ramp Documentation

Release 0.1

Ken Van Haren

January 04, 2013

CONTENTS

Ramp is a python package for rapid machine learning prototyping. It provides a simple, declarative syntax for exploring features, algorithms and transformations quickly and efficiently. At its core it's a unified pandas-based framework for working with existing python machine learning and statistics libraries (scikit-learn, rpy2, etc.).

CHAPTER

ONE

FEATURES

- Fast caching and persistence of all intermediate and final calculations – nothing is recomputed unnecessarily.
- Advanced training and preparation logic. Ramp respects the current training set, even when using complex trained features and blended predictions, and also tracks the given preparation set (the x values used in feature preparation – e.g. the mean and stdev used for feature normalization.)
- A growing library of feature transformations, metrics and estimators. Ramp's simple API allows for easy extension.

Contents:

1.1 Getting started

1.1.1 Installation

Minimum requirements:

- Numpy
- Scipy
- Pandas
- Scikit-learn

Recommended packages:

- PyTables (for fast persistence)
- nltk (for various text features)
- Gensim (for topic modeling)

Install:

```
pip install ramp
```

or:

```
python setup.py install
```

1.1.2 Basic Usage

See Getting started with Ramp: Detecting Insults

1.2 DataContext

class ramp.context.**DataContext**(*store=None*, *data=None*, *train_index=None*, *prep_index=None*)

All Ramp analyses require a DataContext. A DataContext represents the environment of the analysis. Most importantly this means for a given store and pandas index value, Ramp will consider the data immutable – it will not check the data again to see if it has changed.

Args

store: An instance of *store.Store* or a path. If a path Ramp will default to an *HDFPickleStore* at that path if PyTables is installed, a *PickleStore* otherwise. Defaults to MemoryStore.

data: a pandas DataFrame. If all data has been precomputed this may not be required.

train_index: a pandas Index specifying the data instances to be used in training. Stored results will be cached against this. If not provided, the entire index of *data* will be used.

prep_index: a pandas Index specifying the data instances to be used in prepping ("x" values). Stored results will be cached against this. If not provided, the entire index of *data* will be used.

load_context(*name*)

Loads a previously saved context with given name, assigning the stored training and prep indices and returning any stored config.

save_context(*name*, *config=None*)

Saves this context (specifically it's train and prep indices) to it's store with the given name, along with the config, if provided.

1.3 Configurations

class ramp.configuration.**Configuration**(*target=None*, *features=None*, *model=None*, *metrics=None*, *reporters=None*, *column_subset=None*, *prediction=None*, *predictions_name=None*, *actual=None*)

Defines a specific data analysis model, including features, estimator and target metric. Can be stored (pickled) and retrieved.

Args

target: *Feature* or basestring specifying the target ("y") variable of the analysis.

features: an iterable of *Features* <*Feature*> to be used by the estimator in the analysis.

model: an estimator instance compatible with sklearn estimator conventions. (has fit(x, y) and predict(y) methods).

metrics: an iterable of evaluation 'Metric's used to score predictions.

reporters: an iterable of *Reporter* objects

prediction: a *Feature* transformation of the special *predictions_name* column used to post-process predictions prior to metric scoring.

predictions_name: a unique string used as a column identifier for model predictions. Must be unique among all feature names: eg '$logreg_predictions$'

actual: a *Feature*. Used if *target* represents a transformation that is NOT the actual target "y" values. Used in conjuction v *prediction* to allow model training, predictions and scoring to operate on different values.

class ramp.configuration.**ConfigFactory**(*base_config*, ***kwargs*)

Provides an iterator over passed in configuration values, allowing for easy exploration of models.

Args

base_config: The base *Configuration* to augment

kwargs: Can be any keyword accepted by *Configuration*. Values should be iterables.

1.4 Features

Features are the core of Ramp. They are descriptions of transformations that operate on DataFrame columns.

Things to note:

- Features attempt to store everything they compute for later reuse. They base cache keys on the pandas index and column name, but not the actual data, so for a given column name and index, a Feature will NOT recompute anything, even if you have changed the value inside. (This applies only in the context of a single storage path. Separate stores will not collide of course.)
- Features may depend on target "y" values (Feature selectors for instance). These features will only be built with values in the given *DataContext*'s train_index.
- Similarly, Features may depend on other "x" valeus. For instance, you normalize a column (mean zero, stdev 1) using certain rows. These prepped values are stored as well so they can be used in "un-prepped" contexts (such as prediction on a hold out set). The given *DataContext*'s prep_index indicates which rows are to be used in preparation.
- Feature instances should not store state (except temporarily while being created they have an attached DataContext object). So the same feature object can be re-used in different contexts.

1.4.1 Creating your own features

Extending ramp with your own feature transformations is fairly straightforward. Features that operate on a single feature should inherit from *Feature*, features operating on multiple features should inherit from *ComboFeature*. For either of these, you will need to override the *_create* method, as well as optionally *__init__* if your feature has extra params. Additionally, if your feature depends on other "x" values (for example it normalizes columns using the mean and stdev of the data), you will need to define a *_prepare* method that returns a dict (or other picklable object) with the required values. To get these "prepped" values, you will call *get_prep_data* from your *_create* method. A simple (mathematically unsafe) normalization example:

```
class Normalize(Feature):

    def _prepare(self, data):
        cols = {}
        for col in data.columns:
            d = data[col]
            m = d.mean()
            s = d.std()
            cols[col] = (m, s)
        return cols

    def _create(self, data):
        col_stats = self.get_prep_data(data)
        d = DataFrame(index=data.index)
        for col in data.columns:
            m, s = col_stats[col]
```

```
            d[col] = data[col].map(lambda x: (x - m)/s)
        return d
```

This allows ramp to cache prep data and reuse it in contexts where the initial data is not available, as well as prevent unnecessary recomputation.

1.5 Feature Reference

class ramp.features.base.**AsFactor**(*feature*, *levels=None*)
Maps nominal values to ints and stores mapping. Mapping may be provided at definition.

levels is list of tuples

get_name(*factor*)

class ramp.features.base.**AsFactorIndicators**(*feature*, *levels=None*)
Maps nominal values to indicator columns. So a column with values ['good', 'fair', 'poor'], would be mapped to two indicator columns (the third implied by zeros on the other two columns)

class ramp.features.base.**BaseFeature**(*feature*)

create(*context*, **args*, ***kwargs*)

depends_on_other_x()

depends_on_y()

unique_name

class ramp.features.base.**ColumnSubset**(*feature*, *subset*, *match_substr=False*)

class ramp.features.base.**ComboFeature**(*features*)
Abstract base for more complex features

Inheriting classes responsible for setting human-readable description of feature and parameters on _name attribute.

column_rename(*existing_name*, *hsh=None*)
Like unique_name, but in addition must be unique to each column of this feature. accomplishes this by prepending readable string to existing column name and replacing unique hash at end of column name.

combine(*datas*)
Needs to be overridden

create(*context*, *force=False*)
Caching wrapper around actual feature creation

create_data(*force*)

create_key()

depends_on_other_x()

depends_on_y()

get_prep_data(*data=None*, *force=False*)

get_prep_key()
Stable, unique key for this feature and a given prep_index and train_index. we key on train_index as well because prep data may involve training.

hash_length = 8

re_hsh = <_sre.SRE_Pattern object at 0x2fe4418>

unique_name
Must provide a unique string as a funtion of this feature, its parameter settings, and all it's contained features. It should also be readable and maintain a reasonable length (by hashing, for instance).

class ramp.features.base.**ConstantFeature**(*feature*)

create(*context*, **args*, ***kwargs*)

class ramp.features.base.**Contain**(*feature*, *min=None*, *max=None*)
Trims values to inside min and max.

class ramp.features.base.**Discretize**(*feature*, *cutoffs*, *values=None*)
Bins values based on given cutoffs.

discretize(*x*)

class ramp.features.base.**DummyFeature**
For testing

create(*context*, **args*, ***kwargs*)

ramp.features.base.**F**
alias of Feature

class ramp.features.base.**Feature**(*feature*)
Base class for features operating on a single feature.

create_data(*force*)
Overrides *ComboFeature* create_data method to only operate on a single sub-feature.

class ramp.features.base.**FillMissing**(*feature*, *fill_value*)
Fills *na* values (pandas definition) with *fill_value*.

class ramp.features.base.**GroupAggregate**(*features*, *function*, *name=None*, *data_column=None*, *trained=False*, *groupby_column=None*, ***groupargs*)
Computes an aggregate value by group.

Groups can be specified with kw args which will be passed to the pandas *groupby* method, or by specifying a *groupby_column* which will group by value of that column.

depends_on_y()

class ramp.features.base.**GroupMap**(*feature*, *function*, *name=None*, ***groupargs*)
Applies a function over specific sub-groups of the data Typically this will be with a MultiIndex (hierarchical index). If group is encountered that has not been seen, defaults to global map. TODO: prep this feature

class ramp.features.base.**IndicatorEquals**(*feature*, *value*)
Maps feature to one if equals given value, zero otherwise.

class ramp.features.base.**Length**(*feature*)
Applies builtin *len* to feature.

class ramp.features.base.**Log**(*feature*)
Takes log of given feature. User is responsible for ensuring values are in domain.

class ramp.features.base.**Map**(*feature*, *function*, *name=None*)
Applies given function to feature. Feature *cannot* be anonymous (ie lambda). Must be defined in top level (and thus picklable).

class ramp.features.base.**MissingIndicator**(*feature*)
Adds a missing indicator column for this feature. Indicator will be 1 if given feature *isnan* (numpy definition), 0 otherwise.

class ramp.features.base.**Normalize**(*feature*)
Normalizes feature to mean zero, stdev one.

class ramp.features.base.**Power**(*feature*, *power=2*)
Takes feature to given power. Equivalent to operator: F('a') ** power.

class ramp.features.base.**ReplaceOutliers**(*feature*, *stdevs=7*, *replace='mean'*)

is_outlier(*x*, *mean*, *std*)

ramp.features.base.**contain**(*x*, *mn*, *mx*)

1.5.1 Text Features

class ramp.features.text.**CapitalizationErrors**(*feature*)

class ramp.features.text.**CharFreqKL**(*feature*)

class ramp.features.text.**CharGrams**(*feature*, *chars=4*)

class ramp.features.text.**ClosestDoc**(*feature*, *text*, *doc_splitter=<bound method SentenceTokenizer.tokenize of <ramp.features.text.SentenceTokenizer object at 0x30d8610>>*, *tokenizer=<function tokenize at 0x30b6b18>*, *sim=<function jaccard at 0x30db410>*)

make_docs(*data*)

score(*data*)

class ramp.features.text.**Dictionary**(*mindocs=3*, *maxterms=100000*, *maxdocs=0.9*, *force=False*)

get_dict(*context*, *docs*)

get_tfidf(*context*, *docs*)

name(*docs*, *type_='dict'*)

class ramp.features.text.**ExpandedTokens**(*feature*)

class ramp.features.text.**KeywordCount**(*feature*, *words*)

class ramp.features.text.**LDA**(**args*, ***kwargs*)

class ramp.features.text.**LSI**(**args*, ***kwargs*)

class ramp.features.text.**LongestSentence**(*feature*)

class ramp.features.text.**LongestWord**(*feature*)

class ramp.features.text.**LongwordCount**(*feature, lengths=[6, 7, 8]*)

class ramp.features.text.**NgramCompare**(*feature*, **args*, ***kwargs*)

class ramp.features.text.**NgramCounts**(*feature*, *mindocs=50*, *maxterms=100000*, *maxdocs=1.0*, *bool_=False*, *verbose=False*)

class ramp.features.text.**Ngrams**(*feature*, *ngrams=1*)

class ramp.features.text.**NonDictCount**(*feature*, *exemptions=None*)

class ramp.features.text.**RemoveNonDict**(*feature*)
Expects tokens

class ramp.features.text.**SelectNgramCounts**(*feature*, *selector*, *target*, *n_keep=50*, *train_only=False*, **args*, ***kwargs*)

depends_on_y()

select(*x*, *y*)

class ramp.features.text.**SentenceCount**(*feature*)

class ramp.features.text.**SentenceLSI**(**args*, ***kwargs*)

make_docs(*data*)

class ramp.features.text.**SentenceLength**(*feature*)

class ramp.features.text.**SentenceSlice**(*feature*, *start=0*, *end=None*)

class ramp.features.text.**SentenceTokenizer**

re_sent = <_sre.SRE_Pattern object at 0x30c9210>

tokenize(*s*)

class ramp.features.text.**SpellingErrorCount**(*feature*, *exemptions=None*)

class ramp.features.text.**StringJoin**(*features*, *sep='|'*)

combine(*datas*)

class ramp.features.text.**TFIDF**(*feature*, *mindocs=50*, *maxterms=10000*, *maxdocs=1.0*)

class ramp.features.text.**Tokenizer**(*feature*, *tokenizer=<function tokenize_keep_all at 0x30b6b90>*)

class ramp.features.text.**TopicModelFeature**(*feature*, *topic_modeler=None*, *num_topics=50*, *force=False*, *stored_model=None*, *mindocs=3*, *maxterms=100000*, *maxdocs=0.9*, *tokenizer=<function tokenize at 0x30b6b18>*)

make_engine(*docs*)

make_vectors(*data*, *n=None*)

class ramp.features.text.**TreebankTokenize**(*feature*)

tokenizer = None

class ramp.features.text.**VocabSize**(*feature*)

class ramp.features.text.**WeightedWordCount**(*feature*)

ramp.features.text.**char_kl**(*txt*)

ramp.features.text.**chargrams**(*s*, *n*)

ramp.features.text.**count_spell_errors**(*toks*, *exemptions*)

ramp.features.text.**expanded_tokenize**(*s*)

ramp.features.text.**is_nondict**(*t*)

ramp.features.text.**jaccard**(*a*, *b*)

ramp.features.text.**make_docs_hash**(*docs*)

ramp.features.text.**ngrams**(*toks*, *n*, *sep='|'*)

ramp.features.text.**nondict_w_exemptions**(*toks*, *exemptions*, *count_typos=False*)

ramp.features.text.**train**(*features*)

ramp.features.text.**words**(*text*)

1.5.2 Combo Features

class ramp.features.combo.**Add**(*features*, *name=None*, *fillna=0*)

combine(*datas*)

class ramp.features.combo.**ComboMap**(*features*, *name=None*, *fillna=0*)
abstract base for binary operations on features

class ramp.features.combo.**DimensionReduction**(*feature*, *decomposer=None*)

combine(*datas*)

class ramp.features.combo.**Divide**(*features*, *name=None*, *fillna=0*)

combine(*datas*)

class ramp.features.combo.**Interactions**(*features*)
Inheriting classes responsible for setting human-readable description of feature and parameters on _name attribute.

combine(*datas*)

class ramp.features.combo.**Multiply**(*features*, *name=None*, *fillna=0*)

combine(*datas*)

class ramp.features.combo.**OutlierCount**(*features*, *stdevs=5*)

combine(*datas*)

is_outlier(*x*, *mean*, *std*)

1.5.3 Trained Features

class ramp.features.trained.**FeatureSelector**(*features*, *selector*, *target*, *n_keep=50*, *train_only=True*, *cache=False*)
train_only: if true, features are selected only using training index data (recommended)

combine(*datas*)

depends_on_y()

select(*x*, *y*)

class ramp.features.trained.**Predictions**(*config*, *name=None*, *external_context=None*, *cv_folds=None*, *cache=False*)
If cv-folds is specified, will use k-fold cross-validation to provide robust predictions. (The predictions returned are those predicted on hold-out sets only.) Will not provide overly-optimistic fit like Predictions will, but can increase runtime significantly (nested cross-validation). Can be int or iteratable of (train, test) indices

depends_on_other_x()

depends_on_y()

get_context()

class ramp.features.trained.**Residuals**(*config*, *name=None*, *external_context=None*, *cv_folds=None*, *cache=False*)

If cv-folds is specified, will use k-fold cross-validation to provide robust predictions. (The predictions returned are those predicted on hold-out sets only.) Will not provide overly-optimistic fit like Predictions will, but can increase runtime significantly (nested cross-validation). Can be int or iteratable of (train, test) indices

1.6 Stores

class ramp.store.**MemoryStore**(*path=None*, *verbose=False*)

Caches values in-memory, no persistence.

class ramp.store.**PickleStore**(*path=None*, *verbose=False*)

Pickles values to disk and caches in memory.

class ramp.store.**HDFPickleStore**(*path=None*, *verbose=False*)

Attempts to store objects in HDF5 format (numpy/pandas objects). Pickles them to disk if that's not possible; also caches values in-memory.

CHAPTER

TWO

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

r