The Irismé (Information Retrieval is mé) Report

Word count: 1930

Introduction

Irismé is a simple chatbot with a bit of flair. How she works is very simple. After introducing herself and providing the user with some information about how to quit() or ask for their name, she enters an infinite while loop and gives the user an opportunity to say something. Based on the response, she can recognise 4 user intents namely:

- **HI - Hello/greetings**
- **ST - Small talk**
- **NC - Name change**
- **IR - Information request**

Before moving on, I'd like to mention that I've used the some samples from https://github.com/rahul051296/small-talk-rasa-stack/blob/master/data/nlu.md to provide data for the **HI** and **ST** categories.

So upon user input, Irismé uses her intent classifier, which is built through a K-NN classifier, to predict what it is the user wants. Once classified, she will run through here various if statements to check for what to do in what case and then one of either two things happen.
Either she gets an intent of anything but **IR** which she then handles accordingly and gets responses from a settings file which contains various variables used throughout the project including the variables storing her personal responses. These are lists which contain 3-4 strings each and the final response is given selected randomly through random.choice().
If the response is an **IR** Irismé invokes the retrieve_answer() from the answer_finder module and through using TfidfVectorizer and cosine_similarity she manages to find the most suitable answer for the given question and handles multiple answer finds by doing the exact same thing on this new list of potential answers. That's pretty much all there is to it from a high level view.

Design

Name management and small talk

Name management and small talk are the simplest parts of the bot. The user can request their name from Irismé anytime by typing in “Say my name” to which the bot will send a response.

The user’s name is saved in settings.py which is really just a cheap and easy config file. Whenever the user does a name change, their detail is updated in settings and any responses that use the username are reinitialised after setting the name.

It follows that all responses as well as the greeting is stored here. Irismé has two types of responses for the same intent when it comes to **HI** and **ST**. Once an intent of either type has been received, a flag is set in main.py which will then do a random.choose() from the whatever list within settings.py was specified. This way she seems a lot more natural albeit limited due to only two types of small talk intents.

Intent matching and Question answering

Before talking about either part I would point out that I do not use stop word filtering at all. I took this approach because in the case of the data that is being worked with here, documents are extremely short so from that aspect alone, you end having a lot of sentences consistent of mostly stop words. But what was the main deciding factor was that in the context of questions (and even other intents used), regular stop words like “a”, “with”, “what”, “do” have extremely high semantic meaning relative to regular usage so stop word removal does not make much sense here.

To start off with, for both Intent matching and question answering, I used the same text preprocessing steps which are described in the following module.

text_preprocessing.py

It does what it says. Other modules only call preprocess_data or preprocess_datum (when having only a single string) which internally do the following in the order listed:

- Clean_text is called. Here all punctuation barring **'s** is removed and text is lowered.
- Before passing the newly cleaned text on, I use the contractions library to convert stuff like "I've" and "haven't" into "I have" and "have not".
- The text is tokenized and tagged for parts of speech
- The text is lemmatized and formatted appropriately to be returned.

I used WordNetLemmatizer as it was simple, lightweight and I knew how to use it when starting off. Given that I only managed to make a start during the grace period, there were other parts of the system I invested my time into and hence did not explore other options here.

Intent matching with IntentClassifier()

Intent matching is the simpler part of the ML side of things. I created this class and put it into its own module. An instance is initialised in settings.py and works the following way:

- The Intent classifier dataset, which was partly made up of an external data source referenced earlier on, is read and the formatted accordingly (questions are sent to one list, their corresponding intent is sent to another).
- The questions are preprocessed through preprocess_data()
- The intents and the questions are used as dependent and independent variables respectively for test_train_split along with other parameters that led to optimum performance.
- When it comes to vectorizing, I tried using Tfidf vectorizer here because it required less steps to do the same thing but for some reason classification performance was higher with using both CountVectorizer and TfidfTransformer.
- For the actual classification, I used an KNeighborsClassifier. It seemed like the most suited choice for the data at hand given that there were only 4 classes to learn and KNN was very fast. 5 neighbors with a test size of 0.225 yielded the highest accuracy and f1 score of both being around 89%.

Question answering

This was the most challenging and satisfying part of the project. When developing the very first parts of the code, I ran into a problem. I could not figure out for the life of me, how I could retrieve an original question after vectorizing it. This was crucial because I needed the vectorized question in order to gauge similarity with a vectorized user input and then look up the answer in the dataset via the original question. While I'm sure there is the "data-science" way to do it via csr.matrix and numpy operations, I came up with the following solution which enabled me to couple and **keep track** of which vector belongs to which dataset question.

tfidf_vectorizer_tracker.py

It's dumb but it w

The class contains a TfidfVectorizer referenced by vectorizer, two dictionaries being pre_post_pairs and vector_document_tracker and finally vector_space which would store the result of vectorizer.fit_transform.

The class initialises as follows:

- The initializer is passed the questions we wish to pair up with their vectorized value.
- The pre_post_pairs is injected with the questions with every key being a question and its respective value being that very same question except we use text_preprocessing.py to clean it up.
- We then preprocess the questions as a whole rather than one by that result is what vectorizer calls fit_transform() on.
- Now we have the questions vectorized and ready to be used but there is still no way to back trace the vectors to the original question and by extension the answer hence the following code occurs:

```
for document, preprocessed_document in self.pre_post_pairs.items():
    self.document_vector_tracker[document] = self.vectorizer.transform([preprocessed_document])
```

- This image should describe what's going on a lot better than I can. The end result is now we have a way of tracking every question vector such that when we find the highest similarity between input vector and a given question vector, we can easily get a hold of the question we need to look up the answer to return to the user.

answer_finding.py

This is the meat of the pie. The similarity of an input is measured against all stored questions. The one with the highest similarity score is used to look up the answer. If there is more than one answer, the same thing happens again except only this time, the user input is compared to the newly returned list of answers to find the one with the highest similarity and then finally return that as the answer. The rationale here is that many questions in the dataset have multiple answers with different keywords (on top of the actual question words) and such, perhaps the user's input contains certain ones that make one potential answer more suitable than another.

It seemed pretty clear that cosine_similarity was the most suitable distance measure. It's fast and it works really well with sparse vectors which is exactly what we have given the breadth and size of the dataset.

When the user intent is an **IR,** main.py calls this module's retrieve_answer() which does the following

- A TfidfVectorizerTracker (**TFVT**) already initialised, fitted and transformed on the main dataset transforms the user input and then loops through its document_vector_tracker to compare the

- Takes in user input and passes it to get_likely_answer() which uses it, a TfidfVectorizerTracker (**TFVT**) that was initialised in settings.py and the Information retrieval dataset also set in settings.py.
- Get_likely_answer uses the TFVT's vectorizer, which was already trained on all the dataset's questions, to transform the users input.
- The TFVT's document_vector tracker is looped through in order to extract every (preprocessed) question vector and:
    - Get the cosine_similarity of the vectorizes input and the question vector
    - If the similarity_score is higher than the currently highest_score then set the tracker key (original question) to be the top_result
- Once all questions have been iterated through, we return the appropriate data based on whether or not we have only one or multiple answers
- The function returns the answer along with a bad_match variable which is set to True if the question with the highest score has a similarity of less than 0.5 which will cause Irismé to respond accordingly.

Evaluation

On a general level, the first thing that comes to mind is adding an **NR** (Name request) intent would have been handy. I chose not to do so because of time constraints preventing me from finding suitable data samples in order to be representative of the intent and not decrease the overall performance of the intent classifier. Adding to that, Irismé can only respond to greetings and stuff like “How’s your day” and “How’ve things been”. Having a few more small talk intents would have been nice.
Perhaps to make the small talk even more dynamic, there could be some machine learning done on those inputs. For example, “How’s the weather?” should return a different answer to “How’s life” and so on.

My TfidifVectorizerTracker might not have been needed had I known how to backtrack from the vector to the original question properly. That would involve learning more about the underlying data structures that you get when doing nlp which would only equip me better when it comes to this kind of work.

Regarding the test questions provided, Irismé did quite well. If she had a low scoring top_result, she would print out a disclaimer before giving the answer. The threshold is at <50% similarity which results in the answer to single malt whiskey being disclaimed even though it was correct. The same happened for the Atlantis question which was a simple typo (although I wish Atlantis was actually a thing.) The hockey question returned the only valid result without being disclaimed which it should have been in my opinion. That’s interesting to say the least and given more time, I would have delved deeper into how similarity scores worked such that stuff like the latter doesn’t happen

Overall, answers do fit expectations and some degree of typos, depending on the question will yield the same answer although sometimes it veers off and produces something unexpected. As for small talk, Irismé is extremely informal and the idea behind that was I wanted her replies to be as if it was a friend texting you. So long as something fits within the small talk intents, a natural response is given within limits.