Semi-Automatic Schema Matching: Challenges and a Composable Matcher Based Solution

August 10, 2018

Student: J. Bottelier 10747338
Supervisor: dr. Z. (Zhiming) Zhao

Company: FDMediaGroup
Supervisor: Thijs Alberts

August 10, 2018

Abstract

During data integration it often occurs that two databases with different schemas have to be integrated. This process is called schema matching. Automating part of or the entire processes of schema matching can essentially accelerate the data integration procedure of human experts and thus reduce the overall time cost. A semi-automated solution could be that a system predicts the mapping based on the schema contents, a human expert could then evaluate the predicted mapping.

This thesis discusses a highly configurable framework that utilizes hierarchical classification in order to match schemas. The experiments performed within this thesis show that the configurability and hierarchical classification improves the matching result, and it proposes an algorithm to automatically optimize such a hierarchy (pipeline).

Contents

1 INTRODUCTION

Many online services nowadays require users to input their raw data into a system. Problems with such raw data can occur when developers do not account for all the possible flaws within this data. It could be the case that information is missing, or is written in the wrong format.

This thesis is written as part of an internship at Company.Info, where such a problem is also occurring. Company.info provides complete, reliable, up-to-date company information and latest business news from all organizations in the Netherlands. Clients often use their services to enrich the data they have in their own databases, or to fill in missing data. When a client wants to use a particular part of the service they can upload a csv file. The Company.info service then consists of filling in the missing data in the csv files. Their platform can fill in all the column and row data based on the instances that are already present within the file. Their system first however needs to know what data is already present before querying in their own database to fill in the missing data. Recognizing the types of data that are already present in the file is a time consuming task currently performed by humans. Automating this process could reduce the cost of mapping the csv columns to the internal Company.Info database.

During data integration it often occurs that two databases with different schemas have to be integrated. This process is called schema matching. It is described as the task of identifying semantically equivalent or similar elements in two different schemas[12]. In this situation, the same problem occurs, a mapping has to be created from one database to the other. Automated data integration offers opportunities to solve these problems by letting machines interpret the data and automatically create a mapping based on semantic or syntactic features.

Automating part of or the entire processes of schema mapping can essentially accelerate the data integration procedure of human experts and thus reduce the overall time cost. However, several challenges make such automated mapping difficult or even impossible. Many problems can occur during the mapping process. Matches might not be found, or even worse, false positives are found. In addition, one data source might not fully match with the other data source, data source A could contain information that does not cohere with the data found in data source B. Source A could also contain less information than source B, in which case a complete mapping is impossible.

Automating the process is a difficult task if you consider all of the different formats and data types a schema might contain. Features which contain useful information for dataset X may not be applicable to dataset Y. Because of the diversity in the datasets of this problem domain, it would be useful to have a framework in which you can experiment with automating the mapping process and which can also be heavily customized according to the needs of the data. Because of the heavy customizability, it would also be useful if such a test framework would provide feedback on how the matching process can be improved.

Since completely automating the mapping process could be impossible in certain cases, human interpretation can not be excluded from the mapping process. This is why this thesis focuses on semi-automating the process, which could reduce the time cost for creating a mapping.

1.1 Research Objectives

Automatically creating a mapping from one schema to the other is difficult task. A semi-automated solution could be that a system predicts the mapping based on the schema contents, a human expert could then evaluate the predicted mapping. There are many challenges and questions that need answering when creating such a system. Since all databases are different, a general solution might not be applicable. Customization is therefore needed for each use case. Guiding users through the customization process can aid in predicting a better mapping and therefore improving the semi-automated mapping process.

The goal of this study is to create a framework that can measure the performance of definable algorithms whose goal is to map schemas or to map single entities into a schema. This study will aim to answer the following question:

How can an effective semi-automated schema matching pipeline be created and customized for a given dataset?

To answer this research question, the following helper questions need to be answered:

- What are the algorithms that can be used for schema matching?
- What are the key performance indicators for schema matching?
- What are the limits of the semi-automated framework?
- How can outliers be included in the customization of a semi-automated schema match-

ing framework?

This thesis will report on how the framework works, on how such a configurable pipeline can be build, compare the performance of different matchers across datasets and a report on the constraints of the framework. The framework is published for reproduction and further research within the scientific community at https://github.com/JordyBottelier/arpsas.

During section 2 we will introduce what the current state of the art schema matching possibilities and principles are and how they will be applied in this thesis. In section 3 we will explain how the framework was designed and how it works. After that we will introduce the experiments that will be performed using the framework in order to validate its functionality and to answer the research questions.

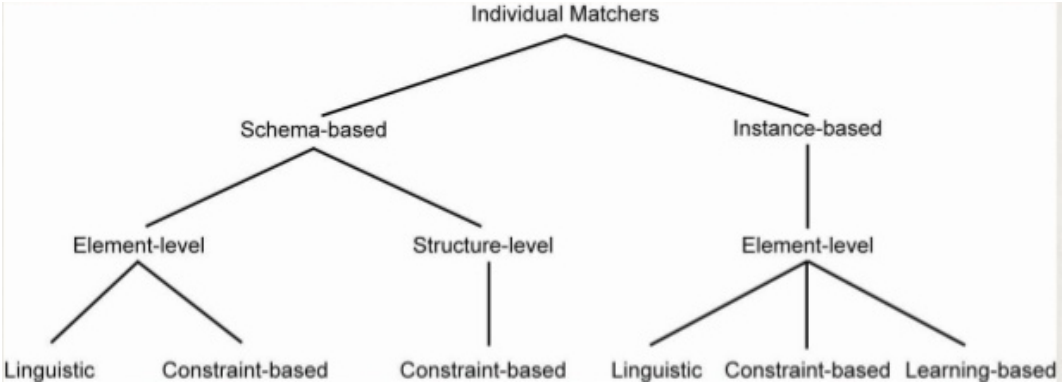
2 STATE OF THE ART

The work that will be discussed in this section has been selected in order to elaborate on the different schema matching approaches that already exist, and to elaborate on the implementation possibilities for a schema matching pipeline.

2.1 Schema Matching

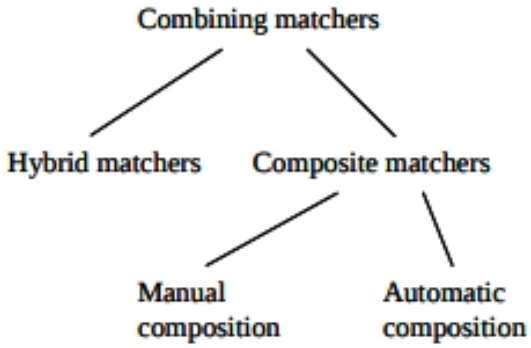
Schema-matching is a large research field. Different solutions and approaches to the problem have been proposed and collections of them can be found in survey papers such as Rahm and Bernstein, 2001 [12] or Giunchiglia et al., 2005 [7]. A taxonomy of the different solutions exists, which was once proposed by Bernstein et al., 2001 [2]. When matching schemas, two sub-problems can be distinguished. First of all, there is the problem of creating individual matchers based on single match criterion as depicted in figure 1. Secondly there is the problem of combining the individual matchers into either a hybrid matcher (which uses multiple matching criteria for a single match), or combining results from multiple individual matchers into a composite matcher (figure 2). A matcher is a component in the system that can create a mapping from one schema element or structure to the other.

Figure 1: Schema Matching: Individual Classification



Source: A survey of approaches to automatic schema matching[12]

Figure 2: Schema Matching: Hybrid Classification



Source: A survey of approaches to automatic schema matching[12]

The individual matchers follow the following classification schemes:

- *Schema-based* vs *Instance-based:* Matching approaches use either the schema contents (instance data) or only schema-level information.
- *Element* vs *structure:* A match can be created for an individual schema element or for a combination of elements in a structure (such as a column).
- *Linguistic* vs *Contraint-based:* An individual matcher can use linguistic features such as names or descriptions. A feature is an individual measurable property or characteristic of a schema element, such as the amount of words in a single entry. Another approach is to base a match on the data type, value ranges, uniqueness or foreign keys.
- *Learning Based:* Matching done based on machine learning, which in turn can use constraints or linguistic features.

Automated schema matching can be rule based (constraints) or learning based [1]. Rule based classifiers match schemas based on pre-defined rules, the "knowledge" of the system is often coded into these rules. By applying different algorithms, a mapping is often computed between two schemas or instances are matched with corresponding columns. After researching the approaches using Google Scholar, we can conclude that from all novel publications regarding automated schema matching, most utilize a learning based approach based on linguistic features.

This study aims to create a framework in which the user can experiment with learning-based and constraint-based approaches on an element or structural level in order to create and customize an effective schema matching pipeline for their own database.

2.2 Named Entity Classification

Schema matching on element level shares many aspects of the named entity classification field. Entities are things such as persons, locations or organizations. The field of named entity classification tries to classify unknown entities based on a variety of methods [9].

Supervised Learning (SL) algorithms in this field often use corpus based techniques (like Natural Language Models, or a bag-of-words approach). The downside of these techniques is that they require a large corpus and string-representations of entities that are not present in the corpus can often not be classified correctly.

Semi-Supervised Learning (SSL) algorithms often rely on a set of rules or constraints to train a prediction component. Such rules can be used for a program to detect new entities that can be used for the learning or classification process. Advantages of such methods are that little initial training data is needed, and that the program will train itself over time. The downside of such algorithms is that the performance can deteriorate quite quickly when noisy data is introduced [13][9].

Unsupervised Learning (UL) is usually based on clustering groups of entities together based on their context, lexical patterns, or statistics computed on a large unannotated corpus[9]. Studies also exist where syntactical features are used to cluster entities together [4].

The methods used within the entity classification field usually rely on the detection of the entity within a text (Named Entity Recognition or NER), and then a classification using the context of the entity, as well as linguistic features. The classification of entities based on solely linguistic features can also be used on schema instances. Instances from an input schema could be mapped by a classification strategy to a target schema. Syntactical (or linguistic) features defined in [4] or [9] could be used when creating feature vectors for either supervised or unsupervised learning algorithms. When classifying entire columns a strategy could also be to make a prediction for every individual element and then classifying the column as the most occurring prediction.

2.3 Existing Schema Matchers

There are already existing matchers such as Automatch, COMA, and SemInt. There are many more, and they all focus on different aspects and implementations of schema matching. COMA follows a composite approach, which provides an extensible library of different match-

ers and supports various ways for combining match results [6]. The system utilizes schema information, such as element and structural properties. COMA operates on XML structures and returns matches on an element level.

The framework from this thesis is most similar to Automatch. Automatch is a single strategy schema matcher that uses a Naive Bayes matching approach. It uses instance characteristics to match attributes from a relational source schema to a previously constructed global schema[6]. Data in ARPSAS is also matched to such a global schema. ARPSAS however leaves room for implementation, you can define and test your own strategies.

SemInt computes a feature vector for each database attribute with values ranging from 0 to 1. Schematic data and instance data are both used in this process. These signatures are then used to first cluster similar attributes from the first schema and then to find the best matching cluster for attributes from the second schema [8].

There are many more examples of functioning schema matchers[6]. These three were selected to give the reader an impression of the implementation possibilities. Implementations can differ based on the structure of the schema (tree or column based), and based on the type of data.

The current work in the schema matching field however is limited by a single implementation per matcher. ARPSAS differs from the presented matchers in the sense that it is not a fully functioning matcher upon initiation. Users can create and experiment with their own schema matching pipeline in order to customize it to fit their specific problem. There exists no fully automated schema matching solution yet. In all of the presented schema-matchers there is a human present evaluating the mapping. This is not something that ARPSAS is to overcome. ARPSAS aims to improve on the customizability of the matching process, and with that optimizing that process. Not all schema matchers are also available for free, ARPSAS is, at https://github.com/JordyBottelier/arpsas.

3 THE ARPSAS FRAMEWORK

ARPSAS stands for: A Reconfigurable Pipeline for Semi-Automatic Schema Matching, and aims to solve the customizability of the matching problem by providing an environment in which a user can create, configure and experiment with their own schema-matching pipeline. We define a pipeline in this context as a chain of matchers (section 2.1) that is used to classify data. The goal of such a pipeline in the schema matching context should be to automatically map new schemas into a pre-defined global schema. During the next sections, we will refer to this pre-defined global schema as the 'target-schema' or simply 'target'. We will refer to the new schemas that need to be mapped as the 'source-schemas' or 'sources'. In this section the framework will be explained.

3.1 Requirements

The ARPSAS framework has been designed to be generic in order to support users in trying different approaches to match schemas. Before building the framework there were a couple of things to consider:

- Why should a pipeline for semi-automated schema matching be adaptable?
- What configurability or adaptability should the framework provide?
- What can we do to make the framework useful for generic purposes?

The goal of this particular study is to experiment with creating and customizing an effective schema matching pipeline for a given dataset to see if customization leads to an improved matching result. Since a lot of datasets are different and require a different pipeline configuration, the framework should be adaptable. If we want to allow users to experiment with such a pipeline, we first have to define what aspects such a pipeline has. This is where the state of the art section (section 2) finds its use. In this section we investigated what possibilities there were to match schemas automatically. It should be possible to implement any sort of learning or constraint based matcher. In order to do this the framework should therefore be highly flexible in several places:

1. The framework should be able to pre-process data and store it for future usage.
2. The framework should allow a user to pass any structure or element data (section 2.1) from a source or target schema to the framework for pre-processing.

3. It should be possible to pass any pre-processed data to a matching component in the framework.
4. It should be possible to experiment with any sort of matching algorithm within a matching component.
5. It should be possible within the framework to create a pipeline of matchers which the user can configure.

The framework can also finds its use outside of a schema matching context. Users could use the configurable pipeline structure to experiment with optimizing hierarchical classification within any environment. The contribution of ARPSAS is also more valuable if the pipeline structure can be used for experimentation outside of the schema matching context. Even though actually utilizing ARPSAS for this purpose is out of scope for this research project, it should still be possible because it adds more value to the contribution. Therefore, creating and configuring the pipeline should be independent of utilizing it for schema matching.

3.2 Architecture

In order to satisfy the requirements from the previous section, the architecture has been split into two separate components depicted in figures 3 and 4. The sequence diagrams are depicted in figures 5 and 6.

Figure 3: ARPSAS System Architecture

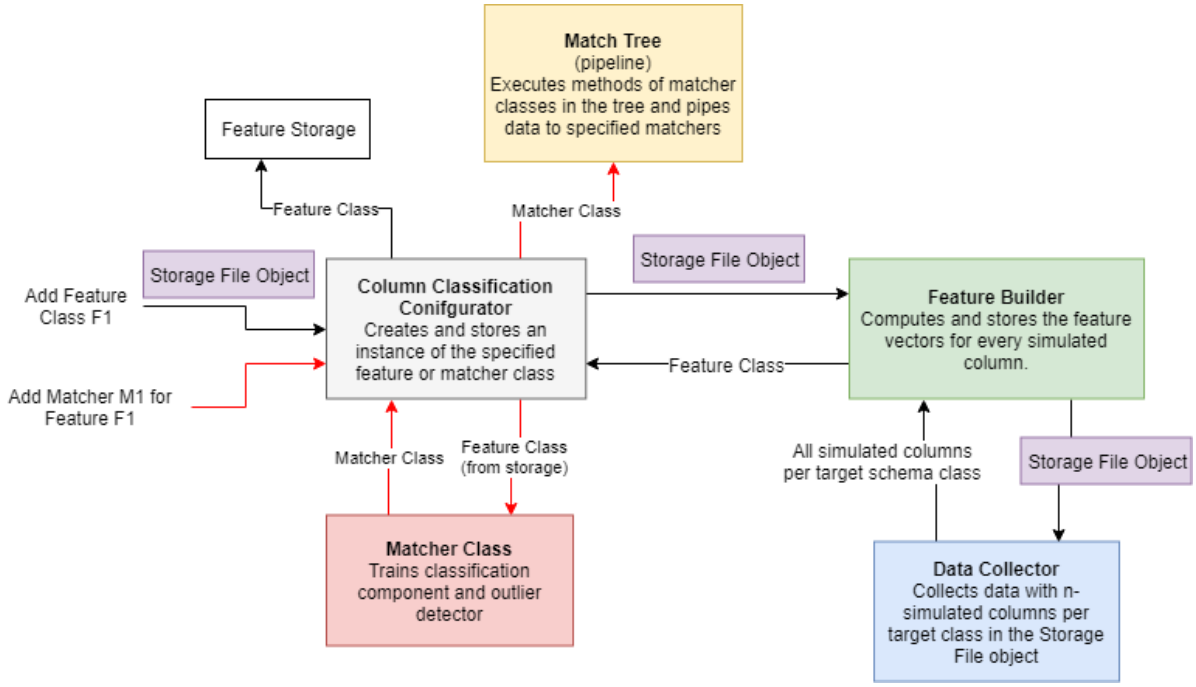


Figure 4: ARPSAS Schema Matching Architecture

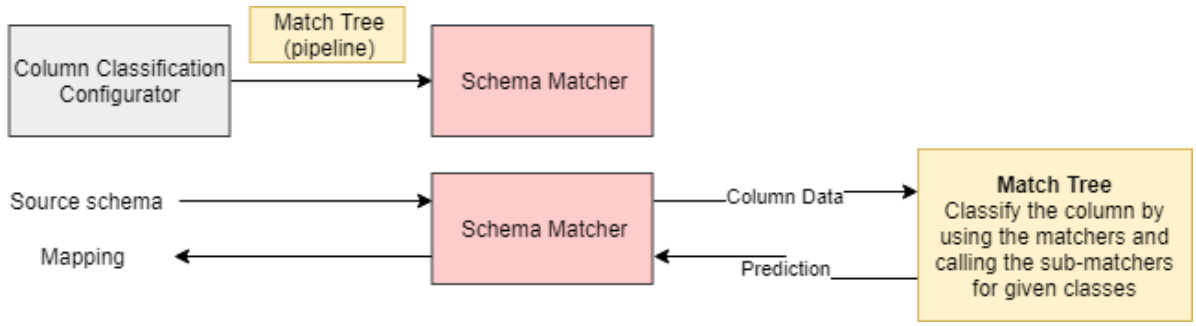


Figure 5: ARPSAS Pipeline Configuration Sequence Diagram

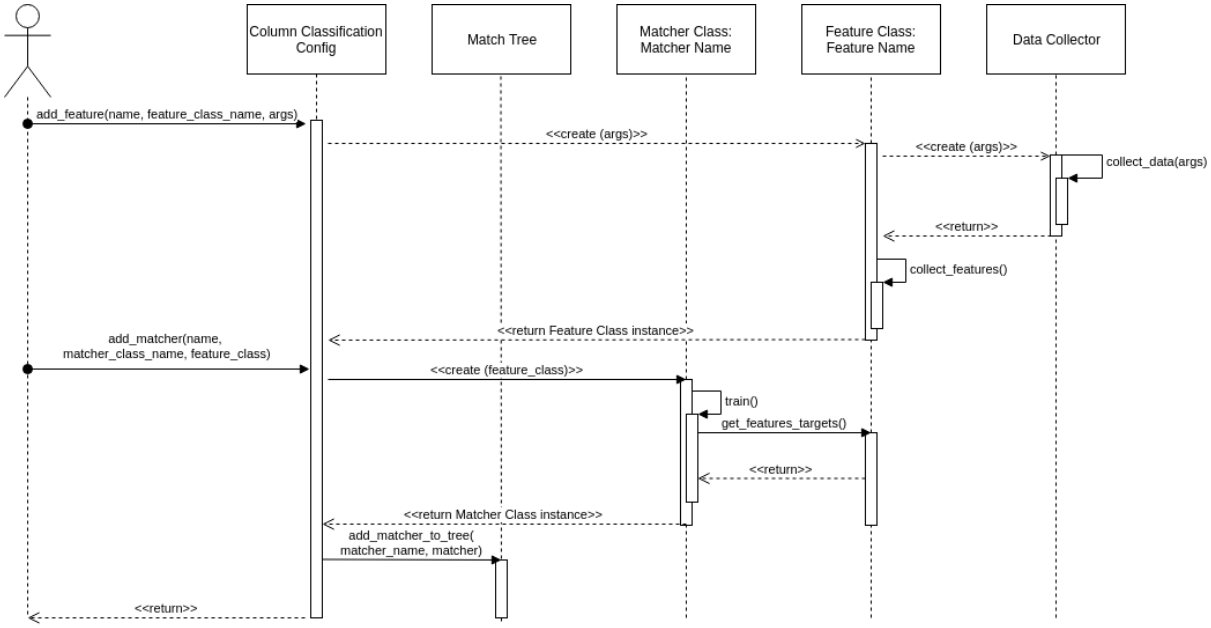
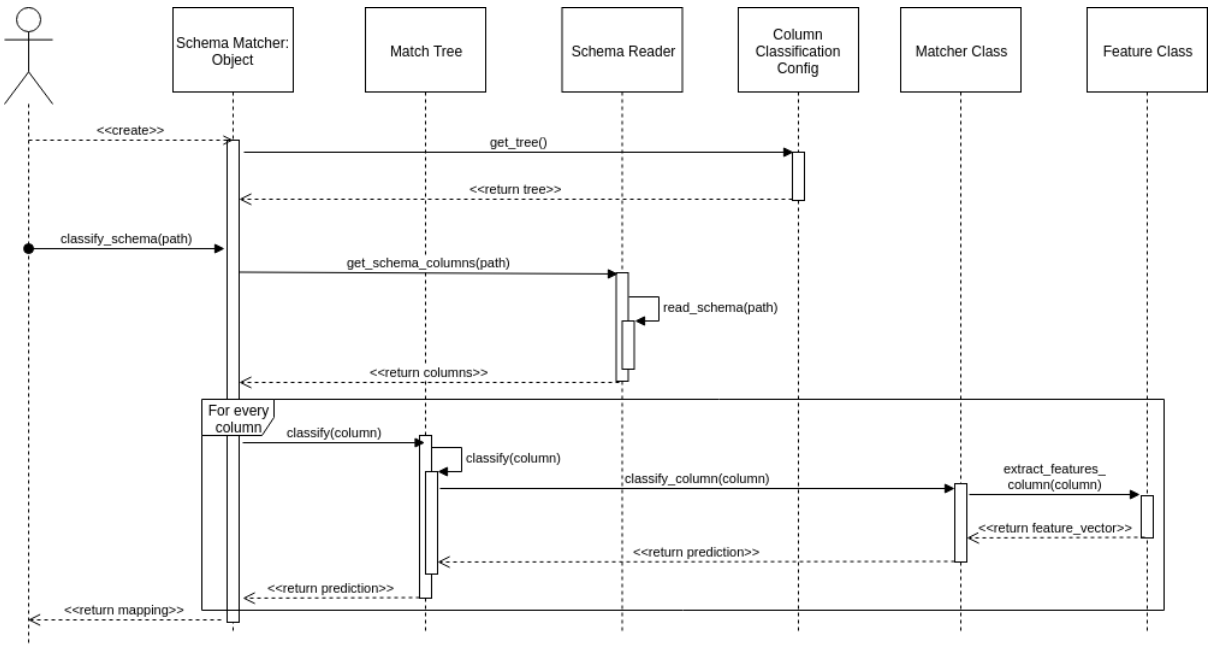


Figure 6: ARPSAS Schema Matching Sequence Diagram



We will argue that this architecture satisfies the requirements while we explain how it works.

The pre-processing components discussed in section 3.1 are called *feature builders*, and the matching components are called *matcher classes*.

Based on the requirement, we designed functional components which will be discussed in the upcoming sections. The interactions among these components are depicted in figures 5 and 6. Globally, the framework works in a schema matching context using the following steps:

1. Data Collection: Collect all the column data from the target schema for the features builders.
2. Feature Building: Build feature vectors for the target schema column data.
3. Matcher Building: Use the feature vectors to train a matching component.
4. Building a pipeline: Build a pipeline of matchers.
5. Load the pipeline: Load the pipeline into the schema matcher.

There are already options for each step created in the framework, but a user is free to implement their own. If they want to build a pipeline of matchers and still be able to use all of the provided methods they should however stick to the given format of methods specified in the framework for each class.

3.3 Data Collector

The first component in the system is the optional data collector. It is optional because you can also provide data manually. This is done to satisfy the requirement that any sort of data can be passed to the pre-processing components. The data collector itself was designed to support an experimentational setup. It has configurable options which allow a user to change one variable within the test setup at a time and then run experiments using the collected data.

The data collector reads lists from files. If you arrange for your target schema data to be separated per class in different folders, the data collector can read it, and immediately process it. You can tell the data collector to collect files from different folders and merge these files

under a single class name using a data-map. Such a data-map is presented in the following example:

```
data_map = {
    'numbers': ['postcode', 'telephone_nr'],
    'name': ['company_name', 'city'],
    'email': ['email'],
    'address': ['address'],
    'domain_name': ['domain_name_1']
  }
```

In this example, all data from the files of the folders 'postcode' and 'telephone_nr' will be read and placed together under the class 'numbers'. The same will be done for the 'name' class. As shown, you can also simply use files from a single folder per class. You can also provide a list of classes to the Data Collector. The list items however should correspond to the names of the folders in which your classes lie. These names will be used as class names. The data-map or list of classes are passed to the data collector by using a *Storage Files* class instance, as is depicted in figure 3.

The data-map structure was created to already allow more flexibility within the configuration of a pipeline, a user can now easily group data together under a single class-name. This is useful when you want different types of data to be classified together under a single class in order to separate it later in the pipeline.

The target schema data that is collected is used, during this research, for training and testing the matchers (after processing). The target schema can have multiple columns. Each column in this target schema is a class for the matchers. A feature (more on this in section 3.4) could be computed over the data from an entire column. If we would compute the feature vectors for the target schema, in this way, we would end up with a single feature for each class in the target schema. Nearly all machine learning algorithms however require more than a single feature vector per class in order to work effectively.

The Data Collector solves this problem by being able to randomly divide the data of a single target-schema column into multiple columns for each class. When testing the matchers, it is useful to be able to simulate the number of target schema columns that will be used for the training process per class in order to find out how they influence the test results.

You can provide the Data Collector with parameters for the amount of target schema columns you want to simulate per class and the amount of instances you want in every simulated

target schema column. The data of a single target schema column is randomly divided among the simulated columns. The output of the data collectors looks as follows:

```
output = {
    'class1':[col_1, .. col_n],
    'class2':[col_1, .. col_n]
}
where:
col_n = ['instance1', 'instance2', .. 'instance n']
```

If a user wants to provide their own data, they should pass it on in the same format, at least if they want to utilize the pre-defined features.

3.4 Features

Feature components in the system are designed to further pre-process the target schema data and to store this within the framework for later usage. The data collected by the Data Collector (or ones own data) should be used first by the feature builders. Since this project focuses on learning-based schema matching, the feature builders that have been implemented are used to transform the target schema column data in feature vectors per column upon initialization (as is depicted in figure 5). The output of the feature builders here is a list of feature vectors and a list of target values (classes). Features should be designed according to your classification needs. Feature builders can be adapted to pre-process any kind of data, and already allow a user to experiment with different algorithms. They should be used to solve the pre-processing problems and allow more flexibility within the entire environment.

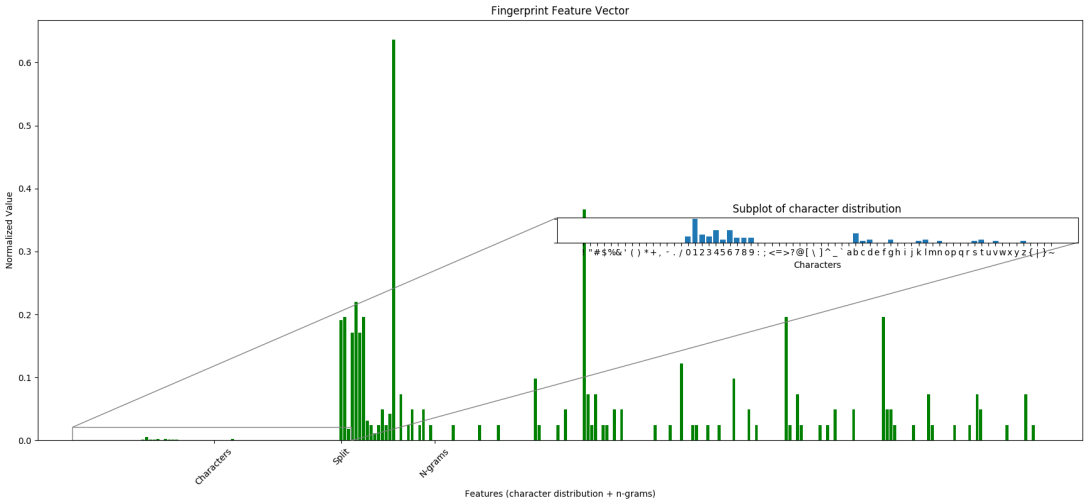
The feature builders that will be discussed in the next sections have been designed to pre-process data for the experiments that will be performed. A user should adapt feature builders according to the data in their target schema. The goal of a feature is to process the data in such way that a matcher can differentiate between the different columns in the processed data. There are four feature builders already implemented in the framework:

3.4.1 Fingerprint

This feature class calculates the datapoints based on the character distributions and n-grams of the inserted column data. For each column, all characters present are counted, as well

as the N-grams, and then the result is normalized. For a random example class, the full fingerprint feature-vector is shown in figure 7.

Figure 7: Feature: Fingerprint



Fingerprint character distribution and n-gram example for a column

N-grams are re-occurring sequences of N characters. They can be used to find patterns in words. For instance, a lot of street names contain the word 'lane'. If the N-grams *la*, *an* and *ne* are found often in a column, the chance is higher that the column consists of street names in comparison to postal codes (which usually only have 2 letters so this pattern could never occur).

3.4.2 Syntax Feature Model

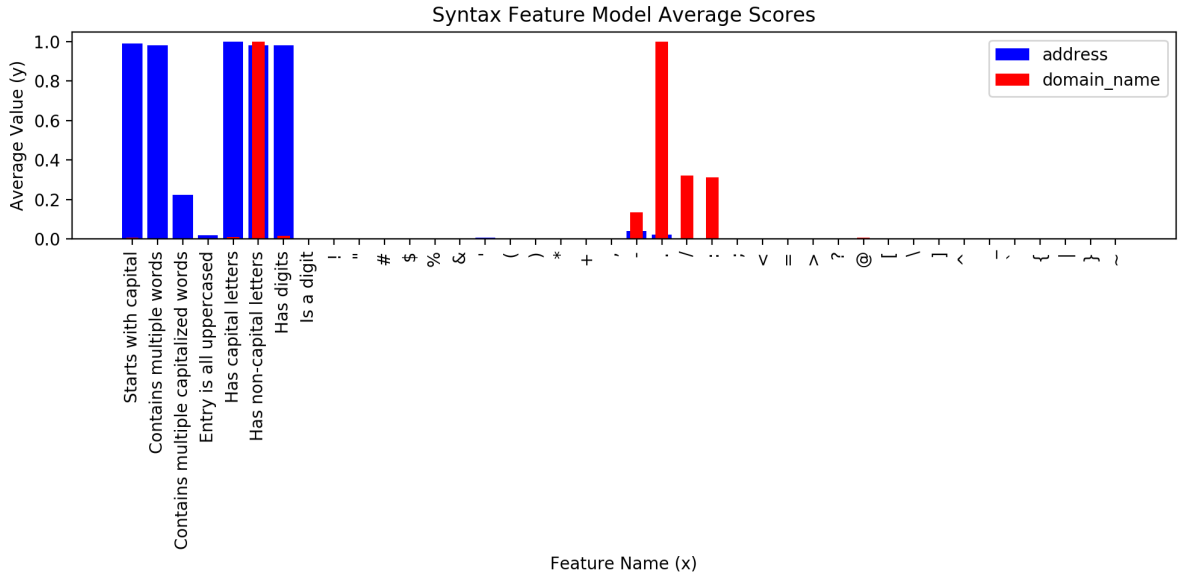
Based on a combinations of the features used in [4] and [9], the syntax feature model is a simple instance-based feature class that checks whether or not the following data is present:

- Instance starts with a capital letter.
- Instance contains multiple words.
- Instance contains multiple capitalized words.

- Instance is all upper cased.
- Instance has special character (x).
- Instance has letters uppercase.
- Instance has letters lowercase.
- Instance has digits.
- Instance is a digit.

For every instance the feature vector is stored, as well as the target value. An example of the average output feature for a column (not an instance) is presented in figure 8.

Figure 8: Feature: Syntax Feature



Average syntax feature model vector of a column

3.4.3 Topic Model

The previously presented features are more syntax based, however, lots of schemas contain only textual data. In these cases a word-embedding model would be more useful. Therefore we already build in a feature which simply creates a corpus out of the instances found in the column. These can later be used by natural language models or word embedding models.

The Word2Vec language model that has already been implemented in ARPSAS uses a uni-gram model, this means that every single word is added to the model as opposed to a bi-gram model where only combinations of two words are added. The Word2Vec model computes the feature vector for every word in a corpus in 200 dimensions. Preliminary experiments determined that these parameters (a uni-gram model with 200 dimensions) would result in the highest accuracy.

When training the matchers, the corpus for every column in a class is ran through the model. The feature vectors that are outputted by the model are used as training data for a classifier. Upon classification, a corpus is created from a source-schema column. The source corpus is ran through the Word2Vec model, and the total average feature vector is inserted into the classifier for classification.

3.4.4 Number Feature

Lastly, lots of databases contain columns which are solely populated by numbers. These can be hard to separate since their ranges could heavily overlap. Already implemented in ARPSAS is a feature class that builds a feature vector based upon the average number in a column, again the character distribution, the average length of the numbers and wether or not it is an integer or a float.

3.4.5 Synopsis

The features already implemented in the framework are meant to be used as a basis or step-up for a user. They have been created to classify the data from Company.Info, and a user should define their own based upon their data. In the feature builders the user already has the power to create the setup for a hybrid matcher, by combining multiple features into one single feature vector (as was done in the fingerprint feature, which is a hybrid of a character distribution and an n-gram model). Feature builders should store their pre-processed data so it can be used later by the matchers. The implemented feature vectors and their targets are stored in lists as is shown in the following example:

```
features = [
    feature_vector_1,
    feature_vector_2,
    ..
    feature_vector_N
```

```
]
targets = [class_1, class_1, class_2... class_N]

where:
feature_vector_N = [datapoint_1, datapoint_2, .., datapoint_N]
```

3.5 Matchers

After the features and the targets are computed, they can be utilized by the matchers to classify data. Matchers can consist of either rules and constraints (currently not implemented), or machine learning components. Data from multiple features can be combined by a single matcher or multiple smaller components can classify the data in order to make a prediction. This means these classes can act as a hybrid and as a composite matcher.

Matchers can perform differently on different datasets. It would be useful for a user to test their implementation and configuration of a matcher on the initial training data in order to get an indication on how well the matcher can perform on test data. This is why each matcher component can be tested by a test class which implements a k-fold test. During a k-fold test, all the target-schema column data is randomly divided into k equal sized sub-samples. Of the k sub-samples, a single sub-sample is used as the validation data for testing the model, and the remaining sub-samples are used as training data. This cross-validation process is repeated k times, once for each validation fold [5]. Such a test can be used to determine if a specific matcher can recognize the given classes with high accuracy.

For each feature class there is also an implemented matcher class in the framework. If a user wants to utilize his own matcher class within the entire framework (as opposed to use a single matcher as a stand alone), the user should follow the formats and implementations stated by the framework. Each matcher class should be able to classify instances, entire columns and preferably also provide a confidence for the prediction. These requirements were put on the matcher classes in order for them to all function within the matching pipeline. As long as these requirements are satisfied, any matching algorithm can be implemented. The default classifier (a Support Vector Machine) for each matcher can be overwritten based on the users needs. The machine learning components inside matchers should be trained upon initiation.

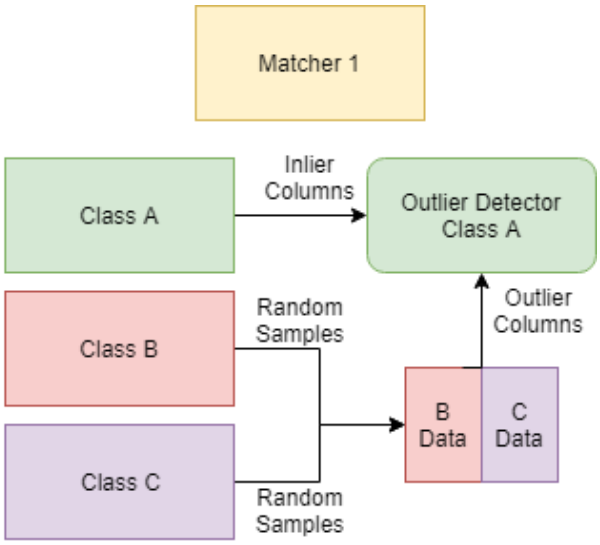
3.5.1 Outlier Detection

Another problem in schema matching is the detection of outliers. Outliers are in this case columns that contain data that is not part of the training set, and therefore does not have to be mapped. If outliers are not recognized but present, they will always be mapped incorrectly.

Each class in the framework comes with its own outlier detector. After a matcher classifies data, a specifically trained binary classifier is used to classify the same data. This binary classifier is trained to recognize if data for that particular class is an inlier or an outlier.

Outlier data is generated by randomly combining the computed feature vectors from other target-schema classes. The creation of the outlier detectors per class is illustrated in figure 9.

Figure 9: Outlier Detection



For 'Matcher 1', which has been trained to classify classes A, B and C, the outlier detector is trained for class A by using the data from class A as input for inliers and randomly sampled data from the other classes as outlier data (the 'not A' class).

Upon classification, the outlier detectors can optionally be called. Users of ARPSAS are free to define and test their own outlier detection algorithms. This particular way of detecting

outliers (by generating outlier data and training a binary classifier for each class) was chosen because there was no actual outlier data available at the time that the system was produced, and initial tests showed that this way of detecting outliers achieved the highest accuracy. The Scikit-learn OneClassSVM classifier was also tested but was outperformed by this simple binary classification.

3.6 Building Pipelines

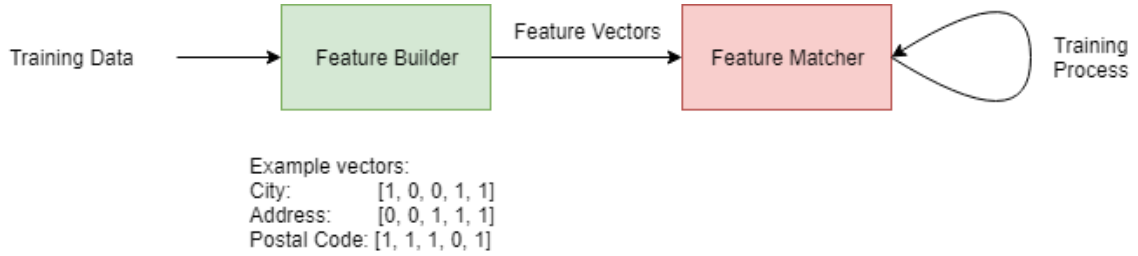
3.6.1 Column Classification Configurator

Now that feature and matcher classes have been explained we can introduce the hierarchical classification system that uses these classes and allows a user to configure their schema-matching pipeline. It is useful to be able to reproduce such pipeline when iteratively configuring and optimizing it. To accommodate this, the configuration of the entire classification pipeline is stored in the *Column Classification Configurator* (CCC) object. As depicted in figure 3 and 5, a user first has to add feature builders to the configuration. After these feature builders are added, a user can add matcher classes which utilize these feature builders (implementation details can be found in the framework). This was done in order to allow a user to easily experiment with different features and matchers. We will first recap on how the implementation of a single matcher works before we discuss how an entire pipeline of matchers can be created.

3.6.2 Current Setup

Upon initiation of the framework there is the training phase in which the features which were added to the CCC collect their data. Data is utilized to train machine learning components which will be utilized to classify columns based on their feature vectors. The overall flow of the training phase can be seen in figure 10. The input schema in this simple example contains columns with data of addresses, cities and postal codes.

Figure 10: Flow of Training-Phase



After the matching component is trained, this tiny set up can already be used to classify columns as is depicted in figure 11.

Figure 11: Flow of Matching-Phase

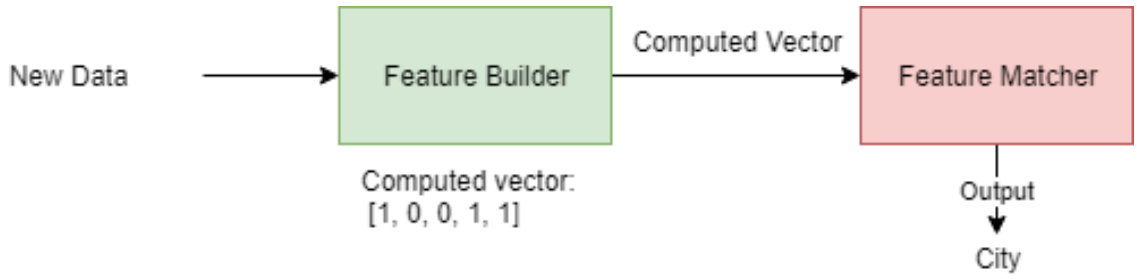
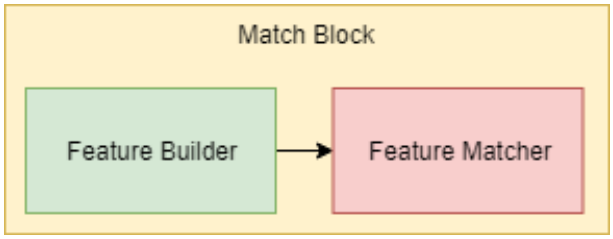


Figure 11 only shows the result for a single column in the schema, but all columns should obviously be ran through the feature builder and matcher in order to create a mapping for a schema as is depicted in figure 6.

3.6.3 Pipelines

When two columns contain syntactical similar information, it is possible that confusion can occur between them during the matching-phase. This can be solved by using a different feature class which could possibly aid in correctly distinguishing the two columns upon classification. Because of this, a user should be able to specify within the framework that a specified matcher should be called when a certain class is predicted. ARPSAS allows a user to define pipelines, or '*match trees*'. The concept is explained using figures 12, 13 and 14.

Figure 12: Defining a Match Block

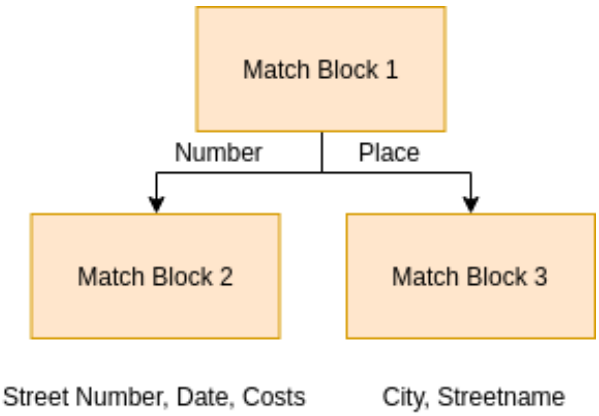


The combination of a feature builder and feature matcher is called a '*match block*'

When a matcher is added to the CCC, it is also inserted into the match tree according to the specification of the user. A user can define if data should be send to a different matcher depending on the classification result of the previous matcher. For example, the following pseudo code could create the pipeline depicted in figure 13:

```
ccc = Column_Classification_Configurator()
ccc.add_matcher('match_block_1', 'Fingerprint_Matcher') # main matcher
ccc.add_matcher('match_block_2', 'Number_Matcher', ('match_block_1', '
                                    number'))
ccc.add_matcher('match_block_3', 'Syntax_Matcher', ('match_block_1', '
                                    place'))
```

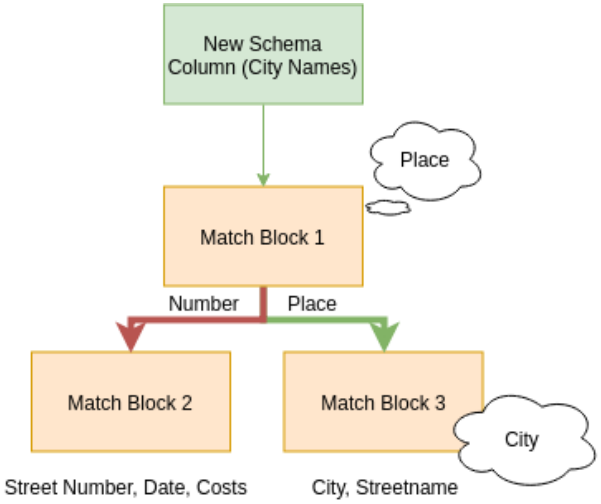
Figure 13: Creating a Match Tree



Matchers are trained upon initialization, before they are added to the entire tree. With the shown code, we first tell the CCC to create an initial matcher which is a Fingerprint matcher. We then add a Number matcher. This matcher is called when the main matcher predicts that the inserted data belongs to the 'number' class. If this is the case, than the data is send to this

matcher for further classification. In figure 13, match block 1 pipes column data to match block 2 in case the column is classified as the 'Number' class. Data is piped to match block 3 if the data is classified as part of the 'Place' class. This setup can be used to classify column data more specifically following the tree-structure. An example classification using this tree can be seen in figure 14.

Figure 14: Classifying a Column Using the Match Tree



The example shows that the data is piped from match block 1 to match block 3. This block is specifically trained to separate city names from street names, and finalizes the classification process as a tree leaf. A pipeline should be fitted specifically for a single global target schema. The outlier detectors (discussed in section 3.5.1) can be called in the tree leafs upon classification time to check if a column should be mapped or if it is an outlier.

The concept of classifying data in such a tree structure has been inspired by the hierarchical classification field [14]. Hierarchical classification has not yet been applied in a schema matching context, but could reduce the amount of errors as opposed to a single classifier [14][15].

3.7 Loading the Pipeline

The match tree which the user can create and customize using the CCC can be loaded into the schema matcher component as is depicted in figure 4. This schema matching component is not part of the architecture depicted in figure 3. The schema matching component reads a source schema and calls the methods from the match tree in order to classify column data

and create a mapping. By introducing this schema-matching class we separate the mapping of actual schemas from the task of building, customizing and using the pipeline. This is done so the match tree can also be used outside of a schema matching context even though this is out of scope for this project.

3.8 Implementation

The system is completely written, and can be extended, in Python. This was done because during this research project we aimed to follow a learning based approach within the framework, and Python is the most supported language for this purpose [11]. It was also done because the creators of the framework were most experienced with Python. ARPSAS does not contain an interface, and any changes or configurations have to be written in Python code.

4 EXPERIMENTS

ARPSAS is an environment which allows users to create and customize hierarchical classification pipelines which can be used to optimize a schema matching classification result. Now that the ARPSAS framework and its possibilities are explained we can introduce experiments to test the setup and the matchers. There are lots of aspects that could be tested for given a dataset. We want to answer the following main-question:

How can an effective semi-automated schema matching pipeline be created and customized for a given dataset?

Therefore we aim our experiments at performance of the previously discussed features and matchers. We hope to find rough guidelines that users can follow in order to optimize their own schema-matching pipeline.

Since outlier detection is optional and addresses a different kind of matching problem (namely the problem of excluding columns from the mapping), the experiments will be ran twice, once using the implemented outlier detection, and once not using the implemented outlier detection.

The experiments that will be performed are designed to test if and how a schema matching pipeline can be optimized for a given dataset. To test if a hierarchical classification pipeline can improve the matching result we will first test the performance of each individual matching component. After this, we will test if we can optimize the classification result within these single components already by tweaking the configuration for these components (the actual parameters that will be tweaked are discussed in each experiment subsection). We will then show how a hierarchical classification pipeline can be created and fitted to a dataset. Finally, we will test if this pipeline improves the matching result by comparing it to the initial experiment.

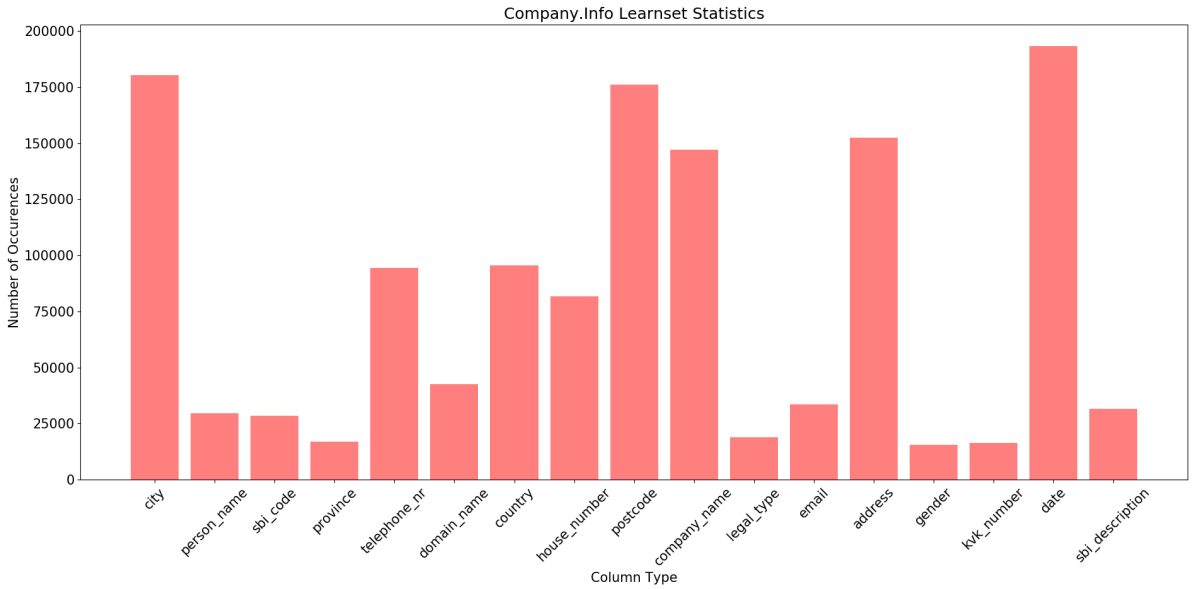
4.1 Datasets

4.1.1 Company.Info Dataset

The first dataset that will be used is the dataset created from Company.Info csv data. This set has been labeled and divided randomly into two sets, a learn- and test-set. The learnset has been processed to be easily read by the data collector (section 3.3), the contents per class

within the dataset are presented in figure 15. A class is a collection of instances in the data that all belong to the same category within the target schema.

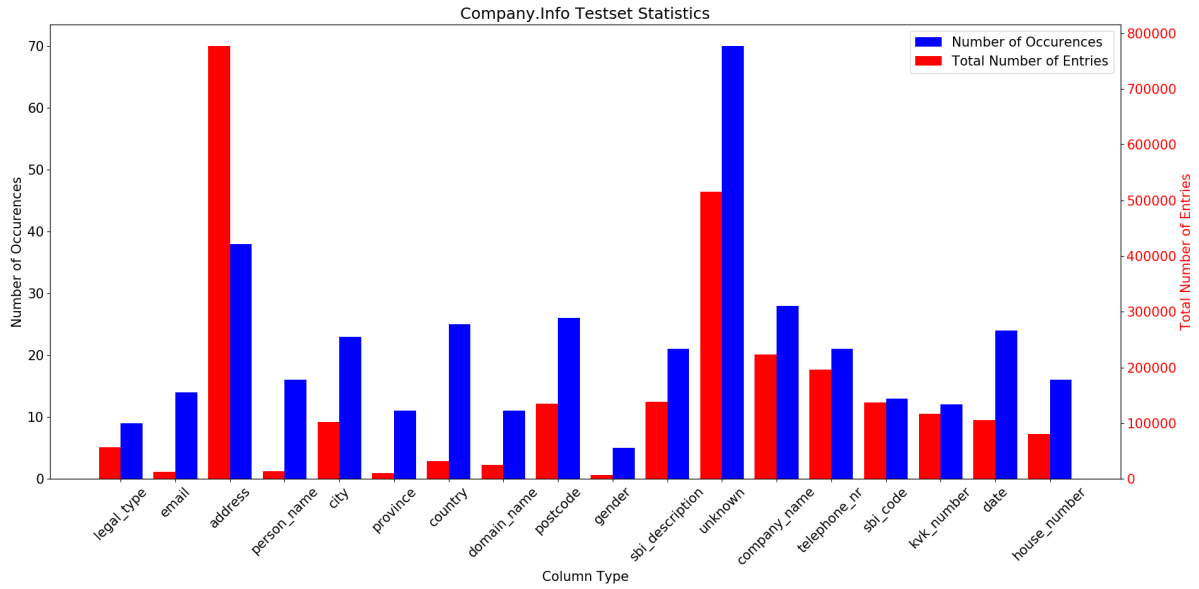
Figure 15: Company Info Learnset



Number of instances per class in the Company.Info learnset.

This dataset contains only inliers and is used for training the system only. The training and testing data is separated so we can experiment with the number of training columns independently. The testset contains both inliers and outliers. The contents of the dataset per class are presented in figure 16.

Figure 16: Company Info Testset



Total number of instances (instances) and occurrences (columns) in the Company.Info testset.

The class names accurately represent the type of data extracted from the csv columns. The data is not distributed very evenly among the classes, both in the learn and testset. This does not matter for the learnset since we can simulate the number columns per class and specify the amount of instances we want each column to posses. This does matter for the testset. If a matcher is good at classifying addresses, in comparison to provinces, it will likely have a much higher score overall since addresses occur way more often. Outliers also heavily influence the results since they are the most occurring class. If the outlier detectors introduce a lot of confusion the scores will drop heavily.

4.2 CKAN CERIF

The second dataset that will be used is the CKAN-CERIF dataset. CKAN is an open-source data management system (DMS) for powering data hubs and data portals. This data catalogue system is often used by public institutions to share their data with the general public [3]. The Common European Research Information Format (CERIF) is a data model that allows for a meta-data representation of research entities. Both models are not in csv format. The CKAN dataset consists of xml files while the CERIF dataset consists of rdf files. CKAN model data can be partially mapped to CERIF data. The goal of the experiments performed with these

datasets are to research if ARPSAS can also be applied to non-column type data structures. It is useful to know this because data integration also often occurs for these (non-column) types of data.

The mapping of the CKAN data to the CERIF model has already been done manually. Both datasets and the mapping were provided by the University of Amsterdam. The goal is to train a schema-matching pipeline for the CERIF data, so that the CKAN data in the future can automatically be mapped to the CERIF format. There are two main challenges when mapping these datasets with ARPSAS:

1. ARPSAS was not designed to work with the xml and rdf format.
2. A lot of data in the CERIF model is automatically generated from the CKAN data. Generating data by combining multiple columns is something ARPSAS can not do.

Both problems are solved during a pre-processing step that has not been built-in in the system. Both formats can be converted to json. The xml files are converted using the Parker convention. The Parker convention[10] ignores xml-attributes and simply recreates the xml structure but in json. This conversion was chosen because attributes were not present in the CKAN-xml data. The rdf-files are converted by unpacking the rdf and recursively looping through the rdf-tree, starting at the root, and adding the instances to the json dictionary.

ARPSAS can also not work with the json format, but this model can more easily be manipulated. All values in the json dictionaries were removed from the tree-like structures and placed in a column structure by using the path to the tree-leafs as a new column name. This conversion is presented in the following example:

```
json_data =
{
    main_key: {
        key_1 : value_1,
        key_2 : value_2
    }
    second_key: {
        key_1 : value_3,
        key_2 : value_4
    }
}

column_structure = [
(main_key_key_1 , value_1),
```

```
(main_key_key_2 , value_2),
(second_key_key_1 , value_3),
(second_key_key_1 , value_4)
]
```

By doing this for all the xml and rdf files and accumulating values with the same column name, we do end up with a column structure. This can be used by ARPSAS. Eliminating the tree structure does remove the meaning of each key in the tree. With this experimental setup we'd like to see if the tree-type data is still classifiable based on the tree-leafs, and with that testing if ARPSAS can be applied to non-column database data.

The CERIF data is the target schema, and will therefore be used to train a pipeline. The problem however is that a lot of values in the CERIF model are generated from the CKAN data. The generation of data happens during the mapping process and is done by combining multiple elements from the CKAN data into a single CERIF element. The generation of such values is something that ARPSAS can not do and is out of scope for this project.

To still create a mapping that could be used during the experiments, all column data from both datasets was compared. If two columns from both datasets contained largely the same information, we consider them to be a match, and we give the CKAN column the appropriate label (column name) from the CERIF column. If a column from the CKAN data did not match with any CERIF column we consider it to be an outlier.

The mapping and labeling of the CKAN data was done by using all the data, but for the experiments all data is separated before any of the previously discussed conversions (flattening of the tree-structures) were performed into a learnset and a testset.

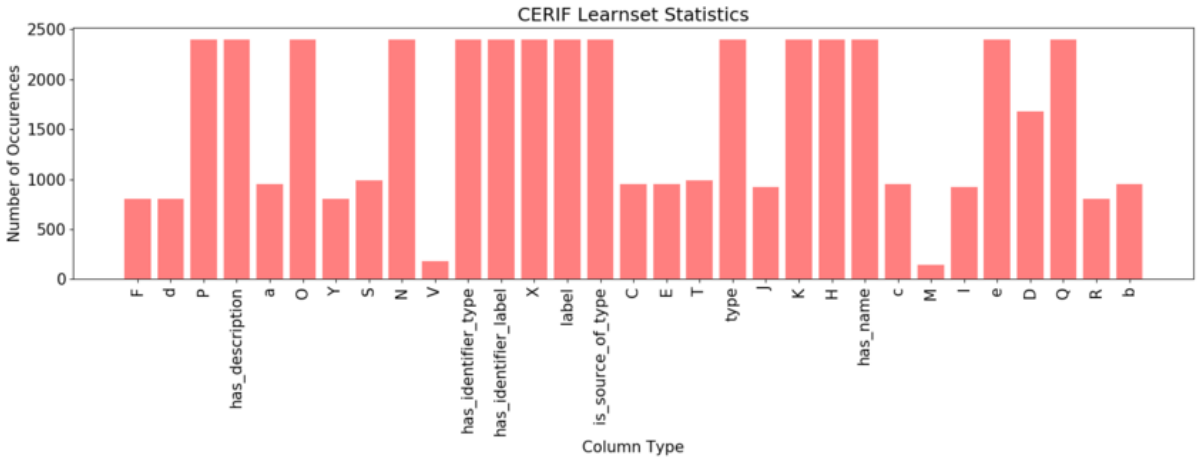
Since the transformation of a tree-structure to a column structure by using the tree-path as a column name often ends up with a abnormally long column name, it was decided that for the experiments and results large column labels will be abstracted using the following mapping:

```
{   'A': 'is_source_of_has_classification_has_term',
    'B': 'is_destination_of_has_source_is_source_
            of_has_destination_has_URI',
    'C': 'is_destination_of_has_source_
            is_source_of_has_destination_type',
    'D': 'is_source_of_has_destination_type',
    'E': 'is_destination_of_has_source_is_source_of_has_endDate',
    'F': 'has_identifier_is_source_of_has_endDate',
```

```
    'H': 'has_identifier_has_id_value',
    'I': 'is_destination_of_has_source_has_identifier_has_URI',
    'J': 'is_destination_of_has_source_has_identifier_type',
    'K': 'is_destination_of_type',
    'M': 'is_destination_of_has_source_is_
            source_of_has_destination_has_name',
    'N': 'is_destination_of_has_source_type',
    'O': 'is_destination_of_has_classification_type',
    'P': 'has_identifier_has_URI',
    'Q': 'is_source_of_has_classification_type',
    'R': 'has_identifier_is_source_of_has_classification_type',
    'S': 'is_destination_of_has_endDate',
    'T': 'is_destination_of_has_startDate',
    'V': 'is_destination_of_has_source_has_identifier_has_id_value',
    'X': 'is_source_of_has_endDate',
    'Y': 'has_identifier_is_source_of_has_startDate',
    'a': 'is_destination_of_has_source_
            is_source_of_has_classification_type',
    'b': 'is_destination_of_has_source_is_source_of_type',
    'c': 'is_destination_of_has_source_is_source_of_has_startDate',
    'd': 'has_identifier_is_source_of_type',
    'e': 'is_source_of_has_startDate',
    'has_description': 'has_description',
    'has_identifier_label': 'has_identifier_label',
    'has_identifier_type': 'has_identifier_type',
    'has_name': 'has_name',
    'is_source_of_type': 'is_source_of_type',
    'label': 'label',
    'type': 'type',
    'unknown': 'unknown'
}
```

The CERIF learnset contains the following data:

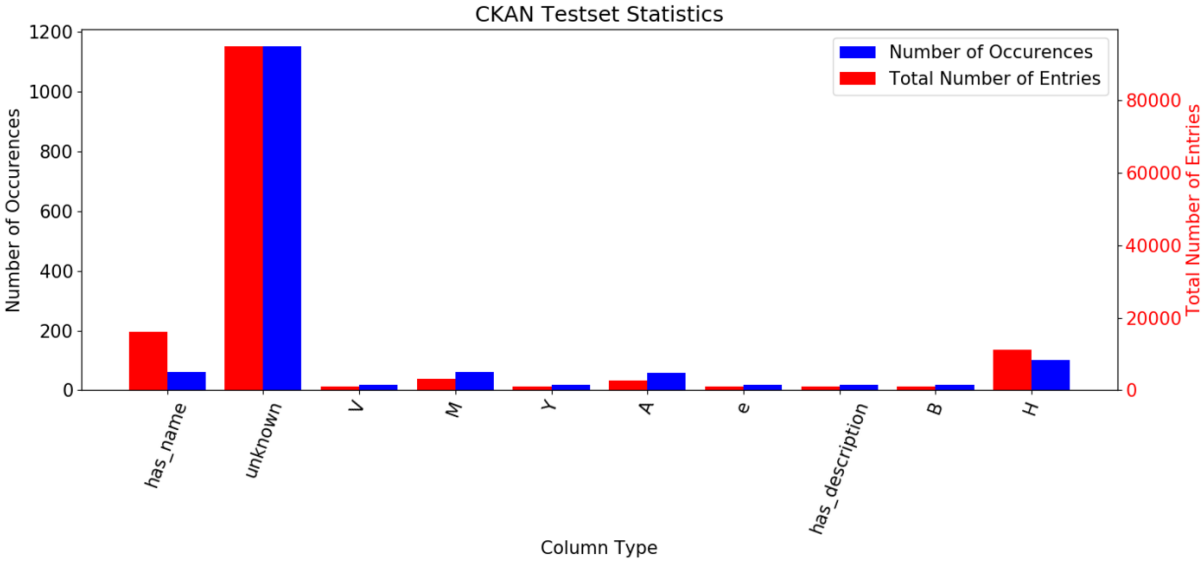
Figure 17: CERIF Learnset



Number of instances per class in the CERIF learnset.

Each file in the CKAN and CERIF dataset contains a single model. This model in turn contains the meta-data for one single entity, which is stored in a tree structure. For each csv file that was created from the CKAN or CERIF data, 50 models were used. During this conversion, columns that ended up with less than 20 instances were removed. These parameters can be set and investigated but this is out of scope for this project. The CKAN testset contains the data presented in figure 18.

Figure 18: CKAN Testset



Number of instances per class in the CERIF learnset.

As you can see, there are 9 classes in the CKAN set that could be mapped directly to the CERIF dataset. The majority of the data consists of outliers (figure 18). Outliers therefore affect the results heavily.

4.3 Metrics

Before we can do any experiments we have to define the metrics that we are going to use. For every experiment we can test two aspects of the framework:

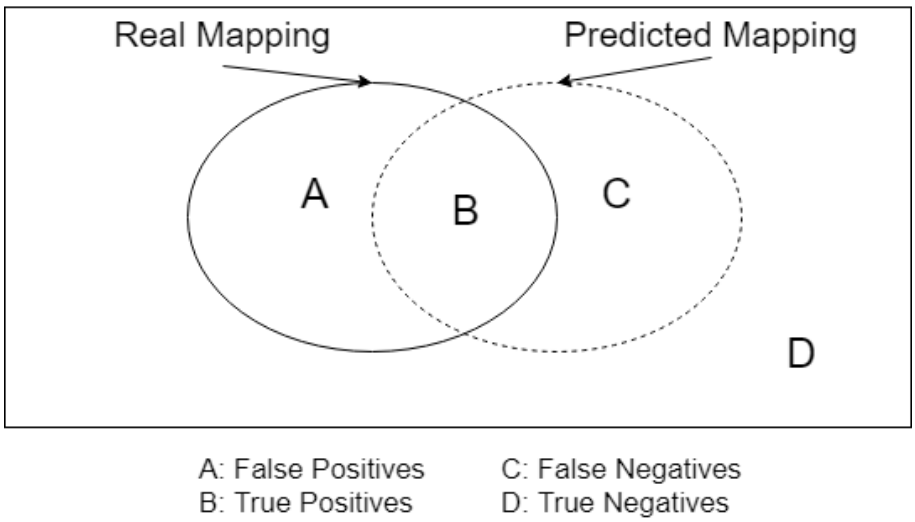
1. How well are columns classified (and therefore mapped) when we can assume they are inliers?
2. How well are columns classified when we have to deal with outliers?

These two aspects are not only tested to measure the performance of the matchers, but also to measure the influence of outliers on the result. This is important for two reasons. First of all, by testing these two aspects independently we get an indication of how well the outlier detection is working. Secondly, outliers are by far the most occurring class in both datasets (figures 16 and 18), therefore, if the outlier detection does not perform very well, the results will be heavily influenced.

As in many other machine learning applications, the metrics precision, recall, F-measure and accuracy will be used to validate the performance of the pipeline[6][16]. As illustrated in figures 19 and 20, the metrics are computed using the following outcome variables:

- False Negatives (A): Inlier is detected as an outlier or column is classified incorrectly, and therefore mapped incorrectly.
- True Positives (B): Column is mapped to the correct outcome.
- False Positives (C): Column is classified incorrectly or seen as inlier while it is an outlier.
- True Negatives (D): Column is not part of the dataset and the outlier is correctly detected.

Figure 19: Illustration of Metrics 1



Definition of false positives, false negatives, true negatives and true positives [6].

Figure 20: Illustration of Metrics 2

		Actual	
		Positive	Negative
Predicted	Positive	True Positive	False Positive
	Negative	False Negative	True Negative

Definition of false positives, false negatives, true negatives and true positives.

Precision, recall, the F-measure and accuracy are defined as the following (using figure 19 as a reference):

- $Precision = \frac{B}{B+C}$ This measure reflects the portion of correct matches among all of the found matches.
- $Recall = \frac{B}{A+B}$ This measure reflects the portion of the correct matches among all of the matches that could have been found.
- $F-Measure(\alpha) = \frac{Precision*Recall}{(1-\alpha)*Precision+\alpha*Recall}$

- $Accuracy = \frac{B+D}{A+B+D}$ Reflects the percentage of correctly classified data.

In the ideal situation there are no false positives or false negatives and every column is mapped correctly: $Precision = Recall = Accuracy = 1$. Solely relying on the precision and recall metrics alone is not a good idea, since both metrics can be manipulated to be as high as possible at the cost of one another. We would achieve very high precision if we only return a match that we are 100% sure of, but the recall would be low. If we return as many matches as possible for any column (map the column to multiple other columns), then we would achieve very high recall but very low precision. The F-Measure combines both metrics and allows a user to shift the focus of the measure using the α parameter. If $\alpha \to 1$, more importance is attached to precision. If $\alpha \to 0$, more importance is attached to recall. For our experiments we will use the harmonic mean between precision and recall ($\alpha = 0.5$):

- $F - Measure = 2 * \frac{Precision * Recall}{Precision + Recall}$

The precision and F-measure metrics can not be used when we do not have outliers in our dataset, therefore we will only use the accuracy metric when we are validating the pipeline without using outliers.

4.4 Experiment 1: Baseline

4.4.1 Goal

Before experiments with the entire classification pipeline will be ran it is necessary to have a baseline. We will measure the performance of each each individual matcher, so we can later determine if building and optimizing a hierarchical classification pipeline improves the matching result. It is also useful to know if a pipeline is even needed. If the stand-alone matchers (section 3.5) are already able to classify the entire dataset correctly than there is no reason to research this topic any further since there can be no more improvement. If this is the case than the experiments have to be performed on other datasets.

4.4.2 Validation and Method

The matching performance of any schema-matching algorithm can be defined as the portion of data (in this case columns) that is correctly mapped from the source schema to the target

schema. We want to know the performance of the matchers across both datasets, so we can measure for improvement in a later experiment. The metrics discussed in section 4.3 will be used to measure the actual performance.

This experiment has to be performed on both datasets in order to have a baseline measurement against which we can measure in a later experiment. The experiment will be performed twice for all matchers, first without using the outlier detection, and then with the outlier detection. We do this in order to measure the influence of outliers on the matching result. The resulting confusion matrices of this experiment will also be used in a later experiment to see if we can reduce this confusion among classes.

We expect the Fingerprint matcher to perform the best when classifying all classes, since initial tests showed that it was very accurate when classifying the initial dataset as a standalone matcher. We expect the Word2Vec matcher to perform the worst since lots of columns do not contain repeated segments of text. We should mention that we expect the matchers to perform differently on the datasets since each matcher is designed to classify different sorts of data.

It is expected that the accuracy using the outlier detection will be lower in comparison to the tests with only inliers because there are less classes to classify and therefore the chance of predicting the correct column type is higher. The number-matcher is excluded from the tests since there are only 3 numerical data types present in the dataset.

4.5 Experiment 2: Number of Columns

4.5.1 Goal

With the data collector, discussed in section 3.3, it is possible to simulate the number of columns that are used to train the matchers. When speaking of 'simulating the number of columns', we mean that all the data of a single class in the training set is stacked together into one giant column, and that all the data in this column is then randomly divided over n 'simulated' columns. This experiment is used to determine the optimal number of columns that is used to train each matcher (section 3.5), if there even is an optimum. We define an optimum in this context as a peak in the scoring results.

There is a finite set of data, and data needs to be divided among the columns during the training phase (section 3.6.2). When more columns are simulated, there are less instances

in each column. It is important to know if an optimum in the results caused by the number of simulated columns can be found. By testing this variable, we could possibly exclude this factor from further experiments and strengthen the validity. If an optimum can be found we can use it to improve the scoring of the matchers during further experiments.

4.5.2 Validation and Method

In order to find out if there is an optimal number of simulated columns to train a matcher, we will train every matcher with an increasing number of simulated columns and then test them against the complete testset. All instances in the entire dataset will be used. A peak in performance (measured by the metrics defined in section 4.3) will mean that there is an optimum number of columns.

The instance based matcher classes (Syntax Feature Model, section 3.4.2) are not dependent on the number of columns so therefore they will be excluded from this test. We expect that using a lot of columns results in lower scores, since few instances will be left in each column. This might cause a computed feature vector to be an inaccurate representation for a column in its class. We also expect that using very little columns results in a lower test score. If there are very little columns, there is very little training data for the matchers, which in turn might cause an underfit.

4.6 Experiment 3: Number of Instances per Column

4.6.1 Goal

The data collector (section 3.3) allows us to specify the amount of instances we'd like to have in each simulated column. It is important to know how many instances have to be used in order to get a reliable result. Having a lot of instances in a column might cause an overfit, and having too little might cause an underfit. By testing this variable, we could possibly exclude this factor from further experiments and strengthen the validity. If an optimum can be found we can use it to improve the scoring of the matchers during further experiments.

4.6.2 Validation and Method

To test the influence of the number of instances in each simulated column, we will train the matchers with a set number of simulated columns and increase the number of instances in these column for each test. A peak in the performance (measured by the metrics defined in section 4.3) will mean that there is a certain number of instances per column needed in order for a feature vector (which is computed from this column) to be a reliable representation for a class.

It is expected that instance based matchers perform better when as many instances as possible are used. We also expect that for the matchers that focus on column-wide features that it is better to use as many instances as possible since this might cause the computed feature vectors to be more reliable (a more accurate representation for the columns in that class).

4.7 Experiment 4: The Influence of Sub-matchers

4.7.1 Goal

As discussed in section 3.6.3, ARPSAS can be used to build a pipeline of match blocks in order to classify a column hierarchically. The goal of this experiment is to determine if hierarchical classification can improve the matching result. We will research how such a pipeline of match blocks can be created, and how it influences the results as compared to the initial baseline experiment.

4.7.2 Validation and Method

For each dataset, a pipeline will iteratively be build. Depending on the confusion matrices from experiment 1, sub-matchers will be introduced to the pipeline in steps. The influence of these sub-matchers on the matching result will be measured during each step by classifying the entire testset. The match scores will be used as a measurement of performance in order to find out if the introduction of the new sub-matchers actually increases or decreases the matching result. The resulting confusion matrices from each step will be compared to the confusion matrices from the previous step in order to find out how the addition of sub-matchers has influenced the confusion among the classes in the dataset.

We expect that the performance of the matchers will increase during each iteration. We expect the complete tree of matchers to have a better performance than a single match block in all cases since it reduces the amount of classes per classifier. By using sub-matchers it is also possible to pipe classified data to data-specific match-blocks, which should also increase accuracy.

4.8 Experiment 5: Additional Matcher Cost

4.8.1 Goal

Depending on the needs of the user, it is useful to know how much extra time is needed for training the pipeline and classifying a column when adding more match blocks. Therefore we want to measure how much additional time is needed for classification for each added match block. Classification time is important when you are dealing with systems that have to integrate giant heaps of data quickly. The training time of the pipeline could be important when you want to retrain the system often depending on the classification result. The increase in training and classification time will indicate how ARPSAS scales.

4.8.2 Validation and Method

To test how the size of the pipeline influences the training and classification time, we will incrementally add sub-matchers of the same type (all matchers can be viewed in section 3.5) to a linear pipeline. During each increment the training time will be measured, and columns will be classified.

We expect that for each additional match block the classification time will increase linearly because the same operations (loading the data, computing the feature vector, and classifying) are repeated for each match block (section 3.6.3). We expect the training time to increase linearly as well for the same reason.

5 RESULTS

Each experiment is ran twice:

1. Once without using the outlier detection, where only the accuracy metric is used (this is done because there are no true negatives in this dataset, see section 4.3).
2. Once using the outlier detection, where the precision, recall, F-measure and accuracy metrics are used.

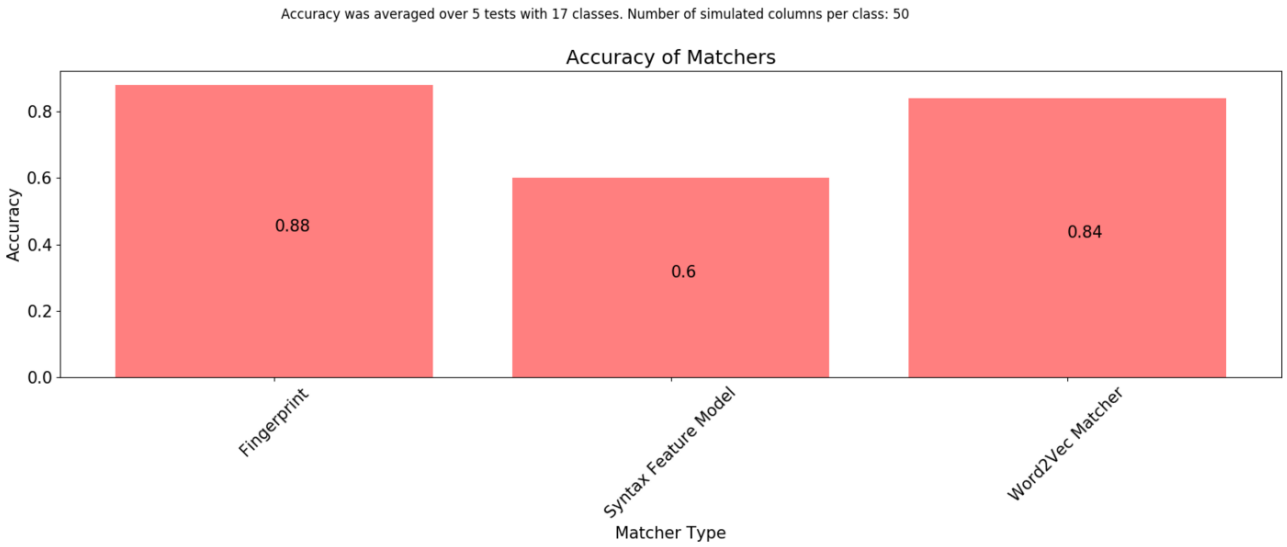
The results of the experiments without outlier detection will be referred to as sub-experiment 1, and the results of the experiments executed with outlier detection will be referred to as sub-experiment 2. In short; experiment 1.1 is equal to experiment 1 without using outlier detection.

5.1 Experiment 1: Baseline

The result of this experiment will be used as a baseline for further experiments with both datasets. The results of the inlier and outlier experiments are shown in their respective subsections. The syntax feature model matcher was ran with 5000 instances of data per class with the Company.Info data. This limitation was applied due to time constraints. All other tests were ran with as much data as possible. The number of simulated columns was the same for all classes per dataset. The number of simulated columns was reduced for the CKAN-CERIF dataset since it had less instances overall. Each test was ran 5 times and the average results are presented.

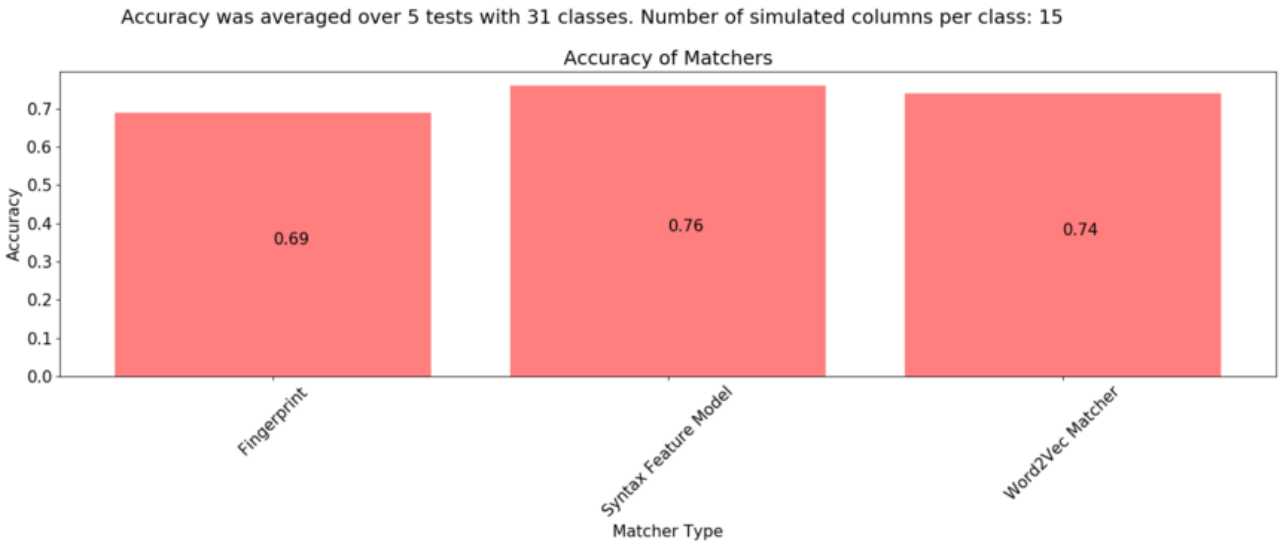
5.1.1 Experiment 1.1

Figure 21: Result of baseline experiment 1.1



Tested on the Company.Info dataset

Figure 22: Result of baseline experiment 1.1



Tested on the CKAN-CERIF dataset

The results of this experiment are very interesting. For each dataset, the matchers already perform differently. For the Company.Info dataset the fingerprint matcher performs the

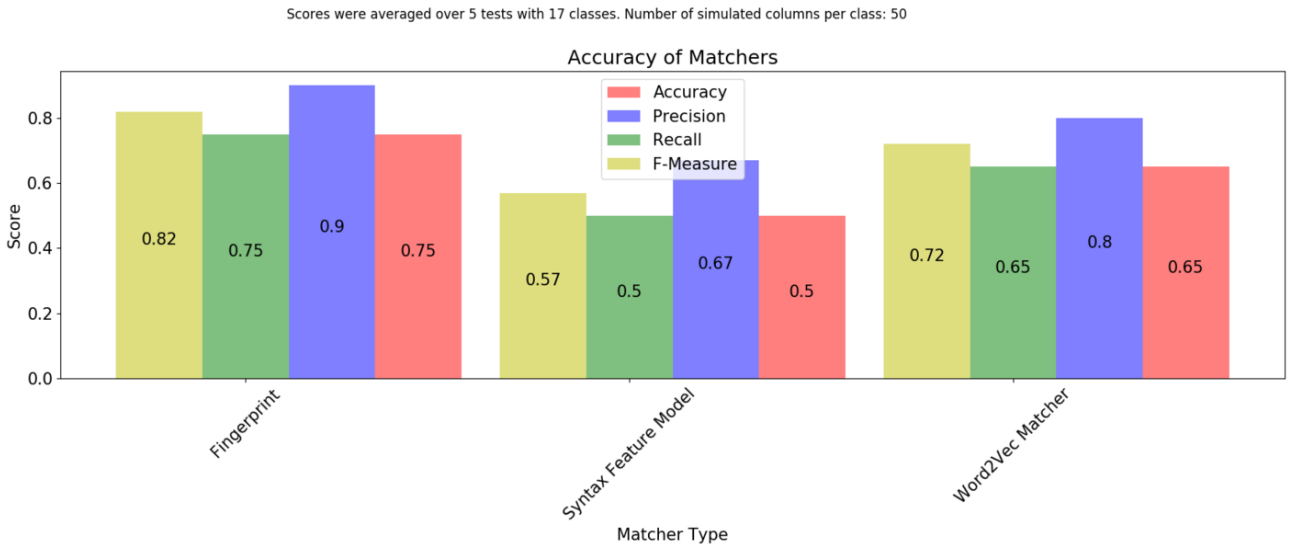
best as expected, but the opposite is true for the CKAN-CERIF dataset. Here the fingerprint matcher performs the worst, and the Syntax Feature Model outperforms all matchers.

This result could be caused by multiple factors. First of all there is the imbalance in the test-sets to consider. If matchers perform well on the most occurring classes, accuracy will of course be higher. A good example here is the CKAN-CERIF datasets. The *has_name* class, and class *H* are by far the most occurring in the testset, and the Syntax Feature Model matcher performs very well on distinguishing these exact two classes from the others as can be seen in the confusion matrix of figure 41 of the appendix. The fingerprint matcher (figure 40 of the appendix) performs well on the class *V* but since this class is less occurring, the confusion in the more prominent classes significantly affects the result.

The imbalance in the learnsets could also play a role in the classification process, but this factor will be tested in further experiments with the number of columns to be used, and the number of instances to be used per column.

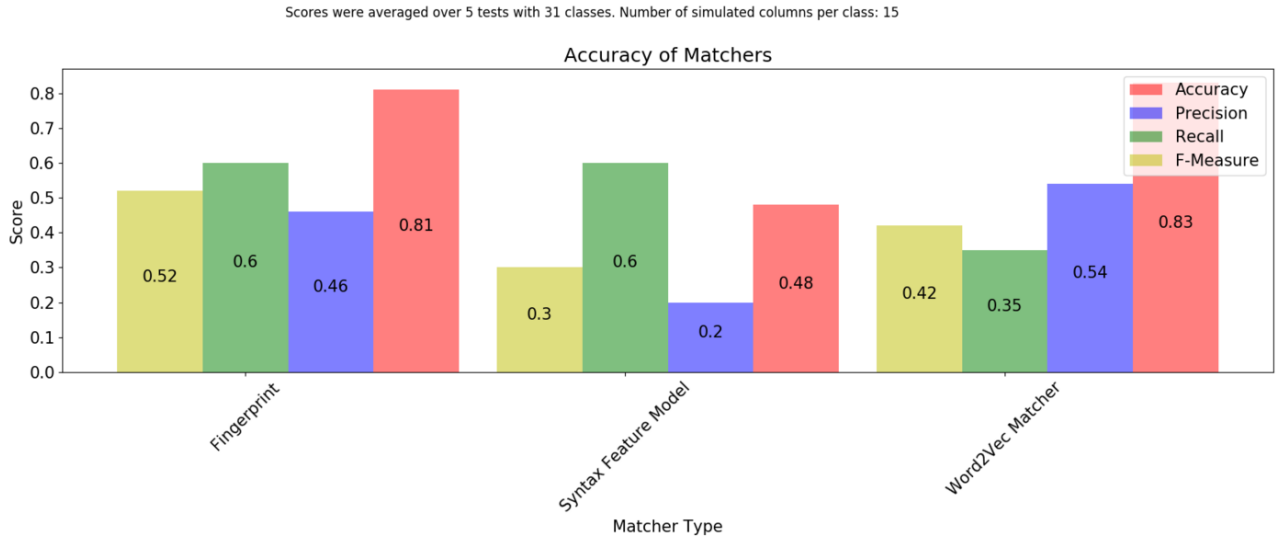
5.1.2 Experiment 1.2

Figure 23: Result of baseline experiment 1.2



Tested on the Company.Info dataset

Figure 24: Result of baseline experiment 1.2



Tested on the CKAN-CERIF dataset

The results of experiment 1.2 are again interesting. When looking at the recall scores (the portion of correct matches among all of the matches that could have been found), we can definitely see that there is a lot of confusion between the inliers and outliers. This is also supported by all of the confusion matrices in section 9.2 of the appendix, inliers are often detected as outliers but not the other way around. This provides insight in the performance of the outlier detector.

Accuracy drops when outliers have to be detected for the Company.Info dataset but the opposite is true for the CKAN-CERIF dataset for the fingerprint matcher. This is definitely because of the imbalance in the testset. Since outliers are often identified correctly, and outliers are the most occurring class in the testset, accuracy will rise because the portion of correctly detection outliers simply outweighs the portion of incorrectly classified inliers. The precision heavily dropped in the CKAN-CERIF experiment when the outlier detection was turned on, and this is the result of the high confusion between outliers and inliers.

Both experiment 1.1 and 1.2 do however show that certain matchers are more capable of classifying specific classes, supporting the hypothesis (made in section 3.6.3) that piping data to specific matchers could reduce confusion within a classification pipeline. Since not all data in both datasets is classified correctly, the experiments show that an improvement can be made to the classification process.

5.2 Experiment 2: Number of Columns

This experiment was only performed on the Company.Info dataset since it was larger and thus allowed for a wider testing range. Each test was ran 3 times and the average results are presented.

Figure 25: Result of Experiment 2.1

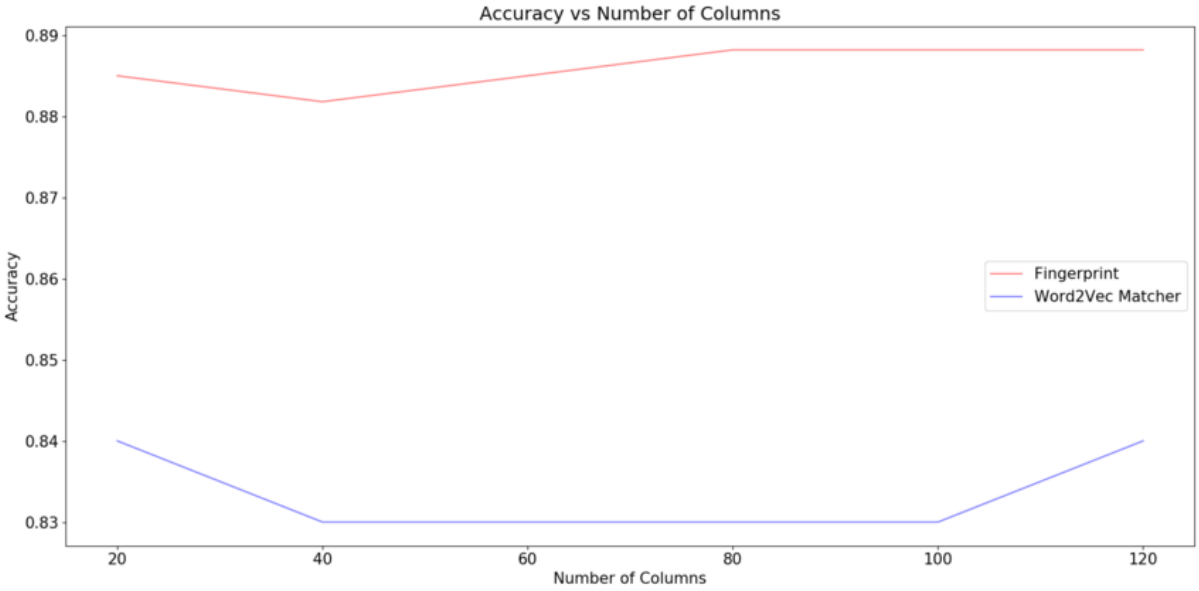


Figure 26: Result of Experiment 2.2

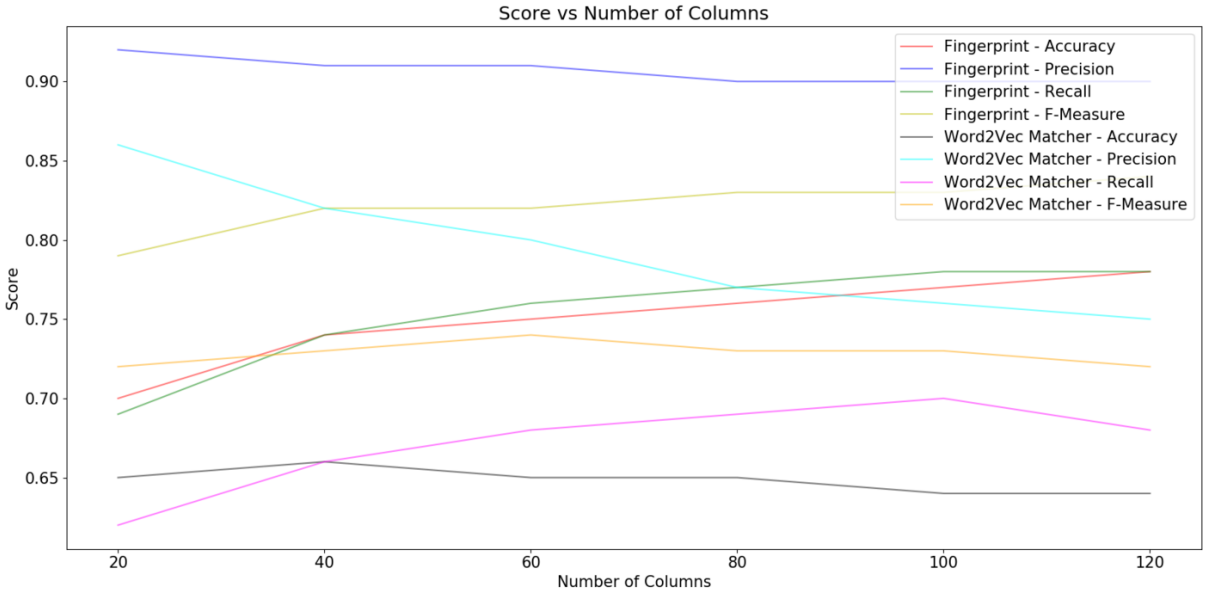


Figure 25 shows that accuracy results do not differ very heavily depending on the number of columns, all accuracies remain within a 1 percent deviation of the original measurement using 20 columns, indicating that the number of simulated columns used to train the matchers does not influence the matching result heavily.

The large differences between the accuracies of figures 25 and 26 point out that the detection of outliers does influence the performance heavily. The Fingerprint matcher and outlier detector performs better on all metrics when more columns are inserted, this contradicts the stable result of experiment 2.1. This difference indicates that the outlier detector performs better when more columns are used.

The results of the experiments performed on the Word2Vec matcher are also interesting. Overall, the accuracy in experiment 2.2 remains stable as happened in experiment 2.1, but the precision heavily drops and the recall increases as more columns are used. When looking at the formulas defined in section 4.3, we can see that the decrease in precision is caused by the increasing misclassification of outliers. The recall increases because inliers are classified better. This overall means that the outlier detector of the Word2Vec matcher starts to perform worse and the matcher starts classifying inliers better. This could be a coincidence but it could also be the case that the noise that is produced as input for the outlier detector resembles the actual outliers more closely when less simulated columns are used. Because of this it is hard

to say what the optimal number of classes is for all cases but we now do know that, when not using the outlier detection, the accuracy largely remains stable independent of the number of columns.

5.3 Experiment 3: Number of Instances per Column

This experiment was only performed on the Company.Info dataset since it was larger and thus allowed for a wider testing range. The result unfortunately is not averaged over multiple tests due to time constraints, this is a thread to the validity of the results. The number of simulated columns for each class is set for both the Fingerprint matcher and the Word2Vec matcher at 100 columns. The final 0 on the axis means that all instances present in the class were divided among the 100 columns per class.

The Syntax matcher classifies based on instance data, in order to make a fair comparison the syntax matcher was provided with $(num\_columns * num\_instances)$ instances per class.

Figure 27: Result of Experiment 3.1

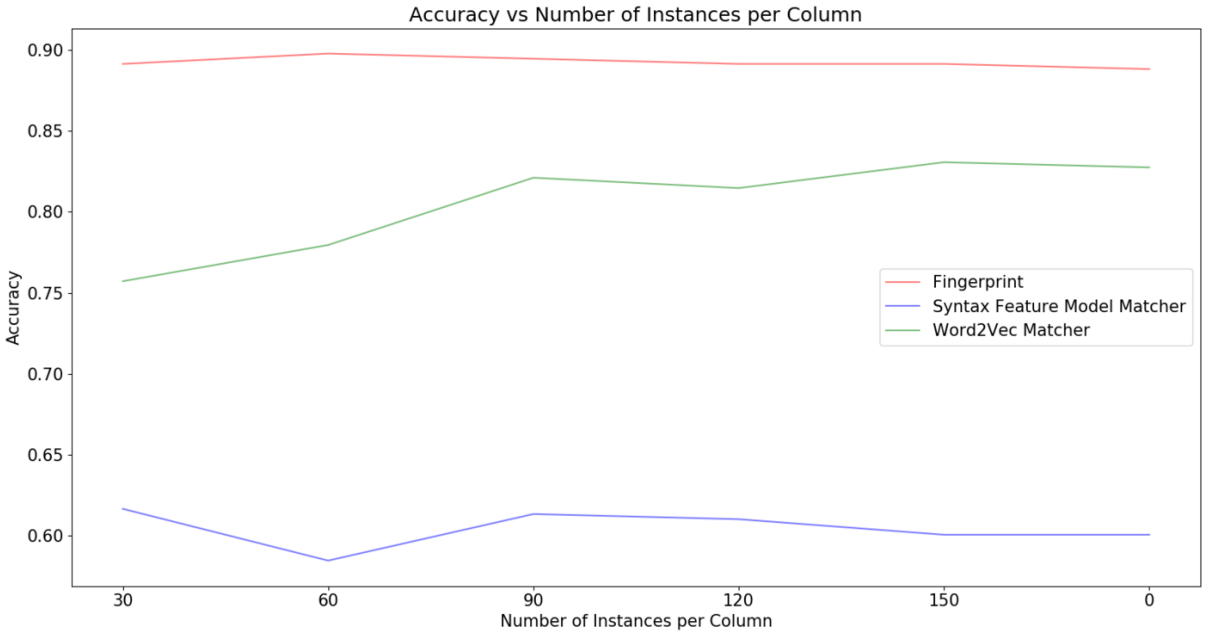
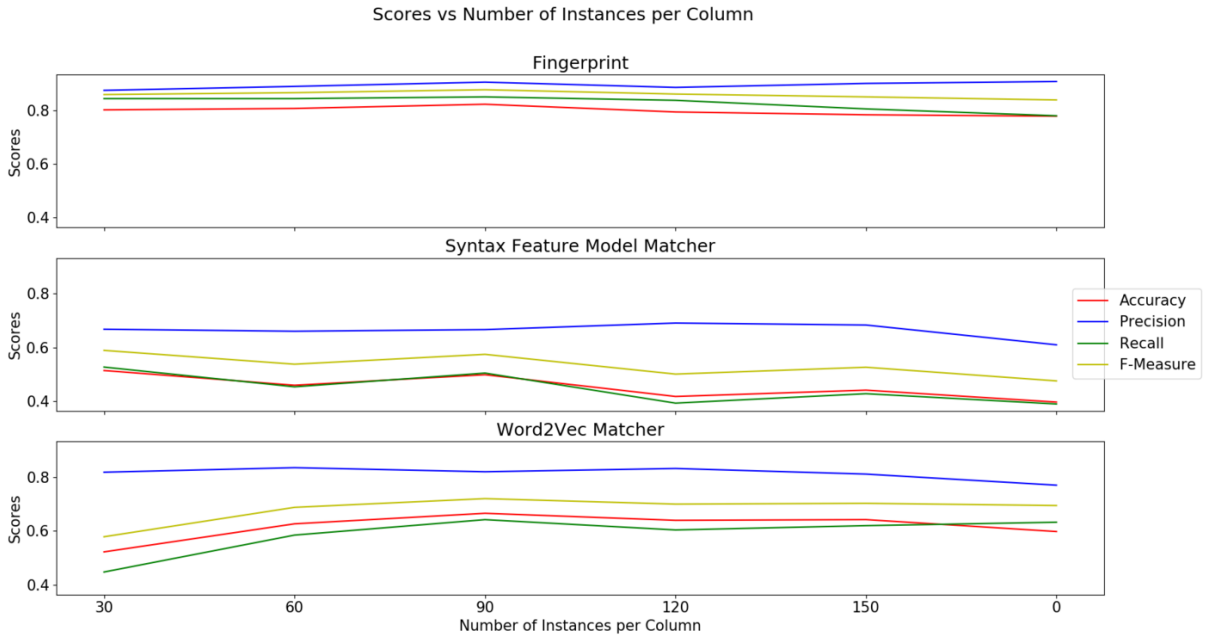


Figure 28: Result of Experiment 3.2



Since the results are not averaged, fluctuations from the mean might be the result of coincidence. Figure 27 shows that the Fingerprint and Syntax matcher both perform fairly constant, while the Word2Vec matcher performance increases as more instances are applied to the given amount of columns. The Fingerprint matcher has a small optimum at 60 instances. As more instances are added, performance slightly decreases for the Fingerprint and Syntax matchers.

Figure 28 shows that the performance of the Syntax matcher slightly decreases as more instances are added, and shows that the Word2Vec matcher has an optimum around 90 instances per simulated column. This optimum can also be seen in figure 28. Again the scores of the Syntax matcher and the Fingerprint matcher decrease as the number of instances increases, as could be seen in figure 27.

Both the inlier and outlier tests show a lot of similarities. The conclusion we can draw from both of these tests combined is that the number of instances that should be used differs per matcher, which was also the case for experiment 2.

5.4 Experiment 4: Pipeline

The aim of this experiment is to test the hypothesis made in section 3.6.3, namely that hierarchical classification could improve the matching result. To test this we will create a pipeline for both datasets which will be specifically designed based on the confusion matrices of experiment 1.1. The process of creating a match-tree and fitting it to the data will be discussed for each dataset.

Previous experiments show that outliers heavily influence the results of the experiments due to both their sheer portion of occurrences in the testsets, and the inaccuracy of the implemented outlier detectors. The outliers are only detected at the leaf matchers of the entire match-tree (discussed in section 3.5.1), and until those leafs are reached, data is treated as an inlier, a better outlier detection algorithm could therefore immediately heavily influence the results. Because of this, we will fit the pipeline according to the inlier results, and after this is done measure the overall performance with the outlier detection included.

5.4.1 Company.Info Data

At any point during this experiment please look at figure 29 if the setup is unclear. This image shows a representation of the complete and finalized pipeline that was the result of the entire experiment.

In the confusion matrices from experiment 1.1 (appendix section 9.1) we can see that the Syntax Feature Model has the following classification properties:

1. The Syntax Feature Model can excellently distinguish email addresses, postcodes, and domain names from all other classes.
2. There is only mild confusion between telephone numbers and dates.
3. There is no confusion between all the classes that contain textual data and the classes that contain numerical data.

We can utilize these properties to add the first 3 sub-matchers to the pipeline.

1. One for textual data (such as names, addresses, countries and provinces) since this contains a lot of confusion in all classes. The Word2Vec Matcher will be used to further classify these since it performs the best on this type of data (see confusion matrix 39).

2. One for the telephone numbers and dates. The Fingerprint matcher will be used to further classify this data since it can distinguish between these too perfectly (see confusion matrix 37).
3. One for numerical data. The Word2Vec Matcher will also be used for this data (again, see confusion matrix 39).

After these inital sub-matchers were installed, the experiments were ran again and there was already improvement. The confusion matrix in figure 49 (in the appendix section 9.3) shows dates are now already classified perfectly, and so are the telephone numbers if they advance to the Fingerprint Matcher. Sometimes the telephone numbers are passed on to the numerical Word2Vec matcher, therefore we need to further classify the data.

We are now left with 2 match blocks that still have a lot of confusion:

1. The textual data
2. The numerical data

We will first focus on the numerical data. The results from experiment 1.1 show that the Word2Vec matcher matches house numbers and telephone numbers better than all other classes. Therefore we use this property to 'cut' those classes from the others. We are then left with only *kvk_numbers* and *sbi_codes*. The specifically designed number matcher will be used to lastly classify these two.

This results in the confusion matrix presented in figure 50. The improvement of the classification result between the numerical data can already be seen. Since there is no matcher that can classify the numerical classes better than what is presented in this figure, we consider this side of the tree to be finished. We can now proceed to add sub-classifiers to the textual-data-side of the tree.

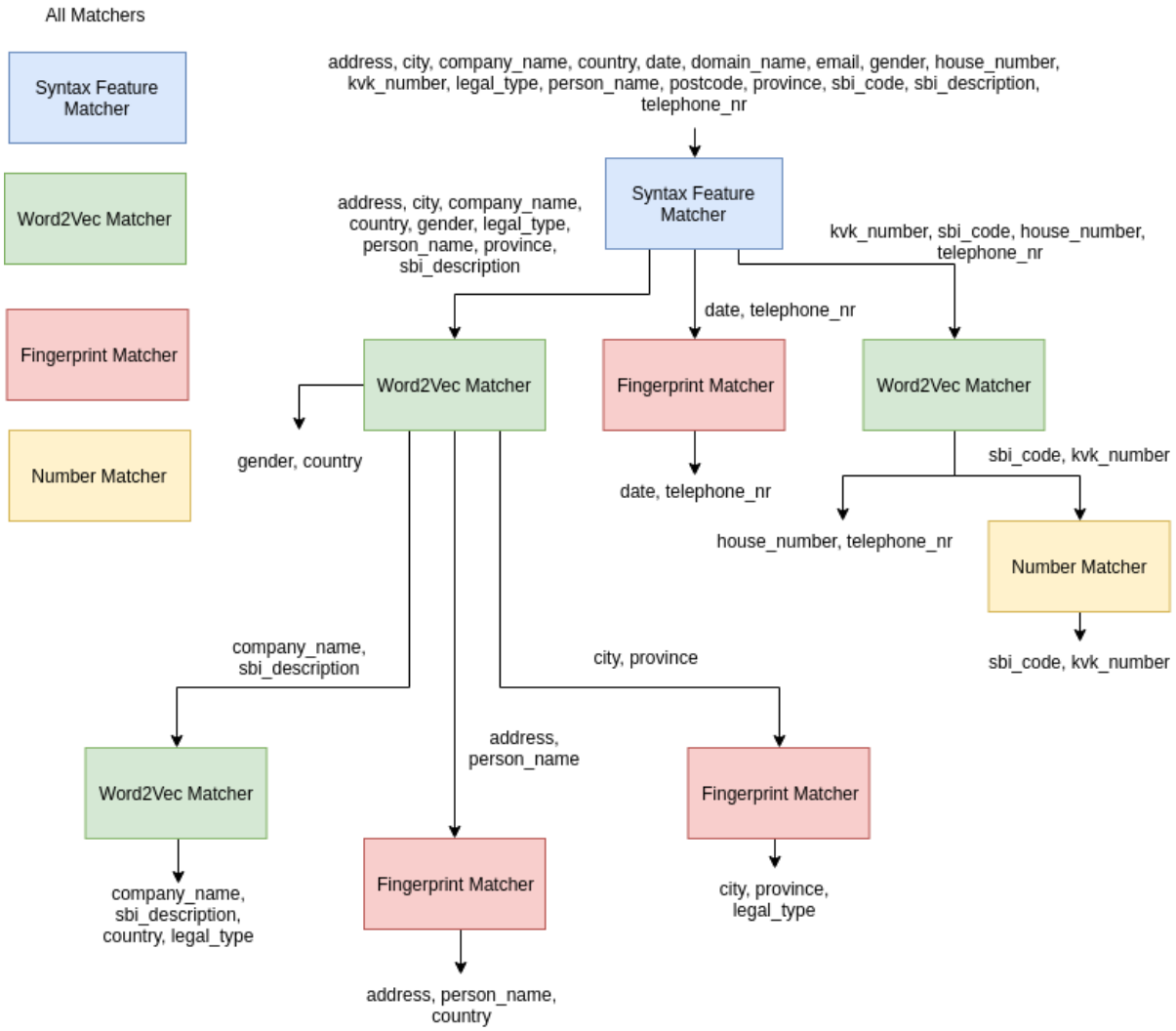
If we look at the confusion matrix we see that if the classes country, gender or legal_type are predicted, the prediction is always correct. We can use this property to already finalize the classification process for these classes. There is however a lot of confusion between the following classes:

1. Cities and provinces: The fingerprint matchers confusion matrix shows that it can differentiate between these two classes better than all other matchers. Therefore we add another Fingerprint sub-matcher to the textual data to differentiate between the cities and the provinces.

2. Company names and *sbi_descriptions*: The Word2Vec matcher is better in differentiating between these two classes than all other matchers, therefore it will be used.
3. Addresses and (person) names: The Fingerprint matchers showed little confusion in experiment 1.1 between these two classes, therefore it will be used in this case.

Cases exist where the classes *legal_type*, country and gender are misclassified. They then end up in one of the three new sub-matchers. To catch some of those cases, we also added these classes to the final leaf sub-matchers of the tree, so they might still be classified correctly. The final match-tree is depicted in figure 29.

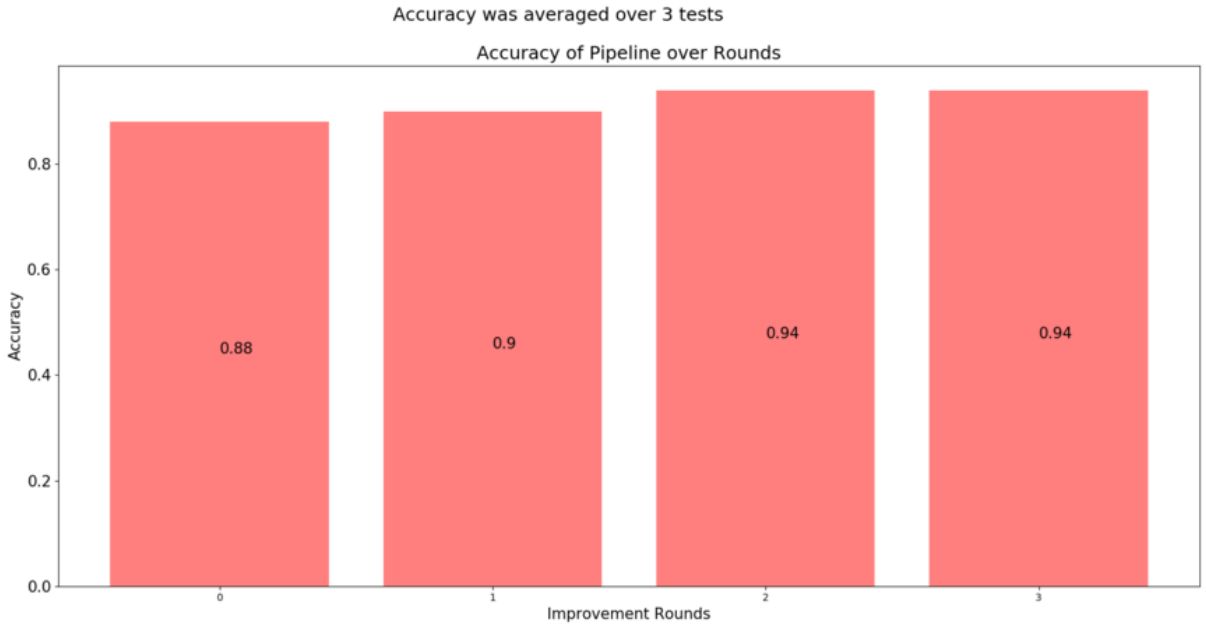
Figure 29: Finalized Fitted Pipeline for the Company.Info dataset



The improving accuracy of the pipeline (figure 30) shows that creating sub-matchers for classes that contain a lot of confusion has a positive effect on the overall classification outcome, as was expected. The results in figure 31 show that, when tested using the outlier detection, accuracy, recall and the f-measure scores all improve. Only the precision drops, this is probably because outliers are detected worse.

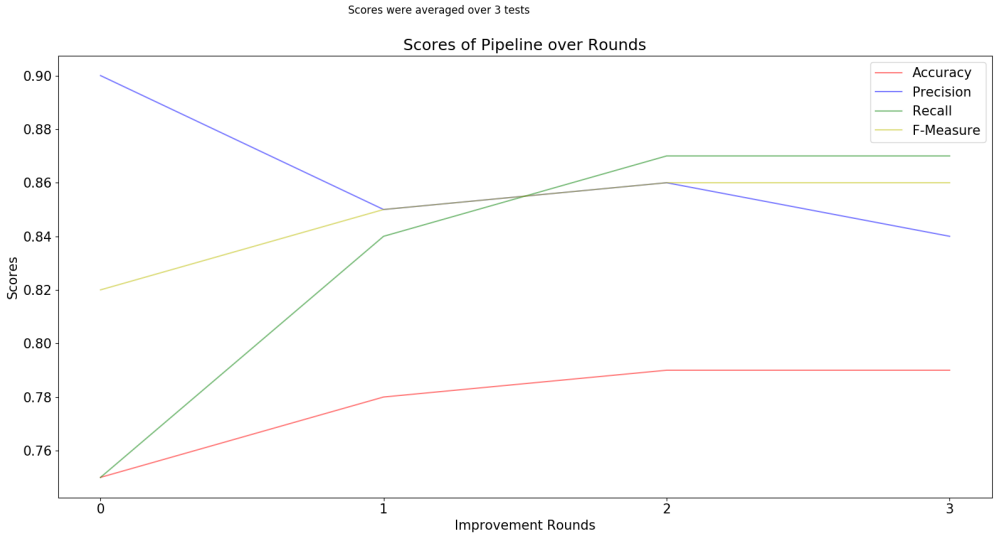
The reason outliers are probably detected worse is because of the way noise is produced for the binary classification (section 3.5.1). When there are a lot of classes from which we can randomly combine data, it could be that the produced noise more accurately resembles an outlier in comparison to the case where we only have a single class from which we can extract data to produce noise.

Figure 30: Result of Experiment 4.1 with fitted pipeline



The first round (round 0) presents the result of the baseline experiment. The other rounds present the result of each step where one or more sub-matchers were added.

Figure 31: Result of Experiment 4.2 with fitted pipeline



The first round (round 0) presents the result of the baseline experiment. The other rounds present the result of each step where one or more sub-matchers were added.

5.4.2 CKAN-CERIF Data

At any point during this experiment please look at figure 32 if the setup is unclear. This image shows a representation of the complete and finalized pipeline that was the result of the entire experiment.

In the confusion matrices from experiment 1.1 (appendix section 9.1), we can see that the matchers have the following properties:

1. No distinguishment can me made between the classes *e* and *Y*. After looking at their actual class names we could see that both classes consisted of dates, with most of them being the same, therefore we accept in this implementation that this class is classified incorrectly.
2. The classes *V*, *e* and *Y* are distinguished perfectly from all others by the Fingerprint matcher.
3. The Fingerprint matcher does not classify any of the data as a class that does not occur in the test set, all others do.

4. The class *has_description* is classified perfectly by the Word2Vec Matcher. For class *M*, the Word2Vec matcher only holds confusion with classes *V* and *label*
5. For class *H* the Syntax Feature Matcher only holds confusion with class *V*, and it holds confusion with only class *M* for the *has_name* class.

Using these properties we can easily find the optimal matcher configuration. The optimal configuration is as follows:

1. First we use the Fingerprint matcher to first extract classes *V*, *e* and *Y* from all others. Only the classes *M, H, has_description,* and *has_name* name are passed on to the next sub-matcher. The Fingerprint matcher has to be used first because it is the only matcher that does not classify any of the data as a class that does not occur in the test set.
2. Secondly the Word2Vec matcher is used to extract the classes *M* and *has_description*. All other data is passed to the next sub-matcher.
3. Lastly the Syntax Matcher makes the final classification.

As we can see in figure 33, the accuracy heavily improves. The confusion matrix (figure 52) shows that the accuracy would have nearly been perfect if it wasn't for the classes *e* and *Y*. Here we see the downside of applying ARPSAS to a flattened tree structure data. The instances of both tree leafs are highly similar but have a different meaning depending on the path to the leaf. ARPSAS can not differ between these because it only used the actual leaf data. Perhaps a different feature builder or matcher could have classified it correctly.

The result of experiment 4.2, depicted in figure 34, is interesting. All scores dropped heavily except for the recall. We can see that outliers are again detected less accurately, which was also the case during experiment 4.2 for the Company.Info dataset. The rise in recall is because inliers in the testset are classified more accurately.

Figure 32: Finalized Fitted Pipeline for the CKAN-CERIF dataset

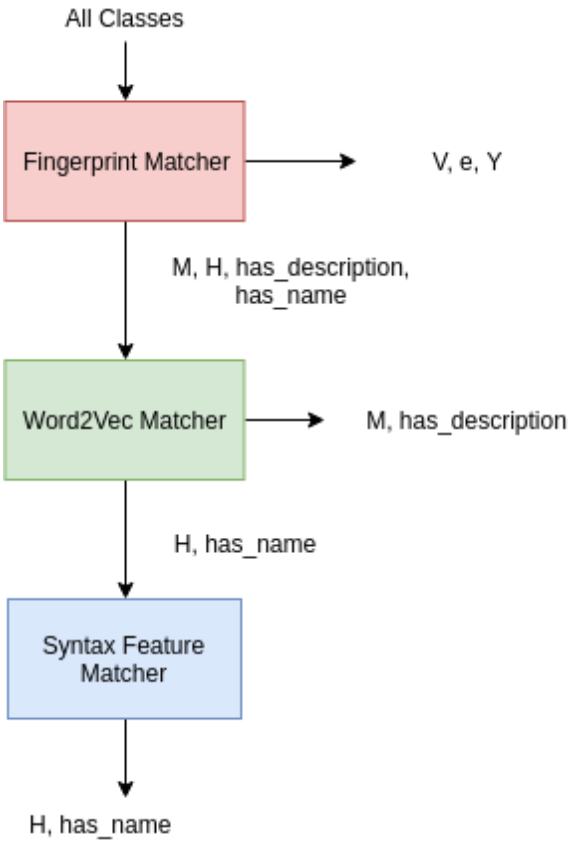


Figure 33: Result of Experiment 4.1 with fitted pipeline

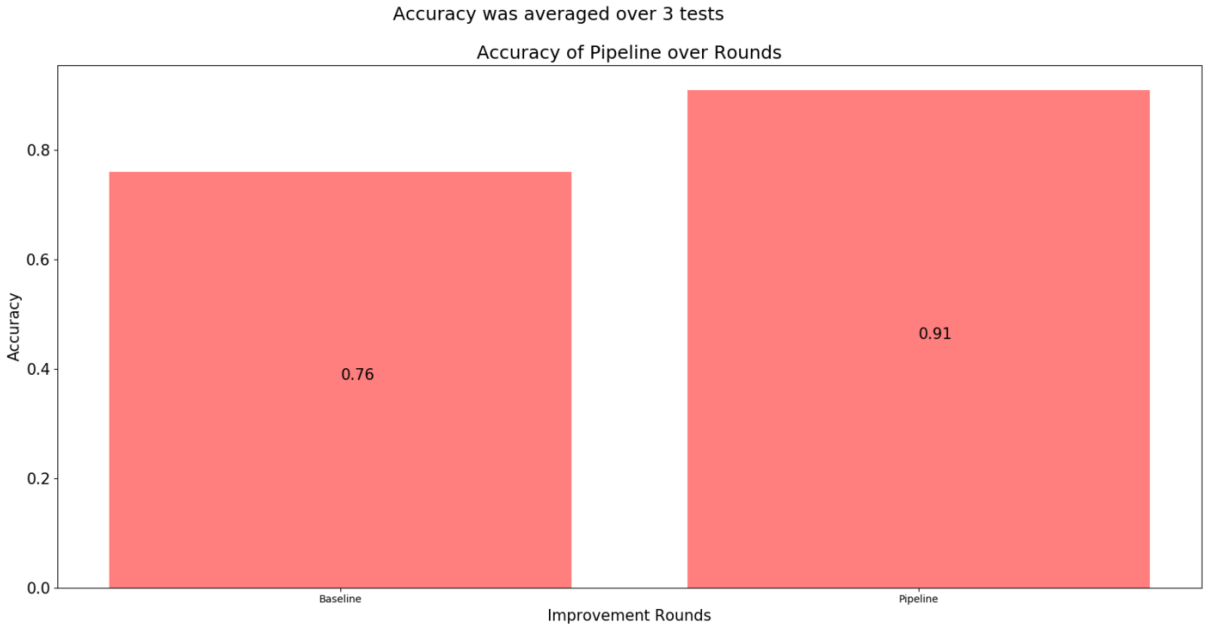
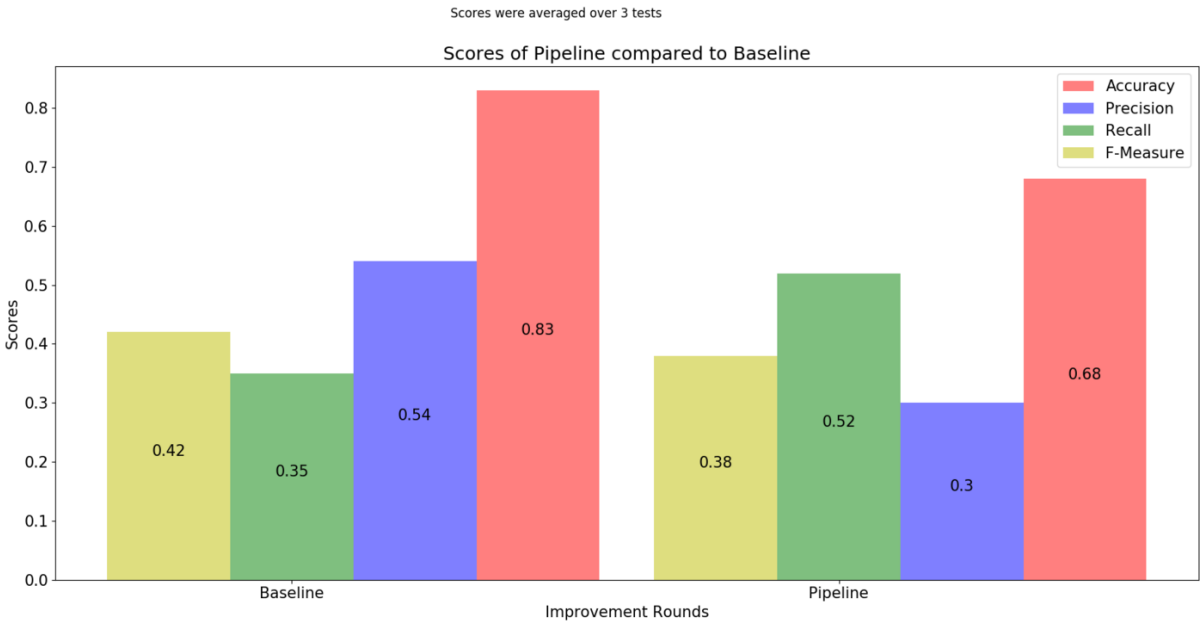


Figure 34: Result of Experiment 4.2 with fitted pipeline



5.5 Experiment 5: Additional Matcher Cost

The previous experiment shows that it is useful to add sub-matchers in order to fit a pipeline to your data. It is also useful to know what the addition of sub-matchers does to the training and classification time of a pipeline. For all matcher classes it is expected that the time it takes to train the system and classify data increases linearly over the number of additional sub-matchers.

These results were gathered by building a pipeline which pipes all data to the next matcher regardless of the result, this was done because data had to pass through all of the sub-matchers.

Figure 35: Result of Experiment 5.1

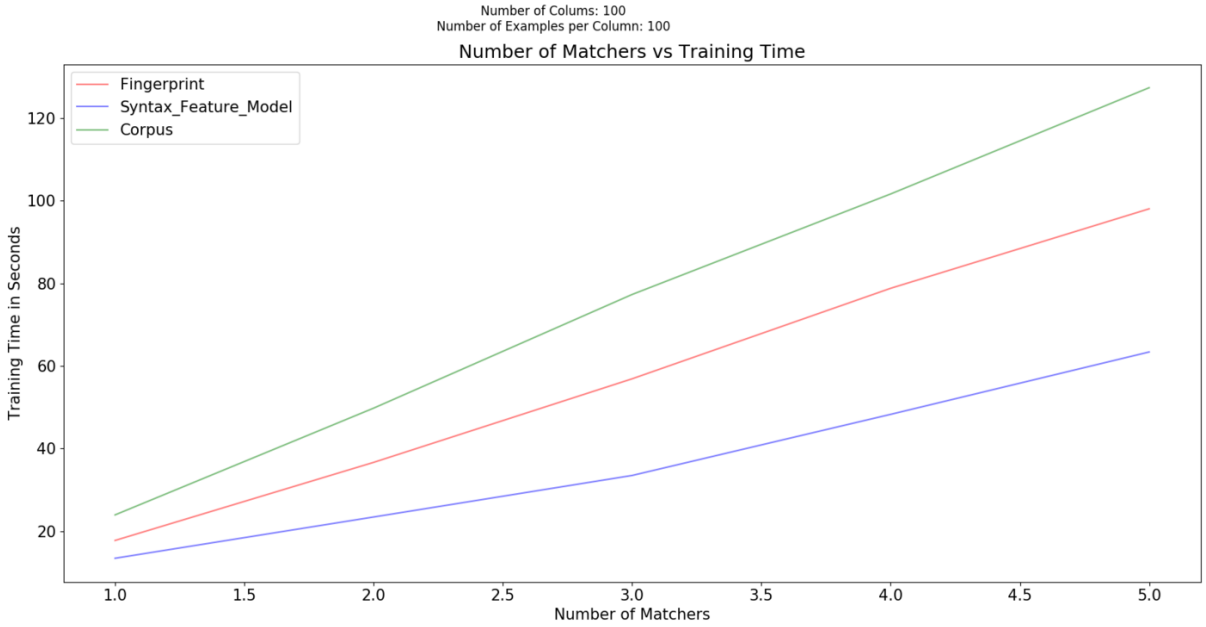
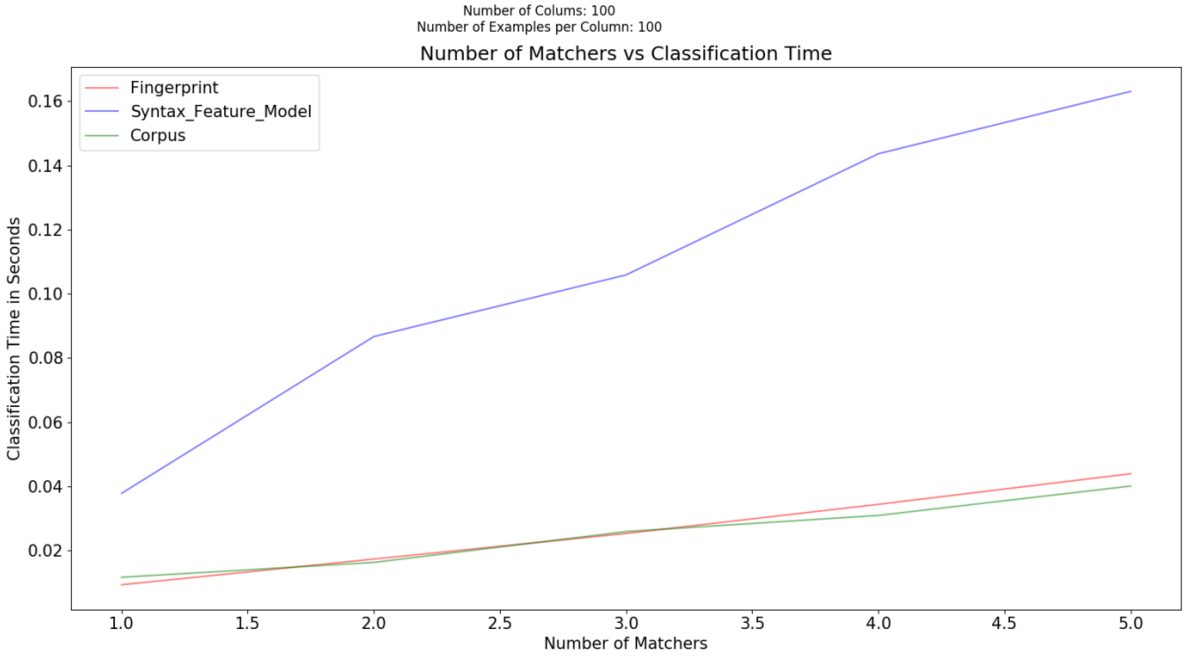


Figure 36: Result of Experiment 5.2



For each point in the graph, the same random column was classified a hundred times for a more accurate result

As expected, the training and classification times increase nearly linearly. This was very predictable since the task of reading and pre-processing the data, and training the matchers, is the same for every sub-matcher. We do want to point out that the results for the training time (figure 35) were not averaged over multiple experiments, this could explain why the result is not completely linear.

5.6 Evaluation of Validity

The results for every experiment performed in this thesis are heavily dependent on the datasets. If the CKAN-CERIF dataset would have more inliers, the results could have been completely different. Outliers also influence the results heavily in each experiment, not only because they are in both cases the most occurring class, but a better and more stable outlier detection algorithm could have boosted the scores for each experiment. However, it was also very interesting to see how this particular algorithm performed during the experiments, and how it performed worse when less classes were assigned to a matcher.

Executing the experiments twice, once with outlier detection and once without it, did help

with researching the influence of the outlier detection on the overall system, and helped with excluding it as a factor for the test results. The metrics (especially recall) also contributed in providing insight in the test results where outlier detection was used.

The portioning of each occurring class in the testset also is a thread for validity which has been named multiple times already. This influences the metrics presented in nearly all figures, but not the normalized confusion matrices (all of them are in the appendix). These show that certain results are still reliable even though class sizes differ heavily in the testset. For example, the confusion matrices show that adding sub-matchers and fitting a pipeline does reduce the confusion when it comes to inliers in all cases.

The training time and classification time of a matcher is also something that is completely dependent on the size of your data. More instances in a column requires more computation time. The result of this experiment however does show that this time is linear for same-sized data over additional sub-matchers.

Another thread to validity is the way of how the pipeline was optimized. It would have been cleaner if the besides the learn- and testset there would have been a second testset on which the fitted pipeline could have been tested in order to confirm the results.

Experiments 2 and 3 could also be dependent on the data in the testset. If columns in the testset had a lot of instances, then adding more instances to the training columns could have improved the result (more resemblance of training data to the actual data). It would be interesting to see the influence of the number of instances in the testset columns on the overall scores of the matchers.

6 DISCUSSION

The results of the experiment 1 clearly confirm the hypothesis that different matchers operate better on certain classes within a schema than others. This supported by both figures 22 and 21, and by confusion matrices of the experiment (section 9.1). By confirming the this hypothesis we are one step closer to confirming that a hierarchical classification pipeline can reduce confusion among the matchers.

Experiment 2 shows that when only using inliers, the accuracy largely remains stable (figure 25) independent on the number of columns that is used to train the matching component. This strengthens the validity of the results produced by experiment 4, since this factor does not impact the results. The hypothesis that using a very large or low number of columns affects the matching result in a negative way can therefore be rejected. We do however want to point out that the validity of this result is still threatened by a dependency on the dataset. The results of this test could have been different if a smaller dataset was used.

Experiment 3 shows that the number of instances that is used within a simulated column differs from matcher to matcher (figures 27 and 28). The hypothesis that as many instances as possible should be used is rejected since an optimum can be found for all matchers in figure 27. The reason an optimum can be found is probably because of the over- and underfit problem discussed in section 4.6. Even though the optimums found were very close to the initial results of the baseline experiment, by using these optimums we could already improve the matching results during experiment 4.

The results of experiment 4 confirm the hypothesis that a hierarchical classification pipeline can reduce confusion among the matchers. We can see this in figures 30 and 33 by the improvement in accuracy when sub-matchers are introduced. What is more interesting however is how such a pipeline can be created. Currently there is no algorithm implemented that automatically builds the pipeline given a dataset and a *calibration set.* During the experiment, sub-matchers were introduced based on the human interpretation of the confusion matrices. We found that the following guidelines helped creating the pipelines:

1. Start off by obtaining confusion matrices for each matcher, for the entire testset.
2. Based on the confusion matrices, identify classifiers that are able to separate certain classes perfectly from the start.
3. Based on the confusion matrices, group the classes that contain a lot of confusion among each other together.

4. Matchers should group the classes that contain a lot of confusion among each other, and pipe these to sub-matchers who are better in distinguishing between the classes of these groups.

5. Whenever and as early as possible, use the matchers that identify certain classes perfectly to eliminate confusion by using them as a tree leaf for the classes that they classify correctly.

Using these guidelines we can create an hypothesis for an algorithm that can automatically configure such a pipeline given a target-schema:

```
match_blocks = get_match_blocks()
confusion_matrices = get_confusion_matrices_matchers(dataset, matchers)
pipeline = get_blank_pipeline()
add_correct_predictions(pipeline, confusion_matrices,
                                    confusion_matrices, match_blocks)
create_sub_trees(pipeline, confusion_matrices, match_blocks)

# Fill the pipeline with matchers that
# predict a given class 100% correctly in
# order to already remove those classes
def add_correct_predictions(pipeline, confusion_matrices,
                                    base_confusion_matrices,
                                    match_blocks):
    for matcher in match_blocks:
        for classname in confusion_matrices[matcher]:
            if base_confusion_matrices[classname].correct_prediction ==
                                                    1:
                # Add tree leaves for the classes
                # that can be predicted perfectly.
                # The rest of the data should be send through the
                                                        pipeline
                pipeline.add_match_block(matcher, classname)

# all data that is left contains confusion among all matchers.
# These classes should be grouped. We should first find
# a matcher which can group these classes together
# (as best as possible),
# then we should find out if we can find a matcher which can classify
# parts of these groups correctly.
def create_sub_trees(pipeline, confusion_matrices, match_blocks):
    # get sub group based on confusion matrices
    class_groups, group_confusions = get_class_groups(
```

```
                                        confusion_matrices)
    # Add matchers for classes that can differentiate between any of
                                        the classes within the groups
    add_correct_predictions(pipeline, group_confusions,
                                        confusion_matrices,
                                        match_blocks)
    sub_groups = {}
    for matcher in match_blocks:
        # get_sub_groups should find out
        # how good a matcher is at separating
        # the groups from the rest of the data.
        sub_groups[matcher.name] = get_sub_groups(class_groups, matcher
                                            )

    # retrieve the matcher that can
    # separate the groups as good as possible.
    best_matcher, groups = get_best_match_block(sub_groups)
    pipeline.add_match_block(best_matcher, groups)

    # Recursively add more branches to the match tree until
    # all groups have been classified as good as possible
    for group in groups:
        create_sub_trees(pipeline, group_confusions[groups],
                                            match_blocks)

# The algorithm should terminate
# when no matcher can improve the classification
# result of the group anymore,
# or separate the group any better.
```

This algorithm is of course written in pseudo-code and in future research should be developed further. The key of improving the pipeline however is the introduction of sub-matchers that can differentiate between classes where confusion occurs.

As is shown in experiment 5 and figures 35 and 36, ARPSAS scales linearly in classification and training time when more sub-matchers are introduced. There are however more metrics that could be tested in the future such as memory usage. We can confirm the hypothesis of linear scaling for the given amount of sub-matchers but we can not confirm that this hypothesis also holds when more sub-matchers are introduced than that were present during the experiment.

7 CONCLUSIONS

The research question for this thesis was: How can an effective semi-automated schema matching pipeline be created and customized for a given dataset? Before we can answer this we first have to answer the helper questions:

- What are the algorithms that can be used for schema matching?
- What are the key performance indicators for schema matching?
- What are the limits of the semi-automated framework?
- How can outliers be included in the customization of a semi-automated schema matching framework?

In the state of the art section (section 2 the classification of schema matching algorithms was shown, and ARPSAS prototype has been designed to allow implementation for all of them so a user can experiment with building a customized pipeline configuration. In order to match schemas, a user can first of all create individual matchers based on a single or multiple match criterion (a hybrid matcher). A user can also combine the results of multiple hybrid matchers into a composite matcher. The match criterion can consist of element- or structure-level schema data. The decision making components in these algorithms can base their decision on either a set of rules and constraints, linguistic data, or machine learning.

The performance of any pipeline should be measured by the portion of correctly mapped columns or instances (structures or elements, section 2.1). Depending on what your specific goal is, you could configure the pipeline to have either a higher precision (only predict cases of which you are very certain and classify others as unknown) or a higher recall (predict as much as possible, don't detect outliers). A user should choose his or her measurement according to the use case.

The experiments on the CKAN-CERIF dataset showed that ARPSAS is not necessarily domain specific. The results indicated that collecting the instances in the tree leafs into a column and collectively classifying all the tree leafs together produces good accuracy (91%). For good measure, this method should be compared to a schema-matcher that is specifically designed to match tree-type structures such as COMA (section 2.3). The confusion matrices of these experiments do however point out that this method does not work if the instances contain similar data, but have a different meaning depending on the tree branch. This is not necessarily a limitation in the framework, a different algorithm (which can be implemented

in ARPSAS) could have performed better. A limitation of the framework is however shown by the CKAN-CERIF dataset: The framework can not handle data that is largely generated during a mapping process.

Currently, another limitation of the framework is the automation of the pipeline configuration process. The algorithm presented in section 6 has not been tested. It would also have been nice if the framework could provide suggestions based on tests executed on the target-schema data.

A limitation which was present in the experiments, but should be investigated in future work, is the detection of outliers. By allowing users to implement their own outlier detection algorithms in the match-tree leafs they can experiment with correctly detecting outliers after their classification pipeline is optimized for all inliers.

It is shown that ARPSAS can aid in creating an optimized schema-matching pipeline by enabling a user to implement sub-matchers for classes that contain confusion among each other. The confusion matrices give an insight in the constraints of the algorithms, matchers and pipelines that a user defines. If accuracy is low, but could be improved, more sub-matchers should be introduced (if you want higher accuracy), but this is traded off against classification and training time, as was shown in experiment 5.

8 FUTURE WORK

ARPSAS can be used to further research schema-matching algorithms based upon column classification. The matchers can be modified to implement many more different algorithms, it would be interesting to see where more of the constraints of the framework lie. It would also be interesting to see how the existing matchers perform on different datasets, or if it is possible to match a single schema into the other with only a single column per class.

Future work should include experimentation with the automation of the pipeline creation, or a more precisely defined algorithm to find the optimal configuration based upon the confusion matrices. Within the current setup, ARPSAS is used for schema-matching, but as stated in section 3, the match-tree can be used to hierarchically classify any type of data, depending on the implementation within the feature builders and matcher classes.

During these experiments, the pipelines were specifically fitted to the testset, it would also be interesting to see if the result are the same if the pipeline is fitted to a separated part of the

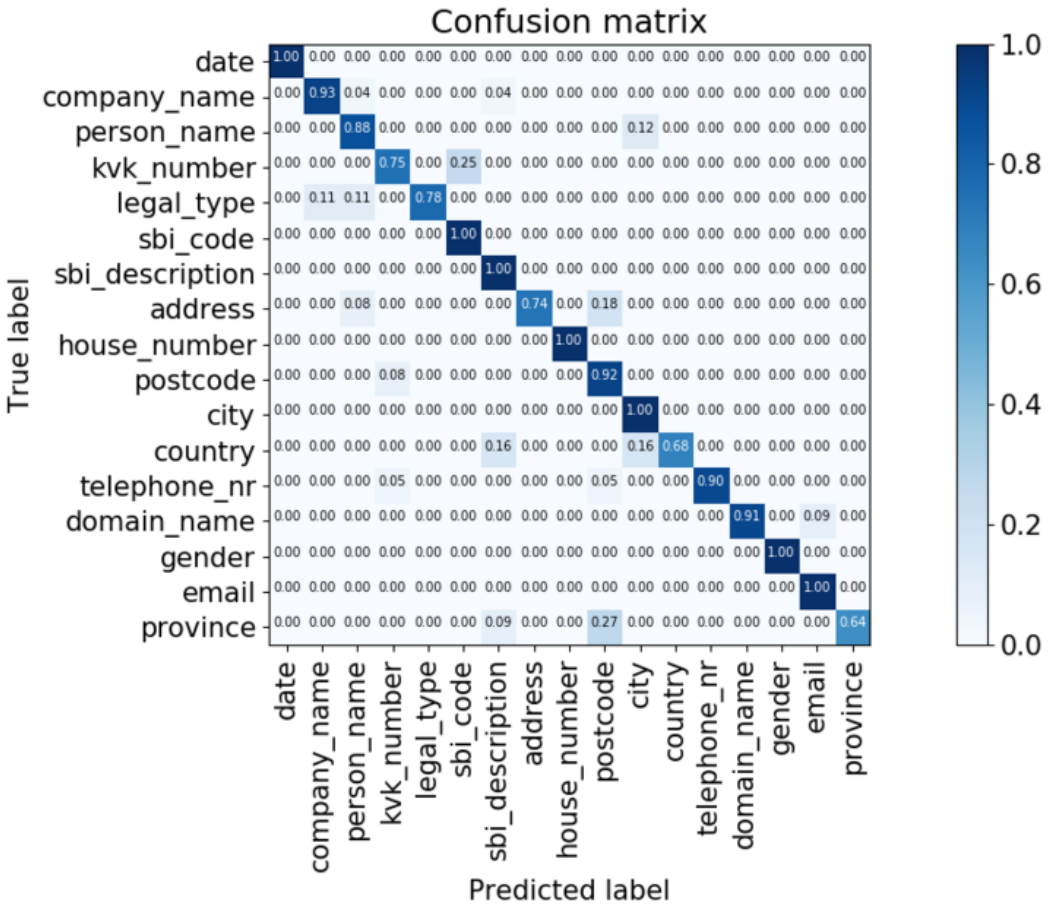
learnset.

ARPSAS lets its users try out different pipelines and matchers. All of the results gathered in these experiments can be different depending on the dataset and on the defined features and matchers. Since the outlier detection that was implemented did not perform as well as expected but did heavily affect the results, it is recommended that for future work, a better outlier detection is investigated.

9 APPENDIX

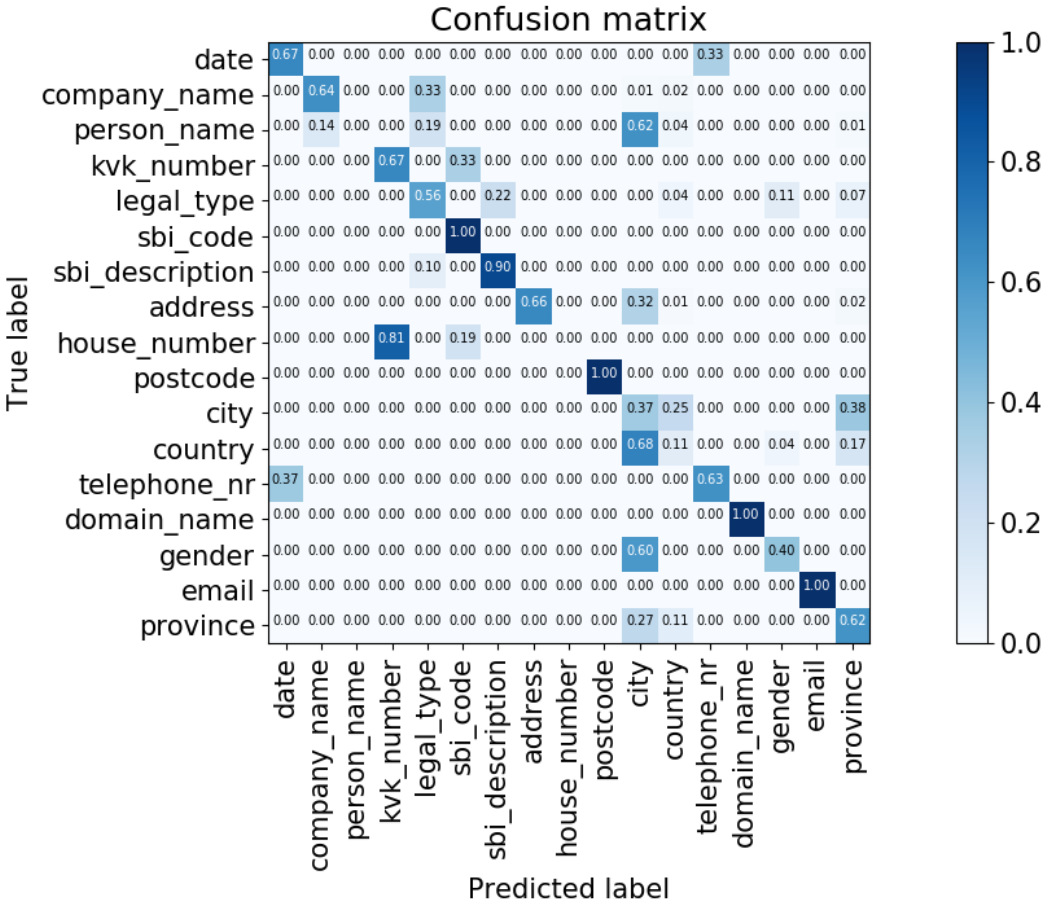
9.1 Confusion Matrices Experiment 1.1

Figure 37: Experiment 1.1, Company Info Data



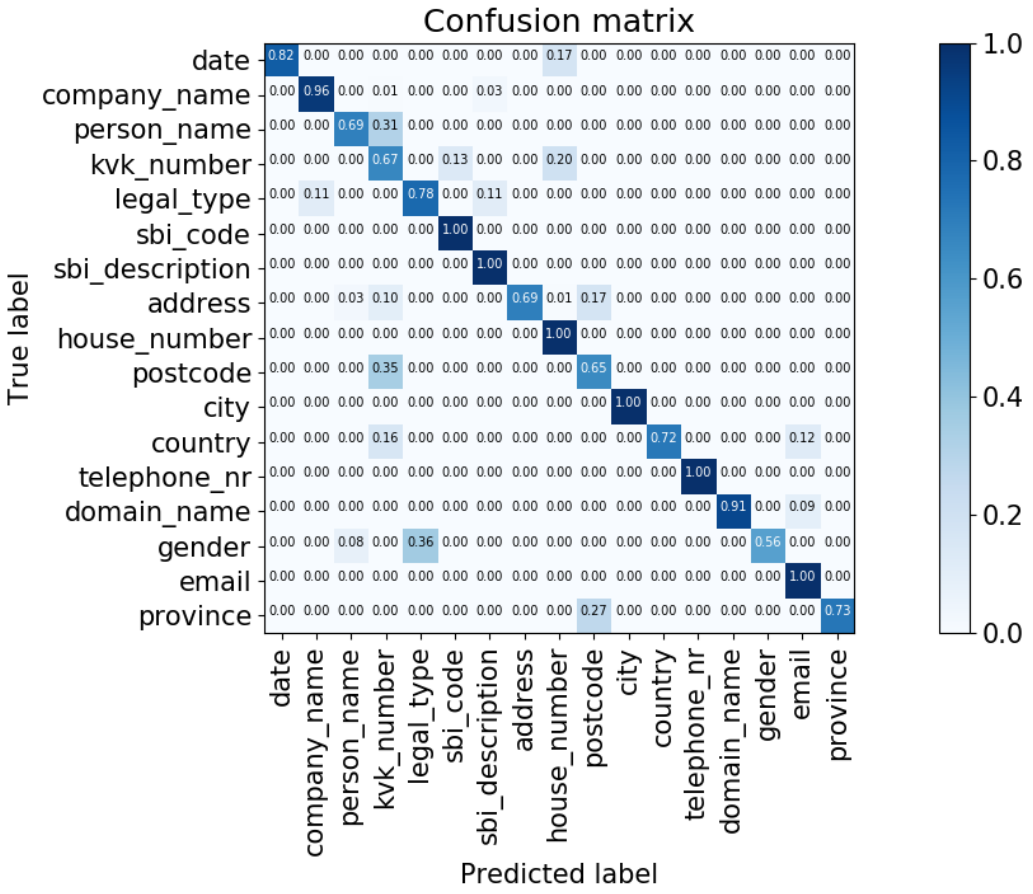
Fingerprint Confusion Matrix

Figure 38: Experiment 1.1, Company Info Data



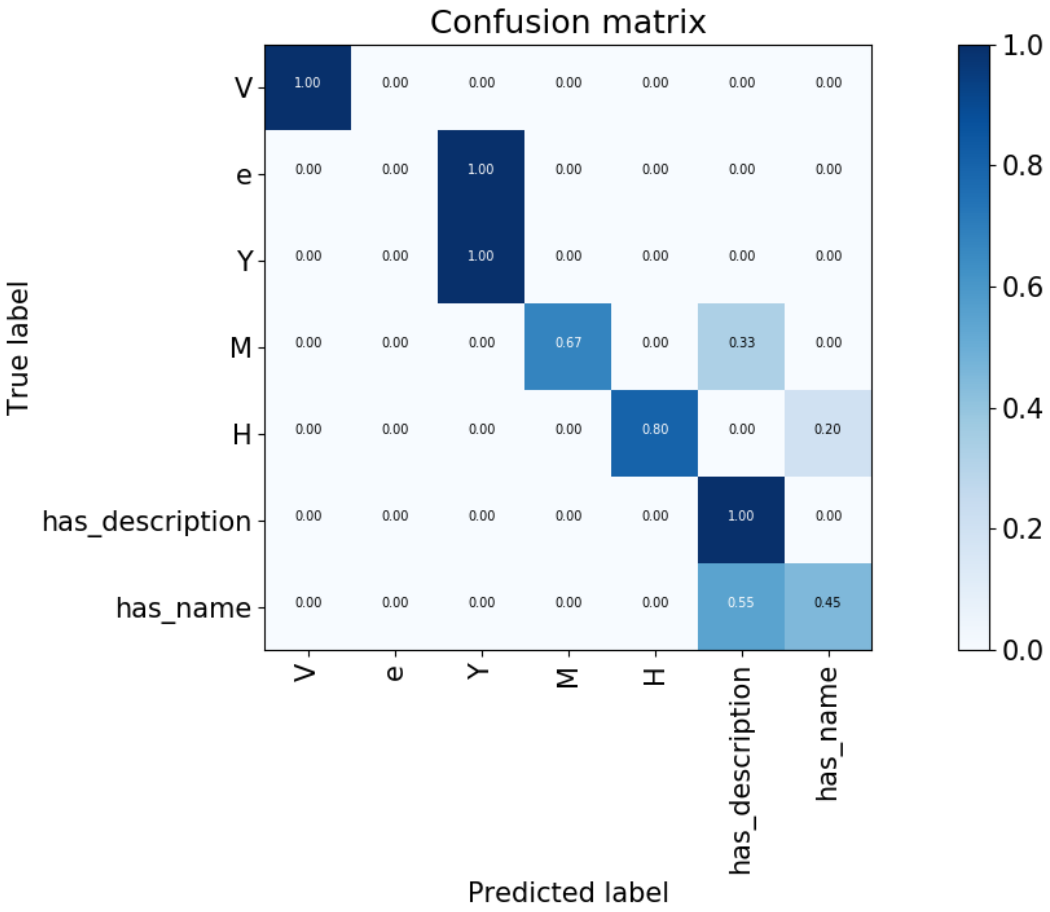
Syntax Feature Model Confusion Matrix

Figure 39: Experiment 1.1, Company Info Data



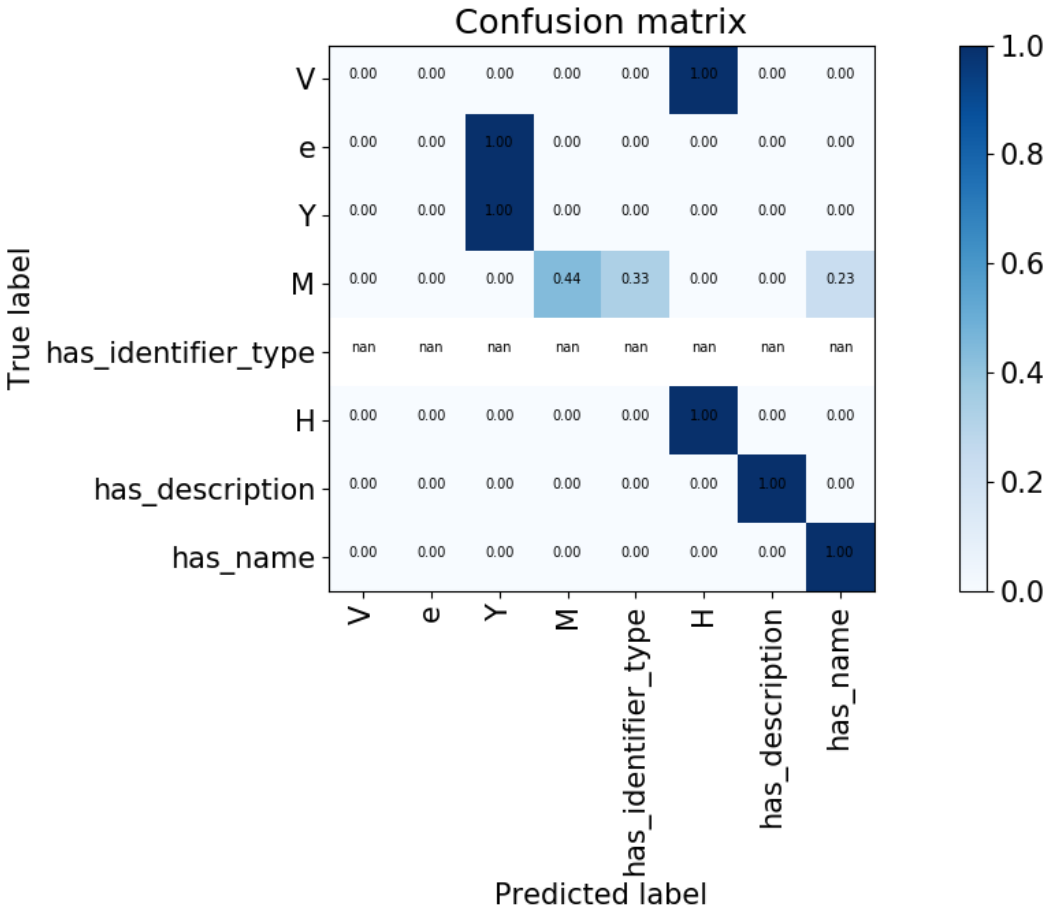
Word2Vec Confusion Matrix

Figure 40: Experiment 1.1, CKAN Data



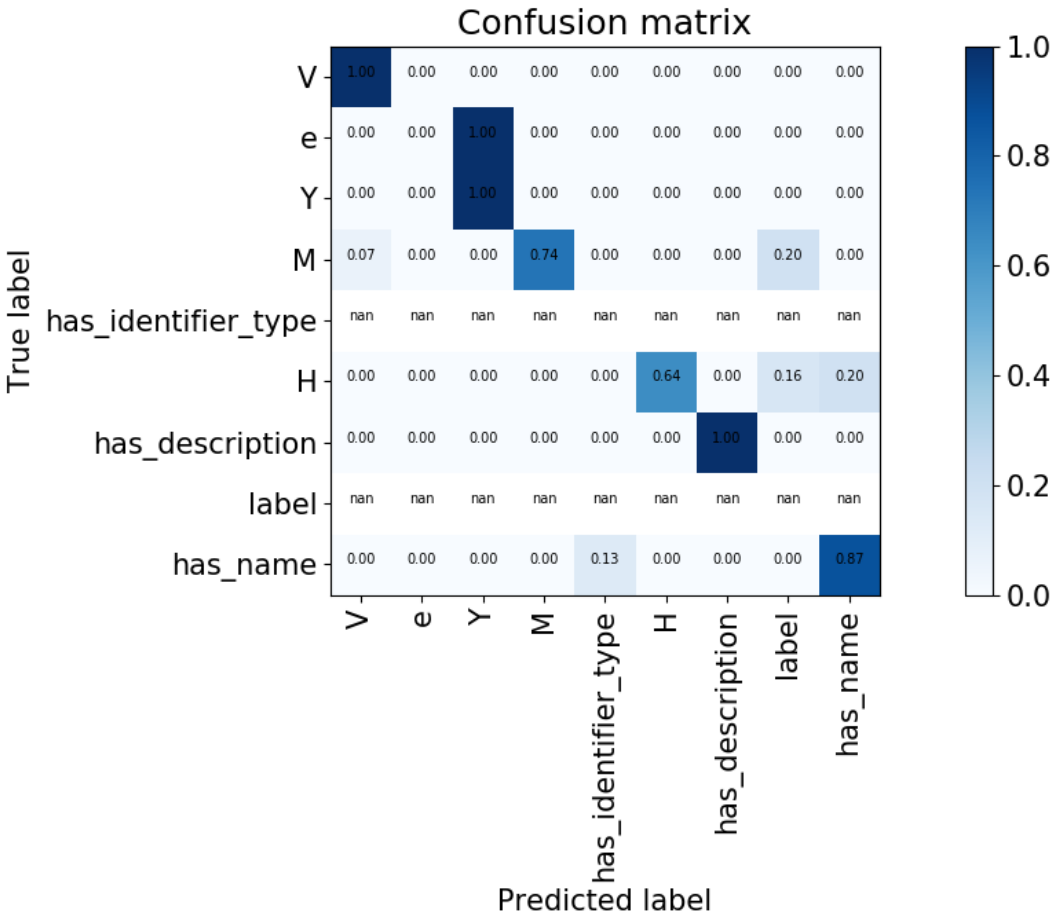
Fingerprint Confusion Matrix

Figure 41: Experiment 1.1, CKAN Data



Syntax Feature Model Confusion Matrix

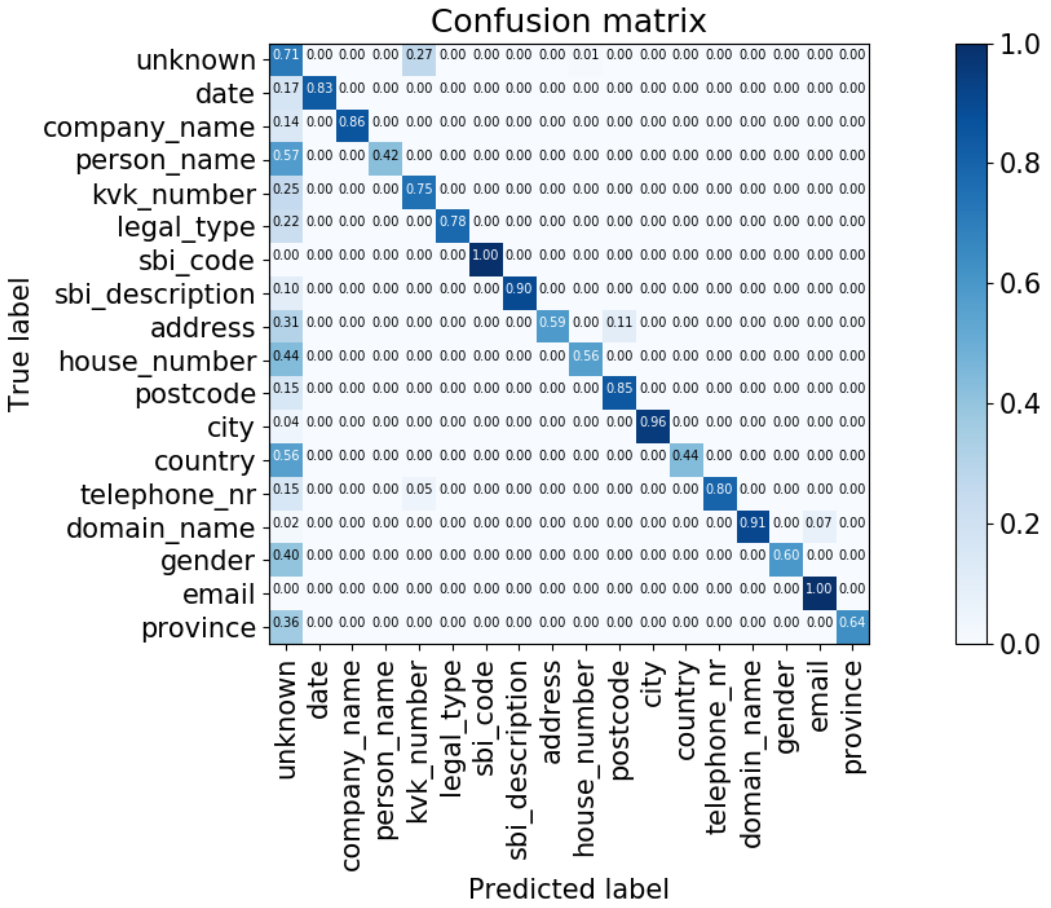
Figure 42: Experiment 1.1, CKAN Data



Word2Vec Confusion Matrix

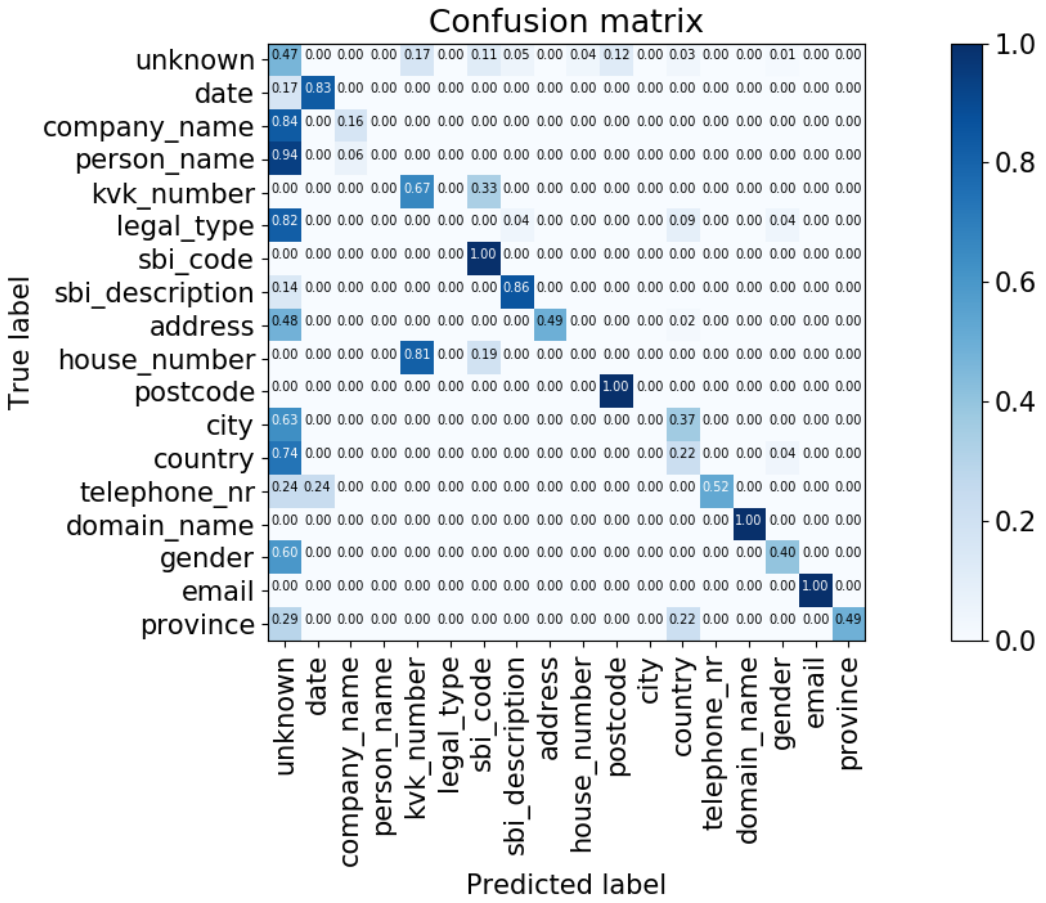
9.2 Confusion Matrices Experiment 1.2

Figure 43: Experiment 1.2, Company Info Data



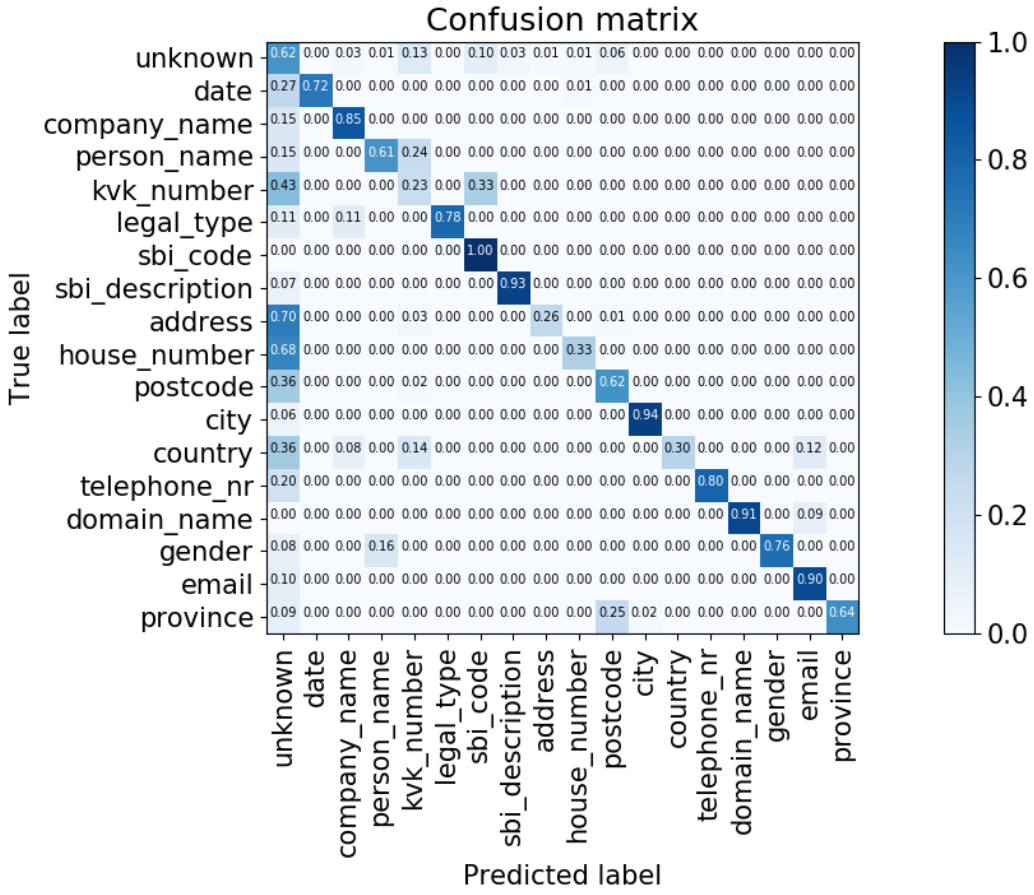
Fingerprint Confusion Matrix

Figure 44: Experiment 1.2, Company Info Data



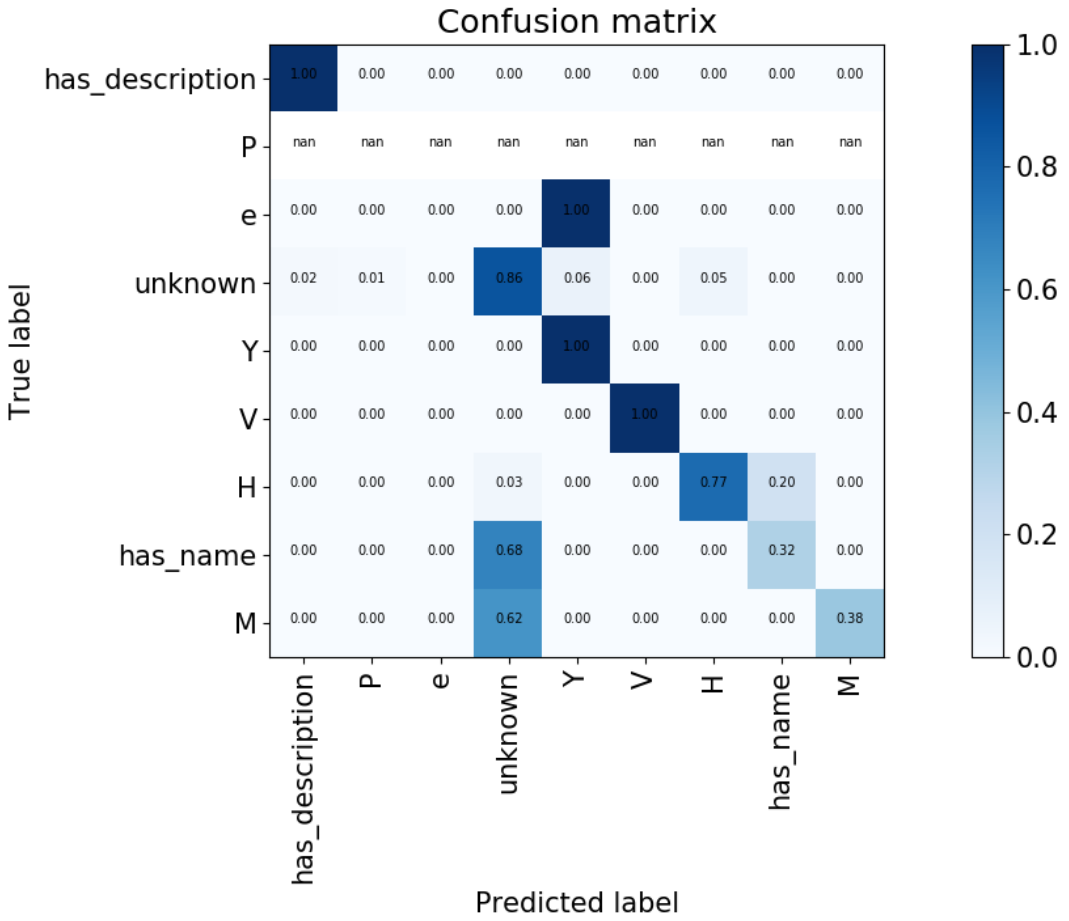
Syntax Feature Model Confusion Matrix

Figure 45: Experiment 1.2, Company Info Data



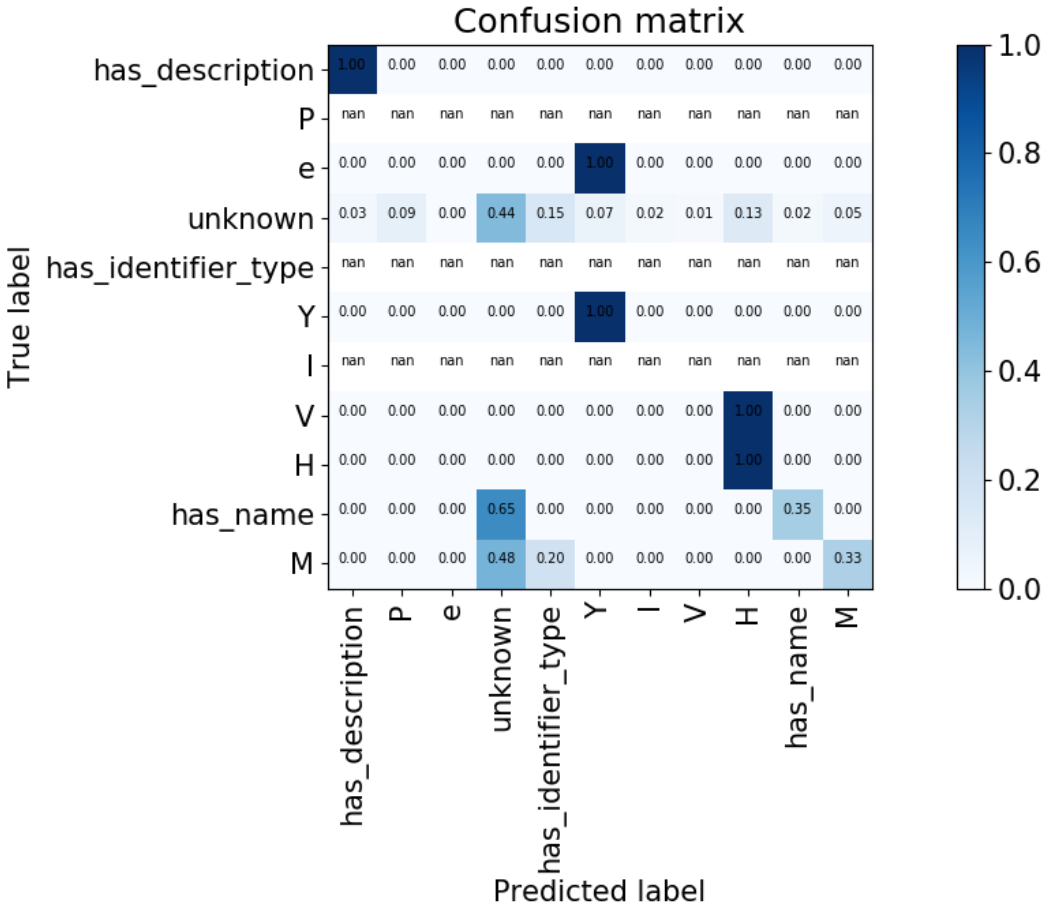
Word2Vec Confusion Matrix

Figure 46: Experiment 1.2, CKAN Data



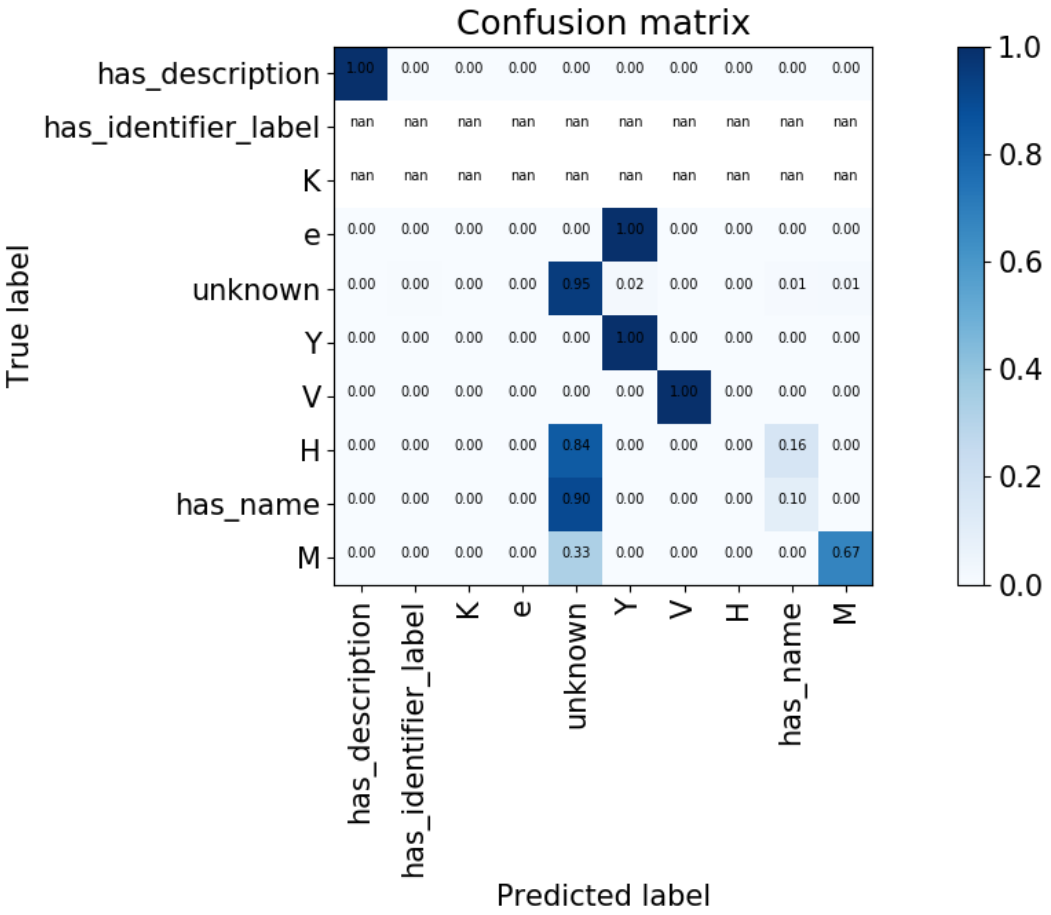
Fingerprint Confusion Matrix

Figure 47: Experiment 1.2,CKAN Data



Syntax Feature Model Confusion Matrix

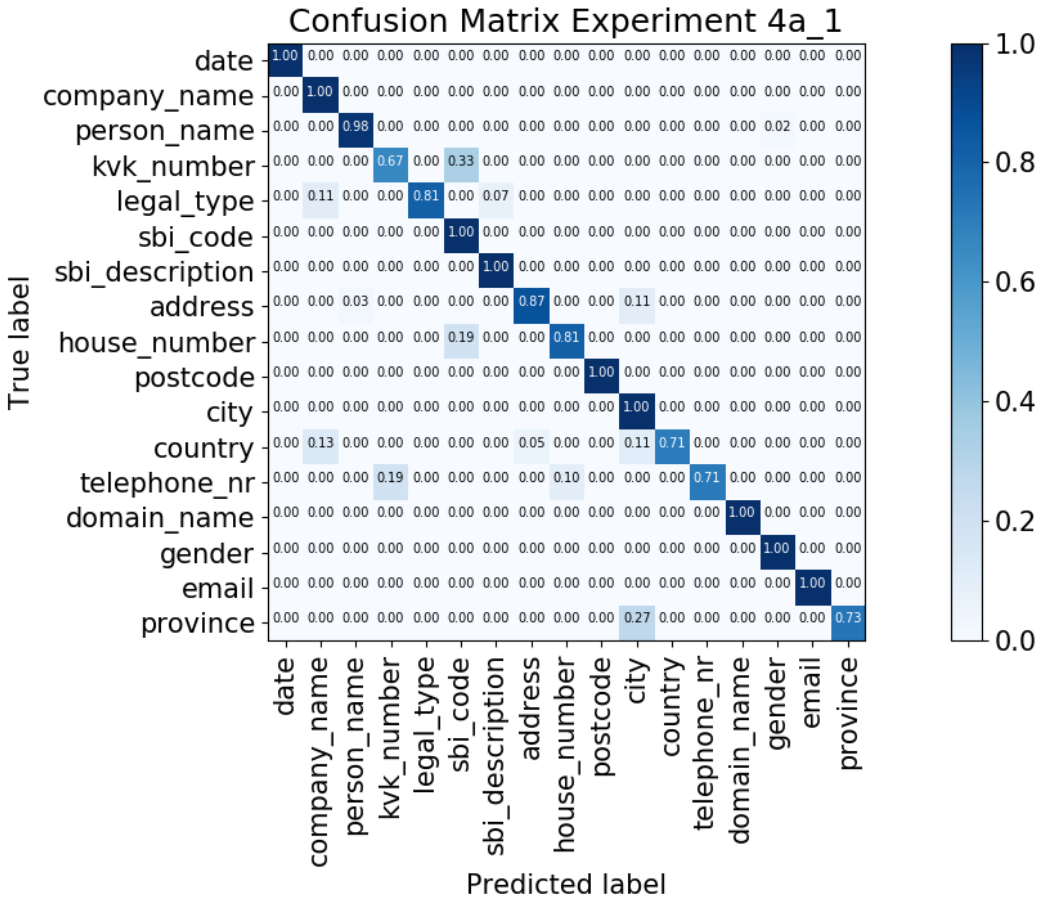
Figure 48: Experiment 1.2,CKAN Data



Word2Vec Confusion Matrix

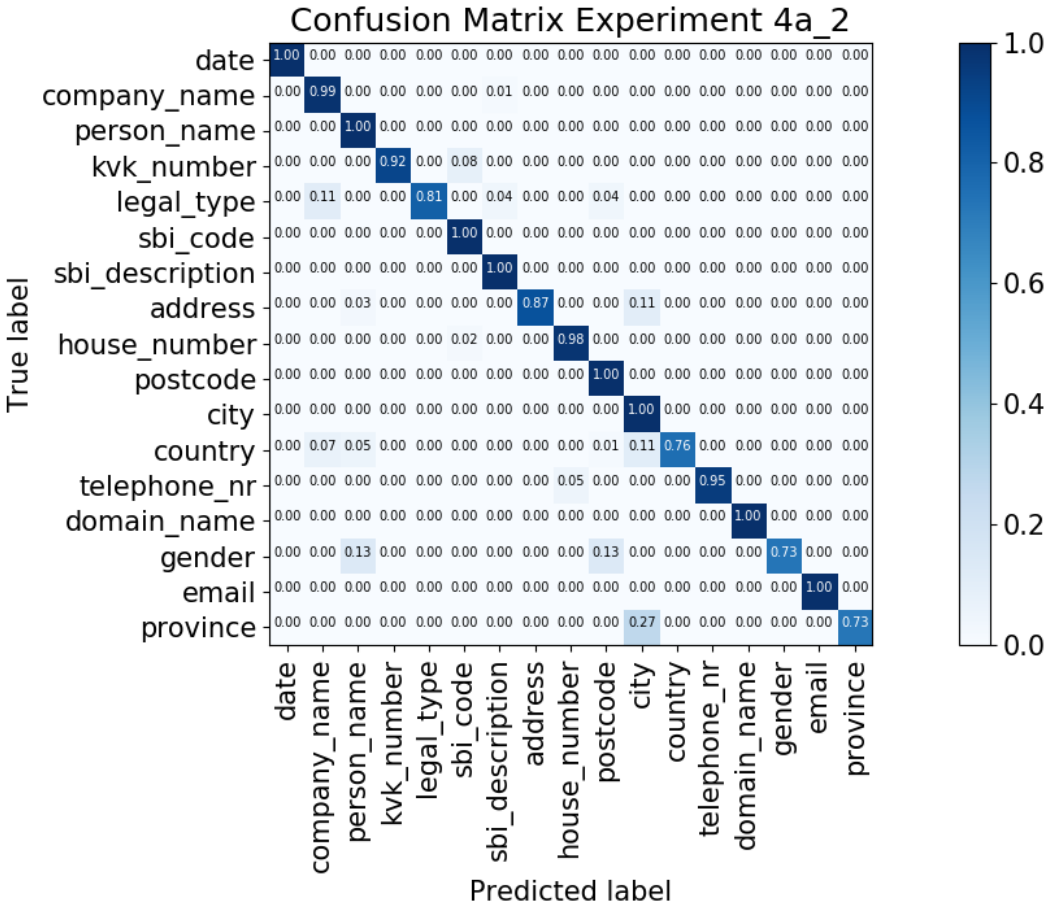
9.3 Confusion Matrices Experiment 4a

Figure 49: Experiment 4.1, Company Info Data



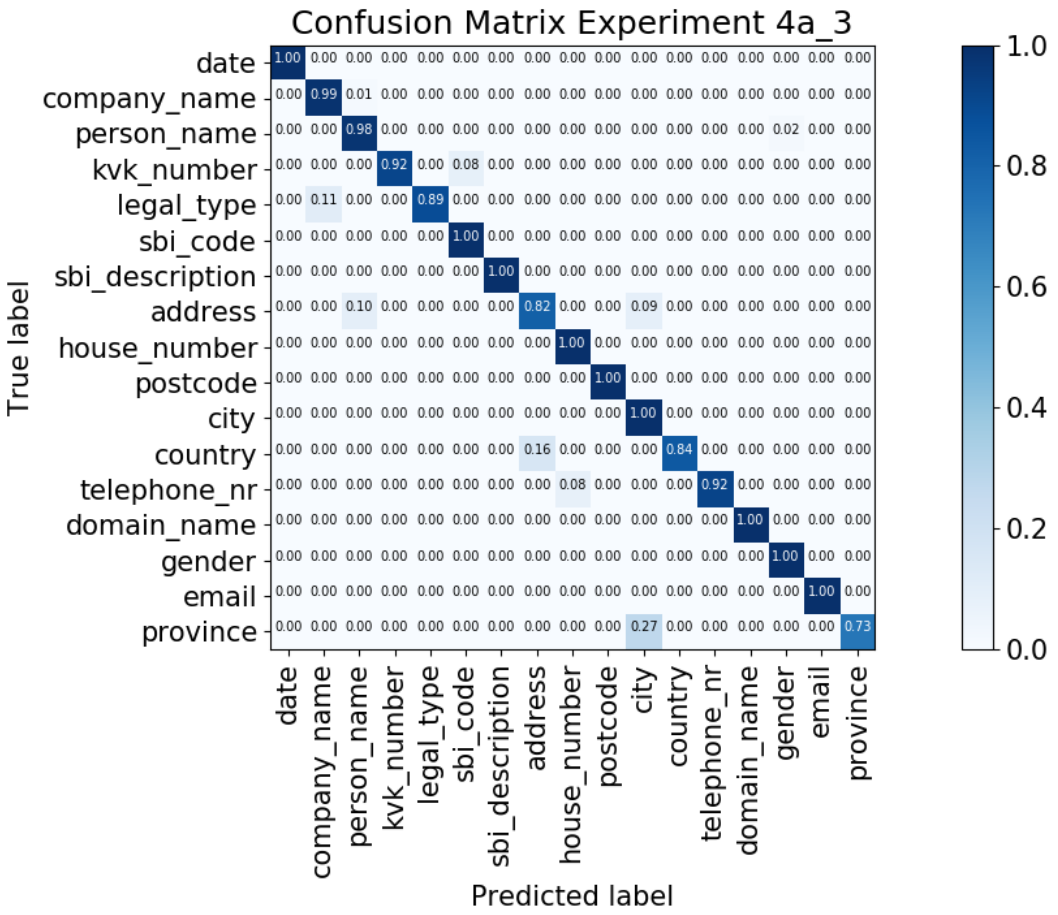
Round 1 Confusion Matrix

Figure 50: Experiment 4.1, Company Info Data



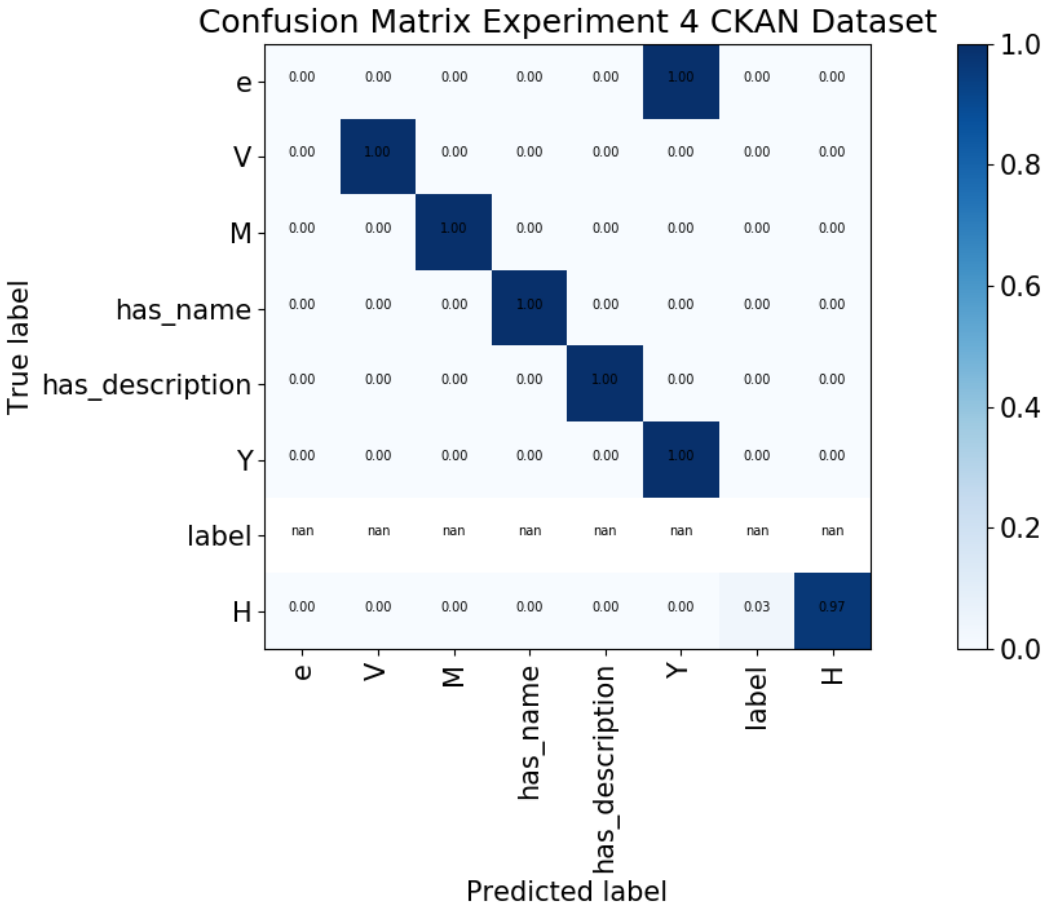
Round 2 Confusion Matrix

Figure 51: Experiment 4.1, Company Info Data



Complete Pipeline Confusion Matrix

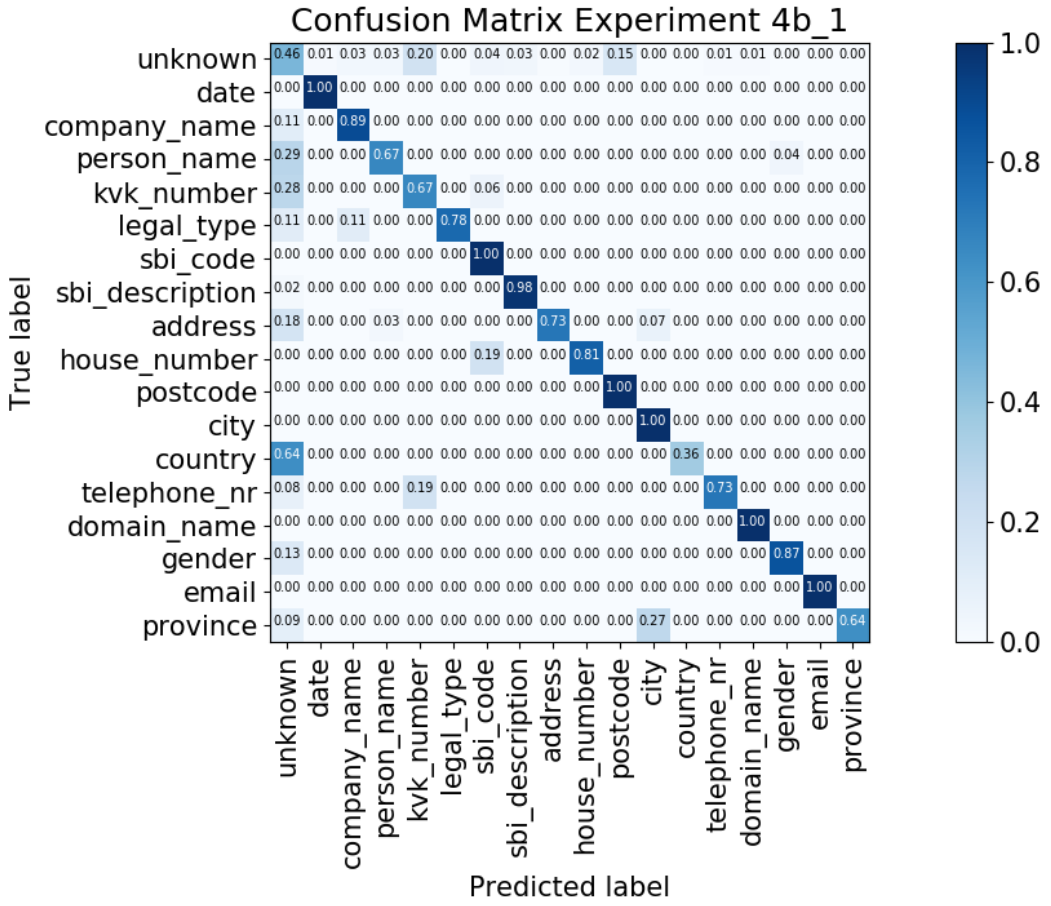
Figure 52: Experiment 4.1, CKAN Data



Complete Pipeline Confusion Matrix

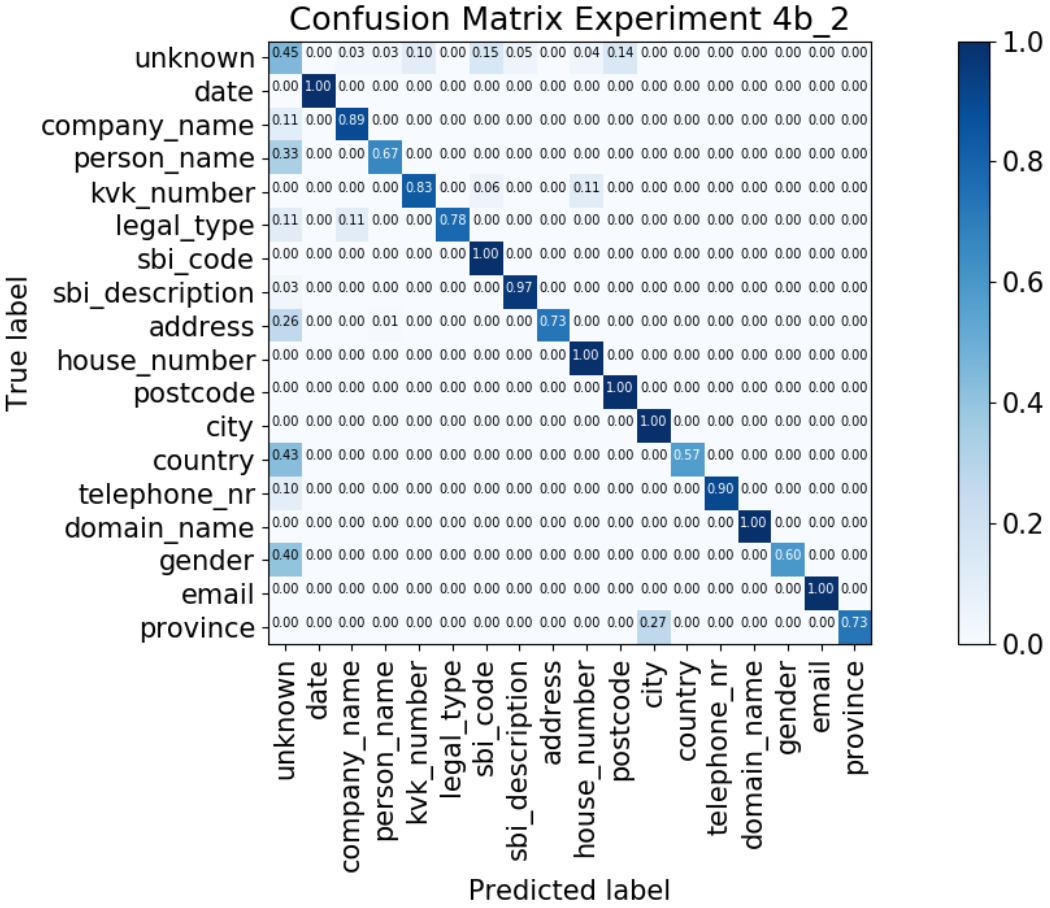
9.4 Confusion Matrices Experiment 4.2

Figure 53: Experiment 4.2, Company Info Data



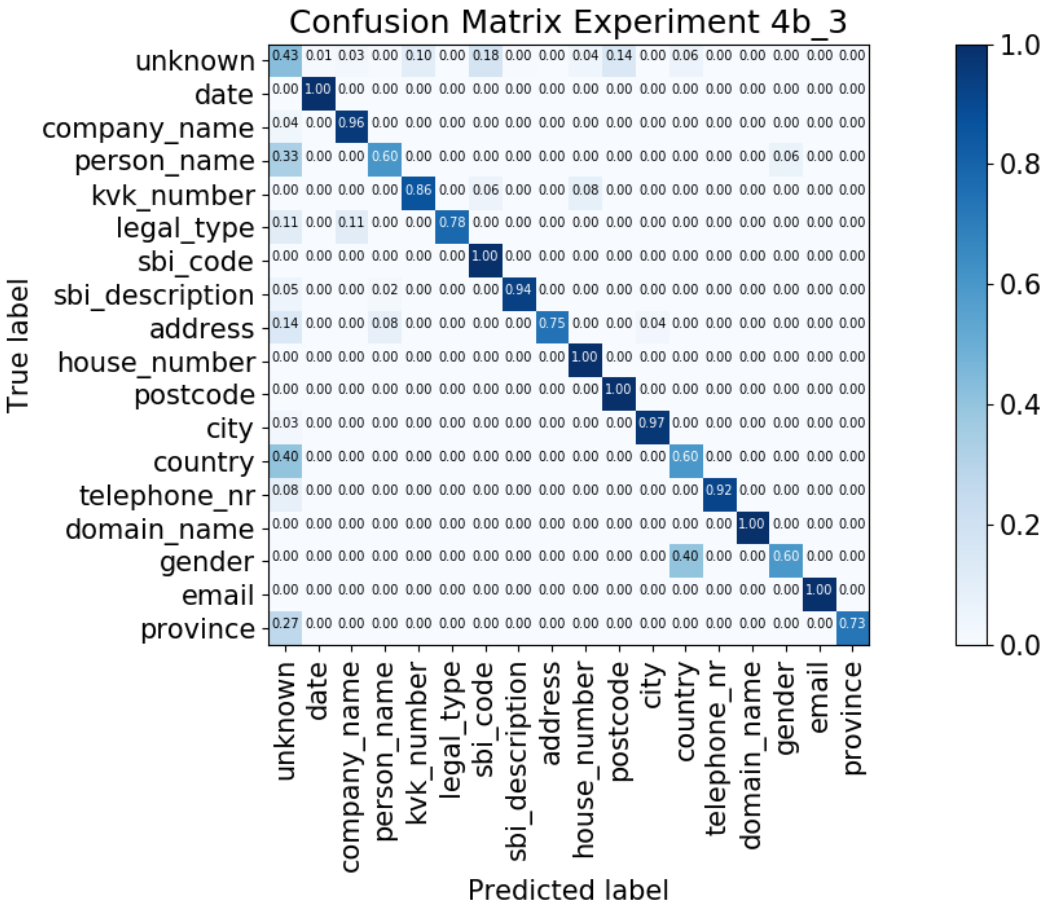
Round 1 Confusion Matrix

Figure 54: Experiment 4.2, Company Info Data



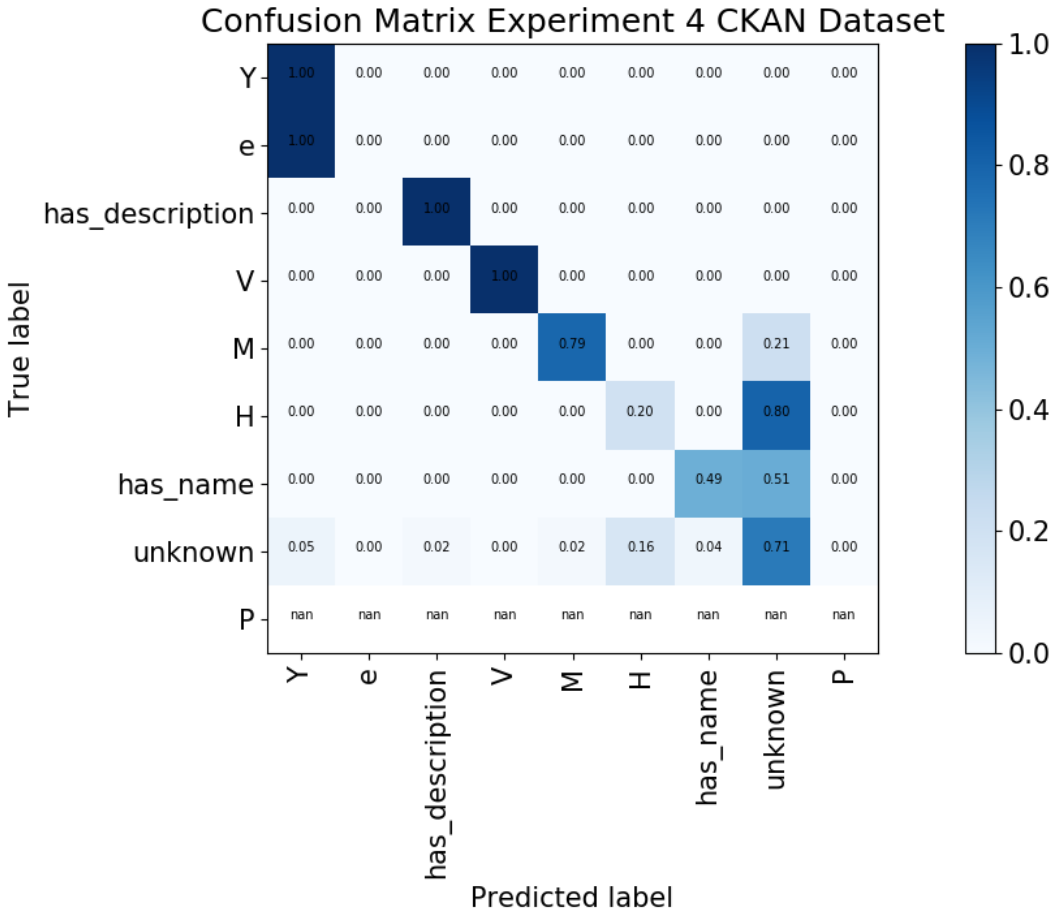
Round 2 Confusion Matrix

Figure 55: Experiment 4.2, Company Info Data



Complete Pipeline Confusion Matrix

Figure 56: Experiment 4.2, CKAN Data



Complete Pipeline Confusion Matrix

REFERENCES

[1] Jacob Berlin and Amihai Motro. “Database schema matching using machine learning with feature selection”. In: *International Conference on Advanced Information Systems Engineering*. Springer. 2002, pp. 452–466.

[2] Philip A Bernstein, Jayant Madhavan, and Erhard Rahm. “Generic schema matching, ten years later”. In: *Proceedings of the VLDB Endowment* 4.11 (2011), pp. 695–701.

[3] *CKAN Wikipedia*. `https://en.wikipedia.org/wiki/CKAN`. Accessed: 2018-12-06.

[4] Michael Collins and Yoram Singer. “Unsupervised models for named entity classification”. In: *1999 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*. 1999.

[5] *Cross-validation*. `https://en.wikipedia.org/wiki/Cross-validation_(statistics)`. Accessed: 2018-18-06.

[6] Hong-Hai Do, Sergey Melnik, and Erhard Rahm. “Comparison of schema matching evaluations”. In: *Net. ObjectDays: International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World*. Springer. 2002, pp. 221–237.

[7] Fausto Giunchiglia, Pavel Shvaiko, and Mikalai Yatskevich. “Semantic schema matching”. In: *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer. 2005, pp. 347–365.

[8] Wen-Syan Li and Chris Clifton. “SEMINT: A tool for identifying attribute correspondences in heterogeneous databases using neural networks”. In: *Data & Knowledge Engineering* 33.1 (2000), pp. 49–84.

[9] David Nadeau and Satoshi Sekine. “A survey of named entity recognition and classification”. In: *Lingvisticae Investigationes* 30.1 (2007), pp. 3–26.

[10] *Parker Convention*. `https://developer.mozilla.org/en-US/docs/Archive/JXON`. Accessed: 2018-12-06.

[11] Fabian Pedregosa et al. “Scikit-learn: Machine learning in Python”. In: *Journal of machine learning research* 12.Oct (2011), pp. 2825–2830.

[12] Erhard Rahm and Philip A Bernstein. “A survey of approaches to automatic schema matching”. In: *the VLDB Journal* 10.4 (2001), pp. 334–350.

[13] Ellen Riloff, Rosie Jones, et al. “Learning dictionaries for information extraction by multi-level bootstrapping”. In: *AAAI/IAAI*. 1999, pp. 474–479.

[14] Carlos N Silla and Alex A Freitas. “A survey of hierarchical classification across different application domains”. In: *Data Mining and Knowledge Discovery* 22.1-2 (2011), pp. 31–72.

[15] Aixin Sun and Ee-Peng Lim. “Hierarchical text classification and evaluation”. In: *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*. IEEE. 2001, pp. 521–528.

[16] Simon Tong and Daphne Koller. “Support vector machine active learning with applications to text classification”. In: *Journal of machine learning research* 2.Nov (2001), pp. 45–66.