Snorkel Documentation

Release 0.4.0

Alex Ratner, Stephen Bach, Henry Ehrenberg

January 13, 2017

Contents


snorkel

CHAPTER 1

Contexts

Preprocessed input data is represented in Snorkel as a hierarchy of *Context* subclass objects. For example, as is currently default for text: Corpus -> Document -> Sentence -> Span.

1.1 Core Data Models

class snorkel.models.context.**Context**(***kwargs*)
A piece of content from which Candidates are composed.

class snorkel.models.context.**Corpus**(***kwargs*)
A set of Documents, uniquely identified by a name.

Corpora have many-to-many relationships with Documents, so users can create new subsets, supersets, etc.

child_context_stats(*parent_context*)
Given a parent context class, gets all the child context classes, and returns histograms of the number of children per parent.

stats()
Print summary / diagnostic stats about the corpus

class snorkel.models.context.**Document**(***kwargs*)
A root Context.

class snorkel.models.context.**Sentence**(***kwargs*)
A sentence Context in a Document.

class snorkel.models.context.**Span**(***kwargs*)
A span of characters, identified by Context id and character-index start, end (inclusive).

char_offsets are **relative to the Context start**

class snorkel.models.context.**TemporaryContext**
A context which does not incur the overhead of a proper ORM-based Context object. The TemporaryContext class is specifically for the candidate extraction process, during which a CandidateSpace object will generate many TemporaryContexts, which will then be filtered by Matchers prior to materialization of Candidates and constituent Context objects.

Every Context object has a corresponding TemporaryContext object from which it inherits.

A TemporaryContext must have specified equality / set membership semantics, a stable_id for checking uniqueness against the database, and a promote() method which returns a corresponding Context object.

class snorkel.models.context.**TemporarySpan**(*parent*, *char_start*, *char_end*, *meta=None*)
The TemporaryContext version of Span

char_to_word_index(*ci*)
Given a character-level index (offset), return the index of the **word this char is in**

get_attrib_span(*a*, *sep=' '*)
Get the span of sentence attribute _a_ over the range defined by word_offset, n

get_attrib_tokens(*a='words'*)
Get the tokens of sentence attribute _a_ over the range defined by word_offset, n

word_to_char_index(*wi*)
Given a word-level index, return the character-level index (offset) of the word's start

snorkel.models.context.**construct_stable_id**(*parent_context*, *polymorphic_type*, *relative_char_offset_start*, *relative_char_offset_end*)
Contruct a stable ID for a Context given its parent and its character offsets relative to the parent

snorkel.models.context.**split_stable_id**(*stable_id*)

Split stable id, returning:

- Document (root) stable ID
- Context polymorphic type
- Character offset start, end *relative to document start*

Returns tuple of four values.

1.2 Core Objects for Preprocessing and Loading

class snorkel.parser.**CorpusParser**(*doc_parser*, *sent_parser*, *max_docs=None*)
Invokes a DocParser and runs the output through a SentenceParser to produce a Corpus.

class snorkel.parser.**DocParser**(*path*, *encoding='utf-8'*)
Parse a file or directory of files into a set of Document objects.

parse()
Parse a file or directory of files into a set of Document objects.

•Input: A file or directory path.

•**Output: A set of Document objects, which at least have a _text_ attribute,** and possibly a dictionary of other attributes.

class snorkel.parser.**HTMLDocParser**(*path*, *encoding='utf-8'*)
Simple parsing of raw HTML files, assuming one document per file

class snorkel.parser.**TSVDocParser**(*path*, *encoding='utf-8'*)
Simple parsing of TSV file with one (doc_name <tab> doc_text) per line

class snorkel.parser.**TextDocParser**(*path*, *encoding='utf-8'*)
Simple parsing of raw text files, assuming one document per file

class snorkel.parser.**XMLMultiDocParser**(*path*, *doc='.//document'*, *text='./text/text()'*, *id='./id/text()'*, *keep_xml_tree=False*)
Parse an XML file _which contains multiple **documents**_ into a set of Document objects.

Use XPath queries to specify a _document_ object, and then for each document, a set of _text_ sections and an _id_.

Note: Include the full document XML etree in the attribs dict with keep_xml_tree=True

CHAPTER 2

Candidates

In order to apply machine learning–i.e., in this case, a classifier–to information extraction problems, we need to have a base set of objects that are being classified. In Snorkel, these are the *Candidate* subclasses, which are defined over *Context* arguments, and represent *potential* mentions to extract. We use *Matcher* operators to extract a set of *Candidate* objects from the input data.

2.1 Core Data Models

class snorkel.models.candidate.**Candidate**(***kwargs*)
An abstract candidate relation.

New relation types should be defined by calling candidate_subclass(), **not** subclassing this class directly.

class snorkel.models.candidate.**CandidateSet**(***kwargs*)
A set of Candidates, uniquely identified by a name.

CandidateSets have many-to-many relationships with Candidates, so users can create new subsets, supersets, etc.

stats(*gold_set=None*)
Print diagnostic stats about CandidateSet.

snorkel.models.candidate.**candidate_subclass**(*class_name*, *args*, *table_name=None*)
Creates and returns a Candidate subclass with provided argument names, which are Context type. Creates the table in DB if does not exist yet.

Import using:

```
from snorkel.models import candidate_subclass
```

Parameters

- **class_name** – The name of the class, should be "camel case" e.g. NewCandidateClass
- **args** – A list of names of consituent arguments, which refer to the Contexts–representing mentions–that comprise the candidate
- **table_name** – The name of the corresponding table in DB; if not provided, is converted from camel case by default, e.g. new_candidate_class

2.2 Core Objects for Candidate Extraction

class snorkel.candidates.**CandidateExtractor**(*candidate_class*, *cspaces*, *matchers*, *self_relations=False*, *nested_relations=False*, *symmetric_relations=True*)

An operator to extract Candidate objects from Context objects.

Parameters

- **candidate_class** – The type of relation to extract, defined using *snorkel.models.candidate_subclass*
- **cspaces** – one or list of *CandidateSpace* objects, one for each relation argument. Defines space of Contexts to consider
- **matchers** – one or list of *snorkel.matchers.Matcher* objects, one for each relation argument. Only tuples of Contexts for which each element is accepted by the corresponding Matcher will be returned as Candidates
- **self_relations** – Boolean indicating whether to extract Candidates that relate the same context. Only applies to binary relations. Default is False.
- **nested_relations** – Boolean indicating whether to extract Candidates that relate one Context with another that contains it. Only applies to binary relations. Default is False.
- **symmetric_relations** – Boolean indicating whether to extract symmetric Candidates, i.e., rel(A,B) and rel(B,A), where A and B are Contexts. Only applies to binary relations. Default is True.

class snorkel.candidates.**CandidateSpace**

Defines the **space** of candidate objects Calling _apply(x)_ given an object _x_ returns a generator over candidates in _x_.

class snorkel.candidates.**Ngrams**(*n_max=5, split_tokens=['-', '/']*)

Defines the space of candidates as all n-grams (n <= n_max) in a Sentence _x_, indexing by **character offset**.

snorkel.candidates.**gold_stats**(*candidates*, *gold*)

Return precision and recall relative to a "gold" CandidateSet

class snorkel.matchers.**Concat**(**children*, ***opts*)

Selects candidates which are the concatenation of adjacent matches from child operators NOTE: Currently slices on **word index** and considers concatenation along these divisions only

class snorkel.matchers.**DateMatcher**(***kwargs*)

Matches Spans that are dates, as identified by CoreNLP.

A convenience class for setting up a RegexMatchEach to match spans for which each token was tagged as a date.

class snorkel.matchers.**DictionaryMatch**(**children*, ***opts*)

Selects candidate Ngrams that match against a given list d

class snorkel.matchers.**LambdaFunctionMatch**(**children*, ***opts*)

Selects candidate Ngrams that match against a given list d

class snorkel.matchers.**LocationMatcher**(***kwargs*)

Matches Spans that are the names of locations, as identified by CoreNLP.

A convenience class for setting up a RegexMatchEach to match spans for which each token was tagged as a location.

class snorkel.matchers.**Matcher**(**children*, ***opts*)
Applies a function f : c -> {True,False} to a generator of candidates, returning only candidates _c_ s.t. _f(c) == True_, where f can be compositionally defined.

apply(*candidates*)
Apply the Matcher to a **generator** of candidates Optionally only takes the longest match (NOTE: assumes this is the *first* match)

f(*c*)
The recursively composed version of filter function f By default, returns logical **conjunction** of operator and single child operator

class snorkel.matchers.**MiscMatcher**(***kwargs*)
Matches Spans that are miscellaneous named entities, as identified by CoreNLP.

A convenience class for setting up a RegexMatchEach to match spans for which each token was tagged as miscellaneous.

class snorkel.matchers.**NgramMatcher**(**children*, ***opts*)
Matcher base class for Ngram objects

class snorkel.matchers.**NumberMatcher**(***kwargs*)
Matches Spans that are numbers, as identified by CoreNLP.

A convenience class for setting up a RegexMatchEach to match spans for which each token was tagged as a number.

class snorkel.matchers.**OrganizationMatcher**(***kwargs*)
Matches Spans that are the names of organizations, as identified by CoreNLP.

A convenience class for setting up a RegexMatchEach to match spans for which each token was tagged as an organization.

class snorkel.matchers.**PersonMatcher**(***kwargs*)
Matches Spans that are the names of people, as identified by CoreNLP.

A convenience class for setting up a RegexMatchEach to match spans for which each token was tagged as a person.

class snorkel.matchers.**RegexMatch**(**children*, ***opts*)
Base regex class- does not specify specific semantics of *what* is being matched yet

class snorkel.matchers.**RegexMatchEach**(**children*, ***opts*)
Matches regex pattern on **each token**

class snorkel.matchers.**RegexMatchSpan**(**children*, ***opts*)
Matches regex pattern on **full concatenated span**

class snorkel.matchers.**SlotFillMatch**(**children*, ***opts*)
Matches a slot fill pattern of matchers _at the character **level**_

class snorkel.matchers.**Union**(**children*, ***opts*)
Takes the union of candidate sets returned by child operators

CHAPTER 3

Annotations

One of the core operations in Snorkel is _annotating_ the candidates in various ways. We can think of generating features for the candidates as annotating them (creating a *Feature* object), and can also view supervision via labeling functions as annotating them (creating a *Label* object).

3.1 Core Data Models

class snorkel.models.annotation.**AnnotationKey**(***kwargs*)
The Annotation key table is a mapping from unique string names to integer id numbers. These strings uniquely identify who or what produced an annotation.

class snorkel.models.annotation.**AnnotationKeySet**(***kwargs*)
A many-to-many set of AnnotationKeys.

class snorkel.models.annotation.**AnnotationMixin**
Mixin class for defining annotation tables.

An annotation is a value associated with a Candidate. Examples include labels, features, and predictions.

New types of annotations can be defined by creating an annotation class and corresponding annotation, for example:

```
from snorkel.models.annotation import AnnotationMixin
from snorkel.models.meta import SnorkelBase

class NewAnnotation(AnnotationMixin, SnorkelBase):
    value = Column(Float, nullable=False)


# The entire storage schema, including NewAnnotation, can now be initialized with the following
import snorkel.models
```

The annotation class should include a Column attribute named value.

class snorkel.models.annotation.**Feature**(***kwargs*)
An element of a representation of a Candidate in a feature space.

A Feature's annotation key identifies the definition of the Feature, e.g., a function that implements it or the library name and feature name in an automatic featurization library.

class snorkel.models.annotation.**Label**(***kwargs*)
A discrete label associated with a Candidate, indicating a target prediction value.

Labels are used to represent both human-provided annotations and the output of labeling functions.

A Label's annotation key identifies the person or labeling function that provided the Label.

class snorkel.models.annotation.**Prediction**(***kwargs*)

A probability associated with a Candidate, indicating the degree of belief that the Candidate is true.

A Prediction's annotation key indicates which process or method produced the Prediction, e.g., which model with which ParameterSet.

3.2 Core Objects for Annotations (Features, Labels)

CHAPTER 4

Learning

4.1 Core Data Models

4.2 Core Objects for Learning + Inference

class snorkel.learning.**LSTM**
Long Short-Term Memory.

class snorkel.learning.**LogRegSKLearn**
Logistic regression.

class snorkel.learning.**NoiseAwareModel**(*bias_term=False*)
Simple abstract base class for a model.

load(*session*, *param_set_name*)
Load the Parameters into self.w, given ParameterSet.name

predict(*X*, *b=0.5*)
Return numpy array of elements in {-1,0,1} based on predicted marginal probabilities.

save(*session*, *param_set_name*)
Save the Parameter (weight) values, i.e. the model, as a new ParameterSet

train(*X*, *training_marginals*, ***hyperparams*)
Trains the model; also must set self.X_train and self.w

snorkel.learning.**exact_data**(*X*, *w*, *evidence=None*)
We calculate the exact conditional probability of the decision variables in logistic regression; see sample_data

snorkel.learning.**log_odds**(*p*)
This is the logit function

snorkel.learning.**odds_to_prob**(*l*)

This is the inverse logit function logit^{-1}:

l = log

rac{p}{1-p} exp(l) =

rac{p}{1-p} p =

rac{exp(l)}{1 + exp(l)}

snorkel.learning.**sample_data**(*X*, *w*, *n_samples*)

Here we do Gibbs sampling over the decision variables (representing our objects), o_j corresponding to the columns of X The model is just logistic regression, e.g.

P(o_j=1 | X_{*,j}; w) = *logit^{-1}(w dot X_{*,j})*

This can be calculated exactly, so this is essentially a noisy version of the exact calc...

snorkel.learning.**transform_sample_stats**(*Xt*, *t*, *f*, *Xt_abs=None*)

Here we calculate the expected accuracy of each LF/feature (corresponding to the rows of X) wrt to the distribution of samples S:

E_S[accuracy_i] = E_(t,f)[

rac{TP + TN}{TP + FP + TN + FN}] =

rac{X_{i|x_{ij}>0}*t - X_{i|x_{ij}<0}*f}{t+f} =

rac12left(rac{X*(t-f)}{t+f} + 1 ight)

CHAPTER 5

Etc: Viewing and Annotating Data, Writing LFs

5.1 Using the Viewer to Inspect and Annotate Data

5.2 Helpers for Writing Labeling Functions

snorkel.lf_helpers.**contains_token**(*c*, *tok*, *attrib='words'*, *case_sensitive=False*)
Checks if any of the contituent Spans contain a token :param attrib: The token attribute type (e.g. words, lemmas, poses)

snorkel.lf_helpers.**get_between_tokens**(*c*, *attrib='words'*, *n_max=1*, *case_sensitive=False*)
TODO: write doc_string

snorkel.lf_helpers.**get_doc_candidate_spans**(*c*)
Get the Spans in the same document as Candidate c, where these Spans are arguments of Candidates.

snorkel.lf_helpers.**get_left_tokens**(*c*, *window=3*, *attrib='words'*, *n_max=1*, *case_sensitive=False*)
Return the tokens within a window to the _left_ of the Candidate. For higher-arity Candidates, defaults to the _first_ argument. :param window: The number of tokens to the left of the first argument to return :param attrib: The token attribute type (e.g. words, lemmas, poses)

snorkel.lf_helpers.**get_matches**(*lf, candidate_set, match_values=[1, -1]*)
A simple helper function to see how many matches (non-zero by default) an LF gets. Returns the matched set, which can then be directly put into the Viewer.

snorkel.lf_helpers.**get_right_tokens**(*c*, *window=3*, *attrib='words'*, *n_max=1*, *case_sensitive=False*)
Return the tokens within a window to the _right_ of the Candidate. For higher-arity Candidates, defaults to the _last_ argument. :param window: The number of tokens to the right of the last argument to return :param attrib: The token attribute type (e.g. words, lemmas, poses)

snorkel.lf_helpers.**get_sent_candidate_spans**(*c*)
Get the Spans in the same Sentence as Candidate c, where these Spans are arguments of Candidates.

snorkel.lf_helpers.**get_tagged_text**(*c*)
Returns the text of c's parent context with c's unary spans replaced with tags {{A}}, {{B}}, etc. A convenience method for writing LFs based on e.g. regexes.

snorkel.lf_helpers.**get_text_between**(*c*)
Returns the text between the two unary Spans of a binary-Span Candidate, where both are in the same Sentence.

snorkel.lf_helpers.**get_text_splits**(*c*)
Given a k-arity Candidate defined over k Spans, return the chunked parent context (e.g. Sentence) split around the k constituent Spans.

NOTE: Currently assumes that these Spans are in the same Context

snorkel.lf_helpers.**is_inverted**(*c*)
Returns True if the ordering of the candidates in the sentence is inverted.

5.3 Helpers for Loading External Annotations

class snorkel.loaders.**ExternalAnnotationsLoader**(*session*, *candidate_class*, *candidate_set*, *annotation_key*, *expand_candidate_set=False*)
Class to load external annotations.

add(*temp_contexts*)
Adds a candidate to a new or existing candidate_set.

Parameters **temp_contexts** – This is a *dictionary* of *TemporaryContext* objects corresponding to the args of

the Candidate class.

snorkel.loaders.**create_or_fetch**(*session*, *set_class*, *instance_or_name*)
Returns a named set ORM object given an instance or name as string

S