

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

**PROJECT REPORT**

**2021**

**Embedded Driven Semantic Table Understanding**

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 see 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 then 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 of the edges. 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 (i.e. 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 insights.

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 can assist with type and entity identification of tabular data. By creating individual modules for parsing tabular data, identifying candidate entities from knowledge graphs and having multiple mechanisms of predicting column 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 assume input tabular data in a csv format and use dbpedia KB as the reference for identifying entities and types. The pipeline implements a simple voting algorithm of the identified objects, a more elaborate approach inspired by TF-IDF used in information retrieval as well as a more sophisticated set of CNN classifiers (ColNet) for type identification. Finally, the predicted column types are utilised to enhance the entity annotation of the cell values in the tabular data.

Pipelines like the one implemented by this project, can be used in information retrieval from large unstructured data that could subsequently inform the analysis of 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. Finally, cell values that have failed to be annotated with an entity by the pipeline can then be fed back to the domain experts and knowledge engineers to enhance the KB with the potentially missing entities.

## 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 as follows:

* Column-Type Annotation (CTA): Assign a class from the KG to an entire column of the table
* Cell-Entity Annotations (CEA): Assign an individual entity of the KG to each specific cell

This will be done using existing knowledge graphs (e.g. DBpedia, WikiData) as reference by considering the following tasks:

* Retrieve candidate entities using the provided lookup service APIs as well as custom SPARQL queries
* Examine the effect of using the KG hierarchy “rdfs:subClassOf” to filter down candidate classes
* Use the retrieved entities to predict the correct type for the target columns in the tabular data
* Replicate the code given by ColNet and use that as a baseline to:
  + train individual ccn classifiers for each candidate class and
  + use them to predict the type of the target columns
* Similar to the type prediction, implement a pipeline to identify the entity that each cell value corresponds to
* Use the predicted type of the columns to enhance the performance of entity identification

## Methods used

As mentioned in previous sections, the end-to-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.

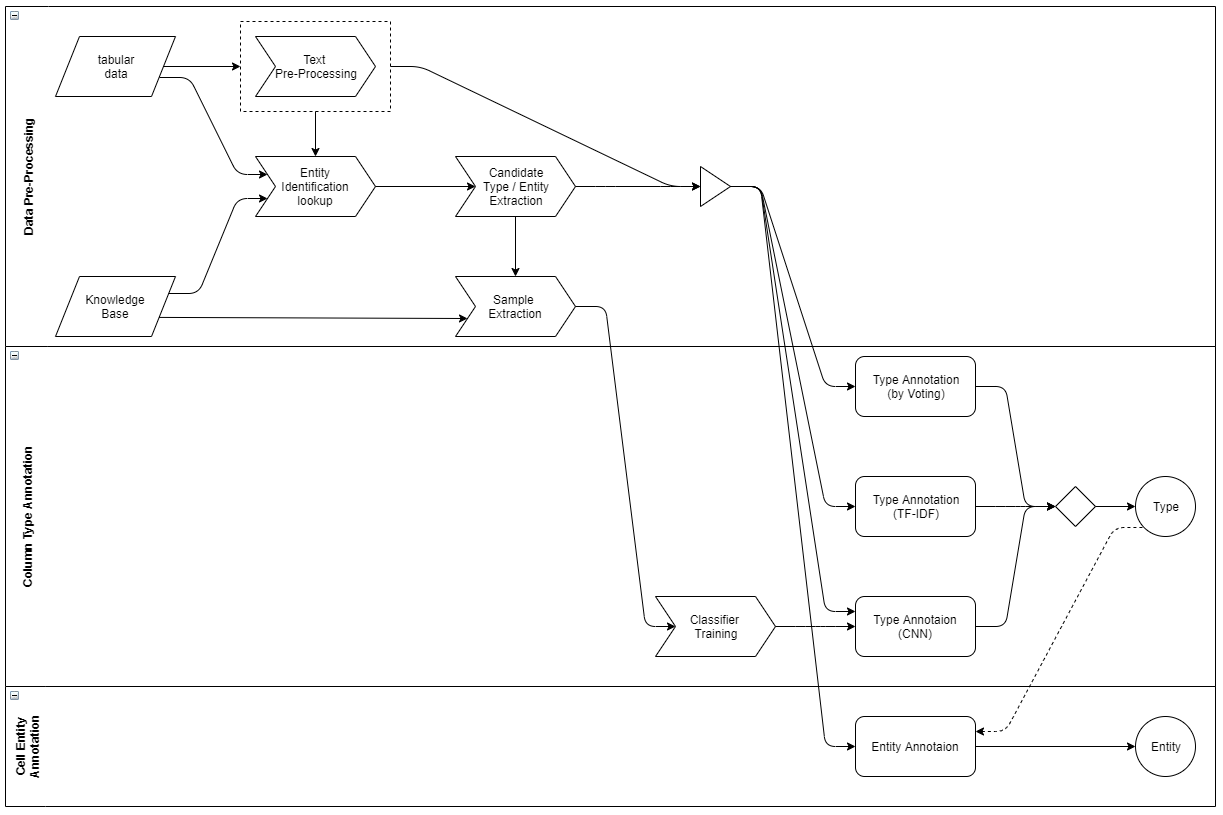


Figure 1. Process of predicting types and entities for tabular data from a reference KG

**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. The knowledge based used for this component in the current project is 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 candidate classes considered in voting and TF-IDF as well as the number of classifiers that are assessed 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 the reverse path is followed. Thereby, entities are extracted, at random, from a given class.

**Classifier Training:** in this step a binary cnn model is created for every candidate type (more details on this in Chapter 3) and trained 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 Type:** In this step all previous processing is brought together to predict a type for every target column in the input tabular data. Several approaches to predict the type of a column have been implemented:

* Voting: This approach simply selects the type with the most votes based on the simple lookup
* TF-IDF: In this approach the candidate types are scored using a TF-IDF formula and sorted based on the results
* CNN: This approach ranks the candidate types based on the score of the relevant binary classifiers for the given column in the input data

**Predict Entity:** This step annotates the cell values with an entity based on the lookup results and optionally using the predicted classes from the previous step to enhance the precision

## 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 structured to save the data for subsequent runs and avoid repeating the lookup step (which is rather time consuming and requires to be online) unless new input data or reference knowledge base were considered. The catch-up jsons files expedited testing as the lookup step takes a lot of time to complete and is deterministic i.e. 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 and the tf-idf logic

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 the solution was designed in a way that can save and load models. These models can be trained offline for any number of classes from the reference knowledge base (which is the time consuming part) and readily used for the type annotation task as and when needed.

With the classifiers implemented the last design step for the CTA 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.

Finally, after a series of experiments highlighted the best approach for CTA the pipeline for the CEA was designed and implemented. This pipeline reused a lot of element from the previous one (i.e. the candidate entities, the predicted classes) and proposed a few different ways of entity identification optionally using as additional input the best results of the CTA pipeline.

## 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 as a series of high-level steps of the proposed implementation. Moreover, the 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 project.

The next chapter (Chapter 2) follows 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 pipelines’ 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 the next steps 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 and the entities that are relevant to cell values. Therefore, by definition we needed to have two sets of inputs:

* The *tabular data* to analyse and predict the types / entities of and
* A *reference knowledge base* containing candidate types / entities the columns / cell would be matched with accordingly

### Tabular Data

In theory, the tabular data could be in any format however for the purposes of this project the input used was from the SemTab Challenges (2019 and 2020) who provided the data in csv files. Each csv file has a header row with the column titles, 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 pipelines:

* Targets: Files containing the column indexes from each file that need to be considered for type annotation or column/row indexes for the cells that need to be annotated with an entity
* 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 annotations, the pipelines use them in order to filter down the columns / cells that need to be considered for the prediction. The pipelines also use the ground truth files to evaluate the annotation precision.

### 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 annotation steps. Throughout this project json was used as a flexible structure for the data so that they can be easily accessible. Moreover, a dictionary is a mutable structure in Python so updates on it 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 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 ignores 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 does 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 has been predicted, to assess the accuracy of each approach that is being tested. Once more if the ground truth, referenced in 3.1.1, is not available, this key 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": {  
 ...

}

}

Restructuring the csv inputs into the above dictionary structure enables the rest of the pipeline to remain agnostic of the input format. Therefore, if the tabular data was presented in a different structure (i.e. not the one provided by SemTab), so long as a component could be built to transform the data in the above dictionary, the pipeline could still proceed with the type annotation without any further changes.

## 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 entities and their types. 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 keywords 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 text pre-processing step originally provisioned in the pipeline (e.g. stemming, lemmatizing) unnecessary. The only pre-processing of the cell value that is performed is the removal of characters (‘[’ and ‘]’) that appeared in some cell values and made 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 followed by less relevant results. For the purposes of this analysis the process is flexible enough to store the x top lookup results for each cell value since. The value of x has been defaulted to 5 to enable more candidate entities and classes to be accessed for each column while at the same time not exploding the size and computational requirements of the experiments (if a very high value was selected for x).

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). Only eWe 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 retrieved entities from the API. There are many instances where, for whatever reason, the lookup API response fails to retrieve the class(es) 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)’ and it is retrieved without any associated classes (i.e. <Classes/>).

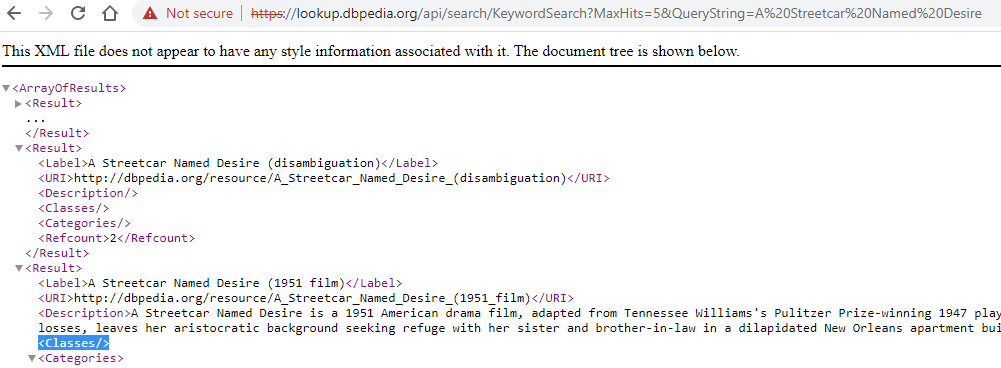


Figure 2. Results from the dbpedia lookup api when querying the cell value ‘A Streetcar Named Desire’

However, when visiting the actual URI of the retrieved entity it appears that the entity does indeed have 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 send 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.

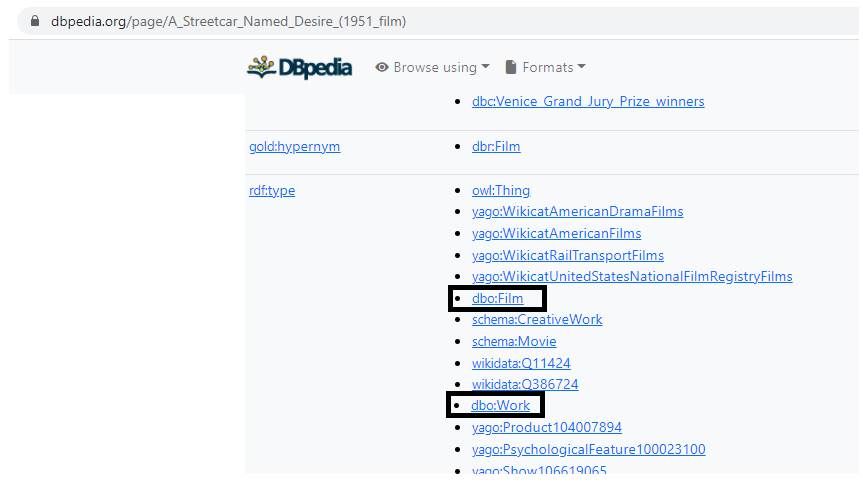


Figure 3. Types associated to the entity ‘https://dbpedia.org/page/A\_Streetcar\_Named\_Desire\_(1951\_film)’

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 columns 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 in tabular data 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 in its own right with the following keys:

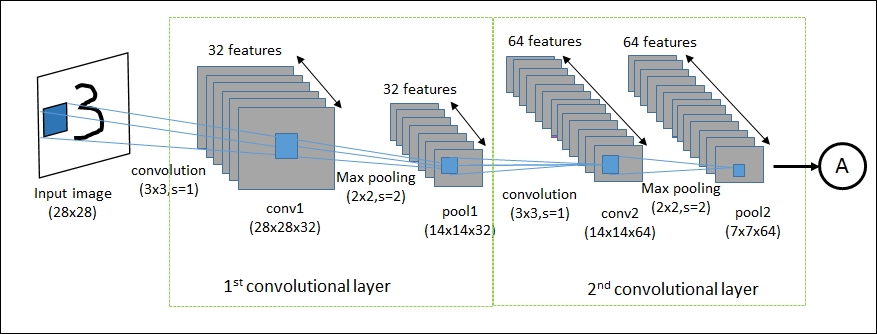
* location: This is an array of tuples where each tuple contains a (file, column index) describing where the value appeared in. As mentioned earlier, each value may appear only once in a given column therefore for the CTA task the actual index of the row the cell value appeared is irrelevant
* candidate\_entities: 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 starting with 1 for the most relevant and ending at 5 for the least relevant. It’s not necessary that all 5 ranks will appear as keys for a given cell value since the pipeline ignores candidate entities that don’t have types in the dbo namespace. This is the reason why rank 2 is missing from the above example
  + candidate\_classes: an array of all the dbo types associated to the specific entity (the can be one or more type per entity)

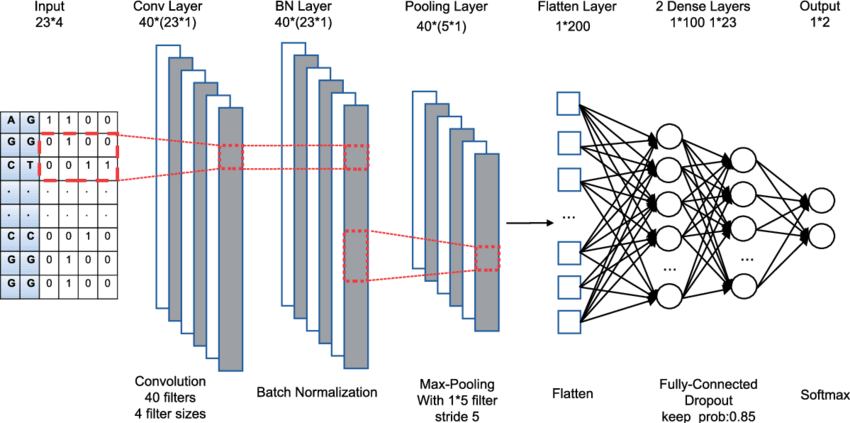
Finally, on top of the *data* and *cell\_values* structure mentioned in 3.2 and 3.3 (above) respectively there are two more structures that reshape that data in order to be used for the next steps of the pipeline. The first is a structure that is used to predict a class by voting whereby each input csv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

MORE….

## Create and train convolutional neural network models

The idea behind the use of the CNNs to annotate a column is to combine multiple cell values from one column to identify collectively the appropriate classes. The CNN created for each candidate types comprises an input layer, followed by several convolutional layers (2-3) with max pulling for feature selection and dropout. The output of each convolutional layer is passed on to the next apart from the very last one that going into a dense fully connected layer after having been flatten and finally there is the last dense layer to provide the binary classification. Relu is used as the activation function for the convolutional and dense layers





Limitations on training size some classes are overlapping so what is a film may also be a playwright

The next sections describe the pre-processing used to create the samples for the input layer. As shown in the above Figure, this input essentially an x by y matrix of numerical values. To reshape the cell values into a format that is compatible with the input layer, the following steps are followed

### Create synthetic columns of cell values

For every column, several synthetic column samples are created with a synthetic column of size x (parameterised as ‘synthetic\_column\_size’). Two approaches have been used for the sample generation:

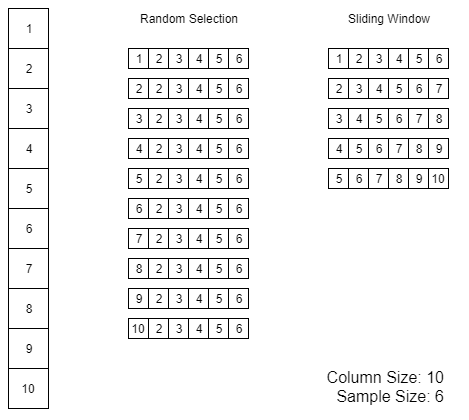


Figure 4. Sample generation from a column of size 10 using the random selection and sliding window approaches

**Random selection:** In this case, the synthetic column contains the cell value itself plus an additional x-1 randomly selected cells. In case the column length is smaller than the number of samples needed (i.e. < x-1) then all cell values are selected and the remaining positions in the synthetic column are populated with nulls.

**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 less samples than the random selection (i.e. [column\_size – x + 1] samples as opposed to [column\_size] samples. 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.

Example

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']

### Break sample of values to sample of words

The next step is to convert the list of cell values in the synthetic column above to a list of words. That is done by replacing special characters, which are typically used to separate words (e.g. '\_', '-', '.', '/', '"', "'"), with spaces. Finally the derived string value is tokenised 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. For this reason the pipeline calculate the average word length the cell values (typically a float between 2 and 3) and sets the value of the sequence size in multiples of the sample column size. For the experiments presented in Chapter 4 these [sample column size] = 10 and the sequence size is three times that i.e. [sequence size] = 30.

### Word2Vec

Finally, each word is converted into a numerical vector based on the pretrained model that was mentioned in 3.1.3. so that vectors of semantically related words approach appear closer to one another. The size of the word vector in the model used is 200 therefore each sample is finally a 30 x 200 size matrix.

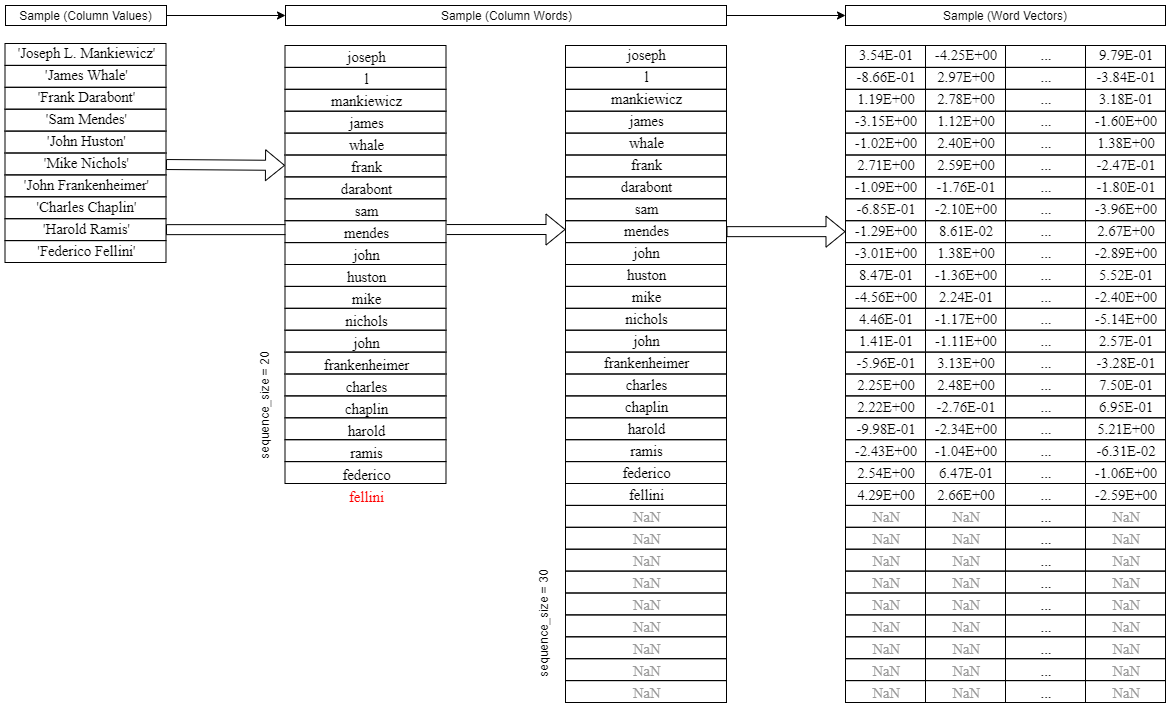


Figure 5. Creation of samples for the input layer of the cnn (used for training and predictions)

## Column Type Annotation

Having pre-processed the data, extracted candidate entities / types and trained the convolutional neural networks for every potential candidate type that appears in the data, the next step of the type annotation pipeline is to predict the types of the target columns. The following sections present the different approaches implemented for the task at hand

### Voting

Using the list of candidate entities and types of each cell value of a given column, the pipeline proceeds with a very basic column annotation by considering the types that are proposed by the retrieved candidate entities. There are several 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 in the api response, as well as
* the entity hierarchies within a knowledge base (i.e. parent child relationship by the rdf:subClassOf predicate) and
* the frequency of the candidate entities in the target columns

### TF-IDF

### CNN

## Cell Entity Annotation

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

# Results

This section presents the results for the experiments conducted with different permutations of the pipeline.

DATASET

## CTA

To assesses the prediction results, the gt truth (i.e. the expected type per column) reference data is used. There are two measures employed:

* Strict precision: This only considers when the predicted class(es) contain the actual class. If the actual class is not in the predicted class(es) the column is counted as a false positive
* Relaxed precision: This measure will also give .5 a point when the predicted class(es) contain a parent class that the actual class is a subClassOf.

### Lookup Voting

This series of experiments aims to find the optimum way of deriving the type of a column (using DBpedia as the reference knowledge graph) based exclusively on the results from the entity lookup API and querying the SPARQL endpoint.

#### Assess Ranking – Equal votes

In this series of tests, a vector of candidate classes is created for each column *Cij*:

(4.1)

where *Cij* is the vector of column *j* in file *i* and ptype\_1…ptype\_n are the number of votes that a candidate class has received, calculated as follows:

(4.2)

where *N* is the total number of candidate entities retrieved for the cell values in column *j* in file *i* and rdf:type(e) is the class that is assigned to the entity e. Please note that in case a retrieved entity *e* is allocated more than one classes, then it contributes more than one votes (i.e. one in each class) and N is also increased accordingly.

Finally, as depicted in (4.2) each entity’s vote has the same weight, regardless of their rank in the retrieved results. For instance, if type *film* is suggested by two entity results, one with rank=1 and one with rank=3 the total votes type film will receive will be 2 (i.e. )

As mentioned in 3.3 the pipeline retrieves the top 5 results from the lookup API. In this experiment we set a threshold increasing from 1 all the way through to 5 to see how the effect of being more relaxed with the retrieved lookup results.

Figure 4. Results of lookup voting for increasing thresholds of candidate entity ranks

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Rank (<=1) | Rank (<=2) | Rank (<=3) | Rank (<=4) | Rank (<=5) |
| 1 | 24.77 (34.86) | 15.6 (27.06) | 17.43 (28.9) | 18.35 (28.9) | 16.51 (27.06) |
| 2 | 57.8 (64.22) | 48.62 (58.26) | 45.87 (56.42) | 43.12 (54.59) | 34.86 (48.62) |
| 3 | 74.31 (80.28) | 62.39 (72.02) | 61.47 (70.64) | 61.47 (70.64) | 57.8 (67.89) |
| 4 | 83.49 (87.16) | 77.98 (83.03) | 75.23 (80.28) | 76.15 (81.19) | 70.64 (78.44) |
| 5 | 88.07 (89.91) | 87.16 (88.99) | 82.57 (84.86) | 81.65 (83.94) | 77.98 (82.11) |
| 6 | 88.99 (90.83) | 88.07 (89.45) | 84.4 (86.7) | 83.49 (85.78) | 83.49 (85.78) |
| 7 | 89.91 (91.74) | 89.91 (91.28) | 87.16 (88.99) | 84.4 (87.61) | 83.49 (87.16) |
| 8 | 90.83 (92.2) | 89.91 (91.28) | 88.07 (90.37) | 85.32 (88.99) | 86.24 (89.45) |
| 9 | 90.83 (92.2) | 90.83 (92.2) | 88.99 (90.83) | 89.91 (91.74) | 88.99 (91.28) |
| 10 | 92.66 (93.12) | 92.66 (94.04) | 90.83 (91.74) | 90.83 (92.2) | 90.83 (92.2) |
| 11 | 92.66 (93.12) | 93.58 (94.5) | 92.66 (93.58) | 92.66 (93.58) | 91.74 (93.12) |
| 12 | 92.66 (93.12) | 93.58 (94.5) | 92.66 (93.58) | 92.66 (93.58) | 92.66 (93.58) |
| 13 | 92.66 (93.12) | 93.58 (94.5) | 92.66 (93.58) | 92.66 (93.58) | 92.66 (93.58) |
| 14 | 92.66 (93.12) | 93.58 (94.5) | 92.66 (93.58) | 92.66 (93.58) | 92.66 (93.58) |
| 15 | 92.66 (93.12) | 93.58 (94.5) | 93.58 (94.5) | 94.5 (94.95) | 92.66 (93.58) |

Table 1. Results of lookup voting for increasing thresholds of candidate entity ranks

As illustrated by the results allowing more candidate entities in the majority vote is hurting the precision of the prediction in comparison to only retrieving the top 1 class. For instance, the expected class is at the top 1 spot for 24.77% of the columns when using only rank 1 entities as opposed to 15.6% when using the ranks 1 and two. In raw number this mean 27 columns have a correct type predicted as opposed to 17 columns.

#### Assess Ranking – Weighted votes

This is a variation of the previous experiment series where the retrieved entities get a vote that is inversely proportional to their rank. The calculation of the vote is as follows

(4.3)

where r is the rank the entity came in in the results. For instance, as illustrated in Figure 2 there are two entities retrieved:

* <http://dbpedia.org/resource/A_Streetcar_Named_Desire> with r = 1 and
* <http://dbpedia.org/resource/A_Streetcar_Named_Desire_(1951_film)> with r = 3

The first entity contributes a vote of 1/1 = 1 in all three associated types [‘Play’, ‘WrittenWork’, ‘Work’] and the second entity contributes a vote of 1/3 = 0.33 in all three associated types [‘Film’, ‘Work’]. The results are presented below in the same setup as in 4.1.1.1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Rank (<=1) | Rank (<=2) | Rank (<=3) | Rank (<=4) | Rank (<=5) |
| 1 | 24.77 (34.86) | 17.43 (28.9) | 20.18 (31.65) | 20.18 (31.19) | 19.27 (30.28) |
| 2 | 57.8 (64.22) | 49.54 (58.26) | 46.79 (56.88) | 44.95 (55.96) | 44.95 (55.96) |
| 3 | 74.31 (80.28) | 64.22 (72.94) | 61.47 (71.1) | 61.47 (71.1) | 61.47 (70.64) |
| 4 | 83.49 (87.16) | 79.82 (84.86) | 77.98 (83.03) | 77.06 (82.11) | 77.06 (82.11) |
| 5 | 88.07 (89.91) | 88.07 (89.91) | 85.32 (87.16) | 83.49 (86.24) | 83.49 (86.24) |
| 6 | 88.99 (90.83) | 88.99 (90.37) | 86.24 (88.07) | 83.49 (87.16) | 83.49 (87.16) |
| 7 | 89.91 (91.74) | 90.83 (92.2) | 89.91 (91.28) | 88.99 (90.83) | 88.07 (90.37) |
| 8 | 90.83 (92.2) | 90.83 (92.2) | 90.83 (92.2) | 90.83 (92.2) | 88.99 (90.83) |
| 9 | 90.83 (92.2) | 91.74 (93.12) | 90.83 (92.2) | 90.83 (92.2) | 89.91 (91.28) |
| 10 | 92.66 (93.12) | 93.58 (94.04) | 91.74 (92.66) | 91.74 (92.66) | 90.83 (91.74) |
| 11 | 92.66 (93.12) | 93.58 (94.04) | 91.74 (92.66) | 92.66 (93.12) | 92.66 (93.58) |
| 12 | 92.66 (93.12) | 93.58 (94.04) | 92.66 (93.12) | 92.66 (93.58) | 92.66 (93.58) |
| 13 | 92.66 (93.12) | 93.58 (94.04) | 93.58 (94.5) | 93.58 (94.5) | 93.58 (94.5) |
| 14 | 92.66 (93.12) | 93.58 (94.04) | 93.58 (94.5) | 93.58 (94.5) | 93.58 (94.5) |
| 15 | 92.66 (93.12) | 93.58 (94.5) | 93.58 (94.5) | 93.58 (94.5) | 93.58 (94.5) |

Table 2. Results of lookup voting for increasing thresholds of candidate entity ranks(rank-weighted vote)

As expected, when only considering the top 1 result the precision of the weighted vote is the same as that of equal vote since all the weights are 1 and so the two methods collapse. However, we see that increasing the rank gradually to 5 the resulting precision is marginally better with the weighted votes compared to the equal vote strategy at least when only a few classes from Cij are considered. As we consider more of the voted classes, the effect of the weighted vote wears off. Figure 5 shows the results when all 5 ranks are considered, and it is apparent that when selecting the less than the top 6 voted classes the precision with the weights is always higher. However, from the top 6 onwards the precision of the two methods flips back and forth.

Figure 5. Results of lookup voting when all 5 ranks are considered with equal vote and majority vote respectively

Remove a class if at least one of its offspring are present

### Lookup with TFIDF

For the next set of experiments, we deviated away from having each candidate entity cast votes for the associated classes and followed an approach (inspired by DAGOBAH) that resembles the logic of TFIDF.

First, we calculate the equivalent of the term frequency as the follows:

(4.4)

where is the frequency of the candidate class in the given column of the file, an is the total number of candidate classes appearing in this column. Once again, the experiments examine different thresholds for the rank of the lookup results so when the number of candidate classes will be greater than the number when .

Next, we calculate the inverse document frequency as follows:

(4.5)

where is the total number of target columns (i.e. columns we want to predict the type of) and is the total number of columns for which the specific type appears at least once as a candidate class. The denominator is adjusted by adding 1 to avoid division by zero for types that are not present in any target columns even though that use case is not possible in this implementation since we would not be calculating the tf-idf for classes that don’t appear even once.

Finally, for each candidate class for each column we calculate the tf-idf score as follows:

(4.6)

Following the implementation of the above tf-idf approach the same series of results where executing, gradually increasing the rank of the considered candidates by one. The results are illustrated in Table 3:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Rank (<=1) | Rank (<=2) | Rank (<=3) | Rank (<=4) | Rank (<=5) |
| 1 | 75.23 (78.9) | 72.48 (76.61) | 67.89 (70.18) | 54.13 (58.72) | 52.29 (53.67) |
| 2 | 81.65 (84.4) | 79.82 (83.49) | 77.06 (80.28) | 68.81 (72.02) | 69.72 (72.48) |
| 3 | 87.16 (88.53) | 88.99 (90.83) | 84.4 (85.78) | 78.9 (80.28) | 74.31 (76.15) |
| 4 | 87.16 (88.99) | 88.99 (90.83) | 88.07 (89.45) | 83.49 (84.4) | 78.9 (79.36) |
| 5 | 88.07 (90.37) | 91.74 (93.12) | 89.91 (90.83) | 86.24 (86.7) | 80.73 (81.19) |
| 6 | 88.99 (90.83) | 92.66 (94.04) | 90.83 (92.2) | 88.07 (88.99) | 81.65 (82.57) |
| 7 | 90.83 (92.2) | 93.58 (94.5) | 92.66 (93.58) | 90.83 (91.28) | 84.4 (84.86) |
| 8 | 91.74 (92.66) | 94.5 (94.95) | 92.66 (93.58) | 90.83 (91.74) | 85.32 (86.24) |
| 9 | 91.74 (92.66) | 94.5 (94.95) | 94.5 (94.95) | 90.83 (92.2) | 86.24 (87.16) |
| 10 | 93.58 (93.58) | 94.5 (94.95) | 94.5 (94.95) | 91.74 (92.66) | 86.24 (87.16) |
| 11 | 93.58 (93.58) | 95.41 (95.41) | 95.41 (95.41) | 94.5 (94.95) | 88.99 (89.91) |
| 12 | 93.58 (93.58) | 95.41 (95.41) | 95.41 (95.41) | 94.5 (94.95) | 89.91 (90.83) |
| 13 | 93.58 (93.58) | 95.41 (95.41) | 95.41 (95.41) | 94.5 (94.95) | 90.83 (91.28) |
| 14 | 93.58 (93.58) | 95.41 (95.41) | 95.41 (95.41) | 94.5 (94.95) | 92.66 (93.12) |
| 15 | 93.58 (93.58) | 95.41 (95.41) | 95.41 (95.41) | 94.5 (94.95) | 92.66 (93.12) |

Table 3. Results of lookup using the tfidf logic for increasing thresholds of candidate entity ranks

Compared to the previous two voting approaches, the tf-idf approach has a much superior performance with the precision of the rank 1 results jumping from ~25% with voting to ~75% with the tf-idf logic. The intuition behind this superior performance is that tf-idf penalises types that may be too generic and thus appearing very frequently as candidate types for many columns and promotes more specific (usually subclasses of the above). However, it still allows for those generic classes to be selected as types for a column if their frequency of appearance for a specific column is quite high.

To further support this theory, we executed a small series of experiments where we filtered the list of candidate classes of a column by removing parent classes when any of the classes offsprings also appeared in the list of candidates.

The results are shown in Figure 6. It is clear that if we only considered the top 1 candidate class as the prediction even the crude approach of removing the parents provides better results that the equal and weighted vote however it is also quite inferior when compared to the tf-idf logic. Moreover, the results indicate that by removing the parent classes we filter out some useful candidates. For instance for both tf-idf and equal/weighted vote the expected class for 90% of the columns is in the top 5 candidate classes but this percentage drops to 60% when removing the parent classes from the list of candidates.

Figure 6. Comparative results of prediction based on voting with all candidate classes, with a subset of candidate classes removing parents and with tf-if for rank 1 entitities

To further explain the above results, assume the following 2 lists of 5 candidate types for two different columns (taken from actual data that were processed in the pipeline). Only the top 5 candidates instead of the full list of candidate classes are presented for illustration purposes:

|  |  |
| --- | --- |
| **File 1**  Filename: 58891288\_0\_1117541047012405958  Ground Truth: Film  Column: 1  Equal Vote:  ('Work', 20.77922078),  ('Film', 15.58441558),  ('Agent', 6.49350649),  ('Organisation', 5.19480519),  ('Location', 3.8961039),  ('Place', 3.8961039)  TF-IDF:  ('Film', 0.16422260610432587),  ('Work', 0.10112317149884212),  ('Book', 0.048329385843283626),  ('WrittenWork', 0.044680536816172754),  ('City', 0.027370434350720983),  ('Settlement', 0.02242739889841565)] | **File 2**  Filename: TOUGH\_WEB\_MISSP\_celebrities  Ground Truth: Person  Column: 1  Equal Vote:  ('Agent', 33.33333333),  ('Person', 33.33333333),  ('Artist', 8.33333333),  ('Athlete', 8.33333333),  ('BaseballPlayer', 8.33333333),  ('MusicalArtist', 8.33333333)  TF-IDF:  ('Person', 0.23980503579351845),  ('BaseballPlayer', 0.22801842317240886),  ('Athlete', 0.134119826036175),  ('Artist', 0.1199615729698919),  ('MusicalArtist', 0.11490952115185567),  ('Agent', 0.09754419253721236) |

In the first file the ground truth is Film and is a subClassOf type Work which also appears in the list.

In the second file the ground truth is Person which is a parent class of Artist and Athlete that appear in the list, Artist is the parent of class MusicalArtist and Athlete is the parent class of BaseballPlayer.

Let’s examine the effect of the different approaches:

**Equal Vote:** The equal vote will always favour the parent class since a **Film** in the majority of the cases is also a **Work.** In fact there are certain retrieved entities that are not the classified as Film but rather as a Play, for instance, because the lookup return the theatrical rather that the cinema version. As a result this voting strategy may prove problematic when the expected type is a specific rather than a generic one. For the column in File 2 the equal vote strategy scores the Person class higher than its offspring (even though there is still an issue with very generic types such as Agent)

**Removing of Parents**: In this approach the list of candidates for File 1 will reduce to [Film, Agent, Organisation, Location, Place] and between them the highest scored type i.e. Film, is the correct one. However in the case of the second file, by recursively removing the parents, the remaining candidates are reduced to [BaseballPlayer, MusicalArtist] none of which is generic enough to describe the type of the values in the target column. Therefore this strategy of removing the parents will also fail to predict the correct class.

**TF-IDF**: The tfidf approach seems to be working in both cases and producing the correct prediction.

For file 1, given that a more generic class is most probably to appear more frequently in other columns as well (for instance columns describing Artwork, MusicalWork, WrittenWork, etc) the idf, which is the inverse log frequency will be lower. As a result it will decrease the tfidf score of Work, allowing the class Film, which has a comparable tf in this column, to rise above. In this case, Film is the dominant offspring for this column as a result tf-if will select that.

For file 2, however, no subclass of the parent Person appears to be dominant. As a result, even though the idf of person is lower compared to 'BaseballPlayer' and ‘MusicalArtist’ the term frequency of those 2 offspring is so low (there is no dominant between them) that the tfidf of either of them is still lower that that of the Person, which being a more generic class in this case, can better describe all the values in this column

Returning back to the results in Table 3, similar to the two test series presented in 4.1.1 the results when only considering entities of rank 1 are superior to those when taking into account lower ranks. For instance, the precision drops by 1/3 from ~75% to ~54% when considering entities up to rank 5. This is the final confirmation that allowing anything other that the most relevant retrieved entity introduces more noise, on top of additional candidate to process.

Moreover, the TF-IDF approach seems to converge with the voting approach both of which manage to come up with a relatively good top 5 that will contain the expected class for approximately 90% of the target columns.

## CEA

15.7%(1,900-2,350)

# Discussion

9.5%(1,150-1,400)

# Evaluation, Reflections, and Conclusions

10%(1,200-1,500)

# Glossary

|  |  |
| --- | --- |
| Term | Description |
| Knowledge Base |  |
| Class/ Type |  |
| Entity |  |
|  |  |
|  |  |
|  |  |
|  |  |

# References

# Appendix