# recsplain

*Release 0.1*

**ArgMaxML**

**Nov 24, 2022**

# CONTENTS

The Recsplain System makes recommendations and explains them.

It recommends items based on item similarity or user preferences. It explains the recommendations in terms of overall similarity and feature-to-feature similarity.

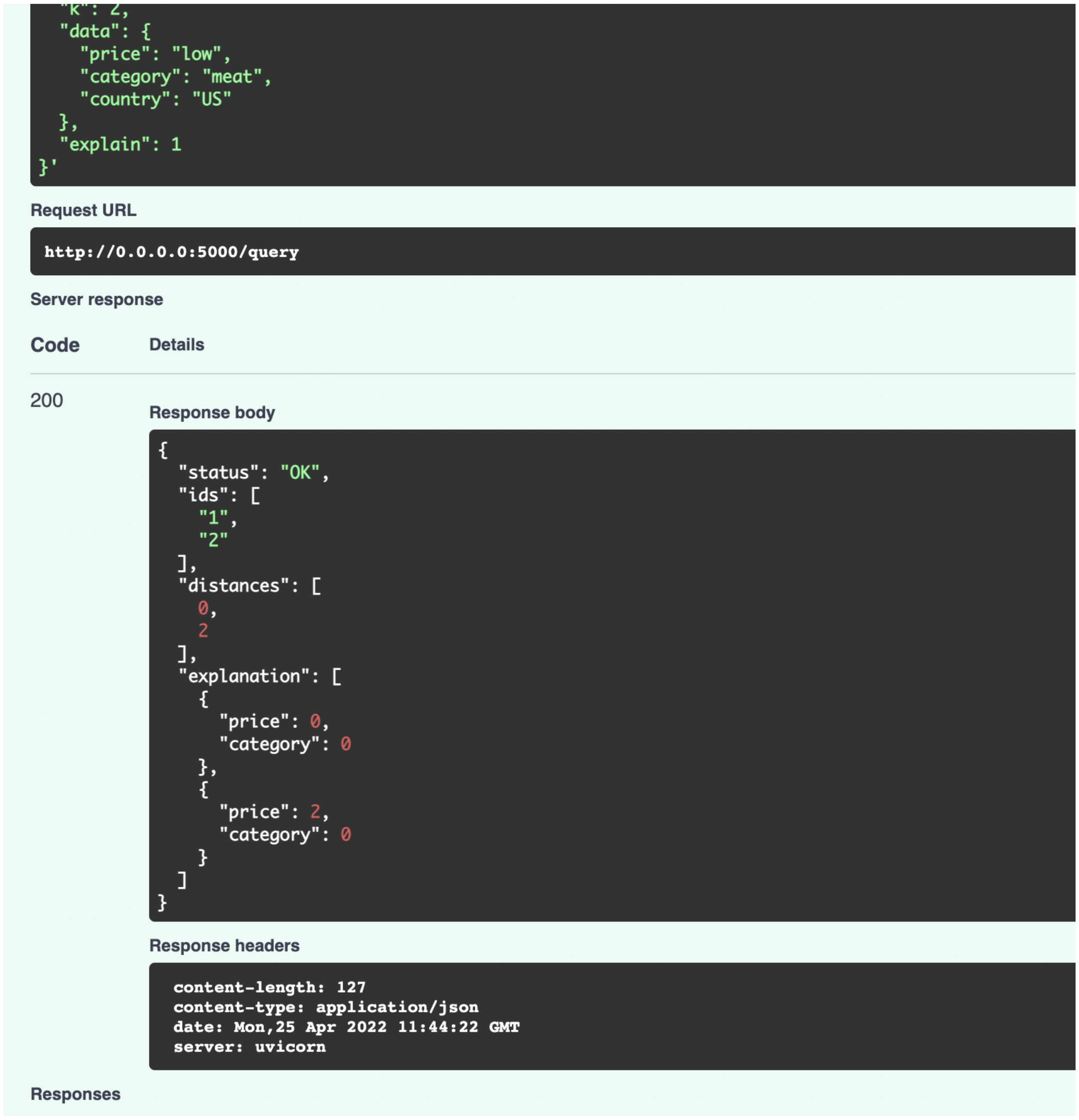
Install it in your app, use it with your data, and customize it how you want.

CHAPTER

ONE

# EXPLAINABLE RECOMMENDATIONS

Here is an example item similarity search. You can see the request and response in the image below.



```
  "data": {
    "price": "low",
    "category": "meat",
    "country": "US"
  },
  "explain": 1
}'
```

Request URL

```
http://0.0.0.0:5000/query
```

Server response

Code Details

200

Response body

```
{
  "status": "OK",
  "ids": [
    "1",
    "2"
  ],
  "distances": [
    0,
    2
  ],
  "explanation": [
    {
      "price": 0,
      "category": 0
    },
    {
      "price": 2,
      "category": 0
    }
  ]
}
```

Response headers

```
 content-length: 127
 content-type: application/json
 date: Mon,25 Apr 2022 11:44:22 GMT
 server: uvicorn
```

Responses

The request is based on a search item that has three features. It is a US-based product in the meat category and low in price.

```
import recsplain as rx

item_query_data = {
  "k": 2,
  "data": {
    "price": "low",
    "category": "meat",
    "country": "US"
  },
```

(continues on next page)

(continued from previous page)

```
  "explain": 1
}

rec_strategy.query(**item_query_data)
```

The response body in the image above contains the recommendations and explanations.

The ids are ordered by index position from most to least recommended. The lowest index position is the most recommended.

```
{
  "status": "OK",
  "ids": ["1", "2"],
  "distances": [0, 2],
  "explanation": [
    {
      "price": 0,
      "category": 0
    },
    {
      "price": 2,
      "category": 0
    }
  ]
}
```

Distances explain item similarity based on all features and weights. Explanations provide distances for each feature. The distances and explanations correspond to the ids by index position.

Lower values correspond to greater similarity.

In the example, the system recommends item 1 more than item 2 because item 1 has a lower distance.

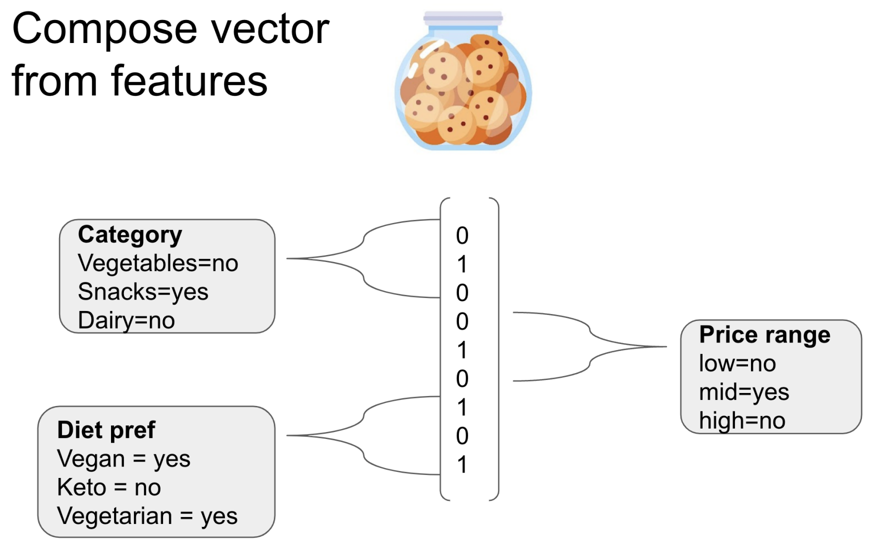
Item 1 has a lower distance because it has a lower distance for price than B and they are equal distance in category.

CHAPTER

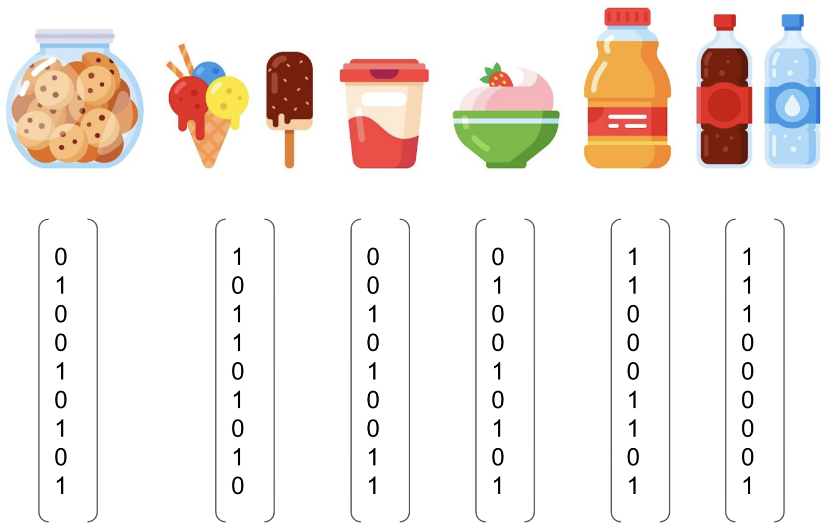
# TWO

## HOW IT WORKS

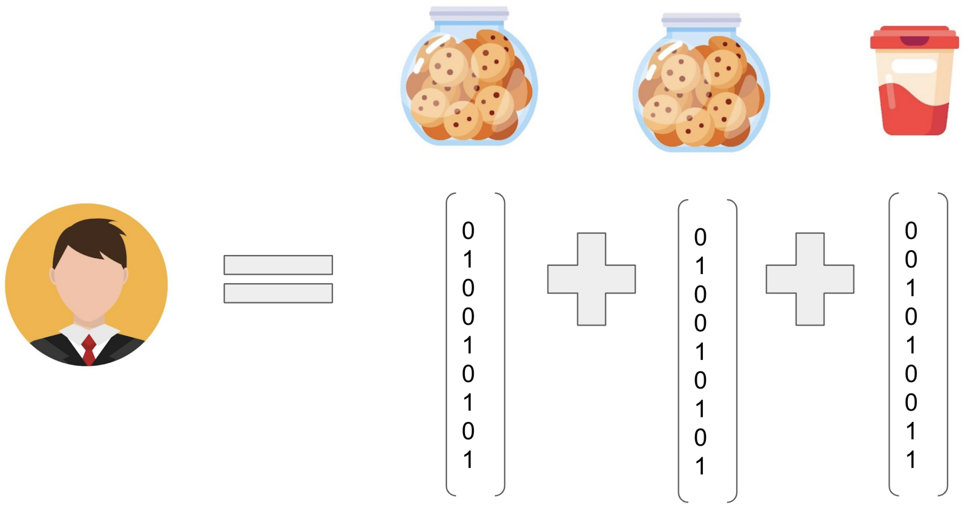
For item similarity, Recsplain turns items into weighted feature vectors.



The system compares item feature vectors to one another to calculate how similar they are.



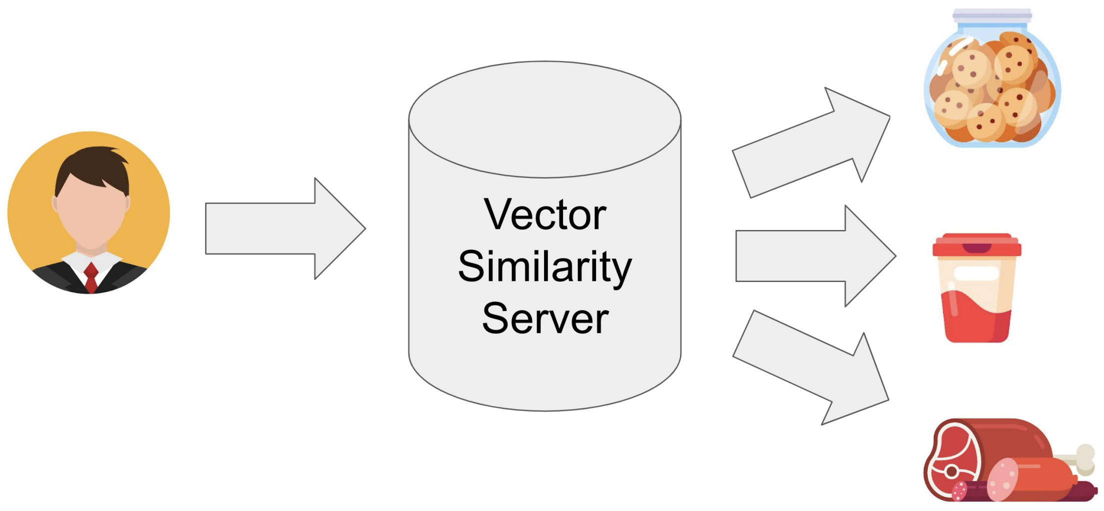
For user preferences, Recsplain turns a user into an item feature vector based on the user's previous history with the items.



For a user that bought cookies twice and a coffee once

**Note:** For example, a customer of an online store who bought two cookies and a glass of milk has an item feature vector that is a blend of the item vectors for two cookies and milk.

The system compares the user's item feature vector to the indexed item feature vectors to calculate how similar the items are those the items in the user's history. The more similar, the higher the recommendation.



**Note:** To a customer who previously bought two cookies and a glass of milk, the system recommends other items that have similar features to those purchases.

CHAPTER

THREE

# FIELD TYPES & SCHEMA

Use the field types and schema to configure the Recsplain filters and encoders.

Filters determine which items are compared to one another. Encoders determine how they are compared.

Here is an example configuration.

```
import recsplain as rx

config_data = {
  "filters": [{ "field": "country", "values": ["US", "EU"] }],
  "encoders": [
    {
      "field": "price",
      "values": ["low", "mid", "high"],
      "type": "onehot",
      "weight": 1
    },
    {
      "field": "category",
      "values": ["dairy", "meat"],
      "type": "onehot",
      "weight": 2
    }
  ],
  "metric": "l2"
}

rec_strategy = rx.AvgUserStrategy()
rec_strategy.init_schema(**config_data)
```

**Filter Fields**

The filter fields are hard filters. They separate items into different partitions. Only items within the same partition are compared to one another.

The example above creates two partitions. One for US items and another for EU.

**Encoder Fields**

The encoder fields are soft filters for fuzzy matching. They determine how item features are compared within a partition.

The example above selects the one-hot encoder for each of the item features.

**Note:** Learn more about the one-hot and other available *Encoders*.

### User Encoders

When recommending items for a user, Recsplain has special encoders you should use. Currently, user encoders encode the user's history into a feature vector.

**Note:** **ArgMaxML** created Recsplain. We are focused on creating software the enables you to integrate recommendation engines into your product to increase customer engagement.

CHAPTER

FOUR

# GET STARTED

Follow the guide below to start making explainable recommendations with Recsplain.

Start with:

- *Installation*
- *Configuration*
- *Index*

Then start searching by:

- *Item Similarity*
- *User Preference*

**Note:** Learn more about the methods in the *Reference* .

## 4.1 Installation

Import the package using the following import statement.

```
import recsplain as rx
```

**Note:** Learn more about the method in the *Installation* reference.

## 4.2 Configuration

First, you need to do is to configure the user recommendation strategy by using:

```
rec_strategy = rx.AvgUserStrategy()
```

**Note:** More strategies to come.

Use the `init_schema` method to configure the system so that it knows how to partition and compare feature vectors.

Here is an example of how to call the `init_schema` method.

```
import recsplain as rx

config_data = {
  "filters": [{ "field": "country", "values": ["US", "EU"] }],
  "encoders": [
    {
      "field": "price",
      "values": ["low", "mid", "high"],
      "type": "onehot",
      "weight": 1
    },
    {
      "field": "category",
      "values": ["dairy", "meat"],
      "type": "onehot",
      "weight": 2
    }
  ],
  "metric": "l2"
}

rec_strategy = rx.AvgUserStrategy()
rec_strategy.init_schema(**config_data)
```

`Weights` are used to set the relative importance the system should attribute to this feature in the similarity check.

This is the response from `init_schema`. The first element is an array of the filters created and the second element is a dictionary of the features and their corresponding vector size.

```
[('US',), ('EU',)], {'price': 4, 'category': 3}
```

**Note:** Encoder type one-hot save one spot for unknown `feature_sizes` so the size is N + 1.

**Note:** Learn more about the method in the *configuration* reference.

## 4.3 Index

Use the `index` method to add items to the Recsplain system so that it has items to partition and compare.

Here is an example of how to call the `index` method.

```
import recsplain as rx

config_data = {
  "filters": [{ "field": "country", "values": ["US", "EU"] }],
  "encoders": [
    {
      "field": "price",
      "values": ["low", "mid", "high"],
```

(continues on next page)

(continued from previous page)

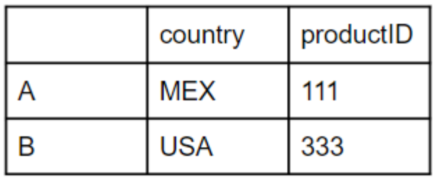
```
      "type": "onehot",
      "weight": 1
    },
    {
      "field": "category",
      "values": ["dairy", "meat"],
      "type": "onehot",
      "weight": 2
    }
  ],
  "metric": "l2"
}

index_data = [
  {
    "id": "1",
    "price": "low",
    "category": "meat",
    "country": "US"
  },
  {
    "id": "2",
    "price": "mid",
    "category": "meat",
    "country": "US"
  },
  {
    "id": "3",
    "price": "low",
    "category": "dairy",
    "country": "US"
  },
  {
    "id": "4",
    "price": "high",
    "category": "meat",
    "country": "EU"
  }
]

rec_strategy = rx.AvgUserStrategy()
rec_strategy.init_schema(config_data)
rec_strategy.index(index_data)
```

This is the response from `index`. The first element is a list of errors and the second elemnt is the number of partitions affected by the indexing.

```
[], 2
```

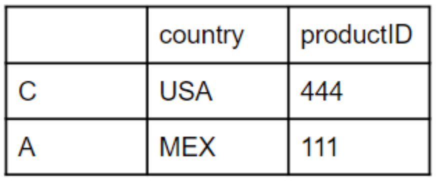
**Note:** If you do not index items, when you search there will be nothing to check the search against for similarity.

**Note:** When reusing the index method, using the same id twice creates duplicate entries in the index. In the example below the index method is called twice with the same entry. In the index table both entries are created.



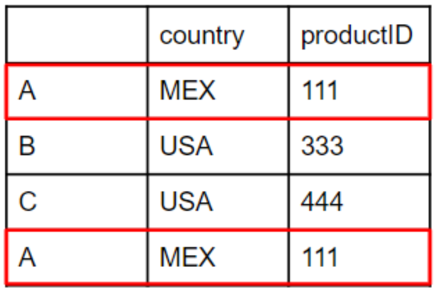
| | country | productID |
|---|---|---|
| A | MEX | 111 |
| B | USA | 333 |

Table 1



| | country | productID |
|---|---|---|
| C | USA | 444 |
| A | MEX | 111 |

Table 2



| | country | productID |
|---|---|---|
| A | MEX | 111 |
| B | USA | 333 |
| C | USA | 444 |
| A | MEX | 111 |

Index

**Note:** Learn more about the method in the *index* reference.

## 4.4 Item Similarity

Use the `query` method to search by item.

The method returns explainable recommendations for indexed items that are similar to the search item.

Here is an example of how to call the `query` method.

```
import recsplain as rx

item_query_data = {
  "k": 2,
  "data": {
    "price": "low",
    "category": "meat",
    "country": "US"
  },
  "explain": 1
```

(continues on next page)

(continued from previous page)

```
}

rec_strategy.query(**item_query_data)
```

This is the response from `query`. The first element is the ids of the recommended items, the second element is the distances of each of the recommended items and the third element is the explanation of how much each feature contributed to the overall distance.

```
('1', '2') (0.0, 2.0) [{'price': 0.0, 'category': 0.0}, {'price': 2.0, 'category': 0.0}]
```

**Note:** Learn more about the method in the *item similarity* reference.

## 4.5 User Preference

Use the `user_query` method to search by user.

The method returns explainable recommendations for indexed items that the user likely prefers.

Here is an example of how to call the `user_query` method.

```
import recsplain as rx

user_query_data = {
  "k": 2,
  "item_history": ["1", "3", "3"],
  "user_data": {
    "country": "US"
  }
}

rec_strategy.user_query(**user_query_data)
```

This is the response from `user_query`. The first element is the ids of the recommended items and the second element is the distance of each of these items from the user's representation (as given by the items history).

```
['3', '1'] [2.0, 2.0]
```

**Note:** Learn more about the method in the *user preference* reference.

CHAPTER

FIVE

# SAVING AND LOADING MODELS

Save and load models for your Recsplain system.

## 5.1 Save

Use the `save_model` method to save the model to your computer.

Here is an example of how to call the `save_model` method.

```
import recsplain as rx

model_name = "your-model-name"

rx.save_model(model_name)
```

It returns the saved model in JSON format.

## 5.2 Load

Use the `load_model` method to load a model to your system.

Here is an example of how to call the `load_model` method.

```
import recsplain as rx

model_name = "your-model-name"

rx.load_model(model_name)
```

CHAPTER

SIX

# SPINNING UP A SERVER

Use the Recsplain system as a web server. It allows you to run searches over the internet.

Send the search item or user in the payload of an HTTP request to your Recsplain server and get recommendations and explanations in response.

You also can use the web server to configure, index, and otherwise use the system.

## 6.1 Installation

Import the package using the following import statement.

```
import recsplain as rx
```

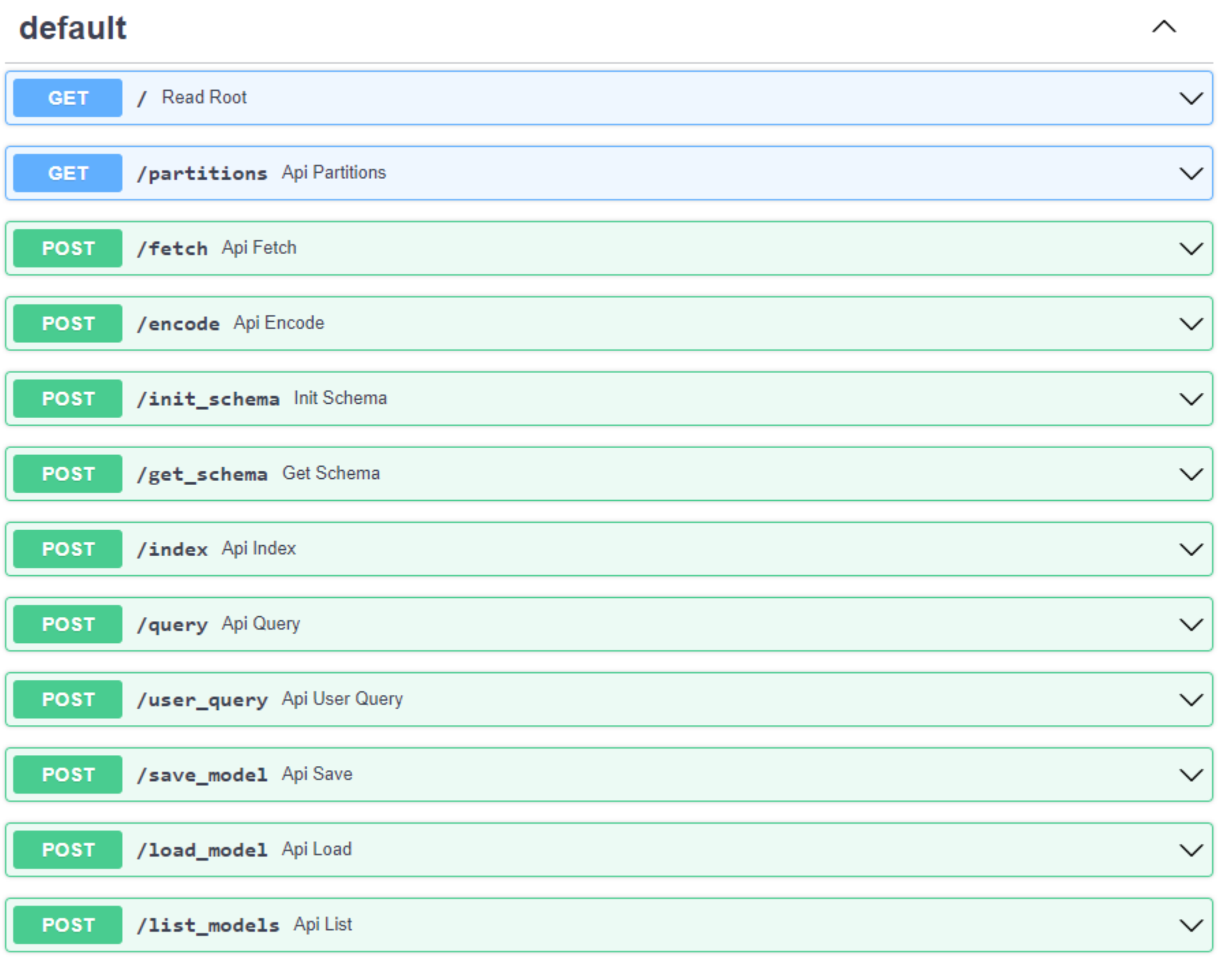
## 6.2 Running Server

To run the sever, enter the following command in your terminal.

```
python -m recsplain
```

Browse to http://127.0.0.1:5000/docs.

You should see a swagger interface for the REST API.


default

GET / Read Root

GET /partitions Api Partitions

POST /fetch Api Fetch

POST /encode Api Encode

POST /init_schema Init Schema

POST /get_schema Get Schema

POST /index Api Index

POST /query Api Query

POST /user_query Api User Query

POST /save_model Api Save

POST /load_model Api Load

POST /list_models Api List

## 6.3 Calling Server

Instead of calling the package methods, call the routes to index, configure, search, otherwise interact with the system.

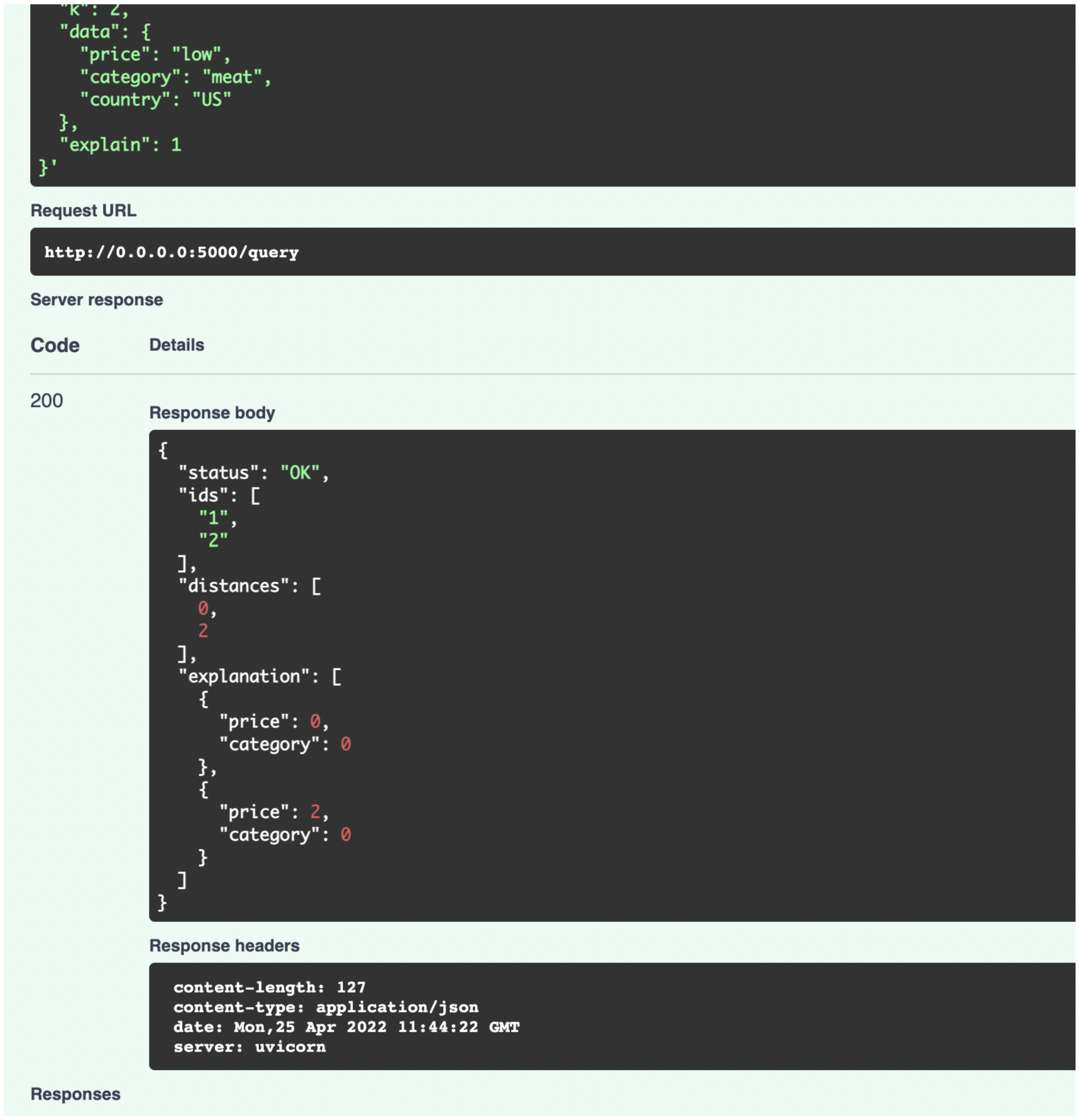
Follow the same steps as in the *Get Started* document for configuring and indexing before searching by item or user.

**Steps are:**

1. init_schema - create the schema
2. index_item - index items
3. index_user - index users

After you index and configure, send an item or user to the system and get explainable recommendations in response by using `query` and `user` Respectively.

Send data in the body of the HTTP requests and get data in the HTTP response body.



```
  "k": 2,
  "data": {
    "price": "low",
    "category": "meat",
    "country": "US"
  },
  "explain": 1
}'
```

Request URL

```
http://0.0.0.0:5000/query
```

Server response

Code Details

200

Response body

```
{
  "status": "OK",
  "ids": [
    "1",
    "2"
  ],
  "distances": [
    0,
    2
  ],
  "explanation": [
    {
      "price": 0,
      "category": 0
    },
    {
      "price": 2,
      "category": 0
    }
  ]
}
```

Response headers

```
content-length: 127
content-type: application/json
date: Mon,25 Apr 2022 11:44:22 GMT
server: uvicorn
```

Responses

CHAPTER

SEVEN

# REFERENCE

The reference supplements the *Welcome*, *Get Started*, *Saving and Loading Models*, *Spinning Up A Server*, and *Reference* guides.

Here is the Reference Table of Contents

## 7.1 Installation

Install the Recplain system and then use it as a web server or import it into your code and call the methods directly.

### 7.1.1 pip install

Install the Recsplain system from PyPI using `pip`.

```
pip install recsplain
```

After you install the package, use it by either:

- Running a server to call the functions from the rest API or
- By importing into your code using Python bindings

### 7.1.2 Run Server

To use the system as a web server, enter the following command in your terminal.

```
python -m recsplain
```

Browse to http://127.0.0.1:5000/docs.

You should see a swagger interface for the REST API.

## 7.2 Configure

Read below to learn more about the inputs and outputs.

### 7.2.1 Inputs

The `init_schema` method requires the following inputs:

- `filters`
- `encoders`
- `metric`

#### Filters

Filters control which items the system checks for similarity each time you run an item or user query.

As the example above demonstrates, each filter is comprised of a field and possible values for the field. The two most common hard filters are location and language.

**Note:** The schema needs to include all possible values for each encoder field.

**Note:** Each field should correspond to a field for the items in your item database and the values to possible values for those fields.

When you run the `init_schema` method, the system creates a partition for each filter value.

In each partition are the indexed items whose value for the filter field matches the value for that partition.

When you search, the system checks the search item or user against the items in a particular partition only if the search item's or user's value for that feature matches the partition's value.

**Note:** In the example above, the system creates two partitions. One for US items and another for EU. When searching, if the search item or user is based in the US, the system only searches the US partition, not the EU partition.

Therefore, `filters` are hard filters and are used to separate or exclude items for comparison.

#### Encoders

Encoders control how the system compares items within each partition.

As you see in the example above, each encoder is comprised of a field, possible values for the field, an encoder type, and a weight.

Here is what each does:

- `field`: a feature to use in the similarity check
- `values`: values to check for the field
- `type`: the type of encoder to use for checking similarity for this feature
- `weight`: the relative importance the system should attribute to this feature in the similarity check

Unlike the filters, the encoders are not hard filters and therefore do not play a role in creating the partitions.

Instead, the encoders are used when the user searches by item or user to find similar items.

They are soft filters that dictate how the system checks for similarity.

The encoder fields should be a field that the items in your database have or could have.

The values for each field in the encoder should be values that each item could potentially have for that field.

The type of encoder sets how the system calculates similarity.

**Note:** Check out the *list of encoders* to learn what encoders you can use and how they work.

The weight tells the system the relative importance of each feature in the encoder.

**Note:** In the example, category is twice as important as price.

#### Metric

Metric is the method to use when calculating the returned distance from the similarity server for each item.

#### types of matrics:

- `l2`: the default metric.
- `cosine`: the cosine metric.

### 7.2.2 Outputs

The `init_schema` method returns an object containing:

- `partitions`
- `vector_size`
- `feature_sizes`

#### Partitions

The `partitions` value is the number of partitions the system made based on your configuration.

When you index items, the items are added to the partitions only if the item meets the filter criteria.

**Note:** A partition is an instance of the similarity server.

As explained above, the number of partitions is based on the number of values `init_schema` has for `filters`.

### Feature Sizes

Each encoder has a feature size.

The feature size is the number of distinct feature values for each encoder, plus one. The plus one is to account for unknown feature values.

In the example above, the price encoder has three values: `["low", "mid", "high"]`.

Its feature size, therefore, is 4 because of its three values and the possibility for unknown values.

Similarly, the category feature size is 3 because of its two values and the possibility for an unknown.

### Vector Size

The vector size is the sum of the features sizes.

In the example above, the vector size is 7. Here is why. The the price encoder has 3 values and therefore a feature size of 4. The category encoder has 2 values and therefore a feature size of 3. Therefore, the overall feature size is 7.

### Total Items

The total items is the total number of items indexed.

**Note:** Learn more about *indexing items from your database*.

## 7.3 Index

Read below to learn more about the inputs and outputs.

### 7.3.1 Inputs

The `index` method requires that you input data for each item that you want in the system.

The data is an array of item objects.

Each item object should have an id and a field for each item feature.

The id should be a unique value and serves an important role in the similarity check and the results because the system uses the id in the similarity check and the id is how you identify the item in the results for each query.

Thererfore, the id should be a value that makes it easy to identify each item.

**Note:** For example, it is common to use the SKU number of a product as the value for the id.

Also, notice in the example that the fields other than the id appear as either a filter or encoder field in the `init_schema` example code.

**Note:** Check out the `init_schema` *example configuration*.

Call the `index` method as many times as you want. Each time you call it, the data you send is added to the existing data without replacing the existing data.

### 7.3.2 Outputs

The `index` method returns the number of affected partitions.

A partition is an affected partition if the system added the item to the partition.

An item is added only to the partitions that it matches. An item matches a partition if the item has the feature value corresponding to the partition filter field.

**Note:** Using the example from the *configuration* page, indexing an item sold only in the US would affect one partition, whereas indexing an item sold in both the US and EU would affect two partitions.

## 7.4 Item query

Read below to learn more about the inputs and outputs.

### 7.4.1 Inputs

The `query` method takes the following inputs:

- `k`
- `data`
- `explain`

#### k

The `k` value is the number of similar items you want the system to return.

#### data

The data is an object containing fields and values for the features of the item you are searching for.

**Note:** Like the item features in the filters and indexing stages, the search item data fields should correspond to a field in your item database.

#### explain

The explain value tells the system if you want explanations about the recommendations.

Send a value of `1` for explain in order to get explanations.

**Note:** To not include explanations in the results, simply do not include the `explain` field when you call the function.

### 7.4.2 Outputs

The `query` method returns an object containing:

- `ids`
- `distances`
- `explanations`

**Note:** Explanations are optional. To include them in the response, see above.

#### ids

The ids are the item recommendations and are ordered by index position from most to least similar to the search input.

The item at index position 0 is the most similar item and the item in the last index position is the least similar.

**Note:** In the example, A is the top recommendation and has a distance of 1 from the search item. B is the second next best recommendation and has a distance of 3 from the search item.

#### distances

The distance values tell you how similar each result is to the search item.

**Note:** The index positions of the distances correspond to the index positions of the ids.

The smaller the distance between two vectors, the more similar the items are to one another.

The distance is an overall similarity value based on comparing the vector for one indexed item to the vector for the search item.

#### explanations

The explanations tell you more about how the system calculated the distances by providing distance values for each encoder.

**Note:** The index positions of the explanations correspond to the index positions of the ids.

In the example above, A is overall more similar to the search item than B is to the search item.

The explanations show why.

It is because A has a smaller distance for category than B by 8 and is greater distance for price than B but by only 2.

Plus, the encoder configurations gave category a weight of 2 and price a weight of 1 making category twice as important as price.

**Note:** Because A beats B on category by 4x more than B beats A on price and because category is greater weight, A has two reasons to be more similar to the search than B has.

## 7.5 User query

Read below to learn more about the inputs and outputs.

### 7.5.1 Inputs

The `user_query` method requires the following inputs:

- `k`
- `item_history`
- `data`
- `explain`

#### k

The `k` value is the number of items you want the system to return as recommendations.

#### item history

The `item history` is an array of item ids that the user has previous history with.

The system uses the item history to convert a user to an item vector.

**Note:** A common example is an array of ids for items the user previously purchased. In the example code, this user previously bought item 1 one time and item 3 twice.

The system uses the features of the items in the array to create an item vector that represents the user based on the features of those items.

The system knows the features of each item in the array because you tell the system the item features when you index the items.

---

**Note:** The system compare the user's item vector to the item vectors for the indexed items. In other words, if a customer bought three bananas, an apple, and a carrot, their user vector represents a combination of the features from three bananas, an apple, and a carrot.

---

The Recsplain system compares the user's item vector to the item vector for each indexed item to calculate distance.

#### data

The data is an object containing fields and values about the user your are searching for. Each user data field should correspond to a field in your indexed items.

---

**Note:** User data is most commonly used as hard filters. For instance, in the example in these docs, the system will only recommend US items to the user, not EU items.

---

#### explain

The explain value tells the system if you want explanations about the recommendations.

Send a value of `1` for explain in order to get explanations.

---

**Note:** To not include explanations in the results, simply do not include the explain field when you call the function.

---

### 7.5.2 Outputs

The `user_query` method returns an object containing:

- `ids`
- `distances`
- `explanations`

---

**Note:** Explanations are optional. To include them in the response, see above.

---

#### ids

The ids are the item recommendations and are ordered by index position from most to least similar to the user's item vector.

The item at index position 0 is the item the user most likely prefers and the item in the last index position is the item the user least likely prefers.

---

**Note:** In the example, A is the top recommendation and B is the next best recommendation.

---

### distances

The distance values tell you how likely the user is to prefer the item.

The index positions of the ids correspond to the index positions of the distances.

**Note:** In the example, A is the top recommendation and has a distance of 0.888898987902 from the search item. B is the next best recommendation and has a distance of 3.555675839384 from the search item.

The smaller the distance for an item, the more likely the user is to prefer the item.

The distance is an overall similarity value based on comparing the vector for one indexed item to the user's item vector.

**Note:** The results for the user search are the same as for the item search except the user search distances are floats instead of integers.

### explanations

The explanations tell you more about how the system calculated the distances by providing distance values for each encoder.

In the example above, the user is more likely to prefer A than B.

The explanations show why.

It is because A has a lower distance for category than B and a lower distance for price than B.

**Note:** Remember to take the encoder weights into account when reviewing the explanations. The encoder configurations in the example weighted category twice as important as price.

Because A beats B on category and on price, A has two reasons to be more similar to the search than B has.

## 7.6 Encoders

Recsplain comes with a variety of encoders.

**Note:** The code for the encoders is in encoders.py

Here is the list.

### 7.6.1 NumericEncoder

Use for numeric data. example:

```
{
  "field": "fat_precentage",
  "values": np.linspace(0, 100, num=101),
  "type": "numeric"",
  "weight": 1
}
```

### 7.6.2 OneHotEncoder

Use for categorical data. First category is saved for "unknown" entries. example:

```
{
  "field": "category",
  "values": ["dairy", "pasrty", "meat"],
  "type": "onehot",
  "weight": 1
}
```

### 7.6.3 StrictOneHotEncoder

Use for categorical data. No "unknown" category. example:

```
{
  "field": "category",
  "values": ["dairy", "pasrty", "meat"],
  "type": "strictonehot",
  "weight": 1
}
```

### 7.6.4 OrdinalEncoder

Use for ordinal data. `window` is the allowed similarity leakage between closed values. example:

```
{
  "field": "price",
  "values": ["low", "mid", "high"],
  "type": "ordinal",
  "weight": 1,
  "window": [0.1,1,0.1]

}
```

### 7.6.5 BinEncoder

Use for binning data. `values` is the boundaries of the bins. example:

```
{
  "field": "product_color",
  "values": ['blue', 'red', 'green'],
  "type": "bin",
  "weight": 1,
}
```

### 7.6.6 BinOrdinalEncoder

Use for binning ordinal data.

`values` is the boundaries of the bins.

`window` is the allowed similarity leakage between closed values.

example:

```
{
  "field": "price",
  "values": [10, 50, 100, 500, 1000],
  "type": "binordinal",
  "weight": 1,
  "window": [0.2,1,0.1]
}
```

### 7.6.7 HierarchyEncoder

Use for hierarchical data. example:

```
{
  "field": "sub_category",
  "values": {"meat":["chicken","beef"],"dairy": ['milk','yogurt'],"pastry":['bread',
↪'baguette']},
  "type": "hierarchy",
  "weight": 1,
}
```

### 7.6.8 NumpyEncoder

User defined encoder as numpy array.

### 7.6.9 JSONEncoder

User defined encoder as json.

### 7.6.10 QwakEncoder

Use with qwak data format.

## 7.7 What is Recsplain

The Recsplain system is an explainable recommendation system.

It consists of a tabular similarity search server that calculates item similarity and recommends items with explanations.

### 7.7.1 Recommendation system

The Recsplain recommendation engine uses machine learning to recommend items based on how similar an item is to another item or how similar an item is to a user's preferences.

After you configure the system and index your items, you can use the system to:

- Search by item to find similar items in your database based on item features
- Recommend items to users based on the user's history with the items

The applications are virtually endless.

**Note:** One common application of Recsplain is in online stores where the system recommends products to customers based on items the customer already bought from the store.

### 7.7.2 Create your own

Use Recsplain to create your very own recommendation engine using your configurations and your data.

It is easy to install and customize to suit your needs. You can configure the system to make recommendations based on your needs by using:

- Filters to categorically exclude and separate data for comparisons
- Encoders to dictate how the system checks whether items are similar

After configuring the system, easily index a list of items from your database to start recommending items to your users.

### 7.7.3 Make recommendations

Use the system as a webserver or by calling the methods from the Recsplain code itself to recommend items by item or by user.

Search by item to get similar items ordered by most to least similar, their distances from the search item based on the item vectors, and explanations about the recommendations.

Search by user to get items the user most likely prefers ordered from most to least likely, each item's distance from the user based on the user's vector, and explanations about the recommendations.

### 7.7.4 Use explanations

The Recsplain system explains its recommendations so that you can better understand the results.

The explanations tell you the degree of similarity for each feature so that you can better understand the order of the recommendations and the distance for each item.

The explanations are a more granular type of result than the overall distance value.

### 7.7.5 Get started

Read about *how it works* or just *get started*!

## 7.8 How it works

Using the Recsplain system is straightforward and does not require you to know any machine learning or advanced math.

Rather, you just need to install the Recsplain Python package into your project, configure the system, and index your items.

After setup, you can search by item or user to recommend items based on the similarity of your indexed items to the search item or user.

**Note:** You can use Recsplain as a webserver or with Python bindings to call the methods in your code.

When searching by item, the system creates a numerical vector representing the search item based on the search item features.

The system compares the vector for your search item to each item vector for the items you indexed to calculate the distance between the search item vector and each database item vector.

When searching by user, the system creates a numerical vector representing the user as an item vector based on the user's history with the items

The system compares the item vector for your user to each item vector for the items you indexed to calculate the distance between the user's item vector and each database item vector.

The system recommends items based on the distances. The smaller the distance between vectors, the more similar they are and the stronger the recommendation.

Read below to learn more about setup and how it works.

### 7.8.1 Easy Setup

It is just three easy steps to get started.

First, install the package.

Second, configure the system with your search filters, encoders, and metric.

Third, index the items from your database that you want to use in the system for checking similarity.

That is it. You can start making recommendations!

---

**Note:** Try it now by following the get started<get-started> guide!

---

Make recommendations in two different ways.

- Search by item for similar items based on item features
- Search by user for items the user is most likely to prefer based on their history with the items

---

**Note:** Learn more about Recsplain by exploring more about how the system works.

---

### 7.8.2 Custom configuration

The Recsplain system is customizable in several important ways.

#### Similarity check

First, customize how the system organizes and compares the items in your database. Simply send your configuration data to the system.

The configuration data consists of your filters, encoders, and metric.

Filters are hard filters. The system uses the hard filters to separate the indexed items into partitions. Use the partitions to control which items are checked for similarity each time you run an item or user query.

---

**Note:** When searching, the system checks whether the search item or user is similar to the items within a particular partition **only if** the search item or user fits the filter criteria for that partition.

---

Encoders are soft filters. Use them to control how the system checks for similarity.

---

**Note:** The system uses encoders to determine **how to check** for similarity within each partition.

---

#### Data Index

Second, customize the data by indexing data from your database. This way you can make recommendations based on your actual data.

You can index all your database items or just the data that you want to include as possible recommendations.

### 7.8.3 Query multiple ways

Recommend items by checking for similarity based on an item or a user.

**Note:** The system has separate methods for item and user searches.

When you search, you send the system data about the search item or user.

The search item data consists of an id and values for item features that correspond to the filters and encoders.

The search user data consists of a user id and an item history, like a purchase history, where each item in the history is and id for an indexed item.

**Note:** If you are using Recsplain as a web server, you send the search data in the body of a POST request. If you are using it with Python bindings, call the search method and pass your item data as an argument.

### 7.8.4 Understand results

Each time you search by item or user, the system returns items it deems similar to the search item or user and explanations for each item in the results.

The system returns the items in an array ordered by most to least similar. The first item in the array is the item that is most similar and the last item in the array is the least similar.

The degree of similarity is measured using the distance between the indexed item vectors and the vector for the search item or user.

When searching by item, similarity consists of comparing the search item vector to the vector for each item in the database.

When searching by user, similarity consists of creating an item vector for the user based on the user's history with the item and comparing this user vector to the item vector for each indexed item.

For each item in the array, the system also returns an array of distances telling you how similar each item is to the search item or user.

Optionally, the system also returns an array of explanations consisting of more granular result data from which the system derived the final recommendations and overall distances.