

**City, University of London in MSc Data Science**

**PROJECT REPORT**

**2022**

**TITLE OF THE THESIS**

Zacharias Detorakis ([zacharias.detorakis@city.ac.uk](mailto:zacharias.detorakis@city.ac.uk))

Supervised by: Ernesto Jiménez-Ruiz ([ernesto.jimenez-ruiz@city.ac.uk](mailto:ernesto.jimenez-ruiz@city.ac.uk))

*By submitting this work, I declare that this work is entirely my own except those parts duly identified and referenced in my submission. It complies with any specified word limits and the requirements and regulations detailed in the assessment instructions and any other relevant programme and module documentation. In submitting this work, I acknowledge that I have read and understood the regulations and code regarding academic misconduct, including that relating to plagiarism, as specified in the Programme Handbook. I also acknowledge that this work will be subject to a variety of checks for academic misconduct.*

*Signed:*

To my mum who’s no longer able to follow my progress in life but whose values will always be a shinning beacon for me to follow…

**Abstract**

This project aims to enhance a set of data given as input (e.g. tabular data) with semantic meaning using existing Knowledge Graphs (KG) as reference. The approach that has been implemented is inspired by ColNet, a system implemented as part of the SemTab challenge, and comprises a series of modules for i) parsing the tabular data ii) using the (cell) values to identify candidate entities with lexical similarity and their candidate KG classes iii) training a set of binary cnn classifiers (one per class) and then employing different ways of predicting the class of each column using the above. A pretrained word2vec model was used to train the cnns and transform the column values to inputs for predication.

WILL WE DO A CEA?

The results indicate that…

**Keywords:** Knowledge Graphs, Convolutional Neural Networks, Class, Entity, Word2Vec

Table of Contents

[1 Introduction and Objectives 4](#_Toc87295654)

[1.1 Background of the problem 4](#_Toc87295655)

[1.2 Reasons of the choice of the project and beneficiaries 4](#_Toc87295656)

[1.3 Objectives of the project 5](#_Toc87295657)

[1.4 Methods used 5](#_Toc87295658)

[1.5 Work plan 6](#_Toc87295659)

[1.6 Major changes of the goals or methods that happened during the project 6](#_Toc87295660)

[1.7 Report outline 6](#_Toc87295661)

[2 Context 7](#_Toc87295662)

[3 Methods 7](#_Toc87295663)

[4 Results 7](#_Toc87295664)

[5 Discussion 7](#_Toc87295665)

[6 Evaluation, Reflections, and Conclusions 7](#_Toc87295666)

[Glossary 8](#_Toc87295667)

[References 8](#_Toc87295668)

[Appendix 8](#_Toc87295669)

# Introduction and Objectives

## Background of the problem

One of the main problems of computer science is the ability to represent human knowledge and model the world in a form that can be understood and processed by computers. Once this problem is resolved, models could them be subsequently trained to perform complex reasoning and draw conclusions and new knowledge in an intelligent way that resembles human intelligence.

The World Wide Web Consortium (W3C), has developed two languages ​​of knowledge representation for the Internet: RDF(S) and OWL. These languages form the syntax of knowledge bases KB or Knowledge Graphs (KG). Simply put, a knowledge graph provides a structured representation of information using a directed vector of the form (subject, predicate, object) according to the Resource Description Framework (RDF). Each node in that triple (i.e. subject and object) represent an entity belonging to a class and the edge between them (i.e. the predicate) represents the relationship between the classes the edges belong to. Several knowledge graphs have been developed by domain experts that have transferred their specialized knowledge into these ontologies and knowledge engineers, that have translated this knowledge into a standard language such as OWL.

In most cases, however, for reasons of convenience, simpler forms of knowledge representation are chosen, such as plain text in natural language (unstructured information). Such cases of unstructured or semi-structured information are text files, txt files, excel files, html files posted on the internet, information entered as text in database systems, etc. In this (unstructured) form, knowledge is not comprehensible to computers and cannot be used in its entirety to draw useful conclusions.

In order to solve this problem, the scientific community has turned its attention to areas such as natural language processing and the automatic extraction of terms. With the tools built into these areas, we try to automatically extract knowledge from unstructured language descriptions and use it to map to existing ontologies and even expand them, creating new specialized classes or entities. The exported knowledge can then be used to classify objects of our world into categories (classes) of these well-established knowledge bases.

## Reasons of the choice of the project and beneficiaries

The main contribution of this project it to create an end-to-end pipeline with that assist with type identification of tabular data. By creating individual modules for parsing tabular data, identifying candidate entities from knowledge graphs and having multiple mechanisms of predicting types this system can be used as an extensible framework to swap in/out different types of input data, knowledge bases and classification models.

The main components created as part of this project is assuming tabular data in a csv format, using dbpedia KB as the reference for identifying entities and types and assumes a simple voting algorithm of the identified objects as well as a more sophisticated set of CNN classifiers (ColNet) for type identification.

Such systems can be used in information retrieval from large unstructured data that could then inform the analysis on information maintained by online applications or other systems that generate data in the internet of things. Although the KG often suffer from knowledge gaps (i.e. not all entities of a given class exist as instances of that class) suggesting even a type for a given column could also be a useful first step in analysis to limit the universe of the exploratory data analysis EDA an analyst may need to do.

## Objectives of the project

A set of objectives were set out, in order to answer the research question on how to enhance a set of data given as input (e.g. tabular data) with semantic meaning (e.g. class / entity identification) using existing knowledge graphs (e.g. DBpedia, WikiData) as reference:

* Replicate the code given by ColNet and use that as a baseline, train individual ccn classifiers for each candidate class and use them to predict the type of the target columns
* Examine the effect of using the KG hierarchy “rdfs:subClassOf” to filter down candidate classes
* Test the effect of different hyperparameters for training the CNN models for class prediction
* Combine the prediction of the CNNs and candidate class based on majority vote
* CEA

## Methods used

As mentioned in previous sections, the end 2 end pipeline comprises multiple modules with catch points in between in order to be able to plug different implementations in and out. Figure 1 illustrates high level the steps of the proposed method.

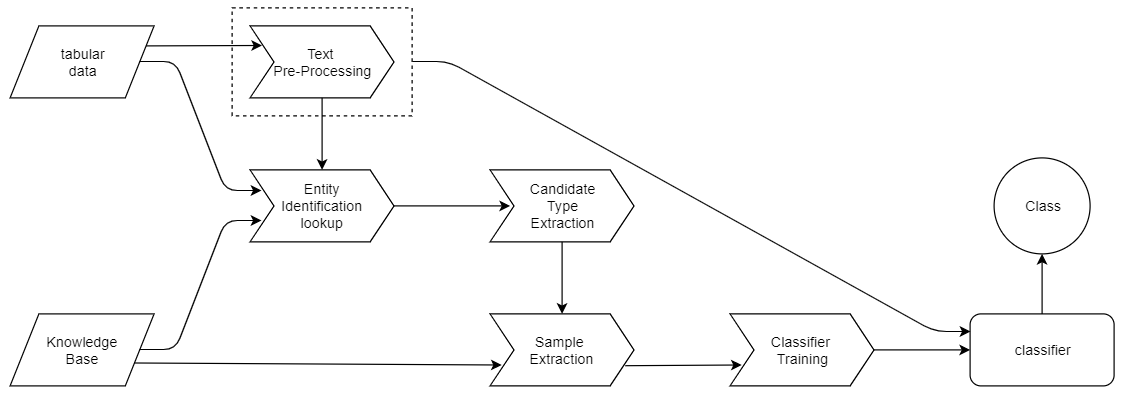


Figure 1. Process of predicting classes for tabular data from a reference KG

**Parsing:** Reading tabular data and storing them into a json object that can then be used by the next step

**Lookup (entity + class):** The lookup step is the module that queries the reference knowledge base with the cell values to retrieve candidate entities. For this component the knowledge base used in dbpedia and there are two types of lookups:

* The entity lookup using the dbpedia lookup endpoint and the
* SPARQL endpoint that retrieved the classes of the identified entities in case the lookup URL failed to retrieve them

The lookup URL gives the ability to limit the number of retrieved results, since it performs fuzzy matching based on the input string (i.e. cell value). The limit was kept quite flexible to allow for more results to be retrieved however further downstream we imposed stricter limits to narrow down the number of classifiers that are assesses as candidates for each column

Given that the dbpedia lookup URL is doing a fuzzy matching on the query string, the ‘text pre-processing’ step in Figure 1 was skipped as part of this iteration.

**Sample extraction:** this step is used in order to get training data for the candidate rdf:type classifiers. This is a similar step to the above when we extracted the class from the entity but here he follow the reverse path. We extract, at random, entities from a given class.

**Classifier Training:** in this step we create a binary cnn model for every candidate type (more details on this in Chapter 2) and we train those models with positive and negative samples. The positive samples are taken from the sample extraction of the previous step whereas the negative samples are taken from candidate entities that DO NOT belong to the current class the model is trained for.

**Predict Class:** finally, in this last step we bring everything together to predict a type for ever target column in the input tabular data. The are a couple of ways to predict the type of a column:

* By simply selecting the type with the most votes based on the simple lookup or
* By considering the outcome of the binary classifiers for each candidate class of the given column in the input data

## Work plan

To achieve the solution of the components listed above, the work was divided in individual milestones dedicated to the predefined modules.

After completing the initial literature review at the early stages of the project to better understand the background and related work the next task was to design the text processing feature and integration with the reference knowledge base. As part of that design, several json structures were decided to save the data for subsequent runs and avoid repeating the lookup step unless we got new input data or reference knowledge base. The catch-up jsons files expedited tests as the lookup step takes a lot of time to complete and always yields the same results for the same (input, knowledge base) combination. Implementation of this step followed shortly after using Python (jupyter notebooks for debugging at this stage)

Getting candidate classes and entities at this point enabled us to get the first type identification experiments in place using just a majority vote of candidate classes

The next major step following on from this was the design and implementation of the binary cnn classifiers. This was yet again another computational and time expensive step so we designed a solution that can save and load models that can be trained offline for any number of classes from the reference knowledge base.

With the classifiers implemented the last design step for the initial end-to-end pipeline was to come up with a way of using the prediction results to finally decide on a column class and perform experiment to identify which hyperparameter combinations would work best in the given project setup.

In parallel to the above, we started completing the relevant sections of the report as soon as e.g. context or intermediate experimental results because available for reporting and discussion. Thereby the report was prepared alongside the design and implementation to capture the details as they were worked on.

CEA\_CTA????

## Major changes of the goals or methods that happened during the project

TBC

## Report outline

This first chapter of the report provides a brief background of the area that the project is placed in and an introduction of the problem that the project is trying to resolve. The inspiration for this project is also stated along with a proposed methodology a high-level steps of the proposed implementation. Moreover, a set of objectives are set in terms of what the project is aiming to achieve along with a plan on how these objectives will be achieved within the timeframe that has been allocated to this exercise.

In the following chapter (Chapter 2) we follow on from the introduction to provide a more deep dive view of the theorical background the project was built up on, including detailed literature review. The two main sections in that chapter revolve around the theory of knowledge graphs and the convolutional neural networks.

After covering the theory supporting this analysis, the next chapter (Chapter 3) focuses on the methods that were applied during the implementation of this research project. Details about each of the end-to-end steps mentioned in paragraph 1.4 as well as key decisions taken for the execution of several experiments will be discussed in detail. Therefore, this chapter will set the framework on which the remaining chapters of this report will expand with presenting the results.

Following on from Chapter 3, the next chapter (Chapter 4) presents the experimental results of the implemented methods for the cases that were designed. These experimental results will highlight strengths and weaknesses on the approach and will inform next step for future analysis.

In Chapter 5 we will pick up the critical discussion around the results of the previous chapter to analyse further the efficiency of the chosen models and get potential ideas for future improvements.

Finally, Chapter 6 presents a platform for Reflection, and at the same time concludes this project while offering a general evaluation of its results. Any potential for future work will also be presented in this section.

(2,200\_2,650)

# Context

21.5(2,600-3,200\_4,700-5,420)

# Methods

## Input Data

The scope of the project is to automatically predict the types of columns in tabular data. Therefore, by definition we needed to have two sets of inputs:

* The tabular data to analyse and predict the classes of
* A reference knowledge base containing candidate types they columns would be assigned to

### Tabular Data

The tabular data could really be in any format however for the purposes of this project the input used was from the SemTab Challenge (TBD) who provided the data in csv files. Each csv file has a header row with tiles, and several rows of data. On top of the tabular data that form the input dataset SemTab also provide two more datasets that are optional for the end-to-end pipeline:

* A file containing the column indexes from each file that need to be considered for type prediction and
* A file with the ground truth (i.e. the actual classes from the reference knowledge base corresponding to each of the columns)

Even though these additional files are optional for the end-to-end the pipeline uses them in order to filter down the columns that need to be considered for the prediction and also uses the ground truth to evaluate the prediction accuracy.

### Reference Knowledge Base

The reference knowledge base really depends on the input tabular data. However, for the experiments that were run as part of this project DBpedia was considered as input.

### Word2Vec

In order to provide vectors of words as input to a neural network we need to convert each work to a numerical vector in a way the relevant words are closer in space than less relevant words. For the purposes of this project a pre-trained word2vec model that is readily available was used to convert strings to numerical vectors.

## Data Processing

The next step is to load the data from the inputs mentioned above in a structure that can then be used further down in the lookup, training and predictions steps. Throughout this project json was used as a flexible structure for the data so that they can been easily accessible and also dictionary is a mutable structure so updating the dictionary could be done at different stages.

The format of the dictionary called ‘data’ is illustrated below. As shown in the example each csv file is a separate object in the dictionary with the following attributes that are also dictionaries:

* column\_titles: This attribute holds the titles of the columns as specified in the incoming csv. This is an optional feature that could be deactivated in case the incoming data do not have any titles. For the majority of the analysis and testing done on this project, column titles are indeed ignored
* data: This is the key attribute and has the cell values for each of the columns in the tabular data. The processing of loading data takes into account the target columns mentioned in 3.1.1 and ignore any other column. This feature can also be disabled in case the target columns are not available. Additionally, when loading the data, the function provides the option of storing every cell value in a column or only the unique values that appear in that specific column. Experiments have shown that keeping multiple instances of the same value so not improve the results therefore unless specifically mentioned in the experiment, the default for the function is to retrieve the unique cell values

Due to the above statement the cell values across the columns are not aligned. i.e. the nth value in the array for column 1 in the below example doesn’t correspond to the nth value in column 3.

* gt: Finally, this attribute has the expected type for each of the columns. This attribute is only used at the final stage of the process, after a type as been predicted, to assess the accuracy of each model that is being tested. Once more if the ground truth, referenced in 3.1.1, is not available, the attribute can be eliminated from the dictionary without any impact in the downstream pipeline of predicting the column types

{

    "58891288\_0\_1117541047012405958": {

        "column\_titles": {

            "1": "Title",  
 "3": "Director(s)"  
 },

        "data": {

            "1": [

                "Gone with the Wind",

                "The Shawshank Redemption",

                "The Battleship Potemkin",

                ...,

                "Gladiator"

            ],

            "3": [

                "Mel Gibson",

                "Orson Welles",

                "Francois Truffaut",

                ...,

                "Woody Allen"

            ]

        },

        "gt": {

            "1": "Film",

            "3": "Person"

        }

    },

    "8468806\_0\_4382447409703007384": {  
 ...

}

}

## Entity Lookup

With the data loaded in the data dictionary the next step is to lookup the cell values in the DBpedia endpoint and get the candidate classes and entities. Lookups against other reference knowledge graphs haven’t been considered as part of this project as the target types were all from DBpedia, however the module can be replaced as long as the output results are still logged in the structure that will be described later in this section

The DBpedia lookup comprises two steps. The second being an optional one:

### Step 1: Cell Value Lookup DBpedia API

For this first step the function is making a call to the DBpedia lookup API.

http://lookup.dbpedia.org/api/search/KeywordSearch?**MaxHits**=5&**QueryString**=Cell Value

The API provides two keys for the request as follows:

**QueryString:** This is a mandatory field that contains the keyword that needs to be queried. The API does a fuzzy matching between the keyword and the dbpedia entity labels, so the retrieved results are not always exact matches on the label. This is the reason that rendered the value processing step originally provisioned in the pipeline (e.g. stemming, lemmatizing) unnecessary. The only pre-processing of the cell value that was performed is the removal of characters (‘[’ and ‘]’) that appeared in some cell values and rendered the API request invalid

**MaxHits:** This is an optional attribute that enables the user to limit the number of the returned results. As mentioned above the results are retrieved based on a fuzzy matching with the most relevant appearing at the top and less relevant near the bottom. For the purposes of this analysis the process flexible to store the 5 top lookup results for each cell value since that enables more candidate classes to be accessed for each column.

The response from the API is an xml that is being parsed by the lookup function to retain the URI of the retrieved entity, and the URI(s) of the associated class(es). We also maintained the rank (i.e. place in the top 5) the result came in

### Step 2: Retrieve Entity Type(s)

The next step was added later on in the process when the analysis of the lookup results illustrated gaps in the results of the API. There are many instances where, for whatever reason, the lookup API response fails to retrieve the classes of the identified entity. In the example below the third result of the request for the cell value ‘A Streetcar Named Desire’

(i.e.<https://lookup.dbpedia.org/api/search/KeywordSearch?MaxHits=5&QueryString=A%20Streetcar%20Named%20Desire> ) is URI ‘https://dbpedia.org/page/A\_Streetcar\_Named\_Desire\_(1951\_film)’ that is retrieved without any associated classes (i.e. <Classes/>).

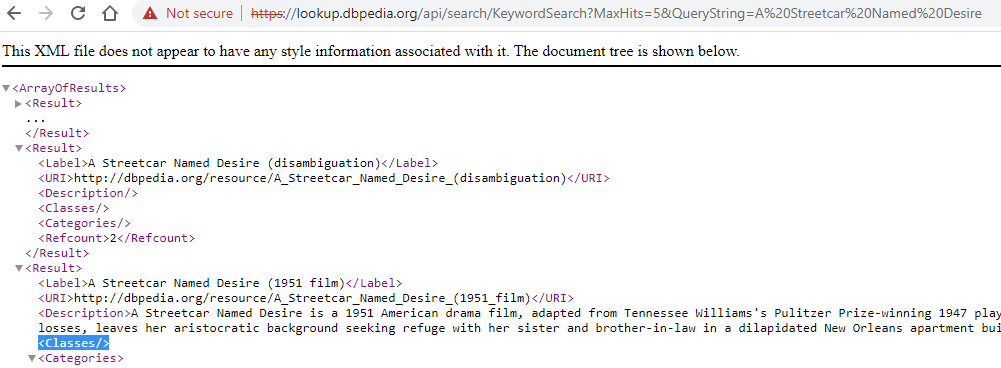


Figure 2. Results from the dbpedia lookup api when querying the cell value ‘A Streetcar Named Desire’

However, when visiting the actual URI of the retrieved entity it appears that the entity has indeed associated rdf:types [Figure 3]. To overcome this issue, the process performs a second lookup for those entities that came back without a type. This time it using the dbpedia sparql endpoint to sent a request of the specific entity URI and retrieve any rdf:types in the dbo namespace. In Figure 3 the two relevant types have been highlighted with a black frame.

Finally, if the retrieved entity doesn’t have any associated type (e.g. the second result in the lookup response http://dbpedia.org/resource/A\_Streetcar\_Named\_Desire\_(disambiguation)) then the entity isn’t considered at all in the structure the process is creating

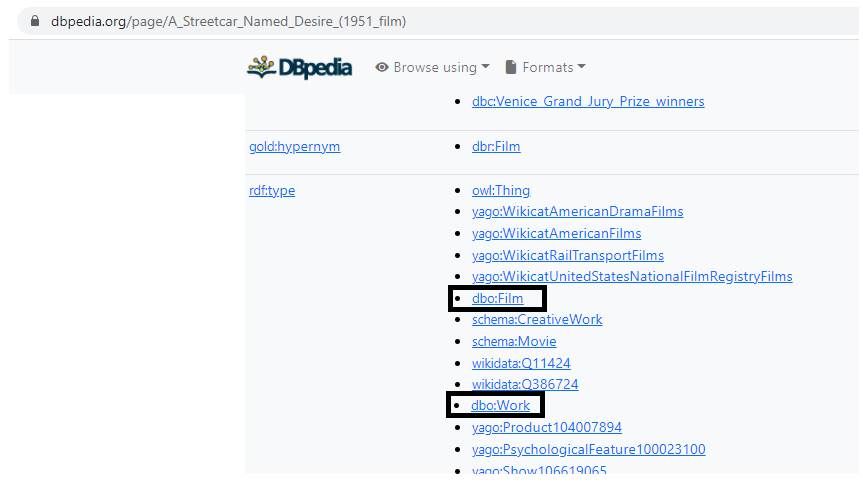


Figure 3. Types associated to the entity ‘https://dbpedia.org/page/A\_Streetcar\_Named\_Desire\_(1951\_film)’

Each cell value is only looked up once when it appears for the first time in the ‘data’ json, however the process still keeps track of any additional column the same cell value might have appeared in, as well as all candidate entities and classes it may have matched to.

Once again, the results of this lookup are maintained in a json object so that the lookup process need only be run once (even offline) for every new dataset

The outcome of the lookup is stored in the cell\_values dictionary as follows. The example below shows the results for the cell value for ‘A Streetcar Named Desire’ as a continuation of the previous example

"A Streetcar Named Desire": {

        "location": [

            [

                "58891288\_0\_1117541047012405958",

                1

            ],

            [

                "20135078\_0\_7570343137119682530",

                1

            ],

            [

                "35188621\_0\_6058553107571275232",

                1

            ]

        ],

        "candidate\_entities": {

            "A\_Streetcar\_Named\_Desire": {

                "rank": 1,

                "candidate\_classes": [

                    "Play",

                    "WrittenWork",

                    "Work"

                ]

            },

            "A\_Streetcar\_Named\_Desire\_(1951\_film)": {

                "rank": 3,

                "candidate\_classes": [

                    "Film",

                    "Work"

                ]

            },

            "The\_Originals\_(season\_3)": {

                "rank": 4,

                "candidate\_classes": [

                    "TelevisionSeason",

                    "Work"

                ]

            },

            "A\_Streetcar\_Named\_Desire\_(opera)": {

                "rank": 5,

                "candidate\_classes": [

                    "TelevisionShow"

                ]

            }

        }

In this json structure each cell value is a dictionary with the following keys:

* location: This is an array of tuples where each tuple contains a (file, column index) the value appeared in. As mentioned earlier each value may appear only once in a given column
* candidate\_entities: this is a dictionary where each retrieved entity (up to 5) is a separate key. Within the entity keys the structure provisions for:
  + rank: the order in which the entity appeared in the results from the lookup API
  + candidate\_classes: an array of all the dbo types associated to the specific entity

Finally on to of the data and cell\_values structure mentioned in 3.2 and 3.3 (above) respectively there are two more structure that reshape that data in order to be used for the next steps of the pipeline. The first is a structure that is used to predict a class by voting whereby each input csv is represented as a key with the following attributes:

## Predict by Voting

Having come up with a list of candidate entities and types of each cell value of a given column there pipeline the proceeds to a very basic prediction of the type by considering the types that are proposed by the retrieved candidate entities. There are a number of experiments that will be presented in Section 4 in terms of how the voting works. Basic permutations take into account the rank of an entity as well as the entity hierarchies within a knowledge base (i.e. parent child relationship by the rdf:subClassOf predicate)

## Training the CNN

For every cell value in the column, we create a synthetic column of size x (parameterised as ‘synthetic\_column\_size’). This synthetic column contains the cell value itself plus an additional x-1 selected cell values from the same column.

There are two approaches we’ve followed:

**Random selection:** In this case, the x-1 cell values where randomly selected. In case the column length is smaller than the number of samples we need to select (i.e. < x-1) then all cell values were selected and the remaining positions in the synthetic column are populated with nulls.

i.e.

cell\_value = ‘Joseph L. Mankiewicz’

synthetic column = ['Joseph L. Mankiewicz', 'James Whale', 'Frank Darabont', 'Sam Mendes', 'John Huston', 'Mike Nichols', 'John Frankenheimer', 'Charles Chaplin', 'Harold Ramis', 'Federico Fellini']

**Sliding Window:** **:** In this case, a window of size x starting from the current cell value is used to generate the sample. For the next sample, the window slides by one position and so on and so forth. This technique will generate a few less samples i.e. column\_size – x + 1 as opposed to column\_size samples in the random selection. In case column\_size – x + 1 only one sample will be created which will have all values in the column and nulls for the remaining positions needed to complete the correct sample size x.

ILLUSTRATION

The next step is to convert the list of cell values in the synthetic column above to a list of words. We do that by removing special characters usually used to separate words (e.g. '\_', '-', '.', '/', '"', "'") and replacing them with spaces. Finally, we tokenise the derived string using the space (i.e. ‘ ’) as the delimiter. The size of this sequence is typically longer than the size of the synthetic column in order to allow for cell values comprising more than one words. Any words produced by the tokenizer that cannot fit the length of the sequence are dropped. For instance, in the above example where synthetic\_column\_size = 10 if sequence\_size = 20 then the word ‘felini’ will be dropped and the sequence for the specific synthetic columns will be

synthetic\_column\_sequence\_20 = ['joseph', 'l', 'mankiewicz', 'james', 'whale', 'frank', 'darabont', 'sam', 'mendes', 'john', 'huston', 'mike', 'nichols', 'john', 'frankenheimer', 'charles', 'chaplin', 'harold', 'ramis', 'federico']

On the flipside if the produced words from the tokeniser are less than the length of the sequence then the remaining positions are once again filled up with nulls. For instance, in the same example, if sequence\_size = 30 the produced sequence will be as follows

synthetic\_column\_sequence\_30 = ['joseph', 'l', 'mankiewicz', 'james', 'whale', 'frank', 'darabont', 'sam', 'mendes', 'john', 'huston', 'mike', 'nichols', 'john', 'frankenheimer', 'charles', 'chaplin', 'harold', 'ramis', 'federico', 'fellini', 'NaN', 'NaN', 'NaN', 'NaN', 'NaN', 'NaN', 'NaN', 'NaN', 'NaN']

There needs to be a balance as to how long the sequence should be in relation to the synthetic column size. If the ratio is too low, then we may end up losing many words from the cell values but if too big we will have longer sequences to process with many nulls that are not adding any value to the classification.

26.5(3,200\_4,000\_6250)

# Results

15.7%(1,900-2,350)

# Discussion

9.5%(1,150-1,400)

# Evaluation, Reflections, and Conclusions

10%(1,200-1,500)

# Glossary

# References

# Appendix

|  |  |  |
| --- | --- | --- |
| **name** | **address** | **Restaurant\_name** |
|  |  |  |
|  |  |  |
|  |  |  |

Table 1. Examples or restaurant names reused for different restaurants