1400 days ago

Creating your own estimator in scikit-learn

I had an interesting problem in my work and I finally had to get to something I'd been thinking for some time now.

My problem consists of using Recurrent Neural Networks (which were implemented in Lua here), to which I had to input some text files preprocessed by Python. It produces output to stdout, which was then parsed and preprocessed again by Python as an input to another Python package, which again produces some output files, which were again parsed by Python :D . That's what I called fun. Of course, all parts of processes had some parameters which needed to be tuned by running it on the validation set and then tested on testing set. It seems unbearable to do it manually. Hence, I decided to create my own estimator using `scikit-learn` and then use `Pipeline` and `GridSearchCV` for automatizing whole process and parameter tuning.

In this little example I will just give summary and an example of creating your own estimator. It is based on informations on this site: Rolling your own estimator (scikit-learn docs) .

The example there is not very representative and so I will try to come out with something more clear.

You should be acknowledged what the objects are (that what you define with class keyword).

Building an object

Decide what type of object you want to create. You need to choose one of these: *Classifier*, *Clusterring*, *Regressor* and *Transformer*. The classifier is self-explanatory -- you give some input X and get the class of which it probably belongs (e.g. Naive Bayes Classifier). An example of Regressor is e.g. Linear Regression. You give it some input X and get estimations of variable Y. Then there is clusterring, where I'm not going into more details now. The last one, Transformer, is for transforming the data -- it takes X and returns changed X. An example of this might be PCA.

After you decided which one suits to your needs you subclass `BaseEstimator` **and** an appropriate class for your type (one of `ClassifierMixin, RegressorMixin, ClusterMixin, TransformerMixin`).

Now you need of course to decide what parameters an estimator receives. Decide what input it takes and what output it returns. It's all you need for now.

Abiding scikit-learn rules

It's good to know some additional rules.

- All arguments of `__init__` must have default value, so it's possible to initialize the classifier just by typing `MyClassifier()`
- No confirmation of input parameters should be in `__init__` method! That belongs to fit method.
- All arguments of `__init__` method should have the same name as they will have as the attributes of created object (see Simplification part about automatizing this).
- **Do not** take data as argument here! It should be in fit method.

Before building your estimator, you should abide following rules.

get_params and set_params

All estimators must have `get_params` and `set_params` functions. They are inherited when you subclass `BaseEstimator` and I would recommend not to override these function (just not state them in definition of your classifier).

Fit method

In fit method you should implement all the hard work. At first you should check the parameters. Secondly, you should take and process the data.

You'll almost surely want add some new attributes to your object which are created in fit method. These

should be ended by `_` at the end, e.g. `self.fitted_`.

And finally you should return `self`. This is again for compatibility reasons with common interface of scikit-learn.

Response y vector

There might be cases when you do not need to input a response vector (as in example below). Nevertheless, for implementation reasons you need to add this vector to your definitions in case you want to use `GridSearch` and so. It's good to initialize it with `None` value.

Additional requirements, score and GridSearch

As usually, everything which should be hidden to user should start with `_`.

For correct function of GridSearch, it is usually necessary to override `score` method to your needs. Why? GridSearch needs to recognize if given model is better or not. And he does it very simply by calling `score` method and then use rule **bigger is better**. Hence, whatever crazy scoring function you have, you basically must represented in some metric in numbers.

An example of MeanClassifier

Let's say that we'll want to create a classifier which will classify an input to two classes: to class 0 if a value is

smaller than some number and 1 if the value is bigger or equal than some number. What about the "some number"? Let's say that it will be the mean + intValue of exactly 20 input values in a list. Our classifier will hence take one vector X (of length 5) and every record will be assigned 1 or 0 (resp. True or False). Furthermore it is somehow unsupervised classifier -- we do input a response `y` vector.

Additionally, let's say we want to find a configuration of `intValue` with given input `X` in which there will be the biggest number of `True` values (which in our case means bigger then `mean + intValue`, so basically we want `intValue` as low as possible, but forget about that for now :D).

Here is the code:

```
from sklearn.base import BaseEstimator, ClassifierMixin

class MeanClassifier(BaseEstimator, ClassifierMixin):
    """An example of classifier"""

    def __init__(self, intValue=0, stringParam="defaultValue", o
        """
        Called when initializing the classifier
        """
        self.intValue = intValue
        self.stringParam = stringParam

        # THIS IS WRONG! Parameters should have same name as att
        self.differentParam = otherParam


    def fit(self, X, y=None):
        """
        This should fit classifier. All the "work" should be don
```

```
        Note: assert is not a good choice here and you should ra
        use try/except blog with exceptions. This is just for sh
        """

        assert (type(self.intValue) == int), "intValue parameter
        assert (type(self.stringParam) == str), "stringValue para
        assert (len(X) == 20), "X must be list with numerical va

        self.treshold_ = (sum(X)/len(X)) + self.intValue  # mean

        return self

    def _meaning(self, x):
        # returns True/False according to fitted classifier
        # notice underscore on the beginning
        return( True if x >= self.treshold_ else False )

    def predict(self, X, y=None):
        try:
            getattr(self, "treshold_")
        except AttributeError:
            raise RuntimeError("You must train classifer before p

        return([self._meaning(x) for x in X])

    def score(self, X, y=None):
        # counts number of values bigger than mean
        return(sum(self.predict(X)))
```

And now you should be able to normally use `GridSearch`:

```
from sklearn.grid_search import GridSearchCV

X_train = [i for i in range(0, 100, 5)]
X_test = [i + 3 for i in range(-5, 95, 5)]
tuned_params = {"intValue" : [-10,-1,0,1,10]}

gs = GridSearchCV(MeanClassifier(), tuned_params)

# for some reason I have to pass y with same shape
# otherwise gridsearch throws an error. Not sure why.
```

```
gs.fit(X_test, y=[1 for i in range(20)])

gs.best_params_ # {'intValue': -10} # and that is what we expect
```

Simplification of init method

The initialization of init method becomes very boring when having a lot of attributes, since you will see the following a lot:

```
def __init__(self, arg1, arg2, arg3, ..., argN):
    self.arg1 = arg1
    self.arg2 = arg2
    .
    .
    .
    self.argN = argN
```

this task can be made automatic by using `inspect` module and `setattr`. Import `inspect` by `import inspect` and then you can use:

```
def __init__(self, arg1, arg2, arg3, ..., argN):

    # print("Initializing classifier:\n")

    args, _, _, values = inspect.getargvalues(inspect.currentfra
    values.pop("self")

    for arg, val in values.items():
        setattr(self, arg, val)
        # print("{} = {}".format(arg,val)
```