Information Retrieval Project

Pet Adoption Search Engine

Andrea Cardia, Fabrizio De Castelli

December 16, 2022

1. Introduction

In this project, we collected information regarding pets from three websites that contained adoption listings for different types of animals.
Thus, we built a collection of documents in which each document represented a pet with associated various characteristics such as name, age, size, a description, and so on.
We chose not to homogenize the information extracted from each site in order not to have in the end an overly structured collection of data, and thus to stay within the proper scope of Information Retrieval. The sites from which data were extracted to build the collection were:

- petsmartcharities (`https://petsmartcharities.org`)
- bestfriends (`https://bestfriends.org`)
- petrescue (`https://www.petrescue.com.au`)

These sites were chosen on the basis of their high structural form; this feature is essential for the crawler to be able to extract information easily. We will look more deeply into this issue in the next section.

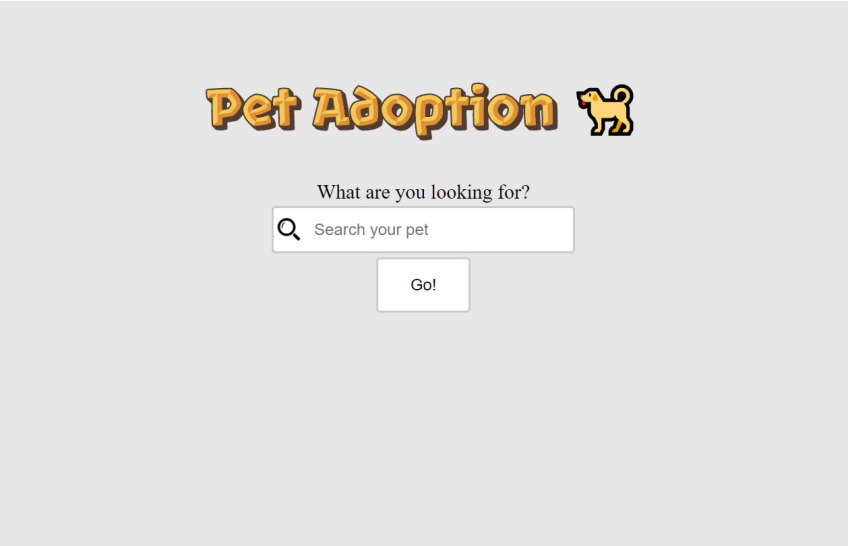
2. Crawling

The first thing to do to build a search engine is crawling. This involves automatically crawling several web pages (in our case the three listed in the introduction) and extracting the data of interest. To build our pet search engine we used `Python`'s API `scrapy`.
Using this tool, several requests were sent to the servers of the three different Web sites (one Web site at a time) going through all the pages that were available to gather information about the pets present.
In the end, our collection boasted truly all kinds of pets! From the typical dogs and cats, to the most sought-after piglets, hamsters, horses, guinea pigs, parrots and various feathered animals, and many more! The places from which all these pets came were the places where the sites from which the information was extracted operated: namely, the U.S. and Australia.
At the end of this phase the final product of our work were three `json` files ready to be indexed; in these files there were pets extracted from each of the three Web sites. The indexing phase will be described in the next section.

3. Indexing

For indexing the collection, we used the open-source platform `Solr` and we uploaded our files to be indexed by the system. This indexing was done without favoring any particular field and with the standard analyzer; we made this choice in anticipation of an eventual user-made query in which the match must be searched across all fields of each individual object (the pet).

4. Front-end



The front-end design was implemented in `html` and `css`. Our interface is minimal, we chose to display only the search bar and a button `Go!` to enter the user's query. Results are then displayed in a tabular format, where in each box the user can get some information about the animal itself:

- Photo;
- Name;
- Breed;
- Fields in which the query matched (for example Story, Location, Age, ...).

The user can click inside a box to be redirected to the page of the animal to see additional information about him/her and maybe decide to adopt it! The link refers to some page belonging the original website we chose to crawl the information from. Since all websites are dynamic it might be possible that some pets are not available for adoption anymore, one way to avoid it could be to periodically crawl the websites in order to keep the dataset updated.
We decided to let the user the possibility to click on a button that refers to another page of our search engine called `Suggested Pets`, situated at the bottom of query results, where he/she can get other automatically suggested pets from our collection, according to `Solr`'s `MoreLikeThis` feature (we are going to explain the details in Section 6 of this Report).

5. Back-end

The back-end was implemented in `Python`, using `flask` and `jinja2`. The search engine is loaded at `127.0.0.1:5000` and has three `html` pages: `base.html`, `index.html` and `suggest.html`. The base page is loaded when the user access to the url and it is extended once the user triggers some event, as a query or a button click. The extension of the page is managed by `jinja2`, that allows to put some `Python` data inside the `html` source code.
When the user enters a query, he/she is implicitly sending a request to `Solr` through the script `app.py`. This application will then process results and send it to the `html` code, depending on in which page the user is (if in index page or suggest page). Below follows how to run the code, after having started `Solr` server:

- Open anaconda or miniconda command prompt;
- Go to the directory where `app.py` is located. You should see `app.py`, a directory called `static` and a directory called `templates`. `static` contains only static files such as `.css` files and images, while `templates` contains `html` scripts;
- Activate a `Python` environment by typing `conda activate environment_name`;
- Install flask library by typing `conda install flask`;
- Type `set FLASK_APP=app`, (if your `Python` script is called let's say `program.py` then the command would be: `set FLASK_APP=program`);
- Type `set FLASK_ENV=development`, to enable debugging (not mandatory);
- Run the program by typing `flask run`;
- Open url `127.0.0.1:5000` on a browser.

Queries are scored with a `AND` operator, since we have many similar animals and we wanted to have more precise matching inside fields, in case of multi term queries.
We decided to display only the 20 most relevant pets as output of the query, this setting can be easily modified in `app.py` by changing `rows` field to any desired value.
We also decided to keep track of matched fields for each retrieved animal by using the default `solr highlighting` component with this configuration:

- `hl.fl=*` To get highlighting on every field;
- `hl=true` To enable `highlighting` component.

6. Implemented Features

We chose Results presentation as a simple feature (tabular format, as discussed in the Front-end section) and Automatic Recommendation as a complex feature. The latter was implemented in `Solr` with `MoreLikeThis` feature: we chose to enable this component by expanding the query with `mlt` fields:

- `mlt.fl=feature,color` Can be modified according to which field we want to use to generate suggestion;
- `mlt.mintf=1` Namely minimun term frequency for each specified field in the previous point, to get a more general match of terms in each field.
- `mlt.count=1` Allows to get only one suggested pet per query result, thus we have number of suggestion to be smaller than the number of query results;
- `mlt=true` To enable `MoreLikeThis` feature.

Each field we want to build the suggestion process with has to be modified in the `managed-schema.xml` inside our core: it is enough to add `termVectors="true"` at the bottom of the field definition, for each field. In our case we modified color and feature field definitions as follows:

- `<field name="color" type="text_general" termVectors="true"/ >`
- `<field name="feature" type="text_general" termVectors="true"/ >` .

The handling of suggested items was assigned to a `Python` thread running in background to slightly improve the performance when a query is typed.

7. Evaluation

We carried out only the User Evaluation without worrying about the System Evaluation.
To do this, we asked four different users to complete three tasks that we had prepared in advance. The tasks consisted of asking the user to collect information (formulate a query of his or her own) based on a trace that we proposed from time to time. We initially proposed traces that would prompt the user to formulate two queries of low difficulty, then two queries of medium difficulty, and finally two of fair difficulty.
For simple queries we proposed: ”Searching for a cat with a certain eye color” and ”Searching for an animal of a certain breed”.
For medium difficulty queries: ”Searching for an animal of a certain size and color” and ”Searching for an animal in America of a certain age (in the young/old form)”.
For queries of moderate difficulty instead: ”Search for a stray animal (specifying male or female) of a certain age (again in the young/old form)” and ”Search for an uncommon animal with a certain personality (behavior)”.
The results were:

- First Type of Query:

- A user searched for a tiger but could not find it;
- A user wanted a cocker dog with blue eyes but didn't find anything;
- A user clicked on a cat and didn't like it;
- A user was happy with browsing on our site because and she got satisfied with the search;
- A user laughed at the display of a pig in the first query when he was expecting a brown-eyed dog but was left is satisfied with the rest;

- Second Type of Query:
  - A user is disappointed because we don't have an automatic word checker like she is used to from google (maybe a joke);
  - There are some gaps in the user interface, but the user finds relevant animals provided as a result;
  - He looked up the company offering adoptions and as the last feedback told us "Your system works well!";
  - One user said that overall the usability of the system was fine because it matched most of the requirements he wanted to achieve, although some off-topic animals were displayed. This user did not see any suggestions;
- Third Type of Query:
  - The user tried to decide whether our system is intelligent;
  - During a search for a stray animal, it was classified as young, but noted that we actually do not know if it was young when it was found in the street;
  - The age search returned some misleading records for one user, but overall it was rated positive.
  - For a user, the search for "non-breed puppy" yielded little information;
  - For a user, a few cats and a purebred dog (not the one he wanted) were retrieved;
  - A user searched for an aggressive bird, but none was displayed;
  - A search for hamsters did not go well because several cats were displayed;

8. Conclusions

Our system can be improved since the results of the questionnaires were not great on some points.
In particular we report that some users gave a not-so-high rating on the confidence of using the system, that functions were not that well integrated (perhaps due to the fact that we did not implement a filtering mechanism).