dd_api Documentation

Release 1.0

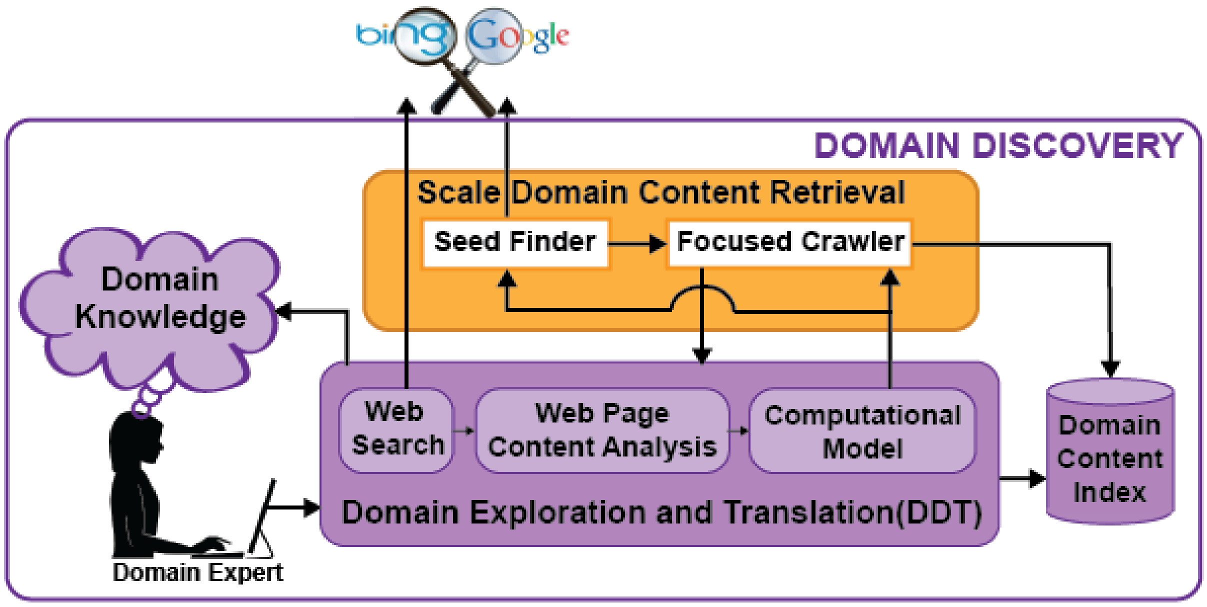
Yamuna Krishnamurthy

Apr 25, 2017

Contents

Domain Discovery is the process of acquiring, understanding and exploring data for a specific domain. Some example domains include human trafficking, illegal sale of weapons and micro-cap fraud. While acquiring knowledge about a domain humans usually start with a conception of that domain. This conception is based on prior knowledge of parts of the domain. The process of gaining a more complete knowledge of the domain involves using this prior knowledge to obtain content that provides additional information about that domain that was previously unknown. This new knowledge of the domain now becomes prior knowledge leading to an iterative process of domain discovery as illustrated in Figure 2. The goals of this iterative domain discovery process are:

- complete the human's knowledge of the domain
- acquire sufficient content that captures the human coginition of the domain to translate into a computational model



The Domain Discovery API formalizes the human domain discovery process by defining a set of operations that capture the essential tasks that lead to domain discovery on the Web as we have discovered in interacting with the Subject Matter Experts (SME)s. The API facilitates:

- Creation of different user interfaces to satisfy different DD needs
- Configure and understand different DD workflows
- Scripting DD

CHAPTER 1

Contents

Installation

Building and deploying the Domain Discovery can be done using its Makefile to create a local development environment. The conda build environment is currently only supported on 64-bit OS X and Linux.

First install conda, either through the Anaconda or miniconda installers provided by Continuum. You will also need Git, a Java Development Kit and Maven. These are system tools that are generally not provided by conda.

Clone the DD API repository and enter it:

```
>>> git clone https://github.com/ViDA-NYU/domain_discovery_API
>>> cd domain_discovery_API
```

Use the *make* command to build DD API and download/install its dependencies.

```
>>> make
```

Now you can use the API

DD API Operations

`models.domain_discovery_model.`**`random`**`()` → x in the interval [0, 1).

Acquire Content

class `models.domain_discovery_model.`**`DomainModel`**

`queryWeb`(*terms*, *max_url_count=100*, *session=None*)
Issue query on the web: results are stored in elastic search, nothing returned here.

Parameters: terms (string): Search query string max_url_count (int): Number of pages to query. Maximum allowed = 100 session (json): should have domainId

Returns: None

uploadUrls(*urls_str*, *session*)
Download pages corresponding to already known set of domain URLs

Parameters: urls_str (string): Space separated list of URLs session (json): should have domainId

Returns: number of pages downloaded (int)

getForwardLinks(*urls*, *session*)
The content can be extended by crawling the given pages one level forward. The assumption here is that a relevant page will contain links to other relevant pages.

Parameters: urls (list): list of urls to crawl forward session (json): should have domainId

Return: None (Results are downloaded into elasticsearch)

getBackwardLinks(*urls*, *session*)
The content can be extended by crawling the given pages one level back to the pages that link to them. The assumption here is that a page containing the link to the given relevant page will contain links to other relevant pages.

Parameters: urls (list): list of urls to crawl backward session (json): should have domainId

Return: None (Results are downloaded into elasticsearch)

Annotate Content

class models.domain_discovery_model.**DomainModel**

setPagesTag(*pages*, *tag*, *applyTagFlag*, *session*)
Tag the pages with the given tag which can be a custom tag or 'Relevant'/'Irrelevant' which indicate relevance or irrelevance to the domain of interest. Tags help in clustering and categorizing the pages. They also help build computational models of the domain.

Parameters: pages (urls): list of urls to apply tag tag (string): custom tag, 'Relevant', 'Irrelevant' applyTagFlag (bool): True - Add tag, False - Remove tag session (json): Should contain domainId

Returns: Returns string "Completed Process"

setTermsTag(*terms*, *tag*, *applyTagFlag*, *session*)
Tag the terms as 'Positive'/'Negative' which indicate relevance or irrelevance to the domain of interest. Tags help in reranking terms to show the ones relevan to the domain.

Parameters: terms (string): list of terms to apply tag tag (string): 'Positive' or 'Negative' applyTagFlag (bool): True - Add tag, False - Remove tag session (json): Should contain domainId

Returns: None

Summarize Content

class models.domain_discovery_model.**DomainModel**

extractTerms(*opt_maxNumberOfTerms=40*, *session=None*)
Extract most relevant unigrams, bigrams and trigrams that summarize the pages. These could provide unknown information about the domain. This in turn could suggest further queries for searching content.

Parameters: opt_maxNumberOfTerms (int): Number of terms to return

session (json): should have domainId

Returns: array: [[term, frequencyInRelevantPages, frequencyInIrrelevantPages, tags], ...]

`make_topic_model`(*session*, *tokenizer*, *vectorizer*, *model*, *ntopics*)

Build topic model from the corpus of the supplied DDT domain.

The topic model is represented as a topik.TopikProject object, and is persisted in disk, recording the model parameters and the location of the data. The output of the topic model itself is stored in Elasticsearch.

Parameters:

domain (str): DDT domain name as stored in Elasticsearch, so lowercase and with underscores in place of spaces.

tokenizer (str): A tokenizer from `topik.tokenizer.registered_tokenizers`

vectorizer (str): A vectorization method from `topik.vectorizers.registered_vectorizers`

model (str): A topic model from `topik.vectorizers.registered_models`

ntopics (int): The number of topics to be used when modeling the corpus.

Returns:

model: topik model, encoding things like term frequencies, etc.

Organize Content

class `models.domain_discovery_model.`**`DomainModel`**

`getPagesProjection`(*session*)

Organize content by some criteria such as relevance, similarity or category which allows to easily analyze groups of pages. The 'x','y' co-ordinates returned project the page in 2D maintaining clustering based on the projection chosen. The projection criteria is specified in the session object

Parameters: session: Should Contain 'domainId' Should contain 'activeProjectionAlg' which takes values 'tsne', 'pca' or 'kmeans' currently

Returns dictionary in the format:{ 'last_downloaded_url_epoch': 1432310403 (in seconds) 'pages': [[url1, x, y, tags,
] }

Filter Content

class `models.domain_discovery_model.`**`DomainModel`**

`getPages`(*session*)

Find pages that satisfy the specified criteria. One or more of the following criteria are specified in the session object as 'pageRetrievalCriteria':

'Most Recent', 'More like', 'Queries', 'Tags', 'Model Tags', 'Maybe relevant', 'Maybe irrelevant', 'Unsure'

and filter by keywords specified in the session object as 'filter'

Parameters: session (json): Should contain 'domainId','pageRetrievalCriteria' or 'filter'

Returns: json: {url1: {snippet, image_url, title, tags, retrieved}} (tags are a list, potentially empty)

Generate Model

class `models.domain_discovery_model.`**`DomainModel`**

`createModel`(*session*, *zip=True*)

Create an ACHE model to be applied to SeedFinder and focused crawler. It saves the classifiers, features, the training data in the <project>/data/<domain> directory. If zip=True all generated files and folders are zipped into a file.

Parameters: session (json): should have domainId

Returns: None

CHAPTER 2

Links

- GitHub repository

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index