tutorial in jupyter notebook

July 8, 2020

Tutorial for using autoBioSeqpy as modules in script

Using the autoBioSeqpy via command line is a good way for data modeling, but usually users would like to use autoBioSeqpy for something more, such as converting the FASTA data into matrix or combine the modeling to another workflow.

This notebook provided the usage for using autoBioSeqpy as a library which could be imported normally. Moreover, few alternatives provided for some special case. We hope this tutorial could help user for understanding this tool deeply.

This notebook is available in jupyter notebook (editing and running is possible), but if you didn't install jupyter notebook, two copies in PDF and HTML version are available as well (read only).

1. First step: Initializing

As the first step, initializing is necessary for autoBioSeqpy by setting the search path.

The method'import' in python is a normal way for using a module in a default search path, but this time autoBioSeqpy didn't provide a way for 'install' and thus the search path is necessary for providing.

If autoBioSeqpy is already in your search path, you can import it directly.

1.1 Initializing the search path

There are several ways for importing the self-made modules/libraries. Considering that user might have different environment, please change the variable 'libPath' into the path where the tool located.

An alternative is provided, and available if uncomment it.

[1]:
```python
import os, sys
libPath = '../' #please change it into your search path if necessary
sys.path.append(libPath)
#alternative
#os.chdir(libPath)
```

[2]:
```python
import numpy as np #for some analysis
```

2. Data processing
   Usually, users will write a script to converting the FASTA into matrix, here autoBioSeqpy provided few ways for the matrix creation.

2.1 import the related library

Library 'dataProcess' is provided for matrix creation form FASTA data. The necessary function such as suffle the samples and spliting the dataset into pieces (for cross validation) are provided as well.

Since the location is added into the search path in section 1.1, here we only need to import it as a module.

```
[3]: import dataProcess
```

2.2 Usage of module dataProcess

To load a dataset, first we need to instantiate an object and then using loading.

When intorducing it, some cases and parameters will be explained as well.

2.2.1 A detailed description for training data

dataType

Since Protein, DNA and RNA have their FASTA, we have to decide the type of this data. In our standalone script, 'dataType' is a parameter, but here we don't have to determine it as a parameter directly.

dataEncodingType

This is a parameter for set the way to encoding the FASTA into matrix. Currently there are two encoding types available: 'onehot' and 'dict'. If 'dict' choosed, a character (e.g. A/G/C/T for DNA) is represented as a number (such as A:1 T:2 C:3 T:4).

Alternetivly, if choose 'onehot',a character will be represented as an array (such as A:[1,0,0,0] G:[0,1,0,0] C:[0,0,1,0] T[0,0,0,1]).

In this example, only 'dict' is used since it is better for displaying.

```
[4]: dataEncodingType = 'dict'
```

useKMer and KMerNum

Usually we would like to consider taking not only one FASTA residue for encoding, but also its neighbors. The parameter 'useKMer' is an implementation for the environment encoding.

For example, if a sequence is ATTACT, and 'KMerNum' is 3, then the first A will be considered as 'ATT'.

Note that the shape of dataset will be expanded accordingly (see the manual for more details). And usually the 'useKMer' is used when 'dataEncodingType' set as 'oneHot'. And thus in this notebook, we don't use KMer since the encoding type is 'dict'.

If you are interesting for the KMer, please change the 'dataEncodingType' as 'onehot' and turn on the 'useKMer' by set it into True.

```
[5]: useKMer = True
     KMerNum = 3 #If useKMer is False, the KMerNum is inactive, and thus it doesn't␣
      ↪matter how much the value is.
```

2.2.1.1 featureGenerator: the encoder

Now we can initialize a featureGenerator. A featureGenerator is a class for encoding the FASTA sequence.

There are three featureGenerator available: ProteinFeatureGenerator, DNAFeatureGenerator and RNAFeatureGenerator, you could use one of them according to the datatype.

In this notebook, protein data is used, thus we use ProteinFeatureGenerator as the featureGenerator.

[6]:
```
featureGenerator = dataProcess.ProteinFeatureGenerator(dataEncodingType,␣
 ↪useKMer=useKMer, KMerNum=KMerNum)
#featureGenerator = dataProcess.DNAFeatureGenerator(dataEncodingType,␣
 ↪useKMer=useKMer, KMerNum=KMerNum) #alternative for DNA data
#featureGenerator = dataProcess.RNAFeatureGenerator(dataEncodingType,␣
 ↪useKMer=useKMer, KMerNum=KMerNum) #alternative for RNA data
```

2.2.1.2 File format and class DataLoader

With the encoder, now it's possible to read the FASTA data and encode them into matrix.

autoBioSeqpy provided a class 'DataLoader' for handle all the file reading things. So here we need to introduce the format of the FASTA file.

File format

The so called 'file format' is the normal FASTA format. That is, a sequence is started by '> name&information' and then few lines of FASTA characters followed. There is no limitation of the number of characters in a line, you can try few lines with not more than 60 characters or only 1 line with all characters. For example, the both formats are supported and can even mixed in one file: >case1 only 1 line XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX… >case2 few lines XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX … XXXXXXXXXXXXXX

label

The only thing should be concerned is the label of a file. Usually there are two (e.g. 1/-1 or 1/0) or more (e.g. case1, case2, case3, …) labels for a dataset, but the class DataLoader can handle one dataset with the same label, which means if a data has 3 labels, at least 3 files is necessary for reading.

In this notebook, the provided data is a binary classification, and therefore only two labels, 1 for positive samples and 0 negative samples, are used.

spcLen

Here is another problem: usually the length is not the same for different sequences.

To address it, the 'spcLen' is provided. If the length of an input sequence is larger than the 'spcLen', the exceed part will be ignored, and if the length is less than 'spcLen', zeros (or zero arrays) will be added to make the length up to spcLen.

[7]:
```
spcLen = 100
```

DataLoader

The class 'DataLoader' is a class for loading a file, thus usually we need at least two DataLoader for different label.

In this notebook, two files, 'train_pos.txt' and 'train_neg.txt', provided for training dataset, which can be found in 'data' folder. The labels are '1' and '0' respectively.

[8]:
```
#the paths
dataTrainFilePaths = ['../examples/typeIII_secreted_effectors_prediction/data/
 ↪train_pos.txt','../examples/typeIII_secreted_effectors_prediction/data/
 ↪train_neg.txt']
#the related labels
dataTrainLabel = [1, 0]
#a list for recording the DataLoader
trainDataLoaders = []
for i,dataPath in enumerate(dataTrainFilePaths):
    #init
    dataLoader = dataProcess.DataLoader(label = dataTrainLabel[i],␣
 ↪featureGenerator=featureGenerator)
    #file read
    dataLoader.readFile(dataPath, spcLen = spcLen)
    trainDataLoaders.append(dataLoader)
```

2.2.1.3 DataSetCreator: merge different dataLoader

After FASTA loading and encoding, now we can generate the matrix by merging the dataLoaders. The class 'DataSetCreator' is provided for the matrix merging and the necessary functions, such as sample shuffle and dataset split, are provided.

NOTE: Since the DataSetCreator is able to merge different DataLoader no matter whether the label is the same or not, thus if you have multiple files with the same label, you don't have to merge them by hand, just merger them here.

[9]:
```
#init
trainDataSetCreator = dataProcess.DataSetCreator(trainDataLoaders)
```

Then we can generate the matrix by using the method 'DataSetCreator' if the test dataset is in other files.

The parameter 'toSuffle' is a switch to s

[10]:
```
#get dataset
trainDataMat, trainLabelArr = trainDataSetCreator.getDataSet(toShuffle=True)
```

We can have a look of the matrix and labels, all of them are numpy array.

[11]:
```
print('Matrix with shape %d x %d:' %(trainDataMat.shape[0],trainDataMat.
 ↪shape[1]))
print(trainDataMat)
```

```python
print('\n')
print('The labels with length %d:' %(len(trainLabelArr)))
print(trainLabelArr)
```

```
Matrix with shape 907 x 98:
[[ 8583 12259  2369 ...  7103  8918  3389]
 [ 8557 11579  2271 ...  1042  9518  1413]
 [ 8511 10378  6207 ...  6362  7230 12225]
 ...
 [ 8352  6250  4334 ... 11207 10171   817]
 [ 8611 12986  3695 ...  2177  3880 13001]
 [ 8205  2431 10488 ...  2514 12639 12261]]


The labels with length 907:
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

Now with the use of featureGenerator, DataLoader and DataSetCreator, the dataset is generated.

2.2.2 The same process for test data

There are two way for generating the test data set: 1) from FASTA file or 2) from a built dataset.

2.2.2.1 generate test dataset from other FASTA files

Sometimes the test data if from another bath/experiment, in this case, just generate the test dataset in the same way when generating the training set.

For example, in this notebook, we can load provided test data in folder 'data/protein/test'.

NOTE: You can skip this subsection if you want to generate them by splitting. The parameter spcLen and object featureGenerator should be the same. And when generating the matrix, usually we don't have to shuffle the sample since it will not be used in training.

```
[12]: #the paths
dataTestFilePaths = ['../examples/typeIII_secreted_effectors_prediction/data/
 ↪test_pos.txt','../examples/typeIII_secreted_effectors_prediction/data/test_neg.
 ↪txt']
#the related labels
dataTestLabel = [1, 0]
#a list for recording the DataLoader
testDataLoaders = []
for i,dataPath in enumerate(dataTestFilePaths):
    #init
    dataLoader = dataProcess.DataLoader(label = dataTestLabel[i],␣
 ↪featureGenerator=featureGenerator)
    #file read
    dataLoader.readFile(dataPath, spcLen = spcLen)
    testDataLoaders.append(dataLoader)

testDataSetCreator = dataProcess.DataSetCreator(testDataLoaders)
testDataMat, testLabelArr = testDataSetCreator.getDataSet(toShuffle=False)
```

We can have a look as well

```
[13]: print('Matrix with shape %d x %d:' %(testDataMat.shape[0],testDataMat.shape[1]))
print(testDataMat)
print('\n')
print('The labels with length %d:' %(len(testLabelArr)))
print(testLabelArr)
```

```
Matrix with shape 227 x 98:
[[ 8539 11104  7505 ...  1363   291  7572]
 [ 8219  2784  2081 ...  6128  1150 12333]
 [ 8244  3442  1615 ...  9857 10224  2194]
 ...
 [ 8179  1745 10229 ...   911  6114   787]
 [ 8304  5009  7206 ...  7802  9530  1730]
 [ 8387  7167 10600 ...  5192 11963 12264]]


The labels with length 227:
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0]
```

2.2.2.2 Alternative: generate test dataset by spliting a built one

Sometimes, we only have one dataset and all the samples in it could be used for either training or test. Thus autoBioSeqpy provided a way for splitting the dataset into two.

The method is provided in class DataSetCreator. A new parameter dataSplitScale is provided to control the splitting ratio, if the 'dataSplitScale' is 0.8, then the training dataset is 80% and the test dataset is 20% from the provided dataset.

In this notebook, we use the trainDataSetCreator as the example.

NOTE: This subsection is an alternative of section 2.2.2.1. You can choose one as you need.

```
[14]: dataSplitScale = 0.8
      trainDataMat, testDataMat, trainLabelArr, testLabelArr = trainDataSetCreator.
       ↪getTrainTestSet(dataSplitScale, toShuffle=True)
```

We can have a look as well

```
[15]: print('Training:')
      print('Matrix with shape %d x %d:' %(trainDataMat.shape[0],trainDataMat.
       ↪shape[1]))
      print(trainDataMat)
      print('\n')
      print('The labels with length %d:' %(len(trainLabelArr)))
      print(trainLabelArr)
      print('\n######################################################################################
      print('########################################################################################
      print('Testing:')
      print('Matrix with shape %d x %d:' %(testDataMat.shape[0],testDataMat.shape[1]))
      print(testDataMat)
      print('\n')
      print('The labels with length %d:' %(len(testLabelArr)))
      print(testLabelArr)
```

```
Training:
Matrix with shape 725 x 98:
[[ 8566 11820  8532 ...  4097  1067 10169]
 [ 8249  3572  5001 ...   480 12483  8200]
 [ 8246  3494  2977 ...    27   711   926]
 ...
 [ 8199  2269  6267 ...  6173  2319  7571]
 [ 8571 11944 11766 ...  1720  9584  3123]
 [ 8258  3801 10948 ...  6087    78  2035]]
```

```
The labels with length 725:
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

################################################################################
########
################################################################################
########

Testing:
Matrix with shape 182 x 98:
[[ 8244  3442  1615 ...  9872 10614 12326]
 [ 8255  3722  8904 ...  3054  9103  8207]
 [ 8586 12339  4447 ...   706   793  3045]
 ...
 [ 8352  6253  4404 ... 11082  6935  4560]
 [ 8494  9932 12172 ...  9465    32   847]
 [ 8348  6139  1444 ... 11198  9933 12197]]


The labels with length 182:
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

3. Data Modeling and Testing

After the data generated in section 2, now the dataset is available for modeling. Since it is a matrix, the data could be used for not only deep learning but also other machine learning as well.

Here we made a brief introduce by using keras for deep learning, and provided a traditional example by using random forest at last.

3.1 Using keras for modeling

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano.

Keras is useful for modeling the dataset by both 'dict' and 'onehot', but the neural network and related parameters should be set carefully.

Few templates provided in the folder 'models', users could copy them directly and change few related parameters such as to make sure the shape of the data is the same as the kernel size of the first layer.

In this notebook, since 'dict' is used as the encoding, the 1D neural network is a good choice for modeling, therefore, the model in 'model/CNN_Conv1D+GlobalMaxPooling.py'. Here the maxlen should be changed as the same with spcLen.

Here are two ways for using the model, one is write (or copy/post) the code in the script directly, another one is read a built model (in .json format) by using our provided module.

3.1 building keras model directly

3.1.1 model generating

As mentioned before, users could write any code for building keras neural network, but should modify the parameters manually.

[14]:
```
os.environ["CUDA_VISIBLE_DEVICES"] = '-1' #force using CPU, comment it for using␣
 ↪GPU
```

[15]:
```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.layers import Embedding
from keras.layers import Conv1D, GlobalMaxPooling1D
from keras import optimizers




# set parameters:
embedding_size = 128
filters = 250
kernel_size = 3
hidden_dims = 250
batch_size = 40
epochs = 25

#the parameter which need to modified
if useKMer:
    maxlen = spcLen - KMerNum + 1
    max_features = 26 ** KMerNum
```

```
else:
    maxlen = spcLen
    max_features = 26


print('Building model...')
model = Sequential()
# we start off with an efficient embedding layer which maps amino acids
# indices into embedding_dims dimensions
model.add(Embedding(max_features, embedding_size, input_length = maxlen))
model.add(Dropout(0.2))
# we add a Convolution1D, which will learn filters word group filters of
# size filter_length:
model.add(Conv1D(filters,kernel_size,padding = 'valid',activation =␣
 ↪'relu',strides = 1))
# we use max pooling:
model.add(GlobalMaxPooling1D())
# We add a vanilla hidden layer:
model.add(Dense(hidden_dims))
model.add(Dropout(0.2))
model.add(Activation('relu'))
# We project onto a single unit output layer, and squash it with a sigmoid:
model.add(Dense(1))
model.add(Activation('sigmoid'))

model.compile(loss = 'binary_crossentropy',optimizer = optimizers.Adam(),metrics␣
 ↪= ['acc'])

model.summary()
```

```
Using TensorFlow backend.

Building model...
Model: "sequential_1"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 98, 128)           2249728
_________________________________________________________________
dropout_1 (Dropout)          (None, 98, 128)           0
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 96, 250)           96250
_________________________________________________________________
global_max_pooling1d_1 (Glob (None, 250)               0
_________________________________________________________________
dense_1 (Dense)              (None, 250)               62750
_________________________________________________________________
```

```
dropout_2 (Dropout)          (None, 250)               0
_________________________________________________________________
activation_1 (Activation)    (None, 250)               0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 251
_________________________________________________________________
activation_2 (Activation)    (None, 1)                 0
=================================================================
Total params: 2,408,979
Trainable params: 2,408,979
Non-trainable params: 0
_________________________________________________________________
```

3.1.2 training

After the model built, now the dataset is ready for training, keras provided a framework for the training phase, just use it for the training dataset.

NOTE: The parameters batch_size and epochs are defined above.

analysisPlot

analysisPlot is a module provided for analyze the modeling process when using keras. We can import it easily since the search path is set before.

[16]:
```python
import analysisPlot
```

[17]:
```python
history = analysisPlot.LossHistory()
model.fit(trainDataMat, trainLabelArr,batch_size = batch_size,epochs =␣
 ↪epochs,validation_split = 0.1,callbacks = [history])
```

```
E:\Anaconda3\lib\site-
packages\tensorflow_core\python\framework\indexed_slices.py:433: UserWarning:
Converting sparse IndexedSlices to a dense Tensor of unknown shape. This may
consume a large amount of memory.
  "Converting sparse IndexedSlices to a dense Tensor of unknown shape. "

Train on 816 samples, validate on 91 samples
Epoch 1/25
816/816 [==============================] - 1s 1ms/step - loss: 0.6649 - acc:
0.6299 - val_loss: 0.6022 - val_acc: 1.0000
Epoch 2/25
816/816 [==============================] - 1s 942us/step - loss: 0.6526 - acc:
0.6287 - val_loss: 0.3976 - val_acc: 1.0000
Epoch 3/25
816/816 [==============================] - 1s 971us/step - loss: 0.6107 - acc:
0.6287 - val_loss: 0.4637 - val_acc: 1.0000
Epoch 4/25
816/816 [==============================] - 1s 978us/step - loss: 0.5182 - acc:
0.6483 - val_loss: 0.3803 - val_acc: 1.0000
Epoch 5/25
```

```
816/816 [==============================] - 1s 978us/step - loss: 0.2869 - acc:
0.9608 - val_loss: 0.3575 - val_acc: 0.9231
Epoch 6/25
816/816 [==============================] - 1s 969us/step - loss: 0.0620 - acc:
0.9963 - val_loss: 0.2415 - val_acc: 0.9451
Epoch 7/25
816/816 [==============================] - 1s 971us/step - loss: 0.0105 - acc:
1.0000 - val_loss: 0.1417 - val_acc: 0.9560
Epoch 8/25
816/816 [==============================] - 1s 978us/step - loss: 0.0035 - acc:
1.0000 - val_loss: 0.1644 - val_acc: 0.9451
Epoch 9/25
816/816 [==============================] - 1s 980us/step - loss: 0.0021 - acc:
1.0000 - val_loss: 0.1514 - val_acc: 0.9560
Epoch 10/25
816/816 [==============================] - 1s 983us/step - loss: 0.0013 - acc:
1.0000 - val_loss: 0.1468 - val_acc: 0.9560
Epoch 11/25
816/816 [==============================] - 1s 975us/step - loss: 9.5325e-04 -
acc: 1.0000 - val_loss: 0.1551 - val_acc: 0.9560
Epoch 12/25
816/816 [==============================] - 1s 975us/step - loss: 7.7185e-04 -
acc: 1.0000 - val_loss: 0.1534 - val_acc: 0.9560
Epoch 13/25
816/816 [==============================] - 1s 974us/step - loss: 5.5317e-04 -
acc: 1.0000 - val_loss: 0.1479 - val_acc: 0.9560
Epoch 14/25
816/816 [==============================] - 1s 976us/step - loss: 4.5754e-04 -
acc: 1.0000 - val_loss: 0.1532 - val_acc: 0.9560
Epoch 15/25
816/816 [==============================] - 1s 970us/step - loss: 3.4461e-04 -
acc: 1.0000 - val_loss: 0.1422 - val_acc: 0.9560
Epoch 16/25
816/816 [==============================] - 1s 970us/step - loss: 2.8570e-04 -
acc: 1.0000 - val_loss: 0.1490 - val_acc: 0.9560
Epoch 17/25
816/816 [==============================] - 1s 975us/step - loss: 2.4461e-04 -
acc: 1.0000 - val_loss: 0.1516 - val_acc: 0.9560
Epoch 18/25
816/816 [==============================] - 1s 970us/step - loss: 2.1665e-04 -
acc: 1.0000 - val_loss: 0.1497 - val_acc: 0.9560
Epoch 19/25
816/816 [==============================] - 1s 974us/step - loss: 1.9670e-04 -
acc: 1.0000 - val_loss: 0.1560 - val_acc: 0.9560
Epoch 20/25
816/816 [==============================] - 1s 978us/step - loss: 1.5177e-04 -
acc: 1.0000 - val_loss: 0.1552 - val_acc: 0.9560
Epoch 21/25
```

```
816/816 [==============================] - 1s 979us/step - loss: 1.6182e-04 -
acc: 1.0000 - val_loss: 0.1558 - val_acc: 0.9560
Epoch 22/25
816/816 [==============================] - 1s 975us/step - loss: 1.2017e-04 -
acc: 1.0000 - val_loss: 0.1520 - val_acc: 0.9560
Epoch 23/25
816/816 [==============================] - 1s 978us/step - loss: 1.3496e-04 -
acc: 1.0000 - val_loss: 0.1572 - val_acc: 0.9560
Epoch 24/25
816/816 [==============================] - 1s 969us/step - loss: 1.0609e-04 -
acc: 1.0000 - val_loss: 0.1584 - val_acc: 0.9560
Epoch 25/25
816/816 [==============================] - 1s 968us/step - loss: 1.0967e-04 -
acc: 1.0000 - val_loss: 0.1542 - val_acc: 0.9560
```

```
[17]: <keras.callbacks.callbacks.History at 0x1e7e97ab888>
```

3.1.3 testing and output analysis

The frame for predicting is provided by keras as well, therefore we can make the predict as well.

```
[18]: predicted_Probability = model.predict(testDataMat)
      prediction = model.predict_classes(testDataMat)
```

3.1.4 showing modeling figures and predicting preference

Usually users would like to know the predicting performance, therefore the related function is provided as well.

Users could get how the loss changes as the epoch increasing, and some metrices (ACC, Recall, MCC…) as well.

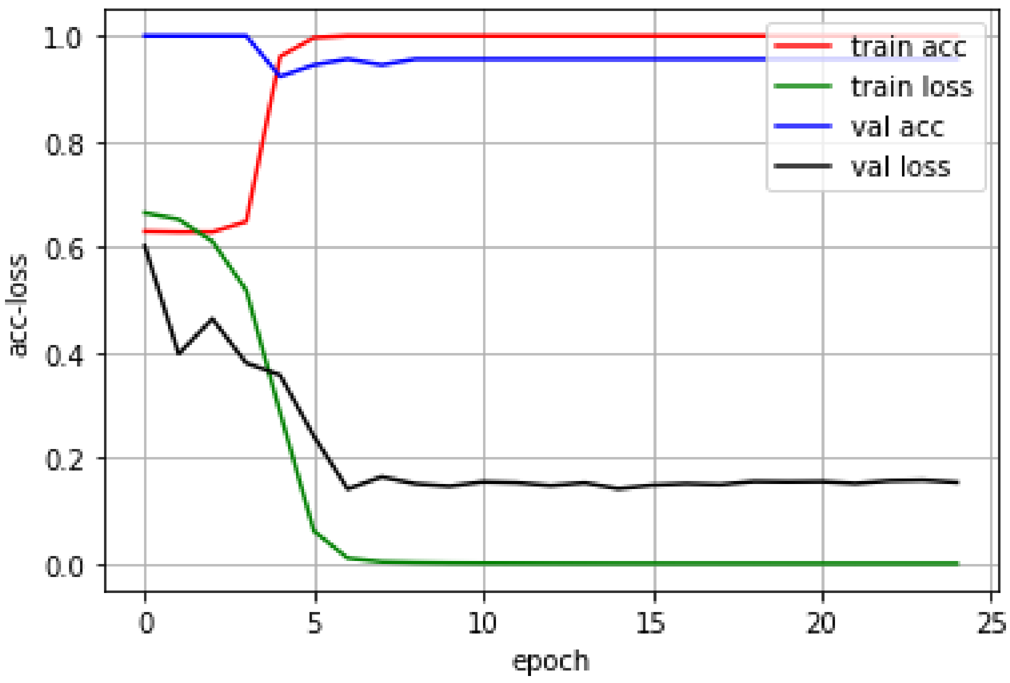
All the figure is available for save by change the parameter savePath to a real path.

This time the metrices are available in sklearn, import them at first.

```
[19]: from sklearn.metrics import␣
        ↪accuracy_score,f1_score,roc_auc_score,recall_score,precision_score,confusion_matrix,matthews_
```

The change of loss

```
[20]: history.loss_plot('epoch',showFig=True,savePath=None)
```



confusion matrix and related metrices

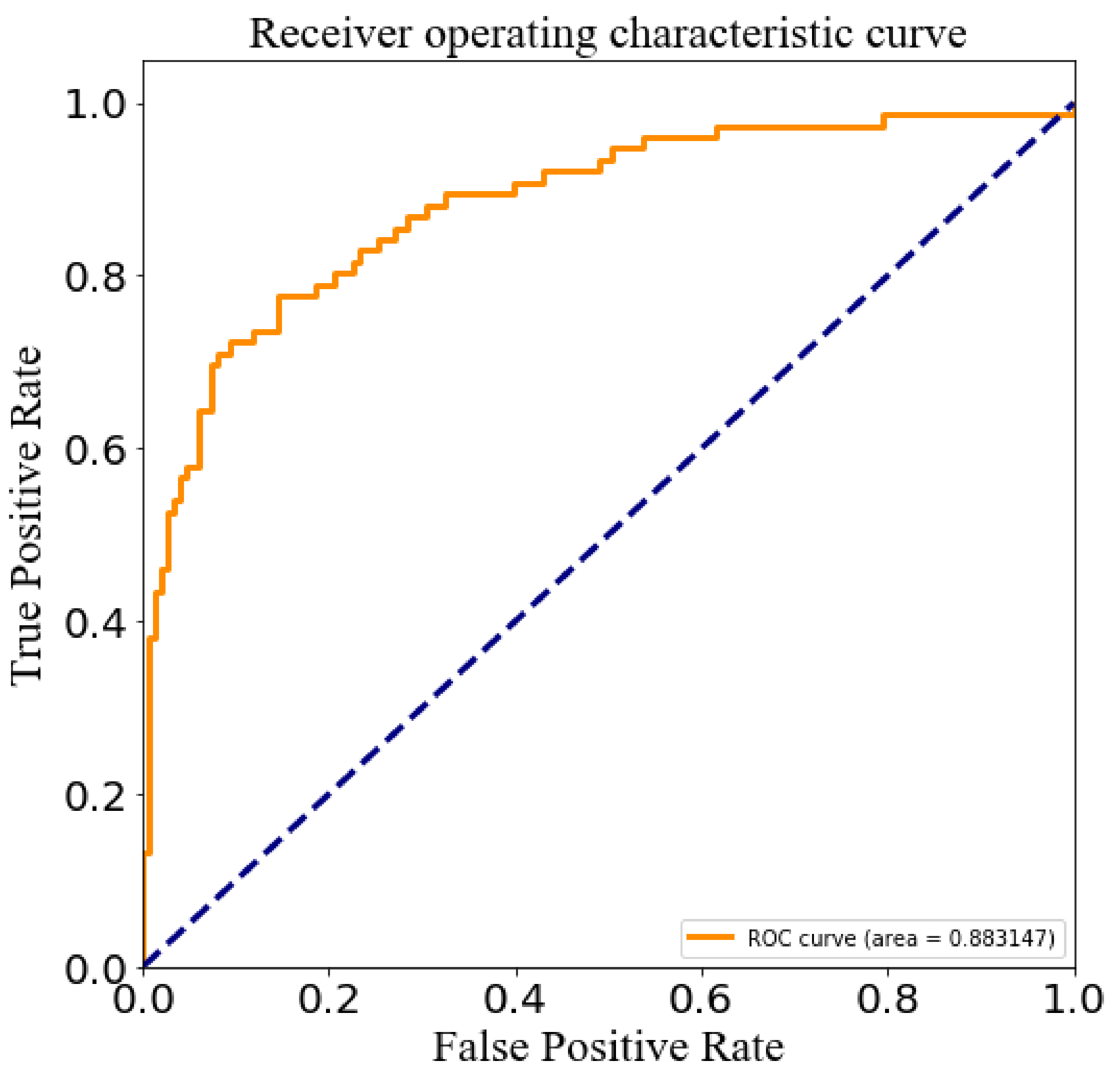
```
[21]: cm=confusion_matrix(testLabelArr,prediction)
print(cm)
print("ACC: %f "%accuracy_score(testLabelArr,prediction))
print("F1: %f "%f1_score(testLabelArr,prediction))
print("Recall: %f "%recall_score(testLabelArr,prediction))
print("Pre: %f "%precision_score(testLabelArr,prediction))
print("MCC: %f "%matthews_corrcoef(testLabelArr,prediction))
fpr,tpr,threshold = roc_curve(testLabelArr, predicted_Probability)
auc_roc = auc(fpr,tpr)
print("AUC: %f "%auc_roc)
```

```
[[142   9]
 [ 28  48]]
ACC: 0.837004
F1: 0.721805
Recall: 0.631579
Pre: 0.842105
MCC: 0.622460
AUC: 0.883147
```

ROC curve

```
[22]: analysisPlot.
  ↪plotROC(testLabelArr,predicted_Probability,showFig=True,savePath=None)
```

```
<Figure size 432x288 with 0 Axes>
```



3.1.5 Save/Load a module (optional)

As mentioned before, keras is able to save a built model and read it again, it is available for establish a model without the data or using it for transfer learning.

Therefore, a shor part of the code (i.e. in our module 'moduleRead') is provided here for implement this function.

Model save

Not only the module, but also the weight could be saved.

[23]:
```
modelSavePath = './tmpModel.json'
weightSavePath = './tmpWeight.bin'
```

[24]:
```
model_json = model.to_json()
with open(modelSavePath, "w") as json_file:
    json_file.write(model_json)
model.save_weights(weightSavePath)
```

Model Load

[25]:
```
from keras.models import model_from_json
```

[26]:
```
json_file = open(modelSavePath, 'r')
loaded_model_json = json_file.read()
json_file.close()
loaded_model = model_from_json(loaded_model_json)
if not weightSavePath is None:
    loaded_model.load_weights(weightSavePath)
```

Sometimes a loaded model should be recompiled before training

[27]:
```
model = loaded_model
model.compile(loss = 'binary_crossentropy',optimizer = optimizers.Adam(),metrics␣
 ↪= ['acc'])
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 98, 128)           2249728
_________________________________________________________________
dropout_1 (Dropout)          (None, 98, 128)           0
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 96, 250)           96250
_________________________________________________________________
global_max_pooling1d_1 (Glob (None, 250)               0
_________________________________________________________________
dense_1 (Dense)              (None, 250)               62750
_________________________________________________________________
dropout_2 (Dropout)          (None, 250)               0
_________________________________________________________________
activation_1 (Activation)    (None, 250)               0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 251
_________________________________________________________________
activation_2 (Activation)    (None, 1)                 0
=================================================================
Total params: 2,408,979
Trainable params: 2,408,979
Non-trainable params: 0
```

And then, you can use the loaded model for training/predict as you want.

3.2 pre-trained embedding (Alternative of 3.1)

In bio-sequence modeling, usually only the training set will be used for model training, but sometimes users would like to make some global normalizing before it.

Therefore, here we provided an alternative way for making the embedding for the overall dataset before the formal training.

3.2.1 Embedding layer preparing

Firstly, we provide a model, the construction could be simpler than the formal one.

[28]:
```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.layers import Embedding
from keras.layers import Conv1D, GlobalMaxPooling1D
from keras import optimizers





# set parameters:
embedding_size = 128
filters = 250
kernel_size = 3
hidden_dims = 250
batch_size = 40
epochs = 25

#the parameter which need to modified
if useKMer:
    maxlen = spcLen - KMerNum + 1
    max_features = 26 ** KMerNum

else:
    maxlen = spcLen
    max_features = 26

print('Building model...')
preModel = Sequential()
# we start off with an efficient embedding layer which maps amino acids
# indices into embedding_dims dimensions
preModel.add(Embedding(max_features, embedding_size, input_length = maxlen))
preModel.add(Conv1D(filters,kernel_size,padding = 'valid',activation =␣
 ↪'relu',strides = 1))
# we use max pooling:
preModel.add(GlobalMaxPooling1D())
# We add a vanilla hidden layer:

preModel.add(Dense(1))
preModel.add(Activation('sigmoid'))

preModel.compile(loss = 'binary_crossentropy',optimizer = optimizers.
 ↪Adam(),metrics = ['acc'])

preModel.summary()
```

```
Building model...
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, 98, 128)           2249728
_________________________________________________________________
conv1d_2 (Conv1D)            (None, 96, 250)           96250
_________________________________________________________________
global_max_pooling1d_2 (Glob (None, 250)               0
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 251
_________________________________________________________________
activation_3 (Activation)    (None, 1)                 0
=================================================================
Total params: 2,346,229
Trainable params: 2,346,229
Non-trainable params: 0
_________________________________________________________________
```

3.2.2 Generating the overall dataset and fitting

The generating is to concatenate the training and testing dataset. The training (fitting) will be launched afterwards.

[29]:
```
import analysisPlot
```

[30]:
```
overallDataMat = np.concatenate([trainDataMat,testDataMat])
overallLabel = np.concatenate([trainLabelArr,testLabelArr])
print(overallDataMat.shape, overallLabel.shape)
```

```
(1134, 98) (1134,)
```

[31]:
```
history = analysisPlot.LossHistory()
preModel.fit(overallDataMat, overallLabel,batch_size = batch_size,epochs =␣
↪epochs,validation_split = 0.1,callbacks = [history])
```

```
E:\Anaconda3\lib\site-
packages\tensorflow_core\python\framework\indexed_slices.py:433: UserWarning:
Converting sparse IndexedSlices to a dense Tensor of unknown shape. This may
consume a large amount of memory.
  "Converting sparse IndexedSlices to a dense Tensor of unknown shape. "
```

```
Train on 1020 samples, validate on 114 samples
Epoch 1/25
1020/1020 [==============================] - 1s 965us/step - loss: 0.6619 - acc:
0.6294 - val_loss: 0.4297 - val_acc: 1.0000
Epoch 2/25
1020/1020 [==============================] - 1s 884us/step - loss: 0.5828 - acc:
0.6304 - val_loss: 0.4447 - val_acc: 1.0000
Epoch 3/25
1020/1020 [==============================] - 1s 901us/step - loss: 0.4912 - acc:
0.7304 - val_loss: 0.4199 - val_acc: 0.9737
Epoch 4/25
1020/1020 [==============================] - 1s 900us/step - loss: 0.3625 - acc:
0.9235 - val_loss: 0.3333 - val_acc: 0.9737
Epoch 5/25
1020/1020 [==============================] - 1s 905us/step - loss: 0.2224 - acc:
0.9794 - val_loss: 0.2851 - val_acc: 0.9737
Epoch 6/25
1020/1020 [==============================] - 1s 897us/step - loss: 0.1134 - acc:
0.9971 - val_loss: 0.2168 - val_acc: 0.9649
Epoch 7/25
1020/1020 [==============================] - 1s 898us/step - loss: 0.0506 - acc:
1.0000 - val_loss: 0.1733 - val_acc: 0.9561
Epoch 8/25
1020/1020 [==============================] - 1s 888us/step - loss: 0.0238 - acc:
1.0000 - val_loss: 0.1517 - val_acc: 0.9474
Epoch 9/25
1020/1020 [==============================] - 1s 888us/step - loss: 0.0127 - acc:
1.0000 - val_loss: 0.1475 - val_acc: 0.9386
Epoch 10/25
1020/1020 [==============================] - 1s 894us/step - loss: 0.0077 - acc:
1.0000 - val_loss: 0.1416 - val_acc: 0.9386
Epoch 11/25
1020/1020 [==============================] - 1s 894us/step - loss: 0.0052 - acc:
1.0000 - val_loss: 0.1425 - val_acc: 0.9298
Epoch 12/25
1020/1020 [==============================] - 1s 890us/step - loss: 0.0037 - acc:
1.0000 - val_loss: 0.1486 - val_acc: 0.9298
Epoch 13/25
```

```
1020/1020 [==============================] - 1s 892us/step - loss: 0.0028 - acc:
1.0000 - val_loss: 0.1547 - val_acc: 0.9211
Epoch 14/25
1020/1020 [==============================] - 1s 889us/step - loss: 0.0022 - acc:
1.0000 - val_loss: 0.1599 - val_acc: 0.9211
Epoch 15/25
1020/1020 [==============================] - 1s 891us/step - loss: 0.0018 - acc:
1.0000 - val_loss: 0.1626 - val_acc: 0.9211
Epoch 16/25
1020/1020 [==============================] - 1s 890us/step - loss: 0.0014 - acc:
1.0000 - val_loss: 0.1689 - val_acc: 0.9211
Epoch 17/25
1020/1020 [==============================] - 1s 893us/step - loss: 0.0012 - acc:
1.0000 - val_loss: 0.1711 - val_acc: 0.9211
Epoch 18/25
1020/1020 [==============================] - 1s 892us/step - loss: 0.0010 - acc:
1.0000 - val_loss: 0.1787 - val_acc: 0.9211
Epoch 19/25
1020/1020 [==============================] - 1s 888us/step - loss: 8.7452e-04 -
acc: 1.0000 - val_loss: 0.1837 - val_acc: 0.9211
Epoch 20/25
1020/1020 [==============================] - 1s 889us/step - loss: 7.5850e-04 -
acc: 1.0000 - val_loss: 0.1869 - val_acc: 0.9211
Epoch 21/25
1020/1020 [==============================] - 1s 890us/step - loss: 6.6641e-04 -
acc: 1.0000 - val_loss: 0.1916 - val_acc: 0.9211
Epoch 22/25
1020/1020 [==============================] - 1s 888us/step - loss: 5.8762e-04 -
acc: 1.0000 - val_loss: 0.1953 - val_acc: 0.9211
Epoch 23/25
1020/1020 [==============================] - 1s 885us/step - loss: 5.2300e-04 -
acc: 1.0000 - val_loss: 0.1989 - val_acc: 0.9211
Epoch 24/25
1020/1020 [==============================] - 1s 890us/step - loss: 4.6895e-04 -
acc: 1.0000 - val_loss: 0.2024 - val_acc: 0.9211
Epoch 25/25
1020/1020 [==============================] - 1s 890us/step - loss: 4.2210e-04 -
acc: 1.0000 - val_loss: 0.2068 - val_acc: 0.9211
```

```
[31]: <keras.callbacks.callbacks.History at 0x1e7e53218c8>
```

3.2.4 Extracting the weights of embedding layer

Now we get the weights from the built model (i.e. preModel), then the weights could be extracted from the embedding layer.

```
[32]: for l in preModel.layers:
          print(l.name)
```

```
embedding_2
conv1d_2
global_max_pooling1d_2
dense_3
activation_3
```

From the code above we can see that the first layer is the embedding layer, thus extract the weights by using 'get_weights()'.

[33]:
```python
#get the weights
embeddingLayer = preModel.layers[0]
embeddingWeights = embeddingLayer.get_weights()
print(embeddingWeights)
```

```
[array([[ 0.02291623, -0.02706047,  0.0009923 , ..., -0.0114282 ,
         0.04809933, -0.02255343],
       [-0.00857742,  0.04148132,  0.07143663, ...,  0.05337083,
         0.01992362, -0.01415086],
       [ 0.07462072, -0.05381688,  0.04782455, ..., -0.00455487,
        -0.03060702, -0.00116512],
       ...,
       [-0.04508328,  0.00993661, -0.00987247, ..., -0.00843386,
        -0.00508151, -0.02714957],
       [-0.00925578, -0.00170332,  0.02548107, ..., -0.01350963,
        -0.00681692,  0.02636449],
       [ 0.02280037,  0.04090396,  0.04468245, ..., -0.02398411,
        -0.00431467,  0.03117689]], dtype=float32)]
```

3.2.5 Build a new model and import the extrated weights and fitting

A same model as in section 3.1.1 will be generated, but the difference is the weights of the embedding layer will be changed to the last one.

Please note that here the data for training is just the training dataset (trainDataMat in this case), the overall dataset is only for generating the weight of embedding layer.

First, build a same model as 3.1.1

Please note that the structure (i.e. the parameter) of the embedding layer shold be the same as 3.2.1, otherwise the weight could not be imported.

[34]:
```python
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.layers import Embedding
from keras.layers import Conv1D, GlobalMaxPooling1D
from keras import optimizers




# set parameters:
embedding_size = 128
```

```
filters = 250
kernel_size = 3
hidden_dims = 250
batch_size = 40
epochs = 25

#the parameter which need to modified
if useKMer:
    maxlen = spcLen - KMerNum + 1
    max_features = 26 ** KMerNum

else:
    maxlen = spcLen
    max_features = 26


print('Building model...')
newModel = Sequential()

newModel.add(Embedding(max_features, embedding_size, input_length = maxlen))
newModel.add(Dropout(0.2))
newModel.add(Conv1D(filters,kernel_size,padding = 'valid',activation =␣
 ↪'relu',strides = 1))
newModel.add(GlobalMaxPooling1D())
newModel.add(Dense(hidden_dims))
newModel.add(Dropout(0.2))
newModel.add(Activation('relu'))
newModel.add(Dense(1))
newModel.add(Activation('sigmoid'))

newModel.compile(loss = 'binary_crossentropy',optimizer = optimizers.
 ↪Adam(),metrics = ['acc'])

newModel.summary()
```

```
Building model...
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, 98, 128)           2249728
_________________________________________________________________
dropout_3 (Dropout)          (None, 98, 128)           0
_________________________________________________________________
conv1d_3 (Conv1D)            (None, 96, 250)           96250
_________________________________________________________________
global_max_pooling1d_3 (Glob (None, 250)               0
_________________________________________________________________
dense_4 (Dense)              (None, 250)               62750
_________________________________________________________________
dropout_4 (Dropout)          (None, 250)               0
_________________________________________________________________
activation_4 (Activation)    (None, 250)               0
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 251
_________________________________________________________________
activation_5 (Activation)    (None, 1)                 0
=================================================================
Total params: 2,408,979
Trainable params: 2,408,979
Non-trainable params: 0
_________________________________________________________________
```

Then, import the weight to the related layer (the first layer in this example).

```
[35]: newModel.layers[0].set_weights(embeddingWeights)
```

Then train the new model as usual

```
[36]: history = analysisPlot.LossHistory()
      newModel.fit(trainDataMat, trainLabelArr,batch_size = batch_size,epochs =␣
       ↪epochs,validation_split = 0.1,callbacks = [history])
```

```
E:\Anaconda3\lib\site-
packages\tensorflow_core\python\framework\indexed_slices.py:433: UserWarning:
Converting sparse IndexedSlices to a dense Tensor of unknown shape. This may
consume a large amount of memory.
  "Converting sparse IndexedSlices to a dense Tensor of unknown shape. "

Train on 816 samples, validate on 91 samples
Epoch 1/25
816/816 [==============================] - 1s 1ms/step - loss: 0.5083 - acc:
0.7243 - val_loss: 0.1536 - val_acc: 1.0000
Epoch 2/25
816/816 [==============================] - 1s 960us/step - loss: 0.0724 - acc:
1.0000 - val_loss: 0.0120 - val_acc: 1.0000
Epoch 3/25
816/816 [==============================] - 1s 975us/step - loss: 0.0039 - acc:
1.0000 - val_loss: 0.0023 - val_acc: 1.0000
Epoch 4/25
816/816 [==============================] - 1s 975us/step - loss: 8.4156e-04 -
acc: 1.0000 - val_loss: 0.0032 - val_acc: 1.0000
Epoch 5/25
816/816 [==============================] - 1s 974us/step - loss: 5.0911e-04 -
acc: 1.0000 - val_loss: 0.0035 - val_acc: 1.0000
Epoch 6/25
```

```
816/816 [==============================] - 1s 974us/step - loss: 4.0844e-04 -
acc: 1.0000 - val_loss: 0.0030 - val_acc: 1.0000
Epoch 7/25
816/816 [==============================] - 1s 997us/step - loss: 3.0999e-04 -
acc: 1.0000 - val_loss: 0.0027 - val_acc: 1.0000
Epoch 8/25
816/816 [==============================] - 1s 981us/step - loss: 2.6769e-04 -
acc: 1.0000 - val_loss: 0.0028 - val_acc: 1.0000
Epoch 9/25
816/816 [==============================] - 1s 973us/step - loss: 3.3147e-04 -
acc: 1.0000 - val_loss: 0.0016 - val_acc: 1.0000
Epoch 10/25
816/816 [==============================] - 1s 975us/step - loss: 2.0364e-04 -
acc: 1.0000 - val_loss: 0.0015 - val_acc: 1.0000
Epoch 11/25
816/816 [==============================] - 1s 978us/step - loss: 1.6598e-04 -
acc: 1.0000 - val_loss: 0.0017 - val_acc: 1.0000
Epoch 12/25
816/816 [==============================] - 1s 976us/step - loss: 1.2639e-04 -
acc: 1.0000 - val_loss: 0.0019 - val_acc: 1.0000
Epoch 13/25
816/816 [==============================] - 1s 973us/step - loss: 1.2472e-04 -
acc: 1.0000 - val_loss: 0.0020 - val_acc: 1.0000
Epoch 14/25
816/816 [==============================] - 1s 978us/step - loss: 9.7308e-05 -
acc: 1.0000 - val_loss: 0.0022 - val_acc: 1.0000
Epoch 15/25
816/816 [==============================] - 1s 978us/step - loss: 8.0542e-05 -
acc: 1.0000 - val_loss: 0.0021 - val_acc: 1.0000
Epoch 16/25
816/816 [==============================] - 1s 973us/step - loss: 8.9115e-05 -
acc: 1.0000 - val_loss: 0.0021 - val_acc: 1.0000
Epoch 17/25
816/816 [==============================] - 1s 973us/step - loss: 6.9674e-05 -
acc: 1.0000 - val_loss: 0.0021 - val_acc: 1.0000
Epoch 18/25
816/816 [==============================] - 1s 975us/step - loss: 6.8290e-05 -
acc: 1.0000 - val_loss: 0.0020 - val_acc: 1.0000
Epoch 19/25
816/816 [==============================] - 1s 986us/step - loss: 6.3112e-05 -
acc: 1.0000 - val_loss: 0.0019 - val_acc: 1.0000
Epoch 20/25
816/816 [==============================] - 1s 985us/step - loss: 6.0432e-05 -
acc: 1.0000 - val_loss: 0.0018 - val_acc: 1.0000
Epoch 21/25
816/816 [==============================] - 1s 969us/step - loss: 5.3223e-05 -
acc: 1.0000 - val_loss: 0.0018 - val_acc: 1.0000
Epoch 22/25
```

```
816/816 [==============================] - 1s 981us/step - loss: 4.5742e-05 -
acc: 1.0000 - val_loss: 0.0017 - val_acc: 1.0000
Epoch 23/25
816/816 [==============================] - 1s 971us/step - loss: 4.0128e-05 -
acc: 1.0000 - val_loss: 0.0017 - val_acc: 1.0000
Epoch 24/25
816/816 [==============================] - 1s 975us/step - loss: 3.8958e-05 -
acc: 1.0000 - val_loss: 0.0017 - val_acc: 1.0000
Epoch 25/25
816/816 [==============================] - 1s 965us/step - loss: 3.5064e-05 -
acc: 1.0000 - val_loss: 0.0018 - val_acc: 1.0000
```

```
[36]: <keras.callbacks.callbacks.History at 0x1e7e7d180c8>
```

3.2.6 Testing and output analysis by using the new model.

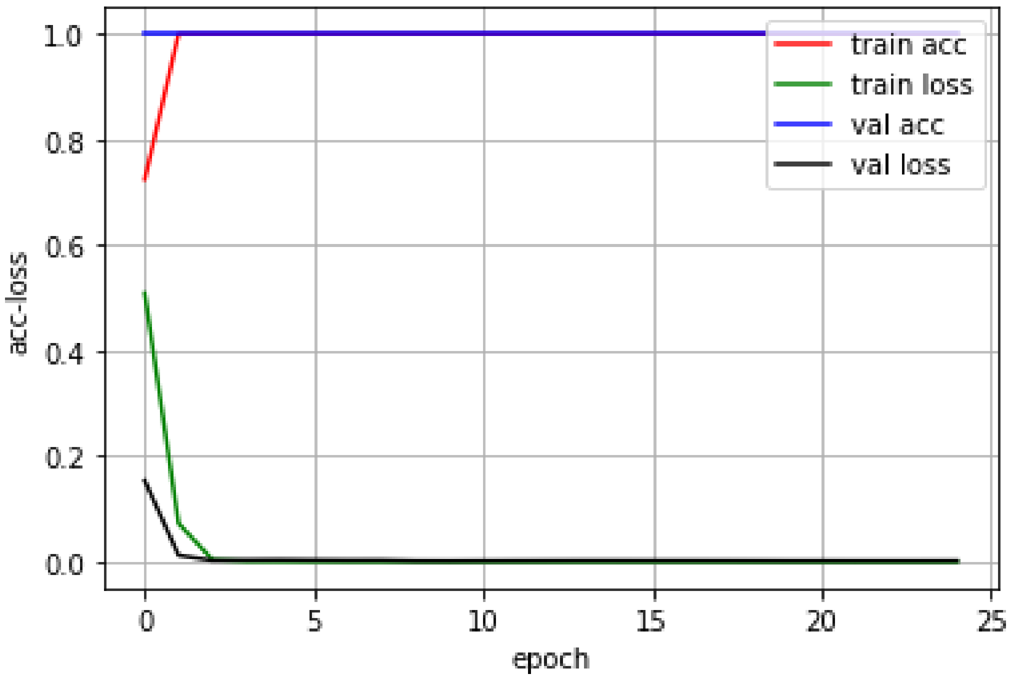
Since the overall embedding will take the test set in count, usually the predicting performance will be better than in section 3.1. But please use this way for training the bio-sequence carefully, especially for finding new biological insight.

```
[37]: predicted_Probability = newModel.predict(testDataMat)
      prediction = newModel.predict_classes(testDataMat)
```

```
[38]: from sklearn.metrics import␣
       ↪accuracy_score,f1_score,roc_auc_score,recall_score,precision_score,confusion_matrix,matthews_
```

The change of loss

```
[39]: history.loss_plot('epoch',showFig=True,savePath=None)
```



confusion matrix and related metrices

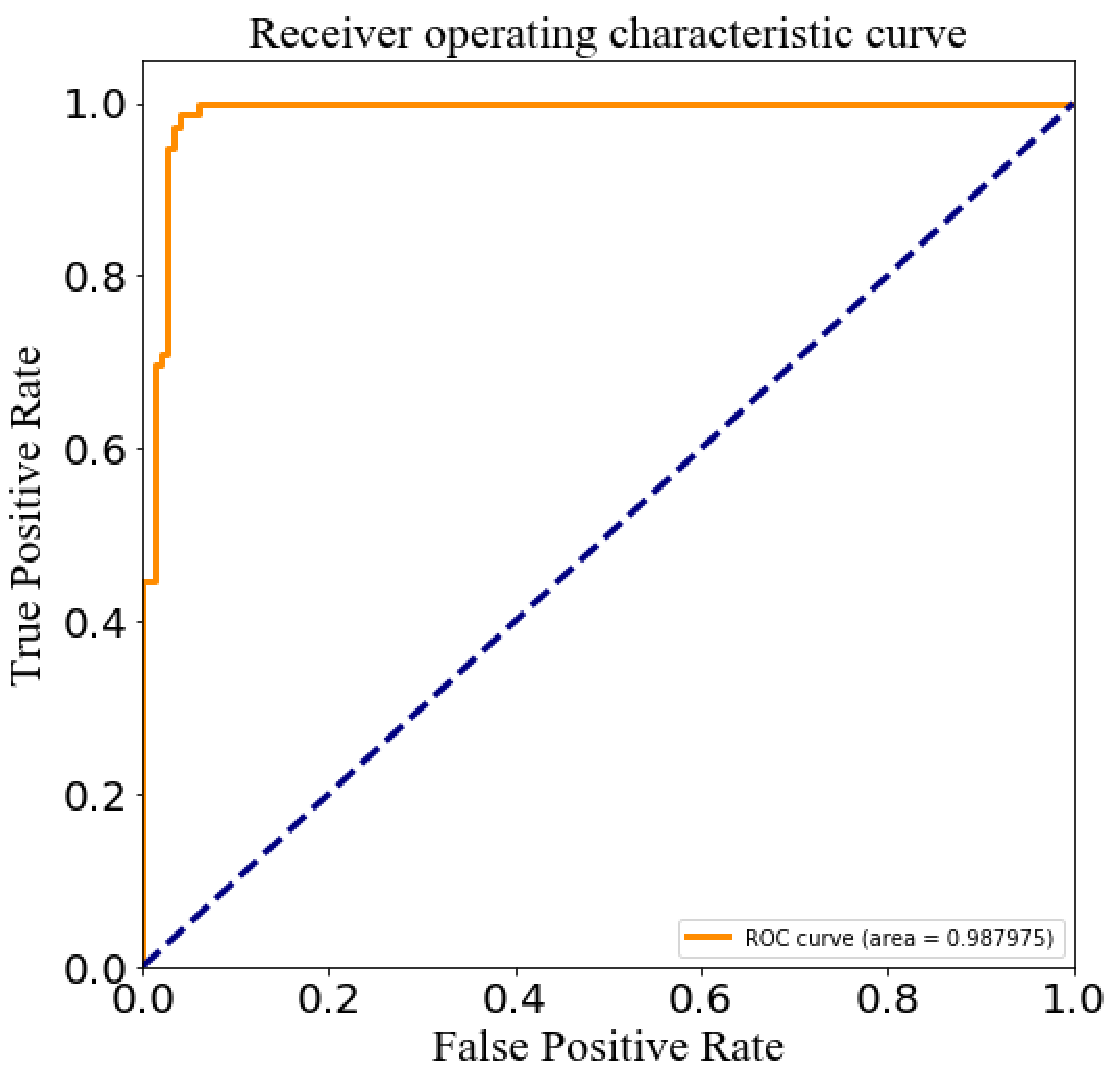
```
[40]: cm=confusion_matrix(testLabelArr,prediction)
      print(cm)
      print("ACC: %f "%accuracy_score(testLabelArr,prediction))
      print("F1: %f "%f1_score(testLabelArr,prediction))
      print("Recall: %f "%recall_score(testLabelArr,prediction))
      print("Pre: %f "%precision_score(testLabelArr,prediction))
      print("MCC: %f "%matthews_corrcoef(testLabelArr,prediction))
      fpr,tpr,threshold = roc_curve(testLabelArr, predicted_Probability)
      auc_roc = auc(fpr,tpr)
      print("AUC: %f "%auc_roc)
```

```
[[137  14]
 [  0  76]]
ACC: 0.938326
F1: 0.915663
Recall: 1.000000
Pre: 0.844444
MCC: 0.875301
AUC: 0.987975
```

ROC curve

```
[41]: analysisPlot.
       ↪plotROC(testLabelArr,predicted_Probability,showFig=True,savePath=None)
```

```
<Figure size 432x288 with 0 Axes>
```



3.3 Modeling with other machine learning method

Since we got a matrix, it is possible for using this matrix for many training works other than deep learning. The result is available for comparison or make some further analysis.

Here we provided a brief sample for using random forest for modeling and predicting.

[42]:
```
from sklearn.ensemble import RandomForestClassifier

#init
rf = RandomForestClassifier(n_estimators=10, max_depth=None,min_samples_split=2,␣
 ↪bootstrap=True)
```

```
#training
rf.fit(trainDataMat, trainLabelArr)

#predicting
rfPrediction = rf.predict(testDataMat)
```

confusion matrix and related metrices

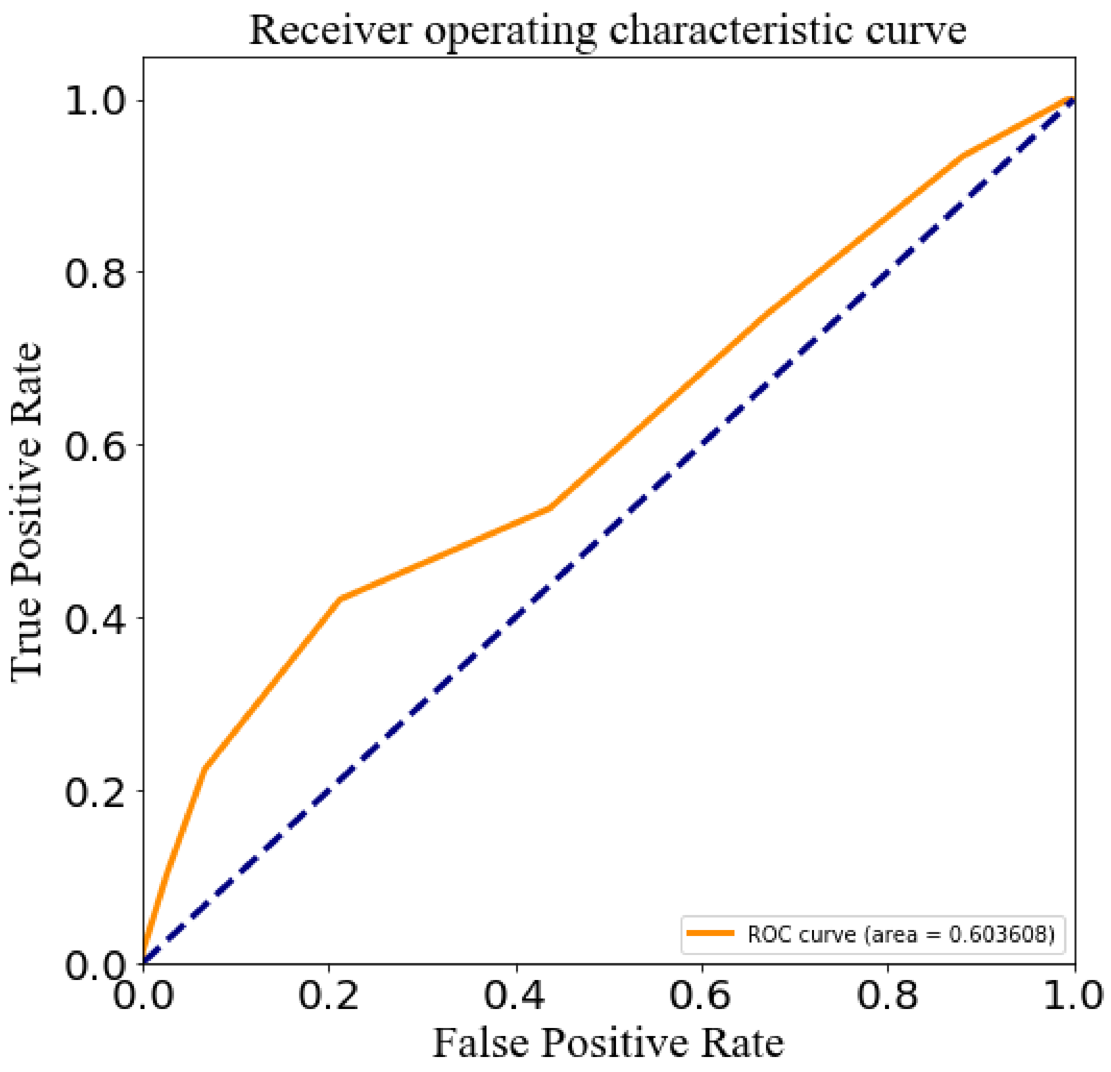
```
[43]: cm=confusion_matrix(testLabelArr,rfPrediction)
print(cm)
print("ACC: %f "%accuracy_score(testLabelArr,rfPrediction))
print("F1: %f "%f1_score(testLabelArr,rfPrediction))
print("Recall: %f "%recall_score(testLabelArr,rfPrediction))
print("Pre: %f "%precision_score(testLabelArr,rfPrediction))
print("MCC: %f "%matthews_corrcoef(testLabelArr,rfPrediction))
#for auc...
rfPredictedProbability = np.array(rf.predict_proba(testDataMat))
fpr,tpr,threshold = roc_curve(testLabelArr, rfPredictedProbability[:,1])
auc_roc = auc(fpr,tpr)
print("AUC: %f "%auc_roc)
```

```
[[141  10]
 [ 59  17]]
ACC: 0.696035
F1: 0.330097
Recall: 0.223684
Pre: 0.629630
MCC: 0.229544
AUC: 0.603608
```

ROC curve

```
[44]: rfPredictedProbability = np.array(rf.predict_proba(testDataMat))
analysisPlot.plotROC(testLabelArr,rfPredictedProbability[:
 ↪,1],showFig=True,savePath=None)
```

```
<Figure size 432x288 with 0 Axes>
```



4. Conclusion

In this notebook, we introduced how to use autoBioSeqpy for file reading and engaging it into a research workflow. We hope this notebook could help users to understand the basic way for using it for data transferring and evaluating the modeling result. Then users could use it as a part of their own researches.

We are looking forward to receive any feedback and suggesting. If you have any problem in using this tool, please do not hesitate to connect us at ljs@swmu.edu.cn, thanks.