User Profile-based Information Retrieval System

Github: https://github.com/soomin-jeong/profile-based-information-retrieval

Soo Min Jeong
EIT Digital Master - Data Science
Universidad Politécnica de Madrid

1. Introduction

When Youtube introduced a recommendation system based on Deep Learning, its watch time increased by 2000% in the next three years, reaching one third of the global population. (Engström, E., & Strimling, P., 2020). Recommending contents to a user is reading a particular context. Based on the trace that the user left on the platform, they predict users' taste, classify contents into certain categories of tastes, and match the contents with a user. It is an art of reading a mind.

The reason why the recommendation system got so popular is that it can trap a user into the service. The users do not have to sweat to try and fail looking for something interesting to themselves. The service knows better about users' taste than the users themselves, and it knows better about the contents on it. It is a winning game of asymmetric information.

If we know a user's taste and some contents labeled with the lists of tastes, we can recommend contents to a user. This project is a simplified version of reading a context out of a user. Under assumption that we know a user's taste and define it into a few words, this project associates some text with the words. We call the words defining a user's taste '*profile*' here. The system designed in this project learns data and profiles and recommends a specific data to a user based on their profile.

2. Literature

Based on the acknowledgement of the importance of this topic, there have been many publications with different approaches of solution. *Chen and Kuo* introduced a solution to update the Information Retrieval System by learning users' cognition of the term to understand their cognitive semantics (2000). *Arezki, Poncelet, Dray, & Pearson* extended this solution with PQIR model. PQIR model is a matrix model to represent documents, requests, users' profiles, and term set indexing (2004). In this model, they tried to integrate users' interests in the Information Retrieval process.

In the same year, they released another paper with a concrete problem field, World Wide Web(WWW) to solve profile-based information retrieval problems. Their concept of 'profile' was defined more complex as a user's '*knowledge acquired through time on the thematic of request*' (Arezki, Poncelet, Dray, & Pearson, 2004).

Most of the papers assumed that they already have a great deal of dataset prepared to match with the users' interests, and focus on how to interpret users' interest. This project, however, rather focuses on how to prepare a dataset to be matched with the users' interest and train a machine learning model in a short period of time for better accessibility.

3. Features

a. Initial Data

The program includes initial data of three virtual users with different multiple interests. The details are as follows.

```
{"user1": ["politics", "soccer"],
 "user2": ["music", "films"],
 "user3": ["cars", "politics"],
 "user4": ["soccer"] }
```

Code 1: Initial Dataset of Users

The interests are in a single set, and each topic is with a text dataset as below. How the dataset was collected is described at *b.Web Scraper*.

```
Police body cam video shows George Floyd struggle, then takedownGeorge Floyd's
struggle with three police officers trying to arrest him, seen on body-camera
video, included Floyd's panicky cries of "I'm sorry, I'm sorry" and "I'm
claustrophobic!" as the officers tried to push Floyd into the back of a police
car.
Biden announces huge 'once-in-a-generation' infrastructure plan
Google Maps to default to routes with the lowest carbon impactThe tech giant is
using AI to improve its popular maps app to help combat climate change.
Three months on, what impact has Brexit had on UK-EU trade?
```

Code 2: Initial Dataset of Interest Topic 'Car'

b. Web Scraper

The web scraper collects text dataset about users' interests. It scrapes titles and summary of articles on euronews.com. The website was chosen because it publishes news with a great coverage of the topic and holds a simple structure favorable to web-scraping. Most of the features are as standard web scrapers except a little twist in the beginning to click a button to agree for cookie collection. Per topic, it scrapes 200 pages, which results in approximately 1,600 lines of text. Once the scraping is complete, it saves the text data into a designated folder.

The program is scalable to new users and corresponding new interests. Though three users, their interests, and the scraped text data is already stored for quick start, a program user can always add a new user with new interests, and restart the web scraper. This feature also allows a user to update the text dataset even without any new user added. Further details of how to start web scraping will be discussed at ***4. Implementation***.

c. Document Preprocessor

Before converting the sentences into feature vectors, they need to be polished for better accuracy. Here are four strategies of preprocessing. You may find the code in **`data_preprocessor.py`.**

- i. Reformat
  1. lower the capital letters
  2. remove the white spaces
  3. remove numbers: Since the training text was scraped from an online newspaper, they contain datetimes, and numerical data to be factually precise. They are semantically weakening the model if left.
- ii. Tokenize: Divide each sentence into lists of words
- iii. Remove Stopwords: This project utilized the stop words dataset from **`nltk`**'s english stopwords dataset.
- iv. Stem: Stemmer returns a root of the input word.

d. Word Embedding

After preprocessing, the input sentences are in a format of list of strings. To represent them in a more semantically meaningful format, TF-IDF is implemented. By adjusting TF-IDF, we can expect the learning process would be more vulnerable to a document with more occurence of the query and scarce vocabularies. Code-wisely, the lists were vectorized by `CounterVectorizer` and transformed by `TfidfTransformer` in the `sklearn` library.

e. Machine Learning Classifiers

Once the preprocessed query and documents for training dataset is in place, the classifiers are ready to train and predict. Among many classifiers, four were tested: Naive Bayes Classifier, SVM Classifier, Decision Tree Classifier, Random Forest Classifier). Characteristics and design of each classifier model is not described here as they are out of scope in this project.

To wrap up the process so far, each model will train on the dataset (i) scraped from a newspaper website (ii) labeled with interest, (iii) preprocess semantically, and (iv) transformed into TF-IDF vectors.

As consistent machine learning models, they share a great deal of structure in common such as following a pipeline of stages, training on data, prediction on test data, and saving and loading the trained model. In that, they were implemented in a group of subclasses. To run these four models sequentially, Factory Design Pattern was used. Further description will be discussed at ***4. Implementation***.

In regard to the feature to save and load the trained models, it is not significantly efficient at this moment because the training process takes a few seconds with the current dataset (approx. ~6400 sentences). However, the feature will be computationally helpful when the system is expected to hold a greater volume of dataset and more number of users with more interests.
You may find the examples of the prediction and accuracy at ***5.Results***.

f. How to run the program

This is a guideline to run a program with options and features. Please refer to `README.md` for installation guidelines. At the beginning of the program, The program allows the user 3 features. You may add `--help` when running the program to see all the options available.

i. Re-training of the dataset: argument `--retrain`

- The program is designed to start training when it cannot find a saved model. When it has a saved model already, it does not train each time and simply predict based on the saved model. The argument is expected to be used when a user modified or optimized a machine learning model and would like to retrain the models. When re-train is finished, the program shows the accuracy of each new-ly trained model as below.

```
[NaiveBayes] Accuracy: 0.5762013729977117
[SVM] Accuracy: 0.5624713958810069
[DecisionTree] Accuracy: 0.3327231121281464
[RandomForest] Accuracy: 0.4956521739130435
```

ii. Re-building of the dataset: argument `--rebuild`

- Re-building has two features: re-scraping the training dataset and re-training the machine learning. The program scrapes a newspaper website, which is highly likely to be out-of-date when the time goes by. When a user wants to update the training dataset up-to-date, one can run re-building and the program will re-scrape and re-train based on the newly scraped dataset.

iii. Adding a new user with new interests: argument `--newuser`

- Initially the program holds 4 users with exemplary interests. If a user would like to add a new user with new interest, they can add this argument. This argument will prompt a guideline to enter interests divided by commas(,). When a user types in the interests, the program checks if the new user has a new interest which has not been scrapped yet. If there are new topics to scrape, it starts to scrap, and train the machine learning models including the new topics just as `--rebuild` does.

Without the parameters, the program pops an input prompt to ask for a document to match users' interest as below.

```
Enter a document to match users' interests:
```

g. Output Printer

Without the parameters, the program pops an input prompt to ask for a document to match users' interest as below.
With a document typed in (in this case *'I want to buy audi, bmw, or cadillac'.*) The program returns users with the interest, and the interest topic. If the interest topic matches with multiple users, it returns all.

```
Enter a document to match users' interests: I want to buy audi,
bmw, or cadillac.

The matched user is 'user3' with interest in 'car'.
```

4. Implementation

The structure mainly consists of three parts: training data scraper, data preprocessor, and machine learning. When a user introduces a document into the service, the service scrapes articles about pre-assigned users' profiles (interests), preprocess the scraped data, train the data with the machine learning algorithms, and predict users who are likely to be interested in the document. During the process, it saves training data and trained models to improve efficiency on the next implementation. The overall flow of the program is as below:

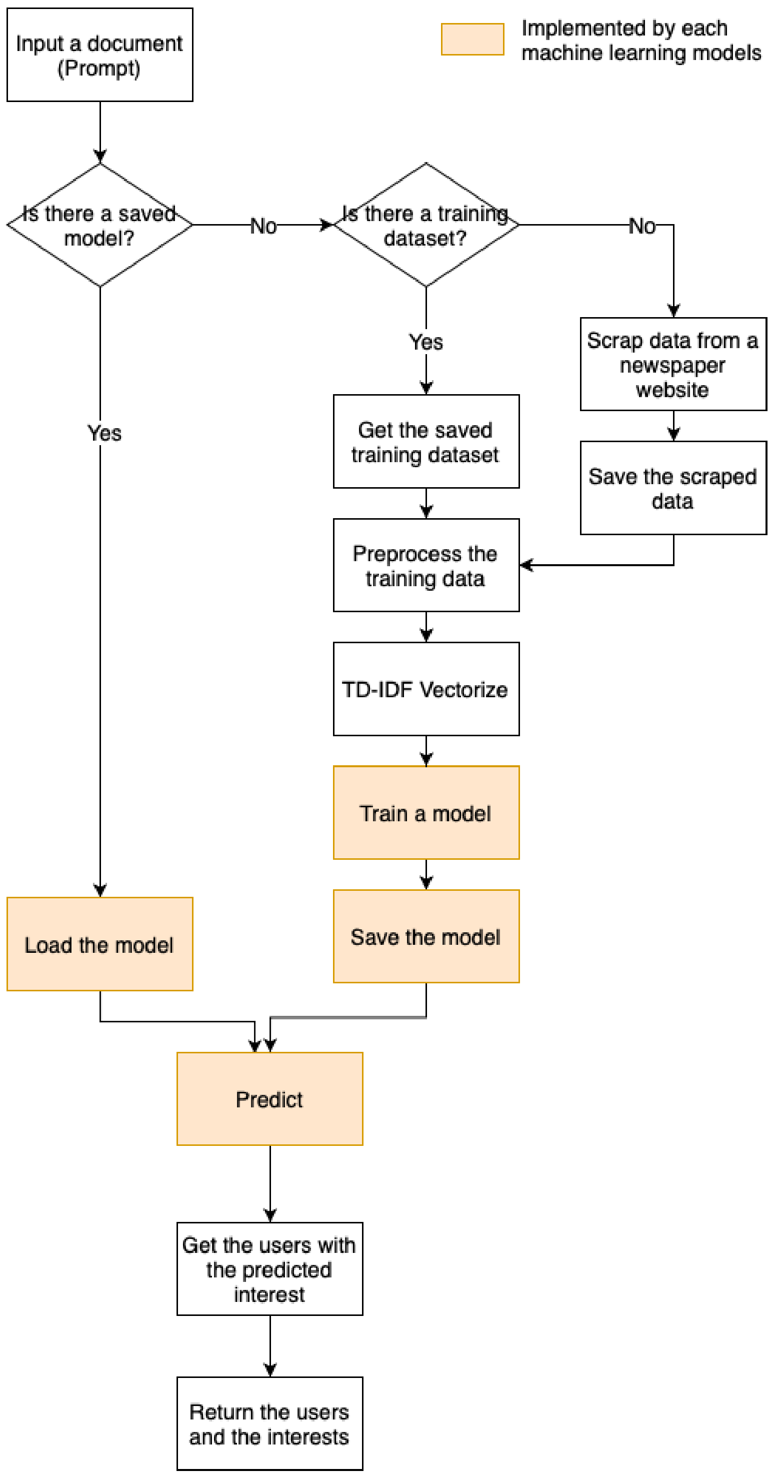
Input a document (Prompt)
Implemented by each machine learning models
Is there a saved model?
No
Is there a training dataset?
No
Yes
Scrap data from a newspaper website
Yes
Get the saved training dataset
Save the scraped data
Preprocess the training data
TD-IDF Vectorize
Train a model
Load the model
Save the model
Predict
Get the users with the predicted interest
Return the users and the interests

Figure 1. Flow of the Program

a. Training Data Scraper (`data_builder.py`)

The first and essential part of the program is the web scraper. It scraps Euronews by searching each interest as a topic for 20 pages. You may refer to code down below.

```python
class DataBuilder:

   def crawl_euronews(self, topic: str):
       chrome_options = webdriver.ChromeOptions()
       chrome_options.add_argument("--headless")

       driver = webdriver.Chrome(options=chrome_options)
       driver.delete_all_cookies()

       url_head = "https://www.euronews.com/search?query=" + topic
       data = []
       n_pages = 100
       FIRST_PAGE = True

       for page in range(1, n_pages):
           url = url_head + "&p=" + str(page)
           driver.get(url)

           # handle 'Accept-Cookie' popup
           if FIRST_PAGE:
               button = driver.find_element_by_id("didomi-notice-agree-button")
               button.click()
               FIRST_PAGE = False

           # crawl title and short summary of the articles
           soup = BeautifulSoup(driver.page_source, 'html.parser')
           object_bodies = soup.find_all('div', {"class": "m-object__body"})
           for each in object_bodies:
               title = each.find(class_="m-object__title__link")
               contents = each.find(class_="m-object__description")
               title = title.text.strip()
               if contents:
                   title += contents.text.strip()
               data.append(title)
       driver.quit()
       filepath = os.path.join(DATA_SAVE_DIR, topic)
       save_data(filepath, data)

   def build_training_data(self, new_topics=None):
       topics = interest_integrater.interests

       if new_topics:
           topics = [each for each in new_topics if each not in 
interest_integrater.interests]
       for each in topics:
           print("Crawling about {}... Please wait...".format(each))
           self.crawl_euronews(each)
```

b. Data Preprocessor (data_preprocessor.py)

The data preprocessor prepares the scraped text by tokenizing, removing stop words and stemming.

```python
def formatize(query: str) -> str:
   query = query.lower()  # lower the capital letters
   query = query.strip()  # remove whitespace
   query = re.sub(r"[0-9]\w+|[0-9]", "", query)
   return query


def tokenize_words(query: str) -> list:
   tokens = word_tokenize(query)
   alphabet_pattern = re.compile('[a-zA-Z]+')
   alpha_only = []
   for each in tokens:
       if alphabet_pattern.match(each) != None:
           alpha_only.append(each)
   return alpha_only


def stemmer(query: list) -> list:
   ps = PorterStemmer()
   return [ps.stem(each) for each in query]


def remove_stopwords(query: list) -> list:
   to_remove = stopwords.words('english')
   return [each for each in query if each not in to_remove]


def preprocess_sentence(sentence: str) -> list:
   sentence = formatize(sentence)
   tokens = tokenize_words(sentence)
   tokens = remove_stopwords(tokens)
   tokens = stemmer(tokens)
   return tokens
```

c. Machine Learning (ml_models.py)

Since all the classifiers are a part of the sklearn library, they share similar architecture. Therefore, features in common were made as a parent class called MachineLearningModel.

```python
class MachineLearningModel:
   def __init__(self, model_name: str, pipeline: Pipeline):
       self.trained_model = None
       self.model_name = model_name
       self.model_fp = os.path.join('models', self.model_name)
```

```python
        self.pipeline = pipeline

    def load_model(self):
        return pickle.load(open(self.model_fp, 'rb'))

    def train_model(self, x_train, x_test, y_train, y_test):
        model = self.pipeline
        model.fit(x_train, y_train)
        self.trained_model = model
        predicted = model.predict(x_test)
        accuracy = np.mean(predicted == y_test)
        print("[{}] Accuracy: {}".format(self.model_name, accuracy))
        self.save_model(model)
        return model

    def predict(self, doc):
        model = self.trained_model or self.load_model()
        return model.predict([doc])

    def get_trained_model(self, x_train, y_train, x_test, y_test):
        try:
            return self.load_model()
        except FileNotFoundError:
            return self.train_model(x_train, y_train, x_test, y_test)
        except TypeError:
            return self.train_model(x_train, y_train, x_test, y_test)

    def save_model(self, model):
        with open(self.model_fp, "wb") as f:
            pickle.dump(model, f)
```

```python
NaiveBayesClassifier = MachineLearningModel('NaiveBayes',
Pipeline([('count_vec', CountVectorizer()), ('tfidf', TfidfTransformer()),
('nb', MultinomialNB(fit_prior=False))]))

SVMClassifier = MachineLearningModel('SVM',
Pipeline([('count_vec', CountVectorizer()), ('tfidf', TfidfTransformer()),
('svm', SGDClassifier(random_state=1))]))

DecisionTree = MachineLearningModel('DecisionTree',
Pipeline([('count_vec', CountVectorizer()), ('tfidf', TfidfTransformer()),
('dt', DecisionTreeClassifier(max_depth=8,
random_state=4))]))

RandomForest = MachineLearningModel('RandomForest', Pipeline([('count_vec',
CountVectorizer()), ('tfidf', TfidfTransformer()), ('rf',
RandomForestClassifier(max_depth=10,
random_state=4))]))
```

Once the details of the pipeline were decided, each classifier in the format of subclass was appended into one list and added on a Factory model called `MachineLearningModelFactory`. Other layers would only access the Factory model to train or predict a model, and instead the Factory model runs the functions of all the machine learning models as a whole. When predicting the final label, it collects predictions of each model and decide with the most counts.

```
class MachineLearningModelFactory:
   def __init__(self, ml_models):
       self.ml_models = ml_models

   def train_model(self, x_train, x_test, y_train, y_test):
       for each in self.ml_models:
           each.train_model(x_train, x_test, y_train, y_test)

   def predict(self, doc):
       preds = []
       for each in self.ml_models:
           each_pred = each.predict(doc)
           preds.append(each_pred)
       return max(preds, key=preds.count)


ml_factory = MachineLearningModelFactory([NaiveBayesClassifier,
                                         SVMClassifier,
                                         DecisionTree,
                                         RandomForest])
```

You can find the code here: *https://github.com/soomin-jeong/profile-based-information-retrieval*

5. Results

The system may improve the accuracy by scraping more text data from more sources, and tuning the parameters of each machine learning model. With the minimum tuning on each model and the current dataset, the accuracy is as below:

Classifier	**Accuracy**
Naive Bayes	57.62%
SVM	56.24%
Decision Tree	33.27%
Random Forest	49.56%

Table 1. Accuracy of each model

6. Conclusion

What differentiates this project from a simple text classification is how to design the system to understand a user's taste. Under assumption that we can summarize one's interests and tastes into a few words as 'car', 'politics', or 'soccer', the barrier lies in how to understand these simple words and match with the long document. Solving this part was a deeply intriguing journey.

My solution was to scrap a newspaper website. The reason why the dataset could lead to a significant result is supposedly that newspapers represent natural language. They are written with the most up-to-date issues, in a natural and concise tone, and by humans. In that the queries will be inserted by a human as well, the dataset well covers the expected input.

Though some of the programming principles are not acknowledged for data science projects, I would like to make a humble recognition that I tried to write the code in a reusable and understandable way. I hope the code would be read by a few more people than only myself.

7. Reference

Arezki, R., Poncelet, P., Dray, G., & Pearson, D. W. (2004). Information retrieval model based on user profile. Artificial Intelligence: Methodology, Systems, and Applications, 490-499. doi:10.1007/978-3-540-30106-6_50

Arezki, R., Poncelet, P., Dray, G., & Pearson, D. W. (2004). Web information retrieval based on user profile. Lecture Notes in Computer Science, 275-278. doi:10.1007/978-3-540-27780-4_31

Chen, P., & Kuo, F. (2000). An information retrieval system based on a user profile. Journal of Systems and Software,54(1), 3-8. doi:10.1016/s0164-1212(00)00021-2

Engström, E., & Strimling, P. (2020). Deep learning diffusion by infusion into preexisting technologies – implications for users and society at large. Technology in Society, 63, 101396. doi:10.1016/j.techsoc.2020.101396