1 Emotional Reasoning Chat Bot

1.1 1 - Introduction

The goal of the proseminar is to create a chatbot that is able to perform emotional reasoning.

> Emotional reasoning is a cognitive process by which a person concludes that his/her emotional reaction proves something is true, regardless of the observed evidence.
>
> -- *Wikipedia*

Reasoning processes of humans are mainly different than reasoning principles of machines, as humans made decisions basing not only on logics and facts, but basing on emotions, biases and comparatively very limited information.

This project will collect and design ontologies and data representing human emotions and design reasoning mechanisms employing these. The reasoning mechanisms can be inbuilt in practical scenarios e.g. in online marketing - designing of a chatbot that appeals to emotions of humans and trying to change their behaviour. Or a character that goes beyond the "personal assistant" mode.

1.2 2 - Domain Overview

- Linked Open Vocabularies.
- Google dataset search.
- CKAN (datasets search).

1.2.1 Movies and their Non-numerical reviews for Deducing Emotions

- A small set of 16 movies
- A large set of movies usually used for binary sentiment classification (positive|negative)
- Another set of 25,000 movies also for binary sentiment classsification

1.3 3 - Initial Vocabulary Selection and Domain Specification

When it comes to vocabularies regarding emotions, there are a couple of choices. In the following segment we briefly discuss the vocabularies we looked at and why we landed on *Onyx* in combination with *WNAffect*.

1.3.1 Emotion Markup Language (EmotionML)

EmotionML

EmotionML can be used in three different areas: manual annotation of data, automatic recognition of emotion-related states from user behavior and generation of emotion-related system behavior. While the last two points may sound well suited for our project, we found that *Onyx* was better suited for extracting emotions from text.

1.3.2 Human Emotions Onotology (HEO)

Human Emotions Ontology (HEO)

HEO was created in order to annotate multimedia with emotions. Again, this sounds perfect for our project, but *Onyx* is still better suited when it comes to emotions extracted from text.

1.3.3 Human Stress Ontology (HSO)

Human Stress Ontology (HSO)

As its name already gives away, HSO is specifically designed to model stress factors and relations between said factors and their countermeasures. While HSO would be great for a project where movies are suggested in order to lower a user's stress level, it is not well suited for our project.

1.3.4 Linked Data Models for Emotion and Sentiment Analysis W3C Community Group

Linked Data Models for Emotion and Sentiment Analysis W3C Community Group

While the ideas of the W3C Community regarding modeling emotions sound great, the community does not seem to be very active.

1.3.5 Onyx - An Emotion Modelling Ontology

- Specification
- Onyx: A Linked Data Approach to Emotion Representation

Onyx was designed for emotions which have been extracted from text. This suits our needs perfectly, since we want to extract the user's emotional state from their chat messages and the emotional response to a movie from the movie's reviews.

1.3.6 WordNet-Affect Taxonomy

WordNet-Affect Taxonomy

WNAffect was desigend to link words to affects (emotions). Again this makes a lot of sense for us and *Onyx* uses the terminology of WNAffect.

After deciding on how to model emotions, we did a mockup of what our knowledge graph might look like:

1.3.7 Our Knowledge Graph

A first glance of our graph model.

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns# >.
@prefix onyx: <http://gsi.dit.upm.es/ontologies/onyx/ns>.
@prefix wna: <http://gsi.dit.upm.es/ontologies/wnaffect/ns>.

mcb:User a rdfs:Class;
    rdfs:comment "A user that is chatting with the bot";
    rdfs:label "User";
    onyx:hasEmotion [
        onyx:hasEmotionCategory wna:anger;
        onyx:hasEmotionIntensity :1.0;
    ];
    mcb:hasMovieHistory [
        schema:Movie mcb:pulp_fiction;
        schema:Movie mcb:deadpool;
    ];

mcb:Movie a rdfs:Class;
    rdfs:comment "A Movie";
    rdfs:label "Movie";
    mcb:hasRelatedEmotions [
        onyx:Emotion onyx:Sad;
        onyx:Emotion onyx:Thrilled;
    ];

mcb:hasRelatedEmotions a rdf:Property;
    rdfs:comment "Movies emotions related to them";
    rdfs:label "hasRelatedEmotions";
    rdfs:type rdf:Bag;
    rdfs:domain mcb:Movie;

mcb:hasMovieHistory a rdf:Property;
    rdfs:comment "A users history of watched movies";
    rdfs:label "MovieHistory";
    rdfs:type rdf:Bag;
    rdfs:domain mcb:User;
```

The prefix *mcb* stands for "movie chat bot" which is a placeholder for the parts which we wanted to model ourselves. However, since there are a lot of vocabularies for movies, it would have been senseless to create our own which is why we used *schema.org/Movie* in the final approach. This ontology covers pretty much every property of movies imaginable which is nice, but also too much for our project which is why we limited it to identifier, name, duration, release (dateCreated), description (text), aggregated rating (aggregateRating), actors, characters and poster (image).

1.3.8 Our Crawler

Since we did not find a dataset including movies and their reviews, we had to crawl some review sites by ourselves. The crawler is written in javascript and works by looking for specified *html tags* in the page using *jQuery*. In order to make it suitable for multiple data sources/review sites, the crawler is initialized with a config via *config.json*, with tags via *movies.json* and *reviews.json* and with a list of targets via *target.json*.

The results for movies are stored in seperate jsons which are linked by the movies id. Later these jsons can be processed by our mapper in order to map the metadata to our ontology.

1.3.8.1 config.json

- target_entities (movies and reviews)
  - base: the base url of the movie review website
  - postfix: the sub path of the url which comes after the movie id
  - save_dir: the directory where the result jsons are going to be stored
- already_used_targets: used to keep track of retrieved movies
- to_be_used_targets: used to keep track of movies going to be fetched
- targets: list of movies which will be retrieved be the crawler when run
- targets_dir: directory of the target jsons

Example:

```
{
    "target_entities":{
        "movies":{
            "base":"https://www.imdb.com/title/",
            "postfix":"",
            "save_dir":"../../data/imdb/movies/",
            "tags":"./tags/movies.json"
        },
        "reviews":{
            "base":"https://www.imdb.com/title/",
            "postfix":"/reviews?ref_=tt_urv",
            "save_dir":"../../data/imdb/reviews/",
            "tags":"./tags/reviews.json"
        }
    },
    "already_used_targets":[],
    "to_be_used_targets":[],
    "targets":["starwars.json"],
    "targets_dir":"./targets/"
}
```

1.3.8.2 movies.json and reviews.json

- singular: items where a quantity of exactly 1 is expected
    - get_text: retrieve the text of a html tag
    - get_attr: retrieve the attribute of a html tag
- plural: items where a quantity of >1 is expected
    - get_text: retrieve the text of a html tag
    - get_attr: retrieve the attribute of a html tag

Example:

```
{
    "singular":{
        "get_text":{
            "movie-duration":"div.subtext time",
            "movie-rating-count":"span[itemprop='ratingCount']",
            "movie-rating-value":"span[itemprop='ratingValue']",
            "movie-best-rating":"span[itemprop='bestRating']",
            "movie-year":"span#titleYear a",
            "movie-director":"div.credit_summary_item a",
            "movie-description":"div.inline.canwrap p span"
        },
        "get_attr":{
            "movie-title":["meta[property='og:title']","content"],
            "movie-image":["meta[property='og:image']","content"],
            "movie-short-description":["meta[name='description']","content"],
            "movie-url":["meta[property='og:url']","content"],
            "movie-id":["meta[property='pageId']","content"]
        }
    },
    "plural":{
        "get_text":{
            "movie-character":"td.character a"
        },
        "get_attr":{
            "movie-actor-img":["td.primary_photo a img.loadlate","src"],
            "movie-actor":["td.primary_photo a img","alt"]
        }
    }
}
```

1.3.8.3 targets.json

A simple array containg movie ids (Example: starwars.json contains the ids of all Star Wars movies).

1.3.9 Our Mapper

The mapper takes the jsons created by the crawler and puts all of the data into our knowledge graph. This process is actually very simple since the jsons contain simple dictionaries (key value pairs).

1.3.9.1 Movies

1. Iterate over all actors and create an instance of *schema:Person* for each actor
2. Create a *schema:aggregateRating* containing the number of ratings, the rating and the max. rating
3. Create an instance of *schema:Movie* and add all of the crawled metadata to it

1.3.9.2 Reviews

1. Create an entity of the review and add all of the crawled metadata to it
2. Create an *onyx:EmotionSet* which links an *wnaffect* emotion to the review
3. Create an analysis which describes how the *onyx:EmotionSet* was obtained (currently this is done randomly)

Example (sub graph of our knowledge graph):

```
@base <http://movie.chatbot.org/>.
@prefix schema: <http://schema.org/>.
@prefix onyx: <http://www.gsi.dit.upm.es/ontologies/onyx/ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix prov: <http://www.w3.org/ns/prov#>.
@prefix wnaffect: <http://www.gsi.dit.upm.es/ontologies/wnaffect/ns#>.

<#Hate>
    onyx:hasEmotionCategory wnaffect:Hate.

< ... > // emotions left out for readability

<#EmiliaClarke>
    rdf:type schema:Person;
    schema:name "Emilia Clarke".

< ... > // actors left out for readability

<#tt3778644-aggregateRating>
    schema:ratingCount 166,212;
    schema:ratingValue 7.0;
    schema:bestRating 10;
    schema:itemReviewed <#tt3778644>.

<#tt3778644>
    rdf:type schema:Movie;
    schema:identifier "tt3778644";
    schema:name "Solo A Star Wars Story (2018)";
    schema:duration "2h15min";
    schema:dateCreated "2018";
    schema:text "Description...";
    schema:aggregateRating <#tt3778644-aggregateRating>;
    schema:actor "Emilia Clarke";
    < ... > // actors left out for readability
    schema:character "Qi'ra";
    < ... > // characters left out for readability
    schema:image "https://m.media-amazon.com/images/M/MV5BOTM2NTI3NTc3Nl5BMl5BanBnXkFtZTgwNzM1OTQyNTM@
```

```
<#tt3778644-poopville>
    schema:about <#tt3778644>;
    schema:author "poopville";
    schema:dateCreated "Mon Aug 06 2018 00:00:00 GMT+0200 (CEST)";
    schema:reviewRating 7;
    schema:headline "Review...".

<#tt3778644-poopville-sentiment>
    rdf:type onyx:EmotionSet;
    onyx:emotionText "Review...";
    onyx:describesObject <#tt3778644-poopville-sentiment>;
    onyx:hasEmotion <#Hate>;
    prov:Entity <#tt3778644-poopville>.

<#tt3778644-poopville-sentiment-analysis>
    onyx:algorithm "RNG";
    onyx:source "http://www.imdb.com/title/tt3778644/reviews";
    onyx:usesEmotionalModel "http://www.gsi.dit.upm.es/ontologies/wnaffect#WNAModel";
    prov:generated <#tt3778644-poopville-sentiment>.
```

1.4 4 - Exploratory queries on the loaded dataset

1.4.0.1 The distinct wikidata types that may be aligned with the types in your dataset (advanced query)

```
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
SELECT DISTINCT (?wdc AS ?WikiDataClass) (?eqc AS ?EquivalentClass) WHERE {
    SERVICE <https://query.wikidata.org/sparql> {
    # wdt:P1709 - equivalent class
    # its description even says from other ontologies
        ?wdc wdt:P1709 ?eqc .
    }
    [] a ?eqc .
}
```

WikiDataClass	EquivalentClass
http://www.wikidata.org/entity/Q5	schema:Person
http://www.wikidata.org/entity/Q215627	schema:Person
http://www.wikidata.org/entity/Q11424	schema:Movie

1.4.0.2 Total number of triples

```
SELECT (COUNT(?s) AS ?triples)
WHERE {
    ?s ?p ?o
}
```

triples
"2715"^^xsd:integer

1.4.0.3 Total number of instantiations

```
SELECT (COUNT(?s) AS ?instances)
WHERE {
    [] a ?c
}
```

instances
"302"^^xsd:integer

1.4.0.4 Total number of distinct classes

```
SELECT (COUNT(DISTINCT ?c) AS ?classes)
WHERE {
    [] a ?c
}
```

classes
"10"^^xsd:integer

1.4.0.5 Total number of distinct properties

```
SELECT (COUNT(DISTINCT ?p) AS ?properties)
WHERE {
    [] ?p []
}
```

properties
"36"^^xsd:integer

1.4.0.6 List of all classes used in your dataset per data source (see named graphs)

```
SELECT DISTINCT (?g AS ?graphs) (?c AS ?classes)
WHERE {
    GRAPH ?g {
        [] a ?c
    }
}
```

graphs	classes
http://imdb.com/1	person:
http://imdb.com/1	movie:
http://imdb.com/1	onyx:EmotionSet
http://imdb.com/2	person:

graphs	classes
http://imdb.com/2	movie:
http://imdb.com/2	onyx:EmotionSet

1.4.0.7 List of all properties used in your dataset per data source

```
SELECT DISTINCT (?g AS ?graphs) (?p AS ?properties)
WHERE {
    GRAPH ?g {
        [] ?p []
    }
}
```

graphs	properties
http://imdb.com/1	rdf:type
http://imdb.com/2	rdf:type
http://imdb.com/1	onyx:hasEmotionCategory
http://imdb.com/2	onyx:hasEmotionCategory
...	...

1.4.0.8 Total number of instances per class per data source (reasoning on and off)

```
SELECT DISTINCT (?g as ?graphs) (?c as ?classes) (COUNT(?s) AS ?instances)
WHERE {
    GRAPH ?g {
        ?s a ?c
    }
}
GROUP BY ?g ?c
```

graphs	classes	instances
http://imdb.com/1	person:	"56"^^xsd:integer
http://imdb.com/1	movie:	"4"^^xsd:integer
http://imdb.com/1	onyx:EmotionSet	"94"^^xsd:integer
http://imdb.com/2	person:	"42"^^xsd:integer
http://imdb.com/2	movie:	"3"^^xsd:integer
http://imdb.com/2	onyx:EmotionSet	"44"^^xsd:integer

1.4.0.9 Total number of distinct subjects per property per data source

```
SELECT DISTINCT (?g as ?graphs) (?p AS ?properties) (COUNT(DISTINCT ?s) AS ?subjects)
WHERE {
    GRAPH ?g {
        ?s ?p []
    }
}
GROUP BY ?g ?p
```

graphs	properties	subjects
http://imdb.com/1	rdf:type	"154"^^xsd:integer
http://imdb.com/2	rdf:type	"89"^^xsd:integer
http://imdb.com/1	onyx:hasEmotionCategory	"29"^^xsd:integer
http://imdb.com/2	onyx:hasEmotionCategory	"29"^^xsd:integer
...	...	...

1.4.0.10 Total number of distinct objects per property per data source

```
SELECT DISTINCT (?g as ?graphs) (?p AS ?properties) (COUNT(DISTINCT ?o) AS ?objects)
WHERE {
    GRAPH ?g {
        [] ?p ?o
    }
}
GROUP BY ?g ?p
```

graphs	properties	objects
http://imdb.com/1	rdf:type	"3"^^xsd:integer
http://imdb.com/2	rdf:type	"3"^^xsd:integer
http://imdb.com/1	onyx:hasEmotionCategory	"29"^^xsd:integer
http://imdb.com/2	onyx:hasEmotionCategory	"29"^^xsd:integer
...	...	...

1.4.0.11 Distinct properties used on top 5 classes in terms of amount of instances (reasoning on and off)

```
SELECT DISTINCT (?p AS ?properties)
WHERE {
    ?s ?p [] .
    ?s a ?c .
    {
        # sub query for top 5 classes in terms of amount of instances
        SELECT ?c (COUNT(?s) as ?count)
        WHERE {
            ?s a ?c
        }
        GROUP BY ?c
        ORDER BY DESC (COUNT(?s))
        LIMIT 5
    }
}
```

properties
rdf:type
onyx:emotionText
onyx:describesObject
onyx:hasEmotion
...

1.5 Natural Language Processing

1.5.1 npm natural

- Text tagger
- Automatically Constructing a Dictionary for Information Extraction Tasks

1.5.2 POS tagger

tag	meaning	examples
CC	Coord Conjunction	and,but,or
CD	Cardinal number	one,two
DT	Determiner	the,some
EX	Existential there	there
FW	Foreign Word	mon dieu
IN	Preposition	of,in,by
JJ	Adjective	big
JJR	Adj., comparative	bigger
JJS	Adj., superlative	biggest
LS	List item marker	1,One
MD	Modal	can,should
NN	Noun, sing. or mass	dog

tag	meaning	examples
NNP	Proper noun, sing.	Edinburgh
NNP	S Proper noun, plural	Smiths
NNS	Noun, plural	dogs
POS	Possessive ending	O's
PDT	Predeterminer	all, both
PP$	Possessive pronoun	my,one's
PRP	Personal pronoun	I,you,she
RB	Adverb	quickly
RBR	Adverb, comparative	faster
RBS	Adverb, superlative	fastest
RP	Particle	up,off
SYM	Symbol	+,%,&
TO	ÒtoÓ	to
UH	Interjection	oh, oops
VB	verb, base form	eat
VBD	verb, past tense	ate
VBG	verb, gerund	eating
VBN	verb, past part	eaten
VBP	Verb, present	eat
VBZ	Verb, present	eats
WDT	Wh-determiner	which,that
WP	Wh pronoun	who,what
WP$	Possessive-Wh	whose
WRB	Wh-adverb	how,where
,	Comma	,
.	Sent-final punct	. ! ?
:	Mid-sent punct.	: ; Ñ
$	Dollar sign	$

tag	meaning	examples
#	Pound sign	#
"	quote	"
(	Left paren	(
)	Right paren	)

Table is from here.

1.6 Response Sentence Creation

- npm naturals bayesian classifier