may introduce many redundant operations unless the algorithm is regulated using incremental updates. A more recent development in automated feature engineering involves a Deep Feature Synthesis algorithm,

[4] Automated retrieval of specific information in a text.
[5] https://www.expert.ai/blog/entity-extraction-work/

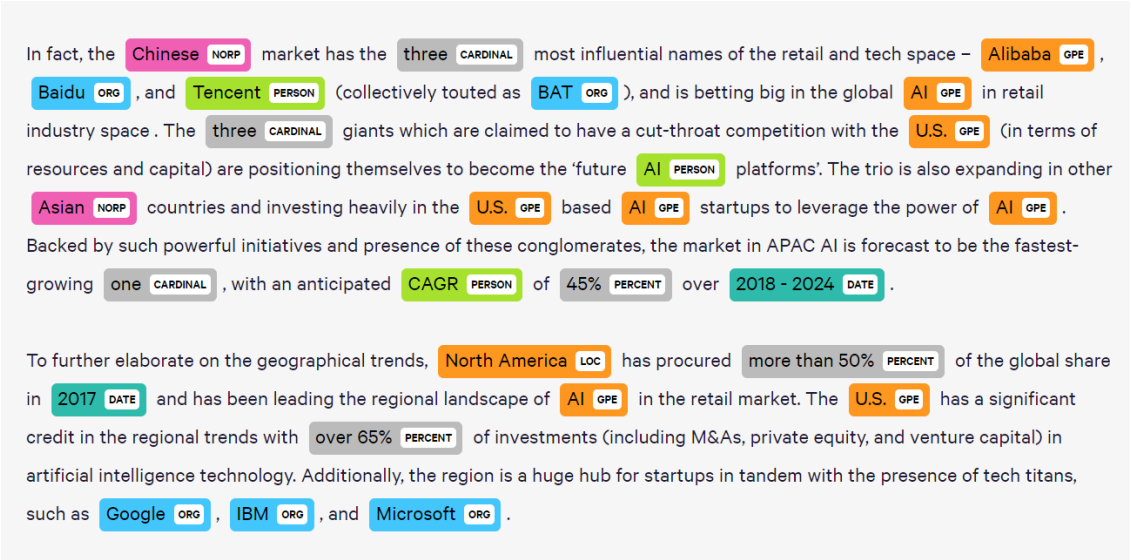


Figure 3.6: An example of NERC on a piece of text. Image taken from Medium at `https://medium.com/@b.terryjack/nlp-pretrained-named-entity-recognition-7caa5cd28d7b`

which was able to beat 615 of 906 human teams in terms of feature engineering [39, 47]. Tools that include Deep Feature Synthesis for automated feature engineering include Featuretools [4], OneBM [56], and ExploreKit [49]. These tools are aimed at transforming relational datasets into feature matrices for machine learning, rather than cleaning the data that is presented to them.

3.4 Encoding techniques

Several encoding techniques can be applied to different types of string data. One type of encoding that is commonly used in structured data is categorical encoding. Machine learning models exist that work exclusively on numerical values because they execute certain operations that can only be performed on numbers. Encoding ensures that string data is transformed to a numerical representation such that they can be used in such machine learning applications [79]. Categorical string data can be subdivided into ordinal data (categories have an order) and nominal data (categories have no order, such as `cold`, `warm`, and `hot`). A significant amount of research and development has been done to develop encoding strategies for a variety of applications [29, 91, 8, 68]. However, robustly automating these encoding strategies remains challenging and relatively open. This section describes some commonly used categorical encoders and a few experimental encoders that were shown to outperform them and are deemed promising for applications in an automated environment.

3.4.1 Common practices

One-hot encoding This encoder is used when the data is nominal and mutually exclusive [29]. The encoder maps every category in a column to a unique binary value of length equal to the number of categories. More specifically, for n categories, one-hot encoding maps these categories to n unique binary values containing a single one and $n-1$ zeroes. Despite its popularity and performance, the learning algorithms may suffer from high dimensionality since the dimensions of the encoded data depend on the number of categories in the data [19]. Furthermore, similar entries may be placed into separate categories when the data is unprocessed (e.g., entries contain errata). Alternatives to one-hot encoding that decrease the dimensionality include dummy encoding (similar to one-hot encoding except that it maps the categories to $n-1$ unique binary values),

effect encoding, and binary encoding [29].

Ordinal encoding This encoder maps every unique categorical value to a positive integer. For example, the entries `'cold'`, `'warm'`, and `'hot'` would be encoded as `1`, `2`, and `3` respectively. The encoder assigns the order either at random or by a pre-defined order that the user passes to the encoder. Given that integers are naturally ordered, ordinal encoding helps to reflect the ordering of the original data to the machine learning model[6]. However, the downside of this encoder is that it is sub-optimal for nominal data. If nominal data is encoded using the ordinal encoder, learning algorithms may incorrectly assume that order plays a role, possibly making models unnecessarily complex and prone to overfitting[7].

Hash encoding Hash encoding represents categorical data in the form of hashes. This technique has the major advantage that the length of the hash does not depend on the number of categories in the data [91]. This advantage implies that hash-encoded data can have much fewer dimensions than one-hot encoded data. A drawback to this method is that it is prone to collision and loss of information, which could result in errors in the trained machine learning model [91].

Target encoding In contrast to the previously mentioned encoders, target encoding is an encoding technique based on both the dependent variable and the categorical variable [66]. An example of a target encoder can be found in the `scikit-learn` library [67]. This target encoding can be applied on both continuous and categorical targets. For categorical targets, the features are replaced with a mix of a posterior probability of the target given the particular categorical value and the prior probability of the target over all the training data. Each feature is encoded using the following formula:

$$S_i = \underbrace{\lambda(n_i)\frac{n_{iY}}{n_i}}_{\text{posterior probability}} + \underbrace{(1-\lambda(n_i))\frac{n_Y}{n_{TR}}}_{\text{prior probability}}$$

Where $\lambda(n_i)$ represents a weighting factor bounded between 0 and 1, n_{iY} represents the ratio between the number of observations of Y for similar cells i, n_i represents the size of the cell, n_Y represents the total number of observations of Y, and n_{TR} represents the number of records contained in the training set.

When the target values are continuous, the features are replaced with a mix of the expected value of the target given the particular categorical value and the expected value of the target over all the training data. The formula is similar to that of categorical targets, except that the mean of target value Y is considered across the training data. Each feature is encoded using the following formula:

$$S_i = \underbrace{\lambda(n_i)\frac{\sum_{k\in L_i} Y_k}{n_i}}_{\text{exp. value given particular value}} + \underbrace{(1-\lambda(n_i))\frac{\sum_{k=1}^{N_{TR}} Y_k}{n_{TR}}}_{\text{exp. value over all samples}}$$

Where L_i is the set of observations, of size n_i, for which $X = X_i$, and Y_k represents the target attribute at row k. The target encoder also considers that there could be specific categories in the data that rarely occur but are not outliers.

Geocoding This encoding technique is useful for encoding geographical string data in structured and unstructured data into a representation of latitude and longitude values. These encodings could be useful when models need to be trained with geographical data. Python libraries `pgeocode` [8] and `geopy` [5] promise to be effective solutions in realizing this type of encoding.

[6] https://machinelearningmastery.com/one-hot-encoding-for-categorical-data/

[7] https://towardsdatascience.com/a-common-mistake-to-avoid-when-encoding-ordinal-features-79e402796ab4

Word embeddings This encoding technique takes words or phrases from a vocabulary and maps these to a value in a vector space. It is beneficial for data that consists of words or sentences. Word2Vec is a well-established approach to perform this encoding and provides practical model architectures to make effective embeddings from such datasets [68, 69]. An example of how words can be represented in a three-dimensional vector space can be seen in Figure 3.7. Global Vectors for Word Representation (GloVe) is another approach to create word vectors [78]. It is a hybrid method that uses machine learning and statistics to construct low-rank approximations to retrieve latent features, including those for comparative and superlative words. An approach to use word embeddings as an encoding type in an automated setting is to use a library of trained word vectors (such as GloVe) in combination with random embeddings for words that are not present in the trained model.

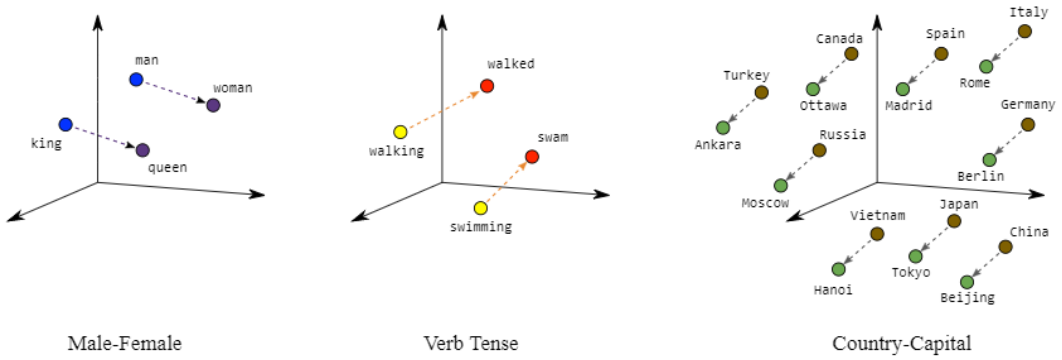


Figure 3.7: Examples of several word embeddings and the possibility to produce certain analogies. Image taken from https://developers.google.com/machine-learning/crash-course/embeddings/translating-to-a-lower-dimensional-space.

3.4.2 Similarity encoding

In unstructured data, the set of possible categories is unknown due to their lack of standardization. Unprocessed data can suffer from a wide range of errors such as errata (typos) [51]. If these are not fixed, entries with (un)intentional errors are viewed as separate categories by the statistical model. To tackle this issue, Cerda et al. (2018) proposed an encoding technique that performs a one-hot encoding based on string similarity measures [25].

This encoding strategy works as follows. Let $\text{sim} : (\mathbb{S} \times \mathbb{S}) \to [0, 1]$ be any string-based similarity measure[8] such that:

$$\text{sim}(s_1, s_2) = \text{sim}(s_2, s_1), \quad \forall s_1, s_2 \in \mathbb{S}$$

Furthermore, let V be a categorical variable of cardinality k. The similarity encoding that they propose replaces all instances of V with category $d^i, i = 1...n$ by a feature vector $\mathbf{x}^i \in \mathbb{R}^k$ such that:

$$\mathbf{x}^i = [\text{sim}(d^i, d_1), \text{sim}(d^i, d_2), ..., \text{sim}(d^i, d_k)]$$

This encoder can be demonstrated with an example. Let $x_1 = \text{Paris}$ and $x_2 = \text{Parisian}$, with 3-gram similarity as the string-based similarity measure. The 3-gram similarity measure breaks down entries into consecutive grams of length three and calculates the ratio of the intersection and the union of the grams of two entries. In our example, we obtain the 3-grams $x_{1,grams} = \{\text{Par}, \text{ari}, \text{ris}\}$

[8] These measure the similarity between two strings, with $\mathbb{S}$ as the set of strings. A list of commonly used measures can be found in [36].

and $x_{2,grams} = \{\text{Par}, \text{ari}, \text{ris}, \text{isi}, \text{sia}, \text{ian}\}$, and we can observe that three of the 3-grams are similar for both words. The 3-gram similarity measure for each possible pair of entries is then:

$$\begin{aligned}\text{sim}(x_1, x_1) &= 1\\ \text{sim}(x_1, x_2) &= \frac{3}{6} = 0.5\\ \text{sim}(x_2, x_2) &= 1\end{aligned}$$

The similarity encoder using the 3-gram similarity measure would encode both entries as follows:

$$\begin{aligned}\mathbf{x}_1 &= [\text{sim}(x_1, x_1), \text{sim}(x_1, x_2)] = [1, 0.5]\\ \mathbf{x}_2 &= [\text{sim}(x_2, x_1), \text{sim}(x_2, x_2)] = [0.5, 1]\end{aligned}$$

To tackle any possible high-cardinality encodings that derive from this strategy, they also explored some dimensionality reduction methods to reduce the number of dimensions in the encoding [25].

3.4.3 Categorical encoders for high-cardinality strings

Research is still ongoing about providing accurate categorical encodings of string data without the need for data cleaning with limited dimensionality. Cerda et al. (2020) proposed two novel encoding methods for high-cardinality string data: a *min-hash encoder* and a *Gamma-Poisson matrix factorization encoder* [24]. They have shown that these encoding strategies provide a scalable and automated replacement for cleaning and encoding categorical data.

Min-hash encoder This encoder is based on the min-hash function, which is one of the most famous functions of the locality-sensitive hashing (LSH) family [21, 35]. The LSH family consists of algorithmic techniques that hash similar inputs into the same "buckets" with high probability [82] and is originally designed to retrieve documents that are similar in terms in terms of the Jaccard coefficient[9]. This function can be used as an encoder by building it using salt numbers (random data) instead of random permutations to make it computationally efficient, as generating random permutations may take some time. Let $\mathcal{X}^*$ be a totally ordered set, with non-empty and finitely cardinal $\mathcal{X} \subseteq \mathcal{X}^*$. Furthermore, let h_j be a hash function on $\mathcal{X}^*$ with salt value j. We can then construct the min-hash function $Z(\mathcal{X})$ as:

$$Z(\mathcal{X}) \stackrel{\text{def}}{=} \min_{x \in \mathcal{X}} h_j(x)$$

For the problem regarding categorical data, Cerda et al. (2020) rely on a fast approximation of the Jaccard coefficient between two sets of consecutive n-grams for a string s (denoted as $J(\mathcal{G}(s_i), \mathcal{G}(s_j))$). With $\mathbf{x}^{\text{min-hash}}$ representing the min-hash feature map, they define the min-hash encoder as:

$$\mathbf{x}^{\text{min-hash}}(s) \stackrel{\text{def}}{=} [Z_1(\mathcal{G}(s)), ..., Z_d(\mathcal{G}(s))] \in \mathbb{R}^d$$

By considering the hash functions as random processes, it can be implied using Equation 6 from [24] that the encoder has the property that:

$$\frac{1}{d} \cdot \mathbb{E}\big[\mathbf{x}^{\text{min-hash}}(s_i) - \mathbf{x}^{\text{min-hash}}(s_j)\big] = J(\mathcal{G}(s_i), \mathcal{G}(s_j))$$

Cerda et al. (2020) state that this encoder is especially suitable for categorical encoding since it is fast to compute and is stateless. Furthermore, they state that the min-hash encoder can form inclusion relations of strings into an order relation in the feature space, as can be seen in Figure

[9] The Jaccard coefficient is used to calculate the similarity and diversity of sample sets.

3.8. However, they also mention that the encoding is hard to invert and interpret in terms of the original string entries because the encoder relies on hashing [24]. In practice, users can define the number of dimensions the encoded data must have, and any given column of data is encoded in those dimensions using the min-hash feature map, where each dimension also represents an order relation in the feature space.

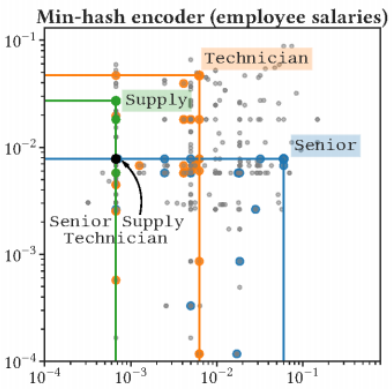


Figure 3.8: A visual representation of the order relations of strings created by the min-hash encoder. In this example, three containment regions are laid out (Supply, Technician, and Senior) where containment in a word region can imply that a string entry contains this word (dots are grey if the entry does not contain the containment region word). The entry "Senior Supply Technician" is one of the few that crosses all three regions, implying an order is established based on how many substrings are contained in an entry [24].

Gamma-Poisson matrix factorization To allow high-cardinality strings to have some degree of interpretability compared to the min-hash encoder, Cerda et al. (2020) designed the Gamma-Poisson matrix factorization encoder (gap encoder). This encoding strategy makes use of a generative model of strings from latent categories using the Gamma-Poisson model [23]. Given that these string entries may contain errata and are relatively smaller compared to the entries that the model was initially developed for (text documents), Cerda et al. relied on the sub-string representation of said entries (i.e., they represent each entry by its count vector of character-level structure of n-grams). Each string entry described by its count vector is then modeled as a linear combination of unknown prototypes that represent the latent categories of the entry. An example of how the gap encoder allows users to interpret the data entries and their relation to different latent topics produced by the model is depicted in Figure 3.9.

3.4.4 Determining the order in ordinal data

As mentioned in Subsection 3.4.1, the ordinal encoder helps to reflect the ordering of the original data to the machine learning model. When the ordering is not passed through the encoder as a parameter, the encoder will take the lexicographical ordering of the original data, which does not always represent the true order of the data. This subsection will cover two approaches that aim to define the natural ordering of entries in ordered data automatically.

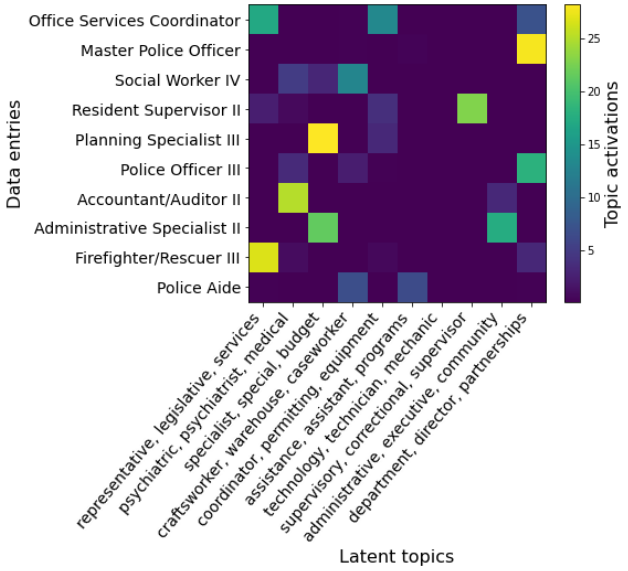
Heuristic approach Ordinal data may consist of quantifiers, comparatives-superlatives, and antonyms of certain string entries. Based on these properties, a heuristic approach can be set up to determine the order of data based on the contents of each entry. The method could do so by comparing every entry to a list of pre-defined word(s) that represent one of the aforementioned properties. Pre-defined words for each property could be manually set or determined by a linguistic

	employee_position_title
0	Office Services Coordinator
1	Master Police Officer
2	Social Worker IV
3	Resident Supervisor II
4	Planning Specialist III
...	...
380	Labor Relations Advisor
381	Director Office of Intergovernmental Relations
382	Senior Public Health Advisor
383	Psychiatric Nurse Clinical Specialist
384	Supervisor Transportation Systems Technical Ce...

(a) Data before applying the gap encoder.

	employee_position_title
0	[17.057869161872322, 0.06150746848751993, 0.05...
1	[0.16426548472894062, 0.0628892088870163, 0.10...
2	[0.061680932126910934, 5.042261051661167, 3.06...
3	[2.4370153941102126, 0.8975359756986669, 0.108...
4	[0.061467989494949124, 0.0641300947009987, 28....
...	...
380	[3.8574512179435514, 0.06579649781822677, 0.05...
381	[5.255207332870798, 0.12510972883593321, 0.081...
382	[4.995448576635727, 0.15909665754090171, 0.302...
383	[0.0724383637792775, 27.939071514809168, 17.36...
384	[0.14817563803141875, 3.638911453163109, 0.093...

(b) Data after applying the gap encoder.



(c) Heatmap where the x-axis displays the ten top-three latent topics of the data generated by the Gap encoder, and the y-axis displays the first ten entries of the data. A higher topic activation for a given entry implies that the entry has a higher affinity with one of the top-three latent topic compared to other latent topics. This topic activation directly translates itself into the encoded value, as can be seen by comparing the entries in (a) to the encoded results in (b)

Figure 3.9: Application and interpretability of the Gap encoder on string data.

Python package. For example, several linguistic websites such as linguapress[10] and curso-ingles[11] contain lists of quantifiers or comparatives and superlatives that can be included in the dictionary of the approach. Another example is the Natural Language Toolkit (`nltk`) package, which provides a built-in WordNet module that can be used to fetch a list of antonyms for a given word [70]. The obtained variations can be ordered heuristically and each entry in the given data will be ranked based on where they are placed within that order. However, a disadvantage to this approach is

[10] https://linguapress.com/
[11] https://www.curso-ingles.com/

that it will only work accurately when all entries are in the correct format and contain one or more of the aforementioned properties.

Sentiment analysis Another approach to determining the order in data is to take a trained sentiment analysis tool and evaluate each string entry using sentiment analysis. In general, sentiment analysis tools evaluate sentences based on their degree of positivity or negativity. For example, the sentence "I hate you" would receive a negative evaluation because the word "hate" is determined to be negative in the pre-defined sentiment analyzer. Similarly, this approach could be applied for evaluating string entries based on their degree of positivity or negativity and order each entry based on this score. The remainder of this paragraph describes some trained sentiment analysis tools from various NLP libraries.

The `nltk` package contains the VADER sentiment analysis tool by Hutto et al. (2014), who claim that VADER outperforms individual human raters and generalizes more favorably across contexts than any of their benchmarks on microblog-like contexts [43]. VADER makes use of a bag-of-words approach, which essentially consists of a lookup table for positive and negative words and heuristics for certain keywords to determine the overall intensity of sub-sentences. The disadvantage of this approach is that it only works on words contained within the corpus of the model. This limitation implies that errata or words that are not part of the vocabulary are seen as "neutral" and will not receive any score. An example of how a sentence is evaluated using `nltk` can be seen in Table 3.1.

Words	i	really	liked	the	food	but	the	service	was	terrible	**Total**
Score	0	0	0.4215	0	0	0	0	0	0	-0.4767	-0.4773

Table 3.1: The sentiment intensity evaluation of `nltk` VADER on an example sentence.

The TextBlob package works similarly to `nltk`; it makes use of a bag-of-words classifier but includes a subjectivity analysis to evaluate how opinionated a piece of text is [62]. The major disadvantage of this package in our setting is that there are no heuristics to take quantifiers or negations into account. The absence of these heuristics implies that a significant amount of sub-sentences (such as "not bad") are incorrectly evaluated, making the package sub-optimal for determining the ordering in data.

The `FlairNLP` (Flair) package provides a unified and straightforward interface for word and document embeddings [11]. Flair includes a sentiment analysis classifier based on a character-level long short-term memory (LSTM) neural network that predicts sentiments while taking the letter and word sequences into account. Because of the properties of the LSTM, the trained model can handle negations, intensifiers, and out-of-vocabulary (OOV) words, which makes it a powerful tool even when unknown words or errata are presented to the model. The ordinal data could be passed to model, which will then give each entry a sentiment score. The data can then be ordered according to this score.

3.5 Other data cleaning techniques

Several other data cleaning techniques can be used to handle string data. However, given the context of the problem, these techniques might be less relevant on their own compared to the techniques that were previously mentioned in this review. For each technique, we list some basic approaches and some slightly more sophisticated strategies that can be applied in different cases.

3.5.1 Handling missing values

A significant number of real-life datasets contain incomplete or missing observations, which is an issue for most machine learning algorithms as they only work if the data is complete. Missing

values also occur in categorical data, implying that missing values also need to be addressed in this work for completeness. Missing values can be addressed by removing or imputing them from the data. However, it is essential to consider which technique benefits the results from the data. Missing values that are not appropriately handled may introduce issues during training [94]. For example, removing a significant number of rows to handle missing values may introduce bias to machine learning algorithms. This subsection describes the types of missing data that are possibly present in datasets and how to handle them.

In 1976, three different types of missing data were identified by Donald Rubin and are still used to date to apply the optimal missing value handling technique [85].

The first type of missing data is data missing completely at random (**MCAR**). In this case, the probability that a data point is missing from the dataset is equal for all points, suggesting that missing values are unrelated to the dataset itself and that no imputation technique or deletion will impact the performance. In other words, the data that is MCAR is unrelated to observed or unobserved values, and no structural relationship to the missingness can be established in any way. We can test the data using Little's Test of MCAR, which is a chi-square statistical test that tests whether the null hypothesis 'The missingness mechanism of the incomplete dataset is MCAR' can be rejected [60]. In other words, this test checks for the correlation of a missing value in a feature and the value of any other of the features, where correlation indicates that the missing values are MAR.

The second type of missing data is data missing at random (**MAR**). For MAR, the probability that a value from a specific group is missing is the same for all values contained within the same group of the observed data. In other words, MAR data is unrelated to unobserved values and may be related to observed values. According to Stef van Buuren, modern missing data methods generally start from the MAR assumption.

The third type of missing data is data missing not at random (**MNAR**). In this case, the probability that a value is missing varies from one observation to another for unknown reasons. In other words, the missing data is related to the values that they are supposed to represent. An example of MNAR data includes censored data, such as the body weights of overweight individuals who are less likely to report their weight. Missing data is classified as MNAR when neither MCAR nor MAR holds. This case is the most complex and cannot be determined solely on the observed data and might require a domain expert to handle these cases accordingly. Since the latter might not be feasible in an automated setting (since a domain expert is not always present). It is therefore assumed in this work that MNAR data is handled similarly to MAR.

Deleting entries A simple approach to handle missing values is to remove rows or columns that contain missing values. This approach is not costly and can be applied to all types of data. However, the disadvantage of this approach is the significant loss of information when a relatively large amount of data is missing throughout the dataset, which can introduce bias in the data and reduce performance. In practice, it is regarded as a rule-of-thumb to remove rows containing missing values when the total number of missing values accounts for at most 5% of all data.

Mean or mode This simple technique replaces missing values with the mean (for numerical data) or the mode (for categorical data) of non-missing cases of that variable. Despite its simplicity, it has the significant drawback of possibly making the data biased for data that is MAR or MNAR.

Multinomial logistic regression (MLR) Can be used to predict the probabilities of the different possible outcomes of categorical data, given a set of variables[12]. This technique works under the assumption that each independent variable has a single value for each case. A disadvantage to MLR is that it does not scale well, implying that it becomes computationally expensive to use when the number of categories is high.

[12] https://en.wikipedia.org/wiki/Multinomial_logistic_regression

Multiple Imputation This technique makes use of single imputation techniques such as mean or mode, except that the imputed values are drawn multiple times from a distribution rather than once. This technique creates multiple datasets containing imputed values, which are then combined into one result by, for example, calculating the mean, variance, and confidence interval of the variable of concern [93]. The advantage of using multiple imputations is that it works well on all cases of missing values (MCAR, MAR, and MNAR). Several approaches of the multiple imputer have been developed, such as multiple imputations by chained equations (MICE) [14] and `scikit-learn` multivariate imputer, which models each feature with missing values as an estimator function of other features [77].

Predictive mean matching (PMM) This technique builds small subsets of data points that have matching outcome variables[13]. The data points containing missing values will then be filled with real values sampled from other entries in the same subset. This method consequently reduces the bias introduced by imputation [13]. This method was proven to work on both numerical and categorical variables, according to van Buuren et al. [22].

Hot deck imputation This technique finds substitute values from similar entries that are similar to those that are missing data. The similarity between entries can be determined by various algorithms that compute the distance between entries. An example of an algorithm that could be used is the k-nearest neighbors algorithm.

Imputation through deep learning It is possible to lift limitations from conventional imputation methods by making use of deep learning methods. Depending on the deep learning method used, they are (1) able to compute both numerical and non-numerical values (n-gram models, Long Short-Term Memory models (LSTMs) [16], and RandomForests [88]), (2) able to determine underlying distributions (General Adversarial Networks (GANs) [92] and Variational Autoencoders (VAEs) [65]), and (3) computationally efficient (n-gram models and LSTMs [16]).

3.5.2 Outlier detection

Outliers (also referred to as anomalies) are data points that deviate significantly from most other obtained results [40]. For numerical values, it is possible to detect outliers by finding specific patterns or observations in the data that do not follow the expected behavior [28]. According to Hampel (1973), the total number of outliers in a dataset is around 5 to 10% of the given dataset [38], depending on the domain of the data and the method used to record it. In the context of categorical (string) values, there is no concept of outlier detection similar to that of numerical values, as each value counts as a label. This work considers errata (typos) as string outliers. For a string entry to be categorized as an outlier, it must be strongly similar to another entry in the data regarding their string similarities, and it must occur significantly less frequently compared to the strongly similar entry. A disadvantage to this approach is that edge cases where a category is similar to another category that occurs more often might also be considered an outlier. These criteria can therefore be relatively robust if configured accordingly. This subsection covers a few outlier detection methods for numerical values and some string metrics that can be used to determine potential outliers in categorical values.

Statistical outlier detection A common statistical measure that is used to detect outliers is the interquartile range (IQR). In this measure, all data points that fall below Q1 - 1.5·IQR and above Q3 + 1.5·IQR are considered potential outliers[14]. This method is useful when the distribution of the data is known beforehand or when the data follows a univariate distribution[15].

[13] http://stefvanbuuren.name/fimd/sec-pmm.html.
[14] https://en.wikipedia.org/wiki/Interquartile_range
[15] A probability distribution of a single random variable.

Distance-based outlier detection Local outlier factor (LOF) computes the local density deviation of a given data point with respect to its neighbors [15, 20]. In this method, a data point is considered an outlier when its density is significantly smaller than the density of its neighbors. Next to LOF, one can also make use of a One-Class Support Vector Machine (SVM). This technique finds a hyperplane whose distance is maximized from the origin and also separates all the data points from the origin [86], consequently creating a decision boundary in which a small fraction of all data points falls outside. These points are then considered to be outliers.

Cluster-based outlier detection These are generally unsupervised detection methods that group similar data points into clusters. Outliers do not fall into these clusters and can be detected in this way. k-means clustering, as stated by Chandola et al. (2009), is a clustering technique that makes k clusters out of the data and calculates the distance of each data point to its neighboring cluster [28]. A drawback to this method is that a group of outliers can also make clusters and remain undetected. Another clustering technique is the isolation forest [61], which detects outliers based on their frequency and difference in values with respect to normal data.

String metrics These metrics are used in mathematics to measure the distance between two strings of text. One of the most well-known and basic string metrics is the Levenshtein distance (or edit distance), developed by Vladimir Levenshtein in 1966 [57]. The distance between two words is calculated by taking the minimum number of single-character edits required to change one word into the other, where an edit is either an insertion, deletion, or substitution. More formally, given two strings x, y of lengths $|x|$, $|y|$ respectively, the edit distance is equal to $\texttt{dist}(x, y)$, where

$$\texttt{dist}(x,y) = \begin{cases} |x| & \text{if } |y| = 0 \\ |y| & \text{if } |x| = 0 \\ \texttt{dist}(x[1..], y[1..]) & \text{if } x[0] = y[0] \\ 1 + \texttt{min} \begin{cases} \texttt{dist}(x[1..], y) \\ \texttt{dist}(x, y[1..]) \\ \texttt{dist}(x[1..], y[1..]) \end{cases} & \text{otherwise} \end{cases}$$

Other distance metrics that can be considered for string outlier detection include the Jaro-Winkler distance metric, which favors strings that are similar up to a certain prefix [80], and the Jaccard distance metric, which compares the intersection of characters to the union of all characters [44]. This metric is also used by Cerda et al. (2018) in their similarity encoder to calculate the similarity between two entries [25].

3.6 Existing data cleaning systems

This section describes some commercial state-of-the-art data preprocessing systems that data scientists can currently use. We distinguish each system based on its level of automation. After all systems are described, a brief explanation is provided on the gap this work aims to fill and how to tackle potential shortcomings that are introduced regarding the existing systems.

3.6.1 Manually operated systems

These systems allow users to have full control over the data cleaning process, such as OpenRefine [7], Data Ladder [2], and DataCleaner [3]. They allow users to repair and transform data with the help of an interactive and user-friendly interface. Some of these tools also contain intelligent heuristics to suggest what to do with certain parts of the data at certain steps in the process. An example of the user interface of one of the manually operated programs (OpenRefine) is depicted in Figure 3.10.

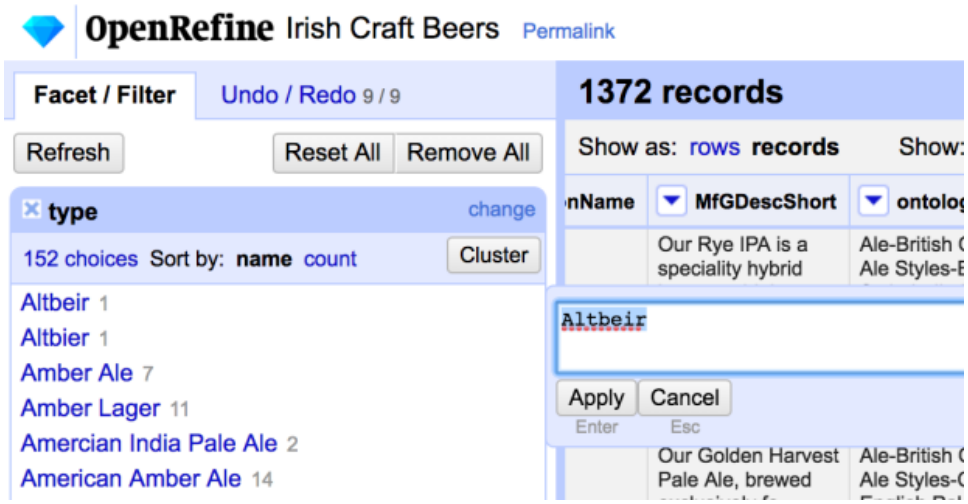


Figure 3.10: Part of the user interface of OpenRefine, in which users can easily navigate and filter data based on specific criteria and observe that certain entries might be the cause of unnecessarily high-cardinality in the data or errata (e.g., Altbeir vs. Altbier). Screenshot taken from `http://digitalnomad.ie/simple-openrefine-tutorial/`.

A disadvantage of such systems is that they still require a significant amount of time from users, despite the intelligent suggestions incorporated into the system. Furthermore, manual cleaning still relies on user interaction in the data cleaning process, which is what we aim to minimize.

3.6.2 Semi-automated systems

These systems offer complete control over the data cleaning process and allow users to define some degree of automation of the cleaning process. Semi-automated systems, therefore, allow the incorporation of domain knowledge to increase the data quality significantly [95]. Systems such as Trifacta Wrangler [9] and Cloudingo [1] aid users with interpreting, cleaning, and transforming data. Trifacta Wrangler, for example, provides intelligent, visual guidance using (undisclosed) 'artificial intelligence' to recommend possible data cleaning steps to increase the data quality. In addition to the guidance, they also allow users to set up automated pipelines to take care of repetitive data cleaning tasks for newly obtained data, as depicted in Figure 3.11. Another semi-automated tool that was recently introduced is CoClean [72], which has an integrated collaboration functionality that enables domain knowledge injection by multiple users simultaneously. However, the disadvantages of these systems are similar to that of the manually operated systems; they still require a human in the loop to clean the data and to set up the automated steps and pipelines. Even though this probably needs to be performed once, it still require these users to have some domain expertise to construct efficient pipelines. Furthermore, due to the market that these systems are designed for, most of them tend to be unavailable (or as a free trial) to the general public or require purchasing, which might discourage users from using these tools for personal use.

3.6.3 Automated systems

In addition to manual and semi-automated systems, several automated data cleaning systems have been introduced. Some of their main functionalities include data repairing, data benchmarks (i.e., evaluating the quality of the data based on criteria), and pipeline generation. These systems have proven that data cleaning can be automated and, therefore, less time-consuming. Examples of such data cleaning tools are ActiveClean [54], AlphaClean [55], HoloClean [83], SampleClean [53], and Raha [63]. These systems optimize the quality of the data or the generated pipelines using, for example, search algorithms, heuristics, or weakly supervised learning algorithms.

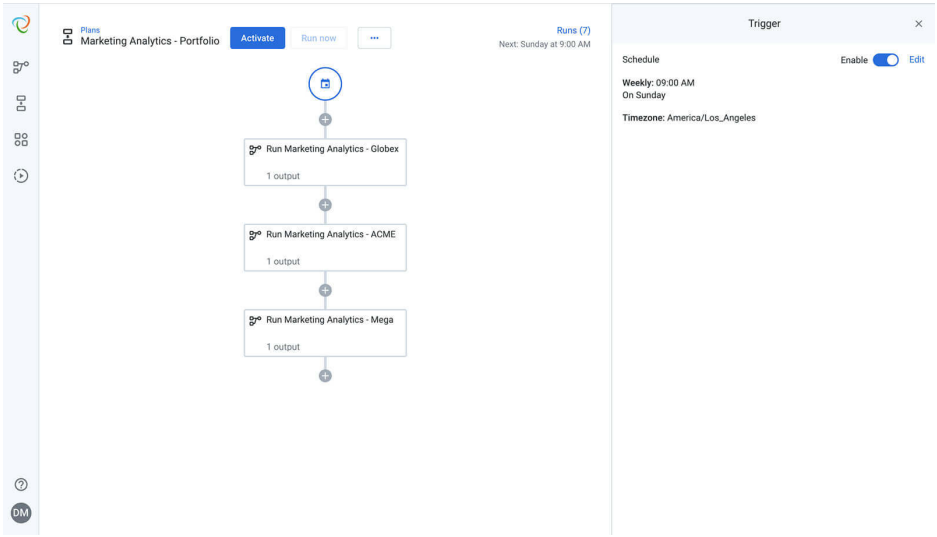


Figure 3.11: User interface of Trifacta Wrangler, showing how automated pipelines can be set up to perform a set of (cleaning) steps. Image taken from https://www.trifacta.com/.

However, most of these systems are not focused on automated string handling [53, 54, 83] or are focused on specific steps such as error detection and correction [55, 63]. Furthermore, these systems are rarely used by data scientists due to their lack of trust in the automation process [72], since assuring high quality data usually requires a human in the loop. Without any overhead or control in the process, there can be a degree of uncertainty about whether the system applies correct cleaning strategies. The degree of uncertainty would increase based on how generalized the system is. These are essential aspects to consider when designing an automated data cleaning system. For example, ensuring a degree of trust and certainty can be achieved by communicating to users which decisions were made during the process and which data cleaning steps were applied.

3.6.4 Addressing shortcomings of current systems

This work aims to fill the gap between automated string (feature type) preprocessing and encoding. Current systems only provide intelligent heuristics for cleaning datasets (sometimes without focusing on string feature types) or are limited to automating cleaning steps for numerical data. When constructing an automated system, it is essential to assess the shortcomings present in such systems (e.g., trust and uncertainty) and attempt to minimize them. As mentioned earlier, ensuring a degree of trust and certainty could be achieved by communicating to users which decisions were made during the process and which data cleaning steps were applied. Furthermore, the limited number of steps in an automated system can be addressed by solely focusing on handling string data. However, this has the disadvantage that numerical data is not thoroughly cleaned. Future work could address this concern by constructing automated numerical value cleaning on top of this work.

Chapter 4

Methodology

Based on the literature review of the previous chapter, it is now possible to design a methodology to handle the processing and encoding of string features robustly. In this thesis, we decided on developing a Python framework based on recent literature to address some of the issues regarding automated string handling. More specifically, the goal of the framework is to automate the handling and cleaning of string data, enrich the string data where possible, and inform users about various properties regarding string data in the dataset.

This chapter provides an overview of the framework and its design, along with an in-depth description of each module relevant to the scope of the thesis. Furthermore, additional modules that were implemented for completeness but are less relevant to the scope of the thesis are also briefly explained. A brief demonstration of how the framework is used in practice is found in Appendix A

4.1 Global framework overview

The main procedure of the framework consists of five steps made up of six separate modules in total. The modularity of the framework has the advantage that users can run modules separately without having to follow the entire cleaning process. This might, however, require users to pass additional information (e.g., data types) to certain methods. A representation of the workflow of the framework is depicted in Figure 4.1.

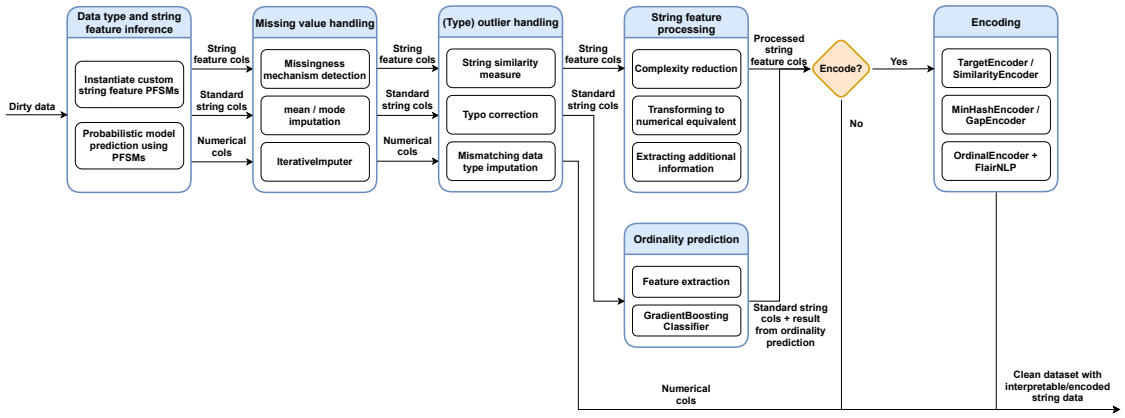


Figure 4.1: General overview of the workflow of the framework.

The framework takes a Pandas DataFrame as input and any of the following (optional) parameters:

- `y`: a column containing the target values or classes of the dataset (set to `None` by default). If information is passed through this parameter, the framework can encode the column and is able to use the target encoder for encoding string columns in the data.
- `encode`: a Boolean indicating whether the data needs to be encoded or not (set to `True` by default). This option is useful to users who want to interpret the cleaned string data or leverage the framework into their data preprocessing pipeline.
- `dense_encoding`: a Boolean indicating whether encoded entries consisting of multiple features should be put into one column as an array (set to `True` by default). If this parameter is set to False, the encoded data is split into multiple columns, each consisting of one feature. This option is helpful to reduce the number of columns in the data to retain a clear overview. However, for training, it might still be necessary to create a column for each feature.
- `display_info`: a Boolean indicating whether summarized information on each column must be displayed after the framework is finished (set to `True` by default). This option allows users to interpret the data and see an overview of the decisions made by the framework. Currently, the information consists of the number of unique values, the data types, the missing values, the outliers, the ordinality, and which encoding has or would be applied on each column.

Before the first step in the procedure, the framework checks and removes any empty columns to ensure robustness during missing value handling.

The first step in the procedure is to infer the data type or the string feature type of each column in the DataFrame. Each string column in the dataset is inferred either as a specific string feature type or as a 'standard' string whose feature type could not be inferred. Both cases are handled differently in the fourth step. In addition to data type and string feature type inference, the framework also identifies missing values and data type outliers in the data, which are handled in the second and third steps, respectively, to ensure robustness in later steps. This property makes it essential to perform data type and feature type inference in the first step of the framework.

The second step makes sure that all missing values in the data are addressed. The specific mechanisms (deletion or imputation) ensure that the most fitting missing value handling is applied and that missing values will not cause any issues in the upcoming steps.

The third step aims to remove and replace any outlying values in string data if applicable. The framework handles both data type outliers and string outliers. As mentioned in 3.5.2, a string entry is considered an outlier when its frequency is disproportionate compared to another entry it shares a high similarity score with. In other words, this step aims at handling entries with incorrect data types and strings that contain errata (typos).

The fourth step consists of two modules. The columns that were inferred in step one as string features are further processed with the aim to, for example, reduce string complexity or obtain additional features that can be useful for further data analysis by domain experts. The standard string columns are classified into either nominal or ordinal data to ensure that the correct encoding strategy can be applied.

The fifth step is to encode each string column according to its cardinality and ordinality. As mentioned before, this step is executed by default unless the user indicated that the data should not be encoded.

Finally, either the encoded DataFrame or the processed DataFrame will be returned. In both cases, additional information on the DataFrame is provided to the user (if indicated) to give some insight into the decisions made during the process, such as the technique applied to handle missing values or which encoder is applied for each string column.

The following sections give an in-depth description of how each module operates and which techniques are used.

4.2 String feature type inference

Before being able to handle data properly, it is vital to distinguish between different data types (e.g., integers, strings, datetime, etc.) that are present such that the framework can apply the corresponding cleaning steps to each column. In addition to detecting different data types, it must also be possible to detect specific string feature types in the data. This property allows the framework to assign specific processing steps that can benefit data enrichment and complexity reduction. Furthermore, it is also relevant to detect missing entries and potential outliers in the data to ensure that the framework is robust to these cases and can correctly identify and handle them accordingly. Therefore, this module aims to infer different data types and string features in the data, along with detecting missing values and outliers.

To achieve this goal, we build on top of `ptype`; the work of Ceritli et al. (2020) on data type inference using Probabilistic Finite-State Machines (PFSMs) [27]. As mentioned in 3.1.2, PFSMs are a class of mathematical models that represent a state machine where each transition has a certain probability associated with it. Ceritli et al. constructed a PFSM for each data type and missing values and set up an inference model which was able to outperform existing methods [27]. In their library, each PFSM is instantiated as a separate class which is part of a superclass that handles the model inference. Each PFSM class handles the construction of the state machine using the `greenery` library [6], which converts any regular expression into a state machine, and the initialization of the weights of each transition from the constructed state machine. Upon instantiation of a `ptype` session, users are able to fit the data into the model using `schema_fit()` and output the results using `show()`, as can be seen in Figure 4.2. The results include the inferred type, the normal values, the missing values, and the anomalous values (i.e., outlying data type) for each column in the data.

```
In [14]: schema = ptype.schema_fit(df)
         schema.show()
```

Out[14]:

	horsepower	curb-weight	length	width	height	engine-size	highway-mpg	price
type	integer	integer	float	float	float	integer	integer	integer
normal values	[100, 101, 102, 106, 110, 111, 112, 114, 115, ...	[1488, 1713, 1819, 1837, 1874, 1876, 1889, 189...	[141.10, 144.60, 150.00, 155.90, 156.90, 157.1...	[60.30, 61.80, 62.50, 63.40, 63.60, 63.80, 63....	[47.80, 48.80, 49.40, 49.60, 49.70, 50.20, 50....	[103, 108, 109, 110, 111, 119, 120, 121, 122, ...	[16, 17, 18, 19, 20, 22, 23, 24, 25, 26, 27, 2...	[10198, 10245, 10295, 10345, 10595, 10698, 107...
missing values	[?]	[]	[]	[]	[]	[]	[]	[?]
anomalous values	[]	[]	[]	[]	[]	[]	[]	[]

Figure 4.2: Using data type inference using `ptype` on the Car Evaluation dataset. Image taken from https://github.com/alan-turing-institute/ptype/blob/develop/notebooks/intro-to-ptype.ipynb.

Leveraging this technique onto our framework for (string feature) type inference is done as follows. We first create a set of new PFSMs based on nine different string features. Constructing new PFSMs is done by creating regular expressions that characterize these strings. After that, the PFSMs are instantiated as new classes and included in the list of available PFSMs that `ptype` use upon instantiation of a new session, where they are used as part of the generative model in the posterior distribution. This approach is advantageous because the inference model does not require alteration to handle these newly included PFSMs. Given this property, it becomes possible to perform missing value and outlier detection on the selected string feature types, which would otherwise be more difficult to implement using different methods. As mentioned by Ceritli et al. (2020), the PFSMs can be trained to tune the probabilities that were initially assigned to them (which is a uniform probability distribution) [27]. Training would allow the "correct" machine to give higher probabilities to the observed entries. However, we decided not to apply training on the string feature type PFSMs, as it already showed to perform well without training during implementation and evaluation. Furthermore, training does not insert or remove transitions between states or create new states based on the training data, making it less attractive to consider

relative to the added benefit. As a source of inspiration for deciding which string features were worth addressing, we made use of the variable types presented in Featuretools[1] that users can assign to data columns [4].

We implemented the following string feature type PFSMs in our framework:

Coordinate This string feature type represents GPS or Degree-Minute-Second (DMS) coordinates such as `N29.10.56 W90.00.00`, `N29:10:56`, and `29°10'56.22"N`. Coordinates can be distinguished from other string features based on the following characteristics:

- Two sequences of at most two digits and a third sequence which is a float with at most two digits before the decimal point.
- A character that separates the three sequences of digits (e.g., `.` or `:`). In the context of DMS coordinates, these characters are `°`, `'`, and `"` respectively.
- A cardinal direction at the beginning or at the end of the string (i.e., `N`, `E`, `S`, and `W`).

Day This string feature type represents the names of the seven days in the week, such as `Monday`. These names can appear in data in several formats. For example, `Monday` can be written as `Mon` and `Mo`. Days can be distinguished from other string features based on the following characteristics:

- A prefix of at least two characters, indicating the day of the week (e.g., `Mo` for Monday, `Th` for Thursday, etc.).
- The suffix `day`, if present.
- A distinct set of characters that comes after the prefix and before the suffix (e.g., if the string is `Thursday`, then `Th` should be followed by `urs`).

E-mail This string feature type represents all valid e-mail addresses from any domain such as `Jane@tue.nl` and `john.doe@hotmail.co.uk`. This feature can be distinguished from others based on the following characteristics:

- The character `@` which is between two sets of characters.
- A substring in front of the `@` (i.e., the name of the e-mail) which is composed of valid characters (e.g., the e-mail address `#@*%#$@hotmail.com` is invalid as the characters before the final `@` cannot be included in an e-mail name).
- A substring that comes after the `@` which is composed of valid characters and at least one dot inbetween those characters (e.g., `name@hotmail` is not a valid e-mail address as the domain name is incomplete).

Filepath This string feature type represents paths within a local system such as `C:/Windows/` and `C:/Users/Documents`. This string feature type can be written in a handful of varieties, making it slightly more challenging to make a robust regular expression out of. In this thesis, we consider filepath string features to be either a path with or without a file name (and possibly a file extension) at the end. Filepaths can be distinguished from other string features by the following characteristics:

- A series of substrings which are separated from each other using either `/` or `\` (e.g., `home/users`).
- Each substring cannot contain any of the following characters: `\/:*?"<>|`
- If present, a prefix that represents the root disk or a sequence of dots followed by a slash or a backslash (e.g., `C:/`, `../`, etc.).

[1] The variable types can be found at https://featuretools.alteryx.com/en/stable/getting_started/variables.html

Month This string feature type represents all twelve months in a year, such as `January` and `April`. Similar to the day string feature, months also may appear in several varieties. Months can be abbreviated based on their unique prefix (e.g., `January` can be abbreviated to `Jan`) or they can be accompanied by a day and year, which makes the string feature become a specific DateTime format (e.g., `January 1, 2000`). With this knowledge in mind, months can be distinguished from other string features based on the following characteristics:

- A prefix of at least three characters, representing a unique month (e.g., `Apr`).
- If present, the remaining substring that comes after the prefix (e.g., `il` comes after the prefix `Apr`).
- If present, a sequence of at most two digits before or after the month which represents a day in the month (e.g., `1 January` or `January 1`).
- If present, a sequence of at most four digits or a sequence with prefix `'` followed by two digits that comes after the month which represents the year (e.g., `January 2000` or `January '00`). Both day and year can be present at the same time.

Numerical There are various entries that are relatively easy for users to distinguish as numerical values but are usually inferred as strings by any type detection or inference technique due to certain non-numerical characters. For example, the entry `100-200` can be interpreted by users as a range between 100 and 200, whereas a program would identify this entry as a string because of the hyphen. It is therefore important to categorize such entries as a string feature type to properly handle and process them. We obtain numerical string features using any of the following characteristics:

- Between two sequences of digits, one of the following characters: `-+_/:;&'` A space or the substring `to` is also applicable (e.g., `100 to 200`).
- Before a single sequence of digits, any of the following words: `Less than`, `Lower than`, `Under`, `Below`, `Greater than`, `Higher than`, `Over`, `Above`.
- Before or after a single sequence of digits, any of the following characters: `<>+$%=`

Sentence This string feature type is composed of a sequence of words, typically found in datasets containing reviews or descriptions. It is slightly more challenging to express this feature type as a regular expression compared to the other string features because of its overlapping characteristics with regular string entries consisting of a couple of words. However, it is still possible to perform string feature type inference for sentences based on the following characteristics:

- A substring of characters followed by a space for at least five times (i.e., the entry is at least six words long).

URL This string feature type represents any link to a website or domain such as `https://www.tue.nl/` and `http://canvas.tue.nl/login`. The characteristics of this string feature type are similar to that of filepaths, with a few exceptions:

- An optional suffix which represents a certain protocol (e.g., `http://`).
- A series of at most four character sequences, separated from each other by a dot (e.g., `www.google.com` or `google.com`). Note that the last (pair of) sequence(s) contain(s) at most three characters.

Zip code This string feature type represents zip or postal codes from a handful of countries. Note that we can only infer zip codes containing non-numerical characters as numerical-only zip codes are much more difficult to infer using PFSMs without overlapping actual numerical features. Given that each country that uses non-numerical postal codes has a specific set of characteristics, we decided to use some of the regular expressions created by GitHub user jamesbar2[2].

After each string feature type PFSM is created and included to the instantiated `ptype` session, users are now able to infer these string features similarly to how data types are inferred in `ptype`.

4.3 Processing string features

Once the type inference determined that there are certain known string features in the data, it is now possible to process these further based on which string feature type was inferred. We can perform specific processing automatically because type inference is based on regular expressions, essentially describing the format of each string feature. This processing step is aimed at accomplishing at least one of the following goals for each string feature:

- Reduce overall string complexity by, e.g., removing common substrings.
- Assign or perform specific encoding techniques.
- Enhance the current dataset by extracting additional information from certain string features.

The techniques applied to each string feature type are related to at least one of these goals and vary in complexity based on string feature complexity and possible information extraction. For example, it may be more valuable to reduce the number of words in a sentence string feature to only the most relevant ones, whereas zip codes may be used to fetch additional data such as latitude and longitude values. Each technique was applied based on what is thought to be the most suitable technique to accomplish one of the goals mentioned above. After each string feature type is processed, the framework will also return a number alongside the data to indicate whether the data needs to be encoded and which encoding technique should be applied. The value of this number is based on the resulting data type after processing and the cardinality or ordinality of each string feature type (which is pre-defined for each type based on empirical observation of what each string feature represents). The techniques applied on each inferred string feature type are described in the remainder of this section.

Coordinate Recall that this string feature type is formatted with specific separators and a letter at the start or the end of the string. It is also possible that the entries are composed of two coordinates simultaneously. Thus, with $c \in \{N, E, S, W\}$ and $x \in [0..9]$, the following formats are possible:

- $c\ xx.xx.xx$, $xx.xx.xx\ c$, or $xx.xx.xx\ c\ xx.xx.xx\ c$
- $c\ xx:xx:xx.xxxx$, $xx:xx:xx.xxxx\ c$, or $xx:xx:xx.xxxx\ c\ xx:xx:xx.xxxx\ c$
- $c\ xx^\circ\ xx'\ xx.xxxx"$, $xx^\circ\ xx'\ xx.xxxx"\ c$, or $xx^\circ\ xx'\ xx.xxxx"\ c\ xx^\circ\ xx'\ xx.xxxx"\ c$

The goal for processing this string feature type is to transform each entry into a numerical representation of the coordinate. This numerical representative is also known as a latlong value (latitude, longitude, or both). Additionally, if two coordinates are given for each entry, we can extract additional information from this feature because these pinpoint the exact location in the world. The workflow of this process is depicted in Figure 4.3.

[2]The list of international postal codes can be found at https://gist.github.com/jamesbar2/1c677c22df8f21e869cca7e439fc3f5b.

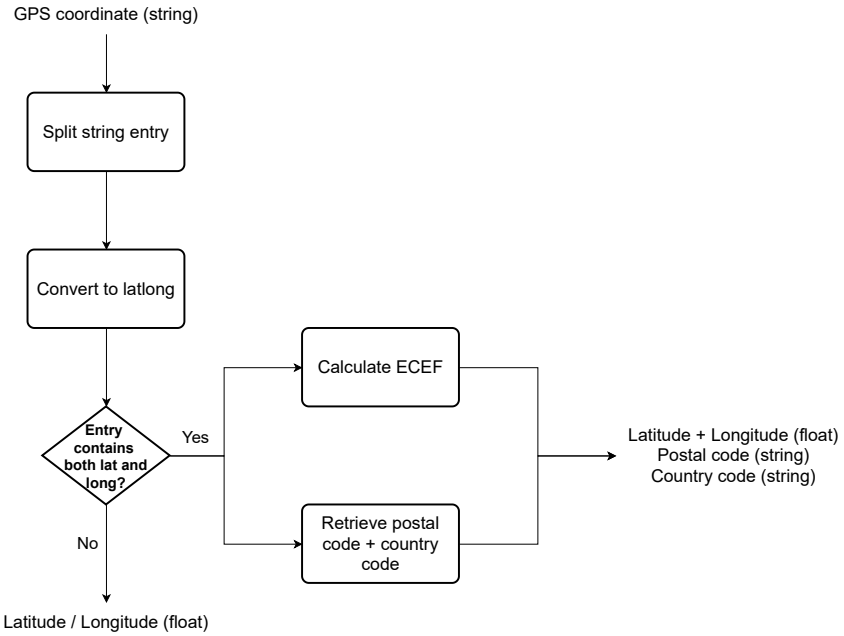


Figure 4.3: The overall workflow for processing coordinate string features.

In the first step, the string feature is split up into two separate parts for each coordinate in the entry, in which one part represents the cardinal direction of the coordinate, and the other part represents the numerical information. The part containing the numerical information is further split up into three separate parts at the indices where the separators are (e.g., `90.00.00` is split up into `[90, 00, 00]`). This results in a list consisting of several parts.

In the second step, the current format of the coordinate string feature is converted into the corresponding decimal latlong value. The string feature type is formatted using degrees, minutes, and seconds. This format can be converted to representative decimal values using the following formula[3]:

$$\text{decimal} = \begin{cases} -(\text{degrees} + \frac{\text{minutes}}{60} + \frac{\text{seconds}}{3600}) & \text{if } c \in \{S, W\} \\ \text{degrees} + \frac{\text{minutes}}{60} + \frac{\text{seconds}}{3600} & \text{otherwise} \end{cases}$$

In the final step, additional information is extracted if the string feature contains both the latitude and the longitude values. We can distinguish between entries containing only one or two latlong values based on the length of the list. Note that latitude and longitude values that appear in separate columns are seen as two separate coordinate columns containing either latitude or longitude values. Once this distinction is made, the following operations are performed:

- Including the Earth-Centered, Earth-Fixed (ECEF) representative of the latlong value in the data. This representative represents latlong values as x, y, z-coordinates on a three-dimensional graph. To calculate these coordinates, we use an approximation of the coordinate transformation in Hoffmann-Wellenhof et al. [41] which gives us the following equations:

[3]Taken from *Geographic coordinate conversion* at `https://en.wikipedia.org/wiki/Geographic_coordinate_conversion`

$$
\begin{aligned}
x &= \sin\left(\frac{\pi}{2} - \text{latitude}\right) \cdot \cos(\text{longitude}) \\
y &= \sin\left(\frac{\pi}{2} - \text{latitude}\right) \cdot \sin(\text{longitude}) \\
z &= \cos\left(\frac{\pi}{2} - \text{latitude}\right)
\end{aligned} \tag{4.1}
$$

Even though latlong values are simple to use with different applications, some users might prefer the ECEF representative as these values can be plotted in a three-dimensional graph. Both latlong and ECEF are returned to the user, who may choose between using either of them based on their preference. Additionally, the decision to include both relies on the fact that removing either the latlong values or the ECEF values takes less effort from the user than transforming these values manually.

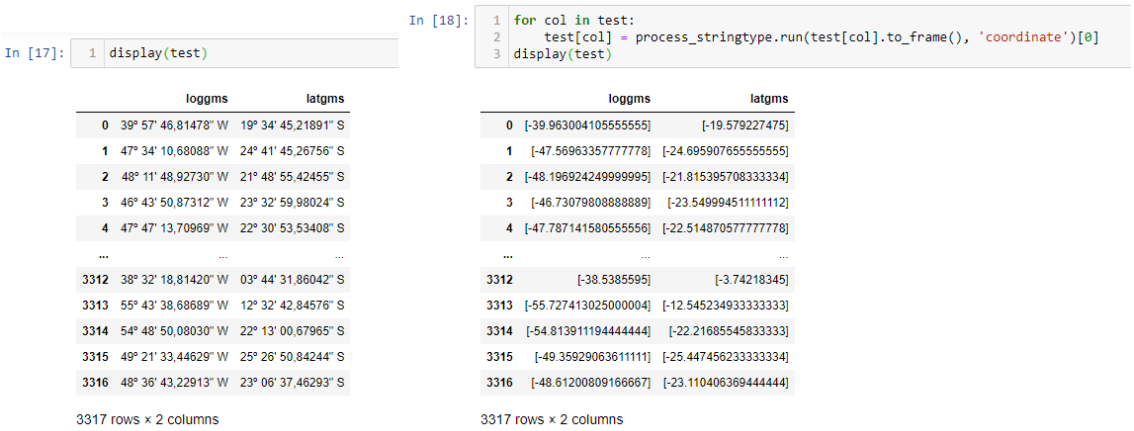
- Including additional information on the location of the latlong value in the data. The `geopy` library [5] is a client for several popular geocoding web services. It makes use of a selection of APIs in order to obtain additional information for a given input. In our case, we can communicate the latlong values to one of these APIs to obtain metadata about which country and postal code the latlong values point to. In case no country or postal code can be found, the resulting value is set to `unknown`. These additional features are included in the data to uncover latent relations between entries that are not observed in the latlong values, which could increase the overall performance of a machine learning model running on the data. Postal codes and country codes may be already included in the data, which is not necessarily a problem as the user can remove the additional information with little effort.

After these operations are performed, the string feature has been processed and additional features have been obtained. If the user decides to encode the data, the extracted postal and country codes will receive a nominal encoding in the final step of the framework. An example of how the data is transformed is depicted in Figure 4.4.

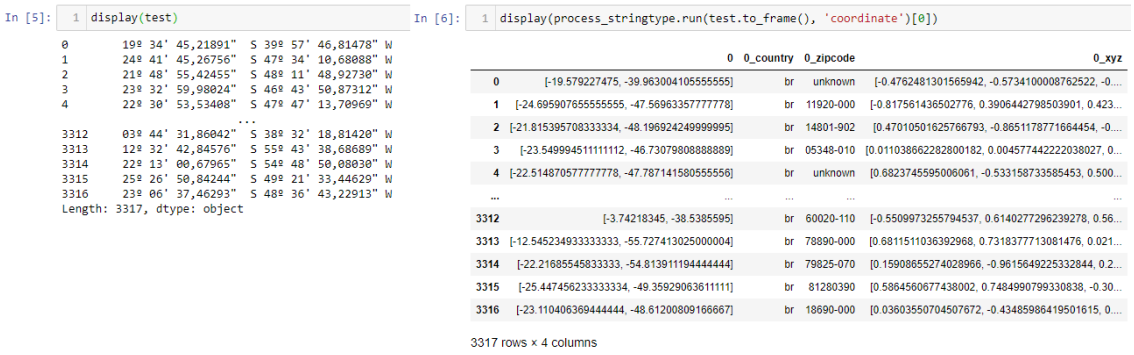
Day As this string feature type is the least complex out of all inferred string features, it is also the simplest to process. The aim of processing this string feature type is to remove as much redundant information as possible, as this may help reduce the encoding time while maintaining the same task performance. This aim is achieved by only considering the first two characters in each entry and removing the remaining ones. Considering only the first two characters for days is the most reduction that can be applied while still being able to make a distinction between each unique day of the week. If the user decides to encode the data, this string feature will receive a nominal encoding in the final step of the framework. An example of how the data is transformed is depicted in Figure 4.5.

E-mail The goal of processing this string feature type is to reduce its overall string complexity such that the overall dimensionality of the encoded strings is reduced. This aim is relatively simple to achieve since all e-mail addresses adhere to a specific format, namely `name@domain.topleveldomain`. The process is split up into two main steps.

The first step involves removing the longest common suffix of all entries. Removing the longest common suffix helps reduce string length, which consequently can impact string complexity. It also helps with emphasizing the main differences between all entries, as the proportion of differences between each entry is increased. In order to remove the longest common suffix, it is possible to make use of the built-in `os` library in Python. Using the library, one can call the function `os.path.commonprefix(list)` to obtain the longest common prefix of length n, which is a function that iterates over a list of strings and finds the longest common prefix. After the longest common prefix is found, slicing away the first n characters for each entry gives the desired result. To find the longest common suffix, one can reverse all the strings before calling



(a) Single coordinate per column.



(b) Both coordinates in a single column.

Figure 4.4: Coordinate string features before and after processing.

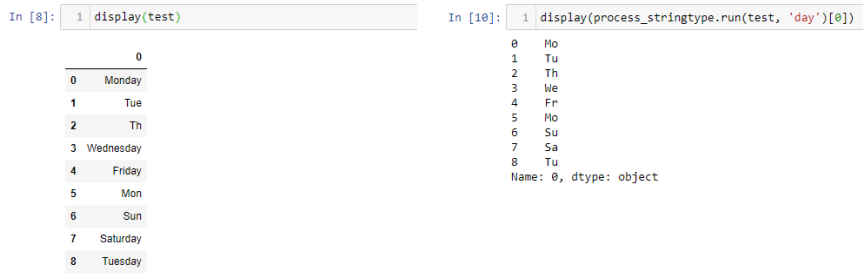


Figure 4.5: Day string features before and after processing.

`os.path.commonprefix(list)`. In terms of reducing string complexity, the worst-case scenario occurs when no common suffix was found, and the best-case scenario occurs when the longest common suffix is equal to `@domain.topleveldomain` and only the usernames remain in the list.

The second step removes any special characters from the remaining data and also the top-level domain if these were still present. The reason for removing the top-level domain is because this substring is believed to be the least beneficial in terms of performance and computation time.

After both steps are performed, the final string feature will at best be reduced to contain only the names of each e-mail address and at worst be reduced to the `name` and `domain`. If the user

decides to encode the data, this string feature will receive a nominal encoding in the final step of the framework. An example of how the data is transformed is depicted in Figure 4.11.

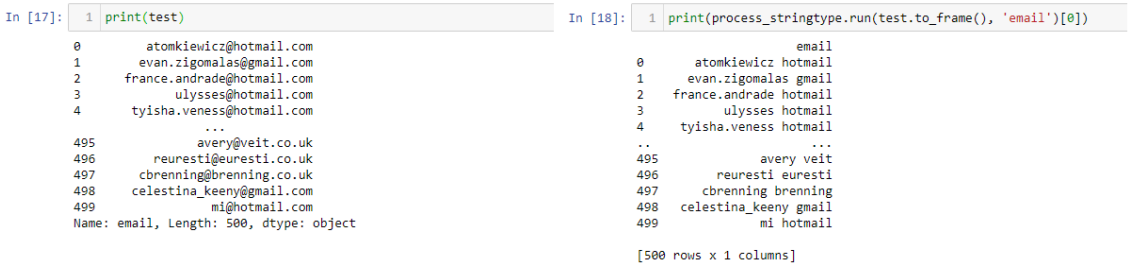
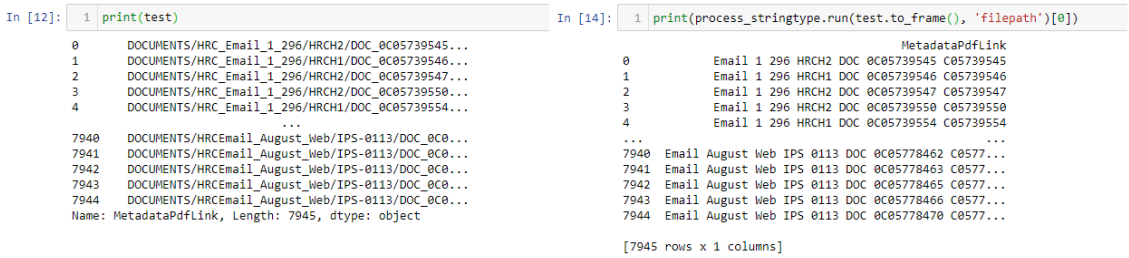


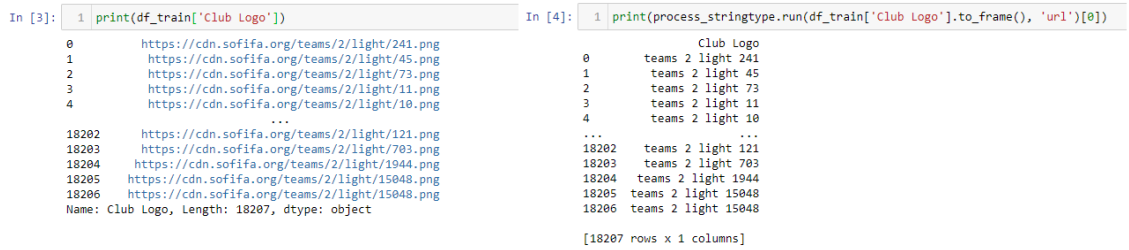
Figure 4.6: E-mail string features before and after processing.

Filepath and URL This process is aimed to reduce string complexity and is similar to how e-mail addresses are processed. Given that filepaths are formatted hierarchically, and URLs also adhere to a specific format, it is relatively simple to remove redundant substrings for both of these.

The procedure is split up into two separate parts. The first part removes the longest common prefix and suffix from all entries to emphasize the differences between all entries and reduce overall string complexity. Removing the longest common prefix and suffix is done using the same procedure as described for e-mail addresses. The second step is to replace all special characters with a space using regular expression splits and list filtering for consistency. The final result is a string feature with reduced complexity and consistent characters. If the user decides to encode the data, these string features will receive a nominal encoding in the final step of the framework. An example of how the data is transformed is depicted in Figure 4.7.



(a) Filepaths.



(b) URLs.

Figure 4.7: Filepath and URL string features before and after processing.

Month Given that a variety of formats of this string feature type can be inferred, several distinctions have to be made before the feature can be processed correctly. The goal of processing this string feature is to encode this feature into a numerical representation without loss of interpretability by the user. An overview of the process is depicted in Figure 4.8.

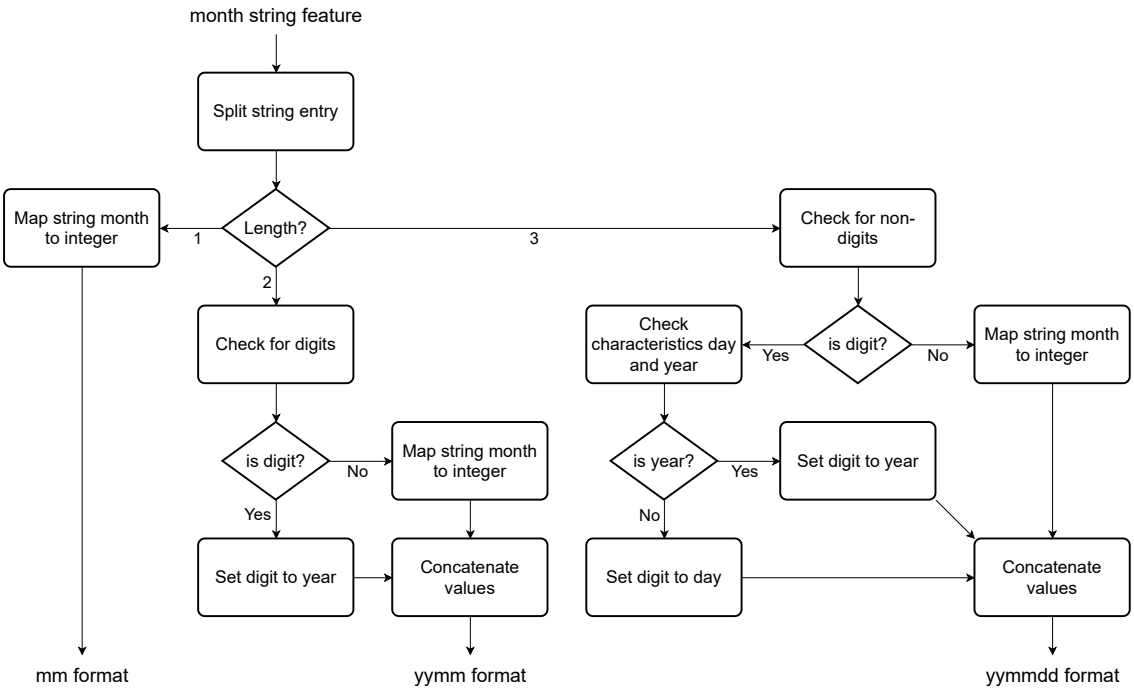


Figure 4.8: The overall workflow for processing month string features.

The string feature can take up any of the following formats:

- `month`: a string of at least length 3 representing the month (e.g., `January` or `Jan`).
- `yyyy month`, `'yy month`, `month yyyy`, `month 'yy`: The month and year, where the year is either all four digits or the last two digits of the year (e.g., `'21 Jan`). Note that the separator is not limited to a space.
- `dd month yyyy`, `dd month 'yy`, `month dd yyyy`, `month dd 'yy`: The day, month, and year of an entry, where days can also be a single digit value (e.g., `January 1, '21`).

The process starts by splitting and removing all special characters except for the apostrophe, resulting in a list with a length of at most 3. The next step is to identify what each component in the list may represent based on the length of the list.

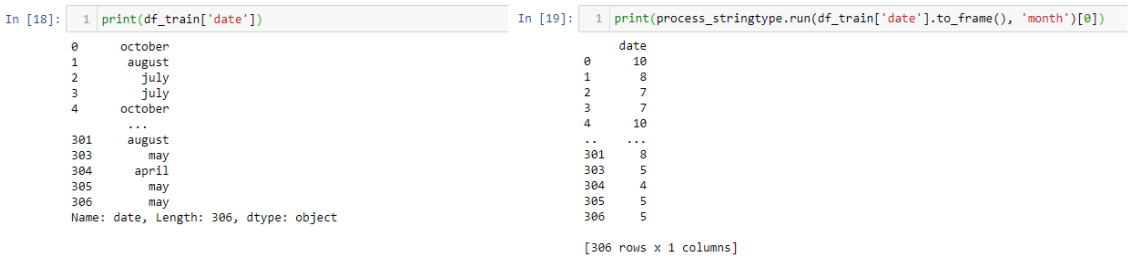
If the length of the list is 1, the list can only contain the month and the corresponding numerical representation of the month will be assigned to this entry.

If the length of the list is 2, the list will contain a month and year. In this case, the entries will be distinguished based on whether the value is numeric and whether the value contains an apostrophe. In any case, the last two digits of the year will be concatenated to the numerical representation of the month, and this new value will be transformed into an integer.

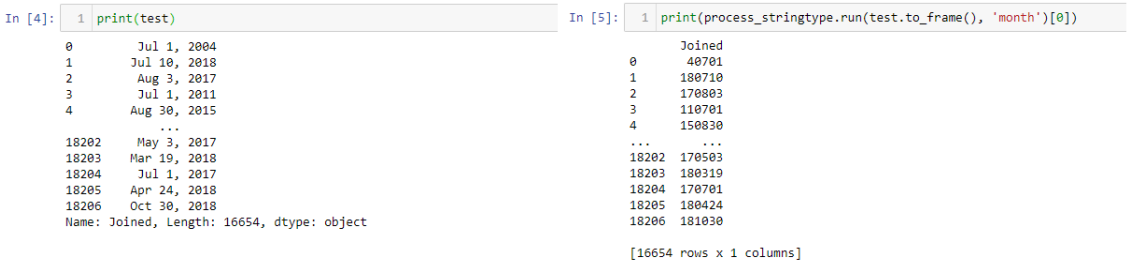
If the length of the list is 3, it means that the list is composed of a day, month, and year. It is necessary to distinguish between day and year, as these can occur in any particular order in the entry. Out of all three entries, the month will first be determined because it will be used to limit how many days this particular month has. Next, an item in the list will be defined as a year when there are four digits in the entry, when the item contains an apostrophe, or when a day has already been defined. The value is set as a day when none of the previous options apply and when

the value does not exceed the maximum value of the day of the corresponding month. Finally, all three digits will be concatenated and transformed into an integer.

After all three cases have been considered, each variety of the string feature will have been properly processed into their respective numerical values, i.e., `mm`, `yymm`, and `yymmdd`, respectively. As the string feature is already transformed to its numerical representation, no encoding would be required if the user requested so. An example of how the data is transformed is depicted in Figure 4.9.



(a) Only months.



(b) Day, month, and year.

Figure 4.9: Month string features before and after processing..

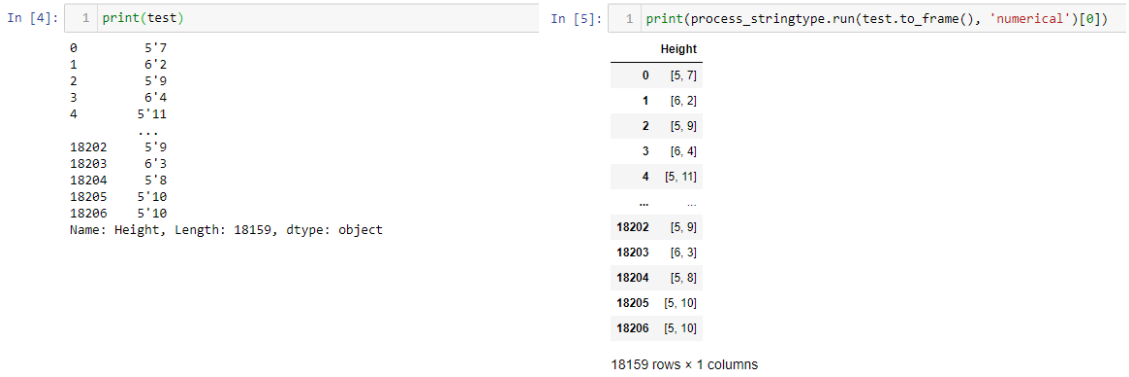
Numerical Recall that there exist numerical values that are inferred as strings because they contain special characters. Inferred numerical string features represent either an interval or a numerical value with at least one special character (e.g., `$500`, `80+3`). The goal for processing this string feature type is to determine the order of the values and to encode these accordingly.

The procedure consists of two steps. In the first step, the string feature format is determined by checking which part of the regular expression of the PFSM matches with the entry and is then processed accordingly. Making this distinction before processing ensures that the correct ordering of the values is not altered. The processed entries are coupled with their original value for the second step. Each case is processed in the following way:

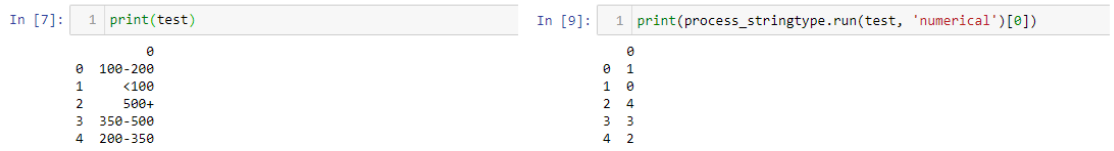
- Intervals such as `100-200`: The entries are split on the special character and the mean of the two numbers of each entry is taken using `numpy.mean`.
- Singular value intervals and values with special characters as a prefix or suffix such as `<100` and `20%`: Remove all special characters.
- Non-interval values such as `80+3`: Each entry is split based on the special characters, and each split is considered as a new feature in the DataFrame. For example, the value `80+3` is processed into `[80, 3]` and now represents two features instead of one. For these values,

one could also consider prepending all but the first number in the split with zeroes and concatenating the splits such that all entries have the same length. However, this idea was omitted since it requires more work without being beneficial in terms of performance.

In the second step, the tuples containing original and processed values of the interval data are sorted on the latter value using the built-in `sort()` function. The result is a list of tuples in the correct order, and the encoded values are simply the mapping of each original value to the index of the list where the value is positioned. As the string feature is already transformed to its numerical representation, no encoding would be required if the user requested so. An example of how different numerical string features are processed is depicted in Figure 4.10.



(a) Non-interval data.



(b) Interval data.

Figure 4.10: Numerical string features before and after processing.

Sentence The goal for processing sentence string features is to remove redundancy in the entry and make them more relevant for use in tabular data. The technique used to achieve this goal is the `nltk` word tokenizer [17], which takes a sentence and divides these into tuples containing each word and their associated part of speech. Then, every word associated with a noun is joined together with a space into a single string which is then passed on. The resulting entry is a group of nouns that supposedly represent the essence of the sentence and are ready to be encoded in the next step of the framework. If the user decides to encode the data, this string feature type will receive a nominal encoding in the final step of the framework. An example of the procedure is depicted in Figure 4.11.

Note that in the context of our thesis, we want to process sentences to be considered useful for certain tasks performed on tabular data without increasing the dimensionality of the data immensely. This statement implies that this processing step may not be the ideal procedure in the context of specific NLP tasks.

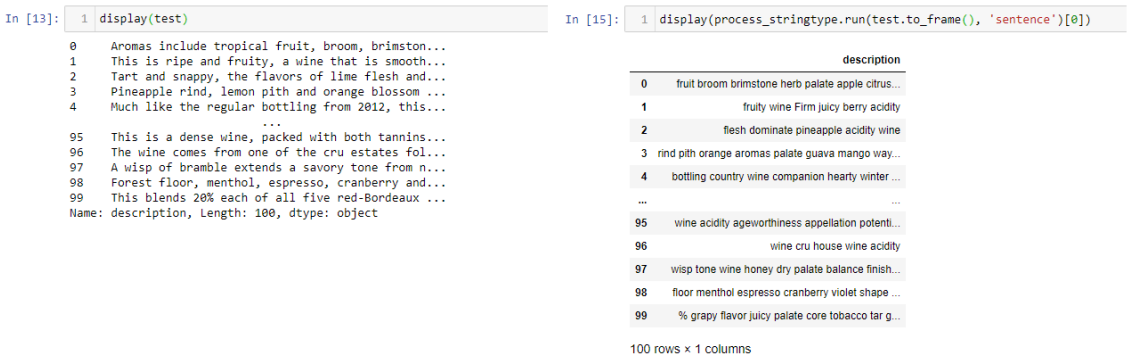


Figure 4.11: Sentence string features before and after processing.

Zip code Processing this string feature type is similar to that of coordinate string features, except that the first and second steps are unnecessary. The goal of processing this string feature type is to extract additional features from each entry to potentially discover latent relations that are not presented by zip codes. Using the `geopy` library, we can extract the following features for each entry:

- City
- Country
- Latitude and longitude

In case the API was not able to identify a feature corresponding to the zip code, the resulting value for that entry will be set to `unknown`. Additionally, we calculate the ECEF representative of the latlong value using the formulas described in 4.1 to provide users with the choice to use either one. If the user decides to encode the data, the zip code string feature will receive a nominal encoding in the final step of the framework. An example of how the data is transformed is depicted in Figure 4.12.

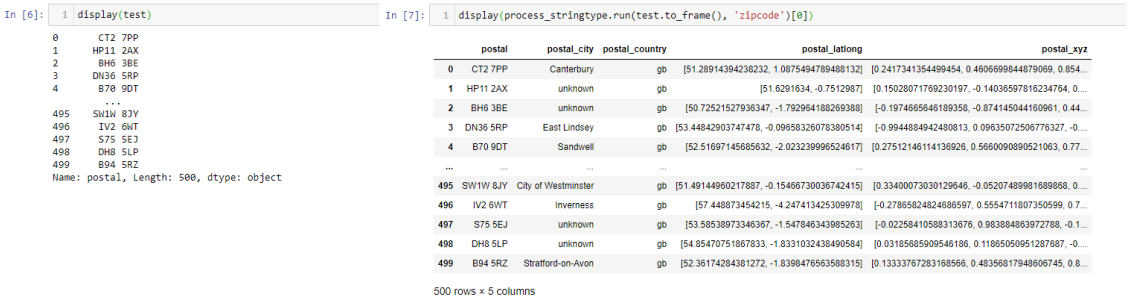


Figure 4.12: zip code string features before and after processing.

4.4 Ordinality detection

There will be string feature types that the PFSMs could not explicitly infer because they have an 'unknown' type. In this case, such string columns are inferred as standard strings. In order to make the most out of these unidentified string features, it is possible to determine whether the data in the column is ordered or unordered (i.e., whether the data is ordinal or nominal). Ordinal data is typically described as nominal data with a defined ranking, where the distance between each rank may vary [10]. Examples of ordinal string data can be found in surveys, where questions and possible answers are laid out to the participants in a Likert-type scale[4]. Recall from the literature review that encoding ordinal data using an order-based encoder helps to reflect the ordering and structure of the original data to the machine learning model and the user. Furthermore, it helps to reduce the complexity of the data after encoding when the correct encoding techniques are applied. For example, applying a one-hot encoder to all string columns regardless of ordinality would significantly increase the dimensionality of the data, whereas an approach to apply an ordinal encoder to all ordered data would result in a significantly lower dimensionality increase. Thus, in this step of the overall procedure, the goal is to predict which of the standard string columns are ordered such that these can be appropriately encoded in the last step. The prediction should be accurate and robust to changes in data characteristics such as length and values.

4.4.1 Choosing a model

To achieve this goal, we can make use of one of the classification models for ordinality prediction that were described in Section 3.2. At first, it seemed that the Bayesian approach proposed by Valera et al. (2017) was viable for our work to discover the statistical data type of a column [90]. However, this approach was not found to be the most optimal strategy for this framework. As mentioned in the previous chapter, their approach exploits the key idea that a latent structure in the data captures statistical dependencies among different objects. For this, the assumption is made that the input data has a latent distribution from which a classification can be derived. This assumption might be error-prone when nominal and ordinal data share the same latent distribution or when the correct latent distribution is not used. This limitation is observed in the results of the paper of Valera et al., where some columns are inferred as "categorical or ordinal", as can be seen in Figure 4.13.

Adult		German	
Attribute	Type	Attribute	Type
age (74)	ord.	status account (4)	cat.
workclass (8)	cat.	duration (69)	ord.
final weight	positive	credit hist. (5)	cat./ord.
education (16)	cat.	purpose (10)	cat./ord.
education num. (16)	cat.	amount	interval
marital status (7)	cat.	savings (5)	ord.
occupation (14)	cat./ord.	installment (5)	cat./ord.
relationship (6)	ord.	personal status (4)	cat.
race (5)	cat.	debtors (4)	ord.
sex (2)	binary	residence (3)	cat.
capital-gain	real	property (4)	cat./ord.
capital-loss	real	age (57)	count
hours per week (99)	cat./ord.	plans (3)	cat.
native-country (41)	ord.	housing (3)	ord.
		# credits (4)	ord.
		job (4)	ord.

Figure 4.13: Results from Valera et al. (2017) on the Adult and German datasets. Note that certain columns are inferred as `cat./ord.` as the latent distribution in the dataset can resemble both statistical types [90].

[4]The Likert scale is a psychometric scale that could, for example, be used to measure the level of agreement of participants on a particular subject. This scale has been developed by Rensis Likert in 1932 [58].

As an alternative solution, we propose to solve the prediction of ordinality in data as a binary classification task. In the previous chapter, we described four different classifiers that could be used to address the problem:

- **Decision tree classifier**: The simplest model out of the four, relatively easy to work with, and relatively fast in terms of prediction time. However, decision trees are non-robust to small changes in the data and are prone to overfitting due to their simplicity.
- **Random forest classifier**: Tackles the issue of overfitting for decision trees, but it is generally not memory efficient and its performance can still be affected by various data characteristics.
- **Gradient boosting classifier**: Tackles both issues of the random forest classifier and is known to outperform. The model is slightly more complex compared to the previous two.
- **Neural network classifier for tabular data**: Recent studies have shown that they can outperform gradient boosting classifiers in terms of accuracy either by constructing tree-like neural networks or by leveraging feature groups obtained by gradient boosting decision trees [50]. These models are much more complex in comparison to the previous three classifiers.

Out of these four classifiers, we opted for the gradient boosting classifier (GBC) due to its overall performance relative to the overall complexity of the model. The idea is to train the GBC on a set of labeled training samples and to store this trained GBC into the framework.

After choosing a classifier, it is essential to make sure that it is robust to differences between various datasets. We propose a heuristic implementation based on feature engineering that extracts relevant features from the data for classifying ordinality to tackle this problem. As mentioned in Section 3.3.3, feature engineering is the process of applying domain knowledge to raw data in order to extract specific characteristics and properties. The characteristics and properties can be retrieved from any dataset regardless of its composition, making it a valuable strategy to generate data samples from data columns that differ in length and context.

4.4.2 Implementation

The process of classifying whether data is ordered or unordered can be divided into two parts, namely (1) feature extraction from the data using feature engineering and (2) classification of the obtained sample using a trained GBC. An overview of the process of the workflow is depicted in Figure 4.14.

In the first step, eight features are extracted from a column of string data. Each feature is constructed based on how relevant it would be to solving the problem. Some of these features are general, whereas other features may resemble characteristics typically found in either ordinal or nominal data. After each feature is extracted, they are appended to a list that represents a single data sample. A brief description is provided regarding each feature's relevance for classification and how these features are obtained from the data.

- *The total number of rows in the column*: It is possible that the number of rows in combination with other extracted features can increase the performance of the classifier. Obtaining this value is done by measuring the length of the column.
- *The number of unique values in the column*: In general, nominal data tends to vary more in cardinality as opposed to ordinal data. Furthermore, some ordinal data columns tend to adhere to Likert-scale characteristics regarding the possible number of unique entries, which also limits its cardinality. The value is obtained by counting all unique entries in a column.
- *The ratio between the number of unique values and the total number of rows*: As a rule of thumb, some domain experts tend to classify data as ordinal when the ratio between the unique values and the total number of rows is at most 0.05. The ratio for nominal data

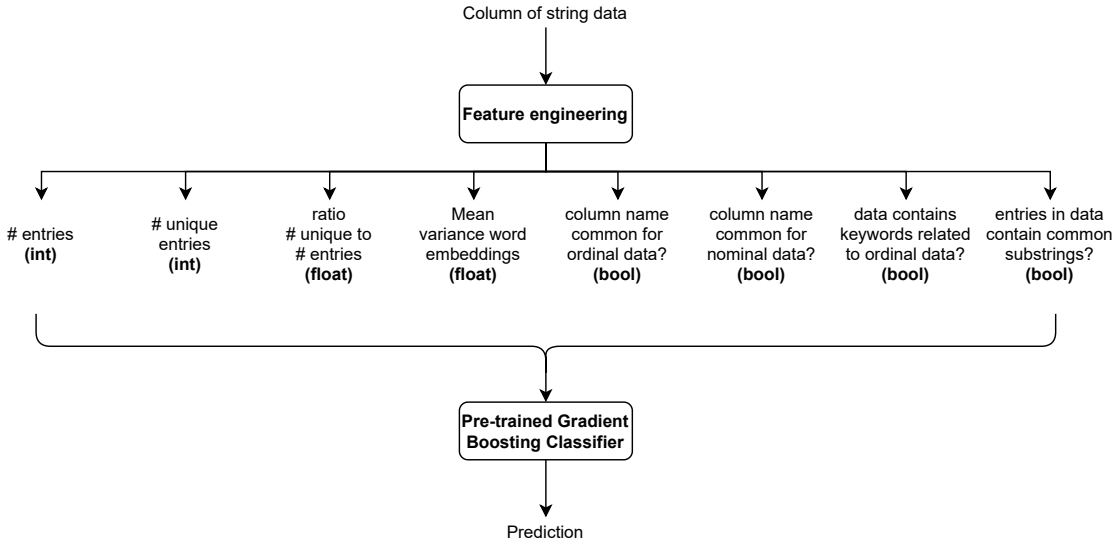


Figure 4.14: Workflow of ordinality prediction for a given string column.

tends to be at most 0.2. As a result of this rule of thumb, we extract the ratio for use in the classifier by dividing the number of unique values by the total number of rows.

- *The mean of the variance of the distance between the word embeddings of unique entries*: The idea behind extracting this feature is that the word embeddings of certain entries showcase interesting linear substructures in the word vector space. By taking a pre-trained word vector space, the classifier may distinguish between ordered and unordered data based on differences in the substructure. The first step is to split each entry into a set of words, which are then embedded using a pre-trained word vector space. This work makes use of the Wikipedia word vector space by GloVe, which consists of over 400 000 words in the vocabulary embedded into a 50-dimensional vector space [78], to assign each word in an entry to a vector. A random point in the vector space will be assigned to a word if it does not appear in the pre-trained corpus. Next, the mean of all dimensions for each word vector in the entry is calculated such that all word vectors of the entry are now represented as a single point in the 50-dimensional vector space. After this is done for all entries in the column, the variance between each dimension is calculated. Finally, the mean of each dimension is taken, and the resulting value is a single float value that can be used to potentially distinguish ordered data from unordered data.

- *Whether the column name is commonly used in ordinal data*: There are a set of keywords that can commonly be found in column names for ordinal data. Examples of certain keywords include `grade`, `stage`, and `opinion`. By checking whether a column name is contained within one of those keywords and vice versa, we can tell whether the data in the column is more likely to be ordinal or not. Note that keywords for column names used in this work are created based on domain expertise, which means that results may vary when other keywords are used.

- *Whether the column name is commonly used in nominal data*: The approach of extracting this feature is similar to checking ordinal traits in the column name, except that the names obtained via domain expertise are now commonly used in nominal data. Typical nominal column names include `address`, `city`, `name`, and `type`. Again, since the names are based on domain expertise, results in performance may vary when other names are used.

- *Whether the unique entries contain keywords that are commonly found in ordinal data*: Extracting this feature is similar to the two aforementioned techniques, except that ordinality

will now be implied based on keywords commonly found in ordinal data. The keywords that were used in this work are adjectives and nouns that are typically found in Likert-scale questionnaires [64].

- *Whether the unique entries share a number of common substrings*: Ordinal data tends to contain entries with overlapping substrings. For example, the strings `disagree`, `agree`, and `wholeheartedly agree` all contain the substring `agree`. By checking whether there are common substrings in the data of sufficient length, the classifier may associate the occurrence of substrings with the implication that the data is ordinal.

In the second step, the trained GBC is loaded from memory, and the new data is passed to the model. The model then predicts the original string column based on the features in the data. The model is trained on a meta-dataset consisting of the aforementioned features from 149 string columns (81 ordinal and 68 nominal string columns) from a total of 29 datasets. The meta-dataset took at most two hours to create and the GBC took less than ten seconds to train, which is feasible considering that these operations only needed to be performed once. Nevertheless, it could be efficiently retrained on new data when desired.

Both steps are performed on all string columns in the data. The result is a list of integers of either 0 (column classified as ordinal) or 1 (column classified as nominal), where the i-th value in the list corresponds to the i-th string column processed in this module. This list is used to assign the appropriate encoding technique to each string column.

4.5 Encoding

After all inferred string feature type columns are processed and the ordinality of other string columns is determined, the string data is ready to be encoded based on the result of the previous step. Each string column is encoded using either a nominal encoder or an ordinal encoder. Generally, encoding string data to a numerical representation is required for most machine learning models as they cannot deal with non-numerical values. This module aims to encode all string columns properly while ensuring a reasonable dimensionality in the resulting dataset. This goal can be split up into two parts. The first part of the goal is ensuring that nominal string columns are encoded in an efficient and scalable manner while retaining (most of) the information, regardless of their cardinality. The second part of the goal is ensuring that ordinal string columns are encoded such that the actual order between different categories is properly captured since this might prove beneficial for the performance of a machine learning task. The approaches to achieving these sub-goals and their implementation are described in the upcoming subsections. An overview of the workflow of the encoding step is depicted in Figure 4.15.

Note that the string data is only encoded if the user indicated that this should happen. Giving the users the option not to encode the string data allows them to apply an encoding technique themselves and deploy the library in an automated pipeline. Furthermore, this option allows users to analyze the cleaned data after being processed instead of performing tasks on them shortly after.

4.5.1 Nominal encoding

Since the data potentially consists of morphological variants or has a high cardinality, the most optimal choice for encoding the data in an automated setting would be to make use of the `dirty_cat` library by Cerda et al. [26]. The significant benefit these encoders offer compared to the standard nominal encoders is that they take morphological variants into account. These morphological variants are, for example, different string categories that represent the same entity. `dirty_cat` can capture and represent these similarities in the encoded data. Furthermore, the high-cardinality string encoders of `dirty_cat` ensure that the dimensionality of the encoded data does not exceed a certain number of dimensions, which reduces the overhead and complexity of the resulting data

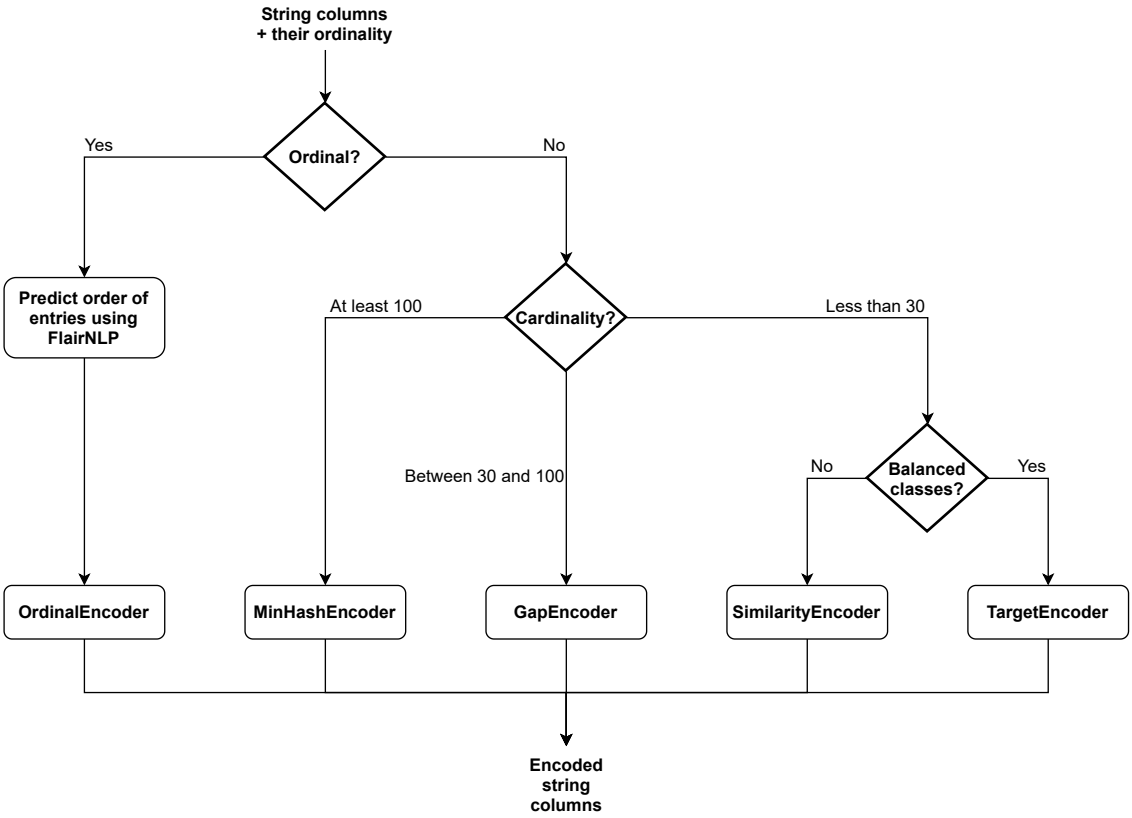


Figure 4.15: Workflow of encoding a given string column and the passed or predicted ordinality.

while maintaining most of the relevant information needed for learning algorithms. In addition to the `dirty_cat` encoders, the target encoder is also implemented when the target classes are balanced, i.e., when the frequency of each target class is relatively the same. This work considers target classes as balanced when the standard deviation of all class frequencies is at most half the mean of all class frequencies. The most-fitting encoder is chosen based on a rule mentioned in the paper on encoding high-cardinality strings by Cerda et al. [26]. The authors state a possible rule for encoding tabular data into an AutoML pipeline, which is to apply the one-hot encoder for low-cardinal data (which is a cardinality of at most 30) and to apply the Gamma-Poisson encoder or min-hash encoder otherwise [24]. In this work, we decided to modify this rule in the following manner:

- When the cardinality is lower than 30 and the target values are balanced: encode data using the target encoder
- When the cardinality is lower than 30 and the target values are imbalanced: encode data using the similarity encoder
- When the cardinality is between 30 and 100: encode data using the Gamma-Poisson encoder
- When the cardinality is at least 100: encode data using the min-hash encoder

We opted for using the similarity encoder over the one-hot encoder because the string data may still be prone to morphological variants at this cardinality. In that case, the similarity encoder is slightly more useful in highlighting similarities in the encoded data. Additionally, the target encoder was chosen over the similarity encoder in some cases, as it appeared in the past to perform better on target classes that are relatively balanced during the first evaluation iteration. As for the

high-cardinality encoders, the Gamma-Poisson encoder is used for its interpretability possibility when the cardinality is at most 100. However, as this encoder does not scale well, the min-hash encoder is used for higher cardinalities since it scales significantly better. An example of the impact that these sophisticated encoding techniques can have on the dimensionality of the data is depicted in Figure 4.16.

	category
0	Poetry
1	Narrative Film
2	Music
3	Film & Video
4	Restaurants
...	...
766	MY LIFE IN THE MISSISSIPPI DELTA
767	so get on board!
768	Head in the Clouds
769	Post Production
770	a bike quest in New Zealand

771 rows × 1 columns

(a) Before encoding.

	category_0	category_1	category_2	category_3	category_4	category_5	category_6	category_7	category_8	category_9
0	0.086652	0.062062	5.874149	0.054713	0.057286	0.064107	0.070079	0.065312	0.066256	0.099383
1	0.075818	0.414538	0.888601	9.378475	0.064283	0.061347	0.088099	5.422764	0.082638	2.023436
2	0.050808	0.056476	0.056359	4.492969	0.051590	0.054589	0.052861	0.074767	0.052049	0.057532
3	0.075903	0.051503	0.066372	0.054057	0.055846	0.050676	0.064766	14.901252	0.083563	0.096061
4	0.868383	0.067667	0.163613	11.590447	0.058381	0.602130	0.187742	0.069705	0.196428	0.195506
...	...	...	...	...	...	...	...	...	...	...
766	8.927545	0.069997	0.066716	1.053702	8.353297	0.099871	27.564985	0.668096	0.125548	0.070243
767	1.818154	17.861351	0.076877	0.061080	0.084169	0.055357	0.394864	1.413011	0.076715	1.158421
768	1.101141	0.081357	1.348013	0.080782	21.112024	0.053198	0.092909	1.926156	0.088782	0.115638
769	0.057643	0.054335	1.762490	14.279150	0.057750	0.053363	0.057193	0.061196	0.070850	5.046030
770	0.074600	0.066514	0.290505	1.720342	0.094906	6.151156	20.739945	0.121027	10.159437	0.081568

771 rows × 10 columns

(b) After using Gamma-Poisson encoder.

	category_0	category_1	category_2	category_3	category_4	category_5	...	category_765	category_766	category_767	category_768	category_769	category_770
0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...
766	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
767	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
768	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
769	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0
770	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0

771 rows × 771 columns

(c) After using one-hot encoder.

Figure 4.16: An example of the difference in dimensionality that the `dirty_cat` encoders offer. Notice how the dimensions the Gamma-Poisson encoded data are limited to 771 by 10, whereas the dimensions of the one-hot encoded data scales with the number of unique entries.

4.5.2 Ordinal encoding

Ordinal data is encoded using the built-in ordinal encoder from `scikit-learn` [77]. The benefit of this encoder is its scalability regarding dimensionality and speed while still being indicative of ordering in the categories. Typically, users can pass a list to the encoder that contains each category in the correct order to ensure that the correct ordering is maintained. However, in an automated setting with no user in the process, the encoder takes the lexicographical order of entries, which is not always the correct order.

To counter this issue, we propose an approach to determine the order in ordinal data automatically such that this order can be given to the ordinal encoder. The approach is to make use of the text sentiment intensity analyzer provided by `FlairNLP`, which is typically used to analyze how intense an input string is in terms of its positivity and negativity. As mentioned in Section 3.4.4, the sentiment intensity analyzer predicts the sentiment for a given input and how intense it is for a given input while taking the letter and word sequences into account. Since this classifier is based on a character-level LSTM neural network, their pre-trained network can handle negations, intens-

ifiers, errata, and out-of-vocabulary (OOV) words. These properties make the classifier an ideal candidate for determining the order in ordinal data when the categories are based on sentiments and when the categories may still consist of slight spelling mistakes.

In our approach, all unique entries of an ordinal column are passed to the sentiment analyzer. The sentiment analyzer then returns a value between 0 and 1 and a label that indicates whether the string is either negative or positive. The results of the sentiment analyzer are processed into numerical values and are coupled with the corresponding strings. Afterward, the list of tuples is sorted on the numerical values, and the original strings are extracted in the resulting order. The result is a list of ordered strings using `FlairNLP` from the most negative sentiments to the most positive, which is also the ordering that will be passed on to the ordinal encoder. An example of how the ordering in a list of unique entries is determined is depicted in Figure 4.17.

	SteamMachinesConceptLike
0	I really like the idea.
1	I'm rather neutral about it.
2	I like the idea.
3	I don't like the idea that much.
4	I don't like the idea at all.

(a) Input data.

	entry	sentiment	value
0	I really like the idea.	POSITIVE	0.998494
1	I'm rather neutral about it.	NEGATIVE	0.998065
2	I like the idea.	POSITIVE	0.996989
3	I don't like the idea that much.	NEGATIVE	0.999953
4	I don't like the idea at all.	NEGATIVE	0.999987

(b) Results from FlairNLP.

	entry	value
0	I don't like the idea at all.	-0.999987
1	I don't like the idea that much.	-0.999953
2	I'm rather neutral about it.	-0.998065
3	I like the idea.	0.996989
4	I really like the idea.	0.998494

(c) Processing and sorting results.

Figure 4.17: An example of applying `FlairNLP` on a sample of data to determine the order.

4.6 Other modules

This section describes other modules in the library that are implemented. Despite the necessity of implementing these modules to ensure robustness in the framework, they are not the main focus of this thesis.

4.6.1 Imputing missing values

The tool provides an imputation of missing values for each column. An overview of the imputation process is depicted in Figure 4.18.

During the stage where each data type is inferred for each column, the PFSMs also check whether the columns contain a set of characters that are used to represent missing values (e.g., `?` and `na`). The procedure starts by converting any of the inferred missing values into `numpy.nan`. This step is done to ensure that all possible missing values are converted to a single value.

After converting the missing values, the proportion is calculated between the number of rows containing missing values and the total number of rows. If this proportion is lower than 0.05, we can delete the rows containing missing values without significantly affecting the model's performance.

If the proportion is 0.05 or higher, we check whether the missing values are either MCAR or MAR/MNAR. This check is done by performing Little's MCAR test as described in Section 3.5. The implementation of Little's MCAR test has been derived from the Python implementation created by Rianne Schouten[5]. If the missing values are most likely MCAR, we know that the applied imputation technique will not influence the performance of the model. In this case, the data is imputed using the mean imputation for numerical values and the mode imputation for string values.

If the missing values are not MCAR, it means that the data is either MAR or MNAR. Since it is not possible to distinguish between MAR or MNAR without input from the user, the missing values are imputed using the iterative imputer of `scikit-learn` [77]. This imputation technique

[5]The original implementation can be found at https://github.com/RianneSchouten/pymice/blob/6ff7be8d7455adb45ee88456c289e7009f34a034/pymice/exploration/mcar_tests.py#L40.

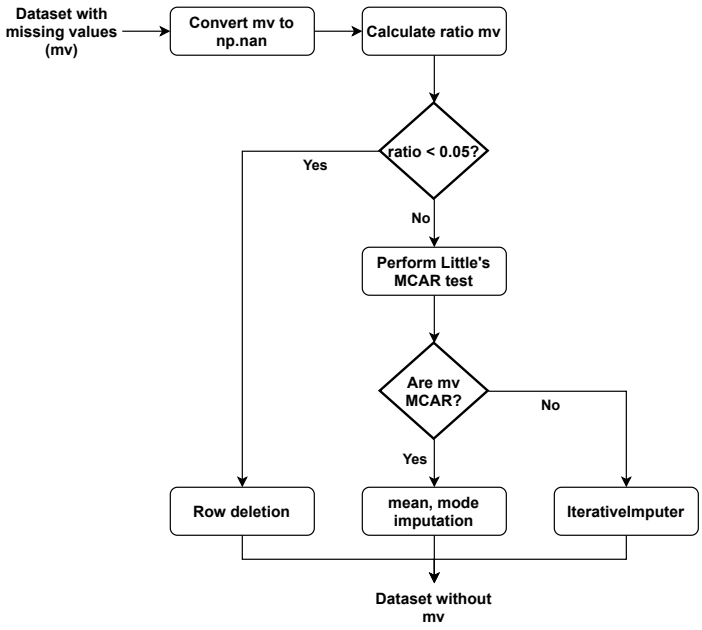


Figure 4.18: Workflow of missing value imputation.

was found to be one of the few that worked for both cases without possibly introducing a significant bias in the data.

4.6.2 Handling errata and data type outliers

The tool provides outlier handling for string data to ensure that small errata in categories are repaired. A string entry is considered an outlier when it has a string similarity score with another string entry with a significantly higher frequency in the data. In our work, the string similarity between two strings is calculated with the n-gram similarity with $n = 3$, which splits both strings into consecutive n-grams and calculates the proportion of the intersection and union of both string grams. For example, the 3-grams of strings $s_1 =$ `example` and $s_2 =$ `exams` are { `exa`, `xam`, `amp`, `mpl`, `ple` } and { `exa`, `xam`, `ams` } respectively. The 3-gram similarity of both strings is then

$$\text{sim}_{3\text{-gram}}(s_1, s_2) = \frac{|3\text{-gram}(s_1) \cap 3\text{-gram}(s_2)|}{|3\text{-gram}(s_1) \cup 3\text{-gram}(s_2)|} = \frac{|\{\texttt{exa}, \texttt{xam}\}|}{|\{\texttt{exa}, \texttt{xam}, \texttt{amp}, \texttt{mpl}, \texttt{ple}, \texttt{ams}\}|} = \frac{2}{6}$$

This measure is used to find pairs of strings with the highest similarity. After that, the ratio of frequencies is calculated for each pair of strongly similar strings. Given that this module was not in the focus of the thesis, the set thresholds are based on rules of thumb and empirical evaluation. More precisely, the thresholds for string similarity and the frequency ratio have been set to at least 0.75 and lower than 0.05, respectively. Future work could investigate a more clever approach to obtain more optimized results. If the potential outlier passes both thresholds, it is most likely an outlying value, and the most similar string replaces the outlier.

In addition to outlier handling for string data, data type outliers (e.g., a single string entry in an integer-only column) for other columns are also handled to ensure robustness in later steps and during model training. The framework first checks whether any non-outlying values are similar to the outlier according to the same criteria mentioned for the string outliers. If at least one similar entry is found, the most similar entry replaces the outlier. If no similar entries are found, the outlier is replaced by a random non-outlying value. In this case, we assume that the outlier is a missing value whose missingness is MCAR.

Chapter 5

Evaluation

This chapter covers the evaluation of the framework as a whole and all relevant components of the framework described in the previous chapter individually. For each component, we describe the goal of the experiment, how the experiment is set up, and how success is measured. Furthermore, the results of the experiments are reported and discussed. All experiments are performed on a machine with an Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz and 16 GB RAM. In order to allow for reproducibility of the results, a description of all the datasets that were used is provided in Appendix B and the source code of the framework can be found at `https://github.com/ml-tue/automated-string-cleaning`.

5.1 Global framework evaluation

5.1.1 Experiment set-up

To evaluate the overall performance of the framework, we cleaned ten datasets containing either a classification or regression task and at least one string feature or standard string column. A brief description of each dataset and which model is used to solve the task are described in Table 5.1 and Appendix B. A gradient boosting classifier or regressor (both using default hyperparameter settings) is used depending on the prediction task that needs to be solved, and either the accuracy or mean absolute error (MAE) is reported. The target variable of each dataset is predicted using stratified five-fold cross-validation, where the average score is compared against the average score of a baseline process. The baseline process entails basic encoding techniques (i.e., using either ordinal encoding only or target encoding only for all categorical columns) and imputation techniques for missing values (i.e., mean or mode imputation, depending on whether the data is numerical or categorical). The one-hot encoder was not used for the baseline because some datasets contain high-cardinality features, making comparisons infeasible. After the results are obtained, all values are transformed around the mean of the baseline for each dataset using the following formula:

$$x'_D = \begin{cases} \left(\frac{x_D}{\mu_B} - 1\right) \cdot 100 & \text{if metric} = \text{accuracy} \\ -\left(\frac{x_D}{\mu_B} - 1\right) \cdot 100 & \text{if metric} = \text{MAE} \vee \text{time} \end{cases} \tag{5.1}$$

Where D is the list of results, B is the list of results of the baseline, x_D is item $x \in D$, x'_D is the transformed value of $x \in D$, and μ_B is the mean of all values in B. Note that the baseline results are also transformed using this formula. In case the results from the MAE or running times are transformed, the transformation is negated since comparisons of MAEs and running times are based on "the lower, the better", whereas accuracies are compared based on "the higher, the better". By applying the transformation to the results, it becomes visible whether the framework offers any improvements and how well these improvements are in proportion to the baseline. All

the results on the left of the baseline indicate a percentual performance decrease compared to the baseline, whereas results on the right of the baseline indicate a percentual performance increase compared to the baseline.

This evaluation aims to discuss the effectiveness of the framework in comparison to manual cleaning using common practice and whether it is a feasible option to automate string cleaning.

Dataset	Number of rows	Number of numerical columns	Number of categorical columns	Average cardinality	Number of missing values	Target column
automobile	205	16	10	6.10	5356	price
fifa	18207	45	44	988	79089	Value
HR-analytics	19158	3	11	17.6	20733	target
HR-employee-attrition	1470	26	9	3.44	0	MonthlyIncome
mushrooms	8124	0	23	5.17	2480	class
registered-companies*	1992170	3	14	488796	2110412	COMPANY_STATUS
SF-crime*	878049	2	7	59062	0	Category
StudentsPerformance	1000	3	5	3.4	0	writing score
winemag-130k	129971	3	11	232612	26909	points
xAPI-Edu	480	4	13	5.46	0	Class

* Due to the size of these datasets, a subset of 100 000 random samples is used for evaluation

Table 5.1: Description of the datasets used for the global framework evaluation.

5.1.2 Results

The results of the evaluation are depicted in Figure 5.1 and Table 5.2. We observe that our framework provides at least the same performance, if not better, as the baseline and the target encoder for all the datasets. This observation indicates that our framework is generally a suitable and automated alternative to string handling and data cleaning. Upon closer inspection of the datasets and the corresponding components, it appears that a significant performance improvement can be observed when the data consists of columns with high-cardinality strings or balanced target values. These improvements are most likely due to the processing of string features and the novel encoding techniques, as some of these were also shown to give promising results in their paper [24, 25].

The framework performs slightly worse than the target encoder on the HR-analytics and winemag datasets. For HR-analytics, the reduced performance may be caused by folds that are disadvantageous to the framework, as the most significant performance difference between methods is lower than half a percent. Upon closer inspection of the winemag dataset, it appears that the more unsatisfactory performance is mainly related to the encoding technique applied to specific columns. More specifically, the target encoder may be optimal for certain high-cardinality columns compared to the Gamma-Poisson and min-hash encoders due to a mismatch between the data and the methodology of these encoders (i.e., focus on string similarity between entries). As the Gamma-Poisson and min-hash encoders both rely on substring counts and string similarities, it might be possible that incorrect relations are established during model fitting, resulting in slightly more unsatisfactory performance than the target encoder.

In general, it appears that most datasets suffer from outlier folds with significantly more unsatisfactory results compared to the other folds. These outliers could result from numerical outliers in the data or unfortunate splits during training. As this might impact some of the results significantly, it is worth considering mitigating this variance by, for example, shuffling the samples.

An alternative approach (which does not require shuffling the data) to confirm performance differences regardless of outlying folds is to run a statistical test to determine whether the difference between the three methods on the current results is statistically significant. In this statistical test, the results from the classification and regression tasks are computed separately, implying that the statistical analysis is conducted on three populations with 25 paired samples and a family-wise

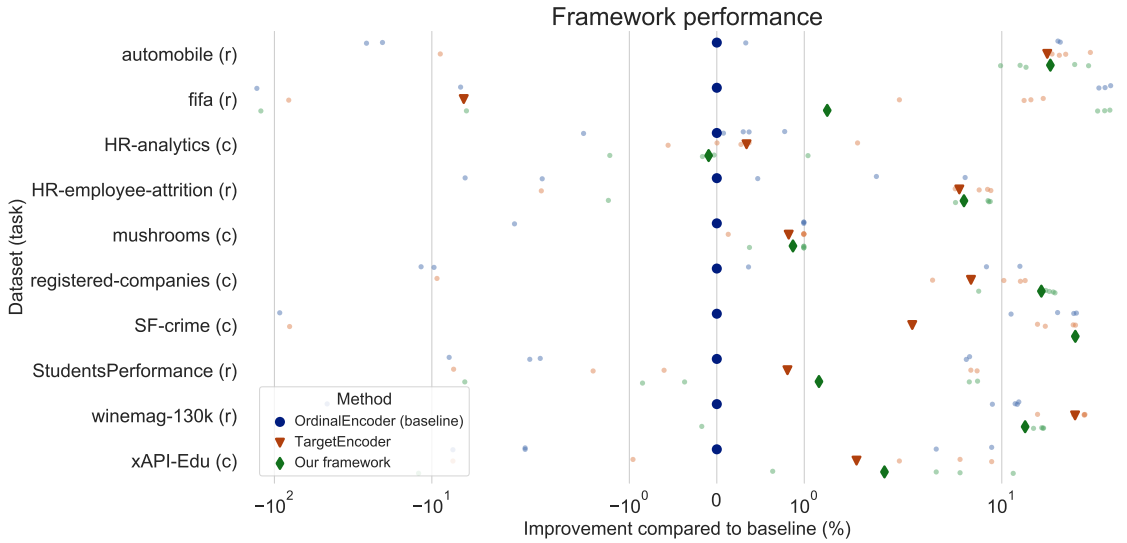


Figure 5.1: The relative performance of the framework against baseline preprocessing using the ordinal encoder and the target encoder on regression (r) and classification (c) tasks.

Dataset	Method	Accuracy
HR-analytics	OE	0.783 ± 6.22e-3
	TE	**0.786 ± 5.57e-3**
	Ours	0.782 ± 6.30e-3
mushrooms	OE	0.990 ± 1.50e-2
	TE	0.999 ± 3.00e-3
	Ours	**0.999 ± 2.00e-3**
registered-companies	OE	0.717 ± 6.91e-2
	TE	0.763 ± 6.19e-2
	Ours	**0.845 ± 3.90e-2**
SF-crime	OE	0.766 ± 0.358
	TE	0.787 ± 0.320
	Ours	**0.991 ± 1.12e-3**
xAPI-Edu	OE	0.652 ± 3.64e-2
	TE	0.663 ± 3.58e-2
	Ours	**0.665 ± 5.21e-2**

(a) Classification tasks

Dataset	Method	MAE
automobile	OE	2883 ± 601
	TE	2322 ± 436
	Ours	**2295 ± 291**
fifa	OE	357.5 ± 242
	TE	379.9 ± 135
	Ours	**353.0 ± 231**
HR-employee-attrition	OE	826.8 ± 33.0
	TE	782.3 ± 32.1
	Ours	**779.2 ± 30.7**
StudentsPerformance	OE	3.025 ± 0.163
	TE	3.000 ± 0.161
	Ours	**2.989 ± 0.148**
winemag-130k	OE	2.087 ± 0.484
	TE	**1.477 ± 0.135**
	Ours	1.793 ± 0.150

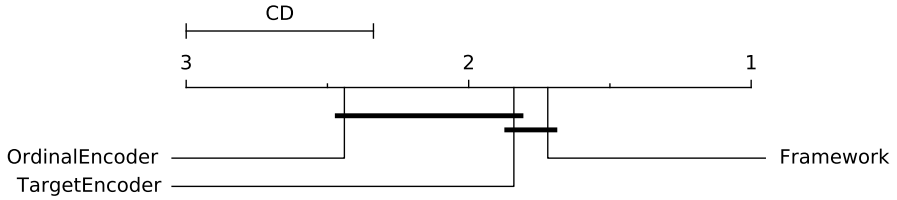
(b) Regression tasks

Table 5.2: The performance of the framework (Ours) against baseline preprocessing using the ordinal encoder (OE) and the target encoder (TE) on different learning tasks.

significance level of $\alpha = 0.05$. Based on the Shapiro-Wilk test to assess normality of the samples, we reject the null hypothesis that the population is normal for all three methods for both tasks (classification: $p_{OE} = 0.006$, $p_{TE} = 0.001$, $p_F = 0.000$; regression: $p_{OE} = 0.000$, $p_{TE} = 0.000$, $p_F = 0.000$). Based on these observations, we use the non-parametric Friedman test to determine significant differences between the median values of the population and the post-hoc Nemenyi test to infer which differences are significant for both learning tasks. The results of these tests for classification and regression are shown in Figures 5.2b and 5.3b, respectively.

Method	MR	MED	MAD	CI	γ	Magnitude
Framework	1.720	0.861	0.194	[0.729, 1.000]	0.000	negligible
TargetEncoder	1.840	0.791	0.179	[0.667, 1.000]	0.374	small
OrdinalEncoder	2.440	0.785	0.222	[0.635, 1.000]	0.360	small

(a) Summary of the Friedman test on the three methods; MR = Mean Rank, MED = Median, MAD = Mean Absolute Deviation, CI = Confidence Interval, γ = Gamma effect size, Magnitude = Effect size.

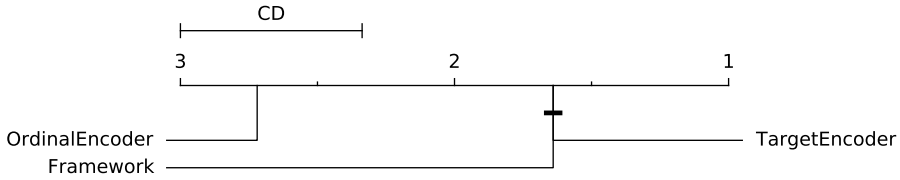


(b) Critical Difference (CD) diagram to visualize the results of the Nemenyi post-hoc test. The horizontal lines indicate that differences are not significant.

Figure 5.2: Friedman test with Nemenyi post-hoc test on the three methods concerning classification tasks.

Method	MR	MED	MAD	CI	γ	Magnitude
TargetEncoder	1.640	-307.947	-454.503	[-1829.350, -1.390]	0.000	negligible
Framework	1.640	-212.261	-312.169	[-1856.656, -1.713]	-0.245	small
OrdinalEncoder	2.720	-209.511	-307.915	[-2204.346, -1.833]	-0.254	small

(a) Summary of the Friedman test on the three methods; MR = Mean Rank, MED = Median, MAD = Mean Absolute Deviation, CI = Confidence Interval, γ = Gamma effect size, Magnitude = Effect size.



(b) Critical Difference (CD) diagram to visualize the results of the Nemenyi post-hoc test. The horizontal lines indicate that differences are not significant.

Figure 5.3: Friedman test with Nemenyi post-hoc test on the three methods concerning regression tasks. Note that the negative MAE is taken for this test, as it better reflects the performance in this test.

Based on these statistical tests, we reject the null hypotheses of the Friedman tests for both classification ($p = 0.015$) and regression ($p = 0.000$) that there is no difference in the central tendency of the three methods. The results indicate that the framework has the highest mean rank (MR) for classification tasks and a close second-place MR for regression tasks. Therefore, we assume a statistically significant difference between the median values of the methods for both tasks. Furthermore, based on the post-hoc Nemenyi test, we observe no significant difference between the ordinal encoder and the target encoder and between the target encoder and the

framework for classification tasks. As for regression tasks, we observe no significant difference between the target encoder and the framework. All other differences appear to be statistically significant. Therefore, the analysis indicates that the difference between the ordinal encoder and the framework is statistically significant, whereas the difference between the target encoder and the framework is not.

To summarize, based on all the aforementioned results, it can be said that the framework provides a feasible automated solution to identifying and cleaning string data, but further study can still be conducted to increase the overall performance.

5.2 String feature inference

5.2.1 Experiment set-up

To evaluate the performance of this module, we infer the string feature of 33 string columns originating from 19 datasets in total that were downloaded from Kaggle. Each dataset contains at least one of the string features discussed earlier, and each column has been manually labeled for evaluation (ground truth). To ensure that all string feature PFSMs are evaluated, the set of used columns contains all the discussed string features at least once.

The performance of each PFSM is evaluated based on the inference result with respect to the ground truth. Furthermore, to investigate whether each string feature can accurately be presented as a PFSM, we evaluate the reported outliers after inference. The outliers that are reported could contain a certain number of false negatives. If the proportion of false negatives reported is relatively significant, it is worth investigating how this proportion could be reduced.

5.2.2 Results

The results of the evaluation are summarized in Table 5.3. The table contains a column indicating which string feature PFSM is evaluated. Furthermore, the number of columns containing the corresponding string feature, the number of correctly inferred columns, and the accuracy are also shown in the table. Finally, the last two columns in the table correspond to the number of false negative outliers detected by the PFSM and the ratio of false negative outliers to the total number of entries in the data.

String feature	Number of columns	Number of correctly inferred columns	Accuracy	Number of false negative outliers	Ratio of false negative outliers
Coordinate	2	2	1.0	0	-
Day	1	1	1.0	0	-
E-mail	4	4	1.0	0	-
Filepath	5	4	0.80	0	-
Month	3	3	1.0	0	-
Numerical	6	6	1.0	0	-
Sentence	4	3	0.75	42158	0.32
URL	4	4	1.0	0	-
Zip code	4	4	1.0	0	-

Table 5.3: Results of string feature inference using PFSMs.

As can be seen from Table 5.3, the performance of the PFSMs is generally ideal for the provided datasets. Most string feature PFSMs report a perfect accuracy except for the filepath and sentence PFSMs. The overall performance is most likely related to the fact that the string features can easily be represented as regular expressions, as rule-based approaches achieve a relatively high score when the data format adheres to the rules.

However, rule-based approaches also come with the disadvantage that all features must adhere to a specific format. In other words, the PFSMs will report incorrect results when the format of the data in a column is slightly off. Take the filepath PFSM as an example. One of the columns that were used for evaluation contained filepaths in the format `filename.extension`, whereas the regular expression which the PFSM is based on also requires at least one slash or backslash at the beginning of such a string (i.e., `/filename.extension`). A similar case can be seen for the sentence PFSM, where a column consisting of short sentences is inferred as regular strings. One could argue that these mistakes can be prevented by including the format in the regular expression. However, one must also acknowledge that expressions of other PFSMs may overlap when the expression is more generalized (e.g., expressions for URLs or regular strings).

Out of the nine PFSMs, only the sentence state machine reported false-negative outliers. Out of the 130 217 entries, 42 158 entries were incorrectly classified as outliers. This result implies that slightly over 32% of the entries would more likely be classified as another data type or string feature rather than the ground truth. This detail can be problematic when the sentences in a column are exclusively made up of false negative entries, as these columns would be misclassified.

Upon closer inspection of the data, it was discovered that entries containing certain symbols or relatively shorter sentences are not considered as sentences by the PFSM system. The misclassifications are related to the regular expression that was used to create the sentence PFSM. This regular expression does not take all symbols into account and only considers an entry as a sentence consisting of at least six words. It is possible to generalize the regular expression to lower the number of false negatives. However, as mentioned earlier, this also increases the probability that the PFSM will overlap with other PFSMs, potentially making the results ambiguous and incorrect.

Overall, it can be argued that eight out of nine string features that were evaluated can be adequately represented by PFSMs using the current regular expressions. The only string feature that might benefit more from other techniques is sentences, as creating a regular expression to detect a variety of sentences in different formats is nontrivial compared to the other string features.

5.3 Processing inferred string features

5.3.1 Experiment set-up

To evaluate the effectiveness of our processing steps, we process six datasets containing either a classification or regression task and at least one of the string features that our probabilistic model can infer. A relevant machine learning model (gradient boosting classifier or regressor with default hyperparameter settings) is fitted on the data using stratified five-fold cross-validation, and either the average accuracy or MAE is reported. As for the baseline, the dataset is cleaned the same way as the dataset containing the processed string feature, except that the string feature in question is regarded as a standard string and is not uniquely processed. More specifically, after string feature inference, the string feature to be evaluated is labeled as a standard string instead and will be checked for missing values, outliers, and ordinality before being encoded accordingly. For each dataset used, additional information on the task, the model, and the target column is provided in Appendix B.1. Furthermore, the processing and training times of both instances are measured by taking the average over five runs. After the results are obtained, all values are transformed around the mean of the baseline for each dataset using Equation 5.1.

This experiment aims to determine the usefulness of processing the inferred string features using our approach, which type of processing techniques are worth considering the most, and which string features are the most valuable to consider processing.

5.3.2 Results

To evaluate the effects of string feature processing, we compare the processing time, training time, and model performance against our framework when no processing is applied for a specific string

feature. The results are depicted in Figure 5.4 and Table 5.4 in which various observations can be made.

Firstly, we observe that the model performance is either close to the baseline or slightly improved compared to the baseline. This detail can indicate that processing certain string features can positively impact the performance of the model in the best case and a performance decrease of at most one percent in the worst case.

Secondly, we observe that some string features take a significant amount of time to process, which is the case for the datasets that process sentence and zip code string features. These significant preprocessing time differences may be related to cardinality in the data and the dependency of APIs to fetch additional information. However, it is noticeable that the processed datasets display improved model performance. Thus, this observation indicates that processing certain string features may only be worth considering when performance and information fetching are prioritized. In our case, processing for performance over speed would apply to sentence and zip code string features.

Thirdly, we observe that preprocessing and task times for half of the evaluated datasets are reduced by at least ten percent, some at the cost of at most one percent of the model performance. These occurrences are evident when processing month, numerical, and URL string features. These string features are processed to reduce string complexity and correctly transform them into their numerical representation. These results indicate that reducing string complexity and transforming quasi-numerical values can be a preferred strategy to reduce running times while maintaining most of the model performance.

Finally, it appears that processing day string features using the current technique is not beneficial in terms of performance and slightly in terms of training time. This observation could indicate that different techniques may need to be considered or to drop any processing techniques from this string feature.

Based on all results, it can be argued that the most valuable processing techniques to apply for improved running times are string complexity reduction and quasi-numerical value transformation. Furthermore, processing string features such as sentences and zip codes using certain information extraction techniques is only valuable when performance is a priority and time is not an issue.

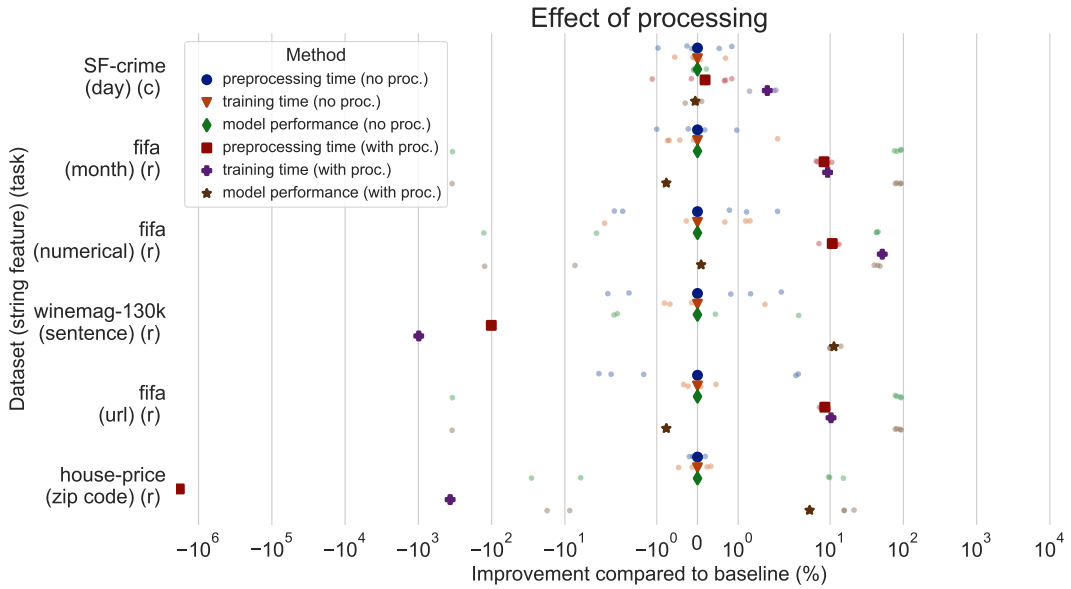


Figure 5.4: Difference in performance metrics (accuracy/MAE, preprocessing time, and training time) with and without processing of known string feature types on regression (r) or classification (c) tasks.

Dataset (string feature)	Method	Preprocessing time	Training time	Accuracy (classification)	MAE (regression)
SF-crime	Without processing	2789 ± 22.5	1.17e4 ± 49.3	**0.992 ± 1.11e-3**	-
(day)	With processing	**2778 ± 19.1**	**1.15e4 ± 47.3**	0.991 ± 1.12e-3	-
fifa	Without processing	345.2 ± 1.72	52.50 ± 0.457	-	1238 ± 2.12e3
(month)	With processing	**319.7 ± 4.04**	**47.68 ± 0.191**	-	**1237 ± 2.12e3**
fifa	Without processing	382.9 ± 6.36	150.3 ± 2.31	-	353.3 ± 235
(numerical)	With processing	**341.9 ± 8.16**	**72.89 ± 1.13**	-	**353.0 ± 231**
winemag-130k	Without processing	**4465 ± 67.1**	**31.48 ± 0.382**	-	1.823 ± 3.91e-2
(sentence)	With processing	9012 ± 131	342.7 ± 15.8	-	**1.617 ± 2.71e-2**
fifa	Without processing	349.7 ± 7.29	51.87 ± 0.225	-	**1233 ± 2.12e3**
(url)	With processing	**319.7 ± 4.04**	**46.76 ± 0.295**	-	1236 ± 2.12e3
house-price	Without processing	**0.5246 ± 4.92e-3**	**4.706 ± 2.50e-2**	-	1.246e5 ± 1.98e4
(zip code)	With processing	9573 ± 204	22.06 ± 0.259	-	**1.180e5 ± 1.91e4**

Table 5.4: Differences in performance metrics (accuracy/MAE, preprocessing time, and training time) with and without processing of known string feature types on different learning tasks.

5.4 Ordinality detection

5.4.1 Experiment set-up

In order to train and evaluate the performance of this module, 149 string columns are used from a total of 29 datasets downloaded from the UCI Machine Learning Repository and Kaggle. Out of the 149 columns, there are 81 ordinal and 68 nominal string columns. For each column, the eight features mentioned in Section 4.4.2 are extracted, and a label is appended indicating whether the column is nominal or ordinal. This procedure gives us 149 data points in total, each containing eight features for training and evaluating the model. The proposed model is the gradient boosting classifier provided by `sklearn` with hyperparameter `max_depth=2` and default for all other hyperparameters. The classifier's performance is evaluated using Leave-One-Out Cross-Validation (LOOCV) on the 149 data points and compared to the ground truth. Additionally, the F1 score, precision, recall, and Area Under the Curve (AUC) score are provided. Based on these criteria, we can discuss whether the extracted column features provide sufficient information for the model and how suitable the model is in a practical setting.

5.4.2 Results

The results of the evaluation are summarized in Table 5.5. Firstly, the reported accuracy sits at 0.98, with a standard deviation of 0.14. The relatively high standard deviation comes from evaluating using LOOCV, as this method results in an unbiased estimate of the performance. Despite the relatively high standard deviation, the performance is still acceptable when considering the worst case of the standard deviation. Secondly, the confusion matrix shows that one nominal and two ordinal columns have been misclassified, which is insignificant compared to the total amount of correctly classified ordinal and nominal columns. This insignificance is also noticeable in the precision, recall, and F1 score. Finally, the high AUC score indicates that the quality of prediction of the model is excellent.

Based on these results, it can be said that the gradient boosting classifier performs well when it has to classify order in the data given the extracted features. It also indicates that the extracted features using the proposed heuristics were indicative enough of whether the data is nominal or ordinal.

Metric	Score
Accuracy	0.980 ± 0.140
F1 score	0.978
Precision	0.971
Recall	0.985
AUC score	0.980

(a) Score for each metric

		Predicted class	
		Ordinal	Nominal
Actual class	Ordinal	79	2
	Nominal	1	67

(b) Confusion matrix

Table 5.5: Results of ordinality prediction using the gradient boosting classifier.

5.5 Ordinal encoding

5.5.1 Experiment set-up

To evaluate the performance of our approach, we determine the order of the string entries of 81 columns from a total of 11 datasets downloaded from the UCI Machine Learning Repository and Kaggle. Each column consists of at least two entries that are naturally ordered. The performance of our approach is measured based on the ground truth of the order by calculating the Spearman's Rank Correlation Coefficient of both orders. As for the baseline, we compute the ordering using the default settings of the ordinal encoder provided by `scikit-learn`. The comparison between the baseline and our approach tells us whether our approach is suitable for finding order in the data and in which cases the baseline method may perform better than our approach. After the correlation is computed for all columns, the mean score and the standard deviation are reported.

Furthermore, to evaluate whether the order affects the performance of a machine learning model, four out of the 11 datasets are used to evaluate the overall performance on the corresponding prediction task and the running times in seconds of preprocessing when our approach is applied in practice. The results are obtained by taking the mean of five preprocessing and training runs using stratified five-fold cross-validation. The tasks to be solved are either classification or regression tasks and are evaluated using the MAE and the accuracy metrics, respectively. The results of our approach will once again be compared to the baseline and the ground truth (oracle) undergoing the same data preparation procedure. For each dataset, the task, model, and target column are described in Appendix B.2.2.

5.5.2 Results

The summarized results of the evaluation for the ordering are shown in Table 5.6. More details on these results can be found in Tables C.1 and C.2 in Appendix C. Furthermore, the performance results are summarized in Table 5.8 and visualized in Figure 5.5.

Ordering method	Rank Correlation
baseline	0.1562 ± 0.5313
FlairNLP	**0.7189 ± 0.5356**

Table 5.6: Spearman's Rank Correlation Coefficient for ordinal encoders vs. ground truth ordering.

Based on Table 5.6, it appears that our approach significantly outperforms the baseline when it comes to determining the natural order. Furthermore, the standard deviation of both methods is relatively high to the point that the performance can fluctuate significantly. The high standard deviation may be related to the scale of Spearman's Rank Correlation (which ranges from -1 up to 1) and how sensitive the correlation calculation is for columns with few unique values to order.

Upon closer inspection of the results in Appendix C, we observe that our approach performs well for ordinal entries that consist of multiple words and performs poorly when numbers play a crucial role in ordering entries. This result makes sense as `FlairNLP` is used originally to analyze the sentiment and the intensity of sentences. Furthermore, as our approach orders entries from negative to positive, the order of all columns that scale from the least bad to the worst are inverted. This property could also have contributed to the high standard deviation reported. This result is not necessarily a problem as the order between values is still correct. Additionally, the baseline method scores high on alphabetically ordered entries and numerically ordered entries. This result makes sense since the encoder orders entries based on lexicographical order when no order is provided. These observations suggest that a sophisticated hybrid between the baseline and our approach might perform even better than each method separately.

As for evaluating the performance difference between passing different encoding orders, based on Table 5.8 and Figure 5.5, it appears that the difference in model performance between each method is insignificant in most cases. Surprisingly, it appears that the perfectly ordered encoding (oracle) appears to perform poorly compared to the baseline and `FlairNLP`. The reason for this occurrence is currently unknown and may be related to unlucky splits during cross-validation.

However, to ensure that the current results are reliable, a Friedman test is conducted on the transformed values to confirm whether the differences between the three methods are statistically significant. The transformed values are considered instead of separating the analysis on the learning tasks because of the small number of samples available. The result of the analysis is visible in Table 5.7. Based on the p-value, we reject the null hypothesis of the Friedman test that there is no difference in the central tendency of the three methods. Therefore, we assume that there is no statistically significant difference between the median values of the methods.

Method	MR	MED	MAD	CI	p-value
FlairNLP	1.600	0.097	0.781	[-0.835, 89.689]	
baseline	2.150	0.005	0.181	[-1.755, 98.604]	0.086
oracle	2.250	-1.539	2.312	[-23.812, 55.110]	

Table 5.7: Summary of the Friedman test on the three methods; MR = Mean Rank, MED = Median, MAD = Mean Absolute Deviation, CI = Confidence Interval

Furthermore, we observe that preprocessing using our approach always takes significantly longer than the baseline and the oracle. This detail is expected since `FlairNLP` needs to process and predict the order for each column it receives, which takes longer than default ordering or processing a pre-defined order. It does appear that training times are reduced in some cases when `FlairNLP` is used. However, this time reduction is smaller than the extra time spent determining the order using `FlairNLP`.

These results indicate that the order in which the data is encoded is not essential for regression and classification tasks. Additionally, it appears that using `FlairNLP` is sub-optimal when the user does not require the interpretability of the data due to its preprocessing time.

Dataset	Ordering method	Preprocessing time		Training time		Accuracy (classification)		MAE (regression)	
car	baseline	2.02e-2	± 2.83e-3	1.99e-3	± 2.03e-4	0.699	± 1.89e-2	-	
	FlairNLP	26.6	± 0.136	**1.99e-3**	**± 1.08e-5**	0.701	± 1.96e-2	-	
	oracle	**1.45e-2**	**± 2.91e-3**	2.10e-3	± 1.78e-4	**0.701**	**± 1.91e-2**	-	
nursery	baseline	3.87e-2	± 2.41e-3	**3.87e-3**	**± 6.12e-4**	0.662	± 7.36e-3	-	
	FlairNLP	35.6	± 0.671	4.52e-3	± 1.19e-3	**0.662**	**± 7.08e-3**	-	
	oracle	**1.70e-2**	**± 1.16e-3**	4.11e-3	± 8.11e-4	0.663	± 7.23e-3	-	
solar-flare	baseline	4.34e-2	± 4.74e-3	8.17e-2	± 7.03e-3	-		**0.179**	**± 0.184**
	FlairNLP	21.9	± 0.488	**7.93e-2**	**± 5.11e-3**	-		0.180	± 0.184
	oracle	**2.08e-2**	**± 2.86e-3**	7.97e-2	± 7.33e-3	-		0.181	± 0.186
soybean	baseline	**0.159**	**± 1.51e-2**	2.84e-3	± 1.28e-3	0.858	± 4.58e-2	-	
	FlairNLP	26.7	± 0.483	**2.58e-3**	**± 5.61e-4**	**0.860**	**± 4.74e-2**	-	
	oracle	0.193	± 2.78e-2	4.13e-3	± 8.14e-4	0.846	± 7.23e-3	-	

Table 5.8: Performance result of different ordering methods on machine learning tasks.

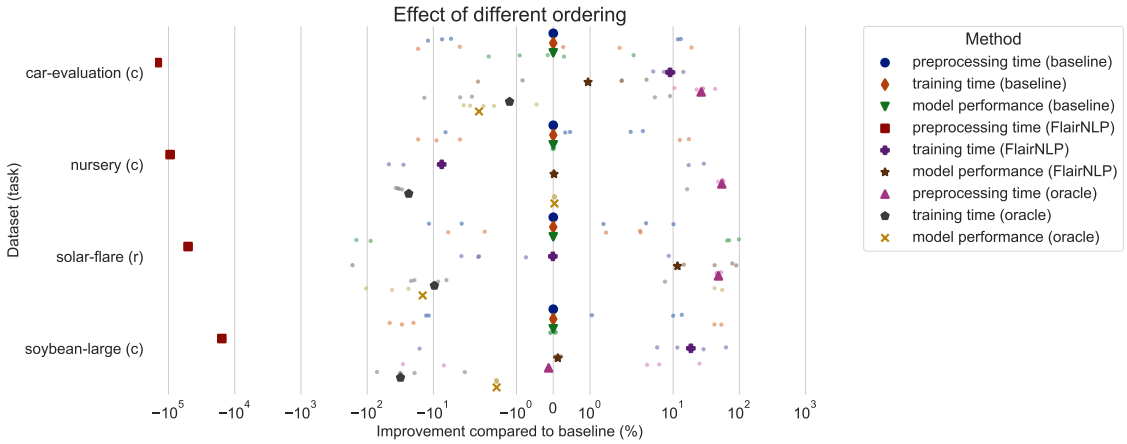


Figure 5.5: Difference in performance metrics (accuracy/MAE, preprocessing time, and training time) between different ordering strategies on regression (r) or classification (c) tasks.

Chapter 6

Conclusions

Data cleaning is a crucial step to ensure that high-quality data is used to analyze and fit machine learning models. However, due to the time-consuming nature of data cleaning, many users spend a significant amount of time to ensure that the data is up to standard. Tools have been developed to tackle this issue, but they still require users to interact and clean the data themselves. The automation of data cleaning is still a relatively open research field. While there are a handful of feasible solutions available to automatically preprocess numerical data, a system that can robustly preprocess different types of string features does not yet exist.

This thesis focused on automated handling and cleaning of strings in tabular data. We investigated and applied novel techniques in string handling and encoding and combined them into a Python framework that allows for automated string data cleaning. The framework can infer various string features using a probabilistic approach that uses Probabilistic Finite-State Machines constructed from regular expressions. Additionally, these inferred string features are further processed to reduce string complexity and extract additional features relevant to further analysis. If no specific string feature could be inferred using the probabilistic approach, our implemented pretrained classifier can accurately distinguish between ordered string values and unordered string values based on certain features in the data. Finally, users can decide whether the data is encoded or not. If the data requires to be encoded, the most fitting encoder is used based on previously made inferences and predictions. The encoders used in this work vary from simple techniques to novel techniques that are robust to morphological variants and high-cardinality in the data. In the end, users receive a clean dataset of high quality that can be used for analysis or machine learning tasks.

Based on our evaluation of the framework and its modules, several conclusions were drawn. First of all, the framework proved to perform well on the evaluated data, outperforming both baselines in most of the datasets. Improvements can be made by re-evaluating and optimizing the preprocessing steps for specific string feature types and by considering more encoding options to use in specific cases. Secondly, it appears that string feature inference using custom PFSMs works well when the string features adhere to the same format as the regular expression used to construct the PFSM. This property indicates that string feature types with a slightly different format than the PFSMs corresponding to these types might not be subject to inference. However, this problem did not seem to occur often in the results. Thirdly, it can be argued that the most valuable processing techniques to apply for improved running times are string complexity reduction and quasi-numerical value transformation. Furthermore, processing string features such as sentences and zip codes using certain information extraction techniques are valuable when performance is a priority and time is not an issue. Fourthly, using a gradient boosting classifier on various extracted characteristics worked excellent for classifying ordinal and nominal string columns. This result indicates that classifying string columns more generally is undoubtedly feasible and can automatically suggest specific encoding techniques. Finally, it turns out that the text sentiment intensity analyzer of `FlairNLP` was shown to outperform the ordinal encoder when

it comes to encoding ordinal data according to the actual order. However, it can be argued that this approach does not have a much-added benefit to the overall performance of a machine learning model on a prediction task. Therefore, it might only be useful in the context of interpretability for the users after the framework has finished its operations.

To summarize, the framework and its components seem to be suitable approaches to automate the process of handling and cleaning string data into high-quality data, and we hope that this framework and open-source implementation will speed up research in these areas.

6.1 Future work

Given the broadness of the topic, there is much room for improvement for the various implementations by building upon existing methods or designing more complex frameworks for specific steps. Listed below are some of the potential future work that can be picked up.

- **General improvements**: Currently, the framework only provides automated string handling for tabular data and associated prediction tasks. It is possible to expand the framework to handle strings in non-tabular data, such as pieces of text for NLP tasks. Additionally, the framework has only been evaluated on a gradient tree boosting learner. The evaluation could also be extended to compare the performance differences of other learners (e.g., linear models or support vector machines) to investigate the influence of the learner on the results. Finally, (interactive) visualizations could be introduced to improve the interpretability of the cleaned data for users.

- **String feature inference**: The current implementation of string feature inference is relatively strict. As mentioned before, all string features that we are currently inferring need to match the whole pattern of the Probabilistic Finite-State Machine. Possible future work could perhaps investigate the possibility of relaxing this constraint via sub-pattern matching and better use of type probabilities in subsequent processing (e.g., if it is only 60% certain that a string feature represents a date, a more robust encoding is needed, or a human should be brought into the loop). One could also investigate whether it is valuable to split the data into subsets and let the PFSMs predict each subset separately to allow policies such as majority voting to make the final inference. Additionally, more useful string features could be investigated and handled via regular expressions.

- **Processing string features**: Inferred string features are currently processed to extract additional features, reduce complexity, and apply unique encoding strategies. Future work could investigate different goals or processing techniques that may benefit certain string features more than what is currently applied.

- **Ordinality prediction**: Classifying ordinal and nominal columns is currently done using feature engineering that is partly based on domain knowledge. It would be interesting to investigate whether some features that depend on domain knowledge could be omitted to allow for more objective features that dictate the ordinality of data. It is also interesting to investigate a combination of multiple strategies to give a final predicament on the ordinality of the data. For example, the Bayesian approach by Valera et al. (2017) could be worth revisiting and added to the system to see whether it might benefit the whole classification step.

- **Encoding**: The best nominal encoding technique to be applied is currently determined based on the cardinality of the data and whether target classes are balanced. A possible research direction could be to create a system that recommends or applies an encoding technique that is the most applicable based on various other features of the data (e.g., the similarity between entries and other features). The encoding techniques in this system could be basic, more sophisticated, or state-of-the-art. Additionally, research could be done on optimizing the heuristics for selecting the currently implemented encoders. Furthermore, there is still

room for improvement when it comes to predicting the order in ordinal data. Although `FlairNLP` was designed for a different purpose, it has been demonstrated to perform well in some aspects and poorly in others. This result indicates that there is still room to improve on aspects where the predictor performed poorly by, for example, also taking numbers into account when weighing the sentiments.

- **Outlier detection**: The current implementation for outlier detection and handling for string entries is relatively simple. As this was not part of the scope of the thesis, further research on automated outlier and (automated) errata handling for string data could be worth investigating for optimizing the current decision criteria.

Bibliography

[1] Cloudingo. https://cloudingo.com/. 22

[2] Data Ladder. https://dataladder.com/. 21

[3] DataCleaner. https://datacleaner.github.io/. 21

[4] Featuretools. https://www.featuretools.com/. 12, 27

[5] geopy 2.1.0. https://github.com/geopy/geopy. 13, 31

[6] greenery. https://pypi.org/project/greenery/. 26

[7] OpenRefine. https://openrefine.org/. 21

[8] Roman Yurchark. pgeocode 0.3.0. https://github.com/symerio/pgeocode. 12, 13

[9] Trifacta. https://www.trifacta.com/. 22

[10] Alan Agresti. *Categorical data analysis*, volume 482. John Wiley & Sons, 2003. 38

[11] Alan Akbik, Tanja Bergmann, Duncan Blythe, Kashif Rasul, Stefan Schweter, and Roland Vollgraf. FLAIR: An easy-to-use framework for state-of-the-art NLP. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, pages 54–59, 2019. 18

[12] Enrique Alfonseca and Suresh Manandhar. An unsupervised method for general named entity recognition and automated concept discovery. In *Proceedings of the 1st international conference on general WordNet, Mysore, India*, pages 34–43, 2002. 11

[13] Paul Allison. Imputation by predictive mean matching: Promise & peril. *Statistical Horizons*, 2015. 20

[14] Melissa J Azur, Elizabeth A Stuart, Constantine Frangakis, and Philip J Leaf. Multiple imputation by chained equations: what is it and how does it work? *International journal of methods in psychiatric research*, 20(1):40–49, 2011. 20

[15] Stephen D Bay and Mark Schwabacher. Mining distance-based outliers in near linear time with randomization and a simple pruning rule. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 29–38, 2003. 21

[16] Felix Biessmann, David Salinas, Sebastian Schelter, Philipp Schmidt, and Dustin Lange. Deep Learning for Missing Value Imputation in Tables with Non-Numerical Data. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pages 2017–2025, 2018. 20

[17] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.", 2009. 36

[18] Vinayak R. Borkar, Kaustubh Deshmukh, and Sunita Sarawagi. Automatically extracting structure from free text addresses. *IEEE Data Eng. Bull.*, 23(4):27–32, 2000. 3

[19] Léon Bottou and Olivier Bousquet. The tradeoffs of large scale learning. *Advances in neural information processing systems*, 20:161–168, 2007. 12

[20] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. LOF: identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 93–104, 2000. 21

[21] Andrei Z Broder, Moses Charikar, Alan M Frieze, and Michael Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60(3):630–659, 2000. 15

[22] S van Buuren and Karin Groothuis-Oudshoorn. mice: Multivariate imputation by chained equations in R. *Journal of statistical software*, pages 1–68, 2010. 20

[23] John Canny. GaP: a factor model for discrete data. In *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 122–129, 2004. 16

[24] Patricio Cerda and Gaël Varoquaux. Encoding high-cardinality string categorical variables. *IEEE Transactions on Knowledge and Data Engineering*, 2020. vi, 15, 16, 42, 47

[25] Patricio Cerda, Gaël Varoquaux, and Balázs Kégl. Similarity encoding for learning with dirty categorical variables. *Machine Learning*, 107(8-10):1477–1494, 2018. 14, 15, 21, 47

[26] Cerda, Patricio and Varoquaux, Gaël. dirty_cat: machine learning on dirty categories, 2020. 41, 42

[27] Taha Ceritli, Christopher KI Williams, and James Geddes. ptype: probabilistic type inference. *Data Mining and Knowledge Discovery*, 34(3):870–904, 2020. vi, 5, 6, 10, 26

[28] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):1–58, 2009. 20, 21

[29] Jacob Cohen, Patricia Cohen, Stephen G West, and Leona S Aiken. *Applied multiple regression/correlation analysis for the behavioral sciences*. Routledge, 2013. 12, 13

[30] CrowdFlower. Data Science Report, 2016. 1, 3

[31] Anna Veronika Dorogush, Vasily Ershov, and Andrey Gulin. CatBoost: gradient boosting with categorical features support. *arXiv preprint arXiv:1810.11363*, 2018. 9

[32] Oren Etzioni, Michael Cafarella, Doug Downey, Ana-Maria Popescu, Tal Shaked, Stephen Soderland, Daniel S Weld, and Alexander Yates. Unsupervised named-entity extraction from the web: An experimental study. *Artificial intelligence*, 165(1):91–134, 2005. 11

[33] Jerome Friedman, Trevor Hastie, Robert Tibshirani, et al. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001. 8, 9

[34] Jerome H Friedman. Stochastic gradient boosting. *Computational statistics & data analysis*, 38(4):367–378, 2002. 9

[35] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity Search in High Dimensions via Hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, page 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. 15

[36] Wael H Gomaa, Aly A Fahmy, et al. A survey of text similarity approaches. *International Journal of Computer Applications*, 68(13):13–18, 2013. 14

[37] Krzysztof Grabczewski and Norbert Jankowski. Feature selection with decision tree criterion. In *Fifth International Conference on Hybrid Intelligent Systems (HIS'05)*, pages 6–pp. IEEE, 2005. 8, 11

[38] Frank R Hampel. Robust estimation: A condensed partial survey. *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete*, 27(2):87–104, 1973. 20

[39] Larry Hardesty. Automating big-data analysis, Oct 2015. 12

[40] Douglas M Hawkins. *Identification of outliers*, volume 11. Springer, 1980. 20

[41] Bernhard Hofmann-Wellenhof, Herbert Lichtenegger, and James Collins. *Global positioning system: theory and practice*. Springer Science & Business Media, 2012. 30

[42] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001. 10

[43] Clayton Hutto and Eric Gilbert. Vader: A parsimonious rule-based model for sentiment analysis of social media text. In *Proceedings of the International AAAI Conference on Web and Social Media*, volume 8, 2014. 18

[44] Paul Jaccard. The distribution of the flora in the alpine zone. 1. *New phytologist*, 11(2):37–50, 1912. 21

[45] Daniel Jurafsky and H James. Martin: Speech and Language Processing: An Introduction to Natural Language Processing, Speech Recognition, and Computational Linguistics, 2008. 10

[46] Sean Kandel, Andreas Paepcke, Joseph M Hellerstein, and Jeffrey Heer. Enterprise data analysis and visualization: An interview study. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2917–2926, 2012. 3

[47] James Max Kanter and Kalyan Veeramachaneni. Deep Feature Synthesis: Towards automating data science endeavors. In *2015 IEEE international conference on data science and advanced analytics (DSAA)*, pages 1–10. IEEE, 2015. 12

[48] Epaminondas Kapetanios, Doina Tatar, and Christian Sacarea. *Natural language processing: semantic aspects*. CRC Press, 2013. 11

[49] Gilad Katz, Eui Chul Richard Shin, and Dawn Song. Explorekit: Automatic feature generation and selection. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 979–984. IEEE, 2016. 12

[50] Guolin Ke, Jia Zhang, Zhenhui Xu, Jiang Bian, and Tie-Yan Liu. TabNN: A universal neural network solution for tabular data. 2018. vi, 9, 10, 39

[51] Won Kim, Byoung-Ju Choi, Eui-Kyeong Hong, Soo-Kyung Kim, and Doheon Lee. A taxonomy of dirty data. *Data mining and knowledge discovery*, 7(1):81–99, 2003. 14

[52] Arno J Knobbe, Arno Siebes, and Daniël Van Der Wallen. Multi-relational decision tree induction. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 378–383. Springer, 1999. 11

[53] Sanjay Krishnan, Jiannan Wang, Michael J Franklin, Ken Goldberg, Tim Kraska, Tova Milo, and Eugene Wu. SampleClean: Fast and Reliable Analytics on Dirty Data. *IEEE Data Eng. Bull.*, 38(3):59–75, 2015. 3, 22, 23

[54] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J Franklin, and Ken Goldberg. Activeclean: Interactive data cleaning for statistical modeling. *Proceedings of the VLDB Endowment*, 9(12):948–959, 2016. 3, 22, 23

[55] Sanjay Krishnan and Eugene Wu. AlphaClean: Automatic Generation of Data Cleaning Pipelines. *CoRR*, abs/1904.11827, 2019. 3, 22, 23

[56] Hoang Thanh Lam, Johann-Michael Thiebaut, Mathieu Sinn, Bei Chen, Tiep Mai, and Oznur Alkan. One button machine for automating feature engineering in relational databases, 2017. 12

[57] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union, 1966. 21

[58] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932. 38

[59] Dekang Lin and Xiaoyun Wu. Phrase clustering for discriminative learning. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 1030–1038, 2009. 11

[60] Roderick JA Little. A test of missing completely at random for multivariate data with missing values. *Journal of the American statistical Association*, 83(404):1198–1202, 1988. 19

[61] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation-based anomaly detection. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 6(1):1–39, 2012. 21

[62] Steven Loria. TextBlob Documentation. *Release 0.15*, 2, 2018. 18

[63] Mohammad Mahdavi, Ziawasch Abedjan, Raul Castro Fernandez, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. Raha: A configuration-free error detection system. In *Proceedings of the 2019 International Conference on Management of Data*, pages 865–882, 2019. 3, 22, 23

[64] Michael S Matell and Jacob Jacoby. Is there an optimal number of alternatives for Likert scale items? Study I: Reliability and validity. *Educational and psychological measurement*, 31(3):657–674, 1971. 41

[65] John T McCoy, Steve Kroon, and Lidia Auret. Variational autoencoders for missing data imputation with application to a simulated milling circuit. *IFAC-PapersOnLine*, 51(21):141–146, 2018. 20

[66] Daniele Micci-Barreca. A Preprocessing Scheme for High-Cardinality Categorical Attributes in Classification and Prediction Problems, 2001. 13

[67] Daniele Micci-Barreca. A Preprocessing Scheme for High-Cardinality Categorical Attributes in Classification and Prediction Problems. *SIGKDD Explor. Newsl.*, 3(1):27–32, July 2001. 13

[68] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013. 12, 14

[69] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26:3111–3119, 2013. 14

[70] George A Miller. WordNet: a lexical database for English. *Communications of the ACM*, 38(11):39–41, 1995. 17

[71] Heiko Müller and Johann-Christph Freytag. *Problems, methods, and challenges in comprehensive data cleansing*. Professoren des Inst. Für Informatik, 2005. 1

[72] Mashaal Musleh, Mourad Ouzzani, Nan Tang, and AnHai Doan. CoClean: Collaborative Data Cleaning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2757–2760, 2020. 22, 23

[73] David Nadeau and Satoshi Sekine. A survey of named entity recognition and classification. *Lingvisticae Investigationes*, 30(1):3–26, 2007. 11

[74] Joel Nothman, Nicky Ringland, Will Radford, Tara Murphy, and James R Curran. Learning multilingual named entity recognition from Wikipedia. *Artificial Intelligence*, 194:151–175, 2013. 11

[75] Norman W. Paton. Automating Data Preparation: Can We? Should We? Must We? 2019. 3

[76] Azaria Paz. *Introduction to probabilistic automata*. Academic Press, 1971. 6

[77] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. 20, 43, 44

[78] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014. 14, 40

[79] Kedar Potdar, Taher S Pardawala, and Chinmay D Pai. A comparative study of categorical variable encoding techniques for neural network classifiers. *International journal of computer applications*, 175(4):7–9, 2017. 12

[80] Nataliya Prokoshyna, Jaroslaw Szlichta, Fei Chiang, Renée J Miller, and Divesh Srivastava. Combining quantitative and logical data cleaning. *Proceedings of the VLDB Endowment*, 9(4):300–311, 2015. 21

[81] Michael O Rabin. Probabilistic automata. *Information and control*, 6(3):230–245, 1963. 6

[82] Anand Rajaraman and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge University Press, 2011. 15

[83] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. HoloClean: Holistic Data Repairs with Probabilistic Inference. *CoRR*, abs/1702.00820, 2017. 3, 22, 23

[84] Timothy Rozario, Troy Long, Mingli Chen, Weiguo Lu, and Steve Jiang. Towards automated patient data cleaning using deep learning: A feasibility study on the standardization of organ labeling. *arXiv preprint arXiv:1801.00096*, 2017. 3

[85] Donald B Rubin. Inference and missing data. *Biometrika*, 63(3):581–592, 1976. 19

[86] Bernhard Schölkopf, Robert C Williamson, Alex Smola, John Shawe-Taylor, and John Platt. Support vector method for novelty detection. *Advances in neural information processing systems*, 12:582–588, 1999. 21

[87] Muzammil Shahbaz, Phil McMinn, and Mark Stevenson. Automated discovery of valid test strings from the web using dynamic regular expressions collation and natural language processing. In *2012 12th International Conference on Quality Software*, pages 79–88. IEEE, 2012. 10

[88] Daniel J Stekhoven and Peter Bühlmann. MissForest — non-parametric missing value imputation for mixed-type data. *Bioinformatics*, 28(1):112–118, 2012. 20

[89] The pandas development team. pandas-dev/pandas: Pandas, February 2020. 5

[90] 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. PMLR, 06–11 Aug 2017. vii, 7, 38

[91] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 1113–1120, 2009. 12, 13

[92] Jinsung Yoon, James Jordon, and Mihaela Van Der Schaar. GAIN: Missing data imputation using generative adversarial nets. *arXiv preprint arXiv:1806.02920*, 2018. 20

[93] Yang C Yuan. Multiple imputation for missing data: Concepts and new development (Version 9.0). *SAS Institute Inc, Rockville, MD*, 49(1-11):12, 2010. 20

[94] Shichao Zhang, Xindong Wu, and Manlong Zhu. Efficient missing data imputation for supervised learning. In *9th IEEE International Conference on Cognitive Informatics (ICCI'10)*, pages 672–679. IEEE, 2010. 19

[95] Marc-André Zöller and Marco F Huber. Benchmark and Survey of Automated Machine Learning Frameworks. *arXiv preprint arXiv:1904.12054*, 2019. 22

Appendix A

Demonstration

This appendix provides a brief installation guide and a few examples of the framework in action.

A.1 Installing the framework

Users can install the framework using pip or by cloning the GitHub repository:

```
# pip
pip install git+https://github.com/ml-tue/automated-string-cleaning.git

# GitHub clone
git clone https://github.com/ml-tue/automated-string-cleaning.git
```

When cloning from GitHub, it might be necessary to install relevant packages like so:

```
    pip install -r requirements.txt
```

A.2 Example 1: Clean and encode the Academic Performance dataset

In this example, we clean and encode a dataset containing string columns.

1. Download the Students' Academic Performance dataset from https://www.kaggle.com/aljarah/xAPI-Edu-Data.
2. Load the dataset as a pandas DataFrame in Jupyter Notebook.

```
import pandas as pd

X = pd.read_csv(r'<path-to-csv>/<filename>.csv')
display(X)
```

	gender	NationalITy	PlaceofBirth	StageID	GradeID	SectionID	Topic	Semester	Relation	raisedhands	VisITedResources	AnnouncementsView	Discussion	ParentAnsweringSurvey	ParentschoolSatisfaction	StudentAbsenceDays	Class
0	M	KW	KuwaIT	lowerlevel	G-04	A	IT	F	Father	15	16	2	20	Yes	Good	Under-7	M
1	M	KW	KuwaIT	lowerlevel	G-04	A	IT	F	Father	20	20	3	25	Yes	Good	Under-7	M
2	M	KW	KuwaIT	lowerlevel	G-04	A	IT	F	Father	10	7	0	30	No	Bad	Above-7	L
3	M	KW	KuwaIT	lowerlevel	G-04	A	IT	F	Father	30	25	5	35	No	Bad	Above-7	L
4	M	KW	KuwaIT	lowerlevel	G-04	A	IT	F	Father	40	50	12	50	No	Bad	Above-7	M
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
475	F	Jordan	Jordan	MiddleSchool	G-08	A	Chemistry	S	Father	5	4	5	8	No	Bad	Above-7	L
476	F	Jordan	Jordan	MiddleSchool	G-08	A	Geology	F	Father	50	77	14	28	No	Bad	Under-7	M
477	F	Jordan	Jordan	MiddleSchool	G-08	A	Geology	S	Father	55	74	25	29	No	Bad	Under-7	M
478	F	Jordan	Jordan	MiddleSchool	G-08	A	History	F	Father	30	17	14	57	No	Bad	Above-7	L
479	F	Jordan	Jordan	MiddleSchool	G-08	A	History	S	Father	35	14	23	62	No	Bad	Above-7	L

3. Run the framework on the data in one of the following ways:

   - Default settings: the encoded data is fitted in the original column and no target encoder is applied since no target column is passed through.

```
from auto_string_cleaner import main

X = main.run(X)
```

   - With dense encoding disabled: each dimension in the encoded data is put in a separate column.

```
from auto_string_cleaner import main

X = main.run(X, dense_encoding=False)
```

   - With target encoder enabled: separate the target column from the dataset and pass it through the framework.

```
from auto_string_cleaner import main

y = X.iloc[:, 'Class']
X = X.drop(columns=['Class'])
X, y = main.run(X, y)
```

4. During cleaning, the framework prints at which step it currently is and some of the changes that it makes to the data.

```
> Performing pre-checks...
> Inferring data types and string features...
> Checking and handling any missing values...
> Checking and handling any string and data type outliers...
> Processing string features in the data...
> Predicting ordinality of string columns without string features...
> Encoding string data...
```

5. At the end, a brief overview is given in some of the decisions that were made during the process. Users can display the data to see the result of using the framework (image example uses default settings).

	Number of unique values	Type	Missing values	Outliers	Ordinal?	Encoding
gender	1	boolean	[]	[M]	NaN	NaN
NationalITy	14	string	[]	[]	No	SimilarityEncoder
PlaceofBirth	14	string	[]	[]	No	SimilarityEncoder
StageID	3	string	[]	[]	No	SimilarityEncoder
GradeID	10	string	[]	[G-02, ..., G-12]	No	SimilarityEncoder
SectionID	3	string	[]	[]	No	SimilarityEncoder
Topic	12	string	[]	[]	No	SimilarityEncoder
Semester	1	boolean	[]	[S]	NaN	NaN
Relation	2	string	[]	[]	No	SimilarityEncoder
raisedhands	82	integer	[]	[]	NaN	NaN
VisITedResources	89	integer	[]	[]	NaN	NaN
AnnouncementsView	88	integer	[]	[]	NaN	NaN
Discussion	90	integer	[]	[]	NaN	NaN
ParentAnsweringSurvey	2	boolean	[]	[]	NaN	NaN
ParentschoolSatisfaction	2	string	[]	[]	No	SimilarityEncoder
StudentAbsenceDays	2	numerical	[]	[]	NaN	NaN
Class	3	string	[]	[]	No	SimilarityEncoder

	raisedhands	VisITedResources	AnnouncementsView	Discussion	NationalITy	PlaceofBirth	StageID	GradeID	SectionID	Topic	Relation	ParentschoolSatisfaction	Class	gender	Semester	ParentAnsweringSurvey	StudentAbsenceDays
0	15	16	2	20	[0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. ...	[0.03125, 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. ...	[0.0169491525423728...	[0.33333333333333... 0.33333333333333...	[1. 0. 0. 0.]	[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. ...	[1. 0. 0.]	[0.05, 1.0]	[0. 0. 1.]	0	0	1	0
1	20	20	3	25	[0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. ...	[0.03125, 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. ...	[0.0169491525423728...	[0.33333333333333... 0.33333333333333...	[1. 0. 0. 0.]	[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. ...	[1. 0. 0.]	[0.05, 1.0]	[0. 0. 1.]	0	0	1	0
2	10	7	0	30	[0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. ...	[0.03125, 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. ...	[0.0169491525423728...	[0.33333333333333... 0.33333333333333...	[1. 0. 0. 0.]	[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. ...	[1. 0. 0.]	[1.0, 0.05]	[0. 1. 0.]	0	0	0	1
3	30	25	5	35	[0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. ...	[0.03125, 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. ...	[0.0169491525423728...	[0.33333333333333... 0.33333333333333...	[1. 0. 0. 0.]	[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. ...	[1. 0. 0.]	[1.0, 0.05]	[0. 1. 0.]	0	0	0	1
4	40	50	12	50	[0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. ...	[0.03125, 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. ...	[0.0169491525423728...	[0.33333333333333... 0.33333333333333...	[1. 0. 0. 0.]	[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. ...	[1. 0. 0.]	[1.0, 0.05]	[0. 0. 1.]	0	0	0	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
475	5	4	5	8	[0. 0. 0.1111111111111111, 0. 0. 1. 0. 0. 0. 0. ...	[0. 0. 0.1111111111111111, 0. 0. 1. 0. 0. 0. 0. ...	[0.29411764705882354, 1.0, 0.03125]	[0.33333333333333... 0.33333333333333... 0.333...	[1. 0. 0. 0.]	[0. 0. 0.0212765957446808, 1. 0. 0.021276595744...	[1. 0. 0.]	[1.0, 0.05]	[0. 1. 0.]	0	0	0	1
476	50	77	14	28	[0. 0. 0.1111111111111111, 0. 0. 1. 0. 0. 0. 0. ...	[0. 0. 0.1111111111111111, 0. 0. 1. 0. 0. 0. 0. ...	[0.29411764705882354, 1.0, 0.03125]	[0.33333333333333... 0.33333333333333... 0.333...	[1. 0. 0. 0.]	[0. 0. 0.4. 0.0212765957446808, 0. 0. 0. 0. 1. 0....	[1. 0. 0.]	[1.0, 0.05]	[0. 0. 1.]	0	0	0	0
477	55	74	25	29	[0. 0. 0.1111111111111111, 0. 0. 1. 0. 0. 0. 0. ...	[0. 0. 0.1111111111111111, 0. 0. 1. 0. 0. 0. 0. ...	[0.29411764705882354, 1.0, 0.03125]	[0.33333333333333... 0.33333333333333... 0.333...	[1. 0. 0. 0.]	[0. 0. 0.4. 0.0212765957446808, 0. 0. 0. 0. 1. 0....	[1. 0. 0.]	[1.0, 0.05]	[0. 0. 1.]	0	0	0	0
478	30	17	14	57	[0. 0. 0.1111111111111111, 0. 0. 1. 0. 0. 0. 0. ...	[0. 0. 0.1111111111111111, 0. 0. 1. 0. 0. 0. 0. ...	[0.29411764705882354, 1.0, 0.03125]	[0.33333333333333... 0.33333333333333... 0.333...	[1. 0. 0. 0.]	[0. 0. 0.0243902439024390... 0.1428571428571428...	[1. 0. 0.]	[1.0, 0.05]	[0. 1. 0.]	0	0	0	1
479	35	14	23	62	[0. 0. 0.1111111111111111, 0. 0. 1. 0. 0. 0. 0. ...	[0. 0. 0.1111111111111111, 0. 0. 1. 0. 0. 0. 0. ...	[0.29411764705882354, 1.0, 0.03125]	[0.33333333333333... 0.33333333333333... 0.333...	[1. 0. 0. 0.]	[0. 0. 0.0243902439024390... 0.1428571428571428...	[1. 0. 0.]	[1.0, 0.05]	[0. 1. 0.]	0	0	0	1

A.3 Example 2: Clean the Wine Reviews dataset

In this example, we clean a dataset containing high-cardinality string columns and some string features.

1. Download the Wine Reviews dataset from `https://www.kaggle.com/zynicide/wine-reviews`.

2. Load the dataset as a pandas DataFrame in Jupyter Notebook.

```
import pandas as pd

X = pd.read_csv(r'<path-to-csv>/<filename>.csv')
display(X)
```

	country	description	designation	points	price	province	region_1	region_2	taster_name	taster_twitter_handle	title	variety	winery
0	Italy	Aromas include tropical fruit, broom, brimston...	Vulkà Bianco	87	NaN	Sicily & Sardinia	Etna	NaN	Kerin O'Keefe	@kerinokeefe	Nicosia 2013 Vulkà Bianco (Etna)	White Blend	Nicosia
1	Portugal	This is ripe and fruity, a wine that is smooth...	Avidagos	87	15.0	Douro	NaN	NaN	Roger Voss	@vossroger	Quinta dos Avidagos 2011 Avidagos Red (Douro)	Portuguese Red	Quinta dos Avidagos
2	US	Tart and snappy, the flavors of lime flesh and...	NaN	87	14.0	Oregon	Willamette Valley	Willamette Valley	Paul Gregutt	@paulgwine	Rainstorm 2013 Pinot Gris (Willamette Valley)	Pinot Gris	Rainstorm
3	US	Pineapple rind, lemon pith and orange blossom ...	Reserve Late Harvest	87	13.0	Michigan	Lake Michigan Shore	NaN	Alexander Peartree	NaN	St. Julian 2013 Reserve Late Harvest Riesling ...	Riesling	St. Julian
4	US	Much like the regular bottling from 2012, this...	Vintner's Reserve Wild Child Block	87	65.0	Oregon	Willamette Valley	Willamette Valley	Paul Gregutt	@paulgwine	Sweet Cheeks 2012 Vintner's Reserve Wild Child...	Pinot Noir	Sweet Cheeks
...	...	...	...	...	...	...	...	...	...	...	...	...	...

3. Run the framework on the data.

```
from auto_string_cleaner import main

X = main.run(X, encode=False)
```

4. During cleaning, the framework prints at which step it currently is and some of the changes that it makes to the data.

```
> Performing pre-checks...
> Inferring data types and string features...
> Checking and handling any missing values...
>> Missing values imputed using IterativeImputer
> Checking and handling any string and data type outliers...
>> Outlier found in column "designation". Outlier "'Unfiltered'" replaced by "Unfiltered".
>> Outlier found in column "designation". Outlier "Brut Rose" replaced by "Brut Rosé".
...
>> Outlier found in column "designation". Outlier "Criança" replaced by "Crianza".
>> Outlier found in column "designation". Outlier "Cuveé" replaced by "Cuvée".
> Processing string features in the data...
> Predicting ordinality of string columns without string features...
```

5. At the end, a brief overview is given in some of the decisions that were made during the process. Users can display the data to see the result of using the framework.

	Number of unique values	Type	Missing values	Outliers	Ordinal?	Encoding
country	43	string	[nan]	[]	No	GapEncoder
description	119955	sentence	[]	[<outlier length too large>]	NaN	MinHashEncoder
designation	37963	string	[nan]	[%@#$!, ..., \P"]	No	MinHashEncoder
points	21	integer	[]	[]	NaN	NaN
price	405	float	[nan]	[]	NaN	NaN
province	425	string	[nan]	[Bío Bío Valley, ..., Župa]	No	MinHashEncoder
region_1	1227	string	[nan]	[<outlier length too large>]	No	MinHashEncoder
region_2	17	string	[nan]	[Napa-Sonoma]	No	SimilarityEncoder
taster_name	19	string	[nan]	[Kerin O'Keefe]	No	SimilarityEncoder
taster_twitter_handle	15	string	[nan]	[]	No	SimilarityEncoder
title	118840	sentence	[]	[<outlier length too large>]	NaN	MinHashEncoder
variety	705	string	[nan]	[Albariño, ..., Žilavka]	No	MinHashEncoder
winery	16755	string	[]	[1+1=3, ..., Štoka]	No	MinHashEncoder

	points	price	country	designation	province	region_1	region_2	taster_name	taster_twitter_handle	variety	winery	description	title
0	87.0	27.0	Italy	Vulkà Bianco	Sicily & Sardinia	Etna	Napa	Kerin O'Keefe	@kerinokeefe	White Blend	Nicosia	fruit broom brimstone herb palate apple citrus...	Nicosia
1	87.0	15.0	Portugal	Avidagos	Douro	Mâcon-Milly Lamartine	Napa	Roger Voss	@vossroger	Portuguese Red	Quinta dos Avidagos	fruity wine Firm juicy berry acidity	dos
2	87.0	14.0	US	Jester Sangiovese	Oregon	Willamette Valley	Willamette Valley	Paul Gregutt	@paulgwine	Pinot Gris	Rainstorm	flesh dominate pineapple acidity wine	Rainstorm
3	87.0	13.0	US	Reserve Late Harvest	Michigan	Lake Michigan Shore	Napa	Alexander Peartree	@AnneInVino	Riesling	St. Julian	rind pith orange aromas palate guava mango way...	St. Julian 2013 Reserve Late Harvest Riesling ...
4	87.0	65.0	US	Vintner's Reserve Wild Child Block	Oregon	Willamette Valley	Willamette Valley	Paul Gregutt	@paulgwine	Pinot Noir	Sweet Cheeks	bottling country wine companion hearty winter ...	Sweet Cheeks 2012 Vintner's Reserve Wild Child...
...	...	...	...	...	...	...	...	...	...	...	...	...	...

Appendix B

List of datasets

In this appendix, a list of all datasets used during the evaluation is provided, including additional information on what was used from the data to run the corresponding learning task.

B.1 String feature inference and processing

Some of the datasets listed below were used to evaluate both string feature inference and string feature processing. The datasets without model, task, and target were only used to evaluate string feature inference based on the manually assigned ground truth of the columns. The datasets with the previously mentioned components were used for both string feature inference (with manual labeling of string feature) and string feature processing (evaluation by running the associated task).

B.1.1 Coordinate

Digital altimetric data information - GPS. Columns: *latgms, loggms*. https://www.kaggle.com/mpwolke/cusersmarildownloadsgpscsv

B.1.2 Day

San Francisco Crime Classification. Columns: *DayOfWeek*. Model: *GradientBoostingClassifier*. Task: *Classification*. Target: *Category*. https://www.kaggle.com/kaggle/san-francisco-crime-classification

B.1.3 E-mail

Data_UK. Columns: *email*. https://www.kaggle.com/phool1804/data-uk

Enrico's Email Flows. Columns: *sender, receiver*. https://www.kaggle.com/emarock/enricos-email-flows

Indian Companies Registration Data [1857 - 2020]. Columns: *EMAIL_ADDR*. Model: *GradientBoostingClassifier*. Task: *Classification*. Target: *COMPANY_STATUS* https://www.kaggle.com/rowhitswami/all-indian-companies-registration-data-1900-2019

B.1.4 Filepath

Collection of Classification & Regression Datasets. Columns: *Image Index*. https://www.kaggle.com/balakrishcodes/others?select=xrayfull.csv

Hillary Clinton's Emails. Columns: *MetadataPdfLink*. https://www.kaggle.com/kaggle/hillary-clinton-emails?select=Emails.csv

Liver and Liver Tumor Segmentation. Columns: *filepath, liver_maskpath, tumor_maskpath*. https://www.kaggle.com/andrewmvd/lits-png?select=lits_df.csv

B.1.5 Month

FIFA 19 complete player dataset. Model: *GradientBoostingRegressor*. Task: *Regression*. Target: *Value*. Columns: *Joined*. https://www.kaggle.com/karangadiya/fifa19

Netflix Movies and TV Shows. Columns: *date_added*. https://www.kaggle.com/shivamb/netflix-shows

Metacritic-Game Releases by Score. Columns: *Date*. https://www.kaggle.com/abhishekdataset/metacriticgame-releases-by-score

B.1.6 Numerical

FIFA 19 complete player dataset. Columns: *LS, ST, RS, LW*. Model: *GradientBoostingRegressor*. Task: *Regression*. Target: *Value*. https://www.kaggle.com/karangadiya/fifa19

HR Analytics: Job Change of Data Scientists. Columns: *company_size*. https://www.kaggle.com/arashnic/hr-analytics-job-change-of-data-scientists?select=aug_train.csv

Students' Academic Performance Dataset. Columns: *StudentAbsenceDays*. https://www.kaggle.com/aljarah/xAPI-Edu-Data

B.1.7 Sentence

Wine Reviews. Columns: *description*. Model: *GradientBoostingRegressor*. Task: *Regression*. Target: *points* https://www.kaggle.com/zynicide/wine-reviews

World Development Indicators. Columns: *SpecialNotes, SystemOfNationalAccounts*. https://www.kaggle.com/worldbank/world-development-indicators

B.1.8 URL

FIFA 19 complete player dataset. Columns: *Photo, Flag, Club Logo*. Model: *GradientBoostingRegressor*. Task: *Regression*. Target: *Value*. https://www.kaggle.com/karangadiya/fifa19

Walmart Product Details 2020. Columns: *Product Url*. https://www.kaggle.com/promptcloud/walmart-product-details-2020

B.1.9 Zip code

Data_UK. Columns: *postal*. https://www.kaggle.com/phool1804/data-uk

OpenAddresses - Europe. Columns: *POSTCODE*. https://www.kaggle.com/openaddresses/openaddresses-europe?select=netherlands.csv

OpenAddresses - North America (excluding U.S.). Columns: *POSTCODE*. https://www.kaggle.com/openaddresses/openaddresses-north-america-excluding-us?select=bermuda.csv

House Price Data, England & Wales, 2015 to 2019. Columns: *SS2 6ST*. Model: *GradientBoostingRegressor*. Task: *Regression*. Target: *249995*. https://www.kaggle.com/dmaso01dsta/house-price-data-england-wales-2015-to-2019

B.2 Statistical type detection and determining the order in data

B.2.1 Nominal datasets

[**NeurIPS 2020**] **Data Science for COVID-19 (DS4C)**. Columns: *province, city.* https://www.kaggle.com/kimjihoo/coronavirusdataset?select=Case.csv

[**NeurIPS 2020**] **Data Science for COVID-19 (DS4C)**. Columns: *type, gov_policy.* https://www.kaggle.com/kimjihoo/coronavirusdataset?select=Policy.csv

AB_NYC_2019. Columns: *name, host_name, neighborhood_group, neighborhood, room_type.* https://www.kaggle.com/chadra/ab-nyc-2019

Automobile Dataset. Columns: *make.* https://www.kaggle.com/toramky/automobile-dataset

Craft Beers Dataset. Columns: *style.* https://www.kaggle.com/nickhould/craft-cans?select=beers.csv

Craft Beers Dataset. Columns: *city, state.* https://www.kaggle.com/nickhould/craft-cans?select=breweries.csv

FIFA 19 complete player dataset. Columns: *Nationality, Club.* https://www.kaggle.com/karangadiya/fifa19

FiveThirtyEight Comic Characters Dataset. Columns: *ALIGN, EYE, HAIR.* https://www.kaggle.com/fivethirtyeight/fivethirtyeight-comic-characters-dataset?select=dc-wikia-data.csv

HR Analytics: Job Change of Data Scientists. Columns: *city, major_discipline, company_type.* https://www.kaggle.com/arashnic/hr-analytics-job-change-of-data-scientists?select=aug_train.csv

IBM HR Analytics Employee Attrition & Performance. Columns: *Department, EducationField, JobRole.* Model: *GradientBoostingRegressor.* Task: *Regression.* Target: *MonthlyIncome.* https://www.kaggle.com/pavansubhasht/ibm-hr-analytics-attrition-dataset

Kickstarter Projects. Columns: *category, main_category, currency, country.* https://www.kaggle.com/kemical/kickstarter-projects?select=ks-projects-201612.csv

Mushroom Classification. Columns: *class, cap-shape, cap-surface, cap-color, bruises, odor, gill-attachment, gill-spacing, gill-size, gill-color, stalk-shape, stalk-root, stalk-surface-above-ring, stalk-surface-below-ring, stalk-color-above-ring, stalk-color-below-ring, veil-type, veil-color, ring-numer, ring-type, spore-print-color, population, habitat.*
https://www.kaggle.com/uciml/mushroom-classification

Pokemon with stats. Columns: *Type 1, Type 2.* https://www.kaggle.com/abcsds/pokemon

Ramen Ratings. Columns: *Brand, Variety, Style, Country.* https://www.kaggle.com/residentmario/ramen-ratings

Stroke Prediction Dataset. Columns: *work_type, smoking_status.* https://www.kaggle.com/fedesoriano/stroke-prediction-dataset

Students' Academic Performance Dataset. Columns: *PlaceOfBirth, GradeID, SectionID, Topic*. https://www.kaggle.com/aljarah/xAPI-Edu-Data

Students Performance in Exams. Columns: *race/ethnicity*. https://www.kaggle.com/spscientist/students-performance-in-exams

Wine Reviews. Columns: *country, province, region_1, variety*. Model: *GradientBoostingRegressor*. Task: *Regression*. Target: *points*. https://www.kaggle.com/zynicide/wine-reviews

B.2.2 Ordinal datasets

Some of the datasets listed below were used for both ordinality prediction and determining the order in ordinal data. The datasets without model, task, and target were only used to classify the ordinality based on the manually labeled ground truth of the columns. The datasets with the previously mentioned components were used for both ordinality prediction (with manual labeling of ordinality) and performance evaluation of FlairNLP (evaluation by running the associated task).

Amazon - Ratings (Beauty Products). Columns: *Rating*. https://www.kaggle.com/skillsmuggler/amazon-ratings?select=ratings_Beauty.csv

Audiology (Original) Data Set. Columns: *air, ar_c, ar_u, bone, o_ar_c, o_ar_u, speech*. https://archive.ics.uci.edu/ml/datasets/Audiology+%28Original%29

Basic Income Survey - 2016 European Dataset. Columns: *dem_education_level, awareness, vote, age_group*. https://www.kaggle.com/daliaresearch/basic-income-survey-european-dataset

Car Evaluation Data Set. Columns: *buying, maint, doors, persons, lug_boot, safety, class value*. Model: *GradientBoostingClassifier*. Task: *Classification*. Target: *Class Values*. https://archive.ics.uci.edu/ml/datasets/Car+Evaluation

Earthquake Magnitude, Damage and Impact. Columns: *damage_overall_colapse, damage_overall_leaning, damage_grade, technical_solution_proposed*. https://www.kaggle.com/arashnic/earthquake-magnitude-damage-and-impact?select=csv_building_damage_assessment.csv

Earthquake Magnitude, Damage and Impact. Columns: *education_level_household_head*. https://www.kaggle.com/arashnic/earthquake-magnitude-damage-and-impact?select=csv_household_demographics.csv

Hayes-Roth Data Set. Columns: *age, educational level, marital status*. https://archive.ics.uci.edu/ml/datasets/Hayes-Roth

Linux Gamers Survey, Q1 2016. Columns: *LinuxUserHowLong, DesktopLinuxGamerHowLong, HeavyGamer, LinuxExclusivity, LinuxGamingHabitChange, LinuxGamingHabitFuture, LinuxGamingMachineShared, FolksAroundYouAwareLinux, LinuxGamesPurchaseFrequency, SatisfactionSteam, SatisfactionGOG, SatisfactionHB, DistroChangeFrequency, DistroImpactPerformance, HardwareUpgradeIntent, AwarenessBrandedSteamMachines, AwarenessSteamController, AwarenessSteamLink, SteamMachineConceptLike, SteamMachinesExpandLinuxDoubtful, SteamMachinesLaunchEvaluation, SteamMachinesAwarenessAlienware, SteamMachinesAwarenessZotac, SteamMachinesAwarenessSyber, SteamMachinesWantToBuy, MachinesMaximumPrice, MachinesDIYIntent, SteamControllerPurchaseIntent, SteamOSEverTried, SteamIHSUage, SteamLinkPurchaseIntent, WINEUsageVanilla, PlayOnLinux, Crossover, WINEEvaluation*. https://www.kaggle.com/sanqualis/linuxgamerssurvey

Nursery Data Set. Columns: *parents, has_nurs, form, housing, finance, social, health.* Model: *GradientBoostingClassifier.* Task: *Classification.* Target: *Nursery.* https://archive.ics.uci.edu/ml/datasets/Nursery

Solar Flare Data Set. Columns: *activity, evolution, previous_24h_flare_activity_code, area.* Model: *GradientBoostingRegressor.* Task: *Regression.* Target: *C-class, M-class, X-class.* https://archive.ics.uci.edu/ml/datasets/Solar+Flare

Soybean (Large) Data Set. Columns: *precip, temp, crop-hist, area-damaged, severity, stem-cankers.* Model: *GradientBoostingClassifier.* Task: *Classification.* Target: *class.* https://archive.ics.uci.edu/ml/datasets/Soybean+%28Large%29

Appendix C

Additional tables

C.1 Evaluation determining order

Dataset	Column	FlairNLP order com -pared to ground truth	Edit distance	Accuracy
Audiology	Air	4, 1, 2, 3, 5	2	0.60
	ar_c, ar_u, o_ar_c, o_ar_u	1, 2, 3	0	1.0
	Bone	1, 2, 3, 4	0	1.0
	Speech	1, 2, 3, 4, 5, 6	0	1.0
Soybean-large	Precip, temp	2, 1, 3	2	0.33
	Crop-hist	3, 2, 4, 1	3	0.25
	Area-damaged	1, 2, 3, 4	0	1.0
	Severity	2, 3, 1	2	0.33
	Stem-cankers	1, 2, 4, 3	2	0.50
Basic income survey	Dem_education_level	2, 1, 3, 4	2	0.50
	Awareness	1, 2, 3, 4	0	1.0
	Vote	1, 3, 2, 4, 5	2	0.60
	Age_group (n)	3, 2, 1	2 (0)	0.33 (1.0)
building_damage_assessment	Collapse, leaning	1, 3, 2	2	0.33
	Damage_grade	2, 4, 1, 3, 5	4	0.20
	Proposed	2, 1, 3, 4	2	0.50
Building_household_demographics	Household head	1, 14, 15, 2, 19, 11, 4, 9, 5, 10, 12, 6, 8, 3, 16, 7, 13, 18, 17	15	0.21
Car evaluation	Buying, maint	1, 2, 3, 4	0	1.0
	Doors	2, 1, 3, 4	2	0.50
	Persons	1, 2, 3	0	1.0
	Lug_boot	1, 3, 2	2	0.33
	Safety	1, 2, 3	0	1.0
	Class value	1, 4, 2, 3	2	0.50
Flare	Activity (n)	2, 1	2 (0)	0.0 (1.0)
	Evolution	1, 2, 3	0	1.0
	Flare_activity	1, 2	0	1.0
	Area	1, 2	0	1.0
Hayes-roth	Age	1, 4, 3, 2	2	0.50
	Education level	3, 1, 4, 2	4	0.0
	Marital status	4, 1, 3, 2	3	0.25
Nursery	Parents	2, 1, 3	2	0.33
	Has_nurs	2, 3, 5, 4, 1	3	0.40
	Form	3, 2, 1, 4	2	0.50
	Children	1, 3, 2, 4	2	0.50
	Housing	2, 3, 1	2	0.33

	Finance	1, 2	0	1.0
	Social (n)	3, 2, 1	2 (0)	0.33 (1.0)
	Health	1, 2, 3	0	1.0
Ratings_beauty	Rating	1, 2, 3, 4, 5	0	1.0
Steam-linux-survey-v2	LinuxUserHowLong, DesktopLinuxGamerHowLong	1, 2, 3, 4, 7, 5, 6	2	0.71
	HeavyGamer	1, 2, 3, 5, 4	2	0.60
	LinuxExclusivity	6, 1, 5, 3, 4, 2	4	0.33
	LinuxGamingHabitChange	1, 3, 2, 4, 5	2	0.60
	LinuxGamingHabitFuture	1, 3, 2, 5, 4	3	0.40
	LinuxGamingMachineShared	1, 2, 3	0	1.0
	FolksAroundYouAwareLinux	1, 2, 4, 3	2	0.50
	LinuxGamesPurchaseFrequency	1, 2, 5, 4, 3, 6, 7	2	0.71
	SatisfactionSteam	1, 3, 2, 4, 5	2	0.60
	SatisfactionGOG, SatisfactionHB	1, 2, 4, 3, 5, 6	2	0.67
	DistroChangeFrequency	1, 4, 7, 5, 6, 2, 3	5	0.29
	DistroImpactPerformance	1, 2, 3, 4	0	1.0
	HardwareUpgradeIntent	3, 1, 2, 4, 5	2	0.60
	AwarenessBrandedSteamMachines, AwarenessSteamController, AwarenessSteamLink	1, 3, 2, 4, 5	2	0.60
	SteamMachineConceptLike	1, 2, 3, 4	0	1.0
	SteamMachinesExpandLinuxDoubtful	1, 2, 3, 4, 5	0	1.0
	SteamMachinesLaunchEvaluation	3, 2, 4, 5, 1, 6	3	0.50
	SteamMachinesAwarenessAlienware	4, 2, 5, 1, 3, 6	4	0.33
	SteamMachinesAwarenessZotac	3, 1, 2, 4	2	0.50
	SteamMachinesAwarenessSyber	4, 2, 1, 3, 5	3	0.40
	SteamMachinesWantToBuy	1, 2, 3, 4, 5, 7, 6	2	0.71
	MachinesMaximumPrice	1, 3, 2, 4, 5, 6, 7	2	0.71
	MachinesDIYIntent	1, 3, 2, 4, 5, 6	2	0.67
	SteamControllerPurchaseIntent	1, 2, 3, 5, 4, 6, 7	2	0.71
	SteamOSEverTried	2, 1, 3	2	0.33
	SteamIHSUsage	2, 3, 1, 4	2	0.50
	SteamLinkPurchaseIntent	1, 2, 3, 5, 6, 7, 4	3	0.50
	WINEUsageVanilla, PlayOnLinux, Crossover	2, 1, 3, 4, 5	2	0.60
	WINEEvaluation	1, 2, 3, 4, 5	0	1.0

Table C.1: Detailed results of using FlairNLP to determine the order. Note that a (n) indicates that the result can potentially be negated.

Dataset	Column	Order compared to ground truth	Edit distance	Accuracy
audiology	Air	2, 3, 1, 5, 4	3	0.4
	ar_c, ar_u, o_ar_c, o_ar_u	1, 3, 2	2	0.33
	Bone	3, 4, 2, 1	4	0.0
	Speech	5, 4, 3, 1, 6, 2	5	0.17
backup-large	Precip, temp	3, 1, 2	2	0.33
	Crop-hist	1, 4, 3, 2	2	0.5
	Area-damaged	2, 1, 3, 4	2	0.5
	Severity	1, 2, 3	0	1.0
	Stem-cankers	4, 3, 1, 2	4	0.0
basic_income_dataset_dalia	Dem_education_level	4, 2, 3, 1	2	0.5
	Awareness	2, 1, 3, 4	2	0.5
	Vote	3, 2, 4, 1, 5	3	0.4
	Age_group	1, 2, 3	0	1.0
building_damage_assessment	Collapse, leaning	1, 2, 3	0	1.0
	Damage_grade	1, 2, 3, 4, 5	0	1.0
	Proposed	3, 2, 1, 4	2	0.5
building_household_demographics	Household head	16, 3, 12, 4, 5, 6, 7, 8, 9, 10, 11, 1, 13, 17, 15, 2, 19, 18, 13	9	0.53
car-evaluation	Buying, maint	3, 1, 2, 4	2	0.5
	Doors	1, 2, 3, 4	0	1.0
	Persons	1, 2, 3	0	1.0
	Lug_boot	3, 2, 1	2	0.33
	Safety	3, 1, 2	2	0.33
	Class value	2, 3, 1, 4	2	0.5
flare	Activity	2, 1	2	0.0
	Evolution	1, 3, 2	2	0.33
	Flare_activity	2, 1	2	0.0
	Area	2, 1	2	0.0
hayes-roth	Age	1, 3, 4, 2	2	0.5
	Education level	3, 2, 1, 4	2	0.5
	Marital status	4, 3, 2, 1	4	0.0
nursery	Parents	3, 2, 1	2	0.33
	Has_nurs	4, 3, 2, 1, 5	4	0.2
	Form	1, 2, 4, 3	2	0.5
	Children	1, 2, 3, 4	0	1.0
	Housing	1, 3, 2	2	0.33
	Finance	1, 2	0	1.0
	Social	1, 3, 2	2	0.33
	Health	3, 2, 1	2	0.33
ratings_beauty	Rating	3, 5, 4, 2, 1	5	0.0
Linux	LinuxUserHowLong, DesktopLinuxGamerHowLong	6, 5, 2, 4, 3, 1, 7	5	0.0
	HeavyGamer	3, 1, 2, 5, 4	3	0.4
	LinuxExclusivity	6, 3, 4, 2, 5, 1	4	0.33
	LinuxGamingHabitChange	2, 4, 1, 5, 3	4	0.2
	LinuxGamingHabitFuture	2, 3, 1, 4	2	0.5
	LinuxGamingMachineShared	1, 2, 3	0	1.0
	FolksAroundYouAwareLinux	1, 2, 4, 3	2	0.5
	LinuxGamesPurchaseFrequency	6, 3, 4, 7, 1, 2, 5	6	0.14
	SatisfactionSteam	3, 4, 2, 1, 5	4	0.2
	SatisfactionGOG, SatisfactionHB	4, 5, 3, 2, 1, 6	4	0.33
	DistroChangeFrequency	7, 4, 3, 2, 1, 5, 6	5	0.29
	DistroImpactPerformance	2, 1, 4, 3	3	0.25
	HardwareUpgradeIntent	1, 2, 3, 5, 4	2	0.6

	AwarenessBrandedSteamMachines, AwarenessSteamController, AwarenessSteamLink	2, 5, 1, 4, 3	4	0.2
	SteamMachineConceptLike	1, 3, 4, 2	2	0.5
	SteamMachinesExpandLinuxDoubtful	2, 5, 1, 3, 4	4	0.2
	SteamMachinesLaunchEvaluation	3, 1, 5, 4, 6, 2	5	0.17
	SteamMachinesAwarenessAlienware	1, 6, 5, 4, 2, 3	4	0.33
	SteamMachinesAwarenessZotac	1, 4, 2, 3	2	0.5
	SteamMachinesAwarenessSyber	1, 5, 4, 2, 3	4	0.2
	SteamMachinesWantToBuy	6, 2, 1, 5, 3, 4	5	0.17
	MachinesMaximumPrice	2, 3, 4, 5, 6, 1, 7	2	0.71
	MachinesDIYIntent	6, 1, 5, 3, 4, 2	4	0.33
	SteamControllerPurchaseIntent	7, 3, 6, 5, 1, 4, 2	6	0.14
	SteamOSEverTried	1, 3, 2	2	0.33
	SteamIHSUsage	1, 2, 4, 3	2	0.5
	SteamLinkPurchaseIntent	7, 3, 4, 1, 2, 6, 5	5	0.29
	WINEUsageVanilla, PlayOnLinux, Crossover	1, 2, 3, 5, 4	2	0.6
	WINEEvaluation	3, 2, 4, 5, 1	3	0.4

Table C.2: Detailed results of the baseline to determine the order.

From strings to data science: a practical framework for automated string handling

John W. van Lith (✉) and Joaquin Vanschoren

Faculty of Mathematics and Computer Science, Eindhoven University of Technology
jlith1997@gmail.com

Abstract. Many machine learning libraries require that string features be converted to a numerical representation for the models to work as intended. Categorical string features can represent a wide variety of data (e.g., zip codes, names, marital status), and are notoriously difficult to preprocess automatically. In this paper, we propose a framework to do so based on best practices, domain knowledge, and novel techniques. It automatically identifies different types of string features, processes them accordingly, and encodes them into numerical representations. We also provide an open source Python implementation[1] to automatically preprocess string data in tabular datasets and demonstrate promising results on a wide range of datasets.

Keywords: Data cleaning · String features · Automated data science.

1 Introduction

Datasets acquired from the real world often contain categorical string data, such as zip codes, names, or occupations. Many machine learning algorithms require that such string features be converted to a numerical representation to work as intended. Depending on the type of data, specific processing is required. For example, geographical string data (e.g., addresses) may be best expressed by latitudes and longitudes. Data scientists are required to manually preprocess such unrefined data, requiring a significant amount of time, up to 60% of their day [10]. Automated data cleaning tools exist but often fail to robustly address the wide variety of categorical string data. This paper presents a framework that systematically identifies various types of categorical string features in tabular datasets and encodes them appropriately. We also present an open-source Python implementation that we evaluate on a wide range of datasets.

2 Challenges and Related Work

Our framework addresses a range of challenges. First, *type detection* aims to identify predefined 'types' of string data (e.g., dates) that require special preprocessing. Probabilistic Finite State Machines (PFSMs) [9] are a practical solution based on regular expressions and can produce type probabilities. They can also detect missing or anomalous values, such as numeric values in a string column.

[1] Open-source library: https://github.com/ml-tue/automated-string-cleaning

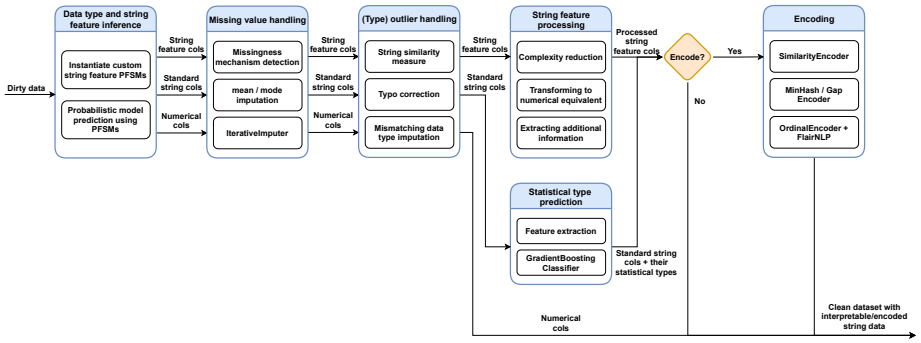


Fig. 1: The overall workflow of the framework.

Statistical type inference predicts a feature's statistical type (e.g., ordinal or categorical) based on the intrinsic data distribution. Valera et al. [29] use a Bayesian approach to discovers whether features are ordinal, categorical, or real-valued, although some manual assessment is still needed. Other techniques that predict classes using features of the data use random forest and gradient boosting classifiers [11]. Similar techniques could be leveraged to achieve class prediction for the statistical type.

Encoding techniques convert categorical string data to numeric values, which is challenging because there may be small errors (e.g., typos) and intrinsic meaning (e.g., a time or location). Cerda et al. [7, 8] use string similarity metrics and min-hashing to tackle morphological variants of the same string. Geocoding APIs (e.g., pgeocode and geopy) can convert geographical strings to coordinates [3, 5]. For ordinal string data, heuristic approaches exist that could determine order based on antonyms, superlatives, and quantifiers, e.g. using WordNet [22] or sentiment intensity analyzers (e.g. VADER, TextBlob, and FlairNLP) [6, 13, 20].

Methods have been proposed exist that recognize, categorize, and process different string entities based on regular expressions [28] or domain knowledge that can outperform human experts [12, 14]. Data cleaning tools exist that are manually operated [1, 2, 4], semi-automated [23, 30], or fully automated [15–17, 21, 26]. At present, however, these automated tools do not focus on string handling [15, 16, 26] or they focus on specific steps such as error correction [17, 21].

3 Methodology

Our framework, shown in Fig. 1, is designed to detect and appropriately encode different types of string data in tabular datasets. First, we use PFSMs to infer whether a column is numerical, a known type of string feature (e.g., a date), or any other type of 'standard' string data. Based on this first categorization, appropriate missing value and outlier handling methods are applied to the entire dataset to repair inconsistencies. Next, columns with recognized string types go through intermediate type-specific processing, while the remaining columns are classified based on their statistical type (e.g., nominal or ordinal). Finally, the data is encoded by applying the most fitting encoding for each feature.

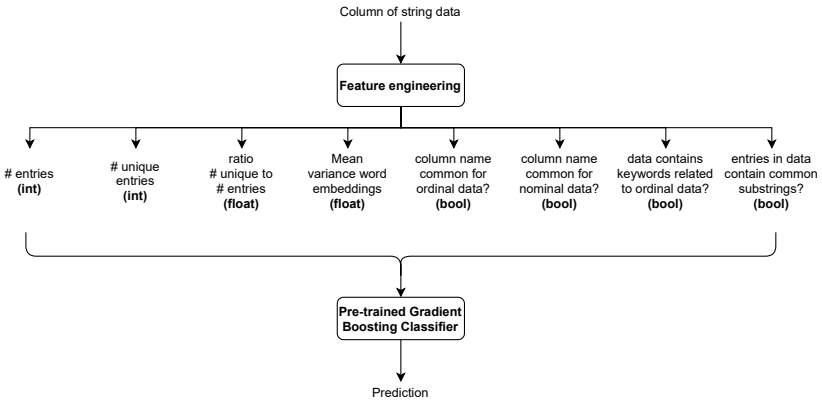


Fig. 2: Workflow of predicting the statistical type for a given string column.

3.1 String feature inference

In the first step, we build on PFSMs and the ptype library [9]. We created PFSMs based on regular expressions for nine types of string features: coordinates, days, e-mail addresses, filepaths, months, numerical strings, sentences, URLs, and zip codes. A detailed description for each of these can be found in Appendix A. The PFSMs were trained on a range of datasets, listed in Appendix C, for which we manually annotated the ground truth string types.

3.2 Handling missing values and outliers

Next, missing values are imputed based on the feature type and the missingness of the data [19, 27] (missing at random, missing not at random, missing completely at random) using mean/mode imputation or a multivariate imputation technique [24]. Minor typos are corrected using string metrics [18], and data type outliers are corrected if applicable to ensure robustness in the remaining steps.

3.3 Processing inferred string features

Next, we perform intermediate processing of all the string types identified by the PFSMs. First, we simplify the strings, for instance, by removing redundant words in sentences. Second, we assign or perform specific encoding techniques, such as replacing a date with year-month-day values. Third, we include additional information, such as fetching latitude and longitude values for zip codes.

3.4 Statistical type prediction

String features not identified by the PFSMs are marked as 'standard' strings. For these, we infer their statistical type, i.e., whether they contain ordered (ordinal) data or unordered (nominal) data. The prediction is based on eight properties extracted from the feature, shown in Fig. 2, including the uniqueness of the string values, whether the column name or values suggests ordinality, and whether a GloVe word embedding of the string values shows clear relationships between the values. The rationale behind these is explained in Appendix B. These features are fed to a gradient boosting classifier to predict the statistical type. This classifier was trained on real-world features, manually annotated, listed in Appendix C.

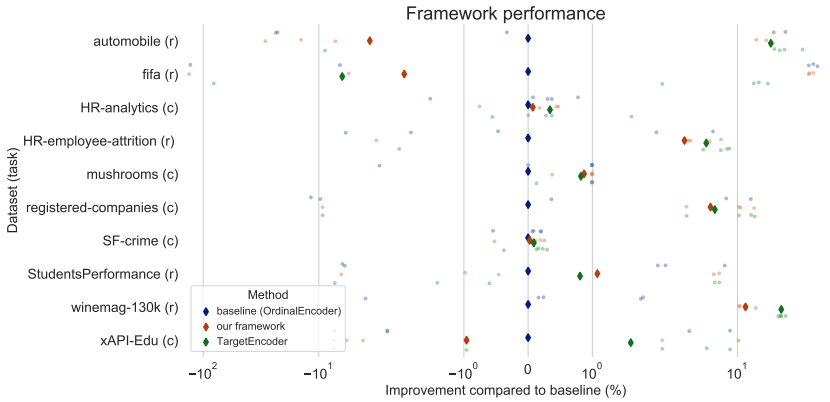


Fig. 3: Relative performance of our framework against baseline preprocessing.

3.5 Encoding

Finally, an appropriate encoding is applied for each categorical string feature. For the string types identified by the PFSMs, a predefined encoding is applied, per Appendix A. For nominal data, we use the dirty_cat library due to its robustness to morphological variants and high cardinality features [7, 8]. We apply the similarity, Gamma-Poisson, and min-hash encoders on nominal data when the cardinality is below 30, below 100, and at least 100, respectively. For ordinal data, a simple ordinal encoder is applied where the ordering is defined by a text sentiment intensity analyzer (FlairNLP [6]).

4 Evaluation

We evaluate our framework and its individual components on a range of real-world datasets with categorical string features, listed in Appendix C. Performance is evaluated using the downstream performance of gradient boosting models trained on the encoded data. These models are intrinsically robust against high-dimensional encodings, hence ensuring a stringent evaluation. We use stratified 5-fold cross-validation in all experiments. The evaluation metrics are accuracy for classification tasks and MAE for regression tasks.

Global framework evaluation. First, we evaluate the framework as a whole and compare it to a baseline where the data is manually preprocessed using mean/mode imputation for missing values and ordinal or target encoding for the categorical string features. Fig. 3 shows the relative performance differences for each of the five folds and their mean. These results indicate that our framework can be a suitable automated alternative to string handling. On some datasets, the automated encodings prove suboptimal, which warrants further study.

Feature type inference. In Table 1, we compare the predictions of our PFSMs against the ground truth feature types. Most PFSMs report perfect accuracy. Filepaths and sentences are detected with 70-80% accuracy. In the latter, the exact format of the data can often be unexpected. Outliers are also detected correctly, except for sentence PFSMs, where out of the 130217 entries, 42158 entries were false positives, which is certainly a point for improvement.

String feature	Nr. of columns	Nr. of correctly inferred columns	Accuracy	Nr. of false negative outliers	Ratio of false negative outliers
Coordinate	2	2	1.0	0	-
Day	1	1	1.0	0	-
E-mail	4	4	1.0	0	-
Filepath	5	4	0.80	0	-
Month	3	3	1.0	0	-
Numerical	6	6	1.0	0	-
Sentence	4	3	0.75	42158	0.32
URL	4	4	1.0	0	-
Zip code	3	3	1.0	0	-

Table 1: Results of string feature inference using PFSMs.

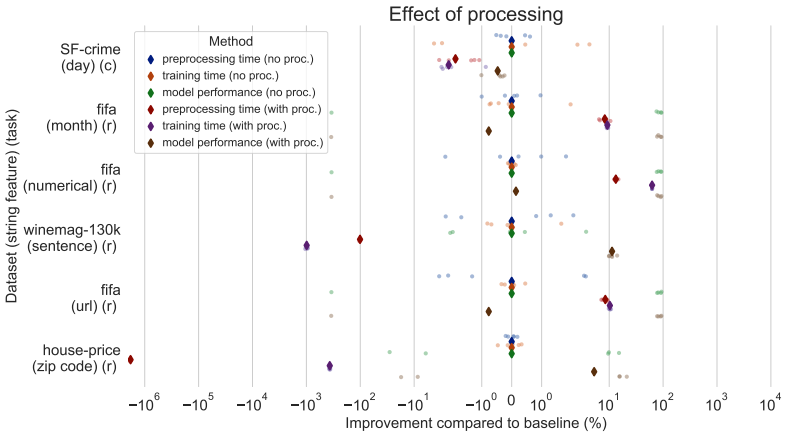


Fig. 4: Difference in performance metrics (accuracy/MAE, preprocessing time, and training time) with and without processing of known string feature types.

Processing inferred string features. Fig. 4 compares the performance of our framework with and without the intermediate processing for the string feature types identified by the PFSMs. For some features, this processing causes a 10% performance improvement, while on others, it remains about the same. This processing does require extra processing time, caused by API latency and text processing, yet it seems worth the extra time for zip codes and sentences. Moreover, the reduced string complexity (removing redundant words) and conversion of numerical strings (e.g. '> 10') into numerical representations reduce the training time by at least ten percent on half of the datasets.

Statistical type prediction. Tables 2a and 2b evaluate the gradient boosting classifier that predicts whether standard string features are nominal or ordinal, by comparison against the ground truth using leave-one-out cross-validation. These predictions are highly accurate, with very few misclassifications. Hence, our eight extracted features are highly indicative of ordinality in the feature values.

Ordinal encoding. Finally, we compare the ordinal encoding based on sentiment intensity (FlairNLP) vs. the baseline ordinal encoding by comparing them to an oracle with the ground-truth ordering. Table 2c and Fig. 5 show that FlairNLP significantly outperforms the baseline, although the effect on down-

Metric	Score
Accuracy	0.980 ± 0.14
F1 score	0.978
Precision	0.971
Recall	0.985
AUC score	0.980

(a) Statistical type prediction scores

Actual class \ Predicted class	Ordinal	Nominal
Ordinal	79	2
Nominal	1	67

(b) Statistical type prediction: Confusion matrix

Ordering method	Rank Correlation
Baseline	0.1562 ± 0.5313
FlairNLP	**0.7189 ± 0.5356**

(c) Spearman Rank Correlation for ordinal encoders vs. ground truth ordering.

Table 2: Results from various modules of the framework.

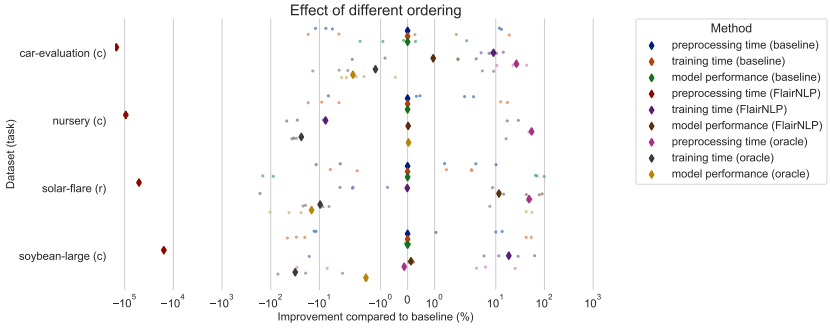


Fig. 5: Difference in performance metrics between different ordering strategies.

stream model performance is limited ($< 1\%$), and FlairNLP does require significantly more preprocessing time.

5 Conclusions and future work

The automation of data cleaning is still a fledgling open research field. We presented a framework that combines state-of-the-art techniques and additional novel components to enable automated string data cleaning. This framework shows promising results, and some of its novel components perform very well, especially in terms of identifying special types of categorical string data and adequately processing and encoding them. However, several challenges remain. First, string feature type inference using PFSMs based on regular expressions is sensitive to the exact formatting of strings. More robust techniques are needed, such as sub-pattern matching and better use of type probabilities in subsequent processing. For instance, if it is only 60% certain that a string feature represents a date, a more robust encoding is needed, or a human should be brought in the loop. Second, the string type-specific processing and final encoding were suboptimal or no better than the baseline on some datasets. These provide interesting cases for further study. Finally, the encoding of ordinal string data still leaves room for improvement. The sentiment intensity-based encoding has shown to perform well on some aspects and poorly on others. We believe that more sophisticated approaches are possible, e.g., paying special attention to numbers appearing in the string data. Overall, we hope that this framework and open-source implementation will speed up research in these areas.

Acknowledgements The authors would like to thank Marcos de Paula Bueno on his valuable feedback on this paper.

References

1. Data Ladder. https://dataladder.com/, https://dataladder.com/
2. DataCleaner. https://datacleaner.github.io/, https://datacleaner.github.io/
3. geopy 2.1.0. https://github.com/geopy/geopy
4. OpenRefine. https://openrefine.org/
5. Roman Yurchark. pgeocode 0.3.0. https://github.com/symerio/pgeocode
6. Akbik, A., Bergmann, T., Blythe, D., Rasul, K., Schweter, S., Vollgraf, R.: FLAIR: An easy-to-use framework for state-of-the-art NLP. In: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations). pp. 54–59 (2019)
7. Cerda, P., Varoquaux, G.: Encoding high-cardinality string categorical variables. IEEE Transactions on Knowledge and Data Engineering (2020)
8. Cerda, P., Varoquaux, G., Kégl, B.: Similarity encoding for learning with dirty categorical variables. Machine Learning **107**(8-10), 1477–1494 (2018)
9. Ceritli, T., Williams, C.K., Geddes, J.: ptype: probabilistic type inference. Data Mining and Knowledge Discovery **34**(3), 870–904 (2020). https://doi.org/10.1007/s10618-020-00680-1
10. CrowdFlower: Data Science Report (2016)
11. Grabczewski, K., Jankowski, N.: Feature selection with decision tree criterion. In: Fifth International Conference on Hybrid Intelligent Systems (HIS'05). pp. 6–pp. IEEE (2005)
12. Hardesty, L.: Automating big-data analysis (Oct 2015), https://news.mit.edu/2015/automating-big-data-analysis-1016
13. Hutto, C., Gilbert, E.: Vader: A parsimonious rule-based model for sentiment analysis of social media text. In: Proceedings of the International AAAI Conference on Web and Social Media. vol. 8 (2014)
14. Kanter, J.M., Veeramachaneni, K.: Deep Feature Synthesis: Towards automating data science endeavors. In: 2015 IEEE international conference on data science and advanced analytics (DSAA). pp. 1–10. IEEE (2015)
15. Krishnan, S., Wang, J., Franklin, M.J., Goldberg, K., Kraska, T., Milo, T., Wu, E.: SampleClean: Fast and Reliable Analytics on Dirty Data. IEEE Data Eng. Bull. **38**(3), 59–75 (2015)
16. Krishnan, S., Wang, J., Wu, E., Franklin, M.J., Goldberg, K.: Activeclean: Interactive data cleaning for statistical modeling. Proceedings of the VLDB Endowment **9**(12), 948–959 (2016)
17. Krishnan, S., Wu, E.: AlphaClean: Automatic Generation of Data Cleaning Pipelines. CoRR **abs/1904.11827** (2019), http://arxiv.org/abs/1904.11827
18. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. In: Soviet physics doklady. vol. 10, pp. 707–710. Soviet Union (1966)
19. Little, R.J.: A test of missing completely at random for multivariate data with missing values. Journal of the American statistical Association **83**(404), 1198–1202 (1988)
20. Loria, S.: TextBlob Documentation. Release 0.15 **2** (2018)

21. Mahdavi, M., Abedjan, Z., Castro Fernandez, R., Madden, S., Ouzzani, M., Stonebraker, M., Tang, N.: Raha: A configuration-free error detection system. In: Proceedings of the 2019 International Conference on Management of Data. pp. 865–882 (2019)
22. Miller, G.A.: WordNet: a lexical database for English. Communications of the ACM **38**(11), 39–41 (1995)
23. Musleh, M., Ouzzani, M., Tang, N., Doan, A.: CoClean: Collaborative Data Cleaning. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. pp. 2757–2760 (2020)
24. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)
25. Pennington, J., Socher, R., Manning, C.D.: Glove: Global vectors for word representation. In: Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP). pp. 1532–1543 (2014)
26. Rekatsinas, T., Chu, X., Ilyas, I.F., Ré, C.: HoloClean: Holistic Data Repairs with Probabilistic Inference. CoRR **abs/1702.00820** (2017), `http://arxiv.org/abs/1702.00820`
27. Rubin, D.B.: Inference and missing data. Biometrika **63**(3), 581–592 (1976)
28. Shahbaz, M., McMinn, P., Stevenson, M.: Automated discovery of valid test strings from the web using dynamic regular expressions collation and natural language processing. In: 2012 12th International Conference on Quality Software. pp. 79–88. IEEE (2012)
29. Valera, I., Ghahramani, Z.: Automatic Discovery of the Statistical Types of Variables in a Dataset. In: Precup, D., Teh, Y.W. (eds.) Proceedings of the 34th International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 70, pp. 3521–3529. PMLR (06–11 Aug 2017)
30. Zöller, M.A., Huber, M.F.: Benchmark and Survey of Automated Machine Learning Frameworks. arXiv preprint arXiv:1904.12054 (2019)

A Details on string features and processing

Coordinate This is a string feature that represent GPS or Degree-Minute-Second (DMS) coordinates such as `N29.10.56 W90.00.00`, `N29:10:56`, and `29° 10'56.22"N`. Coordinates can be distinguished from other string features based on the following characteristics:

- Two sequences of at most two digits and a third sequence which is a float with at most two digits before the decimal point.
- A character that separates the three sequences of digits (e.g., `.` or `:`). In the context of DMS coordinates, these characters are `°`, `'`, and `"` respectively.
- A cardinal direction at the beginning or at the end of the string (i.e., `N`, `E`, `S`, and `W`).

This string feature is processed as follows. First, the string feature is split up into two separate parts for each coordinate in the entry, in which one part represents the cardinal direction of the coordinate and the other part represents the numerical information. Second, the current format of the coordinate string feature is converted into the corresponding decimal latlong value. The string feature is formatted using degrees, minutes, and seconds. This format can be converted to representative decimal values using the following formula[2]:

$$\text{decimal} = \begin{cases} -(\text{degrees} + \frac{\text{minutes}}{60} + \frac{\text{seconds}}{3600}) & \text{if } c \in \{S, W\} \\ \text{degrees} + \frac{\text{minutes}}{60} + \frac{\text{seconds}}{3600} & \text{otherwise} \end{cases}$$

Last, additional information is extracted in case the string feature contains both the latitude and the longitude values. The additional information that can be extracted includes Earth-Centered Earth-Fixed representations of the latlong value and postal codes and country codes via geopy [3]. If the user decides to encode the data, the extracted postal and country codes will receive a nominal encoding in the final step of the framework.

Day This string feature represent the names of the seven days in the week such as `Monday`. These names can appear in data in several formats. For example, `Monday` can be written as `Mon` and `Mo`. Days can be distinguished from other string features based on the following characteristics:

- A prefix of at least two characters, indicating the day of the week (e.g., `Mo` for Monday, `Th` for Thursday, etc.).
- The suffix `day`, if present.
- A distinct set of characters that comes after the prefix and before the suffix (e.g., if the string is `Thursday`, then `Th` should be followed by `urs`).

[2] Taken from "Geographic coordinate conversion" at `https://en.wikipedia.org/wiki/Geographic\_coordinate\_conversion`

As this string feature is the least complex out of all inferred string features, it is also the most simple to process. Considering only the first two characters for days is the most reduction that can be done while still being able to make a distinction between each unique day of the week. If the user decides to encode the data, this string feature will receive a nominal encoding in the final step of the framework.

E-mail This string feature represents all valid e-mail addresses from any domain such as `Jane@tue.nl` and `john.doe@hotmail.co.uk`. This feature can be distinguished from others based on the following characteristics:

- The character `@` which is between two sets of characters.
- A substring in front of the `@` (i.e., the name of the e-mail) which is composed of valid characters (e.g., the e-mail address `#@*%#$@hotmail.com` is invalid as the characters before the final `@` cannot be included in an e-mail name).
- A substring that comes after the `@` which is composed of valid characters and at least one dot inbetween those characters (e.g., `name@hotmail` is not a valid e-mail address as the domain name is incomplete).

This string feature is processed as follows. We first remove the longest common suffix of all entries. Then, additional special characters are removed to simplify the values. If the user decides to encode the data, this string feature will receive a nominal encoding in the final step of the framework.

Filepath This string feature represents paths within a local system such as `C:/Windows/` and `C:/Users/Documents`. Filepaths can be distinguished from other string features by the following characteristics:

- A series of substrings which are separated from each other using either `/` or `\` (e.g., `home/users`).
- Each substring cannot contain any of the following characters: `\/:*?"<>|`
- If present, a prefix that represents the root disk or a sequence of dots followed by a slash or a backslash (e.g., `C:/`, `../`, etc.).

Processing this string feature is similar to how e-mail addresses are processed and is aimed to reduce string complexity. For this feature, the longest common prefix and suffix are removed from all entries and all special characters are removed. If the user decides to encode the data, this string feature will receive a nominal encoding in the final step of the framework.

Month This string feature represents the non-numerical representation of a month with or without year and day and can be distinguished based on the following criteria:

- A prefix of at least three characters, representing a unique month (e.g., `Apr`).

– If present, the remaining substring that comes after the prefix (e.g., `il` comes after the prefix `Apr`).
– If present, a sequence of at most two digits before or after the month which represents a day in the month (e.g., `1 January` or `January 1`).
– If present, a sequence of at most four digits or a sequence with prefix `'` followed by two digits that comes after the month which represents the year (e.g., `January 2000` or `January '00`). Both day and year can be present at the same time.

Processing this string feature is based on the format that is being presented. Each format is split up into individual components that represent either a day, month, or year. The key step in this procedure is to ensure that the string representative of the month is turned into the corresponding numerical representation. After this key step is performed, all values are concatenated to each other according to the format `yyyymmdd`. The overall workflow of this processing step is depicted in Fig. 6. As the string feature is already transformed to its numerical representation, no encoding would be required if the user requested so.

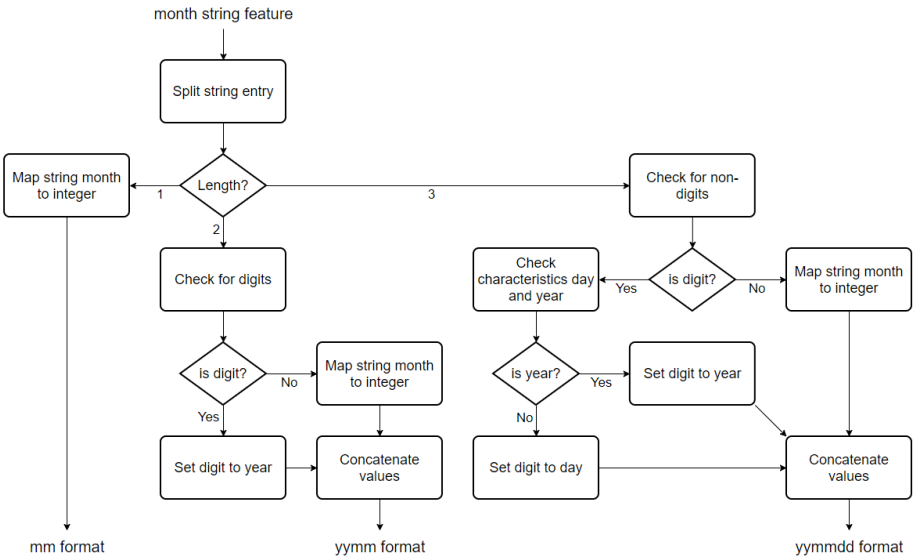


Fig. 6: The overall workflow for processing month string features.

Numerical There are a variety of entries that are relatively easy for users to distinguish as numerical values but are usually inferred as strings by any type detection or inference technique due to the presence of certain non-numerical characters. It is therefore important to categorize such entries as a string feature

to properly handle and process them. We obtain numerical string features using any of the following characteristics:

- Between two sequences of digits, one of the following characters: `-+_/:;&'` A space or the substring `to` is also applicable (e.g., `100 to 200`).
- Before a single sequence of digits, any of the following words: `Less than`, `Lower than`, `Under`, `Below`, `Greater than`, `Higher than`, `Over`, `Above`.
- Before or after a single sequence of digits, any of the following characters: `<>+$%=`

Numerical string features are processed based on what they represent. If the string feature represents a range of values, the mean of the range is calculated for each entry. After that, the range entries are encoded according to the numerical order of the ranges. If the string feature does not represent a range, we remove all special characters and consider all resulting numbers as separate numerical features. As the string feature is already transformed to its numerical representation, no encoding would be required if the user requested so.

Sentence This string feature is composed of a sequence of words, typically found in datasets that contain reviews or descriptions. It is slightly more difficult to express this string feature as a regular expression compared to the others because of its overlapping characteristics with regular string entries that consists of a couple of words. However, it is still possible to perform string feature inference for sentences based on the following characteristics:

- A substring of characters followed by a space for at least five times (i.e., the entry is at least six words long).

The goal for processing sentence string features is to remove redundancy in the entry and to make these more relevant for use in tabular data. The technique used to achieve this goal is the NLTK word tokenizer, which takes a sentence and divides these into tuples containing each word and their associated part of speech. Then, every word that is associated with a noun is joined together with a space into a single string which is then passed on. The result is a group of nouns that are supposed to represent the essence of the sentence and are ready to be encoded in the next step of the library. If the user decides to encode the data, this string feature will receive a nominal encoding in the final step of the framework.

URL This string feature represents any link to a website or domain such as `https://www.tue.nl/` and `http://canvas.tue.nl/login`. The characteristics of this string feature is similar to that of filepaths, with a few exceptions:

- An optional suffix which represents a certain protocol (e.g., `http://`).
- A series of at most four character sequences, separated from each other by a dot (e.g., `www.google.com` or `google.com`). Note that the last (pair of) sequence(s) contain(s) at most three characters.

Processing this string feature follows the same procedure as filepaths. If the user decides to encode the data, this string feature will receive a nominal encoding in the final step of the framework.

Zip code This string feature represents zip or postal codes from a handful of countries. Note that we are only able to infer zip codes that contain non-numerical characters as numerical-only zip codes are much more difficult to infer using PFSMs without overlapping actual numerical features.

Processing this string feature is mainly done to extract additional information from each entry. In our case, we make use of the geopy library to fetch the latitude, longitude, and country code of the zip code. Furthermore, we also calculate the ECEF coordinate using the latitude and longitude values. If the user decides to encode the data, the zip code string feature will receive a nominal encoding in the final step of the framework.

B Details on ordinality feature extraction

- *The total number of rows in the column*: It is possible that the number of rows in combination with other extracted features can increase the performance of the classifier. Obtaining this value is done by measuring the length of the column.
- *The number of unique values in the column*: In general, nominal data tends to vary more in cardinality as opposed to ordinal data. Furthermore, some ordinal data columns tend to adhere to Likert-scale characteristics regarding the possible number of unique entries, which also limits its cardinality. The value is obtained by counting all unique entries in a column.
- *The ratio between the number of unique values and the total number of rows*: As a rule of thumb, some domain experts tend to classify data as ordinal when the ratio between the unique values and the total number of rows is at most 0.05. The ratio for nominal data tends to be at most 0.2. As a result of this rule of thumb, we extract the ratio for use in the classifier by dividing the number of unique values by the total number of rows.
- *The mean of the variance of the distance between the word embeddings of unique entries*: The idea behind extracting this feature is that the word embeddings of certain entries showcase interesting linear substructures in the word vector space. By taking a pre-trained word vector space, the classifier may be able to make a distinction between ordered and unordered data based on differences in the substructure. The first step is to split each entry into a set of words which are then embedded using a pre-trained word vector space. This work makes use of the Wikipedia word vector space by GloVe, which consists of over 400 000 words in the vocabulary embedded into a 50-dimensional vector space [25], to assign each word in an entry to a vector. A random point in the vector space will be assigned to a word in case it does not appear in the pre-trained corpus. Next, the mean of all dimensions for each word vector in the entry is calculated such that all word vectors of

the entry are now represented as a single point in the 50-dimensional vector space. After this is done for all entries in the column, the variance between each dimension is calculated. Finally, the mean of each dimension is taken and the resulting value is a single float value that can be used to potentially distinguish ordered data from unordered data.

- *Whether the column name is commonly used in ordinal data*: There are a set of keywords that can commonly be found in column names for ordinal data. Examples of certain keywords include `grade`, `stage`, and `opinion`. By checking whether a column name is contained within one of those keywords and vice versa, we are able to tell whether the data in the column is more likely to be ordinal or not. Note that keywords for column names used in this work are created based on domain expertise, which means that results may vary when other keywords are used.
- *Whether the column name is commonly used in nominal data*: The approach of extracting this feature is similar to that of checking ordinal traits in the column name, except that the names obtained via domain expertise are now commonly used in nominal data. Typical nominal column names include `address`, `city`, `name`, and `type`. Again, since the used names are based on domain expertise, results in performance may vary when other names are used.
- *Whether the unique entries contain keywords that are commonly found in ordinal data*: Extracting this feature is similar to the two aforementioned techniques, except that ordinality will now be implied based on keywords that are commonly found in ordinal data. The keywords that were used in this work are adjectives and nouns that are typically found in Likert-scale questionnaires.
- *Whether the unique entries share a number of common substrings*: Ordinal data tends to contain entries with overlapping substrings. For example, the strings `disagree`, `agree`, and `wholeheartedly agree` all contain the substring `agree`. By checking whether there are common substrings in the data of sufficient length, it is possible that the classifier associates the occurrence of substrings with the implication that the data is ordinal.

C Datasets

A list of all datasets that were used during the evaluation are provided here, including additional information on what was used from the data.

C.1 String feature inference and processing

Some of the datasets listed below were used to evaluate both string feature inference and string feature processing. The datasets without model, task, and target were only used to evaluate string feature inference based on the manually assigned ground truth of the columns. The datasets with the previously mentioned components were used for both string feature inference (with manual labeling of string feature) and string feature processing (evaluation by running the associated task).

Coordinate

- **Digital altimetric data information - GPS**. Columns: *latgms, loggms*. `https://www.kaggle.com/mpwolke/cusersmarildownloadsgpscsv`

Day

- **San Francisco Crime Classification**. Columns: *DayOfWeek*. Model: *GradientBoostingClassifier*. Task: *Classification*. Target: *Category*. `https://www.kaggle.com/kaggle/san-francisco-crime-classification`

E-mail

- **Data_UK**. Columns: *email*. `https://www.kaggle.com/phool1804/data-uk`
- **Enrico's Email Flows**. Columns: *sender, receiver*. `https://www.kaggle.com/emarock/enricos-email-flows`
- **Indian Companies Registration Data [1857 - 2020]**. Columns: *EMAIL_ADDR*. `https://www.kaggle.com/rowhitswami/all-indian-companies-registration-data-1900-2019`

Filepath

- **Collection of Classification & Regression Datasets**. Columns: *Image Index*. `https://www.kaggle.com/balakrishcodes/others?select=xrayfull.csv`
- **Hillary Clinton's Emails**. Columns: *MetadataPdfLink*. `https://www.kaggle.com/kaggle/hillary-clinton-emails?select=Emails.csv`
- **Liver and Liver Tumor Segmentation**. Columns: *filepath, liver_maskpath, tumor_maskpath*. `https://www.kaggle.com/andrewmvd/lits-png?select=lits\_df.csv`

Month

- **FIFA 19 complete player dataset**. Model: *GradientBoostingRegressor*. Task: *Regression*. Target: *Value*. Columns: *Joined*. https://www.kaggle.com/karangadiya/fifa19
- **Netflix Movies and TV Shows**. Columns: *date_added*. https://www.kaggle.com/shivamb/netflix-shows
- **Metacritic-Game Releases by Score**. Columns: *Date*. https://www.kaggle.com/abhishekdataset/metacriticgame-releases-by-score

Numerical

- **FIFA 19 complete player dataset**. Columns: *LS, ST, RS, LW*. Model: *GradientBoostingRegressor*. Task: *Regression*. Target: *Value*. https://www.kaggle.com/karangadiya/fifa19
- **HR Analytics: Job Change of Data Scientists**. Columns: *company_size*. https://www.kaggle.com/arashnic/hr-analytics-job-change-of-data-scientists?select=aug_train.csv
- **Students' Academic Performance Dataset**. Columns: *StudentAbsenceDays*. https://www.kaggle.com/aljarah/xAPI-Edu-Data

Sentence

- **Wine Reviews**. Columns: *description*. Model: *GradientBoostingRegressor*. Task: *Regression*. Target: *points* https://www.kaggle.com/zynicide/wine-reviews
- **World Development Indicators**. Columns: *SpecialNotes, SystemOfNationalAccounts*. https://www.kaggle.com/worldbank/world-development-indicators

URL

- **FIFA 19 complete player dataset**. Columns: *Photo, Flag, Club Logo*. Model: *GradientBoostingRegressor*. Task: *Regression*. Target: *Value*. https://www.kaggle.com/karangadiya/fifa19
- **Walmart Product Details 2020**. Columns: *Product Url*. https://www.kaggle.com/promptcloud/walmart-product-details-2020

Zip code

- **Data_UK**. Columns: *postal*. https://www.kaggle.com/phool1804/data-uk
- **OpenAddresses - Europe**. Columns: *POSTCODE*. https://www.kaggle.com/openaddresses/openaddresses-europe?select=netherlands.csv

- **OpenAddresses - North America (excluding U.S.)**. Columns: *POSTCODE*. https://www.kaggle.com/openaddresses/openaddresses-north-america-excluding-us?select=bermuda.csv
- **House Price Data, England & Wales, 2015 to 2019**. Columns: *SS2 6ST*. Model: *GradientBoostingRegressor*. Task: *Regression*. Target: *249995*. https://www.kaggle.com/dmaso01dsta/house-price-data-england-wales-2015-to-2019

C.2 Statistical type prediction and determining order in data

Some of the datasets listed below were used for both statistical type prediction and determining the order in ordinal data. The datasets without model, task, and target were only used to classify the statistical type based on the manually labeled ground truth of the columns. The datasets with the previously mentioned components were used for both statistical type prediction (with manual labeling of ordinality) and performance evaluation of FlairNLP (evaluation by running the associated task).

Nominal datasets

- **[NeurIPS 2020] Data Science for COVID-19 (DS4C)**. Columns: *province, city*. https://www.kaggle.com/kimjihoo/coronavirusdataset?select=Case.csv
- **[NeurIPS 2020] Data Science for COVID-19 (DS4C)**. Columns: *type, gov_policy*. https://www.kaggle.com/kimjihoo/coronavirusdataset?select=Policy.csv
- **AB_NYC_2019**. Columns: *name, host_name, neighborhood_group, neighborhood, room_type*. https://www.kaggle.com/chadra/ab-nyc-2019
- **Automobile Dataset**. Columns: *make*. https://www.kaggle.com/toramky/automobile-dataset
- **Craft Beers Dataset**. Columns: *style*. https://www.kaggle.com/nickhould/craft-cans?select=beers.csv
- **Craft Beers Dataset**. Columns: *city, state*. https://www.kaggle.com/nickhould/craft-cans?select=breweries.csv
- **FIFA 19 complete player dataset**. Columns: *Nationality, Club*. https://www.kaggle.com/karangadiya/fifa19
- **FiveThirtyEight Comic Characters Dataset**. Columns: *ALIGN, EYE, HAIR*. https://www.kaggle.com/fivethirtyeight/fivethirtyeight-comic-characters-dataset?select=dc-wikia-data.csv
- **HR Analytics: Job Change of Data Scientists**. Columns: *city, major_discipline, company_type*. https://www.kaggle.com/arashnic/hr-analytics-job-change-of-data-scientists?select=aug_train.csv

- **IBM HR Analytics Employee Attrition & Performance**. Columns: *Department, EducationField, JobRole*. Model: *GradientBoostingRegressor*. Task: *Regression*. Target: *MonthlyIncome*.
  `https://www.kaggle.com/pavansubhasht/ibm-hr-analytics-attrition-dataset`
- **Kickstarter Projects**. Columns: *category, main_category, currency, country*.
  `https://www.kaggle.com/kemical/kickstarter-projects?select=ks-projects-201612.csv`
- **Mushroom Classification**. Columns: *class, cap-shape, cap-surface, cap-color, bruises, odor, gill-attachment, gill-spacing, gill-size, gill-color, stalk-shape, stalk-root, stalk-surface-above-ring, stalk-surface-below-ring, stalk-color-above-ring, stalk-color-below-ring, veil-type, veil-color, ring-numer, ring-type, spore-print-color, population, habitat*.
  `https://www.kaggle.com/uciml/mushroom-classification`
- **Pokemon with stats**. Columns: *Type 1, Type 2*.
  `https://www.kaggle.com/abcsds/pokemon`
- **Ramen Ratings**. Columns: *Brand, Variety, Style, Country*.
  `https://www.kaggle.com/residentmario/ramen-ratings`
- **Stroke Prediction Dataset**. Columns: *work_type, smoking_status*.
  `https://www.kaggle.com/fedesoriano/stroke-prediction-dataset`
- **Students' Academic Performance Dataset**. Columns: *PlaceOfBirth, GradeID, SectionID, Topic*.
  `https://www.kaggle.com/aljarah/xAPI-Edu-Data`
- **Students Performance in Exams**. Columns: *race/ethnicity*.
  `https://www.kaggle.com/spscientist/students-performance-in-exams`
- **Wine Reviews**. Columns: *country, province, region_1, variety*. Model: *GradientBoostingRegressor*. Task: *Regression*. Target: *points*.
  `https://www.kaggle.com/zynicide/wine-reviews`

Ordinal datasets

- **Amazon - Ratings (Beauty Products)**. Columns: *Rating*.
  `https://www.kaggle.com/skillsmuggler/amazon-ratings?select=ratings\_Beauty.csv`
- **Audiology (Original) Data Set**. Columns: *air, ar_c, ar_u, bone, o_ar_c, o_ar_u, speech*.
  `https://archive.ics.uci.edu/ml/datasets/Audiology+\%28Original\%29`
- **Basic Income Survey - 2016 European Dataset**. Columns: *dem_education_level, awareness, vote, age_group*.
  `https://www.kaggle.com/daliaresearch/basic-income-survey-european-dataset`
- **Car Evaluation Data Set**. Columns: *buying, maint, doors, persons, lug_boot, safety, class value*. Model: *GradientBoostingClassifier*. Task: *Classification*.

Target: *Class Values*.
https://archive.ics.uci.edu/ml/datasets/Car+Evaluation

- **Earthquake Magnitude, Damage and Impact**. Columns: *damage_overall_colapse, damage_overall_leaning, damage_grade, technical_solution_proposed*.
https://www.kaggle.com/arashnic/earthquake-magnitude-damage-and-impact?select=csv_building_damage_assessment.csv
- **Earthquake Magnitude, Damage and Impact**. Columns: *education_level_household_head*.
https://www.kaggle.com/arashnic/earthquake-magnitude-damage-and-impact?select=csv_household_demographics.csv
- **Hayes-Roth Data Set**. Columns: *age, educational level, marital status*.
https://archive.ics.uci.edu/ml/datasets/Hayes-Roth
- **Linux Gamers Survey, Q1 2016**. Columns: *LinuxUserHowLong, DesktopLinuxGamerHowLong, HeavyGamer, LinuxExclusivity, LinuxGamingHabitChange, LinuxGamingHabitFuture, LinuxGamingMachineShared, FolksAroundYouAwareLinux, LinuxGamesPurchaseFrequency, SatisfactionSteam, SatisfactionGOG, SatisfactionHB, DistroChangeFrequency, DistroImpactPerformance, HardwareUpgradeIntent, AwarenessBrandedSteamMachines, AwarenessSteamController, AwarenessSteamLink, SteamMachineConceptLike, SteamMachinesExpandLinuxDoubtful, SteamMachinesLaunchEvaluation, SteamMachinesAwarenessAlienware, SteamMachinesAwarenessZotac, SteamMachinesAwarenessSyber, SteamMachinesWantToBuy, MachinesMaximumPrice, MachinesDIYIntent, SteamControllerPurchaseIntent, SteamOSEverTried, SteamIHSUage, SteamLinkPurchaseIntent, WINEUsageVanilla, PlayOnLinux, Crossover, WINEEvaluation*.
https://www.kaggle.com/sanqualis/linuxgamerssurvey
- **Nursery Data Set**. Columns: *parents, has_nurs, form, housing, finance, social, health*. Model: *GradientBoostingClassifier*. Task: *Classification*. Target: *Nursery*.
https://archive.ics.uci.edu/ml/datasets/Nursery
- **Solar Flare Data Set**. Columns: *activity, evolution, previous_24h_flare_activity_code, area*. Model: *GradientBoostingRegressor*. Task: *Regression*. Target: *C-class, M-class, X-class*.
https://archive.ics.uci.edu/ml/datasets/Solar+Flare
- **Soybean (Large) Data Set**. Columns: *precip, temp, crop-hist, area-damaged, severity, stem-cankers*. Model: *GradientBoostingClassifier*. Task: *Classification*. Target: *class*.
https://archive.ics.uci.edu/ml/datasets/Soybean+\%28Large\%29