UNIVERSITY OF TRENTO - Italy

Department of Information Engineering and Computer Science

Master's Degree in
Computer Science

FINAL DISSERTATION

SOWEEGO:
ensemble learning applied to record linkage

Supervisor:
Andrea Passerini
Marco Fossati

Student:
Andrea Tupini
194578

Accademic Year 2018/2019

Acknowledgements

First of all I want to thank my family for their support and constant motivation. Especially my parents, without who's help I wouldn't be able be here. To my in-laws, for their help, suggestions, and support.

I also want to thank my supervisors, Andrea Passerini, and Marco Fossati, for the invaluable guidance they provided during this whole process.

To my wife, Kate, for her constant encouragement and understanding. For putting up with me and basically keeping me alive this past few months.

Finally, I want to thank all the open source community who made this work possible .

Contents

1 Introduction

In this era of *big data*, there are scores of datasets of every kind, for all sorts of human endeavours. Nowadays, most companies strive to create some sort of dataset of its customers and enlarge it as the years go by, allowing them to better understand their clientele and provide the best service possible. In many domains, however, there is no need to start a dataset from scratch, since much of the information is readily available in public knowledge bases. *Wikidata* is one such knowledge base, and it is the workhorse that powers the *Wikipedia* free encyclopedia, and all of Wikimedia's projects.

From the perspective of Wikidata, having these external datasets consume its knowledge is normal. On the other hand, there might be an external dataset which grows to contain some information that might become useful to Wikidata, since it will allow the enrichment of its own knowledge base. This information is usually manually added by volunteers, but may sometimes be too much, some of it might be overlooked. One of Wikidata's concerns is maintaining the highest quality of data, however, some of the data suffer from lack of references which are invaluable to verify the accuracy of the information contained on their datasets.

The root of the problem may come from the fact that references are only added for a subset of the data, like some of the most visited or sought after articles. The project `soweego` aims to automate this task by automatically adding external references to Wikidata information.

`soweego` is an open source project of the Wikimedia Foundation, which uses techniques of record linkage together with machine learning classifiers to find the external references for Wikidata entities. `soweego` does this by finding potential matches of Wikidata entities in external databases and then ranking the likelihood of these potential matches using machine learning. The external databases `soweego` considers when performing these matches are the following:

- `IMDb`: for information regarding *actors, musicians, directors, producers,* and *writers.*
- `Musicbrainz` and `Discogs`: for information regarding *musicians* and *bands.*

This dissertation aims at improving the prediction power of `soweego`. Specifically, the current set of machine learning classifiers available in the `soweego` project will be improved by finding the best configuration for them. Following that, the idea of *ensemble classifiers* will be explored, in which all of the classifiers available in `soweego` will be joined, and the *opinion* of the whole group will be considered when making a prediction. Motivating this choice is the fact that considering the viewpoint of multiple base classifiers, may smooth out errors that might be produced by any classifier individually.

In chapter 2, the technique of *record linkage* will be explained. This is commonly used when working with a problem where there is a need to find matching entries across different datasets. It will also be shown how the process of performing record linkage can be thought of as a pipeline with different steps, each of which is in charge of performing a different transformation on the data. Common terms that will be used when talking about record linkage, and in this dissertation in general, can be found in the Glossary.

A more in depth description of the problem that `soweego` solves will be presented in chapter 3. This section will also present an overview of the data that will be used, and which areas the current project aims to improve.

`soweego`, as record linkage, can be viewed as a series of steps that are closely related to that of the record linkage framework presented in chapter 2. In chapter 4, an explanation about every part of `soweego` will be given, together with what happens during each part.

The data used in this project will be described in chapter 5. Here the external databases will be explored, centering around explaining its shape, provided fields, and the quality of each database. After this, an explanation of how the data is preprocessed before being consumed by `soweego` is given. Finally, an explanation of each feature extracted from the data is given.

In chapter 6 an explanation of each of the machine learning classifiers provided by `soweego` is given. These will be called the *baseline classifiers* since are what the new *ensemble classifiers* will be compared against. An explanation of the ensemble models used will also be given in this chapter.

Following that, chapter 7 proceeds to show how the *configuration* of the baseline classifiers was optimized, and the evaluation performances obtained by the baseline classifiers, after said optimization, and the ensemble classifiers. An explanation of the metrics used, and a comparison of the results between baseline and ensembles will also be given in this chapter, where the differences between the performances obtained are explored.

Finally, in chapter 8 it will be discussed how ensembles classifiers are better across all considered metrics than almost all of the baseline models. According to the results obtained, ensembles are better than all baseline classifiers at predicting when a pair of records is actually different (e.g. only pairs which the classifier is very sure about are classified as matches). As such, ensemble models are a great choice when they are used in a situation where predicting a negative sample as positive is not desirable. As shown, there are also differences among the evaluated ensemble models, some are more *precise* (they tend to mark as matches pairs which are actually matches), while others are more *inclusive* (they tend to recognize more matches as so, even if some non-matches are also included).

1.1 Contributions

As said in the introduction, the present dissertation centers around contributing to the open source project `soweego`. The main contributions made to the project as part of this work are the following:

- The creation of an *importer* functionality which allows `soweego` to work with data from IMDb. This functionality will be explained in subsection 4.1.1.
- Implement some of the *baseline* algorithms (explained in section 6.1) used by `soweego`. The implemented models are specifically: *logistic regression*, *random forest*, and *single layer* and *multi layer* perceptrons.
- Optimize the *configuration* of the baseline algorithms employing a grid search procedure. This is explained in section 7.2.
- Implement 4 different methods to *ensemble* the baseline classifiers, described in section 6.2.
- Evaluate the performance of the baseline and ensemble classifiers when applied to the current problem. This is done in section 7.3 and section 7.4 respectively.

- Compare the evaluation performance of *ensembles* and *baseline* classifiers to see how their behaviour differs and if there was any improvement when using ensembles (section 7.5).

1.2 A brief overview of Wikidata

This section will give a brief overview of what Wikidata is and how it works. These concepts will be extensively used throughout the discussion.

Wikidata, is defined as the *knowledge base* that supports *Wikipedia* and all of the *Wikimedia Foundation* projects[1]. A *knowledge base* is a collection of statements of and about the world. There are many different knowledge bases available on the web but, *Wikidata* is one of the most complete out there, it also has a very active and extense community of users and volunteers.

In this specific *knowledge base* there are different *items* which are interconnected and integrated with each other. They achieve this through the use of a *Wikidata* construct called *properties*.

Each *item* is identified by using a special tag named `QID` (Q-identifier), it was given this name because each of these unique values starts with the letter *Q*. Information about a specific item (QID) is recorded by the use of *statements* which are *key-value* pairs that contain specific information about each item.

The key of each statement is a *property*, and the value can be a reference to another item via its QID, a string, a number, or media files[2]. Statements can also be extended by the use of *qualifiers* which are *extra information* about the statement. An example of this can be a person who studied at some university for a certain period of time (this duration would be the qualifier).

Properties[3], like items, have their own ID. These are known as `PID` (Property-ID). Some properties may accept only one value or multiple values, and some may also be constrained in some specific way: for example, its value can only be a string composed of one letter.

In Wikidata both *items* and *properties* are associated with a name and a brief description, the main difference is how they are used: items, have the goal of representing a specific entity while properties are used to express information regarding said entity. Another difference between them would be that *items* are displayed on their respective pages on Wikipedia and *properties* are not.

Consider the QID *Q42*, which identifies the writer *Douglas Adams*. Below is a table which shows some of the statements about him using QIDs and PIDs.

PID	QID
P31 (instance of)	Q5 (human)
P21 (sex)	Q6581097 (male)
P19 (place of birth)	Q350 (Cambridge)
P106 (occupation)	Q28389 (screenwriter)
	Q6625963 (novelist)
	Q245068 (comedian)

Table 1.1: Wikidata statements describing Douglas Adams

Here it is possible to see that this way of expressing information is very flexible and allows for a

[1]https://wikimediafoundation.org/
[2]https://www.wikidata.org/wiki/Help:Statements
[3]https://www.wikidata.org/wiki/Help:Properties

quick and rich construction of knowledge.

2 State of the art

This chapter will mainly cover *Record Linkage*, a standard method used to link a pair of databases. The next section will give a brief introduction to the record linkage workflow, and mention some of the most common terms used in this document. section 2.2 will show how record linkage can be thought of as a pipeline, and every step will be explained in depth. Finally, section 2.3 will mention related work.

2.1 Introduction to Record Linkage

Record linkage is commonly used when there is a need for data integration (e.g. merging two datasets). This is achieved by finding pairs of records which represent the same underlying *entity*, and by joining them according to a previously defined set of criteria. It is possible to apply the technique both for connecting a pair of databases (which we call *catalogs* in this use case) and for removing duplicates from one single catalog.

For a more formal description of the problem, suppose there are two datasets (Ψ and Ω). The goal of record linkage is to find all those pairs (x, y) where $x \in \Psi$ and $y \in \Omega$, such that both x and y refer to the same underlying entity. Hence, the algorithm must compare the entities from Ψ and Ω, and decide which of them are a match (i.e., refer to the same underlying entity) and which are not. Record Linkage can also be naturally expressed or viewed as a problem of clustering, in which the objective is to cluster all the entities representing the same *latent entity* together.

The goal, as defined in [34], is to find two different sets M and U, where M is composed of all matching pairs and U contains all non-matching ones:

$$
\begin{aligned}
M &= \{(x, y); x = y; x \in \Psi; y \in \Omega\} \\
U &= \{(x, y); x \neq y; x \in \Psi; y \in \Omega\}
\end{aligned}
$$

The problem of joining different catalogs is trivial when the catalogs have a shared, unique identifier. For example, when joining the datasets of customers from two banks, and each customer is associated with their government ID, or some other unique identifier. In this case, it is just a matter of matching on this attribute and merging the resulting pairs.

However, it is commonly the case (as is the case for this project as well), that catalogs do not have a common unique identifying attribute, thus requiring a less trivial approach.

As shown later in this chapter (subsection 2.1.1), there are two main methods of performing record linkage, but basically, they can be separated into methods which match directly comparing pairs of fields, and methods which measure the similarity between said pairs.

Besides not having a unique identifier, it may also be the case that the datasets that need to be merged, have some mistakes. This is something especially noticeable when we are working with data representing people. These are some of the mistakes that might come up:

- The names were misspelled.
- The first and last names are reversed.
- The format of the date is not standardized.
- The dates presented are simply not possible since they are too forward in the future or too far in the past.

Many of these can be somewhat reduced by preprocessing the dataset, but others may not, and it is necessary to account for them when working with the data.

To give an example of a situation in which *record linkage* would be used, suppose there are two databases of clients that need to be joined for some company. The tables below represent said databases.

First Name	Last Name	Birth Year
Brenda	Williams	1990
Meng	Yu	1950
Gary	Crowley	1956
Bruno	Lima	1956

Table 2.1: Example catalog from company #1

First Name	Last Name	Birth Year
Emma	Farr	1971
Yu	Meng	1950
Juri	Petrov	1932
Gary	Crowley	1956

Table 2.2: Example catalog from company #2

The catalogs above do not have a unique identifier that unequivocally links them, so it is necessary to check the available attributes. The fields can be compared in turn and if they all match then it can be said that two entities are the same. For example, it can be easily seen that *Gary Crowley* is in both datasets since an entity with said name exists in both and their birth dates match. On the other hand, nothing can be confidently said about the other possible matches since none of them all the fields match. There is a pair (*Meng Yu* in the first dataset, and *Yu Meng* in the second) which might be the same person since the names are similar and they share a birth year. For this last instance, there is no way to be sure if the first and last names are reversed in one of the datasets or if they are actually different people. This kind of situation is quite common in real-world datasets.

2.1.1 Main Approaches

There are two main approaches[1] to consider when talking about record linkage: 1) *deterministic*, and 2) *probabilistic*.

The main difference between these is how the pairs of entities are compared. A more thorough description of each method follows.

[1]Besides these two approaches, in the record linkage literature we commonly find the term *clerical*, which entails *manual* work, i.e., by a clerk.

Deterministic

In deterministic record linkage, all fields are compared to see if they match. If the fields do match, then it can be said that a pair of entities are the same. However, if any of the fields that are being compared only partially match or do not match at all, then the pair is considered as a non-match.

Take for instance the task of joining a catalog that provides information regarding where someone lives with another catalog that provides their age. Both catalogs only contain the name of the person as an identifying feature, so a composition of these two fields (attributes) can be used to identify each entity.

First Name	Last Name	Age
Sarah	Miller	42
John	Piers	35

Table 2.3: Deterministic example, catalog 1

First Name	Last Name	Address
Sarah	Miller	New York
Johnny	Piers	Berlin

Table 2.4: Deterministic example, catalog 2

After matching these results, there is a clear match for *Sarah Miller*, but no available match for *John Piers* and *Johnny Piers*, since the names are different and there is no way to be sure if *Johnny* is just a nickname (making them the same person) or if these two entities are different people.

Note also that in this example both rows coincide completely on the *Last Name* field. However, when doing deterministic record linkage partial matches are not considered [30], because there can be no uncertainty in the match.

It should be noted that this linking procedure is not optimal when the provided data is not very clean. Preprocessing steps can be taken to ensure that the catalogs are as uniform as possible, but even so, it is never easy to know if there might be a match missing or not.

Probabilistic

In probabilistic record linkage [34, 55, 63] all fields are compared, taking into account how good is each field at telling if a pair of entities is a match or not. This is done by using a weight w_i for each field.

The value of this weights is derived, for a pair of entities r, from the probabilities u_i and m_i. Where u_i is the probability that field i matched and that $r \in U$ (the pair is not a match), and m_i is the probability that the field i matches and $r \in M$ (the pair is a match).

For example, if entities on the dataset currently being worked with have a *weekday* field then it can be said that this field has 7 possible values, one for each day of the week. Moreover, if every day of the week appears in the datasets approximately the same number of times, then the probability that for a random entity to have a specific weekday is 1/7, and this is the $u_{weekday}$ probability for that field.

On the other hand, considering that the data has a perfect quality, if two entities are the same then this it is 100% probable that their weekday field matches. But if the dataset has some error, say that the *weekday* field is wrong 4% of the time, then the probability that a pair $r \in M$ agrees on field *weekday* is 96%. This values are rarely known beforehand and may be estimated from the dataset [15].

The weight w_i for each field is then calculated as:

$$w_i = \begin{cases} \log_2(\frac{m_i}{u_i}) & \text{if } a_i = b_i \\ \log_2(\frac{1-m_i}{1-u_i}) & \text{if } a_i \neq b_i \end{cases}$$

Where a_i and b_i are the field i of the entities $(a, b) = r$. If the fields match then the first case is taken, otherwise the fields don't match and the second is taken. The sum $W = \sum_{i=1} \hat{} N w_i$ for every field in the pair can then be taken and compared against a set of upper t_u and lower t_l thresholds to know if the pair is a non-match, a match, or a potential match. These thresholds need to be determined a priori.

$$\begin{cases} \text{non-match,} & \text{if } W \leq l_l \\ \text{potential-match,} & \text{if } l_l < W < l_u \\ \text{match,} & \text{if } l_u \leq W \end{cases}$$

An extension to the probabilistic record linkage method presented above is that instead of computing a weight w_i for each pair of fields i, what is done is that this is instead substituted for a similarity metric that says how similar the fields are [58, 71]. This metric is obtained by a *comparison function*, which takes a pair of fields and tells, in percentage, how similar they are. One naive example of such a comparison function is the percentage of characters that two texts have in common.

Following the example catalogs defined in the previous section, now it is possible to compare the *First Name* and *Last Name* fields separately. Suppose they are compared with the function mentioned above that provides the percentage of letters both fields have in common.

$$fc(s1, s2) = \frac{\|s1 \cap s2\|}{\|s1 \cup s2\|}$$

It might be that both last names are exactly the same, as is with the first name for *Sarah*. However, with the first name for *John* it would show a similarity of only 0.8 with *Johnny*. Taking both the similarity given for the first and last names then it is possible to make a better informed final decision on whether both records should be marked as a definite match or not. For instance, it may be said that if the average similarity assigned to all fields is greater than 0.5 then it may be stated that a pair is a match. Under these assumptions *John Piers* and *Johnny Piers* will be classified as the same person.

The probabilistic record linkage approach is the one that will be implemented in this project.

2.2 Record Linkage as a Workflow

An abstraction that is commonly used when performing record linkage is that of viewing the problem as a workflow [15], where each step is in charge of performing a different transformation to the data. In probabilistic record linkage [34] the resulting matches are usually segmented into three sets depending on how sure the probabilistic model was that a pair was a match or not. For example, pairs for which the certainty of the model falls in the interval $[0, 0.4]$ are *non-matches*. Pairs whose certainty of a match is in $[0.4, 0.6]$ are *potential matches*. And those whose certainty is in the range $[0.6, 1.0]$ are *matches*.

Figure 2.1 shows a graphical representation of said steps.

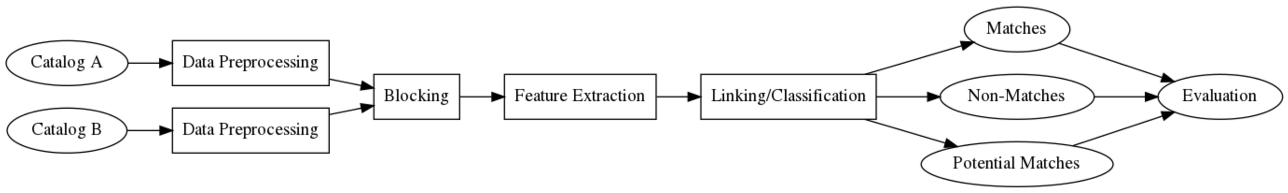


Figure 2.1: Record linkage workflow.

This process is separated into five steps and it may be applied to n input catalogs, they all need to go through the *Data Preprocessing* step so they are as uniform and clean as possible.

In the following sections a more in-depth description of each step will be provided.

2.2.1 Data Preprocessing

This is the first and most significant step in which the data is preprocessed. During this step, the catalog will be taken in and its data will be rearranged and *cleaned up*. The actual meaning of this depends on the kind of data that the specific catalog contains and it is mostly up to the designer of the system to decide in which way the presented data should be preprocessed [60]. However, some of the most common cleaning procedures which are almost always done are:

1. Normalize the case of the text (e.g. make sure all text is lowercase).
2. Remove invalid characters for the current encoding.
3. Standardize all characters (e.g. transform accented characters into their *non-accented* equivalent).
4. Ensure all dates comply with a given format, such as *DD/MM/YYYY*.
5. Remove empty values or fill them with a default value.
6. Standardize the *gender* representation. (i.e. either *m* or *male*)
7. Normalize names and addresses. [17]

The goal of this step is for the catalogs and their data to become as similar as possible amongst themselves, so that there will be no need to deal with the potential differences between them once the matching process has begun.

It has also been shown [19] that a well made data preprocessing step is necessary for the record linkage algorithm to have positive performance results.

For example, consider that the following data is available in some input catalogs. Note that in this step no distinction has been made on whether the catalog is a source catalog or not, so in this specific example the `Catalog ID`/`Target ID` fields will be omitted.

First Name	Last Name	Birth Date
Анника	Цлаире	12 Sep. 1948
Abélia	Benoît	14/4/1987
Dave	Smith	7/28/1976

Table 2.5: Data preprocessing example: dirty catalog.

Before proceeding it is necessary to clean this data. Specifically, since the first entity has its first and last names written with Cyrillic letters, and the date format is different from what is expected. The second entity's first and last name use accented letters. And finally, the date of the third entity also does not respect the stated date requirements (the month and day are reversed).

After passing the input catalog through the data preprocessing step the following output will be obtained:

First Name	Last Name	Birth Date
Annika	Claire	12/9/1948
Abelia	Benoit	14/4/1987
Dave	Smith	28/7/1976

Table 2.6: Data preprocessing example: cleaned catalog.

Notice that now all the entities conform with the required format. Once all the catalogs have been cleaned they may be sent over to the next step of the workflow (*blocking*).

2.2.2 Blocking

Once the cleaned entities from the previous step are available, it is necessary to define which of them should be compared with one another. By default, the pairwise comparison will be performed over all entities from all catalogs. However, this can quickly become infeasible when catalogs start getting bigger, or when their number starts to increase.

Also, if the dataset being utilized has many different entities then when comparing all entities against each other it will show that most of these are not matches.

It is generally the case when comparing all entities with each other that the number of comparisons needed increases quadratically with the size of the catalogs, while the number of true matches increases linearly [15].

Blocking (also known as *indexing*) is defined as comparing a specific entity only against entities that might be a match. When matching said entity against a dataset, it is possible to use a simple rule to define a subset of all entities composed only of those which the rule suggests are possible matches.

This new set of entities made to be matched against is not guaranteed to contain any match, nor it is guaranteed that none of the entities in the set which are an actual match were left out, but it entails a much smaller number of comparisons. If a good blocking rule is chosen then the record linkage process becomes much more efficient.

Suppose there are two catalogs, Ψ and Ω, and one specific entity is being considered $x \in \Psi$ and there is a need to know which entities in the catalog Ω match with it. Rather than comparing x with the whole Ω dataset (which would result in $\Psi \times \Omega$ comparisons), what must be done is derive a subset of possible matches using a blocking rule r, which takes a pair of entities and shows if they might be a *match* according to the rule or not.

$$blocked = y; r(x, y) = 1; y \in \Omega;$$

There are many possible blocking rules that can be used [16], ranging from the very simple *exact string match*, to the more complex phonetical q-gram comparison of text. A more complex algorithm may be more precise than a simpler one, but is usually slower. For example, a metric that compares multiple subsets of a text may be resistant to minor spelling errors, while one that compares the string character by character may not. However, the latter is less computationally expensive to perform.

These rules are applied to a pair of fields from two different entities. For example, it is possible to apply an *exact string match* rule to the *name* field of two entities and see that they match if their names are exactly the same. This simple operation makes it easy and fast to segment the dataset into clusters of possible matches.

The decision regarding the *rule* used for blocking is part of the record linkage problem, and it needs to be simple enough so that performance does not suffer, but good enough so that the resulting set of potential matches includes real matches.

It is also possible to freely use multiple blocking rules to get multiple sets of possible matches and then use the union of these to do the final comparisons.

The concept of *blocking* has been used in record linkage almost since the start of the field [34], and it is now an integral part of most record linkage solutions [72]. The field of blocking is a big area of research in and of itself, with many different methods proposed over the years [1, 16].

However, the evaluation of these methods is outside the scope of the current dissertation and it will be limited to work with one of the most basic blocking rules, which is a blocking technique that divides two strings into their corresponding tokens, and if any of these match then the pair is considered to be a potential match.

For now, only blocking for fields which are strings has been discussed. But blocking can be applied to many other fields as well (although strings are the most common ones). For example, other possible ways in which blocking could be applied are:

- Gather all entities with a specific *zip code*.
- Gather all entities which were born on a specific *year*; etc.

Blocking is quite flexible and it is up to the designer how strict he or she wishes to make it. Although it is agreed [16] that it is better to have a more *lax* blocking technique which includes entities that are not a match and later filter this out during the actual *linking* step (see subsection 2.2.4), than using a more precise blocking but which leaves out potential matches.

In other words, when choosing a blocking rule its preferable to choose one which tends to include more entities than necessary, than having one which only includes a few non-matches. The choice of this trade-off needs to be done on an per problem basis.

As an example suppose the following two catalogs are available. They both contain the first and last names and the zip code of each person. The first also contains whether said person is *male* or *female*, while the second catalog contains the *academic title* of each person.

Source ID	First Name	Last Name	Zip Code	Sex
A1	Anna	Claire	27321	f
A2	Claudio	Bari	34918	m

Table 2.7: Source Catalog: Blocking example.

Target ID	First Name	Last Name	Zip Code	Education
B1	Annika	Claire	27321	Ph.D.
B2	Claude	Bari	34918	Bachelor
B3	Jane	Stel	34918	Bachelor

Table 2.8: Target Catalog: Blocking example.

Suppose the goal is to perform blocking on the *Zip Code* field, and to achieve that, *exact match* strategy is used. It is possible to see that in total there are only two zip codes (*27321* and *34918*), so after blocking there will end up being two sets. The first one (for zip code *27321*) will contain the pair $[(A1, B1)]$, both of which could arguably be the same person. In the second set, the remaining pairs $[(A2, B2), (A2, B3)]$ can be found.

For the first set its only necessary to compare the source entity *Anna Claire* with *one* target entity. In the second case, the source entity *Claudio Bari* must be compared with two target entities. In both cases, the total number of comparisons for a source entity is less than what would be needed without blocking (3 comparisons).

The gain in the example above doesn't seem like much, but it is easy to imagine that this procedure would be very effective at reducing the total number of comparisons once the number of zip codes and, by extension the amount of sets generated by blocking, start to increase in number.

In short, the goal of blocking is to reduce as much as possible the number of comparisons that each entity will be subject to by using a *blocking rule* which finds the best potential matches for a said entity in an efficient way.

2.2.3 Feature Extraction

The feature extraction step is in charge of taking a pair of entities and extracting from them a set of *features*, which characterize how similar the entities in said pair are. The pair of entities for which the features are extracted are those pairs which the previous *blocking* step said may be good candidates for a match.

As mentioned before, each entity is composed of a series of fields. By comparing said fields, and extracting a value from each comparison, a new vector can be generated. This vector, called *feature vector*, characterizes the similarity between each field of the pair of entities.

The comparison itself is made by applying a function to a given pair of fields.

$$feature_i = f(entity_y.field_i, entity_z.field_q)$$

The function f is a *comparison function*, which shows the similarity, or distance, between two fields of the same kind (for example, the *name* field of both entities). As such, the decision of which functions to select is part of the record linkage problem, and the designer is free to use as many as needed (i.e. if desired the designer could compare two textual fields from a pair of entities using multiple string similarity functions).

Some of these comparison functions yield a percentage that describes how similar two specific fields are. Others only provide information on whether or not two fields are the same or not. The latter are referred to as **exact match** functions.

When one of the fields being compared has no value then the function treats the pair as if it was not a match and returns a similarity of zero.

For example, one such comparison function can be the *Levenshtein distance* [49], which provides the difference between two strings. For this specific case it will be used to see how similar two strings truly are (this similarity metric will be explained more in depth in subsection 5.3.6).

Actually, the `Levenshtein` function will show a distance that increases the more two strings are dissimilar, which is not the desired behaviour. What is needed is for the result to be 0 when the strings are dissimilar and 1 when they're the same. So the distance function will need to be adapted to be the 1− [this value divided by the length of the longest string[2]]. Basically, the new function is just the complement of the percentage of the maximum value that can be returned by `Levenshtein(a, b)`.

$$lev(a, b) = 1 - \frac{Levenshtein(a, b)}{max(|a|, |b|)}$$

To continue with the example from the last section, consider this Table 2.7 as the source catalogue, and Table 2.8 as the target catalogue. Also suppose that the second set yielded by the blocking procedure above $[(A2, B2), (A2, B3)]$ is being used. Applying the `lev` distance mentioned above to the first and last name fields will yield the following results:

Pair IDs	Lev(First Name)	Lev(Last Name)
(A2, B2)	0.714	1.0
(A2, B3)	0.142	0.0

Table 2.9: Example features obtained using adapted Levenshtein distance.

As expected, the names *Claudio* and *Claude* are quite similar, and their last name is an exact match. On the other hand *Claudio* and *Jane* are very dissimilar, with their last name (*Bari* and *Stel* respectively) being as different as possible.

In this example, only two fields were used and one comparison function was applied, in practice however more fields and potentially many different comparison functions can be used for the same fields.

2.2.4 Linking/Classification

In the *linking* (also called *classification*) step, the list of features extracted in the previous part is consequently fed to a decision model that takes the features and then decides whether said pair is a match or not.

There are many decision models that can be used for this step [38]. The *go to* model for record linkage is the basic probabilistic model defined in [34]. Other possible models [32] include using the *k-means* [41] clustering algorithm to cluster the feature vectors into *matched*, *unmatched* and *potential match* clusters, or if there is labeled data available then it is possible to train a supervised machine learning algorithm to learn when to classify feature vectors as a match or not.

Regardless of the model used, the output of this step can be either two sets (*matches* and *non-matches*) or three sets (*matches*, *non-matches*, and *potential matches*). Usually, the set of potential matches is reviewed manually to define which are actually a match [30].

To follow with the previous example, consider as input to the linking step the feature vectors in Table 2.9. And suppose that as a decision model a simple rule-based model is being used:

[2]which is also, incidentally, the maximum value the `Levenshtein` function can return

$$decision\_model(f_i) = \begin{cases} \text{match}, & \text{if } mean(f_i) \geq 0.5 \\ \text{non-match}, & \text{otherwise} \end{cases}$$

Where f_i is one of the feature vectors. Applying this model to the incoming feature vectors would yield the following results:

Pair IDs	$mean(f_i)$	Model's decision
(A2, B2)	0.857	match
(A2, B3)	0.071	non-match

Table 2.10: Example of the model's decisions given the input feature vectors in Table 2.9.

With these results it is now possible to merge the corresponding entities (in this case $(A2, B2)$). Just to clarify, the procedure to join the data in the pairs predicted as *matches* is not part of the record linkage process.

2.2.5 Evaluation

As the name suggests, in the evaluation step, the evaluation of the record linkage procedure performance is made. Record linkage can be thought of as a supervised machine learning problem. As such, its results are evaluated like those of any other supervised algorithm.

To do this its necessary to have some previously labeled data that may be used to calculate the metrics given the prediction of the model. This *data* is nothing more than the labeled pairs of entities that say whether they are actually a match or not[3], and it should be different from the one that's currently being classified.[4].

The metrics considered when evaluating a record linkage pipeline are the commonly used [59] *precision*, *recall*, and F_1-*Score*.

Before defining the above metrics more in depth, its necessary to first define the concept of **true/-false positive** and **true/false negatives** from which they are derived.

		True Conditions	
		Real Positives	Real Negatives
Predicted Conditions	Predicted Positives	True Positives	False Positives
	Predicted Negatives	False Negatives	True Negatives

Table 2.11: Confusion Matrix: Definition

True Positives (TP) are the elements predicted as *positive* and are labeled as *positive* in the evaluation data.

False Positives (FP) are the elements predicted as *positive* but are really labeled as *negative* in the evaluation data.

False Negatives (FN) are the elements predicted as *negatives* which are really labeled as *positive* in the evaluation data.

[3]same as the *training set* when referring to supervised machine learning
[4]*classification set* in supervised learning

True Negatives (TN) are the elements predicted as *negative* and are really labeled as *negative* in the evaluation data.

The *counts* of the different clusters defined above are what is used to compute the metrics of *precision*, *recall*, and *F1-Score*. All of the employed metrics have results in the range $[0, 1]$, where 0 is the worst score and 1 the best.

Precision

Precision is a metric that measures the percentage of *True Positives* amongst all the elements which were predicted as *positive*. The less *negative examples* are miss-classified as *positive*, the higher the *Precision* will be. It is defined as:

$$Precision = \frac{\|TP\|}{\|TP + FP\|}$$

Recall

Recall, on the other hand, measures the percentage of *positive* values which where actually predicted as *positive*. The less *positive* examples are miss-classified as *negative*, the higher the *Recall* will be. It is defined as:

$$Recall = \frac{\|TP\|}{\|TP + FN\|}$$

Recall alone is not an informative metric, since achieving a high recall is quite easy: just classifying everything as *positive* will yield a perfect *recall*. For instance, if all samples are predicted as *positive* then $FN = \varnothing$, meaning that $\frac{\|TP\|}{\|TP\|+0} = 1$.

F_1-Score

The F_1 score is the *harmonic mean* of the *precision* and the *recall*. It is 1 when both *precision* and *recall* are 1, and it is 0 when they both are 0. This metric is commonly used to join its two component metrics into one. It is defined as:

$$F_1 = 2 * \frac{Precision * Recall}{Precision + Recall}$$

Even though the F_1 metric is commonly used for evaluating machine learning problems, its usage for record linkage has been criticized [39] since it presupposes that both *precision* and *recall* are of equal importance. In some scenarios, be them record linkage or other machine learning problems, it might be that a higher *recall* is more valued than a higher *precision* (or vice versa). In this cases the F_1 metric should be used with care, and analyzed together with *precision* and *recall*.

2.3 Related Work

This section shows related work which might be relevant to the present dissertation. Of special interest are any applications of machine learning for record linkage problems, and especially works that have successfully applied *ensemble* classifiers in a record linkage pipeline.

In [14], an SVM is trained as the model which distinguishes if two entities are the same or not given their comparison vectors. In this work, the authors start by identifying *clear match* examples, which are those whose elements in the comparison vectors are near to 1. They then proceed to train the SVM on these *clear matches*. The rest of the data then gets processed, and a prediction for each pair is obtained. Those pairs which are confidently predicted as matches are used, together with the original training data, to train a new SVM, iterating this process multiple times.

In [31] different classification models are tested for unsupervised record linkage. Specifically the comparison was made considering *naive bayes*, *support vector machines*, *decision trees*, and *bayesian networks*. After the comparison the result was that among these, *SVM* and *naive bayes* classifiers performs best. The way in which the training is done is similar to [14], where the set of clear entity pairs are chosen as initial training set, and then the classifier is trained in an iterative way, each time including in the training set the top predictions made during the previous generations.

In [70] a *single layer perceptron* is used to separate matching pairs from non-matching pairs. The authors report using this classifier outperforms the basic probabilistic record linkage approach.

Instead of using a simple comparison function, in [37] a convolutional neural network (CNN) is used to extract how similar two strings of text are. The authors compare this with the *Jaccard* text similarity metric and find that a record linkage pipeline using the CNN extracted comparisons has a better performance thank one using Jaccard similarity.

In [21], a record linkage pipeline is built that leverage a decision tree classifier to derive a series of rules to classify a pair of entities by looking directly at the fields of said entities.

The TAILOR framework was developed in [32] and provides a series of tools to perform record linkage. For using unlabelled datasets the authors propose clustering the unlabelled pairs of entities based on their comparison vectors into three clusters (matched, non-matches, and potential matches). The elements of the *matches* and *non-matches* clusters are then used to train a decision tree. The authors found that their method had better performance than the standard *probabilistic record linkage* with thresholds.

In [46], the goal was to find, in a database of patents, all those entities which are the same. The *DBSCAN* clustering algorithm is used to perform the blocking, and then the entities are fed to a *random forest classifier* to predict if they are a match or not.

An ensemble composed of an *SVM*, *logistic regression*, and *neural network* is made in [68]. This ensemble is used as a classifier in a record linkage pipeline that needs to find the information of unique entities across multiple databases of medical domain. In this work, the authors optimize the base algorithms by means of a grid search procedure, and report that the ensemble has a better performance than any of the base classifiers when these are evaluated individually

3 Problem Statement

Wikidata is a huge repository of knowledge which people from all over the world consume. However, a big part of the information it contains can not be verified since it is missing references to external, and trusted, sources of information. These references are essential to ensure that the content is of the highest quality.

Currently, the latest available snapshot in the Wikimedia Grafana interface[1], states that there are a total of *744.4 million* **statements** in Wikidata. Of these, a full *200.5 million (26.89%)* is unreferenced, which represents a large chunk of the total statements. This means that a sure way to improve the quality of Wikidata is to develop a way in which these statements can be populated with its respective reference.

`soweego` is a great answer to this problem since it strives to automatically link Wikidata with external catalogs, making its content more trustworthy. Even though the project has done great contributions by constructing a machine learning based pipeline to automatically link entities, there is still some space for improvement. Specifically, `soweego`'s pipeline allows for only one machine learning algorithm to operate at a time. This means that, when executing the pipeline, a single algorithm needs to be chosen. This is not an easy task since not all algorithms perform optimally on all catalogs. Meaning that a trade-off must ultimately be made.

`soweego` may benefit from considering the opinion of more than one algorithm at a time since it allows the final result to be derived from a wider spectrum of information provided by a diverse set of classifiers.

3.1 Use Case

As a starting point, `soweego` currently focuses on the *people* domain. At the time of this writing, the *Wikidata Statistics* page[2], states that there are a total of 61 million entities, of which 5.2 million are people (roughly 10% of all entities), which is a large portion of all entities.

Note that even though the development is centered around *people* the idea is to develop a flexible system that can easily be extended to include external catalogs containing different kinds of entities and not exclusively centering on people.

For the purpose of this dissertation the following ones were selected as a starting point[3]. Note that this section is limited to mention the catalogs and their entities, and a more in depth explanation of each will be done in section 5.2.

3.1.1 Internet Movie Database (IMDb)

IMDb[4] provides a complete and up to date list of information about movies and the people that appear in them. They provide a dump of their data which is free to use, although somewhat limited on the information it contains.

The *people dump* from IMDb states the three primary professions of each person, which are the three main jobs for which they have appeared in credits of movies.

This information may be leveraged by separating the incoming data into their corresponding professions. These professions will come to be the *type of entities* that are provided by the external catalog. In IMDb, these professions are not mutually exclusive, so there might be a person which is an **actor** and also a **musician**. The following list contains all the types of entities extracted from IMDb:

- Actor
- Director

[1]Snapshot for *2019-09-05* can be seen at https://grafana.wikimedia.org/dashboard/snapshot/jWeAFAgOQczHXK62h7rXvJPH2Zh4D609

[2]https://www.wikidata.org/wiki/Wikidata:Statistics

[3]https://meta.wikimedia.org/wiki/Grants:Project/Hjfocs/soweego/Timeline#Monthly_updates

[4]https://www.imdb.com/

- Musician
- Producer
- Writer

IMDb's dump does not provide such a clean separation of the different types of entities. This clear segmentation was the result of an analysis and conversation made by the whole development group[5] with the scope of finding the best matches between the IMDb professions and Wikidata entities.

IMDb's data is mostly provided by contributors. However, information like *name*, *description*, *image* (either the profile image of a person or the cover for a movie/TV-Series), are manually curated by the IMDb team. Automated access to the data itself is, as said above, restricted since IMDb only provides a partial dump of their whole database, and using web crawlers to access more information is not permitted according to their terms of usage.

3.1.2 MusicBrainz

MusicBrainz[6] is a music encyclopedia, which contains metadata about music, musicians, and bands. Note that in this chapter's introduction it was mentioned that `soweego` works mainly with people, however now it is possible to see that it also includes *musical bands*. This is because *bands* have very similar fields to people, and it is an example of the project's flexibility.

The MusicBrainz database is collectively maintained, with the help of the Musicbrainz team. All of Musicbrainz data is in the public domain, and as such, it is free to use.

The types of entities utilized for this project from the Musicbrainz database are:

- Musician
- Band

3.1.3 Discogs

The **Discogs**[7] database is also maintained in a crowd sourced way. It contains information about musical records, musicians, and bands. The database is in the public domain, so it can be used freely.

From the Discogs database the information retrieved are the following entity types:

- Musician
- Band

4 Approach

This chapter will cover the architecture of `soweego` project, and a description of each of its parts, and how it relates to the record linkage *workflow*.

[5] https://github.com/Wikidata/soweego/issues/165
[6] https://musicbrainz.org/
[7] https://www.discogs.com

4.1 System Architecture

soweego presents a pipeline though which data flows, and is composed of a series of steps which closely resemble the record linkage workflow (explained in depth in section 2.2). The pipeline is executed once for each *concrete entity type*[1].

Specifically, the soweego pipeline is structured as follows:

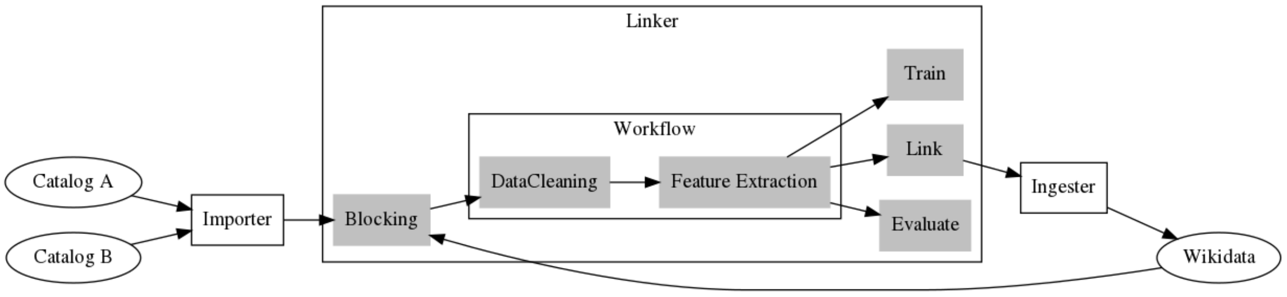


Figure 4.1: Structure of soweego. *Catalog A* and *Catalog B* represent external databases which will be imported into *Wikidata*.

A section will be dedicated to each step, but just to give an overview, below is a brief description of each one. To clarify the figure above, the *Workflow* is a module that contains functionality for data cleaning and feature extraction. And the *linker* is a module which contains functionality for blocking, the *workflow*, and *train, link, evaluate*.

1. Importer: reads the data from both the external databases (target catalogs) and Wikidata (source catalog),

2. Blocking: takes the preprocessed data and clusters each source entity on the Wikidata catalog with potential matches in the target catalogs.

3. DataCleaning: Takes the pairs of potential matches and cleans them (the *Data Preprocessing* step in the record linkage workflow).

4. Feature extractor: takes the cleaned pairs from *DataCleaning* and extracts a number of features, as previously defined in subsection 2.2.3.

5. Train, Evaluate: train or evaluate the performance of a machine learning algorithm given the features extracted in the previous step.

6. Link: uses a previously trained machine learning algorithm to predict the final labels on some classification data. These predictions are done on features extracted from said data.

7. Ingester: is in charge of taking the predicted matches and uploading the confident predictions (those above a threshold) to Wikidata, while potential matches are uploaded to the Mix'n'match[2] platform to be reviewed manually by the community.

The current work will be mainly centered around the development of the linker modules and small parts of the importer. Although for the discussion of results (chapter 7) only the results of evaluate

[1]Every different combination of *catalog* + *entity* is a concrete entity type, since each represents a different entity type with different provenance. As seen later, each is saved as a different *model* in soweego's internal database

[2]https://tools.wmflabs.org/mix-n-match

and `link` module will be evaluated. The remaining modules were developed by the rest of the team working on `soweego`[3].

4.1.1 Importer

The importer is the first step in the pipeline, and is also the one that is in charge of taking an external source of data (for example, a `csv` file), and then interpreting it, and saving the results to `soweego`'s internal database so that it can be easily consumed by the following steps.

The importer is composed of one *adapter* for each external catalog. The adapter is the *importer* for a specific catalog and is in charge of downloading the latest available dump for the catalog.

Once this is done the adapter reads the obtained data and prepossesses it, so that it can be imported into the database. The adapter is necessary because the database dump from each catalog contains different fields, in different formats which need to be converted to a common representation.

After cleaning, the data is then saved into the database. An important thing to note is that all of the dumps include the ID of each entity in the catalog's database, this is what will then be used as the *target id (TID)*: the id of the target entities used during the record linkage process.

The importer also defines a series of ORM database *models* [2], which are specific to each catalog, one for each concrete entity type. For example, the importer defines an `ImdbWriter` database model which stands for entities of type *writer* which come from the *IMDb* catalog.

Using this modular design, allows to add new external catalogs to `soweego` without much complexity.

Importing data from Wikidata

There is a special importer for downloading data from Wikidata. This Wikidata importer is in charge of downloading both the training and classification sets. These may then be used, together with the respective training or classification sets from an external catalog, to train a machine learning model or to classify the data using a previously trained mode.

These data sets are downloaded separately for each combination of catalog-entity. They will be used as the *source catalogs* during the record linkage process.

Following is an overview of the procedure used in each case.

Downloading training set: To download the Wikidata training set what needs to be done is to receive the *QIDs* representing the catalog and the type of entity. This is achieved by asking Wikidata, via a *SPARQL* query, to provide all the items it has which are of the specified entity type[4] and which also have a *statement* linking them to the given target catalog identifier.

These identifier statements can be easily used to check which of the items provided by Wikidata have also been included in the catalog's dump.

The intersection, those with the same *target id*, between this data and the data that resides in the project's database for the specific catalog-entity defines the labeled data for said entity that can be used for training and evaluation.

Downloading classification set: The classification set is also downloaded on a per *catalog-entity* basis. The procedure for downloading it is very similar to that already mentioned above.

[3] https://github.com/Wikidata/soweego/

[4] for example, all items which are *directors*

The difference is that now Wikidata is asked to only send the data regarding items of a specific entity type (for example, all *Musicians*), and that are not already linked with the specified catalog.

4.1.2 Blocking

When performing the *blocking* procedure, `soweego` compares the entities by using the *name_ tokens* field. This field gets created during the *import* process. It consists of a list of all parts of the name of the entity. For example, the name *Bernie Stewart* is composed of the parts *Bernie* and *Stewart*.

When performing blocking, the first thing that needs to be done is to receive a Wikidata set, and the concrete entity type for which `soweego` is currently running. Then, for every entity in the Wikidata set, the blocking is made by getting all entities in the external catalog which share *name tokens* with the Wikidata entity.

To prevent getting too many results it is allowed for only a maximum of 10 external entities to be retrieved for each Wikidata entity.

For example, suppose a Wikidata entity is currently processed, and in its *name_ tokens* field it has the following list of tokens: `[George, Hamilton]`, and suppose that the external catalog dataset for said entity is as follows:

name_tokens	TID
[George, Thut]	m3289
[Natalie, Hamilton]	m2390
[George, Hamilton]	m9942
[Lucy, Waters]	m5932

Table 4.1: Blocking example external catalog

After blocking with the rule mentioned above, its possible to find that the Wikidata entity may be a possible match with `[m3289, m2390, and m9932]`, but is not a match with `m5932`.

The main reason this blocking rule was chosen, is because all entities have a name field from which the tokens can be derived.

Choosing another field (like *birth date* + *death date*) might yield better results, but some entities do not have said field, which makes the blocking difficult.

4.1.3 Data Cleaning

Data cleaning is a step of the *workflow*, and is executed for each entity in the pairs defined during the previous *blocking* step before giving these *cleaned* entities to *feature extraction*.

The ways in which the entities get *cleaned* is defined in depth in subsection 4.1.3.

4.1.4 Feature Extractor

An in depth description of each feature used will be given in section 5.3. For the moment, a high level overview of how the *feature extraction* step works in `soweego` will be given.

The project leverages the `Python Record Linkage Toolkit` [25] to extract the features. By using this framework the process gets simplified, it only needs the definition of which feature functions to use, and to which fields they should be applied. After this is done, the only thing that remains is feeding to the framework the pairs of potential matches as extracted during the blocking step, and it will automatically yield the extracted features for said pairs.

For example, consider that this step gets as an input the list of potential matches extracted during the previous blocking procedure. An example of such list of pairs can be:

```
[('Q185002', 'nm0310359'), ('Q185002', 'nm2242569'), ('Q185002', 'nm0267785')]
```

Where the first element in each pair is the QID of a Wikidata item, and the second element is the ID of the element in an external catalog (in this case, these are the IDs of entities in the IMDb catalog). Besides these pairs, there is also access to the information about each entity (its fields).

Finally, the feature extraction procedure also defines a list of which comparison functions to apply to which fields of the source (Wikidata) and target (IMDb in this case) entities.

The feature extraction proceeds in such a way that for every pair of entities the list of comparison functions mentioned above gets applied to each them. Each one of these *function applications* will generate a value which will come to be a *feature* that characterizes the relationship between the two entities.

For example, consider using only one comparison function: the same modified *Levenshtein* distance that was exposed in subsection 2.2.3. It is specified that this function should be applied to the fields with the following names: *[(source.name, target.name), (source.description, target.description)]*. Below is a figure which might help explain how this application happens.

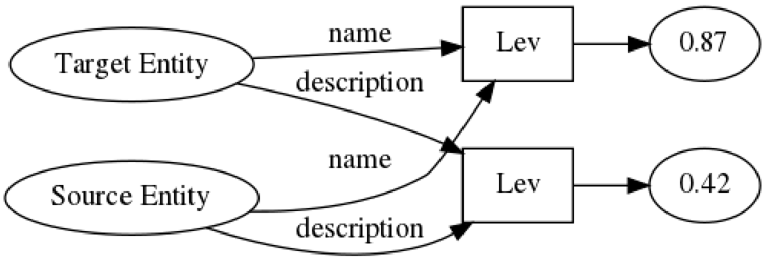


Figure 4.2: Example of applying the *Lev* function to a pair of entities to extract features.

If the procedure mentioned above is applied to all the pairs in the list, in the end there should be a table where each row contains the *feature vector* characterizing every pair:

Pair	Lev_name	Lev_description
`('Q185002', 'nm0310359')`	0.87	0.42
`('Q185002', 'nm2242569')`	0.52	0.24
`('Q185002', 'nm0267785')`	0.74	0.53
⋮	⋮	⋮

Table 4.2: Example results of performing feature extraction using the `Lev` function, on fields `name` and `description`.

For `soweego`, the list of functions applied may change depending on which fields are available in the target and source catalogs. For example, extracting a feature that compares the `description` of both entities only if both have said field. This means that the number of feature vectors extracted for different concrete entities may be different.

Once the feature vectors for all pairs become available, they may be fed to the *train/evaluate/link* modules.

4.1.5 Train, Evaluate, Link

These three can be treated described as one module since their functionality is very similar. The *train* module trains one of the machine learning classifiers presented chapter 6 on a given dataset. The *evaluate* module performs the evaluation of said classifier (the evaluation step is explained more in depth in chapter 7). And the *link* module uses the trained classifier to predict whether the unlabelled pairs given by the blocking step are matches or not.

All of these modules receive as input the matrix of feature vectors from the previous step and feeds it to one of the machine learning algorithms.

Besides this, the *link* module also applies a couple of *post-classification* steps on the predictions. Specifically, what it does is to apply the following *linking rules* to each result in turn.

Wikidata URL if the target in the corresponding pair that is currently processing has a URL, and said URL points to a Wikipedia/Wikidata page then it can automatically extract from said URL which Wikidata entity the target should be associated with.

If this extracted entity is the same as the source entity in the pair then the probability that they match is automatically changed to 1.

Name Rule this rule can be optionally activated when more confident results are desired, and by default is turned off. What it does is that taking the current pair, it checks all possible fields which may be a name, and if there is no intersection among the tokenized contents of these fields then the probability of a match is decreased to 0.

4.1.6 Ingester

The first step performed by the ingester is to separate the given predictions into *non-matches*, *potential-matches*, and *confident-matches*. This is done by simply separating the predictions according to some threshold.

For example, all predictions below 0.4 are *non-matches*, above 0.4 but below 0.7 are *potential-matches*, and everything above that are matches.

The reason for this separation is because `soweego` uploads directly to Wikidata only those predictions which are very likely to be true, whilst the *potential-matches* are uploaded to a service called *Mix'n'match*, where the Wikidata community can check manually each proposed pair and define if it is actually a match or not.

Confident results are *uploaded* by editing the relevant statement of each Wikidata item. For example, if QID *Q185002* is a match with IMDb TID *nm0310359* then the ingester will proceed to create or update a statement on the item with the given QID, using the property *P345*[5] and using as a value the given TID.

The mentioned property is used to link an item with an IMDb ID (similar properties exist in Wikidata for all the other external catalogs used).

4.2 Implementation Details

All the development of `soweego` was made with the Python [67] programming language. To run the internal database MariaDB[6] was used. (which is the same as that used by Wikidata, so there

[5] https://www.wikidata.org/wiki/Property:P345
[6] https://mariadb.com/

is minimum friction when moving the application from development to production). The details of communicating with the database were abstracted away with the use of `SQLAlchemy` ORM [2], which provides an object like interface to the database, and simplifies many of the most common operations.

To simplify the implementation of the record linkage workflow the `Python Record Linkage Toolkit` [25] was used, which implements many useful functionalities for this goal. Mainly it eases the feature extraction and linking procedure. The `Pandas` [51] library is employed to make it easier to handle the data.

Some of the machine learning models provided were leveraged by `Scikit-Learn` [57] to be used in the linker, as well as their *grid search* functionality to find the optimal hyperparameters.

All the visualizations in this dissertation have been done by using the tools *Matplotlib* [44], and *Seaborn* [69].

Finally, the `ML-Ensemble` [35] framework was used to make it easier to implement the different ensemble models used.

Development of `soweego` happens locally and new releases are uploaded to a virtual private server which is provided by Wikimedia for that purpose.

5 Data

This chapter will be dedicated to showing and exploring the data that was used (next section), then mentioning which pre-processing steps the data undergoes and what its final shape is (section 5.2). Following that, the list of features extracted during the *feature extraction* step of `soweego` will be discussed. Finally, a potential difference between the training and classification data is exposed in section 5.5.

5.1 Original Shape of the Data

This section will explore waht is the shape of the data received from the external catalogs, as well as that received from Wikidata. Since they are all different, a section will be dedicated to each catalog.

It must be pointed out for the external catalogs, that even though they are maintained by a group of contributors and, in some cases also by dedicated teams, it does not mean that the data is 100% clean.

During development, there have been multiple encounters with malformed values (like birth or death dates in the future), spelling errors in names, missing values (most commonly, missing birth and death dates), among others.

As such, it must be kept in mind that the information coming from the external catalogs should not be considered the absolute truth.

5.1.1 IMDb

IMDb provides multiple dumps of their data[1], each is a snapshot of the database up until the current date, containing different information. The one used in this project, which is the one that contains

[1] https://www.imdb.com/interfaces/

information about people in IMDb's database, is `name.basics`[2]. The specific dump that is going to be discussed was downloaded from the site on *3 October 2019*.

The data dump is one single tab separated file. It contains 6 different columns. An example of some rows and an explanation of each field are mentioned below.

nconst	primaryName	birthYear	deathYear	primaryProfession	knownForTitles
nm0000001	Fred Astaire	1899	1987	actor,miscellaneous	tt0043044,tt0053137
nm0000002	Lauren Bacall	1924	2014	actress,soundtrack	tt0038355,tt0071877
nm0000003	Brigitte Bardot	1934	\N	actress,producer	tt0054452,tt0057345

Table 5.1: Sample of three rows from the raw IMDb data dump. The list of professions and *knownForTitles* have been shortened to make the table fit. Note that IMDb uses \N to symbolize a null value.

nconst This is the column that holds the ID of the person in IMDb's internal database. This ID is what will later be used as the TID to identify a target entity from this catalog.

primaryName Is the full name of the person. Usually consists of a first name followed by a last name, but may include the initials of middle names as well. As will be shown later, there are even some entities that have more than 3 parts to their name.

birthYear The year in which the person was born.

deathYear The year in which the person died.

primaryProfession This field contains a comma separated list of main professions each person has, this list may contain a maximum of three professions. These *professions* are names that describe the occupation the person is known for, that is, reasons for which the person appeared in the credits of a movie or TV-series.

These professions are not always specific and sometimes act like clusters of similar, and more specific, professions.

knownForTitles List of movie/TV-series IDs the person is known for. This information is not currently used in our project.

The interpretation of most fields is very straightforward. The only exception is the *primaryProfession* field, which will be explained a bit more here. IMDb defines a series of *clusters of professions*, or *occupations*, which group together the accepted professions in the database. The most common professions have their own cluster (like *actor*), while less common ones are clustered together (for example, *music_department* which stands for all occupations related to making music for a movie or TV-series).

The actual possible values for the *primaryProfession* field are:

• actor • actress • animation_department • art_department • art_director • assistant_director • camera_department • casting_department • casting_director • cinematographer • composer • costume_department • costume_designer • director • editor • electrical_department • executive • location_management • make_up_department • manager • music_department

[2] https://datasets.imdbws.com/name.basics.tsv.gz

• producer • production_department • production_designer • production_manager • publicist • script_department • set_decorator • sound_department • soundtrack • special_effects • stunts • talent_agent • transportation_department • visual_effects • writer

All the items in this list correspond to respective professions in Wikidata, meaning its possible to later compare these with the profession associated with a Wikidata item, and from the comparison derive a feature. The feature mentioned will be explained later in section 5.3.

There are a total of 9,613,243 entries in the dump. Of these, all of them have some value in their *primaryName* field. There are 194,469 entries that have a single word in as the name, all the others have 2 or more. In Figure 5.1 its possible to see that the great majority of entities have names composed of two tokens, and the next most common amount of tokens is 3 (possibly those with a middle name), followed by those with only one token, and finally all those entities with names containing 4 or more token.

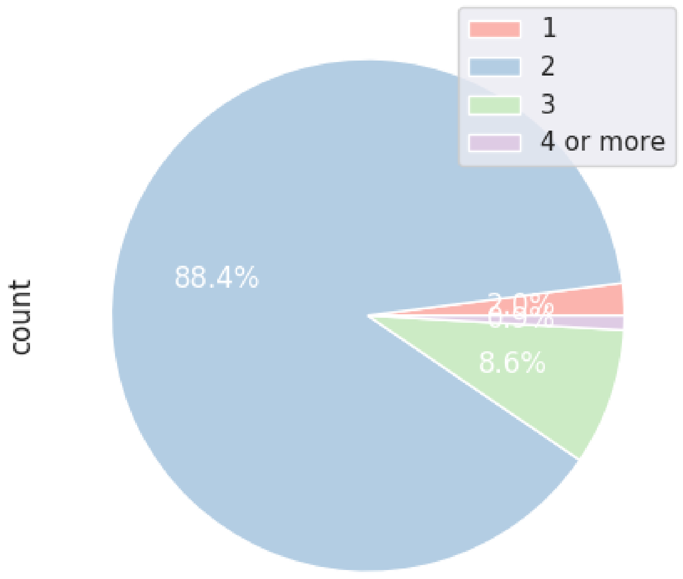


Figure 5.1: Amount of tokens in names of IMDb entities.

Of all the entries, there are only 488,727 which have a *birthDate*, and 170,613 have a *deathDate*. The Figure 5.2 show the distributions of birth and death dates for IMDb. It can be noted that most of the entities were born around the 20th century, and there are fewer entities from years closer to the present (possibly because those individuals are still too young to have any reason to possess a page in IMDb). On the other hand, death dates behave inversely: the greatest amount of deceased entities seems to have died in the latest years.

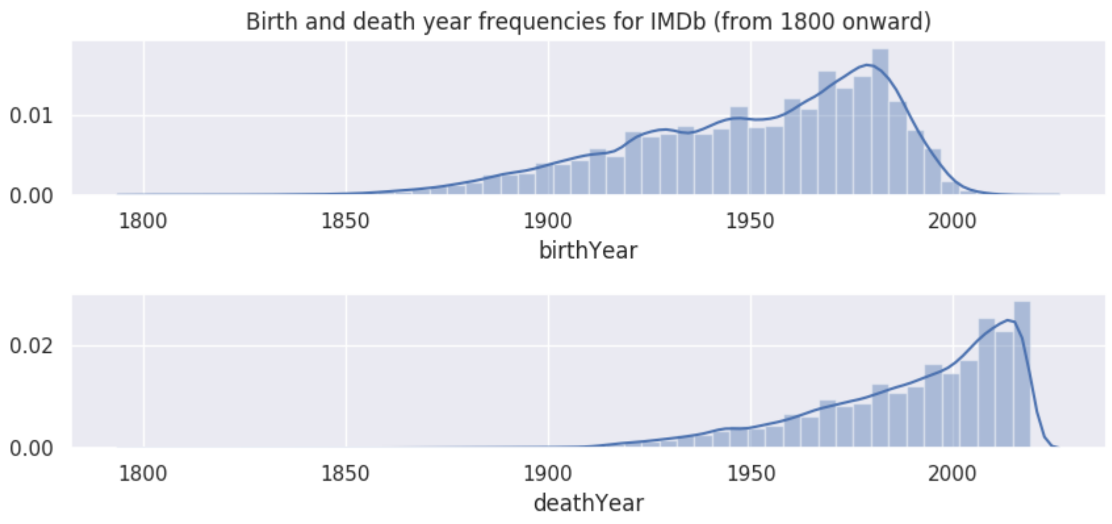


Figure 5.2: Birth (top) and death dates (bottom) for IMDb entities. Note that to generate this figure only dates later than 1800 were considered. 568 entities were born before the 1800s and 305 entities who died before the same date. These entities are not being considered for the graphs above, in an effort to make them easier to interpret, but they are considered during the record linkage process.

The professions field is quite good: only 1,881,284 do not have a profession. The remaining 7,731,959 entities either have 1, 2 or 3 professions listed. Figure 5.3 is a pie chart showing their distribution. It is possible to see that roughly 3/4 of the entities have only one profession, and the last quarter seems to be equally shared by entities with 2 and 3 professions.

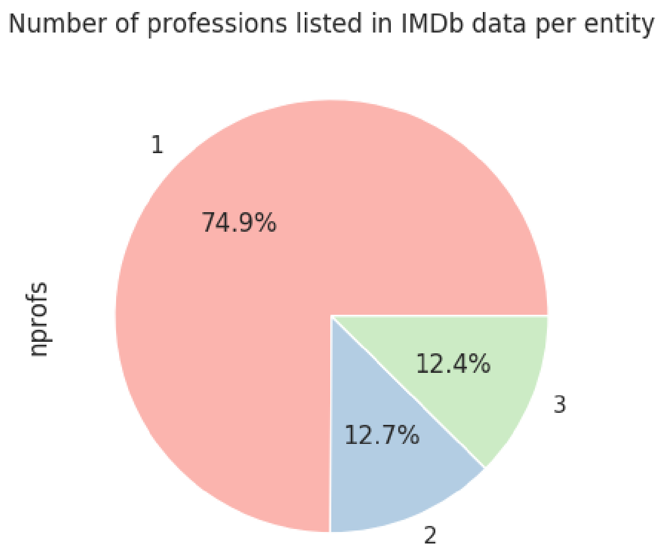


Figure 5.3: Counts of the number of professions listed for IMDb entities.

In Figure 5.4 a bar plot can be seen that shows the amount of times each profession appeared in the dump. And Figure 5.5 shows the percentage of entities which only had one of the professions.

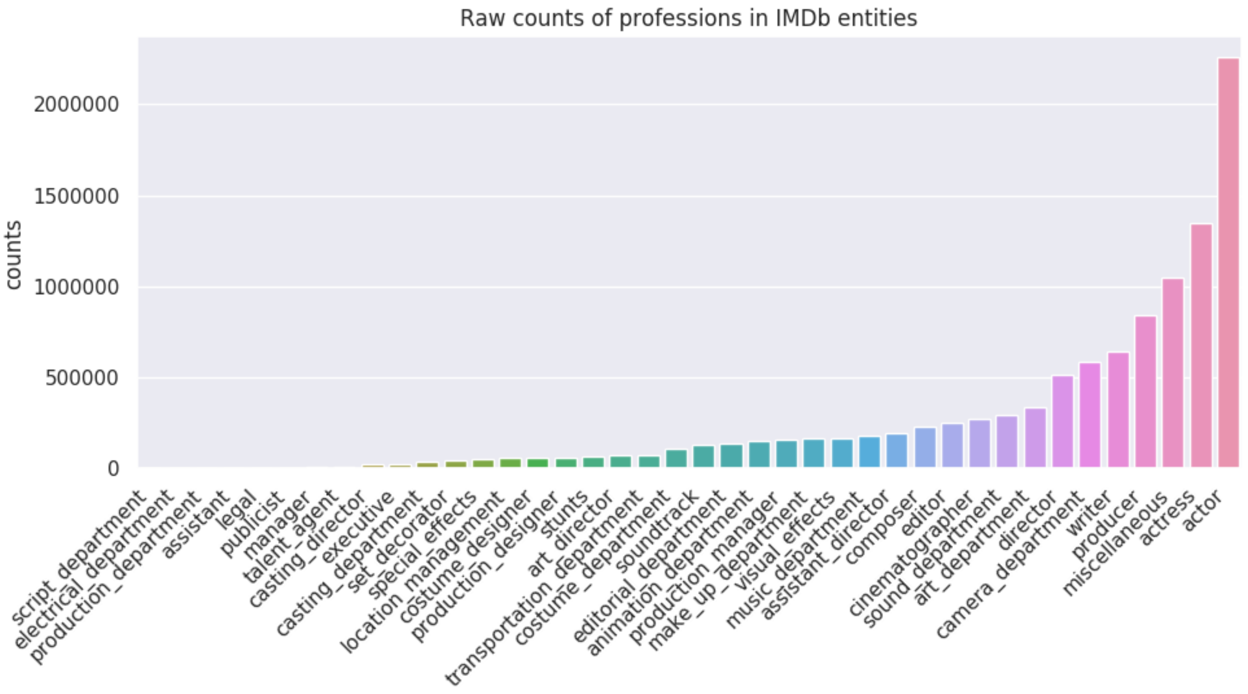


Figure 5.4: Raw counts of the number of professions listed for IMDb entities.

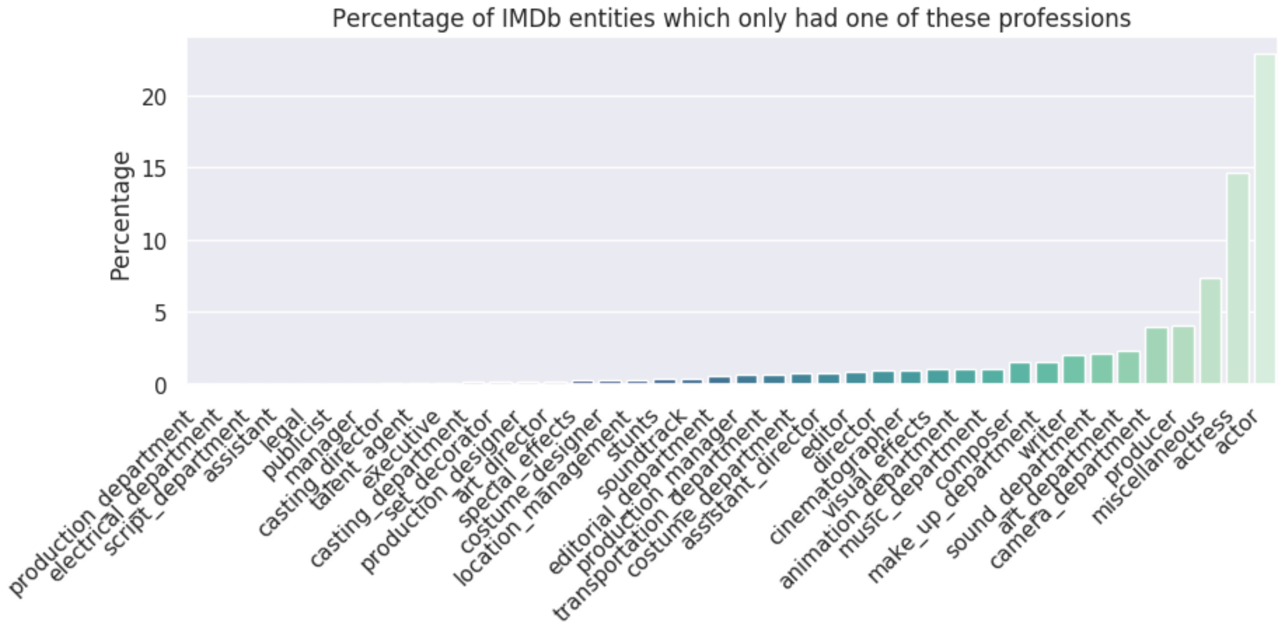


Figure 5.5: Percentage of how many times each profession appeared as the single profession of an entity.

By looking at Figure 5.6 a heatmap can be seen that describes the co-occurrence of the different professions (how frequent is it that an entity has them listed together in its *primaryProfessions* field). It is possible to see there are some interesting correlations among them, for example, that the *director* profession almost always appears together with the *writer* profession. This shows that there is an underlying structure in the professions, which is another motivation to create a feature that leverages this field.

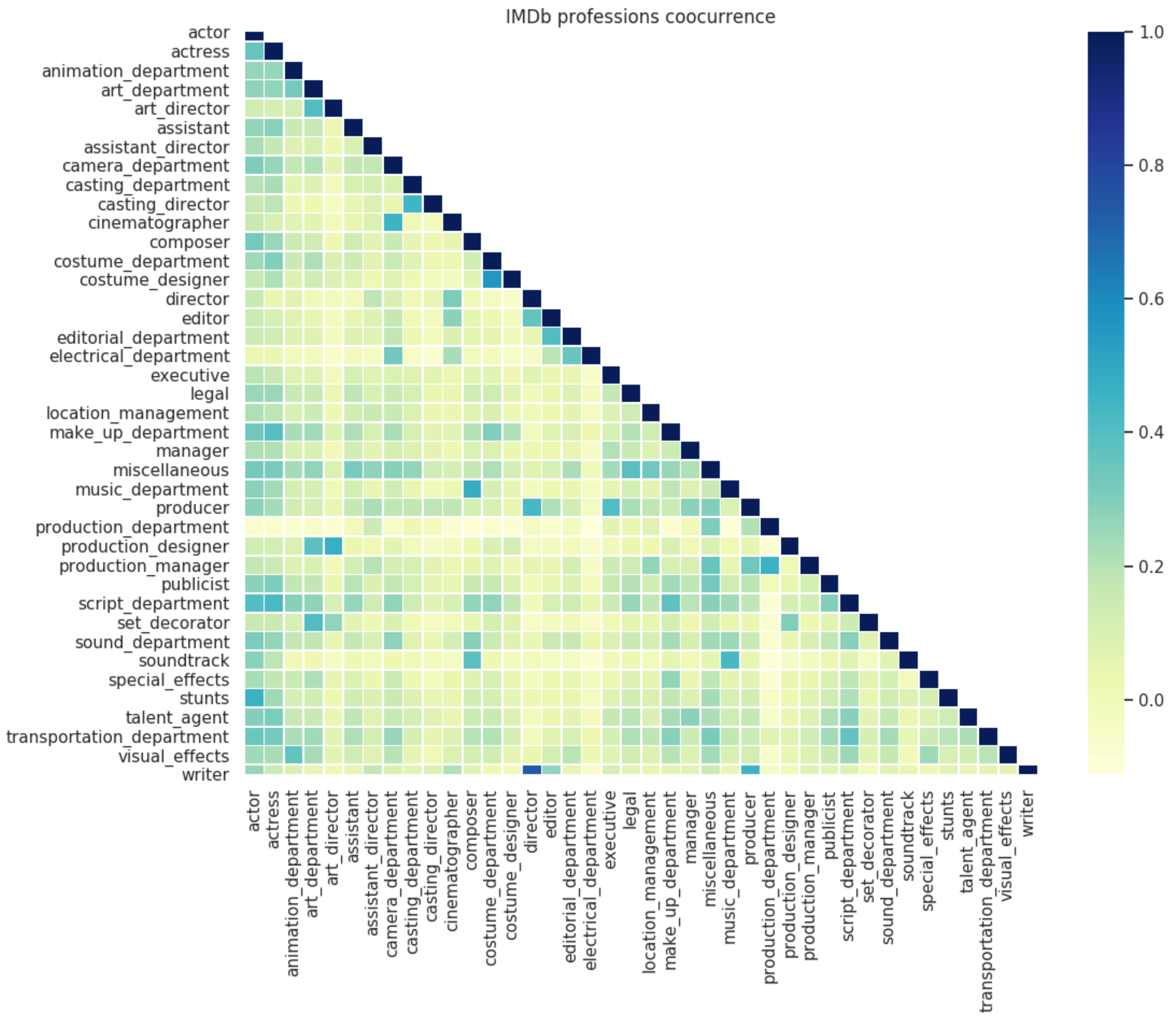


Figure 5.6: Co-occurrence of IMDb professions.

It is evident that the IMDb dump does not provide much information that can be used. The name fields seem to be the ones with the highest quality (no empty values). Birth and death dates are quite informative but sadly only a few entities have values there. The remaining field, *primaryProfessions*, is also quite good in terms of data quality, and its underlying structure will allow the creation of an effective feature from it.

5.1.2 Discogs

Discogs provides multiple data dumps[3], and maintains records of dumps released in different dates. The one used for this project is *discogs_20190901_artists*, whose last revision was on *2019-09-08T13:03:25.000Z*.

This dump contains information about artists, which can be either single musicians or music groups. The dump is an XML file and for each entity (either band or musician) provides the following information:

identifier This is the ID of the entity in the Discogs database. It will be the field used as the TID for this catalog.

name The main artistic name associated with the entity.

[3]https://data.discogs.com/

realname The real name of the entity, which might differ from its artist name.

namevariations Possible different artist names for this entity.

data_quality It is a Discogs specific metric that shows how good the data is. Its possible values are • Correct • Complete and Correct • Needs Vote • Needs Minor Changes • Needs Major Changes • Entirely Incorrect

profile This is the profile description of an entity.

living_links This field holds a list of hyperlinks associated with the entity. They are usually links that go to external sites (for example, Twitter, Wikipedia, YouTube, etc).

group This field is specified for entities which are a musician. It specifies which groups the musician belongs to. This project only considers *musicians* as those who have some value in this field. This information is only used to determine which entries are *musicians* and is not otherwise leveraged in the project.

members This field is specified for bands, and shows which musicians are associated with which bands. This project only considers as *bands* those entities which have some value in this field. This information is only used to determine which entries are *bands* and is not otherwise leveraged in the project.

There are a total of 6,432,264 entities in the Discogs dump. Of these, 1,129,711 have a value in either the *band* or *members* field. This is the final number of entities `soweego` will import into its database, so these are the ones for which all the next metrics will be derived.

There are 384,352 entities that have some value in the *members* field, meaning that these are the bands in the data dump. And there are 749,499 which have a value in the *groups* field (these are the musicians). There are also 4,140 entities which have *both* members and groups. Figure 5.7 shows the most common number of bands a musician belongs to and the most common number of members a band has. It can be seen that it is more common for bands to have multiple members, whilst it is not very common for a musician to be part of more than one band.

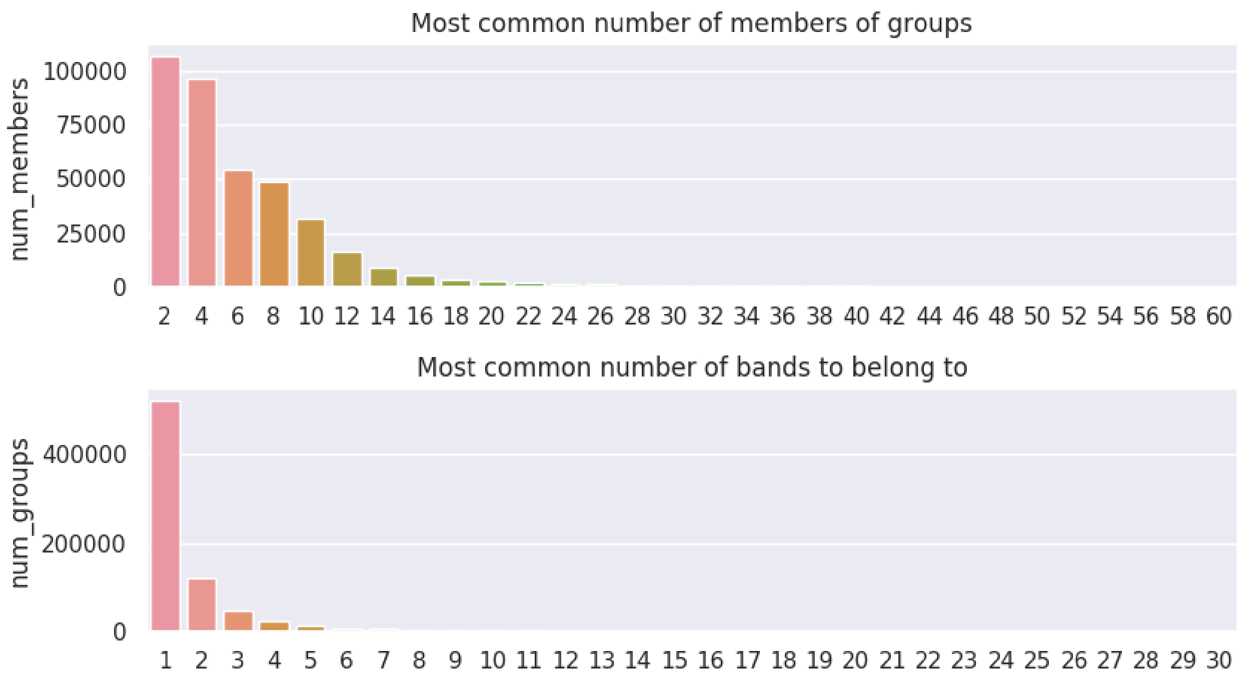


Figure 5.7: The plots above show what is the most common number of members in a band (top) and the most common number of bands a musician belongs to.

Of all the entities, 935,456 are not associated with a real name. However, all of them do have a *name*. Most of these are composed of 2 tokens. Figure 5.8 shows the distribution of the number of tokens for the names of the entities. Notice that 2 is the most common, but 3 and 4 are also quite common.

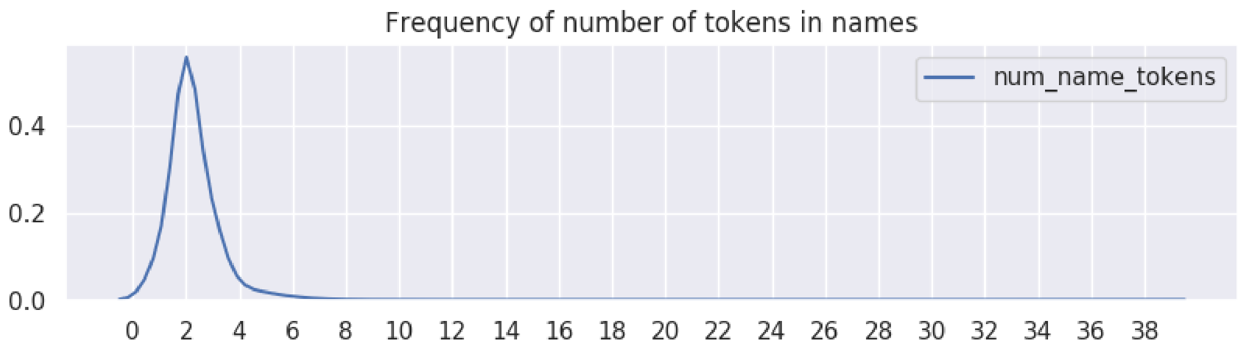


Figure 5.8: Distribution of the number of tokens in names of Discogs entities.

There are 492,315 entities which have at least one value in the *namevariations* field. Figure 5.9 shows which are the most common number of alternative names, among entities who actually have alternative names. Notice that it is much more likely to have only one name variation, with the likelihood of having another decreasing almost by half with each extra alternative name.

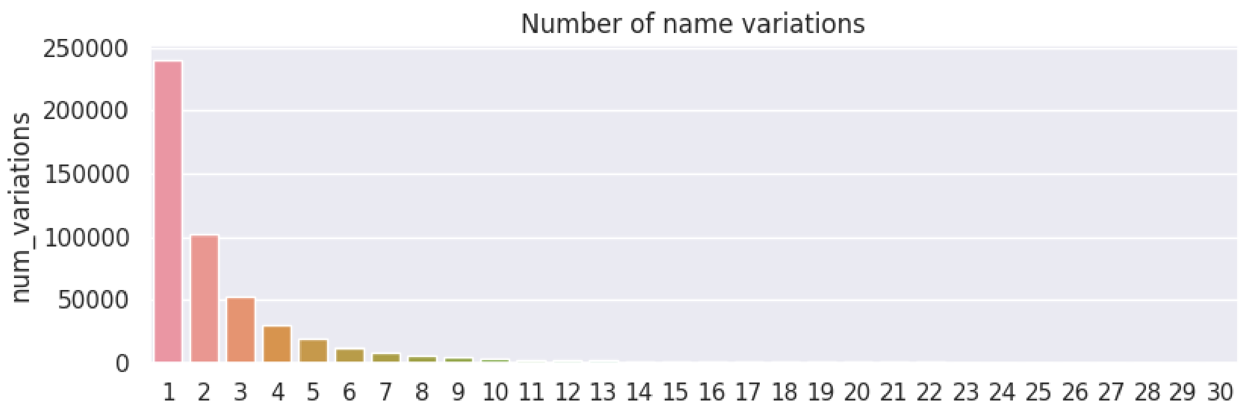


Figure 5.9: Distribution of the 30 most common number of tokens in names of Discogs entities.

Regarding the *profile* field, which comes to be like the description of the entity, 487,588 have some kind of value in it. Figure 5.10 shows the distribution of how many words can be expected to be found in such description (among entities who do have one). Notice that if an entity has a profile description then the most likely scenario is that it will have one composed of only two words. It is interesting to notice that having only one word in the description is almost as likely as having 7 of them.

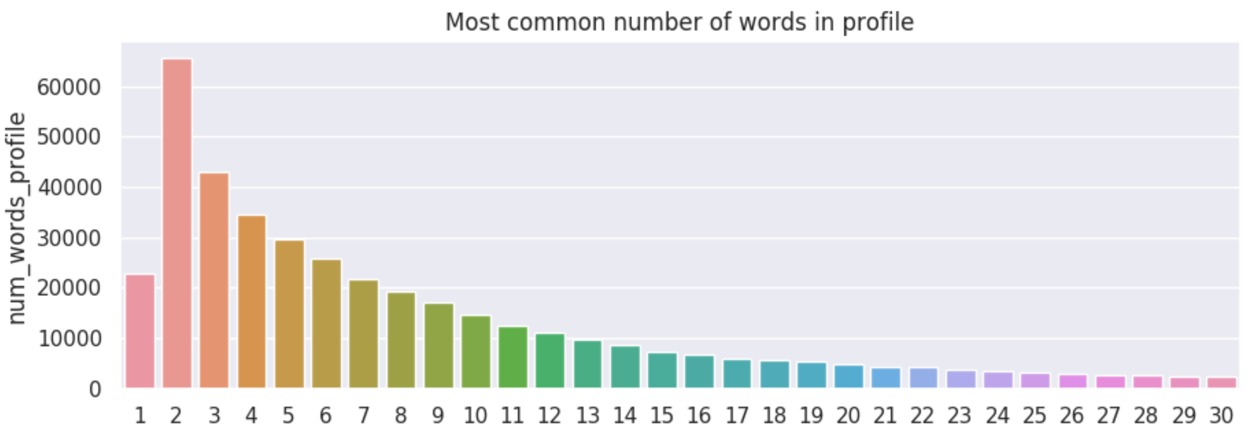


Figure 5.10: Distribution of the 30 most common amount of number of words in the *profile* of Discogs entities.

Discogs also provides how many entities are associated with links to external pages. In the dump, there are 243,757 entities with external links. Figure 5.11 shows which amount of links is the most common among entities that do have links. It seems to be very common, if an entity has at least one link, that it actually has only one. Entities with 3 and 4 links can be found, but 5 or more are much less common.

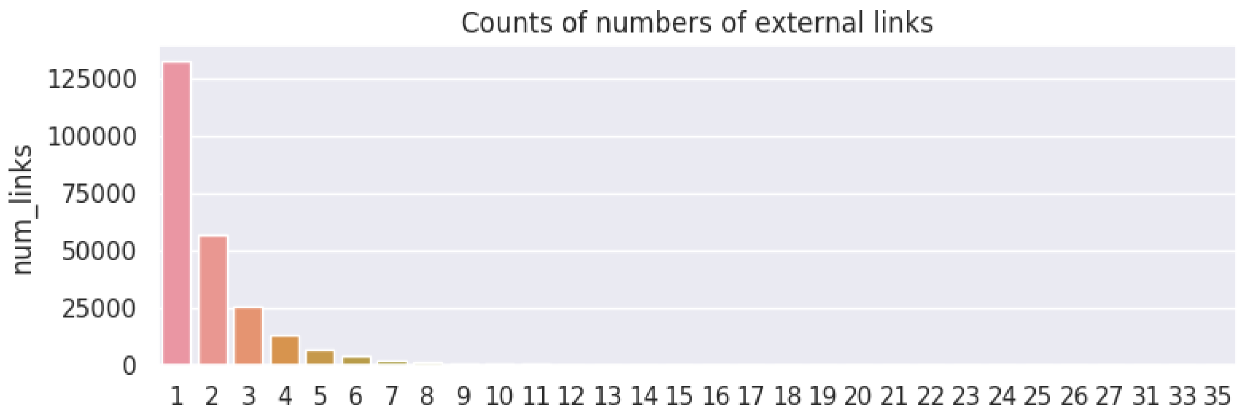


Figure 5.11: Distribution of the 30 most common amount of external links associated with Discogs entities.

The *data_ quality* field shows how good the data about a certain entity is. All entities have a value in this field. Figure 5.12 provides the distribution of the various data quality labels. It is possible to see that more than half of them are marked as *Needs Vote*, little less than a quarter are *Correct*, and the rest are labeled as incorrect (in different degrees).

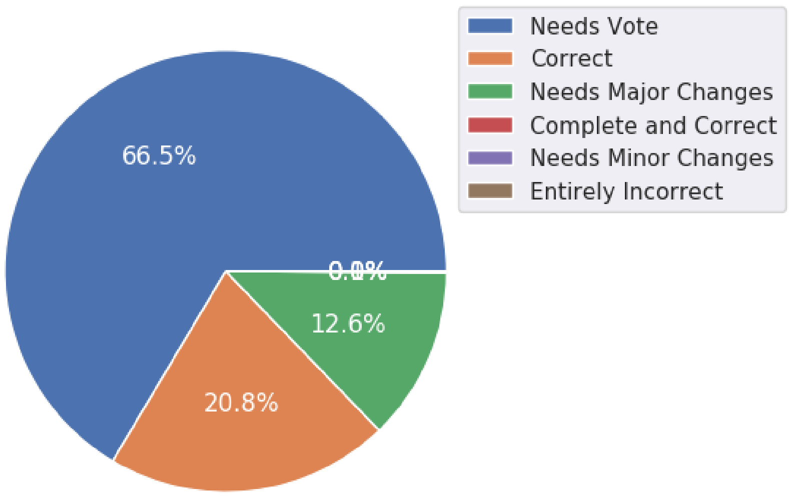


Figure 5.12: Percentages of entities marked with a given *data quality* label.

5.1.3 Musicbrainz

The Musicbrainz dump[4] is a snapshot of the whole dataset. As such, it contains a lot of data that is not of any interest to this project. The dump used for this catalog is the following:

- 20191002-001702/mbdump.tar.bz2

In the dump, there are a total of 1,003,366 musicians and 437,287 bands. The dump provides the following information about each of these entities:

[4]https://musicbrainz.org/doc/MusicBrainz_Database/Download

gid The id of the entity in Musicbrainz's database. This is the ID that will be used as the TID during the record linkage procedure.

name Name of the entity.

birth day, birth month, birth year Define individually which is the day, month, and year a person was born in. If one of these is unknown then the value is *NULL*. By having this separate, the *precision* with which a date is known can be derived. This will be useful later when deriving features from said dates.

death day, death month, death year Same as the fields mentioned above, but for death dates.

birth place, death place Specify information on where the person was born, and/or died. For bands, the birth and death date symbolize where the band was formed and where it separated.

gender Specifies the gender of the entity (*male* / *female*).

type_id This field specifies which type is the entity in question. Possible values are: *person*, *character*, *orchestra*, *choir*, *group*. Entities with any of the first two values are considered *musicians*, whilst those with any of the last three are considered *bands*.

links The external links an entity is associated with.

relations with bands It specifies the relation among artists and bands, specifically, the list of bands to which they belong.

All entities have a name associated with them. Figure 5.13 shows the distributions for said number of tokens. It is interesting to note that bands are almost as likely to have a name composed by a single token than they are to have one with two. Another interesting thing is that the names of bands are more likely than those of people to contain more than two tokens.

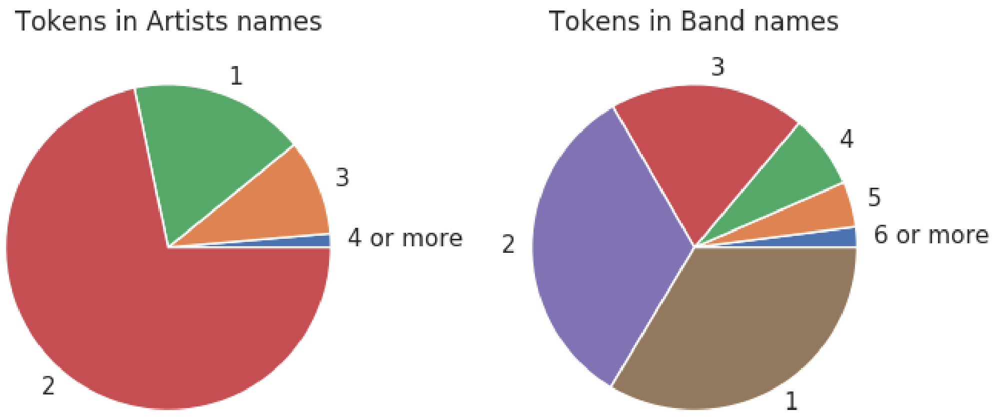


Figure 5.13: Percentage of entities which have a name composed of a specific number of tokens.

As per gender distribution, there are much more *males* than *females*. There are 767,616 musicians for which gender is specified. This distribution is shown in Figure 5.14. Notice that less than a quarter of entities are *female*.

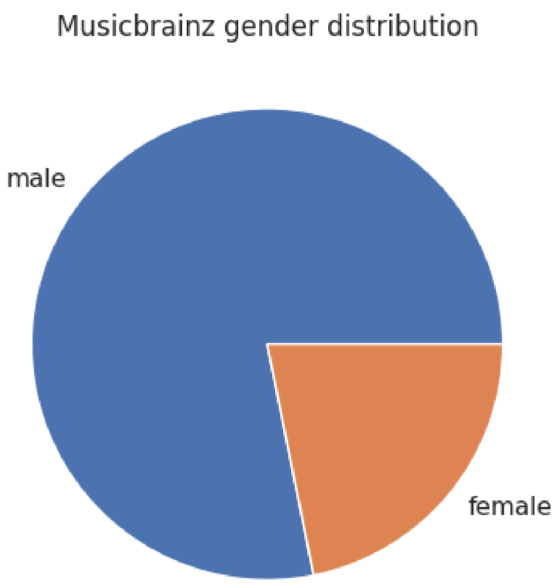


Figure 5.14: Percentage of entities separated by gender.

There are 297,011 musicians and 164,060 bands which are associated with a birth date. There are also 88,191 musicians and 34,213 bands who have a death date defined. The distribution of the frequency of said dates can be seen in Figure 5.15. Notice how the curve symbolizing the birth date of musicians is much more smooth than that of groups, and there seems to have been a peak in the number of groups created in the latest years. This suggests the adaptation of this digital platform is more common for the newer generation of bands. Also, death years seem to follow this same behaviour.

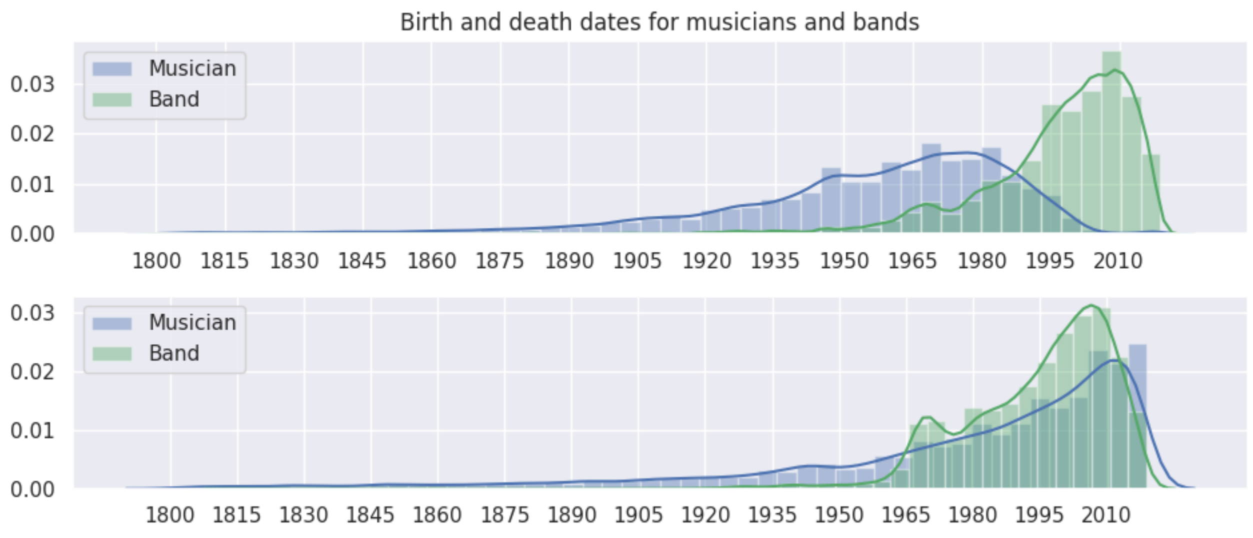


Figure 5.15: Distribution plot of the years musicians were born or died; and years in which bands were created or dissolved. Top shows the distribution of birth/creation years, while bottom shows the *death/dissolve* years. Dates for musicians are in *blue*, and those for and bands are *green*. Note that both graphs were created by considering dates later than *1800* and earlier than *2020* (which at the time of this writing is in the future). However, dates outside this range, were used during the performance of record linkage.

There are 223,340 musicians and 94,826 bands associated with a birthplace. Similarly, there are 53,631 musicians and 4,885 bands associated with a death place. These behave similarly as the birth and death dates, where there are more entries with a birth date than death date.

There are 517,524 musicians and 266,206 bands which are associated with at least one external link. Figure 5.16 shows which percentage of musicians or bands have a certain number of links. It can be seen that both more commonly than not have only one link. However, it is more common for bands to have more links than musicians.

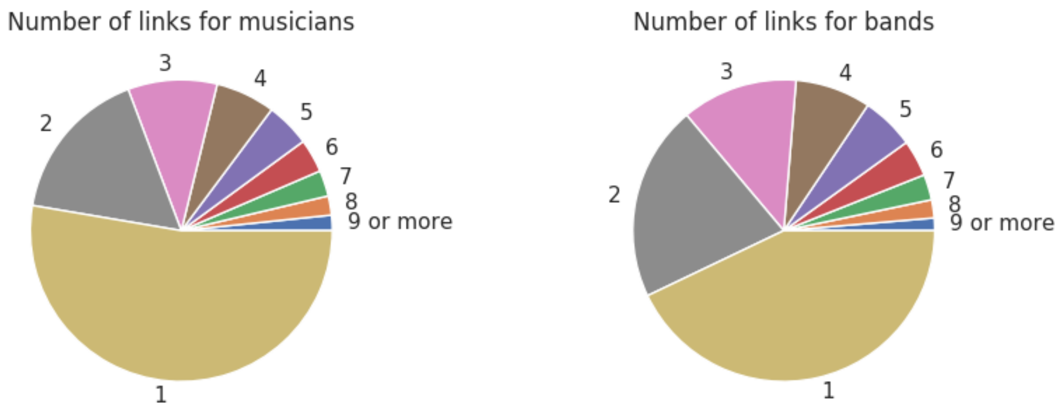


Figure 5.16: Distributions of the number of external links for musicians and for bands.

5.1.4 Wikidata

For Wikidata, training and classification *sets* were downloaded, one pair for each concrete entity. As explained in subsubsection 4.1.1, the training data already contains the labels, or true links, for each entity, and is what is used to train the different machine learning algorithms. This means that each Wikidata *training* dump has a field named *TID*, which is the ID of the target catalog. On the other hand, the classification data are the entities that don't have target identifiers and need to be linked. This data is downloaded through Wikidata's public SPARQL endpoint,[5] which allows us to easily get the relevant statements about a given entity.

The following are the fields that are gathered from Wikidata.

TID ID of the entity in a specific external catalog. Only found in the dumps used for training. The actual values may change depending on which catalog this dump was downloaded for. For example: if the dump was downloaded to be used with IMDb, then the TIDs are referenced to IMDb entities. If it was downloaded for Discogs, then the TID will reference Discogs entities.

QID This is the unique identifier of the Wikidata entity. This is what will be used as the *source ID* while performing record linkage. Note that in the present work the terms *QID* and *source ID* are used interchangeably.

name The name of the entity in Wikidata.

url List of URLs associated with the entity. These are URLs about the entity and may point to Wikidata pages in other languages, or external catalogs.

[5]https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service

description The description of the entity in Wikidata.

born Specifies the date the entity was born.

died Specifies the date the entity died.

sex_or_gender Specifies the gender of the entity.

place_of_birth Specifies the entity was born.

place_of_death Specifies the place where the entity died.

pseudonym Is a list of alternative names by which the entity might be known.

occupation List of QID's representing the professions an entity is known to have performed.

birth_name The name of the entity, given at birth.

given_name The first name or forename of the entity.

family_name The last name or surname of the entity.

Table 5.2 shows which is the size of the datasets, both the training and classification ones, for each concrete entity. It seems that the datasets for musicians are among the largest in both cases. In fact, they are except for the *IMDb Actor* training set, which is the largest by approximately 50k. On the other hand, the smallest classification set is *IMDb Producer*, by almost 20k. In the classification sets, the ones for IMDb *producer* and *director* are much smaller in comparison to all the other.

Catalog	**Entity**	**Number of Entities Train**	**Number of Entities Classification**
Discogs	Band	43,428	36,200
Discogs	Musician	88,069	183,483
IMDb	Actor	177,417	89,081
IMDb	Director	46,904	8,407
IMDb	Musician	61,034	210,017
IMDb	Producer	19,861	2,075
IMDb	Writer	41,373	15,143
Musicbrainz	Band	47,096	32,407
Musicbrainz	Musician	118,107	153,437

Table 5.2: Sizes of Wikidata dumps for training and classification.

5.2 Preprocessing

This section will show which steps the raw data undergoes before it can be used in the *feature extraction* steps. There are two different preprocessing steps, one for data coming from Wikidata, and another for data coming from external catalogs. There are also some steps that are applied to data coming from both sources. That is why this section will be separated into three subsections, one explaining each set of steps. During the importing process, the raw dump of the external catalogs also gets briefly preprocessed, these modifications will be mentioned in the last subsection.

It is important to note that the preprocessing for *Wikidata* or *External* data happen before the *Shared* preprocessing.

All of these preprocessing steps, except for those in made in the *importer*, are done when the data is loaded from the database, immediately before the blocking step.

In the cases of *Shared*, *Wikidata*, and *External* preprocessing, it is done for chunks of data at a time, currently 1000. The *Importer* preprocessing, on the other hand, is done for each entity separately.

5.2.1 Shared Preprocessing

Normalize dates

The dates are by default saved together with their precision. The precision is a number which signals which is the most specific part of the date that it can confidently be said is correct. Especially for data coming from Wikidata, this precision can be anything from knowing the exact second something happened to knowing in which range of one billion years it happened[6].

For the current application, it is important that for all dates it is known at least the year. For instance, the worst precision, which is is any number ≤ 9, will be converted to a precision of 9, which indicates that at most the year can be trusted. A precision of 10 indicates that both the month and year are known. A precision of 11 or more indicates that the year, month and day are known. And so on until a precision of 14, which indicates that everything up to the second is known.

This date normalization what it does is convert the dates to *Pandas* [51] *Period* objects, which can contain the precision of the date internally.

This procedure is performed for both birth and death dates.

Normalize names

This step normalizes all possible *name fields*, which are:

- name
- alias
- birth name
- family name
- given name
- pseudonym
- real name

Any of these fields can contain a list of possible names for an entity. The way in which they are normalized is by performing the following steps on each possible name:

1. Remove any white space padding which the name may have on the left or right.
2. Normalize all characters to the *ascii* alphabet. This means removing the *accent* from characters, and converting characters in other alphabets (e.g. Cyrillic) to *ascii* characters.
3. Make the name lowercase.

[6] https://www.wikidata.org/wiki/Special:ListDatatypes

Normalize occupations

This step simply takes the list of all occupations an entity has and converts them to a set (i.e. removes the duplicates). It also removes *empty occupations*. If the occupations are not already a list then they get converted to one before the list is converted to a set.

5.2.2 Wikidata Preprocessing

Drop columns with empty values

As the name of this step implies, what is done is that all columns in the entity which are composed of only *null* values are removed.

Ensure one target ID per QID

This step is exclusively applied to Wikidata entities that come from a training dump. It ensures that all QIDs in the positive samples of the training data are only associated with one TID. This is done by checking all QIDs which are associated with more than one TID, all associations besides the first one are dropped.

Tokenize names

This step is applied to all *name* fields, which were defined earlier in the shared *Normalize names* step. This step performs the same normalization on the names as the previously mentioned step. But in addition to that, it also tokenizes the resulting normalized names.

This is done by splitting the resulting normalized names on the white space that separates each part of the name, and then removing from this list of *name parts* all those which are considered *stopwords* according to a previously defined list. Finally, this filtered list is converted to a set so that it only contains unique *tokens*.

If an entity has more than one name in a given name field then the tokens coming from all names are added to the set mentioned above.

Tokenize URLs

Each URL is first separated into the following five components:

<scheme>://<netloc>/<path>?<query>#<fragment>

Then the *netloc* is further splitted into tokens, where each token is each word in the string, these are called *domain tokens*. From these *domain tokens*, common top level domains (*com*, *org*, *net*, *info*, *fm*), as well as domain prefixes (*www*, *m*, *mobile*) are removed.

Following this, the *path* is splitted into its respective chunks, by splitting on the slash (/) character. The words in these chunks are extracted, and these are then added via *union* to the set of tokens coming from the domain.

Finally, the *query* is split into its constituent words and these are also added via *union* as part of the tokens from the *domain* and *path*.

5.2.3 External Data Preprocessing

Drop columns with empty values

This step is the same as what is done for Wikidata. Columns which are entirely composed of *null* values are dropped.

Pair dates with precision

The step is executed once for birth dates and another for death dated. It simply takes the value of the date, and the precision for said date, and joins them in a tuple. The reason for this is that then this tuple can be given to the shared *Normalize dates* step, which will extract the appropriate Pandas Period objects.

After joining the precision and dates, the precision column is eventually dropped from the data set.

Aggregate data on target ID

If the data currently being processed contains more than one row for some TID then the values of these rows are joined together by performing a *set union* on the values of each field. For example, say that the table below is the input data:

TID	Name	Zip Code
6567	[Gary, George]	[93124]
6567	[George, Paul]	None
3251	[John]	None

There are two entries with the TID *6567*. If they are then aggregated using *set union* as mentioned above, the output will look like the following table:

TID	Name	Zip Code
6567	[Gary, George, Paul]	[93124]
3251	[John]	None

Note that now the information of both rows with TID *6567* have been joined into the same row.

5.2.4 Preprocessing During Import

The following steps are applied on the raw catalog dump data while it is being imported into `soweego`'s internal database. Said preprocessing steps are:

1. `Tokenize names` – this is done in the same way as is done for Wikidata, explained in subsubsection 5.2.2.

2. `Tokenize URLs` – also the same as done for Wikidata, explained in subsubsection 5.2.2.

3. `Tokenize description` – this step essentially the same as the *Tokenize names* one. The difference is that here the tokenization is done on the *description* field instead of the *name*.

5.3 Features

This section will be centered around explaining said list of functions, and on what fields they're applied. Each feature will be explained in depth in it's own subsection.

Note that not all features get extracted every time: they're only applied if possible. For example, if we have a feature function which needs to compare URLs, it will only be used if both entities have URLs to compare. `soweego` automatically checks that entities have the fields a feature function depends on before applying it.

5.3.1 Exact match on names

This feature is extracted by taking the *name* fields of both entities and checking if they're the same. This field may have more than one value (e.g., in the case of name variations), so the feature compares all possible pairs of names, and optimistically yields *1* if any of the comparisons is a match. Remember that the values of features may go from *0* (not a match), to *1* (perfect match).

For clarification, below is the algorithm used to find the perfect match. Note that this same procedure can be applied when performing *perfect match* on any two lists of values.

```python
def exact_match_name(source_entity, target_entity):
    for source_name in source_entity.names:
        for target_name in target_entity.names:
            if source_name == target_name:
                return 1
    return 0
```

5.3.2 Exact match on URLs

Similarly to the feature defined above, both source and target entities may have a list of URLs associated with them. The returned feature will be *1* if any pair of URLs is exactly the same, and 0 otherwise.

5.3.3 Shared tokens in URLs

This feature expects that both entities that are being compared have a list of URL tokens (which were extracted during the preprocessing step: subsubsection 5.2.2). Besides this, it also receives a set of stop words β, which are words that shouldn't be considered during the comparison process.

This function takes all the URL tokens for each entity i and converts them to a set, so there are two sets α which contain all URL tokens for a specific entity. All stop words are then removed from these sets $\sigma = \alpha \setminus \beta$. Finally the final score is calculated as follows:

$$\frac{min(\|\sigma_{source}\|, \|\sigma_{target}\|)}{\|\sigma_{source} \cap \sigma_{target}\|}$$

Basically, this function measures the percentage of elements of the smallest set which are shared among both sets of tokens. If the smallest set is composed of only one element and this element is also in the largest sent then the value of this feature will be *1*. If on the other hand the value is not contained in the largest set then the value will be *0*.

Using a set of stop words greatly diminishes the amount of times the function classifies as similar

URLs which are not really similar. This set of stop words was constructed by taking all URL tokens in the database, counting them, and choosing the most common 850. Terms in this set are mainly domains to common websites like *Facebook*, *YouTube*, etc, as well as common words and names.

5.3.4 Similar Birth Dates

This function compares a pair of dates by comparing only the parts that are certainly correct. For example, remember that in subsection 5.1.3 it was said that for *Musicbrainz* we separately get the value of the birth year, month, and day, and that some or all of these may not be specified. IMDb specifies only the year, and Discogs does not provide us with birth nor death dates at all.

This means that dates coming from *Musicbrainz* have a variable precision, depending on what was specified in for the entity in the dump file, but the highest possible precision to know all three of these. For IMDb, on the other hand, only the year of the dates can be assumed to be correct.

The *date precision* specifies which parts of the dates can be trusted. It goes from less specific to more specific. For example the most general a precision can be is to have only the year as the trusted part of the date. The next, more specific step, is to trust the month, followed by the day, etc.

This feature function supposes that the entities have a birth date δ_i, and also have a specified precision ϕ_i for said date. A date has at most 3 parts (year, month, and day), and as such also has a maximum precision of 3, since the most precise part of the date is the day. A given pair of dates can only be compared up to the lowest common precision, this is just $min(\phi_{target}, \phi_{source})$.

The comparison can start from the lowest precision and keeping a counter ω to see how many parts of the dates are the same. If after comparing the lowest precision part (the year) yields that they're the same then the counter is increased and the next, most specific part, can be compared. The process is stopped when the maximum precision is reached (*day*) or when the parts currently being compared are not equal.

More formally, the following pseudocode defines the algorithm:

```
shared_precision = min($\phi_{target}, \phi_{source}$) # the maximum precision we can compare
$\omega$ = 0 # counter to track how many parts the dates have in common

for date_part in 0..shared_precision:

    # get the part corresponding to the current precision, starting from the
    # lowest, which is the year
    source_part ← $\delta_{source}$[date_part]
    target_part ← $\delta_{target}$[date_part]

    if source_part == target_part:
        $\omega$ += 1
    else:
        break
```

Finally, the feature is then calculated as follows.

$$\frac{\omega}{\text{max\_precision} + 1}$$

Basically, this is just the percentage of sequential parts that are the same among all parts that can be compared. The lowest precision, *year*, corresponds to a precision of *0*, that is why we add the +1 in the denominator.

For IMDb dates this function just compares the years, and if they're the same then returns *1*, or *0* if they are different. For *Musicbrainz*, which has more *flexible* dates, this function is much more useful, since it allows the comparison with arbitrary precision values.

Note that if entities have more than one birth date then all the possible pairs are compared and the best result is returned.

5.3.5 Similar Death Dates

This feature is exactly the same as the one shown above for birth dates, but is applied to death dates instead.

5.3.6 Levenshtein distance on name tokens

Levenshtein distance [49] is a string comparison metric which measures how different two given strings are by considering how many insertions, deletions, and updates need to be made to make both strings equal. Below is the representation[7] of the Levenshtein distance as a dynamic programming algorithm:

$$lev_{a,b}(i,j) = \begin{cases} max(i,j), & \text{if } min(i,j) = 0 \\ min \begin{cases} lev_{a,b}(i-1,j)+1 \\ lev_{a,b}(i,j-1)+1 \\ lev_{a,b}(i-1,j-1)+1*(a_i \neq b_j) \end{cases} & \text{otherwise} \end{cases}$$

Where a and b are the strings that are currently being compared. i and j are the indices of the characters currently being compared. Initially the process starts from the end of the strings, so $i = |a|$ and $j = |b|$. In the original formulation, the strings are *1-indexed* (*0* means the string has been completely processed). $lev_{a,b}(i,j)$ stands for the distance between the first i characters of a, and the j characters of b.

The calculation of the distance starts by checking if either of i or j are 0. If they are, then the number of changes needed is the same as the number of characters still remaining in the longest string (which are the number of characters that need to be inserted into the shortest one).

If on the other hand none of i, j are 0, then the function is recursively applied to all possible variations (in the min branch), and the minimum among them is returned. The first variation corresponds to deleting a character from a. The second corresponds to inserting a value in a. Finally, the thirst stands for a replacement.

Note the +1 at the end of all variations. These are the penalty paid each time one of these branches needs to be taken. In the third variation, the penalty is only applied of the characters at positions i and j are different (i.e. there is need to perform an update). If the strings are the same then this third branch will be taken for every character comparison, and the penalty will always be *0*.

[7]Adapted from https://en.wikipedia.org/wiki/Levenshtein_distance

The resulting score increases the more *dissimilar* two strings are. However, since feature functions need to return 0 when dissimilar and 1 when similar, the result of the Levenshtein function is adapted in the following way:

$$\frac{Lev(a, b)}{max(|a|, |b|)}$$

If the two strings being compared are completely different then every character of the shortest string must be changed, and extra characters need to be inserted to make it equal to the longest string. These changes sum up to maximum string length. In this case, the division would yield *1*, effectively signaling that the strings are completely different.

If the entities have more than one set of name tokens (i.e. they have multiple names associated with them, which have all been tokenized) then the function shown above is applied to all possible combinations and the best result is returned. Note that before comparison, each list of tokens (which corresponds to only one name) is joined into a single string.

5.3.7 Cosine similarity on name tokens

Cosine similarity has long been used in the field of information retrieval [65] since it allows to measure the distance between two *documents* by checking which words they have in common. This behaviour is similar to what the current function should to, the difference is that instead of names we want to count the occurrence of *bi-grams*.

More in general, n-grams, are defined as all the possible, n sequential elements in a piece of text. The *elements* in question can be letters, words, or etc. For example, taking the name *Sarah*, and considering the possible n-grams of characters, the following table can be generated. Note that the *white space* before and after the name are being considered as characters, this will be shown as an underscore (_) in the examples below

Name	_sarah_
1-gram (uni-gram)	_-s-a-r-a-h-_
2-gram (bi-gram)	_s-sa-ar-ra-ah-h_
3-gram (tri-gram)	_sa-sar-ara-rah-ah_
⋮	⋮

This function compares the names of two entities by first extracting bi-grams from the names and then constructing a vector specifying which bi-grams are contained in each of the names. These vectors are finally compared using *cosine similarity*.

For example, suppose the function is being applied to *Sarah* and *Sara*. The corresponding bigrams would be: *s-sa-ar-ra-ah-h* for the first, and *s-sa-ar-ra-a* for the second. These two sets of bi-grams can them be joined into a single set $\beta = \{s, sa, ar, ra, ah, h, a\}$. This last set represents all the possible bi-grams that can be found in any of the two names.

By traversing this set, a vector ρ can be constructor for each entity e, where the the value at index i is defined as.

$$\rho_i = \begin{cases} 1, & \text{if } \beta_i \in e.nameBigrams \\ 0, & \text{otherwise} \end{cases}$$

For example, the resulting vectors for *sarah* and *sara*, considering the β defined above would be $\rho_1 = [1,1,1,1,1,1,0]$ and $\rho_2 = [1,1,1,1,0,0,1]$ respectively.

These vectors can then be passed to the cosine similarity function, which is defined as the dot product of the vectors normalized by their lengths:

$$\frac{\rho_1 \cdot \rho_2}{\|\rho_1\|\|\rho_2\|} = \frac{\sum_{i=1}^{n} \rho_{1_i}\rho_{2_i}}{\sqrt{\sum_{i=1}^{n} \rho_{1_i}^2}\sqrt{\sum_{i=1}^{n} \rho_{2_i}^2}}$$

Applying this function to the example above would yield a similarity of 0.73.

The vectors can be thought of as hyperplanes in an n-dimensional space (where n is the number of extracted bi-grams). As such, the cosine similarity simply measures the angle between these hyperplane. As all the values in the vectors are non-negative (either 0 or 1), the value returned by the cosine similarity is bounded in the range $[0,1]$: *0* when orthogonal and *1* when parallel.

If the entities are associated with more than one name, then this procedure is applied to all possible pairs and the best result is returned.

5.3.8 Weighted intersection of name tokens

This function is similar to subsection 5.3.3, in that the amount of shared tokens are the key part of the feature. This function supposes that both entities have the list of name tokens. It proceeds to transform these lists into a set of all tokens τ_e for each entity. The common tokens are defined as the set $\iota = \tau_{source} \cap \tau_{target}$, and the set of all tokens is defined as $\upsilon = \tau_{source} \cup \tau_{target}$.

This function also makes use of a list of words which get penalized. This contains mainly common filler words in the names of bands. For example, words like: *the*, *band*, *orchestra*, *boys*, etc. Using this set μ, it is possible to find how many of the shared words should be penalized: $\phi = |\iota \cap \mu|$.

The final result of this function is defined as:

$$\frac{|\iota| - (\phi * 0.9)}{|\upsilon|}$$

The 0.9 is just a weighting term to specify how much the penalization should count. If it was 1, then every shared token which is also in the list of penalized words would not be counted as *shared.* The value of 0.9 was decided empirically. The function is basically the percentage of shared tokens among all tokens in both entities.

5.3.9 Cosine similarity on description

Similarly to subsection 5.3.7, this function leverages the cosine similarity function. The difference is that the function does not count the number of bi-grams. Instead, what is done is that a vocabulary composed of all the words in the description coming from both entities is constructed. Then a vector is created for each entities that specifies if a certain word in the vocabulary is in that entity's description (same as with bi-grams but with words intead). The cosine similarity is then calculated on these two vectors.

5.3.10 Match on occupation QIDs

This feature function is currently only used for IMDb entities since those are the only which have an *occupation* field. Remember that subsection 5.1.1 mentioned how IMDb provided information about the three most frequent professions each entity is associated with, and that these *imdb-professions* can be mapped to the QIDs of professions in Wikidata. This function leverages this relation.

Data coming from Wikidata, when downloaded for IMDb entities, contains all the occupation QIDs associated with the entity. These might not always be the same as those saved in IMDb. For example, the occupation for the Wikidata entity might be a more specific occupation than that in IMDb, or on the other hand it may be less specific. A concrete example of this might be the occupation *voice actor (Q2405480)* in Wikidata. There is no profession in IMDb which matches *voice actor*, however it could be considered as a kind of *actor*, and someone with profession *actor* in IMDb should be considered as a positive match with a Wikidata entity who is a *voice actor* (at least when considering the occupations).

For this reason, this function does not compare the occupations directly, instead it also considers all *sub* and *super* classes of said occupations (more specific and more general versions, respectively). For example, the occupation *voice actor (Q2405480)* is a sub-class of *actor (Q33999)*, and this in turn is a sub-class of *artist (Q483501)*.

The information about which are the super and sub-classes is obtained directly from Wikidata by means of queries to it's SPARQL API[8]. Note that this proceedure gets *all* sub and super classes not only the immediately super or sub. For example, the result of getting the sub-classes of *artist (Q483501)* will include both *actor* and *voice actor*. The process can also be thought of as getting the elements from a tree, where the root is the most general super-class, and the leaves are the most specific sub-classes.

The function starts by getting the occupation sets from two entities, these are then extended to include all the sub and super *occupations* of the original occupations in those sets, creating one ω_e for each entity. The result of the function is then calculated as:

$$\frac{|\omega_{source} \cap \omega_{target}|}{min(|\omega_{source}|, |\omega_{target}|)}$$

This is basically the percentage of the smallest set of occupations which is contained in the largest set. If the base occupations are the same then the sets will also be the same. If the base occupations are the same, but one of the entities has one extra (or more) different occupations, then the smallest set will still be contained in the largest one, and it will be considered as a perfect match. When the base occupations are different, but related (e.g. all occupations are related with music) then it is expected that some non-negligible subset of the super classes will be the same.

Currently, the smallest set is usually IMDb. As seen in Figure 5.3, most IMDb entities only have one occupation associated with them, and they can have at most three. On the other hand, Wikidata entities have no limit as to how many entities they can be associated with.

5.4 Feature Exploration

To better understand the features exposed in the section above, and to learn what is their behaviour in a real-world scenario, this section will proceed to mention some basic statistical information about them. Questions of special interest are:

- Which is the average value and standard deviation for a feature?
- Which features are always zero?
- Which features are extracted for a specific catalog?

[8] https://query.wikidata.org/sparql

- Is there an important variation among features extracted from training and classification data?
    - What about the variation among features for pairs in the training data which are labelled as matches and those that are non-matches?

Table 5.3, presents the features extracted for each concrete entity in the *classification set*. The first number in each cell is the average value, and the number below it, in parenthesis, is the standard deviation. If a cell has no value then it means that the given concrete entity does not have the required fields to extract said features and so it was not used. An example of this is the *born_ similar* which was not applied to entities of the *Discogs* catalog since they do not have birth dates. Note that to make the tables fit nicely in the page, the names of the features have been shortened. Below is a mapping from the complete feature name to its shortened version:

- Similar Birth Dates : brn.sim
- Cosine similarity on description : des.cos
- Similar Death Dates : die.sim
- Exact match on names : nm.exct
- Levenshtein distance on name tokens : nmt.lev
- Weighted intersection of name tokens : nmt.shr
- Cosine similarity on name tokens : nmt.cos
- Match on occupation QIDs : occ.shr
- Exact match on URLs : url.exct
- Shared tokens in URLs : url.shr

By looking at Table 5.3, the first thing one can notice is that all catalogs have seven features, except for *Musicbrainz* which has eight. It is also clear which features were extracted from which catalogs.

The mean for some features seem to be maintained similar across each concrete entity. For example the born similarity is 0.005 ± 0.001 in 5 out of 7 concrete entities that have it. However, *nm.exct* varies a bit more, with it going from *0.04* for *Discogs/Musician* to *0.14* for *Musicbrainz/Band.*

	discogs		imdb					musicbrainz	
	band	musician	actor	director	musician	producer	writer	band	musician
brn.sim			0.005 (0.067)	0.006 (0.076)	0.003 (0.050)	0.005 (0.073)	0.005 (0.068)	0.000 (0.000)	0.006 (0.076)
des.cos	0.010 (0.042)	0.013 (0.040)							
die.sim			0.001 (0.032)	0.001 (0.037)	0.001 (0.029)	0.002 (0.046)	0.003 (0.053)	0.000 (0.000)	0.001 (0.037)
nm.exct	0.077 (0.266)	0.040 (0.197)	0.107 (0.308)	0.091 (0.288)	0.042 (0.200)	0.122 (0.328)	0.070 (0.254)	0.140 (0.347)	0.051 (0.220)
nmt.lev	0.544 (0.271)	0.658 (0.272)	0.467 (0.131)	0.457 (0.130)	0.452 (0.128)	0.465 (0.124)	0.452 (0.123)	0.489 (0.261)	0.470 (0.163)
nmt.shr	0.307 (0.215)	0.197 (0.132)	0.306 (0.197)	0.321 (0.205)	0.278 (0.140)	0.317 (0.217)	0.265 (0.173)	0.402 (0.260)	0.285 (0.162)
nmt.cos	0.527 (0.252)	0.437 (0.207)	0.554 (0.177)	0.549 (0.180)	0.510 (0.147)	0.551 (0.186)	0.511 (0.166)	0.608 (0.215)	0.515 (0.162)
occ.shr			0.014 (0.103)	0.123 (0.289)	0.103 (0.271)	0.033 (0.119)	0.017 (0.084)		
url.exct	0.001 (0.037)	0.001 (0.026)						0.035 (0.183)	0.006 (0.078)
url.shr	0.012 (0.091)	0.004 (0.049)						0.047 (0.191)	0.008 (0.077)

Table 5.3: Average features and their standard deviation. Extracted for classification dumps.

On the other hand, Table 5.4 shows the mean and standard deviation for features extracted from the training set. The first thing that can be noticed is that in almost all cases the *mean* is higher for features in the training set than for the respective feature/concrete-entity in the classification set. Another thing that should be pointed out is that the standard deviation in the training data features is also higher in most cases.

The above suggests that there is a substantial difference among training and classification data. To better understand this difference, Figure 5.17 shows two heatmaps which have been generated by drawing the respective mean feature values. The heatmap shows in gray those feature/concrete-entities for which there is no value, and the rest is showed in some shade of red.

It is easy to see that features do behave similarly across the different heatmaps. For instance, features which are darker (higher mean) in the classification set are also, generally, darker in the training set. However, many features seem to have a much lower mean in the classification set than in the training set. This might mean that either there are more negative examples in the classification data, or the quality of the classification data is not very good.

Examples which can readily be discerned from the graphic are: *nm,exct*, *born and died similar*, *occ.shr*, *url.exct*, *url.shr*, and *des.cos*. Interestingly, it also seems that *nmtshr* has, in most cases, a higher mean than the respective entries in the classification set.

	discogs		imdb					musicbrainz	
	band	musician	actor	director	musician	producer	writer	band	musician
brn.sim			0.121 (0.326)	0.090 (0.286)	0.089 (0.285)	0.088 (0.283)	0.112 (0.315)	0.000 (0.004)	0.175 (0.378)
des.cos	0.028 (0.074)	0.037 (0.077)							
die.sim			0.043 (0.203)	0.036 (0.186)	0.038 (0.191)	0.032 (0.177)	0.053 (0.225)	0.000 (0.010)	0.060 (0.236)
nm.exct	0.316 (0.465)	0.247 (0.431)	0.205 (0.403)	0.183 (0.387)	0.177 (0.382)	0.177 (0.382)	0.186 (0.389)	0.285 (0.452)	0.234 (0.423)
nmt.lev	0.622 (0.280)	0.732 (0.274)	0.475 (0.127)	0.462 (0.123)	0.468 (0.132)	0.466 (0.122)	0.464 (0.123)	0.547 (0.276)	0.511 (0.185)
nmt.shr	0.457 (0.330)	0.264 (0.254)	0.291 (0.244)	0.290 (0.243)	0.262 (0.223)	0.234 (0.222)	0.246 (0.222)	0.447 (0.303)	0.303 (0.241)
nmt.cos	0.641 (0.291)	0.494 (0.272)	0.541 (0.210)	0.526 (0.210)	0.506 (0.201)	0.484 (0.203)	0.499 (0.203)	0.641 (0.244)	0.535 (0.216)
occ.shr			0.028 (0.129)	0.209 (0.308)	0.185 (0.321)	0.137 (0.229)	0.141 (0.225)		
url.exct	0.065 (0.246)	0.038 (0.191)						0.195 (0.396)	0.157 (0.363)
url.shr	0.127 (0.295)	0.078 (0.245)						0.122 (0.239)	0.108 (0.242)

Table 5.4: Average features and their standard deviation. Extracted for training dumps.

Now, the used features can be said to be effective if they acually separate the positive from negative examples. For this purpose, Table 5.5 and Table 5.6 present the mean and standard deviation values for positive and negative training examples respectively. It is easily seen that in most cases the mean values in the positive samples are much higher than those in the negative samples, meaning that the features are indeed significant to distinguish whether a sample is positive or not.

Figure 5.18 shows two heatmaps similar to those described for the classification and training sets. It is easy to see that training samples have a higher mean value for almost all features/concrete-entities. An interesting thing to notice, however, is that *nmt.lev* and *nmt.cos* have quite high mean values compared to all the other values in the *negative set*. A reason for this might be that these string metrics are good at detecting partial matches on text, while the other feature metrics work by either predicting an exact match or the number of shared tokens (which themselves must exactly match).

Among all features it seems that the most significant ones for separating positive and negative samples are:

- *nm.exct*, which is almost 1 in the positive set, and very low in the negative set. This is expected considering that the training set was created to match on this value.
- *born and died similar* are much higher for the positive set.
- *url features* are also much higher for positive samples, meaning that the URLs related to an

entity are actually good identifiers of entities.

- Finally, it should also be pointed out that the *occ.shr* feature is similar in value between the two sets, but is a little higher in positive samples. An exception to this last is *IMDb/Actor*, which seems to have the same mean value for this feature in both sets. Quite possibly the reason for this is because negative samples are also constructed from the same type of entity as the positive samples (e.g. *musician, actor, director, etc*), so it is expected for some of their occupations to match.

	discogs		imdb					musicbrainz	
	band	musician	actor	director	musician	producer	writer	band	musician
brn.sim			0.683 (0.465)	0.538 (0.499)	0.588 (0.492)	0.581 (0.493)	0.659 (0.474)	0.000 (0.008)	0.800 (0.395)
des.cos	0.057 (0.108)	0.102 (0.121)							
die.sim			0.244 (0.430)	0.218 (0.413)	0.253 (0.435)	0.217 (0.412)	0.318 (0.466)	0.000 (0.020)	0.278 (0.446)
nm.exct	0.945 (0.228)	0.987 (0.112)	0.953 (0.211)	0.955 (0.207)	0.966 (0.182)	0.964 (0.185)	0.963 (0.190)	0.968 (0.175)	0.977 (0.150)
nmt.lev	0.758 (0.252)	0.885 (0.203)	0.527 (0.087)	0.525 (0.093)	0.541 (0.109)	0.529 (0.089)	0.525 (0.090)	0.729 (0.257)	0.609 (0.193)
nmt.shr	0.774 (0.306)	0.584 (0.319)	0.589 (0.341)	0.606 (0.351)	0.549 (0.349)	0.495 (0.353)	0.511 (0.343)	0.786 (0.298)	0.590 (0.326)
nmt.cos	0.883 (0.224)	0.816 (0.218)	0.813 (0.184)	0.819 (0.186)	0.785 (0.193)	0.755 (0.201)	0.774 (0.188)	0.895 (0.184)	0.802 (0.194)
occ.shr			0.073 (0.188)	0.458 (0.341)	0.470 (0.390)	0.369 (0.279)	0.324 (0.269)		
url.exct	0.218 (0.413)	0.172 (0.377)						0.819 (0.385)	0.732 (0.443)
url.shr	0.398 (0.418)	0.328 (0.425)						0.468 (0.255)	0.488 (0.289)

Table 5.5: Average features and their standard deviation extracted for positive training samples (those labelled as matches).

	discogs		imdb					musicbrainz	
	band	musician	actor	director	musician	producer	writer	band	musician
brn.sim			0.005 (0.067)	0.004 (0.064)	0.004 (0.061)	0.003 (0.059)	0.005 (0.068)	0.000 (0.000)	0.007 (0.078)
des.cos	0.016 (0.049)	0.019 (0.045)							
die.sim			0.001 (0.034)	0.001 (0.032)	0.001 (0.032)	0.001 (0.029)	0.001 (0.037)	0.000 (0.000)	0.001 (0.029)
nm.exct	0.057 (0.231)	0.045 (0.207)	0.050 (0.217)	0.036 (0.186)	0.042 (0.202)	0.043 (0.202)	0.034 (0.181)	0.075 (0.263)	0.033 (0.179)
nmt.lev	0.566 (0.272)	0.691 (0.277)	0.464 (0.131)	0.450 (0.125)	0.456 (0.132)	0.456 (0.123)	0.452 (0.125)	0.491 (0.256)	0.484 (0.174)
nmt.shr	0.327 (0.238)	0.176 (0.137)	0.229 (0.159)	0.230 (0.155)	0.213 (0.145)	0.190 (0.151)	0.194 (0.138)	0.343 (0.217)	0.226 (0.129)
nmt.cos	0.541 (0.254)	0.407 (0.212)	0.485 (0.166)	0.471 (0.162)	0.458 (0.159)	0.437 (0.164)	0.445 (0.156)	0.563 (0.203)	0.462 (0.157)
occ.shr			0.018 (0.111)	0.162 (0.277)	0.136 (0.280)	0.097 (0.193)	0.105 (0.197)		
url.exct	0.001 (0.035)	0.002 (0.040)						0.002 (0.047)	0.001 (0.035)
url.shr	0.015 (0.090)	0.010 (0.072)						0.015 (0.079)	0.005 (0.048)

Table 5.6: Average features and their standard deviation extracted for negative training samples (those labelled as not matches).

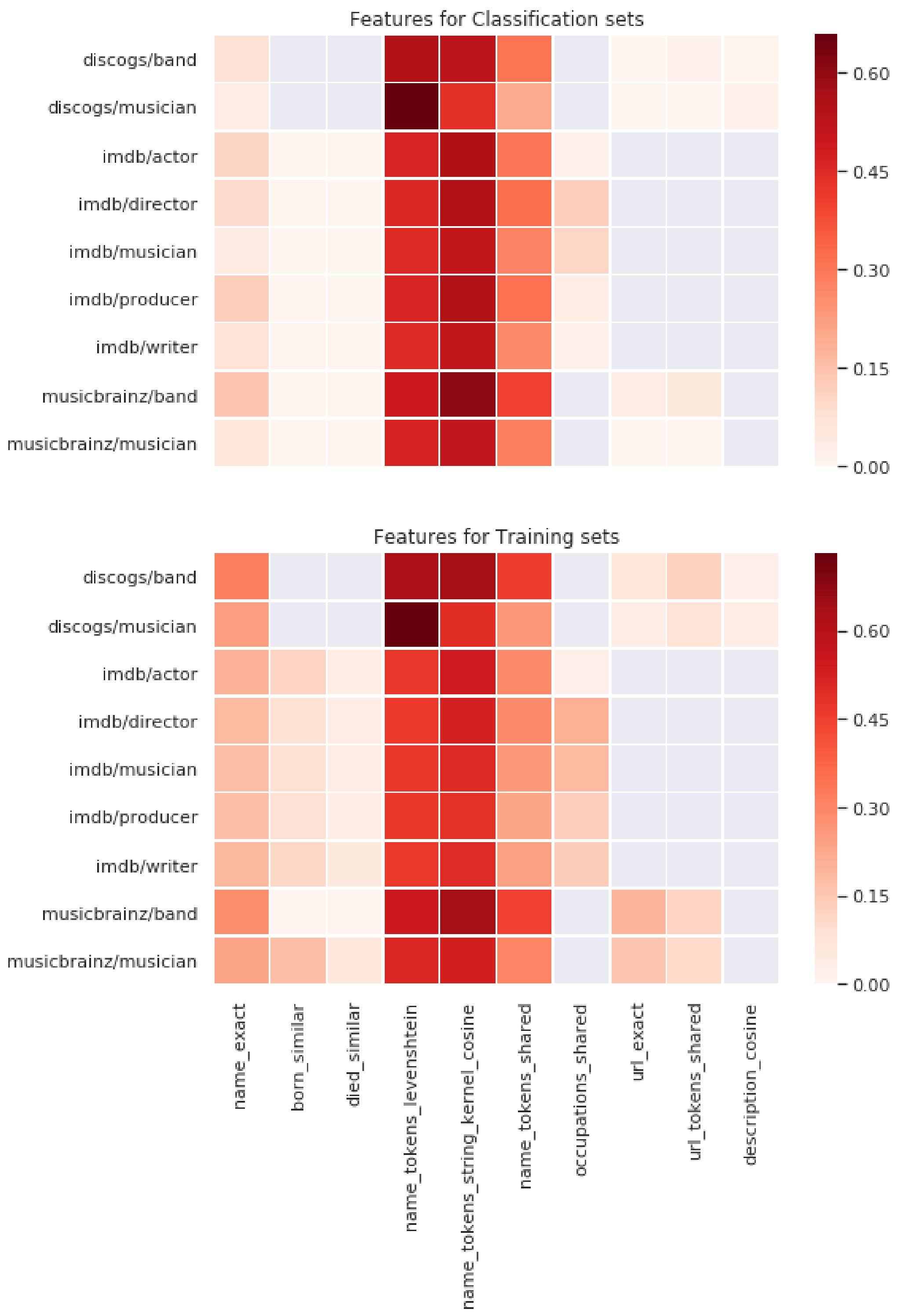


Figure 5.17: Average values of the features obtained for each concrete entity. Separated by classification and training datasets.

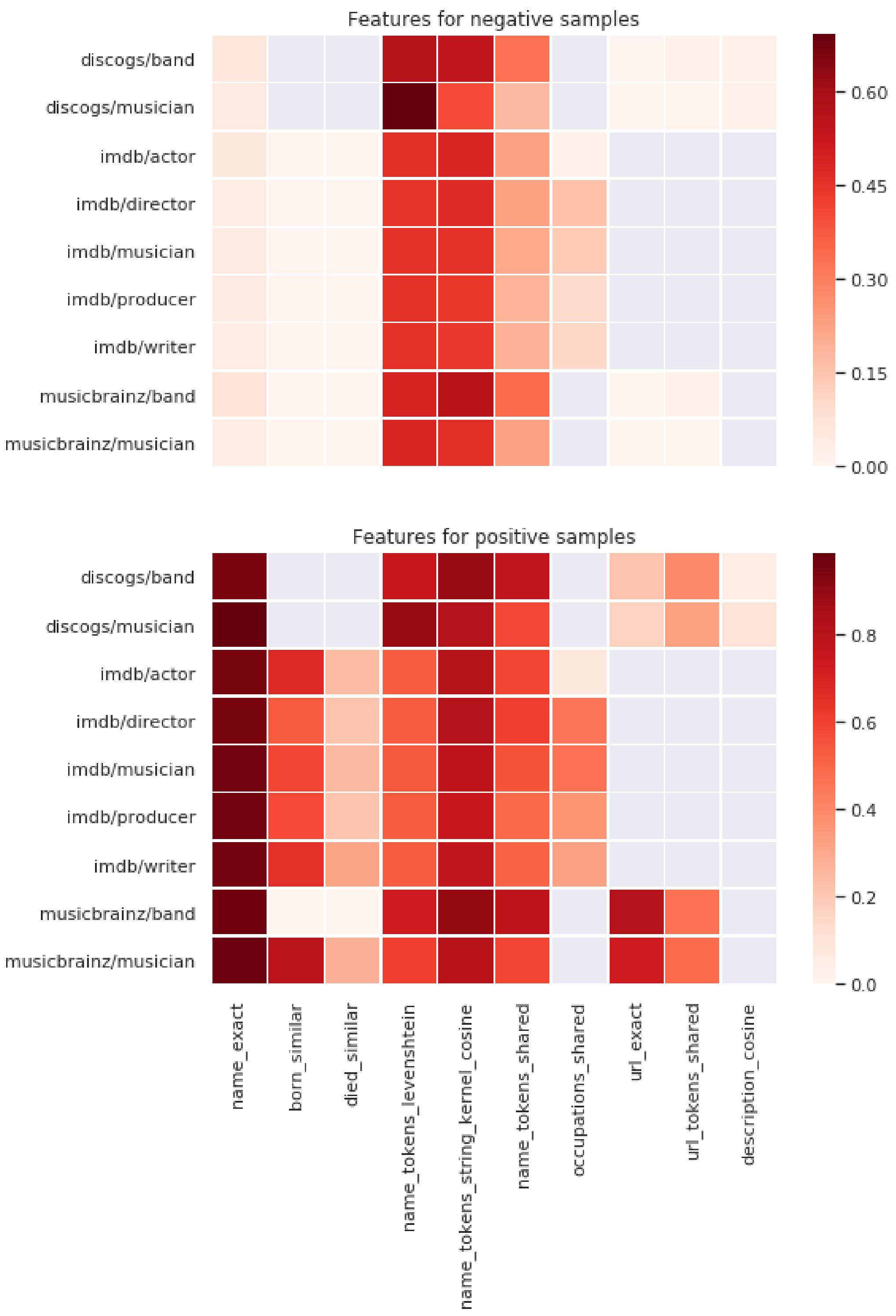


Figure 5.18: Average values of the features obtained for each concrete entity. Separated by positive (matches) and negative (non-matches) labelled points in the training data.

For completeness, section A.2 contains distribution plots of the feature values for training and the classification sets, and also for positive and negative samples.

5.5 Difference between training and real-world data

Seeing that the catalogs used are maintained mainly by contributors, one may think that more important people/bands have more information on them than less important ones. From this follows that these entities may have already been linked with the respective external catalog entity. As such, one can assume that the positive samples in the training data are composed mainly of these already labelled, *more important*, entities, and the classification data are *less important* entities, and less accurately labelled.

This hypothesis, that training data is better quality than classification data, may impact the ability to extract meaningful features from the data. For example, it may be more common in the classification set than in the training set that an entity has no *birth date*, or that this date is wrong.

Table 5.7 and Table 5.8 show the percentage of zero values obtained for each feature, for each concrete entity. Most features behave similarly across the different sets, for example *nmt.lev* which is always positive. Of more interest are the features *born and die similar*, which are almost always zero in the classification set, but are considerably *better* in the training set. Similarly, *nm.exact* appears to have much more zero values in the classification data than in the training data.

If training and classification sets were of the same quality then this difference can be attributed to the simple fact that the classification set contains more negative examples than the training set. However, in the current case there is no way to know except by evaluating the classification predictions, which is something that will be done by the Wikidata community once said predictions are uploaded.

Regardless, this is something that must be kept in mind since the final performance of the algorithms on the classification data may actually be different from that on the training data.

		discogs		imdb					musicbrainz	
		band	musician	actor	director	musician	producer	writer	band	musician
C	brn.sim			0.995	0.994	0.997	0.995	0.995	1.000	0.993
	des.cos	0.919	0.850							
	die.sim			0.999	0.999	0.999	0.998	0.997	1.000	0.998
	nm.exct	0.923	0.960	0.893	0.909	0.958	0.878	0.930	0.860	0.949
	nmt.lev	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
	nmt.shr	0.000	0.001	0.002	0.002	0.001	0.002	0.001	0.000	0.001
	nmt.cos	0.064	0.058	0.000	0.000	0.000	0.000	0.000	0.005	0.001
	occ.shr			0.978	0.815	0.851	0.920	0.955		
	url.exct	0.999	0.999						0.965	0.994
	url.shr	0.975	0.988						0.931	0.984
T	brn.sim			0.879	0.910	0.911	0.912	0.888	1.000	0.822
	des.cos	0.783	0.666							
	die.sim			0.957	0.964	0.962	0.968	0.947	1.000	0.939
	nm.exct	0.684	0.753	0.795	0.817	0.823	0.823	0.814	0.715	0.766
	nmt.lev	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
	nmt.shr	0.000	0.000	0.001	0.001	0.001	0.000	0.000	0.000	0.001
	nmt.cos	0.053	0.065	0.000	0.000	0.000	0.000	0.000	0.006	0.002
	occ.shr			0.944	0.618	0.699	0.698	0.681		
	url.exct	0.935	0.962						0.805	0.843
	url.shr	0.809	0.882						0.750	0.806

Table 5.7: Percentage of zero values extracted for a given feature, for each concrete entity. Results have been separated by set of procedence: (C)lassification, (T)raining.

		discogs		imdb					musicbrainz	
		band	musician	actor	director	musician	producer	writer	band	musician
N	brn.sim			0.995	0.996	0.996	0.997	0.995	1.000	0.992
	des.cos	0.843	0.747							
	die.sim			0.999	0.999	0.999	0.999	0.999	1.000	0.999
	nm.exct	0.943	0.955	0.950	0.964	0.958	0.957	0.966	0.925	0.967
	nmt.lev	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
	nmt.shr	0.000	0.001	0.001	0.001	0.001	0.001	0.000	0.000	0.001
	nmt.cos	0.061	0.074	0.000	0.000	0.000	0.000	0.000	0.006	0.002
	occ.shr			0.965	0.685	0.765	0.764	0.746		
	url.exct	0.999	0.998						0.998	0.999
	url.shr	0.959	0.967						0.944	0.975
P	brn.sim			0.317	0.462	0.412	0.419	0.341	1.000	0.191
	des.cos	0.637	0.370							
	die.sim			0.756	0.782	0.747	0.783	0.682	1.000	0.718
	nm.exct	0.055	0.013	0.047	0.045	0.034	0.036	0.037	0.032	0.023
	nmt.lev	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
	nmt.shr	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
	nmt.cos	0.035	0.031	0.000	0.000	0.000	0.000	0.000	0.009	0.002
	occ.shr			0.844	0.265	0.313	0.314	0.351		
	url.exct	0.782	0.828						0.181	0.268
	url.shr	0.446	0.571						0.118	0.179

Table 5.8: Percentage of zero values in features extracted for (P)ositive and (N)egative samples of the training set.

6 Algorithms

This chapter will cover the machine learning algorithms used in the project. The first section will talk about the *baseline* classifiers, a set of standalone classification algorithms. Each algorithm has *hyper-parameters*, which are parameters that regulate the functionality of different parts of each algorithm. These are unique to each, and the choice of the best hyperparameter is usually different for diverse problems. section 7.2 will talk about how and according to which procedures the hyperparameters for these baseline algorithms were chosen. Following that, section 6.2 will talk about ensemble models, the motivation for using them, and which techinques for creating them were employed in the project.

As said in subsection 2.2.4, the linking process is one of the main processes in the record linkage workflow. This linking can either be done using a rule-based approach, or a machine learning algorithm can be used to learn when to classify a pair as a match based on the features extracted for that pair. As such, having a well tuned and well performing algorithm is essential for an effective record linkage.

A machine learning algorithm [62] is an algorithm capable of learning something when given some data. There are many kinds, but those that are of interest for this project are *supervised algorithms*, which are able to draw some inferences about data they've never seen (classification data) when first shown some data similar to that inferences want to be drawn about (training data).

This capacity of the algorithm of correctly predicting never before seen samples of data is called *generalization*, and it hints at the fact that the algorithm is learning some underlying structure, or patter, in the data it trains on.

For the current project, these algorithms will be shown pairs of entities that are the same, and pairs that are not, and the goal is for the algorithm to discern, for a never seen before pair, whether it is a match or not.

6.1 Baseline Classifiers

All of the baseline algorithms (except for *mlp* and *slp*, which come from the same family) used are based on a different underlying approach of understanding data and learning something from it. As such, all of these capture the information seen in the training data in different ways.

The above means that there is a fundamental difference among most of these baseline algorithms, and choosing a *best* one is a very difficult task, since it really depends on the application it is needed for.

This section strives to explain each baseline algorithm so that the differences among them may be understood. In chapter 7 the results of these baseline algorithms will be compared against that obtained by the ensemble models (explained in the next section), to see if there is any improvement when employing the idea of considering more than one algorithm to get a prediction.

6.1.1 Linear SVM

In its simplest sense, SVM [22, 23], and by extension LSVM, are models that learn to segment data into two categories (a binary classifier). The model supposes that the data is sees is spread on a space of n dimensions. When training, the SVM model will learn a hyperplane (which is just an $(n-1)$-dimensional line) that separates the points in each category by a *gap* which should be wide as possible. The implementation of LSVM used in this project is the one provided by the LIBLINEAR [33] library, which is accessed using the functionality provided by Scikit-Learn [57].

Suppose that an SVM gets as training input a series of points in the following format: $(\vec{x}_1, y_1), ..., (\vec{x}_n, y_n)$. Where $\vec{x}_i$ is an n-dimensional vector representing the point, and y_i is the value saying to which class this vector belongs to: either -1 or 1. As said above, the goal is to find a hyperplane which is as far as possible from the nearest points in each group (see Figure 6.1). A hyperplane can be thought of as a series of $\vec{x}$ for which the following is true:

$$\vec{w} \cdot \vec{x} - b = 0$$

When classifying new samples, the model will suppose these new samples are points in a space, and then just checks on which side of the hyperplane the new points are, and assign the respective label accordingly.

SVM algorithm can also modify the input data that they get in such a way that the space representation they learn is different from the one the input data actually represents. This is usually done to map the input data to a higher dimensional space, making it *"easier"* to separate by a gap. This is called the *kernel trick* [7, 23]. However, this technique won't be used in the current work since what will be used is an SVM which separates said space *linearly*. The main reason why non-linear SVMs were not used is because the training time for the current implementation offerred by Scikit-Learn is "... scales at least quadratically with the number of samples and may be impractical beyond tens of thousands of samples"[1].

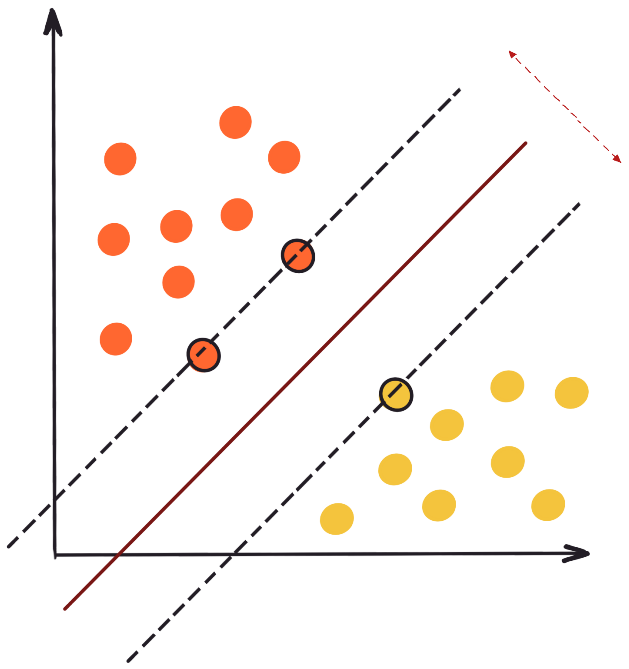


Figure 6.1: Representation of the hyperplane found by an SVM with the gap that separates the categories. The colored points represent points in the *input space*, each color is a different category. The bold points are the *support vectors* found in this specific instance, which is just another name for points on the margin.

If $\vec{w}$ is the vector defining the hyperplane, then the constraints which define on which side of the margin the data points should be is defined as:

$$\begin{cases} w^T x_i \geq 1, & \text{if } y_i = 1 \\ w^T x_i \leq -1, & \text{if } y_i = -1 \end{cases}$$

The size of this margin can then be calculated by measuring the distance among the hyperplanes that pass through the points defined at the limits of this constraints (the hyperplanes that go through the support vectors): when they are $w^T x_i = 1$ and $w^T x_i = -1$ respectively. This distance is given by:

$$\frac{2}{\|w\|}$$

And as such, minimizing $\|w\|$, subject to the constraints presented above, will increase the margin while keeping all points correctly classified.

This is what is called a **hard margin**, meaning that points must *strictly* be on one side of the margin or the other, and is only useful if the data is linearly separable. The implementation in LIBLINEAR [33] uses a **soft margin**, which allows the points to be inside the margin, but adds a penalty which is proportional from the distance to the correct margin. This can be defined by the following optimization problem:

[1]https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC

$$\min_{\|w\|} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \zeta(w, x_i, y_i)$$
$$\zeta(w, x_i, y_i) = \max(0, 1 - y_i w^T x_i)^2$$

Where w is the vector defining the hyperplane, x_i and y_i are the input features and label for element i. $C > 0$ is a penalty parameter and $\zeta(w, x_i, y_i)$ is the *penalty function*, which is 0 whenever the $w^T x_i$ gives the correct label. This $w^T x_i$ is ≥ 1 if $y_i = 1$, and is ≤ -1 if $y_i = -1$.

The C parameter regulates how important is it that points are on the correct side of the margin, a higher C will increase the penalty, while a lower C will decrease it. This is a *hyperparameter*, whose choice must be made on a *per problem* basis.

Note that in many SVM representations there is also a *bias* term B which is part of the equations. In this case, however, the authors of the LIBLINEAR implementation decided to append B to the end of $\vec{w}$ and $\vec{x}$.

The method used to solve this optimization problem is a *coordinate descent method*, presented in [43]. Which approximates the optimal $\vec{w}$ by a at most ϵ, which is also called the *tolerance margin*. The approximation is done in $O(log(1/\epsilon))$ iterations. This ϵ is another hyperparameter that must be tuned for *lsvm*.

Once this optimum $\vec{w}$ is found, then it can be easily used for classification. New points are classified as follows:

$$y_p = \begin{cases} 1, & \text{if } w^T x > 0 \\ -1, & \text{otherwise} \end{cases}$$

6.1.2 Naive Bayes

The implementation used for this classifier is that of *Bernoulli Naive Bayes* [50, 52] classifier provided by the Scikit-Learn library [57]. It expects the features in the input vector $\vec{x}$ to be composed of only *binary values* (*1 or 0*). If the value of a feature is not binary (e.g. 0.6) then it is converted to binary by checking if the value is above a certain threshold *binarize*, if it is then it is converted to 1, and 0 otherwise. This threshold is one of the hyperparameters of this classifier.

For this model, it is supposed that all features in a feature vector are statistically independent [50] given the class said feature vector represents.

Suppose that the input to the model is a feature vector $\vec{x}$ where x_i represents the presence of a specific feature i in the vector (either 0 or 1). Suppose also that y stands for one of the possible classes $\vec{x}$ can be assigned to. The probability of getting a class y given the feature vector $\vec{x}$ received as input is:

$$P(y|\vec{x}) = P(y|x_1, x_2, ..., x_n)$$

Using Bayes theorem this can be expressed as:

$$P(y|\vec{x}) = \frac{P(y)P(\vec{x}|y)}{P(\vec{x})}$$

Where $P(y)$ is the probability of a given class, and is termed the *prior*. $P(\vec{x}|y)$ is the probability of seeing the specific feature vector considering that vector is of class y, this term is named the *likelihood*. Finally, $P(\vec{x})$, which is named the *evidence*, shows how likely is it, in general, to see the given feature vector.

Since the interest is in finding out which is the actual y. Only the *prior* and *likelihood* are necessary for this. The *evidence* can be thought of as a *scaling* constant. By using the *chain rule*, the numerator in the equation above can be presented as the joint probability $P(y)P(\vec{x}|y) = P(x_1, x_2, ..., x_n, y)$. And again by using the chain rule, this can be expressed as:

$$P(x_1, x_2, ..., x_n, y) = P(x_1|x_2, x_3, ..., x_n, y)P(x_2, x_3, ..., x_n, y)$$

This process can be continued until the joint probability is expanded to:

$$P(x_i|x_{i+1}, x_{i+2}, ..., x_n, y)...P(x_{n-1}|x_n, y)P(x_n|y)P(y)$$

As said before, the assumption is that all these x_i are independent among themselves given the class y (this is the *naive* assumption in *Naive Bayes*). Because of this, the chain of probabilities above can be shortened to:

$$\begin{aligned} P(y)P(\vec{x}|y) &\approx P(x_i|y)P(x_{i+1}|y)...P(x_n|y)P(y) \\ &= P(y)\prod_{i=1}^{n} P(x_i|y) \end{aligned}$$

Finally, the original formulation of the probability of a class given the input feature vector can be expressed as

$$P(y)|\vec{x}) = \frac{P(y)\prod_{i=1}^{n} P(x_i|y)}{P(x_1, x_2, ..., x_n)}$$

The denominator $P(x_1, x_2, ..., x_n)$ is constant given the input, and as such it just scales the probability of the predicted class but holds no actual information about what the class is. To find the class it is only necessary to see which, among the possible classes, has the higher probability. This means that the denominator can be safely ignored.

$$\hat{y} = \underset{y}{\text{argmax}}\ P(y)\prod_{i=1}^{n} P(x_i|y) \tag{6.1}$$

The first term $P(y)$ is simply the frequency of each class: what percentage of the time this class is present in the training data. And the second parameter is defined as follows:

$$P(x_i|y) = P(i|y)x_i + (1 - P(i|y))(1 - x_i)$$

In the equation above, the first term, $P(i|y)x_i$, is zero when $x_i = 0$, while the second term is zero when $x_i = 1$. The first term indicates the probability of i appearing in class y, while the second term is the complement: the probability of not having feature i in class y.

$P(i|y)$ is the smoothed probability of having the feature i appear in the class y. This is calculated as follows

$$P(i|y) = \frac{N_{yi} + \alpha}{N_y + \alpha 2}$$

Where $N_{yi} = \sum_{x \in T} x_i$ is the count of all the times feature i appears in class y in the training set T. And $N_y = \sum_{i=1}^{n} N_{yi}$ is the total number of features for y in the training set. α is a hyperparameter of the model and must be set accordingly. However, it is common to set its value > 0 to account for the problem of getting a feature not seen before. If $\alpha = 0$ and $N_{yi} = 0$ (i has never been seen for y in the training set), then when evaluating Equation 6.1 the value for y would be 0.

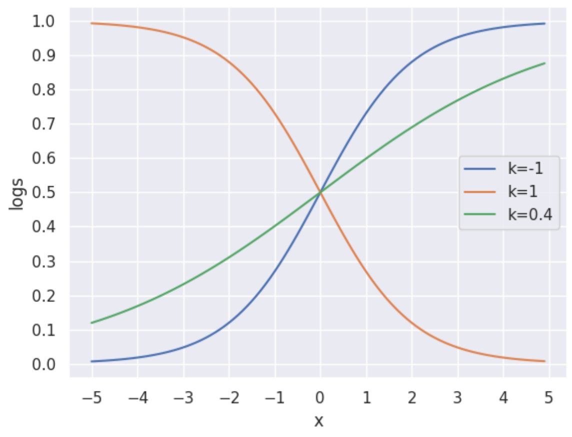
The assumption that the features are independent among themselves means that the probabilities mentioned above can be separately calculated for each feature. This makes naive bayes is efficient even when the input has a high number of dimensions [75].

6.1.3 Logistic Regression

Even though it has *regression* in its name, this model is actually a classifier, and is specifically used to make binary predictions. To clarify the meaning of *regression*, it is a machine learning task in which the predicted output is continuous (i.e. a numerical value) instead of a class label. The probability predicted by this classifier is modelled using a logistic function. This function is defined as:

$$f(x) = \frac{L}{1 + e^{-k(x - x_0)}}$$

Where L is the maximum value this for function, which for the current application will be 1 since it is a probability. x_0 is the value on the x-axis at which the function has a value of $L/2$. And k is the steepness of the curve. The function will approaches its limits asymptotically: L as x approaches $+\infty$, and 0 when it approaches $-\infty$. The following figure shows the curves given by this function with $L = 1$, $x_0 = 0$, and different values for k.



The implementation of logistic regression used for this project works in a similar way as linear SVM, explained in subsection 6.1.1. In fact, this implementation is also provided by LIBLINEAR [33]. As *lsvm*, this model proceeds to solve an optimization problem with the data available during training, the formulation is quite similar and the only thing that changes is the *loss function*:

$$\min_{\|w\|} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \zeta(w, x_i, y_i)$$
$$\zeta(w, x_i, y_i) = \log(1 + e^{-y_i w^T x_i})$$

This optimization problem is solved with the *coordinate descent method*, presented in [43] (which is the same method used to solve the optimization for *lsvm*). Remember that LIBLINEAR appends the bias parameter B to the end of $\vec{w}$ and $\vec{x}$, so this does not appear explicitly in the equation to be optimized. y_i is the class label and should be either -1 or 1. Finally, as for *lsvm*, C is a penalty constant.

The goal is to find the parameters, in this case $\vec{w}$, which best describes best the logistic function that represents the actual probability of an event given the an input vector.

After training the probability [33] of a class y given the input vector $\vec{x}$ can be obtained by evaluating the logistic function with the parameters found during optimization.

$$P(y|\vec{x}) = \frac{1}{1 + e^{-yw^T x}} \tag{6.2}$$

6.1.4 Random Forest

An *ensemble forest* [11] is a collection of *decision trees* classifiers. This is actually an ensemble, specifically a *bagging ensemble*, which will be explained in further detain in section 6.2. The *decision tree* classifier will be explained shortly.

The idea behind *random forest* is to have a collection of *decision trees* which have been trained on different subsets of the data. The idea is to take simple, and potentially unstable models, and join their predictions so that the final predictor has less variace [9].

When creating a random forest, a collection of k decision trees is created. If the training data T is composed of n samples then each decision tree is trained on n samples. However, instead of giving all trees the same set of samples, what is done is that a new training set T'_i is obtained for each tree. This T'_i is constructed by random sampling, with replacement, until the amount of samples in T'_i is the same as in T. *With replacement* means that sampling the same entry from T multiple times is allows.

Each *decision tree* is trained individually, on their own training set T'_i. And when it comes the time to predict the label for a feature vector x, the prediction of all decision trees will be asked for, and these will be averaged to get the final prediction [57].

The implementation of *random forest*, and of *decision trees*, comes from the Scikit-Learn [57] library.

Decision Tree

A *decision tree*, as the name implies, is just a series of simple *decisions rules* which are implied from the data during training. These rules are of form: *if something then do this, otherwise do this other thing*. Most commonly than not, decision trees have multiple levels, meaning that these decisions rules are nested.

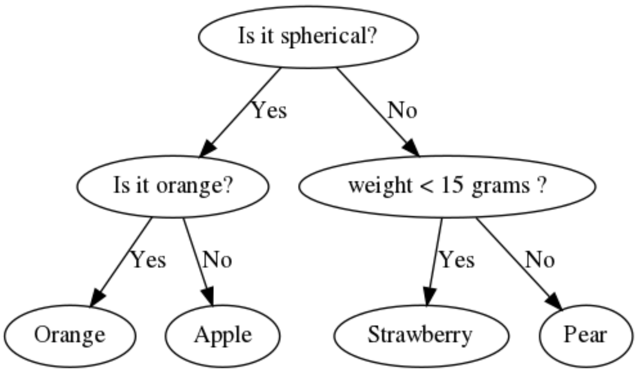


Figure 6.2: Example of a decision tree which classify fruits according to their shape, solor, and weight.

It can be seen that at each step in the decision tree in Figure 6.2 a test on a different *value* is made, and each branch specifies what to do depending on the outcome of said test. The *leaves* (i.e. the bottom nodes) of the tree specify the outcome of the evaluation. The top node of the tree is called the *root*, and specifies the starting point for the evaluation.

The first node created for a new decision tree is the *root node*. The way in which the algorithm decides which attribute to test at each node is by finding the test which will make the greatest reduction on the entropy of the subsets, or in other words, the greatest increase in *information gain*. Both of these concepts are explained further in depth below.

In example Figure 6.2 the first test made is on the shape of the fruit, if multiple points are evaluated on this decision tree at the same time then it can thought of as if they splitted at this node: all *spherical* fruits go to the left branch, while all those that are not spherical take the right branch. These two divisions are what are referred two when saying that the entropy is measured on the resulting subsets.

Entropy is no other than the degree of disorder in a set. A completely homogeneous set has an entropy of 0, while the largest value is obtained when the different classes in the set have the same amount of representatives. For the current project, the amount of classes being used is 2 (match and non-match). When using only two classes, the maximum value for the entropy is 1. The total entropy of a set S is measured with the following formula:

$$H(S) = \sum_{i=1}^{C} -p_i \log_2 p_i$$

Where C are all the classes in the set, and p_i is the percentage of times that class i appears in the set. To calculate the entropy of the subsets, after splitting the data on a node, the sum of the entropy of all subsets needs to be calculated. Continuing with the fruits example, say the set coming from the node which did the splitting is T, and there are $z \in T$ which are *spherical*, and $k \in T$ which are not. The entropy after splitting on whether the fruit is spherical or not is calculated as:

$$H(S|Spherical) = \frac{|z|}{|T|}H(z) + \frac{|k|}{|T|}H(k)$$

Where $|\cdot|$ denotes the amount of elements in the set. It can be seen that the above is the entropy of each subset weighted by the percentage of elements in that subset with respect to the number of elements in the parent set (which is T in this case).

In more general terms, the above can be seen as a way in which to measure the total entropy of a set after it has been splitted on an attribute A. Finally, the information gain is just the change in entropy obtained when splitting the original set on said A.

$$IG(S, A) = H(S) - H(S|A)$$

The attribute on which each node splits, for a given dataset, is then selected as:

$$\underset{A}{\operatorname{argmax}}\ IG(S, A)$$

6.1.5 Single Layer Perceptron

The Single Layer Perceptron (SLP) [36] is an algorithm which can learn a linear function by observing some data. Single layer perceptrons have also been used with success in record linkage problems in [70]. The implementation of SLP used is provided by the Keras library [13].

For a series on inputs $(\vec{x}, y)$, where $\vec{x}$ are vectors of features and y are the classes, an SLP can be thought of a model which learns the following parameters $\vec{w}$ and b:

$$y_i \approx \lambda(\vec{w} \cdot \vec{x} + b)$$

Where $\vec{w}$ is the vector of weights, which is the same shape as the input vector $\vec{x}$. $\cdot$ is the *dot product* between the input vector and the *weights vector*. λ is called the *activation function* and is usually a nonlinear transformation on the weighted sum coming from the inputs.

The activation function used for this project is the *sigmoid* function, which is very similar to the logistic function defined in subsection 6.1.3, and is defined as $\sigma(x) = \frac{1}{1+e^{-x}}$. Using a sigmoid as the activation function allows to interpret the output as the probability of belonging to a class.

The following figure shows an example of a SLP with $\|x\| = \|w\| = 4$.

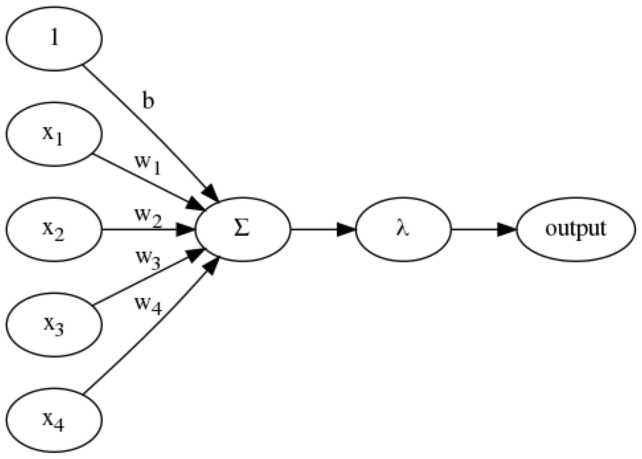


Figure 6.3: Example of a SLP with 4 inputs.

For completeness, provided below is the algorithm that can be used to learn the parameters $\vec{w}$ and b from samples $(x_i,\ y_i)$ in a training set T. Note that y_i' is the prediction the perceptron gives for x_i. δ is the learning rate, and is a scalar which regulates how fast the weights are updated. *Epochs* is the number of times the training set will be processed. For this project, instead of using the basic algorithm mentioned below, the optimization algorithm *Nadam* [29] is used.

```
w, b = initializeWAndB() # initialized to small random values

for e in 0..$epochs$:

    for $x_i, y_i$ in $T$:
        # get prediction
        $y'_i$ = $\lambda(\vec{w} \cdot \vec{x} + b)$
        $dy = y_i - y'_i$

        # tune the weights based on the error
        $w_j = w_j + \delta * dy * x_{i,j}$

        # update the bias
        $b = b + \delta * dy$
```

SLPs are quite flexible, however it has been shown that they only work well for linearly separable data. In [53] it was shown that SLPs cannot learn the XOR function.

6.1.6 Multi-Layer Perceptron

Multi-layer perceptron (MLP [36] is the natural extension of the SLP presented in the last section. As the name implies, with MLP there are multiple perceptrons stacked one after the other. Specifically, an MLP consists of at least three layers: an input layer, one or more hidden layers, and one output layer. Besides having more nodes (which gives MLP more representational power), the main difference between SLP and MLP is that the latter can handle also data which is non-linearly separable.

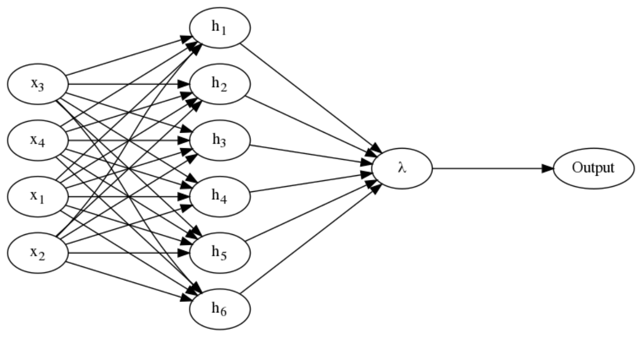


Figure 6.4: Example of a MLP with 4 input notes, one hidden layer of size 6, and one output node. Note that to save space the *biases* are not shown in the figure, but each node has one, and it works in the same way as for SLP.

The example above shows the structure of an MLP. As with SLP, each edge contains a weight the inputs are multiplied by. The inputs and their endpoint nodes in the hidden layer can be thought of as just multiple SLPs (each node in the hidden layer represents one SLP). The output of these SLPs is then used by the following layer as input to that layer's set of perceptrons (also called neurons). Each neuron has its own activation function: for the current work the *sigmoid* function is used as the activation of the output nodes, all other nodes in the hidden layers use *SELU* [47] which is just a function ensures that negative values grow *slower* than positive values.

Given enough training samples $(\vec{x_i}, y_i)$, where $\vec{x_i}$ is a feature vector received as input and y_i is the label for said feature vector, a MLP can learn the parameters θ for a function f such that $y_i \approx f(\vec{x_i}; \theta)$. This parameter θ is optimized so that the value of the function is as close as possible to the target y_i. It has been shown [24] that, in theory, a MLP with as few as one hidden layer [42], is able to represent any real function $f*$.

If the weights of a node can be thought of as a vector $\vec{w}$ then the weights of a whole layer can be thought of as a matrix W, and the biases for each layer can be represented as a vector $\vec{b}$. With this in mind, the evaluation of the MLP can be done by simply start evaluating from the input and continue until the output. For example, the network presented in the figure above would be evaluated is as follows.

$$Output = \lambda_o(\vec{b}_{h\to o} + W_{h\to o}^T \cdot \lambda_h(\vec{b}_{i\to h} W_{i\to h}^T \cdot \vec{x}))$$

Where $\vec{x}$ is the input vector. $W_{i\to h}$ and $\vec{b}_{i\to h}$ are the matrix of weights and the bias (respectively) that connect the input layer with the hidden layer. λ_h is the activation function performed for the hidden layer. $W_{h\to o}$ and $\vec{b}_{h\to o}$ are the matrix and bias of weights that connect the hidden layer with the output layer. Finally λ_o is the activation function performed at the output layer.

The above is what is called the *forward pass* when evaluating an MLP. If the parameters θ still need to be trained then also a backward pass needs to be performed, this ensures that the parameters are updated to learn a specific function.

θ is nothing more than the collection of weights and biases that the network uses when evaluating a certain input. They are automatically tuned employing an algorithm called backpropagation [61], whose main contribution is that it allows attributing the classification error to weights that are not directly connected with the output node. This is done by finding the error attributed to each node of the last layer of hidden networks. Then these errors are *propagated back* though each subsequent layer until they reach the input layer. Once the errors for each node are known, the nodes can update their weights through *gradient descent*, with just means the weights will be *increased* or *decreased* by a small amount in the direction which decreases the error.

In the current project, the used structure of MLP has 4 layers: an input layer, a first hidden layer of 128 neurons, a second hidden layer of 32 neurons, and finally the output layer, with just one output node. Instead of performing the process mentioned above to find the optimal θ manually, the optimizer *Adadelta* [74] is used. The implementation of MLP used in this project is provided by the Keras library [13].

6.2 Ensemble Models

All of the baseline classifiers presented in the previous section are able to capture specific facets of the data. For example, some may perform better for certain distribution of features and some for others. When using only models from the set of *baseline models* presented above, one by definition needs to choose one of the algorithms and ignores the predictions of the others. Besides this, the task of deciding which parameter is the *best* can be quite difficult, since no model may perform better in all cases.

Luckily there is the possibility to use more than one model at once, and join the predictions of these *base classifiers* in some way. This is called an *ensemble classifier* [76]. There are multiple kinds of ensemble classifiers [26, 56], with the difference being mainly on how they're constructed and how the predictions of the base classifiers are joined. However, in this work only *bagging* and *stacked* ensembles will be explored.

The ensembles presented in this section use as their base classifiers all the baseline models except for *Linear SVM* since this is incapable if predicting probabilities (it only predicts labels).

subsection 6.2.1 will explain the motivation for using an ensemble of models. Following that, subsection 6.2.2 and subsection 6.2.3 will explain the voting and stacked ensembles used in the project.

6.2.1 Motivation

As said in the introduction to this section, different base models may capture different aspects of a particular problem. This means that some models may work better than others in some situations. Using an ensemble allows us to mix all of the base model's predictions so that the final classification is as good as possible. It has been shown that joining the predictions of multiple classifiers trained to solve the same problem usually improves the result [40, 56, 64].

Another way this can be seen is that the base classifiers may have a high bias or high variance, and joining them (e.g. averaging the results) reduces said bias/variance [8]. As a side note, it is desirable for a machine learning algorithm to have a both low *variance* and a low *bias* [54]. However, obtaining both of these is very difficult. A model with high variance *variance* typically overfits the noise in the data, and a model with a high bias tends to underfit the data.

From the above also follows that a good ensemble is one where the base classifiers are as diverse as possible [76].

Ensemble models are commonly seen in the top places of machine learning competitions [3]. They have also been successfully employed in diverse areas [76] like computer vision, computer security, intrusion detection, in medical applications, among others.

6.2.2 Bagging Ensembles

The main idea of *bagging* [8, 76] is that if the collection of base models have all been trained on different data then they can be said to be independent among themselves. This means that their predictions can be averaged to consider different opinions about a problem [8]. This decreases the error, especially if the base models are unstable ones [76]. As an example, the *random forest* classifier presented in subsection 6.1.4, is an ensemble of *decision trees* using *bagging*. A decision tree on its own normally does not perform very well, but its performance is increased when joined with that of other trees.

The name *bagging* is an abbreviation for *bootstrapping* and *aggregating*. The *aggregation* refers to joining the predictions of different classifiers. This is usually done either by averaging their predictions (*soft voting*) or by choosing as final label that which was predicted the greatest amount of time (majority vote/*hard voting*).

Since it is seldom the case that enough training data is available to split it into disjoint sets, one for each classifier, the technique of *bootstrapping* needs to be used. This refers to the fact that each base classifier can be trained on a different set of data, which is created from the original training set $|T| = n$ by uniformly sampling, with replacement, n elements from T. In this way, a new set T_i is created for each base model.

By constructing new training sets in such a way, it is possible that a set has multiple entries of an item in the training set, or that it has no entries at all of that item. In [8] it is shown that the probability of sampling a given element $i \in T$ at least once, is $1 - \frac{1}{e} \approx 63.2\%$. This means that for each created T_i there are around 36.8% of unique examples that do not appear in it.

Suppose h_i represents the ith classifier, f is the real function that the ensemble should learn, $\vec{x}$ is the input vector, and that the problem being dealt with is a binary classification problem, where labels can be either $\{-1, 1\}$. Each h_i has its probability of being mistaken ϵ.

$$P(h_i(\vec{x}) \neq f(\vec{x})) = \epsilon$$

If there are a total of B base classifiers, then the final prediction of the ensemble H would be:

$$H(\vec{x}) = \mathtt{sign}\Big(\sum_{i=1}^{B} h_i(\vec{x})\Big) \tag{6.3}$$

This means that the complete ensemble H will only make a classification mistake when more than half of its base classifiers make one. Moreover, as shown in, [76], theoretically the maximum error of the ensemble decreases exponentially as the number of base classifiers approaches $+\infty$. Although this is impossible to do in practice, and [76] shows that the error in an ensemble tends to decrease with each extra base classifier added, until it a certain number is attained and the error then decreases asymptotically. This number is dependent on the problem.

The implementation of bagging ensembles used is provided by the Scikit-Learn [57] library. Specifically, two main models of this class are used, which mainly differ in how the predictions of base classifiers are joined. These are explained in the following subsections.

Hard Voting Classifier

A *hard voting classifier* is an ensemble model that joins the predictions of its base classifiers using a *majority vote* scheme. This type of ensemble does not consider how *sure* its base classifiers are of a given prediction, it only considers how many times each label was predicted. This might be helpful when the base classifiers are not very good at giving predictions (e.g. Naive Bayes [27, 57]).

For a binary classification problem, the output of a hard voting ensemble can be calculated with Equation 6.3, supposing that the individual classifiers h_i are predicting the class label and not the probability of the outcome being positive.

Soft Voting Classifier

A *soft voting classifier* is a bagging ensemble that joins the predictions of its base classifiers by means of averaging them. This method of aggregating the predictions considers how *sure* each base classifier is about said prediction.

Supposing that the output of each base classifier h_i is the probability of a positive outcome, the output of the *soft voting ensemble* can be calculated as:

$$H(\vec{x}) = \frac{1}{B}\sum_{i=1}^{B} h_i(\vec{x})$$

6.2.3 Stack Based Ensembles

The main idea behind stacked classifiers [10, 73], which is also named as *Super Learner* [66], is that of aggregating the results of the base classifiers by means of a *meta learner*. In the most basic case, a stacked ensemble is composed of one collection of base classifiers followed by a *meta learner* (which is just another classifier), that uses the prediction made by these base classifiers to make the final decision. This can, however, be extended, and an arbitrary number of *layers* of classifiers can be used.

The way in which a stacked ensemble is trained differs from the procedure mentioned for the baseline and bagging classifiers, especially since it needs to account for the fact that the classifiers are layered. When training, only the first layer directly see the training data. Further layers see only the predictions of the layers before. The training algorithm also makes use of the concept of k-fold to generate the training data for further layers (the process of using k-fold cross-validation will be explained later in section 7.1). Following is the explanation of how the training of a stacked ensemble with two layers, one input and a meta layers, proceeds.

Suppose the algorithm is provided with a training set T where $|T| = n$, and $T_i = (x, y)$. Also, suppose that the set of base learners is L. The first layer needs to process the original training set, and generate from it a new training set T' which will be fed to the meta layer. If the original training set T had n elements, then the training set which will be generated by the L base classifiers is a $n \times L$ matrix, which will be denoted as Z, and is produced by the following steps:

1. Split the original training date T using k-fold, with $k = \alpha$.

2. For each split, train each base classifier on the train part of the split and then predict the validation part. Aggregate all the predictions obtained. In such a way, each classifier produces a n sized vector of predictions.

3. Stack the n predictions obtained by each classifier, so that now the results are a matrix of shape $n \times L$.

This matrix Z will then be given as input data to the meta classifier together with the original n labels for the training data. In such a way, the first row of Z corresponds to the predictions the base classifier made for T_{1x} whose correct label is T_{1y}. Then the meta classifier can normally be trained on Z and said labels. Following that, each base classifier is trained on the complete training set T.

To generate a prediction for a new input vector, the predictions of the first layer are then obtained and stacked into a $n \times L$ matrix, which is then fed to the meta layer, which will proceed to give the final prediction.

The process described above is for a stacked ensemble with two layers, but it can easily be generalized to any number of layers by just repeating the steps described for the base classifier once for each extra layer.

In this project, 2-folds are used to generate the training data for the meta layer. The implementation used is provided by the *ML-Ensemble* [35] library.

Gated Classifier

The *gated classifier* is a two-layer stacked classifier with the first layer composed of all the baseline models, and using SLP as the meta classifier. The idea behind this was for the SLP to act like a *gate*, where the predicted probability coming from each base classifier is weighted by some value and the outcome is just the weighted sum of these predictions, passed through a sigmoid function (the activation function used for SLP).

It can be thought of like a mix between *stacked ensemble* and the *bagging ensembles* shown in the previous section. *Bagging* ensembles are similar to this *gate ensembles* but without weighting the prediction of their base classifiers.

Stacked Classifier

This classifier is composed of three layers. The first two layers are composed by the baseline classifiers, and it also uses SLP as a meta learner. The predicted probabilities from the first layer go into the second layer, which in turn sends its probabilities to the meta layer.

7 Results

This chapter presents the results obtained when evaluating the different models, both baseline and ensembles. The evaluation results for both the baseline and ensemble classifiers presented in this section have been obtained by performing a 5-fold cross-validation procedure on the training data (on overview of which is given in the next section). The data used to generate these results was the Wikidata training dumps and the external catalog dumps mentioned in section 5.1.

Before training and evaluating the baseline models, a grid search was performed to find the best hyperparameters for each of these. This is explained in section 7.2.

The metrics measured during evaluation are: *F1 score*, *Precision*, and *Recall.* These have been previously explained in subsection 2.2.5.

Besides presenting the results, section 7.5 will strive to compare the results obtained for the baseline and ensemble models and point out any differences.

7.1 Cross Validation

When evaluating the performance of a given machine learning algorithm, it is desirable to know what the performance of said algorithm will be on the real world data[1] the one for which the final values want to be known. However, it is impossible to do so, because this would require to have a labeled version of this real-world data.

Ideally, the training data should come from a distribution similar to the classification data, if not the performance of the algorithm will not be satisfactory when applied to the classification set. This assumption, that the training and classification datasets are **i.i.d.**[2] [20], allows us to take a piece of the training set and set it aside to act as an *approximation* of the classification set. This piece we set aside is called the *test set.*

The way this can be used is that we separate the whole training data D into a *test set* S and a *training set* T such that:

[1]also known as *classification* data or classification set
[2]Independent and identically distributed.

$$S \cup T = D$$
$$S \cap T = \varnothing$$

The learning algorithm is then trained only on T, and finally an estimate of the error the algorithm will get when predicting the real classification data can be obtained by measuring the error obtained when predicting V.

The conditions mentioned above must be respected for this to work. Especially $S \cap T = \varnothing$, since if any element $S_i \in T$ then the algorithm will have seen the actual label of the said element during training, and this may influence the estimate performance calculated on the test set since the classification set is unknown to the algorithm at prediction time. The algorithm may have learned the label for said item, and as such will affect the final error measured on the validation set by making it look better than it actually is.

Another consideration to keep in mind is that the actual validation set should not be used to tune the hyperparameters of the algorithm since this will only make the algorithm perform better in the validation data, but not necessarily better in the classification data. Once again, this may cause the performance of the algorithm to seem better than it actually is. To tune hyperparameters what can be done is separating the training data into a further third set, called the *validation set* V, which is also subject to $V \cap T \cap S = \varnothing$. With this, now the training can be done on T, hyperparameters can be tuned to minimize the error on V, and finally the estimate of the classification set error can be obtained by evaluating the algorithm on S.

In some cases, however, this is not desirable since it means the training data will be split three ways, and a big part of it will not be seen by the model at all. An example case maybe when there is only a small training set, or the algorithm being used needs a lot of data to be trained properly (e.g. when using a neural network).

Another possible problem may be that the error metrics obtained may be dependent on the specific separation of the dataset that was made to create T, V, and S. For example, the algorithm may appear to have a great performance on S, but if the training set were to be shuffled and split again then the performance would be much worse.

A better approach, which permits the use of most of the training data, and also provide an approximation of a *split-independent* error value is to use k-fold cross validation [12, 48]. In this process, we repeatedly (k times) split the training set D into T and V as mentioned above, train on the current split T, measure the error on V and then average all the error measures to get a final error. Using a $k > 1$ reduces variability in the final error metric since these get averaged over many possible ways of splitting the training data.

The algorithm for k-fold cross validation is as follows:

```
errors = [] # list that will hold the errors obtained during each fold
for i in 0..$k$:
    $T, V$ ← splitTrainingData($D$)

    $V_{features}, V_{labels} \leftarrow$ getFeaturesAndLabels($V$)
    $T_{features}, T_{labels} \leftarrow$ getFeaturesAndLabels($T$)

    model = initializeClassifier()
    model.fit($T_{features}, T_{labels}$) # train the classifier

    predictions = model.predict($V_{features}$)

    error = measureError(predictions, $V_{labels}$)
    errors.append(error)

finalError = average(errors)
```

Note that in this example *error* was mentioned as the metric that is being extracted. However, any metric which depends on V can be extracted in such a way. For this work, the metrics obtained during the cross-validation procedure are *F1-Score*, *Precision*, and *Recall*.

7.2 Baseline Hyperparameter Optimization

As seen in chapter 6, all classifiers have hyperparameters which can be tuned to change the behavior of each algorithm. Setting the correct value for these hyperparameters is essential to get a good performance from them.

Finding the best hyperparameter for a model usually involves a lot of manual tinkering, trying different combinations until the best set of hyperparameters is found. This procedure can actually be automated by using the *Grid Search* [18] functionality provided by Scikit-Learn [57]. In grid search, a set of possible values for each hyperparameter is provided, and the algorithm will go through each possible combination of these and tries them on the classifier.

The classifier is evaluated using each combination of hyperparameters, and finally, the combination which provided the best evaluation performance is returned. This is a lengthy process but since the evaluations are independent of each other they can be done in parallel [4].

For the current project, the evaluation of the hyperparameters was done by performing 5-fold cross-validation with the classifier initialized using the specified parameters. After the 5-fold cross-validation is complete, the average performance of these is returned. The *grid search* eventually returns the combination of hyperparameters which gave the best *average* performance while executing the *cross validation*.

Since the grid search procedure, plus the 5-fold cross-validation, is quite a lengthy procedure, it was chosen to perform this optimization by only optimizing the performance of the models on *Discogs/Musician*. The procedure searches for the combination of hyperparameters which makes the classifier give the best *F1* score.

The following tables show the hyperparameters tried for each baseline algorithm. In bold are the hyperparameter which was chosen as the best by the grid search procedure. A brief clarification will also be given about the parameters which were not explicitly mentioned in the algorithm's description.

Naive Bayes

alpha	binarize
1	0.9
0.1	0.8
0.01	0.7
0.001	0.6
0.0001	0.5
	0.4
	0.3
	0.2
	0.1

Logistic Regression

C	class_weight	max_iter	solver	tol
100	balanced	200	sag	1e-05
10	**None**	**100**	saga	0.0001
1.0			lbfgs	**0.001**
0.1			**liblinear**	
0.01				

max_iter is the maximum amount of iterations the optimization algorithm can perform when finding the optimal parameters for logistic regression. Said optimizer is defined by the hyperparameter **solver**. **class_weight** may be used to give *extra weight* to each class, using the *None* option means that all classes have a weight of 1.

Linear SVM

C	dual	max_iter	tol
100	False	2000	1e-05
10	**True**	**1000**	0.0001
1.0			**0.001**
0.1			
0.01			

Random Forest

bootstrap	criterion	max_features	n_estimators
False	**entropy**	**None**	**500**
True	gini	log2	350
		sqrt	200
			100

Specifying a **max_features** allows the random forest algorithm to train each decision tree only on a certain subset of the features, this is used as an extra source of randomness.

Single Layer Perceptron

activation	batch_size	epochs	optimizer
sigmoid	2048	3000	**Nadam**
	1024	2000	Adadelta
	512	**1000**	RMSprop
	256	100	adam

batch_size defines the among of items that the perceptron processes at once. The error at each evaluation is performed by considering the total error on the batch.

Multi-Layer Perceptron

batch_size	epochs	hidden_activation	hidden_layer_dims	optimizer	output_activation
512	2000	**selu**	[128, 32, 32]	Nadam	**sigmoid**
	1000	tanh	[256, 128, 32]	**Adadelta**	
		relu	**[128, 32]**	adam	

7.3 Baseline Evaluation

This section presents the results of performing a 5-fold cross-validation on the baseline models presented in section 6.1.

The table below shows the performance metrics for each classifier applied to each concrete entity. The bold values in the *mean* columns are the best value for a given concrete entity. Note that the names of the algorithms have been shortened to make the table fit nicely on the page. The abbreviations are as follow:

- Linear Support Vector Machine: lsvm
- Logistic Regression: lr
- Multi layer perceptron: mlp
- Naive Bayes: nb
- Random Forest: rf
- Single layer perceptron: slp

Catalog	Entity	Model	F1.Mean	F1.STD	Prec.Mean	Prec.STD	Rec.Mean	Rec.STD
discogs	band	lr	0.924791	0.001550	0.908952	0.001476	0.941198	0.003025
discogs	band	lsvm	0.925262	0.001267	0.896373	0.002110	**0.956090**	0.003388
discogs	band	mlp	**0.928931**	0.001310	**0.915249**	0.001875	0.943032	0.001727

discogs	band	nb	0.922386	0.001457	0.891462	0.001633	0.955542	0.003274
discogs	band	rf	0.921673	0.001084	0.909216	0.002648	0.934490	0.002988
discogs	band	slp	0.925647	0.001462	0.908264	0.001465	0.943716	0.003188
discogs	musician	lr	0.938138	0.001693	0.917022	0.002191	0.960250	0.001370
discogs	musician	lsvm	0.936944	0.001224	0.905424	0.002080	**0.970741**	0.000880
discogs	musician	mlp	**0.940715**	0.001443	**0.921507**	0.001770	0.960754	0.003867
discogs	musician	nb	0.931622	0.001314	0.903489	0.002155	0.961565	0.000812
discogs	musician	rf	0.933460	0.002427	0.917496	0.003021	0.949992	0.002296
discogs	musician	slp	0.938168	0.001664	0.917755	0.002291	0.959510	0.001258
imdb	actor	lr	0.893573	0.000579	0.878388	0.001838	0.909299	0.001406
imdb	actor	lsvm	0.894011	0.000789	0.875260	0.002088	0.913592	0.001998
imdb	actor	mlp	0.896922	0.000400	0.880657	0.003412	0.913832	0.004129
imdb	actor	nb	0.889563	0.001311	0.825956	0.002002	**0.963789**	0.001153
imdb	actor	rf	**0.899161**	0.000836	0.879621	0.001427	0.919592	0.001620
imdb	actor	slp	0.893205	0.000591	**0.881499**	0.001829	0.905232	0.001676
imdb	director	lr	0.913846	0.000694	0.892154	0.004468	0.936658	0.003834
imdb	director	lsvm	0.912727	0.001597	0.880551	0.005850	0.947397	0.003670
imdb	director	mlp	**0.919340**	0.000980	**0.899994**	0.002724	0.939548	0.002002
imdb	director	nb	0.907755	0.002541	0.855312	0.005063	**0.967075**	0.001997
imdb	director	rf	0.916903	0.000940	0.892806	0.003096	0.942355	0.002601
imdb	director	slp	0.914070	0.000545	0.895414	0.003822	0.933551	0.003516
imdb	musician	lr	0.917958	0.001717	0.908442	0.002911	0.927680	0.001280
imdb	musician	lsvm	0.917906	0.001399	0.909440	0.002345	0.926534	0.001067
imdb	musician	mlp	**0.925888**	0.001165	**0.942860**	0.001545	0.909520	0.001863
imdb	musician	nb	0.904327	0.013959	0.868388	0.055110	**0.948158**	0.033475
imdb	musician	rf	0.922135	0.000594	0.932216	0.000733	0.912272	0.001018
imdb	musician	slp	0.917503	0.001383	0.909845	0.002659	0.925296	0.000865
imdb	producer	lr	0.906045	0.002643	0.898064	0.005640	0.914192	0.000826
imdb	producer	lsvm	0.905975	0.003885	0.894173	0.006799	0.918114	0.001219
imdb	producer	mlp	**0.909398**	0.002913	0.894253	0.007249	0.925119	0.003188
imdb	producer	nb	0.898657	0.004254	0.842785	0.008230	**0.962525**	0.001450
imdb	producer	rf	0.906239	0.004720	0.895270	0.008984	0.917553	0.005214
imdb	producer	slp	0.905526	0.002380	**0.903146**	0.006456	0.907958	0.001993
imdb	writer	lr	0.928108	0.002082	0.924621	0.001792	0.931635	0.004264
imdb	writer	lsvm	0.928606	0.001983	0.919005	0.002274	0.938421	0.003574
imdb	writer	mlp	**0.930302**	0.002571	0.924571	0.003714	0.936149	0.006365
imdb	writer	nb	0.922121	0.002580	0.882778	0.004051	**0.965145**	0.002172
imdb	writer	rf	0.929301	0.002919	0.922216	0.004160	0.936508	0.003393
imdb	writer	slp	0.927724	0.001915	**0.924983**	0.002891	0.930499	0.003970
musicbrainz	band	lr	0.918022	0.002327	0.937643	0.002251	0.899217	0.003973
musicbrainz	band	lsvm	0.913877	0.001689	**0.961199**	0.003976	0.871046	0.006050
musicbrainz	band	mlp	**0.921134**	0.002588	0.926546	0.004402	**0.915818**	0.004989
musicbrainz	band	nb	0.907094	0.001994	0.953012	0.001335	0.865401	0.002974

musicbrainz	band	rf	0.918844	0.002231	0.938321	0.005419	0.900192	0.003714
musicbrainz	band	slp	0.915977	0.002302	0.946439	0.002538	0.887441	0.005184
musicbrainz	musician	lr	0.953083	0.001301	0.943755	0.002042	0.962600	0.001165
musicbrainz	musician	lsvm	0.954410	0.001406	0.942882	0.002112	0.966225	0.001090
musicbrainz	musician	mlp	**0.957375**	0.001629	0.944146	0.004491	**0.971004**	0.002647
musicbrainz	musician	nb	0.952783	0.001847	**0.953561**	0.002394	0.952008	0.001847
musicbrainz	musician	rf	0.953673	0.001146	0.943972	0.001591	0.963577	0.001052
musicbrainz	musician	slp	0.953054	0.001383	0.943662	0.002094	0.962638	0.001341

The columns name **.Mean* are the mean value of the relative metric obtained during the cross-validation procedure. Similarly, the columns named **.STD* are the standard deviation of said metrics.

It's interesting to note that the best *F1.Mean* for all models seems to have been obtained on the entity *Musicbrainz/Musician*, and the second-best for *Discogs/Musician*. For most of the models, the third-best *F1.Mean* is obtained on *IMDb/Writer* (only Naive Bayes has *Discogs/Band* as the third best, although the values are very similar with *Writer*).

When it comes to *precision* things are less *unanimous*. Most models still have *Musicbrainz/Musician* as their best precision, except for *lsvm*, and *slp*, who both have *Musicbrainz/Band*, although for this the second best is also *Musicbrainz/Musician*. The place of second best precision among the first set of classifiers is *Musicbrainz/Band*, except for *mlp*, who has *IMDb/Musician* in the second place.

For *Recall*, all models except for *lsvm* and *Naive Bayes* have *Musicbrainz/Musician* as their top concrete entity. *lsvm* and *naive bayes* have *Discogs/Musician* and *IMDb/Director* respectively. All models except *lsvm* and *Naive Bayes* have *Discogs/Musician* as their second best. *lsvm* and *naive bayes* have *Musicbrainz/Musician* and *IMDb/Writer* respectively.

The results presented above are quite interesting since they show that features extracted for *Musicbrainz/Musicians* are more informative than those for other concrete entities. Another reason for this may be, as seen in section 5.4, that the *Musicbrainz* catalog has eight possible features while all other catalogs have seven.

The table below shows which were the average performance of each model on each catalog in general. The rows have been ordered by *F1 Score*. It can be noted that the average *STD* is quite low compared to the score values.

Catalog	Model	Avg.F1	Avg.F1.STD	Avg.Prec	Avg.Prec.STD	Avg.Rec	Avg.Rec.STD
musicbrainz	mlp	0.939254	0.002109	0.935346	0.004446	0.943411	0.003818
musicbrainz	rf	0.936259	0.001688	0.941146	0.003505	0.931885	0.002383
musicbrainz	lr	0.935553	0.001814	0.940699	0.002146	0.930909	0.002569
musicbrainz	slp	0.934516	0.001842	0.945051	0.002316	0.925040	0.003263
musicbrainz	lsvm	0.934144	0.001548	0.952041	0.003044	0.918635	0.003570
musicbrainz	nb	0.929939	0.001920	0.953287	0.001865	0.908705	0.002410
imdb	mlp	0.916370	0.001606	0.908467	0.003729	0.924834	0.003509
imdb	rf	0.914748	0.002002	0.904426	0.003680	0.925656	0.002769
imdb	lr	0.911906	0.001543	0.900334	0.003330	0.923893	0.002322
imdb	lsvm	0.911845	0.001931	0.895686	0.003871	0.928812	0.002306
imdb	slp	0.911606	0.001363	0.902977	0.003531	0.920507	0.002404

imdb	nb	0.904485	0.004929	0.855044	0.014891	0.961338	0.008049
discogs	mlp	0.934823	0.001377	0.918378	0.001822	0.951893	0.002797
discogs	slp	0.931907	0.001563	0.913009	0.001878	0.951613	0.002223
discogs	lr	0.931464	0.001622	0.912987	0.001833	0.950724	0.002197
discogs	lsvm	0.931103	0.001246	0.900899	0.002095	0.963415	0.002134
discogs	rf	0.927566	0.001755	0.913356	0.002835	0.942241	0.002642
discogs	nb	0.927004	0.001386	0.897476	0.001894	0.958554	0.002043

	Best Avg.F1	Best Avg.Prec	Best Avg.Rec
Musicbrainz	mlp	nb	mlp
IMDb	mlp	mlp	nb
Discogs	mlp	mlp	lsvm

Table 7.3: Best performing baseline models for each catalog according to the three main metrics.

It seems that overrall *mlp* is the best performing baseline model. It is interesting to note that although *naive bayes* appears to have the highest *precision* for *Musicbrainz* is is also the model that has the lowest *recall* for that same catalog. The same happens for IMDb, *naive bayes* has the best *recall* but the lowest precision. A slight exception is Discogs, where *lsvm* has the best *recall*, and is the second worst in *precision*.

Finally, the table below contains a summary of the metrics obtained by each classifier, these were obtained by averaging the metrics obtained for each concrete entity. The best value for each metric is denoted in **bold**.

Model	Avg.F1	Avg.F1.STD	Avg.Prec	Avg.Prec.STD	Avg.Recall	Avg.Recall.STD
mlp	**0.925556**	0.001667	**0.916643**	0.003465	0.934975	0.003420
rf	0.922377	0.001877	0.914570	0.003453	0.930726	0.002655
lr	0.921507	0.001621	0.912116	0.002734	0.931414	0.002349
slp	0.921208	0.001514	0.914556	0.002894	0.928427	0.002555
lsvm	0.921080	0.001693	0.909367	0.003293	0.934240	0.002548
nb	0.915145	0.003473	0.886305	0.009108	**0.949023**	0.005462

Table 7.4: Average metrics obtained for the baseline classifiers.

It can be seen in Table 7.4 that *mlp* is the model with the best *F1 score* (as expected from the results presented in Table 7.3). It also seems that *mlp* has the best average *precision* and is second in best *recall*, with the first place being taken by *naive bayes*.

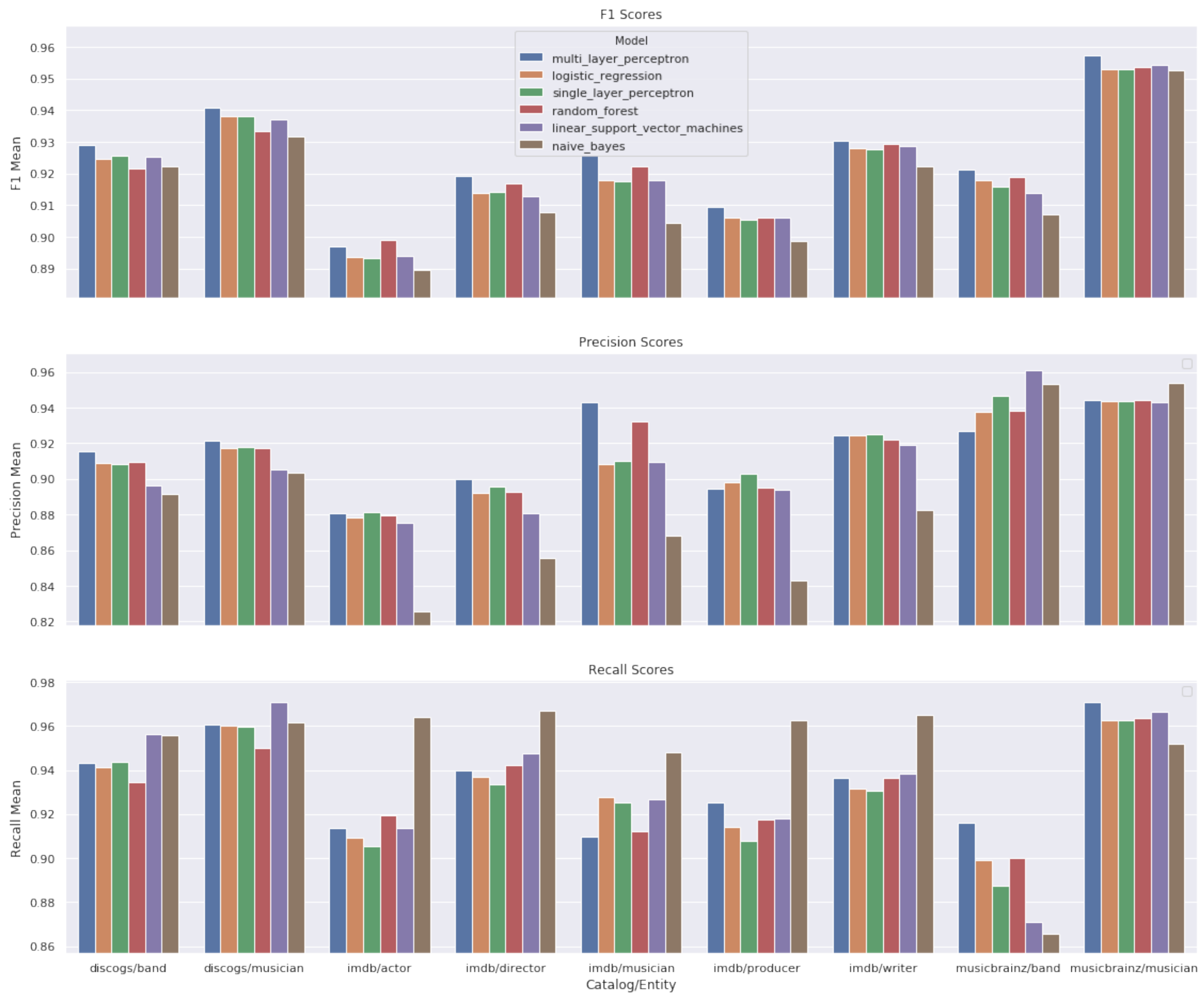


Figure 7.1: Barplots of the evaluation metrics for the baseline models. Separated by concrete entity.

A graphical representation of the table above can be seen in Figure 7.1. Here it can be better appreciated which concrete entities are easier to classify for which algorithms. For instance, it seems that all models have trouble with *IMDb/Actor*, in all metrics. It can also be seen that *naive bayes* tends to have a high recall in all cases, and poor *precision* in most. Another interesting thing to note is that all models have a much lower recall for *Musicbrainz/Band*, even if the precision they attain for this same entity is quite good when compared against the others. As expected, *Musicbrainz/Musician* is the overall best classified concrete entity.

It can also be appreciated that while *mlp* almost always has the best performance, some other models may have similar or event better scores for some metrics. For instance, all models have a higher precision for *Musicbrainz/Band* than *mlp*, and most have a higher recall for *IMDb/Musician*. This suggests that using only the *opinion* of *mlp* would not be the optimal strategy to get the best performance. A better approach would be to mix these predictions through an ensemble method.

7.4 Ensemble Evaluation

This section presents the results obtained during the 5-fold cross-validation procedure when applied to the ensemble models presented in section 6.2. The results are presented once for each concrete entity and model and the mean values in bold show the best value obtained for said concrete entity. Note that the names of the ensemble classifiers have been shortened to make the table fit properly. Below is the mapping from the full names of the classifiers to their shortened version/

- Gated Classifier : gc
- Stacked Classifier : sc
- Hard Voting Classifier : vh
- Soft Voting Classifier : vs

Catalog	Entity	Model	F1.Mean	F1.STD	Prec.Mean	Prec.STD	Rec.Mean	Rec.STD
discogs	band	gc	0.927455	0.001522	**0.912845**	0.002324	0.942566	0.004981
discogs	band	sc	**0.927771**	0.001080	0.911809	0.006866	0.944456	0.009776
discogs	band	vh	0.926795	0.001145	0.905189	0.001827	0.949465	0.002661
discogs	band	vs	0.927576	0.001300	0.904116	0.002522	**0.952312**	0.004592
discogs	musician	gc	0.936638	0.002652	**0.922472**	0.004920	0.951308	0.006874
discogs	musician	sc	0.938836	0.002473	0.917001	0.002568	0.961746	0.003667
discogs	musician	vh	**0.939658**	0.001932	0.913881	0.002899	0.966938	0.002004
discogs	musician	vs	0.938896	0.001494	0.909274	0.004537	**0.970543**	0.003097
imdb	actor	gc	0.897257	0.002191	0.895105	0.003315	0.899486	0.007650
imdb	actor	sc	0.890012	0.007978	**0.907933**	0.016179	0.873941	0.029767
imdb	actor	vh	0.897184	0.000752	0.882077	0.009962	**0.913098**	0.012148
imdb	actor	vs	**0.899699**	0.001028	0.887665	0.005118	0.912139	0.006529
imdb	director	gc	0.918505	0.001411	0.900678	0.002997	0.937067	0.003067
imdb	director	sc	0.917923	0.001777	**0.901563**	0.004736	0.934969	0.007596
imdb	director	vh	0.918177	0.001400	0.890278	0.004667	**0.947915**	0.003379
imdb	director	vs	**0.919262**	0.001497	0.894902	0.008123	0.945108	0.006606
imdb	musician	gc	0.924625	0.002018	0.943199	0.010644	0.906929	0.007107
imdb	musician	sc	0.924930	0.001915	**0.947724**	0.003561	0.903238	0.004935
imdb	musician	vh	0.921577	0.002711	0.918733	0.022289	**0.925297**	0.017423
imdb	musician	vs	**0.926019**	0.001244	0.939099	0.007632	0.913418	0.007684
imdb	producer	gc	**0.910555**	0.003934	0.902453	0.007668	**0.918955**	0.010547
imdb	producer	sc	0.904998	0.008882	0.899323	0.020682	0.912508	0.035943
imdb	producer	vh	0.907517	0.004611	0.901155	0.004867	0.914052	0.009619
imdb	producer	vs	0.909610	0.003182	**0.905090**	0.005243	0.914192	0.002709
imdb	writer	gc	**0.931719**	0.002789	0.928561	0.007822	0.934984	0.005804
imdb	writer	sc	0.931043	0.003075	0.929588	0.007946	0.932652	0.009692
imdb	writer	vh	0.931065	0.002249	0.920422	0.007199	**0.942038**	0.005549
imdb	writer	vs	0.931617	0.002161	**0.929887**	0.004030	0.933369	0.002867
musicbrainz	band	gc	0.918238	0.003816	0.948221	0.012288	0.890520	0.016863
musicbrainz	band	sc	**0.919791**	0.003254	0.941948	0.011043	**0.898935**	0.013037
musicbrainz	band	vh	0.917456	0.002139	0.953082	0.007505	0.884490	0.006698
musicbrainz	band	vs	0.916297	0.002493	**0.954029**	0.002458	0.881462	0.005454

musicbrainz	musician	gc	0.955526	0.001926	0.951364	0.006050	0.959821	0.007915
musicbrainz	musician	sc	0.955643	0.001373	**0.955374**	0.003750	0.955933	0.003327
musicbrainz	musician	vh	**0.956088**	0.001293	0.946047	0.003334	**0.966356**	0.001537
musicbrainz	musician	vs	0.954936	0.001831	0.955208	0.002438	0.954666	0.001676

Interestingly it can be observed that all classifiers have the same concrete entity for their top 5 F1 scores. In order from best to worst, these are: • Musicbrainz/Musician • Discogs/Musician • IMDb/Writer • Discogs/Band • IMDb/Musician. In all cases it seems that *IMDb/Actor* is the concrete entity with the worst F1 score.

For precision, all entities except for *hard voting* have *Musicbrainz/Musician* as their top value. *Hard voting* instead has *Musicbrainz/Band.* As for the worst precision, this seems to be, in all cases except for *stacked classifier*, to be for *IMDb/Actor. Stacked classifier* has *IMDb/Producer* as its worst precision.

All classifiers except for *gate classifier*, have as their best average recall *Discogs/Musician* followed by *Musicbrainz/Musician. Gate classifier* also has these two but in reversed positions. The worst recall seems to be assigned to *Musicbrainz/Band* in all cases except for *stacked classifier* which has *IMDb/Actor* as its worst.

Figure 7.2 shows a graphical representation of the table above.

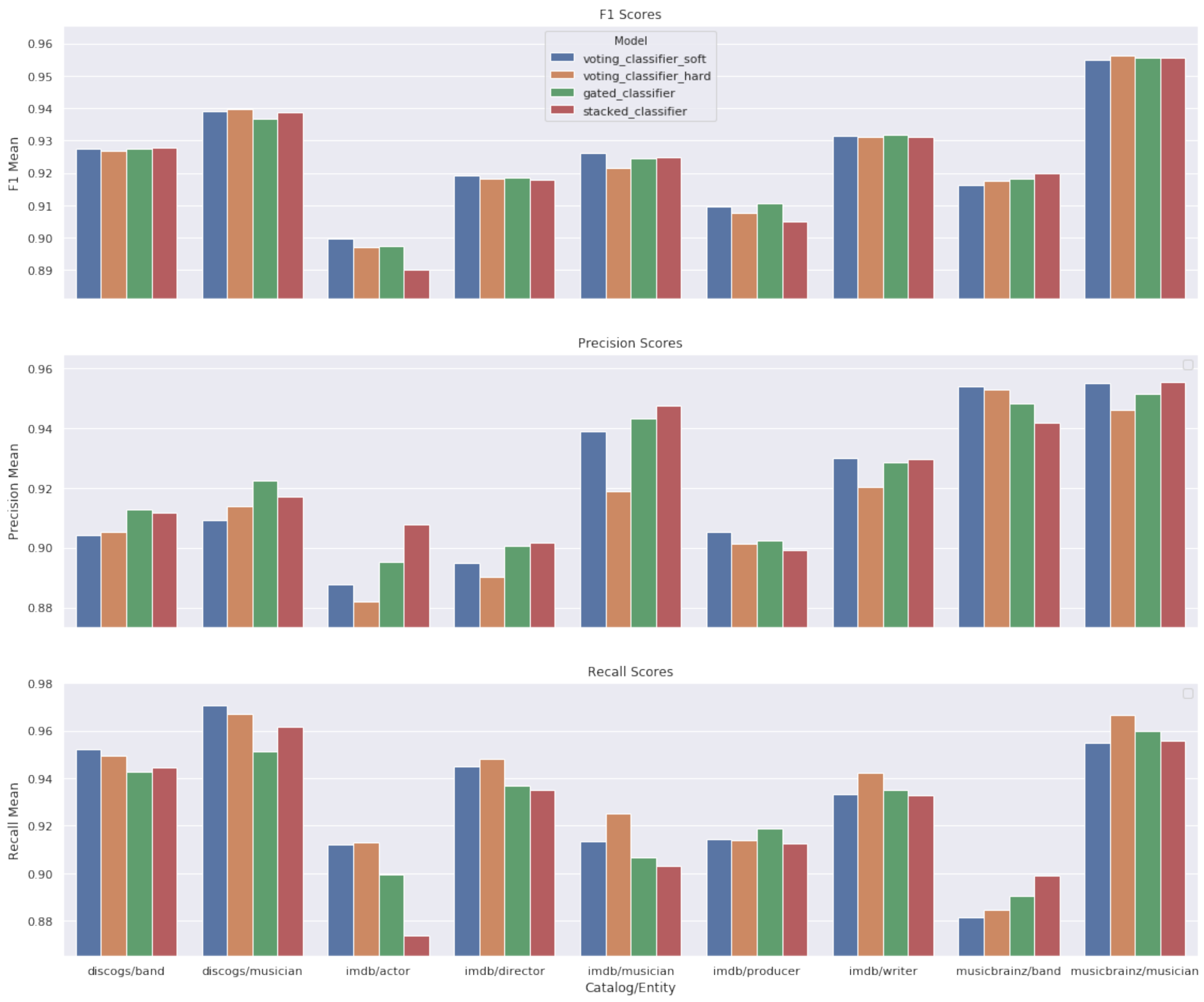


Figure 7.2: Barplots of the evaluation metrics for ensemble models. Separated by concrete entity.

In the figure above it can be better appreciated which classifiers perform best for which metrics. It is also clear which concrete entities are easier for the models to predict. As for the baseline models, it still appears that *IMDb/Actor* is the hardest one to classify. Stack classifier seems to yield an acceptable precision for this concrete entity, however, its recall is quite low. As with the baseline models, it can also be noted that *Musicbrainz/Band* has high precision and low recall.

The F1 scores are all similar when talking about the same concrete entity. The ones that show some more variation are *IMDb/Actor* and *IMDb/Producer*.

As for the recall, it can be seen that in almost all cases the best models are one of the *voting* classifiers. The only exception is for *IMDb/Producer*, where *Gated classifier* gives the best recall.

The table below has the average metrics that each classifier obtained for each catalog in general. These have been sorted by decreasing value of F1 score.

Catalog	Model	Avg.F1	Avg.F1.STD	Avg.Prec	Avg.Prec.STD	Avg.Rec	Avg.Rec.STD
musicbrainz	sc	0.937717	0.002314	0.948661	0.007397	0.927434	0.008182
musicbrainz	gc	0.936882	0.002871	0.949793	0.009169	0.925171	0.012389
musicbrainz	vh	0.936772	0.001716	0.949564	0.005420	0.925423	0.004117
musicbrainz	vs	0.935617	0.002162	0.954619	0.002448	0.918064	0.003565

imdb	vs	0.917241	0.001822	0.911329	0.006029	0.923645	0.005279
imdb	gc	0.916532	0.002469	0.913999	0.006489	0.919484	0.006835
imdb	vh	0.915104	0.002345	0.902533	0.009797	0.928480	0.009624
imdb	sc	0.913781	0.004725	0.917226	0.010621	0.911462	0.017587
discogs	sc	0.933304	0.001776	0.914405	0.004717	0.953101	0.006722
discogs	vs	0.933236	0.001397	0.906695	0.003530	0.961428	0.003844
discogs	vh	0.933226	0.001538	0.909535	0.002363	0.958201	0.002333
discogs	gc	0.932046	0.002087	0.917658	0.003622	0.946937	0.005928

	Best Avg.F1	Best Avg.Prec	Best Avg.Rec
Musicbrainz	sc	vs	sc
IMDb	vs	sc	vh
Discogs	sc	gc	vs

Table 7.7: Best performing ensemble models for each catalog according to the three main metrics.

There is no clear distinction about which is the best ensemble model. Both *soft voting* and *stack* classifier seem to have quite good performance. Stack classifier performs better for catalogs which have more information (Discogs and Musicbrainz), while soft voting performs best for IMDb that has less.

Voting classifier appears to have the best precision for *Musicbrainz*, however, this is also the model which has the worst *recall* for that catalog. On the other hand, *stack classifier*, which is the best F1 and recall on this catalog, is also the one with the worst precision.

For IMDb the best F1 score is given to *soft voting*, which interestingly it is not the best at precision nor recall, those places are given to *stack* and *hard voting* classifiers respectively. However, *stack classifier* is the one with the worst recall, and *hard voting* is the one with the lowest precision. For this catalog, it can also be seen that the *stack* classifier, which performs well for *Musicbrainz* and *Discogs* is actually the one with the worst F1 score.

The best F1 score for Discogs is given to the *stack classifier*. Like in IMDb, this best F1 model does not have the best precision nor recall. Best precision is given by the *gated classifier*, which is also the model with the worst recall and F1 score. *Soft voting* classifier is the one that gives the best recall, and it also has the worst precision.

The table below shows a summary of the scores obtained by each model. These have been obtained by averaging all the model's scores. The rows are ordered by *Avg. F1* in decreasing order. The best value for each metric is denoted in **bold**.

Model	Avg. F1	Avg. F1.STD	Average Prec	Avg. Prec.STD	Avg. Recall	Avg. Recall.STD
vs	**0.924879**	0.001803	0.919919	0.004678	**0.930801**	0.004579
gc	0.924502	0.002473	0.922766	0.006448	0.926848	0.007868
vh	0.923946	0.002026	0.914540	0.007172	0.934405	0.006780
sc	0.923439	0.003534	**0.923585**	0.008592	0.924264	0.013082

Table 7.8: Average metric score for each ensemble classifier.

From the results above it can be seen that both *stack based* models have the best average precision.

These are: *gated classifier*, which has the best precision, and *stack classifier*, which has the second-best. On the other hand, *voting based* classifiers have the best recall: *hard voting*, which has the best recall, and *soft voting* which has the second-best.

Soft voting also has the best precision when compared with *hard voting.* Probably because soft voting considers an average of its base models' predictions, so if a model is *very* certain about a match then this amount will be taken into consideration for the final prediction. On the other hand, *hard voting* only considers how many base models predicted a positive or negative, and not how certain they were about this prediction.

An important thing to note in the table above is that *soft voting* is also the classifier with the least standard deviation in all its metrics measurements. This means that the metrics obtained during the 5-fold cross-validation did not change much across different folds, meaning that the final performance of the model is not that dependent upon the distribution of the data it was trained with. This is important since it gives more certainty that real-world performance will be similar to that evaluated[3].

7.5 Comparative Evaluation

The results showed in the previous two sections for baseline and ensemble models seem quite good. However, it still needs to be decided which technique has the overall best performance, and, if possible, also define a classifier that can be said is the best among the pool of available classifiers. This section strives to answer these questions.

The table below shows the best three models for each metric, each cell (which contains the model and their respective score for that metric) has been sorted in descending order.

[3]If the classification data distribution is similar to that of the training set.

Catalog	Entity	Model	F1.Mean	Model	Prec.Mean	Model	Recall.Mean
discogs	band	mlp^+	0.928931	mlp^+	0.915249	$lsvm^+$	0.956090
		sc^*	0.927771	gc^*	0.912845	nb^+	0.955542
		vs^*	0.927576	sc^*	0.911809	vs^*	0.952312
	musician	mlp^+	0.940715	gc^*	0.922472	$lsvm^+$	0.970741
		vh^*	0.939658	mlp^+	0.921507	vs^*	0.970543
		vs^*	0.938896	slp^+	0.917755	vh^*	0.966938
imdb	actor	vs^*	0.899699	sc^*	0.907933	nb^+	0.963789
		rf^+	0.899161	gc^*	0.895105	rf^+	0.919592
		gc^*	0.897257	vs^*	0.887665	mlp^+	0.913832
	director	mlp^+	0.919340	sc^*	0.901563	nb^+	0.967075
		vs^*	0.919262	gc^*	0.900678	vh^*	0.947915
		gc^*	0.918505	mlp^+	0.899994	$lsvm^+$	0.947397
	musician	vs^*	0.926019	sc^*	0.947724	nb^+	0.948158
		mlp^+	0.925888	gc^*	0.943199	lr^+	0.927680
		sc^*	0.924930	mlp^+	0.942860	$lsvm^+$	0.926534
	producer	gc^*	0.910555	vs^*	0.905090	nb^+	0.962525
		vs^*	0.909610	slp^+	0.903146	mlp^+	0.925119
		mlp^+	0.909398	gc^*	0.902453	gc^*	0.918955
	writer	gc^*	0.931719	vs^*	0.929887	nb^+	0.965145
		vs^*	0.931617	sc^*	0.929588	vh^*	0.942038
		vh^*	0.931065	gc^*	0.928561	$lsvm^+$	0.938421
musicbrainz	band	mlp^+	0.921134	$lsvm^+$	0.961199	mlp^+	0.915818
		sc^*	0.919791	vs^*	0.954029	rf^+	0.900192
		rf^+	0.918844	vh^*	0.953082	lr^+	0.899217
	musician	mlp^+	0.957375	sc^*	0.955374	mlp^+	0.971004
		vh^*	0.956088	vs^*	0.955208	vh^*	0.966356
		sc^*	0.955643	nb^+	0.953561	$lsvm^+$	0.966225

Table 7.9: Best three models and their scores for each respective metric. These results are shown for each concrete entity. To better differentiate the provenance of the each classifier, their names have been super-scripted with a $^+$ if they are models from the baseline, while those symbolizing ensemble models were super-scripted with *

It can be noticed that in almost all cases, the list of *top 3* models for each metrics includes both classifiers coming from the set of baseline classifiers as well as ensemble classifiers. The exceptions for this are *Precision* for *IMDb/Actor*, where the list of top-3 is composed only of ensemble models. The *Recall* for the same model, and *Musicbrainz/Band* is instead composed only of baseline models. The list for *Recall*, for *IMDb/Musician* is also composed of only baseline classifiers. All classifiers in the lists for *F1 score* and *precision* of *IMDb/Writer* are exclusively from the set of ensemble classifiers.

The pie plots below show the number of times each model appeared in the first, second, or third positions.

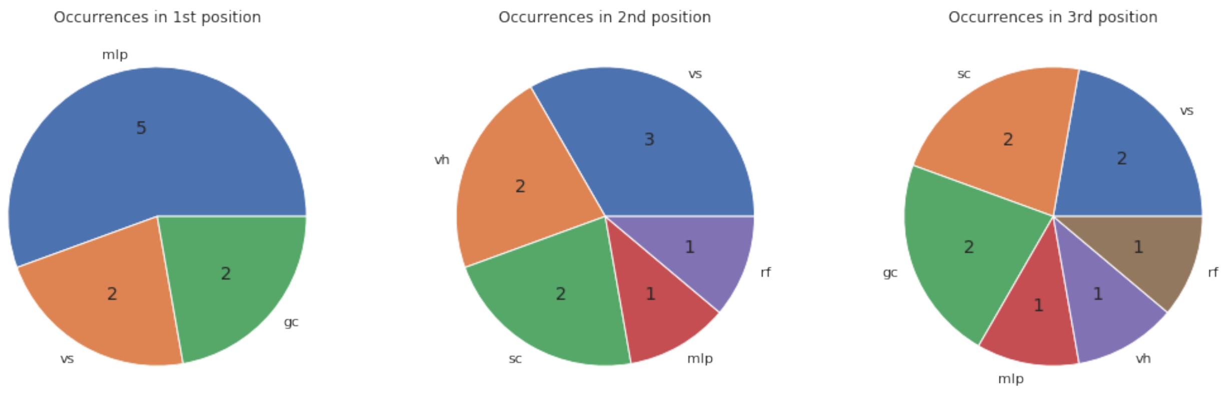


Figure 7.3: Pie plots show the occurrences of each model in each position of the *top 3*, for the F1 metric.

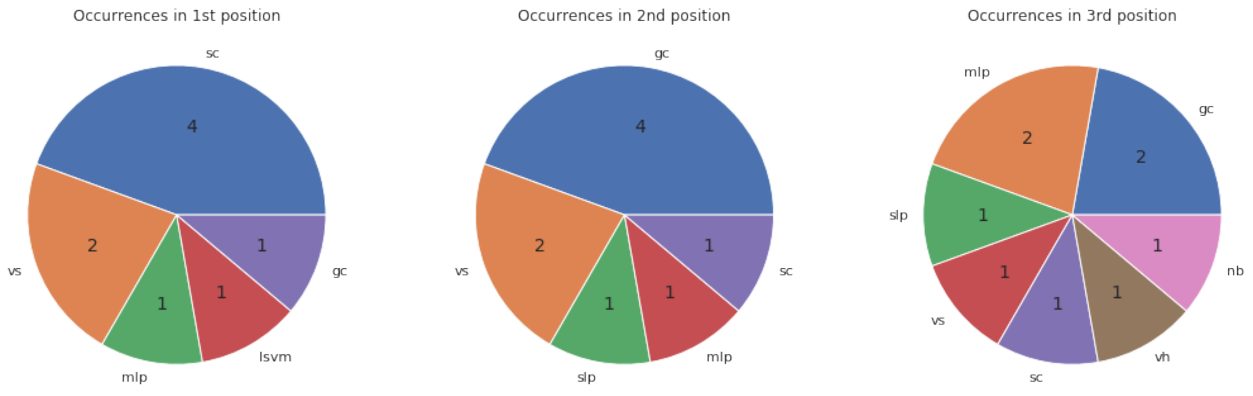


Figure 7.4: Pie plots show the occurrences of each model in each position of the *top 3*, for the *Precision* metric.

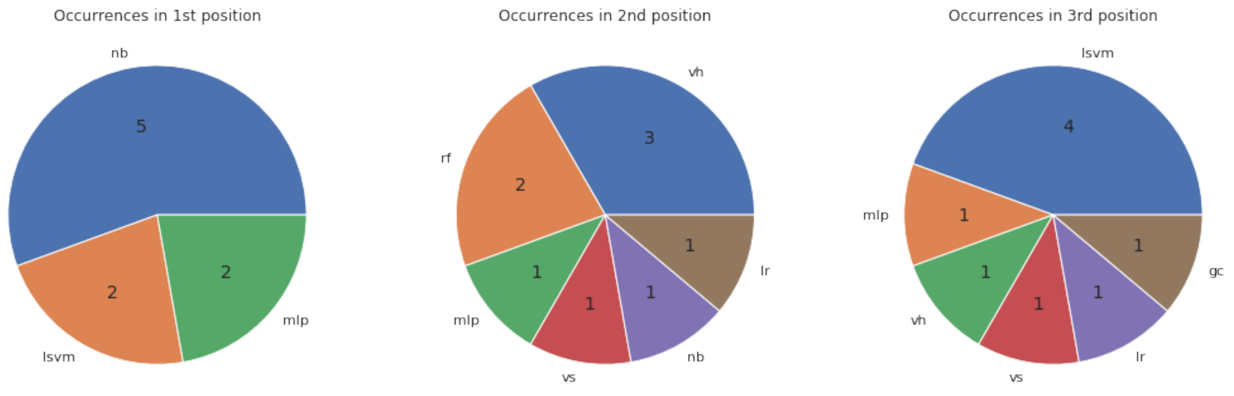


Figure 7.5: Pie plots show the occurrences of each model in each position of the *top 3*, for the *Recall* metric.

It can be seen in Figure 7.3 that *mlp* appears as the model with the highest score more than half the time. Followed by *soft voting* and *gate classifier*, which both appear in two of the cases. Soft voting is also the model that appears more time in as the second-best F1. Both *mlp* and *vs* appear among the top-3 for F1 in 7 out of the nine cases.

Figure 7.4 shows the occurrences for the precision metric. Here it can be see that ensemble based classifiers appear a total of 7 times out of 9 as the top-1 model: *stack classifier* appears 4 times, *soft* *voting* appears 2, and *gated classifier* appears once. The baseline models which appear as top-1 once each are *mlp* and *lsvm*. Something similar happens for the model which appears in the second place: 7 times out of 9 there is an ensemble model (4 times *gc*, 2 times *vs*, and once *sc*), while the remaining two spots are taken by *mlp* and *slp*.

For the recall case, in Figure 7.5, it can be seen that there are no cases where an ensemble classifier appears as the one with the best score. It seems that the the best recall is given to *nb*, which appears as top-1 5 out of 9 times, and *lsvm* and *mlp* which appear twice each. It can be seen that in the second-best recall spot there are ensemble models. 4 out of the 9 cases for this second spot are taken by ensemble classifiers (3 times *vh*, and once *vs*).

Note that even though *naive bayes* appears in the first and second places for recall, it does not appear as one of the top 3 for any of the other metrics, except for the one time it appears in the third position for precision. It is also interesting to notice that *stack classifier*, which is the one which most commonly appears as top precision, does not appear even once among the top 3 recalls.

From the above, it can be seen that ensemble models tend to have higher precision than baseline models. However, they also have a lower recall. A pattern also seems to be emerging in regards to which models appear as the best. For instance, it can be said that both *mlp* and *vs* have good performance in most of the cases. Also, it seems that ensemble models perform better, more commonly than baseline classifiers. The exception is *mlp*, which appears to perform best in some cases.

The table below shows the average metrics obtained for the top 5 models, ordered in descending order by F1 score.

Model	Avg.F1	Avg.F1.STD	Avg.Prec	Avg.Prec.STD	Avg.Recall	Avg.Recall.STD
mlp	0.925556 (1)	0.001667	0.916643 (4)	0.003465	0.934975 (1)	0.003420
vs	0.924879 (2)	0.001803	0.919919 (3)	0.004678	0.930801 (3)	0.004579
gc	0.924502 (3)	0.002473	0.922766 (2)	0.006448	0.926848 (4)	0.007868
vh	0.923946 (4)	0.002026	0.914540 (5)	0.007172	0.934405 (2)	0.006780
sc	0.923439 (5)	0.003534	0.923585 (1)	0.008592	0.924264 (5)	0.013082

Table 7.10: Average metrics for top-5 models ordered by F1 score. The parenthesis to the right of the average scores show the respective position to respect to the other values of that same score.

In the table above it can be seen that *mlp* is the best performing classifier when it comes to *average F1*, and is also the model with the best *average recall*. However, *mlp* is also the model with the second-worst precision (*vh* is the model with the worst precision).

Another interesting thing to notice is that in this table, *mlp* is the only baseline model in the top-5. Notice also the standard deviations, *ensembles* seem to have higher values for all metrics than *mlp*.

Now the question arises of whether these averages actually tell the behaviour of the algorithm, or whether some of the algorithms had a very good performance for one of the catalog entities. This behaviour can be better seen in the next figure.

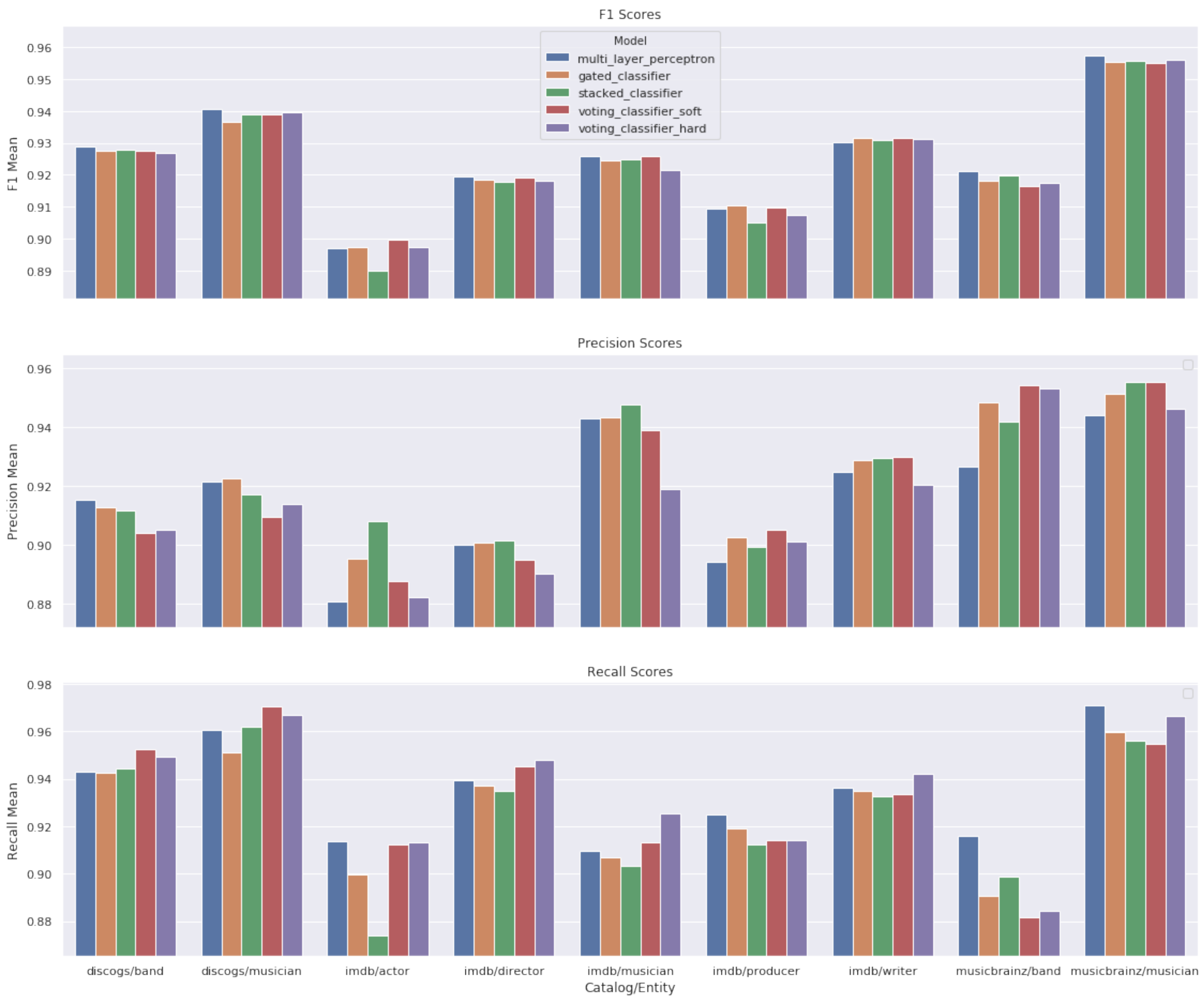


Figure 7.6: Barplot showing the metrics obtained by all top-5 classifiers.

In the figure above it can be appreciated that the F1 scores are almost always similar, except for the *Musicbrainz* entities, where *mlp* performs better. For *IMDb/Actor*, which has been shown to be problematic for all models to classify in the previous two sections, *mlp* does not perform very well, instead the best scores are obtained for *vs* and *gc*.

For the precision values, ensemble models seem to be more consistent, especially *stack classifier*. The precision of *mlp* is high for some concrete entities (e.g. *Discogs/Band*), but it is actually lower than for the other models for most other entities (e.g. both *Musicbrainz* entities). This result was, however, already expected from what was shown in Table 7.9.

For recall, all models seem to perform quite similarly. The exceptions are for entities of the *Musicbrainz* catalog, where *mlp* has much higher recall than the ensemble models, which may also explain why it has a higher F1 for this same entities even if its precision is not that good.

It is interesting that the performance of *mlp* is better for recall and F1, considering that the ensembles have a *mlp* as one of their base models. Some reasons for this may be that there are some classifiers among the base models which are producing more noise than actually useful prediction.

For example, one of the least performing baseline models, specifically those with low recall, like *slp* may be driving the ensemble to make some predictions which cause it to miss-classify some *positive* examples, and as such decreasing the overall recall. A more careful selection of the base models may help to solve these issues by only including models that are helpful for the ensemble.

8 Conclusion

Ensembles have better precision but tend to have a lower recall. Because of this their F1 suffers, and as such they do not manage to get the top score for this metric. However, if the situation we're currently in values precision above recall, ensemble models are the best choice.

As seen in the results comparison section, the *multi-layer perceptron* performs well in most cases, and the aggregated complexity of training an ensemble of models may not be worth it even if they do provide a higher precision. Obviously, the decision of whether this is worth or not must be taken on a per problem basis. Besides this, 4 out of the 5 models with the best F1 score, presented in the last section, are ensemble models, showing that they're still better than most baseline models.

Even though the performance of the ensembles in terms of *F1* and *recall*, was not the best, they did show a very good precision. Meaning that the mixing of opinions from its base classifiers actually supplies valuable information to the prediction process, and makes it more sensitive to *negative* samples. Their application may make more sense if the problem currently being worked on does not tolerate *false positives.*

Another thing that should be kept in mind is that the results presented for the ensemble models are for ensembles who use as their base classifiers all models available in the set of baseline classifier. This might not be optimal, and as said in the previous section, there may be some models that are driving the ensembles to make miss-predictions. More careful construction of the base model set may improve the final performance of the ensembles.

For the context of `soweego`, a classifier with high precision allows for the confident upload those matches the classifier is sure about, so as to not degrade the quality of the data in Wikidata by adding false matches. Even if the matches found by a more precise algorithm may be less than those found by one with higher recall. The matches of which the model is unsure, but thinks are may be matches (e.g. those with a probability of match $50\% < x < 80\%$) can be uploaded to the `Mix'n'match` platform to be manually reviewed by the community. In this way, none of the potential matches is thrown away.

The process has up to now been considered as a single execution of the `soweego` pipeline. However, in reality this will be an iterative process, where the pipeline gets executed repeatedly after some interval of time. This means that the next time `soweego` is executed it will use all the confident matches it found in the previous run, plus those added by the community in the interval, and use those to train a new generation of classifiers. Doing so will improve the amount of training data available for the machine learning algorithms, which in turn means they may be better learn how to separate entities as matches and non-matches. This approach, of iterative training a record linkage pipeline, has been successfully used in [5]. It is, however, important to pick an appropriate interval of time to give time to the community to delete all incorrect matches made by `soweego` in the previous execution, so that these do not get included in the training data for the next generation.

8.1 Future Work

For cases where there are different, but related, entities, a technique called *collective record linkage* [6, 28, 45] has been developed. In these scenarios, a model is constructed (usually a graph) that symbolizes the dependencies between different entities. For example, the similarity between two actors may be dependent on the similarity score between said entities and the similarity between the movies they appear in.

It would be a good idea to use one of the collective matching methods to leverage the information of composed entities. For instance, in the data currently being used in the project we know that *actors* are part of *movies*, and that *musicians* are part of *bands*. The similarity among *actors/musicians* may be dependent upon the similarity of the movies in which they appear, or the bands in which they have performed.

The ensemble classifiers employed in this dissertation are all composed of the same base classifiers. Specifically, all of the baseline classifiers that are able to provide a probability output (i.e. all besides *lsvm*) have been used to create the ensembles. There may be some of these that drive the ensembles to make certain decisions, or may actually be contributing only noise for the final prediction. A better way to chose the structure of the ensembles would be to try different base classifiers, and see which combination is the optimal.

Another related to the above that should be considered is that the hyperparameters of the base classifiers were optimized by considering the model as an individual. It might be that the optimal hyperaparameters for the model when it is part of an ensemble are different from those used when it makes predictions individually.

The base classifiers of the ensembles only contain *one* of each type of baseline classifier. It would be interesting to also consider varying the number of times each base classifier appears besides varying the which of them are allowed to be part of the ensemble.

Seeing that ensembles did present an improvement, in general, when compared against the individual baseline classifiers, it would be interesting to see how other ensembling techniques which were not explored in the current dissertation, would perform.

The evaluations presented in this work are on only on datasets about people and bands. It would be interesting to see if these performances would still be similar for datasets about different types of entities, like *movies* and *musical works*. Maybe the fact that the distribution of some of their data fields like *name* would introduce important changes in the performance of the classifiers.

The grid search procedure to find the optimal hyperparameters of the baseline classifiers was only performed by evaluating said models on the *Discogs/Musician* concrete entity. As seen in section 5.4, features for different catalogs have markedly different distributions. It would be interesting to see the performance of each classifier when optimized separately for each catalog.

Bibliography

[1] Rohan A. Baxter, Peter Christen, and Tim Churches. A comparison of fast blocking methods for record linkage. In *KDD 2003*, 2003.

[2] Michael Bayer. Sqlalchemy. In Amy Brown and Greg Wilson, editors, *The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks*. aosabook.org, 2012.

[3] Robert M. Bell and Yehuda Koren. Lessons from the netflix prize challenge. *SIGKDD Explor. Newsl.*, 9(2):75–79, December 2007.

[4] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, February 2012.

[5] Indrajit Bhattacharya and Lise Getoor. Iterative record linkage for cleaning and integration. In *Proceedings of the 9th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, DMKD '04, pages 11–18, New York, NY, USA, 2004. ACM.

[6] Indrajit Bhattacharya and Lise Getoor. Collective entity resolution in relational data. *ACM Transactions on Knowledge Discovery from Data*, 1(1):5–es, 2007.

[7] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, pages 144–152, New York, NY, USA, 1992. ACM.

[8] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, Aug 1996.

[9] Leo Breiman. Bias, variance, and arcing classifiers. Technical report, Tech. Rep. 460, Statistics Department, University of California, Berkeley . . . , 1996.

[10] Leo Breiman. Stacked regressions. *Machine Learning*, 24(1):49–64, July 1996.

[11] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[12] Gavin C. Cawley and Nicola L.C. Talbot. On over-fitting in model selection and subsequent selection bias in performance evaluation. *J. Mach. Learn. Res.*, 11:2079–2107, August 2010.

[13] François Chollet et al. Keras. https://keras.io, 2015.

[14] Peter Christen. Automatic record linkage using seeded nearest neighbour and support vector machine classification. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 151–159, New York, NY, USA, 2008. ACM.

[15] Peter Christen. *Data matching : concepts and techniques for record linkage, entity resolution, and duplicate detection*. Springer, Berlin New York, 2012.

[16] Peter Christen. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE Transactions on Knowledge and Data Engineering*, 24(9):1537–1555, 2012.

[17] Tim Churches, Peter Christen, Kim Lim, and Justin Xi Zhu. Preparation of name and address data for record linkage using hidden markov models. *BMC Medical Informatics and Decision Making*, 2(1), December 2002.

[18] Marc Claesen and Bart De Moor. Hyperparameter search in machine learning, 2015.

[19] D E Clark. Practical introduction to record linkage for injury research. *Injury prevention : journal of the International Society for Child and Adolescent Injury Prevention*, 10(3):186–91, 2004.

[20] Aaron Clauset. A brief primer on probability distributions. In *Santa Fe Institute*, 2011.

[21] Munir Cochinwala, Verghese Kurien, Gail Lalk, and Dennis Shasha. Efficient data reconciliation. *Information Sciences*, 137(1-4):1–15, September 2001.

[22] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, September 1995.

[23] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, 2000.

[24] Balázs Csanád Csáji. Approximation with artificial neural networks. *Faculty of Sciences, Etvs Lornd University, Hungary*, 24:48, 2001.

[25] Jonathan de Bruin. recordlinkage (version 0.13.2). https://recordlinkage.readthedocs.io/en/latest/about.html, 2019. [Online; accessed 27-September-2019].

[26] Thomas G Dietterich. Ensemble methods in machine learning. In *International workshop on multiple classifier systems*, pages 1–15. Springer, 2000.

[27] Pedro Domingos and Michael Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, 29(2/3):103–130, 1997.

[28] Xin Dong, Alon Halevy, and Jayant Madhavan. Reference reconciliation in complex information spaces. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 85–96, New York, NY, USA, 2005. ACM.

[29] Timothy Dozat. Incorporating nesterov momentum into adam. 2016.

[30] Stacie B Dusetzina and Meyer A M. An overview of record linkage methods. https://www.ncbi.nlm.nih.gov/books/NBK253312/, Sep 2014. [Online; accessed 28-September-2019].

[31] Mohammadreza Ektefa, Fatimah Sidi, Hamidah Ibrahim, Marzanah A Jabar, and Sara Memar. A comparative study in classification techniques for unsupervised record linkage model. *Journal of Computer Science*, 7(3):341, 2011.

[32] M.G. Elfeky, V.S. Verykios, and A.K. Elmagarmid. TAILOR: a record linkage toolbox. In *Proceedings 18th International Conference on Data Engineering*. IEEE Comput. Soc.

[33] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. *J. Mach. Learn. Res.*, 9:1871–1874, June 2008.

[34] Ivan P. Fellegi and Alan B. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64(328):1183–1210, 1969.

[35] Sebastian Flennerhag. Ml-ensemble, November 2017.

[36] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[37] Ram Deepak Gottapu, Cihan Dagli, and Bharami Ali. Entity resolution using convolutional neural network. *Procedia Computer Science*, 95:153–158, 2016.

[38] Lifang Gu and Rohan Baxter. *Decision Models for Record Linkage*, pages 146–160. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006.

[39] David Hand and Peter Christen. A note on using the f-measure for evaluating record linkage algorithms. *Statistics and Computing*, 28(3):539–547, 2017.

[40] L.K. Hansen and P. Salamon. Neural network ensembles. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(10):993–1001, 1990.

[41] J. A. Hartigan and M. A. Wong. Algorithm AS 136: A k-means clustering algorithm. *Applied Statistics*, 28(1):100, 1979.

[42] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991.

[43] Cho-Jui Hsieh, Kai-Wei Chang, Chih-Jen Lin, S. Sathiya Keerthi, and S. Sundararajan. A dual coordinate descent method for large-scale linear svm. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 408–415, New York, NY, USA, 2008. ACM.

[44] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[45] Dmitri V. Kalashnikov and Sharad Mehrotra. Domain-independent data cleaning via analysis of entity-relationship graph. *ACM Trans. Database Syst.*, 31(2):716–767, June 2006.

[46] Kunho Kim, Madian Khabsa, and C. Lee Giles. Random forest dbscan for uspto inventor name disambiguation, 2016.

[47] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks, 2017.

[48] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, pages 1137–1143, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

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

[50] Andrew Mccallum and Kamal Nigam. A comparison of event models for naive bayes text classification. *Work Learn Text Categ*, 752, 05 2001.

[51] Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.

[52] Vangelis Metsis and et al. Spam filtering with naive bayes – which naive bayes? In *THIRD CONFERENCE ON EMAIL AND ANTI-SPAM (CEAS*, 2006.

[53] Marvin Minsky and Seymour A Papert. *Perceptrons: An introduction to computational geometry.* MIT press, 2017.

[54] Paul Munro, Hannu Toivonen, Geoffrey I. Webb, Wray Buntine, Peter Orbanz, Yee Whye Teh, Pascal Poupart, Claude Sammut, Caude Sammut, Hendrik Blockeel, Dev Rajnarayan, David Wolpert, Wulfram Gerstner, C. David Page, Sriraam Natarajan, and Geoffrey Hinton. Bias variance decomposition. In *Encyclopedia of Machine Learning*, pages 100–101. Springer US, 2011.

[55] Howard B. Newcombe and James M. Kennedy. Record linkage: Making maximum use of the discriminating power of identifying information. *Commun. ACM*, 5(11):563–566, November 1962.

[56] David Opitz and Richard Maclin. Popular ensemble methods: An empirical study. *Journal of artificial intelligence research*, 11:169–198, 1999.

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

[58] Edward H Porter, William E Winkler, et al. Approximate string comparison and its effect on an advanced record linkage system. In *Advanced record linkage system. US Bureau of the Census, Research Report.* Citeseer, 1997.

[59] David Powers and Ailab. Evaluation: From precision, recall and f-measure to roc, informedness, markedness & correlation. *J. Mach. Learn. Technol*, 2:2229–3981, 01 2011.

[60] Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulletin*, 23:2000, 2000.

[61] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, October 1986.

[62] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[63] Adrian Sayers, Yoav Ben-Shlomo, Ashley W Blom, and Fiona Steele. Probabilistic record linkage. *International Journal of Epidemiology*, 45(3):954–964, December 2015.

[64] Robert E. Schapire. The strength of weak learnability. *Machine Learning*, 5(2):197–227, June 1990.

[65] Amit Singhal. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24:35–43, 2001.

[66] Mark J Van der Laan, Eric C Polley, and Alan E Hubbard. Super learner. *Statistical applications in genetics and molecular biology*, 6(1), 2007.

[67] G. van Rossum. Python tutorial. Technical Report CS-R9526, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, May 1995.

[68] Kha Vo, Jitendra Jonnagaddala, and Siaw-Teng Liaw. Statistical supervised meta-ensemble algorithm for medical record linkage. *Journal of Biomedical Informatics*, 95:103220, July 2019.

[69] Michael Waskom, Olga Botvinnik, Paul Hobson, John B. Cole, Yaroslav Halchenko, Stephan Hoyer, Alistair Miles, Tom Augspurger, Tal Yarkoni, Tobias Megies, Luis Pedro Coelho, Daniel Wehner, Cynddl, Erik Ziegler, Diego0020, Yury V. Zaytsev, Travis Hoppe, Skipper Seabold, Phillip Cloud, Miikka Koskinen, Kyle Meyer, Adel Qalieh, and Dan Allan. Seaborn: V0.5.0 (november 2014), 2014.

[70] D. Randall Wilson. Beyond probabilistic record linkage: Using neural networks and complex features to improve genealogical record linkage. In *The 2011 International Joint Conference on Neural Networks*. IEEE, July 2011.

[71] William E Winkler. String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. 1990.

[72] William E Winkler. Overview of record linkage and current research directions. In *Bureau of the Census*. Citeseer, 2006.

[73] David H Wolpert. Stacked generalization. *Neural networks*, 5(2):241–259, 1992.

[74] Matthew D. Zeiler. Adadelta: An adaptive learning rate method, 2012.

[75] Harry Zhang. The optimality of naive bayes. volume 2, 01 2004.

[76] Zhi-Hua Zhou. *Ensemble methods: foundations and algorithms*. Chapman and Hall/CRC, 2012.

Apendix A Apendix

A.1 Glossary

This section will review the terminology which is extensively used in this work when talking about data and it's sources.

A.1.1 Entity

An entity refers to a single data point in a dataset (catalog). *Entities* are what is matched when performing record linkage. An entity is composed of one identifier, and a collection of fields (either quantitative or qualitative) which are specific to each problem.

In the people use case, the entities will most likely have fields representing personal information *(name, birth date, death date, etc.)*. If, on the other hand there is a problem where its necessary to perform record linkage on houses, then it would probably have fields like `total area, number of floors, number of rooms, etc`.

A.1.2 Catalog

In the present work, any source of entities is called a *catalog*. The terms *catalog* and *dataset* are synonyms. Any source of data can be a catalog: books, databases, spreadsheets, etc.

Note that the term *catalog* is not specific to Record Linkage. It is instead a term adopted from the Mix'n'Match[1] project, and it signifies *sources of data which can be imported into Wikidata*. For our project, the term is changed slightly to mean *any source of data*.

An *external catalog* is a *catalog* that is not *Wikidata*, and from which information should be imported into the latter.

As stated before, entities are always assigned an identifier. This said identifier is always unique, and specific to a distinct catalog (rarely this identifier is shared among different catalogs).

When performing record linkage, it is desirable to end up with a list of identifiers pairs, where each pair describes a single real-world entity, and its identifiers tells us which record in the corresponding dataset has information on the specific entity.

A.1.3 Source and Target

As stated before, the goal is to take Wikidata entities and find their corresponding links in external catalogs. The relation between these two can be captured by the terms *source* and *target*. *Source* is

[1] `https://tools.wmflabs.org/mix-n-match`

the catalog where the entities that should be matched are taken from (i.e. *Wikidata*), and *target* is the catalog against which the entities from the source should be matched (i.e. the external catalogs).

This is especially useful when there is no need to match all the catalogs amongst themselves, since matching a *source catalog* against multiple target ones requires a much more diminished number of comparisons than it would do if matching all catalogs among themselves.

When using a *source catalog*, the record linkage algorithm is no longer matching source entities into a unique latent (underlying) entity, rather, it will match them against a concrete entity (e.g. an entity from the source catalog).

In related record linkage studies, these terms are also commonly referred to as [63] *master file* (instead of source), and *file of interest* (instead of target) .

A.2 Distribution of features

This section contains the plots which characterize the distributions of feature values extracted for different concrete entities.

Figure A.1 and Figure A.2 present the distribution plot of features in the classification set. While Figure A.3 and Figure A.4 presents the distribution for the training set.

Figure A.5 and Figure A.6 show the distribution for features of negative samples from the training set. While Figure A.7 and Figure A.8 show those for the positive samples.

By comparing the classification and training sets one of the first things that can be noticed is that for features in all cases there is one feature which is almost always one, while in the classification set this happens only in some cases. The featrures with this behaviour are one of the *exact measures*: *exact anmes*, *exact url*.

It is also interesting to note that in most plots, but especially in those for the classification set, the majority of the features values seem to be below *0.7*.

For the training set distributions there is a bit more variations, and feature values are also higher feature values are more common. Here it can be noted that the *exact* features are much more prominent than in the classification set.

Comparing the positive and negative training distributions one can notice that for the positive distributions, in all cases, the exact matches measures are much higher than the others.

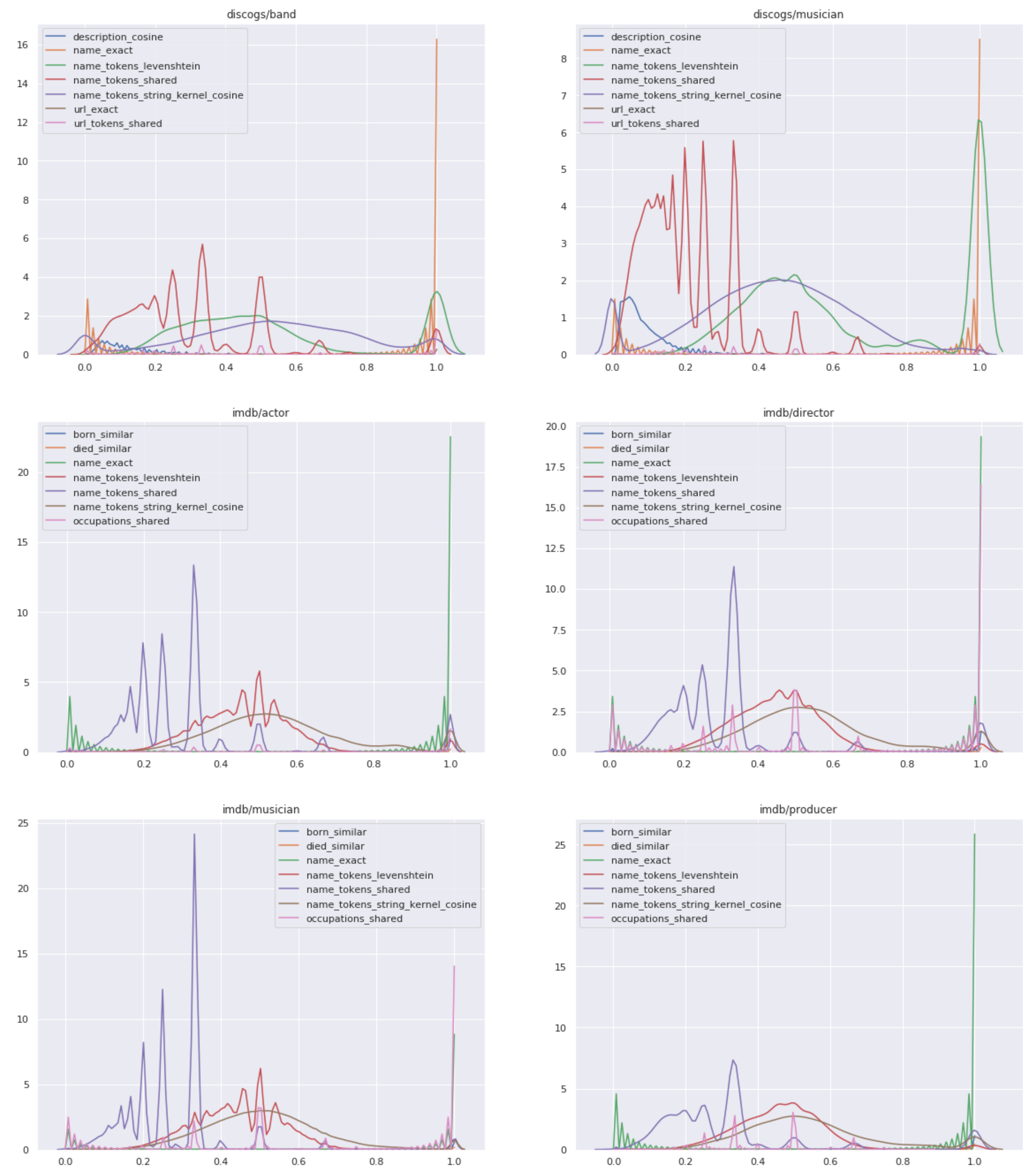


Figure A.1: Feature distribution for the classification set.

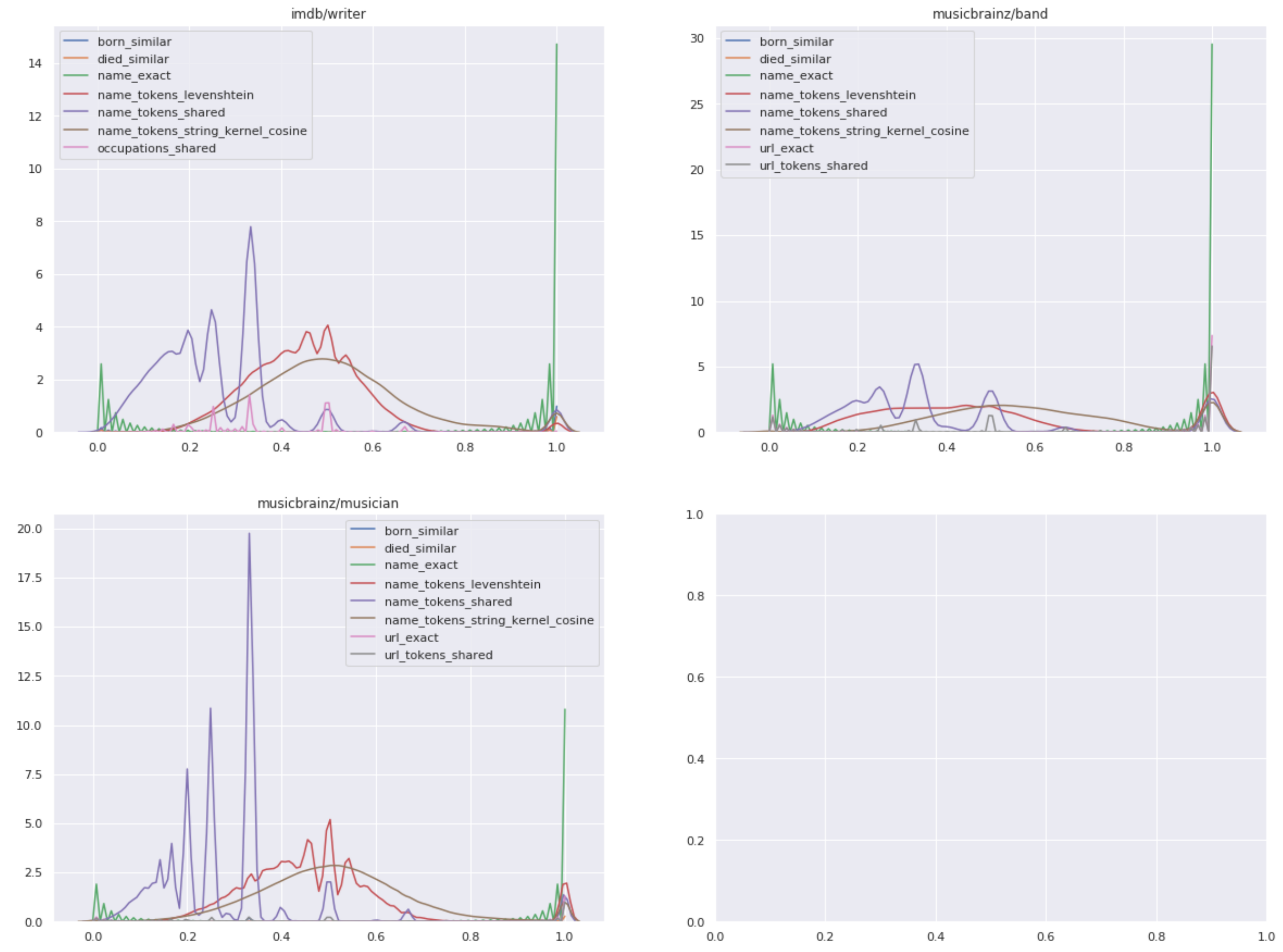


Figure A.2: Feature distribution for the classification set.

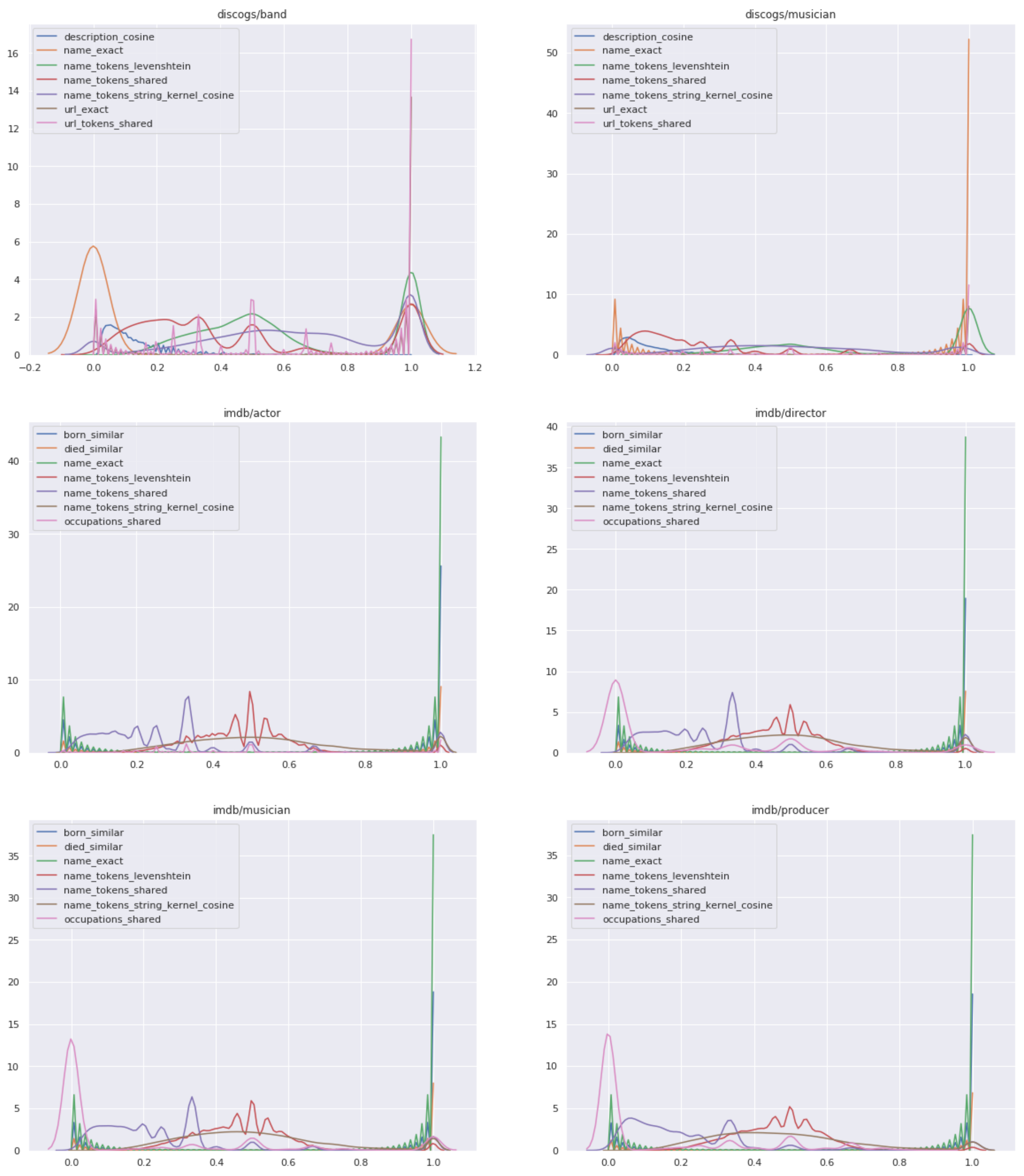


Figure A.3: Feature distribution for the training set (1).

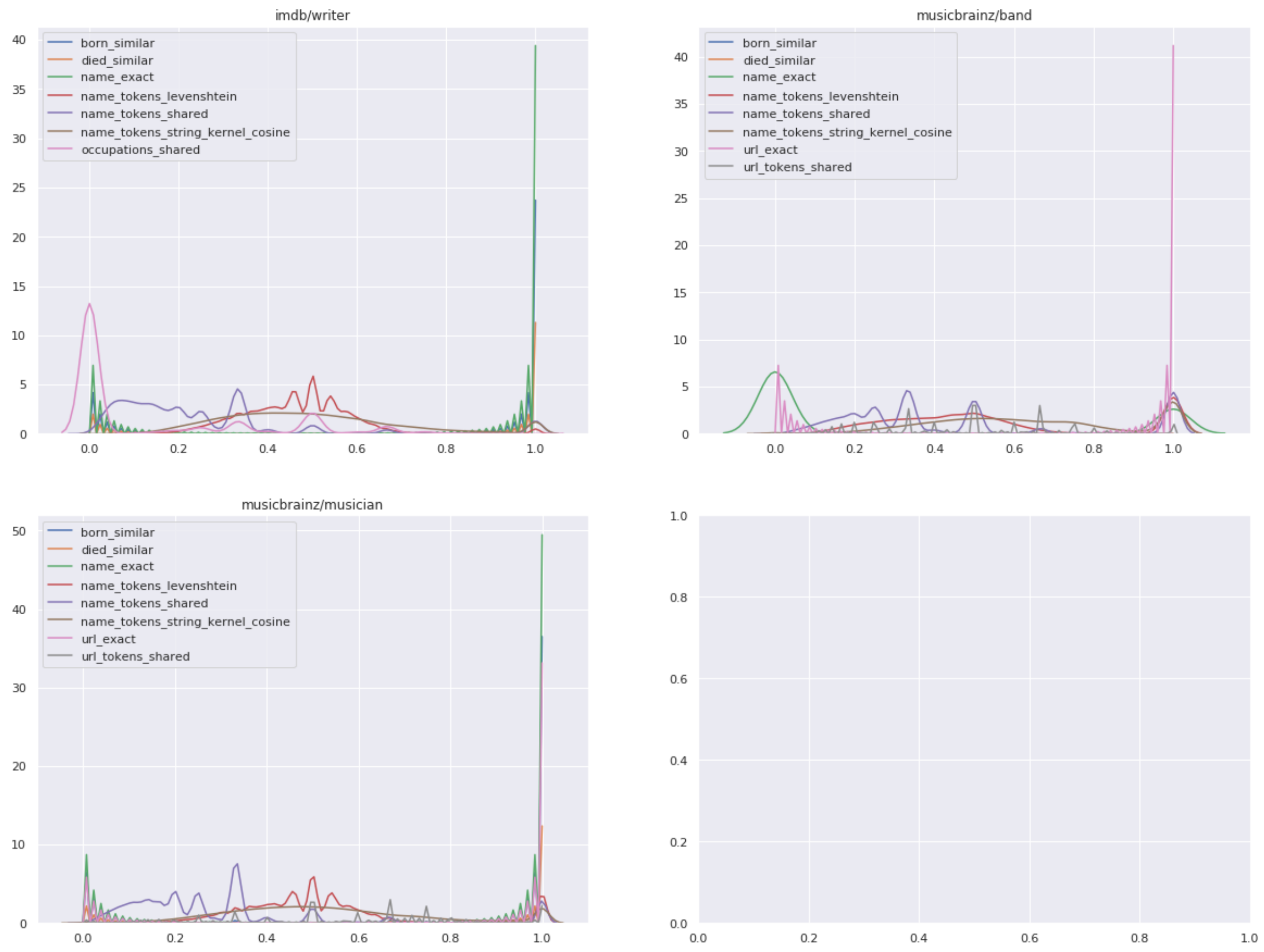


Figure A.4: Feature distribution for the training set (2).

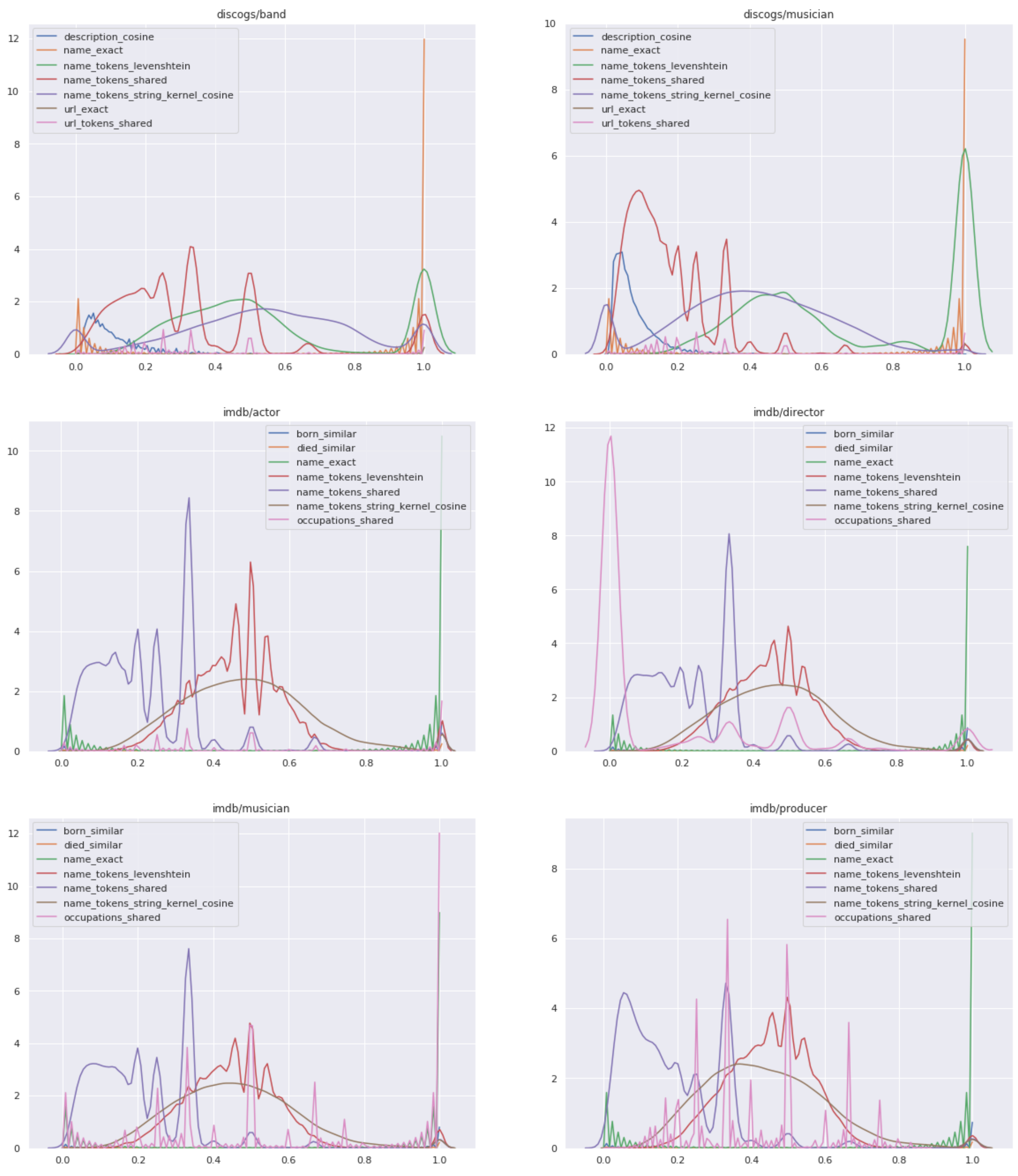


Figure A.5: Feature distribution for the negative samples of the training set (1).

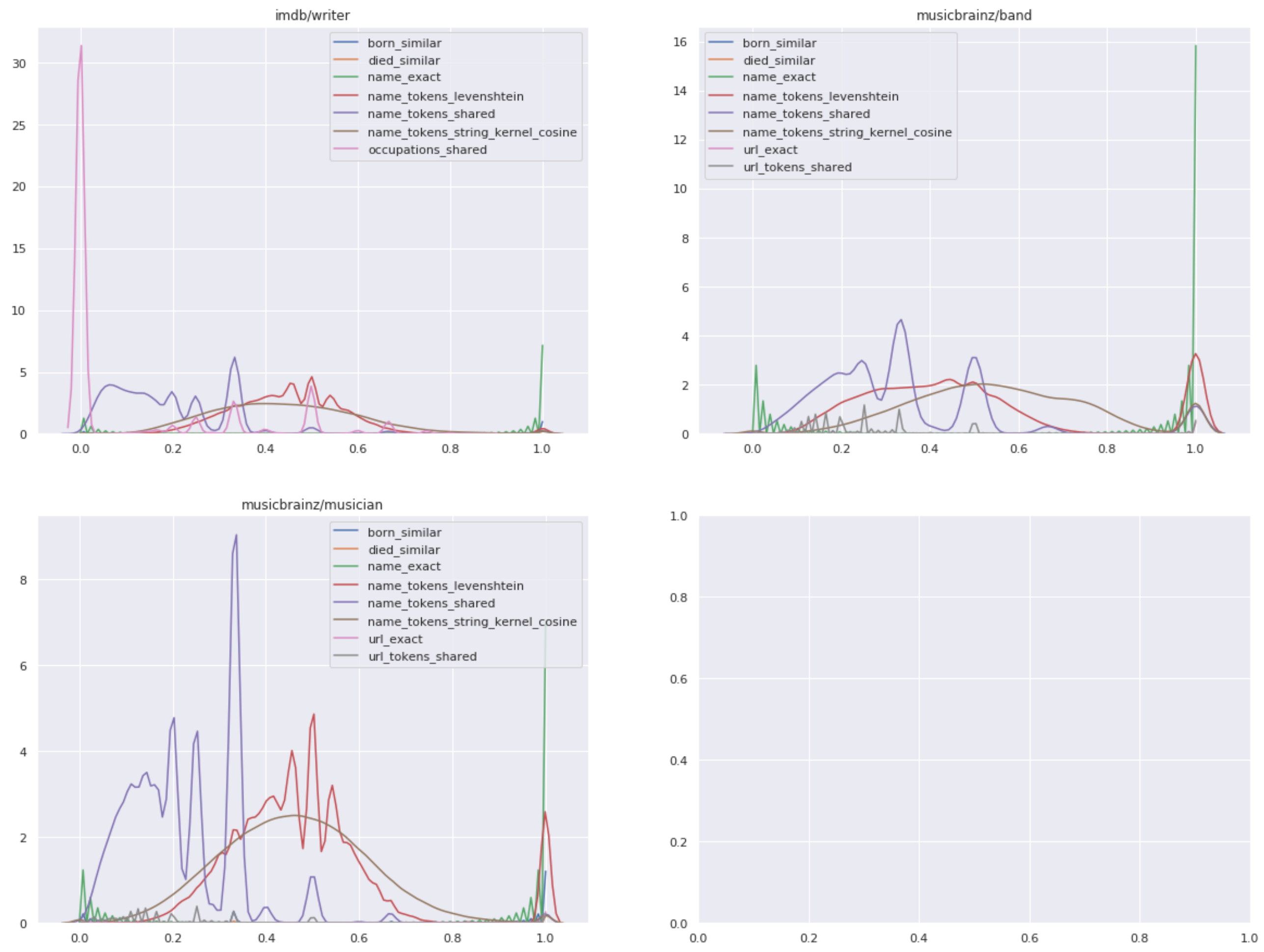


Figure A.6: Feature distribution for the negative samples of the training set (2).

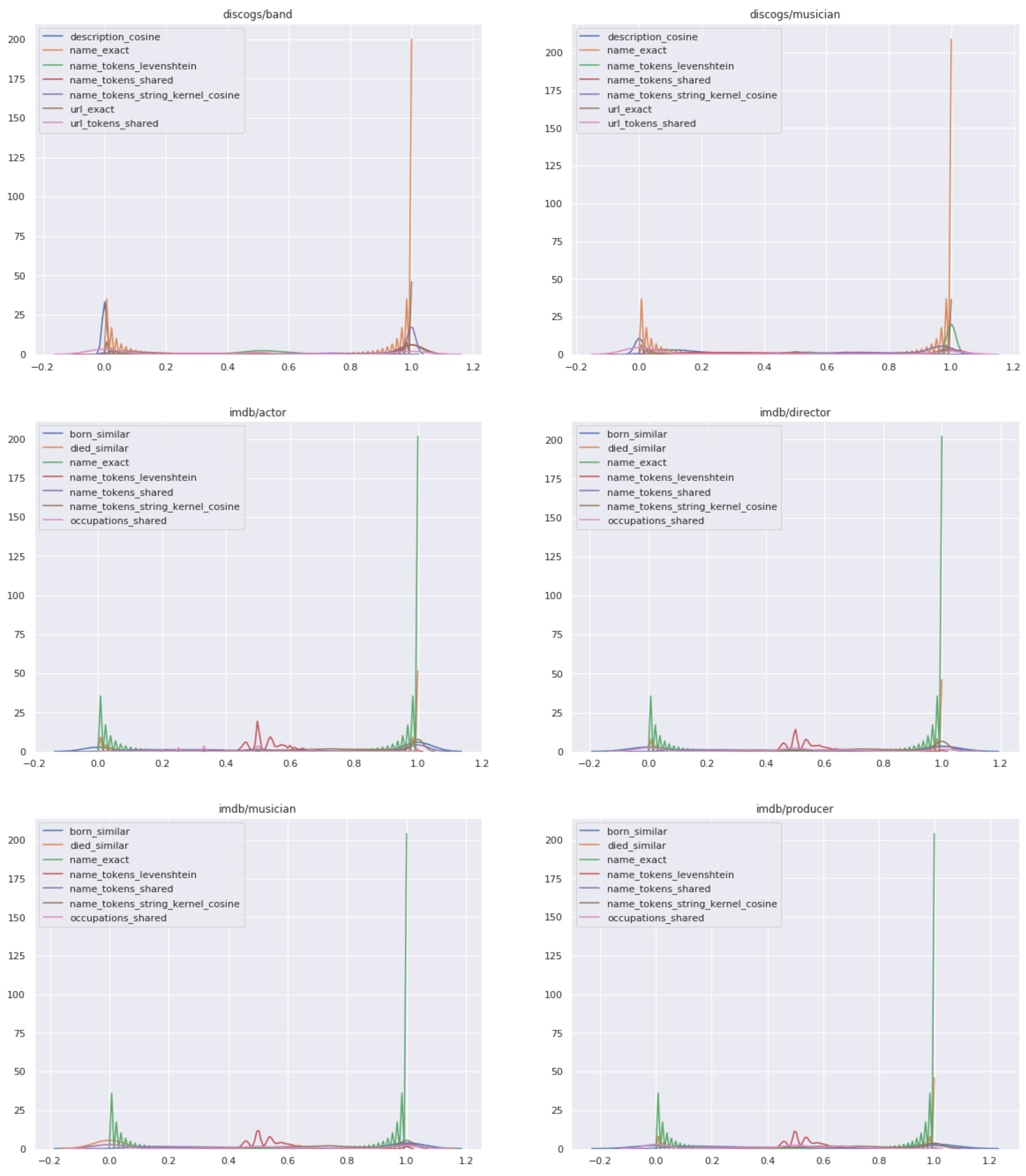


Figure A.7: Feature distribution for the positive samples of the training set (1).

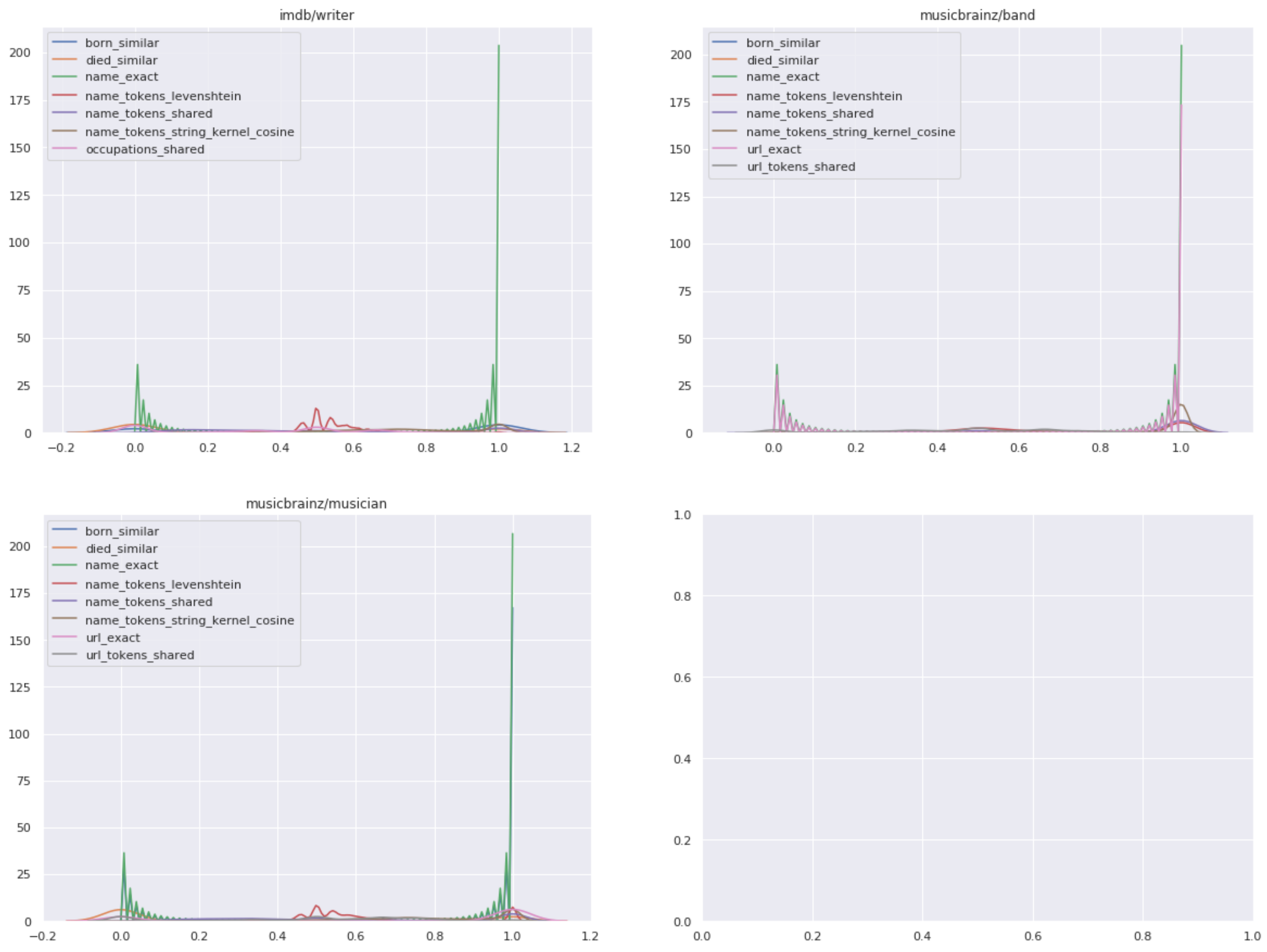


Figure A.8: Feature distribution for the positive samples of the training set (2).