diffeo Documentation

Release 0.3.4

Diffeo, Inc.

August 19, 2015

Contents

Contents:

CHAPTER 1

dossier.fc — feature collections

Collections of named features.

This module provides *dossier.fc.FeatureCollection* and a number of supporting classes. A feature collection is a dictionary mapping a feature name to a feature representation. The representation is typically a *dossier.fc.StringCounter*, an implementation of collections.Counter, which is fundamentally a mapping from a string value to an integer.

This representation allows multiple values to be stored for multiple bits of information about some entity of interest. The weights in the underlying counters can be used to indicate how common a particular value is, or how many times it appears in the source documents.

```
president = FeatureCollection()
president['NAME']['Barack Obama'] += 1
president['entity_type']['PER'] += 1
president['PER_ADDRESS']['White House'] += 1
president['PER_ADDRESS']['1600 Pennsylvania Ave.'] += 1
```

Feature collections can have representations other than the basic string-counter representation; these representations may not preserve the source strings, but will still be suitable for machine-learning applications.

Feature collections can be serialized to **RFC 7049** CBOR format, similar to a binary JSON representation. They can also be stored sequentially in flat files using *dossier.fc.FeatureCollectionChunk* as an accessor.

class dossier.fc.**FeatureCollection**(*data=None*, *read_only=False*)

Bases: _abcoll.MutableMapping

A collection of features.

This is a dictionary from feature name to a collections.Counter or similar object. In typical use callers will not try to instantiate individual dictionary elements, but will fall back on the collection's default-value behavior:

```
fc = FeatureCollection()
fc['NAME']['John Smith'] += 1
```

The default default feature type is *StringCounter*.

Feature collection construction and serialization:

__init__(*data=None*, *read_only=False*)

Creates a new empty feature collection.

If data is a dictionary-like object with a structure similar to that of a feature collection (i.e., a dict of multisets), then it is used to initialize the feature collection.

classmethod loads(*data*)
Create a feature collection from a CBOR byte string.

dumps()
Create a CBOR byte string from a feature collection.

classmethod from_dict(*data*, *read_only=False*)
Recreate a feature collection from a dictionary.

The dictionary is of the format dumped by *to_dict()*. Additional information, such as whether the feature collection should be read-only, is not included in this dictionary, and is instead passed as parameters to this function.

to_dict()
Dump a feature collection's features to a dictionary.

This does not include additional data, such as whether or not the collection is read-only. The returned dictionary is suitable for serialization into JSON, CBOR, or similar data formats.

static register_serializer(*feature_type*, *obj*)
This is a **class** method that lets you define your own feature type serializers. `tag` should be the name of the feature type that you want to define serialization for. Currently, the valid values are `StringCounter`, `Unicode`, `SparseVector` or `DenseVector`.

Note that this function is not thread safe.

`obj` must be an object with three attributes defined.

`obj.loads` is a function that takes a CBOR created Python data structure and returns a new feature counter.

`obj.dumps` is a function that takes a feature counter and returns a Python data structure that can be serialized by CBOR.

`obj.constructor` is a function with no parameters that returns the Python `type` that can be used to construct new features. It should be possible to call `obj.constructor()` to get a new and empty feature counter.

Feature collection values and attributes:

read_only
Flag if this feature collection is read-only.

When a feature collection is read-only, no part of it can be modified. Individual feature counters cannot be added, deleted, or changed. This attribute is preserved across serialization and deserialization.

generation
Get the generation number for this feature collection.

This is the highest generation number across all counters in the collection, if the counters support generation numbers. This collection has not changed if the generation number has not changed.

DISPLAY_PREFIX = '#'
Prefix on names of features that are human-readable.

Processing may convert a feature `name` to a similar feature `#name` that is human-readable, while converting the original feature to a form that is machine-readable only; for instance, replacing strings with integers for faster comparison.

EPHEMERAL_PREFIX = '_'
Prefix on names of features that are not persisted.

to_dict() and *dumps()* will not write out features that begin with this character.

Feature collection computation:

__add__(*other*)
Add features from two FeatureCollections.

```
>>> fc1 = FeatureCollection({'foo': Counter('abbb')})
>>> fc2 = FeatureCollection({'foo': Counter('bcc')})
>>> fc1 + fc2
FeatureCollection({'foo': Counter({'b': 4, 'c': 2, 'a': 1})})
```

Note that if a feature in either of the collections is not an instance of `collections.Counter`, then it is ignored.

__sub__(*other*)
Subtract features from two FeatureCollections.

```
>>> fc1 = FeatureCollection({'foo': Counter('abbb')})
>>> fc2 = FeatureCollection({'foo': Counter('bcc')})
>>> fc1 - fc2
FeatureCollection({'foo': Counter({'b': 2, 'a': 1})})
```

Note that if a feature in either of the collections is not an instance of `collections.Counter`, then it is ignored.

__mul__(*coef*)

__imul__(*coef*)
In-place multiplication by a scalar.

total()
Returns sum of all counts in all features that are multisets.

merge_with(*other*, *multiset_op*, *other_op=None*)
Merge this feature collection with another.

Merges two feature collections using the given `multiset_op` on each corresponding multiset and returns a new *FeatureCollection*. The contents of the two original feature collections are not modified.

For each feature name in both feature sets, if either feature collection being merged has a `collections.Counter` instance as its value, then the two values are merged by calling *multiset_op* with both values as parameters. If either feature collection has something other than a `collections.Counter`, and *other_op* is not `None`, then *other_op* is called with both values to merge them. If *other_op* is `None` and a feature is not present in either feature collection with a counter value, then the feature will not be present in the result.

Parameters

- **other** (*FeatureCollection*) – The feature collection to merge into `self`.
- **multiset_op** (*fun(Counter, Counter) -> Counter*) – Function to merge two counters
- **other_op** (*fun(object, object) -> object*) – Function to merge two non-counters

Return type *FeatureCollection*

class dossier.fc.**StringCounter**(**args*, ***kwargs*)
Bases: `collections.Counter`

Simple counter based on exact string matching.

This is a subclass of `collections.Counter` that includes a generation counter so that it can be used in a cache.

StringCounter is the default feature type in a feature collection, so you typically don't have to instantiate a *StringCounter* explicitly:

```
fc = FeatureCollection()
fc['NAME']['John Smith'] += 1
```

But instantiating directly works too:

```
sc = StringCounter()
sc['John Smith'] += 1

fc = FeatureCollection({'NAME': sc})
fc['NAME']['John Smith'] += 1
assert fc['NAME']['John Smith'] == 2
```

Note that instances of this class support all the methods defined for a `collections.Counter`, but only the ones unique to *StringCounter* are listed here.

__init__(**args*, ***kwargs*)

Initialize a *StringCounter* with existing counts:

```
>>> sc = StringCounter(a=4, b=2, c=0)
>>> sc['b']
2
```

See the documentation for `collections.Counter` for more examples.

truncate_most_common(**args*, ***kwargs*)

Sorts the counter and keeps only the most common items up to `truncation_length` in place.

read_only

Flag indicating whether this collection is read-only.

This flag always begins as `False`, it cannot be set via the constructor for compatibility with `collections.Counter`. If this flag is set, then any operations that mutate it will raise `ReadOnlyException`.

generation

Generation number for this counter instance.

This number is incremented by every operation that mutates the counter object. If two collections are the same object and have the same generation number, then they are identical.

Having this property allows a pair of *id(sc)* and the generation to be an immutable hashable key for things like memoization operations, accounting for the possibility of the counter changing over time.

```
>>> sc = StringCounter({'a': 1})
>>> cache = {(id(sc), sc.generation): 1}
>>> (id(sc), sc.generation) in cache
True
>>> sc['a']
1
>>> (id(sc), sc.generation) in cache
True
>>> sc['a'] += 1
>>> sc['a']
2
>>> (id(sc), sc.generation) in cache
False
```

class dossier.fc.**SparseVector**
Bases: object

An abstract class for sparse vectors.

Currently, there is no default implementation of a sparse vector.

Other implementations of sparse vectors *must* inherit from this class. Otherwise they cannot be used inside a *dossier.fc.FeatureCollection*.

class dossier.fc.**DenseVector**
Bases: object

An abstract class for dense vectors.

Currently, there is no default implementation of a dense vector.

Other implementations of dense vectors *must* inherit from this class. Otherwise they cannot be used inside a *dossier.fc.FeatureCollection*.

dossier.fc.**FeatureCollectionChunk**
alias of <Mock id='140505177601168'>

class dossier.fc.**ReadOnlyException**
Bases: dossier.fc.exceptions.BaseException

Code attempted to modify a read-only feature collection.

This occurs when adding, deleting, or making other in-place modifications to a *FeatureCollection* that has its *read_only* flag set. It also occurs when attempting to make changes to a *StringCounter* contained in such a collection.

class dossier.fc.**SerializationError**
Bases: dossier.fc.exceptions.BaseException

A problem occurred serializing or deserializing.

This can occur if a *FeatureCollection* has an unrecognized feature type, or if a CBOR input does not have the correct format.

CHAPTER 2

dossier.store — store feature collections

A simple storage interface for feature collections.

mod:*dossier.store* provides a convenient interface to a mod:*kvlayer* table for storing *dossier.fc.FeatureCollection*. The interface consists of methods to query, search, add and remove feature collections from the store. It also provides functions for defining and searching indexes.

Using a storage backend in your code requires a working `kvlayer` configuration, which is usually written in a YAML file like so:

```
kvlayer:
  app_name: store
  namespace: dossier
  storage_type: redis
  storage_addresses: ["redis.example.com:6379"]
```

And here's a full working example that uses local memory to store feature collections:

```
from dossier.fc import FeatureCollection
from dossier.store import Store
import kvlayer
import yakonfig

yaml = """
kvlayer:
  app_name: store
  namespace: dossier
  storage_type: local
"""
with yakonfig.defaulted_config([kvlayer], yaml=yaml):
    store = Store(kvlayer.client())

    fc = FeatureCollection({u'NAME': {'Foo': 1, 'Bar': 2}})
    store.put([('1', fc)])
    print store.get('1')
```

See the documentation for `yakonfig` for more details on the configuration setup.

Another example showing how to store, retrieve and delete a feature collection:

```
fc = dossier.fc.FeatureCollection()
fc[u'NAME'][u'foo'] += 1
fc[u'NAME'][u'bar'] = 42

kvl = kvlayer.client()
```

```
store = dossier.store.Store(kvl)

store.put('{yourid}', fc)
assert store.get('{yourid}')[u'NAME'][u'bar'] == 42
store.delete('{yourid}')
assert store.get('{yourid}') is None
```

Here is another example that demonstrates use of indexing to enable a poor man's case insensitive search:

```
fc = dossier.fc.FeatureCollection()
fc[u'NAME'][u'foo'] += 1
fc[u'NAME'][u'bar'] = 42

kvl = kvlayer.client()
store = dossier.store.Store(kvl)

# Index transforms must be defined on every instance of `Store`.
# (The index data is persisted; the transforms themselves are
# ephemeral.)
store.define_index(u'name_casei',
                   create=feature_index(u'NAME'),
                   transform=lambda s: s.lower().encode('utf-8'))

store.put('{yourid}', fc)  # `put` automatically updates indexes.
assert list(store.index_scan(u'name_casei', 'FoO'))[0] == '{yourid}'
```

class dossier.store.**Store**(*kvlclient*, *impl=None*, *feature_indexes=None*)

A feature collection database.

A feature collection database stores feature collections for content objects like profiles from external knowledge bases.

Every feature collection is keyed by its `content_id`, which is a byte string. The value of a `content_id` is specific to the type of content represented by the feature collection. In other words, its representation is unspecified.

__init__(*kvlclient*, *impl=None*, *feature_indexes=None*)

Connects to a feature collection store.

This also initializes the underlying kvlayer namespace.

Parameters **kvl** (kvlayer.AbstractStorage) – kvlayer storage client

Return type *Store*

get(*content_id*)

Retrieve a feature collection from the store. This is the same as get_many([content_id])

If the feature collection does not exist `None` is returned.

Return type *dossier.fc.FeatureCollection*

get_many(*content_id_list*)

Yield (content_id, data) tuples for ids in list.

As with *get()*, if a content_id in the list is missing, then it is yielded with a data value of *None*.

Return type yields tuple(str, *dossier.fc.FeatureCollection*)

put(*items*, *indexes=True*)

Add feature collections to the store.

Given an iterable of tuples of the form `(content_id, feature collection)`, add each to the store and overwrite any that already exist.

This method optionally accepts a keyword argument *indexes*, which by default is set to `True`. When it is `True`, it will *create* new indexes for each content object for all indexes defined on this store.

Note that this will not update existing indexes. (There is currently no way to do this without running some sort of garbage collection process.)

Parameters **`items`** (*iterable*) – iterable of `(content_id, FeatureCollection)`.

`delete`(*content_id*)
Delete a feature collection from the store.

Deletes the content item from the store with identifier `content_id`.

Parameters **`content_id`** (*str*) – identifier for the content object represented by a feature collection

`delete_all`()
Deletes all storage.

This includes every content object and all index data.

`scan`(**key_ranges*)
Retrieve feature collections in a range of ids.

Returns a generator of content objects corresponding to the content identifier ranges given. *key_ranges* can be a possibly empty list of 2-tuples, where the first element of the tuple is the beginning of a range and the second element is the end of a range. To specify the beginning or end of the table, use an empty tuple *()*.

If the list is empty, then this yields all content objects in the storage.

Parameters **`key_ranges`** – as described in `kvlayer._abstract_storage.AbstractStorage()`

Return type generator of (`content_id`, *dossier.fc.FeatureCollection*).

`scan_ids`(**key_ranges*)
Retrieve content ids in a range of ids.

Returns a generator of `content_id` corresponding to the content identifier ranges given. *key_ranges* can be a possibly empty list of 2-tuples, where the first element of the tuple is the beginning of a range and the second element is the end of a range. To specify the beginning or end of the table, use an empty tuple *()*.

If the list is empty, then this yields all content ids in the storage.

Parameters **`key_ranges`** – as described in `kvlayer._abstract_storage.AbstractStorage()`

Return type generator of `content_id`

`scan_prefix`(*prefix*)
Returns a generator of content objects matching a prefix.

The `prefix` here is a prefix for `content_id`.

Return type generator of (`content_id`, *dossier.fc.FeatureCollection*).

`scan_prefix_ids`(*prefix*)
Returns a generator of content ids matching a prefix.

The `prefix` here is a prefix for `content_id`.

Return type generator of `content_id`

Methods for indexing:

index_scan (*idx_name*, *val*)
Returns ids that match an indexed value.

Returns a generator of content identifiers that have an entry in the index `idx_name` with value `val` (after index transforms are applied).

If the index named by `idx_name` is not registered, then a `KeyError` is raised.

Parameters

- **idx_name** (*unicode*) – name of index
- **val** (unspecified (depends on the index, usually `unicode`)) – the value to use to search the index

Return type generator of `content_id`

Raises `KeyError`

index_scan_prefix (*idx_name*, *val_prefix*)
Returns ids that match a prefix of an indexed value.

Returns a generator of content identifiers that have an entry in the index `idx_name` with prefix `val_prefix` (after index transforms are applied).

If the index named by `idx_name` is not registered, then a `KeyError` is raised.

Parameters

- **idx_name** (*unicode*) – name of index
- **val_prefix** – the value to use to search the index

Return type generator of `content_id`

Raises `KeyError`

index_scan_prefix_and_return_key (*idx_name*, *val_prefix*)
Returns ids that match a prefix of an indexed value, and the specific key that matched the search prefix.

Returns a generator of (index key, content identifier) that have an entry in the index `idx_name` with prefix `val_prefix` (after index transforms are applied).

If the index named by `idx_name` is not registered, then a `KeyError` is raised.

Parameters

- **idx_name** (*unicode*) – name of index
- **val_prefix** – the value to use to search the index

Return type generator of (`index key`, `content_id`)

Raises `KeyError`

define_index (*idx_name*, *create*, *transform*)
Add an index to this store instance.

Adds an index transform to the current FC store. Once an index with name `idx_name` is added, it will be available in all `index_*` methods. Additionally, the index will be automatically updated on calls to `put()`.

If an index with name `idx_name` already exists, then it is overwritten.

Note that indexes do *not* persist. They must be re-defined for each instance of *`Store`*.

For example, to add an index on the `boNAME` feature, you can use the `feature_index` helper function:

```
        store.define_index('boNAME',
                           feature_index('boNAME'),
                           lambda s: s.encode('utf-8'))
```

Another example for creating an index on names:

```
        store.define_index('NAME',
                           feature_index('canonical_name', 'NAME'),
                           lambda s: s.lower().encode('utf-8'))
```

Parameters

- **idx_name** (*unicode*) – The name of the index. Must be UTF-8 encodable.
- **create** – A function that accepts the `transform` function and a pair of `(content_id, fc)` and produces a generator of index values from the pair given using `transform`.
- **transform** – A function that accepts an arbitrary value and applies a transform to it. This transforms the *stored* value to the *index* value. This *must* produce a value with type *str* (or *bytes*).

dossier.store.**feature_index**(**feature_names*)

Returns a index creation function.

Returns a valid index `create` function for the feature names given. This can be used with the *Store.define_index()* method to create indexes on any combination of features in a feature collection.

Return type `(val -> index val) -> (content_id, FeatureCollection) -> generator of [index val]`

CHAPTER 3

dossier.label — store ground truth data as labels

A simple storage interface for labels (ground truth data).

dossier.label provides a convenient interface to a `kvlayer` table for storing ground truth data, otherwise known as "labels." Each label, at the highest level, maps two things (addressed by content identifiers) to a coreferent value. This coreferent value is an indication by a human that these two things are "the same", "not the same" or "I don't know if they are the same." *Sameness* in this case is determined by the human doing the annotation.

Each label also contains an `annotator_id`, which identifies the human that created the label. A timestamp (in milliseconds since the Unix epoch) is also included on every label.

3.1 Example

Using a storage backend in your code requires a working `kvlayer` configuration, which is usually written in a YAML file like so:

```
kvlayer:
  app_name: store
  namespace: dossier
  storage_type: redis
  storage_addresses: ["redis.example.com:6379"]
```

And here's a full working example that uses local memory to store labels:

```
from dossier.label import Label, LabelStore, CorefValue
import kvlayer
import yakonfig

yaml = """
kvlayer:
  app_name: store
  namespace: dossier
  storage_type: local
"""
with yakonfig.defaulted_config([kvlayer], yaml=yaml):
    label_store = LabelStore(kvlayer.client())

    lab = Label('a', 'b', 'annotator', CorefValue.Positive)
    label_store.put(lab)

    assert lab == label_store.get('a', 'b', 'annotator')
```

See the documentation for `yakonfig` for more details on the configuration setup.

class dossier.label.**LabelStore**(*kvlclient*)
A label database.

__init__(*kvlclient*)
Create a new label store.

Parameters **kvlclient** (kvlayer._abstract_storage.AbstractStorage) – kvlayer client

Return type *LabelStore*

put(**labels*)
Add a new label to the store.

Parameters **label** (*Label*) – label

get(*cid1*, *cid2*, *annotator_id*, *subid1=''*, *subid2=''*)
Retrieve a label from the store.

When `subid1` and `subid2` are empty, then a label without subtopic identifiers will be returned.

If there are multiple labels stored with the same parts, the single most recent one will be returned. If there are no labels with these parts, `exceptions.KeyError` will be raised.

Parameters

- **cid1** (*str*) – content id
- **cid2** (*str*) – content id
- **annotator_id** (*str*) – annotator id
- **subid1** (*str*) – subtopic id
- **subid2** (*str*) – subtopic id

Return type *Label*

Raises `KeyError` if no label could be found.

directly_connected(*ident*)
Return a generator of labels connected to `ident`.

`ident` may be a `content_id` or a `(content_id, subtopic_id)`.

If no labels are defined for `ident`, then the generator will yield no labels.

Note that this only returns *directly* connected labels. It will not follow transitive relationships.

Parameters **ident** (`str` or `(str, str)`) – content id or (content id and subtopic id)

Return type generator of *Label*

connected_component(*ident*)
Return a connected component generator for `ident`.

`ident` may be a `content_id` or a `(content_id, subtopic_id)`.

Given an `ident`, return the corresponding connected component by following all positive transitivity relationships.

For example, if `(a, b, 1)` is a label and `(b, c, 1)` is a label, then `connected_component('a')` will return both labels even though `a` and `c` are not directly connected.

(Note that even though this returns a generator, it will still consume memory proportional to the number of labels in the connected component.)

Parameters **ident** (`str` or `(str, str)`) – content id or (content id and subtopic id)

Return type generator of *Label*

expand(*ident*)
Return expanded set of labels from a connected component.

The connected component is derived from `ident`. `ident` may be a `content_id` or a `(content_id, subtopic_id)`. If `ident` identifies a subtopic, then expansion is done on a subtopic connected component (and expanded labels retain subtopic information).

The labels returned by *LabelStore.connected_component()* contains only the *Label* stored in the *LabelStore*, and does not include the labels you can infer from the connected component. This method returns both the data-backed labels and the inferred labels.

Subtopic assignments of the expanded labels will be empty. The `annotator_id` will be an arbitrary `annotator_id` within the connected component.

Parameters

- **content_id** (*str*) – content id
- **value** (*CorefValue*) – coreferent value

Return type `list` of *Label*

everything(*include_deleted=False*, *content_id=None*, *subtopic_id=None*)
Returns a generator of all labels in the store.

If *include_deleted* is `True`, labels that have been overwritten with more recent labels are also included. If *content_id* is not `None`, only labels for that content ID are retrieved; and then if *subtopic_id* is not `None`, only that subtopic is retrieved, else all subtopics are retrieved. The returned labels will always be in sorted order, content IDs first, and with those with the same content, subtopic, and annotator IDs sorted newest first.

Return type generator of *Label*

delete_all()
Deletes all labels in the store.

class dossier.label.**Label**(*content_id1*, *content_id2*, *annotator_id*, *value*, *subtopic_id1=None*, *subtopic_id2=None*, *epoch_ticks=None*, *rating=None*, *meta=None*)
An immutable unit of ground truth data.

This is a statement that the item at *content_id1*, *subtopic_id1* refers to (or doesn't) the same thing as the item at *content_id2*, *subtopic_id2*. This assertion was recorded by *annotator_id*, a string identifying a user, at *epoch_ticks*, with *CorefValue* *value*.

On creation, the tuple is normalized such that the pair of *content_id1* and *subtopic_id1* are less than the pair of *content_id2* and *subtopic_id2*.

Labels are comparable, sortable, and hashable. The sort order compares the two content IDs, the two subtopic IDs, *annotator_id*, *epoch_ticks* (most recent is smallest), and then other fields.

content_id1
The first content ID.

content_id2
The second content ID.

annotator_id
An identifier of the user making this assertion.

value
A *CorefValue* stating whether this is a positive or negative coreference assertion.

subtopic_id1
An identifier defining a section or region of *content_id1*.

subtopic_id2
An identifier defining a section or region of *content_id2*.

epoch_ticks
The time at which *annotator_id* made this assertion, in seconds since the Unix epoch.

rating
An additional score showing the relative importance of this label, mirroring `streamcorpus.Rating`.

meta
Any additional meta data about this label, structured as a dictionary.

__init__(*content_id1*, *content_id2*, *annotator_id*, *value*, *subtopic_id1=None*, *subtopic_id2=None*, *epoch_ticks=None*, *rating=None*, *meta=None*)
Create a new label.

Parameters are assigned to their corresponding fields, with some normalization. If *value* is an `int`, the corresponding *CorefValue* is used instead. If *epoch_ticks* is `None` then the current time is used. If *rating* is `None` then it will be 1 if *value* is `CorefValue.Positive` and 0 otherwise.

__contains__(*v*)
Tests membership of identifiers.

If *v* is a tuple of `(content_id, subtopic_id)`, then the pair is checked for membership. Otherwise, *v* must be a `str` and is checked for equality to one of the content ids in this label.

As a special case, if *v* is `(content_id, None)`, then *v* is treated as if it were `content_id`.

```
>>> l = Label('c1', 'c2', 'a', 1)
>>> 'c1' in l
True
>>> 'a' in l
False
>>> ('c1', None) in l
True
>>> ('c1', 's1') in l
False
>>> ll = Label('c1', 'c2', 'a', 1, 's1', 's2')
>>> 'c1' in l
True
>>> ('c1', None) in l
True
>>> ('c1', 's1') in l
True
```

other(*content_id*)
Returns the other content id.

If `content_id == self.content_id1`, then return `self.content_id2` (and vice versa). Raises `exceptions.KeyError` if *content_id* is neither one.

```
>>> l = Label('c1, 'c2', 'a', 1)
>>> l.other('c1')
'c2'
>>> l.other('c2')
'c1'
>>> l.other('a')
Traceback (most recent call last):
```

```
        ...
    KeyError: 'a'
```

subtopic_for(*content_id*)
Get the subtopic id that corresponds with a content id.

```
    >>> l = Label('c1', 'c2', 'a', 1, 's1', 's2')
    >>> l.subtopic_for('c1')
    's1'
    >>> l.subtopic_for('c2')
    's2'
    >>> l.subtopic_for('a')
    Traceback (most recent call last):
        ...
    KeyError: 'a'
```

Parameters **content_id** (*str*) – content ID to look up

Returns subtopic ID for *content_id*

Raises **exceptions.KeyError** if *content_id* is neither content ID in this label

same_subject_as(*other*)
Determine if two labels are about the same thing.

This predicate returns `True` if *self* and *other* have the same content IDs, subtopic IDs, and annotator ID. The other fields may have any value.

```
    >>> t = time.time()
    >>> l1 = Label('c1', 'c2', 'a', CorefValue.Positive,
    ...            epoch_ticks=t)
    >>> l2 = Label('c1', 'c2', 'a', CorefValue.Negative,
    ...            epoch_ticks=t)
    >>> l1.same_subject_as(l2)
    True
    >>> l1 == l2
    False
```

static **most_recent**(*labels*)
Filter an iterator to return the most recent for each subject.

labels is any iterator over *Label* objects. It should be sorted with the most recent first, which is the natural sort order that `sorted()` and the *LabelStore* adapter will return. The result of this is a generator of the same labels but with any that are not the most recent for the same subject (according to *same_subject_as()*) filtered out.

class dossier.label.**CorefValue**
A human-assigned value for a coreference judgement.

The judgment is always made with respect to a pair of content items.

Variables

- **Negative** – The two items are not coreferent.
- **Unknown** – It is unknown whether the two items are coreferent.
- **Positive** – The two items are coreferent.

3.1.1 `dossier.label` command-line tool

`dossier.label` command-line tool.

`dossier.label` is a command line application for viewing the raw label data inside the database. Generally, this is a debugging tool for developers.

Run `dossier.label --help` for the available commands.

CHAPTER 4

dossier.models — Active learning

autodoc: failed to import module u'dossier.models'; the following exception was raised: Traceback (most recent call last): File "/home/docs/checkouts/readthedocs.org/user_builds/dossier-stack/envs/latest/local/lib/python2.7/site-packages/sphinx/ext/autodoc.py", line 385, in import_object __import__(self.modname) File "/home/docs/checkouts/readthedocs.org/user_builds/dossier-stack/envs/latest/lib/python2.7/site-packages/dossier.models-0.6.17.dev29-py2.7.egg/dossier/models/__init__.py", line 14, in <module> from dossier.models.pairwise import PairwiseFeatureLearner, similar, dissimilar File "/home/docs/checkouts/readthedocs.org/user_builds/dossier-stack/envs/latest/lib/python2.7/site-packages/dossier.models-0.6.17.dev29-py2.7.egg/dossier/models/pairwise.py", line 22, in <module> from sklearn.metrics.pairwise import pairwise_distances File "/home/docs/checkouts/readthedocs.org/user_builds/dossier-stack/envs/latest/local/lib/python2.7/site-packages/scikit_learn-0.16.1-py2.7-linux-x86_64.egg/sklearn/metrics/__init__.py", line 7, in <module> from .ranking import auc File "/home/docs/checkouts/readthedocs.org/user_builds/dossier-stack/envs/latest/local/lib/python2.7/site-packages/scikit_learn-0.16.1-py2.7-linux-x86_64.egg/sklearn/metrics/ranking.py", line 24, in <module> from scipy.sparse import csr_matrix RuntimeError: sys.path must be a list of directory names

CHAPTER 5

dossier.web — DossierStack web services

5.1 dossier.web provides REST web services for Dossier Stack

class dossier.web.**WebBuilder**(*add_default_routes=True*)

A builder for constructing DossierStack web applications.

DossierStack web services have a lot of knobs, so instead of a single function with a giant list of parameters, we get a "builder" that lets one mutably construct data for building a web application.

These "knobs" include, but are not limited to: adding routes or other Bottle applications, injecting services into routes, setting a URL prefix and adding filters and search engines.

__init__(*add_default_routes=True*)

Introduce a new builder.

You can use method chaining to configure your web application options. e.g.,

```
app = WebBuilder().enable_cors().get_app()
app.run()
```

This code will create a new Bottle web application that enables CORS (Cross Origin Resource Sharing).

If add_default_routes is False, then the default set of routes in dossier.web.routes is not added. This is only useful if you want to compose multiple Bottle applications constructed through multiple instances of WebBuilder.

get_app()

Eliminate the builder by producing a new Bottle application.

This should be the final call in your method chain. It uses all of the built up options to create a new Bottle application.

Return type bottle.Bottle

mount(*prefix*)

Mount the application on to the given URL prefix.

Parameters **prefix** (*str*) – A URL prefixi

Return type *WebBuilder*

set_config(*config_instance*)

Set the config instance.

By default, this is an instance of dossier.web.Config, which provides services like kvlclient and label_store. Custom services should probably subclass dossier.web.Config, but it's not

strictly necessary so long as it provides the same set of services (which are used for dependency injection into Bottle routes).

Parameters **config_instance** (dossier.web.Config) – A config instance.

Return type *WebBuilder*

add_search_engine(*name*, *engine*)
Adds a search engine with the given name.

engine must be the **class** object rather than an instance. The class *must* be a subclass of dossier.web.SearchEngine, which should provide a means of obtaining recommendations given a query.

The engine must be a class so that its dependencies can be injected when the corresponding route is executed by the user.

If engine is None, then it removes a possibly existing search engine named name.

Parameters

- **name** (*str*) – The name of the search engine. This appears in the list of search engines provided to the user, and is how the search engine is invoked via REST.
- **engine** (*type*) – A search engine *class*.

Return type *WebBuilder*

add_filter(*name*, *filter*)
Adds a filter with the given name.

filter must be the **class** object rather than an instance. The class *must* be a subclass of dossier.web.Filter, which should provide a means of creating a predicate function.

The filter must be a class so that its dependencies can be injected when the corresponding route is executed by the user.

If filter is None, then it removes a possibly existing filter named name.

Parameters

- **name** (*str*) – The name of the filter. This is how the search engine is invoked via REST.
- **engine** (*type*) – A filter *class*.

Return type *WebBuilder*

add_routes(*routes*)
Merges a Bottle application into this one.

Parameters **routes** (bottle.Bottle or *[bottle route]*.) – A Bottle application or a sequence of routes.

Return type *WebBuilder*

inject(*name*, *closure*)
Injects closure() into name parameters in routes.

This sets up dependency injection for parameters named name. When a route is invoked that has a parameter name, then closure() is passed as that parameter's value.

(The closure indirection is so the caller can control the time of construction for objects. For example, you may want to check the health of a database connection.)

Parameters

- **name** (*str*) – Parameter name.

- **closure** (*function*) – A function with no parameters.

Return type *WebBuilder*

enable_cors()
Enables Cross Origin Resource Sharing.

This makes sure the necessary headers are set so that this web application's routes can be accessed from other origins.

Return type *WebBuilder*

Here are the available search engines by default:

class dossier.web.search_engines.**plain_index_scan**(*store*)
Return a random sample of an index scan.

This scans all indexes defined for all values in the query corresponding to those indexes.

class dossier.web.search_engines.**random**(*store*)
Return random results with the same name.

This finds all content objects that have a matching name and returns `limit` results at random.

If there is no `NAME` index defined, then this always returns no results.

Here are the available filter predicates by default:

class dossier.web.filters.**already_labeled**(*label_store*)
Filter results that have a label associated with them.

If a result has a *direct* label between it and the query, then it will be removed from the list of results.

class dossier.web.filters.**nilsimsa_near_duplicates**(*label_store*, *store*, *nilsimsa_feature_name='nilsimsa_all'*, *threshold=119*)
Filter results that nilsimsa says are highly similar.

To perform an filtering, this requires that the FCs carry StringCounter at *nilsimsa_feature_name* and results with nilsimsa comparison higher than the *threshold* are filtered. *threshold* defaults to 119, which is in the range [-128, 128] per the definition of nilsimsa. *nilsimsa_feature_name* defaults to 'nilsimsa_all'.

A note about speed performance: the order complexity of this filter is linear in the number of results that get through the filter. While that is unfortunate, it is inherent to the nature of using comparison-based locality sensitive hashing (LSH). Other LSH techniques, such as shingle hashing with simhash tend to have less fidelity, but can be efficiently indexed to allow O(1) lookups in a filter like this.

Before refactoring this to use nilsimsa directly, this was using a "kernel" function that had nilsimsa buried inside it, and it had this kind of speed performance:

dossier/web/tests/test_filter_preds.py::test_near_duplicates_speed_perf 4999 filtered to 49 in 2.838213 seconds, 1761.319555 per second

After refactoring to use nilsimsa directly in this function, the constant factors get better, and the order complexity is still linear in the number of items that the filter has emitted, because it has to remember them and scan over them. Thresholding in the nilsimsa.compare_digests function helps considerably: four times faster on this synthetic test data when there are many different documents, which is the typical case:

Without thresholding in the nilsimsa.compare_digests: dossier/web/tests/test_filter_preds.py::test_nilsimsa_near_duplicates_speed
5049 filtered to 49 in 0.772274 seconds, 6537.834870 per second
dossier/web/tests/test_filter_preds.py::test_nilsimsa_near_duplicates_speed_perf
1049 filtered to 49 in 0.162775 seconds, 6444.477004 per second
dossier/web/tests/test_filter_preds.py::test_nilsimsa_near_duplicates_speed_perf 209 filtered to 9 in 0.009348 seconds, 22357.355097 per second

With thresholding in the nilsimsa.compare_digests: dossier/web/tests/test_filter_preds.py::test_nilsimsa_near_duplicates_speed_p 5049 filtered to 49 in 0.249705 seconds, 20219.853262 per second dossier/web/tests/test_filter_preds.py::test_nilsimsa_near_duplicates_speed_perf 1549 filtered to 49 in 0.112724 seconds, 13741.549025 per second dossier/web/tests/test_filter_preds.py::test_nilsimsa_near_duplicates_speed_perf 209 filtered to 9 in 0.009230 seconds, 22643.802754 per second

Some useful utility functions.

`dossier.web.`**`streaming_sample`**(*seq*, *k*, *limit=None*)

Streaming sample.

Iterate over seq (once!) keeping k random elements with uniform distribution.

As a special case, if `k` is `None`, then `list(seq)` is returned.

Parameters

- **`seq`** – iterable of things to sample from
- **`k`** – size of desired sample
- **`limit`** – stop reading `seq` after considering this many

Returns list of elements from seq, length k (or less if seq is short)

5.1.1 Search engine and filter interfaces

class `dossier.web.interface.`**`SearchEngine`**

Bases: *dossier.web.interface.Queryable*

Defines an interface for search engines.

A search engine, at a high level, takes a query feature collection and returns a list of results, where each result is itself a feature collection.

Note that this is an abstract class. Implementors must provide the *SearchEngine.recommendations()* method.

`__init__`()

Create a new search engine.

The creation of a search engine is distinct from the operation of a search engine. Namely, the creation of a search engine is subject to dependency injection. The following parameters are special in that they will be automatically populated with special values if present in your `__init__`:

•**kvlclient**: `kvlayer._abstract_storage.AbstractStorage`

•**store**: *dossier.store.Store*

•**label_store**: *dossier.label.LabelStore*

If you want to expand the set of items that can be injected, then you must subclass `dossier.web.Config`, define your new services as instance attributes, and set your new config instance with `dossier.web.Config.set_config()`.

Return type A callable with a signature isomorphic to `dossier.web.SearchEngine.__call__()`.

`recommendations`()

Return recommendations.

The return type is loosely specified. In particular, it must be a dictionary with at least one key, `results`, which maps to a list of tuples of `(content_id, FC)`. The returned dictionary may contain other keys.

results()
Returns results as a JSON encodable Python value.

This calls *SearchEngine.recommendations()* and converts the results returned into JSON encodable values. Namely, feature collections are slimmed down to only features that are useful to an end-user.

respond(*response*)
Perform the actual web response.

This is usually just a JSON encoded dump of the search results, but implementors may choose to implement this differently (e.g., with a cache).

Parameters **response** (bottle.Response) – A web response object.

Return type *str*

add_filter(*name*, *filter*)
Add a filter to this search engine.

Parameters **filter** (*Filter*) – A filter.

Return type *SearchEngine*

create_filter_predicate()
Creates a filter predicate.

The list of available filters is given by calls to add_filter, and the list of filters to use is given by parameters in params.

In this default implementation, multiple filters can be specified with the filter parameter. Each filter is initialized with the same set of query parameters given to the search engine.

The returned function accepts a (content_id, FC) and returns True if and only if every selected predicate returns True on the same input.

class dossier.web.interface.**Filter**
Bases: *dossier.web.interface.Queryable*

A filter for results returned by search engines.

A filter is a yakonfig.Configurable object (or one that can be auto-configured) that returns a callable for creating a predicate that will filter results produced by a search engine.

A filter has one abstract method: *Filter.create_predicate()*.

create_predicate()
Creates a predicate for this filter.

The predicate should accept a tuple of (content_id, FC) and return True if and only if the given result should be included in the list of recommendations provided to the user.

class dossier.web.interface.**Queryable**
Queryable supports parameterization from URLs and config.

Queryable is meant to be subclassed by things that have two fundamental things in common:

1. Requires a single query identifier.
2. Can be optionally configured from either user provided URL parameters or admin provided configuration.

Queryable provides a common interface for these two things, while also providing a way to declare a schema for the parameters. This schema is used to convert values from the URL/config into typed Python values.

Parameter schema

The `param_schema` class variable can define rudimentary type conversion from strings to typed Python values such as `unicode` or `int`.

`param_schema` is a dictionary that maps keys (parameter name) to a parameter type. A parameter type is itself a dictionary with the following keys:

type Required. Must be one of `'bool'`, `'int'`, `'float'`, `'bytes'` or `'unicode'`.

min Optional for `'int'` and `'float'` types. Specifies a minimum value.

max Optional for `'int'` and `'float'` types. Specifies a maximum value.

encoding Specifies an encoding for `'unicode'` types.

If you want to inherit the schema of a parent class, then you can use:

```
param_schema = dict(ParentClass.param_schema, **{
    # your extra types here
})
```

Variables

- **query_content_id** – The query content id.
- **query_params** – The raw query parameters, as a `bottle.MultiDict`.
- **config_params** – The raw configuration parameters. This must be maintained explicitly, but will be incorporated in the values for `params`. If `k` is and `config_params`, then `k`'s default value is `config_params[k]` (which is overridden by `query_params[k]` if it exists).
- **params** – The combined and typed values of `query_params` and `config_params`.

__init__()

Creates a new instance of *Queryable*.

This initializes a default empty state, where all parameter dictionaries are empty and `query_content_id` is `None`.

To take advantage of dependency injected configuration, you'll want to write your own constructor that sets config parameters explicitly:

```
def __init__(self, param1=None, param2=5):
    self.config_params = {
        'param1': param1,
        'param2': param2,
    }
    super(MyClass, self).__init__()
```

It's important to call the constructor after `config_params` has been set so that the schema is applied correctly.

set_query_id(*query_content_id*)

Set the query id.

Parameters **query_content_id** (*str*) – The query content identifier.

Return type *Queryable*

set_query_params(*query_params*)

Set the query parameters.

The query parameters should be a dictionary mapping keys to strings or lists of strings.

Parameters **query_params** (`name |--> (str | [str])`) – query parameters

Return type *Queryable*

add_query_params (*query_params*)
Overwrite the given query parameters.

This is the same as *Queryable.set_query_params()*, except it overwrites existing parameters individually whereas `set_query_params` deletes all existing key in `query_params`.

5.1.2 Web service for active learning

dossier.web.routes is a REST stateful web service that can drive Dossier Stack's an active ranking models and user interface, as well as other search technologies.

There are only a few API end points. They provide searching, storage and retrieval of feature collections along with storage of ground truth data as labels. Labels are typically used in the implementation of a search engine to filter or improve the recommendations returned.

The API end points are documented as functions in this module.

`dossier.web.routes.`**v1_search** (*request*, *response*, *visid_to_dbid*, *config*, *search_engines*, *filters*, *cid*, *engine_name*)
Search feature collections.

The route for this endpoint is: `/dossier/v1/<content_id>/search/<search_engine_name>`.

`content_id` can be any *profile* content identifier. (This restriction may be lifted at some point.) Namely, it must start with `p|`.

`engine_name` corresponds to the search strategy to use. The list of available search engines can be retrieved with the *v1_search_engines()* endpoint.

This endpoint returns a JSON payload which is an object with a single key, `results`. `results` is a list of objects, where the objects each have `content_id` and `fc` attributes. `content_id` is the unique identifier for the result returned, and `fc` is a JSON serialization of a feature collection.

There are also two query parameters:

- **limit** limits the number of results to the number given.
- **filter** sets the filtering function. The default filter function, `already_labeled`, will filter out any feature collections that have already been labeled with the query `content_id`.

`dossier.web.routes.`**v1_search_engines** (*search_engines*)
List available search engines.

The route for this endpoint is: `/dossier/v1/search_engines`.

This endpoint returns a JSON payload which is an object with two keys: `default` and `names`. `default` corresponds to a chosen default search engine. This value will *always* correspond to a valid search engine. `names` is an array of all available search engines (including `default`).

`dossier.web.routes.`**v1_fc_get** (*visid_to_dbid*, *store*, *cid*)
Retrieve a single feature collection.

The route for this endpoint is: `/dossier/v1/feature-collections/<content_id>`.

This endpoint returns a JSON serialization of the feature collection identified by `content_id`.

`dossier.web.routes.`**v1_fc_put** (*request*, *response*, *visid_to_dbid*, *store*, *cid*)
Store a single feature collection.

The route for this endpoint is: `PUT /dossier/v1/feature-collections/<content_id>`.

content_id is the id to associate with the given feature collection. The feature collection should be in the request body serialized as JSON.

This endpoint returns status 201 upon successful storage. An existing feature collection with id content_id is overwritten.

dossier.web.routes.**v1_random_fc_get**(*response*, *dbid_to_visid*, *store*)
Retrieves a random feature collection from the database.

The route for this endpoint is: GET /dossier/v1/random/feature-collection.

Assuming the database has at least one feature collection, this end point returns an array of two elements. The first element is the content id and the second element is a feature collection (in the same format returned by *dossier.web.routes.v1_fc_get()*).

If the database is empty, then a 404 error is returned.

Note that currently, this may not be a uniformly random sample.

dossier.web.routes.**v1_label_put**(*request*, *response*, *visid_to_dbid*, *config*, *label_hooks*, *label_store*, *cid1*, *cid2*, *annotator_id*)
Store a single label.

The route for this endpoint is: PUT /dossier/v1/labels/<content_id1>/<content_id2>/<annotator_id>.

content_id are the ids of the feature collections to associate. annotator_id is a string that identifies the human that created the label. The value of the label should be in the request body as one of the following three values: -1 for not coreferent, 0 for "I don't know if they are coreferent" and 1 for coreferent.

Optionally, the query parameters subtopic_id1 and subtopic_id2 may be specified. Neither, both or either may be given. subtopic_id1 corresponds to a subtopic in content_id1 and subtopic_id2 corresponds to a subtopic in content_id2.

This endpoint returns status 201 upon successful storage. Any existing labels with the given ids are overwritten.

dossier.web.routes.**v1_label_direct**(*request*, *response*, *visid_to_dbid*, *dbid_to_visid*, *label_store*, *cid*, *subid=None*)
Return directly connected labels.

The routes for this endpoint are /dossier/v1/label/<cid>/direct and /dossier/v1/label/<cid>/subtopic/<subid>/direct.

This returns all directly connected labels for cid. Or, if a subtopic id is given, then only directly connected labels for (cid, subid) are returned.

The data returned is a JSON list of labels. Each label is a dictionary with the following keys: content_id1, content_id2, subtopic_id1, subtopic_id2, annotator_id, epoch_ticks and value.

dossier.web.routes.**v1_label_connected**(*request*, *response*, *visid_to_dbid*, *dbid_to_visid*, *label_store*, *cid*, *subid=None*)
Return a connected component of positive labels.

The routes for this endpoint are /dossier/v1/label/<cid>/connected and /dossier/v1/label/<cid>/subtopic/<subid>/connected.

This returns the edges for the connected component of either cid or (cid, subid) if a subtopic identifier is given.

The data returned is a JSON list of labels. Each label is a dictionary with the following keys: content_id1, content_id2, subtopic_id1, subtopic_id2, annotator_id, epoch_ticks and value.

dossier.web.routes.**v1_label_expanded**(*request*, *response*, *label_store*, *visid_to_dbid*, *dbid_to_visid*, *cid*, *subid=None*)
Return an expansion of the connected component of positive labels.

The routes for this endpoint are `/dossier/v1/label/<cid>/expanded` and `/dossier/v1/label/<cid>/subtopic/<subid>/expanded`.

This returns the edges for the expansion of the connected component of either `cid` or `(cid, subid)` if a subtopic identifier is given. Note that the expansion of a set of labels does not provide any new information content over a connected component. It is provided as a convenience for clients that want all possible labels in a connected component, regardless of whether one explicitly exists or not.

The data returned is a JSON list of labels. Each label is a dictionary with the following keys: `content_id1`, `content_id2`, `subtopic_id1`, `subtopic_id2`, `annotator_id`, `epoch_ticks` and `value`.

`dossier.web.routes.`**`v1_label_negative_inference`**(*request*, *response*, *visid_to_dbid*, *dbid_to_visid*, *label_store*, *cid*)

Return inferred negative labels.

The route for this endpoint is: `/dossier/v1/label/<cid>/negative-inference`.

Negative labels are inferred by first getting all other content ids connected to `cid` through a negative label. For each directly adjacent `cid'`, the connected components of `cid` and `cid'` are traversed to find negative labels.

The data returned is a JSON list of labels. Each label is a dictionary with the following keys: `content_id1`, `content_id2`, `subtopic_id1`, `subtopic_id2`, `annotator_id`, `epoch_ticks` and `value`.

5.1.3 Managing folders and sub-folders

In many places where active learning is used, it can be useful to provide the user with a means to group and categorize topics. In an active learning setting, it is essential that we try to capture a user's grouping of topics so that it can be used for ground truth data. To that end, `dossier.web` exposes a set of web service endpoints for managing folders and subfolders for a particular user. Folders and subfolders are stored and managed by *`dossier.label`*, which means they are automatically available as ground truth data.

The actual definition of what a folder or subfolder is depends on the task the user is trying to perform. We tend to think of a folder as a general topic and a subfolder as a more specific topic or "subtopic." For example, a topic might be "cars" and some subtopics might be "dealerships with cars I want to buy" or "electric cars."

The following end points allow one to add or list folders and subfolders. There is also an endpoint for listing all of the items in a single subfolder, where each item is a pair of `(content_id, subtopic_id)`.

In general, the identifier of a folder/subfolder is also used as its name, similar to how identifiers in Wikipedia work. For example, if a folder has a name "My Cars", then its identifier is `My_Cars`. More specifically, given any folder name `NAME`, its corresponding identifier can be obtained with `NAME.replace(' ', '_')`.

All web routes accept and return *identifiers* (so space characters are disallowed).

`dossier.web.routes.`**`v1_folder_list`**(*request*, *kvlclient*)

Retrieves a list of folders for the current user.

The route for this endpoint is: `GET /dossier/v1/folder`.

(Temporarily, the "current user" can be set via the `annotator_id` query parameter.)

The payload returned is a list of folder identifiers.

`dossier.web.routes.`**`v1_folder_add`**(*request*, *response*, *kvlclient*, *fid*)

Adds a folder belonging to the current user.

The route for this endpoint is: `PUT /dossier/v1/folder/<fid>`.

If the folder was added successfully, `201` status is returned.

(Temporarily, the "current user" can be set via the `annotator_id` query parameter.)

dossier.web.routes.**v1_subfolder_list**(*request*, *response*, *kvlclient*, *fid*)
Retrieves a list of subfolders in a folder for the current user.

The route for this endpoint is: `GET /dossier/v1/folder/<fid>/subfolder`.

(Temporarily, the "current user" can be set via the `annotator_id` query parameter.)

The payload returned is a list of subfolder identifiers.

dossier.web.routes.**v1_subfolder_add**(*request*, *response*, *kvlclient*, *fid*, *sfid*, *cid*, *subid=None*)
Adds a subtopic to a subfolder for the current user.

The route for this endpoint is: `PUT /dossier/v1/folder/<fid>/subfolder/<sfid>/<cid>/<subid>`.

`fid` is the folder identifier, e.g., `My_Folder`.

`sfid` is the subfolder identifier, e.g., `My_Subtopic`.

`cid` and `subid` are the content id and subtopic id of the subtopic being added to the subfolder.

If the subfolder does not already exist, it is created automatically. N.B. An empty subfolder cannot exist!

If the subtopic was added successfully, `201` status is returned.

(Temporarily, the "current user" can be set via the `annotator_id` query parameter.)

dossier.web.routes.**v1_subtopic_list**(*request*, *response*, *kvlclient*, *fid*, *sfid*)
Retrieves a list of items in a subfolder.

The route for this endpoint is: `GET /dossier/v1/folder/<fid>/subfolder/<sfid>`.

(Temporarily, the "current user" can be set via the `annotator_id` query parameter.)

The payload returned is a list of two element arrays. The first element in the array is the item's content id and the second element is the item's subtopic id.

dossier.web.routes.**v1_folder_delete**(*request*, *response*, *kvlclient*, *fid*, *sfid=None*, *cid=None*, *subid=None*)
Deletes a folder, subfolder or item.

The routes for this endpoint are:

- `DELETE /dossier/v1/folder/<fid>`
- `DELETE /dossier/v1/folder/<fid>/subfolder/<sfid>`
- `DELETE /dossier/v1/folder/<fid>/subfolder/<sfid>/<cid>`
- `DELETE /dossier/v1/folder/<fid>/subfolder/<sfid>/<cid>/<subid>`

dossier.web.routes.**v1_folder_rename**(*request*, *response*, *kvlclient*, *fid_src*, *fid_dest*, *sfid_src=None*, *sfid_dest=None*)
Rename a folder or a subfolder.

The routes for this endpoint are:

- `POST /dossier/v1/<fid_src>/rename/<fid_dest>`
- `POST /dossier/v1/<fid_src>/subfolder/<sfid_src>/rename/ <fid_dest>/subfolder/<sfid_dest>`

Foldering for Dossier Stack.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

d

Symbols

A

C

D

E

F

G