Feature Forge Documentation

Release 0.1.6

Daniel Moisset

June 15, 2015

Contents

Contents:

CHAPTER 1

Key concepts

Machine Learning problems typically involve extracting several pieces of information from your raw data. For example, if you are building a system to classify e-mail as spam or non-spam, your "raw data" is some representation of an email message, and relevant properties might be the presence of images, what is the domain of the sender, if a relevant keyword is in the subject line, the body length, etc.

Each element of raw data to work with (in the above example, an email message) is called a *data point*; you can use any python data structure you like to represent those (strings, dictionaries, tuples, objects, etc). Each of the distinct properties you analyze from a data point is called a *feature* (also called "attribute" in some of the literature, but we'll stick to the *feature* nomenclature)

In python code, a feature will essentially be a function that maps data points into *feature values*. For some features, there's a limited set of possible values (the feature "presence of images" ranges into "yes"/"no"), others can have discrete numeric values (example: "count of bytes in the message"), and others can have continuous values (example: "ratio of upper/lowercase characters in the subject").

It is also possible to think about sets of related features as a vector typed feature value. For example, instead of having a feature "frequency of the letter A", another "frequency of the letter B", and another 24 up to Z, you could have a single feature "frequency of each letter" with a value which is a 26-element vector with each element in [0..1]

CHAPTER 2

Feature definition

2.1 The basics

Feature Forge provides you some help with the boilerplate of defining features and help you testing them

In its most basic form, you can define a feature just as a function, for example

```
# Assuming our data points are dictionaries

def body_length(message):
    return len(message["body"])
```

Your function should take a single argument (of whatever type you are using as a data point) and return a value of any of the following types:

- a string (with a tag, for multi-valued features)
- a list or set of strings (or other hashables objects)
- a number (float or int, for numerical features)
- a list/vector of numbers for vector-typed features

2.2 Testing

Even if a lot of the features you write have only a few lines of code, making mistakes while writing features is surprisingly common and dangerous. Most features are not used directly by your code, and the results aren't directly exposed to a person. Instead, most features are applied to a large, unknown set of data, and its results aggregated statistically. If you're lucky, the mistakes will produce exceptions that you'll be able to detect and fix, but in many cases the feature is producing a wrong result, and it's not even obvious that there's a problem when looking at a statistical aggregate value.

Other problem that you'll find frequently is that your source of data points may not be completely "clean", and some of the input entries have not just wrong values but also different structure (a missing field, or a numeric field with a string value).

Feature Forge provides you some tools to document and check that your features are getting correct data, that (for some handcrafted examples) it is producing the right result, and that it tolerates some sample data that our tool generates automatically.

Validation requires some additional information about the feature: specifically the main type/properties of the input and output values of the feature. As a way to specify those, FeatureForge provides a system based on Vladimir

Keleshev's *schema* python module. A quick look at its documentation should help you to know what you can express with it. In this documentation we will provide a few examples useful for common machine learning situations.

In the next example, data points are dictionaries with at least a key called "body" tied to a unicode string (the message body), and feature value is a non-negative integer. You can specify that in the following way

```
from featureforge.feature import input_schema, output_schema

@input_schema({"body": unicode})
@output_schema(int, lambda i: i >= 0)
def body_length(message):
    return len(message["body"])
```

There a few important aspects of this code:

- Adding the specifications above does not affect the behavior of your function when using in a standalone way. It is just a declarative specification that is usable by the testing and vectorization components of Feature Forge
- When you use dictionaries as schemas, the semantic is a bit more relaxed than in the schema library, because it implicitly allows additional keys so an input {"subject": u"Hi", "body": u"I'm here"} would be a valid input for the input schema above.

Once you have defined your feature like this, you can write a basic test case

```
import unittest
from featureforge.validate import BaseFeatureFixture, EQ, IN, APPROX, RAISES

class TestBodyLength(unittest.TestCase, BaseFeatureFixture):
    feature = body_length
    fixtures = dict(
            test_eq=({"body": u"hello"}, EQ, 5),
            test_approx=({"body": "world!"}, APPROX, 6.00001),
            test_in=({"body": u"x"}, IN, [1,3,7]),
            test_raise=({}, RAISES, ValueError),
        )
```

When running the testcase above with a test runner, you'll have the feature tested against each row of the fixture. Note that the fixture is a dictionary where the keys are labels (which are used in the error message if the check fails), and a triple with input data, a predicate, and an argument for the predicate. The example above covers the possible predicates: equality, approximate equality, inclusion in a set, or failing. Note that any input which does not validate the input_schema will result on a ValueError while testing.

The test case above will also verify that the outputs satisfy the output schema.

The last validation that the test above gives you is what we call a "fuzzy" validation. Some random values are generated (following the input schema), and the feature is called on those values. This check allows you to "stress" the feature code and detect edge cases that might make it fail, or produce a value that does not follow the output schema. Fuzzy validation only is available with schemas describing "standard" python objects (numbers, strings, lists, dictionaries), and may not be available if you add complex lambda expressions or custom types to your schemas. This might be extended to support custom generator of data points in the future.

The fuzzy generator supports: `int`, `float`, `bool`, `str`, `unicode`, `datetime`. Also nested combinations of those built with `dict` (with literal string keys), `list`, `tuple`, `set` and `frozenset`. `schema.Or` is supported. `schema.And` only works if the first argument is a type, and the other conditions are "easy" to validate, where "easy" means "it is likely to find a valid value after a few hundred tries of the random value generator"

It is also possible to extend the class above adding additional test methods, just like you do in any *TestCase* subclass.

2.3 Feature factories and renaming

Sometimes it's useful to use some python metaprogramming tricks to build many similar features. Let's say that your data point is a social media post, represented as a dictionary with many numeric values representing different properties (number of comments, number of likes, number of shares) that you want to extract as features. Writing those features is quite easy, although somewhat tedious and repetitive if you have tens of those properties. You could instead do the following

```
# This is a simple example, don't use this code, there's a better way in a
# later example.
def int_property(label):
    @input_schema({label: int})
    @output_schema(int)
    def get_property(post):
        return post[label]
    return get_property

likes = int_property('likes')
comments = int_property('comments')
shares = int_property('shares')
... etc ...
```

The code above works but has a problem. When using the rest of the framework and having any kind of problem with the features defined this way (for example a test failure, or invalid result when applying to data), the error message will show the name of the function that defines how to compute the feature, and for all of these the function is called "get_property", so you'll lose useful debugging information.

FeatureForge allows you to rename the name used in error messages to provide more valuable error messages, you just need to add a *feature_name* decorator to the function where you can specify a better name (it doesn't need to be a valid python identifier). The better way to write the example above would be

```
def int_property(label):
    @input_schema({label: int})
    @output_schema(int)
    @feature_name("int_property[%s]" % label)
    def get_property(post):
        return post[label]
    return get_property

likes = int_property('likes')
comments = int_property('comments')
... etc ...
```

If you do this, any message related to the features will describe something like "int_property[likes]" which is specific to which of all the features is involved.

2.4 Advanced feature definition

In some cases, when a feature has many parameters or a complex initialization code, it might be more practical to define them as classes instead of functions (bare or decorated). You can do that by defining a subclass of *featureforge.feature.Feature*. In that case, you need to add the feature evaluation code in *_evaluate*, any initialization code you like in *__init__*, and define class attributes *input_schema* and *output_schema*. This is a possible example

```
from featureforge.feature import Feature, soft_schema
from schema import Schema
```

```
class SubjectHasBadWord(Feature):
    input_schema = soft_schema(subject=str)
    # The above is equivalent to
    #   input_schema = Schema({"subject": str, str: Optional(object)})
    output_schema = Schema(bool)

    def __init__(self, bad_words_filename):
        self.bad_words = set(open(bad_words_filename).readlines())

    def _evaluate(self, message):
        subject_words = set(message["subject"]).split()
        return bool(subject_words & self.bad_words)

has_bad_word_english = SubjectHasBadWord("english-badwords.txt")
has_bad_word_spanish = SubjectHasBadWord("spanish-badwords.txt")
```

Note that here each feature is an instance of the subclass. You can use these instances interchangeably with functions in the rest of the library, and you can even call them as functions. One difference with the function API is that if you call them (i.e. *has_bad_word_spanish(some_message)*) and the input/output isn't valid, this will automatically check it and produce an exception. The exception produced is a subclass of ValueError, more precisely *has_bad_word_spanish.InputValueError* or *has_bad_word_spanish.OutputValueError*.

Also note that the schemas have to be built explicitly using the schema module, and that there is no "syntax sugar" for defining schemas as there is in the function decorators. To achieve the same flexibility of the decorators, you might find useful the *soft_schema* function and the *ObjectSchema* class.

If you somehow want the behavior of a Feature subclass (being able to call it with input/output checking, having access to its schema as attributes, etc.), you can use the *make_feature* function, which takes a function-based feature and turns it into a *Feature* instance. you can also use *make_feature* as a decorator if you prefer defining functions but working with objects.

As an implementation detail (and subject to change), all the internals of our library work with *Feature* instances and call *make_feature* internally when receiving a function. So it's possible that you see some Feature() code in a tracebuck while debugging. But for most cases, the function based API should be enough and is more user friendly

2.5 Advanced testing

In most cases, the tests you want for a feature are the ones provided in *BaseFeatureFixture* (and additional ones that you can add). But sometimes you might need to do something more complicated like skipping fuzzy tests, or testing more than one feature in the same *TestCase*, or generating the fixtures on the fly.

If that's the case, you can inherit instead *validate.FeatureFixtureCheckMixin*. This class doesn't define any test (so you have to write it explicitly), but it defines two assertions: *assert_feature_passes_fixture* and *assert_passes_fuzz*. The latter also allows you to manually control how many data points to generate.

Check the API documentation for details on those.

2.6 Specifying schemas

The complete documentation for the schema library is at <https://github.com/halst/schema/blob/master/README.rst>. However we provide here some examples for typical situations.

Output schemas are typically easier than input schemas. Feature values can not be an arbitrary object if you want to feed them to machine learning algorithms, so typically one of the following `output_schema` will be ok:

- `int` or `float` for numeric features
- `unicode` for enumerations (on python 3, `str`)
- `list(int)` or `list(float)` for numeric vectors
- `list(int)` or `list(float)` for numeric vectors
- `list(hashable)` or `set(hashable)` or `tuple(hashable)` for bag of words, where `hashable` is any hashable except numeric.

In addition to a type specification, it is sometimes useful to add to the schema one lambda with an assertion over the value. For example, a feature that always returns positive floats may be specified as

```
@output_schema(float, lambda v: v > 0.0)
```

and a feature that always returns pairs of numbers which are never (0.0, 0.0) can be specified as

```
@output_schema(tuple(float), lambda v: len(v) == 2 and v != (0.0, 0.0))
```

Note that schemas like `tuple(float)` mean "a tuple where all elements are floats" but does not specify its length. A schema `tuple(flaot, int)` means "a tuple where every element is a float or an int"

Data points tend to be more complex objects than feature values, so there is a wider ranges of recipes for input schema. Something important to take into account is that in most problems you have a single type for data point (even when you have many features with different output_schemas). Even if it's possible to build a single, large specification for data points and use it in all your features, that makes testing complicated, because you'll have to build a large, complicated object in your fixtures even if your feature cares about a small aspect of the data (which is the most typical case).

Let's say for example that our data point are email messages, and that a typical data point is a dictionary like this

```
{
    "sender": {
        "address": "johndoe@example.com",
        "label": "John M. Doe"
    },
    "recipient": {
        "address": "janedoe@example.com",
        "label": "Jane N. Doe"
    },
    "subject": "hello",
    "body": "message",
    "date": (2014, 02, 20)
}
```

it's possible to build a general schema like

```
data_schema = schema.Schema({
    "sender": {
        "address": str,
        "label": str
    }
    "recipient": {
        "address": str,
        "label": str
    }
    "subject": str
    "body": str
    "date": schema.And(
        tuple(int),
        lambda date: len(date)==3,
        lambda (y,m,d): 1 <= m <= 12 and 1 <= d <= MONTH_LENGTH[m]
```

```
    )
})
```

And then use it in all your features as `@input_schema(data_schema)`. However, you'll have an easier time specifying, testing, and modifying your system if you specify only what's relevant for each feature; for example

```
@input_schema({"subject": str})
def words_in_subject(...): ...

@input_schema({"sender": {"address": str}, "recipient": {"address": str}})
def sender and_recipient_in_same_domain(...): ...
```

input schemas also allow specifying schemas for attributes of objects if your data point is some custom object, or something like a `namedtuple`. If the example above had a nested tuple structure, these are the schemas you should use

```
@input_schema(subject=str)
def words_in_subject(...): ...

@input_schema(sender=ObjectSchema(address=str),
              recipient=ObjectSchema(address=str))
def sender and_recipient_in_same_domain(...): ...
```

The general rules for `input_schema` and `output_schema` are:

- All position arguments are considered as conditions to be satisfied
- Dictionaries have a slightly different definition that in the standard schema library; they allow additional string keys besides the ones explicitly specified. This change is valid recursively (i.e., if your schema has dictionaries inside dictionaries, the inner dictionaries also allow additional keys)
- You can also use *schema.Schema*, *schema.And*, *schema.Or*, etc. it's recommended (but not required) that you use a type as the first argument of *schema.And*; doing so helps the fuzzy test generator.
- keyword arguments are considered schema conditions for the attributes of the data point. You can mix them with positional arguments; all the conditions must hold. Attributes not mentioned in the schema are allowed and ignored

CHAPTER 3

Cookbook

These are a few tips and tricks to solve common issues:

3.1 Handling "holes" in your input

Sometimes your input data has "holes" in it. For example you might have a data point for people which is a namedtuple(name, age, address), but you do not know the age of everyone so the age field is sometimes an int, and other times a `None`. In that case, you can specify the input schema as

```
@input_schema(age=schema.Or(int, None))
```

note that your feature code will have to handle the case when the input has a None, probably using a `if data_point.age is None: ...`

3.2 Pickling

You might find useful to pickle an object containing features. This is useful while using scikit-learn when you have some classificator or regressor which uses the features as a first step in the pipeline and you want to store the configuration for future uses

Unfortunately, the python decorator mechanism introduces some problems regarding pickling, and even if there is a workaround for some simple cases (using *functools.wraps*), we haven't found a good way to pickle features defined as decorated functions.

If you need to persist the features, you should stick to the class based approach for defining them. It's more verbose, but Feature subclasses should be easy to serialize with pickle or a similar tool if you follow the recipe for defining them used in this document.

CHAPTER 4

Feature evaluation

The most typical use case for features is evaluating a set of them on a collection of data points to feed that information into a machine learning algorithm. FeatureForge provides an evaluation tool that, given a list of features and a collection of data points allows you to produce a matrix with the evaluation results in a format suitable for input into scikit-learn algorithms.

In addition to that, some tools are provided to map back the results (which are essentially numeric) into feature names in order to make analysis of the data easier

4.1 Basic usage

The core class for applying features to values is *featureforge.vectorizer.Vectorizer*. This class is instantiated with a list of features to be used, and then follows the standard pipeline API from scikit learn described at <http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html>.

Roughly, to use it standalone what you do is:

```
v = Vectorizer([some_feature, some_other_feature])
# data is a sequence of data points
v.fit(data)
result = v.transform(data)
```

After this, *result* contains a matrix with the feature evaluations (see the *Generated output*) section below.

In a more typical use, you'll want to put this inside a scikit-learn pipeline, something like this:

```
from sklearn import Pipeline
v = Vectorizer([some_feature, some_other_feature])
... build other steps like classificators/regressors ...
p = Pipeline([v, step2, step3])

p.fit(data)
# Here you can use p methods depending on what you built. See the scikit
# documentation for examples
```

4.2 Generated output

The output for the vectorizer (i.e., the result of its transform() method) is a matrix with one row per each data point, and some columns for each feature. The mapping from feature to columns depends on the type of the feature value:

- Numeric values are mapped to a column with the numeric value

- Enumerated values are mapped to one column for each possible value of the enumeration, with values 0 or 1. For example a feature with possible values “animal”, “vegetable”, “mineral” will be mapped to 3 columns, and for each row only one will have a 1 and the rest will have zeroes.
- Vector features are mapped to a number of columns equal to the size of the vector.
- Bag of Words values, very similary to Enumerated values, are mapped to one column for each possible value, with values from 0 to the number of times that each value appears in the bag. For example a feature with possible values “red”, “green”, “blue” will be mapped to 3 columns, and for each row may be all zeroes if evaluated with empty sequence, all ones if evaluated with [”red”, “green”, “blue”], or a 3 and two zeroes if evaluated with [”red”, “red”, “red”]

One consequence of this is that any tool that operates on your data afterwards and returns column indexes will return numbers that are not trivially mapped to your features. For example, some scikit-learn algorithms provide some analysis of the matrix telling you which columns are best correlated with some property. If you use those you will get a result like “columns 37 and 42 have high correlation”, but you probably want to know the name of the features which are related to columns 37 and 42. To do so, the vectorizer provides a *column_to_feature(i)* method, which takes a column number and returns a tuple (feature, info), where feature is the original feature (remember that you can get a better description printing f.name when f is a feature). The second field in the result tuple is *None* for numeric features, the label for enumerated or bag features (i.e., “animal”, “vegetable” or “mineral”) and the index for vector features.

4.3 Sparse vs Dense Matrices

By default, Vectorizer will construct a sparse numpy matrix which in the general case will consume significanly less memory. Anyway, by passing *sparse=False* as an argument when instantiating *Vectorizer* you can change this to use a dense matrix instead.

4.4 Tolerant evaluation

If your data is not completely clean or there are bugs in the implementation of your features, it is possible that during vectorization an exception will be raised, which stops the process. This is specially annoying if this happens after a long run time (computing features can take a lot of time for big datasets and or complex features) just because a silly bug that is triggered in a single broken or unusual datapoint among millions of them.

For many experiments, the result you get is useful even if you drop some samples, or even if you ignore a few buggy features while letting the other ones be evaluated.

FetureForge provides a “tolerant” version of the vectorizer that can be used by passing *tolerant=True* as an argument when instantiating *Vectorizer*. When in tolerant mode, the following things happen:

- Data points are always discarded when failing: If a given sample fails when evaluating a feature with it, no matter what, no matter when, the data point is discarded.
- If a feature evaluation fails in the first 100 samples, it’s assumed to be broken and it’s excluded.
- If a feature evaluation fails more than 5 times, it’s assumed to be broken and it’s excluded.
- When a feature is excluded, samples discarded because of that feature are re-checked.

Right now, the configuration values for the policy are hardcoded.

Note that the process described above can result on a matrix that is missing some rows (data points) and some columns (features).

CHAPTER 5

Experimentation Support

5.1 Key concepts

As a Machine Learning developer, we face a lot of times the need of doing some fine tuning of some feature, some classifier, regressor, etc. And when the theory was not enough, we just experiment. Running the same script several times, each with small differences on the arguments, is not hard. But when it comes to be able to later decide which of the executions provided the best performance/accuracy/whatever you want to measure, it's not that easy. And to be honest, it's a bit boring to have to be writing down each experiment result (and it's initial configuration) by hand.

5.2 The basics

Feature Forge **experimentation** module can help you to automate the execution of your experiments, storing the results of each one on a database that can be queried later for decision taking.

In its most basic form, on an empty file (let's name it *my_experiments.py*) you just define an experiment as a function

```
def train_and_evaluate_classifier(config_dict):
    # creates a classifier based on config_dict,
    # later trains it with some fixed train data and
    # evaluates it with some other fixed test data
    result = do_stuff(...)
    return {
        'accuracy': ...,
        'elapsed_time': ...,
        'other_metric': ...
    }
```

Your function should take a single argument (which is a dictionary that defines the configuration of the experiment) and return another dictionary with the results of the experiment.

After that, add a few lines wrapping your function like this:

```
from featureforge.experimentation import runner


def train_and_evaluate_classifier(config_dict):
    ... # your function


if __name__ == '__main__':
    runner.main(train_and_evaluate_classifier)
```

And that's it, you have a ready to use experiments runner, that will:

- accept from the command line a JSON file containing a sequence of dicts (each of those will be passed to your function as an experiment configuration)
- log the results on a MongoDB (you need to provide URI and database name)

For more details, just run:

```
$ python my_experiments.py -h
```

5.3 Parallelism

The experiment runner provides a simple but effective support for running several experiments in parallel, by simply running the same script several times. Each time that an experiment is about to be started, the runner attempts to book it on the database. If it was already booked, then that experiment will be ignored by this runner, and the next configuration will be attempted.

In this way, you can run this script as many times as desired, even from different computers, all of them booking and saving experiment results to a shared database.

Tips:

- Monitor the memory usage of each experiment. Running several in parallel may use all the memory available, slowing down the entire experimentation.
- Bookings are not forever. By default they last 10 minutes, but you can set it to whatever you want. Once that an experiment booking expires, it may be booked again and re-run by anyone.

5.4 Dynamic experiment configuration

In the general case, among the static experiment configuration, it's very important to also provide some info about the context in which the experiment ran.

So, following our example of experimenting to get the best classifier, besides having the classifier name and it's parameters as part of a given experiment, like this

```
single_experiment_config = {
    "regression_method": "dtree",
    "regression_method_configuration": {
        "min_samples_split": 25
        }
    }
```

it's equally important to be sure that all experiments were run with a the same version of code, or that the data-sets used for training and testing are always the same, etc. If you have more than one evaluation data set it's important to be able to find out which results correspond to each data set.

Because of that need, we highly recommend you to define a *configuration extender*, like this

```
from featureforge.experimentation import runner

def train_and_evaluate_classifier(config_dict):
    ... # your function

def extender(config):
    """
```

```
    Receives a copy of the the experiment configuration before
    attempting to book, and returns it modified with the extra
    details desired.
    """
    # whatever you want, for instance:
    config['train_data_hash'] = your_md5_function('train', ...)
    config['test_data_hash'] = your_md5_function('test', ...)
    config['code_version'] = your_definition_of_current_version(...)
    ...
    return config

if __name__ == '__main__':
    runner.main(train_and_evaluate_classifier, extender)
```

We provide a built-in utility for using the current git branch (and modifications) as part of the configuration:

```
if __name__ == '__main__':
    runner.main(
        train_and_evaluate_classifier,
        use_git_info_from_path='/path/to/my_repo/')
```

For other version control systems, or any other things you may need, use the *extender* callback.

5.5 Exploring the finished experiments

Once you run all the experiments, you will have everything stored on the MongoDB. For each experiment, it's configuration and results will be stored on a single Document, like this:

- Field "*marshalled_key*": string text representing the hashed experiment configuration. Used as identifier for bookings.
- **Field "*experiment_status*": one of the following**
  - "*status_booked*": experiment was booked but not finished yet.
  - "*status_solved*": experiment was reported as finished.
- Field "*booked_at*": time-stamp of the experiment booking.
- Field "*results*": only available for finished experiments. It's a dictionary that contain as sub-fields all the results of the experiment.
- Any other field on the Document, was part of the experiment configuration.

You can access, filter and see the finished experiments simply using the mongo shell, or with python like this:

```
from featureforge.experimentation.stats_manager import StatsManager

sm = StatsManager(None, 'Your-database-name')

for experiment in sm.iter_results():
    print(experiment.results)
```

5.6 Important Notes and Details

5.6.1 Configuration dicts

- Simple data types:

  In order to easily create booking-tickets from configuration dictionaries, they can't contain more than built-in objects (sets, lists, tuples, strings, booleans or numbers).

- Lists, tuples or sets, be careful with the ordering:

  Be very careful with config value that are sequences. If your experiment configuration needs to provide the features to use, probably their ordering is not important, so you should pass them as a *set*, and not as a tuple or a list. Otherwise, these 2 configurations are going to be treated as different when they shouldn't:

```
config_a = {
    "regression_method": "dtree",
    "regression_method_configuration": {
        "min_samples_split": 25
        },
    "features": ["FeatureA", "FeatureB", "FeatureC"]
    }

config_a_again = {
    "regression_method": "dtree",
    "regression_method_configuration": {
        "min_samples_split": 25
        },
    "features": ["FeatureC", "FeatureA", "FeatureB"]
    }
```

CHAPTER 6

Indices and tables

- search