concept_formation Documentation

Release 0.3.2

Christopher J MacLellan and Erik Harpstead

Sep 08, 2017

Contents

This is a Python library of algorithms that perform concept formation written by Christopher MacLellan (http://www.christopia.net) and Erik Harpstead (http://www.erikharpstead.net).

CHAPTER 1

Overview

In this library, the COBWEB and COBWEB/3 algorithms are implemented. These systems accept a stream of instances, which are represented as dictionaries of attributes and values (where values can be nominal for COBWEB and either numeric or nominal for COBWEB/3), and learns a concept hierarchy. The resulting hierarchy can be used for clustering and prediction.

This library also includes TRESTLE, an extension of COBWEB and COBWEB/3 that support structured and relational data objects. This system employs partial matching to rename new objects to align with previous examples, then categorizes these renamed objects.

Lastly, we have extended the COBWEB/3 algorithm to support three key improvements. First, COBWEB/3 now uses an unbiased estimator to calculate the standard deviation of numeric values. This is particularly useful in situations where the number of available data points is low. Second, COBWEB/3 supports online normalization of the continuous values, which is useful in situations where numeric values are on different scales and helps to ensure that numeric values do not impact the model more than nominal values. Finally, it is assumed that there is some base noise in measuring continuous values, this noise insures that the probability of any one value never exceeds 1, even when the standard deviation is small.

CHAPTER 2

Installation

You can install this software using pip:

```
pip install -U concept_formation
```

You can install the latest version of the code directly from github:

```
pip install -U git+https://github.com/cmaclell/concept_formation@master
```

CHAPTER 3

Important Links

- Source code: https://github.com/cmaclell/concept_formation
- Documentation: http://concept-formation.readthedocs.org

CHAPTER 4

Examples

We have created a number of examples to demonstrate the basic functionality of this library. The examples can be found here.

CHAPTER 5

Citing this Software

If you use this software in a scientific publiction, then we would appreciate citation of the following paper:

MacLellan, C.J., Harpstead, E., Aleven, V., Koedinger K.R. (2016) TRESTLE: A Model of Concept Formation in Structured Domains. Advances in Cognitive Systems, 4, 131-150.

Bibtex entry:

```
@article{trestle:2016a,
author={MacLellan, C.J. and Harpstead, E. and Aleven, V. and Koedinger, K.R.},
title={TRESTLE: A Model of Concept Formation in Structured Domains},
journal={Advances in Cognitive Systems},
volume={4},
year={2016}
}
```

Contents:

Instance Representation

The primary classes in the concept_formation package (*CobwebTree*, *Cobweb3Tree*, and *TrestleTree*) learn from instances that are represented as python dictionaries (i.e., lists of attribute values). This representation is different from the feature vector representation that is used by most machine learning packages (e.g., ScikitLearn). The concept_formation package uses the dictionary format instead of feature vectors for two reasons: dictionaries are more human readable and dictionaries offer more flexibility in the kinds of data that can be represented (e.g., attributes in dictionaries can have other dictionaries as values). Furthermore, it is a general format that many other representations, such as JSON, can be easily converted into. In fact, the concept_formation package has methods for facilitating such conversions.

Attributes

The concept_formation package supports four kinds of attributes:

Constant Attributes The default attribute type. Constant attributes are typically strings but any attribute that does not satisfy the conditions for the other categories will be assumed to be constant.

Variable Attributes Any attribute that can be renamed to maximize mapping between an instance and a concept. This allows for matching attributes based on the similarity of their values rather than strictly on their attribute names. Variable are denoted with a question mark `'?'` as their first element (e.g., `'?variable-attribute'`).

Relational Attributes An attribute that represents a relationship between other attributes or values of the instance. Relation attributes are represented as tuples (e.g., `('relation', 'obj1', 'obj2')`). Relations can only be in the top level of the instance (i.e., component values, described below, cannot contain relations). If a relationship needs to be expressed between attributes of component values, then preorder unary relations can be used. For example, to express a relationship of feature1 of subobject1 I might have: `('relation', ('feature1', 'subobject1'))`.

Hidden Attributes Attributes that are maintained in the concept knowledge base but are not considered during concept formation. These are useful for propagating unique ids or other bookkeeping labels into the knoweldge base without biasing concept formation. Hidden attributes are denoted as constant or relational attributes that have an `'_'` as their first element (i.e., `attribute[0] == '_'`). For constants, this means that the first character is an underscore (e.g., `"_hidden"`). For relations, this means that the first element in the tuple is an string underscore (e.g., `('_', 'hidden-relation', 'obj')`).

Only the **constant** and **hidden** attributes are supported by *CobwebTree* and *Cobweb3Tree*. *TrestleTree* supports all attribute types.

In general attribute names must be hashable (so they can be used in a dictionary and must be zero index-able (e.g., `attribute[0]`, so that they can be tested to determine if they are hidden.

Values

For each of these attribute type, the concept_formation package supports three kinds of values:

Nominal Values All non-numerical values (typically strings or booleans).

Numerical Values All values that are recognized by Python as numbers (i.e., `isinstance(val, Number)`).

Component Values All dictionary values (i.e., sub-instances). All component values are internally converted into unary relations, so unary relations can also be used directly. For example `{'subobject': {'attr': 'value'}}` is equivalent to `{('attr', 'subobject'): 'value'}`. Note that sub-instances cannot contain relations. Instead include the relations in the top-level instance and use unary relations to refer to elements of sub-instances (e.g., `('relation1' ('att1', 'subobject'))`).

The *CobwebTree* class supports only **nominal** values. The *Cobweb3Tree* supports both **nominal** and **numeric** values. Finally, the *TrestleTree* supports all value types.

Example Instance

Here is an instance that provides an example of each of these different attribute-value type combinations:

```
# Data is stored in a list of dictionaries where values can be either nominal,
# numeric, hidden, component, unbound attributes, or relational.
In [1]: instance = {'f1': 'v1', # constant attribute with nominal value
   ...:             'f2': 2.6, # constant attribute with numerical value
   ...:             'f3': {'sub-feature1': 'v1'}, # constant attribute with component␣
↪value
   ...:             '?f4': 'v1', # variable attribute with nominal value
   ...:             '?f5': 2.6, # variable attribute with numerical value
   ...:             '?f6': {'sub-feature1': 'v1'}, # variable attribute with␣
↪component value
```

```
   ...:              ('some-relation', 'f3', '?f4'): True, #relation attribute with␣
↪nominal value
   ...:              ('some-relation2', 'f3', '?f4'): 2.6, #relation attribute with␣
↪numeric value
   ...:              ('some-relation3', 'f3', '?f4'): {'sub-feature1': 'v1'},
↪#relation attribute with component value
   ...:              ('some-relation4', 'f3', ('sub-feature1', '?f4')): True, #␣
↪relation attribute that uses unary relation to access sub-feature1 of ?f4. It also␣
↪has a nominal value.
   ...:              '_f7': 'v1', # hidden attribute with nominal value
   ...:              '_f8': 2.6, # hidden attribute with numeric value
   ...:              '_f9': {'sub-feature1': 'v1'}, # hidden attribute with component␣
↪value
   ...:             }
   ...:
```

Examples

Fast Example

```
In [1]: from pprint import pprint

In [2]: from concept_formation.trestle import TrestleTree

In [3]: from concept_formation.cluster import cluster

# Data is stored in a list of dictionaries where values can be either nominal,
# numeric, component.
In [4]: data = [{'f1': 'v1', #nominal value
   ...:           'f2': 2.6, #numeric value
   ...:           '?f3': {'sub-feature1': 'v1'}, # component value
   ...:           '?f4': {'sub-feature1': 'v1'}, # component value
   ...:           ('some-relation','?f3','?f4'): True #relational attribute
   ...:          },
   ...:          {'f1': 'v1', #nominal value
   ...:           'f2': 2.8, #numeric value
   ...:           '?f3': {'sub-feature1': 'v2'}, # component value
   ...:           '?f4': {'sub-feature1': 'v1'}, # component value
   ...:           ('some-relation','?f3','?f4'): True #relational attribute
   ...:          }]
   ...:

# Data can be clustered with a TrestleTree, which supports all data types or
# with a specific tree (CobwebTree or Cobweb3Tree) that supports subsets of
# datatypes (CobwebTree supports only Nominal and Cobweb3Tree supports only
# nominal or numeric).
In [5]: tree = TrestleTree()

In [6]: tree.fit(data)

# Trees can be printed in plaintext or exported in JSON format
In [7]: print(tree)
|-{"('some-relation', u'?o2', u'?o1')": {'True': 1}, 'f1': {'v1': 2}, 'f2': {'
↪#ContinuousValue#': 2.7000 (0.1772) [2]}, "('sub-feature1', u'?o2')": {'v1': 1, 'v2
↪': 1}, "('some-relation', u'?o1', u'?o2')": {'True': 1}, "('sub-feature1', u'?o1')
↪': {'v1': 2}}: 2.0
```

```
       |-{'f1': {'v1': 1}, 'f2': {'#ContinuousValue#': 2.6000 (0.0000) [1]}, '('some-
↪relation', u'?o1', u'?o2')': {'True': 1}, '('sub-feature1', u'?o2')': {'v1': 1}, '(
↪'sub-feature1', u'?o1')': {'v1': 1}}: 1.0
       |-{'('some-relation', u'?o2', u'?o1')': {'True': 1}, 'f1': {'v1': 1}, 'f2': {'
↪#ContinuousValue#': 2.8000 (0.0000) [1]}, '('sub-feature1', u'?o2')': {'v2': 1}, '(
↪'sub-feature1', u'?o1')': {'v1': 1}}: 1.0


In [8]: pprint(tree.root.output_json())
{u'children': [{u'children': [],
                u'counts': {"('some-relation', u'?o1', u'?o2')": {'True': 1},
                            "('sub-feature1', u'?o1')": {'v1': 1},
                            "('sub-feature1', u'?o2')": {'v1': 1},
                            'f1': {'v1': 1},
                            'f2': {u'#ContinuousValue#': {u'mean': 2.6,
                                                          u'n': 1.0,
                                                          u'std': 0.0}}},
                u'name': u'Concept1',
                u'size': 1.0},
               {u'children': [],
                u'counts': {"('some-relation', u'?o2', u'?o1')": {'True': 1},
                            "('sub-feature1', u'?o1')": {'v1': 1},
                            "('sub-feature1', u'?o2')": {'v2': 1},
                            'f1': {'v1': 1},
                            'f2': {u'#ContinuousValue#': {u'mean': 2.8,
                                                          u'n': 1.0,
                                                          u'std': 0.0}}},
                u'name': u'Concept10',
                u'size': 1.0}],
 u'counts': {"('some-relation', u'?o1', u'?o2')": {'True': 1},
             "('some-relation', u'?o2', u'?o1')": {'True': 1},
             "('sub-feature1', u'?o1')": {'v1': 2},
             "('sub-feature1', u'?o2')": {'v1': 1, 'v2': 1},
             'f1': {'v1': 2},
             'f2': {u'#ContinuousValue#': {u'mean': 2.7,
                                           u'n': 2.0,
                                           u'std': 0.17724538509055118}}},
 u'name': u'Concept9',
 u'size': 2.0}

# Trees can also be used to infer missing attributes of new data points.
In [9]: new = {'f2': 2.6, '?f3': {'sub-feature1': 'v1'},
   ...:        '?f4': {'sub-feature1': 'v1'}}
   ...:

# Here we see that 'f1' and 'some-relation' are infered.
In [10]: pprint(tree.infer_missing(new))
{'?f3': {'sub-feature1': 'v1'},
 '?f4': {'sub-feature1': 'v1'},
 'f1': 'v1',
 'f2': 2.6,
 ('some-relation', '?f3', '?f4'): True}

# They can also be used to predict specific attribute values
In [11]: concept = tree.categorize(new)

In [12]: print(concept.predict('f1'))
v1
```

```
# Or to get the probability of a particular attribute value
In [13]: print(concept.probability('f1', 'v1'))
1.0

# Trees can also be used to produce flat clusterings
In [14]: new_tree = TrestleTree()

In [15]: clustering = cluster(new_tree, data)

In [16]: print(clustering)
[[u'Concept31', u'Concept40']]
```

Cobweb:

Clustering Mushrooms

```
from __future__ import print_function
from __future__ import unicode_literals
from __future__ import absolute_import
from __future__ import division
from random import shuffle
from random import seed

import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.feature_extraction import DictVectorizer
from sklearn.metrics import adjusted_rand_score

from concept_formation.cobweb import CobwebTree
from concept_formation.cluster import cluster
from concept_formation.datasets import load_mushroom

seed(0)
mushrooms = load_mushroom()
shuffle(mushrooms)
mushrooms = mushrooms[:150]

tree = CobwebTree()
mushrooms_no_class = [{a: mushroom[a] for a in mushroom
                       if a != 'classification'} for mushroom in mushrooms]
clusters = cluster(tree, mushrooms_no_class)[0]
mushroom_class = [mushroom[a] for mushroom in mushrooms for a in mushroom
                  if a == 'classification']
ari = adjusted_rand_score(clusters, mushroom_class)

dv = DictVectorizer(sparse=False)
mushroom_X = dv.fit_transform(mushrooms_no_class)

pca = PCA(n_components=2)
mushroom_2d_x = pca.fit_transform(mushroom_X)

colors = ['b', 'g', 'r', 'y', 'k', 'c', 'm']
clust_set = {v:i for i,v in enumerate(list(set(clusters)))}
class_set = {v:i for i,v in enumerate(list(set(mushroom_class)))}
```

```
for class_idx, class_label in enumerate(class_set):
    x = [v[0] for i,v in enumerate(mushroom_2d_x) if mushroom_class[i] == class_label]
    y = [v[1] for i,v in enumerate(mushroom_2d_x) if mushroom_class[i] == class_label]
    c = [colors[clust_set[clusters[i]]] for i,v in enumerate(mushroom_2d_x) if
         mushroom_class[i] == class_label]
    plt.scatter(x, y, color=c, marker=r"$ {} $".format(class_label[0]), label=class_
↪label)

plt.title("COBWEB Mushroom Clustering (ARI w/ Hidden Edibility Labels = %0.2f)" %␣
↪(ari))
plt.xlabel("PCA Dimension 1")
plt.ylabel("PCA Dimension 2")
plt.legend(loc=4)
plt.show()
```

Mushroom Edibility Prediction

```
from __future__ import print_function
from __future__ import unicode_literals
from __future__ import absolute_import
from __future__ import division
import matplotlib.pyplot as plt
import numpy as np
from random import seed

from concept_formation.examples.examples_utils import avg_lines
from concept_formation.evaluation import incremental_evaluation
from concept_formation.cobweb import CobwebTree
from concept_formation.dummy import DummyTree
from concept_formation.datasets import load_mushroom

seed(0)
num_runs = 30
num_examples = 30
mushrooms = load_mushroom()

naive_data = incremental_evaluation(DummyTree(), mushrooms,
                                    run_length=num_examples,
                                    runs=num_runs, attr="classification")
cobweb_data = incremental_evaluation(CobwebTree(), mushrooms,
                                    run_length=num_examples,
                                    runs=num_runs, attr="classification")

cobweb_x, cobweb_y = [], []
naive_x, naive_y = [], []

for opp in range(len(cobweb_data[0])):
  for run in range(len(cobweb_data)):
    cobweb_x.append(opp)
    cobweb_y.append(cobweb_data[run][opp])

for opp in range(len(naive_data[0])):
  for run in range(len(naive_data)):
    naive_x.append(opp)
    naive_y.append(naive_data[run][opp])
```

```
cobweb_x = np.array(cobweb_x)
cobweb_y = np.array(cobweb_y)
naive_x = np.array(naive_x)
naive_y = np.array(naive_y)

cobweb_y_smooth, cobweb_lower_smooth, cobweb_upper_smooth = avg_lines(cobweb_x,␣
↪cobweb_y)
naive_y_smooth, naive_lower_smooth, naive_upper_smooth = avg_lines(naive_x, naive_y)

plt.fill_between(cobweb_x, cobweb_lower_smooth, cobweb_upper_smooth, alpha=0.5,
                 facecolor="green")
plt.fill_between(naive_x, naive_lower_smooth, naive_upper_smooth, alpha=0.5,
                 facecolor="red")

plt.plot(cobweb_x, cobweb_y_smooth, label="COBWEB", color="green")
plt.plot(naive_x, naive_y_smooth, label="Naive Predictor", color="red")

plt.gca().set_ylim([0.0,1.0])
plt.gca().set_xlim([0,max(naive_x)-1])
plt.title("Incremental Mushroom Edibility Prediction Accuracy")
plt.xlabel("# of Training Examples")
plt.ylabel("Avg. Probability of True Class (Accuracy)")
plt.legend(loc=4)

plt.show()
```

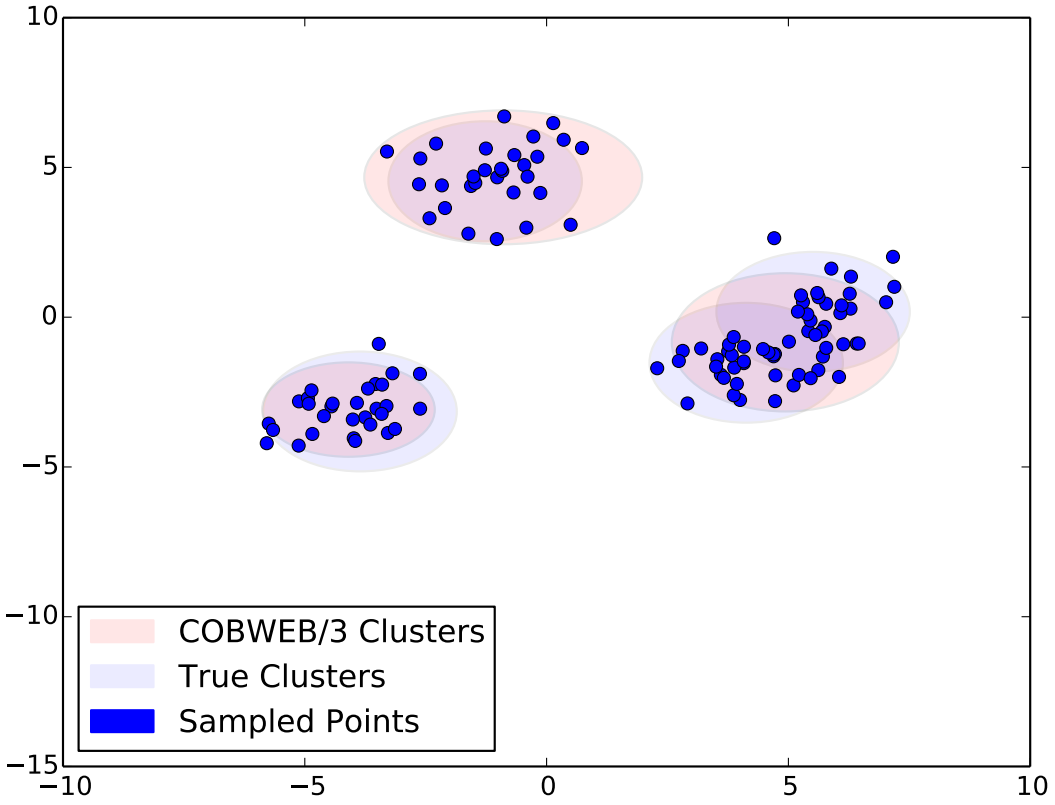
Cobweb/3:

Clustering Simulated 2D Data

```
from __future__ import print_function
from __future__ import unicode_literals
from __future__ import absolute_import
from __future__ import division
from random import normalvariate
from random import shuffle
from random import uniform
from random import seed
#import time

import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from matplotlib.patches import Ellipse

from concept_formation.cobweb3 import Cobweb3Tree
from concept_formation.cobweb3 import cv_key as cv

seed(0)

num_clusters = 4
num_samples = 30
sigma = 1

xmean = [uniform(-8, 8) for i in range(num_clusters)]
ymean = [uniform(-8, 8) for i in range(num_clusters)]
label = ['bo', 'bo', 'bo', 'bo', 'bo', 'bo', 'bo']
```

```
shuffle(label)
label = label[0:num_clusters]

data = []
actual = []
clusters = []
for i in range(num_clusters):
    data += [{'x': normalvariate(xmean[i], sigma), 'y':
              normalvariate(ymean[i], sigma), '_label': label[i]} for j in
             range(num_samples)]
    actual.append(Ellipse([xmean[i], ymean[i]], width=4*sigma,
                          height=4*sigma, angle=0))
shuffle(data)
trained = []

#plt.ion()

#plt.show()

tree = Cobweb3Tree()

# draw the actual sampling distribution
for c in actual:
    c.set_alpha(0.08)
    c.set_facecolor("blue")
    plt.gca().add_patch(c)

for datum in data:
    #train the tree on the sampled datum
    tree.ifit(datum)
    trained.append(datum)

    # remove old cluster circles
    for c in clusters:
        c.remove()

    # 4 * std gives two std on each side (~95% confidence)
    clusters = [Ellipse([cluster.av_counts['x'][cv].unbiased_mean(),
                         cluster.av_counts['y'][cv].unbiased_mean()],
                        width=4*cluster.av_counts['x'][cv].unbiased_std(),
                        height=4*cluster.av_counts['y'][cv].unbiased_std(),
                        angle=0) for cluster in tree.root.children]

    # draw the cluster circles
    for c in clusters:
        c.set_alpha(0.1)
        c.set_facecolor('red')
        plt.gca().add_patch(c)

    # draw the new point
    plt.plot([datum['x']], [datum['y']], datum['_label'])


    #plt.draw()
    #time.sleep(0.0001)

plt.axis([-10, 10, -15, 10])
red_patch = mpatches.Patch(color='red', alpha=0.1)
```

```
blue_patch = mpatches.Patch(color='blue', alpha=0.08)
samples_patch = mpatches.Patch(color='blue')
plt.legend([red_patch, blue_patch, samples_patch], ['COBWEB/3 Clusters',
                                                    'True Clusters',
                                                    'Sampled Points'], loc=3)

#plt.ioff()
plt.show()
```



Clustering Irises

```
from __future__ import print_function
from __future__ import unicode_literals
from __future__ import absolute_import
from __future__ import division
from random import shuffle
from random import seed

import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.feature_extraction import DictVectorizer
from sklearn.metrics import adjusted_rand_score
```

```
from concept_formation.cobweb3 import Cobweb3Tree
from concept_formation.cluster import cluster
from concept_formation.datasets import load_iris

seed(0)
irises = load_iris()
shuffle(irises)

tree = Cobweb3Tree()
irises_no_class = [{a: iris[a] for a in iris if a != 'class'} for iris in irises]
clusters = cluster(tree, irises_no_class)[0]
iris_class = [iris[a] for iris in irises for a in iris if a == 'class']
ari = adjusted_rand_score(clusters, iris_class)

dv = DictVectorizer(sparse=False)
iris_X = dv.fit_transform([{a:iris[a] for a in iris if a != 'class'} for iris in␣
↪irises])
pca = PCA(n_components=2)
iris_2d_x = pca.fit_transform(iris_X)

colors = ['b', 'g', 'r', 'y', 'k', 'c', 'm']
shapes = ['o', '^', '+']
clust_set = {v:i for i,v in enumerate(list(set(clusters)))}
class_set = {v:i for i,v in enumerate(list(set(iris_class)))}

for class_idx, class_label in enumerate(class_set):
    x = [v[0] for i,v in enumerate(iris_2d_x) if iris_class[i] == class_label]
    y = [v[1] for i,v in enumerate(iris_2d_x) if iris_class[i] == class_label]
    c = [colors[clust_set[clusters[i]]] for i,v in enumerate(iris_2d_x) if
         iris_class[i] == class_label]
    plt.scatter(x, y, color=c, marker=shapes[class_idx], label=class_label)

plt.title("COBWEB/3 Iris Clustering (ARI = %0.2f)" % (ari))
plt.xlabel("PCA Dimension 1")
plt.ylabel("PCA Dimension 2")
plt.legend(loc=4)
plt.show()
```

Predicting Iris Classification

```
from __future__ import print_function
from __future__ import unicode_literals
from __future__ import absolute_import
from __future__ import division

import matplotlib.pyplot as plt
import numpy as np
from random import seed

from concept_formation.examples.examples_utils import avg_lines
from concept_formation.evaluation import incremental_evaluation
from concept_formation.cobweb3 import Cobweb3Tree
from concept_formation.dummy import DummyTree
from concept_formation.datasets import load_iris

seed(0)
```

```
num_runs = 30
num_examples = 20
irises = load_iris()

naive_data = incremental_evaluation(DummyTree(), irises,
                                    run_length=num_examples,
                                    runs=num_runs, attr="class")
cobweb_data = incremental_evaluation(Cobweb3Tree(), irises,
                                    run_length=num_examples,
                                    runs=num_runs, attr="class")
cobweb_x, cobweb_y = [], []
naive_x, naive_y = [], []

for opp in range(len(cobweb_data[0])):
  for run in range(len(cobweb_data)):
    cobweb_x.append(opp)
    cobweb_y.append(cobweb_data[run][opp])

for opp in range(len(naive_data[0])):
  for run in range(len(naive_data)):
    naive_x.append(opp)
    naive_y.append(naive_data[run][opp])

cobweb_x = np.array(cobweb_x)
cobweb_y = np.array(cobweb_y)
naive_x = np.array(naive_x)
naive_y = np.array(naive_y)

cobweb_y_smooth, cobweb_lower_smooth, cobweb_upper_smooth = avg_lines(cobweb_x,␣
↪cobweb_y)
naive_y_smooth, naive_lower_smooth, naive_upper_smooth = avg_lines(naive_x, naive_y)

plt.fill_between(cobweb_x, cobweb_lower_smooth, cobweb_upper_smooth, alpha=0.5,
                 facecolor="green")
plt.fill_between(naive_x, naive_lower_smooth, naive_upper_smooth, alpha=0.5,
                 facecolor="red")

plt.plot(cobweb_x, cobweb_y_smooth, label="COBWEB/3", color="green")
plt.plot(naive_x, naive_y_smooth, label="Naive Predictor", color="red")

plt.gca().set_ylim([0.00,1.0])
plt.gca().set_xlim([0,max(naive_x)-1])
plt.title("Incremental Iris Classification Prediction Accuracy")
plt.xlabel("# of Training Examples")
plt.ylabel("Avg. Probability of True Class (Accuracy)")
plt.legend(loc=4)

plt.show()
```

Predicting with Noisy Nominal and Numeric Features

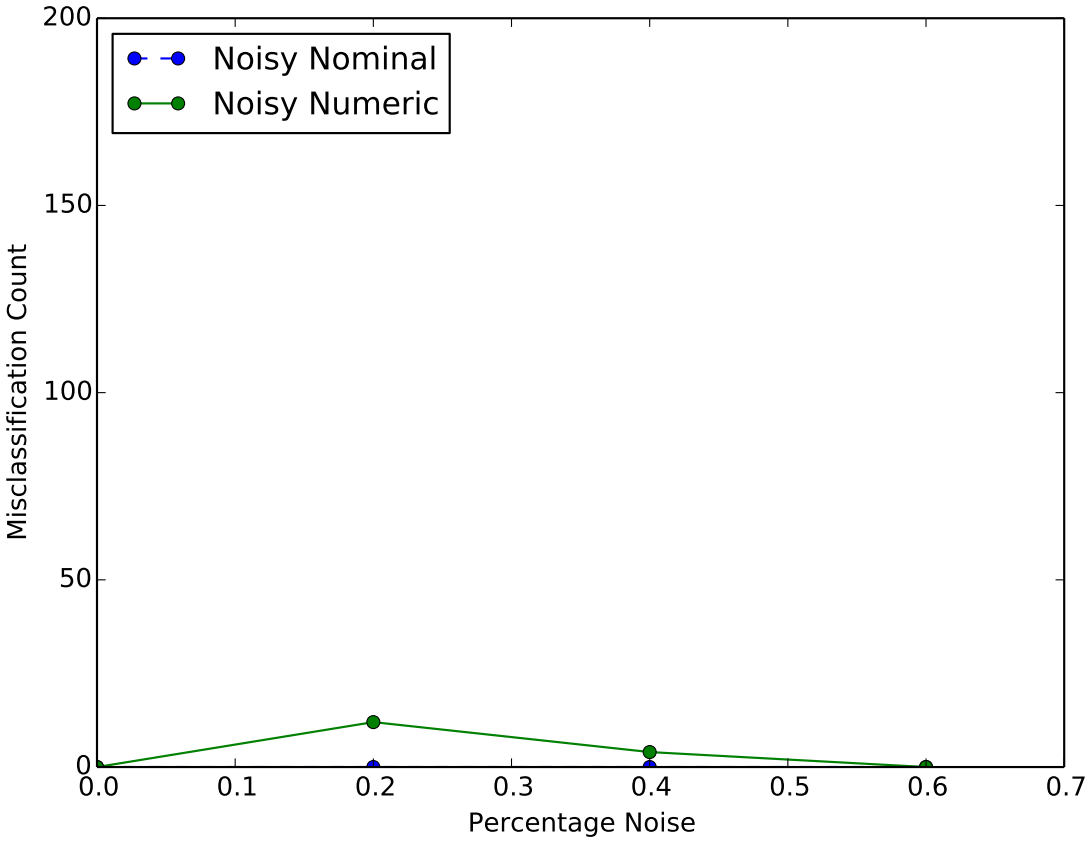
```
import numpy as np
import matplotlib.pyplot as plt
from random import choice
from random import shuffle
from random import random
```

```
from random import seed

from concept_formation.cobweb3 import Cobweb3Tree
from concept_formation.cluster import cluster

seed(0)

def run_clust_exp(nominal_noise=0, numeric_noise=0, scaling=False):
    data = []

    for i in range(60):
        x = {}
        x['_label'] = "G1"

        if random() >= nominal_noise:
            x['f1'] = "G1f1"
        else:
            x['f1'] = choice(['G2f1', 'G3f1'])

        if random() >= nominal_noise:
            x['f2'] = choice(["G1f2a", "G1f2b"])
        else:
            x['f2'] = choice(["G2f2a", "G2f2b","G3f2a", "G3f2b"])

        if random() >= numeric_noise:
            x['f3'] = np.random.normal(4,1,1)[0]
        else:
            x['f3'] = choice([np.random.normal(10,1,1)[0],np.random.normal(16,1,
↪1)[0]])

        if random() >= numeric_noise:
            x['f4'] = np.random.normal(20,2,1)[0]
        else:
            x['f4'] = choice([np.random.normal(32,2,1)[0],np.random.normal(44,2,
↪1)[0]])

        data.append(x)

    for i in range(60):
        x = {}
        x['_label'] = "G2"

        if random() >= nominal_noise:
            x['f1'] = "G2f1"
        else:
            x['f1'] = choice(["G2f1", "G3f1"])

        if random() >= nominal_noise:
            x['f2'] = choice(["G2f2a", "G2f2b"])
        else:
            x['f2'] = choice(["G1f2a", "G1f2b", "G3f2a", "G3f2b"])

        if random() >= numeric_noise:
            x['f3'] = np.random.normal(10,1,1)[0]
        else:
            x['f3'] = choice([np.random.normal(4,1,1)[0],np.random.normal(16,1,1)[0]])

        if random() >= numeric_noise:
```

```
            x['f4'] = np.random.normal(32,2,1)[0]
        else:
            x['f4'] = choice([np.random.normal(20,2,1)[0],np.random.normal(44,2,
↪1)[0]])

        data.append(x)

    for i in range(60):
        x = {}
        x['_label'] = "G3"

        if random() >= nominal_noise:
            x['f1'] = "G3f1"
        else:
            x['f1'] = choice(["G1f1", "G2f1"])

        if random() >= nominal_noise:
            x['f2'] = choice(["G3f2a", "G3f2b"])
        else:
            x['f2'] = choice(["G1f2a", "G1f2b", "G2f2a", "G2f2b"])

        if random() >= numeric_noise:
            x['f3'] = np.random.normal(16,1,1)[0]
        else:
            x['f3'] = choice([np.random.normal(4,1,1)[0],np.random.normal(10,1,1)[0]])

        if random() >= numeric_noise:
            x['f4'] = np.random.normal(44,2,1)[0]
        else:
            x['f4'] = choice([np.random.normal(20,2,1)[0],np.random.normal(32,2,
↪1)[0]])

        data.append(x)

    shuffle(data)
    t = Cobweb3Tree(scaling=scaling)
    clustering = cluster(t, data)
    return data, clustering[0]

def run_noise_exp(scaling=False):
    noise = np.arange(0.0, 0.8, 0.2)
    print(noise)

    miss_nominal = []
    miss_numeric = []
    #aris = []

    for n in noise:
        data, clustering = run_clust_exp(n,0,scaling)
        confusion = {}
        for i,c in enumerate(clustering):
            if c not in confusion:
                confusion[c] = {}
            if data[i]['_label'] not in confusion[c]:
                confusion[c][data[i]['_label']] = 0
            confusion[c][data[i]['_label']] += 1
        print(confusion)
```

```
        totals = sorted([(sum([confusion[c][g] for g in
                               confusion[c]]), c) for c in confusion], reverse=True)
        top = [c for t,c in totals[:3]]
        miss = 0

        for c in confusion:
            v = sorted([confusion[c][g] for g in confusion[c]], reverse=True)
            if c not in top:
                miss += v[0]

            for minorities in v[1:]:
                miss += 2 * minorities

        #labels = [d['_label'] for d in data]
        #aris.append(ari(labels, clustering))
        miss_nominal.append(miss)

    for n in noise:
        data, clustering = run_clust_exp(0,n, scaling)
        confusion = {}
        for i,c in enumerate(clustering):
            if c not in confusion:
                confusion[c] = {}
            if data[i]['_label'] not in confusion[c]:
                confusion[c][data[i]['_label']] = 0
            confusion[c][data[i]['_label']] += 1
        print(confusion)

        totals = sorted([(sum([confusion[c][g] for g in
                               confusion[c]]), c) for c in confusion], reverse=True)
        top = [c for t,c in totals[:3]]
        miss = 0

        for c in confusion:
            v = sorted([confusion[c][g] for g in confusion[c]], reverse=True)
            if c not in top:
                miss += v[0]

            for minorities in v[1:]:
                miss += 2 * minorities

        # labels = [d['_label'] for d in data]
        # aris.append(ari(labels, clustering))
        miss_numeric.append(miss)

    return noise, miss_nominal, miss_numeric

nominal = []
numeric = []

for i in range(2):
    noise, miss_nominal, miss_numeric = run_noise_exp(scaling=0.5)
    nominal.append(miss_nominal)
    numeric.append(miss_numeric)
    noise = noise

nominal = np.array(nominal)
numeric = np.array(numeric)
```

```
nominal = np.mean(nominal, axis=0)
numeric = np.mean(numeric, axis=0)

nominal_line, = plt.plot(noise, miss_nominal, linestyle="--", marker="o", color="b")
numeric_line, = plt.plot(noise, miss_numeric, linestyle="-", marker="o", color="g")
plt.legend([nominal_line, numeric_line], ["Noisy Nominal", "Noisy Numeric"],
           loc=2)

plt.xlabel("Percentage Noise")
plt.ylabel("Misclassification Count")


plt.ylim(0,200)
plt.show()
```



Cobweb3 Regression Example

```
from __future__ import print_function
from __future__ import unicode_literals
from __future__ import absolute_import
from __future__ import division
```

```
import numpy as np
from sklearn.tree import DecisionTreeRegressor
from concept_formation.trestle import TrestleTree
import matplotlib.pyplot as plt
from random import seed

# Create a random dataset
rng = np.random.RandomState(1)
seed(0)
X = np.sort(5 * rng.rand(80, 1), axis=0)
y = np.sin(X).ravel()
y[::5] += 3 * (0.5 - rng.rand(16))

# Fit regression models (Decision Tree and TRESTLE)
# For TRESTLE the y attribute is hidden, so only the X is used to make
# predictions.
dtree = DecisionTreeRegressor(max_depth=3)
dtree.fit(X, y)
ttree = TrestleTree()
training_data = [{'x': float(X[i][0]), '_y': float(y[i])} for i,v in
                 enumerate(X)]
ttree.fit(training_data, iterations=1)

# Predict
X_test = np.arange(0.0, 5.0, 0.01)[:, np.newaxis]
y_dtree = dtree.predict(X_test)
y_trestle = [ttree.categorize({'x': float(v)}).predict('_y') for v in X_test]

# Plot the results
plt.figure()
plt.scatter(X, y, c="k", label="Data")
plt.plot(X_test, y_trestle, c="g", label="TRESTLE", linewidth=2)
plt.plot(X_test, y_dtree, c="r", label="Decison Tree (Depth=3)", linewidth=2)
plt.xlabel("Data")
plt.ylabel("Target")
plt.title("TRESTLE/Decision Tree Regression")
plt.legend(loc=3)
plt.show()
```

Trestle:

Clustering RumbleBlocks Towers

```
from __future__ import print_function
from __future__ import unicode_literals
from __future__ import absolute_import
from __future__ import division
from random import shuffle
from random import seed

from sklearn.metrics import adjusted_rand_score
import matplotlib.pyplot as plt

from concept_formation.trestle import TrestleTree
from concept_formation.cluster import cluster
from concept_formation.datasets import load_rb_wb_03
```

```
from concept_formation.preprocessor import ObjectVariablizer

seed(0)

towers = load_rb_wb_03()
shuffle(towers)
towers = towers[:60]

variablizer = ObjectVariablizer()
towers = [variablizer.transform(t) for t in towers]

tree = TrestleTree()
clusters = cluster(tree, towers, maxsplit=10)
human_labels = [tower['_human_cluster_label'] for tower in towers]

x = [num_splits for num_splits in range(1,len(clusters)+1)]
y = [adjusted_rand_score(human_labels, split) for split in clusters]
plt.plot(x, y, label="TRESTLE")

plt.title("TRESTLE Clustering Accuracy (Given Human Ground Truth)")
plt.ylabel("Adjusted Rand Index (Agreement Correcting for Chance)")
plt.xlabel("# of Splits of Trestle Tree")
plt.legend(loc=4)
plt.show()
```

Cluster Split Search

```
from __future__ import print_function
from __future__ import unicode_literals
from __future__ import absolute_import
from __future__ import division
from random import shuffle
from random import seed

from sklearn.metrics import adjusted_rand_score
import matplotlib.pyplot as plt
import numpy as np

from concept_formation.trestle import TrestleTree
from concept_formation.cluster import cluster_split_search
from concept_formation.cluster import AIC, BIC, AICc, CU
from concept_formation.datasets import load_rb_wb_03
from concept_formation.datasets import load_rb_com_11
from concept_formation.datasets import load_rb_s_13
from concept_formation.preprocessor import ObjectVariablizer

seed(5)

hueristics = [AIC,BIC,CU,AICc]

def calculate_aris(dataset):
    shuffle(dataset)
    dataset = dataset[:60]

    variablizer = ObjectVariablizer()
    dataset = [variablizer.transform(t) for t in dataset]
```

```
    tree = TrestleTree()
    tree.fit(dataset)

    clusters = [cluster_split_search(tree,dataset,h,minsplit=1,maxsplit=40,mod=False)␣
↪for h in hueristics]
    human_labels = [dataset['_human_cluster_label'] for dataset in dataset]

    return [max(adjusted_rand_score(human_labels, huer),0.01) for huer in clusters]

x = np.arange(len(hueristics))
width = 0.3

hueristic_names = ['AIC','BIC','CU','AICc']
# for i in range(len(clusters)):
#     hueristic_names[i] +=  '\nClusters='+str(len(set(clusters[i])))

b1 = plt.bar(x-width,calculate_aris(load_rb_wb_03()),width,color='r',alpha=.8,align=
↪'center')
b2 = plt.bar(x,calculate_aris(load_rb_com_11()),width,color='b',alpha=.8,align='center
↪')
b3 = plt.bar(x+width,calculate_aris(load_rb_s_13()),width,color='g',alpha=.8,align=
↪'center')
plt.legend((b1[0],b2[0],b3[0]),('wb_03','com_11','s_13'))
plt.title("TRESTLE Clustering Accuracy of Best Clustering by Different Hueristics")
plt.ylabel("Adjusted Rand Index (Agreement Correcting for Chance)")
plt.ylim(0,1)
plt.xlabel("Hueristic")
plt.xticks(x,hueristic_names)
plt.show()
```

Predicting RumbleBlocks Tower Stability

```
from __future__ import print_function
from __future__ import unicode_literals
from __future__ import absolute_import
from __future__ import division
import matplotlib.pyplot as plt
from random import seed
import numpy as np

from concept_formation.examples.examples_utils import lowess
from concept_formation.evaluation import incremental_evaluation
from concept_formation.trestle import TrestleTree
from concept_formation.dummy import DummyTree
from concept_formation.datasets import load_rb_s_07
from concept_formation.datasets import load_rb_s_07_human_predictions
from concept_formation.preprocessor import ObjectVariablizer

seed(5)

num_runs = 30
num_examples = 29
towers = load_rb_s_07()

variablizer = ObjectVariablizer()
```

```
towers = [variablizer.transform(t) for t in towers]

naive_data = incremental_evaluation(DummyTree(), towers,
                                    run_length=num_examples,
                                    runs=num_runs, attr="success")
cobweb_data = incremental_evaluation(TrestleTree(), towers,
                                     run_length=num_examples,
                                     runs=num_runs, attr="success")

human_data = []
key = None
human_predictions = load_rb_s_07_human_predictions()
for line in human_predictions:
    line = line.rstrip().split(",")
    if key is None:
        key = {v: i for i, v in enumerate(line)}
        continue
    x = int(line[key['order']])-1
    y = (1 - abs(int(line[key['correctness']]) -
                 int(line[key['prediction']])))
    human_data.append((x, y))

human_data.sort()

cobweb_x, cobweb_y = [], []
naive_x, naive_y = [], []
human_x, human_y = [], []

for opp in range(len(cobweb_data[0])):
    for run in range(len(cobweb_data)):
        cobweb_x.append(opp)
        cobweb_y.append(cobweb_data[run][opp])

for opp in range(len(naive_data[0])):
    for run in range(len(naive_data)):
        naive_x.append(opp)
        naive_y.append(naive_data[run][opp])

for x, y in human_data:
    human_x.append(x)
    human_y.append(y)

cobweb_x = np.array(cobweb_x)
cobweb_y = np.array(cobweb_y)
naive_x = np.array(naive_x)
naive_y = np.array(naive_y)
human_x = np.array(human_x)
human_y = np.array(human_y)

cobweb_y_smooth, cobweb_lower_smooth, cobweb_upper_smooth = lowess(cobweb_x,
                                                                   cobweb_y)
naive_y_smooth, naive_lower_smooth, naive_upper_smooth = lowess(naive_x,
                                                                naive_y)
human_y_smooth, human_lower_smooth, human_upper_smooth = lowess(human_x,
                                                                human_y)

plt.fill_between(cobweb_x, cobweb_lower_smooth, cobweb_upper_smooth, alpha=0.5,
                 facecolor="green")
```

```
plt.fill_between(naive_x, naive_lower_smooth, naive_upper_smooth, alpha=0.5,
                 facecolor="red")
plt.fill_between(human_x, human_lower_smooth, human_upper_smooth, alpha=0.3,
                 facecolor="blue")

plt.plot(cobweb_x, cobweb_y_smooth, label="TRESTLE", color="green")
plt.plot(naive_x, naive_y_smooth, label="Naive Predictor", color="red")
plt.plot(human_x, human_y_smooth, label="Human Predictions", color="blue")

plt.gca().set_ylim([0.00, 1.0])
plt.gca().set_xlim([0, max(naive_x)-1])
plt.title("Incremental Tower Success Prediction Accuracy")
plt.xlabel("# of Training Examples")
plt.ylabel("Avg. Probability of True Success Label (Accuracy)")
plt.legend(loc=4)

plt.show()
```

Predicting Quadruped Types

```
from __future__ import print_function
from __future__ import unicode_literals
from __future__ import absolute_import
from __future__ import division
import matplotlib.pyplot as plt
import numpy as np
from random import seed

from concept_formation.examples.examples_utils import avg_lines
from concept_formation.evaluation import incremental_evaluation
from concept_formation.trestle import TrestleTree
from concept_formation.dummy import DummyTree
from concept_formation.datasets import load_quadruped
from concept_formation.preprocessor import ObjectVariablizer

seed(0)

num_runs = 5
num_examples = 15
animals = load_quadruped(num_examples)

variablizer = ObjectVariablizer()
animals = [variablizer.transform(t) for t in animals]

for animal in animals:
    animal['type'] = animal['_type']
    del animal['_type']

naive_data = incremental_evaluation(DummyTree(), animals,
                                    run_length=num_examples,
                                    runs=num_runs, attr="type")
trestle_data = incremental_evaluation(TrestleTree(), animals,
                                    run_length=num_examples,
                                    runs=num_runs, attr="type")

trestle_x, trestle_y = [], []
```

```
naive_x, naive_y = [], []
human_x, human_y = [], []

for opp in range(len(trestle_data[0])):
  for run in range(len(trestle_data)):
    trestle_x.append(opp)
    trestle_y.append(trestle_data[run][opp])

for opp in range(len(naive_data[0])):
  for run in range(len(naive_data)):
    naive_x.append(opp)
    naive_y.append(naive_data[run][opp])

trestle_x = np.array(trestle_x)
trestle_y = np.array(trestle_y)
naive_x = np.array(naive_x)
naive_y = np.array(naive_y)

trestle_y_avg, _, _ = avg_lines(trestle_x, trestle_y)
naive_y_avg, _, _ = avg_lines(naive_x, naive_y)

plt.plot(trestle_x, trestle_y_avg, label="TRESTLE", color="green")
plt.plot(naive_x, naive_y_avg, label="Naive Predictor", color="red")

plt.gca().set_ylim([0.00,1.0])
plt.gca().set_xlim([0,max(naive_x)-1])
plt.title("Incremental Quadruped Prediction Accuracy")
plt.xlabel("# of Training Examples")
plt.ylabel("Avg. Probability of True Quadruped Type (Accuracy)")
plt.legend(loc=4)

plt.show()
```

concept_formation package

concept_formation.cobweb module

The Cobweb module contains the *CobwebTree* and *CobwebNode* classes which are used to achieve the basic Cobweb functionality.

CobwebTree

class concept_formation.cobweb.**CobwebTree**

The CobwebTree contains the knoweldge base of a partiucluar instance of the cobweb algorithm and can be used to fit and categorize instances.

categorize(*instance*)

Sort an instance in the categorization tree and return its resulting concept.

The instance is passed down the categorization tree according to the normal cobweb algorithm except using only the best operator and without modifying nodes' probability tables. **This process does not modify the tree's knowledge** for a modifying version of labeling use the *CobwebTree.ifit()* function

Parameters **instance** (*Instance*) – an instance to be categorized into the tree.

Returns A concept describing the instance

Return type *CobwebNode*

See also:

CobwebTree.cobweb()

clear()
Clears the concepts of the tree.

cobweb(*instance*)
The core cobweb algorithm used in fitting and categorization.

In the general case, the cobweb algorith entertains a number of sorting operations for the instance and then commits to the operation that maximizes the *category utility* of the tree at the current node and then recurses.

At each node the alogrithm first calculates the category utility of inserting the instance at each of the node's children, keeping the best two (see: *CobwebNode.two_best_children*), and then calculates the category_utility of performing other operations using the best two children (see: *CobwebNode.get_best_operation*), commiting to whichever operation results in the highest category utility. In the case of ties an operation is chosen at random.

In the base case, i.e. a leaf node, the algorithm checks to see if the current leaf is an exact match to the current node. If it is, then the instance is inserted and the leaf is returned. Otherwise, a new leaf is created.

Note: This function is equivalent to calling *CobwebTree.ifit()* but its better to call ifit because it is the polymorphic method siganture between the different cobweb family algorithms.

Parameters **instance** (*Instance*) – an instance to incorporate into the tree

Returns a concept describing the instance

Return type *CobwebNode*

See also:

CobwebTree.ifit(), *CobwebTree.categorize()*

fit(*instances*, *iterations=1*, *randomize_first=True*)
Fit a collection of instances into the tree.

This is a batch version of the ifit function that takes a collection of instances and categorizes all of them. The instances can be incorporated multiple times to burn in the tree with prior knowledge. Each iteration of fitting uses a randomized order but the first pass can be done in the original order of the list if desired, this is useful for initializing the tree with specific prior experience.

Parameters

- **instances** ([*Instance*, *Instance*, ...]) – a collection of instances
- **iterations** (*int*) – number of times the list of instances should be fit.
- **randomize_first** (*bool*) – whether or not the first iteration of fitting should be done in a random order or in the list's original order.

ifit(*instance*)
Incrementally fit a new instance into the tree and return its resulting concept.

The instance is passed down the cobweb tree and updates each node to incorporate the instance. **This process modifies the tree's knowledge** for a non-modifying version of labeling use the *CobwebTree.categorize()* function.

Parameters **instance** (*Instance*) – An instance to be categorized into the tree.

Returns A concept describing the instance

Return type *CobwebNode*

See also:

CobwebTree.cobweb()

infer_missing(*instance*, *choice_fn=u'most likely'*, *allow_none=True*)

Given a tree and an instance, returns a new instance with attribute values picked using the specified choice function (either "most likely" or "sampled").

Todo

write some kind of test for this.

Parameters

- **instance** (*Instance*) – an instance to be completed.
- **choice_fn** (*a string*) – a string specifying the choice function to use, either "most likely" or "sampled".
- **allow_none** (*Boolean*) – whether attributes not in the instance can be inferred to be missing. If False, then all attributes will be inferred with some value.

Returns A completed instance

Return type *Instance*

CobwebNode

class concept_formation.cobweb.**CobwebNode**(*otherNode=None*)

A CobwebNode represents a concept within the knoweldge base of a particular *CobwebTree*. Each node contains a probability table that can be used to calculate the probability of different attributes given the concept that the node represents.

In general the *CobwebTree.ifit()*, *CobwebTree.categorize()* functions should be used to initially interface with the Cobweb knowledge base and then the returned concept can be used to calculate probabilities of certain attributes or determine concept labels.

This constructor creates a CobwebNode with default values. It can also be used as a copy constructor to "deep-copy" a node, including all references to other parts of the original node's CobwebTree.

Parameters **otherNode** (CobwebNode) – Another concept node to deepcopy.

attrs(*attr_filter=None*)

Iterates over the attributes present in the node's attribute-value table with the option to filter certain types. By default the filter will ignore hidden attributes and yield all others. If the string 'all' is provided then all attributes will be yielded. In neither of those cases the filter will be interpreted as a function that returns true if an attribute should be yielded and false otherwise.

category_utility()
Return the category utility of a particular division of a concept into its children.

Category utility is always calculated in reference to a parent node and its own children. This is used as the heuristic to guide the concept formation process. Category Utility is calculated as:

$$CU(\{C_1, C_2, \cdots, C_n\}) = \frac{1}{n}\sum_{k=1}^{n} P(C_k)\left[\sum_i \sum_j P(A_i = V_{ij}|C_k)^2\right] - \sum_i \sum_j P(A_i = V_{ij})^2$$

where n is the numer of children concepts to the current node, $P(C_k)$ is the probability of a concept given the current node, $P(A_i = V_{ij}|C_k)$ is the probability of a particular attribute value given the concept C_k, and $P(A_i = V_{ij})$ is the probability of a particular attribute value given the current node.

In general this is used as an internal function of the cobweb algorithm but there may be times when it would be useful to call outside of the algorithm itself.

Returns The category utility of the current node with respect to its children.

Return type float

compute_relative_CU_const(*instance*)
Computes the constant value that is used to convert between CU and relative CU scores. The constant value is basically the category utility that results from adding the instance to the root, but none of the children. It can be computed directly as:

$$const = \frac{1}{n}\sum_{k=1}^{n}\left[\frac{C_k.count}{count+1}\sum_i \sum_j P(A_i = V_{ij}|C)^2\right] - \sum_i \sum_j P(A_i = V_{ij}|UpdatedRoot)^2$$

where n is the number of children of the root, C_k is child k, $C_k.count$ is the number of instances stored in child C_k, $count$ is the number of instances stored in the root. Finally, $UpdatedRoot$ is a copy of the root that has been updated with the counts of the instance.

Parameters **instance** (*Instance*) – The instance currently being categorized

Returns The value of the constant used to relativize the CU.

Return type float

create_child_with_current_counts()
Create a new child (to the current node) with the counts initialized by the *current node's counts*.

This operation is used in the speical case of a fringe split when a new node is created at a leaf.

Returns The new child

Return type *CobwebNode*

create_new_child(*instance*)
Create a new child (to the current node) with the counts initialized by the *given instance*.

This is the operation used for creating a new child to a node and adding the instance to it.

Parameters **instance** (*Instance*) – The instance currently being categorized

Returns The new child

Return type *CobwebNode*

cu_for_fringe_split(*instance*)
Return the category utility of performing a fringe split (i.e., adding a leaf to a leaf).

A "fringe split" is essentially a new operation performed at a leaf. It is necessary to have the distinction because unlike a normal split a fringe split must also push the parent down to maintain a proper tree

structure. This is useful for identifying unnecessary fringe splits, when the two leaves are essentially identical. It can be used to keep the tree from growing and to increase the tree's predictive accuracy.

Parameters **instance** (*Instance*) – The instance currently being categorized

Returns the category utility of fringe splitting at the current node.

Return type float

See also:

CobwebNode.get_best_operation()

cu_for_insert(*child*, *instance*)
Compute the category utility of adding the instance to the specified child.

This operation does not actually insert the instance into the child it only calculates what the result of the insertion would be. For the actual insertion function see: *CobwebNode.increment_counts()* This is the function used to determine the best children for each of the other operations.

Parameters

- **child** (CobwebNode) – a child of the current node
- **instance** (*Instance*) – The instance currently being categorized

Returns the category utility of adding the instance to the given node

Return type float

See also:

CobwebNode.two_best_children() and *CobwebNode.get_best_operation()*

cu_for_merge(*best1*, *best2*, *instance*)
Return the category utility for merging the two best children.

This does not actually merge the two children it only calculates what the result of the merge would be. For the actual merge operation see: *CobwebNode.merge()*

Parameters

- **best1** (CobwebNode) – The child of the current node with the best category utility
- **best2** (CobwebNode) – The child of the current node with the second best category utility
- **instance** (*Instance*) – The instance currently being categorized

Returns The category utility that would result from merging best1 and best2.

Return type float

See also:

CobwebNode.get_best_operation()

cu_for_new_child(*instance*)
Return the category utility for creating a new child using the particular instance.

This operation does not actually create the child it only calculates what the result of creating it would be. For the actual new function see: *CobwebNode.create_new_child()*.

Parameters **instance** (*Instance*) – The instance currently being categorized

Returns the category utility of adding the instance to a new child.

Return type float

See also:

CobwebNode.get_best_operation()

cu_for_split(*best*)
Return the category utility for splitting the best child.

This does not actually split the child it only calculates what the result of the split would be. For the actual split operation see: *CobwebNode.split()*. Unlike the category utility calculations for the other operations split does not need the instance because splits trigger a recursive call on the current node.

Parameters **best** (CobwebNode) – The child of the current node with the best category utility

Returns The category utility that would result from splitting best

Return type float

See also:

CobwebNode.get_best_operation()

depth()
Returns the depth of the current node in its tree

Returns the depth of the current node in its tree

Return type int

expected_correct_guesses()
Returns the number of correct guesses that are expected from the given concept.

This is the sum of the probability of each attribute value squared. This function is used in calculating category utility.

Returns the number of correct guesses that are expected from the given concept.

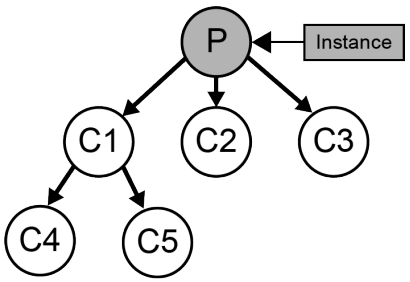
Return type float

gensym()
Generate a unique id and increment the class _counter.

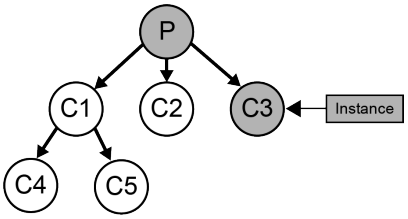
This is used to create a unique name for every concept. As long as the class _counter variable is never externally altered these keys will remain unique.

get_best_operation(*instance, best1, best2, best1_cu, possible_ops=[u'best', u'new', u'merge', u'split']*)
Given an instance, the two best children based on category utility and a set of possible operations, find the operation that produces the highest category utility, and then return the category utility and name for the best operation. In the case of ties, an operator is randomly chosen.

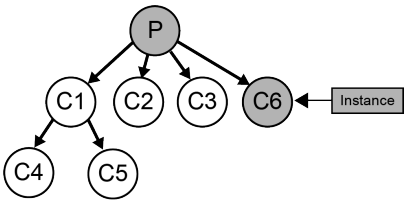
Given the following starting tree the results of the 4 standard Cobweb operations are shown below:



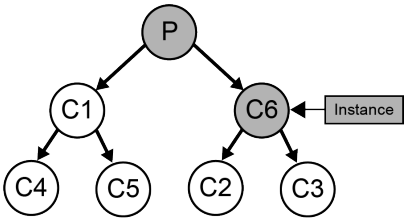
•**Best** - Categorize the instance to child with the best category utility. This results in a recurisve call to *cobweb*.



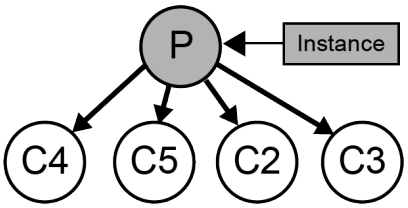
•**New** - Create a new child node to the current node and add the instance there. See: *create_new_child*.



•**Merge** - Take the two best children, create a new node as their mutual parent and add the instance there. See: *merge*.



•**Split** - Take the best node and promote its children to be children of the current node and recurse on the current node. See: *split*



Each operation is entertained and the resultant category utility is used to pick which operation to perform. The list of operations to entertain can be controlled with the possible_ops parameter. For example, when performing categorization without modifying knoweldge only the best and new operators are used.

Parameters

- **instance** (*Instance*) – The instance currently being categorized
- **best1** (*(float, CobwebNode)*) – A tuple containing the relative cu of the best child and the child itself, as determined by *CobwebNode.two_best_children()*.
- **best2** (*(float, CobwebNode)*) – A tuple containing the relative cu of the second best child and the child itself, as determined by *CobwebNode.two_best_children()*.
- **possible_ops** (*["best", "new", "merge", "split"]*) – A list of operations from ["best", "new", "merge", "split"] to entertain.

Returns A tuple of the category utility of the best operation and the name of the best operation.

Return type (cu_bestOp, name_bestOp)

get_weighted_values(*attr*, *allow_none=True*)
Return a list of weighted choices for an attribute based on the node's probability table.

This calculation will include an option for the change that an attribute is missing from an instance all together. This is useful for probability and sampling calculations. If the attribute has never appeared in the tree then it will return a 100% chance of None.

Parameters

- **attr** (*Attribute*) – an attribute of an instance
- **allow_none** (*Boolean*) – whether attributes in the nodes probability table can be inferred to be missing. If False, then None will not be cosidered as a possible value.

Returns a list of weighted choices for attr's value

Return type [(*Value*, float), (*Value*, float), ...]

increment_counts(*instance*)
Increment the counts at the current node according to the specified instance.

Parameters **instance** (*Instance*) – A new instances to incorporate into the node.

is_exact_match(*instance*)
Returns true if the concept exactly matches the instance.

Parameters **instance** (*Instance*) – The instance currently being categorized

Returns whether the instance perfectly matches the concept

Return type boolean

See also:

CobwebNode.get_best_operation()

is_parent(*other_concept*)
Return True if this concept is a parent of other_concept

Returns `True` if this concept is a parent of other_concept else `False`

Return type bool

log_likelihood(*child_leaf*)
Returns the log-likelihood of a leaf contained within the current concept. Note, if the leaf contains multiple instances, then it is treated as if it contained just a single instance (this function is just called multiple times for each instance in the leaf).

merge(*best1*, *best2*)
Merge the two specified nodes.

A merge operation introduces a new node to be the merger of the the two given nodes. This new node becomes a child of the current node and the two given nodes become children of the new node.

Parameters

- **best1** (`CobwebNode`) – The child of the current node with the best category utility
- **best2** (`CobwebNode`) – The child of the current node with the second best category utility

Returns The new child node that was created by the merge

Return type *CobwebNode*

num_concepts()
Return the number of concepts contained below the current node in the tree.

When called on the `CobwebTree.root` this is the number of nodes in the whole tree.

Returns the number of concepts below this concept.

Return type int

output_json()
Outputs the categorization tree in JSON form

Returns an object that contains all of the structural information of the node and its children

Return type obj

predict(*attr*, *choice_fn=u'most likely'*, *allow_none=True*)
Predict the value of an attribute, using the specified choice function (either the "most likely" value or a "sampled" value).

Parameters

- **attr** (*Attribute*) – an attribute of an instance.
- **choice_fn** (*a string*) – a string specifying the choice function to use, either "most likely" or "sampled".
- **allow_none** (*Boolean*) – whether attributes not in the instance can be inferred to be missing. If False, then all attributes will be inferred with some value.

Returns The most likely value for the given attribute in the node's probability table.

Return type *Value*

pretty_print(*depth=0*)
Print the categorization tree

The string formatting inserts tab characters to align child nodes of the same depth.

Parameters **depth** (*int*) – The current depth in the print, intended to be called recursively

Returns a formated string displaying the tree and its children

Return type str

probability(*attr*, *val*)
Returns the probability of a particular attribute value at the current concept. This takes into account the possibilities that an attribute can take any of the values available at the root, or be missing.

If you you want to check if the probability that an attribute is missing, then check for the probability that the val is `None`.

Parameters

- **attr** (*Attribute*) – an attribute of an instance
- **val** (*Value*) – a value for the given attribute or None

Returns The probability of attr having the value val in the current concept.

Return type float

relative_cu_for_insert(*child*, *instance*)
Computes a relative CU score for each insert operation. The relative CU score is more efficient to calculate for each insert operation and is guranteed to have the same rank ordering as the CU score so it can be used

to determine which insert operation is best. The relative CU can be computed from the CU using the following transformation.

$$relative_cu(cu) = (cu - const) * n * (count + 1)$$

where *const* is the one returned by `CobwebNode.compute_relative_CU_const()`, n is the number of children of the current node, and *count* is the number of instances stored in the current node (the root).

The particular *const* value was chosen to make the calculation of the relative cu scores for each insert operation efficient. When computing the CU for inserting the instance into a particular child, the terms in the formula above can be expanded and many of the intermediate calculations cancel out. After these cancelations, computing the relative CU for inserting into a particular child C_i reduces to:

relative_cu_for_insert(C_i) = (C_i.count + 1) * sum_i sum_j P(A_i = V_{ij}| UpdatedC_i)^2 - (C_i.count) * sum_i sum_j P(A_i = V_{ij}| C_i)^2

where $UpdatedC_i$ is a copy of C_i that has been updated with the counts from the given instance.

By computing relative_CU scores instead of CU scores for each insert operation, the time complexity of the underlying Cobweb algorithm is reduced from $O(B^2 \times log_B(n) \times AV)$ to $O(B \times log_B(n) \times AV)$ where B is the average branching factor of the tree, :math‘n‘ is the number of instances being categorized, A is the average number of attributes per instance, and V is the average number of values per attribute.

Parameters

- **child** (CobwebNode) – a child of the current node
- **instance** (*Instance*) – The instance currently being categorized

Returns the category utility of adding the instance to the given node

Return type float

shallow_copy()

Create a shallow copy of the current node (and not its children)

This can be used to copy only the information relevant to the node's probability table without maintaining reference to other elements of the tree, except for the root which is necessary to calculate category utility.

split(*best*)

Split the best node and promote its children

A split operation removes a child node and promotes its children to be children of the current node. Split operations result in a recursive call of cobweb on the current node so this function does not return anything.

Parameters **best** (CobwebNode) – The child node to be split

two_best_children(*instance*)

Calculates the category utility of inserting the instance into each of this node's children and returns the best two. In the event of ties children are sorted first by category utility, then by their size, then by a random value.

Parameters **instance** (*Instance*) – The instance currently being categorized

Returns the category utility and indices for the two best children (the second tuple will be None if there is only 1 child).

Return type ((cu_best1,index_best1),(cu_best2,index_best2))

update_counts_from_node(*node*)

Increments the counts of the current node by the amount in the specified node.

This function is used as part of copying nodes and in merging nodes.

Parameters **node** (CobwebNode) – Another node from the same CobwebTree

concept_formation.cobweb3 module

The Cobweb3 module contains the *Cobweb3Tree* and *Cobweb3Node* classes, which extend the traditional Cobweb capabilities to support numeric values on attributes.

Cobweb3Tree

class concept_formation.cobweb3.**Cobweb3Tree**(*scaling=0.5*, *inner_attr_scaling=True*)

Bases: *concept_formation.cobweb.CobwebTree*

The Cobweb3Tree contains the knowledge base of a partiucluar instance of the Cobweb/3 algorithm and can be used to fit and categorize instances. Cobweb/3's main difference over Cobweb is the ability to handle numerical attributes by applying an assumption that they should follow a normal distribution. For the purposes of Cobweb/3's core algorithms a numeric attribute is any value where `isinstance(instance[attr], Number)` returns `True`.

The scaling parameter determines whether online normalization of continuous attributes is used, and to what standard deviation the values are scaled to. Scaling divides the std of each attribute by the std of the attribute in the root divided by the scaling constant (i.e., $\sigma_{root}/scaling$ when making category utility calculations. Scaling is useful to balance the weight of different numerical attributes, without scaling the magnitude of numerical attributes can affect category utility calculation meaning numbers that are naturally larger will recieve preference in the category utility calculation.

Parameters

- **scaling** (*a float greater than 0.0, None, or False*) – The number of standard deviations numeric attributes are scaled to. By default this value is 0.5 (half a standard deviation), which is the max std of nominal values. If disabiling scaling is desirable, then it can be set to False or None.
- **inner_attr_scaling** – Whether to use the inner most attribute name when scaling numeric attributes. For example, if *('attr', '?o1')* was an attribute, then the inner most attribute would be 'attr'. When using inner most attributes, some objects might have multiple attributes (i.e., 'attr' for different objects) that contribute to the scaling.
- **inner_attr_scaling** – boolean

categorize(*instance*)

Sort an instance in the categorization tree and return its resulting concept.

The instance is passed down the categorization tree according to the normal cobweb algorithm except using only the best operator and without modifying nodes' probability tables. **This process does not modify the tree's knowledge** for a modifying version of labeling use the `CobwebTree.ifit()` function

Parameters **instance** (*Instance*) – an instance to be categorized into the tree.

Returns A concept describing the instance

Return type *CobwebNode*

See also:

`CobwebTree.cobweb()`

clear()

Clears the concepts of the tree, but maintains the scaling parameter.

cobweb(*instance*)
A modification of the cobweb function to update the scales object first, so that attribute values can be properly scaled.

fit(*instances*, *iterations=1*, *randomize_first=True*)
Fit a collection of instances into the tree.

This is a batch version of the ifit function that takes a collection of instances and categorizes all of them. The instances can be incorporated multiple times to burn in the tree with prior knowledge. Each iteration of fitting uses a randomized order but the first pass can be done in the original order of the list if desired, this is useful for initializing the tree with specific prior experience.

Parameters

- **instances** ([*Instance*, *Instance*, ...]) – a collection of instances
- **iterations** (*int*) – number of times the list of instances should be fit.
- **randomize_first** (*bool*) – whether or not the first iteration of fitting should be done in a random order or in the list's original order.

get_inner_attr(*attr*)
Extracts the inner most attribute name from the provided attribute, if the attribute is a tuple and inner_attr_scaling is on. Otherwise it just returns the attribute. This is used to for normalizing attributes.

```
>>> t = Cobweb3Tree()
>>> t.get_inner_attr(('a', '?object1'))
'a'
>>> t.get_inner_attr('a')
'a'
```

ifit(*instance*)
Incrementally fit a new instance into the tree and return its resulting concept.

The cobweb3 version of the `CobwebTree.ifit()` function. This version keeps track of all of the continuous

Parameters **instance** (*Instance*) – An instance to be categorized into the tree.

Returns A concept describing the instance

Return type *Cobweb3Node*

See also:

`CobwebTree.cobweb()`

infer_missing(*instance*, *choice_fn=u'most likely'*, *allow_none=True*)
Given a tree and an instance, returns a new instance with attribute values picked using the specified choice function (either "most likely" or "sampled").

Todo

write some kind of test for this.

Parameters

- **instance** (*Instance*) – an instance to be completed.
- **choice_fn** (*a string*) – a string specifying the choice function to use, either "most likely" or "sampled".

- **allow_none** (*Boolean*) – whether attributes not in the instance can be inferred to be missing. If False, then all attributes will be inferred with some value.

Returns A completed instance

Return type *Instance*

update_scales(*instance*)
Reads through all the attributes in an instance and updates the tree scales object so that the attributes can be properly scaled.

Cobweb3Node

class concept_formation.cobweb3.**Cobweb3Node**(*otherNode=None*)
Bases: *concept_formation.cobweb.CobwebNode*

A Cobweb3Node represents a concept within the knoweldge base of a particular *Cobweb3Tree*. Each node contians a probability table that can be used to calculate the probability of different attributes given the concept that the node represents.

In general the *Cobweb3Tree.ifit()*, *Cobweb3Tree.categorize()* functions should be used to initially interface with the Cobweb/3 knowledge base and then the returned concept can be used to calculate probabilities of certain attributes or determine concept labels.

attrs(*attr_filter=None*)
Iterates over the attributes present in the node's attribute-value table with the option to filter certain types. By default the filter will ignore hidden attributes and yield all others. If the string 'all' is provided then all attributes will be yielded. In neither of those cases the filter will be interpreted as a function that returns true if an attribute should be yielded and false otherwise.

category_utility()
Return the category utility of a particular division of a concept into its children.

Category utility is always calculated in reference to a parent node and its own children. This is used as the heuristic to guide the concept formation process. Category Utility is calculated as:

$$CU(\{C_1, C_2, \cdots, C_n\}) = \frac{1}{n}\sum_{k=1}^{n} P(C_k)\left[\sum_i \sum_j P(A_i = V_{ij}|C_k)^2\right] - \sum_i \sum_j P(A_i = V_{ij})^2$$

where n is the numer of children concepts to the current node, $P(C_k)$ is the probability of a concept given the current node, $P(A_i = V_{ij}|C_k)$ is the probability of a particular attribute value given the concept C_k, and $P(A_i = V_{ij})$ is the probability of a particular attribute value given the current node.

In general this is used as an internal function of the cobweb algorithm but there may be times when it would be useful to call outside of the algorithm itself.

Returns The category utility of the current node with respect to its children.

Return type float

compute_relative_CU_const(*instance*)
Computes the constant value that is used to convert between CU and relative CU scores. The constant value is basically the category utility that results from adding the instance to the root, but none of the children. It can be computed directly as:

$$const = \frac{1}{n}\sum_{k=1}^{n}\left[\frac{C_k.count}{count+1}\sum_i \sum_j P(A_i = V_{ij}|C)^2\right] - \sum_i \sum_j P(A_i = V_{ij}|UpdatedRoot)^2$$

where n is the number of children of the root, C_k is child k, $C_k.count$ is the number of instances stored in child C_k, $count$ is the number of instances stored in the root. Finally, $UpdatedRoot$ is a copy of the root that has been updated with the counts of the instance.

Parameters **instance** (*Instance*) – The instance currently being categorized

Returns The value of the constant used to relativize the CU.

Return type float

create_child_with_current_counts()
Create a new child (to the current node) with the counts initialized by the *current node's counts*.

This operation is used in the speical case of a fringe split when a new node is created at a leaf.

Returns The new child

Return type *CobwebNode*

create_new_child(*instance*)
Create a new child (to the current node) with the counts initialized by the *given instance*.

This is the operation used for creating a new child to a node and adding the instance to it.

Parameters **instance** (*Instance*) – The instance currently being categorized

Returns The new child

Return type *CobwebNode*

cu_for_fringe_split(*instance*)
Return the category utility of performing a fringe split (i.e., adding a leaf to a leaf).

A "fringe split" is essentially a new operation performed at a leaf. It is necessary to have the distinction because unlike a normal split a fringe split must also push the parent down to maintain a proper tree structure. This is useful for identifying unnecessary fringe splits, when the two leaves are essentially identical. It can be used to keep the tree from growing and to increase the tree's predictive accuracy.

Parameters **instance** (*Instance*) – The instance currently being categorized

Returns the category utility of fringe splitting at the current node.

Return type float

See also:

`CobwebNode.get_best_operation()`

cu_for_insert(*child*, *instance*)
Compute the category utility of adding the instance to the specified child.

This operation does not actually insert the instance into the child it only calculates what the result of the insertion would be. For the actual insertion function see: `CobwebNode.increment_counts()` This is the function used to determine the best children for each of the other operations.

Parameters

- **child** (`CobwebNode`) – a child of the current node
- **instance** (*Instance*) – The instance currently being categorized

Returns the category utility of adding the instance to the given node

Return type float

See also:

`CobwebNode.two_best_children()` and `CobwebNode.get_best_operation()`

cu_for_merge(*best1*, *best2*, *instance*)
Return the category utility for merging the two best children.

This does not actually merge the two children it only calculates what the result of the merge would be. For the actual merge operation see: `CobwebNode.merge()`

Parameters

- **best1** (`CobwebNode`) – The child of the current node with the best category utility
- **best2** (`CobwebNode`) – The child of the current node with the second best category utility
- **instance** (*Instance*) – The instance currently being categorized

Returns The category utility that would result from merging best1 and best2.

Return type float

See also:

`CobwebNode.get_best_operation()`

cu_for_new_child(*instance*)
Return the category utility for creating a new child using the particular instance.

This operation does not actually create the child it only calculates what the result of creating it would be. For the actual new function see: `CobwebNode.create_new_child()`.

Parameters **instance** (*Instance*) – The instance currently being categorized

Returns the category utility of adding the instance to a new child.

Return type float

See also:

`CobwebNode.get_best_operation()`

cu_for_split(*best*)
Return the category utility for splitting the best child.

This does not actually split the child it only calculates what the result of the split would be. For the actual split operation see: `CobwebNode.split()`. Unlike the category utility calculations for the other operations split does not need the instance because splits trigger a recursive call on the current node.

Parameters **best** (`CobwebNode`) – The child of the current node with the best category utility

Returns The category utility that would result from splitting best

Return type float

See also:

`CobwebNode.get_best_operation()`

depth()
Returns the depth of the current node in its tree

Returns the depth of the current node in its tree

Return type int

expected_correct_guesses()
Returns the number of attribute values that would be correctly guessed in the current concept. This extension supports both nominal and numeric attribute values.

The typical Cobweb/3 calculation for correct guesses is:

$$P(A_i = V_{ij})^2 = \frac{1}{2 * \sqrt{\pi} * \sigma}$$

However, this does not take into account situations when $P(A_i) < 1.0$. Additionally, the original formulation set σ to have a user specified minimum value. However, for small lower bounds, this lets cobweb achieve more than 1 expected correct guess per attribute, which is impossible for nominal attributes (and does not really make sense for continuous either). This causes problems when both nominal and continuous values are being used together; i.e., continuous attributes will get higher preference.

To account for this we use a modified equation:

$$P(A_i = V_{ij})^2 = P(A_i)^2 * \frac{1}{2 * \sqrt{\pi} * \sigma}$$

The key change here is that we multiply by $P(A_i)^2$. Further, instead of bounding σ by a user specified lower bound (often called acuity), we add some independent, normally distributed noise to sigma: $\sigma = \sqrt{\sigma^2 + \sigma_{noise}^2}$, where $\sigma_{noise} = \frac{1}{2*\sqrt{\pi}}$. This ensures the expected correct guesses never exceeds 1. From a theoretical point of view, it basically is an assumption that there is some independent, normally distributed measurement error that is added to the estimated error of the attribute (https://en.wikipedia.org/wiki/Sum_of_normally_distributed_random_variables). It is possible that there is additional measurement error, but the value is chosen so as to yield a sensical upper bound on the expected correct guesses.

Returns The number of attribute values that would be correctly guessed in the current concept.

Return type float

gensym()

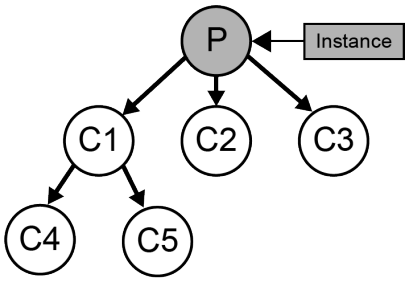
Generate a unique id and increment the class _counter.

This is used to create a unique name for every concept. As long as the class _counter variable is never externally altered these keys will remain unique.

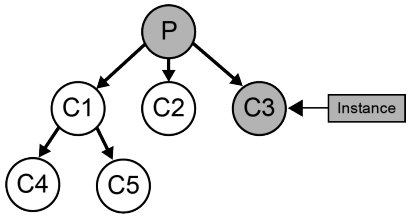
get_best_operation(*instance, best1, best2, best1_cu, possible_ops=[u'best', u'new', u'merge', u'split']*)

Given an instance, the two best children based on category utility and a set of possible operations, find the operation that produces the highest category utility, and then return the category utility and name for the best operation. In the case of ties, an operator is randomly chosen.

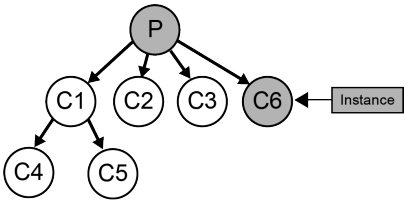
Given the following starting tree the results of the 4 standard Cobweb operations are shown below:



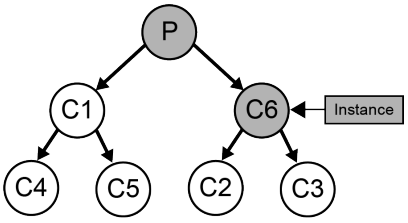
•**Best** - Categorize the instance to child with the best category utility. This results in a recurisve call to *cobweb*.



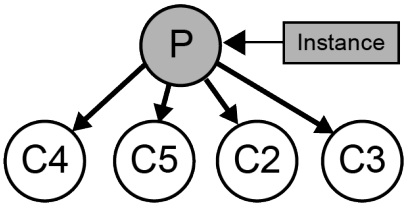
•**New** - Create a new child node to the current node and add the instance there. See: *create_new_child*.



•**Merge** - Take the two best children, create a new node as their mutual parent and add the instance there. See: *merge*.



•**Split** - Take the best node and promote its children to be children of the current node and recurse on the current node. See: *split*



Each operation is entertained and the resultant category utility is used to pick which operation to perform. The list of operations to entertain can be controlled with the possible_ops parameter. For example, when performing categorization without modifying knoweldge only the best and new operators are used.

Parameters

- **instance** (*Instance*) – The instance currently being categorized
- **best1** (*(float, CobwebNode)*) – A tuple containing the relative cu of the best child and the child itself, as determined by `CobwebNode.two_best_children()`.
- **best2** (*(float, CobwebNode)*) – A tuple containing the relative cu of the second best child and the child itself, as determined by `CobwebNode.two_best_children()`.
- **possible_ops** (*["best", "new", "merge", "split"]*) – A list of operations from ["best", "new", "merge", "split"] to entertain.

Returns A tuple of the category utility of the best operation and the name of the best operation.

Return type (cu_bestOp, name_bestOp)

get_weighted_values(*attr*, *allow_none=True*)

Return a list of weighted choices for an attribute based on the node's probability table.

This calculation will include an option for the change that an attribute is missing from an instance all together. This is useful for probability and sampling calculations. If the attribute has never appeared in the tree then it will return a 100% chance of None.

Parameters

- **attr** (*Attribute*) – an attribute of an instance
- **allow_none** (*Boolean*) – whether attributes in the nodes probability table can be inferred to be missing. If False, then None will not be cosidered as a possible value.

Returns a list of weighted choices for attr's value

Return type [(*Value*, float), (*Value*, float), ...]

increment_counts(*instance*)

Increment the counts at the current node according to the specified instance.

Cobweb3Node uses a modified version of *CobwebNode.increment_counts* that handles numerical attributes properly. Any attribute value where `isinstance(instance[attr], Number)` returns `True` will be treated as a numerical attribute and included under an assumption that the number should follow a normal distribution.

Warning: If a numeric attribute is found in an instance with the name of a previously nominal attribute, or vice versa, this function will raise an exception. See: *NumericToNominal* for a way to fix this error.

Parameters **instance** (*Instance*) – A new instances to incorporate into the node.

is_exact_match(*instance*)

Returns true if the concept exactly matches the instance.

Parameters **instance** (*Instance*) – The instance currently being categorized

Returns whether the instance perfectly matches the concept

Return type boolean

See also:

`CobwebNode.get_best_operation()`

is_parent(*other_concept*)

Return True if this concept is a parent of other_concept

Returns `True` if this concept is a parent of other_concept else `False`

Return type bool

log_likelihood(*child_leaf*)

Returns the log-likelihood of a leaf contained within the current concept. Note, if the leaf contains multiple instances, then it is treated as if it contained just a single instance (this function is just called multiple times for each instance in the leaf).

merge(*best1*, *best2*)

Merge the two specified nodes.

A merge operation introduces a new node to be the merger of the the two given nodes. This new node becomes a child of the current node and the two given nodes become children of the new node.

Parameters

- **best1** (`CobwebNode`) – The child of the current node with the best category utility
- **best2** (`CobwebNode`) – The child of the current node with the second best category utility

Returns The new child node that was created by the merge

Return type *CobwebNode*

num_concepts()
Return the number of concepts contained below the current node in the tree.

When called on the `CobwebTree.root` this is the number of nodes in the whole tree.

Returns the number of concepts below this concept.

Return type int

output_json()
Outputs the categorization tree in JSON form.

This is a modification of the *CobwebNode.output_json* to handle numeric values.

Returns an object that contains all of the structural information of the node and its children

Return type obj

predict(*attr*, *choice_fn=u'most likely'*, *allow_none=True*)
Predict the value of an attribute, using the provided strategy.

If the attribute is a nominal then this function behaves the same as *CobwebNode.predict*. If the attribute is numeric then the mean value from the `ContinuousValue` is chosen.

Parameters

- **attr** (*Attribute*) – an attribute of an instance.
- **allow_none** (*Boolean*) – whether attributes not in the instance can be inferred to be missing. If False, then all attributes will be inferred with some value.

Returns The most likely value for the given attribute in the node's probability table.

Return type *Value*

pretty_print(*depth=0*)
Print the categorization tree

The string formatting inserts tab characters to align child nodes of the same depth. Numerical values are printed with their means and standard deviations.

Parameters **depth** (*int*) – The current depth in the print, intended to be called recursively

Returns a formated string displaying the tree and its children

Return type str

probability(*attr*, *val*)
Returns the probability of a particular attribute value at the current concept.

This takes into account the possibilities that an attribute can take any of the values available at the root, or be missing.

For numerical attributes it returns the integral of the product of two gaussians. One gaussian has $\mu = val$ and $\sigma = \sigma_{noise} = \frac{1}{2*\sqrt{\pi}}$ (where σ_{noise} is from *Cobweb3Node.expected_correct_guesses* and ensures the probability or expected correct guesses never exceeds 1). The second gaussian has the mean ad std values from the current concept with additional gaussian noise (independent and normally distributed noise with $\sigma_{noise} = \frac{1}{2*\sqrt{\pi}}$).

The integral of this gaussian product is another gaussian with μ equal to the concept attribut mean and $\sigma = \sqrt{\sigma_{attr}^2 + 2 * \sigma_{noise}^2}$ or, slightly simplified, $\sigma = \sqrt{\sigma_{attr}^2 + 2 * \frac{1}{2*\pi}}$.

Parameters

- **attr** (*Attribute*) – an attribute of an instance
- **val** (*Value*) – a value for the given attribute

Returns The probability of attr having the value val in the current concept.

Return type float

relative_cu_for_insert(*child*, *instance*)

Computes a relative CU score for each insert operation. The relative CU score is more efficient to calculate for each insert operation and is guranteed to have the same rank ordering as the CU score so it can be used to determine which insert operation is best. The relative CU can be computed from the CU using the following transformation.

$$relative_cu(cu) = (cu - const) * n * (count + 1)$$

where *const* is the one returned by `CobwebNode.compute_relative_CU_const()`, n is the number of children of the current node, and $count$ is the number of instances stored in the current node (the root).

The particular $const$ value was chosen to make the calculation of the relative cu scores for each insert operation efficient. When computing the CU for inserting the instance into a particular child, the terms in the formula above can be expanded and many of the intermediate calculations cancel out. After these cancelations, computing the relative CU for inserting into a particular child C_i reduces to:

relative_cu_for_insert(C_i) = (C_i.count + 1) * sum_i sum_j P(A_i = V_{ij}| UpdatedC_i)^2 - (C_i.count) * sum_i sum_j P(A_i = V_{ij}| C_i)^2

where $UpdatedC_i$ is a copy of C_i that has been updated with the counts from the given instance.

By computing relative_CU scores instead of CU scores for each insert operation, the time complexity of the underlying Cobweb algorithm is reduced from $O(B^2 \times log_B(n) \times AV)$ to $O(B \times log_B(n) \times AV)$ where B is the average branching factor of the tree, :math‘n‘ is the number of instances being categorized, A is the average number of attributes per instance, and V is the average number of values per attribute.

Parameters

- **child** (CobwebNode) – a child of the current node
- **instance** (*Instance*) – The instance currently being categorized

Returns the category utility of adding the instance to the given node

Return type float

shallow_copy()

Create a shallow copy of the current node (and not its children)

This can be used to copy only the information relevant to the node's probability table without maintaining reference to other elements of the tree, except for the root which is necessary to calculate category utility.

split(*best*)

Split the best node and promote its children

A split operation removes a child node and promotes its children to be children of the current node. Split operations result in a recursive call of cobweb on the current node so this function does not return anything.

Parameters **best** (CobwebNode) – The child node to be split

two_best_children(*instance*)

Calculates the category utility of inserting the instance into each of this node's children and returns the best two. In the event of ties children are sorted first by category utility, then by their size, then by a random value.

Parameters **instance** (*Instance*) – The instance currently being categorized

Returns the category utility and indices for the two best children (the second tuple will be None if there is only 1 child).

Return type ((cu_best1,index_best1),(cu_best2,index_best2))

update_counts_from_node(*node*)

Increments the counts of the current node by the amount in the specified node, modified to handle numbers.

> **Warning:** If a numeric attribute is found in an instance with the name of a previously nominal attribute, or vice versa, this function will raise an exception. See: *NumericToNominal* for a way to fix this error.

Parameters **node** (Cobweb3Node) – Another node from the same Cobweb3Tree

concept_formation.trestle module

The Trestle module contains the *TrestleTree* class, which extends Cobweb3 to support component and relational attributes.

TrestleTree

class concept_formation.trestle.**TrestleTree**(*scaling=0.5*, *inner_attr_scaling=True*)

Bases: *concept_formation.cobweb3.Cobweb3Tree*

The TrestleTree instantiates the Trestle algorithm, which can be used to learn from and categorize instances. Trestle adds the ability to handle component attributes as well as relations in addition to the numerical and nominal attributes of Cobweb and Cobweb/3.

The scaling parameter determines whether online normalization of continuous attributes is used, and to what standard deviation the values are scaled to. Scaling divides the std of each attribute by the std of the attribute in the root divided by the scaling constant (i.e., $\sigma_{root}/scaling$ when making category utility calculations. Scaling is useful to balance the weight of different numerical attributes, without scaling the magnitude of numerical attributes can affect category utility calculation meaning numbers that are naturally larger will recieve preference in the category utility calculation.

Parameters

- **scaling** (*a float greater than 0.0, None, or False*) – The number of standard deviations numeric attributes are scaled to. By default this value is 0.5 (half a standard deviation), which is the max std of nominal values. If disabiling scaling is desirable, then it can be set to False or None.
- **inner_attr_scaling** – Whether to use the inner most attribute name when scaling numeric attributes. For example, if *('attr', '?o1')* was an attribute, then the inner most attribute would be 'attr'. When using inner most attributes, some objects might have multiple attributes (i.e., 'attr' for different objects) that contribute to the scaling.
- **inner_attr_scaling** – boolean
- **structure_map_internally** (*boolean*) – Determines whether structure mapping is used at each node during categorization (and when merging), this drastically reduces performance, but allows the category structure to influcence structure mapping.

categorize(*instance*)
Sort an instance in the categorization tree and return its resulting concept.

The instance is passed down the the categorization tree according to the normal cobweb algorithm except using only the new and best opperators and without modifying nodes' probability tables. **This does not modify the tree's knowledge base** for a modifying version see *TrestleTree.ifit()*

This version differs fomr the normal *CobwebTree.categorize* and *Cobweb3Tree.categorize* by structure mapping instances before categorizing them.

Parameters **instance** (*Instance*) – an instance to be categorized into the tree.

Returns A concept describing the instance

Return type *CobwebNode*

See also:

TrestleTree.trestle()

clear()
Clear the tree but keep initialization parameters

cobweb(*instance*)
A modification of the cobweb function to update the scales object first, so that attribute values can be properly scaled.

fit(*instances*, *iterations=1*, *randomize_first=True*)
Fit a collection of instances into the tree.

This is a batch version of the ifit function that takes a collection of instances and categorizes all of them. The instances can be incorporated multiple times to burn in the tree with prior knowledge. Each iteration of fitting uses a randomized order but the first pass can be done in the original order of the list if desired, this is useful for initializing the tree with specific prior experience.

Parameters

- **instances** ([*Instance*, *Instance*, ...]) – a collection of instances
- **iterations** (*int*) – number of times the list of instances should be fit.
- **randomize_first** (*bool*) – whether or not the first iteration of fitting should be done in a random order or in the list's original order.

gensym()
Generates unique names for naming renaming apart objects.

Returns a unique object name

Return type '?o'+counter

get_inner_attr(*attr*)
Extracts the inner most attribute name from the provided attribute, if the attribute is a tuple and inner_attr_scaling is on. Otherwise it just returns the attribute. This is used to for normalizing attributes.

```
>>> t = Cobweb3Tree()
>>> t.get_inner_attr(('a', '?object1'))
'a'
>>> t.get_inner_attr('a')
'a'
```

ifit(*instance*)
Incrementally fit a new instance into the tree and return its resulting concept.

The instance is passed down the tree and updates each node to incorporate the instance. **This modifies the tree's knowledge** for a non-modifying version see: *TrestleTree.categorize()*.

This version is modified from the normal *CobwebTree.ifit* by first structure mapping the instance before fitting it into the knoweldge base.

Parameters **instance** (*Instance*) – an instance to be categorized into the tree.

Returns A concept describing the instance

Return type *Cobweb3Node*

See also:

TrestleTree.trestle()

infer_missing(*instance*, *choice_fn=u'most likely'*, *allow_none=True*)

Given a tree and an instance, returns a new instance with attribute values picked using the specified choice function (either "most likely" or "sampled").

Todo

write some kind of test for this.

Parameters

- **instance** (*Instance*) – an instance to be completed.
- **choice_fn** (*a string*) – a string specifying the choice function to use, either "most likely" or "sampled".
- **allow_none** (*Boolean*) – whether attributes not in the instance can be inferred to be missing. If False, then all attributes will be inferred with some value.

Returns A completed instance

Return type instance

trestle(*instance*)

The core trestle algorithm used in fitting and categorization.

This function is similar to *Cobweb.cobweb* The key difference between trestle and cobweb is that trestle performs structure mapping (see: *structure_map*) before proceeding through the normal cobweb algorithm.

Parameters **instance** (*Instance*) – an instance to be categorized into the tree.

Returns A concept describing the instance

Return type *CobwebNode*

update_scales(*instance*)

Reads through all the attributes in an instance and updates the tree scales object so that the attributes can be properly scaled.

concept_formation.cluster module

The cluster model contains functions computing clustering using CobwebTrees and their derivatives.

concept_formation.cluster.**AIC**(*clusters*, *leaves*)

Calculates the Akaike Information Criterion of the a given clustering from a given tree and set of instances.

This can be used as one of the heuristic functions in *cluster_split_search()*.

$$AIC = 2k - 2\ln(\mathcal{L})$$

- $\ln(\mathcal{L})$ is the total log-likelihood of the cluster concepts
- k is the total number of unique attribute value pairs in the tree root times the number of clusters

Parameters

- **clusters** ({*CobwebNode*, *CobwebNode*, ...}) – A unique set of cluster concepts from the tree
- **tree** (*CobwebTree*, *Cobweb3Tree*, or *TrestleTree*) – The tree that the clusters come from (used to calculated number of parameters)
- **instances** ([*Instance*, *Instance*, ...]) – The set of clustered instances

Returns The AIC of the clustering

Return type float

concept_formation.cluster.**AICc**(*clusters*, *leaves*)

Calculates the Akaike Information Criterion of the a given clustering from a given tree and set of instances with a correction for finite sample sizes.

This can be used as one of the heuristic functions in *cluster_split_search()*.

$$AICc = 2k - 2\ln(\mathcal{L}) + \frac{2k(k+1)}{n-k-1}$$

- $\ln(\mathcal{L})$ is the total log-likelihood of the cluster concepts
- k is the total number of unique attribute value pairs in the tree root times the number of clusters
- n is the number of instances.

Given the particular math of AICc I decided that is n-k-1 = 0 then the function will return float('inf')

Parameters

- **clusters** ({*CobwebNode*, *CobwebNode*, ...}) – A unique set of cluster concepts from the tree
- **tree** (*CobwebTree*, *Cobweb3Tree*, or *TrestleTree*) – The tree that the clusters come from (used to calculated number of parameters)
- **instances** ([*Instance*, *Instance*, ...]) – The set of clustered instances

Returns The AIC of the clustering

Return type float

concept_formation.cluster.**BIC**(*clusters*, *leaves*)

Calculates the Bayesian Information Criterion of the a given clustering from a given tree and set of instances.

This can be used as one of the heuristic functions in *cluster_split_search()*.

$$BIC = k\ln(n) - 2\ln(\mathcal{L})$$

- $\ln(\mathcal{L})$ is the total log-likelihood of the cluster concepts

- k is the total number of unique attribute value pairs in the tree root times the number of clusters
- n is the number of instances.

Parameters

- **clusters** ({*CobwebNode*, *CobwebNode*, ...}) – A unique set of cluster concepts from the tree
- **tree** (*CobwebTree*, *Cobweb3Tree*, or *TrestleTree*) – The tree that the clusters come from (used to calculated number of parameters)
- **instances** ([*Instance*, *Instance*, ...]) – The set of clustered instances

Returns The BIC of the clustering

Return type float

concept_formation.cluster.**CU**(*cluster*, *leaves*)

Calculates the Category Utility of a tree state given clusters and leaves.

This just uses the basic category utility function of a node created from the leave instances. This also negates the result so it can be minimized like the other heuristic functions.

Todo

we might want to do this with infered missing instances rather than leaves

---

Parameters

- **clusters** ({*CobwebNode*, *CobwebNode*, ...}) – A unique set of cluster concepts from the tree
- **tree** (*CobwebTree*, *Cobweb3Tree*, or *TrestleTree*) – The tree that the clusters come from (used to calculated number of parameters)
- **instances** ([*Instance*, *Instance*, ...]) – The set of clustered instances

Returns The CU of the clustering

Return type float

concept_formation.cluster.**cluster**(*tree*, *instances*, *minsplit=1*, *maxsplit=1*, *mod=True*)

Categorize a list of instances into a tree and return a list of flat cluster labelings based on successive splits of the tree.

Parameters

- **tree** (*CobwebTree*, *Cobweb3Tree*, or *TrestleTree*) – A category tree to be used to generate clusters.
- **instances** ([*Instance*, *Instance*, ...]) – A list of instances to cluster
- **minsplit** (*int*) – The minimum number of splits to perform on the tree
- **maxsplit** (*int*) – the maximum number of splits to perform on the tree
- **mod** (*bool*) – A flag to determine if instances will be fit (i.e. modifying knoweldge) or categorized (i.e. not modifiying knowledge)

Returns a list of lists of cluster labels based on successive splits between minsplit and maxsplit.

Return type [[minsplit clustering], [minsplit+1 clustering], .. [maxsplit clustering]]

See also:

cluster_iter()

concept_formation.cluster.**cluster_iter**(*tree*, *instances*, *heuristic=<function CU>*, *minsplit=1*, *maxsplit=100000*, *mod=True*, *labels=True*)

This is the core clustering process that splits the tree according to a given heuristic.

concept_formation.cluster.**cluster_split_search**(*tree*, *instances*, *heuristic=<function CU>*, *minsplit=1*, *maxsplit=1*, *mod=True*, *labels=True*, *verbose=False*)

Find a clustering of the instances given the tree that is based on successive splittings of the tree in order to minimize some heuristic function.

Parameters

- **tree** (*CobwebTree*, *Cobweb3Tree*, or *TrestleTree*) – A category tree to be used to generate clusters.
- **instances** ([*Instance*, *Instance*, ...]) – A list of instances to cluster
- **heuristic** (*a function.*) – A heuristic function to minimize in search
- **minsplit** (*int*) – The minimum number of splits to perform on the tree
- **maxsplit** (*int*) – the maximum number of splits to perform on the tree
- **mod** (*bool*) – A flag to determine if instances will be fit (i.e. modifying knoweldge) or categorized (i.e. not modifiying knowledge)
- **verbose** (*bool*) – If True, the process will print the heuristic at each split as it searches.

Returns a list of cluster labels based on the optimal number of splits according to the heuristic

Return type list

See also:

cluster_iter()

concept_formation.cluster.**depth_labels**(*tree*, *instances*, *mod=True*)

Categorize a list of instances into a tree and return a list of lists of labelings for each instance based on different depth cuts of the tree.

The returned matrix is max(conceptDepth) X len(instances). Labelings are ordered general to specific with final_labels[0] being the root and final_labels[-1] being the leaves.

Parameters

- **tree** (*CobwebTree*, *Cobweb3Tree*, or *TrestleTree*) – A category tree to be used to generate clusters, it can be pre-trained or newly created.
- **instances** ([*Instance*, *Instance*, ...]) – A list of instances to cluster
- **mod** (*bool*) – A flag to determine if instances will be fit (i.e. modifying knoweldge) or categorized (i.e. not modifiying knowledge)

Returns a list of lists of cluster labels based on each depth cut of the tree

Return type [[root labeling], [depth1 labeling], .. [maxdepth labeling]]

concept_formation.cluster.**k_cluster**(*tree*, *instances*, *k=3*, *mod=True*)

Categorize a list of instances into a tree and return a flat cluster where `len(set(clustering)) <= k`.

Clusterings are generated by successively splitting the tree until a split results in a clustering with > k clusters at which point the clustering just before that split is returned. It is possible for this process to return a clustering with < k clusters but not > k clusters.

Parameters

- **tree** (*CobwebTree*, *Cobweb3Tree*, or *TrestleTree*) – A category tree to be used to generate clusters, it can be pre-trained or newly created.
- **instances** ([*Instance*, *Instance*, ...]) – A list of instances to cluster
- **k** (*int*) – A desired number of clusters to generate
- **mod** (*bool*) – A flag to determine if instances will be fit (i.e. modifying knoweldge) or categorized (i.e. not modifiying knowledge)

Returns a flat cluster labeling

Return type [label1, label2, ..]

See also:

cluster_iter()

Warning: k must be >= 2.

concept_formation.evaluation module

The evaluation module contains functions for evaluating the predictive capabilities of CobwebTrees and their derivatives.

concept_formation.evaluation.**incremental_evaluation**(*tree*, *instances*, *attr*, *run_length*, *runs=1*, *score=<function probability>*, *randomize_first=True*)

Given a set of instances and an attribute, perform an incremental prediction task; i.e., try to predict the attribute for each instance before incorporating it into the tree. This will give a type of cross validated result and gives a sense of how performance improves over time.

Incremental evaluation can use different scoring functions depending on the desired evaluation task:

- *probability()* - The probability of the target attribute's value being present (i.e., accuracy). This is the default scoring function.
- *error()* - The difference between the target attribute's value and the one predicted by the tree's concept.
- *absolute_error()* - Returns the absolute value of the error.
- *squared_error()* - Returns the error squared.

Parameters

- **tree** (*CobwebTree*, *Cobweb3Tree*, or *TrestleTree*) – A category tree to evaluate.
- **instances** ([*Instance*, *Instance*, ...]) – A list of instances to use for evaluation
- **attr** (*Attribute*) – A target instance attribute to use in evaluation.
- **run_length** (*int*) – The number of training instances to use within a given run.
- **runs** (*int*) – The number of restarted runs to perform
- **score** (*function*) – The scoring function to use for evaluation (default probability)

- **randomize_first** (*bool*) – Whether to shuffle the first run of instances or not.

Returns A table that is *runs* x *run_length* where each row represents the score for successive instances within a run.

Return type A table of scores.

concept_formation.evaluation.**probability**(*tree*, *instance*, *attr*, *val*)

Returns the probability of a particular value of an attribute in the instance. One of the scoring functions for incremental_evaluation.

If the instance currently contains the target attribute a shallow copy is created to allow the attribute to be predicted.

Warning: This is an older function in the library and we are not quite sure how to set it up for component values under the new representation and so for the time being it will raise an Exception if it encounts a component.

Parameters

- **tree** (*CobwebTree*, *Cobweb3Tree*, or *TrestleTree*) – A category tree to evaluate.
- **instance** (*{a1:v1, a2:v2, ..}*) – An instance to use query the tree with
- **attr** (*Attribute*) – A target instance attribute to evaluate probability on
- **val** (A *Nominal* or *Numeric* value.) – The target value of the given attr

Returns The probabily of the given instance attribute value in the given tree

Return type float

concept_formation.evaluation.**error**(*tree*, *instance*, *attr*, *val*)

Computes the error between the predicted value and the actual value for an attribute. One of the scoring functions for incremental_evaluation.

Warning: We are not quite sure how to compute error or squared for a *Numeric values* being missing (e.g., 0-1 vs. scale of the numeric value cannot be averaged). So currently, this scoring function raises an Exception when it encounters a missing nunmeric value. We are also not sure how to handle error in the case of *Component Values* so it will also throw an exception if encounters one of those.

Parameters

- **tree** (*CobwebTree*, *Cobweb3Tree*, or *TrestleTree*) – A category tree to evaluate.
- **instance** (*{a1:v1, a2:v2, ..}*) – An instance to use query the tree with
- **attr** (*Attribute*) – A target instance attribute to evaluate error on
- **val** (A *Nominal* or *Numeric* value.) – The target value of the given attr

Returns The error of the given instance attribute value in the given tree

Return type float, or int in the nominal case.

concept_formation.evaluation.**absolute_error**(*tree*, *instance*, *attr*, *val*)

Returns the absolute error of the tree for a particular attribute value pair. One of the scoring functions for incremental_evaluation.

Parameters

- **tree** (*CobwebTree*, *Cobweb3Tree*, or *TrestleTree*) – A category tree to evaluate.
- **instance** (*{a1:v1, a2:v2, ..}*) – An instance to use query the tree with
- **attr** (*Attribute*) – A target instance attribute to evaluate error on
- **val** (A *Nominal* or *Numeric* value.) – The target value of the given attr

Returns The error of the given instance attribute value in the given tree

Return type float, or int in the nominal case.

See also:

error()

concept_formation.evaluation.**squared_error**(*tree*, *instance*, *attr*, *val*)

Returns the squared error of the tree for a particular attribute value pair. One of the scoring functions for incremental_evaluation.

Parameters

- **tree** (*CobwebTree*, *Cobweb3Tree*, or *TrestleTree*) – A category tree to evaluate.
- **instance** (*{a1:v1, a2:v2, ..}*) – An instance to use query the tree with
- **attr** (*Attribute*) – A target instance attribute to evaluate error on
- **val** (A *Nominal* or *Numeric* value.) – The target value of the given attr

Returns The error of the given instance attribute value in the given tree

Return type float, or int in the nominal case.

See also:

error()

concept_formation.preprocessor module

This module contains an number of proprocessors that can be used on various forms of raw input data to convert an instance into a shape that Trestle would better understand. Almost all preprocessors preserve the original semantics of an instance and are mainly being used to prep for Trestle's internal operations.

Two abstract preprocessors are defined:

- *Preprocessor* - Defines the general structure of a preprocessor.
- *Pipeline* - Allows for chaining a collection of preprocessors together.

Trestle's normal implementation uses a standard pipeline of preprocessors that run in the following order:

1. *SubComponentProcessor* - Pulls any sub-components present in the instance to the top level of the instance and adds `has-component` relations to preserve semantics.
2. *Flattener* - Flattens component instances into a number of tuples (i.e. `(attr,component)`) for faster hashing and access.
3. ***StructureMapper***
   - Gives any variables unique names so they can be renamed in matching without

   colliding, and matches instances to the root concept.

The remaining preprocessors are helper classes designed to support data that is not stored in Trestle's conventional representation:

- *Tuplizer* - Looks for relation attributes denoted as strings (i.e. `'(relation e1 e1)'`) and replaces the string attribute name with the equivalent tuple representation of the relation.
- *ListProcessor* - Search for list values and extracts their elements into their own objects and replaces the list with ordering and element-of relations. Intended to preserve the semenatics of a list in JSON representation.
- *ObjectVariablizer* - Looks for component objects within an instance and variablizes their names by prepending a `'?'`.
- *NumericToNominal* - Converts numeric values to nominal ones.
- *NominalToNumeric* - Converts nominal values to numeric ones.

class concept_formation.preprocessor.**ExtractListElements**(*gensym=None*)
Bases: *concept_formation.preprocessor.Preprocessor*

A pre-processor that extracts the elements of lists into their own objects

Find all lists in an instance and extract their elements into their own subjects of the main instance.

This is a first subprocess of the *ListProcessor*. None of the list operations are part of *StructureMapper*'s standard pipeline.

Reset the symbol generator for doctesting purposes. >>> _reset_gensym() >>> import pprint >>> instance = {"a": "n", "list1": ["test", {"p": "q", "j": "k"}, {"n": "m"}]} >>> pp = ExtractListElements() >>> instance = pp.transform(instance) >>> pprint.pprint(instance) {'?o1': {'val': 'test'},

'?o2': {'j': 'k', 'p': 'q'}, '?o3': {'n': 'm'}, 'a': 'n', 'list1': ['?o1', '?o2', '?o3']}

Reset the symbol generator for doctesting purposes. >>> _reset_gensym() >>> import pprint >>> instance = {"att1": "V1", 'subobj': {"list1": ["a", "b", "c", {"B": "C", "D": "E"}]}} >>> pprint.pprint(instance) {'att1': 'V1', 'subobj': {'list1': ['a', 'b', 'c', {'B': 'C', 'D': 'E'}]}} >>> pp = ExtractListElements() >>> instance = pp.transform(instance) >>> pprint.pprint(instance) {'att1': 'V1',

'subobj': {'?o1': {'val': 'a'}, '?o2': {'val': 'b'}, '?o3': {'val': 'c'}, '?o4': {'B': 'C', 'D': 'E'}, 'list1': ['?o1', '?o2', '?o3', '?o4']}}

```
>>> instance = pp.undo_transform(instance)
>>> pprint.pprint(instance)
{'att1': 'V1', 'subobj': {'list1': ['a', 'b', 'c', {'B': 'C', 'D': 'E'}]}}
```

transform(*instance*)
Find all lists in an instance and extract their elements into their own subjects of the main instance.

undo_transform(*instance*)
Undoes the list element extraction operation.

class concept_formation.preprocessor.**Flattener**
Bases: *concept_formation.preprocessor.Preprocessor*

Flattens subobject attributes.

Takes an instance that has already been standardized apart and flattens it.

Hierarchy is represented with periods between variable names in the flattened attributes. However, this process converts the attributes with periods in them into a tuple of objects with an attribute as the last element, this is more efficient for later processing.

This is the third and final operation in *StructureMapper*'s standard pipeline.

```
>>> import pprint
>>> flattener = Flattener()
>>> instance = {'a': 1, 'c1': {'b': 1, '_c': 2}}
>>> pprint.pprint(instance)
{'a': 1, 'c1': {'_c': 2, 'b': 1}}
>>> instance = flattener.transform(instance)
>>> pprint.pprint(instance)
{'a': 1, ('_', ('_c', 'c1')): 2, ('b', 'c1'): 1}
>>> instance = flattener.undo_transform(instance)
>>> pprint.pprint(instance)
{'a': 1, 'c1': {'_c': 2, 'b': 1}}
```

```
>>> instance = {'l1': {'l2': {'l3': {'l4': 1}}}}
>>> pprint.pprint(instance)
{'l1': {'l2': {'l3': {'l4': 1}}}}
>>> instance = flattener.transform(instance)
>>> pprint.pprint(instance)
{('l4', ('l3', ('l2', 'l1'))): 1}
>>> instance = flattener.undo_transform(instance)
>>> pprint.pprint(instance)
{'l1': {'l2': {'l3': {'l4': 1}}}}
```

transform(*instance*)
Perform the flattening procedure.

undo_transform(*instance*)
Undo the flattening procedure.

class concept_formation.preprocessor.**ListProcessor**
Bases: *concept_formation.preprocessor.Preprocessor*

Preprocesses out the lists, converting them into objects and relations.

This preprocessor is a pipeline of two operations. First it extracts elements from any lists in the instance and makes them their own subcomponents with unique names. Second it removes the lists altogether and replaces them with a series of relations that both express that subcomponents are elments of the list and the order that they existed in. These two operations transform the list in a way that preserves the semenatics of the original list but makes them compatible with Trestle's understanding of component objects.

None of the list operations are part of *StructureMapper*'s standard pipeline.

Warning: The ListProcessor's undo_transform function is not guaranteed to be deterministic and attempts a best guess at a partial ordering. In most cases this will be fine but in complex instances with multiple lists and user defined ordering relations it can break down. If an ordering cannot be determined then ordering relations are left in place.

Reset the symbol generator for doctesting purposes. >>> _reset_gensym() >>> import pprint >>> instance = {"att1": "val1", "list1":["a", "b", "a", "c", "d"]} >>> lp = ListProcessor() >>> instance = lp.transform(instance) >>> pprint.pprint(instance) {'?o1': {'val': 'a'},

'?o2': {'val': 'b'}, '?o3': {'val': 'a'}, '?o4': {'val': 'c'}, '?o5': {'val': 'd'}, 'att1': 'val1', 'list1': {}, ('has-element', 'list1', '?o1'): True, ('has-element', 'list1', '?o2'): True, ('has-element', 'list1', '?o3'): True, ('has-element', 'list1', '?o4'): True, ('has-element', 'list1', '?o5'): True, ('ordered-list', 'list1', '?o1', '?o2'): True, ('ordered-list', 'list1', '?o2', '?o3'): True, ('ordered-list', 'list1', '?o3', '?o4'): True, ('ordered-list', 'list1', '?o4', '?o5'): True}

```
>>> instance = lp.undo_transform(instance)
>>> pprint.pprint(instance)
{'att1': 'val1', 'list1': ['a', 'b', 'a', 'c', 'd']}
```

Reset the symbol generator for doctesting purposes. >>> _reset_gensym() >>> instance = {‘l1’: [’a’, {‘in1’: 3, ‘in2’: 4}, {‘ag’: ‘b’, ‘ah’: ‘c’}, 12, ‘again’]} >>> lp = ListProcessor() >>> instance = lp.transform(instance) >>> pprint.pprint(instance) {‘?o1’: {‘val’: ‘a’},

‘?o2’: {‘in1’: 3, ‘in2’: 4}, ‘?o3’: {‘ag’: ‘b’, ‘ah’: ‘c’}, ‘?o4’: {‘val’: 12}, ‘?o5’: {‘val’: ‘again’}, ‘l1’: {}, (‘has-element’, ‘l1’, ‘?o1’): True, (‘has-element’, ‘l1’, ‘?o2’): True, (‘has-element’, ‘l1’, ‘?o3’): True, (‘has-element’, ‘l1’, ‘?o4’): True, (‘has-element’, ‘l1’, ‘?o5’): True, (‘ordered-list’, ‘l1’, ‘?o1’, ‘?o2’): True, (‘ordered-list’, ‘l1’, ‘?o2’, ‘?o3’): True, (‘ordered-list’, ‘l1’, ‘?o3’, ‘?o4’): True, (‘ordered-list’, ‘l1’, ‘?o4’, ‘?o5’): True}

```
>>> instance = lp.undo_transform(instance)
>>> pprint.pprint(instance)
{'l1': ['a', {'in1': 3, 'in2': 4}, {'ag': 'b', 'ah': 'c'}, 12, 'again']}
```

Reset the symbol generator for doctesting purposes. >>> _reset_gensym() >>> instance = {‘tta’: ‘alpha’, ‘ttb’:{‘tlist’: [’a’, ‘b’, {‘sub-a’: ‘c’, ‘sub-sub’: {‘s’: ‘d’, ‘sslist’: [’w’, ‘x’, ‘y’, {‘issue’: ‘here’}]}}, ‘g’]}} >>> pprint.pprint(instance) {‘tta’: ‘alpha’,

‘ttb’: {‘tlist’: [’a’, ‘b’, {‘sub-a’: ‘c’,

‘sub-sub’: {‘s’: ‘d’, ‘sslist’: [’w’, ‘x’, ‘y’, {‘issue’: ‘here’}]}},

‘g’]}}

```
>>> lp = ListProcessor()
>>> instance = lp.transform(instance)
>>> pprint.pprint(instance)
{'tta': 'alpha',
 'ttb': {'?o1': {'val': 'a'},
         '?o2': {'val': 'b'},
         '?o3': {'sub-a': 'c',
                 'sub-sub': {'?o4': {'val': 'w'},
                             '?o5': {'val': 'x'},
                             '?o6': {'val': 'y'},
                             '?o7': {'issue': 'here'},
                             's': 'd',
                             'sslist': {}}},
         '?o8': {'val': 'g'},
         'tlist': {}},
 ('has-element', ('sslist', ('sub-sub', ('?o3', 'ttb'))), '?o4'): True,
 ('has-element', ('sslist', ('sub-sub', ('?o3', 'ttb'))), '?o5'): True,
 ('has-element', ('sslist', ('sub-sub', ('?o3', 'ttb'))), '?o6'): True,
 ('has-element', ('sslist', ('sub-sub', ('?o3', 'ttb'))), '?o7'): True,
 ('has-element', ('tlist', 'ttb'), '?o1'): True,
 ('has-element', ('tlist', 'ttb'), '?o2'): True,
 ('has-element', ('tlist', 'ttb'), '?o3'): True,
 ('has-element', ('tlist', 'ttb'), '?o8'): True,
 ('ordered-list', ('sslist', ('sub-sub', ('?o3', 'ttb'))), '?o4', '?o5'): True,
 ('ordered-list', ('sslist', ('sub-sub', ('?o3', 'ttb'))), '?o5', '?o6'): True,
 ('ordered-list', ('sslist', ('sub-sub', ('?o3', 'ttb'))), '?o6', '?o7'): True,
 ('ordered-list', ('tlist', 'ttb'), '?o1', '?o2'): True,
 ('ordered-list', ('tlist', 'ttb'), '?o2', '?o3'): True,
 ('ordered-list', ('tlist', 'ttb'), '?o3', '?o8'): True}
```

```
>>> instance = lp.undo_transform(instance)
>>> pprint.pprint(instance)
{'tta': 'alpha',
 'ttb': {'tlist': ['a',
                   'b',
                   {'sub-a': 'c',
                    'sub-sub': {'s': 'd',
                                'sslist': ['w', 'x', 'y', {'issue': 'here'}]}},
                   'g']}}
```

transform(*instance*)
Extract list elements and replace lists with ordering relations.

undo_transform(*instance*)
Attempt to reconstruct lists from ordering relations and add extracted list elements back to constructed lists.

class concept_formation.preprocessor.**ListsToRelations**
Bases: *concept_formation.preprocessor.Preprocessor*

Converts an object with lists into an object with sub-objects and list relations.

This is a second subprocess of the *ListProcessor*. None of the list operations are part of *StructureMapper*'s standard pipeline.

Reset the symbol generator for doctesting purposes. >>> _reset_gensym() >>> ltr = ListsToRelations() >>> import pprint >>> instance = {"list1": ['a', 'b', 'c']} >>> instance = ltr.transform(instance) >>> pprint.pprint(instance) {'list1': {},

('has-element', 'list1', 'a'): True, ('has-element', 'list1', 'b'): True, ('has-element', 'list1', 'c'): True, ('ordered-list', 'list1', 'a', 'b'): True, ('ordered-list', 'list1', 'b', 'c'): True}

```
>>> instance = {"list1": ['a', 'b', 'c'], "list2": ['w', 'x', 'y', 'z']}
>>> instance = ltr.transform(instance)
>>> pprint.pprint(instance)
{'list1': {},
 'list2': {},
 ('has-element', 'list1', 'a'): True,
 ('has-element', 'list1', 'b'): True,
 ('has-element', 'list1', 'c'): True,
 ('has-element', 'list2', 'w'): True,
 ('has-element', 'list2', 'x'): True,
 ('has-element', 'list2', 'y'): True,
 ('has-element', 'list2', 'z'): True,
 ('ordered-list', 'list1', 'a', 'b'): True,
 ('ordered-list', 'list1', 'b', 'c'): True,
 ('ordered-list', 'list2', 'w', 'x'): True,
 ('ordered-list', 'list2', 'x', 'y'): True,
 ('ordered-list', 'list2', 'y', 'z'): True}
```

Reset the symbol generator for doctesting purposes. >>> _reset_gensym() >>> ltr = ListsToRelations() >>> import pprint >>> instance = {'o1': {"list1":['c','b','a']}} >>> instance = ltr.transform(instance) >>> pprint.pprint(instance) {'o1': {'list1': {}},

('has-element', ('list1', 'o1'), 'a'): True, ('has-element', ('list1', 'o1'), 'b'): True, ('has-element', ('list1', 'o1'), 'c'): True, ('ordered-list', ('list1', 'o1'), 'b', 'a'): True, ('ordered-list', ('list1', 'o1'), 'c', 'b'): True}

```
>>> instance = ltr.undo_transform(instance)
>>> pprint.pprint(instance)
{'o1': {'list1': ['c', 'b', 'a']}}
```

transform(*instance*)

undo_transform(*instance*)

Traverse the instance and turns each set of totally ordered list relations into a list.

If there is a cycle or a partial ordering, than the relations are not converted and left as they are.

class concept_formation.preprocessor.**NameStandardizer**(*gensym=None*)

Bases: *concept_formation.preprocessor.Preprocessor*

A preprocessor that standardizes apart object names.

Given an instance rename all the components so they have unique names.

This will rename component attributes as well as any occurance of the component's name within relation attributes. This renaming is necessary to allow for a search between possible mappings without collisions.

This is the first operation in *StructureMapper*'s standard pipeline.

Parameters **gensym** (*a function*) – a function that returns unique object names (str) on each call. If None, then *default_gensym()* is used, which keeps a global object counter.

Reset the symbol generator for doctesting purposes. >>> _reset_gensym() >>> import pprint >>> instance = {'nominal': 'v1', 'numeric': 2.3, 'c1': {'a1': 'v1'}, '?c2': ... {'a2': 'v2', '?c3': {'a3': 'v3'}}, '(relation1 c1 ?c2)': ... True, 'lists': [{'c1': {'inner': 'val'}}, 's2', 's3'], ... '(relation2 (a1 c1) (relation3 (a3 (?c3 ?c2))))': 4.3, ... ('relation4', '?c2', '?c4'):True} >>> tuplizer = Tuplizer() >>> instance = tuplizer.transform(instance) >>> std = NameStandardizer() >>> std.undo_transform(instance) Traceback (most recent call last):

...

Exception: Must call transform before undo_transform! >>> new_i = std.transform(instance) >>> old_i = std.undo_transform(new_i) >>> pprint.pprint(instance) {'?c2': {'?c3': {'a3': 'v3'}, 'a2': 'v2'},

'c1': {'a1': 'v1'}, 'lists': [{'c1': {'inner': 'val'}}, 's2', 's3'], 'nominal': 'v1', 'numeric': 2.3, ('relation1', 'c1', '?c2'): True, ('relation2', ('a1', 'c1'), ('relation3', ('a3', ('?c3', '?c2')))): 4.3, ('relation4', '?c2', '?c4'): True}

```
>>> pprint.pprint(new_i)
{'?o1': {'?o2': {'a3': 'v3'}, 'a2': 'v2'},
 'c1': {'a1': 'v1'},
 'lists': [{'c1': {'inner': 'val'}}, 's2', 's3'],
 'nominal': 'v1',
 'numeric': 2.3,
 ('relation1', 'c1', '?o1'): True,
 ('relation2', ('a1', 'c1'), ('relation3', ('a3', ('?o2', '?o1')))): 4.3,
 ('relation4', '?o1', '?o3'): True}
>>> pprint.pprint(old_i)
{'?c2': {'?c3': {'a3': 'v3'}, 'a2': 'v2'},
 'c1': {'a1': 'v1'},
 'lists': [{'c1': {'inner': 'val'}}, 's2', 's3'],
 'nominal': 'v1',
 'numeric': 2.3,
 ('relation1', 'c1', '?c2'): True,
 ('relation2', ('a1', 'c1'), ('relation3', ('a3', ('?c3', '?c2')))): 4.3,
 ('relation4', '?c2', '?c4'): True}
```

transform(*instance*)
Performs the standardize apart tranformation.

undo_transform(*instance*)
Undoes the standardize apart tranformation.

class concept_formation.preprocessor.**NominalToNumeric**(*on_fail=u'break'*, **attrs*)
Bases: *concept_formation.preprocessor.OneWayPreprocessor*

Converts nominal values to numeric ones.

Cobweb3 and *Trestle* will treat anything that passes `isinstance(instance[attr],Number)` as a numerical value. Because of how they store numerical distribution information, If either algorithm encounts a numerical value where it previously saw a nominal one it will throw an error. This preprocessor is provided as a way to address that problem by unifying the value types of attributes across an instance.

Because parsing numbers is a less automatic function than casting things to strings this preprocessor has an extra parameter from *NumericToNominal*. The on_fail parameter determines what should be done in the event of a parsing error and provides 3 options:

- `'break'` - Simply raises the ValueError that caused the problem and fails. **(Default)**
- `'drop'` - Drops any attributes that fail to parse. They would be treated as missing by categorization.
- `'zero'` - Replaces any problem values with `0.0`.

This is a helper function preprocessor and so is not part of *StructureMapper*'s standard pipeline.

```
>>> import pprint
>>> ntn = NominalToNumeric()
>>> instance = {"a":"123","b":"12.1241","c":"134"}
>>> instance = ntn.transform(instance)
>>> pprint.pprint(instance)
{'a': 123.0, 'b': 12.1241, 'c': 134.0}
```

```
>>> ntn = NominalToNumeric(on_fail='break')
>>> instance = {"a":"123","b":"12.1241","c":"bad"}
>>> instance = ntn.transform(instance)
Traceback (most recent call last):
    ...
ValueError: could not convert string to float: 'bad'
```

```
>>> ntn = NominalToNumeric(on_fail="drop")
>>> instance = {"a":"123","b":"12.1241","c":"bad"}
>>> instance = ntn.transform(instance)
>>> pprint.pprint(instance)
{'a': 123.0, 'b': 12.1241}
```

```
>>> ntn = NominalToNumeric(on_fail="zero")
>>> instance = {"a":"123","b":"12.1241","c":"bad"}
>>> instance = ntn.transform(instance)
>>> pprint.pprint(instance)
{'a': 123.0, 'b': 12.1241, 'c': 0.0}
```

```
>>> ntn = NominalToNumeric("break","a","b")
>>> instance = {"a":"123","b":"12.1241","c":"bad"}
>>> instance = ntn.transform(instance)
>>> pprint.pprint(instance)
{'a': 123.0, 'b': 12.1241, 'c': 'bad'}
```

Parameters

- **on_fail** (*'break', 'drop', or 'zero'*) – defines what should be done in the event of a numerical parse error
- **attrs** (*strings*) – A list of specific attributes to convert. If left empty all non-component values will be converted.

transform(*instance*)
Transform target attribute values to numeric if they are valid nominals.

class concept_formation.preprocessor.**NumericToNominal**(**attrs*)
Bases: *concept_formation.preprocessor.OneWayPreprocessor*

Converts numeric values to nominal ones.

Cobweb3 and *Trestle* will treat anything that passes `isinstance(instance[attr],Number)` as a numerical value. Because of how they store numerical distribution information, If either algorithm encounts a numerical value where it previously saw a nominal one it will throw an error. This preprocessor is provided as a way to address that problem by unifying the value types of attributes across an instance.

This is a helper function preprocessor and so is not part of *StructureMapper*'s standard pipeline.

```
>>> import pprint
>>> ntn = NumericToNominal()
>>> instance = {"x":12.5,"y":9,"z":"top"}
>>> instance = ntn.transform(instance)
>>> pprint.pprint(instance)
{'x': '12.5', 'y': '9', 'z': 'top'}
```

```
>>> ntn = NumericToNominal("y")
>>> instance = {"x":12.5,"y":9,"z":"top"}
>>> instance = ntn.transform(instance)
>>> pprint.pprint(instance)
{'x': 12.5, 'y': '9', 'z': 'top'}
```

Parameters **attrs** (*strings*) – A list of specific attributes to convert. If left empty all numeric values will be converted.

transform(*instance*)
Transform target attribute values to nominal if they are numeric.

class concept_formation.preprocessor.**ObjectVariablizer**(**attrs*)
Bases: *concept_formation.preprocessor.OneWayPreprocessor*

Converts all attributes with dictionary values into variables by adding a question mark.

Attribute names beginning with *?* are treated as bindable variables while all other attributes names are considered constants. This process searches through an instances and variablizes attributes that might not have been defined this way in the original data.

This is a helper function preprocessor and so is not part of *StructureMapper*'s standard pipeline.

```
>>> from pprint import pprint
>>> instance = {"ob1":{"myX":12.4,"myY":13.1,"myType":"square"},"ob2":{"myX":9.5,
↪"myY":12.6,"myType":"rect"}}
>>> ov = ObjectVariablizer()
>>> instance = ov.transform(instance)
>>> pprint(instance)
{'?ob1': {'myType': 'square', 'myX': 12.4, 'myY': 13.1},
```

```
 '?ob2': {'myType': 'rect', 'myX': 9.5, 'myY': 12.6}}
>>> instance = ov.undo_transform(instance)
>>> pprint(instance)
{'?ob1': {'myType': 'square', 'myX': 12.4, 'myY': 13.1},
 '?ob2': {'myType': 'rect', 'myX': 9.5, 'myY': 12.6}}
>>> instance = {"p1":{"x":12,"y":3},"p2":{"x":5,"y":14},"p3":{"x":4,"y":18},
↪"setttings":{"x_lab":"height","y_lab":"age"}}
>>> ov = ObjectVariablizer("p1","p2","p3")
>>> instance = ov.transform(instance)
>>> pprint(instance)
{'?p1': {'x': 12, 'y': 3},
 '?p2': {'x': 5, 'y': 14},
 '?p3': {'x': 4, 'y': 18},
 'setttings': {'x_lab': 'height', 'y_lab': 'age'}}
```

Parameters **attrs** (*strings*) – A list of specific attribute names to variablize. If left empty then all variables will be converted.

transform(*instance*)
Variablize target attributes.

class concept_formation.preprocessor.**OneWayPreprocessor**
Bases: *concept_formation.preprocessor.Preprocessor*

A template class that defines a transformation function that only works in the forward direction. If undo_transform is called then an exact copy of the given object is returned.

undo_transform(*instance*)
No-op

class concept_formation.preprocessor.**Pipeline**(**preprocessors*)
Bases: *concept_formation.preprocessor.Preprocessor*

A special preprocessor class used to chain together many preprocessors. Supports the same transform and undo_transform functions as a regular preprocessor.

transform(*instance*)
Apply a series of transformations to the instance.

undo_transform(*instance*)
Undo the series of transformations done to the instance.

class concept_formation.preprocessor.**Preprocessor**
Bases: object

A template class that defines the functions a preprocessor class should implement. In particular, a preprocessor should tranform an instance and implement a function for undoing this transformation.

batch_transform(*instances*)
Transforms a collection of instances.

batch_undo(*instances*)
Undoes transformation for a collection of instances

transform(*instance*)
Transforms an instance.

undo_transform(*instance*)
Undoes a transformation to an instance.

class concept_formation.preprocessor.**Sanitizer**(*spec=u'trestle'*)
Bases: *concept_formation.preprocessor.OneWayPreprocessor*

This is a preprocessor that santizes instances to adhere to the general expectations of either Cobweb, Cobweb3 or Trestle. In general this means enforcing that attribute keys are either of type str or tuple and that relational tuples contain only values of str or tuple. The main reason for having this preprocessor is because many other things are valid dictionary keys in python and its possible to have weird behavior as a result.

```
>>> from pprint import pprint
>>> instance = {'a1':'v1','a2':2,'a3':{'aa1':'1','aa2':2},1:'v2',len:'v3',('r1',2,
↪'r3'):'v4',('r4','r5'):{'aa3':4,3:'v6'}}
>>> pprint(instance)
{<built-in function len>: 'v3',
 1: 'v2',
 'a1': 'v1',
 'a2': 2,
 'a3': {'aa1': '1', 'aa2': 2},
 ('r1', 2, 'r3'): 'v4',
 ('r4', 'r5'): {3: 'v6', 'aa3': 4}}
>>> san = Sanitizer('cobweb')
>>> inst = san.transform(instance)
>>> pprint(inst)
{'1': 'v2',
 '<built-in function len>': 'v3',
 'a1': 'v1',
 'a2': 2,
 'a3': "{'aa1':'1','aa2':2}",
 ('r1', 2, 'r3'): 'v4',
 ('r4', 'r5'): "{3:'v6','aa3':4}"}
>>> san = Sanitizer('trestle')
>>> inst = san.transform(instance)
>>> pprint(inst)
{'1': 'v2',
 '<built-in function len>': 'v3',
 'a1': 'v1',
 'a2': 2,
 'a3': {'aa1': '1', 'aa2': 2},
 ('r1', '2', 'r3'): 'v4',
 ('r4', 'r5'): {'3': 'v6', 'aa3': 4}}
```

transform(*instance*)

class concept_formation.preprocessor.**SubComponentProcessor**
Bases: *concept_formation.preprocessor.Preprocessor*

Takes a flattened instance and moves sub-objects (not sub-attributes) to be top-level objects and adds has-component relations to preserve semantics.

This process is primairly used to improve matching by having all sub- component objects exist as their own top level objects with relations describing their original position in the hierarchy. This allows the structure mapper to partially match against subobjects.

This is the second operation in *TrestleTree*'s standard pipeline (after flattening).

> **Warning:** This assumes that the *NameStandardizer* has been run on the instance first otherwise there can be name collisions.

Reset the symbol generator for doctesting purposes. >>> _reset_gensym() >>> import pprint >>> flattener =

Flattener() >>> psc = SubComponentProcessor() >>> instance = {“a1”: “v1”, ”?sub1”: {“a2”: “v2”, “a3”: 3}, ... ”?sub2”: {“a4”: “v4”, ”?subsub1”: {“a5”: “v5”, “a6”: “v6”}, ... ”?subsub2”:{”?subsubsub”: {“a8”: “V8”}, “a7”: 7}}, ... (‘ordered-list’, (‘list1’, (‘?o2’, ‘?o1’)), ‘b’, ‘a’): ... True} >>> pprint.pprint(instance) {‘?sub1’: {‘a2’: ‘v2’, ‘a3’: 3},

‘?sub2’: {‘?subsub1’: {‘a5’: ‘v5’, ‘a6’: ‘v6’}, ‘?subsub2’: {‘?subsubsub’: {‘a8’: ‘V8’}, ‘a7’: 7}, ‘a4’: ‘v4’},

‘a1’: ‘v1’, (‘ordered-list’, (‘list1’, (‘?o2’, ‘?o1’)), ‘b’, ‘a’): True}

```
>>> instance = psc.transform(flattener.transform(instance))
>>> pprint.pprint(instance)
{'a1': 'v1',
 ('a2', '?sub1'): 'v2',
 ('a3', '?sub1'): 3,
 ('a4', '?sub2'): 'v4',
 ('a5', '?subsub1'): 'v5',
 ('a6', '?subsub1'): 'v6',
 ('a7', '?subsub2'): 7,
 ('a8', '?subsubsub'): 'V8',
 ('has-component', '?o1', '?o2'): True,
 ('has-component', '?sub2', '?subsub1'): True,
 ('has-component', '?sub2', '?subsub2'): True,
 ('has-component', '?subsub2', '?subsubsub'): True,
 ('ordered-list', ('list1', '?o2'), 'b', 'a'): True}
>>> instance = psc.undo_transform(instance)
>>> instance = flattener.undo_transform(instance)
>>> pprint.pprint(instance)
{'?sub1': {'a2': 'v2', 'a3': 3},
 '?sub2': {'?subsub1': {'a5': 'v5', 'a6': 'v6'},
           '?subsub2': {'?subsubsub': {'a8': 'V8'}, 'a7': 7},
           'a4': 'v4'},
 'a1': 'v1',
 ('ordered-list', ('list1', ('?o2', '?o1')), 'b', 'a'): True}
```

transform(*instance*)

Travese the instance for objects that contain subobjects and extracts the subobjects to be their own objects at the top level of the instance.

undo_transform(*instance*)

Removes the has-component relations by adding the elements as subobjects.

If a objects is a child in multiple has-component relationships than it is left in relational form (i.e., it cannot be expressed in sub-object form).

class concept_formation.preprocessor.**Tuplizer**

Bases: *concept_formation.preprocessor.Preprocessor*

Converts all string versions of relations into tuples.

Relation attributes are expected to be specified as a string enclosed in `( )` with values delimited by spaces. We conventionally use a prefix notation for relations `(related a b)` but this preprocessor should be flexible enough to handle postfix and prefix.

This is a helper function preprocessor and so is not part of *StructureMapper*‘s standard pipeline.

```
>>> tuplizer = Tuplizer()
>>> instance = {'(foo1 o1 (foo2 o2 o3))': True}
>>> print(tuplizer.transform(instance))
{('foo1', 'o1', ('foo2', 'o2', 'o3')): True}
```

```
>>> print(tuplizer.undo_transform(tuplizer.transform(instance)))
{'(foo1 o1 (foo2 o2 o3))': True}
```

```
>>> instance = {'(place x1 12.4 9.6 (div width 18.2))':True}
>>> tuplizer = Tuplizer()
>>> tuplizer.transform(instance)
{('place', 'x1', 12.4, 9.6, ('div', 'width', 18.2)): True}
```

transform(*instance*)
Convert at string specified relations into tuples.

undo_transform(*instance*)
Convert tuple relations back into their string forms.

concept_formation.preprocessor.**default_gensym**()
Generates unique names for naming renaming apart objects.

Returns a unique object name

Return type 'o'+counter

concept_formation.preprocessor.**get_attribute_components**(*attribute*, *vars_only=True*)
Gets component names out of an attribute

```
>>> from pprint import pprint
>>> attr = ('a', ('sub1', '?c1'))
>>> get_attribute_components(attr)
{'?c1'}
```

```
>>> attr = '?c1'
>>> get_attribute_components(attr)
{'?c1'}
```

```
>>> attr = ('a', ('sub1', 'c1'))
>>> get_attribute_components(attr)
set()
```

```
>>> attr = 'c1'
>>> get_attribute_components(attr)
set()
```

concept_formation.preprocessor.**rename_relation**(*relation*, *mapping*)
Takes a tuplized relational attribute (e.g., `('before', 'o1', 'o2')`) and a mapping and renames the components based on the mapping. This function contains a special edge case for handling dot notation which is used in the NameStandardizer.

Parameters

- **attr** (*Relation Attribute*) – The relational attribute containing components to be renamed
- **mapping** (*dict*) – A dictionary of mappings between component names

Returns A new relational attribute with components renamed

Return type tuple

```
>>> relation = ('foo1', 'o1', ('foo2', 'o2', 'o3'))
>>> mapping = {'o1': 'o100', 'o2': 'o200', 'o3': 'o300'}
```

```
>>> rename_relation(relation, mapping)
('foo1', 'o100', ('foo2', 'o200', 'o300'))
```

```
>>> relation = ('foo1', ('o1', ('o2', 'o3')))
>>> mapping = {('o1', ('o2', 'o3')): 'o100'}
>>> rename_relation(relation, mapping)
('foo1', 'o100')
```

concept_formation.continuous_value module

class concept_formation.continuous_value.**ContinuousValue**

This class is used to store the number of samples, the mean of the samples, and the squared error of the samples for *Numeric Values*. It can be used to perform incremental estimation of the attribute's mean, std, and unbiased std.

Initially the number of values, the mean of the values, and the squared errors of the values are set to 0.

biased_std()

Returns a biased estimate of the std (i.e., the sample std)

Returns biased estimate of the std (i.e., the sample std)

Return type float

combine(*other*)

Combine another ContinuousValue's distribution into this one in an efficient and practical (no precision problems) way.

This uses the parallel algorithm by Chan et al. found at: https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Parallel_algorithm

Parameters **other** (ContinuousValue) – Another ContinuousValue distribution to be incorporated into this one.

copy()

Returns a deep copy of itself.

Returns a deep copy of the continuous value

Return type *ContinuousValue*

integral_of_gaussian_product(*other*)

Computes the integral (from -inf to inf) of the product of two gaussians. It adds gaussian noise to both stds, so that the integral of their product never exceeds 1.

Use formula computed here: http://www.tina-vision.net/docs/memos/2003-003.pdf

output_json()

scaled_biased_std(*scale*)

Returns an biased estimate of the std (see: *ContinuousValue.biased_std()*), but also adjusts the std given a scale parameter.

This is used to return std values that have been normalized by some value. For edge cases, if scale is less than or equal to 0, then scaling is disabled (i.e., scale = 1.0).

Parameters **scale** (*float*) – an amount to scale biased std estimates by

Returns A scaled unbiased estimate of std

Return type float

scaled_unbiased_mean(*shift*, *scale*)
Returns a shifted and scaled unbiased mean. This is equivelent to (self.unbiased_mean() - shift) / scale

This is used as part of numerical value scaling.

Parameters

- **shift** (*float*) – the amount to shift the mean by
- **scale** (*float*) – the amount to scale the returned mean by

Returns `(self.mean - shift) / scale`

Return type float

scaled_unbiased_std(*scale*)
Returns an unbiased estimate of the std (see: *ContinuousValue.unbiased_std()*), but also adjusts the std given a scale parameter.

This is used to return std values that have been normalized by some value. For edge cases, if scale is less than or equal to 0, then scaling is disabled (i.e., scale = 1.0).

Parameters **scale** (*float*) – an amount to scale unbiased std estimates by

Returns A scaled unbiased estimate of std

Return type float

unbiased_mean()
Returns an unbiased estimate of the mean.

Returns the unbiased mean

Return type float

unbiased_std()
Returns an unbiased estimate of the std, but for n < 2 the std is estimated to be 0.0.

This implementation uses Bessel's correction and Cochran's theorem: https://en.wikipedia.org/wiki/Unbiased_estimation_of_standard_deviation#Bias_correction

Returns an unbiased estimate of the std

Return type float

See also:

concept_formation.utils.c4()

update(*x*)
Incrementally update the mean and squared mean error (meanSq) values in an efficient and practical (no precision problems) way.

This uses and algorithm by Knuth found here: https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance

Parameters **x** (*Number*) – A new value to incorporate into the distribution

update_batch(*data*)
Calls the update function on every value in a given dataset

Parameters **data** (*[Number, Number, ..]*) – A list of numberic values to add to the distribution

concept_formation.structure_mapper module

This module contains the *StructureMapper* class which is used rename variable attributes it improve the category utility on instances.

It is an instance of a *preprocessor* with a `transform()` and `undo_tranform()` methods.

class concept_formation.structure_mapper.**StructureMapper**(*base*)
Bases: *concept_formation.preprocessor.Preprocessor*

Structure maps an instance that has been appropriately preprocessed (i.e., standardized apart, flattened, subcomponent processed, and lists processed out). Transform renames the instance based on this structure mapping, and return the renamed instance.

Parameters

- **base** (*TrestleNode*) – A concept to structure map the instance to
- **gensym** (*function*) – a function that returns unique object names (str) on each call

Returns A flattened and mapped copy of the instance

Return type instance

get_mapping()
Returns the currently established mapping.

Returns The current mapping.

Return type dict

transform(*target*, *initial_mapping=None*)
Transforms a provided target (either an instance or an av_counts table from a CobwebNode or Cobweb3Node).

Parameters **target** (*instance or av_counts table (from CobwebNode or Cobweb3Node)*) – An instance or av_counts table to rename to bring into alignment with the provided base.

Returns The renamed instance or av_counts table

Return type instance or av_counts table

undo_transform(*target*)
Takes a transformed target and reverses the structure mapping using the mapping discovered by transform.

Parameters **target** (*previously structure mapped instance or av_counts table (from CobwebNode or Cobweb3Node)*) – A previously renamed instance or av_counts table to reverse the structure mapping on.

Returns An instance or concept av_counts table with original object names

Return type dict

class concept_formation.structure_mapper.**StructureMappingOptimizationProblem**(*initial*, *parent=None*, *action=None*, *initial_cost=0*, *extra=None*)
Bases: py_search.base.Problem

A class for describing a structure mapping problem to be solved using the py_search library. This class defines the node_value, the successor, and goal_test methods used by the search library.

Unlike StructureMappingProblem, this class uses a local search approach; i.e., given an initial mapping it tries to improve the mapping by permuting it.

goal_test (*node*)
This should always return False, so it never terminates early.

node_value (*node*)
The value of a node (based on mapping_cost).

random_successor (*node*)
Similar to the successor function, but generates only a single random successor.

successors (*node*)
An iterator that returns all successors.

swap_two (*o1*, *o2*, *mapping*, *unmapped_cnames*, *target*, *base*, *node*)
returns the child node generated from swapping two mappings.

swap_unnamed (*o1*, *o2*, *mapping*, *unmapped_cnames*, *target*, *base*, *node*)
Returns the child node generated from assigning an unmapped component object to one of the instance objects.

concept_formation.structure_mapper.**bind_flat_attr** (*attr*, *mapping*)
Renames an attribute given a mapping.

Parameters

- **attr** (*str or tuple*) – The attribute to be renamed
- **mapping** (*dict*) – A dictionary of mappings between component names
- **unnamed** (*dict*) – A list of components that are not yet mapped.

Returns The attribute's new name or None if the mapping is incomplete

Return type str

```
>>> attr = ('before', '?c1', '?c2')
>>> mapping = {'?c1': '?o1', '?c2':'?o2'}
>>> bind_flat_attr(attr, mapping)
('before', '?o1', '?o2')
```

```
>>> attr = ('ordered-list', ('cells', '?obj12'), '?obj10', '?obj11')
>>> mapping = {'?obj12': '?o1', '?obj10':'?o2', '?obj11': '?o3'}
>>> bind_flat_attr(attr, mapping)
('ordered-list', ('cells', '?o1'), '?o2', '?o3')
```

If the mapping is incomplete then returns partially mapped attributes

```
>>> attr = ('before', '?c1', '?c2')
>>> mapping = {'?c1': 'o1'}
>>> bind_flat_attr(attr, mapping)
('before', 'o1', '?c2')
```

```
>>> bind_flat_attr(('<', ('a', '?o2'), ('a', '?o1')), {'?o1': '?c1'})
('<', ('a', '?o2'), ('a', '?c1'))
```

```
>>> bind_flat_attr(('<', ('a', '?o2'), ('a', '?o1')),
...                {'?o1': '?c1', '?o2': '?c2'})
('<', ('a', '?c2'), ('a', '?c1'))
```

concept_formation.structure_mapper.**contains_component**(*component*, *attr*)

Return `True` if the given component name is in the attribute, either as part of a hierarchical name or within a relations otherwise `False`.

Parameters

- **component** (*str*) – A component name
- **attr** – An attribute name

Returns `True` if the component name exists inside the attribute name `False` otherwise

Return type bool

```
>>> contains_component('?c1', ('relation', '?c2', ('a', '?c1')))
True
>>> contains_component('?c3', ('before', '?c1', '?c2'))
False
```

concept_formation.structure_mapper.**flat_match**(*target*, *base*, *initial_mapping=None*)

Given a base (usually concept) and target (instance or concept av table) this function returns a mapping that can be used to rename components in the target. Search is used to find a mapping that maximizes the expected number of correct guesses in the concept after incorporating the instance.

The current approach is to refine the initially provided mapping using a local hill-climbing search. If no initial mapping is provided then one is generated using the Munkres / Hungarian matching on object-to-object assignment (no relations). This initialization approach is polynomial in the size of the base.

Parameters

- **target** (*Instance* or av_counts obj from concept) – An instance or concept.av_counts object to be mapped to the base concept.
- **base** (*TrestleNode*) – A concept to map the target to
- **initial_mapping** (*A mapping dict*) – An initial mapping to seed the local search

Returns a mapping for renaming components in the instance.

Return type dict

concept_formation.structure_mapper.**get_component_names**(*instance*, *vars_only=True*)

Given an instance or a concept's probability table return a list of all of the component names. If vars_only is false, than all constants and variables are returned.

Parameters

- **instance** (*an instance*) – An instance or a concept's probability table.
- **vars_only** (*boolean*) – Whether or not to return only variables (i.e., strings with a names with a '?' at the beginning) or both variables and constants.

Returns A frozenset of all of the component names present in the instance

Return type frozenset

```
>>> instance = {('a', ('sub1', 'c1')): 0, ('a', 'c2'): 0,
...             ('_', '_a', 'c3'): 0}
>>> names = get_component_names(instance, False)
```

```
>>> frozenset(names) == frozenset({'c3', 'c2', ('sub1', 'c1'), 'sub1', 'a',
...                                ('a', ('sub1', 'c1')), ('a', 'c2'),
...                                 'c1'})
True
>>> names = get_component_names(instance, True)
>>> frozenset(names) == frozenset()
True
```

```
>>> instance = {('relation1', ('sub1', 'c1'), 'o3'): True}
>>> names = get_component_names(instance, False)
>>> frozenset(names) == frozenset({'o3', ('relation1', ('sub1', 'c1'),
...                                    'o3'), 'sub1', ('sub1', 'c1'),
...                                'c1', 'relation1'})
True
```

concept_formation.structure_mapper.**hungarian_mapping**(*inames*, *cnames*, *target*, *base*)

Utilizes the hungarian/munkres matching algorithm to compute an initial mapping of inames to cnames. The base cost is the expected correct guesses if each object is matched to itself (i.e., a new object). Then the cost of each object-object match is evaluated by setting each individual object and computing the expected correct guesses.

Parameters

- **inames** (*collection*) – the target component names
- **cnames** (*collection*) – the base component names
- **target** (*Instance* or av_counts obj from concept) – An instance or concept.av_counts object to be mapped to the base concept.
- **base** (*TrestleNode*) – A concept to map the target to

Returns a mapping for renaming components in the instance.

Return type frozenset

concept_formation.structure_mapper.**is_partial_match**(*iAttr*, *cAttr*, *mapping*)

Returns True if the instance attribute (iAttr) partially matches the concept attribute (cAttr) given the mapping.

Parameters

- **iAttr** (*str or tuple*) – An attribute in an instance
- **cAttr** (*str or tuple*) – An attribute in a concept
- **mapping** (*dict*) – A mapping between between attribute names
- **unnamed** (*dict*) – A list of components that are not yet mapped.

Returns `True` if the instance attribute matches the concept attribute in the mapping otherwise `False`

Return type bool

```
>>> is_partial_match(('<', ('a', '?o2'), ('a', '?o1')),
...                  ('<', ('a', '?c2'), ('b', '?c1')), {'?o1': '?c1'})
False
```

```
>>> is_partial_match(('<', ('a', '?o2'), ('a', '?o1')),
...                  ('<', ('a', '?c2'), ('a', '?c1')), {'?o1': '?c1'})
True
```

```
>>> is_partial_match(('<', ('a', '?o2'), ('a', '?o1')),
...                  ('<', ('a', '?c2'), ('a', '?c1')),
...                  {'?o1': '?c1', '?o2': '?c2'})
True
```

concept_formation.structure_mapper.**mapping_cost**(*mapping*, *target*, *base*)

Used to evaluate a mapping between a target and a base. This is performed by renaming the target using the mapping, adding it to the base and evaluating the expected number of correct guesses in the newly updated concept.

Parameters

- **mapping** (*frozenset or dict*) – the mapping of target items to base items
- **target** (*an instance or concept.av_counts*) – the target
- **base** (*a concept*) – the base

concept_formation.structure_mapper.**rename_flat**(*target*, *mapping*)

Given an instance and a mapping rename the components and relations and return the renamed instance.

Parameters

- **instance** (*instance*) – An instance to be renamed according to a mapping
- **mapping** (*dict*) –

  param mapping A dictionary of mappings between component

  names

Returns A copy of the instance with components and relations renamed

Return type instance

```
>>> import pprint
>>> instance = {('a', '?c1'): 1, ('good', '?c1'): True}
>>> mapping = {'?c1': '?o1'}
>>> renamed = rename_flat(instance,mapping)
>>> pprint.pprint(renamed)
{('a', '?o1'): 1, ('good', '?o1'): True}
```

concept_formation.datasets module

concept_formation.visualize module

The visualize module provides functions for generating html visualizations of trees created by the other modules of concept_formation.

concept_formation.visualize.**visualize**(*tree*, *dst=u'.'*, *recreate_html=True*)

Create an interactive visualization of a concept_formation tree and open it in your browswer.

Note that this function will create html, js, and css files in the destination directory provided. By default this will always recreate the support html, js, and css files but a flag can turn this off.

Parameters

- **tree** (*CobwebTree*, *Cobweb3Tree*, or *TrestleTree*) – A category tree to visualize
- **dst** (*str*) – A directory to generate visualization files into
- **create_html** (*bool*) – A flag for whether new supporting html files should be created

concept_formation.visualize.**visualize_clusters**(*tree*, *clusters*, *dst=u'.'*, *recreate_html=True*)

Create an interactive visualization of a concept_formation tree trimmed to the level specified by a clustering from the cluster module.

This visualization differs from the normal one by trimming the tree to the level of a clustering. Basically the output traverses down the tree but stops recursing if it hits a node in the clustering. Both label or concept based clusterings are supported as the relevant names will be extracted.

Note that this function will create html, js, and css files in the destination directory provided. By default this will always recreate the support html, js, and css files but a flag can turn this off.

Parameters

- **tree** (*CobwebTree*, *Cobweb3Tree*, or *TrestleTree*) – A category tree to visualize
- **clusters** (*list*) – A list of cluster labels or concept nodes generated by the cluster module.
- **dst** (*str*) – A directory to generate visualization files into
- **create_html** (*bool*) – A flag for whether new supporting html files should be created

concept_formation.visualize.**visualize_no_leaves**(*tree*, *cuts=1*, *dst=u'.'*, *recreate_html=True*)

Create an interactive visualization of a concept_formation tree cuts levels above the leaves and open it in your browswer.

This visualization differs from the normal one by trimming the leaves from the tree. This is often useful in seeing patterns when the individual leaves are overly frequent visual noise.

Note that this function will create html, js, and css files in the destination directory provided. By default this will always recreate the support html, js, and css files but a flag can turn this off.

Parameters

- **tree** (*CobwebTree*, *Cobweb3Tree*, or *TrestleTree*) – A category tree to visualize
- **cuts** (*int*) – The number of times to trim up the leaves
- **dst** (*str*) – A directory to generate visualization files into
- **create_html** (*bool*) – A flag for whether new supporting html files should be created

concept_formation.utils module

The utils module contains a number of utility functions used by other modules.

concept_formation.utils.**c4**(*n*)

Returns the correction factor to apply to unbias estimates of standard deviation in low sample sizes. This implementation is based on a lookup table for n in [2-29] and returns 1.0 for values >= 30.

```
>>> c4(3)
0.886226925452758
```

concept_formation.utils.**isNumber**(*n*)

Check if a value is a number that should be handled differently than nominals.

concept_formation.utils.**mean**(*values*)

Computes the mean of a list of values.

This is primarily included to reduce dependency on external math libraries like numpy in the core algorithm.

Parameters **values** (*list*) – a list of numbers

Returns the mean of the list of values

Return type float

```
>>> mean([600, 470, 170, 430, 300])
394.0
```

concept_formation.utils.**most_likely_choice**(*choices*)

Given a list of tuples [(val, prob),...(val, prob)], returns the value with the highest probability. Ties are randomly broken.

```
>>> options = [('a',.25),('b',.12),('c',.46),('d',.07)]
>>> most_likely_choice(options)
'c'
>>> most_likely_choice(options)
'c'
>>> most_likely_choice(options)
'c'
```

Parameters **choices** (*[(val, prob),..(val, prob)]*) – A list of tuples

Returns the val with the hightest prob

Return type val

concept_formation.utils.**std**(*values*)

Computes the standard deviation of a list of values.

This is primarily included to reduce dependency on external math libraries like numpy in the core algorithm.

Parameters **values** (*list*) – a list of numbers

Returns the standard deviation of the list of values

Return type float

```
>>> std([600, 470, 170, 430, 300])
147.32277488562318
```

concept_formation.utils.**weighted_choice**(*choices*)

Given a list of tuples [(val, prob),...(val, prob)], return a randomly chosen value where the choice is weighted by prob.

Parameters **choices** (*[(val, prob),..(val, prob)]*) – A list of tuples

Returns A choice sampled from the list according to the weightings

Return type val

```
>>> from random import seed
>>> seed(1234)
>>> options = [('a',.25),('b',.12),('c',.46),('d',.07)]
>>> weighted_choice(options)
'd'
>>> weighted_choice(options)
'c'
>>> weighted_choice(options)
'a'
```

See also:

`CobwebNode.sample`

concept_formation.dummy module

The dummy module contains the *DummyTree* class, which can be used as a naive baseline for comparison against CobwebTrees. This class makes predictions based on the overall average of instances it has seen.

class concept_formation.dummy.**DummyTree**
Bases: *concept_formation.trestle.TrestleTree*

The DummyTree is designed to serve as a naive baseline to compare *Trestle* to. The DummyTree can perform *structure mapping* but in all other respects it is effectively a tree that consists of only a root.

categorize(*instance*)
Return the root of the tree. Because the DummyTree contains only 1 node then it will always categorize instances to that node.

This process does not modify the tree's knoweldge. For a modifying version see: *DummyTree.ifit()*.

Parameters **instance** (*Instance*) – an instance to be categorized into the tree.

Returns the root node of the tree containing everything ever added to it.

Return type *Cobweb3Node*

gensym()
Generates unique names for naming renaming apart objects.

Returns a unique object name

Return type '?o'+counter

ifit(*instance*, *do_mapping=False*)
Just maintain a set of counts at the root and use these for prediction.

The structure_map parameter determines whether or not to do structure mapping. This is disabled by default to get a really naive model.

This process modifies the tree's knoweldge. For a non-modifying version see: *DummyTree.categorize()*.

Parameters

- **instance** (*Instance*) – an instance to be categorized into the tree.
- **do_mapping** (*bool*) – a flag for whether or not to do structure mapping.

Returns the root node of the tree containing everything ever added to it.

Return type *Cobweb3Node*

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

D

E

F

G

H

I

K

L

M

N

O

P

V

W