Travel Agency

Travel Agency Project has consist of recommendation algorithm that can be used to recommend travel destinations to user.

Project is using Django Framework for easy communication between backend and frontend. SQL is decided to be used for the storage and since project is based o Django, MySQL has being used for the project.

The aim of the project is to create a recommendation as a service system, where the system will be tested on an example which in our case is travel agency domain. The project is designed to be as generic as possible, so it can be used for other domains too.

Python has being used for frontend and backend parts of the project.

Content

In this document, you will find information regarding how system is working, which recommendation algorithms are implemented and what kind of parameters they need.

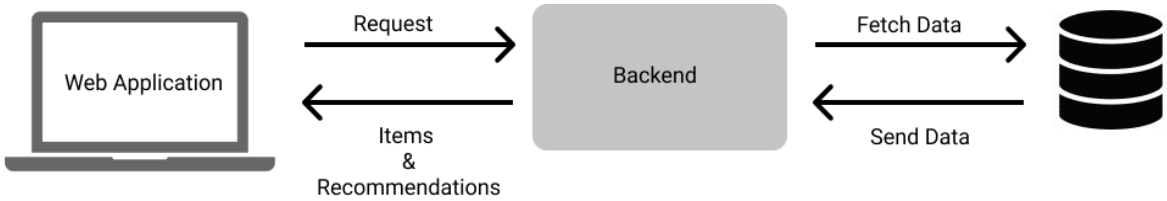
Created documentations so far:

- README.md
- travel_agency.md
    - Models
    - Views
    - Doc2Vec
    - Item2Vec
    - Rnn
- helper_scripts.md
    - DatabaseOperations
    - UtilityFunctions
    - PreProcess

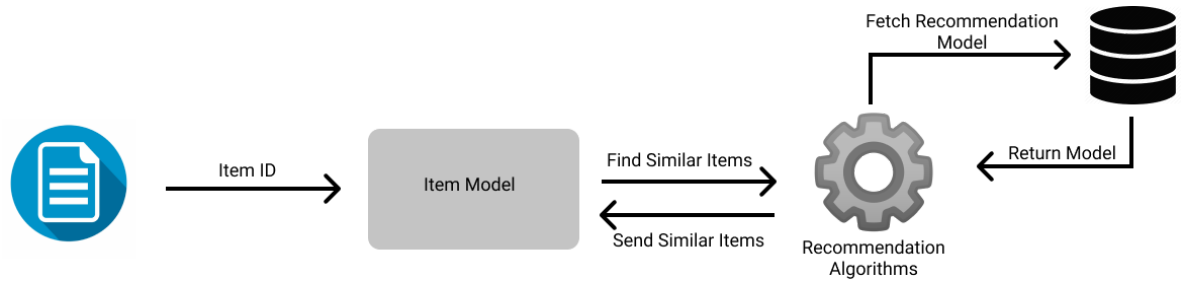
Work Flow

Below there are 2 high level diagram to understand how system is working.

High Level System Diagram



High Level Backend Diagram



Shortly, the general flow of the web application is:

- User clicks on an item.
- Item id is send to Item Model in the backend.
- Item Model runs recommendation methods (e.g item2vec_recommend()) which will
  - Find and load the current recommendation model from the database.
  - Find the similar items based on the given item id.
  - Return topn recommended item.
- Return recommended item list to the frontend.

Models

A model is the single, definitive source of information about our data. It contains the essential fields and behaviors of the data we are storing. Generally, each model maps to a single database table.

In our case, we have two models to work on:

- Item
- User

Item Model

Item model has 3 fields named as:

- **object_id:** ID of the travel item
- **attribute:** An attribute for travel item. (e.g. price, destination, length etc.)
- **value:** Value of the item attribute (e.g. 500, Warsaw, 30)

```
class Item(models.Model):
    object_id = models.IntegerField(null=False, unique=False)
    attribute = models.TextField(max_length=75, unique=False, null=False)
    value     = models.TextField(max_length=3000, unique=False, blank=True)
```

There are currently 3 implementations of getting a recommendation. Each recommender algorithm has unique tag which is shown below:

1. Item2Vec = 'i2v'
2. Doc2Vec = 'd2v'
3. Recurrent Neural Network (Sequence Prediction) = 'rnn'

Every recommendation model has active state in the database to make the model current recommender. Item class has recommendation_algorithm setter to change current recommendation model. One must call the setter and give the recommendation model tag.

Additionally, Item Class has a recommendation method to find recommendation for the item that user currently checking.

```
def recommend_item():
```

Summary

Train and save Doc2Vec Model.

Workflow

- Get current recommendation and call corresponding recommendation algorithm.

Return

Method will update the current recommendation method (change its active state to True in database) and return corresponding recommendations.

(This method should only return the recommendations and not change current recommender. However, selecting a recommender based on user id functionality has not been implemented. Thus, I am setting an active recommender model here.)

```
def doc2vec_recommend():
```

Summary

Find recommendations by using Doc2Vec Model.

Workflow

- Find Doc2Vec Recommendations.
- Create an item dictionary to be sent to front end

Return

List of recommended items

```
def item2vec_recommend():
```

Summary

Find recommendations by using Item2Vec Model.

Workflow

- Find Item2Vec Recommendations.
- Create an item dictionary to be sent to front end

Return

List of recommended items

```
def rnn_recommend():
```

Summary

Find recommendations by using RNN Model.

Workflow

- Find RNN Recommendations.
- Create an item dictionary to be sent to front end

Return

List of recommended item

User Model

User Model has 3 fields named as:

- **user_id:** ID of the user.
- **object_id:** ID of the item that user clicked on.
- **last_modified:** Timestamp of the user click.

```
class Item(models.Model):
    user_id = models.IntegerField(null=False, unique=False)
    object_id = models.IntegerField(unique=False, null=False)
    last_modified = models.DateField(unique=False, blank=True)
```

Views

Django views are a key component of applications built with the framework. At their simplest they are a Python function or class that takes a web request and return a web response. **Views** are used to do things like fetch objects from the database, modify those objects if needed, render forms, return HTML, and much more.

In our case, we have two views to work on:

- Home View
- Item Detail View

Home View

A view for Home page of the web application.

```
model = Item
paginate_by = 6
template_name = "home.html"
ordering = ['id']
Item.recommendation_algorithm = 'i2v'
```

View takes Item class and add as its context. The page is paginated by 6 items which means that there will be new page added after every 6th item.

View uses 'home.html' as template.

Items are ordered by their ids.

Default recommender is 'i2v'

```
# TODO: We can change the recommendation algorithm based on user id.
# TODO: Might be better to order items by the date which they are added to DB
```

Home View has only one method which is named as **get_context_data**

Context Data

Since Home View is using Item model, we can access all the items in database from this class. However, those items are stored in triplestore(?) structure and must be pre processed before used in the front end.

That is why we need create a 'destination_dictionary' and add it into context of the web page.

```
def get_context_data(self, **kwargs):
    # Get the context of the page and create a dictionary which will have
information for the items on the homepage.
    context = super().get_context_data(**kwargs)
    context['destination_dictionary'] = create_item_dict(context, False)

    return context
```

create_item_dict(context, False) method is explained

Item Detail View

A view to represent single item with its details.

```
    model = Item
    template_name = "item.html"
```

Yes, it is just 2 lines of code. It uses Item model and 'item.html' as its template page.

Django does the rest of the work by using its 'magic'.

Like in Home View, we need to pre process item details in order to use in front end. We will use the same technique like we used in Home View.

```
def get_context_data(self, **kwargs):
    # Get the context of the page and create an item dictionary for a specific
item that    user is currently visiting.
    context = super().get_context_data(**kwargs)
    context['destination_dictionary'] = create_item_dict(context, True)

    return context
```

create_item_dict(context, True) method is explained

Recommendation Algorithms

Doc2Vec

Word2Vec is a well known concept, used to generate representation vectors out of words.

In our case, we have multiple values for each item. We can merge those attributes and create one sentence for each item. Then we can train a Doc2Vec Model to use for recommendations.

Below, you can find detailed information of the Doc2Vec Implementation to the Travel Agency Project.

Methods

Implementation consists of 2 main methods, named as:

- train_doc2vec
- doc2vec_calculate_similarity

and 2 helper methods, named as:

- find_vector_representation
- read_corpus

Train Item2Vec Model

```
def train_doc2vec():
```

Summary

Train and save Doc2Vec Model.

Workflow

- Convert nested item dictionary to dataframe
- Use the clean text dataframe and create corpus to train Doc2Vec Model.
- Create a Doc2Vec model and build a vocabulary by using the corpus.
- Save Doc2Vec Model and Vocabulary to database
- *Important*: Create **items.pickle** file. We need to have doc_vectors to find similar items, and that is why Doc2Vec Vocabulary must be saved to database.
    - **items.pickle** structure as follows:

        { 'index': { 'doc_tag' : [doc_vector, words] } }

Return

Model is saved to database.

Calculate Similarity Scores

```
def doc2vec_calculate_similarity(_item_id):
```

Summary

Creates a recommendation list that contains ids of the item that is similar to given id.

Workflow

- Load the Doc2Vec Model and Doc2Vec Vocabulary from database.
- Find the vector representation of the **_item_id** by using **find_vector_representation()** method. Create a recommendation list and start appending predicted item ids (Note that some item ids might not be in the database)
- Note that vector representation must be converted to numpy array before being sent to the model.
- Find the similar items.
- Get the similar doc ids and find the corresponding item ids.

Return

List of item ids to use for recommendation

Finding Vector Representation of Items

```
def find_vector_representation(_item_id, _items):
```

Summary

Finds the given item in the item vocabulary.

Return

Vector Representation of the item.

Read Corpus

```
def read_corpus(df):
```

Summary

Tag the documents to be used in Doc2Vec Model Training.

Workflow

- Iterate through the item documents and tag them by using item ids.

Return

Tagged documents

Item2Vec

Item2Vec produces embedding for items in a latent space. The method is capable of inferring item-item relations even when user information is not available.

The difference between Item2Vec and Word2Vec is that instead of having **words**, we have **items** and instead of having **sentences**, we have **item sequences**.

Below, you can find detailed information of the Item2Vec Implementation to the Travel Agency Project.

Methods

Implementation consists of 2 main methods, named as:

- train_item2vec
- item2vec_calculate_similarity

Train Item2Vec Model

```
def train_item2vec():
```

Summary

Train and save item2vec Model.

Workflow

- In order to use items as "words" we need to find user sequences. Currently, this is done by using Pandas:
  - First, data must be grouped by user ids. Then, we need to have user sequences for individual users to put each item which is belong to that user.
- After creating the sequences, we can use the sequence data to feed item2vec model.
- Since item2vec is same as working with word2vec, we will just give user items in the user sequence to train our model.
- Save the model to database.

Return

Model is saved to database.

Calculate Similarity Scores

```
def item2vec_calculate_similarity(_item_id):
```

Summary

Creates a recommendation list that contains ids of the item that is similar to given id.

Workflow

- Load the item2vec model from database.
- Use model to find similar items to given item id.
- Create a recommendation list and start appending predicted item ids (Note that some item ids might not be in the database)
- Return the list.

Return

List of item ids to use for recommendation

Rnn

Rnn for Session Based Recommendation. Based on 'https://github.com/mquad/sars_tutorial'.

Below, you can find detailed information of the Rnn Implementation to the Travel Agency Project.

Methods

Implementation consists of 2 main methods, named as:

- train_rnn
- rnn_calculate_similarity

Train RNN Model

```
def train_rnn():
```

Summary

Train and save Rnn Model.

Workflow

- Create a RnnRecommender object.
- Prepare the training and testing data.
- Parse training data to GRU4Rec Format.
- Fit the model.
- Save the model to database.

Return

Model is saved to database.

Calculate Similarity Scores

```
def item2vec_calculate_similarity(_item_id):
```

Summary

Creates a recommendation list that contains ids of the item that is similar to given id.

Workflow

- Load the Rnn model from database.
- Use model to find similar items to given item id.
- Create a recommendation list and start appending predicted item ids
- Return the list.

Return

List of item ids to use for recommendation