Homework 3 Report

Matheus Schmitz | USC ID: 5039286453

Task 1.1

Both of my datasets, which are movie datasets (IMDB and TMD), contain the exact same attributes, which are: name, year, director(s), writer(s), and actor(s).
In choosing my attributes for record linkage I decided to opt for those which were the most likely to be present in the samples and to be correct, in other words, I opted for the most salient movie attributes, which are name, year and directors.
One important caveat is that both dataset posses missing values. This raises the need to handling the missing values with the following three limitations: (1) RLTK requires all attributes to be strings, (2) RLTK cannot adequately handle missing values, and (3) the ensuing steps will employ string matching.
Accounting for those limitations I've encoded all missing values as the following string "<___>". This sting was chosen for three reasons: (1) it does not contain words, avoiding erroneous matches during string match, (2) is has only 5 characters, which means that I'll be penalized by the penalty function in my string matcher which subtracts score from strings of length 6 or less, and (3) it does not contain numbers, ensuring all years with missing data will raise an error in a try-except function, allowing the custom treatment of missing year data by the except part of the function.
In addition to the missing values in the loaded rows, the dataset also contain unreadable rows, which have either more or less elements than expected by the csv reader, thus it was necessary to load the data using pandas.read_csv with the parameter *error_bad_lines=False* which allows the problematic rows to simply be ignored and not included in the data.

Task 1.2

The first step of my blocking technique consisted in creating the tokens to be used in blocking. For that I have employed the following feature engineering pipeline:

1) Merge movie name and list of directors into a single string.
2) Remove certain common stopwords.
3) Tokenize the reduced string.

The reasoning for step #1 is that I want to ensure that if a datapoint (movie record) is missing either its name or director, that the datapoint can nevertheless still be blocked with its matching pair.

The usage of stopword removal in step #2 is due to the high frequency of certain terms (eg: "the", "a" "of") in movie names, which if kept will result in a subpar reduction ratio by the blocking algorithm. Since movie names will have more than only the removed stopwords on their titles, plus the fact that the names of directors are also when tokenizing, makes it possible to achieve a state in which removing the stopwords will improve the reduction ratio without hampering pairs completeness. Certain stopwords such as "yes" or "no" have been kept as they are not frequent in movie names meaning that keeping them does not worsen reduction ratio by much yet avoids unintended side-effects on pairs completeness.

Finally step #3 is simply applying RLTK's CrfTokenizer() to convert the derived string into tokens.

After the tokens were obtained, blocking them was done utilizing RLTK's TokenBlockGenerator().

Upon evaluation of the blocking technique employed, I have obtained the following scores:

$$Reduction\ Ratio:\ 0.9648$$

$$Pairs\ Completness:\ 1.00$$

Task 1.3

Field Similarities

For string similarity comparison, I have decided to stick to the same set of three most salient features: movie name, year and director(s).

For calculating movie name similarities I've used ratio of matching substrings, as defined by python's difflib.SequenceMatcher.ratio() (https://docs.python.org/3/library/difflib.html#difflib.SequenceMatcher.ratio). This metric is obtained by the following formula:

$$ratio = \frac{2 * matches}{len(string1) + len(string2)}$$

This metric has value 1 if both strings are identical and value 0 if they are completely different. Further clarifications on the metric can be found at: https://stackoverflow.com/questions/35517353/how-does-pythons-sequencematcher-work.

The reasoning for this choice instead of the popular "longest substring" was made entirely based on experimental results which showed it to achieve better performance across various assessment metrics (performance, recall, f1-score, etc.). Moreso, experimentation also shower two "types" of movie titles to be frequently resulting in false-positive matches: (1) movie sequels (eg: "Ice Age" x "Ice Age 2"), and (2) movies with short titles ("Mad Max" x "Mad Man"), thus I've included a penalization factor in the similarity scores, which multiplies the similarity score by 0.9 to the power of x={0,4}, where x increases by 1 each time one of the two movie titles being compared matches either a sequel or a short name.

The second similarity computation looks at director names, which are concatenated into a single string when there is more than one director for a given movie. This string is then converted into a similarity score using the same ratio metric described in the paragraph above. There is no penalization factor for director name similarity.

Lastly the similarity comparison looks at release dates (year) for both movies, and calculates a score by using the following formula:

$$similarity = \frac{1}{(1 + abs(year_{movie_1} - year_{movie2})}$$

If both movies have the same year the resulting score is 1, if there is a one-year difference, the score is 0.5, for a two-year difference the score is 0.333, and so on.

If any of the movies is missing the year attribute this comparison returns a None, which is used to adjust the weights on the combinatory function.

Combinatory Function

The three scores mentioned above are combined using the following function:

```
if sim_year != None:
    element_similarity = ((4/6) * sim_name) + ((1/6) * sim_director) + ((1/6) * sim_year)
else:
    element_similarity = ((4.5/6) * sim_name) + ((1.5/6) * sim_director)
```

Implicit in the function above is the fact that the weights will vary depending on whether it was possible to calculate the similarity between years. If it was possible, then 4/6 of the weight is allocated to the name and 1/6 to each of directors and year. If it was not possible then 4.5/6 of the weight is allocated to name and 1.5/6 to directors (in other words year's weight is evenly split among the other variables.

The choice of function was made with consideration to the fact that many more movies will match in year or director, meaning that those a not as valuable in identifying pairs as is name, and thus it is reasonable to give higher weightage to the similarity score derived from movie names, which are much more unique. Both year and directors receive the same weight as I found no strong argument for favoring one over the other.

Model Evaluation

In addition to developing a similarity scoring function, the definition as to whether a candidate pair of movies represents a single movie or not is also dependent the threshold chosen for arbitrating that the score represents a match.

As an improvement over a naive random guessing of a threshold, I have employed a grid search that scans over various threshold levels in increments of 0.05 (5%), generating multiple assessment metrics related to each threshold. The best threshold is then chosen based on the highest F1-Score obtained. The table below contains the results from the grid search over threshold values.

	TP	FP	TN	FN	TPR	FPR	TNR	FNR	Precision	Recall	F1-Score
0.00	256.0	144.0	0.0	0.0	1.000000	1.000000	0.000000	0.000000	0.640000	1.000000	0.780488
0.05	256.0	144.0	0.0	0.0	1.000000	1.000000	0.000000	0.000000	0.640000	1.000000	0.780488
0.10	256.0	144.0	0.0	0.0	1.000000	1.000000	0.000000	0.000000	0.640000	1.000000	0.780488
0.15	256.0	144.0	0.0	0.0	1.000000	1.000000	0.000000	0.000000	0.640000	1.000000	0.780488
0.20	256.0	144.0	0.0	0.0	1.000000	1.000000	0.000000	0.000000	0.640000	1.000000	0.780488
0.25	256.0	144.0	0.0	0.0	1.000000	1.000000	0.000000	0.000000	0.640000	1.000000	0.780488
0.30	256.0	144.0	0.0	0.0	1.000000	1.000000	0.000000	0.000000	0.640000	1.000000	0.780488
0.35	256.0	136.0	8.0	0.0	1.000000	0.944444	0.055556	0.000000	0.653061	1.000000	0.790123
0.40	256.0	114.0	30.0	0.0	1.000000	0.791667	0.208333	0.000000	0.691892	1.000000	0.817891
0.45	256.0	82.0	62.0	0.0	1.000000	0.569444	0.430556	0.000000	0.757396	1.000000	0.861953
0.50	256.0	43.0	101.0	0.0	1.000000	0.298611	0.701389	0.000000	0.856187	1.000000	0.922523
0.55	256.0	25.0	119.0	0.0	1.000000	0.173611	0.826389	0.000000	0.911032	1.000000	0.953445
0.60	256.0	16.0	128.0	0.0	1.000000	0.111111	0.888889	0.000000	0.941176	1.000000	0.969697
0.65	256.0	7.0	137.0	0.0	1.000000	0.048611	0.951389	0.000000	0.973384	1.000000	0.986513
0.70	256.0	7.0	137.0	0.0	1.000000	0.048611	0.951389	0.000000	0.973384	1.000000	0.986513
0.75	256.0	6.0	138.0	0.0	1.000000	0.041667	0.958333	0.000000	0.977099	1.000000	0.988417
0.80	254.0	3.0	141.0	2.0	0.992188	0.020833	0.979167	0.007812	0.988327	0.992188	0.990253
0.85	254.0	1.0	143.0	2.0	0.992188	0.006944	0.993056	0.007812	0.996078	0.992188	0.994129
0.90	222.0	1.0	143.0	34.0	0.867188	0.006944	0.993056	0.132812	0.995516	0.867188	0.926931
0.95	207.0	0.0	144.0	49.0	0.808594	0.000000	1.000000	0.191406	1.000000	0.808594	0.894168
0.99	202.0	0.0	144.0	54.0	0.789062	0.000000	1.000000	0.210938	1.000000	0.789062	0.882096

From the above table it can see that the best threshold is 0.85. The metrics associated with this threshold are:

$$Precision = 0.9961$$
$$Recall = 0.9922$$
$$F1\ Score = 0.9941$$

Task 2.1

The model I have developed to generate the RDF data required no custom classes as all the necessary ones were available at schema.org.

The model defines a class Movie over my custom namespace (my_ns:Movie), which inherits from schema.org's default Movie class (rdfs:subClassOf schema:Movie).

Additionally the custom class specifies literals for all attributes resulting from the merge, with the movie name being defined as schema:txt, the movie's release year as xsd:gYear (a class that takes only the 4-

digit representation of a year in the Gregorian calendar), and the remaining attributes (directors, writers and actors) being defined as schema:Person.

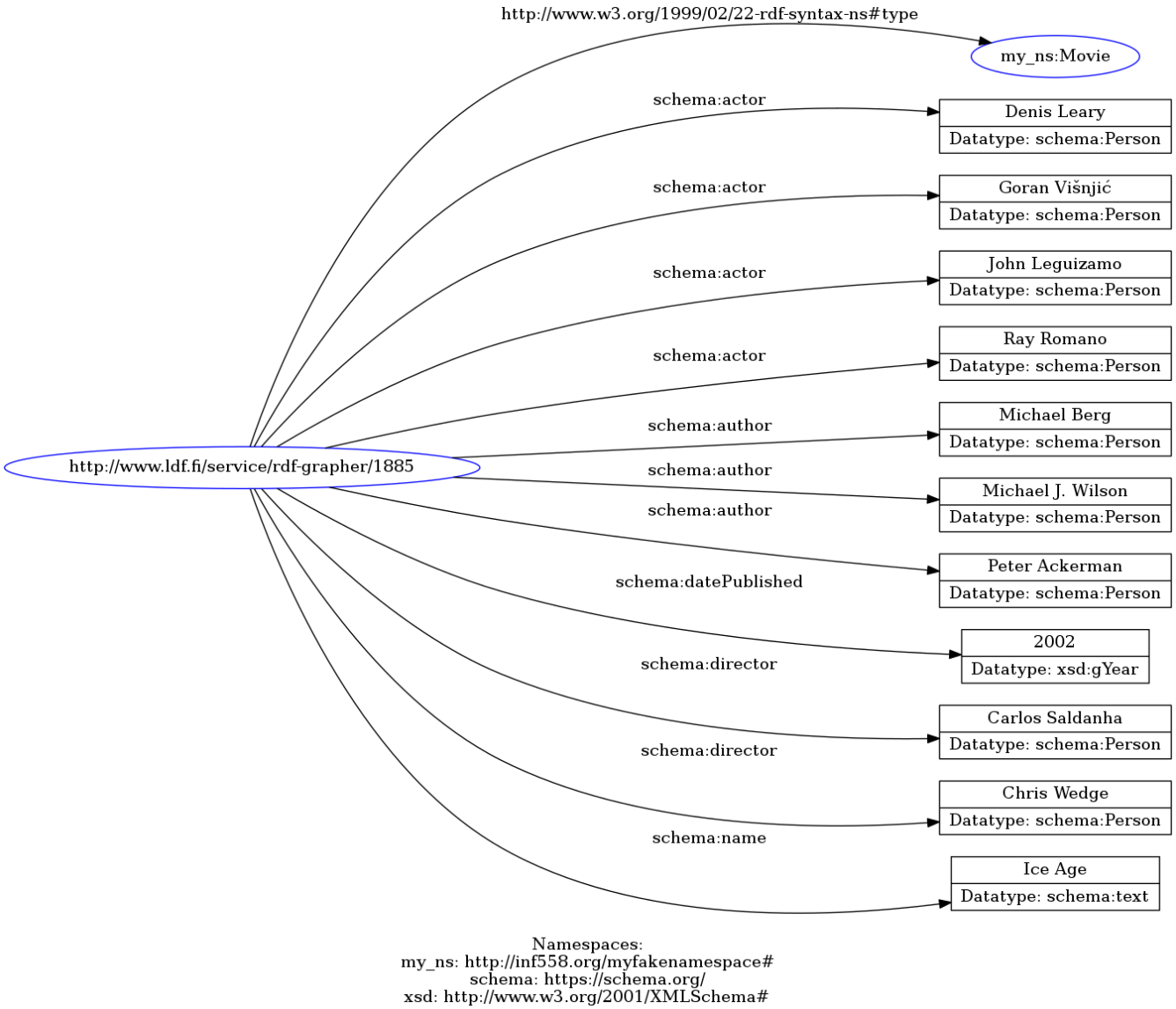
```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix schema: <http://schema.org/> .
@prefix my_ns: <http://inf558.org/myfakenamespace#> .


#### Movie Class ####
my_ns:Movie a schema:Class ;
    rdfs:subClassOf schema:Movie ;
    schema:name schema:text ;         # movie name
    schema:datePublished xsd:gYear ; # year of release
    schema:director schema:Person ;  # movie directors
    schema:author schema:Person ;    # movie writers
    schema:actor schema:Person ;     # movie actors
```

Task 2.2

```
# Populate the RDF with predictions with a positive (1) label
for idx, row in tqdm(predictions_df[predictions_df['LABEL']==1].iterrows(),
                     total=predictions_df[predictions_df['LABEL']==1].shape[0]):

    # URI
    node_uri = URIRef(str(row['IMDB_ID']))
    g.add((node_uri, RDF.type, MYNS.Movie))

    # Name
    name_imdb = str(df_imdb[df_imdb['ID'] == str(row['IMDB_ID'])]['name'].values[0])
    name_tmd = str(df_tmd[df_tmd['ID'] == str(row['TMD_ID'])]['title'].values[0])
    name = name_imdb if name_imdb != '<___>' else name_tmd if name_tmd != '<___>' else None
    g.add((node_uri, SCHEMA.name, Literal(name, datatype=SCHEMA.text)))

    # Year
    year_imdb = str(df_imdb[df_imdb['ID'] == str(row['IMDB_ID'])]['year'].values[0])
    year_tmd = str(df_tmd[df_tmd['ID'] == str(row['TMD_ID'])]['year'].values[0])
    year = int(float(year_imdb)) if year_imdb != '<___>' else int(float(year_tmd)) if year_tmd != '<___>' else None
    g.add((node_uri, SCHEMA.datePublished, Literal(year, datatype=XSD.gYear)))

    # Director(s)
    director_imdb = df_imdb[df_imdb['ID'] == str(row['IMDB_ID'])]['director'].values[0].split(';')
    director_imdb = [name.strip() for name in director_imdb if name != '<___>']
    director_tmd = df_tmd[df_tmd['ID'] == str(row['TMD_ID'])]['director(s)'].values[0].split(';')
    director_tmd = [name.strip() for name in director_tmd if name != '<___>']
    directors = list(set(director_imdb + director_tmd))
    [g.add((node_uri, SCHEMA.director, Literal(director, datatype=SCHEMA.Person))) for director in directors]

    # Writer(s)
    writers_imdb = df_imdb[df_imdb['ID'] == str(row['IMDB_ID'])]['writers'].values[0].split(';')
    writers_imdb = [name.strip() for name in writers_imdb if name != '<___>']
    writers_tmd = df_tmd[df_tmd['ID'] == str(row['TMD_ID'])]['writer(s)'].values[0].split(';')
    writers_tmd = [name.strip() for name in writers_tmd if name != '<___>']
    writers = list(set(writers_imdb + writers_tmd))
    [g.add((node_uri, SCHEMA.author, Literal(writer, datatype=SCHEMA.Person))) for writer in writers]

    # Actor(s)
    actors_imdb = df_imdb[df_imdb['ID'] == str(row['IMDB_ID'])]['actors'].values[0].split(';')
    actors_imdb = [name.strip() for name in actors_imdb if name != '<___>']
    actors_tmd = df_tmd[df_tmd['ID'] == str(row['TMD_ID'])]['actor(s)'].values[0].split(';')
    actors_tmd = [name.strip() for name in actors_tmd if name != '<___>']
    actors = list(set(actors_imdb + actors_tmd))
    [g.add((node_uri, SCHEMA.actor, Literal(actor, datatype=SCHEMA.Person))) for actor in actors]

# Save to disk using turtle format
g.serialize('Matheus_Schmitz_hw03_triples.ttl.', format="turtle")
```

```
100%|██████████████████████████████████████████████████████████████████████| 27506/27506 [06:27<00:00, 70.94it/s]
```

Task 2.3


http://www.w3.org/1999/02/22-rdf-syntax-ns#type
my_ns:Movie
schema:actor
Denis Leary
Datatype: schema:Person
schema:actor
Goran Višnjić
Datatype: schema:Person
schema:actor
John Leguizamo
Datatype: schema:Person
schema:actor
Ray Romano
Datatype: schema:Person
schema:author
Michael Berg
Datatype: schema:Person
schema:author
Michael J. Wilson
Datatype: schema:Person
schema:author
Peter Ackerman
Datatype: schema:Person
schema:datePublished
2002
Datatype: xsd:gYear
schema:director
Carlos Saldanha
Datatype: schema:Person
schema:director
Chris Wedge
Datatype: schema:Person
schema:name
Ice Age
Datatype: schema:text
http://www.ldf.fi/service/rdf-grapher/1885
Namespaces:
my_ns: http://inf558.org/myfakenamespace#
schema: https://schema.org/
xsd: http://www.w3.org/2001/XMLSchema#