EURECOM

Music Recommendation when Exploring a City

Authors:
Lorenzo Canale
Fabio Ellena

Supervisor:
Pasquale Lisena
Raphaël Troncy

Abstract: Our project aims at making the user discover the existing connections that link music to a Place, by stimulating his curiosity proposing noncommon relations. In this project, we show a new approach for finding artists linked to a place of interest (POI). We propose an entity linking framework built on top of DBpedia, 3cixty, and DOREMUS that is able to reach editorial accuracy in the interlinking process. Furthermore, we present a graph exploration algorithm that is able to find deep connections in a short time and discriminate between them by looking at those that can be the most interesting.

Contents

1 Introduction

Most of the available music recommender systems suggest music without taking into consideration the user's context [1]. This means that a generic recommender system will propose the same set of songs regardless of the user mood, location, activity. Finding music that suits a POI can be viewed as a context-aware recommendation problem, the place is the context for consuming the recommendation. The main challenge that one must face when addressing the above-mentioned goal is related to the fact that POIs and music are two rather different domains, and there is no obvious way to match such heterogeneous items. However, with the advent of the Semantic Web, new opportunities arise to face the above difficulties. We will show that using generic semantic datasets we can recommend artists that are linked to the user location.

1.1 Motivation and context

In the past, many tried to link POIs and music in many different ways. managed to link music and POIs by using a common set of tags that describe emotional properties both for music and POIs [2]. This method was completely supervised, in fact it required the definition of a set of emotions for songs and POIs. This operation has been done by hand and is hardly scalable. In another project [3], a semantic approach has been used by restricting the subspace of DBpedia [1] by identifying classes related to the domains of interest, and the relations existing between instances of such classes. In this way, they managed to find relations in a set of paths constrained to pass belong to those classes. Looking at the future works proposition, they proposed the idea to exploit arbitrary semantic relations between POIs and musicians. In order to do this, they proposed to use Relfinder [4] and set the connections' weights using an heuristic. In fact, one of the bad things was that connections weights were assigned by experts and were not provided in the paper. Our intention is to build our approach on top of this paper and find relations in a completely unsupervised way: this means that we intend to define a pathfinder that finds paths and a path discriminator that selects the interesting relations. We aim at reducing the need for human intervention in defining classes and weights with the hope that this approach will be completely data driven and easily scalable to different cities. Moreover, we will use different datasets with the intention to exploit their implicit advantages and obtain overall a better recommender system.

1.2 Research problems

Finding relations inside DBpedia is a known problem which has been solved in the past. So we could simply look for POIs and Artists in DBpedia and then link them using known algorithms. Unfortunately, DBpedia is not perfect: it is true that it contains

[1] DBpedia: http://wiki.dbpedia.org/

POIs and Artists, but these entities are not cleanly defined, they usually contain dirty data that can tremendously affect our performances. On the other side, POIs and Artists can be found in highly specialized datasets, but unfortunately, they do not contain links to each other, and even worse they are not linked to DBpedia. This means that by using these databases alone we cannot find relations. What is done in this cases is to link highly specialized datasets to DBpedia, in this way we can exploit the advantages of both worlds: highly specialized datasets give us confidence and trust in our main entities (POIs and Artists), while DBpedia allows us to find relations between them thanks to its generality. Once we have our two sets, we need to find paths that link them. This problem has been solved in the past with Relfinder, a tool that is able to find relations between entities. What we have seen is that POIs and Artists are generally far from each other, in fact usually we need 5 hops to link them, this is more that the average DBpedia degree and it is also more than what Relfinder can do, in fact, Relfinder query times are too high once we set more that 4 hops. This lead us to create a pathfinder that can work in acceptable times with 100.000 pairs that are far from each other.

1.3 Contributions

In this project, we show a new approach for finding artists linked to a place of interest (POI). We propose an entity linking framework built upon DBpedia, 3cixty [2], and Doremus [3] that is able to reach editorial accuracy in the interlinking process. Furthermore, we present a graph exploration algorithm that is able to find deep connections in a short time and discriminate between them by looking at those that can be the most interesting.

1.4 Report structure

The report is divided into five sections, one for each part of the project. The first two parts are about linking external datasets to DBpedia, this is also known as entity linking and we perform it in the optic of enriching our data with the inter-concept links present in DBpedia. The third treats the Pathfinder, a module of our system that is able to find deep connections between two entities in DBpedia. The fourth describes the whole implementation of the REST API that allows the application to work smoothly. The fifth part covers the mobile web application and the technologies involved in its functioning.

[2] 3cixty: https://www.3cixty.com/
[3] DOREMUS: http://www.doremus.org/

2 Interlinking and enriching POIs

2.1 Problem description

Interlinking different datasets is not trivial, many approaches that aimed at a semi-supervised interlinking have been proposed, but this is still an active research topic. One of the main problems that must be faced in an interlinking process is the ontology definition, in fact, specialized datasets tend to have a very strict ontology, while DBpedia does not enforce it. This means that in the case of POIs, not all places in nice are linked to nice, thus it is difficult to even correctly define a POI in Nice. Since we are dealing with a medium sized city such as Nice, we can't allow loosing some matchings and we need to carefully analyze the DBpedia ontology in order to leave out a minimum part of Nice POIs.

2.2 Datasets: 3cixty, DBpedia, Wikidata

2.2.1 3cixty

3cixty is a semantic dataset that contains aggregated informations of POIs and events. The dataset is the result of an aggregation process that join the POI informations from different datasets e.g. Facebook, Foursquare, Yelp ... In this project 3cixty is used as a trusted source of POIs for Nice. POIs have many informations, such as the category, the address, comments from users, and coordinates. In order to get all the POIs in Nice, we simply used the POIs endpoint selecting the Nice city.

2.2.2 DBpedia

DBpedia is a project that aims at connecting the semantic world. We used it to get the POIs in Nice. In order to get the POIs from DBpedia, it is necessary to perform SPARQL queries against the SPARQL endpoint. POIs are retrieved using two different sparql queries that target different kind of resources:

1. We perform a query that looks for all Geoentities that are in the area of nice. An entity is in the area of Nice if its coordinates are in a custom box that contains all the area of Nice. Unfortunately there are many POIs that are without coordinates,so they are not selected by this query.

```
SELECT ?place ?placeLabel ?lat ?long
WHERE {
    ?place geo:lat ?lat.
    ?place geo:long ?long.
    ?place rdfs:label ?placeLabel.
    FILTER(
```

```
            xsd:double(?lat)  <= 43.80 &&
            xsd:double(?lat) >= 43.63 &&
            xsd:double(?long) <= 7.36 &&
            xsd:double(?long) >= 7.14
        )
    }
```

Figure 1: Area for POIs selection

2. We take all resources that have as a parent either the Nice resource or the Category:Nice resource. Then we repeat this operation for each subcategory of Category:Nice in a recursive way. Among these resources, we keep those which type is in a set of classes defined by hand.

2.2.3 Wikidata

Wikidata [4] is a project that aims at connecting the semantic world. We used it to get the POIs in Nice. Respect to DBpedia, Wikidata has a well defined ontology that is strictly enforced. This means that it is easier to select what we want. Moreover, most of POIs in wikidata have coordinates, so we can get them with a single query. Since wikidata supports complex geo-based queries, we can make a query that looks for all POIs in a range of 25 Kilometers from the center of Nice.

[4] Wikidata: https://www.wikidata.org/

2.3 Entity Linking process

In order to link our two set, we copy them locally and then we perform the linking process. Having the entities locally is crucial, in fact it allows us to make way more complex operations respect to those that can be done with SPARQL queries.

2.3.1 Properties

The first phase is to actually get the interesting properties of the entities. Regarding those that come from DBpedia, we use only the labels. Regarding the labels, we take them in all the available languages. In this way we can do a multilingual match that improves our accuracy. Since this is not intuitive, here is an example. DBpedia labels: 'Conservatory of Nice' 'Conservatoire de Nice' 3cixty label: 'Conservatoire de nice' In this case if we use the english label, we do not obtain a perfect match, while with the french label we obtain a perfect match. We do not use coordinates because not all entities have them. Moreover, we have seen experimentally that using the DBpedia coordinates we obtain worse results because they are inaccurate. Regarding 3cixty entities, they have very accurate coordinates, and few labels. For this phase we do not need the coordinates, while regarding the labels, we keep the first one.

2.3.2 Transformations

At this point of the pipeline, we have labels that are encoded into unicode utf-8. Common similarity metrics accepts only plain ASCII strings as input, so we need to convert them into ASCII. Since utf-8 can represent way more characters than ASCII, we need to strip non ASCII characters from our labels. This is done in an intelligent way using the python library *'unidecode'*[5], that is able to convert from unicode to ASCII by changing or deleting a minimum set of characters. For example, an accented letter like 'è' can't be represented in ASCII, and most of the libraries simply delete it, while *'unidecode'* converts it to the nearest ASCII character: in this case 'è' is converted to 'e'. In this way we can retain most of the original information after this conversion process.

The second step is probably the most important, and is the normalization of names. Some similarity metrics are extremely sensitive to noise. For example, when we use simple levenstein distance between ('airport','airport nice') is 5, while the distance ('airport','port') is 4. This is enough to raise some concerns: the simple addition of the city name can turn a perfect match into a mediocre one. In order to fix this problem we add the city name to all the labels where it is not present.

The third step is to strip all non alphabetical characters and to convert to lowercase all strings, this off course gives us cleaner strings to compare. The whole transformation pipeline aims at maximizing the similarity measure between correct matches. This is

[5]Unidecode: https://pypi.python.org/pypi/Unidecode

extremely important, because later we can set a threshold to separate matches that have a good confidence from matches with a low confidence. If we do not perform the transformation phase, we would obtain correct matches that have the same confidence of the incorrect ones because we have basically some noise in all the strings.

2.3.3 Similarity measures and matching algorithm

Now that strings are clean, we can proceed to the actual matching. A first problem rises: we want to match a 3cixty poi to a dbpedia poi, or we want to match a dbpedia poi to a 3cixty poi? In the first case we are sure that each 3cixty poi will be matched to a single dbpedia poi, but there is the risk that multiple 3cixty poi are matched to the same DBpedia poi. Alternatively, we are sure that each DBpedia poi is matched to at most a 3cixty poi, but there is the risk that many DBpedia poi are matched to the same 3cixty poi. Regarding DBpedia, we are sure that POIs are univoque, while we know that for the same POI, in 3cixty there are multiple ones. Since at the end we will be using 3cixty coordinates, we opt for the first choice: if a DBpedia poi will be matched to multiple 3cixty pois, than we will follow an heuristic to choose the best match.

Regarding the similarity metric among strings, we use multiple similarity measures based on levenstein distance. We chose to use different metrics because we have seen experimentally that in this specific case, strings that represents POIs, a single similarity metric is not enough. From a technical point of view, we used a set of the similarity metrics provided by the 'fuzzywuzzy'[6] python module.

2.3.4 Aggregations

Until now, we took one by one the 3cixty labels and we calculated the different similarity metrics against all DBpedia labels. The aggregation algorithm is run for each 3cixty POI and can be described as follow:

1. For each similarity metric, take the three best matching labels and for each of them, calculate the average of all the similarity metrics.

2. The final match is the POI corresponding with the label with the highest score.

Why we don't simply take the average, or the minimum, or the maximum? The reasoning behind this is that by taking the average we are flattening all the scores, and there is the risk that the string with the highest similarity is a string that is mediocre in all the different metrics. If a string is mediocre in all comparisons, then we do not trust it. Instead, with our methodology we are keeping only those matches that excel in some similarity metric, and then we take the one that performs better on average. We think

[6]FuzzyWuzzy: https://pypi.python.org/pypi/fuzzywuzzy

that in this way we can filter out all those matches with average strings that matches almost anything.

2.3.5 Filtering

Now that the matching is done, we need to filter our results.

The first operation to do is to eliminate all those POIs that match with Nice, but that have a label different from Nice. This happens because we added Nice to all the strings. That addition was helpful, and it also created a safe default that we can easily filter. Most of these POIs filtered in this way are acronyms that match with Nice e.g. 'ATP Nice' matches with Nice because half of the string is Nice and is artificial, because we added it.

The second operation to perform is the filtering based on the average score. The average score can be interpreted as a confidence. Intuitively, high scores corresponds to correct matches, while low scores corresponds to bad matches. The importance of the selection, transformation, similarity metric and aggregation pipeline is extremely high, in fact we want a confidence which we can trust. An untrusted confidence gives us no additional informations about good matches and bad matches. At this point we filter all matches with a score below a given threshold.

Now we are at the half of our work, for each 3cixty POI we have a link to a DBpedia POI, but we might have multiple 3cixty POIs linked to the same DBpedia POI. We aim to a 1 to 1 relation, so we need to choose among a group of 3cixty POIs linked to the same DBpedia POI, the most representative. We have few possibilities:

1. Pick the POI that minimize the distance to all the other POIs, this is like applying the K-medoids clustering with K equal 1.
2. Pick the POI which is nearest to the center
3. Pick the POI which is nearest to the DBpedia POI, if coordinates are available
4. Pick the POI which is nearest to the position found on Google Maps [7]
5. Link our data to GeoNames [8], a dataset of POIs

The first method might be the best one, but it works if most of the POIs are actually clustered. If they are very sparse it does not work. The second method is biased towards POIs near the center. The third method can work only with DBpedia POIs with coordinates, and gives good results if DBpedia coordinates are good, but we already know that this is not true. The fourth method might be the best one, but we need to look outside the semantic world and trust the Google Maps API. Here the problem is

[7] Google Maps: https://www.google.fr/maps
[8] GeoNames: http://www.geonames.org/

that we have no control over the API, but we know that Google Maps is very good at matching a name with a place. The fourth method could be the best one, in fact we can exploit GeoNames coordinates that are trusted. The main problem is the fact that not all DBpedia POIs have a corresponding POI. For simplicity and reliability, we used the Google Maps API because we trust it. We use the API by sending the DBpedia label to the service, the response contains the coordinates of the POI associated to the DBpedia label. At this point we select the nearest 3cixty POI and we keep it if the distance is less than 2 kilometers.

2.4 Evaluation (Google Maps)

The evaluation part is done using the Google Maps API. As done before, we use the distance between the 3cixty POI coordinates and the POI coordinates provided by the API. Since we used the Google Maps API during our process, we expect that distances are low, in fact we know that each POI distance is less than 2 kilometers. What we want to check here is the distance distribution.

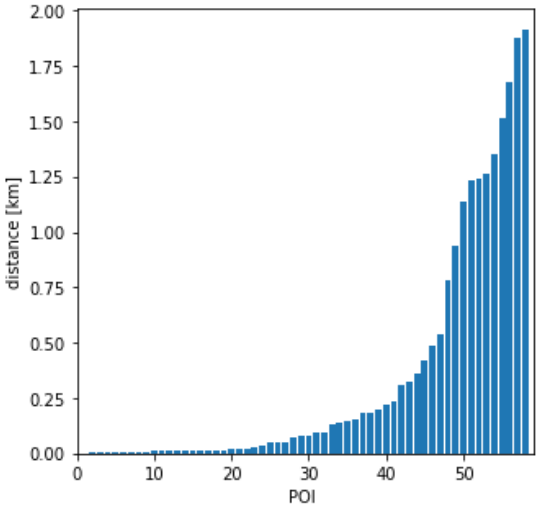


Figure 2: distance distribution of POIs

Looking at the distance distribution we note that most of the POIs are actually near to the Google Maps coordinates, while for few POIs this is not true. The main reason for this is the inherent error that arise when we define a POI which occupies a big surface with coordinates that refer to a specific point. For example the surface of the airport of Nice is quite big and different services can locate it in different points of his surface. In the case of the airport, the distance is actually 500 meters. This means that if it is true that most of the POIs are under 500 meters, we can't use this as a threshold because we

would leave out a large part of the correct POIs. At the end of our POI linking process we have 58 POIs that will be matched with the artists.

3 Interlinking and enriching Artists

3.1 Problem description

DBpedia contains Artists and their works in a disorganized way. This consist a huge limit because we do not trust the data that we have in DBpedia. Even the simpler operations, such as finding all artists and all songs are extremely difficult due to the non-strict ontology of DBpedia. In order to have some order, we use DOREMUS, which contains trusted informations about artists and songs in a strict ontology.

3.2 Datasets: DOREMUS, DBpedia, Wikidata

3.2.1 DOREMUS

DOREMUS is a semantic dataset that aims to a fine description of musical works in the fields of traditional and classical music: related musical or creation events, relations to authors, cultural background, interpretations, social functions, etc. With DOREMUS we can have absolute trust in the data regarding artists, and we can then found corresponding artists in DBpedia with a custom linking process. Retrieving all artists from DOREMUS is extremely easy, in fact we just need to ask for the Persons in the datasets that are linked to a work at least. The performed query is written below; we retrieved not only the artist uri but also the artist name, the id, the born year, the death year and the DBpedia "sameAs" already present. These informations are useful as input data for our similarity metric and for the final evaluation.

```
select $uri_doremus
(group_concat( distinct $id_doremus;separator="|||") as $id_doremus)
(group_concat(distinct $name_doremus;separator="|||") as $name_doremus)
(group_concat(distinct $born_year;separator="|||") as $born_year)
(group_concat( distinct $death_year;separator="|||") as $death_year)
(group_concat(distinct $dbpedia;separator="|||") as $dbpedia)
where {
$uri_doremus foaf:name $name_doremus.
$uri_doremus ecrm:P131_is_identified_by $id_doremus.
$uri_doremus a ecrm:E21_Person.
$activity ecrm:P14_carried_out_by $uri_doremus.
OPTIONAL{$uri_doremus owl:sameAs $dbpedia.
FILTER (strStarts(str($dbpedia), 'http://dbpedia.org/resource/'))}
OPTIONAL{$uri_doremus ecrm:P98i_was_born $born_year}
```

```
OPTIONAL{$uri_doremus ecrm:P100i_died_in $death_year}
}
group by $uri_doremus
order by $name_doremus
```

3.2.2 DBpedia

In this case we don't use directly today's DBpedia version but a DBpedia 2015 dump. This because we used an Elastic Search in which this dump was already uploaded.

3.2.3 Wikidata

If Wikidata was extremely reliable for the POIs, it is not that strict about musical artists, in fact we need to get all persons and then filter them in some way, looking at classes that are related to music. In particular we considered all people that are composers of some works.

3.3 Entity Linking process (DBpedia)

Here we do the same thing that we did with POIs, we need to extract interesting properties from our entities and then use them to find the correct matches. With POIs we had one hundred entities from DBpedia, and we could do all operations locally. Here we would like to do the same, but we are dealing with millions of persons, and retrieving them and doing all operations locally is feasible but takes more time. For each artist in DOREMUS, we would compare his label with the label of millions of persons, and in Doremus we have thousands of artists.

In order to speedup operations, we use a dump of DBpedia indexed on ElasticSearch. This allow us to perform all the comparison in a fraction of time thanks to the use of indexes.

Instead for Wikidata the process was different because we didn't use the Elastic Search; in fact this was already set up for DBpedia but not for Wikidata. In addition there are less artists in Wikidata so the problem to handle million of entities is reduced.

This difference of implementation for DBpedia and Wikidata was also the occasion to experiment two different streets, with and without the Elastic Search. We'll resume advantages and issues of both methods at the end of this session.

3.3.1 Retrieving Elastic Search Resources

We are not interested in retrieving all DBpedia people but only the ones that are linked with the Doremus entities. So we query ElasticSearch two times for every Doremus artist considering the name sting and the id string(the id is composed by the name and the birth and death date).

1. `GET dbpedia201510/_search?q=*NAME_DOREMUS*`
2. `GET dbpedia201510/_search?q=*ID_DOREMUS*`

Performing this operation,for every Doremus instance, two results can occur:

1. we find a list of possible DBpedia matching candidates;
2. we don't find linked DBpedia entities; this is pretty rare and it means that probably this entity is not present on DBpedia;

Considering the first case, ES returns also a score, associated with every entity; the first time we performed the matching, we have considered this; however we have noted that it drives to bad results. So we have preferred to not consider it and to score every returned entity using the method that we'll describe below.

3.3.2 Enrich Elastic Search data

At the end of the Elastic Search queries we had basically a table formed by these columns:

1. doremus artist uri;
2. doremus artist name;
3. doremus artist birth year;
4. doremus artist death year;
5. doremus candidate uri;

Every row is formed by a possible match Doremus artist - DBpedia artist candidate. Our goal is to understand which is, between the possible candidates, the exact one. On the Doremus side we had all the information associated to every artists as label,birth, death. On the other side instead we have only a possible matching URI. So we queried DBpedia SPARQL endpoint to get these informations:

1. the list of associated labels/names;
2. the birth year;
3. the death year;
4. the related categories;

To get the labels we checked both `rdfs:labels` and `foaf:name`. In addition we took the labels associated to each language to have better chances of matching them with the Doremus one. Actually this is the real sense of querying Dbpedia endpoint; in fact also the ES retrieves the linked labels, but not for each language.

To get the years informations we have considered different properties because the most of the time the birth year is written as object of the property `dct:subject`. Actually it's part of the object, e.g.`dct:1998_births`. However there are cases in which the year is saved as object of other properties as `dbo:birthDate`, `dbo:birthYear`, `dbo:birthDate`, `dbp:dateOfBirth`. The same considerations could be done for the death year. In addition performing a query for every year asking about all people born in that year is faster than the inverse: performing a query for every resource asking about the death and the birth year. In fact we have a lot of entities retrieved by the ES queries; instead we have only about 2000 years from 0 to 2017. We used the first approach joining the result with ElasticSearch entities at the end. To be faster, we'll perform a query for every 10 years, using the UNION operator.

We had also to check that the entities retrieved by the Elastic Search are composers,musicians, artists or at least people. So it was necessary to get the types of each resource, in particular the ones linked with the idea of composer. When we performed the final scoring we assigned higher scores more specific the category is. The considered categories are:

1. for people
   a) `http://schema.org/Person`
   b) `http://xmlns.com/foaf/0.1/Person`
   c) `http://dbpedia.org/ontology/Person`
   d) `http://dbpedia.org/class/yago/Person100007846`
2. for creators `http://dbpedia.org/class/yago/Person100007846`
3. for artists `http://dbpedia.org/class/yago/Artist109812338`
4. for musicians `http://dbpedia.org/class/yago/Musician110339966`
5. for composers `http://dbpedia.org/class/yago/Composer109947232`

3.3.3 Transformations

At this step we had all data to compute the link DBpedia and Doremus; however we had to perform a cleaning step on the Doremus years. We have noted two kinds of dirty data:

1. imprecise years: there are some years not very precise as "18..". We deduce that this probably means that we know the century in which the artist is born but not the specific year; we preserved this information because it could be useful also if it is more generic than a specific ;

2. years containing strings: 'ca1500', 'compositeur', 'ca 1720', 'ca 1400', 'ca 1750', 'ca 1430', 'ca1750', 'ca1761', 'ca 1580', 'ca 1736', 'ca1551.', 'ca 1555', 'fl 1658', 'ca 1500', 'ca 1700', 'ca 1516', 'ca 1550', 'ca 1760', 'ca 1706', 'vers 1550', 'ca 1570'; we deduced that "ca" and "vers" mean around so we preserved; however we deleted the "compositeur" case that it's probably a mistake.

3.3.4 Similarity measures

We defined a method to score the Doremus-DBpedia artist links. This criteria is composed by these similarity measures:

1. **The names/labels similarity**: to compute it we used the Python module "fuzzywuzzy". We deleted from the DBpedia artists candidates the ones that present a very low label similarity with the Doremus one.

2. **The years similarity**: for every pair Doremus artist-DBpedia artist we checked if the born year and the death year was the same. However the year information were not always present in both Doremus and DBpedia. So we can have different possibilities:
   a) if the complete information is present on both sides and fitted we assigned an high score to the possible DBpedia artist candidate;
   b) if the year information is present only on one side we assigned score 0 but we continued to consider the DBpedia entity as a possible candidate,because we haven't enough information to say that the 2 entities are not the same;
   c) if the information is present on both sides but it's different:we deleted the DBpedia artist from the candidates. Actually we didn't eliminate the candidate if the years differed for 1 or 2 years, because we have noted that, especially for the old artists, the dates are not always precise;
   d) if we had only the information about the century: if it matched we preserved the candidate assigning it a low score; alternatively we deleted it;
   e) if the birth year was present on both sides and fitted but the death year is not present on both sides, it could mean that the artist is not already death. To check that we basically looked at the birth date and, if it was not too far from the current date, we considered the artist as still alive, and we increased the score also for the death year matching basing the rise on the probably to

be alive computed through a Gaussian distribution. This score rise was less significant than when we had explicitly the same death year;

3. **The category similarity**: finally we checked the categories of the DBpedia entity: if it was a composer we assigned an high score, alternately we assigned a score as lower as the category is more general. The score assignment decreases following a Gaussian distribution. If the entity is not a person we assigned a negative score.

3.3.5 Aggregations

After having the four different similarity scores we computed a weighted sum of them to get the final matching score. Actually it was not a precise weighted sum; in fact we wanted to keep into account some particular combinations; for example if the category is too generic and the label score is too low, we delete the candidate also if the years matches. We have taken this decision after some trials, noting that considering also these candidates, the number of errors increases because it's more difficult to identify a good threshold to separate bad and good matchings.

TODO insert aggregation functions formula

3.3.6 Filtering

TODO insert graph

The graph above shows the scores distribution; the majority of matchings seems to have score near 1 that is good; however some noise it is still present; to remove it we cut all matchings with score lower that 0.5. 0.5 generally means that we don't have informations about the death and birth years but label matched perfectly and the DBpedia entity had an `rdf:type` that clearly indicates that it is a composer.

It is worth noting that a pre-filtering step was performed during the aggregation phase because, as we explained, we deleted for which the category is too generic and the label score is too low.

3.4 Entity Linking process (Wikidata)

Now we'll briefly describe the principal differences in the Wikidata artist Linking respect to DBpedia one. It's important to note that we used only the DBpedia artists to search paths between them and POIs, so we spent more time to improve the DBpedia linking respect to the Wikidata one. However we obtained also with less effort good results; in fact the Wikidata retrieved entities are less that the DBpedia on but the data are organized very well, so is easier to apply similarity metrics.

3.4.1 Retrieving Wikidata Entity

As anticipated at the beginning of this section, we'll not use the Elastic Search but we considered all Wikidata entities that are composers of some works, looking at a specific property that denotes the composition act. For each of these entities we performed a second query to get the birth and death year. In this case we didn't get the category information because we have already considered only the composers.

3.4.2 Transformations

The only data that we transformed were the Doremus one because they presented the same problems than before for some years strings.

3.5 Evaluation

The evaluation of the found links is done by comparing them with those provided by ISNI. ISNI is the ISO certified global standard number for identifying the millions of contributors to creative works. By achieving these goals the ISNI will act as a bridge identifier across multiple domains and become a critical component in Linked Data and Semantic Web applications.

3.5.1 DBpedia performances

ISNI matches are made by persons in the editorial world, so we can consider them as a ground truth that can be trusted. By comparing our matches we want to see if our performances are acceptable and if we can discover new links between ISNI numbers and DBpedia.

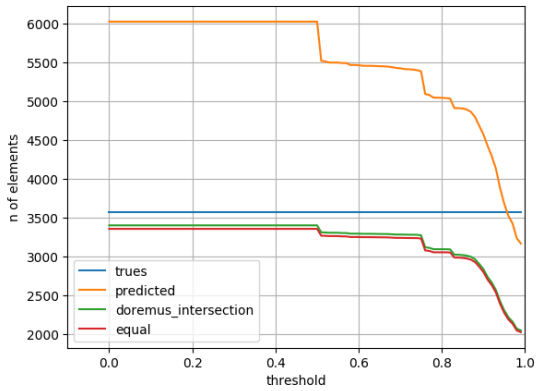


Figure 3: Counts of artist sets.

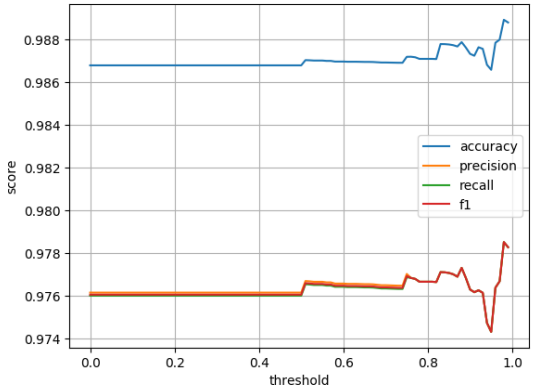


Figure 4: Metrics.

Since the matching confidence vary with the threshold, we can plot all the important counts and metrics respect to the threshold used. In the figure 3 and 4 there are all the counts of the important sets that describe our matching. Trues indicates the number of DOREMUS entities matched by ISNI, they are 3500 and of course they are independent from the threshold. Predicted indicates the number of DOREMUS entities matched by us, the number vary with the confidence, there are different drops that corresponds to crucial thresholds defined by our matching algorithm.

The most important thing is that their count goes from 3000 to 6000, this indicates that we are matching way more entities than ISNI. Doremus intersection represents the number of entities that are matched from both ISNI and us. This shows us that there is a part of the DOREMUS entities that ISNI matches and that we are not able to match.

Now, the following sets are all calculated from the intersection set, in fact we cannot compare the matching performances using entities that are not present in both sets. Equal show the number of correct matches, we see that this basically overlap with the intersection set, meaning that ISNI and our algorithm match a DOREMUS entity with the same DBpedia entity. Since ISNI matches have a professional validity, this means that we can have a great confidence in the matches produced by our algorithm.

Since equal and DOREMUS intersection are not exactly the same, it means that either our algorithm or ISNI are doing a wrong match. Looking more closely we saw that in some cases ISNI matches were wrong, especially in corner cases, where there were two persons with the same name. This corner cases can fool also an expert, and this is the reason why in our algorithm we embedded the comparison of birth-date and death-date in DBpedia with those present in Doremus. This allowed us to disambiguate between different persons with the same name. In other cases this was not enough, there are persons with the same name and birth/death dates, or whose dates are not available, that can be correctly classified looking at other fields, such as their profession.

Looking at the usual metrics, we see that they are extremely high for this kind of match. This means that we are doing an almost perfect match. We can state that the proposed method based on combinations of labels, classes and dates is able to match almost perfectly the artists of the two datasets. Moreover, we are able to find 2000 artists that at the time of writing are not present in the ISNI register.

3.5.2 Wikidata performances

todo: fabio

4 Path finder

4.1 Problem description

Once we have a direct mapping between entities that comes from specialized datasets and DBpedia, we actually need to link them. The whole problem can be solved by writing a series of queries that find paths of a given depth between two resources. The main problems that needs to be resolved are:

1. To write a query or a set of queries that finds all possible paths with given depth between two resources
2. Make sure that query response time scales well when the dataset id large and the depth is high.

4.2 Related work (Relfinder)

The path finding problem has already been solved in the past with Relfinder: a tool that is able to find all paths with a specified depth between two entities. The first problem to address is to write a set of queries that can find all paths that have the intermediate node in a fixed position:

$$a - - > b - - > c - - > d - - > e = m$$

$$a - - > b - - > c - - > d = m < - - e$$

$$a - - > b - - > c = m < - - d < - - e$$

$$\dots$$

These path are not able to model all possible paths, in fact we considered only paths with at most one direction change. When we consider paths with at most two direction changes, we can model more complex relations:

$$a - - > b - - > c - - > d - - > e = m$$

$$a - - > b - - > c - - > d = m < - - e$$

$$a - - > b < - - c = m - - > d - - > e$$

$$\dots$$

Relfinder is able to generate all paths with a given depths and with at most 2 direction changes. These paths can express most of the interesting relations between two entities, in fact when we consider more complex paths, we start loosing significance. For each entity pair, Relfinder generate all possible queries with at most the given depth and makes them, one by one. When the depth increases, the number of possible path shapes

increases too in a non linear way, and the whole process takes too time. The problem is that the resulting queries take too much time, and when we repeat this process for thousands of pairs, we incur in a scalability problem, the whole process would takes whole weeks.

4.3 Proposed approach

Our approach builds on top of the Relfinder, looking at the parts that slows the whole process. The problem is that DBpedia is small world and its average degree is 4.3, this means that on average we can reach every entity in at most 4-5 hops. Unfortunately our case does not make part of that average, and many times we need 5-6 hops to connect two entities. The problem is that when we make a query that looks for paths with depth 6 we are basically querying the whole DBpedia, which at the time of writing is composed of 8.8 billion triples. It is our belief that a query that tries to find a path between two entities, is converted by the SPARQL engine in what we can compare to a join for each hop. This, with the small world characteristic of DBpedia means that we are doing huge joins for each query. Another way to visualize the query is a Breath First Search that starts from the source entity, and it expands hop after hop until the specified depth is reached. In this way is even more easy to see why we end up by querying billions of triples.

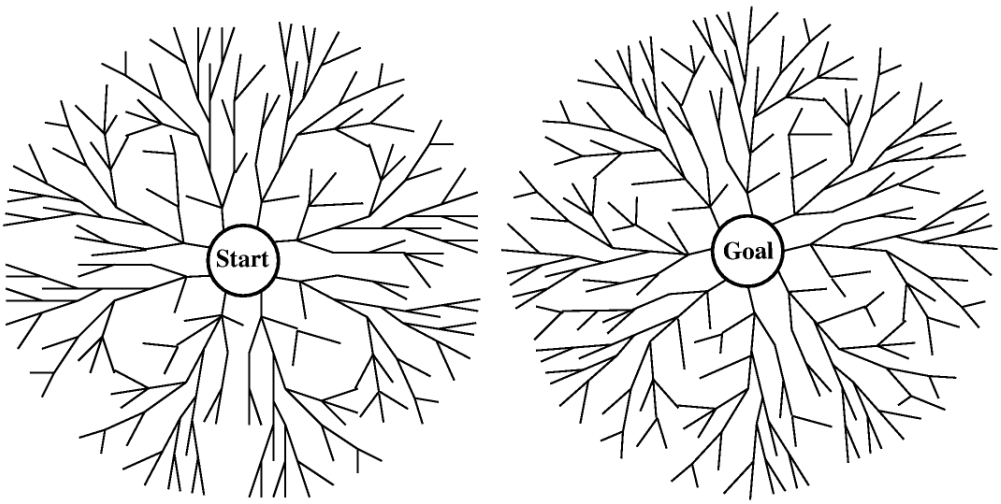


Figure 5: Bidirectional BFS

A common improvement of BFS is the two-way BFS, where the same algorithm is applied both at the source and the destination. In this way we can see that we can reach the same middle node by querying a way smaller portion of the graph. Looking at paths found by Relfinder, we have seen that usually one direction inversion cover most of the

cases, by limiting this we are actually simplifying a lot our problem. Then, we have seen that for artists and POIs, the connection is most of the times the result of a common node. This means that the direction must change at least one time. We reduced all the different cases to one. In fact what we do is to make two separate queries with depth from the source and from the destination. These queries return the list of all half-paths. We make a set of the involved nodes and then we intersect the two sets, in order to find the common nodes. At this point we operate joins that create full paths. The whole matching of POIs and Artists requires only some hours against the weeks that would take Relfinder.

4.4 Selection Algorithm

Now that we have all our path, we need to find the best ones, those that shows interesting links between POIs and Artists. todo: lorenzo Moreover, we find a good use of an equation proposed in another paper. todo. cite

At the end, we select 5 Artists for each POI looking at those artists with the highest score, where the score is given by their most interesting path.

4.5 Paths Evaluation

Evaluating the goodness of the found paths is not trivial, and an ideal approach would be to ask to real users for a feedback. What we can do is to analyze the found paths and see if they show interesting connections. The following path is generated in Place Masséna:

```
"http://dbpedia.org/resource/Chick_Corea",
"http://dbpedia.org/property/associatedActs",
"http://dbpedia.org/resource/Herbie_Hancock",
"http://dbpedia.org/ontology/instrument",
"http://dbpedia.org/resource/Fairlight_CMI",
"http://purl.org/dc/terms/subject",
"http://dbpedia.org/resource/Category:Music_technology",
"http://www.w3.org/2004/02/skos/core#broader",
"http://dbpedia.org/resource/Category:Musical_instruments",
"http://www.w3.org/2004/02/skos/core#broader",
"http://dbpedia.org/resource/Category:Musical_instrument_museums",
"http://purl.org/dc/terms/subject",
"http://dbpedia.org/resource/Palais_Lascaris"
```

This path can be considered quite interesting, in fact it is able to link the Palais Lascaris with an Artist because it played with Herbie Hancock, which used in his exhibitions the Fairlight CMI.

```
"http://dbpedia.org/resource/Aristide_Bruant",
"http://purl.org/dc/terms/subject",
"http://dbpedia.org/resource/Category:Nightclub_owners",
"http://www.w3.org/2004/02/skos/core#broader",
"http://dbpedia.org/resource/Category:Nightclubs",
"http://www.w3.org/2004/02/skos/core#broader",
"http://dbpedia.org/resource/Category:Music_venues",
"http://www.w3.org/2004/02/skos/core#broader",
"http://dbpedia.org/resource/Category:Music_venues_by_country",
"http://www.w3.org/2004/02/skos/core#broader",
"http://dbpedia.org/resource/Category:Music_venues_in_France",
"http://purl.org/dc/terms/subject",
"http://dbpedia.org/resource/Palais_Nikaia"
```

This path shows one of the defects of our path selection, in fact here we see that most of the relations are given by intermediate categories. This is not a bad thing by itself, but a long chain of categories usually means that the two entities on the edges will meet thanks to a very generic category. On the other hand, this connection is ranked as interesting, and it is partially true: how many artists are Nightclub owners?

Another example that shows that categories can be really explanatory is the following one:

```
"http://dbpedia.org/resource/Yannick_Noah",
"http://purl.org/dc/terms/subject",
"http://dbpedia.org/resource/Category:French_tennis_coaches",
"http://www.w3.org/2004/02/skos/core#broader",
"http://dbpedia.org/resource/Category:Tennis_in_France",
"http://www.w3.org/2004/02/skos/core#broader",
"http://dbpedia.org/resource/Category:Tennis_venues_in_France",
"http://purl.org/dc/terms/subject",
"http://dbpedia.org/resource/Nice_Lawn_Tennis_Club"
```

Here the connection is almost direct: Yannick Noah was a tennis coach and as expected he is linked to the Nice Lawn Tennis Club. What is clear from this examples is that paths are really heterogeneous and it is difficult to find a rule that discriminates good paths with good results.

5 Server Architecture

The server communicates with the client application using a REST API that consists of two endpoints:

1. POIs endpoint: allows to get all POIs

2. Playlist endpoint: allows to get the playlist ID associated to the current position

The whole server is implemented using the python Flask framework [9]. This extensible framework allows to develop REST APIs requiring a minimum amount of boilerplate code.

5.1 Database

The result of the ETL process can be interpreted as tables or collections in a database. Until now, we managed to link 3cixty POIs to DBpedia POIs and DOREMUS artists to DBpedia artists. Then, the result of the pathfinder was a list of paths that connects artists to POIs. Since we will use the Spotify API [10], the next step is the linking phase between each artist to the Spotify artist, and then from this to a set of Spotify tracks. At the end we want to have a direct connection between a DBpedia artist and his tracks on Spotify.

5.2 Spotify API usage

Spotify provides a public API that allows to perform all the operations that can usually be done using the Spotify application. These include artist and tracks search, CRUD operations on user playlists and the listening of the tracks' preview. All the above mentioned operation are essential for the usage of the application.

5.2.1 Authentication

Since the end user listen to a Spotify loaded dynamically by the application, there is no need for a client authentication. This is perfect because it allows to any user to use our application. Regarding the server, at startup it performs a token based authentication, where the server get possess of a private token used to perform all CRUD operations on the playlist catalog. It is important to note that for the correct functioning of the application, a Spotify developer account is needed, then the associated keys are needed in order to obtain the privileges to manage the accounts playlists.

5.2.2 Playlist

A playlist can be seen as an ordered list of tracks. With the correct privileges, it can be dynamically created. At creation time, it is required to give a name to the playlist, and the whole playlist object is returned. The returned objects contains a unique playlist

[9] Flask: http://flask.pocoo.org/
[10] Spotify: https://www.spotify.com/

ID and many other information that we don't need. Actually, the only information we need is the playlist ID that we can use to access it in the future. It is important to note that we can access to the playlist in two ways: from the server we can access it using the playlist name, in fact we will make sure during the playlist name generation to have unique names. This will be useful to check the presence of a playlist with a given name. From the client point of view, the only access possible is through the playlist ID, in fact it is unique over all the Spotify playlists.

5.2.3 Artist

Artists on Spotify can be searched by name and usually, a simple search by name gives as a result different artists with the name that matches part of the search string. What we need is to link a Doremus artist to a Spotify artist, but in this case we cannot build a proper linking pipeline because the Spotify API is like a black box. What we did is to test it against different kind of inputs with different kind of noises, such as non alphabetic characters, name repetition, the specification of the genre or the word 'composer'. What we have found is that the Spotify search API is quite robust regarding noise determined by name repetition and non alphabetic characters, while it is sensitive to additional words. This means that in order to get the correct artist we need to provide name and surname of the artist. These information are the key component of the label of DOREMUS artists. Since there is a one to one correspondence between DBpedia artists and DOREMUS artists, we can find name and surname instantly. At this point we can perform the API search request, and it returns a list of artist objects. Each object contains different informations regarding each artist, such as his popularity and his genres. Here we use only the popularity by taking the most popular one, but in the future we could use the genre information to perform a better disambiguation.

5.2.4 Tracks

Tracks are the main object of Spotify and the API gives us different ways to search for them. A possibility is to search them by artist. We can either get the tracks of a specific album of an artist, or the most popular tracks of an artist. In this project we use the second way because it gives us a number of advantages:

1. It allows us to get the most popular tracks, this is useful because the user can directly connect to a popular tracks, while this could be more difficult with non popular tracks

2. It allows us to filter outliers, in fact the artist search is not perfect and we can get an artist different from the one we looked for. These are usually outliers or non popular artists for which there are no popular songs. We can use this peculiarity

to filter these unwanted artists. An artist with less than 10 popular tracks will be filtered.

But what is popularity? Unfortunately we do not know how Spotify computes it, but we know that it is based on the playcount and recentness of a track. Among all properties that characterize a track, we only need the name and the ID. The name is required for simple visualization purposes, while the track ID is essential for what we want to do: add tracks to a playlist. This operation can be done with an API call, and it requires the playlist ID and the track IDs.

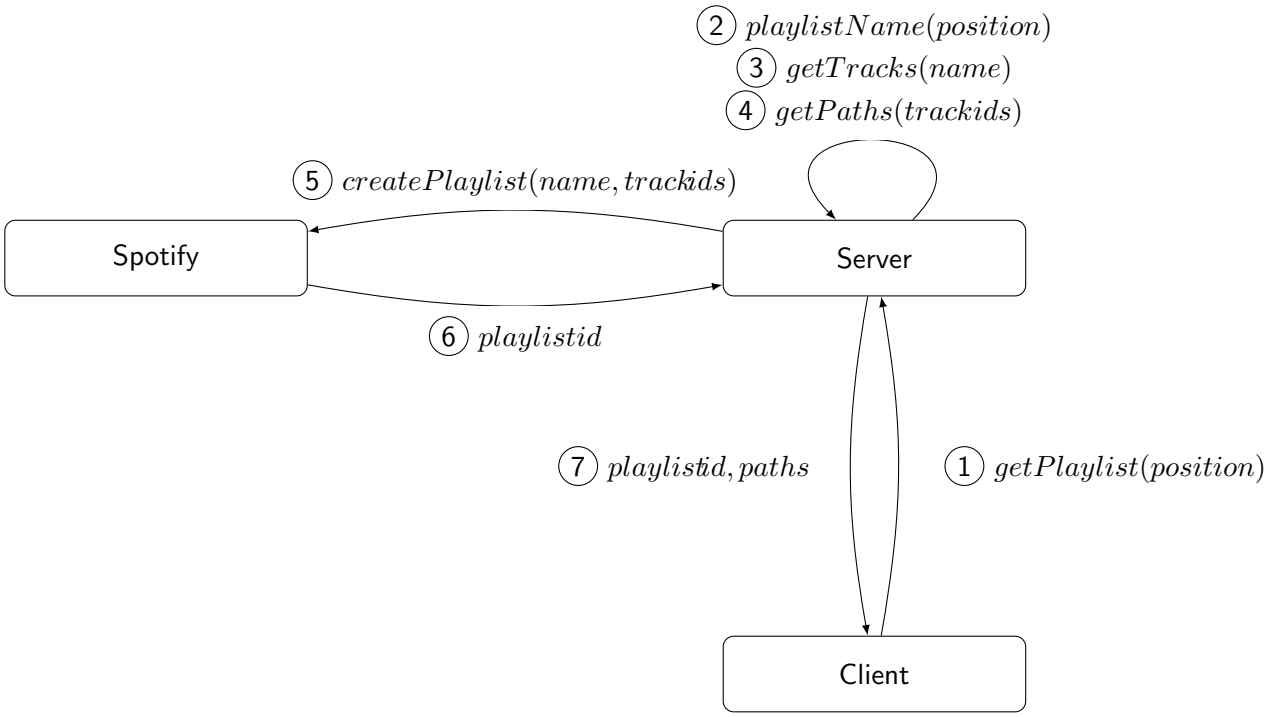
5.3 POIs Endpoint

The first endpoint that is called by the client is the POIs endpoint. This endpoint allows to return all POIs in the database that then are drawn on the application map view.

5.4 Playlist Endpoint

The main endpoint is the one that exposes the tracks and the paths related to a specific geoposition. The whole process can be summarized in:

1. Take user position and validate it.
2. Calculate nearest POIs and assign weights, from these create a unique playlist name.
3. Select the tracks for each artist linked with the poi.
4. Select the paths associated to each track.
5. Create a playlist if it does not exist and add previously selected tracks.
6. Save the playlist ID and attach to each track the path between the poi and the track's artist.
7. Return the playlist ID and the track-path mappings as a JSON.



5.4.1 Nearest POIs selection algorithm

Once the client position is validated, it is necessary to select the POIs that will be used for the recommendation. In fact, it is not possible to directly recommend tracks based on positions because otherwise we would have a playlist for each possible coordinates pair. The following operations needs to be done:

1. Calculate distance from the each POI to the user position. Since we are using coordinates, the distance is calculated using the haversine distance, which is very accurate and fast.

2. Add to the list the nearest POI, regardless of the distance.

3. Add the remaining two nearest POIs if their distance is inferior of 1 km.

4. For each element in the list, add 10 meters to the distance and then take the log2. By adding 10 meters we are adding a bias to all our distances, in this way positions too near to us are not over-weighted respect far distances. Then, the log2 squashes distances until 1 km to a restricted range of values.

5. Calculate repartition weights using the following equation:

$$w_i = \left\lfloor 10 \frac{\frac{1}{\log_2(d_i + 10)}}{\sum \frac{1}{\log_2(d_i + 10)}} \right\rceil$$

   In this way we obtain normalized weights from 0 to 10 that are inversely proportional to the log base 2 of the distance.

At this point we have a direct mapping between POIs weights and a position

5.4.2 Artist' tracks selection algorithm

In order to understand the link between the recommended tracks and the POIs, we need to go deeper in the process that links a POI with his tracks. For each artist there is a direct mapping with its tracks. The problem is: given a playlist name, find the correct tracks. The playlist name encodes the weights of each poi, and each weight express exactly the number of tracks that must be chosen for the POI associated. Since each POI is linked to different artists, we need a way to select songs from them in a balanced way. What we do is to cycle between all the artists to take each time the most important song that is not already present in the playlist. In this way we obtain different songs from different artists, and since the artist and songs are static, the tracks in the playlist will be always the same. In the future we think to add randomization to the whole selection process.

5.4.3 Response generation

1. Create a playlist with the generated playlist name if it does not exist and add previously selected tracks.
2. Attach to each track the path between the poi and the track's artist.
3. Return the playlist ID and the track-path mappings as a JSON.

6 Mobile Web Application

The Web application is implemented using AngularJS and its common MVVM model. Using the Angular framework we are able to create complex single page applications that reduces to the minimum the client server interactions. Another advantage of AngularJS is his widespread adoption and the presence of numerous libraries that communicates naturally with it. One of those is the Google Map module that we use to handle all the POI navigation.

6.0.1 Music Player

The music player is implemented using the Spotify Play Button. This is a special ¡iframe¿ that loads a simplified Spotify player for a specified playlist. This solution is extremely convenient because it simply needs a playlist ID to run. Since the playlists are created server side and are recovered using the REST API, the whole process is very simple. Unfortunately it is not possible to manipulate programmatically the ¡iframe¿ and this limit the whole user experience. As an improvement, we could build a custom player that still uses the Spotify API.

6.1 Google Map API

6.1.1 POIs placement

todo: Lorenzo

6.1.2 Navigation

6.2 Relation visualization (and Path format)

todo: Lorenzo

7 Conclusion and Future Work

todo: fabio-lorenzo, improve In this work, we presented CityMusic: a context-aware content-based recommender system for artists. Regarding the first part: the interlinking, we showed that is possible to link POIs and artists to DBpedia achieving good metrics.

We worked on the Relfinder algorithm, by making it scalable. We think that having a scalable solution for the relation finding is crucial for any project that aims to use heavily the huge knowledge graph that DBpedia is.

Our future plans are:

1. Improve the relation finding algorithm allowing it to find more complex relations without affecting performances.
2. Improve the relation selection algorithm: this part needs a big rework because in the selected paths, there are some that are not meaningful for the user.
3. Evaluate the overall approach with real users
4. Use additional context informations available in 3cixty and DOREMUS. In this optic, 3cixty provides information about events and this could be added in the recommendation engine, maybe suggesting songs or artists that will be played in a place in the near future.

5. Integrate the music context in the recommendation: now we recommend popular songs because we are not able to choose among them based on the current context. By linking DOREMUS songs with DBpedia we hope to extend the semantic recommendation taking into account song-specific relations with a POI. Linking songs to DBpedia is hard, but we showed that is possible, the main problem is to find good relations between POIs and Songs. This operation is even harder than linking artists to POIs because songs on DBpedia are very poor regarding of links with other resources.

References

[1] Peter Knees and Markus Schedl. A survey of music similarity and recommendation from music context data. *ACM Trans. Multimedia Comput. Commun. Appl.*, 10(1):2:1–2:21, December 2013.

[2] Marius Kaminskas, Francesco Ricci, and Markus Schedl. Location-aware music recommendation using auto-tagging and hybrid matching. In *Proceedings of the 7th ACM Conference on Recommender Systems*, RecSys '13, pages 17–24, New York, NY, USA, 2013. ACM.

[3] Marius Kaminskas, Ignacio Fernández-Tobías, Francesco Ricci, and Iván Cantador. Knowledge-based music retrieval for places of interest. In *Proceedings of the Second International ACM Workshop on Music Information Retrieval with User-centered and Multimodal Strategies*, MIRUM '12, pages 19–24, New York, NY, USA, 2012. ACM.

[4] Philipp Heim, Sebastian Hellmann, Jens Lehmann, Steffen Lohmann, and Timo Stegemann. Relfinder: Revealing relationships in rdf knowledge bases. In *Proceedings of the 4th International Conference on Semantic and Digital Media Technologies: Semantic Multimedia*, SAMT '09, pages 182–187, Berlin, Heidelberg, 2009. Springer-Verlag.