splearn: a python machine learning package

Caihua Wang
Email: 490419716@qq.com

2014/12/12

1 splearn overview

In general, a learning problem considers a set of n samples of data and then tries to predict properties of unknown data. We can separate learning problems in a few large categories:

- **supervised learning**, in which the data comes with additional attributes that we want to predict. This problem can be either:
  - **classification**: samples belong to two or more classes and we want to learn from already labeled data how to predict the class of unlabeled data. Another way to think of classification is as a discrete (as opposed to continuous) form of supervised learning where one has a limited number of categories and for each of the n samples provided, one is to try to label them with the correct category or class.
  - **regression**: if the desired output consists of one or more continuous variables, then the task is called regression.
- **unsupervised learning**, in which the training data consists of a set of input vectors x without any corresponding target values. The goal in such problems may be to discover groups of similar examples within the data, where it is called clustering, or to determine the distribution of data within the input space, known as density estimation, or to project the data from a high-dimensional space down to two or three dimensions for the purpose of visualization.

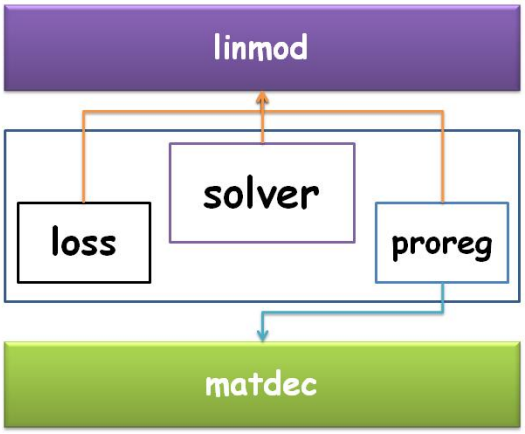

Figure 1: The structure of splearn

Recently, a lot of machine learning package for python (such as, sklearn, http://scikit-learn.sourceforge.net/stable/index.html) have been developed, and the classes and functions in the packages are easy to use. However, the existing package ignore some important aspects:

- Existing package ignoring of some important **loss functions**.
- Existing package ignoring of some important **regularization items**.
- Some methods in existing package are **not efficient**.

splearn is a limited machine learning package, which offers sparse linear classifiers and regressors in supervised learning and provides sparse matrix decomposition algorithms in unsupervised learning. **splearn** considers more loss functions (Table 1) and regularization items (Table 2) than other python machine learning packages, and the algorithms in **splearn** are neither as efficient as state-of-the-art or comparable to the state-of-the-art. In a word, **splearn** is an efficient python machine learning package specialized in sparse linear supervised models (classification and regression) and in sparse matrix decomposition.

1.1 The structure of splearn

splearn is well organized, the structure show in Figure 1.

Table 1: The loss functions

Type	Name	t	L(t)
Regression	squarederror	$y - f(x)$	$L(t) = \frac{1}{2}t^2$
Regression	huberloss	$y - f(x)$	$L(t) = \begin{cases} \frac{1}{2}t^2, & \lvert t\rvert \le \delta \\ \delta(\lvert t\rvert - \frac{1}{2}\delta), & \lvert t\rvert > \delta \end{cases}$
Classification	logit	$yf(x)$	$L(t) = \log(1 + \exp(-t))$
Classification	modifiedhuberloss	$yf(x)$	$L(t) = \begin{cases} 0 & t > 1 \\ (t-1)^2 & -1 \le t \le 1 \\ -4t & t < -1 \end{cases}$
Classification	huberizedhinge	$yf(x)$	$L(t) = \begin{cases} 0 & t > 1 \\ \frac{1}{2}(1-t)^2/\delta & 1-\delta \le t \le 1 \\ 1 - t - \frac{1}{2}\delta & t < 1-\delta \end{cases}$
Classification	squaredhinge	$yf(x)$	$L(t) = \begin{cases} 0 & t > 1 \\ \frac{1}{2}(1-t)^2 & t \le 1 \end{cases}$

Table 2: Regularization

Regularization	$\Omega(x)$
lasso	$\lambda\|x\|_1$
elastic net	$\lambda_2\|x\|_2^2 + \lambda_1\|x\|_1$
fused lasso	$\lambda_2 \sum_{i=2} \|x_i - x_{i-1}\| + \lambda_1\|x\|_1$
sparseness	$(\sqrt{n} - \|x\|_1/\|x\|_2)/(\sqrt{n}-1) \leq \lambda$
group lasso	$\lambda \sum_k \|x_k\|_2$
sparse group lasso	$\lambda_2 \sum_k \|x_k\|_2 + \lambda_1\|x\|_1$

loss, **proreg**, **solver** are three sub-package that transparent to users. The sub-package **loss** provide loss functions (Table 1) in group and non-group fashion. The sub-package **proreg** provides various algorithms for projection subproblem.

$$\min : \frac{1}{2}\|x - v\|_2^2 + \Omega(x) \tag{1}$$

where $\Omega(x)$ are shown in Table 2. The sub-package **solver** combines the two and offers fundamental algorithms (such as FISTA, CD, GMD, LARS) to train a model.

linmdel (linear model) and **matdec** (matrix decomposition) offer APIs accessible to users. **linmdel** provides two classes for group sparse linear model (GSLM) and non-group sparse linear model (NGSLM) respectively. **matdec** offers four functions: ssvd, spca, spcoding and snmf, for matrix decomposition.

```
>>> import splearn as sp
>>> dir(sp)
['GSLM', 'NGSLM', '__builtins__', '__doc__', '__file__',
'__name__', '__package__', '__path__', 'accmse',
'cwd_elasticnet', 'cwd_lasso', 'lars', 'lars_path',
'linmod', 'loadata', 'loss', 'matdec', 'plotpath', 'proreg',
'snmf', 'solver', 'spca', 'spcoding', 'ssvd', 'utils']
```

1.2 Install splearn and getting help

1.2.1 Install splearn

Installing **splearn** is easy. Just unzip the downloaded package and change the working directory to the unzipped package directory, then run python as

follows:

```
>>> python setup.py install
>>> python
>>> import splearn as sp
>>>
```

If the package installed successfully, there would be no error or warning.

1.2.2 Getting help

If the package is installed successfully, getting help would be as conveniens as other packages. For example,

```
>>> import splearn as sp
>>> help(sp.snmf)
Help on function snmf in module splearn.matdec.SNMF:

snmf(X, k, alpha, nsubiter=30, eps=0.0001)
    sparse non-nrgative matrix factorization (snmf)
    Parameters
    ----------

    X : 2d numpy array, the input matrix
    k : int, the component you want to get
    alpha : float, the regularization factor, must
            be positive

    Example
    ----------

    >> from splearn import snmf
    >> from numpy.random import uniform
    >> import numpy as np
    >> X = uniform(0, 100, size=(125, 50))
    >> B, H = snmf(X, k=10, alpha=0.2)
    >> print X - np.dot(B, H)
    >>

>>>
```

2 splearn APIs

In this section, we give the detailed information (function description and examples) for user assessable APIs in splearn.

2.1 NGSLM

NGSLM is a class for nog-group sparse linear model. NGSLM is easy to use and the detailed information as follows:

```
NGSLM(loss,delta,regu,reargs,fitintercept=True,alpha=0.05)
 |
 |  Parameters
 |  ----------
 |  loss : string
 |         specify loss function by a string. loss must be
 |         one of the following:
 |              'squarederror'    #Regression
 |              'huberloss'       #Regression
 |              'logit'                #Classification
 |              'modifiedhuberloss'    #Classification
 |              'huberizedhinge'       #Classification
 |              'squaredhinge'         #Classification
 |
 |  delta : float | None
 |          the parameter in loss function.
 |          'huberloss' : delta in [0, inf], usually not very
 |                        big, say smaller than 10.
 |          'huberizedhinge' : delta in (0, 1), usually close
 |                             to 1, say 0.8.
 |          For other loss functions, delta is not used, so
 |          pls set 'delta = None'.
 |
 |  regu : string
 |         specify regularization by a string. regu must be
 |         one of the following:
 |              'lasso', 'elasticnet', 'fusedlasso'
 |
 |  reargs : dictionary
 |          specify supper parameters for different
 |          regularizations,
 |          'lasso': reargs = {'lam': val}
 |          'elasticnet', 'fusedlasso': reargs = {'lam1':
                              val1, 'lam2': val2}
```

```
|
|   fitintercept : bool
|           whether fit intercept.
|
|   alpha : float
|           if fit intercept, the parameter alpha is the
|           learning rate for fitting intercept, so alpha
|           should not too large, alpha in [0.01, 1.0].
|
|   Attributes
|   ----------
|   `coeff_` : the model parameter, a 1d numpy array.
|   `intercept_` : intercept, a float or None.
|
|   Methods defined here:
|
|   __init__(self, loss='squarederror', delta=None,
|             regu='lasso', reargs=None, fitintercept=True,
|             alpha=0.05)
|
|   fit(self, X, y, weight=None, theta=None)
|       The training function.
|       X : 2d numpy array, the input predictors
|       y : 1d numpy array, the labels
|       weight: the aeight of samples
|       theta: the inital parameters
|
|   predict(self, X)
|       X : 2d numpy array, the input predictors
|
|   predict_proba(self, X)
|       X : 2d numpy array, the input predictors
```

NGSLM has three methods: **fit**, **predict** and **predict_proba**. **fit** is used to train a model, **predict** is used to predict the labels of given X, and **predict_proba** is used to predict probability (only for classification). Here is an example:

```
>> from splearn import NGSLM
>> from numpy.random import uniform, randn
>> import numpy as np
>>
>> X = uniform(-10, 10, size=(500, 200))
>> theta = uniform(-10, 10, size=200)
>> y = np.dot(X, theta) + randn(200)
```

```
>>
>> reargs = {'lam': 100}
>> lm = NGSLM(loss='squarederror', delta=None, regu='lasso',
             reargs=reargs)
>>
>> lm.fit(X, y)
>> py = lm.predict(X)
>> print lm.coeff_
>> print lm.intercept_
>>
```

2.2 GSLM

GSLM is a class for group sparse linear model. GSLM is similar to NGSLM but in group fashion, the detailed information as follows:

```
GSLM(loss, delta, regu, reargs, grouplist, groupweight,
 |    fitintercept=True, alpha=0.05)
 |
 |  Parameters
 |  ----------
 |  loss : string
 |         specify loss function by a string. loss must be
 |         one of the following:
 |              'squarederror'     #Regression
 |              'huberloss'        #Regression
 |              'logit'                 #Classification
 |              'modifiedhuberloss'     #Classification
 |              'huberizedhinge'        #Classification
 |              'squaredhinge'          #Classification
 |
 |  delta : float | None
 |          the parameter in loss function.
 |          'huberloss' : delta in [0, inf], usually not very
 |                        big, say smaller than 10.
 |          'huberizedhinge' : delta in (0, 1), usually close
 |                             to 1, say 0.8.
 |          For other loss functions, delta is not used, so
 |          pls set 'delta = None'.
 |
 |  regu : string
 |         specify regularization by a string. regu must be
 |         one of the following:
 |              'grouplasso', 'sparsegrouplasso'
```

```
|
|   reargs : dictionary
|           specify supper parameters for different
|           regularization,
|           'grouplasso': reargs = {'lam': val}
|           'sparsegrouplasso': reargs = {'lam1': val1,
|                                         'lam2': val2}
|
|   grouplist : list of tuple (cannot be None)
|           grouplist = [(0, 5), (5, 20), (20, 30), (30, 45)]
|           start with 0, end with the number of predictors.
|           the end point of last group must identity with
|           the next group's start point.
|
|   groupweight : list of float | None
|           if groupweight is not None, then the length of
|           groupweight must equal to the length of grouplist
|                                 .
|
|
|   fitintercept : bool
|           whether fit intercept.
|
|   alpha : float
|           if fit intercept, the parameter alpha is the
|           learning rate for fitting intercept, so alpha
|           should not too large, alpha in [0.01, 1.0].
|
|   Attributes
|   ----------
|   `coeff_` : the model parameter, a 1d numpy array.
|   `intercept_` : intercept, a float or None.
|
|   Methods defined here:
|
|   __init__(self, loss, delta, regu, reargs, grouplist,
|            groupweight, fitintercept=True, alpha=0.01)
|
|   fit(self, X, y, weight=None, theta=None)
|       The training function.
|       X : 2d numpy array, the input predictors
|       y : 1d numpy array, the labels
|       weight: the aeight of samples
|       theta: the inital parameters
|
```

```
|  predict(self, X)
|      X : 2d numpy array, the input predictors
|
|  predict_proba(self, X)
|      X : 2d numpy array, the input predictors
```

GSLM also has three methods: **fit**, **predict** and **predict_proba**. **fit** is used to train a model, **predict** is used to predict the labels of given X, and **predict_proba** is used to predict probability (only for classification). Here is an example:

```
>> from splearn import GSLM
>> from numpy.random import uniform, randn
>> import numpy as np
>>
>> X = uniform(-10, 10, size=(500, 200))
>> theta = uniform(-10, 10, size=200)
>> y = np.dot(X, theta) + randn(200)
>>
>> reargs = {'lam': 4}
>> grouplist = [(0, 20), (20, 50), (50, 80), (80, 120),
                (120, 155), (155, 187), (187, 200)]
>>
>> lm = GSLM(loss='squarederror', delta=None, regu='lasso',
             reargs=reargs,grouplist=grouplist,
             groupweight=None)
>>
>> lm.fit(X, y)
>> py = lm.predict(X)
>> print lm.coeff_
>> print lm.intercept_
>>
```

2.3 matdec

We describe the APIs in **matdec** in this subsection. **matdec** contains four functions: ssvd, spca, spcoding and snmf.

2.3.1 ssvd

Sparse singular value decomposition (ssvd) is a common algorithm used in wide area. We introduce various sparse criterion (such as lasso, elastic net,

fused lasso and sparseness) for ssvd in **splearn**, and the algorithms are efficient. Here is the detailed information and example:

```
ssvd(X, k, pfname1, args1, pfname2, args2)
    Parameters
    ----------

    X : 2d numpy array, the input matrix
    k : int, the number of svd laters
    pfname*, args*:
            pfname* : string, specify the regularization item
                      'lasso', 'elasticnet', 'fusedlasso' and
                      'sparseness'
            args* : dictionary, specify the regularization
                    supper parameters,
                    'lasso': args = {'lam': val}, val > 0
                    'sparseness' : args = {'lam': val},
                                   val in (0, 1)
                    'elasticnet', 'fusedlasso':
                           args = {'lam1': val1, 'lam2':val2}
            * : 1, 2.
                    1 --> U's row
                    2 --> V's row

    Example
    ----------

    >> from splearn import ssvd
    >> from numpy.random import uniform
    >>
    >> X = uniform(-10, 10, size=(50, 80))
    >> args1 = {'lam':0.5}
    >> args2 = {'lam':0.7}
    >> U, D, V = ssvd(X, 10,
                  pfname1='sparseness', args1=args1,
                  pfname2='sparseness', args2=args2)
    >>
```

2.3.2 spcoding

Sparse coding (spcoding) is very similar to **ssvd**. **ssvd** return three matrices, while **spcoding** return two. The first matrix returned by spcoding is unit norm in columns. Here is the detailed information and example:

```
spcoding(X, k, pfname1, args1, pfname2, args2)
    Parameters
    ----------

    X : 2d numpy array, the input matrix
    k : int, the number of svd laters
    pfname*, args*:
            pfname* : string, specify the regularization item
                      'lasso', 'elasticnet', 'fusedlasso' and
                      'sparseness'
            args* : dictionary, specify the regularization
                    supper parameters,
                    'lasso': args = {'lam': val}, val > 0
                    'sparseness' : args = {'lam': val},
                                   val in (0, 1)
                    'elasticnet', 'fusedlasso':
                           args = {'lam1': val1, 'lam2':val2}
            * : 1, 2.
                    1 --> U's row
                    2 --> V's row

    Example
    ----------

    >> from splearn import spcoding
    >> from numpy.random import uniform
    >>
    >> X = uniform(-10, 10, size=(50, 80))
    >> args1 = {'lam':0.5}
    >> args2 = {'lam':0.7}
    >> U, D, V = spcoding(X, 10,
                  pfname1='sparseness', args1=args1,
                  pfname2='sparseness', args2=args2)
    >>
```

2.3.3 spca

Sparse principal component analysis (spca) is very efficient comparing to the one in **sklearn**. We also provide different kinds of regularization items for spca. Note: the project directions for ordinary PCA should be orthogonality, but we do not guarantee orthogonality in **spca**. However, we have make sure that the project directions do not correlate too much. Here is the detailed

information and example:

```
spca(X, k, pfname, args)
    Parameters
    ----------

    X : 2d numpy array, the input matrix
    k : int, the number of svd laters
    pfname, args:
            pfname : string, specify the regularization item
                     'lasso', 'elasticnet', 'fusedlasso' and
                     'sparseness'
            args : dictionary, specify the regularization
                                          supper parameters,
                     'lasso': args = {'lam': val}, val > 0
                     'sparseness' : args = {'lam': val}
                                    val in (0, 1)
                     'elasticnet', 'fusedlasso':
                          args = {'lam1': val1, 'lam2':val2}

    Example
    ----------

    >> from splearn import spca
    >> from numpy.random import uniform
    >>
    >> X = uniform(-10, 10, size=(50, 80))
    >> args1 = {'lam':0.5}
    >> args2 = {'lam':0.7}
    >> U, V = spca(X, 10,
                   pfname1='sparseness', args1=args1,
                   pfname2='sparseness', args2=args2)
    >>
```

2.3.4 snmf

Sparse non-negative matric factorization (snmf) is a widely used algorithm in image process and bioinformatics. We adopt the SVD to initial the parameters **B** and **H** in **snmf**, which was proved to be efficient. What's more, we also modified and improved the multiply rule (by introducing the update rule in FISTA) in original **nmf** algorithm. As a result, **snmf** in **splearn** is very efficient. Here is the detailed information and example:

```
snmf(X, k, alpha, nsubiter=30, eps=0.0001)
    Parameters
    ----------

    X : 2d numpy array, the input matrix
    k : int, the component you want to get
    alpha : float, the regularization factor, must be
            positive

    Example
    ----------

    >> from splearn import snmf
    >> from numpy.random import uniform
    >> import numpy as np
    >> X = uniform(0, 100, size=(125, 50))
    >> B, H = snmf(X, k=10, alpha=0.2)
    >> print X - np.dot(B, H)
    >>
```