

Handout for Machine Learning using Scikit-Learn

FINA 8823

Juerui Feng, Xinyuan Shao and Keer Yang

February 20, 2018

Contents

1	Introduction	2
2	Installing Python via Anaconda	2
2.1	What is Anaconda	2
2.2	How to install Anaconda	2
3	Accessing SEC Filling through EDGAR	4
3.1	Overview	4
3.2	Using the EDGAR Index Files	4
3.3	Example: Shareholder Activism	6
4	Web scraping	7
4.1	Description	7
5	Scikit-Learn	11
5.1	Cross Validation	11
5.2	Supervised Neural Network	13
5.3	Gradient Boosting	15

1 Introduction

This documentation is made for the introduction of machine learning techniques for corporate finance. We use a classification example to apply the technique. To be specific, we use multi-layer perception and gradient boosting in the classification of hedge fund activist targets, based on firm characteristics. In the process, we show how to extract information from SEC filings, and then apply the scikit-learn to corporate financial data. The key of this document is not to form a real research question and tackle it, but to show how the necessary toolbox are executed.

The rest of the sections are organized as: Section 2 introduces how to install python and get it running. Section 3 presents an introduction to SEC EDGAR system. Section 4 shows how to download and processing the EDGAR data. Section 5 shows how to apply scikit-learn to the data.

2 Installing Python via Anaconda

2.1 What is Anaconda

Anaconda Distribution is an open source, easy-to-install high performance Python and R Distribution, with the conda package and environment manager and collection of 1000+ open source packages with free community support.

Scientific packages in Python usually require a specific version of Python to run, and it's difficult to keep them from interacting with each other and update them. Anaconda Distribution makes getting and maintaining these packages quick and easy.

2.2 How to install Anaconda

To install the Anaconda Distribution, you can either directly download the package from <https://www.anaconda.com/download/#windows>

```

PS C:\WINDOWS\system32>
PS C:\WINDOWS\system32> conda install numpy
Solving environment: done

## Package Plan ##

  environment location: C:\ProgramData\Anaconda3

  added / updated specs:
    - numpy

The following packages will be downloaded:

  package-----|----- build                    3.6 MB
  numpy-1.13.3      |      py36h4a99626_2

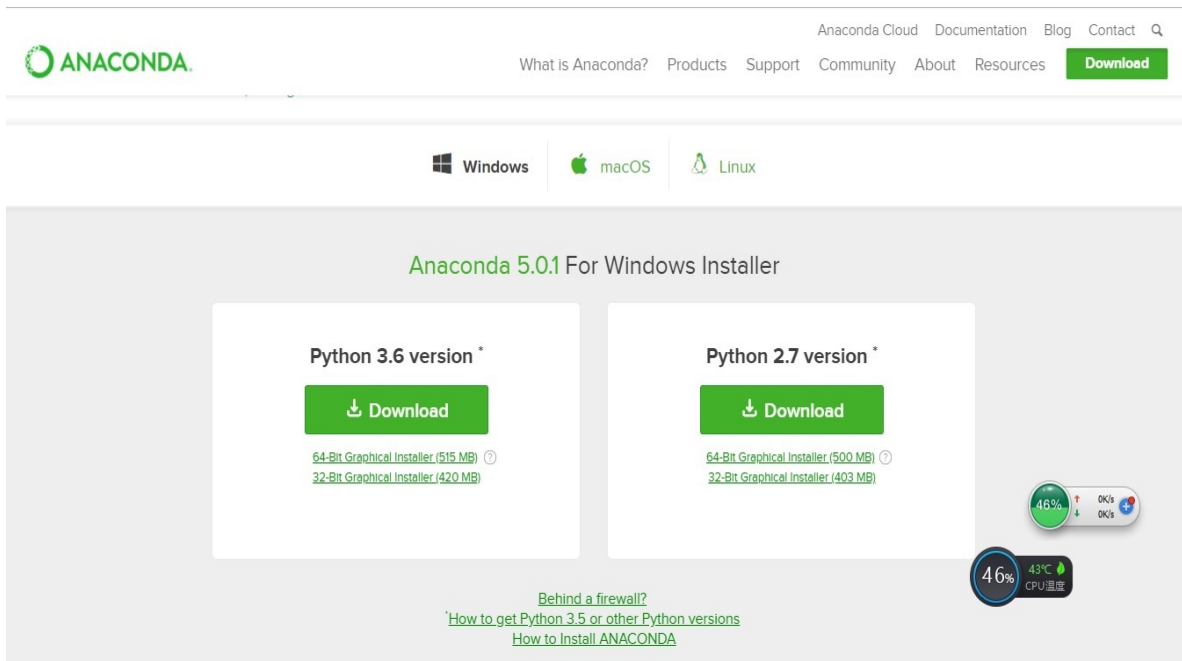
The following packages will be UPDATED:

  numpy: 1.13.3-py36ha320f96_0 --> 1.13.3-py36h4a99626_2

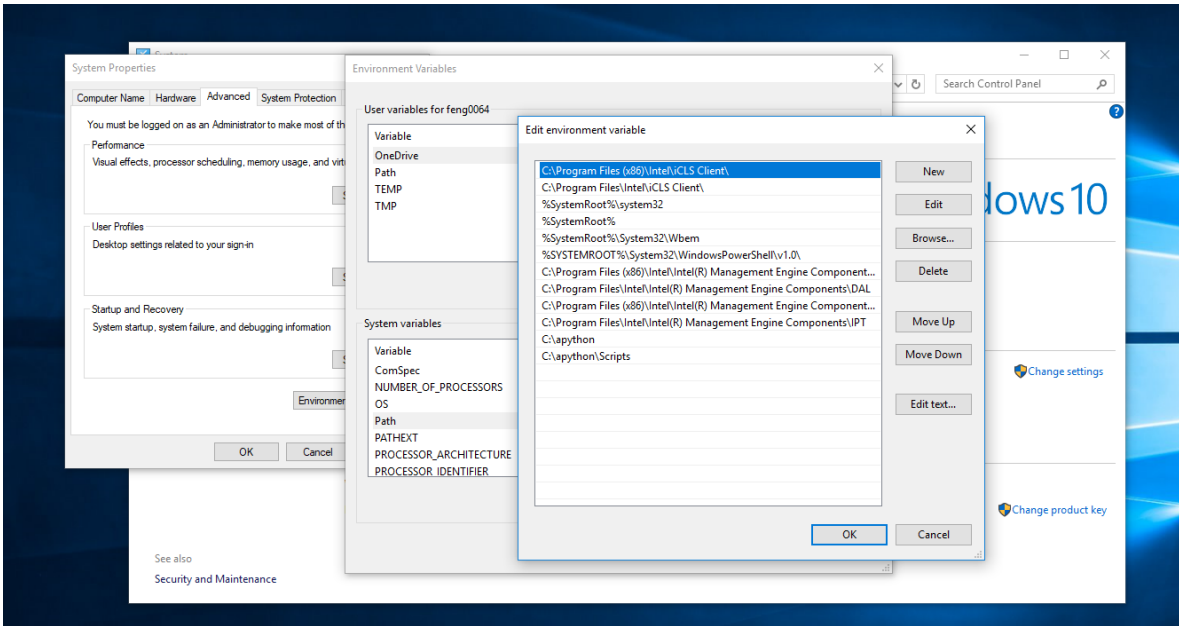
Proceed ([y]/n)? y

Downloading and Extracting Packages
numpy 1.13.3: ##### | 100%
Preparing transaction: done
Verifying transaction: done
Executing transaction: done

```



or download Miniconda from <https://conda.io/miniconda.html>, which includes only conda and its dependencies. Once Miniconda is installed, you can use the command to install any other packages. For example, If you want to access Python via other command line prompt like cmd.exe, you will need to add the Python directory to the "Path" system environment variable. In windows, go to **Control Panel** → **System Properties** → **Environment Variables**, select the **Path**, click **Edit** and append both the Python path and the Scripts path to the end of the string. With that in place, you can start the Python interpreter on any command prompt by invoking the Python command.



3 Accessing SEC Filing through EDGAR

3.1 Overview

The full SEC filings can be accessed on the EDGAR (Electronic Data Gathering, Analysis, and Retrieval system). Companies began filing through EDGAR 1994-95 with various phase-in periods for different form types.

The SEC filings are disseminated to the public through the EDGAR Public Dissemination Service (PDS) System, which is managed by Attain, LLC and available to paid subscribers. The SEC public filings are also available free of charge in a browsable/searchable format on the SEC's website at <https://www.sec.gov/cgi-bin/browse-edgar?action=getcurrent> and can be searched through <https://www.sec.gov/edgar/searchedgar/companysearch.html>.

3.2 Using the EDGAR Index Files

Indexes to all public filings are available from 1994Q3 through the present and located in the following browsable directories:

- <http://www.sec.gov/Archives/edgar/daily-index/>—daily index files through the current year;
- <https://www.sec.gov/Archives/edgar/full-index/>—Full indexes offer a "bridge" between quarterly and daily indexes, compiling filings from the beginning of the

current quarter through the previous business day. At the end of the quarter, the full index is rolled into a static quarterly index.

Each directory and all child subdirectories contain 3 files to assist in automated crawling of these directories. (Note that these are not visible through directory browsing.)

- index.html (the web browser would normally receive these)
- index.xml (an XML structured version of the same content)
- index.json (a JSON structured vision of the same content)

The EDGAR indexes list the following information for each filing: Company Name, Form Type, CIK (Central Index Key), Date Filed, and File Name (including folder path).

Four types of indexes are available. The company, form, and master indexes contain the same information sorted differently.

- company–sorted by company name
- form–sorted by form type
- master–sorted by CIK number
- XBRL–list of submissions containing XBRL financial files, sorted by CIK number; these include Voluntary Filer Program submissions

Once we understand the filings and compile the index, we can download the data accordingly.

After compiling the full index of all the files, we proceed as follows:

1. Subset the index file based on the time period (year 2011) and file type (13D). This step is done using SAS, and create the file list we want to download;
2. Using python (urllib package) to download the files from the list. It is helpful to customize the name of the downloaded, as we will need to extract the date information later. The specific naming rule in this project is 13D+CIK+_Date

The exact code for implementation is :

```
import pandas as pd
tempdata = pd.DataFrame.from_csv('list_13d.csv', header=0, index_col=None)
```

```
import urllib.request

for index,row in tempdata.iterrows():
    url = 'https://www.sec.gov/Archives/' +row['file']+'.txt'
    name='13D'+str(row['CIK'])+'_'+str(row['DATE_FILED'])
    con = urllib.request.urlopen(url)
    with open(os.path.basename(name),"wb") as f:
        f.write(con.read())
```

3.3 Example: Shareholder Activism

Schedule 13D filing is the most important source to collect shareholder activism events. Under Section 13(d) of the 1934 Exchange Act, the investors with interest in influencing the management of the company are mandatory to file with the SEC within 10 days of acquiring more than 5% of any class of securities of a publicly traded company. The 13D filings could inform the market that the filers may force changes or seek control at the target firms.

Python is extremely powerful for analyzing these data for two reasons. First, Python is a phenomenally good tool for text analysis. With Python one could extract various of useful information from 13D filing for her specific research objectives. An example is the item 4 in 13D filings, which discloses the reasons for the filers to acquire the shares, and what approaches the filers apply to achieve their objectives, for instance, if the filers have communicated with the management or filed a proxy statement. One could examine how outcomes of activism vary across different objectives and tactics. Second, the machine learning function of Python can be applied to analyze how activists choose their targets, and the relationship between activism and firm outcomes.

In this project we conduct two exercises. First, we download the 13D filing data from EDGAR database, and use Python to extract the CUSIP of the target firms, which is the identifier for merging with other datasets. Next, we apply machine learning techniques to predict whether a firm is the target of activist hedge funds based on a sample in 2011. Again, we emphasize more on the implementation of method, not the question itself.

4 Web scraping

4.1 Description

The general idea is to extract the text in the tables from each files, and then identify the 9-digit CUSIP that are adjacent to the string “CUSIP”

This is the core step to extract the target company CUSIP from 13D filings. The general steps can be summarized as such:

1. Use Beautiful Soup to read in the filings and extract the information
2. Use Regular Expression to pin down the 9-digit CUSIP

Beautiful Soup

Beautiful soup is a powerful tool to processing web scraping tasks. In this project, the filings are stored under html files, so we need to make the program understand the structure (indeed, the files look more friendly to computers than human).

Specifically, we first identify that the CUSIP numbers are stored in some of the “tables” in the filings. Using Beautiful soup, the program is able to go through all the tables in all the files, and extract the texts and data from the tables. By the end of the step, we have a list of all the information in the tables.

Below is an example of the information we are trying to extract The documentation of Beautiful Soup can be found at <https://www.crummy.com/software/BeautifulSoup/bs4/doc/>

Here are the exact steps for the processing:

1. An example of 13D filing that we can extract CUSIP is as below. This is what we see on the webpage that is readable. The task is to extract the nine digit CUSIP

CUSIP No. 095180105		13D	Page 2
1.	NAME OF REPORTING PERSON: Stockbridge Fund, L.P.		
2.	CHECK THE APPROPRIATE BOX IF A MEMBER OF A GROUP:		(a) <input type="checkbox"/> (b) <input checked="" type="checkbox"/>
3.	SEC USE ONLY		
4.	SOURCE OF FUNDS:	OO	
5.	CHECK BOX IF DISCLOSURE OF LEGAL PROCEEDINGS IS REQUIRED PURSUANT TO ITEM 2(d) OR 2(e):		<input type="checkbox"/>
6.	CITIZENSHIP OR PLACE OF ORGANIZATION:	Delaware	
NUMBER OF SHARES BENEFICIALLY OWNED BY EACH REPORTING PERSON WITH	7.	SOLE VOTING POWER:	1,517,536 (see Item 5)†
	8.	SHARED VOTING POWER:	0
	9.	SOLE DISPOSITIVE POWER:	1,517,536 (see Item 5)†
	10.	SHARED DISPOSITIVE POWER:	0
11.	AGGREGATE AMOUNT BENEFICIALLY OWNED BY EACH REPORTING PERSON:		1,517,536 (see Item 5)†
12.	CHECK BOX IF THE AGGREGATE AMOUNT IN ROW (11) EXCLUDES CERTAIN SHARES:		<input type="checkbox"/>
13.	PERCENT OF CLASS REPRESENTED BY AMOUNT IN ROW (11):		3.1% (see Item 5)*
14.	TYPE OF REPORTING PERSON:		PN

2. The html file corresponding is as such:

```
<table cellpadding="0" cellspacing="0" width="100%" style="FONT-FAMILY: times new roman;
FONT-SIZE: 10pt; FONT-SIZE: 10pt; FONT-FAMILY: times new roman">
<tr>
<td align="left" valign="top" width="43%" style="BORDER-BOTTOM: black 2px solid; BORDER-
LEFT: black 2px solid; PADDING-LEFT: 0pt; MARGIN-LEFT: 9pt; BORDER-TOP: black 2px solid;
BORDER-RIGHT: black 2px solid">
<div style="LINE-HEIGHT: 10.25pt; TEXT-INDENT: 0pt; DISPLAY: block; MARGIN-LEFT: 9pt;
MARGIN-RIGHT: 0pt" align="left"><font style="DISPLAY: inline; FONT-FAMILY: times new roman;
FONT-SIZE: 12pt">CUSIP No.<font style="DISPLAY: inline; FONT-SIZE: 12pt">&#160;
</font>095180105</font></div>
</td>
<td valign="top" width="8%" style="TEXT-ALIGN: center; TEXT-INDENT: 0pt; MARGIN-LEFT: 0pt;
MARGIN-RIGHT: 0pt">
<div style="TEXT-ALIGN: center; LINE-HEIGHT: 10.25pt; TEXT-INDENT: 0pt; DISPLAY: block;
MARGIN-LEFT: 0pt; MARGIN-RIGHT: 0pt"><font style="DISPLAY: inline; FONT-FAMILY: times new
roman; FONT-SIZE: 12pt">13D</font></div>
</td>
<td valign="top" width="34%" style="BORDER-BOTTOM: black 2px solid; BORDER-LEFT: black 2px
solid; BORDER-TOP: black 2px solid; BORDER-RIGHT: black 2px solid">
<div style="LINE-HEIGHT: 10.25pt; TEXT-INDENT: 0pt; DISPLAY: block; MARGIN-LEFT: 0pt;
MARGIN-RIGHT: 0pt" align="center"><font style="DISPLAY: inline; FONT-FAMILY: times new
roman; FONT-SIZE: 12pt">Page 2</font></div>
</td>
</tr></table>
</div>
</div>
</div>
```

3. For this specific table, if we go through all the tags and extract the text and data. Specifically, we repeat through two tags (1) tr, which defines a row and (2) td, which defines a cell

```

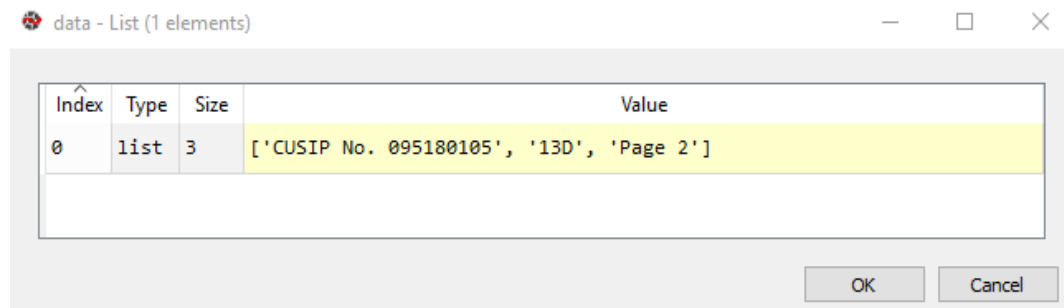
x2=BeautifulSoup(x1,"lxml")

tables = x2.find_all('table')
record = []
# loop through the tables to scrape the text and numbers
# from the 13D filing
for table in tables:
    data = []

    # each table will be examined
    rows = table.find_all('tr')
    for row in rows:
        cols = row.find_all('td')
        cols = [ele.text.strip() for ele in cols]
        data.append([ele for ele in cols if ele])

```

The above code will generate an object “data”:



This step is repeated for each table under the same file, and we record the information. From this step, we can proceed to identify the nine digits CUSIP from the list of strings.

Regular Expression

The task is to extract the nine digit number from “data”. We rely on the regular expression to derive the pattern.

A regular expression is a sequence of characters that define a search pattern. It is very useful for searching a specific pattern in a long string. Some of the typical expressions can be found [here](#)

Regular expression cheatsheet	
Special characters	
<code>\</code>	escape special characters
<code>.</code>	matches any character
<code>^</code>	matches beginning of string
<code>\$</code>	matches end of string
<code>[sb-d]</code>	matches any chars 's', 'b', 'c' or 'd'
<code>[^a-c6]</code>	matches any char except 'a', 'b', 'c' or '6'
<code>R S</code>	matches either regex <code>R</code> or regex <code>S</code>
<code>()</code>	creates a capture group and indicates precedence
Quantifiers	
<code>*</code>	0 or more (append <code>?</code> for non-greedy)
<code>+</code>	1 or more (append <code>?</code> for non-greedy)
<code>?</code>	0 or 1 (append <code>?</code> for non-greedy)
<code>{m}</code>	exactly <code>m</code> occurrences
<code>{m, n}</code>	from <code>m</code> to <code>n</code> . <code>m</code> defaults to 0, <code>n</code> to infinity
<code>{m, n}?</code>	from <code>m</code> to <code>n</code> , as few as possible
Special sequences	
<code>\A</code>	start of string
<code>\b</code>	matches empty string at word boundary (between <code>\w</code> and <code>\w</code>)
<code>\B</code>	matches empty string not at word boundary
<code>\d</code>	digit
<code>\D</code>	non-digit
<code>\s</code>	whitespace: <code>[\t\n\r\f\v]</code>
<code>\S</code>	non-whitespace
<code>\w</code>	alphanumeric: <code>[0-9a-zA-Z_]</code>
<code>\W</code>	non-alphanumeric
<code>\Z</code>	end of string
<code>\g<id></code>	matches a previously defined group
Special sequences	
<code>(?iLmsux)</code>	matches empty string, sets re.X flags
<code>(?:...)</code>	non-capturing version of regular parentheses
<code>(?P...)</code>	matches whatever matched previously named group
<code>(?P=)</code>	digit
<code>(?#...)</code>	a comment; ignored
<code>(?=...)</code>	lookahead assertion: matches without consuming
<code>(?!...)</code>	negative lookahead assertion
<code>(<=...)</code>	lookbehind assertion: matches if preceded
<code>(<!=...)</code>	negative lookbehind assertion
<code>(? (id)yes no)</code>	match 'yes' if group 'id' matched, else 'no'

Based on tartley's [python-regex-cheatsheet](#).

For our purposes, the exact code to extract the cusip from data is:

```
y = [str(x) for x in data]
ytest1 = "".join(y)
# remove special symbols
ytest2 = re.sub('[^0-9a-zA-Z]+', '', ytest1)
# remove the space (xa0) and text 13D
yytest = re.sub('xa0|13D', '', ytest2)

if 'CUSIP' in yytest:
    # print(yytest)
    # keep only the digits and cusip
    test_re = re.findall(r'\d+|CUSIP', yytest)
```

The result “test_re” looks like:

test_re - List (3 elements)

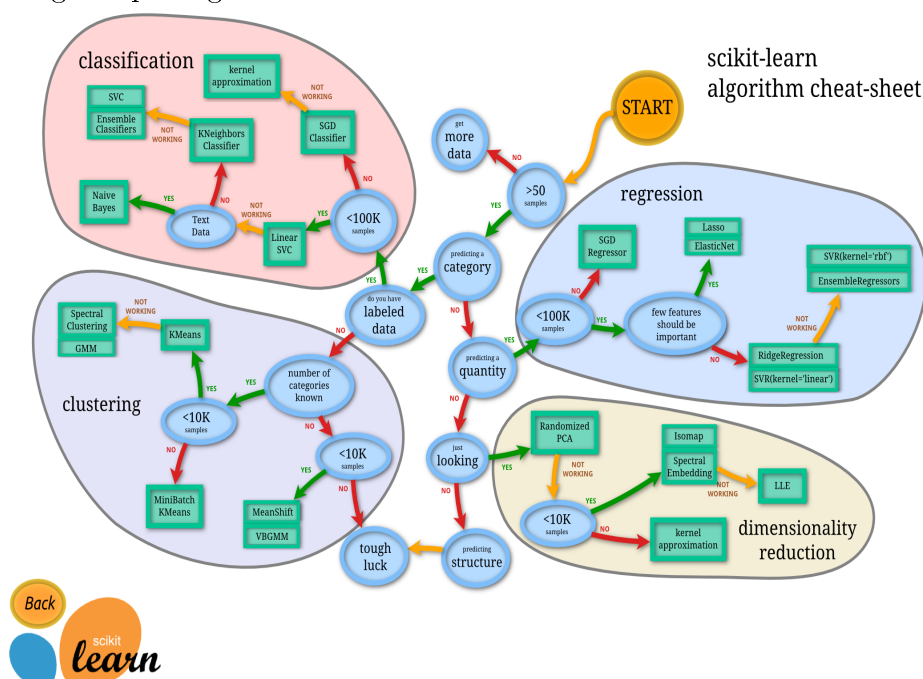
Index	Type	Size	Value
0	str	1	CUSIP
1	str	1	095180105
2	str	1	2

OK Cancel

Then we get the exact 9-digit CUSIP from the filing.

5 Scikit-Learn

Scikit-Learn is a machine learning library for python. You can easily use the package to implement machine learning analysis. Various machine learning algorithms including supervised learning, unsupervised learning, model selection and evaluation can be done using the package.

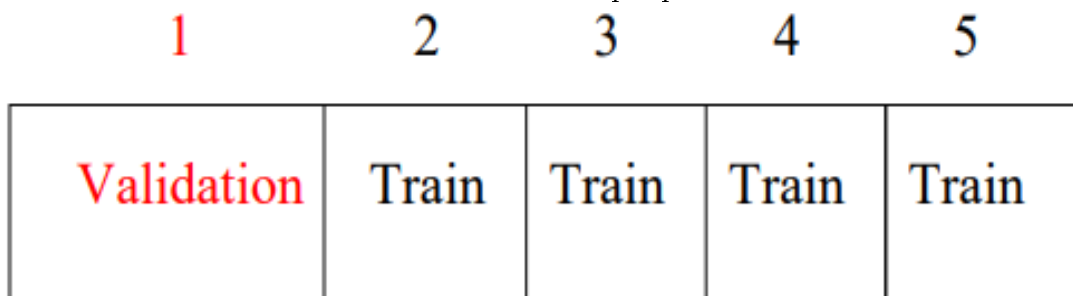


5.1 Cross Validation

Machine learning algorithms are complicated methods, you can add different layers of non-linear functions to try to fit one particular dataset. This leads to an easily over-

fitting problem (just thinking that even a polynomial can approximate any function without incurring too much trouble). We do not want our model, our "estimated coefficients" to just be able to explain the characteristics of that particular dataset. Hence we need measure the model's out of sample prediction power.

Take the K-Folder Cross-validation we are using in our project as an example. We divide the sample into K equal part, and estimate the model using K -1 parts together as training sample, and the left one as test sample. So can then have K different model estimates and can conduct K times out of sample prediction validation.



Python Implementation of Cross Validation

```
kf = KFold(n_splits = 5)
scores = pd.DataFrame(columns=['MLP', 'GB'])
for train_indices, test_indices in kf.split(x):
    # print("Multi-layer perception Score")
    clf.fit(x[train_indices], y[train_indices])
    clf_score = clf.score(x[test_indices], y[test_indices])
    # print(clf.score(x[test_indices], y[test_indices]))
```

This is the example of our code implementing K-folder cross validation. The main idea is quite simple, you can think of it as a random number generator.

- First you define a k-folder object, and set number of folders as 5 (`n_splits = 5`)
- Second, using `kf.split(x)` to generate both `train_indices` and `test_indices` for input data X.
- Use `train_indices` to do train the sample, and use `test_indices` to test the out of sample goodness of fit.

- Collect K different out of sample goodness of fit results, and compare between models.

```
class sklearn.model_selection. KFold (n_splits=3, shuffle=False, random_state=None)
```

[\[source\]](#)

K-Folds cross-validator

Provides train/test indices to split data in train/test sets. Split dataset into k consecutive folds (without shuffling by default).

Each fold is then used once as a validation while the k - 1 remaining folds form the training set.

Read more in the [User Guide](#).

Parameters: **n_splits** : int, default=3

Number of folds. Must be at least 2.

shuffle : boolean, optional

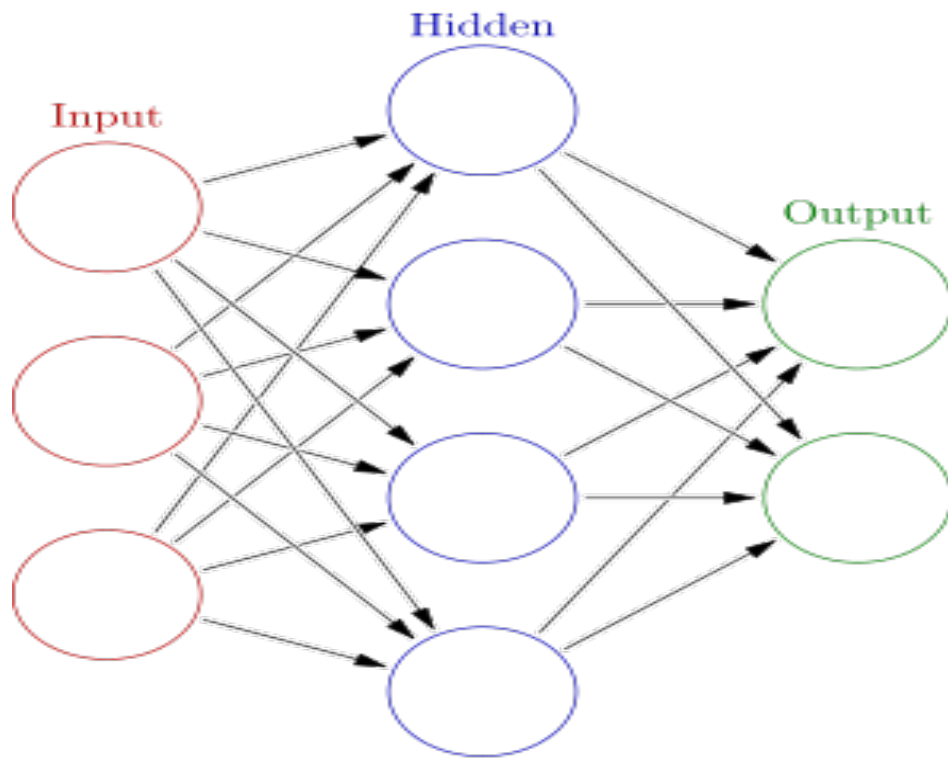
Whether to shuffle the data before splitting into batches.

random_state : int, RandomState instance or None, optional, default=None

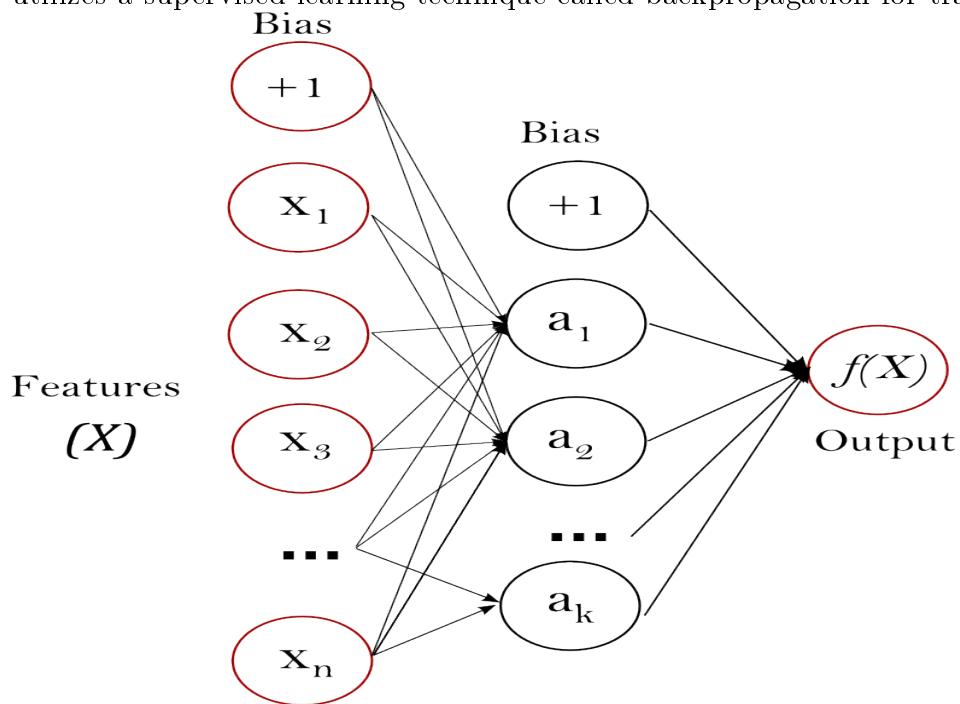
If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random. Used when `shuffle == True`.

5.2 Supervised Neural Network

Neural network models are machine learning models inspired by the biological neural networks that constitute animal brains. An artificial neural network is an interconnected group of nodes, akin to the vast network of neurons in a brain. Here, each circular node represents an artificial neuron and an arrow represents a connection from the output of one artificial neuron to the input of another. Researchers can choose different number of layers accordingly based on different data analysis tasks.



Multi-layer Perceptron (MLP) is one particular supervised learning neural network model, It can learn non-linear approximator for either classification or regression. MLP utilizes a supervised learning technique called backpropagation for training.



Python Implementation of MLP Classification

```
from sklearn.neural_network import MLPClassifier

clf = MLPClassifier(solver='lbfgs', alpha=1e-5, hidden_layer_sizes=(5, 2),
                    random_state=1)

clf.fit(x[train_indices], y[train_indices])

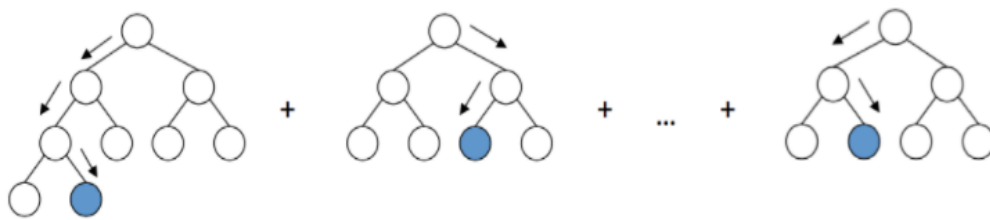
clf_score = clf.score(x[test_indices], y[test_indices])
```

- First import MLPClassifier from scikit learn
- Second, set parameters for a MLPClassifier object. “lbfgs” is an optimizer in the family of quasi-Newton methods for the weight optimization. Here we set 5 hidden layer and 2 hidden units. Random_state is the seed used by the random number generator
- Use X[train_indices] and y[train_indices] to fit the model.
- Calculate goodness of fit in test sample.

5.3 Gradient Boosting

Gradient boosting is a machine learning technique for regression and classification problems, which produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees. Ensemble methods use multiple learning algorithms to obtain better predictive performance than could be obtained from any of the constituent learning algorithms alone Gradient boosting combines weak “learners” into a single strong learner in an iterative fashion.

The general idea is to minimize some penalty function (e.g. mean squared error) from some prediction. The intuition behind gradient boosting algorithm is to repetitively leverage the patterns in residuals and strengthen a model with weak predictions and make it better.



<https://medium.com/mlreview/gradient-boosting-from-scratch-1e317ae4587d> provides an intuitive introduction to gradient boosting

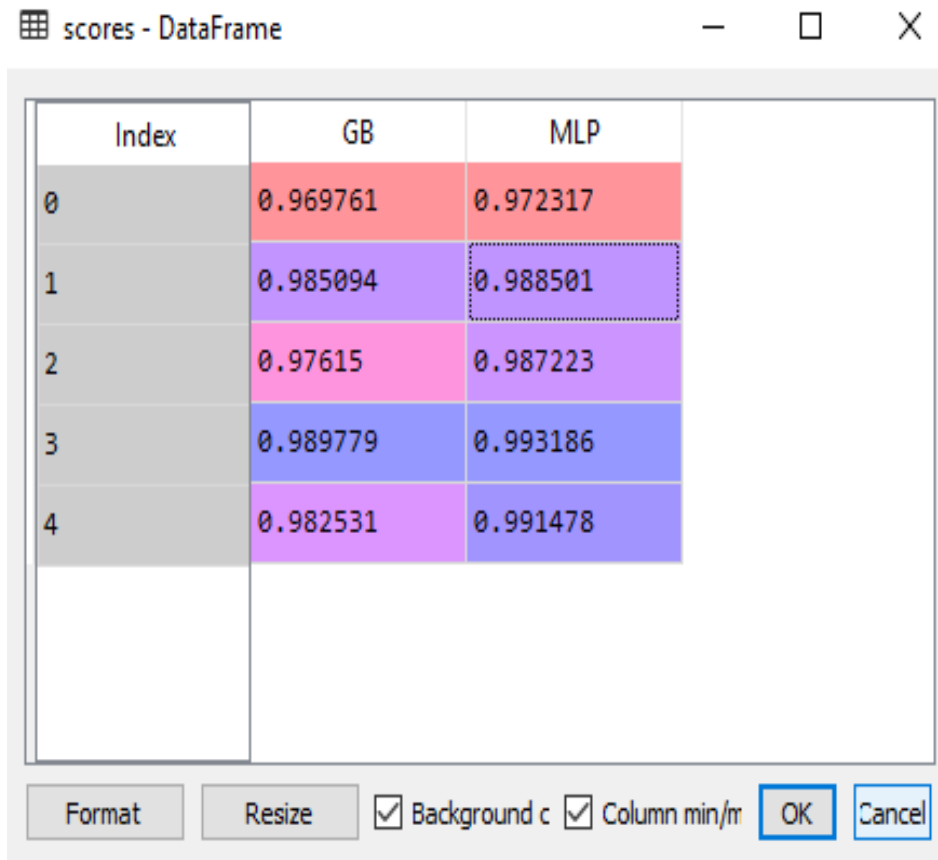
Python Implementation of Gradient Boosting Classifier

```
from sklearn.ensemble import GradientBoostingClassifier

gbdt = GradientBoostingClassifier(
    n_estimators=100,max_depth=3,random_state=1)
gbdt.fit(x[train_indices], y[train_indices])
gb_score = gbdt.score(x[test_indices], y[test_indices])
```

- First import GradientBoostingClassifier from scikit learn ensemble
- Second, set parameters for a GradientBoostingClassifier object. “n_estimators” is the number of boosting stages to perform. “max_depth” is the maximum depth of the individual regression estimators. The maximum depth limits the number of nodes in the tree. “Random_state” is the seed used by the random number generator
- Use X[train_indices] and y[train_indices] to fit the model.
- Calculate goodness of fit in test sample.

The results of the MLP and GB is:



Index	GB	MLP
0	0.969761	0.972317
1	0.985094	0.988501
2	0.97615	0.987223
3	0.989779	0.993186
4	0.982531	0.991478

Format Resize ☒ Background c ☒ Column min/m OK Cancel