Final Report

Final Report: Movie Information Chatbot

Introduction

This report details the development of a sophisticated chatbot designed to interact in chatroom environments, focusing on providing information related to movies. The bot integrates multiple modules for natural language processing (NLP), named entity recognition (NER), relation extraction, and recommendation generation, offering a comprehensive solution for various user inquiries.

Technical Implementation

- Python: Primary programming language.
- Spacy: For NER and text processing.
- RDFlib: For handling RDF graph data.
- Sklearn: Used in calculating embeddings and pairwise distances.
- Pandas & Numpy: For data manipulation and mathematical operations.
- Speakeasy API: For chatroom interactions.
- NumPy : Handles the loading and processing of pre-trained entity and relation embeddings.
- flair : Powers the NER and POS tagging to extract entities and understand sentence structure.
- transformers : Provides pre-trained models for NER and token classification. Levenshtein : Assists in finding the closest matching string from a list, aiding relation extraction.
- inflect : Used to handle pluralization and singularization of nouns during relation extraction.
- re : Enables regular expression operations critical for parsing and extracting information from RDF data.

Methodology

1. Data Loading: The bot loads RDF graph data, embeddings, and other necessary datasets at initialization.
2. Message Processing: On receiving a message, the bot first attempts to extract entities and relations. Depending on the query type, it then either constructs a SPARQL query for direct answers or resorts to embedding-based inference.
3. Response Generation: The bot generates responses based on the query type – direct answers from RDF data, recommendations using embeddings, or crowdsourced information.
4. Multimodal Interaction: In addition to textual responses, the bot can handle image-based queries, providing a richer interaction experience.

Key Components

1. **Agent Class**: The main class that handles the bot's operations. It includes methods for loading data, handling messages and reactions, and generating responses.
2. **NER Module (NER_tool)**: Utilizes Spacy's transformer-based model to extract entities from the text.
3. **Relation Extraction (relation_extractor)**: Processes sentences to extract potential relations using POS tagging and word forms.
4. **Recommendation System (MovieRecommender)**: Generates movie recommendations based on user queries. It appears to use RDF data and embeddings for this purpose.
5. **Crowdsourcing Module (Crowdsource)**: Handles cases where the bot resorts to crowdsourced data to find answers, using a pre-processed dataset.
6. **SPARQL Queries**: Used for extracting data from RDF graphs, particularly for direct answers related to movies.

7. **Embedding-based Response Generation**: For cases where direct answers are not found, the bot uses embeddings to generate probable responses.

Key Features

- **Handling of various query types**: Our bot can handle direct queries, recommendations, and more complex inquiries.
- **Multimodal Responses**: Ability to respond with text and images
- **Embedding-based Inference**: For questions that cannot be directly answered from the RDF graph, the bot uses embeddings to infer likely answers.
- **Integration with Chatrooms**: The bot is designed to interact in chatroom environments, responding to both messages and reactions.

Key Capabilites

- **Factual Data Retrieval**: Extracts precise information from RDF graphs.
- **Embedding-Based Inference**: Infers answers using calculated embeddings when RDF data doesn't suffice.
- **Multimodal Interactions**: Handles both text and image-based queries.
- **Recommendations**: Provides movie suggestions based on user preferences.
- **Crowdsourced Data Processing**: Utilizes crowdsourced information for complex queries.

Code Snippets and Explanations

Factual Data Retrieval

The chatbot extracts factual information using SPARQL queries. Here's a snippet demonstrating a SPARQL query construction:

```
def construct_query(self, entity, relation):
    return f''' PREFIX wdt: <http://www.wikidata.org/prop/direct/> SELECT ?obj WHERE {{ ?ent rdfs:label '{entity}'@en . ?ent wdt:{relat
```

This function dynamically constructs a query to fetch data related to the extracted entity and its relation. Apart from extracting one certain movie or other instance, we need to upgrade our NER system to makesure that it can detect multiple movie names for recommendation. Thus, we change the NER from using BERT-based model to Spacy. Now it can extract multiple movies in one sentence.

```
def get_relations(self, message, relations, graph, WDT):
        # lemmatizer = WordNetLemmatizer()
        nouns = []
        relation = []
        auxiliary_verbs = ["is", "are", "am", "was", "were", "have", "has", "had", "be", "been", "do", "does", "did", "will", "shall",
        properties = set()
        properties.update(self.noun_film_properties)
        properties.update(relations)
        inflect_engine = inflect.engine()

        pos = self.get_pos(message)
        for pair in pos:
            if pair[0].lower() in auxiliary_verbs:
                continue
            if pair[1][:2] == "VB":
                noun_conversions = get_word_forms(pair[0].lower())['n']
                matching_conversions = set(properties).intersection(noun_conversions)
                for noun in matching_conversions:
                    if noun in relations:
                        nouns.append(noun)
                    elif noun in self.noun_film_properties:
                        for k, v in self.noun_map.items():
                            if noun in v:
                                nouns.append(k)
                                break
            elif pair[1][:2] == 'NN':
                #check if plural
                if pair[0][-1] == 'S':
                    noun = inflect_engine.singular_noun(pair[0].lower())
                else:
                    noun = pair[0].lower()
```

```
                if noun in relations:
                    nouns.append(noun)
                elif noun in self.noun_film_properties:
                    for k, v in self.noun_map.items():
                        if noun in v:
                            nouns.append(k)
                            break
            elif pair[0] == 'executive':
                nouns.append('executive producer')
                break

        # relation = ''.join(relation)

        for noun in set(nouns):
            if noun == 'producer':
                relation.append({'relation': noun, 'ids': 'P162'})
                break
            elif noun == 'excutive producer':
                relation.append({'relation': noun, 'ids': 'P1431'})
                break
            relation.append({'relation': noun, 'ids': self.getRelation_ID(graph, noun, WDT)})
        return relation

    def getRelation_ID(self, graph, relation, WDT):
        '''
        recieves nouns (relations) and converts them to their URI ID
        '''

        query = f'''
            prefix wdt: <http://www.wikidata.org/prop/direct/>
            prefix wd: <http://www.wikidata.org/entity/>

            SELECT ?res
            WHERE{{
                ?res rdfs:label "{relation}"@en.
                }}'''
        URIs = list(filter(lambda x: WDT in x, [x[0] for x in list(graph.query(query))]))

        rel = []
        for uri in URIs:
            if WDT in uri:
                relId = re.match("{}(.*)".format(WDT), uri)[1]
                rel.append(relId)
        return rel
```

```
class NER_tool:
    def __init__(self):
        # Load spaCy's transformer-based model
        self.nlp = spacy.load("en_core_web_trf")
        print('spaCy NER model initialized.')

    def extract_entities(self, text):
        # Process the text and extract entities
        doc = self.nlp(text)
        entities = [entity.text for entity in doc.ents]
        return entities
```

Embedding-Based Inference

When direct answers are not available, the bot uses embeddings for inference:

```
def create_response_emb(self, ent_id, relation, entity_emb, ent2id, relation_emb, rel2id, ent2lbl, id2ent):
        ent_uri = URIRef(f"http://www.wikidata.org/entity/{ent_id}")
        rel_uri = URIRef(f"http://www.wikidata.org/prop/direct/{relation[0]['ids'][0]}")
        head = entity_emb[ent2id[ent_uri]]
        pred = relation_emb[rel2id[rel_uri]]
        lhs = head + pred
        dist = pairwise_distances(lhs.reshape(1, -1), entity_emb).reshape(-1)
        rank = dist.argsort()
        most_likely = rank[0]
        emb_result = ent2lbl[id2ent[most_likely]]
        return emb_result
```

This method calculates embeddings to infer the most likely response based on the proximity of embeddings.

Multi-Modal Interactions

The bot can handle image-based queries, enhancing user experience:

The code snippet can be formatted as follows:

```
logging.info("Multimedia mode")
            mutltimedia_mode = True
            imdb_query = f'''
                PREFIX wd: <http://www.wikidata.org/entity/>
                PREFIX wdt: <http://www.wikidata.org/prop/direct/>

                SELECT ?imdbID
                WHERE {{
                    wd:{ent_id_short} wdt:P345 ?imdbID.
                }}
            '''
            imdb_id = self.run_sparql_query(imdb_query)
            logging.info(f"imdb id: {imdb_id}")
            im_id = ''
            for image in self.images:
                if image['cast'] == [imdb_id] and image['type'] != 'poster':
                    im_id = image['img']
                    im_id = 'image:'+im_id.strip('.jpg')
                    break
            if im_id:
                formatted_response = f" There you go... {im_id}"
            else:
                formatted_response = f" Couldn't find a image of {entity_name}"

            return formatted_response
```

This conditional statement checks for image-related keywords in the user's message and triggers image retrieval. After that it searches the IMDB ID and the goes to the provided image.json to find a ID with a picutre and sends the jpg path to the chat where it will be outputed as an image.

Recommendations

For recommendations, the bot uses user preferences to suggest movies:

```
class MovieRecommender:
    """
    A class for a movie recommender system using RDF graph and entity embeddings.
    """

    def movie_recom_movie(self, movie_ids, top_n):
        """
        Recommend movies based on a set of input movie IDs.
        """
        recom_movie = []
        input_embeddings = [self.entity_emb[self.ent2id[self.WD[id]]] for id in movie_ids]
        ent_mean = np.mean([self.entity_emb[self.ent2id[self.WD[id]]] for id in movie_ids], 0)
        similarity_scores = cosine_similarity([ent_mean], self.entity_emb)[0]
        recommended_movie_indices = np.argsort(similarity_scores)[::-1]
        for idx in recommended_movie_indices:
            if len(recom_movie) >= top_n:
                break
            current_emb = self.entity_emb[idx]
            if not any(np.array_equal(current_emb, inp_emb) for inp_emb in input_embeddings):
                movie_name = self.ent2lbl[self.id2ent[idx]]
                recom_movie.append(movie_name)

        return recom_movie

    def genre_recom_movie(self, genre):
        """
        Recommend movies based on a specified genre.
        """
        query = f'''
        PREFIX wdt: <http://www.wikidata.org/prop/direct/>
        PREFIX wd: <http://www.wikidata.org/entity/>
        PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

        SELECT ?movie ?movieLabel WHERE {{
            ?movie wdt:P31 wd:Q11424;  # Instance of film
                wdt:P136 ?genre.    # Has genre
            ?genre rdfs:label "{genre}"@en.  # Genre name in English
            OPTIONAL {{ ?movie rdfs:label ?movieLabel FILTER(LANG(?movieLabel) = "en") }}
        }}
        LIMIT 5
        '''
```

```
        qres = list(self.graph.query(query))
        return [str(row.movieLabel) for row in qres if row.movieLabel]

    def actor_recom_movie(self, actor):
        """
        Recommend movies based on a specified actor.
        """
        query = f'''
        PREFIX wdt: <http://www.wikidata.org/prop/direct/>
        PREFIX wd: <http://www.wikidata.org/entity/>
        PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

        SELECT ?movie ?movieLabel WHERE {{
            ?person rdfs:label "{actor}"@en.
            {{
                ?movie wdt:P161 ?person.
            }}
            UNION
            {{
                ?movie wdt:P57 ?person.
            }}
            OPTIONAL {{ ?movie rdfs:label ?movieLabel FILTER(LANG(?movieLabel) = "en") }}
        }}
        LIMIT 5
        '''
        qres = list(self.graph.query(query))
```

This function generates movie recommendations tailored to the user's stated preferences.

As for the recommendation by movie, we use the cosine similarities. Firstly, all embeddings of movies will be find and used to calculate the average vector. Then we'll find the top-k nearest vector movie embeddings from the average vector and select them as the recommendations. As for genre and person recommendation, we use sparql query to search for movies sorted by IMDB id.

Crowdsourced Data Processing

The bot integrates crowdsourced data for complex queries:

Here is the formatted code snippet:

```
import rdflib
import pandas as pd
import numpy as np
import csv
from rdflib.namespace import RDFS
from ner import NER_tool
from get_relations import relation_extractor
from rdflib import URIRef

class Crowdsource:
    def __init__(self, graph, relations, lbl2ent, ent2lbl, WD, WDT):
        self.crowdSource = pd.read_csv('dataset/crowd_data/crowd_data.tsv', sep='\t')
        self.crowdSource['LifetimeApprovalRate'] = self.crowdSource['LifetimeApprovalRate'].str.rstrip('%').astype('float')
        self.crowdSource = self.crowdSource.sort_values(by=['HITId', 'LifetimeApprovalRate', 'WorkTimeInSeconds'], ascending=[True,
True, True])
        self.crowdSource = self.crowdSource.drop(self.crowdSource.groupby('HITId').head(2).index)
        self.ner = NER_tool()
        self.relation_model = relation_extractor()
        self.relations = relations
        self.lbl2ent = lbl2ent
        self.ent2lbl = ent2lbl
        self.graph = graph
        self.WD = WD
        self.WDT = WDT
        self.extract = None


    def crowdSearch(self):
        data = self.crowdSource
        id = self.extract['HITTypeId'].unique()[0]
        data = data[data['HITTypeId'] == id]
        data = data.groupby(['HITId', 'AnswerLabel']).size().reset_index(name='count')
        data['count'].astype(int)
        data = data.pivot(index='HITId', columns='AnswerLabel', values='count')
        data.fillna(0, inplace=True)
        # data['total'] = data['CORRECT'] + data['INCORRECT']
        data['P_i'] = (data['CORRECT']**2 + data['INCORRECT']**2 - 3)/6
        data['P_i'].astype(float)
        P_c = data['CORRECT'].sum(axis=0)/(3*len(data))
        P_i = data['INCORRECT'].sum(axis=0)/(3*len(data))
```

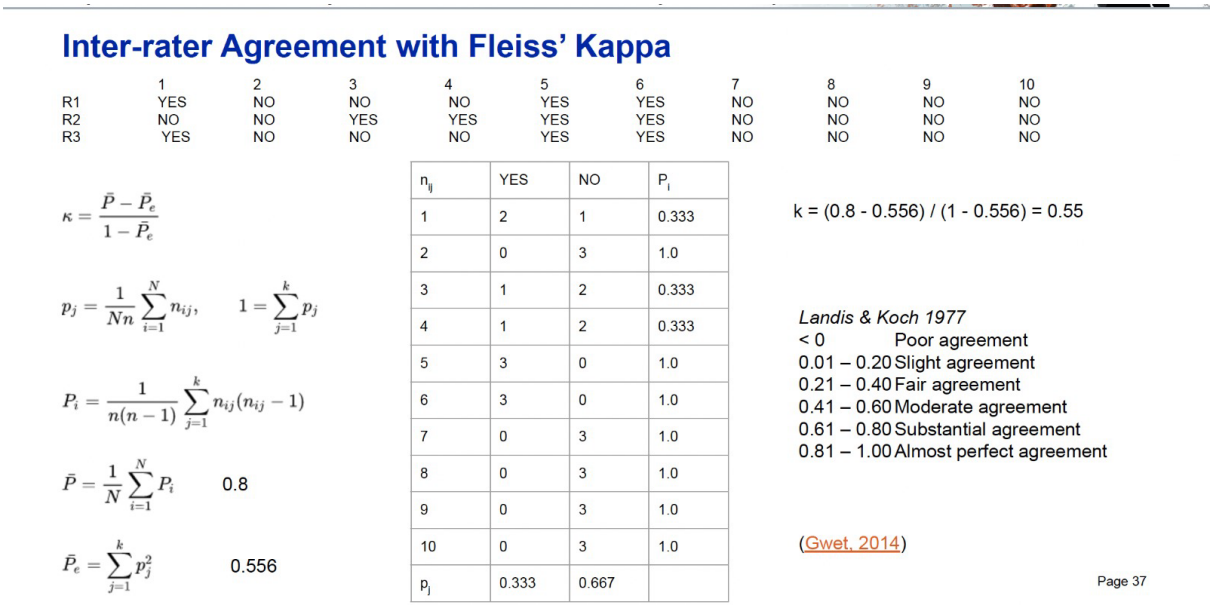
```
        P = data['P_i'].sum(axis=0)/len(data)
        P_e = P_c**2 + P_i**2
        kappa = round(((P - P_e)/(1 - P_e)), 3)

        result = self.extract['Input3ID'].iloc[0]
        answer_label = self.extract['AnswerLabel']

        if result[:1] == 'wd':
            full_uri = f"http://www.wikidata.org/entity/{result[3:]}"
            entity_uri = URIRef(full_uri)
            result = self.ent2lbl[entity_uri]os*(pos-1)+neg*(neg-1))/(n*(n-1)))


        return result, answer_label, kappa
```

This method processes crowdsourced data, cleaning it based on specific criteria, and calculates Cohen's kappa for inter-rater agreement.



Inter-rater Agreement with Fleiss' Kappa

	1	2	3	4	5	6	7	8	9	10
R1	YES	NO	NO	NO	YES	YES	NO	NO	NO	NO
R2	NO	NO	YES	YES	YES	YES	NO	NO	NO	NO
R3	YES	NO	NO	NO	YES	YES	NO	NO	NO	NO

$$\kappa = \frac{\bar{P} - \bar{P}_e}{1 - \bar{P}_e}$$

$$p_j = \frac{1}{Nn} \sum_{i=1}^{N} n_{ij}, \qquad 1 = \sum_{j=1}^{k} p_j$$

$$P_i = \frac{1}{n(n-1)} \sum_{j=1}^{k} n_{ij}(n_{ij} - 1)$$

$$\bar{P} = \frac{1}{N} \sum_{i=1}^{N} P_i \qquad 0.8$$

$$\bar{P}_e = \sum_{j=1}^{k} p_j^2 \qquad 0.556$$

n_{ij}	YES	NO	P_i
1	2	1	0.333
2	0	3	1.0
3	1	2	0.333
4	1	2	0.333
5	3	0	1.0
6	3	0	1.0
7	0	3	1.0
8	0	3	1.0
9	0	3	1.0
10	0	3	1.0
p_j	0.333	0.667	

k = (0.8 - 0.556) / (1 - 0.556) = 0.55

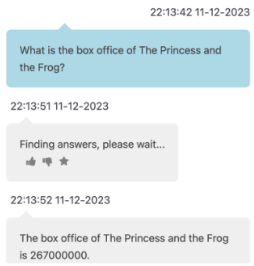
Landis & Koch 1977
< 0 Poor agreement
0.01 – 0.20 Slight agreement
0.21 – 0.40 Fair agreement
0.41 – 0.60 Moderate agreement
0.61 – 0.80 Substantial agreement
0.81 – 1.00 Almost perfect agreement

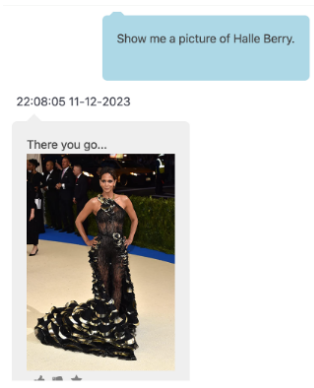
(Gwet, 2014)

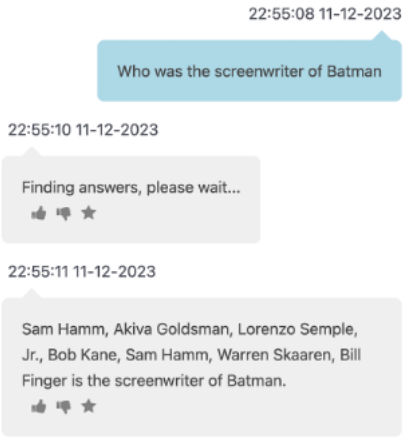
Page 37

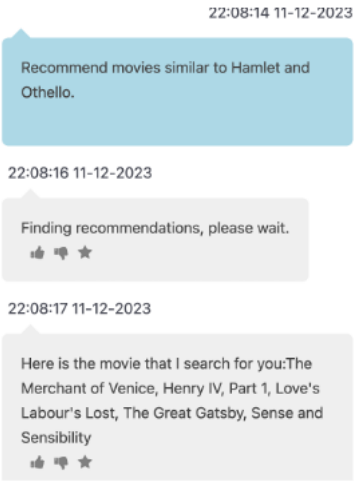
We extract the answer by the given entity id and relation id. Before that the whole data needs a cleaning by removing the data for the same question which has the least two lifetimeapproval rate (if its the same delete the shorter worktime)
When calculating the inter-rate agreement, we calculate it for each question regarded by the HITTypeId, which means that if this id is the same for questions, their rate is the same
And the inter-rate formula is the Cohen's kappa, Crowd on web I or II.

Screenshots






22:55:08 11-12-2023
Who was the screenwriter of Batman
22:55:10 11-12-2023
Finding answers, please wait...
22:55:11 11-12-2023
Sam Hamm, Akiva Goldsman, Lorenzo Semple, Jr., Bob Kane, Sam Hamm, Warren Skaaren, Bill Finger is the screenwriter of Batman.


22:08:14 11-12-2023
Recommend movies similar to Hamlet and Othello.
22:08:16 11-12-2023
Finding recommendations, please wait.
22:08:17 11-12-2023
Here is the movie that I search for you:The Merchant of Venice, Henry IV, Part 1, Love's Labour's Lost, The Great Gatsby, Sense and Sensibility